UG235.06: Bootloading and OTA with Silicon Labs Connect v2.x

This chapter of the *Connect v2.x User's Guide* explains the bootloader options (standalone, application, and Over the Air (OTA)) available for use within Connect-based applications. The *Connect v2.x User's Guide* assumes that you have already installed the Simplicity Studio development environment and the Flex SDK, and that you are familiar with the basics of configuring, compiling, and flashing Connect-based applications. Refer to *UG235.01: Developing Code with Silicon Labs Connect v2.x* for an overview of the chapters in the *Connect v2.x User's Guide*.

KEY POINTS

- Introduces different bootloaders.
- Describes using the Gecko Bootloader.
- Describes how to set up a Silicon Labs Connect application for bootloading.
- Describes how to use the standalone bootloader with an SOC or NCP Host application.
- Describes how to use the application bootloader for OTA.

The *Connect v2.x User's Guide* is a series of documents that provides in-depth information for developers who are using the Silicon Labs Connect Stack for their application development. If you are new to Connect and the Proprietary Flex SDK, see *QSG138: Proprietary Flex SDK v2.x Quick Start Guide*.

Proprietary is supported on all EFR32FG devices. For others, check the device's data sheet under Ordering Information > Protocol Stack to see if Proprietary is supported. In Proprietary SDK version 2.7.n, Connect is not supported on EFR32xG22.

1. Introduction

It is often required to update firmware on devices when it is not feasible to connect a J-Link debugger. In these cases, a standalone bootloader is ideal because it makes it possible to update the firmware through a Universal Asynchronous Receiver/Transmitter (UART) or Serial Peripheral Interface (SPI) connection.

In some cases, even connecting a device to a computer is problematic. In these instances, an OTA (Over the Air) bootloading process can work. This requires an application bootloader which can update the firmware from an onboard storage (like an SPI flash memory or part of the MCU flash memory). However, the OTA image transfer is still the responsibility of the main application.

For more details on bootloading basics, see *UG103.06: Bootloading Fundamentals*.

2. The Gecko Bootloader

Silicon Labs Connect only supports the Gecko Bootloader which is available for the Wireless Gecko (EFR32™) portfolio. For the stand-alone bootloader, the *UART XMODEM Bootloader* example is recommended. For the application bootloader, either the *SPI Flash Storage Bootloader* (if SPI flash is attached the EFR32) or the *Internal Storage Bootloader* applications are recommended. Note that the OTA protocol available for Silicon Labs Connect only supports **single image** bootloaders.

Once you have compiled the bootloader, make sure to flash it (the `<projectname\>-combined.s37` file) on your device before flashing the main firmware.

For more details on using the Gecko Bootloader examples, see *AN1085: Using the Gecko Bootloader with Silicon Labs Connect*. For more information on the Gecko Bootloader, see *UG266: Silicon Labs Gecko Bootloader User's Guide*.

3. Setting Up a Silicon Labs Connect Application for Bootloading

To make an application ready for bootloader usage, select either the Standalone or the Application bootloader on the HAL tab of Application Builder. This will:

- Modify the linker script on EFR32xG1 (where the first 16kB of the flash memory is used by the bootloader).
- Modify the linker script to reserve the first 4B of RAM for communication between the bootloader and the main firmware.
- Add GBL file (binary image for bootloading) generation to the build process.
- Add a compile time define that can be used by bootloader-related code.
- Add c functions that can be used to communicate with the bootloader from the main firmware (for example, to switch to bootloader mode).
- Add the application properties data that can be read by the bootloader from a GBL image.

Some specific application features like OTA needs some plugins as well. These requirements will be discussed in the following chapters.

4. Using the Standalone Bootloader with a System on Chip Application

The standalone bootloader can be used without further modification in the application code. If you enable the GPIO activation feature in the Gecko Bootloader (enabled by default), you can enter bootloader mode by resetting the MCU while pushing the activation button. Then, you can communicate with the bootloader as described in *AN1085: Using the Gecko Bootloader with Silicon Labs Connect*.

Alternatively, you can enter bootloader mode by using the following function in your application:

```
halLaunchStandaloneBootloader(STANDALONE_BOOTLOADER_NORMAL_MODE)
```

For further details on the communication API with the bootloader, see the file

```
platform/base/hal/micro/bootloader-interface-standalone.h
```

5. Using the Standalone Bootloader with an NCP-Host Application

The standalone bootloader can be used in NCP-host mode to update the firmware on the NCP device from the host device. To use this, make sure standalone bootloader on your NCP device and the Bootloader NCP CLI plugin in your host application are enabled (both are enabled by default). This will add a three CLI commands to the NCP-host:

- `bootloader info` – Returns information of the bootloader.
- `bootloader launch` – Starts the bootloader.
- `bootloader load-image` – Loads a firmware image using an external tool.

To use the third command:

1. Compile the host application on the host (currently only POSIX operating systems like Linux are supported). Go to the folder named protocol/flex/ and call `make -f connect/plugins/serial-bootloader/bootload-ncp-uart-app.mak`

   This will generate the binary *bootloader-ncp-uart-app* under

   `protocol/flex/connect/build/bootloader-ncp-uart-app/`
2. Once you have this binary and the firmware image on your host machine, you can load it using the following commands (note the use of double quotes around the string parameters):

   `bootloader launch`

   then

   `bootloader load-image "<path to loader>" "<path to the firmware> <offset> <size>" [other bootloader options]`
   - `<path to loader>` is the path to *bootloader-ncp-uart-app* compiled in step 1.
   - `<path to the image>` is the path to the firmware image to load.
   - `<offset>` is the offset inside the image where loading should start (typically 0).
   - `<size>` is the size of the image or `0xFFFFFFFF` to use the whole file.
   - `[other bootloader options]` is where further options can be passed to the *bootloader-ncp-uart-app*.

     Note: option "`-p <serial port>`" is mandatory. See `bootloader-ncp-uart-app -h` for further details.

6. Using the Application Bootloader for OTA

6.1 Broadcast or Unicast

Silicon Labs Connect provides two OTA methods:

- Broadcast: Image is sent to all devices (although a device could ignore it, so in that sense, it is actually multicast). Only single-hop range is supported—that is, the coordinator cannot bootload endpoints behind range extender.
- Unicast: Image is sent to a single device. If the server and client are in the same Connect network, it will work. Packets are ACKed (both per-hop and end-to-end). The drawback is that it can only target a single device. Therefore, bootloading multiple targets is slow and puts a serious load on the network.

Both broadcast and unicast use a single source to distribute the image. This is called the OTA server. The devices that download the image are called OTA clients. Both broadcast and unicast supports secure messages.

Neither OTA method is available currently for MAC mode. Sleepy end devices cannot be bootloaded using the broadcast method. Although unicast bootloading of a sleepy end device is theoretically possible, it would take an extremely long time and Silicon Labs does not recommend it.

6.2 OTA Plugins

The following plugins are required to test OTA in an SoC application.

For both broadcast and unicast OTA:

- Bootloader interface: enables the application to communicate with the bootloader (for example, the flash memory is written using bootloader APIs).
- OTA Bootloader test common: implements generic bootloader CLI commands such as erase.

For broadcast OTA:

- OTA Broadcast Bootloader client: implements OTA image reception.
- OTA Broadcast Bootloader server: implements OTA image transmission.
- OTA Broadcast Bootloader test: Implements CLI over the bootloader client/server and connects the OTA plugins with the bootloader interface.

For unicast OTA:

- OTA Unicast Bootloader client: implements OTA image reception.
- OTA Unicast Bootloader server: implements OTA image transmission.
- OTA Unicast Bootloader test: Implements CLI over the bootloader client/server and connects the OTA plugins with the bootloader interface.

All plugins are open source.

6.3 Theory of Operation: Broadcast OTA

The broadcast OTA completes these steps:

1. Clients who want to download the same image set up the same tag.
2. The server starts broadcasting tagged image segments (a chunk of the firmware that fits into a connect data frame with memory address).
3. The clients store the broadcasted image segments. Based on the addresses, they also recognize missing segments.
4. The server stops the broadcast after 512 segments and asks each client (with unicast message) what segments are missing.
5. The clients report the missing segments and the server collects this information.
6. The server re-broadcasts the missing segments.
7. The server asks again for missing segments and this loop continues until all the clients have the first 512 segments.
8. The server broadcasts the next 512 segments and this loop continues until all the clients have the full image.

The tags can be used in a network where not all devices run the same firmware, or it can be used for versioning to make sure a device will not participate in the OTA process if it already has the same image.

6.4 Theory of Operation: Unicast OTA

Unicast OTA is much simpler. It completes these steps:

1. The clients set up a tag.
2. The server sends a handshake, telling the client the image size and the tag.
3. The client responds to the handshake.
4. The server sends a tagged image segment (a chunk of the firmware that fits into a connect data frame with memory address).
5. The client responds with a segment response.
6. If the server did not receive a response, it sends the segment again.
7. If the server did receive the response, it sends the next segment, and continues this loop until the client has the whole image.

The tag is still available. It can be used for versioning. The server retries to transmit the unacknowledged segment multiple times before it stops the transmission. The client has a five second timeout before it drops the current firmware download. A resume function has been implemented in Flex 2.6. The client counts how many segments were received. If the connection was lost between the server and the client, once the server re-initiates the download, the client informs the server to send only the remaining portion during the handshake. The segment counter on the client side is cleared if the tag changed or a flash erase command has been executed by the client. If the communication was interrupted, the re-initiation of the download is the responsibility of the customer's application on the server side.

6.5 Remote Bootload Request

Both broadcast and unicast OTA plugins implement command messages which makes remote bootload requests possible—that is, the OTA server can request bootload from clients, and in response, the clients will load the downloaded image into the main memory and boot into that.

7. Example: OTA Bootloading Sensor(s) in a Sensor-sink Demo

The following example walks through using OTA in a sensor-sink demo using CLI commands. The sink will be the OTA server and the sensor(s) will be the OTA client(s). The example will use sensor/sink if the context is the Connect network and OTA client/server if the context is the OTA process.

The example should work with any Software Development Kit that is supported by Silicon Labs Connect, but the internal flash bootloader example is only available for 1MB (EFR32xG12) and 512kB (EFR32xG13) devices.

7.1 Create Projects

1. Create either of the single image bootloader examples that are available for your devices. There is no need to modify it.
2. Create a connect sink application. Enable Application Bootloader on the HAL tab of AppBuilder and enable the following plugins:
   - Bootloader interface
   - OTA Bootloader test common
   - OTA Broadcast or Unicast Bootloader server
   - OTA Broadcast or Unicast Bootloader test

   These will be the OTA client projects.
3. Create two sensor applications. The example will flash one with regular J-Link and use OTA to bootload the other. Make sure you make some difference between the two applications—for example, change the heartbeat period to 12 (3 seconds) or change the name on the General tab. On both projects, enable Application Bootloader on the HAL tab and enable the following plugins:
   - Bootloader interface
   - OTA Bootloader test common
   - OTA Broadcast or Unicast Bootloader client
   - OTA Broadcast or Unicast Bootloader test

   These will be the OTA client projects.
4. Generate and build all four projects (bootloader, sink, two sensors).

7.2 Flashing Applications

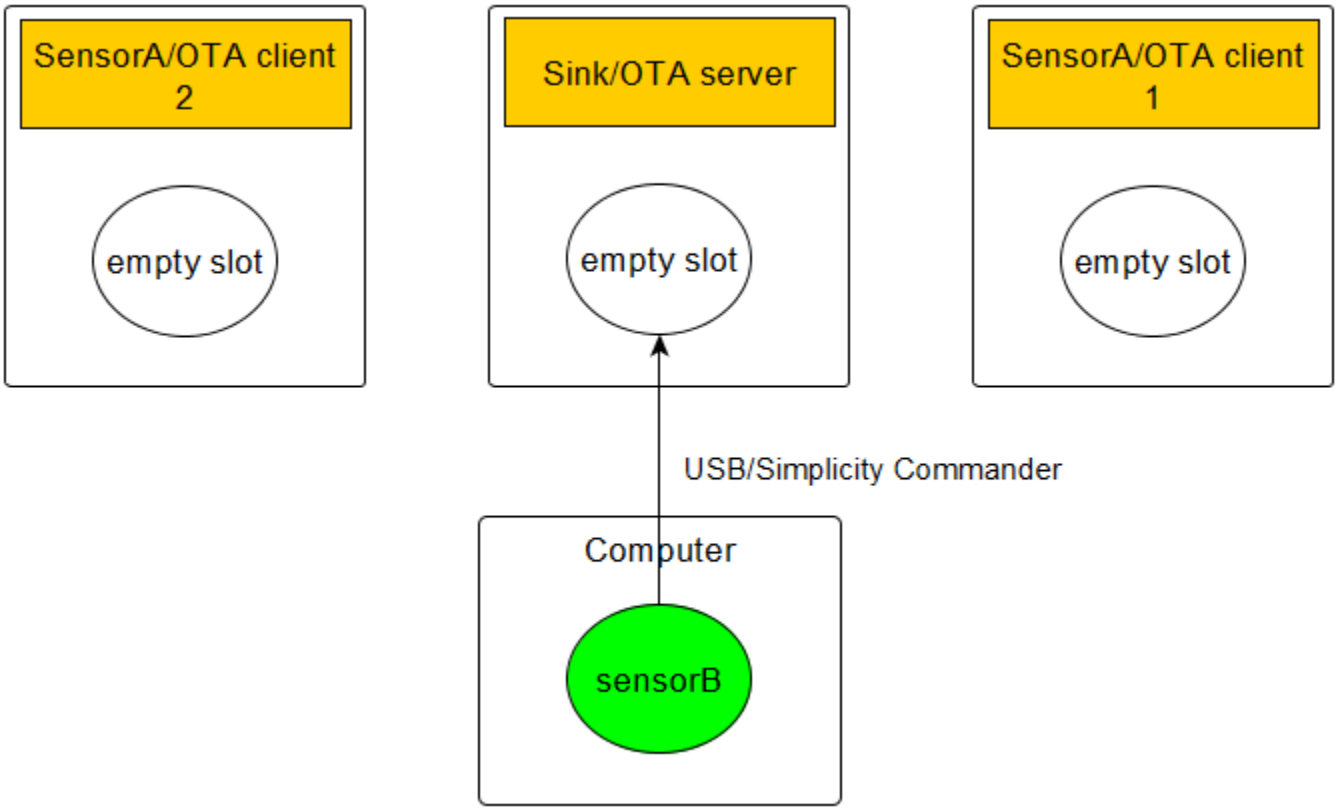
For the applications to work, you first need to flash the bootloader itself. The simplest way to flash it is to use Simplicity Commander. Make sure you use the `<bootloader app name>-combined.s37` file because this is the only image with the full bootloader: Gecko Bootloader is a two-stage bootloader, which means there is a very small first stage which can update the bootloader itself (this is not demonstrated in this example). Only the `-combined.s37` file includes this first stage. All other binaries are just the second stage so they will not work by themselves.

Next, flash the sink and one of the sensor applications. You can use almost any method, just make sure you don't use the "`bin` file because it does not include address information and might overwrite the bootloader. Also, make sure you **do not erase** the flash memory before flashing the application.

If both the bootloader and the application are present, it should start as a usual Connect project (heartbeat LED blinks, CLI available).

7.3 Loading the Image to the OTA Server

Next, you need to flash the GBL file of the other sensor to the bootloader's storage slot on the sink/OTA server as shown in the following figure.



The easiest way to do that is to use Simplicity Commander's command line options.

7.3.1 Loading to SPI Flash

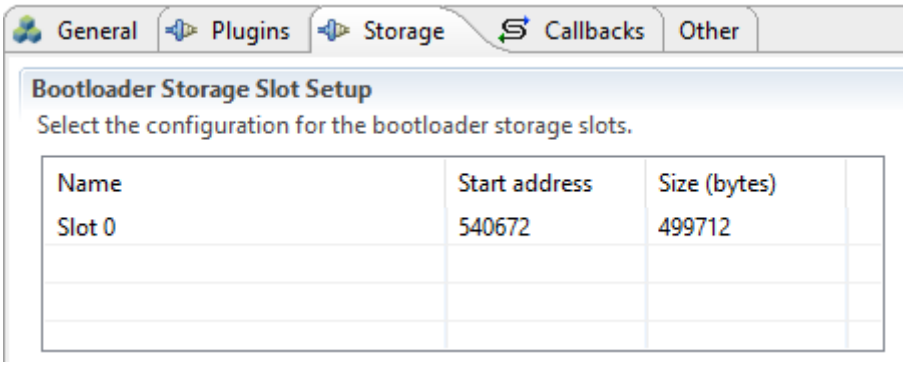
To load <application image>.gbl to the SPI flash, use the following command:

```
commander extflash write <application image>.gbl
```

This method is only available on Silicon Labs Development Kits. It will lock the MCU so reset the device when it is finished.

7.3.2 Loading Internal Flash

To flash to internal storage, first figure out the starting address of the slot. You can read that from the storage tab in the AppBuilder of the bootloader project. For example, for 1MB devices, it is 540672 by default as shown in the following figure.



Simplicity Commander will recognize the GBL file and you do not want to decode the address information from there (as it would flash it to address 0): Basically, you want it to handle as a binary blob. If you rename the GBL file to .bin, you get exactly that. After that, you can flash it either from the GUI (with start address) or from the command line by executing the following command:

```
commander flash --address 84000 <application image>.bin
```

Where `84000` is 540642 in hexadecimal and <application `image>.bin` is the renamed GBL file.

For more information, see *UG162: Simplicity Commander Reference Guide*.

7.3.3 Preparing the Devices for Bootloading

The next steps are CLI commands on the devices.

7.3.3.1 Preparing the Storage

Use these commands to prepare the storage on the device.

- 7.3.3.1.1 Sink/OTA Server
- 7.3.3.1.1 Sink/OTA Server

7.3.3.1.1 Sink/OTA Server

```
bootloader init
bootloader validate-image
```

The second command will check the image, which should look like this (with many more dots):

```
image....done!
Image is valid!
```

7.3.3.1.2 Sensor/OTA Client

```
bootloader init
bootloader flash-erase
```

The second command will erase the flash, which is required before writing it. Its output should look like this (with many more dots):

```
flash erase started
flash erasing slot 0 started
................................
flash erase successful!
```

7.3.3.2 Preparing the Network

These commands are the usual ones to create and join to a network.

- 7.3.3.2.1 Sink/OTAServer
- 7.3.3.2.2 Sensor/OTA Client

7.3.3.2.1 Sink/OTAServer

```
form 0
pjoin 120
```

7.3.3.2.2 Sensor/OTA Client

```
join 0
```

It is a good idea to set a slower report rate to make the CLI more usable:

```
set-report-period 10000
```

If everything went well, you should see the communication between the sink and the sensor every 10 seconds.

You also need to set a bootloader tag on the sensor:

```
bootloader set-tag 0xaa
```

You will need the node ID of the sensor for the next steps. You can get it using the info command:

```
node id: 0x0001
```

The next steps are different for unicast and broadcast OTA.

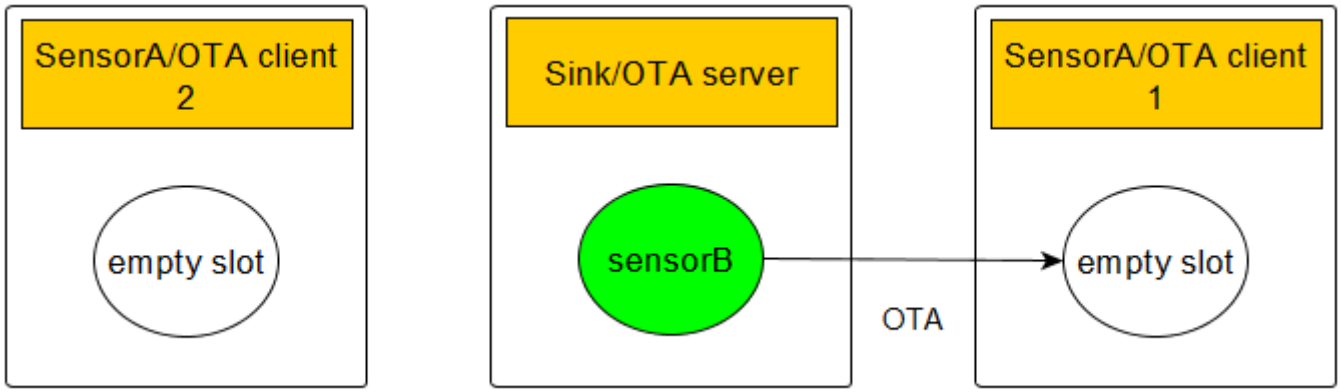
7.3.4 Unicast OTA

Use the following commands on the OTA server.

```
bootloader unicast-set-target 0x0001
```

This will set the destination of the OTA packets to the `0x0001` which is the node ID from the previous step.

With the next command you start the OTA distribution itself as shown in the following figure.



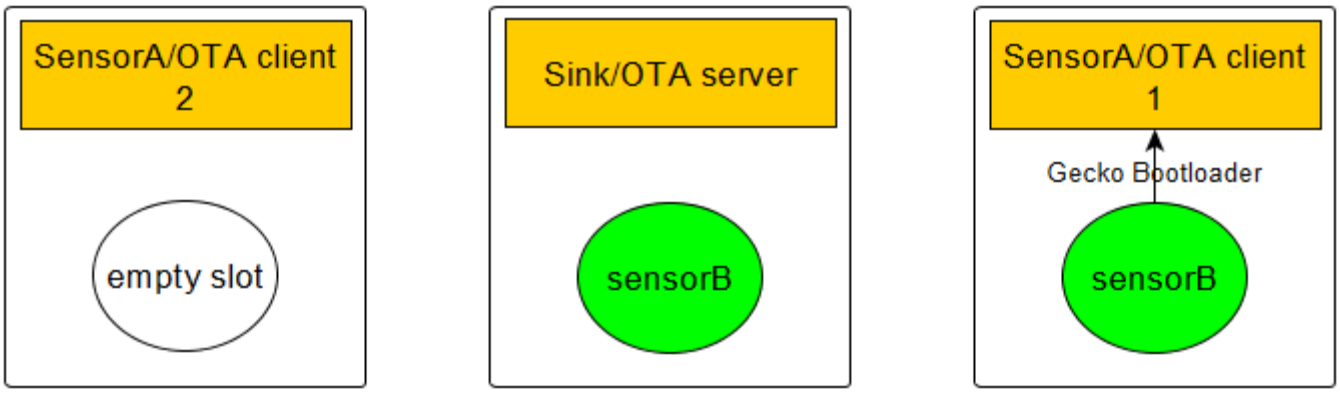

```
bootloader unicast-distribute <size> 0xaa
```

Where `<size>` is the size of the GBL image in bytes and `0xaa` is the tag you set up previously on the OTA client.

This starts the OTA process. You should see `get segment` lines on the OTA server (that is, reading segments from flash memory) and `incoming segment` lines on the OTA client. It should end with `image distribution completed, 0x00` on the server side and `Image download COMPLETED tag=0xAA size=\<size\>` on the client side.

At this point, `bootloader validate-image` should return with `valid` on the client as well.

Next, you are going to request bootloading as shown in the following figure.



To do that, execute the following command on the OTA server:

```
bootloader unicast-request-bootload 1000 0xaa
```

Where `1000` is a timeout in ms and `0xaa` is the tag again.

7.3.4.1 Unicast Download Resume Feature

The OTA download feature in Flex 2.6 supports resuming the download from the segment at which the transmission was interrupted. To test this feature, reset the server during download (by hardware reset pin or issuing the `reset` command in the CLI. When the reset cycle has completed, issue the same commands on the server as were issued on the first attempt:

```
bootloader init
bootloader unicast-set-target 0x0001
bootloader unicast-distribute <size> 0xaa
```

The download should start from the point where it was interrupted. There is no limit to the number of times the interrupt and resume can be repeated. If the tag has been changed or the flash has been erased on the client, the download will start from the beginning (that is, the whole image will be downloaded).

7.3.5 Broadcast OTA

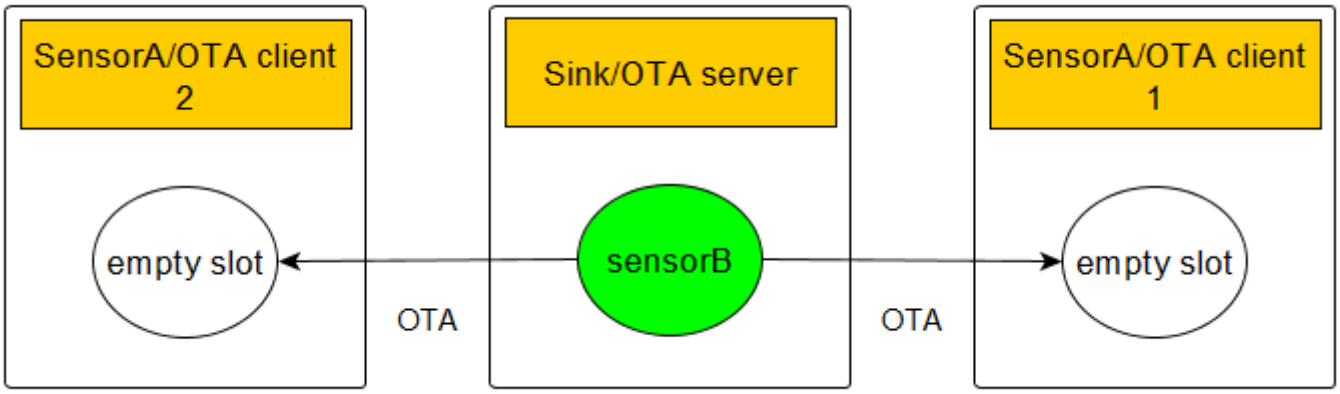
First, you set up the target list on the server. This is the list of the client's node IDs. This is used for missing segment request and bootload request messages. The maximum number of clients is limited in the broadcast plugin to 50.

The following command on the OTA server sets the node ID of the client at index `0` to `0x0001`:

```
bootloader set-target 0 0x0001
```

You can set up multiple targets with the same command by incrementing the index.

With the next command, you are starting the OTA distribution itself as shown in the following figure.



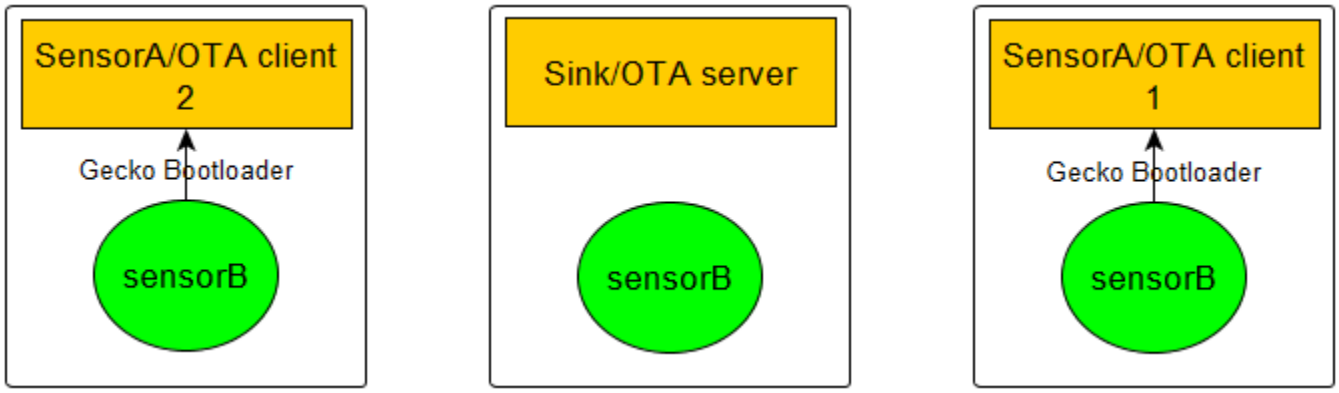

```
bootloader distribute <size> 0xaa 1
```

Where `<size>` is the size of the GBL image in bytes and `0xaa` is the tag you set up previously on the OTA client and `1` is number of clients you set up using `set-target`.

This starts the OTA process. You should see get segment lines on the OTA server (that is, reading segments from flash memory) and `incoming segment` lines on the OTA client. It should end with `image distribution completed, 0x00` on the server side and `Image download COMPLETED tag=0xAA size=<size>` on the client side.

At this point, `bootloader validate-image` should return with `valid` on the client(s) as well.

Next, you are going to request bootloading as shown in the following figure.



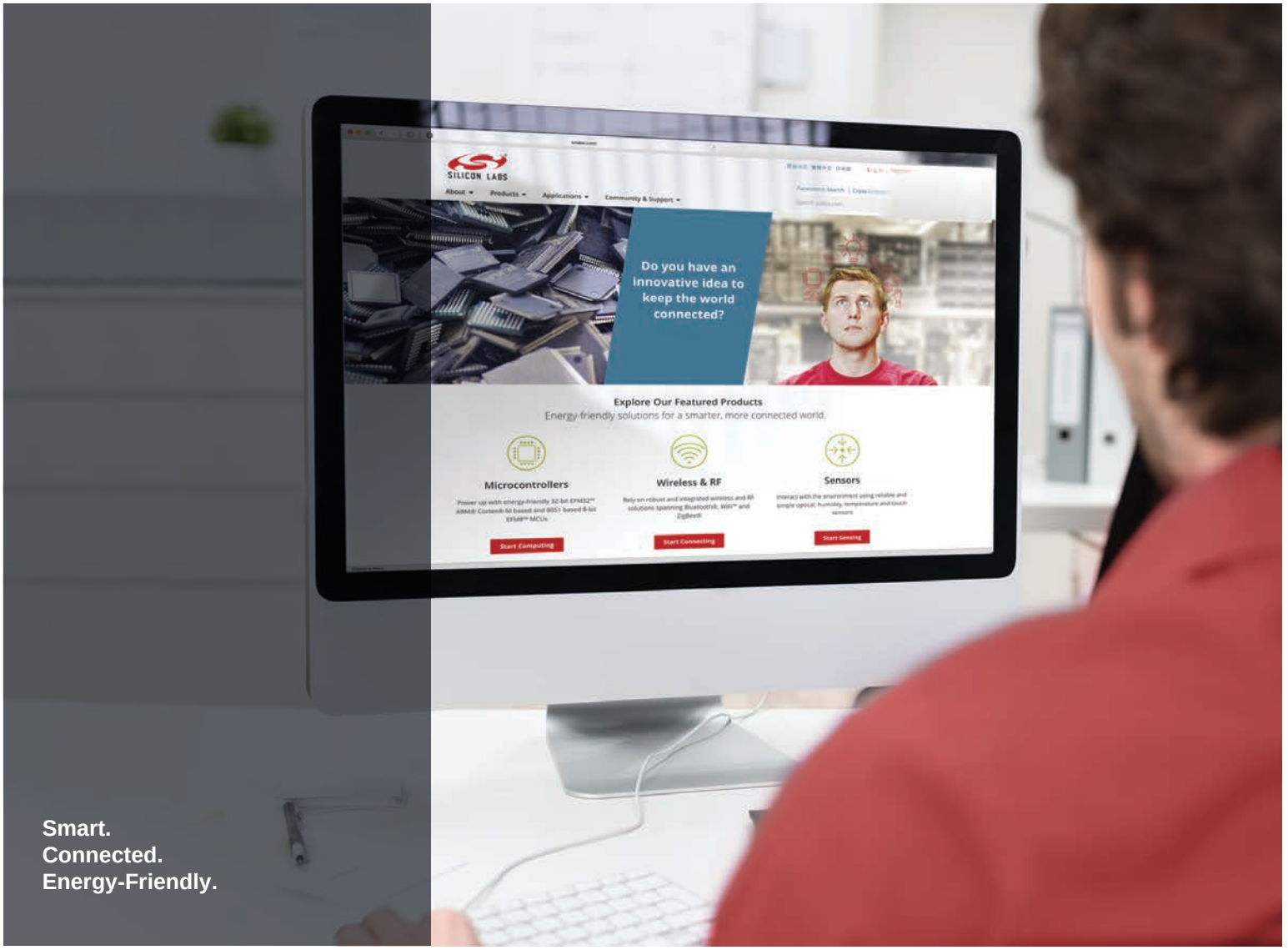
To do that, execute the following command on the OTA server:

```
bootloader request-bootload 1000 0xaa 1
```

Where `1000` is a timeout in ms and the last two parameters are the same as above for `distribute`.

7.4 Bootloading Finished

Both unicast and broadcast bootload request should immediately return with `bootload request completed`. After the timeout, you should see the bootloading of the OTA client. After bootloading, the sensor should join again because the network information is stored in NVM, but this time you should see the differences you set up for the OTA image.



Products
www.silabs.com/products

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