Device Firmware Update (DFU) Guide

M-20888-004
Septmebner 2023

Table of Contents

CHAPTER 1

Introduction

1.1 SUMMARY

IMPORTANT: onsemi plans to lead in replacing the terms “white list", "master" and "slave” as noted in this product release. We have a plan to work with other companies to identify an industry wide solution that can eradicate non-inclusive terminology but maintains the technical relationship of the original wording. Once new terminologies are agreed upon, we will update all documentation live on the website and in all future released documents.

This group of topics provides information about the tools, protocols and firmware required to perform wired and Firmware Over the Air (FOTA) firmware updates using the RSL15 Evaluation and Development Board. This guide provides an overview of the sample bootloader and its usage, includes details about the FOTA firmware images and tools, walks you step-by-step through your first firmware update using a pre-configured sample application, and shows you how to modify an existing application to support FOTA updates.

These FOTA topics are intended for firmware developers who are designing and implementing RSL applications with FOTA capability. Updating firmware is also possible with a wired connection using UART or SPI, as described in the Bootloader topic.

1.2 DOCUMENT CONVENTIONS

The following typographical conventions are used in this documentation:

`monospace font`

Assembly code, macros, functions, registers, defines and addresses.

italics

File and path names, or any portion of them.

`<angle brackets and bold>`

Optional parameters and placeholders for specific information. To use an optional parameter or replace a placeholder, specify the information within the brackets; do not include the brackets themselves.

Bold

GUI items (text that can be seen on a screen).

Note, Important, Caution, Warning

Information requiring special notice is presented in several attention-grabbing formats depending on the consequences of ignoring the information:

NOTE: Significant supplemental information, hints, or tips.

IMPORTANT: Information that is more significant than a Note; intended to help you avoid frustration.

CAUTION: Information that can prevent you from damaging equipment or software.

WARNING: Information that can prevent harm to humans.

Registers:

Registers are shown in `monospace font` using their full descriptors, depending on which core the register is accessing. The full description takes the form **`<PREFIX><GROUP>_<REGISTER>`**.

All registers are accessible from the Arm Cortex-M33 processor.

A register prefix of `D_` is used in the following circumstances:

- In cases where there are multiple instances of a block of registers, the summary of the registers at the beginning of the Register section have slightly different names from the detailed register sections below that table. For example, the `DMA*_CFG0` registers are referred to as `DMA_CFG0` when we are defining bit-fields and settings.

The firmware provides access to these registers in two ways:

- In the flat header files (e.g.: *sk5_hw_flat_cid*.h*), each register is individually accessible by directly using the naming provided in this manual. This is helpful for assembly and low-level C programming.
- In the normal header files (e.g.: *sk5_hw_cid*.h*), each register group forms a structure, with the registers being defined as members within that structure. The structures defined by these header files provide access to registers under the naming conventions `PREFIX_GROUP->REGISTER` (for the structure) and `GROUP->REGISTER` (for the register).
- For more information, see the Hardware Definitions chapter of the *RSL15 Firmware Reference*.

Default settings for registers and bit fields are marked with an asterisk (*).

Any undefined bits must be written to 0, if they are written at all.

Numbers

In general, numbers are presented in decimal notation. In cases where hexadecimal or binary notation is more convenient, these numbers are identified by the prefixes "0x" and "0b" respectively. For example, the decimal number 123456 can also be represented as 0x1E240 or 0b11110001001000000.

Sample Rates

All sample rates specified are the final decimated sample rates, unless stated otherwise.

1.3 FURTHER READING

1.4 FURTHER READING

The following documents are installed with the RSL15 system, in the default location *C:/Users/***`<your_user_name>`***/AppData/Local/Arm/Packs/ONSemiconductor/RSL15/***`<version_number>`***/documentation*. These manuals are available only in PDF format:

- *Arm TrustZone CryptoCell-312 Software Developers Manual*
- multiple CEVA manuals in the */ceva* folder

For even more information, consult these publicly-available documents:

- *Armv8M Architecture Reference Manual* (PDF download available from https://developer.arm.com/documentation/ddi0553/latest).
- *Arm Cortex-M33 Processor Technical Reference Manual*, revision r1p0, from https://developer.arm.com/documentation/100230/0100
- *Bluetooth Core Specification version 5.2*, available from https://www.bluetooth.com/specifications/adopted-specifications
- TrustZone documentation available from the Arm website at https://developer.arm.com/ip-products/security-ip/trustzone/trustzone-for-cortex-m
- Other ArmCortex-M33 publications, available from the Arm website at https://developer.arm.com/ip-products/processors/cortex-m/cortex-m33

For information about the Evaluation and Development Board Manual and its schematics, go to the RSL15 web page and navigate to the EVB page.

CHAPTER 2

Overview

These topics provide an overview of the toolset ecosystem allowing firmware over-the-air updates, and lists the necessary prerequisites for performing them.

2.1 PREREQUISITES

- RSL15 SDK CMSIS-Pack version 1.0 or later (available at https://www.onsemi.com/)
- RSL10 USB Dongle
- RSL15 Evaluation and Development Board (EVB)
- BLE Explorer (available at https://www.onsemi.com/)
- Python v3.7 or later:
    - Install package *ecdsa* version 0.13 or later.
    - Install package *pyserial* version 3.2 or later.
    - Make sure Python is added to the system path.
    - You can install the above packages using PyPI (for example, `python -m pip install ecdsa`).

- The mobile application (see Chapter 10 "RSL FOTA Mobile Application" on page 44).

CHAPTER 3

Bootloader

The bootloader source code is provided as a sample application in the RSL15 Software Development Kit (SDK), and you can use it to create your own custom bootloader.

IMPORTANT: The information in this topic refers to the general non-secure bootloader sample application. If you are using the RSL15 security features described in the *Security User's Guide*, **use the** ***secure_bootloader*** **sample application instead. For information about using the secure bootloader, refer to the Secure Bootloader Guide.**

3.1 OVERVIEW

The bootloader is a program that can update firmware images, initialize a device and possibly perform some sanity checks. The most important feature provided by the bootloader sample is the Firmware Over the Air (FOTA) functionality. Firmware can be loaded from a host microcontroller over UART or over the air from another wireless device using FOTA.

This document describes the bootloader firmware application and development tools. It shows you:

- How the bootloader firmware works
- How to create bootloader compatible applications and firmware images for RSL15
- How the bootloader protocol between the bootloader firmware and the UART updater PC tool works with the RSL15 EVB

3.2 BOOTLOADER FIRMWARE

The bootloader firmware application must be initially loaded onto an RSL15 device. It can be loaded via SWD/JTAG.

The bootloader divides the main flash memory into two areas: the app download area and the app execution area. This subdivision provides a starting point for users who want to use the bootloader for firmware update purposes. The bootloader application is the first entry point after reset. It is located at the base address of the main flash (0x00100000) and uses the first 32 KB of memory. To be compatible with the bootloader, the linker memory map of a user application needs to be updated to add a new BOOTLOADER area and shift the application origin by 32 KB (address 0x00108000).

3.2.1 Vector Table Positions

In addition, the boot image vector table contains two special items on positions 8 and 9, respectively: the application version descriptor and the image size. This means that modifications on the linker script (*sections.ld*) and the startup code (*startup.S*) are required to make a typical sample application (for example, *blinky*) compatible with the bootloader. See the "Vector Table Positions" figure (Figure 12)

NOTE: The modified linker script (*sections.ld*) and startup code (*startup.S*) files are available in the *utility* folder of the *bootloader* sample application.

3.2.2 Layout in Flash

The *bootloader* application assumes a specific layout in the main flash memory, as shown in the "Bootloader Application Layout in Flash" table (Table 1):

Table 1. Bootloader Application Layout in Flash

Area	Size	Flash Start Address
bootloader	32 KB	0x00100000
app execution	236 KB	0x00108000
app download	236 KB	0x00143000

3.2.3 Power-on-Reset

Upon a power-on-reset, the bootloader checks if there is a valid boot image in the app download area (second half of flash). If there is one, it copies/overwrites this image into the app execution area, invalidates the data in the download area and boots this new image. If no valid boot image is found in the download area, the bootloader verifies and boots from the app execution area. If no valid image is found in either area, the bootloader keeps the device trapped and prints an error message.

3.2.4 Activating the Updater

The bootloader updater is activated on one of these two occasions:

- The bootloader detects an invalid user application in the flash memory.
- The UPDATE_GPIO pin is held low while resetting the device (configurable in *drv.targ.h*)

3.2.5 Downloading the .bin Image

Using a PC and the RSL15 EVB, the download of the *.bin* image file can be performed by the provided *updater.py* PC tool, as illustrated in the "Updater Tool Downloads .bin Image File" figure (Figure 1), below:

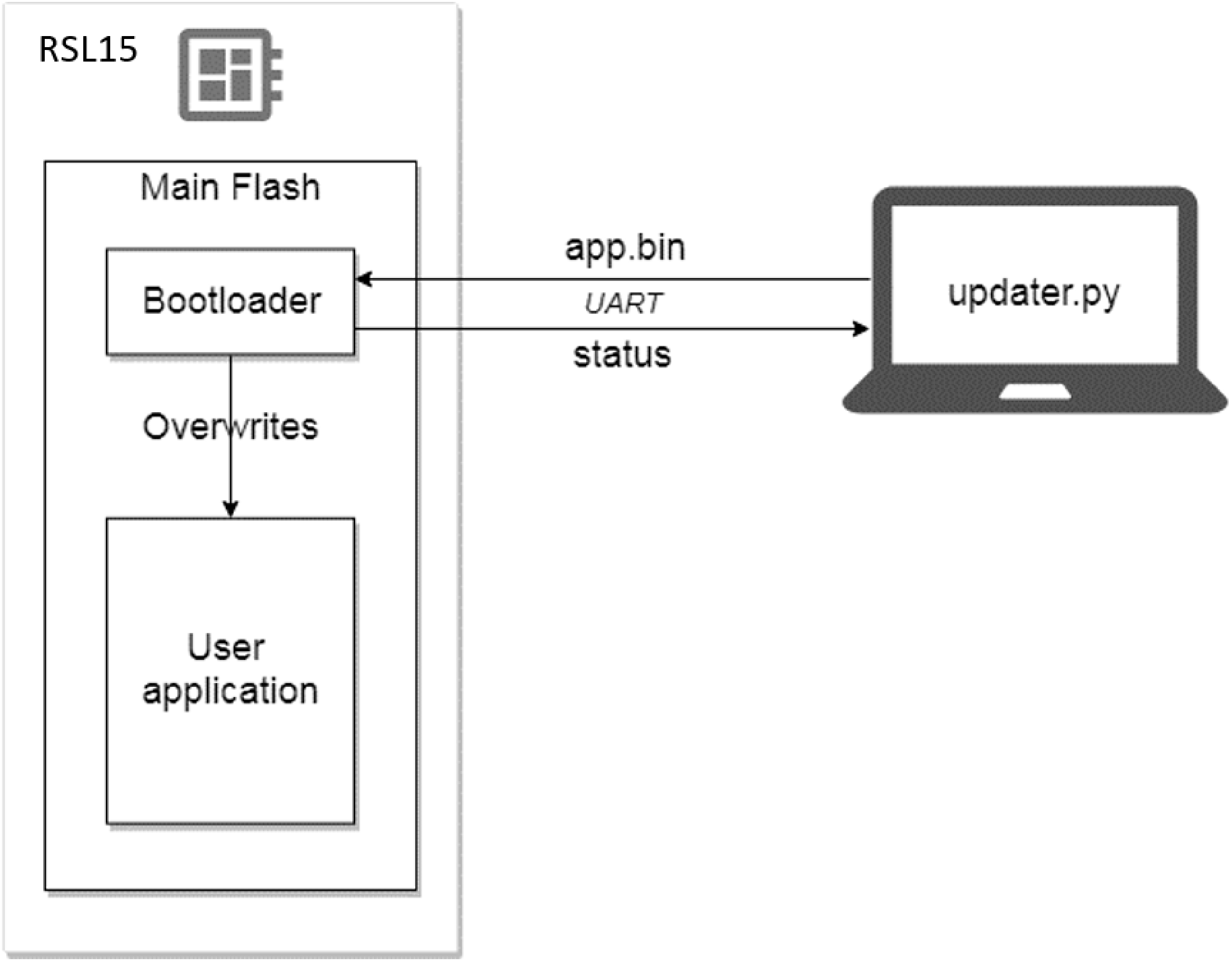


Figure 1. Updater Tool Downloads .bin Image File

3.2.6 Image Format

The application firmware image to be updated via the bootloader must be created out of the *.elf* file by calling *objcopy* for the GCC toolchain and the *_FROMELF.EXE_* tool for the Keil Arm toolchain, as explained in the *Tutorial 2* section of the bootloader sample *readme*.

3.2.7 Load Image

The bin image file can be loaded onto the RSL15 EVB using SEGGER® J-Link Commander or the UART updater.py PC tool, as explained in the *Tutorial 3* and *Tutorial 4* sections of the bootloader sample *readme*.

3.3 THE BOOTLOADER PROTOCOL FOR WRITING BINARY FILES USING UART

The bootloader protocol on the PC side is implemented in the *updater.py* tool. After activating the updater mode, the PC side has to query the bootloader version, the currently installed application version, and the flash memory sector size, with the `HELLO` command. Then the image is transferred and programmed using the `PROG` command. Once the programming is complete, the device is set to application mode.

A command can consist of several messages, but every message from the PC side must be confirmed by the RSL15 bootloader firmware before the PC side can send the next message. Except for the standard RESP message, every message is appended with a CCITT-CRC.

3.3.1 RESP

The standard response is a two-byte message. In the first byte the type is encoded: 0x55 stands for `NEXT` and 0xAA stands for `END`. The second byte for type = `NEXT` is always 0; for type = `END`, the second byte contains an error code:

- 0 = `NO_ERROR`
- 1 = `BAD_MSG`
- 2 = `UNKNOWN_CMD`
- 3 = `INVALID_CMD`
- 4 = `GENERAL_FLASH_FAILURE`
- 5 = `WRITE_FLASH_NOT_ENABLED`
- 6 = `BAD_FLASH_ADDRESS`
- 7 = `ERASE_FLASH_FAILED`
- 8 = `BAD_FLASH_LENGTH`
- 9 = `INACCESSIBLE_FLASH`
- 10 = `FLASH_COPIER_BUSY`
- 11 = `PROG_FLASH_FAILED`
- 12 = `VERIFY_FLASH_FAILED`
- 13 = `VERIFY_IMAGE_FAILED`

A message from the PC side with a bad CRC is always confirmed with `RESP(END, BAD_MSG)` by the device running the bootloader firmware. A command message with an unknown command code is confirmed with `RESP (END,UNKNOWN_CMD)`. A message with invalid parameters is confirmed with `RESP(END, INVALID_CMD)`. The standard response is the only message without an appended CCITT-CRC.

3.3.2 HELLO

The `HELLO` command message has no parameters, but because every command must be of the same length, the `HELLO` command message is padded with null bytes. The `HELLO` response message has three parameters:

1. The bootloader version **`<boot_ver>`** of type `Sys_Boot_app_version_t`.
2. The version of the currently installed application **`<app_ver>`**, also of type `Sys_Boot_app_version` (if no application is installed, **`<app_ver>`** is filled with null bytes; if `app_version` in the interrupt vector table is 0, then the application ID of **`<app_ver>`** is set to `??????`).
3. The sector size of the RSL15 flash memory in bytes.

3.3.3 PROG

The `PROG` command message has three parameters:

1. Image start address
2. Image length in bytes
3. Image hash as Ethernet CRC32

If the start address and length are valid, the command is confirmed with `RESP(NEXT)`; otherwise, `RESP(END, INVALID_CMD)` is sent. After a positive confirmation, the PC side sends data messages containing the image data in sector-sized blocks of bytes, until it has sent the last data message containing the last part of the image. Every data message is confirmed with `RESP(NEXT)` until the last data message, which is confirmed with `RESP(END, NO_ERROR)`. If an error occurs during image transmission or programming, the next confirmation is a `RESP(END,` **`<error code>`**`)`. In this case, the PC side must start the whole sequence over again.

3.3.4 Restart

Similar to the `HELLO` command, the `RESTART` command message has no parameters, and the command message is padded with null bytes. If there is a valid bootloader, this command is confirmed with `RESP(END, NO_ERROR)` and the device is rebooted. Otherwise, it is confirmed with `RESP(END, NO_VALID_BOOTLOADER)` and no operation is performed. This command is usually executed after a successful firmware update, i.e., a sequence of `PROG` command messages.

The "Sequence Diagram" figure (Figure 2) illustrates a typical message exchange between the bootloader firmware (running on RSL15) and the PC tool:

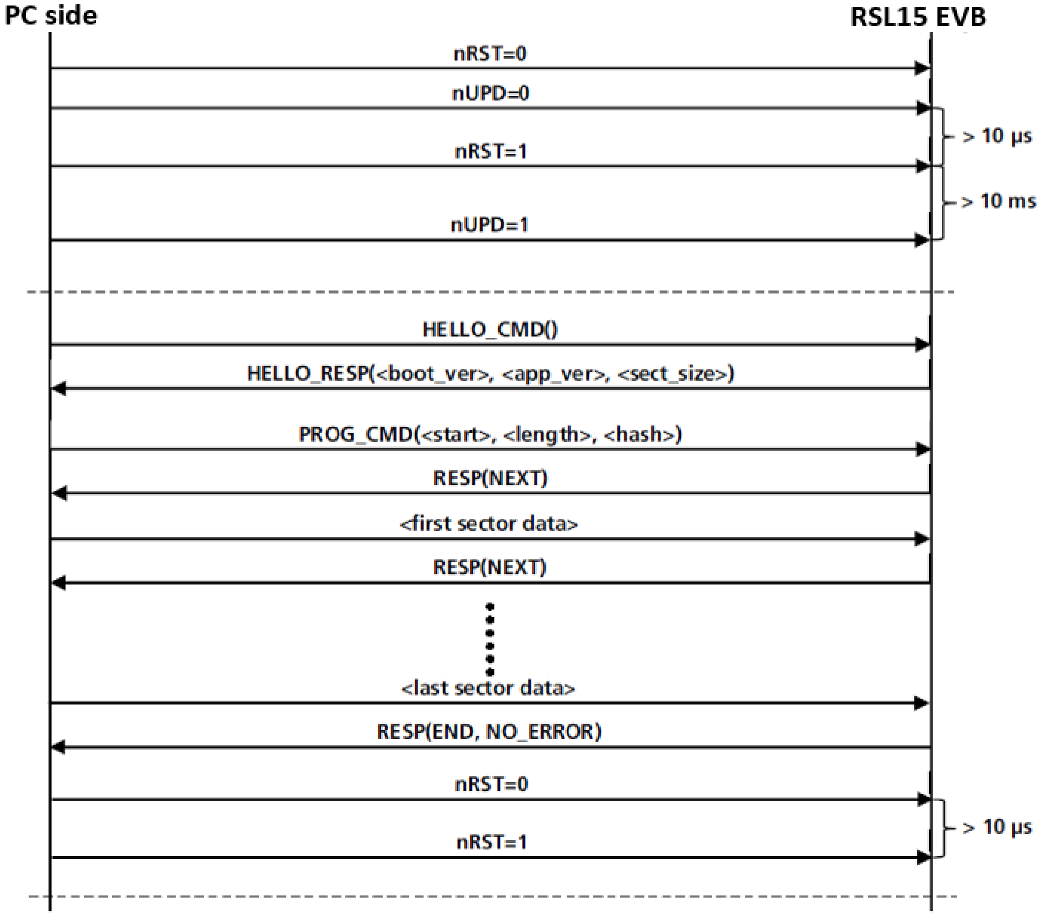


Figure 2. Sequence Diagram

CHAPTER 4

The FOTA Firmware

This topic describes the memory partitioning, setup, startup of the firmware, and updates to the application.

4.1 FOTA OVERVIEW

The RSL software ecosystem includes a set of tools that allows firmware over-the-air (FOTA) updates over a Bluetooth® Low Energy wireless link. On the PC side, a Python utility (*mkfotaimg.py*) generates FOTA-compatible firmware images, and BLE Explorer transfers the images to the remote device. BLE Explorer scans, connects and transmits the firmware image. The remote RSL device firmware side consists of a *bootloader* program, sample code, and a FOTA Bluetooth Low Energy stack that contains the device firmware update (DFU) Bluetooth Low Energy component. The "FOTA Update Setup" figure (Figure 3), below, illustrates a typical FOTA update setup from the PC point of view:

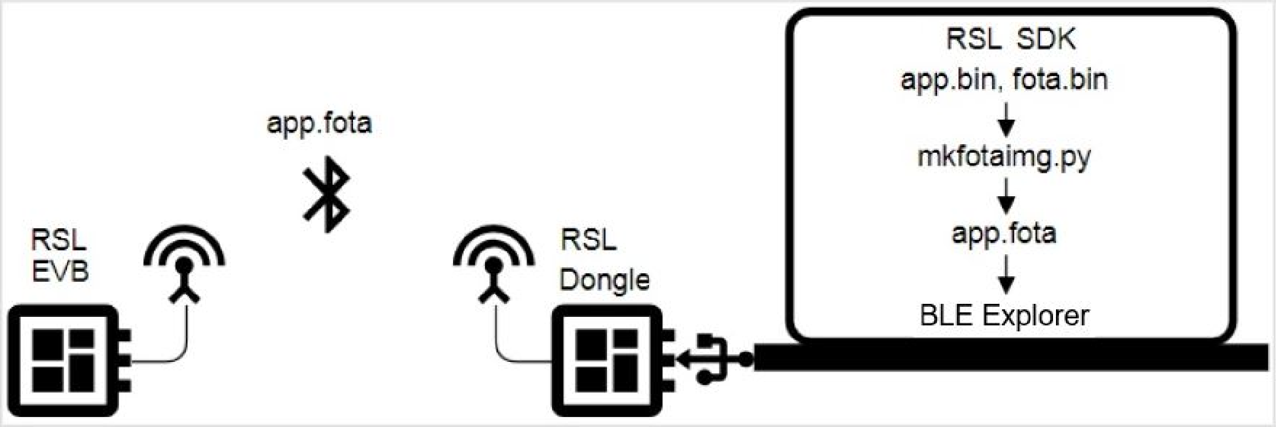


Figure 3. FOTA Update Setup

4.2 FOTA PARTITIONING

A device capable of receiving FOTA updates contains firmware composed of three parts, as illustrated in the "Memory Structure" figure (Figure 4):

1. Bootloader
2. FOTA Bluetooth Low Energy stack including DFU component (*fota.bin* sub-image)
3. User application (*app.bin* sub-image)

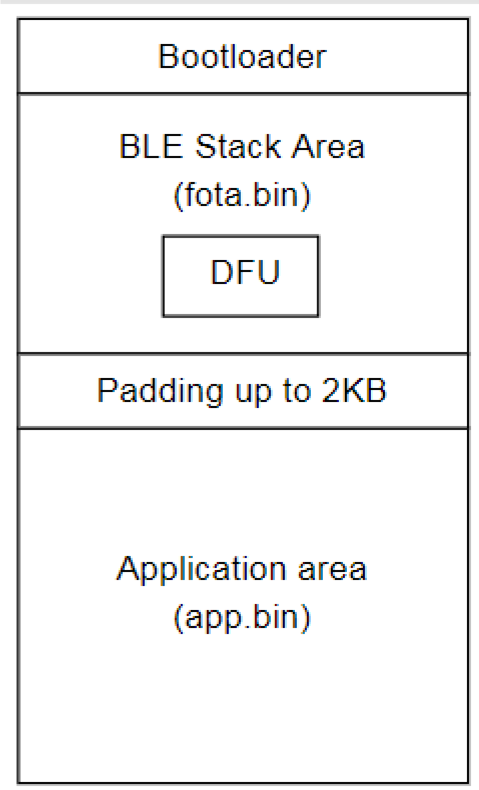


Figure 4. Memory Structure

NOTE: For specific information about using FOTA in secure applications, see the *readme* file in the *secure_fota_blinky* sample application as well as its referenced documentation.

Each part depends on the previous one, except the *bootloader*, which is standalone. The FOTA Bluetooth Low Energy stack depends on the *bootloader*. The user application depends on the FOTA Bluetooth Low Energy stack, and therefore, also depends on the *bootloader*. In terms of size, the bootloader reserves memory. The FOTA Bluetooth Low Energy stack contains the device firmware update (DFU) component and the padding between the sub-images. The memory available in flash for the application is the amount of memory that is left once memory is reserved for the FOTA Bluetooth Low Energy Stack and bootloader. The boundary between the Bluetooth Low Energy stack and application areas can be dynamic, so it is possible to increase the Bluetooth Low Energy stack size at the expense of the application size, and vice versa.

When building a FOTA-compatible RSL sample application, you may notice that the *libfota.a* library is linked. This library functions as a stub object, and implements the Bluetooth Low Energy library functions as pointers to the real implementation, located in the *fota.bin* area. This makes the link between the application and the FOTA Bluetooth Low Energy stack particularly strong, because the application code calls functions from the Bluetooth Low Energy stack directly. Therefore, an application can only run together with the specific FOTA Bluetooth Low Energy stack revision that is used when building that application. For this reason, it is not possible to update the Bluetooth Low Energy stack without updating the application. On the other hand, it is possible to perform a FOTA update of the user application only.

Two types of FOTA updates are possible:

- Application only update
- Application + FOTA Bluetooth Low Energy stack update

The DFU component, an application embedded in the FOTA Bluetooth Low Energy stack area, implements a DFU Bluetooth Low Energy custom service for FOTA updates. It contains characteristics that allow a client device to gather information from the installed firmware (such as version numbers and IDs) and download the firmware image. As the DFU is embedded into the FOTA Bluetooth Low Energy stack area, FOTA updates are possible even when no valid user application is available in the device. More details about this component are provided in Chapter 7 "The DFU" on page 27.

4.3 FIRMWARE STARTUP

Upon boot-up, the *bootloader* checks whether there is a valid user application or FOTA Bluetooth Low Energy stack programmed. The sequence of operations is as follows:

1. If there is a valid user application, start it.
2. If no valid application is found, start the FOTA Bluetooth Low Energy stack DFU component (so the device can receive FOTA updates).
3. If no valid FOTA Bluetooth Low Energy stack is found, start the *bootloader* updater (in this case, the device can only receive firmware updates over UART).

The FOTA Bluetooth Low Energy stack DFU component can be activated from the user application at any time, through a call to `Sys_Fota_StartDfu()`. More details about this are provided in later sections.

4.4 APPLICATION ONLY UPDATE

To update the application, the currently installed user application needs to start the DFU component from the Bluetooth Low Energy stack. The DFU component then starts the FOTA process and receives the new application image. The application image embeds the Build ID, calculated by the GNU linker over all symbols of *fota.bin* (see Table 5 for more about the FOTA stack Build ID). If this information does not match the installed Bluetooth Low Energy stack revision, the FOTA update is aborted. At this point the currently installed application is not destroyed.

If the FOTA Bluetooth Low Energy stack revision is compatible, the currently installed application is erased and the new image is programmed as the data comes in. When the whole image is programmed successfully, the DFU component marks the new application as valid and performs a restart. If the application update is aborted for any reason (power loss, for example), this causes the application area to contain an invalid image. The update process can be restarted from the beginning in this case, as the device still contains a valid FOTA Bluetooth Low Energy stack with the DFU component.

4.5 APPLICATION + FOTA BLUETOOTH LOW ENERGY STACK UPDATE

This type of update is performed in multiple steps:

1. Start the DFU component.
2. When the FOTA process receives a Bluetooth Low Energy stack image instead of an application image, no revision check is performed, and the downloaded image is saved in the download area. The FOTA process executes code from the Bluetooth Low Energy stack area, so it is not possible to directly replace the Bluetooth Low Energy stack.
3. After completely programming the new Bluetooth Low Energy stack image into the download area, the DFU component performs a reset and thereby activates the *bootloader*.
4. The *bootloader* detects a valid Bluetooth Low Energy stack image in the download area and copies it to the Bluetooth Low Energy stack area. This is now possible because the previous Bluetooth Low Energy stack is no

longer needed. After successfully copying the new Bluetooth Low Energy stack, the *bootloader* invalidates the Bluetooth Low Energy stack in the download area and proceeds with the firmware startup.

5. As with a usual *bootloader* startup, the DFU component of the new Bluetooth Low Energy stack detects no valid application, and therefore starts the FOTA DFU again to receive the new application image.
6. From this point on, the process is the same as it is with an application only upgrade.

Because the download area is used to temporarily hold the Bluetooth Low Energy stack image, the size of the Bluetooth Low Energy stack cannot exceed the size of the download area. This limits the Bluetooth Low Energy stack size to a maximum of 234 KB for non-secure bootloader and 218 KB for secure bootloader.

If the *bootloader* copy operation of the Bluetooth Low Energy stack image from the download area to the Bluetooth Low Energy stack area is aborted, at next startup the *bootloader* detects the still-valid Bluetooth Low Energy stack image in the download area and repeats the copy. This makes it possible to recover from power loss during the update process.

CHAPTER 5

Performing Your First FOTA Update

Before we dive into all details regarding the FOTA tools, firmware images, and protocol specifications, this topic walks you step-by-step through the process of performing your first FOTA update. The goal is to provide you a basic hands-on understanding of the RSL FOTA update process, and to ensure that your hardware and software are correctly setup. This topic shows how to:

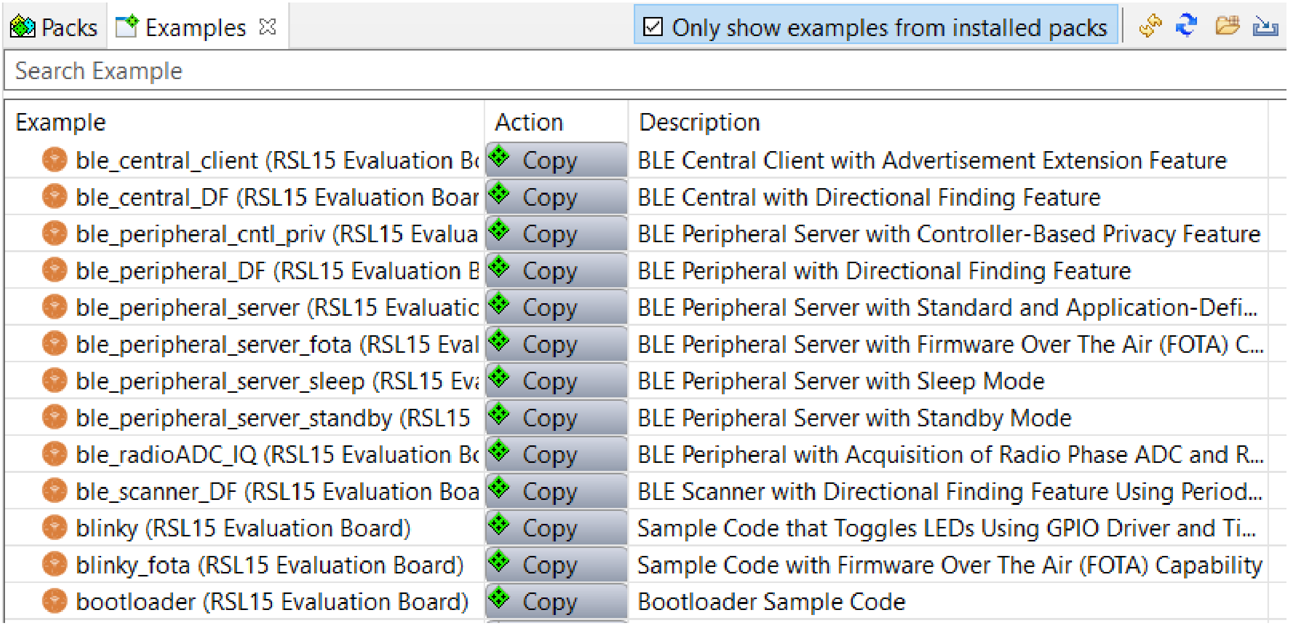
1. Generate a FOTA firmware image using the preconfigured *ble_peripheral_server_fota* sample application. This application is similar to the *ble_peripheral_server* application with added features to support FOTA updates.
2. Set up the RSL*bootloader* and load a firmware image using UART.
3. Perform a FOTA update using BLE Explorer. Alternatively, a FOTA update can be performed using the mobile application. See Chapter 10 "RSL FOTA Mobile Application" on page 44.

NOTE: For information on performing a secure FOTA update with a secure application, refer to the instructions and examples in the *secure_blinky_fota* sample application's *readme* file.

This tutorial assumes that you have installed the prerequisites and have the required version of the RSL CMSIS-Pack installed (see the *RSL Getting Started Guide* for instructions on how to import a CMSIS-Pack).

5.1 GENERATING THE FOTA FIRMWARE IMAGE

1. Open the Examples tab in the Pack Manager perspective to see example projects, included in the RSL CMSIS-Pack.
2. Find the *ble_peripheral_server_fota* example project and click the **Copy** button to import it into your workspace. (See the "Importing the FOTA Sample Project" figure (Figure 5).)

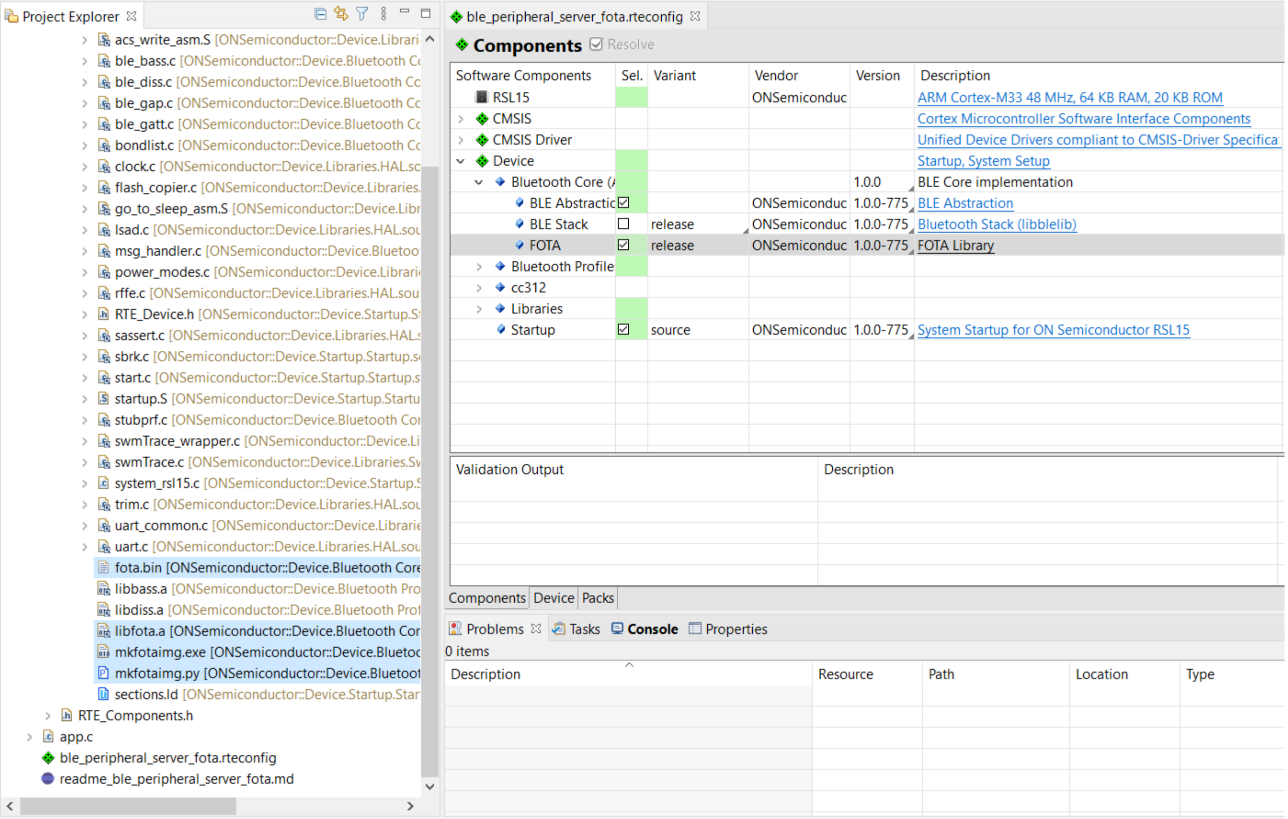


Example	Action	Description
ble_central_client (RSL15 Evaluation Bo	Copy	BLE Central Client with Advertisement Extension Feature
ble_central_DF (RSL15 Evaluation Boar	Copy	BLE Central with Directional Finding Feature
ble_peripheral_cntl_priv (RSL15 Evalua	Copy	BLE Peripheral Server with Controller-Based Privacy Feature
ble_peripheral_DF (RSL15 Evaluation B	Copy	BLE Peripheral with Directional Finding Feature
ble_peripheral_server (RSL15 Evaluatic	Copy	BLE Peripheral Server with Standard and Application-Defi...
ble_peripheral_server_fota (RSL15 Eval	Copy	BLE Peripheral Server with Firmware Over The Air (FOTA) C...
ble_peripheral_server_sleep (RSL15 Eva	Copy	BLE Peripheral Server with Sleep Mode
ble_peripheral_server_standby (RSL15	Copy	BLE Peripheral Server with Standby Mode
ble_radioADC_IQ (RSL15 Evaluation Bo	Copy	BLE Peripheral with Acquisition of Radio Phase ADC and R...
ble_scanner_DF (RSL15 Evaluation Boa	Copy	BLE Scanner with Directional Finding Feature Using Period...
blinky (RSL15 Evaluation Board)	Copy	Sample Code that Toggles LEDs Using GPIO Driver and Ti...
blinky_fota (RSL15 Evaluation Board)	Copy	Sample Code with Firmware Over The Air (FOTA) Capability
bootloader (RSL15 Evaluation Board)	Copy	Bootloader Sample Code

Figure 5. Importing the FOTA Sample Project

3. The **C/C++** perspective opens and displays your newly copied project. In the **Project Explorer** panel, you can expand your project folder and explore the files inside your project, as seen in the "Files in FOTA Sample

Project" figure (Figure 6). On the right side, the *ble_peripheral_server_fota.rteconfig* file displays the selected software components, including the new component named **FOTA**. If you expand **RTE** > **Device** > **`<device>`**, you can find the FOTA library (*libfota.a*), the FOTA Bluetooth Low Energy Stack binary file (*fota.bin*), and *mkfotaimg.exe*. These files are automatically added to your sample project once the **FOTA** component is selected.

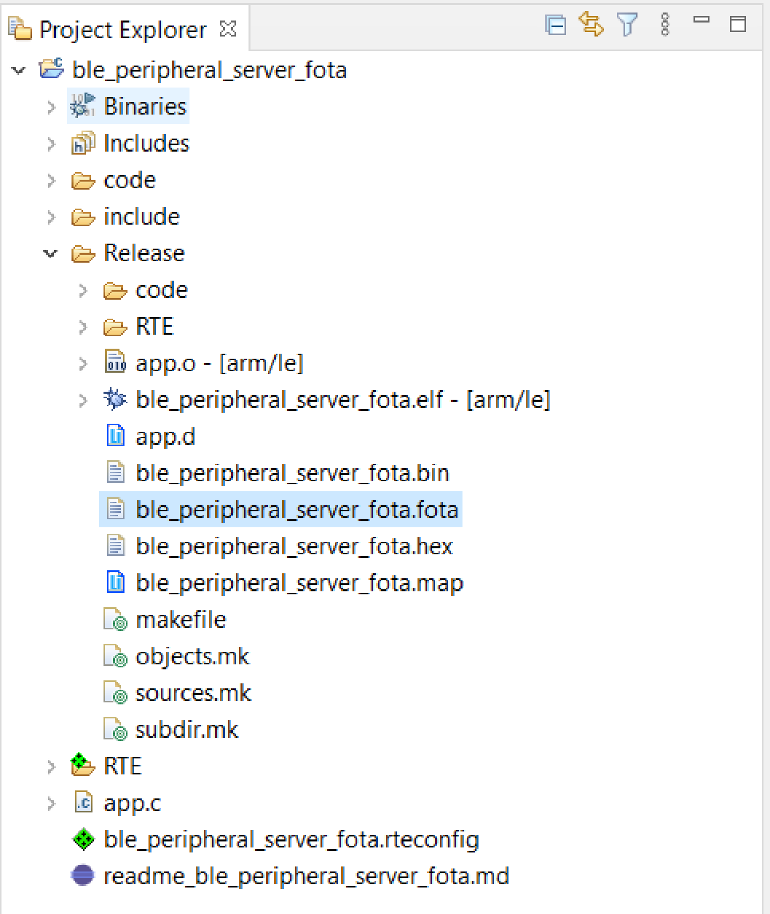


Project Explorer
acs_write_asm.S [ONSemiconductor::Device.Librari
ble_bass.c [ONSemiconductor::Device.Bluetooth C
ble_diss.c [ONSemiconductor::Device.Bluetooth Cc
ble_gap.c [ONSemiconductor::Device.Bluetooth Cc
ble_gatt.c [ONSemiconductor::Device.Bluetooth Cc
bondlist.c [ONSemiconductor::Device.Bluetooth Cc
clock.c [ONSemiconductor::Device.Libraries.HAL.sc
flash_copier.c [ONSemiconductor::Device.Libraries.
go_to_sleep_asm.S [ONSemiconductor::Device.Libr
lsad.c [ONSemiconductor::Device.Libraries.HAL.sou
msg_handler.c [ONSemiconductor::Device.Bluetoo
power_modes.c [ONSemiconductor::Device.Librari
rffe.c [ONSemiconductor::Device.Libraries.HAL.sou
RTE_Device.h [ONSemiconductor::Device.Startup.S
sassert.c [ONSemiconductor::Device.Libraries.HAL.
sbrk.c [ONSemiconductor::Device.Startup.Startup.s
start.c [ONSemiconductor::Device.Startup.Startup.s
startup.S [ONSemiconductor::Device.Startup.Startu
stubprf.c [ONSemiconductor::Device.Bluetooth Cor
swmTrace_wrapper.c [ONSemiconductor::Device.Li
swmTrace.c [ONSemiconductor::Device.Libraries.Sv
system_rsl15.c [ONSemiconductor::Device.Startup.
trim.c [ONSemiconductor::Device.Libraries.HAL.sou
uart_common.c [ONSemiconductor::Device.Librarie
uart.c [ONSemiconductor::Device.Libraries.HAL.sou
fota.bin [ONSemiconductor::Device.Bluetooth Core
libbass.a [ONSemiconductor::Device.Bluetooth Pro
libdiss.a [ONSemiconductor::Device.Bluetooth Prof
libfota.a [ONSemiconductor::Device.Bluetooth Cor
mkfotaimg.exe [ONSemiconductor::Device.Bluetoc
mkfotaimg.py [ONSemiconductor::Device.Bluetoot
sections.ld [ONSemiconductor::Device.Startup.Star
RTE_Components.h
app.c
ble_peripheral_server_fota.rteconfig
readme_ble_peripheral_server_fota.md
ble_peripheral_server_fota.rteconfig
Components
Resolve
Validation Output
Description
Components
Device
Packs
Problems
Tasks
Console
Properties
0 items
Description
Resource
Path
Location
Type

Software Components	Sel.	Variant	Vendor	Version	Description
RSL15			ONSemiconduc		ARM Cortex-M33 48 MHz, 64 KB RAM, 20 KB ROM
CMSIS					Cortex Microcontroller Software Interface Components
CMSIS Driver					Unified Device Drivers compliant to CMSIS-Driver Specifica
Device					Startup, System Setup
Bluetooth Core (/				1.0.0	BLE Core implementation
BLE Abstractic	☑		ONSemiconduc	1.0.0-775	BLE Abstraction
BLE Stack	☐	release	ONSemiconduc	1.0.0-775	Bluetooth Stack (libblelib)
FOTA	☑	release	ONSemiconduc	1.0.0-775	FOTA Library
Bluetooth Profile					
cc312					
Libraries					
Startup	☑	source	ONSemiconduc	1.0.0-775	System Startup for ON Semiconductor RSL15

Figure 6. Files in FOTA Sample Project

4. Build the *ble_peripheral_server_fota* project. After a successful build, the *ble_peripheral_server_fota.fota* image is generated under the *Release* folder, as seen in the ".fota Image in Release Folder" figure (Figure 7):



Project Explorer

ble_peripheral_server_fota
- Binaries
- Includes
- code
- include
- Release
  - code
  - RTE
  - app.o - [arm/le]
  - ble_peripheral_server_fota.elf - [arm/le]
  - app.d
  - ble_peripheral_server_fota.bin
  - ble_peripheral_server_fota.fota
  - ble_peripheral_server_fota.hex
  - ble_peripheral_server_fota.map
  - makefile
  - objects.mk
  - sources.mk
  - subdir.mk
- RTE
- app.c
- ble_peripheral_server_fota.rteconfig
- readme_ble_peripheral_server_fota.md

Figure 7. .fota Image in Release Folder

The generated *.fota* file contains both the FOTA Bluetooth Low Energy stack sub-image (*fota.bin*) and the application sub-image. This file can be used by BLE Explorer to perform a FOTA update, or by the bootloader UART PC updater tool to perform a UART firmware update.

IMPORTANT: The bootloader UART PC updater tool is found as file *updater.py* in the *bootloader/utility* folder. For more information, see Section 5.2 "Setting Up the Bootloader and Loading a Firmware Image Using UART" on page 20.

5.2 SETTING UP THE BOOTLOADER AND LOADING A FIRMWARE IMAGE USING UART

If you are running a FOTA update for the first time, your RSL EVB does not have the *bootloader* flashed into it. Set up the *bootloader* using these steps:

1. Import the *bootloader* sample application, available in the **Examples** tab.
2. Build the bootloader sample application and flash *bootloader.hex*.
3. Connect the GPIO defined as the `UPDATE_GPIO` in the *bootloader* sample application to the ground and reset your board. After reset, your RSL EVB has activated its updater mode. In this mode, the *bootloader* is waiting for commands over UART to download a new user application firmware image.

4. Use the Windows Device Manager to find out the COM port number assigned to your RSL EVB, identified by the J-Link CDC UART Port (COM3, as seen in the "RSL EVB Com Port Number" figure (Figure 8)):

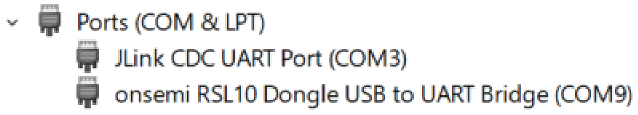

Ports (COM & LPT)
JLink CDC UART Port (COM3)
onsemi RSL10 Dongle USB to UART Bridge (COM9)

Figure 8. RSL EVB Com Port Number

5. Using a command prompt, navigate to the *bootloader/utility* folder. You can see the *updater.py* tool together with additional *.dll* dependencies.
6. Invoke the *updater.py* tool to load the *ble_peripheral_server_fota.fota* image over UART with the following command:

```
 > python updater.py COM3 ble_peripheral_server_fota.fota
 Image      : BPS  ver=1.0.0 / FOTA   ver=1.0.0
 Bootloader : BOOTL* ver=1.0.0
 *****************************************************************************************
******************************
```

You can expect similar output. For each flash sector transferred and written to the flash memory, an asterisk (*) symbol is printed on the screen.

If you find errors executing this step, make sure you have the required versions of Python and the *pyserial* package.

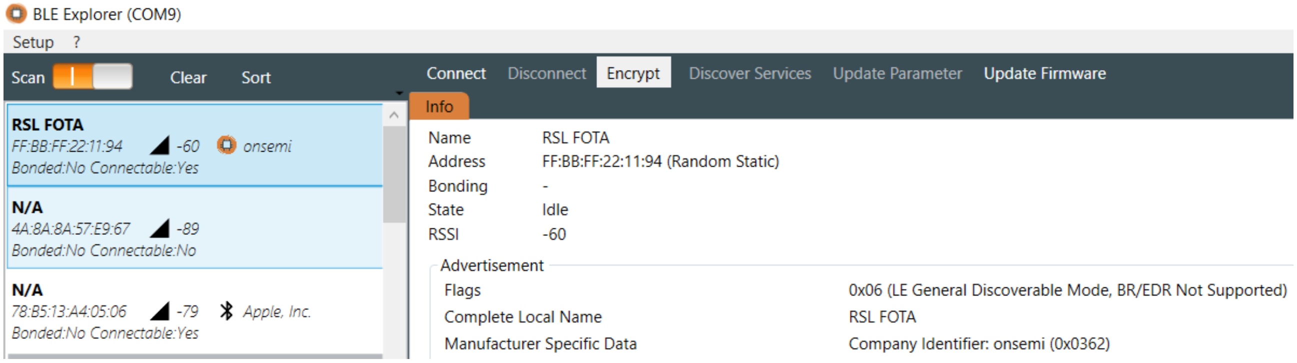
7. An LED on the EVB that is attached to one of the device’s GPIOs begins to blink.

Your device is now ready to perform a FOTA update, as it contains all the required components: the *bootloader* program and the FOTA Bluetooth Low Energy stack.

5.3 PERFORMING A FOTA UPDATE USING BLE EXPLORER

BLE Explorer is a desktop application that runs on Windows®, developed to work with the RSL USB Dongle. To use BLE Explorer for a FOTA update, follow these steps:

1. Connect FOTA_GPIO (GPIO1) to ground. After a few seconds, you can see the name **RSL FOTA** on the RSL BLE Explorer, as shown in the "FOTA Running" figure (Figure 9). After checking the name **RSL FOTA**, disconnect FOTA_GPIO from ground.
2. Click on the **Update Firmware** option on the BLE Explorer and select the **ble_peripheral_server_fota.fota** image file. (See Chapter 1 "Performing FOTA Update with BLE Explorer" on page 1). There is a 30-second wait for image file selection. After 30 seconds, control returns to the *ble_peripheral_server_fota* application.



RSL FOTA
FF:BB:FF:22:11:94 -60 onsemi
Bonded:No Connectable:Yes

N/A
4A:8A:8A:57:E9:67 -89
Bonded:No Connectable:No

N/A
78:B5:13:A4:05:06 -79 Apple, Inc.
Bonded:No Connectable:Yes

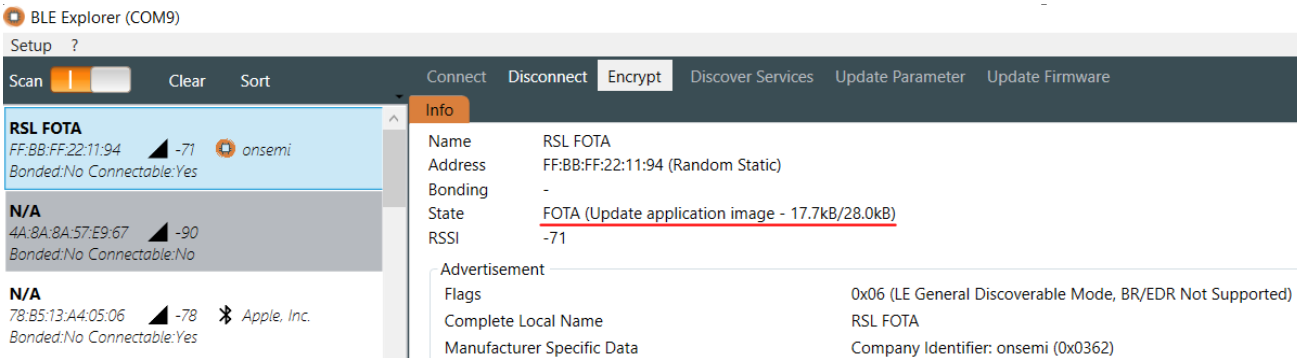
Name	RSL FOTA
Address	FF:BB:FF:22:11:94 (Random Static)
Bonding	-
State	Idle
RSSI	-60

Advertisement

Flags	0x06 (LE General Discoverable Mode, BR/EDR Not Supported)
Complete Local Name	RSL FOTA
Manufacturer Specific Data	Company Identifier: onsemi (0x0362)

Figure 9. FOTA Running

3. Once the FOTA firmware image file is selected, it starts updating the firmware, as shown in the "Updating Firmware" figure (Figure 10).



RSL FOTA
FF:BB:FF:22:11:94 -71 onsemi
Bonded:No Connectable:Yes

N/A
4A:8A:8A:57:E9:67 -90
Bonded:No Connectable:No

N/A
78:B5:13:A4:05:06 -78 Apple, Inc.
Bonded:No Connectable:Yes

Name	RSL FOTA
Address	FF:BB:FF:22:11:94 (Random Static)
Bonding	-
State	FOTA (Update application image - 17.7kB/28.0kB)
RSSI	-71

Advertisement

Flags	0x06 (LE General Discoverable Mode, BR/EDR Not Supported)
Complete Local Name	RSL FOTA
Manufacturer Specific Data	Company Identifier: onsemi (0x0362)

Figure 10. Updating Firmware

4. Upon firmware update completion, **Firmware update succeeded** is displayed.

CHAPTER 6

FOTA Image

The FOTA Image (*.fota* file) consists of two sub-images, with padding to a multiple of 2048 bytes in between so that the start of the second image lies on an RSL flash sector boundary.

6.1 OVERVIEW

The FOTA image's first sub-image is the FOTA Bluetooth Low Energy Stack (*fota.bin*) and the second one is the user application (*app.bin*). The Python utility *mkfotaimg.py* generates the FOTA image, as illustrated below in the "Image Format" figure (Figure 11):

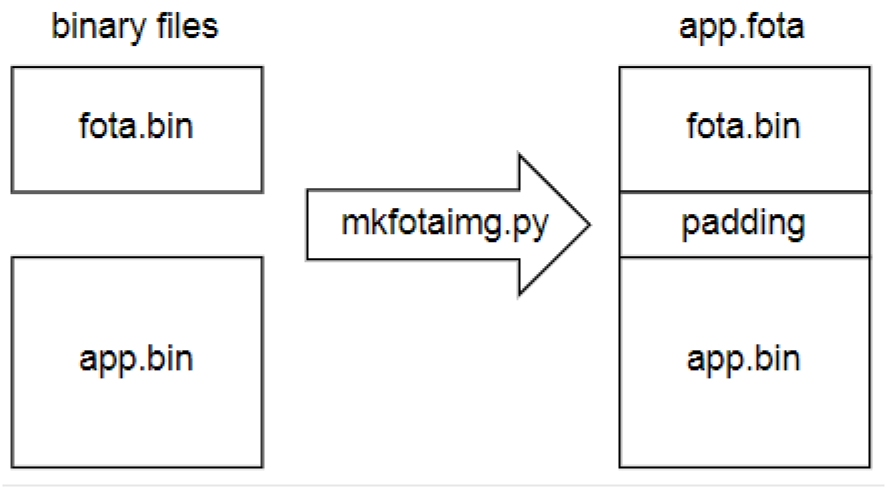


Figure 11. Image Format

The *.fota* file can then be used as input to the *updater.py* tool to perform a UART firmware update via the *bootloader*. For more information about the *bootloader*, see the *Bootloader User's Guide*.

6.2 MKFOTAIMG.PY

The *mkfotaimg.py* tool takes two positional arguments as inputs: the FOTA Bluetooth Low Energy stack sub-image and the user application sub-image. Its usage and optional arguments are shown below.

Usage:

```
 > mkfotaimg.py [-h] [--version] [-d UUID] [-s SecureBoot] [-i UUID][-n NAME] [-o OUT-
IMG] FOTA-IMG APP-IMG
```

Arguments: see the "Positional Arguments" table (Table 2) and the "Optional Arguments" table (Table 3).

Table 2. Positional Arguments

Argument	Meaning
FOTA-IMG	FOTA stack sub-image containing the Bluetooth Low Energy stack and the DFU component (.bin file)
APP-IMG	application sub-image (.bin file)

Table 3. Optional Arguments

Argument	Meaning
-h, --help	show help message and exit
--version	show program's version number and exit
-d UUID, --devid UUID	device UUID to embed in the image (default: no ID)
-s SecureBoot, --secure SecureBoot	the size of secure bootloader for the secure FOTA (default: non-secure)
-i UUID, --srvid UUID	advertised UUID to embed in the image (default: DFU service UUID)
-n NAME, --name NAME	advertised name to embed in the image (default: `ON RSL FOTA`)
-o OUT-IMG	name of output image file (default: **`<APP-IMG>`**.*fota*)

This Python utility can be integrated into the post-build steps of Eclipse to generate the *.fota* format file every time a user application project is built. This configuration is available in **Project** > **Properties** > **C/C++ Build** > **Settings** > **Build Steps** > **Post-Build Steps**. Two steps are required to generate the *.fota* image. First, we need to generate the *app.bin*, which is the application sub-image. This is done using the *objcopy* tool, as follows:

```
arm-none-eabi-objcopy -O binary <app_name>.elf <app_name>.bin
```

Then, we can call the *mkfotaimg.py* utility to generate the final FOTA image (*app.fota file*) from the stack sub-image (*fota.bin*) and the application sub-image (*app.bin*) generated above the *.fota* image, as follows:

```
mkfotaimg.py <app_name>.fota  fota.bin <app_name>.bin
```

In order to invoke both commands, the two can be combined with the “&&” operator. Chapter 8 "Integrating FOTA Into Your Application" on page 33, walks you step-by-step through generically configuring this post-build step for any application, including the path for the Python utility, FOTA stack binary file, and application binary file.

6.3 SUB-IMAGE FORMAT

A sub-image contains the 1-to-1 flash content for RSL devices. We use positions 8 and 9 of the vector table to store pointers for the version info and the image descriptor, as shown in the "Vector Table Positions" figure (Figure 12):

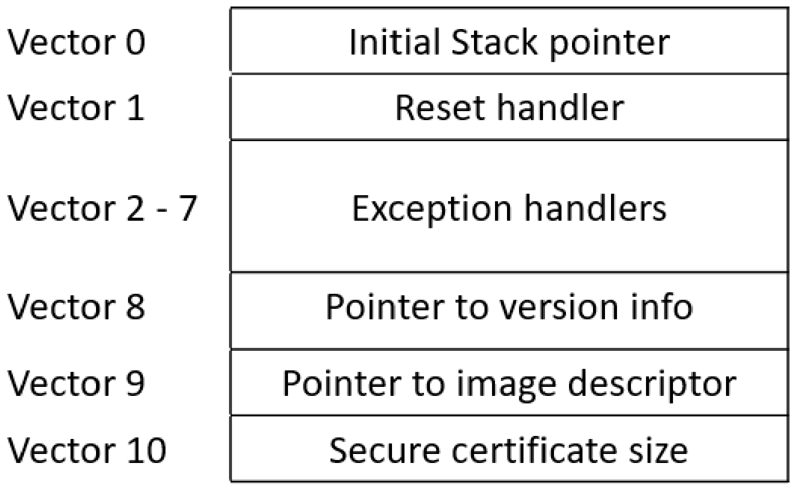


Figure 12. Vector Table Positions

Every vector is a 32-bit value. With the Reset handler vector, the image start address can be calculated as:

$$< \mathit{image\ start\ address} > = < \mathit{Reset\ handler\ vector} > \backslash\&\sim\mathrm{0x7FF}$$

To find the version info and image descriptor, we calculate the corresponding image offsets by subtracting the image start address from the vector value:

$$<\text{offset version info}> = <\text{Pointer to version info}> - <\text{image start address}>$$

$$<\text{offset image descriptor}> = <\text{Pointer to image descriptor}> - <\text{image start address}>$$

All multi-byte values in the description are in little-endian format.

The version info has the following format:

```
typedef struct
{
    char     id[6];        /* ID string */
    uint16_t num;          /* <major[15:12]>.<minor[11:8]>.<revision[7:0]> */
} version;

typedef struct
{
    version  img_ver      /* image version */
    uint8_t  dev_id[16]; /* device UUID set by mkfotaimg (default all 0s) */
} version_info;
```

In the FOTA stack sub-image, the version info is directly followed by a configuration structure:

```
typedef struct
{
    uint32_t length;        /* length of this structure in bytes */
    uint8_t pub_key[64];    /* public signing key set by mkfotaimg */
                            /* (default all 0s)

    uint8_t srv_id[16];     /* service UUID used when advertising */
                            /* set by mkfotaimg (defaults to the  */

                            /* DFU service ID)

    uint16_t dev_name_len;  /* device name length set by mkfotaimg */
                            /* (defaults to 13)

    uint8_t dev_name[29];   /* device name used when advertising */
                            /* set by mkfotaimg (defaults to */

                            /* "RSL FOTA ") */

  } config_info;
```

The image descriptor has the following format:

```
typedef struct
{
    uint32_t image_size;  /* image size in bytes excluding the signature */
    uint32_t build_id[8]; /* FOTA stack build ID */
} image_descriptor;
```

The FOTA stack build IDs from the two sub-images must match; otherwise it means that the application image has been linked against another version of the FOTA stack image as included in the FOTA image, in which case an `IMAGE_DNL_BAD_BUILDID` error is generated. The public key derived from the signing key is stored in the configuration structure of the FOTA stack sub-image (see above `struct config_info` field `pub_key`). The image size found in the image descriptor excludes the signature, so you must add 64 to the size to get the total sub-image size.

CHAPTER 7

The DFU

The Device Firmware Update (DFU) component, which is embedded in the FOTA stack, acts as the server for the DFU service. This component performs the actual update via the DFU custom service.

7.1 DFU COMPONENT

The DFU component is activated automatically at device startup if no valid application sub-image is found on the device. The application can also start the DFU component by calling the function `Sys_Fota_StartDfu()`.

IMPORTANT: Sometimes smartphones do not handle a service changed indication correctly. Therefore, in the DFU initialization, the first two bits of the device's Bluetooth address are set to 0 to force the smartphone to perform a service discovery.

7.2 UPDATE SEQUENCE

The DFU client scans for a device advertising the configured service ID (normally the DFU service ID). When the device is found, the DFU client connects to the device and reads its characteristics (Device ID, Versions, Build ID). The client must only accept FOTA images with matching Device IDs, unless the Device ID characteristic is all 0s, which means the device is compatible with all FOTA images.

If the Build ID of the application sub-image differs from the Build ID characteristic, the client needs to download the FOTA stack sub-image first. After this downloads successfully, the client needs to disconnect from the device, which restarts the device with the new FOTA stack. Now the client needs to reconnect to the same device.

If the Build ID of the application sub-image matches the Build ID characteristic, the client needs to download the Application sub-image. After successfully downloading, the client needs to disconnect from the device, which restarts the device with the new user application.

7.3 FOTA STACK SOURCE CODE

The RSL CMSIS-Pack includes a prebuilt version of the FOTA Stack available in the form of a software component, as shown in Chapter 1 "Performing Your First FOTA Update" on page 1. The *fota.bin* and *libfota.a* files are located under **`<cmsis_pack_root>`**\ONSemiconductor\**`<device>`\`<version>`**\lib\Release. The source code to generate these files is also available in the CMSIS-Pack, under **`<cmsis_pack_root>`**\ONSemiconductor\**`<device>`\`<version>`**\firmware\source\lib\fota.

If you would like to customize and rebuild these files, follow these steps:

1. Import the source code project. Navigate to **File** > **Import** > **General** > **Existing Projects into Workspace**, set the **Select root directory:** option with the path to the source code **`<cmsis_pack_root>`**\ONSemiconductor\**`<device>`\`<version>`**\firmware\source\lib\fota, mark the checkbox **Copy projects into workspace**, and click **Finish**. The project appears in the left side of **Project Explorer**, as shown in the "Copying the Project" figure (Figure 13).

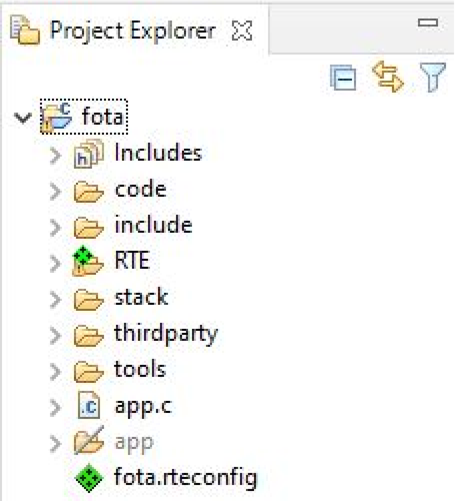
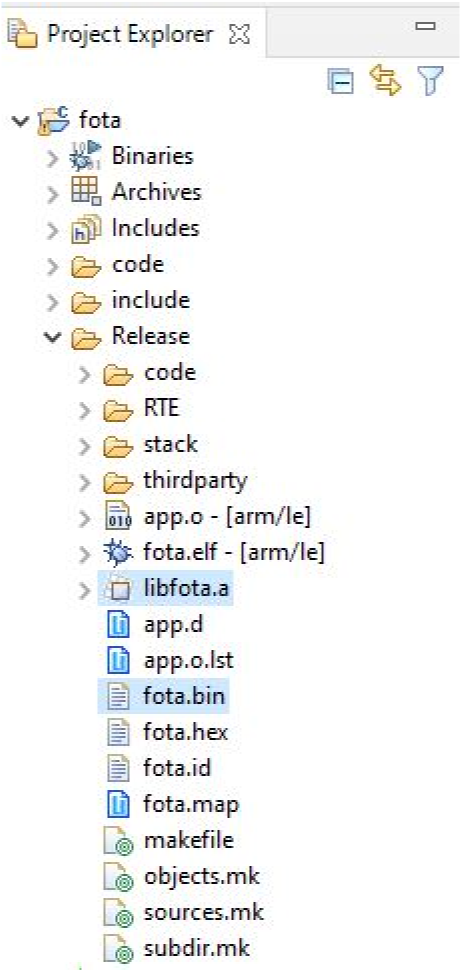


Figure 13. Copying the Project

2. Modify the source code. For example, change the FOTA stack version number in *app.conf.h*.

```
#define FOTA_VER_MAJOR                          1

        #define FOTA_VER_MINOR                          1 //0

        #define FOTA_VER_REVISION                       0
```

3. Build the project. After building successfully, *libfota.a* and *fota.bin* are generated under the *Debug* or *Release* folder, as shown in the "Building the Project" figure (Figure 14).



Project Explorer

```
fota
    Binaries
    Archives
    Includes
    code
    include
    Release
        code
        RTE
        stack
        thirdparty
        app.o - [arm/le]
        fota.elf - [arm/le]
        libfota.a
        app.d
        app.o.lst
        fota.bin
        fota.hex
        fota.id
        fota.map
        makefile
        objects.mk
        sources.mk
        subdir.mk
```

Figure 14. Building the Project

4. To use your customized files, replace the *libfota.a* and the *fota.bin* in your in your CMSIS-Pack installation (**`<cmsis_pack_root>`***ONSemiconductor***`<device>`****<version>***lib**Release*).
5. After replacing these files, it is necessary to refresh the *RTE* folder of existing projects in your workspace, so that the IDE uses the newly generated files. The simplest way to do this is to delete the *fota.bin* and *libfota.a* files under **RTE** > **Device** > **`<device>`**, and right-click the project name and choose **Refresh** to refresh your sample project (for example, you can do this in *ble_peripheral_server_fota*).
6. Rebuild the existing project so that a new *.fota* image is generated based on your modified *libfota.a* and *fota.bin* files.
7. Run BLE Explorer to perform a FOTA update. The tool detects the updated version number of the stack **FOTA stack version: FOTA `<higher_version_number>`**, in comparison to the one installed on the device. (In this example, we are using **FOTA stack version: FOTA 1.1.0**, for when version 1.0.0 is installed.) This means a full update is required (FOTA Stack + application). This is performed in two steps: first, the FOTA stack is updated; next, the device resets and the user application is updated.

If you would like to create the secure FOTA library for secure applications, follow these steps:

1. Make sure you have the FOTA application in your workspace.
2. Right-click the FOTA project name and choose **3 Release_Secure** under **Build configurations** > **Set Active.**
3. You can find the defined symbols under **Preprocessor** in **Settings for the Secure FOTA:**
    - `CFG_SECURE_FOTA`
    - `BL_APPLICATION_BASE=0x10D800`
    - `BL_DOWNLOAD_BASE=0x144000`
    - `BL_SECURE_STORAGE_SIZE=0x2C00`

    `BL_APPLICATION_BASE` is the start address of FOTA, `BL_DOWNLOAD_BASE` is the start address of the download area, and `SECURE_STORAGE_SIZE` is the size of the secure storage area.

4. You can find the option for the secure bootloader size in **Project** > **Properties** > **C/C++ Build** > **Settings** > **Build Steps** > **Post-build steps**.
    - `Release_Secure Bootloader_size` 0xD800

    0xD800 means the secure bootloader size.

5. Build the project. After building successfully, *libfota.a* and *fota.bin* are generated under the *Release_Secure* folder.
6. To use the secure FOTA files, replace the *libfota.a* and the *fota.bin* files in your CMSIS-Pack installation **`<cmsis_pack_root>`**\ONSemiconductor\<device>**`<version>`**\lib\Release_Secure).

7.4 DFU BLUETOOTH LOW ENERGY SERVICE

On the RSL device the DFU service is used as the server; the DFU service client runs on the PC (BLE Explorer). The "DFU Service UUID" table (Table 4) shows the UUID of the DFU service.

Table 4. DFU Service UUID

Requirement	UUID
Mandatory for DFU component, optional for device application	b2152466-d600-11e8-9f8b-f2801f1b9fd1

If the device application does not implement this service, another method must be used to activate the DFU mode in the device (e.g. connect FOTA_GPIO to ground).

7.5 DFU SERVICE CHARACTERISTICS

The "DFU Service Characteristics and Their Properties" table (Table 5) lists the DFU service characteristics and provides information about them.

Table 5. DFU Service Characteristics and Their Properties

Characteristic	UUID	Properties	Length	Description	Requirements
Transport	b2152466-d601-11e8-9f8b-f2801f1b9fd1	Notify, Write without response	variable (max. 512)	Internally used characteristic to transport data with the DFU protocol.	Mandatory for DFU component, prohibited for device application.
Device ID	b2152466-d602-11e8-9f8b-f2801f1b9fd1	Read	16	Device ID as found in the version info of the FOTA stack sub-image. Only FOTA images with the same device ID are compatible, unless this characteristic is all 0s, in which case all FOTA images are compatible.	Mandatory for DFU component, optional for device application.
BootLoader Version	b2152466-d603-11e8-9f8b-f2801f1b9fd1	Read	8 (Format struct version)	Version of the installed BootLoader.	Mandatory for DFU component, optional for device application.
FOTA Stack Version	b2152466-d604-11e8-9f8b-f2801f1b9fd1	Read	8 (Format struct version)	Version of the installed FOTA stack sub-image (of type struct version).	Mandatory for DFU component, optional for device application
Application Version	b2152466-d605-11e8-9f8b-f2801f1b9fd1	Read	8 (Format struct version)	Version of the installed application sub-image. If no valid application sub-image is currently installed, then all 0s is returned.	Mandatory for DFU component, optional for device application.
FOTA Stack Build ID	b2152466-d606-11e8-9f8b-f2801f1b9fd1	Read	32	Build ID as found in the descriptor of the FOTA stack sub-image. If the Build ID of the FOTA image to download is different from the one in this characteristic, then both sub-images must be updated; otherwise, updating only the application sub-image is sufficient.	Mandatory for DFU component, optional for device application
Enter DFU	b2152466-d607-11e8-9f8b-f2801f1b9fd1	Write	1	A write with the value 1 switches from the Application mode to the DFU mode.	Prohibited for DFU component, mandatory for device application.

7.6 DFU PROTOCOL

The application layer uses a Command/Response scheme. Every Command/Response consists of a standard header and an optional body. The header has the following format:

```
typedef struct
{
    uint8_t  code;      /* unique Command/Response code */
    uint8_t  param[3]; /* Command/Response specific parameters*/
    uint32_t body_len; /* length of the following body (0 = no body)*/
} header;
```

Currently only a single Command/Response is specified: IMAGE_DOWNLOAD (code = 1). The command is used to transfer a sub-image to the RSL device. The three parameters are not used by the command and must be set to 0. The body contains one of the two sub-images including the signature. The response signals back the success or failure of the operation; in the case of failure, the RSL device can send the response before the whole command body is transferred. The response itself has no body. `Param[0]` is used for the status; the other parameters are not used, and must be set to 0.

The "Status Codes" table (Table 6), below, shows the specific status codes:

Table 6. Status Codes

Code	Meaning
0	The sub-image has been downloaded successfully
1	Download rejected due to incompatible Device ID
2	Download rejected due to incompatible Build ID (only for application sub-images)
3	Download rejected due to image size too large or small
4	Download failed due to flash storage error
5	Download failed due to invalid signature
6	Download rejected due to invalid start address

For the transport layer, an HDLC-like protocol with windowing is used to transport the upper layer SDUs. For the data-link layer, the transport characteristic of the DFU service is used.

CHAPTER 8

Integrating FOTA Into Your Application

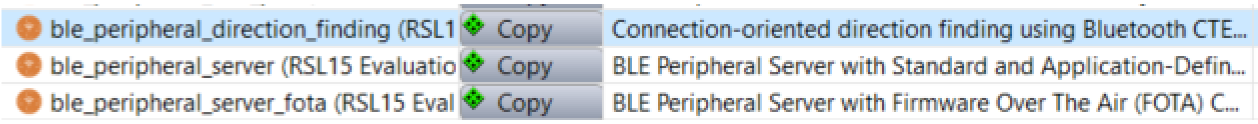
This topic shows how to modify an existing sample application to make it capable of receiving FOTA updates. We start with the *ble_peripheral_direction_finding* application, and walk you step-by-step through all the required project configurations and firmware changes. Then we show how to perform a FOTA update to confirm that the integration has been executed successfully.

NOTE: This information refers to the general FOTA usage. For information on performing a secure FOTA update, refer to the instructions and examples in the *secure_blinky_fota* sample application's *readme* file.

8.1 MODIFYING THE APPLICATION

Modify the application with the following steps:

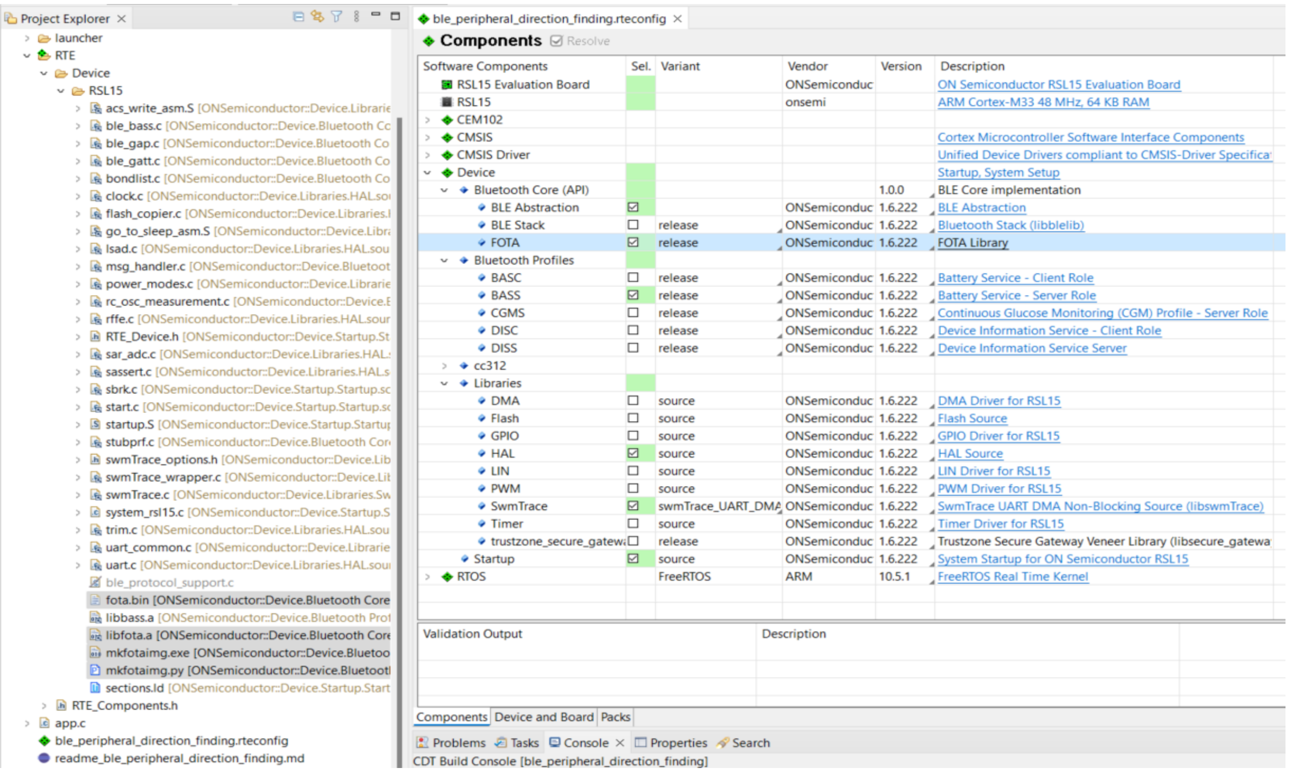
1. Copy the *ble_peripheral_direction_finding* sample application into your workspace. The **C/C++** perspective opens and displays your newly copied project, as shown below in the "Bringing the Project Into the Workspace" figure (Figure 15).



ble_peripheral_direction_finding (RSL1	Copy	Connection-oriented direction finding using Bluetooth CTE...
ble_peripheral_server (RSL15 Evaluatio	Copy	BLE Peripheral Server with Standard and Application-Defin...
ble_peripheral_server_fota (RSL15 Eval	Copy	BLE Peripheral Server with Firmware Over The Air (FOTA) C...

Figure 15. Bringing the Project Into the Workspace

2. Add the FOTA software component to your project by using the RTE Configuration Wizard (the *ble_peripheral_direction_finding.rteconfig* file). When you select the **FOTA** component and click save, *libfota.a*, *fota.bin*, *mkfotaimg.exe* and the tool *mkfotaimg.py* are copied into your project, under **RTE** > **Device** > **`<device>`**. In addition, you need to deselect the **BLE Stack** component as shown in the "Adding the FOTA Software Component" figure (Figure 16).


Project Explorer
launcher
RTE
Device
RSL15
acs_write_asm.S [ONSemiconductor::Device.Librarie
ble_bass.c [ONSemiconductor::Device.Bluetooth Co
ble_gap.c [ONSemiconductor::Device.Bluetooth Co
ble_gatt.c [ONSemiconductor::Device.Bluetooth Co
bondlist.c [ONSemiconductor::Device.Bluetooth Co
clock.c [ONSemiconductor::Device.Libraries.HAL.so
flash_copier.c [ONSemiconductor::Device.Libraries.I
go_to_sleep_asm.S [ONSemiconductor::Device.Libr
lsad.c [ONSemiconductor::Device.Libraries.HAL.sou
msg_handler.c [ONSemiconductor::Device.Bluetoot
power_modes.c [ONSemiconductor::Device.Librarie
rc_osc_measurement.c [ONSemiconductor::Device.E
rffe.c [ONSemiconductor::Device.Libraries.HAL.sour
RTE_Device.h [ONSemiconductor::Device.Startup.St
sar_adc.c [ONSemiconductor::Device.Libraries.HAL.
sassert.c [ONSemiconductor::Device.Libraries.HAL.s
sbrk.c [ONSemiconductor::Device.Startup.Startup.sc
start.c [ONSemiconductor::Device.Startup.Startup.sc
startup.S [ONSemiconductor::Device.Startup.Startup
stubprf.c [ONSemiconductor::Device.Bluetooth Core
swmTrace_options.h [ONSemiconductor::Device.Lib
swmTrace_wrapper.c [ONSemiconductor::Device.Lib
swmTrace.c [ONSemiconductor::Device.Libraries.Sw
system_rsl15.c [ONSemiconductor::Device.Startup.S
trim.c [ONSemiconductor::Device.Libraries.HAL.sou
uart_common.c [ONSemiconductor::Device.Librarie
uart.c [ONSemiconductor::Device.Libraries.HAL.sou
ble_protocol_support.c
fota.bin [ONSemiconductor::Device.Bluetooth Core
libbass.a [ONSemiconductor::Device.Bluetooth Prof
libfota.a [ONSemiconductor::Device.Bluetooth Core
mkfotaimg.exe [ONSemiconductor::Device.Bluetoo
mkfotaimg.py [ONSemiconductor::Device.Bluetootl
sections.ld [ONSemiconductor::Device.Startup.Start
RTE_Components.h
app.c
ble_peripheral_direction_finding.rteconfig
readme_ble_peripheral_direction_finding.md
ble_peripheral_direction_finding.rteconfig
Components
Resolve
Validation Output
Description
Components
Device and Board
Packs
Problems
Tasks
Console
Properties
Search
CDT Build Console [ble_peripheral_direction_finding]

Software Components	Sel.	Variant	Vendor	Version	Description
RSL15 Evaluation Board			ONSemiconduc		ON Semiconductor RSL15 Evaluation Board
RSL15			onsemi		ARM Cortex-M33 48 MHz, 64 KB RAM
CEM102					
CMSIS					Cortex Microcontroller Software Interface Components
CMSIS Driver					Unified Device Drivers compliant to CMSIS-Driver Specifica
Device					Startup, System Setup
Bluetooth Core (API)				1.0.0	BLE Core implementation
BLE Abstraction	☑		ONSemiconduc	1.6.222	BLE Abstraction
BLE Stack	☐	release	ONSemiconduc	1.6.222	Bluetooth Stack (libblelib)
FOTA	☑	release	ONSemiconduc	1.6.222	FOTA Library
Bluetooth Profiles					
BASC	☐	release	ONSemiconduc	1.6.222	Battery Service - Client Role
BASS	☑	release	ONSemiconduc	1.6.222	Battery Service - Server Role
CGMS	☐	release	ONSemiconduc	1.6.222	Continuous Glucose Monitoring (CGM) Profile - Server Role
DISC	☐	release	ONSemiconduc	1.6.222	Device Information Service - Client Role
DISS	☐	release	ONSemiconduc	1.6.222	Device Information Service Server
cc312					
Libraries					
DMA	☐	source	ONSemiconduc	1.6.222	DMA Driver for RSL15
Flash	☐	source	ONSemiconduc	1.6.222	Flash Source
GPIO	☐	source	ONSemiconduc	1.6.222	GPIO Driver for RSL15
HAL	☑	source	ONSemiconduc	1.6.222	HAL Source
LIN	☐	source	ONSemiconduc	1.6.222	LIN Driver for RSL15
PWM	☐	source	ONSemiconduc	1.6.222	PWM Driver for RSL15
SwmTrace	☑	swmTrace_UART_DMA	ONSemiconduc	1.6.222	SwmTrace UART DMA Non-Blocking Source (libswmTrace)
Timer	☐	source	ONSemiconduc	1.6.222	Timer Driver for RSL15
trustzone_secure_gatewa	☐	release	ONSemiconduc	1.6.222	Trustzone Secure Gateway Veneer Library (libsecure_gatewa
Startup	☑	source	ONSemiconduc	1.6.222	System Startup for ON Semiconductor RSL15
RTOS		FreeRTOS	ARM	10.5.1	FreeRTOS Real Time Kernel

Figure 16. Adding the FOTA Software Component

3. Modify the source code to set the FOTA version number and ID:
   a. In *app.h*, define your version numbers and ID:

```
        /*----------------------------------------------

* Application Version
* ------------------------------------------------------------------------------ */

        #define APP_VER_ID                  "DF"

        #define APP_VER_MAJOR               1

        #define APP_VER_MINOR               0

        #define APP_VER_REVISION            0
```

   b. In *app.c*, include *sys_fota.h* and use the `SYS_FOTA_VERSION` macro to set the version:

```
#include "sys_fota.h"

/* ------------------------------------------------------------------------------
* Application Version
* ------------------------------------------------------------------------- */
SYS_FOTA_VERSION(APP_VER_ID, APP_VER_MAJOR, APP_VER_MINOR, APP_VER_REVISION);
```

4. Modify the sample application to add random and seed initialization functions in *app.c*. These functions are initially defined in *ble_protocol_support.c*, which is excluded for the FOTA stack, as shown in the "Adding the FOTA Software Component" figure (Figure 16).

```
void srand_func(uint32_t seed)
{
        srand(seed);

}


int rand_func(void)
{
        return rand();

}
```

5. Modify the sample application to activate the DFU when the FOTA_GPIO on the RSL EVB is connected to ground.
    a. In *app.h*, define an appropriate GPIO as FOTA_GPIO. For example:

```
        #define FOTA_GPIO                     1
```

    b. In *app_init.c*, modify the `DeviceInit()` function to configure GPIO1 as a GPIO input:

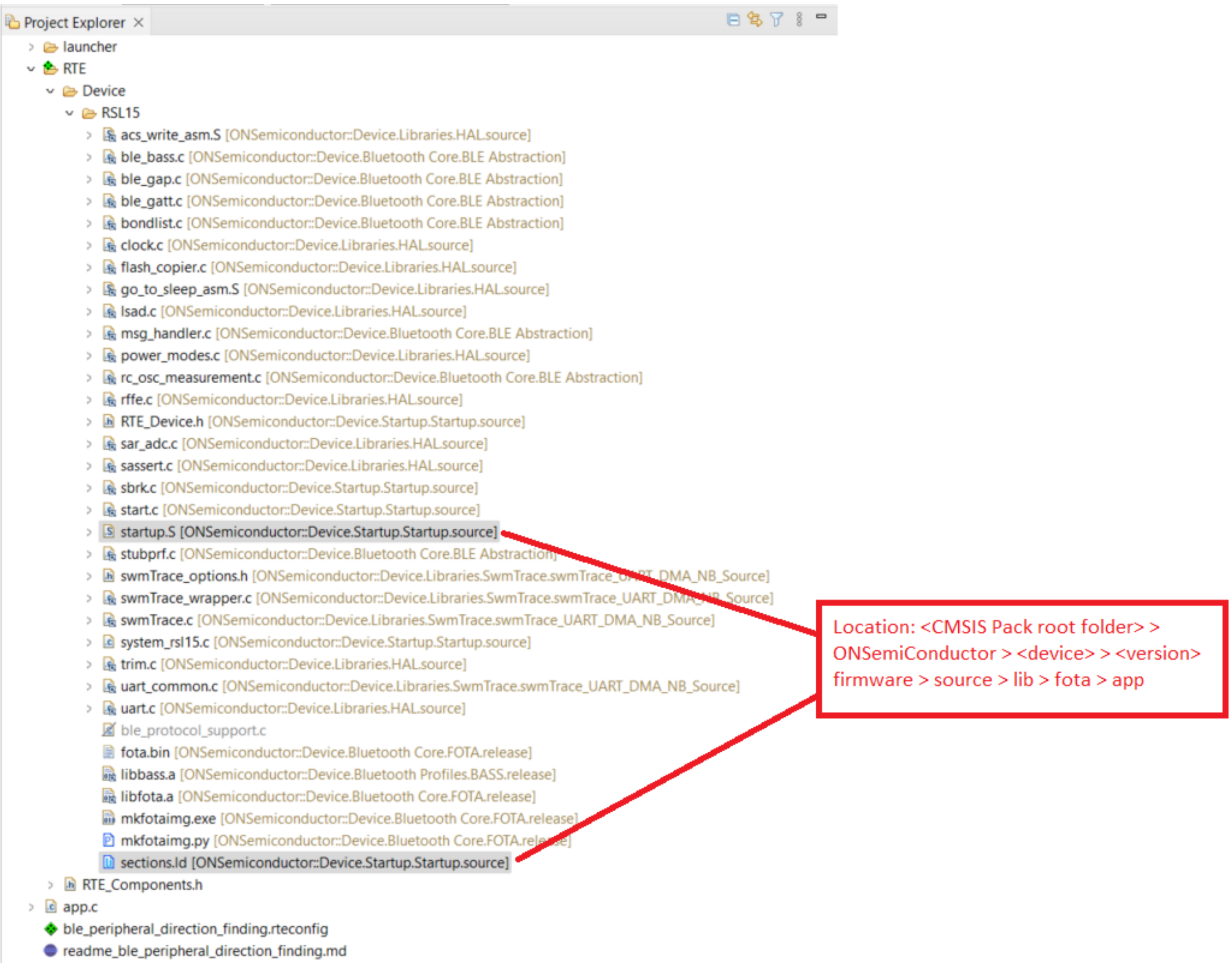
```
void DeviceInit(void)
{
...
    /* Configure FOTA_GPIO as GPIO input */
SYS_GPIO_CONFIG(FOTA_GPIO, (GPIO_MODE_GPIO_IN | GPIO_LPF_DISABLE | GPIO_WEAK_PULL_UP |
GPIO_6X_DRIVE));
...
}
```

    c. In *app.c*, add the code below in the `while(1)` loop of the `main()` function, to start the DFU when the FOTA GPIO is connected to ground:

```
while (1)
{
    ...

    /* Start update when FOTA_GPIO is connected to ground */

    if ((Sys_GPIO_Read(FOTA_GPIO)) == 0)
    {
        Sys_Fota_StartDfu(1);
    }

    ...
}
```

The DFU component is activated by calling the `Sys_Fota_StartDfu(mode)` function defined in *sys_fota.h*. The mode value 0 is for an application without the Bluetooth Low Energy stack (for example, *blinky*), and 1 is for an application that uses the Bluetooth Low Energy stack (such as *ble_peripheral_server*).

6. Replace your linker script *sections.ld* file with the one that contains the FOTA placement, and also replace your *startup.S* file, as shown in the "Replacing the sections.ld and startup.S Files" figure (Figure 17). The new linker script and *startup.S* file can be copied from *ble_peripheral_server_fota*, or directly from the CMSIS-Pack root folder as follows:
    - Non-Bluetooth Low Energy Application:
        - *sections.ld* and *startup.s*: **`<CMSIS Pack root folder>`** > **ONSemiconductor** > **`<device>`** > **`<version>`** > **firmware** > **source** > **lib** > **fota** > **app** > **non_ble**
    - Bluetooth Low Energy Application:
        - *sections.ld* and *startup.s*: **`<CMSIS Pack root folder>`**> **ONSemiconductor** > **`<device>`** > **`<version>`** > **firmware** > **source** > **lib** > **fota** > **app**



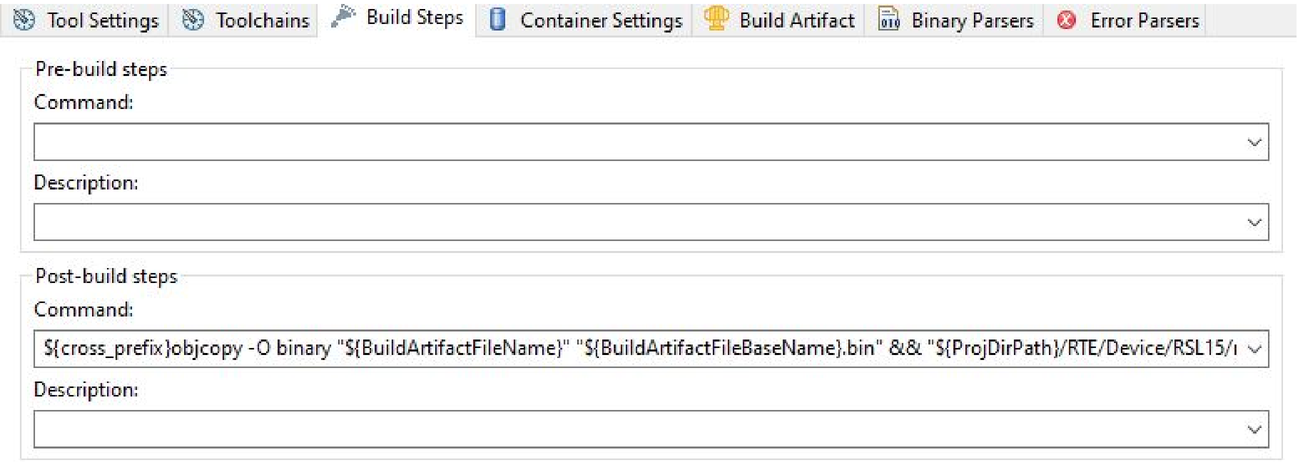
- launcher
- RTE
    - Device
        - RSL15
            - acs_write_asm.S [ONSemiconductor::Device.Libraries.HAL.source]
            - ble_bass.c [ONSemiconductor::Device.Bluetooth Core.BLE Abstraction]
            - ble_gap.c [ONSemiconductor::Device.Bluetooth Core.BLE Abstraction]
            - ble_gatt.c [ONSemiconductor::Device.Bluetooth Core.BLE Abstraction]
            - bondlist.c [ONSemiconductor::Device.Bluetooth Core.BLE Abstraction]
            - clock.c [ONSemiconductor::Device.Libraries.HAL.source]
            - flash_copier.c [ONSemiconductor::Device.Libraries.HAL.source]
            - go_to_sleep_asm.S [ONSemiconductor::Device.Libraries.HAL.source]
            - lsad.c [ONSemiconductor::Device.Libraries.HAL.source]
            - msg_handler.c [ONSemiconductor::Device.Bluetooth Core.BLE Abstraction]
            - power_modes.c [ONSemiconductor::Device.Libraries.HAL.source]
            - rc_osc_measurement.c [ONSemiconductor::Device.Bluetooth Core.BLE Abstraction]
            - rffe.c [ONSemiconductor::Device.Libraries.HAL.source]
            - RTE_Device.h [ONSemiconductor::Device.Startup.Startup.source]
            - sar_adc.c [ONSemiconductor::Device.Libraries.HAL.source]
            - sassert.c [ONSemiconductor::Device.Libraries.HAL.source]
            - sbrk.c [ONSemiconductor::Device.Startup.Startup.source]
            - start.c [ONSemiconductor::Device.Startup.Startup.source]
            - startup.S [ONSemiconductor::Device.Startup.Startup.source]
            - stubprf.c [ONSemiconductor::Device.Bluetooth Core.BLE Abstraction]
            - swmTrace_options.h [ONSemiconductor::Device.Libraries.SwmTrace.swmTrace_UART_DMA_NB_Source]
            - swmTrace_wrapper.c [ONSemiconductor::Device.Libraries.SwmTrace.swmTrace_UART_DMA_NB_Source]
            - swmTrace.c [ONSemiconductor::Device.Libraries.SwmTrace.swmTrace_UART_DMA_NB_Source]
            - system_rsl15.c [ONSemiconductor::Device.Startup.Startup.source]
            - trim.c [ONSemiconductor::Device.Libraries.HAL.source]
            - uart_common.c [ONSemiconductor::Device.Libraries.SwmTrace.swmTrace_UART_DMA_NB_Source]
            - uart.c [ONSemiconductor::Device.Libraries.HAL.source]
            - ble_protocol_support.c
            - fota.bin [ONSemiconductor::Device.Bluetooth Core.FOTA.release]
            - libbass.a [ONSemiconductor::Device.Bluetooth Profiles.BASS.release]
            - libfota.a [ONSemiconductor::Device.Bluetooth Core.FOTA.release]
            - mkfotaimg.exe [ONSemiconductor::Device.Bluetooth Core.FOTA.release]
            - mkfotaimg.py [ONSemiconductor::Device.Bluetooth Core.FOTA.release]
            - sections.ld [ONSemiconductor::Device.Startup.Startup.source]
    - RTE_Components.h
- app.c
- ble_peripheral_direction_finding.rteconfig
- readme_ble_peripheral_direction_finding.md

Location: <CMSIS Pack root folder> > ONSemiConductor > <device> > <version> firmware > source > lib > fota > app

Figure 17. Replacing the sections.ld and startup.S Files

7. Modify the project post-build steps (see the "Post-Build Steps" figure (Figure 18)) to generate the FOTA image in both Debug and Release build configurations, by going to **Project** > **Properties** > **C/C++ Build** > **Settings** > **Build Steps** > **Post-build steps**. Two steps are required to generate the *.fota* image. First, we need to generate the application sub-image *.bin* file using *objcopy*. Then, we use the *mkfotaimg.exe* tool to generate the FOTA image (*.fota* file). Both commands can be concatenated with `&&` and added to the post-build steps as in the example that follows. To use this command, copy the code from the **Post-Build steps** of the existing sample application *ble_peripheral_server_fota*, as the code in the text below is not directly copyable.

```
  ${cross_prefix}objcopy -O binary "${BuildArtifactFileName}"
"${BuildArtifactFileBaseName}.bin"
&& "${ProjDirPath}/RTE/Device/RSL15/mkfotaimg.exe" -o
"${BuildArtifactFileBaseName}.fota"
"${ProjDirPath}/RTE/Device/RSL15/fota.bin" "${BuildArtifactFileBaseName}.bin"
```



```
${cross_prefix}objcopy -O binary "${BuildArtifactFileName}" "${BuildArtifactFileBaseName}.bin" && "${ProjDirPath}/RTE/Device/RSL15/
```

Figure 18. Post-Build Steps

8. Build the *ble_peripheral_direction_finding* application. If no error occurs, the FOTA image (*ble_peripheral_direction_finding.fota*) can be found in the *Debug* or *Release* folder.

8.2 PERFORMING A FOTA UPDATE

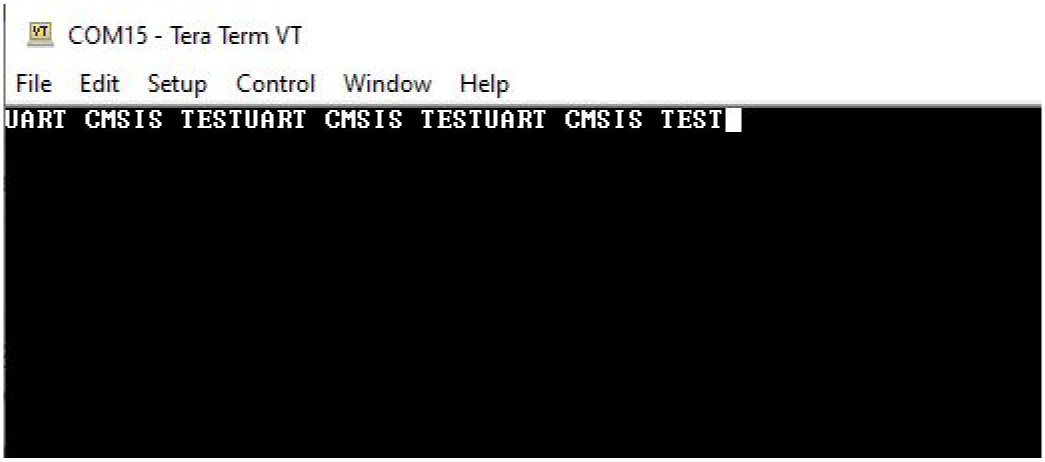
1. Make sure you have *bootloader* running on your RSL EVB. If you have executed the steps in Chapter 5 "Performing Your First FOTA Update" on page 18, you already have *bootloader* flashed on your board and you can skip this step. Otherwise, refer to Section 5.2 “Setting Up the Bootloader and Loading a Firmware Image Using UART” on page 20 for instructions.
2. Activate the *bootloader* updater mode: Connect the UPDATE_GPIO from the *bootloader* application to ground on the RSL EVB, and then push the reset button. The bootloader updater is now active, waiting for a firmware image over UART.

    NOTE: If there is a valid user application in flash (for example, the *ble_peripheral_server_fota*) and no command is received over UART after 30 seconds, the *bootloader* updater times out and reboots into the user application.
3. Load the *ble_peripheral_direction_finding.fota* image using the *bootloader* with the following command:

```
> python updater.py COM3 ble_peripheral_direction_finding.fota
```

```
Image       : DF ver=1.0.0 / FOTA   ver=1.0.0
Application: FOTA ver=1.0.0
Bootloader : BOOTL* ver=1.0.0
*********************************************************************************
```

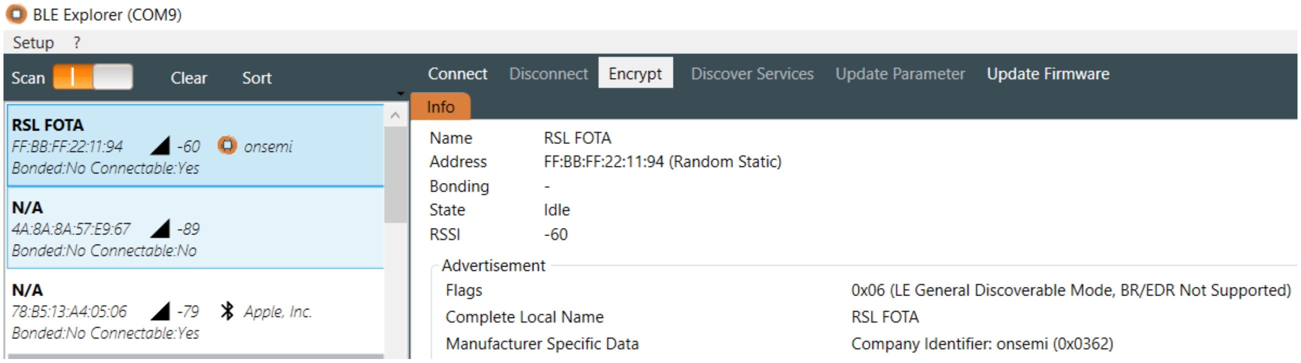
After loading the image, the *bootloader* resets the device and boots up the user application.



```
UART CMSIS TESTUART CMSIS TESTUART CMSIS TEST
```

Figure 19. Application Running

4. Connect the FOTA_GPIO (GPIO1) to ground so that the device starts the FOTA DFU mode. You can see the name **RSL FOTA** on the RSL BLE Explorer. After checking that the name is **RSL FOTA**, disconnect FOTA_GPIO from ground, as in the "FOTA DFU Mode Running" figure (Figure 20), below:



RSL FOTA
FF:BB:FF:22:11:94 -60 onsemi
Bonded:No Connectable:Yes

N/A
4A:8A:8A:57:E9:67 -89
Bonded:No Connectable:No

N/A
78:B5:13:A4:05:06 -79 Apple, Inc.
Bonded:No Connectable:Yes

Name	RSL FOTA
Address	FF:BB:FF:22:11:94 (Random Static)
Bonding	-
State	Idle
RSSI	-60

Advertisement

Flags	0x06 (LE General Discoverable Mode, BR/EDR Not Supported)
Complete Local Name	RSL FOTA
Manufacturer Specific Data	Company Identifier: onsemi (0x0362)

Figure 20. FOTA DFU Mode Running

Note: after 30 seconds, the FOTA DFU Mode times out and reboots into the user application.

5. Within 30 seconds of activating the FOTA DFU Mode, Click on the **Update Firmware** option on the BLE Explorer and select the **ble_peripheral_direction_finding.fota** image file.
6. You can repeat these steps to update the firmware with the *ble_peripheral_server_fota.fota* image as well. The steps described here can be applied to any Bluetooth Low Energy sample application.

CHAPTER 9

The Secure Bluetooth Low Energy FOTA Application

The *secure_blinky_fota* application shows how the firmware of a secure non-Bluetooth Low Energy application is updated by FOTA and how to sign the firmware. Here you can learn how to create a secure Bluetooth Low Energy FOTA application and how to update the application's firmware via FOTA.

9.1 WORKING WITH BLE_PERIPHERAL_SERVER_FOTA

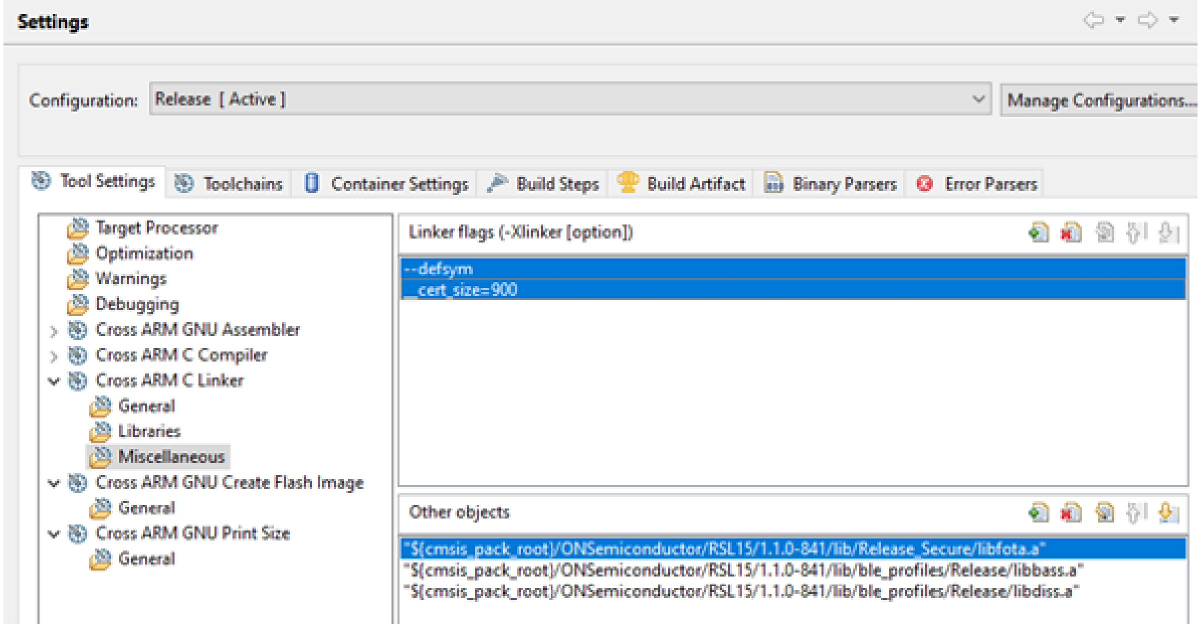
Before starting to create a secure Bluetooth Low Energy FOTA application, you must have a provisioned RSL15 Evaluation and Development Board and the BLE Explorer utility.

This section is divided into four steps: of generating the application's firmware image, the signing process, making FOTA image, and testing.

9.1.1 Step 1: Generate the application’s firmware image

To generate the firmware image, perform these steps:

1. Import the *secure_bootloader* and build with the **bl_D_Secure** configuration option. After building successfully, *secure_bootloader.hex* is generated under the *bl_D_Secure* folder.
2. Import the FOTA project. Navigate to **File** > **Import** > **General** > **Existing Projects into Workspace**, set the **Select root directory:** option with the path to the source code **`<cmsis_pack_root>`***/ONSemiconductor/***`<device>`***/***`<version>`***/firmware/source/lib/fota*, mark the checkbox **Copy projects into workspace**, and click **Finish**. The project appears in the left side of the Project Explorer view. Build the project with the **Release_Secure** configuration option. After building successfully, *libfota.a*, *fota.bin* and *fota.hex* are generated under the *Release_Secure* folder and replace the *libfota.a* and the *fota.bin* files in your CMSIS-Pack installation at **`<cmsis_pack_root>`***/ONSemiconductor/***`<device>`***/***`<version>`***/lib/Release_Secure.*
3. Import the *ble_peripheral_server_fota*, update the configurations, and build, using these steps:
   a. Add `-defsym` and `__cert_size=900` in the **Linker flags** in miscellaneous of **Cross ARM C Linker**.
      NOTE: The `__cert_size` of 900 is calculated from the content certificate size, which is 868, + 32 bytes of padding. This is the minimum size required.
      If you want to use key certificates, you must add an additional 840 to `__cert_size` for each key certificate being used. For example, if you want to use one key certificate, then __cert_size = 868 (content certificate) + 32 (byte padding) + 840 (`key1` certificate size) = 1740.
   b. Change the path of *libfota.a* to use the Secure FOTA library `${cmsis_pack_root}/ONSemiconductor/<device>/<version>/lib/Release_Secure/libfota.a` (see the "Updating Secure Configuration" figure (Figure 21)).



Configuration: Release [Active]

Linker flags (-Xlinker [option])

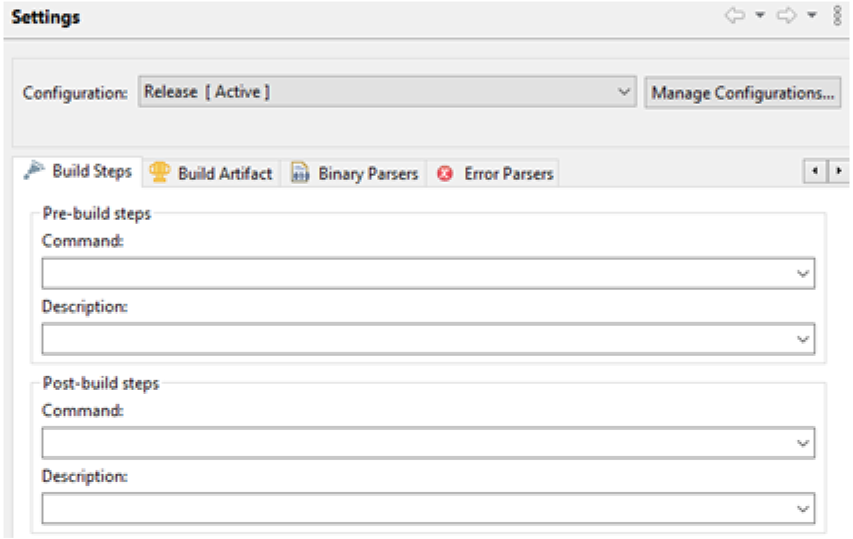
```
--defsym
__cert_size=900
```

Other objects

```
"${cmsis_pack_root}/ONSemiconductor/RSL15/1.1.0-841/lib/Release_Secure/libfota.a"
"${cmsis_pack_root}/ONSemiconductor/RSL15/1.1.0-841/lib/ble_profiles/Release/libbass.a"
"${cmsis_pack_root}/ONSemiconductor/RSL15/1.1.0-841/lib/ble_profiles/Release/libdiss.a"
```

Figure 21. Updating Secure Configuration

c. Delete the *fota.bin* and *libfota.a* files under **RTE** > **Device** > **RSL15**, and copy the *fota.bin* and *libfota.a* files from **`<cmsis_pack_root>`***/ONSemiconductor/***`<device>`***/***`<version>`***/lib/Release_Secure.*
d. Delete **Command** in **Post-build steps** (see the "Deleting Command in Post-Build Steps" figure (Figure 22)).



Configuration: Release [Active]

Figure 22. Deleting Command in Post-Build Steps

e. Build the *ble_peripheral_server_fota*. After building successfully, *ble_peripheral_server_fota.hex* is generated under the *Release* folder.

9.1.2 Step 2: Sign the Application's Firmware Image

This step assumes your RSL15 workspace has a directory containing RSLSec which has sub-directories of *assets/keys/hbk0* and *assets/keys/hbk1*. You should have your own hbk0 and hbk1 keys. To sign the firmware image, run a command prompt and go to the directory where RSLSec is located. Then do the following:

1. Generate RSA keys and certificate.
    a. Create RSA keys to be used when creating the three-certificate RoT chain:

```
 rslsec trust make hbk1 ./assets/keys/hbk1_key_1
 rslsec trust make hbk1 ./assets/keys/hbk1_key_2
```

    b. Create the key certificate:

```
 mkdir .\assets\cert\hbk1
rslsec trust cert key --out ./assets/cert/hbk1/key_1.crt --hbk hbk1 --keypair
./assets/keys/hbk1/hbk1.prv.pem --pwd ./assets/keys/hbk1/hbk1.pwd --pubkey
./assets/keys/hbk1_key_1/hbk1_key_1.pub.pem
```

```
 rslsec trust cert key --out ./assets/cert/hbk1/key_2.crt --hbk hbk1 --
keypair ./assets/keys/hbk1_key_1/hbk1_key_1.prv.pem --pwd ./assets/keys/hbk1_
key_1/hbk1_key_1.pwd --pubkey ./assets/keys/hbk1_key_2/hbk1_key_2.pub.pem
```

2. Create the content certificate for the *secure_bootloader* application.
    a. Create the content certificate:

```
 rslsec trust cert content --out ./assets/cert/hbk1/content.crt --keypair
./assets/keys/hbk1_key_2/hbk1_key_2.prv.pem --pwd ./assets/keys/hbk1_key_
2/hbk1_key_2.pwd --image ../secure_bootloader/bl_D_Secure/secure_
bootloader.hex --key1 --key2 --target RSL15
```

    b. Pack the signed secure bootloader image with the certificates:

```
 mkdir .\assets\apps\RSL15\hbk1

rslsec trust pack --out ./assets/apps/RSL15/hbk1/signed_secure_bootloader.hex
--key1 ./assets/cert/hbk1/key_1.crt --key2 ./assets/cert/hbk1/key_2.crt --
content ./assets/cert/hbk1/content.crt --image ../secure_bootloader/bl_D_
Secure/secure_bootloader.hex --target RSL15 --firstKeyAddress 0 --
secondKeyAddress 0 --contentAddress 0
```

    The *signed_secure_bootloader.hex* file is created in the directory.

3. Create the content certificate for the FOTA.
    a. Create the content certificate:

```
 rslsec trust cert content --out ./assets/cert/hbk1/content.crt --keypair
./assets/keys/hbk1_key_2/hbk1_key_2.prv.pem --pwd ./assets/keys/hbk1_key_
2/hbk1_key_2.pwd --image ../fota/Release_Secure/fota.hextarget RSL15
```

    b. Pack the signed secure FOTA image with the certificates:

```
 rslsec trust pack --out ./assets/apps/RSL15/hbk1/signed_fota.hex --bin
./assets/apps/RSL15/hbk1/signed_fota.bin --content
./assets/cert/hbk1/content.crt --image ../fota/Release_Secure/fota.hex --
target RSL15 --contentAddress 0 --config 0
```

    The *signed_fota.hex* and *signed_fota.bin* files are created in the directory.

4. Create the content certificate for the *ble_peripheral_server_fota* application.
    a. Create the content certificate:

```
 rslsec trust cert content --out ./assets/cert/hbk1/content.crt --keypair
./assets/keys/hbk1_key_2/hbk1_key_2.prv.pem --pwd ./assets/keys/hbk1_key_
2/hbk1_key_2.pwd --image ../ble_peripheral_server_fota/Release/ble_
peripheral_server_fota.hex --target RSL15
```

    b. Pack the signed *ble_peripheral_server_fota* image with the certificates:

```
 rslsec trust pack --out ./assets/apps/RSL15/hbk1/signed_ble_peripheral_
server_fota.hex --bin ./assets/apps/RSL15/hbk1/signed_ble_peripheral_server_
fota.bin --content ./assets/cert/hbk1/content.crt --image ../ble_peripheral_
server_fota/Release/ble_peripheral_server_fota.hex --target RSL15 --
contentAddress 0 --config 0
```

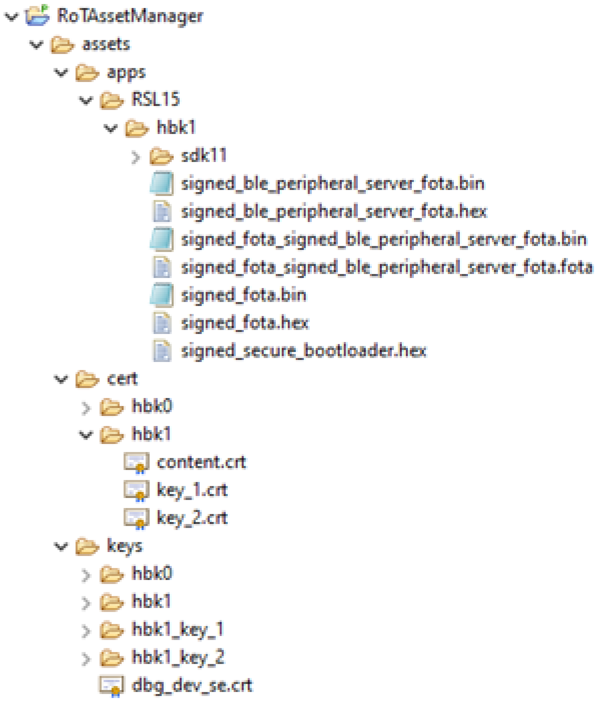
    The *signed_ble_peripheral_server_fota.hex* and *signed_ble_peripheral_server_fota.bin* files are created in the directory.

9.1.3 Step 3: Create a FOTA Image

To create the FOTA image, run *mkfotaimg.exe* (located in `${<cmsis_pack_root>}/ONSemiconductor/<device>/<version>/firmware/source/lib/fota/tools/mkfotaimg.exe`), with *signed_fota.bin* and *signed_ble_peripheral_server_fota.bin*, to create the FOTA image file:

```
 mkfotaimg.exe -o ./assets/apps/RSL15/hbk1/signed_fota_signed_ble_peripheral_server_
fota.fota ./assets/apps/RSL15/hbk1/signed_fota.bin ./assets/apps/RSL15/hbk1/signed_ble_
peripheral_server_fota.bin -s 0xD800
```

The *signed_fota_signed_ble_peripheral_server_fota.fota* file is created in the directory.



- RoTAssetManager
  - assets
    - apps
      - RSL15
        - hbk1
          - sdk11
          - signed_ble_peripheral_server_fota.bin
          - signed_ble_peripheral_server_fota.hex
          - signed_fota_signed_ble_peripheral_server_fota.bin
          - signed_fota_signed_ble_peripheral_server_fota.fota
          - signed_fota.bin
          - signed_fota.hex
          - signed_secure_bootloader.hex
    - cert
      - hbk0
      - hbk1
        - content.crt
        - key_1.crt
        - key_2.crt
    - keys
      - hbk0
      - hbk1
      - hbk1_key_1
      - hbk1_key_2
      - dbg_dev_se.crt

Figure 23. Secure Keys and Certificate and Signed Firmware Images

9.1.4 Step 4: Test the Secure ble_peripheral_server_fota Application

To test the application, perform these steps:

1. Open J-link Commander and connect to RSL15. Load the *signed_secure_bootloader.hex* file into the provisioned device:

```
J-Link>loadfile <path_to_hexfile>\signed_secure_bootloader.hex
```

2. After copying *signed_fota_signed_ble_peripheral_server_fota.fota* and renaming the image to *.bin*, load the *signed_fota_signed_ble_peripheral_server_fota.bin* file into the provisioned device:

```
 J-Link>loadbin <path_to_binary>\signed_fota_signed_ble_peripheral_server_fota.bin
0x10D800
```

   You can see the LED blinking on your EVB. Your device is now ready to perform a FOTA update.

3. Updating the application
   a. Run the BLE Explorer utility.
   b. BLE Explorer scans for *ble_peripheral_server_fota.*
   c. Connect `FOTA_GPIO` to ground on your EVB to start FOTA DFU mode.
   d. You can see the name RSL FOTA in the BLE Explorer window.
   e. Connect to the name RSL FOTA.
4. Click on the **Update Firmware** option in the BLE Explorer window, and select the *signed_fota_signed_secure_ble_peripheral_server_fota.fota* image file.

CHAPTER 10

RSL FOTA Mobile Application

RSL FOTA is a simple mobile application for iOS and Android, created to demonstrate Firmware-Over-The-Air (FOTA) for onsemi RSL Bluetooth Low Energy devices. The RSL FOTA application acts as a central device to scan, connect and transmit the firmware image to a remote device. The remote RSL device firmware must have FOTA-enabled firmware to receive the FOTA firmware image.

10.1 RSL FOTA APPLICATION

Follow these steps to use the RSL FOTA Mobile Application:

1. Download and install the RSL FOTA application from the Google Play Store or the Apple App Store.
2. Launch the application and select the FOTA enabled firmware device from the list of devices, as shown below in the "RSL FOTA Mobile Application Showing Bluetooth Low Energy Devices" figure (Figure 24).

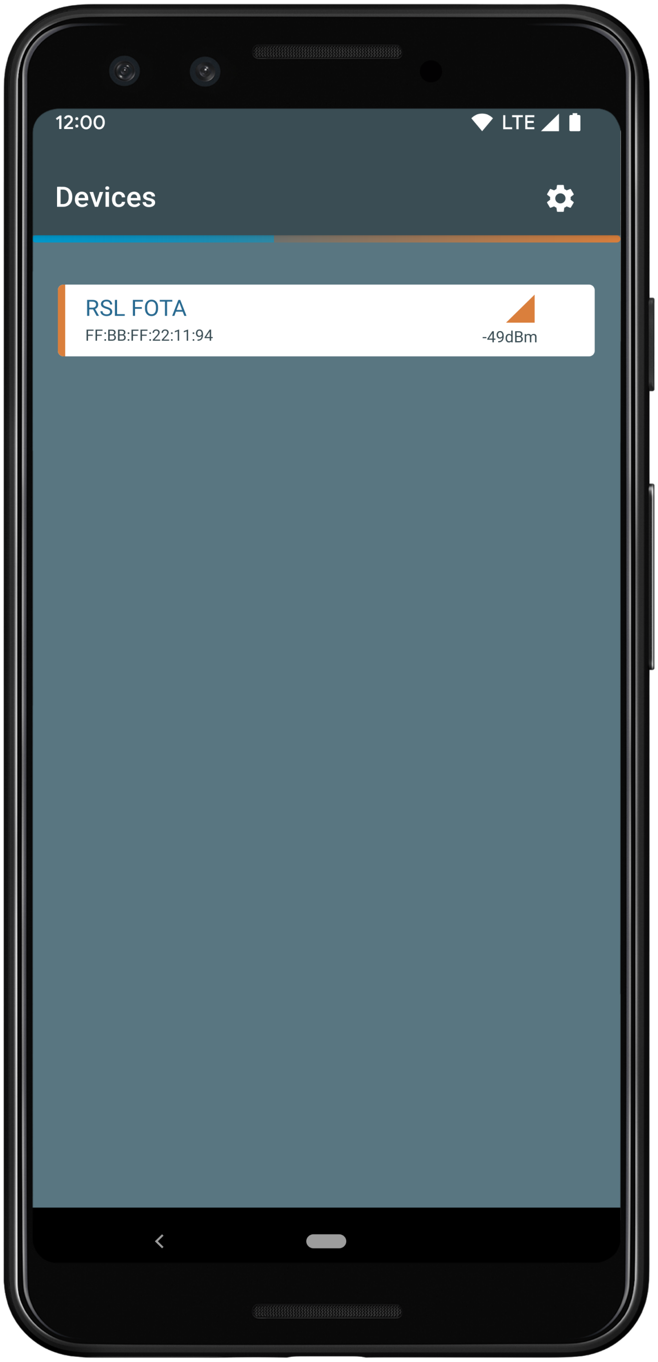
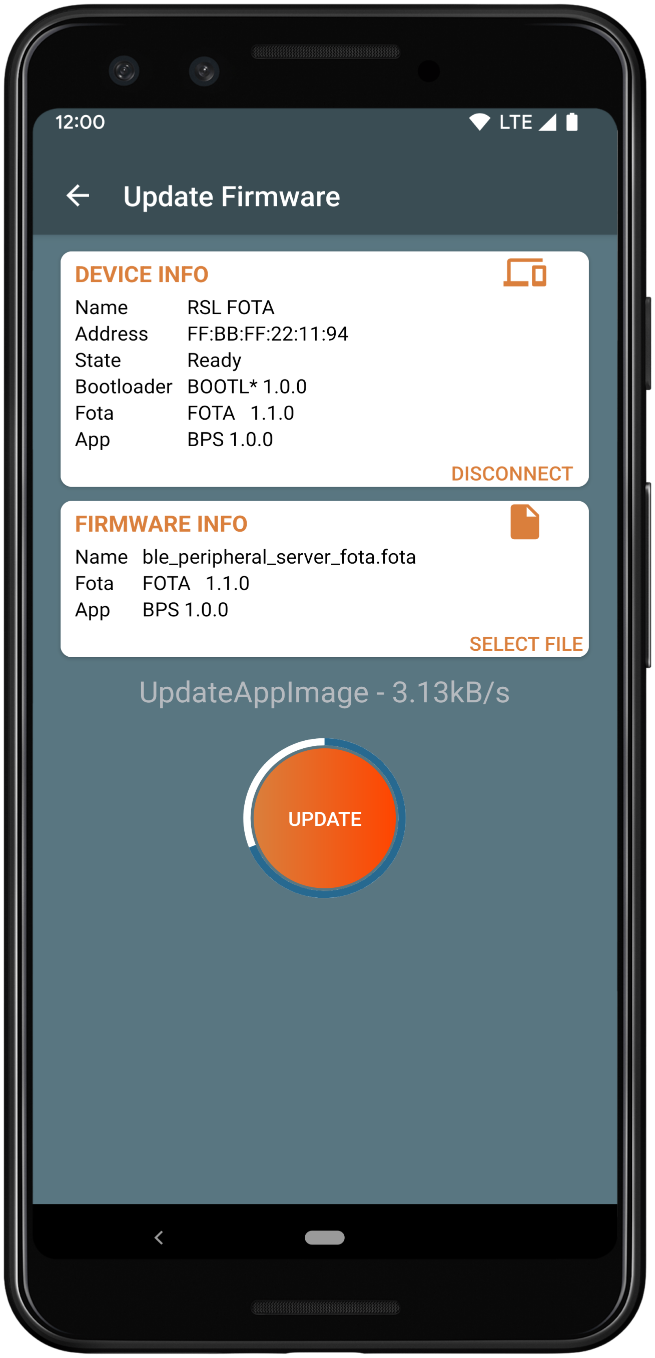


Figure 24. RSL FOTA Mobile Application Showing Bluetooth Low Energy Devices

3. When the appropriate device is selected, select the firmware image by clicking the **Select File** button.
4. Once the FOTA firmware image file is selected, click the **Update** button (see the "RSL FOTA Mobile Application: Updating Firmware" figure (Figure 25)).



DEVICE INFO

Name RSL FOTA
Address FF:BB:FF:22:11:94
State Ready
Bootloader BOOTL* 1.0.0
Fota FOTA 1.1.0
App BPS 1.0.0

FIRMWARE INFO

Name ble_peripheral_server_fota.fota
Fota FOTA 1.1.0
App BPS 1.0.0

UpdateAppImage - 3.13kB/s

Figure 25. RSL FOTA Mobile Application: Updating Firmware

5. Upon firmware update completion, **Update finished with code: 0 (Success)** is displayed.

10.2 RSL FOTA ANDROID LIMITATIONS

Note the following important details when using the RSL FOTA Mobile Application:

- Android must be version 6.0 or higher.
- Device location permission is needed to scan for Bluetooth Low Energy devices. If the permission is not granted at app startup, the app is prevented from finding any Bluetooth Low Energy devices.
- Device storage permission is needed to select a FOTA file.
- The FOTA file can be selected from the *Downloads* folder. The file can be transferred to the device, either over USB or by email.

10.3 RSL FOTA IOS LIMITATIONS

Note the following important details when using the RSL FOTA iOS® Mobile Application:

- The RSL FOTA Mobile Application for iOS requires the Bluetooth Low Energy feature to be enabled to scan for Bluetooth Low Energy devices.
- The easiest way to add a new FOTA file to the RSL FOTA Mobile Application is to send an email to the device with the file attached. After downloading the file from the email, press the **File** icon once more, and a popup is displayed where the user can select which app to use to open the file. Select the **RSL FOTA Mobile Application**. The file is then imported to the application, and is visible when the user presses the **Select File** button on the **Update Firmware** screen.

PUBLICATION ORDERING INFORMATION

LITERATURE FULFILLMENT:
Literature Distribution Center for onsemi
19521 E. 32nd Pkwy, Aurora, Colorado 80011 USA
Phone: 303-675-2175 or 800-344-3860 Toll Free USA/Canada
Fax: 303-675-2176 or 800-344-3867 Toll Free USA/Canada
Email: orderlit@onsemi.com

N. American Technical Support:
800-282-9855 Toll Free USA/Canada
Europe, Middle East and Africa Technical Support: Phone: 421 33 790 2910

onsemi Website: www.onsemi.com

Order Literature: http://www.onsemi.com/orderlit

For additional information, please contact your local Sales Representative