GR5xx DFU Development Application Note

Version: 1.6

Release Date: 2024-09-24

Shenzhen Goodix Technology Co., Ltd.

Trademarks and Permissions

GOODiX and other Goodix trademarks are trademarks of Shenzhen Goodix Technology Co., Ltd. All other trademarks and trade names mentioned in this document are the property of their respective holders.

Disclaimer

Information contained in this document is intended for your convenience only and is subject to change without prior notice. It is your responsibility to ensure its application complies with technical specifications.

Shenzhen Goodix Technology Co., Ltd. (hereafter referred to as “Goodix”) makes no representation or guarantee for this information, express or implied, oral or written, statutory or otherwise, including but not limited to representation or guarantee for its application, quality, performance, merchantability or fitness for a particular purpose. Goodix shall assume no responsibility for this information and relevant consequences arising out of the use of such information.

Without written consent of Goodix, it is prohibited to use Goodix products as critical components in any life support system. Under the protection of Goodix intellectual property rights, no license may be transferred implicitly or by any other means.

Shenzhen Goodix Technology Co., Ltd.

Headquarters: Floor 13, Phase B, Tengfei Industrial Building, Futian Free Trade Zone, Shenzhen, China

TEL: +86-755-33338828 Zip Code: 518000

Website: www.goodix.com

Preface

Purpose

This document introduces Device Firmware Upgrade (DFU) principles, GR5xx DFU schemes, App bootloader applications, and firmware upgrade through GRToolbox (Android) App (a Goodix proprietary Bluetooth LE debugging tool) and DFU Master, to help users quickly get started with GR5xx DFU schemes.

Audience

This document is intended for:

- Device user
- Developer
- Test engineer
- Technical support engineer

Release Notes

This document is the seventh release of *GR5xx DFU Development Application Note*, corresponding to Bluetooth Low Energy (Bluetooth LE) GR5xx System-on-Chip (SoC) series.

Revision History

Version	Date	Description
1.0	2023-04-20	Initial release
1.1	2023-05-10	Updated the sections "Supported Platform" and "Cross-platform Porting of DFU Master".
1.2	2023-08-22	Updated the following sections: "Supported Platform", "Upgrade of Unencrypted and Unsigned Firmware", "External Flash Resource Upgrade", and "Upgrade via UART".
1.3	2023-09-24	Updated the section "Supported Platform".
1.4	2023-11-06	Updated the approaches for obtaining GProgrammer, GRToolbox, and GRUart.
1.5	2024-02-27	Updated the section "Supported Platform".
1.6	2024-09-24	• Updated the section "Supported Platform". • Updated the hardware configuration in "Upgrade via UART". • Updated the section "Introduction to App Bootloader Project". • Updated the sections " Data Sent from the Host" in the "Operate System Info Command", " Program Start Command", and " Program End Command".

Contents

Preface I

1 Introduction 1

1.1 DFU Communication Mode 1

1.2 DFU Working Mode 1

2 DFU Scheme Design 5

2.1 Background Dual-bank DFU Mode 5

2.1.1 Flash Layout 5

2.1.2 Firmware Download Procedure 6

2.1.3 App Bootloader Boot Procedure 7

2.2 Non-background Single-bank DFU Mode 8

2.2.1 Flash Layout 9

2.2.2 Firmware Download Procedure 9

2.2.3 App Bootloader Boot Procedure 10

2.3 Comparison of Upgrade Speeds 11

2.4 Firmware Format 13

3 Introduction to App Bootloader Project 15

4 Upgrade with GRToolbox 20

4.1 Supported Platform 20

4.2 Preparation 20

4.3 Upgrade of Unencrypted and Unsigned Firmware 21

4.3.1 Firmware Configuration 21

4.3.2 Firmware Programming 22

4.3.3 Creating Target Firmware for Upgrade 23

4.3.4 To Enter DFU Interface of GRToolbox 25

4.3.5 Firmware Upgrade 27

4.3.5.1 Background dual-bank DFU mode 27

4.3.5.2 Non-background Single-bank DFU Mode 29

4.4 Upgrade of Encrypted and Signed Firmware 30

4.4.1 eFuse Setting 31

4.4.2 Download to eFuse 33

4.4.3 Firmware Configuration 34

4.4.4 Generating Encrypted and Signed Firmware 34

4.4.5 Firmware Upgrade 34

4.5 Upgrade of Signed and Unencrypted Firmware 35

4.5.1 Firmware Configuration 35

4.5.2 Generating Signed and Unencrypted Firmware 35

4.5.3 Firmware Upgrade 35

4.6 Resource Upgrade..... 36
4.6.1 Internal Flash Resource Upgrade.....36
4.6.2 External Flash Resource Upgrade..... 37

5 DFU Porting Method..... 42

6 Upgrade Through DFU Master.....50

6.1 Introduction to DFU Master.....50
6.2 Cross-platform Porting of DFU Master.....50
6.3 Instructions on Upgrade Through DFU Master.....55
6.3.1 Preparation.....55
6.3.2 Upgrade via UART.....55
6.3.3 Upgrade via Bluetooth LE.....58

7 Considerations.....60

7.1 Deinitializing Peripherals used in App Bootloader Before Jumping from App Bootloader to App Firmware.... 60
7.2 Setting the DFU Task Stack Size of Application Firmware in RTOS Environment According to Specific GR5xx SoCs..... 60

8 Appendix: DFU Communication Protocols..... 61

8.1 Basic Frame..... 61
8.1.1 Frame Structure..... 61
8.1.2 Byte Order.....61
8.2 Appendix: DFU Command Set.....61
8.2.1 Get Info Command..... 62
8.2.1.1 Data Sent from the Host.....62
8.2.1.2 Response Data from the Device..... 62
8.2.2 Operate System Info Command.....63
8.2.2.1 Data Sent from the Host.....63
8.2.2.2 Response Data from the Device..... 64
8.2.3 DFU Mode Set Command.....64
8.2.3.1 Data Sent from the Host.....64
8.2.3.2 Response Data from the Device..... 65
8.2.4 DFU Firmware Info Get Command..... 65
8.2.4.1 Data Sent from the Host.....65
8.2.4.2 Response Data from the Device..... 65
8.2.5 Program Start Command..... 66
8.2.5.1 Data Sent from the Host.....66
8.2.5.2 Response Data from the Device..... 67
8.2.6 Program Flash Command.....67
8.2.6.1 Data Sent from the Host.....68
8.2.6.2 Response Data from the Device..... 68
8.2.7 Writing Firmware in Fast Mode.....69
8.2.7.1 Data Sent from the Host.....69
8.2.7.2 Response Data from the Device..... 69

8.2.8 Program End Command....69
8.2.8.1 Data Sent from the Host....69
8.2.8.2 Response Data from the Device....70
8.2.9 Config External Flash Command....70
8.2.9.1 Data Sent from the Host....70
8.2.9.2 Response Data from the Device....71
8.2.10 Get Flash Information Command....72
8.2.10.1 Data Sent from the Host....72
8.2.10.2 Response Data from the Device....72

1 Introduction

Device Firmware Upgrade (DFU) is a boot loading mechanism to upgrade firmware for target devices, allowing developers to quickly fix defects and enrich product features.

This document focuses on DFU working principles, GR5xx DFU schemes, and firmware upgrade steps.

Before getting started, you can refer to the following documents.

Table 1-1 Reference documents

Name	Description
Developer guide of the specific GR5xx System-on-Chip (SoC)	Introduces GR5xx Software Development Kit (SDK) and how to develop and debug applications based on the SDK.
GProgrammer User Manual	Lists GProgrammer operational instructions including downloading firmware to and encrypting firmware on GR5xx SoCs.
J-Link/J-Trace User Guide	Provides J-Link operational instructions. Available at https://www.segger.com/downloads/jlink/UM08001_JLink.pdf.
Keil User Guide	Offers detailed Keil operational instructions. Available at https://www.keil.com/support/man/docs/uv4/.
Bluetooth Core Spec	Offers official Bluetooth standards and core specification from Bluetooth SIG.

1.1 DFU Communication Mode

Two DFU communication modes are supported: wireless communication and wired communication.

- Wireless communication: This mode is also known as over-the-air (OTA) DFU (commonly referred to as OTA), which means firmware upgrade is achieved through wireless communication. Wireless communication via 2G/3G/4G network, Wi-Fi, and Bluetooth can all be used for DFU.
- Wired communication: Mainly include communication through UART, USB, and SPI.

1.2 DFU Working Mode

- DFU schemes are categorized into background DFU mode and non-background DFU mode which are applicable to both wireless and wired communication modes.
  - Background DFU mode: The firmware is received by the application which can perform other tasks at the same time.
  - Non-background DFU mode: The firmware is received by Bootloader which cannot perform other tasks at the same time.
- According to the occupied storage area, DFU can also be divided into dual-bank DFU and single-bank DFU.
  - Dual-bank DFU: The received firmware is first cached in a designated area; once firmware check passes, it will be copied to the target area.

- Single-bank DFU: The received firmware is written to the target area, directly overwriting the original firmware.

For background DFU, only the dual-bank mode can be used to store the original and new firmware, whereas for non-background DFU, either single-bank mode or dual-bank mode can be used. Based on this, common DFU working modes include

- Background dual-bank DFU mode
- Non-background dual-bank DFU mode
- Non-background single-bank DFU mode

Details are as follows:

- The schematic diagram of the background dual-bank DFU mode is shown in Figure 1-1. The upgrade process mainly consists of the following steps:
  1. The application receives the firmware from the host.
  2. The application writes the received firmware to the Bank1 area.
  3. After the firmware is written, check the new firmware in the Bank1 area. After the check passes, jump to and run the bootloader firmware.
  4. Bootloader copies the new firmware to the Bank0 area, and checks the new firmware. After the check passes, jump to and run the new firmware.

Note:

The device refers to the firmware, and the host refers to the mobile App if a mobile App is used.

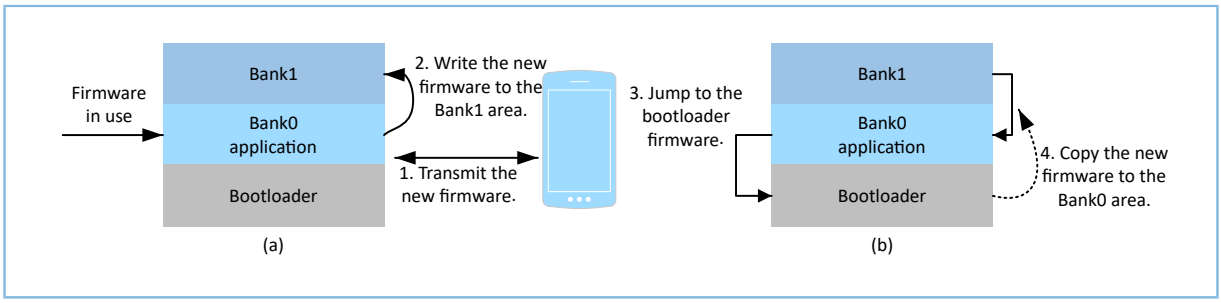

Figure 1-1 Schematic diagram of background dual-bank DFU mode

- The schematic diagram of the non-background dual-bank DFU mode is shown in Figure 1-2. The upgrade process mainly consists of the following steps:
  1. Bootloader receives the firmware from the host.
  2. Bootloader writes the received firmware to the Bank1 area.
  3. After the firmware is written, check the new firmware in the Bank1 area. After the check passes, bootloader copies the new firmware to the Bank0 area and then checks the new firmware. After the second check passes, jump to and run the new firmware.

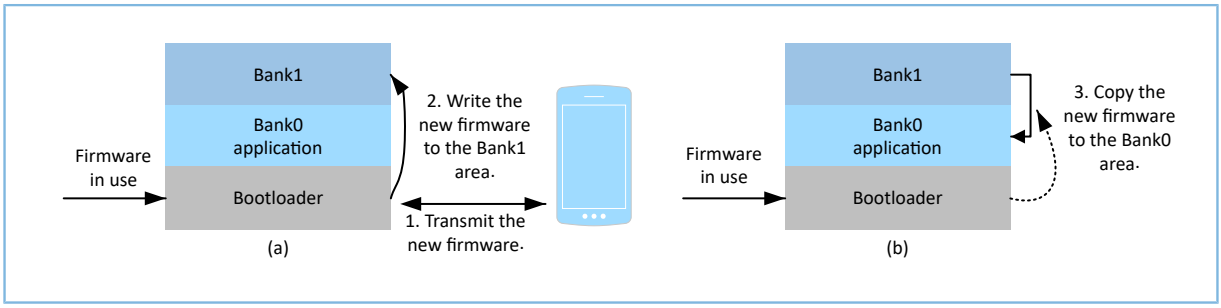

Figure 1-2 Schematic diagram of non-background dual-bank DFU mode

- The schematic diagram of the non-background single-bank DFU mode is shown in Figure 1-3. Bootloader receives the firmware from the host and writes the new firmware to the Bank0 area. After writing, check the new firmware. After the check passes, jump to and run the Bank0 application.

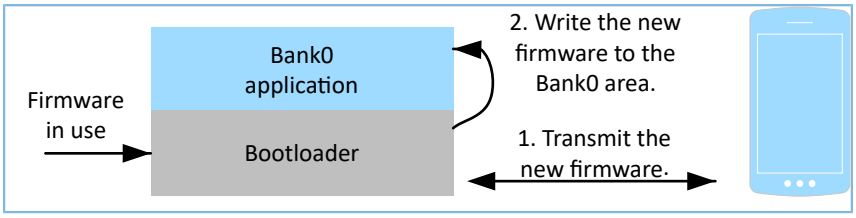

Figure 1-3 Schematic diagram of non-background single-bank DFU mode

Differences between the three DFU working modes are listed below.

Table 1-2 Differences between three DFU working modes

DFU Mode	Flash Size Required for Upgrading the Same Firmware	Backup Mechanism	Bootloader Firmware Size	Performing Other Tasks During DFU
Background dual-bank DFU mode	Large	Supported	Small	Supported
Non-background single-bank DFU mode	Small	Not supported	Large	Not supported
Non-background dual-bank DFU mode	Large	Supported	Large	Not supported

Note:

- When the same firmware is upgraded in either background dual-bank DFU mode or non-background dual-bank DFU mode, a large amount of Flash memory is required because two Flash areas are needed to store the new firmware and the original firmware, respectively.
- Backup mechanism means that when the received new firmware is damaged, the system continues running the original firmware.
- The Bootloader firmware in non-background single-bank DFU mode is smaller in size than that of the other two modes, because only the copying and jumping functionalities are needed in this mode and no data interaction with the host is required.
- In non-background single-bank DFU mode and non-background dual-bank DFU mode, jumping to bootloader firmware is required and performing other tasks during upgrade is not supported. In contrast, the background dual-bank DFU mode allows performing other tasks during upgrade, improving the user experience.

As listed in Table 1-2, the advantage of the non-background dual-bank DFU mode is also available in background dual-bank DFU mode, whereas the non-background single-bank DFU mode has a unique advantage when compared with other two modes. Based on this, GR5xx provides two DFU working modes: background dual-bank DFU mode and non-background single-bank DFU mode.

In applications, you can choose an appropriate DFU working mode according to the Flash size and the size of the firmware to be upgraded. If (Flash size–Bootloader firmware size–Parameter storage space)/2 ≥ Size of the firmware to be upgraded, the background dual-bank DFU mode is recommended; if (Flash size–Bootloader firmware size–Parameter storage space)/2 < Size of the firmware to be upgraded, the non-background single-bank DFU mode is recommended.

Note:

The **parameter storage space** refers to an area in Flash for storing non-volatile data.

2 DFU Scheme Design

This chapter elaborates on DFU schemes in background dual-bank DFU mode and non-background single-bank DFU mode of GR5xx.

2.1 Background Dual-bank DFU Mode

2.1.1 Flash Layout

Flash layout in background dual-bank DFU mode is as follows.

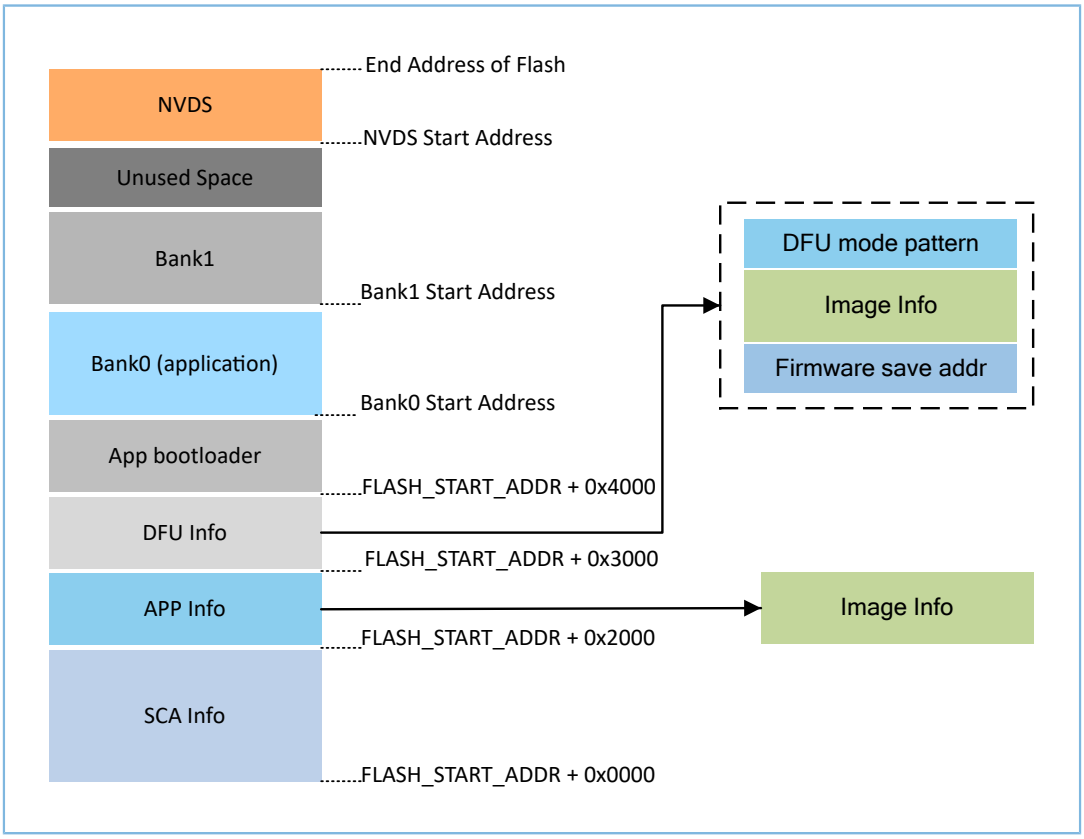

Figure 2-1 Flash layout design

- SCA Info: System Configuration Area (SCA) to store system information and App bootloader boot parameter configurations
- APP Info: application firmware info area to store the parameters for running the application firmware in the Bank0 area
- DFU Info: DFU firmware info area to store information of the new firmware in the Bank1 area
  - Firmware save addr: start address to store the new firmware
  - Image Info: parameter information of the new firmware
  - DFU mode pattern: Identify the DFU mode in use.

- App bootloader: an area that stores the App bootloader firmware and in which the firmware is running
- Bank0: an area that stores the application firmware and in which the firmware is running
- Bank1: an area that caches the new firmware; the firmware that passes the validity check will be copied to Bank0.
- Non-volatile Data Storage (NVDS): non-volatile data storage area

2.1.2 Firmware Download Procedure

As shown in Figure 2-2, receiving the firmware from the host in background dual-bank DFU mode requires running the application in the Bank0 area.

1. The Bank0 application receives the firmware from the host.
2. The Bank0 application writes the received firmware to the Bank1 area.
3. After firmware is written, the Bank0 application updates the new firmware information (Image Info in Figure 2-1), the start address to store the new firmware (Firmware save addr in Figure 2-1), and the current DFU mode (DFU mode pattern in Figure 2-1) to the DFU Info area.
4. Complete the update in the DFU Info area, and reset the device.
5. After the device is reset, run the App bootloader firmware according to the boot procedure in Figure 2-3.

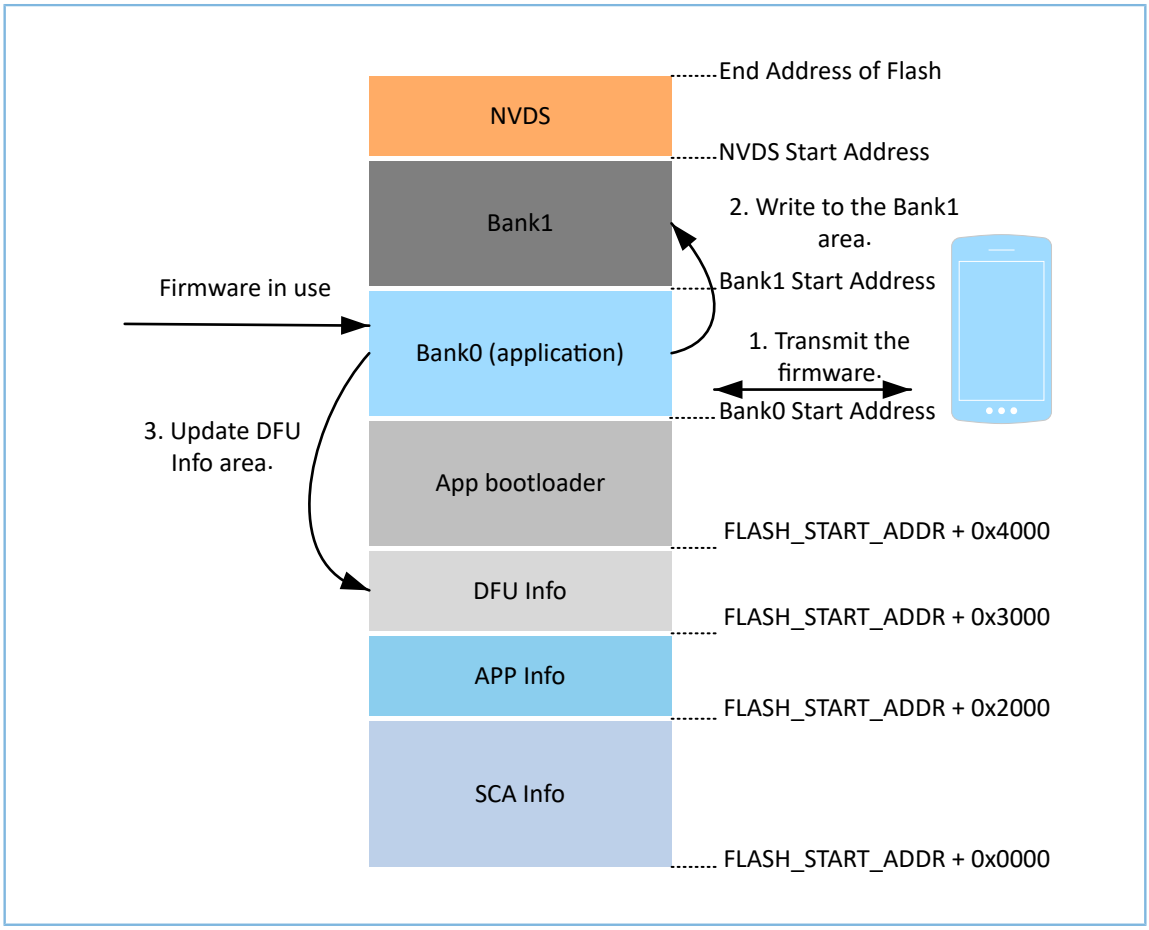

Figure 2-2 Firmware download procedure

2.1.3 App Bootloader Boot Procedure

After the device is reset, App bootloader firmware performs firmware copy and check according to the information in the DFU Info area updated by the Bank0 application, and then jumps to and runs the application firmware. The detailed boot procedure is as follows.

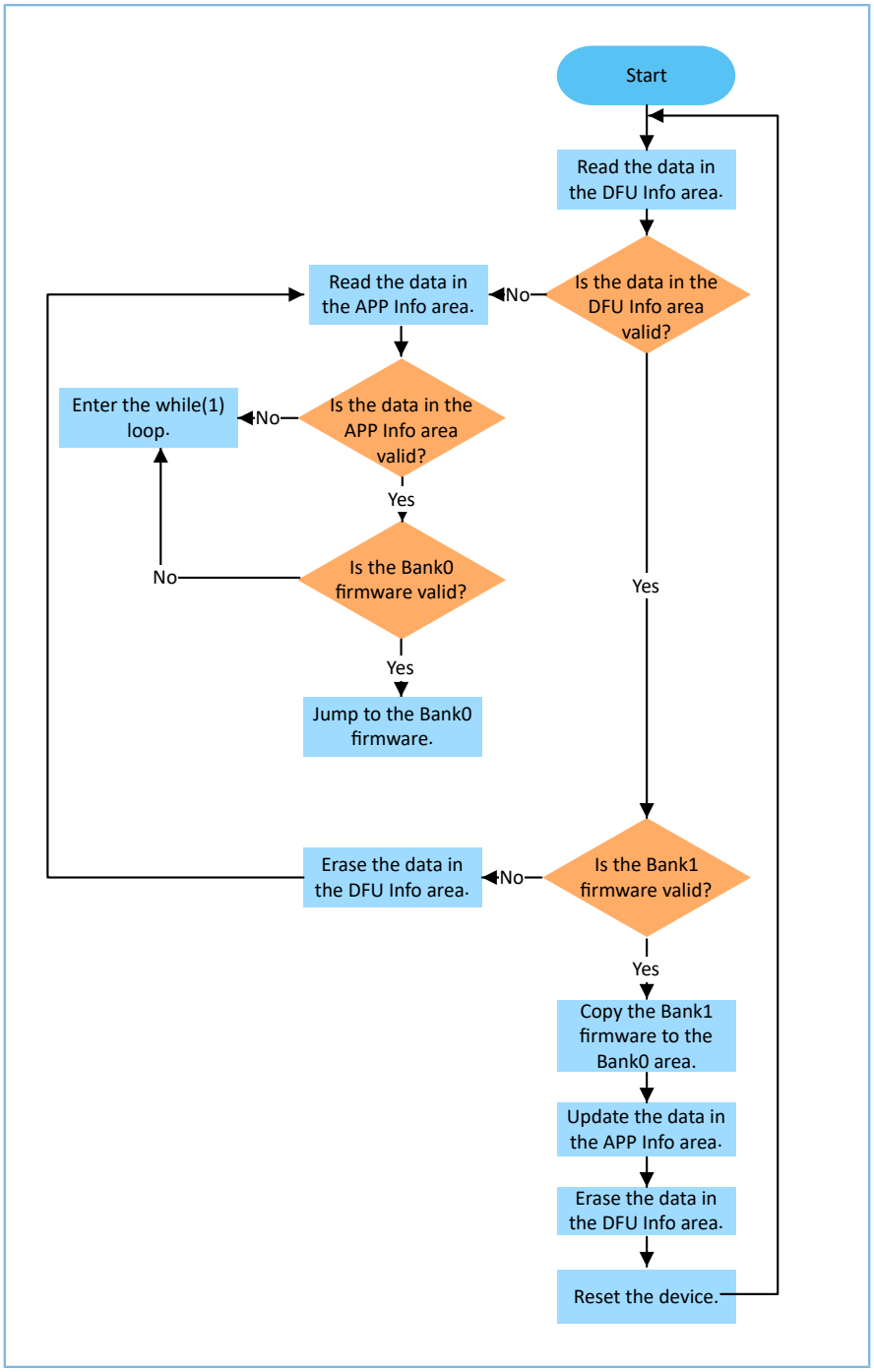

Figure 2-3 App bootloader boot procedure

Note:

- "Is the data in the DFU Info area valid?" refers to whether there is any data in the DFU Info area.
- "Is the data in the APP Info area valid?" refers to whether there is any data in the APP Info area.

2.2 Non-background Single-bank DFU Mode

2.2.1 Flash Layout

Flash layout in non-background single-bank DFU mode is as follows.

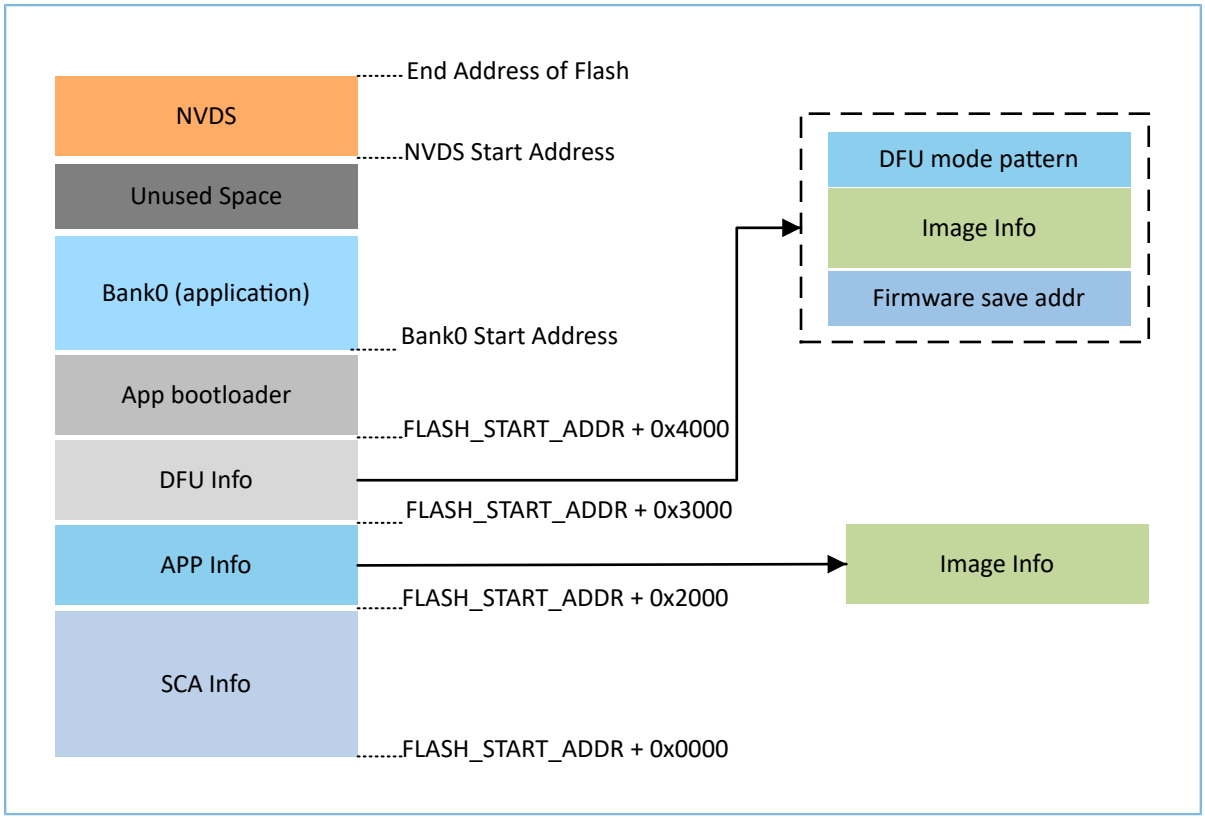

Figure 2-4 Flash layout

Flash layout in this mode has no Bank1 area, which is the only difference between the two modes. For detailed descriptions about Flash areas, refer to "Section 2.1.1 Flash Layout".

2.2.2 Firmware Download Procedure

Two scenarios are supported in this mode: with/without the Bank0 firmware.

- For the scenario with the Bank0 firmware: The firmware download procedure is shown as follows.

  1. The application in the Bank0 firmware is running. It receives the upgrade mode command issued by the host.
  2. The Bank0 firmware receives the non-background single-bank DFU mode command from the host, then writes the mode to the DFU Info area, and finally resets the device.
  3. After the device is reset, the application in the App bootloader firmware is running. It receives the firmware from the host.
  4. The App bootloader firmware writes the new firmware to the Bank0 area, then writes the new firmware information to the APP Info area and erases the data in the DFU Info area, and finally resets the device.

  After the device is reset, run the new firmware in the Bank0 area according to the boot procedure in Figure 2-6.

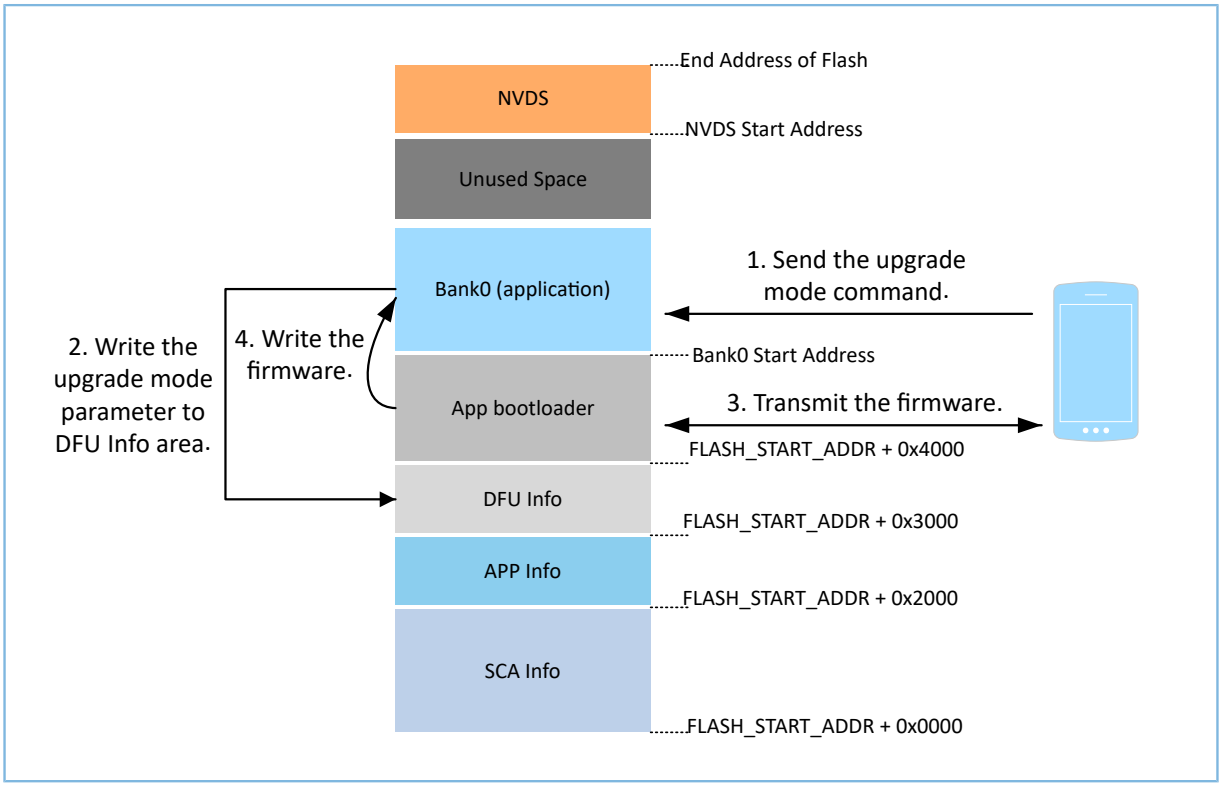

Figure 2-5 Firmware download procedure

According to the steps above, the host first connects to the Bank0 firmware; after receiving the non-background single-bank DFU mode command, the Bank0 firmware resets the device. Then, the mobile App (GRToolbox) needs to reconnect to the App bootloader firmware. To ensure accurate connection of the mobile App to the App bootloader firmware, GR5xx provides a solution: Assuming the Bluetooth device address of the application firmware is x, after the mobile App jumps to the bootloader, the Bluetooth device address will change to x+1. This enables the mobile phone to automatically connect to the App bootloader firmware by the address + 1.

- For the scenario without the Bank0 firmware: App bootloader receives the DFU mode command and writes the DFU mode to the DFU Info area. No re-connection via Bluetooth is required. Other procedures are the same as those for the scenario with the Bank0 firmware.

2.2.3 App Bootloader Boot Procedure

In non-background single-bank DFU mode, no Bank1 area is available to cache the new firmware. Therefore, the DFU Info area stores DFU mode parameters only. The App bootloader boot procedure is as follows:

1. Read the non-background single-bank DFU mode command, and then start DFU.
2. After firmware download completes, update the new firmware parameters to the APP Info area, and then reset the device.
3. After the device is reset, read the data in the APP Info area, and then jump to and run the application firmware according to the information in the APP Info area.

The following figure shows the boot procedure:

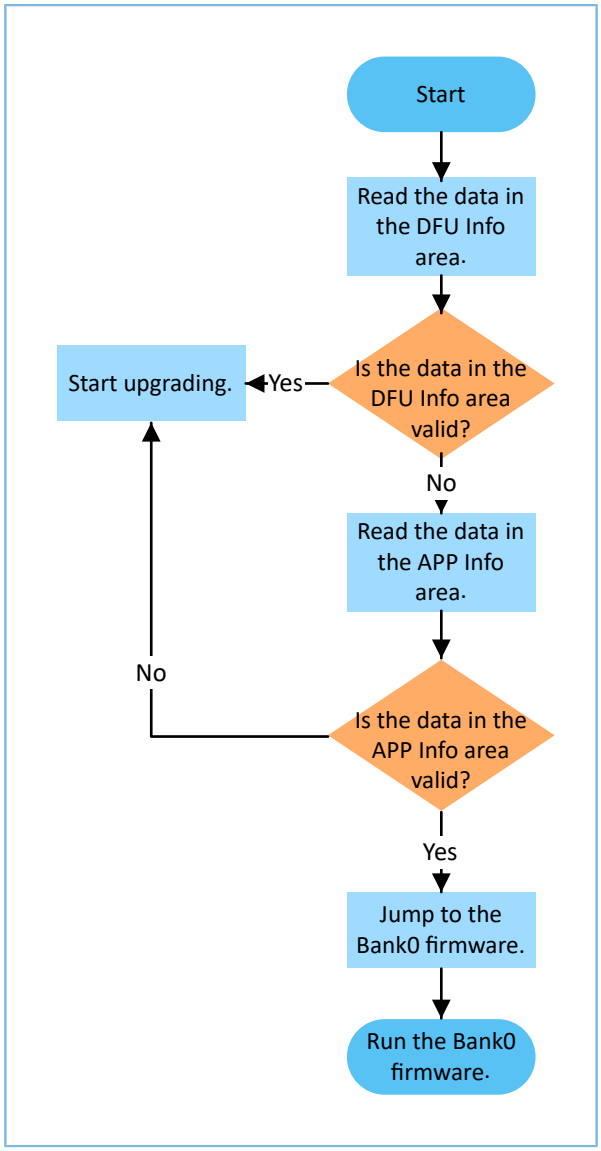

Figure 2-6 App bootloader boot procedure

2.3 Comparison of Upgrade Speeds

GR5xx offers two firmware upgrade speeds: normal mode and fast mode, with the latter being faster than the former. For details between the two speeds, refer to the table below.

Table 2-1 Comparison between normal mode and fast mode

Mode	Firmware Transfer Mode	RAM Requirement	Time Required	Description
Normal mode	For each frame of data transferred, the device will reply with the check value of the current data.	2 KB buffer to receive firmware	Long	In this mode, if an error occurs on a frame of data before the firmware is sent completely, the error can be detected immediately, and the upgrade process will be terminated.

Mode	Firmware Transfer Mode	RAM Requirement	Time Required	Description
Fast mode	The device will reply with check values only after the firmware is written completely.	8 KB buffer to receive firmware under the maximum bandwidth	Short	This mode can greatly shorten the time when you upgrade large firmware. However, data will be checked only after the firmware is sent completely.

Under the maximum bandwidth, the upgrade speed in fast mode can be about 6 times as fast as that in normal mode. Theoretically, error(s) may occur in fast mode during transmission, and the error(s) will only be discovered in the final check. Therefore, the time cost in this mode is relatively high if an error occurs. However, in actual tests, the probability of such errors is very low. Therefore, when upgrading large firmware, you can use fast mode to improve efficiency.

The introduction of fast mode also leads to an increase in RAM usage, so the RAM space for firmware upgrade is limited in some applications. In this case, using fast mode may cause insufficient system RAM space, and therefore, only normal mode can be used for upgrading. For application scenarios with limited RAM space, relevant configurations also need to be made on the firmware side, to minimize RAM usage. The configuration file is in SDK _Folder\components\libraries\dfu_port\dfu_port.h, and the detailed configuration is as follows. Based on this configuration, RAM usage can be minimized.

```
#define ONCE_WRITE_DATA_LEN                 1024

#define DFU_BUFFER_SIZE                     2048
```

The data interaction mechanisms for the two modes are as follows.

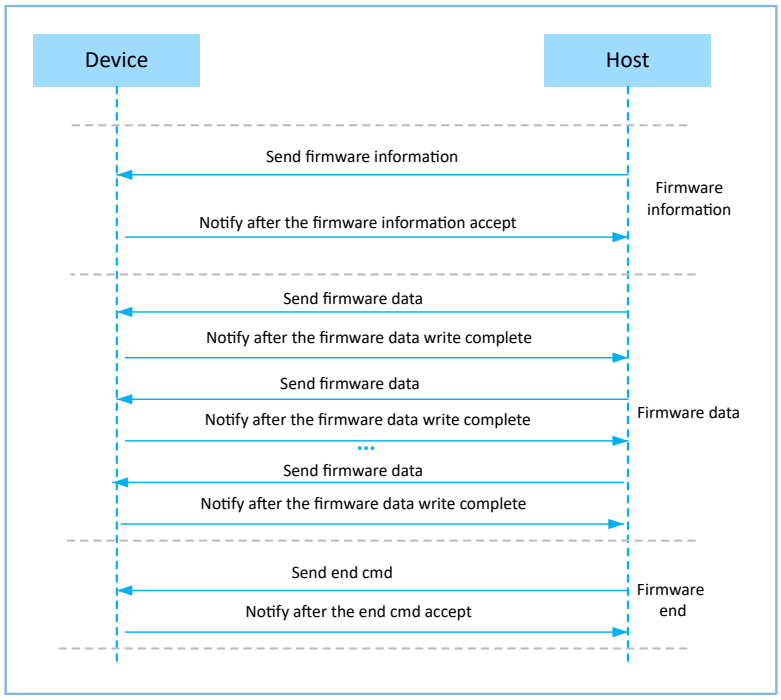

Figure 2-7 Normal mode

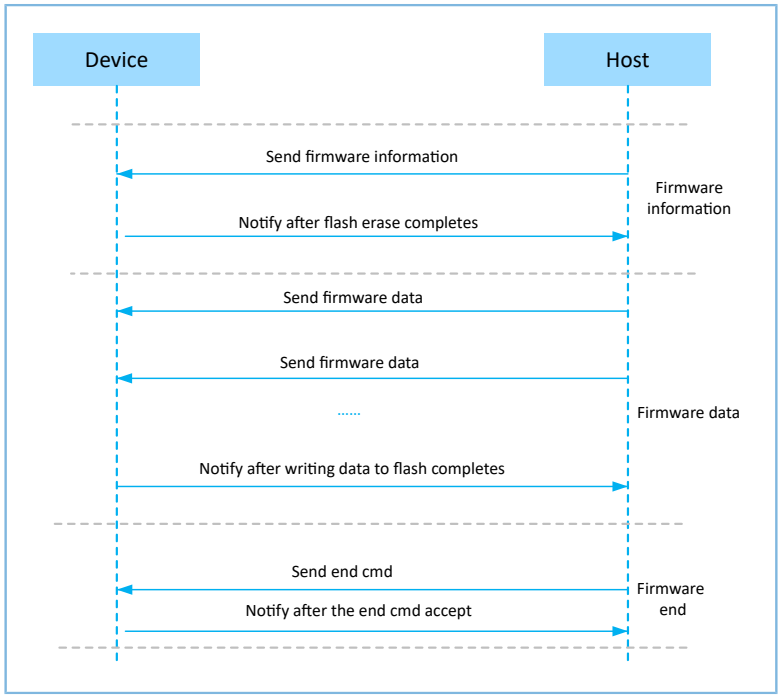

Figure 2-8 Fast mode

In normal mode, the device needs to reply each firmware data frame sent by the host. However, in fast mode, the device will reply only after all data frames have been sent. Therefore, fast mode has a higher upgrade speed than normal mode.

2.4 Firmware Format

To ensure security, it is necessary to sign or encrypt the firmware during transmission. GR5xx provides three firmware formats: unencrypted and unsigned firmware, encrypted and signed firmware, and signed firmware. The .bin formats for the three types of firmware are shown as follows.

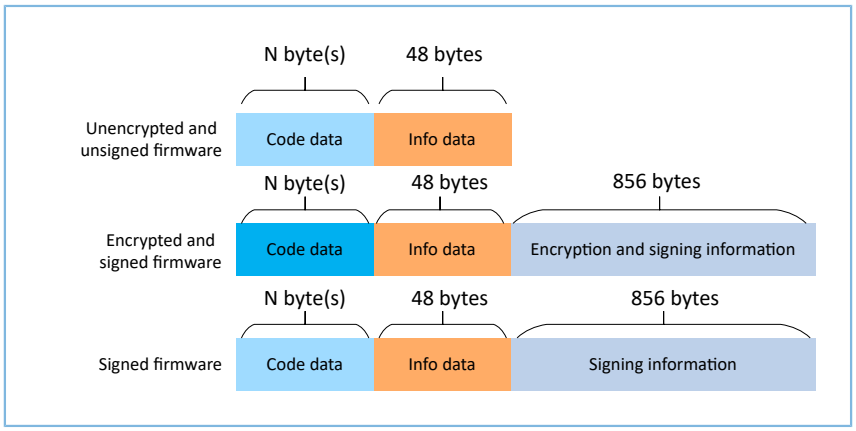

Figure 2-9 Firmware format

Each field of the data format is detailed below:

- Code data: firmware data that shall be 16-byte aligned. N indicates a variable length.
- Info data: firmware description (Image Info in “Section 2.1.1 Flash Layout”)
- Encryption and signing information: information needed for encrypting and signing an unencrypted and unsigned firmware file
- Signing information: information needed for signing an unencrypted and unsigned firmware file

3 Introduction to App Bootloader Project

The App bootloader project is in SDK_Folder\projects\ble\dfu\app_bootloader, and the project directory structure is shown below.

Note:

SDK_Folder is the root directory of the GR5xx SDK in use.

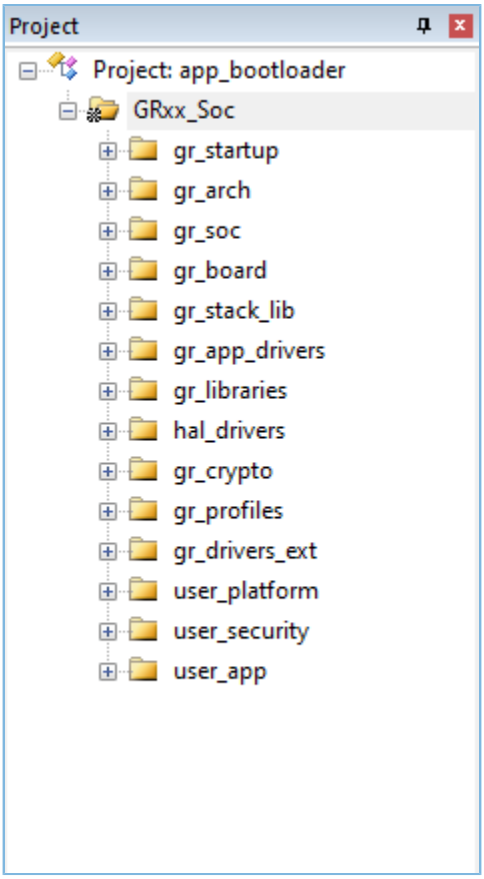

Figure 3-1 Project directory

Note:

Only the App bootloader project in the GR5405 SDK contains the hal_drivers directory.

The files and corresponding functionalities in each group are described in detail as follows:

Table 3-1 Group details of the App bootloader project

Group	Description
gr_startup	Assembly startup file
gr_arch	SoC architecture files
gr_soc	SoC initialization file
gr_board	Starter Kit Board (SK Board) initialization file

Group	Description
gr_stack_lib	Protocol stack library file
gr_app_drivers	App driver files
gr_libraries	Component files, including DFU components
hal_drivers	HAL driver files
gr_crypto	Cryptography algorithm files
gr_profiles	OTA profile files
user_platform	File for users to initialize the platform
user_security	Encrypting and signing files
user_app	App bootloader project functionality files

The App bootloader firmware is needed for the entire upgrade process in both background dual-bank DFU mode and non-background single-bank DFU mode. Main functionalities of the App bootloader firmware:

- Application firmware startup
- Verification of firmware signature and checksum
- Firmware download (receiving firmware from the host)
- Firmware programming

The functionalities of the App bootloader vary depending on the specific upgrade mode. To avoid redundant functionalities, relevant macros are provided to trim down the functionalities. The following table details the configuration items of App bootloader.

Note:

- *bootloader_config.h* is in SDK_`Folder\projects\ble\dfu\app_bootloader\Src\config`.
- *custom_config.h* is in SDK_`Folder\projects\ble\dfu\app_bootloader\Src\config`.

Table 3-2 App bootloader configuration items

File Name	Macro	Description
bootloader_config.h	BOOTLOADER_DFU_BLE_ENABLE	• 0: Disable DFU communication via Bluetooth Low Energy (Bluetooth LE). In background dual-bank DFU mode, only jumping and copying functionalities of App bootloader are needed, so this macro can be disabled. • 1: Enable DFU communication via Bluetooth LE. In non-background single-bank DFU mode, App bootloader will receive firmware information, and this macro needs to be enabled to upgrade firmware via Bluetooth LE.
	BOOTLOADER_DFU_UART_ENABLE	• 0: Disable DFU communication via UART. This macro can be disabled in background dual-bank DFU mode.

File Name	Macro	Description
		• 1: Enable DFU communication via UART. In non-background single-bank DFU mode, this macro needs to be enabled to upgrade firmware via UART.
	BOOTLOADER_DFU_ENABLE	The macro can be used with its default configuration to indicate whether App bootloader enables the DFU functionality. The value of this macro depends on the values of the above two macros, BOOTLOADER_DFU_BLE_ENABLE and BOOTLOADER_DFU_UART_ENABLE.
	BOOTLOADER_WDT_ENABLE	Enable/Disable the watchdog. • 0: Disable • 1: Enable When a program crashes, the system can be reset through watchdog feeding. If there is a risk of system crashes, this macro needs to be enabled.
	BOOTLOADER_SIGN_ENABLE	To upgrade an encrypted or signed firmware file, enable this macro to enable App bootloader to perform signature verification on the firmware. • 0: Disable signature verification. • 1: Enable signature verification.
	BOOTLOADER_PUBLIC_KEY_HASH	After signature verification is enabled, public_key_hash shall be configured to correctly complete the signature verification process of the new firmware by App bootloader firmware. For detailed configuration, refer to “Section 4.4.3 Firmware Configuration”.
	DFU_FW_SAVE_ADDR	In App bootloader, this macro represents the start address to store the Bank0 application firmware. For the specific value, refer to “Section 2.2.1 Flash Layout”. In ble_app_template_dfu firmware, this macro represents the start address to store the Bank1 firmware (the address displayed in the **Copy Address(0x)** field in GRToolbox in background dual-bank DFU mode). For specific configuration, refer to “Section 2.2.1 Flash Layout”.
	BOOTLOADER_BOOT_PORT_ENABLE	When users do not use the GR5xx DFU scheme, enable this macro to use a custom scheme. • 0: Use the default scheme provided by Goodix. • 1: Use a custom scheme.
	APP_FW_COMMENTS	The default configuration is “ble_app_temp”. This macro must be set to match the application in the Bank0 area, to allow the application to jump from App bootloader firmware to application firmware.

File Name	Macro	Description
		When App bootloader starts, if the APP Info data is detected to be invalid, it will further search for data in SCA and match the firmware COMMENTS in the SCA data with this macro. If the match is successful, the SCA data will be updated to the APP Info area, and the firmware will be started according to the matched SCA data. **Note:** • APP_FW_COMMENTS supports up to 12 characters. • The first 12 characters of the firmware file name should correspond to APP_FW_COMMENTS.
custom_config.h	APP_LOG_ENABLE	In development and debug phase, this macro allows users to view the log information output by App bootloader. • 0: Disable log information. • 1: Enable log information.
	APP_CODE_LOAD_ADDR	This macro represents the start address in Flash to store the App bootloader firmware. For the specific value, refer to "Section 2.1.1 Flash Layout" or "Section 2.2.1 Flash Layout" according to the upgrade mode.
	APP_CODE_RUN_ADDR	This macro represents the start address in Flash to run the App bootloader firmware. Generally, the value is consistent with that of APP_CODE_LOAD_ADDR.
N/A	ENABLE_DFU_SPI_FLASH	This macro sets whether to support external Flash to upgrade resource data. Enabling this macro allows upgrading resource data for external Flash in App bootloader firmware. Add the macro to a project in Keil by clicking **Options for Target** > **C/C++** > **Preprocessor Symbols** > **Define**. For instructions on operation, refer to "Section 4.6.2 External Flash Resource Upgrade".

To use a custom DFU scheme, set BOOTLOADER_BOOT_PORT_ENABLE in *bootloader_config.h* to 1 and add the custom DFU scheme to SDK_Folder\projects\ble\dfu\app_bootloader\Src\user\bootloader_boot_port.c. The framework of the code snippet is as follows.

```
#include "bootloader_config.h"
#if BOOTLOADER_BOOT_PORT_ENABLE
#include "bootloader_boot.h"
void bootloader_dfu_task(void)
{
}
void bootloader_verify_task(void)
{
}
void bootloader_jump_task(void)
{
```

```
}
#endif
```

Note:

To make the App bootloader firmware smallest (minimum code size), it is necessary to adopt the background dual-bank DFU mode, and disable DFU communication via Bluetooth LE of App bootloader (BOOTLOADER_DFU_BLE_ENABLE), DFU communication via UART (BOOTLOADER_DFU_UART_ENABLE), log information (APP_LOG_ENABLE), and external Flash (ENABLE_DFU_SPI_FLASH).

4 Upgrade with GRToolbox

This chapter elaborates on how to upgrade GR5xx firmware with GRToolbox.

4.1 Supported Platform

Table 4-1 Supported development platform

Software Development Platform	Development Board
GR551x SDK V2.0.1 and later versions	GR5515-SK-BASIC
GR5526 SDK V1.0.1 and later versions	GR5526-SK-BASIC
GR5525 SDK V0.8.0 and later versions	GR5525-SK-BASIC
GR533x SDK V0.9.0 and later versions	GR5331-SK-BASIC
GR5405 SDK V1.1.6 and later versions	GR5405-SK-BASIC

4.2 Preparation

- **Hardware preparation**

Table 4-2 Hardware preparation

Name	Description
Development board	Starter Kit Board of the corresponding SoC
Android phone	A mobile phone running on Android 5.0 (KitKat) or later
Connection cable	USB Type-C cable (Micro USB 2.0 cable for GR551x SoCs)
DuPont wire	3
J-Link debug probe	JTAG emulator launched by SEGGER. For more information, visit https://www.segger.com/products/debug-probes/j-link/.

- **Software preparation**

Table 4-3 Software preparation

Name	Description
Windows	Windows 7/Windows 10
J-Link driver	A J-Link driver. Available at https://www.segger.com/downloads/jlink/.
Keil MDK5	An integrated development environment (IDE). MDK-ARM Version 5.20 or later is required. Available at https://www.keil.com/download/product/.
GProgrammer (Windows)	A programming tool. Available at https://www.goodix.com/en/software_tool/gprogrammer_ble.
GRUart (Windows)	A serial port debugging tool. Available at https://www.goodix.com/en/download?objectId=43&objectType=software.

Name	Description
GRToolbox (Android) V2.16 and later versions	A Bluetooth LE debugging tool. Available at https://www.goodix.com/en/software_tool/grtoolbox.

Note:

- To use the DFU scheme introduced in this document, GRToolbox must be in V2.16 or later versions, and the firmware on the device must satisfy the requirements on SDK versions in Table 4-1.
- If users use the DFU scheme introduced here in the application firmware, but use the Second Boot firmware as the startup firmware (for GR551x SDK V1.7.0 and earlier versions or GR5526 SDK V1.0.0 and earlier versions), they can complete firmware upgrade with GRToolbox V2.16 and later versions in background dual-bank DFU mode.

The following sections will introduce upgrade steps for unencrypted and unsigned firmware, encrypted and signed firmware, and signed firmware respectively by taking GR551x for example.

4.3 Upgrade of Unencrypted and Unsigned Firmware

4.3.1 Firmware Configuration

The following two projects are needed for upgrade with GRToolbox.

- Project in SDK_Folder\projects\ble\dfu\app_bootloader\Keil_5
- Project in SDK_Folder\projects\ble\ble_peripheral\ble_app_template_dfu\Keil_5

1. Configure the App bootloader project.

   The configuration items of the App bootloader project for upgrading unencrypted and unsigned firmware are listed below.

Note:

- *bootloader_config.h* is in SDK_Folder\projects\ble\dfu\app_bootloader\Src\config.
- *custom_config.h* is in SDK_Folder\projects\ble\dfu\app_bootloader\Src\config.

Table 4-4 Configuration items of the App bootloader project (for upgrading unencrypted and unsigned firmware)

File Name	Macro	Value
bootloader_config.h	BOOTLOADER_DFU_BLE_ENABLE	1: Enable DFU communication via Bluetooth LE.
	BOOTLOADER_DFU_UART_ENABLE	0: Disable DFU communication via UART.
	BOOTLOADER_WDT_ENABLE	1: Enable
	BOOTLOADER_SIGN_ENABLE	0: Disable signature verification.
	BOOTLOADER_BOOT_PORT_ENABLE	0: Use the default scheme provided by Goodix.
	APP_FW_COMMENTS	"ble_app_temp"

File Name	Macro	Value
	DFU_FW_SAVE_ADDR	(FLASH_START_ADDR + 0x30000) **Note:** **FLASH_START_ADDR** may vary according to a specific SoC.
custom_config.h	APP_LOG_ENABLE	1: Enable log information.
	APP_CODE_LOAD_ADDR	(Flash_START_ADDR+0x40000) **Note:** **FLASH_START_ADDR** may vary according to a specific SoC.
	APP_CODE_RUN_ADDR	

2. Configure the ble_app_template_dfu project.

Note:

- *custom_config.h* is in SDK_Folder\projects\ble\ble_peripheral\ble_app_template_dfu\Src\config.
- *user_periph_setup.c* is in SDK_Folder\projects\ble\ble_peripheral\ble_app_template_dfu\Src\platform.

Table 4-5 Configuration items of the ble_app_template_dfu project (for upgrading unencrypted and unsigned firmware)

File Name	Macro	Value
custom_config.h	APP_CODE_LOAD_ADDR	0x01030000 **Note:** The address may vary according to a specific SoC.
	APP_CODE_RUN_ADDR	
user_periph_setup.c	DFU_FW_SAVE_ADDR	(FLASH_START_ADDR + 0x60000) **Note:** The start address of the Bank1 area in Flash cannot overlap the Bank0 area and the App bootloader area.

4.3.2 Firmware Programming

Compile the App bootloader and ble_app_template_dfu projects, to generate *app_bootloader.bin* and *ble_app_template_dfu.bin* respectively; then import the two .bin files into GProgrammer for programming and set the App bootloader firmware as the boot firmware. Firmware programming with GProgrammer is shown as follows.

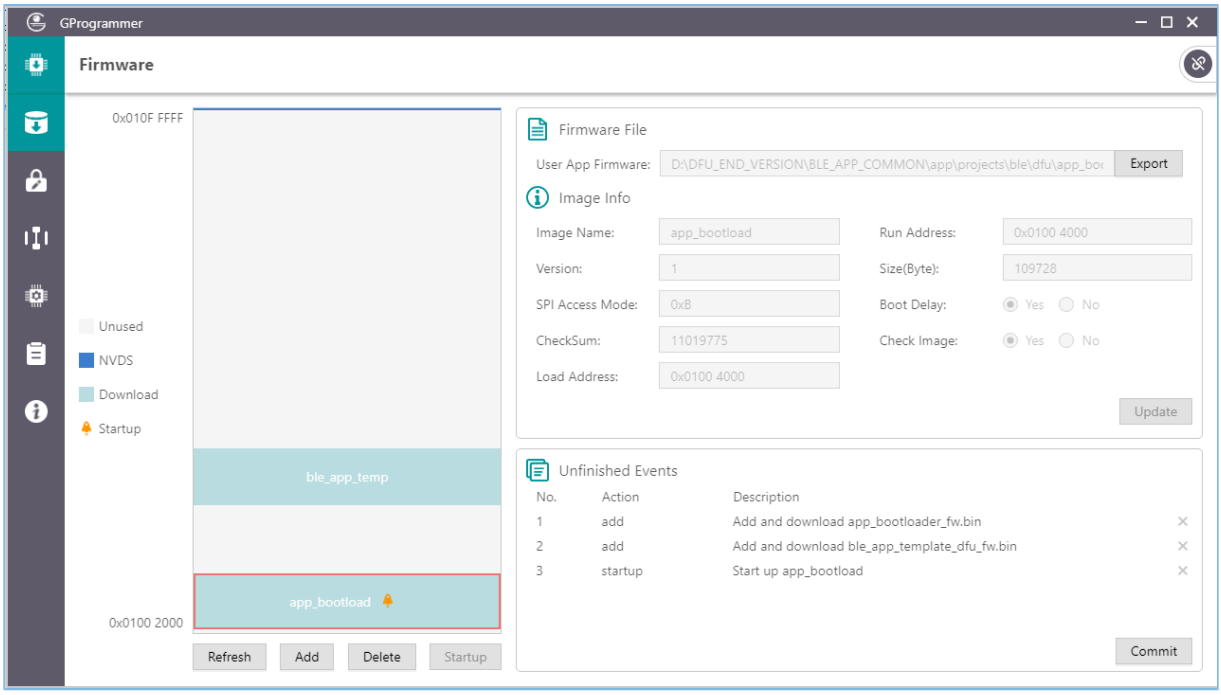

Figure 4-1 Firmware programming

After firmware programming, GRUart will output logs as follows.

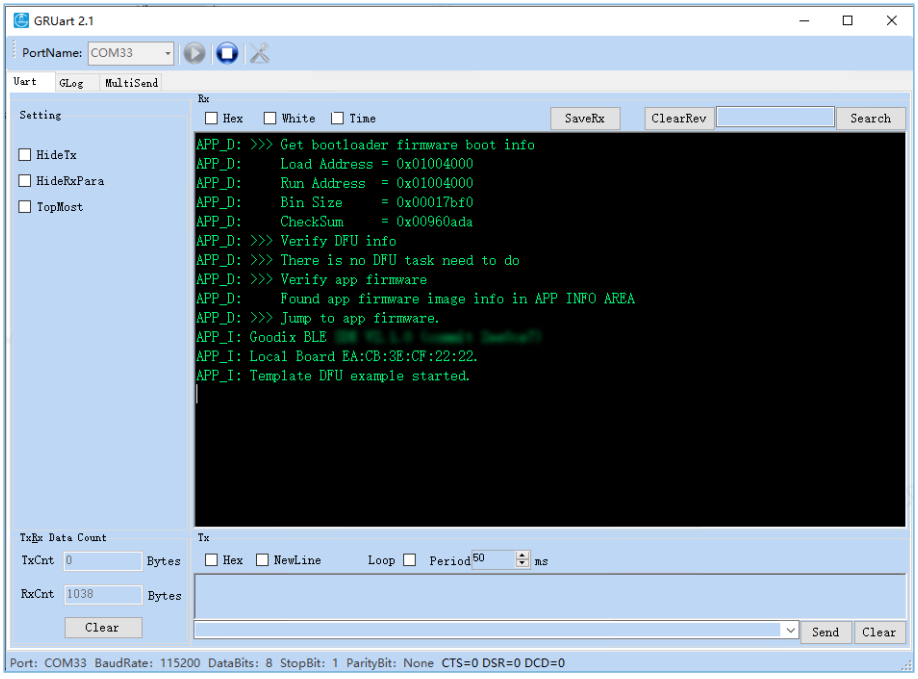

Figure 4-2 Log information

4.3.3 Creating Target Firmware for Upgrade

Open SDK_Folder\projects\ble\ble_peripheral\ble_app_template_dfu, then copy the ble_app_template_dfu project in this directory and rename the project as ble_app_template_dfu_mine, and finally open the project in Keil. Follow the steps below to create target firmware for upgrade.

1. Name the target firmware.

   Change the advertising name of the device in *user_app.c* (path: SDK_Folder\projects\ble\ble_peripheral\ble_app_template_dfu_mine\Src\user) to **Goodix_Tem_New**:

   (1) Change the advertising name of the device.

```
#define DEVICE_NAME "Goodix_Tem_New"
```

   (2) Modify the name used for scan response.

```
static const uint8_t s_adv_rsp_data_set[] =
{
 // Complete Name
 0x0f,
 BLE_GAP_AD_TYPE_COMPLETE_NAME,
 'G', 'o', 'o', 'd', 'i', 'x', '_', 'T', 'e', 'm', '_', 'N', 'e', 'w',
 // Manufacturer specific adv data type
 0x05,
 BLE_GAP_AD_TYPE_MANU_SPECIFIC_DATA,
 // Goodix SIG Company Identifier: 0x04F7
 0xF7,
 0x04,
 // Goodix specific adv data
 0x02, 0x03,
};
```

2. Modify the load address and run address. Take GR551x as an example. Modify APP_CODE_RUN_ADDR and APP_CODE_LOAD_ADDR to 0x01030000. For configurations of other SoC series, refer to "Section 2.1.1 Flash Layout" or "Section 2.2.1 Flash Layout".

Table 4-6 To modify the load address and run address

File Name	Macro	Value
custom_config.h	APP_CODE_LOAD_ADDR	0x01030000
	APP_CODE_RUN_ADDR	

3. Generate a .bin file.

   (1) Recompile the project. After recompilation, *ble_app_template_dfu.bin* is generated in ble_app_template_dfu_mine\keil_5\Listings.

   (2) The firmware generated by Keil contains no firmware information at the end. However, during upgrade, the transmitted firmware needs to contain firmware information. Therefore, after *ble_app_template_dfu.bin* is generated by Keil, it needs to be imported into GProgrammer.

   (3) Then, the target firmware, *ble_app_template_dfu_fw.bin*, is generated in ble_app_template_dfu_mine\keil_5\Listings, as shown below.

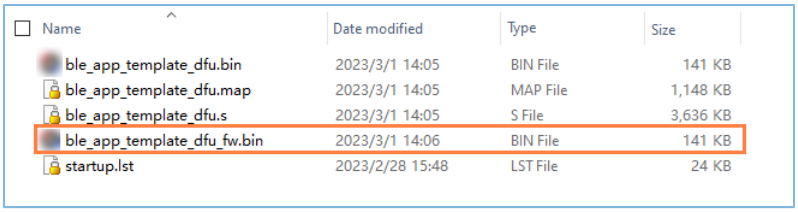

Figure 4-3 Generated target firmware

(4) Save *ble_app_template_dfu_fw.bin* to the root directory of the mobile phone.

4.3.4 To Enter DFU Interface of GRToolbox

Two approaches are available to enter the **DFU** interface of GRToolbox:

- Click in the upper-right corner on the **Device** interface.
- Click **Application** > **DFU**.

Note:

GR551x is taken as an example for the GRToolbox screenshots mentioned in this section.

The two approaches are detailed below.

- Click in the upper-right corner on the **Device** interface.
  1. Start GRToolbox; select **Goodix_Tem_DFU** in the device list on the **Device** interface and establish connection (Figure 4-4).
  2. Click in the upper-right corner on the **Device** interface (Figure 4-5) to jump to the **DFU** interface (Figure 4-6).

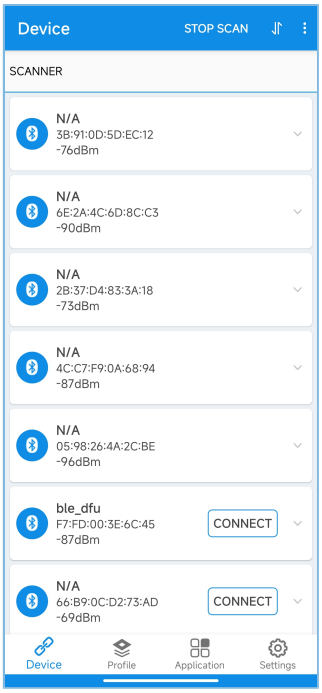

Figure 4-4 Selecting **Goodix_Tem_DFU**

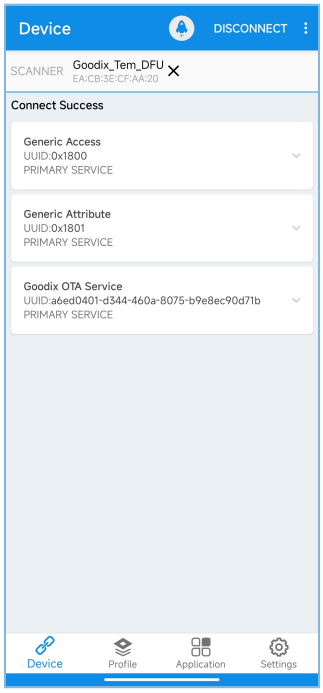

Figure 4-5 Clicking

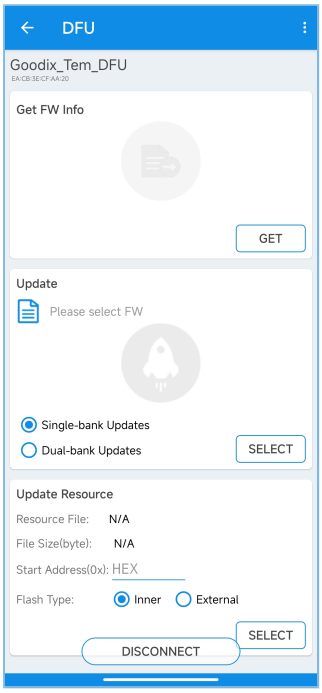

Figure 4-6 **DFU** interface

- Click **Application** > **DFU**.

  1. Start GRToolbox; click **Application** > **DFU**, as shown in Figure 4-7.
  2. Enter the **DFU** interface (Figure 4-8). As no device is connected, click **CONNECT** at the bottom to enter the device connection interface, as shown in Figure 4-9.
  3. Select **Goodix_Tem_DFU** from the device list to enter DFU mode, as shown below.

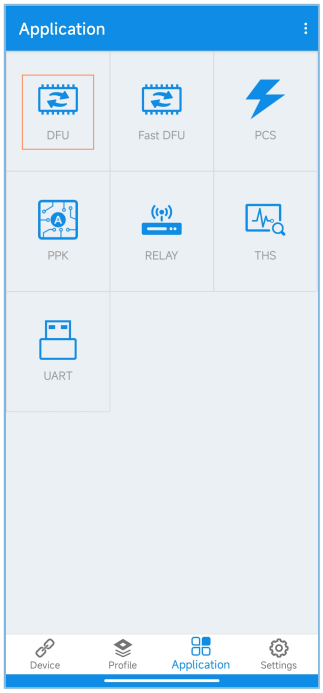

Figure 4-7 Clicking **DFU**

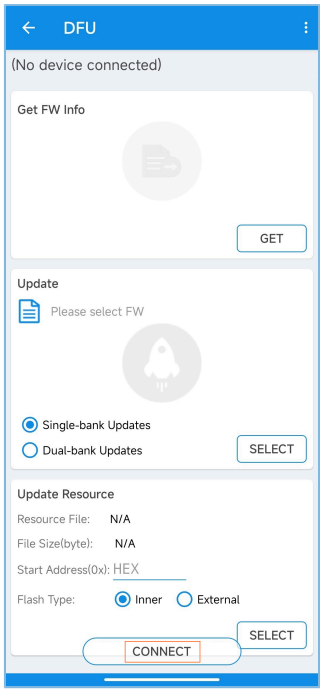

Figure 4-8 **DFU** interface

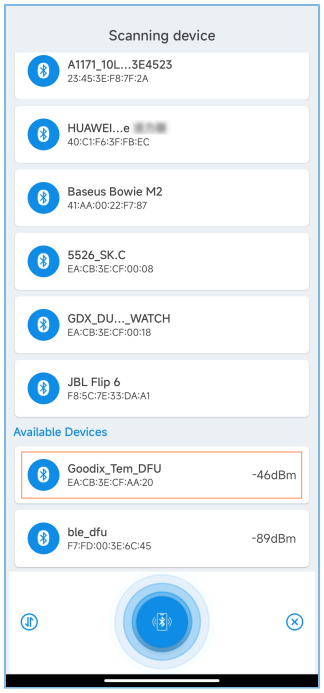

Figure 4-9 Scanning device

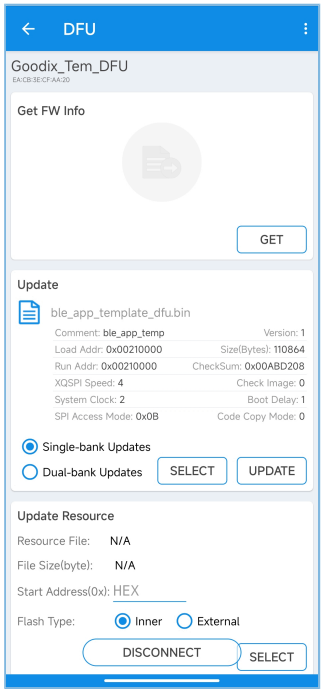

Figure 4-10 Entering DFU mode

4.3.5 Firmware Upgrade

This section focuses on detailed upgrade operations in background dual-bank DFU mode and non-background single-bank DFU mode.

4.3.5.1 Background dual-bank DFU mode

1. After entering DFU mode on GRToolbox, click **SELECT** to import *ble_app_template_dfu_fw.bin* in the root directory, and select **Dual-bank Updates**.
2. After **Dual-bank Updates** is selected, **Copy Address(0x)** appears below (Figure 4-11).

   By default, **Copy Address(0x)** is greyed out and cannot be modified. If **Copy Address(0x)** is incorrect, click ⋮ in the upper-right corner and select **Customize Copy Address** to modify the address, as shown in Figure 4-13.

   Fast Mode and **Write Ctrl Point** in Figure 4-12 are described below:

   - **Fast Mode**: Select whether to enable fast mode. If **Fast Mode** is not selected, firmware upgrade will be performed in normal DFU mode, which is slower than fast mode.
   - **Write Ctrl Point**: Send a command to start DFU. In applications, the DFU task is not always running, to reduce power consumption. If the DFU task for the device connected with GRToolbox is not running now and firmware information needs to be obtained through GRToolbox before the upgrade starts, you can click **Write Ctrl Point** to send a command to start the DFU task. For DFU task stop and start mechanisms, refer to "Chapter 5 DFU Porting Method".

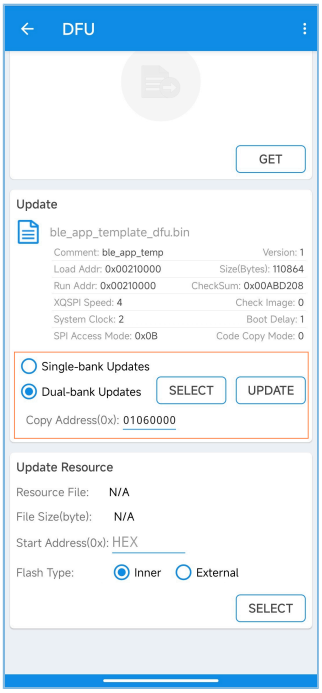

Figure 4-11 Selecting **Dual-bank Updates**

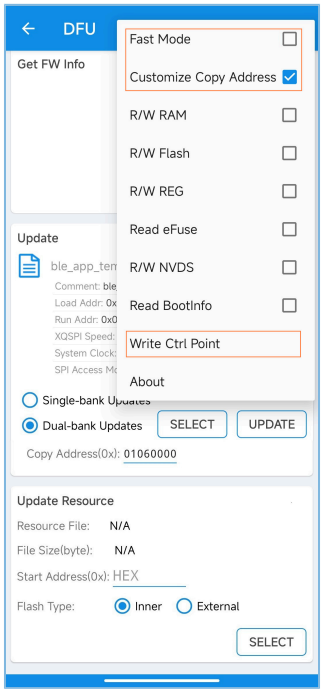

Figure 4-12 To customize copy address

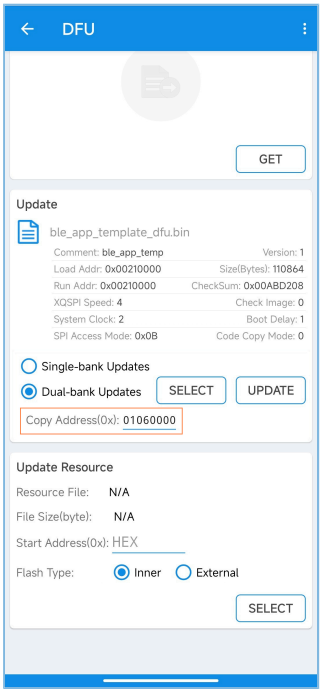

Figure 4-13 To modify **Copy Address**

3. Click **UPDATE** to start upgrade, as shown in Figure 4-14. During the upgrade process, there will be a circular progress view indicating the upgrade progress. After upgrade completes, “Upgrade completed.” will be displayed at the bottom.

   When the target firmware is created, the advertising name of the device has been modified to **Goodix_Tem_New**, so you can search for firmware named as **Goodix_Tem_New** to verify whether the upgrade has completed. As shown in Figure 4-15, **Goodix_Tem_New** in the device list indicates that upgrade succeeds.

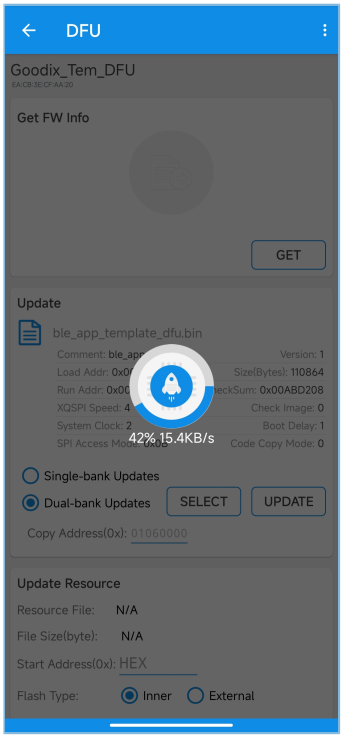

Figure 4-14 Upgrade progress

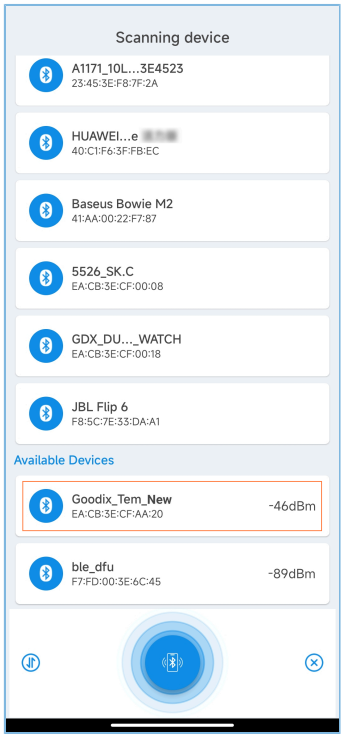

Figure 4-15 Searching for “Goodix_Tem_New”

4.3.5.2 Non-background Single-bank DFU Mode

The non-background single-bank DFU mode is applicable to two scenarios:

- The ble_app_template_dfu firmware is running.
- The application firmware is damaged, so the App bootloader firmware is running.

Operation steps for the two scenarios are detailed below.

- The ble_app_template_dfu firmware is running.

  The steps for non-background single-bank DFU mode are basically the same as those for background dual-bank DFU mode.

  1. Start GRToolbox; search for **Goodix_Tem_DFU** and establish connection; then enter the **DFU** interface and select **Single-bank Updates**, as shown in Figure 4-16.
  2. To upgrade in fast mode, select **Fast Mode** in the upper-right corner (Figure 4-17).
  3. After setting, click **UPDADE**.

     The time taken to enter the circular progress view interface in this mode is longer than that in background dual-bank DFU mode. This is because currently it is required to jump from the ble_app_template_dfu firmware to App bootloader firmware and re-establish connection. After reconnection, you can see the circular progress view (Figure 4-18). After upgrade completes, “Update completed.” will be displayed at the bottom.

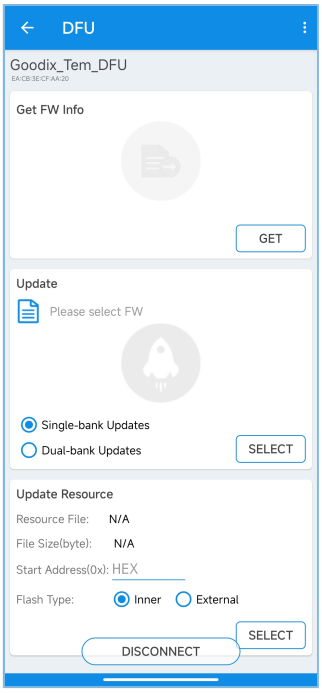

Figure 4-16 Selecting **Single-bank Updates**

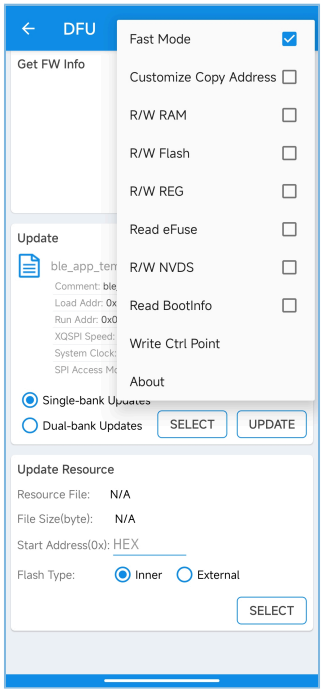

Figure 4-17 Selecting **Fast Mode**

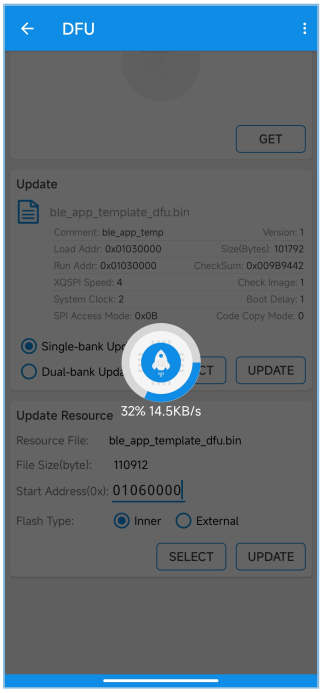

Figure 4-18 Upgrade progress

- The App bootloader firmware is running.

  If the application firmware is damaged or there is no application firmware, firmware upgrade needs to be performed in App bootloader. The operations are basically the same as those for upgrade in ble_app_template_dfu, except for different advertising names. The advertising name of the device to be connected is "Bootloader_OTA". The operations after connection are the same as those for upgrade in ble_app_template_dfu. You can refer to the steps for upgrade in ble_app_template_dfu.

4.4 Upgrade of Encrypted and Signed Firmware

Setting the system to encryption mode requires setting eFuse, which contains information on product configuration, security mode control, and keys for encryption and signing. Therefore, if no setting information has been downloaded to eFuse, you need to set eFuse first, and then download the setting information to eFuse. To enter the encryption page on GProgrammer, click **Encrypt & Sign** on the toolbar on the left, as shown below.

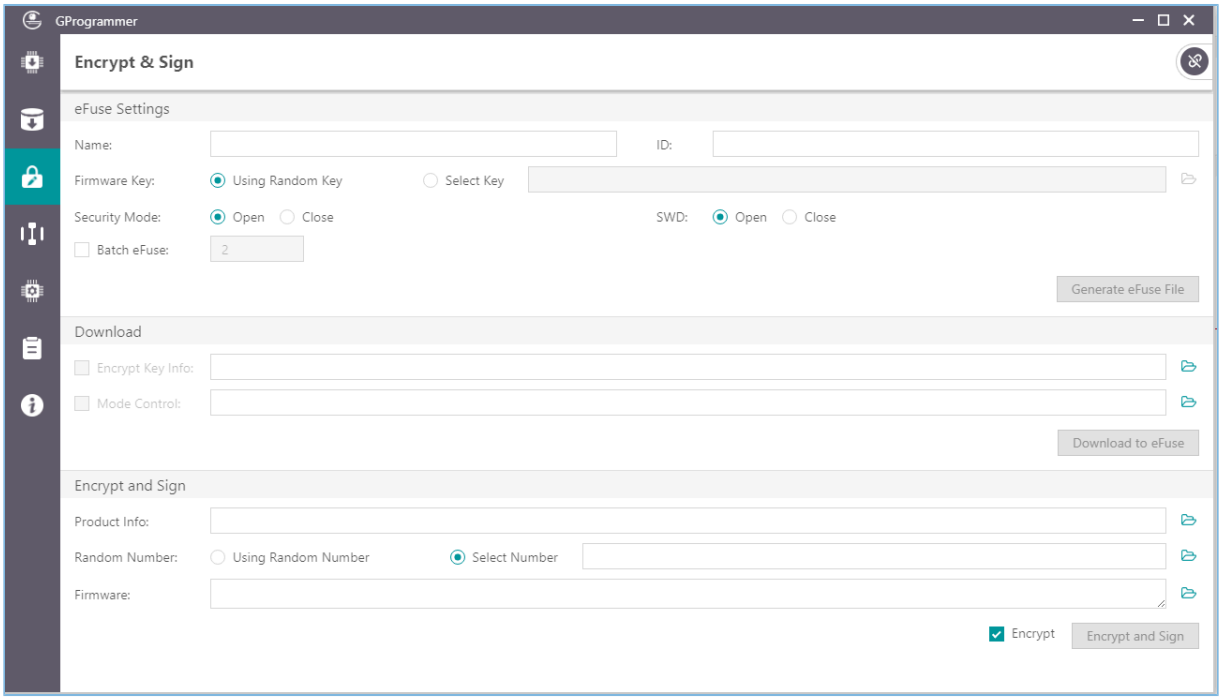

Figure 4-19 **Encrypt & Sign** interface

Note:

GR533x SoCs do not support encryption and signing related functionalities.

4.4.1 eFuse Setting

You can specify **Name**, **ID**, and **Firmware Key** on the **Encrypt & Sign** interface of GProgrammer, and configure **Security Mode** and **SWD**, to generate eFuse files.

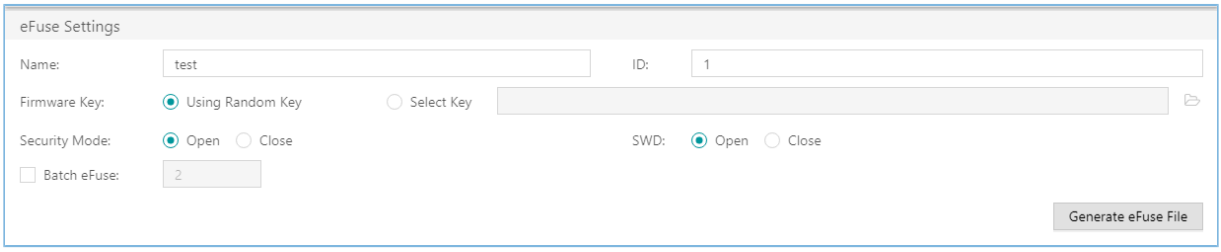

Figure 4-20 **eFuse Settings** pane

eFuse is a one-time programmable (OTP) memory with random access interfaces in SoCs. Parameters in Figure 4-20 are detailed below:

- **Firmware Key**: Firmware keys can be random keys automatically generated by software. Users can also add key files themselves.
- **Security Mode**: Select **Open** to enable security mode. Once enabled, this mode cannot be disabled.
- **SWD**: Select **Close** to disable SWD. In such case, users can upgrade firmware through DFU.

- **Batch eFuse**: To generate a specific number of firmware key files, select **Batch eFuse** and enter the number. The files are used for one-key-for-one-device scenario. If **Batch eFuse** is not selected, only one data key file will be generated.

The files are generated as below:

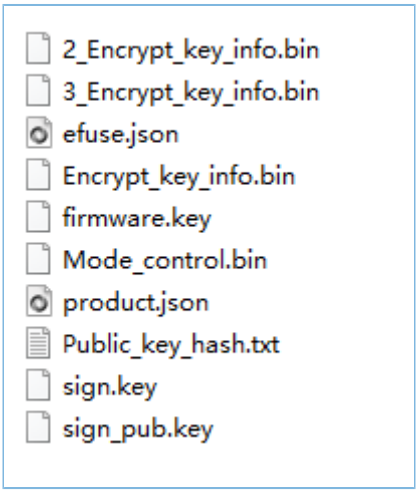

Figure 4-21 Generated files

- *efuse.json*: a temporary file
- *Encrypt_key_info.bin*, *2_Encrypt_key_info.bin*, and *3_Encrypt_key_info.bin*: files to be downloaded to eFuse, covering information on products, encryption, and signing. These files shall be downloaded to and stored in eFuse.
- *firmware.key*: a private key for encrypting firmware
- *Mode_control.bin*: a file covering information on security mode and SWD. The file shall be downloaded to and stored in eFuse.
- *product.json*: a product information file. This file shall be imported to GProgrammer for encrypting or signing firmware.
- *sign.key*: a private key to generate signatures
- *sign_pub.key*: a public key to verify signatures
- *Public_key_hash.txt*: a public key hash to verify signatures

To make file download to eFuse or firmware encryption and signing user-friendly, the paths for *Encrypt_key_info.bin* and *Mode_control.bin* are added to the **Download** area by default; the path for *product.json* is added to the **Product Info** pane in the **Encrypt and Sign** area by default.

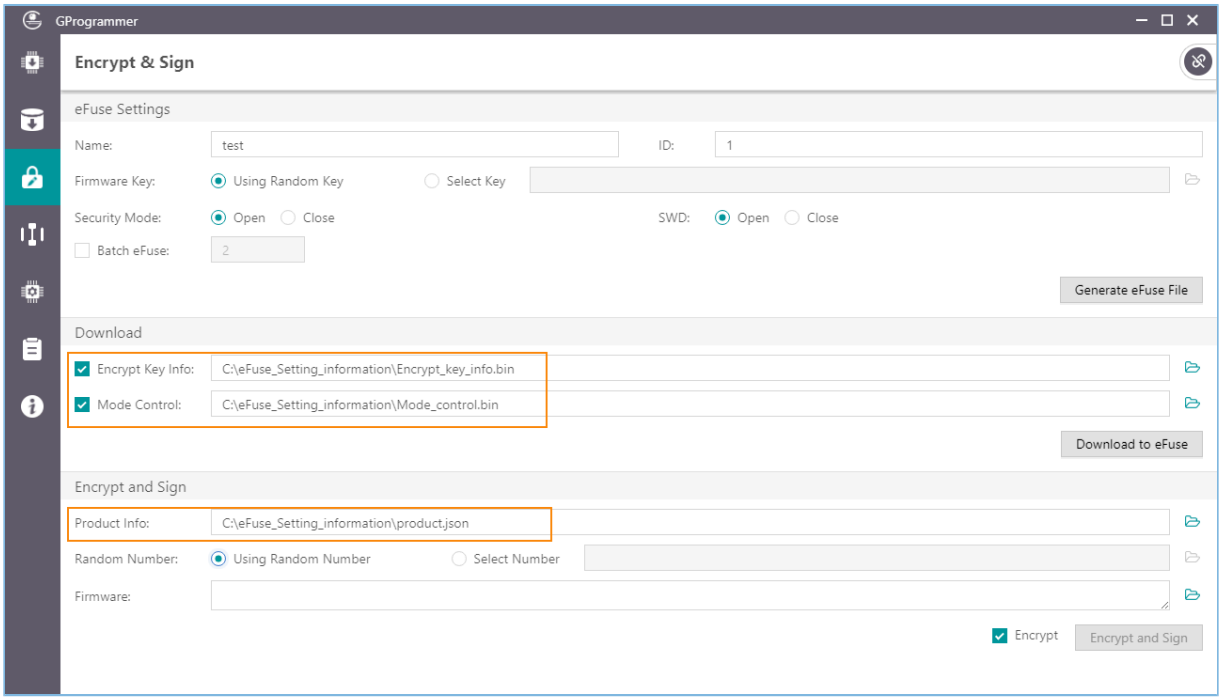

Figure 4-22 **Encrypt & Sign** interface

4.4.2 Download to eFuse

For users who have completed eFuse settings, click **Download to eFuse** to download the files to eFuse. Otherwise, users need to manually add *Encrypt_key_info.bin* and *Mode_control.bin* before downloading the files to eFuse.

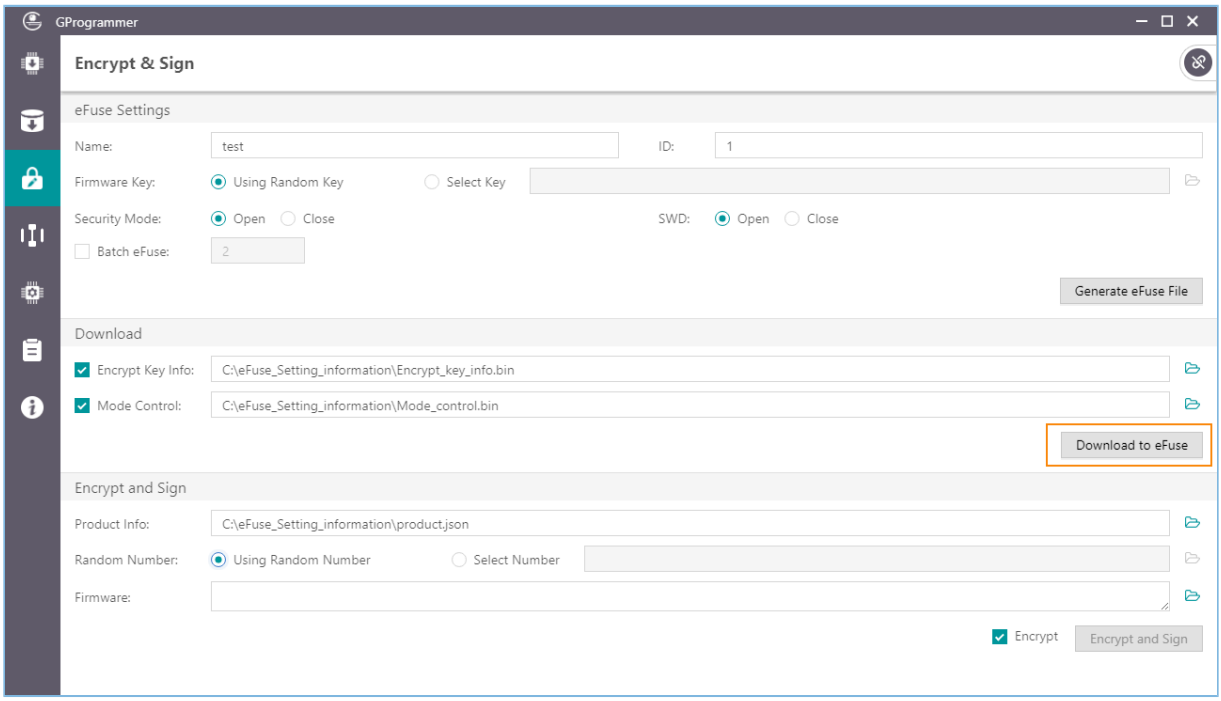

Figure 4-23 Downloading files to eFuse

4.4.3 Firmware Configuration

To upgrade encrypted and signed firmware, you need to configure the App bootloader related parameters, to verify the application firmware before jumping to it.

Note:

bootloader_config.h is in SDK_Folder\projects\ble\dfu\app_bootloader\Src\config.

Table 4-7 Configuration of *bootloader_config.h* (encryption and signing)

File Name	Macro	Value
bootloader_config.h	BOOTLOADER_SIGN_ENABLE	1: Enable signature verification.
	BOOTLOADER_PUBLIC_KEY_HASH	Values in *Public_key_hash.txt*

BOOTLOADER_PUBLIC_KEY_HASH is a set of values stored in *Public_key_hash.txt* as shown in Figure 4-21. Copy the values from the file and paste them into the BOOTLOADER_PUBLIC_KEY_HASH macro in *bootloader_config.h*.

4.4.4 Generating Encrypted and Signed Firmware

When encryption mode is enabled, only firmware that has been encrypted and signed can be downloaded to Flash. GProgrammer allows users to encrypt and sign multiple firmware files by using one set of product information (**Product Info**) and one random number (**Random Number**). When adding more than one firmware file by clicking **Encrypt & Sign** > **Encrypt and Sign** > **Firmware**, separate each file path with a semicolon (;), as shown in Figure 4-24.

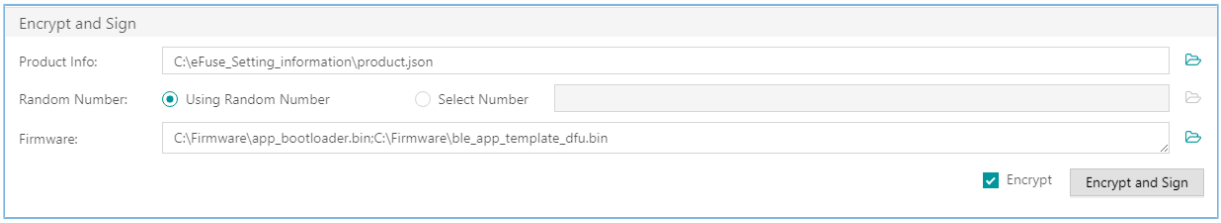

Figure 4-24 Adding more than one firmware file

In an encrypted SoC, it is required to encrypt both App bootloader firmware and ble_app_template_dfu firmware, so that the application can run correctly. The encrypted and signed firmware files are shown as follows:

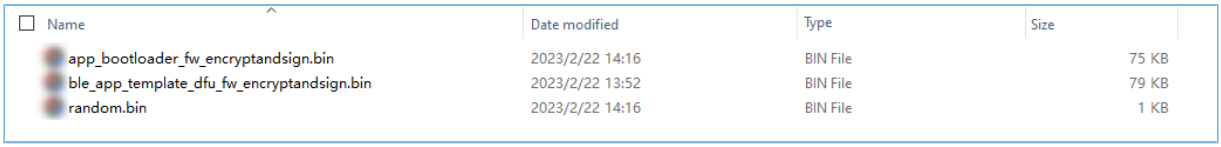

Figure 4-25 Firmware files after encryption and signing

4.4.5 Firmware Upgrade

The operations to upgrade encrypted and signed firmware through GRToolbox are almost the same as those for unencrypted and unsigned firmware, except that the encrypted and signed firmware is stored in the mobile phone. For specific operations, refer to "Section 4.3.5 Firmware Upgrade".

4.5 Upgrade of Signed and Unencrypted Firmware

Signing firmware is to prevent the firmware from being tampered with by third parties during transmission. Therefore, after the firmware is written and before jumping to the application firmware, the signature of the application firmware needs to be verified.

4.5.1 Firmware Configuration

Only signed and unencrypted firmware requires relevant configuration in App bootloader firmware. The relevant configuration items are shown in the following table:

Note:

bootloader_config.h is in SDK_Folder\projects\ble\dfu\app_bootloader\Src\config.

Table 4-8 Configuration of *bootloader_config.h* (signing)

File Name	Macro	Value
bootloader_config.h	BOOTLOADER_SIGN_ENABLE	1: Enable signature verification.
	BOOTLOADER_PUBLIC_KEY_HASH	Values in *Public_key_hash.txt*

4.5.2 Generating Signed and Unencrypted Firmware

Sign the firmware with GProgrammer, as shown in Figure 4-26. According to "Section 2.4 Firmware Format", the length of the contents added to the end of both encrypted firmware and signed firmware is the same, and the steps for generating them are basically the same. The difference is that to generate signed and unencrypted firmware, do not select **Encrypt** in the **Encrypt & Sign** > **Encrypt and Sign** area of GProgrammer. Figure 4-27 shows the generated signed firmware.

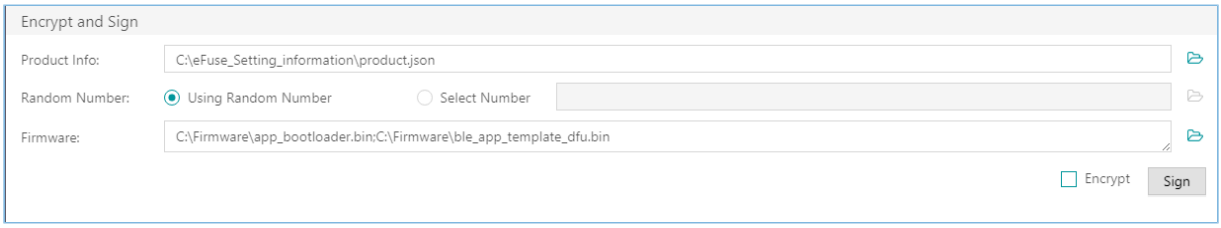

Figure 4-26 Firmware signing interface

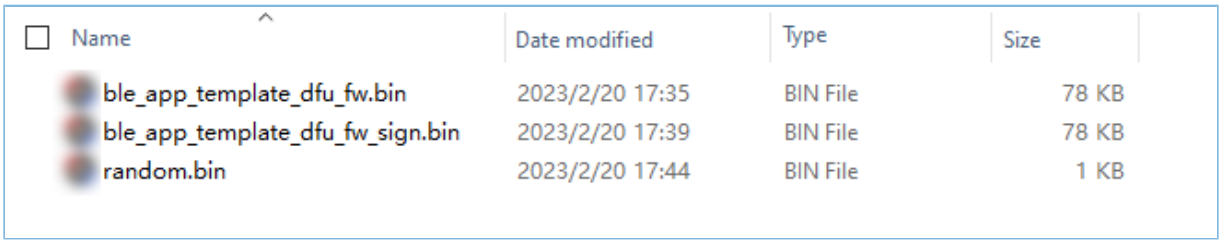

Figure 4-27 Generated signed firmware

4.5.3 Firmware Upgrade

The steps for upgrading signed and unencrypted firmware are the same as those for unencrypted and unsigned firmware. Refer to “Section 4.3.5 Firmware Upgrade”.

4.6 Resource Upgrade

Resource upgrade refers to upgrading images, fonts, audio files, and other non-code data. GR5xx supports resource upgrade for internal Flash and external Flash.

- Internal resource upgrade: Add DFU-related components to upgrade resources in the app_bootloader firmware or application firmware. No other configuration is required.
- External resource upgrade: Add DFU-related components and configure related macros in Keil.

4.6.1 Internal Flash Resource Upgrade

Steps for internal Flash resource upgrade:

1. Set **Start Address(0x)** and **Flash Type**, as shown in Figure 4-28.

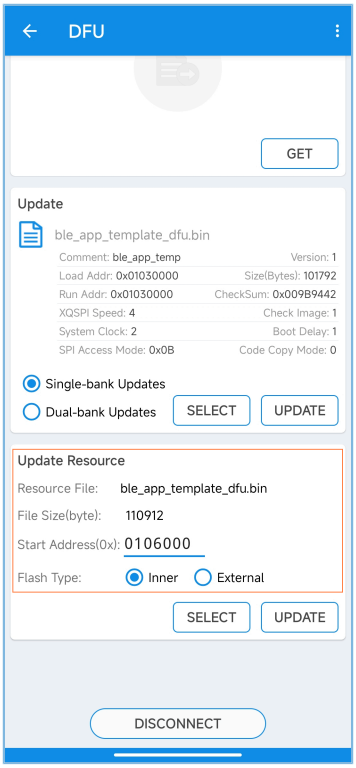

Figure 4-28 Internal Flash resource upgrade

   - **Start Address(0x)**: the start address to store the data; it needs to be set by developers in advance, to avoid overwriting other useful data during resource data download. For GR551x, considering the locations of *app_bootloader.bin* and *ble_app_template_dfu.bin* in Flash, you need to set the start address to 0x01060000. For other SoC series, refer to “Section 2.1.1 Flash Layout” or “Section 2.2.1 Flash Layout”.
   - **Flash Type**: **Inner** refers to Flash inside the SoC; **External** refers to external Flash connected to the SoC.

2. Click **UPDATE**. Then, the upgrade progress is shown as follows.

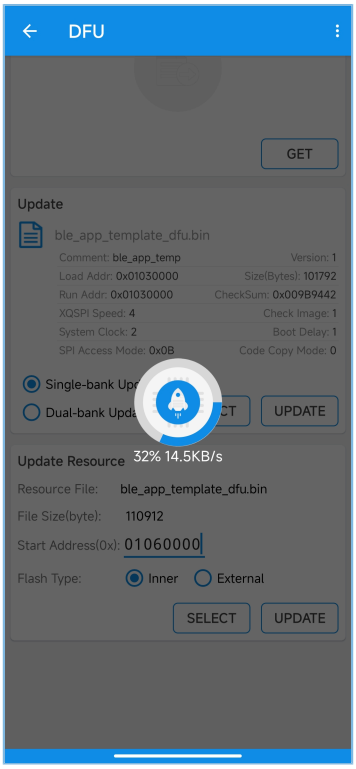

Figure 4-29 Internal resource upgrade progress

3. After resource download, the firmware in use will check the resource data. If the data passes the check, “Upgrade completed.” will be prompted at the bottom of GRToolbox.

4.6.2 External Flash Resource Upgrade

External Flash resources can be upgraded in both App bootloader firmware and ble_app_template_dfu firmware.

- A common scenario is to perform external Flash resource upgrade in ble_app_template_dfu firmware. Configure "ENABLE_DFU_SPI_FLASH" in the ble_app_template_dfu project, as shown in Figure 4-30, to enable the current project to perform external Flash resource upgrade.

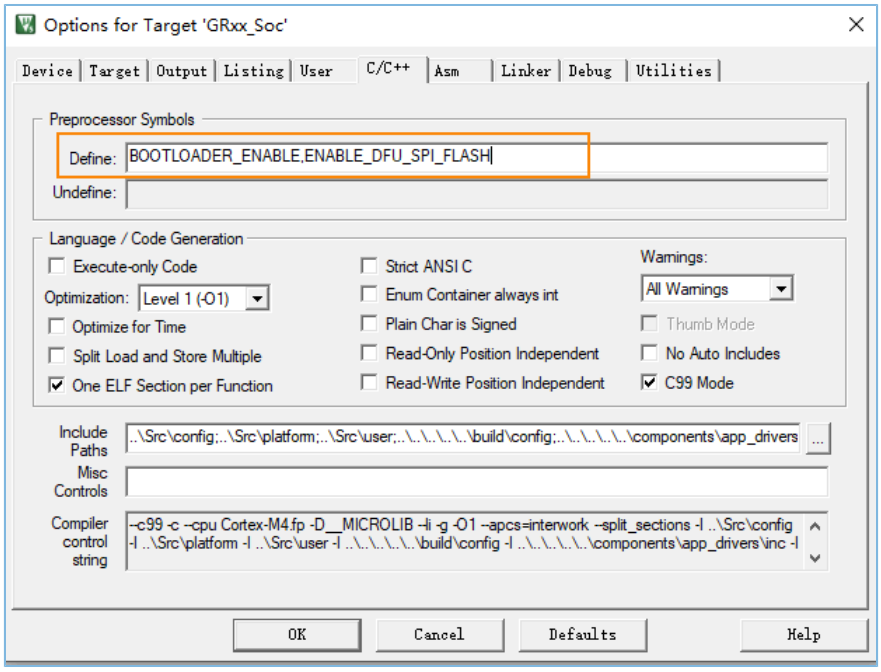

Figure 4-30 To enable external Flash resource upgrade

- To upgrade external Flash resources in App bootloader firmware, you need to add “ENABLE_DFU_SPI_FLASH” as shown in Figure 4-30, and add a macro to enable Bluetooth LE in the app_bootloader project. The table below lists the configuration in detail.

Table 4-9 Configuration of *bootloader_config.h*

File Name	Macro	Value
bootloader_config.h	BOOTLOADER_DFU_BLE_ENABLE	1: Enable DFU communication via Bluetooth LE.

After configuration, steps for external Flash resource upgrade in both App bootloader firmware and ble_app_template_dfu firmware are the same as follows:

1. Configure memory I/O, as shown below.

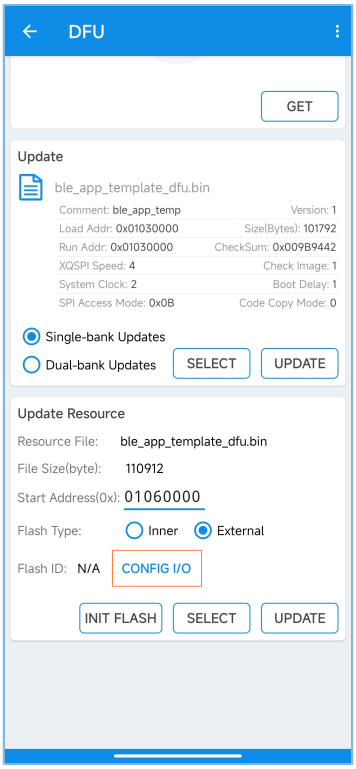

Figure 4-31 To configure memory I/O

To configure I/O interfaces, you can choose from SPI, QSPI0, and QSPI1 according to the communication mode between external Flash and GR5xx. In the example, the on-board external Flash is used for upgrading, and the on-board external Flash I/O interfaces for different SoC series are shown in the following table:

Table 4-10 On-board external Flash I/O interfaces

Development Board	I/O Type	CS	CLK	IO0	IO1	IO2	IO3
GR5515-SK-BASIC	QSPI1	GPIO_15	GPIO_9	GPIO_8	GPIO_14	GPIO_13	GPIO_12
GR5525-SK-BASIC	QSPI0	GPIO_15	GPIO_18	GPIO_19	GPIO_14	GPIO_13	GPIO_12
GR5526-SK_BASIC	QSPI0	GPIO_26	GPIO_21	GPIO_22	GPIO_23	GPIO_24	GPIO_25

Note:

- GR5331-SK-BASIC has no on-board external Flash or QSPI. To perform external Flash resource upgrade, you need to use SPI to drive the external Flash. For detailed operations, refer to *GR533x Datasheet*. The GR533x I/Os used for SPI shall be connected to external Flash pins with DuPont wires.
- GR5405-SK-BASIC has on-board external Flash, but no QSPI. You need to use SPI to drive the external Flash. You can perform Flash read/write/erase operations on demand.

Take GR5515-SK-BASIC as an example. Select **CONFIG I/O**, and then make configurations as shown in Figure 4-32.

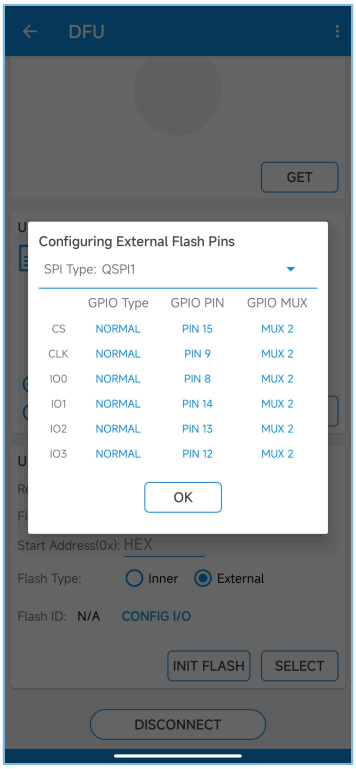

Figure 4-32 Pin configurations for external Flash

2. After configuration, click **UPDATE**. Then, the upgrade progress is shown as follows.

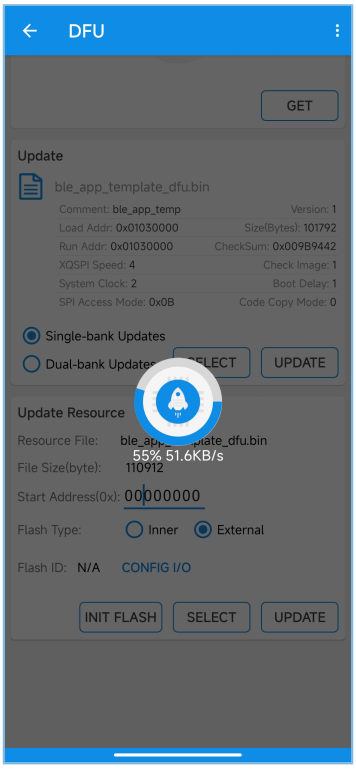

Figure 4-33 Upgrade progress

3. After upgrade completes, the firmware in use will check the data in external Flash. If the data passes the check, “Upgrade completed.” will be prompted at the bottom of GRToolbox.

5 DFU Porting Method

This chapter introduces how to port DFU functionalities by taking the ble_app_template_freertos example project (in SDK_Folder\projects\ble\ble_peripheral\ble_app_template_freertos\Keil_5) as an example, to help users apply the GR5xx DFU scheme to their customized projects. The porting process mainly consists of the following steps:

1. Initialize DFU.
2. Add DFU scheduler.
3. Initialize DFU services (communicating via Bluetooth LE) and receive/process data.
4. Add DFU components.
5. Add OTA Profile.

Detailed steps are provided below.

1. Initialize DFU. The code is in app_periph_init() in *user_periph_setup.c*, as shown below:

```
#include "dfu_port.h"
void app_periph_init(void)
{
    dfu_uart_init();
    dfu_uart_ctrl_pin_init();
    dfu_port_init(uart_send_data, DFU_FW_SAVE_ADDR, &dfu_pro_call);
}
```

dfu_uart_init() is a UART initialization function. For communication via UART, UART needs to be initialized. Take GR551x as an example. The initialization code snippet when data is transmitted via UART1 is as follows:

```
static app_uart_params_t dfu_uart_param;
#define DFU_UART_RX_BUFF_SIZE  0x400
#define DFU_UART_TX_BUFF_SIZE  0x400
static uint8_t s_dfu_uart_rx_buffer[DFU_UART_RX_BUFF_SIZE];
static uint8_t s_dfu_uart_tx_buffer[DFU_UART_TX_BUFF_SIZE];

static void dfu_uart_init(void)
{
    app_uart_tx_buf_t uart_buffer;

    uart_buffer.tx_buf       = s_dfu_uart_tx_buffer;
    uart_buffer.tx_buf_size  = DFU_UART_TX_BUFF_SIZE;

    dfu_uart_param.id                   = APP_UART1_ID;
    dfu_uart_param.init.baud_rate       = APP_UART_BAUDRATE;
    dfu_uart_param.init.data_bits       = UART_DATABITS_8;
    dfu_uart_param.init.stop_bits       = UART_STOPBITS_1;
    dfu_uart_param.init.parity          = UART_PARITY_NONE;
    dfu_uart_param.init.hw_flow_ctrl    = UART_HWCONTROL_NONE;
    dfu_uart_param.init.rx_timeout_mode = UART_RECEIVER_TIMEOUT_ENABLE;
    dfu_uart_param.pin_cfg.rx.type      = APP_UART1_RX_IO_TYPE;
    dfu_uart_param.pin_cfg.rx.pin       = APP_UART1_RX_PIN;
    dfu_uart_param.pin_cfg.rx.mux       = APP_UART1_RX_PINMUX;
    dfu_uart_param.pin_cfg.rx.pull      = APP_UART_RX_PULL;
    dfu_uart_param.pin_cfg.tx.type      = APP_UART1_TX_IO_TYPE;
    dfu_uart_param.pin_cfg.tx.pin       = APP_UART1_TX_PIN;
    dfu_uart_param.pin_cfg.tx.mux       = APP_UART1_TX_PINMUX;
```

```
    dfu_uart_param.pin_cfg.tx.pull      = APP_UART_TX_PULL;

    app_uart_init(&dfu_uart_param, dfu_uart_evt_handler, &uart_buffer);
    app_uart_receive_async(APP_UART1_ID, s_dfu_uart_rx_buffer,
                           sizeof(s_dfu_uart_rx_buffer));
}
```

During upgrade via UART, you not only need to initialize UART TX and RX pins, but also initialize a control pin to wake up the device which is in sleep and to start the stopped DFU task. Therefore, if sleep mechanism is adopted for the device, an AON pin shall be selected as the control pin. Take GR551x as an example. The initialization code snippet is as follows when AON_GPIO_PIN_1 is selected as the control pin:

```
#define DFU_UART_CTRL_PIN    AON_GPIO_PIN_1

void dfu_uart_ctrl_pin_init(void)
{
    app_io_init_t io_init = APP_IO_DEFAULT_CONFIG;

    io_init.pull = APP_IO_PULLUP;
    io_init.mode = APP_IO_MODE_IT_FALLING;
    io_init.pin  = DFU_UART_CTRL_PIN;
    io_init.mux  = APP_IO_MUX;
    app_io_init(APP_IO_TYPE_AON, &io_init);

    app_io_event_register_cb(APP_IO_TYPE_AON, &io_init, app_io_event_handler,
                             "DFU uart ctrl pin interrupt");
}
```

In the application firmware, DFU_FW_SAVE_ADDR represents the start address to copy the firmware in background dual-bank DFU mode. DFU_FW_SAVE_ADDR can be set as follows:

```
#define DFU_FW_SAVE_ADDR (FLASH_START_ADDR + 0x60000)
```

Note:

This value is variable. Do not set it to an address conflicting with that of the App bootloader firmware and the Bank0 firmware.

uart_send_data() is a UART TX function, which needs to be registered for upgrade via UART. This function is defined as follows:

```
static void uart_send_data(uint8_t *data, uint16_t size)
{
    app_uart_transmit_async(APP_UART1_ID, data, size);
}
```

dfu_pro_call() is a callback function to print the progress during upgrade. The code snippet is shown below:

```
static void dfu_program_start_callback(void);
static void dfu_programing_callback(uint8_t pro);
static void dfu_program_end_callback(uint8_t status);

static dfu_pro_callback_t dfu_pro_call =
{
    .dfu_program_start_callback = dfu_program_start_callback,
    .dfu_programing_callback = dfu_programing_callback,
```

```
    .dfu_program_end_callback = dfu_program_end_callback,
};

static void dfu_program_start_callback(void)
{
    APP_LOG_DEBUG("Dfu start program");
}

static void dfu_programing_callback(uint8_t pro)
{
    APP_LOG_DEBUG("Dfu programing---%d%%", pro);
}

static void dfu_program_end_callback(uint8_t status)
{
    APP_LOG_DEBUG("Dfu program end");
    if (0x01 == status)
    {
        APP_LOG_DEBUG("status: successful");
    }
    else
    {
        APP_LOG_DEBUG("status: error");
    }
}
```

2. Create DFU signal amount and a DFU scheduling task.

```
#include "dfu_port.h"

#define DFU_TASK_STACK_SIZE             ( 1024 * 2 )
TaskHandle_t         dfu_task_handle;
SemaphoreHandle_t    xDfuSemaphore;
uint8_t              start_dfu_task_flag = 0;


int main(void)
{
    ……
    xDfuSemaphore = xSemaphoreCreateBinary();
    xTaskCreate(vStartTasks, "create_task", 512, NULL, 0, NULL);
    vTaskStartScheduler();
    for(;;);
}

static void vStartTask(void *arg)
{
    ……
    xTaskCreate(dfu_schedule_task, "dfu_schedule_task", DFU_TASK_STACK_SIZE, NULL,
                configMAX_PRIORITIES - 2,  &dfu_task_handle);
    ……
    vTaskDelete(NULL);
}



static void dfu_schedule_task(void *p_arg)
{
    while (1)
    {
        if (!start_dfu_task_flag)
```

```
        {
            xSemaphoreTake(xDfuSemaphore, portMAX_DELAY);
        }
        dfu_schedule();
    }
}
```

In applications, the current DFU task needs to be paused to reduce power consumption when there is no data interaction between the host and the device. Based on this, GR5xx allows adding a timeout mechanism and signal amount to the DFU task. As shown in the above code, after the DFU task is created, it waits for the signal amount and starts running after receiving the signal amount. The release method of the DFU signal amount varies in different communication modes. For communication via Bluetooth LE, the signal amount is released by sending a DFU Enter command through the Control Point characteristic. For communication via UART, the signal amount is released by triggering an external interrupt through the control pin.

If the device does not receive the data from the host within a certain period of time, it is considered that the current DFU task has ended. In the timeout callback function, reset the start flag of the DFU task to stop executing the DFU task and wait for the DFU signal amount. The timeout mechanism is implemented through the app_timer software timer component. When using app_timer, add `#include "app_timer.h"` to the code. The timer initialization function is defined as follows. This function needs to be called in app_periph_init().

```
static app_timer_id_t  s_cmd_wait_timer_id;

static void dfu_timer_init(void)
{
    app_timer_create(&s_cmd_wait_timer_id, ATIMER_REPEAT, cmd_wait_timeout_handler);
}
```

The timeout callback function is defined as follows:

```
static uint16_t        s_dfu_last_count;
static uint16_t        s_dfu_curr_count;
extern uint8_t         start_dfu_task_flag;

static void cmd_wait_timeout_handler(void* p_arg)
{
    if (s_dfu_last_count < s_dfu_curr_count)
    {
        s_dfu_last_count = s_dfu_curr_count;
    }
    else
    {
        s_dfu_curr_count = 0;
        s_dfu_last_count = 0;
        dfu_cmd_parse_state_reset();
        fast_dfu_state_machine_reset();
        start_dfu_task_flag = 0;
        dfu_timer_stop();
    }
}
```

As shown above, if the host does not send data to the device within a certain period of time, the timeout function will be triggered, which will call dfu_cmd_parse_state_reset(). dfu_cmd_parse_state_reset() is implemented in the SDK and can be called directly; it is to reset the state machine for parsing the DFU command, to facilitate data parsing in the next upgrade. fast_dfu_state_machine_reset() also needs to be called in the

timeout function; it is to reset the state machine of the fast mode, to facilitate data transmission between the host and device in the next upgrade. fast_dfu_state_machine_reset() is currently not implemented in *dfu_port.c* of GR551x SDK V2.0.1 and GR5525 SDK V0.8.0; it needs to be implemented by users by referring to the following code:

```
void fast_dfu_state_machine_reset(void)
{
    s_fast_dfu_mode = 0;
    s_program_end_flag = 0;
    s_fast_dfu_state = FAST_DFU_INIT_STATE;
}
```

Then, reset the start flag of the DFU task and stop the DFU software timer. The code snippet to stop the software timer is as follows:

```
static void dfu_timer_stop(void)
{
    app_timer_stop(s_cmd_wait_timer_id);
}
```

3. Initialize DFU services and receive/process data. To perform DFU in communication mode via Bluetooth LE, initialize DFU-related services in service_init() in *user_app.c* and add `#include "dfu_port.h"` as follows:

```
static void services_init(void)
{
    dfu_service_init(dfu_enter);
}
```

By default, dfu_enter() is defined as empty. Users can add code to the function to trigger the DFU task. In addition, to implement the DFU timeout mechanism, call the timer start function in dfu_enter(). When the device receives the DFU Enter command, it releases the DFU signal amount, sets the start flag of the DFU task, and starts the software timer for the DFU timeout mechanism. The code snippet to be added is as follows:

```
extern SemaphoreHandle_t    xDfuSemaphore;
extern uint8_t         start_dfu_task_flag;

static void dfu_enter(void)
{
    if (!start_dfu_task_flag)
    {
        start_dfu_task_flag = 1;
        xSemaphoreGive(xDfuSemaphore);
        dfu_timer_start();
    }
}
```

The implementation code of dfu_timer_start() is as follows:

```
#define DFU_CMD_WAIT_TIMEOUT            4000

void dfu_timer_start(void)
{
    app_timer_start(s_cmd_wait_timer_id, DFU_CMD_WAIT_TIMEOUT, NULL);
```

```
}
```

If communication mode via UART is adopted, there is no dfu_enter command and no corresponding dfu_enter callback function. In this mode, the DFU task is woken up in the interrupt service function of the control pin initialized in Step 1. The interrupt service function is defined as follows:

```
extern SemaphoreHandle_t    xDfuSemaphore;
extern uint8_t              start_dfu_task_flag;

void app_io_event_handler(app_io_evt_t *p_evt)
{
    app_io_pin_state_t pin_level = APP_IO_PIN_RESET;

    if (p_evt->pin == DFU_UART_CTRL_PIN)
    {
        pin_level = app_io_read_pin(APP_IO_TYPE_AON, DFU_UART_CTRL_PIN);
        if (pin_level == APP_IO_PIN_RESET)
        {
            do
            {
                pin_level = app_io_read_pin(APP_IO_TYPE_AON, DFU_UART_CTRL_PIN);
            } while(pin_level == APP_IO_PIN_SET);

            if (!start_dfu_task_flag)
            {
                start_dfu_task_flag = 1;
                xSemaphoreGive(xDfuSemaphore);
                dfu_timer_start();
            }
        }
    }
}
```

After starting the DFU software timer, you need to call the function that increments count at the data reception location. The function for incrementing count is defined as follows:

```
void dfu_rev_cmd_count(void)
{
    s_dfu_curr_count++;
}
```

When DFU is performed in communication mode via Bluetooth LE, call dfu_rev_cmd_count() at the location of receiving data via Bluetooth LE (in otas_evt_process(otas_evt_t *p_evt) in *dfu_port.c*). The data reception function for DFU is also added here. The code snippet is shown below:

```
static void otas_evt_process(otas_evt_t *p_evt)
{
    switch(p_evt->evt_type)
    {
        ......
        case OTAS_EVT_RX_RECEIVE_DATA:
        dfu_rev_cmd_count();
        if (!s_fast_dfu_mode || s_program_end_flag)
        {
            dfu_ble_receive_data_process(p_evt->p_data, p_evt->length);
        }
        else
```

```
        {
            s_fast_dfu_state = FAST_DFU_PROGRAM_FLASH_STATE;
            fast_dfu_write_data_to_buffer(p_evt->p_data, p_evt->length);
        }
        break;
        ......
    }
}
```

When DFU is performed in communication mode via UART, call dfu_rev_cmd_count() at the location of receiving data via UART (in dfu_uart_evt_handler()). The data reception function for DFU is also added here. The code snippet is shown below:

```
void dfu_uart_evt_handler(app_uart_evt_t * p_evt)
{
    switch(p_evt->evt_type)
    {
    ......
        case APP_UART_EVT_RX_DATA:
        dfu_rev_cmd_count();
        dfu_uart_receive_data_process(s_dfu_uart_rx_buffer, p_evt->data.size);
        app_uart_receive_async(APP_UART1_ID, s_dfu_uart_rx_buffer,
                               DFU_UART_RX_BUFF_SIZE);
        break;
    ......
    }
}
```

Note:

If the DFU task for the device connected with GRToolbox is not running now and you need to obtain firmware information through GRToolbox before clicking **UPDATE**, you can click **Write Ctrl Point** to send a command to wake up the DFU task. For details, refer to "Section 4.3.5 Firmware Upgrade".

4. Add *dfu_port.c* and *otas.c* to gr_libraries and gr_profiles project folders respectively, as shown below.

Note:

dfu_port.c and *otas.c* are in `SDK_Folder\components\libraries\dfu_port` and `SDK_Folder\components\profiles\otas` respectively.

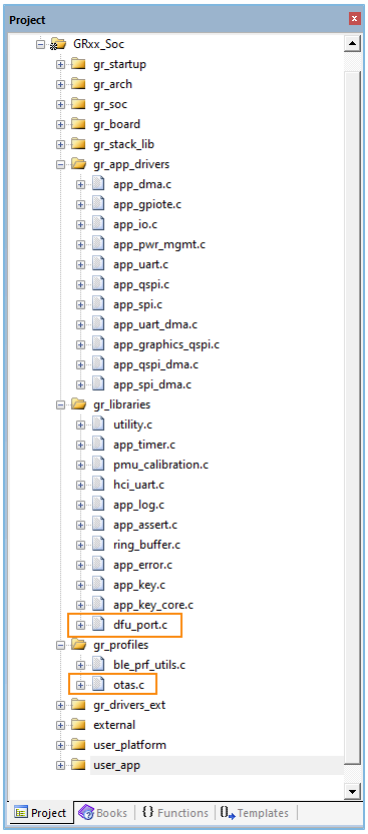

Figure 5-1 Keil project list

5. In *custom.config.h*, set APP_CODE_LOAD_ADDR and APP_CODE_RUN_ADDR to FLASH_START_ADDR + 0x30000, so that they will not overlap the address of the boot firmware. Here, GR551x is taken as an example and APP_CODE_RUN_ADDR and APP_CODE_LOAD_ADDR are set to 0x01030000.

```
// <o> Code load address
// <i> Default:  0x01002000
#define APP_CODE_LOAD_ADDR      0x01030000

// <o> Code run address
// <i> Default:  0x01002000
#define APP_CODE_RUN_ADDR       0x01030000
```

6. To allow the App bootloader firmware to correctly jump to the application firmware, the “COMMENTS” of both needs to be matched. The default “COMMENTS” of the App bootloader firmware is “ble_app_temp”, so you need to define the “COMMENTS” of the ble_app_template_freertos firmware in *custom.config.h* as follows:

```
#define APP_FW_COMMENTS                         "ble_app_temp"
```

6 Upgrade Through DFU Master

This chapter focuses on upgrade through DFU master.

6.1 Introduction to DFU Master

Firmware upgrade can be done not only by using GRToolbox to communicate with the device, but also through DFU master in certain scenarios. GR5xx SDK provides a dfu_master component and the corresponding dfu_master example project. Firmware upgrade through the dfu_master component can be done in communication mode via UART or Bluetooth LE, with the upgrade principles shown in the following:

- The diagram for communication via UART is shown below.

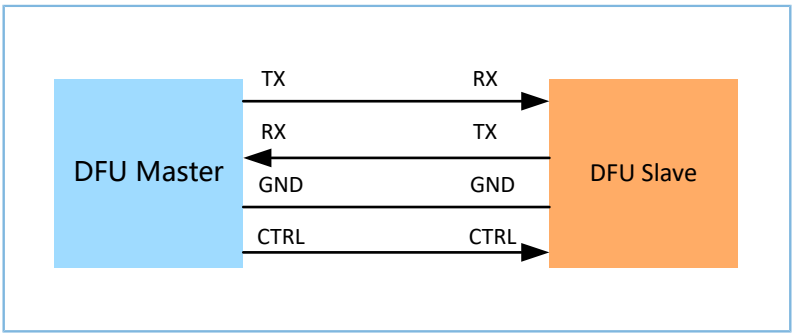

Figure 6-1 Upgrade via UART

- The diagram for communication via Bluetooth LE is shown below.

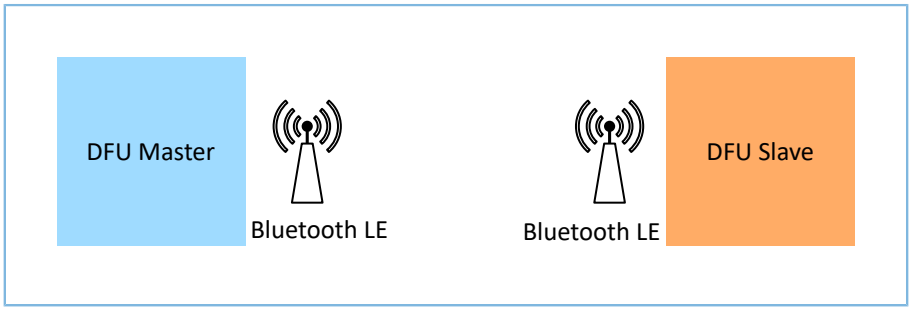

Figure 6-2 Upgrade via Bluetooth LE

Note:

The GR5xx dfu_master scheme supports background dual-bank DFU mode only. In communication mode via UART, firmware can be upgraded in normal mode; in communication mode via Bluetooth LE, firmware can be upgraded in both normal mode and fast mode.

6.2 Cross-platform Porting of DFU Master

The GR5xx dfu_master component is in SDK_`Folder\components\libraries\dfu_master`. Follow the steps below to port the dfu_master component to other platforms:

1. Register the dfu_master-related APIs. The registration code is as follows:

```
static dfu_m_func_cfg_t s_dfu_m_func_cfg =
{
    .dfu_m_get_img_info = dfu_m_get_img_info,
    .dfu_m_get_img_data = dfu_m_get_img_data,
    .dfu_m_send_data    = dfu_uart_data_send,
    .dfu_m_fw_read      = hal_flash_read,
    .dfu_m_event_handler = dfu_m_event_handler,
};
```

Note:

There is no dfu_m_fw_read() provided in GR551x SDK V2.0.1 and GR5525 SDK V0.8.0.

dfu_m_get_img_info() is defined as follows. It is to obtain the image_info of the to-be-transmitted firmware and save the information to the img_info variable. The function needs to be implemented by users according to the location of the image_info.

```
static void dfu_m_get_img_info(dfu_img_info_t *img_info)
{

}
```

dfu_m_get_img_data() is defined as follows. It is to obtain the data with the specified “addr” and “length” of the to-be-transmitted firmware and save the data to p_data.

```
static void dfu_m_get_img_data(uint32_t addr, uint8_t *p_data, uint16_t length)
{

}
```

dfu_uart_data_send() is defined as follows. It is to transmit the p_data with the specified “length” via UART to GRUart. If data is transmitted via Bluetooth LE, the registered API needs to be a Bluetooth LE data transmission function.

```
void dfu_uart_data_send(uint8_t *p_data, uint16_t length)
{

}
```

dfu_m_fw_read() is to read the flag bit in the firmware information of the to-be-transmitted firmware. In the dfu_master example project, the to-be-transmitted firmware is stored in Flash, so hal_flash_read() is registered. Users need to register this API according to the way to read firmware data by the platform in use.

dfu_m_event_handler() is an event callback function after the upgrade command is executed. You can refer to the following to implement this function.

```
static void dfu_m_event_handler(dfu_m_event_t event, uint8_t pre)
{
    switch(event)
    {
        case PRO_START_SUCCESS:
            APP_LOG_DEBUG("Upgrade Start");
            break;

        case PRO_FLASH_SUCCESS:
```

```
            APP_LOG_DEBUG("Upgrade Progress %d%%", pre);
            break;

        case PRO_END_SUCCESS:
            APP_LOG_DEBUG("Upgrade End");
            user_master_status_set(MASTER_IDLE);
            break;
    ......
    }
}
```

2. Define the upgrade parameters before upgrade starts. Parameters vary according to the specific communication mode (via UART and via Bluetooth LE). In communication mode via UART/Bluetooth LE, information and data of the to-be-transmitted firmware need to be obtained. In communication mode via Bluetooth LE, you also need to determine whether to enable the fast mode. In the GR5xx dfu_master example project, the upgrade parameters can be defined according to the following principle:

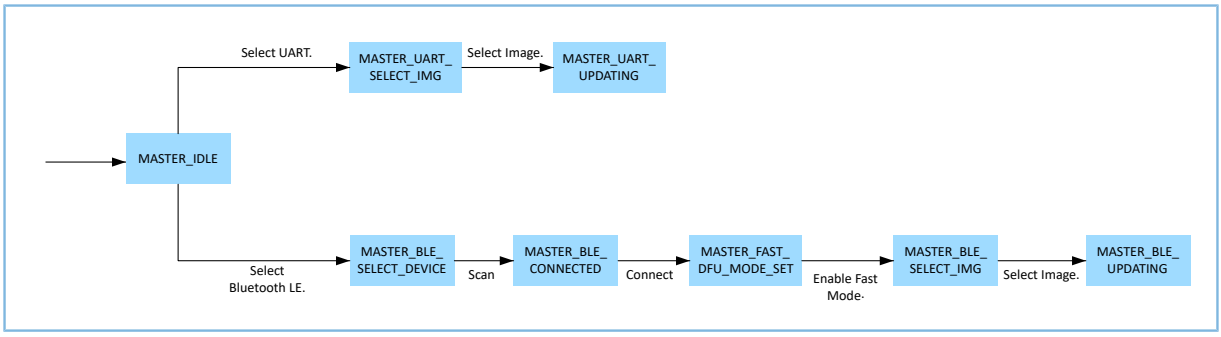

Figure 6-3 State of dfu_master before upgrade

- MASTER_IDLE: idle state
- MASTER_UART_SELECT_IMG: Select firmware in communication mode via UART.
- MASTER_UART_UPDATING: upgrading state in communication mode via UART
- MASTER_BLE_SELECT_DEVICE: Select device in communication mode via Bluetooth LE.
- MASTER_BLE_CONNECTED: connected state in communication mode via Bluetooth LE
- MASTER_FAST_DFU_MODE_SET: enablement state of fast mode
- MASTER_BLE_SELECT_IMG: Select firmware in communication mode via Bluetooth LE.
- MASTER_BLE_UPDATING: upgrading state in communication mode via Bluetooth LE

Users can define parameters before upgrade on their platforms according to the above state diagram.

3. Port the dfu_master component. To port the dfu_master component to other platforms, you need to ensure the header files included in *dfu_master.c* are platform-independent, and the following two header files shall be retained:

```
#include "dfu_master.h"
```

```
#include <string.h>
```

dfu_m_system_info_get() in *dfu_master.c* requires the start address of the target device. The Flash start addresses of GR5xx SoCs are shown in the following table:

Table 6-1 Flash start addresses of GR5xx SoCs

Part Number	FLASH_START_ADDR
GR551x	0x01000000
GR5526	0x00200000
GR5525	0x00200000
GR533x	0x00200000
GR5405	0x00200000

Take GR551x for example. Add the following macro definition to *dfu_master.c*:

```
#define FLASH_START_ADDR    0x01000000
```

For firmware upgrade in communication mode via Bluetooth LE, select whether to enable the fast mode. When the fast mode is enabled, the master will not wait for response from the device after sending the firmware data. After the first frame of data is sent, the second frame will be sent, and so on. In the dfu_master example project, when data is sent in fast mode, ble_send_cplt_flag will be set in the Bluetooth LE send complete event callback after each data frame is sent. The code snippet is as follows:

```
static void otas_c_evt_process(otas_c_evt_t *p_evt)
{
        ……
        switch (p_evt->evt_type)
        {
                ……
                case OTAS_C_EVT_TX_CPLT:
                        if (fast_dfu_mode == 0x00)
                        {
                                dfu_m_send_data_cmpl_process();
                        }
                        else if (fast_dfu_mode == 0x02 && s_program_size ! = 0)
                        {
                                ble_send_cplt_flag = 1;
                        }
                        break;
                ……
        }
        ……
}
```

Note:

fast_dfu_mode in the code snippet above can be set to two values, as described below:

- 0x00: Disable
- 0x02: Enable

You need to ensure the header files included in *dfu_master.h* are platform-independent, and the following two header files shall be retained:

```
#include <stdbool.h>
#include <stdint.h>
```

Add the following boot_info_t type definition to *dfu_master.h* in GR551x SDK V2.0.1 and GR5525 SDK V0.8.0.

```
typedef struct
{
    uint32_t bin_size;
    uint32_t check_sum;
    uint32_t load_addr;
    uint32_t run_addr;
    uint32_t xqspi_xip_cmd;
    uint32_t xqspi_speed: 4;
    uint32_t code_copy_mode: 1;
    uint32_t system_clk: 3;
    uint32_t check_image:1;
    uint32_t boot_delay:1;
    uint32_t is_dap_boot:1;
    uint32_t reserved:21;
} boot_info_t;
```

4. Add the DFU command scheduler and the DFU command parser. The DFU command scheduler needs to be called in a loop during upgrade. The code snippet is as follows:

```
while(1)
{
    dfu_m_schedule(app_dfu_rev_cmd_cb);
}
```

app_dfu_rev_cmd_cb() is the function that increments count for the DFU timeout mechanism. For implementation of the timeout mechanism, refer to "Chapter 5 DFU Porting Method".

The DFU command parser shall be called at the location where the master receives data returned from the to-be-upgraded device. For example, in communication mode via Bluetooth LE, the DFU command parser shall be called at the following location:

```
static void otas_c_evt_process(otas_c_evt_t *p_evt)
{
        ……
        switch (p_evt->evt_type)
        {
                ……
                case OTAS_C_EVT_PEER_DATA_RECEIVE:
                       dfu_m_cmd_prase(p_evt->p_data, p_evt->length);
                       break;
```

```
                    ......
                }
                ......
}
```

6.3 Instructions on Upgrade Through DFU Master

6.3.1 Preparation

Refer to “Section 4.2 Preparation”.

6.3.2 Upgrade via UART

1. Hardware configuration

   Set up the environment for upgrade via DFU master UART. UART1 is used for data transmission for all series of GR5xx SoCs. The TX and RX pins corresponding to different series of GR5xx SoCs are shown in the following table:

Table 6-2 Hardware configuration

Hardware Platform	TX Pin	RX Pin
GR5515-SK-BASIC	GPIO_30	GPIO_26
GR5526-SK-BASIC	GPIO_32	GPIO_33
GR5525-SK-BASIC	GPIO_7	GPIO_6
GR5331-SK-BASIC	GPIO_5	GPIO_6
GR5405-SK-BASIC	AON_GPIO_3	AON_GPIO_2

   In the following, GR5515-SK-BASIC is taken as an example to introduce upgrade steps in detail. GPIO_30 and GPIO_26 are the TX and RX pins of UART1, respectively. The two development boards communicate with each other through their UART1 pins. The schematic diagram is shown as follows.

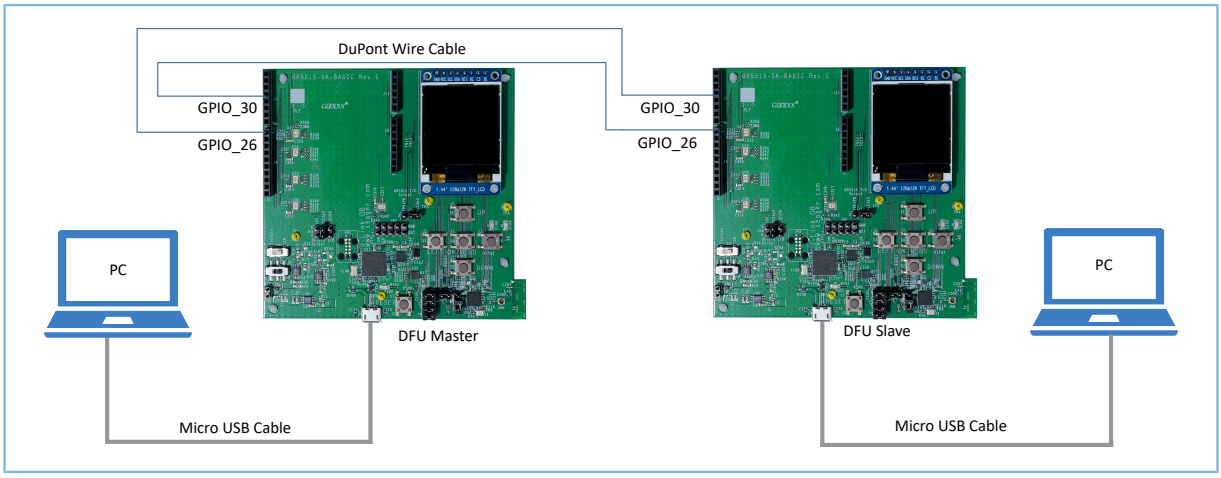

Figure 6-4 Environment for upgrade via UART

2. Steps

(1) For upgrade via UART in RTOS environment, the DFU task might not always run, and the system might be in sleep state. In this case, the CTRL wire shown in Figure 6-1 is needed.

If DFU master needs to use the DFU function, it sends a signal (can be an I/O level transition signal) through the CTRL wire to notify the DFU slave that it needs to run the DFU task.

(2) The default load address and run address of the dfu_master firmware are FLASH_START_ADDR + 0x2000. Download the dfu_master firmware to the device, and store the firmware for upgrade on the device as well.

(3) Refer to "Section 4.3.3 Creating Target Firmware for Upgrade" to create the target firmware for upgrade. The load address and run address of the target firmware are FLASH_START_ADDR + 0x30000. Take GR551x for example. Download the dfu_master firmware and the firmware for upgrade to the device, as shown in Figure 6-5.

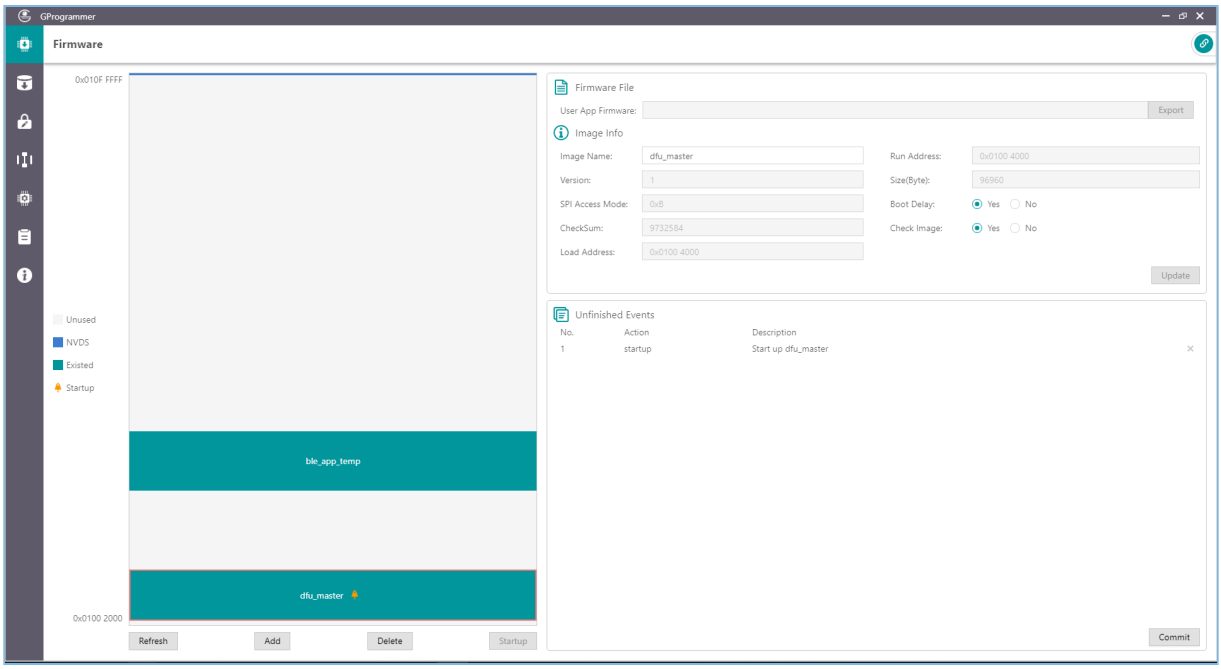

Figure 6-5 Downloading the dfu_master firmware and the firmware for upgrade

(4) The app_bootloader firmware and the ble_app_template_dfu firmware need to be downloaded to the DFU slave, as shown in Figure 6-6. For details, refer to *GProgrammer User Manual*.

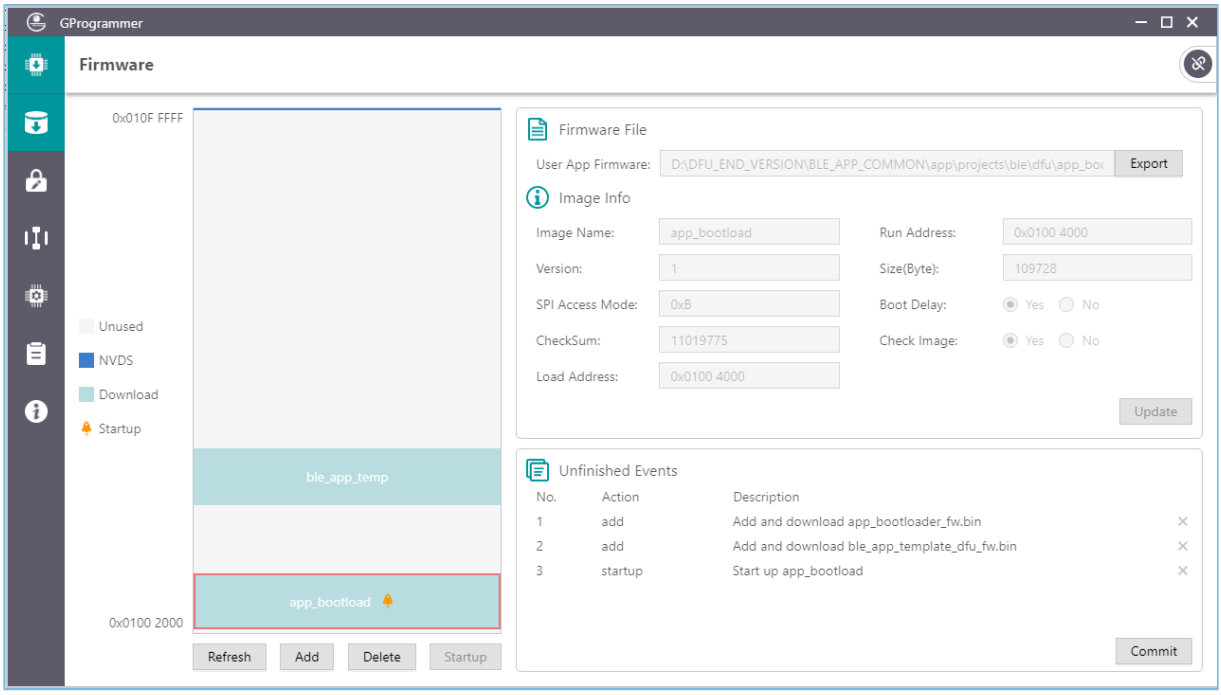

Figure 6-6 Downloading the app_bootloader firmware and the ble_app_template_dfu firmware

(5) After download, connect the DFU master and the DFU slave with DuPont wires (as shown in Table 6-3), and connect UART0 of both DFU master and DFU slave to the PC with Micro USB cables.

Table 6-3 To connect DFU master and DFU slave

DFU Master	**DFU Slave**
UART1	UART1
GPIO_30	GPIO_26
GPIO_26	GPIO_30

(6) Start GRUart and view the DFU master logs (as shown below). You can upgrade firmware via UART or Bluetooth LE. Inputting **1** in the **Tx** pane indicates upgrade via UART.

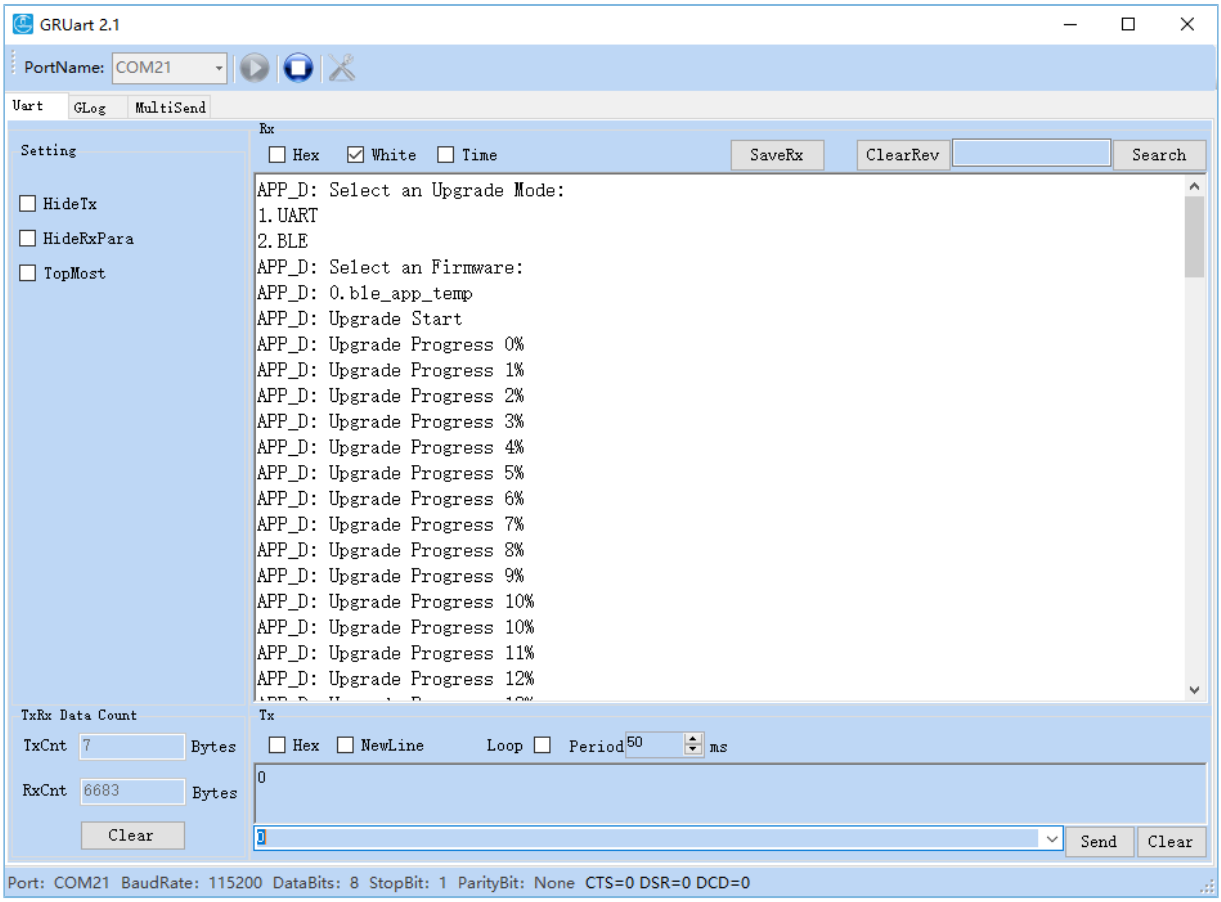

Figure 6-7 Logs for upgrade via UART

(7) Select the firmware version for upgrade, and enter the upgrade mode. Then, the upgrade progress will be displayed on GRUart.

6.3.3 Upgrade via Bluetooth LE

Control Point will be used to wake up the system if the firmware is upgraded in RTOS environment, the DFU task is not running, and the system is in sleep state. Then, the system will execute the DFU task.

The steps for upgrade via Bluetooth LE and via UART are almost the same, except that one more step is added for upgrade via Bluetooth LE. The specific steps are as follows:

1. Select **BLE**.
2. Select **Normal DFU** or **Fast DFU**.
3. Select the firmware version for upgrade, and enter the upgrade mode. Then, the upgrade progress will be displayed on GRUart.

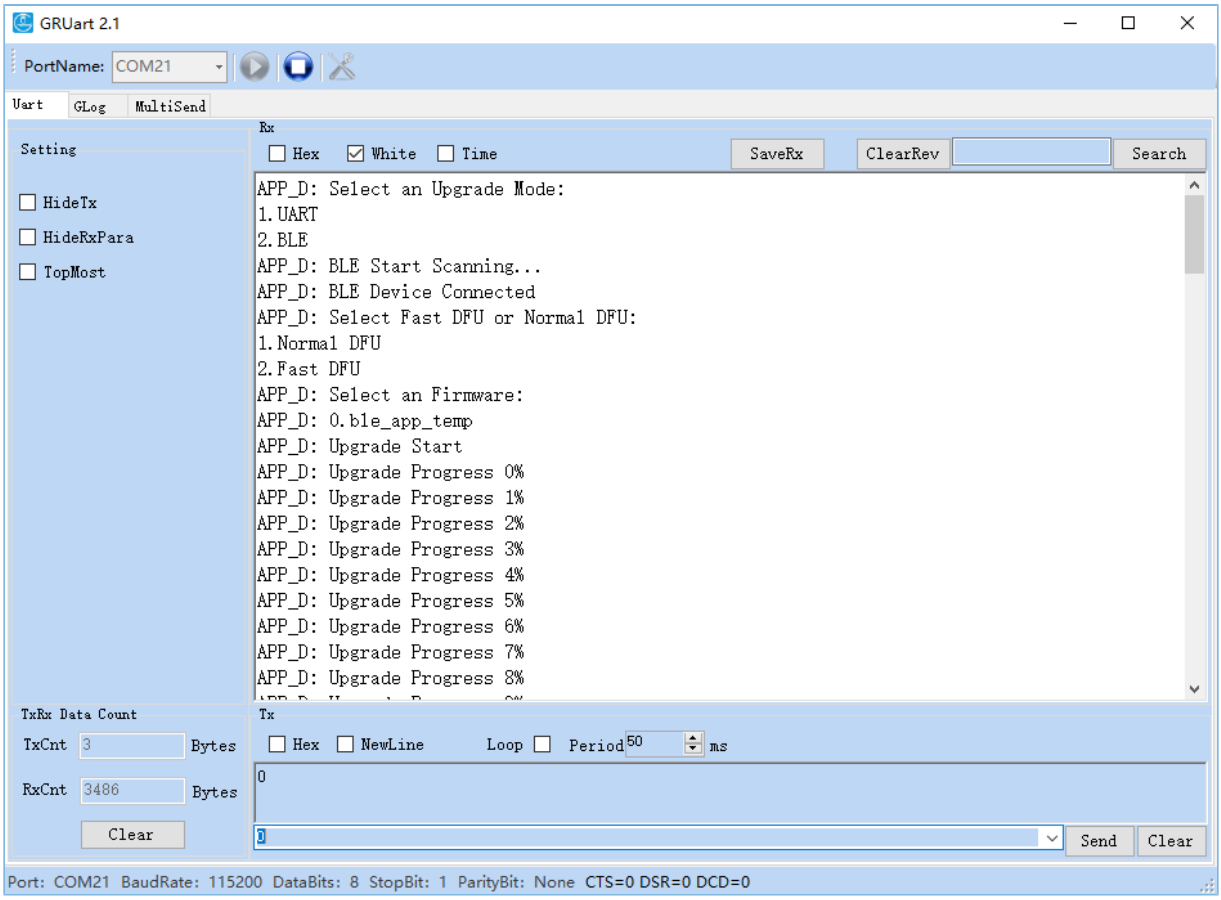

Figure 6-8 Logs for upgrade via Bluetooth LE

7 Considerations

This chapter provides information worthy of particular attention during DFU.

7.1 Deinitializing Peripherals used in App Bootloader Before Jumping from App Bootloader to App Firmware

- Reason

  The following operations are required before jumping from App bootloader to App firmware.

  1. Disable the interrupt.
  2. Clear the pending bit.
  3. Deinitialize peripherals used in App bootloader.

- Solution

  Operations 1 and 2 have been implemented by GR5xx SDK. Therefore, to avoid firmware jump failure or abnormal power consumption of the application firmware, it is necessary to deinitialize the peripherals used by the App bootloader before jumping.

7.2 Setting the DFU Task Stack Size of Application Firmware in RTOS Environment According to Specific GR5xx SoCs

- Reason

  DFU implementation for GR551x series is different from that for other SoC series, so a different task stack size is required.

- Solution

  At least 6 KB needs to be allocated for DFU task stack of GR551x, and at least 1 KB for other SoC series.

8 Appendix: DFU Communication Protocols

The firmware upgrade between the host and the device is based on DFU communication protocols.

8.1 Basic Frame

The basic frame defines the lowest-level data packet structure in communication. The application data packet protocol is based on the basic frame, in the “Data” field of the basic frame. If the basic frame length exceeds the maximum payload of link communication, the host needs to segment the frame for transmission. After receiving the correct frame header and data length, the device starts processing the data.

8.1.1 Frame Structure

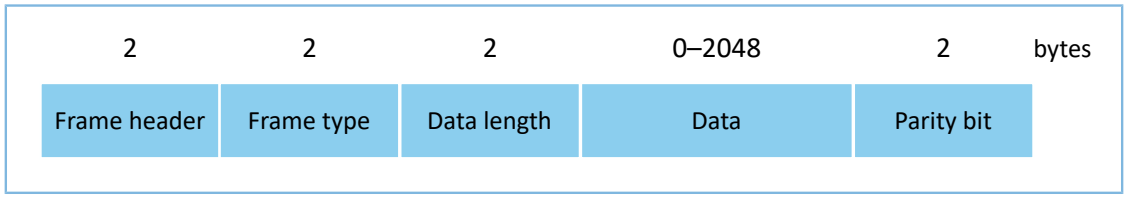

Figure 8-1 Frame structure

- Frame header: the start of a frame, represented by 0x47 and 0x44 which are the ASCII code values of characters ‘G’ and ‘D’
- Frame type: used to distinguish data types in the “Data” field
- Data length: length of data in the “Data” field
- Data: data with configurable length; maximum length: 2048 bytes
- Parity bit: 16-bit checksum for frame type, data length, and data

8.1.2 Byte Order

The little-endian mode is adopted for the “Data” field of the basic frame. The low byte data shall be stored at low addresses in Flash, and the high byte data at high addresses.

8.2 Appendix: DFU Command Set

DFU commands are delivered by the host and received by the device. The DFU command set is listed as follows.

Table 8-1 DFU command description

Command	Command Code	Description
Get Info	0x01	The host gets the system information and DFU version of the SoC.
Operate System Info	0x27	The host sends this command to handle data in the System Configuration Area of the device, including reading and updating the data.
DFU Mode Set	0x41	The host sets the DFU mode on the device.
DFU Firmware Info Get	0x42	The host gets firmware information on the device.

Command	Command Code	Description
Program Start	0x23	The host sends Image Info and information about whether to enable the fast mode when programs are downloaded to the device.
Program Flash	0x24	The host writes firmware data to the device.
Program End	0x25	The host sends this command to notify the device that the programming data has been sent.
Config External Flash	0x2A	Configure external Flash.
Get Flash Information	0x2B	Get Flash information.

8.2.1 Get Info Command

The Get Info command is used to get the system information of the SoC; after receiving the command, the device will send information on ID, Flash, RAM, and stack version of the SoC to the host.

8.2.1.1 Data Sent from the Host

Table 8-2 Format of data sent through the Get Info command

Byte No.	Description	Valid Value	Remarks
0–1	Frame header	0x4744	Represented by 0x47 and 0x44 which are the ASCII code values of characters 'G' and 'D'
2–3	Frame type	0x01	Get Info command
4–5	Data length	0	No data content
6–7	Checksum	0x00–0xFF	16-bit checksum for frame type and data length

8.2.1.2 Response Data from the Device

Table 8-3 Format of data replied through the Get Info command

Byte No.	Description		Valid Value	Remarks
0–1	Frame header		0x4744	Represented by 0x47 and 0x44 which are the ASCII code values of characters 'G' and 'D'
2–3	Frame type		0x01	Get Info command
4–5	Data length		1 or 9	If getting information fails, the data field only contains the response.
6	Data content	Response	0x01/0x02	• 0x01: Getting system information succeeds. • 0x02: Getting system information fails.
7		Stack Major	0x00–0xff	Main version number
8		Stack Minor	0x00–0xff	Minor version number
9–10		Stack Build	0x00–0xff	Build number

Byte No.	Description		Valid Value	Remarks
11–14		Stack SVN	0x00–0xff	SVN number
15		SDK Major	0x00–0xff	Main version number
16		SDK Minor	0x00–0xff	Minor version number
17–18		SDK Build	0x00–0xff	Build number
19–22		SDK SVN	0x00–0xff	SVN number
23		DFU Version	0x02 or invalid data bit	Get the DFU version. • 0x02: DFU schemes introduced in this document • Invalid data bit: DFU schemes in earlier versions
24–25	Checksum		0x00–0xFF	16-bit checksum for frame type, data length, and data content

8.2.2 Operate System Info Command

The host sends this command to handle data in the System Configuration Area of the device, including reading and updating the data.

8.2.2.1 Data Sent from the Host

Table 8-4 Format of data sent through the Operate System Info command

Byte No.	Description		Valid Value	Remarks
0–1	Frame header		0x4744	Represented by 0x47 and 0x44 which are the ASCII code values of characters ‘G’ and ‘D’
2–3	Frame type		0x27	Operate System Info command
4–5	Data length		N	Length of data in the “Data” field
6	Data content	Operating command	0x00/0x01	• 0x00: Get the data in the System Configuration Area. • 0x01: Update the data in the System Configuration Area.
7–10		Start address	0x00–0xFF	A valid address in the System Configuration Area
11–12		Data length	0x00–0xFF	Length of data to be read from the start address; it cannot exceed the System Configuration Area.
13–N		Content update	0x00–0xFF	If it is a data get command, there is no need to update the content segment.
N+1 to N+2	Checksum		0x00–0xFF	16-bit checksum for frame type, data length, and data content

Note:

N in the table above indicates variable length for the “Data” field:

- For a data update command, N ranges from 14 bytes to 1036 bytes.
- For a data read command, the “Data” field has a fixed length of 7 bytes.

8.2.2.2 Response Data from the Device

Table 8-5 Format of data replied through the Operate System Info command

Byte No.	Description		Valid Value	Remarks
0–1	Frame header		0x4744	Represented by 0x47 and 0x44 which are the ASCII code values of characters ‘G’ and ‘D’
2–3	Frame type		0x27	Operate System Info command
4–5	Data length		N	Length of data in the “Data” field
6	Data content	Response	0x01/0x02	• 0x01: Data read succeeds. • 0x02: Data read fails.
7		Operating command	0x00/0x10/0x01/0x11	• 0x00: Read the data in the System Configuration Area (unencrypted SoCs). • 0x10: Read the data in the System Configuration Area (encrypted SoCs). • 0x01: Update the data in the System Configuration Area (unencrypted SoCs). • 0x11: Update the data in the System Configuration Area (encrypted SoCs).
8–11		Start address	0x00–0xFF	If it is a data update command, this data segment is invalid.
12–13		Data length	0x00–0xFF	
14–N		Data	0x00–0xFF	
N+1 to N+2	Checksum		0x00–0xFF	16-bit checksum for frame type, data length, and data content

8.2.3 DFU Mode Set Command

This command is used to set the DFU mode on GRToolbox. Two DFU modes are available: background dual-bank DFU mode and non-background single-bank DFU mode.

8.2.3.1 Data Sent from the Host

Table 8-6 Format of data sent through the DFU Mode Set command

Byte No.	Description	Valid Value	Remarks
0–1	Frame header	0x4744	Represented by 0x47 and 0x44 which are the ASCII code values of characters 'G' and 'D'
2–3	Frame type	0x41	DFU Mode Set command
4–5	Data length	0x01	1-byte data content
6	Data content	0x01 0x02	• 0x01: background dual-bank DFU mode • 0x02: non-background single-bank DFU mode
7–8	Checksum	0x00–0xFF	16-bit checksum for frame type, data length, and data content

8.2.3.2 Response Data from the Device

The device will not respond to the DFU Mode Set command.

8.2.4 DFU Firmware Info Get Command

This command allows the host to get firmware information stored in the APP Info area of the firmware.

8.2.4.1 Data Sent from the Host

Table 8-7 Format of data sent through the DFU Firmware Info Get command

Byte No.	Description	Valid Value	Remarks
0–1	Frame header	0x4744	Represented by 0x47 and 0x44 which are the ASCII code values of characters 'G' and 'D'
2–3	Frame type	0x42	DFU Firmware Info Get command
4–5	Data length	0	No data content
6–7	Checksum	0x00–0xFF	16-bit checksum for frame type and data length

8.2.4.2 Response Data from the Device

Table 8-8 Format of data replied through the DFU Firmware Info Get command

Byte No.	Description		Valid Value	Remarks
0–1	Frame header		0x4744	Represented by 0x47 and 0x44 which are the ASCII code values of characters 'G' and 'D'
2–3	Frame type		0x42	DFU Firmware Info Get command
4–5	Data length			Response data length
6	Data content	Response	0x01 0x02	• 0x01: Getting information succeeds. • 0x02: Getting information fails.

Byte No.	Description		Valid Value	Remarks
7–10		Copy_load_addr		Storage start address for firmware upgrade
11		Run_position	0x00 0x01	• 0x00: The App bootloader firmware is running. • 0x01: The App firmware is running.
12–59		Image_Info		Firmware information stored in the APP Info area
59–61	Checksum		0x00–0xFF	16-bit checksum for frame type, data length, and data content

8.2.5 Program Start Command

The Program Start command is used to send firmware information and information about whether to enable the fast mode.

8.2.5.1 Data Sent from the Host

Table 8-9 Format of data sent through the Program Start command

Byte No.	Description	Valid Value	Remarks
0–1	Frame header	0x4744	Represented by 0x47 and 0x44 which are the ASCII code values of characters 'G' and 'D'
2–3	Frame type	0x23	Program Start command
4–5	Data length	N	• If the Program Start command targets firmware, the "Data" field has 41 bytes, including 1 byte for Flash type and 40 bytes for Image Info. • If the Program Start command targets data, the "Data" field has 9 bytes, including 1 byte for Flash type, 4 bytes for start address, and 4 bytes for data content.
6	Upgrade type	0x00 0x01 0x02 0x03 0x10 0x20 0x12 0x22	• 0x00: Upgrade unencrypted and unsigned firmware in normal mode in internal Flash. • 0x10: Upgrade unencrypted and signed firmware in normal mode in internal Flash. • 0x20: Upgrade encrypted and signed firmware in normal mode in internal Flash. • 0x02: Upgrade unencrypted and unsigned firmware in fast mode in internal Flash. • 0x12: Upgrade unencrypted and signed firmware in fast mode in internal Flash. • 0x22: Upgrade encrypted and signed firmware in fast mode in internal Flash. • 0x01: Upgrade resources in normal mode in external Flash. • 0x03: Upgrade resources in fast mode in external Flash.
7–N	Data content	0x00–0xFF	Data content to be written to the device
N+1 to N+2	Checksum	0x00–0xFF	16-bit checksum for frame type, data length, and data content

8.2.5.2 Response Data from the Device

The response data from the device varies according to the specific upgrade mode.

- The response data frame from the device in normal mode is shown as follows.

Table 8-10 Format of data replied through the Program Start command in normal mode

Byte No.	Description	Valid Value	Remarks
0–1	Frame header	0x4744	Represented by 0x47 and 0x44 which are the ASCII code values of characters 'G' and 'D'
2–3	Frame type	0x23	Program Start command
4–5	Data length	0x01	Length of data in the "Data" field
6	Response	0x01/0x02	◦ 0x01: Succeeded. ◦ 0x02: Failed.
7–8	Checksum	0x00–0xFF	16-bit checksum for frame type, response, and data length

- The response data frame from the device in fast mode is shown as follows.

Table 8-11 Format of data replied through the Program Start command in fast mode

Byte No.	Description	Valid Value	Remarks
0–1	Frame header	0x4744	Represented by 0x47 and 0x44 which are the ASCII code values of characters 'G' and 'D'
2–3	Frame type	0x23	Program Start command
4–5	Data length	0x04	Length of data in the "Data" field
6	Response	0x01/0x02	◦ 0x01: Succeeded. ◦ 0x02: Failed.
7	Erasing state	0x00 0x01 0x02 0x03 0x04 0x05 0x06	◦ 0x00: The start address of the Flash area to be erased is not 4 KB aligned. ◦ 0x01: Start erasing. ◦ 0x02: in normal erasing operation ◦ 0x03: Erasing completes and data can be delivered. ◦ 0x04: The Flash area to be erased overlaps the current running area. ◦ 0x05: Erasing fails. ◦ 0x06: invalid Flash area to be erased
8–9	Number of erased pages	0x00–0xFF	Number of erased pages
10–11	Checksum	0x00–0xFF	16-bit checksum for frame type, response, data length, erasing state, and number of erased pages

8.2.6 Program Flash Command

The Program Flash command is used for DFU in normal mode only. For DFU in fast mode, the data is transmitted directly when you write data to the firmware, instead of by data frames.

8.2.6.1 Data Sent from the Host

Table 8-12 Format of data sent through the Program Flash command

Byte No.	Description		Valid Value	Remarks
0–1	Frame header		0x4744	Represented by 0x47 and 0x44 which are the ASCII code values of characters 'G' and 'D'
2–3	Frame type		0x24	Program Flash command
4–5	Data length		N	Max. length: 2 KB
6	Data content	Program type	0x00 0x01 0x02 0x10 0x11 0x12	• 0x00: Store data after erasing the internal Flash page at a specified address. • 0x01: Store data in internal Flash according to Image Info sent by the Program Start command. • 0x02: Call a Flash write API to write data to internal Flash. • 0x10: Store data after erasing the external Flash page at a specified address. • 0x11: Store data in external Flash according to Image Info sent by the Program Start command. • 0x12: Call a Flash write API to write data to external Flash.
7–10		Start address	0x00–0xFF	Valid Flash address of the device
11–12		Data length	0x00–0xFF	The maximum data length is recommended to be 1024 bytes.
13–N		Data	0x00–0xFF	Data sent from the host to the device
N+1 to N+2	Checksum		0x00–0xFF	16-bit checksum for frame type, data length, and data content

8.2.6.2 Response Data from the Device

Table 8-13 Format of data replied through the Program Flash command

Byte No.	Description	Valid Value	Remarks
0–1	Frame header	0x4744	Represented by 0x47 and 0x44 which are the ASCII code values of characters 'G' and 'D'
2–3	Frame type	0x24	Program Flash command
4–5	Data length	0x01	1 byte (for response)
6	Data content	0x01/0x02	• 0x01: Succeeded.

Byte No.	Description	Valid Value	Remarks
			• 0x02: Failed.
7–8	Checksum	0x00–0xFF	16-bit checksum for frame type, data length, and data content

8.2.7 Writing Firmware in Fast Mode

8.2.7.1 Data Sent from the Host

In fast mode, when each frame of data is written during DFU, the firmware data is sent directly in the maximum transmission unit (MTU) of Bluetooth, so no data frame format is needed.

8.2.7.2 Response Data from the Device

Table 8-14 Format of data replied for writing firmware in fast mode

Byte No.	Description	Valid Value	Remarks
0–1	Frame header	0x4744	Represented by 0x47 and 0x44 which are the ASCII code values of characters 'G' and 'D'
2–3	Frame type	0xFF	Fast DFU Writing Data to Flash Complete command
4–5	Data length	0x01	Length of data in the "Data" field
6	Response	0x01/0x02	• 0x01: Succeeded. • 0x02: Failed.
7–8	Checksum	0x00–0xFF	16-bit checksum for frame type, data length, and response

8.2.8 Program End Command

The host sends this command to notify the device that the programming data has been sent.

8.2.8.1 Data Sent from the Host

Table 8-15 Format of data sent through the Program End command

Byte No.	Description		Valid Value	Remarks
0–1	Frame header		0x4744	Represented by 0x47 and 0x44 which are the ASCII code values of characters 'G' and 'D'
2–3	Frame type		0x25	Program End command
4–5	Data length		0x05	Length of data in the "Data" field
6	Data content	Reset type flag	0x00 0x01 0x02 0x12	• 0x00: Store the firmware Image Info in the Img Info area of SCA, and do not run the firmware after reset. • 0x01: Store the firmware Image Info in the APP Info area in Flash, and run the firmware immediately after reset. • 0x02: Download data to internal Flash without operations to the APP Info and SCA areas.

Byte No.	Description		Valid Value	Remarks
				• 0x12: Download data to external Flash.
7–10		Checksum of programming file	0x00–0xFF	Checksum of transmitted file
11–12	Checksum		0x00–0xFF	16-bit checksum for frame type, data length, and data content

8.2.8.2 Response Data from the Device

The response data from the device will be different depending on whether DFU is performed in fast mode. In fast mode, an additional firmware data checksum will be replied to help the host to verify if the firmware being sent is correct. The data frame format is shown below:

Table 8-16 Format of data replied through the Program End command

Byte No.	Description	Valid Value	Remarks
0–1	Frame header	0x4744	Represented by 0x47 and 0x44 which are the ASCII code values of characters 'G' and 'D'
2–3	Frame type	0x25	Program End command
4–5	Data length	0x04	Length of data in the "Data" field
6	Response	0x01/0x02	• 0x01: Succeeded. • 0x02: Failed.
7–10	Firmware data checksum (DFU in fast mode)	0x00–0xFF	Return the checksum calculated by the firmware.
11–12	Checksum	0x00–0xFF	16-bit checksum for frame type, response, data length, and data content

8.2.9 Config External Flash Command

The host sends this command to configure the external Flash SPI of the device.

8.2.9.1 Data Sent from the Host

Table 8-17 Format of data sent through the Config External Flash command

Byte No.	Description	Valid Value	Remarks
0–1	Frame header	0x4744	Represented by 0x47 and 0x44 which are the ASCII code values of characters 'G' and 'D'
2–3	Frame type	0x2A	Config External Flash command
4–5	Data length	N	Length of data in the "Data" field
6	External Flash type	0x01/0x02	• 0x01: SPI Flash • 0x02: QSPI Flash

Byte No.	Description		Valid Value	Remarks
7–9	Data content	CS IO TYPE	00–04	Select GPIO type.
		CS PIN	00–15	Select GPIO pins.
		CS IO MUX	00–09	Select Pin Mux.
10–12		CLK IO TYPE	00–03	Select GPIO type.
		CLK PIN	00–31	Select GPIO pins.
		CLK IO MUX	00–09	Select Pin Mux.
13–15		MOSI(IO0) IO TYPE	00–03	Select GPIO type.
		MOSI(IO0) PIN	00–31	Select GPIO pins.
		MOSI(IO0)IO MUX	00–09	Select Pin Mux.
16–18		MISO(IO1)IO TYPE	00–03	Select GPIO type.
		MISO(IO1) PIN	00–31	Select GPIO pins.
		MISO(IO1)IO MUX	00–09	Select Pin Mux.
19–21		IO2IO TYPE	00–03	Select GPIO type; valid for QSPI only.
		IO2 PIN	00–31	Select GPIO pins; valid for QSPI only.
		IO2 IO MUX	00–09	Select Pin Mux; valid for QSPI only.
22–24		IO3IO TYPE	00–03	Select GPIO type; valid for QSPI only.
		IO3 PIN	00–31	Select GPIO pins; valid for QSPI only.
		IO3 IO MUX	00–09	Select Pin Mux; valid for QSPI only.
25		QSPI ID	00–02	QSPI module ID, valid for QSPI only
26–27	Checksum		0x00–0xFF	16-bit checksum for frame type, data length, external Flash type, and data content

8.2.9.2 Response Data from the Device

Table 8-18 Format of data replied through the Config External Flash command

Byte No.	Description	Valid Value	Remarks
0–1	Frame header	0x4744	Represented by 0x47 and 0x44 which are the ASCII code values of characters ‘G’ and ‘D’
2–3	Frame type	0x2A	Initialize External Flash command
4–5	Data length	0x01	1 byte (for response)
6	Data content	0x01/0x02	• 0x01: Initialization succeeds.

Byte No.	Description	Valid Value	Remarks
			• 0x02: Initialization fails.
7–8	Checksum	0x00–0xFF	16-bit checksum for frame type, data length, and data content

8.2.10 Get Flash Information Command

The host sends this command to get internal/external Flash information from the device, including Flash ID and Flash size. External Flash size is available through the Serial Flash Discoverable Parameters (SFDP) protocol. For all Flash chips supporting the SFDP protocol, the host can get the Flash size by sending this command.

8.2.10.1 Data Sent from the Host

Table 8-19 Format of data sent through the Get Flash Information command

Byte No.	Description	Valid Value	Remarks
0–1	Frame header	0x4744	Represented by 0x47 and 0x44 which are the ASCII code values of characters 'G' and 'D'
2–3	Frame type	0x2B	Get external Flash ID.
4–5	Data length	0x01	1 byte
6	Flash type	0x00/0x01	• 0x00: internal Flash • 0x01: external Flash
7–8	Checksum	0x00–0xFF	16-bit checksum for frame type, data length, and Flash type

8.2.10.2 Response Data from the Device

Table 8-20 Format of data replied through the Get Flash Information command

Byte No.	Description		Valid Value	Remarks
0–1	Frame header		0x4744	Represented by 0x47 and 0x44 which are the ASCII code values of characters 'G' and 'D'
2–3	Frame type		0x2B	Get external Flash ID.
4–5	Data length		0x09	1 byte (for response)
6	Response		0x01/0x02	• 0x01: Data read succeeds. • 0x02: Data read fails.
7–14	Content	Flash ID	0x00–0xFF	Flash ID
		Flash size	0x00–0xFF	Flash size
14–15	Checksum		0x00–0xFF	16-bit checksum for frame type, data length, response, and data content