GR551x Developer Guide

Version: 2.6

Release Date: 2023-01-19

Shenzhen Goodix Technology Co., Ltd.

Trademarks and Permissions

Disclaimer

Shenzhen Goodix Technology Co., Ltd.

Headquarters: Floor 12-13, 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 the Software Development Kit (SDK) of the Goodix GR551x Bluetooth Low Energy (Bluetooth LE) System-on-Chip (SoC) and Keil for program development and debugging, to help you quickly get started with secondary development of Bluetooth LE applications.

Audience

This document is intended for:

- GR551x user
- GR551x developer
- GR551x tester
- Technical writer

Release Notes

This document is the fourteenth release of *GR551x Developer Guide*, corresponding to GR551x SoC series.

Revision History

Version	Date	Description
1.0	2019-12-08	Initial release
1.3	2020-03-16	Updated the descriptions in “RAM Power Management” and “Output Debug Logs”.
1.5	2020-05-30	Updated the models of GR551x SoCs in “Introduction” and the descriptions in “Memory Mapping”, “GR551x SDK Directory Structure”, and “Development and Debugging with GR551x SDK”.
1.6	2020-06-30	Optimized the SCA layout in “SCA”, and updated “RAM Power Management” and “Output Debug Logs”.
1.7	2020-08-30	Introduced GR5515I0ND in the GR551x series in “Introduction”.
1.8	2020-09-25	• Updated the RAM layout in Mirror Mode and added descriptions about how to determine the start address of the RAM segment “App Code Execution Region” in “RAM Layout in Mirror Mode”; • Updated the path of *GR551x_8MB_Flash.FLM* in the GR551x SDK in “Download .hex Files to Flash” and added a note for the “No Cortex-M SW Device Found” error occurring during .hex file download.
1.9	2020-11-25	• Optimized descriptions about NVDS in “NVDS” and updated the value range of idx to 0x0000–0x3FFF. • Added configuration parameters in “Configure custom_config.h”, including CHIP_TYPE, CFG_MAX_LEG_EXT_ADVS, CFG_MAX_PER_ADVS, CFG_MAX_SCAN, and CFG_MAX_PER_ADV_SYNC.

Version	Date	Description
2.0	2021-01-05	Subdivided “Output Debug Logs” into “Parameter Configuration”, “Module Initialization”, and “Application”.
2.1	2021-01-27	Optimized descriptions in “Output Debug Logs” and “Configure custom_config.h”.
2.2	2021-04-15	Updated the firmware name in “Generate Firmware”.
2.3	2021-06-22	Updated the parameters of *custom_config.h* in “Configure custom_config.h”.
2.4	2021-08-20	• Added GR5515IENDU and GR5515I0NDA SoCs in “Introduction”. • Updated parameters in *custom_config.h* in “Configure custom_config.h”.
2.5	2022-02-20	• Added GR5513BENDU to the “Introduction” chapter. • Modified relevant parameters in the "Configuring custom_config.h" section. • Modified the steps to configure After Build in the "Configuring After Build" section. • Modified the firmware files in the "Generating Firmware" section. • Modified the steps to download .hex files to SoC Flash in the "Downloading .hex Files to Flash" section. • Modified relevant descriptions in the "Debugging" section according to SDK changes. • Added two chapters: "Development and Debugging with GR551x SDK in GCC" and "Development and Debugging with GR551x SDK in IAR".
2.6	2023-01-19	• Deleted the GR5515I0ND SoC. • Updated the GR5515RGBD status from "NRND" to "Active".

Contents

Preface..I

1 Introduction..1

1.1 GR551x SDK..1

1.2 BLE Stack..1

2 GR551x Bluetooth LE Software Platform..4

2.1 Hardware Architecture..4

2.2 Software Architecture..5

2.3 Memory Mapping..6

2.4 Flash Memory Mapping..7

2.4.1 SCA..8

2.4.2 NVDS..10

2.5 RAM Mapping..11

2.5.1 RAM Layout in XIP Mode..12

2.5.2 RAM Layout in Mirror Mode..13

2.5.3 RAM Power Management..14

2.6 GR551x SDK Directory Structure..15

3 Bootloader..18

4 Development and Debugging with GR551x SDK in Keil..20

4.1 Installing Keil MDK..20

4.2 Installing GR551x SDK..21

4.3 Building a Bluetooth LE Application..21

4.3.1 Preparing ble_app_example..21

4.3.2 Configuring a Project..25

4.3.2.1 Configuring *custom_config.h*..25

4.3.2.2 Configuring Memory Layout..30

4.3.2.3 Configuring After Build..31

4.3.3 Adding User Code..32

4.3.3.1 Modifying the main() Function..32

4.3.3.2 Implementing Bluetooth LE Business Logics..33

4.3.3.3 Scheduling BLE_Stack_IRQ, BLE_SDK_IRQ, and Applications..35

4.4 Generating Firmware..36

4.5 Downloading .hex Files to Flash..37

4.6 Debugging..40

4.6.1 Configuring the Debugger..40

4.6.2 Starting Debugging..42

4.6.3 Debugging in Mirror Mode..43

4.6.4 Outputting Debug Logs..44

4.6.4.1 Module Initialization... 44
4.6.4.2 Application...45
4.6.5 Debugging with GRToolbox...47

5 Development and Debugging with GR551x SDK in GCC..48

6 Development and Debugging with GR551x SDK in IAR...49

7 Glossary.. 50

1 Introduction

The Goodix GR551x System-on-Chip (SoC) is a single-mode low-power SoC that supports Bluetooth 5.1. It can be configured as a Broadcaster, an Observer, a Central, or a Peripheral, and supports the combination of all the above roles, making it an ideal choice for Internet of Things (IoT) and smart wearable devices.

Based on ARM® Cortex®-M4F CPU core, the GR551x SoC integrates the Bluetooth 5.1 Protocol Stack, a 2.4 GHz RF transceiver, on-chip programmable flash memory, RAM, and multiple peripherals.

GR551x SoCs are available in multiple packages (see Table 1-1) that meet your diverse project demands.

Table 1-1 GR551x series

GR551x Series	**GR5515IGND**	**GR5515IENDU**	**GR5515I0NDA**	**GR5515RGBD**	**GR5515GGBD**	**GR5513BEND**	**GR5513BENDU**
CPU	Cortex®-M4F	Cortex®-M4F	Cortex®-M4F	Cortex®-M4F	Cortex®-M4F	Cortex®-M4F	Cortex®-M4F
RAM	256 KB	256 KB	256 KB	256 KB	256 KB	128 KB	128 KB
Flash	1 MB	512 KB	N/A	1 MB	1 MB	512 KB	512 KB
Package (mm)	QFN56 (7 x 7 x 0.75)	QFN56 (7 x 7 x 0.75)	QFN56 (7 x 7 x 0.75)	BGA68 (5.3 x 5.3 x 0.88)	BGA55 (3.5 x 3.5 x 0.60)	QFN40 (5 x 5 x 0.75)	QFN40 (5 x 5 x 0.75)
I/O Number	39	39	39	39	29	22	22

Note:

- GR5515IENDU and GR5513BENDU are embedded with wide-voltage Flash, with Flash power supply from 1.65 V to 3.6 V.
- GR5515I0NDA supports external wide-voltage Flash (range: 1.65 V–3.6V).

1.1 GR551x SDK

The GR551x SDK provides comprehensive software development support for GR551x SoCs. The GR551x SDK contains BLE Protocol Stack (BLE Stack) APIs, System APIs, peripheral drivers, a tool for generating and downloading .hex files, project example code, and related user documents.

The GR551x SDK version mentioned in this document is applicable to all GR551x SoCs.

1.2 BLE Stack

The architecture of BLE Stack is shown in Figure 1-1.

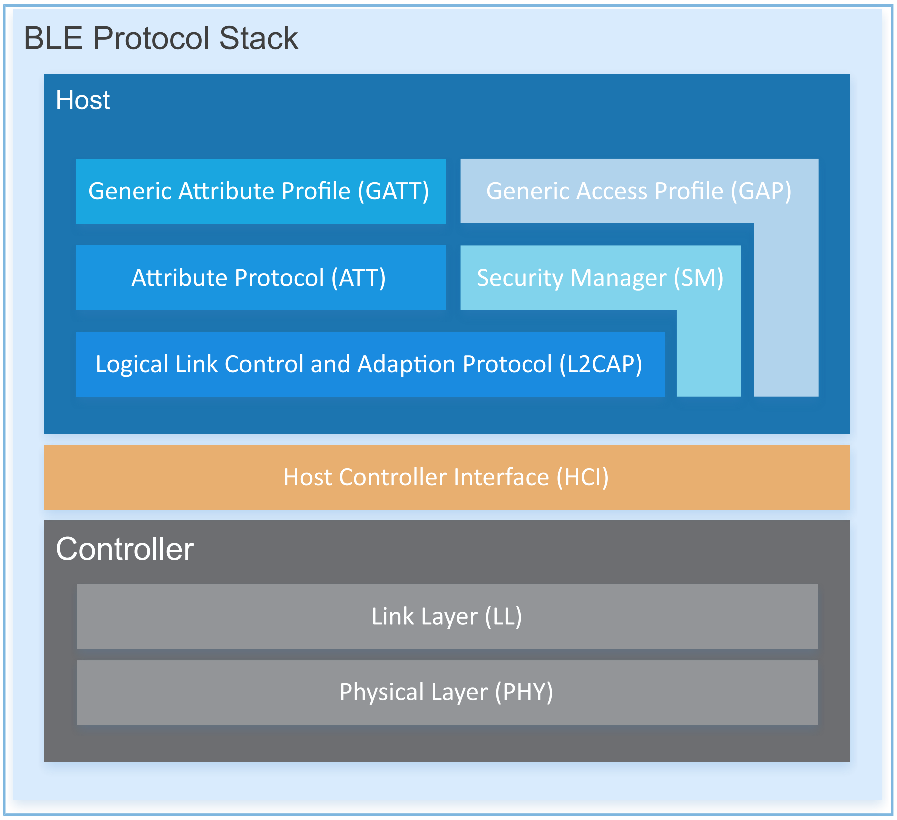

BLE Protocol Stack
Host
Generic Attribute Profile (GATT)
Generic Access Profile (GAP)
Attribute Protocol (ATT)
Security Manager (SM)
Logical Link Control and Adaption Protocol (L2CAP)
Host Controller Interface (HCI)
Controller
Link Layer (LL)
Physical Layer (PHY)

Figure 1-1 BLE Stack architecture

The BLE Stack consists of the Controller, the Host-Controller Interface (HCI), and the Host.

Controller

- Physical Layer (PHY) supports 1-Mbps and 2-Mbps adaptive frequency hopping and Gaussian Frequency Shift Keying (GFSK).
- Link Layer (LL) controls the RF state of devices. Devices are in one of the following five modes, and can switch between the modes on demand: Standby, Advertising, Scanning, Initiating, and Connection.

HCI

- HCI enables communications between Host and Controller, supported by software interfaces or standard hardware interfaces, for example, UART, Secure Digital (SD), or USB. HCI commands and events are transferred between Host and Controller through HCI.

Host

- Logical Link Control and Adaption Protocol (L2CAP) provides channel multiplexing and data segmentation and reassembly services for upper layers. It also supports logic end-to-end data communications.
- Security Manager (SM) defines pairing and key distribution methods, providing upper-layer protocol stacks and applications with end-to-end secure connection and data exchange functions.

- Generic Access Profile (GAP) provides upper-layer applications and profiles with interfaces to communicate and interact with protocol stacks, which fulfills functions such as advertising, scanning, connection initiation, service discovery, connection parameter update, secure process initiation, and response.
- Attribute Protocol (ATT) defines service data interaction protocols between a server and a client.
- Generic Attribute Profile (GATT) is based on the top of ATT. It defines a series of communications procedures for upper-layer applications, profiles, and services to exchange service data between GATT Client and GATT Server.

For more information about Bluetooth LE technologies and protocols, visit the Bluetooth SIG official website: www.bluetooth.com.

Specifications of GAP, SM, L2CAP, and GATT are provided in Bluetooth Core Spec. Specifications of other profiles/ services at the Bluetooth LE application layer are available on the GATT Specs page. Assigned numbers, IDs, and code which may be used by Bluetooth LE applications are listed on the Assigned Numbers page.

2 GR551x Bluetooth LE Software Platform

The GR551x Software Development Kit (SDK) is designed for GR551x SoCs, to help users develop Bluetooth LE applications. It integrates Bluetooth 5.1 APIs, System APIs, and peripheral driver APIs, with various example projects and instruction documents for Bluetooth and peripheral applications. Application developers are able to quickly develop and iterate products based on example projects in the GR551x SDK.

2.1 Hardware Architecture

The GR551x hardware architecture is shown as follows. This section introduces the modules in a GR551x SoC. For more information, see *GR551x Datasheet*.

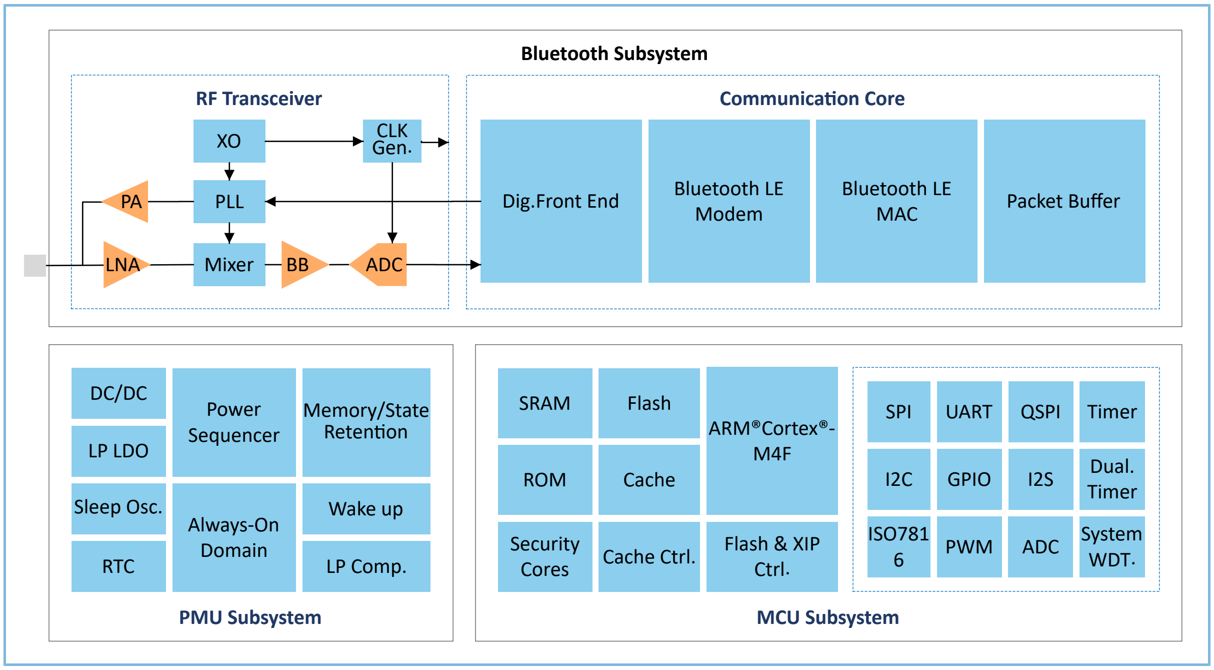


Figure 2-1 GR551x hardware architecture

- ARM® Cortex®-M4F: GR551x CPU. BLE Stack and application code run on the CPU.
- RAM: random access memory that provides memory space for program execution
- ROM: read-only memory, solidifying Bootloader and the software part of BLE Stack
- Security Cores: the secure computing engine unit, mainly including TRNG, AES, SHA, and PKC modules, which allows to check encrypted user application firmware. The encrypted firmware is checked through the secure boot process in ROM (In *Bluetooth Core Spec*, the secure computing unit is an independent module in Communication Core, and is irrelevant to Security Cores).
- Peripherals: GPIO, DMA, I2C, SPI, UART, PWM, Timer, and other hardware
- RF Transceiver: 2.4 GHz RF signal transceiver
- Communication Core: PHY of Bluetooth 5.1 Protocol Stack Controller. It is also the interface between the software protocol stack and 2.4 GHz RF hardware.

- Power Management Unit (PMU): It supplies power for system modules, and sets reasonable parameters for modules, including DC/DC, IO-LDO, Dig-LDO, and RF Subsystem, based on configuration parameters and current running state.
- Flash: flash memory unit packaged on the SoC. It stores user code and data, and supports the Execute in Place (XIP) Mode for user code.

2.2 Software Architecture

The software architecture of the GR551x SDK is shown in Figure 2-2.

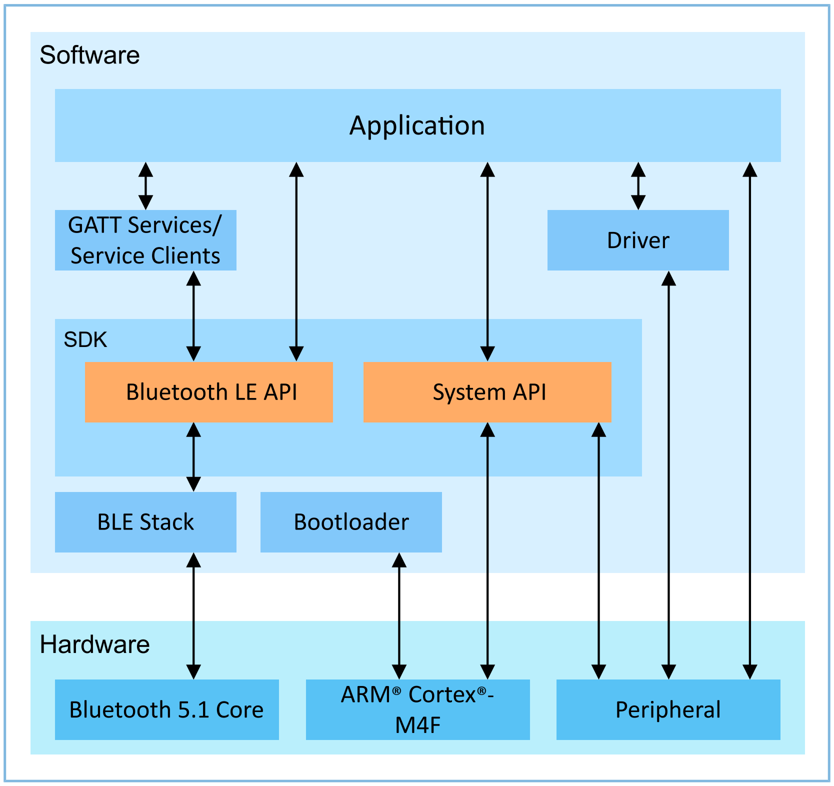


Figure 2-2 GR551x software architecture

- Bootloader

  It is a boot program used for GR551x software and hardware environment initialization, and to check and start applications.
- BLE Stack

  It is the implementation core of BLE protocol stacks. It consists of Controller, HCI, and Host protocols (including ATT, L2CAP, GAP, SM, and GATT), and supports roles of Broadcaster, Observer, Peripheral, and Central.
- Bluetooth LE SDK

  It refers to software development kit that provides easy-to-use SDK Bluetooth LE APIs and SDK System APIs.
  - SDK Bluetooth LE APIs: include L2CAP, GAP, SM, and GATT APIs.

  - SDK System APIs: provide API definitions for Non-volatile Data Storage (NVDS), Device Firmware Update (DFU), system power management, and generic system-level access interfaces.

- Application

  The SDK provides abundant Bluetooth and peripheral example projects. Each project contains compiled binary files; users can download these files to GR551x SoCs for operation and test. GRToolbox (Android) in the SDK also provides corresponding functions as most Bluetooth applications do, to help users with tests.

- Drivers

  API definitions and descriptions on peripheral drivers. For more information about drivers, see *GR551x HAL and LL Drivers User Manual*.

2.3 Memory Mapping

The memory mapping of a GR551x SoC is shown in Figure 2-3.

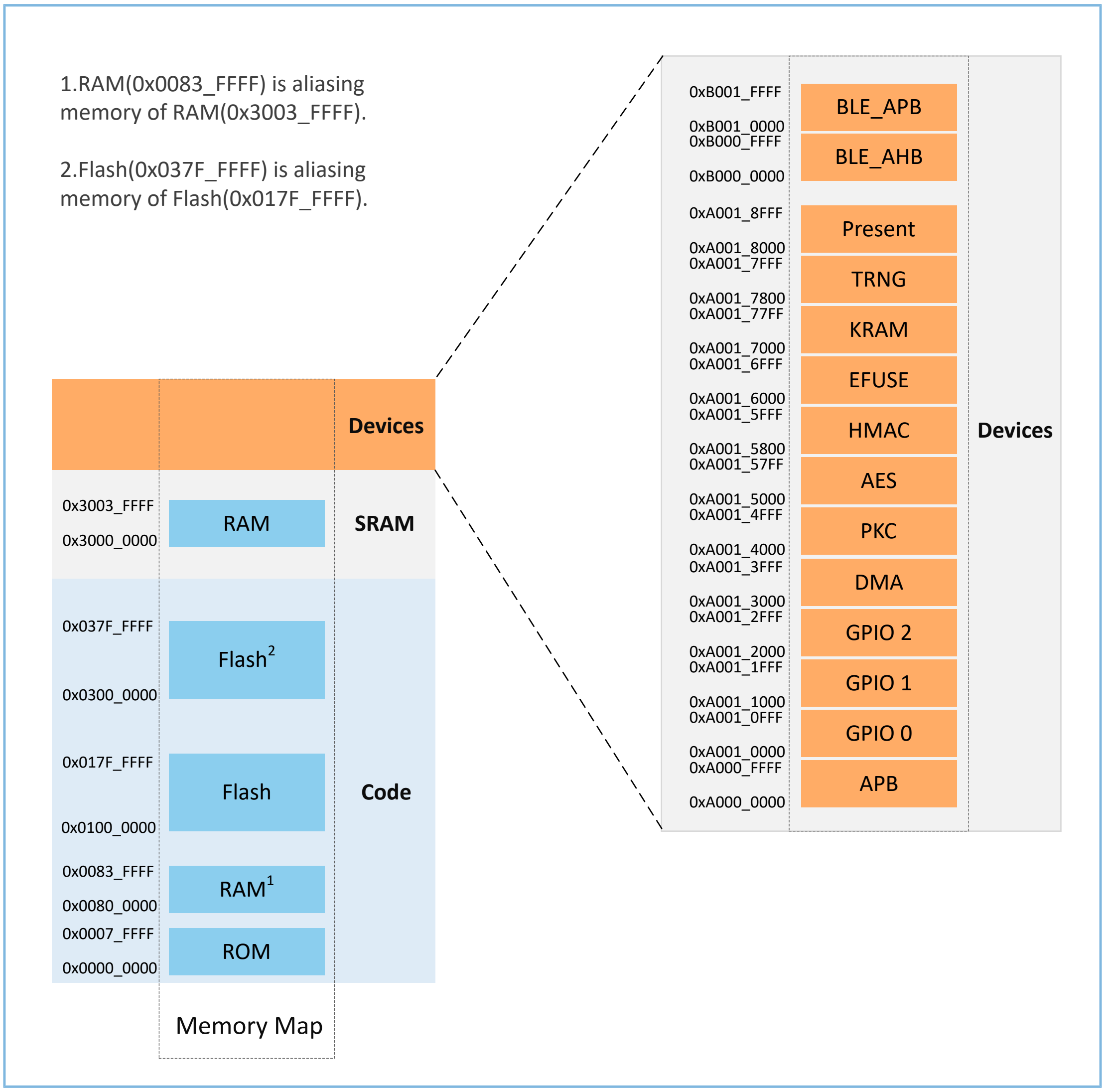


Figure 2-3 GR551x memory mapping

Note:

- GR5515 SoCs:

  RAM: 0x3000_0000 to 0x3003_FFFF or 0x0080_0000 to 0x0083_FFFF, 256 KB in total

  Flash: 0x0100_0000 to 0x010F_FFFF or 0x0300_0000 to 0x030F_FFFF, 1 MB in total

- GR5513 SoCs:

  RAM: 0x3000_0000 to 0x3001_FFFF or 0x0080_0000 to 0x0081_FFFF, 128 KB in total

  Flash: 0x0100_0000 to 0x0107_FFFF or 0x0300_0000 to 0x0307_FFFF, 512 KB in total

2.4 Flash Memory Mapping

GR551x packages an on-chip erasable flash memory, which supports XQSPI bus interface. This flash memory physically consists of several 4 KB flash sectors; it can be logically divided into storage areas for different purposes based on application scenarios.

The flash memory layout of a typical GR5515 application scenario is shown in Figure 2-4.

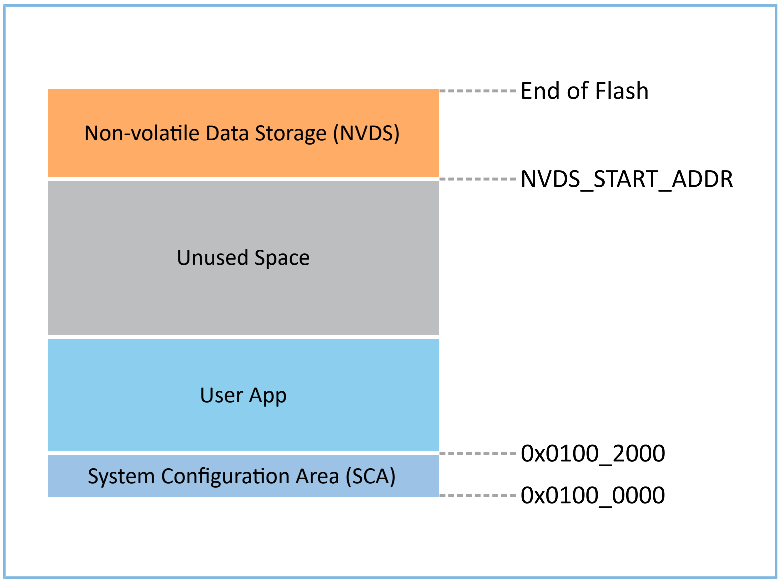


Figure 2-4 Flash memory layout

- System Configuration Area (SCA): an area to store system boot parameter configurations
- User App: storage area for application firmware
- Unused Space: a free area for developers. For example, developers can store new application firmware in the Unused Space temporarily during DFU.
- NVDS: Non-volatile Data Storage area

Note:

By default, NVDS occupies the last sector of flash memory. You can reasonably configure the start address of NVDS and the number of occupied sectors according to flash memory layout of products. For more information about the configuration, see "Section 4.3.2.1 Configuring custom_config.h".

Important: The start address of NVDS shall be aligned with that of the flash sectors.

2.4.1 SCA

SCA is in the first two sectors (8 KB in total; 0x0100_0000 to 0x0100_2000) of flash memory. It stores flags and other system configuration parameters used during system boot. The SDK toolchain generates an SCA image file based on the user configuration file *custom_config.h* (path: `Project_Folder\Src\config`), and programs the image info to SCA. See "Section 4.4 Generating Firmware". Figure 2-5 shows the SCA layout.

Note:

`Project_Folder` is the root directory of the project.

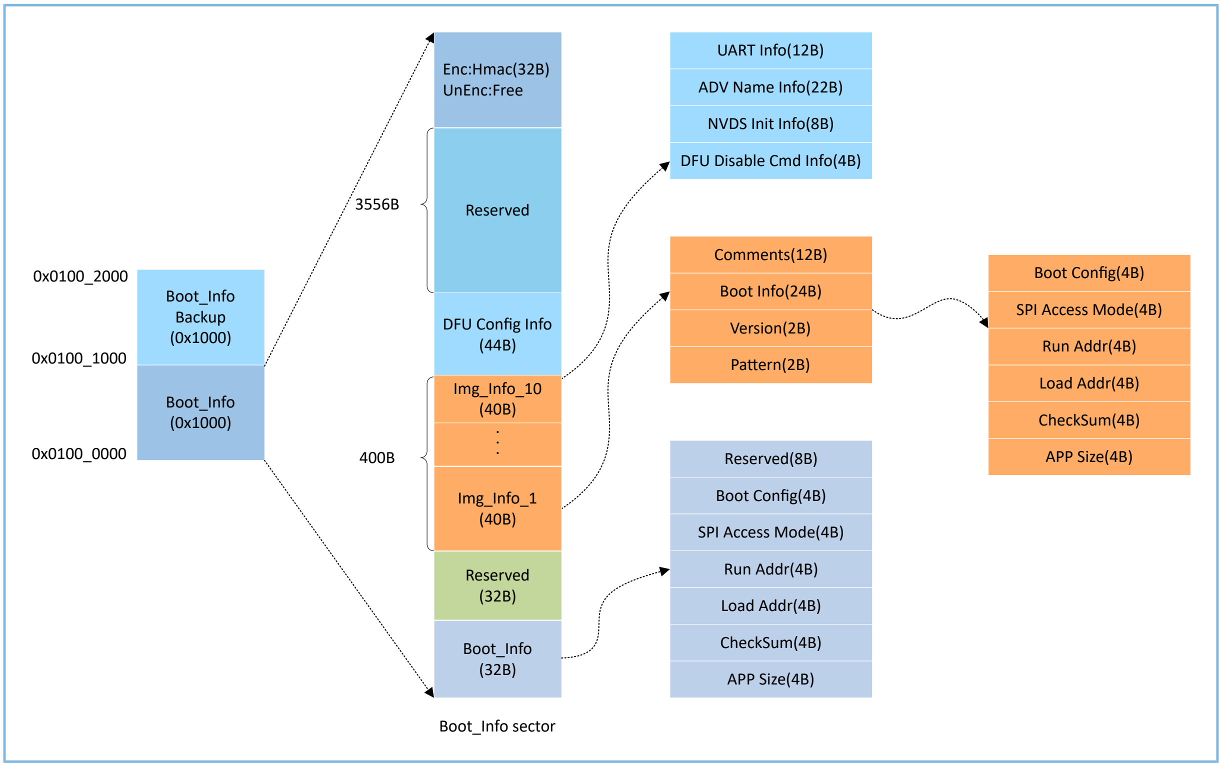


Figure 2-5 SCA layout

- The Boot_Info and the Boot_Info Backup store the same information, and the latter is the backup of the Boot_Info.
  - In non-security mode, the Bootloader obtains boot information from Boot_Info by default.
  - In security mode, the Bootloader checks Boot_Info first; if the check fails, the Bootloader checks Boot_Info Backup and obtains boot information from it.

- The firmware boot information is stored in the Boot_Info (32 B) area. The Bootloader checks and jumps to the entry address of the firmware based on the boot information.
    - The Boot Config area stores the system boot configuration information.
    - The SPI Access Mode area stores the SPI access mode configuration which is the fixed configuration of the system and cannot be modified.
    - The Run Addr area stores the firmware run address, corresponding to APP_CODE_RUN_ADDR in the *custom_config.h*.
    - The Load Addr area stores the firmware storage address, corresponding to APP_CODE_LOAD_ADDR in the *custom_config.h*.
    - The CheckSum area stores the firmware checksum which is computed automatically by the SDK toolchain after firmware is generated.
    - The APP Size area stores the firmware size which is computed automatically by the SDK toolchain after firmware is generated.
- Up to 10 pieces of firmware information (image info) can be stored in Img_Info areas. Firmware information is stored in Img_Info areas when you use GProgrammer to download firmware or update firmware in DFU Mode.
    - The Comments area stores the descriptive information about firmware and supports up to 12 characters. The SDK toolchain uses the firmware name as Comments after firmware is generated.
    - The Boot_Info (24 B) area stores the firmware boot information which is the same as the low 24-byte information in the Boot_Info (32 B) area mentioned above.
    - The Version area stores the firmware version, corresponding to VERSION in the *custom_config.h*.
    - The Pattern area stores a fixed value: 0x4744.
- The DFU Config Info area stores configurations of DFU module in ROM. You can call corresponding APIs to change the data stored in this area to configure DFU module.
    - The UART Info area stores UART configurations of DFU module, including status bit, baud rate, and GPIO configurations.
    - The ADV Name Info area stores advertising configurations of DFU module, including status bit, advertising name, and advertising length.
    - The NVDS Init Info area stores initialization configurations of NVDS system in DFU module, including status bit, NVDS area size, and start address.
    - The DFU Disable Cmd Info area stores DFU disable command configurations of DFU module, including status bit and Disable DFU Cmd (2 B, set as Bitmask). You can set the Disable DFU Cmd value to disable a DFU command.
- The HMAC check value is stored in the HMAC area. This area is valid only in security mode. For more information about the security mode, see *GR551x Firmware Encryption Application Note*.

2.4.2 NVDS

NVDS is a lightweight logical data storage system based on flash Hardware Abstract Layer (flash HAL). NVDS is located in the flash memory and data in it will not get lost in power-off status. By default, NVDS occupies the last sector of the flash memory for storage.

NVDS is an ideal choice to store small data blocks, for example, application configuration parameters, calibration data, states, and user information. BLE Stack will also store parameters (such as device binding information) in NVDS.

NVDS has the following characteristics:

- Each storage item (TAG) has a unique TAG ID for identification. User applications can read and change data according to TAG IDs, regardless of physical storage addresses.
- It is optimized based on medium characteristics of flash memory and supports data check, word alignment, defragmentation, and erase balance.
- The size and the start address of NVDS are configurable. The flash memory is divided into sectors (4 KB/sector). NVDS can be configured as several sectors, and the configured start address shall be 4 KB-aligned.

NVDS provides the following five simple APIs to manipulate non-volatile data in flash.

Table 2-1 NVDS APIs

Function Prototype	**Description**
uint8_t nvds_init(uint32_t start_addr, uint8_t sectors)	Initialize the flash sectors used by NVDS.
uint8_t nvds_get(NvdsTag_t tag, uint16_t *p_len, uint8_t *p_buf)	Read data according to TAG IDs from NVDS.
uint8_t nvds_put(NvdsTag_t tag, uint16_t len, const uint8_t *p_buf)	Write data to NVDS and mark the data with TAG IDs. If no TAG exists, create one.
uint8_t nvds_del(NvdsTag_t tag)	Remove the corresponding data of a TAG ID in NVDS.
uint16_t nvds_tag_length(NvdsTag_t tag)	Obtain the data length of a specified TAG.

For more information about NVDS APIs, see *GR551x API Reference* or the NVDS header file (available in `SDK_Folder\components\sdk\gr55xx_nvds.h`).

Figure 2-6 shows the format of the data stored in NVDS:

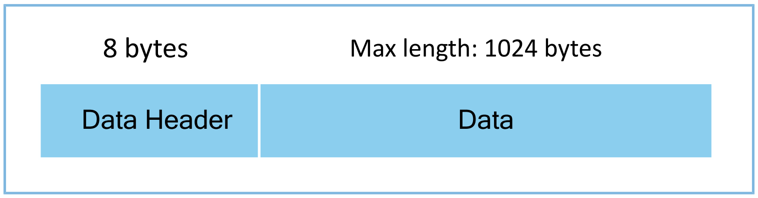


Figure 2-6 NVDS data format

Table 2-2 shows the data header format:

Table 2-2 Data header format

Byte	Name	Description
0–1	tag	Data tag
2–3	len	Data length
4–4	checksum	Checksum of data header
5–5	value_cs	Data checksum
6–7	reserved	Reserved field

Note:

BLE Stack also stores some parameters in NVDS. Therefore, it is required to allocate a flash storage area to NVDS. By default, the GR551x SDK uses the last sector of flash memory for NVDS. You can modify macros NVDS_START_ADDR and NVDS_NUM_SECTOR in *custom_config.h* to configure the start address and the size of NVDS. BLE Stack and applications share the same NVDS storage area. However, TAG ID namespace is divided into different categories. You can only use the TAG ID name category assigned to applications.

- Applications have to use NV_TAG_APP(idx) to obtain the TAG ID of application data. The TAG ID is used as an NVDS API parameter.
- Applications cannot use idx as the NVDS API parameter directly. The idx value ranges from 0x0000 to 0x3FFF.

Before running an application for the first time, you can use GProgrammer to write the initial TAG value used by BLE Stack and the application to NVDS. If you specify an NVDS area start address, instead of using the default NVDS area in the GR551x SDK, make sure the start address configured in GProgrammer is the same as that defined in the *custom_config.h*. For more information about configuration of the NVDS area start address in the *custom_config.h*, see "Section 4.3.2.1 Configuring custom_config.h".

2.5 RAM Mapping

The RAM of a GR5515 SoC is 256 KB in size with the start address of 0x3000_0000. It consists of 11 RAM Blocks. For the first 4 RAM Blocks, each is 8 KB; for the others, each is 32 KB. Each RAM Block can be powered on/off by software independently.

Note:

The GR5515 SoC provides an aliasing memory with the start address of 0x0080_0000 for RAM with the start address of 0x3000_0000, as shown in Figure 2-3. If the run address of code is within the range of the aliasing memory address, code can run faster in RAM. By default, the aliasing memory is enabled in the GR551x SDK.

The 256 KB RAM layout is shown in Figure 2-7:

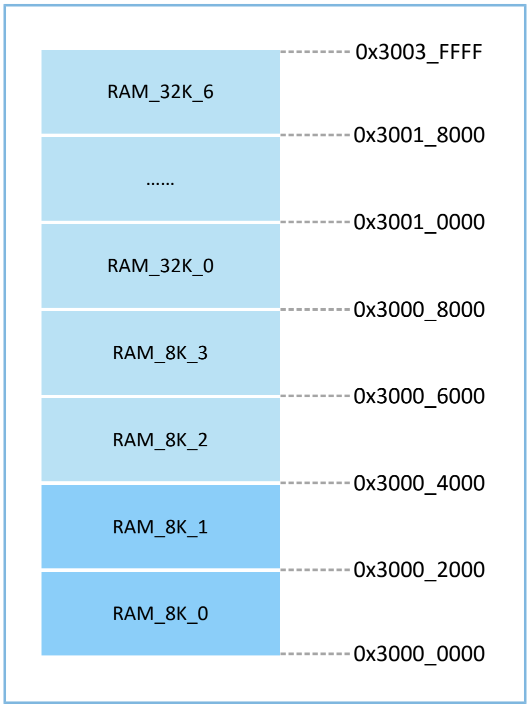


Figure 2-7 256 KB RAM layout

Running modes for applications include XIP and Mirror modes. For more information about configurations, see **APP_CODE_RUN_ADDR** in "Section 4.3.2.1 Configuring custom_config.h". RAM layouts of the two modes are different.

Table 2-3 Running modes for applications

Running Mode	Description
XIP Mode	It refers to Execute in Place Mode. User applications are stored in on-chip flash, and applications use the same space for running and loading. When the system is powered on, it fetches and executes commands from flash directly through the Cache Controller.
Mirror Mode	In Mirror Mode, user applications are stored in on-chip flash, and the running space of applications is RAM. During application boot, applications are loaded into RAM from external flash after check is completed, and the system jumps to RAM for operation.

Note:

Continuous access to flash is required in XIP Mode. Therefore, power consumption in this mode is a little higher than that in Mirror Mode.

2.5.1 RAM Layout in XIP Mode

The typical RAM layout in XIP Mode is shown in Figure 2-8. You are able to modify the layout based on product needs.

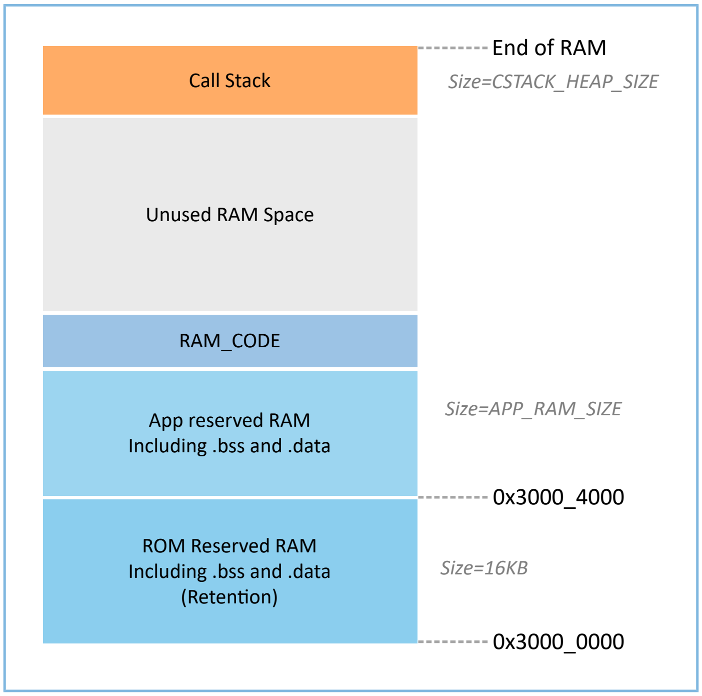


Figure 2-8 RAM layout in XIP Mode

The layout in XIP Mode allows application code to be run directly in the code loading area, so that more RAM space is available for applications. It is required to turn off the XIP Mode during flash data update. As a result, the CPU is unable to fetch commands from flash. Therefore, the run address of flash driver code shall be directed to RAM during compiling and linking. The scatter file used by the GR551x SDK example projects defines a RAM_CODE area to run code whose run address is in RAM. In Sleep Mode, the RAM Block occupied by the RAM_CODE area shall be in RETENTION Mode.

Note:

You cannot remove the RAM_CODE segment from the scatter file. For more information about the scatter file, see "Section 4.3.2.2 Configuring Memory Layout".

2.5.2 RAM Layout in Mirror Mode

The typical RAM layout in Mirror Mode is shown in Figure 2-9. You are able to modify the layout based on product needs.

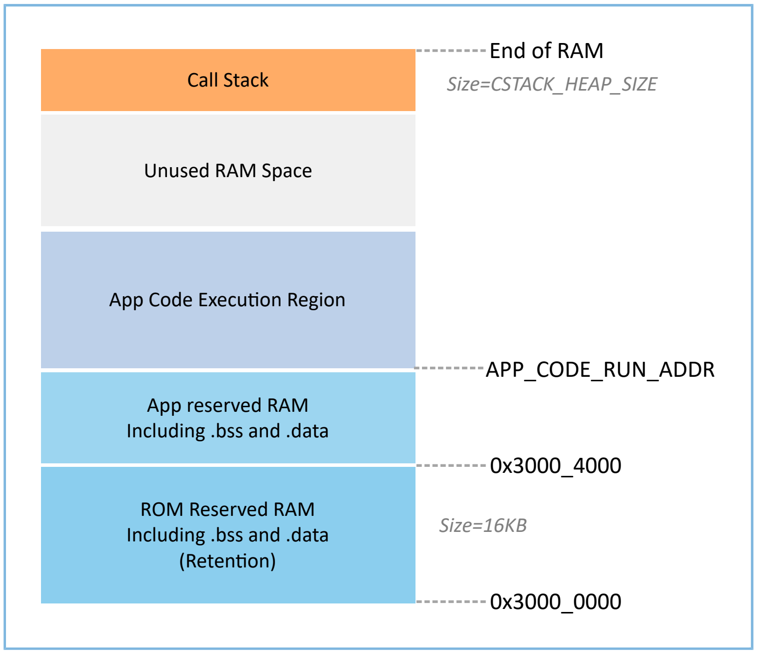


Figure 2-9 RAM layout in Mirror Mode

The layout in Mirror Mode allows application code to be run in RAM. When the SoC is powered on, it goes into the cold boot process. The Bootloader copies application code from flash to the RAM segment **App Code Execution Region**. When the SoC is awoken from Sleep Mode, it goes into the warm boot process. To shorten the warm boot time, the Bootloader does not copy the application code again to the RAM segment **App Code Execution Region**.

The start address of the RAM segment **App Code Execution Region** is determined by the macro APP_CODE_RUN_ADDR in *custom_config.h*. Developers shall determine the value of APP_CODE_RUN_ADDR based on the use of .data and .bss segments in the application, to prevent address overlapping with the .bss segment at a low address or the call stack segment at a high address. The distribution of RAM segments can be obtained from the .map file.

It is recommended that developers use RAM Aliasing Memory address (0x0080_0000–0x0083_FFFF) to set APP_CODE_RUN_ADDR. In the case of RAM segment overlapping, an error will occur and the overlapping position will be prompted during project building, to help developers check and quickly locate the RAM segment overlapping.

2.5.3 RAM Power Management

Each RAM Block has three power modes: POWER OFF, RETENTION, and FULL.

- The FULL Mode corresponds to the Active Mode of the system; MCU is permitted to read from and write to RAM Blocks.
- RETENTION Mode is mainly used in Sleep Mode of the system. Data in RAM Blocks in this power mode does not get lost and is ready for use by the system when it switches from Sleep Mode to Active Mode.
- RAM Blocks in POWER OFF Mode are powered down, and data stored in these blocks gets lost. Users shall store the data in advance.

By default, the PMU in the GR551x enables all RAM power sources when the system starts. The GR551x SDK also provides a complete set of RAM power management APIs. You can configure the power of RAM Blocks based on application needs.

By default, the automatic RAM power management mode will be enabled when the system starts. The system will control the power of RAM Blocks automatically according to RAM usage by applications. The configuration rules are provided as follows:

- When the system is in Active Mode, unused RAM Blocks are set to POWER OFF Mode, and RAM Blocks to be used are set to FULL Mode.
- When the system enters Sleep Mode, unused RAM Blocks remain in POWER OFF Mode, and RAM Blocks to be used are set to RETENTION Mode.

Configurations in practice are described below:

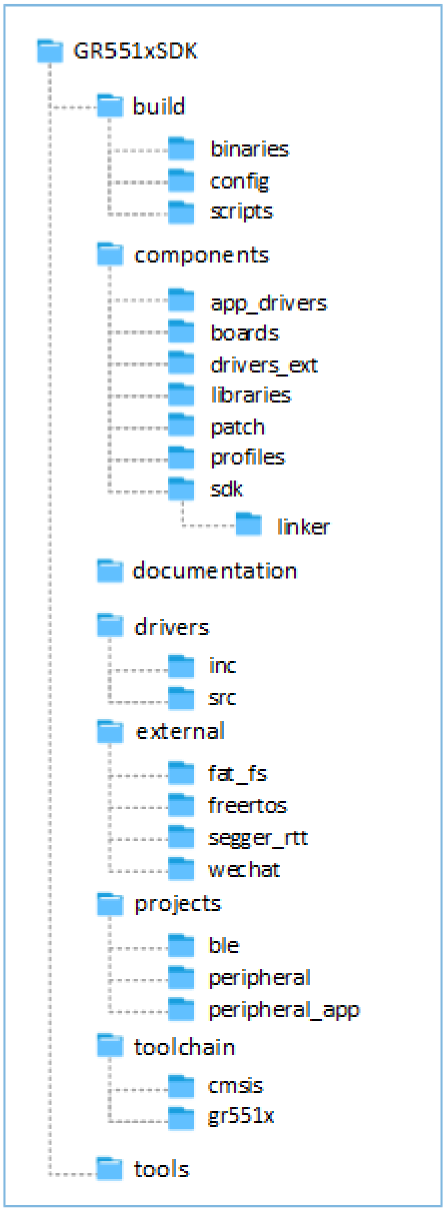
- In Bluetooth LE applications, two RAM Blocks, RAM_8K_0 and RAM_8K_1, are reserved for Bootloader and BLE Stack only, not available for applications. When the system is in Active Mode, these two RAM Blocks shall be in FULL Mode; when the system is in Sleep Mode, they shall be in RETENTION Mode. Non-Bluetooth LE applications can use these two RAM Blocks.
- Purposes of RAM_8K_2 and subsequent RAM Blocks are defined by applications. Generally, user data and the code segments to be executed in RAM are defined in continuous segments starting from RAM_8K_2; top of function call stacks are defined in upper address part of RAM. The power mode of these RAM Blocks can be enabled, or controlled by applications.

Note:

1. Only if a RAM Block is in FULL Mode, an MCU is permitted to access it.
2. To manage the RAM power sources and use a RAM Block which is not included in the memory layout information of applications, you need to call the mem_pwr_mgmt_mode_set_from(uint32_t start_addr, uint32_t size) function during application initialization, to power the RAM Block on.
3. Details about RAM power management APIs are in `SDK_Folder\components\sdk\platform_sdk.h`. SDK_Folder is the root directory of GR551x SDK.

2.6 GR551x SDK Directory Structure

The folder directory structure of the GR551x SDK is shown in Figure 2-10.



```
GR551xSDK
    build
        binaries
        config
        scripts
    components
        app_drivers
        boards
        drivers_ext
        libraries
        patch
        profiles
        sdk
            linker
    documentation
    drivers
        inc
        src
    external
        fat_fs
        freertos
        segger_rtt
        wechat
    projects
        ble
        peripheral
        peripheral_app
    toolchain
        cmsis
        gr551x
    tools
```

Figure 2-10 GR551x SDK directory structure

Detailed descriptions of folders in the GR551x SDK are shown in Table 2-4.

Table 2-4 GR551x SDK folders

Folder	**Description**
build\binaries	It contains tools used for the build process.
build\config	It is the project configuration directory that stores the *custom_config.h* template file. Contents in this file are used to configure projects, and to provide related input parameters for the SDK toolchain.
build\scripts	It contains batch files and script files for building, downloading, and debugging.
components\app_drivers	It contains driver API source code, which are easy to use for application developers.

Folder	Description
components\boards	It contains header files corresponding to the development board.
components\drivers_ext	It contains drivers of third-party components on the development board.
components\libraries	It contains libraries provided in the GR551x SDK.
components\patch	It contains ROM Patch files which provide incremental updates for BLE Stack in ROM.
components\profiles	It contains source files of GATT Services/Service Clients implementation examples provided in the GR551x SDK.
components\sdk	It contains API header files provided in the GR551x SDK.
components\sdk\linker	It contains symbol table files and library files provided in the GR551x SDK for the linker.
documentation	It contains the GR551x API references.
drivers\inc	It contains HAL and LL header files of the GR551x peripheral drivers.
drivers\src	It contains HAL source code of the GR551x peripheral drivers.
external\fat_fs	It contains source code of FatFs, which is a third-party program.
external\freertos	It contains source code of FreeRTOS, which is a third-party program.
external\segger_rtt	It contains source code of SEGGER RTT, which is a third-party program.
external\wechat	It contains source code of WeChat, which is a third-party program.
projects\ble	It contains Bluetooth LE application project examples, such as Heart Rate Sensor and Proximity Reporter.
projects\peripheral	It contains peripheral project examples of a GR551x SoC.
projects\peripheral_app	It contains project examples of peripheral applications of a GR551x SoC.
toolchain\cmsis	It contains toolchain files of CMSIS.
toolchain\gr551x	It contains toolchain documents of the compilation platform.
tools	It contains development and debugging software of the GR551x.

3 Bootloader

The GR551x supports two code running modes: XIP and Mirror. When the system is powered on, the Bootloader first reads the system boot configuration information from SCA, then performs application firmware integrity check and system initialization configuration accordingly, and finally jumps to the code running space to run code. The boot procedures may vary in different running modes.

- In XIP Mode, the Bootloader first initializes Cache and XIP controllers after finishing application firmware check, and then jumps to the code run address in flash to run code.
- In Mirror Mode, after finishing application firmware check, the Bootloader loads the code in flash to corresponding RAM running space based on system configurations, resets flash interfaces, and jumps to RAM to run code.

The application boot procedures of the GR551x SDK are shown in Figure 3-1.

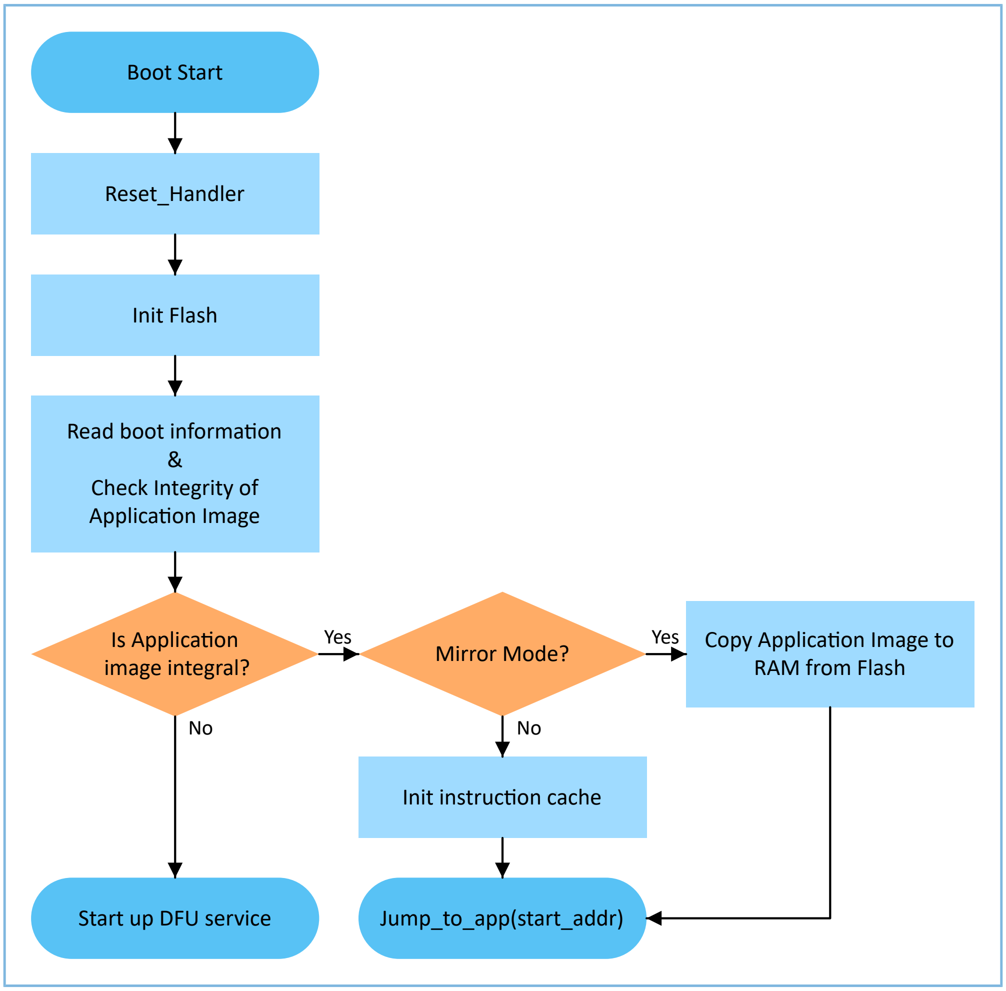


Figure 3-1 Application boot procedures of the GR551x SDK

1. When the device is powered on, CPU jumps to 0x0000_0000 and executes the reset_handler in ROM to enter the Bootloader.
2. Bootloader initializes flash.

3. Bootloader reads boot information from SCA in flash and checks application firmware integrity.

Note:

The GR551x enhances security by encrypting and signing application firmware.

- Security mode: If the security mode is enabled, the Bootloader reads boot information from SCA and performs HMAC check; after successful checking, the Bootloader decrypts SCA boot information and then implements the verification signature process in the security boot process, to guarantee firmware integrity and prevent tampering or disguise; if the signature verification is successful, the automatic decryption function is enabled. For more information, see *GR551x Firmware Encryption Application Note*.
- Non-security mode: If the security mode is not enabled, the Bootloader uses SCA boot information to check CRC integrity check for application firmware.

---

4. If the integrity check fails, the Bootloader starts the BLE DFU Service. You can update application firmware in flash through this service and the App on the mobile phone.
5. If the integrity check passes, the Bootloader determines a running mode.
    - In XIP Mode, the Bootloader jumps to the application firmware in flash to start implementation after XIP configuration is completed.
    - In Mirror Mode, the Bootloader copies the application firmware in flash to a specified segment in RAM, and then runs the application firmware in RAM.

4 Development and Debugging with GR551x SDK in Keil

This chapter introduces how to build, compile, download, and debug Bluetooth LE applications with the GR551x SDK in Keil.

4.1 Installing Keil MDK

Keil MDK-ARM IDE (Keil) is an Integrated Development Environment (IDE) provided by ARM® for Cortex® and ARM devices. You can download and install the Keil installation package from the Keil official website https://www.keil.com/demo/eval/arm.htm. For the GR551x SDK, Keil V5.20 or a later version shall be installed.

Note:

For more information about how to use Keil MDK-ARM IDE, see online manuals provided by ARM: https://www.keil.com/support/man_arm.htm.

The main interface of Keil is shown in Figure 4-1.

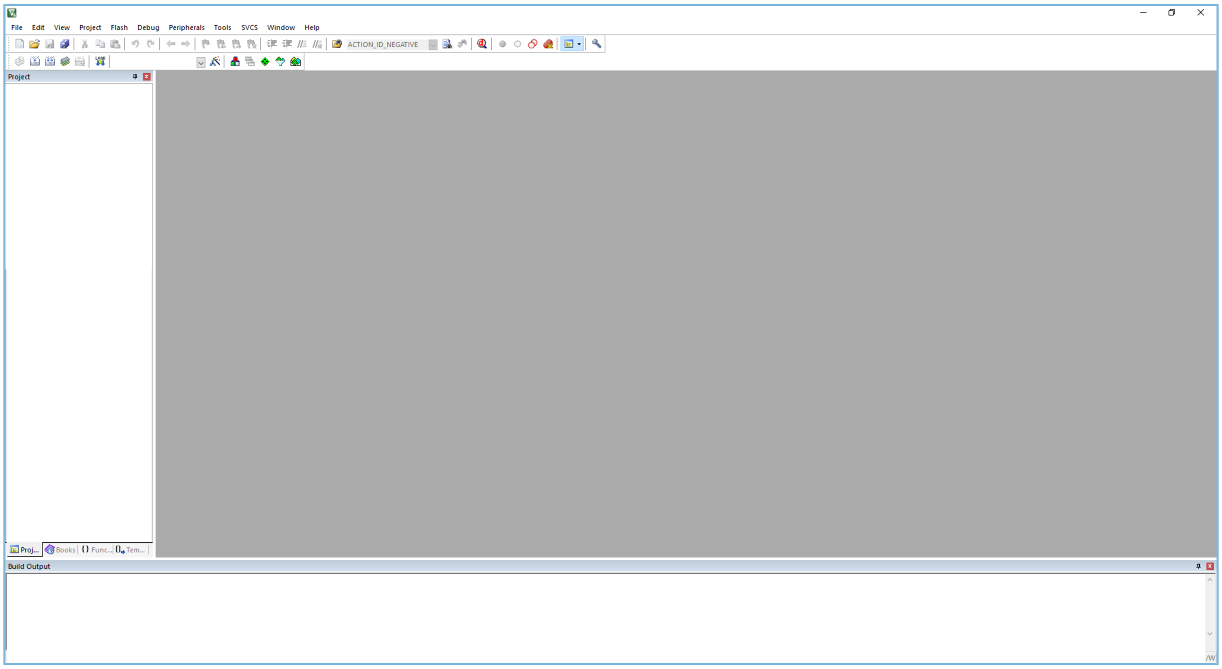


Figure 4-1 Keil interface

Frequently used function buttons of Keil are shown in Table 4-1.

Table 4-1 Frequently used function buttons of Keil

Keil Icon	Description
🪄	Options for target
🔍	Start/Stop Debug Session
LOAD	Download

Keil Icon	Description
	Build

4.2 Installing GR551x SDK

The GR551x SDK is ready for use after the GR551x SDK software package is extracted. No manual installation is required.

Note:

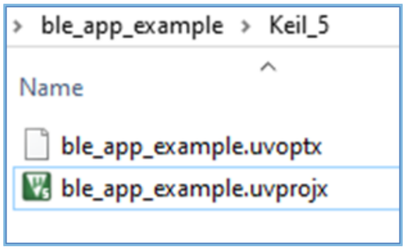
- SDK_Folder is the root directory of GR551x SDK.
- Keil_Folder is the root directory of Keil.

4.3 Building a Bluetooth LE Application

This section introduces how to build a Bluetooth LE application.

4.3.1 Preparing ble_app_example

Open `SDK_Folder\projects\ble\ble_peripheral\`, copy *ble_app_template* to the current directory, and rename it as *ble_app_example*. Rename the base name of .uvoptx and .uvprojx files in `ble_app_example\Keil_5` as *ble_app_example*.



› ble_app_example › Keil_5

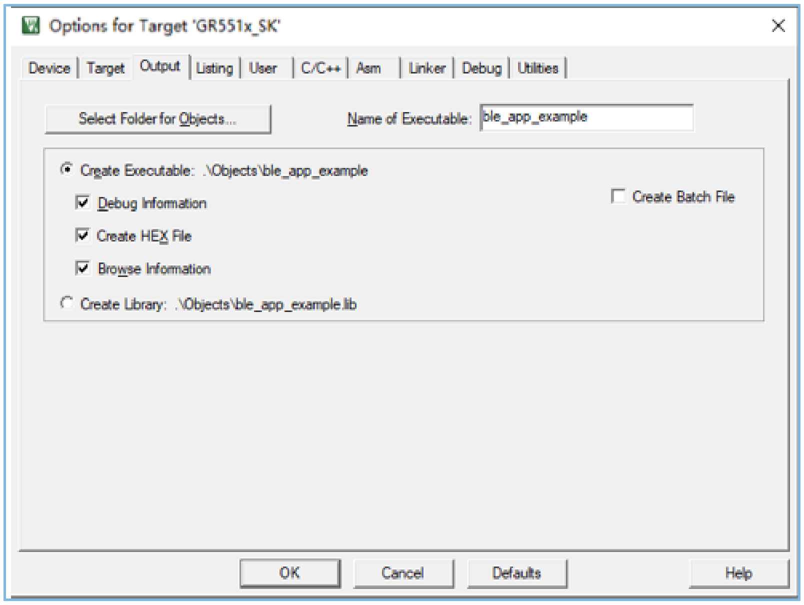
Name

ble_app_example.uvoptx

ble_app_example.uvprojx

Figure 4-2 ble_app_example folder

Double-click *ble_app_example.uvprojx* to open the project example in Keil. Click , and select **Output** in **Options for Target 'GR551x_SK'**; enter **ble_app_example** in **Name of Executable**.



Name of Executable: ble_app_example

● Create Executable: .\Objects\ble_app_example

☑ Debug Information

☑ Create HEX File

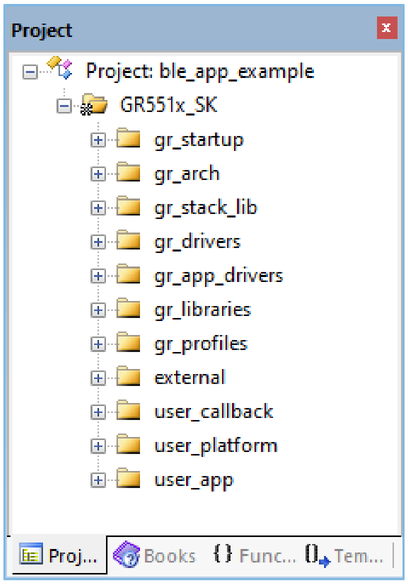
☑ Browse Information

☐ Create Batch File

○ Create Library: .\Objects\ble_app_example.lib

Figure 4-3 Modifications to **Name of Executable**

All groups of the ble_app_example project are available in the **Project Window** of Keil.



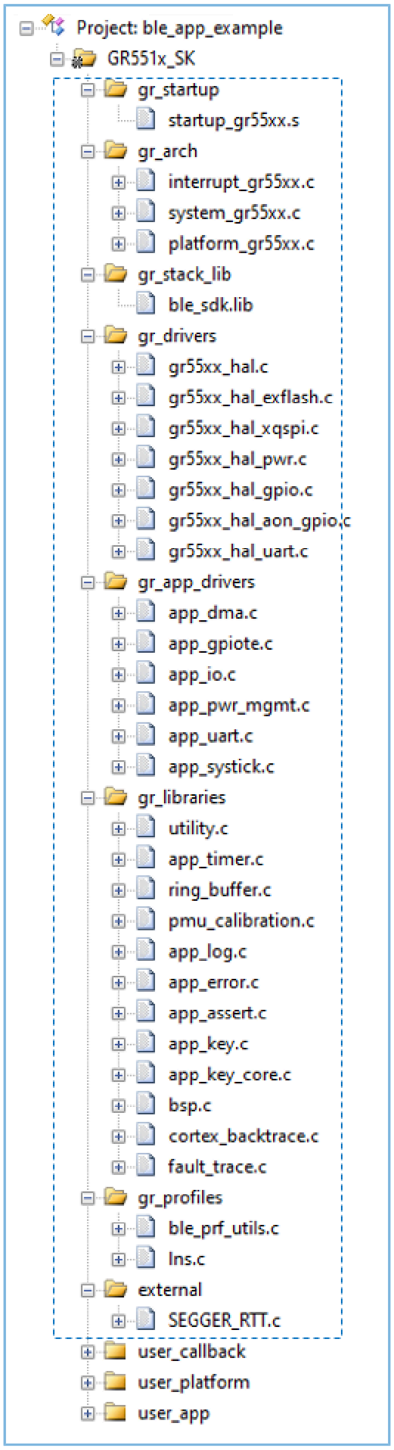
Project: ble_app_example
- GR551x_SK
  - gr_startup
  - gr_arch
  - gr_stack_lib
  - gr_drivers
  - gr_app_drivers
  - gr_libraries
  - gr_profiles
  - external
  - user_callback
  - user_platform
  - user_app

Figure 4-4 Project ble_app_example

Groups of the ble_app_example project are mainly in two categories: SDK groups and User groups.

- SDK groups

  The SDK groups include gr_startup, gr_arch, gr_stack_lib, gr_drivers, gr_app_drivers, gr_libraries, gr_profiles, and external.



- Project: ble_app_example
  - GR551x_SK
    - gr_startup
      - startup_gr55xx.s
    - gr_arch
      - interrupt_gr55xx.c
      - system_gr55xx.c
      - platform_gr55xx.c
    - gr_stack_lib
      - ble_sdk.lib
    - gr_drivers
      - gr55xx_hal.c
      - gr55xx_hal_exflash.c
      - gr55xx_hal_xqspi.c
      - gr55xx_hal_pwr.c
      - gr55xx_hal_gpio.c
      - gr55xx_hal_aon_gpio.c
      - gr55xx_hal_uart.c
    - gr_app_drivers
      - app_dma.c
      - app_gpiote.c
      - app_io.c
      - app_pwr_mgmt.c
      - app_uart.c
      - app_systick.c
    - gr_libraries
      - utility.c
      - app_timer.c
      - ring_buffer.c
      - pmu_calibration.c
      - app_log.c
      - app_error.c
      - app_assert.c
      - app_key.c
      - app_key_core.c
      - bsp.c
      - cortex_backtrace.c
      - fault_trace.c
    - gr_profiles
      - ble_prf_utils.c
      - lns.c
    - external
      - SEGGER_RTT.c
    - user_callback
    - user_platform
    - user_app

Figure 4-5 SDK groups

Source files in the SDK groups are not required to be modified. Group descriptions are provided below:

Table 4-2 SDK groups

SDK Group Name	Description
gr_startup	It contains the system boot file.

SDK Group Name	Description
gr_arch	It contains initialization configuration files and system interrupt implementation files for System Core and PMU.
gr_stack_lib	It contains the GR551x SDK .lib file.
gr_drivers	It contains drive source files of SoC peripherals. You can add hardware drives on demand.
gr_app_drivers	It contains driver API source files, which are easy to use for application developers. You can add related application drivers on demand.
gr_libraries	It contains open source files of common assistant software modules and peripheral drivers provided by in the SDK.
gr_profiles	It contains source files of GATT Services/Service Clients. You can add necessary GATT source files for projects.
external	It contains source files for third-party programs, such as FreeRTOS and SEGGER RTT. You can add third-party programs on demand.

- User groups

  User groups include user_callback, user_platform, and user_app.

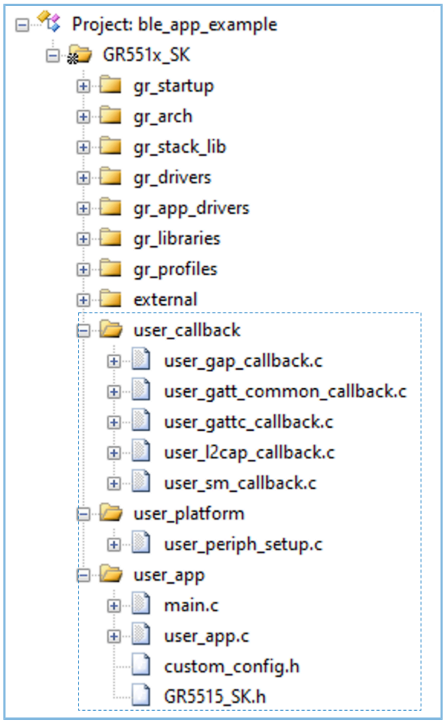


Figure 4-6 User groups

Functions for source files in User groups need to be implemented by developers. Group descriptions are provided below:

Table 4-3 User groups

User Group Name	Description
user_callback	It contains BLE Stack callback functions, which are implemented in applications. For more information about callbacks, see Table 4-5.
user_platform	It implements software and hardware resource setting and application initialization; you need to execute corresponding APIs on demand.
user_app	It contains main() function entries, and other source files created by developers, which are used to configure runtime parameters of BLE Stack and execute event handlers of GATT Services/Service Clients.

4.3.2 Configuring a Project

You should configure corresponding project options according to product characteristics, including NVDS, code running mode, memory layout, After Build and other configuration items.

4.3.2.1 Configuring *custom_config.h*

The *custom_config.h* file is used to configure parameters of application projects. A *custom_config.h* template is provided in `SDK_Folder\build\config\`. The *custom_config.h* of each application example project is in `Src\config` under project directory.

Table 4-4 Parameters in the *custom_config.h*

Macro	Description
CHIP_TYPE	Select the chip type. • 0: GR5515 • 1: GR5513
ENCRYPT_ENABLE	It is used to enable/disable firmware encryption. Default: 0 • 0: Disable firmware encryption; support removing related encryption code to save RAM space. • 1: Enable firmware encryption.
EXT_EXFLASH_ENABLE	Use external Flash or not. • 0: No • 1: Yes
PLATFORM_INIT_ENABLE	It is used to enable/disable platform initialization. If this macro is disabled, the SoC Bluetooth LE and sleep functions are disabled. Default value for the macro: 1 • 0: Disable • 1: Enable

Macro	Description
SYS_FAULT_TRACE_ENABLE	It is used to enable/disable Callstack Trace Info printing. If printing is enabled, the Callstack Trace Info is printed through serial ports when a HardFault occurs. • 0: Disable Callstack Trace Info printing. • 1: Enable Callstack Trace Info printing.
APP_DRIVER_USE_ENABLE	It is used to enable/disable the App Drivers module. • 0: Disable the App Drivers module. • 1: Enable the App Drivers module.
APP_LOG_ENABLE	It is used to enable/disable the APP LOG module. • 0: Disable the APP LOG module. • 1: Enable the APP LOG module.
APP_LOG_STORE_ENABLE	It is used to enable/disable the APP LOG STORE module. • 0: Disable the APP LOG STORE module. • 1: Enable the APP LOG STORE module.
APP_LOG_PORT	It is used to set the output port of the APP LOG module. • 0: UART • 1: J-Link RTT • 2: ARM ITM
SK_GUI_ENABLE	It is used to enable/disable the GUI module on GR5515 Starter Kit Board. • 0: Disable the GUI module. • 1: Enable the GUI module.
DEBUG_MONITOR	It is used to enable/disable the Debug Monitor module. • 0: Disable the Debug Monitor module. • 1: Enable the Debug Monitor module.
DTM_TEST_ENABLE	It is used to enable/disable DTM Test. • 0: Disable DTM Test. • 1: Enable DTM Test.
DFU_ENABLE	It is used to enable/disable DFU. • 0: Disable DFU. • 1: Enable DFU.
FLASH_PROTECT_PRIORITY	During flash write or erase, applications can block the interrupts with priority level lower than or equal to a set value. When FLASH_PROTECT_PRIORITY is set to N, interrupt requests with a priority level not higher than N are suspended. After erase is completed, flash responds to the suspended interrupt

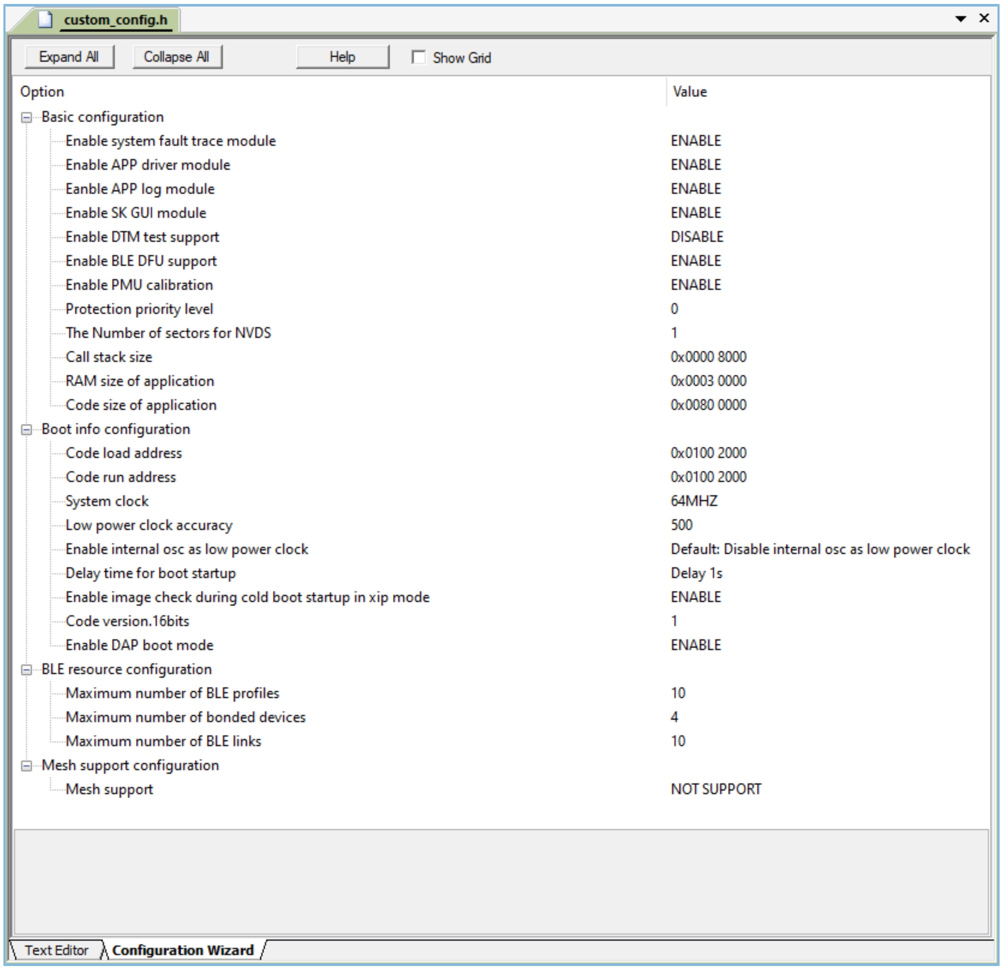
Macro	Description
	requests. By default, flash does not respond to any interrupt request during erase. Developers can set a value on demand.
NVDS_START_ADDR	Start address of NVDS. By default, this macro is defined in *cutom_config.h*. However, the defined initial value only applies to the scenario where the applied NVDS contains only one sector. If more than one sector is needed, users shall calculate the start address (4 KB-aligned required) according to the applied sector number.
NVDS_NUM_SECTOR	It represents the number of flash sectors for NVDS.
CSTACK_HEAP_SIZE	You can adjust the sizes of Call Stack and Heap for applications according to practical usage of applications. The value shall not be less than 6 KB. The default is 16 KB. After compilation of an example project, a Maximum Stack Usage is provided in `Keil_5\Objects\<project_name>.htm` for reference.
ENABLE_BACKTRACE_FEA	Enable/Disable stack backtrace functionality. • 0: Disable stack backtrace. • 1: Enable stack backtrace.
CHIP_VER*	It indicates the SoC version number.
APP_CODE_LOAD_ADDR*	It represents the start address of the application storage area. This address shall be within the flash address range.
APP_CODE_RUN_ADDR*	It represents the start address of the application running space. • If the value is the same as APP_CODE_LOAD_ADDR, applications run in XIP Mode. • If the value is within the RAM address range, applications run in Mirror Mode.
SYSTEM_CLOCK	It represents the system clock frequency. Optional values are provided as follows: • 0: 64 MHz • 1: 48 MHz • 2: 16 MHz (XO) • 3: 24 MHz • 4: 16 MHz • 5: 32MHz (PLL)
CFG_LF_ACCURARY_PPM	It represents the Bluetooth LE low frequency sleep clock accuracy. The value shall range from 1 to 500 (unit: ppm).
CFG_LPCLK_INTERNAL_EN	It is used to enable/disable the OSC clock inside an SoC as the Bluetooth LE low-frequency sleep clock. If the OSC clock is enabled, CFG_LF_ACCURARY_PPM will be set to 500 automatically. • 0: Disable. • 1: Enable.

Macro	Description
CFG_CRYSTAL_DELAY	It is used to set the delay time of chips after RTC is enabled in PMU or parameters like RTC GM are modified. It shall be configured based on the stabilization time after starting oscillating obtained in practical crystal oscillator tests. Value range: 100–500 (unit: ms); default: 100 ms.
BOOT_LONG_TIME*	It is used to set necessary 1-second delay (during SoC boot before implementing the second half Bootloader). • 0: No delay. • 1: Delay for 1 second.
BOOT_CHECK_IMAGE*	It determines whether to check the image during cold boot in XIP mode. • 0: Do not check. • 1: Check.
VERSION	It represents the version number of application firmware; length: 2 bytes; it is stored in hexadecimal format.
EXFLASH_WAKEUP_DELAY	During warm boot, set the delay time for waking up Flash and reading the chip ID. Value range: 0–10; unit: 5 μs. Setting the value to 0 indicates no delay. Each time the value increases by 1, the delay time increases by 5 μs.
CFG_MAX_BOND_DEVS	It represents the maximum number of bonded devices supported by applications. You should set the value on demand. A larger value means more RAM space to be occupied.
CFG_MAX_PRFS	It represents the maximum number of GATT Profiles/Services included in applications. Set the value on demand: A larger value means to occupy more RAM space.
CFG_MAX_CONNECTIONS	It represents the maximum number of connected devices supported by applications, and the number shall not be greater than 10. You can set the value based on needs. A larger value means more RAM space to be occupied by BLE Stack Heaps. The size of BLE Stack Heaps is defined by the following four macros in *flash_scatter_config.h*, which cannot be changed by developers: • ENV_HEAP_SIZE • ATT_DB_HEAP_SIZE • KE_MSG_HEAP_SIZE • NON_RET_HEAP_SIZE
CFG_MAX_ADVS	The maximum number of Bluetooth LE legacy advertising and extended advertising supported by applications
CFG_MAX_ADV_DATA_LEN_SUPPORT	Support legacy advertising with data length at 31 bytes or not. • 0: No • 1: Yes
CFG_MAX_PER_ADVS	The maximum number of Bluetooth LE periodic advertising supported by applications

Macro	Description
	Note: The sum of configured legacy/extended advertising value (CFG_MAX_LEG_EXT_ADVS) and the configured periodic advertising value (CFG_MAX_PER_ADVS) shall not exceed 5.
CFG_MAX_SCAN	The maximum Bluetooth LE scanning number supported by applications; max: 1
CFG_MAX_SYNCS	Number of synchronized periodic advertising; used for reserving RAM for BLE Protocol Stack. Developers can set the value according to the number of synchronized periodic advertising in use. Max: 5
CFG_MESH_SUPPORT	Support Mesh or not. • 0: No • 1: Yes
CFG_LCP_SUPPORT	Support the LCP module or not. • 0: No • 1: Yes
APP_DRV_I2C/SPI/UART_DMA_ENABLE	Enable/Disable DMA mode of the I2C/SPI/UART APP Driver module. • 0: Disable • 1: Enable
APP_DRV_I2C/SPI/QSPI/ UART_IT_ENABLE	Enable/Disable interrupt mode of the I2C/SPI/UART APP Driver module. • 0: Disable • 1: Enable
APP_INFO_COMMENTS*	Define firmware comments. Maximum length: 12 characters. • When a firmware name length is longer than 12 characters with this macro not defined, the firmware comments is set to the first 12 characters of the firmware name by default. • When a firmware name length is shorter than 12 characters with this macro not defined, the firmware comments is set to the name of the firmware by default.

*: Macros marked with an asterisk (*) in the table above are used to initialize the BUILD_IN_APP_INFO structure which is defined at 0x200 address of the firmware. During system boot, the Bootloader reads the configuration information of the firmware from the address as boot parameters.

Notes in the *custom_config.h* comply with *Configuration Wizard Annotations* of Keil. Therefore, you can use the graphic Keil Configuration Wizard to configure project parameters of applications. It is highly recommended to use the Wizard to prevent inputting invalid parameter values.



Option	Value
Basic configuration	
Enable system fault trace module	ENABLE
Enable APP driver module	ENABLE
Eanble APP log module	ENABLE
Enable SK GUI module	ENABLE
Enable DTM test support	DISABLE
Enable BLE DFU support	ENABLE
Enable PMU calibration	ENABLE
Protection priority level	0
The Number of sectors for NVDS	1
Call stack size	0x0000 8000
RAM size of application	0x0003 0000
Code size of application	0x0080 0000
Boot info configuration	
Code load address	0x0100 2000
Code run address	0x0100 2000
System clock	64MHZ
Low power clock accuracy	500
Enable internal osc as low power clock	Default: Disable internal osc as low power clock
Delay time for boot startup	Delay 1s
Enable image check during cold boot startup in xip mode	ENABLE
Code version.16bits	1
Enable DAP boot mode	ENABLE
BLE resource configuration	
Maximum number of BLE profiles	10
Maximum number of bonded devices	4
Maximum number of BLE links	10
Mesh support configuration	
Mesh support	NOT SUPPORT

Figure 4-7 Configuration Wizard for *custom_config.h*

4.3.2.2 Configuring Memory Layout

Keil defines memory segments for the linker in .sct files. The GR551x SDK provides an example *flash_scatter_common.sct* for application developers. The macros used by this .sct file are defined in the *flash_scatter_config.h*.

Note:

In Keil, __attribute__((section("name"))) can be used to place a function or a variable at a separate memory segment, and the "name" depends on your choice. A scatter (.sct) file specifies the location for a named segment. For example, place Zero-Initialized (ZI) data of applications at the segment named "__attribute__((section(".bss.app")))".

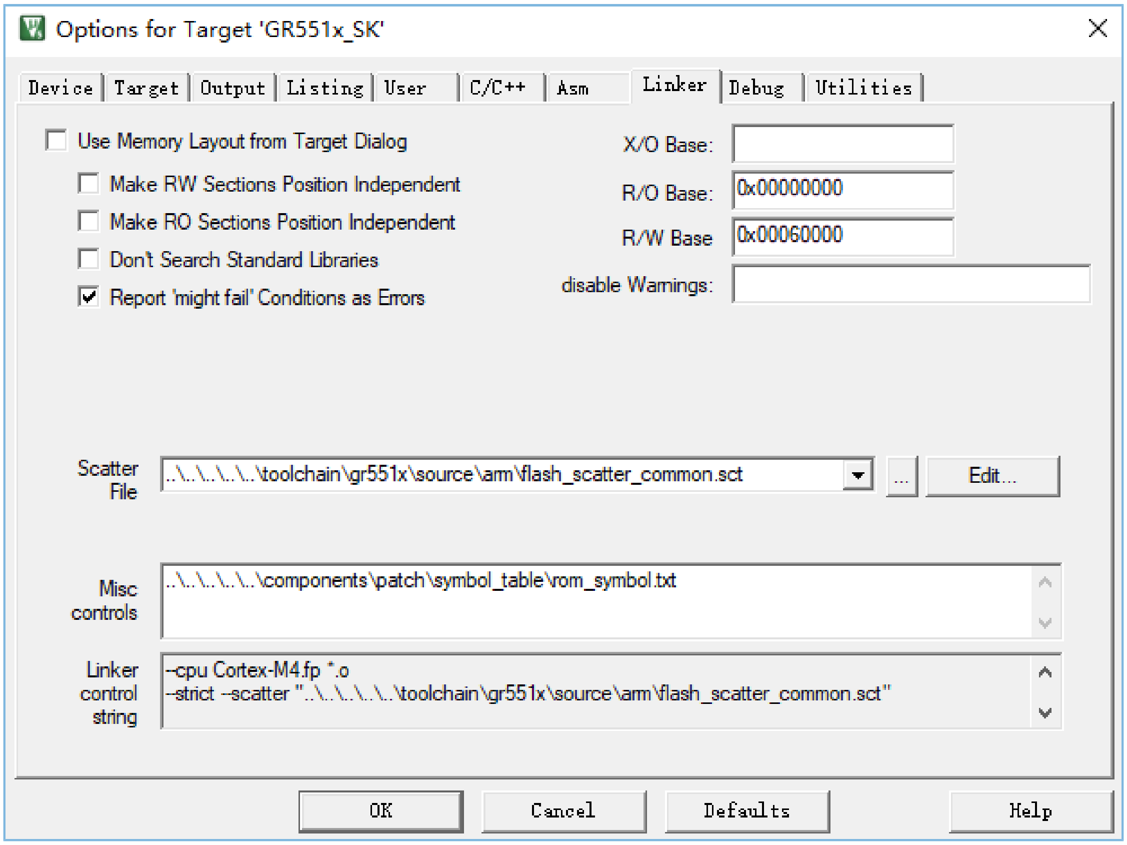
You can follow the steps below to configure the memory layout:

1. Click (**Options for Target**) on the Keil toolbar and open the **Options for Target 'GR551x_SK'** dialog box. Select the **Linker** tab.
2. On the **Scatter File** bar, click **...** to browse and select the *flash_scatter_common.sct* file in `SDK_Folder\toolchain\gr551x\source\arm`; or copy the scatter (.sct) file and its .h file to the ble_app_example project directory and then select the scatter file.

Note:

"#! armcc -E -I ..\Src\config\ --cpu Cortex-M4" in the *flash_scatter_common.sct* specifies an Include path, which is the path of the *custom_config.h* of an application project. A wrong path results in a linker error.

3. Click **Edit...** to open the .sct file, and modify corresponding code based on product memory layout.



☐ Use Memory Layout from Target Dialog
☐ Make RW Sections Position Independent
☐ Make RO Sections Position Independent
☐ Don't Search Standard Libraries
☑ Report 'might fail' Conditions as Errors

X/O Base:
R/O Base: 0x00000000
R/W Base 0x00060000
disable Warnings:

Scatter File: ..\..\..\..\..\toolchain\gr551x\source\arm\flash_scatter_common.sct

Misc controls: ..\..\..\..\..\components\patch\symbol_table\rom_symbol.txt

Linker control string:

```
--cpu Cortex-M4.fp *.o
--strict --scatter "..\..\..\..\..\toolchain\gr551x\source\arm\flash_scatter_common.sct"
```

Figure 4-8 Configuration of scatter file

4. Click **OK** to save the settings.

4.3.2.3 Configuring After Build

After Build in Keil can specify a command line to run after a project is built. By default, the ble_app_template project is equipped with `After Build` command. You do not need to configure **After Build** manually for the ble_app_example project that is based on *ble_app_template*.

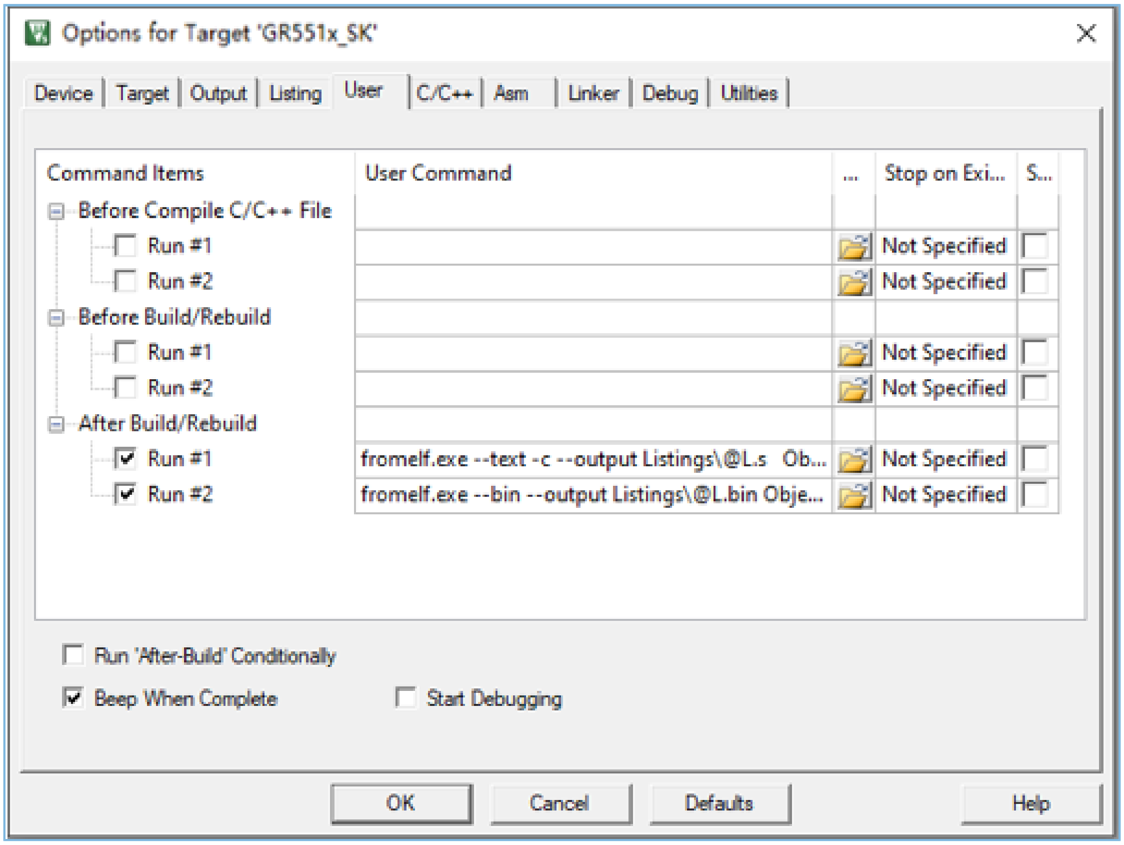
If you build a project, follow the steps below to configure **After Build**:

1. Click (**Options for Target**) on the Keil toolbar and open the **Options for Target 'GR551x_SK'** dialog box. Select the **User** tab.

2. From the options expanded from **After Build/Rebuild**, select **Run #1**, and type **fromelf.exe --text -c --output Listings\@L.s Objects\@L.axf** in the corresponding **User Command** field. This step helps you utilize Keil fromelf to generate a compiling file based on the selected .axf file.

3. From the options expanded from **After Build/Rebuild**, select **Run #2**, and type **fromelf.exe --bin --output Listings \@L.bin Objects\@L.axf** in the corresponding **User Command** field. This step helps you utilize Keil fromelf to generate a .bin file based on the selected .axf file.

4. Click **OK** to save the settings.



Command Items	User Command	...	Stop on Exi...	S...
⊟ Before Compile C/C++ File				
☐ Run #1			Not Specified	☐
☐ Run #2			Not Specified	☐
⊟ Before Build/Rebuild				
☐ Run #1			Not Specified	☐
☐ Run #2			Not Specified	☐
⊟ After Build/Rebuild				
☑ Run #1	fromelf.exe --text -c --output Listings\@L.s Ob...		Not Specified	☐
☑ Run #2	fromelf.exe --bin --output Listings\@L.bin Obje...		Not Specified	☐

☐ Run 'After-Build' Conditionally

☑ Beep When Complete ☐ Start Debugging

Figure 4-9 Configuration of **After Build**

4.3.3 Adding User Code

You can modify corresponding code in the *ble_app_example* on demand.

4.3.3.1 Modifying the main() Function

Code of a typical *main.c* file is provided as follows:

```
/**@brief Stack global variables for Bluetooth protocol stack. */
STACK_HEAP_INIT(heaps_table);
…
int main (void)
{
    /** Initialize user peripherals. */
    app_periph_init();

    /** Initialize BLE Stack. */
    ble_stack_init(&&m_app_ble_callback, &heaps_table);

    // Main Loop
    while (1)
    {
       /*
        * Add Application code here, e.g. GUI Update.
        */
        app_log_flush();
        pwr_mgmt_schedule();
    }
}
```

- STACK_HEAP_INIT(heaps_table) defines four global arrays as Heaps for BLE Stack. Do not modify the definition; otherwise, BLE Stack cannot work. For more information about Heap size, see CFG_MAX_CONNECTIONS in "Section 4.3.2.1 Configuring custom_config.h".
- You can initialize peripherals in app_periph_init(). In development and debugging phases, the SYS_SET_BD_ADDR in this function can be used to set a temporary Public Address. The *user_periph_setup.c* in which this function is contained includes the following main code:

```
/**@brief Bluetooth device address. */
static const uint8_t s_bd_addr[SYS_BD_ADDR_LEN] = {0x11, 0x11, 0x11, 0x11,0x11, 0x11};
…
void app_periph_init(void)
{
   SYS_SET_BD_ADDR(s_bd_addr);
   bsp_uart_init();
   app_log_assert_init();
   pwr_mgmt_mode_set(PMR_MGMT_SLEEP_MODE);
}
```

- You should add main loop code of applications to "while(1) { }", for example, code to handle external input and update GUI.
- When using the APP LOG module, call the app_log_flush() in the main loop. This is to ensure logs are output completely before the SoC enters Sleep Mode. For more information about the APP LOG module, see "Section 4.6.4 Outputting Debug Logs".
- Call the pwr_mgmt_shcedule() to implement automatic power management to reduce system power consumption.

4.3.3.2 Implementing Bluetooth LE Business Logics

Related Bluetooth LE business logics of applications are driven by a number of Bluetooth LE SDK callbacks which are defined in the GR551x SDK. Applications need to register these callbacks in the GR551x SDK to obtain operation results or state change notifications of BLE Stack. Bluetooth LE SDK callbacks are called in the interrupt context of Bluetooth LE SDK IRQ. Therefore, do not perform long-running operations in callbacks, for example, blocking function call and infinite loop; otherwise, the system is blocked, causing BLE Stack and the SDK Bluetooth LE module unable to run in a normal timing.

Bluetooth LE SDK callbacks are categorized into different files by GAP, Security Manager, L2CAP, GATT Common, and GATT Client. All callback functions supported by the GR551x SDK are listed in Table 4-5.

Table 4-5 Bluetooth LE SDK callback functions

File Name	Callback Struct	Callback Function
user_app.c	app_callback_t	app_ble_init_cmp_callback
user_gap_callback.c	gap_cb_fun_t	app_gap_param_set_cb
		app_gap_psm_manager_cb
		app_gap_phy_update_cb
		app_gap_dev_info_get_cb
		app_gap_adv_start_cb
		app_gap_adv_stop_cb
		app_gap_scan_req_ind_cb
		app_gap_adv_data_update_cb
		app_gap_scan_start_cb
		app_gap_scan_stop_cb
		app_gap_adv_report_ind_cb
		app_gap_sync_establish_cb
		app_gap_stop_sync_cb
		app_gap_sync_lost_cb
		app_gap_connect_cb
		app_gap_disconnect_cb
		app_gap_connect_cancel_cb
		app_gap_auto_connection_timeout_cb
		app_gap_peer_name_ind_cb
		app_gap_connection_update_cb
		app_gap_connection_update_req_cb
		app_gap_connection_info_get_cb
		app_gap_peer_info_get_cb
		app_gap_le_pkt_size_info_cb
		app_rslv_addr_read_cb
user_l2cap_callback.c	l2cap_lecb_cb_fun_t	app_l2cap_lecb_conn_req_cb
		app_l2cap_lecb_conn_cb
		app_l2cap_lecb_add_credits_ind_cb
		app_l2cap_lecb_disconn_cb
		app_l2cap_lecb_sdu_recv_cb
		app_l2cap_lecb_sdu_send_cb
		app_l2cap_lecb_credit_add_cmp_cb

File Name	Callback Struct	Callback Function
user_sm_callback.c	sec_cb_fun_t	app_sec_enc_req_cb
		app_sec_enc_ind_cb
		app_sec_keypress_notify_cb
user_gatt_common_callback.c	gatt_common_cb_func_t	app_gatt_mtu_exchange_cb
		app_gatt_prf_register_cb
user_gattc_callback.c	gattc_cb_fun_t	app_gattc_srvc_disc_cb
		app_gattc_inc_srvc_disc_cb
		app_gattc_char_disc_cb
		app_gattc_char_desc_disc_cb
		app_gattc_read_cb
		app_gattc_write_cb
		app_gattc_ntf_ind_cb
		app_gattc_srvc_browse_cb
		app_gattc_cache_update_cb

You need to implement necessary Bluetooth LE SDK callbacks according to functional requirements of your products. For example, if a product does not support Security Manager, you do not need to implement corresponding callbacks; if the product supports GATT Server only, you do not need to implement the callbacks corresponding to GATT Client. Only those callback functions required for products are to be implemented.

For more information about the usage of Bluetooth LE APIs and callback APIs, see the source code of Bluetooth LE examples in *GR551x Bluetooth Low Energy Stack User Guide*, `SDK_Folder\documentation\GR551x_API_Reference`, and `SDK_Folder\projects\ble`.

4.3.3.3 Scheduling BLE_Stack_IRQ, BLE_SDK_IRQ, and Applications

BLE Stack is the implementation core of BLE protocol stacks. It directly operates the hardware mentioned in *Bluetooth 5.1 Core* (see "Section 2.2 Software Architecture"). Therefore, BLE_Stack_IRQ has the second-highest priority after SVCall IRQ, which ensures that BLE Stack runs strictly in a time sequence specified in *Bluetooth Core Spec*.

Note:

The system_priority_init() in `SDK_Folder\toolchain\gr551x\source\system_gr55xx.c` is used to set default interrupt priority of modules in the system.

A state change of BLE Stack triggers BLE_SDK_IRQ interrupt with lower priority. In this interrupt handler, the Bluetooth LE SDK callbacks (to be executed in applications) are called to send state change notifications of BLE Stack and related business data to applications. You should avoid performing long-running operations in these callbacks, and shall move such operations to the main loop or user thread for processing. You can use the module in `SDK_Folder\components\libraries\app_queue`, or your own application framework, to transfer events from Bluetooth LE SDK callbacks to the main loop. For more information about processing in the user thread, see *GR551x FreeRTOS Example Application*.

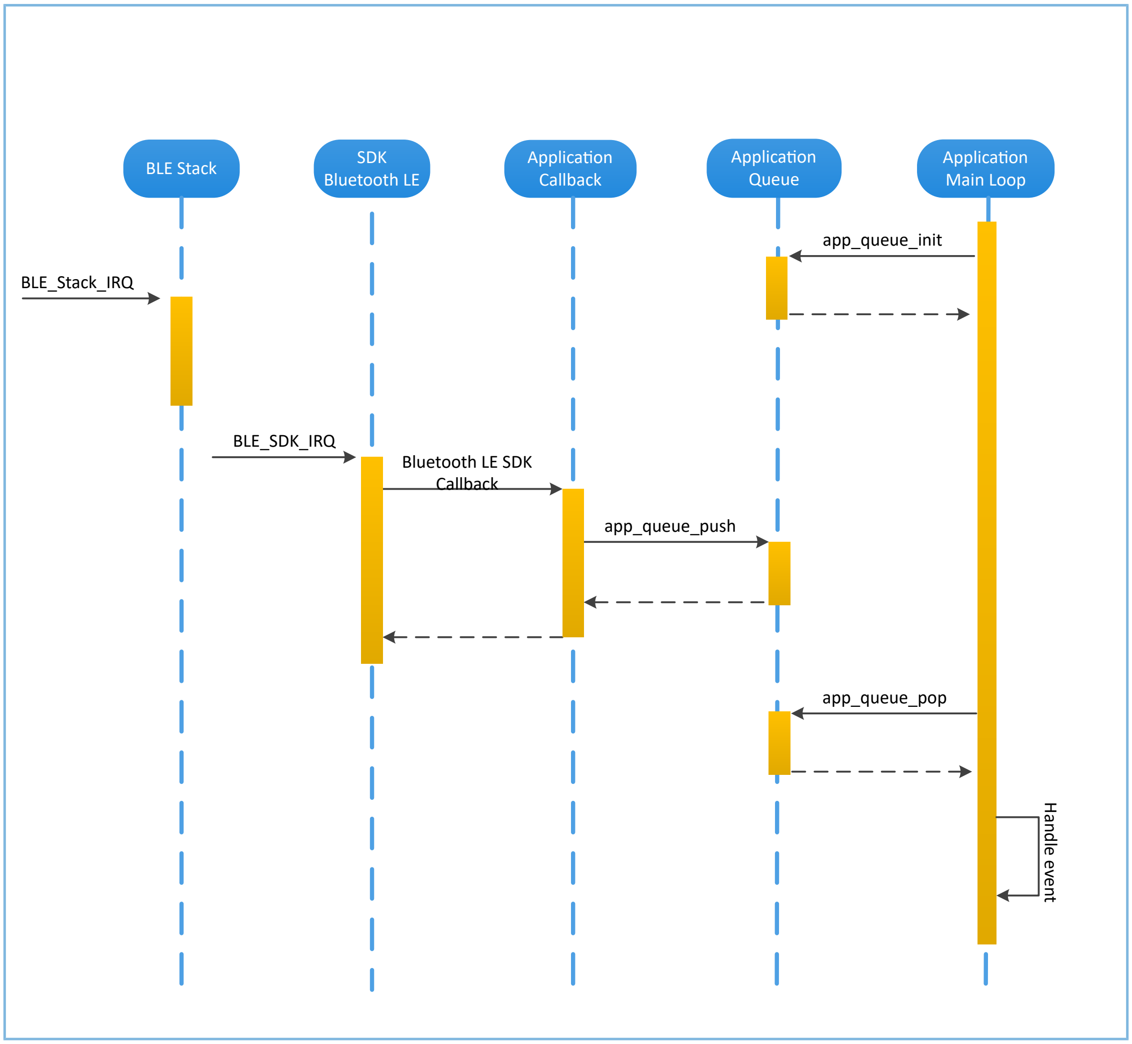


Figure 4-10 Non-OS system schedule

4.4 Generating Firmware

After building a Bluetooth LE application, you can directly click **Build** on the Keil toolbar to build a project. After the project is built, the following firmware files are generated in `Keil_5\Listings` and `Keil_5\Objects` respectively in the project directory. Both of the firmware files can be downloaded to a GR551x SoC via GProgrammer. For details, see *GProgrammer User Manual*. The *ble_app_example.hex* file can be downloaded to a GR551x SoC via Keil MDK.

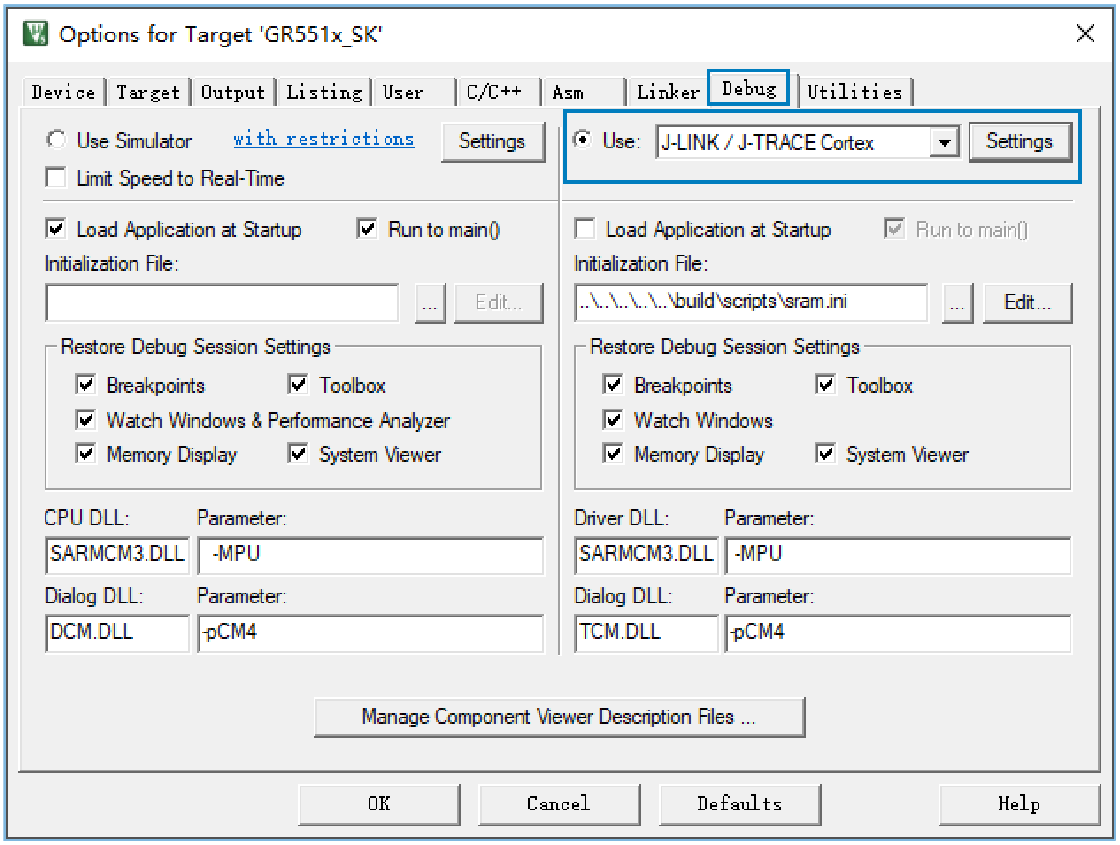
Table 4-6 Generated firmware

Name	Description
ble_app_example.bin	Binary application firmware; can be downloaded to an SoC through GProgrammer
ble_app_example.hex	Hexadecimal application firmware; can be downloaded to an SoC through GProgrammer or Keil MDK

4.5 Downloading .hex Files to Flash

After .hex files are generated, you need to download these files to flash. Specific steps are provided below:

1. Configure Keil flash programming algorithm.
    (1). Copy the *GR5xxx_16MB_Flash.FLM* in `SDK_Folder\build\binaries\xflash_flm_tools\Keil` to `Keil_Folder\ARM\Flash`.
    (2). Click (**Options for Target**) on the Keil toolbar, open the **Options for Target 'GR551x_SK'** dialog box, and select **Debug** tab. Click **Settings** on the right side of **Use: J-LINK/J-TRACE Cortex**.



○ Use Simulator

☐ Limit Speed to Real-Time

☑ Load Application at Startup ☑ Run to main()

Initialization File:

Restore Debug Session Settings

☑ Breakpoints ☑ Toolbox

☑ Watch Windows & Performance Analyzer

☑ Memory Display ☑ System Viewer

CPU DLL: SARMCM3.DLL Parameter: -MPU

Dialog DLL: DCM.DLL Parameter: -pCM4

● Use: J-LINK / J-TRACE Cortex

☐ Load Application at Startup ☑ Run to main()

Initialization File: ..\..\..\..\..\build\scripts\sram.ini

Restore Debug Session Settings

☑ Breakpoints ☑ Toolbox

☑ Watch Windows

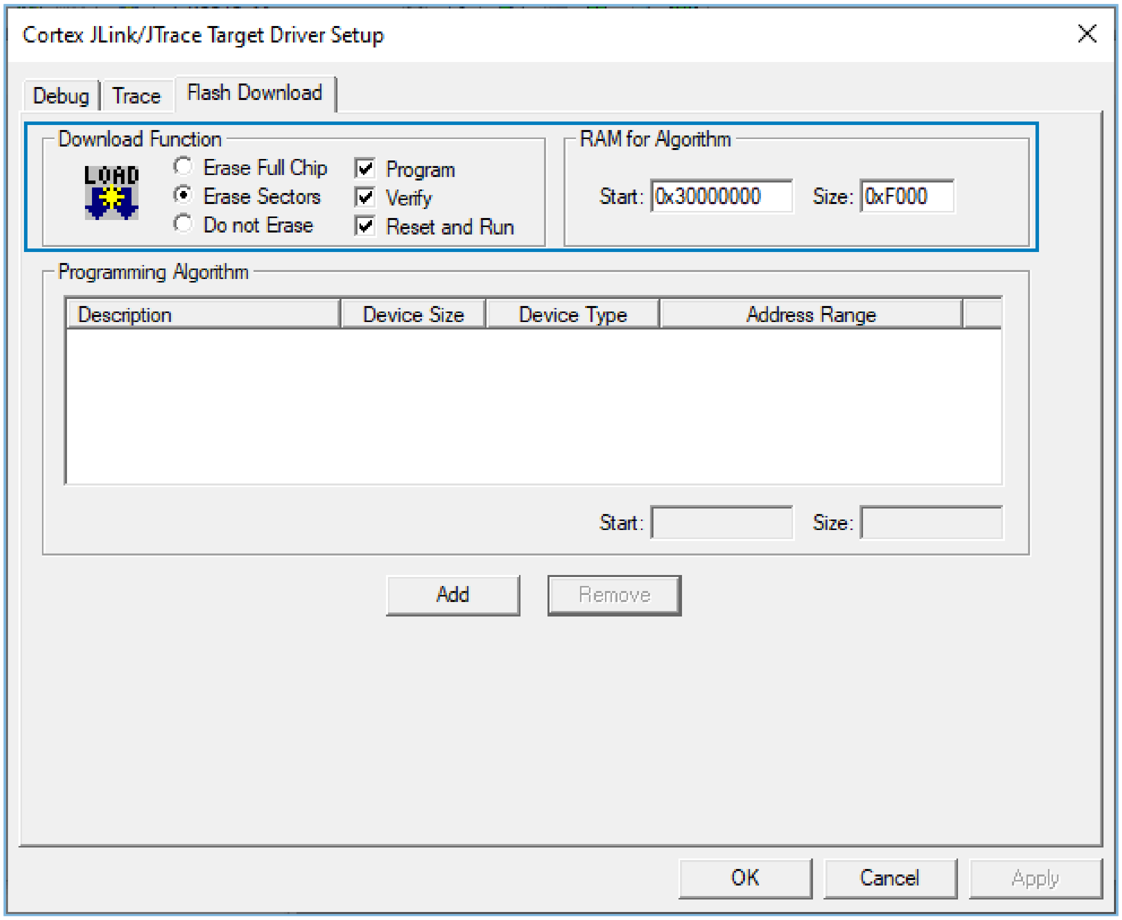
☑ Memory Display ☑ System Viewer

Driver DLL: SARMCM3.DLL Parameter: -MPU

Dialog DLL: TCM.DLL Parameter: -pCM4

Figure 4-11 **Debug** tab

    (3). In the **Cortex JLink/JTrace Target Driver Setup** window, select **Flash Download**. In the **Download Function** pane, you can set the erase type and check optional items: **Program**, **Verify**, and **Reset and Run**. Default configurations of Keil are shown below:



Download Function

○ Erase Full Chip

● Erase Sectors

○ Do not Erase

☑ Program

☑ Verify

☑ Reset and Run

RAM for Algorithm

Start: 0x30000000 Size: 0xF000

Start: Size:

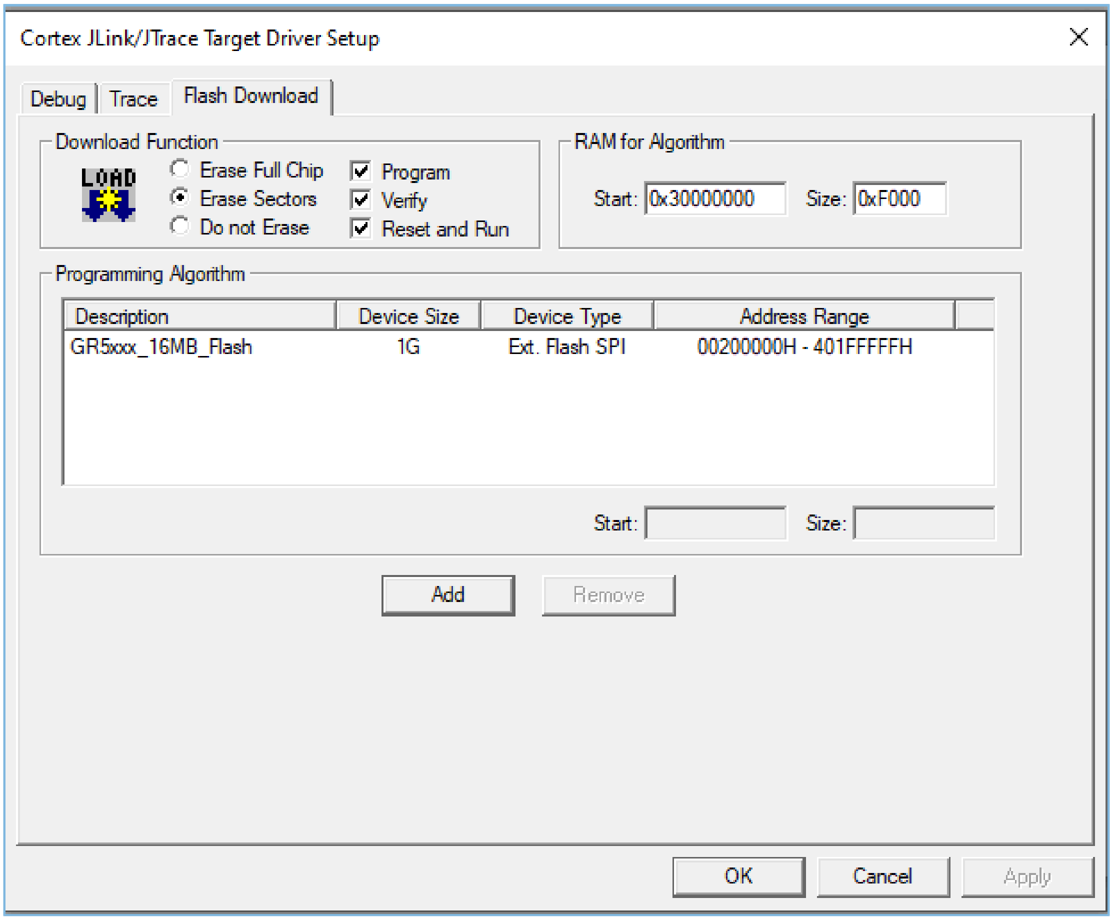
Figure 4-12 Choosing **Download Function**

Note:

If the Bootloader is in Mirror mode, you should change the erase type to **Erase Full Chip**.

---

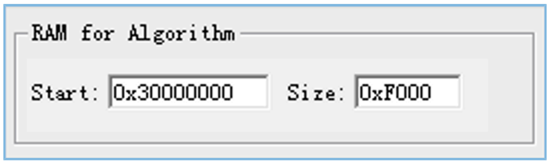
(4). Click **Add** to add the *GR5xxx_16MB_Flash.FLM* to the **Programming Algorithm**.



Description	Device Size	Device Type	Address Range
GR5xxx_16MB_Flash	1G	Ext. Flash SPI	00200000H - 401FFFFFH

Figure 4-13 Adding *GR5xxx_16MB_Flash* to **Programming Algorithm**

(5). Configure **RAM for Algorithm**, which defines address space to load and implement the programming algorithm. Enter the start address of RAM in GR551x in the **Start** input field: **0x30000000**. Enter **0xF000** in the **Size** input field.



Start: 0x30000000 Size: 0xF000

Figure 4-14 Settings of **RAM for Algorithm**

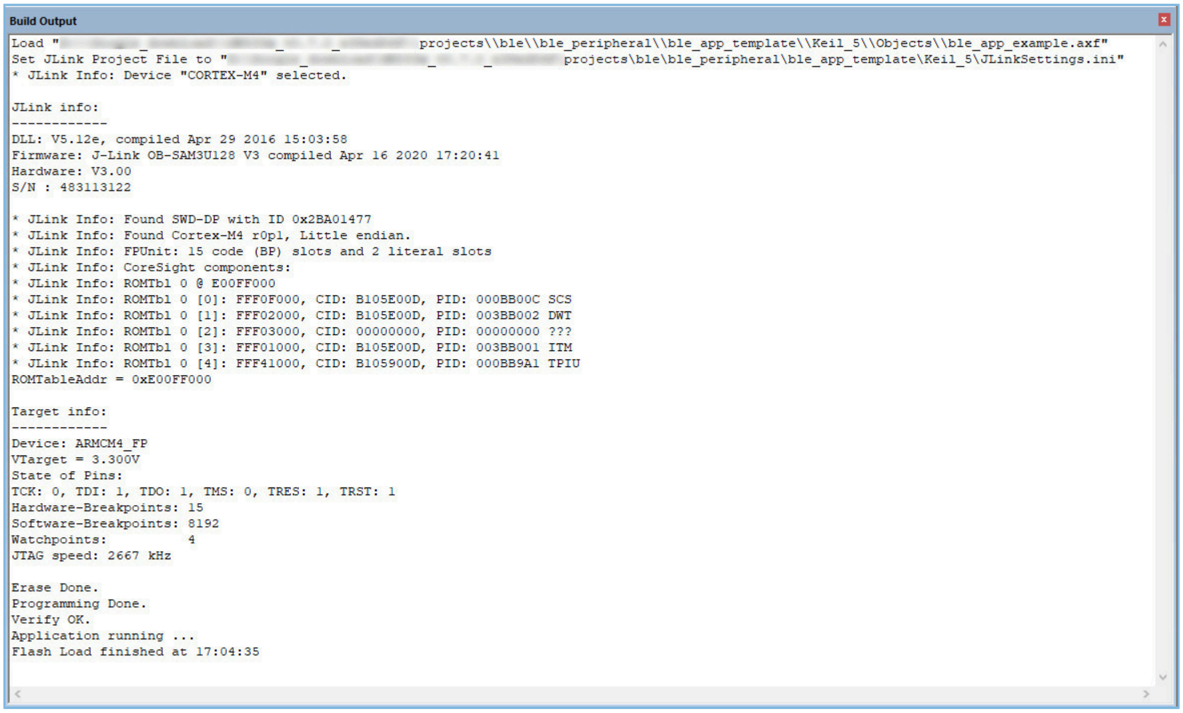
(6). Click **OK** to save the settings.

2. Download the .hex file.

After completing configuration, click LOAD (**Download**) on the Keil toolbar to download the .hex file to flash. After download is completed, the following results are displayed in the **Build Output** window of Keil.

Note:

During file download, if “No Cortex-M SW Device Found” pops up, it indicates the SoC may be in sleep state currently (the firmware with sleep mode enabled is running), so the .hex file cannot be downloaded to flash. In this case, developers need to press **RESET** on the GR5515 SK Board and wait for about 1 second; then click 📥 (**Download**) to re-download the file.



```
Load "[redacted]projects\\ble\\ble_peripheral\\ble_app_template\\Keil_5\\Objects\\ble_app_example.axf"
Set JLink Project File to "[redacted]projects\ble\ble_peripheral\ble_app_template\Keil_5\JLinkSettings.ini"
* JLink Info: Device "CORTEX-M4" selected.

JLink info:
------------
DLL: V5.12e, compiled Apr 29 2016 15:03:58
Firmware: J-Link OB-SAM3U128 V3 compiled Apr 16 2020 17:20:41
Hardware: V3.00
S/N : 483113122

* JLink Info: Found SWD-DP with ID 0x2BA01477
* JLink Info: Found Cortex-M4 r0p1, Little endian.
* JLink Info: FPUnit: 15 code (BP) slots and 2 literal slots
* JLink Info: CoreSight components:
* JLink Info: ROMTbl 0 @ E00FF000
* JLink Info: ROMTbl 0 [0]: FFF0F000, CID: B105E00D, PID: 000BB00C SCS
* JLink Info: ROMTbl 0 [1]: FFF02000, CID: B105E00D, PID: 003BB002 DWT
* JLink Info: ROMTbl 0 [2]: FFF03000, CID: 00000000, PID: 00000000 ???
* JLink Info: ROMTbl 0 [3]: FFF01000, CID: B105E00D, PID: 003BB001 ITM
* JLink Info: ROMTbl 0 [4]: FFF41000, CID: B105900D, PID: 000BB9A1 TPIU
ROMTableAddr = 0xE00FF000

Target info:
------------
Device: ARMCM4_FP
VTarget = 3.300V
State of Pins:
TCK: 0, TDI: 1, TDO: 1, TMS: 0, TRES: 1, TRST: 1
Hardware-Breakpoints: 15
Software-Breakpoints: 8192
Watchpoints:          4
JTAG speed: 2667 kHz

Erase Done.
Programming Done.
Verify OK.
Application running ...
Flash Load finished at 17:04:35
```

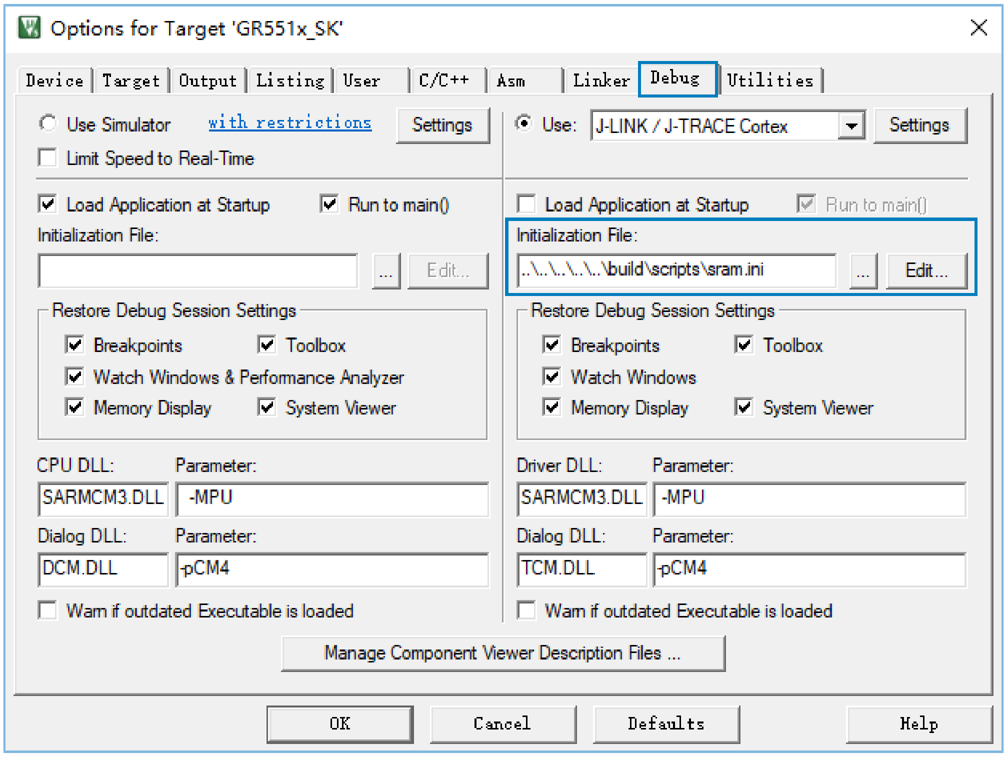
Figure 4-15 Download results

4.6 Debugging

Keil provides a debugger for online code debugging. The debugger supports setting six hardware breakpoints and multiple software breakpoints. It also provides developers with diverse debug commands.

4.6.1 Configuring the Debugger

Configure the debugger before debugging. Click 🔧 (**Options for Target**) on the Keil toolbar to open the **Options for Target ‘GR551x_SK’** dialog box, and then select **Debug** tab. In the window, software simulation debugging displays on the left, and online hardware debugging displays on the right. Bluetooth LE example projects adopt the online hardware debugging. Related default configurations of the debugger are shown as follows:



○ Use Simulator
☐ Limit Speed to Real-Time
☑ Load Application at Startup ☑ Run to main()
Initialization File:

Restore Debug Session Settings
☑ Breakpoints ☑ Toolbox
☑ Watch Windows & Performance Analyzer
☑ Memory Display ☑ System Viewer

CPU DLL: SARMCM3.DLL Parameter: -MPU
Dialog DLL: DCM.DLL Parameter: -pCM4
☐ Warn if outdated Executable is loaded

● Use: J-LINK / J-TRACE Cortex
☐ Load Application at Startup ☑ Run to main()
Initialization File: ..\..\..\..\..\build\scripts\sram.ini

Restore Debug Session Settings
☑ Breakpoints ☑ Toolbox
☑ Watch Windows
☑ Memory Display ☑ System Viewer

Driver DLL: SARMCM3.DLL Parameter: -MPU
Dialog DLL: TCM.DLL Parameter: -pCM4
☐ Warn if outdated Executable is loaded

Figure 4-16 Debugger configuration

The default initialization file *sram.ini* is in `SDK_Folder\build\scripts`. You can use this file directly, or copy it to the project directory.

Note:

SDK_Folder is the root directory of GR551x SDK.

The initialization file *sram.ini* contains a set of debug commands, which are executed during debugging. On the **Initialization File** bar, click **Edit...** on the right side, to open the *sram.ini* file. Example code of *sram.ini* is provided as follows:

```
/**
*****************************************************************
* GR551x object loading script through debugger interface
* (e.g.Jlink, *etc).
* The goal of this script is to load the Keils's object file to the
* GR551x RAM
* assuring that the GR551x has been previously cleaned up.
*****************************************************************
*/
// Debugger reset(check Keil debugger settings)
// Preselected reset type(found in Options->Debug->Settings)is
// Normal(0);
// -Normal:Reset core & peripherals via SYSRESETREQ & VECTRESET bit
// RESET
// Load object file
LOAD %L
// Load stack pointer
SP = _RDWORD(0x00000000)
// Load program counter
```

```
$ = _RDWORD(0x00000004)
// Write 0 to vector table register, remap vector
_WDWORD(0xE000ED08, 0x00000000)
```

Note:

Keil supports executing debugger commands set by developers in the following order:

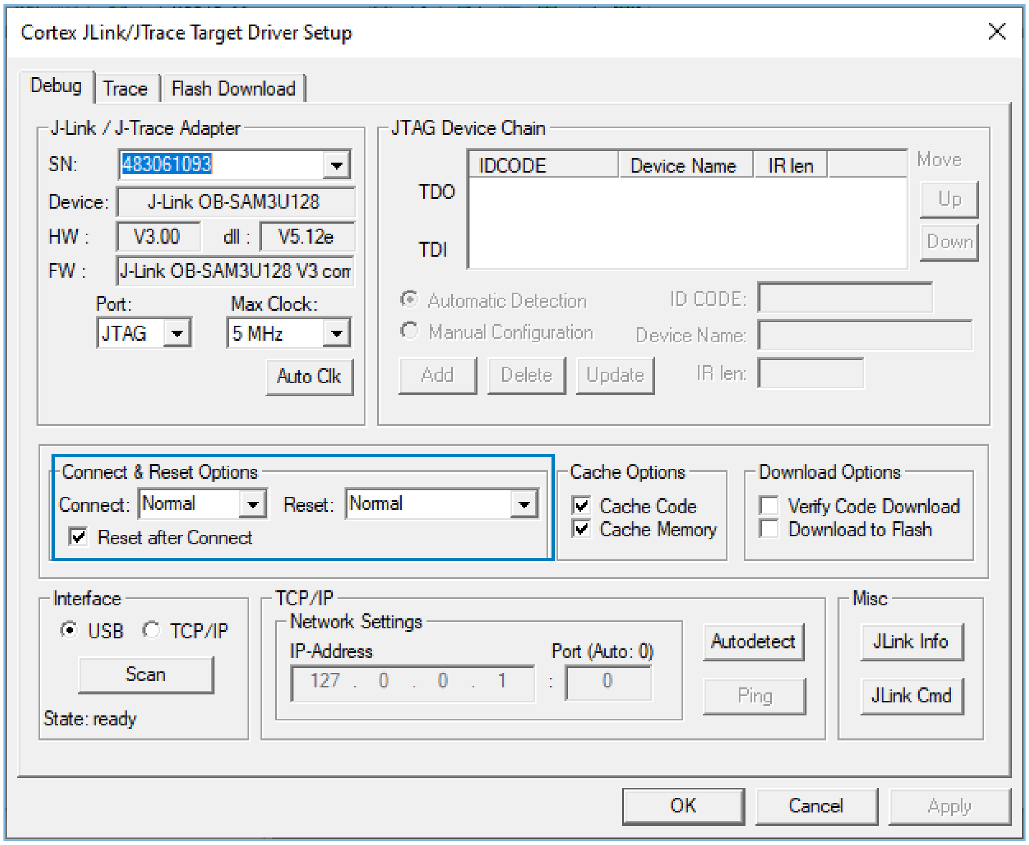
1. When **Load Application at Startup** (**Options for Target 'GR551x_SK'** > **Debug** > **Load Application at Startup**) is enabled, the debugger first loads the file under **Name of Executable** (**Options for Target 'GR551x_SK'** > **Output** > **Name of Executable**).
2. Execute the command in the file specified in **Options for Target 'GR551x_SK'** > **Debug** > **Initialization File**.
3. When options under **Options for Target 'GR551x_SK'** > **Debug** > **Restore Debug Session Settings** are checked, restore corresponding **Breakpoints**, **Watch Windows**, **Memory Display**, and other settings.
4. When **Options for Target 'GR551x_SK'** > **Debug** > **Run to main()** is checked, or the command `g,main` is discovered in the **Initialization File**, the debugger automatically starts executing CPU commands, until running to the main() function.

4.6.2 Starting Debugging

After completing debugger configuration, click (**Start/Stop Debug Session**) on the Keil toolbar, to start debugging.

Note:

Make sure that both options under **Connect & Reset Options** are set to **Normal**, as shown in Figure 4-17. This is to ensure when you click **Reset** on the Keil toolbar after enabling **Start Debug Session**, the program can run normally.



SN: 483061093
Device: J-Link OB-SAM3U128
HW: V3.00 dll: V5.12e
FW: J-Link OB-SAM3U128 V3 com
Port: JTAG
Max Clock: 5 MHz
● Automatic Detection
○ Manual Configuration
Connect: Normal
Reset: Normal
☑ Reset after Connect
☑ Cache Code
☑ Cache Memory
☐ Verify Code Download
☐ Download to Flash
● USB ○ TCP/IP
State: ready
IP-Address: 127 . 0 . 0 . 1
Port (Auto: 0): 0

Figure 4-17 Setting **Connect & Reset Options** to **Normal**

For debugging in XIP Mode, no other matters need attention. However, there are some additional notes for debugging in Mirror Mode. See "Section 4.6.3 Debugging in Mirror Mode".

4.6.3 Debugging in Mirror Mode

In Mirror Mode, you shall set breakpoints after the application firmware is copied to RAM.

Note:

If breakpoints are set within the RAM address range, Keil uses software breakpoints to save hardware resources (replace the original commands with BKPT instructions). After you set breakpoints, the Bootloader copies the application firmware to an address where breakpoints are set. The BKPT instructions of the address then are overwritten by the application firmware, and applications cannot stop when running to the address. For more information, see the ARM Keil official document *Breakpoints are not hit when debugging in RAM*.

You should set breakpoints before executing the main() function of applications. Follow the steps below to set breakpoints:

1. Add **__BKPT(X)** to the first line of the main() function. Example code is provided below:

```
int main(void)
{
    __BKPT(0);
    app_periph_init();                /*<init user periph .*/
```

```
…
```

2. Click **Build** on the Keil toolbar to compile and link code.
3. Click (**Start/Stop Debug Session**) on the Keil toolbar to start debugging. The application stops at **__BKPT(0)** when it starts debugging.
4. Set new breakpoints in the application.
5. Press F10 (not F5) to step over the next code line, so that you can continue debugging code in a normal way.

Note:

Pressing F10 allows to execute the next code line only; press F5 allows to execute all the rest code lines. Keil only responds to F10 when it hits **__BKPT**.

4.6.4 Outputting Debug Logs

The GR551x SDK supports outputting debug logs of applications from hardware ports in a customized output mode. Hardware ports include UART, J-Link RTT, and ARM Instrumentation Trace Macrocell (ARM ITM). The GR551x SDK provides an APP LOG module to facilitate log output. To use the APP LOG module, users need to enable APP_LOG_ENABLE in *custom_config.h*, and configure APP_LOG_PORT based on the needed output port.

4.6.4.1 Module Initialization

After configuration, you need to set log parameter by calling `app_log_init()` during peripheral initialization and to initialize the APP LOG module by registering log output APIs and Flush APIs. The APP LOG module supports using the `printf()` (a C standard library function) and APP LOG APIs to output debug logs. If you use APP LOG APIs, you can optimize logs by setting log level, log format, filter type, or other parameters; if you use `printf()`, the LOG parameter can be set to **NULL**.

Call the initialization function (see `SDK_Folder\components\libraries\bsp\bsp.h` for details) of the corresponding module according to the output port configured, and register corresponding Send and Flush APIs. See user_log_debug_init() for details. The APIs are provided as follows when UART is configured as the output port.

```
static void user_log_debug_init(void)
{
    app_log_init_t log_init;

    log_init.filter.level = APP_LOG_LVL_DEBUG;
    log_init.fmt_set[APP_LOG_LVL_ERROR] = APP_LOG_FMT_ALL & (~APP_LOG_FMT_TAG);
    log_init.fmt_set[APP_LOG_LVL_WARNING] = APP_LOG_FMT_LVL;
    log_init.fmt_set[APP_LOG_LVL_INFO] = APP_LOG_FMT_LVL;
    log_init.fmt_set[APP_LOG_LVL_DEBUG] = APP_LOG_FMT_LVL;

    app_log_init(&log_init, bsp_uart_send, bsp_uart_flush);

#if APP_LOG_STORE_ENABLE
    app_log_store_info_t store_info;
    app_log_store_op_t   op_func;

    store_info.nv_tag   = APP_LOG_NVDS_TAG;
    store_info.db_addr  = APP_LOG_DB_START_ADDR;
    store_info.db_size  = APP_LOG_DB_SIZE;
    store_info.blk_size = APP_LOG_ERASE_BLK_SIZE;

    op_func.flash_init  = hal_flash_init;
    op_func.flash_erase = hal_flash_erase;
    op_func.flash_write = hal_flash_write;
    op_func.flash_read  = hal_flash_read;
    op_func.time_get    = NULL;

    app_log_store_init(&store_info, &op_func);
#endif

}
```

Note:

- The input parameters of app_log_init() include the log initialization parameter, log output API, and Flush API (registration not required).
- GR551x SDK provides an APP LOG STORE module, which supports storing the debugging logs in flash and outputting the logs from flash. To use the APP LOG STORE module, users need to enable APP_LOG_STORE_ENABLE in *custom_config.h*. This module is configured in the ble_app_rscs project (in SDK_F`older\projects\ble\ble_peripheral\ble_app_rscs`), which can be used as a reference for users to use the APP LOG STORE module.
- Application logs output by using `printf()` cannot be stored by the APP LOG STORE module.

When the debugging logs are output through UART, the implemented log output API and Flush API are bsp_uart_send and bsp_uart_flush, respectively. bsp_uart_send implements app_uart asynchronous output API (app_uart_transmit_async) and hal_uart synchronous output API (hal_uart_transmit). Users can choose a proper log output port according to specific applications. As a uart_flush API, bsp_uart_flush is used to output the to-be-sent data cached in the memory in interrupt mode. Contents in both bsp_uart_send and bsp_uart_flush can be overwritten by users.

When the debugging logs are output through J-Link RTT or ARM ITM, the implemented log output APIs are bsp_segger_rtt_send() and bsp_itm_send(). No Flush API is implemented when either J-Link RTT or ARM ITM is configured as the output port.

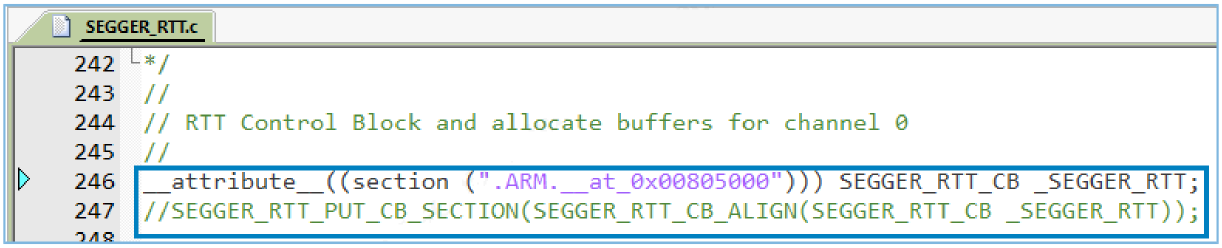
4.6.4.2 Application

After completing initialization of the APP LOG module, you can use any of the following four APIs to output debug logs:

- APP_LOG_ERROR()
- APP_LOG_WARNING()
- APP_LOG_INFO()
- APP_LOG_DEBUG()

In interrupt output mode, call `app_log_flush()` function to output all the debug logs cached, to ensure that all debug logs are output before the SoC is reset or the system enters the Sleep Mode.

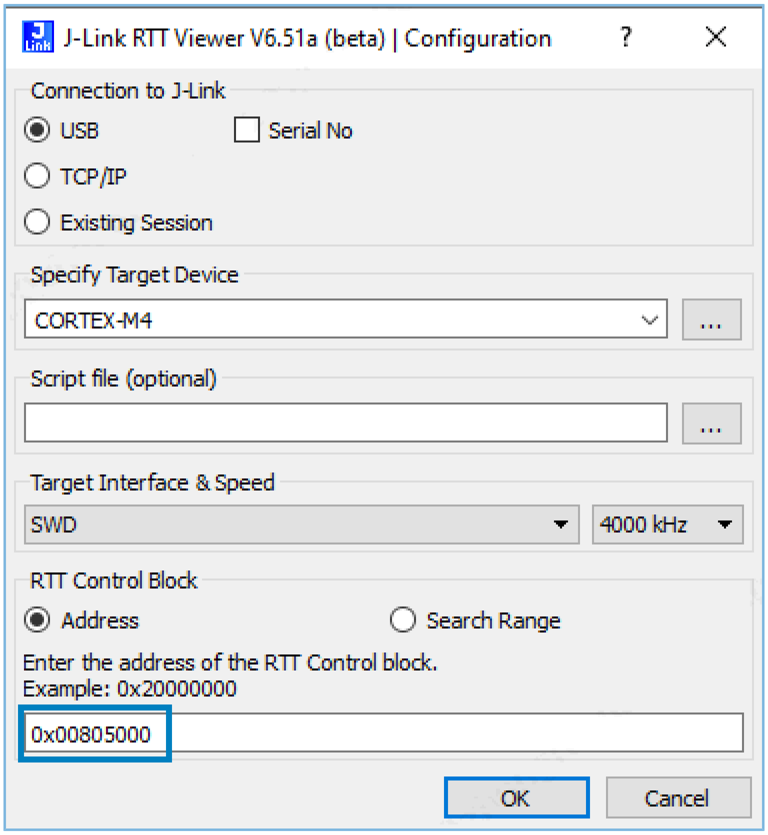
To output logs through J-Link RTT, it is recommended to modify *SEGGER_RTT.c* as follows.



```
*/
//
// RTT Control Block and allocate buffers for channel 0
//
__attribute__((section (".ARM.__at_0x00805000"))) SEGGER_RTT_CB _SEGGER_RTT;
//SEGGER_RTT_PUT_CB_SECTION(SEGGER_RTT_CB_ALIGN(SEGGER_RTT_CB _SEGGER_RTT));
```

Figure 4-18 To create an RTT Control Block and place it at 0x00805000

You also need to configure in J-Link RTT Viewer, as shown in Figure 4-19.



Connection to J-Link
● USB ☐ Serial No
○ TCP/IP
○ Existing Session

Specify Target Device: CORTEX-M4

Script file (optional):

Target Interface & Speed: SWD 4000 kHz

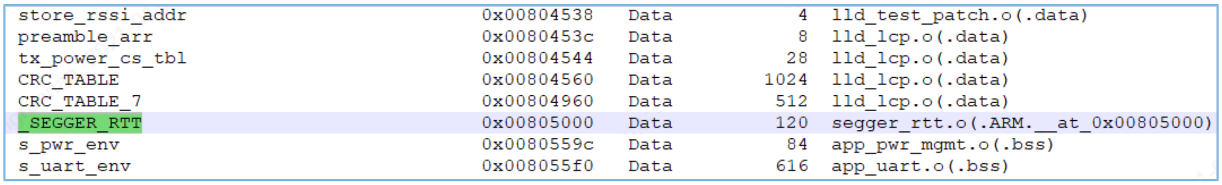
RTT Control Block
● Address ○ Search Range

Enter the address of the RTT Control block.
Example: 0x20000000

0x00805000

Figure 4-19 To configure J-Link RTT Viewer

The address of **RTT Control Block** is specified in **Address**, the value of which can be obtained by inquiring the address of the **_SEGGER_RTT** structure in the .map file (generated during project compilation). If an RTT Control Block has been created as recommended in Figure 4-18 and is placed at 0x00805000, then **0x00805000** can be entered in the **Address** field.



```
store_rssi_addr                        0x00804538   Data        4  lld_test_patch.o(.data)
preamble_arr                           0x0080453c   Data        8  lld_lcp.o(.data)
tx_power_cs_tbl                        0x00804544   Data       28  lld_lcp.o(.data)
CRC_TABLE                              0x00804560   Data     1024  lld_lcp.o(.data)
CRC_TABLE_7                            0x00804960   Data      512  lld_lcp.o(.data)
_SEGGER_RTT                            0x00805000   Data      120  segger_rtt.o(.ARM.__at_0x00805000)
s_pwr_env                              0x0080559c   Data       84  app_pwr_mgmt.o(.bss)
s_uart_env                             0x008055f0   Data      616  app_uart.o(.bss)
```

Figure 4-20 To obtain the RTT Control Block address

4.6.5 Debugging with GRToolbox

The GR551x SDK provides an Android App, GRToolbox, to debug GR551x Bluetooth LE applications, which is in `SDK_folder\tools\GRToolbox\GRToolbox-Version.apk`. GRToolbox integrates the following functions:

- General Bluetooth LE scanning and connecting; characteristics read/write
- Demos for standard profiles, including Heart Rate and Blood Pressure
- Goodix-customized applications

5 Development and Debugging with GR551x SDK in GCC

GNU Compiler Collection (GCC) is an open-source, cross-platform compiler system developed by the GNU Project running on both Linux and Windows operating systems.

Note:

For Linux operating systems, Ubuntu 16.04 LTS or later LTS versions are recommended.

For details on how to install and use GCC in applications development and debugging, see *GR551x GCC User Manual*.

6 Development and Debugging with GR551x SDK in IAR

IAR Embedded Workbench IDE for Arm (IAR EWARM, mentioned as IAR below) is an IDE built by IAR Systems, supporting Windows operating system. You can download the IAR installation file on IAR Systems official website. Currently, IAR for ARM 8.2.22 and later versions are supported on the SDK.

For details on how to install and use IAR in applications development and debugging, see *GR551x IAR User Manual*.

7 Glossary

Table 7-1 Glossary

Acronym	Description
ATT	Attribute Protocol
Bluetooth LE	Bluetooth Low Energy
DAP	Debug Access Port
DFU	Device Firmware Update
GAP	Generic Access Profile
GATT	Generic Attribute Profile
GFSK	Gaussian Frequency Shift Keying
HAL	Hardware Abstract Layer
HCI	Host-Controller Interface
IoT	Internet of Things
L2CAP	Logical Link Control and Adaption Protocol
LL	Link Layer
NVDS	Non-volatile Data Storage
OTA	Over The Air
PMU	Power Management Unit
PHY	Physical Layer
RF	Radio Frequency
SCA	System Configuration Area
SDK	Software Development Kit
SM	Security Manager
SoC	System on Chip
XIP	Execute in Place