

GR551x Developer Guide

Version: 1.6

Release Date: 2020-06-30

Shenzhen Goodix Technology Co., Ltd.

Copyright © 2020 Shenzhen Goodix Technology Co., Ltd. All rights reserved.

Any excerption, backup, modification, translation, transmission or commercial use of this document or any portion of this document, in any form or by any means, without the prior written consent of Shenzhen Goodix Technology Co., Ltd is prohibited.

Trademarks and Permissions

GODIX 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: 2F. & 13F., Tower B, Tengfei Industrial Building, Futian Free Trade Zone, Shenzhen, China

TEL: +86-755-33338828 FAX: +86-755-33338830

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 fourth 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 "Section 2.5.3 RAM Power Management" and "Section 4.6.4 Output Debug Logs".
1.5	2020-05-30	Updated the GR551x series models in "Chapter 1 Introduction" and the descriptions in "Section 2.3 Memory Mapping", "Section 2.6 GR551x SDK Directory Structure", and "Chapter 4 Development and Debugging with GR551x SDK".
1.6	2020-06-30	Optimized the SCA layout in "Section 2.4.1 SCA", and updated "Section 2.5.3 RAM Power Management" and "Section 4.6.4 Output Debug Logs".



Contents

Preface	I
1 Introduction	1
1.1 GR551x SDK	
1.2 BLE Stack	1
2 GR551x Bluetooth LE Software Platform	3
2.1 Hardware Architecture	
2.2 Software Architecture	4
2.3 Memory Mapping	5
2.4 Flash Memory Mapping	6
2.4.1 SCA	7
2.4.2 NVDS	
2.5 RAM Mapping	9
2.5.1 RAM Layout in XIP Mode	
2.5.2 RAM Layout in Mirror Mode	
2.5.3 RAM Power Management	
2.6 GR551x SDK Directory Structure	
3 Bootloader	
4 Development and Debugging with GR551x SDK	
4.1 Install Keil	
4.2 Install GR551x SDK	
4.3 Build a Bluetooth LE Application	19
4.3.1 Prepare ble_app_example	
4.3.2 Configure a Project	23
4.3.2.1 Configure custom_config.h	
4.3.2.2 Configure Memory Layout	
4.3.2.3 Configure After Build	
4.3.3 Add User Code	29
4.3.3.1 Modify the main() Function	29
4.3.3.2 Implement Bluetooth LE Business Logics	30
4.3.3.3 Schedule BLE_Stack_IRQ, BLE_SDK_IRQ, and Applications	
4.4 Generate .hex Files	33
4.5 Download .hex Files to Flash	
4.6 Debugging	
4.6.1 Configure the Debugger	
4.6.2 Start Debugging	
4.6.3 Debug in Mirror Mode	40
4.6.4 Output Debug Logs	41

_				
Co	nt	e	n	ts

4.6.5 Debug with GRToolbox	44
5 Glossary and Abbreviations	.45

1 Introduction

The Goodix GR551x 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.

GR551x Series	GR5515IGND	GR5515RGBD	GR5515GGBD	GR5513BEND
CPU	Cortex [®] -M4F	Cortex [®] -M4F	Cortex [®] -M4F	Cortex [®] -M4F
RAM	256 КВ	256 КВ	256 КВ	128 КВ
Flash	1 MB	1 MB	1 MB	512 КВ
Package	QFN56	BGA68	BGA55	QFN40

Table 1-1 Packages of GR551x SoCs

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.

BL	BLE Protocol Stack		
F	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)		
C	Controller		
Link Layer (LL)			
	Physical Layer (PHY)		
-			

Figure 1-1 BLE Stack architecture

GODIX

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 device.
- 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: <u>www.bluetooth.com</u>.

Specifications of GAP, SM, L2CAP, and GATT are provided in <u>Bluetooth Core Spec v5.1</u>. Specifications of other profiles/ services at the Bluetooth LE application layer are available on the <u>GATT Specs</u> page. Assigned numbers, IDs, and code which may be used by Bluetooth LE applications are listed on the <u>Assigned Numbers</u> page.

2 GR551x Bluetooth LE Software Platform

The GR551x 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*.

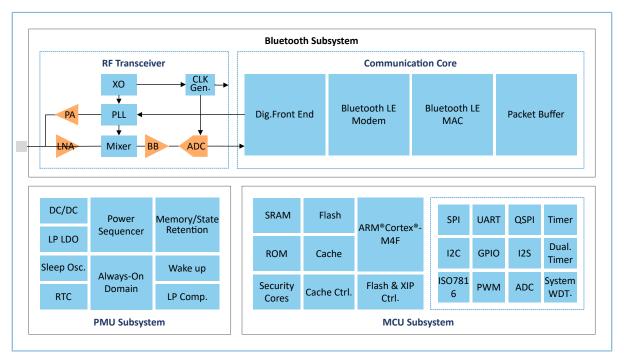


Figure 2-1 GR551x hardware architecture

- 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.

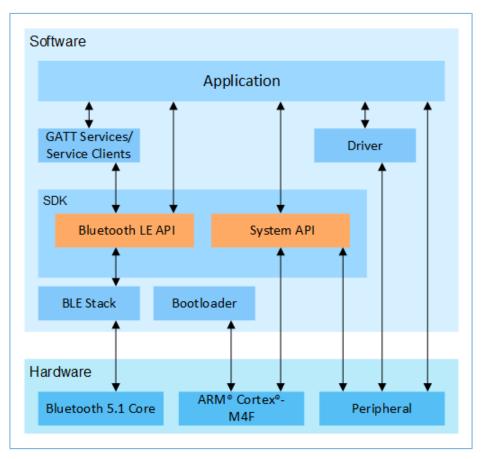


Figure 2-2 GR551x software architecture

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. Android applications in the SDK also provide 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.

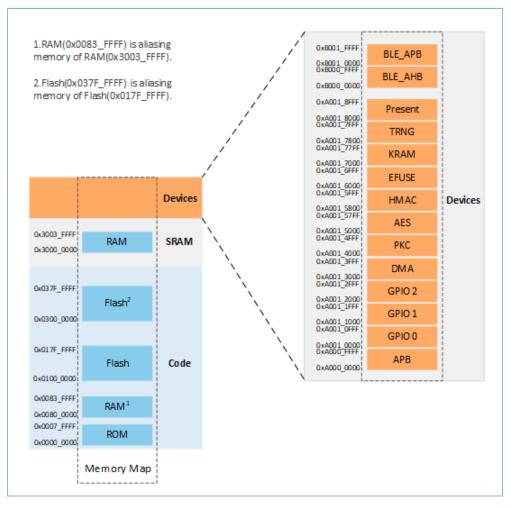


Figure 2-3 GR551x memory mapping

GODIX

🛄 Note:

• GR5515 SoCs:

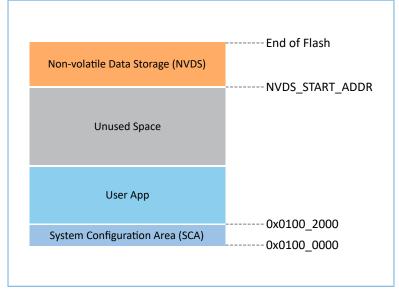
RAM: 0x3000_0000 to 0x3003_FFFF or 0x0080_0000 to 0x0083_FFFF, 256 KB in total Flash: 0x0100_0000 to 0x010F_FFFF or 0x0300_0000 to 0x030F_FFFF, 1 MB in total

• 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

GODIX

🛄 Note:

By default, NVDS occupies the last several sectors 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 Configure 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: SDK_Folder\Src\config), and programs the image info to SCA. See "Section 4.4 Generate .hex Files". Figure 2-5 shows the SCA layout.

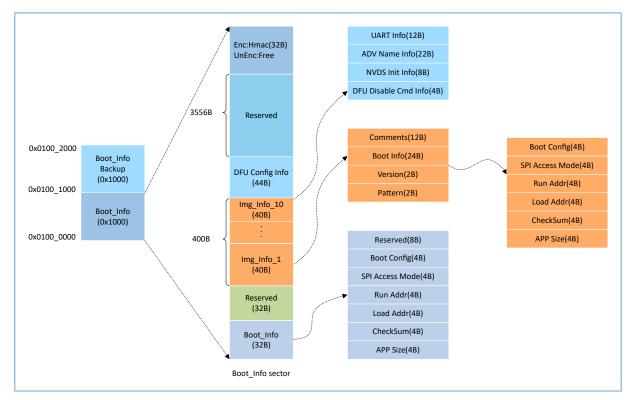


Figure 2-5 SCA layout

- 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). It 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.
- It is an ideal choice to store a small amount of small data blocks, for example, application configuration parameters, states, and user information.
- The default size is a sector (4 KB).
- One TAG supports up to 1024 bytes of data.

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

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
uint8 t nvds del(NvdsTag t tag)	exists, create one. 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.

Table 2-1 NVDS APIs

For more information about NVDS APIs, see GR551x API Reference.

🛄 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 several sectors of flash memory for 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 0x0001 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 Configure 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 2-6:

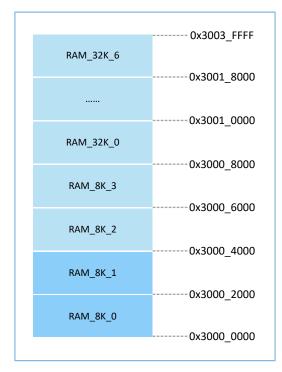


Figure 2-6 256 KB RAM layout

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 Configure custom_config.h". RAM layouts of the two modes are different.

Table 2-2 Ru	nning mo	des for ap	plications
--------------	----------	------------	------------

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 Coeffe Coeffe Coeffe
Mirror Mode	from flash directly through the Cache Controller. 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 2-7. You are able to modify the layout based on product needs.

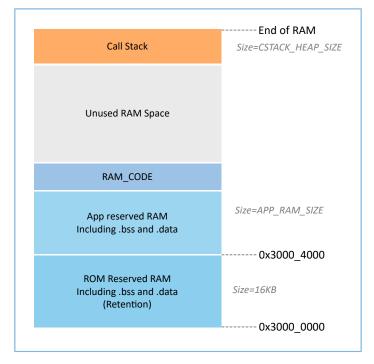


Figure 2-7 RAM layout in XIP Mode

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 Configure Memory Layout".

2.5.2 RAM Layout in Mirror Mode

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

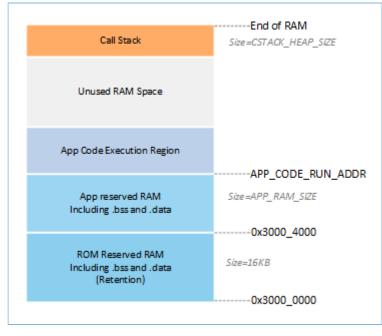


Figure 2-8 RAM layout in Mirror Mode

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".

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.

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.

GODIX

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.

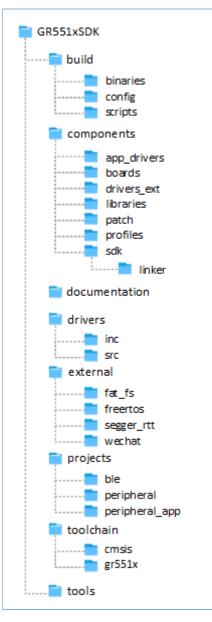
🛄 Note:

- 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 2-9.







Detailed descriptions of folders in the GR551x SDK are shown in Table 2-3.

Table 2-3	GR551x SDK folders
-----------	--------------------

Folder	Description
build\binaries	It contains tools used for the build process.
	It is the project configuration directory that stores the <i>custom_config.h</i> template file.
build\config	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.
componets\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.

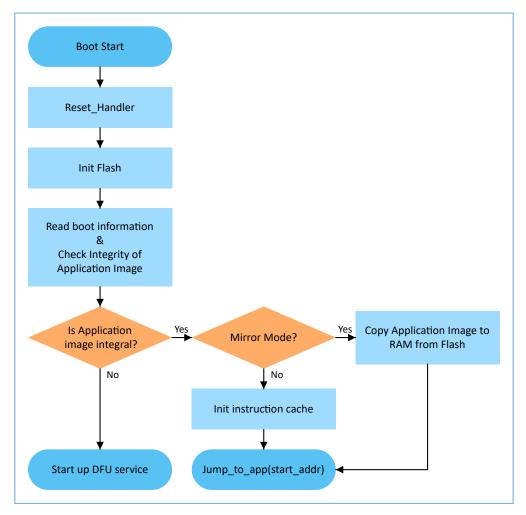


Figure 3-1 Application boot procedures of the GR551x SDK

- 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

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

4.1 Install Keil

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 <u>https://www.keil.com/</u><u>demo/eval/arm.htm</u>. 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: <u>http://www.keil.com/support/man_arm.htm</u>.

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

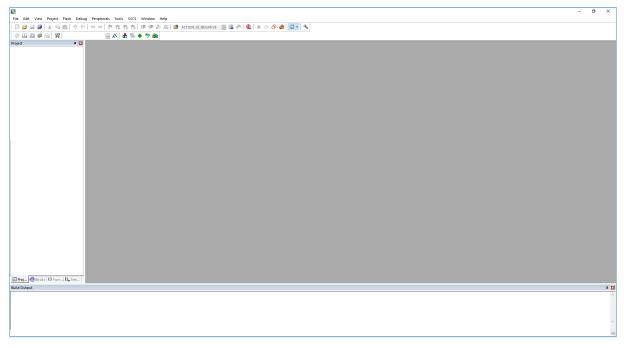


Figure 4-1 Keil interface

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

Table 4-1 Frequently used function buttons of Keil

Keil Icon	Description
ж.	Options for target
Q	Start/Stop Debug Session
Lond	Download



Keil Icon	Description
	Build

4.2 Install 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 Build a Bluetooth LE Application

This section introduces how to build a Bluetooth LE application.

4.3.1 Prepare 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*.

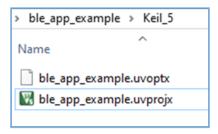


Figure 4-2 ble_app_example folder

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



🔣 Options for Target 'GR551x_SK'	×
Device Target Output Listing User C/C++ Asm Linker Debug Utilities	
Select Folder for Objects Name of Executable: ble_app_example	
 Create Executable: .\Objects\ble_app_example ✓ Debug Information Create HEX File ✓ Browse Information Create Library: .\Objects\ble_app_example.lib 	
OK Cancel Defaults Help	

Figure 4-3 Modifications to Name of Executable

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

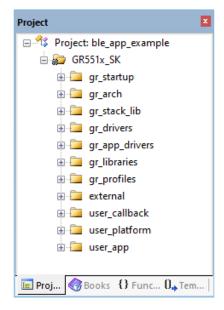


Figure 4-4 Project ble_app_example

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.

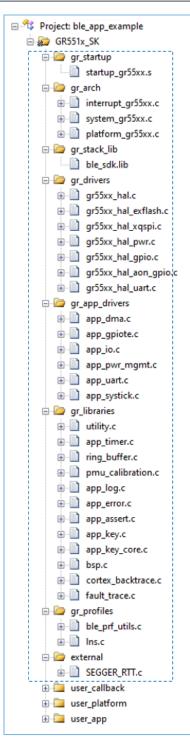


Figure 4-5 SDK groups

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.

GODIX

SDK Group Name	Description
	It contains initialization configuration files and system interrupt implementation files for System Core
gr_arch	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.

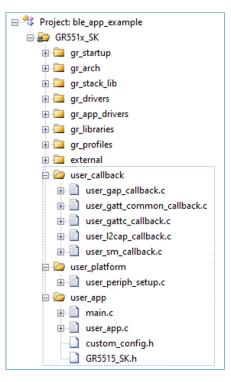


Figure 4-6 User groups

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.	
	It contains main() function entries, and other source files created by developers, which are used to	
user_app	configure runtime parameters of BLE Stack and execute event handlers of GATT Services/Service	
	Clients.	

4.3.2 Configure 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 Configure custom_config.h

The *custom_config.h* 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.

Macro	Description	
	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.	
SYS_FAULT_TRACE_ENABLE	0: Disable Callstack Trace Info printing.	
	1: Enable Callstack Trace Info printing.	
	It is used to enable/disable the App Drivers module.	
APP_DRIVER_USE_ENABLE	0: Disable the App Drivers module.	
	1: Enable the App Drivers module.	
	It is used to enable/disable the APP LOG module.	
APP_LOG_ENABLE	0: Disable the APP LOG module.	
	1: Enable the APP LOG module.	
	It is used to enable/disable the GUI module on GR5515 Starter Kit Board.	
SK_GUI_ENABLE	0: Disable the GUI module.	
	1: Enable the GUI module.	
DTM TEST ENABLE	It is used to enable/disable DTM Test.	
	0: Disable DTM Test.	

Table 4-4 Parameters in the c	custom config.h
	ustoni_conjig.n



Macro	Description	
	1: Enable DTM Test.	
	It is used to enable/disable DFU.	
DFU_ENABLE	0: Disable DFU.	
	1: Enable DFU.	
	It is used to enable/disable PMU CALIBRATION. When PMU CALIBRATION is enabled, the system	
	monitors temperatures and voltages automatically with adaptive adjustment. It shall be enabled in	
PMU_CALIBRATION_ENABLE	high/low temperature scenarios. It is recommended to enable this parameter by default.	
	0: Disable PMU CALIBRATION.	
	1: Enable PMU CALIBRATION.	
	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	
FLASH_PROTECT_PRIORITY	N are suspended. After erase is completed, flash responds to the suspended interrupt requests. By	
	default, flash does not respond to any interrupt request during erase. Developers can set a value on	
	demand.	
NVDS_NUM_SECTOR	It represents the number of flash sectors for NVDS.	
	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 32 KB.	
CSTACK_HEAP_SIZE	After compilation of ble_app_example, a Maximum Stack Usage is provided in	
	Keil_5\Objects\ble_app_example.htm for reference.	
	It represents the RAM size occupied by all global variables of applications. To allow disabling/	
APP_RAM_SIZE	enabling non-retention RAM Blocks in Sleep Mode, the RAM size shall be set to make the boundary	
	of the RAM segment align with that of the RAM Block.	
	It represents the size of code segments occupied by applications in flash. You can set the value	
APP_MAX_CODE_SIZE	according to the actual code size of applications.	
	It represents the start address of the application storage area. This address shall be within the flash	
APP_CODE_LOAD_ADDR*	address range.	
	It represents the start address of the application running space.	
APP_CODE_RUN_ADDR*	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.	
	It represents the system clock frequency. Optional values are provided as follows:	
	0: 64 MHz	
SYSTEM_CLOCK*	1: 48 MHz	
	2: 16 MHz (XO)	
	3: 24 MHz	



Macro	Description
	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).
	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.
CFG_LPCLK_INTERNAL_EN	0: Disable.
	1: Enable.
	It is used to set necessary 1-second delay (during SoC boot before implementing the second half
	Bootloader).
BOOT_LONG_TIME*	0: No delay.
	1: Delay for 1 second.
	It determines whether to check the image during cold boot in XIP mode.
BOOT_CHECK_IMAGE	0: Do not check.
	1: Check.
	It represents the version number of application firmware; length: 2 bytes; it is stored in hexadecimal
VERSION*	format.
	It is used to enable/disable DAP Boot Mode.
DAP_BOOT_ENABLE	0: Disable DAP Boot Mode.
	1: Enable DAP Boot Mode.
	It represents the maximum number of bonded devices supported by applications. You should set the
CFG_MAX_BOND_DEV_NUM	value on demand. A larger value means more RAM space to be occupied.
	It represents the maximum number of GATT Profiles/Services included in applications. Set the value
CFM_MAX_PRF_NB	on demand: A larger value means to occupy more RAM space.
	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
CFG_MAX_CONNECTIONS	Heaps is defined by the following four macros in <i>flash_scatter_config.h</i> , which cannot be changed by
	developers:
	ENV_HEAP_SIZE
	ATT_DB_HEAP_SIZE
	KE_MSG_HEAP_SIZE
	NON_RET_HEAP_SIZE
	Support Mesh or not.
CFG_MESH_SUPPORT	

Macro	Description
	1: Yes

*: The *ble_tools.exe* in the SDK toolchain reads the macro value to generate an SCA image file. The Bootloader reads the value from SCA and uses it as a boot parameter.

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.

custom_config.h Expand All Collapse All Help T Show Grid	▼ X
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
\ Text Editor	

Figure 4-7 Configuration Wizard for custom_config.h

4.3.2.2 Configure 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:

- 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.

Options for Target 'GR551x_SK'	×				
Device Target Output Listing User C/C++ Asm Linker Debug Utilities Use Memory Layout from Target Dialog X/O Base:	-				
□ Make RW Sections Position Independent R/O Base: 0x00000000 □ Make RO Sections Position Independent R/W Base 0x00060000 □ Don't Search Standard Libraries disable Warnings: □					
Scatter	Edit				
Misc\\\components\patch\symbol_table\rom_symbol.txt	\sim				
Linker control string					
OK Cancel Defaults	Help				

Figure 4-8 Configuration of scatter file

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

4.3.2.3 Configure After Build

After Build in Keil can specify an executable program or batch file to run after a project is built. By default, the ble_app_template project adds the *after_build.bat* file to **After Build**. You do not need to configure **After Build** manually for the ble_app_example project based on *ble_app_template*. For more information about functions of *after_build.bat* provided in the GR551x SDK, see "Section 4.4 Generate .hex Files".

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

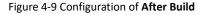
- Click Coptions for Target) on the Keil toolbar and open the Options for Target 'GR551x_SK' dialog box. Select the User tab.
- 2. From the options expanded from After Build/Rebuild, check Run #1, and then click it browse and select the after_build.bat file in SDK_Folder\build\scripts; or copy the after_build.bat file in SDK_Folder \build\scripts to the ble_app_example project directory and select the file.

🛄 Note:

Some relative paths are provided in the *after_build.bat* file, for example, the path of the SDK tools directory. If the batch file is no longer in SDK_Folder\build\scripts, modify these relative paths to avoid any **After Build** error.

Pevice Target Output Listin				
Command Items	User Command		Stop on Exi	S
Before Compile C/C++ File				
Run #1		2	Not Specified	\Box
🗌 🗌 Run #2		2	Not Specified	\Box
Before Build/Rebuild				
Run #1		2	Not Specified	\Box
🗌 🗌 Run #2		2	Not Specified	\Box
After Build/Rebuild				
🔽 Run #1	\\\\build\scripts\after_build.bat @L @K	2	Not Specified	\Box
2) 🗌 🗌 Run #2		2	Not Specified	
☐ Run 'After-Build' Conditionally ✓ Beep When Complete	Start Debugging			

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



4.3.3 Add User Code

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

4.3.3.1 Modify 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 Configure 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 Output Debug Logs".



 Call the pwr_mgmt_shcedule() to implement automatic power management to reduce system power consumption.

4.3.3.2 Implement 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.

File Name	Callback Struct	Callback Function
main.c	app_callback_t	app_ble_init_cmp_callback
	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
user_gap_callback.c		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

Table 4-5 Bluetooth LE SDK callback functions



File Name	Callback Struct	Callback Function
		app_gap_connection_info_get_cb
		app_gap_peer_info_get_cb
		app_gap_le_pkt_size_info_cb
		app_rslv_addr_read_cb
	l2cap_lecb_cb_fun_t	app_l2cap_lecb_conn_req_cb
		app_l2cap_lecb_conn_cb
		app_l2cap_lecb_add_credits_ind_cb
user_l2cap_callback.c		app_l2cap_lecb_disconn_cb
		app_l2cap_lecb_sdu_recv_cb
		app_I2cap_lecb_sdu_send_cb
		app_l2cap_lecb_credit_add_cmp_cb
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
		app_gattc_srvc_disc_cb
	gattc_cb_fun_t	app_gattc_inc_srvc_disc_cb
		app_gattc_char_disc_cb
		app_gattc_char_desc_disc_cb
user_gattc_callback.c		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*,

 ${\tt SDK_Folder\backslash documentation\backslash GR551x_API_Reference, and {\tt SDK_Folder\backslash projects\backslash ble}.$

4.3.3.3 Schedule 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\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*.

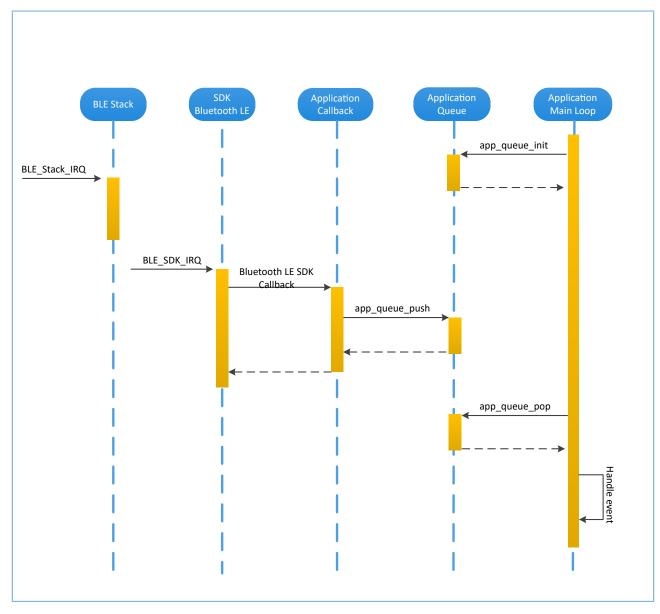


Figure 4-10 Non-OS system schedule

4.4 Generate .hex Files

After a Bluetooth LE application is built, Keil automatically runs the *after_build.bat*. The *after_build.bat* calls the *ble_tools.exe* to read configuration parameters in the user configuration file *custom_config.h*, and generates an SCA image file which merges with the information in the application firmware from Keil to generate a .hex file.

After building a Bluetooth LE application, you can directly click **Build** on the Keil toolbar to build a project. After the project is built, *ble_app_example_fw.bin*, *ble_app_example_fw.hex*, and *load_app.hex* are generated in Keil_5\build in the project directory.

🛄 Note:

- Contents in *load_app.hex* and *ble_app_example_fw.hex* are the same.
- The *load_app.hex* is used for firmware download with the same download script. For more information, see the *download.ini* file mentioned in "Section 4.5 Download .hex Files to Flash".

4.5 Download .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 *GR551x_8MB_Flash.FLM* in SDK_Folder\build\binaries 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.

Options for Target 'GR551x_SK'	×
Device Target Output Listing User C/C++ A	Asm Linker Debug Vtilities
C Use Simulator <u>with restrictions</u> Settings	O Use: J-LINK / J-TRACE Cortex ▼ Settings
Load Application at Startup Run to main() Initialization File:	Load Application at Startup Run to main() Initialization File:
Restore Debug Session Settings Image: Breakpoints Image: Toolbox Image: Breakpoints Image:	Restore Debug Session Settings
Dialog DLL: Parameter: DCM.DLL pCM4 Manage Component Vie	Dialog DLL: Parameter: TCM.DLL pCM4 ewer Description Files
OK Car	ncel Defaults Help

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:

Cortex JLink/JTrace Target Driver S	etup				×
Debug Trace Flash Download					
Download Function C Erase Full Chip Erase Sectors C Do not Erase		RAM for Alg		0xF000	
Programming Algorithm					
Description	Device Size De	evice Type	Address Rang	e	
		Start:	Size:		
	Add	Remove			
			ОК	Cancel App	yly

Figure 4-12 Choosing Download Function

(4). Click Add to add the GR551x 8MB Flash.FLM to the Programming Algorithm.



Cortex JLink/JTrace Target Driver S	etup			×
Debug Trace Flash Download				
Download Function C Erase Full Chip Erase Sectors Do not Erase	Verify		Algorithm Dx30000000 Size: 0xF00	0
- Programming Algorithm				
Description GR551x 8Mb Flash	Device Size 8M	Device Type Ext. Flash SPI	Address Range 01000000H - 017FFFFFH	
		Start: 🛛	Dx01000000 Size: 0x008	00000
	Add	Remove		
			OK Cancel	Apply

Figure 4-13 Adding the GR551x 8 MB flash 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.

RAM for Algorithm	
Start: 0x30000000	Size: 0xF000

Figure 4-14 Settings of RAM for Algorithm

- (6). Click **OK** to save the settings.
- 2. Configure the Configure Flash Menu Command.
 - (1). Click K (Options for Target) on the Keil toolbar, open the Options for Target 'GR551x_SK' dialog box, and select Utilities tab.
 - (2). On the Init File bar in the Configure Flash Menu Command pane, click ... to browse and select the *download.ini* file in SDK_Folder\build\scripts; or copy the *download.ini* file to the project directory and select it.



I Options for Target 'GR551x_SK'	Х
Device Target Output Listing User C/C++ Asm Linker Debug Utilities	
- Configure Flash Menu Command	
Use Debug Driver Settings 🗌 Update Target before Debugging	
Init File:	
C Use External Tool for Flash Programming	
Command:	
Arguments:	
🗖 Run Independent	
Configure Image File Processing (FCARM):	
Output File: Add Output File to Group:	
gr_startup 💌	
Image Files Root Folder:	
	,
OK Cancel Defaults Help	

Figure 4-15 Initialization file for programming

🛄 Note:

By default, Use Target Driver for Flash Programming and Use Debug Driver are checked.

Do not check Update Target before Debugging.

3. Download the .hex file.

After completing configuration, click (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.

Build Output	ņ	x
/**	_	~

*GR551x object loading script through download interface (e.g.Jlink, etc).		
*The goal of this script is to load the Keils's object file to the GR551x Flash		

*/		
//Load object file		
LOAD .\build\load_app.hex		
Erase Done.		
Programming Done.		
Verify OK.		
* JLink Info: Reset: Halt core after reset via DEMCR.VC_CORERESET.		
* JLink Info: Reset: Reset device via AIRCR.SYSRESETREQ.		
Application running		
Flash Load finished at 17:23:59		¥
<	>	

Figure 4-16 Download results

GODIX

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 Configure 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:

🕅 Options for Target 'GR551x_SK'	×
Device Target Output Listing User C/C++ A	Asm Linker Debug Utilities
C Use Simulator <u>with restrictions</u> Settings Limit Speed to Real-Time	Use: J-LINK / J-TRACE Cortex Settings
Load Application at Startup Run to main() Initialization File: Load Application at Startup Load Application Load Appl	Load Application at Startup Run to main() Initialization File: Edit Edit
Restore Debug Session Settings Breakpoints Toolbox Watch Windows & Performance Analyzer Memory Display System Viewer	Restore Debug Session Settings Image: Breakpoints Image: Toolbox Image: Breakpoints Image:
CPU DLL: Parameter: SARMCM3.DLL -MPU Dialog DLL: Parameter: DCM.DLL -pCM4	Driver DLL: Parameter: SARMCM3.DLL -MPU Dialog DLL: Parameter: TCM.DLL -pCM4
Wam if outdated Executable is loaded Manage Component Vie	Warn if outdated Executable is loaded
OK Car	ncel Defaults Help

Figure 4-17 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).
```

GODIX

```
* 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(0x0000000)
// Load program counter
= RDWORD(0x0000004)
// Write 0 to vector table register, remap vector
WDWORD(0xE000ED08, 0x0000000)
```

🛄 Note:

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

- 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 Start 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 4-18. This is to ensure when you click **Reset** on the Keil toolbar after enabling **Start Debug Session**, the program can run normally.



Cortex JLink/JTrace Target Driver Setup	×
Debug Trace Flash Download	
J-Link / J-Trace Adapter SN: 483061093 ▼ Device: J-Link OB-SAM3U128 HW : V3.00 dll : V5.12e FW : J-Link OB-SAM3U128 V3 com Port: Max Clock: JTAG ▼ 5 MHz ▼ Auto Clk	JTAG Device Chain IDCODE Device Name IR len Move TDO Up Down TDI Down Down © Automatic Detection ID CODE: Output © Manual Configuration Device Name: Output Add Delete Update IR len:
Connect & Reset Options Connect: Normal Reset: Normal Reset: Normal	mal Cache Options Download Options Cache Code Code Cache Memory Download to Flash
Interface TCP/IP © USB © TCP/IP Scan State: ready	Autodatest II ink Infe
	OK Cancel Apply

Figure 4-18 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 Debug in Mirror Mode".

4.6.3 Debug 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 <u>BKPT(X)</u> to the first line of the main() function. Example code is provided below:

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

GODIX

...

- 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 Output 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. You can use the following macros to configure output mode and hardware ports.

Macro	Description	Path
	It selects the hardware port used for	
	log output.	SDK_Folder\components\libraries
APP_LOG_DEVICE_TYPE	1: Output logs through UART.	\app log\app log.h
	2: Output logs through J-Link RTT.	(app_10g (app_10g.11
	3: Output logs through ARM ITM.	
	It enables/disables the coloring	SDK_Folder\components\libraries
APP_LOG_COLOR_ENABLE	format of log output texts.	\app_log\app_log.h
APP_LOG_TAG_ENABLE	It enables/disables adding tags in log	SDK_Folder\components\libraries
	output texts.	\app_log\app_log.h

Table 4-6 Configuration parameters of APP LOG module

🛄 Note:

- The Blocking Mode blocks task scheduling of BLE Stack. If applications output excessive debug logs at one time, the time sequence to execute BLE Stack tasks may be destroyed, resulting in such problems as delayed response or disconnection. However, this mode ensures that all debug logs are output completely.
- In Interrupt Mode, first, cache the debug logs to be output; then, use the UART interrupt sending mode to
 output cached debug logs in fragmented units through multiple output. When UART is outputting debug logs in
 interrupt sending mode, this program can run normally and does not interfere with the time sequence to execute
 BLE Stack tasks. However, if the size of debug logs to be output exceeds the cache size, some debug logs will get
 lost.

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.

The APP LOG module provides some APIs for developers to set the log level and the log format. If you use printf() only, you can set the log level and the log format to NULL.

The code of initializing the APP LOG module is provided in app_log_assert_init() in *user_periph_setup.c*. If you output logs by hardware UART before calling the function, you need to call void bsp_uart_init(void) to initialize the hardware UART. By default, void bsp_uart_init(void) outputs logs in Interrupt Mode. The API code is provided below:

```
static void app log assert 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);
    fault trace db init();
}
void bsp uart init(void)
#if APP DRIVER USE ENABLE
    app uart tx buf t uart buffer;
    app uart params t uart param;
    uart buffer.tx buf = s uart tx buffer;
    uart buffer.tx buf size = UART TX BUFF SIZE;
    uart param.id
                                        = APP UART ID;
    uart_param.init.baud_rate
uart_param.init.data_bits
uart_param.init.stop_bits
uart_param.init.parity
uart_param.init.parity
uart_param.init.parity
uart_param.init.parity
    uart_param.init.hw_flow_ctrl = UART_HWCONTROL_NONE;
```



```
uart param.init.rx timeout mode = UART RECEIVER TIMEOUT ENABLE;
    uart_param.pin_cfg.rx.type= APP_UART_RX_IO_TYPE;uart_param.pin_cfg.rx.pin= APP_UART_RX_PIN;uart_param.pin_cfg.rx.mux= APP_UART_RX_PINMUX;uart_param.pin_cfg.rx.pull= APP_UART_RX_PULL;uart_param.pin_cfg.tx.type= APP_UART_TX_IO_TYPE;uart_param.pin_cfg.tx.pin= APP_UART_TX_PIN;uart_param.pin_cfg.tx.pin= APP_UART_TX_PIN;uart_param.pin_cfg.tx.mux= APP_UART_TX_PIN;uart_param.pin_cfg.tx.mux= APP_UART_TX_PIN;uart_param.pin_cfg.tx.mux= APP_UART_TX_PIN;
                                           = APP UART TX_PINMUX;
     uart param.pin cfg.tx.mux
                                           = APP_UART_TX_PULL;
     uart param.pin cfg.tx.pull
                                            = APP UART TYPE INTERRUPT;
    uart_param.use_mode.type
     app uart init(&uart param, app uart evt handler, &uart buffer);
#else
     gpio init t gpio config = GPIO DEFAULT CONFIG;
     gpio config.mode = GPIO MODE MUX;
     gpio config.pin = SERIAL PORT TX PIN;
     gpio config.mux = SERIAL PORT TX PINMUX;
     hal gpio init (SERIAL PORT PORT, &gpio config);
     gpio config.pin = SERIAL PORT RX PIN;
     qpio config.mux = SERIAL PORT RX PINMUX;
     hal gpio init(SERIAL PORT PORT, &gpio config);
     s uart handle.p instance
                                                = SERIAL PORT GRP;
                                               = SERIAL PORT BAUDRATE;
     s uart handle.init.baud rate
                                               = UART DATABITS_8;
     s uart handle.init.data bits
                                                = UART STOPBITS 1;
     s uart handle.init.stop bits
                                                 = UART PARITY NONE;
     s uart handle.init.parity
                                              = UART HWCONTROL NONE;
     s uart handle.init.hw flow ctrl
     s uart handle.init.rx timeout mode = UART RECEIVER TIMEOUT ENABLE;
    hal_uart_deinit(&s_uart_handle);
    hal_uart_init(&s_uart_handle);
#endif
}
void bsp uart send(uint8 t *p data, uint16 t length)
#if APP DRIVER USE ENABLE
    app uart transmit async(APP UART ID, p data, length);
#else
    hal uart transmit(&s uart handle, p data, length, 5000);
#endif
}
void bsp uart flush (void)
#if APP DRIVER USE ENABLE
     app uart flush (APP UART ID);
#endif
}
```

🛄 Note:

- bsp_uart_init implements hardware port initialization of app_uart and hal_uart APIs. app_uart_init initializes the UART hardware port in Interrupt Mode; hal_uart_init only implements UART hardware port initialization in bsp_uart_init. For details about calling APIs in Interrupt Mode or Blocking Mode, see "Section 2.23 HAL UART Generic Driver" in *GR551x HAL and LL Drivers User Manual*. Users can overwrite this API.
- bsp_uart_send implements two output APIs: app_uart asynchronous (app_uart_transmit_async API) and hal_uart synchronous (hal_uart_transmit API). Users can choose an appropriate log output method on demand. Users can overwrite this API.
- bsp_uart_flush is a UART flush API to output the unsent data cached in the memory in Interrupt Mode. Users can overwrite this API.

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_WARINING()
- 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.

4.6.5 Debug with GRToolbox

The GR551x SDK provides an Android App, GRToolbox, to debug GR551x Bluetooth LE applications, which is in SDK_folder\tools\android\GRToolbox.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 Glossary and Abbreviations

Table 5-1 Glossary and abbreviations

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
НСІ	Host-Controller Interface
ют	Internet of Things
L2CAP	Logical Link Control and Adaption Protocol
LL	Link Layer
NVDS	Non-volatile Data Storage
ΟΤΑ	Over The Air
PMU	Power Management Unit
РНҮ	Physical Layer
RF	Radio Frequency
SCA	System Configuration Area
SDK	Software Development Kit
SM	Security Manager
SoC	System on Chip
XIP	Execute in Place