Application Note

Document No.: AN1130

G32R501 Keil Linker Script Application Note

Version: V1.1

1 Introduction

In the MDK-ARM development environment, the linker script files (Linker Script) are used to instruct the linker on how to map the code and data into the memory of the target microcontroller. They are crucial in the development of G32R5xx series MCU, as they define the layout of programs and data in different memory areas, ensuring that the code can run correctly, and the memory resources can be effectively used.

This document explains and describes the usage of commonly used linker script files for the G32R5xx series MCU development, for user reference.

Note: The above files are located in the following directory of G32R5xx_SDK:
G32R5xx_SDK_VERSION#/device_support/DEVICE_GPN/common/sct

Contents

1 Introduction 1

2 Memory Allocation of G32R501 3

2.1 Memory Allocation of G32R501 Single-core MCU 3

2.2 Memory Allocation of G32R501 Dual-core MCU 3

3 Memory Allocation of Each Linker Script File (Linker Script) 5

4 Use Linker Script File (.sct) in MDK 6

4.1 Select Linker Script Files (.sct) 6

4.2 Custom Configuration 6

5 Create Linker Script Files (.sct) 10

5.1 Syntax Rules of Scatter Files 10

5.2 Application Scenario Modification Examples 13

6 Revision 19

2 Memory Allocation of G32R501

Before using the linker script files, the corresponding memory allocation of each chip needs to be understood. Some products of the G32R501 series MCU are configured with dual cores, while others are configured with single core. The memory allocations under different configurations are not the same. But they all have the following memory areas:

- ITCM (Instruction Tightly Coupled Memory): Used to store instructions and provide high-speed instruction execution. In single-core mode, CPU0 has a larger ITCM (64KB), while in dual-core mode, the size of ITCM is 48KB (CPU0) and 8KB (CPU1).
- DTCM (Data Tightly Coupled Memory): Used to store data and provide high-speed data access. In single-core mode, CPU0 has 16KB DTCM, while in dual-core mode, the size of DTCM is 16KB (CPU0) and 8KB (CPU1).
- Flash: Non-volatile memory, used to store program code and static data. Either in single-core or dual-core mode, the size of Flash is 640KB.
- SRAM: Static random access memory, which can be used for general data storage. In both single-core and dual-core modes, the base address and size of SRAM remain unchanged, but in dual-core mode, it is necessary to ensure that the areas used by different cores do not overlap.

The prerequisite for selecting appropriate linker script files for development is that users need to understand the memory allocation of the target chip.

2.1 Memory Allocation of G32R501 Single-core MCU

In single-core mode, CPU0 uses the following memory allocation:

Table 1 Memory Allocation of G32R501 Single-core MCU

Memory area	Base address	Size
ITCM RAM	0x00000000	64KB
DTCM RAM	0x20000000	16KB
Flash	0x08000000 (ITCM: 0x00100000)	640KB
SRAM1	0x20100000	8KB
SRAM2	0x20200000	8KB
SRAM3	0x20300000	32KB

2.2 Memory Allocation of G32R501 Dual-core MCU

In dual-core mode, CPU0 and CPU1 share some memory areas, but they also have their own dedicated memory areas:

Table 2 Memory Allocation of G32R501 Dual-core MCU

Memory area	Base address	Size	Remarks
ITCM RAM (CPU0)	0x00000000	48KB	ITCM of CPU0
ITCM RAM (CPU1)	0x00000000	8KB	ITCM of CPU1
DTCM RAM (CPU0)	0x20000000	16KB	DTCM of CPU0
DTCM RAM (CPU1)	0x20000000	8KB	DTCM of CPU1
Flash	0x08000000 (ITCM: 0x00100000)	640KB	Shared Flash
SRAM1	0x20100000	8KB	Shared SRAM1
SRAM2	0x20200000	8KB	Shared SRAM2
SRAM3	0x20300000	32KB	Shared SRAM3

3 Memory Allocation of Each Linker Script File (Linker Script)

As described in Chapter 2, the memory structures of G32R501 series MCU are different, and the memory allocations in single-core and dual-core modes are also different. Therefore, it is necessary to create different linker script files based on specific scenarios and requirements to optimize system performance and memory utilization.

This chapter provides a brief description of memory allocation and purpose of linker script files.

The table below explains some selected important .sct files. For other .sct files that are not exemplified in the SDK, refer to the explanations in the table below.

Table 3 List of .sct Files in SDK (excerpt)

No.	File name	Description
1	g32r501dxy_cpu0_cbus_flash.sct	Code running and download location: 0x08000000, using CBUS interface
2	g32r501dxy_cpu0_itcm_flash.sct	Code running and download location: 0x00100000, using ITCM interface
3	g32r501dxy_cpu1_cbus_flash.sct	Code running and download location: 0x08040000, using CBUS interface
4	g32r501dxy_cpu1_itcm_flash.sct	Code running and download location: 0x00140000, using ITCM interface
5	g32r501dxy_cpu0_cbus_flash_secure.sct	Secure boot .sct example file for dual-core CPU0
6	g32r501dxy_cpu1_cbus_flash_secure.sct	Secure boot .sct example file for dual-core CPU1
7	g32r501dxy_cpu0_itcm_ram.sct	Code running and download location: 0x00000000, using ITCM interface
8	g32r501xc_cbus_flash.sct	Code running and download location: 0x08000000, using CBUS interface
9	g32r501xc_itcm_flash.sct	Code running and download location: 0x08000000, using CBUS interface
10	g32r501xy_cbus_flash.sct	Code running and download location: 0x08000000, using CBUS interface
11	g32r501xy_cbus_flash_secure.sct	Single-core Secure boot .sct example file
12	g32r501xy_itcm_flash.sct	Code running and download location: 0x00100000, using ITCM interface
13	g32r501xy_itcm_ram.sct	Code running and download location: 0x00000000, using ITCM interface

4 Use Linker Script File (.sct) in MDK

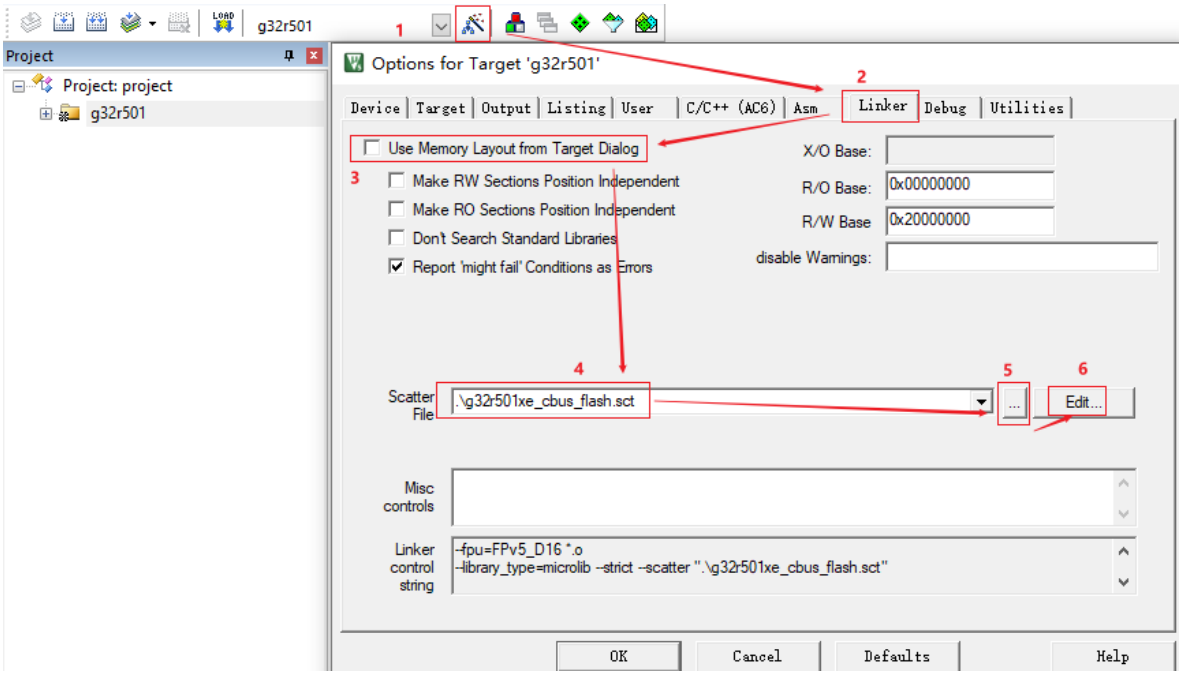
In the MDK development environment, using linker script files (.sct) can effectively manage and configure the memory layout of MCU. The steps and detailed description on how to select and customize linker script files in MDK are provided below.

4.1 Select Linker Script Files (.sct)

After copying the "*.sct" file to the MDK project, use the file in the project according to the steps below.

1. Open MDK project: Enter the MDK project.
2. Enter the project settings window: Click the "Project" in the menu bar or directly select "Options for Target" from the toolbar.
3. Select the "Linker" tab: In the opened project settings window, switch to the "Linker" tab.
4. Select the "..." option: In the file selection box that pops up in " …", click the browse button to select the corresponding "*.sct" file.
5. Select the "Edit" option: The selected "*.sct" files can be edited.
6. Save configuration: Click the "OK" button.

Figure 1 How to Select or Edit "*.sct" Files



4.2 Custom Configuration

After the "*.sct" file is selected, the following custom configuration can be set in the Configuration Wizard window to meet specific requirements.

Figure 2 Select Configuration Wizard Editing Window

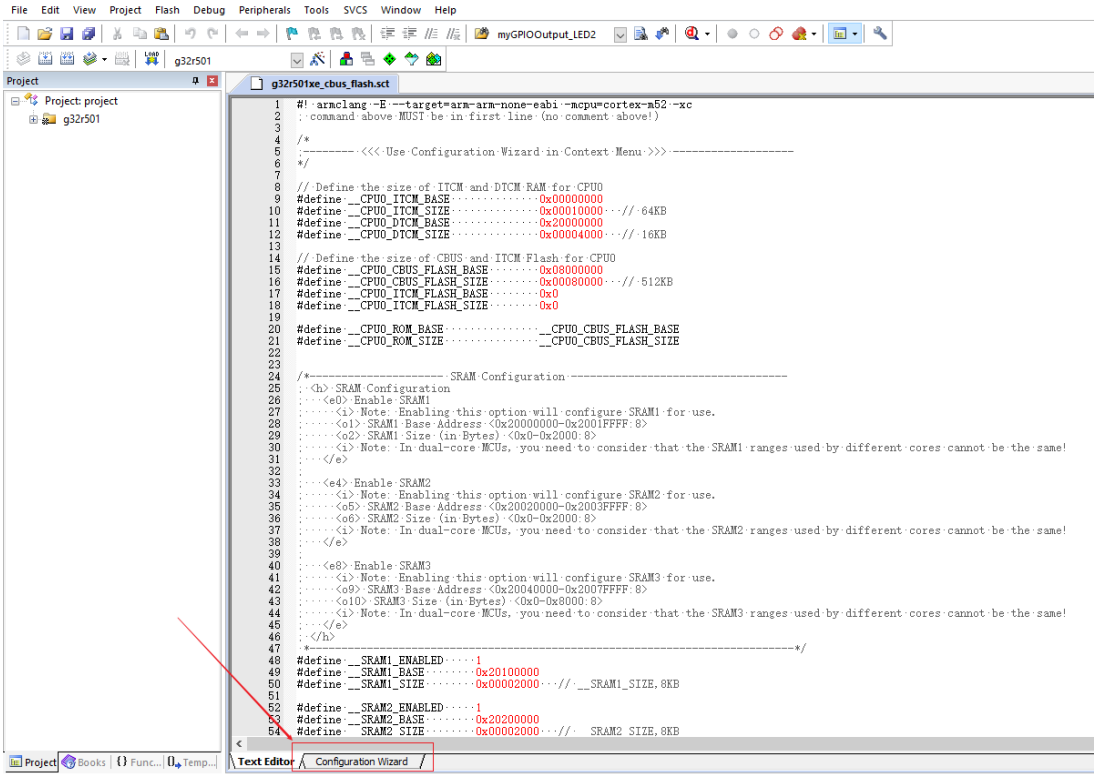
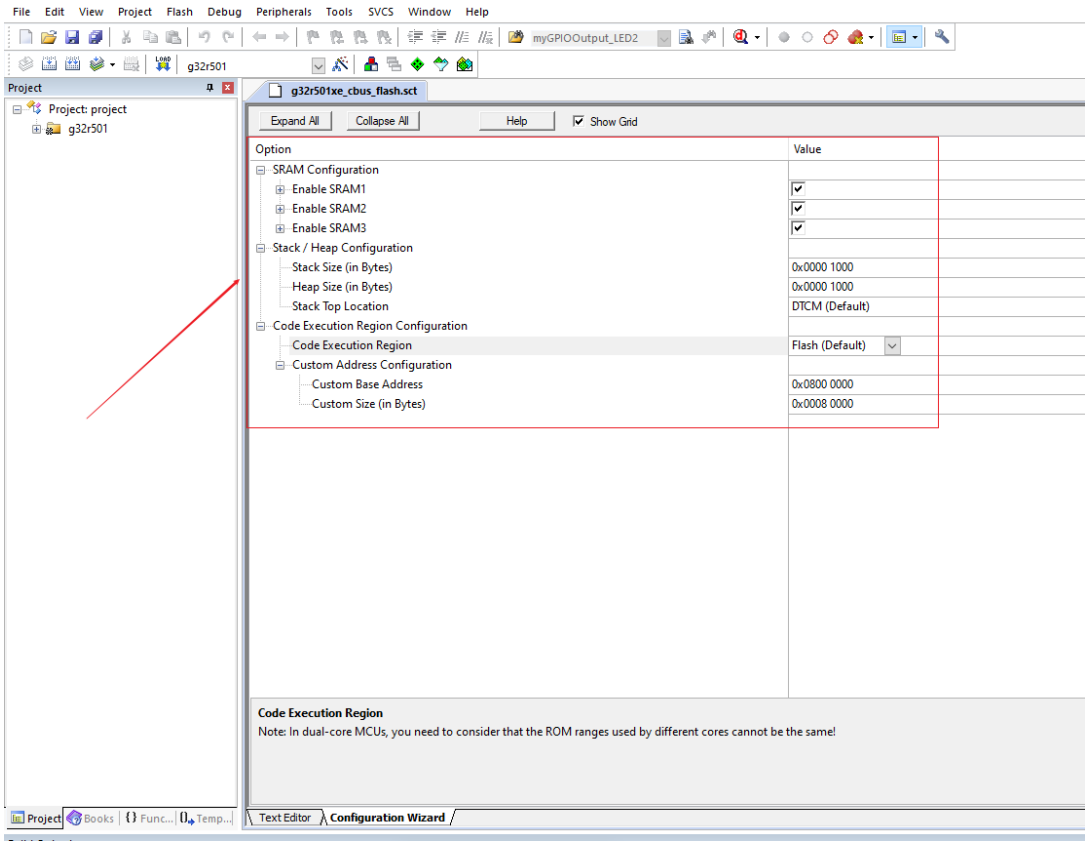


Figure 3 Configuration Wizard Editing Window



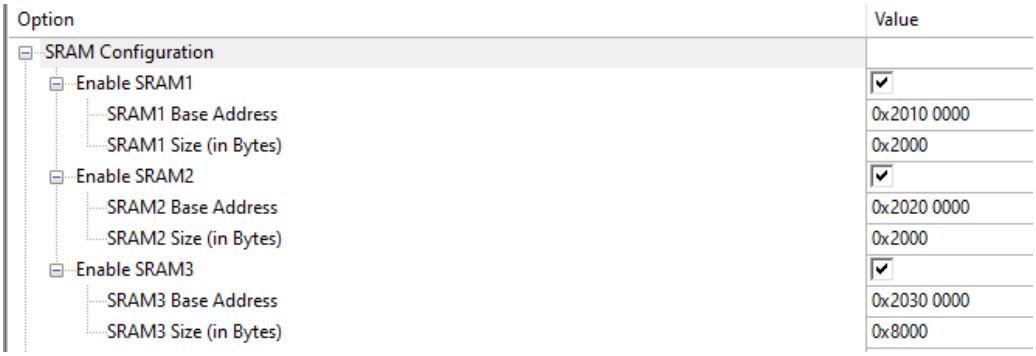
4.2.1 SRAM configuration

Different SRAM areas can be enabled or disabled in the configuration.

- SRAM1: Configure the base address and size, with a typical size of 8KB.
- SRAM2: Configure the base address and size, with a typical size of 8KB.
- SRAM3: Configure the base address and size, with a typical size of 32KB.

Note: To avoid memory conflicts, users must ensure that the SRAM regions used by different cores do not overlap in dual-core mode.

Figure 4 Enable or Disable Different SRAM Areas



4.2.2 Stack/Heap Configuration

The size and location of the stack and heap (DTCM, SRAM1, SRAM2, or SRAM3) can be set in this configuration.

- Default stack location: By default, the stack is stored in the DTCM of the chip.

Figure 5 Setting the Size and Location of the Stack and Heap

Note: The data accessed by DMA should not be placed in DTCM, but should be placed in a separate area such as dma_data segment, and be stored in other locations (SRAM1, SRAM2, SRAM3).

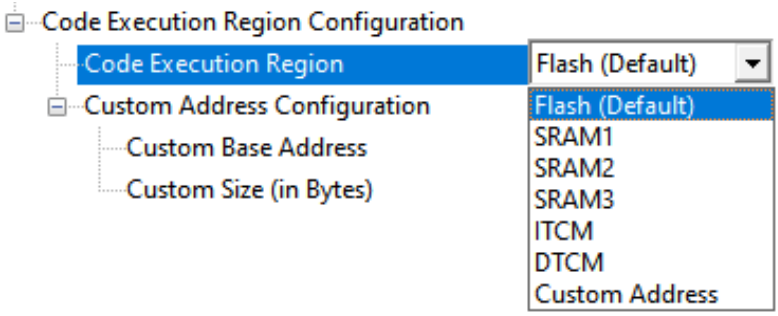
4.2.3 Code execution area configuration

The execution area (Flash, SRAM, ITCM, DTCM, or custom address) of the code can be selected in this configuration. If choosing to customize address, users must verify the validity of the configured region content.

Code execution area options:

- Flash: The default code storage and execution area.
- SRAM: SRAM1, SRAM2, or SRAM3 can be selected as the code execution area to improve execution speed.
- ITCM: Used to store critical code to provide faster execution speed.
- DTCM: Used for storing data to provide fast data access.
- Custom address: Users can manually set the code execution area.

Figure 6 Selecting Execution Area of the Code



Note: After modification, click “Save” to save the configuration. Changes will take effect upon the next compilation.

5 Create Linker Script Files (.sct)

Users can create linker script files (.sct) suitable for projects based on the MDK-ARM linker script file (.sct) format. For more information, see the format requirements for the MDK-ARM linker script files (.sct) and the content about memory allocation of G32R501 in Chapter 2.

5.1 Syntax Rules of Scatter Files

A scatter file contains one or more load regions. Each load region can contain one or more execution regions. An example is as follows:

```
;Load Region1
LOAD_ROM_1 0x0000
{
    ;Execution Region1
    EXEC_ROM_1 0x0000
    {
        ;Input Section1
        program1.o (+RO)
    }
    ;Execution Region2
    DRAM 0x18000 0x8000
    {
        ;Input Section2
        program1.o (+RW,+ZI)
    }
}
```

- Load Region:

  The standard form of a load region is:

```
// Items in brackets are optional
LoadRegionName (BaseAddress | ("+" AddressOffset(+<offset>))) [AttributeList] [MaxSize]
```

```
{
    ExecutionRegion
}
```

- LoadRegionName: Used by the linker to identify different load regions.
- BaseAddress: Specifies the start address of codes and data in the load region. Address offset is optional and must meet alignment requirements. <offset> represents the address located <offset> bytes after the end of the previous load region. If this is the first load region, the start address does not depend on any previous load region. <offset> indicates that the base address is located <offset> bytes from zero.
- Attributes: Specifies the properties of the load region. Common attributes include:

  ABSOLUTE---------- Absolute address (default)

  ALIGN <alignment>---------- Address alignment

  NOCOMPRESS---------- Specifies that the content of this load region is not compressed in the image file
- MaxSize: Specifies the maximum size of the load region. If the allocated size exceeds max_size, armlink will generate an error.

- ExecutionRegion:

  The standard form is:

```
// Items in brackets are optional
ExecutionRegionName (BaseAddress | ("+"AddressOffset (+<offset>))) [Attributes] [MaxSize | <length>]
{
    InputSections
}
```

- ExecutionRegionName: Used by the linker to identify different execution regions.
- BaseAddress: Similar to load region.
- Attributes: Specify the attributes of the load region. Common attributes include:

  ABSOLUTE----------Absolute address (default)

  ALIGN <alignment>----------Address alignment

  NOCOMPRESS----------Specify that the content of a load region is not

compressed in the image file.

EMPTY [-]<length>---------- Reserves an empty memory block of given size in the execution region, typically used by stack. Sections selectors cannot be placed in regions with EMPTY attribute. <length> indicates downward growing stack size. If length is negative, <base_address> is considered as the end address of the region.

- MaxSize: Similar to load region.

- InputSections:

The standard form is:

Input Section Description Common forms: Module Selector (+Input Section Attributes) Module Selector (Input Symbol Style) Module Selector (Input Section Style)

- Module Selector:
  - Wildcard character "*" and .ANY can be used, where .ANY selects all .o and .lib files but with lowest priority.
  - Using ".o" selects all .o files.
  - Using ".lib" selects all .lib files.
- Input Section Attributes: Can add input section attributes after the module selector. Each attribute descriptor is added with "+", separated by space or comma, for example, .ANY(+RO +ZI).

Table 4 Attribute Descriptions

Attribute Descriptor	Description
RO-CODE/CODE	Read-only code segment
RO-DATA/CONST	Read-only data segment
RO/TEXT	RO-CODE+ RO-DATA
RW-CODE	Readable and writable code segment
RW-DATA	Readable and writable code segment
RW/DATA	RW-CODE+ RW-DATA
XO	Executable-only region
ZI/BSS	Zero-initialized readable and writable data segment
ENTRY	Section entry

Additionally, the following pseudo attributes are recognized:

- FIRST.
- LAST.

- Input Symbol Style: Select the input section by symbols. Symbol style needs be modified using the ":gdef:" prefix.
- Input Section Style: The input section style can be used to select and control specific sections.
  - For example, InRoot$$Sections: Used to define symbols for specific sections to place some sections into specific memory regions.
  - RESET: Usually defines the program reset vector where the processor jumps to execute the startup code upon power-on or reset.

5.2 Application Scenario Modification Examples

5.2.1 Using Custom Linker Macros

In the file g32r501.h, macros are defined to control the memory layout of SDK code, as shown in the following figure:

Figure 7 Custom Linker Macros

```
// Define macros to place functions or variables in specific memory sections
// Usage:
// - Use these macros before function declarations or variable definitions to place them in specific sections.
// - Ensure that the linker script (MDK) or ICF file (IAR), or LD script (GCC) includes the corresponding sections.

#if defined (__CC_ARM) || defined (__ICCARM__) || defined (__GNUC__) || defined (__ARMCC_VERSION)
  #define SECTION_ITCM_INSTRUCTION  __attribute__((section("itcm.instruction")))
  #define SECTION_ITCM_RAMFUNC      __attribute__((section("itcm.ramfunc")))
  #define SECTION_DTCM_DATA         __attribute__((section("dtcm.data")))
  #define SECTION_DTCM_BSS          __attribute__((section("dtcm.bss")))
  #define SECTION_SRAM1_SHARE_DATA  __attribute__((section("sram1.share_data")))
  #define SECTION_SRAM2_SHARE_DATA  __attribute__((section("sram2.share_data")))
  #define SECTION_SRAM3_SHARE_DATA  __attribute__((section("sram3.share_data")))
#else
  #error "Unsupported compiler: no section macros can be defined."
#endif
```

Table 5 Custom Linker Section Base Addresses

Specific section name	Link base address
itcm.instruction	ITCM(0x00000000)
itcm.ramfunc	ITCM(0x00000000)
dtcm.data	DTCM(0x20000000)
dtcm.bss	DTCM(0x20000000)
sram1.share_data	SRAM1(0x20000000)
sram2.share_data	SRAM2(0x20200000)
sram3.share_data	SRAM3(0x20300000)

Users can specify the locations of functions and variables in ITCM, DTCM, and SRAM by using macros in g32r501.h. If these macros do not meet requirements, users can refer to the following examples to write custom linker script files (.sct) and add custom linker sections.

5.2.2 Specifying Function and Variable Link Regions

Example 1: Place variable Data_Ex in DTCM

- Determine the memory region to store the function or variable:
  - Data_Ex-------0x20000000 – 0x2000C000(DTCM)
- Add a new input section in the specified memory region:
  - Add a new input section to the execution region belonging to DTCM, custom section name is data.ex

```
RW_RAM_CPU0_DTCM __CPU0_DTCM_BASE __CPU0_DTCM_SIZE {
        ;Newly added input section
        .ANY (data.ex)
        .ANY (+RW +ZI)
}
```

- For implementation, control by using`__attribute__((section("xxx")))`

```
__attribute__((section("data.ex ")))
uint32_t Data_Ex;
```

5.2.3 Specifying Function and Variable Link Addresses

Users may need to place certain functions or variables at specific addresses during application development. Keil generally provides two methods, with details listed below.

Example 2: Place variable Data_Ex at address 0x20300000

- Method 1: For implementation, control by using`__attribute__((__used__, section(".ARM.__at_0x20300000"))).`

```
__attribute__((__used__, section(".ARM.__at_0x20300000")))
const uint32_t Data_Ex;
```

Note: This method can only be applied to read-only variables qualified with the const keyword. If it is applied to readable and writable variables, a warning will be generated.

- Method 2: Modify the .sct file. Specify the storage address through the .sct file.

  - First modify the .sct file by defining a custom execution region, setting the start address to the specific address where the user needs to link the variable. Then, customize an input section name.

```
RW_RAM 0x20001000 0x400 {
        ;Newly added input section
        .ANY (data.ex)
}
```

  - For implementation, `__attribute__((section("xxx")))` can be used to control the linking of variables to the newly added execution region specified in the linker script (.sct) file.

```
__attribute__((section("data.ex ")))
uint32_t Data_Ex;
```

By following the methods listed above, the variables can be linked to the specified addresses.

5.2.4 Application Partitioning Regions

To achieve better memory management and performance optimization, partitioning the application into regions using scatter loading is an important strategy.

Example:

```
#define __RO_BASE    __CPU0_ROM_BASE
#define __RO_SIZE    __CPU0_ROM_SIZE

#define __CPU0_ROM_BASE              __CPU0_CBUS_FLASH_BASE
#define __CPU0_ROM_SIZE              __CPU0_CBUS_FLASH_SIZE

#define __CPU0_CBUS_FLASH_BASE       0x08000000
#define __CPU0_CBUS_FLASH_SIZE       0x00050000   // 320KB

#define __SRAM3_BASE                 0x20300000
#define __SRAM3_SIZE                 0x00008000   // __SRAM3_SIZE,32KB

#define __SRAM1_BASE                 0x20100000
#define __SRAM1_SIZE                 0x00002000   // __SRAM1_SIZE,8KB
```

```
#define __CPU0_DTCM_BASE              0x20000000
#define __CPU0_DTCM_SIZE              0x00004000   // 16KB

LR_ROM __RO_BASE __RO_SIZE {
  ER_ROM __RO_BASE __RO_SIZE {
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
   .ANY (+XO)
  }

  RW_RAM_CPU0_DTCM __CPU0_DTCM_BASE __CPU0_DTCM_SIZE {
    .ANY (dtcm.data)
    .ANY (dtcm.bss)
    .ANY (+RW +ZI)
  }

  RW_RAM_SRAM1 __SRAM1_BASE __SRAM1_SIZE {
    .ANY (sram1.share_data)
    .ANY (+RW +ZI)
  }
}

LR_ROM_SRAM3 __SRAM3_BASE __SRAM3_SIZE {
  ER_ROM_SRAM3 __SRAM3_BASE __SRAM3_SIZE {
    .ANY (+CONST)
    .ANY (share.data)
  }
}
```

This linker script structure clearly defines two load regions for storing constant data and read-only code respectively.

- The memory region LR_ROM_SRAM3 starts at __SRAM3_BASE with size __SRAM3_SIZE, usually used for specific read-only data.
- LR_ROM __RO_BASE __RO_SIZE defines a read-only memory region named LR_ROM, starting at __RO_BASE with size __RO_SIZE.

5.2.4.1 Dual-core Example Link Files

The dual-core example .sct files are located in the SDK\device_support\g32r501\common\sct

directory:

- g32r501dxy_cpu0_cbus_flash.sct
- g32r501dxy_cpu1_cbus_flash.sct

In dual-core example files, the Flash is split into two parts. If the user needs to modify the Flash size allocation for the two cores, simply modify the macro definitions in the example files.

Example: Modify CPU0's Flash size to 384KB and CPU1's Flash size to 256KB

```
// Define the size of CBUS and ITCM Flash for CPU0/CPU1
#define __CPU0_CBUS_FLASH_BASE          0x08000000
#define __CPU0_CBUS_FLASH_SIZE          0x00060000    // Modification, 384KB
#define __CPU1_CBUS_FLASH_BASE          0x08060000    // Modification
#define __CPU1_CBUS_FLASH_SIZE          0x00040000    // Modification, 256KB
#define __CPU0_ITCM_FLASH_BASE          0x0
#define __CPU0_ITCM_FLASH_SIZE          0x0
#define __CPU1_ITCM_FLASH_BASE          0x0
#define __CPU1_ITCM_FLASH_SIZE          0x0

#define __CPU0_ROM_BASE                 __CPU0_CBUS_FLASH_BASE
#define __CPU0_ROM_SIZE                 __CPU0_CBUS_FLASH_SIZE
#define __CPU1_ROM_BASE                 __CPU1_CBUS_FLASH_BASE
#define __CPU1_ROM_SIZE                 __CPU1_CBUS_FLASH_SIZE
```

5.2.4.2 Secure Boot Example Link Files

For scenarios where data read by CPU from Flash must not be encrypted. When designing the secure boot example .sct file, read-only data needs to be separated. This helps users clearly know where the Flash region stores the read-only data and ensures that this region is not set with encryption permissions.

The dual-core example .sct file is located at SDK\device_support\g32r501\common\sct\g32r501dxy_cpu0_cbus_flash_secure.sct. Users can refer to this file for specific design on how to separate CPU read data in Flash.

In this file, the non-encryptable region by default starts at address 0x08050000. If the user

needs to modify this region, just change the start address macro definition in the g32r501dxy_cpu0_cbus_flash_secure.sct file.

For example, modify the start address and size of the region where encryption is not allowed. The .sct file provided by the SDK has a default start address of 0x08050000 and a size of 64KB, which are changed to 0x08020000 and 128KB respectively.

```
// Define the size of ITCM and DTCM RAM for CPU0
#define __CPU0_ITCM_BASE                    0x00000000
#define __CPU0_ITCM_SIZE                    0x0000C000     // 48KB
#define __CPU0_DTCM_BASE                    0x20000000
#define __CPU0_DTCM_SIZE                    0x00004000     // 16KB

// Define the size of CBUS and ITCM secure Flash for CPU0,
// This area allows users to be set with encryption permissions.
#define __CPU0_CBUS_FLASH_SECURE_BASE       0x08000000
#define __CPU0_CBUS_FLASH_SECURE_SIZE       0x00020000     // 128KB
#define __CPU0_ITCM_FLASH_SECURE_BASE       0x0
#define __CPU0_ITCM_FLASH_SECURE_SIZE       0x0

// Define the size of CBUS and ITCM unsecure Flash for CPU0,
// This area does not allow users to be set with encryption permissions.
#define __CPU0_CBUS_FLASH_UNSECURE_BASE     0x08020000     // Non-encryptable region modified to 0x08020000
#define __CPU0_CBUS_FLASH_UNSECURE_SIZE     0x00020000     // 128KB
#define __CPU0_ITCM_FLASH_UNSECURE_BASE     0x0
#define __CPU0_ITCM_FLASH_UNSECURE_SIZE     0x0

// Define the size of CBUS Flash for CPU1
#define __CPU1_CBUS_FLASH_BASE              0x08060000
#define __CPU1_CBUS_FLASH_SIZE              0x00040000     // 256KB
```

6 Revision

Table 6 Document Revision History

Date	Version	Change History
January 2025	1.0	New
April 2025	1.1	Add descriptions of the syntax rules for the scatter file.

Statement

This document is formulated and published by Geehy Semiconductor Co., Ltd. (hereinafter referred to as “Geehy”). The contents in this document are protected by laws and regulations of trademark, copyright and software copyright. Geehy reserves the right to make corrections and modifications to this document at any time. Read this document carefully before using Geehy products. Once you use the Geehy product, it means that you (hereinafter referred to as the “users”) have known and accepted all the contents of this document. Users shall use the Geehy product in accordance with relevant laws and regulations and the requirements of this document.

1. Ownership

This document can only be used in connection with the corresponding chip products or software products provided by Geehy. Without the prior permission of Geehy, no unit or individual may copy, transcribe, modify, edit or disseminate all or part of the contents of this document for any reason or in any form.

The “极海” or “Geehy” words or graphics with “®” or “™” in this document are trademarks of Geehy. Other product or service names displayed on Geehy products are the property of their respective owners.

2. No Intellectual Property License

Geehy owns all rights, ownership and intellectual property rights involved in this document.

Geehy shall not be deemed to grant the license or right of any intellectual property to users explicitly or implicitly due to the sale or distribution of Geehy products or this document.

If any third party’s products, services or intellectual property are involved in this document, it shall not be deemed that Geehy authorizes users to use the aforesaid third party’s products, services or intellectual property. Any information regarding the application of the product, Geehy hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party, unless otherwise agreed in sales order or sales contract.

3. Version Update

Users can obtain the latest document of the corresponding models when ordering Geehy products.

If the contents in this document are inconsistent with Geehy products, the agreement in the sales order or the sales contract shall prevail.

4. Information Reliability

The relevant data in this document are obtained from batch test by Geehy Laboratory or cooperative third-party testing organization. However, clerical errors in correction or errors caused by differences in testing environment may occur inevitably. Therefore, users should understand that Geehy does not bear any responsibility for such errors that may occur in this document. The relevant data in this document are only used to guide users as performance parameter reference and do not constitute Geehy's guarantee for any product performance.

Users shall select appropriate Geehy products according to their own needs, and effectively verify and test the applicability of Geehy products to confirm that Geehy products meet their own needs, corresponding standards, safety or other reliability requirements. If losses are caused to users due to user's failure to fully verify and test Geehy products, Geehy will not bear any responsibility.

5. Legality

USERS SHALL ABIDE BY ALL APPLICABLE LOCAL LAWS AND REGULATIONS WHEN USING THIS DOCUMENT AND THE MATCHING GEEHY PRODUCTS. USERS SHALL UNDERSTAND THAT THE PRODUCTS MAY BE RESTRICTED BY THE EXPORT, RE-EXPORT OR OTHER LAWS OF THE COUNTRIES OF THE PRODUCTS SUPPLIERS, GEEHY, GEEHY DISTRIBUTORS AND USERS. USERS (ON BEHALF OR ITSELF, SUBSIDIARIES AND AFFILIATED ENTERPRISES) SHALL AGREE AND PROMISE TO ABIDE BY ALL APPLICABLE LAWS AND REGULATIONS ON THE EXPORT AND RE-EXPORT OF GEEHY PRODUCTS AND/OR TECHNOLOGIES AND DIRECT PRODUCTS.

6. Disclaimer of Warranty

THIS DOCUMENT IS PROVIDED BY GEEHY "AS IS" AND THERE IS NO WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, TO THE EXTENT PERMITTED BY APPLICABLE LAW.

GEEHY'S PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED FOR USE AS CRITICAL COMPONENTS IN MILITARY, LIFE-SUPPORT, POLLUTION CONTROL, OR HAZARDOUS SUBSTANCES MANAGEMENT SYSTEMS, NOR WHERE FAILURE COULD RESULT IN INJURY, DEATH, PROPERTY OR ENVIRONMENTAL DAMAGE.

IF THE PRODUCT IS NOT LABELED AS "AUTOMOTIVE GRADE," IT SHOULD NOT BE CONSIDERED SUITABLE FOR AUTOMOTIVE APPLICATIONS. GEEHY ASSUMES NO LIABILITY FOR THE USE BEYOND ITS SPECIFICATIONS OR GUIDELINES.

THE USER SHOULD ENSURE THAT THE APPLICATION OF THE PRODUCTS COMPLIES WITH ALL RELEVANT STANDARDS, INCLUDING BUT NOT LIMITED TO SAFETY, INFORMATION SECURITY, AND ENVIRONMENTAL REQUIREMENTS. THE USER ASSUMES FULL RESPONSIBILITY FOR THE SELECTION AND USE OF GEEHY PRODUCTS. GEEHY WILL BEAR NO RESPONSIBILITY FOR ANY DISPUTES ARISING FROM THE SUBSEQUENT DESIGN OR USE BY USERS.

7. Limitation of Liability

IN NO EVENT, UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL GEEHY OR ANY OTHER PARTY WHO PROVIDES THE DOCUMENT AND PRODUCTS "AS IS", BE LIABLE FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, DIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE DOCUMENT AND PRODUCTS (INCLUDING BUT NOT LIMITED TO LOSSES OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY USERS OR THIRD PARTIES). THIS COVERS POTENTIAL DAMAGES TO PERSONAL SAFETY, PROPERTY, OR THE ENVIRONMENT, FOR WHICH GEEHY WILL NOT BE RESPONSIBLE.

8. Scope of Application

The information in this document replaces the information provided in all previous versions of the document.