GigaDevice Semiconductor Inc.

Scatter Loading Instructions of GD32 Embedded Builder for Arm Cortex-M Processor

Application Note
AN311

Revision 1.0

(Dec. 2025)

Table of Contents

List of Figures

List of Tables

1. Introduction

`ld` is one of the Linkers in the GNU binutils toolset. It links various object files and library files, redirects their data, and completes symbol resolution. Linking primarily involves four tasks: storage allocation, symbol management, libraries, and relocation. Therefore, based on the requirements of project code, we can modify this file to specify the storage location of the code sections.

This application note is based on the GD32H75x series, using the GD32H759I-EVAL development board with Embedded Builder version 1.5.18. The main implemented functions are as follows:

- Implement user-defined global variables and arrays loaded to specified locations.
- Implement user-defined functions loaded to specified locations.
- Implement user-specified .c files loaded to specified locations.
- Load to the specified position of SDRAM to achieve the above function.

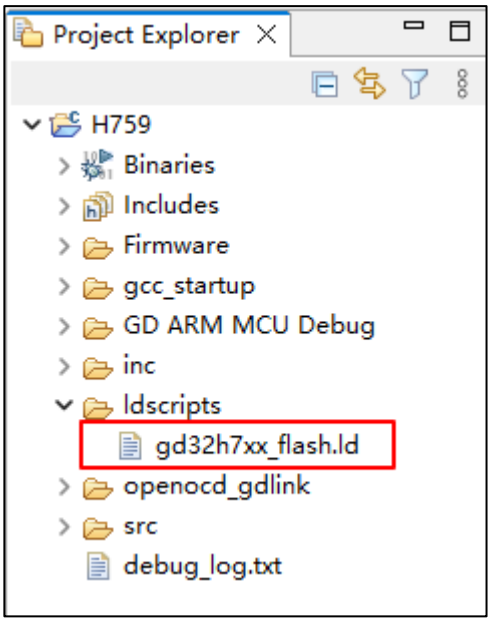
This manual is also applicable to GD32 Arm Cortex® M3/M4/M23/M33/M7 core MCU products.

2. Implementation of scatter loading in Embedded Builder

2.1. Use manually written ld file

This project directly uses the manually written ld file. In Embedded Builder, double-click the gd32h7xx_flash.ld file under "Window->Show View->Project Explorer->ldscripts". The project directory structure is shown in ***Figure 2-1. Project directory structure in GD32 Embedded Builder***, as illustrated in the project directory structure of Embedded Builder. Alternatively, user can open and edit it in the directory "..\ldscripts\gd32h7xx_flash.ld".

Figure 2-1. Project directory structure in GD32 Embedded Builder



2.2. VMA and LMA

In the linker script, VMA and LMA are two core concepts used to control the distribution of program sections in memory. VMA (Virtual Memory Address) is the memory address where a section resides during program execution, also known as the runtime address. LMA (Load Memory Address) is the physical address where a section is stored during programming, also known as the load address. To implement scatter loading functionality, it is essential to understand the core concepts of VMA and LMA. For a detailed introduction to VMA and LMA, refer to "***AN200 GD32 Embedded Builder Linker Script File Introduction***".

2.3. Load global variables and global arrays to the specified location

2.3.1. Custom section

Add user-defined sections in the linker script to allocate different types of data to specified addresses, as shown in ***Table 2-1. Overview of address allocation for loading global variables and global arrays to specified locations***. The linker file code is as shown in ***Table 2-2. Allocation of global variables and global arrays to specified locations in gd32h7xx_flash.ld***.

Table 2-1. Overview of address allocation for loading global variables and global arrays to specified locations

Segment type	Section name	FLASH address	RAM address	Description
Initialize variables	.data.user_vars	0x08008000	0x24008000	Variable with an initial value
Initialize array	.data.user_arrays	0x08009000	0x24009000	Array with an initial value
Uninitialized variables	.bss.user_vars	-	0x2400A000	Zero-initialized variables
Uninitialized array	.bss.user_arrays	-	0x2400B000	Zero-initialized array
Constant array	.rodata.user_const_arrays	0x08007000	-	Const array

Table 2-2. Allocation of global variables and global arrays to specified locations in gd32h7xx_flash.ld

```
/* user initializes variable section */
.user_global_vars ALIGN(0x24008000, 4) : AT(ALIGN(0x08008000, 4))
{
    . = ALIGN(4);
    __user_global_vars_start__ = .;
    KEEP(*(.data.user_vars))
    . = ALIGN(4);
    __user_global_vars_end__ = .;
    __user_global_vars_size__ = __user_global_vars_end__ - __user_global_vars_start__;
}
__user_global_vars_loadaddr__ = LOADADDR(.user_global_vars);
/* user initializes array section */
.user_global_arrays ALIGN(0x24009000, 4) : AT(ALIGN(0x08009000, 4))
{
```

```
        . = ALIGN(4);
        __user_global_arrays_start__ = .;
        KEEP(*(.data.user_arrays))
        . = ALIGN(4);
        __user_global_arrays_end__ = .;
        __user_global_arrays_size__ = __user_global_arrays_end__ - __user_global_arrays_start__;
    }
    __user_global_arrays_loadaddr__ = LOADADDR(.user_global_arrays);
    /* user uninitialized variable section */
    .user_uninit_vars ALIGN(0x2400A000, 4) (NOLOAD) :
    {
        . = ALIGN(4);
        __user_uninit_vars_start__ = .;
        KEEP(*(.bss.user_vars))
        . = ALIGN(4);
        __user_uninit_vars_end__ = .;
        __user_uninit_vars_size__ = __user_uninit_vars_end__ - __user_uninit_vars_start__;
    }
    /* user uninitialized array section */
    .user_uninit_arrays ALIGN(0x2400B000, 4) (NOLOAD) :
    {
        . = ALIGN(4);
        __user_uninit_arrays_start__ = .;
        KEEP(*(.bss.user_arrays))
        . = ALIGN(4);
        __user_uninit_arrays_end__ = .;
        __user_uninit_arrays_size__ = __user_uninit_arrays_end__ - __user_uninit_arrays_start__;
    }
    /* user constant array section */
    .user_const_arrays ALIGN(0x08007000, 4) :
    {
        . = ALIGN(4);
        __user_const_arrays_start__ = .;
        KEEP(*(.rodata.user_const_arrays))
        . = ALIGN(4);
        __user_const_arrays_end__ = .;
        __user_const_arrays_size__ = __user_const_arrays_end__ - __user_const_arrays_start__;
    } > FLASH
```

2.3.2. Variable definitions in C code

Define section attribute macros in the header file to improve code readability, as shown in

Table 2-3. Header file macro definitions for loading global variables and global arrays to specified locations.

Table 2-3. Header file macro definitions for loading global variables and global arrays to specified locations

```
/* initialized data section attributes - will be loaded from FLASH to RAM */
#define USER_INIT_VAR           __attribute__((section(".data.user_vars")))
#define USER_INIT_ARRAY         __attribute__((section(".data.user_arrays")))
/* uninitialized data section attributes - only in RAM, cleared at startup */
#define USER_UNINIT_VAR         __attribute__((section(".bss.user_vars")))
#define USER_UNINIT_ARRAY       __attribute__((section(".bss.user_arrays")))

#define USER_CONST_ARRAY        __attribute__((section(".rodata.user_const_arrays")))
```

Variable definitions are illustrated in *Table 2-4. Global variable and global array loading to specified location variable definitions*.

Table 2-4. Global variable and global array loading to specified location variable definitions

```
/* user initialized global variables definition - will be placed in .data.user_vars section */
USER_INIT_VAR int32_t     system_status = 1;
USER_INIT_VAR uint32_t    device_serial = 0x12345678;
USER_INIT_VAR uint16_t    firmware_version = 0x0102;
USER_INIT_VAR uint8_t     system_config = 0xAB;

/* user initialized global arrays definition - will be placed in .data.user_arrays section */
USER_INIT_ARRAY uint16_t calibration_data[10] = {
100, 200, 300, 400, 500, 600, 700, 800, 900, 1000
};

USER_INIT_ARRAY uint8_t device_info[32] = {
'G','D','3','2','H','7','x','x',' ','D','e','v','i','c','e',0
};

/* User uninitialized global variables definition - will be placed in .bss.user_vars section */
USER_UNINIT_VAR volatile uint32_t     tick_counter;
USER_UNINIT_VAR volatile uint16_t     error_code;
USER_UNINIT_VAR volatile uint8_t      system_state;
USER_UNINIT_VAR uint32_t                  runtime_config;

/* User uninitialized global arrays definition - will be placed in .bss.user_arrays section */
USER_UNINIT_ARRAY uint8_t     uart_tx_buffer[512];
USER_UNINIT_ARRAY uint8_t     uart_rx_buffer[512];
USER_UNINIT_ARRAY uint16_t    adc_samples[256];
```

```
USER_UNINIT_ARRAY char          log_buffer[1024];

/* Const array definition */
USER_CONST_ARRAY const uint16_t lookup_table[256] = {
0x0000, 0x0001, 0x0004, 0x0009,
};
```

2.3.3. Custom initialization implementation

Call the custom section initialization function at system startup, as shown in ***Table 2-5. Global variable and global array loading to specified location data section initialization implementation***.

Table 2-5. Global variable and global array loading to specified location data section initialization implementation

```
/* Data initialization function implementation */
void UserData_Init(void)
{
    uint32_t *src, *dst;
    uint32_t size;
    /* 1. Initialize user global variables segment - copy from FLASH to RAM */
    src = &__user_global_vars_loadaddr__;
    dst = &__user_global_vars_start__;
    size = ((uint32_t)&__user_global_vars_end__ - (uint32_t)&__user_global_vars_start__) / 4;

    for(uint32_t i = 0; i < size; i++) {
        *dst++ = *src++;
    }

    /* 2. Initialize user global arrays segment - copy from FLASH to RAM */
    src = &__user_global_arrays_loadaddr__;
    dst = &__user_global_arrays_start__;
    size = ((uint32_t)&__user_global_arrays_end__ - (uint32_t)&__user_global_arrays_start__) / 4;

    for(uint32_t i = 0; i < size; i++) {
        *dst++ = *src++;
    }
    /* 3. Clear user uninitialized variables segment */
    dst = &__user_uninit_vars_start__;
    size = ((uint32_t)&__user_uninit_vars_end__ - (uint32_t)&__user_uninit_vars_start__) / 4;

    for(uint32_t i = 0; i < size; i++) {
        *dst++ = 0;
```

```
    }
    /* 4. Clear user uninitialized arrays segment */
    dst = &__user_uninit_arrays_start__;
    size = ((uint32_t)&__user_uninit_arrays_end__ - (uint32_t)&__user_uninit_arrays_start__) / 4;

    for(uint32_t i = 0; i < size; i++) {
        *dst++ = 0;
    }
}
```

Note:

1. Initialization should be placed before the cache initialization function.
2. Constant arrays do not require initialization operations.

2.3.4. Test verification

Verify global variable and global array loading to specified location memory layout through UART printing. Code is shown in ***Table 2-6. Global variable and global array loading to specified location memory verification code***.

Table 2-6. Global variable and global array loading to specified location memory verification code

```
void UserData_PrintMemoryInfo(void)
{
    printf("=== User-defined Memory Segments Information ===\r\n");

    printf("1. Initialized variables segment:\r\n");
    printf("    RAM address: 0x%08X - 0x%08X (%d bytes)\r\n",
           (uint32_t)&__user_global_vars_start__,
           (uint32_t)&__user_global_vars_end__,
           (uint32_t)&__user_global_vars_end__ - (uint32_t)&__user_global_vars_start__);
    printf("    FLASH source: 0x%08X\r\n", (uint32_t)&__user_global_vars_loadaddr__);

    printf("2. Initialized arrays segment:\r\n");
    printf("    RAM address: 0x%08X - 0x%08X (%d bytes)\r\n",
           (uint32_t)&__user_global_arrays_start__,
           (uint32_t)&__user_global_arrays_end__,
           (uint32_t)&__user_global_arrays_end__ - (uint32_t)&__user_global_arrays_start__);
    printf("    FLASH source: 0x%08X\r\n", (uint32_t)&__user_global_arrays_loadaddr__);

    printf("3. Uninitialized variables segment:\r\n");
    printf("    RAM address: 0x%08X - 0x%08X (%d bytes)\r\n",
           (uint32_t)&__user_uninit_vars_start__,
```

```
                (uint32_t)&__user_uninit_vars_end__,
                (uint32_t)&__user_uninit_vars_end__ - (uint32_t)&__user_uninit_vars_start__);

    printf("4. Uninitialized arrays segment:\r\n");
    printf("     RAM address: 0x%08X - 0x%08X (%d bytes)\r\n",
                (uint32_t)&__user_uninit_arrays_start__,
                (uint32_t)&__user_uninit_arrays_end__,
                (uint32_t)&__user_uninit_arrays_end__ - (uint32_t)&__user_uninit_arrays_start__);
    printf("5. Constant arrays segment:\r\n");
    printf("     FLASH address: 0x%08X - 0x%08X (%d bytes)\r\n",
                (uint32_t)&__user_const_arrays_start__,
                (uint32_t)&__user_const_arrays_end__,
                (uint32_t)&__user_const_arrays_end__ - (uint32_t)&__user_const_arrays_start__);
    printf("============================================\r\n\r\n");

}

int main(void)
{
    gd_eval_com_init(EVAL_COM);
    /*system initialization */
    printf("GD32H7xx User-defined Data Segment Example\r\n");
    printf("==========================================\r\n");

    /* initialize user data segment */
    UserData_Init();
    /* enable the CPU cache */
    cache_enable();
    mpu_config();
    /* print memory information */
    UserData_PrintMemoryInfo();

    /* verify initialization data */
    printf("=== Verify Initialization Data ===\r\n");
    printf("system_status = %d (addr: 0x%08X)\r\n", system_status, (uint32_t)&system_status);
    printf("device_serial = 0x%08X (addr: 0x%08X)\r\n", device_serial, (uint32_t)&device_serial);
    printf("calibration_data[0] = %d (addr: 0x%08X)\r\n", calibration_data[0],
(uint32_t)&calibration_data[0]);
    printf("calibration_data[9] = %d (addr: 0x%08X)\r\n", calibration_data[9],
(uint32_t)&calibration_data[9]);
    printf("calibration_data array base addr: 0x%08X\r\n", (uint32_t)calibration_data);
    printf("device_info = %s (addr: 0x%08X)\r\n", (char*)device_info, (uint32_t)device_info);
```

```
    /* verify const array */
    printf("lookup_table[0] = 0x%04X (addr: 0x%08X)\r\n", lookup_table[0], (uint32_t)&lookup_table[0]);
    printf("lookup_table[255] = 0x%04X (addr: 0x%08X)\r\n", lookup_table[255], (uint32_t)&lookup_table[255]);
    printf("lookup_table array base addr: 0x%08X\r\n", (uint32_t)lookup_table);

    /* test uninitialized data usage */
    printf("\r\n=== Test Uninitialized Data Usage ===\r\n");

    /* buffer filling test */
    for(int i = 0; i < 10; i++) {
        uart_tx_buffer[i] = i + 0x30;    // '0' to '9'
        adc_samples[i] = i * 100;
    }
    uart_tx_buffer[10] = '\0';

    printf("uart_tx_buffer = %s (addr: 0x%08X)\r\n", (char*)uart_tx_buffer, (uint32_t)uart_tx_buffer);
    printf("adc_samples[5] = %d (addr: 0x%08X)\r\n", adc_samples[5], (uint32_t)&adc_samples[5]);
    printf("adc_samples array base addr: 0x%08X\r\n", (uint32_t)adc_samples);

    printf("\r\n=== System Ready ===\r\n");
    printf("User-defined data segments test completed successfully!\r\n");

    /* main loop - simple endless loop */
    while(1) {
    }
}
```

Test print logs are shown in ***Table 2-7. Global variable and global array loading to specified location logs***, with test results consistent with actual allocation.

Table 2-7. Global variable and global array loading to specified location logs

```
GD32H7xx User-defined Data Segment Example
==========================================
Starting initialization of user-defined data segments...
Initializing user global variables segment...
  Address range: 0x24008000 - 0x2400800C
  Data size: 12 bytes
Initializing user global arrays segment...
  Address range: 0x24009000 - 0x24009034
  Data size: 52 bytes
```

```
Clearing user uninitialized variables segment...
   Address range: 0x2400A000 - 0x2400A00C
   Data size: 12 bytes
Clearing user uninitialized arrays segment...
   Address range: 0x2400B000 - 0x2400BA00
   Data size: 2560 bytes
User data segments initialization completed!

=== User-defined Memory Segments Information ===
1. Initialized variables segment:
     RAM address: 0x24008000 - 0x2400800C (12 bytes)
     FLASH source: 0x08008000
2. Initialized arrays segment:
     RAM address: 0x24009000 - 0x24009034 (52 bytes)
     FLASH source: 0x08009000
3. Uninitialized variables segment:
     RAM address: 0x2400A000 - 0x2400A00C (12 bytes)
4. Uninitialized arrays segment:
     RAM address: 0x2400B000 - 0x2400BA00 (2560 bytes)
5. Constant arrays segment:
     FLASH address: 0x0800A000 - 0x0800A200 (512 bytes)
==========================================

=== Verify Initialization Data ===
system_status = 1 (addr: 0x24008000)
device_serial = 0x12345678 (addr: 0x24008004)
calibration_data[0] = 100 (addr: 0x24009000)
calibration_data[9] = 1000 (addr: 0x24009012)
calibration_data array base addr: 0x24009000
device_info = GD32H7xx Device (addr: 0x24009014)
lookup_table[0] = 0xA55A (addr: 0x0800A000)
lookup_table[255] = 0x0000 (addr: 0x0800A0FE)
lookup_table array base addr: 0x0800A000

=== Test Uninitialized Data Usage ===
uart_tx_buffer = 0123456789 (addr: 0x2400B000)
adc_samples[5] = 500 (addr: 0x2400B40A)
adc_samples array base addr: 0x2400B400

=== System Ready ===
User-defined data segments test completed successfully!
```

2.4. Load functions to the specified location

2.4.1. Custom section

Add user-defined sections in the linker script to allocate functions to specific addresses, with address allocation as shown in ***Table 2-8. Overview of function loading to specified addresses***, with linker file code shown in ***Table 2-9. Loading global variables and global arrays to specified locations in gd32h7xx_flash.ld***.

Table 2-8. Overview of function loading to specified addresses

Segment type	Section name	FLASH address	RAM address	Description
FLASH function	.text.flash_functions	0x08010000	-	Execute directly in FLASH
RAM function	.text.ram_functions	Auto allocation	0x2400C000	Copy to RAM for execution
ITCM function	.text.itcm_functions	Auto allocation	0x00001000	Copy to ITCMRAM for execution
DTCM function	.text.dtcm_functions	Auto allocation	0x20001000	Copy to DTCMRAM for execution

Table 2-9. Loading global variables and global arrays to specified locations in gd32h7xx_flash.ld

```
/* User custom function scatter loading */
.user_flash_functions 0x08010000 :
{
. = ALIGN(4);
__user_flash_functions_start__ = .;
KEEP(*(.text.flash_functions))
. = ALIGN(4);
__user_flash_functions_end__ = .;
__user_flash_functions_size__ = __user_flash_functions_end__ - __user_flash_functions_start__;
} > FLASH

.user_ram_functions 0x2400C000 :
{
. = ALIGN(4);
__user_ram_functions_start__ = .;
KEEP(*(.text.ram_functions))
. = ALIGN(4);
__user_ram_functions_end__ = .;
__user_ram_functions_size__ = __user_ram_functions_end__ - __user_ram_functions_start__;
} > AXISRAM AT> FLASH
__user_ram_functions_loadaddr__ = LOADADDR(.user_ram_functions);
```

```
.user_itcm_functions 0x00001000 :
{
. = ALIGN(4);
__user_itcm_functions_start__ = .;
KEEP(*(.text.itcm_functions))
. = ALIGN(4);
__user_itcm_functions_end__ = .;
__user_itcm_functions_size__ = __user_itcm_functions_end__ - __user_itcm_functions_start__;
} > ITCMRAM AT> FLASH
__user_itcm_functions_loadaddr__ = LOADADDR(.user_itcm_functions);

.user_dtcm_functions 0x20001000 :
{
. = ALIGN(4);
__user_dtcm_functions_start__ = .;
KEEP(*(.text.dtcm_functions))
. = ALIGN(4);
__user_dtcm_functions_end__ = .;
__user_dtcm_functions_size__ = __user_dtcm_functions_end__ - __user_dtcm_functions_start__;
} > DTCMRAM AT> FLASH
__user_dtcm_functions_loadaddr__ = LOADADDR(.user_dtcm_functions);
```

2.4.2. Function definition in C code

Define section attribute macros in the header file to improve code readability, as shown in ***Table 2-10. Header file macro definitions for loading functions to specified locations***.

Table 2-10. Header file macro definitions for loading functions to specified locations

```
/* Function section attributes */
#define FLASH_FUNCTION      __attribute__((section(".text.flash_functions")))
#define RAM_FUNCTION        __attribute__((section(".text.ram_functions")))
#define ITCM_FUNCTION       __attribute__((section(".text.itcm_functions")))
#define DTCM_FUNCTION       __attribute__((section(".text.dtcm_functions")))
```

Variable definition example as shown in ***Table 2-11. Function definition and implementation for loading functions to specified locations***.

Table 2-11. Function definition and implementation for loading functions to specified locations

```
FLASH_FUNCTION int FlashFunction_Calculate(int a, int b)
{
    printf("FlashFunction_Calculate: a=%d, b=%d\r\n", a, b);
    int result = 0;
```

```
    for(int i = 0; i < 50; i++) {
        result += (a * b) + (i % 10);
    }
    printf("FlashFunction_Calculate result: %d\r\n", result);
    return result;
}

FLASH_FUNCTION void FlashFunction_ProcessData(uint8_t* data, uint32_t len)
{
    printf("FlashFunction_ProcessData: processing %d bytes\r\n", len);
    for(uint32_t i = 0; i < len; i++) {
        data[i] = data[i] ^ 0x5A;
    }
    printf("Data processing completed\r\n");
}

RAM_FUNCTION int RamFunction_FastProcess(int data)
{
    //printf("RamFunction_FastProcess: input=0x%08X\r\n", data);
    volatile int temp = data;
    temp = (temp << 1) ^ 0xA5A5;
    temp = (temp >> 1) | 0x5A5A0000;
    //printf("RamFunction_FastProcess result: 0x%08X\r\n", temp);
    return temp;
}

RAM_FUNCTION void RamFunction_ProcessBuffer(uint16_t* buffer, uint32_t size)
{
    printf("RamFunction_ProcessBuffer: processing %d elements\r\n", size);
    for(uint32_t i = 0; i < size; i++) {
        buffer[i] = (buffer[i] << 1) | (buffer[i] >> 15);
    }
    printf("Buffer processing completed\r\n");
}

ITCM_FUNCTION int ItcmFunction_CriticalTask(int input)
{
    printf("ItcmFunction_CriticalTask: input=%d\r\n", input);
    register int result = input;
    result = (result * 3) + 7;
    result = result ^ 0x12345678;
    result = (result << 4) | (result >> 28);
```

```
    printf("ItcmFunction_CriticalTask result: %d\r\n", result);
    return result;
}

ITCM_FUNCTION uint32_t ItcmFunction_BitOperation(uint32_t value)
{
    printf("ItcmFunction_BitOperation: input=0x%08X\r\n", value);
    uint32_t result = value;
    result = ~result;
    result = (result & 0xAAAAAAAA) | ((result & 0x55555555) << 1);
    result = (result & 0xCCCCCCCC) | ((result & 0x33333333) << 2);
    printf("ItcmFunction_BitOperation result: 0x%08X\r\n", result);
    return result;
}

DTCM_FUNCTION int DtcmFunction_DataProcess(int value)
{
    printf("DtcmFunction_DataProcess: input=%d\r\n", value);
    static int buffer[32];
    for(int i = 0; i < 32; i++) {
        buffer[i] = value + (i * 10);
    }
    int sum = 0;
    for(int i = 0; i < 32; i++) {
        sum += buffer[i];
    }
    printf("DtcmFunction_DataProcess result: %d\r\n", sum);
    return sum;
}

DTCM_FUNCTION void DtcmFunction_ArrayOperation(uint32_t* array, uint32_t count)
{
    printf("DtcmFunction_ArrayOperation: processing %d elements\r\n", count);
    for(uint32_t i = 0; i < count; i++) {
        array[i] = array[i] * 2 + 1;
    }
    printf("Array operation completed\r\n");
}
```

2.4.3. Custom initialization implementation

Call the custom code section initialization function during system startup, as shown in ***Table***

2-12. Initialization implementation for loading functions to specified locations.

Table 2-12. Initialization implementation for loading functions to specified locations

```
void UserFunctions_Init(void)
{
    uint32_t *src, *dst;
    uint32_t size;

    /* Copy RAM functions */
    src = &__user_ram_functions_loadaddr__;
    dst = &__user_ram_functions_start__;
    size = ((uint32_t)&__user_ram_functions_end__ - (uint32_t)&__user_ram_functions_start__) /
4;

    for(uint32_t i = 0; i < size; i++) {
        *dst++ = *src++;
    }
    /* Copy ITCM functions */
    src = &__user_itcm_functions_loadaddr__;
    dst = &__user_itcm_functions_start__;
    size = ((uint32_t)&__user_itcm_functions_end__ - (uint32_t)&__user_itcm_functions_start__) /
4;

    for(uint32_t i = 0; i < size; i++) {
        *dst++ = *src++;
    }

    /* Copy DTCM functions */
    printf("Copying DTCM functions...\r\n");
    src = &__user_dtcm_functions_loadaddr__;
    dst = &__user_dtcm_functions_start__;
    size = ((uint32_t)&__user_dtcm_functions_end__ - (uint32_t)&__user_dtcm_functions_start__)
/ 4;

    for(uint32_t i = 0; i < size; i++) {
        *dst++ = *src++;
    }
}
```

Note: Initialization is placed before the cache initialization function.

2.4.4. Test verification

Test the memory layout verification of loading functions to specified locations through UART printing, with the code shown in ***Table 2-13. Memory verification code for loading functions to specified locations***.

Table 2-13. Memory verification code for loading functions to specified locations

```
void TestUserFunctions(void)
{
    printf("=== Testing User Functions ===\r\n");

    uint8_t test_data[16] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
                             0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10};

    uint16_t test_buffer[8] = {100, 200, 300, 400, 500, 600, 700, 800};
    uint32_t test_array[5] = {10, 20, 30, 40, 50};

    printf("\n--- Testing FLASH Functions ---\r\n");
    FlashFunction_Calculate(15, 25);
    FlashFunction_ProcessData(test_data, 8);

    printf("\n--- Testing RAM Functions ---\r\n");
    RamFunction_FastProcess(0x12345678);
    RamFunction_ProcessBuffer(test_buffer, 8);

    printf("\n--- Testing ITCM Functions ---\r\n");
    ItcmFunction_CriticalTask(42);
    ItcmFunction_BitOperation(0xAAAA5555);

    printf("\n--- Testing DTCM Functions ---\r\n");
    DtcmFunction_DataProcess(100);
    DtcmFunction_ArrayOperation(test_array, 5);

    printf("\n================================\r\n\r\n");
}

/*!
    \brief        main function
    \param[in]    none
    \param[out] none
    \retval       none
*/
void main(void)
```

```
{
    gd_eval_com_init(EVAL_COM);
    printf("GD32H7xx User Functions Address Test\r\n");
    printf("====================================\r\n");
    /* initialize user functions */
    UserFunctions_Init();
    /* enable the CPU cache */
    cache_enable();
    mpu_config();

    /* print all function addresses */
    PrintFunctionAddresses();
    /* test all functions */
    TestUserFunctions();

    printf("All function tests completed!\r\n");
    printf("Function address verification complete.\r\n\r\n");

    /* main application loop with periodic address display */
    uint32_t counter = 0;

    while(1) {
        counter++;

        if(counter % 5 == 1) {
            /* periodically show function addresses */
            printf("=== Periodic Address Check %d ===\r\n", counter);
            printf("FlashFunction_Calculate: 0x%08X\r\n", (uint32_t)FlashFunction_Calculate);
            printf("RamFunction_FastProcess:                    0x%08X\r\n",
(uint32_t)RamFunction_FastProcess);
            printf("ItcmFunction_CriticalTask: 0x%08X\r\n", (uint32_t)ItcmFunction_CriticalTask);
            printf("DtcmFunction_DataProcess:                   0x%08X\r\n",
(uint32_t)DtcmFunction_DataProcess);
            printf("============================\r\n\r\n");
        }

        /* call functions to verify they work at reported addresses */
        FlashFunction_Calculate(counter, counter + 10);
        RamFunction_FastProcess(counter * 0x1000);
        ItcmFunction_CriticalTask(counter + 100);
        DtcmFunction_DataProcess(counter * 50);
```

```
        /* delay between iterations */
        for(volatile int i = 0; i < 2000000; i++);

        /* stop after several iterations */
        if(counter >= 3) {
            printf("Address verification demo completed.\r\n");
            break;
        }
    }

    while(1) {
    }
}
```

Test print log as shown in ***Table 2-14. Load function to the specified location log***, test results are consistent with the actual allocation.

Table 2-14. Load function to the specified location log

```
GD32H7xx User Functions Address Test
==================================
Initializing user functions...
Copying RAM functions...
  RAM functions: 0x2400C000 - 0x2400C0B0 (176 bytes)
Copying ITCM functions...
  ITCM functions: 0x00001000 - 0x000010A0 (160 bytes)
Copying DTCM functions...
  DTCM functions: 0x20001000 - 0x200010E0 (224 bytes)
User functions initialization completed!


=== Function Entry Addresses ===

--- FLASH Functions (Direct Execution) ---
Section Range: 0x08010000 - 0x080100C4
FlashFunction_Calculate        : 0x08010001
FlashFunction_ProcessData      : 0x08010071

--- RAM Functions (Copied from FLASH) ---
FLASH Source: 0x080100C4
RAM Target    : 0x2400C000 - 0x2400C0B0
RamFunction_FastProcess        : 0x2400C001
RamFunction_ProcessBuffer      : 0x2400C035
```

```
--- ITCM Functions ---
FLASH Source: 0x08010174
ITCM Target : 0x00001000 - 0x000010A0
ItcmFunction_CriticalTask     : 0x00001001
ItcmFunction_BitOperation     : 0x00001049

--- DTCM Functions ---
FLASH Source: 0x08010214
DTCM Target : 0x20001000 - 0x200010E0
DtcmFunction_DataProcess       : 0x20001001
DtcmFunction_ArrayOperation : 0x2000107D

================================

=== Testing User Functions ===

--- Testing FLASH Functions ---
FlashFunction_Calculate: a=15, b=25
FlashFunction_Calculate result: 18975
FlashFunction_ProcessData: processing 8 bytes
Data processing completed

--- Testing RAM Functions ---
RamFunction_ProcessBuffer: processing 8 elements
Buffer processing completed

--- Testing ITCM Functions ---
ItcmFunction_CriticalTask: input=42
ItcmFunction_CriticalTask result: 591753169
ItcmFunction_BitOperation: input=0xAAAA5555
ItcmFunction_BitOperation result: 0x88888888

--- Testing DTCM Functions ---
DtcmFunction_DataProcess: input=100
DtcmFunction_DataProcess result: 8160
DtcmFunction_ArrayOperation: processing 5 elements
Array operation completed

================================

All function tests completed!
Function address verification complete.
```

```
=== Periodic Address Check 1 ===
FlashFunction_Calculate: 0x08010001
RamFunction_FastProcess: 0x2400C001
ItcmFunction_CriticalTask: 0x00001001
DtcmFunction_DataProcess: 0x20001001
==============================

FlashFunction_Calculate: a=1, b=11
FlashFunction_Calculate result: 775
ItcmFunction_CriticalTask: input=101
ItcmFunction_CriticalTask result: 591754465
DtcmFunction_DataProcess: input=50
DtcmFunction_DataProcess result: 6560
FlashFunction_Calculate: a=2, b=12
FlashFunction_Calculate result: 1425
ItcmFunction_CriticalTask: input=102
ItcmFunction_CriticalTask result: 591754257
DtcmFunction_DataProcess: input=100
DtcmFunction_DataProcess result: 8160
FlashFunction_Calculate: a=3, b=13
FlashFunction_Calculate result: 2175
ItcmFunction_CriticalTask: input=103
ItcmFunction_CriticalTask result: 591754305
DtcmFunction_DataProcess: input=150
DtcmFunction_DataProcess result: 9760
Address verification demo completed.
```

2.5. Load .c file code section to the specified location

2.5.1. Custom section

Add a user-defined section in the linker script to allocate the code segment of the .c file to a specified address, as shown in ***Table 2-15. Overview of .c loading to specified address allocation***, and the linker file code is shown in ***Table 2-16. Allocation of global variables and global arrays to specified location in gd32h7xx_flash.ld***.

Table 2-15. Overview of .c loading to specified address allocation

Module file	Section name	FLASH address	RAM address	Description
flash_module.c	.flash_module_functions	0x08020000	-	Execute directly in FLASH
ram_module.c	.ram_module_functions	Auto allocation	0x2400D000	Copy to RAM for

Module file	Section name	FLASH address	RAM address	Description
				execution
itcm_module.c	.itcm_module_functions	Auto allocation	0x00002000	Copy to ITCMRAM for execution
dtcm_module.c	.dtcm_module_functions	Auto allocation	0x20002000	Copy to DTCMRAM for execution
sram0_module.c	.sram0_module_functions	Auto allocation	0x3000800	Copy to SRAM0 for execution

Table 2-16. Allocation of global variables and global arrays to specified location in gd32h7xx_flash.ld

```
/* user file-based function scatter loading */
.flash_module_functions ALIGN(0x08020000, 4) :
{
    . = ALIGN(4);
    __flash_module_functions_start__ = .;
    KEEP(*flash_module.o(.text))
    KEEP(*flash_module.o(.text*))
    . = ALIGN(4);
    __flash_module_functions_end__ = .;
} > FLASH

.ram_module_functions ALIGN(0x2400D000,4):
{
    . = ALIGN(4);
    __ram_module_functions_start__ = .;
    KEEP(*ram_module.o(.text))
    KEEP(*ram_module.o(.text*))
    . = ALIGN(4);
    __ram_module_functions_end__ = .;
} > AXISRAM AT> FLASH
__ram_module_functions_loadaddr__ = LOADADDR(.ram_module_functions);

.itcm_module_functions    ALIGN(0x00002000,4):
{
    . = ALIGN(4);
    __itcm_module_functions_start__ = .;
    KEEP(*itcm_module.o(.text))
    KEEP(*itcm_module.o(.text*))
    . = ALIGN(4);
    __itcm_module_functions_end__ = .;
} > ITCMRAM AT> FLASH
```

```
__itcm_module_functions_loadaddr__ = LOADADDR(.itcm_module_functions);

.dtcm_module_functions    ALIGN(0x20002000,4):
{
    . = ALIGN(4);
    __dtcm_module_functions_start__ = .;
    KEEP(*dtcm_module.o(.text))
    KEEP(*dtcm_module.o(.text*))
    . = ALIGN(4);
    __dtcm_module_functions_end__ = .;
} > DTCMRAM AT> FLASH
__dtcm_module_functions_loadaddr__ = LOADADDR(.dtcm_module_functions);

.sram0_module_functions    ALIGN(0x30000800,4):
{
    . = ALIGN(4);
    __sram0_module_functions_start__ = .;
    KEEP(*sram0_module.o (.text))
    KEEP(*sram0_module.o(.text*))
    . = ALIGN(4);
    __sram0_module_functions_end__ = .;
} > SRAM0 AT> FLASH
__sram0_module_functions_loadaddr__ = LOADADDR(.sram0_module_functions);
```

2.5.2. File definition in C code

Create dtcm_module.c, flash_module.c, itcm_module.c, ram_module.c, and sram0_module.c files along with their corresponding header files, and define respective functions in each .c file. Details as shown in ***Table 2-17. Definition of .c files***.

Table 2-17. Definition of .c files

```
/* dtcm_module.c */
int DtcmModule_Function1(int a, int b)
{
    /* Function with fast data access */
    printf("DtcmModule_Function1: a=%d, b=%d\r\n", a, b);
    dtcm_buffer[dtcm_counter % 64] = a + b;
    dtcm_counter++;
    return dtcm_buffer[(dtcm_counter - 1) % 64];
}
int DtcmModule_Function2(int x)
{
    /* Data-intensive function */
```

```
    printf("DtcmModule_Function2: x=%d\r\n", x);
    for(int i = 0; i < 32; i++) {
        dtcm_buffer[i] = x * i;
    }
    int sum = 0;
    for(int i = 0; i < 32; i++) {
        sum += dtcm_buffer[i];
    }
    return sum;
}
/* flash_module.c */
int FlashModule_Function1(int a, int b)
{
    /* Complex calculation function executed from FLASH */
    printf("FlashModule_Function1: a=%d, b=%d\r\n", a, b);
    int result = 0;
    for(int i = 0; i < 50; i++) {
        result += (a * b) + (i % 7);
    }
    return result;
}

int FlashModule_Function2(int x)
{
    /* Mathematical processing function */
    printf("FlashModule_Function2: x=%d\r\n", x);
    int temp = x;
    temp = temp * temp + 100;
    temp = temp % 1000;
    return temp;
}
/* itcm_module.c */
int ItcmModule_Function1(int a, int b)
{
    /* Ultra-fast critical function in ITCM */
    register int result = a * b;
    result = result + (result >> 4);
    result = result ^ 0x12345678;
    return result;
}

int ItcmModule_Function2(int x)
```

```
{
    /* Real-time processing function */
    register int temp = x;
    temp = (temp * 7) + 13;
    temp = temp & 0xFFFF;
    return temp;
}
/* ram_module.c */
int RamModule_Function1(int a, int b)
{
    /* Fast calculation function executed from RAM */
    printf("RamModule_Function1: a=%d, b=%d\r\n", a, b);
    register int result = a + b;
    result = result << 2;
    result = result ^ 0xAAAA;
    return result;
}

int RamModule_Function2(int x)
{
    /* High-speed processing function */
    printf("RamModule_Function2: x=%d\r\n", x);
    volatile int temp = x;
    for(int i = 0; i < 20; i++) {
        temp = (temp << 1) | (temp >> 31);
    }
    return temp;
}
/* sram0_module.c */
int Sram0Module_Function1(int a, int b)
{
    /* Function executed from SRAM0 */
    printf("Sram0Module_Function1: a=%d, b=%d\r\n", a, b);
    int result = a - b;
    if(result < 0) result = -result;
    return result * 2;
}
int Sram0Module_Function2(int x)
{
    /* Special processing function */
    printf("Sram0Module_Function2: x=%d\r\n", x);
    int temp = x;
```

```
    temp = temp + 0x1000;
    temp = temp & 0x7FFF;
    return temp;
}
```

2.5.3. Custom initialization implementation

Call the custom code section initialization function at system startup, as shown in ***Table 2-18. Initialization implementation of .c files loaded to specified location***.

Table 2-18. Initialization implementation of .c files loaded to specified location

```
void ModuleLoader_Init(void)
{
    uint32_t *src, *dst;
    uint32_t size;
    /* copy RAM module functions */
    src = &__ram_module_functions_loadaddr__;
    dst = &__ram_module_functions_start__;
    size = ((uint32_t)&__ram_module_functions_end__ -
(uint32_t)&__ram_module_functions_start__) / 4;

    for(uint32_t i = 0; i < size; i++) {
        *dst++ = *src++;
    }

    /* copy ITCM module functions */
    src = &__itcm_module_functions_loadaddr__;
    dst = &__itcm_module_functions_start__;
    size = ((uint32_t)&__itcm_module_functions_end__ -
(uint32_t)&__itcm_module_functions_start__) / 4;

    for(uint32_t i = 0; i < size; i++) {
        *dst++ = *src++;
    }

    /* copy DTCM module functions */
    src = &__dtcm_module_functions_loadaddr__;
    dst = &__dtcm_module_functions_start__;
    size = ((uint32_t)&__dtcm_module_functions_end__ -
(uint32_t)&__dtcm_module_functions_start__) / 4;

    for(uint32_t i = 0; i < size; i++) {
        *dst++ = *src++;
```

```
    }

    /* copy SRAM0 module functions */
    src = &__sram0_module_functions_loadaddr__;
    dst = &__sram0_module_functions_start__;
    size = ((uint32_t)&__sram0_module_functions_end__ -
(uint32_t)&__sram0_module_functions_start__) / 4;

    for(uint32_t i = 0; i < size; i++) {
        *dst++ = *src++;
    }
}
```

Note: Initialization is placed before the cache initialization function.

2.5.4. Test verification

By using UART printing, test the function loading into the specified memory layout for verification. Code as shown in ***Table 2-19. Verification code for .c file loaded into the specified memory***.

Table 2-19. Verification code for .c file loaded into the specified memory

```
void TestAllModules(void)
{
    int result;
    uint8_t test_data[16] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
                             0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10};

    printf("=== Testing C File-based Module Functions ===\r\n");

    /* Test FLASH module */
    printf("\n--- Testing FLASH Module (flash_module.c) ---\r\n");
    printf("Executing FlashModule_Function1 at 0x%08X\r\n", (uint32_t)FlashModule_Function1);
    result = FlashModule_Function1(15, 25);
    printf("Result: %d\r\n", result);

    printf("Executing FlashModule_Function2 at 0x%08X\r\n", (uint32_t)FlashModule_Function2);
    result = FlashModule_Function2(100);
    printf("Result: %d\r\n", result);

    printf("Executing FlashModule_ProcessData at 0x%08X\r\n",
(uint32_t)FlashModule_ProcessData);
    FlashModule_ProcessData(test_data, 8);
```

```
    /* Test RAM module */
    printf("\n--- Testing RAM Module (ram_module.c) ---\r\n");
    printf("Executing RamModule_Function1 at 0x%08X\r\n", (uint32_t)RamModule_Function1);
    result = RamModule_Function1(10, 20);
    printf("Result: %d\r\n", result);

    printf("Executing RamModule_Function2 at 0x%08X\r\n", (uint32_t)RamModule_Function2);
    result = RamModule_Function2(0x1234);
    printf("Result: 0x%08X\r\n", result);

    printf("Executing RamModule_ProcessData at 0x%08X\r\n",
(uint32_t)RamModule_ProcessData);
    RamModule_ProcessData(test_data, 8);

    /* Test ITCM module */
    printf("\n--- Testing ITCM Module (itcm_module.c) ---\r\n");
    printf("Executing ItcmModule_Function1 at 0x%08X\r\n", (uint32_t)ItcmModule_Function1);
    result = ItcmModule_Function1(7, 11);
    printf("Result: %d\r\n", result);

    printf("Executing ItcmModule_Function2 at 0x%08X\r\n", (uint32_t)ItcmModule_Function2);
    result = ItcmModule_Function2(500);
    printf("Result: %d\r\n", result);

    printf("Executing ItcmModule_ProcessData at 0x%08X\r\n",
(uint32_t)ItcmModule_ProcessData);
    ItcmModule_ProcessData(test_data, 8);

    /* Test DTCM module */
    printf("\n--- Testing DTCM Module (dtcm_module.c) ---\r\n");
    printf("Executing DtcmModule_Function1 at 0x%08X\r\n", (uint32_t)DtcmModule_Function1);
    result = DtcmModule_Function1(33, 44);
    printf("Result: %d\r\n", result);

    printf("Executing DtcmModule_Function2 at 0x%08X\r\n", (uint32_t)DtcmModule_Function2);
    result = DtcmModule_Function2(200);
    printf("Result: %d\r\n", result);

    printf("Executing DtcmModule_ProcessData at 0x%08X\r\n",
(uint32_t)DtcmModule_ProcessData);
    DtcmModule_ProcessData(test_data, 8);
```

```
    /* Test SRAM0 module */
    printf("\n--- Testing SRAM0 Module (sram0_module.c) ---\r\n");
    printf("Executing Sram0Module_Function1 at 0x%08X\r\n", (uint32_t)Sram0Module_Function1);
    result = Sram0Module_Function1(60, 40);
    printf("Result: %d\r\n", result);

    printf("Executing Sram0Module_Function2 at 0x%08X\r\n", (uint32_t)Sram0Module_Function2);
    result = Sram0Module_Function2(0x8000);
    printf("Result: 0x%04X\r\n", result);

    printf("Executing Sram0Module_ProcessData at 0x%08X\r\n",
(uint32_t)Sram0Module_ProcessData);
    Sram0Module_ProcessData(test_data, 8);

    printf("\n========================================\r\n\r\n");
}

/*!
    \brief      main function
    \param[in]  none
    \param[out] none
    \retval     none
*/
void main(void)
{
    gd_eval_com_init(EVAL_COM);

    printf("GD32H7xx C File-based Scatter Loading Example\r\n");
    printf("==========================================\r\n");
    printf("This example demonstrates loading entire C files\r\n");
    printf("to different memory regions for optimized execution.\r\n\r\n");

    /* Initialize all modules */
    ModuleLoader_Init();

    /* Enable the CPU cache */
    cache_enable();
    mpu_config();

    /* Print all module function addresses */
    PrintModuleAddresses();
```

```
/* Test all modules */
TestAllModules();

printf("All C file-based module tests completed!\r\n");
printf("Each module file is loaded to its designated memory region.\r\n");

uint32_t loop_count = 0;

while(1) {
    loop_count++;

    /* Periodic address verification */
    if(loop_count % 10 == 1) {
        printf("\n=== Periodic Address Verification (Loop %d) ===\r\n", loop_count);
        printf("flash_module.c functions in FLASH:\r\n");
        printf("    FlashModule_Function1: 0x%08X\r\n", (uint32_t)FlashModule_Function1);

        printf("ram_module.c functions in AXISRAM:\r\n");
        printf("    RamModule_Function1    : 0x%08X\r\n", (uint32_t)RamModule_Function1);

        printf("itcm_module.c functions in ITCMRAM:\r\n");
        printf("    ItcmModule_Function1 : 0x%08X\r\n", (uint32_t)ItcmModule_Function1);

        printf("dtcm_module.c functions in DTCMRAM:\r\n");
        printf("    DtcmModule_Function1 : 0x%08X\r\n", (uint32_t)DtcmModule_Function1);

        printf("sram0_module.c functions in SRAM0:\r\n");
        printf("    Sram0Module_Function1: 0x%08X\r\n", (uint32_t)Sram0Module_Function1);
        printf("================================================\r\n\r\n");
    }

    /* Simple delay */
    for(volatile int i = 0; i < 2000000; i++);

    /* Stop after a few loops for demonstration */
    if(loop_count >= 3) {
        printf("C file-based scatter loading demonstration completed.\r\n");
        break;
    }
}
```

```
    while(1) {

    }
}
```

Test log as shown in ***Table 2-20. Log for loading .c file into the specified memory***. The test result is consistent with the actual allocation.

Table 2-20. Log for loading .c file into the specified memory

```
GD32H7xx C File-based Scatter Loading Example
==============================================
This example demonstrates loading entire C files
to different memory regions for optimized execution.

Initializing C file-based module functions...
Copying RAM module functions...
  RAM module: 0x2400D000 - 0x2400D158 (344 bytes)
Copying ITCM module functions...
  ITCM module: 0x00002000 - 0x000020E0 (224 bytes)
Copying DTCM module functions...
  DTCM module: 0x20002000 - 0x20002198 (408 bytes)
Copying SRAM0 module functions...
  SRAM0 module: 0x30000800 - 0x30000928 (296 bytes)
C file-based module functions initialization completed!


=== Module Function Entry Addresses ===

--- FLASH Module (flash_module.c) ---
Section Range: 0x08020000 - 0x08020130
FlashModule_Function1          : 0x08020001
FlashModule_Function2          : 0x08020065
FlashModule_ProcessData        : 0x080200B1
FlashModule_GetProcessedValue: 0x08020115

--- RAM Module (ram_module.c) ---
FLASH Source: 0x08020130
RAM Target   : 0x2400D000 - 0x2400D158
RamModule_Function1            : 0x2400D001
RamModule_Function2            : 0x2400D035
RamModule_ProcessData          : 0x2400D075
RamModule_SortArray            : 0x2400D11F
```

```
--- ITCM Module (itcm_module.c) ---
FLASH Source: 0x08020288
ITCM Target : 0x00002000 - 0x000020E0
ItcmModule_Function1             : 0x00002001
ItcmModule_Function2             : 0x0000202D
ItcmModule_ProcessData           : 0x0000204F
ItcmModule_CriticalCalculation: 0x000020B3

--- DTCM Module (dtcm_module.c) ---
FLASH Source: 0x08020368
DTCM Target : 0x20002000 - 0x20002198
DtcmModule_Function1             : 0x20002001
DtcmModule_Function2             : 0x2000206D
DtcmModule_ProcessData           : 0x200020D9
DtcmModule_GetBufferSum          : 0x20002155

--- SRAM0 Module (sram0_module.c) ---
FLASH Source: 0x08020500
SRAM0 Target: 0x30000800 - 0x30000928
Sram0Module_Function1            : 0x30000801
Sram0Module_Function2            : 0x30000839
Sram0Module_ProcessData          : 0x3000086D
Sram0Module_ValidateAndProcess: 0x300008F9

========================================

=== Testing C File-based Module Functions ===

--- Testing FLASH Module (flash_module.c) ---
Executing FlashModule_Function1 at 0x08020001
FlashModule_Function1: a=15, b=25
Result: 18897
Executing FlashModule_Function2 at 0x08020065
FlashModule_Function2: x=100
Result: 100
Executing FlashModule_ProcessData at 0x080200B1
FlashModule_ProcessData: processing 8 bytes

--- Testing RAM Module (ram_module.c) ---
Executing RamModule_Function1 at 0x2400D001
RamModule_Function1: a=10, b=20
```

```
Result: 43730
Executing RamModule_Function2 at 0x2400D035
RamModule_Function2: x=4660
Result: 0xFFFFFFFF
Executing RamModule_ProcessData at 0x2400D075
RamModule_ProcessData: processing 8 bytes

--- Testing ITCM Module (itcm_module.c) ---
Executing ItcmModule_Function1 at 0x00002001
Result: 305419817
Executing ItcmModule_Function2 at 0x0000202D
Result: 3513
Executing ItcmModule_ProcessData at 0x0000204F

--- Testing DTCM Module (dtcm_module.c) ---
Executing DtcmModule_Function1 at 0x20002001
DtcmModule_Function1: a=33, b=44
Result: 77
Executing DtcmModule_Function2 at 0x2000206D
DtcmModule_Function2: x=200
Result: 99200
Executing DtcmModule_ProcessData at 0x200020D9
DtcmModule_ProcessData: processing 8 bytes

--- Testing SRAM0 Module (sram0_module.c) ---
Executing Sram0Module_Function1 at 0x30000801
Sram0Module_Function1: a=60, b=40
Result: 40
Executing Sram0Module_Function2 at 0x30000839
Sram0Module_Function2: x=32768
Result: 0x1000
Executing Sram0Module_ProcessData at 0x3000086D
Sram0Module_ProcessData: processing 8 bytes

==========================================

All C file-based module tests completed!
Each module file is loaded to its designated memory region.

=== Periodic Address Verification (Loop 1) ===
flash_module.c functions in FLASH:
  FlashModule_Function1: 0x08020001
```

```
ram_module.c functions in AXISRAM:
   RamModule_Function1    : 0x2400D001
itcm_module.c functions in ITCMRAM:
   ItcmModule_Function1 : 0x00002001
dtcm_module.c functions in DTCMRAM:
   DtcmModule_Function1 : 0x20002001
sram0_module.c functions in SRAM0:
   Sram0Module_Function1: 0x30000801
================================================

C file-based scatter loading demonstration completed.
```

2.6. SDRAM scatter loading implementation.

The implementation of SDRAM scatter loading can refer to sections ***2.3***, ***2.4*** and ***2.5***. Before performing custom initialization, EXMC initialization and MPU configuration need to be implemented. Code as shown in ***Table 2-21. SDRAM initialization code***.

Table 2-21. SDRAM initialization code

```
#include "exmc_sdram.h"
/*!
    \brief          initialize the sdram
    \param[in]    none
    \param[out] none
    \retval         none
*/
void DoInit(void)
{
    /* configure the clock of EXMC */
    rcu_exmc_config();
    /* configure the MPU */
    mpu_config();
    /* configure the EXMC access mode */
    exmc_synchronous_dynamic_ram_init(EXMC_SDRAM_DEVICE0);
    __IO int i,j;
    for(i=0;i<500;i++){
        for(j=0;j<5000;j++);
    }
}
```

Note: In the default configuration of the M7 core, certain addresses are in regions where instruction execution is prohibited. Thus, if code is loaded into those regions, errors will occur during execution. The SDRAM address allocation in the EXMC of GD32H7xx is 0xC0000000-

0xDFFFFFFF, which lies within this prohibited address range. Configure the MPU (Memory Protection Unit) registers to enable instruction execution in the 0xC0000000 address range.

3. Revision history

Table 3-1. Revision history

Revision No.	Description	Date
1.0	Initial Release	Dec.30 2025

Important Notice

This document is the property of GigaDevice Semiconductor Inc. and its subsidiaries (the "Company"). This document, including any product of the Company described in this document (the “Product”), is owned by the Company according to the laws of the People’s Republic of China and other applicable laws. The Company reserves all rights under such laws and no Intellectual Property Rights are transferred (either wholly or partially) or licensed by the Company (either expressly or impliedly) herein. The names and brands of third party referred thereto (if any) are the property of their respective owner and referred to for identification purposes only.

To the maximum extent permitted by applicable law, the Company makes no representations or warranties of any kind, express or implied, with regard to the merchantability and the fitness for a particular purpose of the Product, nor does the Company assume any liability arising out of the application or use of any Product. Any information provided in this document is provided only for reference purposes. It is the sole responsibility of the user of this document to determine whether the Product is suitable and fit for its applications and products planned, and properly design, program, and test the functionality and safety of its applications and products planned using the Product. The Product is designed, developed, and/or manufactured for ordinary business, industrial, personal, and/or household applications only, and the Product is not designed or intended for use in (i) safety critical applications such as weapons systems, nuclear facilities, atomic energy controller, combustion controller, aeronautic or aerospace applications, traffic signal instruments, pollution control or hazardous substance management; (ii) life-support systems, other medical equipment or systems (including life support equipment and surgical implants); (iii) automotive applications or environments, including but not limited to applications for active and passive safety of automobiles (regardless of front market or aftermarket), for example, EPS, braking, ADAS (camera/fusion), EMS, TCU, BMS, BSG, TPMS, Airbag, Suspension, DMS, ICMS, Domain, ESC, DCDC, e-clutch, advanced-lighting, etc.. Automobile herein means a vehicle propelled by a self-contained motor, engine or the like, such as, without limitation, cars, trucks, motorcycles, electric cars, and other transportation devices; and/or (iv) other uses where the failure of the device or the Product can reasonably be expected to result in personal injury, death, or severe property or environmental damage (collectively "Unintended Uses"). Customers shall take any and all actions to ensure the Product meets the applicable laws and regulations. The Company is not liable for, in whole or in part, and customers shall hereby release the Company as well as its suppliers and/or distributors from, any claim, damage, or other liability arising from or related to all Unintended Uses of the Product. Customers shall indemnify and hold the Company, and its officers, employees, subsidiaries, affiliates as well as its suppliers and/or distributors harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of the Product.

Information in this document is provided solely in connection with the Product. The Company reserves the right to make changes, corrections, modifications or improvements to this document and the Product described herein at any time without notice. The Company shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. Information in this document supersedes and replaces information previously supplied in any prior versions of this document.