GigaDevice Semiconductor Inc.

GD32 Embedded Builder Linker Script Files Introduction for Arm Cortex-M Processor

Application Note
AN200

Revision 1.0

(Dec. 2025)

Table of Contents

List of Figures

List of Tables

1. Introduction

ARM GCC (GNU Compiler Collection) is an open-source compiler toolset for ARM architecture, maintained and developed by the GNU organization. ARM GCC includes a series of tools such as the compiler (arm-none-eabi-gcc), assembler (arm-none-eabi-as), linker (arm-none-eabi-ld), and debugger (gdb), used for developing embedded system software based on ARM architecture.

ARM LD (GNU linker) is a linker within the GNU toolchain, used to link compiled object files (including executable files, library files, etc.) into the final executable file or shared library. In ARM architecture embedded system development, ARM LD is widely utilized for generating target files for ARM architecture.

GD32 MCU covers ARM Cortex® M3/M4/M23/M33/M7 core products. During development, the Embedded Builder IDE based on the GCC toolchain can be used for software development, utilizing linker script files to define the target device's memory layout and related configurations. This manual aims to assist developers in understanding and using GD32 MCU linker scripts in Embedded Builder, as well as configuring some common issues. Linker scripts are the rules and configuration files on which the compiler and linker rely when generating executable files. Through linker scripts, developers can control the placement of program code and data, define memory layouts, and other critical tasks.

GD32 MCU linker scripts typically include the following sections:

- MEMORY: Defines the memory layout of the target device, including the start address and size of Flash memory and RAM.
- SECTIONS: Defines the location and attributes of each section, including code, read-only data, read-write data, and uninitialized data.
- ENTRY: Specifies the program's entry address, typically the Reset_Handler function.
- GROUP: Used to package multiple library files into a logical library for combined processing during linking.

This manual analyzes using the GD32MCU generic linker script as an example, applicable to GD32 Arm Cortex® M3/M4/M23/M33/M7 core MCU products.

2. Linker script description

2.1. MEMORY

In GCC ld files, the MEMORY{} syntax is used to define the memory layout of the target device. In GD32MCU development, this syntax specifies which memory regions of the target device should store the program's code and data.

Table 2-1. General form of MEMORY{} syntax

```
MEMORY {
   memory_region_name(attribute) : ORIGIN = origin_address, LENGTH = length;
   /* More memory region definitions */
}
```

Here, memory_region_name is the identifier for naming the memory region, origin_address is the starting address of the memory region, and length is the size of the memory region. Attribute defines the properties of the memory region, such as readable, writable, and executable. Common attributes include:

Table 2-2. Common attributes of MEMORY

`rx`: Readable and Executable.
`xrw`: Executable, Readable, and Writable.
`rw`: Readable and Writable.
`x`: Executable.

Table 2-3. GD32MCU MEMORY region definitions

```
MEMORY
{
FLASH (rx)          : ORIGIN = 0x08000000, LENGTH = 1024K
RAM (xrw)           : ORIGIN = 0x24000000, LENGTH = 512K
}
```

In ***Table 2-3. GD32MCU MEMORY region definitions***, two memory regions are defined: FLASH and RAM. The starting address of FLASH is 0x08000000, with a size of 1024K, and the starting address of RAM is 0x24000000, with a size of 512K. The FLASH region is readable and executable, while the RAM region is readable, writable, and executable.

2.2. ENTRY

In the linker script, the ENTRY command is used to specify the program's entry point, which is the starting address for program execution. Specifically, the ENTRY command allows you to designate a symbol as the entry point, and when the program is loaded and executed, the controller will start execution from this symbol.

The general syntax of the ENTRY command:

Table 2-4. General form of ENTRY syntax

```
ENTRY(symbol)
```

Where symbol is a symbol defined during linking, typically the name of a function or label. During program loading and execution, the controller will start executing from the address represented by this symbol. Typically, the ENTRY instruction specifies the address of the Reset vector defined in the program's startup file, which is the first instruction of the program.

Table 2-5. GD32MCU ENTRY specifies the function entry point

```
ENTRY(Reset_Handler)
```

In GD32MCU, Reset_Handler is the program’s entry function. The ENTRY(Reset_Handler) command designates the Reset_Handler function as the program’s entry point. When the program is loaded and executed, the processor will start executing from the address represented by the Reset_Handler function.

2.3. SECTIONS

In the linker script, the SECTIONS command is used to define the allocation and arrangement of various sections of the program. Sections are logical units of the program in memory, including code sections, data sections, stack sections, etc. The SECTIONS command allows explicit specification of the memory distribution of these sections.

The following is the general syntax of the SECTIONS command:

Table 2-6. General form of SECTIONS syntax

```
SECTIONS {
        section1 : {
                /* content of section1 */
        }
        section2 : {
                /* content of section2 */
        }
        /* definition of more sections */
}
```

In the SECTIONS command, multiple sections can be defined, and each section can be assigned specific attributes and contents, including code sections, data sections, read-only sections, and other custom sections. The definition of these sections directly affects the structure and functionality of the generated executable or library files.

2.3.1. VMA and LMA

In GCC link scripts, VMA and LMA are two core concepts used to control the distribution of program sections in memory.

- VMA (Virtual Memory Address) refers to the virtual memory address, which is the memory address where the section resides during program execution, also known as the runtime address.
- LMA (Load Memory Address) refers to the load memory address, which is the physical address where the section is stored during programming, also known as the storage address.

In most cases, these two addresses are the same (e.g., code running directly in Flash). However, in embedded systems, they are often different. The most typical example is the initialization data section (.data): it must be stored in non-volatile Flash (LMA) but must be copied to readable and writable RAM (VMA) during program execution.

Common writing methods of VMA

The common writing methods of VMA are shown in ***Table 2-7. Common writing methods of VMA***.

Table 2-7. Common writing methods of VMA

1. Using memory regions

```
.data : {
    *(.data*)                   /* Include content from all .data related sections */
} > RAM                         /* VMA points to the RAM memory region */
/*
 * Explanation:
 * - Uses predefined memory region names.
 * - The linker automatically allocates starting from the base address of the RAM region.
 * - If the RAM region is already occupied by other sections, allocation starts from the next free location.
*/

.text : {
    *(.text*)                   /* Include content from all code sections */
} > FLASH                       /* VMA points to the FLASH memory region */
/*
```

```
 * Explanation:
 * - Code sections typically execute directly from FLASH.
 * - The FLASH region usually has read-only and executable attributes.
 * - VMA = LMA; no data copying is required.
*/
```

2. Directly specifying addresses

```
.data 0x20000000 : {          /* Directly specify VMA as 0x20000000 */
    *(.data*)                 /* Include content from all .data sections */
}
/*
 * Explanation:
 * - Specifies the starting address of the section.
 * - Suitable for scenarios requiring precise control over memory layout.
 * - Disadvantages: Lacks flexibility; high risk of address conflicts.
 * - If the address is already occupied, the linker will report an error.
 * - Note: The address must be within a valid memory range.
*/
```

3. Using alignment functions

```
.data ALIGN(4) : {                    /* Align VMA to a 4-byte boundary */
    *(.data*)
}
/*
 * Explanation:
 * - ALIGN(4): Aligns the current location counter to a 4-byte boundary.
 * - If the current location is already aligned, the address remains unchanged; if not, it is adjusted upwards to the nearest 4-byte boundary.
 */

.data ALIGN(0x20000000, 8) : {        /* 8-byte alignment for a specified address */
    *(.data*)                         /* Include content from all .data sections */
}
/*
 * Explanation:
 * - ALIGN(0x20000000, 8): Aligns 0x20000000 to an 8-byte boundary.
 * - First argument: Base address; Second argument: Alignment bytes (must be a power of 2).
 * - Alignment examples:
 *     ALIGN(0x20000001, 8) = 0x20000008    (Aligned upwards)
 *     ALIGN(0x20000000, 8) = 0x20000000    (Already aligned)
 * - 8-byte alignment is suitable for 64-bit data or double types.
*/
```

4. Using address expressions

```
.heap (. + 0x1000) : {                /* Heap VMA = Current Location + 4096 bytes */
```

```
    _heap_start = .;                    /* Record the start address of the heap */
    . = . + 0x1000;                     /* Reserve 4KB of heap space */
    _heap_end = .;                      /* Record the end address of the heap */
}
/*
 * Explanation:
 * - . (Dot): Represents the current location counter.
 * - 0x1000 = 4096 bytes = 4KB.
 * - Sets the heap start address by offsetting 0x1000 from the current location.
 * - Suitable for scenarios where section addresses are calculated dynamically.
 * - Commonly used for reserving space for the stack and heap.
 */

.data (ORIGIN(RAM) + 0x100) : {     /* .data section VMA = RAM start address + 256 bytes */
    *(.data*)                       /* Include content from all .data sections */
}
/*
 * Explanation:
 * - ORIGIN(RAM): Retrieves the start address of the RAM memory region.
 * - 0x100 = 256 bytes; reserves 256 bytes of space at the beginning of RAM.
 * - Commonly used to reserve space for the interrupt vector table or other special purposes.

*/
```

Common writing methods of LMA

The common writing methods of LMA are shown in ***Table 2-8. Common writing methods of LMA***.

Table 2-8. Common writing methods of LMA

```
1.  LMA = VMA
.text : {
    *(.text*)       /* LMA = VMA, code executes directly from Flash */
} > FLASH


2.  Using AT Syntax
/* Specific Address */
.data : AT(0x08010000) {    /* LMA = 0x08010000 (Specific address in FLASH) */
    *(.data*)               /* Includes content of all .data sections */
} > RAM                     /* VMA = RAM region (Accessed in RAM at runtime) */
/*
 * Explanation:
 * - Data is stored at address 0x08010000 in FLASH during compilation.
 * - The CPU accesses the RAM region address when the program is running.
```

```
 * - Startup code is required to copy the data from 0x08010000 to RAM.
 */
```

3. Simplified Syntax

```
/* Using Symbols */
.data : AT(_etext) {     /* LMA = Address of the _etext symbol (usually the end of the code section) */
    *(.data*)            /* Includes content of all .data sections */
} > RAM                  /* VMA = RAM region */

/* Using Expressions */
.data : AT(ADDR(.text) + SIZEOF(.text)) {        /* LMA = .text section address + .text section size
*/
    *(.data*)                                    /* Includes content of all .data sections */
} > RAM                                          /* VMA = RAM region */
/*
 * Explanation:
 * - ADDR(.text): Gets the VMA address of the .text section.
 * - SIZEOF(.text): Gets the size of the .text section.
 * - The calculation result is the address immediately following the .text section.
 * - Ensures the .data section follows the .text section tightly, with no gaps.
*/
```

4. Aligned LMA

```
.data : {                        /* Start of section content definition */
    *(.data*)                    /* Includes content of all .data sections */
} >RAM AT>FLASH                  /* VMA = RAM region, LMA = FLASH region */
/*
 * Explanation:
 * - >RAM: Specifies using the RAM memory region for VMA (runtime address).
 * - AT>FLASH: Specifies using the FLASH memory region for LMA (storage address).
 * - Concise and clear, avoids address calculations.
 * - Equivalent to: AT(>FLASH) { } > RAM
 * - The linker automatically allocates continuous addresses in the FLASH region.
*/

.data : AT(ALIGN(0x08008000, 4)) {   /* LMA aligned to the 4-byte boundary of 0x08008000 */
    *(.data*)                        /* Includes content of all .data sections */
} > RAM                              /* VMA = RAM region */
/*
 * Explanation:
 * - ALIGN(0x08008000, 4): Aligns 0x08008000 to a 4-byte boundary.
 * - If 0x08008000 is already 4-byte aligned, the address remains unchanged.
 * - If not aligned, it adjusts upwards to the nearest 4-byte boundary.
 * - Alignment helps improve memory access performance; some hardware requires specific address
```

```
alignment.
 * - Common alignment values: 4 bytes (32-bit), 8 bytes (64-bit).
 */
/* Alignment Examples Explanation */
/* ALIGN(0x08008001, 4) = 0x08008004   Aligned upwards */
/* ALIGN(0x08008000, 4) = 0x08008000   Already aligned, unchanged */
/* ALIGN(0x08008003, 8) = 0x08008008   Aligned upwards to 8-byte boundary */
```

Combined application of VMA and LMA

In practical engineering, we typically combine the aforementioned syntax to simultaneously control the alignment and addresses of both VMA and LMA. Below is a general section definition template, as shown in . For more applications, please refer to ***AN311 Scatter loading instructions of GD32 Embedded Builder based on GD32H7xx***.

Table 2-9. Common writing methods of LMA

```
.SECTION_NAME ALIGN(VMA_BASE_ADDR, ALIGN_SIZE) : AT(ALIGN(LMA_BASE_ADDR, ALIGN_SIZE))
{
    . = ALIGN(ALIGN_SIZE);                  /* Alignment */
    __SECTION_NAME_start__ = .;             /* Start Symbol */
    KEEP(*(.input_section_name))            /* Input Section */
    . = ALIGN(ALIGN_SIZE);                  /* End Alignment */
    __SECTION_NAME_end__ = .;               /* End Symbol */
    __SECTION_NAME_size__ = __SECTION_NAME_end__ - __SECTION_NAME_start__;
/* Size */
}
/* Parameter Description:
 * SECTION_NAME          : Output section name (e.g., .user_data)
 * VMA_BASE_ADDR         : VMA base address (Runtime address)
 * LMA_BASE_ADDR         : LMA base address (Load/Storage address)
 * ALIGN_SIZE            : Alignment size in bytes (e.g., 4, 8, 16)
 * input_section_name : Input section name (e.g., .data.user)
 */
```

The following introduces the section contents defined in GD32MCUs.

2.3.2. .vectors

In GD32MCU, a section named .vectors is first defined. The pair of curly braces “{” and “}” are used to define the contents of the section. The contents of the .vectors section are shown in ***Table 2-10. GD32MCU .vectors section***.

Table 2-10. GD32MCU .vectors section

```
/* ISR vectors */
.vectors :
  {
    . = ALIGN(4);
    KEEP(*(.vectors))
    . = ALIGN(4);
    __Vectors_End = .;
    __Vectors_Size = __Vectors_End - __gVectors;
  } >FLASH
```

This code defines a segment named .vectors for storing the interrupt vector table and other data, and allocates this segment to the FLASH memory.

`.= ALIGN(4);`: This line of code is an alignment directive that aligns the current location (.) to a 4-byte boundary by adding enough padding bytes after the current position until the address meets the 4-byte alignment requirement.

`KEEP(*(.vectors))`: This line of code uses the KEEP command to prevent the `.vectors` symbols from being optimized. The `*` indicates matching all symbols named `.vectors`. This statement ensures that all symbols named `.vectors` are retained in the final generated object file.

`. = ALIGN(4);`: Perform an alignment operation again to ensure the end position of the segment (.) meets the 4-byte alignment requirement.

`__Vectors_End = .;`: This line assigns the current location (.) to the `__Vectors_End` symbol, which is used to record the end position of the `.vectors` segment.

`__Vectors_Size = __Vectors_End - __gVectors;`: This line calculates the size of the `.vectors` segment and assigns the result to the `__Vectors_Size` symbol. The size is the difference between the end position of the `.vectors` segment (`__Vectors_End`) and another symbol named `__gVectors` (the symbol name must match the actual startup file).

`>FLASH`: This line indicates that the `.vectors` segment is allocated to the FLASH memory. The linker will place the content of this segment in the appropriate location within the FLASH memory.

2.3.3. .text

The contents of the .text section are shown in *Table 2-11. GD32MCU .text section*.

Table 2-11. GD32MCU .text section

```
.text :
{
    . = ALIGN(4);
    *(.text)
```

```
    *(.text*)
    *(.glue_7)
    *(.glue_7t)
    *(.eh_frame)

    KEEP (*(.init))
    KEEP (*(.fini))
    . = ALIGN(4);
    /* the symbol '_etext' will be defined at the end of code section */
    _etext = .;
} >FLASH
```

The `.text` section is used to store the program's code segment.

. = ALIGN(4);: Ensures the current address is 4-byte aligned.

`*(.text)`, `*(.text*)`, `*(.glue_7)`, `*(.glue_7t)`, `*(.eh_frame)`: Match different code sections and place them in the `.text` segment. Among them, `.text` and `.text*` match all standard code sections, while `.glue_7` and `.glue_7t` match Thumb-1 and Thumb-2 code.

`KEEP(*(.init))`, `KEEP(*(.fini))`: Retain the contents of the initialization and termination sections. These sections typically contain code that needs to be executed at the start and end of the program.

. = ALIGN(4);: Ensure the current address is 4-byte aligned again.

_etext = .;: defines the _etext symbol, indicating the end address of the code section.

>FLASH: Specifies that the contents of this section are placed in Flash memory.

KEEP(*(.init)) and KEEP(*(.fini)) are used in the linker script to ensure that specific code sections are retained during the linking process and are not optimized out. These sections typically contain functions or instructions required to execute during program startup and termination.

*(.init): This expression matches all code sections labeled as .init. In most cases, the .init section contains initialization code that needs to execute during program startup, such as global variable initialization and hardware configuration. This code is typically executed before the main() function.

*(.fini): This expression matches all code sections labeled as .fini. The .fini section contains cleanup code that needs to execute at program termination, such as releasing resources and closing files. This code is typically executed after the main() function returns.

By using the KEEP keyword in the linker script, the code in the .init and .fini sections will not be optimized out by the linker, even if they are not explicitly referenced elsewhere in the program.

2.3.4. .rodata

Content of the .rodata section is as follows *Table 2-12. GD32MCU .rodata section*.

Table 2-12. GD32MCU .rodata section

```
.rodata :
{
   . = ALIGN(4);
   *(.rodata)
   *(.rodata*)
   . = ALIGN(4);
} >FLASH
```

This section is used to store read-only data, typically constant data or strings in the program. Since this data is read-only, it is placed in Flash or other read-only memory.

. = ALIGN(4);: Ensures the current address is 4-byte aligned.

*(.rodata), *(.rodata*): Matches all sections labeled as .rodata and .rodata*, and places them into the .rodata section.

. = ALIGN(4);:Ensure the current address is 4-byte aligned again.

2.3.5. .ARM.xx

Content of the .ARM.xx section is as follows *Table 2-13. GD32MCU .ARM.extab section*.

Table 2-13. GD32MCU .ARM.extab section

```
ARM.extab :
{
   *(.ARM.extab* .gnu.linkonce.armextab.*)
} >FLASH

.ARM : {
   __exidx_start = .;
   *(.ARM.exidx*)
   __exidx_end = .;
} >FLASH

.ARM.attributes : { *(.ARM.attributes) } > FLASH
```

.ARM.extab: Used to store the ARM exception table, typically containing information related to exception handling.

(.ARM.extab .gnu.linkonce.armextab.*): Matches all sections labeled as .ARM.extab* or .gnu.linkonce.armextab.* and places them into the .ARM.extab section.

ARM is used to store the ARM exception table, usually containing instructions or data related to exception handling.

__exidx_start = .;: Defines the symbol __exidx_start, indicating the start address of the exception table.

(.ARM.exidx): Matches all content labeled as .ARM.exidx* and places them in the .ARM section.

__exidx_end = .;: Defines the symbol __exidx_end, indicating the end address of the exception table.

> FLASH: Specifies that the content of this section is placed in Flash memory.

.ARM.attributes is used to store ARM-specific attribute information, usually related to compiler and linker attributes.

{ *(.ARM.attributes) }: Matches all content labeled as .ARM.attributes and places them in the .ARM.attributes section.

> FLASH: Specifies that the content of this section is placed in Flash memory.

2.3.6. .xxx_array

Three sections, .preinit_array, .init_array, and .fini_array, are defined in the linker script to store the function lists to be executed before program execution, during the initialization phase, and at the termination phase.

Content of the .xxx_array section is as follows ***Table 2-14. GD32MCU . xxx_array section***.

Table 2-14. GD32MCU . xxx_array section

```
.preinit_array :
{
   PROVIDE_HIDDEN (__preinit_array_start = .);
   KEEP (*(.preinit_array*))
   PROVIDE_HIDDEN (__preinit_array_end = .);
} >FLASH
.init_array :
{
   PROVIDE_HIDDEN (__init_array_start = .);
   KEEP (*(SORT(.init_array.*)))
   KEEP (*(.init_array*))
   PROVIDE_HIDDEN (__init_array_end = .);
} >FLASH

.fini_array :
{
```

```
    PROVIDE_HIDDEN (__fini_array_start = .);
    KEEP (*(.fini_array*))
    KEEP (*(SORT(.fini_array.*)))
    PROVIDE_HIDDEN (__fini_array_end = .);
} >FLASH
```

.preinit_array

This section is used to store the function list executed before the program runs.

PROVIDE_HIDDEN (__preinit_array_start = .);: Defines a hidden symbol __preinit_array_start, indicating the start address of the .preinit_array section.

KEEP (*(.preinit_array*)): Retains all content labeled as .preinit_array, which typically includes initialization functions to be executed before the program runs.

PROVIDE_HIDDEN (__preinit_array_end = .);: Defines a hidden symbol __preinit_array_end, indicating the end address of the .preinit_array section.

> FLASH: Specifies that the content of this section is placed in Flash memory.

.init_array:

This section is used to store the list of functions executed during the program initialization phase.

PROVIDE_HIDDEN (__init_array_start = .);: Defines a hidden symbol __init_array_start, indicating the start address of the .init_array section.

KEEP (*(SORT(.init_array.*))): Retains all contents marked as .init_array.* and sorts them by name.

KEEP (*(.init_array*)): Retains all contents marked as .init_array, which typically include initialization functions that need to be executed during the program initialization phase.

PROVIDE_HIDDEN (__init_array_end = .);: Defines a hidden symbol __init_array_end, indicating the end address of the .init_array section.

> FLASH: Specifies that the content of this section is placed in Flash memory.

.fini_array

This section is used to store the list of functions executed during the program termination phase.

PROVIDE_HIDDEN (__fini_array_start = .);: Defines a hidden symbol __fini_array_start, indicating the start address of the .fini_array section.

KEEP (*(.fini_array*)): Retains all contents marked as .fini_array, which typically include

cleanup functions that need to be executed during the program termination phase.

KEEP (*(SORT(.fini_array.*))): Retains all contents marked as .fini_array.* and sorts them by name.

PROVIDE_HIDDEN (__fini_array_end = .);: Defines a hidden symbol __fini_array_end, indicating the end address of the .fini_array section.

> FLASH: Specifies that the content of this section is placed in Flash memory.

2.3.7. .data

Content of the .bss section is as follows ***Table 2-15. GD32MCU .data section***.

Table 2-15. GD32MCU .data section

```
/* provide some necessary symbols for startup file to initialize data */
_sidata = LOADADDR(.data);
.data:
{
    . = ALIGN(4);
    /* the symbol '_sdata' will be defined at the data section start */
    _sdata = .;
    *(.data)
    *(.data*)
    . = ALIGN(4);
    /* the symbol '_edata' will be defined at the data section end */
    _edata = .;
} >RAM AT> FLASH
```

This section is used to store initialized data segments. After the program is loaded into RAM, the content of the .data section is copied to the corresponding location in RAM.

_sidata = LOADADDR(.data);: Assigns the load address of the .data section to the _sidata symbol, which is typically used in the startup file for initializing data.

. = ALIGN(4);: Ensures the current address is 4-byte aligned.

_sdata = .;: Defines a symbol _sdata, indicating the start address of the initialized data segment.

*(.data), *(.data*): Matches all content labeled as .data or .data* and places them in the .data section.

Ensure the current address is 4-byte aligned again.

_edata = .;: Defines a symbol _edata, indicating the end address of the initialized data segment.

>RAM AT> FLASH: Specifies that the content of this section is placed in RAM but loaded from FLASH.

2.3.8. .bss

Content of the .bss section is as follows ***Table 2-16. GD32MCU .bss segment***.

Table 2-16. GD32MCU .bss segment

```
. = ALIGN(4);
.bss :
{
  /* the symbol '_sbss' will be defined at the bss section start */
  _sbss = .;
  __bss_start__ = _sbss;
  *(.bss)
  *(.bss*)
  *(COMMON)
  . = ALIGN(4);
  /* the symbol '_ebss' will be defined at the bss section end */
  _ebss = .;
  __bss_end__ = _ebss;
} >RAM
```

.bss is used to store uninitialized data segments. After the program is loaded into RAM, the content of the .bss section is initialized to zero.

. = ALIGN(4);: Ensures the current address is 4-byte aligned.

_sbss = .;: Define a symbol _sbss, indicating the start address of the BSS segment.

__bss_start__ = _sbss;: Define a hidden symbol __bss_start__, representing the start address of the BSS segment.

Match all contents labeled as .bss, .bss*, or COMMON and place them in the .bss section.

. = ALIGN(4); : Ensure the current address is 4-byte aligned again.

Define a symbol _ebss, indicating the end address of the BSS segment.

Define a hidden symbol __bss_end__, representing the end address of the BSS segment.

>RAM: Specify that the contents of this section are placed in RAM.

2.3.9. .heap_stack

Table 2-17. GD32MCU heap stack definition

```
__stack_size = DEFINED(__stack_size) ? __stack_size : 2K;
```

```
__heap_size = DEFINED(__heap_size) ? __heap_size : 1K;
```

This code is a conditional expression used to define symbols named __stack_size and __heap_size and initialize their values.

DEFINED(__stack_size): This is a macro check used to verify whether the symbol __stack_size has already been defined. If __stack_size is defined, it returns true (non-zero); otherwise, it returns false (zero).

?: Conditional operator, similar to the if-else structure, used to select different values based on the truth or falseness of a condition.

__stack_size : 2K: These are the two branches of the conditional expression. If __stack_size is already defined (condition is true), the value of __stack_size is selected as the expression's value; otherwise (condition is false), 2K (2KB) is selected as the expression's value.

If __stack_size is already defined, its original value is retained.

If __stack_size is not defined, initialize it to 2KB (2048 bytes).

__heap_size and __stack_size are treated the same way.

Content of the .heap_stack section is as follows ***Table 2-18. GD32MCU .heap_stack section***.

Table 2-18. GD32MCU .heap_stack section

```
/* heap and stack space */
.heap_stack :
{
. = ALIGN(8);
PROVIDE (end = _ebss);
PROVIDE (_end = _ebss);
. = . + __heap_size;
PROVIDE( _heap_end = . );
. = . + __stack_size;
PROVIDE( _sp = . );
. = ALIGN(8);
} > RAM
```

2.4. GROUP

The GROUP command combines multiple library files into a search group, and the linker repeatedly searches these libraries until all symbols are resolved. This is important for handling interdependencies between libraries, especially potential circular references between libgcc.a and libc.a.

Table 2-19. GD32MCU GROUP

```
/* input sections */
```

```
GROUP(libgcc.a libc.a libm.a libnosys.a)
```

In the provided GROUP statement, libgcc.a, libc.a, libm.a, and libnosys.a are four library files that respectively contain compiler runtime support, the C standard library, the math library, and system call-related functions. By placing these library files in a GROUP statement, the linker can handle them as a whole during processing.

3. Frequently Asked Questions (FAQ)

3.1. How to configure debugging starting from SRAM?

1. Modify the MEMORY allocation in the linker script, confirm the RAM size of the MCU, and divide the RAM into two sections. Taking the GD32F527xS chip as an example, the SRAM size is 512KB, and the MEMORY can be allocated as follows:

Table 3-1. MEMORY allocation

```
MEMORY
{
FLASH (rx)          : ORIGIN = 0x20000000, LENGTH = 256K
RAM (xrw)           : ORIGIN = 0x20040000, LENGTH = 256K
TCMRAM (xrw)        : ORIGIN = 0x10000000, LENGTH = 64K
}
```

The FLASH space is allocated as 256KB, starting at address 0x20000000, and the RAM space is allocated as 256KB, starting at address 0x20040000. Alternatively, it can be modified as needed.

2. In the development code, offset the interrupt vector table to address 0x20000000. Here, NVIC_VECTTAB_RAM defaults to the starting address of the chip's SRAM, and VECT_TAB_OFFSET is the offset of the starting address, which can be modified as needed. Based on the above example, the offset value here is 0x0.

Table 3-2. Interrupt vector table offset

```
nvic_vector_table_set(NVIC_VECTTAB_RAM, VECT_TAB_OFFSET);
```

3. According to the UM manual, modify the chip BOOT level to enable the chip to reset and start from SRAM. Taking the GD32F527xS chip as an example, set BOOT0 and BOOT1 to high level.

4. Recompile the project, and observe through the map file or DEBUG that the MCU successfully starts from the SRAM address.

3.2. How to add the TCMSRAM section?

TCMSRAM is tightly coupled memory SRAM, which can be used to store data. In the GD32F527xS chip, the SRAM size is 64KB. To use this RAM in user code, the following steps are required:

1. Define this memory section in the MEMORY of the linker file.

Table 3-3. MEMORY allocation

```
MEMORY
{
FLASH (rx)          : ORIGIN = 0x20000000, LENGTH = 256K
RAM (xrw)           : ORIGIN = 0x20040000, LENGTH = 256K
TCMRAM (xrw)      : ORIGIN = 0x10000000, LENGTH = 64K
}
```

2. Add the usage of the tcmsram section in the linker file as follows:

Table 3-4. tcmsram section

```
_sitcmram = LOADADDR(.tcmram);
.tcmram :
{
  . = ALIGN(4);
  _stcmram = .;          /* create a global symbol at tcmram start */
  *(.tcmram)
  *(.tcmram*)
  . = ALIGN(4);
  _etcmram = .;          /* create a global symbol at tcmram end */
} >TCMRAM AT> FLASH
```

3. Modify the startup file to initialize the data stored in tcmsram after the .bss initialization is completed.

Table 3-5. tcmsram initialization

```
Zerobss:
  ldr r3, = _ebss
  cmp r2, r3
  bcc FillZerobss
  movs r1, #0
  b Itcmsram_DataInit

Itcmsram_CopyData:
  ldr r3, =_sitcmram
  ldr r3, [r3, r1]
  str r3, [r0, r1]
  adds r1, r1, #4

Itcmsram_DataInit:
  ldr r0, =_stcmram
  ldr r3, = _etcmram
  adds r2, r0, r1
  cmp r2, r3
```

```
  bcc ltcmsram_CopyData

/* Call SystemInit function */
  bl   SystemInit
```

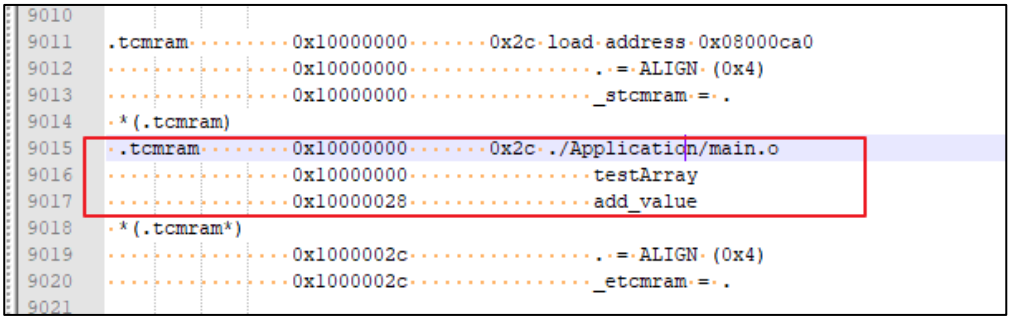
4. Add the definition of arrays or variables to the .tcmsram section in user code. The definition method is as follows:

Table 3-6. Variables and arrays dispersed into the .tcmsram section

```
__attribute__((section(".tcmram"))) uint32_t testArray[10] = {0,1,2,3,4,5,6,7,8,9};
__attribute__((section(".tcmram"))) uint32_t add_value = 0;
```

5. Use the defined arrays and variables in user code. Through the map file, you can observe that the arrays and variables are already stored in TCMSRAM.

Figure 3-1. Map file observation



3.3. How to configure the stack?

3.3.1. Heap memory configuration

Heap memory size configuration

Add the following information to the linker file, where __heap_size can define the heap size as 2KB.

Table 3-7. Stack allocation 1

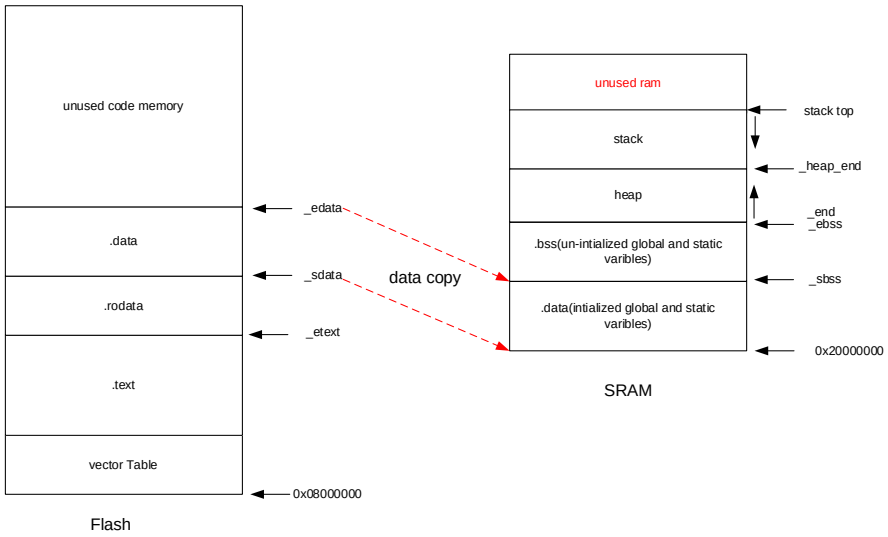
```
.bss :
{
/* the symbol '_sbss' will be defined at the bss section start */
_sbss = .;
__bss_start__ = _sbss;
*(.bss)
*(.bss*)
*(COMMON)
. = ALIGN(4);
```

```
/* the symbol '_ebss' will be defined at the bss section end */
_ebss = .;
__bss_end__ = _ebss;
} >RAM

/* heap and stack space */
.heap_stack :
{
. = ALIGN(8);
PROVIDE (end = _ebss);
PROVIDE (_end = _ebss);
. = . + __heap_size;
PROVIDE( _heap_end = . );
. = . + __stack_size;
PROVIDE( _sp = . );
. = ALIGN(8);
} > SRAM
```

At this point, the memory distribution of SRAM is as follows.

Figure 3-2. SRAM distribution diagram



Note: The starting address of the heap is defined at the position following the end of the .bss section, i.e., the _end symbol, and the address must be double-word aligned.

The stack base address must also be double-word aligned during allocation.

The symbols _etext, _sdata, _edata, _sbss, and _ebss defined in the linker file represent the boundary address information of each section. These symbols are used for the initialization

of .data and .bss in the startup file.

Manual configuration of heap and stack starting address

In the linker file, the manual configuration of the heap starting address is as follows:

Table 3-8. Stack allocation 2

```
__heap_size = DEFINED(__stack_size) ? __heap_size : 2K;
__stack_size = DEFINED(__stack_size) ? __stack_size : 2K;
.heap 0x20001000:
{
. = ALIGN(8);
_end = .;
. += __heap_size;
PROVIDE( _heap_end = . );
} >RAM
.stack 0x20002000:
{
. = ALIGN(8);
. += __stack_size;
PROVIDE( _sp = . );
} >RAM
```

The red section 0x20001000 is the manually specified heap starting address with a size of 2KB, and 0x20002000 is the manually specified stack starting address with a size of 2KB.

Note:

1. When manually setting the addresses of the heap and stack, please pay attention to memory allocation to avoid overlapping address ranges and unexpected errors.
2. Please ensure the starting addresses of the heap and stack are double-word aligned.

4. Revision history

Table 4-1. Revision history

Revision No.	Description	Date
1.0	Initial Release	Dec.03 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.