GigaDevice Semiconductor Inc.

适用于 Arm Cortex-M 处理器的 GD32 Embedded Builder 链接脚本文件说明

应用笔记

AN200

1.0 版本

（2025 年 12 月）

目录

图索引

表索引

1. 简介

ARM GCC（GNU Compiler Collection）是一套针对 ARM 架构的开源编译器工具，由 GNU 组织维护和开发。ARM GCC 包含了一系列工具，如编译器（arm-none-eabi-gcc）、汇编器（arm-none-eabi-as）、链接器（arm-none-eabi-ld）、调试器（gdb）等，用于开发基于 ARM 架构的嵌入式系统软件。

ARM LD（GNU linker）是一款 GNU 工具链中的链接器，用于将编译后的目标文件（包括可执行文件、库文件等）链接成最终的可执行文件或共享库。在 ARM 架构的嵌入式系统开发中，ARM LD 被广泛应用于生成针对 ARM 架构的目标文件。

GD32 MCU 涵盖 Arm Cortex® M3/M4/M23/M33/M7 内核产品，在开发过程中，可以使用基于 GCC 编译工具链的 Embedded builder IDE 进行软件开发，通过链接脚本文件来定义目标设备的内存布局和相关配置。本手册旨在帮助开发人员理解和使用 GD32MCU 在 Embedded builder 中的链接脚本，以及一些常见问题的配置。链接脚本是编译器和链接器在生成可执行文件时所依据的规则和配置文件，通过链接脚本，可以控制程序代码、数据的存放位置，定义内存布局等重要任务。

GD32MCU 链接脚本通常包括了以下几个部分：

- MEMORY：定义了目标设备的内存布局，包括 Flash 存储器和 RAM 的起始地址和大小。
- SECTIONS：定义了各个段（sections）的存放位置和属性，包括代码、只读数据、读写数据、未初始化数据等。
- ENTRY：指定了程序的入口地址，通常是 Reset_Handler 函数。
- GROUP：用于将多个库文件打包成一个逻辑库，在链接过程中一起处理。

本手册以 GD32MCU 通用链接文件为例进行分析，适用于 GD32 Arm Cortex® M3/M4/M23/M33/M7 内核 MCU 产品。

2. 链接脚本说明

2.1. MEMORY

在 GCC ld 文件中，MEMORY{}语法用于定义目标设备的内存布局。在 GD32MCU 开发中，通过该语法指定程序的代码和数据应该存储在目标设备的哪些内存区域中。

表 2-1. MEMORY{}语法的一般形式

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

其中，memory_region_name 是内存区域命名的标识符，origin_address 是该内存区域的起始地址，length 是该内存区域的长度。Attribute 定义内存区域的属性，如可读（readable）、可写（writable）、可执行（executable）等。常见的属性包括：

表 2-2. MEMORY 常见属性

- `rx`：可读可执行（Readable and Executable）。
- `xrw`：可执行可读可写（Executable, Readable and Writable）。
- `rw`：可读可写（Readable and Writable）。
- `x`：可执行（Executable）。

表 2-3. GD32MCU MEMORY 内存区域定义

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

在*表 2-3. GD32MCU MEMORY 内存区域定义*中，定义了两个内存区域：FLASH 和 RAM。FLASH 的起始地址是 0x08000000，长度是 1024K，RAM 的起始地址是 0x24000000，长度是 512K。FLASH 区域可读可执行，RAM 区域可读可写可执行。

2.2. ENTRY

在链接脚本中，ENTRY 命令用于指定程序的入口点（Entry Point），即程序执行的起始地址。具体来说，ENTRY 命令允许您指定一个符号作为程序的入口点，当程序被加载和执行时，控

制器将从这个符号开始执行。

ENTRY 命令的一般语法：

表 2-4. ENTRY 语法的一般形式

```
ENTRY(symbol)
```

其中，symbol 是一个在链接过程中定义的符号，通常是一个函数或标签的名称。在程序加载和执行时，控制器将从这个符号所代表的地址开始执行。通常，ENTRY 指令指定程序的启动文件中定义的 Reset 向量的地址，即程序启动的第一条指令。

表 2-5. GD32MCU ENTRY 指定函数入口点

```
ENTRY(Reset_Handler)
```

在 GD32MCU 中，Reset_Handler 是程序的入口函数，通过 ENTRY(Reset_Handler)命令将 Reset_Handler 函数指定为程序的入口点。当程序被加载和执行时，处理器将从 Reset_Handler 函数所代表的地址开始执行。

2.3. SECTIONS

在链接脚本中，SECTIONS 命令用于定义程序的各个段（Sections）的分配和排列方式。段是程序在内存中的逻辑单元，包括代码段、数据段、堆栈段等，SECTIONS 命令允许明确指定这些段在内存中的分布情况。

以下是 SECTIONS 命令的一般语法：

表 2-6. SECTIONS 语法的一般形式

```
SECTIONS {
        section1 : {
                /* section1 的内容 */
        }
        section2 : {
                /* section2 的内容 */
        }
        /* 更多段的定义 */
}
```

在 SECTIONS 命令中，可以定义多个段，包括代码段、数据段、只读段等，以及其他自定义的段，并为每个段指定相应的属性和内容。这些段的定义将直接影响生成的可执行文件或库文件的结构和功能。

2.3.1. VMA 和 LMA

在 GCC 链接脚本中，VMA 和 LMA 是两个核心概念，用于控制程序段在内存中的分布。

- VMA (Virtual Memory Address)即虚拟内存地址，是程序运行时段所在的内存地址，也称

为运行时地址。

- LMA (Load Memory Address) 即加载内存地址，是程序烧录时段存储的物理地址，也称为存储地址。

在大多数情况下，这两个地址是相同的（例如直接在 Flash 中运行的代码）。但在嵌入式系统中，它们经常不同。最典型的例子是初始化数据段（.data）：它必须存储在非易失性的 Flash 中（LMA），但在程序运行时必须被复制到可读写的 RAM 中（VMA）。

VMA 常见写法

常见的 VMA 写法如*表 2-7. VMA 常见写法*所示。

表 2-7. VMA 常见写法

1. 使用内存区域

```
.data : {
    *(.data*)                   /* 包含所有.data 相关的段内容 */
} > RAM                         /* VMA 指向 RAM 内存区域 */
/*
 * 说明：
 * - 使用预定义的内存区域名称
 * - 链接器自动从 RAM 区域的起始地址开始分配
 * - 如果 RAM 区域已被其他段占用，则从空闲位置开始
*/
.text : {
    *(.text*)                   /* 包含所有代码段内容 */
} > FLASH                       /* VMA 指向 FLASH 内存区域 */
/*
 * 说明：
 * - 代码段通常直接在 FLASH 中执行
 * - FLASH 区域通常具有只读和可执行属性
 * - VMA = LMA，无需数据拷贝
 */
```

2. 直接指定地址

```
.data 0x20000000 : {        /* 直接指定 VMA 为 0x20000000 */
    *(.data*)               /* 包含所有.data 段内容 */
}
/*
 * 说明：
 * - 指定段的起始地址
* - 适用于需要精确控制内存布局的场景
 * - 缺点：缺乏灵活性，地址冲突风险高
 * - 如果该地址已被占用，链接器会报错
 * - 注意：地址必须在有效的内存范围内
```

```
*/
3.  使用对齐函数
.data ALIGN(4) : {                          /* VMA 对齐到 4 字节边界 */
    *(.data*)
}
/*
 * 说明：
 * - ALIGN(4): 将当前位置对齐到 4 字节边界
* - 如果当前位置已对齐，地址不变，如果未对齐，向上调整到最近的 4 字节边界
 */
.data ALIGN(0x20000000, 8) : {          /* 指定地址的 8 字节对齐 */
    *(.data*)                           /* 包含所有.data 段内容 */
}
/*
 * 说明：
 * - ALIGN(0x20000000, 8): 将 0x20000000 对齐到 8 字节边界
 * - 第一个参数：基准地址，第二个参数：对齐字节数（必须是 2 的幂）
 * - 对齐示例：
 *    ALIGN(0x20000001, 8) = 0x20000008   (向上对齐)
 *    ALIGN(0x20000000, 8) = 0x20000000   (已对齐)
 * - 8 字节对齐适用于 64 位数据或 double 类型
*/
4.  使用地址表达式
.heap (. + 0x1000) : {                      /* 堆 VMA = 当前位置 + 4096 字节 */
    _heap_start = .;                        /* 记录堆的起始地址 */
    . = . + 0x1000;                         /* 预留 4KB 堆空间 */
    _heap_end = .;                          /* 记录堆的结束地址 */
}
/*
 * 说明：
 * - .(点): 表示当前的位置计数器
 * - 0x1000 = 4096 字节 = 4KB
 * - 在当前位置基础上偏移 0x1000 作为堆的起始地址
 * - 适用于动态计算段地址的场景
 * - 常用于栈和堆的空间预留
 */

.data (ORIGIN(RAM) + 0x100) : {     /* .data 段 VMA = RAM 起始地址 + 256 字节 */
    *(.data*)                           /* 包含所有.data 段内容 */
}
/*
 * 说明：
```

```
 * - ORIGIN(RAM): 获取 RAM 内存区域的起始地址
 * - 0x100 = 256 字节，在 RAM 起始位置预留 256 字节空间
 * - 常用于为中断向量表或其他特殊用途预留空间
*/
```

LMA 常见写法

常见的 LMA 写法如*表 2-8. LMA 常见写法*所示。

表 2-8. LMA 常见写法

```
1.  LMA = VMA
.text : {
    *(.text*)          /* LMA = VMA，代码直接在 Flash 执行 */
} > FLASH
2.  使用 AT 语法
/* 具体地址 */
.data : AT(0x08010000) {      /* LMA = 0x08010000（FLASH 中的具体地址） */
    *(.data*)                 /* 包含所有.data 段的内容 */
} > RAM                       /* VMA = RAM 区域（运行时在 RAM 中访问） */
/*
 * 说明：
 * - 数据在编译时存储到 FLASH 的 0x08010000 地址
 * - 程序运行时 CPU 访问 RAM 区域的地址
 * - 需要启动代码将数据从 0x08010000 复制到 RAM
 */
/* 使用符号 */
.data : AT(_etext) {     /* LMA = _etext 符号的地址（通常是代码段结束位置） */
    *(.data*)            /* 包含所有.data 段的内容 */
} > RAM                  /* VMA = RAM 区域 */

/* 使用表达式 */
.data : AT(ADDR(.text) + SIZEOF(.text)) {        /* LMA = .text 段地址 + .text 段大小 */
    *(.data*)                                    /* 包含所有.data 段的内容 */
} > RAM                                          /* VMA = RAM 区域 */
/*
 * 说明：
 * - ADDR(.text): 获取.text 段的 VMA 地址
 * - SIZEOF(.text): 获取.text 段的大小
 * - 计算结果是.text 段结束后的地址
 * - 确保.data 段紧跟在.text 段之后，无空隙
*/
3.  简化写法
.data : {                        /* 段内容定义开始 */
```

```
    *(.data*)                   /* 包含所有.data 段的内容 */
} >RAM AT>FLASH                 /* VMA=RAM 区域, LMA=FLASH 区域 */
/*
 * 说明：
 * - >RAM: 指定 VMA 使用 RAM 内存区域（运行时地址）
 * - AT>FLASH: 指定 LMA 使用 FLASH 内存区域（存储地址）
* - 简洁明了，避免了地址计算
 * - 等价于: AT(>FLASH) { } > RAM
 * - 链接器自动在 FLASH 区域分配连续地址
*/
4.  对齐的 LMA
.data : AT(ALIGN(0x08008000, 4)) {   /* LMA 对齐到 0x08008000 的 4 字节边界 */
    *(.data*)                       /* 包含所有.data 段的内容 */
} > RAM                             /* VMA = RAM 区域 */
/*
 * 说明：
 * - ALIGN(0x08008000, 4): 将 0x08008000 对齐到 4 字节边界
 * - 如果 0x08008000 已经 4 字节对齐，则地址不变
 * - 如果不对齐，则向上调整到最近的 4 字节边界
 * - 对齐有助于提高内存访问性能，某些硬件要求特定的地址对齐
 * - 常用对齐值：4 字节(32 位)、8 字节(64 位)
 */
/* 对齐示例解释 */
/* ALIGN(0x08008001, 4) = 0x08008004   向上对齐 */
/* ALIGN(0x08008000, 4) = 0x08008000   已对齐，不变 */
/* ALIGN(0x08008003, 8) = 0x08008008   向上对齐到 8 字节边界 */
```

VMA 与 LMA 组合应用

在实际的工程中，我们通常会结合使用上述语法，同时控制 VMA 和 LMA 的对齐与地址。以下是一个通用的段定义模板，如*表 2-9. VMA 与 LMA 组合写法*所示。更多应用可参考*《AN311 GD32H7xx Embedded Builder 分散加载说明》*。

表 2-9. VMA 与 LMA 组合写法

```
.SECTION_NAME ALIGN(VMA_BASE_ADDR, ALIGN_SIZE) : AT(ALIGN(LMA_BASE_ADDR, ALIGN_SIZE))
{
    . = ALIGN(ALIGN_SIZE);              /* 对齐 */
    __SECTION_NAME_start__ = .;         /* 起始符号 */
    KEEP(*(.input_section_name))        /* 输入段 */
    . = ALIGN(ALIGN_SIZE);              /* 结束对齐 */
    __SECTION_NAME_end__ = .;           /* 结束符号 */
    __SECTION_NAME_size__ = __SECTION_NAME_end__ - __SECTION_NAME_start__;
```

```
                                                    /* 大小 */
}
/* 参数说明：
 * SECTION_NAME          : 输出段名称（如 .user_data）
 * VMA_BASE_ADDR         : VMA 基准地址（运行时地址）
 * LMA_BASE_ADDR         : LMA 基准地址（加载/存储地址）
 * ALIGN_SIZE            :  对齐字节数（如 4, 8, 16）
 * input_section_name :    输入段名称（如 .data.user）
 */
```

下面介绍在 GD32MCU 定义的段内容。

2.3.2. .verctors

在 GD32MCU 中首先定义了. 定义一个名为.vectors 的段，“{” 和 “}” 这对大括号用于定义段的内容。.vectors 段内容如*表 2-10. GD32MCU .vectors 段*。

表 2-10. GD32MCU .vectors 段

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

这段代码定义了一个名为.vectors 的段，用于存放向量表（interrupt vector table）等内容，并将该段分配到 FLASH 存储器中。

. = ALIGN(4);：这行代码是一个对齐指令，它将当前位置（.）对齐到 4 字节边界。在当前位置后面添加足够的填充字节，直到地址满足 4 字节对齐的要求。

KEEP(*(.vectors))：这行代码使用 KEEP 命令保持.vectors 符号不被优化。使用*表示匹配所有名称为 .vectors 的符号。这该语句确保所有名称为.vectors 的符号都被保留在最终生成的目标文件中。

. = ALIGN(4);：再次进行对齐操作，确保段结束位置（.）满足 4 字节对齐的要求。

__Vectors_End = .;：这行代码将当前位置(.)的值赋给__Vectors_End 符号，用于记录 .vectors 段的结束位置。

__Vectors_Size = __Vectors_End - __gVectors;：这行代码计算.vectors 段的大小，并将结果赋给 __Vectors_Size 符号。大小是 .vectors 段结束位置和另一个名为__gVectors 的符号(符号名称需要与实际启动文件保持一致）值之间的差值。

>FLASH：这行代码表示将 .vectors 段分配到 FLASH 存储器中。链接器会将该段的内容放置在 FLASH 存储器的适当位置。

2.3.3. .text

.text 段内容如*表 2-11. GD32MCU .text 段*。

表 2-11. GD32MCU .text 段

```
.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
```

.text 段用于存放程序的代码段。

. = ALIGN(4);：确保当前地址是 4 字节对齐的。

(.text)、(.text*)、*(.glue_7)、*(.glue_7t)、*(.eh_frame)：匹配不同的代码段，将它们放置在.text 段中。其中，.text 和.text*匹配所有标准的代码段，.glue_7 和.glue_7t 匹配 Thumb-1 和 Thumb-2 的代码。

KEEP (*(.init))、KEEP (*(.fini))：保留初始化和终止段中的内容。这些段通常包含了程序启动时和结束时需要执行的代码。

. = ALIGN(4);：再次确保当前地址是 4 字节对齐的。

_etext = .;：定义 _etext 符号，表示代码段的结束地址。

>FLASH：指定这个段的内容被放置在 Flash 存储器中

KEEP (*(.init))和 KEEP (*(.fini))这两个语句在链接脚本中用于确保特定的代码段在链接过程中被保留，不会被优化掉。这些代码段通常包含一些在程序启动和终止时需要执行的特定函数或指令。

*(.init)：这个表达式匹配所有标记为.init 的代码段。在大多数情况下，.init 段中包含了在程序启动时需要执行的一些初始化代码，例如初始化全局变量、配置硬件等。这些代码通常在 main()

函数之前执行。

*(.fini)：这个表达式匹配所有标记为.fini 的代码段。.fini 段中包含了在程序终止时需要执行的一些清理代码，例如释放资源、关闭文件等。这些代码通常在 main()函数返回之后执行。

通过在链接脚本中使用 KEEP 关键字，这些.init 和.fini 段中的代码将不会被链接器优化掉，即使在程序的其他地方没有显式地引用它们。

2.3.4. .rodata

.rodata 段内容如*表 2-12. GD32MCU .rodata 段*。

表 2-12. GD32MCU .rodata 段

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

这个段用于存放只读数据（read-only data），通常是程序中的常量数据或字符串。因为这些数据是只读的，所以被放置在 Flash 等只读存储器中。

. = ALIGN(4);：确保当前地址是 4 字节对齐的。

(.rodata)、(.rodata*)：匹配所有标记为.rodata 和.rodata*的只读数据段，并将它们放置在.rodata 段中。

. = ALIGN(4);：再次确保当前地址是 4 字节对齐的。

2.3.5. .ARM.xx

.ARM.extab 段内容如*表 2-13. GD32MCU . ARM.extab 段*。

表 2-13. GD32MCU . ARM.extab 段

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

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

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

.ARM.extab 用于存放 ARM 异常表（exception table），通常是一些与异常处理相关的信息。

(.ARM.extab .gnu.linkonce.armextab.*) ：匹配所有标记为.ARM.extab*或 .gnu.linkonce.armextab.*的内容，并将它们放置在.ARM.extab 段中。

.ARM 用于存放 ARM 异常表（exception table），通常包含了一些与异常处理相关的指令或数据。

__exidx_start = .;：定义符号__exidx_start，表示异常表的起始地址。

(.ARM.exidx)：匹配所有标记为.ARM.exidx*的内容，并将它们放置在.ARM 段中。

__exidx_end = .;：定义符号 __exidx_end，表示异常表的结束地址。

>FLASH：指定这个段的内容被放置在 Flash 存储器中。

.ARM.attributes 用于存放 ARM 特有的属性信息，通常是一些与编译器、链接器相关的属性。

{ *(.ARM.attributes) }：匹配所有标记为.ARM.attributes 的内容，并将它们放置在.ARM.attributes 段中。

> FLASH：指定这个段的内容被放置在 Flash 存储器中

2.3.6. .xxx_array

链接脚本中定义了三个段，.preinit_array、.init_array 和.fini_array，这些段用于存放在程序运行前、初始化阶段和结束阶段需要执行的函数列表。

.xxx_array 段内容如*表 2-14. GD32MCU . xxx_array 段*。

表 2-14. GD32MCU . xxx_array 段

```
.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

这个段用于存放在程序运行前执行的函数列表。

PROVIDE_HIDDEN(__preinit_array_start = .);：定义了一个隐藏符号 __preinit_array_start，表示.preinit_array 段的起始地址。

KEEP (*(.preinit_array*))：保留所有标记为.preinit_array 的内容，这些内容通常包含了一些在程序运行前需要执行的初始化函数。

PROVIDE_HIDDEN (__preinit_array_end = .);：定义了一个隐藏符号 __preinit_array_end，表示.preinit_array 段的结束地址。

>FLASH：指定这个段的内容被放置在 Flash 存储器中。

.init_array

这个段用于存放在程序初始化阶段执行的函数列表。

PROVIDE_HIDDEN (__init_array_start = .);：定义了一个隐藏符号 __init_array_start，表示.init_array 段的起始地址。

KEEP (*(SORT(.init_array.*)))：保留所有标记为 .init_array.*的内容，并按照它们的名称进行排序。

KEEP (*(.init_array*))：保留所有标记为.init_array 的内容，这些内容通常包含了一些在程序初始化阶段需要执行的初始化函数。

PROVIDE_HIDDEN(__init_array_end = .);：定义了一个隐藏符号__init_array_end，表示.init_array 段的结束地址。

>FLASH：指定这个段的内容被放置在 Flash 存储器中。

.fini_array

这个段用于存放在程序结束阶段执行的函数列表。

PROVIDE_HIDDEN (__fini_array_start = .);：定义了一个隐藏符号__fini_array_start，表示.fini_array 段的起始地址。

KEEP (*(.fini_array*))：保留所有标记为.fini_array 的内容，这些内容通常包含了一些在程序结束阶段需要执行的清理函数。

KEEP (*(SORT(.fini_array.*)))：保留所有标记为.fini_array.*的内容，并按照它们的名称进行排序。

PROVIDE_HIDDEN (__fini_array_end = .);：定义了一个隐藏符号__fini_array_end，表示.fini_array 段的结束地址。

>FLASH：指定这个段的内容被放置在 Flash 存储器中。

2.3.7. .data

.data 段内容如*表 2-14. GD32MCU . xxx_array 段*。

表 2-15. GD32MCU . data 段

```
/* 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 end start */
  _sdata = .;
  *(.data)
  *(.data*)
  . = ALIGN(4);
  /* the symbol '_edata' will be defined at the data section end */
  _edata = .;
} >RAM AT> FLASH
```

这个段用于存放初始化的数据段，在程序加载到 RAM 中后，.data 段中的内容会被复制到 RAM 中相应的位置。

_sidata = LOADADDR(.data);：将.data 段的加载地址赋值给_sidata 符号，这个符号通常用于启动文件中初始化数据的处理。

. = ALIGN(4);：确保当前地址是 4 字节对齐的。

_sdata = .;：定义了一个符号 _sdata，表示初始化数据段的起始地址。

(.data)、(.data*)：匹配所有标记为.data 或.data*的内容，并将它们放置在.data 段中。

. = ALIGN(4);：再次确保当前地址是 4 字节对齐的。

_edata = .;：定义了一个符号 _edata，表示初始化数据段的结束地址。

>RAM AT> FLASH：指定这个段的内容被放置在 RAM 中，但是从 FLASH 中加载。

2.3.8. .bss

.bss 段内容如*表 2-16. GD32MCU . bss 段*。

表 2-16. GD32MCU . bss 段

```
. = 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 用于存放未初始化的数据段，在程序加载到 RAM 中后，.bss 段中的内容会被初始化为零。

. = ALIGN(4);：确保当前地址是 4 字节对齐的。

_sbss = .;：定义了一个符号 _sbss，表示 BSS 段的起始地址。

__bss_start__ = _sbss;：为 BSS 段的起始地址定义了一个隐藏符号__bss_start__。

(.bss)、(.bss*)、*(COMMON)：匹配所有标记为 .bss、.bss*或 COMMON 的内容，并将它们放置在.bss 段中。

. = ALIGN(4);：再次确保当前地址是 4 字节对齐的。

_ebss = .;：定义了一个符号_ebss，表示 BSS 段的结束地址。

__bss_end__ = _ebss;：为 BSS 段的结束地址定义了一个隐藏符号__bss_end__。

>RAM：指定这个段的内容被放置在 RAM 中。

2.3.9. .heap_stack

表 2-17. GD32MCU 堆栈定义

```
__stack_size = DEFINED(__stack_size) ? __stack_size : 2K;
__heap_size = DEFINED(__heap_size) ? __heap_size : 1K;
```

这段代码是一个条件表达式，用于定义名为__stack_size 和__heap_size 的符号，并初始化它的值。

DEFINED(__stack_size)：这是一个宏判断，用于检查符号__stack_size 是否已经被定义过。如果__stack_size 已经被定义过，则返回 true（非零值）；否则返回 false（零值）。

?：条件运算符，类似于 if-else 结构，用于根据条件的真假来选择不同的值。

__stack_size : 2K：这是条件表达式的两个分支。如果__stack_size 已经被定义过（条件为真），则选择__stack_size 的值作为表达式的值；否则（条件为假），选择 2K（2KB）作为表达式的值。

如果__stack_size 已经被定义过，则保留其原有的值；

如果__stack_size 没有被定义过，则将其初始化为 2KB（2048 字节）。

__heap_size 和__stack_size 同理。

.heap_stack 段内容如*表 2-18. GD32MCU .heap_stack 段*。

表 2-18. GD32MCU .heap_stack 段

```
/* 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

GROUP 命令将多个库文件组成一个搜索组，链接器会反复搜索这些库，直到所有符号都被解析。这对于解决库之间的相互依赖很重要，特别是 libgcc.a 和 libc.a 之间可能存在的循环引用。

表 2-19. GD32MCU GROUP

```
/* input sections */
GROUP(libgcc.a libc.a libm.a libnosys.a)
```

提供的 GROUP 语句中，libgcc.a、libc.a、libm.a 和 libnosys.a 是四个库文件，它们分别包含了编译器运行时支持、C 标准库、数学库以及系统调用相关的函数。通过将这些库文件放在一个 GROUP 语句中，可以使链接器在处理时将它们作为一个整体来处理。

3. 常见问题

3.1. 如何配置从 SRAM 启动调试

1. 修改链接文件中 MEMORY 内存分配空间，确认芯片 RAM 大小，将 RAM 空间划分为两块，以 GD32F527xS 芯片为例，芯片 SRAM 大小为 512KB，那么可以将 MEMORY 分配如下：

表 3-1. MEMORY 分配

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

其中 FLASH 空间分配为 256KB，起始地址为 0x20000000，RAM 空间分配为 256KB，起始地址为 0x20040000，或者根据实际需求进行修改即可。

2. 在开发的代码中将中断向量表偏移到 0x20000000 地址。其中 NVIC_VECTTAB_RAM 默认为芯片 SRAM 起始地址，VECT_TAB_OFFSET 为起始地址的偏移地址，可根据需求进行修改，根据上面例子，这里偏移值为 0x0。

表 3-2. 中断向量表偏移

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

3. 根据 UM 手册，更改芯片 BOOT 电平，让芯片复位从 SRAM 启动。以 GD32F527xS 芯片为例，将 BOOT0 和 BOOT1 选择为高电平。

4. 重新编译工程，通过 map 文件或者进入 DEBUG 可以观察到 MCU 成功从 SRAM 地址启动。

3.2. 如何添加 TCMSRAM 段

TCMSRAM 是紧耦合存储器 SRAM，可以用来存放数据，在 GD32F527xS 芯片中，该 SRAM 大小为 64KB，用户代码中要使用这段 RAM，需要如下操作：

1. 在链接文件 MEMORY 中定义这段内存。

表 3-3. MEMORY 分配

```
MEMORY
{
    FLASH (rx)        : ORIGIN = 0x20000000, LENGTH = 256K
```

```
    RAM (xrw)       : ORIGIN = 0x20040000, LENGTH = 256K
    TCMRAM (xrw)    : ORIGIN = 0x10000000, LENGTH = 64K
}
```

2. 在链接文件中添加 tcmsram 该段的使用，具体如下：

表 3-4. tcmsram 段

```
_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. 修改启动文件，在启动文件中.bss 初始化完成后，添加对存放在 tcmsram 中的数据进行初始化。

表 3-5. tcmsram 初始化

```
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 Itcmsram_CopyData

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

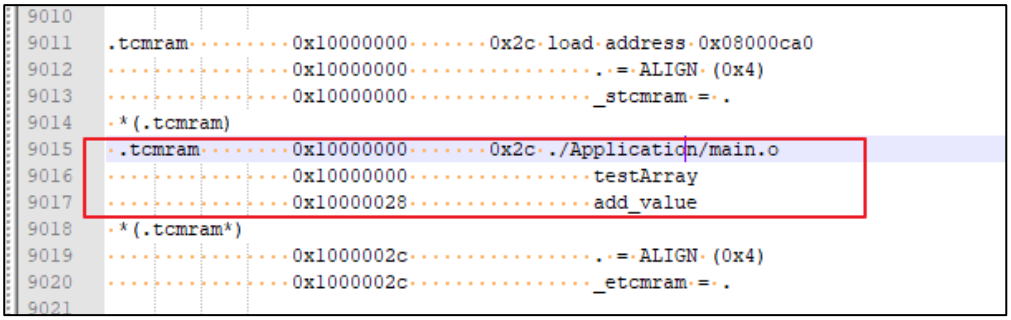
4. 在用户代码中添加定义数组或者变量到.tcmram 段，定义方法如下：

表 3-6. 变量和数组分散加载到.tcmsram 段

```
__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. 在用户代码中使用上面定义的数组和变量，通过 map 文件可以观察到数组和变量已经存放到 TCMSRAM 中。

图 3-1. Map 文件查看



3.3. 如何配置堆栈

3.3.1. 堆内存配置

堆内存大小配置

在链接文件中添加如下信息，其中__heap_size 可以定义堆大小为 2KB。

表 3-7. 堆栈分配 1

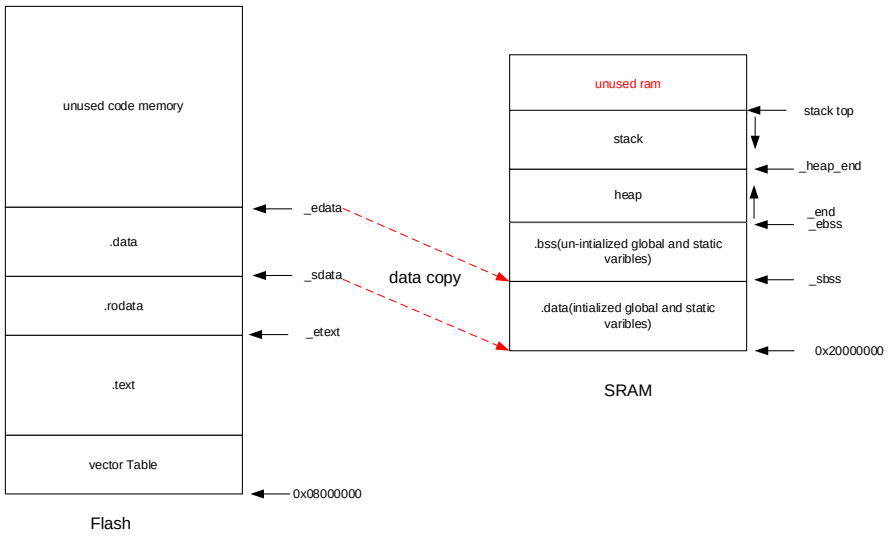
```
.bss :
{
  /* the symbol '_sbss' will be defined at the bss section start */
  _sbss = .;
  __bss_start__ = _sbss;
  *(.bss)
  *(.bss*)
  *(COMMON)
  . = ALIGN(4);
  /* the symbol '_ebss' will be defined at the bss section end */
  _ebss = .;
  __bss_end__ = _ebss;
} >RAM

/* 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
```

此时，SRAM 的内存分布如下所示。

图 3-2. SRAM 分布图



注意：在.bss 段结束后位置定义为堆的起始地址，即_end 符号，地址需要双字对齐。

栈底地址在分配时也需要双字对齐。

在链接文件中定义的_etext、_sdata、_edata、_sbss、_ebss 这些符号为各个段的边界地址信息，在启动文件中对.data 和.bss 的初始化，会使用到这些符号。

堆栈起始地址手动配置

在链接文件中堆起始地址手动配置如下：

表 3-8. 堆栈分配 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
```

其中红色部分 0x20001000 为手动指定的堆起始地址，大小为 2KB，0x20002000 为手动指定的堆起始地址，大小为 2KB。

注意：

1. 堆和栈的地址手动设置时，请注意内存分配，避免地址段重合，发生未知错误。
2. 堆栈的起始地址请注意双字对齐。

4. 版本历史

表 4-1. 版本历史

版本号	说明	日期
1.0	首次发布	2025 年 12 月 3 日

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.