使用分散加载描述文件

本章介绍如何将 ARM® 链接器 armlink 与分散加载描述文件配合使用以创建复杂映像。本章分为以下几节：

5.1 关于分散加载

映像由区和输出节组成。映像中的每个区可以包含不同的加载和执行地址。有关详细信息，请参阅第3-2 页的*指定映像结构*。

要构建映像的内存映射，链接器必须具有：

- 描述如何将输入节划分到输出节和区中的分组信息
- 描述区位于内存映射中的地址的位置信息

通过使用分散加载机制，您可以使用文本文件中的描述为链接器指定映像的内存映射。分散加载为您提供了对映像组件分组和位置的全面控制。分散加载可以用于简单映像，但它通常仅用于具有复杂内存映射的映像，即多个区在加载和执行时分散在内存映射中。

5.1.1 为分散加载定义的符号

当链接器使用分散加载描述文件创建映像时，它会创建一些与区相关的符号。第4-3 页的*与区相关的符号*对这些符号进行了介绍。仅当代码引用这些特殊符号时，链接器才会创建它们。

未定义的符号

请注意，在使用分散加载描述文件时，*不会* 定义以下符号：

- `Image$$RW$$Base`
- `Image$$RW$$Limit`
- `Image$$RO$$Base`
- `Image$$RO$$Limit`
- `Image$$ZI$$Base`
- `Image$$ZI$$Limit`

有关详细信息，请参阅第4-3 页的*访问链接器定义的符号*。

如果使用分散加载描述文件，但没有指定任何特殊区名称，也没有重新实现 `__user_initial_stackheap()`，则库会生成错误消息。

有关详细信息，请参阅：

- 《库和浮点支持指南》中第2-67 页的*调整运行时内存模型*
- 《开发指南》中第3-13 页的*放置堆栈和堆*

5.1.2 使用分散加载描述文件指定堆栈和堆

ARM C 库提供了 __user_initial_stackheap() 函数的多个实现，可以根据分散加载描述文件中给出的信息自动选择正确的函数实现。

要选择两个区内存模型，请在分散加载描述文件中定义两个名为 ARM_LIB_HEAP 和 ARM_LIB_STACK 的特殊执行区。两个区均具有 EMPTY 属性。这导致库选择使用以下符号值的非缺省 __user_initial_stackheap() 实现：

- Image$$ARM_LIB_STACK$$Base
- Image$$ARM_LIB_STACK$$ZI$$Limit
- Image$$ARM_LIB_HEAP$$Base
- Image$$ARM_LIB_HEAP$$ZI$$Limit

只能指定一个 ARM_LIB_STACK 或 ARM_LIB_HEAP 区，并且必须分配大小，例如：

```
ARM_LIB_HEAP 0x20100000 EMPTY 0x100000-0x8000  ; Heap starts at 1MB
                                               ; and grows upwards
ARM_LIB_STACK 0x20200000 EMPTY -0x8000         ; Stack space starts at the end
                                               ; of the 2MB of RAM
                                               ; And grows downwards for 32KB
```

注意

如果使用上面的堆栈函数，则还必须在汇编源代码中包含一个 IMPORT __use_two_region_memory，或在 C/C++ 源代码中包含一个 #pragma import(__use_two_region_memory)，因为不会自动选择双区模型。

可以使用 EMPTY 属性定义单个名为 ARM_LIB_STACKHEAP 的执行区，强制 __user_initial_stackheap() 使用合并的堆栈/堆区。这导致 __user_initial_stackheap() 使用符号 Image$$ARM_LIB_STACKHEAP$$Base 和 Image$$ARM_LIB_STACKHEAP$$ZI$$Limit 的值。

注意

如果重新实现 __user_initial_stackheap()，这将覆盖所有库实现。

5.1.3 何时使用分散加载

链接器的命令行选项提供了一些对数据和代码位置的控制，但要对位置进行全面控制，则需要使用比命令行中的输入内容更详细的指令。需要或最好使用分散加载描述的情况包括：

复杂内存映射

如果必须将代码和数据放在多个不同的内存区域中，则需要使用详细指令指定将哪个节放在哪个内存空间中。

不同类型的内存

许多系统都包含多种不同的物理内存设备，如闪存、 ROM、 SDRAM 和快速 SRAM。分散加载描述可以将代码和数据与最适合的内存类型相匹配。例如，可以将中断代码放在快速 SRAM 中以缩短中断响应时间，而将不经常使用的配置信息放在较慢的闪存中。

内存映射的外围设备

分散加载描述可以将数据节准确放在内存映射中的某个地址，以便能够访问内存映射的外围设备。

位于固定位置的函数

可以将函数放在内存中的相同位置，即使已修改并重新编译周围的应用程序。这有助于实现跳转表。

使用符号标识堆和堆栈

链接应用程序时，可以为堆和堆栈位置定义一些符号。

因此，几乎总是需要使用分散加载来实现嵌入式系统，因为这些系统使用 ROM、 RAM 和内存映射的外围设备。

——注意——

如果针对 Cortex™-M3 处理器进行编译，则会包括固定的内存映射，因此可以使用分散加载描述文件来定义堆栈和堆。在示例目录 *install_directory*\RVDS\Examples\..\Cortex-M3\Example3 中提供了这样一个示例。

5.1.4 分散加载命令行选项

用于分散加载的 armlink 命令行选项为：

`--scatter=`*`description_file`*

它指示链接器按照 *description_file* 中的描述构建映像内存映射。在《链接器参考指南》的第 3 章 *分散加载描述文件的形式语法*中介绍了描述文件的格式。

有关分散加载描述文件的其他信息，请参阅：

- 第5-9 页的*指定区和节地址的示例*
- 第5-31 页的*简单映像的等效分散加载描述*

5.1.5 具有简单内存映射的映像

图 5-1 中的分散加载描述将对象文件中的段加载到内存中，它与第5-6 页的图 5-2 中所示的映射相对应。区的最大大小设置是可选的，但是如果包含这些设置，则使链接器能够检查区是否没有溢出其边界。

在此示例中，可通过将 `--ro-base 0x0` 和 `--rw-base 0x10000` 指定为链接器的命令行选项来获得相同的结果。

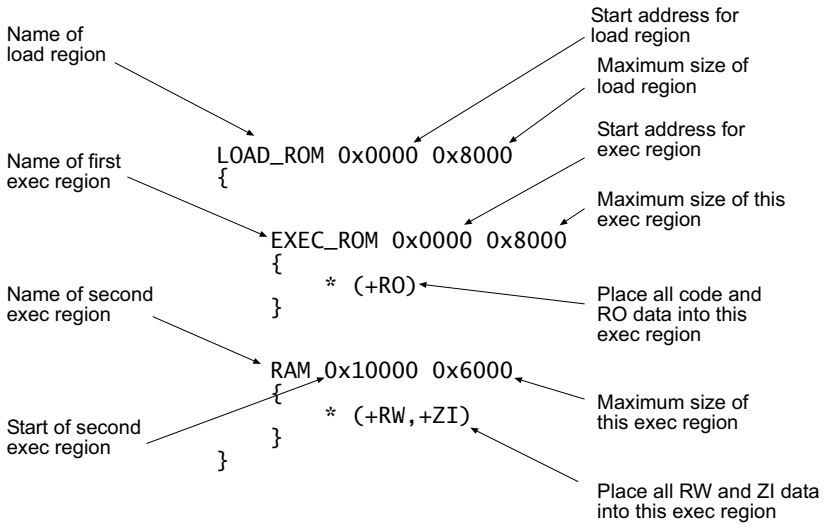

图 5-1 分散加载描述文件中的简单内存映射

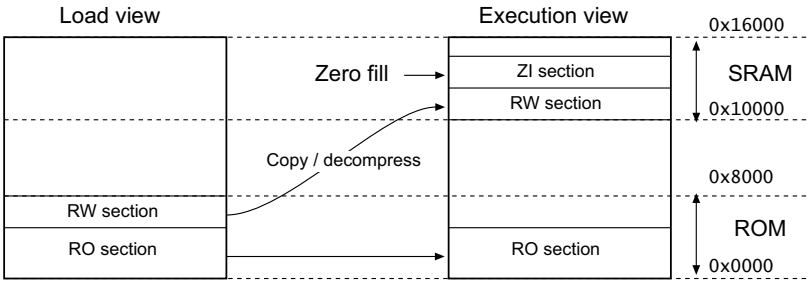

图 5-2 简单分散加载内存映射

5.1.6 具有复杂内存映射的映像

图 5-3 中的分散加载描述将 program1.o 和 program2.o 文件中的段加载到内存中，它与第5-8 页的图 5-4 中所示的映射相对应。

与第5-6 页的图 5-2 中所示的简单内存映射不同，不能只通过使用基本命令行选项为链接器指定此应用程序。

小心

图 5-3 中的分散加载描述仅指定 program1.o 和 program2.o 的代码和数据位置。如果链接其他模块 （如 program3.o）并使用此描述文件，则不会指定 program3.o 的代码和数据位置。

除非对代码和数据位置的要求非常严格，否则建议使用 * 或 .ANY 说明符来放置其余代码和数据。有关详细信息，请参阅第5-11 页的*将区放在固定地址中*。

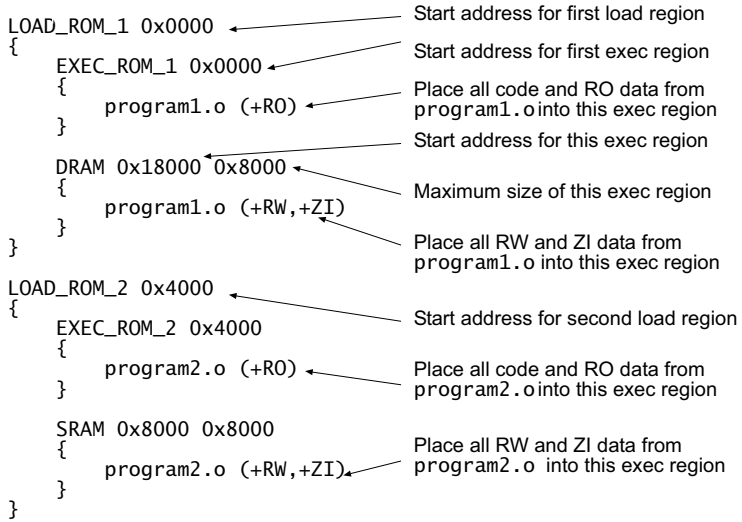

图 5-3 分散加载描述文件中的复杂内存映射

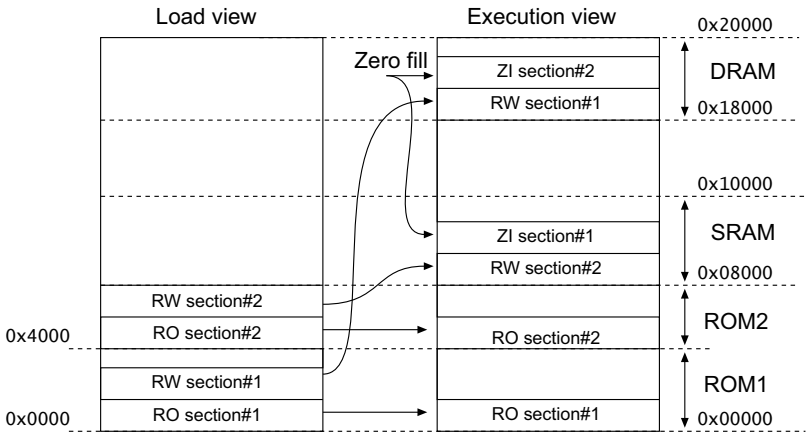

图 5-4 复杂分散加载内存映射

5.2 指定区和节地址的示例

本节介绍输入和执行节、区和预处理指令。有关访问位于固定地址中的数据和函数的示例，请参阅 《开发指南》中的第 3 章 *嵌入式软件开发*。

5.2.1 在分散加载描述中选择中间代码输入节

中间代码用于在 ARM 和 Thumb 代码之间切换，或者执行比一条指令中指定的跳转范围更大的程序跳转。请参阅第3-17 页的*中间代码*。可以使用分散加载描述文件来放置链接器生成的中间代码输入节。分散加载描述文件中的一个执行区最多可以包含 `*(Veneer$$Code)` 节选择器。

如果此操作是安全的，链接器则会将中间代码输入节放到 `*(Veneer$$Code)` 节选择器识别的区中。由于地址范围问题或执行区大小限制，可能无法将中间代码输入节分配到区中。如果不能将中间代码添加到指定的区中，则会将它添加到包含生成中间代码的重定位输入节的执行区中。

注意

在早期版本的 ARM 工具中，分散加载描述文件中的 `*(IWV$$Code)` 实例自动转换为 `*(Veneer$$Code)`。应在新描述中使用 `*(Veneer$$Code)`。

如果执行区中的代码数量超过以下数量，则会忽略 `*(Veneer$$Code)`：Thumb 代码 4Mb、 Thumb-2 代码 16Mb 以及 ARM 代码 32Mb。

5.2.2 创建根执行区

如果指定了映像的初始入口点，或者由于您仅使用一个 `ENTRY` 指令而使链接器创建了一个初始入口点，则必须确保该入口点位于根区中。根区是指加载地址和执行地址相同的区。如果初始入口点不在根区中，则链接会失败，且链接器会生成错误消息。

要在分散加载描述文件中将区指定为根区，您可以执行以下任一操作：

- 显式地将 `ABSOLUTE` 指定为执行区属性 （或将其作为缺省设置），并且第一个执行区及其所在的加载区使用相同的地址。要使执行区地址与加载区地址相同，请执行以下任一操作：
  - 为执行区基址和加载区基址指定相同的数值
  - 为加载区中的第一个执行区指定 `+0` 偏移。

    如果为加载区中的所有后续执行区指定 0 偏移 (`+0`)，则它们均为根区。

请参阅示例 5-1。

- 使用 FIXED 执行区属性以确保特定区的加载地址和执行地址是相同的。请参阅示例 5-2 和第5-11 页的图 5-5。

可以使用 FIXED 属性将任何执行区放在 ROM 中的特定地址。有关详细信息，请参阅第5-11 页的*将区放在固定地址中*。

示例 5-1 指定相同的加载和执行地址

```
LR_1 0x040000          ; load region starts at 0x40000
{                      ; start of execution region descriptions
    ER_RO 0x040000     ; load address = execution address
    {
        * (+RO)        ; all RO sections (must include section with
                       ; initial entry point)
    }
    ...                ; rest of scatter description
}
```

示例 5-2 使用 FIXED 属性

```
LR_1 0x040000              ; load region starts at 0x40000
{                          ; start of execution region descriptions
    ER_RO 0x040000         ; load address = execution address
    {
        * (+RO)            ; RO sections other than those in init.o
    }
    ER_INIT 0x080000 FIXED ; load address and execution address of this
                           ; execution region are fixed at 0x80000
    {
        init.o(+RO)        ; all RO sections from init.o
    }
    ...                    ; rest of scatter description
}
```

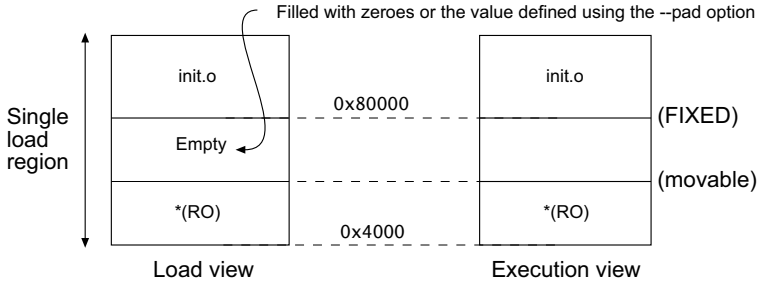

图 5-5 固定执行区的内存映射

将区放在固定地址中

可以在执行区分散加载描述文件中使用 FIXED 属性来创建根区，将在固定地址加载和执行这些区。

FIXED 用于在单个加载区内 （因而通常为单个 ROM 设备）创建多个根区。例如，可以使用此属性将函数或数据块 （如常数表或校验和）放到 ROM 中的固定地址，以便可以通过指针方便地对其进行访问。

例如，如果指定将某些初始化代码放在 ROM 开头，而将校验和放在 ROM 末尾，则可能不会使用某些内存内容。应使用 * 或 .ANY 模块选择器填充初始化块末尾和数据块开头之间的区。

注意

要使代码更易于维护和调试，请在分散加载描述文件中使用最少数量的位置说明，而由链接器提供函数和数据的详细位置信息。

无法指定已部分链接的组件对象。例如，如果部分链接 obj1.o、obj2.o 和 obj3.o 对象以生成 obj_all.o，则会在结果对象中弃用结果组件对象的名称。因此，不能按名称引用其中的某个对象，例如 obj1.o。只能引用合并的对象 obj_all.o。

将函数和数据放在特定地址中

通常，编译器通过单个源文件生成 RO、RW 和 ZI 节。这些区包含源文件中的所有代码和数据。要将单个函数或数据项放在固定地址中，您必须允许链接器单独处理该函数或数据，而与输入文件的其余部分分开。

链接器可以使用两种方法，将某个节放在特定地址中：

- 可以使用只选择一个节的节描述在所需地址创建执行区。

- 对于特别命名的节，链接器可以从节名称中获取放置地址。这些特别命名的节称为 __at 节。有关详细信息，请参阅第5-16 页的*使用 __at 节将节放在特定地址中*。

要将函数或变量放在特定地址中，必须将其放在自己的节中。可以使用几种方法来执行此操作：

- 将函数或数据项放在其自己的源文件中。
- 使用 --split_sections 编译器选项可为源文件中的每个函数分别生成一个 ELF 节。请参阅 《编译器参考指南》中第2-103 页的*--split_sections*。

  对于一些函数，此选项将稍微增大代码的大小，因为它减小了函数之间共享地址、数据和字符串文字的可能性。但是，通过指定 armlink --remove 以允许链接器删除未使用的函数，这有助于减小最终映像的总大小。
- 使用 __attribute__((section("*name*"))) 可创建多个命名节。请参阅 《编译器参考指南》中第4-45 页的*__attribute__((section))*。
- 使用汇编语言中的 AREA 指令。在汇编代码中，最小的可定位单元是 AREA。有关详细信息，请参阅 《汇编器指南》。

显式地使用分散加载放置已命名的节

示例 5-3 中的分散加载描述文件将：

- 初始化代码放在地址 0x0 中，其后是其余 RO 代码和除 data.o 对象中的 RO 数据之外的所有 RO 数据
- 所有全局 RW 变量放在 RAM 的 0x400000 中
- data.o 的 RO-DATA 表放在固定地址 0x1FF00 处。

示例 5-3 节放置

```
LR1 0x0 0x10000
{
    ER1 0x0 0x2000                ; Root Region, containing init code
    {                             ; place init code at exactly 0x0
        init.o (Init, +FIRST)
        * (+RO)                   ; rest of code and read-only data
    }
    RAM  0x400000                 ; RW & ZI data to be placed at 0x400000
    {
        * (+RW +ZI)
    }
    DATABLOCK 0x1FF00 FIXED 0xFF  ; execution region fixed at 0x1FF00
    {                             ; maximum space available for table is 0xFF
        data.o(+RO-DATA)          ; place RO data between 0x1FF00 and 0x1FFFF
    }
}
```

注意

在某些情况下，不适合将 FIXED 和单个加载区配合使用。下面是其他一些指定固定位置的方法：

- 如果加载程序可以处理多个加载区，请将 RO 代码或数据放在其自己的加载区中。
- 如果不需要将函数或数据放在 ROM 中的固定位置，请使用 ABSOLUTE 而不是 FIXED。加载程序随后会将数据从加载区复制到 RAM 中的指定地址。ABSOLUTE 是缺省属性。

- 要将数据结构放在内存映射的 I/O 位置，请使用两个加载区并指定 UNINIT。UNINIT 可确保不会将内存位置初始化为零。有关详细信息，请参阅 《开发指南》中第 3 章 *嵌入式软件开发*。

使用 __attribute__((section("name")))

标准编码方法是，将代码或数据对象放在其自己的源文件中，然后放置对象文件节。但是，也可以使用 __attribute__((section("name"))) 和分散加载描述文件来放置已命名的节。应创建一个模块 （如 adder.c）并显式地命名节，如示例 5-4 中所示。

示例 5-4 命名节

```
int variable __attribute__((section("foo"))) = 10;
```

可以使用分散加载描述文件指定已命名的节的放置位置，请参阅示例 5-5。如果代码和数据节的名称相同，则先放置代码节。

示例 5-5 放置节

```
FLASH 0x24000000 0x4000000
{
    ...                                 ; rest of code

    ADDER 0x08000000
    {
        adder.o (foo)                   ; select section foo from adder.o
    }
}
```

使用 __at 节将节放在特定地址中

可以为节指定一个特殊名称，以编码必须将其放置到的地址。您可以按以下方式指定名称：

.ARM.__at_*address*

其中：

address 是所需的节地址。可以按十六进制或十进制指定此地址。采用 .ARM.__at_*address* 格式的节是由缩写 __at 引用的。

在编译器中，可通过以下方式将变量分配给 __at 节：使用 __attribute__((section("*name*"))) 显式命名节或使用属性 __at 为您设置节的名称。请参阅示例 5-6。

注意

在使用 __attribute__((at(<address>))) 时， __AT 节名称中表示 *address* 的部分会标准化为 8 位十六进制数字。仅当您尝试在分散加载描述文件中按名称匹配节时，节的名称才有意义。

示例 5-6 将变量分配给 __at 节

```
; place variable1 in a section called .ARM.__at_0x00008000
int variable1 __attribute__((at(0x8000))) = 10;

; place variable2 in a section called .ARM.__at_0x8000
int variable2 __attribute__((section(".ARM.__at_0x8000"))) = 10;
```

有关详细信息，请参阅 《编译器参考指南》中第4-42 页的 *__attribute__((at(address)))* 和第4-45 页的 *__attribute__((section))*。

限制

- __at 节地址范围不能重叠，除非将重叠节放在不同的重叠区中
- 不允许在与位置无关的执行区中使用 __at 节
- 不能引用 __at 节的链接器定义的符号 $$Base、$$Limit 和 $$Length
- 不能在 System V 和 BPABI 可执行文件和 BPABI DLL 中使用 __at 节
- __at 节地址必须是其对齐边界的倍数
- __at 节忽略所有 +FIRST 或 +LAST 排序约束。

自动放置

此模式是使用链接器命令行选项 --autoat 启用的。有关详细信息，请参阅 《链接器参考指南》中第2-4 页的*--[no_]autoat*。

使用 --autoat 选项进行链接时，分散加载选择器不会放置 __at 节。相反，链接器将 __at 节放在兼容区中。如果找不到兼容区，链接器将为 __at 节创建加载和执行区。

链接器使用 --autoat 创建的所有执行区均具有 UNINIT 分散加载属性。如果需要对 ZI __at 节进行零初始化，则必须将其放在兼容区中。链接器使用 --autoat 创建的执行区必须具有至少 4 字节对齐的基址。如果有任何区未正确对齐，则链接器会产生错误消息。

兼容区是指：

- __at 地址位于执行区基址和限制范围内，其中限制是指基址 + 最大执行区大小。如果未设置最大大小，则假定限制为无限大的值。
- 加载区至少满足以下条件之一：
  - 它包含一个按标准分散加载规则与 __at 节相匹配的选择器
  - 它至少具有一个类型与 __at 节相同的节 （RO、 RW 或 ZI）
  - 它没有 EMPTY 属性。

    注意

    链接器将类型为 RW 的 __at 节视为与 RO 兼容。

示例 5-7 说明了具有如下类型的节：.ARM.__at_0x0 RO、 .ARM.__at_0x2000 RW、.ARM.__at_0x4000 ZI 和 .ARM.__at_0x8000 ZI。

示例 5-7 __at 节的自动放置

```
LR1 0x0
{
    ER_RO 0x0 0x2000
    {
        *(+RO)      ; .ARM.__at_0x0 lies within the bounds of ER_RO
    }
    ER_RW 0x2000 0x2000
    {
        *(+RW)      ; .ARM.__at_0x2000 lies within the bounds of ER_RW
    }
    ER_ZI 0x4000 0x2000
```

```
    {
        *(+ZI)      ; .ARM.__at_0x4000 lies within the bounds of ER_ZI
    }
}

; the linker creates a load and execution region for the __at section
; .ARM.__at_0x8000 because it lies outside all candidate regions.
```

手动放置

可以使用标准分散加载规则来确定用于放置 __at 节的执行区。

示例 5-8 说明了只读节 .ARM.__at_0x2000 和读写节 .ARM.__at_0x4000 的放置方式。在手动模式下，不会自动创建加载区和执行区。如果不能将 __at 节放在执行区中，则会产生错误。

示例 5-8 __at 节的手动放置

```
LR1 0x0
{
    ER_RO 0x0 0x2000
    {
        *(+RO)               ; .ARM.__at_0x0 is selected by +RO
    }
    ER_RO2 0x2000
    {
        *(.ARM.__at_0x2000)  ; .ARM.__at_0x2000 is selected by .ARM.__at_0x2000
    }
    ER2 0x4000
    {
        *(+RW +ZI)           ; .ARM.__at_0x4000 is selected by +RW
    }
}
```

将键放在闪存中

某些闪存设备需要将键写入到地址中以激活某些功能。__at 节提供了一种简单方法，将值写入到特定地址中。

假定设备的闪存范围从 0x8000 到 0x10000，并且需要将键放在地址 0x8000 中。若要对 __at 节执行此操作，必须声明一个变量，以便编译器可以生成名为 .ARM.__at_0x8000 的节。请参阅第5-16 页的示例 5-6。

示例 5-9 演示了一个手动放置闪存执行区的分散加载描述文件。

示例 5-9 手动放置闪存执行区

```
ER_FLASH 0x8000 0x2000
{
    *(+RO)                      ; other code, read-only data, and padding if
reqd
    *(.ARM.__at_0x8000)         ; key
}
```

示例 5-10 演示了一个自动放置闪存执行区的分散加载描述文件。可以使用链接器命令行选项 --autoat 来启用自动放置。

示例 5-10 自动放置闪存执行区

```
ER_FLASH 0x8000 0x2000
{
    *(+RO)                ; other code and read-only data, the
                          ; __at section is automatically selected
}
```

将结构映射到外围寄存器上

要将未初始化的变量放在外围寄存器上，您可以使用 ZI __at 节。假定有一个寄存器可在 0x10000000 处使用，则可定义一个名为 .ARM.__at_0x10000000 的 ZI __at 节。例如：

```
int foo __attribute__((section( ".ARM.__at_0x10000000" ), zero_init));
```

示例 5-11 演示了一个手动放置 ZI __at 节的分散加载描述文件。

示例 5-11 手动放置 ZI __at 节

```
ER_PERIPHERAL 0x10000000 UNINIT
{
    *(.ARM.__at_0x10000000)
}
```

使用自动放置时，假定 0x10000000 附近没有其他执行区，链接器将在 0x10000000 处自动创建一个包含 UNINIT 属性的区。

5.2.3 使用重叠区放置节

可以在分散加载描述文件中使用 OVERLAY 属性将多个执行区放置在同一地址处。需要使用重叠区管理器以确保一次只实例化一个执行区。ARM RealView 编译工具不提供重叠区管理器。

示例 5-12 在 RAM 中定义了一个静态节，后跟一系列重叠区。此处，一次只能实例化其中的一个节。

示例 5-12 指定根区

```
EMB_APP 0x8000
{
    .
    .
    STATIC_RAM 0x0                  ; contains most of the RW and ZI code/data
    {
            * (+RW,+ZI)
    }
    OVERLAY_A_RAM 0x1000 OVERLAY    ; start address of overlay...
    {
            module1.o (+RW,+ZI)
    }
    OVERLAY_B_RAM 0x1000 OVERLAY
    {
            module2.o (+RW,+ZI)
    }
    ...                             ; rest of scatter description...
}
```

标记为 OVERLAY 的区不会在启动时由 C 库初始化。重叠区使用的内存的内容由重叠区管理器负责。因此，重叠区管理器必须复制任何代码和数据，并在实例化某个区时初始化任何 ZI。如果该区包含初始化的数据，则还需要使用 NOCOMPRESS 避免进行 RW 数据压缩。

链接器定义的符号可以用于获取复制代码和数据所需的地址，有关详细信息，请参阅第4-3 页的*访问链接器定义的符号*。

OVERLAY 属性可以在地址与另一个区不同的单个区上使用。因此，重叠区可以用于防止 C 库启动代码初始化特定的区。与任何重叠区一样，这些区必须在代码中手动初始化。

重叠区可以有相对基址。具有 +offset 基址的重叠区的行为取决于该区之前的区以及 +offset 的值 （+0 有特殊含义）。

表 5-1 显示了 +offset 在用于 OVERLAY 属性时的效果。在分散加载描述文件中，REGION1 出现在 REGION2 之前的相邻位置上。

表 5-1 在重叠区中使用相对偏移量

REGION1 是否设置了 OVERLAY	+OFFSET	REGION2 基址
否	*<offset>*	REGION1 Limit + *<offset>*
是	+0	REGION1 Base Address
是	*<non-0 offset>*	REGION1 Limit + *<non-0 offset>*

示例 5-13 演示了相对偏移量如何用于重叠区及其对执行区地址的作用。

如果非重叠区域的长度未知，可以使用 0 相对偏移量指定重叠区的开始地址，以便将其放在静态节末尾的相邻位置。

示例 5-13 重叠区中的相对偏移量示例

```
EMB_APP 0x8000
{
    CODE 0x8000
    {
        *(+RO)
    }

    # REGION1 Base = CODE limit
    REGION1 +0 OVERLAY
    {
        module1.o(*)
    }

    # REGION2 Base = REGION1 Base
    REGION2 +0 OVERLAY
    {
        module2.o(*)
    }

    # REGION3 Base = REGION2 Base = REGION1 Base
    REGION3 +0 OVERLAY
    {
        module3.o(*)
    }
```

```
    # REGION4 Base = REGION3 Limit + 4
    Region4 +4 OVERLAY
    {
        module4.o(*)
    }
}
```

5.2.4 为根区分配节

有许多 ARM 库节必须放置在根区中，例如 __main.o、__scatter*.o、__dc*.o 和 *Region$$Table。此列表在不同版本中可能有差异。链接器可以使用 InRoot$$Sections 自动放置所有这些节，而不会影响将来的使用。

可以使用分散加载描述文件按照与已命名节相同的方法指定根节。示例 5-14 使用节选择器 InRoot$$Sections 放置必须位于根区中的所有节。

示例 5-14 指定根区

```
ROM_LOAD 0x0000 0x4000
{
  ROM_EXEC 0x0000 0x4000      ; root region at 0x0
  {
    vectors.o (Vect, +FIRST)  ; Vector table
    * (InRoot$$Sections)      ; All library sections that must be in a
                              ; root region, for example, __main.o,
                              ; __scatter*.o, __dc*.o, and * Region$$Table
  }
  RAM 0x10000 0x8000
  {
    * (+RO, +RW, +ZI)         ; all other sections
  }
}
```

5.2.5 保留空白区

可以在执行区分散加载描述中使用 EMPTY 属性，为堆栈保留一个空白内存块。

该内存块并不构成加载区的一部分，而是在执行时分配使用的。由于它是作为虚 ZI 区创建的，因此链接器使用以下符号对其进行访问：

- Image$$region_name$$ZI$$Base
- Image$$region_name$$ZI$$Limit
- Image$$region_name$$ZI$$Length.

如果指定的长度为负值，则将该地址作为区结束地址。它必须是绝对地址，而不是相对地址。例如，第5-26 页的示例 5-15 中说明的执行区定义 STACK 0x800000 EMPTY -0x10000 定义了一个名为 STACK 的区，它的开始地址是 0x7F0000，结束地址是 0x800000。

注意

在运行时，不会将为 EMPTY 执行区创建的虚 ZI 区初始化为零。

如果地址采用相对格式 (+*n*)，并且长度为负值，链接器将生成错误。

示例 5-15 为堆栈保留区

```
LR_1 0x80000                            ; load region starts at 0x80000
{
    STACK 0x800000 EMPTY -0x10000        ; region ends at 0x800000 because of the
                                        ; negative length. The start of the
region
                                        ; is calculated using the length.
    {
                                        ; Empty region used to place stack
    }
    HEAP +0 EMPTY 0x10000               ; region starts at the end of previous
                                        ; region. End of region calculated using
                                        ; positive length
    {
                                        ; Empty region used to place heap
    }
    ...                                 ; rest of scatter description...
}
```

第5-27 页的图 5-6 是此示例的图形表示形式。

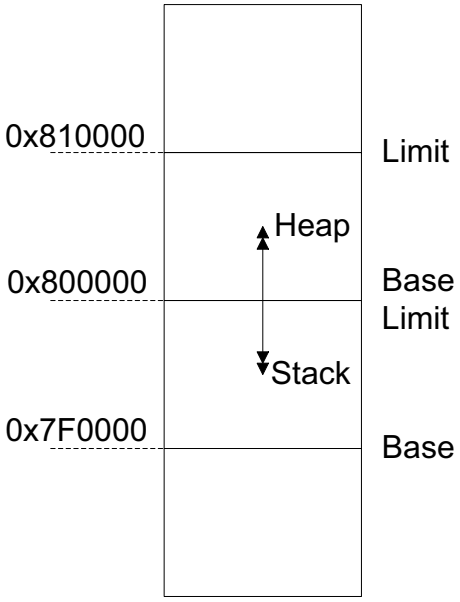

图 5-6 为堆栈保留区

在此示例中，链接器生成以下符号：

```
Image$$STACK$$ZI$$Base      = 0x7f0000
Image$$STACK$$ZI$$Limit     = 0x800000
Image$$STACK$$ZI$$Length    = 0x10000
Image$$HEAP$$ZI$$Base       = 0x800000
Image$$HEAP$$ZI$$Limit      = 0x810000
Image$$HEAP$$ZI$$Length     = 0x10000
```

注意

EMPTY 属性仅适用于执行区。如果在加载区定义中使用 EMPTY 属性，链接器将生成警告并忽略该属性。

链接器检查用于 EMPTY 区的地址空间是否与任何其他执行区重叠。

5.2.6 放置 ARM 库

可以将 ARM 标准 C 和 C++ 库中的代码放在分散加载描述文件中。应使用 *armlib 或 *armlib*，以使链接器能够解析分散加载文件中的库命名。例如：

```
ER 0x2000
{
    *armlib/c_* (+RO)                   ; all ARM-supplied C libraries
    ...                                 ; rest of scatter description...
}
```

示例 5-16 说明了如何放置库代码。

注意

示例 5-16 在路径名中使用正斜杠，以确保在 Windows 和 Unix 平台上能够识别它们。

示例 5-16 放置 ARM 库代码

```
ROM1 0
{
    * (InRoot$$Sections)
    * (+RO)
    ROM2 0x1000
    {
        *armlib/c_* (+RO)         ; all ARM-supplied C library functions
    }
}
ROM3 0x2000
{
    *armlib/h_* (+RO)                   ; just the ARM-supplied __ARM_*
                                        ; redistributable library functions
}
RAM1 0x3000
{
    *armlib* (+RO)                      ; all other ARM-supplied library code
                                        ; e.g. floating-point libraries
}
RAM2 0x4000
{
    * (+RW, +ZI)
}
```

5.2.7 在页边界上创建区

ALIGN 指令生成一个 ELF 文件，该文件可以直接加载到目标中，其中每个执行区都始于一个页边界。

示例 5-17 假定页面大小为 65536，并生成一个 ELF 文件，其中每个区都始于一个新页面。

示例 5-17 在页边界上创建区

```
LR1 +4 ALIGN 65536              ; load region at 65536
{
    ER1 +0 ALIGN 65536          ; first region at first page boundary
    {
        *(+RO)                  ; all RO sections are placed consecutively here
    }
    ER2 +0 ALIGN 65536          ; second region at next available page boundary
    {
        *(+RW)                  ; all RW sections are placed consecutively here
    }
    ER3 +0 ALIGN 65536          ; third region at next available page boundary
   {
        *(+ZI)                  ; all ZI sections are placed consecutively here
    }
    ...                         ; rest of scatter description...
}
```

5.2.8 使用预处理指令

可以使用分散加载描述文件中的第一行指定链接器为处理该文件而调用的预处理器。此命令的格式如下：

```
#! <preprocessor> [pre_processor_flags]
```

通常情况下，此命令是 `#! armcc -E`。

链接器可以使用一组有限的运算符执行简单的表达式求值，即 +、-、 *、 /、AND、 OR 和括号。OR 和 AND 实现遵循 C 运算符优先级规则。

您可以将预处理指令添加到分散加载描述文件的顶部。例如：

```
#define ADDRESS 0x20000000
#include "include_file_1.h"
```

链接器解析预处理的分散加载描述文件，其中将这些指令视为注释并忽略。

举一个简单的例子：

```
#define AN_ADDRESS (BASE_ADDRESS+(ALIAS_NUMBER*ALIAS_SIZE))
```

使用以下指令：

```
#define BASE_ADDRESS 0x8000
#define ALIAS_NUMBER 0x2
#define ALIAS_SIZE 0x400
```

如果分散加载描述文件包含：

```
LOAD_FLASH AN_ADDRESS    ; start address
```

进行预处理后，将对其进行计算，结果为：

```
LOAD_FLASH ( 0x8000 + ( 0x2 * 0x400 ))  ; start address
```

在进行计算后，链接器解析分散加载文件以生成加载区：

```
LOAD_FLASH 0x8808 ; start address
```

有关详细信息，请参阅 《链接器参考指南》中第2-42 页的*--predefine="string"*。

5.3 简单映像的等效分散加载描述

第3-22 页的*使用命令行选项创建简单映像*中介绍了如何使用以下命令行选项创建简单映像类型：--reloc、--ro-base、--rw-base、--ropi、--rwpi 和 --split。通过使用 --scatter 命令行选项以及包含相应分散加载描述之一的文件，可以创建相同的映像类型。

5.3.1 类型 1，一个加载区和几个连续执行区

此类映像由加载视图中的单个加载区以及执行视图中的三个执行区组成。执行区是在内存映射中连续放置的。

--ro-base *address* 指定包含 RO 输出节的区的加载和执行地址。示例 5-18 演示了与使用 --ro-base 0x040000 等效的分散加载描述。

示例 5-18 单个加载区和几个连续执行区

```
LR_1 0x040000     ; Define the load region name as LR_1, the region starts at 0x040000.
{
    ER_RO +0      ; First execution region is called ER_RO, region starts at end of previous region.
                  ; However, since there is no previous region, the address is 0x040000.
    {
        * (+RO)   ; All RO sections go into this region, they are placed consecutively.
    }
    ER_RW +0      ; Second execution region is called ER_RW, the region starts at the end of the
                  ; previous region. The address is 0x040000 + size of ER_RO region.
    {
        * (+RW)   ; All RW sections go into this region, they are placed consecutively.
    }
    ER_ZI +0      ; Last execution region is called ER_ZI, the region starts at the end of the
                  ; previous region at 0x040000 + the size of the ER_RO regions + the size of
                  ; the ER_RW regions.
    {
        * (+ZI)   ; All ZI sections are placed consecutively here.
    }
}
```

示例 5-18 中所示的描述创建一个映像，其中包含一个名为 LR_1 的加载区，其加载地址为 0x040000。

该映像包含三个名为 ER_RO、ER_RW 和 ER_ZI 的执行区，它们分别包含 RO、 RW 和 ZI 输出节。RO 和 RW 是根区。ZI 是在运行时动态创建的。ER_RO 的执行地址是 0x040000。通过在执行区描述中使用 *+offset* 格式的基址指示符，可以在内存映射中连续放置所有三个执行区。这样，即可紧靠前一个执行区后面放置下一个执行区。

--reloc 选项用于生成可重定位的映像。单独使用时，--reloc 生成的映像类似于简单类型 1，但单个加载区具有 RELOC 属性。

修改后的 ropi 示例版本

在此版本中，执行区连续放置在内存映射中。但是，--ropi 将包含 RO 输出节的加载和执行区标记为与位置无关。

示例 5-19 演示了与使用 --ro-base 0x010000 --ropi 等效的分散加载描述。

示例 5-19 与位置无关的代码

```
LR_1 0x010000 PI        ; The first load region is at 0x010000.
{
    ER_RO +0            ; The PI attribute is inherited from parent.
                        ; The default execution address is 0x010000, but the code can be moved.
    {
        * (+RO)         ; All the RO sections go here.
    }
    ER_RW +0 ABSOLUTE   ; PI attribute is overridden by ABSOLUTE.
    {
        * (+RW)         ; The RW sections are placed next. They cannot be moved.
    }
    ER_ZI +0            ; ER_ZI region placed after ER_RW region.
    {
        * (+ZI)         ; All the ZI sections are placed consecutively here.
    }
}
```

如示例 5-19 中所示， RO 执行区 ER_RO 从加载区 LR_1 继承 PI 属性。下一个执行区 ER_RW 被标记为 ABSOLUTE，并使用 *+offset* 格式的基址指示符。这可防止 ER_RW 从 ER_RO 继承 PI 属性。另外，由于 ER_ZI 区的偏移为 +0，因此它从 ER_RW 区继承 ABSOLUTE 属性。

5.3.2 类型 2，一个加载区和几个不连续的执行区

此类映像由加载视图中的单个加载区以及执行视图中的三个执行区组成。它与类型 1 映像相似，但 RW 执行区与 RO 执行区不相邻。

`--ro-base=`*address1* 指定包含 RO 输出节的区的加载和执行地址。`--rw-base=`*address2* 指定 RW 执行区的执行地址。

示例 5-20 演示了与使用 `--ro-base=0x010000 --rw-base=0x040000` 等效的分散加载描述。

示例 5-20 单个加载区和多个执行区

```
LR_1 0x010000        ; Defines the load region name as LR_1
{
    ER_RO +0         ; The first execution region is called ER_RO and starts at end of previous region.
                     ; Since there is no previous region, the address is 0x010000.
    {
        * (+RO)      ; All RO sections are placed consecutively into this region.
    }
    ER_RW 0x040000   ; Second execution region is called ER_RW and starts at 0x040000.
    {
        * (+RW)      ; All RW sections are placed consecutively into this region.
    }
    ER_ZI +0         ; The last execution region is called ER_ZI.
                     ; The address is 0x040000 + size of ER_RW region.
    {
        * (+ZI)      ; All ZI sections are placed consecutively here.
    }
}
```

此描述创建一个映像，其中包含一个名为 `LR_1` 的加载区，其加载地址是 `0x010000`。

该映像包含三个名为 `ER_RO`、`ER_RW` 和 `ER_ZI` 的执行区，它们分别包含 RO、 RW 和 ZI 输出节。RO 区是根区。`ER_RO` 的执行地址是 `0x010000`。

`ER_RW` 执行区与 `ER_RO` 不相邻。其执行地址是 `0x040000`。

`ER_ZI` 执行区紧靠上一个执行区 `ER_RW` 后面放置。

rwpi 示例变化版本

这与使用 --rw-base 的类型 2 映像相似， RW 执行区与 RO 执行区是分开的。但是，--rwpi 将包含 RW 输出节的执行区标记为与位置无关。

示例 5-21 演示了与使用 --ro-base=0x010000 --rw-base=0x018000 --rwpi 等效的分散加载描述。

示例 5-21 与位置无关的数据

```
LR_1 0x010000           ; The first load region is at 0x010000.
{
    ER_RO +0            ; Default ABSOLUTE attribute is inherited from parent. The execution address
                        ; is 0x010000. The code and ro data cannot be moved.
    {
        * (+RO)         ; All the RO sections go here.
    }
    ER_RW 0x018000 PI   ; PI attribute overrides ABSOLUTE
    {
        * (+RW)         ; The RW sections are placed at 0x018000 and they can be moved.
    }
    ER_ZI +0            ; ER_ZI region placed after ER_RW region.
    {
        * (+ZI)         ; All the ZI sections are placed consecutively here.
    }
}
```

RO 执行区 ER_RO 从加载区 LR_1 继承 ABSOLUTE 属性。下一个执行区 ER_RW 被标记为 PI。另外，由于 ER_ZI 区的偏移为 +0，因此它从 ER_RW 区继承 PI 属性。

还可以编写类似的分散加载描述，以对应于 --ropi 和 --rwpi 与类型 2 和类型 3 映像的其他组合用法。

5.3.3 类型 3，两个加载区和几个不连续的执行区

类型 3 映像由加载视图中的两个加载区以及执行视图中的三个执行区组成。它们与类型 2 映像相似，但类型 2 映像中的单个加载区现在拆分为两个加载区。

可以使用以下链接器选项重定位并拆分加载区：

--reloc　　--reloc --split 组合生成的映像类似于简单类型 3，但两个加载区现在具有 RELOC 属性。

--ro-base=*address1*

指定包含 RO 输出节的区的加载和执行地址。

--rw-base=*address2*

指定包含 RW 输出节的区的加载和执行地址。

--split 将缺省的单个加载区 （包含 RO 和 RW 输出节）拆分为两个加载区。一个加载区包含 RO 输出节；另一个加载区包含 RW 输出节。

示例 5-22 演示了与使用 --ro-base=0x010000 --rw-base=0x040000 --split 等效的分散加载描述。

在本示例中：

- 此描述创建一个映像，其中包含两个名为 LR_1 和 LR_2 的加载区，它们的加载地址是 0x010000 和 0x040000。
- 该映像包含三个名为 ER_RO、ER_RW 和 ER_ZI 的执行区，它们分别包含 RO、RW 和 ZI 输出节。ER_RO 的执行地址是 0x010000。
- ER_RW 执行区与 ER_RO 不相邻。其执行地址是 0x040000。
- ER_ZI 执行区紧靠上一个执行区 ER_RW 后面放置。

示例 5-22 多个加载区

```
LR_1 0x010000     ; The first load region is at 0x010000.
{
    ER_RO +0      ; The address is 0x010000.
    {
        * (+RO)
    }
}
LR_2 0x040000     ; The second load region is at 0x040000.
{
    ER_RW +0      ; The address is 0x040000.
    {
        * (+RW)   ; All RW sections are placed consecutively into this region.
    }
    ER_ZI +0      ; The address is 0x040000 + size of ER_RW region.
    {
```

```
        * (+ZI)   ; All ZI sections are placed consecutively into this region.
    }
}
```

可重定位加载区示例变化版本

此类型 3 映像也由加载视图中的两个加载区以及执行视图中的三个执行区组成。但是，用于指定两个加载区的 --reloc 现在具有 RELOC 属性。

示例 5-23 演示了与使用 --ro-base 0x010000 --rw-base 0x040000 --reloc --split 等效的分散加载描述。

示例 5-23 可重定位的加载区

```
LR_1 0x010000 RELOC
{
     ER_RO + 0
     {
         * (+RO)
     }
}

LR2 0x040000 RELOC
{
     ER_RW + 0
     {
         * (+RW)
     }

     ER_ZI +0
     {
         * (+ZI)
     }
}
```