| 嵌入式培训专家

10年口碑积累，成功培养50000多名研发工程师，铸就专业品牌形象

华清远见的企业理念是不仅要做良心教育、做专业教育，更要做受人尊敬的职业教育。

《CORTEX-M3+UCOS-II 嵌入式系统开发入门与应用》

作者：华清远见

专业始于专注 卓识源于远见

第 4 章 基于 RealView 开发环境的嵌入式软件开发

专业始于专注 卓识源于远见

4.1 RealView 编译器的缺省行为

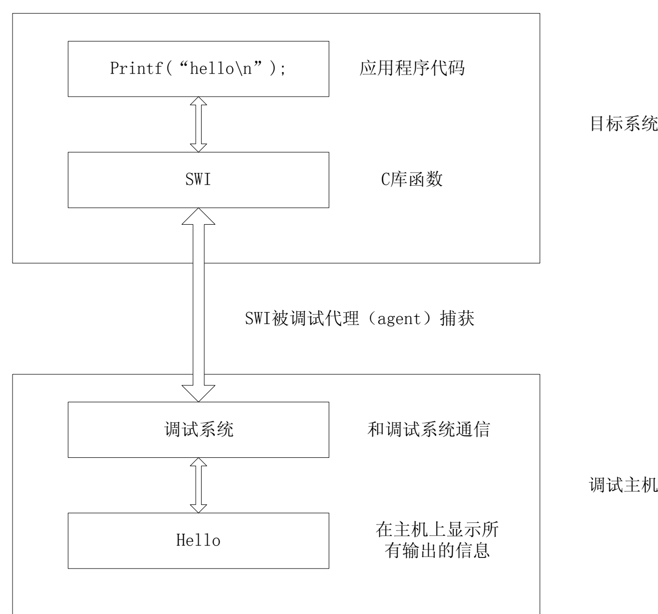
多数嵌入式应用程序最初都是在原型环境下开发的。无论什么样的原型环境的资源与最终产品环境都是有差异的。因此，考虑如何将嵌入式应用程序从其所依赖的开发工具或调试环境中移植到在目标硬件上独立运行是非常重要的。

开始编写嵌入式应用程序时，开发者可能并不清楚目标硬件的具体规格。如目标系统使用了什么样的外围设备，存储器映射情况甚至不能确定处理器的型号。
为在了解这些详细信息前能够继续软件的开发，RVCT 工具提供了很多默认的操作，使用户能编译和调试与目标系统无关的应用程序代码。下面详细介绍这些编译选项，只有深入了解这些编译选项设置，才能使开发更顺利的进行。

4.1.1 Semihosting

1．Semihosting 简介

在 RVCT C 库中，对某些 ISO C 功能的支持由主机调试环境提供。提供该功能的机制被称为 Semihosting[1]。大多数的 ARM 调试系统都支持 Semihosting 机制，如 ReslView Debugger AXD 等。
调试系统提供这种机制是非常有用的，因为用于开发使用的硬件系统经常没有最终系统的所有输入和输出设备。在这种情况下，Semihosting 可让主机代替目标系统提供这些设备的功能。举例来说，此机制可以用于启用 C 库中的函数（例如，printf()和 scanf()）使用主机的屏幕和键盘，而不使用目标系统的屏幕和键盘。
半主机由一组已定义的 SWI 操作来实现。应用程序调用相应的 SWI，然后由调试代理程序（Debug Agent）处理 SWI 异常。调试代理程序完成系统与主机之间的通信。
图 4.1 显示了 Semihosting 机制的处理过程。



[1] 在一些 ARM 的中文参考文献中，将 Semihosting 译为半主机。

华清远见

图 4.1 Semihosting 机制的处理过程

在很多情况下，Seimhosting SWI 由库函数内的代码调用。应用程序也可以直接调用。有关支持 ARM C 库中 Seimhosting 的详细信息，请参阅 ARM 相关文档。

2．Semihosting 软件接口

ARM 和 Thumb SWI 指令包含一个软中断号，该中断号可以被应用程序使用。此编号可以由系统中的 SWI 处理程序进行解码。有关 SWI 处理程序的详细信息，请参阅本书中 ARM 异常处理一节。

Semihosting 使用固定的中断号调用相应的处理程序。用于 Jemihosting 的 SWI 是：

- 0x123456（在 ARM 状态下）;
- 0xAB（在 Thumb 状态下）。

值得注意的是，用户在编写自己的中断处理程序时，避免使用 Semihosting 已经使用的中断向量号。

调试代理通过 SWI 的中断向量号识别该软中断是目标系统提出的 Semihosting 请求。具体是何种 Semihosting 请求（如是键盘输入请求或屏幕显示请求）通过向寄存器 r0 传递不同的参数进行区分。所有其他参数通过一个数据块进行传递。该数据块的地址通过寄存器 r1 传递给中断处理程序。软中断的处理结果放在 r0 中返回，也可以通过显式的返回值，或传递数据块的指针带回程序的处理结果。即使未返回结果，也假定 r0 是被使用的。

用 r0 传递的可用 Semihosting 操作编号分配如下。

- 0x00-0x31：这些编号由 ARM 公司使用。
- 0x32-0xFF：这些编号由 ARM 公司保留，以备将来使用。
- 0x100-0x1FF：这些编号保留给用户应用程序。
- 0x200-0xFFFFFFFF：这些编号未定义。当前未使用并且不推荐使用这些编号。

在以下部分中，操作名称之后的括号中的编号是调用 Semihosting 操作时放入 r0 的值。例如，SYS_OPEN (0x01)。

如果从汇编语言代码中调用 SWI，最好使用 semihost.h 中定义的操作名称。可以用 EQU 伪操作定义操作名称。例如：

```
SYS_OPEN    EQU 0x01
SYS_CLOSE   EQU 0x02
```

3．Semihosting 需求函数

Semihosting 需求函数如表 4.1 所示。如果使用默认的 Semihosting 功能，用户不需要编写任何代码。也可以重新实现部分的输入/输出函数，将这些函数和标准 Semihosting 混合使用。

表 4.1 Semihosting 函数列表

函 数 名 称	描 述
SYS_OPEN (0x01)	打开文件
SYS_CLOSE (0x02)	关闭使用 SYS_OPEN 打开的文件
SYS_WRITEC (0x03)	向控制台输出字符
SYS_WRITE0 (0x04)	将空终止的字符串写入控制台
SYS_WRITE (0x05)	写入主机上的文件

续表

函 数 名 称	描 述
SYS_READ (0x06)	将文件内容读取到缓存器
SYS_READC (0x07)	从控制台读取字节
SYS_ISERROR (0x08)	确定返回代码是否错误
SYS_ISTTY (0x09)	检查文件是否连接到交互设备
SYS_SEEK (0x0A)	搜索到文件中的某个位置
SYS_FLEN (0x0C)	返回文件的长度
SYS_TMPNAM (0x0D)	返回文件的临时名称
SYS_REMOVE (0x0E)	删除主机上的文件
SYS_RENAME (0x0F)	重命名主机上的文件
SYS_CLOCK (0x10)	执行开始后的厘秒数
SYS_TIME (0x11)	1970 年 1 月 1 日到现在的秒数
SYS_SYSTEM (0x12)	将命令传递给主机命令行解释程序
SYS_ERRNO (0x13)	获得 C 库 errno 变量的值
SYS_GET_CMDLINE (0x15)	获得用于调用可执行程序的命令行
SYS_HEAPINFO (0x16)	获得系统堆参数
SYS_ELAPSED (0x30)	获得自执行开始的目标滴答声数目
SYS_TICKFREQ (0x31)	确定滴答声的频率

华清远见

4.1.2 C 库结构

从概念上来讲，C 库函数可被化分成两类，一类为 ISO C 语言的规范部分，该部分的主要功能是向用户提供一个调用接口；另一类为 ISO C 语言规范提供支持。图 4.2 显示了这两类函数在 C 库中的结构。

对部分 ISO C 功能的支持是由主机调试环境在支持函数的设备驱动程序级别提供的。例如，RVCT C 库通过写入调试器控制台窗口来实现 ISO C printf()系列函数。通过调用__sys_write()来提供该功能。这是一个执行半主机 SWI 的支持函数，使字符串被写入到控制台。

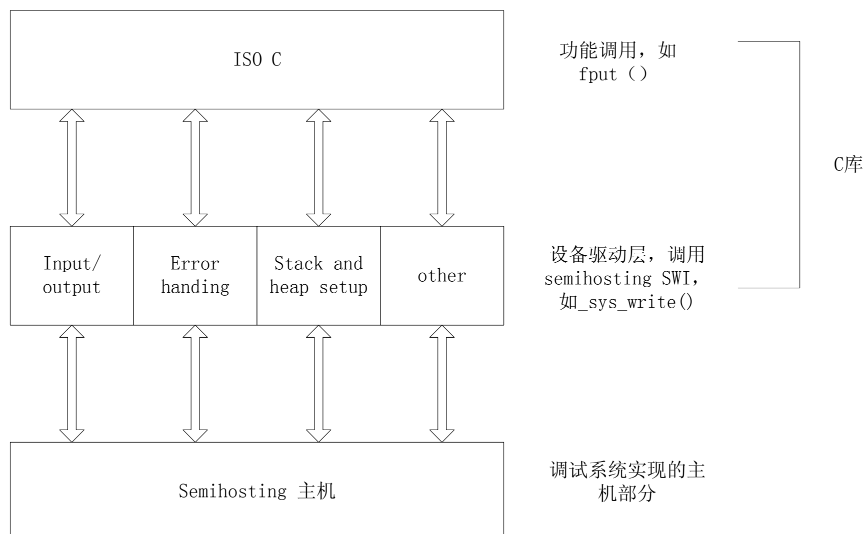


图 4.2　C 库的函数结构

4.1.3　默认存储器映射

对于没有描述存储器映射的映像（Image），RVCT 根据默认存储器映射放置代码和数据。默认的存储器映射如图 4.3 所示。

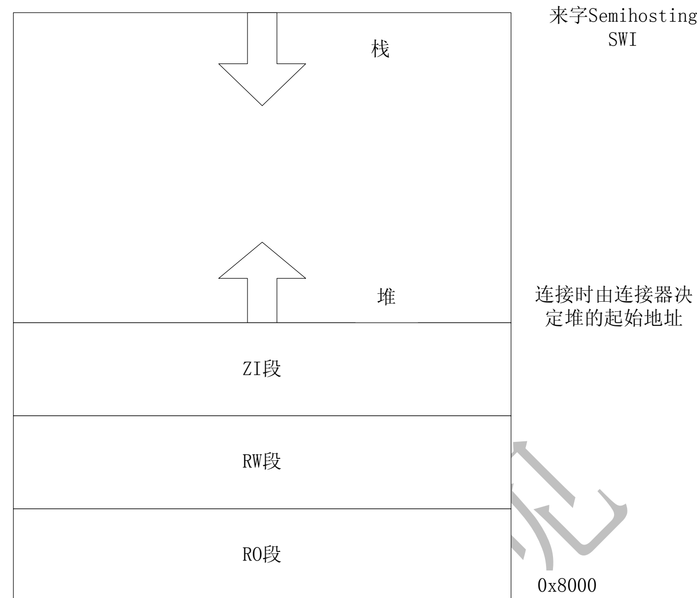


图 4.3　默认存储器映射

结合图 4.3，可以看出默认的存储器映射使用以下规则。

- 链接映像，在地址 0x8000 加载并运行。首先放置所有的 RO（只读）段，其次是 RW（读写）段，然后是 ZI（零初始化）段。
- 堆（Heap）直接从 ZI 段的顶端地址算起，因此，其准确位置在链接时决定。
- 栈（Stack）的起始地址在应用程序启动过程时由 Semihosting 操作提供。具体 Semihosting 操作设置的值因调试系统的不同而不同。

① RealView ARMulator ISS（RVISS）设置为配置文件 peripherals.ami 中设定的值。默认值是 0x08000000。
② Multi-ICE 将该地址设置为调试器内部变量 top_of_memory 的值。默认值是 0x00080000。

4.1.4　链接程序放置规则

链接程序遵守一组规则，以决定代码和数据位于存储器中的什么位置，如图 4.4 所示。

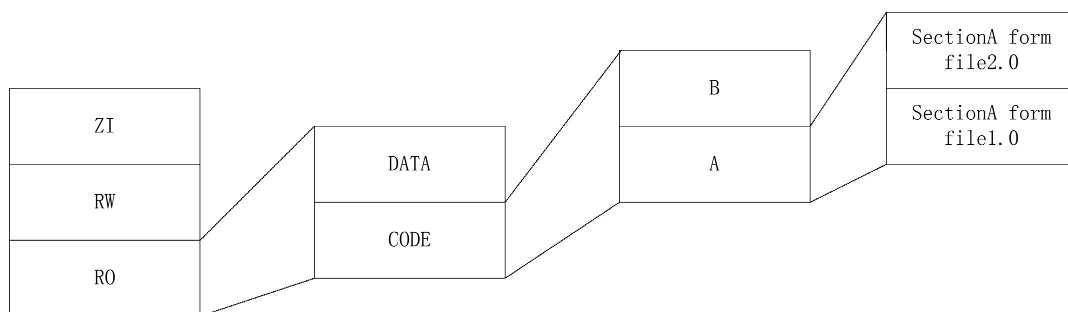


图 4.4　链接程序放置规则

华清远见

链接程序放置遵循以下规则。

（1）映像首先按属性组织：RO 段在最低的存储器地址，其次是 RW 段，然后是 ZI 段。每一种属性中代码在数据之前。

（2）链接程序按名称的字母顺序放置输入段（Section）。输入段名称即汇编程序 AREA 伪操作定义的名称。

（3）在输入段中，独立对象的代码和数据按照对象文件在链接程序命令行中被指定的顺序放置。

要精确放置代码和数据，ARM 公司建议不要过分依靠这些规则。相反，必须使用分散加载机制来完全控制代码和数据的放置。

4.1.5 应用程序启动

多数嵌入式系统中，执行主任务前都要执行初始化序列来设置系统。默认的 RVCT 初始化序列如图 4.5 所示。

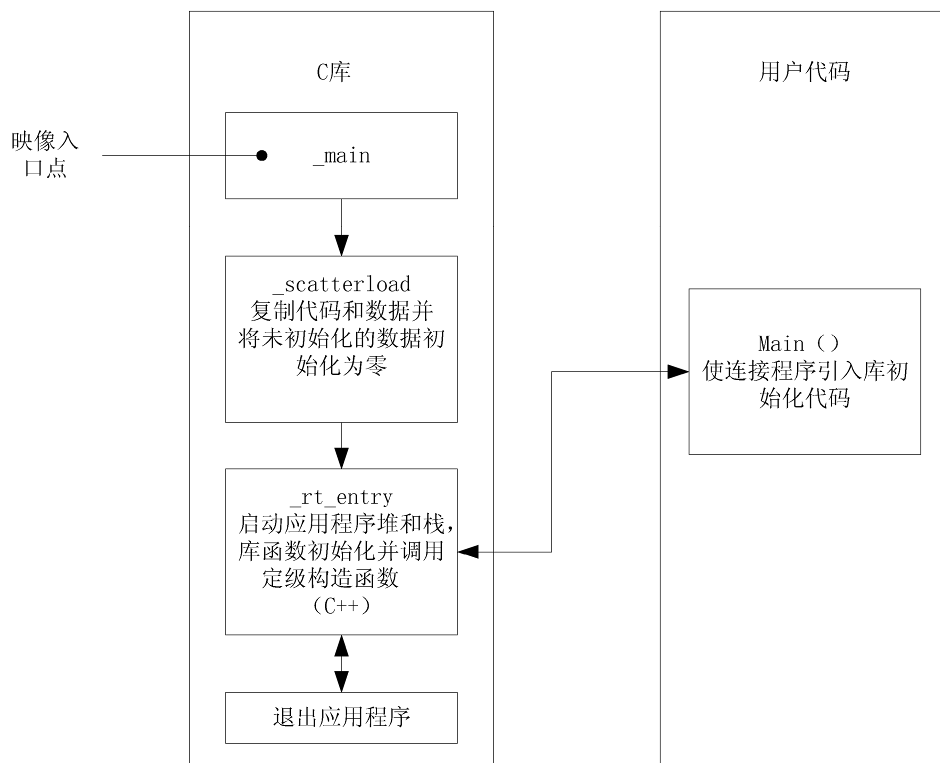


图 4.5 默认 RVCT 初始化序列

在进入用户代码（main()）前，初始化序列可分成 3 个功能块：__main 直接跳转到__scatterload，__scatterload 负责建立运行时的映像存储器映射，而__rt_entry（运行时的入口）则负责初始化 C 库。

__scatterload 执行代码和数据复制以及 ZI 数据的清零。对于 ZI 数据的清零和未改变的 RW 数据来说，这一步总是要做的。

__scatterload 跳转到__rt_entry。它设置应用程序的栈和堆，初始化库函数及其静态数据，并调用任何全局声明的对象的构造函数（仅 C ++）。

然后__rt_entry 跳转到应用程序入口 main()。主应用程序的结束执行时，__rt_entry 将库关闭，然后把控制权交还给调试器。

RVCT 中，函数 main()有一个特殊含义。main()函数的存在强制链接程序链接到__main 和__rt_entry 中的初始化代码。没有 main()函数，就不会链接到初始化进程，那么一些标准 C 库功能就不会得到支持。

集团官网：www.hqyj.com 嵌入式学院：www.embedu.org 企业学院：www.farsight.com.cn 北京 | 上海 | 深圳 | 成都 | 南京 | 武汉 | 西安 | 广州
移动互联网学院：www.3g-edu.org 物联网学院：www.topsight.cn 咨询电话：400-706-1880，15010390966

4.2 调整 C 库使其适应目标硬件

默认情况下，C 库利用 semihostig 机制来提供设备驱动级的功能，使得主机能够用作输入和输出设备。这种机制对于嵌入式开发十分有用，因为用于开发的硬件系统通常没有最终系统的输入和输出设备。
本节介绍如何重定向代码中的 Simehosting 库函数，使其真正适用目标系统。

4.2.1 C 库函数重定向

所谓 C 库函数重定向，就是用户使用自己编写的函数代码代替 C 库中的函数，使最终程序更适用于实际的目标硬件。图 4.6 显示了 C 库函数重定向的过程。

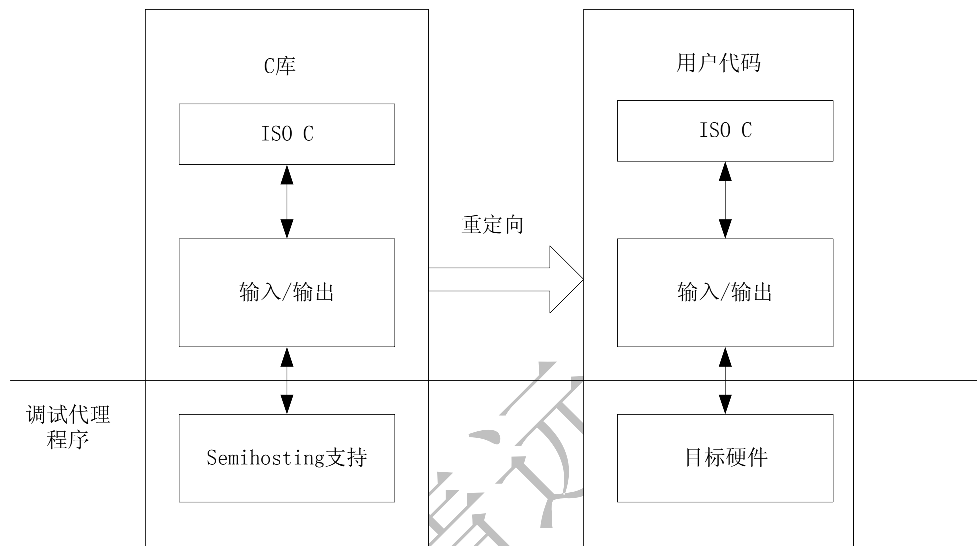


图 4.6 C 库函数重定向

最简单的函数重定向的例子就是用户希望 fputc()函数能够将字符从目标系统的串口输出，而不是在调试时将字符从调试器的控制台输出，这时就需要重新实现该函数。下面的例子将 fputc()的输入字符参数重新指向一个连续输出函数 sendchar()，将该例在一个独立的源文件中实现的。这样，fputc()在依目标而定的输出和 C 库标准输出函数之间充当一个抽象层。
例子程序的代码如下：

```
extern void sendchar(char *ch);
int fputc(int ch, FILE *f)
{   /* e.g. write a character to an UART */
    char tempch = ch;
    sendchar(&tempch);
    return ch;
}
```

4.2.2 从最终代码映像中去掉 Semihosting

在一个实际的应用程序中，不可能支持 Semihosting 的 SWI 操作机制。因此，必须在最终的代码映像中去掉 C 库中的 Semihosting 函数。
为确保最终映像文件中没有链接 Semihosting 的 SWI 的函数，必须引入符号__use_no_ semihosting_swi。使用方法如下。

- 在 C 模块中，使用#pragma 命令：

```
#pragma import(__use_no_semihosting_swi)
```

- 在汇编语言模块中，使用 IMPORT 命令：

```
IMPORT __use_no_semihosting_swi
```

如果在程序中引入了__use_no_semihosting_swi，但最终映像仍链接了 Semihosting 库，链接器会报告如下错误：

```
Error：L6200E：Symbol __semihosting_swi_guard multiply defined (by use_semi.oand use_no_semi.o)
```

为帮助找出这些使用了 Semihosting 的函数，可以使用-verbose 链接选项。这样，在输出结果中，C 库函数将被标以__I_use_semihosting_swi 标记。下面这段连接器的输出显示了使用-verbose 链接选项后的结果。

```
Loading member sys_exit.o from c_a__un.l.
            definition: _sys_exit
            reference : __I_use_semihosting_swi
```

这时，要使程序正确执行，用户必须为标记了的函数提供自己的实现方法。
值得注意的是，链接器不会报告应用程序代码中的任何使用 Semihosting SWI 的函数。只有当从 C 库链接了使用 Semihosting SWI 的函数时才发生错误。

4.3 映像文件存储器映射调整

4.3.1 关于分散加载

华清远见

映像由域（Regions）和输出段（Output Sections）组成。每个域可以有不同的加载地址和执行地址。
分散加载可以更加方便、准确地指定映像存储器映射，为映像组件分组和布局提供了全面控制。它能够描述由载入时和执行时分散在存储器映射中的多个区组成的复杂映像映射。虽然，分散加载可以用于简单映像，但它通常仅用于具有复杂存储器映射的映像。
要构建映像的存储器映射，必须向 armlink 提供以下信息。

- 分组信息：决定如何将各输入段组织成相应的输出段和域。
- 定位信息：决定各域在存储空间的起始地址。

有两种方法可以实现指定映像文件的分组和定位信息：如果映像文件中地址映射关系比较简单，可以使用命令行选项；如果映像文件中地址映射关系比较复杂，可以使用一个配置文件。使用该配置文件可以提供链接器相关的地址映射关系。配置文件又叫 Scatter 文件，是一个文本文件，通过下面的链接选项来实现。

```
-scatter filename
```

1．为分散加载定义的符号

当 armlink 使用 Scatter 文件创建映像时，它创建了一些域相关符号，表 4.2 所示为这些符号的意义。

表 4.2　　域相关符号

符　　号	意　　义
Load$$region_name$$Base	域的载入地址
Image$$region_name$$Base	域的执行地址
Image$$region_name$$Length	执行域字节长段的（4 的倍数）
Image$$region_name$$Limit	执行区末尾地址
Image$$region_name$$ZI$$Base	执行域中 ZI 段的执行地址

Image$$region_name$$ZI$$Length	ZI 输出段的长的（4 的倍数）
Image$$region_name$$ZI$$Limit	执行域中 ZI 段的末尾地址

2．使用 Scatter 文件的优势

链接程序的命令行选项提供了对数据和代码布局的控制，但要实现对布局的全面控制命令行输入的指令是远远不够的。在下面一些情况下，就需要使用 Scatter 文件对映像布局进行控制。

（1）需要实现复杂存储器映射。

系统中的代码和数据必须放在多个不同存储器区域中，这样连接器必须知道哪个段放在哪个储存器空间的详细信息。这种情况下，最好用 Scatter 文件实现代码映像的分散加载。

（2）系统中存在多种不同类型存储器。

许多系统包含多种不同类型存储器，如 flash 存储器、ROM、SDRAM 和快速 SRAM。分散载入描述可以将代码和数据放置在最适合的存储器类型中。例如，中断代码可能放在快速 SRAM 中，以加快中断响应时间，而不频繁使用的配置信息可能放在较慢的 Flash 存储器中。

（3）存储器映射 I/O。

分散载入描述可以将数据精确定位在内存地址中，而避免数据和内存映射外围地址相冲突。

（4）位于固定位置函数。

可以将特定函数放在存储器中的同一个位置，这样即使周围的应用程序已经被修改并重新编译，也可以使具有特定功能的函数地址保持不变。

（5）使用符号识别堆和栈。

可以为堆和栈的位置定义符号，链接应用程序时可以指定该封闭模块的位置。

随着目前嵌入式系统越来越复杂，系统中可能同时使用 Flash、ROM 和 RAM，所以建议在生产系统映像时使用 Scatter 文件。

3．分散加载命令行选项

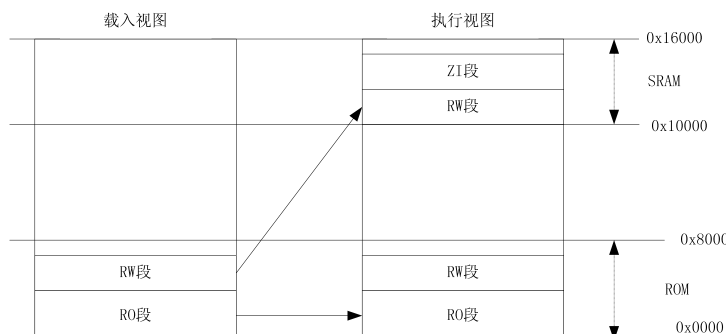
可以使用下面的命令行选项使用分散加载文件：

```
-scatter description_file_name
```

使用该命令可以使连接器使用命令中给出的 description_file_name 文件生成最终的映像文件。

4．简单存储器映像举例

例如，一个实际系统的存储器映射如图 4.7 所示。



华清远见

图 4.7 简单存储器映射

为了实现图 4.7 所示的存储器映射，使用图 4.8 所示的 Scatter 文件。

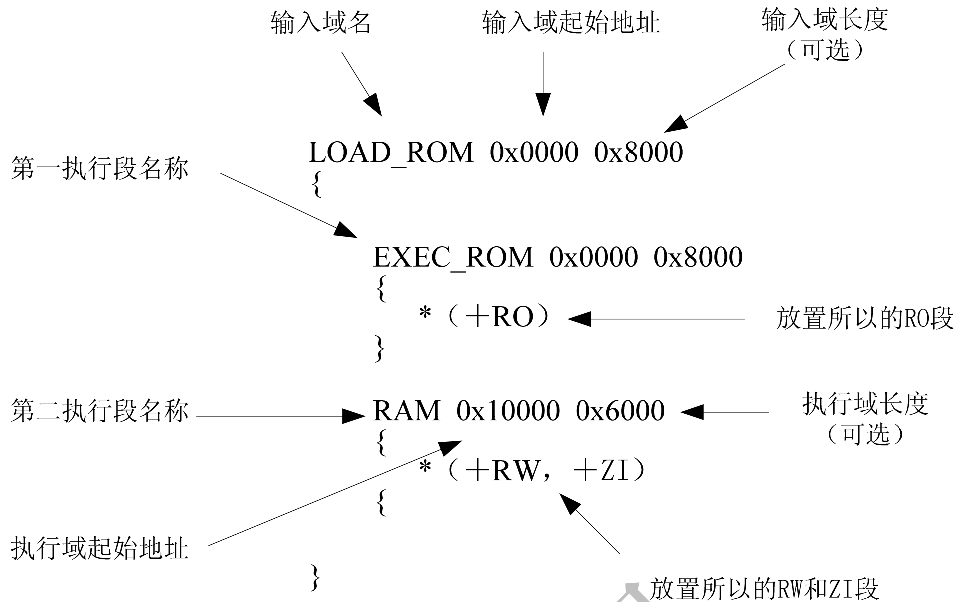
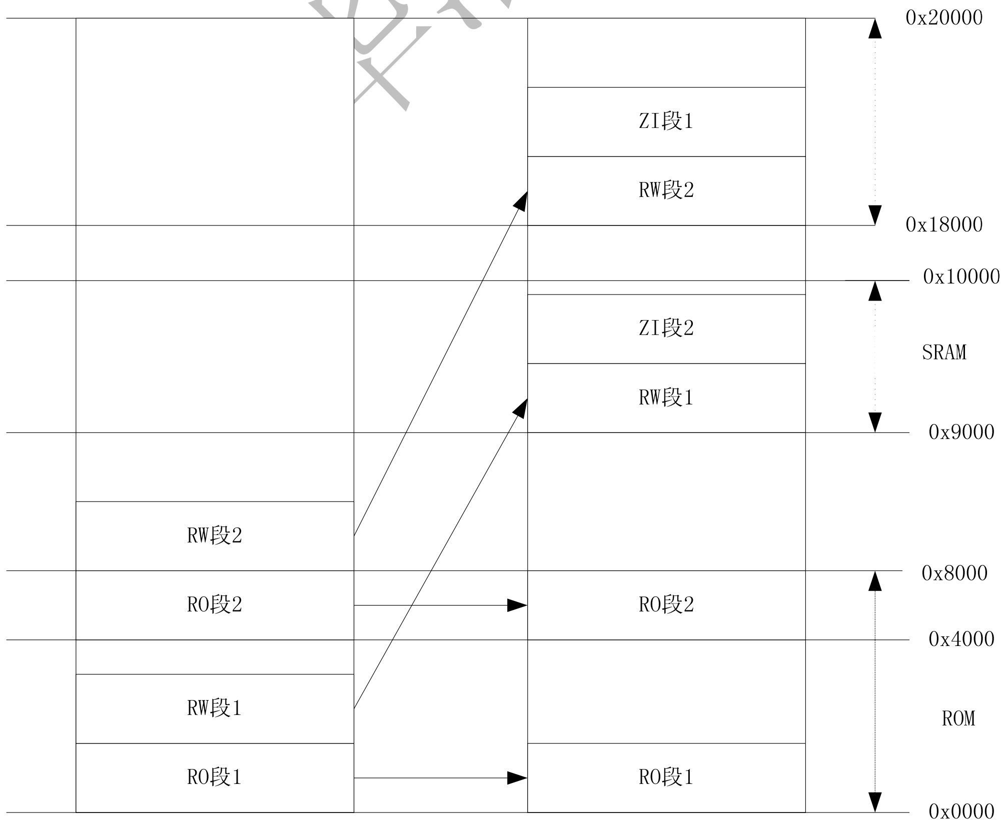


图 4.8 实现简单内存映射的 Scatter 文件

5．负杂存储器映像实现举例

一个复杂存储器映射如图 4.9 所示。



华清远见

图 4.9 复杂存储器映射实例

为了实现图 4.9 的存储器映射，使用图 4.10 所示的 Scatter 文件。

```
LOAD_ROM_1  0x0000                     ← 第一个加载时域的起始地址
{
  EXEC_ROM_1  0x0000                   ← 第一个运行时域的起始地址
     {
       program1.o（+RO）               ← 放置program.o中所以的RO段
     }
     SRAM  0x9000                      ← 运行时域的起始地址
     {
       program1.o（+RW，  +ZI）        ← 放置program.o中所以的RW和ZI 段
     }
}

LOAD_ROM_2  0x4000                     ← 第二个加载时域的起始地址
{
  EXEC_ROM_2  0x4000                   ← 运行时域的起始地址
     {
       program2.o（+RO）
     }
     DRAM  0x18000                     ← 运行时域的起始地址
     {
       program2.o（+RW，  +ZI）
     }
}
```

图 4.10 复杂存储器映射的 Scatter 文件

上面两个例子中，简单存储器映射可以使用命令行选项实现，但第二个复杂存储器映射的例子却只能使用 Scatter 文件实现。

4.3.2 Scatter 文件语法

分散载入描述文件是一个文本文件，它向 armlink 描述目标系统的存储器映射。如果从命令行加载 Scatter 文件，可以使用任意类型的文件扩展名。

在 Scatter 文件中，用户可以指定以下存储器映像内容：

- 每个载入区的载入地址和最大尺寸；
- 每个载入区的属性；
- 从每个载入区派生的执行区；
- 每个执行区的执行地址和最大尺寸；
- 每个执行区的输入节。

描述文件的格式反映出载入区、执行区和输入节的层次结构。

1．BNF 的表示法和语法

所谓 BNF（Backus Naur Format）即 Scatter 文件所用的形式语言。表 4.3 概括了其所用的符号和语法规则。

表 4.3 BNF 语法

符　号	说　明
”	引号用于表示 BNF 语法中的字符被用作普通字符。

华清远见

	例如，定义 B"+"C，它只能替换为模式 B+C。而定义 B+C 可以替换为模式 BC、BBC 或 BBBC
A ::= B	将 A 定义为 B。例如，A::= B"+" | C 表示 A 相当于 B+或 C。 在其组件方面，::=表示法用于定义高级结构。每个组件可能还有一个::=定义，对更简单的组件进行定义。 例如，A::= B 以及 B::= C | D 表示定义 A 相当于模式 C 或 D
[A]	可选元素 A。例如，A::= B[C]D 表示定义 A 可以扩展为 BD 或 BCD
A+	元素 A 可以出现一次或多次。例如，A::= B+表示定义 A 可以扩展为 B、BB 或 BBB 等
A*	元素 A 可以不出现或多次出现
A|B	出现元素 A 或 B，但不能同时出现
（A|B）	元素 A 和 B 组合在一起。 这在使用|操作符时，或重复复杂模式时尤其适用。 例如，A::= (B C) + (D | E)表示定义 A 可以扩展为 BCD、BCE、BCBCD、BCBCE、BCBCBCD 或 BCBCBCE

2. Scatter 文件语法概述

分散加载描述 scatter_description 被定义为一个或多个 load_region_description 模式：

```
Scatter_description ::=
  load_region_description+
```

加载域描述 load_region_description 被定义为载入区名称，可以选择性地在其后跟随属性或尺寸说明符，以及一个或多个执行区描述：

```
load_region_description ::=
  load_region_name (base_address | ("+" offset)) [attributes] [max_size]
  "{"
        execution_region_description+
  "}"
```

执行域描述 execution_region_description 被定义为执行区名称，是一种基址规范，可以选择性地在其后跟随属性或尺寸说明符，以及一个或多个输入段描述：

```
execution_region_description ::=
  exec_region_name (base_address | "+" offset) [attribute_list] [max_size | "-"
length]
        "{"
            input_section_description*
        "}"
```

输入段描述 input_section_description 被定义为源模块选择程序模式，可以在其后选择性地跟随输入节选择程序：

```
input_section_description ::=
  module_select_pattern
  [ "("
        ("+" input_section_attr | input_section_pattern)
        ([","] "+" input_section_attr | "," input_section_pattern))*
   ")" ]
```

图 4.11 所示为一个典型的分散载入描述文件的内容和组织结构。

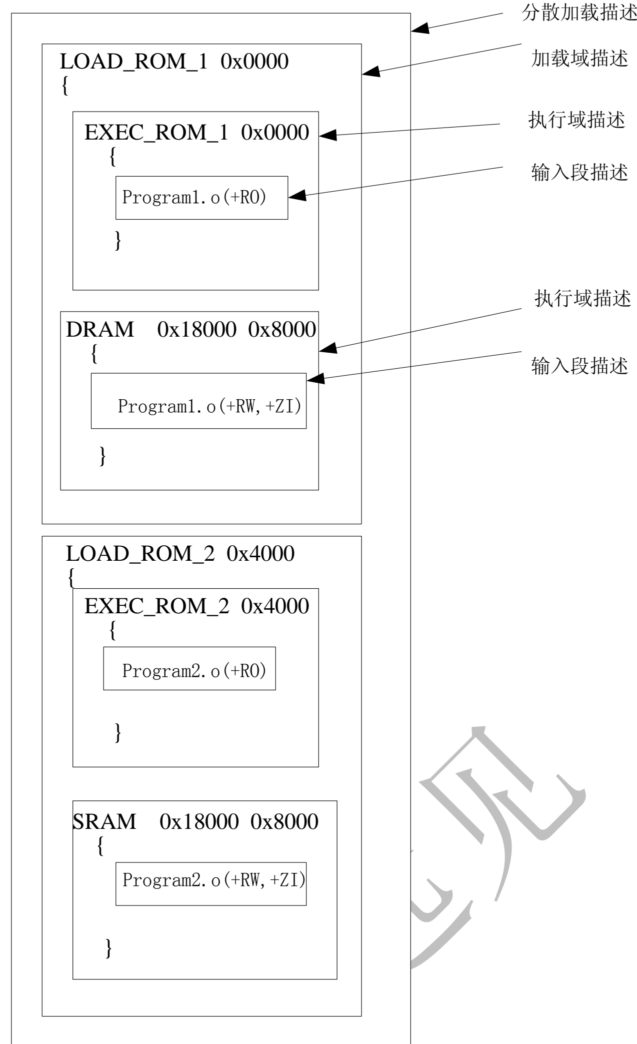


图 4.11 典型的分散载入描述文件的内容和组织结构

3．加载域描述

一个加载域具有以下属性。

- 名称：链接程序使用它识别不同的加载域。
- 基址：载入视图中的代码和数据的起始地址。
- 属性：可选。
- 最大尺寸：可选。
- 执行区列表：这些执行区标识执行视图中模块的类型和位置。

图 4.12 所示为加载域的描述。

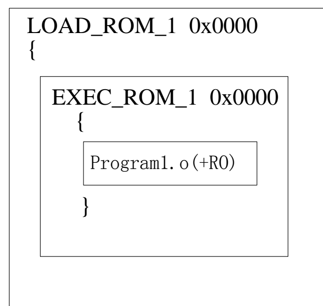


图 4.12 加载域描述

BNF 语法如下：

```
    load_region_description ::=
      load_region_name  (base_address  | ("+" offset)) [attribute_list] [ max_size
    ]
          "{"
            execution_region_description+
          "}"
```

语法说明如下。

① load_region_name 为加载域的名称。只有前 31 个字符有效。该名称仅用于识别每个域。值得注意的是，load_region_name 与执行域 exec_region_name 不同，load_region_name 不用于生成 Load$$region_name 符号。

② base_address 是区中对象的链接地址。base_address 必须是一个字对齐数值。

③ +offset 描述基址，它从前一个加载域的末尾偏移 offset 个字节。offset 的值必须能被 4 整除。如果是第一个加载域，则+offset 表示该域的基地址是从 0 之后的 offset 字节开始。

④ attribute_list 指定如下加载域内容的属性。

- PI：位置独立。
- RELOC：可重定位。
- OVERLAY：重叠。
- ABSOLUTE：绝对地址。
- NOCOMPRESS：代码不被压缩。

可以指定这些属性中的一项（除 NOCOMPRESS 外，其他四项属性为互斥关系）。默认的加载域属性是 ABSOLUTE。具有 PI、RELOC 或 OVERLAY 属性之一的加载域可以有重叠的地址范围。对于 ABSOLUTE 加载域，armlink 不允许重叠的地址范围。OVERLAY 关键字允许在同一个地址有多个执行区。

⑤ max_size 指定加载域的最大尺寸（如果指定了可选的 max_size 值，但分配给该区的字节超过 max_size 字节，armlink 将生成错误）。

⑥ execution_region_description 指定执行区名称、地址和内容。

华清远见

4．执行域描述符

执行域具有以下一些属性：

- 域名称；
- 执行域基地址（支持绝对地址的或相对地址的）；
- 执行域的最大尺寸（可选）；
- 指定执行域属性；
- 一个或多个输入段描述（放在本执行区中的模块）。

图 4.13 所示为一个典型的执行域描述。

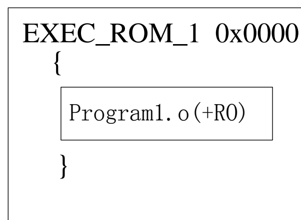


图 4.13　执行域描述

执行域描述符中的 BNF 语法如下：

```
    execution_region_description ::=
      exec_region_name (base_address | "+" offset) [attribute_list] [max_size | "-"
```

```
    length]
        "{"
            input_section_description+
        "}"
```

其语法说明如下。
① exec_region_name 为执行域命名（只有前 31 个字符有效）。
② base_address 是域中对象的链接地址。base_address 必须是字对齐的。
③ +offset 描述基址，它从前一个执行区的末尾偏移 offset 个字节。offset 的值必须能被 4 整除。如果前面没有执行区（即这是载入区中的第一个执行区），则+offset 表示基址从它所在的载入区的基址之后 offset 个字节开始。如果使用+offset 格式，并且所在的加载域具有 RELOC 属性，则执行区继承该 RELOC 属性。但是，如果使用固定的 base_address，则随后出现的 offset 不继承 RELOC 属性。
④ attribute_list 指定以下执行区内容的属性。

- PI：位置独立。
- OVERLAY：重叠。
- ABSOLUTE：绝对地址。域的执行地址由 base_designator 指定。
- FIXED：固定地址。执行域的加载地址和执行地址都由 base_designator 指定。base_designator 必须是绝对基址，或者偏移量为+0。
- EMPTY：它在执行区中保留一个已知长度的空白存储器块，通常用作堆或栈。
- PADVALUE：指定填充字的默认值，如果在域定义中指定了该属性，则必须为该属性赋值。使用该属性的例子如下：

```
    EXEC 0x10000 PADVALUE 0xffffffff EMPTY ZEROPAD 0x2000
```

通过该 Scatter 文件描述符，创建了一个长度为 0x2000 的域，该域中的所有内容用 0xffffffff 填充。

- ZEROPAD 0：初始化一块内容全为 0 的内存区域，并将其作为一个输入段填充到 ELF 映像文件中。这样减少了在运行时将某段内存初始化为 0 的操作。
- UNINIT：指示该段为不能被初始化为 0。

⑤ max_size 为可选的参数，如果分配给域的存储器超过 max_size 字节，则它指示 armlink 生成错误。
⑥ -length 表示如果指定的长度为负值，则 base_address 是域的结束地址。它通常与 EMPTY 一起使用，以表示在存储器中变小的栈。
当确定执行域属性时，注意以下几点。
（1）PI、OVERPLAY、FIXED 和 ABSOLUTE 为并列关系属性，某一个执行域只能为这四种属性之一。如果没有指定，ABSOLUTE 为其默认属性。
（2）使用+offset 格式的 base_designator 的执行区继承前一个执行区的属性（如果它是加载域中的第一个执行区，则继承所在加载域的属性，），或者具有 ABSOLUTE 属性。
（3）不能为执行域显式指定 RELOC 属性。该属性只能从前面的执行域或父区继承才能具有 RELOC 属性。
（4）被指定了 PI 或 OVERLAP 属性的执行域，可以有重叠的地址范围。但对于 ABLOUTE 和 FIXED 属性的执行域，ARM 编译器不允许有重叠的地址范围。
（5）RW 段默认使用压缩属性。如果不想连接器对该段进行压缩，必须在 Scatter 文件中使用 NOCOMPRESS 显示声明。
（6）UNINIT 指定执行区中的 ZI 输出节（如果有）不被初始化为 0。使用它可以创建包含未初始化数据或存储器映射 I/O 的执行区。

5．输入段描述符

输入段由以下部分组成。

- 模块名称（目标文件名称、库成员名称或库文件名称）。模块名称可以使用通配符。
- 输入段名称或输入节属性，如 READ-ONLY 或 CODE。

图 4.14 显示了输入段描述符的基本组成。

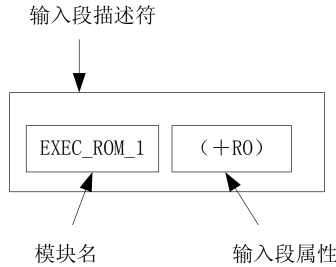


图 4.14 输入段描述符

BNF 语法为：

```
input_section_description ::=
  module_select_pattern
      ["("
          ("+" input_section_attr | input_section_pattern)
          ([","] "+" input_section_attr | "," input_section_pattern))*
       ")"]
```

其语法说明如下。

① module_select_pattern：这是由文字文本构成的模式。"*" 通配符匹配 0 个或多个字符，而 "?" 匹配任何单个字符。匹配不区分大小写。

使用*.o 可以匹配所有对象。使用*可以匹配所有目标文件和库。

当满足下列条件之一时，连接器认为 module_selector_pattern 与输入段的匹配。

- 包含输入段的目标文件与 module_selector_pattern 匹配。
- 包含输入段的库成员名称（不带路径名）与 module_selector_pattern 匹配。
- 从其中提取段的库全名（包含路径名）。如果名称包含空格，使用通配符可以简化搜索。例如，使用*libname.lib 匹配 C:\lib dir\libname.lib。

另外，ARM 链接器支持特殊的模块选择程序模式 ".ANY"，允许将输入节分配给执行区，而无需考虑其父模块。使用.ANY 以任意分配方式填充执行区。

② input_section_attr：输入段属性符定义了一个用逗号隔开的模式类别。该类表中的每个模式定义了输入段名称或输入段属性匹配方式。当匹配模式使用输入段名称时，它前面必须使用符号 "+"，而符号 "+" 前面紧接的逗号可以省略。

输入段属性不区分大小写。可以是下列属性之一：

- RO-CODE；
- RO-DATA；
- RO，同时选择 RO-CODE 和 RO-DATA；
- RW-DATA；
- RW-CODE；
- RW，同时选择 RW-CODE 和 RW-DATA；
- ZI；
- ENTRY，包含 ENTRY 点的节。

可以识别以下同义词：

- CODE 代表 RO-CODE；
- CONST 代表 RO-DATA；
- TEXT 代表 RO；
- DATA 代表 RW；
- BSS 代表 ZI。

华清远见

可以识别以下伪属性：

- FIRST ；
- LAST。

如果对输入段的排列顺序有特殊的要求，如特定的输入段必须是域中的第一个输入节，而包含校验和的输入段必须是最后一个输入段，可以使用 FIRST 和 LAST 标记执行区中的第一个和最后一个段。

FIRST 或 LAST 伪属性必须放在属性列表的最后。

特殊的模块选择程序模式“.ANY”允许在不考虑其父模块的情况下，将输入段分配给执行域。使用一个或多个“.ANY”模式以任意分配方式填充执行域。在大多数情况下，使用单个“.ANY”相当于使用“*”模块选择属性。

在分散载入描述文件中不能使用两个“*”选择属性。但是，可以使用两个变形的选择程序，例如，*A 和 *B，也可以将.ANY 选择属性与模块选择属性一起使用。*模块选择属性的优先级比.ANY 高。如果删除了文件中包含*选择属性的部分，.ANY 选择属性才能在链接时起作用。

在解析所有其他（非.ANY）输入段描述，并且将输入段分配给最匹配的执行区之后，才解析使用.ANY 模块选择程序模式的 input_section_descriptions。如果有一个以上.ANY 模式，则链接程序尽可能多地填充第一个.ANY，然后开始填充下一个.ANY。

每个未被分配的剩余输入段将被分配给具有以下特性的执行区：

- 最大的剩余空间（由 max_size 的值和已分配给该区的输入段的尺寸确定）；
- 匹配.ANY 的 input_section_description；
- 与输入段的存储器属性相匹配的存储器访问属性（如果有）；
- input_section_pattern。

4.3.3 Scatter 文件典型用法

1．创建启动域

所谓启动域就是加载地址和执行地址相同的域。系统执行的初始入口点必须要在启动域中，否则链接器将报告以下错误：

```
Entry point (0x00000000) lies within non-root region ER_ROM
```

在 Scatter 文件中确定启动域可以使用下面两种方法。

（1）使用 ABSOLUTE 设置执行区属性，并且对第一个执行区及其所在的加载区使用相同的地址。为确保执行域地址和加载域地址相同，可以将加载域的起始地址和执行域的起始地址设为相同的值；或者将第一个执行域的地址偏移量设为 0。

下面的例子，指定了一个驱动域。

```
BOOT  0x0000 ; 加载域的起始地址在 0x0
{
EXER  0x0000 ; 指定加载域和执行域的地址相同
{
* (+RO) ;必须将启动域包含在内
}
;其他执行域
}
```

（2）使用 FIXED 执行域属性，确保指定域的载入地址和执行地址相同。

下面的例子显示了使用 FIXED 属性，将执行域的起始地址固定在 ROM 中。

```
BOOT  0x0000 ; 加载域的起始地址在 0x0
{
EXER  0x0000 ; 指定加载域和执行域的地址相同
```

```
{
* (+RO) ;必须将启动域包含在内
}

  EXER_INIT  0x8000  FIXED
  {
     init.o(+RO)
  }
}
```

（3）如果使用分散加载，负责创建执行域的代码和数据不能将其自身复制到另一位置，因此启动域必须包含以下内容。

- _main.o 和_scatter*.o：包含复制代码和数据的代码。
- Region$$Table 和 ZISection$$Table 段，包含要复制代码和数据的地址。
- _dc*.o 执行代码压缩。

可以使用 armlinker 产生的 InRoot$$Sections 符号放置启动代码。因为这些代码被定义为只读属性，如果 Scatter 文件中包含了"*(+RO)"，则表示启动域中包含了这些代码。或者显式地使用 InRoot$$Sections 符号在 Scatter 文件中对以上代码进行配置。

下面的例子显示了如何在 Scatter 文件中使用 InRoot$$Sections 链接符号，放置启动域。

```
LOADREG 0x8000                     ;
{
    ROOT  0x8000
    {
           * (InRoot$$Sections)    ; 放置启动域
    }
    OTHER  0x100000
    {
           * (RO,+RW,+ZI)
    }
                              ; 其他 Scatter 文件描述
}
```

2．为执行域确定固定地址

可以在执行区分散加载描述中使用 FIXED 属性来创建根区，该根区在固定地址载入和执行。

FIXED 可以用于在单一加载域内（因此通常用于单个 ROM 设备）创建多个根区。

例如，使用 FIXED 属性将函数或数据块（如常数表或校验和）放在 ROM 中的固定地址，这样就可以使用指针很方便的对其进行访问。

下面的例子显示了如何放置单个目标内容。

```
LOADREG1 0x0 0x10000
{
    EXECREG1 0x0 0x1000                ; 启动域，包含初始化代码
    {                                  ; 将初始化代码放在 0x0 地址
        init.o (Init, +FIRST)
        * (+RO)                        ; 随后排放余下的只读数据
    }
    RAM  0x400000  0x2000          ; 将可读可写数据放在 0x400000 地址
    {
        * (+RW +ZI)
    }
```

```
        DATABLOCK 0x4FF00 FIXED 0xFF  ; 执行域放在 0x4FF00 地址
        {                                 ; 限制该域的最大长度为 0xFF
            data.o(+RO-DATA)              ; 将只读数据放在 0x1FF00 和 0x1FFFF 之间
        }
    }
```

通过上面的 Scatter 文件，可以将初始化代码放在 0x0 处，其后是其他 RO 代码和除了 data.o 对象中的 RO 数据之外的所有 RO 数据。所有全局的 RW 变量放在 RAM 中 0x400000 处。最好将 data.o 的 RO-DATA 只读数据表放在地址 0x4FF00 处，并指定其最大长度为 0xFF。

上例中将代码或数据对象放在其各自的源文件中，然后放置目标文件域，这些操作方式是 ARM 公司建议的标准编码方式。然而，方便起见，可以使用编译指示#pragma 和分散载入描述文件放置已命名的域。下面的例子创建模块 dump.c 并显式命名域。

```
    // file dump.c
        int a = 10;                       // 放入数据域
        short b[100];                  // 放入 bss 段
        int const c[3] = {1,2,3};      // 放入.constdata 段
        int func1(int a) {return a*1;}    // 放入 .text 段
        #pragma arm section rwdata = "foo", code ="foo"
        int x = 5;                             // 在 foo 的数据域
        char *s = "abc";                       // s3 在 code 段, "abc" 在 .constdata
        int func2(int x)  {return x+1;}        // 放入 foo 的.text 段
        #pragma arm section code, rwdata  // 返回
```

使用下面的 Scatter 文件指定上面的代码在内存中的放置位置。如果代码和数据段的名称相同，则首先放置代码段。

```
    FLASH 0x10000000 0x2000000
    {
        FLASH 0x10000000 0x2000000
        {
            init.o (Init, +First)      ; 放置初始化代码
            * (+RO)                       ;
        }
        RAM 0x0000
        {
            vectors.o (Vect, +First)      ;放置向量表
            * (+RW,+ZI)                   ;
        }
        DUMP 0x08000000
        {
            dump.o (foo)                  ;
        }
    }
```

通过上面的 Scatter 文件，将 init 中的初始化段放在 0x10000000 地址，并将除 foo 外的只读数据 func1 和 c[]放在该初始段的后面，接下来的执行域 RAM 放置向量表，最后的 DUMP 域放置由#pragma 指定的段 dump。

3．在代码映像中保留空白域

可以在 Scatter 中使用 EMPTY 属性为栈保留一个空白存储器块。该存储块不构成载入区的一部分，但指定在执行时使用。由于它创建为虚 ZI 区，所以 armlink 使用以下符号访问它：

- 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 Image$$region_name$$ZI$$Limit 视为域的结束地址。它应该是绝对地址，而不是相对地址。下面例子显示了如何在 Scatter 文件中预留一个空白区域。

```
LOADREGION  0x700000                   ; 加载域的起始地址在 0x700000
{                                      ;
    STACK 0x7000000 EMPTY -0x10000     ; 该域的结束地址为 0x700000，因为其长度为负
                                  ;
region
                                  ;
    {
                                  ; 预留空白区放置栈
    }
    HEAP +0 EMPTY 0x10000              ; 栈的起始地址在上个预留区域介绍地址
                                  ;
                                  ;
    {
                                  ; 预留空白区域放置堆
    }
    ; rest of scatter description...
}
```

在上面的例子中定义了一个执行域 STACK 0x7000000 EMPTY -0x10000，它从地址（0x7000000～0x1000）开始，在地址 0x7000000 结束。

在此示例中，链接程序生成符号：

Image$$STACK$$ZI$$Base = 0x6ff0000
Image$$STACK$$ZI$$Limit = 0x7000000
Image$$STACK$$ZI$$Length = 0x1000
Image$$HEAP$$ZI$$Base = 0x7000000
Image$$HEAP$$ZI$$Limit = 0x7010000
Image$$HEAP$$ZI$$Length = 0x1000

EMPTY 属性仅适用于执行区。如果在载入区定义中使用 EMPTY 属性，则链接程序生成警告信息并忽略该属性。链接程序检查用于 EMPTY 区的地址空间，不与任何其他执行区重叠。

4. 使用 OVERLAY 关键字

在 ARM 以前的编译器中，没有提供地址空间的重叠管理。如果有运行时域地址空间重叠，需要用户自己提供地址空间重叠的管理机制。但在 RVDS 的编译器中，提供了运行时域属性关键字 OVERLAY，用户可以使用该关键字生成自己的重叠空间。

下面例子显示了如何使用 OVERLAY 关键字，生成运行时域的重叠空间。

```
LOADREG 0x8000
{
                                  ;
```

```
    STATIC_RAM 0x0                    ; 静态 RAM 区，包含大部分的 RW 和 ZI
    {
          * (+RW,+ZI)
    }
    OVERLAY_A_RAM 0x1000 OVERLAY      ; 重叠区...
    {
          module1.o (+RW,+ZI)
    }
    OVERLAY_B_RAM 0x1000 OVERLAY
    {
          module2.o (+RW,+ZI)
    }
                                      ;
}
```

5．在 Scatter 文件中使用预处理伪操作

可在 Scatter 文件的第一行加上需要编译器进行预处理的操作。语法格式如下：

```
#! <preprocessor> [pre_processor_flags]
LOAD_FLASH ( 0x8000 + ( 0x2 * 0x400 )) ;
```

例如：

```
#! armcc  -E
```

连接器可以对预处理的表达式进行简单的计算，可以识别简单的运算符，如＋、－、×、/、AND 和 OR。如：

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

同时，也可以在 Scatter 文件头加一些预处理的伪操作，如：

```
#define ADDRESS 0x20000000
#include "include_file_1.h"
#define BASE_ADDRESS 0x8000
#define ALIAS_NUMBER 0x2
#define ALIAS_SIZE 0x400
```

关于在 Scatter 文件中，使用预处理的更详细的信息，请参见 ARM 相关文件。

4.3.4 等效的简单映像分散载入描述

前面介绍了分散加载的命令行选项，如-ro-base、-rw-base、-reloc、-split、-ropi 和-rwpi。但在实际编程时，因为使用 Scatter 文件可以产生更清晰的内存映像视图，所以最好使用 Scatter 文件对映像进行加载。
本节详细介绍如何将各分散加载的命令行选项替换为 Scatter 文件。

1．-ro-base address 选项的替换

使用-ro-base address 命令行链接产生的内存映像由一个加载域和 3 个执行域组成。执行域放在存储器映像中的相邻位置。
选项中的 address 指定了加载域和第一个执行域的起始地址（加载域和第一个执行域的起始地址相同）。
下面的例子显示了与“-ro-base 0x8000”命令行选项等价的 Scatter 文件。

```
LOADREG 0x8000  ; 定义加载域的起始地址 0x8000
```

```
{                  ;
    ROM  +0        ; 定义第一个执行域的起始地址，该地址与加载域的起始地址相同为 0x8000
                   ;
    {
        *(+RO)     ; 该域放置所有的 RO 段
    }
    RAM_RW  +0     ; 定义第二个执行域，起始地址为 0x8000＋ROM 段大小
                   ;
    {
        *(+RW)     ; 将所有的 RW 代码放置在该段
    }
    RAM_ZI +0      ;定义 ZI 段
                   ;ZI 段的起始地址为 0x8000＋ROM 段的大小＋RAM_RW 段的大小
                   ;.
    {
        *(+ZI)     ; 放置所有的 ZI 段
    }
}
```

上例中的 Scatter 文件创建的映像由一个加载域和 3 个执行域组成。加载域的起始地址为 0x8000。3 个执行域分别为 ROM、RAM_RW 和 RAM_ZI，它们分别包含 RO、RW 和 ZI 输出段。RO 和 RAM_RW 为启动域，RAM_ZI 在执行时动态创建。ROM 的执行地址是 0x8000，通过对执行区描述使用+offset 格式的基址指定程序，所有 3 个执行域在存储器映射中相邻放置，即前一个执行域的末尾放置后一个执行域。

如果链接程序时，将-ro-base 选项和-ropi 混合使用，则可以生成位置无关代码。

下面的例子显示了与-ro-base 0x8000 -ropi 等效的 Scatter 文件。

```
LOADREG  0x8000  PI   ; 加载域的地址为 0x8000，并指定该加载域的属性为 PI
{
    ROM  +0           ; 第一执行域的地址为 0x8000，而且该执行域继承了加载域的 PI 属性
                      ;所有该域的执行地址是可变的
    {
        *(+RO)        ; 放置所有的 RO 段
    }
    RAM_RW +0 ABSOLUTE    ; 使用 ABSOLUTE 属性代替 PI 属性
    {
        *(+RW)        ; 放置 RW 段
    }
    RAM_ZI +0
    {
        *(+ZI)
    }
}
```

执行域 ROM 从 LOADREG 加载域继承 PI 属性。下一个执行域 RAM_RW 被标记为 ABSOLUTE，所以其不再具有 PI 属性。另外，因为 RAM_ZI 域使用了＋0 的偏移量，所以它从 RAM_RW 域继承 ABSOLUTE 属性。

2．--ro-base 和--rw-base 选项的替换

使用--ro-base 和--rw-base 选项链接的映像也由一个加载域和 3 个执行域组成，它与类型 1 生成的映像十分相似，只是此类映像的 RW 执行区与 RO 执行区不相邻。
在-ro-base 选项中指定加载域的起始地址，在--rw-base 选项中指定执行域的地址。
下面的例子显示与使用-ro-base 0x8000 -rw-base 0x040000 等效的分散载入描述。

```
LOADREG  0x8000       ;定义加载域的起始地址为 0x8000
{
    ROM_RO +0         ; 定义第一个执行域的起始地址为 0x8000

    {
        * (+RO)           ; 在该域中放置所有的 RO 段
    }
    RAM_RW 0x040000       ; 第二个执行域名为 RAM_RW，起始地址为 0x40000
    {
        * (+RW)           ; 放置所有的 RW 段
    }
    RAM_ZI +0

    {
        * (+ZI)           ;放置所有的 ZI 段
    }
}
```

该 Scatter 文件创建的映像有一个名为 LOADREG 的加载域，载入地址是 0x8000。该映像有 3 个执行区，分别为 ROM、RAM_RW 和 RAM_ZI，它们分别包含 RO、RW 和 ZI 输出段。其中，RO 域是启动域，执行地址是 0x8000，RAM_RW 执行域与第一个执行域 RAM_RW 不相邻。其执行地址是 0x040000。紧随其后的执行区 RAM_ZI 放置所有的 ZI 数据。
另外，也可以将-rw-base 和位置无关选项-rwpi 配合使用，将 RW 输出节的执行区标记为位置独立。
下面的例子显示了使用-ro-base 0x8000 -rw-base 0x40000 -rwpi 等效的 Scatter 文件。

```
LOADREG  0x0x8000    ; 定义加载域的起始地址为 0x8000
{
    ROM  +0          ;定义第一执行域，其起始地址为 0x8000

    {
        *(+RO)            ; 放置所有 RO 段
    }
    RAM_RW  0x40000  PI   ;设置第二执行域的属性为 PI 属性
    {
        *(+RW)
    }
    ER_ZI +0         ; 继承了 PI 属性
    {
        *(+ZI)
    }
}
```

第一个执行域 ROM 从加载域 LOADREG 继承 ABSOLUTE 属性。第二个执行区 RAM_RW 标记为 PI 属性。另外，因为 ER_ZI 区的偏移为+0，所以它从 RAM_RW 区继承 PI 属性。

3．--reloc-split 选项的替换

使用-split 选项生成的映像由两个加载域和 3 个执行域组成。
使用以下的链接选项重新分割并定位加载域。

- -reloc

组合使用-reloc -split 生成具有两个加载域的映像，并且使加载域具有 RELOC 属性。

- -ro-base address1

指定包含 RO 输出段的域的载入地址和执行地址。

- -ro-base address2

指定包含 RW 输出段的域的载入地址和执行地址。

- -split

将默认的单一加载域（包含 RO 和 RW 输出段的加载域）分成两个加载域。一个载入域包含 RO 输出段，另一个包含 RW 输出段。
下面的例子显示了与使用-ro-base 0x8000 -rw-base 0x040000 -split 等效的 Scatter 文件。

```
LOADREG1  0x8000     ; 指定第一个加载域的起始地址为 0x8000
{
    ROM  +0
    {
        *(+RO)
    }
}
LOADREG2  0x040000       ; 第二个加载域的起始地址为 0x40000
{
    RAM_RW +0
    {
        *(+RW)          ; 放置所有的 RW 段
    }
    RAM_ZI +0
    {
        *(+ZI)
    }
}
```

使用上例中的 Scatter 文件创建的内存映像有两个加载域，分别为 LOADREG1 和 LOADREG2，它们的起始地址分别为 0x8000 和 0x040000。
该映像文件有 3 个执行域，分别为 ROM、RAM_RW 和 RAM_ZI，它们分别包含 RO、RW 和 ZI 输出段。
ROM 的执行地址是 0x8000。
RAM_RW 执行域与 ROM 不相邻。其执行地址是 0x040000。
执行域 RAM_ZI 紧随 RAM_RW 域放置。
可以使用-reloc 选项和-split 选项配合使用，指定两个加载域具有 RELOC 属性。
下面的例子显示与使用-ro-base 0x8000 -rw-base 0x040000 -reloc -split 等效的 Scatter 文件。

```
LOADREG  0x010000 RELOC
{
     ROM  + 0
     {
         * (+RO)
     }
}
LOADREG  0x040000 RELOC
```

```
{
    RAM_RW + 0
    {
        * (+RW)
    }
    RAM_ZI +0
    {
        * (+ZI)
    }
}
```

4.4 复位和初始化

任何运行在实际硬件上的嵌入式应用程序，都必须在启动时实现一些基本的系统初始化。本节将对此予以详细讨论。

4.4.1 初始化序列

图 4.15 显示了一个适用于 ARM 嵌入式系统的初始化序列。

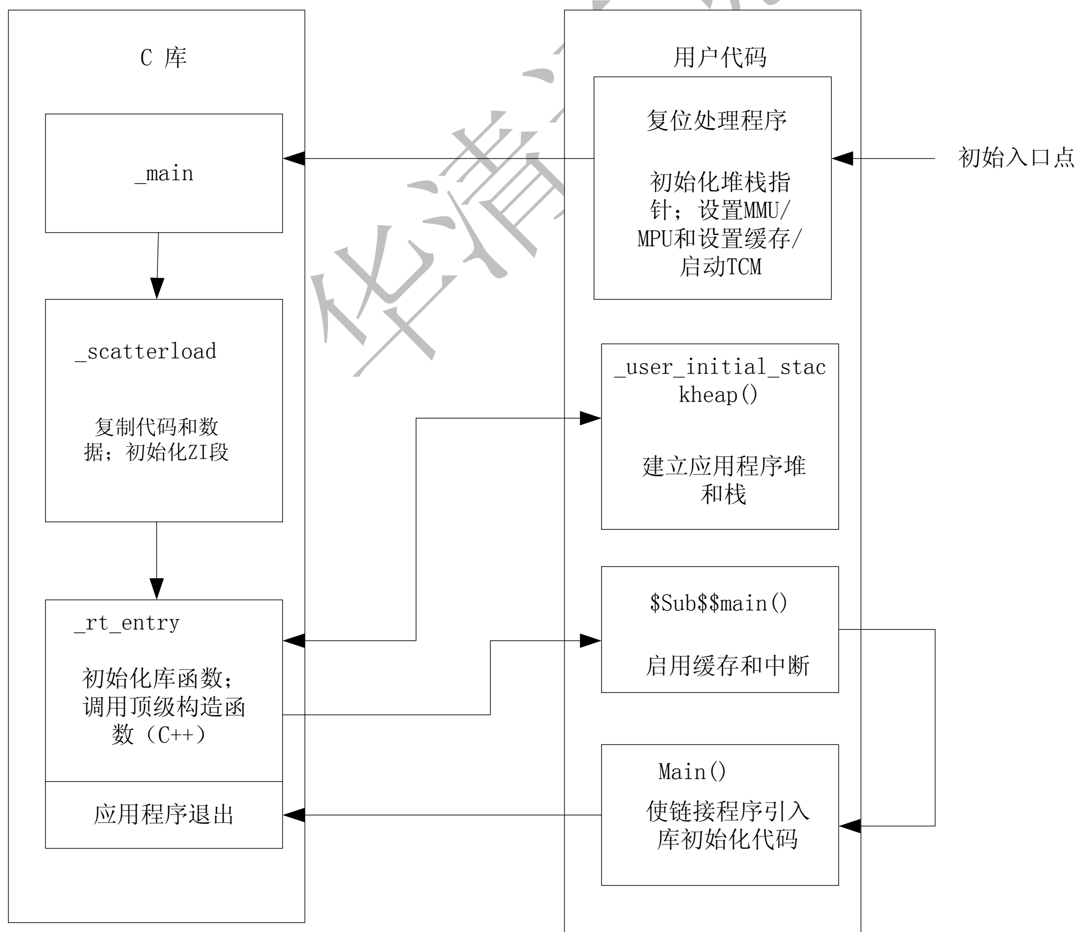


图 4.15 ARM 嵌入式系统的初始化序列

系统启动时立即执行复位处理程序，然后进入$Sub$$main()的代码执行。

复位处理程序是用汇编语言编写的代码块，它在系统复位时执行，完成系统必须初始化操作。对于具有局部存储器的内核，如 Caches、紧密耦合存储器（TCM）、存储管理单元（MMU）和存储器保护单元（MPU）等，在初始化过程这一阶段完成必要的配置。复位处理程序在执行之后，通常跳转到__main 以开始 C 库的初始化序列。

4.4.2 向量表

所有的 ARM 系统都有一个向量表（vector table）。向量表不是初始化序列的一部分，但是，对每个要处理的异常，它必须存在。这些地址通常包含以下形式的跳转指令。

- B<address>：该条指令实现了相对于 pc 的跳转。
- LDR pc，[pc，offset]：这条指令将异常处理程序的入口地址从存储器装载到 PC。该地址是一个 32 位的绝对地址。由于有额外的存储器访问，装载跳转地址会使分支跳转到特定处理程序，给系统执行带来延时。不过，可以使用这种方法跳转到存储空间内的任意地址。
- MOV pc，＃immediate：将一个立即数复制到 PC。使用该指令可以跨越整个地址空间，但是受到地址对齐问题的限制。这个地址必须是一个由 8 位立即数循环右移偶数次得到的。

另外，也可以在向量表中使用其他类型的指令。例如，FIQ 处理程序可以从地址 0x1c 处开始执行。因为它位于向量表的最后，这样 FIQ 处理程序就可以不用跳转，立即从 FIQ 向量地址处开始执行。

下面的例子显示了一个使用 LDR 指令的向量表装载过程。

```
;***********************************
;* VECTOR TABLE                    *
;***********************************
    AREA    vectors, CODE
   ENTRY

    ; Define standard ARM vector table

INT_Vectors
    LDR     PC, INT_Reset_Addr
    LDR     PC, INT_Undef_Addr
    LDR     PC, INT_Software_Addr
    LDR     PC, INT_Prefetch_Addr
    LDR     PC, INT_Data_Addr
    LDR     PC, INT_Reserved_Addr
    LDR     PC, INT_IRQ_Addr
    LDR     PC, INT_FIQ_Addr
```

在向量表的入口处要有 ENTRY 标识。该标识通知链接程序该代码是一个可能的入口点，因而在链接时，不能被清除。

4.4.3 ROM/RAM 重映射

启动时，0x0 处必须要有一条有效指令，因此，复位时 0x0000 地址必须为非易失性存储器，如 ROM 或 Flash。可以将 ROM 定位在 0x0 处。但是，这样配置有几个缺点。首先 ROM 存取速度通常较 RAM 要慢，当跳转到异常处理程序时，系统性能可能会大受影响。其次，如将向量表放于 ROM 中，则运行时不能修改。

存储器地址重映射（Memory Remap）是当前很多先进控制器所具有的功能。所谓地址重映射就是可以通过软件配置来改变一块存储器物理地址的一种机制或方法。

当一段程序对运行自己的存储器进行重映射时，需要特别注意保证程序执行流程在重映射前后的承接关系。实现重映射的关键就是要使程序指针在 remap 以后能继续往下得到正确的指令。本书中介绍两种实现重映射的机制，不同的系统可能会有多种灵活的 remap 方案，用户在具体实现时要具体分析。

1．先搬移再映射（Remap after Copy）

图 4.16 所示为一种典型的存储器地址重映射情况。

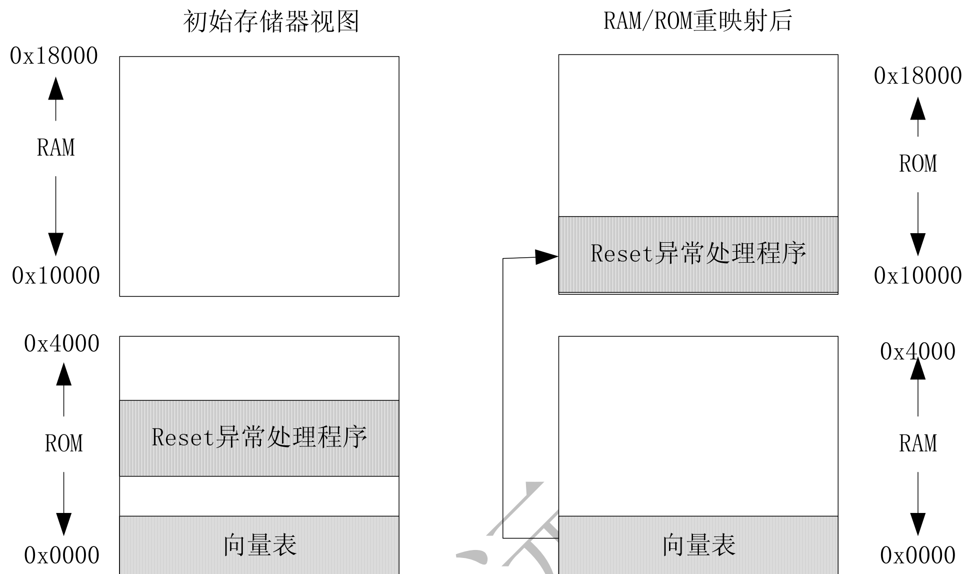


图 4.16 ROM/RAM 重映射（1）

原来 RAM 和 ROM 各有自己的地址，进行重映射以后 RAM 和 ROM 的地址都发生了变化。这种情况下，可以采用以下方案。

① 上电后，从 0x0 地址的 ROM 开始往下执行。
② 根据映射前的地址，对 RAM 进行必要的代码和数据拷贝。
③ 拷贝完后，进行 remap 操作。
④ 因为 RAM 在 remap 前准备好了内容，使得 PC 指针能继续在 RAM 里取到正确的指令。

2．先映射后搬移（Copy after Remap）

系统上电后的默认状态是 0x0 地址上放有 ROM。这块 ROM 有两个地址：从 0 起始和从 0x10000 起始，里面存储了初始化代码。当进行地址 remap 以后，从 0x0 起始的地址被定向到 RAM 上，ROM 则只保留有唯一的从 0x10000 起始的地址。

如果存储在 ROM 里的复位异常处理程序（Reset-Handler）一直在 0x0～0x4000 的地址上运行，则当执行完 remap 以后，下面的指令将从 RAM 里预取，这必然会导致程序执行流程的中断。根据系统特点，可以用下面的办法来解决这个问题。

① 上电后系统从 0x0 地址开始自动执行，设计跳转指令在 remap 发生前使 PC 指针指向 0x10000 开始的 ROM 地址中去，因为不同地址指向的是同一块 ROM，所以程序能够顺利执行。
② 这时候 0x0～0x4000 的地址空间空闲，不被程序引用，执行 remap 后把 RAM 引进。因为程序一直在 0x10000 起始的 ROM 空间里运行，remap 对运行流程没有任何影响。
③ 通过在 ROM 里运行的程序，对 RAM 进行相应的代码和数据拷贝，完成应用程序运行的初始化。

图 4.17 显示了 ROM 和 RAM 重映射的第二种解决方案。

集团官网：www.hqyj.com 嵌入式学院：www.embedu.org 企业学院：www.farsight.com.cn 北京 | 上海 | 深圳 | 成都 | 南京 | 武汉 | 西安 | 广州
移动互联网学院：www.3g-edu.org 物联网学院：www.topsight.cn 咨询电话：400-706-1880，15010390966

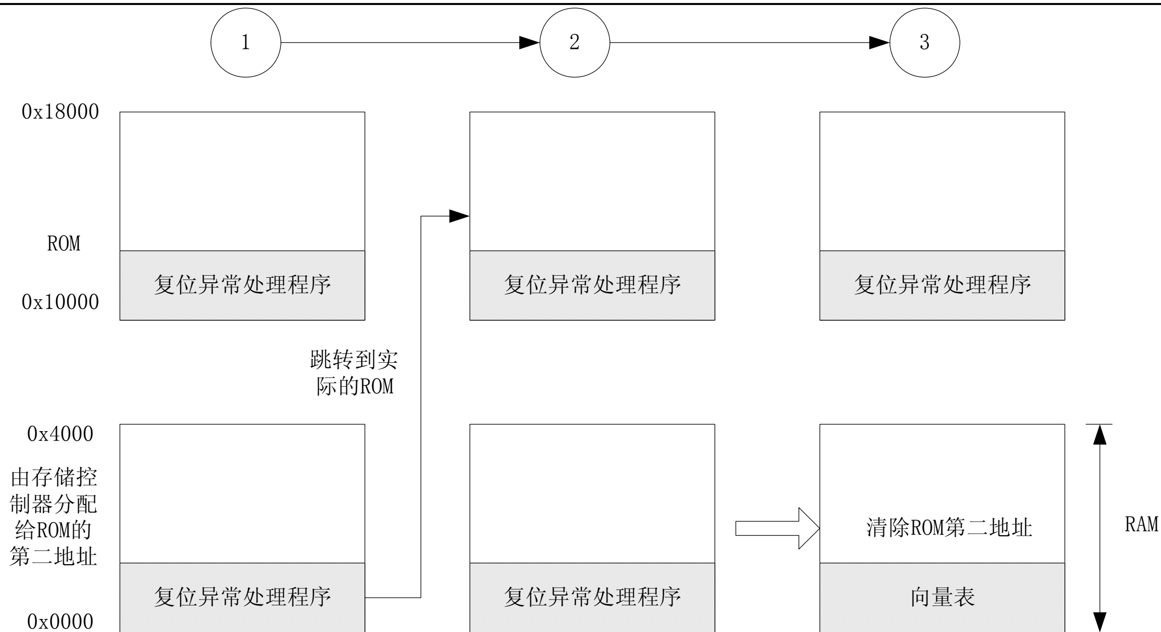


图 4.17 ROM/RAM 重映射（2）

该 ROM 与 RAM 地址重映射的方法可以应用于任何具有 ROM/RAM 重映射机制的平台，但是内存重映射的地址根据具体平台的不同而不同。

图 4.17 显示的地址重映射例子中，第一条指令实现从 ROM 临时地址（0x0 地址）到实际 ROM 的跳转。之后，控制寄存器的重映射位，清除 ROM 的临时地址设置。该代码通常在系统复位后立即执行。重新映射必须在执行 C 库初始化代码前完成。

在具有 MMU 的系统中，可通过在系统启动时配置 MMU 来实现重映射。

下面的例子显示了在 ARM 的 Integrator 开发板上实现的 ROM/RAM 重映射过程。

```
; --- Integrator CM control reg
CM_ctl_reg      EQU   0x1000000C      ; 定义 CM 控制寄存器地址
Remap_bit       EQU   0x04            ;CM 控制寄存器重映射掩码
    ENTRY
;复位异常处理程序开始
; 执行跳转指令，转到实际的 ROM 执行
        LDR     pc, =Instruct_2
Instruct_2
; 设置 CM 控制寄存器的重映射位
        LDR     r1, =CM_ctl_reg
        LDR     r0, [r1]
        ORR     r0, r0, #Remap_bit
        STR     r0, [r1]
; 重映射后，RAM 在 0x0 地址
; 将向量表从 ROM 复制到 RAM （由 __main 函数完成）
```

4.4.4 与局部存储器设置有关的考虑事项

许多 ARM 处理器内核具有片上存储器系统，如 MMU 或 MPU。这些设备通常是在系统启动过程中进行设置并启用的，因此，带有局部存储器系统的内核的初始化序列需要特别地考虑。

在前面所述的代码启动的过程中，__main 中 C 库初始化代码负责建立代码执行时的内存映像，在跳转到__main 前，必须建立处理器内核的运行时存储器视图。这就是说，在复位处理程序中必须设置并启用 MMU 或 MPU。

另外，在跳转到__main 前（通常在 MMU/MPU 设置前），必须启用紧耦合存储器 TCM（Tightly Coupled Memory），因为在通常情况下都是采用分散加载方法将代码和数据装入 TCM。当 TCM 启用后，用户不必存取由 TCM 屏蔽的存储器。

在跳转到__main 前，如果启用了 Cache，可能还会遇到 Cache 一致性的问题：__main 中的函数将程序代码从其加载域拷贝到执行域，在此过程中是将指令作为数据进行处理。这样，一些指令可能被放入数据 Cache 中，在执行这些指令时，由于找不到地址路径而产生错误。为了避免 Cache 一致性的问题，在 C 库初始化序列执行完成后再启用 Cache。

4.4.5 栈指针初始化

在程序的初始化代码中，用户必须要为处理器用到的各种模式设置堆栈。也就是说，复位处理程序必须为应用程序所使用的任何执行模式的栈指针分配初始值。

下面的例子显示了如何在初始化代码中启用不同模式下的堆栈。

```
; 启用系统模式堆栈

    LDR     r2,INT_System_Stack         ; 将系统堆栈的全局变量放到 r2 中
    STR     sp,[r2]                     ; 将系统堆栈指针存储到系统模式下的 sp

    ; 启用系统堆栈限制（为 ARM 编译器的堆栈检测做准备）

    SUB     r1,sp,#SYSTEM_STACK_SIZE        ; 跳转堆栈指针
    BIC     r1,r1,#0x03                 ; 4 字节对齐
    MOV     r10,r1                          ; 将堆栈的限制放入 r10 寄存器（AAPCS 规则）
    LDR     r2,INT_System_Limit         ; 得到堆栈限制全局变量地址
    STR     r1,[r2]                     ; 将堆栈限制存入全局变量

    ; 切换到 IRQ 模式

    MRS     r0,CPSR                     ;得到当前的 CPSR 值
    BIC     r0,r0,#MODE_MASK            ; 清除模式位
    ORR     r1,r0,#IRQ_MODE             ; 设为 IRQ 模式
    MSR     CPSR_cxsf,r1                ; 切换到 IRQ 模式

    ;启用 IRQ 模式堆栈

    LDR     sp,=INT_Irq_SP              ; 将 IRQ 模式堆栈指针放入 sp_irq
   ; 切换到 FIQ

    ORR     r1,r0,#FIQ_MODE             ; 设置 FIQ 模式位
    MSR     CPSR_cxsf,r1                ; 切换到 FIQ 模式

    ; Set-up FIQ stack

    LDR     sp,=INT_Fiq_SP              ; 得到 FIQ 模式指针

    ; 切换到 Abort 模式
```

```
        ORR     r1,r0,#ABT_MODE              ; 设置 Abort 模式位
        MSR     CPSR_cxsf,r1                 ; 切换到 ABT 模式

        ; 启用 Abort 堆栈

        LDR     sp,=INT_Abort_SP

        ; 切换到未定义异常模式

        ORR     r1,r0,#UNDEF_MODE
        MSR     CPSR_cxsf,r1

        ;启用未定义指令模式堆栈

        LDR     sp,=INT_Undefined_SP

       ; Set-up System/User stack
    ……
    ……
```

为了设置栈指针，进入每种模式（中断禁用）并为栈指针分配适合的值。要利用软件栈检查，也必须在此设置栈限制。

复位处理程序中设置的栈指针和栈限制值由 C 库初始化代码作为参数自动传递给__user_initial_stackheap()。因此，不允许__user_initial_stackheap()更改这些值。

下面的例子显示了如何实现__user_initial_stackheap()，该段代码可以和上面的堆栈指针设置程序配合使用。

```
        IMPORT heap_base
        EXPORT __user_initial_stackheap()
    __user_initial_stackheap()
    ; heap base could be hard coded, or placed by description file
        LDR   r0,=heap_base
        ; r1 contains SB value
        MOV   pc,lr
```

4.4.6 硬件初始化

一般情况下，系统初始化代码和主应用程序是分开的。系统初始化要在主应用程序启动前完成。但部分与硬件相关的系统初始化过程，如启用 Cache 和中断，必须在 C 库初始化代码执行完成后才能执行。

为了在进入主应用程序之前，完成系统初始化，可以使用$sub 和$super 函数标识符在进入主程序之前插入一个例程。这一机制可以在不改变源代码的情况下扩展函数的功能。

下面的例子说明了如何使用$sub 和$super 函数标识。链接程序通过调用$sub$$main()函数取代对 main()的调用。所以用户可以在自己编写的$sub$$main()例程中启用 Cache 或使能中断。

```
    extern void $Super$$main(void);
    void $Sub$$main(void)
    {
        cache_enable();    // enables caches
        int_enable();      // enables interrupts
        $Super$$main();    // calls original main()
    }
```

在$Sub$$main(void)函数中，链接程序通过调用$Super$$main()，使代码跳转到实际的 main()函数中。

在完成硬件初始化之后，必须考虑主应用程序运行在何种模式。如果应用程序运行在特权模式（Privileged mode），只需在退出复位处理程序前切换到适当的模式；如果应用程序运行在用户模式下，要在完成系统初始化之后，再切换到用户模式。模式的切换工作，一般在$Sub$$main（void）函数中完成。

联系方式

集团官网：www.hqyj.com 嵌入式学院：www.embedu.org 移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn 物联网学院：www.topsight.cn 研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路 250 号银海大厦 11 层 B 区，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-22193762

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218

华清远见