Technical Note

分散加载文件浅释

ARM 嵌入式开发

TN01010101 V0.00 Date:2008/01/01 工程技术笔记

类别	内容
关键词	分散加载、Scatter
摘　要	本文主要介绍分散加载文件的格式及应用

目　录

1. 适用范围

有时候用户希望将不同代码放在不同存储空间，也就是通过编译器生成的映像文件需要包含多个域，每个域在加载和运行时可以有不同的地址。要生成这样的映像文件，必须通过某种方式告知编译器相关的地址映射关系。

在 Keil/ADS/IAR 等编译工具中，可通过分散加载机制实现。分散加载通过配置文件实现，这样的文件称为分散加载文件。本文重点介绍 Keil 的分散加载文件配置。

2. 基础知识

2.1 基本概念

要了解分散加载文件前首先需要对以下各个概念进行了解，可参考程序清单 2.1。

- Code：为程序代码部分；
- RO-Data：表示程序定义的常量及 const 型数据；
- RW-Data：表示已经初始化的静态变量，变量有初值；
- ZI-Data：表示未初始化的静态变量，变量无初值。

程序清单 2.1 各类型数据声明

```
#define       DATA          (0x10000000)          /* RO-Data      */
char const    GcChar = 5;                         /* RO-Data      */
char          GcStr[] = "string.";                /* RW-Data      */
char          GcZero;                             /* ZI-Data      */
```

Keil 工程编译完成后，查看其的 map 文件，可得到结果如程序清单 2.2 类似。

程序清单 2.2 map 文件信息

```
==============================================================================
    Total RO  Size (Code + RO-Data)                  768 (   0.75kB)
    Total RW  Size (RW-Data + ZI-Data)              2060 (   2.01kB)
    Total ROM Size (Code + RO-Data + RW-Data)        780 (   0.76kB)
==============================================================================
```

由程序清单 2.2 所示的 map 文件可看出：

ROM（Flash）Size = Code＋RO-Data＋RW-Data = 0.76KB；

RAM Size = RW-Data＋ZI-Data = 2.01KB。

为什么上述的 RW-Data 既占用 Flash 又占用 RAM 么，变量不是放在 RAM 中的么，为什么会占用 Flash？因为 RW 数据不能像 ZI 那样“无中生有”的，ZI 段数据只要求其所在的区域全部初始化为零，所以只需要程序根据编译器给出的 ZI 基址及大小来将相应的 RAM 清零。但 RW 段数据却不这样做，所以编译器为了完成所有 RW 段数据赋值，其先将 RW 段的所有初值，先保存到 Flash 中，程序执行时，再 Flash 中的数据搬运到 RAM 中，所以 RW 段即占用 Flash 又占用 RAM，且占用的空间大小是相等的。

这里有必要再了解一下，ZI 和 RW 段数据的赋值在一工程中是在什么地方实现的？首先，变量必先要初始化才能使用，否则初值不正确，而在 main()函数后变量已经可以正常使用，那就是说变量的初始化是在之前完成的，查看这之前的代码只有__main()一个函数，除了赋初值外，都还做了什么呢？

__main()这个函数主要由以下两部分功能组成，如下所示。

- _main()：完成代码和数据的拷贝，并把 ZI 数据区清零。代码拷贝可将代码拷贝到另外一个映射空间并执行，如将代码拷贝到 RAM 执行；数据拷贝完成 RW 段数据赋值；数据区清零完成 ZI 段数据赋值。以上的代码和分散加载文件密切相关。
- _rt_entry()：进行 STACK 和 HEAP 等的初始化。最后_rt_entry 跳进 main()函数入口。当 main()函数执行完后，_rt_entry 又将控制权交还给调试器。

3. 分散加载文件概述

分散加载（scatter）文件是一个文本文件，它可以用来描述连接器生成映像文件时需要的信息。通过编写一个分散加载文件来指定 ARM 连接器在生成映像文件时如何分配 Code、RO-Data，RW-Data，ZI-Data 等数据的存放地址。如果不用分散加载文件指定，那么 ARM 连接器会按照默认的方式来生成映像文件。一般情况下我们不需要使用分散加载文件。但对一些特殊的情况例如需要将不同的程序代码存储到不同的地址区域时需要修改分散加载文件。

1. 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：

2. 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些数据放在哪个内存空间中。

3. 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 中以缩短中断等待时间，而将不经常使用的配置信息放在较慢的闪存中。

4. 内存映射的 I/O

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。

5. 位于固定位置的函数

可以将函数放在内存中的固定位置，即使已修改并重新编译周围的应用程序。

6. 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。

4. 分散加载文件语法

分散加载文件主要由一个加载时域和多个运行时域组成，其大致结构如图 4.1 所示。

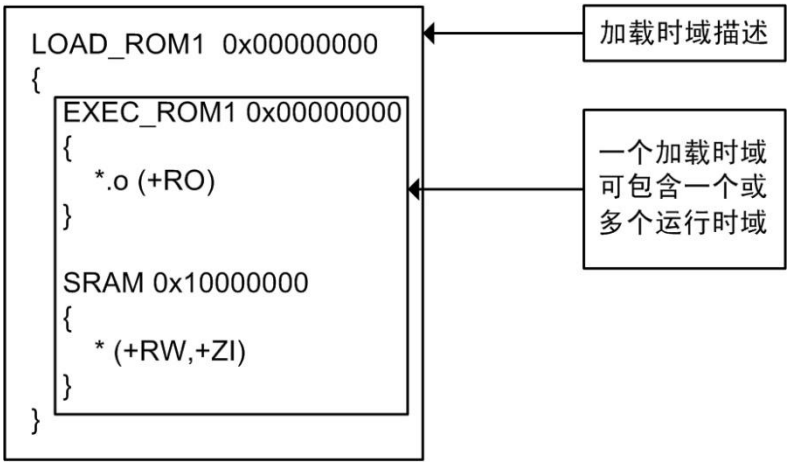

图 4.1　分散加载文件语法结构

下面分别介绍一下加载时域和运行时域特点及用法。

4.1　加载时域的描述

加载时域语法格式如程序清单 4.1 所示，其每项含义解释如下。

程序清单 4.1　加载时域描述语法描述

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

- load_region_name：为本加载时域的名称，名称可以按照用户意愿自己定义，该名称中只有前 31 个字符有意义；
- base_designator：用来表示本加载时域的起始地址，可以有下面两种格式中的一种：
  - ◆ base_address：表示本加载时域中的对象在连接时的起始地址，地址必须是字对齐的；
  - ◆ +offset：表示本加载时域中的对象在连接时的起始地址是在前一个加载时域的结束地址后偏移量 offset 字节处。本加载时域是第一个加载时域，则它的起始地址即为 offset，offset 的值必须能被 4 整除。
- attribute_list：指定本加载时域内容的属性，包含以下几种，默认加载时域的属性是 ABSOLUTE。
  - ◆ ABSOLUTE：绝对地址；
  - ◆ PI：与位置无关；
  - ◆ RELOC：可重定位；
  - ◆ OVERLAY：覆盖；

- ◆ NOCOMPRESS：不能进行压缩。

- max_size：指定本加载时域的最大尺寸。如果本加载时域的实际尺寸超过了该值，连接器将报告错误，默认取值为 0xFFFFFFFF；
- execution_region_description：表示运行时域，后面有个+号，表示其可以有一个或者多个运行时域，关于运行时域的介绍请看后面。

注：程序清单 4.1 中'|'、'[]' 符号的使用请参考 BNF 语法。

4.2 运行时域的描述

加载时域语法格式如程序清单 4.2 所示，其每项含义解释如下。

程序清单 4.2 运行时域语法描述

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

- exec_region_name：为为本加载时域的名称，名称可以按照用户意愿自己定义，该名称中只有前 31 个字符有意义；
- base_address：用来表示本加载时域的起始地址，可以有下面两种格式中的一种：
  - ◆ base_address：表示本加载时域中的对象在连接时的起始地址，地址必须是字对齐的；
  - ◆ +offset：表示本加载时域中的对象在连接时的起始地址是在前一个加载时域的结束地址后偏移量 offset 字节处，offset 的值必须能被 4 整除。
- attribute_list：指定本加载时域内容的属性：
  - ◆ ABSOLUTE：绝对地址；
  - ◆ PI：与位置无关；
  - ◆ RELOC：可重定位；
  - ◆ OVERLAY：覆盖；
  - ◆ FIXED：固定地址。区加载地址和执行地址都是由基址指示符指定的，基址指示符必须是绝对基址，或者偏移为 0。
  - ◆ ALIGNalignment：将执行区的对齐约束从 4 增加到 alignment。alignment 必须为 2 的正数幂。如果执行区具有 base_address，则它必须为 alignment 对齐。如果执行区具有 offset，则链接器将计算的区基址与 alignment 边界对齐；
  - ◆ EMPTY：在执行区中保留一个给定长度的空白内存块，通常供堆或堆栈使用。
  - ◆ ZEROPAD：零初始化的段作为零填充块写入 ELF 文件，因此，运行时无需使用零进行填充；
  - ◆ PADVALUE：定义任何填充的值。如果指定 PADVALUE，则必须为其赋值；
  - ◆ NOCOMPRESS：不能进行压缩；
  - ◆ UNINIT：未初始化的数据。
- max_size：指定本加载时域的最大尺寸。如果本加载时域的实际尺寸超过了该值，连接器将报告错误，默认取值为 0xFFFFFFFF；

- length：如果指定的长度为负值，则将 base_address 作为区结束地址。它通常与 EMPTY 一起使用，以表示在内存中变小的堆栈。
- input_section_description：指定输入段的内容。

4.3 输入段描述

输入段语法描述如程序清单 4.3 所示。

程序清单 4.3 输入段语法描述

```
module_select_pattern [ "(" input_section_selector ( "," input_section_selector )* ")" ]
        ("+" input_section_attr | input_section_pattern | input_symbol_pattern)
```

- module_select_pattern：目标文件滤波器，支持使用通配符“*”与“?”。其中符号“*”代表零个或多个字符，符号“？”代表单个字符。进行匹配时所有字符不区分大小写。当 module_select_pattern 与以下内容之一相匹配时，输入段将与模块选择器模式相匹配：
  - 包含段和目标文件的名称；
  - 库成员名称（不带前导路径名）；
  - 库的完整名称（包括路径名）。如果名称包含空格，则可以使用通配符简化搜索。例如，使用*libname.lib 匹配 C:\lib dir\libname.lib。
- nput_section_attr：属性选择器与输入段属性相匹配。每个 input_section_attr 的前面有一个“+”号。如果指定一个模式以匹配输入段名称，名称前面必须有一个“+”号。可以省略紧靠“+”号前面的任何逗号。选择器不区分大小写。可以识别以下选择器：
  - 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：即包含 ENTRY 点的段。

可以识别以下同义词：

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

可以识别以下伪属性：

- FIRST；
- LAST。

通过使用特殊模块选择器模式.ANY 可以将输入段分配给执行区，而无需考虑其父模块。

可以使用一个或多个.ANY 模式以任意分配方式填充运行时域。在大多数情况下，使用单个.ANY 等效于使用*模块选择器。

5. 分散加载应用实例

5.1 一个普通的分散加载配置

假设，一个 Cortex-M3 内核的 LPC17xx 微控制器有 Flash、RAM 的资源如下：

- Flash 基址：0x00000000，大小：256 KByte；
- RAM 基址：0x10000000，大小：32 Kbyte。

那这一个分散加载文件应该怎样描述呢？可参考如程序清单 5.1 所示的配置。

程序清单 5.1 一个普通的分散加载文件配置

```
LR_IROM1 0x00000000 0x00040000 {                ; 定义一个加载时域，域基址：0x00000000，域大
                                                ; 小为 0x00040000，对应实际 Flash 的大小
  ER_IROM1 0x00000000 0x00040000 {              ; 定义一个运行时域，第一个运行时域必须和加载
                                                ; 时域起始地址相同，否则库不能加载到该时域的
                                                ; 错误，其域大小一般也和加载时域大小相同
    *.o (RESET, +First)                         ; 将 RESET 段最先加载到本域的起始地址外，即
                                                ; RESET 的起始地址为 0，RESET 存储的是向量表
    .ANY (+RO)                                  ; 加载所有匹配目标文件的只读属性数据，包含：
                                                ; Code、RW-Code、RO-Data。
  }

  RW_IRAM1 0x10000000 0x00008000 {              ; 定义一个运行时域，域基址：0x10000000，域大
                                                ; 小为 0x00008000，对应实际 RAM 大小
    * (+RW +ZI)                                 ; 加载所有区配目标文件的 RW-Data、ZI-Data
                                                ; 这里也可以用.ANY 替代*号
  }
}
```

5.2 多块 RAM 的分散加载文件配置

还是上述的 MCU，假设其增加了另外一块 RAM，其资源如下：

- Flash 基址：0x00000000，大小：256 KByte；
- RAM1 基址：0x10000000，大小：32 Kbyte；
- RAM2 基址：0x2007C000，大小：32 Kbyte。

如果我想将这两块不连续的 RAM 都使用起来（可使用 64KB RAM）？分散加载文件应怎样描述呢？可参考如程序清单 5.2 所示配置。

程序清单 5.2 双 RAM 的加载文件配置

```
LR_IROM1 0x00000000 0x00040000 {
  ER_IROM1 0x00000000 0x00040000 {
    *.o (RESET, +First)
    .ANY (+RO)
  }
```

```
  RW_IRAM1 0x10000000 0x00008000 {          ; 定义 RAM1 的运行时域
    .ANY (+RW +ZI)                          ; 使用.ANY 进行随意分配变量，这里不能使用*号
                                            ; 替代，*表示匹配有所有的目标文件，这样变量就
                                            ; 无法分配到第二块 RAM 空间了
  }

  RW_IRAM2 0x2007C000 0x00008000 {          ; 定义第 RAM2 的运行时域
    .ANY (+RW +ZI)                          ; 同样使用.ANY 随意分配变量的方式
  }

                                            ; 如果还有另多的 RAM 块，在这里增加新的运行
                                            ; 时域即可，格式和 RAM2 的定义相同
}
```

如程序清单 5.2 所示的分散加载文件配置，确实可将两块 RAM 都使用起来，即有 64KB 的 RAM 可以使用，但其并不能完全等价于一个 64KB 的 RAM，实际应用可能会碰到如下的问题。

如我在 main.c 文件中声明了 1 个 40KB 的数组，如程序清单 5.3 所示，程序中红色部分已注释，可暂不理会。

程序清单 5.3　定义大数组出错

```
// main.c 文件
…
unsigned char     GucTest0[40 * 1024];              // 定义一个 40KB 的数组
//unsigned char    GucTest1[20 * 1024];              // 定义第一个 20KB 数组
//unsigned char    GucTest2[20 * 1024];              // 定义第二个 20KB 数组
…
// end of file
```

如程序清单 5.3 所示的程序在编译时会出现错误，并提示没有足够的空间，为什么呢？原因为数组是一个整体，其内部元素的地址是连续的，不能分割的，但是在两个不连续的 32KB 空间中，是没办法分配出一个连续的 40KB 地址空间，所以编译会提示空间不足，分配 40KB 数组失败。

还是程序清单 5.3 所示的程序，去掉 40KB 的数据，换成 2 个 20KB 的数组（即红色注释代码部分），编译结果又会如何呢？

编译结果还是提示空间不足，这又是为什么呢？这里出错的原因其实和上面的原因是相同的，首先，重温一个，.ANY 的作用，.ANY 是一个通配符，当其与以下内容之一相匹配时将进行选择。

- 包含段和目标文件的名称；
- 库成员名称（不带前导路径名）；
- 库的完整名称（包括路径名）。如果名称包含空格，则可以使用通配符简化搜索。例如，使用*libname.lib 匹配 C:\lib dir\libname.lib。

后面两个和本次讨论的话题无关，再仔细的看第一个匹配项为：包含段和目标文件，关于段的解释请参考“5.5 段在分散加载文件中的应用”小节。段这里先不用理会，因为这里没有用到段，所以只剩下目标文件。要注意是“目标文件”，不是其它，即是说一个 C 文件

编译后，其所有的变量、代码都会作为一个整体。所以定义两个 20KB 和定义了一个 40KB，在编译器看来都是一样，就是这个 C 文件总共定义了 40KB 的空间，我要用 40KB 的空间来分配它，因此会出现同样的错误。

关于大数组分配的解决方法，有两种，分别如下：

- 将数组分开在不同的 C 文件中定义，避免在同一个 C 文件定义的数据大小总量超过其中最大的分区；
- 将一个 C 数组，使用段定义，使其从该 C 文件中独立出来，这样编译器就不会将它们作为一个整体来划分空间了，其示例如程序清单 5.4 所示，解释请参考“5.5 段在分散加载文件中的应用”小节。

程序清单 5.4　使用段的方式分配多数组

```
#pragma arm section zidata = "SRAM"          // 在 C 文件中定义新的段
unsigned char     GucTest1[20 * 1024];       // 定义第一个 20KB 数组
#pragma arm section                          // 恢复原有的段

unsigned char     GucTest2[20 * 1024];       // 定义第二个 20KB 数组，这 20KB 数组不会和
                                             // GucTest1 作为一个整体来划分空间
```

5.3　多块 Flash 的分散加载文件配置

再一下上述的 MCU，假其增加多了一块 Flash，不是 RAM，其资源如下。

- Flash1 基址：0x00000000，大小：256 KByte；
- Flash2 基址：0x20000000，大小：2048 KByte；
- RAM 基址：0x10000000，大小：32 Kbyte；

注意这里多增加的一块的不是 RAM，而是 Flash，其情况会如何呢？假设其相同，那写法应该就是如程序清单 5.5 所示的样子。

程序清单 5.5　双 Flash 的错误配置示例 1

```
LR_IROM1 0x00000000 0x00040000 {
  ER_IROM1 0x00000000 0x00040000 {          ; 定义 Flash1 运行时域
    *.o (RESET, +First)                     ; 先加载向量表
    .ANY (+RO)                              ; 随意分配只读数据
  }

  ER_IROM2 0x20000000 0x00200000 {          ; 定义 Flash2 运行时域
    .ANY (+RO)                              ; 随意分配只读数据
  }

  RW_IRAM1 0x10000000 0x00008000 {
    .ANY (+RW +ZI)
  }
}
```

如程序清单 5.5 的分散加载配置，进行编译却出错了，错误提示如程序清单 5.6 所示。

程序清单 5.6　双 Flash 编译出错句子

```
..Error: L6202E: __main.o(!!!main) cannot be assigned to non-root region 'ER_IROM2'
..Error: L6202E: __scatter.o(!!!scatter) cannot be assigned to non-root region 'ER_IROM2'
..Error: L6202E: __scatter_copy.o(!!handler_copy) cannot be assigned to non-root region 'ER_IROM2'
..Error: L6202E: __scatter_zi.o(!!handler_zi) cannot be assigned to non-root region 'ER_IROM2'
..Error: L6202E: anon$$obj.o(Region$$Table) cannot be assigned to non-root region 'ER_IROM2'
..Error: L6203E: Entry point (0x20000001) lies within non-root region ER_IROM2.
```

该错误的意思是说，__main.o、__scatter.o、__scatter_copy.o 等不能被加载到第二块 Flash 的运行时域 ER_IROM2，也就是说这几项目数据只能加载到 ER_IROM1 的运行时域。

为什么这些数据不能放到第二个运行时域呢？后面再进行解释。

无论如何，但总的来说是使用.ANY 引起的错误，.ANY 是让编译器随意分配数据，所以数据有可能被分配到 ER_IROM2。如果手动这出错的几项到 ER_IROM1，结果又会如何呢？请看如程序清单 5.7 所示的配置。

程序清单 5.7　双 Flash 的错误配置示例 2

```
LR_IROM1 0x00000000 0x00040000 {
  ER_IROM1 0x00000000 0x00040000 {
    *.o (RESET, +First)                    ; 先加载向量表
    __main.o                               ; 手动加载到 ER_IROM1，避免自动分配引起错误
    __scatter.o                            ; 避免自动分配
    __scatter_copy.o                       ; 避免自动分配
    __scatter_zi.o                         ; 避免自动分配
    * (Region$$Table)                      ; 避免自动分配
    .ANY (+RO)
  }

  ER_IROM2 0x20000000 0x00200000 {
    .ANY (+RO)
  }

  RW_IRAM1 0x10000000 0x00008000 {
    .ANY (+RW +ZI)
  }
}
```

终于，编译没有错误了，但在软件仿真下，现象明显不正确，main()函数都跑不到，这？

这个问题得从第一个时域与加载时域的关系来进行说明，其关系如下所示。

1.　第一个运行时域存放的代码不会进行额外拷贝

因为分散加载文件有一项很强大的功能，就是可以将 Flash 的代码拷贝到 RAM 中运行，这一段拷贝代码就存在于__main()函数中，但拷贝代码不能拷贝自身，所以必须规定有一个运行时域中存放的代码是不会被拷贝的，这个指的就是第一个运行时域。

拷贝代码为什么不能拷贝自身呢？举个例子，假设 A 代码是要被拷贝到 RAM 执行的代码，那 A 代码必先存储于 Flash 中，然后被拷贝到 RAM 中。这样 A 代码不就存在两段了么，但是只能有一段是有效的，就像定义了两个相同名字的函数，最终只能留下一个。所以最终被认定为拷贝后的代码才是有效的。那就是说 A 代码从 Flash 中拷贝到了 RAM，RAM 中代

码才是有效，Flash 中的代码是无效的，其它程序调用 A 代码也是调用 RAM 中的代码而不是 Flash 的。那如果 A 是一段拷贝代码，那就会发生如下的现象，一段程序调用 RAM 中的 A 代码，RAM 中的 A 代码再将自己从 Flash 中拷贝到 RAM。

结论，一段代码必须先完成拷贝，才能被执行。换句理解就是拷贝代码前包括自身的所有代码都不能拷贝，也就是说这些代码全部都必须放在第一个运行时域中。

2. 规定其余运行时域中存放的代码均会被拷贝

一个加载时域，只需要一个不拷贝的运行时域即可。所以规定其余所有的运行时域中的代码均会被拷贝。

3. 第一个运行时域的基址必须与加载域基址相同

为了保证第一个运行时域的代码能够被正确存储和执行，因此要求第一个运行时域的基址必须和加载时域的基址相同。

看完了第一个运行时域与加载时域的关系，那程序清单 5.5、程序清单 5.7 出现错误的原因就很明了了。

- 程序清单 5.5 出错原因，是因为__main()等拷贝代码必须存放在第一个运行时域中；
- 程序清单 5.7 出错的原因如程序清单 5.8 所示。

程序清单 5.8　程序清单 5.7 出错的原因

```
  ER_IROM2 0x20000000 0x00200000 {          ; 定义 Flash1 运行时域
    .ANY (+RO)                              ; 随意分配只读数据，但代码存放在第二个运行时
                                            ; 域中，所以该代码是运行时才被拷贝到这里，
                                            ; 那就是说要往 Flash1 直接写数据，当然会导致
                                            ; 程序出错了
  }
```

程序清单 5.9 给出了双 Flash 的分散加载文件正确配置示例，如下所示。

程序清单 5.9　双 Flash 的正确配置示例

```
LR_IROM1 0x00000000 0x00040000 {          ; 定义 Flash1 的加载域
  ER_IROM1 0x00000000 0x00040000 {
    *.o (RESET, +First)
    .ANY (+RO)                            ; 随机分配只读数据
  }

  RW_IRAM1 0x10000000 0x00008000 {
    .ANY (+RW +ZI)
  }
}

LR_IROM2 0x20000000 0x00200000 {          ; 定义 Flash2 的加载域
  ER_IROM2 0x20000000 0x00200000 {
    .ANY (+RO)                            ; 随机分配只读数据，代码不会进行拷贝
  }
}
```

5.4 Flash 特殊要求应用

以上述的 MCU 为例，我的片内 Flash 前面 0x200 个字节要存放向量表，不能包含其它任何代码，应该怎么操作呢？

当然，按照程序清单 5.9 的方式也可以，不过该方式在一个比较大的缺点。使用这种方式生成的 Bin 文件有两个，如果用来下载不太方便，那有没有只生成一个 Bin 文件的方式呢？答案当然是有的，如程序清单 5.10 所示。但这种方式也有一个缺点，就是其是以填充的方式产生的成一个 Bin 文件，当两个运行时域地址相距很大时，就会导致填充出来的 Bin 文件非常大，所以不适合于程序清单 5.9 所示的双 Flash 应用。

程序清单 5.10　Flash 特殊要求应用示例

```
LR_IROM1 0x00000000 0x00040000 {
    ER_IROM1 0x00000000 0x200 {
        *.o (RESET, +First)
    }

    ER_IROM2 0x200 FIXED 0x3FD00 {          ; FIXED 修饰，让其和第一个运行域关联起来，
                                            ; 合成一个运行域
        .ANY (+RO)
    }

    RW_IRAM1 0x10000000 0x00008000 {
        .ANY (+RW +ZI)
    }
}
```

5.5 段在分散加载文件中的应用

段在分散加载文件中，应用也非常重要的，首先了解一下在汇编怎么产生段？可参考如程序清单 5.11 所示的操作方法。

程序清单 5.11　在汇编中定义段的方法

```
AREA sectionname{,attr}{,attr} ... END
```

- Sectionname：指的是段的名字，段的名字可以任意命名，但如果该段名非以字母开关，则必须在段名前后加上|标号，如：|1_DataArea|。另外有一些段名，是之前延续下来的，如|.text|，使用该段名，意为将该代码与 C 库关联起来；
- attr：可以连接一个或多个属性字，常见的有如下 4 种，还有一些未进行列举。
  - ◆ ALIGN = n：n 取值 0~31，段以 2n 字节对齐，如：n = 3，表示该段以 8 字节对齐；
  - ◆ ASSOC = section：
  - ◆ CODE：标志该段为代码，默认为 READONLY；
  - ◆ NOINIT：标志该段数据，不进行初始化。

在 C 语言中指定段的操作方法如程序清单 5.12 所示。

程序清单 5.12　在 C 中定义段的方法

```
#pragma arm section [sort_type][[=]"name"]] [,sort_type="name"]*
…
#pragma arm section                                    //  恢复原有段名
```

- sort_type：为 code、rwdata、rodata、zidata 其之一。
- Name：任意段名，如果“sort_type”指定了但没有指定“name”，那么之前的修改的段名将被恢复成默认值。

C 语言中指定段的示例如程序清单 5.13 所示。

程序清单 5.13　C 中指定段示例

```
#pragma arm section rwdata = "SRAM",zidata = "SRAM"
      static OS_STK    SecondTaskStk[256];          //  “rwdata”“zidata”将定位在“sram”段中。
#pragma arm section                                  // 恢复默认设置
```

这样声明之后，分散加载文件就可以直接对段进行操作了，如程序清单 5.14 所示。

程序清单 5.14　段使用示例

```
RW_IRAM1    0x10000000    0x00008000
{
    .ANY(sram)                                       ;// 将 SRAM 段加载到 RW_IRAM1 域
}
```

可以这样理解段，一个 C 文件或 ASM 文件是一个完整的目标文件段，使用段去修饰后，会直接在该文件下产生新的段，新的段和目标文件段是不重叠的、独立。这也是程序清单 5.4 编译能通过的原因。

5.6　程序拷贝到 RAM 中执行应用

程序清单 5.15 所示的分散加载文件，是将 main.c 代码全部拷贝到 RAM 中运行的示例，其余代码还是存放在 Flash 中。

程序清单 5.15　main.c 代码拷贝到 RAM 中运行示例

```
LR_IROM1 0x00000000 0x00040000 {
  ER_IROM1 0x00000000 0x00040000 {
    *.o (RESET, +First)
    .ANY (+RO)                                       ; 加载所有匹配的只读代码，不包含 main.c
  }

  RW_IRAM1 0x10000000 0x00008000 {                   ; 定义 RAM 的运行时域
    main.o (+RO)                                     ; 加载 main.c 文件到 RAM 中执行
    .ANY (+RW +ZI)                                   ; 加载所有匹配的变量
  }
}
```

程序清单 5.16 所示的分散加载文件，实现了将所有代码拷贝到 RAM 执行，这里所说的全部代码不包括 main()函数前要执行的代码，原因这里不再重述。

程序清单 5.16　将全部代码拷贝到 RAM 中运行示例

```
LR_IROM1 0x00000000 0x00040000 {
    ER_IROM1 0x00000000 0x00040000 {
        *.o (RESET, +First)                    ; 向量表
        startup_LPC17xx.o                      ; 包含复位入口
        __main.o                               ; 拷贝代码 1
        __scatter.o                            ; 拷贝代码 2
        __scatter_copy.o                       ; 拷贝代码 3
        __scatter_zi.o                         ; 拷贝代码 4
        * (Region$$Table)                      ; 拷贝代码 5
    }

    RW_IRAM1 0x10000000 0x00008000 {
        .ANY (+RO)                          ; 加载剩下的所有代码
        .ANY (+RW +ZI)
    }
}
```

6. 免责声明

广州周立功单片机科技有限公司随附提供的软件或文档资料旨在提供给您(本公司的客户)使用，仅限于且只能在本公司制造或销售的产品上使用。

该软件或文档资料为本公司和/或其供应商所有，并受适用的版权法保护，版权所有。如有违反，将面临相关适用法律的刑事制裁，并承担违背此许可的条款和条件的民事责任。

本公司保留在不通知读者的情况下，有修改文档或软件相关内容的权利，对于使用中所出现的任何效果，本公司不承担任何责任。

该软件或文档资料“按现状”提供，不提供保证，无论是明示的、暗示的还是法定的保证。这些保证包括(但不限于)对出于某一特定目的应用此软件的适销性和适用性默示的保证。在任何情况下，公司不会对任何原因造成的特别的、偶然的或间接的损害负责。