armlink 第四章 scatter 文件舉例

原創 安仔都有人用 2020-04-14 16:37

armlink 第四章 scatter 文件舉例

在前面學習了基本術語和概念之後，本章是加強 scatter 編寫能力的章節。

4.1 什麼時候使用 scatter 文件

scatter 文件通常用於嵌入式系統中，因為這些系統含有 ROM，RAM，還有一些內存映射的外設。下面的場景常使用 scatter 文件：

1. 複雜的內存映射：放在不同內存區域的 section，需要使用 scatter 文件來更精細的操控放置的位置
2. 不同的存儲類型：許多系統包含各種各樣的存儲設備，如 flash,ROM,SDRAM,SRAM 等。這時可以使用 scatter 文件，將更適合的存儲區域放置更適合的代碼。例如：中斷代碼放置在 SRAM 中，已達到快速響應的目的；而不頻繁訪問的配置信息可以放置在 flash 存儲中。
3. 內存映射的外設：在內存映射機制下，scatter 文件可以在一個精確的地址放置數據 section。這樣訪問這個數據 section 就相當於訪問對應的外設。
4. 在固定地址存放函數：即使修改並重新編譯了應用程序，而跟應用程序緊密相關的函數還是可以放置在一個固定的位置。這個對於跳轉表的實現非常有用。
5. 使用符號標記堆和棧：當應用被鏈接時，可以為堆和棧定義符號

4.2 在 scatter 文件中指定堆和棧

在 c 語言中，常常需要兩個存儲區域，堆和棧。在所有的內存都由我們分配的情況下，堆和棧也需要我們進行分配。

在程序開始運行之前，會調用__user_setup_stackheap()函數，它負責初始化堆和棧。而這個函數根據我們在 scatter 文件中的設置來初始化。

要想正確的初始化堆和棧。我們需要在 scatter 文件中定義兩個特殊的執行 region。分別叫做 ARM_LIB_HEAP 和 ARM_LIB_STACK。這兩段內存由 c 庫進行初始化，所以不能放置任何

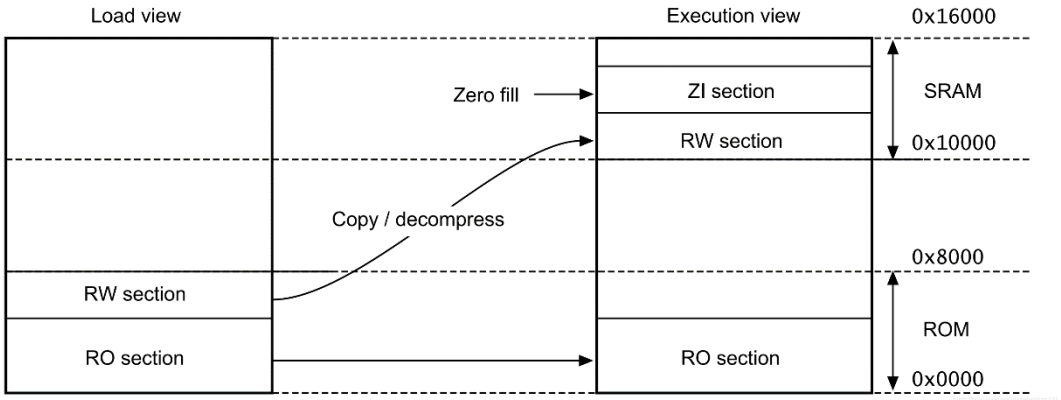
輸入 section，此時應該設置 EMPTY 屬性。同時也可以給這兩個內存區域設置基址和大小。如下：

```
LOAD_FLASH …
{
    …
    ARM_LIB_STACK 0x40000 EMPTY -0x20000 ; 棧區，向下增長
    { }
    ARM_LIB_HEAP 0x28000000 EMPTY 0x80000 ; 堆區向上增長
    { }
    …
}
```

當然還有更簡單的用法，只需要定義一個特殊的執行 region，叫做 ARM_LIB_STACKHEAP，同樣他需要有 EMPTY 屬性，並設置基址和大小

4.3 使用 scatter 文件描述一個簡單的鏡像

如下圖，是一個簡單的鏡像內存視圖。



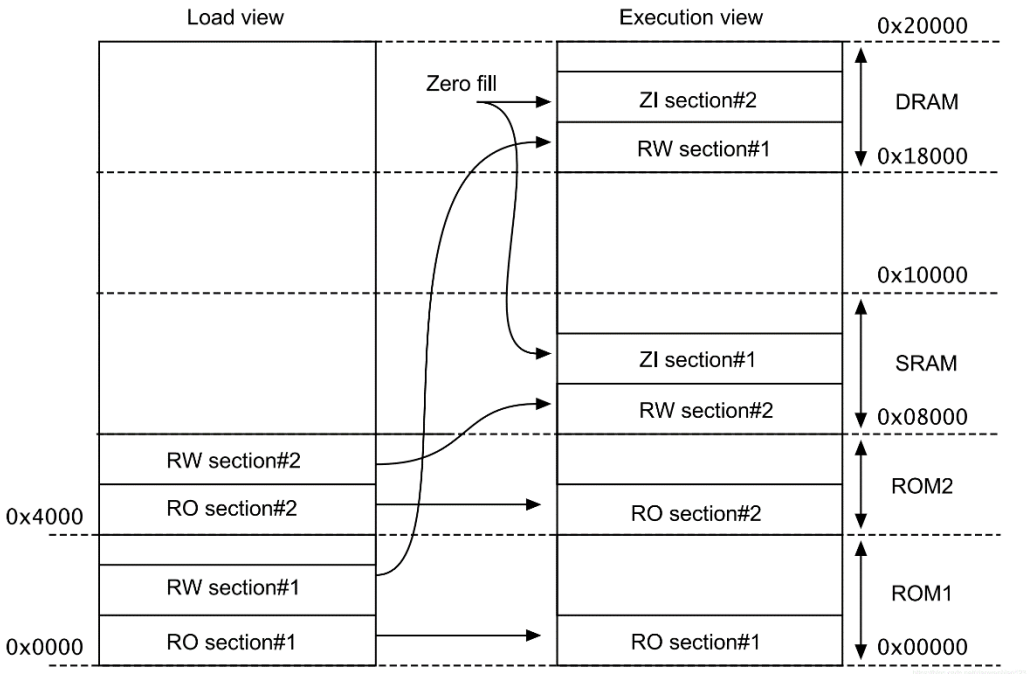
下面根據這個圖，來寫一個 scatter 文件

```
LOAD_ROM 0x0000 0x8000 ; 加載 region 的名字叫 LOAD_ROM
; 基址 0x0000
; 最大大小 0x8000
{
    EXEC_ROM 0x0000 0x8000 ; 第一執行 region 的名字叫做 EXEC_ROM
    ; 基址 0x000
    ; 最大大小 0x8000
```

```
    {
        * (+RO) ; 放置所有的代碼和 RO 數據
    }
    SRAM 0x10000 0x6000 ; 第二個執行 region 叫 SRAM
    ; 基址 0x10000
    ; 最大大小 0x6000
    {
        * (+RW, +ZI) ; 放置所有的 RW 數據和 ZI 數據
    }
}
```

4.4 使用 scatter 文件描述一個稍微複雜的鏡像

如下圖



下面的例子展示了上圖對應的 scatter 描述

```
LOAD_ROM_1 0x0000 ; 第一個加載 region 的基址為 0x0000
{
    EXEC_ROM_1 0x0000 ; 第一個執行 region 的基址為 0x0000
    {
        program1.o (+RO) ; 放置 programe1.o 中的所有的代碼和 RO 數據
```

```
    }
    DRAM 0x18000 0x8000 ; 這個執行 region 的基址為 0x18000,最大大小為 0x8000
    {
        program1.o (+RW, +ZI) ; 放置 program1.o 中的所有的 RW 數據和 ZI 數據
    }
}
LOAD_ROM_2 0x4000 ; 第二個加載 region 的基址為 0x4000
{
    EXEC_ROM_2 0x4000
    {
        program2.o (+RO) ; 放置 programe2.o 中的所有的代碼和 RO 數據
    }
    SRAM 0x8000 0x8000
    {
        program2.o (+RW, +ZI) ; 放置 program2.o 中所有的 RW 數據，和 ZI 數據
    }
}
```

注意：在上面這個例子中，如果再次新增一個 program3.o 文件。我們需要將 program3.o 也放置進去。當然，你也可以使用通配符*，或者.ANY 來匹配剩下的所有文件。

4.5 在指定地址放置函數和數據

為了單獨放置函數和數據，需要將這些函數和數據與源文件中的剩下的部分分開來對待。

鏈接器有兩種方法使我們能夠在指定位置放置 section：

1. 在 scatter 文件中定義一個指定位置的執行 region，然後在這個 region 中放置需要的 section。
2. 定義"__at" section .這些特殊 section 能夠根據名字而獲得放置地址。

為了將函數或者數據放置在一個特殊位置，他們必須放置在某個 section 中。下面幾種方式可以達到此目的：

1. 放置函數和數據在他們自己單獨的源文件中
2. 使用__attribute__((at(address)))將變量放置在指定位置的 section 中
3. 使用__attribute__((section("name")))將函數或者變量放置在指定名字的 section 中。
4. 在彙編代碼中，使用 AREA 僞指令。因為 AREA 僞指令是彙編當中最小的可定位的單元

5. 使用–split_sectoins 編譯選項，為每個源文件中的函數生成一個 section

下面舉例說明。

4.5.1 不使用 scatter，在指定的地址放置變量

1. 創建 main.c 源文件包含下面的代碼

```
#include <stdio.h>
extern int sqr(int n1);
int gSquared __attribute__((at(0x5000))); //放在 0x5000
int main()
{
    gSquared=sqr(3);
    printf("Value squared is: %d\n", gSquared);
}
```

2. 創建 function.c 源文件包含下面的代碼

```
int sqr(int n1)
{
    return n1*n1;
}
```

3. 編譯並連接源文件：

```
armcc -c -g function.c
armcc -c -g main.c
armlink --map function.o main.o -o squared.axf
```

--map 表示顯示內存映射。

在上面例子中，__attribute__((at(0x5000)))指示全局變量 gSquared 放置於絕對地址 0x5000 處。

內存映射如下：

```
…
Load Region LR$$.ARM.__at_0x00005000 (Base: 0x00005000, Size: 0x00000000,
Max: 0x00000004,ABSOLUTE)
    Execution Region ER$$.ARM.__at_0x00005000 (Base: 0x00005000, Size:
0x00000004, Max:0x00000004, ABSOLUTE, UNINIT)
    Base Addr  Size       Type Attr Idx E Section Name       Object
    0x00005000 0x00000004 Zero RW   13  .ARM.__at_0x00005000 main.o
```

4.5.2 使用 scatter，在一個命名的 section 中放置變量

1. 創建 main.c 包含如下源文件

```
#include <stdio.h>
extern int sqr(int n1);
int gSquared __attribute__((section("foo"))); //放置在名字叫 foo 的 section 中
int main()
{
    gSquared=sqr(3);
    printf("Value squared is: %d\n", gSquared);
}
```

2. 創建 functio.c 包含下面代碼

```
int sqr(int n1)
{
    return n1*n1;
}
```

3. 創建 scatter 文件 scatter.scat 包含如下配置

```
LR1 0x0000 0x20000
{
    ER1 0x0 0x2000
    {
        *(+RO) ; 餘下的代碼和只讀數據放置在此處
    }
    ER2 0x8000 0x2000
    {
        main.o
    }
    ER3 0x10000 0x2000
    {
        function.o
        *(foo) ; 將 gSquared 放置在此處
    }
    ; RW 和 ZI 數據放置在 0x200000 處
    RAM 0x200000 (0x1FF00-0x2000)
    {
        *(+RW, +ZI)
    }
```

```
    ARM_LIB_STACK 0x800000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP +0 EMPTY 0x10000
    {
    }
}
```

ARM_LIB_STACK 和 ARM_LIB_HEAP region 是必須的，因為程序和 c 庫進行鏈接

4. 編譯並鏈接源文件

```
armcc -c -g function.c
armcc -c -g main.c
armlink --map --scatter=scatter.scat function.o main.o -o squared.axf
```

上例，__attribute__((section("foo"))) 指示了 gSquared 被放置在名字叫 foo 的 section 中。scatter 文件也說明了將 foo section 放置在 ER3 執行 region 中。

內存映射如下：

```
Load Region LR1 (Base: 0x00000000, Size: 0x00001570, Max: 0x00020000, ABSOLUTE)
…
    Execution Region ER3 (Base: 0x00010000, Size: 0x00000010, Max: 0x00002000, ABSOLUTE)
    Base Addr  Size       Type Attr Idx E Section Name Object
    0x00010000 0x0000000c Code RO   3   .text     function.o
    0x0001000c 0x00000004 Data RW   15  foo       main.o
…
```

4.5.3 在指定位置放置變量

1. 創建 main.c 文件，如下；

```
#include <stdio.h>
extern int sqr(int n1);
// 在 0x10000 處放置
const int gValue __attribute__((section(".ARM.__at_0x10000"))) = 3;
int main()
{
```

```
    int squared;
    squared=sqr(gValue);
    printf("Value squared is: %d\n", squared);
}
```

2. 創建 functon.c 源文件，如下：

```
int sqr(int n1)
{
    return n1*n1;
}
```

3. 創建 scatter 文件 scatter.scat 如下：

```
LR1 0x0
{
    ER1 0x0
    {
        *(+RO) ; 剩下的只讀代碼
    }
    ER2 +0
    {
        function.o
        *(.ARM.__at_0x10000) ; 放置 gValue 在 0x10000
    }
    ; RW 和 ZI 放置在 0x200000
    RAM 0x200000 (0x1FF00-0x2000)
    {
        *(+RW, +ZI)
    }
    ARM_LIB_STACK 0x800000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP +0 EMPTY 0x10000
    {
    }
}
```

4. 編譯並鏈接源文件

```
armcc -c -g function.c
armcc -c -g main.c
armlink --no_autoat --scatter=scatter.scat --map function.o main.o -o
squared.axf
```

從內存映射圖中，可以看到變量在 ER2 中的 0x10000 處

```
…
Execution Region ER2 (Base: 0x00001578, Size: 0x0000ea8c, Max: 
0xffffffff, ABSOLUTE)
Base Addr  Size       Type Attr Idx   E Section Name     Object
0x00001578 0x0000000c Code RO   3     .text              function.o
0x00001584 0x0000ea7c PAD
0x00010000 0x00000004 Data RO   15    .ARM.__at_0x10000  main.o
…
```

在這個例子中，ER1 的大小未知。因此，gValue 可能被放置 ER1 也可能放置在 ER2 中。

為了保證放在 ER2 中，你必須在 ER2 中包含對應的 section 匹配文字。並且在鏈接的時候，還必須指定--no_autoat 命令行選項。

如果忽略了--no_autoat 選項，gValue 將被單獨放置，對應於

`LR$$.ARM.__at_0x10000` 加載 region。

該 region 包含的執行 region 為

`ER$$.ARM.__at_0x10000`

注意：at 形式的縮寫。

```
//放置 variable1 在 .ARM.__AT_0x00008000 處
int variable1 __attribute__((at(0x8000))) = 10;
//放置 variable2 在.ARM.__at_0x8000 處
int variable2 __attribute__((section(".ARM.__at_0x8000"))) = 10;
```

上面的 section 名字，忽略大小寫

__at 具有如下的限制：

1. __at section 的地址範圍內不能覆蓋。
2. __at section 不準放在位置無關的執行 region 中
3. __at section 不能引用鏈接器定義的這些符號：

`$$Base,$$Limit,$$Length.`

4. __at section 不準用在 SysV,BPABI,以及 BPABI 的動態鏈接庫上。
5. __at section 的地址必須是對齊的整數倍

6. __at section 忽略+FIRST 後者+LAST

4.6 __at section 的自動放置

鏈接器自動放置__at section。當然也可以手動放置，在下一小節中介紹

鏈接器通過--autoat 指示鏈接器自動放置__at setcion。這個選項默認是打開的。

當使用--autoat 鏈接時，__at section 不會被放置在 scatter 文件中的與 section 模式字符串匹配的 region 中。而是將這個 section 放在一個兼容的 region 中。如果沒有兼容的 region，則創建兼容的 region。

帶有--autoat 選項的所有鏈接器，創建的 region 都有 UNINIT 屬性。如果需要將這個__at section 放置在一個 ZI region 中，則必須放置在兼容 region 中。

兼容 region 滿足如下條件：

1. __at 的地址剛好處在執行 region 的地址範圍內。如果一個 region 沒有設置最大大小，鏈接器將排除__at section 之後，計算大小，這個大小再加上一個常量作為其最後的大小。這個常量默認值為 10240 字節。他可以通過--max_er_extension 命令行選項來調整。
2. 這個執行 region 還需要滿足如下的條件：

- 具有模式字符串，並能夠匹配這個 section
- 至少有一個 section 和__at section 具有相同的類型（RO,RW,ZI）
- 沒有 EMPTY 屬性

來個例子：

```
//放置 RW 變量在叫做.ARM.__at_0x02000 的 section 中
int foo __attribute__((section(".ARM.__at_0x2000"))) = 100;
//放置 ZI 變量在.ARM.__at_0x4000 的 section 中
int bar __attribute__((section(".ARM.__at_0x4000"),zero_init));
//放置 ZI 變量在.ARM.__at_0x8000 的 section 中
int variable __attribute__((section(".ARM.__at_0x8000"),zero_init));
```

對應的 scatter 文件如下：

```
LR1 0x0
{
```

```
    ER_RO 0x0 0x2000
    {
        *(+RO) ; .ARM.__at_0x0000 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
    }
}
; 鏈接器為.ARM.__at_0x8000 創建一個加載和執行 region。因為它超出了所有候選 region 的大小。
```

4.7 手動放置__at section

使用--no_autoat 命令行選項，然後使用標準的模式匹配字符串去控制__at section 的放置。

舉例如下：

```
//放置 RO 變量在.ARM.__at_0x2000
const int FOO __attribute__((section(".ARM.__at_0x2000"))) = 100;
//放置 RW 變量在.ARM.__at_0x4000
int bar __attribute__((section(".ARM.__at_0x4000")));
```

對應的 scatter 文件如下：

```
LR1 0x0
{
    ER_RO 0x0 0x2000
    {
        *(+RO) ; .ARM.__at_0x0000 is selected by +RO
    }
    ER_RO2 0x2000
    {
        *(.ARM.__at_0x02000) ; .ARM.__at_0x2000 is selected by the section named
```

```
    ; .ARM.__at_0x2000
    }
    ER2 0x4000
    {
        *(+RW +ZI) ; .ARM.__at_0x4000 is selected by +RW
    }
}
```

4.8 使用__at 映射一個外設寄存器

為了將一個未初始化的變量映射為一個外設寄存器。可以使用 ZI __at section。

假設這個寄存器的地址為 0x10000000，定義一個 section 叫做.ARM.__at_0x10000000.如下：

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

手動放置的 scatter 文件如下：

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

4.9 使用.ANY 來放置未分配的 section

在大多數情況下，單個.ANY 等價於使用*。但是，.ANY 可以出現在多個執行 region 中。

4.9.1 多個.ANY 的放置規則

當使用多個.ANY 時，鏈接器有自己默認的規則來組織 section。

當多個.ANY 存在於 scatter 文件中時，鏈接器以 section 的大小，從大到小排序。

如果多個執行 region 都有相同特性（後文稱為等價）的.ANY,那麼這個 section 會被分配到具有最多可用空間的 region 中。

例如：

1. 如果有兩個等價的執行 region，一個大小為 0x2000，另一個沒有限制。那麼.ANY 匹配的 section 會放置在第二個中。
2. 如果有兩個等價的執行 region，一個大小為 0x2000，另一個為 0x3000. 那麼.ANY 匹配的 section 會先放置在第二個中，直到第二個的大小小於第一個。
   相當於這個兩個執行 region 在交替放置。

4.9.2 命令行選項控制多個.ANY 的放置

可以通過命令行選項，控制.ANY 的排序，下面的命令行選項是可用的:

1. --any_placement=algorithm algorithm 是如下之一：first_fit,worst_fit,best_fit,或者 next_fit
2. --any_sort_order=order.此處 order 是如下之一：cmdline 或者 descending_size

當你想要按照順序填充 region 時，使用 first_fit

當你想要填滿整個 region 時，使用 best_fit

當你想要均勻填充 region 時，使用 worst_fit

當你想要更精確的填充時，使用 next_fit

因為，鏈接器會產生 veneer 代碼以及填充數據，而這些代碼的是在.ANY 匹配之後產生的。所以，如果.ANY 將 region 填滿，則很有可能導致整個 regoin 無法放置，鏈接器產生的代碼。鏈接器會產生如下的錯誤。

```
Error: L6220E: Execution region regionname size (size bytes) exceeds limit (limit bytes)
```

--any_contingency 選項防止鏈接器將 region 的大小填滿。它保留 region 的一部分空間。當鏈接器產生的代碼沒有空間時，就使用這部分保留的空間。

first_fit 和 best_fit 默認打開這個選項。

4.9.3 優先級

.ANY 還可以指定優先級。

優先級通過後面接一個數字來表示，從 0 開始遞增。數字越大優先級越高。

例子如下：

```
lr1 0x8000 1024
{
    er1 +0 512
    {
        .ANY1(+RO) ; 優先級較低，和 er3 交替均勻填充
    }
    er2 +0 256
    {
        .ANY2(+RO) ; 優先級最高，先填充這個
    }
    er3 +0 256
    {
        .ANY1(+RO) ;優先級較低，和 er1 交替均勻填充
    }
}
```

4.9.4 指定.ANY 的最大大小

使用 ANY_SIZE max_size 指定最大大小。

例子如下：

```
LOAD_REGION 0x0 0x3000
{
    ER_1 0x0 ANY_SIZE 0xF00 0x1000
    {
        .ANY
    }
    ER_2 0x0 ANY_SIZE 0xFB0 0x1000
    {
        .ANY
    }
    ER_3 0x0 ANY_SIZE 0x1000 0x1000
    {
        .ANY
    }
```

```
}
```

上面例子中：

1. ER_1 有 0x100 的保留空間，該保留空間用於鏈接器產生的內容
2. ER_2 有 0x50 的保留空間
3. ER_3 沒有保留空間。region 將會被填滿。應該將 ANY_SIZE 的大小，限制在 region 大小的 98%以內。以預留 2%用於鏈接器產生的內容。

4.9.5 例子 1

有 6 個同樣大小的 section。如下：

名字	大小
sec1	0x4
sec2	0x4
sec3	0x4
sec4	0x4
sec5	0x4
sec6	0x4

對應的 scatter 文件如下：

```
LR 0x100
{
    ER_1 0x100 0x10
    {
        .ANY
    }
    ER_2 0x200 0x10
    {
        .ANY
    }
}
```

1. 對於 first_fit: 首先分配所有 section 到 ER_1 中，然後再是 ER_2 中
2. 對於 next_fit：跟 first_fit 一樣，但是 ER_1 會被填滿，然後被標記為 FULL。
3. 對於 best_fit: 首先 sec1 分配到 ER_1 中，然後 ER_2 和 ER_1 優先級相同，且 ER_2 空間比 ER_1 空間大，接著分配 sec2 到 ER_1 中。直到 ER_1 填滿

4. 對於 worst_fit:首先分配 sec1 到 ER_1 中，然後 ER_2 空間比 ER_1 大，接著分配 sec2 到 ER_2 中。剩下的兩個 region 空間一樣大，且優先級相同，然後選擇 scatter 的第一個，將 sec3 分配到 ER_1 中，依次類推。

4.9.6 例子 2——使用 next_fit

有下面的 section：

名字	大小
sec1	0x14
sec2	0x14
sec3	0x10
sec4	0x4
sec5	0x4
sec6	0x4

對應的 scatter 如下：

```
LR 0x100
{
    ER_1 0x100 0x20
    {
        .ANY1(+RO-CODE)
    }
    ER_2 0x200 0x20
    {
        .ANY2(+RO)
    }
    ER_3 0x300 0x20
    {
        .ANY3(+RO)
    }
}
```

詳細步驟如下：

1. 首先 sec1 被分配給 ER_1.因為 ER_1 有更佳的匹配。ER_1 現在還剩下 0x6 個字節

2. 鏈接器嘗試將 sec2 分配給 ER_1,因為它有更佳的匹配。但是 ER_1 沒有足夠的空間。因此 ER_1 被標記為 FULL，並且在後續的過程中再也不會考慮給 ER_1 分配 section。鏈接器選擇 ER_3,因為它有更高的優先級
3. 鏈接器嘗試將 sec3 分配給 ER_3，但是無法放入，因此被標記為 FULL，接著鏈接器將 sec3 放在 ER_2 中。
4. 鏈接器現在處理 sec4.它大小為 0x4，適合 ER_1 和 ER_3.但是這兩個在前面步驟中被標記為 FULL。因此剩下的 section 被放置在 ER_2 中。
5. 如果還有一個 section 叫做 sec7，且大小為 0x8.他將鏈接失敗。

4.9.7 例子三

有兩個文件 sections_a.o 和 sections_b.o，如下：

名字	大小
seca_1	0x4
seca_2	0x4
seca_3	0x10
seca_4	0x14
名字	**大小**
secb_1	0x4
secb_2	0x4
secb_3	0x10
secb_4	0x14

使用如下命令：

```
--any_sort_order=descending_size sections_a.o sections_b.o --scatter scatter.txt
```

排序之後，如下：

名字	大小
seca_4	0x14
secb_4	0x14
seca_3	0x10
secb_3	0x10
seca_1	0x4
seca_2	0x4
secb_1	0x4

名字	大小
secb_2	0x4

如果使用如下命令：

```
--any_sort_order=cmdline sections_a.o sections_b.o --scatter scatter.txt
```

排序之後如下：

名字	大小
seca_1	0x4
secb_1	0x4
seca_2	0x4
secb_2	0x4
seca_3	0x10
secb_3	0x10
seca_4	0x14
secb_4	0x14

4.10 控制 venner 的放置

在 scatter 文件中，還可以放置 venner 代碼。使用 Venner$$Code 來匹配 venner 代碼。

4.11 帶有 OVERLAY 屬性的放置

可以在同一個地址中，放置多個執行 region。因此在某一個時刻，只有一個執行 region 被激活。

如下面的例子：

```
EMB_APP 0x8000
{
    ...
    STATIC_RAM 0x0
    {
        *(+RW,+ZI)
    }
    OVERLAY_A_RAM 0x1000 OVERLAY
    {
```

```
        module1.o (+RW,+ZI)
    }
    OVERLAY_B_RAM 0x1000 OVERLAY
    {
        module2.o (+RW,+ZI)
    }
    ...
}
```

被 OVERLAY 標記的 region，在啓動的時候，不會被 c 庫初始化。而這部分內存的內容由 overlay 管理器負責。如果這部分 region 包含有初始化數據。需要使用 NOCOMPRESS 屬性來阻止 RW 數據的壓縮。

OVERLAY 屬性還可以用在單個執行 region 中，因此，這個 region 可以被用作：防止 c 庫初始化某個 region

OVERLAY region 也可以使用相對基址。如果他們有相同的偏移，則連續放置在一起。

如下例子：

```
EMB_APP 0x8000{
    CODE 0x8000
    {
        *(+RO)
    }
    # REGION1 的基址為 CODE 的結尾
    REGION1 +0 OVERLAY
    {
        module1.o(*)
    }
    # REGION2 的基址為 REGION1 的基址
    REGION2 +0 OVERLAY
    {
        module2.o(*)
    }
    # REGION3 的基址和 REGION2 的基址相同
    REGION3 +0 OVERLAY
    {
        module3.o(*)
```

```
    }
    # REGION4 的基址為 REGION3 的結尾+4
    Region4 +4 OVERLAY
    {
        module4.o(*)
    }
}
```

4.12 預留一個空 region

可以在 scatter 文件中，預留一個空的內存區域，比如：將此區域用於棧。使用 EMPTY 屬性可以達到此效果。

為了預留一個空的內存用於棧。對應的加載 region 沒有，執行 region 在執行時被分配。它被當做 dummy ZI region 對待，鏈接器使用下面的符號訪問它：

```
1. Image$$region_name$$ZI$$Base
2. Image$$region_name$$ZI$$Limit
3. Image$$region_name$$ZI$$Length
```

注意：dummy ZI region 在運行時並不會被初始化為 0

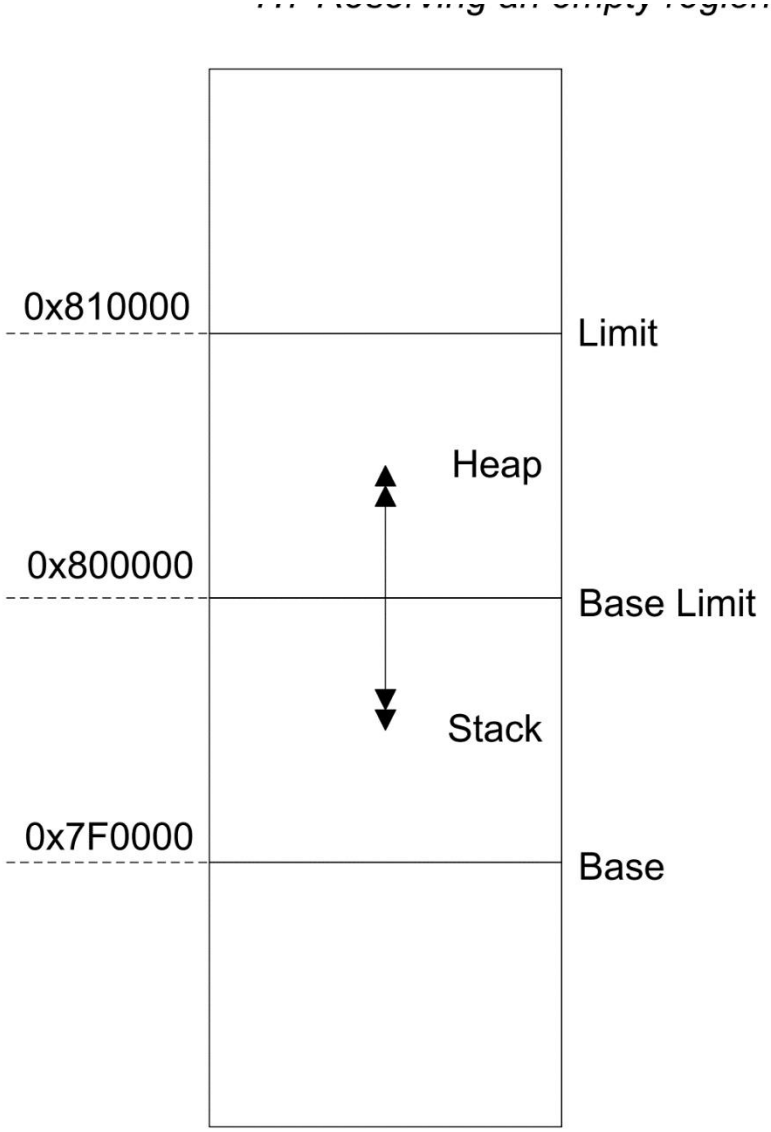
如果長度為負數，給定的地址就是結束地址。

例子如下：

```
LR_1 0x80000 ;加載 region 從 0x80000 開始
{
    STACK 0x800000 EMPTY -0x10000 ;region 結束地址為 0x800000,開始地址使用長度進行計算
    {
        ;空 region 用於放置棧
    }
    HEAP +0 EMPTY 0x10000 ; region 從上一個 region 結束處開始。
    {

    }
    ...
}
```

下圖展示了這個例子：



4.13 c 和 c++ 庫代碼的放置

可以在 scatter 文件中，放置 c 和 c++ 庫代碼。

在 scatter 文件中使用，*armlib* 或者 *cpplib* 來索引庫名字。一些 ARM c c++庫的 section 必須放在 root region 中。例如：__main.o,__scatter*.o,__dc*.o,*Region$$Table.

鏈接器可以在 InRoot$$Sections 中自動的，可靠的，放置這些 section。

例子 1 如下：

```
ROM_LOAD 0x0000 0x4000
{
    ROM_EXEC 0x0000 0x4000 ; 在 0x0 處的 root region
    {
        vectors.o (Vect, +FIRST) ; 向量表
        * (InRoot$$Sections) ; 所有的庫 section 必須放置在 root region 中。如
__main.o,__scatter*.o,__dc*.o,*Region$$Table
    }
    RAM 0x10000 0x8000
    {
        * (+RO, +RW, +ZI) ; 所有的其他的 section
    }
}
```

例子 2:arm c 庫的例子

```
ROM1 0
{
    * (InRoot$$Sections)
    * (+RO)
}
ROM2 0x1000
{
    *armlib/c_* (+RO) ; 所有 arm 支持的 c 庫函數
}
ROM3 0x2000
{
    *armlib/h_* (+RO) ; just the ARM-supplied __ARM_*
    ; redistributable library functions
}
RAM1 0x3000
{
    *armlib* (+RO) ; 其他的 arm 支持的庫，如，浮點庫
}
RAM2 0x4000
{
    * (+RW, +ZI)
}
```

名稱 ARM lib 表示位於 install_directory\lib\armlib 目錄中的 ARM C 庫文件

例子 3：arm c++ 庫代碼的放置

```
#include <iostrem>
using namespace std;
extern "C" int foo(){
    cout << "Hello" << endl;
    return 1;
}
```

為了放置 c++ 庫代碼，定義如下的 scatter 文件

```
LR 0x0
{
    ER1 0x0
    {
        *armlib*(+RO)
    }
    ER2 +0
    {
        *cpplib*(+RO)
        *(.init_array) ; .init_array 必須顯示放置，因為它被兩個 region 共享，鏈接器無法決定怎麼放置
    }
    ER3 +0
    {
        *(+RO)
    }
    ER4 +0
    {
        *(+RW,+ZI)
    }
}
```

名稱 install_directory\lib\armlib 表示位於 armlib 目錄中的 ARM C 庫文件

名稱 install_directory\lib\cpplib 表示位於 cpplib 目錄中的 ARM c++庫文件

4.14 scatter 文件的預處理

在 scatter 文件的第一行，設置一個預處理命令。然後鏈接器會調用相應的預處理器先處理這個文件。預處理命令的格式如下：

```
#! preprocessor [pre_processor_flags]
```

最常見的預處理命令如下：

```
#! armcc -E
```

舉例如下：

```
#! armcc -E
#define ADDRESS 0x20000000
#include "include_file_1.h"
lr1 ADDRESS
{

}
```

也可以在命令行中，進行預處理，如下：

```
armlink --predefine="-DADDRESS=0x20000000" --scatter=file.scat
```

armlink 系列完。

下一篇 從 arm 彙編到使用彙編點亮一個 LED。