ARM Cortex-M底层技术（十三）手把手教你写分散加载 - weixin_39118482的博客

还记得之前教大家写的启动代码吗？木看过滴，出门左转，第四篇【编写自己的启动代码】，当然仅仅能编写自己的启动代码怎么够，说了辣么多分散加载的东东，是时候检验一下我们的水平了，合上书，来出题考试了~【自己编写分散加载】。

来司机们，将装B进行到底~

首先，看看我们之前第四篇文章里面的简易版分散加载：

如下，之前按着没讲，前面罗里吧嗦的扯了辣么多分散加载，大家现在再回头看下这个分散加载，估计都看的明白了吧~

```
load_rom 0x00000000 0x00080000
{
        vector_rom 0x00000000 0x400
```

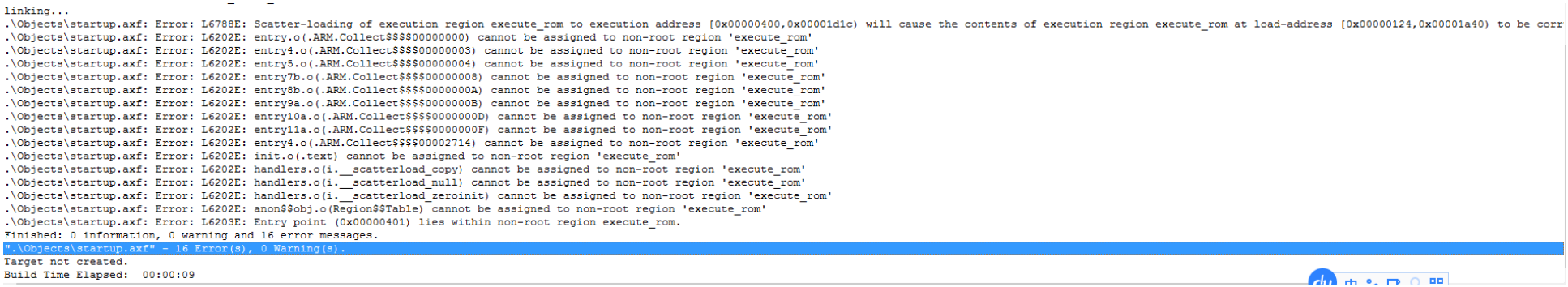
```
        {
                *( vector_table, +first)
        }

        execute_rom 0x400 FIXED 0x0007FC00
        {
                *( InRoot$$Sections )
                .any( +ro )
        }

        execute_data 0x20000000 0x00010000
        {
                .any ( +rw +zi )
        }

        ARM_LIB_HEAP  +0 empty 0x400 {}
        ARM_LIB_STACK 0x20020000  empty -400 {}
}
```

这里还是简单扯几点：

- 没有使用预处理器，比较Low逼；
- 就一个加载域、两个RO运行域（一个项链表运行域，一个根域）、一个RW+ZI运行域以及堆+栈的分配；
- 第三，参考以上两点~~~~

深入理解“FIXED”关键字

但是其实这里还是有一些技能点的~比如：根域的FIXED属性，你若把这个FIXED属性去掉，试试会发生什么？？？？



没错，链接器罢工了，16个错，看一下，主要是加载域和运行域的地址不匹配导致的错误；

原因是这样的：

1. 任何MCU的分散加载必须有一个根区，这点我们在之前讲过；
2. 根区是指加载域和运行域相同的地址的区（因为根区里面要放置C Library&分散加载相关代码，而分散加载本身不能被分散加载，所以根区的加载域与运行域必须相同）；
3. 程序的入口地址必须在根区中，因为程序的入口显然不能被分散加载，所以必须在根区中；
4. 如果运行域基址与加载域基址相同则默认可以看做根区（这点很好理解，参考第二点，根区就是运行域与加载域相同的区）；
5. 加载域后续的执行域指定“+0”偏移，则这些执行域默认都为根区（因为地址上是连续的，基址又与加载域基址相同）；
6. 如果基址不相同，则需要使用FIXED关键字指定根区；
7. FIXED关键字只能用于指定执行域，作用是确保执行域与加载域地址相同。

可能说了这么多还是比较费解，下面我们做一些实验来验证以上的罗里吧嗦看不懂的废话，做一下实验就都懂了：

实验一：

把所有不能被分散加载的代码内容放到与加载域地址相同的执行域里面去，如下：

```
1. load_rom 0x00000000 0x00080000
2. {
3.         vector_rom 0x00000000 0x2000
4.         {
5.                 *( vector_table, +first)
6.                 *( InRoot$$Sections )
```

```
        }
        execute_rom 0x2000 0x0007D000
        {
                .any( +ro )
        }
        execute_data 0x20000000 0x00010000
        {
                .any ( +rw +zi )
        }
        ARM_LIB_HEAP  +0 empty 0x400 {}
        ARM_LIB_STACK 0x20020000  empty -400 {}
}
```

- vector_rom显然是起始地址与加载域相同的执行域，默认的根区；
- 不能被分散加载的有：分散加载本身、部分C Library代码（注意不是全部，具体是哪些不行小编我也没研究辣么深入）、程序入口等；
- 把包含程序入口的Vector_table以及包含分散加载的InRoot$$Sections标号放人与加载域地址相同的执行域中，把不能被分散加载的部分与可以被分散加载的部分分开；

编译&链接，OK，有兴趣的可以自己试一下。

实验二：

看如下分散加载，先说结果，也可以完全正常的编译&链接运行的，至于为什么，前面写过的：

```
load_rom 0x00000000 0x00080000
{
        vector_rom 0x00000000 0x400
        {
                *( vector_table, +first)
        }
        execute_rom +0
        {
```

```
                *( InRoot$$Sections )
                .any( +ro )
        }
        execute_data 0x20000000 0x00010000
        {
                .any ( +rw +zi )
        }
        ARM_LIB_HEAP  +0 empty 0x400 {}
        ARM_LIB_STACK 0x20020000  empty -400 {}
}
```

当然还有最开始的直接加FIXED关键字的版本也是OK的。

然后我们来搞一个可以装逼版本的分散加载：

说是装逼其实也很简单，就是把预处理器用起来，在分散加载文件的顶格写下如下语句：

#! armcc -E

调用预处理器，然后开始使用预处理器耍流氓：

把之前直接赋值的地址数据替代掉，如：

```
#define m_interrupts_start             0x00000000
#define m_interrupts_size              0x00000400
```

然后分散加载就变成以下这个样子：

```
#! armcc -E
#if (defined(__ram_vector_table__))
  #define __ram_vector_table_size__    0x00000400
```

```
#else
  #define __ram_vector_table_size__    0x00000000
#endif
#define m_interrupts_start             0x00000000
#define m_interrupts_size              0x00000400
#define m_text_start                   0x00000400
#define m_text_size                    0x0007FC00
#define m_interrupts_ram_start         0x20000000
#define m_interrupts_ram_size          __ram_vector_table_size__
#define m_data_start                   (m_interrupts_ram_start + m_interrupts_ram_size)
#define m_data_size                    (0x00028000 - m_interrupts_ram_size)
#define m_usb_sram_start               0x40100000
#define m_usb_sram_size                0x00002000
/* USB BDT size */
#define usb_bdt_size                   0x0
/* Sizes */
#if (defined(__stack_size__))
  #define Stack_Size                   __stack_size__
#else
  #define Stack_Size                   0x0400
#endif
#if (defined(__heap_size__))
  #define Heap_Size                    __heap_size__
#else
  #define Heap_Size                    0x0400
#endif
LR_m_text m_interrupts_start m_text_start+m_text_size-m_interrupts_start { ; load region size_region
  VECTOR_ROM m_interrupts_start m_interrupts_size { ; load address = execution address
    * (RESET,+FIRST)
  }
  ER_m_text m_text_start FIXED m_text_size { ; load address = execution address
    * (InRoot$$Sections)
```

```
    .ANY (+RO)
  }
#if (defined(__ram_vector_table__))
  VECTOR_RAM m_interrupts_ram_start EMPTY m_interrupts_ram_size {
  }
#else
  VECTOR_RAM m_interrupts_start EMPTY 0 {
  }
#endif
  RW_m_data m_data_start m_data_size-Stack_Size-Heap_Size { ; RW data
    .ANY (+RW +ZI)
  }
  ARM_LIB_HEAP +0 EMPTY Heap_Size {    ; Heap region growing up
  }
  ARM_LIB_STACK m_data_start+m_data_size EMPTY -Stack_Size { ; Stack region growing down
  }
}
LR_m_usb_bdt m_usb_sram_start usb_bdt_size {
  ER_m_usb_bdt m_usb_sram_start UNINIT usb_bdt_size {
    * (m_usb_bdt)
  }
}
LR_m_usb_ram (m_usb_sram_start + usb_bdt_size) (m_usb_sram_size - usb_bdt_size) {
  ER_m_usb_ram (m_usb_sram_start + usb_bdt_size) UNINIT (m_usb_sram_size - usb_bdt_size) {
    * (m_usb_global)
  }
}
```

如上，比较专业的分散加载写法（不是我写的，小编我人懒，于是Copy了原厂的，不过跟我们之前写的简易分散加载结构是一样的，要点我们基本都讲到了，这里主要区别只是使用了预处理器的功能）

分散加载的部分暂时写到这里，一共写了6-7篇文章，当然还有很多内容没有覆盖到，以后我们碰到了再详细写几篇提高篇的内容，大部分基本原理讲清楚了，把这些内容消化掉，足够应付大多数应用了。

下面的文章我们会进入下一个大的专题，就是调试技术，也会有多篇文章，详细介绍深入的调试技巧。