ARM Cortex-M底层技术（十一）KEIL MDK 分散加载示例3-单独函数/变量的指定加载 - weixin_39118482的博客

本文链接：https://blog.csdn.net/weixin_39118482/article/details/80175284
分散加载示例3-单独函数/变量的指定加载

小编我一向主张在实战中学习，不主张直接去去学习规则&定义，太枯燥，在实际应用中去摸索，才会真正理解具体的技术细节，下面我们就通过实际的简单用例来搞清楚分散加载。

将函数和变量放到特定的指定加载地址的方法：

通常，编译器通过单个源文件生成RO、RW和ZI节。要将单个函数或者数据固定放在特定的地址上，我们必须允许链接器单独处理这个函数或数据并且与其他的部分分开。

一共有4种方法可以完成这个神奇的操作，我们会逐一介绍：

<1>使用--split_sections这个编译器选项，为每个函数单独生成一个节（分散加载操作以节为单位），再单独分配这些节；

<2>使用__attribute__((at(address)))来指定放置；

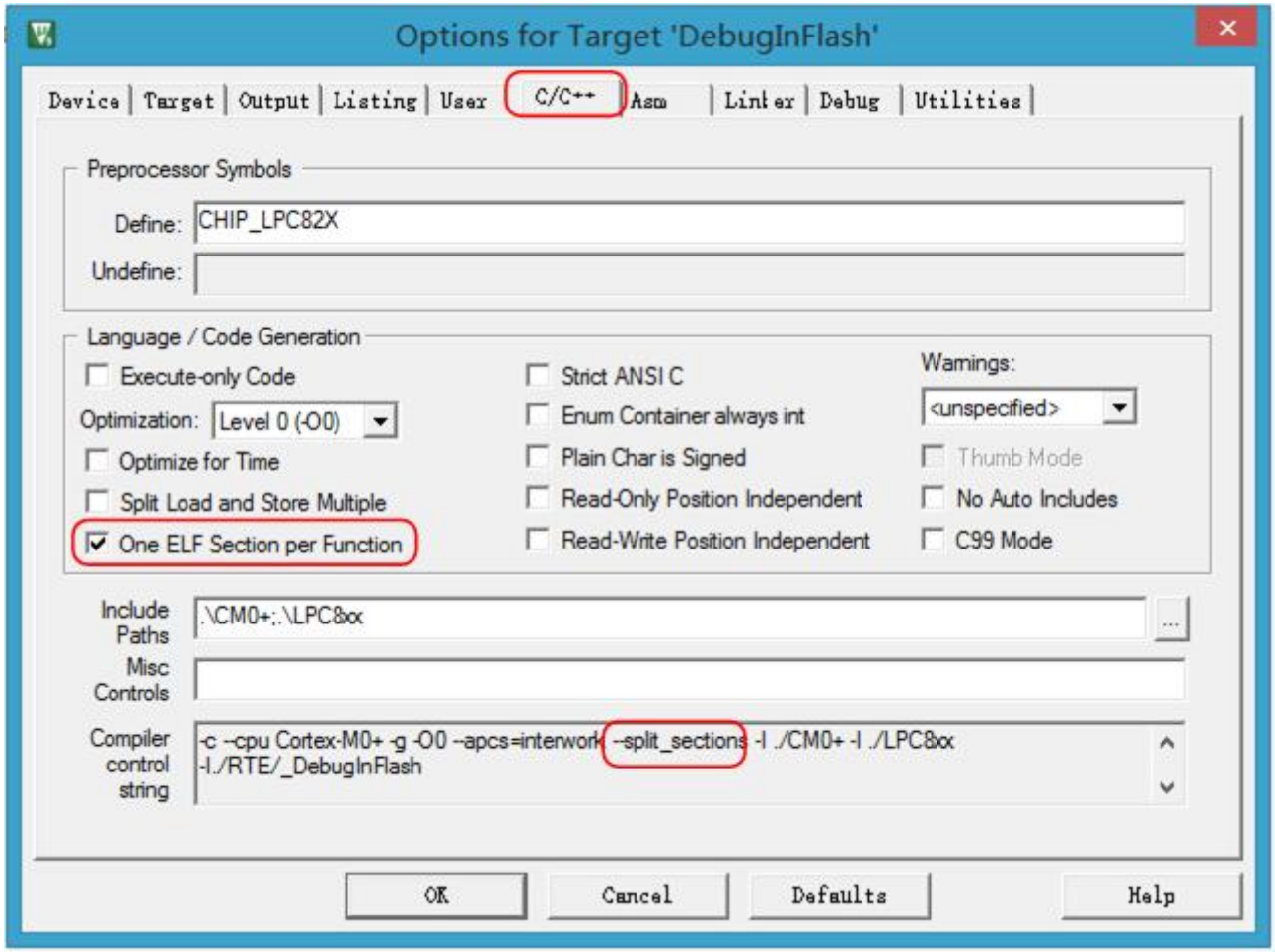
<3>使用__attribute__((section("name")))来放置一个被命名的节；

<4>使用汇编来指定生成节，然后再放置这些节。

汇编我们不讨论，因为很少有人用（其实是小编我汇编太差，搞不定……），我们重点讨论前3种方法。

使用--split_sections生成单独的ELF节

在如下图的位置上勾选“One ELF Section pre Function”然后在编译器控制串窗口里面，就是最下面的红色方框中就可以看到多了一个--split_sections这个编译器选项。



这个编译器选项啥意思？简单来说就是让每个函数生成一个独立的节。不勾选这个选项，是以文件为单位生成节的，比如一个main.c文件，会生成一个main.o的节，一个startup.s（汇编）也会生成一个startup.o的节，但并不是以函数为单位的。勾选了上图的选项够每个函数都会生成一个独立的节，我们看下以下的对比图就比较清晰了。

```
0x00000164   0x00000164   0x0000002c   Code   RO            51    .text               startup_lpc82x.o
0x00000190   0x00000190   0x0000008c   Code   RO            59    .text               main.o
0x0000021c   0x0000021c   0x00000006   Code   RO            93    .text               c_p.l(heapauxi.o)
0x00000222   0x00000222   0x0000003e   Code   RO           119    .text               c_p.l(sys_stackheap_outer.o)
0x00000260   0x00000260   0x00000010   Code   RO           122    .text               c_p.l(exit.o)
0x00000270   0x00000270   0x00000008   Code   RO           134    .text               c_p.l(libspace.o)
0x00000278   0x00000278   0x0000000c   Code   RO           191    .text               c_p.l(sys_exit.o)
0x00000284   0x00000284   0x00000002   Code   RO           210    .text               c_p.l(use_no_semi.o)
0x00000286   0x00000286   0x00000000   Code   RO           212    .text               c_p.l(indicate_semi.o)
0x00000286   0x00000286   0x0000001c   Code   RO            22    i.__ARM_common_switch8  system_lpc8xx.o
0x000002a2   0x000002a2   0x00000002   PAD
0x000002a4   0x000002a4   0x00000020   Data   RO           262    Region$$Table       anon$$obj.o
0x000002c4   0x000002c4   0x00000038   PAD
0x000002fc   0x000002fc   0x00000004   Code   RO            50    .ARM.__at_0x02FC    startup_lpc82x.o
0x00000300   0x00000300   0x00000224   Code   RO             3    .text               system_lpc8xx.o
0x00000524   0x00000524   0x0000015a   Code   RO            89    .text               c_p.l(aeabi_sdiv.o)
```

【图1】未勾选--split_sections的.map文件节选

```
0x00000164   0x00000164   0x0000002c   Code   RO            57    .text               startup_lpc82x.o
0x00000190   0x00000190   0x0000015a   Code   RO           107    .text               c_p.l(aeabi_sdiv.o)
0x000002ea   0x000002ea   0x00000006   Code   RO           111    .text               c_p.l(heapauxi.o)
0x000002f0   0x000002f0   0x00000008   Code   RO           152    .text               c_p.l(libspace.o)
0x000002f8   0x000002f8   0x00000002   Code   RO           228    .text               c_p.l(use_no_semi.o)
0x000002fa   0x000002fa   0x00000000   Code   RO           230    .text               c_p.l(indicate_semi.o)
0x000002fa   0x000002fa   0x00000002   PAD
0x000002fc   0x000002fc   0x00000004   Code   RO            56    .ARM.__at_0x02FC    startup_lpc82x.o
0x00000300   0x00000300   0x0000003e   Code   RO           137    .text               c_p.l(sys_stackheap_outer.o)
0x0000033e   0x0000033e   0x00000010   Code   RO           140    .text               c_p.l(exit.o)
0x0000034e   0x0000034e   0x00000002   PAD
0x00000350   0x00000350   0x0000000c   Code   RO           209    .text               c_p.l(sys_exit.o)
0x0000035c   0x0000035c   0x0000000c   Ven    RO           288    Veneer$$Code        anon$$obj.o
0x00000368   0x00000368   0x00000034   Code   RO            65    i.GPIOInit          main.o
0x0000039c   0x0000039c   0x000001a4   Code   RO             3    i.SystemCoreClockUpdate  system_lpc8xx.o
0x00000540   0x00000540   0x00000084   Code   RO             4    i.SystemInit        system_lpc8xx.o
0x000005c4   0x000005c4   0x0000001c   Code   RO            28    i.__ARM_common_switch8  system_lpc8xx.o
0x000005e0   0x000005e0   0x0000003c   Code   RO            66    i.main              main.o
0x0000061c   0x0000061c   0x00000020   Data   RO           280    Region$$Table       anon$$obj.o
```

【图2】勾选--split_sections的.map文件节选

如上，图1只有几个文件生成的节，而图2明显多出了几个节，比如i.GPIOInit以及i.main都成了独立的节，而在这之前，他们只是main.o这个节里面的一部分。***勾选了--split_sections选项生成的节都带有“i.”的前缀***，这个标记方便大家识别。

我们把各个函数独立成一个节之后，我们就可以在分散加载中放置他们了，比如我们把程序中的GPIOInit()函数放到SRAM中，分散加载需要按照如下调整：

```
1. LR_IROM1 0x00000000 0x00008000  {
2.   ER_IROM1 0x00000000 0x00008000  {
3.    *.o (RESET, +First)
4.    *(InRoot$$Sections)
5.    .ANY (+RO)
6.   }
7.   RW_IRAM1 0x10001000 0x00001000  {
8.    .ANY (+RW +ZI)
9.   }
10. }
11. LR_IROM2  0x10000000 0x00001000
12. {
13.   ER_IROM2 0x10000000 0x00001000{
14.         main.o(i.GPIOInit)
15.   }
16. }
```

我们验证下刚才的操作，查看.map文件：

```
GPIOInit                                0x10000001   Thumb Code    40  main.o(i.GPIOInit)
```

OK，GPIOInit函数被加载到了SRAM的地址上。

使用--split_sections生成单独的ELF节存在的问题

这种方式会导致一定量的存储空间的浪费，并会使性能略微降低。如果大家使用的是8KB、16KB、32KB这样Flash较小的MCU，不建议用这种方式来做，但如果Flash空间较大同时又需要制定很多函数、数据的地址的时候，这种方法就高效得多。

使用__attribute__((at(address)))来指定放置

这种比较简单，直接看示例：

```
int var1 __attribute__((at(0x10000000))) = 0x55;
```

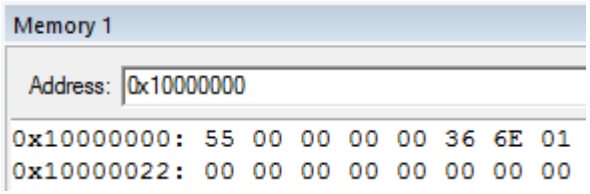
代码直接向0x10000000地址分配一个变量，并赋值0x55；

然后编译&链接，我们看下.map文件：

```
0x10000000   0x00000680   0x00000004   Data   RW           60    .ARM.__AT_0x10000000  main.o
0x10000004   0x00000684   0x00000004   Data   RW            4    .data                 system_lpc8xx.o
0x10000008         -       0x00000060   Zero   RW          138    .bss                  c_p.l(libspace.o)
0x10000068         -       0x00000000   Zero   RW           48    HEAP                  startup_lpc82x.o
0x10000068         -       0x00000200   Zero   RW           47    STACK                 startup_lpc82x.o
```

非常直接，直接多了一个.ARM.__AT_0x10000000的节，被放到了0x10000000的地址上。

DEBUG一下看看Memory数据：



0x10000000地址上0x55的值已经存在，证明变量被成功定位。

改一下程序我们再看看：

```
1. int var1 __attribute__((at(0x10000000))) = 0x55;
2. int var2 __attribute__((at(0x10001000))) = 0x55;
```

变量被成功定位。

按照如上再增加一个变量var2，并分配到0x10001000地址上，大家猜一下会是什么结果？会如愿达成我们的目的吗？

```
Build target 'DebugInFlash'
compiling main.c...
linking...
.\Out\DebugInFlash\LPC8xx.axf: Error: L6971E: main.o(.ARM.__AT_0x10001000) type RW incompatible with startup_lpc82x.o(STACK) type ZI in er RW_IRAM1.
Not enough information to list image symbols.
Not enough information to list load addresses in the image map.
Finished: 2 information, 0 warning and 1 error messages.
".\Out\DebugInFlash\LPC8xx.axf" - 1 Error(s), 0 Warning(s).
Target not created.
Build Time Elapsed:  00:00:01
```

抱歉，出错了，链接器报错（注意不是编译器是链接器报错）错误原因是地址冲突不匹配，而且是跟STACK的栈空间不匹配：

WHY？？？为什么var1就没事呢？

看出错提示来看是与栈空间冲突了，但实际还真不是这样的，这个错误的真正原因小编我其实到现在也没有真正找到，怀疑是在栈空间后面的地址空间不能分配其他节，因为var1成功了而var2却失败了，var2地址是在栈空间后面，我怀疑是栈空间之后不能正常分配导致的，但没有在官方文档里面找到确切的理论支持。

我尝试了下把STACK空间挪到后面，就OK了，分散加载如下（还需要配套修改启动代码，大家有兴趣可以自己试一试，启动代码修改的地方较多，我就不列出来了，有兴趣的根据我们之前文章的内容自己去尝试修改）：

```
1. LR_IROM1 0x00000000 0x00008000  {
2.   ER_IROM1 0x00000000 0x00008000  {
3.    *.o (RESET, +First)
4.    *(InRoot$$Sections)
5.    .ANY (+RO)
6.   }
7.   RW_IRAM1 0x10000000 0x00002000  {
8.    .ANY (+RW +ZI)
9.   }
10.  ARM_LIB_STACK 0x10002000 EMPTY -0x200{}
11. }
```

如上，我把STACK空间挪到了片内SRAM的尾部，再根据需要修改启动代码，重新编译，就可以成功，DEBUG一下也可以看到var1和var2都被正确的分散加载以及正确的赋值。

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运行域中使用 __at 节
- 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。

使用__attribute__((section("name")))来放置一个被命名的节

还是直接上代码：

```
int var __attribute__((section("mySection"))) = 0x55;
```

这里我们构建了一个名叫“mySection”的节（其实就是变量var的别称），然后我们修改对应的分散加载文件：

```
1. LR_IROM1 0x00000000 0x00008000  {
2.   ER_IROM1 0x00000000 0x00008000  {
3.    *.o (RESET, +First)
4.    *(InRoot$$Sections)
5.    .ANY (+RO)
6.   }
7.   ER_mySection 0x10000000
8.   {
9.         main.o(mySection)
10.  }
11.  RW_IRAM1 0x1000000 0x00002000  {
12.   .ANY (+RW +ZI)
13.  }
14.  ARM_LIB_STACK 0x10002000 EMPTY -0x200{}
15. }
```

编译&链接我们看一下结果（对应的.map文件）：

```
Execution Region ER_mySection (Exec base: 0x10000000, Load base: 0x00000680, Size: 0x00000004, Max: 0xffffffff, ABSOLUTE)

Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name        Object

0x10000000   0x00000680   0x00000004   Data   RW          59    mySection           main.o
```

可以看到运行域地址被放到0x10000000地址上了，加载地址在0x680上。这样我们就把var变量放到了我们指定的地址上了。

使用__attribute__((section("name")))来放置一个函数可以吗？

可以，而且用法一样，如下：

```
1. __attribute__((section("mySection"))) void GPIOInit (void)
2. {
3.     xxxx
4.     xxxx
5. }
```

然后按照之前一样的方法修改分散加载描述文件，可以把这个函数加载到指定地址上：

结果如下（对应的.map文件）：

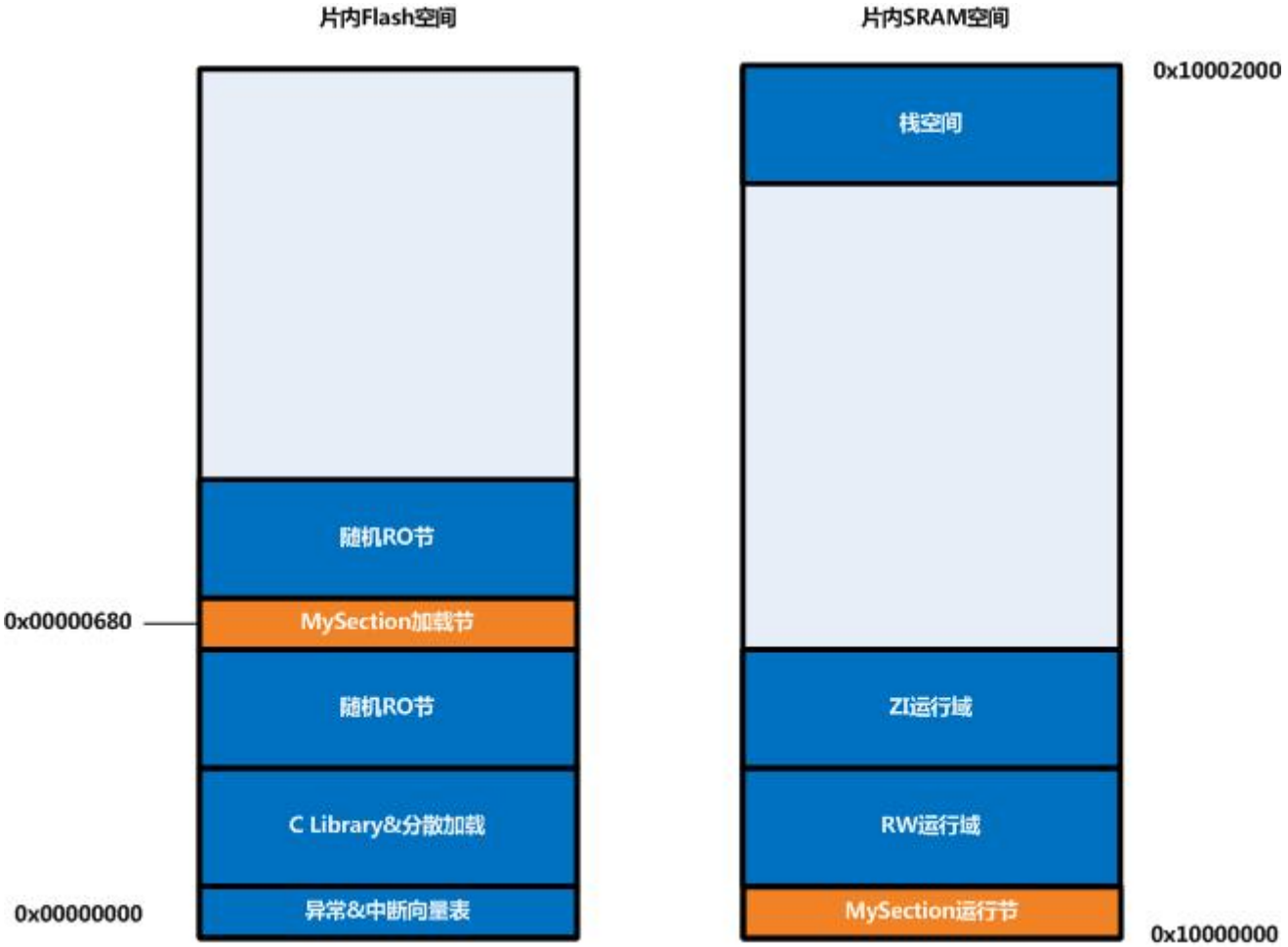
```
Execution Region ER_mySection (Exec base: 0x10000000, Load base: 0x00000680, Size: 0x00000034, Max: 0xffffffff, ABSOLUTE)

Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name        Object

0x10000000   0x00000680   0x00000034   Code   RO          59    mySection           main.o
```

可以直观的看到“mySection”这个节被加载到了0x10000000的运行域了，而且与上一个变量加载例子不同的是，类型变成了RO属性“Code”（上一个例子是RW属性的DATA），因为我们这次加载的是一个函数，所以类型当然是Code了。

过程执行完毕后，Memory内部空间分布示意图如下：


片内Flash空间
片内SRAM空间
0x10002000
栈空间
随机RO节
0x00000680
MySection加载节
随机RO节
ZI运行域
C Library&分散加载
RW运行域
0x00000000
异常&中断向量表
MySection运行节
0x10000000