ARM Cortex-M底层技术（十）KEIL MDK 分散加载示例2-代码加载到片内SRAM中运行以及部分规则 - weixin_39118482的博客

本文链接：https://blog.csdn.net/weixin_39118482/article/details/80162158

KEIL MDK 分散加载示例2-代码加载到片内SRAM中运行&部分规则

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。

简单示例

这个功能是非常有用的，这个小编我之前的文章提过，是一种可以显著增加程序运算速度的方法，所以比较常用，可以把核心算法相关的文件分散加载到SRAM区域上去以加快程序运行速度。

按照上个示例的方式修改分散加载，只是这次我们要把加载域以及RO运行域放到片内SRAM上。代码加载到SRAM上运行就意味着RO、RW、ZI都要放到SRAM上，这种用法显然对片内SRAM的大小是一个重大考验，当然我们的示例比较简单，重点在于展示用法，我们先来看把所有代码都加载到片内SRAM上的操作。当然这种用法比较暴力，小编我并不推荐这种玩法，只把对时间要求最高的部分代码加载到SRAM上才是真实会经常用到的玩法，这里我们先来看简单暴力的：把所有代码加载到SRAM上的玩法。

LPC824有8KB SRAM，我们简单点把前4KB作为程序的加载域以及RO运行域，后4KB作为RW+ZI的运行域，如下：

```
LR_IROM1 0x10000000 0x00001000  {    ; load region size_region
  ER_IROM1 0x10000000 0x00001000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
  }
  RW_IRAM1 0x10001000 0x00002000  {  ; RW data
   .ANY (+RW +ZI)
  }
}
```

修改对应的ini文件，如下：

```
FUNC void Setup () {
  SP = _RDWORD(0x10000000);                         // Setup Stack Pointer
  PC = _RDWORD(0x10000004);                         // Setup Program Counter
  _WDWORD(0xE000ED08, 0x10000000);                  // Setup Vector Table Offset Register
}

LOAD .\Out\DebugInRAM\LPC8xx.axf INCREMENTAL          // Download

Setup();                                          // Setup for Running
```

编译链接，然后我们看一下对应的.map文件（之前文章提过的）：

LPC8xx\startup\startup_lpc82x.s	0x00000000	Number	0	startup_lpc82x.o ABSOLUTE
LPC8xx\system_LPC8xx.c	0x00000000	Number	0	system_lpc8xx.o ABSOLUTE
USER CODE\\main.c	0x00000000	Number	0	main.o ABSOLUTE
USER CODE\main.c	0x00000000	Number	0	main.o ABSOLUTE
dc.s	0x00000000	Number	0	dc.o ABSOLUTE
.ARM.__at_0x02FC	0x000002fc	Section	4	startup_lpc82x.o(.ARM.__at_0x02FC)
RESET	0x10000000	Section	192	startup_lpc82x.o(RESET)
!!!main	0x100000c0	Section	8	__main.o(!!!main)
!!!scatter	0x100000c8	Section	60	__scatter.o(!!!scatter)
!!handler_copy	0x10000104	Section	26	__scatter_copy.o(!!handler_copy)
!!handler_zi	0x10000120	Section	28	__scatter_zi.o(!!handler_zi)
.ARM.Collect$$libinit$$00000000	0x1000013c	Section	2	libinit.o(.ARM.Collect$$libinit$$00000000)
.ARM.Collect$$libinit$$00000002	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$00000002)
.ARM.Collect$$libinit$$00000004	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$00000004)
.ARM.Collect$$libinit$$0000000A	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$0000000A)
.ARM.Collect$$libinit$$0000000C	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$0000000C)
.ARM.Collect$$libinit$$0000000E	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$0000000E)
.ARM.Collect$$libinit$$00000011	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$00000011)
.ARM.Collect$$libinit$$00000013	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$00000013)
.ARM.Collect$$libinit$$00000015	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$00000015)
.ARM.Collect$$libinit$$00000017	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$00000017)
.ARM.Collect$$libinit$$00000019	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$00000019)
.ARM.Collect$$libinit$$0000001B	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$0000001B)
.ARM.Collect$$libinit$$0000001D	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$0000001D)
.ARM.Collect$$libinit$$0000001F	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$0000001F)
.ARM.Collect$$libinit$$00000021	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$00000021)
.ARM.Collect$$libinit$$00000023	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$00000023)
.ARM.Collect$$libinit$$00000025	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$00000025)
.ARM.Collect$$libinit$$0000002C	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$0000002C)
.ARM.Collect$$libinit$$0000002E	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$0000002E)
.ARM.Collect$$libinit$$00000030	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$00000030)
.ARM.Collect$$libinit$$00000032	0x1000013e	Section	0	libinit2.o(.ARM.Collect$$libinit$$00000032)
.ARM.Collect$$libinit$$00000033	0x1000013e	Section	2	libinit2.o(.ARM.Collect$$libinit$$00000033)
.ARM.Collect$$libshutdown$$00000000	0x10000140	Section	2	libshutdown.o(.ARM.Collect$$libshutdown$$00000000)
.ARM.Collect$$libshutdown$$00000002	0x10000142	Section	0	libshutdown2.o(.ARM.Collect$$libshutdown$$00000002)
.ARM.Collect$$libshutdown$$00000004	0x10000142	Section	0	libshutdown2.o(.ARM.Collect$$libshutdown$$00000004)
.ARM.Collect$$libshutdown$$00000007	0x10000142	Section	0	libshutdown2.o(.ARM.Collect$$libshutdown$$00000007)
.ARM.Collect$$libshutdown$$0000000A	0x10000142	Section	0	libshutdown2.o(.ARM.Collect$$libshutdown$$0000000A)
.ARM.Collect$$libshutdown$$0000000C	0x10000142	Section	0	libshutdown2.o(.ARM.Collect$$libshutdown$$0000000C)
.ARM.Collect$$libshutdown$$0000000F	0x10000142	Section	0	libshutdown2.o(.ARM.Collect$$libshutdown$$0000000F)
.ARM.Collect$$libshutdown$$00000010	0x10000142	Section	2	libshutdown2.o(.ARM.Collect$$libshutdown$$00000010)
.ARM.Collect$$rtentry$$00000000	0x10000144	Section	0	__rtentry.o(.ARM.Collect$$rtentry$$00000000)
.ARM.Collect$$rtentry$$00000002	0x10000144	Section	0	__rtentry2.o(.ARM.Collect$$rtentry$$00000002)
.ARM.Collect$$rtentry$$00000004	0x10000144	Section	6	__rtentry4.o(.ARM.Collect$$rtentry$$00000004)
.ARM.Collect$$rtentry$$00000009	0x1000014a	Section	0	__rtentry2.o(.ARM.Collect$$rtentry$$00000009)

可以看到，除了CRP-KEY（加密位）以外的所有代码包括__main都被加载到了0x10000000起始的地址上。对比一下没改之前的，注意地址的变化：

```
  LPC8xx\\system_LPC8xx.c                  0x00000000   Number         0  system_lpc8xx.o ABSOLUTE
  LPC8xx\startup\startup_lpc82x.s          0x00000000   Number         0  startup_lpc82x.o ABSOLUTE
  LPC8xx\system_LPC8xx.c                   0x00000000   Number         0  system_lpc8xx.o ABSOLUTE
  USER CODE\\main.c                        0x00000000   Number         0  main.o ABSOLUTE
  USER CODE\main.c                         0x00000000   Number         0  main.o ABSOLUTE
  dc.s                                     0x00000000   Number         0  dc.o ABSOLUTE
  !!!main                                  0x000000c0   Section        8  __main.o(!!!main)
  !!!scatter                               0x000000c8   Section       60  __scatter.o(!!!scatter)
  !!handler_copy                           0x00000104   Section       26  __scatter_copy.o(!!handler_copy)
  !!handler_zi                             0x00000120   Section       28  __scatter_zi.o(!!handler_zi)
  .ARM.Collect$$libinit$$00000000          0x0000013c   Section        2  libinit.o(.ARM.Collect$$libinit$$00000000)
  .ARM.Collect$$libinit$$00000002          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$00000002)
  .ARM.Collect$$libinit$$00000004          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$00000004)
  .ARM.Collect$$libinit$$0000000A          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$0000000A)
  .ARM.Collect$$libinit$$0000000C          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$0000000C)
  .ARM.Collect$$libinit$$0000000E          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$0000000E)
  .ARM.Collect$$libinit$$00000011          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$00000011)
  .ARM.Collect$$libinit$$00000013          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$00000013)
  .ARM.Collect$$libinit$$00000015          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$00000015)
  .ARM.Collect$$libinit$$00000017          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$00000017)
  .ARM.Collect$$libinit$$00000019          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$00000019)
  .ARM.Collect$$libinit$$0000001B          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$0000001B)
  .ARM.Collect$$libinit$$0000001D          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$0000001D)
  .ARM.Collect$$libinit$$0000001F          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$0000001F)
  .ARM.Collect$$libinit$$00000021          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$00000021)
  .ARM.Collect$$libinit$$00000023          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$00000023)
  .ARM.Collect$$libinit$$00000025          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$00000025)
  .ARM.Collect$$libinit$$0000002C          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$0000002C)
  .ARM.Collect$$libinit$$0000002E          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$0000002E)
  .ARM.Collect$$libinit$$00000030          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$00000030)
  .ARM.Collect$$libinit$$00000032          0x0000013e   Section        0  libinit2.o(.ARM.Collect$$libinit$$00000032)
  .ARM.Collect$$libinit$$00000033          0x0000013e   Section        2  libinit2.o(.ARM.Collect$$libinit$$00000033)
  .ARM.Collect$$libshutdown$$00000000      0x00000140   Section        2  libshutdown.o(.ARM.Collect$$libshutdown$$00000000)
  .ARM.Collect$$libshutdown$$00000002      0x00000142   Section        0  libshutdown2.o(.ARM.Collect$$libshutdown$$00000002)
  .ARM.Collect$$libshutdown$$00000004      0x00000142   Section        0  libshutdown2.o(.ARM.Collect$$libshutdown$$00000004)
  .ARM.Collect$$libshutdown$$00000007      0x00000142   Section        0  libshutdown2.o(.ARM.Collect$$libshutdown$$00000007)
  .ARM.Collect$$libshutdown$$0000000A      0x00000142   Section        0  libshutdown2.o(.ARM.Collect$$libshutdown$$0000000A)
  .ARM.Collect$$libshutdown$$0000000C      0x00000142   Section        0  libshutdown2.o(.ARM.Collect$$libshutdown$$0000000C)
  .ARM.Collect$$libshutdown$$0000000F      0x00000142   Section        0  libshutdown2.o(.ARM.Collect$$libshutdown$$0000000F)
  .ARM.Collect$$libshutdown$$00000010      0x00000142   Section        2  libshutdown2.o(.ARM.Collect$$libshutdown$$00000010)
```

跑一下程序，完美运行。

上面示例存在的问题

上面示例有一个致命问题，导致其完全无法在实际中使用，细心的童鞋估计看粗来了，代码的加载域在SRAM上，而SRAM的数据掉电就会丢失，使得上面的这个玩法完全不具备实用性，那么让加载域在Flash上，RO执行域在SRAM这样再不同的地址空间上可行吗？我们试一试：

```
LR_IROM1 0x00000000 0x00008000  {    ; load region size_region
  ER_IROM1 0x10000000 0x00001000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
  }
  RW_IRAM1 0x10001000 0x00001000  {  ; RW data
   .ANY (+RW +ZI)
  }
}
```

如上：很简单，加载域在Flash上，RO运行域在片内SRAM上的前4KB空间上，RW+ZI运行域在后4KB的SRAM上，它既解决了上面那个示例掉电的问题，又解决了运行速度问题。

关键是能运行吗？编译&链接看看？

```
linking...
.\Out\DebugInFlash\LPC8xx.axf: Error: L6202E: __main.o(!!!main) cannot be assigned to non-root region 'ER_IROM1'
.\Out\DebugInFlash\LPC8xx.axf: Error: L6202E: __scatter.o(!!!scatter) cannot be assigned to non-root region 'ER_IROM1'
.\Out\DebugInFlash\LPC8xx.axf: Error: L6202E: __scatter_copy.o(!!handler_copy) cannot be assigned to non-root region 'ER_IROM1'
.\Out\DebugInFlash\LPC8xx.axf: Error: L6202E: __scatter_zi.o(!!handler_zi) cannot be assigned to non-root region 'ER_IROM1'
.\Out\DebugInFlash\LPC8xx.axf: Error: L6202E: anon$$obj.o(Region$$Table) cannot be assigned to non-root region 'ER_IROM1'
.\Out\DebugInFlash\LPC8xx.axf: Error: L6203E: Entry point (0x100000c1) lies within non-root region ER_IROM1.
.\Out\DebugInFlash\LPC8xx.axf: Error: L6221E: Load region LR_IROM1 with Load range [0x00000000,0x000006bc) overlaps with Load region LR$$.ARM.__at_0x02FC with Load range [0x000002fc,0x00000300).
Finished: 0 information, 0 warning and 7 error messages.
".\Out\DebugInFlash\LPC8xx.axf" - 7 Error(s), 0 Warning(s).
Target not created.
Build Time Elapsed:  00:00:00
```

如上：链接器不干了，一火车皮的错误，最多的提示是“non-root region xxxxxx”这是啥？？？？

WHY？？

把代码放到Flash上存储，解决掉电问题，运行放在SRAM上解决速度问题。但上面的分散加载呢？加载域的起始地址是Flash的起始地址，运行域的起始地址是SRAM上，看似跟我们的分散加载描述具备一致性，但为什么不行呢？

原因在于分散加载本身不能被分散加载，好绕口，啥意思？比如你拍照（只有一台相机），但是你却想把自己的相机也拍到照片里面，这显然是

做不到的。分散加载也一样，它是一段在__main()中的代码（之前启动代码部分文章多次提过），这段代码会被一起烧写到MCU里面，把各个数据段搬到指定的位置上也是由分散加载来执行的，但我们再仔细看一下这个分散加载：

```
LR_IROM1 0x00000000 0x00008000 {    ; load region size_region
  ER_IROM1 0x10000000 0x00001000 { ; load address = execution address
      xxxx
  }
  RW_IRAM1 0x10001000 0x00001000 { ; RW data
      xxxx
  }
}
```

所有的RO执行域都被放到了0x10000000起始大小为0x1000的地址上面了，这个工作显然是要分散加载来做的，但是分散加载本身又在起始地址为0x10000000大小为0x1000的这个本身就需要被分散加载的空间上了，这就是矛盾！也就是分散加载本身不能被分散加载的意思了（分散加载受到的一个限制是负责创建执行区的代码和数据不能将自己复制到另一个位置）。*所以在Keil的分散加载中，第一个RO运行域必须与加载域基地址相同*。用这种方法来确保分散加载可以被正确的找到以及执行。

修改一下再运行看看

了解了问题原因，我们再修改下上面的示例：

```
LR_IROM1 0x00000000 0x00008000  {    ; load region size_region
  ER_IROM1 0x00000000 0x00001000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
  }
  ER_IROM2 0x10000000 0x00001000{
    .ANY (+RO)
  }
  RW_IRAM1 0x10001000 0x00001000  {  ; RW data
   .ANY (+RW +ZI)
  }
}
```

如上示例中，我们强制性的把RO运行域中分散加载的代码段放到了第一个RO运行域中而且起始地址与加载域一致。用户代码的RO段我们放到SRAM上，我们编译链接看一下。这里就直接说结论，编译链接可以过，但是运行之后程序会死在__main里面，WHY？？？？

原因是这样的，ER_IROM2这个RO执行域是在程序运行起来之后再分散加载到SRAM上面的，辣么也就是要向加载域里面写数据，这显然是不行的，所以我们需要如下修改：

```
LR_IROM1 0x00000000 0x00008000  {    ; load region size_region
  ER_IROM1 0x00000000 0x00008000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
  }
  RW_IRAM1 0x10001000 0x00001000  {  ; RW data
   .ANY (+RW +ZI)
  }
}

LR_IROM2  0x10000000 0x00001000
{
  ER_IROM2 0x10000000 0x00001000{
    .ANY (+RO)
  }
}
```

DEBUG，程序完美运行。

“InRoot$$Sections”与“根区”的说明：

我们到现在为止看到了许多分散加载都有“InRoot$$Sections”这个节，这是啥东东呢？

我们上面提到过分散加载受到的一个限制是负责创建执行区的代码和数据不能将自己复制到另一个位置，***分散加载需要有一个根区，根区本质是一个RO执行域，其执行地址与其加载地址相同***。

根区中需要包含以下几个节：

<1>复制代码和数据的程序代码，也就是__main.o和__scatter*.o(__scatter.o、__scatter_copy.o、__scatter_zi.o等)

<2>执行压缩的__dc*.o（同样包含很多以__dc开头的obj文件）

<3>Region$$Table节，包含要复制和压缩的带么和数据的地址

如果要在其他非根区内指定*（+RO）那么需要使用“InRoot$$Sections”来在根区显示的指定这些节。

简单来说，就是每个分散加载都要有一个根区，根区需要指定“InRoot$$Sections”节或者等效的其他节（简单来说就是分散加载、 main、数据压缩这些东西必须要放到根区）。

这个等效节是指什么？

比如在我们的这个简单示例里面，不过不用“InRoot$$Sections”来指定根区，那么如下的编写方式也是OK的：

```
ER_IROM1 0x00000000 0x00008000  {  ; load address = execution address
  *.o (RESET, +First)
  __main.o
  __scatter.o
  __scatter_copy.o
  __scatter_zi.o
  anon$$obj.o
}
```

不同程序这里写法不同，在我们的简单示例里面这样足够，具体哪些部分要放到根区里面，可以直接去看.map文件把链接到工程中的C Library的obj文件放进来就可以了，RT以及浮点的部分与之无关。如果你不想这么麻烦就直接使用“InRoot$$Sections”标号来指定根区也可以。