零死角玩转STM32

MDK的编译过程及文件类型全解

淘宝：firestm32.taobao.com

论坛： www.firebbs.cn

扫描进入淘宝店铺

主讲内容

01 编译过程

02 程序的组成、存储与运行

03 编译工具链

04 MDK工程的文件类型

05 实验：自动分配变量到外部SRAM

06 实验：优先使用内部SRAM并分配堆到外部SRAM

MDK的编译过程及文件类型全解

4. o、axf及elf文件

.o、.elf、*.axf、*.bin及*.hex文件都存储了编译器根据源代码生成的机器码，根据应用场合的不同，它们又有所区别。

ELF文件说明

.o、.elf、*.axf以及前面提到的lib文件都是属于目标文件，它们都是使用ELF格式来存储的，关于ELF格式的详细内容请参考配套资料里的《ELF文件格式》文档了解，它讲解的是Linux下的ELF格式，与MDK使用的格式有小区别，但大致相同。在本教程中，仅讲解ELF文件的核心概念。

ELF是Executable and Linking Format的缩写，译为可执行链接格式，该格式用于记录目标文件的内容。在Linux及Windows系统下都有使用该格式的文件(或类似格式)用于记录应用程序的内容，告诉操作系统如何链接、加载及执行该应用程序。

MDK的编译过程及文件类型全解

ELF文件说明

目标文件主要有如下三种类型：

- **可重定位的文件(Relocatable File)**，包含基础代码和数据，但它的代码及数据都没有指定绝对地址，因此它适合于与其他目标文件链接来创建可执行文件或者共享目标文件。 这种文件一般由编译器根据源代码生成。

  例如MDK的armcc和armasm生成的*.o文件就是这一类，另外还有Linux的*.o 文件，Windows的 *.obj文件。

MDK的编译过程及文件类型全解

ELF文件说明

- **可执行文件(Executable File)**，它包含适合于执行的程序，它内部组织的代码数据都有固定的地址(或相对于基地址的偏移)，系统可根据这些地址信息把程序加载到内存执行。这种文件一般由链接器根据可重定位文件链接而成，它主要是组织各个可重定位文件，给它们的代码及数据一一打上地址标号，固定其在程序内部的位置，链接后，程序内部各种代码及数据段不可再重定位(即不能再参与链接器的链接)。

例如MDK的armlink生成的*.elf及*.axf文件，(使用gcc编译工具可生成*.elf文件，用armlink生成的是*.axf文件，*.axf文件在*.elf之外，增加了调试使用的信息，其余区别不大，后面我们仅讲解*.axf文件)，另外还有Linux的/bin/bash文件，Windows的*.exe文件。

MDK的编译过程及文件类型全解

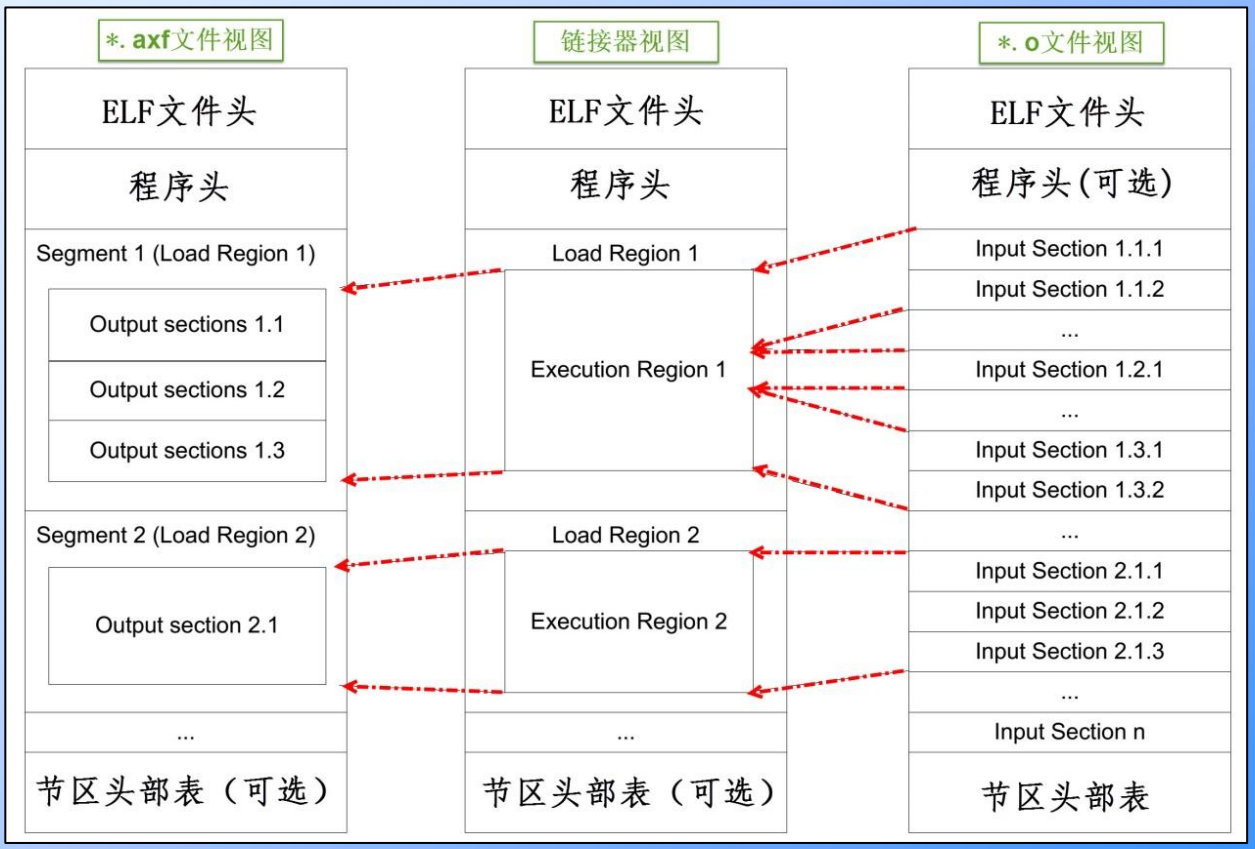
ELF文件说明

- **共享目标文件(Shared Object File)**， 它的定义比较难理解，我们直接举例，MDK生成的*.lib文件就属于共享目标文件，它可以继续参与链接，加入到可执行文件之中。另外，Linux的.so，如/lib/ glibc-2.5.so，Windows的DLL都属于这一类。

MDK的编译过程及文件类型全解

o文件与axf文件的关系

根据上面的分类，我们了解到，*.axf文件是由多个*.o文件链接而成的，而*.o文件由相应的源文件编译而成，一个源文件对应一个*.o文件。它们的关系如下：



MDK的编译过程及文件类型全解

o文件与axf文件的关系

图中的中间代表的是armlink链接器，在它的右侧是输入链接器的*.o文件，左侧是它输出的*axf文件。

可以看到，由于都使用ELF文件格式，*.o与*.axf文件的结构是类似的，它们包含ELF文件头、程序头、节区(section)以及节区头部表。各个部分的功能说明如下：

- ELF文件头用来描述整个文件的组织，例如数据的大小端格式，程序头、节区头在文件中的位置等。
- 程序头告诉系统如何加载程序，例如程序主体存储在本文件的哪个位置，程序的大小，程序要加载到内存什么地址等等。MDK的可重定位文件*.o不包含这部分内容，因为它还不是可执行文件，而armlink输出的*.axf文件就包含该内容了。

MDK的编译过程及文件类型全解

o文件与axf文件的关系

- 节区是*.o文件的独立数据区域，它包含提供给链接视图使用的大量信息，如指令(Code)、数据(RO、RW、ZI-data)、符号表(函数、变量名等)、重定位信息等，例如每个由C语言定义的函数在*.o文件中都会有一个独立的节区；
- 存储在最后的节区头则包含了本文件节区的信息，如节区名称、大小等等。

总的来说，链接器把各个*.o文件的节区归类、排列，根据目标器件的情况编排地址生成输出，汇总到*.axf文件。

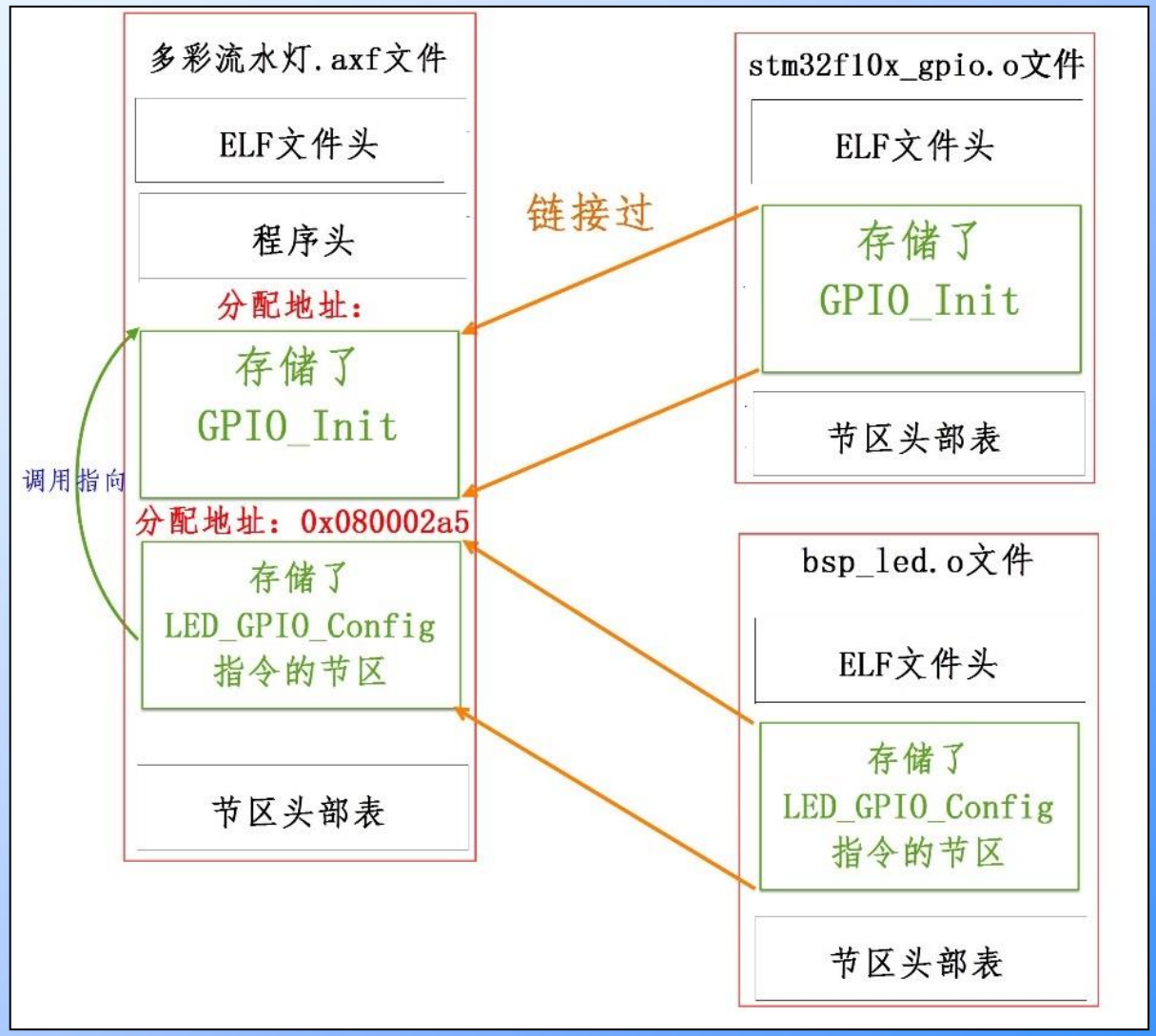
MDK的编译过程及文件类型全解

o文件与axf文件的关系

例如：“多彩流水灯”工程中在“bsp_led.c”文件中有一个LED_GPIO_Config函数，而它内部调用了“stm32f10x_gpio.c”的GPIO_Init函数，经过armcc编译后，LED_GPIO_Config及GPIO_Iint函数都成了指令代码，分别存储在bsp_led.o及stm32f10x_gpio.o文件中，这些指令在*.o文件都没有指定地址，仅包含了内容、大小以及调用的链接信息，而经过链接器后，链接器给它们都分配了特定的地址，并且把地址根据调用指向链接起来。

MDK的编译过程及文件类型全解

o文件与axf文件的关系

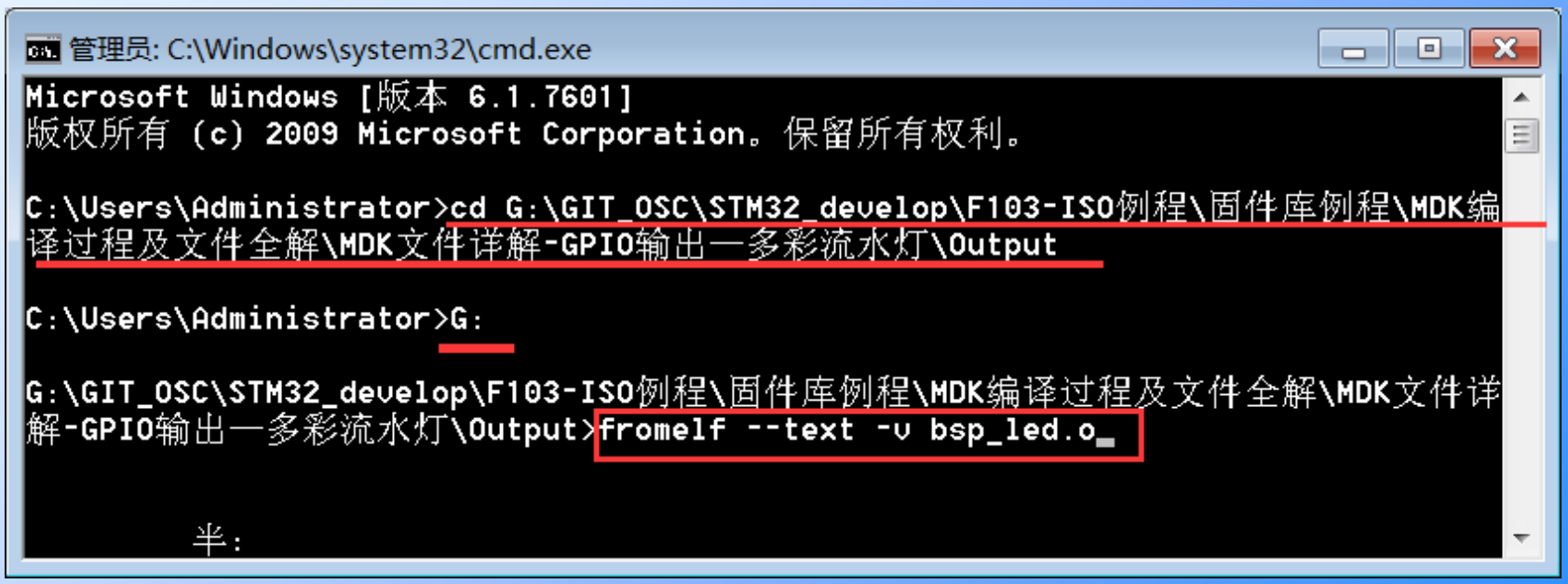


MDK的编译过程及文件类型全解

ELF文件头

接下来可以看看具体文件的内容，使用fromelf文件可以查看*.o、*.axf及*.lib文件的ELF信息。

使用命令行，切换到文件所在的目录，输入“fromelf –text –v bsp_led.o”命令，可控制输出bsp_led.o的详细信息，利用“-c、-z”等选项还可输出反汇编指令文件、代码及数据文件等信息，可亲手尝试一下。



MDK的编译过程及文件类型全解

ELF文件头

为了便于阅读，我已使用fromelf指令生成了“多彩流水灯.axf”、“bsp_led”及“多彩流水灯.lib”的ELF信息，并已把这些信息保存在独立的文件中，在配套资料的“elf信息输出”文件夹下可查看：

fromelf选项	可查看的信息	生成到配套资料里相应的文件
-v	详细信息	bsp_led_o_elfInfo_v.txt/流水灯_axf_elfInfo_v.txt
-a	数据的地址	bsp_led_o_elfInfo_a.txt/流水灯_axf_elfInfo_a.txt
-c	反汇编代码	bsp_led_o_elfInfo_c.txt/流水灯_axf_elfInfo_c.txt
-d	data section的内容	bsp_led_o_elfInfo_d.txt/流水灯_axf_elfInfo_d.txt
-e	异常表	bsp_led_o_elfInfo_e.txt/流水灯_axf_elfInfo_e.txt
-g	调试表	bsp_led_o_elfInfo_g.txt/流水灯_axf_elfInfo_g.txt
-r	重定位信息	bsp_led_o_elfInfo_r.txt/流水灯_axf_elfInfo_r.txt
-s	符号表	bsp_led_o_elfInfo_s.txt/流水灯_axf_elfInfo_s.txt
-t	字符串表	bsp_led_o_elfInfo_t.txt/流水灯_axf_elfInfo_t.txt
-y	动态段内容	bsp_led_o_elfInfo_y.txt/流水灯_axf_elfInfo_y.txt
-z	代码及数据的大小信息	bsp_led_o_elfInfo_z.txt/流水灯_axf_elfInfo_z.txt

MDK的编译过程及文件类型全解

ELF文件头

直接打开“elf信息输出”目录下的bsp_led_o_elfInfo_v.txt文件，可看到如下内容：

代码清单 42-1 bsp_led.o 文件的 ELF 文件头(可到“bsp_led_o_elfInfo_v.txt”文件查看)

```
1
2  ==============================================================================
3
4  ** ELF Header Information
5
6  File Name:
7  bsp_led.o       //bsp_led.o 文件
8
9  Machine class: ELFCLASS32 (32-bit)  //32 位机
10     Data encoding: ELFDATA2LSB (Little endian) //小端格式
11     Header version: EV_CURRENT (Current version)
12     Operating System ABI: none
13     ABI Version: 0
14     File Type: ET_REL (Relocatable object) (1) //可重定位类型
15     Machine: EM_ARM (ARM)
16
17     Entry offset (in SHF_ENTRYSECT section): 0x00000000
18     Flags: None (0x05000000)
19
20     ARM ELF revision: 5 (ABI version 2)
21
22     Header size: 52 bytes (0x34)
23     Program header entry size: 0 bytes (0x0)  //程序头大小
24     Section header entry size: 40 bytes (0x28)
25
26     Program header entries: 0
27     Section header entries: 178
28
29     Program header offset: 0 (0x00000000) //程序头在文件中的位置(没有程序头)
30     Section header offset: 378972 (0x0005c85c)//节区头在文件中的位置
31
32     Section header string table index: 175
33
34
==============================================================================
```

MDK的编译过程及文件类型全解

ELF文件头

在上述代码中已加入了部分注释，解释了相应项的意义，值得一提的是在这个*.o文件中，它的ELF文件头中告诉我们它的程序头(Program header)大小为“0 bytes”，且程序头所在的文件位置偏移也为“0”，这说明它是没有程序头的。

MDK的编译过程及文件类型全解

程序头

接下来打开“流水灯_axf_elfInfo_v.txt”文件，查看工程的*.axf文件的详细信息：

代码清单 42-2 *.axf 文件中的 elf 文件头及程序头(可到“流水灯_axf_elfInfo_v.txt”文件查看)

```
1
2  ============================================================================
3
4  ** ELF Header Information
5
6  File Name:
7  流水灯.axf            //流水灯.axf 文件
8
9  Machine class: ELFCLASS32 (32-bit)       //32 位机
10     Data encoding: ELFDATA2LSB (Little endian)  //小端格式
11     Header version: EV_CURRENT (Current version)
12     Operating System ABI: none
13     ABI Version: 0
14     File Type: ET_EXEC (Executable) (2)      //可执行文件类型
15     Machine: EM_ARM (ARM)
16
17     Image Entry point: 0x08000131
18     Flags: EF_ARM_HASENTRY (0x05000002)
19
20     ARM ELF revision: 5 (ABI version 2)
21
22     Built with
23 Component: ARM Compiler 5.05 update 2 (build 169) Tool: armasm [4d0f2f]
24 Component: ARM Compiler 5.05 update 2 (build 169) Tool: armlink [4d0f33]
25
26     Header size: 52 bytes (0x34)
27     Program header entry size: 32 bytes (0x20)  //程序头大小
28     Section header entry size: 40 bytes (0x28)
29
30     Program header entries: 1
31     Section header entries: 16
32
33     Program header offset: 279836 (0x0004451c)  //程序头在文件中的位置
34     Section header offset: 279868 (0x0004453c)  //节区头在文件中的位置
35
36     Section header string table index: 15
37
38     ============================================================================
39
40     ** Program header #0
41
42     Type           : PT_LOAD (1)      //表示这是可加载的内容
43     File Offset    : 52 (0x34)        //在文件中的偏移
44     Virtual Addr   : 0x08000000       //虚拟地址(此处等于物理地址)
45     Physical Addr  : 0x08000000       //物理地址
46     Size in file   : 3176 bytes (0xc68)  //程序在文件中占据的大小
47     Size in memory: 4200 bytes (0x1068) //若程序加载到内存，占据的内存空间
48     Flags          : PF_X + PF_W + PF_R + PF_ARM_ENTRY (0x80000007)
49     Alignment      : 8                //地址对齐
```

MDK的编译过程及文件类型全解

程序头

```
38      ===================================================================
39
40      ** Program header #0
41
42      Type           : PT_LOAD (1)      //表示这是可加载的内容
43      File Offset    : 52 (0x34)        //在文件中的偏移
44      Virtual Addr   : 0x08000000       //虚拟地址(此处等于物理地址)
45      Physical Addr  : 0x08000000       //物理地址
46      Size in file   : 3176 bytes (0xc68)   //程序在文件中占据的大小
47      Size in memory: 4200 bytes (0x1068) //若程序加载到内存，占据的内存空间
48      Flags          : PF_X + PF_W + PF_R + PF_ARM_ENTRY (0x80000007)
49      Alignment      : 8                //地址对齐
```

对比之下，可发现*.axf文件的ELF文件头对程序头的大小说明为非0值，且给出了它在文件的偏移地址，在输出信息之中，包含了程序头的详细信息。可看到，程序头的“Physical Addr”描述了本程序要加载到的内存地址“0x08000000”，正好是STM32内部FLASH的首地址；“size in file”描述了本程序占据的空间大小为“3176 bytes”，它正是程序烧录到FLASH中需要占据的空间

MDK的编译过程及文件类型全解

节区头

在ELF的原文件中，紧接着程序头的一般是节区的主体信息，在节区主体信息之后是描述节区主体信息的节区头，先来看看节区头中的信息了解概况。通过对比*.o文件及*.axf文件的节区头部信息，可以清楚地看出这两种文件的区别。

代码清单 42-3 *.o 文件的节区信息(“bsp_led_o_elfInfo_v.txt”文件)

```
1  ============================================================================
2  ** Section #1
3
4  Name         :
5  i.LED_GPIO_Config      //节区名
6  //此节区包含程序定义的信息，其格式和含义都由程序来解释。
7  Type         : SHT_PROGBITS (0x00000001)
9  //此节区在进程执行过程中占用内存。 节区包含可执行的机器指令。
10 Flags        : SHF_ALLOC + SHF_EXECINSTR (0x00000006)
12 Addr         : 0x00000000     //地址
13 File Offset  : 52 (0x34)      //在文件中的偏移
14 Size         : 96 bytes (0x60)    //大小
15 Link         :
16 SHN_UNDEF
17 Info         : 0
18 Alignment    : 4          //字节对齐
19 Entry Size   : 0
20 ====================================
```

MDK的编译过程及文件类型全解

节区头

这个节区头描述的是该函数被编译后的节区信息，其中包含了节区的类型(指令类型SHT_PROGBITS)、节区应存储到的地址(0x00000000)、它主体信息在文件位置中的偏移(52)以及节区的大小(96 bytes)。

由于*.o文件是可重定位文件，所以它的地址并没有被分配，是0x00000000（假如文件中还有其它函数，该函数生成的节区中，对应的地址描述也都是0）。当链接器链接时，根据这个节区头信息，在文件中找到它的主体内容，并根据它的类型，把它加入到主程序中，并分配实际地址，链接后生成的*.axf文件，再来看看它的内容：

MDK的编译过程及文件类型全解

代码清单 42-4 *.axf 文件的节区信息(“流水灯_axf_elfInfo_v.txt”文件)

```
1  ============================================================================
2  ** Section #1
3
4  Name         : ER_IROM1       //节区名
6  //此节区包含程序定义的信息，其格式和含义都由程序来解释。
7  Type         : SHT_PROGBITS (0x00000001)
9  //此节区在进程执行过程中占用内存。 节区包含可执行的机器指令
10 Flags        :
11 SHF_ALLOC + SHF_EXECINSTR (0x00000006)
12 Addr         : 0x08000000     //地址
13 File Offset  : 52 (0x34)
14 Size         : 3136 bytes (0xc40)//大小
15 Link         :
16 SHN_UNDEF
17 Info         : 0
18 Alignment    : 4
19 Entry Size   : 0
20 ==================================
21 ** Section #2
22
23 Name         : RW_IRAM1//节区名
25 //包含将出现在程序的内存映像中的为初始
26 //化数据。 根据定义， 当程序开始执行， 系统
27 //将把这些数据初始化为 0。
28
29 Type         : SHT_PROGBITS (0x00000001)
31 //此节区在进程执行过程中占用内存。 节区包含进程执行过程中将可写的数据。
32 Flags        :
33 SHF_ALLOC + SHF_WRITE (0x00000003)
34 Addr         : 0x20000000      //地址
35 File Offset  : 3188 (0xc74)     //大小
36 Size         : 40 bytes (0x28)
37 Link         :
38 SHN_UNDEF
39 Info         : 0
40 Alignment    : 4
41 Entry Size   : 0
42 ==================================
```

MDK的编译过程及文件类型全解

节区头

在*.axf文件中，主要包含了两个节区，一个名为ER_IROM1，一个名为RW_IRAM1，这些节区头信息中除了具有*.o文件中节区头描述的节区类型、文件位置偏移、大小之外，更重要的是它们都有具体的地址描述，其中ER_IROM1的地址为0x08000000，而RW_IRAM1的地址为0x20000000，它们正好是STM32内部FLASH及SRAM的首地址，对应节区的大小就是程序需要占用FLASH及SRAM空间的实际大小。

也就是说，经过链接器后，它生成的*.axf文件已经汇总了其它*.o文件的所有内容，生成的ER_IROM1节区内容可直接写入到STM32内部FLASH的具体位置。例如，前面*.o文件中的i.LED_GPIO_Config节区已经被加入到*.axf文件的ER_IROM1节区的某地址。

MDK的编译过程及文件类型全解

节区主体及反汇编代码

使用fromelf的-c选项可以查看部分节区的主体信息，对于指令节区，可根据其内容查看相应的反汇编代码，打开“bsp_led_o_elfInfo_c.txt”文件可查看这些信息：

代码清单 42-5 *.o 文件的 LED_GPIO_Config 节区及反汇编代码(bsp_led_o_elfInfo_c.txt 文件)

```
1  ================================================================================
2
3  ** Section #1 'i.LED_GPIO_Config' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
4      Size   : 96 bytes (alignment 4)
5      Address: 0x00000000
6
7      $t
8      i.LED_GPIO_Config
9      LED_GPIO_Config
10   //  地址         内容   [ASCII 码(无意义)]   内容对应的代码
11   0x00000000:    b508          ..       PUSH     {r3,lr}
12   0x00000002:    2101          .!       MOVS     r1,#1
13   0x00000004:    2008          .        MOVS     r0,#8
14   0x00000006:    f7fffffe      ....     BL       RCC_APB2PeriphClockCmd
15   0x0000000a:    2020                   MOVS     r0,#0x20
16   0x0000000c:    f8ad0000      ....     STRH     r0,[sp,#0]
17   0x00000010:    2010          .        MOVS     r0,#0x10
18   0x00000012:    f88d0003      ....     STRB     r0,[sp,#3]
19   0x00000016:    2003          .        MOVS     r0,#3
20   0x00000018:    f88d0002      ....     STRB     r0,[sp,#2]
21   0x0000001c:    4669          iF       MOV      r1,sp
22   0x0000001e:     480f         .H       LDR      r0,[pc,#60] ; [0x5c] = 0x40010c00
23   0x00000020:    f7fffffe      ....     BL       GPIO_Init
24   0x00000024:    2001          .        MOVS     r0,#1
25   /*以下内容省略...*/
```

MDK的编译过程及文件类型全解

节区主体及反汇编代码

可看到，由于这是*.o文件，它的节区地址还是没有分配的，基地址为0x00000000，接着在LED_GPIO_Config标号之后，列出了一个表，表中包含了地址偏移、相应地址中的内容以及根据内容反汇编得到的指令。细看汇编指令，还可看到它包含了跳转到RCC_APB2PeriphClockCmd及GPIO_Init标号的语句，而且这两个跳转语句原来的内容都是“f7fffffe”，这是因为还*.o文件中并没有RCC_APB2PeriphClockCmd及GPIO_Init标号的具体地址索引，在*.axf文件中，这是不一样的。

MDK的编译过程及文件类型全解

节区主体及反汇编代码

接下来我们打开“多彩流水灯_axf_elfInfo_c.txt”文件，查看*.axf文件中，ER_IROM1节区中对应LED_GPIO_Config的内容：

代码清单 42-6*.axf 文件的 LED_GPIO_Config 反汇编代码(流水灯_axf_elfInfo_c.txt 文件)

```
1 LED GPIO Config
 2  0x08000b7c:    b508          ..        PUSH     {r3,lr}
 3  0x08000b7e:    2101          .!        MOVS     r1,#1
 4  0x08000b80:    2008          .         MOVS     r0,#8
 5  0x08000b82:    f7fffefd      ....      BL       RCC_APB2PeriphClockCmd ; 0x8000980
 6  0x08000b86:    2020                    MOVS     r0,#0x20
 7  0x08000b88:    f8ad0000      ....      STRH     r0,[sp,#0]
 8  0x08000b8c:    2010          .         MOVS     r0,#0x10
 9  0x08000b8e:    f88d0003      ....      STRB     r0,[sp,#3]
10  0x08000b92:    2003          .         MOVS     r0,#3
11  0x08000b94:    f88d0002      ....      STRB     r0,[sp,#2]
12  0x08000b98:    4669          iF        MOV      r1,sp
13  0x08000b9a:    480f          .H        LDR      r0,[pc,#60] ; [0x8000bd8] = 0x40010c00
14  0x08000b9c:    f7fffc34      ..4.      BL       GPIO_Init ; 0x8000408
15  0x08000ba0:    2001          .         MOVS     r0,#1
16  0x08000ba2:    f8ad0000      ....      STRH     r0,[sp,#0]
17  0x08000ba6:    4669          iF        MOV      r1,sp
18  0x08000ba8:    480b          .H        LDR      r0,[pc,#44] ; [0x8000bd8] = 0x40010c00
19  /*以下内容省略...*/
```

MDK的编译过程及文件类型全解

节区主体及反汇编代码

可看到，除了基地址以及跳转地址不同之外，LED_GPIO_Config中的内容跟*.o文件中的一样。另外，由于*.o是独立的文件，而*.axf是整个工程汇总的文件，所以在*.axf中包含了所有调用到*.o文件节区的内容。例如，在“bsp_led_o_elfInfo_c.txt”(bsp_led.o文件的反汇编信息)中不包含RCC_APB2PeriphClockCmd及GPIO_Init的内容，而在“流水灯_axf_elfInfo_c.txt” (流水灯.axf文件的反汇编信息)中则可找到它们的具体信息，且它们也有具体的地址空间。

在*.axf文件中，跳转到RCC_APB2PeriphClockCmd及GPIO_Init标号的这两个指令后都有注释，分别是“; 0x8000980”及“; 0x8000408”，它们是这两个标号所在的具体地址，而且这两个跳转语句的跟*.o中的也有区别，内容分别为“f7fffefd”及“f7fffc34”(*.o中的均为f7ffffe)。这就是链接器链接的含义，它把不同*.o中的内容链接起来了。

MDK的编译过程及文件类型全解

分散加载代码

学习至此，还有一个疑问，前面提到程序有存储态及运行态，它们之间应有一个转化过程，把存储在FLASH中的RW-data数据拷贝至SRAM。然而我们的工程中并没有编写这样的代码，在汇编文件中也查不到该过程，芯片是如何知道FLASH的哪些数据应拷贝到SRAM的哪些区域呢？

通过查看“多彩流水灯_axf_elfInfo_c.txt”的反汇编信息，了解到程序中具有一段名为“__scatterload”的分散加载代码，它是由armlink链接器自动生成的。

MDK的编译过程及文件类型全解

分散加载代码

代码清单 42-7 分散加载代码(多彩流水灯_axf_elfInfo_c.txt 文件)

```
1  .text
2    __scatterload
3    __scatterload_rt2
4     0x08000bdc:    4c06        .L      LDR      r4,[pc,#24] ; [0x8000bf8] = 0x8000c20
5     0x08000bde:    4d07        .M      LDR      r5,[pc,#28] ; [0x8000bfc] = 0x8000c40
6     0x08000be0:    e006        ..      B        0x8000bf0 ; __scatterload + 20
7     0x08000be2:    68e0        .h      LDR      r0,[r4,#0xc]
8     0x08000be4:    f0400301    @...    ORR      r3,r0,#1
9     0x08000be8:    e8940007    ....    LDM      r4,{r0-r2}
10    0x08000bec:    4798        .G      BLX      r3
11    0x08000bee:    3410        .4      ADDS     r4,r4,#0x10
12    0x08000bf0:    42ac        .B      CMP      r4,r5
13    0x08000bf2:    d3f6        ..      BCC      0x8000be2 ; __scatterload + 6
14    0x08000bf4:    f7fffaa0    ....    BL       __main_after_scatterload ; 0x8000138
15   $d
16    0x08000bf8:    08000c20     ...    DCD    134220832
17    0x08000bfc:    08000c40    @...    DCD    134220864
18   $t
```

这段分散加载代码包含了拷贝过程(LDM复制指令)，而LDM指令的操作数中包含了加载的源地址，这些地址中包含了内部FLASH存储的RW-data数据。而“__scatterload ”的代码会被“__main”函数调用，__main在启动文件中的“Reset_Handler”会被调用，因而，在主体程序执行前，已经完成了分散加载过程。

MDK的编译过程及文件类型全解

分散加载代码

代码清单 42-8 __main 的反汇编代码（部分，流水灯_axf_elfInfo_c.txt 文件）

```
1  $t
2  .ARM.Collect$$$$00000000
3  .ARM.Collect$$$$00000001
4  __Vectors_End
5  __main
6  _main_stk
7    0x08000130:  f8dfd00c  ....  LDR    sp,__lit__00000000 ; [0x8000140] = 0x20000428
8  .ARM.Collect$$$$00000004
9  _main_scatterload
10   0x08000134:    f000fd52    ..R.    BL       __scatterload ; 0x8000bdc
```

THANKS

论坛： www.firebbs.cn

淘宝：firestm32.taobao.com

扫描进入淘宝店铺