# 程序中的段

李　云
Blog: yunli.blog.51cto.com

## 摘要

了解一个程序的组成结构对于嵌入式系统开发非常的重要，通过对于程序中段的了解也有助于我们进一步的去了解我们的编程语言。

本文采用独特的示例程序帮助读者理解程序中的段，以及变量是如何分配到不同的段的。读者需要注意的是你要学习的不只是文章中的内容，你更应当注意文章中所使用的方法。因为，掌握方法能让你学会如何去寻求一些问题的答案。

## 关键词

程序段　　.text　　.data　　.rdata　　.bss

## 参考资料

《什么是 boot loader》
《堆和栈》
《熟悉 binutils 工具集》

如果你读过了《熟悉 binutils 工具集》一文，那么你应当已经知道了如何使用 objdump 工具来查看一个程序文件中的段信息。在所有的程序中都存在三个段，它们是.text、.data（或.rdata）和.bss 段。下面我们就来说一说每个段的作用是什么，对于 C 程序中的元素（函数或变量），看看它们分别是放在哪一个段中的。

先来说一说.text 段，这个段可能是最容易让我们明白是放什么的，其中存放的是处理器的机器指令。不论我们是用 C 语言还是用 C++语言编写程序，其最后都得转换成处理器的机器指令才能运行。我们用高级语言所写的程序，经过编译器编译或是汇编器汇编以后，就会产生机器指令和数据。数据存放在哪，我们后面会说到，这里只关心机器指令。对于我们所设计的应用程序，往往需要多个源程序文件，每一个文件都会先被单独编译成一个目标文件，在每一个目标文件中通常都会有.text 段。将多个目标文件经连接器连接以后，除了解决各源文件之间的函数数引用外，还得将所有目标文件中的.text 段合在一起，最终生成可执行文件中的一个.text 段。从处理器的角度来看，每一个函数其实就是一些指令的组合，而函数的地址就是这上指令组合在内存中的存储起始地址，函数调用也就是跳转到相应的函数地址处去执行机器指令。因此，我们也不难想像连接器在将多个.text 段合成一个时，不是简单的将它们“堆”在一起就完事，还需要处理各个段之间的函数引用问题。在嵌入式系统中，如果处理器是带有 MMU（Memory Management Unit，内存管理单元），那么当我们的可执行程序被加载到内存以后，通常都会将.text 段所在的内存空间设置为只读，以保护.text 中的代码不会被意外的被改写（比如在程序出错时）。当然，如果没有 MMU 就无法获得这种代码保护功能。

指令是存放在.text 段的，下面我们来说说数据存放在哪，毫无疑问是存放在.data 和.bss 段的。那为什么要将数据分成两个段呢？请先看图 1 所示的 section1.c 源代码。

**section1.c**

```
int main ()
{
```

```
    return 0;
}
```

图 1

将 section1.c 编译成可执行文件，然后用 objdump 工具来量看一下它的段信息，操作命令及结果如图 2 所示。

```
yunli.blog.51cto.com ~
$gcc section1.c -o section1.exe
yunli.blog.51cto.com ~
$strip section1.exe
yunli.blog.51cto.com ~
$objdump -h section1.exe

section1.exe:     file format pei-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000428  00401000  00401000  00000400  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE, DATA
  1 .data         0000000c  00402000  00402000  00000a00  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .rdata        00000044  00403000  00403000  00000c00  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .bss          00000060  00404000  00404000  00000000  2**3
                  ALLOC
  4 .idata        00000190  00405000  00405000  00000e00  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

图 2

在输出的 objdump 信息中，请注意其中我所标记出来的三个段的大小。现在，在 section1.c 中定义两个全局变量，改变后的代码如图 3 所示。

**section2.c**

```
int g_initialized = 0x5A5A5A5A;
int g_uninitialized;

int main ()
{
    return 0;
}
```

图 3

其中的 g_initialized 是被初始化了的全局变量，而 g_uninitialized 是没有被初始化的。下面是将 section2.c 编译后所看到的 objdump 信息。

```
yunli.blog.51cto.com ~
$gcc section2.c -o section2.exe
yunli.blog.51cto.com ~
$strip section2.exe
yunli.blog.51cto.com ~
$objdump -h section2.exe
```

```
section2.exe:     file format pei-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000428  00401000  00401000  00000400  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE, DATA
  1 .data         00000010  00402000  00402000  00000a00  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .rdata        00000044  00403000  00403000  00000c00  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .bss          00000070  00404000  00404000  00000000  2**3
                  ALLOC
  4 .idata        00000190  00405000  00405000  00000e00  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

图 4

与图 2 信息所不同的是，.data 段和.bss 段的大小有了改变。这是因为在 section2.c 中增加了两个变量。对于初始化好的变量，其被存放在.data 段中，而对于没有初始化好的段其被存放在.bss 段中。由于我们定义的 g_initialized 类型是 int，这一类型在一个 32 位的处理器上也是 32 位，加上.data 段的字节对齐数也是 4 个字节（从 objdump 的输出信息中可以看出每一个段的字节对齐数），所以造成.data 段刚好增加 4 个字节。通过 objdump 我们可以看到.data 段中的内容，如图 5 所示。

```
yunli.blog.51cto.com ~
$objdump -s -j .data section2.exe

section2.exe:     file format pei-i386

Contents of section .data:
 402000 00000000 00000000 00000000 5a5a5a5a    ............ZZZZ
```

图 5

从图 5 的输出信息来看，的确是在.data 段中增加了四个字节，而且其内容就是我们在 C 程序中所定义的 0x5A5A5A5A。对于上面的输出信息，你需要注意处理器的大端模式（big endian）和小端模式（little endian），有可能你所看到的信息与你在程序中定义的在字节序上是相反的。

接下来我们看一看.bss 段的变化，从图 4 的 objdump 所输出信息来看，当我们定义了一个没有初始化的全局变量后，其在.bss 段增长了 16 个字节，但是.bss 段的字节对齐数却是 8 字节，对于为什么在.bss 段上分配 16 个字节，我现在也不能告诉你为什么，在我有机会研究后再告诉你为什么。不管如何，我们知道了没有初始化的全局变量是放在.bss 段的。由于.bss 段中存放的数据是没有初始化好的，所以从理论上来说，不需要将其内容象.data 段那样存放在我们的程序文件中，我们可以查看一下程序文件中.bss 段中的内容，其结果如图 6 所示。

```
yunli.blog.51cto.com ~
$objdump -s -j .data section2.exe

section2.exe:     file format pei-i386
```

图 6

从显示结果来看程序文件中.bss 段的内容的确是空的，那是不是意味着所有存放在.bss 段中的

未初始化的全局变量在程序运行时其值都是随机的呢？不是！当程序被 boot loader 加载以后，boot loader 再将执行权交给被加载程序之前，它会将.bss 段内存区全部初始化为 0。这就是为什么没有初始化值的非指针全局变量其初始值总是为 0，而对于指针其初始值则是 NULL。

现在我相信你明白了.data 段与.bss 段的区别了，对于初始化好的全局变量，编译器将其地址分配在.data 段中，当程序被 boot loader 加载时，则只需将位于程序文件中的.data 数据复制到内存所对应的地址空间，从而一次性的完成所有需要始化的全局变量的初始化，这样操作是不是很简单啊？对于.bss 段，由于其中的变量是没有初始化好的，所以不需要在程序文件中保存其内容，这样的好处是能减小程序文件的大小。而当 boot loader 加载程序时，会自动对被加载程序的.bss 段进行清零。

此外，通过我们在《熟悉 binutils 工具集》中介绍的 nm 工具，我们可以验证两个变量所分配的段信息。验证结果如图 7 所示，其中的字母 D 表示是.data 段，而 B 则表示的是.bss 段。

```
yunli.blog.51cto.com ~
$gcc section2.c -o section2.exe
yunli.blog.51cto.com ~
$nm -n section2.exe
…显示结果有删减…
0040200c D _g_initialized
00404040 B _g_uninitialized
```

图 7

section2.c 中我们定义的是两个非静态的全局变量，将其定义成静态的全局变量时，结果仍然一样吗？请看 section3.c 编译后采用 objdump 所显示的结果，源程序和结果分别在图 8 和图 9 中列出。

**section3.c**

```
static int g_initialized = 0x5A5A5A5A;
static int g_uninitialized;

int main ()
{
    return 0;
}
```

图 8

```
yunli.blog.51cto.com ~
$gcc section3.c -o section3.exe
yunli.blog.51cto.com ~
$strip section3.exe
yunli.blog.51cto.com ~
$objdump -h section3.exe

section3.exe:     file format pei-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000428  00401000  00401000  00000400  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE, DATA
  1 .data         00000010  00402000  00402000  00000a00  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .rdata        00000044  00403000  00403000  00000c00  2**2
```

```
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .bss          00000070  00404000  00404000  00000000  2**3
                  ALLOC
  4 .idata        00000190  00405000  00405000  00000e00  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

图 9

从结果来看，全局变量的分配空间与加不加 static 是没有关系的。这其实也是好理解解的，因为.data 与.bss 是从数据是否需要初始化进行区分的，而 static 与数据是否初始化是正交的，static 只是表示变量只能被本文件中的函数访问。

那对于函数内部的 static 变量也是这样吗？我们知道，当一个函数内定义非静态的变量时，其是被分配在栈上的，那加了 static 后，还是分配在栈上吗？我们需要通过一定的实验来看个究竟。图 10 是我们需要用到的 section4.c 源程序，而 objdump 的输出结果在图 11 中给出。

**section4.c**

```
int main ()
{
      static int g_initialized = 0x5A5A5A5A;
      static int g_uninitialized;
      return 0;
}
```

图 10

```
yunli.blog.51cto.com ~
$gcc section4.c -o section4.exe
yunli.blog.51cto.com ~
$strip section4.exe
yunli.blog.51cto.com ~
$objdump -h section4.exe

section4.exe:     file format pei-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000428  00401000  00401000  00000400  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE, DATA
  1 .data         00000010  00402000  00402000  00000a00  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .rdata        00000044  00403000  00403000  00000c00  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .bss          00000070  00404000  00404000  00000000  2**3
                  ALLOC
  4 .idata        00000190  00405000  00405000  00000e00  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

图 11

与图 4 相类比会发现，对于静态的局部变量，编译器对其段的分配与全局变量是一样的。这也是好理解的，因为静态的局部变量我们需要一直保存其值，当下次进入函数体时，其值仍然维持在最后一次改变，而如果是放在栈上，这是做不到的，因为函数所占用的栈空间当函数返回后就不再

属于这个函数了。

上面我们讲了 int 类型的全局变量，下面我们再来看一看字符串全局变量，为此我们需要一个新的测试程序 —— section5.c，其源程序如图 12 所示。

**section5.c**

```
char g_char [] = "Hello World!";

int main ()
{
	return 0;
}
```

图 12

图 13 是 section5.c 的编译及段显示结果。从图中可以看出，毫无意外的 g_char 变量也是被分配在.data 段内，但其分配的长度却是 16 个字节，而不是实际的 13 个字节，原因是显然的 —— 为了做到 4 字节对齐。

```
yunli.blog.51cto.com ~
$gcc section5.c -o section5.exe
yunli.blog.51cto.com ~
$strip section5.exe
yunli.blog.51cto.com ~
$objdump -h section5.exe

section5.exe:     file format pei-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000428  00401000  00401000  00000400  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE, DATA
  1 .data         0000001c  00402000  00402000  00000a00  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .rdata        00000044  00403000  00403000  00000c00  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .bss          00000060  00404000  00404000  00000000  2**3
                  ALLOC
  4 .idata        00000190  00405000  00405000  00000e00  2**2
                  CONTENTS, ALLOC, LOAD, DATA

yunli.blog.51cto.com ~
$objdump -s -j .data section5.exe

section5.exe:     file format pei-i386

Contents of section .data:
 402000 00000000 00000000 00000000 48656c6c  ............Hell
 402010 6f20576f 726c6421 00000000           o World!....
```

图 13

接下来，我们对 section5.c 做一点小小的改动，那就是在 g_char 变量前加 const 关键字，改变后的源程序如图 14 所示。

**section6.c**

```
const char g_char [] = "Hello World!";

int main ()
{
	return 0;
}
```

图 14

图 15 是 section6.c 的编译及段显示结果。这次与 section5.exe 的结果有所不同，增加的段是.rdata 而不是.data。.rdata 是用来存放只读实始化变量的，当我们在源程序中的 g_char 变量前面加了 const 后，编译器知道个字符串是永远不会改变的，或说是只读的，所以将其分配到.rdata 段中。在图 15 的后面显示了.rdata 段中的内容，从中你确实可以看到 g_char 变量是分配在其中的。与.data 段所不同的是，对于有 MMU 的嵌入式系统，.rdata 段也会采用.text 段相同的保护方式，即将.rdata 段设置成只读，以防止其被意外改写。通常，.rdata 会与.text 放在一个连续的内存空间中。

```
yunli.blog.51cto.com ~
$gcc section6.c -o section6.exe
yunli.blog.51cto.com ~
$strip section6.exe
yunli.blog.51cto.com ~
$objdump -h section6.exe

section6.exe:     file format pei-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000428  00401000  00401000  00000400  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE, DATA
  1 .data         0000000c  00402000  00402000  00000a00  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .rdata        00000054  00403000  00403000  00000c00  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .bss          00000060  00404000  00404000  00000000  2**3
                  ALLOC
  4 .idata        00000190  00405000  00405000  00000e00  2**2
                  CONTENTS, ALLOC, LOAD, DATA

yunli.blog.51cto.com ~
$objdump -s -j .rdata section6.exe

section6.exe:     file format pei-i386

Contents of section .rdata:
 403000 63796767 63635f73 2d312e64 6c6c005f  cyggcc_s-1.dll._
 403010 5f726567 69737465 725f6672 616d655f  _register_frame_
 403020 696e666f 005f5f64 65726567 69737465  info.__deregiste
```

```
403030 725f6672 616d655f 696e666f 00000000   r_frame_info....
403040 48656c6c 6f20576f 726c6421 00000000   Hello World!....
403050 00000000                              ....
```

图 15

至此，我相你应当非常明白程序中的变量是如果分配到不同的段的，现在是总结一下的时候了。采用将不同的变量分配到连续的一个段中，对于程序的初始化非常的方便。对于初始化好的程序变量，当程序被加载时，boot loader 只要将程序文件中的.data 或.rdata 段拷贝到内存中，从而一次性的完成对所有需初始化变量的初始化操作。对于没有初始化的变量，其被存放在.bss 段中，但在我们的程序文件中并没有这一个段的具体内容，这是为了节省程序文件的存储空间。当程序被 boot loader 加载时，boot loader 会负责将被加载程序的.bss 段全部清零，因此，对于没有初始化的指针变量其初始值是 NULL，而对于非指针变量则是 0。全局变量或是函数中的静态变量，其在程序编译完成时就决定了内存空间的分配，其所占用的内存只有当程序退出时才释放，意识到这一点非常的重要。

## 致读者

如果你觉得本文的哪些地方需要改进或是存在一些不明白的地方，请点击这里并留言。如果你想参与讨论嵌入式系统开发相关的话题，请加入技术圈（g.51cto.com/UltraEmbedded）。

## 修订历史

| 日 期 | 修 订 说 明 |
|---|---|
| 2009-08-04 | 新文档 |