

基于 Hayden Luo Bios 的 BootLoader 源代码级分析第一部分——

第一篇：GNU tools 开发 ARM 程序及生成映象文件机理

ARM 学习报告 002 2004-5-12

在我的 ARM 学习报告 001 中，用了一个简单的 MySComm4510b 程序，对 ARM 系统的映象文件的生成和执行过程做了一个很初浅的介绍，希望可以给初学者一个入门的启示。分析用的是 ADS1.2 和 ADW，基本上都是 ARM 公司的编译工具，设置的选项不是很多，这也是 Windwos 上开发工具所共有的优点。可是，最近已经十分流行在 ARM 上开发基于 linux 的系统，uClinux 已是如火如荼。用嵌入式 linux，当然首先要进攻 Bootloader 这个高级一些的 linux 下的 ARM 程序。也许很多人觉得能用起来就可以了，不过作为学习来说，个人认为还是深入一些比较好！知其然，知其所以然，才利于将来的其他程序或系统的开发。这个程序看起来不小，不入虎穴，焉得虎子！

在这一篇我只介绍在 linux 下用 GNU 工具集开发 ARM 程序的一个机理！基本不牵涉到源程序内部的内容。因为我觉得懂得一个概念比读懂一段程序要重要的多，尤其是在嵌入式系统开发中。

我写的东西完全按照我的思路和学习过程来叙述的，和一般的书籍不一样，所以也许你会不习惯这种思维，不过，相信这样的叙述可以给你带来意想不到的轻松和结果！让我们开始学习吧！

小插曲：

我接触 linux 也是挺长时间了，大概 99 年就开始接触，不过早些时候，学东西总是浮于表面，linux 的安装和操作很熟，配置和使用其中的软件也很熟，但是编程——就学了一些在 linux 下编些最简单的 C++ 程序，简直都不能说用过 gcc、makefile 和 gdb 了，因为就是敲个命令加常用选项，大概都只是知道有这么个东西了，其内部机制完全不懂，当然在 PC 下编简单程序，也不用懂什么映象文件、存储分配阿什么的，很多东西操作系统都替你考虑好了。

这几天准备学习那个“在 linux 下开发的 bootloader”，才真正开始深入到其中。这里要感谢 twentyone 推荐的 VMware4.0，为我们这些 linux 入门者真是提供了很大的方便，方便在 windows 下使用比较真实的 linux（本来我还想用 cygwin，可是发现这个软件问题太多），我安装的是 Redhat9.0。

正是由于这种基础，所以我第一眼看到 head.S 文件，就有了很多疑惑，相信很多人都有（做个调查？），不过都“一笑而过”，反正看得懂就好了！这个鬼文件，怎么是 C 和汇编的结合，但好像又不符合 ARM 规范嘛，怪怪的，刚刚开始我还没有意识到这是在 linux 下编译的，还一直认为是 ARM 公司的汇编格式的扩展，找了好久相关的文档定义，现在想想真是一个大傻。

后来才从 makefile 中才“领悟”到原来压根就是用 arm-elf-gcc 编译这个文件，终于认识到自己在 linux 面前实在太无知了，不过，无知没有关系，因为我们还可以学嘛！！

一 HEAD.S 和 Makefile（根目录下）

1.1 arm-elf-gcc

我想我还是接着那个小插曲说吧，就从 HEAD.S 和 Makefile 这两个文件开始 bootloader

分析，我个人认为，这两个文件再加上 BIOS.LD，就足够让我们这些很多初学者学好久了，因为 linux 和 gnu 实在是太博大精深了！

看看我是如何发现用 arm-elf-gcc 编译 HEAD.S 的☺：（以下是 Mikefile 的一部分）

```
CROSS = arm-elf-
```

```
CC = $(CROSS)gcc
AS = $(CROSS)as
AR = $(CROSS)ar
LD = $(CROSS)ld
NM = $(CROSS)nm
RM = rm
OBJCOPY = $(CROSS)objcopy
BIN2C = ./tools/bin2c
ZIP = gzip
```

```
CFLAGS = -fno-builtin -nostdlib -Wall -O2 -fomit-frame-pointer -I.
```

```
AFLAGS = -mapcs-32 -msoft-float -mno-fpu -I.
```

```
LDFLAGS= $(CFLAGS) -Wl,-elf2flt
```

```
.....
```

```
.....
```

```
%.o: %.c
```

```
$(CC) $(CFLAGS) -c -o $@ $<
```

```
%.o: %.S
```

```
$(CC) $(AFLAGS) -c -o $@ $<
```

如果 linux 初学者还看不懂上面这些内容，也许应该先花些时间看看 Makefile，很多有关 linux 编程的书都有介绍。

从最后一个规则可以看出，在编译.S 文件时用了以下命令：

```
arm-elf-gcc -mapcs-32 -msoft-float -mno-fpu -I. -c -o HEAD.o HEAD.S
```

以上所有的编译选项都可以在“Using and Porting the GNU Compiler Collection”文档中找到相应的解释：-mapcs-32：生成可以运行在具有32位程序计数器的处理器上的目标文件，并且遵循APCS-32函数调用标准。-msoft-float -mno-fpu：输出的文件包含对浮点运算库函数的调用。

arm-elf-gcc 是个交叉编译器，在很多地方都可以下载这个编译器的安装包，我们要编译 uClinux 时，所需要的工具中就包括了 arm-elf-gcc。在 linux 中用 man 打开 arm-elf-gcc 的帮助文件，鬼，打开的就是 gcc 帮助文件。其实 arm-elf-gcc 就是 gcc 用于 ARM 体系的移植版本，很多使用选项和 gcc 其实相同，况且我们大多用的都是一些标准的选项，所以也不必为它那么多选项吓到！

这时又开始纳闷了，怎么 gcc 也可以编译汇编语言呢，不是 as 才可以的吗？（初学者的困惑）

在 GNU C compile 帮助文件中我们可以看到：

Compilation can involve up to four stages: preprocessing, compilation proper, assembly and linking, always in that order. For any given input file, the file name suffix determines what kind of

compilation is done: (一般编译连接处理过程包括四个阶段: 预处理, 编译, 汇编编译, 连接。后缀名将决定该文件被采取何种编译处理)

- .c C source; preprocess,
- .C C++ source; preprocess, compile, assemble
- .cc C++ source; preprocess, compile, assemble
- .cxx C++ source; preprocess, compile, assemble
- .m Objective-C source; preprocess, compile, assemble
- .i preprocessed C; compile, assemble
- .ii preprocessed C++; compile, assemble
- .s Assembler source; assemble
- **.S Assembler source; preprocess, assemble (S 是大写)**
- .h Preprocessor file; not usually named on command line
- other An object file to be fed straight into linking. Any file name with no recognized suffix is treated this way.

其中:

file.s

Assembler code.

file.S

Assembler code which must be preprocessed.

原来 arm-elf-gcc 会自动根据文件的后缀名和编译选项, 调用相应的工具如 as,ld 等完成编译连接这 4 个过程。head.S 虽然是汇编代码, 却又必须被预处理 (preprocess), 和 head.s 是不一样的。这正是为什么在 head.S 中仍然可以存在如/*Copyright*/注释 # include 等等这些预处理操作的原因。它要先被 arm-elf-gcc 预处理, 然后 arm-elf-gcc 才调用 arm-elf-as 来汇编编译成目标文件 head.o。

这时“小插曲”里的疑惑应该解决了一些了, 最起码不会感觉这个文件怪怪的。不过, 还是有些标号和定义不知“出自哪里”, **看起来好像都懂**, 比如.globl、.section、stext:、beq 1f 等等, 但心里又没有底, 那就继续往下看吧!

1.2 The gnu Assembler

现在我们所要解决的疑惑对后面映象文件的分析有很大的帮助, 我们要的是官方的正式文档, 不是靠个人猜测, 否则以后自己就很难利用 GNU 工具来开发系统。

刚刚我们已经知道.S文件是经过预操作和汇编两步而生成.o文件, 预操作应该不会处理.globl等这些定义, 那么应该就是和汇编器有关了, 去看看“ The gnu Assembler ”文档, 果然, .globl等就是其中的定义和符号。

GAS 实在是太博大精深了, 当我一直看这个文档时, 觉得有一种很过瘾的感觉。这里我就把 HEAD.S 中遇到的定义和符号说明一下:

.globl symbol : 这个定义使被定义的符号可以被 ld (连接工具) 引用, 假如你是在一个源程序中定义, 那么**同这个源程序连接在一起的其他源程序文件**都可以引用这个符号 (翻译的不好, 大家最好还是看英文原版)。

.long expressions : 和.int 一样, 就是在运行时给后面表达式赋值。字节数或字节顺序和编译的对象平台有关。

.align exp : 使“位置计数器”作相应的对齐操作。

Label: : 当一个符号后面紧跟冒号, 这个符号 (symbol) 就是一个标号 (label), 其值

为“活动位置计数器”的当前值（翻译语）。其实就是我们在高级语言中用过的标号！！也是很好理解。

0、1、2：：这些是局部符号名，用于编译器或程序员临时使用。一共有10个局部符号名，可以重复使用，其名字为0, 1, 2, ... 9，使用的时候，按照标号的格式“N:”(N为0~9的数字)，“Nb”表示“离得最近的前一个标号”，“Nf”表示“离得最近的后一个标号”，我们可以从HEAD.S中理解这个局部符号。

以上我们基本上对HEAD.S中涉及到非ARM正规汇编格式的其他操作或符号作了相应介绍，其实我觉得最好还是看上面提到的英文原版文档（用GOOGLE搜）。

其实我花这么大的笔墨来写前面的东西，主要想描述我自己在学习的过程中怎么发现并引出一个很重要的东西，也是这篇文章的主角。细心的朋友可能发现我在上面漏介绍了一个定义，那就是.section！！

二 .section

有些ARM映象文件基础的人大都可以从HEAD.S看出.section的大概意思，顾名思义，.section就是段的意思，类似于ARM映象文件中的“输出段”。在ARM映象文件中，一个域包括至多三个输出段RO, RW, ZI, 那么用GNU生成的映象文件呢？是不是也是这样呢？怎么去控制相应的ro_base rw_base entry呢？怎么最后生成整个可执行的映象文件呢？

如果对ARM很基本的映象文件格式还是很模糊的话，最好先看看我的ARM学习报告001(<http://bbs.edw.com.cn/disppbbs.asp?boardID=20&ID=28310&page=1>)，报告001中讲的ARM映象文件的轮廓还是比较清晰简单的，只有一个源文件，而且是汇编格式的，起码比我下面要讲的GNU映象文件结构要清晰一些。

我依然延续我在报告001中的采用的描述顺序，先来讲讲用GNU工具集生成的可在ARM上运行的映象文件的“概貌”，一个感性认识很重要！

在这里我举的例子是我们常用的bios (bootloader)，对于最后生成的bios可执行文件来说，实际上是由各个object文件连接而成，可以在Makefile(根目录)中看出。那么各个object文件怎么生成？又怎么最后串在一起形成完整的可执行文件，这里估计说也说不清楚，还是先看看简单的吧！不过，我们还是以这个bootloader为总体分析的对象！毕竟就是准备分析它！

所以我从bios中最吸引人的HEAD.S入手！它给我的印象太深了☺，给我的触动也最深！但HEAD.S又不能单独成为一个ARM可执行映象文件，编译后只能生成一个组成bios的HEAD.o，不过没事，从HEAD.o就已经可以看出很多东西了！因为它就是组成最后可执行文件bios的头号功臣！

2.1 HEAD.o的格式——ELF文件格式

在讲GNU tools如何生成可在ARM上运行的映象文件前，我觉得先认识一下elf文件格式是很有必要的。有关section的内容很多都可以从这里感性认识到，看到！！GNU tools在编译连接时，生成的目标文件.o和可执行文件的格式都是elf格式的（早期也有a.out格式，现在已经没有不用了）。

ELF：可执行连接格式 (Executable and Linking Format)。可执行连接格式是 UNIX 系统实验室(USL)作为应用程序二进制接口(Application Binary Interface(ABI)而开发和发布的。工具接口标准委员会(TIS)选择了正在发展中的 ELF 标准作为工作在 32 位 INTEL 体系上不同操作系统之间可移植的二进制文件格式。（arm 的映象文件 axf 很像是 elf 的简化版，就是说

arm 可能是学 GNU 的噢☺)

2.1.1 elf文件格式部分简介

首先我们应该明白 elf 格式是 gnu 编译结果——object 文件的默认格式，Object 文件格式有三种类型：

1. 可重定位(relocatable)文件：一个可重定位(relocatable)文件保存着代码和适当的数据，用来和其他的 object 文件一起来创建一个可执行文件或者是一个共享文件。**这个很重要！在 bios 中生成的很多.o 文件都是可重定位的！**
2. 可执行(executable)文件：一个可执行(executable)文件保存着一个用来执行的程序；该文件指出了 exec(BA_OS)如何来创建程序进程映象。是的，可执行文件也是 elf 格式的，所以我们才非了解不可！
3. 共享 object 文件：一个共享 object 文件保存着代码和合适的的数据，用来被下面的两个链接器链接。第一个是连接编辑器[请参看 ld(SD_CMD)]，可以和其他的可重定位和共享 object 文件来创建其他的 object。第二个是动态链接器，联合一个可执行文件和其他的共享 object 文件来创建一个进程映象。(暂时可以不管)

那么我们就来看看那 elf 文件格式：由于 elf 文件既可以是可重定位的 object 文件，又可以是可执行的，所以我们可以从不同角度来观察 elf 文件的格式。如下图。

图 1: Object 文件格式

Linking 视角	Execution 视角
=====	=====
ELF header	ELF header
Program header table (optional)	Program header table
Section 1	Segment 1
...	Segment 2
Section n	...
Section header table	Section header table (optional)

每一个 elf 文件都是以一个 ELF header 的结构开始的。该结构为 52 个字节长，并且包含了一个信息部分，这些信息部分描述了文件的内容。例如，前 16 个字节包含了一个“标识符”，它包含了 ELF 文件的魔术数，但字节的标记表明是 32 位的还是 64 位的，小端序还是大端序，等等。在 elf header 包含的其他的信息还有，例如：目标体系；ELF 文件是否是可执行的还是 OBJECT 文件还是一个共享的库；程序的开始地址；

program header table 和 **section header table** 在文件的偏移量。两个表可以出现在文件的任何地方，但是以前经常是直接跟在 ELF HEADER 后面，后来出现在文件的末尾或许是靠近末尾。两个表有相似的功能，都是为了鉴别文件的组成。但是，**section header table 更关注的是识别在程序中不同部分在什么地方**，然而，program header table 描述的是哪里和如何把那些部分转载到内存中。简单的说，section header table 是被编译器(compiler)和连接器(linker)使用，program header table 是被程序转载器(loader)使用。对 object 文件，program header table 是可选的，实际上从来也没有出现过。同样的，对于可执行文件来说，section header table 也是可选的，但是它却总是存在于可执行文件中。

那我们最关心的是什么呢？是 **section header table**，因为我们是要分析 section 的。不

过稍稍分析一下 elf 文件头部吧！做事情总要有头有尾吧！

下面的例子和图都是用 ultraedit 打开的 HEAD.o 文件！

2.1.2 ELF header

该header一定在elf文件的头部，大小为52字节，即0x00~0x33

ELF Header 数据结构如下：

```
#define EI_NIDENT 16
```

```
typedef struct {
    unsigned char e_ident[EI_NIDENT];
    Elf32_Half e_type;
    Elf32_Half e_machine;
    Elf32_Word e_version;
    Elf32_Addr e_entry;
    Elf32_Off e_phoff;
    Elf32_Off e_shoff;          /*section header table 的偏移地址*/
    Elf32_Word e_flags;
    Elf32_Half e_ehsize;
    Elf32_Half e_phentsize;
    Elf32_Half e_phnum;
    Elf32_Half e_shentsize;    /*每个 section header 的字节数*/
    Elf32_Half e_shnum;       /*section header table 中 section header 的数目*/
    Elf32_Half e_shstrndx;
} Elf32_Ehdr;
```

关于这个数据结构的具体内容我就不介绍了，太烦了，大家感兴趣的话可以参考文档

<http://www.muppetlabs.com/~breadbox/software/ELF.txt>。

具体例子见图 2（打开的是 bios 中的 head.o）：

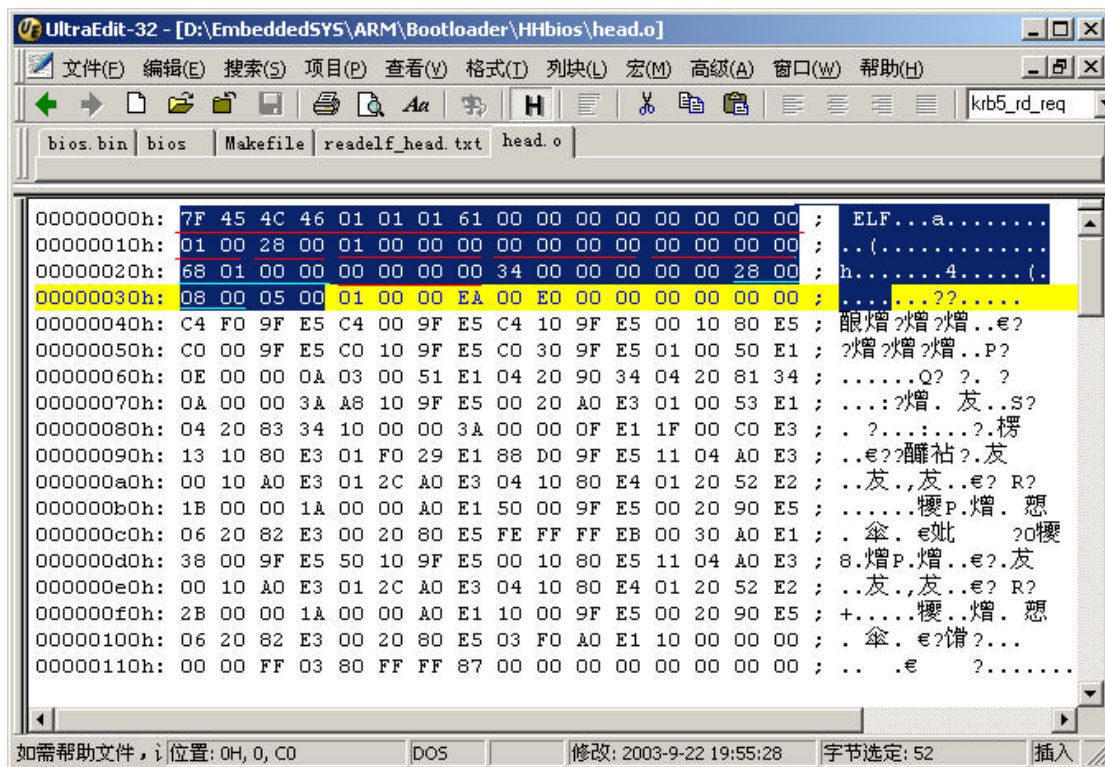
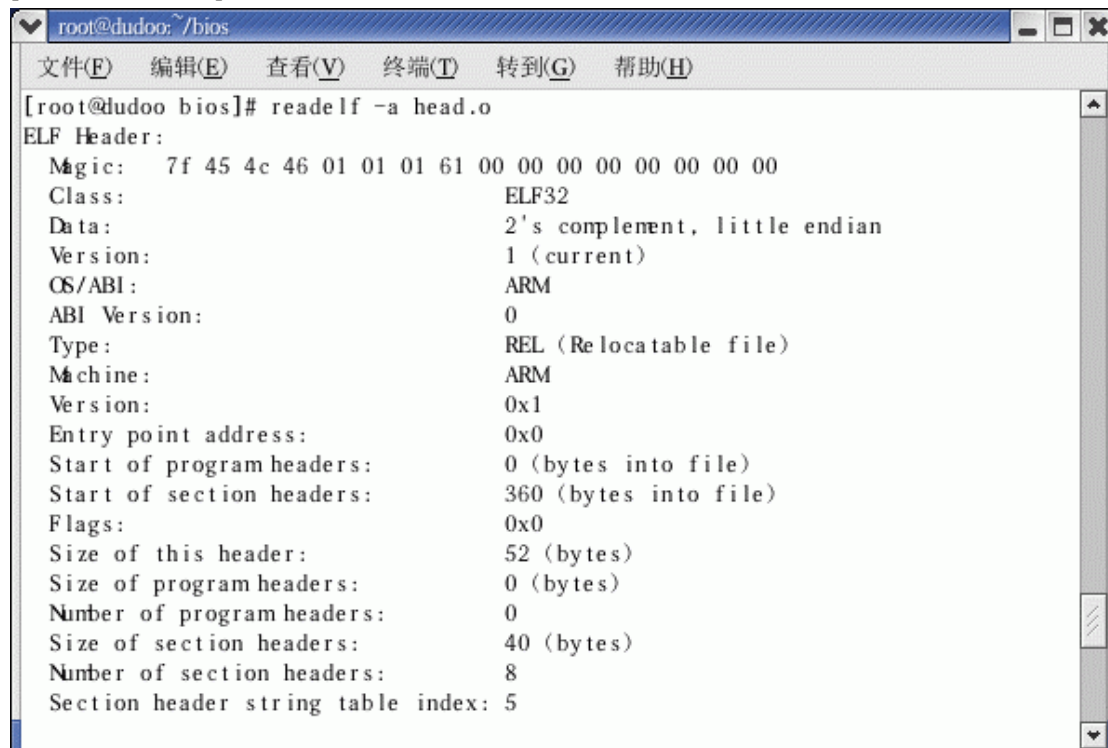


图 2 ultraedit 打开的 HEAD.o

该 ELF_header 包括了相关信息，在 linux 下我们用 readelf 命令可以读出相关信息，如下：

```
[root@dudoo bios]# readelf -a head.o
```



```

root@dudoo:~/bios
文件(F) 编辑(E) 查看(V) 终端(T) 转到(G) 帮助(H)
[root@dudoo bios]# readelf -a head.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00 00
  Class:                   ELF32
  Data:                     2's complement, little endian
  Version:                  1 (current)
  OS/ABI:                   ARM
  ABI Version:              0
  Type:                     REL (Relocatable file)
  Machine:                  ARM
  Version:                  0x1
  Entry point address:      0x0
  Start of program headers: 0 (bytes into file)
  Start of section headers: 360 (bytes into file)
  Flags:                    0x0
  Size of this header:      52 (bytes)
  Size of program headers:  0 (bytes)
  Number of program headers: 0
  Size of section headers:  40 (bytes)
  Number of section headers: 8
  Section header string table index: 5

```

图 3 在 linux 下用 readelf 读出的 head.o 的头部分

我们可以用 ultraedit 打开的二进制图和 readelf 读出的内容一一对应起来看，可能更直观一些，由于我们比较关心的是 section header 相关的内容，所以用蓝色画了出来：

```

Start of section headers:      68 01 00 00      ( 360 )
Size of section headers:      28 00          ( 40 )
Number of section headers :    08 00          ( 8 )

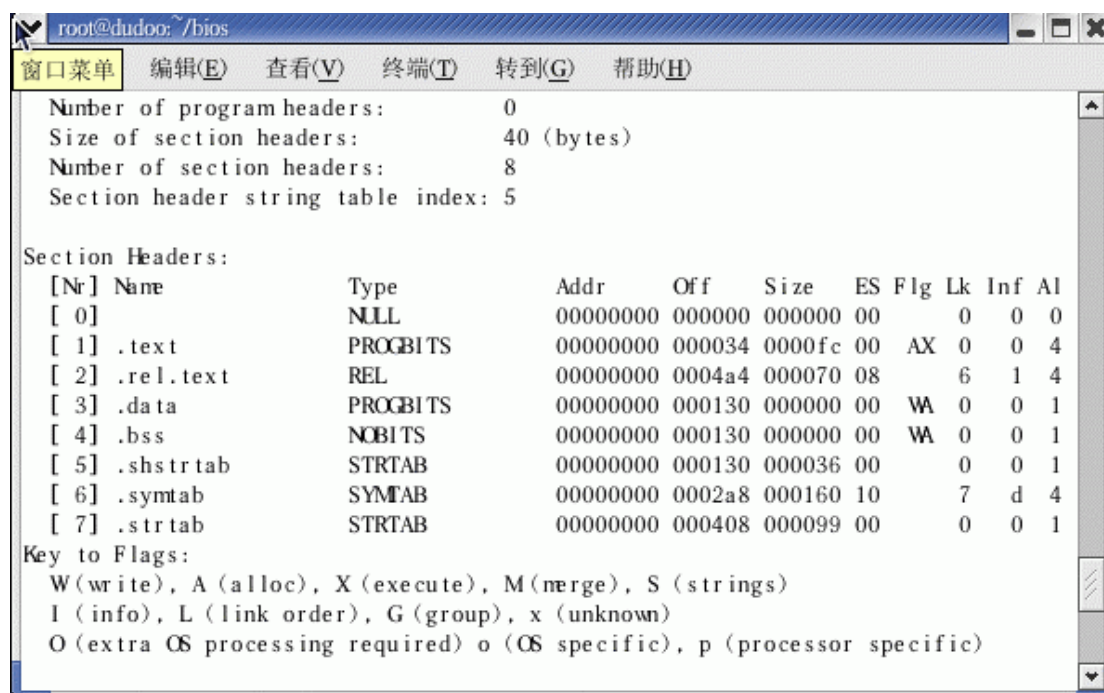
```

这些值应该“反过来”看哦！而且是十六进制的。可以看出，和 readelf 是一样的值。

介绍了这么多头部分，主要想引出 section header table。从 Start of section headers 也就是 e_shoff 可以得到 section header table 的地址。

2.2.3 Section header table

了解完 elf header 后，我们就应该来了解一下 section header table，因为它太重要了。Section header table，顾名思义，是这个 Object 文件中所包含的 Section 的一张索引表，这么说可能还不是很形象，那么我们再来看看 readelf 读出的其他信息：



```

root@dudoc: ~/bios
窗口菜单 编辑(E) 查看(V) 终端(T) 转到(G) 帮助(H)
Number of program headers:      0
Size of section headers:       40 (bytes)
Number of section headers:     8
Section header string table index: 5

Section Headers:
[Nr] Name           Type           Addr          Off           Size          ES Flg Lk  Inf Al
[ 0] NULL           NULL          00000000     000000     000000     00   AX 0   0  0
[ 1] .text          PROGBITS     00000000     000034     0000fc     00   AX 0   0  4
[ 2] .rel.text      REL          00000000     0004a4     000070     08    6 1   4
[ 3] .data          PROGBITS     00000000     000130     000000     00   WA 0   0  1
[ 4] .bss           NOBITS       00000000     000130     000000     00   WA 0   0  1
[ 5] .shstrtab      STRTAB       00000000     000130     000036     00    0 0   1
[ 6] .symtab        SYMTAB       00000000     0002a8     000160     10    7 d   4
[ 7] .strtab        STRTAB       00000000     000408     000099     00    0 0   1

Key to Flags:
W(write), A(alloc), X(execute), M(merge), S(strings)
I(info), L(link order), G(group), x(unknown)
O(extra OS processing required) o(OS specific), p(processor specific)

```

图 4 在 linux 下用 readelf 读出的 head.o 的 section header table 部分

我们在 elf header 中已经知道了 **Number of section headers = 8**，从上图也可以看出有 8 个 section，正是这些 section 包含了这个 object 文件 (head.o) 中的代码和数据等大部分重要信息，这些组织和 ARM 映象文件也是有相似之处的。其中的 .text、.data、.bss 也许你在 bios 的某些地方见过噢！

由图 4 可知，HEAD.o 由 8 个 section 组成。这些 section 的信息，例如名字，类型，地址，偏移量，大小，标识等等（如上图）都在 section header table 中得到了表示。而 section header table 其实就是 8 个相同（40 个）字节大小的 section header 组成的，这些 section header 包含了每个 section 的相关信息。

所以，从 elf header 可以定位到 section header table，从 section header table 又可以定位到每个 section header，从 section header 就可以直接找到每个 section 的内容了，所有这些都是这么串起来的！！这也是为什么 elf 格式的文件可以用来连接的原因！——可以很轻松就找到各个 section 进行连接！

Section header 的信息如上图 readelf 读出所示，其数据结构如下面定义，我们也可以按照资料一一对应起来。

```

typedef struct {
    Elf32_Word sh_name;
    Elf32_Word sh_type;
    Elf32_Word sh_flags;
    Elf32_Addr sh_addr;
    Elf32_Off sh_offset;
    Elf32_Word sh_size;
    Elf32_Word sh_link;
    Elf32_Word sh_info;
    Elf32_Word sh_addralign;
    Elf32_Word sh_entsize;
}

```



```
} Elf32_Shdr;
```

其中和我们比较密切、比较重要的是 sh_name、sh_type、sh_offset 三个量：

- sh_name：该成员指定了这个 section 的名字。它的值是 section 报头字符表 section 的索引。
- sh_type：该成员把 sections 按内容和意义分类。
- sh_offset：该成员变量给出了该 section 的字节偏移量(从文件开始计数)。

上面说了这么多 elf 文件格式，就是想引出三个很重要的 section，所以，如果你看不懂上面的 elf 文件也没有关系，只要大概知道用 gcc 生成的 object 文件和可执行文件有这么个回事就好了！**关键知道其中包含了 section !!**

到此为止，我们分析了 GNU 编译后生成的映象文件或即将成为映象文件的 object 文件的格式——elf 文件格式，并用了相关的工具演示了 elf 中的内容。目的就是想告诉大家，**用 GNU 编译后生成的映象文件或即将成为映象文件的 object 文件中包含了一些 section!那到底包含了哪些 section，这些 section 怎么生成的？这些 section 有什么用？**

2.2 section 出场

让我们来看看 HEAD.o 中包含的 8 个 section 吧！从图 4 中可以看出，

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.text	PROGBITS	00000000	000034	0000fc	00	AX	0	0	4
[2]	.rel.text	REL	00000000	0004a4	000070	08		6	1	4
[3]	.data	PROGBITS	00000000	000130	000000	00	WA	0	0	1
[4]	.bss	NOBITS	00000000	000130	000000	00	WA	0	0	1
[5]	.shstrtab	STRTAB	00000000	000130	000036	00		0	0	1
[6]	.symtab	SYMTAB	00000000	0002a8	000160	10		7	d	4
[7]	.strtab	STRTAB	00000000	000408	000099	00		0	0	1

- 第一个 section 是空的，也是必须的，规定这样，没有办法！
- 第二个是 .text：这个 section 保存着程序的 ``text`` 或者说是**可执行指令，类似于 RO**
- 第三个是 .rel.text 这个 section 保存着重定位的信息，也就是说包含了 .text 段中可重定位的信息
- 第四个是 .data 这个 sections 保存着初始化了的数据，那些数据存在于程序内存映象中，**相当于 RW**
- 第五个是 .bss 该 section 保存着未初始化的数据，这些数据存在于程序内存映象中。通过定义，当程序开始运行，系统初始化那些数据为 0。该 section 不占文件空间。**相当于 ZI**
- 第六个、第七个、第八个是系统使用的 section，在编译连接时使用，生成真正在 ROM 中执行的 bin 文件时，这些 section 被删掉了，所以和我们无关。

这样分析后，让我们下个结论吧！在 The gnu Assembler 文档中：

An object file written by *as* has at least three sections, any of which may be empty. These are named text, data and bss sections.

Within the object file, the text section starts at address 0, the data section follows,

and the bss section follows the data section.

也就是说,用gcc编译后,生成的object文件最起码包含了三个默认段(section):text, data and bss sections,哪怕是空的,这个和ARM映象文件中域包含RO,RW,ZI真是有异曲同工之处。同时,生成object文件时,text段在0x0处,data段紧跟其后,最后是bss段。也和ARM映象文件中的加载域的组织一样。

我打开了bios中好几个object文件,例如setup.o、bios.o等等,发现基本上都如上面所叙述那样,除了系统使用的.shstrtab、.symtab、.strtab三个section外,.rel.text在连接后也没有其他作用,基本上只有.text、.data、.bss三个重要section,还有一个很重要的section,在HEAD.o没有发现,就是.rodata

.rodata : 这个section保存着只读数据,在进程映象中构造不可写的段。该段也是系统自动生成。

让我们来通过另一个工具 objdump 来看看 section,这个工具把和我们无关的 section 都过滤了,不过包含了.rodata,见图5。



```

root@dudoo: ~/bios/setup
文件(E) 编辑(E) 查看(V) 终端(T) 转到(G) 帮助(H)
[root@dudoo setup]# objdump -h setup.o
setup.o:      file format elf32-little

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text          00000960  00000000  00000000  00000034  2**2
                CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data          00000000  00000000  00000000  00000994  2**0
                CONTENTS, ALLOC, LOAD, DATA
  2 .bss           00000000  00000000  00000000  00000994  2**0
                ALLOC
  3 .rodata        000002e4  00000000  00000000  00000994  2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA

[root@dudoo setup]#

```

图5 用objdump看到的bios/setup/setup.o文件的section

【在生成最后的可执行文件时,其他系统使用的段都不被连接进去,连接进去的只有刚刚提到的四个段(section):.text、.rodata、.data、.bss。而且一般还将.rodata连接到.text段中以保持和ARM映象文件3个段(RO、RW、ZI)的兼容性。】

对了,讲了这么久,还没有回答HEAD.S中的那个问题:.section是做什么用的?☺

For ELF targets, the assembler supports another type of .section directive for compatibility with the Solaris assembler:

.section "name" [, flags...]

Note that the section name is quoted. There may be a sequence of comma separated

flags:

#alloc section is allocatable

#write section is writable

```
#execinstr section is executable
```

OK,由HEAD.S引出的问题，到此告了一个段落！！

看了以上又臭又长的分析，你了解大概说了个什么意思吗？一言蔽之，就是“gcc编译文件后一般自动生成的几个重要section，包括.text .data .bss，这些重要的段和ARM的RO、RW、ZI都可以一一对应的上，不过这里的section还不能运行，因为还没有连接重定位”。

已经找到了section，也就是RO,RW,ZI，仅仅找到这些段还没有用阿，毕竟他们还是单独的个体，没有连接起来，等于一盘散沙！也就是说，我们怎么才可以把bios中那么多object文件中的这些section连接起来呢？怎么设置相应的ro_base，rw_base，entry呢？怎么重定位哪些符号呢？天啦，还有很多问题等着我们，不过，这是挑战，也会有成功的快乐！

让我们再往下分析！

三 GNU ld——连接工具

连接工具ld读取要进行连接的object文件，将其中的内容组成一个可执行的文件，连接过程包含了相同section的整合、重定位等操作！当然，所有这些操作，建立在as生成了elf格式的object文件基础上！也就是我们刚刚分析的层次上！以下摘自GAS：

ld moves blocks of bytes of your program to their run-time addresses. These blocks slide to their run-time addresses as rigid units; their length does not change and neither does the order of bytes within them. Such a rigid unit is called a section. Assigning runtime addresses to sections is called relocation.

这句话很精辟的解释了ld的操作和section的关系，每个object文件中的每个sectin都作为一个整体，不可以随便改变整体内部的一切东西，只能分配给这个section以运行的地址，这个过程就叫重定位（relocation）

在这里举的例子就是最后要生成的bios可执行文件，正如Makefile所写那样：

```
bios: $(OBJ)
```

```
$(LD) -p -X -T bios.ld $(OBJ) -o bios
```

```
OBJ = head.o bios.o gunzip.o utils.o console.o bioscall.o sysinit.o biosapi.o setup.o fdisk.o
tftp.o
```

假设其中的OBJ都已经生成，那么我们怎么连接他们呢？这么讲会不会太直接了一些？！

这里又引出了一个很重要的文件：bios.ld

3.1 bios.ld

请去看看bios.ld，这是和HEAD.S、Makefile同样排名前三的重要文件！！该文件在根目录下。

```
OUTPUT_ARCH(arm)
```

```
ENTRY(stext)
```

```
SECTIONS
```

```
{
```

```
.text 0x01000000 : {          /* Real text segment          */
```

```
    _text = .;          /* Text and read-only data    */
```

```
    *(.text)
```

```
        *(.rodata)
        . = ALIGN(4);
        _etext = .;          /* End of text section      */
    }

    .data 0x03fe0000 : {
        __data_start = .;
        *(.data)
        . = ALIGN(4);
        _edata = .;
    }

    .bss : {
        __bss_start = .;    /* BSS                */
        *(.bss)
        . = ALIGN(4);
        _end = .;
    }

    .stack 0x03fe1000 : {
        __stack = .;       /* STACK                */
    }
}
```

大家不妨自己先看看这个文件，看到这么多熟悉的名称，和熟悉的定义，应该可以领悟出一些东西吧！！

（时间过了15分钟）

到底这些类似脚本语言的命令有什么作用了？也许你刚刚已经猜到，不过我们还是不要妄加评论，来看看官方文档的说法吧

官方文档为：Using ld——The GNU linker，google上应该可以搜到的！

介绍起linker，内容又好多阿！这篇文章写得我实在很想知道什么时候可以结束阿！不过，为了初学者可以少走弯路，为了给嵌入式同行们的开发学习奉献我非常微薄的力量☺，我还是继续了！！如果你继续不下去了，那可以不必勉强了，休息一下！

脚本到底有什么用呢？

The main purpose of the linker script is to describe how the sections in the input files should be mapped into the output file, and to control the memory layout of the output file.

从上面可以看出，连接脚本主要用来描述输入文件的段如何被映射到输出文件中，并且控制输出文件的存储配置！

要看懂这个连接脚本 (linker script), 就要懂得连接文件的格式 (format), 我们就上面的 bios.ld 来说明连接脚本的格式吧, 在整个 bios 中, 所有的 ld 脚本文件基本都是这个样的。

- **OUTPUT_ARCH(bfdarch)** Specify a particular output machine architecture. 当然是指定输出文件的机器架构, 这里就是 arm 架构
- **ENTRY(symbol)** 这个很明显, 当然是定义程序入口点, 也就是说 bios 的程序入口点在 head.S 中的 stext 的位置
- **SECTIONS** The SECTIONS command tells the linker how to map input sections into output sections, and how to place the output sections in memory. 一看就知道这是最重要的命令了!

3.2 连接脚本中的 SECTIONS 命令

SECTIONS 里主要负责输入文件到输出文件的映射, 输出文件中段在内存中的位置。命令格式为:

```
SECTIONS
{
    sections-command
    sections-command
    ...
}
```

每个 sections-command 可以是 ENTRY(), 符号赋值, 输出段 (output section) 描述, 还有覆盖描述, 在上面的例子中我们可以看到输出段的描述, 符号赋值操作, 因为这个也是脚本文件的主要作用, ENTRY 放在前面了。

输出段的描述格式如下:

```
section [address] [(type)]: [AT(lma)]
{
    output-section-command
    output-section-command
    ...
} [>region] [AT>lma region] [:phdr :phdr ...] [=fillexp]
```

这里面包含了很多附加选项 (蓝色标志), 其实很多都没有用到。看看上面的例子, 先看看 .text section 吧, 还记得这个段吧, 如果忘了, 那你可以先休息一下, 再从头开始看吧☺

```
.text 0x01000000 : {                /* Real text segment      */
    _text = .;                       /* Text and read-only data */
    *(.text)
    *(.rodata)
    . = ALIGN(4);
    _etext = .;                      /* End of text section    */
}
```

对着上面的格式, 可以看出: .text 段被连接到 0x01000000 处, 这个其实就是 ro_base 噢! 然后接下来的是赋值语句:

```
_text = .;
```

注意，. 这个特殊的符号是表示当前的位置计数器，这个赋值也就是将当前地址（0x01000000）赋值给_text这个变量，这个变量可以在程序中引用，所以大家在HEAD.S中看到了这个变量，它的值就是.text的起始地址，也就是ARM映象文件的|Image\$\$RO\$\$Base|。最后一句赋值语句 _etext = .;也是相同的意思，就是将当前的地址计数器的值赋给_etext，这里注意当前地址计数器的值已经等于.text的开始值（0x01000000）加上了所有的.text段，再加上所有的.rodata后，得到的最新值，其实也就是ARM映象文件中的|Image\$\$RO\$\$LIMIT|（经过对齐操作后）。. = ALIGN(4);是地址对齐操作。

那么，*(.text)和 *(.rodata) 这两个操作是什么意思呢？这是 input section description，输入段描述。*表示通配符，就是被连接的所有object文件中的.text和.rodata段都被连接进这个输出段中，其实我们仔细想想，这个和ARM中的输入段，输出段，域好像概念上就是一样的嘛！！

其他的两个.data和.bss，还有.stack也是很.text 一样！同样我们也在HEAD.S中用到了__bss_start和_end 的，他们代表的意思，你应该可以看得懂了吧！他们一样可以被引用。

```
.data 0x03fe0000 : {
    __data_start = .;
    *(.data)
    . = ALIGN(4);
    _edata = .;
}
```

其中0x03fe0000就是rw_base，这个段就是要移动的数据段，所以HEAD.S中我们看到了一份代码就是关于移动这个数据段的，引用了__data_start和_edata这个数据段的头尾变量，不信可以去看看噢！

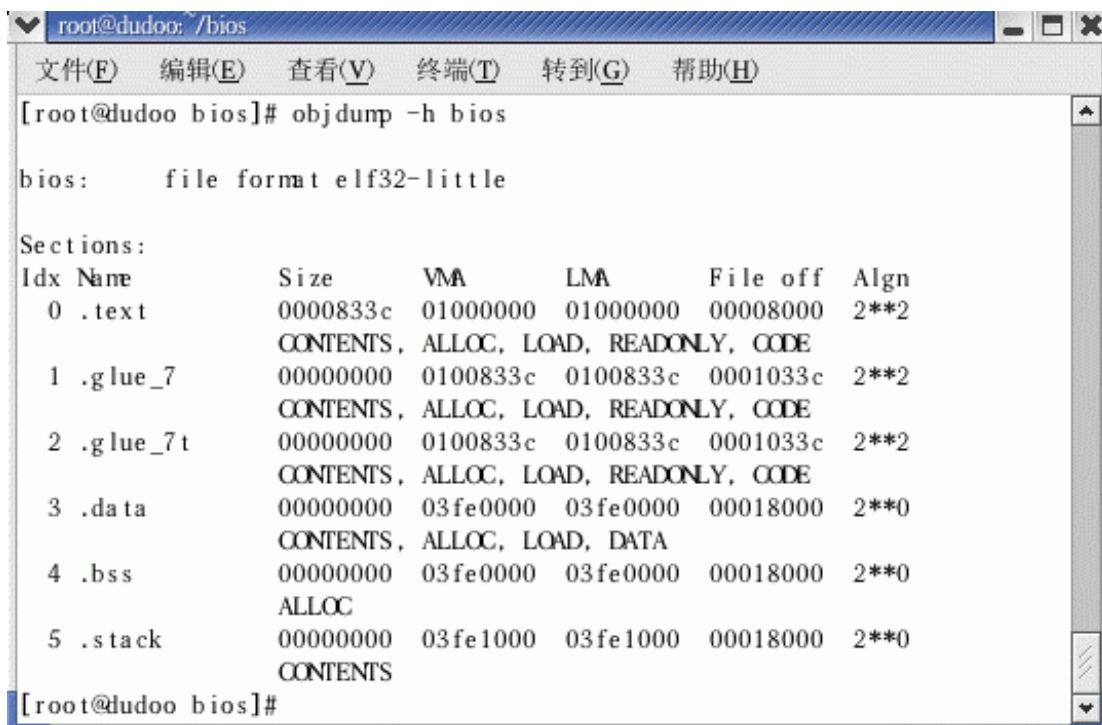
```
.bss : {
    __bss_start = .; /* BSS */
    *(.bss)
    . = ALIGN(4);
    _end = .;
}
```

其中.bss就没有规定地址，那么就是当前地址计数器的值，也就是紧跟在RW后面了还有一个stack段：

```
.stack 0x03fe1000 : {
    __stack = .; /* STACK */
}
```

其中0x03fe1000是其开始地址，那么也就是__stack=0x03fe1000，这个变量也在HEAD.S中被引用阿！！

下面我们最后来看看一张图



```
root@dudoo: /bios
文件(F) 编辑(E) 查看(V) 终端(T) 转到(G) 帮助(H)
[root@dudoo bios]# objdump -h bios

bios:      file format elf32-little

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text          0000833c  01000000  01000000  00008000  2**2
             CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .glue_7        00000000  0100833c  0100833c  0001033c  2**2
             CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .glue_7t       00000000  0100833c  0100833c  0001033c  2**2
             CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .data          00000000  03fe0000  03fe0000  00018000  2**0
             CONTENTS, ALLOC, LOAD, DATA
  4 .bss           00000000  03fe0000  03fe0000  00018000  2**0
             ALLOC
  5 .stack         00000000  03fe1000  03fe1000  00018000  2**0
             CONTENTS

[root@dudoo bios]#
```

图6 用objdump显示bios的段section

大家可以根据这张表，核对一下刚刚的连接内存分配，其中.glue_7和.glue_7t应该是系统的“粘合”用的段，没有什么其他意思，可以忽略它，在最后的objcopy中自然会删掉它！大家主要从中感受一下刚刚连接过程！

总结来说，就是最后生成的bios这个可执行文件共4个输出段（section），分别为.text、.data、.bss、.stack，稍微比ARM映象文件多了一个.stack段，不过不影响映象文件执行。对于每个输出段来说，分别由组成该可执行文件的object文件中的相应的section输入段组成。比如说最后的bios就是一个ARM域，那么.text、.data、.bss就分别是RO、RW、ZI输出段，组成这些输出段的是各个被连接文件的“属性”相同的输入段，这里的“属性”的区分仍然是按“只读：指令和只读数据”“可读写数据”“初始化为0数据”分类的，只是由GNU用.text、.data、.bss这几个名字来代替了而已！！

最后再来一张图，帮助理解上面这段话吧，顺便结束这篇文章！

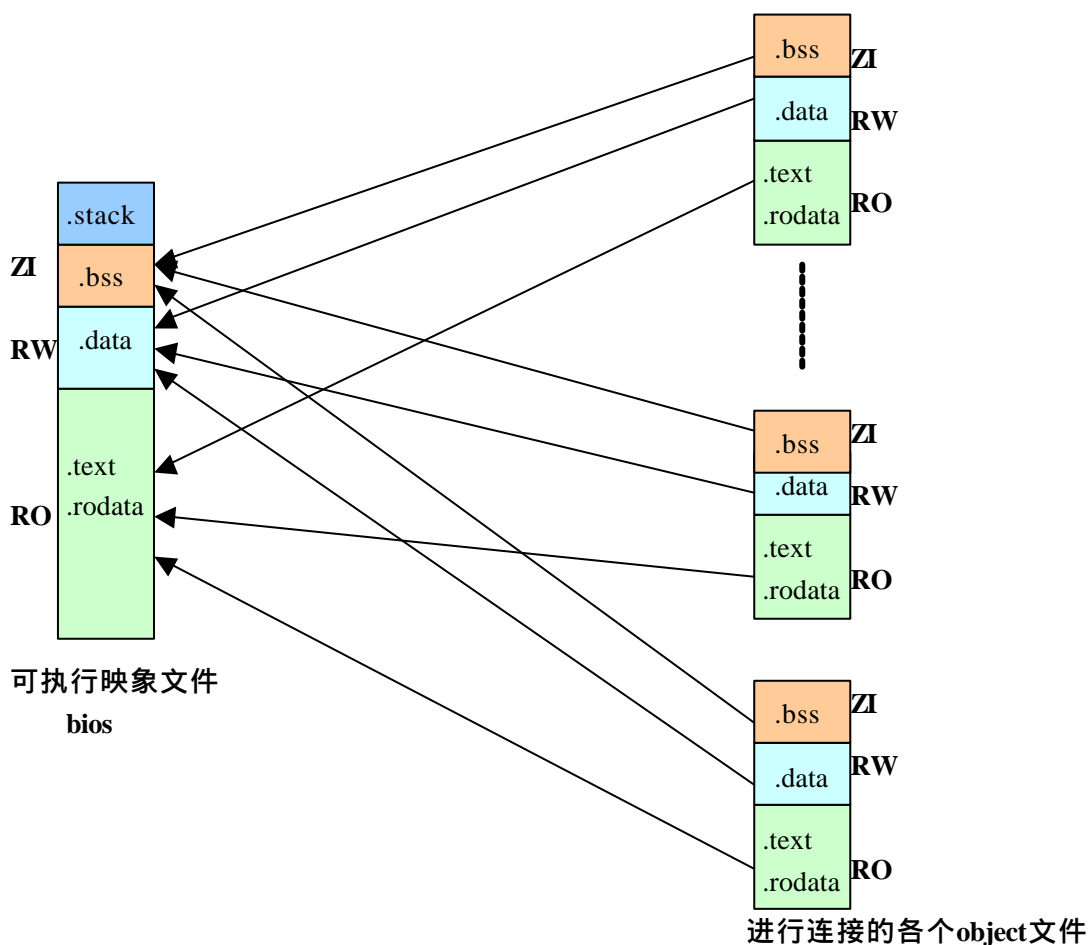


图7 连接总图

特别提示：由于bios比较大，所以其中的文件之间的连接关系比较复杂，采用的一些方法也比较奇特，比如，将一些object文件变为只包含一个大数组的C文件，然后再进行编译，形成整个object文件的.text段，再配合config.h，从而实现相互正确连接，这些原理，我们将在报告003为大家讲述！这篇文章只是一个介绍段和连接的基础篇。

对于GNU tools 开发ARM程序及生成映象文件机理，以上基本上都提到了，由于内容牵涉太多，一时可能不能一下体会理解，慢慢多看看，再结合报告001，就会更有体会了。我没有办法写得更“罗嗦”，因为我也没有力气再写了……实在看不太懂，可以在让我们在ARM报告003再见吧！

byebye！

文章版权属于杜云海（ duyunhai@hotmail.com ）， 转载请注明作者及网站（ www.seajia.com ）

快乐，一个人独享，只有一份快乐；和100个人分享，有100份快乐；和1000人，10000人分享，那么快乐就无止境地增长着