ARM GCC linker 脚本介绍

Team MCUZONE
整理自网络文章

在输入文件在进行链接的时，每个链接都由链接脚本控制着，脚本由链接器命令语言组成。脚本的主要目的是描述如何把输入文件中的节（sections）映射到输出文件中，并控制输出文件的存储布局。大多数的链接脚本就是做这些事情的，但在有必要时，脚本也可以指导链接器执行一些其他的操作。

链接器总是使用链接器脚本，如果你没有提供一个自定义的脚本文件的话，编译器会使用一个缺省的脚本。

1 链接器脚本的基本概念

链接器把一些输入文件联合在一起，生成输出文件。输出的文件和输入文件都是特定的 object 文件格式，每个文件都可被称为对象文件（object file）,而且，输出文件还经常被称为可执行文件。但这里我们依然称之为对象文件。每个对象文件在其中都包含有一个**段**（section）列表，我们有时称输入文件中的**段**（section）为输入**段**（input section），同样，输出文件中的节称为输出**段**（output section）。

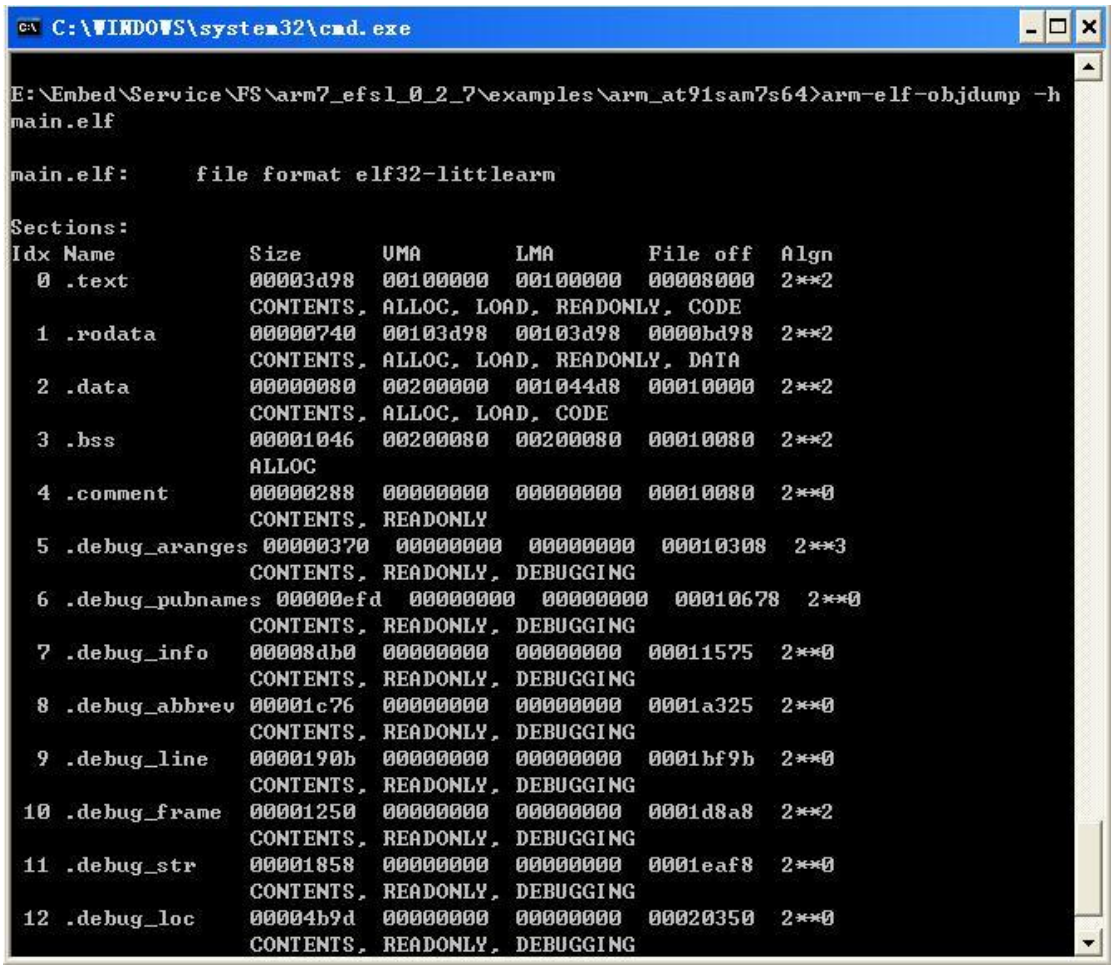
对象文件中的每一个**段**都有名字和大小。大多数的**段**还有一个相连的数据块，就是有名的"section contents"。一个被标记为可加载（loadable）的**段**，意味着在输出文件运行时，contents 可以被加载到内存中。没有 contents 的节也可以被加载，实际上除了一个数组被设置外，没有其他的东西被加载（在一些情况下，存储器必须被清 0）。而既不是可加载的又不是可分配的（allocatable）**段**，通常包含了某些调试信息。

每个可加载或可分配的输出**段**（output section）都有 2 个地址。第一个是虚拟存储地址 VMA（virtual memory address），这是在输出文件执行时该**段**所使用的地址。第二个是加载存储地址 LMA（load memory address），这是该**段**被加载时的地址。在大多数情况下，这两个地址是相同的。举个例子说明不同时的情况：当一个数据节（data section）加载在 ROM 中，后来在程序开始执行时又拷贝到 RAM 中（在基于 ROM 的系统中，这种技术经常用在初始化全局变量中）。在这种基于 ROM 的系统情况下，这时，ROM 地址是 LMA，而内存地址是 VMA。

要查看一个对象文件中各个节，可以使用 objdump，并使用"-h"参数。

下图显示了改参数的执行结果，注意**段**名和地址。

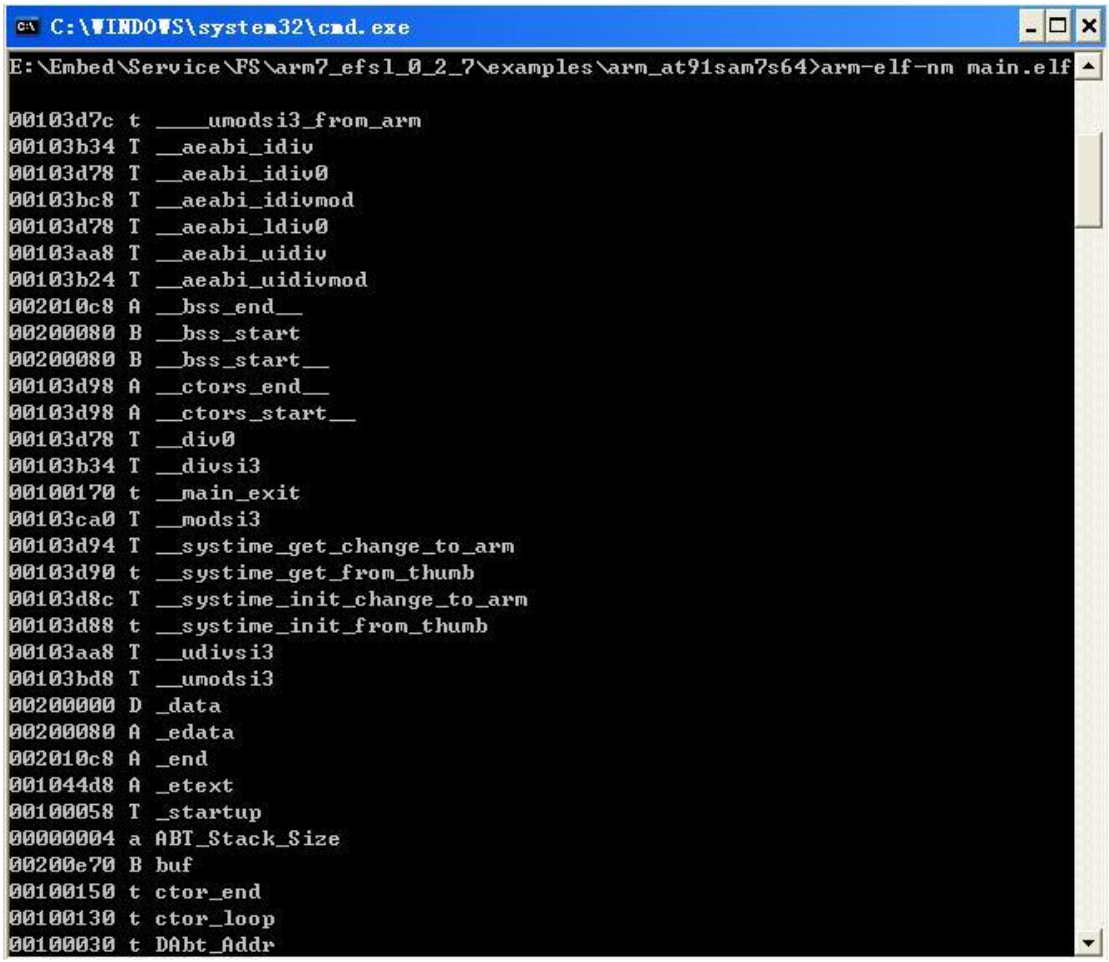
该输出信息可以用来确定每个**段**的实际尺寸和位置。



每个对象文件也有一个符号（symbols）列表，这就是著名的符号表(symbols table)。一个符号可以是"已定义"（defined）或"无定义"（undefined）的。每个符号有名字，并且每个定义了的符号还有地址。在你编译一个 c/c++程序成对象文件时，每个定义的函数，全局变量，静态变量，都可以有一个"已定义"的符号。输入文件中引用的每个没有定义的函数和全局变量则变成"无定义"的符号。

使用 nm 可以查看对象文件中的符号，objdump 并使用"-t"选项也可以。

该信息输出可以用于定位变量和模块的位置。



2 链接器脚本格式

链接器脚本是一个文本文件。

链接器脚本是一个命令序列，每个命令是一个关键字，可能还带着参数，又或者是对一个符号的赋值。可以使用分号来隔开命令，而空格则通常被忽略。

像文件名，格式名等字符串通常直接输入，如果文件名包含有像用于分割文件名的逗号等有其他用处的字符的话，你可以用双引号把文件名括起来。当然没有办法在文件名中使用双引号了。

可以使用注释，就像在 C 中，定界符是"/*"和"*/"，和 C 中一样，注释在语法上等同于空格。

3 简单的脚本例子

很多的链接脚本都比较简单。可能最简单的链接器脚本只有一个命令： 'SECTIONS'。使用'SECTIONS'命令描述输出文件的内存布局。

'SECTIONS'命令功能强大。这里描述一个简单的应用。假设只有代码（code），初始数据（initialized data）和未初始化的数据（uninitialized data）。它们要分别被放到'.text', '.data', '.bss'段中。更进一步假定它们是输入文件中的所有的节。

这个例子中，代码要加载到地址 0x10000，数据要从地址 0x8000000 开始。链接脚本如下：

```
SECTIONS
{
  . = 0x10000;
  .text : { *(.text) }
```

```
  . = 0x8000000;
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

'SECTIONS'命令的关键字是'SECTIONS'，接着是一系列的符号(symbol)赋值，输出**段**（output section）描述被大括号包括着。

上面例子中，在'SECTIONS'命令里面，第一行设置一个值到一个特殊的符号'.'，它是位置计数器（location counter），（像程序计数器 PC）。如果没有以某种其他的方式指定输出**段**(output section)的地址，地址就会是位置计数器中设置的当前值。而后，位置计数器就会以输出**段**的大小增加其值。在'SECTIONS'命令的开始，位置计数器是 0。

第 2 行定义'.text'输出段。冒号是必须的语法。在大括号里，输出节名字之后，你要列出要输出段的名字，它们会放入输出段中。通配符"*"匹配任何文件名，表达式"*(.text)"意味着所有的输入文件中的输入段".text"。

因为在输出段".text"定义前，位置计数器被定义为 0x10000，所以链接器会设置输出文件中的".text"段的开始地址为 0x10000。

剩下的行定义输出文件中的".data"和".bss"段。链接器会把输出段".data"放置到地址 0x8000000。之后，链接器把输出段".data"的大小加到位置计数器的值 0x8000000， 并立即设置".bss"输出段，效果是在内存中，".bss"段会紧随".data"之后。

链接器会确保每个输出段都有必要的对齐，它会在需要时增加位置计数器的值。在上面的例子中，指定的".text"和".data"段的地址都是符合对齐条件的，但是链接器可能会在".data"和".bss"间生成一个小间隙。

4 简单的链接器脚本命令

4.1 设置入口点

在一个程序中第一个指令称为入口点（entry point）。可以使用 ENTRY 链接器脚本命令来设置入口点。参数是一个符号名。

ENTRY(symbol)

有几种不同的方式来设置入口点。链接器会依次用下面的方法尝试设置入口点，当遇到成功时则停止。

命令行选项"-e" entry

脚本中的"ENTRY(symbol)"

如果有定义"start"符号，则使用 start 符号（symbol）。

如果存在".text"节，则使用第一个字节的地址。

地址 0。

4.2 处理文件的命令

有一些处理文件的命令：

INCLUDE filename

在该点包含名称 filename 的链接脚本文件。文件会在当前路径下进行搜索，还有通过选项"-L"指定的目录。可以进行嵌套包含，你可以最多嵌套 10 层。

INPUT（file,file,…） 或 INPUR(file file …)

INPUT 命令指示链接器在链接中包含指定的文件，好像它们命名在命令行上一样。

例如，如果你总是要在链接时包含"subr.o"，但是你又不想很烦地每次在命令行上输入，你

可以在你的链接脚本中使用"INPUT(subr.o)"。
实际上，如果你喜欢，你可以在链接脚本中列出所有的输入文件，然后使用选项"-T"来调用链接器，而不用做其他的。
链接器首先尝试打开当前目录下的文件，如果没有，就通过存档库搜索路径进行搜索。你可以查看"-L"选项说明。
如果你使用`INPUT(-l file)`，ld 会把它转化成 libfile.a，就想在命令行中使用"-l"参数。
当你在一个隐式链接脚本中使用"INPUT"命令时，在链接器脚本文件被包含的点，文件会被包含进去。这会影响文档（archive）的搜索。

GROUP(file, file, …) GROUP(file file …)

GROUP 命令类似于 INPUT 命令，处了其文件为文档 archive 外。它们会被重复搜索，知道没有新的无定义(undefined)引用被创建。请查看"-("参数的描述。

OUTPUT（filename）

该命令指定输出文件的名字。相当于命令行中的`-o filename`参数。如果都使用了，命令行选项会优先。
可以用 OUTPUT 命令来定义一个缺省的输出文件名，而不是无用的缺省`a.out`。

SEARCH_DIR(path)

此命令添加路径 path 到 ld 搜索库文档 achive 的路径列表中。相当于命令行方式下的`-L path`。如果都设置了，则都添加到列表中，而且，命令行中的在前，优先搜索。

STARTUP(filename)

此命令和 INPUT 命令相似，处了 filename 会成为第一个被链接的输入文件外，就想在命令行上被第一个输入。如果处理入口点总是第一个文件的开始，在这样的系统中，这会很有用。

4.3 处理对象文件（object file）格式的命令

处理对象文件格式的命令只有 2 个。

OUTPUT_FORMAT(bfdname) OUTPUT_FORMAT(default, big, little)

此命令为用户的输出文件命名 BFD 格式。相当于命令行中使用选项`--oformat bfdname`，如果都使用了，命令行方式优先。
可以使用 3 个参数的 OUTPUT_FORMAT 指令，指定使用的不同的格式，就像命令行方式下的选项`-EB`, `-EL`。这样允许链接器脚本设置输出格式是指定的 endianness 编码。
如果没有使用`-EB`或`-EL`指定 endianness，输出格式会是第一个参数 default。如果使用`-EB`，那么使用第二个参数 big，使用了`-EL`，则使用参数 little。
例如，目标 MIPS ELF 使用的缺省链接脚本使用命令：
OUTPUT_FORMAT(elf32-bigmips, elf32-bigmips, elf32-littlemips)
这就是说，输出的缺省格式是 elf32-bigmips，但是如果用户在命令行指定了选项`-EL`，则使用 elf32-littlemips。

TARGET(bfdname)

当读取输入文件时，此命令为用户命名 BFD 格式。它会影响随后的 INPUT 和 GROUP 命令。此命令相当于命令行方式的选项`-b bfdname`。 如果使用了 TARGET 命令，而没有使用 OUTPUT_FORMAR，那么最后的 TARGET 命令也设置输出文件的格式。

4.4 其它的链接器脚本命令

有一些其他的脚本命令：

ASSERT(exp, message)

确保表达式 exp 非零。如果为 0，则退出链接，返回错误码，打印指定的消息 message。

EXTERN(symbol symbol …)

强制要进入输出文件的指定的符号成为无定义 undifined 的符号。这么做，可以触发从标志

库对附加模块的链接。可以列出多个符号 symbol。对每个 EXTERN，你可以使用 EXTERN 多次。这个命令和命令行下的选项`-u`产生一样的效果。

FORCE_COMMON_ALLOCATION

此命令的效果和命令行的选项`-d`一样，让 ld 分配空间给公共 common 的符号，即使通过选项`-r`指定的是一个重定位的输出文件。

INHIBIT_COMMON_ALLOCATION

此命令和命令行选项`--no-define-common`有同样效果：让 ld 忽略对公共符号的赋值，即使是一个非可重定偎的（non-relocatable）输出文件

NOCROSSREFS(section section …)

此命令也许可以用来告诉 ld 在指定的输出节中对任何的引用发出一个错误。

在某些特别类型的程序中，特别是嵌入式系统，如果使用覆盖图 overlays，当一个节加载到内存中，而另一个节不在。这两个节间的任何方向的引用都会出错，例如，一个节中的代码调用另一个节中的函数。

NOCROSSREFS 命令带有一个输出节名字列表。如果 ld 测试在这些节之间有任何的交叉引用，就会报告一个错误，并返回一个非 0 的状态。

注意，此指令使用的时输出节，而不是输入节。

OUTPUT_ARCH(bfdarch)

指定一个特定的机器架构输出。参数是 BFD 库中使用的名字。可以使用 objdump 指定参数选项`-f`来查看架构。如 ARM。

```
E:\Embed\Service\FS\arm7_efsl_0_2_7\examples\arm_at91sam7s64>arm-elf-objdump -f
main.elf

main.elf:     file format elf32-littlearm
architecture: arm, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00100000
```

可以通过使用`**>REGION'**把一个段赋给前面已经定义的一个内存区域。这里有一个简单的例子**:**

```
MEMORY { rom : ORIGIN = 0x1000, LENGTH = 0x1000 }
SECTIONS { ROM : { *(.text) } >rom }
```

5 为符号指定值

在脚本中，可以为一个符号 symbol 指定一个值。这样会把符号定义为全局符号 symbol。

5.1 简单赋值

使用任何的 C 赋值操作来给一个符号赋值。像下面这样：

symbol = expression ;

symbol += expression ;

symbol -= expression ;

symbol *= expression ;

symbol /= expression ;

symbol <<= expression ;

symbol >>= expression ;

symbol &= expression ;
symbol |= expression ;
第一个例子中，定义了一个符号 symbol，并赋值为 expression。其他的例子中，symbol 必须已被定义，根据操作调整其值。
特殊的符号名"."指示的是位置计数器 location counter。只有在 SECTIONS 命令中才可以使用。
表达式后的"；"是必须的。你可以像命令一样按它们的顺序写符号赋值，或者在 SECTIONS 命令中像一个语句一样。或者作为 SECTIONS 命令中输出节描述器的一部分。
例子：

```
floating_point = 0;
SECTIONS
{
  .text :
    {
      *(.text)
      _etext = .;
    }
  _bdata = (. + 3) & ~ 3;
  .data : { *(.data) }
}
```

在例子中，符号`floating_point`会被定义为 0。符号_etext 会被定义为最近的输入段.text 地址，也就是.text 段的结束地址(end of .text)。符号_bdata 定义为接着.text 输出节的地址，但对齐到了 4 字节的边界。

5.2 PROVIDE

在一些情况下，链接器脚本想要定义一个符号，这个符号仅仅被引用但没有被任何链接器中包含的对象定义。例如，传统的链接器定义符号"etext"作为一个函数名，而不会遇到错误。PROVIDE 关键字可以用来定义像这样的符号，仅仅在它被引用却没有定义时。语法是：PROVIDE（symbol=expression）。
下面是使用 PROVIDE 定义"etext"的例子：

```
SECTIONS
{
  .text :
    {
      *(.text)
      _etext = .;
      PROVIDE(etext = .);
    }
}
```

在例子中，如果程序定义了"_etext"（有下划线），链接器会给出多个定义的错误。另一方面，如果程序定义"etext"（没有下划线），链接器会在程序中静静地使用这个定义。如果程序引用"etext"但没有定义它，链接器会使用脚本中的定义。

完全的 linker 脚本的内容远不止这些，但是看了上面的说明，再来看 winarm 中提供的 ld 文件，理解起来应该容易些。

```
/*----------------------------------------------------------------------------*/
/*-            ATMEL Microcontroller Software Support  -  ROUSSET  -         */
/*----------------------------------------------------------------------------*/
/* The software is delivered "AS IS" without warranty or condition of any     */
/* kind, either express, implied or statutory. This includes without          */
/* limitation any warranty or condition with respect to merchantability or    */
/* fitness for any particular purpose, or against the infringements of        */
/* intellectual property rights of others.                                    */
/*----------------------------------------------------------------------------*/
/*- File source          : GCC_FLASH.ld                                       */
/*- Object               : Linker Script File for Flash Workspace             */
/*- Compilation flag     : None                                               */
/*-                                                                           */
/*- 1.0 20/Oct/04 JPP    : Creation                                           */
/*----------------------------------------------------------------------------*/

/* slightly modified for the WinARM example - M.Thomas (not Atmel) */

/*
//*** <<< Use Configuration Wizard in Context Menu >>> ***
*/


/*
// <h> Memory Configuration
//     <h> Code (Read Only)
//         <o>   Start <0x0-0xFFFFFFFF>
//         <o1> Size   <0x0-0xFFFFFFFF>
//     </h>
//     <h> Data (Read/Write)
//         <o2> Start <0x0-0xFFFFFFFF>
//         <o3> Size   <0x0-0xFFFFFFFF>
//     </h>
//     <h> Top of Stack (Read/Write)
//         <o4> STACK <0x0-0xFFFFFFFF>
//     </h>
// </h>
*/

/* Memory Definitions */

/* mt change code origin from 0x00000000 */
MEMORY
{
```

```
   CODE (rx) : ORIGIN = 0x00100000, LENGTH = 0x00010000
   DATA (rw) : ORIGIN = 0x00200000, LENGTH = 0x00004000
   STACK (rw) : ORIGIN = 0x00204000,LENGTH = 0x00000000
}


/* Section Definitions */

SECTIONS
{
   /* first section is .text which is used for code */
   . = 0x0000000;
   .text : { *cstartup.o (.text) }>CODE =0
   .text :
   {
      *(.text)                        /* remaining code */

      *(.glue_7t) *(.glue_7)

   } >CODE =0

   . = ALIGN(4);

   /* .rodata section which is used for read-only data (constants) */

   .rodata :
   {
      *(.rodata)
   } >CODE

   . = ALIGN(4);

   _etext = . ;
   PROVIDE (etext = .);

   /* .data section which is used for initialized data */

   .data : AT (_etext)
   {
      _data = . ;
      *(.data)
      SORT(CONSTRUCTORS)
   } >DATA
   . = ALIGN(4);
```

```
_edata = . ;
  PROVIDE (edata = .);

/* .bss section which is used for uninitialized data */

.bss :
{
    __bss_start = . ;
    __bss_start__ = . ;
    *(.bss)
    *(COMMON)
}
. = ALIGN(4);
__bss_end__ = . ;
__bss_end__ = . ;
_end = .;
    . = ALIGN(4);
  .int_data :
  {
  *(.internal_ram_top)
  }> STACK

PROVIDE (end = .);

/* Stabs debugging sections.    */
.stab                 0 : { *(.stab) }
.stabstr              0 : { *(.stabstr) }
.stab.excl            0 : { *(.stab.excl) }
.stab.exclstr     0 : { *(.stab.exclstr) }
.stab.index           0 : { *(.stab.index) }
.stab.indexstr 0 : { *(.stab.indexstr) }
.comment              0 : { *(.comment) }
/* DWARF debug sections.
      Symbols in the DWARF debugging sections are relative to the beginning
      of the section so we begin them at 0.    */
/* DWARF 1 */
.debug                0 : { *(.debug) }
.line                 0 : { *(.line) }
/* GNU DWARF 1 extensions */
.debug_srcinfo    0 : { *(.debug_srcinfo) }
.debug_sfnames    0 : { *(.debug_sfnames) }
/* DWARF 1.1 and DWARF 2 */
.debug_aranges    0 : { *(.debug_aranges) }
```

```
  .debug_pubnames 0 : { *(.debug_pubnames) }
  /* DWARF 2 */
  .debug_info     0 : { *(.debug_info .gnu.linkonce.wi.*) }
  .debug_abbrev   0 : { *(.debug_abbrev) }
  .debug_line     0 : { *(.debug_line) }
  .debug_frame    0 : { *(.debug_frame) }
  .debug_str      0 : { *(.debug_str) }
  .debug_loc      0 : { *(.debug_loc) }
  .debug_macinfo  0 : { *(.debug_macinfo) }
  /* SGI/MIPS DWARF 2 extensions */
  .debug_weaknames 0 : { *(.debug_weaknames) }
  .debug_funcnames 0 : { *(.debug_funcnames) }
  .debug_typenames 0 : { *(.debug_typenames) }
  .debug_varnames  0 : { *(.debug_varnames) }

}
```

更多的内容可以参考
Using_ld,the_GNU_Linker:
http://www.redhat.com/docs/manuals/enterprise/RHEL-3-Manual/pdf/rhel-ld-en.pdf

Team MCUZONE
www.mcuzone.com