施家琪

# 基本概念

链接器会将许多的输入文件链接成一个输出文件。输出文件和输入文件的格式都是所谓的object文件格式。输出文件经常被称为“可执行”文件，但是为了便于讲解，我们还是称其为object文件。每个object文件都由许多段组成。我们经常把输入文件中的段称为输入段（input section），同样将输出文件中的段称为输出段（output section）。

在object文件中所有的段都有名称和大小。大多数段还包括相关的数据块，这个数据块被称为段内容（section contents）。一个段可能会被标记为可以加载的（loadable），这就意味着当输出文件运行时这个段应该被加载到内存。如果一个段没有内容，那么这个段可能是可分配的（allocatable），这就意味着应该将相应的内存空间保留下来，但是这些空间里面并没有保存什么有效的数据。如果一个段都不是上面所述的两个的话，这个段一般用来包含一些调试信息。

所有的可加载或可分配的输出段都包含两个地址。第一个地址称为虚拟内存地址（VMA）。这个地址是输出文件运行时所位于的地址。另一个地址是加载地址（LMA）。这个地址指出了输出文件将被加载的地址。在大多数情况下这两个地址是相同的。一个不相同的例子是，当一个段被加载在ROM中，然后又被拷贝到RAM中去运行。在这种情况下ROM地址应该是加载地址。而RAM地址就应该是虚拟地址（或称链接地址）。

每个object文件还包括一串符号，这些符号串被称为符号表（symbol table）。一个符号可能是定义了或是没有定义。每个符号都有名称并且所有被定义的符号都有一个地址。如果我们将C或C++程序编译进一个object文件，我们将会得到所有定义函数，全局变量以及静态变量的符号，这些符号都是被定义了的。

# 链接脚本格式及一个简单示例

最简单的链接脚本只有一条命令“SECTIONS”。使用”SECTIONS”命令用来描述输出文件的内存分布（memory layout）。

我们来假设一个程序。这个程序中包括数据段，初始化了的数据段，以及未初始化的数据段。这些段应该叫做“.text”，“.data”，“.bss”。我们再假设这个程序中只有这三个段。

我们说现在这个程序将被加载到0x10000地址上。然后数据必须从0x8000000。以下就是我们的第一个链接脚本。

|  |
| --- |
| SECTIONS  {  . = 0x10000;  .text : { \*(.text) }  . = 0x8000000;  .data : { \*(.text)}  .bss : {\*(.bss)}  } |

SECTIONS命令中第一行的‘.’符号的意思是位置指针（location counter）。这一行的意思是指定位置指针处的地址为0x10000。如果你不用其他的方式（之后将讲这些其他的方式）来定义输出段的地址。那么这个地址将被指定为这个位置指针的值。这个位置指针之后会根据输出段的大小而自动增加。在“SECTIONS”命令的开头，这个指针将被初始化为0。

第二行定义了一个输出段“.text”。在段名称冒号后面的花括号中指明了哪些段将属于这个输出段中。‘\*’是一个通配符，它代表了所有的文件名称。表达式‘\*(.text)’的意思是所有输入文件中的‘.text’段。

在这里由于.text段上面的位置指针被赋值为0x10000，所以.text的地址就是0x10000。之后的.data和.bss和上面就一样了。这里不再赘述。

# 简单编译脚本命令

## 设置入口（Entry Point）

程序被装载后运行的第一条指令被称为程序入口。我们可以使用ENTRY命令来设置程序入口。这个命令的格式是：ENTRY(symbol)。

有很多种方式能设置程序的入口。链接器会依次用下面这些方式来设置入口，如果成功则停止：

1. 链接器中使用‘-e’参数来指明一个符号作为入口地址。
2. 在链接脚本中使用ENTRY(symbol)命令来标识入口地址。
3. 体系相关定义的符号，比如start。
4. .text段的一个字节的地址。
5. 0地址

## 与文件相关的命令

1. INCLUDE filename

在当前位置再加载一个链接脚本，这个链接脚本将会在当前工作目录进行寻找。如果指定了-L选项，那么将在指定地址中进行寻找。

1. INPUT(file, file, …)

INPUT命令让链接器在链接的过程中导入参数中的文件。比如说，如果你想在链接时包含’subr.o’文件，又不愿意每次敲命令行的时候指定。就可以使用INPUT(subr.o)来完成。如果使用INPUT(-lfile),连接器将会将文件名认定为是库文件。这时会加载相应的库文件。

1. OUTPUT(filename)

OUTPUT命令同参数“-o filename”，将指定输出文件的名称。

1. SEARCH\_DIR(path)

SEARCH\_DIR命令同命令行参数“-L path”，将为链接器指定加载库的搜索地址。

## 处理目标文件格式命令

## 内存相关命令

可以将对齐名称加入到已有的内存区域中去。每个名字最多关联一个内存区域。

REGION\_ALIAS(alias, region)

这条命令将创建一个对齐名称alias对应region这个内存区域。这提供了一种弹性的将输出段映射到内存区域的方式。比如：

假设我们编写一个面向嵌入式系统的程序。这个嵌入式系统可能有多种存储设备。所有这些都包括一个通用，易失的内存RAM。有些可能包括只读，不易失的ROM来运行指令执行或只读数据访问。还有一种是不能进行指令执行的ROM2.我们有四个输出段：

1. .text program code;
2. .rodata read-only data;
3. .data read-write initialized data;
4. .bss read-write zero initialized data;

我们的目标是编写一个链接脚本，这个脚本中包括了这个系统独立的部分以及不独立的部分。我们的嵌入式系统有三种不同的内存配置。

A.

.text: RAM

.rodata: RAM

.data:RAM

.bss:RAM

B.

.text:ROM

.rodata:ROM

.data:RAM/ROM

.bss:RAM

C.

.text:ROM

.rodata:ROM2

.data:RAM/ROM2

.bss:RAM

我们的链接脚本通过在家系统的linkcmds.memory文件来分辨配置。下面是我们的链接脚本：

|  |
| --- |
| INCLUDE linkcmds.memory  {  .text:  {  \*(.text)  } > REGION\_TEXT  .rodata:  {  \*(.rodata)  rodata\_end = .;  } > REGION\_RODATA  .data : AT(rodata\_end)  {  data\_start = .;  \*(.data)  } > REGION\_DATA  data\_size = SIZEOF(.data);  data\_load\_start = LOADADDR(.data);  .bss:  {  \*(.bss)  } > REGION\_BSS  } |

现在我们需要定义三个不同的linkcmds.memory：

A：

|  |
| --- |
| MEMORY  {  RAM : ORIGIN = 0, LENGTH = 4M  }  REGION\_ALIAS(“REGION\_TEXT”,RAM);  REGION\_ALIAS(“REGION\_RODATA”,RAM);  REGION\_ALIAS(“REGION\_DATA”,RAM);  REGION\_ALIAS(“REGION\_BSS”,RAM); |

B：

|  |
| --- |
| MEMORY  {  ROM: ORIGIN = 0, LENGTH = 3M  RAM: ORIGIN = 0x100000000， LENGTH = 1M  }  REGION\_ALIAS(“REGION\_TEXT”, ROM);  REGION\_ALIAS(“REGION\_RODATA”, ROM);  REGION\_ALIAS(“REGION\_DATA”, RAM);  REGION\_ALIAS(“REGION\_BSS”, RAM); |

C:

|  |
| --- |
| MEMORY  {  ROM: ORIGIN = 0, LENGTH = 2M  ROM2: ORIGIN = 0x100000000， LENGTH = 1M  RAM:ORIGIN=0x200000000 , LENGTH = 1M  }  REGION\_ALIAS(“REGION\_TEXT”, ROM);  REGION\_ALIAS(“REGION\_RODATA”, ROM2);  REGION\_ALIAS(“REGION\_DATA”, RAM);  REGION\_ALIAS(“REGION\_BSS”, RAM); |

这样的好处在于，我们可以写一个通用的程序来将ROM的内容拷贝到RAM中进行运行：

|  |
| --- |
| #include <string.h>  extern char data\_start[];  extern char data\_size [];  extern char data\_load\_start[];  void copy\_data()  {  If (data\_start != data\_load\_start)  {  Memcpy(data\_start, data\_load\_start, (size\_t) data\_size);  }  } |

# 4. 赋值

我们可以对链接脚本中的符号进行赋值，这样会定义该符号，并把符号放入符号表。这些符号是可以全局访问。

## 4.1 简单赋值

简单赋值的方式类似于C语言的赋值语句。

# 5. SECTIONS命令

SECTIONS命令告诉连接器如何将输入段安排到输出段中。

SECTIONS命令的格式如下：

|  |
| --- |
| SECTIONS  {  sections-command  sections-command  …  } |

每个sections-command可能会是下面的一种：

1. ENTRY命令

2. 符号赋值表达式

3. 输出段描述符

4. overlay描述符

## 5.1. 输出段描述符

完整的输出段描述符应该像下面这个样子：

|  |
| --- |
| section [address] [(type)] :  [AT(lma)]  [ALIGN(section\_align)]  [SUBALIGN(subsection\_align)]  [constraint]  {  output-section-command  output-section-command  } [>region] [AT>lma\_region][:phdr :phdr …][=fillexp] |

一般的输出段描述符不会包括上面所有的内容。

注意，在section周围的空格是必须的。冒号和花括号也是必须的。但是换行和其他空格则是可选的。

所有的output-sections-command可能包括以下这些：

1. 符号赋值表达式

2. 输入段描述符

3. 直接导入的数据

4. 特殊的输出段关键字

## 5.2. 输出段名称

## 5.3. 输出段地址

输出段地址是一个VMA表达式。如果不指明地址，连接器将根据上文自动指明地址。

比如：

.text . : { \*(.text) }

和

.text : { \*(.text) }

是不同的。第一个设置.text段到当前计数指针的地址。第二个也会设置到当前计数指针地址，但会根据输入段的.text对其强制对齐。

地址的表达式可以是任意的。比如，如果我们希望将段根据0x10对其，也就是说低4位全是0，我们可以这样：

.text ALIGN(0x10) : { \*(.text) }

## 5.4 输入段

我们通过制定输入段描述符，来告诉链接器如何将输入段组织成输出段。输入段的名字可以是通配符例如：

\*(.text)

这代表的是所有输入文件的.text段。如果想要在通配符中去掉某些文件的.text段，可以采用EXCLUDE\_FILE，就像下面这样：

\*(EXCLUDE\_FILE( \*crtend.o \*otherfile.o) .ctors)

这样会将crtend.o和otherfile.o排除到ctors段中。

你可以制定某个文件的某个段，就像下面这样：

data.o(.data)

这将制定data.o的data段。更多关于通配符的信息，请查阅link手册。

## 5.5 Comman符号

在多数object文件格式中common符号没有特定的输入段。链接器对common的处理是将其看作一个叫COMMON的输入段。

大多数情况下，common符号将会放入.bss段中，比如：

.bss { \*(.bss) \*(COMMON)}

有些object文件格式包括不止一种的common符号。比如MIPS ELF目标文件就有标准common符号和小common符号两种。在MIPS EFL中，链接器将’COMMON’指定为标准common符号而’.scommon’指定为小common符号。

## 5.6 输入段描述符例子

下面将展示一个完整的例子。这个链接脚本将告诉链接器将all.o的所有段放置在outputa的最开头，而outputa位于地址0x10000。foo.o文件的input1段放在其后。在outputb输出段中，将foo.o的input2段放在开头，将foo1.o的input1段放在其后。在outputc输出段中。将所有文件的input1放在开头。然后将所有文件的input2放在其后。

|  |
| --- |
| SECTIONS{  outputa 0x10000 :  {  all.o  foo.o(.input1)  }  outputb  {  foo.o(.input2)  foo1.o(.input1)  }  Output  {  \*(.input1)  \*(.input2)  }  } |

## 5.7 输出端描述符

一个完整的输出段如下：

|  |
| --- |
| section [address] [(type)] :  [AT(lma)]  [ALIGN(section\_align)]  [SUBALIGN(subsection\_align)]  [constraint]  {  output-section-command  output-section-command  } [>region] [AT>lma\_region][:phdr :phdr …][=fillexp] |

每一个输出段都包括一个type属性，预设了如下属性：

1. NOLOAD 标志为not loadable，当程序运行时该段将不会被加载到硬盘。

2. DSECT

3. INFO

4. OVERLAY

每个段都包括一个虚拟地址（链接地址）和加载地址。AT关键字加上表达式lma将设定该段的加载地址。或者也可以采用’AT>lma\_region，我们可以为加载地址设定一个内存区域。这个例子上面已经有了。

如果既没有设置AT也没有设置AT>的话，链接器将会将虚拟地址和加载地址设置成一样。

举个例子，接下来的一个链接脚本将链接三个输出段，一个叫做‘.text’，这个段从0x1000开始。另一个叫做’.mdata’，这个段加载到.text后面，但是其虚拟地址为0x2000。另一个叫做’.bss’，这个段位于0x3000。符号\_data被定义为0x2000，这个符号用来保存当前虚拟地址。

|  |
| --- |
| SECTIONS  {  .text 0x1000 : { \*(.text) \_text = .;}  .mdata 0x2000 :  AT ( ADDR (.text) + SIZEOF( .text) )  { \_data = .; \*(.data); \_edata = .; }  .bss 0x3000 :  { \_bstart = . ; \*(.bss) \*(COMMON); \_bend = .;}  } |

# 6. 内存命令

MEMORY命令描述了目标内存区域的大小和位置。你可以用它来定义哪些内存区域将被用于链接。

每个链接脚本至少有一个MEMORY命令，下面是其语法：

|  |
| --- |
| MEMORY  {  Name [(attr)] : ORIGIN = origin, LENGTH = len  …  } |

内存区域的名称对外部来说没有意义，也不会和链接脚本或程序中的符号发生冲突。每个内存区域必须在MEMORY命令中包括一个明确的名称。attr字符串是一个可选的特性。这个特性描述了这段内存的特点。下面举个例子就知道了：

|  |
| --- |
| MEMORY  {  rom (rx) : ORIGIN = 0, LENGTH = 256K  ram (!rx): ORIGIN = 0x40000000, l = 4M  } |

l和LENGTH是同样的意思，attr特性很像Linux中的文件权限，’!’的意思是反转当前特性。上面例子中，除了rx外的所有内存都放置在名为ram的内存区域当中。

# 7. 链接脚本中的表达式

链接脚本的表达式语法和C语言的并无两样。

## 7.1 常量

和C语言一样，链接器将以‘0’开头的数字认为是八进制。将‘0x’或‘0X’开头的数字认为是十六进制。除此之外，可以用K代表1024，可以用M代表1024^2。

## 7.2 符号常量

通过CONSTANT(name)命令可以获取一些符号常量：

MAXPAGESIZE

目标的页的最大尺寸。

COMMONPAGESIZE

目标的默认页尺寸

用法如下：

.text ALIGN (CONSTANT (MAXPAGESIZE)) : { \*(.text)}

# 8. 位置指针（Location Counter）

特殊的链接符号’.’总是代表当前位置指针。将值赋给这个符号，将导致当前位置指针的移动。这样会让输出段形成一个空洞。位置指针不允许往回赋值。下面是个小例子。

|  |
| --- |
| SECTIONS  {  output :  {  file1(.text)  . = . + 1000;  File2(.text)  . += 1000;  File3(.text)  } = 0x12345678;  } |

在这个例子中file1的.text段之后有个1000字节大小的空白。整个output输出段的最后的那个=0x12345678表示这些空白的填充值。

# 9. 内置函数

链接脚本包括一些内置函数来方便编写，这些内置函数常用到的包括：

1. ADDR(section)：返回section的虚拟地址（链接地址）。

2. ALIGN(align)：对齐。例如

|  |
| --- |
| SECTIONS{  .data ALIGN(0x2000) : {  \*(.data)  Variable = ALIGN(0x8000)  }  …  } |

上述例子的第一个ALIGN将.data段的位置对0x2000对齐。而第二个ALIGN的用来将Variable赋值。

3. LENGTH(memory)：返回内存区域的长度。

4. LOADADDR(section)：返回section的绝对加载地址。

5. SIZEOF(section)：返回section的以字节为单位的大小。