**3.1 Basic Linker Script Concepts**

We need to define some basic concepts and vocabulary in order to describe the linker script language.

为了描述链接脚本语言,我们需要去定义一些基本的概念和词汇。

The linker combines input files into a single output file.

连接器将所有的输入文件合并为单一的输出文件。

The output file and each input file are in a special data format known as an *object file format.*

输出文件和每个输入文件都采用一种特殊的数据格式，我们称之为目标文件格式。

Each file is called an *object file.*

每一个文件都叫目标文件。

The output file is often called an *executable*, but for our purposes we will also call it an object file.

输出文件通常称为可执行文件，但出于我们的目的，我们也将其称为目标文件。

Each object file has, among other things, a list of sections.

每一个目标文件，除了其他东西以外，还有一系列的段。

We sometimes refer to a section in an input file as an *input section*; similarly, a section in the output file is an *output section.*

我们称输入文件中的段是输入段，同样的，输出文件的段称为输出段。

Each section in an object file has a name and a size.

目标文件中的每一个段都有名字和大小。

Most sections also have an associated block of data, known as the *section contents*.

许多段有一块与之相关的数据，称为节内容section contents。

A section may be marked as*loadable*, which means that the contents should be loaded into memory when the output file is run.

当输出文件在运行时，它的节内容（section contents）应该被加载进内存，那么这个段被称为可加载的。

A section with no contents may be *allocatable*, which means that an area in memory should be set aside, but nothing in particular should be loaded there (in some cases this memory must be zeroed out).

当一个段没有节内容*section contents，*那么这个段可能是可以分配的，意味着应该在内存中空出一块区域，但是什么都不应该加载到这块区域。（这块区域应该被清零）

A section which is neither loadable nor allocatable typically contains some sort of debugging information.

当一个段包含一些调试信息时，这个段既不是可加载的也不是可分配的。

Every loadable or allocatable output section has two addresses.

每一个加载和可分配的输出段都有两个地址。

The first is the *VMA*, or virtual memory address. This is the address the section will have when the output file is run.

这第一个地址叫虚拟内存地址，当这个输出文件运行时，这个段拥有的地址就是虚拟内存地址。

The second is the *LMA*, or load memory address. This is the address at which the section will be loaded.

这第二个地址叫加载内存地址，这个地址是这个段在加载的时候有的。

In most cases the two addresses will be the same. An example of when they might be different is when a data section is loaded into ROM, and then copied into RAM when the program starts up (this technique is often used to initialize global variables in a ROM based system).

在很多情况下，这两个地址是相同的。它们可能不同的一个例子是，数据部分加载到ROM中，然后启动代码复制到RAM中。（这项技术经常用于去初始化在ROM中的全局变量）

In this case the ROM address would be the LMA, and the RAM address would be the VMA.

在这种情况下，ROM中的地址是LMA，RAM中的地址是VMA。

You can see the sections in an object file by using the objdump program with the ‘-h’ option.

你可以用objdump程序中的-h选项来查看目标文件中的段。

Every object file also has a list of *symbols*, known as the *symbol table*. A symbol may be defined or undefined. Each symbol has a name, and each defined symbol has an address, among other information.

每一个目标文件都有一系列的符号，称为符号表。一个符号可能是定义的，也可能是未定义的。除了其他信息，每一个符号都有名字并且每一个定义的符号有地址。

If you compile a C or C++ program into an object file, you will get a defined symbol for every defined function and global or static variable.

如果你编译C/C++程序为目标文件，你将得到每一个定义的函数、全局变量和静态变量的的符号。

Every undefined function or global variable which is referenced in the input file will become an undefined symbol.

在输入文件中引用的每一个未定义的函数或全局变量都将成为未定义的符号。

You can see the symbols in an object file by using the nm program, or by using the objdump program with the ‘-t’ option.

你可以通过程序nm来查看目标文件中的符号。或者程序objdump 通过-t选项来查看。

**3.3 Simple Linker Script Example**

Many linker scripts are fairly simple.

许多链接脚本是非常简单的。

The simplest possible linker script has just one command: ‘SECTIONS’.

可能最简单的链接脚本就仅仅有一个命令’SECTION’。

You use the ‘SECTIONS’ command to describe the memory layout of the output file.

你可以用‘SECTIONS’命令去描述输出文件的内存布局。

The ‘SECTIONS’ command is a powerful command.

‘SECTIONS’命令是一个重要的命令。

Here we will describe a simple use of it.

这儿我们会描述一下他的简单用法。

Let’s assume your program consists only of code, initialized data, and uninitialized data.

我们假设你的程序只包含代码、初始化的数据和未初始化的数据。

These will be in the ‘.text’, ‘.data’, and ‘.bss’ sections, respectively.

他们将分别位于‘.text’, ‘.data’, 和 ‘.bss’ 段。

Let’s assume further that these are the only sections which appear in your input files.

让我们进一步假设他们是你输出文件中唯一的段。

For this example, let’s say that the code should be loaded at address 0x10000, and that the data should start at address 0x8000000. Here is a linker script which will do that:

举个例子，我们让代码段位于地址）0X10000，然后数据起始于地址0x8000000。这个简单的链接脚本如下：

SECTIONS

{

. = 0x10000;

.text : { \*(.text) }

. = 0x8000000;

.data : { \*(.data) }

.bss : { \*(.bss) }

}

You write the ‘SECTIONS’ command as the keyword ‘SECTIONS’, followed by a series of symbol assignments and output section descriptions enclosed in curly braces.

你可以用关键字‘SECTIONS’用作‘SECTIONS’命令，‘SECTIONS’命令包含一系列的符号赋值并且输出段用一对打的花括弧来描述。

The first line inside the ‘SECTIONS’ command of the above example sets the value of the special symbol ‘.’, which is the location counter.

在‘上面这个例子中，‘SECTIONS’命令中第一行设置了特殊符号’.’的值，’.’称为位置计数器。

If you do not specify the address of an output section in some other way (other ways are described later), the address is set from the current value of the location counter.

The location counter is then incremented by the size of the output section. At the start of the ‘SECTIONS’ command, the location counter has the value ‘0’.

The second line defines an output section, ‘.text’. The colon is required syntax which may be ignored for now. Within the curly braces after the output section name, you list the names of the input sections which should be placed into this output section. The ‘\*’ is a wildcard which matches any file name. The expression ‘\*(.text)’ means all ‘.text’ input sections in all input files.

Since the location counter is ‘0x10000’ when the output section ‘.text’ is defined, the linker will set the address of the ‘.text’ section in the output file to be ‘0x10000’.

The remaining lines define the ‘.data’ and ‘.bss’ sections in the output file. The linker will place the ‘.data’ output section at address ‘0x8000000’. After the linker places the ‘.data’ output section, the value of the location counter will be ‘0x8000000’ plus the size of the ‘.data’ output section. The effect is that the linker will place the ‘.bss’ output section immediately after the ‘.data’ output section in memory.

The linker will ensure that each output section has the required alignment, by increasing the location counter if necessary. In this example, the specified addresses for the ‘.text’ and ‘.data’ sections will probably satisfy any alignment constraints, but the linker may have to create a small gap between the ‘.data’ and ‘.bss’ sections.

That’s it! That’s a simple and complete linker script.

3.5 Assigning Values to Symbols

You may assign a value to a symbol in a linker script. This will define the symbol and place it into the symbol table with a global scope.

|  |  |  |
| --- | --- | --- |
| • [Simple Assignments](https://sourceware.org/binutils/docs/ld/Simple-Assignments.html#Simple-Assignments): |  | Simple Assignments |
| • [HIDDEN](https://sourceware.org/binutils/docs/ld/HIDDEN.html#HIDDEN): |  | HIDDEN |
| • [PROVIDE](https://sourceware.org/binutils/docs/ld/PROVIDE.html#PROVIDE): |  | PROVIDE |
| • [PROVIDE\_HIDDEN](https://sourceware.org/binutils/docs/ld/PROVIDE_005fHIDDEN.html#PROVIDE_005fHIDDEN): |  | PROVIDE\_HIDDEN |
| • [Source Code Reference](https://sourceware.org/binutils/docs/ld/Source-Code-Reference.html#Source-Code-Reference): |  | How to use a linker script defined symbol in source code |

**3.5.1 Simple Assignments**

You may assign to a symbol using any of the C assignment operators:

你可以用任何一种C语言的赋值操作给符号赋值。

*symbol* = *expression* ;

*symbol* += *expression* ;

*symbol* -= *expression* ;

*symbol* \*= *expression* ;

*symbol* |= *expression* ;

*symbol* <<= *expression* ;

*symbol* >>= *expression* ;

*symbol* &= *expression* ;

*symbol* |= *expression* ;

The first case will define *symbol* to the value of *expression*.

这第一种情况定义了符号的值为表达式的值。

In the other cases, *symbol* must already be defined, and the value will be adjusted accordingly.

在其他的情况下，符号必须是已经定义了的，这个符号的值也会被改正。

The special symbol name ‘.’ indicates the location counter.

‘.’是个特殊的符号，称为地址计数器。

You may only use this within a SECTIONS command.

你可能只会在SECTIONS命令中使用它。

See Location Counter

更多的详细内容可以看Location Counter章节。

The semicolon after *expression* is required.

表达式后面的逗号是必须的。

Expressions are defined below; see Expressions.

表达式定义如下; 详细内容可以看 Expressions章节。。

You may write symbol assignments as commands in their own right, or as statements within a SECTIONS command, or as part of an output section description in a SECTIONScommand.

你可以把赋值符号写作一个命令，或者写为SECTIONS命令里的表达式，或者作为描述输出段的SECTIONS命令中的一部分。

The section of the symbol will be set from the section of the expression;

符号的部分将从表达式的部分设置

for more information, see Expression Section.

Here is an example showing the three different places that symbol assignments may be used:

这儿有一个例子显示了赋值符号的三种可能用到的地方。

floating\_point = 0;

SECTIONS

{

.text :

{

\*(.text)

\_etext = .;

}

\_bdata = (. + 3) & ~ 3;

.data : { \*(.data) }

}

In this example, the symbol ‘floating\_point’ will be defined as zero. The symbol ‘\_etext’ will be defined as the address following the last ‘.text’ input section. The symbol ‘\_bdata’ will be defined as the address following the ‘.text’ output section aligned upward to a 4 byte boundary.

**3.5.2 HIDDEN**

For ELF targeted ports, define a symbol that will be hidden and won’t be exported. The syntax is HIDDEN(*symbol* = *expression*).

对于ELF目标端口，定义了一个隐藏但不会被导出的符号。这个语法是HIDDEN(*symbol* = *expression*).

Here is the example from [Simple Assignments](https://sourceware.org/binutils/docs/ld/Simple-Assignments.html#Simple-Assignments), rewritten to use HIDDEN:

我们可以使用HIDDEN语法重新写上面的这个简单例子

HIDDEN(floating\_point = 0);

SECTIONS

{

.text :

{

\*(.text)

HIDDEN(\_etext = .);

}

HIDDEN(\_bdata = (. + 3) & ~ 3);

.data : { \*(.data) }

}

In this case none of the three symbols will be visible outside this module.

在这个例子中在这个模块之外三个符号都是不可见的。

**3.5.3 PROVIDE**

In some cases, it is desirable for a linker script to define a symbol only if it is referenced and is not defined by any object included in the link. For example, traditional linkers defined the symbol ‘etext’. However, ANSI C requires that the user be able to use ‘etext’ as a function name without encountering an error. The PROVIDE keyword may be used to define a symbol, such as ‘etext’, only if it is referenced but not defined. The syntax is PROVIDE(*symbol* = *expression*).

Here is an example of using PROVIDE to define ‘etext’:

SECTIONS

{

.text :

{

\*(.text)

\_etext = .;

PROVIDE(etext = .);

}

}

In this example, if the program defines ‘\_etext’ (with a leading underscore), the linker will give a multiple definition error. If, on the other hand, the program defines ‘etext’ (with no leading underscore), the linker will silently use the definition in the program. If the program references ‘etext’ but does not define it, the linker will use the definition in the linker script.

cambridge