NASM 中文手册

第一章: 简介

1.1 什么是 NASM

NASM 是一个为可移植性与模块化而设计的一个 80x86 的汇编器。它支持相当多的目标文件格式，包括 Linux 和'NetBSD/FreeBSD','a.out','ELF','COFF',微软 16 位的'OBJ'和'Win32'。它还可以输出纯二进制文件。它的语法设计得相当的简洁易懂，和 Intel 语法相似但更简单。它支持'Pentium','P6','MMX','3DNow!','SSE' and 'SSE2'指令集，

1.1.1 为什么还需要一个汇编器？

NASM 当初被设计出来的想法是'comp.lang.asm.x86'(或者可能是'alt.lang.asm',我忘了),从本质上讲，是因为没有一个好的免费的 x86 系例的汇编器可以使用，所以，必须有人来写一个。

(*)'a86'不错，但不是免费的，而且你不可能得到 32 位代码编写的功能，除非你付费，它只使用在 dos 上。

(*) 'gas'是免费的，而且在 dos 下和 unix 下都可以使用，但是它是作为'gcc'的一个后台而设计的，并不是很好，'gcc'一直就提供给它绝对正确的代码，所以它的错误检测功能相当弱，还有就是对于任何一个想真正利用它写点东西的人来讲，它的语法简直太可怕了，并且你无法在里面写正确的 16 位代码。

(*) 'as86'是专门为 Minix 和 Linux 设计的，但看上去并没有很多文档可以参考。

(*) 'MASM'不是很好，并且相当贵，还且只能运行在 DOS 下。

(*) 'TASM'好一些，但却极入与 MASM 保持兼容，这就意味着无数的伪操作码和繁琐的约定，并且它的语法本质上就是 MASM 的，伴随着的就是一些自相矛盾和奇怪的东西。它也是相当贵的，并且只能运行在 DOS 下。

所以，只有 NASM 才能使您愉悦得编程。目前，它仍在原型设计阶段-我们不期望它能够超越所有的这些汇编器。但请您发给我们 bug 报告，修正意见，和其他有用的信息，还有其他任何你手头有的对我们有用的信息(感谢所有已经这样在做了的人们)，我们还会不断地改进它。

1.1.2 许可条件

请阅读作为 NASM 发布的一部分的文件'Licence'，只有在该许可条件下你才可以使用 NASM。

1.2 联系信息

当前版本的NASM(0.98.08)由一个开发小组在维护，你可以从'nasm-devel'邮件列表中得到(看下面的链接)，如果你想要报告bug，请先阅读10.2节

NASM有一个主页:'http://www.web-sites.co.uk/nasm'，更多的信息还可以在`http://nasm.2y.net/'上获取。

最初的作者你可以通过email:`jules@dsf.org.uk'和`anakin@pobox.com'和他们联系，但后来的开发小组并不在其中。

最新的NASM发布被上传至官方网站`http://www.web-sites.co.uk/nasm'和`ftp.kernel.org',
`ibiblio.org'

公告被发布至`comp.lang.asm.x86', `alt.lang.asm' 和`comp.os.linux.announce'

如果你想了解NASM beta版的发布，和当前的开发状态，请通过在
`http://groups.yahoo.com/group/nasm-devel',
`http://www.pairlist.net/mailman/listinfo/nasm-devel' and
`http://sourceforge.net/projects/nasm'
注册来捐助'nasm-devel'邮件列表。

在网站Sourceforge上的列表是较好的一个列表，它也是最新nasm源代码与发布的一个网站，另外的列表也是公开的，但有可能不会被继续长期支持。

1.3 安装

1.3.1 在dos和Windows下安装NASM

如果你拿到了NASM的DOS安装包,'nasmXXX.zip'(这里.'XXX'表示该安装包的NASM版本号)，把它解压到它自己的目录下(比如：‘c:\nasm’)

该包中会包含有四个可执行文件：NASM可拟行文件'nasm.exe'和'nasmw.exe',还有NDISASM可执行文件'ndisasm.exe'和'ndisasmw.exe'。文件名以'w'结尾的是'Win32'可执行格式。是运行在'Windows 95'或'Windows NT'的Intel处理器上的，另外的是16位的'DOS'可执行文件。

NASM运行时需要的唯一文件就是它自己的可执行文件，所以可以拷贝'nasm.exe'和'nasmw.exe'的其中一个到你自己的路径下，或者可以编写一个'autoexec.bat'把nasm的路径加到你的'PATH'环境变量中去。(如果你只安装了Win32版本的，你可能希望把文件名改成'nasm.exe'。)

就这样，NASM 装好了。你不需要为了运行 nasm 而让'nasm'目录一直存在(除非你把它加到了你的'PATH'中,所以如果你需要节省空间，你可删掉它,但是，你可能需要保留文档或测试程序。

如果你下载了 DOS 版的源码包，'nasmXXXs.zip',那'nasm'目录还会包含完整的 NASM 源代码，你可以选择一个 Makefiles 来重新构造你的 NASM 版本。

注意源文件`insnsa.c', `insnsd.c', `insnsi.h'和`insnsn.c'是由'standard.mac'中的指令自动生成的，尽管 NASM0.98 发布版中包含了这些产生的文件，你如果改动了 insns.dat,standard.mac 或者文件，可能需要重新构造他们，在将来的源码发布中有可能将不再包含这些文件，多平台兼容的 Perl 可以从 www.cpan.org 上得到。

1.3.2 在 unix 下安装 NASM

如果你得到了 Unix 下的 NASM 源码包'nasm-x.xx.tar.gz'(这里 x.xx 表示该源码包中的 nasm 的版本号),把它解压压到一个目录，比如'/usr/local/src'。包被解压后会创建自己的子目录'nasm-x.xx'

NASM 是一个自动配置的安装包:一旦你解压了它，'cd'到它的目录下，输入'./configuer',该脚本会找到最好的 C 编译器来构造 NASM，并据此建立 Makefiles。

一旦 NASM 被自动配置好后，你可以输入'make'来构造'nasm'和'ndisasm'二进制文件，然后输入'make install'把它们安装到'/usr/local/bin',并把 man 页安装到'/usr/local/man/man1'下的'nasm.1 和'ndisasm.1'或者你可以给配置脚本一个'--prefix'选项来指定安装目录，或者也可以自己来安装。

NASM 还附带一套处理'RDOFF'目标文件格式的实用程序，它们在'rdoff'子目录下，你可以用'make rdf'来构造它们，并使用'make rdf_install'来安装。如果你需要的话。

如果 NASM 在自动配置的时候失败了，你还是可以使用文件'Makefile.unx'来编译它们，把这个文件改名为'Makefile',然后输入'make'。在'rdoff'子目录下同样有一个 Makefile.unx 文件。

第二章 运行 NASM

2.1 NASM 命令行语法

要汇编一个文件，你可以以下面的格式执行一个命令：

```
nasm -f <format> <filename> [-o <output>]
```

比如,

```
nasm -f elf myfile.asm
```

会把文件'myfile.asm'汇编成'ELF'格式 的文件'myfile.o'.还有：

```
nasm -f bin myfile.asm -o myfile.com
```

会把文件'myfile.asm'汇编成纯二进制格式的文件'myfile.com'。

想要以十六进制代码的形式产生列表文件输出，并让代码显示在源代码的左侧，
使用'-l'选项并给出列表文件名，比如：

```
nasm -f coff myfile.asm -l myfile.lst
```

想要获取更多的关于 NASM 的使用信息，请输入：

```
nasm -h
```

它同时还会输出可以使用的输出文件格式，
如果你使用 Linux 并且不清楚你的系统是'a.out'还是'ELF'，请输入：

```
file nasm
```

(在 nasm 二进制文件的安装目录下使用),如果系统输出类似下面的信息：

```
nasm: ELF 32-bit LSB executable i386 (386 and up) Version 1
```

那么你的系统就是'ELF'格式的，然后你就应该在产生 Linux 目标文件时使用选
项'-f elf',如果系统输入类似下面的信息：

```
nasm: Linux/i386 demand-paged executable (QMAGIC)
```

或者与此相似的，你的系统是'a.out'的，那你应该使用'-f aout'(Linux 的'a.out'
系统很久以前就过时了，现在已非常少见。)

就像其他的Unix编译器与汇编器，NASM在碰到错误以前是不输出任何信息的，所以除了出错信息你看不到任何其他信息。

2.1.1 '-o'选项：指定输出文件的文件名。

NASM会为你的输出文件选择一个文件名；具体如何做取决于目标文件的格式，对于微软的目标文件格式('obj'和'win32'),它会去掉你的源文件名的'.asm'扩展名(或者其他任何你喜欢使用的扩展名，NASM并不关心具体是什么)，并替换上'obj'。对于Unix的目标文件格式('aout','coff','elf'和'as86')它会替换成'.o', 对于'rdf',它会使用'.rdf',还有为'bin'格式，它会简单地去掉扩展名，所以'myfile.asm'会产生的一个输出文件'myfile'。

如果输出文件已经存在，NASM会覆盖它，除非它的文件名与输入文件同名，在这种情况下，它会给出一个警告信息，并使用'nasm.out'作为输出文件的文件名。

在某些情况下，上述行为是不能接受的，所以，NASM提供了'-o'选项，它能让你指定你的输出文件的文件名，你使用'-o'后面紧跟你为输出文件取的名字，中间可以加空格也可以不加。比如：

```
nasm -f bin program.asm -o program.com
nasm -f bin driver.asm -odriver.sys
```

请注意这是一个小写的o，跟大写字母O是不同的，大写的是用来指定需要传递的选项的数目，请参阅2.1.15

2.1.2 `-f'选项：指定输出文件的格式。

如果你没有对NASM使用'-f'选项，它会自己为你选择一个输出文件格式。在发布的NASM版本中，缺省的输出格式总是'bin';如果你自己编译你的NASM,你可以在编译的时候重定义'OF_DEFAULT'来选择你需要的缺省格式。

就象'-o','-f'与输出文件格式之间的空格也是可选的，所以'-f elf'和'-felf'都是合法的。

所有可使用的输出文件格式的列表可以通过运行命令'nasm -hf'得到。

2.1.3 `-l' 选项: 产生列表文件

如果你对NASM使用了'-l'选项，后面跟一个文件名，NASM会为你产生一个源文件的列表文件,在里面，地址和产生的代码列在左边，实际的源代码(包括宏扩展，除了那些指定不需要在列表中扩展的宏，参阅4.3.9)列在右边，比如：

```
nasm -f elf myfile.asm -l myfile.lst
```

2.1.4 `-M'选项: 产生 Makefile 依赖关系.

该选项可以用来向标准输出产生 makefile 依赖关系，可以把这些信息重定向到一个文件中以待进一步处理，比如：

NASM -M myfile.asm > myfile.dep

2.1.5 `-F'选项: 选择一个调试格式

该选项可以用来为输出文件选择一个调试格式，语法跟-f 选项相册，唯一不同的是它产生的输出文件是调试格式的。

一个具体文件格式的完整的可使用调试文件格式的列表可通过命令'nasm -f <format> -y'来得到。

这个选项在缺省状态下没有被构建时 NASM。如何使用该选项的信息请参阅 6.10

2.1.6 `-g' 选项:使调试信息有效。

该选项可用来在指定格式的输出文件中产生调试信息。
更多的信息请参阅 2.1.5

2.1.7 `-E' 选项: 把错误信息输入到文件。

在'MS-DOS'下，尽管有办法，但要把程序的标准错误输出重定向到一个文件还是非常困难的。因为 NASM 常把它的警告和错误信息输出到标准错误设备，这将导致你在文本编辑器里面很难捕捉到它们。

因此 NASM 提供了一个'-E'选项，带有一个文件名参数，它可以把错误信息输出到指定的文件而不是标准错误设备。所以你可以输入下面这样的命令来把错误重定向到文件：

nasm -E myfile.err -f obj myfile.asm

2.1.8 `-s' 选项: 把错误信息输出到'stdout'

'-s'选项可以把错误信息重定向到'stdout'而不是'stderr'，它可以在'MS-DOS'下进行重定向。想要在汇编文件'myfile.asm'时把它的输出用管道输出给'more'程序，可以这样：

nasm -s -f obj myfile.asm | more

请参考 2.1.7 的'-E'选项.

2.1.9 `-i'选项: 包含文件搜索路径

当NASM在源文件中看到'%include'操作符时(参阅4.6)，它不仅仅会在当前目录下搜索给出的文件，还会搜索'-i'选项在命令行中指定的所有路径。所以你可以从宏定义库中包含进一个文件，比如，输入：

```
nasm -ic:\macrolib\ -f obj myfile.asm
```

(通常，在 '-i'与路径名之间的空格是允许的，并且可选的。)

NASM更多的关注源代码级上的完全可移植性，所以并不理解正运行的操作系统对文件的命名习惯；你提供给'-i'作为参数的的字符串会被一字不差地加在包含文件的文件名前。所以，上例中最后面的一个反斜杠是必要的，在Unix下，一个尾部的正斜线也同样是必要的。

(当然，如果你确实需要，你也可以不正规地使用它，比如，选项'-ifoo'会导致'%incldue "bar.i'去搜索文件'foobar.i'...)

如果你希望定义一个标准的搜索路径，比如像Unix系统下的'/usr/include'，你可以在环境
变量NASMENV中放置一个或多个'-i'(参阅2.1.19)

为了与绝大多数C编译器的Makefile保持兼容，该选项也可以被写成'-I'。

2.1.10 `-p' 选项: 预包含一个文件

NASM允许你通过'-p'选项来指定一个文件预包含进你的源文件。所以，如果运行：

```
nasm myfile.asm -p myinc.inc
```

跟在源文件开头写上'%include "myinc.inc"然后运行'nasm myfile.asm'是等效的。

为和'-I','-D','-U'选项操持一致性，该选项也可以被写成'-P'

2.1.11 `-d'选项: 预定义一个宏。

就像'-p'选项给出了在文件头放置'%include'的另一种实现，'-d'选项给出了在文件中写'%define'的另一种实现，你可以写：

```
nasm myfile.asm -dFOO=100
```

作为在文件中写下面一行语句的一种替代实现：

```
%define FOO 100
```

在文件的开始，你可以取消一个宏定义，同样，选项'-dFOO'等同于代码'%define FOO'。
这种形式的操作符在选择编译时操作中非常有用，它们可以用'%ifdef'来进行测试，比如'-dDEBUG'。

为了与绝大多数 C 编译器的 Makefile 保持兼容，该选项也可以被写成'-D'。

2.1.12 `-u' 选项: 取消一个宏定义。

'-u'选项可以用来取消一个由'-p'或'-d'选项先前在命令行上定义的一个宏定义。

比如，下面的命令语句：

```
nasm myfile.asm -dFOO=100 -uFOO
```

会导致'FOO'不是一个在程序中预定义的宏。这在 Makefile 中不同位置重载一个操作时很有用。

为了与绝大多数 C 编译器的 Makefile 保持兼容，该选项也可以被写成'-U'。

2.1.13 `-e'选项: 仅预处理。

NASM 允许预处理器独立运行。使用'-e'选项(不需要参数)会导致 NASM 预处理输入文件，展开所有的宏，去掉所有的注释和预处理操作符，然后把结果文件打印在标准输出上(如果'-o'选项也被指定的话，会被存入一个文件)。

该选项不能被用在那些需要预处理器去计算与符号相关的表达式的程序中，所以如下面的代码：

```
%assign tablesize ($-tablestart)
```

会在仅预处理模式中会出错。

2.1.14 `-a' 选项: 不需要预处理。

如果 NASM 被用作编译器的后台，那么假设编译器已经作完了预处理，并禁止 NASM 的预处理功能显然是可以节约时间，加快编译速度。'-a'选项(不需要参数)，会让 NASM 把它强大的预处理器换成另一个什么也不做的预处理器。

2.1.15 `-On'选项: 指定多遍优化。

NASM 在缺省状态下是一个两遍的汇编器。这意味着如果你有一个复杂的源文件需要多于两遍的汇编。你必须告诉它。

使用'-O'选项，你可以告诉 NASM 执行多遍汇编。语法如下：

(*)'-OO'严格执行两遍优化，JMP 和 Jcc 的处理和 0.98 版类似，除了向后跳的 JMP 是短跳
转，如果可能，立即数在它们的短格式没有被指定的情况下使用长格式。

(*)'-O1'严格执行两遍优化，但前向分支被汇编成保证能够到达的代码；可能产生比'-OO'更大的代码，但在分支中的偏移地址没有指定的情况下汇编成功的机率更大，

(*)'-On' 多编优化，最小化分支的偏移，最小化带符号的立即数，当'strict'关键字
没有用的时候重载指定的大小(参阅 3.7)，如果 2<=n<=3，会有 5*n 遍,而不是 n 遍。

注意这是一个大写的 O，和小写的 o 是不同的，小写的 o 是指定输出文件的格式，可参阅 2.1.1

2.1.16 `-t'选项: 使用 TASM 兼容模式。

NASM 有一个与 Borlands 的 TASM 之间的受限的兼容格式。如果使用了 NASM 的'-t'选项，就会产生下列变化：

(*)本地符号的前缀由'.'改为'@@'

(*)TASM 风格的以'@'开头的应答文件可以由命令行指定。这和 NASM 支持的'-@resp'风格是不同的。

(*)扩号中的尺寸替换被支持。在 TASM 兼容模式中，方括号中的尺寸替换改变了操作数的尺寸大小，方括号不再支持 NASM 语法的操作数地址。比如，'mov eax,[DWORD VAL]'在 TASM 兼容语法中是合法的。但注意你失去了为指令替换缺省地址类型的能力。

(*)'%arg'预处理操作符被支持，它同 TASM 的 ARG 操作符相似。

(*) `%local'预处理操作符。

(*) `%stacksize'预处理操作符。

(*) 某些操作符的无前缀形式被支持。 (`arg', `elif',`else', `endif', `if', `ifdef', `ifdifi', `ifndef', `include',`local')

(*) 还有很多...

需要更多的关于操作符的信息，请参阅 4.9 的 TASM 兼容预处理操作符指令。

2.1.17 `-w'选项: 使汇编警告信息有效或无效。

NASM 可以在汇编过程中监视很多的情况，其中很多是值得反馈给用户的，但这些情况还不足以构成严重错误以使 NASM 停止产生输出文件。这些情况被以类似错误的形式报告给用户，但在报告信息的前面加上'warning'字样。警告信息不会阻止 NASM 产生输出文件并向操作系统返回成功信息。

有些情况甚至还要宽松：他们仅仅是一些值得提供给用户的信息。所以，NASM 支持'-w'命令行选项。它以使特定类型的汇编警告信息输出有效或无效。这样的警告类型是以命名来描述的，比如，'orphan-labels'，你可以以下列的命令行选项让此类警告信息得以输出：'-w+orphan-labels'，或者以'-w-orphan-labels'让此类信息不能输出。

可禁止的警告信息类型有下列一些：

(*)`macro-params'包括以错误的参数个数调用多行的宏定义的警告。这类警告信息缺省情况下是输出的，至于为什么你可能需要禁止它，请参阅 4.3.1。

(*)`orphan-labels'包含源文件行中没有指令却定义了一个没有结尾分号的 label 的
警告。缺省状况下，NASM 不输出此类警告。如果你需要它，请参阅 3.1 的例子。

(*)'number-overflow'包含那些数值常数不符合 32 位格式警告信息（比如，你很容易打
了很多的 F，错误产生了'0x7fffffffffff')。这种警告信息缺省状况下是打开的。

2.1.18 `-v'选项: 打印版本信息。

输入'NASM -v'会显示你正使用的 NASM 的版本号，还有它被编译的时间。

如果你要提交 bug 报告，你可能需要版本号。

2.1.19 `NASMENV'环境变量。

如果你定义了一个叫'NASMENV'的环境变量，程序会被把它认作是命令行选项附加的一部分，它会在真正的命令行之前被处理。你可以通过在'NASMENV'中使用'-i'选项来定义包含文件的标准搜索路径。

环境变量的值是通过空格符分隔的，所以值'-s ic:\nasmlib'会被看作两个单独的操作。也正因为如此，意味着值'-dNAME='my name'不会象你预期的那样被处理，因为它会在空格符处被分开，NASM 的命令行处理会被两个没有意义的字符串'-dNAME="my'和'name"'给弄混。

为了解决这个问题，NASM 为此提供了一个特性，如果你在'NASMENV'环境变量的第一个字符处写上一个非减号字符，NASM 就会把这个字符当作是选项的分隔符。所以把环境变量设成'!-s!-ic:\nasmlib'跟'-s -ic:\nasmlib'没什么两样，但是

'!-dNAME="my name"就会正常工作了。

这个环境变量以前叫做'NASM',从版本 0.98.32 以后开始叫这个名字。

2.2 MASM 用户速成。

如果你曾使用 MASM 写程序，或者使用在 MASM 兼容模式下使用 TASM, 或者使用'a86'，本节将阐述 MASM 与 NASM 语法之间的主要区别。如果你没有使用过 MASM，那最好先跳过这一节。

2.2.1 NASM 是大小写敏感的。

一个简单的区别是 NASM 是大小写敏感的。当你调用你的符号'foo','Foo',或'FOO'时，它们是不同的。如果你在汇编'DOS'或'OS/2', '.OBJ'文件，你可以使用'UPPERCASE'操作符来保证所有的导出到其他代码模式的符号都是大写的；但是，在仅仅一个单独的模块中，NASM 会区分大小写符事情。

2.2.2 NASM 需要方括号来引用内存地址。

NASM 的设计思想是语法尽可能简洁。它的一个设计目标是，它将在被使用的过程中，尽可能得让用户看到一个单行的 NASM 代码时，就可以说出它会产生什么操作码。你可以在 NASM 中这样做，比如，如果你声明了：

```
foo     equ     1
bar     dw      2
```

然后有两行的代码：

```
        mov     ax,foo
        mov     ax,bar
```

尽管它们有看上去完全相同的语法，但却产生了完全不同的操作码

NASM 为了避免这种令人讨厌的情况，拥有一个相当简单的内存引用语未能。规则是任何对内存中内容的存取操作必须要在地址上加上方括号。但任何对地址值的操作不需要。所以，形如'mov ax,foo'的指令总是代表一个编译时常数，不管它是一个 'EQU'或一个变量的地址；如果要取变量'bar'的内容，你必须与'mov ax,[bar]'。

这也意味着 NASM 不需要 MASM 的'OFFSET'关键字，因为 MASM 的代码'mov ax,offset bar'
同 NASM 的语法'mov ax,bar'是完全等效的。如果你希望让大量的 MASM 代码能够被 NASM 汇编通过，你可以编写'%idefine offset'让预处理器把'OFFSET'处理成一个无操作符。

这个问题在'a86'中就更混乱了。

NASM 因为关注简洁性，同样不支持 MASM 和它的衍生产品支持的的混合语法，比如像:'mov ax, table[bx]',这里，一个中括号外的部分加上括号内的一个部分引用一个内存地址，上面代码的正确语法是：'mov ax,[table+bx] 。同样，'mov ax,es:[di]'也是错误的，正确的应该是'mov ax,[es:di]'。

2.2.3 NASM 不存储变量的类型。

NASM 被设计成不记住你声明的变量的类型。然而，MASM 在看到'var dw 0'时会记住类型，然后就可以隐式地合用'mov var, 2'给变量赋值。NASM 不会记住关于变量'var'的任何东西，除了它的位置，所以你必须显式地写上代码'mov word [var],2'。

因为这个原因，NASM 不支持'LODS','MOVS','STOS','SCANS','CMPS','INS',或'OUTS'指令，仅仅支持形如'LODSB','MOVSW',和'SCANSD'之类的指令。它们都显式地指定被处理的字符串的尺寸。

2.2.4 NASM 不会 `ASSUME'

作为 NASM 简洁性的一部分，它同样不支持'ASSUME'操作符。NASM 不会记住你往段寄存器里放了什么值，也不会自动产生段替换前缀。

2.2.5 NASM 不支持内存模型。

NASM 同样不含有任何操作符来支持不同的 16 位内存模型。程序员需要自己跟踪那些函数需要 far call，哪些需要 near call。并需要确定放置正确的'RET'指令('RETN'或'RETF'; NASM 接受'RET'作为'RETN'的另一种形式);另外程序员需要在调用外部函数时在需要处编写 CALL FAR 指令，并必须跟踪哪些外部变量定义是 far,哪些是 near。

2.2.6 浮点处理上的不同。

NASM 使用跟 MASM 不同的浮点寄存器名：MASM 叫它们'ST(0)','ST(1)'等，而'a86'叫它们'0','1'等，NASM 则叫它们'st0','st1'等。

在版本 0.96 上，NASM 现在以跟 MASM 兼容汇编器同样的方式处理'nowait'形式的指令，0.95 以及更早的版本上的不同的处理方式主要是因为作者的误解。

2.2.7 其他不同。

由于历史的原因，NASM 把 MASM 兼容汇编器的'TBYTE'写成'TWORD'。

NASM 以跟 MASM 不同的一种方式声明未初始化的内存。MASM 的程序员必须使用'stack db 64 dup (?)', NASM 需要这样写：'stack resb 64',读作"保留 64 字节"。为了

保持可移植性，NASM 把'?'看作是符号名称中的一个有效的字符，所以你可以编写这样的代码'? equ 0', 然后写'dw ?'可以做一些有用的事情。'DUP'还是一个不被支持的语法。

另外，宏与操作符的工作方式也与 MASM 完全不同，可以到参阅第 4，第 5 章。

第三章 NASM 语言

3.1 NASM 源程序行的组成。

就像很多其他的汇编器，每一行 NASM 源代码包含(除非它是一个宏，一个预处理操作符，或一个汇编器操作符，参况第 4，5 章)下面四个部分的全部或某几个部分：

```
label:    instruction operands        ; comment
```

通常，这些域的大部分是可选的；label,instruction,comment 存在或不存在都是允许的。当然，operands 域会因为 instruction 域的要求而必需存或必须不存在。

NASM 使用反斜线(\)作为续行符；如果一个以一个反斜线结束，那第二行会被认为是前面一行的一部分。

NASM 对于一行中的空格符并没有严格的限制：labels 可以在它们的前面有空格，或其他任何东西。label 后面的冒号同样也是可选的。(注意到，这意味着如果你想要写一行'lodsb'，但却错误地写成了'lodab'，这仍将是有效的一行，但这一行不做任何事情，只是定义了一个 label。运行 NASM 时带上命令行选项'-w+orphan-labels'会让 NASM 在你定义了一个不以冒号结尾的 label 时警告你。

labels 中的有效的字符是字母，数字，'-','$','#','@','~','.'和'?'。但只有字母'.',(具有特殊含义，参阅 3.9),'_'和'?'可以作为标识符的开头。一个标识符还可以加上一个'$'前缀，以表明它被作为一个标识符而不是保留字来处理。这样的话，如果你想到链接进来的其他模块中定义了一个符号叫'eax'，你可以用'$eax'在 NASM 代码中引用它，以和寄存器的符号区分开。

instruction 域可以包含任何机器指令：Pentium 和 P6 指令，FPU 指令，MMX 指令还有甚至没有公开的指令也会被支持。这些指令可以加上前缀'LOCK','REP','REPE/REPZ'或'REPNE'/'REPNZ'，通常，支持显示的地址尺寸和操作数尺寸前缀'A16','A32','016'和'032'。关于使用它们的一个例子在第九章给出。你也可以使用段寄存器名作为指令前缀： 代码'es mov [bx],ax'等效于代码'mov [es:bx],ax'。我们推荐后一种语法。因为它和语法中的其它语法特性一致。但是对于象'LODSB'这样的指令，它没有操作数，但还是可以有一个段前缀， 对于'es lodsb'没有清晰地语法处理方式

在使用一个前缀时，指令不是必须的，像'CS','A32','LOCK'或'REPE'这样的段前缀可以单独出现在一行上，NASM 仅仅产生一个前缀字节。

作为对实际机器指令的扩展，NASM 同时提供了一定数量的伪操作指令，这在 3.2 节详细描述。

指令操作数可以使用一定的格式：它们可以是寄存器，仅仅以寄存器名来表示(比

如：'ax','bp','ebx','cr0'：NASM 不使用'gas'的语法风格，在这种风格中，寄存器名前必须加上一个'%'符号)，或者它们可以是有效的地址(参阅 3.3)，常数(3.4)，或表达式。

对于浮点指令，NASM 接受各种语法：你可以使用 MASM 支持的双操作数形式，或者你可以使用 NASM 的在大多数情况下全用的单操作数形式。支持的所以指令的语法细节可以参阅附录 B。比如，你可以写：

```
fadd    st1             ; this sets st0 := st0 + st1
fadd    st0,st1         ; so does this

fadd    st1,st0         ; this sets st1 := st1 + st0
fadd    to st1          ; so does this
```

几乎所有的浮点指令在引用内存时必须使用以下前缀中的一个'DWORD',QWORD'或'TWORD'来指明它所引用的内存的尺寸。

3.2 伪指令。

伪指令是一些并不是真正的 x86 机器指令，但还是被用在了 instruction 域中的指令，因为使用它们可以带来很大的方便。当前的伪指令有'DB','DW','DD','DQ'和‘DT’，它们对应的未初始化指令是'RESB','RESW','RESD','RESQ'和'REST'，'INCBIN'命令，'EQU'命令和'TIEMS'前缀。

3.2.1 `DB'一类的伪指令: 声明已初始化的数据。

在 NASM 中，`DB', `DW', `DD', `DQ'和`DT'经常被用来在输出文件中声明已初始化的数据，你可以多种方式使用它们：

```
db    0x55                ; just the byte 0x55
db    0x55,0x56,0x57      ; three bytes in succession
db    'a',0x55            ; character constants are OK
db    'hello',13,10,'$'   ; so are string constants
dw    0x1234              ; 0x34 0x12
dw    'a'                 ; 0x41 0x00 (it's just a number)
dw    'ab'                ; 0x41 0x42 (character constant)
dw    'abc'               ; 0x41 0x42 0x43 0x00 (string)
dd    0x12345678          ; 0x78 0x56 0x34 0x12
dd    1.234567e20         ; floating-point constant
dq    1.234567e20         ; double-precision float
dt    1.234567e20         ; extended-precision float
```

'DQ'和'DT'不接受数值常数或字符串常数作为操作数。

3.2.2 `RESB'类的伪指令：声明未初始化的数据。

`RESB', `RESW', `RESD', `RESQ' and `REST'被设计用在模块的 BSS 段中：它们声明未初始化的存储空间。每一个带有单个操作数，用来表明字节数，字数，或双字数或其他的需要保留单位。就像在 2.2.7 中所描述的，NASM 不支持 MASM/TASM 的扣留未初始化空间的语法'DW ?'或类似的东西：现在我们所描述的正是 NASM 自己的方式。'RESB'类伪指令的操作数是有严格的语法的，参阅 3.8。
比如：

```
buffer:         resb    64              ; reserve 64 bytes
wordvar:        resw    1               ; reserve a word
realarray       resq    10              ; array of ten reals
```

3.2.3 `INCBIN':包含其他二进制文件。

'INCBIN'是从老的 Amiga 汇编器 DevPac 中借过来的：它将一个二进制文件逐字逐句地包含到输出文件中。这能很方便地在一个游戏可执行文件中包含中图像或声音数据。它可以以下三种形式的任何一种使用：

```
incbin  "file.dat"              ; include the whole file
incbin  "file.dat",1024         ; skip the first 1024 bytes
incbin  "file.dat",1024,512     ; skip the first 1024, and
                                ; actually include at most 512
```

3.2.4 `EQU': 定义常数。

'EQU'定义一个符号，代表一个常量值：当使用'EQU'时，源文件行上必须包含一个 label。'EQU'的行为就是把给出的 label 的名字定义成它的操作数(唯一)的值。定义是不可更改的，比如：

```
message         db      'hello, world'
      msglen            equ     $-message
```

把'msglen'定义成了常量 12。'msglen'不能再被重定义。这也不是一个预自理定义：'msglen'的值只被计算一次，计算中使用到了'$'(参阅 3.5)在此时的含义。注意 ‘EQU’ 的操作数也是一个严格语法的表达式。(参阅 3.8)

3.2.5 `TIMES': 重复指令或数据。

前缀'TIMES'导致指令被汇编多次。它在某种程序上是 NASM 的与 MASM 兼容汇编器的'DUP'语法的等价物。你可以这样写：

```
zerobuf:        times 64 db 0
```

或类似的东西，但'TEIMES'的能力远不止于此。'TIMES'的参数不仅仅是一个数值常数，还有数值表达式，所以你可以这样做：

```
buffer: db      'hello, world'
times 64-$+buffer db ' '
```

它可以把'buffer'的长度精确地定义为 64 字节，' TIMES '可以被用在一般地指令上，所以你可像这要编写不展开的循环：

```
            times 100 movsb
```

注意在'times 100 resb 1'跟'resb 100'之间并没有显著的区别，除了后者在汇编时会快上一百倍。

就像'EQU','RESB'它们一样， 'TIMES'的操作数也是严格语法的表达式。(见 3.8)

注意'TIMES'不可以被用在宏上：原因是'TIMES'在宏被分析后再被处理，它允许' TIMES'的参数包含像上面的'64-$+buffer'这样的表达式。要重复多于一行的代码，或者一个宏，使用预处理指令'%rep'。

3.3 有效地址

一个有效地址是一个指令的操作数,它是对内存的一个引用。在 NASM 中，有效地址的语法是非常简单的：它由一个可计算的表达式组成，放在一个中括号内。比如：

```
wordvar dw      123
        mov     ax,[wordvar]
        mov     ax,[wordvar+1]
        mov     ax,[es:wordvar+bx]
```

任何与上例不一致的表达都不是 NASM 中有效的内存引用，比如：'es:wordvar[bx]'。

更复杂一些的有效地址，比如含有多个寄存器的，也是以同样的方式工作：

```
            mov     eax,[ebx*2+ecx+offset]
            mov     ax,[bp+di+8]
```

NASM 在这些有效地址上具有进行代数运算的能力，所以看似不合法的一些有效地址使用上都是没有问题的：

```
        mov     eax,[ebx*5]             ; assembles as [ebx*4+ebx]
        mov     eax,[label1*2-label2]   ; ie [label1+(label1-label2)]
```

有些形式的有效地址在汇编后具有多种形式；在大多数情况下，NASM 会自动产生

最小化的形式。比如，32 位的有效地址'[eax*2+0]'和'[eax+eax]'在汇编后具有完全不同的形式，NASM 通常只会生成后者，因为前者会为 0 偏移多开辟 4 个字节。

NASM 具有一种隐含的机制，它会对'[eax+ebx]'和'[ebx+eax]'产生不同的操作码；通常，这是很有用的，因为'[esi+ebp]'和'[ebp+esi]'具有不同的缺省段寄存器。

尽管如此，你也可以使用关键字'BYTE','WORD','DWORD'和'NOSPLIT'强制 NASM 产生特定形式的有效地址。如果你想让'[eax+3]'被汇编成具有一个 double-word 的偏移域，而不是由 NASM 缺省产生一个字节的偏移。你可以使用'[dword eax+3]'，同样，你可以强制 NASM 为一个第一遍汇编时没有看见的小值产生一个一字节的偏移(像这样的例子，可以参阅 3.8)。比如：'[byte eax+offset]'。有一种特殊情况，‘[byte eax]'会被汇编成'[eax+0]'。带有一个字节的 0 偏移。而'[dword eax]'会带一个 double-word 的 0 偏移。而常用的形式，'[eax]'则不会带有偏移域。

当你希望在 16 位的代码中存取 32 位段中的数据时，上面所描述的形式是非常有用的。关于这方面的更多信息，请参阅 9.2。实际上，如果你要存取一个在已知偏移地址处的数据，而这个地址又大于 16 位值，如果你不指定一个 dword 偏移，NASM 会让高位上的偏移值丢失。

类似的，NASM 会把'[eax*2]'分裂成'[eax+eax]' ,因为这样可以让偏移域不存在以此节省空间；实际上，它也把'[eax*2+offset]'分成'[eax+eax+offset]'，你可以使用 ‘NOSPLIT'关键字改变这种行为：`[nosplit eax*2]'会强制`[eax*2+0]'按字面意思被处理。

3.4 常数

NASM 能理解四种不同类型的常数：数值，字符，字符串和浮点数。

3.4.1 数值常数。

一个数值常数就只是一个数值而已。NASM 允许你以多种方式指定数值使用的进制，你可以以后缀'H','Q','B'来指定十六进制数，八进制数和二进制数，或者你可以用 C 风格的前缀'0x'表示十六进制数，或者以 Borland Pascal 风格的前缀'$'来表示十六进制数，注意，'$'前缀在标识符中具有双重职责(参阅 3.1)，所以一个以'$'作前缀的十六进制数值必须在'$'后紧跟数字，而不是字符。

请看一些例子：

```
mov     ax,100          ; decimal
mov     ax,0a2h         ; hex
mov     ax,$0a2         ; hex again: the 0 is required
mov     ax,0xa2         ; hex yet again
mov     ax,777q         ; octal
```

```
mov     ax,10010011b    ; binary
```

3.4.2 字符型常数。

一个字符常数最多由包含在双引号或单引号中的四个字符组成。引号的类型与使用跟 NASM 其它地方没什么区别，但有一点，单引号中允许有双引号出现。

一个具有多个字符的字符常数会被 little-endian order，如果你编写：

```
mov eax,'abcd'
```

产生的常数不会是`0x61626364'，而是`0x64636261'，所以你把常数存入内存的话，它会读成'abcd'而不是'dcba'。这也是奔腾的'CPUID'指令理解的字符常数形式(参阅 B.4.34)

3.4.3 字符串常数。

字符串常数一般只被一些伪操作指令接受，比如'DB'类，还有'INCBIN'。

一个字符串常数和字符常数看上去很相像，但会长一些。它被处理成最大长度的字符常数之间的连接。所以，以下两个语句是等价的：

```
db    'hello'               ; string constant
db    'h','e','l','l','o'   ; equivalent character constants
```

还有，下面的也是等价的:

```
dd    'ninechars'           ; doubleword string constant
dd    'nine','char','s'     ; becomes three doublewords
db    'ninechars',0,0,0     ; and really looks like this
```

注意，如果作为'db'的操作数，类似'ab'的常数会被处理成字符串常量，因为它作为字符常数的话，还不够短，因为，如果不这样，那'db 'ab'会跟'db 'a''具有同样的效果，那是很愚蠢的。同样的，三字符或四字符常数会在作为'dw'的操作数时被处理成字符串。

3.4.4 浮点常量

浮点常量只在作为'DD','DQ','DT'的操作数时被接受。它们以传统的形式表达：数值，然后一个句点，然后是可选的更多的数值，然后是选项'E'跟上一个指数。句点是强制必须有的，这样，NASM 就可以把它们跟'dd 1'区分开，它只是声明一个整型常数，而'dd 1.0'声明一个浮点型常数。

一些例子:

```
dd     1.2                        ; an easy one
dq     1.e10                      ; 10,000,000,000
dq     1.e+10                     ; synonymous with 1.e10
dq     1.e-10                     ; 0.000 000 000 1
dt     3.14159265358979238462 ; pi
```

NASM 不能在编译时求浮点常数的值。这是因为 NASM 被设计为可移植的，尽管它常产生 x86 处理器上的代码，汇编器本身却可以和 ANSI C 编译器一起运行在任何系统上。所以，汇编器不能保证系统上总存在一个能处理 Intel 浮点数的浮点单元。所以，NASM 为了能够处理浮点运算，它必须含有它自己的一套完整的浮点处理例程，它大大增加了汇编器的大小，却获得了并不多的好处。

3.5 表达式

NASM 中的表达式语法跟 C 里的是非常相似的。

> NASM 不能确定编译时在计算表达式时的整型数尺寸：因为 NASM 可以在 64 位系统上非常好的编译和运行，不要假设表达式总是在 32 位的寄存器中被计算的，所以要慎重地对待整型数溢出的情况。它并不总能正常的工作。NASM 唯一能够保证的是：你至少拥有 32 位长度。

NASM 在表达式中支持两个特殊的记号，即'$'和'$$',它们允许引用当前指令的地址。'$'计算得到它本身所在源代码行的开始处的地址；所以你可以简单地写这样的代码'jmp $'来表示无限循环。'$$'计算当前段开始处的地址，所以你可以通过($-$$)找出你当前在段内的偏移。

NASM 提供的运算符以运算优先级为序列举如下：

3.5.1 `|': 位或运算符。

运算符'|'给出一个位级的或运算，所执行的操作与机器指令'or'是完全相同的。位或是 NASM 中优先级最低的运算符。

3.5.2 `^': 位异或运算符。

`^' 提供位异或操作。

3.5.3 `&': 位与运算符。

`&' 提供位与运算。

3.5.4 `<<' and `>>': 位移运算符。

`<<' 提供位左移，跟 C 中的实现一样，所以'5<<3'相当于把 5 乘上 8。'>>'提供位右移。在 NASM 中，这样的位移总是无符号的，所以位移后，左侧总是以零填充，并不会有符号扩展。

3.5.5 `+' and `-': 加与减运算符。

'+'与'-'运算符提供完整的普通加减法功能。

3.5.6 `*', `/', `//', `%'和`%%': 乘除法运算符。

'*'是乘法运算符。'/'和'//'都是除法运算符，'/'是无符号除，'//'是带符号除。同样的，'%'和'%%'提供无符号与带符号的模运算。

同 ANSI C 一样，NASM 不保证对带符号模操作执行的操作的有效性。

因为'%'符号也被宏预处理器使用，你必须保证不管是带符号还是无符号的模操作符都必须跟有空格。

3.5.7 一元运算符: `+', `-', `~'和`SEG'

这些只作用于一个参数的一元运算符是 NASM 的表达式语法中优先级最高的。'-'把它的操作数取反，'+'不作任何事情(它只是为了和'-'保持对称)，'~'对它的操作数取补码，而'SEG'提供它的操作数的段地址(在 3.6 中会有详细解释)。

3.6 `SEG'和`WRT'

当写很大的 16 位程序时，必须把它分成很多段，这时，引用段内一个符号的地址的能力是非常有必要的，NASM 提供了'SEG'操作符来实现这个功能。

'SEG'操作符返回符号所在的首选段的段基址，即一个段基址，当符号的偏移地址以它为参考时，是有效的，所以，代码：

```
mov     ax,seg symbol
mov     es,ax
mov     bx,symbol
```

总是在'ES:BX'中载入一个指向符号'symbol'的有效指针。

而事情往往可能比这还要复杂些：因为 16 位的段与组是可以相互重叠的，你通常可能需要通过不同的段基址，而不是首选的段基址来引用一个符号，NASM 可以让你这样做，通过使用'WRT'关键字，你可以这样写：

```
mov     ax,weird_seg        ; weird_seg is a segment base
```

```
mov     es,ax
mov     bx,symbol wrt weird_seg
```

会在'ES:BX'中载入一个不同的，但功能上却是相同的指向'symbol'的指针。

通过使用'call segment:offset'，NASM 提供 fall call(段内)和 jump,这里'segment'和'offset'都以立即数的形式出现。所以要调用一个远过程，你可以如下编写代码：

```
call    (seg procedure):procedure
call    weird_seg:(procedure wrt weird_seg)
```

(上面的圆括号只是为了说明方便，实际使用中并不需要)

NASM 支持形如'call far procedure'的语法，跟上面第一句是等价的。'jmp'的工作方式跟'call'在这里完全相同。

在数据段中要声明一个指向数据元素的远指针，可以象下面这样写：

```
dw      symbol, seg symbol
```

NASM 没有提供更便利的写法，但你可以用宏自己建造一个。

3.7 `STRICT': 约束优化。

当在汇编时把优化器打开到 2 或更高级的时候(参阅 2.1.15)。NASM 会使用尺寸约束('BYTE','WORD','DWORD','QWORD',或'TWORD')，会给它们尽可能小的尺寸。关键字'STRICT'用来制约这种优化，强制一个特定的操作数为一个特定的尺寸。比如，当优化器打开，并在'BITS 16'模式下：

```
push dword 33
```

会被编码成 `66 6A 21',而

```
push strict dword 33
```

会被编码成六个字节，带有一个完整的双字立即数`66 68 21 00 00 00'.

而当优化器关闭时，不管'STRICT'有没有使用，都会产生相同的代码。

3.8 临界表达式。

NASM 的一个限制是它是一个两遍的汇编器；不像 TASM 和其它汇编器，它总是只做两遍汇编。所以它就不能处理那些非常复杂的需要三遍甚至更多遍汇编的源代码。

第一遍汇编是用于确定所有的代码与数据的尺寸大小，这样的话，在第二遍产生代码的时候，就可以知道代码引用的所有符号地址。所以，有一件事 NASM 不能处理，那就是一段代码的尺寸依赖于另一个符号值，而这个符号又在这段代码的后面被声明。比如：

```
  times (label-$) db 0
label:  db      'Where am I?'
```

'TIMES'的参数本来是可以合法得进行计算的，但 NASM 中不允许这样做，因为它在第一次看到 TIMES 时的时候并不知道它的尺寸大小。它会拒绝这样的代码。

```
  times (label-$+1) db 0
  label:  db      'NOW where am I?'
```

在上面的代码中，TIMES 的参数是错误的。

NASM 使用一个叫做临界表达式的概念，以禁止上述的这些例子，临界表达式被定义为一个表达式，它所需要的值在第一遍汇编时都是可计算的，所以，该表达式所依赖的符号都是之前已经定义了的，'TIMES'前缀的参数就是一个临界表达式；同样的原因，'RESB'类的伪指令的参数也是临界表达式。

临界表达式可能会出现下面这样的情况：

```
                mov     ax,symbol1
symbol1         equ     symbol2
symbol2:
```

在第一遍的时候，NASM 不能确定'symbol1'的值，因为'symbol1'被定义成等于'symbols2',而这时，NASM 还没有看到 symbol2。所以在第二遍的时候，当它遇上'mov ax,symbol1',它不能为它产生正确的代码，因为它还没有知道'symbol1'的值。当到达下一行的时候，它又看到了'EQU'，这时它可以确定 symbol1 的值了，但这时已经太晚了。

NASM 为了避免此类问题，把'EQU'右侧的表达式也定义为临界表达式，所以，'symbol1'的定义在第一遍的时候就会被拒绝。

这里还有一个关于前向引用的问题：考虑下面的代码段：

```
  mov     eax,[ebx+offset]
```

```
offset  equ     10
```

NASM 在第一遍的时候，必须在不知道'offset'值的情况下计算指令'mov eax,[ebx+offset]'的尺寸大小。它没有办法知道'offset'足够小，足以放在一个字节的偏移域中，所以，它以产生一个短形式的有效地址编码的方式来解决这个问题；在第一遍中，它所知道的所有关于'offset'的情况是：它可能是代码段中的一个符号，而且，它可能需要四字节的形式。所以，它强制这条指令的长度为适合四字节地址域的长度。在第二遍的时候，这个决定已经作出了，它保持使这条指令很长，所以，这种情况下产生的代码没有足够的小，这个问题可以通过先定义 offset 的办法得到解决，或者强制有效地址的尺寸大小，象这样写代码：

```
[byte ebx+offset]
```

3.9 本地 Labels

NASM 对于那些以一个句点开始的符号会作特殊处理,一个以单个句点开始的 Label 会被处理成本地 label, 这意味着它会跟前面一个非本地 label 相关联. 比如:

```
label1  ; some code

.loop
        ; some more code

        jne     .loop
        ret

label2  ; some code

.loop
        ; some more code

        jne     .loop
        ret
```

上面的代码片断中,每一个'JNE'指令跳至离它较近的前面的一行上,因为'.loop'的两个定义通过与它们前面的非本地 Label 相关联而被分离开来了。

对于本地 Label 的处理方式是从老的 Amiga 汇编器 DevPac 中借鉴过来的；尽管如此，NASM 提供了进一步的性能，允许从另一段代码中调用本地 labels。这是通过在本地 label 的前面加上非本地 label 前缀实现的：第一个.loop 实际上被定义为'label1.loop'，而第二个符号被记作'label2.loop'。所以你确实需要的话你可写：

```
label3  ; some more code
        ; and some more

        jmp label1.loop
```

有时，这是很有用的(比如在使用宏的时候)，可以定义一个 label,它可以在任何地方被引用，但它不会对常规的本地 label 机制产生干扰。这样的 label 不能是非本地 label,因为非本地 label 会对本地 labels 的重复定义与引用产生干扰；也不能是本地的，因为这样定义的宏就不能知道 label 的全称了。所以 NASM 引进了第三类 label,它只在宏定义中有用：如果一个 label 以一个前缀'..@'开始，它不会对本地 label 产生干扰，所以，你可以写：

```
label1:                         ; a non-local label
.local:                         ; this is really label1.local
..@foo:                         ; this is a special symbol
label2:                         ; another non-local label
.local:                         ; this is really label2.local

        jmp     ..@foo          ; this will jump three lines up
```

NASM 还能定义其他的特殊符号，比如以两个句点开始的符号，比如'..start'被用来指定'.obj'输出文件的执行入口。(参阅 6.2.6)

第四章 NASM 预处理器

NASM 拥有一个强大的宏处理器，它支持条件汇编，多级文件包含，两种形式的宏(单行的与多行的)，还有为更强大的宏能力而设置的‘context stack’机制
预处理指令都是以一个’%’打头。

预处理器把所有以反斜杠(\)结尾的连续行合并为一行，比如：

```
%define THIS_VERY_LONG_MACRO_NAME_IS_DEFINED_TO \
        THIS_value
```

、
会像是单独一行那样正常工作。

4.1 单行的宏。

4.1.1 最常用的方式: `%define’

单行的宏是以预处理指令’%define’定义的。定义工作同 C 很相似，所以你可以这样做：

```
%define ctrl    0x1F &
%define param(a,b) ((a)+(a)*(b))

        mov     byte [param(2,ebx)], ctrl 'D'
```

会被扩展为：

```
        mov     byte [(2)+(2)*(ebx)], 0x1F & 'D'
```

当单行的宏被扩展开后还含有其它的宏时，展开工作会在执行时进行，而不是定义时，如下面的代码：

```
%define a(x)    1+b(x)
%define b(x)    2*x

        mov     ax,a(8)
```

会如预期的那样被展开成’mov ax, 1+2*8’， 尽管宏’b’并不是在定义宏 a 的时候定义的。

用’%define’定义的宏是大小写敏感的：在代码’%define foo bar’之后，只有’foo’会被扩展成’bar’：’Foo’或者’FOO’都不会。用’%idefine’来代替’%define’(i 代表’insensitive’)，你可以一次定义所有的大小写不同的宏。所以

'%idefine foo bar'会导致'foo','FOO','Foo'等都会被扩展成'bar'。

当一个嵌套定义(一个宏定义中含有它本身)的宏被展开时，有一个机制可以检测到，并保证不会进入一个无限循环。如果有嵌套定义的宏，预处理器只会展开第一层，因此，如果你这样写：

```
%define a(x)    1+a(x)

                mov     ax,a(3)
```

宏 `a(3)'会被扩展成'1+a(3)'，不会再被进一步扩展。这种行为是很有用的，有关这样的例子请参阅 8.1。

你甚至可以重载单行宏：如果你这样写：

```
%define foo(x)   1+x
%define foo(x,y) 1+x*y
```

预处理器能够处理这两种宏调用，它是通过你传递的参数的个数来进行区分的，所以'foo(3)'会变成'1+3'，而'foo(ebx,2)'会变成'1+ebx*2'。尽管如此，但如果你定义了：

```
%define foo bar
```

那么其他的对'foo'的定义都不会被接受了：一个不带参数的宏定义不允许对它进行带有参数进行重定义。

但这并不能阻止单行宏被重定义：你可以像这样定义，并且工作得很好：

```
%define foo bar
```

然后在源代码文件的稍后位置重定义它：

```
%define foo baz
```

然后，在引用宏'foo'的所有地方，它都会被扩展成最新定义的值。这在用'%assign'定义宏时非常有用(参阅 4.1.5)

你可以在命令行中使用'-d'选项来预定义宏。参阅 2.1.11

4.1.2 %define 的增强版: `%xdefine'

与在调用宏时展开宏不同，如果想要调用一个嵌入有其他宏的宏时，使用它在被定义的值，你需要'%define'不能提供的另外一种机制。解决的方案

是使用'%xdefine'，或者它的大小写不敏感的形式'%xidefine'。

假设你有下列的代码：

```
%define  isTrue  1
      %define  isFalse isTrue
      %define  isTrue  0

      val1:    db      isFalse

      %define  isTrue  1

      val2:    db      isFalse
```

在这种情况下，'val1'等于 0,而'val2'等于 1。这是因为，当一个单行宏用'%define'定义时，它只在被调用时进行展开。而'isFalse'是被展开成'isTrue'，所以展开的是当前的'isTrue'的值。第一次宏被调用时，'isTrue'是 0,而第二次是 1。

如果你希望'isFalse'被展开成在'isFalse'被定义时嵌入的'isTrue'的值，你必须改写上面的代码，使用'%xdefine':

```
      %xdefine isTrue  1
      %xdefine isFalse isTrue
      %xdefine isTrue  0

      val1:    db      isFalse

      %xdefine isTrue  1

      val2:    db      isFalse
```

现在每次'isFalse'被调用，它都会被展开成 1，而这正是嵌入的宏'isTrue'在'isFalse'被定义时的值。

4.1.3 : 连接单行宏的符号： `%+'

一个单行宏中的单独的记号可以被连接起来，组成一个更长的记号以待稍后处理。这在很多处理相似的事情的相似的宏中非常有用。

举个例子，考虑下面的代码:

```
      %define BDASTART 400h                ; Start of BIOS data area
```

```
struc   tBIOSDA                         ; its structure
        .COM1addr       RESW    1
        .COM2addr       RESW    1
        ; ..and so on
endstruc
```

现在，我们需要存取 tBIOSDA 中的元素，我们可以这样：

```
        mov     ax,BDASTART + tBIOSDA.COM1addr
        mov     bx,BDASTART + tBIOSDA.COM2addr
```

如果在很多地方都要用到，这会变得非常的繁琐无趣，但使用下面的宏会大大减小打字的量：

```
; Macro to access BIOS variables by their names (from tBDA):

%define BDA(x)  BDASTART + tBIOSDA. %+ x
```

现在，我们可以象下面这样写代码：

```
        mov     ax,BDA(COM1addr)
        mov     bx,BDA(COM2addr)
```

使用这个特性，我们可以简单地引用大量的宏。(另外，还可以减少打字错误)。

4.1.4 取消宏定义: `%undef'

单行的宏可以使用'%undef'命令来取消。比如，下面的代码：

```
%define foo bar
%undef  foo

        mov     eax, foo
```

会被展开成指令'mov eax, foo'，因为在'%undef'之后，宏'foo'处于无定义状态。

那些被预定义的宏可以通过在命令行上使用'-u'选项来取消定义，参阅 2.1.12。

4.1.5 预处理器变量 : `%assign'

定义单行宏的另一个方式是使用命令'%assign'(它的大小写不敏感形式

是%iassign，它们之间的区别与'%idefine','%idefine'之间的区别完全相同)。

'%assign'被用来定义单行宏，它不带有参数，并有一个数值型的值。它的值可以以表达式的形式指定，并要在'%assing'指令被处理时可以被一次计算出来，

就像'%define','%assign'定义的宏可以在后来被重定义，所以你可以这样做：

```
%assign i i+1
```

以此来增加宏的数值

'%assing'在控制'%rep'的预处理器循环的结束条件时非常有用：请参阅 4.5 的例子。另外的关于'%assign'的使用在 7.4 和 8.1 中的提到。

赋给'%assign'的表达式也是临界表达式(参阅 3.8)，而且必须可被计算成一个纯数值型(不能是一个可重定位的指向代码或数据的地址，或是包含在寄存器中的一个值。)

4.2 字符串处理宏: `%strlen' and `%substr'

在宏里可以处理字符串通常是非常有用的。NASM 支持两个简单的字符串处理宏，通过它们，可以创建更为复杂的操作符。

4.2.1 求字符串长度: `%strlen'

'%strlen'宏就像'%assign'，会为宏创建一个数值型的值。不同点在于'%strlen'创建的数值是一个字符串的长度。下面是一个使用的例子：

```
%strlen charcnt 'my string'
```

在这个例子中，'charcnt'会接受一个值 8,就跟使用了'%assign'一样的效果。在这个例子中，'my string'是一个字面上的字符串，但它也可以是一个可以被扩展成字符串的单行宏，就像下面的例子：

```
%define sometext 'my string'
%strlen charcnt sometext
```

就像第一种情况那样，这也会给'charcnt'赋值 8

4.2.2 取子字符串: `%substr'

字符串中的单个字符可以通过使用'%substr'提取出来。关于它使用的一个例子可能比下面的描述更为有用：

```
%substr mychar  'xyz' 1         ; equivalent to %define mychar 'x'
%substr mychar  'xyz' 2         ; equivalent to %define mychar 'y'
%substr mychar  'xyz' 3         ; equivalent to %define mychar 'z'
```

在这个例子中，mychar 得到了值'z'。就像在'%strlen'(参阅 4.2.1)中那样，第一个参数是一个将要被创建的单行宏，第二个是字符串，第三个参数指定哪一个字符将被选出。注意，第一个索引值是 1 而不是 0，而最后一个索引值等同于'%strlen'给出的值。如果索引值超出了范围，会得到一个空字符串。

4.3 多行宏: `%macro'

多行宏看上去更象 MASM 和 TASM 中的宏：一个 NASM 中定义的多行宏看上去就象下面这样：

```
%macro  prologue 1

        push    ebp
        mov     ebp,esp
        sub     esp,%1

%endmacro
```

这里，定义了一个类似 C 函数的宏 prologue：所以你可以通过一个调用来使用宏：

```
myfunc:   prologue 12
```

这会把三行代码扩展成如下的样子：

```
myfunc: push    ebp
        mov     ebp,esp
        sub     esp,12
```

在'%macro'一行上宏名后面的数字'1'定义了宏可以接收的参数的个数。宏定义里面的'%1'是用来引用宏调用中的第一个参数。对于一个有多个参数的宏，参数序列可以这样写：'%2','%3'等等。

多行宏就像单行宏一样，也是大小写敏感的，除非你使用另一个操作符
‘%imacro'

如果你必须把一个逗号作为参数的一部分传递给多行宏，你可以把整
个参数放在一个括号中。所以你可以象下面这样编写代码：

```
%macro  silly 2

    %2: db      %1

%endmacro

        silly 'a', letter_a             ; letter_a:  db 'a'
        silly 'ab', string_ab           ; string_ab: db 'ab'
        silly {13,10}, crlf             ; crlf:      db 13,10
```

4.3.1 多行宏的重载

就象单行宏，多行宏也可以通过定义不同的参数个数对同一个宏进行多次
重载。而这次，没有对不带参数的宏的特殊处理了。所以你可以定义：

```
%macro  prologue 0

        push    ebp
        mov     ebp,esp

%endmacro
```

作为函数 prologue 的另一种形式，它没有开辟本地栈空间。

有时候，你可能需要重载一个机器指令；比如，你可能想定义：

```
%macro  push 2

        push    %1
        push    %2

%endmacro
```

这样，你就可以如下编写代码：

```
        push    ebx             ; this line is not a macro call
        push    eax,ecx         ; but this one is
```

通常，NASM 会对上面的第一行给出一个警告信息，因为'push'现在被定义成
了一个宏，而这行给出的参数个数却不符合宏的定义。但正确的代码还是
会生成的，仅仅是给出一个警告而已。这个警告信息可以通过

'-w'macro-params' 命令行选项来禁止。(参阅 2.1.17)。

4.3.2 Macro-Local Labels

NASM 允许你在多行宏中定义 labels.使它们对于每一个宏调用来讲是本地的:所以多次调用同一个宏每次都会使用不同的 label.你可以通过在 label 名称前面加上'%%'来实现这种用法.所以,你可以创建一条指令,它可以在'Z'标志位被设置时执行'RET'指令,如下:

```
%macro  retz 0

        jnz     %%skip
        ret
    %%skip:

%endmacro
```

你可以任意多次的调用这个宏,在你每次调用时的时候,NASM 都会为'%%skip'建立一个不同的名字来替换它现有的名字.NASM 创建的名字可能是这个样子的:'..@2345.skip',这里的数字 2345 在每次宏调用的时候都会被修改.而'..@'前缀防止 macro-local labels 干扰本地 labels 机制,就像在 3.9 中所描述的那样.你应该避免在定义你自己的宏时使用这种形式('..@'前缀,然后是一个数字,然后是一个句点),因为它们会和 macro-local labels 相互产生干扰.

4.3.3 不确定的宏参数个数.

通常,定义一个宏,它可以在接受了前面的几个参数后, 把后面的所有参数都作为一个参数来使用,这可能是非常有用的,一个相关的例子是,一个宏可能用来写一个字符串到一个 MS-DOS 的文本文件中,这里,你可能希望这样写代码:

```
        writefile [filehandle],"hello, world",13,10
```

NASM 允许你把宏的最后一个参数定义成"贪婪参数", 也就是说你调用这个宏时,使用了比宏预期得要多得多的参数个数,那所有多出来的参数连同它们之间的逗号会被作为一个参数传递给宏中定义的最后一个实参,所以,如果你写:

```
%macro  writefile 2+

        jmp     %%endstr
  %%str:        db      %2
  %%endstr:
        mov     dx,%%str
        mov     cx,%%endstr-%%str
```

```
        mov     bx,%1
        mov     ah,0x40
        int     0x21

    %endmacro
```

那上面使用'writefile'的例子会如预期的那样工作:第一个逗号以前的文本[filehandle]会被作为第一个宏参数使用,会被在'%1'的所有位置上扩展,而所有剩余的文本都被合并到'%2'中,放在 db 后面.

这种宏的贪婪特性在 NASM 中是通过在宏的'%macro'一行上的参数个数后面加上'+'来实现的.

如果你定义了一个贪婪宏,你就等于告诉 NASM 对于那些给出超过实际需要的参数个数的宏调用该如何扩展; 在这种情况下,比如说,NASM 现在知道了当它看到宏调用'writefile'带有 2,3 或 4 个或更多的参数的时候,该如何做.当重载宏时,NASM 会计算参数的个数,不允许你定义另一个带有 4 个参数的'writefile'宏.

当然,上面的宏也可以作为一个非贪婪宏执行,在这种情况下,调用语句应该象下面这样写:

```
        writefile [filehandle], {"hello, world",13,10}
```

NASM 提供两种机制实现把逗号放到宏参数中,你可以选择任意一种你喜欢的形式.

有一个更好的办法来书写上面的宏,请参阅 5.2.1

4.3.4 缺省宏参数.

NASM 可以让你定义一个多行宏带有一个允许的参数个数范围.如果你这样做了,你可以为参数指定缺省值.比如:

```
    %macro  die 0-1 "Painful program death has occurred."

        writefile 2,%1
        mov     ax,0x4c01
        int     0x21

    %endmacro
```

这个宏(它使用了 4.3.3 中定义的宏'writefile')在被调用的时候可以有一个错误信息,它会在退出前被显示在错误输出流上,如果它在被调用时不带参数

,它会使用在宏定义中的缺省错误信息.

通常,你以这种形式指定宏参数个数的最大值与最小值; 最小个数的参数在宏调用的时候是必须的,然后你要为其他的可选参数指定缺省值.所以,当一个宏定义以下面的行开始时:

```
%macro foobar 1-3 eax,[ebx+2]
```

它在被调用时可以使用一到三个参数, 而'%1'在宏调用的时候必须指定,'%2'在没有被宏调用指定的时候,会被缺省地赋为'eax','%3'会被缺省地赋为'[ebx+2]'.

你可能在宏定义时漏掉了缺省值的赋值, 在这种情况下,参数的缺省值被赋为空.这在可带有可变参数个数的宏中非常有用,因为记号'%0'可以让你确定有多少参数被真正传给了宏.

这种缺省参数机制可以和'贪婪参数'机制结合起来使用;这样上面的'die'宏可以被做得更强大,更有用,只要把第一行定义改为如下形式即可:

```
%macro die 0-1+ "Painful program death has occurred.",13,10
```

最大参数个数可以是无限,以'*'表示.在这种情况下,当然就不可能提供所有的缺省参数值. 关于这种用法的例子参见 4.3.6.

4.3.5 `%0': 宏参数个数计数器.

对于一个可带有可变个数参数的宏, 参数引用'%0'会返回一个数值常量表示有多少个参数传给了宏.这可以作为'%rep'的一个参数(参阅 4.5),以用来遍历宏的所有参数. 例子在 4.3.6 中给出.

4.3.6 `%rotate': 循环移动宏参数.

Unix 的 shell 程序员对于'shift' shell 命令再熟悉不过了,它允许把传递给 shell 脚本的参数序列(以'$1,'$2'等引用)左移一个,所以, 前一个参数是‘$1'的话左移之后，就变成' $2'可用了，而在'$1'之前是没有可用的参数的。

NASM 具有相似的机制，使用'%rotate'。就象这个指令的名字所表达的，它跟 Unix 的'shift'是不同的，它不会让任何一个参数丢失，当一个参数被移到最左边的时候，再移动它，它就会跳到右边。

'%rotate'以单个数值作为参数进行调用(也可以是一个表达式)。宏参数被循环左移，左移的次数正好是这个数字所指定的。如果'%rotate'的参数是负数，那么宏参数就会被循环右移。

所以，一对用来保存和恢复寄存器值的宏可以这样写：

```
%macro  multipush 1-*

  %rep  %0
        push    %1
  %rotate 1
  %endrep

%endmacro
```

这个宏从左到右为它的每一个参数都依次调用指令'PUSH'。它开始先把它的第一个参数'%1'压栈，然后调用'%rotate'把所有参数循环左移一个位置，这样一来，原来的第二个参数现在就可以用'%1'来取用了。重复执行这个过程，直到所有的参数都被执行完(这是通过把'%0'作为'%rep'的参数来实现的)。这就实现了把每一个参数都依次压栈。

注意，'*'也可以作为最大参数个数的一个计数，表明你在使用宏'multipush'的时候，参数个数没有上限。

使用这个宏，确实是非常方便的，执行同等的'POP'操作，我们并不需要把参数顺序倒一下。一个完美的解决方案是，你再写一个'multipop'宏调用，然后把上面的调用中的参数复制粘贴过来就行了，这个宏会对所有的寄存器执行相反顺序的 pop 操作。

这可以通过下面定义来实现：

```
%macro  multipop 1-*

  %rep %0
  %rotate -1
        pop     %1
  %endrep

%endmacro
```

这个宏开始先把它的参数循环右移一个位置，这样一来，原来的最后一个参数现在可以用'%1'引用了。然后被 pop,然后，参数序列再一次右移，倒数第二个参数变成了'%1'，就这样，所以参数被以相反的顺序一一被执行。

4.3.7 连结宏参数。

NASM 可以把宏参数连接到其他的文本中。这个特性可以让你声明一个系例的符号，比如，在宏定义中。你希望产生一个关于关键代码的表格，而代码

跟在表中的偏移值有关。你可以这样编写代码：

```
%macro keytab_entry 2

    keypos%1    equ     $-keytab
                db      %2

%endmacro

keytab:
          keytab_entry F1,128+1
          keytab_entry F2,128+2
          keytab_entry Return,13
```

会被扩展成：

```
keytab:
keyposF1        equ     $-keytab
                db      128+1
keyposF2        equ     $-keytab
                db      128+2
keyposReturn    equ     $-keytab
                db      13
```

你可以很轻易地把文本连接到一个宏参数的尾部，这样写即可：'%1foo'。

如果你希望给宏参数加上一个数字，比如，通过传递参数'foo'来定义符号'foo1'和'foo2',但你不能写成'%11'，因为这会被认为是第 11 个参数。你必须写成'%{1}1'，它会把第一个 1 跟第二个分开

这个连结特性还可以用于其他预处理问题中，比如 macro-local labels(4.3.2)和 context-local labels(4.7.2)。在所有的情况中，语法上的含糊不清都可以通过把'%'之后，文本之前的部分放在一个括号中得到解决：所以'%{%foo}bar 会把文本'bar'连接到一个 macro-local label:' %%foo'的真正名字的后面(这个是不必要的，因为就 NASM 处理 macro-local labels 的机制来讲，'%{%foo}bar 和%%foobar 都会被扩展成同样的形式,但不管怎么样，这个连结的能力是在的)

4.3.8 条件代码作为宏参数。

NASM 对于含有条件代码的宏参数会作出特殊处理。你可以以另一种形式'%+1'来使用宏参数引用'%1'，它告诉 NASM 这个宏参数含有一个条件代码，如果你调用这个宏时，参数中没有有效的条件代码，会使预处理器报错。

为了让这个特性更有用，你可以以'%-1'的形式来使用参数，它会让 NASM 把

这个条件代码扩展成它的反面。所以 4.3.2 中定义的宏'retz'还可以以下面的方式重写：

```
%macro  retc 1

        j%-1    %%skip
        ret
  %%skip:

%endmacro
```

这个指令可以使用'retc ne'来进行调用，它会把条件跳转指令扩展成'JE'，或者'retc po'会把它扩展成'JPE'。

'%+1'的宏参数引用可以很好的把参数'CXZ'和'ECXZ'解释为有效的条件代码；但是，'%-1'碰上上述的参数就会报错，因为这两个条件代码没有相反的情况存在。

4.3.9 禁止列表扩展。

当 NASM 为你的源程序产生列表文件的时候，它会在宏调用的地方为你展开多行宏，然后列出展开后的所有行。这可以让你看到宏中的哪些指令展开成了哪些代码；尽管如此，有些不必要的宏展开会把列表弄得很混乱。

NASM 为此提供了一个限定符'.nolist',它可以被包含在一个宏定义中，这样，这个宏就不会在列表文件中被展开。限定符'.nolist'直接放到参数的后面，就像下面这样：

```
%macro foo 1.nolist
```

或者这样：

```
%macro bar 1-5+.nolist a,b,c,d,e,f,g,h
```

4.4 条件汇编

跟 C 预处理器相似，NASM 允许对一段源代码只在某特定条件满足时进行汇编，关于这个特性的语法就像下面所描述的：

```
%if<condition>
    ;if <condition>满足时接下来的代码被汇编。
%elif<condition2>
    ; 当 if<condition>不满足，而<condition2>满足时，该段代码被汇编。
%else
```

```
        ;当<condition>跟<condition2>都不满足时，该段代码被汇编。
    %endif
```

'%else'跟'%elif'子句都是可选的，你也可以使用多于一个的'%elif'子句。

4.4.1 `%ifdef': 测试单行宏是否存在。

'%ifdef MACRO'可以用来开始一个条件汇编块，跟在它后面的代码当且仅当一个叫做'MACRO'单行宏被定义时才会被会汇编。如果没有定义，那么'%elif'和'%else'块会被处理。

比如，当调试一个程序时，你可能希望这样写代码：

```
            ; perform some function
    %ifdef DEBUG
            writefile 2,"Function performed successfully",13,10
    %endif
            ; go and do something else
```

你可以通过使用命令行选项'-dDEBUG'来建立一个处理调试信息的程序，或不使用该选项来产生最终发布的程序。

你也可以测试一个宏是否没有被定义，这可以使用'%ifndef'。你也可以在'%elif'块中测试宏定义，使用'%elifdef'和'%elifndef'即可。

4.4.2 `ifmacro': 测试多行宏是否存在。

除了是测试多行宏的存在的，'%idmacro'操作符的工作方式跟'%ifdef'是一样的。

比如，你可能在编写一个很大的工程，而且无法控制存在链接库中的宏。你可能需要建立一个宏，但必须先确保这个宏没有被建立过，如果被建立过了，你需要为你的宏换一个名字。

如果你定义的一个有特定参数个数与宏名的宏与现有的宏会产生冲突，那么'%ifmacro'会返回真。比如：

```
%ifmacro MyMacro 1-3

        %error "MyMacro 1-3" causes a conflict with an existing macro.

    %else

        %macro MyMacro 1-3
```

```
                ; insert code to define the macro

        %endmacro

    %endif
```

如果没有现有的宏会产生冲突，这会建立一个叫'MyMacro 1-3"的宏，如果会有冲突，那么就产生一条警告信息。

你可以通过使用'%ifnmacro'来测试是否宏不存在。还可以使用'%elifmacro'和'%elifnmacro'在'%elif'块中测试多行宏。

4.4.3 `%ifctx': 测试上下文栈。

当且仅当预处理器的上下文栈中的顶部的上下文的名字是'ctxname'时，条件汇编指令'%ifctx ctxname'会让接下来的语句被汇编。跟'%ifdef'一样，它也有'%ifnctx','%elifctx','%elifnctx'等形式。

关于上下文栈的更多细节，参阅 4.7, 关于'%ifctx'的一个例子，参阅 4.7.5.

4.4.4 `%if': 测试任意数值表达式。

当且仅当数值表达式'expr'的值为非零时，条件汇编指令'%if expr'会让接下来的语句被汇编。使用这个特性可以确定何时中断一个'%rep'预处理器循环，例子参阅 4.5。

'%if'和'%elif'的表达式是一个临界表达式(参阅 3.8)

'%if' 扩展了常规的 NASM 表达式语法，提供了一组在常规表达式中不可用的相关操作符。操作符'=','<','>','<=','>='和'<>'分别测试相等，小于,大于，小于等于，大于等于，不等于。跟 C 相似的形式'=='和'!='作为'=','<>'的另一种形式也被支持。另外，低优先级的逻辑操作符'&&','^^',和'||'作为逻辑与，逻辑异或，逻辑或也被支持。这些跟 C 的逻辑操作符类似（但 C 没有提供逻辑异或），这些逻辑操作符总是返回 0 或 1，并且把任何非零输入看作 1（所以，比如， '^^'它会在它的一个输入是零，另一个非零的时候，总返回 1)。这些操作符返回 1 作为真值，0 作为假值。

4.4.5 `%ifidn' and `%ifidni': 测试文本相同。

当且仅当'text1'和'text2'在作为单行宏展开后是完全相同的一段文本时，结构'%ifidn text1,text2'会让接下来的一段代码被汇编。两段文本在空格个数上的不同会被忽略。

'%ifidni'和'%ifidn'相似，但是大小写不敏感。

比如，下面的宏把一个寄存器或数字压栈，并允许你把 IP 作为一个真实的寄存器使用：

```
%macro  pushparam 1

  %ifidni %1,ip
        call    %%label
  %%label:
  %else
        push    %1
  %endif

%endmacro
```

就像大多数的'%if'结构，'%ifidn'也有一个'%elifidn',并有它的反面的形式'%ifnidn','%elifnidn'.相似的，'%ifidni'也有'%elifidni',`%ifnidni'和`%elifnidni'。

4.4.6 `%ifid', `%ifnum', `%ifstr': 测试记号的类型。

有些宏会根据传给它们的是一个数字，字符串或标识符而执行不同的动作。比如，一个输出字符串的宏可能会希望能够处理传给它的字符串常数或一个指向已存在字符串的指针。

当且仅当在参数列表的第一个记号存在且是一个标识符时，条件汇编指令'%ifid'会让接下来的一段代码被汇编。'%ifnum'相似。但测试记号是否是数字；'%ifstr'测试是否是字符串。

比如，4.3.3 中定义的宏'writefile'可以用'%ifstr'作进一步改进，如下：

```
%macro writefile 2-3+

    %ifstr %2
        jmp     %%endstr
    %if %0 = 3
      %%str:    db      %2,%3
    %else
      %%str:    db      %2
    %endif
      %%endstr: mov     dx,%%str
                mov     cx,%%endstr-%%str
  %else
```

```
                mov     dx,%2
                mov     cx,%3
    %endif
                mov     bx,%1
                mov     ah,0x40
                int     0x21

%endmacro
```

这个宏可以处理以下面两种方式进行的调用：

```
writefile [file], strpointer, length
writefile [file], "hello", 13, 10
```

在第一种方式下，'strpointer'是作为一个已声明的字符串的地址，而'length'作为它的长度；第二种方式中，一个字符串被传给了宏，所以宏就自己声明它，并为它分配地址和长度。

注意，'%ifstr'中的'%if'的使用方式：它首先检测宏是否被传递了两个参数（如果是这样，那么字符串就是一个单个的字符串常量，这样'db %2'就足够了）或者更多（这样情况下，除了前两个参数，后面的全部参数都要被合并到'%3'中，这就需要'db %2,%3'了。）

常见的'%elifXXX','%ifnXXX'和'%elifnXXX'/版本在'%ifid','%ifnum',和'%ifstr'中都是存在的。

4.4.7 `%error': 报告用户自定义错误。

预处理操作符'%error'会让 NASM 报告一个在汇编时产生的错误。所以，如果别的用户想要汇编你的源代码，你必须保证他们用下面的代码定义了正确的宏：

```
%ifdef SOME_MACRO
    ; do some setup
%elifdef SOME_OTHER_MACRO
    ; do some different setup
%else
    %error Neither SOME_MACRO nor SOME_OTHER_MACRO was defined.
%endif
```

然后，任何不理解你的代码的用户都会被汇编时得到关于他们的错误的警告信息，不必等到程序在运行时再出现错误却不知道错在哪儿。

4.5 预处理器循环: `%rep'

虽然 NASM 的'TIMES'前缀非常有用，但是不能用来作用于一个多行宏，因为它是在 NASM 已经展开了宏之后才被处理的。所以，NASM 提供了另外一种形式的循环，这回是在预处理器级别的：'%rep'。

操作符'%rep'和'%endrep'('%rep'带有一个数值参数，可以是一个表达式；'%endrep'不带任何参数)可以用来包围一段代码，然后这段代码可以被复制多次，次数由预处理器指定。

```
%assign i 0
%rep    64
        inc     word [table+2*i]
%assign i i+1
%endrep
```

这段代码会产生连续的 64 个'INC'指令，从内存地址'[table]'一直增长到'[table+126]'。

对于一个复杂的终止条件，或者想要从循环中 break 出来，你可以使用'%exitrep'操作符来终止循环，就像下面这样：

```
fibonacci:
%assign i 0
%assign j 1
%rep 100
%if j > 65535
    %exitrep
%endif
        dw j
%assign k j+i
%assign i j
%assign j k
%endrep

fib_number equ ($-fibonacci)/2
```

上面的代码产生所有 16 位的 Fibonacci 数。但要注意，循环的最大次数还是要作为一个参数传给'%rep'。这可以防止 NASM 预处理器进入一个无限循环。在多任务或多用户系统中，无限循环会导致内存被耗光或其他程序崩溃。

4.6 包含其它文件。

又一次使用到一个跟 C 预处理器语法极其相似的操作符，它可以让你在你的代码中包含其它源文件。这可以通过'%include'来实现：

```
%include "macros.mac"
```

这会把文件'macros.mac'文件中的内容包含到现在的源文件中。

被包含文件会被在当前目录下寻找(就是你在运行 NASM 时所在的目录，并不是 NASM 可执行文件所在的目录或源程序文件所在的目录),你可以在 NASM 的命令行上使用选项'-i'来增加搜索路径。

C 语言中防止文件被重复包含的习惯做法在 NASM 中也适用：如果文件 'macros.mac'中有如下形式的代码：

```
%ifndef MACROS_MAC
    %define MACROS_MAC
    ; now define some macros
%endif
```

这样多次包含该文件就不会引起错误，因为第二次包含该文件时，什么也不会发生，因为宏'MACROS_MAC'已经被定义过了。

在没用'%include'操作符包含一个文件时，你可以强制让这个文件被包含进来，做法是在 NASM 命令行上使用'-p'选项

4.7 上下文栈。

那些对一个宏定义来讲是本地的 Labels 有时候还不够强大：有时候，你需要能够在多个宏调用之间共享 label。比如一个'REPEAT'...'UNTIL'循环，'REPEAT'宏的展开可能需要能够去引用'UNTIL'中定义的宏。而且在使用这样的宏时，你可能还会嵌套多层循环。

NASM 通过上下文栈提供这个层次上的功能。预处理器维护了一个包含上下文的栈，每一个上下文都有一个名字作为标识。你可以通过指令'%push'往上下文栈中加一个新的上下文，或通过'%pop'去掉一个。你可以定义一些只针对特定上下文来说是本地的 labels。

4.7.1 `%push' and `%pop': 创建和删除上下文。

'%push'操作符用来创建一个新的上下文，然后把它放在上下文栈的顶端。'%push'需要一个参数，它是这个上下文的名字，例如：

```
%push     foobar
```

这会把一个新的叫做'foobar'的上下文放到栈顶。你可以在一个栈中拥有

多个具有相同名字的上下文：它们之间仍旧是可以区分的。

操作符'%pop'不需要参数，删除栈顶的上下文，并把它销毁，同时也删除跟它相关的 labels。

4.7.2 Context-Local Labels

就像'%%foo'会定义一个对于它所在的那个宏来讲是本地的 label 一样，'%$foo'会定义一个对于当前栈顶的上下文来讲是本地的 lable。所以，上文提到的'REPEAT','UNTIL'的例子可以以下面的方式实现：

```
%macro repeat 0

    %push   repeat
    %$begin:

%endmacro

%macro until 1

        j%-1    %$begin
    %pop

%endmacro
```

然后象下面这样使用它：

```
        mov     cx,string
        repeat
        add     cx,3
        scasb
        until   e
```

它会扫描每个字符串中的第四个字节，以查找在 al 中的字节。

如果你需要定义，或存取对于不在栈顶的上下文本地的 label，你可以使用'%$$foo',或'%$$$foo'来存取栈下面的上下文。

4.7.3 Context-Local 单行宏。

NASM 也允许你定义对于一个特定的上下文是本地的单行宏，使用的方式大致相面：

```
%define %$localmac 3
```

这会定义一个对于栈顶的上下文本地的单行宏'%$localmax',当然，在又一个'%push'操作之后，它还是可以通过'%$$localmac'来存取。

4.7.4 `%repl': 对一个上下文改名。

如果你需要改变一个栈顶上下文的名字（比如，为了响应'%ifctx'），你可以在'%pop'之后紧接着一个'%push'；但它会产生负面效应，会破坏所有的跟栈顶上下文相关的 context-local labels 和宏。

NASM 提供了一个操作符'%repl',它可以在不影响相关的宏与 labels 的情况下，为一个上下文换一个名字，所以你可以把下面的破坏性代码替换成另一种形式：

```
%pop
%push    newname
```

换成不具破坏性的版本： `%repl newname'.

4.7.5 使用上下文栈的例子: Block IFs

这个例子几乎使用了所有的上下文栈的特性，包括条件汇编结构'%ifctx'，它把一个块 IF 语句作为一套宏来执行：

```
%macro if 1

          %push if
          j%-1  %$ifnot

      %endmacro

      %macro else 0

        %ifctx if
              %repl   else
              jmp     %$ifend
              %$ifnot:
        %else
              %error  "expected `if' before `else'"
        %endif

      %endmacro

      %macro endif 0
```

```
    %ifctx if
          %$ifnot:
          %pop
    %elifctx      else
          %$ifend:
          %pop
    %else
          %error  "expected `if' or `else' before `endif'"
    %endif

%endmacro
```

这段代码看上去比上面的`REPEAT'和`UNTIL'宏要饱满多了。因为它使用了条件汇编去验证宏以正确的顺序被执行（比如，不能在'if'之间调用'endif')如果出现错误，执行'%error'。

另外，'endif'宏要处理两种不同的情况，即它可能直接跟在'if'后面，也可能跟在'else'后面。它也是通过条件汇编，判断上下文栈的栈顶是'if'还是'else',并据此来执行不同的动作。

'else'宏必须把上下文保存到栈中，好让'if'宏跟'endif'宏中定义的'%$ifnot'引用。但必须改变上下文的名字，这样'endif'就可以知道这中间还有一个'else'。这是通过'%repl'来做这件事情的。

下面是一个使用这些宏的例子:

```
          cmp     ax,bx

          if ae
                cmp     bx,cx

                if ae
                      mov     ax,cx
                else
                      mov     ax,bx
                endif

          else
                cmp     ax,cx

                if ae
                      mov     ax,cx
                endif
```

```
        endif
```

通过把在内层'if'中描述的另一个上下文压栈，放在外层'if'中的上下文的上面，这样，'else'和'endif'总能引用到匹配的'if'或'else'。这个块-'IF'宏处理嵌套的能力相当好，

4.8 标准宏。

NASM 定义了一套标准宏，当开始处理源文件时，这些宏都已经被定义了。如果你真的希望一个程序在执行前没有预定义的宏存在，你可以使用'%clear'操作符清空预处理器的一切。

大多数用户级的操作符（第五章）是作为宏来运行的，这些宏进一步调用原始的操作符；这些在第五章介绍。剩余的标准宏在这里进行描述。

4.8.1 `__NASM_MAJOR__', `__NASM_MINOR__', `__NASM_SUBMINOR__'和`___NASM_PATCHLEVEL__': NASM 版本宏。

单行宏`__NASM_MAJOR__', `__NASM_MINOR__',`__NASM_SUBMINOR__'和`___NASM_PATCHLEVEL__'被展开成当前使用的 NASM 的主版本号,次版本号，子次版本号和补丁级。所在，在 NASM 0.98.32p1 版本中，`__NASM_MAJOR__'被展开成 0，`__NASM_MINOR__'被展开成 98，`__NASM_SUBMINOR__'被展开成 32，`___NASM_PATCHLEVEL__'被定义为 1。

4.8.2 `__NASM_VERSION_ID__': NASM 版本 ID。

单行宏`__NASM_VERSION_ID__'被展开成双字整型数，代表当前使用的版本的 NASM 的全版本数。这个值等于把`__NASM_MAJOR__',`__NASM_MINOR__',`__NASM_SUBMINOR__'和`___NASM_PATCHLEVEL__'连结起来产生一个单个的双字长整型数。所以，对于 0.98.32p1,返回值会等于

```
        dd      0x00622001
```

或者

```
        db      1,32,98,0
```

注意，上面两行代码产生的是完全相同的代码，第二行只是用来指出内存中存在的各个值之间的顺序。

4.8.3 `__NASM_VER__': NASM 版本字符串。

单行宏`__NASM_VER__'被展开成一个字符串，它定义了当前使用的 NASM 的

版本号。所以，在 NASM0.98.32 下：

```
db      __NASM_VER__
```

会被展开成：

```
db      "0.98.32"
```

4.8.4 `__FILE__' and `__LINE__': 文件名和行号。

就像 C 的预处理器，NASM 允许用户找到包含有当前指令的文件的文件名和行数。宏'__FILE__'展开成一个字符串常量，该常量给出当前输入文件的文件名(如果含有'%include'操作符，这个值会在汇编的过程中改变)，而`__LINE__'会被展开成一个数值常量，给出在输入文件中的当前行的行号。

这些宏可以使用在宏中，以查看调试信息，当在一个宏定义中包含宏'__LINE__'时（不管 是单行还是多行），会返回宏调用，而不是宏定义处的行号。这可以用来确定是否在一段代码中发生了程序崩溃。比如，某人可以编写一个子过程'stillhere',它通过'EAX'传递一个行号，然后输出一些信息，比如:"line 155: still here'。你可以这样编写宏：

```
%macro  notdeadyet 0

        push    eax
        mov     eax,__LINE__
        call    stillhere
        pop     eax

%endmacro
```

然后，在你的代码中插入宏调用，直到你发现发生错误的代码为止。

4.8.5 `STRUC' and `ENDSTRUC': 声明一个结构体数据类型。

在 NASM 的内部，没有真正意义上的定义结构体数据类型的机制； 取代它的是，预处理器的功能相当强大，可以把结构体数据类型以一套宏的形式来运行。宏 'STRUCT' 和'ENDSTRUC'是用来定义一个结构体数据类型的。

'STRUCT'带有一个参数，它是结构体的名字。这个名字代表结构体本身，它在结构体内的偏移地址为零，名字加上一个_size 后缀组成的符号，用一个'EQU'给它赋上结构体的大小。一旦 'STRUC'被执行，你就开始在定义一个结构体，你可以用'RESB'类伪指令定义结构体的域,然后使用'ENDSTRUC'来结束定义。

比如，定义一个叫做'mytype'的结构体，包含一个 longword,一个 word,一个 byte,和一个字符串，你可以这样写代码：

```
struc   mytype

  mt_long:      resd    1
  mt_word:      resw    1
  mt_byte:      resb    1
  mt_str:       resb    32

endstruc
```

上面的代码定义了六个符号：'m_long'在地置 0(从结构体'mytype'开头开始到这个 longword 域的偏移),`mt_word'在地置 4, `mt_byte'6, `mt_str'7, `mytype_size'是 39,而`mytype' 自己在地置 0

之所以要把结构体的名字定义在地址零处，是因为要让结构体可以使用本地 labels 机制的缘故：如果你想要在多个结构体中使用具有同样名字的成员，你可以把上面的结构体定义成这个样子：

```
struc mytype

  .long:        resd    1
  .word:        resw    1
  .byte:        resb    1
  .str:         resb    32

endstruc
```

在这个定义中，把结构体域的偏移值定义成了：'mytype.long', `mytype.word', `mytype.byte' and `mytype.str'.

NASM 因此而没有内部的结构体支持，也不支持以句点形式引用结构体中的成员，所以代码'mov ax, [mystruc.mt_word]' 是非法的，'mt_word'是一个常数，就像其它类型的常数一样，所以，正确的语法应该是 'mov ax,[mystruc+mt_word]'或者`mov ax,[mystruc+mytype.word]'.

4.8.6 `ISTRUC', `AT' and `IEND': 声明结构体的一个实例。

定义了一个结构体类型以后，你下一步要做的事情往往就是在你的数据段中声明一个结构体的实例。NASM 通过使用'ISTRUC'机制提供一种非常简单的方式。在程序中声明一个'mytype'结构体，你可以象下面这样写代码：

```
mystruc:
```

```
istruc mytype

    at mt_long, dd      123456
    at mt_word, dw      1024
    at mt_byte, db      'x'
    at mt_str,  db      'hello, world', 13, 10, 0

iend
```

‘AT’宏的功能是通过使用'TIMES'前缀把偏移位置定位到正确的结构体域上，然后，声明一个特定的数据。所以，结构体域必须以在结构体定义中相同的顺序被声明。

如果为结构体的域赋值要多于一行，那接下的内容可直接跟在'AT'行后面，比如：

```
    at mt_str,  db      123,134,145,156,167,178,189
                db      190,100,0
```

按个人的喜好不同，你也可以不在'AT'行上写数据，而直接在第二行开始写数据域：

```
    at mt_str
            db      'hello, world'
            db      13,10,0
```

4.8.7 `ALIGN' and `ALIGNB': 数据对齐

宏'ALIGN'和'ALIGNB'提供一种便捷的方式来进行数据或代码的在字，双字，段或其他边界上的对齐(有些汇编器把这两个宏叫做'EVEN')，有关这两个宏的语法是：

```
    align   4               ; align on 4-byte boundary
    align   16              ; align on 16-byte boundary
    align   8,db 0          ; pad with 0s rather than NOPs
    align   4,resb 1        ; align to 4 in the BSS
    alignb  4               ; equivalent to previous line
```

这两个个参数都要求它们的第一个参数是 2 的幂；它们都会计算需要多少字节来来存储当前段，当然这个字节数必须向上对齐到一个 2 的幂值。然后用它们的第二个参数来执行'TIMES'前缀进行对齐。

如果第二个参数没有被指定，那'ALIGN'的缺省值就是'NOP',而'ALIGNB'的缺省值就是'RESB 1'.当第二个参数被指定时，这两个宏是等效的。通常，你可以在

数据段与代码段中使用'ALIGN',而在 BSS 段中使用'ALIGNB',除非有特殊用途，一般你不需要第二个参数。

作为两个简单的宏，'ALIGN'与'ALIGNB'不执行错误检查：如果它们的第一个参数不是 2 的某次方，或它们的第二个参数大于一个字节的代码，他们都不会有警告信息，这两种情况下，它们都会执行错误。

'ALIGNB'(或者，'ALIGN'带上第二个参数'RESB 1')可以用在结构体的定义中：

```
struc mytype2

        mt_byte:
              resb 1
              alignb 2
        mt_word:
              resw 1
              alignb 4
        mt_long:
              resd 1
        mt_str:
              resb 32

      endstruc
```

这可以保证结构体的成员被对齐到跟结构体的基地址之间有一个正确的偏移值。

最后需要注意的是，'ALIGN'和'ALIGNB'都是以段的开始地址作为参考的，而不是整个可执行程序的地址空间。如果你所在的段只能保证对齐到 4 字节的边界，那它会对齐对 16 字节的边界，这会造成浪费，另外，NASM 不会检测段的对齐特性是否可被'ALIGN'和'ALIGNB'使用。

4.9 TASM 兼容预处理指令。

接下来的预处理操作符只有在用'-t'命令行开关把 TASM 兼容模式打开的情况下才可以使用(这个开关在 2.1.16 介绍过)

(*) `%arg' (见 4.9.1 节)

(*) `%stacksize' (见 4.9.2 节)

(*) `%local' (见 4.9.3 节)

4.9.1 `%arg'操作符

'%arg'操作符用来简化栈上的参数传递操作处理。基于栈的参数传递在很多高级语言中被使用，包括 C，C++和 Pascal。

而 NASM 企图通过宏来实现这种功能(参阅 7.4.5)，它的语法使用上不是很舒服，而且跟 TASM 之间是不兼容的。这里有一个例子，展示了只通过宏'%arg'来处理：

```
some_function:

    %push     mycontext         ; save the current context
    %stacksize large            ; tell NASM to use bp
    %arg      i:word, j_ptr:word

        mov     ax,[i]
        mov     bx,[j_ptr]
        add     ax,[bx]
        ret

    %pop                        ; restore original context
```

这跟在 7.4.5 中定义的过程很相似，把 j_ptr 指向的值加到 i 中，然后把相加的结果在 AX 中返回，对于'push'和'pop'的展开请参阅 4.7.1 关于上下文栈的使用。

4.9.2 `%stacksize'指令。

'%stacksize'指令是跟'%arg'和'%local'指令结合起来使用的。它告诉 NASM 为'%arg'和'%local'使用的缺省大小。'%stacksize'指令带有一个参数，它是'flat','large'或'small'。

```
%stacksize flat
```

这种形式将使 NASM 使用相对于'ebp'的基于栈的参数地址。它假设使用一个近调用来得到这个参数表。(比如，eip 被压栈).

```
%stacksize large
```

而这种形式会用'bp'来进行基于栈的参数寻址，假设使用了一个远调用来获得这个地址(比如，ip 和 cs 都会被压栈)。

```
%stacksize small
```

这种形式也使用'bp'来进行基于栈的参数寻址，但它跟'large'不同，因为他假设 bp 的旧值已经被压栈。换句话说，你假设 bp,ip 和 cs 正在栈顶，在它们下面的所有本地空间已经被'ENTER'指令开辟好了。当和'%local'指令结合的时候，这种形式特别有用。

4.9.3 `%local'指令。

'%local'指令用来简化在栈框架中进行本地临时栈变量的分配。C 语言中的自动本地变量是这种类型变量的一个例子。'%local'指令跟'%stacksize'一起使用的时候特别有用。并和'%arg'指令保持兼容。它也允许简化对于那些用'ENTER'指令分配在栈中的变量的引用(关于 ENTER 指令，请参况 B.4.65)。这里有一个关于它们的使用的例子：

```
silly_swap:

    %push mycontext             ; save the current context
    %stacksize small            ; tell NASM to use bp
    %assign %$localsize 0       ; see text for explanation
    %local old_ax:word, old_dx:word

        enter   %$localsize,0   ; see text for explanation
        mov     [old_ax],ax     ; swap ax & bx
        mov     [old_dx],dx     ; and swap dx & cx
        mov     ax,bx
        mov     dx,cx
        mov     bx,[old_ax]
        mov     cx,[old_dx]
        leave                   ; restore old bp
        ret                     ;

    %pop                        ; restore original context
```

变量'%$localsize'是在'%local'的内部使用，而且必须在'%local'指令使用前，被定义在当前的上下文中。不这样做，在每一个'%local'变量声明的地方会引发一个表达式语法错误。它然后可以用在一条适当的 ENTER 指令中。

4.10 其他的预处理指令。

NASM 还有一些预处理指令允许从外部源中获取信息，现在，他们包括：

下面的预处理指令使 NASM 能正确地自理 C++/C 语言预处理器的输出。

(*) `%line' 使 NASM 能正确地自理 C++/C 语言预处理器的输出。(参阅 4.10.1)

(*) `%!' 使 NASM 从一个环境变量中读取信息，然后这些信息就可以在你的程序中使用了。(4.10.2)

4.10.1 `%line'操作符。

'%line'操作符被用来通知NASM，输入行与另一个文件中指定的行号相关。一般这另一个文件会是一个源程序文件，它作为现在NASM的输入，但它是一个预处理器的输出。'%line'指令允许NASM输出关于在这个源程序文件中的指定行号的信息，而不是被NASM读进来的整个文件。

这个预处理指令通常不会被程序员用到，但会让预处理器的作者感兴趣，'%line'的使用方法如下：

```
%line nnn[+mmm] [filename]
```

在这个指令中，'nnn'指定源程序文件中与之相关的特定行，'mmm'是一个可选的参数，它指定一个行递增的值；每一个被读进来的源文件行被认为与源程序文件中的'mmm'行相关。最终，'filename'可选参数指定源程序文件的文件名。

在读到一条'%line'预处理指令后，NASM会报告与指定的值相关的所有的文件名和行号

4.10.2 `%!'`<env>': 读取一个环境变量。

`%!<env>' 操作符可以在汇编时读取一个环境变量的值，这可以用在一个环境变量的内容保存到一个字符串中。该字符串可以用在你程序的其他地方。

比如，假设你有一个环境变量'FOO',你希望把'FOO'的值嵌入到你的程序中去。你可以这样做：

```
%define FOO    %!FOO
%define quote   '

tmpstr  db      quote FOO quote
```

在写的时候，在定义'quote'时，它会产生一个'没有结束的字符串'的警告信息,它会自己在读进来的字符串的前后加上一个空格。我没有办法找到一个简单的工作方式(尽管可以通过宏来创建)，我认为，你没有必要学习创建更为复杂的宏，或者如果你用这种方式使用这个特性，你没有必要使用额外的空间。

第五章：汇编器指令

尽管 NASM 极力避免 MASN 和 TASM 中的那些庸肿复杂的东西，但还是不得不支持少量的指令，这些指令在本章进行描述。

NASM 的指令有两种类型：用户级指令和原始指令。一般地，每一条指令都有一个用户级形式和原始形式。在大多数情况下，我们推荐用户使用有户级指令，它们以宏的形式运行，并去调用原始形式的指令。

原始指令被包含在一个方括号中；用户级指令没有括号。

除了本章所描述的这些通用的指令，每一种目标文件格式为了控制文件格式的一些特性，可以使用一些另外的指令。这些格式相关的指令在第六章中跟相关的文件格式一起进行介绍。

5.1 `BITS'：指定目标处理器模式。

'BITS'指令指定 NASM 产生的代码是被设计运行在 16 位模式的处理器上还是运行在 32 位模式的处理器上。语法是'BITS 16'或'BITS 32'

大多数情况下，你可能不需要显式地指定'BITS'。'aout','coff','elf'和'win32'目标文件格式都是被设计用在 32 位操作系统上的，它们会让 NASM 缺省选择 32 位模式。而'obj'目标文件格式允许你为每一个段指定'USE16'或'USE32'，然后 NASM 就会按你的指定设定操作模式，所以多次使用'BITS'是没有必要的。

最有可能使用'BITS'的场合是在一个纯二进制文件中使用 32 位代码；这是因为'bin'输出格式在作为 DOS 的'.COM'程序，DOS 的'.SYS'设备驱动程序，或引导程序时，默认都是 16 位模式。

如果你仅仅是为了在 16 位的 DOS 程序中使用 32 位指令，你不必指定'BITS 32'，如果你这样做了，汇编器反而会产生错误的代码，因为这样它会产生运行在 16 位模式下，却以 32 位平台为目标的代码。

当 NASM 在'BITS 16'状态下时，使用 32 位数据的指令可以加一个字节的前缀 0x66，要使用 32 位的地址，可以加上 0x67 前缀。在'BITS 32'状态下，相反的情况成立，32 位指令不需要前缀，而使用 16 位数据的指令需要 0x66 前缀，使用 16 位地址的指令需要 0x67 前缀。

'BITS'指令拥有一个等效的原始形式：[BITS 16]和[BITS 32]。而用户级的形式只是一个仅仅调用原始形式的宏。

5.1.1 `USE16' & `USE32'：BITS 的别名。

'USE16'和'USE32'指令可以用来取代'BITS 16'和'BITS 32',这是为了和其他汇编器保持兼容性。

5.2 `SECTION'或`SEGMENT': 改变和定义段。

'SECTION'指令('SEGMENT'跟它完全等效)改变你正编写的代码将被汇编进的段。在某些目标文件格式中，段的数量与名称是确定的；而在别一些格式中，用户可以建立任意多的段。因此，如果你企图切换到一个不存在的段，'SECTION'有时可能会给出错误信息，或者定义出一个新段，

> Unix 的目标文件格式和'bin'目标文件格式，都支持标准的段'.text','.data'和'bss'段，与之不同的的，'obj'格式不能辩识上面的段名，并需要把段名开头的句点去掉。

5.2.1 宏 `__SECT__'

'SECTION'指令跟一般指令有所不同，的用户级形式跟它的原始形式在功能上有所不同，原始形式[SECTION xyz],简单地切换到给出的目标段。用户级形式，'SECTION xyz'先定义一个单行宏'__SECT__',定义为原始形式[SECTION]，这正是要执行的指令，然后执行它。所以，用户级指令：

```
        SECTION .text
```

被展开成两行：

```
%define __SECT__        [SECTION .text]
        [SECTION .text]
```

用户会发现在他们自己的宏中，这是非常有用的。比如，4.3.3 中定义的宏'writefile'以下面的更为精致的写法会更有用：

```
%macro  writefile 2+

        [section .data]

  %%str:        db      %2
  %%endstr:

        __SECT__

        mov     dx,%%str
        mov     cx,%%endstr-%%str
        mov     bx,%1
```

```
        mov     ah,0x40
        int     0x21

%endmacro
```

这个形式的宏，一次传递一个用出输出的字符串，先用原始形式的'SECTION'切换至临时的数据段，这样就不会破会宏'__SECT__'。然后它把它的字符串声明在数据段中，然后调用'__SECT__'切换加用户先前所在的段。这样就可以避免先前版本的'writefile'宏中的用来跳过数据的'JMP'指令，而且在一个更为复杂的格式模型中也不会失败，用户可以把这个宏放在任何独立的代码段中进行汇编。

5.3 `ABSOLUTE': 定义绝对 labels。

'ABSOLUTE'操作符可以被认为是'SECTION'的另一种形式：它会让接下来的代码不在任何的物理段中，而是在一个从给定地址开始的假想段中。在这种模式中，你唯一能使用的指令是'RESB'类指令。

`ABSOLUTE'可以象下面这样使用:

```
absolute 0x1A

    kbuf_chr    resw    1
    kbuf_free   resw    1
    kbuf        resw    16
```

这个例子描述了一个关于在段地址 0x40 处的 PC BIOS 数据域的段，上面的代码把'kbuf_chr'定义在 0x1A 处，'kbuf_free'定义在地址 0x1C 处，'kbuf'定义在地址 0x1E。

就像'SECTION'一样，用户级的'ABSOLUTE'在执行时会重定义'__SECT__'宏。

'STRUC'和'ENDSTRUC'被定义成使用'ABSOLUTE'的宏（同时也使用了'__SECT__'）

'ABSOLUTE'不一定需要带有一个绝对常量作为参数：它也可以带有一个表达式（实际上是一个临界表达式，参阅 3.8），表达式的值可以是在一个段中。比如，一个 TSR 程序可以在用它重用它的设置代码所占的空间：

```
        org     100h                ; it's a .COM program

        jmp     setup               ; setup code comes last

        ; the resident part of the TSR goes here
setup:
        ; now write the code that installs the TSR here
```

```
absolute setup

runtimevar1     resw    1
runtimevar2     resd    20

tsr_end:
```

这会在 setup 段的开始处定义一些变量，所以，在 setup 运行完后，它所占用的内存空间可以被作为 TSR 的数据存储空莘而得到重用。符号'tsr_end'可以用来计算 TSR 程序所需占用空间的大小。

5.4 `EXTERN': 从其他的模块中导入符中。

'EXTERN'跟 MASM 的操作符'EXTRN'，C 的关键字'extern'极其相似：它被用来声明一个符号，这个符号在当前模块中没有被定义，但被认为是定义在其他的模块中，但需要在当前模块中对它引用。不是所有的目标文件格式都支持外部变量的：'bin'文件格式就不行。

'EXTERN'操作符可以带有任意多个参数，每一个都是一个符号名：

```
extern  _printf
extern  _sscanf,_fscanf
```

有些目标文件格式为'EXTERN'提供了额外的特性。在所有情况下，要使用这些额外特性，必须在符号名后面加一个冒号，然后跟上目标文件格式相关的一些文字。比如'obj'文件格式允许你声明一个以外部组'dgroup'为段基址一个变量，可以象下面这样写：

```
extern  _variable:wrt dgroup
```

原始形式的'EXTERN'跟用户级的形式有所不同，因为它只能带有一个参数：对于多个参数的支持是在预处理器级上的特性。

你可以把同一个变量作为'EXTERN'声明多次：NASM 会忽略掉第二次和后来声明的，只采用第一个。但你不能象声明其他变量一样声明一个'EXTERN'变量。

5.5 `GLOBAL': 把符号导出到其他模块中。

'GLOBAL'是'EXTERN'的对立面：如果一个模块声明一个'EXTERN'的符号，然后引用它，然后为了防止链接错误，另外某一个模块必须确实定义了该符号，然后把它声明为'GLOBAL',有些汇编器使用名字'PUBLIC'。

'GLOBAL'操作符所作用的符号必须在'GLOBAL'之后进行定义。

'GLOBAL'使用跟'EXTERN'相同的语法，除了它所引用的符号必须在同一样模块中已经被定义过了，比如：

```
global _main
_main:
        ; some code
```

就像'EXTERN'一样，'GLOBAL'允许目标格式文件通过冒号定义它们自己的扩展。比如'elf'目标文件格式可以让你指定全局数据是函数或数据。

```
global  hashlookup:function, hashtable:data
```

就象'EXTERN'一样，原始形式的'GLOBAL'跟用户级的形式不同，仅能一次带有一个参数

5.6 `COMMON': 定义通用数据域。

'COMMON'操作符被用来声明通用变量。一个通用变量很象一个在非初始化数据段中定义的全局变量。所以：

```
common  intvar  4
```

功能上跟下面的代码相似：

```
global  intvar
section .bss

intvar  resd    1
```

不同点是如果多于一个的模块定义了相同的通用变量，在链接时，这些通用变量会被合并，然后，所有模块中的所有的对'intvar'的引用会指向同一片内存。

就角'GLOBAL'和'EXTERN','COMMON'支持目标文件特定的扩展。比如，'obj'文件格式允许通用变量为 NEAR 或 FAR,而'elf'格式允许你指定通用变量的对齐需要。

```
common  commvar  4:near  ; works in OBJ
common  intarray 100:4   ; works in ELF: 4 byte aligned
```

它的原始形式也只能带有一个参数。

5.7 `CPU': 定义 CPU 相关。

'CPU'指令限制只能运行特定 CPU 类型上的指令。

选项如下:

(*) `CPU 8086' 只汇编 8086 的指令集。

(*) `CPU 186' 汇编 80186 及其以下的指令集。

(*) `CPU 286' 汇编 80286 及其以下的指令集。

(*) `CPU 386' 汇编 80386 及其以下的指令集。

(*) `CPU 486' 486 指令集。

(*) `CPU 586' Pentium 指令集。

(*) `CPU PENTIUM' 同 586。

(*) `CPU 686' P6 指令集。

(*) `CPU PPRO' 同 686

(*) `CPU P2' 同 686

(*) `CPU P3' Pentium III and Katmai 指令集。

(*) `CPU KATMAI' 同 P3

(*) `CPU P4' Pentium 4 (Willamette)指令集

(*) `CPU WILLAMETTE' 同 P4

(*) `CPU IA64' IA64 CPU (x86 模式下)指令集

所有选项都是大小写不敏感的，在指定 CPU 或更低一级 CPU 上的所有指令都会被选择。缺省情况下，所有指令都是可用的。

第六章：输出文件的格式

NASM 是一个可移植的汇编器，它被设计为可以在任何 ANSI C 编译器支持的平台上被编译，并可以产生在各种 intel x86 系例的操作系统上运行的代码。为了做到这一点，它拥有大量的可用的输出文件格式，使用命令行上的选项'-f'可以选择。每一种格式对于 NASM 的语法都有一定的扩展，关于这部分内容，本章将详细介绍。

就象在 2.1.1 中所描述的，NASM 基于输入文件的名字和你选择的输出文件的格式为你的输出文件选择一个缺省的名字。这是通过去掉源文件的扩展名('.asm 或'.s'或者其他你使用的扩展名)，然后代之以一个由输出文件格式决定的扩展名。这些输出格式相关的扩展名会在下面一一给出。

6.1 `bin': 纯二进制格式输出。

'bin'格式不产生目标文件：除了你编写的那些代码，它不在输出文件中产生任何东西。这种纯二进制格式的文件可以用在 MS-DOS 中：'.COM'可执行文件和'.SYS'设备驱动程序就是纯二进制格式的。纯二进制格式输出对于操作系统和引导程序开发也是很有用的。

'bin'格式支持多个段名。关于 NASM 处理'bin'格式中的段的细节，请参阅 6.1.3。

使用'bin'格式会让 NASM 进入缺省的 16 位模式（参阅 5.1)。为了能在'bin'格式中使用 32 位代码，比如在操作系统的内核代码中。你必须显式地使用'BITS 32'操作符。

'bin'没有缺省的输出文件扩展名：它只是把输入文件的扩展名去掉后作为输出文件的名字。这样，NASM 在缺省模式下会把'binprog.asm'汇编成二进制文件'binprog'。

6.1.1 `ORG': 二进制程序的起点位置。

'bin'格式提供一个额外的操作符，这在第五章已经给出'ORG'.'ORG'的功能是指定程序被载入内存时，它的起始地址。

比如，下面的代码会产生 longword: `0x00000104':

```
        org     0x100
        dd      label
label:
```

跟 MASM 兼容汇编器提供的'ORG'操作符不同，它们允许你在目标文件中跳转，

并覆盖掉你已经产生的代码，而 NASM 的'ORG'就象它的字面意思“起点”所表示的，它的功能就是为所有内部的地址引用增加一个段内偏移值；它不允许 MASM 版本的'org'的任何其他功能。

6.1.2 `bin'对`SECTION'操作符的扩展。

'bin'输出格式扩展了'SECTION'(或者'SEGMENT')操作符，允许你指定段的对齐请求。这是通过在段定义行的后面加上'ALIGN'限定符实现的。比如：

```
section .data   align=16
```

它切换到段'.data',并指定它必须对齐到 16 字节边界。

'ALIGN'的参数指定了地址值的低位有多少位必须为零。这个对齐值必须为 2 的幂。

6.1.3 `Multisection' 支持 BIN 格式.

'bin'格式允许使用多个段，这些段以一些特定的规则进行排列。

(*) 任何在一个显式的'SECTION'操作符之前的代码都被缺省地加到'.text'段中。

(*) 如果'.text'段中没有给出'ORG'语句，它被缺省地赋为'ORG 0'。

(*) 显式地或隐式地含有'ORG'语句的段会以'ORG'指定的方式存放。代码前会填充 0，以在输出文件中满足 org 指定的偏移。

(*) 如果一个段内含有多个'ORG'语句，最后一条'ORG'语句会被运用到整个段中，不会影响段内多个部分以一定的顺序放到一起。

(*) 没有'ORG'的段会被放到有'ORG'的段的后面，然后，它们就以第一次声明时的顺序被存放。

(*) '.data'段不像'.text'段和'.bss'段那样，它不遵循任何规则，

(*) '.bss'段会被放在所有其他段的后面。

(*) 除非一个更高级别的对齐被指定，所有的段都被对齐到双字边界。

(*) 段之间不可以交迭。

6.2 `obj': 微软 OMF 目标文件

'obj'文件格式（因为历史的原因，NASM 叫它'obj'而不是'omf'）是 MASM 和 TASM 可以产生的一种格式，它是标准的提供给 16 位的 DOS 链接器用来产生'.EXE'文件的格式。它也是 OS/2 使用的格式。

'obj'提供一个缺省的输出文件扩展名'.obj'。

'obj'不是一个专门的 16 位格式，NASM 有一个完整的支持，可以有它的 32 位扩展。32 位 obj 格式的文件是专门给 Borland 的 Win32 编译器使用的，这个编译器不使用微软的新的'win32'目标文件格式。

'obj'格式没有定义特定的段名字：你可以把你的段定义成任何你喜欢的名字。一般的，obj 格式的文件中的段名如：`CODE', `DATA'和`BSS'.

如果你的源文件在显式的'SEGMENT'前包含有代码，NASM 会为你创建一个叫做`__NASMDEFSEG'的段以包含这些代码.

当你在 obj 文件中定义了一个段，NASM 把段名定义为一个符号，所以你可以存取这个段的段地址。比如：

```
segment data

dvar:   dw      1234

segment code

function:
        mov     ax,data         ; get segment address of data
        mov     ds,ax           ; and move it into DS
        inc     word [dvar]     ; now this reference will work
        ret
```

obj 格式中也可以使用'SEG'和'WRT'操作符，所以你可以象下面这样编写代码：

```
extern  foo

      mov   ax,seg foo            ; get preferred segment of foo
      mov   ds,ax
      mov   ax,data               ; a different segment
      mov   es,ax
      mov   ax,[ds:foo]           ; this accesses `foo'
      mov   [es:foo wrt data],bx  ; so does this
```

6.2.1 `obj' 对`SEGMENT'操作符的扩展。

obj 输出格式扩展了'SEGMENT'(或'SECTION')操作符，允许你指定段的多个属性。这是通过在段定义行的末尾添加额外的限定符来实现的，比如：

```
segment code private align=16
```

这定义了一个段'code'，但同时把它声明为一个私有段，同时，它所描述的这个部分必须被对齐到 16 字节边界。

可用的限定符如下:

(*) `PRIVATE', `PUBLIC', `COMMON'和`STACK' 指定段的联合特征。`PRIVATE'段在连接时不和其他的段进行连接；'PUBLIC'和'STACK'段都会在连接时连接到一块儿；而'COMMON‘段都会在同一个地址相互覆盖，而不会一接一个连接好。

(*) 就象上面所描述的，'ALIGN'是用来指定段基址的低位有多少位必须为零，对齐的值必须以 2 的乘方的形式给出，从 1 到 4096；实际上，真正被支持的值只有 1，2，4，16，256 和 4096，所以如果你指定了 8，它会自动向上对齐到 16，32，64 会对齐到 128 等等。注意，对齐到 4096 字节的边界是这种格式的 PharLap 扩展，可能所有的连接器都不支持。

(*) 'CLASS'可以用来指定段的类型；这个特性告诉连接器，具有相同 class 的段应该在输出文件中被放到相近的地址。class 的名字可以是任何字。比如'CLASS=CODE'。

(*) 就象`CLASS', `OVERLAY'通过一个作为参数的字来指定，为那些有覆盖能力的连接器提供覆盖信息。

(*) 段可以被声明为'USE16'或'USE32'，这种选择会对目标文件产生影响，同时在段内 16 位或 32 位代码分开的时候，也能保证 NASM 的缺省汇编模式

(*) 当编写 OS/2 目标文件的时候，你应当把 32 位的段声明为'FLAT'，它会使缺省的段基址进入一个特殊的组'FLAT'，同时，在这个组不存在的时候，定义这个组。

(*) obj 文件格式也允许段在声明的时候，前面有一个定义的绝对段地址，尽管没有连接器知道这个特性应该怎么使用；但如果你需要的话，NASM 还是允许你声明一个段如下面形式：`SEGMENT SCREEN ABSOLUTE=0xB800'`ABSOLUTE'和`ALIGN'关键字是互斥的。

NASM 的缺省段属性是`PUBLIC', `ALIGN=1', 没有 class,没有覆盖, 并 `USE16'.

6.2.2 `GROUP': 定义段组。

obj 格式也允许段被分组，所以一个单独的段寄存器可以被用来引用一个组中的所有段。NASM 因此提供了'GROUP'操作符，据此，你可以这样写代码：

```
segment data

        ; some data

segment bss

        ; some uninitialised data

group dgroup data bss
```

这会定义一个叫做'dgroup'的组，包含有段'data'和'bss'。就象'SEGMENT','GROUP'会把组名定义为一个符号，所以你可以使用'var wrt data'或者'var wrt dgroup'来引用'data'段中的变量'var'，具体用哪一个取决于哪一个段值在你的当前段寄存器中。

如果你只是想引用'var',同时，'var'被声明在一个段中，段本身是作为一个组的一部分，然后，NASM 缺省给你的'var'的偏移值是从组的基地址开始的，而不是段基址。所以，'SEG var'会返回组基址而不是段基址。

NASM 也允许一个段同时作为多个组的一个部分，但如果你真这样做了，会产生一个警告信息。段内同时属于多个组的那些变量在缺省状况下会属于第一个被声明的包含它的组。

一个组也不一定要包含有段；你还是可以使用'WRT'引用一个不在组中的变量。比如说，OS/2 定义了一个特殊的组'FLAT'，它不包含段。

6.2.3 `UPPERCASE': 在输出文件中使大小写敏感无效。

尽管 NASM 自己是大小写敏感的，有些 OMF 连接器并不大小写敏感；所以，如果 NASM 能输出大小写单一的目标文件会很有用。'UPPERCASE'操作符让所有的写入到目标文件中的组，段，符号名全部强制为大写。在一个源文件中，NASM 还是大小写敏感的；但目标文件可以按要求被整个写成是大写的。

'UPPERCASE'写在单独一行中，不需要任何参数。

6.2.4 `IMPORT': 导入 DLL 符号。

如果你正在用 NASM 写一个 DLL 导入库,'IMPORT'操作符可以定义一个从 DLL 库中导入的符号,你使用'IMPORT'操作符的时候,你仍旧需要把符号声明为'EXTERN'.

'IMPORT'操作符需要两个参数,以空格分隔,它们分别是你希望导入的符号的名

称和你希望导入的符号所在的库的名称,比如:

```
import  WSAStartup wsock32.dll
```

第三个参数是可选的,它是符号在你希望从中导入的链接库中的名字,这样的话,
你导入到你的代码中的符号可以和库中的符号不同名,比如:

```
import  asyncsel wsock32.dll WSAAsyncSelect
```

6.2.5 `EXPORT': 导出 DLL 符号.

'EXPORT'也是一个目标格式相关的操作符,它定义一个全局符号,这个符号可以被
作为一个 DLL 符号被导出,如果你用 NASM 写一个 DLL 库.你可以使用这个操作符,在
使用中,你仍旧需要把符号定义为'GLOBAL'.

'EXPORT'带有一个参数,它是你希望导出的在源文件中定义的符号的名字.第二个
参数是可选的(跟第一个这间以空格分隔),它给出符号的外部名字,即你希望让使
用这个 DLL 的应用程序引用这个符号时所用的名字.如果这个名字跟内部名字同名,
可以不使用第二个参数.

还有一些附加的参数,可以用来定义导出符号的一些属性.就像第二个参数, 这些
参数也是以空格分隔.如果要给出这些参数,那么外部名字也必须被指定,即使它跟
内部名字相同也不能省略,可用的属性如下:

(*) 'resident'表示某个导出符号在系统引导后一直常驻内存.这对于一些经常使用
的导出符号来说,是很有用的.

(*) `nodata'表示导出符号是一个函数,这个函数不使用任何已经初始化过的数据.

(*) `parm=NNN', 这里'NNN'是一个整型数,当符号是一个在 32 位段与 16 位段之间的
调用门时,它用来设置参数的尺寸大小(占用多少个 wrod).

(*) 还有一个属性,它仅仅是一个数字,表示符号被导出时带有一个标识数字.

比如:

```
export  myfunc
export  myfunc TheRealMoreformalLookingFunctionName
export  myfunc myfunc 1234  ; export by ordinal
export  myfunc myfunc resident parm=23 nodata
```

6.2.6 `..start': 定义程序的入口点.

'OMF'链接器要求被链接进来的所有目标文件中,必须有且只能有一个程序入口点,

当程序被运行时,就从这个入口点开始.如果定义这个入口点的目标文件是用NASM 汇编的,你可以通过在你希望的地方声明符号'..start'来指定入口点.

6.2.7 `obj'对`EXTERN'操作符的扩展.

如果你以下面的方式声明了一个外部符号:

```
extern  foo
```

然后以这样的方式引用'mov ax,foo',这样只会得到一个关于 foo 的偏移地址,而且这个偏移地址是以'foo'的首选段基址为参考的(在'foo'被定义的这个模块中指定的段).所以,为了存取'foo'的内容,你实际上需要这样做:

```
mov     ax,seg foo      ; get preferred segment base
mov     es,ax           ; move it into ES
mov     ax,[es:foo]     ; and use offset `foo' from it
```

这种方式显得稍稍有点笨拙,实际上如果你知道一个外部符号可以通过给定的段或组来进行存的话,假定组'dgroup'已经在 DS 寄存器中,你可以这样写代码:

```
mov     ax,[foo wrt dgroup]
```

但是,如果你每次要存取'foo'的时候,都要打这么多字是一件很痛苦的事情;所以NASM 允许你声明'foo'的另一种形式:

```
extern  foo:wrt dgroup
```

这种形式让 NASM 假定'foo'的首选段基址是'dgroup';所以,表达式'seg foo'现在会返回'dgroup',表达式'foo'等同于'foo wrt dgroup'.

缺省的'WRT'机制可以用来让外部符号跟你程序中的任何段或组相关联.他也可以被运用到通用变量上,参阅 6.2.8.

6.2.8 `obj'对`COMMON'操作符的扩展.

'obj'格式允许通用变量为 near 或 far;NASM 允许你指定你的变量属于哪一类,语法如下:

```
common  nearvar 2:near   ; `nearvar' is a near common
common  farvar  10:far   ; and `farvar' is far
```

Far 通用变量可能会大于 64Kb,所以 OMF 可以把它们声明为一定数量的指定的大小的元素.比如,10byte 的 far 通用变量可以被声明为 10 个 1byte 的元素,5 个 2byte 的元素,或 2 个 5byte 的元素,或 1 个 10byte 的元素.

有些'OMF'链接器需要元素的 size,同时需要变量的 size,当在多个模块中声明通用变量时可以用来进行匹配.所以 NASM 必须允许你在你的 far 通用变量中指定元素的 size.这可以通过下面的语法实现:

```
common  c_5by2  10:far 5        ; two five-byte elements
common  c_2by5  10:far 2        ; five two-byte elements
```

如果元素的 size 没有被指定,缺省值是 1.还有,如果元素 size 被指定了,那么'far'关键字就不需要了,因为只有 far 通用变量是有元素 size 的.所以上面的声明等同于:

```
common  c_5by2  10:5            ; two five-byte elements
common  c_2by5  10:2            ; five two-byte elements
```

这种扩展的特性还有,'obj'中的'COMMON'操作符还可以象'EXTERN'那样支持缺省的'WRT'指定,你也可以这样声明:

```
common  foo     10:wrt dgroup
common  bar     16:far 2:wrt data
common  baz     24:wrt data:6
```

6.3 `win32': 微软 Win32 目标文件

'win32'输出格式产生微软 win32 目标文件,可以用来给微软连接器进行连接,比如 Visual C++.注意 Borland Win32 编译器不使用这种格式,而是使用'obj'格式(参阅 6.2)

'win32'提供缺省的输出文件扩展名'.obj'.

注意,尽管微软声称 Win32 目标文件遵循'COFF'标准(通用目标文件格式),但是微软的 Win32 编译器产生的目标文件和一些 COFF 连接器(比如 DJGPP)并不兼容,反过来也一样.这是由一些 PC 相关的语义上的差异造成的. 使用 NASM 的'coff'输出格式,可以产生能让 DJGPP 使用的 COFF 文件; 而这种'coff'格式不能产生能让 Win32 连接器正确使用的代码.

6.3.1 `win32'对`SECTION'的扩展.

就象'obj'格式,'win32'允许你在'SECTION'操作符的行上指定附加的信息,以用来控制你声明的段的类型与属性.对于标准的段名'.text','.data',和'.bss',类型和属性是由 NASM 自动产生的,但是还是可以通过一些限定符来重新指定:

可用的限定符如下:

(*) 'code'或者'text',把一个段定义为一个代码段,这让这个段可读并可执行,但是

不能写,同时也告诉连接器,段的类型是代码段.

(*) 'data'和'bss'定义一个数据段,类似'code',数据段被标识为可读可写,但不可执
行,'data'定义一个被初始化过的数据段,'bss'定义一个未初始化的数据段.

(*) 'rdata'声明一个初始化的数据段,它可读,但不能写.微软的编译器把它作为一个存放常量的地方.

(*) 'info'定义一个信息段,它不会被连接器放到可执行文件中去,但可以传递一些信息给连接器.比如,定义一个叫做'.drectve'信息段会让连接器把这个段内的内容解释为命令行选项.

(*) 'align='跟上一个数字,就象在'obj'格式中一样,给出段的对齐请求.你最大可以指定 64:Win32 目标文件格式没有更大的段对齐值.如果对齐请求没有被显式指定,缺省情况下,对于代码段,是 16byte 对齐,对于只读数据段,是 8byte 对齐,对
于数据段,是 4byte 对齐.而信息段缺省对齐是 1byte(即没有对齐),所以对它来说,
指定的数值没用.

如果你没有指定上述的限定符,NASM 的缺省假设是:

```
section .text    code  align=16
section .data    data  align=4
section .rdata   rdata align=8
section .bss     bss   align=4
```

任何其的段名都会跟'.text'一样被对待.

6.4 `coff': 通用目标文件格式.

'coff'输出类型产生'COFF'目标文件,可以被 DJGPP 用来连接.

'coff'提供一个缺省的输出文件扩展名'.o'.

'coff'格式支持跟'win32'同样的对于'SECTION'的扩展,除了'align'和'info'限定符不被支持.

6.5 `elf': 可执行可连接格式目标文件.

'elf'输出格式产生'ELF32'(可执行可连接格式)目标文件,这种格式用在 Linux, Unix System V 中,包括 Solaris x86, UnixWare 和 SCO Unix. 'elf'提供一个缺省的输出文件扩展名'.o'.

6.5.1 `elf'对`SECTION'操作符的扩展.

就象'obj'格式一样,'elf'允许你在'SECTION'操作符行上指定附加的信息,以控制你声明的段的类型与属性.对于标准的段名'.text','.data','.bss',NASM 都会产生缺省的类型与属性.但还是可以通过一些限定符与重新指定.

可用的限定符如下:

(*) 'alloc'定义一个段,在程序运行时,这个段必须被载入内存中,'noalloc'正好相反,比如信息段,或注释段.

(*) 'exec'把段定义为在程序运行的时候必须有执行权限.'noexec'正好相反.

(*) `write'把段定义为在程序运行时必须可写,'nowrite'正好相反.

(*) `progbits'把段定义为在目标文件中必须有实际的内容,比如象普通的代码段与数据段,'nobits'正好相反,比如'bss'段.

(*) `align='跟上一个数字,给出段的对齐请求.

如果你没有指定上述的限定符信息,NASM 缺省指定的如下:

```
section .text    progbits  alloc  exec    nowrite  align=16
section .rodata  progbits  alloc  noexec  nowrite  align=4
section .data    progbits  alloc  noexec  write    align=4
section .bss     nobits    alloc  noexec  write    align=4
section other    progbits  alloc  noexec  nowrite  align=1
```

(任何不在上述列举范围内的段,在缺省状况下,都被作为'other'段看待).

6.5.2 地址无关代码: `elf'特定的符号和 `WRT'

'ELF'规范含有足够的特性以允许写地址无关(PIC)的代码,这可以让 ELF 非常方便地共享库.尽管如此,这也意味着 NASM 如果想要成为一个能够写 PIC 的汇编器的话,必须能够在 ELF 目标文件中产生各种奇怪的重定位信息,

因为'ELF'不支持基于段的地址引用,'WRT'操作符不象它的常规方式那样被使用,所以,NASM 的'elf'输出格式中,对于'WRT'有特殊的使用目的,叫做:PIC 相关的重定位类型.

'elf'定义五个特殊的符号,它们可以被放在'WRT'操作符的右边用来实现 PIC 重定位类型.它们是`..gotpc', `..gotoff', `..got', `..plt' and `..sym'.它们的功能简要介绍如下:

(*) 使用'wrt ..gotpc'来引用以 global offset table 为基址的符号会得到当前段的起始地址到 global offset table 的距离.(`_GLOBAL_OFFSET_TABLE_'是引用 GOT 的标准符号名).所以你需要在返回结果前面加上'$$'来得到 GOT 的真实地址.

(*) 用'wrt ..gotoff'来得到你的某一个段中的一个地址实际上得到从 GOT 的的起始地址到你指定的地址之间的距离,所以这个值再加上 GOT 的地址为得到你需要的那个真实地址.

(*) 使用'wrt ..got'来得到一个外部符号或全局符号会让连接器在含有这个符号的地址的 GOT 中建立一个入口,这个引用会给出从 GOT 的起始地址到这个入口的一个距离;所以你可以加上 GOT 的地址,然后从得到的地址处载入,就会得到这个符号的真实地址.

(*) 使用'wrt ..plt'来引用一个过程名会让连接器建立一个过程连接表入口,这个引用会给出 PLT 入口的地址.你可以在上下文使用这个引用,它会产生 PC 相关的重定位信息,所以,ELF 包含引用 PLT 入口的非重定位类型

(*) 略

在 8.2 章中会有一个更详细的关于如何使用这些重定位类型写共享库的介绍

6.5.3 `elf'对`GLOBAL'操作符的扩展.

'ELF'目标文件可以包含关于一个全局符号的很多信息,不仅仅是一个地址:他可以包含符号的 size,和它的类型.这不仅仅是为了调试的方便,而且在写共享库程序的时候,这确实是非常有用的.所以,NASM 支持一些关于'GLOBAL'操作符的扩展,允许你指定这些特性.

你可以把一个全局符号指定为一个函数或一个数据对象,这是通过在名字后面加上一个冒号跟上'function'或'data'实现的.('object'可以用来代替'data')比如:

```
global    hashlookup:function, hashtable:data
```

把全局符号'hashlookup'指定为一个函数,把'hashtable'指定为一个数据对象.

你也可以指定跟这个符号关联的数据的 size,可以一个数值表达式(它可以包含 labels,甚至前向引用)跟在类型后面,比如:

```
global  hashtable:data (hashtable.end - hashtable)

hashtable:
```

```
        db this,that,theother  ; some data here
.end:
```

这让 NASM 自动计算表的长度,然后把信息放进'ELF'的符号表中.

声明全局符号的类型和 size 在写共享库代码的时候是必须的,关于这方面的更多信息,参阅 8.2.4.

6.5.4 `elf'对`COMMON'操作符的扩展.

'ELF'也允许你指定通用变量的对齐请求.这是通过在通用变量的名字和 size 的后面加上一个以冒号分隔的数字来实现的,比如,一个 doubleword 的数组以 4byte 对齐比较好:

```
common  dwordarray 128:4
```

这把 array 总的 size 声明为 128bytes,并确定它对齐到 4byte 边界.

6.5.5 16 位代码和 ELF

'ELF32'规格不提供关于 8 位和 16 位值的重定位,但 GNU 的连接器'ld'把这作为一个扩展加进去了.NASM 可以产生 GNU 兼容的重定位,允许 16 位代码被'ld'以'ELF'格式进行连接.如果 NASM 使用了选项'-w+gnu-elf-extensions',如果一个重定位被产生的话,会有一条警告信息.

6.6 `aout': Linux `a.out' 目标文件

'aout'格式产生'a.out'目标文件,这种格式在早期的 Linux 系统中使用(现在的 Linux 系统一般使用 ELF 格式,参阅 6.5),这种格式跟其他的'a.out'目标文件有所不同,文件的头四个字节的魔数不一样;还有,有些版本的'a.out',比如 NetBSD 的,支持地址无关代码,这一点,Linux 的不支持.

'a.out'提供的缺省文件扩展名是'.o'.

'a.out'是一种非常简单的目标文件格式.它不支持任何特殊的操作符,没有特殊的符号,不使用'SEG'或'WRT',对于标准的操作符也没有任何扩展.它只支持三个标准的段名'.text','.data','.bss'.

6.7 `aoutb': NetBSD/FreeBSD/OpenBSD `a.out'目标文件.

'aoutb'格式产生在 BSD unix,NetBSD,FreeBSD,OpenBSD 系统上使用的'a.out'目标文件. 作为一种简单的目标文件,这种格式跟'aout'除了开头四字节的魔数不一样,其他完全相同.但是,'aoutb'格式支持跟 elf 格式一样的地址无关代码,所以你可以使用它来写'BSD'共享库.

'aoutb'提供的缺省文件扩展名是'.o'.

'aoutb'不支持特殊的操作符,没有特殊的符号,只有三个殊殊的段名'.text','.data'和'.bss'.但是,它象 elf 一样支持'WRT'的使用,这是为了提供地址无关的代码重定位类型.关于这部分的完整文档,请参阅 6.5.2

'aoutb'也支持跟'elf'同样的对于'GLOBAL'的扩展:详细信息请参阅 6.5.3.

6.8 `as86': Minix/Linux `as86'目标文件.

Minix/Linux 16 位汇编器'as86'有它自己的非标准目标文件格式. 虽然它的链接器'ld86'产生跟普通的'a.out'非常相似的二进制输出,在'as86'跟'ld86'之间使用的目标文件格式并不是'a.out'.

NASM 支持这种格式,因为它是有用的,'as86'提供的缺省的输出文件扩展名是'.o'

'as86'是一个非常简单的目标格式(从 NASM 用户的角度来看).它不支持任何特殊的操作符,符号,不使用'SEG'或'WRT',对所有的标准操作符也没有任何扩展.它只支持三个标准的段名:'.text','.data',和'.bss'.

6.9 `rdf': 可重定位的动态目标文件格式.

'rdf'输出格式产生'RDOFF'目标文件.'RDOFF'(可重定位的动态目标文件格式)
`RDOFF'是 NASM 自产的目标文件格式,是 NASM 自已设计的,它被反映在汇编器的内部结构中.

'RDOFF'在所有知名的操作系统中都没有得到应用.但是,那些正在写他们自己的操作系统的人可能非常希望使用'RDOFF'作为他们自己的目标文件格式,因为'RDOFF'被设计得非常简单,并含有很少的冗余文件头信息.

NASM 的含有源代码的 Unix 包和 DOS 包中都含有一个'rdoff'子目录,里面有一套 RDOFF 工具:一个 RDF 连接器,一个 RDF 静态库管理器,一个 RDF 文件 dump 工具,还有一个程序可以用来在 Linux 下载入和执行 RDF 程序.

'rdf'只支持标准的段名'.text','.data','.bss'.

6.9.1 需要一个库: `LIBRARY'操作符.

'RDOFF'拥有一种机制,让一个目标文件请求一个指定的库被连接进模块中,可以是在载入时,也可以是在运行时连接进来.这是通过'LIBRARY'操作符完成的,它带有一个参数,即这个库的名字:

```
library  mylib.rdl
```

6.9.2 指定一个模块名称: `MODULE'操作符.

特定的'RDOFF'头记录被用来存储模块的名字.它可以被用在运行时载入器作动态连接.'MODULE'操作符带有一个参数,即当前模块的名字:

```
module  mymodname
```

注意,当你静态连接一个模块,并告诉连接器从输出文件中除去符号时,所有的模块名字也会被除去.为了避免这种情况,你应当在模块的名字前加一个'$',就像:

```
module  $kernel.core
```

6.9.3 `rdf'对`GLOBAL'操作符的扩展.

'RDOFF'全局符号可以包含静态连接器需要的额外信息.你可以把一个全局符号标识为导出的,这就告诉连接器不要把它从目标可执行文件中或库文件中除去.就象在'ELF'中一样,你也可以指定一个导出符号是一个过程或是一个数据对象.

在名字的尾部加上一个冒号和'exporg',你就可以让一个符号被导出:

```
global  sys_open:export
```

要指定一个导出符号是一个过程(函数),你要在声明的后南加上'proc'或'function'

```
global  sys_open:export proc
```

相似的,要指定一个导出的数据对象,把'data'或'object'加到操作符的后面:

```
global  kernel_ticks:export data
```

6.10 `dbg': 调试格式.

在缺省配置下,'dbg'输出格式不会被构建进 NASM 中.如果你是从源代码开始构建你自己的 NASM 可执行版本,你可以在'outform.h'中定义'OF_DBG'或在编译器的命令行上定义,这样就可以得到'dbg'输出格式.

'dbg'格式不输出一个普通的目标文件;它输出一个文本文件,包含有一个关于到输出格式的最终模块的转化动作的列表.它主要是用于帮助那些希望写自己的驱动程序的用户,这样他们就可以得到一个关于主程序的各种请求在输出中的形式的完整印象.

对于简单的文件,可以简单地象下面这样使用:

```
nasm -f dbg filename.asm
```

这会产生一个叫做'filename.dgb'的诊断文件.但是,这在另一些目标文件上可能工作得并不是很好,因为每一个目标文件定义了它自己的宏(通常是用户级形式的操作符),而这些宏在'dbg'格式中并没有定义.因此,运行 NASM 两遍是非常有用的,这是为了对选定的源目标文件作一个预处理:

```
nasm -e -f rdf -o rdfprog.i rdfprog.asm
nasm -a -f dbg rdfprog.i
```

这先把'rdfprog.asm 先预处理成'rdfprog.i',让 RDF 特定的操作符被正确的转化成原始形式.然后,被预处理过的源程序被交给'dbg'格式去产生最终的诊断输出.

这种方式对于'obj'格式还是不能正确工作的,因为'obj'的'SEGMENT'和'GROUP'操作符在把段名与组名定义为符号的时候会有副作用;所以程序不会被汇编.如果你确实需要 trace 一个 obj 的源文件,你必须自己定义符号(比如使用'EXTERN')

'dbg'接受所有的段名与操作符,并把它们全部记录在自己的输出文件中.

第七章: 编写 16 位代码 (DOS, Windows 3/3.1)

本章将介绍一些在编写运行在'MS-DOS'和'Windows 3.x'下的 16 位代码的时候需要用到的一些常见的知识.涵兽了如果连接程序以生成.exe 或.com 文件,如果编写.sys 设备驱动程序,以及 16 位的汇编语言代码与 C 编译器和 Borland Pascal 编译器之间的编程接口.

7.1 产生'.EXE'文件.

DOS 下的任何大的程序都必须被构建成'.EXE'文件,因为只有'.EXE'文件拥有一种内部结构可以突破 64K 的段限制.Windows 程序也需要被构建成'.EXE'文件,因为 Windows 不支持'.COM'格式.

一般的,你是通过使用一个或多个'obj'格式的'.OBJ'目标文件来产生'.EXE'文件的,用连接器把它们连接到一起.但是,NASM 也支持通过'bin'输出格式直接产生一个简单的 DOS '.EXE'文件(通过使用'DB'和'DW'来构建 exe 文件头),并提供了一组宏帮助做到这一点.多谢 Yann Guidon 贡献了这一部分代码.

在 NASM 的未来版本中,可能会完全支持'.EXE'文件.

7.1.1 使用'obj'格式来产生'.EXE'文件.

本章选描述常见的产生'.EXE'文件的方法:把'.OBJ'文件连接到一起.

大多数 16 位的程序语言包都附带有一个配套的连接器,如果你没有,有一个免费的叫做 VAL 的连接器,在`x2ftp.oulu.fi'上可以以'LZH'包的格式得到.也可以在`ftp.simtel.net'上得到. 另一个免费的 LZH 包(尽管这个包是没有源代码的),叫做 FREELINK,可以在`www.pcorner.com'上得到. 第三个是'djlink',是由 DJ Delorie 写的,可以在`www.delorie.com'上得到. 第四个 'ALINK', 是由 Anthony A.J. Williams 写的,可以在`alink.sourceforge.net'上得到.

当把多个'.OBJ'连接进一个'.EXE'文件中的时候,你需要保证它们当中有且仅有一个含有程序入口点(使用'obj'格式定义的特殊符号'..start'参阅 6.2.6).如果没有模块定义入口点,连接器就不知道在输出文件的文件头中为入口点域赋什么值,如果有多个入口被定义,连接器就不知道到底该用哪一个.

一个关于把 NASM 源文件汇编成'.OBJ'文件,并把它连接成一个'.EXE'文件的例子在这里给出.它演示了定义栈,初始化段寄存器,声明入口点的基本做法.这个文件也在 NASM 的'test'子目录中有提供,名字是'objexe.asm'.

```
segment code

..start:
```

```
        mov     ax,data
        mov     ds,ax
        mov     ax,stack
        mov     ss,ax
        mov     sp,stacktop
```

这是一段初始化代码，先把 DS 寄存器设置成指定数据段，然后把‘SS’和‘SP’寄存器设置成指定提供的栈。注意，这种情况下，在'mov ss,ax'后，有一条指令隐式地把中断关闭掉了，这样抗敌，在载入 'SS'和‘SP’的过程中就不会有中断发生，并且没有可执行的栈可用。

还有，一个特殊的符号'..start'在这段代码的开头被定义，它表示最终可执行代码的入口点。

```
        mov     dx,hello
        mov     ah,9
        int     0x21
```

上面是主程序：在'DS:DX'中载入一个指向欢迎信息的指针('hello'隐式的跟段‘data'相关联,' data'在设置代码中已经被载入到‘DS‘寄存器中，所以整个指针是有效的)，然后调用 DOS 的打印字符串功能调用。

```
        mov     ax,0x4c00
        int     0x21
```

这两句使用另一个 DOS 功能调用结束程序。

```
segment data

hello:  db      'hello, world', 13, 10, '$'
```

数据段中含有我们想要显示的字符串。

```
segment stack stack
        resb 64
stacktop:
```

上面的代码声明一个含有 64bytes 的未初始化栈空间的堆栈段，然后把指针' stacktop'指向它的顶端。操作符'segment stack stack'定义了一个叫做‘stack'的段，同时它的类型也是'STACK'.后者并不一定需要，但是连接串可能会因为你的程序中没有段的类型为'STACK'而发出警告。

上面的文件在被编译为'.OBJ'文件中，会自动连接成为一个有效的'.EXE'文件，当运行它时会打印出'hello world'，然后退出。

7.1.2 使用’ bin’格式来产生`.EXE’文件。

’.EXE’文件是相当简单的，所以可以通过编写一个纯二进制文件然后在前面连接上一个 32bytes 的头就可以产生一个’.exe’的文件了。这个文件头也是相当简单，它可以通过使用 NASM 自己的’DB’和’DW’命令来产生，所以你可以使用’bin’输出格式直接产生’.EXE’文件。

在 NASM 的包中，有一个’misc’子目录，这是一个宏文件’exebin.mac’。它定义了三个宏`EXE_begin’,`EXE_stack’和`EXE_end’.

要通过这种方法产生一个’.EXE’文件，你应当开始的时候先使用’%include’载入’exebin.mac’宏包到你的源文件中。然后，你应当使用’EXE_begin’宏(不带任何参数)来产生文件头数据。然后像平常一样写二进制格式的代码-你可以使用三种标准的段’.text’,’.data’,’.bss’.在文件的最后，你应当调用’EXE_end’宏(还是不带任何参数)，它定义了一些标识段 size 的符号，而这些宏会由’EXE_begin’产生的文件头代码引用。

在这个模块中，你最后的代码是写在’0x100’开始的地址处的，就像是’.COM’文件-实际上，如果你剥去那个 32bytes 的文件头，你就会得到一个有效的’.COM’程序。所有的段基址是相同的，所以程序的大小被限制在 64K 的范围内，这还是跟一个’.COM’文件相同。’ORG’操作符是被’EXE_begin’宏使用的，所以你不必自己显式的使用它

你可以直接使用你的段基址，但不幸的是，因为这需要在文件头中有一个重定位，事情就会变得更复杂。所以你应当从’CS’中拷贝出一个段基址。

进入你的’.EXE’文件后，’SS:SP’已经被正确的指向一个 2Kb 的栈顶。你可以通过调用’EXE_stack’宏来调整缺省的 2KB 的栈大小。比如，把你的栈 size 改变到 64bytes,你可以调用’EXE_stack 64’

一个关于以这种方式产生一个’.EXE’文件的例子在 NASM 包的子目录’test’中,名字是’binexe.asm’

7.2 产生`.COM’文件

一个大的 DOS 程序最好是写成’.EXE’文件，但一个小的程序往往最好写成’.COM’文件。’.COM’文件是纯二进制的，所以使用’bin’输出格式可以很容易的地产生。

7.2.1 使用`bin’格式产生`.COM’ 文件。

’.COM’文件预期被装载到它们所在段的’100h’偏移处(尽管段可能会变)。然后从 100h 处开始执行，所以要写一个’.COM’程序，你应当象下面这样写代码：

```
        org 100h

section .text

start:
        ; put your code here

section .data

        ; put data items here

section .bss

        ; put uninitialised data here
```

'bin'格式会把'.text'段放在文件的最开始处，所以如果你需要，你可以在开始编写代码前先声明 data 和 bss 元素，代码段最终还是会放到文件的最开始处。

BSS(未初始化过的数据)段本身在'.COM'文件中并不占据空间：BSS 中的元素的地址是一个指向文件外面的一个空间的一个指针，这样做的依据是在程序运行中，这样可以节省空间。所以你不应当相信当你运行程序时，你的 BSS 段已经被初始化为零了。

为了汇编上面的程序，你应当象下面这样使用命令行：

```
nasm myprog.asm -fbin -o myprog.com
```

如果没有显式的指定输出文件名，这个'bin'格式会产生一个叫做'myprog'的文件，所以你必须重新给它指定一个文件名。

7.2.2 使用`obj'格式产生`.COM'文件

如果你在写一个'.COM'文件的时候，产生了多于一个的模块，你可能希望汇编成多个'.OBJ'文件，然后把它们连接成一个'.COM'程序。如果你拥有一个能够输出'.COM'文件的连接器，你可以做到这一点。或者拥有一个转化程序(比如，'EXE2BIN')把一个'.EXE'输出文件转化为一个'.COM'文件也可。

如果你要这样做，你必须注意几件事情：

(*) 第一个含有代码的目标文件在它的代码段中，第一句必须是：'RESB 100h'。

这是为了保证代码在代码段基址的偏移'100h'处开始，这样，连接器和转化程序在产生.com 文件时，就不必调整地址引用了。其他的汇编器是使用'ORG'操作符来达到此目的的，但是'ORG'在 NASM 中对于'bin'格式来说是一个格式相关的操作符，会表达不同的含义。

(*) 你不必定义一个堆栈段。

(*) 你的所有段必须放在一个组中，这样每次你的代码或数据引用一个符号偏移时，所有的偏移值都是相对于同一个段基址的。这是因为，当一个'.COM'文件载入时，所有的段寄存器含有同一个值。

7.3 产生`.SYS'文件

MS-DOS 设备驱动-'SYS'文件-是一些纯二进制文件，跟.com 文件相似，但有一点，它们的起始地址是 0，而不是'100h'。因此，如果你用'bin'格式写一个设备程序，你不必使用'ORG'操作符，因为'bin'的缺省起始地址就是零。相似的，如果你使用'obj'，你不必在代码段的起始处使用'RESB 100h'

'.SYS'文件拥有一个文件头，包含一些指针，这些指针指向设备中完成实际工作的不同的子过程。这个结构必须在代码段的起始处被定义，尽管它并不是实际的代码。

要得到关于'.SYS'文件的更多信息，头结构中必须包含的数据，有一本以 FAQ 列表的形式给出的书可以在`comp.os.msdos.programmer'得到。

7.4 与 16 位 C 程序之间的接口。

本章介绍编写调用 C 程序的汇编过程或被 C 程序调用的汇编过程的基本方法。要做到这一点，你必须把汇编模块写成'.OBJ'文件，然后把它和你的 C 模块一起连接，产生一个混合语言程序。

7.4.1 外部符号名。

C 编译器对所有的全局符号(函数或数据)的名字有一个转化，它们被定义为在名字前面加上一个下划线，就象在 C 程序中出现的那样。所以，比如，一个 C 程序的函数'printf'对汇编语言程序中来说，应该是'_printf'。你意味着在你的汇编程序中，你可以定义前面不带下划线的符号，而不必担心跟 C 中的符号名产生冲突。

如果你觉得下划线不方便，你可以定义一个宏来替换'GLOBAL'和'EXTERN'操作符：

```
%macro  cglobal 1

  global  _%1
  %define %1 _%1

%endmacro
```

```
%macro  cextern 1

  extern  _%1
  %define %1 _%1

%endmacro
```

(这些形式的宏一次只带有一个参数；'%rep'结构可以解决这个问题)。

如果你象下面这样定义一个外部符号：

```
cextern printf
```

这个宏就会被展开成：

```
extern  _printf
%define printf _printf
```

然后，你可用把'printf'作为一个符号来引用，预处理器会在必要的时候在前面加上一个下划线。

'cglobal'宏以相似的方式工作。

7.4.2 内存模式。

NASM 没有提供支持各种 C 的内存模式的直接机制；你必须自己记住你在何种模式下工作。这意味着你自己必须跟踪以下事情：

(*) 在使用单个代码段的模式中(tiny small 和 compact)函数都是 near 的，这表示函数指针在作为一个函数参数存入数据段或压栈时，有 16 位长并只包含一个偏移域(CS 寄存器中的值从来不改变，总是给出函数地址的段地真正部分),函数调用就使用普通的 near'CALL'指令，返回使用'RETN'(在 NASM 中，它跟'RET'同义)。这意味着你在编写你自己的过程时，应当使用'RETN'返回，你调用外部 C 过程时，可以使用 near 的'CALL'指令。

(*) 在使用多于一个代码段的模块中(medium, large 和 huge)函数是 far 的，这表示函数指针是 32 位长的(包含 16 位的偏移值和紧跟着的 16 位段地址)，这种函数使用'CALL FAR'进行调用(或者'CALL seg:offset')而返回使用'RETF'。同样的，你编写自己的过程时，应当使用'RETF'，调用外部 C 过程应当使用'CALL FAR'。

(*) 在使用单个数据段的模块中(tiny, small 和 medium)，数据指针是 16 位

长的，只包含一个偏移域(’ DS ‘寄存器的值不改变，总是给出数据元素的地址的段地址部分)。

(*) 在使用多于一个数据段的模块中(compact, large 和 huge)，数据指针是 32 位长的，包含一个 16 位的偏移跟上一伟 16 位的段地址。你还是应当小心，不要随便改变了 ds 的值而没有恢复它，但是 ES 可以被随便用来存取 32 位数据指针的内容。

7.4.3 函数定义和函数调用。

16 位程序中的 C 调用转化如下所示。在下面的描述中，_caller_和_callee_分别表示调用者和被调用者。

(*) caller 把函数的参数按相反的顺序压栈，(从右到左，所以第一个参数被最后一个压栈)。

(*) caller 然后执行一个’CALL’指令把控制权交给 callee。根据所使用的内存模式，’CALL’可以是 near 或 far。

(*) callee 接收控制权，然后一般会(尽管在没有带参数的函数中，这不是必须的)在开始的时候把’ SP ‘的值赋给’ BP ‘，然后就可以把 ‘BP’作为一个基准指针用以寻找栈中的参数。当然，这个事情也有可能由 caller 来做，所以，关于’BP’的部分调用转化工作必须由 C 函数来完成。因此 callee 如果要把’BP’设为框架指针，它必须把先前的 BP 值压栈。

(*) 然后 callee 可能会以’BP’相关的方式去存取它的参数。在[BP]中存有 BP 在压栈前的那个值；下一字 word,在[BP+2]处，是返回地址的偏移域，由’CALL’指令隐式压入。在一个 small 模式的函数中。在[BP+4]处是参数开始的地方；在 large 模式的函数中，返回地址的段基址部分存在[BP+4]的地方，而参数是从[BP+6]处开始的。最左边的参数是被后一个被压入栈的,所以在’BP’的这点偏移值上就可以被取到;其他参数紧随其后,偏移地址是连续的.这样,在一个象’printf’这样的带有一定数量的参数的函数中,以相反的顺序把参数压栈意味着函数可以知道从哪儿获得它的第一个参数,这个参数可以告诉接接下来还有多少参数,和它们的类型分别是什么.

(*) callee 可能希望减小’sp’的值,以便在栈中分配本地变量,这些变量可以用’BP’负偏移来进行存取.

(*) callee 如果想要返回给 caller 一个值,应该根据这个值的大小放在’AL’,’AX’或’DX:AX’中.如果是浮点类型返回值,有时(看编译器而定)会放在’ST0’中.

(*) 一旦 callee 结束了处理,它如果分配过了本地空间,就从’BP’中恢复’SP’的值,然后把原来的’BP’值出栈,然后依据使用的内存模式使用’RETN’或’RETF’返回值.

(*) 如果 caller 从 callee 中又重新取回了控制权,函数的参数仍旧在栈中,所以它需要加一个立即常数到'SP'中去,以移除这些参数(不用执行一系列的 pop 指令来达到这个目的).这样,如果一个函数因为匹配的问题偶尔被以错误的参数个数来调用,栈还是会返回一个正常的状态,因为 caller 知道有多少个参数被压了,它会把它们正确的移除.

这种调用转化跟 Pascal 程序的调用转化是没有办法比较的(在 7.5.1 描述).pascal 拥有一个更简单的转化机制,因为没有函数拥有可变数目的参数.所以 callee 知道传递了多少参数,它也就有能力自己来通过传递一个立即数给'RET'或'RETF'指令来移除栈中的参数,所以 caller 就不必做这个事情了.同样,参数也是以从左到右的顺序被压栈的,而不是从右到左,这意味着一个编译器可以更方便地处理。

这样，如果你想要以 C 风格定义一个函数，应该以下面的方式进行：这个例子是在 small 模式下的。

```
global  _myfunc

_myfunc:
        push    bp
        mov     bp,sp
        sub     sp,0x40         ; 64 bytes of local stack space
        mov     bx,[bp+4]       ; first parameter to function

        ; some more code

        mov     sp,bp           ; undo "sub sp,0x40" above
        pop     bp
        ret
```

在巨模式下，你应该把'RET'替换成'RETF'，然后应该在[BP+6]的位置处寻找第一个参数，而不是[BP+4].当然，如果某一个参数是一个指针的话，那参数序列的偏移值会因为内存模式的改变而改变：far 指针作为一个参数时在栈中占用 4bytes，而 near 指针只占用两个字节。

另一方面，如果从你的汇编代码中调用一个 C 函数，你应该做下面的一些事情：

```
extern  _printf

      ; and then, further down...

      push    word [myint]        ; one of my integer variables
      push    word mystring       ; pointer into my data segment
```

```
        call    _printf
        add     sp,byte 4           ; `byte' saves space

        ; then those data items...

segment _DATA

myint           dw    1234
mystring        db    'This number -> %d <- should be 1234',10,0
```

这段代码在 small 内存模式下等同于下面的 C 代码：

```
int myint = 1234;
printf("This number -> %d <- should be 1234\n", myint);
```

在 large 模式下，函数调用代码可能更象下面这样。在这个例子中，假设 DS 已经含有段'_DATA'的段基址，你首先必须初始化它：

```
        push    word [myint]
        push    word seg mystring   ; Now push the segment, and...
        push    word mystring       ; ... offset of "mystring"
        call    far _printf
        add    sp,byte 6
```

这个整型值在栈中还是占用一个字的空间，因为 large 模式并不会影响到'int'数据类型的 size.printf 的第一个参数(最后一个压栈)，是一个数据指针，所以含有一个段基址和一个偏移域。在内存中，段基址应该放在偏移域后面，所以，必须首先被压栈。(当然，'PUSH DS'是一个取代'PUSH WORD SEG mystring'的更短的形式，如果 DS 已经被正确设置的话)。然后，实际的调用变成了一个 far 调用，因为在 large 模式下，函数都是被 far 调用的；调用后，'SP'必须被加上 6,而不是 4，以释放压入栈中的参数。

7.4.4 存取数据元素。

要想获得一个 C 变量的内容，或者声明一个 C 语言可以存取的变量，你只需要把变量名声明为'GLOBAL'或'EXTERN'即可。(再次提醒，就象在 7.4.1 中所介绍的，变量名前需要加上一个下划线)这样，一个在 C 中声明的变量'ini i'可以在汇编语中以下述方式存取：

```
extern _i

        mov ax,[_i]
```

而要声明一个你自己的可以被 C 程序存取的整型变量如：'extern int j'，你可

以这样做(确定你下在'_DATA'段中)：

```
global  _j

_j      dw      0
```

要存取 C 的数组，你需要知道数组元素的 size.比如，'int'变量是 2byte 长，所以如果一个 C 程序声明了一个数组'int a[10]',你可象这样存取'a[3]':'mov ax,[_a+6]'.(字节偏移 6 是通过数组下标 3 乘上数组元素的 size2 得到的。) 基于 C 的 16 位编译器的数据 size 如下：1 for `char', 2 for `short' and `int', 4 for `long' and `float', and 8 for `double'.

为了存取 C 的数据结构，你必须知道从结构的基地址到你所感兴趣的域的偏移地址。你可以通过把 C 结构定义转化为 NASM 的结构定义(使用'STRUC'),或者计算这个偏移地址然后进行相应操作。

以上述任何一种方法实现，你必须得阅读你的 C 编译器的手册去找出他是如何组织数据结构的。NASM 在它的宏'STRUC'中不给出任何对结构体成员的对齐操作，所以你可能会发现结构体类似下面的样子：

```
struct {
    char c;
    int i;
} foo;
```

可能就是 4 字节长，而不是三个字，因为'int'域会被对齐到 2byte 边界。但是，这种排布的特性在 C 编译器中很可能只是一个配置选项，使用命令行选项或者'#pragma'行。所以你必须找出你的编译器是如何实现这个的。

7.4.5 `c16.mac': 与 16 位 C 接口的帮助宏。

在 NASM 包中，在'misc'子目录下，是一个宏文件'c16.mac'。它定义了三个宏'proc','arg'和'endproc'。这些被用在 C 风格的过程定义中，它们自动完成了很多工作，包括对调用转化的跟踪。

(另外一种选择是，TASM 兼容模式的'arg'现在也被编译进了 NASM 的预处理器，详见 4.9)

关于在汇编函数中使用这个宏的一个例子如下：

```
proc    _nearproc

%$i     arg
%$j     arg
```

```
        mov     ax,[bp + %$i]
        mov     bx,[bp + %$j]
        add     ax,[bx]

endproc
```

这把'_nearproc'定义为一个带有两个参数的一个过程，第一个('i')是一个整型数，第二个('j')是一个指向整型数的指针，它返回'i+*j'。

注意，'arg'宏展开的第一行有一个'EQU'，而且因为在宏调用的前面的那个label 在宏展开后被加在了第一行的前面，所以'EQU'能否工作取决于'%$i'是否是一个关于'BP'的偏移值。同时一个对于上下文来说是本地的 context-local 变量被使用，它被'proc'宏压栈，然后被'endproc'宏出栈，所以，在后来的过程中，同样的参数名还是可以使用，当然，你不一定要这么做。

宏在缺省状况下把过程代码设置为 near 函数(tiny,small 和 compact 模式代码)，你可以通过代码'%define FARCODE'产生 far 函数(medium, large 和 huge 模式代码)。这会改变'endproc'产生的返回指令的类型，还会改变参数起始位置的偏移值。这个宏在设置内容时，本质上并依赖数据指针是 near 或 far。

'arg'可以带有一个可选参数，给出参数的 size。如果没有 size 给出，缺省设置为2，因为绝大多数函数参数会是'int'类型。

上面函数的 large 模式看上去应该是这个样子：

```
%define FARCODE

proc    _farproc

%$i     arg
%$j     arg     4
        mov     ax,[bp + %$i]
        mov     bx,[bp + %$j]
        mov     es,[bp + %$j + 2]
        add     ax,[bx]

endproc
```

这利用了'arg'宏的参数定义参数的 size 为 4,因为'j'现在是一个 far 指针。当我们从'j'中载入数据时，我们必须同时载入一个段基址和一个偏移值。

第八章：编写 32 位代码(Unix, Win32, DJGPP)

本章主要介绍在编写运行在 Win32 或 Unix 下的 32 位代码，或与 Unix 风格的编译器，比如 DJGPP 连接的代码时，通常会碰到的一些问题。这里包括如何编写与 32 位 C 函数连接的汇编代码，如何为共享库编写地址无关的代码。

几乎所有的 32 位代码，即在实际使用中的所有运行在'Win32','DJGPP'和所有 PC Unix 变体都运行 在_flat_内存模式下。这意味着段寄存器和页已经被正确设置，以给你一个统一的 32 位的 4Gb 的地址空间，而不管你当前工作在哪个段下，而且你应当完全忽略所有的段寄存器。当写一个平坦(flat)模式的程序代码时，你从来不必使用段重载或改变段寄存器，而且你传给'CALL',和'JMP'的代码段地址，你存取你的变量时使用的数据段地址，你存取局部变量和函数参数时的堆栈段地址实际上都在同一个地址空间中。每一个地址都是 32 位长，只含有一个偏移域

8.1 与 32 位 C 代码之间的接口。

在 7.4 中有很多关于与 16 位 C 代码之间接口的讨论，这些东西有很多在 32 位代码中仍有用。但已经不必担心内存模式和段的问题了，这把问题简化了很多。

8.1.1 外部符号名。

大多数 32 位的 C 编译器共享 16 位编译器的转化机制，即它们定义的所有的全局符号(函数与数据)的名字在 C 程序中出现时由一个下划线加上名字组成。但是，并不是他们中的所有的都这样做::'ELF'标准指出 C 符号在汇编语言中不含有一个前导的下划线。

老的 Linux'a.out'C 编译器，所有的'Win32'编译器，'DJGPP'和'NetBSD''FreeBSD'都使用前导的下划线；对于这些编译器来讲，7.4.1 中给出的宏'cextern'和'cglobal'还会正常工作。对于'ELF'来讲，下划线是没有必要的。

8.1.2 函数定义和函数调用。

32 位程序中的 C 调用转化如下所述。在下面的描述中，_caller_和_callee_用来 o 表示调用函数和被调用函数。

(*) caller 把函数的参数按相反的顺序(从右到左，这样的话，第一个参数被最后一个压栈)依次压栈

(*) 然后,caller 执行一个 near'CALL'指令把控制权传给 callee。

(*) callee 接受控制权，然后一般会(但这实际上不是必须的，如果函数不需要存取它的参数就不用)开始先存储'ESP'的值到'EBP'中，这样就可以使用'EBP'作为一个基指针去栈中寻找参数。但是，这一点也可以放在 caller 中

做，所以，调用转化中'EBP'必须被 C 函数保存起来。因为 callee 要把'EBP'设置为一个框架指针来使用，它必须把先前的值给保存起来。

(*) 然后，callee 就可以通过与'EBP'相关的方式来存取它的参数了。在[EBP]处的双字拥有刚刚被压栈的'EBP'的前一个值；接下来的双字，在[EBP+4]TH ,是被'CALL'指令隐式压入的返回地址。后面才是参数开始的地方，在[EBP+8]处。因为最左边的参数最后一个被压栈，在[EBP]的这个偏移地址上就可以被取得；剩下的参数依次存在后面，以连续增长的偏移值存放。这样，在一个如'printf'的带有一定数量参数的函数中，以相反的顺序把参数压栈意味着函数可以知道到哪儿去找它的第一个参数，这个参数可以告诉它总共有多少参数，它们的类型是什么。

(*) callee 可能也希望能够再次减小'ESP'的值，以为本地变量开辟本地空间，这些变量然后就可以通过'EBP'的负偏移来获取。

(*) callee 如果需要返回给 caller 一个值，需要根据这个值的 size 把它放在'AL','AX'或'EAX'中。浮点数在'ST0'中返回。

(*) 一旦 callee 完成了处理，如果它定义的局部栈变量，它就从'EBP'中恢复'ESP'，然后弹出前一个'EBP'的值，并通过'RET'返回。

(*) 当 caller 从 callee 那里取回了控制权，函数的参数还是放在栈中，所以，它通常给'ESP'加上一个立即常数以移除参数(而不是执行一系列的'pop'指令)。这样，如果一个函数如果因为意外，使用了错误的参数个数，栈还是会返回到正常状态，因为 caller 知道多少参数被压栈了，并可以正确的移除。

对于 Win32 程序使用的 Windows API 调用，有另一个可选的调用转化，对于那些被 Windows API 调用的函数(称为 windows 过程)也一样：他们遵循一个被微软叫做'__stdcall'的转化。这跟 Pascal 的转化比较接近，在这里，callee 通过给'RET'指令传递一个参数来清除栈。但是，参数还是以从右到左的顺序被压栈。

这样，你可以象下面这样定义一个 C 风格的函数：

```
global  _myfunc

_myfunc:
        push    ebp
        mov     ebp,esp
        sub     esp,0x40        ; 64 bytes of local stack space
        mov     ebx,[ebp+8]     ; first parameter to function

        ; some more code
```

```
        leave                   ; mov esp,ebp / pop ebp
        ret
```

另一方面，如果你要从你的汇编代码中调用一个 C 函数，你可以象下面这样写代码：

```
extern  _printf

        ; and then, further down...

        push    dword [myint]   ; one of my integer variables
        push    dword mystring  ; pointer into my data segment
        call    _printf
        add     esp,byte 8      ; `byte' saves space

        ; then those data items...

segment _DATA

myint       dd   1234
mystring    db   'This number -> %d <- should be 1234',10,0
```

这段代码等同于下面的 C 代码：

```
int myint = 1234;
printf("This number -> %d <- should be 1234\n", myint);
```

8.1.3 获取数据元素。

要想获取一个 C 变量的内容，或者声明一个 C 可以获取的变量，你必须把这个变量声明为'GLOBAL'或'EXTERN'(再次提醒，变量名前需要加上一个下划线，就象 8.1.1 中所描述的)，这样，一个被声明为'int i'的 C 变量可以从汇编语言中这样获取：

```
extern _i
        mov eax,[_i]
```

而要定个一个 C 程序可以获取的你自己的变量'extern int j',你可以这样做(确定你正在'_DATA'中)

```
        global _j
_j      dd 0
```

要获取 C 数组，你必须知道数组的元素的 size。比如，'int'变量是 4bytes 长，所以，如果一个 C 程序声明了一个数组'int a[10]',你可以使用代码'mov ax,

[_a+12]来存取变量'a[3]'。(字节偏移 12 是通过数组下标 3 乘上数组元素的 size 4 得到的)。基于 C 的 32 位编译器上的数据的 size 如下：1 for `char', 2 for `short', 4 for `int', `long' and `float', and 8for `double'. Pointers, 32 位的地址也是 4 字节长。

要获取 C 的数据结构体，你必须知道从结构体的基地址到你所需要的域之间的偏移值。你可以把 C 的结构体定义转化成 NASM 的结构体定义(使用'STRUC'),或者计算得到这个偏移值，然后使用它。

以上面任何一种方式实现，你都需要阅读你的 C 编译器的手册找出它是如何组织结构体数据的。NASM 在它的'STRUC'宏中没有给出任何特定的对齐规则，所以如果 C 编译器产生结构体，你必须自己指定对齐规则。你可能发现类似下面的结构体：

```
struct {
    char c;
    int i;
} foo;
```

可能是 8 字节长，而不是 5 字节，因为'int'域会被对齐到 4bytes 边界。但是，这种排布特性有时会是 C 编译器的一个配置选项，可以使用命令行选项或'#progma'行来实现，所以你必须找出你自己的编译器是如何做的。

8.1.4 `c32.mac': 与 32 位 C 接口的帮助宏。

在 NASM 的包中，在'misc'子目录中，有一个宏文件'c32.mac'。它定义了三个宏：'proc','arg'和'endproc'。它们被用来定义 C 风格的过程，它们会自动产生很多代码，并跟踪调用转化过程。

使用这些宏的一个汇编函数的例子如下：

```
proc    _proc32

%$i     arg
%$j     arg
        mov     eax,[ebp + %$i]
        mov     ebx,[ebp + %$j]
        add     eax,[ebx]

endproc
```

它把函数'_proc32'定义成一个带有两个参数的过程，第一个('i')是一个整型数，第二个('j')是一个指向整型数的指针，它返回'i+*j'。

注意，宏'arg'展开后的第一行有个'EQU'，因为在宏调用行的前面的那个 label 被加到了第一行上，'EQU'行就可以正常工作了，它把'%Si'定义为一个以'BP'为基址的偏移值。一个 context-local 变量在这里被使用，被'proc'宏压栈，然后被'endproc'宏出栈，所以，同样的参数名在后来的过程中还是可以使用，当然你不一定要那样做。

'arg'带有一个可选的参数，给出参数的 size。如果没有 size 给出，缺省的是 4，因为很多函数参数都会是'int'类型或者是一个指针。

8.2 编写 NetBSD/FreeBSD/OpenBSD 和 Linux/ELF 共享库

'ELF'在 Linux 下取代了老的'a.out'目标文件格式，因为它包含对于地址无关代码(PIC)的支持，这可以让编写共享库变得很容易。NASM 支持'ELF'的地址无关代码特性，所以你可以用 NASM 来编写 Linux 的'ELF'共享库。

NetBSD,和它的近亲 FreeBSD,OpenBSD,采用了一种不同的方法，它们把 PIC 支持做进了'a.out'格式。NASM 支持这些格式，把它们叫做'aoutb'输出格式，所以你可以在 NASM 下写 BSD 的共享库。

操作系统是通过把一个库文件内存映射到一个运行进程的地址空间上的某一个点来实现载入 PIC 共享库的。所以，库的代码段内容必须不依赖于它被载入到了内存的什么地方。

因此，你不能通过下面的代码得到你的变量：

```
mov      eax,[myvar]              ; WRONG
```

而是通过连接器提供一片内存空间，这片空间叫做全局偏移表(GOT)；GOT 被放到离你库代码的一个常量距离值的地方，所以如果你发现了你的库被载入到了什么地方(这可以通过使用'CALL'和'POP'指令而得到)，你可以得到 GOT 中的地址，然后你就可以通过这个连接器产生的在 GOT 中的入口来载入你的变量的地址。

而 PIC 共享库的数据段就没有这些限制了：因为数据段是可定局的，它必须被拷贝到内存中，而不是仅仅从库文件中作一个映射，所以一旦它被拷贝进来，它就可以被重定位。所以你可以把一些常规的在数据段中重定位的类型用进来，而不必担心会有什么错误发生。

8.2.1 取得 GOT 中的地址。

每个在你的共享库中的代码模块都应当把 GOT 定义为一个导出符号：

```
extern  _GLOBAL_OFFSET_TABLE_    ; in ELF
extern  __GLOBAL_OFFSET_TABLE_   ; in BSD a.out
```

在你的共享库中，那些需要获取你的 data 或 BSS 段中数据的函数，你必须在它们的开头先计算 GOT 的地址。这一般以如下形式编写这个函数：

```
func:   push    ebp
        mov     ebp,esp
        push    ebx
        call    .get_GOT
.get_GOT:
        pop     ebx
        add     ebx,_GLOBAL_OFFSET_TABLE_+$$-.get_GOT wrt ..gotpc

        ; the function body comes here

        mov     ebx,[ebp-4]
        mov     esp,ebp
        pop     ebp
        ret
```

(对于 BSD, 符号`_GLOBAL_OFFSET_TABLE'开头需要两个下划线。)

这个函数的头两行只是简单的标准的 C 风格的开头，用于设置栈框架，最后的三行是标准的 C 风格的结尾，第三行，和倒数第四行，分别保存和恢复'EBS'寄存器，因为 PIC 共享库使用这个寄存器保存 GOT 的地址。

最关键的是'CALL'指令和接下来的两行代码。'CALL'和'POP'一起用来获得 .get_GOT 的地址，不用进一步知道程序被载入到什么地方(因为 call 指令是解码成跟当前的位置相关)。‘ADD’指令使用了一个特殊的 PIC 重定位类型：GOTPC 重定位。通过使用限定符'WRT ..gotpc',被引用的符号(这里是`_GLOBAL_OFFSET_TABLE_',一个被赋给 GOT 的特殊符号)被以从段起始地址开始的偏移的形式给出。(实际上，‘ELF’把它编码为从‘ADD’的操作数域开始的一个偏移，但 NASM 把它简化了，所以你在‘ELF’和‘BSD’中可以用同样的方式处理。)所以，这条指令然后加上段起始地址，然后得到 GOT 的真正的地址。然后减去'.get_GOT'的值，当这条指令执行结束的时候，'EBX'中含有'GOT'的值。

如果你不理解上面的内容，也不用担心：因为没有必要以第二种方式来获得 GOT 的地址，所以，你可以把这三条指令写成一个宏，然后就可以安全地忽略它们：

```
%macro  get_GOT 0

        call    %%getgot
  %%getgot:
        pop     ebx
```

```
        add     ebx,_GLOBAL_OFFSET_TABLE_+$$-%%getgot wrt ..gotpc

    %endmacro
```

8.2.2 寻址你的本地数据元素。

得到 GOT 后，你可以使用它来得到你的数据元素的地址。大多数变量会在你声明过的段中；它们可以通过使用'..gotoff'来得到。它工作的方式如下：

```
        lea     eax,[ebx+myvar wrt ..gotoff]
```

表达式'myvar wrt ..gotoff'在共享库被连接进来的时候被计算，得到从 GOT 地始地址开始的变量'myvar'的偏移值。所以，把它加到上面的'EBX'中,并把它放到'EAX'中.

如果你把一些变量声明为'GLOBAL',而没有指定它们的 size 的话,它们在库中的代码模块间会被共享,但不会被从库中导出到载入它们的程序中.但们还会存在于你的常规 data 和 BSS 段中,所以通过上面的'..gotoff'机制,你可以把它们作为局部变量那样存取

注意,因为 BSD 的'a.out'格式处理这种重定位类型的一种方式,在你要存取的地址处的同一个段内必须至少有一个非本地的符号.

8.2.3 寻址外部和通用数据元素.

如果你的库需要得到一个外部变量(对库来说是外部的,并不是对它所在的一个模块),你必须使用'..got'类型得到它.'..got'类型,并不给你从 GOT 基地址到变量的偏移,给你的是从 GOT 基地址到一个含有这个变量地址的 GOT 入口的偏移,连接器会在构建库时设置这个 GOT 入口,动态连接器会在载入时在这个入口放上正确的地址.所以,要得到一个外部变量'extvar'的地址,并放到 EAX 中,你可以这样写:

```
        mov     eax,[ebx+extvar wrt ..got]
```

这会在 GOT 的一个入口上载入'extvar'的地址.连接器在构建共享库的时候,会搜集每一个'..got'类型的重定位信息,然后构建 GOT,保证它含有每一个必须的入口

通用变量也必须以这种方式被存取.

8.2.4 把符号导出给库用户.

如果你需要把符号导出给库用户,你必须把它们声明为函数或数据,如果它们是数据,你必须给出数据元素的 size.这是因为动态连接器必须为每一个导出的函数

构建过程连接表入口,还要把导出数据元素从库的数据段中移出.

所以,导出一个函数给库用户,你必须这样:

```
global  func:function           ; declare it as a function

func:   push    ebp

        ; etc.
```

而导出一个数据元素,比如数组,你必须这样写代码:

```
global  array:data array.end-array      ; give the size too

array:  resd    128
.end:
```

小心:如果你希望通过把变量声明为'GLOBAL'并指定一个 size,而导出给库用户,这个变量最终会存在于主程序的数据段中,而不是在你的库的数据段内,所以你必须通过使用'..got'机制来获取你自己的全局变量,而不是'..gotogg',就象它是一个外部变量一样(实际上,它已经变成了外部变量).

同样的,如果你需要把一个导出的全局变量的地址存入你的一个数据段中,你不能通过下面的标准方式实现:

```
dataptr:        dd      global_data_item        ; WRONG
```

NASM 会以个普通的重定位解释这段代码,在这里,'global_data_item'仅仅是一个从'.data'段(或者其他段)开始的一个偏移值;所以这个引用最终会指向你的数据段,而不是导出全局变量.

对于上面的代码,你应该这样写:

```
dataptr:        dd      global_data_item wrt ..sym
```

这时使用了一个特殊的'WRT'类型'..sym'来指示 NASM 到符号表中去寻找一个在这个地址的特定符号,而不是通过段基址重定位.

另外一种方式是针对函数的:以下面的方法引用你的一个函数:

```
funcptr:        dd      my_function
```

会给用户一个你的代码的地址,而:

```
funcptr:        dd      my_function wrt .sym
```

会给出过程连接表中的该函数的地址,这是真正的调用程序应该得到的地址.两种地址都是可行的.

8.2.5 从库外调用过程.

从你的共享库外部调用过程必须通过使用过程连接表(PLT)才能实现,PLT 被放在库载入处的一个已知的偏移地址处,所以库代码可以以一种地址无关的方式去调用 PLT.在 PLT 中有跳转到含在 GOT 中的偏移地址的代码,所以对共享库中或主程序中的函数调用可以被转化为直接传递它们的真实地址.

要调用一个外部过程,你必须使用另一个特殊的 PIC 重定位类型,'WRT ..plt'.这个比基于 GOT 的要简单得多:你只需要把调用'CALL printf'替换为 PLT 相关的版本:`CALL printf WRT ..plt'.

8.2.6 产生库文件.

写好了一些代码模块并把它们汇编成'.o'文件后,你就可以产生你的共享库了,使用下面的命令就可以:

```
ld -shared -o library.so module1.o module2.o        # for ELF
ld -Bshareable -o library.so module1.o module2.o    # for BSD
```

对于 ELF,如果你的共享库要放在系统目录'/usr/lib'或'/lib'中,那对连接器使用'-soname'可以把最终的库文件名和版本号放进库中:

```
ld -shared -soname library.so.1 -o library.so.1.2 *.o
```

然后你就可以把'library.so.1.2'拷贝到库文件目录下,然后建立一个它的符号连的妆'library.so.1'.

第九章: 混合 16 位与 32 位代码

本章将介绍一些跟非常用的地址与跳转指令相关的一些问题, 这些问题当你在编写操作系统代码时会常遇上,比如保护模式初始化过程,它需要代码操作混合的段 size,比如在 16 位段中的代码需要去修改在 32 位段中的数据,或者在不同的 size 的段之间的跳转.

9.1 混合 Size 的跳转.

最常用的混合 size 指令的形式是在写 32 位操作系统时用到的:在 16 位模式中完成你的设置,比如载入内核,然后你必须通过切入到保护模式中引导它,然后跳转到 32 位的内核起始地址处.在一个完全 32 位的操作系统中,这是你唯一需要用到混合 size 指令的地方,因为在它之间的所有事情都可以在纯 16 位代码中完成,而在它之后的所在事情都在纯 32 位代码中.

这种跳转必须指定一个 48 位的远地址,因为目标段是一个 32 位段.但是,它必须在 16 位段中被汇编,所以,仅仅如下面写代码:

```
jmp     0x1234:0x56789ABC       ; wrong!
```

不会正常工作,因为地址的偏移域部分会被截断成'0x9ABC',然后,跳转会是一个普通的 16 位远跳转.

Linux 内核的设置代码使用'as86'通过手工编码来产生这条指令,使用'DB'指令, NASM 可以比它更好些,可以自己产生正确的指令,这里是正确的做法:

```
jmp     dword 0x1234:0x56789ABC         ; right
```

'DWORD'前缀(严格地讲,它应该放在冒后的后面,因为它只是把偏移域声明为 doubleword;但是 NASM 接受任何一种形式,因为两种写法都是明确的)强制偏移域在假设你正从一个 16 段跳转到 32 位段的前提下,被处理为 far.

你可以完成一个相反的操作,从一个 32 位段中跳转到一个 16 位段,使用'word'前缀:

```
jmp     word 0x8765:0x4321      ; 32 to 16 bit
```

如果'WORD'前缀在 16 位模式下被指定,或者'DWORD'前缀在 32 位模式下被指定,它们都会被忽略,因为它们每一个都显式强制 NASM 进入一个已进进入的模式.

9.2 在不同 size 的段间寻址.

如果你的操作系统是 16 位与 32 位混合的,或者你正在写一个 DOS 的扩展,你可能

必须处理一些 16 位段和一些 32 位段.在某些地方,你可能最终要在一个 16 位段中编写能获取 32 位段中的数据的代码,或者相反.

如果你要获取的 32 位段中的数据正好在段的前 64K 的范围内,你可以通过普通的 16 位地址操作来达到目的;但是或多或少,你会需要从 16 位模式中处理 32 位的寻址.

最早的解决方案保证你使用了一个寄存器用于保存地址,因为任何在 32 位寄存器中的有效地址都被强制作为一个 32 位的地址,所以,你可以:

```
mov     eax,offset_into_32_bit_segment_specified_by_fs
mov     dword [fs:eax],0x11223344
```

这个不错,但有些笨拙(因为它浪费了一条指令和一个寄存器),如果你已经知道你的目标所在的精确偏移.x86 架构允许 32 位有效地址被指定为一个 4bytes 的偏移,所以,NASM 为什么不为些产生一个最佳的指令呢?

它可以,就象在 9.1 中一样,你只需要在地址前加上一个'DWORD'前缀,然后,它会被强制作为一个 32 位的地址:

```
mov     dword [fs:dword my_offset],0x11223344
```

同样跟 9.1 中一样,NASM 并不关心'DWORD'前缀是在段重载符前,还是这后,所以可以把代码改得好看一些:

```
mov     dword [dword fs:my_offset],0x11223344
```

不要把'DWROD'前缀放在方括号外面,它是用来控制存储在那里的数据的 size 的,而在方括号内的话,它控制地址本身的长度.这两种方式可以被很容易地区分:

```
mov     word [dword 0x12345678],0x9ABC
```

这把一个 16 位的数据放到了一个指定为 32 位偏移的地址中.

你也可以把'WORD'或'DWROD'前缀跟'FAR'前缀放到一起,来间接跳转或调用,比如:

```
call    dword far [fs:word 0x4321]
```

这条指令包含一个指定为 16 位偏移的地址,它载入了一个 48 位的远指针,(16 位段和 32 位段偏移),然后调用这个地址.

9.3 其他的混合 size 指令.

你可能需要用于获取数据的其它的方式可能就是使用字符串指令('LODSx'

'STOSx',等等)或'XLATB'指令.这些指令因为不带有任何参数,看上去好像很难在它们被汇编进 16 位段的时候使它们使用 32 位地址.

而这正是 NASM 的'a16'和'a32'前缀的目的,如果你正在 16 位段中编写'LODSB',但它是被用来获取一个 32 位段中的字符串的,你应当把目标地址载入'ESI',然后编写:

```
a32     lodsb
```

这个前缀强制地址的 size 为 32 位,意思是'LODSB'从[DS:ESI]中载入内容,而不是从[DS:SI]中.要在编写 32 位段的时候,获取在 16 位段中的字符串,相应的前缀'a16'可以被使用.

'a16'和'a32'前缀可以被运用到 NASM 指令表的任何指令上,但是他们中的大多数可以在没有这两个前缀的情况下产生所有有用的形式.这两个前缀只有在那些带有隐式地址的指令中是有效的: `CMPSx' (section B.4.27), `SCASx' (section B.4.286), `LODSx' (section B.4.141), `STOSx' (section B.4.303), `MOVSx' (section B.4.178), `INSx' (section B.4.121), `OUTSx' (section B.4.195), and `XLATB' (section B.4.334).
还有,就是变量压栈与出栈指令,(`PUSHA'和`POPF' 和更常用的`PUSH'和`POP')可以接受'a16'或'a32'前缀在堆栈段用在另一个不同 size 的代码段中的时候,强制一个特定的'SP'或'ESP'被用作栈指针,

'PUSH'和'POP',当在 32 位模式中被用在段寄存器上时,也会有一个不同的行为,它们会一次操作 4bytes,而最高处的两个被忽略,而最底部的两个给出正被操作的段寄存器的值.为了强制 push 和 pop 指令的 16 位行为,你可以使用操作数前缀'o16'

```
o16 push    ss
o16 push    ds
```

这段代码在栈空间中开辟一个 doubleword 用于存放两个段寄存器,而在一般情况下,这一个 doubleword 只会存放一个寄存器的值.

(你也可以使用'o32'前缀在 16 位模式下强制 32 位行为,但这看上去并没有什么用处.)

第十章: 答疑

本章介绍一些用户在使用 NASM 时经常遇到的普遍性问题,并给出解答.同时,如果你发现了这儿还未列出的 BUG,这儿也给出提交 bug 的方法.

10.1 普遍性的问题.

10.1.1 NASM 产生了低效的代码.

我得到了很多关于 NASM 产生了低效代码的 BUG 报告,甚至是产生错误代码,比如像指令'ADD ESP,8'产生的代码.其实这是一个经过深思熟虑设计特性,跟可预测的输出相关:NASM 看到'ADD ESP,8'时,会产生一个预留 32 位偏移的指令形式.如果你希望产生一个节约空间的指令形式,你必须写上'ADD ESP,BYTE 8'.这不是一个 BUG,至多也只能算是一个不好的特性,各人看法不同而已.

10.1.2 我的 jump 指令超出范围.

相似的,人们经常抱怨说在他们使用条件跳转指令时(这些指令缺省状况下是'short'的)经常需要跳转比较远,而 NASM 会报告说'short jump out of range',而不作长远转.

同样,这也是可预测执行的一个部分,但实际上还有一个更有实际的理由.NASM 没有办法知道它产生的代码运行的处理器的类型;所以它自己不能决定它应该产生'Jcc NEAR'类型的指令,因为它不知道它正在 386 或更高一级的处理器上工作.相反,它把可能超出范围的短'JNE'指令替换成一个很短的'JE'指令,这个指令仅仅跳过一个'JMP NEAR'指令;对于低于 386 的处理器,这是一个可行的解决方案,但是对于有较好的分支预测功能的处理器很难有较好的效果,所以可代之以'JNE NEAR'.所以,产生什么的指令还是取决于用户,而不是汇编器本身.

10.1.3 `ORG'不正常工作.

那些用'bin'格式写引导扇区代码的人们经常抱怨'ORG'没有按他们所希望的那样正常工作:为了把'0xAA55'放到 512 字节的引导扇区的末尾,使用 NASM 的人们会这样写:

```
ORG 0

; some boot sector code

ORG 510
DW 0xAA55
```

这不是 NASM 中使用'ORG'的正确方式,不会正常工作.解决这个问题的正确方法是使用

'TIMES'操作符,就象这样:

```
ORG 0

; some boot sector code

TIMES 510-($-$$) DB 0
DW 0xAA55
```

'TIME'操作符会在输出中插入足够数量的零把汇编点移到 510.这种办法还有一个好处,如果你意外地在你的引导扇区中放入了太多的内容,以致超出容量,NASM 会在汇编时检测到这个错误,并报告.所以你最终就不必重汇编并去找出错误所在.

10.1.4 `TIMES'不正常工作.

关于上面代码的另一个普遍性的问题是,有人这样写'TIMES'这一行:

```
TIMES 510-$ DB 0
```

因为'$'是一个纯数字,就像 510,所以它们相减的值也是一个纯数字,可以很好地被 TIMES 使用.

NASM 是一个模块化的汇编器:不同的组成部分被设计为可以很容易的单独重用,所以它们不会交换一些不必要的信息.结果,'BIN'输出格式尽管被'ORG'告知'.text'段应当在 0 处开始,但是不会把这条信息传给表达式的求值程序.所以对求值程序来讲,'$'不是一个纯数值:它是一个从一个段基址开始的偏移值.因为'$'和 510'之间的计算结果也不是一个纯数,而是含有一个段基址.含有一个段基址的结果是不能作为参数传递给'TIMES'的.

解决方案就象上一节所描述的,应该如下:

```
TIMES 510-($-$$) DB 0
```

在这里,'$'和'$$'是从同一个段基址的偏移,所以它们相减的结果是一个纯数,这句代码会解决上述问题,并产生正确的代码.

10.2 Bugs

我们还从来没有发布过一个带有已知 BUG 的 NASM 版本.但我们未知的 BUG 从来就是不停地出现.你发现了任何 BUG,应当首先通过在`https://sourceforge.net/projects/nasm/'(点击 bug)的'bugtracker'提交给我们,如果上述方法不行,请通过 1.2 中的某一个联系方式.

请先阅读 2.2,请不要把列在那儿的作为特性的东西作为 BUG 提交给我们.(如果你认

为这个特性很不好,请告诉我们你认为它应当被修改的原因,而不是仅仅给我们一个'这是一个 BUG')然后请阅读 10.1,请不要把已经列在那里的 BUG 提交给我们.

如果你提交一个 bug,请给我们下面的所有信息.
(这部分信息一般用户并不关心,在些省略,原文请参考 NASM 的英文文档.)

附录 A: Ndisasm

反汇编器, NDISASM

A.1 简介

反汇编器是汇编器 NASM 的一个很小的附属品.我们已经拥有一个具有完整的指令表的 x86 汇编器,如果不把这个指令表尽最大可能地利用起来,似乎很可惜,所以我们又加了一个反汇编器,它共享 NASM 的指令表(并附加上一些代码)

反汇编器仅仅产生二进制源文件的反汇编.NDISASM 不理解任何目标文件格式,就象'objdump',也不理解'DOS .EXE'文件,就象'debug',它仅仅反汇编.

A.2 开始: 安装.

参阅 1.3 的安装指令.NDISASM 就象 NASM,也有一个帮助页,如果你在一个 UNIX 系统下,你可能希望把它放在一个有用的地方.

A.3 运行 NDISASM

要反汇编一个文件,你可以象下面这样使用命令:

```
ndisasm [-b16 | -b32] filename
```

NDISASM 可以很容易地反汇编 16 位或 32 位代码,当然,前提是你必须记得给它指定是哪种方式.如果'-b'开关没有,NDISASM 缺省工作在 16 位模式下.'-u'开关也包含 32 位模式.

还有两个命令行选项,'-r'打印你正运行的 NDISASM 的版本号,'-h'给你一个有关命令行选项的简短介绍.

A.3.1 COM 文件: 指定起点地址.

要正确反汇编一个'DOS.COM'文件,反汇编器必须知道文件中的第一条指令是被装载到地址'0x100'处的,而不是 0,NDISASM 缺省地认为你给它的每一个文件都是装载到 0 处的,所以你必须告诉它这一点.

'-o'选项允许你为你正反汇编的声明一个不同的起始地址.它的参数可以是任何NASM数值格式:缺省是十进制,如果它以''$''或''0x''开头,或以''H'结尾,它是十六进制的,如果以''Q''结尾,它是8进制的,如果是''B''结尾,它是二进制的.

所以,反汇编一个'.COM'文件:

```
ndisasm -o100h filename.com
```

能够正确反汇编.

A.3.2 代码前有数据: 同步.

假设你正反汇编一个含有一些不是机器码的数据的文件,这个文件当然也含有一些机器码.NDISASM会很诚实地去研究数据段,尽它的能力去产生机器指令(尽管它们中的大多数看上去很奇怪,而且有些还含有不常见的前缀,比如:'FS OR AX, 0x240A'')然后,它会到达代码段处.

假设NDISASM刚刚完成从一个数据段中产生一堆奇怪的机器指令,而它现在的位置正处于代码段的前面一个字节处.它完全有可能以数据段的最后一个字节为开始产生另一个假指令,然后,代码段中的第一条正确的指令就看不到了,因为起点已经跳过这条指令,这确实不是很理想.

为了避免这一点,你可以指定一个'同步'点,或者可以指定你需要的同步点的数目(但NDISASM在它的内部只能处理8192个同步点).同步点的定义如下:NDISASM保证会到达这个同步点.如果它认为某条指令会跳过一个同步点,它会忽略这条指令,代之以一个'DB'.所以它会从同步点处开始反汇编,所以你可以看到你的代码段中的所有指令.

同步点是用'-s'选项来指定的:它们以从程序开始处的距离来衡量,而不是文件位置.所以如果你要从32bytes后开始同步一个'.COM'文件,你必须这样做:

```
ndisasm -o100h -s120h file.com
```

而不是:

```
ndisasm -o100h -s20h file.com
```

就象上面所描述的,如果你需要,你可以指定多个同步记号,只要重复'-s'选项即可.

A.3.3 代码和数据混合: 自动(智能)同步.

假设你正在反汇编一个'DOS'软盘引导扇区(可能它含有病毒,而你需要理解病毒,这样你就可以知道它可能会对你的系统造成什么样的损害).一般的,里面会含有'JMP'指令,然后是数据,然后接下来才是代码,所以,这很可能会让NDISASM不能在数据与代码交接处找不到正确的点,所以同步点是必须的.

另一方面,你为什么要手工指定同步点呢?你要找出来的同步点的地址,当然是可以从'JMP'中读取,然后可以用它的目标地址作为一个同步点,而 NDISADM 是否可以为你做到这一点?

答案当然是可以:使用同步开关'-a'(自动同步)或'-i'(智能同步)会启用'自动同步"模式.自动同步模式为 PC 相关的前向引用或调用指令自动产生同步点.(因为 NDISASM 是一遍的,如果它遇上一个目标地址已经被处理过的 PC 相关的跳转,它不能做什么.)

只有 PC 相关的 jump 才会被处理,因为一个绝对跳转可能通过一个寄存器(在这种情况下,NDISASM 不知道这个寄存器中含有什么)或含有一个段地址(在这种情况下,目标代码不在 NDISASM 工作的当前段中,所以同步点不能被正确的设置)

对于一些类型的文件,这种机制会自动把同步点放到所有正确的位置,可以让你不必手工放置同步点.但是,需要强调的是自动模式并不能保证找出所有的同步点,你可能还是需要手工放置同步点.

自动同步模式不会禁止你手工声明同步点:它仅仅只是把自动产生的同步点加上.同时指定'-i'和'-s'选项是完全可行的.

关于自动同步模式,另一个需要提醒的是,如果因为一些讨厌的意外,你的数据段中的一些数据被反汇编成了 PC 相关的调用或跳转指令,NDISASM 可能会很诚实地把同步点放到所有的这些位置,比如,在你的代码段中的某条指令的中间位置.同样,我们不能为此做什么,如果你有问题,你还是必须使用手工同步点,或使用'-k'选项(下面介绍)来禁止数据域的反汇编.

A.3.4 其他选项.

'-e'选项通过忽略一个文件开头的 N 个 bytes 来跳过一个文件的文件头.这表示在反汇编器中,文件头不被计偏移域中:如果你给出'-e10 -o10',反汇编器会从文件开始的 10byte 处开始,而这会对偏称域给出 10,而不是 20.

'-k'选项带有两个逗号--分隔数值参数,第一个是汇编移量,第二个是跳过的 byte 数.这是从汇编偏移量处开始计算的跳过字节数:它的用途是禁止你需要的数据段被反汇编.

A.4 Bug 和改进.

现在还没有已知的 bug.但是,如果你发现了,并有了补丁,请发往`jules@dsf.org.uk'或`anakin@pobox.com', 或者在`https://sourceforge.net/projects/nasm/'上的开发站点,我们会改进它们,请多多给我们改进和新特性的建议.

将来的计划包括能知道特定的指令运行在那种处理器上,并能标出那些对于某些处理器来说过于高级的指令(或是'FPU'指令,或是没有公开的操作符, 或是特权保护模式

指令, 或是其它).