2014年9月5日
Nasm中文手册

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80×86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。

(*) ”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

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:asm")

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

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 [-o]
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

(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

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 -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。

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"

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

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：
(*)”-O0”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
”-O0”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>”: 位移运算符。
`>”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。

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)

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

```
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

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
```

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：

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
```

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
```

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，

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

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。

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
```

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
```

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,0×40
        int     0×21

%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.

occurred."

```
        writefile 2,%1
        mov     ax,0x4c01
        int     0×21

    %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

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

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
```

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

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
   ;if 满足时接下来的代码被汇编。
%elif
   ; 当if不满足，而满足时，该段代码被汇编。
%else
   ;当跟都不满足时，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
```

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

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
```

```
    %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
```

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

”%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

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

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

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:
```

```
    %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
```

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      0×00622001
```

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。

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

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，

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
```

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

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 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

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 `%!”`”: 读取一个环境变量。

`%!” 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   ”
```

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

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
0×66，要使用32位的地址，可以加上0×67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0×66前缀，使
用16位地址的指令需要0×67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

```
    %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,0×40
          int     0×21

    %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
```

这个例子描述了一个关于在段地址0×40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

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

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

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”。

比如，下面的代码会产生longword: `0×00000104”:

```
        org    0×100
        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

(*) 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

```
        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

(*) 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

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

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

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

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.

(*) ”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”.

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:

`..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

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”.

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

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

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

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     0×21
```

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
```

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”0×100”开始的地址

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
```

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

，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
```

%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”。

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

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
```

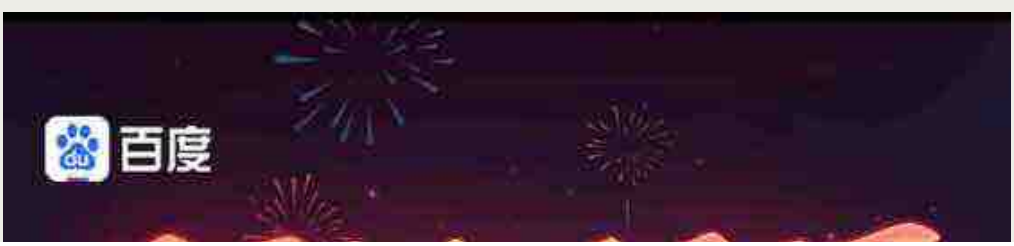
space
mov bx,[bp+4] ; first parameter to function

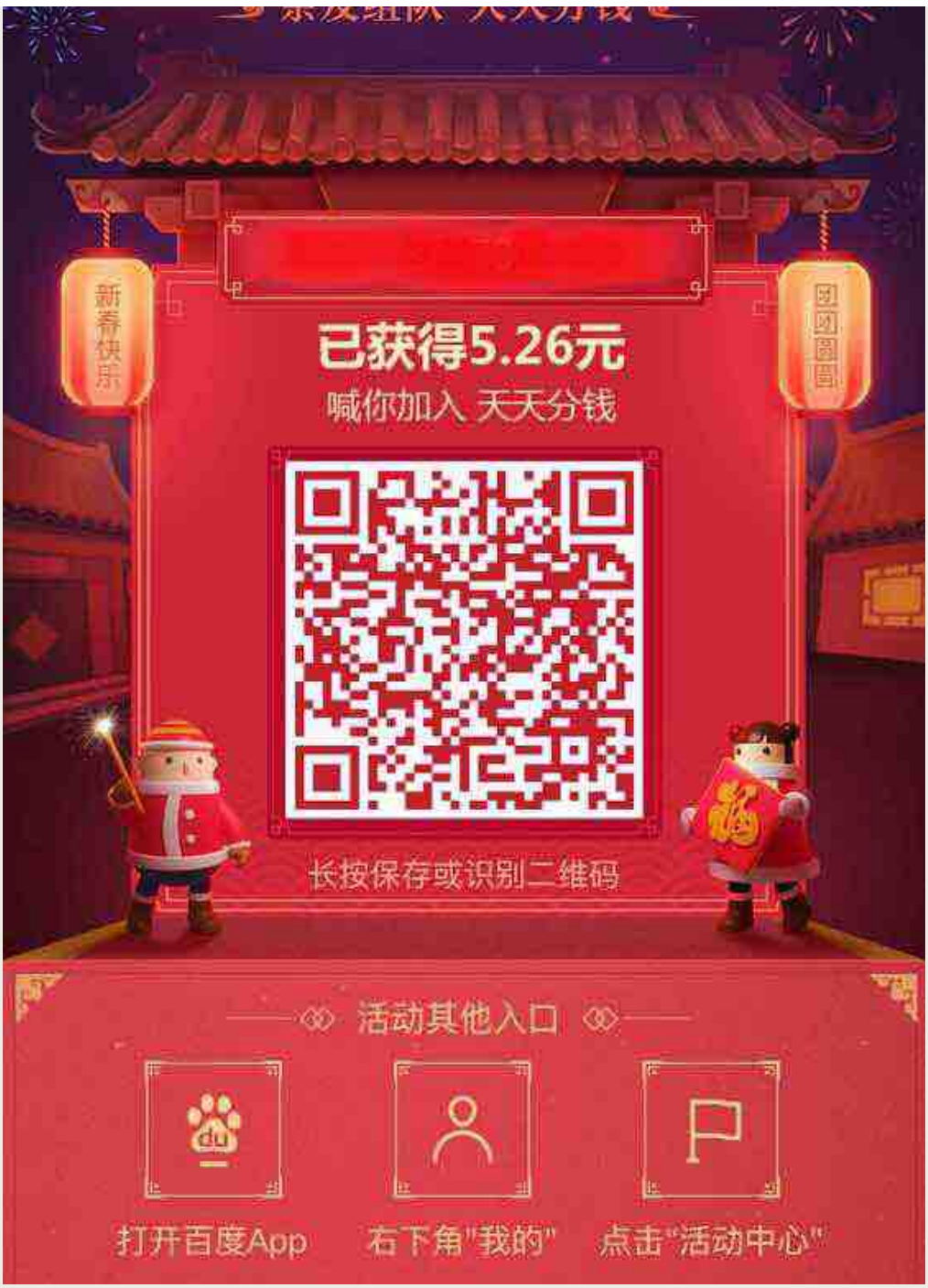
; some more code

mov sp,bp ; undo “sub sp,0×40" 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 %d %d %d
```





学步园
专注于程序开发等技术类文章

前一篇文章
最小生成树之prim算法原理

下一篇文章
Python--绘图工具matplotlib的使用

返回顶部

移动 桌面