IBM PC 汇编语言程序设计

沈美明　温冬婵　编著

清华大学出版社

目　录

第一章 基础知识

1.1 进位计数制与不同基数的数之间的转换

1.1.1 二进制数

进位计数制是一种计数的方法，习惯上最常用的是十进制计数法。一个任意的十进制数可以表示为：

$$a_n a_{n-1} \cdots a_0 . b_1 b_2 \cdots b_m$$

其含意是：

$$a_n \cdot 10^n + a_{n-1} \cdot 10^{n-1} + \cdots + a_0 \cdot 10^0 + b_1 \cdot 10^{-1} + b_2 \cdot 10^{-2} + \cdots + b_m \cdot 10^{-m}$$

其中 $a_i(i=0,1,\cdots,n)$，$b_j(j=1,2,\cdots,m)$是 0,1,2,3,4,5,6,7,8,9 十个数码中的一个。

十进制数的基数为 10，即其数码的个数为 10，且遵循逢十进一的规则。上式中相应于每位数字的 10^k 称为该位数字的权，所以每位数字乘以其权所得到的乘积之和即为所表示数的值。例如：

$$12345.67 = 1 \times 10^4 + 2 \times 10^3 + 3 \times 10^2 + 4 \times 10^1 + 5 \times 10^0 + 6 \times 10^{-1} + 7 \times 10^{-2}$$

十进制数是人们最熟悉、最常用的一种数制，但它不是唯一的数制。例如计时用的时、分、秒就是按 60 进制计数的。基数为 r 的 r 进制数的值可以表示为：

$$a_n \cdot r^n + a_{n-1} \cdot r^{n-1} + \cdots + a_0 \cdot r^0 + b_1 \cdot r^{-1} + b_2 \cdot r^{-2} + \cdots + b_m \cdot r^{-m}$$

其中 a_i,b_i 可以是 0,1,$\cdots$,r－1 中的任一个数码，r^k 则为各位数相应的权。

计算机中为便于存储及计算的物理实现，采用了二进制数。二进制数的基数为 2，只有 0、1 两个数码，并遵循逢二进一的规则，它的各位权是以 2^k 表示的，因此二进制数 $a_n a_{n-1} \cdots a_0 . b_1 b_2 \cdots b_m$ 的值是：

$$a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \cdots + a_0 \cdot 2^0 + b_1 \cdot 2^{-1} + b_2 \cdot 2^{-2} + \cdots + b_m \cdot 2^{-m}$$

其中 a_i,b_i 为 0,1 两个数码中的一个。例如：

$$101101_2 = 1 \times 2^5 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0 = 45_{10}$$

其中数的下标表示该数的基数 r，即二进制的 101101 与十进制的 45 等值。

n 位二进制数可以表示 2^n 个数。例如 3 位二进制数可以表示 8 个数，它们是：

二 进 制 数	000	001	010	011	100	101	110	111
相应的十进制数	0	1	2	3	4	5	6	7

而 4 位二进制数则表示十进制的 0～15 共 16 个数如下：

二 进 制 数	0000	0001	0010	0011	0100	0101	0110	0111
相应的十进制数	0	1	2	3	4	5	6	7
二 进 制 数	1000	1001	1010	1011	1100	1101	1110	1111
相应的十进制数	8	9	10	11	12	13	14	15

为便于人们阅读及书写，经常使用八进制数或十六进制数来表示二进制数。它们的基数和数码表示如表 1.1 所示。

表 1.1 几种常用的进位计数制的基数和数码

进位计数制	基 数	数 码
十六进制数	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
十进制数	10	0,1,2,3,4,5,6,7,8,9
八进制数	8	0,1,2,3,4,5,6,7,
二进制数	2	0,1

按同样的方法，读者可以很容易地掌握八进制和十六进制数的表示方法。可以看出：23_{10}可以表示为 17_{16}、27_8及 10111_2，1.375_{10}可以表示为 1.6_{16}、1.3_8 及 1.011_2 等。在计算机里，通常用数字后面跟一个英文字母来表示该数的数制。十进制数一般用 D(Decimal)、二进制数用 B(Binary)、八进制数用 O(Octal)、十六进制数用 H(Hexadecimal)来表示。例如：117D，1110101B，0075H，…。当然也可以用这些字母的小写形式，本书的后面就采用了这种表示方法。

1.1.2 二进制数和十进制数之间的转换

1.1.2.1 二进制数转换为十进制数

各位二进制数码乘以与其对应的权之和即为与该二进制数相对应的十进制数。例如：

$$1011100.10111B = 2^6 + 2^4 + 2^3 + 2^2 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-5} = 92.71875D$$

1.1.2.2 十进制数转换为二进制数

十进制数转换为二进制数的方法很多，这里只说明比较简单的降幂法及除法两种。

·降幂法

首先写出要转换的十进制数，其次写出所有小于此数的各位二进制权值，然后用要转换的十进制数减去与它最相近的二进制权值，如够减则减去并在相应位记以 1；如不够减则在相应位记以 0 并跳过此位；如此不断反复，直到该数为 0 为止。

例 1.1 N=117D，小于 N 的二进制权为：

64 32 16 8 4 2 1

对应的二进制数是 1 1 1 0 1 0 1

计算过程如下：

$$117 - 2^6 = 117 - 64 = 53 \qquad (a_6 = 1)$$

$$53 - 2^5 = 53 - 32 = 21 \qquad (a_5 = 1)$$

$$21 - 2^4 = 21 - 16 = 5 \qquad (a_4 = 1)$$

$$(a_3 = 0)$$

$$5 - 2^2 = 5 - 4 = 1 \qquad (a_2 = 1)$$

$(a_1 = 0)$

$1 - 2^0 = 1 - 1 = 0 \qquad (a_0 = 1)$

所以 N=117D=1110101B

例 1.2 N=0.8125D，小于此数的二进制权为：

0.5　0.25　0.125　0.0625

对应的二进制数是　1　　1　　0　　1

计算过程如下：

$0.8125 - 2^{-1} = 0.8125 - 0.5 = 0.3125 \qquad (b_1 = 1)$

$0.3125 - 2^{-2} = 0.3125 - 0.25 = 0.0625 \qquad (b_2 = 1)$

$(b_3 = 0)$

$0.0625 - 2^{-4} = 0.0625 - 0.0625 = 0 \qquad (b_4 = 1)$

所以 N=0.8125D=0.1101B

· 除法

把要转换的十进制数的整数部分不断除以 2，并记下余数，直到商为 0 为止。

例 1.3 N=117D

$117/2 = 58 \qquad (a_0 = 1)$

$58/2 = 29 \qquad (a_1 = 0)$

$29/2 = 14 \qquad (a_2 = 1)$

$14/2 = 7 \qquad (a_3 = 0)$

$7/2 = 3 \qquad (a_4 = 1)$

$3/2 = 1 \qquad (a_5 = 1)$

$1/2 = 0 \qquad (a_6 = 1)$

所以 N=117D=1110101B。

对于被转换的十进制数的小数部分则应不断乘以 2，并记下其整数部分，直到结果的小数部分为 0 为止。

例 1.4 N=0.8125D

$0.8125 \times 2 = 1.625 \qquad (b_1 = 1)$

$0.625 \times 2 = 1.25 \qquad (b_2 = 1)$

$0.25 \times 2 = 0.5 \qquad (b_3 = 0)$

$0.5 \times 2 = 1.0 \qquad (b_4 = 1)$

所以 N=0.8125D=0.1101B。

1.1.3 十六进制数及其与二进制、十进制数之间的转换

我们知道，在计算机内部，数的运算和存储都是采用二进制的。但是，二进制数对于人的阅读、书写及记忆都是很不方便的。十进制数虽然是人们最熟悉的一种进位计数制，但它与二进制数之间并无直接的对应关系。为了便于人们对二进制数的描述，应该选择一种易于与二进制数相互转换的数制。显然，使用 2^n 作为基数的数制是能适合人们的这种要求的，常用的有八进制数和十六进制数，我们在这里主要介绍十六进制数。

1.1.3.1 十六进制数的表示

计算机中存储信息的基本单位为一个二进制位(Bit),它可以用来表示 0 和 1 两个数码。此外,由于计算机中常用的字符是采用由 8 位二进制数组成的一个字节(Byte)来表示的,因此字节也成为计算机中存储信息的单位。计算机的字长一般都选为字节的整数倍,如 16 位、32 位、64 位等。一个字节由 8 位组成,它可以用两个四位组(又称半字节)来表示,所以用十六进制数来表示二进制数是比较方便的。

十六进制数的基数为 16,共有 16 个数码,它们是 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F。其中 A 表示十进制的 10,余类推。它们与二进制和十进制数的对应关系如下:

二进制数	0000	0001	0010	0011	0100	0101	0110	0111
十进制数	0	1	2	3	4	5	6	7
十六进制数	0	1	2	3	4	5	6	7
二进制数	1000	1001	1010	1011	1100	1101	1110	1111
十进制数	8	9	10	11	12	13	14	15
十六进制数	8	9	A	B	C	D	E	F

对应于十六进制数中各位的权是 16^k。

1.1.3.2 十六进制数和二进制数之间的转换

由于十六进制数的基数是 2 的幂,所以这两种数制之间的转换是十分容易的。一个二进制数,只要把它从低位到高位每 4 位组成一组,直接用十六进制数来表示就可以了。

例 1.5

```
0011  0101  1011  1111
 3     5     B     F
```

亦即 0011010110111111B=35BFH

反之,把十六进制数中的每一位用 4 位二进制数表示,就形成相应的二进制数了。

例 1.6

```
 A     1     9     C
1010  0001  1001  1100
```

亦即 A19CH=1010000110011100B

1.1.3.3 十六进制数和十进制数之间的转换

各位十六进制数与其对应权值的乘积之和即为与此十六进制数相对应的十进制数。

例 1.7 N=BF3CH

$=11\times16^3+15\times16^2+3\times16^1+12\times16^0$

$=11\times4096+15\times256+3\times16+12\times1$

$=48956D$

十进制数转换为十六进制数也可使用降幂法和除法。

· 降幂法

首先写出要转换的十进制数,其次写出小于该数的十六进制权值,然后找出该数中包含多少个最接近它的权值的倍数,这一倍数即对应位的值,用原数减去此倍数与相应位权值的乘积得到一个差值,再用此差值去找低一位的权值的倍数,如此反复直到差值为 0 为止。

例 1.8

N=48956D　　小于 N 的十六进制权值为

	4096	256	16	1
对应的十六进制数	B	F	3	C

计算过程如下：

$$48956 - 11 \times 4096 = 3900$$
$$3900 - 15 \times 256 = 60$$
$$60 - 3 \times 16 = 12$$
$$12 - 12 \times 1 = 0$$

所以 N＝48956D＝(11)(15)(3)(12)

＝BF3CH

· 除法

把要转换的十进制数的整数部分不断除以 16，并记下余数，直到商为 0 为止。

例 1.9

N＝48956D

48956/16 = 3059	(a_0 = 12)
3059/16 = 191	(a_1 = 3)
191/16 = 11	(a_2 = 15)
11/16 = 0	(a_3 = 11)

所以 N＝48956D＝BF3CH。

对于要转换的十进制数的小数部分，则应不断地乘以 16，并记下其整数部分，直到结果的小数部分为零为止。由于其方法与二、十进制数的转换的方法是相同的，这里不再举例说明。显然，为把一个十进制数转换为二进制数，可以先把该数转换为十六进制数，然后再转换为二进制数，这样可以减少计算次数，反之，要把一个二进制数转换为十进制数，也可采用同样的办法。

1.2 二进制数和十六进制数运算

1.2.1 二进制数的运算

加法规则：

0＋0＝0

0＋1＝1

1＋0＝1

1＋1＝0（进位 1）

乘法规则：

0×0＝0

0×1＝0

1×0＝0

1×1＝1

1.2.2 十六进制数的运算

十六进制数的运算可以采用先把该十六进制数转换为十进制数，经过计算后再把结果转换为十六进制数的方法，但这样做比较繁琐。其实，只要按照逢十六进一的规则，直接用十六进制数来计算也是很方便的。

十六进制加法：

当两个一位数之和 S 小于 16 时，与十进制数同样处理，如两个一位数之和 S≥16 时，则应

该用S－16及进位1来取代S。

例 1.10

$$\begin{array}{r} 05C3H \\ +\ 3D25H \\ \hline 42E8H \end{array}$$

十六进制数的减法也与十进制数类似，够减时可直接相减，不够减时服从向高位借1为16的规则。

例 1.11

$$\begin{array}{r} 3D25H \\ -\ 05C3H \\ \hline 3762H \end{array}$$

十六进制数的乘法可以用十进制数的乘法规则来计算，但结果必须用十六进制数来表示。

例 1.12

$$\begin{array}{r} 05C3H \\ \times\ 00ABH \\ \hline 3F61 \\ +\ 399E \\ \hline 3D941H \end{array}$$

十六进制数的除法可以根据其乘法和减法规则处理，需要时读者可自行处理，这里不再赘述。

1.3 计算机中数和字符的表示

1.3.1 数的补码表示

计算机中的数是用二进制来表示的，数的符号也是用二进制表示的，把一个数连同其符号在内在机器中的表示加以数值化，这样的数称为机器数。一般用最高有效位来表示数的符号，正数用0表示，负数用1表示。机器数可以用不同的码制来表示，常用的有原码、补码和反码表示法。由于多数机器的整数采用补码表示法，IBM PC机也是这样，所以我们在这里只介绍补码表示法。

补码表示法中，正数采用符号-绝对值表示，即数的最高有效位为0表示符号为正，数的其余部分则表示数的绝对值。例如，假设机器字长为8位，则$[+1]_{补}=00000001$，$[+127]_{补}=01111111$，$[+0]_{补}=00000000$。

当用补码表示法来表示负数时则要麻烦一些，负数X用$2^n-|X|$来表示，其中n为机器的字长。当n=8时，$[-1]_{补}=2^8-1=11111111$，而$[-127]_{补}=2^8-127=10000001$，显然，最高有效位为1表示该数的符号为负。应该注意，$[-0]_{补}=2^8=00000000$，所以在补码表示法中0只有一种表示，即00000000。对于10000000这个数，在补码表示法中被定义为－128。这样，8位补码能表示数的范围为－128～＋127。

我们可以用一种比较简单的办法来写出一个负数的补码表示：先写出与该负数相对应的正数的补码表示(用符号-绝对值法)，然后将其按位求反(即0变为1，1变为0)，最后在末位(最低位)加1，就可以得到该负数的补码表示了。

例 1.13 机器字长为16位，写出N＝－117D的补码表示。

＋117D可表示为　0000　0000　0111　0101

按位求反后为　1111　1111　1000　1010

末位加 1 后为　　1111　1111　1000　1011

用十六进制数表示为　F　　F　　8　　B

即 $[-117]_{补}=FF8BH$

例 1.14　如机器字长为 8 位，则 −46D 的补码表示为：

+46 的补码表示	0010	1110
按位求反	1101	0001
末位加 1	1101	0010
用十六进制数表示	D	2

即 $[-46]_{补}=D2H$

至此，读者应该已经学会了一个数的补码表示法。在这里，我们顺便说明一下，用补码表示数时的符号扩展问题。所谓符号扩展是指一个数从位数较少扩展到位数较多（如从 8 位扩展到 16 位，或从 16 位扩展到 32 位）时应该注意的问题。对于用补码表示的数，正数的符号扩展应该在前面补 0，而负数的符号扩展则应该在前面补 1。例如，我们已经知道如机器字长为 8 位，则 $[+46]_{补}=00101110$，$[-46]_{补}=11010010$；如果要把它们从 8 位扩展到 16 位，则

$[+46]_{补}=0000000000101110=002EH$　$[-46]_{补}=1111111111010010=FFD2H$

下面，我们再来讨论一下 n 位补码表示数的范围问题。8 位二进制数可以表示 $2^8=256$ 个数，当它们是补码表示的带符号数时，它们的表数范围是 $-128\leqslant N\leqslant+127$。一般说来，n 位补码表示的数的表数范围是：

$$-2^{n-1}\leqslant N\leqslant 2^{n-1}-1$$

所以 $n=16$ 时的表数范围是：

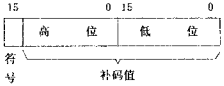
$$-32768\leqslant N\leqslant+32767$$

表 1.2　n 位二进制补码数的表数范围

十进制数	二进制数	十六进制数	十进制数	十六进制数
	n=8		n=16	
+127	01111111	7F	+32767	7FFF
+126	01111110	7E	+32766	7FFE
⋮	⋮	⋮	⋮	⋮
+2	00000010	02	+2	0002
+1	00000001	01	+1	0001
0	00000000	00	0	0000
−1	11111111	FF	−1	FFFF
−2	11111110	FE	−2	FFFE
⋮	⋮	⋮	⋮	⋮
−126	10000010	82	−32766	8002
−127	10000001	81	−32767	8001
−128	10000000	80	−32768	8000

表 1.2 表示当 n=8 和 16 时 n 位二进制补码数的表数范围。

在机器里，为了扩大表数范围，可以用二个机器字来表示一个机器数，这种数称为双字长数或双精度数，其格式为：



其中高位字的最高有效位为符号位，高位字的低 15 位和整个低位字的 16 位联合组成 31 位数来表示数值，因而低位字的最高有效位没有符号意义，只有数值意义。双字长数的表数范围可扩大到：

$$-2^{31} \leqslant N \leqslant 2^{31}-1$$

$2^{31} \approx 2.15\times10^{9}$，可见表数范围扩大了许多。上图中每个字上的 0,15 是位编号，每个机器字都从低位开始给每一位以编号，所以从右至左依次编号为 0,1,…,15。

1.3.2 补码的加法和减位

我们知道，对一个正数的补码表示按位求反后再在末位加 1，可以得到与此正数相应的负数的补码表示。我们把这种对一个二进制数按位求反后在末位加 1 的运算称为求补运算，可以证明补码表示的数具有以下特性：

$$[X]_{补} \xrightarrow{求补} [-X]_{补} \xrightarrow{求补} [X]_{补}$$

在这里，只用例子来说明。由例 1.13 可见：

$$[117]_{补} = 0075H$$

$$[-117]_{补} = FF8BH$$

现对 $[-117]_{补}$ 作求补运算：

$[-117]_{补}$ 为	1111	1111	1000	1011
按位求反后得	0000	0000	0111	0100
末位加 1 后得	0000	0000	0111	0101

此数正是 $[+117]_{补}=0075H$。

这一特性在补码的加、减法运算中很有用。

补码的加法规则是：

$$[X+Y]_{补} = [X]_{补} + [Y]_{补}$$

补码的减法规则是：

$$[X-Y]_{补} = [X]_{补} + [-Y]_{补}$$

其中的 $[-Y]_{补}$ 只要对 $[Y]_{补}$ 求补就可得到。对于这两个规则我们只用例子来说明。读者可以从下面的例子中认识到由于用补码表示数，使计算机中的加、减法运算十分简便，它不必判断数的正负，只要符号位参加运算能自动地得到正确的结果。假设机器字长为 8 位，用下列例子说明补码的加法运算。

例 1.15

十进制	二进制
25	00011001
+32	+00100000
57	00111001

例 1.16

$$\begin{array}{r} 32 \\ +(-25) \\ \hline 7 \end{array} \qquad \begin{array}{r} 00100000 \\ +11100111 \\ \hline \text{C}\ 00000111 \end{array}$$

例 1.17

$$\begin{array}{r} 25 \\ +(-32) \\ \hline -7 \end{array} \qquad \begin{array}{r} 00011001 \\ +11100000 \\ \hline 11111001 \end{array}$$

例 1.18

$$\begin{array}{r} -25 \\ +(-32) \\ \hline -57 \end{array} \qquad \begin{array}{r} 11100111 \\ +11100000 \\ \hline \text{C}\ 11000111 \end{array}$$

可以看出，上述 4 个例子的计算结果都是正确的，在例 1.16 和例 1.18 中，从最高有效位向高位的进位由于机器字长的限制而自动丢失，但这并不会影响运算结果的正确性。同时，机器为了某种需要(将在第三章中说明)将把这一进位值保留在标志寄存器的进位位 C 中。

下面，我们再用四个例子说明补码的减法运算，这里机器字长仍假定为 8 位。

例 1.19

十进制

$$\begin{array}{r} 25 \\ -32 \\ \hline -7 \end{array}$$

$$00011001 - 00100000$$

计算机中用对减数求补的方法把减法转化为加法

二进制

$$\begin{array}{r} 00011001 \\ +11100000 \\ \hline 11111001 \end{array}$$

例 1.20

$$\begin{array}{r} 32 \\ -(-25) \\ \hline 57 \end{array} \qquad \begin{array}{r} 00100000 \\ -(-11100111) \end{array} \qquad \begin{array}{r} 00100000 \\ +00011001 \\ \hline 00111001 \end{array}$$

例 1.21

$$\begin{array}{r} -25 \\ -(+32) \\ \hline -57 \end{array} \qquad \begin{array}{r} 11100111 \\ -00100000 \end{array} \qquad \begin{array}{r} 11100111 \\ +11100000 \\ \hline \text{C}\ 11000111 \end{array}$$

例 1.22

$$\begin{array}{r} -25 \\ -(-32) \\ \hline 7 \end{array} \qquad \begin{array}{r} 11100111 \\ -11100000 \end{array} \qquad \begin{array}{r} 11100111 \\ +00100000 \\ \hline \text{C}\ 00000111 \end{array}$$

可以看出，在机器里，补码减法是用对减数求补后把减法转换为加法进行的，它同样能自动地得到正确的结果。其中例 1.21 和例 1.22 中的由最高有效位向高位的进位同样自动丢失而不

会影响运算的结果。

1.3.3 无符号整数

在某些情况下，要处理的数全是正数，此时再保留符号位就没有意义了。我们可以把最高有效位也作为数值处理，这样的数称为无符号整数。16 位无符号数的表数范围是 0≤N≤65535，8 位无符号数的表数范围是 0≤N≤255。

在计算机中最常用的无符号整数是表示地址的数。此外，如双精度数的低位字也是无符号整数等。在某些情况下，带符号的数(在机器中用补码表示)与无符号数的处理是有差别的，读者在处理数时，应注意它们的区别。

1.3.4 字符表示法

计算机中处理的信息并不全是数，有时需要处理字符或字符串，例如从键盘输入的信息或打印输出的信息都是字符方式输入输出的，因此，计算机必须能表示字符。字符包括：

字母：A、B、…、Z，a、b、…、Z；

数字：0、1、…、9；

专用字符：+、-、*、/、$、SP(space 空格)…

非打印字符：BEL(Bell 响铃)、LF(Line Feed 换行)、CR(Carriage Return 回车)、……

这些字符在机器里必须用二进制数来表示。IBM PC 机采用目前最常用的美国信息交换标准代码 ASCII(American Standard Code for Information Interchange)来表示。这种代码用一个字节(8 位二进制码)来表示一个字符，其中低 7 位为字符的 ASCII 值，最高位一般用作校验位。表 1.3 列出了用十六进制数表示的部分常用字符的 ASCII 值。

表 1.3 常用字符的 7 位 ASCII 值(用十六进制数表示)

字符	ASCII	字符	ASCII	字符	ASCII	字符	ASCII
NUL	00	4	34	M	4D	↑	5E
BEL	07	5	35	N	4E	←	5F
LF	0A	6	36	O	4F	a	61
FF	0C	7	37	P	50	b	62
CR	0D	8	38	Q	51	c	63
SP	20	9	39	R	52	d	64
!	21	:	3A	S	53	e	65
"	22	;	3B	T	54	f	66
#	23	<	3C	U	55	g	67
$	24	=	3D	V	56	h	68
%	25	>	3E	W	57	i	69
&	26	?	3F	X	58	j	6A
'	27	@	40	Y	59	k	6B
(	28	A	41	Z	5A	l	6C
)	29	B	42	[	5B	m	6D
*	2A	C	43	\	5C	n	6E
+	2B	D	44	]	5D	o	6F
,	2C	E	45	^	5E	p	70
-	2D	F	46	←	5F	q	71
.	2E	G	47	`	60	r	72
/	2F	H	48	a	61	s	73
0	30	I	49	b	62	t	74
1	31	J	4A	c	63	u	75
2	32	K	4B	d	64	v	76
3	33	L	4C	e	65	w	77
						x	78
						y	79
						z	7A
						{	7B
						|	7C
						}	7D
						~	7E

1.4 几种基本的逻辑运算

1.4.1 "与"运算(AND)

A	B	A∧B
0	0	0
0	1	0
1	0	0
1	1	1

"与"运算又称逻辑乘,可用符号·或∧来表示。如有A、B两个逻辑变量(每个变量只能有0或1两种取值),可能有的取值情况只有4种,在各种取值的条件下得到的"与"运算结果如右表所示:即只有当A、B两个变量的取值均为1时,它们的"与"运算的结果才是1。

1.4.2 "或"运算(OR)

A	B	A∨B
0	0	0
0	1	1
1	0	1
1	1	1

"或"运算又称逻辑加,可用符号+或∨来表示。"或"运算规则可用右表说明:即A、B两个变量中只要有一个变量取值为1,则它们"或"运算的结果就是1。

1.4.3 "非"运算(NOT)

A	$\overline{A}$
0	1
1	0

如变量为A,则它的"非"运算的结果用$\overline{A}$来表示。"非"运算规则可用右表说明。

1.4.4 "异或"运算(XOR Exclusive—OR)

A	B	A∀B
0	0	0
0	1	1
1	0	1
1	1	0

"异或"运算可用符号∀来表示。其运算规则用右表说明:即当两个变量的取值相异时,它们的"异或"运算结果为1。

所有的逻辑运算都是按位操作的,下面举例说明:

例1.23 如两个变量的取值为X=00FFH和Y=5555H,求$Z_1=X\wedge Y$;$Z_2=X\vee Y$;$Z_3=\overline{X}$;$Z_4=X\veebar Y$的值。

X=0000 0000 1111 1111

Y=0101 0101 0101 0101

Z_1=0000 0000 0101 0101=0055H

Z_2=0101 0101 1111 1111=55FFH

Z_3=1111 1111 0000 0000=FF00H

Z_4=0101 0101 1010 1010=55AAH

习 题

1.1 用降幂法和除法将下列十进制数转换为二进制数和十六进制数:

(1) 369 (2) 10000 (3) 4095 (4) 32767

1.2 将下列二进制数转换为十六进制数和十进制数:

(1) 101101 (2) 10000000 (3) 1111111111111111 (4) 11111111

1.3 将下列十六进制数转换为二进制数和十进制数:

(1) FA　(2) 5B　(3) FFFE　(4) 1234

1.4　完成下列十六进制数的运算，并转换为十进制数进行校核

(1) 3A+B7　(2) 1234+AF　(3) ABCD−FE　(4) 7AB×6F

1.5　下列各种均为十进制数，请用 8 位二进制补码计算下列各题，并用十六进制数表示其运算结果。

(1) (−85)+76　(2) 85+(−76)　(3) 85−76　(4) 85−(−76)　(5) (−85)−76　(6) −85−(−76)

1.6　下列各数为十六进制表示的 8 位二进制数，请说明当它们分别被看作是用补码表示的带符号数或无符号数时，它们所表示的十进制数是什么？

(1) D8　(2) FF

1.7　下列各数均为用十六进制表示的 8 位二进制数。请说明当它们分别被看作是用补码表示的数或字符的 ASCII 时，它们所表示的十进制数及字符是什么？

(1) 4F　(2) 2B　(3) 73　(4) 59

1.8　请写出下列字符串的 ASCII 码值。

For example,

This is a number 3692.

第二章　IBM PC 计算机组织

2.1　计算机系统概述

计算机系统包括硬件和软件两部分。硬件包括电路、插件板、机柜等。软件则是为了运行、管理和维护计算机而编制的各种程序的总和。

2.1.1　硬件

典型计算机的结构可用图 2.1 表示。其中包括中央处理机 CPU（Central Processing Unit）、存储器（Memory）和输入/输出（I/O）子系统三个主要组成部分，用系统总线把它们连接在一起。

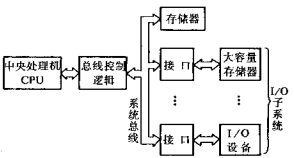


图 2.1　计算机结构

存储器是计算机的记忆部件。人们编写的程序（由指令序列组成）就存放在这里，它也可以存放程序中所用的数据、信息及中间结果。

中央处理机包括运算器和控制器两部分。运算器执行所有的算术和逻辑运算指令，控制器则负责全机的控制工作，它负责把指令逐条从存储器中取出，经译码分析后向全机发出取数、执行、存数等控制命令，以保证正确完成程序所要求的功能。

I/O 子系统一般包括 I/O 设备及大容量存储器两类外部设备。I/O 设备是指负责与计算机的外部世界通信用的输入、输出设备，如显示终端、键盘输入、打印输出等多种类型的外部设备。大容量存储器则是指可存储大量信息的外存储器如磁盘、磁带、光盘等，机器内部的存储器则称为内存储器，简称内存。由于内存的容量有限，所以计算机用外存储器作为内存的后援设备，它的容量可以比内存大很多，但存取信息的速度要比内存慢得多，所以除必要的系统程序外，一般程序（包括数据）是存放在外存中的。只有当运行时，才把它从外存传送到内存的某个区域，再由中央处理机控制执行。

系统总线把 CPU、存储器和 I/O 设备连接起来，用来传送各部分之间的信息。系统总线包括数据线，地址线和控制线三组。数据线传送信息，地址线指出信息的来源和目的地，控制线则规定总线的动作，如方向等。系统总线的工作由总线控制逻辑负责指挥。

IBM PC 机是美国 IBM 公司在 1981 年推出的个人计算机。机中的 CPU 由一个微处理机芯片 8088 组成，其主要性能是：系统时钟 4.77MHz，基本指令 99 条；机内字长 16 位，数据线则为 8 根，地址线有 20 根，提供了 1M 字节的寻址能力；平均运算速度为 0.65MPS（兆次/秒）。IBM PC 机有 1M 字节随机存取的存储器（RAM）及 40K 字节只读存储器（ROM）。ROM 里存放有 BASIC 解释程序及基本输入/输出系统软件 BIOS，它们是只允许读出而不允许用户写入或修改的。IBM PC 机提供了三种基本外设接口，它们是键盘、扬声器和盒式磁带，此外允许用户配

置各种外部设备，如显示器、磁盘、软盘、打印机等。

2.1.2 软件

计算机软件是计算机系统的重要组成部分，它可以分成系统软件和用户软件两大类。系统软件是由计算机的生产厂家提供给用户的一组程序，这些程序是用户使用机器时为产生、准备和执行用户程序所必需的。用户软件则是用户自行编制的各种程序。图 2.2 表示了计算机软件的层次。在这里我们将简要介绍系统软件的组成。

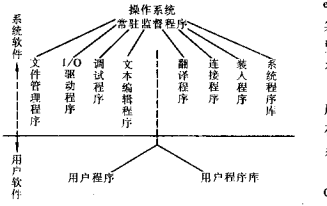


图 2.2 计算机软件层次图

系统软件的核心称为操作系统(Operating System)。操作系统是系统程序的集合，它的主要作用是对系统的硬、软件资源进行合理的管理，为用户创造方便、有效和可靠的计算机工作环境。

操作系统的主要部分是常驻监督程序(monitor)，只要一开机它就存在于内存中，它可以从用户接收命令，并使操作系统执行相应的动作。

I/O 驱动程序(I/O driver)用来对 I/O 设备进行控制或管理。当系统程序或用户程序需要使用 I/O 设备时，就调用 I/O 驱动程序来对设备发出命令，完成 CPU 和 I/O 设备之间的信息传送。

文件管理系统(File management)用来处理存储在外存储器中的大量信息，它可以和外存储器的设备驱动程序相连接，对存储在其中的信息以文件(File)形式进行存取、复制及其他管理操作。

文本编辑程序(Text editor)：文本是指由字母、数字、符号等组成的信息，它可以是一个用汇编语言或高级语言编写的程序，也可以是一组数据或一份报告。文本编辑程序用来建立、输入或修改文本，并使它存入存储器或大容量存储器中。例如 IBM PC 机提供的行编辑程序 EDLIN，可用来建立源文件，修改文本，有删除、插入、编辑和显示行等功能。字处理程序 WORDSTAR 可提供屏幕编辑功能，并能提供各种功能及命令的菜单，使文本的建立和修改更加方便。

翻译程序(Translator)：我们已经知道计算机是通过逐条地执行组成程序的指令来完成人们所给予的任务的，所以指令就是计算机能识别并能直接加以执行的语句，它当然是由二进制代码组成的，这种语言称为机器语言，显然它对于人们是很不方便的。既然计算机能识别的唯一语言是机器语言，而这种语言使编写程序很不方便，所以在计算机语言的发展过程中就出现了汇编语言和高级语言。汇编语言是一种符号语言，它几乎和机器语言一一对应，但在书写时却使用由字符串组成的助记符。例如：加法在汇编语言中是用助记符 ADD 表示的，而机器语言则用 6 位二进制代码来表示。显然，相对于机器语言来说，汇编语言是易于为人们所理解的，但计算机却不能直接识别汇编语言。汇编程序就是用来把由用户编制的汇编语言程序翻译成机器语言程序的一种系统程序。IBM PC 机中的汇编程序有 ASM 和 MASM 两种。ASM 称为小汇编程序，它占有较小的存储区，但功能较弱。MASM 称为宏汇编程序，它需要的存储区较大，但

功能较强，具有宏汇编能力，而 ASM 却不具备这种能力。

高级语言脱离开机器指令用人们更加易于理解的方式来编写程序，当然它们也要翻译成机器语言才能在机器上执行。高级语言的翻译程序有两种方式：一种是先把高级语言程序翻译成机器语言(或先翻译成汇编语言，然后再由汇编程序再次翻译成机器语言)程序，然后再在机器上执行，这种翻译程序称为编译程序(Compiler)，多数高级语言如 PASCAL，FORTRAN 等都采用这种方式。另一种是直接把高级语言程序在机器上运行，一边解释一边执行，这种翻译程序称为解释程序(Interpreter)，如 BASIC 就经常采用这种方式。

系统程序中的翻译程序包括汇编程序，BASIC 解释程序及各种高级语言的编译程序。

连接程序(Linker)用来把要执行的程序与库文件或其他已经翻译的子程序(能完成一种独立功能的程序模块)连接在一起，形成机器能执行的程序。

装入程序(Loader)用来把程序从外存储器传送到内存储器，以便机器执行。例如，计算机开机后就需要立即启动装入程序把常驻监督程序装入存储器，使机器运转起来。又如，用户程序经翻译和连接后，由连接程序直接调用装入程序，把可执行的用户程序装入内存以便执行。

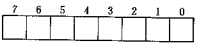
调试程序(Debug)是系统提供给用户的能监督和控制用户程序的一种工具。它可以装入、修改、显示或逐条执行一个程序。在 IBM PC 机上，简单的汇编语言程序可以通过 DEBUG 来建立、修改和执行。

系统程序库(System Library)和用户程序库(User Library)：各种标准程序、子程序及一些文件的集合称为程序库，它可以被系统程序或用户程序调用。操作系统还允许用户建立程序库，以提高不同类型用户的工作效率。

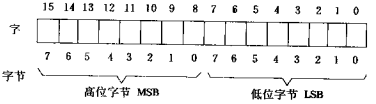
2.2 存 储 器

2.2.1 存储单元的地址和内容

计算机存储信息的基本单位是一个二进制位，一位可存储一个二进制数：0 或 1。每 8 位组成一个字节，位编号如下所示：



IBM PC 机的字长为 16 位，由 2 个字节组成，位编号如下所示：



在存储器里以字节为单位存储信息，为了正确地存放或取得信息，每一个字节单元给以一个存储器地址。地址从 0 开始编号，顺序地每次加 1。在机器里，地址也是用二进制数来表示的。当然它是无符号整数，书写格式为十六进制数。

IBM PC 机的字长为 16 位。既然每个字节单元有一个二进制数表示地址，那么 16 位二进

制数可以表示多少个字节单元的地址呢？显然答案应该是有 2^{16} 个，所以它可以表示的地址范围应该是 0～65535。在计算机里，为方便起见，在讨论存储器的容量时以 $2^{10}=1024$ 为基本单位，称其为 1K。这样，65536 个字节单元的存储容量就是 64K，其地址编号的范围用十六进制数表示为 0～FFFFH。如下所示：

0000,0001,0002,0003,…0009,000A,000B,000C,000D,000E,000F,
0010,0011,… … ,　　0019,001A,001B,001C,001D,001E,001F,
0020,0021,… … ,　　,002F,
⋮
FFE0,FFE1,… … ,　　,FFEE, FFEF,
FFF0,FFF1,… … ,　　,FFFE, FFFF,

一个存储单元中存放的信息称为该存储单元的内容，图 2.3 表示了存储器里存放信息的情况。可以看出，4 号字节单元中存放的信息为 34H，也就是说 4 单元中的内容为 34H，表示为：

(0004) = 34H

但是机器字长是 16 位，大部分数据都是以字为单位表示的。那么一个字怎样存入存储器呢？一个字存入存储器要占有相继的二个字节，存放时，低位字节存入低地址，高位字节存入高地址，也就是说是以相反的次序存入的。这样二个字节单元就构成了一个字单元，字单元的地址采用它的低地址来表示。图 2.3 中 4 号字单元的内容为 1234H，表示为：

(0004) = 1234H

所以同一个地址既可看作字节单元的地址，又可看作字单元的地址，这要根据使用情况确定。可以看出：字单元的地址可以是偶数，也可以是奇数。但是，在机器里，访问存储器(要求取数或存数)都是以字为单位进行的，也就是说，机器是以偶地址访问存储器的。这样，对于奇地址的字单元，要取一个字需要访问二次存储器，当然这样做要花费较多的时间。

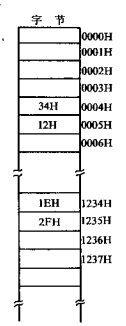


图 2.3 存储单元的地址和内容

如上所述，如果用 X 表示某存储单元的地址，则 X 单元的内容可以表示为(X)；假如 X 单元中存放着 Y，而 Y 又是一个地址，则可用(Y)=((X))来表示 Y 单元的内容。如图 2.3 中

(0004H) = 1234H

而　　(1234H) = 2F1EH

则也可记作　　((0004H)) = 2F1EH。

存储器有这样的特性：它的内容是取之不尽的。也就是说，从某个单元取出其内容后，该单元仍然保存着原来的内容不变，可以重复取出，只有存入新的信息后，原来保存的内容就自动丢失了。

2.2.2 存储器地址的分段

前面已经提到 16 位字长的机器可以访问的最大存储空间为 64K 字节，而 IBM PC 机的最大存储容量为 1M 字节。由于

$$2^{20} = 1048576 = 1024K = 1M$$

所以，要访问 1M 字节空间的存储器必须有 20 位地址，用 16 进制数表示 1M 字节的地址范围

应为 00000～FFFFF。那么，在 16 位字长的机器里，用什么办法来提供 20 位地址呢？在 IBM PC 机里采用了存储器地址分段的办法。

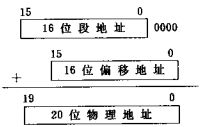
程序员在编制程序时要把存储器划分成段，每个段的大小可达 64K，这样段内地址可以用 16 位表示。实际上，可以根据需要来确定段的大小，它可以是 1 个、100 个、1000 个或在 64K 范围内的任意个字节。IBM PC 机对段的起始地址有所限制，段不能起始于任意地址，而必须从任一小段(paragraph)的首地址开始。机器规定：从 0 地址开始，每 16 个字节为一小段，下面列出了存储器最低地址区的三个小段的地址区间，每行为一小段。

00000,00001,00002,…,0000E,0000F;
00010,00011,00012,…,0001E,0001F;
00020,00021,00022,…,0002E,0002F;
⋮

其中，第一列就是每个小段的首地址。其特征是：在 16 进制表示的地址中，最低为 0(即 20 位地址的低 4 位为 0)。在 1M 字节的地址空间里，共有 64K 个小段首地址，可表示如下：

00000H
00010H
⋮
41230H
41240H
⋮
FFFE0H
FFFF0H

在 1M 字节的存储器里，每一个存储单元都有一个唯一的 20 位地址，称为该存储单元的物理地址。CPU 访问存储器时，必须先确定所要访问的存储单元的物理地址才能取得(或存入)该单元中的内容。20 位物理地址由 16 位段地址和 16 位偏移地址组成，段地址是指每一段的起始地址，由于它必须是小段的首地址，所以其低 4 位一定是 0，这样就可以规定段地址只取段起始地址的高 16 位值。偏移地址则是指在段内相对于段起始地址的偏移值。这样，物理地址的计算方法可以表示如下：



也就是说：把段地址左移 4 位再加上偏移地址值就形成物理地址。或写成：

16d×段地址＋偏移地址＝物理地址

显然，每个存储单元只有唯一的物理地址，但它却可由不同的段地址和不同的偏移地址组成。

在 IBM PC 机中，有四个专门存放段地址的寄存器，称为段寄存器。它们是代码段 CS(Code Segment)、数据段 DS(Data Segment)、堆栈段 SS(Stack Segment)和附加段 ES(Extra Segment)寄存器。每个段寄存器可以确定一个段的起始地址，而这些段则各有各的用途。代码段存放当前

正在运行的程序。数据段存放当前运行程序所用的数据，如果程序中使用了串处理指令，则其源操作数也存放在数据段中。堆栈段定义了堆栈的所在区域，堆栈是一种数据结构，它开辟了一个比较特殊的存储区，并以后进先出的方式来访问这一区域，在下一章里我们还会专门加以说明。附加段是附加的数据段，它是一个辅助的数据区，也是串处理指令的目的操作数存放区。程序员在编制程序时，应该按照上述规定把程序的各部分放在规定的区段之内。

除非专门指定，一般情况下，各段在存储器中的分配是由操作系统负责的。每个段可以独立地占用 64K 存储区，如图 2.4 所示。各段也可以允许重叠，下面的例子就可以说明这种情况。

例 2.1 如果代码段中的程序占有 8K(2000H)存储区，数据段占有 2K(800H)存储区，堆栈段只占有 256 个字节的存储区。此时段区的分配如图 2.5 所示。从图中可以看出：代码段的区域可以是 02000～11FFFH，但由于程序区只需要 8K，所以程序区结束后的第一个小段的首地址就作为数据段的起始地址。也就是说，在这里，代码段和数据段可以重叠在一起。当然每个存储单元的内容是不允许发生冲突的。这里所谓的重叠只是指每个段区的大小允许根据实际需要来分配，而不一定要占有 64K 的最大段空间。实际上，段区的分配工作是由操作系统完成的。但是，系统允许程序员在必要时可指定所需占用的内存区，指定方法将在第四章中说明。

如果程序中的四个段都在 64K 的范围之内，而且程序运行时所需要的信息都在本程序所定义的段区之内，那么，程序员只要在程序的首部设定各段寄存器的值就可以了。如果程序的某一段(例如数据段)在程序运行过程中会超过 64K 空间，或者程序中可能访问除本身四个段以外的其他段区的信息，那么在程序中必须动态地修改段寄存器的内容，以保证所获得的信息的正确性。可见并不会因段区的划分而限制了程序空间。

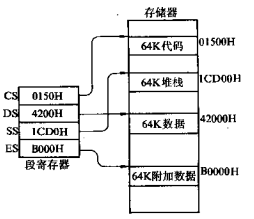


图 2.4 段分配方式之一

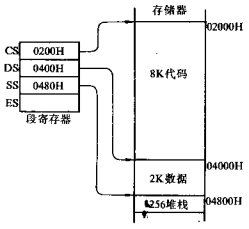


图 2.5 段分配方式之二

这种存储器分段的方法虽然给程序设计带来一定的麻烦，但它可以扩大存储空间，而且对于程序的再定位也是很方便的。这个问题在第十三章中还会专门说明。

2.3 中央处理机

2.3.1 中央处理机 CPU 的组成

CPU 的任务是执行存放在存储器里的指令序列。它由运算器和控制器两部分组成。在 IBM PC 机中它就是一个微处理机芯片 8088。它的组成可简单地用图 2.6 来表示。

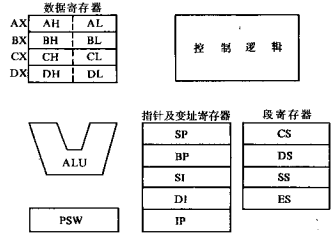


图 2.6 IBM PC 机的 CPU 组成

从图中可以看出，CPU 由三部分组成：

1. 算术逻辑部件 ALU(Arithmetic Logic Unit)用来进行算术和逻辑运算。

2. 控制逻辑负责对全机的控制工作，包括从存储器取出指令，对指令进行译码分析，从存储器取得操作数，发出执行指令的所有命令，把结果存入存储器，以及对总线及 I/O 传送的控制等。

3. 工作寄存器在计算机中起着重要的作用，每一个寄存器相当于运算器中的一个存储单元，但它的存取速度比存储器要快得多。它用来存放计算过程中所需要的或所得到的各种信息，包括操作数地址、操作数及运算的中间结果等。下面我们将专门介绍这些寄存器。

2.3.2 8086/8088 的寄存器组

2.3.2.1 数据寄存器

包括 AX、BX、CX、DX 四个通用寄存器，用来暂时存放计算过程中所用到的操作数、结果或其他信息。它们都可以以字(16 位)的形式访问，或者也可以以字节(8 位)的形式访问。例如对 AX 可以分别访问高位字节 AH 或低位字节 AL。这四个寄存器都是通用寄存器，但它们又可以用于各自的专用目的。

AX(Accumulator)作为累加器用，所以它是算术运算的主要寄存器。另外，所有的 I/O 指令都使用这一寄存器与外部设备传送信息。

BX(Base)可以作为通用寄存器使用，此外在计算存储器地址时，它经常用作基址寄存器。

CX(Count)可以作为通用寄存器使用，此外在循环(LOOP)和串处理指令中用作隐含的计数器。

DX(Data)可以作为通用寄存器使用。一般在作双字长运算时把 DX 和 AX 组合在一起存放一个双字长数，DX 用来存放高位字。此外，对某些 I/O 操作，DX 可用来存放 I/O 的端口地址。

2.3.2.2 指针及变址寄存器

包括 SP、BP、SI、DI 四个 16 位寄存器。它们可以象数据寄存器一样在运算过程中存放操作

数，但它们只能以字(16 位)为单位使用。此外，它们更经常的用途是在段内寻址时提供偏移地址。其中 SP(Stack Pointer)称为堆栈指针寄存器，BP(Base Pointer)称为基址指针寄存器，它们都可以与 SS 寄存器联用来确定堆栈段中的某一存储单元的地址。SP 用来指示栈顶的偏移地址，BP 可作为堆栈区中的一个基地址以便访问堆栈中的其他信息。SI(Source Index)源变址寄存器和 DI(Destination Index)目的变址寄存器一般与 DS 联用，用来确定数据段中某一存储单元的地址。这两个变址寄存器有自动增量和自动减量的功能，所以用于变址是很方便的。在串处理指令中，SI 和 DI 作为隐含的源变址和目的变址寄存器，此时 SI 和 DS 联用，DI 和 ES 联用，分别达到在数据段和附加段中寻址的目的。

2.3.2.3 *段寄存器*

包括 CS、DS、SS 和 ES 四个段寄存器。有关内容上节已作了介绍，这里不再赘述。

2.3.2.4 *控制寄存器*

包括 IP 和 PSW 两个 16 位寄存器。IP(Instruction Pointer)为指令指针寄存器，它用来存放代码段中的偏移地址。在程序运行的过程中，它始终指向下一条指令的首地址，它与 CS 寄存器联用确定下一条指令的物理地址。当这一地址送到存储器后，控制器可以取得下一条要执行的指令，而控制器一旦取得这条指令就马上修改 IP 的内容，使它指向下一条指令的首地址。可见，计算机就是用 IP 寄存器来控制指令序列的执行流程的，因此 IP 寄存器是计算机中很重要的一个控制寄存器。

PSW(Program Status Word)为程序状态字寄存器，这是一个 16 位寄存器，由条件码标志(flag)和控制标志构成，如下所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

其中，条件码标志用来记录程序中运行结果的状态信息。由于这些状态信息往往作为后续条件转移指令的转移控制条件，所以称为条件码。它包括以下 6 位：

OF(Overflow Flag)溢出标志，在运算过程中，如操作数超出了机器能表示的范围则称为溢出。此时 OF 位置 1，否则置 0。

SF(Sign Flag)符号标志，记录运算结果的符号，结果为负时置 1，否则置 0。

ZF(Zero Flag)零标志，运算结果为 0 时 ZF 位置 1，否则置 0。

CF(Carry Flag)进位标志，记录运算时从最高有效位产生的进位值。例如，执行加法指令时，最高有效位有进位时置 1，否则置 0。

AF(Auxiliary carry Flag)辅助进位标志，记录运算时第 3 位(半个字节)产生的进位值。例如，执行加法指令时第 3 位有进位时置 1，否则置 0。

PF(Parity Flag)奇偶标志，用来为机器中传送信息时可能产生的代码出错情况提供检验条件。当结果操作数中 1 的个数为偶数时置 1，否则置 0。

控制标志位有三个：

DF(Direction Flag)方向标志，在串处理指令中控制处理信息的方向用。当 DF 位为 1 时，每次操作后使变址寄存器 SI 和 DI 减量，这样就使串处理从高地址向低地址方向处理。当 DF 为 0 时，则使 SI 和 DI 增量，使串处理从低地址向高地址方向处理。

IF(Interrupt Flag)中断标志，当 IF 为 1 时，允许中断，否则关闭中断。有关中断原理将于第八章说明。

TF(Trap Flag)陷井标志,用于单步方式操作。当 TF 位为 1 时,每条指令执行完后产生陷井,由系统控制计算机;当 TF 位为 0 时,CPU 正常工作不产生陷井。

以上就是 PSW 中各种状态和控制信息的含义。其中控制信息是由系统程序或用户程序根据需要用指令来设置的,而状态信息是由中央处理机根据计算的结果自动设置的。为方便起见,机器提供了设置状态信息的指令。必要时,程序员可使用这些指令来建立状态信息。

在调试程序 DEBUG 中提供了测试标志位的手段,它用符号表示标志位的值。表 2.1 说明每种标志位(因 DEBUG 不提供 TF 的符号,所以表格中未包括 TF 位在内)的符号表示。

表 2.1 PSW 中标志位的符号表示

标 志 名	标志为 1	标志为 0
OF 溢出(是/否)	OV	NV
DF 方向(减量/增量)	DN	UP
IF 中断(允许/关闭)	EI	DI
SF 符号(负/正)	NG	PL
ZF 零(是/否)	ZR	NZ
AF 辅助进位(是/否)	AC	NA
PF 奇偶(偶/奇)	PE	PO
CF 进位(是/否)	CY	NC

2.4 外部设备

计算机运行时的程序和数据都要通过输入设备送入机器,程序运行的结果要通过输出设备送给用户,所以输入、输出设备是计算机必不可少的组成部分。大容量的外存储器(如磁盘)能存储大量信息,也是现代计算机不可缺少的一部分。对于外部设备的管理是汇编语言的重要使用场合之一。

从图 2.1 可见,外部设备与主机(CPU 和存储器)的通信是通过外设接口进行的。每个接口包括一组寄存器。一般说来,这些寄存器有三种不同的用途:

1. 数据寄存器:用来存放要在外设和主机间传送的数据,这种寄存器实际上起缓冲器的作用。

2. 状态寄存器:用来保存外部设备或接口的状态信息,以便 CPU 在必要时测试外设状态,了解外设的工作情况。例如,每个设备都有忙闲位用来标志设备当前是否正在工作,是否有空接受 CPU 给予的新任务等。

3. 命令寄存器:CPU 给外设或接口的控制命令通过此寄存器送给外部设备。例如,CPU 要启动磁盘工作,必须发出启动命令等。

各种外部设备都有以上三种类型的寄存器,只是每个接口所配备的寄存器数量是根据设备的需要确定的。例如,工作方式较简单、速度又慢的键盘只有一个 8 位的数据寄存器,并把状态和命令寄存器合二为一个控制寄存器。又如工作速度快、工作方式又比较复杂的磁盘则需要多个数据、状态和命令寄存器。

为使主机访问外设方便起见,外设中的每个寄存器给予一个端口(Port)地址(又称端口

号)，这样就组成了一个独立于内存储器的 I/O 地址空间。IBM PC 机的 I/O 地址空间可达 64K，所以端口地址的范围是 0000～FFFFH，用 16 位二进制代码来表示。

主机与外设交换信息是通过输入、输出指令来完成的，在第三章我们还要专门说明。主机与外设的信息传送方式可有直接、查询、中断、成组传送等，将在第八章说明。实际上，对外设的管理及信息传送是汇编语言最经常使用的场合，也是最复杂的一部分程序。

为了便于用户使用外设，IBM PC 机提供了两种类型的例行程序供用户调用。一种是 BIOS (Basic Input/Output System)，另一种是 DOS(Disk Operating System)功能调用。它们都是系统编制的子程序，通过中断方式转入所需要的子程序去执行，执行完后返回原来的程序继续执行。这些例行程序有的完成一次简单的外设信息传送，如从键盘输入一个字符，或送一个字符至显示器等；也有的完成相当复杂的一次外设操作，如从磁盘读写一个文件等。总之，操作系统把一些复杂的外设操作编成例行程序，使用户用简单的中断指令(INT)就可以进入这些例行程序，完成所需要的外设操作，所以用户应尽量利用这些系统所提供的工具来编写自己的程序。我们将在第九章中讨论它们的使用方法，第十章至第十二章则将分别说明几种主要外设的典型应用程序。

BIOS 和 DOS 功能调用虽然都是系统提供的例行程序，但是它们之间又有差别。BIOS 存放在机器的只读存储器 ROM 中，所以可以把它看成是机器硬件的一个组成部分，它的层次比 DOS 更低，更接近硬件，因此它的语句要完成每一个对设备的直接命令，或信息传送。DOS 功能调用是操作系统 DOS 的一个组成部分，它在开机时由磁盘装入存储器，在它的例行程序中可以一次或多次调用 BIOS 以完成比 BIOS 更高级的功能。用户需要使用外设时，应尽可能使用层次较高的 DOS 功能调用，但有时它不能满足你的要求，此时，就需要直接调用 BIOS，如果 BIOS 还不能解决你的问题，那么就只好自己编制中断处理程序了。

习　题

2.1　在 IBM PC 机的输入/输出指令中，I/O 端口号通常是由 DX 寄存器提供的，但有时也可以在指令中直接指定 00～FFH 的端口号。试问可直接由指令指定的 I/O 端口数。

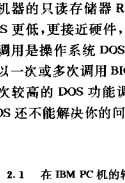


图 2.7　2.3 题的信息存放情况

2.2　有两个 16 位字 1E5H 和 2A3CH 分别存放在 IBM PC 机的存储器的 000B0H 和 000B3H 单元中，请用图表示出它们在存储器里的存放情况。

2.3　在 IBM PC 机的存储器中存放信息如图 2.7 所示。试读出 30022H 和 30024H 字节单元的内容，以及 30021H 和 30022H 字单元的内容。

2.4　段地址和偏移地址为 3017:000A 的存储单元的物理地址是什么？如果段地址和偏移地址是 3015:002A 和 3010:007A 呢？

2.5　如果在一个程序开始执行以前(CS)=0A7F0H，(如 16 进制数的最高位为字母，则应在其前加一个 0)(IP)=2B40H，试问该程序的第一个字的物理地址是多少？

2.6　存储器中每一段最多可有 10000H 个字节。如果用调试程序 DEBUG 的 r 命令在终端上显示出当前各寄存器的内容如下，请画出此时存储器分段的示意图，以及条件标志 OF、SF、ZF、CF 的值。

```
C>debug
-r
Ax=0000  BX=0000  CX=0079  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
```

DS=10E4 ES=10F4 SS=21F0 CS=31FF IP=0100 NV UP DI PL NZ NA PO NC

2.7 下列操作可使用哪些寄存器?

(1) 加法和减法

(2) 循环计数

(3) 乘法和除法

(4) 保存段地址

(5) 表示运算结果为 0

(6) 将要执行的指令地址

(7) 将要从堆栈取出数据的地址

2.8 哪些寄存器可以用来指示存储器地址?

2.9 请将下列左边的项和右边的解释联系起来(把所选字母放括号中):

(1) CPU ( ) A. 保存当前栈顶地址的寄存器。

(2) 存储器 ( ) B. 指示下一条要执行的指令的地址。

(3) 堆栈 ( ) C. 存储程序、数据等信息的记忆装置,PC 机有 RAM 和 ROM 两种。

(4) IP ( ) D. 以后进先出方式工作的存储空间。

(5) SP ( ) E. 把汇编语言程序翻译成机器语言程序的系统程序。

(6) 状态标志 ( ) F. 唯一代表存储空间中每个字节单元的地址。

(7) 控制标志 ( ) G. 能被计算机直接识别的语言。

(8) 段寄存器 ( ) H. 用指令的助记符、符号地址、标号等符号书写程序的语言。

(9) 物理地址 ( ) I. 把若干个模块连接起来成为可执行文件的系统程序。

(10) 汇编语言 ( ) J. 保存各逻辑段的起始地址的寄存器,PC 机有四个:CS、DS、SS、ES。

(11) 机器语言 ( ) K. 控制操作的标志,PC 机有三位:DF、ZF、TF。

(12) 汇编程序 ( ) L. 记录指令操作结果的标志,共六位:OF、SF、ZF、AF、PF、CF。

(13) 连接程序 ( ) M. 分析、控制并执行指令的部件,由算术逻辑部件 ALU 和寄存器组等组成。

(14) 指令 ( ) N. 由汇编程序在汇编过程中执行的指令。

(15) 伪指令 ( ) O. 告诉 CPU 要执行的操作(一般还要指出操作数地址),在程序运行时执行。

第三章　IBM PC 机的指令系统和寻址方式

我们已经知道计算机是通过执行指令序列来解决问题的，因而每种计算机都有一组指令集提供给用户使用，这组指令集就称为计算机的指令系统。目前，一般小型或微型计算机的指令系统可以包括几十种或百余种指令。本章说明 IBM PC 机指令系统以及在指令中为取得操作数地址所使用的寻址方式(Addressing mode)。

计算机中的指令由操作码字段和操作数字段两部分组成。操作码字段指示计算机所要执行的操作，而操作数字段则指出在指令执行操作的过程中所需要的操作数。例如，加法指令除需要指定做加法操作外，还需提供加数和被加数。操作数字段可以是操作数本身，也可以是操作数地址或是地址的一部分，还可以是指向操作数地址的指针或其他有关操作数的信息。

指令的格式一般是：

操作码	操作数	……	操作数

操作数字段可以有一个、二个或三个，通常称为一地址、二地址或三地址指令。例如，单操作数指令就是一地址指令，它只需要指定一个操作数，如加 1 指令只需要指出需要加 1 的操作数。大多数运算型指令都是双操作数指令。对于这种指令，有的机器使用三地址指令，除给出参加运算的两个操作数外，还指出运算结果的存放地址。近代多数机器则使用二地址指令，此时分别称两个操作数为源操作数和目的操作数。尽管在指令执行前这两个操作数都是输入操作数，但指令执行后将把运算结果存放到目的操作数的地址之中，也就是说，经过运算后，参加运算的一个操作数将会丢失，但一般不关心这个问题。如果在以后的运算中还会用到这个操作数的话，则应在运算之前先为它准备一个副本(即提前存储起来)。IBM PC 机的运算型指令就采用这种二地址指令。

指令的操作码字段在机器里的表示比较简单，只需对每一种操作指定确定的二进制代码就可以了。指令的操作数字段的情况就比较复杂了，如果操作数存放在寄存器中，则由于寄存器的数量较少，因而需要指定的操作数地址的位数就较少；但如果操作数存放在存储器里，那么一个存储单元的地址就需要 20 位，怎样设法使它在指令的操作数字段的表示中减少位数呢？另外，从程序运行时的数据结构来看，操作数常常不是单个的数，往往是成组的以表格或数组的形式存放在存储器的某一区中，在这种情况下，指令用什么方式来指定操作数地址更好呢？这在计算机的设计中是一个很重要的问题，它会影响机器运行的速度和效率。下一节将专门讨论这个问题。

我们已经知道，计算机只能识别二进制代码，所以机器指令是由二进制代码组成的。为便于人们使用而采用汇编语言来编写程序。汇编语言是一种符号语言，它用助记符来表示操作码，用符号或符号地址来表示操作数或操作数地址，它与机器指令是一一对应的。在本书中，除 3.2 节将说明机器语言概况外，其它部分均使用汇编语言格式来表示指令。

3.1 IBM PC 机的寻址方式

3.1.1 与数据有关的寻址方式

3.1.1.1 立即寻址方式(Immediate addressing)

操作数直接存放在指令中,紧跟在操作码之后,它作为指令的一部分存放在代码段里,这种操作数称为立即数。立即数可以是 8 位的或 16 位的。如果是 16 位数,则高位字节存放在高地址中,低位字节存放在低地址中。这种方式如图 3.1(1)所示。

立即寻址方式用来表示常数,它经常用于给寄存器赋初值,并且只能用于源操作数字段,不能用于目的操作数字段。

例 3.1 MOV AL,5

则指令执行后,(AL)=05H

例 3.2 MOV AX,3064H

则指令执行后,(AX)=3064H,可用图 3.2 表示。图中指令存放在代码段中,OP 表示该指令的操作码部分,3064 则为立即数,它是指令的一个组成部分。

3.1.1.2 寄存器寻址方式(Register addressing)

操作数在寄存器中,指令指定寄存器号。对于 16 位操作数,寄存器可以是 AX、BX、CX、DX、SI、DI、SP 和 BP 等;对于 8 位操作数,寄存器可以是 AL、AH、BL、BH、CL、CH、DL 和 DH。这种寻址方式由于操作数就在寄存器中,不需要访问存储器来取得操作数,因而可以取得较高的运算速度。这种方式如图 3.1(2)所示。

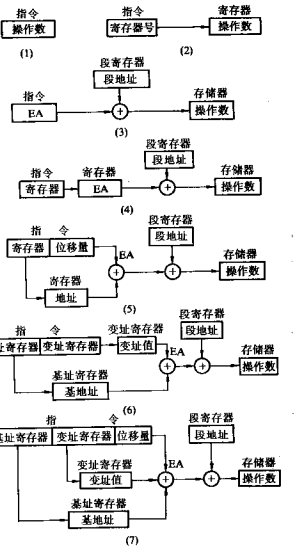


图 3.1 与数据有关的寻址方式

(1)立即 (2)寄存器 (3)直接 (4)寄存器间接

(5)寄存器相对 (6)基址变址 (7)相对基址变址

例 3.3 MOV AX,BX

如指令执行前(AX)=3064H,(BX)=1234H,则指令执行后,(AX)=1234H,(BX)保持不变。

除上述两种寻址方式外,以下各种寻址方式的操作数都在除代码段以外的存储区中,通过采用不同方式求得操作数地址,从而取得操作数。

3.1.1.3 直接寻址方式(Direct addressing)

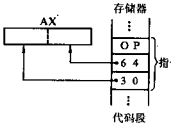


图 3.2 例 3.2 的执行情况

在 IBM PC 机中把操作数的偏移地址称为有效地址 EA (Effective Address)。在直接寻址方式中，有效地址 EA 就在指令中，它存放在代码段中指令操作码之后，但操作数一般存放在数据段中，所以必须先求出操作数的物理地址，然后再访问存储器才能取得操作数，如图 3.1(3)所示。如操作数在数据段中，则物理地址＝16d×(DS)＋EA。IBM PC 机中允许数据存放在数据段以外的其他段中，此时应在指令中指定段跨越前缀，在计算物理地址时应使用指定的段寄存器。

例 3.4 MOV AX,[2000H]

如(DS)＝3000H，则执行情况如图 3.3 所示。

执行结果为：(AX)＝3050H

在汇编语言指令中，可以用符号地址代替数值地址，如：

MOV AX,VALUE

此时 VAIUE 为存放操作数单元的符号地址。如写成：

MOV AX,[VALUE]

也是可以的，两者是等效的。如 VALUE 在附加段中，则应指定段跨越前缀如下：

MOV AX,ES：VALUE

或 MOV AX,ES：[VALUE]

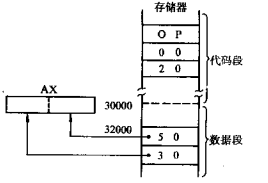


图 3.3 例 3.4 的执行情况

直接寻址方式适用于处理单个变量，例如要处理某个存放在存储器里的变量，可用直接寻址方式把该变量先取到一个寄存器中，然后再作进一步处理。

IBM PC 机中为了使指令字不要过长，规定双操作数指令除立即方式外必须有一个操作数使用寄存器方式，这就是一个变量常常先要送到寄存器去的原因。

3.1.1.4 寄存器间接寻址方式(Register indirect addressing)

操作数的有效地址在基址寄存器 BX、BP 或变址寄存器 SI、DI 中，而操作数则在存储器中，如图 3.1(4)所示。

如果指令中指定的寄存器是 BX、SI、DI，则操作数在数据段中，所以用 DS 寄存器的内容作为段地址，即操作数的物理地址为：

物理地址＝16d×(DS)＋(BX)

或 物理地址＝16d×(DS)＋(SI)

或 物理地址＝16d×(DS)＋(DI)

如指令中指定 BP 寄存器，则操作数在堆栈段中，段地址在 SS 中，所以操作数的物理地址为：

物理地址＝16d×(SS)＋(BP)

例 3.5 MOV AX,[BX]

如果 (DS)＝2000H,(BX)＝1000H,

则 物理地址＝20000＋1000＝21000H

执行情况如图 3.4 所示，执行结果为：

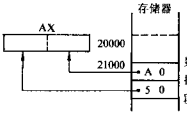


图 3.4 例 3.5 的执行情况

(AX)=50A0H

指令中也可指定段跨越前缀来取得其他段中的数据。如：

MOV AX,ES：[BX]

这种寻址方式可以用于表格处理，执行完一条指令后，只需修改寄存器内容就可取出表格中的下一项。

3.1.1.5 寄存器相对寻址方式(Register relative addressing)(或称直接变址寻址方式)

操作数的有效地址是一个基址或变址寄存器的内容和指令中指定的 8 位或 16 位位移量(displacement)之和。即

$$EA=\left\{\begin{matrix}(BX)\\(BP)\\(SI)\\(DI)\end{matrix}\right\}+\left\{\begin{matrix}8\text{位}\\16\text{位}\end{matrix}\right.\text{位移量}\}$$

同样，除有段跨越前缀者外，对于寄存器为 BX、SI、DI 的情况，段寄存器用 DS，而寄存器 BP 则使用 SS 段寄存器的内容作为段地址。这种寻址方式示于图 3.1(5)。

物理地址=16d×(DS)+(BX) + 8 位位移量

或(SI) 或 16 位位移量

或(DI)

或 物理地址=16d×(SS)+(BP)+ 8 位位移量

或 16 位位移量

例 3.6 MOV AX,COUNT[SI]

(也可表示为 MOV AX,[COUNT+SI])

其中 COUNT 为 16 位位移量的符号地址。

如果 (DS)=3000H,(SI)=2000H,

COUNT=3000H

则 物理地址=30000+2000+3000

=35000H

指令执行情况如图 3.5 所示，执行结果是：

(AX) = 1234H

这种寻址方式同样可用于表格处理，表格的首地址可设置为 COUNT，利用修改基址或变址寄存器的内容来取得表格中的值。

直接变址寻址方式也可以使用段跨越前缀。例如：

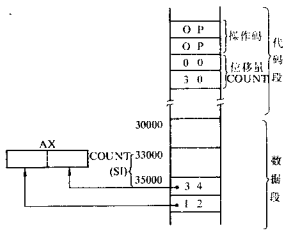


图 3.5 例 3.6 的执行情况

MOV DL,ES：STRING[SI]

3.1.1.6 基址变址寻址方式(Based indexed addressing)

操作数的有效地址是一个基址寄存器和一个变址寄存器的内容之和。两个寄存器均由指令指定。如基址寄存器为 BX 时，段寄存器使用 DS；如基址寄存器为 BP 时，段寄存器则用 SS。因此物理地址为：

物理地址＝16d×(DS)＋(BX)＋(SI)
或(DI)

或　　物理地址＝16d×(SS)＋(BP)＋(SI)
或(DI)

这种寻址方式如图 3.1(6)所示。

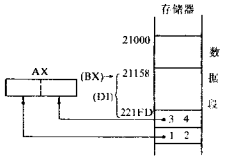


图 3.6　例 3.7 的执行情况

例 3.7　MOV　AX,[BX][DI]
(或写为：　MOV　AX,[BX＋DI])
如　(DS)＝2100H
(BX)＝0158H
(DI)＝10A5H
则　EA＝0158＋10A5＝11FDH
物理地址＝21000＋11FD＝221FDH

指令执行情况如图 3.6 所示。执行结果(AX)＝1234H，

这种寻址方式同样适用于数组或表格处理，首地址可存放在基址寄存器中，而用变址寄存器来访问数组中的各个元素。由于两个寄存器都可以修改，所以它比直接变址方式更加灵活。

此种寻址方式使用段跨越前缀时的格式为：

MOV　AX,ES：[BX][SI]

3.1.1.7　相对基址变址寻址方式(Relative based indexed addressing)

操作数的有效地址是一个基址寄存器和一个变址寄存器的内容和 8 位或 16 位位移量之和。同样，当基址寄存器为 BX 时，使用 DS 为段寄存器；而当基址寄存器为 BP 时，则使用 SS 为段寄存器。因此物理地址为：

物理地址＝16d×(DS)＋(BX)＋(SI)＋8 位位移量
或(DI)　或 16 位位移量

或　　物理地址＝16d×(SS)＋(BP)＋(SI)＋8 位位移量
或(DI)　或 16 位位移量

这种寻址方式可图示为图 3.1(7)。

例 3.8　MOV　AX,MASK[BX][SI](也可以写成 MOV　AX，MASK[BX＋SI]或 MOV　AX,[MASK＋BX＋SI])。
如　(DS)＝3000H,(BX)＝2000H,(SI)＝1000H,MASK＝0250H,
则　物理地址＝16d×(DS)＋(BX)＋(SI)＋MASK
＝30000＋2000＋1000＋0250
＝33250H

指令执行情况如图 3.7 所示。执行结果(AX)＝1234H。

这种寻址方式为堆栈处理提供了方便。一般(BP)可指向栈顶，从栈顶到数组的首地址可用位移量表示，变址寄存器可用来访问数组中的某个元素。

3.1.2　与转移地址有关的寻址方式

这种寻址方式用来确定转移指令及 CALL 指令的转向地址。

3.1.2.1 段内直接寻址(Intrasegment direct addressing)

转向的有效地址是当前IP寄存器的内容和指令中指定的8位或16位位移量之和。如图3.8(1)所示。

这种方式的转向有效地址用相对于当前IP值的位移量来表示，所以它是一种相对寻址方式。指令中的位移量是转向的有效地址与当前IP值之差，所以当这一程序段在内存中的不同区域运行时，这种寻址方式的转移指令本身不会发生变化，这是符合程序的再定位要求的。这种寻址方式适用于条件转移及无条件转移指令，但是当它用于条件转移指令时，位移量只允许8位。无条件转移指令在位移量为8位时称为短跳转。

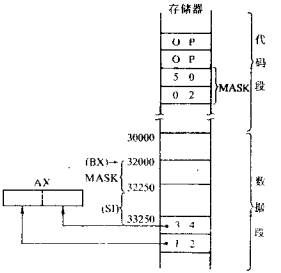


图3.7 例3.8的执行情况

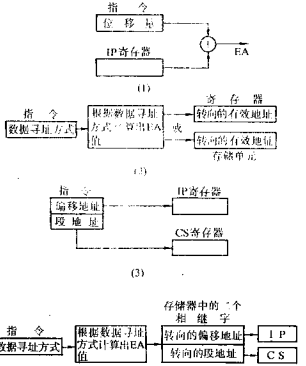


图3.8 与转移地址有关的寻址方式

(1)段内直接 (2)段内间接 (3)段间直接 (4)段间间接

指令的汇编语言格式表示为：

```
JMP     NEAR PTR PROG1A
JMP     SHORT QUEST
```

其中，PROG1A和QUEST均为转向的符号地址，在机器指令中，用位移量来表示。在汇编指令中，如果位移量为16位，则在符号地址前加操作符NEAR PTR，如果位移量为8位，则在符号地址前加操作符SHORT。

3.1.2.2 段内间接寻址(Intrasegment indirect addressing)

转向有效地址是一个寄存器或是一个存储单元的内容。这个寄存器或存储单元的内容可以用数据寻址方式中除立即数以外的任何一种寻址方式取得，所得到的转向的有效地址用来取代IP寄存器的内容。此种寻址方式如图3.8(2)所示。

这种寻址方式以及以下的两

种段间寻址方式都不能用于条件转移指令。也就是说，条件转移指令只能使用段内直接寻址的8位位移量，而JMP和CALL指令则可用四种寻址方式中的任何一种。

段内间接寻址转移指令的汇编格式可以表示为：

```
JMP    BX
JMP    WORD PTR[BP+TABLE]
```

等。其中WORD PTR为操作符，用以指出其后的寻址方式所取得的转向地址是一个字的有效地址，也就是说它是一种段内转移。

以上两种寻址方式均为段内转移，所以直接把求得的转移的有效地址送到IP寄存器就可以了。如果需要计算转移的物理地址，则计算公式应该是：

$$物理地址=16d\times(CS)+EA$$

其中EA即为上述转移的有效地址。

下面举例说明在段内间接寻址方式的转移指令中，转移的有效地址的计算方法。

假设：(DS)=2000H，(BX)=1256H，(SI)=528FH，

位移量=20A1H，(232F7H)=3280H，(264E5H)=2450H。

例3.9 JMP BX

则执行该指令后(IP)=1256H

例3.10 JMP TABLE[BX]

则执行该指令后(IP)=(16d×(DS)+(BX)+位移量)

=(20000+1256+20A1)

=(232F7)

=3280H

例3.11 JMP [BX][SI]

则指令执行后(IP)=(16d×(DS)+(BX)+(SI))

=(20000+1256+528F)

=(264E5)

=2450H

3.1.2.3 *段间直接寻址*(Intersegment direct addressing)

指令中直接提供了转向段地址和偏移地址，所以只要用指令中指定的偏移地址取代IP寄存器的内容，用指令中指定的段地址取代CS寄存器的内容就完成了从一个段到另一个段的转移操作，如图3.8(3)所示。

指令的汇编语言格式可表示为：

```
JMP    FAR PTR NEXTROUTINT
```

其中，NEXTROUTINT为转向的符号地址，FAR PTR则是表示段间转移的操作符。

3.1.2.4 *段间间接寻址*(Intersegment indirect addressing)

用存储器中的二个相继字的内容来取代IP和CS寄存器中的原始内容以达到段间转移的目的。这里存储单元的地址是由指令指定除立即数方式和寄存器方式以外的任何一种数据寻址方式取得，如图3.8(4)所示。

这种指令的汇编语言格式可表示为：

```
JMP    DWORD PTR[INTERS+BX]
```

其中，[INTERS+BX]说明数据寻址方式为直接变址寻址方式，DWORD PTR为双字操作符，说

明转向地址需取双字为段间转移指令。

3.2 IBM PC 机的机器语言指令概况

我们用汇编语言编写的汇编语言程序输入计算机后，由机器提供的“汇编程序”将它翻译成由机器指令组成的机器语言程序，才能由计算机识别并执行。因此汇编语言程序是由汇编程序翻译成可执行的机器语言程序的，一般说来，这一过程不必由人来干预。我们这里只介绍一下基本原理，以便在必要时也可完成类似的工作。

机器语言指令由操作码和地址码两部分组成，下面分别加以说明。

3.2.1 操作码的机器语言表示

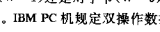
IBM PC 机的机器语言指令是多字节指令，一条指令可以由 1～7 个字节组成。指令的操作码(用 OP 表示)采用二进制代码表示本指令所执行的操作，在 IBM PC 机中，它通常用指令的第一个字节表示，有时由于用 8 位还不够，因此在指令的第二个字节中还可能占有 3 位操作码，除此以外的其他字节则用来表示地址码。

在多数操作码中，常使用某些位来指示某些信息。例如

其中 W 位用来指示本指令是对字(W=1)还是对字节(W=0)进行操作。

d 值在双操作数指令中才有效。IBM PC 机规定双操作数指令的两个操作数必须有一个操作数放在寄存器中，d 位指定寄存器用于目的操作数(d=1)还是源操作数(d=0)。

另外，当使用立即方式寻址时，操作码中用 S 位表示符号扩展：

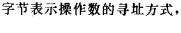


如立即数为 8 位，但要求扩展成 16 位数(高位字节按低位字节的最高有效位作符号扩展)时，S 位为 1。因此，当指令作字节操作时，SW=00；当指令有 16 位立即数且作字操作时 SW=01；*而当指令有 8 位立即数但需要经符号扩展成 16 位立即数作字操作时，则 SW=11*。

由于 IBM PC 的指令格式很多，这里我们只作一些基本情况介绍，必要时读者可通过查手册来了解详细情况。

3.2.2 寻址方式的机器语言表示

IBM PC 机用一个寻址方式字节表示操作数的寻址方式，它通常是机器指令的第二个字节。寻址方式字节可表示如下：



其中 reg 表示寄存器方式，在双操作数指令的情况下规定必须有一个操作数在寄存器中，该寄存器由 reg 字段指定。它与操作码字节中的 W 位相结合确定的寄存器如表 3.1 所示。

mod 字段与 r/m(register/memory)字段结合在一起确定另一个操作数的寻址方式。

- mod=11 为寄存器方式，由 r/m 的内容确定选用哪个寄存器，具体值由表 3.2 确定。

表 3.1 reg 字段所对应的寄存器

reg	w=1	w=0
000	AX	AL
001	CX	CL
010	DX	DL
011	BX	BL
100	SP	AH
101	BP	CH
110	SI	DH
111	DI	BH

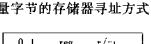
表 3.2 存储器寻址方式

mod / r/m	00	01	10	11	
				w=0	w=1
000	(BX)+(SI) DS	(BX)+(SI)+D8 DS	(BX)+(SI)+D16 DS	AL	AX
001	(BX)+(DI) DS	(BX)+(DI)+D8 DS	(BX)+(DI)+D16 DS	CL	CX
010	(BP)+(SI) SS	(BP)+(SI)+D8 SS	(BP)+(SI)+D16 SS	DL	DX
011	(BP)+(DI) SS	(BP)+(DI)+D8 SS	(BP)+(DI)+D16 SS	BL	BX
100	(SI) DS	(SI)+D8 DS	(SI)+D16 DS	AH	SP
101	(DI) DS	(DI)+D8 DS	(DI)+D16 DS	CH	BP
110	D16 DS	(BP)+D8 SS	(BP)+D16 SS	DH	SI
111	(BX) DS	(BX)+D8 DS	(BX)+D16 DS	BH	DI

其他三种寻址方式则为操作数在存储器中时的寻址方式，由表 3.2 表示，下面分别加以说明：

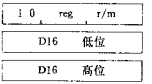
· mod=00 为无位移量字节的存储器寻址方式。其中当 r/m=110 时指定为直接寻址方式，此时寻址方式字节之后跟有 16 位位移量 D16，用来指出操作数的有效地址。

· mod=01 为带有一个位移量字节的存储器寻址方式，因此指令中应有



其中 D8 为 8 位位移量，是带符号数，当用来计算有效地址时，可自动将符号扩展到 16 位。

· mod=10 为带有二个位移量字节的存储器寻址方式，因此指令中有



其中 D16 为 16 位位移量，它也是带符号数。

表 3.2 中每项下的段寄存器是指无段跨越前缀的情况下所使用的隐含的段寄存器。如果指令中指定段跨越前缀，则在机器语言中使用放在指令之前的一个字节来表示，如下所示：

其中 001 及 110 均为段前缀标志，SEG 则指定四个段寄存器中的一个，如表 3.3 所示。

表 3.3 SEG 字段所对应的段寄存器

SEG	段寄存器
00	ES
01	CS
10	SS
11	DS

IBM PC 机规定下列二种情况下不允许使用段跨越前缀：

1. 访问堆栈的指令(如 PUSH 和 CALL 等)使用 SP 作为偏移地址指针，只能使用 SS 作为段寄存器。

2. 串处理指令规定源寄存器使用 SI，源串在 DS 段中，目的寄存器使用 DI，目的串必须在 ES 段中。这里如果要用段跨越前缀则只能用于源而不能用于目的，也就是说，SI 可以更换段寄存器，而 DI 则只能使用 ES 作为段寄存器，不允许用段跨越前缀更换。

3. 指令只能存放在代码段中。

根据以上说明，我们对 IBM PC 机的机器指令情况已经有了一般的了解，下面再以加法指令为例，对机器指令作一些具体的分析。

3.2.3 加法的机器指令举例

加法指令的汇编语言格式为：

```
ADD     DST,SRC
```

其中 SRC 表示源操作数地址，DST 表示目的操作数地址。加法指令所执行的操作可表示为：

(DST)← (DST)+(SRC)

根据不同的寻址方式，加法指令可以有三种格式：

1. ADD　　mem/reg1,mem/reg2

其中 mem/reg1 表示目的操作数地址，mem/reg2 表示源操作数地址，它们都可以指定一个寄存器或一个存储单元的内容作为操作数。这些操作数可以是 8 位的或 16 位的，它们可同时指定

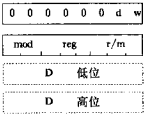
两个寄存器，但不能同时指定两个存储单元，因此它的工作方式可以是以下三种方式中的一种：

a) 寄存器与寄存器的内容相加，结果存入寄存器中。

b) 寄存器与存储单元的内容相加，结果存入存储单元中。

c) 存储单元与寄存器的内容相加，结果存入寄存器中。

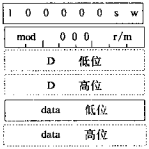
这种寻址方式的机器语言为：



其中 000000 为操作码。W=0 为字节运算，W=1 为字运算。d=0 时，由 mod、r/m 决定目的 mem/reg1，reg 决定源 mem/reg2；d=1 时，由 mod、r/m 决定源 mem/reg2，reg 决定目的 mem/reg1，虚线框中的位移量根据寻址方式确定其有无。

2. ADD mem/reg,data

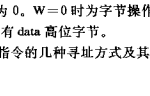
把存储在代码段中的立即数与指定的寄存器或存储单元的内容相加，结果存放于寄存器或存储单元中。指令的机器语言为：



其中第一字节的 100000 及第二字节的 000 均为操作码。W=0 时作字节运算；W=1 时作字运算。当 W=0 时，S 位无效；当 W=1 时，S=0 时立即数为 16 位；而 S=1 时由 8 位立即数的低位符号扩展形成 16 位立即数。位移量则根据寻址方式确定其有无。

3. ADD ac,data

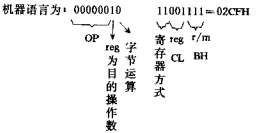
把存储在代码段中的立即数与 AL(W=0 时)或 AX(W=1 时)中的内容相加，结果送回 AL 或 AX 中。指令的机器语言为：



其中 000001 为操作码，S 位指定为 0。W=0 时为字节操作，使用 AL 寄存器；W=1 时为字操作，使用 AX 寄存器。当 W=1 时有 data 高位字节。

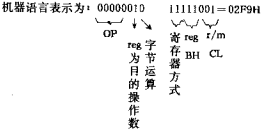
下面用具体例子来说明加法指令的几种寻址方式及其机器语言表示：

例 3.12 ADD CL, BH



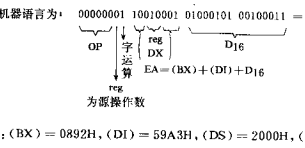
如指令执行前：(CL)＝29H，(BH)＝4DH，则指令执行后(CL)＝76H，(BH)则保持原值不变。

例 3.13 ADD BH, CL



如指令执行前：(CL)＝29H，(BH)＝4DH，则指令执行后(BH)＝76H，而(CL)则保持原值不变。

例 3.14 ADD DISP[BX][DI], DX



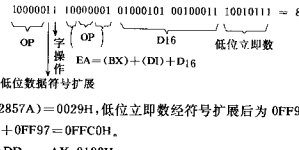
如指令执行前：(BX)＝0892H，(DI)＝59A3H，(DS)＝2000H，(DX)＝04EDH，(2857A)＝0029H，

可求得目的操作数地址＝20000＋0892＋59A3＋2345

＝2857AH

指令执行后，(2857AH)＝0029＋04ED＝0516H，(DX)的内容则不变。

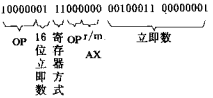
例 3.15 ADD DISP[BX][DI], −105D



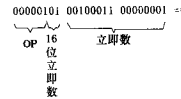
如指令执行前(2857A)＝0029H，低位立即数经符号扩展后为 0FF97H，则指令执行后(2857A)＝0029＋0FF97＝0FFC0H。

例 3.16 ADD AX, 0123H

机器指令为：



如指令执行前(AX)=4567H,则指令执行后(AX)=4567+0123=468AH。这里的机器指令为长格式(格式 2)的表示法,这条指令也可用短格式(格式 3)表示为:



以上五个例子可以说明用机器语言表示指令的方法,也可以看出,即使是同一种指令,不同的寻址方式也会使机器指令占据不同的存储空间。由指令序列组成的程序顺序地存放在存储器的代码段中,例如有下面的程序段:

```
ADD   CL,BH
ADD   DISP[BX][DI],DX
ADD   DISP[BX][DI],-105D
ADD   AX,0123H
      ⋮
      ⋮
```

则当程序执行时,在存储器的代码段中(假设从 00200 单元开始)该程序段占用存储空间的情况如下所示:

存储器地址	内容	
00200	02	第 1 条指令
00201	0F	
00202	01	第 2 条指令
00203	91	
00204	45	
00205	23	
00206	83	第 3 条指令
00207	81	
00208	45	
00209	23	
0020A	97	
0020B	05	第 4 条指令
0020C	23	
0020D	01	
0020E	⋮	
0020F	⋮	
00210		

可见运行着的程序是占有一定的存储空间的,由于不同指令或者同一种指令的不同寻址方式都会影响指令所占空间的大小,因此在编制程序时应该通过选择合理的算法、数据结构、寻址

方式等来节省存储空间。

3.2.4 指令的执行时间

一条指令的执行时间是取指令、取操作数、执行指令及传送结果各个阶段所需时间的总和。要详细讨论各种指令的执行时间是个比较复杂的问题，在这里我们只能讨论一些一般概念。

由于指令是存储在存储器中的，因此运算器要执行指令就需要先访问存储器，把指令取到控制器中进行分析。IBM PC 机采用预取指令的方式工作，所以取指令的时间与其他阶段的时间是重叠的，可以不予考虑。这样，执行一条指令的时间就是指令的基本执行时间及取、存操作数时间的总和。指令的基本执行时间因指令的不同而异，相互之间有很大的差别，而存、取操作数时间又因不同的寻址方式而有所不同，当需要访问存储器取得操作数时还需要考虑计算有效地址 EA 所需要的时间。表 3.4～3.6 分别表示执行不同指令所需的时间，执行加法指令时不同的寻址方式所需的时间，以及在不同的寻址方式下计算 EA 所需的时间。在这些表格中，所有的时间是以时钟周期数表示的(计算机是用节拍来控制顺序工作的，每个节拍可视为一个时钟周期)。

表 3.4 指令的基本执行时间举例

指令	寻址方式	时钟周期数
加法 ADD	寄存器—寄存器	3
传送 MOV	寄存器—寄存器	2
整数乘法 IMUL	16 位寄存器乘	128～154
整数除法 IDIV	16 位寄存器除	165～184
移位	寄存器移 1 位	2
无条件转移 JMP	直接	15
条件转移	不转移	4
	转移	16

表 3.5 加法指令的执行时间

操作数的寻址方式	时钟周期数	访问存储器次数
寄存器到寄存器	3	0
存储器到寄存器	9＋EA	1
寄存器到存储器	16＋EA	2
立即数到寄存器	4	0
立即数到存储器	17＋EA	2

注：EA 为计算有效地址所需时间

表 3.6 计算有效地址 EA 所需时间

寻址方式	时钟周期数
直接	6
寄存器间接	5
寄存器相对	9
基址变址	7～8
相对基址变址	11～12

从表 3.4～3.6 可以看出，不仅不同指令的执行时间差别很大，而且同一种指令使用不同寻址方式时执行时间的差别也是很大的，为了使读者对指令的执行时间有一些具体的了解，下面以加法指令的执行时间为例，作一些具体的计算：

假设时钟频率为 5MHz，则一个时钟周期即为 0.2μs(微秒)。

· 寄存器-寄存器方式的 ADD 指令需要 t=3×0.2=0.6(μs)。

· 存储器-寄存器方式，如存储器使用相对基址变址方式，则

:=(9+EA)×0.2=(9+12)×0.2=4.2(μs)。

如果在此种寻址方式下，求得存储器的有效地址为奇数，则根据 IBM PC 的规定，需要多访问一次存储器，要增加 4 个时钟周期数，则

$$t = (9 + 12 + 4) \times 0.2 = 25 \times 0.2 = 5(\mu s).$$

· 寄存器-存储器方式，如存储器使用相对基址变址方式，则

$$t = (16 + EA) \times 0.2 = (16 + 12) \times 0.2 = 5.6(\mu s).$$

如果此时求得的是奇地址，则需要时间即为

$$t = (16 + 12 + 8) \times 0.2 = 36 \times 0.2 = 7.2(\mu s).$$

可见，即使是同一种 ADD 指令，不同的寻址方式可以使它的执行时间相差一个数量级，因此合理地选择指令及寻址方式的使用是很重要的。完成同样功能的不同程序，可能在占有空间和执行时间上有很大差别，因此在编制程序时，如果对程序所占用的存储空间或程序的执行时间要求不高，那就只要根据题意编制出合乎要求的程序就可以了，当然也应该尽量在空间和时间上提高程序运行的效率。有时，对于程序所占有的存储空间或者对于程序执行的时间要求很高，在这种情况下应仔细斟酌程序的算法、数据结构以及指令与寻址方式的选用，以便编制出符合要求的程序。

3.3 IBM PC 机的指令系统

IBM PC 机的指令系统可以分为以下 6 组：

数据传送指令	串处理指令
算术指令	控制转移指令
逻辑指令	处理机控制指令

下面分别加以说明。

3.3.1 数据传送指令

数据传送指令负责把数据、地址或立即数传送到寄存器或存储单元中。它又可分为四种，分别说明如下：

3.3.1.1 通用数据传送指令

MOV (Move) 传送

PUSH (Push onto the stack) 进栈

POP (Pop from the stack) 出栈

XCHG (Exchange) 交换

· MOV 传送指令

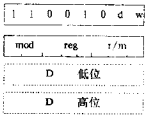
格式为：MOV DST,SRC

执行操作：(DST)←(SRC)

其中 DST 表示目的操作数，SRC 表示源操作数。

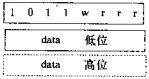
MOV 指令的机器语言可以有 7 种格式：

1. MOV　　mem/reg1，mem/reg2



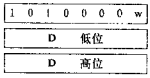
当然，双操作数指令不允许两个操作数都使用存储器，因而两个操作数中必须有一个是寄存器。这种方式不允许指定段寄存器。

2. MOV　　reg，data



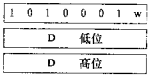
其中 rrr 字段指定寄存器。当然，这种方式也不允许指定段寄存器。

3. MOV　　ac，mem

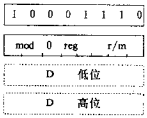


其中 ac 为累加器，D 给出存储单元的偏移地址。

4. MOV　　mem，ac

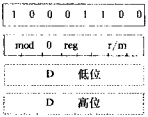


5. MOV　　segreg，mem/reg

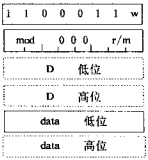


其中 reg 指定段寄存器，但不允许使用 CS 寄存器。此外，这条指令执行完后不响应中断，要等下一条指令执行完后才可能响应中断(有关中断处理问题，将在第八章中说明)。

6. MOV　　mem/reg，segreg



7. MOV mem/reg, data



这种方式的目的操作数只用存储器寻址方式而不用寄存器方式。

以上七种方式说明 MOV 指令可以在 CPU 内或 CPU 和存储器之间传送字或字节，它传送的信息可以从寄存器到寄存器，立即数到寄存器，立即数到存储单元，从存储单元到寄存器，从寄存器到存储单元，从寄存器或存储单元到除 CS 外的段寄存器(注意，立即数不能直接送段寄存器)，从段寄存器到寄存器或存储单元。但是 MOV 指令的目的操作数不允许用立即数方式，也不允许用 CS 寄存器，而且除源操作数为立即数的情况外，两个操作数中必须有一个是寄存器。也就是说，不允许用 MOV 指令在两个存储单元之间直接传送数据。此外，也不允许在两个段寄存器之间直接传送信息。还应该注意的是 MOV 指令不影响标志位。

例 3.17 MOV AX,DATA_SEG
MOV DS,AX

段地址必须通过寄存器如 AX 寄存器送到 DS 寄存器。

例 3.18 MOV AL,'E'

把立即数(字符 E 的 ASCII 码)送到 AL 寄存器。

例 3.19 MOV BX,OFFSET TABLE

把 TABLE 的偏移地址(而不是内容!)送到 BX 寄存器。其中 OFFSET 为属性操作符，表示应把其后跟着的符号地址的值(不是内容)作为操作数。

例 3.20 MOV AX,Y[BP][SI]

把地址为 16d×(SS)+(BP)+(SI)+位移量 Y 的存储单元的内容送给 AX 寄存器。

· PUSH 进栈指令

格式为 PUSH SRC

执行操作： (SP)←(SP)−2
((SP)+1,(SP))←(SRC)

· POP 出栈指令

格式为 POP DST

执行操作： (DST)←((SP)+1,(SP))
(SP)←(SP)+2

这是两条堆栈的进栈和出栈指令。堆栈是以"后进先出"方式工作的一个存储区，它必须存在于堆栈段中，因而其段地址存放于SS寄存器中。它只有一个出入口，所以只有一个堆栈指针寄存器SP，SP的内容在任何时候都指向当前的栈顶，所以PUSH和POP指令都必须根据当前SP的内容来确定进栈或出栈的存储单元，而且必须及时修改指针，以保证(SP)指向当前的栈顶。

堆栈的存取必须以字为单位(IBM PC中不允许字节堆栈)，所以PUSH和POP指令只能作字操作。它们可以使用除立即数以外的其他寻址方式。指令也可以指定段寄存器作为操作数，但POP指令不允许用CS寄存器。

这两条堆栈指令不影响标志位。

例 3.21 PUSH AX

指令执行情况如图3.9所示。

例 3.22 POP AX

指令执行情况如图3.10所示。

堆栈在计算机工作中起着重要的作用，如果在程序中要用到某些寄存器，但它的内容却在将来还有用，这时就可以用堆栈把它们保存下来，然后到必要时再恢复其原始内容。例如：

```
PUSH    AX
PUSH    BX
        :
    其间程序用到AX和BX寄存器
        :
POP     BX
POP     AX
```

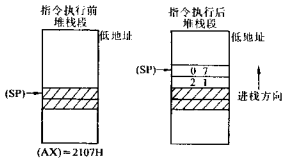


图 3.9 PUSH AX 指令的执行情况

堆栈在子程序结构的程序以及中断程序中也很有用，这将在以后加以说明。

· XCHG 交换指令

格式为 XCHG OPR1,OPR2

执行操作：(OPR1)←→(OPR2)

其中OPR表示操作数。该指令的两个操作数中必须有一个在寄存器中，因此它可以在寄存器之间或者在寄存器和存储器之间交换信息，但不允许使用段寄存器。指令允许字或字节操作，且不影响标志位。

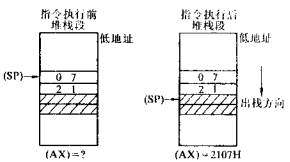


图 3.10 POP AX 指令的执行情况

例 3.23 XCHG BX,[BP+SI]

如指令执行前：

(BX)=6F30H,(BP)=0200H,(SI)=0046H,(SS)=2F00H,(2F246H)=4154H

OPR2的物理地址=2F000+0200+0046=2F246

则指令执行后：

(BX)=4154H (2F246H)=6F30H

3.3.1.2 累加器专用传送指令

IN (Input) 输入

OUT (Output) 输出

XLAT(Translate) 换码

这组指令只限于使用累加器 AX 或 AL 传送信息。

• IN 输入指令

长格式为：	IN	AL,PORT (字节)
	IN	AX,PORT (字)
执行的操作：	(AL)←(PORT) (字节)	
	(AX)←(PORT+1,PORT) (字)	
短格式为：	IN	AL,DX (字节)
	IN	AX,DX (字)
执行的操作：	(AL)←((DX)) (字节)	
	(AX)←((DX)+1,(DX)) (字)	

• OUT 输出指令

长格式为：	OUT	PORT,AL (字节)
	OUT	PORT,AX (字)
执行的操作：	(PORT)←(AL) (字节)	
	(PORT+1,PORT)←(AX) (字)	
短格式为：	OUT	DX,AL (字节)
	OUT	DX,AX (字)
执行操作为：	((DX))←(AL) (字节)	
	((DX)+1,(DX))←(AX) (字)	

在 IBM PC 机里，所有 I/O 端口与 CPU 之间的通信都由 IN 和 OUT 指令来完成。其中 IN 完成从 I/O 到 CPU 的信息传送，而 OUT 完成从 CPU 到 I/O 的信息传送。CPU 只能用累加器(AL 或 AX)接收或发送信息。外部设备最多可有 65536 个 I/O 端口，端口号(即外设的端口地址)为 0000～FFFFH。其中前 256 个端口(0～FFH)可以直接在指令中指定，这就是长格式中的 PORT，此时机器指令用二个字节表示，第二个字节就是端口号。所以用长格式时可以在指令中直接指定端口号，但只限于外设的前 256 个端口。当端口号≥256 时，只能使用短格式，此时，必须先把端口号放到 DX 寄存器中(端口号可以从 0000～0FFFFH)，然后再用 IN 或 OUT 指令来传送信息。必须注意，这里的端口号或 DX 的内容均为地址，而传送的是端口中的信息，而且在用短格式时 DX 的内容就是端口号本身，不需要由任何段寄存器来修改它的值。

IN 和 OUT 指令提供了字和字节两种使用方式，选用哪一种，则取决于外设端口宽度。如端口宽度只有 8 位则只能用字节指令传送信息。

输入、输出指令不影响标志位。

例 3.24
```
IN      AX,28H
MOV     DATA_WORD,AX
```
这两条指令把端口 28 的内容经过 AX 传送到存储单元 DATA_WORD 中。

例 3.25
```
MOV     DX,3FCH
IN      AX,DX
```

从端口 03FCH 送一个字到 AX 寄存器。

例 3.26 OUT 5,AL

从 AL 寄存器输出一个字节到端口 5。

· XLAT 换码指令

格式为： XLAT OPR

或 XLAT

执行的操作： (AL)←((BX)+(AL))

经常需要把一种代码转换为另一种代码，例如把字符的扫描码转换成 ASCII 码或者把数字 0～9 转换成 7 段数码管所需要的相应代码等，XLAT 就是为这种用途所设置的指令。在使用这条指令以前，应先建立一个字节表格，表格的首地址提前存入 BX 寄存器，需要转换的代码应该是相对于表格首地址的位移量也提前存放在 AL 寄存器中，表格的内容则是所要换取的代码，该指令执行后就可在 AL 中得到转换后的代码。该指令可用 XLAT 或 XLAT OPR 两种格式中的任一种，使用 XLAT OPR 时，OPR 为表格的首地址(一般为符号地址)，但在这里的 OPR 只是为提高程序的可读性而设置的，指令执行时只使用预先已存入 BX 中的表格首地址，而并不用汇编格式中指定的值。该指令不影响标志位。

例 3.27 如(BX)=0040H，(AL)=0FH，(DS)=F000H

所建立的表格如图 3.11 所示。

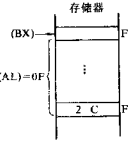


图 3.11 例 3.27 所用的表格

指令

XLAT

把 F0000+0040+0F=F004F 的内容送 AL，所以指令执行后

(AL)=2CH

即指令把 AL 中的代码 0FH 转换为 2CH。

必须注意，由于 AL 寄存器只有 8 位所以表格的长度不能超过 256。

3.3.1.3 地址传送指令

LEA (Load effective address) 有效地址送寄存器

LDS(Load DS with Pointer) 指针送寄存器和 DS

LES(Load ES with Pointer) 指针送寄存器和 ES

这一组指令完成把地址送到指定寄存器的功能。

· LES 有效地址送寄存器指令

格式为： LEA REG,SRC

执行的操作： (REG)←SRC

指令把源操作数的有效地址送到指定的寄存器中。

· LDS 指针送寄存器和 DS 指令

格式为 LDS REG,SRC

执行的操作：(REG)←(SRC)

(DS)←(SRC+2)

把源操作数指定的 4 个相继字节送到由指令指定的寄存器及 DS 寄存器中。该指令常指定 SI 寄存器。

· LES　指针送寄存器和 ES 指令

格式为　　LES　REG,SRC

执行的操作：(REG)←(SRC)

(ES)←(SRC+2)

把源操作数指定的 4 个相继字节送到由指令指定的寄存器及 ES 寄存器中。该指令常指定 DI 寄存器。

以上三条指令指定的寄存器不能使用段寄存器，且源操作数必须使用除立即数方式及寄存器方式以外的其它寻址方式。这些指令不影响标志位。

本组指令把变量的偏移地址(LEA)或段地址和偏移地址(LDS 和 LES)送给寄存器，以提供访问变量的工具。

例 3.28　LEA　BX,[BX+SI+0F62H]

如指令执行前　(BX)=0400H,(SI)=003CH

则指令执行后　(BX)=0400+003C+0F62=139EH

必须注意，在这里 BX 寄存器得到的是偏移地址而不是该存储单元的内容。

例 3.29　LDS　SI,[10H]

如指令执行前　(DS)=C000H,(C0010H)=0180H,(C0012H)=2000H

则指令执行后　(SI)=0180H,(DS)=2000H

例 3.30　LES　DI,[BX]

如指令执行前　(DS)=B000H,(BX)=080AH,(0B080AH)=05AEH,(0B080CH)=4000H

则指令执行后　(DI)=05AEH,　(ES)=4000H

3.3.1.4　标志寄存器传送指令

LAHF (load AH with flags)标志送 AH

SAHF (store AH into flags)AH 送标志寄存器

PUSHF (push the flags)　标志进栈

POPF (pop the flags) 标志出栈

· LAHF　标志送 AH 指令

格式为：　LAHF

执行的操作：(AH)←(PSW 的低字节)

· SAHF　AH 送标志寄存器指令

格式为：　SAHF

执行的操作：(PSW 的低字节)←(AH)

· PUSHF　标志进栈指令

格式为：　PUSHF

执行的操作：(SP)←(SP)−2

((SP)+1,(SP))←(PSW)

· POPF　标志出栈指令

格式为：　POPF

执行的操作：(PSW)←((SP)+1,(SP))

(SP)←(SP)+2

这组指令中的 LAHF 和 PUSHF 不影响标志位。SAHF 和 POPF 则由装入的值来确定标志位的值。

3.3.2 算术指令

IBM PC 机的算术运算指令包括二进制运算及十进制运算指令。算术指令用来执行算术运算，它们中有双操作数指令，也有单操作数指令。如前所述，双操作数指令的两个操作数中除源操作数为立即数的情况外，必须有一个操作数在寄存器中。单操作数指令不允许使用立即数方式。算术指令的寻址方式，均遵循这一规则。

3.3.2.1 加法指令

ADD (add)加法

ADC (add with carry)带进位加法

INC (increment)加 1

· ADD 加法指令

格式： ADD DST,SRC

执行的操作：(DST)←(SRC)+(DST)

· ADC 带进位加法指令

格式： ADC DST,SRC

执行的操作：(DST)←(SRC)+(DST)+CF

其中 CF 为进位位的值。

· INC 加 1 指令

格式： INC OPR

执行的操作：(OPR)←(OPR)+1

以上三条指令都可作字或字节运算，而且除 INC 指令不影响 CF 标志外，它们都影响条件标志位。

条件标志(或称条件码)位中最主要的是 CF、ZF、SF、OF 四位，分别表示进位、结果为零、符号和溢出的情况。其中 ZF、SF 位的设置比较简单，在第二章中已经说明，这里不再赘述，在这里将进一步分析 CF 和 OF 位的设置情况。

执行加法指令时，CF 位是根据最高有效位是否有向高位的进位设置的。有进位时 CF=1，无进位时 CF=0。OF 位则根据操作数的符号及其变化情况来设置：若两个操作数的符号相同，而结果的符号与之相反时 OF=1，否则 OF=0。溢出位 OF 既然是根据数的符号及其变化来设置的，当然它是用来表示带符号数的溢出的，从其设置条件来看结论也是明显的。那么，进位位 CF 的意义是什么呢？在第一章说明补码运算规则的过程中，我们曾经提到在补码加、减法中，从最高有效位向高位的进位说明模运算中的进位自动丢失的现象，对结果并没有影响。那么，为什么还要保存 CF 位呢？

CF 位可以用来表示无符号数的溢出。由于无符号数的最高有效位只有数值意义而无符号意义，所以从该位产生的进位应该是结果的实际进位值，但是在有限数位的范围内就说明了结果的溢出情况，另一方面，它所保存的进位值有时是有用的。例如，双字长数运算时，可以利用进位值把低位字的进位计入高位字中等。这可以根据不同情况在程序中加以处理。

我们已经知道 8 位二进制数可以表示十进制数的范围是：无符号数为 0～255，带符号数为－128～＋127。16 位二进制数可以表示十进制数的范围是：无符号数为 0～65535，带符号数

为－32768～＋32767。下面我们以 8 位数为例分析一下数的溢出情况。

1）带符号数和无符号数都不溢出

二进制加法	看作无符号数	看作带符号数
0000 0100	4	＋4
＋0000 1011	＋11	＋(＋11)
0000 1111	15	＋15
	CF＝0	OF＝0

2）无符号数溢出

二进制加法	看作无符号数	看作带符号数
0000 0111	7	＋7
＋1111 1011	＋251	＋(－5)
0000 0010	258	＋2
↙	CF＝1	OF＝0
1	现为 2 结果错	

3）带符号数溢出

二进制加法	看作无符号数	看作带符号数
0000 1001	9	＋9
＋0111 1100	＋124	＋(＋124)
1000 0101	133	＋133
	CF＝0	OF＝1
		现为－123，结果错

4）带符号数和无符号数都溢出

二进制加法	看作无符号数	看作带符号数
1000 0111	135	(－121)
＋1111 0101	＋245	＋(－11)
0111 1100	380	－132
↙	CF＝1	OF＝1
1	现为 124，结果错	现为 124，结果错

上面的 4 个例子清楚地说明了，OF 位可以用来表示带符号数的溢出，CF 位则可用来表示无符号数的溢出。应该注意，如果 2），4）中的进位值以 $2^8=256$ 为其权值考虑在内时，得到的运算结果应该是正确的。

ADC 及 INC 对条件码的设置方法与 ADD 指令相同，但 INC 指令不影响 CF 标志。

例 3.31 ADD DX，0F0F0H

如指令执行前 (DX)＝4652H 则

```
    4652                0100 0110 0101 0010
  +F0F0      ⇒        +1111 0000 1111 0000
                      ---------------------
                       0011 0111 0100 0010
                       ↙
                       1
```

指令执行后 (DX)＝3742H，ZF＝0，SF＝0，CF＝1，OF＝0

结果正确。

例 3.32 下列指令序列执行两个双精度数的加法。设目的操作数存放在 DX 和 AX 寄存

器中，其中 DX 存放高位字。源操作数存放在 BX、CX 中，其中 BX 存放高位字。如指令执行前：

(DX)＝0002H，　　(AX)＝0F365H

(BX)＝0005H，　　(CX)＝0E024H

指令序列为

```
ADD    AX,CX
ADC    DX,BX
```

则第一条指令执行后

```
  F365
+ E024
------
  D389
   ↙
   1
```

(AX)＝0D389H，SF＝1，ZF＝0，CF＝1，OF＝0

第二条指令执行后

```
  0002
  0005
+    1(CF)
-------
  0008
```

(DX)＝0008H，SF＝0，ZF＝0，CF＝0，OF＝0

因此该指令序列执行完后

(DX)＝0008H，　(AX)＝D389H　结果正确。

可以看出，为实现双精度加法，必须用两条指令分别完成低位字和高位字的加法，而且在高位字相加时，应该使用 ADC 指令以便把前一条 ADD 指令作低位字加法所产生的进位值加入高位字之内。另外，带符号的双精度数的溢出，应该根据 ADC 指令的 OF 位来判别，而作低位加法用的 ADD 指令的溢出是无意义的。

3.3.2.2　减法指令

SUB (subtract)减法

SBB (subtract with borrow)带借位减法

DEC (Decrement)减 1

NEG (Negate)求补

CMP (Compare)比较

• SUB　减法指令

格式：　SUB　DST,SRC

执行的操作：(DST)←(DST)－(SRC)

• SBB　带借位减法指令

格式：　SBB　DST,SRC

执行的操作：(DST)←(DST)－(SRC)－CF

其中 CF 为进位位的值。

• DEC　减 1 指令

格式：　DEC　OPR

执行的操作：(OPR)←(OPR)－1

• NEG　求补指令

格式：　　NEG　　OPR

执行的操作：(OPR)←－(OPR)

亦即把操作数按位求反后末位＋1，因而执行的操作也可表示为：

(OPR)←0FFFFH－(OPR)＋1

· CMP　　比较指令

格式：　　CMP　　OPR1，OPR2

执行的操作：(OPR1)－(OPR2)

该指令与 SUB 指令一样执行减法操作，但它并不保存结果，只是根据结果设置条件标志位，CMP 指令后往往跟着一条条件转移指令，根据比较结果产生不同的程序分支。

以上五种指令均可作字或字节运算，而且除 DEC 不影响 CF 标志外，它们都影响条件标志位。

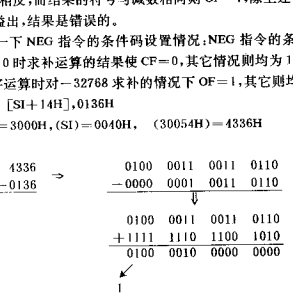
减法运算的条件码情况与加法类似。CF 位说明无符号数相减的溢出，同时它又确实是被减数的最高有效位向高位的借位值。OF 位则说明带符号数的溢出。这里不再详细讨论，只说明其设置方法。减法的 CF 值反映无符号数运算中的借位情况，因此当作为无符号数运算时，若减数＞被减数，此时有借位则 CF＝1，否则 CF＝0。或者也可以简单地用二进制减法运算中最高有效位向高位的进位情况来判别：有进位时 CF＝0，无进位时 CF＝1。减法的 OF 位的设置方法为：若两个数的符号相反，而结果的符号与减数相同则 OF＝1，除上述情况外 OF＝0。OF＝1 说明带符号数的减法溢出，结果是错误的。

这里再简单说明一下 NEG 指令的条件码设置情况：NEG 指令的条件码按求补后的结果设置：只有当操作数为 0 时求补运算的结果使 CF＝0，其它情况则均为 1。所以只有当字节运算时对－128 求补以及字运算时对－32768 求补的情况下 OF＝1，其它则均为 0。

例 3.33　SUB　　[SI＋14H]，0136H

如指令执行前　(DS)＝3000H，(SI)＝0040H，　(30054H)＝4336H

则指令执行后

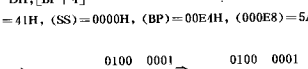


所以，(30054H)＝4200H，SF＝0，ZF＝0，CF＝0，OF＝0

例 3.34　SUB　　DH，[BP＋4]

如指令执行前　(DH)＝41H，(SS)＝0000H，(BP)＝00E4H，(00E8)＝5AH

则指令执行后



所以，(DH)＝0E7H，SF＝1，ZF＝0，CF＝1，OF＝0

例 3.35　设 X，Y，Z 均为双精度数，它们分别存放在地址为 X，X＋2；Y，Y＋2；Z，

Z+2 的存储单元中，存放时高位字在高地址中，低位字在低地址中。下列指令序列实现

$$W \leftarrow x+y+24-z$$

并用 W 和 W+2 单元存放运算结果。

```
MOV   AX,X      ;
MOV   DX,X+2    ;  }
ADD   AX,Y      ;  } X+Y
ADC   DX,Y+2    ;
ADD   AX,24     ;  }
ADC   DX,0      ;  } +24
SUB   AX,Z      ;  }
SBB   DX,Z+2    ;  } -Z
MOV   W,AX      ;  }
MOV   W+2,DX    ;  } 结果存入W,W+2
```

3.3.2.3 乘法指令

MUL (Unsigned Multiple)无符号数乘法

IMUL (Signed Multiple)带符号数乘法

· MUL　无符号数乘法指令

格式：　MUL　SRC

执行的操作：

字节操作数：(AX)←(AL) * (SRC)

字操作数：(DX,AX)←(AX) * (SRC)

· IMUL　带符号数乘法指令

格式：　IMUL　SRC

执行的操作：与 MUL 相同，但必须是带符号数，而 MUL 是无符号数。

在乘法指令里，目的操作数必须是累加器，字运算为 AX，字节运算为 AL。两个 8 位数相乘得到的是 16 位乘积存放在 AX 中，两个 16 位数相乘得到的是 32 位乘积，存放在 DX,AX 中，其中 DX 存放高位字，AX 存放低位字。指令中的源操作数可以使用除立即数方式以外的任一种寻址方式。

MUL 指令和 IMUL 指令的使用条件是由数的格式决定的。很明显(11111111b) * (11111111b)当把它看作无符号数时应为 255d×255d=65025d，而把它看作带符号数时则为(−1)×(−1)=1。因此必须根据所要相乘数的格式来决定选用哪一种指令。

乘法指令对除 CF 和 OF 以外的条件码位无定义(注意：无定义的意义和不影响不同，无定义是指指令执行后这些条件码位的状态不定，而不影响则是指该指令的结果并不影响条件码，因而条件码应保持原状态不变。)，对于 MUL 指令，如果乘积的高一半为 0，即字节操作的(AH)或字操作的(DX)为 0，则 CF 和 OF 均为 0；否则(即字节操作时的(AH)或字操作时的(DX)≠0)则 CF 和 OF 均为 1。这样的条件码设置可以用来检查字节相乘的结果是字节还是字，或者可以检查字相乘的结果是字还是双字。对于 IMUL 指令，如果乘积的高一半是低一半的符号扩展则 CF 和 OF 均为 0，否则就均为 1。

例 3.36　如(AL)=0B4H，(BL)=11H 求执行指令

IMUL　BL 和 MUL　BL 后的乘积值。

(AL)=0B4H 为无符号数的 180D，带符号数的 −76D，

(BL)＝11H 为无符号数的 17D，带符号数的 17D，执行 IMUL BL 的结果为

(AX)＝0FAF4H＝－1292D

CF＝OF＝1

执行 MUL BL 的结果为

(AX)＝0BF4H＝3060D

CF＝OF＝1

3.3.2.4 除法指令

DIV (Unsigned divide)无符号数除法

IDIV (Signed divide)带符号数除法

由于使用除法指令的需要，这里顺便介绍两条符号扩展指令：

CBW (Convert byte to word)字节转换为字

CWD (Convert word to double word)字转换为双字

· DIV 无符号数除法指令

格式为： DIV SRC

执行的操作：

字节操作：16 位被除数在 AX 中，8 位除数为源操作数，结果的 8 位商在 AL 中，8 位余数在 AH 中。表示为：

(AL)←(AX)/(SRC)的商

(AH)←(AX)/(SRC)的余数

字操作：32 位被除数在 DX，AX 中，其中 DX 为高位字；16 位除数为源操作数，结果的 16 位商在 AX 中，16 位余数在 DX 中。表示为：

(AX)←(DX，AX)/(SRC)的商

(DX)←(DX，AX)/(SRC)的余数

商和余数均为无符号数。

· IDIV 带符号数除法指令

格式： IDIV SRC

执行的操作：与 DIV 相同，但操作数必须是带符号数，商和余数也均为带符号数，且余数的符号和被除数的符号相同。

除法指令的寻址方式和乘法指令相同。其目的操作数必须存放在 AX 或 DX，AX 中。而其源操作数可以用除立即数以外的任一种寻址方式。

除法指令对所有条件码位均无定义。

由于除法指令的字节操作要求被除数为 16 位，字操作要求被除数为 32 位，因此往往需要用符号扩展的方法取得除法指令所需要的被除数格式，为此我们介绍两条符号扩展指令如下：

· CBW 字节转换为字指令

格式： CBW

执行的操作：AL 的内容符号扩展到 AH。即如(AL)的最高有效位为 0，则(AH)＝00；如(AL)的最高有效位为 1，则(AH)＝0FFH。

· CWD 字转换为双字指令

格式： CWD

执行的操作：AX 的内容符号扩展到 DX。即如(AX)的最高有效位为 0，则(DX)＝0000；如(AX)的最高有效位为 1，则(DX)＝0FFFFH。

这两条指令都不影响条件码。

在使用除法指令时，还需要注意一个问题，除法指令要求字节操作时商为 8 位，字操作时商为 16 位。如果字节操作时，被除数的高 8 位绝对值＞除数的绝对值；或字操作时，被除数的高 16 位绝对值＞除数的绝对值，商就会产生溢出，在 IBM PC 机中这种溢出是由系统直接转入 0 型中断处理的，有关中断处理问题将在第八章专门说明，为避免出现这种情况，必要时程序应进行溢出判断及处理。

除法运算举例如下：

例 3.37 设(AX)＝0400H，(BL)＝0B4H

即(AX)为无符号数的 1024D，带符号数的＋1024D

(BL)为无符号数的 180D，带符号数的－76D

执行 DIV BL 的结果是：

(AH)＝7CH＝124D 余数

(AL)＝05H＝5D 商

执行 IDIV BL 的结果是：

(AH)＝24H＝36D 余数

(AL)＝0F3H＝－13D 商

例 3.38 算术运算综合举例，计算

$$(V-(X*Y+Z-540))/X$$

其中 X、Y、Z、V 均为 16 位带符号数，已分别装入 X、Y、Z、V 单元中，要求上式计算结果的商存入 AX，余数存入 DX 寄存器。编制程序如图 3.12 所示

```
mov     ax,x        ;multiply x
imul    y           ;  by y and
mov     cx,ax       ;     store product
mov     bx,dx       ;        in BX,CX
mov     ax,z        ;add sign_extended z
cwd                 ;  to the
add     cx,ax       ;     product
adc     bx,dx       ;        in BX,CX
sub     cx,540      ;subtract 540
sbb     bx,0        ;  from BX,CX
mov     ax,v        ;substract (BX,CX)
cwd                 ;  from sign_extended
sub     ax,cx       ;     and
sbb     dx,bx       ;devide by x leave
idiv    x           ;  quotient in AX
                    ;     and remainder in DX
```

图 3.12 实现例 3.38 的程序段

3.3.2.5 十进制调整指令

前面提到的所有算术运算指令都是二进制数的运算指令，但是人们最常用的是十进制数，这样，当用计算机进行计算时，必须先把十进制数转换成二进制数，然后再进行二进制数的计算，计算结果又转换为十进制数输出。为了便于十进制数的计算，计算机还提供了一组十进制数调整指令，这组指令在二进制计算的基础上，给予十进制调整，可以直接得到十进制的结果。在说明这组指令之前，我们首先介绍计算机中常用的表示十进制数的BCD码。

BCD(Binary Coded Decimal)是一种用二进制编码的十进制数，又称二－十进制数。它是用4位二进制数表示一个十进制数码的，由于这4位二进制数的权为8421，所以BCD码又称8421码。十进制数码所对应的BCD码如表3.7所示。

表3.7 BCD码

十进制数码	0	1	2	3	4	5	6	7	8	9
BCD 码	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

在IBM PC机里，表示十进制数的BCD码可以用压缩的BCD码和非压缩的BCD码两种格式来表示。压缩的BCD码(packed BCD format)用4位二进制数表示一个十进制数位，整个十进制数形式为一个顺序的以4位为一组的数串。例如，9502d应表示为：

1001 0101 0000 0010。

非压缩的BCD码(unpacked BCD format)则以8位为一组表示一个十进制数位，8位中的低4位表示8421的BCD码，而高4位则没有意义。例如，9502d则表示为：

uuuu1001 uuuu0101 uuuu0000 uuuu0010。

可以看出数字的ASCII码是一种非压缩BCD码。因为数字的ASCII的高4位值为0011而低4位是以8421码表示的十进制数位，这符合非压缩BCD码高4位无意义的规定。

IBM PC机的十进制调整指令分为两组，下面分别加以说明：

(一) 压缩的BCD码调整指令

DAA (decimal adjust for addifion)加法的十进制调整指令

DAS (decimal adjust for subtraction)减法的十进制调整指令

我们知道，机器所提供的ADD、ADC以及SUB、SBB指令只适用于二进制加、减法，但压缩的BCD码却是一个字节含有两个十进制数位的二进制数。在使用加、减法指令对BCD码运算后必须经调整后才能得到正确的结果。加法的调整规则是：*任意两个用BCD码表示的十进制数位相加的结果，如数值在1010和1111之间或者产生了向高位的进位*则在其上加6就可得到正确的结果。

例如

```
    7          0111
  +6         +0110
   13         1101
            + 110
              0011
              ↙
              1
```

可见第一次得到的1101不是BCD码，根据调整规则应在其上加6，得到个位为3并向高位进位的正确结果。

又如：

低位来的进位值

```
      1                    1
      9                 1001
   +  9              + 1001
   ----              ------
     19                0011
                        ↙
                       1
                    + 110
                    ------
                      1001
```

第一次加法得到的结果有向高位的进位，根据调整规则在其上加 6，得到个位为 9，并保留进位值而得到正确的结果。

· DAA　　加法的十进制调整指令

执行的操作：

(AL)←把 AL 中的和调整到压缩的 BCD 格式，这条指令之前必须执行 ADD 或 ADC 指令，加法指令必须把两个压缩的 BCD 码相加，并把结果存放在 AL 寄存器中。本指令的调整方法是：

如果 AF 标志(辅助进位位)为 1，或者 AL 寄存器的低 4 位是十六进制的 A～F，则 AL 寄存器内容加 06H，且将 AF 位置 1；

如果 CF 标志为 1，或者 AL 寄存器的高 4 位是十六进制的 A～F，则 AL 寄存器内容加 60H，并将 CF 位置 1。

DAA 指令对 OF 标志无定义，但影响所有其它条件标志。

例 3.39

```
ADD     AL,BL
DAA
```

如指令执行前，(AL)＝28，(BL)＝68

执行 ADD 指令后 (AL)＝90，CF＝0，AF＝1

执行 DAA 指令时，因 AF＝1 而做

(AL)←(AL) + 06

得(AL)＝96，CF＝0，AF＝1 结果正确。

例 3.40　如(BCD1)＝1834，(BCD2)＝2789

要求执行　(BCD3)←(BCD1)＋(BCD2)

BCD1 和 BCD2 均为用压缩的 BCD 码表示的十进制数，由于它们都是 4 位数，所以每个数占有 2 个字节，高位数占有高位字节，其存放方式为

(BCD1)＝34，　(BCD1＋1)＝18；

(BCD2)＝89，　(BCD2＋1)＝27。

可写出指令序列如下：

```
MOV     AL,     BCD1
ADD     AL,     BCD2
DAA
MOV     BCD3,   AL
MOV     AL,     BCD1+1
ADC     AL,     BCD2+1
DAA
MOV     BCD3+1, AL
```

第一组四条指令把低位字节相加经调整后存入 BCD3，其中 ADD 指令后(AL)＝34＋89＝BDH，CF＝0，AF＝0；经 DAA 调整后，(AL)＝23，CF＝1，AF＝1。第二组四条指令把高位字节相加经调整后存入 BCD3＋1。其中 ADC 指令后(AL)＝18＋27＋CF＝40，CF＝0，AF＝1；经 DAA 调整后(AL)＝46，CF＝0，AF＝1；最后 (BCD3)＝4623 结果正确。

· DAS　减法的十进制调整指令

执行的操作：

(AL)←把 AL 中的差调整到压缩的 BCD 格式。

这条指令之前必须执行 SUB 或 SBB 指令，减法指令必须把两个 BCD 码相减，并把结果存放在 AL 寄存器中。本指令的调整方法是：

如果 AF 标志为 1，或者 AL 寄存器的低 4 位是十六进制的 A～F，则使 AL 寄存器的内容减去 06H，并将 AF 位置 1；

如果 CF 标志为 1，或者 AL 寄存器的高 4 位是十六进制的 A～F，则使 AL 寄存器的内容减去 60H，并将 CF 位置 1。

DAS 指令对 OF 标志无定义，但影响所有其它条件标志。

例 3.41
```
SUB     AL,  AH
DAS
```
如指令执行前，(AL)＝86，(AH)＝07

执行 SUB 指令后，(AL)＝7FH，CF＝0，AF＝1

执行 DAS 指令时，因 AF＝1，需做

(AL)＝(AL)－06

而得 (AL)＝79，CF＝0，AF＝1，结果正确。

例 3.42　如(BCD1)＝1234，(BCD2)＝4612，试写出指令序列完成 (BCD3)←(BCD1)－(BCD2)，数的存放方式与例 3.40 相同，指令序列如下：
```
MOV     AL,     BCD1
SUB     AL,     BCD2
DAS
MOV     BCD3,   AL
MOV     AL,     BCD1+1
SBB     AL,     BCD2+1
DAS
MOV     BCD3+1, AL
```
第一组四条指令把低位字节相减经十进制调整后存入 BCD3。其中 SUB 指令后(AL)＝22，CF＝0，AF＝0，所以 DAS 并未做什么操作而把结果送往 BCD3。第二组四条指令把高位字节相减经十进制调整后存入 BCD3＋1。其中 SBB 指令后，(AL)＝CCH，CF＝1，AF＝1 经 DAS 调整后(AL)＝66，CF＝1，AF＝1；最后(BCD3)＝6622，到这里，读者一定会说结果错了！其实，结果是对的，6622 是－3378 的十的补码。

到现在为止，我们还没有讨论用 BCD 码表示的十进制数的符号应该怎么来表示的问题。一般说来，可以有两种方法，一种是专门设一个表示符号的字段。例如下一章将讨论的 DT 伪操作用来定义一个十个字节的压缩 BCD 码，其中就规定其最高地址的字节用来表示数的符号，该字节的最高有效位为符号位，其余 7 位为 0，这就提供了用这种方法表示数的符号的手段。另一种方法就是使用十的补码来表示，如十进制数的数位为 n，则任

意整数 d 的十的补码定义为 10^n-d，数位为 n 的十进制数的表数范围为 $-5\times10^{n-1}\sim5\times10^{n-1}-1$。如 n=8，则可用二个字(32 位)来表示一个十进制数，其表数范围为 −50000000～+49999999。例 3.42 中，n=4，用 16 位表示一个带符号数，此时的表数范围是 −5000～+4999，所以 6622 应表示一个负数，它是 −3378。

(二) 非压缩的 BCD 码调整指令

AAA(ASCII adjust for addition)加法的 ASCII 调整指令

AAS(ASCII adjust for subtraction)减法的 ASCII 调整指令

AAM(ASCII adjust for multiplication)乘法的 ASCII 调整指令

AAD(ASCII adjust for division)除法的 ASCII 调整指令

这一组指令适于数字 ASCII 的调整，也适用于一般的非压缩 BCD 码的十进制调整，下面分别说明各条指令的功能。

· AAA 加法的 ASCII 调整指令

执行的操作：

(AL)←把 AL 中的和调整到非压缩的 BCD 格式

(AH)←(AH)+调整产生的进位值

这条指令之前必须执行 ADD 或 ADC 指令，加法指令必须把两个非压缩的 BCD 码相加，并把结果存放在 AL 寄存器中。本指令的调整步骤是：

(1) 如 AL 寄存器的低 4 位在 0～9 之间，且 AF 位为 0，则跳过第(2)步，执行第(3)步；

(2) 如 AL 寄存器的低 4 位在十六进制数 A～F 之间或 AF 为 1，则 AL 寄存器的内容加 6，AH 寄存器的内容加 1，并将 AF 位置 1；

(3) 清除 AL 寄存器的高 4 位；

(4) AF 位的值送 CF 位。

AAA 指令除影响 AF 和 CF 标志外，其余标志位均无定义。

例 3.43
```
ADD    AL,  BL
AAA
```

如指令执行前，(AX)=0535H，(BL)=39H，可见 AL 和 BL 寄存器的内容分别为 5 和 9 的 ASCII。第一条指令执行完后，(AL)=6EH，AF=0；第二条指令进行 ASCII 调整的结果使 (AX)=0604H，AF=1，CF=1。

· AAS 减法的 ASCII 调整指令

执行的操作：

(AL)←把 AL 中的差调整到非压缩的 BCD 格式

(AH)←(AH)−调整产生的借位值

这条指令之前必须执行 SUB 或 SBB 指令，减法指令必须把两个非压缩的 BCD 码相减，并把结果存放在 AL 寄存器中。本指令的调整步骤是：

(1) 如 AL 寄存器的低 4 位在 0～9 之间，且 AF 位为 0，则跳过第(2)步，执行第(3)步；

(2) 如 AL 寄存器的低 4 位在十六进制数 A～F 之间或 AF 位为 1，则把 AL 寄存器的内容减去 6，AH 寄存器的内容减 1，并将 AF 位置 1；

(3) 清除 AL 寄存器的高 4 位；

(4) AF 位的值送 CF 位。

AAS 指令除影响 AF 和 CF 标志外，其余标志位均无定义。

例 3.44 编写程序段实现下式

(DX)←UP1＋UP2－UP3

其中参加运算的数均为二位十进制数。如要求计算 25＋48－19，每个十进制数以非压缩 BCD 格式存入存储器，每个数占有一个字，所以(UP1)＝0205H，(UP2)＝0408H，(UP3)＝0109H。可写出指令序列如图 3.13 所示。

```
mov     ax,0            ;initialize
mov     al,up1          ;add low_order
add     al,up2          ;  digits and
aaa                     ;     adjust
mov     dl,al           ;        put sum in DL
mov     al,up1＋1       ;add high_order
adc     al,up2＋1       ;  digits with carry
aaa                     ;     and adjust
xchg    al,dl           ;exchange AL and DL
sub     al,up3          ;subtract low_order
aas                     ;  digit from sum
                        ;     and adjust
xchg    al,dl           ;exchange AL and DL
sbb     al,up3＋1       ;subtract high_order
aas                     ;  digat from sum
                        ;     and adjust
mov     dh,al           ;move AL to DH
```

图 3.13 实现例 3.44 的程序段

在指令序列的执行过程中首先把 UP1 和 UP2 的低位字节相加，经十进制调整后得(AL)＝03H，CF＝AF＝1；第二步把 UP1 和 UP2 的高位字节以及低位的进位值相加，经十进制调整后得(AL)＝07H，CF＝AF＝0；第三步把前二项和的低位字节和 UP3 的低位字节相减，经十进制调整后得(AL)＝04H，CF＝AF＝1；第四步把前二项和的高位字节减去 UP3 的高位字节及低位字节相减时的借位值，经十进制调整后得(AL)＝05H，CF＝AF＝0；最后结果在 DX 中为(DX)＝0504H，这正是十进制数 54 的非压缩 BCD 格式。

· AAM 乘法的 ASCII 调整指令

执行的操作：

(AX)←把 AL 中的和调整到非压缩的 BCD 格式

这条指令之前必须执行 MUL 指令把两个非压缩的 BCD 码相乘(此时要求其高 4 位为 0)，结果放在 AL 寄存器中。本指令的调整方法是：把 AL 寄存器的内容除以 0AH，商放在 AH 寄存器中，余数保存在 AL 寄存器中。本指令根据 AL 寄存器的内容设置条件码 SF、ZF 和 PF，但 OF、CF、和 AF 位无定义。

例 3.45 MUL AL,BL

AAM

如指令执行前，（AL）＝07H，(BL)＝09H

执行 MUL 后，（AL）＝3FH

执行 AAM 后，（AH）＝06H，(AL)＝03H

例 3.46 编写程序段实现下式

$$C \leftarrow A * B$$

其中 A、B 单元分别存放着二位用非压缩 BCD 码表示的十进制数 34 和 56，因而(A)＝04H，(A＋1)＝03H；(B)＝06H，(B＋1)＝05H。二位十进制数乘法的算法如下：

		A＋1	A
	×	B＋1	B
	C0＋2	C0＋1	C0
＋	C1＋2	C1＋2	C1
	C＋3	C＋2	C＋1	C

因而结果应存放在以 C 为首地址的 4 个相继字节单元中，在计算过程中还应开辟以 C0 和 C1 为首地址的各三个字节单元作存放中间结果用。编制的程序段及说明如下：

指令		执行的动作	(AX)
MOV	AL,A	(AL)←4	04
MUL	B	(AX)←4＊6	0018
AAM		调整	0204
MOV	WORD PTR C0, AX	(C0＋1,C0)←24	0204
MOV	AL ,A＋1	(AL)←3	0203
MUL	B	(AX)←3＊6	0012
AAM		调整	0108
ADD	AL ,C0＋1	(AL)←8＋2	010A
AAA		调整	0200
MOV	WORD PTR C0＋1,AX	(C0＋2,C0＋1)←20	0200
MOV	AL, A	(AL)←4	0204
MUL	B＋1	(AX)←4＊5	0014
AAM		调整	0200
MOV	WORD PTR C1, AX	(C1＋1,C1)←20	0200
MOV	AL,A＋1	(AL)←3	0203
MUL	B＋1	(AX)←3＊5	000F
AAM		调整	0105
ADD	AL, C1＋1	(AL)←5＋2	0107
AAA		调整	0107
MOV	WORD PTR C1＋1,AX	(C1＋2,C1＋1)←17	0107
MOV	AL, C0	(AL)←4	0104
MOV	C,AL	(C)←4	0104
MOV	AL, C0＋1	(AL)←0	0100
ADD	AL,CL	(AL)←0＋0	0100

续表

指　令		执行的动作	(AX)
AAA		调整	0100
MOV	C+1, AL	(C+1)←0	0100
MOV	AL, C0+2	(AL)←2	0102
ADC	AL, C1+1	(AL)←2+7+CF	0109
AAA		调整	0109
MOV	C+2, AL	(C+2)←9	0109
MOV	A,0	(AL)←0	0100
ADC	AL,C1+2	(AL)←0+1+CF	0101
AAA		调整	0101
MOV	C+3,AL	(C+3)←1	0101

最后得到(C)=04H,(C+1)=00H,(C+2)=09H,(C+3)=01H;即乘积为十进制的1904,结果正确。

· AAD　除法的ASCII调整指令

前面所述的加法、减法和乘法的ASCII调整指令都是用加法、减法和乘法指令对两个非压缩的BCD码运算以后,再使用AAA、AAS、AAM指令来对运算结果进行十进制调整的。除法的情况却不同,它是针对以下情况而设立的。

如果被除数是存放在AX寄存器中的二位非压缩BCD数,AH中存放十位数,AL中存放个位数,而且要求AH和AL中的高4位均为0。除数是一位非压缩的BCD数,同样要求高4位为0。在把这两个数用DIV指令相除以前,必须先用AAD指令把AX中的被除数调整成二进制数,并存放在AL寄存器中。因此,AAD指令执行的操作是:

(AL)←10*(AH)+(AL)

(AH)←0

本指令根据AL寄存器的结果设置SF、ZF和PF位,OF和CF和AF无定义。

例3.47　AAD

如指令执行前(AX)=0604H,

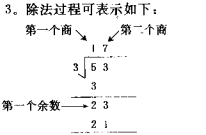
则指令执行后(AX)=0040H

例3.48　编写程序段实现

C←B/A的商

R←B/A的余数

其中B字单元中存放着用非压缩BCD码表示的二位十进制数53,A字节单元中存放着用非压缩BCD码表示的一位数3。除法过程可表示如下:



结果的商存放在字单元 C 中，余数存放在字节单元 R 中。

编制的程序及说明如下：

指　令		执行的动作	(AX)
MOV	AH,0	(AH)←0	00__
MOV	AL,B+1	(AL)←5	0005
DIV	A	除法	0201
MOV	C+1,AL	(C+1)←1	0201
MOV	AL,B	(AL)←3	0203
AAD		调整	0017
DIV	A	除法	0207
MOV	C,AL	(C)←7	0207
MOV	R, AH	(R)←2	0207

最后得到(C)＝07H,(C+1)＝01H,(R)＝02H,即商为 17,余数为 2,结果正确。

3.3.3　逻辑指令

3.3.3.1　*逻辑运算指令*

AND (and)　　逻辑与

OR (or)　　逻辑或

NOT (not)　　逻辑非

XOR (exclusive or)　　异或

TEST (test)　　测试

逻辑运算指令可以对字或字节执行逻辑运算。由于逻辑运算是按位操作的，因此一般说来，其操作数应该是位串而不是数。

· AND　逻辑与指令

格式：　AND　DST,SRC

执行的操作：(DST)←(DST) ∧ (SRC)

· OR　逻辑或指令

格式：　OR　DST, SRC

执行的操作：(DST)←(DST) ∨ (SRC)

· NOT　逻辑非指令

格式：　NOT　OPR

执行的操作：(OPR)←$\overline{(OPR)}$

· XOR　异或指令

格式：　XOR　DST,SRC

执行的操作：(DST)←(DST) ∀ (SRC)

· TEST　测试指令

格式：　TEST　OPR1,OPR2

执行的操作：(OPR1) ∧ (OPR2)

两个操作数相与的结果不保存，只根据其特征置条件码。

以上五种指令中，NOT 不允许使用立即数，其它 4 条指令除非源操作数是立即数，至少有一个操作数必须存放在寄存器中，另一个操作数则可以使用任意寻址方式。它们对标志位的影响情况是：NOT 指令不影响标志位，其它 4 种指令将使 CF 和 OF 为 0，AF 位无定义，而 SF、ZF 和 PF 则根据运算结果设置。

这些指令对处理操作数的某些位很有用，例如可屏蔽某些位（将这些位置 0），或使某些位置 1 或测试某些位等，下面举例说明：

例 3.49 要求屏蔽 0、1 两位，可用 AND 指令并设置常数 0FCH。

```
MOV     AL,0BFH
AND     AL,0FCH
```

这两条指令执行的结果使(AL)＝0BCH

```
        1011  1111
AND     1111  1100
        1011  1100
```

所以用 AND 指令可以使操作数的某些位被屏蔽。只需要把 AND 指令的源操作数设置成一个立即数，并把需要屏蔽的位设为 0，这样指令执行的结果就可把操作数的相应位置 0，其它各位保持不变。

例 3.50 要求第 5 位置 1，可用 OR 指令

```
MOV     AL,43H
OR      AL,20H
```

这两条指令执行后，(AL)＝63H

```
        0100  0011
OR      0010  0000
        0110  0011
```

所以用 OR 指令可以使操作数的某些位置 1，其它位则保持不变。只需要把 OR 指令的源操作数设置为一个立即数，并把需要置 1 的位设为 1，就可达到目的。

例 3.51 要测试操作数的某位是否为 0，可用 TEST 指令，同样把 TEST 指令的源操作数设置成一个立即数，其中需要测试的位应设为 1。

```
MOV     AL,40H
TEST    AL,0AFH
```

这两条指令执行后

```
        0100  0000
AND     1010  1111
        0000  0000
```

这里要求测试第 0、1、2、3、5、7 位是否为 0，根据测试的结果设置条件码为 CF＝OF＝0，SF＝0，ZF＝1，说明所需测试的位均为 0。如果在这两条指令之后跟一条条件转移指令 JNZ，如结果不是 0 则转移，如结果为 0 则顺序往下执行，这样就可根据测试的情况，产生不同的程序分支，转向不同的处理。

例 3.52 要测试操作数的某位是否为 1，则可先把该操作数求反，然后用 TEST 指令测试。如要测试 AL 寄存器中第 2 位是否为 1，如为 1 则转移到 EXIT 去执行，可用下列指令序列：

```
MOV     DL,AL
NOT     DL
TEST    DL,0000 0100B
JE      EXIT
```

如 AL 寄存器的内容为 0FH，为避免破坏操作数的原始内容，把它复制到 DL 去测试，执行完 TEST 指令后，因结果为全 0 而有 ZF＝1，说明操作数的第 2 位为 1 而引起转移到 EXIT 去执行。

例 3.53 要使操作数的某些位变反可以使用 XOR 指令，只要把源操作数的立即数字段的相应位置为 1 就可以达到目的。如果求第 0、1 位变反可使用如下指令：

```
MOV     AL,11H
XOR     AL,3
```

则指令执行后

```
        0001  0001
XOR     0000  0011
        ----------
        0001  0010
```

(AL)＝12H，达到第 0、1 位变反而其它位不变的目的。

例 3.54 XOR 指令还可以用来测试某一操作数是否与另一确定的操作数相等。这种操作在检查地址是否匹配时是常用的。

```
XOR     AX,042EH
JZ      MATCH
```

这两条指令用来检查 AX 的内容是否等于 042EH，若相等则转到 MATCH 去执行匹配时要作的工作，否则执行 JZ 指令下面的程序。

3.3.3.2 移位指令

SHL(shift logical left)	逻辑左移
SAL(shift arithmetic left)	算术左移
SHR(shift logical right)	逻辑右移
SAR(shift arithmetic right)	算术右移
ROL(Rotate left)	循环左移
ROR(Rotate right)	循环右移
RCL(Rotate left through carry)	带进位循环左移
RCR(Rotate right through carry)	带进位循环右移

· SHL 逻辑左移指令

格式： SHL OPR,CNT

执行的操作如图 3.14(1)所示。

其中 OPR 可以是除立即数以外的任何寻址方式。移位次数由 CNT 决定，CNT 可以是 1 或 CL。CNT 为 1 时只移一位，如需要移位的次数大于 1，则可以在该移位指令前把移位次数置于 CL 寄存器中，而移位指令中的 CNT 写为 CL 即可。有关 OPR 及 CNT 的规定适用于以下的移位指令。

· SAL 算术左移指令

格式： SAL OPR,CNT

执行的操作：与 SHL 相同。

· SHR　逻辑右移指令

格式：　　SHR　　OPR，CNT

执行的操作如 3.14(2)所示。

· SAR　　算术右移指令

格式：　　SAR　　OPR，CNT

执行的操作如图 3.14(3)所示。

这里最高有效位右移，同时再用它自身的值填入，即如原来是 0 则仍为 0，原来是 1 则仍为 1。

· ROL　　循环左移指令

格式：　　ROL　　OPR，CNT

执行的操作如图 3.14(4)所示。

· ROR　　循环右移指令

格式：　　ROR　　OPR，CNT

执行的操作如图 3.14(5)所示。

· RCL　　带进位循环左移指令

格式：　　RCL　　OPR，CNT

执行的操作如图 3.14(6)所示。

· RCR　　带进位循环右移指令

格式：　　RCR　　OPR，CNT

执行的操作如图 3.14(7)所示。

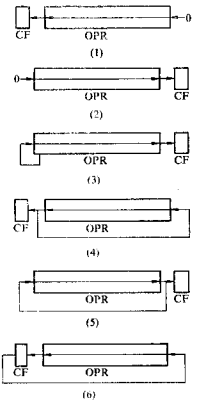


图 3.14　移位指令的操作

(1) 逻辑及算术左移　(5) 循环右移
(2) 逻辑右移　(6) 带进位循环左移
(3) 算术右移　(7) 带进位循环右移
(4) 循环左移

所有移位指令都可以作字或字节操作。它们对条件码的影响是：CF 位根据各条指令的规定设置。OF 位只有当 CNT＝1 时才是有效的，在移位后最高有效位的值发生变化时(原来为 0，移位后为 1；或原来为 1，移位后为 0)OF 位置 1，否则置 0。循环移位指令不影响除 CF 和 OF 以外的其他条件标志。而移位指令则根据移位后的结果设置 SF、ZF 和 PF 位，AF 位则无定义。可以看出，这八种指令可以分为两组：前四种为移位指令，后四种为循环移位指令。循环移位指令可以改变操作数中所有位的位置，在程序中还是很有用的。移位指令则常常用来做乘以 2 或除以 2 的操作。其中算术移位指令适用于带符号数运算，SAL 用来乘 2，SAR 用来除以 2；而逻辑移位指令则用于无符号数运算，SHL 用来乘 2，SHR 用来除以 2。

例 3.55　MOV　　CL，5

　　SAR　　[DI]，CL

如指令执行前：(DS)＝0F800H (DI)＝180AH，(0F980A)＝0064H

则指令执行后：(0F980A)＝0003H，CF＝0

相当于 100d/32d＝3d

例 3.56　MOV　　CL，2

　　SHL　　SI，CL

如指令执行前：(SI)＝1450H

则指令执行后：(SI)＝5140H，CF＝0

相当于 5200d×4d＝20800d

例 3.57 如(AX)＝0012H，(BX)＝0034H 要求把它们装配在一起形成(AX)＝1234H，可编制程序如下：

```
MOV    CL,8
ROL    AX,CL
ADD    AX,BX
```

3.3.4 串处理指令

MOVS (Move string)　串传送

CMPS (Compare string)　串比较

SCAS (Scan string)　串扫描

LODS (Load from string)　从串取

STOS (Store in to string)　存入串

与上述基本指令配合使用的前级有：

REP (Repeat)　重复

REPE/REPZ(Repeat while equal/zero)　相等/为零则重复

REPNE/REPNZ(Repeat while not equal/not zero)　不相等/不为零则重复

串处理指令处理存放在存储器里的数据串，所有串指令都可以处理字节或字。下面，我们分为两组来说明：

3.3.4.1　与 REP 相配合工作的 MOVS，STOS 和 LODS 指令

· REP 重复串操作直到(CX)＝0 为止。

格式：　　REP　string primitive

其中 String Primitive 可为 MOVS，LODS 或 STOS 指令。

执行的操作：

1) 如(CX)＝0 则退出 REP，否则往下执行。

2) (CX)←(CX)－1

3) 执行其后的串指令

4) 重复 1)～3)

· MOVS　串传送指令

格式：可有三种

MOVS　DST,SRC

MOVSB（字节）

MOVSW（字）

其中第二、三种格式明确地注明是传送字节或字，第一种格式则应在操作数中表明是字还是字节操作，例如：

```
MOVS    ES：BYTE PTR[DI],DS：[SI]
```

实际上 MOVS 的寻址方式是隐含的(这在下面所执行的操作中可以看到)，所以这种格式中的 DST 及 SRC 只提供给汇编程序作类型检查用，并且不允许用其他寻址方式来确定操

作数。

执行的操作：

1) ((DI))←((SI))

2) 字节操作：

(SI)←(SI)±1,(DI)←(DI)±1

当方向标志 DF=0 时用+，当方向标志 DF=1 时用−。

3) 字操作：

(SI)←(SI)±2,(DI)←(DI)±2

当方向标志 DF=0 时用+，当方向标志 DF=1 时用−。

该指令不影响条件码。

MOVS 指令可以把由(SI)指向的数据段中的一个字(或字节)传送到由(DI)指向的附加段中的一个字(或字节)中去，同时根据方向标志及数据格式(字或字节)对 SI 和 DI 进行修改。当该指令与前缀 REP 联用时，则可将数据段中的整串数据传送到附加段中去。这里源串必须在数据段中，目的串必须在附加段中，但源串允许使用段跨越前缀来修改。在与 REP 联用时还必须先把数据串的长度送到 CX 寄存器中，以便控制指令结束。因此在执行该指令前，应该先做好以下准备工作：

1) 把存放下数据段中的源串首地址(如反向传送则应是末地址)放入 SI 寄存器中；

2) 把将要存放数据串的附加段中的目的串首地址(或反向传送时的末地址)放入 DI 寄存器中；

3) 把数据串长度放入 CX 寄存器；

4) 建立方向标志。

在完成这些准备工作后就可使用串指令传送信息了。

为了建立方向标志，这里介绍两条指令。

· CLD (Clear direction flag)该指令使 DF=0，在执行串处理指令时可使地址自动增量；

· STD (Set direction flag)该指令使 DF=1，在执行串处理指令时可使地址自动减量。

下面举例说明这一组串指令的使用方法：

例 3.58 在数据段中有一个字符串，其长度为 17，要求把它们传送到附加段中的一个缓冲区中。编制程序如图 3.15 所示。其中 SEGMENT，ENDS 为定义段的伪操作，DB 为定义字节数据的伪操作，这在第四章中将专门说明。在这里，读者只要知道 MESS1 为源串，存放在数据段中从符号地址 MESS1 开始的存储区内，每个字符占有一个字节；MESS2 为目的串，存放在附加段中从符号地址 MESS2 开始空出 17 个字节的存储区内。后面的程序则存放在代码段中。程序运行过程可以用图 3.16 来说明。

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
datarea segment                          ;define data segment
  mess1        db          'personal computer $'
datarea ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
extra   segment                          ;defin extra segment
  mess2        db          17 dup (?)
```

```
extra   ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
code    segment                          ;define code segment
        assume cs:code,ds:datarea,es:extra
                ⋮
;set DS register to current data segment
        mov     ax,datarea               ;datarea segment addr
        mov     ds,ax                    ;  into DS register
;set ES register to current extra segment
        mov     ax,extra                 ;extra segment addr
        mov     es,ax                    ;  into ES register
                ⋮
        lea     si,mess1
        lea     di,mess2
        mov     cx,17
        cld
        rep     movsb
                ⋮
code    ends                             ;end of code segment
```

图 3.15　实现例 3.58 的程序段

通过 MOVS 指令可总结一下串处理指令的特性如表 3.8 所示。这些特性适用于所有的串处理指令。

表 3.8　串处理指令的特性

数据类型	字	字节
方向(DF)	向前(DF=0)	向后(DF=1)
串长度	1	重复 REP(CX)次
源串(SI)	在数据段中(可用段跨越前缀修改)	
目的串(DI)	在附加段中	

· STOS　存入串指令

格式：　STOS　DST

STOSB（字节）

STOSW（字）

执行的操作：

字节操作：((DI))←(AL)，(DI)←(DI)±1

字 操 作：((DI))←(AX)，(DI)←(DI)±2

该指令把 AL 或 AX 的内容存入由(DI)指定的附加段的某单元中，并根据 DF 的值及数据类型修改 DI 的内容。当它与 REP 联用时，可把 AL 或 AX 的内容存入一个长度为(CX)的缓冲区中。上述有关串处理指令的特性也适用于 STOS 指令，该指令也不影响标志位。

STOS 指令在初始化某一缓冲区时很有用。

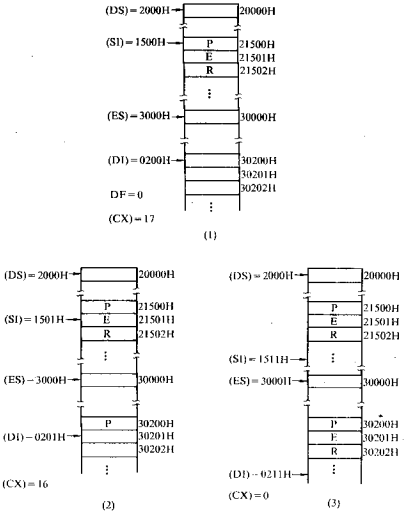


图 3.16 REP MOVSB 指令的执行情况

(1) 预置情况 (2) 执行第一条 MOVSB 后 (3) 执行完 REP MOVSB 后

例 3.58 如果要把附加段中的五个字节缓冲区置为 20H 值，则可根据串处理指令的要求预置指针及方向标志，并用 REP STOSB 指令执行结果如图 3.17 所示。

- LODS 从串取指令

格式： LODS SRC

LODSB (字节)

LODSW (字)

执行的操作：

字节操作：(AL)←((SI)),(SI)←(SI)±1

字 操 作：(AX)←((SI)),(SI)←(SI)±2

该指令把由(SI)指定的数据段中某单元的内容送到 AL 或 AX 中，并根据方向标志及数据类型修改 SI 的内容。指令允许使用段跨越前缀来指定非数据段的存储区。该指令也不影响条件码。

一般说来，该指令不和 REP 联用。有时缓冲区中的一串字符需要逐次取出来测试时，可使用本指令。

使用 LODSW 指令的预置情况及执行结果如图 3.18 所示。

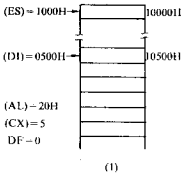


图 3.17 REP STOSB 指令的执行情况
(1) 预置情况 (2) 执行 REP STOSB 指令后

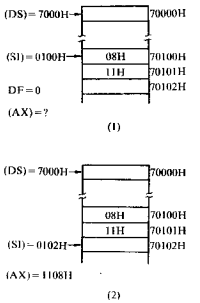


图 3.18 LODSW 指令的执行情况
(1) 预置情况 (2) 执行 LODSW 指令后

3.3.4.2 与 REPE/REPZ 和 REPNE/REPNZ 联合工作的 CMPS 和 SCAS 指令

· REPE/REPZ 当相等/为零时重复串操作

格式： REPE(或 REPZ) String Primitive

其中 String Primitive 可为 CMPS 或 SCAS 指令。

执行的操作：

1) 如(CX)=0 或 ZF=0(即某次比较的结果两个操作数不等)时退出，否则往下执行。

2) (CX)←(CX)−1

3) 执行其后的串指令

4) 重复 1)～3)

实际上 REPE 和 REPZ 是完全相同的，只是表达的方式不同而已。与 REP 相比，除满足(CX)=

0 的条件可结束操作外，还增加了 ZF=0 的条件。也就是说，只要两数相等就可继续比较，如果遇到两数不相等时可提前结束操作。

· REPNE/REPNZ　　当不相等/不为零时重复串操作

格式：　　REPNE(或 REPNZ)　　String Primitive

其中 String Primitive 可为 CMPS 和 SCAS 指令。

执行的操作：

除退出条件为(CX)=0 或 ZF=1 外，其他操作与 REPE 完全相同。也就是说，只要两数比较不相等，就可继续执行串处理指令，如某次两数比较相等或(CX)=0 时，就可结束操作。

· CMPS　　串比较指令

格式：　　CMPS　　SRC,DST

CMPSB (字节)

CMPSW (字)

执行的操作：

1) ((SI))--((DI))

2) 字节操作：(SI)←(SI)±1,(DI)←(DI)±1

字 操 作：(SI)←(SI)±2,(DI)←(DI)±2

指令把由(SI)指向的数据段中的一个字(或字节)与由(DI)指向的附加段中的一个字(或字节)相减，但不保存结果，只根据结果置条件码，指令的其他特性和 MOVS 指令的规定相同。

· SCAS　　串扫描指令

格式：　　SCAS　　DST

SCASB (字节)

SCASW (字)

执行的操作：

字节操作：(AL)—((DI)),(DI)←(DI)±1

字 操 作：(AX)—((DI)),(DI)←(DI)±2

指令把 AL(或 AX)的内容与由(DI)指定的在附加段中的一个字节(或字)进行比较，并不保存结果，只根据结果置条件码。指令的其他特性和 MOVS 的规定相同。

以上两条串处理指令和 REPE/REPZ 或 REPNE/REPNZ 相结合可以用来比较两个数据串，或从一个字符串中查找一个指定的字符。

例 3.60　要求从一个字符串中查找一个指定的字符，可用指令 REPNZ　　SCASB。下面用图 3.19 来表示预置及找到后的情况。由图可见：(AL)中指定的字符为 space(空格)，其 ASCII 码为 20H。开始比较时因(DI)指定的字符与(AL)不符合而不断往下比较，当(DI)=1508H 时比较结果相符，因此 ZF=1，在修改(DI)值后指令停止比较而提前结束，此时

(DI)是相匹配字符的下一个地址；

(CX)是剩下还未比较的字符个数。

所以根据(DI)和(CX)的值可以很方便地找到所需查找字符的位置。

例 3.61　要求比较两个字符串，找出它们不相匹配的位置。这里可使用指令 REPE CMPSB。下面用图 3.20 表示其预置及找到后的情况。

最后，对于串处理指令，我们再说明几个需要注意的问题：

1) 串处理指令在不同的段之间传送或比较数据，如果需要在同一段内处理数据，可以在

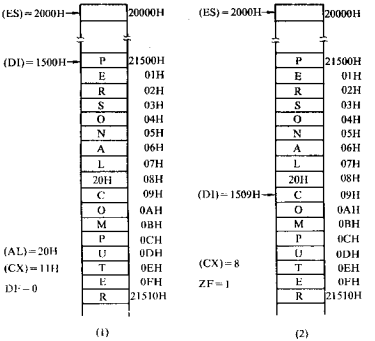


图 3.19 REPNZ SCASB 指令的执行情况

(1) 预置 (2) 执行完 REPNZ SCASB 指令后

DS 和 ES 中设置同样的地址，或者在源操作数字段使用段跨越前缀来实现，例如

```
MOVS    [DI],ES:[SI]。
```

2) 当使用重复前缀时(CX)是每次减 1 的，因此对于字指令来说，预置时设置的值应该是字的个数而不是字节数。

3) 上面所举的例子都把 DF 设置为 0，作正向传送或比较。实际上反向传送也是很有用的，有些情况下必须反向处理，例如需要把数据缓冲区向前(地址增加的方向)错一个字，此时为避免信息的丢失，不可能用正向传送而必须使用反向传送(DF=1)的方法，见图 3.21，注意这里的(DS)和(ES)应设置为同一个值。

3.3.5 控制转移指令

一般情况下指令是顺序地逐条执行的，但实际上程序不可能全部顺序执行而经常需要改变程序的执行流程，这里要介绍的控制转移指令就是用来控制程序的执行流程的。

3.3.5.1 无条件转移指令

· *JMP (jmp)* 跳转指令

无条件地转移到指令指定的地址去执行从该地址开始的指令。可以看出 JMP 指令必须指定转移的目标地址(或称转向地址)。

总的说来，转移可以分成两类，段内转移和段间转移。段内转移是指在同一段的范围之内进行转移，此时只需改变 IP 寄存器的内容，即用新的转移目标地址代替原有的 IP 的值就可达到转移的目的。段间转移则是要转到另一个段去执行程序，此时不仅要修改 IP 寄存器的内容，

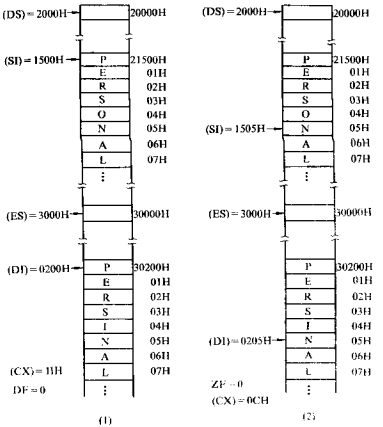


图 3.20 REPE CMPSB 指令的执行情况

(1) 预置 (2) 执行完 REPE CMPSB 指令后

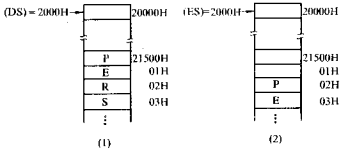


图 3.21 反向传送举例

(1) REP MOVSB 指令执行前 (2) REP MOVSB 指令执行后

还需要修改 CS 寄存器的内容才能达到目的，因此，此时的转移目标地址应由新的段地址和偏移地址两部分组成。有关转移的寻址方式，已于 3.1.2 节作了介绍，这里只简单给出 JMP 指令

的各种格式及其执行的操作：

1）段内直接短转移：

格式：JMP　　SHORT OPR

执行的操作：(IP)←(IP)+8 位位移量

其中 8 位位移量是由目标地址 OPR 确定的。转移的目标地址在汇编格式中可直接使用符号地址，而在机器执行时则是当前的 IP 值(即 JMP 指令的下一条指令的地址)与指令中指定的 8 位位移量之和。位移量需要满足向前或向后转移的需要，因此它是一个带符号数，也就是说这种转移格式只允许在－128 到＋127 字节的范围内转移。

例如，代码段内有一条无条件转移指令如下：

```
        ⋮
        JMP     SHORT HELLO
        ⋮
HELLO:  MOV     AL,3
        ⋮
```

图 3.22 表示了该转移指令的机器语言，以及用位移量来表示转向地址的方法。由图可见位移量 D8＝08H，当前的 IP 值为 0102H，所以转向偏移地址＝0102＋08＝010AH，其符号地址为 HELLO。

2）段内直接近转移：

格式：　　JMP　　NEAR PTR OPR

执行的操作：(IP)←(IP)+16 位位移量

可以看出除位移量为 16 位外，它和段内短转移一样，也采用相对寻址方式，在汇编格式中 OPR 也只需要使用符号地址。由于位移量为 16 位，它可以转移到段内的任一个位置。

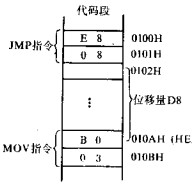


图 3.22　段内短转移举例

3）段内间接转移：

格式：　　JMP　　WORD PTR OPR

执行的操作：(IP)←(EA)

其中有效地址 EA 值由 OPR 的寻址方式确定。它可以使用除立即数方式以外的任一种寻址方式，如果指定的是 16 位寄存器则把寄存器的内容送到 IP 寄存器中；如果指定的是存储器中的一个字，则把该存储单元的内容送到 IP 寄存器中去。在 3.1.2 节中我们已经说明，这里不再赘述。

4）段间直接(远)转移：

格式：　　JMP　　FAR PTR OPR

执行的操作：(IP)←OPR 的段内偏移地址

(CS)←OPR 所在段的段地址

在这里使用的是直接寻址方式。在汇编格式中 OPR 可使用符号地址，而机器语言中则要指定转向地址的偏移地址和段地址。

如程序如下所示，图 3.23 则为 JMP 指令在存储器里的机器语言表示。

```
C1 SEGMENT
        ⋮
        JMP     FAR PTR NEXT_PROG
```

```
C1 ENDS
C2 SEGMENT
        ⋮
NEXT_PROG :
        ⋮
C2 ENDS
```

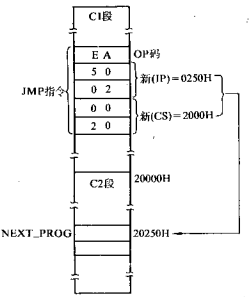


图 3.23　段间直接转移举例

5) 段间间接转移：

格式：　　JMP　　DWORD PTR OPR

执行的操作：(IP)←(EA)

(CS)←(EA+2)

其中 EA 由 OPR 的寻址方式确定，它可以使用除立即数及寄存器方式以外的任何存储器寻址方式，根据寻址方式求出 EA 后，把指定存储单元的字内容送到 IP 寄存器，并把下一个字的内容送到 CS 寄存器，这样就实现了段间跳转。例如，

JMP　　DWORD PTR ALPHA [SP][DI]等。

最后说明一下，JMP 指令不影响条件码。

3.3.5.2　条件转移指令

条件转移指令根据上一条指令所设置的条件码来判别测试条件，每一种条件转移指令有它的测试条件，满足测试条件则转移到由指令指出的转向地址去执行那里的程序；如不满足条件则顺序执行下一条指令。因此，当满足条件时：

(IP)←(IP)+符号扩展到 16 位后的位移量 D8，如不满足测试条件则(IP)不变。可见条件转移指令使用了相对寻址方式，在汇编格式中 OPR 应指定一个目标地址，这个目标地址应在本条转移指令下一条指令地址的－128～＋127 个字节的范围之内。另外，所有的条件转移指令都不影响条件码。下面我们把条件转移指令分为四组来讨论。

1) 根据单个条件标志的设置情况转移。这组包括 10 种指令。它们一般适用于测试某一次运算的结果并根据其不同特征产生程序分支作不同处理的情况。

· JZ(或 JE)(Jump if zero,or equal)　　结果为零(或相等)则转移。

格式：　　JZ(或 JE)　　OPR

测试条件：ZF=1

· JNZ(或 JNE)(Jump if not zero,or not equal)　　结果不为零(或不相等)则转移。

格式：　　JNZ(或 JNE)　　OPR

测试条件：ZF=0

· JS(Jump if sign)　　结果为负则转移

格式：　　JS　　OPR

测试条件：SF=1

· JNS (Jump if not sign)　　结果为正则转移

格式：　JNS　OPR

测试条件：SF=0

· JO (Jump if overflow)　溢出则转移

格式：　JO　OPR

测试条件：OF=1

· JNO (Jump if not overflow)　不溢出则转移

格式：　JNO　OPR

测试条件：OF=0

· JP(或 JPE)(Jump if parity, or parity even)　奇偶位为 1 则转移。

格式：　JP(或 JPE)　OPR

测试条件：PF=1

· JNP (或 JPO)(Jump if not parity, or parity odd)　奇偶位为 0 则转移。

格式：　JNP (或 JPO)　OPR

测试条件：PF=0

· JB(或 JNAE,JC)(Jump if below,or not above or equal,or carry)　低于，或者不高于或等于，或进位为 1 则转移。

格式：　JB(或 JNAE,或 JC)　OPR

测试条件：CF=1

· JNB(或 JAE,JNC)(Jump if not below,or above or equal,or not carry)　不低于，或者高于或等于，或进位为零则转移。

格式：　JNB(或 JAE,或 JNC)　OPR

测试条件：CF=0

最后两种指令在这一组指令中可以只看作 JC 和 JNC，它们只用 CF 的值来判别是否转移。

例 3.62　如果需要根据一次加法运算的结果实行不同的处理，如结果为 0 做动作 2，否则做动作 1，程序框图如图 3.24 所示，编制程序如下：

```
        ⋮
    ADD    AX,TEMP
    JZ     ACTION_2
      ⋮                 } ACTION_1
ACTION_2 :
      ⋮                 } ACTION_2
```

或者：

```
        ⋮
    ADDAX,TEMP
    JNZ ACTION_1
      ⋮                 } ACTION_2
ACTION_1 :
      ⋮                 } ACTION_1
```

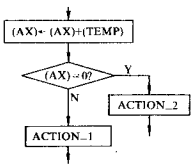


图 3.24　例 3.62 的程序框图

例 3.63　比较两个数是否相等，如相等做动作 1，否则做动作 2。可编写程序为：

```
        ⋮
    CMP   AX,BX
```

```
        JE    ACTION_1
              ⋮                }ACTION_2
ACTION_1 :                     }ACTION_1
              ⋮
```

或者：

```
              ⋮
        CMP   AX,BX
        JNZ   ACTION_2
              ⋮                }ACTION_1
ACTION_2 :                     }ACTION_2
              ⋮
```

2）比较两个无符号数，并根据比较的结果转移。

• JB(或 JNAE,JC)　　低于，或者不高于或等于，或进位位为 1 则转移。

• JNB (或 JAE,JNC)　　不低于，或者高于或等于，或进位位为 0 则转移。

以上两种指令与上一组指令中的最后两种完全相同。

• JBE (或 JNA)(Jump if below or equal,or not above)　　低于或等于，或不高于则转移。

格式：　JBE (或 JNA)　　OPR

测试条件：CF ∨ ZF＝1

• JNBE(或 JA)(Jump if not below or equal,or above)　　不低于或等于，或者高于则转移。

格式：　JNBE(或 JA)　　OPR

测试条件：CF ∨ ZF＝0

3）比较两个带符号数，并根据比较结果转移。

• JL(或 JNGE)(Jump if less,or not greater or equal)　　小于，或者不大于或者等于则转移。

格式：　JL(或 JNGE)　　OPR

测试条件：SF ∀ OF＝1

• JNL (或 JGE)(Jump if not less,or greater or equal)　　不小于，或者大于或等于则转移。

格式：　JNL(或 JGE)　　OPR

测试条件：SF ∀ OF＝0

• JLE(或 JNG)(Jump if less or equal,or not greater)　　小于或等于，或者不大于则转移。

格式：　JLE (或 JNG)　　OPR

测试条件：(SF ∀ OF) ∨ ZF＝1

• JNLE(或 JG)(Jump if not less or equal,or greater)　　不小于或等于，或者大于则转移。

格式：　JNLE(或 JG)　　OPR

测试条件：(SF ∀ OF) ∨ ZF＝0

这两组条件转移指令用于对两个数进行比较，并根据比较结果的＜、≥、≤、＞几种不同情况来判断是否转移。其中前一组 JB、JAE、JBE、JA 四种指令适用于判断无符号数的比较情况，如用于地址比较或双精度数的低位字的比较等；而后一组 JL、JGE、JLE、JG 四种指令则适用于判断带符号数的比较情况。在使用时必须严格地加以区别，否则会引起错误的结果。例如：11111111 和 00000000 两个数相比较，如果把它们看作无符号数则前者大于后者(255＞0)，但

是如果把它们看作带符号数则前者小于后者(−1<0)。因而要根据数的不同类型来选择条件转移指令的道理是很明显的。

这两组指令的测试条件是完全不同的,它们为什么要有这种差别呢?让我们分析一下它们的测试条件。

无符号数比较的转移指令的测试条件是易于理解的。前面已经分析过,当两个无符号数相减时,CF 位的情况说明了是否有借位的问题。有借位时 CF=1,当然它对应于<的条件;CF=0 时没有借位,就必然对应于≥的条件了,这就是 JB 和 JAE 指令的测试条件的建立原因。既然如此,要求≤的 JBE 指令的测试条件为 CF∨ZF=1 以及要求>的 JA 指令的测试条件为 CF∨ZF=0 也是显而易见的了。

带符号数比较的转移指令的测试条件比较复杂,我们先分析要求<的 JL 指令的情况,它的测试条件为 SF ∀ OF=1。

16 位带符号数可以表示数的范围为−32768～+32767,相应的 16 进制表示为 8000～7FFF,可以用图 3.25 表示。由图可见:如正数溢出则进入负数区,而负数溢出则进入正数区。在溢出的情况下,运算结果都是错误的。

异或运算的真值表如表 3.9 所示。下面我们根据真值表表示的四种情况来分析该测试条件的含义。

表 3.9 SF ∀ OF 的真值表

序号	SF	OF	SF ∀ OF
a	0	0	0
b	0	1	1
c	1	0	1
d	1	1	0

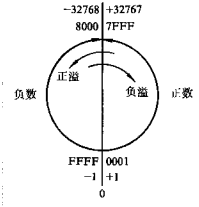


图 3.25 16 位二进制补码表示数的范围

例如:

```
MOV   AX,A
CMP   AX,B
JL    X
```

如果(A)=α,(B)=β,如 α<β 则转移到 X 去执行。

a) α−β 的结果使 SF=0, OF=0。这说明差值为正,且未溢出,结果在右半区,可以判断 α≥β 不满足转移条件 SF ∀ OF=0。

b) α−β 的结果使 SF=0,OF=1。说明差值为正,但有溢出,这种情况必为负溢。现以 4 位二进制数为例,说明如下:

如 α=1100, β=0101

α−β=1100−0101=0111(溢出,结果错)

SF=0, OF=1

可见此时 α<β,应满足测试条件:SF ∀ OF=1。

c) α−β 的结果使 SF=1,OF=0。差值为负,且未溢出说明 α<β 应满足测试条件 SF ∀ OF=1。

d) α−β的结果使SF=1,OF=1,差值为负,但又溢出,说明是正溢。仍以4位二进制数为例说明如下:

如　　α=0101,β=1100

　　α−β=0101−1100=1001(溢出,结果错)

　　SF=1,OF=1

此时α>β不满足测试条件:SF ∀ OF=0。

从上面的分析可以看出,这里不是简单地用差值的符号来判断两个数的大小,而是更全面地考虑到了可能存在的差值溢出的情况,虽然差值溢出了,但我们只关心数的比较而并不关心差值的具体情况,所以使用了SF ∀ OF=1的测试条件就更全面地反映了两数的比较情况。

既然JL(<)的测试条件为SF ∀ OF=1,则JGE(≥)的测试条件为SF ∀ OF=0就是显而易见的了,而且JLE(≤)的测试条件为(SF ∀ OF) ∨ ZF=1以及JG(>)的测试条件为(SF ∀ OF) ∨ ZF=0也都是显而易见的了。

4) 测试CX的值为0则转移指令

· JCXZ(Jump if CX register is zero)　　CX寄存器的内容为零则转移。

格式:　JCXZ　OPR

测试条件:(CX)=0

CX寄存器经常用来设置计数值,所以这条指令可以根据CX寄存器内容的修改情况来产生两个不同程序分支。

下面举例说明转移指令的使用方法。

例3.64　设X、Y均为存放在X和Y单元中的16位操作数,先判X>50否,如满足条件则转移到TOO_HIGH去执行,然后做X−Y,如溢出则转移到OVERFLOW去执行,否则计算|X-Y|,并把结果存入RESULT中。程序如下:

```
            MOV     AX,X
            CMP     AX,50
            JG      TOO_HIGH
            SUB     AX,Y
            JO      OVERFLOW
            JNS     NONNEG
            NEG     AX
NONNEG:     MOV     RESULT,AX
TOO_HIGH:
            ⋮
OVERFLOW:
            ⋮
```

例3.65　α,β为两个双精度数,分别存储于DX、AX及BX、CX中。要求编制一程序使α>β时转向X执行,否则转向Y执行。程序如下:

```
    CMP     DX,BX
    JG      X
    JL      Y
    CMP     AX,CX
    JA      X
Y:  ⋮
```

X：　　⋮

例 3.66　在存储器中有一个首地址为 ARRAY 的 N 字数组，要求测试其中正数、0 及负数的个数。正数的个数放在 DI 中，0 的个数放在 SI 中，并根据 N－(DI)－(SI)求得负数的个数放在 AX 中，如果有负数则转移到 NEG_VAL 中去执行。程序如图 3.26 所示。

```
        mov     cx,N            ;load array size to CX
        mov     bx,0            ;clear BX
        mov     di,bx           ;use DI to count elements>0
        mov     si,bx           ;use SI to count elements=0
again:  cmp     array[bx],0     ;branch to less or eq
        jle     less_or_eq      ;if element less or
                                ;       equal 0
        inc     di              ;otherwise,increment DI
        jmp     short next      ;and branch to next
less_or_eq:
        jl      next            ;if element < 0,go to next
        inc     si              ;otherwise,increment SI
next:   add     bx,2            ;increment offset
        dec     cx              ;decrement loop counter
        jnz     again           ;and repeat if nonzero
        mov     ax,N            ;subtract DI and SI
        sub     ax,di           ;  from N to get no. of
        sub     ax,si           ;     elements < 0
        jz      skip            ;are there negative value?
        jmp     near ptr neg_val;if so,branch
                                ;              to neg_val
skip:
        ⋮
neg_val:
        ⋮
```

图 3.26　实现例 3.66 的程序段

3.3.5.3　循环指令

例 3.66 的测试数组中正数、0 及负数个数的程序实际上已经是一个循环程序了。它每次测试数组中的一个数，测第二个数时仍然使用这组指令，只是对它的地址作了修改，每循环一次测试一个数。为了控制循环的结束，在进入循环前先建立循环次数，以后每循环一次就修改计数值，当计数值为 0 时就结束循环，这一过程可用框图表示如图 3.27。

例 3.66　循环结构的实现方法是

```
    MOV     CX,N
    ⋮
AGAIN:
    ⋮                   } 循环体
```

```
        DEC     CX
        JNZ     AGAIN
```

IBM-PC 机为了简化循环程序的设计(第五章还会专门说明),设计了一组循环指令如下:

LOOP(LOOP)　　循环

LOOPZ/LOOPE(Loop while zero,or equal)　　当为零或相等时循环

LOOPNZ/LOOPNE(Loop while nonzero,or not equal)　　当不为零或不相等时循环

• LOOP 循环指令

格式:　LOOP　OPR

测试条件:(CX)≠0

• LOOPZ/LOOPE 当为零或相等时循环指令

格式:LOOPZ(或 LOOPE)　OPR

测试条件:ZF=1 且(CX)≠0

• LOOPNZ/LOOPNE　当不为零或不相等时循环指令

格式:　LOOPNZ(或 LOOPNE)　OPR

测试条件:ZF=0 且(CX)≠0

这三条指令的执行步骤是:

1) (CX)←(CX)−1

2) 检查是否满足测试条件,如满足则(IP)←(IP)+D8 的符号扩展。

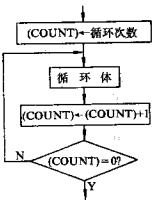


图 3.27　循环程序框图

可见这里使用的是相对方式,在汇编格式中 OPR 必须指定一个表示转向地址的标号(符号地址),而在机器指令里则用 8 位位移量 D8 来表示转向地址与当前 IP 值的差。由于位移量只有 8 位,所以转向地址必须在该循环指令的下一条指令地址的−128～+127 字节的范围之内。当满足测试条件时就转向由 OPR 指定的转向地址去执行,即实行循环,如不满足测试条件则 IP 值不变,即退出循环,程序继续顺序执行。

循环指令不影响条件码。

有了循环指令后,例 3.66 的程序结构可以这样实现:

```
        MOV     CX,N
                :
AGAIN:
                :
        LOOP    AGAIN
```

即用一条循环指令代替了原有的修改循环计数及判断转移条件两条指令。下面我们举几个例子说明循环指令的用法。

例 3.67　有一个首地址为 ARRAY 的 M 字数组,试编写一个程序:求出该数组的内容之和(不考虑溢出),并把结果存入 TOTAL 中。

```
        MOV     CX,M        ;put count in CX
        MOV     AX,0        ;Zero AX
        MOV     SI,AX       ; and SI
START_LOOP:
```

```
        ADD     AX,ARRAY[SI]    ;Add next element to AX
        ADD     SI,2            ; Increment index by two
        LOOP    START_LOOP      ;Repeat loop if CX nonzero
        MOV     TOTAL,AX        ;Store result in TOTAL
```

除 LOOP 指令外另外两条 LOOPZ 和 LOOPNZ 指令提供了提前结束循环的可能性。有时需要在字符串中查找一个字符，在找到后就可提前结束循环而不需要一直查找到底了。此时就可用 LOOPZ 或 LOOPNZ 来处理。

例 3.68 有一串 L 个字符的字符串存储于首地址为 ASCII_STR 的存储区中。如要求在字符串中查找"空格"(ASCII 码为 20H)字符，找到则继续执行，如未找到则转到 NOT_FOUND 去执行，编制实现这一要求的程序如下：

```
        MOV     CX,L            ; Put array size in CX
        MOV     SI,-1           ; initialize index,and
        MOV     AL,20H          ; put code for space in AL
NEXT:   INC     SI              ; Increment index
        CMP     AL,ASCII_STR[SI]  ;Test for space
        LOOPNE  NEXT            ; Loop if not space and count is nonzero
        JNZ     NOT_FOUND       ; if space not found
                                ; branch to NOT_FOUND
        :
NOT_FOUND:
        :
```

在程序执行过程中，有两种可能性：

1) 在查找中找到了"Space"，此时 ZF=1 因此提前结束循环。在执行 JNZ 指令时，因不满足测试条件而顺序地继续执行。

2) 如一直查找到字符串结束还未找到"space"字符，此时因(CX)=0 而结束循环，但在执行 JNZ 指令时因 ZF=0 而转移到 NOT_FOUND 去执行。

在程序继续执行的部分还可以打印出找到"Space"，并打印出(SI)以便确定它在字符串中的位置。在 NOT_FOUND 后可以打印出未找到"Space"等信息。

这个例子和例 3.60 的要求是完全相同的，只是所选择的指令和程序结构不同，同样都可完成所要求的任务。读者可比较选用。

3.3.5.4 子程序(Subroutine)

子程序结构相当于高级语言中的过程(procedure)。为便于模块化程序设计，往往把程序中某些具有独立功能的部分编写成独立的程序模块，称之为子程序。程序中可由调用程序(或称主程序)调用这些子程序，而在子程序执行完后又返回调用程序继续执行。为实现这一功能，IBM-PC 机提供了以下指令：

CALL (call)　调用

RET (return)　返回

由于子程序与调用程序可以在一个段中，也可以不在同一段中，因此这两条指令的格式如下：

· CALL 调用指令

1) 段内直接调用

格式：　　CALL　　DST

执行的操作：(SP)←(SP)－2

((SP)＋1，(SP))←(IP)

(IP)←(IP)＋D_{16}

可以看出，这条指令的第一步操作是把子程序的返回地址(即调用程序中 CALL 指令的下一条指令的地址)存入堆栈中，以便子程序返回主程序时使用。第二步操作则是转移到子程序的入口地址去继续执行。指令中 DST 给出转向地址(即子程序的入口地址，亦即子程序的第一条指令的地址)，D16 即为机器指令中的位移量，它是转向地址和返回地址之间的差值。

2) 段内间接调用

格式：　　CALL　　DST

执行的操作：(SP)←(SP)－2

((SP)＋1，(SP))←(IP)

(IP)←(EA)

其中 EA 是由 DST 的寻址方式所确定的有效地址。

3) 段间直接调用

格式：　　CALL　　DST

执行的操作：(SP)←(SP)－2

((SP)＋1，(SP))←(CS)

(SP)←(SP)－2

((SP)＋1，(SP))←(IP)

(IP)←偏移地址(指令的第 2、3 个字节)

(CS)←段地址(指令的第 4，5 个字节)

它同样是先保留返回地址，然后转移到由 DST 指定的转向地址去执行。由于调用程序和子程序不在同一个段内，因此返回地址的保存以及转向地址的设置都必须把段地址考虑在内。

4) 段间间接调用

格式：　　CALL　　DST

执行的操作：(SP)←(SP)－2

((SP)＋1，(SP))←(CS)

(SP)←(SP)－2

((SP)＋1，(SP))←(IP)

(IP)←(EA)

(CS)←(EA＋2)

其中 EA 是由 DST 的寻址方式确定的有效地址。

在上述 CALL 指令的格式中，并未加上如 NEAR PRT 格式的属性操作符，在实际使用时，读者可根据具体情况加上它。使用方法可参考 JMP 指令的说明。

· RET　返回指令

RET 指令放在子程序的末尾，它使子程序在功能完成后返回调用程序继续执行，而返回地址是调用程序调用子程序(简称转子)时存放在堆栈中的，因此 RET 指令的操作是返回地址出栈送 IP 寄存器(段内或段间)和 CS 寄存器(段间)。

1) 段内返回

格式：　　RET

执行的操作：(IP)←((SP)+1,(SP))

(SP)←(SP)+2

2) 段内带立即数返回

格式：　　RET　　EXP

执行的操作：(IP)←((SP)+1,(SP))

(SP)←(SP)+2

(SP)←(SP)+D_{16}

其中 EXP 是一个表达式，根据它的值计算出来的常数成为机器指令中的位移量 D_{16}。这种指令允许返回地址出栈后修改堆栈的指针，这就便于调用程序在用 CALL 指令调用子程序以前把子程序所需要的参数入栈，以便子程序运行时使用这些参数。当子程序返回后，这些参数已不再有用，就可以修改指针使其指向参数入栈以前的值。这种指令的使用方法将在第六章里说明。

3) 段间返回

格式：　　RET

执行的操作：(IP)←((SP)+1,(SP))

(SP)←(SP)+2

(CS)←((SP)+1,(SP))

(SP)←(SP)+2

4) 段间带立即数返回

格式：　　RET　　EXP

执行的操作：(IP)←((SP)+1,(SP))

(SP)←(SP)+2

(CS)←((SP)+1,(SP))

(SP)←(SP)+2

(SP)←(SP)+D_{16}

这里 EXP 的含义及使用情况与段内带立即数返回指令相同。

CALL 指令和 RET 指令都不影响条件码。

为了说明指令的使用及堆栈情况，举例如下。但这里并不讨论子程序的设计，有关这方面的问题将在第六章说明。

例 3.69　主程序 MAIN 在一个代码段中，子程序 PRO_A、PRO_B、PRO_C 在另一个代码段中。如果这些程序之间的调用关系如图 3.28 所示。则在程序运行时，堆栈情况如图 3.29 所示。读者可以看到在出栈后，堆栈中的内容并未破坏，但如果有新的内容进栈时，原有的内容便自动丢失了。

3.3.5.5　中断

有时当系统运行或者程序运行期间在遇到某些特殊情况时，需要计算机自动执行一组专门的例行程序来进行处理。这种情况称为中断(Interrupt)，所执行的这组程序称为中断例行程序(Interrupt routine)或中断子程序。中断分为内部中断和外部中断两类。内部中断包括象除法运算中遇到需要除以 0 时所产生的中断，或者程序中为了作某些处理而设置的中断指令等。外部中断则主要用来处理 I/O 设备与 CPU 之间的通信。

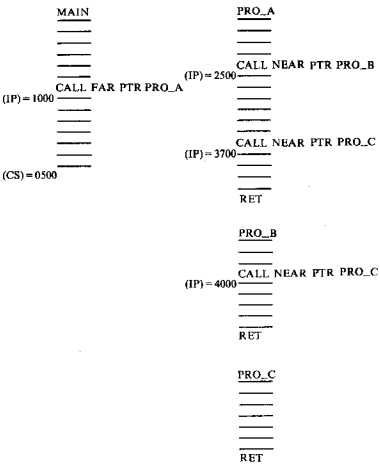


图 3.28　例 3.69 的子程序调用关系

当 CPU 响应一次中断时，也要和调用子程序时类似地把(IP)和(CS)保存入栈。除此之外，为了能全面地保存现场信息，以便在中断处理结束时返回现场，还需要把反映现场状态的(PSW)保存入栈，然后才能转到中断例行程序去执行。当然从中断返回时，除要恢复(IP)和(CS)外，还需要恢复(PSW)。

中断例行程序的入口地址称为中断向量。在 IBM-PC 机中，存储器的最低地址区的 1024 个字节(地址从 00000H 到 003FFH)为中断向量区，其中存放着 256 种类型中断例行程序的入口地址(中断向量)，如图 3.30 所示。由于每个中断向量占有 4 个字节单元，所以中断指令中指定的类型号 N 需要乘以 4 才能取得所指定类型的中断向量，例如如果类型号为 9，则与其相应的中断向量存放在 00024～00027 单元中。IBM-PC 机为每个类型规定了一定的功能，例如类型 0 为除以 0 时的中断例行程序入口，类型 3 为设置断点时的中断例行程序入口，类型 4 为溢出处理的中断例行程序入口，类型 10 为显示设备的中断例行程序入口，类型 20 为程序结束的中断例行程序入口，类型 21 为系统功能调用的中断例行程序入口等。除非特别注明，类型号是以 16 进制形式表示的。

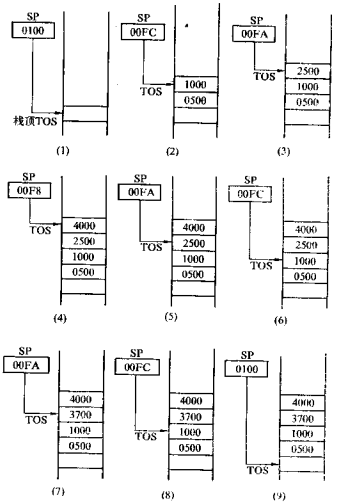


图 3.29　例 3.69 的堆栈情况

(1) MAIN 调用 PRO_A 前　(2) MAIN 调用 PRO_A 后　(3) PRO_A 调用 PRO_B 后
(4) PRO_B 调用 PRO_C 后　(5) PRO_C 返回 PRO_B 后　(6) PRO_B 返回 PRO_A 后
(7) PRO_A 调用 PRO_C 后　(8) PRO_C 返回 PRO_A 后　(9) PRO_A 返回 MAIN 后

有关中断处理问题我们还将在第八章中专门说明，这里只介绍有关中断的几条指令。

INT (Interrupt)　　中断
INTO (Interrupt if overflow)　　如溢出则中断
IRET (Return from interrupt)　　从中断返回

• INT 中断指令

格式：　INT　　TYPE
　　或　　INT
执行的操作：(SP)←(SP)－2
　　((SP)＋1,(SP))←(PSW)
　　(SP)←(SP)－2

((SP)+1,(SP))←(CS)
(SP)←(SP)−2
((SP)+1,(SP))←(IP)
(IP)←(TYPE * 4)
(CS)←(TYPE * 4+2)

其中 TYPE 为类型号，它可以是常数或常数表达式，其值必须在 0～255 的范围内，格式中的 INT 是一个字节的中断指令，它隐含的类型号为 3。INT 指令(包括下面的 INTO)在执行完以上的操作后，再把 IF 和 TF 位置 0，但不影响其余的标志位。

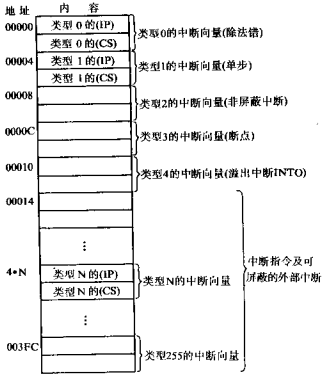


图 3.30　存储器中的中断向量区

· INTO　　若溢出则中断指令

格式：　　INTO

执行的操作：若 OF=1 则：

(SP)←(SP)−2
((SP)+1,(SP))←(PSW)
(SP)←(SP)−2
((SP)+1,(SP))←(CS)
(SP)←(SP)−2
((SP)+1,(SP))←(IP)

(IP)←(10H)

(CS)←(12H)

• IRET　从中断返回指令

格式：　IRET

执行的操作：(IP)←((SP)+1,(SP))

(SP)←(SP)+2

(CS)←((SP)+1,(SP))

(SP)←(SP)+2

(PSW)←((SP)+1,(SP))

(SP)←(SP)+2

该指令的标志位由堆栈中取出的值来设置。

3.3.6　处理机控制指令

3.3.6.1　标志处理指令

除有些指令影响标志位外，IBM-PC 机还提供了一组设置或清除标志位的指令，它们只影响本指令指定的标志，而不影响其他标志位。这些指令是：

- CLC 进位位置 0 指令(Clear carry)CF←0
- CMC 进位位求反指令(Complement carry)CF←$\overline{CF}$
- STC 进位位置 1 指令(Set carry)CF←1
- CLD 方向标志置 0 指令(Clear direction)DF←0
- STD 方向标志位置 1 指令(Set direction)DF←1
- CLI 中断标志置 0 指令(Clear interrupt)IF←0
- STI 中断标志置 1 指令(Set interrupt)IF←1

3.3.6.2　其他处理机控制指令

NOP (No Operation)	无操作
HLT(Halt)	停机
WAIT (Wait)	等待
ESC(Escape)	换码
LOCK(Lock)	封锁

这些指令可以控制处理机状态。它们都不影响条件码。

• NOP 无操作指令

该指令不执行任何操作，其机器码占有一个字节单元，在调试程序时往往用这条指令占有一定的存储单元，以便在正式运行时用其他指令取代。

• HLT 停机指令

该指令可使机器暂停工作，使处理机处于停机状态以便等待一次外部中断到来，中断结束后可继续执行下面的程序。

• WAIT 等待指令

该指令使处理机处于空转状态，它也可以用来等待外部中断发生，但中断结束后仍返回 WAIT 指令继续等待。

• ESC 换码指令

格式 ESC　　mem

其中 mem 指出一个存储单元，ESC 指令把该存储单元的内容送到数据总线去。当然 ESC 指令不允许使用立即数和寄存器寻址方式。这条指令在使用协处理机(Coprocessor)执行某些操作时，可从存储器取得指令或操作数。协处理机(如 8087)则是为提高速度而可以选配的硬件。

· LOCK 封锁指令

该指令是一种前缀，它可与其他指令联合，用来维持总线的锁存信号直到与其联合的指令执行完为止。当 CPU 与其他处理机协同工作时，该指令可避免破坏有用信息。

以上我们已经介绍了 IBM-PC 机的整个指令系统。对一些常用的指令我们作了比较完整的介绍，但对一些不常用的指令只作了简单说明，必要时请读者查阅 IBM PC DOS Macro Assembler 手册。

习　题

3.1　给定

(BX)=637DH，(SI)=2A9BH，位移量 D=7237H，试确定在以下各种寻址方式下的有效地址是什么？

(1) 立即寻址

(2) 直接寻址

(3) 使用 BX 的寄存器寻址

(4) 使用 BX 的间接寻址

(5) 使用 BX 的寄存器相对寻址

(6) 基址变址寻址

(7) 相对基址变址寻址

3.2　试根据以下要求写出相应的汇编语言指令

(1) 把 BX 寄存器和 DX 寄存器的内容相加，结果存入 DX 寄存器中。

(2) 用寄存器 BX 和 SI 的基址变址寻址方式把存储器中的一个字节与 AL 寄存器的内容相加，并把结果送到 AL 寄存器中。

(3) 用寄存器 BX 和位移量 0B2H 的寄存器相对寻址方式把存储器中的一个字和(CX)相加，并把结果送回存储器中。

(4) 用位移量为 0524H 的直接寻址方式把存储器中的一个字与数 2A59H 相加，并把结果送回该存储单元中。

(5) 把数 0B5H 与(AL)相加，并把结果送回 AL 中。

3.3　写出把首地址为 BLOCK 的字数组的第 6 个字送到 DX 寄存器的指令。要求使用以下几种寻址方式：

(1) 寄存器间接寻址

(2) 寄存器相对寻址

(3) 基址变址寻址

3.4　现有(DS)=2000H，(BX)=0100H，(SI)=0002H，(20100)=12H，(20101)=34H，(20102)=56H，(20103)=78H，(21200)=2AH，(21201)=4CH，(21202)=B7H，(21203)=65H，试说明下列各条指令执行完后 AX 寄存器的内容。

(1) MOV　　AX，1200H

(2) MOV　　AX，BX

(3) MOV　　AX，[1200H]

(4) MOV　　AX，[BX]

(5) MOV　　AX，1100[BX]

(6) MOV　　AX，[BX][SI]

(7) MOV AX,1100[BX][SI]

3.5 给定

(IP)=2BC0H,(CS)=0200H,位移量 D=5119H,(BX)=1200H,(DS)=212AH,(224A0)=0600H,(275B9)=098AH,

试为以下的转移指令找出转移的偏移地址。

(1) 段内直接寻址。

(2) 使用 BX 及寄存器间接寻址方式的段内间接寻址。

(3) 使用 BX 及寄存器相对寻址方式的段内间接寻址。

3.6 设当前数据段寄存器的内容为 1B00H,在数据段的偏移地址 2000H 单元内,含有一个内容为 0FF10H 和 8000H 的指针,它们是一个 16 位变量的偏移地址和段地址,试写出把该变量装入 AX 的指令序列,并画图表示出来。

3.7 在 0624 单元内有一条二字节 JMP SHORT OBJ 指令,如其中位移量为(1) 27H,(2) 6BH,(3) 0C6H,试问转向地址 OBJ 的值是多少?

3.8 假定(DS)=2000H,(ES)=2100H,(SS)=1500H,(SI)=00A0H,(BX)=0100H,(BP)=0010H,数据段中变量名 VAL 的偏移地址值为 0050H,试指出下列源操作数字段的寻址方式是什么?其物理地址是多少?

(1) MOV AX,0ABH	(2) MOV AX,BX		
(3) MOV AX,[100H]	(4) MOV AX,VAL		
(5) MOV AX,[BX]	(6) MOV AX,ES:[BX]		
(7) MOV AX,[BP]	(8) MOV AX,[SI]		
(9) MOV AX,[BX+10]	(10) MOV AX,VAL[BX]		
(11) MOV AX,[BX][SI]	(12) MOV AX,VAL[BX][SI]		

3.9 在 ARRAY 数组中依次存储了七个字数据,紧接着是名为 ZERO 的字单元,表示如下:

```
ARRAY   DW   23,36,2,100,32000,54,0
ZERO    DW   ?
```

(1) 如果 BX 包含数组 ARRAY 的初始地址,请编写指令将数据 0 传送给 ZERO 单元。

(2) 如果 BX 包含数据 0 在数组中的位移量,请编写指令将数据 0 传送给 ZERO 单元。

3.10 如 TABLE 为数据段中 0032 单元的符号名,其中存放的内容为 1234H,试问以下两条指令有什么区别?指令执行完后 AX 寄存器的内容是什么?

```
MOV   AX,TABLE
LEA   AX,TABLE
```

3.11 执行下列指令后,AX 寄存器中的内容是什么?

```
TABLE   DW   10,20,30,40,50
ENTRY   DW   3
        ⋮
        MOV   BX,OFFSET TABLE
        ADD   BX,ENTRY
        MOV   AX,[BX]
```

3.12 下列 ASCII 码串(包括空格符)依次存储在起始地址为 CSTRING 的字节单元中:

```
CSTRING   DB   'BASED ADDRESSING'
```

请编写指令将字符串中的第 1 个和第 7 个字符传送给 DX 寄存器。

3.13 已知堆栈段寄存器 SS 的内容是 0FFA0H,堆栈指针寄存器 SP 的内容是 00B0H,先执行两条把 8057H 和 0F79H 分别进栈的 PUSH 指令,再执行一条 POP 指令。试画出堆栈区和 SP 的内容变化过程示意图(标出存储单元的物理地址)。

3.14 设(DS)=1B00H,(ES)=2B00H,有关存储单元的内容如图 3.31 所示。请写出两条指令把字变量 X

装入 AX 寄存器。

3.15 求出以下各十六进制数与十六进制数 62A0 之和，并根据结果设置标志位 SF、ZF、CF 和 OF 的值。

(1) 1234　(2) 4321

(3) CFA0　(4) 9D60

3.16 求出以下各十六进制数与十六进制数 4AF0 的差值，并根据结果设置标志位 SF、ZF、CF 和 OF 的值。

(1) 1234　(2) 5D90

(3) 9090　(4) EA04

3.17 写出执行以下计算的指令序列，其中 X、Y、Z、R、W 均为存放 16 位带符号数单元的地址。

(1) Z←W+(Z-X)

(2) Z←W-(X+6)-(R+9)

(3) Z←(W*X)/(Y+6)，R←余数

(4) Z←((W-X)/5*Y)*2

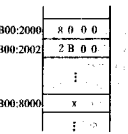


图 3.31　3.14 题的存储区情况

3.18 已知程序段如下：

```
MOV    AX,1234H
MOV    CL,4
ROL    AX,CL
DEC    AX
MOV    CX,4
MUL    CX
INT    20H
```

试问：

(1) 每条指令执行完后，AX 寄存器的内容是什么？

(2) 每条指令执行完后，进位、符号和零标志的值是什么？

(3) 程序结束时，AX 和 DX 的内容是什么？

3.19 下列程序段中的每条指令执行完后，AX 寄存器及 CF、SF、ZF 和 OF 的内容是什么？

```
MOV    AX,0
DEC    AX
ADD    AX,7FFFH
ADD    AX,2
NOT    AX
SUB    AX,0FFFFH
ADD    AX,8000H
SUB    AX,1
AND    AX,58D1H
SAL    AX,1
SAR    AX,1
NEG    AX
ROR    AX,1
```

3.20 变量 DATAX 和变量 DATAY 的定义如下：

```
DATAX    DW    0148H
         DW    2316H
```

```
DATAY    DW    0237H
         DW    4052H
```

请按下列要求写出指令序列：

(1) DATAX 和 DATAY 两个字数据相加，和存放在 DATAY 中。

(2) DATAX 和 DATAY 两个双字数据相加，和存放在从 DATAY 开始的字单元中。

(3) 解释下列指令的作用：

```
STC
MOV    BX,DATAX
ADC    BX,DATAY
```

(4) DATAX 和 DATAY 两个字数据相乘(用 MUL)。

(5) DATAX 和 DATAY 两个双字数据相乘(用 MUL)。

(6) DATAX 除以 23(用 DIV)。

(7) DATAX 双字除以字 DATAY(用 DIV)。

3.21 写出对存放在 DX 和 AX 中的双字长数求补的指令序列。

3.22 试编写一个程序求出双字长数的绝对值。双字长数在 A 和 A+2 单元中，结果存放在 B 和 B+2 单元中。

3.23 写出完成以下操作的程序段。假设各变量的值均为用压缩 BCD 码表示的二位十进制数。

(1) U←V+(S−6)

(2) U←(X+W)−(Z−U)

3.24 如果各变量的值均为用压缩 BCD 码表示的四位十进制数，请完成 3.23 题所要求的工作。

3.25 如果各变量的值均为用非压缩 BCD 码表示的一位十进制数，请完成 3.23 题所要求的工作。

3.26 如果各变量的值均为用非压缩 BCD 码表示的二位十进制数，请完成 3.23 题所要求的工作。如果下式中的 T 为一位非压缩 BCD 数，请写出完成下列操作的程序段。

U←X * Y/T(不要余数)

3.27 假设(BX)=0E3H，变量 VALUE 中存放的内容为 79H，确定下列各条指令单独执行后的结果。

(1) XOR BX,VALUE

(2) AND BX,VALUE

(3) OR BX,VALUE

(4) XOR BX,0FFH

(5) AND BX,0

(6) TEST BX,01H

3.28 试写出执行下列指令序列后 BX 寄存器的内容。执行前(BX)=6D16H。

```
MOV    CL,7
SHR    BX,CL
```

3.29 试用移位指令把十进制数+53 和−49 分别乘以 2。它们应该用什么指令？得到的结果是什么？如果要除以 2 呢？

3.30 试分析下面的程序段完成什么功能？

```
MOV    CL,04
SHL    DX,CL
MOV    BL,AH
SHL    AX,CL
SHR    BL,CL
OR     DL,BL
```

3.31 试写出程序段把 DX,AX 中的双字右移四位。

3.32 假定(DX)=0B9H,(CL)=3,(CF)=1,确定下列各条指令单独执行后 DX 中的值。

(1) SHR DX,1

(2) SAR DX,CL

(3) SHL DX,CL

(4) SHL DL,1

(5) ROR DX,CL

(6) ROL DL,CL

(7) SAL DH,1

(8) RCL DX,CL

(9) RCR DL,1

3.33 下列程序段执行完后,BX 寄存器中的内容是什么?

```
MOV   CL,3
MOV   BX,0B7H
ROL   BX,1
ROR   BX,CL
```

3.34 假设数据定义如下:

```
CONAME   DB   'SPACE EXPLORERS INC'
PRLINE   DB   20 DUP(' ')
```

用串指令编写程序段分别完成以下功能:

(1) 从左到右把 CONAME 中的字符串传送到 PRLINE。

(2) 从右到左把 CONAME 中的字符串传送到 PRLINE。

(3) 把 CONAME 中的第 3 和第 4 个字节装入 AX。

(4) 把 AX 寄存器的内容存入从 PRLINE+5 开始的字节中。

(5) 检查 CONAME 字符串中有无空格字符,如有则把它传送给 BH 寄存器。

3.35 编写程序段,把字符串 STRING 中的'&'字符用空格符代替。

```
STRING   DB   'The date is FEB&03'
```

3.36 假设程序中数据定义如下:

```
STUDENT_NAME   DB   30 DUP(?)
STUDENT_ADDR   DB   9  DUP(?)
PRINT_LINE     DB   132 DUP(?)
```

分别编写下列程序段:

(1) 用空格符清除 PRINT_LINE 域。

(2) 在 STUDENT_ADDR 中查找第一个'—'。

(3) 在 STUDENT_ADDR 中查找最后一个'—'。

(4) 如果 STUDENT_NAME 域中全是空格符时,填入'*'。

(5) 把 STUDENT_NAME 移到 PRINT_LINE 的前 30 个字节中,把 STUDENT_ADDR 移到 PRINT_LINE 的后 9 个字节中。

3.37 编写一程序段,比较两个 5 字节的字符串 OLDS 和 NEWS,如果 OLDS 字符串不同于 NEWS 字符串则执行 NEW_LESS;否则顺序执行程序。

3.38 编写一程序段:查找 TELEPHONE 中的电话号码(10 位)有无'—'字符,如有则转向 FOUND_DASH 去执行;若无则执行 NO_DASH。

3.39 假定 AX 和 BX 中的内容为带符号数,CX 和 DX 中的内容为无符号数,请用比较指令和条件转移指令实现以下判断:

(1) 若 DX 的内容超过 CX 的内容,则转去执行 EXCEED。

(2) 若 BX 的内容大于 AX 的内容，则转去执行 EXCEED。

(3) 若 CX 的内容等于零，则转去执行 ZERO。

(4) BX 与 AX 的内容相比较是否产生溢出？若溢出则转 OVERFLOW。

(5) 若 BX 的内容小于等于 AX 的内容，则转 EQ_SMA。

(6) 若 DX 的内容低于等于 CX 的内容，则转 EQ_SMA。

3.40 试分析下列程序段：

```
ADD     AX,BX
JNO     L1
JNC     L2
SUB     AX,BX
JNC     L3
JNO     L4
JMP     SHORT L5
```

如果 AX 和 BX 的内容给定如下：

	AX	BX
(1)	147B	80DC
(2)	B568	54B7
(3)	42C8	608D
(4)	D023	9FD0
(5)	94B7	B568

问该程序执行完后，程序转向哪里？

3.41 指令 CMP AX,BX 后面跟着一条格式为 J… L1 的条件转移指令，其中…可以是 B、NB、BE、NBE、L、NL、LE、NLE 中的任一个。如果 AX 和 BX 的内容给定如下：

	AX	BX
(1)	1F52	1F52
(2)	88C9	88C9
(3)	FF82	007E
(4)	58BA	020E
(5)	FFC5	FF8B
(6)	09A0	1E97
(7)	8AEA	FC29
(8)	D367	32A6

问以上 8 条转移指令中的哪几条将引起转移到 L1？

3.42 假设 X 和 X+2 单元的内容为双精度数 p，Y 和 Y+2 单元的内容为双精度数 q，(X 和 Y 为低位字) 试说明下列程序段做什么工作？

```
MOV     DX,X+2
MOV     AX,X
ADD     AX,X
ADC     DX,X+2
CMP     DX,Y+2
JL      L2
JG      L1
CMP     AX,Y
JBE     L2
```

```
L1 :    MOV     AX,1
        JMP     SHORT EXIT
L2 :    MOV     AX,2
EXIT :  INT     20H
```

3.43 试编制一个程序段完成图 3.32 中的流程图所规定的功能。

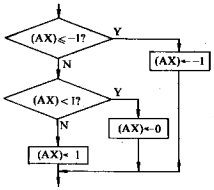


图 3.32 3.43 题的程序流程图

3.44 要求测试在 STATUS 中的一个字节，如果第 1、3、5 位均为 1 则转移到 ROUTINE_1；如果此三位中有两位为 1 则转移到 ROUTINE_2；如果此三位只有一位为 1 则转移到 ROUTINE_3；如果此三位全为 0 则转移到 ROUTINE_4。试画出流程图，并编制相应的程序段。

3.45 已知存储器中有一个首地址为 ARRAY 的 100d 字数组，现要求把数组中的每个数加 1(不考虑溢出的可能性)，试编制完成此功能的程序段。

3.46 用其他指令完成和下列指令同样的功能。

(1) REP MOVSB

(2) REP LODSB

(3) REP STOSB

(4) REPE SCASB

3.47 在下列程序的括号中分别填入如下指令：

(1) LOOP L20

(2) LOOPE L20

(3) LOOPNE L20

试说明在三种情况下，当程序执行完后，AX、BX、CX、DX 四个寄存器的内容分别是什么？

```
TITLE       EXLOOP. COM
CODESG      SEGMENT
            ASSUME CS : CODESG,DS : CODESG,SS : CODESG
            ORG     100H
BEGIN :     MOV     AX,01
            MOV     BX,02
            MOV     DX,03
            MOV     CX,04
L20 :
            INC     AX
            ADD     BX,AX
            SHR     DX,1
            (   )
            RET
CODESG      ENDS
            END     BEGIN
```

3.48 考虑以下的调用序列：

(1) MAIN 调用 NEAR 的 SUBA 过程(返回的偏移地址为 0400)；

(2) SUBA 调用 NEAR 的 SUBB 过程(返回的偏移地址为 0A00)；

(3) SUBB 调用 FAR 的 SUBC 过程(返回的段地址为 B200，偏移地址为 0100)；

(4) 从 SUBC 返回 SUBB；

(5) SUBB 调用 NEAR 的 SUBD 过程(返回的偏移地址为 0C00)；

(6) 从 SUBD 返回 SUBB；

(7) 从 SUBB 返回 SUBA；

(8) 从 SUBA 返回 MAIN；

(9) 从 MAIN 调用 SUBC(返回的段地址是 1000,偏移地址是 0600)；

请画出每次调用及返回时的堆栈状态。

第四章　汇编语言程序格式

4.1　汇编程序功能

图 4.1 表示了对汇编语言程序的处理过程。首先用编辑程序(如行编辑程序 EDLIN 或屏幕编辑程序 WORDSTAR)产生汇编语言的源程序(属性为 ASM 的源文件),源程序就是用汇编语言的语句编写的程序,它是不能为机器所识别的,所以要经过汇编程序加以翻译,因此汇编程序的作用就是把源文件转换成用二进制代码表示的目标文件(称为 OBJ 文件)。在转换的过程中,汇编程序将对源程序进行二遍扫视,如果源程序中有语法错误,则汇编结束后,汇编程序将指出源程序中的错误,用户还可以用编辑程序来修改源程序中的错误,最后得到无语法错误的 OBJ 文件。

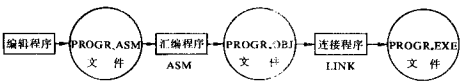


图 4.1　汇编语言程序的建立及汇编过程

OBJ 文件虽然已经是二进制文件,但它还不能直接上机运行,必须经过连接程序(LINK)把目标文件与库文件或其他目标文件连接在一起形成可执行文件(EXE 文件),这个文件可以由 DOS 装入存储器,并在机器上运行。

因此,要在计算机上运行汇编语言程序的步骤是:

1) 用编辑程序建立 ASM 源文件;

2) 用 ASM 程序把 ASM 文件转换成 OBJ 文件;

3) 用 LINK 程序把 OBJ 文件转换成 EXE 文件;

4) 用 DOS 命令直接键入文件名就可执行该程序。

IBM PC 机中有两个汇编程序:小汇编程序 ASM 和宏汇编程序 MASM,宏汇编程序的功能比小汇编程序强,它可以支持宏汇编,但它本身需要较大的存储容量,因此当计算机的存储容量为 64K 时,只能使用小汇编程序;而容量为 96K 时,则可使用两种汇编程序中的任一种。

汇编程序的主要功能是:

1) 检查源程序。

2) 测出源程序中的语法错误,并给出出错信息。

3) 产生源程序的目标程序,并可给出列表文件(同时列出汇编语言和机器语言的文件,称为 LST 文件)。

4) 展开宏指令。

4.2 伪 操 作

汇编语言程序的语句除指令以外还可以由伪操作和宏指令组成。关于宏指令我们将在第七章加以说明，这一节我们只讨论伪操作。伪操作又称为伪指令，它不象机器指令那样是在程序运行期间由计算机来执行的，它是在汇编程序对源程序汇编期间由汇编程序处理的操作，它们可以完成如数据定义、分配存储区、指示程序结束等功能。在这一节里，我们只说明一些常用的伪操作。有些伪操作如有关外部过程所使用的伪操作，有关宏汇编及条件汇编所使用的伪操作等将放在有关的章节中讨论。另外还有一些则在本书中不涉及，需要时请读者查阅手册。

4.2.1 数据定义及存储器分配伪操作

这一类伪操作的格式是：

[Variable] Mnemonic Operand,…,Operand [;Comments]

其中变量(Variable)字段是可有可无的，它用符号地址表示，其作用与指令语句前的标号相同，但它的后面不跟冒号。如果语句中有变量则汇编程序使其记以第一个字节的偏移地址。

注释(Comments)字段用来说明该伪操作的功能，它也是可有可无的。

助记符(Mnemonic)字段说明所用伪操作的助记符，常用的有以下几种：

DB 伪操作用来定义字节，其后的每个操作数都占有一个字节。

DW 伪操作用来定义字，其后的每个操作数占有一个字(低位字节在第一个字节地址中，高位字节在第二个字节地址中)。

DD 伪操作用来定义双字，其后的每个操作数占有二个字。

DQ 伪操作用来定义四个字，其后的每个操作数占有四个字。

DT 伪操作用来定义十个字节，其后的每个操作数占有十个字节，形成压缩的 BCD 码。

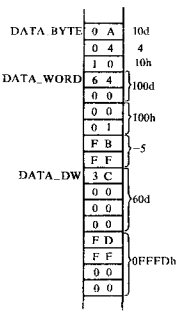


图 4.2 例 4.1 的汇编结果

这些伪操作可以把其后跟着的数据存入指定的存储单元，或者只分配存储器空间而并不存入确定的数值。DW 和 DD 伪操作可以存储偏移地址或完整的地址，下面举例说明：

例 4.1 操作数可以是常数，或者是表达式(根据该表达式可以求得一个常数)，如

```
DATA_BYTE   DB   10,4,10H
DATA_WORD   DW   100,100H,-5
DATA_DW     DD   3*20,0FFFDH
```

汇编程序可以在汇编期间在存储器中存入数据，如图 4.2 所示。

例 4.2 操作数也可以是字符串，如：

```
MESSAGE   DB   'HELLO'
```

则存储器存储情况如图 4.3 (1)所示，而 DB'AB'和 DW'AB'的存储情况则分别如图 4.3 (2)和(3)所示。

例 4.3 操作数？可以保留存储空间，但不存入数据。

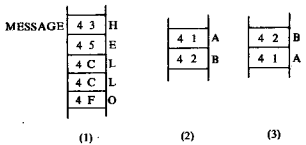


图 4.3 例 4.2 的汇编结果

(1) 字符串的存储 (2) DB 'AB' (3) DW 'AB'

例 4.4

```
ARRAY1  DB  2  DUP(0,1,2,?)
ARRAY2  DB  100  DUP (?)
```

汇编后的存储情况如图 4.5 所示。

由图可见，例 4.4 中的第一个语句和语句

```
ARRAY1  DB  0,1,2,?,0,1,2,?
```

是等价的。

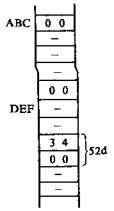


图 4.4 例 4.3 的汇编结果

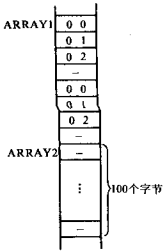


图 4.5 例 4.4 的汇编结果

例 4.5 DUP 操作可以嵌套，例如

```
ARRAY3  DB  100  DUP(0,2  DUP (1,2),0,3)
```

则汇编结果如图 4.6 所示。

可以用 DW 或 DD 伪操作把变量或标号的偏移地址(DW)或整个地址(DD)存入存储器。用 DD 伪操作存入地址时，第一个字为偏移地址，第二个字为段地址。

例 4.6

```
PARAMETER_TABLE  DW  PAR1
```

如：

```
ABC  DB  0,?,?,?,0
DFF  DW  ?,52,?
```

经汇编后的存储情况如图 4.4 所示。

操作数字段还可以使用复制操作符(duplication operator)来复制某个(或某些)操作数。其格式为：

```
repeat_count  DUP(operand,…,operand)
```

其中 repeat_count 可以是一个表达式，它的值应该是一个正整数，用来指定括号中的操作数的重复次数。

```
                DW    PAR2
                DW    PAR3
INTERSEG_DATA   DD    DATA1
                DD    DATA2
```

则汇编后的存储情况如图 4.7 所示。其中偏移地址或段地址均占有一个字，其低位字节占有第一个字节，高位字节占有第二个字节。

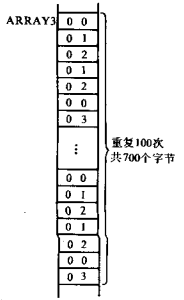


图 4.6　例 4.5 的汇编结果

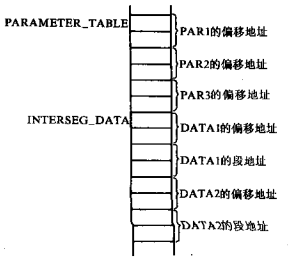


图 4.7　例 4.6 的汇编结果

顺便说明一下，这里操作数字段中的变量或标号可以使用表达式如

Variable±constant expression

label±constant expression

在这种情况下，汇编后，存储器中应该存入表达式的值。

这里，我们再进一步说明变量的类型属性(type attribute)问题。在数据定义伪操作前面的变量的值是该伪操作中的第一个数据项在当前段内的第一个字节的偏移地址。此外，它还有一个类型属性用来表示该语句中的每一个数据项的长度(以字节为单位表示)，因此 DB 伪操作的类型属性为 1，DW 为 2，DD 为 4，DQ 为 8，DT 则为 10，变量表达式的属性和变量是相同的。汇编程序可以用这种隐含的类型属性来确定某些指令是字指令还是字节指令。

例 4.7

```
OPER1    DB    ?,?
OPER2    DW    ?,?
         ⋮
         MOV   OPER1,0
         MOV   OPER2,0
```

则第一条指令应为字节指令，而第二条指令则应为字指令。如果有下列指令序列，

例 4.8

```
OPER1    DB    1,2
```

```
OPER2        DW    1234H,5678H
             ⋮
             MOV   AX, OPER1+1
             MOV   AL,OPER2
```

汇编程序在汇编这一段程序时，能发现两条 MOV 指令的两个操作数的类型属性是不相同的：OPER1+1 为字节类型属性而 AX 为字类型属性；OPER2 为字类型属性而 AL 为字节类型属性。因而汇编程序将指示出错；这两条 MOV 指令中的两个操作数的类型不匹配。

有一个办法可以指定操作数的类型属性，它优先于隐含的类型属性，即使用 PTR 属性操作符。其格式为：

type PTR Variable±constant expression

其中类型可以是 BYTE，WORD 或 DWORD(双字)，这样变量的类型就可以指定了。例 4.8 的第一条指令可以写成：

```
MOV   AX,WORD  PTR  OPER1+1
```

这样就把 OPER1+1 的类型属性指定为字，两个操作数的属性就一致了，汇编时不会出错，而运行时应把 OPER1+1 的字内容送 AX，即把 OPER1+1 的内容送 AL，把 OPER2 的第一个字节的内容送 AH，所以指令执行完后，(AX)=3402H。

例 4.8 的第二条指令应写成

```
MOV   AL,BYTE  PTR  OPER2
```

汇编时不会出错，运行时应把 OPER2 的第一个字节的内容送 AL，即(AL)=34H。而

```
MOV   AL,BYTE  PTR  OPER2+1
```

则应把 OPER2 中的第一个字的高位字节送 AL，即(AL)=12H。

从例 4.8 可以看出：同一个变量可以具有不同的类型属性。除了用属性操作符给以定义外，还可以用 LABEL 伪操作来定义，其格式为：

```
name   LABEL   type
```

对于数据项可表示为：

```
variable_name   LABEL   type
```

其中类型可以是 BYTE，WORD 或 DWORD。

对于可执行的代码，则可表示为：

```
label_name   LABEL   type
```

其中类型可以是 NEAR 或 FAR

例 4.9

```
BYTE_ARRAY   LABEL   BYTE
WORD_ARRAY   DW   50   DUP(?)
```

这样在 100 个字节数组中的第一个字节的地址赋于二个不同类型的变量名：字节类型的变量 BYTE_ARRAY 和字类型变量 WORD_ARRAY。

指令

```
MOV   WORD_ARRAY+2,0
```

把该数组的第 3 个和第 4 个字节置 0，而

```
MOV   BYTE_ARRAY+2,0
```

则把该数组的第 3 个字节置 0。

4.2.2 表达式赋值伪操作 EQU

有时程序中多次出现同一个表达式，为方便起见可以用赋值伪操作给表达式赋予一个名字。其格式如下：

```
Expression_name   EQU   Expression
```

此后，程序中凡需要用到该表达式之处就可以用表达式名来代替了。上式中的表达式可以是任何有效的操作数格式，可以是任何可以求出常数值的表达式，也可以是任何有效的助记符。举例如下：

```
CONSTANT   EQU   256                  数赋以符号名
DATA       EQU   HEIGHT+12            地址表达式赋以符号名
ALPHA      EQU   7
BETA       EQU   ALPHA-2              这是一组赋值伪操作，把7-2=5赋以符号名BETA，
ADDR       EQU   VAR+BETA             VAR+5赋以符号名ADDR。
B          EQU   [BP+8]               变址引用赋以符号名B
P8         EQU   DS:[BP+8]            加段前缀的变址引用赋以符号名P8
```

必须注意 EQU 语句的表达式中如果有变量或标号的表达式，则在该语句前应该先给出它们的定义。例如，语句

```
AB   EQU   DATA_ONE+2
```

则必须放在 DATA_ONE 的定义之后才行，否则汇编程序将指示出错。

另外还有一个与 EQU 相类似的＝伪操作也可以作为赋值伪操作使用。它们之间的区别是 EQU 伪操作中的表达式名是不允许重复定义的，而＝伪操作则允许重复定义。

例如，EMP=6 或 EMP EQU 6 都可以使数 6 赋以符号名 EMP，但不允许两者同时使用。但

```
    ⋮
EMP=7
    ⋮
EMP=EMP+1
    ⋮
```

在程序中是允许使用的，因为＝伪操作允许重复定义。在这种情况下在第一个语句后的指令中 EMP 的值为 7，而在第二个语句后的指令中 EMP 的值则为 8。

4.2.3 段定义伪操作

存储器的物理地址是由段地址和偏移地址组合而成的，汇编程序在把源程序转换为目标程序时，必须确定标号和变量的偏移地址，并且需要把有关信息通过目标模块传送给连接程序，以便连接程序把不同的段和模块连接在一起形成一个可执行程序。为此，需要用段定义伪操作，段定义伪操作的格式如下：

```
segment name     SEGMENT
      ⋮
segment name     ENDS
```

其中删节号部分，对于数据段、附加段和堆栈段来说，一般是存储单元的定义、分配等伪操作；对于代码段则是指令及伪操作。

此外，还必须明确段和段寄存器的关系，这可用 ASSUME 伪操作来实现，其格式为：

```
ASSUME  assignment, …, assignment
```

其中 assignment 说明分配情况，其格式为：

```
segment register name : segment name
```

其中段寄存器名必须是 CS、DS、ES 和 SS 中的一个，而段名则必须是由 SEGMENT 定义的段中的段名。而 ASSUME NOTHING 则可取消前面由 ASSUME 所指定的段寄存器。

举例说明如下：

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * *
data_seg1 segment                ;define data segment
                    :
data_seg1 ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * *
data_seg2 segment                ;define extra segment
                    :
data_seg2 ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * *
code_seg  segment                ;define code segment

        assume cs:code_seg,ds:data_seg1,es:data_seg2
start:                           ;starting execution address
;set DS register to current data segment
        mov  ax,data_seg1        ;data segment addr
        mov  ds,ax               ;  into DS register
;set ES register to current extra segment
        mov  ax,data_seg2        ;extra segment addr
        mov  es,ax               ;  into ES register
                    :
code_seg  ends                   ;end of code segment
;* * * * * * * * * * * * * * * * * * * * * * * * * * * *
        end  start
```

由于 ASSUME 伪操作只是指定某个段分配给哪一个段寄存器，它并不能把段地址装入段寄存器中，所以在代码段中，还必须把段地址装入相应的段寄存器中。为此，在上例的程序中，分别用两条 MOV 指令完成这一操作。如果程序中有堆栈段，也需要把段地址装入 SS 中。但是，代码段不需要这样做，代码段的这一操作是在程序初始化时完成的。

SEGMENT 伪操作还可以增加类型及属性的说明，格式如下：

```
segname  SEGMENT  [align_type]
                  [combine_type]
                  ['class']
          :
segname  ENDS
```

一般情况下，这些说明可以不用。但是，如果需要用连接程序把本程序与其他程序模块相连接时，就需要使用这些说明。分别叙述如下：

· 定位类型(align_type)可以是：

PARA　指定段的起始地址必须从小段边界开始，即段地址的最低的16进制数位必须为0。

BYTE　该段可以从任何地址开始。

WORD　该段必须从字的边界开始，即段地址必须为偶数。

PAGE　该段必须从页的边界开始，即段地址的最低两个16进制数位必须为0(该地址能被256整除)。

· 组合类型　(combine_type)可以是：

PUBLIC　该段连接时将与有相同名字的其他分段连接在一起。其连接次序由连接命令指定。

COMMON　该段在连接时与其他同名分段有相同的起始地址，所以会产生覆盖。COMMON 的连接长度是各分段中的最大长度。

AT　expression　使段的起始地址是表达式所计算出来的16位段地址。但它不能用来指定代码段。

STACK　指定该段在运行时为堆栈段的一部分。

MEMORY　指定该段将分配在所有其他连接在一起的段的前面(在高地址上)，如果连接时有几个指定 MEMORY 的段，则遇到的第一段作为 MEMORY 段，其他则作为 COMMON 段。

· 类别('class')　连接时用于组成段组的名字。

4.2.4　程序开始和结束伪操作

在程序的开始可以用 NAME 或 TITLE 为模块取名字，NAME 的格式是：

```
NAME  module_name
```

汇编程序将以给出的 module_name 作为模块的名字。如果程序中没有 NAME 伪操作，则也可使用 TITLE 伪操作，其格式为：

```
TITLE  text
```

TITLE 伪操作可指定每一页上打印的标题。同时，如果程序中没有使用 NAME 伪操作，则汇编程序将用 text 中的前六个字符作为模块名，text 最多可有60个字符。如果程序中既无 NAME 又无 TITLE 伪操作，则将用源文件名作为模块名。所以 NAME 及 TITLE 伪操作并不是必要的，但一般经常使用 TITLE，以便在列表文件中能打印出标题来。

表示源程序结束的伪操作的格式为：

```
END  [label]
```

其中标号指示程序开始执行的起始地址。如果多个程序模块相连接，则只有主程序要使用标号，其他子程序模块则只用 END 而不必指定标号。图4.8给出求两数之和的绝对值的程序实现，其中用 TITLE 给出标题，用 END START 表示程序结束。汇编程序将在遇 END 时结束汇编，而程序则将从 START 开始执行。

```
title          absolt
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
data_seg    segment                    ;define data segment
  oper1             dw          12
  oper2             dw          230
  result            dw          ?
```

```
data_seg    ends
;*************************************
code_seg    segment                    ;define code segment

            assume          cs:code_seg,ds:data_seg

start:                                 ;starting execution address
            mov     ax,data_seg        ;dataarea segment addr
            mov     ds,ax              ;  into DS register
;MAIN PART OF PROGRAM GOES HERE
            mov     ax,oper1
            add     ax,oper2
            jge     store
            neg     ax
store:      mov     result,ax
            hlt
code_seg    ends                       ;end of code segment
;*************************************
            end     start              ;end assembly
```

图 4.8　求两数和的绝对值的程序

4.2.5　对准伪操作

• EVEN 伪操作使下一个字节地址成为偶数。一个字的地址最好从偶地址开始，所以对于字数组为保证其从偶地址开始，可以在它前面用 EVEN 伪操作来达到这一目的。例如：

```
DATA_SEG        SEGMENT
                ⋮
                EVEN
WORD_ARRAY      DW   100   DUP (?)
                ⋮
DATA_SEG        ENDS
```

• ORG Constant expression

如常数表达式的值为 n，则 ORG 伪操作可以使下一个字节的地址成为常数表达式的值 n。例如：

```
VECTORS     SEGMENT
            ORG   10
VECT1       DW    47A5H
            ORG   20
VECT2       DW    0C596H
            ⋮
VECTORS     ENDS
```

则 VECT1的偏移地址值为0AH，而 VECT2的偏移地址值为14H。

在汇编程序对源程序汇编的过程中，使用地址计数器来保存当前正在汇编的指令的地址。

地址计数器的值可用$来表示，汇编语言允许用户直接用$来引用地址计数器的值，因此

```
JMP   $+8
```

可以表示跳过8个字节的存储区。

在指令和伪操作中也可以直接用$来表示地址计数器的值，如

```
JNE   $+6
```

则转向地址是JNE指令的首地址加上6，即当$用在指令中时，它表示本条指令的第一个字节的地址。在这里，$+6必须是另一条指令的首地址，否则，汇编程序将指示出错信息。当$用在伪操作的参数字段时，则和用在指令中的情况不同，它表示的是地址计数器的当前值。

例4.10

```
ARRAY   DW   1,2,$+4,3,4,$+4
```

如汇编时ARRAY分配的偏移地址为0074，则汇编后的存储区将如图4.9所示。应该注意，ARRAY数组中的两个$+4得到的结果是不同的，这是由于$的值是在不断变化的缘故。当在指令中用到$时，它只代表该指令的首地址，而与$本身所在的字节无关。

ARRAY	0 1	0074
	0 0	
	0 2	
	0 0	
	7 C	
	0 0	
	0 3	
	0 0	
	0 4	
	0 0	
	8 2	
	0 0	

图 4.9 例4.10的汇编结果

4.2.6 基数控制伪操作

汇编程序默认的数为十进制数，因而除非专门指定，汇编程序把程序中出现的数均看作十进制数。为此，当使用其他基数表示的常数时，需要专门给以标记如下：

- 二进制数由一串0、1组成其后跟以字母B，如00101100B。
- 十进制数：由0～9的数字组成的数。一般情况下后面不必加上标记，在指定其他基数的情况下，后面可跟字母D，如178D。
- 十六进制数：由0～9及A～F组成的数，后面跟字母H。这个数的第一个字符必须是0～9，所以如果第一个字符是A～F时，应在其前加上数字0，如0FFFFH。
- 八进制数：由数字0～7组成的数，后面可跟字母O或Q，如1777O。
- RADIX伪操作，可以把默认的基数改变为2～16范围内的任何基数。格式如下：
- RADIX expression

其中表达式用来表示基数值(用十进制数表示)。

例如：

```
MOV   BX,0FFH
MOV   BX,178
```

与

```
- RADIX   16
MOV   BX,0FF
MOV   BX,178D
```

是等价的。应当注意，在用·RADIX 16把基数定为十六进制后，十进制数后面都应跟字母D。在这种情况下，如果某个十六进制数的末字符为D，则应在其后跟字母H，以免与十进制数发生混淆。

· 字符串可以看成串常数，可以用单引号或双引号把字符串放在其中，得到的是字符串的ASCII 码值，例如'ABCD'。

4.3 汇编语言程序格式

汇编语言源程序中的每个语句可以由四项组成，格式如下：

[name] operation operand [;comment]

其中名字项是一个符号。

操作项是一个操作码的助记符，它可以是指令、伪操作或宏指令名。

操作数项由一个或多个表达式组成，它提供为执行所要求的操作而需要的信息。

注释项用来说明程序或语句的功能。；为识别注释项的开始。；也可以从一行的第一个字符开始，此时整行都是注释，常用来说明下面一段程序的功能。

上面四项中带方括号的两项是可有可无的。各项之间必须用"空格"(space)或"横表"(TAB)符隔开。下面分别说明各项的表示方法。

4.3.1 名字项

源程序中用下列字符来表示名字：

字母 A～Z

数字0～9

专用字符?、·、@、_、$

除数字外，所有字符都可以放在源语句的第一个位置。名字中如果用到·则必须是第一个字符。可以用很多字符来说明名字，但只有前面的31个字符能被汇编程序所识别。

一般说来，名字项可以是标号或变量。它们都用来表示本语句的符号地址。它是可有可无的，只有当需要用符号地址来访问该语句时它才需要出现。

· 标号：标号在代码段中定义，后面跟冒号：，它也可以用 LABEL 或 EQU 伪操作来定义。此外它还可以作为过程名定义，这将在以后的章节中加以说明。标号经常在转移指令或 CALL 指令的操作数字段出现，用以表示转向地址。

标号有三种属性：段、偏移及类型。

段属性定义标号的段起始地址，此值必须在一个段寄存器中，而标号的段则总是在 CS 寄存器中。

偏移属性：标号的偏移地址是16位无符号数，它代表从段起始地址到定义标号的位置之间的字节数。

类型属性：用来指出该标号是在本段内引用还是在其它段中引用的。如在段内引用的，则称为 NEAR，指针长度为2字节；如在段外引用，则称为 FAR，指针长度为4字节。

· 变量：变量在除代码段以外的其他段中定义，后面不跟冒号。它也可以用 LABLE 或 EQU 伪操作来定义。变量经常在操作数字段出现。它也有段、偏移及类型三种属性。

段属性定义变量的段起始地址，此值必须在一个段寄存器中。

偏移属性：变量的偏移地址是16位无符号数，它代表从段的起始地址到定义变量的位置之间的字节数。在当前段内给出变量的偏移值等于当前地址计数器的值，当前地址计数器的值可以用 $ 来表示。

类型属性:变量的类型属性定义该变量所保留的字节数。如 BYTE(1个字节长)、WORD(2个字节长)、DWORD(4个字节长)、DQ(8个字节长)、DT(10个字节长),这一点在数据定义伪操作中已作了说明。

在程序中同样的标号或变量的定义只允许出现一次,否则汇编程序会指示出错。

4.3.2 操作项

操作项可以是指令、伪操作或宏指令的助记符。对于指令,汇编程序将其翻译为机器语言指令。对于伪操作,汇编程序将根据其所要求的功能进行处理。对于宏指令,则将根据其定义展开。这在第七章中将专门论述。

4.3.3 操作数项

操作数项由一个或多个表达式组成,多个操作数项之间一般用逗号分开。对于指令,操作数项一般给出操作数地址,它们可能有一个或二个或一个也没有。对于伪操作或宏指令则给出它们所要求的参数。

操作数项可以是常数、寄存器、标号、变量或由表达式组成。在这里,我们将专门对表达式加以说明。表达式是常数、寄存器、标号、变量与一些操作符相组合的序列,可以有数字表达式和地址表达式两种。在汇编期间,汇编程序按照一定的优先规则对表达式进行计算后可得到一个数值或一个地址。为了能了解表达式的组成,我们先介绍一些常用的操作符。

4.3.3.1 算术操作符

算术操作符有十、一、*、/和 MOD。其中 MOD 是指除法运算后得到的余数,如19/7的商是2,而19 MOD 7则为5(余数)。

算术操作符可以用于数字表达式或地址表达式中,但当它用于地址表达式时,只有当其结果有明确的物理意义时其结果才是有效的。例如两个地址相乘或相除是无意义的。在地址表达式中,可以使用十或一,但也必须注意其物理意义,例如把两个不同段的地址相加也是无意义的。经常使用的是地址±数字量,它是有意义的。例如 SUM+1是指 SUM 字节单元的下一个字节单元的地址(注意:不是指 SUM 单元的内容加1),而 SUM-1则是指 SUM 字节单元的前一个字节单元的地址。

例4.11 如要求把首地址为 BLOCK 的字数组的第6个字传送到 DX 寄存器,可用指令如下:

```
MOV  DX,BLOCK+(6-1)*2
```

例4.12 如数组 ARRAY 定义如下,试写出把数组长度(字数)存入 CX 寄存器的指令。

```
ARRAY  DW  1,2,3,4,5,6,7
END    DW  ?
```

其中 END 是为计算数组长度而建立的符号地址,所需指令如下:

```
MOV  CX,(END-ARRAY)/2
```

汇编程序在汇编期间将计算表达式而形成指令

```
MOV  CX,7
```

4.3.3.2 逻辑操作符

它有 AND、OR、XOR 和 NOT。逻辑操作符是按位操作的,它只能用于数字表达式中。

例4.13

```
IN    AL, PORT_VAL
OUT   PORT_VAL  AND  0FEH,AL
```

其中 PORT_VAL 为端口号，OUT 指令中的表达式说明当 PORT_VAL 为偶数时，输出端口号与输入端口号相同，而当 PORT_VAL 为奇数时，则输出端口号比输入端口号小1。

例4.14

```
AND  DX,PORT_VAL  AND  0FEH
```

该指令在汇编时由汇编程序对指令中的表达式进行计算得到一个端口号，而在程序运行时，该指令的执行是把 DX 寄存器的内容与汇编程序计算得到的端口号相"与"，结果保存在 DX 寄存器中。

4.3.3.3 关系操作符

它有 EQ(相等)、NE(不等)、LT(小于)、GT(大于)、LE(小于或等于)、GE(大于或等于)6种。

关系操作符的两个操作数必须都是数字或是同一段内的两个存储器地址。计算的结果应为逻辑值，结果为真，表示为0FFFFH；结果为假，则表示为0。

例4.15

```
MOV  BX,((PORT_VAL  LT  5)AND  20)OR((PORT_VAL  GE  5)AND  30)
```

则当 PORT_VAL<5时，汇编结果应该是：

```
MOV  BX,20
```

否则，汇编结果应该是：

```
MOV  BX,30
```

4.3.3.4 数值回送(Value_returning)操作符

它有 TYPE、LENGTH、SIZE、OFFSET、SEG 5种。这些操作符把一些特征或存储器地址的一部分作为数值回送。下面分别说明各个操作符的功能。

· TYPE

格式为：TYPE Variable 或 label

如果是变量，则汇编程序将回送该变量的以字节数表示的类型：DB 为1，DW 为2，DD 为4，DQ 为8，DT 为10。如果是标号，则汇编程序将回送代表该标号类型的数值：NEAR 为－1，FAR 为－2。

例4.16

```
ARRAY  DW  1,2,3
```

则对于指令 ADD SI,TYPE ARRAY

汇编程序将其形成为：

```
ADD  SI,2
```

· LENGTH

格式为：LENGTH Variable

对于变量中使用 DUP 的情况，汇编程序将回送分配给该变量的单元数。而对于其他情况则回送1。

例4.17

```
FEES  DW  100  DUP(0)
```

对于指令 MOV CX,LENGTH FEES

汇编程序将使其形成为：

```
MOV  CX,100
```

例4.18

```
ARRAY  DW  1,2,3
```

对于指令

```
MOV    CX,LENGTH  ARRAY
```

汇编程序将使其形成为：

```
MOV    CX,1
```

例4.19

```
TABLE  DB  'ABCD'
```

对于指令

```
MOV    CX,LENGTH  TABLE
```

汇编程序将使其形成为：

```
MOV    CX,1
```

· SIZE

格式为：SIZE Variable

汇编程序应回送分配给该变量的字节数。但是，此值是 LENGTH 值和 TYPE 值的乘积，所以，对于例4.17的情况：

```
MOV  CX,SIZE  FEES
```

将形成为

```
MOV  CX, 200
```

对于例4.18的情况：

```
MOV  CX,SIZE  ARRAY
```

将形成为

```
MOV  CX,2
```

而对于例4.19的情况：

```
MOV  CX,SIZE  TABLE
```

将形成为

```
MOV  CX,1
```

· OFFSET

格式为：OFFSET Variable 或 label

汇编程序将回送变量或标号的偏移地址值。

例4.20

```
MOV  BX,OFFSET  OPER_ONE
```

则汇编程序将 OPER_ONE 的偏移地址作为立即数回送给指令，而在执行时则将该偏移地址装入 BX 寄存器中。所以这条指令与指令

```
LEA    BX,OPER_ONE
```

是等价的。

· SEG

格式为：SEG Variable 或 label

汇编程序将回送变量或标号的段地址值。

例4.21 如果 DATA_SEG 是从存储器的05000H 地址开始的一个数据段的段名，OPER1 是该段中的一个变量名，则

```
MOV  BX,SEG  OPER1
```

将把0500H 作为立即数插入指令。实际上由于段地址是由连接程序分配的，所以该立即数是连接时插入的。执行期间则使 BX 寄存器的内容成为0500H。

4.3.3.5 属性操作符

它有 PTR、段操作符、SHORT、THIS、HIGH 和 LOW 6种。

· PTR

格式为： type PTR expression

PTR 用来建立一个符号地址，但它本身并不分配存储器，只是用来给已分配的存储地址赋予另一种属性，使该地址具有另一种类型。格式中的类型字段表示所赋予的新的类型属性，而表达式字段则是被取代类型的符号地址。

例4.22 已有数据定义如下：

```
TWO_BYTE  DW  ?
```

可以用以下语句对这两个字节赋予另一种类型定义：

```
ONE_BYTE      EQU  BYTE  PTR  TWO_BYTE
OTHER_BYTE    EQU  BYTE  PTR  (TWO_BYTE+1)
```

后者也可以写成：

```
OTHER_BYTE      EQU  ONE_BYTE+1
```

这里 ONE_BYTE 和 TWO_BYTE 两个符号地址具有相同的段地址及偏移地址，但是它们的类型属性不同，前者为1，后者为2。

前面已经说明类型可有 BYTE、WORD、DWORD、NEAR 和 FAR 几种，所以 PTR 也可以用来建立字、双字或段内及段间的指令单元。

此外，有时指令要求使用 PTR 操作符。例如用

```
MOV  [BX],5
```

指令把立即数存入 BX 寄存器内容指定的存储单元中，但汇编程序不能分清是存入字单元还是字节单元，此时必须用 PTR 操作符来说明属性，应该写明：

```
MOV  BYTE  PTR [BX],5
```

或

```
MOV  WORD  PTR [BX],5
```

· 段操作符：用来表示一个标量、变量或地址表达式的段属性。例如，用段前缀指定某段的地址操作数

```
MOV  AX,ES:[BX+SI]
```

可见它是用段寄存器：地址表达式来表示的。此外，也可以用段名：地址表达式或组名：地址表达式来表示其段属性。

· SHORT

用来修饰 JMP 指令中转向地址的属性，指出转向地址是在下一条指令地址的±127个字节范围之内。

例如：

```
        JMP     SHORT TAG
                  ⋮
TAG:              ⋮
```

· THIS

格式为 THIS attribute 或 type

它可以象 PTR 一样建立一个指定类型(BYTE、WORD 或 DWORD)的或指定距离(NEAR 或 FAR)的地址操作数。该操作数的段地址和偏移地址与下一个存储单元地址相同。

例如： FIRST_TYPE EQU THIS BYTE

```
WORD_TABLE  DW  100  DUP(?)
```

此时 FIRST_BYTE 的偏移地址值和 WORD_TABLE 完全相同，但它是字节类型的，而 WORD_TABLE 则是字类型的。

又如：

```
START  EQU  THIS  FAR
       MOV  CX,100
```

这样，MOV 指令有一个 FAR 属性的地址 START，这就允许其他段的 JMP 指令直接跳转到 START 来。

· HIGH 和 LOW

称为字节分离操作符，它接收一个数或地址表达式，HIGH 取其高位字节，LOW 取其低位字节。例如：

```
CONST  EQU  0ABCDH
```

则

```
MOV  AH,HIGH  CONST
```

将汇编成

```
MOV  AH,0ABH
```

以上说明了五种类型的常用操作符。我们知道表达式是常数、寄存器、标号、变量和操作符的组合，在计算表达式值时，应该首先计算优先级高的操作符，然后从左到右地对优先级相同的操作符进行计算。括号也可以改变计算次序，括号内的表达式应优先计算。下面给出操作符的优先级别(其中有些操作符我们并未提到过，需要时可从手册中查到)，从高到低排列如下：

1. 在圆括号中的项，方括号中的项，结构变量(变量、字段)，然后是 LENGTH，SIZE、WIDTH 和 MASK。
2. 名：(段取代)。
3. PTR，OFFSET，SEG，TYPE，THIS 及段操作符。
4. HIGH 和 LOW。
5. 乘法和除法：*，/，MOD，SHL，SHR。
6. 加法和减法：+，-。
7. 关系操作：EQ，NE，LT，LE，GT，GE。
8. 逻辑：NOT
9. 逻辑：AND
10. 逻辑：OR，XOR。
11. SHORT

现在，我们可以计算出表达式的值了。当然，实际上表达式的值是由汇编程序计算的，而程序员应该正确掌握书写表达式的方法，以减少出错的可能性。

4.3.4 注释项

注释项用来说明一段程序或一条或几条指令的功能，它是可有可无的。但是，对于汇编语言程序来说，注释项的作用是很明显的，它可以使程序易于被读懂，因此编制汇编语言程序必须写好注释。注释应该写出本条(或本段)指令在程序中的功能和作用，而不应该只写指令的动作。读者在有机会阅读程序例子时，应注意学习注释的写法，在编制程序时，更应学会写好注释。

根据以上说明的汇编语言程序格式及有关的伪操作定义，给出源文件 (ASM) 格式如图 4.10所示。

```
;PROGRAM TITLE GOES HERE——
;   Followed by descriptive phrases
;EQU STATEMENTS GO HERE
;*********************************
datarea segment                 ;define data segment
;DATA GOES HERE
datarea ends
;*********************************
prognam segment                 ;define code segment
;-------------------------------
main    proc    far             ;main part of program
        assume cs:prognam,ds:datarea
start:                          ;starting execution address
;set up stack for return
        push    ds              ;save old data segment
        sub     ax,ax           ;put zero in AX
        push    ax              ;save it on stack
;set DS register to current data segment
        mov     ax,datarea      ;datarea segment addr
        mov     ds,ax           ;  into DS register
;MAIN PART OF PROGRAM GOES HERE
        ret                     ;return to DOS
main    endp                    ;end of main part of program
;-------------------------------
subr1   proc    near            ;define subprocedure
;SUBROUTINE GOES HERE
subr1   endp                    ;end subprocedure
;-------------------------------
prognam ends                    ;end of code segment
;*********************************
        end     start           ;end assembly
```

图 4.10 汇编语言源程序格式举例

这里要说明两点：

1. 其中关于建立过程的 PROC 和 ENDP 伪操作对将在以后的章节中说明，这里只要知道利用这一对伪操作把程序段分为若干个过程，使程序的结构更加清晰就可以了。

2. 这里只定义了最基本的代码段和数据段，如果程序中还需定义附加段和堆栈段，则定义的方式及建立段寄存器的方法是相同的，读者可自行设计。

3. 这里把主程序建立为过程，由 DOS 调用该过程，进入程序后，首先把 DS 的内容和0作为段地址和偏移地址入栈，以便在程序结束时用 RET 指令返回 DOS，这是一种较好的工作方

式。如果在主程序开始时没有用上面三条指令在堆栈中建立返回信息，则在程序结束时就不能直接用 RET 返回指令，而应该使用编号为4C的功能调用返回 DOS，如下所示：

```
MOV   AX,4C00H
INT   21H
```

4.4 汇编语言程序的上机过程

在4.1里，我们已经简单地说明了汇编语言程序从建立到执行的过程，这一节我们将说明这一过程的具体操作方法。

4.4.1 建立汇编语言的工作环境

为运行汇编语言程序至少要在磁盘上建立以下文件：

```
ASM. EXE
LIKE. EXE
EXE2BIN. EXE
DEBUG. COM
EDLIN. COM
WS. COM
WSMSGS. OVR
WSOVLYL. OVR
```

其中，ASM 为小汇编程序，它不支持宏汇编，如果需要宏汇编，则应使用 MASM 程序及其他机器提供的 Macro Assembler disk 上的所有文件。LINK 为连接程序。EXE2BIN 为转换成 COM 文件所需要的程序。COM 文件也是一种可执行文件，我们将在4.4.6节加以说明。DEBUG 是调试程序，它是运行汇编语言程序必不可少的工具。EDLIN 是行编辑程序，最后三个以 WS 字母开头的文件则是 WORDSTAR 程序，它是一种全屏幕汇编程序，比使用 EDLINE 更加方便，读者可任意选用。

4.4.2 建立 ASM 文件

为了说明汇编语言程序上机运行的过程，今举例如下：

例4.23 请把40个字母 a 的字符串从源缓冲区传送到目的缓冲区。我们可以用字处理程序 WORDSTAR 在磁盘上建立源程序，调用 WORDSTAR

```
C> WS↙
```

使用非文本文件方式(n 命令)建立以 EX_MOVS. ASM 为文件名的源文件如图4.11所示。

```
;PROGRAM   TITLE   GOES   HERE——ex_movs
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
data          segment                    ;define data segment
  source_buffer          db          40 dup  ('a')
data          ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
extra         segment                    ;define extra segment
  dest_buffer            db          40 dup  (?)
extra         ends
```

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
code    segment                         ;define code segment
;-----------------------------------------------------------
main    proc    far                     ;main part of program
        assume cs:code,ds:data,es:extra
start:                                  ;starting execution address
;set up stack for return
        push    ds                      ;save old data segment
        sub     ax,ax                   ;put zero in AX
        push    ax                      ;save it on stack
;set DS register to current data segment
        mov     ax,data                 ;data segment addr
        mov     ds,ax                   ;   into DS register
;set ES register to current extra segment
        mov     ax,extra                ;extra segment addr
        mov     es,ax                   ;   into ES register
;MAIN  PART  OF  PROGRAM  GOES  HERE
        lea     si,source_buffer        ;put offset addr of source
                                        ;   buffer in  SI
        lea     di,dest_buffer          ;put offset addr of dest
                                        ;   buffer in  DI
        cld                             ;set DF flag to forward
        mov     cx,40                   ;put count in CX
        rep     movsb                   ;move entire string
        ret                             ;return to DOS
main    endp                            ;end of main part of program
;-----------------------------------------------------------
code    ends                            ;end of code segment
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        end     start                   ;end assembly
```

图 4.11　例4.23的源文件 EX_MOVS. ASM

WORDSTAR 的使用方法可查阅手册。如果你没有 WORDSTAR 则可使用 EDLIN 行编辑程序或其他编辑程序。

4.4.3　用 ASM(或 MASM)程序产生 OBJ 文件

源文件建立后，就要用汇编程序对源文件汇编，汇编后产生二进制的目标文件(OBJ 文件)，其操作与汇编程序回答如下：

```
C>asm ex_movs
The IBM Personal Computer Assembler
Version 1.00 (C) Copyright IBM Corp 1981
Object filename [EX_MOVS.OBJ]:
```

```
Source listing   [NUL. LST]:ex_movs
Cross reference  [NUL. CRF]:ex_movs

Warning  Severe
Errors   Errors
0        0
```

汇编程序的输入文件是 ASM 文件，其输出文件可以有三个，表示于回答的中间三行。第一个是 OBJ 文件，这是我们汇编的主要目的，所以这个文件我们是需要的，对于[EX_MOVS. OBJ]后的:应回答↙，这样就在磁盘上建立了这一目标文件。第二个是 LST 文件，称为列表文件。这个文件同时列出源程序和机器语言程序清单，并给出符号表，因而可使程序调试更加方便。这个文件是可有可无的，如果不需要则可对[NUL. LST]:回答↙；如果需要这个文件，则可回答文件名，这里是 ex_movs↙，这样 EX_MOVS. LST 就建立起来了。图4. 12是该程序的 LST 清单。由于打印机的行宽是80，所以注释部分不能在一行内打印完，而要移到下一行才能打完，所以阅读程序时不大方便。LST 清单的最后部分为段名表和符号表，表中分别给出段名，段的大小及有关属性，以及用户定义的符号名，类型及属性。

```
1
2
3                                       ;PROGRAM  TITLE  GOES  HERE——ex_movs
4                                       ;* * * * * * * * * * * * * *
5     0000                              data      segment
                                        ;define data segment
6     0000    28    [                   source_buffer  db  40  dup ('a')
7             61]
8                   ]
9
10    0028                              data      ends
11                                      ;* * * * * * * * * * * * * *
12    0000                              extra segment
                                        ;define extra segment
13    0000    28    [                   dest_buffer  db  40  dup (?)
14                        ??
15                                  ]
16
17    0028                              extra     ends
18                                      ;* * * * * * * * * * * * * *
19    0000                              code      segment
                                        ;define code segment
20                                      ;————————————————————————————
21    0000                              main      proc      far
                                        ;main part of program
22
23                                            assume cs:code,ds:data,es:extra
24
25    0000                              start:
                                        ;starting execution address
26
27                                      ;set up stack for return
28    0000    1E                        push      ds
                                        ;save old data segment
29    0001    2B    C0                  sub   ax,ax
                                        ;put zero in AX
```

```
30      0003    50              push    ax
                                ;save it on stack
31
32                              ;set DS register to current data segment
33      0004    B8  ---- R      mov     ax,data
                                ;data segment addr
34      0007    8E  D8          mov     ds,ax
                                ;  into DS register
35
36                              ;set ES register to current extr a segment
37      0009    B8  ---- R      mov     ax,extra
                                ;extra segment addr
38      000C    8E  C0          mov     es,ax
                                ;  into ES register
39
40                              ;MAIN PART OF PROGRAM GOES HERE
41      000E    8D  36 0000 R   lea     si,source_buffer
                                ;put offset addr of source
42
                                ;  buffer in SI
43      0012    26: 8D 3E 0000 R lea    di,dest_buffer
                                ;put offset addr of dest
44
                                ;  buffer in DI
45      0017    FC              cld
                                ;set DF flag to forward
46      0018    B9  0028        mov     cx,40
                                ;put count in CX
47      001B    F3/ A4          rep     movsb
                                ;move entire string
48      001D    CB              ret
                                ;return to DOS
49
50      001E                    main    endp
                                ;end of main part of program
51                              ;-------------------------------
52      001E                    code    ends
                                ;end of code segment
53                              ;**************
54
55                              end     start
                                ;end assembly
```

The IBM Personal Computer Assembler 06-24-82 PAGE Symbols-
1

Segments and groups:

Name	Size	align	combine class
CODE	001E	PARA	NONE
DATA	0028	PARA	NONE
EXTRA.	0028	PARA	NONE

Symbols:

Name	Type	Value	Attr
DEST_BUFFER.	L BYTE	0000	EXTRA Length = 0028
MAIN	F PROC	0000	CODE Length = 001E

```
SOURCE_BUFFER. . . . . L BYTE  0000  DATA Length = 0028
START. . . . . L NEAR  0000  CODE

Warning  Severe
Errors   Errors
0        0
```

图 4.12 例4.23的列表文件 EX_MOVS.LST

汇编程序能提供的第三个文件是 CRF 文件，这个文件用来产生交叉引用表 REF，对于一般程序不需要建立此文件，所以对于第三行的[NUL.CRF]：可以用↙来回答，这样就完成了汇编过程。如果你希望建立交叉引用表，则应该用文件名来回答，这里是 ex_movs↙，这就产生了 EX_MOVS.CRF 文件。为了建立交叉引用表，必须调用 CREF 程序，所以如果需要就应该从系统中把 CREF.EXE 文件 COPY 到你的磁盘上，此后就键入

```
C>cref ex_movs↙
```

机器回答

```
List filename [EX_MOVS.REF]:
```

用↙回答就建立了 EX_MOVS.REF 文件，如图4.13所示。

```
Symbol Cross Reference          (# is definition)    Cref-1

CODE . . . . . 19#  23  52
DATA . . . . . 5#   10  23  33
DEST_BUFFER. . . . . 13#  43
EXTRA. . . . . 12#  17  23  37
MAIN . . . . . 21#  50
SOURCE_BUFFER. . . . . 6#  41
START. . . . . 25#  55
```

图 4.13 例4.23的交叉引用表 EX_MOVS.REF

交叉引用表给出了用户定义的所有符号，对于每个符号列出了其定义所在行号(加上#)及引用的行号。可以看出它为大程序的修改提供了方便，而一般较小的程序则可不使用。

到此为止，汇编过程已经完成了。但是，汇编程序还有另一个重要功能：可以给出源程序中的错误信息。警告错误(Wraning Errors)指出汇编程序所认为的一般性错误，严重错误(Severe Errors)则指出汇编程序认为已使汇编程序无法进行正确汇编的错误。除给出错误的个数外，汇编程序还能指出错误信息，如5号错是符号重复定义错，9号则是符号没有定义错等。ASM 程序只给出出错号，附录七给出了汇编出错信息的含义，供读者调试程序时查阅。MASM 程序则可直接显示出错误性质。如果你的程序有错，则应重新调用编辑程序修改错误，并重新汇编直到汇编正确通过为止。当然汇编程序只能指出你的程序中的语法错误，至于程序的算法或编制程序中的其他错误则应在程序调试时去解决。

4.4.4 用 LINK 程序产生 EXE 文件

汇编程序已产生出二进制的目标文件(OBJ)，但 OBJ 文件并不是可执行的文件，还必须使用连接程序(LINK)把 OBJ 文件转换为可执行的 EXE 文件，其原因我们将在第十三章说明。当然，如果一个程序是由多个模块组成时，也应该通过 LINK 把它们连接在一起，操作方法及机器回答如下：

```
C>link ex_movs

IBM 5550 Multistation Linker 2.00
(C) Copyright IBM Corp. 1983

Run File [EX_MOVS.EXE]:
List File [NUL.MAP]: ex_movs
Libraries [.LIB]:
Warning: No STACK segment

There was 1 error detected.
```

LINK 程序有两个输入文件 OBJ 和 LIB。OBJ 是我们需要连接的目标文件，LIB 则是程序中需要用到的库文件，如无特殊需要，则应对[LIB]:回答↙。LINK 程序有两个输出文件，一个是 EXE 文件，这当然是我们需要的，应对[EX_MOVS.EXE]:回答↙，这样就在磁盘上建立了该可执行文件。LINK 的另一个输出文件为 MAP 文件，它是连接程序的列表文件，又称为连接映象(Link map)，它给出每个段在存储器中的分配情况。图4.14给出了 EX_MOVS.MAP 文件。

```
Warning: No STACK segment

 Start   Stop    Length   Name                    Class
 00000H  0001DH  001EH    CODE
 00020H  00047H  0028H    DATA
 00050H  00077H  0028H    EXTRA

 Origin    Group

Program entry point at 0000:0000
```

图 4.14 例4.23的连接映象 EX_MOVS MAP

连接程序给出的无堆栈段的警告性错误并不影响程序的执行，所以，到此为止，连接过程已经结束，可以执行 EX_MOVS 程序了。

4.4.5 程序的执行

在建立了 EXE 文件后，就可以直接从 DOS 执行程序，如下所示：

```
C>ex_movs↙
C>
```

程序运行结束并返回 DOS。如果用户程序已直接把结果在终端上显示出来，那么程序已经运行结束，结果也已经得到了。但是，我们的 EX_MOVS 程序并未显示出结果，那么，你怎么知道程序执行的结果是否正确呢？此外，大部分程序必须经过调试阶段才能纠正程序执行中的错误，得到正确的结果，那么又怎样来调试程序呢？这里要使用 DEBUG 程序。有关 DEBUG 程序的各种命令请读者阅读手册，本书附录六介绍了 DEBUG 的主要命令。这里我们介绍几个最常用的命令。先进入 DEBUG 并装入我们要调试的程序 EX_MOVS.EXE，键入如下：

```
C>debug ex_movs.exe
-
```

debug 以短划线_来回答。我们的目的是要察看我们程序的运行结果，因此我们希望启动程序运行后应停在返回 DOS 以前，为此可先用反汇编命令 U 来确定我们所要设定的断点地址。键入 U 后显示信息如图4.15所示。

```
-u
18F4:0000  1E          PUSH    DS
18F4:0001  2BC0        SUB     AX,AX
18F4:0003  50          PUSH    AX
18F4:0004  B8F618      MOV     AX,18F6
18F4:0007  8ED8        MOV     DS,AX
18F4:0009  B8F918      MOV     AX,18F9
18F4:000C  8EC0        MOV     ES,AX
18F4:000E  8D360000    LEA     SI,[0000]
18F4:0012  26          ES:
18F4:0013  8D3E0000    LEA     DI,[0000]
18F4:0017  FC          CLD
18F4:0018  B92800      MOV     CX,0028
18F4:001B  F3          REPZ
18F4:001C  A4          MOVSB
18F4:001D  CB          RETF
18F4:001E  0000        ADD     [BX+SI],AL
```

图 4.15　例4.23用 DEBUG 调试时，U 命令的显示情况

图中最左边给出了程序所在地的段地址:偏移地址，然后是机器语言指令，右边则是汇编语言指令，我们所需要的断点是在 REP MOVSB 指令运行完后，所以选择其下一条指令的偏移地址001D 为断点。当然，我们在前面已经建立有 EX_MOVS. LST 文件，所以可以从 LST 文件中找到 RET 指令的偏移地址为001D，这里已经不必再用 U 命令来查找了，其实，我们不过在这里又给出一种确定指令地址的方法而已。在确定断点后，就可以用 G 命令使程序启动运行，同时设定断点001D 如下

```
-g1d
AX=18F9 BX=0000 CX=0000 DX=0000 SP=FFFC BP=0000 SI=0028 DI=0028 DS=18F6 ES=
18F9 SS=18F4 CS=18F4 IP=001D NV UP DI PL 2R NA PE NC
18F4:001D CB            RETF
```

程序停在断点处，并显示出所有寄存器以及各标志位的当前值，最后一行给出下一条将要执行指令的地址、机器语言及汇编语言。我们可以从显示的寄存器内容来了解程序运行是否正确。对于我们正在执行的 EX_MOVS 程序，由于它是要求把存储器数据段中的一串字符传送到附加段去，所以单从寄存器的内容看不到程序运行的结果，而需要用 D 命令分别察看数据段和附加段的有关区域，如图4.16所示。

```
d18f6:0
18F6:0000  61 61 61 61 61 61 61 61-61 61 61 61 61 61 61 61   aaaaaaaaaaaaaaaa
18F6:0010  61 61 61 61 61 61 61 61-61 61 61 61 61 61 61 61   aaaaaaaaaaaaaaaa
18F6:0020  61 61 61 61 61 61 61 61-00 00 00 00 00 00 00 00   aaaaaaaa........
18F6:0030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
18F6:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
18F6:0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
18F6:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
18F6:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

```
-d19f9:0
18F4:0000  61 61 61 61 61 61 61 61 -61 61 61 61 61 61 61 61   aaaaaaaaaaaaaaaa
18F4:0010  61 61 61 61 61 61 61 61 -61 61 61 61 61 61 61 61   aaaaaaaaaaaaaaaa
18F4:0020  61 61 61 61 61 61 61 61 -FF FF FF FF E1 18 35 2C   aaaaaaaa....a.5,
18F4:0030  C5 15 00 00 00 00 00 00 -00 00 00 00 00 00 00 00   E...............
18F4:0040  00 00 00 00 00 00 00 00 -00 00 00 00 00 00 00 00   ................
18F4:0050  CD 21 CB 00 00 00 00 00 -00 00 00 00 00 20 20 20   M!K.........
18F4:0060  20 20 20 20 20 20 20 20 -00 00 00 00 00 20 20 20                .....
18F4:0070  20 20 20 20 20 20 20 20 -00 00 00 00 CC CC CC CC                ....LLLL
```

图 4.16　例4.23用 DEBUG 调试时,D 命令的显示情况

其中,左边给出每一小段的起始地址(用段地址:偏移地址表示),然后顺序给出小段中每个字节单元的内容,中间是用十六进制数表示的内容,右边则是用字符表示的内容。可以看出数据段中的字符 a 已正确地传送到了附加段中。这样,我们已达到了目的,可以用 Q 命令退出 DEBUG 程序回到 DOS,如下所示:

```
-q
c>
```

除上述命令外,DEBUG 的常用命令还有 R 可以显示当前寄存器的内容,得到的结果和设断点时给出的格式相同。此外,T 命令可以逐条跟踪执行指令。其他命令请查阅手册中有关 DEBUG 程序的说明。初学汇编语言程序设计的读者必须学会 DEBUG 的使用。虽然一开始要花费一些时间去学习有关命令,但它在调试程序时对你的帮助将是很大的,特别是一些较大的程序没有 DEBUG 的帮助,调试将会是十分困难的。

4.4.6　COM 文件

COM 文件也是一种可执行文件,由程序本身的二进制代码组成,没有 EXE 文件所具有的包括有关文件信息的标题区(header),所以它占有的存储空间比 EXE 文件要小。COM 文件不允许分段,它所占有的空间不允许超过64K。因而只能用来编制较小的程序。由于其小而简单,装入速度比 EXE 文件要快。

使用 COM 文件时,程序不分段,其入口点(开始运行的起始点)必须是100H(其前的256个字节为程序段前缀所在地,这一点在第十二章将作说明),且不必设置堆栈段。在程序装入时,由系统自动把 SP 建立在该段之末。对于所有的过程则应定义为 NEAR。图4.17给出 COM 文件的源程序框架。

```
;PROGRAM  TITLE  GOES  HERE---
;  Followed by descriptive phrases

;EQU statements go here

;* * * * * * * * * * * * * * * * * * * * * * * * * * * *
prognam segment

        org     100h

        assume cs:prognam,ds:prognam,es:prognam,ss:prognam
;-----------------------------------------------------------
main    proc    near

;Program goes here
```

```
        mov     ax,4c00h                ;return to DOS
        int     21h
;(or    int     20h)
main    endp
;-------------------------------------------------
;DATA goes here
prognam ends
;*********************************************
        end     main
```

图 4.17 COM 文件的源程序框架

用户在建立源文件以后，同样经过汇编、连接形成 EXE 文件，然后可以通过 EXE2BIN 程序来建立 COM 文件，操作方法如下：

```
c>exe2bin filename filename.com↙
```

请读者注意上行中第一个 filename 给出已形成的 EXE 文件的文件名，但不必给出文件扩展名。第二个 filename 即为所要求的 COM 文件的文件名，它必须带有文件扩展名 COM，这样就形成了所要的 COM 文件，在 DOS 系统下，可直接在机器上用文件名执行。如果第二个 filename 后不跟扩展名，则将形成 BIN 文件，在 DOS 系统下运行该程序时，必须先用 rename 命令把它改名为 COM 文件才能直接运行。

此外，COM 文件还可以直接在调试程序 DEBUG 中用 A 或 E 命令建立，对于一些短小的程序，这也是一种相当方便的方法。

习　题

4.1　指出下列指令的错误：

(1) MOV　AH,BX

(2) MOV　[BX],[SI]

(3) MOV　AX,[SI][DI]

(4) MOV　MYDAT[BX][SI],ES:AX

(5) MOV　BYTE　PTR[BX],1000

(6) MOV　BX,OFFSET　MYDAT[SI]

(7) MOV　CS,AX

4.2　下面哪些指令是非法的？(假设 OP1,OP2是已经用 DB 定义的变量)

(1) CMP　15,BX

(2) CMP　OP1,25

(3) CMP　OP1,OP2

(4) CMP　AX,OP1

4.3　假设下列指令中的所有标识符均为类型属性为字的变量，请指出下列指令中哪些是非法的？它们的错误是什么？

(1) MOV　BP,AL

(2) MOV　WORD_OP[BX+4*3][DI],SP

(3) MOV　WORD_OP1,WORD_OP2

(4) MOV　AX,WORD_OP1[DX]

(5) MOV　SAVE_WORD,DS

(6) MOV　SP,SS:DATA_WORD[BX][SI]

(7) MOV [BX][SI],2

(8) MOV AX,WORD_OP1+WORD_OP2

(9) MOV AX,WORD_OP1-WORD_OP2+100

(10) MOV WORD_OP1,WORD_OP1-WORD_OP2

4.4 假设 VAR1和 VAR2为字变量,LAB 为标号,试指出下列指令的错误之处:

(1) ADD VAR1,VAR2

(2) SUB AL,VAR1

(3) JMP LAB[SI]

(4) JNZ VAR1

(5) JMP NEAR LAB

4.5 画图说明下列语句所分配的存储空间及初始化的数据值。

(1) BYTE_VAR DB 'BYTE',12,-12H,3 DUP(0,?,2 DUP(1,2),?)

(2) WORD_VAR DW 5 DUP(0,1,2),?,-5,'BY','TE',256H

4.6 试列出各种方法,使汇编程序把5150H 存入一个存储器字中(例如:DW 5150H)。

4.7 请设置一个数据段 DATASG,其中定义以下字符变量或数据变量。

(1) FLD1B 为字符串变量:'personal computer';

(2) FLD2B 为十进制数字节变量:32;

(3) FLD3B 为十六进制数字节变量:20;

(4) FLD4B 为二进制数字节变量:01011001;

(5) FLD5B 为数字的 ASCII 字符字节变量:32654;

(6) FLD6B 为10个零的字节变量;

(7) FLD7B 为零件名(ASCII 码)及其数量(十进制数)的表格:

PART1 20

PART2 50

PART3 14

(8) FLD1W 为十六进制数字变量:FFF0;

(9) FLD2W 为二进制数字变量:01011001;

(10) FLD3W 为(7)中零件表的地址变量;

(11) FLD4W 为包括5个十进制数的字变量:5,6,7,8,9;

(12) FLD5W 为5个零的字变量;

(13) FLD6W 为本段中字数据变量和字节数据变量之间的地址差。

4.8 假设程序中的数据定义如下:

```
PARTNO    DW  ?
PNAME     DB  16  DUP(?)
COUNT     DD  ?
PLENTH    EQU $-PARTNO
```

问 PLENTH 的值为多少?它表示什么意义?

4.9 有符号定义语句如下:

```
BUFF  DB  1,2,3,'123'
EBUFF DB  0
L     EQU EBUFF-BUFF
```

问 L 的值为多少?

4.10 假设程序中的数据定义如下:

```
LNAME     DB  30  DUP(?)
```

```
ADDRESS     DB  30  DUP(?)
CITY        DB  15  DUP(?)
CODE_LIST   DB  1,7,8,3,2
```

(1) 用一条 MOV 指令将 LNAME 的偏移地址放入 AX。

(2) 用一条指令将 CODE_LIST 的头两个字节的内容放入 SI。

(3) 写一条伪操作使 CODE_LENGTH 的值等于 CODE_LIST 域的实际长度。

4.11 试写出一个完整的数据段 DATA_SEG，它把整数5赋与一个字节，并把整数－1,0,2,5和4放在10字数组 DATA_LIST 的头5个单元中，然后写出完整的代码段，其功能为：把 DATA_LIST 中头5个数中的最大值和最小值分别存入 MAX 和 MIN 单元中。

4.12 给出等值语句如下：

```
ALPHA   EQU  100
BETA    EQU  25
GAMMA   EQU  2
```

下列表达式的值是多少？

(1) ALPHA * 100+BETA

(2) ALPHA MOD GAMMA+BETA

(3) (ALPHA+2) * BETA－2

(4) (BETA/3)MOD 5

(5) (ALPHA+3) * (BETA MOD GAMMA)

(6) ALPHA GE GAMMA

(7) BETA AND 7

(8) GAMMA OR 3

4.13 对于下面的数据定义，三条 MOV 指令分别汇编成什么？(可用立即数方式表示)

```
TABLEA   DW   10 DUP (?)
TABLEB   DB   10 DUP (?)
TABLEC   DB   '1234'
  ⋮
MOV      AX,LENGTH TABLEA
MOV      BL,LENGTH TABLEB
MOV      CL,LENGTH TABLEC
```

4.14 对于下面的数据定义，各条 MOV 指令单独执行后，有关寄存器的内容是什么？

```
FLDB     DB      ?
TABLEA   DW      20 DUP (?)
TABLEB   DB      'ABCD'
```

(1) MOV AX,TYPE FLDB

(2) MOV AX,TYPE TABLEA

(3) MOV CX,LENGTH TABLEA

(4) MOV DX,SIZE TABLEA

(5) MOV CX,LENGTH TABLEB

4.15 指出下列伪操作表达方式的错误，并改正之。

(1) DATA_SEG SEG

(2) SEGMENT 'CODE'

(3) MYDATA SEGMENT/DATA

```
        ⋮
        ENDS
(4) MAIN_PROC   PROC FAR
        ⋮
        END   MAIN_PROC
    MAIN_PROC   ENDP
```

4.16 按下面的要求写出程序的框架

(1) 数据段的位置从0E000H开始，数据段中定义一个100字节的数组，其类型属性既是字又是字节；

(2) 堆栈段从小段开始，段组名为STACK；

(3) 代码段中指定段寄存器，指定主程序从1000H开始，给有关段寄存器赋值；

(4) 程序结束。

4.17 假设在数据段X_SEG、附加段Y_SEG和堆栈段Z_SEG中分别定义了字变量X、Y和Z。试编制一完整的程序计算

$$X \leftarrow X+Y+Z$$

4.18 写一个完整的程序放在代码段C_SEG中，要求把数据段D_SEG中的AUGEND和附加段E_SEG中的ADDEND相加，并把结果存放在D_SEG段中的SUM中。其中AUGEND、ADDEND和SUM均为双精度数，AUGEND赋值为99251，ADDEND赋值为－15962。

第五章　循环与分支程序设计

一般说来，编制一个汇编语言程序的步骤如下：

1）分析题意，确定算法。这一步是能否编制出高质量程序的关键，因此不应该一拿到题目就急于写程序，而是应该仔细地分析和理解题意，找出合理的算法及适当的数据结构。

2）根据算法画出程序框图。这一点对初学者特别重要，这样做可以减少出错的可能性。画框图时可以从粗到细把算法逐步地具体化。

3）根据框图编写程序。

4）上机调试程序。任何程序必须经过调试才能检查出你的设计思想是否正确以及你的程序是否符合你的设计思想。在调试程序的过程中应该善于利用机器提供的调试工具（如 DEBUG）来进行工作，你会发现它会给你提供很大的帮助。

第五、六两章将说明汇编语言程序设计方法，我们将举例说明上述程序设计步骤，但上机调试只能由读者自己实践了。

程序有顺序、循环、分支和子程序四种结构形式。顺序程序结构是指完全按顺序逐条执行的指令序列，这在程序段中是大量存在的，但作为完整的程序则很少见，我们不对它作专门讨论。本章说明循环与分支程序结构，第六章则专门讨论子程序结构。

5.1　循环程序设计

5.1.1　循环程序的结构形式

在 3.3.5.3 节有关循环指令的讨论中，我们已经给出了二个简单的循环程序的例子，现在我们来分析一下循环结构。循环程序可以有两种结构形式，如图 5.1 所示。一种是 DO_WHILE 结构形式；另一种是 DO_UNTIL 结构形式。DO_WHILE 结构把对循环控制条件的判断放在循环的入口，先判断条件，满足条件就执行循环体，否则则退出循环。DO_UNTIL 结构则先执行循环体然后再判断控制条件，不满足条件则继续执行循环操作，一旦满足条件则退出循环。这两种结构可以根据具体情况选择使用。一般说来，如果有循环次数等于 0 的可能则应选择 DO_WHILE 结构，否则则使用 DO_UNTIL 结构。不论哪一种结构形式，循环程序可由如下三部分组成：

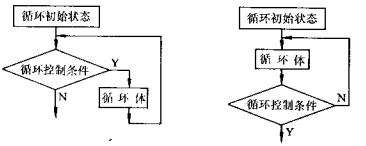


图 5.1　循环程序的结构形式
(1) DO_WHILE 结构　(2) DO_UNTIL 结构

1）设置循环的初始状态。如设置循环次数的计数值，以及为循环体正常工作而建立的初始状态等。

2）循环体。这是循环工作的主体，它由循环的工作部分及修改部分组成。循环的工作部分是为完成程序功能而设计的主要程序段，循环的修改部分则是为保证每一次重复（循环）时，参加执行的信息能发生有规律的变化而建立的程序段。

3）循环控制部分。循环控制本来应该属于循环体的一部分，由于它是循环程序设计的关键，所以要对它作专门的讨论。每个循环程序必须选择一个循环控制条件来控制循环的运行和结束，而合理的选择该控制条件就成为循环程序设计的关键问题。有时循环次数是已知的，此时可以用循环次数作为循环的控制条件，LOOP 指令使这种循环程序设计能很容易地实现。有时循环次数是已知的，但有可能使用其它特征或条件来使循环提前结束，LOOPZ 和 LOOPNZ 指令又是设计这种循环程序的工具。在 3.3.5.3 节中我们已给出的例子就说明了这两种情况。但是，有时循环次数是未知的，那就需要根据具体情况找出控制循环结束的条件。循环控制条件的选择是很灵活的，有时可能的选择方案不止一种，此时就应分析比较选择一种效率最高的方案来实现，下面我们用例子来说明。

5.1.2 循环程序设计方法

例 5.1 试编制一个程序把 BX 寄存器内的二进制数用十六进制数的形式在屏幕上显示出来。

根据题意我们应该把 BX 的内容从左到右每四位为一组在屏幕上显示出来，显然这可以用循环结构来完成，每次循环显示一个十六进制数位，因而循环次数是已知的，计数值为 4。循环体中则应包括从二进制到所显示字符的 ASCII 之间的转换以及每个字符的显示，后者可以使用 DOS 功能调用来实现。画出程序框图如图 5.2 所示。这里采用了循环移位的方法把所要显示的 4 位二进制数移到最右面，以便作数字到字符的转换工作。另外由于数字 0～9 的 ASCII 为 30～39H，而字母 A～F 的 ASCII 为 41～46H，所以在把 4 位二进制数加上 30H 后还需作一次判断，如果为字符 A～F，则还应加上 7 才能显示出正确的十六进制数。

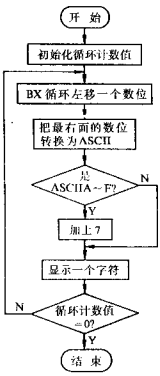


图 5.2 二进制到十六进制数转换的程序框图

我们以 BINIHEX. ASM 为文件名建立源文件如图 5.3 所示。在程序中没有使用 LOOP 指令，这是因为循环移位指令要使用 CL 寄存器，而 LOOP 指令要使用 CX 寄存器，为了解决 CX 寄存器的冲突问题，这里用 CH 寄存器存放循环计数值，而用 DEC 及 JNZ 两条指令完成 LOOP 指令的功能，这说明即使用计数值控制循环结束也不是非用 LOOP 指令不可，这里只是提供了另一种方法而已。当然也可以把计数值初始化为 0，用每循环一次加 1 然后比较次数是否达到要求的方法来实现。或者仍用 LOOP 指令，而用堆栈保存其中的一个信息（例如计数值），来解决 CX 寄存器的冲突问题等。总之，程序设计是很灵活的，只要算法和指令的使用没有错误都可以达到目的，但是怎样做才能使效率最高那是需要仔细斟酌的。

```
prognam segment                ;define code segment
main    proc far
        assume cs:prognam
```

```
start:                          ;starting execution addr
;set up stack for return
        push    ds              ;save old data segment
        sub     ax,ax           ;put zero in AX
        pash    ax              ;save zero on stack

;main part of program
        mov     ch,4            ;number of digits
rotate: mov     cl,4            ;set count to 4 bits
        rol     bx,cl           ;left digit to right
        mov     al,bl           ;  move to AL
        and     al,0fh          ;mask off left digit
        add     al,30h          ;convert hex to ASCII
        cmp     al,3ah          ;is it > 9 ?
        jl      printit         ;jump if digit = 0 to 9
        add     al,7h           ;digit is A to F
printit:
        mov     dl,al           ;put ASCII char in DL
        mov     ah,2            ;display output funct
        int     21h             ;call DOS
        dec     ch              ;done 4 digits?
        jnz     rotate          ;not yet
        ret                     ;return to DOS
main    endp                    ;end of main part of prog.

progmam ends                    ;end of segment

        end                     ;end of assembly
```

图 5.3　二进制到十六进制数转换程序

例 5.2　在 ADDR 单元中存放着数 Y 的地址，试编制一程序把 Y 中 1 的个数存入 COUNT 单元中。

要测出 Y 中 1 的个数就应逐位测试，一个比较简单的办法是可根据最高有效位是否为 1 来计数，然后用移位的方法把各位数逐次移到最高位去。循环的结束可以用计数值为 16 来控制，但更好的办法是结合上述方法可以用测试数是否为 0 来作为结束条件，这样可以在很多情况下缩短程序的执行时间。此外考虑到 Y 本身为 0 的可能性应该采用 DO_WHILE 的结构形式。根据以上考虑可画出图 5.4 的程序框图。编制程序如图 5.5 所示。

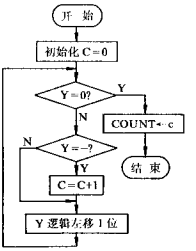


图 5.4　数 1 的程序框图

这个例子说明算法和循环控制条件的选择对程序的工作效率有很大的影响，而循环控制条件的选择又是很灵活的，应该根据具体情况来确定。

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * *
datarea segment                    ;define data segment
addr    dw          number
number dw           Y
count   dw          ?
datarea ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * *
prognam segment                    ;define code segment
;-------------------------------------------------------
main    proc    far                ;main part of program
        assume cs:prognam,ds:datarea
start:                             ;starting execution address
;set up stack for return
        push    ds                 ;save old data segment
        sub     ax,ax              ;put zero in AX
        push    ax                 ;save it on stack
;set DS register to current data segment
        mov     ax,datarea         ;datarea segment addr
        mov     ds,ax              ;  into DS register
;MAIN  PART  OF  PROGRAM  GOES  HERE
        mov     cx,0               ;initialize C to 0
        mov     bx, addr
        mov     ax,[bx]            ;put Y in AX
repeat: test    ax,0ffffh          ;test Y
        jz      exit               ;if Y=0,get exit
        jns     shift              ;if MSB=0,C unchanged
        inc     cx                 ;  else,C=C+1
shift:  shl     ax,1               ;shift Y one bit left
        jmp     repeat             ;repeat
exit:   mov     count,cx
        ret                        ;return to DOS
main    endp                       ;end of main part of prog.
;-------------------------------------------------------
prognam ends                       ;end of code segment
;* * * * * * * * * * * * * * * * * * * * * * * * * * * *
        end     start              ;end assembly
```

图 5.6　数 1 的程序

例 5.3　在附加段中有一个首地址为 LIST 和未经排序的字数组，在数组的第一个字中存放着该数组的长度，数组的首地址已存放在 DI 寄存器中。AX 寄存器中存放着一个数。要求编制一程序：在数组中查找该数，如果找到此数则把它从数组中删除。

这一程序应该首先查找数组中是否有(AX)，如果没有则不对数组作任何处理就结束程

序。如果找到这一元素则应把数组中位于其前(指地址比该元素高)的元素后移一个字(即向低地址方向移动),并修改数组长度值。如果找到的元素正好位于数组末尾,则不必移动任何元素,只要修改数组长度值就可以了。这里第一部分的查找元素可以使用串处理指令,第二部分的删除元素则可使用循环结构,由于查找结束时就可以知道该元素的位置,因此可以作为循环次数已知的情况来设计。画出程序框图如图 5.6 所示。程序的主体部分如图 5.7 所示。

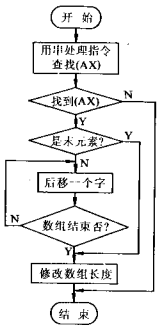


图 5.6 删除数组中一元素的程序框图

```
;Delete the value in AX from an unordered list in the extra
;   segment,if that value is in the list.
;Inputs: (DI)=starting address of the list.
;        First location of the list=Length of list(words)
del_ul  proc    far
        cld                     ;make DF= 0,to scan forward
        push    di              ;save starting address
        mov     cx,es:[di]      ;fetch element count
        add     di,2            ;make DI point to 1st data el.
        repne   scasw           ;value in the list?
        je      delete          ;if so,go delete it
        pop     di              ;otherwise,exit
        jmp     short exit
;The following instruction delete an element from the
;   list,as follows:
;     (1) If the element lies at the end of the list,delete
;         it by decreasing the element count by 1.
```

```
;       (2) Otherwise,delete the element by moving all
;           subsequent elements up by one position.
delete:  jcxz    dec_cnt          ;if (CX)=0,delete last el.
next_el: mov     bx,es:[di]       ;move one el. up in list
         mov     es:[di-2],bx
         add     di,2             ;point to next el.
         loop    next_el          ;repeat until all el. moved
dec_cnt: pop     di               ;decrease el. count by 1
         dec     word ptr es:[di]
exit:    ret                      ;exit
del_ul   endp                     ;end of main part of prog.
```

图 5.7 删除数组中一元素程序

例 5.4 将正数 N 插入一个已整序的字数组的正确位置。该数组的首地址和末地址分别为 ARRAY_HEAD 和 ARRAY_END，其中所有数均为正数且已按递增的次序排列。

由于数组的首地址和末地址都是已知的，因此数组长度是可以确定的。但是这里只要求插入一个数，并不一定要扫描整个数组，所以可以用找到应插入数的位置作为循环的结束条件。此外，为空出要插入数的位置，其前的全部元素，都应前移一个字（即向地址增大的方向移动一个字，这里的前后是指程序运行的方向为前，反之则为后），所以算法上应该从数组的尾部向头部查找，可逐字取出数组中的一个数 K 与 N 作比较，如 K>N，则把 K 前移一个字，然后继续往后查找；如 K≤N，则把 N 插在 K 之前就可以结束程序了。

在考虑算法时，必须把可能出现的边界情况考虑在内。如例 5.3 中对有可能出现要删除的元素正好在数组的末尾的考虑。这个问题是初学者易于忽略的，应该引起注意。在例 5.4 中，应该考虑 N 大于或小于数组中所有数的两种可能性。如果 N 大于数组中所有数，则第一次比较就可以结束循环，也就是说循环次数有可能等于 0，所以应该选用 DO_WHILE 结构形式。如果 N 小于数组中所有数，则必须使循环及时结束，也就是说不允许查找的范围超过数组的首地址，这当然可以把数组的首地址也同时作为结束条件来考虑，或者同时用循环次数作为结束条件之一来考虑。本例的更好办法是：可以利用所有数均为正数的条件，在 ARRAY_HEAD－2 单元中存放'－1'这个数，这样可以保证如果数 N 小于数组中的所有数，那它必然大于－1，这样就可以正确地把 N 放在数组之首了，这样循环的结束仍然可以使用 K>N 这一条件。根据上述有关算法的考虑可画出程序框图如图 5.8 所示。编制程序如图 5.9 所示。

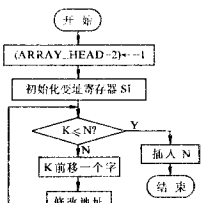


图 5.8 数组中插入一元素的程序框图

```
;*********************************************
datarea segment                  ;define data segment
  x          dw       ?
  array_head dw       3,5,15,23,37,49,52,65,78,99
  array_end  dw       105
```

```
n               dw      32
datarea ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
prognam segment                         ;define code segment
;-----------------------------------------------------------
main    proc    far                     ;main part of program
        assume cs:prognam,ds:datarea
start:                                  ;starting execution address
;set up stack for return
        push    ds                      ;save old data segment
        sub     ax,ax                   ;put zero in AX
        push    ax                      ;save it on stack
;set DS register to current data segment
        mov     ax,datarea              ;datarea segment addr
        mov     ds,ax                   ;  into DS register
;MAIN  PART  OF  PROGRAM  GOES  HERE
        mov     ax,n
        mov     array_head-2,0ffffh
        mov     si,0
compare: cmp    array_end[si],ax
        jle     insert
        mov     bx,array_end[si]
        mov     array_end[si+2],bx
        sub     si,2
        jmp     short  compare
insert: mov     array_end[si+2],ax
        ret                             ;return to DOS
main    endp                            ;end of main part of program
;-----------------------------------------------------------
prognam ends                            ;end of code segment
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        end     start                   ;end assembly
```

图 5.9　数组中插入一元素程序

上述例子说明，循环控制条件是循环程序设计的关键，必须结合对算法的分析和考虑合理地选择。同时必须仔细地考虑边界情况出现的可能性，以免在特殊情况下造成错误。

例 5.5　设有数组 X 和 Y。X 数组中有 $X_1,\cdots,X_{10}$；Y 数组中有 $Y_1,\cdots,Y_{10}$。试编制程序计算

$Z_1=X_1+Y_1$　　$Z_5=X_5-Y_5$　　$Z_8=X_8-Y_8$

$Z_2=X_2+Y_2$　　$Z_6=X_6+Y_6$　　$Z_9=X_9+Y_9$

$Z_3=X_3-Y_3$　　$Z_7=X_7-Y_7$　　$Z_{10}=X_{10}+Y_{10}$

$Z_4=X_4-Y_4$

结果存入 Z 数组。

对于这种问题，我们也可用循环程序结构来完成。已知循环计数值为 10，每次循环的操作数是可以顺序取出的，但所做的操作却有不同，这里有两种操作：加法和减法，为了区别每次应该做哪一种操作，可以设立标志位，如标志位为 0 做加法；为 1 则做减法，这样进入循环后只要判别标志位就可确定应该做的操作了。显然，这里要做 10 次操作就应该设立 10 个标志位，我们把它放在一个存储单元 LOGIC_RULE 中，这种存储单元一般称为逻辑尺，本例设定的逻辑尺为：

0000000011011100

从低位开始所设的标志位反映了每次要做的操作顺序，最高的 6 位没有意义把它们设为 0。可以画出程序框图如图 5.10 所示。编制程序如图 5.11 所示。

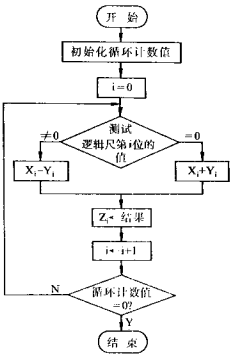


图 5.10 例 5.5 的程序框图

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
datarea segment                    ;define data segment
  x          dw          x1,x2,x3,x4,x5,x6,x7,x8,x9,x10
  y          dw          y1,y2,y3,y4,y5,y6,y7,y8,y9,y10
  z          dw          z1,z2,z3,z4,z5,z6,z7,z8,z9,z10
  logic_rule dw          00dch
datarea ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
prognam segment                    ;define code segment
;-------------------------------------------------------------------------
main    proc    far                ;main part of program
```

```
        assume cs:prognam,ds:datarea

start:                                  ;starting execution address

;set up stack for return
        push    ds                      ;save old data segment
        sub     ax,ax                   ;put zero in AX
        push    ax                      ;save it on stack

;set DS register to current data segment
        mov     ax,datarea              ;datarea segment addr
        mov     ds,ax                   ;  into DS register

;MAIN  PART  OF  PROGRAM  GOES  HERE
        mov     bx,0
        mov     cx,10
        mov     dx,logic rule
next:   mov     ax,x[bx]
        shr     dx,1
        jc      subtract
        add     ax,y[bx]
        jmp     short result
subtract:
        sub     ax,y[bx]
result: mov     z[bx],ax
        add     bx,2
        loop    next
        ret                             ;return to DOS
main    endp                            ;end of main part of program
;-------------------------------------------------------------------
prognam ends                            ;end of code segment
;*********************************************
        end     start                   ;end assembly
```

图 5.11　例 5.5 的程序实现

这种设立逻辑尺的方法是很常用的。例如在矩阵运算中为了跳过操作数为 0 的计算，经常采用这种方法。又如，把一组数据存入存储器时，如果其中数值为 0 的元素很多，也可用这种方法设立一个每位表示一个下标的逻辑尺，(这样的逻辑尺可能占有几个字，由数组的长度确定。)0 元素就可不占有存储单元了。在例 5.5 中，每个标志只占一位，如果要表示的特征数更多，则每个标志可占有几位，在处理方法上是完全相同的。

设立标志位的方法，除了如逻辑尺那样可静态地预置外，还可以在程序中动态地修改标志位的值以达到控制的目的，下例说明这种方法。

例 5.6　试编制一程序：从键盘输入一行字符，要求第一个键入的字符必须是空格符，如不是则退出程序；如是则开始接收键入的字符并顺序存放在首地址为 BUFFER 的缓冲区中(空格符不存入)，直到接收到第二个空格符时退出程序。

这一程序要求接收的字符从空格符开始又以空格符结束，因此程序中必须区分所接收的

字符是否是第一个字符，为此设立作为标志的存储单元 FLAG，一开始将其置为 0，接收第一个字符后可将其置 1。整个程序的框图如图 5.12 所示。图 5.13 则为其程序实现。

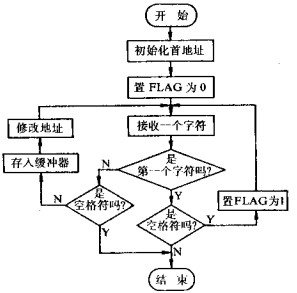


图 5.12 例 5.6 的程序框图

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
datarea segment                          ;define data segment
  buffer          db        80 dup  (?)
  flag            db        ?
datarea ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
prognam segment                          ;define code segment
;-----------------------------------------------------------------
main      proc      far                  ;main part of program
          assume cs:prognam,ds:datarea
start:                                   ;starting execution address
;set up stack for return
          push      ds                   ;save old data segment
          sub       ax,ax                ;put zero in AX
          push      ax                   ;save it on stack
;set DS register to current data segment
          mov       ax,datarea           ;datarea segment addr
          mov       ds,ax                ;  into DS register
;MAIN  PART  OF  PROGRAM  GOES  HERE
          lea       bx,buffer
          mov       flag,0
```

```
next:   mov     ah,01
        int     21h
        test    flag,01h
        jnz     follow
        cmp     al,20h
        jnz     exit
        mov     flag,1
        jmp     next
follow: cmp     al,20h
        jz      exit
        mov     [bx],al
        inc     bx
        jmp     next
exit:   ret                             ;return to DOS
main    endp                            ;end of main part of program
;-------------------------------------------------------------
prognam ends                            ;end of code segment
;*********************************************
        end     start                   ;end assembly
```

图 5.13　例 5.6 的程序实现

5.1.3　多重循环程序设计

循环可以有多重结构。多重循环程序设计的基本方法和单重循环程序设计是一致的，应分别考虑各重循环的控制条件及其程序实现，相互之间不能混淆。另外应该注意在每次通过外层循环再次进入内层循环时，初始条件必须重新设置。下面，我们用例子来说明。

例 5.7　有一个首地址为 A 的 N 字数组，请编制程序使该数组中的数按照从大到小的次序整序。

我们采用起泡排序算法从第一个数开始依次对相邻两个数进行比较，如次序对则不做任何操作；如次序不对则使这两个数交换位置。表 5.1 表示了这种算法的例子，可以看出，在做了第一遍的(N－1)次比较后，最小的数已经放到了最后，所以第二遍比较只需要考虑(N－1)个数，即只需要比较(N－2)次，第三遍则只需要做(N－3)次比较……总共最多(N－1)遍比较就可以完成排序。图 5.14 表示了起泡排序算法的程序框图。图 5.15 则为其程序实现。

表 5.1　起泡排序算法举例

序号	数	比较遍数		
		1	2	3
1	8	8	16	84
2	5	16	84	32
3	16	84	32	16
4	84	32	8	8
5	32	5	5	5

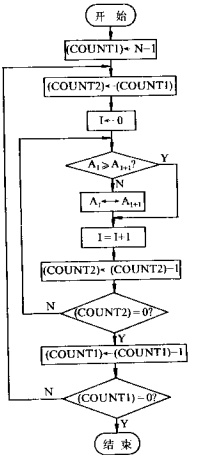


图 5.14 起泡排序算法的程序框图之一

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
datarea segment                          ;define data segment
  a      dw          n  dup  (?)
datarea ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
prognam segment                          ;define code segment
;-------------------------------------------------------------------
main     proc        far                 ;main part of program
         assume cs:prognam,ds:datarea
start:                                   ;starting execution address
;set up stack for return
         push        ds                  ;save old data segment
         sub         ax,ax               ;put zero in AX
         push        ax                  ;save it on stack
;set DS register to current data segment
```

```
        mov     ax,datarea          ;datarea segment addr
        mov     ds,ax               ;  into DS register
;MAIN  PART  OF  PROGRAM  GOES  HERE
        mov     cx,n                ;set count1
        dec     cx                  ;  to n-1
loop1:  mov     di,cx               ;save count1 in DI
        mov     bx,0                ;clear BX
loop2:  mov     ax,a[bx]            ;load a(i) into AX and
        cmp     ax,a[bx+2]          ;  compare with a(i+1)
        jge     cotinue             ;swap if
        xchg    ax,a[bx+2]          ;  a(i) < a(i+1) and
        mov     a[bx],ax            ;store greater number
cotinue: add    bx,2                ;increment index
        loop    loop2               ;if not the end of a pass,
                                    ;repeat
        mov     cx,di               ;restore count1
                                    ;  for the either loop
        loop    loop1               ;if not the final pass
                                    ;repeat
        ret                         ;return to DOS
main    endp                        ;end of main part of program
;-------------------------------------------------
prognam ends                        ;end of code segment
;*********************************************
        end     start               ;end assembly
```

图 5.15　起泡排序算法的程序实现之一

例 5.8　在附加段中有一个字数组，其首地址已存放在 DI 寄存器中，在数组的第一个字中存放着该数组的长度。要求编制一个程序使该数组中的数按照从小到大的次序排列整齐。

这里当然也可以采用起泡排序算法，但是在例 5.7 中，内、外循环的次数都是已知的，在整个程序运行过程中，内循环次数虽然是在不断的变化着，但它是按每次减 1 的规律在变化的，而外循环次数则是一个由数组长度确定的数(其值为 N－1)。也就是说，不管数组的原始排列情况如何，算法保证只要做 N－1 遍比较总可以达到排序的目的。显然，在很多情况下，数组的比较遍数并未达到(N－1)就已经整序完毕，但程序还必须继续运行到(N－1)遍才能结束。为了提高效率，我们可以采用另一种结束外循环的办法：设立一个交换标志位，每次进入外循环就将交换标志置 1，在内循环中每做一次交换操作就将该标志位置 0，在每次内循环结束后可以测试交换标志位，如该位为 0 则再一次进入外循环；如该位为 1，则说明上一遍比较已经未引起交换操作，数组已整序完毕，这样就可以立即结束外循环了。当然，这种方法在数组已整序完毕后会多做一遍比较，但在多数情况下，其比较遍数会少于(N－1)，因而算法效率较高。这种算法的程序框图如图 5.16 所示。程序则表示于图 5.17。

例 5.7 和例 5.8 的情况再次说明算法以及循环控制条件的选择对程序效率的影响。另外，例 5.8 中关于设立测试标志的方法和例 5.6 类似是循环程序设计中常用的一种方法。总之，循

环控制条件的选择是很灵活的，读者应该根据具体情况合理选择。

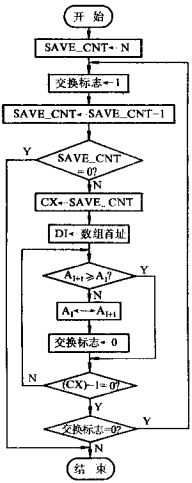


图 5.16 起泡排序算法的程序框图之二

```
;Arranges the 16-bit elements of a list in the extra segment
;   in ascending order, using bubble sort.
;Inputs: ES:DI=Starting address of the list.
;   First location of the list=Length of the list (words)
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
data            segment                    ;define data segment
  save_cnt          dw          ?
  start_addr        dw          ?
data            ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
prognam segment                            ;define code segment
;-------------------------------------------------------------
bubble  proc    far
```

```
        assume cs:progam,ds:data

        push    ds                      ;save caller's registers
        push    cx
        push    ax
        push    bx
        mov     ax,data                 ;initialize DS
        mov     ds,ax
;
        mov     start_addr,di           ;save starting address
                                        ;   in memory
        mov     cx,es:[di]              ;fetch element count N
        mov     save_cnt,cx             ;   save this value in memory
init:   mov     bx,1                    ;exchange flag (BX)=1
        dec     save_cnt                ;get ready for
                                        ;   count-1 compares
        jz      sorted                  ;exit if save_cnt is 0
        mov     cx,save_cnt             ;and load this count
                                        ;   into CX
        mov     di,start_addr           ;load start address
                                        ;   into DI
next:   add     di,2                    ;address a data el.
        mov     ax,es:[di]              ;   and load it into AX
        cmp     es:[di+2],ax            ;is next el. < this el. ?
        jae     cont                    ;no,go check next pair
        xchg    es:[di+2],ax            ;yes,exchange
        mov     es:[di],ax              ;   these elements
        sub     bx,bx                   ;and make exchange flag 0
cont:   loop    next                    ;process entire list
        cmp     bx,0                    ;any exchanges made?
        je      init                    ;if so,process list again
sorted: mov     di,start_addr           ;if not so,
        pop     bx                      ;   restore registers
        pop     ax
        pop     cx
        pop     ds
        ret                             ;exit

bubble  endp
;-----------------------------------------
progam ends                             ;end of code segment
;*************************************
        end                             ;end assembly
```

图 5.17　起泡排序算法的程序实现之二

5.2 分支程序设计

5.2.1 分支程序的结构形式

分支程序结构可以有两种形式，如图 5.18 所示。它们分别相当于高级语言中的 IF_THEN_ELSE 语句和 CASE 语句，它们适用于要根据不同条件作不同处理的情况。IF_THEN_ELSE 语句可以引出两个分支，CASE 语句则可以引出多个分支，不论哪一种形式，它们的共同特点是：运行方向是向前的，在某一种确定条件下，只能执行多个分支中的一个分支。

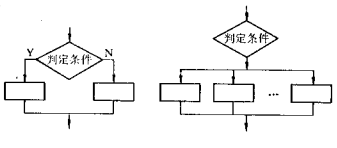


图 5.18 分支程序的结构形式
(1) IF_THEN_ELSE 结构 (2) CASE 结构

5.2.2 分支程序设计方法

程序的分支一般用条件转移指令来产生，在 3.3 节的例 3.66 中区分正数、0 和负数的程序就是分支程序，可以看出，这里利用转移指令不影响条件码的特性，连续地使用条件转移指令使程序产生了三个不同的分支(尽管负数的分支并未执行任何操作)，而对于数组中的每一个数，它只能是三个分支中的某一个。下面我们再来看一个例子。

例 5.9 在附加段中，有一个按从小到大顺序排列的无符号数的数组，其首地址存放在 DI 寄存器中，数组中的第一个单元存放着数组长度。在 AX 中有一个无符号数，要求在数组中查找(AX)，如找到则使 CF=0，并在 SI 中给出该元素在数组中的偏移地址；如未找到则使 CF=1。

我们已经遇到过多个表格查找的例子，都是使用顺序查找的方法，本例是一个已经排序的数组，可以采用折半查找法以提高查找效率。折半查找法先取有序数组的中间元素与查找值进行比较，如相等则查找成功；如查找值大于中间元素，则再取高半部的中间元素与查找值相比较；如查找值小于中间元素，则再取低半部的中间元素与查找值相比较；如此重复直到查找成功或最终未找到该数(查找不成功)为止。折半查找法的效率高于顺序查找法，对于长度为 N 的表格，顺序查找法平均要作 N/2 次比较，而折半查找法的平均比较次数约为 $\log_2 N$。所以，如果数组长度为 100，则顺序查找法平均要做 50 次比较，而折半查找法平均做 7 次比较就可以了。

图 5.19 表示了折半查找算法的程序框图。给出的程序首先把查找值与数组的第一个元素和最后一个元素相比较，如果找到或者该数小于第一个元素或大于最后一个元素则结束查找，否则从 SEARCH 开始折半查找。SEARCH 以前的工作在图 5.19 中未表示出来。从 SEARCH 开始，首先把数组长度作为数组中间元素的下标，把它从数组的第一个单元中取出来并使它成为偶数，然后把下标加到 DI 中以形成数组的中间元素的地址，并开始比较查找。如果比较相等则转至 ALL_DONE 结束查找，否则要确定下一步的查找是在数组的低半部还是高半部中进

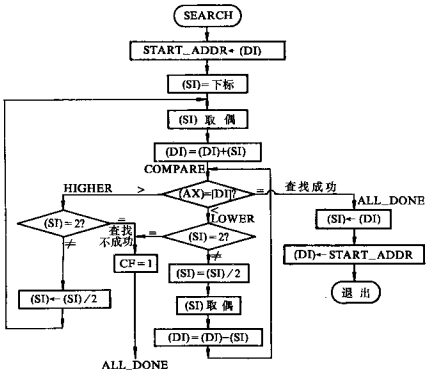


图 5.19　折半查找算法的程序框图

行。它们所要做的工作是：首先检查下标是否等于 2，如等于 2 则说明整个查找失败，应把 CF 置 1 并转至 ALL_DONE 结束查找工作；其次，把下标除以 2 以便作进一步的折半查找，然后使下标形成偶数值；如果要在低半部查找，则从当前的地址(DI)中减去下标值(SI)以形成新的查找地址；如果要在高半部查找则把下标值加到当前地址值中。以上过程重复进行直到下标值减到 2 或者查找值找到为止。编制程序如图 5.20 所示。

```
;Search an ordered list in the extra segment for the word
;   value contained in AX
;Inputs; ES;DI=starting address of the list
;         First location=Length of list(words)
;Results;If the value is in the list,
;           CF=0
;           SI=Offset of matching element.
;         If thr value is not in the list,
;           CF=1
;      SI=Offset of last element compared.
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
dseg        segment                         ;define data segment
  start_addr        dw          ?
dseg        ends
```

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
cseg    segment                         ;define code segment
;-----------------------------------------------------------
b_search        proc    far

        assume cs:cseg,ds:dseg

        push    ds                      ;save caller's DS
        push    ax
        mov     ax,dseg                 ;initialize DS
        mov     ds,ax
        pop     ax

;Find out if AX lies beyond the boundaries of the list.
        cmp     ax,es:[di+2]            ;search value<or=first el. ?
        ja      chk_last                ;no,go check last el.
        lea     si,es:[di+2]            ;yes,fetch addr of first el.
        je      exit                    ;if value = 1st el. ,exit
        stc                             ;if value < 1st el. ,set CF
        jmp     exit                    ;   and then exit
chk_last:
        mov     si,es:[di]              ;point to last el.
        shl     si,1
        add     si,di
        cmp     ax,es:[si]              ;search value>or=last el. ?
        jb      search                  ;no, go search list
        je      exit                    ;yes, exit if value =last el.
        stc                             ;if value >last el. ,set CF
        jmp     exit                    ;   and then exit
; Search for value within the list.
search:  mov    start_addr,di           ;save starting addr in memory
        mov     si,es:[di]              ;fetch index
even_idx:
        test    si,1                    ;force index to an even value
        jz      add_idx
        inc     si
add_idx:  add   di,si                   ;calculate next search addr
compare:  cmp   ax,es:[di]              ;search value found?
        je      all_done                ;if so,exit
        ja      higher                  ;otherwise, find correct half
; These instructions are executed if the search value is
;   lower in the list.
        cmp     si,2                    ;index = 2?
        jne     idx_ok
no_match:
        stc                             ;if so, set CF
```

```
                je          all_done            ;  and exit
idx_ok:         shr         si,1                ;if not, divide index by 2
                test        si,1                ;force index to
                                                ;  an even value
                jz          sub_idx
                inc         si
sub_idx:        sub         di,si               ;calaulate next addr
                jmp         short compare       ;go check this el.
; These instructions are executed if the search value is
;  higher in the list.
higher:         cmp         si,2                ;index = 2 ?
                je          no_match            ;if so, go set CF and exit
                shr         si,1                ;if not, divide index by 2
                jmp         short even_idx      ;go check next el.
; Following are exit instructions.
all_done:
                mov         si,di               ; move compare addr into SI
                mov         di,start_addr       ;restore starting addr
exit:           pop         ds
                ret                             ;  and exit

b_search        endp
;-------------------------------------------------------
cseg            ends                            ;end of code segment
;*********************************************
                end                             ;end assembly
```

图 5.20　折半查找算法的程序实现

如果数组如下：

LIST　DW　12,11,22,33,44,55,66,77,88,99,111,222,333 要求查找的数为(AX)=55。数组长度为 12,第一次比较的是数组的第 6 个元素 66;因 55<66,所以第二次用低半部折半查找,比较的是第 3 个元素 33;因 55>33,所以第三次用高半部折半查找,比较的是第 5 个元素 55。这样经过三次比较后因查找成功而退出程序。

如果要查找的数是(AX)=57。则第一次比较的仍是第 6 个元素 66;因 57<66,所以第二次用低半部折半查找,比较的是第 3 个元素 33;因 57>33,所以第三次用高半部折半查找,比较的是第 5 个元素 55;因 55<57,所以第四次用高半部折半查找,比较的又是第 6 个元素 66,因 57<66,在转低半部折半查找时,因(SI)=2 而以查找失败退出程序。

可以看出,在这个例子中同样用 CMP 或 TEST 指令以及条件转移指令产生二个或多个程序分支。当然,由于多数运算型指令置条件码,所以在条件转移指令之前并不一定要使用 CMP 或 TEST 指令,只要保证使用条件转移指令时的条件码符合要求就可以了。在循环结构中的例 5.5 中建立了逻辑尺,根据对逻辑尺中每一位的不同值的判断产生不同的程序分支,也是分支程序的一种实现方法。以上多个例子都 是既有分支结构又包括循环结构,实际上,多数程序都是各种程序结构的组合。而且,循环结构可以看作分支结构的一种特例,它只是多次走一个分

支，而在满足循环结束条件时走另一个分支罢了。

5.2.3 跳跃表法

分支程序的两种结构形式都可以用上面所述的方法来实现。此外，在实现 CASE 结构时还可以使用跳跃表法使程序能根据不同的条件转移到多个程序分支去，下面举例说明。

例 5.10 试根据 AL 寄存器中哪一位为 1(从低位到高位)把程序转移到 8 个不同的程序分支去。

图 5.21～5.23 分别列出了三种方法编制的程序。其实，这三种方法并无实质的区别，只是其中关键的 JMP 指令所用的寻址方式不同而已。

跳跃表法是一种很有用的分支程序设计方法，希望读者能通过上述例子掌握要领，灵活使用。

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
branch_addresses segment                    ;define data segment
  branch_table  dw      routine_1
                dw      routine_2
                dw      routine_3
                dw      routine_4
                dw      routine_5
                dw      routine_6
                dw      routine_7
                dw      routine_8
branch_addresses ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
procedure_select segment                    ;define code segment
;-----------------------------------------------------------------
main    proc    far                         ;main part of program
        assume cs:procedure_select,ds:branch_addresses
start:                                      ;starting execution address
;set up stack for return
        push    ds                          ;save old data segment
        sub     bx,bx                       ;put zero in BX
        push    bx                          ;  save it on stack
;set DS register to current data segment
        mov     bx,branch_addresses         ;data segment addr
        mov     ds,bx                       ;       into DS register
;MAIN  PART  OF  PROGRAM  GOES  HERE
        cmp     al,0                        ;this test assures that some
                                            ;  bit of AL has been set by
        je      continue_main_line          ;  earlier instruction to
                                            ;       specify a routine
        lea     bx,branch_table             ;BX set to location holding
                                            ;  address of first routine
```

```
l:      shr     al,1;           puts least-significant bit
                                ;   of AL into the CF
        jnb     not_yet         ;if CF=0, the on bit in AL
                                ;   has not yet been found
        jmp     word ptr [bx]   ;if CF=1,then control is
                                ;   transferred
not_yet: add    bx,type branch_table ;if no transfer, then
                                ; the bit that is on has not
                                ;   yet been found,so BX is
                                ;     set to point to the
                                ;       next entry in the
                                ;         address table by
                                ;           adding 2
        jmp     l               ;jump to l to shift
                                ;   and retest
continue_main_line:             ;we reach here only if no
                                ;   bit was set to indicate
                                ;     a desired routine
routine1:
routine2:
        ret
main    endp                    ;end of main part of program
;-------------------------------------------------
procedure_select ends           ;end of code segment
;*********************************
        end     start           ;end assembly
```

图 5.21　用寄存器间接寻址方式实现跳跃表法

```
;*********************************
branch_addresses segment        ;define data segment
  branch_table  dw      routine_1
                dw      routine_2
                dw      routine_3
                dw      routine_4
                dw      routine_5
                dw      routine_6
                dw      routine_7
                dw      routine_8
branch_addresses ends
;*********************************
procedure_select segment        ;define code segment
;-------------------------------------------------
main    proc    far             ;main part of program
```

```
        assume cs:procedure—select,ds:branch_addresses

start:                                  ;starting execution address;set up stack for return
        push    ds                      ;save old data segment
        sub     bx,bx                   ;put zero in BX
        push    bx                      ;  save it on stack
;set DS register to current data segment
        mov     bx,branch_addresses     ;data segment addr
        mov     ds,bx                   ;       into DS register
;MAIN  PART  OF  PROGRAM  GOES  HERE
        cmp     al,0
        je      continue_main—line
        lea     bx,branch_table         ;BX set to location holding
                                        ;  address of first routine
        mov     si,7 * type branch_table ;points initially to
                                        ;  last such entry in list
        mov     cx,8                    ;loop counter allowing
                                        ;  shift maximum
1:      shl     al,1                    ;shifts high—order AL bit
                                        ;  into CF
        jnb     not_yet                 ;if CF=0,routine represented
                                        ;  by that bit not desired
        jmp     word ptr [bx][si]       ;if CF=1,transfer to
                                        ;  procedure represented by
                                        ;    most recent bit tested
not_yet: sub    si,type branch_table    ;adjust index register
                                        ;  to point to "next"
                                        ;    branch address
        loop    1
continue_main_line:
                ⋮
routine1:       ⋮
routine2:       ⋮
        ret
main    endp                            ;end of main part of program
;————————————————————————————————————————
procedure_select ends                   ;end of code segment
;* * * * * * * * * * * * * * * * * * * * * * * * * * * *
        end     start                   ;end assembly
```

图 5.22　用基址变址寻址方式实现跳跃表法

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * *
branch_addresses segment                ;define data segment
```

```
branch_table  dw      routine_1
              dw      routine_2
              dw      routine_3
              dw      routine_4
              dw      routine_5
              dw      routine_6
              dw      routine_7
              dw      routine_8
branch_addresses ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
procedure_select segment                ;define code segment
;-----------------------------------------------------------------
main    proc    far                     ;main part of program
        assume cs:procedure_select,ds:branch_addresses
start:                                  ;starting execution address
;set up stack for return
        push    ds                      ;save old data segment
        sub     bx,bx                   ;put zero in BX
        push    bx                      ;save it on stack
;set DS register to current data segment
        mov     bx,branch_addresses     ;data segment addr
        mov     ds,bx                   ;       into DS register
;MAIN   PART   OF   PROGRAM   GOES   HERE
        cmp     al,0
        je      continue_main_line
        mov     si,0
1:      shr     al,1                    ;puts least significant bit
                                        ;   of AL into CF
        jnb     not_yet                 ;if CF=0, the on bit in AL
                                        ;   has not yet been found
        jmp     branch_table[si]        ;if CF=1,then control
                                        ;   is transferred
not_yet:add     si,type branch_table    ;if no transfer, then
                                        ;   the bit that is on has
                                        ;     not yet been found,so
                                        ;       SI is set to point to
                                        ;         the next entry in
                                        ;           the address table
                                        ;             by adding 2
        jmp     1                       ;jump to 1 to shift
                                        ;   and retest
continue_main_line:                     ;we reach here only if no
                ⋮                       ;   bit was set to indicate
```

```
                                        ;      a desired routine
routine1;      *
                *
routine2;      ⋮
        ret
main    endp                            ;end of main part of program
;-------------------------------------------------------------
procedure_select ends                   ;end of code segment
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        end     start                   ;end assembly
```

图 5.23　用变址寻址方式实现跳跃表法

习　题

5.1　试编写一个汇编语言程序，要求对键盘输入的小写字母用大写字母显示出来。

5.2　编写程序，从键盘接收一个小写字母，然后找出它的前导字符和后续字符，再按顺序显示这三个字符。

5.3　将 AX 寄存器中的16位数分成四组，每组4位，然后把这四组数分别放在 AL、BL、CL 和 DL 中。

5.4　试编写一程序，要求比较两个字符串 STRING1和 STRING2所含字符是否完全相同，若相同则显示'MATCH'，若不相同则显示'NO MATCH'。

5.5　试编写一个程序，要求能从键盘接收一个个位数 N，然后响铃 N 次(响铃的 ASCII 码为07)。

5.6　编写程序，将一个包含有20个数据的数组 M 分成两个数组：正数数组 P 和负数数组 N，并分别把这两个数组中数据的个数显示出来。

5.7　试编制一个汇编语言程序求出首地址为 DATA 的100D 字数组中的最小偶数，并把它存放在 AX 中。

5.8　把 AX 中存放的16位二进制数 K 看作是8个二进制的"四分之一字节"。试编写程序要求数一下值为3(即11B)的四分之一字节数，并将该数在终端上显示出来。

5.9　试编写一个汇编语言程序，要求从键盘接收一个四位的16进制数，并在终端上显示与它等值的二进制数。

5.10　设有一段英文，其字符变量名为 ENG，并以 $ 字符结束。试编写一程序查对单词 SUN 在该文中的出现次数，并以格式"SUN ××××"显示出次数。

5.11　从键盘输入一系列以 $ 为结束符的字符串，然后对其中的非数字字符计数，并显示出计数结果。

5.12　有一个首地址为 MEM 的100D 字数组，试编制程序删除数组中所有为零的项，并将后续项向前压缩，最后将数组的剩余部分补上零。

5.13　在 STRING 到 STRING＋99单元中存放着一个字符串，试编制一个程序测试该字符串中是否存在数字，如有则把 CL 的第5位置1，否则将该位置0。

5.14　在首地址为 TABLE 的数组中按递增次序存放着100H 个16位补码数，试编写一个程序把出现次数最多的数及其出现次数分别存放于 AX 和 CX 中。

5.15　数据段中已定义了一个有 n 个字数据的数组 M，试编写一程序求出 M 中绝对值最大的数，把它放在数据段的 M＋2n 单元中，并将该数的偏移地址存放在 M＋2(n＋1)单元中。

5.16　在首地址为 DATA 的字数组中存放了100H 个16位补码数，试编写一程序求出它们的平均值放在 AX 寄存器中；并求出数组中有多少个数小于此平均值，将结果放在 BX 寄存器中。

5.17　试编制一个程序把 AX 中的16进制数转换为 ASCII 码，并将对应的 ASCII 码依次存放到 MEM 数组中的四个字节中。例如，当(AX)＝2A49H 时，程序执行完后，MEM 中的4个字节内容为39H，34H，41H，32H。

5.18　把0～100D 之间的30个数存入以 GRADE 为首地址的30字数组中，GRADE＋i 表示学号为 i＋1的学

生的成绩。另一个数组 RANK 为30个学生的名次表，其中 RANK+1 的内容是学号为 i+1的学生的名次。编写程序，根据 GRADE 中的学生成绩，将学生名次填入 RANK 数组中。(提示：一个学生的名次等于成绩高于这个学生的人数加1)。

5.19 已知数组 A 包含15个互不相等的整数，数组 B 包含20个互不相等的整数。试编制一程序把既在 A 中又在 B 中出现的整数存放于数组 C 中。

5.20 设在 A、B 和 C 单元中分别存放着三个数。若三个数都不是0，则求出三数之和存放在 D 单元中；若其中有一个数为0，则把其它两单元也清零。请编写此程序。

5.21 试编写一程序，要求比较数组 ARRAY 中的三个16位补码数，并根据比较结果在终端上显示如下信息：

(1) 如果三个数都不相等则显示0；

(2) 如果三个数有二个相等则显示1；

(3) 如果三个数都相等则显示2。

5.22 从键盘输入一系列字符(以回车符结束)，并按字母、数字及其它字符分类计数，最后显示出这三类的计数结果。

5.23 已定义了两个整数变量 A 和 B，试编写程序完成下列功能：

(1) 若两个数中有一个是奇数，则将奇数存入 A 中，偶数存入 B 中；

(2) 若两个数均为奇数，则将两数均加1后存回原变量；

(3) 若两个数均为偶数，则两个变量均不改变。

5.24 假设已编制好5个歌曲程序，它们的段地址和偏移地址存放在数据段的跳跃表 SINGLIST 中。试编制一程序，根据从键盘输入的歌曲编号1～5，转去执行五个歌曲程序中的某一个。

第六章　子程序结构

子程序又称为过程，它相当于高级语言中的过程和函数。在一个程序的不同部分，往往要用到类似的程序段，这些程序段的功能和结构形式都相同，只是某些变量的赋值不同，此时就可以把这些程序段写成子程序形式，以便需要时可以调用它。有些程序段对于某个用户可能只用到一次，但它是一般用户经常用到的，如十进制数转换成二进制数，二进制数转换为十六进制数并显示输出等，对于这些常用的特定功能的程序段也经常编制成子程序的形式供用户使用。

在第十三章里，我们还要介绍模块化程序设计方法，这种方法按照功能把程序划分成多个模块，按模块来编制和调试程序，最后再把它们组合而形成一个大程序。这是一种很好的程序设计方法，而子程序结构就是模块化程序设计的重要工具。

6.1　子程序的设计方法

6.1.1　过程定义伪操作

过程定义伪操作用在过程(子程序)的前后，使整个过程形成清晰的，具有特定功能的代码块。其格式为：

```
procedure   name        PROC        Attribute
                        ⋮
procedure   name        ENDP
```

其中过程名为标识符，它又是子程序入口的符号地址，它的写法和标号的写法相同。属性(Attribute)是指类型属性，它可以是 NEAR 或 FAR。

在第三章中，我们已经介绍过 CALL 和 RET 指令都有 NEAR 或 FAR 的属性，段内调用使用 NEAR 属性，段间调用使用 FAR 属性。为了使用户的工作更加方便，IBM PC 的汇编程序用 PROC 伪操作的类型属性来确定 CALL 和 RET 指令的属性。也就是说，如果所定义的过程是 FAR 属性的，那么对它的调用和返回一定都是 FAR 属性的；如果所定义的过程是 NEAR 属性的，那么对它的调用和返回也一定是 NEAR 属性的。这样，用户只需在定义过程时考虑它的属性，而 CALL 和 RET 的属性可以由汇编程序来确定。用户对过程属性的确定原则很简单，即：

1) 调用程序和过程在同一个代码段中则使用 NEAR 属性；

2) 调用程序和过程不在同一个代码段中则使用 FAR 属性。

下面举例说明：

例 6.1　调用程序和子程序在同一代码段中。

```
MAIN        PROC        FAR
            ⋮
            CALL        SUBR1
            ⋮
```

```
        RET
MAIN    ENDP
;
SUBR1   PROC    NEAR
        ⋮
        RET
SUBR1           ENDP
```

由于调用程序 MAIN 和子程序 SUBR1 是在同一代码段中的，所以 SUBR1 定义为 NEAR 属性，这样 MAIN 中对 SUBR1 的调用和 SUBR1 中的 RET 就都是 NEAR 属性的。但是一般说来，主过程 MAIN 应定义为 FAR 属性，这是由于我们把程序的主过程看作 DOS 调用的一个子过程，因而 DOS 对 MAIN 的调用以及 MAIN 中的 RET 就是 FAR 属性的。当然 CALL 和 RET 的属性是汇编程序确定的，用户只需正确选择 PROC 的属性就可以了。

例 6.1 的情况也可以写成如下的程序：

```
MAIN    PROC    FAR
        ⋮
        CALL    SUBR1
        ⋮
        RET
SUBR1   PROC    NEAR
        ⋮
        RET
SUBR1   ENDP
MAIN    ENDP
```

也就是说，过程定义也可以嵌套，一个过程定义中可以包括多个过程定义。

例 6.2 调用程序和子程序不在同一个代码段内。

```
SEGX    SEGMENT
        ⋮
SUBT    PROC    FAR
        ⋮
        RET
SUBT    ENDP
        ⋮
        CALL    SUBT
        ⋮
SEGX    ENDS
;
SEGY    SEGMENT
        ⋮
        CALL    SUBT
        ⋮
SEGY    ENDS
```

SUBT 为一过程，它有两处被调用，一处是与它在同一段的 SEGX 段内，另一处是在另一段 SEGY 段内，为此 SUBT 必须具有 FAR 属性以适应 SEGY 段调用的需要。SUBT 既有 FAR 属性，

则不论在 SEGX 和 SEGY 段对 SUBT 的调用就都具有 FAR 属性了，这样不会发生什么错误，反之，如果这里的 SUBT 使用了 NEAR 属性，则在 SEGY 段内对它的调用就要出错了。

这里没有讨论调用程序和子程序不在同一程序模块中的情况，这将在第十三章中说明。

6.1.2 子程序的调用和返回

过程的正确执行是由子程序的正确调用及正确返回保证的。IBM PC 机的 CALL 和 RET 指令就完成了调用和返回的功能。为保证其正确性，除 PROC 的属性要正确选择外，还应该注意子程序运行期间的堆栈状态，由于 CALL 时已使返回地址入栈，所以 RET 时应该使返回地址出栈，如果子程序中不能正确使用堆栈而造成执行 RET 前 SP 并未指向进入子程序时的返回地址，则必然会导致运行出错，因此子程序中对堆栈的使用应该特别小心，以免发生错误。

6.1.3 保存与恢复寄存器

由于调用程序(又称主程序)和子程序经常是分别编制的，所以它们所使用的寄存器往往会发生冲突。如果主程序在调用子程序以前的某个寄存器内容在从子程序返回后还有用，而子程序又恰好使用了同一个寄存器，造成破坏了该寄存器的原有内容，那就会造成程序运行错误，这是不能允许的。为避免这种错误的产生在一进入子程序后就应该把子程序所需要使用的寄存器内容保存在堆栈中，而在退出子程序前把寄存器内容恢复原状。例如：

```
SUBT    PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        ⋮
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
SUBT    ENDP
```

在子程序设计时，应仔细考虑哪些寄存器是必须保存的，哪些寄存器是不必要或不应该保存的。一般说来，子程序中用到的寄存器是应该保存的。但是，如果使用寄存器来在主程序和子程序之间传送参数的话，这种寄存器就不一定需要保存，特别是用来向主程序回送结果的寄存器，就更不应该因保存和恢复寄存器而破坏了应该向主程序传送的信息。

6.1.4 子程序的变量传送

调用程序在调用子程序时，经常需要传送一些参数给子程序；子程序运行完后也经常要回送一些信息给调用程序。这种调用程序和子程序之间的信息传送称为变量传送(或称参数传送或过程通信)。变量传送方式可以有以下几种：

6.1.4.1 通过寄存器传送变量

这是最常用的一种方式，使用方便，但参量很多时不能使用这种方法。下面举例说明。

例 6.3 十进制到十六进制数转换程序。程序要求从键盘取得一个十进制数，然后把该数

以十六进制形式在屏幕上显示出来。

我们采用子程序结构，用一个子程序 DECIBIN 实现从键盘取得十进制数并把它转换为二进制数；另一个子程序 BINIHEX 把此二进制数以十六进制数的形式在屏幕上显示出来。为避免屏幕上的重叠，另外用 CRLF 子程序取得回车和换行效果。整个程序结构如图 6.1 所示。在这里各个子程序之间用 BX 寄存器来传送信息。在过程 DECIBIN 中取得的输入数据转换为二进制数后保存在 BX 寄存器中，而过程 BINIHEX 需要把 BX 寄存器中的数用十六进制形式显示出来。也就是说，BX 寄存器用来在过程间传递要转换的数。该程序的实现如图 6.2 所示。

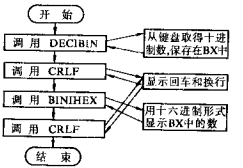


图 6.1 十进制到十六进制数转换的程序结构

```
;DECIHEX—main Program
;   Convert decimal on keybd to hex on screen
;* * * * * * * * * * * * * * * * * * * * * * * * * * * *
decihex segment
        assume cs:decihex

;MAIN  PART  OF  PROGRAM.  Connects  procedures  together.
main    proc    far

repeat: call    decibin         ;keyboard to binary
        call    crlf            ;print cr and lf
        call    binihex         ;binary to screen
        call    crlf            ;print cr and lf
        jmp     repeat          ;do it again

main    endp
;-----------------------------------------------------------
;PROCEDURE  TO  CONVERT  DEC  ON  KEYBD  TO  BINARY,
;  RESULT  IS  LEFT  IN  BX  REGISTER

decibin proc    near

        mov     bx,0            ;clear BX for number
;Get digit from keyboard, convert to binary
newchar:
        mov     ah,1            ;keyboard input
        int     21h             ;call DOS
        sub     al,30h          ;ASCII to binary
        jl      exit            ;jump if < 0
        cmp     al,9d           ;is it >9d ?
        jg      exit            ;yes, not dec digit
        cbw                     ;byte in AL to word in AX
;(digit is now in AX)
```

```
;Multiply number in BX by 10 decimal
            xchg    ax,bx          ;trade digit & number
            mov     cx,10d         ;put 10 dec in CX
            mul     cx             ;number times 10
            xchg    ax,bx          ;trade number & digit
;Add digit in AX to number in BX
            add     bx,ax          ;add digit to number
            jmp     newchar        ;get next digit
exit:
            ret                    ;return from decibin
decibin     endp                   ;end of decibin proc
;-------------------------------------------------
;PROCEDURE TO CONVERT BINARY NUMBER IN BX TO HEX ON CONSOLE
;  SCREEN
binihex     proc    near
            mov     ch,4           ;number of digits
rotate:     mov     cl,4           ;set count to 4 bits
            rol     bx,cl          ;left digit to right
            mov     al,bl          ;move to AL
            and     al,0fh         ;mask off left digit
            add     al,30h         ;convert hex to ASCII
            cmp     al,3ah         ;is it >9?
            jl      printit        ;jump if digit = 0 to 9
            add     al,7h          ;digit is A to F
printit:
            mov     dl,al          ;put ASCII char in DL
            mov     ah,2           ;Display Output funct
            int     21h            ;call DOS
            dec     ch             ;done 4 digits?
            jnz     rotate         ;not yet
            ret                    ;return from binihex
binihex     endp
;-------------------------------------------------
;PROCEDURE TO PRINT CARRIAGE RETURN AND LINEFEED
crlf        proc    near
            mov     dl,0dh         ;carriage return
            mov     ah,2           ;display function
            int     21h            ;call DOS
            mov     dl,0ah         ;linefeed
            mov     ah,2           ;display function
            int     21h            ;call DOS
            ret                    ;return from crlf
```

```
crlf        endp
;-------------------------------------------------
decihex     ends
;*************************************
            end         main
```

图 6.2　十进制到十六进制数转换的程序实现

6.1.4.2　如过程和调用程序在同一源文件(同一程序模块)中，则过程可直接访问模块中的变量。

例 6.4　主程序 MAIN 和过程 PROADD 在同一源文件中，要求用过程 PROADD 累加数组中的所有元素，并把和(不考虑溢出的可能性)送到指定的存储单元中去。图 6.3 表示了该程序。在这里过程 PROADD 直接访问模块的数据区。

```
;*************************************
data        segment                 ;define data segment
  ary       dw          100 dup(?)
  count     dw          100
  sum       dw          ?
data        ends
;*************************************
code        segment                 ;define code segment
;-------------------------------------------------
main        proc    far             ;main part of program
            assume cs:code,ds:data
start:      :                       ;starting execution address
            call    near ptr proadd
            :
main        endp                    ;end of main part of prog.
;-------------------------------------------------
proad       proc    near            ;define subprocedure
            push    ax              ;save the registers
            push    cx
            push    si
            lea     si,ary          ;put addr of ary in si
            mov     cx,count        ;put count in cx
            xor     ax,ax           ;clear ax
next:       add     ax,[si]         ;add the elements of the
            add     si,2            ;   array to AX
            loop    next
            mov     sum,ax          ;store the result in sum
            pop     si              ;restore the registers
            pop     cx
            pop     ax
            ret                     ;return
proadd      endp                    ;end subprocedure
```

```
;-----------------------------------------------
code        ends                        ;end of code segment
;*********************************************
            end     start               ;end assembly
```

图 6.3 例 6.4 的程序实现

6.1.4.3 如过程和调用程序不在同一程序模块中，则它们之间的通信问题将在第十三章中讨论。

如果数据段中有两个数组如下：

```
data        segment
  ary       dw          100  dup(?)
  count     dw          100
  sum       dw          ?
  ;
  num       dw          100  dup(?)
  n         dw          100
  total     dw          ?
data        ends
```

主程序两次调用 PROADD 要求分别累加 ARY 和 NUM 数组的内容。在这种情况下，如果还是采用例 6.4 中所述的变量传送方式的话，那就只有在调用 PROADD 累加 NUM 数组前，先把 NUM 数组及 N 的内容全部传送到 ARY 数组和 COUNT 单元去，PROADD 运行完后，再把 SUM 的结果送到 TOTAL 去。这显然不是一种好办法，为解决这一类问题，下面再介绍两种变量传送的方法。

6.1.4.4 通过地址表传送变量地址

例 6.4 的程序要求，采用通过地址表传送变量法的程序实现如图 6.4 所示。可以看出，这种方法是在主程序中建立一个地址表，把要传送给子程序的参数都存放在地址表中，然后把地址表的首地址通过寄存器 BX 传送到子程序中去。子程序通过地址表取得所需参数，并把结果存入指定的存储单元中去。这样做的结果，对于上述程序中要累加 NUM 数组的内容时，只须在程序中增加下述指令，再调用 PROADD 后就能在 TOTAL 中得到 NUM 的累加和。

```
        MOV     TABLE,OFFSET NUM
        MOV     TABLE+2,OFFSET N
        MOV     TABLE+4,OFFSET TOTAL
        MOV     BX,OFFSET TABLE
        CALL    PROADD
;*****************************************
prog_seg  segment

          org       100h

          assume cs:prog_seg,ds:prog_seg,ss:prog_seg

main      proc      near
                    ⋮
          mov       table,offset ary;put the addresses
          mov       table+2,offset count;   of ary,count and sum
```

```
        mov     table+4,offset sum  ;      in parameter table
        mov     bx,offset table     ;put addr of table in BX
        call    proadd              ;call proadd
                :
        int     20h
main    endp
;-------------------------------------------------------
proadd  proc    near                ;define subprocedure
        push    ax                  ;save registers
        push    cx
        push    si
        push    di
        mov     si,[bx]             ;get the addr of array,
        mov     di,[bx+2]           ;  the value of count,
        mov     cx,[di]             ;        and the addr of
        mov     di,[bx+4]           ;        result
        xor     ax,ax               ;clear AX
next:   add     ax,[si]             ;add the elements
        add     si,2                ;  of the array
        loop    next                ;        to AX
        mov     [di],ax             ;return result
        pop     di                  ;restore registers
        pop     si
        pop     cx
        pop     ax
        ret                         ;return
proadd  endp                        ;end subprocedure
;-------------------------------------------------------
  ary   dw      100 dup(?)          ;reserve 100 words for ary
  count dw      100                 ;  and 1 word for count
  sum   dw      ?                   ;        and 1 word for sum
  table dw      3 dup(?)            ;roserve 3 words for
                                    ;  parameter addresses
;-------------------------------------------------------
prog_seg ends
;*********************************
        end     main
```

图 6.4　例 6.4 采用通过地址表传送变量法的程序实现

此外，图 6.4 的程序采用 COM 文件形式，代码、数据和堆栈都设在一个段中。当然，这并不说明这里必须使用 COM 文件，只是给出另一种程序格式而已。

6.1.4.5　通过堆栈传送参数或参数地址

例 6.4 采用通过堆栈传送参数地址法的程序实现如图 6.5 所示。这种方法是在主程序里把参数地址保存到堆栈中，在子程序里从堆栈中取出参数以达到传送参数的目的。如本例中堆

栈最满时的状态如图 6.6 所示。必须注意，子程序结束时的 RET 指令应使用带常数的返回指令，以便返回主程序后堆栈能恢复原始状态不变。

```
;*******************************
parm_seg        segment                 ;define data segment
        ary     dw      100 dup(?)
        count   dw      100
        sum     dw      ?
parm_seg        ends
;*******************************
stack_seg       segment
        dw      100 dup(?)
  tos   label   word
stack_seg       ends
;*******************************
code1   segment                         ;define code segment
;-------------------------------------------
main    proc    far                     ;main part of program
        assume cs:code1,ds:parm_seg,ss:stack_seg
start:                                  ;starting execution address
;set up SS and SP register
        mov     ax,stack_seg
        mov     ss,ax
        mov     sp,offset tos
;set up stack for return
        push    ds                      ;save old data segment
        sub     ax,ax                   ;put zero in AX
        push    ax                      ;save it on stack
;set DS register to current data segment
        mov     ax,parm_seg             ;data segment addr
        mov     ds,ax                   ;  into DS register
;
        mov     bx,offset ary           ;push addr of ary
        push    bx                      ;  onto stack
        mov     bx,offset count         ;push addr of count
        push    bx                      ;  onto stack
        mov     bx,offset sum           ;push addr of sum
        push    bx                      ;  onto stack
        call    far ptr proadd
                :
        ret
main    endp
;-------------------------------------------
code1   ends
;*******************************
code2   segment
        assume cs:code2
```

```
;-----------------------------------------------------
proadd  proc    far
        push    bp                      ;save BP
        mov     bp,sp                   ;use BP to access parameter
        push    ax                      ;save other registers
        push    cx
        push    si
        push    di
        mov     si,[bp+0ah]             ;get addr of ary
        mov     di,[bp+8]               ;get addr of count
        mov     cx,[di]                 ; CX <-- (count)
        mov     di,[bp+6]               ;get addr of sum
        xor     ax,ax                   ;clear AX
next:   add     ax,[si]                 ;compute sum
        add     si,2
        loop    next
        mov     [di],ax                 ;store result
        pop     di                      ;restore registers
        pop     si
        pop     cx
        pop     ax
        pop     bp
        ret     6                       ;adjust stack and return
proadd  endp
;-----------------------------------------------------
code2   ends
;**********************************
        end     start                   ;end assembly
```

图 6.5　例 6.4 采用通过堆栈传送参数地址法的程序实现

IBM PC 机的结构伪操作 STRUC 给用户提供了很大的方便。下面我们先介绍 STRUC 伪操作的使用方法，然后再把它引入我们的例子中。

结构伪操作 STRUC 是 MASM 支持的一种伪操作，它可以把各种不同类型的数据放在同一个数据结构里，便于某些数据处理的需要。其格式为

```
structure name    STRUC
                  ⋮        (DB、DW、DD 等伪操作序列)
structure name    ENDS
```

其中由 DB、DW、DD 等伪操作定义不同类型的数据，它们可以使用变量名来表示所定义字段的起始地址，所以该变量名又称为字段名。例如

```
PERSONAL_DATA     STRUC
    INITIALS      DB    'XX'
    LAST_NAME     DB    5 DUP (?)
    ID            DB    0,0
    AGE           DB    ?
    WEIGHT        DW    ?
PERSONAL_DATA     ENDS
```

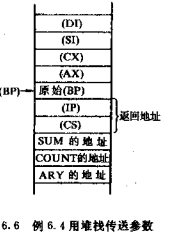


图 6.6　例 6.4 用堆栈传送参数地址时，堆栈最满时的状态

这是一个结构数据，其中每个字段给出不同类型的数据格式，组合而形成表示个人特征的有关信息。但是STRUC伪操作只能定义一种结构模式，它并不能把有关信息存入存储器，为了达到这一目的，必须使用结构预置语句，结构预置语句的格式是：

```
Variable  structure name (preassignment specifications)
```

其中变量为该结构的起始地址，结构名则必须与STRUC语句中的结构名相同，有关预赋值说明的规定如下：

STRUC伪操作中的定义语句可以作为预赋值的值，如果全部采用定义中的赋值，则预赋值说明可以用〈 〉来表示。此外，在一定条件下STRUC中定义的值可以用〈预赋值说明〉来代替。这一条件是：如果一个字段只有一个字符串或一个预赋值的值，则可以由〈预赋值说明〉来代替，如果一个字段中有多个值，则不允许用〈预赋值说明〉来代替。如PERSONAL_DATA结构数据中的LAST_NAME和ID字段是不允许代替的，而INITIALS、AGE和WEIGHT字段则允许用〈预赋值说明〉来代替。

〈预赋值说明〉可以由一连串常数表达式组成，每个字段用一个表达式，相互之间用逗号隔开，如果使用STRUC定义中的值则可不写任何值(缺省)。

例 6.5

```
EMPLOYEE_1   PERSONAL_DATA  ('JR',,,35)
EMPLOYEE_2   PERSONAL_DATA  <  >
```

则经汇编后存储器中存放的信息如图6.7(1)所示。

例 6.6 预赋值说明也可以使用DUP操作符，如：

```
EMPLOYEES   PERSONAL_DATA  100  DUP(<  >)
```

则把PERSONAL_DATA的结构定义顺序复制100次，如图6.7(2)所示。

以上我们已经说明了如何定义一个结构数据模式，以及如何用结构预置语句把结构数据存入存储器中。那么，在汇编语言程序中如何访问结构数据中的变量呢？IBM PC机中规定的格式是：

```
offset notation. field name
```

其中offset notation是指结构数据的首地址，field name则是所要访问的字段名。

例如，

```
MOV    AL,EMPLOYEE_1.LAST_NAME[SI]
```

如(SI)=2，则该指令把EMPLOYEE_1结构中的LAST_NAME字段的第3项送到AL寄存器中。如果BX寄存器的内容是EMPLOYEE_1的偏移地址值，则上述指令也可以表示为：

```
MOV    AL,[BX].LAST_NAME[SI]
```

又如，

```
MOV    AL,EMPLOYEES+4*12.LAST_NAME[SI]
```

则由EMPLOYEES+4*12表示EMPYOLEES中的第5个复制的PERSONAL_DATA结构数据的首地址，因此该指令把第5个结构数据中的LAST_NAME字段的第3项送到AL寄存器中。

上面，我们已经介绍了STRUC伪操作的使用方法。现在，我们来看例6.4的PROADD过程当使用堆栈来传送参数地址时，是如何使用STRUC伪操作的。所编程序如图6.8所示。在这里，由于在主程序中已经把STRUC定义所要求的结构数据存入堆栈，因而不需要再用结构预置语句来把信息存入存储区。在CODE2段中，使用访问结构数据的方法来取得主程序通过堆栈传送过来的参数。很时显，这种方法对于用户来说是很方便的，也可以避免由于计算地址而

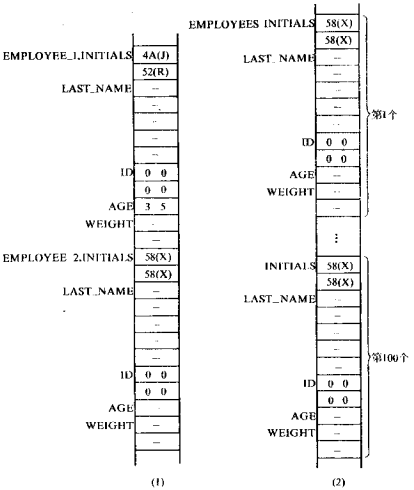


图 6.7　结构数据的预置

(1) 例 6.5 的信息存放情况　(2) 例 6.6 的信息存放情况

引起出错的可能性。

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
parm_seg        segment
  ary           dw      100  dup(?)
  count         dw      100
  sum           dw      ?
parm_seg        ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
stack_seg       segment
                dw      100  dup(?)
  tos           label   word
stack_seg       ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
code1     segment
```

```
        assume  cs:code1,ds:parm_seg,ss:stack_seg
start:                                          ;starting execution address
        mov     ax,stack_seg
        mov     ss,ax
        mov     sp,offset tos
        push    ds
        sub     ax,ax
        push    ax
        mov     ax,parm_seg
        mov     ds,ax
;
        mov     bx,offset ary
        push    bx              ;push addr of ary onto stack
        mov     bx,offset count
        push    bx              ;push addr of count onto stack
        mov     bx,offset sum
        push    bx              ;push addr of sum onto stack
        call    far ptr proadd
        ⋮
        ret
code1   ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
code2   segment
        assume cs:code2
        stack_strc      struc
          save_bp       dw      ?
          save_cs_ip    dw      2  dup(?)
          par3_addr     dw      ?
          par2_addr     dw      ?
          par1_addr     dw      ?
        stack_strc      ends
;-----------------------------------------------------------------
proadd  proc    far
        push    bp              ;save bp
        mov     bp,sp           ;use bp to access parameter
        push    ax              ;save other registers
        push    cx
        push    si
        push    di
        mov     si,[bp].par1_addr;get addr of par1 from stack
        mov     di,[bp].par2_addr;get value of par2
        mov     cx,[di]         ;   and put in CX
        mov     di,[bp].par3_addr;get addr of par3 from stack
        xor     ax,ax           ;clear ax
next:   add     ax,[si]         ;compute sum
        add     si,2
        loop    next
        mov     [di],ax         ;store result
        pop     di              ;restore registers
```

```
        pop     si
        pop     cx
        pop     ax
        pop     bp
        ret     6                   ;adjust stack and return
proadd  endp
;-------------------------------------------------
code2   ends
;*********************************
        end     start               ;end assembly
```

图 6.8 例 6.4 用结构伪操作实现用堆栈传送参数时的程序实现

以上说明的几种传送参数的方法，读者可以根据具体情况选择使用。必须注意，在我们的例子中，后两种方法都用来传送参数地址，实际上它们也可用来传送参数，在掌握了方法以后，读者可以灵活使用。

6.2 嵌套与递归子程序

6.2.1 子程序的嵌套

我们已经知道，一个子程序也可以作为调用程序去调用另一个子程序，这种情况就称为子程序的嵌套。嵌套的层次不限，其层数称为嵌套深度。图 6.9 表示了嵌套深度为 2 时的子程序嵌套情况。

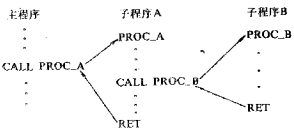


图 6.9 子程序的嵌套

嵌套子程序的设计并没有什么特殊要求，除子程序的调用和返回应正确使用 CALL 和 RET 指令外，要注意寄存器的保存和恢复，以避免各层子程序之间发生因寄存器冲突而出错的情况。如果程序中使用了堆栈，如使用堆栈来传送参数等，则对堆栈的操作要格外小心，避免因堆栈使用中的问题而造成子程序不能正确返回的出错情况。

6.2.2 递归子程序

在子程序嵌套的情况下，如果一个子程序调用的子程序就是它自身，这就称为递归调用。这样的子程序称为递归子程序。递归子程序对应于数学上对函数的递归定义，它往往能设计出效率较高的程序，可完成相当复杂的计算，因而它是很有用的。这里以阶乘函数为例，说明递归子程序的设计方法。

例 6.7 要求编制计算 N!（N≥0）的程序。

$$N! = N * (N-1) * (N-2) * \cdots * 1$$

其递归定义如下：

$$\begin{cases} 0! = 1 \\ N! = N * (N-1)! & (N>0) \end{cases}$$

我们可以根据递归定义来设计该程序。求 N! 本身是一个子程序，由于 N! 是 N 和(N−1)! 的

乘积，所以为求(N－1)!必须递归调用求N!的子程序，但每次调用所使用的参数都不相同。递归子程序的设计必须保证每次调用都不破坏以前调用时所用的参数和中间结果，所以一般把每次调用的参数、寄存器内容以及所有的中间结果都存放在堆栈中。我们把一次调用所保存的信息称为帧(frame)，一般一帧包括所保存的寄存器内容、参数或参数地址和中间结果等。每次调用把一帧信息存入堆栈。递归子程序中还必须包括基数的设置，当调用参数达到基数时还必须有一条条件转移指令实现嵌套退出，保证能按反向次序退出并返回主程序。根据这些要求可画出本例的程序框图如图6.10所示。编制程序如图6.11所示。

在程序运行的过程中，FACT不断调用自身，每调用一次使(N－1)及有关信息入栈，直到N等于基数0为止。当N等于0时开始返回，程序在不断返回的过程中，每次执行N*(N－1)!并保存中间结果直到N等于给定值为止。当N=3时堆栈状态如图6.12所示。在程序运行过程中，每次调用在堆栈中建立一帧，程序返回时，则每退出一帧计算一次中间结果，当程序运行完毕后，堆栈恢复原状，在RESULT单元中得到计算结果值。在N=3的情况下，RESULT单元中的值为6。

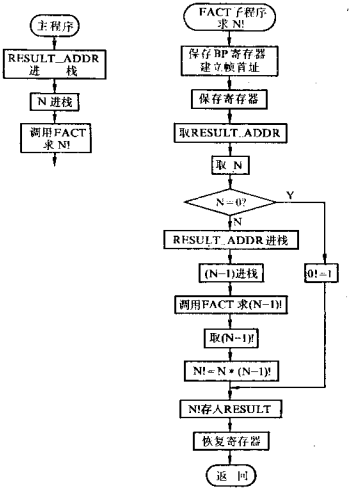


图6.10　计算N!的程序框图

```
;PROGRAM TITLE GOES HERE--- nf
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
data_seg        segment                 ;define data segment
  n_v           dw      ?
  result        dw      ?
data_seg        ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
stack_seg       segment
        dw      128 dup(0)
  tos   label   word
stack_seg       ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
code1   segment                         ;define code segment
;-----------------------------------------------------------
main    proc    far                     ;main part of program
        assume cs:code1,ds:data_seg,ss:stack_seg
start:                                  ;starting execution address
        mov     ax,stack_seg            ;set up stack and SP
        mov     ss,ax
        mov     sp,offset tos
;
        push    ds                      ;save old data segment
        sub     ax,ax                   ;put zero in AX
        push    ax                      ;save it on stack
;
        mov     ax,data_seg             ;data segment addr
        mov     ds,ax                   ;  into DS register
;MAIN PART OF PROGRAM GOES HERE
        mov     bx,offset result        ;push addr of result
        push    bx
        mov     bx,n_v                  ;push n
        push    bx
        call    far ptr fact
                :
        ret                             ;return to DOS
main    endp                            ;end of main part of program
;-----------------------------------------------------------
code1   ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
code    segment
;
  frame struc
        save_bp         dw      ?
        save_cs_ip      dw      2 dup(?)
        n               dw      ?
        result_addr     dw      ?
  frame ends
```

```
;
        assume  cs:code
;-------------------------------------------------------------
fact    proc    far                     ;define subprocedure
        push    bp                      ;save BP
        mov     bp,sp                   ;BP points to frame
        push    bx                      ;save registers used
        push    ax
        mov     bx,[bp].result_addr     ;load addr of result
                                        ;       in to BX
        mov     ax,[bp].n               ;load n
        cmp     ax,0                    ;n=0?
        je      done
        push    bx                      ;push addr of result
                                        ;   for next call
        dec     ax                      ;push n-1
        push    ax                      ;   for next call
        call    far ptr fact
        mov     bx,[bp].result_addr
        mov     ax,[bx]                 ;(AX)=n * result
        mul     [bp].n
        jmp     short return
done:   mov     ax,1                    ;(AX)=1
return: mov     [bx],ax                 ;result=(AX)
        pop     ax                      ;restore registers used
        pop     bx
        pop     bp
        ret     4                       ;return
fact    endp                            ;end subprocedure
;-------------------------------------------------------------
code    ends                            ;end of code segment
;**********************************************************
        end     start                   ;end assembly
```

图 6.11　计算 N! 的程序实现

从上述 N! 的递归子程序中可以看出，由于使用了 STRUC 伪操作使程序的结构清晰，也避免了计算参量地址时可能出现的错误，希望读者能逐步熟悉并使用这种方法。图 6.13 给出不用 STRUC 伪操作所编制的求 N! 的递归子程序供读者参考。这里我们只给出程序，有关运行情况及堆栈状态请读者自行分析。

在编制子程序特别是嵌套或递归子程序时，堆栈的使用是十分频繁的。在这里我们顺便说明一下在堆栈使用过程中应该注意有关堆栈溢出的问题。由于堆栈区域是在堆栈定义时就确定的，因而堆栈工作过程中有可能产生溢出。堆栈溢出有两种可能发生的情况：如堆栈已满，但还想再存入信息，这种情况称为堆栈上溢；另一种情况是如堆栈已空，但还想再取出信息，这种情况称为堆栈下溢。不论上溢或下溢都是不允许的，因此在编制程序时，如果可能发生堆栈溢出则应在程序中采取保护措施。这可用给 SP 规定上、下限，在进栈或出栈操作前先做 SP 和边界值的比较，如溢出则作溢出处理，以避免发生破坏其它存储区或使程序出错的情况。

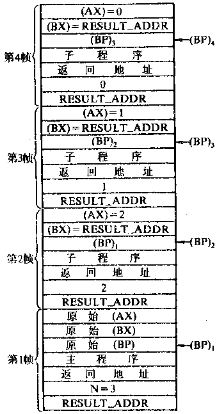


图 6.12 例 6.7 求 3! 时的堆栈状态

```
;*****************************************************
datarea   segment                         ;define data segment
          n         dw          ?
          result    dw          ?
datarea   ends
;*****************************************************
stack_seg           segment
          dw        128 dup(0)
  tos     label     word
stack_seg           ends
;*****************************************************
prognam   segment                         ;define code segment
;-----------------------------------------------------
main      proc      far                   ;main part of program
          assume cs:prognam,ds:datarea,ss:stack_seg
start:                                    ;starting execution address
          mov       ax,stack_seg          ;set up stack and SP
          mov       ss,ax
```

```
        mov     sp,offset tos
;
        push    ds                      ;save old data segment
        sub     ax,ax                   ;put zero in AX
        push    ax                      ;save it on stack
;
        mov     ax,dataarea             ;datarea segment addr
        mov     ds,ax                   ;  into DS register
;MAIN PART OF PROGRAM GOES HERE
        mov     bx,n
        push    bx
        call    fact
        pop     result
        ret                             ;return to DOS
main    endp                            ;end of main part of program
;-----------------------------------------------------------
fact    proc    near                    ;define subprocedure
        push    ax
        push    bp
        mov     bp,sp
        mov     ax,[bp+6]
        cmp     ax,0
        jne     fact1
        inc     ax
        jmp     exit
fact1:
        dec     ax
        push    ax
        call    fact
        pop     ax
        mul     [bp+6]
exit:   mov     [bp+6],ax
        pop     bp
        pop     ax
        ret
fact    endp                            ;end subprocedure
;-----------------------------------------------------------
prognam ends                            ;end of code segment
;*****************************************
        end     start                   ;end assembly
```

图 6.13　例 6.7 在堆栈区域内操作时的程序实现

6.3 子程序举例

例6.8 HEXIDEC是一个把十六进制数转换成十进制数的程序。要求把从键盘输入的0～FFFFH的十六进制正数转换为十进制数并从屏幕上显示出来。

这一例子的功能是和例6.3相反的。它由HEXIBIN和BINIDEC两个主要的子程序组成，由于主程序和子程序在同一个程序模块中，因而省略了对寄存器的保护和恢复工作，子程序之间的变量传送则采用寄存器传送的方式进行。程序可用Ctrl Break退出。程序如图6.14所示。

```
;HEXIDEC——Main program
;   converts hex on keyboard to dec on screen

;EQU STATEMENTS GO HERE

display   equu     2h              ;video output
key_in    equu     1h              ;keyboard input
doscall   equu     21h             ;DOS interrupt number

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
hexidec   segment                  ;define code segment
;-------------------------------------------------------
main      proc     far             ;main part of program

          assume cs:hexidec

;Main part of program links subroutines together

start:                             ;starting execution address

;set up stack for return
          push     ds              ;save old data segment
          sub      ax,ax           ;put zero in AX
          push     ax              ;zero on stack

;MAIN PART OF PROGRAM GOES HERE

          call     hexibin         ;keyboard to binary
          call     crlf            ;print cr & lf
;
          call     binidec         ;binary to decimal
          call     crlf            ;print cr & lf
          jmp      main            ;get next input
;
          ret                      ;return to DOS

main      endp                     ;end of main part of program
;-------------------------------------------------------
hexibin   proc     near            ;define subprocedure

;Subroutine to convert hex on keybd to binary
;   result is left in BX register
```

```
        mov     bx,0            ;clear BX for number
;Get digit from keyboard,convert to binary
newchar:
        mov     ah,key_in       ;keyboard input
        int     doscall         ;call DOS
        sub     al,30h          ;ASCII to binary
        jl      exit            ;jump if < 0
        cmp     al,10d          ;is it > 9d ?
        jl      add_to          ;yes,so it's digit
;not digit (0 to 9),may be letter (A to F)
        sub     al,27h          ;convert ASCII to binary
        cmp     al,0ah          ;is it <0a hex?
        jl      exit            ;yes,not letter
        cmp     al,10h          ;is it >0f hex?
        jge     exit            ;yes,not letter
;is hex digit,add to number in BX
add_to:
        mov     cl,4            ;set shift count
        shl     bx,cl           ;rotate BX 4 bits
        mov     ah,0            ;zero out AH
        add     bx,ax           ;add digit to number
        jmp     newchar         ;get next digit
exit:
        ret                     ;return from hexibin
hexibin endp                    ;end subprocedure
;-------------------------
binidec proc    near
;Subroutine to convert binary number in BX
;  to decimal on console screen
        mov     cx,10000d       ;divide by 10000
        call    dec_div
        mov     cx,1000d        ;divide by 1000
        call    dec_div
        mov     cx,100d         ;divide by 100
        call    dec_div
        mov     cx,10d          ;divide by 10
        call    dec_div
        mov     cx,1d           ;divide by 1
        call    dec_div
        ret                     ;return from binidec
;-------------------------
dec_div proc    near
;Subroutine to divide number in BX by number in CX
```

```
;print quotient on screen
;(numberator in DX+AX,denom in CX)
        mov     ax,bx           ;number low half
        mov     dx,0            ;zero out high half
        div     cx              ;divide by CX
        mov     bx,dx           ;remainder into BX
        mov     dl,al           ;quotient into DL
;print the contents of DL on screen
        add     dl,30h          ;convert to ASCII
        mov     ah,display      ;display function
        int     doscall         ;call DOS
        ret                     ;return from dec_div
dec_div  endp
;-------------------------------------------------
binidec  endp
;-------------------------------------------------
crlf    proc    near
;print carriage return and linefeed
        mov     dl,0ah          ;linefeed
        mov     ah,display      ;display function
        int     doscall         ;call DOS
;
        mov     dl,0dh          ;carriage return
        mov     ah,display      ;display function
        int     doscall         ;call DOS
        ret                     ;return from crlf
crlf    endp
;-------------------------------------------------
hexidec  ends                   ;end of code segment
;*****************************************
        end     start           ;end assembly
```

图 6.14　十六进制到十进制数转换的程序实现

例 6.9　本例为一个简单的信息检索系统。在数据区里有 10 个不同的信息，编号为 0～9，每个信息包括 30 个字符。现要求编制一个程序：从键盘接收 0～9 之间的一个编号，然后在屏幕上显示出相应编号的信息内容。程序如图 6.15 所示。在这个程序里，十个信息组成一个信息表，对信息表的查找是根据从键盘接收的编号来确定的，请读者注意从接收编号后到找到表格中所需区域为止的程序段，这也是查找表格的一种常用方法。此外，程序把显示一个信息编成一个独立功能的子程序 DISPLAY，并把其中常用的显示一个字符的功能也编成一个子程序 DISPCHAR，可以看出这样使程序的结构更加清晰了。

```
;*************************************
datarea     segment                 ;define data segment
thirty      db      30              ;value for mul instruction
```

```
;message   table
msg0       db      'I like my IBM - PC ---------- '
msg1       db      '8088 programming is fun ------ '
msg2       db      'Time to buy more diskettes ---- '
msg3       db      'This program works ----------- '
msg4       db      'Turn off that printer -------- '
msg5       db      'I have more memory than you --- '
msg6       db      'The PSP can be useful -------- '
msg7       db      'BASIC was easier than this --- '
msg8       db      'DOS is indispensable -------- '
msg9       db      'Last massage of the day ------ '
;error     message
errmsg     db      'error!!! invalid parameter!!'
datarea    ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
stack      segment
           db      256 dup(0)      ;256 bytes of stack space
  tos      label   word
stack      ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
prognam    segment                 ;define code segment
;-----------------------------------------------------------
main       proc    far             ;main part of program
           assume cs:prognam,ds:datarea,ss:stack
start:                             ;starting execution address
;set SS register to current stack segment
           mov     ax,stack
           mov     ss,ax
           mov     sp,offset tos
;set up stack for return
           push    ds              ;save old data segment
           sub     ax,ax           ;put zero in AX
           push    ax              ;save it on stack
;set DS register to current data segment
           mov     ax,datarea      ;datarea segment addr
           mov     ds,ax           ;  into DS register
;MAIN PART OF PROGRAM GOES HERE
           mov     ah,1            ;move a parameter from keybd
           int     21h
           sub     al,'0'          ;convert from ASCII to binary
           jc      error           ;branch if not numeric( <'0')
           cmp     al,9            ;check for valid numeric
```

```
        ja      error           ;branch if not valid( >'9')
;select appropriate message from message table
        mov     bx,offset msg0  ;point to first message
        mul     thirty          ;(AX)=(AL)*30
        add     bx,ax           ;point to desired message
        call    display         ;display the message at [BX]
        jmp     start
;display error message for invalid parameter
error:  mov     bx,offset errmsg
        call    display         ;display error message
        ret                     ;return to DOS
;-------------------------------------------------------
;Subroutine to display a message on the screen
;Enter with BX-->message to be displayed
;Message is assumed to be 30 characters long
display proc    near
        mov     cx,30           ;number of char. to display
displ:  mov     dl,[bx]         ;get next char. to display
        call    dispchar        ;display it
        inc     bx              ;point to next char.
        loop    displ           ;do it 30 times
        mov     dl,0dh          ;carriage return
        call    dispchar
        mov     dl,0ah          ;linefeed
        call    dispchar
        ret                     ;return to caller of
                                ;  'display'
display endp
;-------------------------------------------------------
;Subroutine to display a character on the screen
;Enter with (DL)= character to be displayed
dispchar proc   near
        mov     ah,2            ;display out a character
        int     21h
        ret                     ;return to caller of
                                ;  'dispchar'
dispchar endp
;-------------------------------------------------------
main    endp                    ;end of main part of program
;-------------------------------------------------------
prognam ends                    ;end of code segment
;*********************************
        end     start           ;end assembly
```

图 6.15　例 6.9 的程序实现

例 6.10 人名排序程序。先从终端键入最多 30 个人名，当所有人名都进入后，按字母上升的次序将人名排序，并在屏幕上显示已经排序后的人名。程序的总框图如图 6.16 所示，程序则如图 6.17 所示。

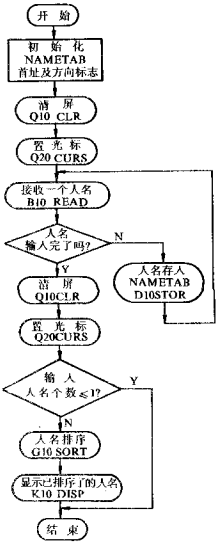


图 6.16 人名排序的程序框图

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
stack       segment
              dw      32  dup(?)
stack       ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
datasg      segment                     ;define data segment
  namepar     label   byte              ;name parameter list;
  maxnlen     db      21                ;max. length
```

```
namelen     db      ?                       ;no. chars entered
namefld     db      21 dup(?)               ;name
crlf        db      13,10,'$'
endaddr     dw      ?
messg1      db      'Name?','$'
namectr     db      0
nametab     db      30 dup(20 dup(' '))  ;name table
namesav     db      20 dup(?),13,10,'$'
swapped     db      0
datasg      ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
codesg      segment                         ;define code segment
;-----------------------------------------------------------------
begin       proc    far                     ;main part of program
            assume cs:codesg,ds:datasg,ss:stack,es:datasg
;set up stack for return
            push    ds                      ;save old data segment
            sub     ax,ax                   ;put zero in AX
            push    ax                      ;save it on stack
;set DS and ES register to current data segment
            mov     ax,datasg               ;data segment addr
            mov     ds,ax                   ;   into DS register
            mov     es,ax                   ;       and ES register
;MAIN PART OF PROGRAM GOES HERE
            cld
            lea     di,nametab
            call    q10clr                  ;clear screen
            call    q20curs                 ;set cursor
a20loop:
            call    b10read                 ;accept name
            cmp     namelen,0               ;any more names?
            jz      a30                     ;no,go to sort
            cmp     namectr,30              ;30 names entered?
            je      a30                     ;yes,go to sort
            call    d10stor                 ;store entered name in table
            jmp     a20loop                 ;end of input
a30:        call    q10clr                  ;clear screen
            call    q20curs                 ;   &set cursor
            cmp     namectr,1               ;one or no name entered?
            jbe     a40                     ;yes,exit
            call    g10sort                 ;sort stored names
            call    k10disp                 ;display sorted names
a40:        ret                             ;terminate
```

```
begin   endp                            ;end of main part of program
;-----------------------------------------------------------------
;       Accept name as input;
b10read proc    near
        mov     ah,09
        lea     dx,messg1               ;display prompt
        int     21h
        mov     ah,0ah
        lea     dx,namepar              ;accept name
        int     21h
        mov     ah,09
        lea     dx,crlf                 ;return/linefeed
        int     21h
;
        mov     bh,0                    ;clear chars after name
        mov     bl,namelen              ;get count of chars
        mov     cx,24
        sub     cx,bx                   ;calc remaining length
b20:    mov     namefld[bx],20h         ;set to blank
        inc     bx
        loop    b20
        ret
b10read endp
;-----------------------------------------------------------------
;       Store name in table;
d10stor proc    near
        inc     namectr                 ;add to number of name
        cld
        lea     si,namefld
        mov     cx,10
        rep     movsw                   ;move name to table
        ret
d10stor endp
;-----------------------------------------------------------------
;       Sort names in table;
g10sort proc    near
        sub     di,40                   ;set up stop address
        mov     endaddr,di
g20:    mov     swapped,0
        lea     si,nametab              ;set up start of table
g30:    mov     cx,20                   ;length of compare
        mov     di,si
        add     di,20                   ;next name for compare
```

```
        mov     ax,di
        mov     bx,si
        repe    cmpsb                   ;compare name to next
        jbe     g40                     ;no exchange
        call    h10xchg                 ;exchange
g40:    mov     si,ax
        cmp     si,endaddr              ;end of table?
        jbe     g30                     ;no,continue
        cmp     swapped,0               ;any swaps?
        jnz     g20                     ;yes,continue
        ret                             ;no,end of sort

g10sort endp
;-------------------------------------------------------------
;       Exchange table entries:

h10xchg proc    near
        mov     cx,10
        lea     di,namesav
        mov     si,bx
        rep     movsw                   ;move lower item to save
;
        mov     cx,10
        mov     di,bx
        rep     movsw                   ;move higher item to lower
;
        mov     cx,10
        lea     si,namesav
        rep     movsw                   ;move save to higher item
        mov     swapped,1               ;signal that exchange made
        ret

h10xchg endp
;-------------------------------------------------------------
;       Display sorted names:

k10disp proc    near
        lea     si,nametab
k20:    lea     di,namesav              ;initialize start of table
        mov     cx,10
        rep     movsw
        mov     ah,9
        lea     dx,namesav
        int     21h                     ;display
        dec     namectr                 ;is this last one?
        jnz     k20                     ;no,loop
        ret                             ;yes,exit
```

```
k10disp endp
;-------------------------------------------------------------
;       Clear screen;
q10clr  proc    near
        mov     ax,0600h
        mov     bh,61h                  ;color(07 for BW)
        sub     cx,cx
        mov     dx,184fh
        int     10h
        ret
q10clr  endp
;-------------------------------------------------------------
;       Set cursor;
q20curs proc    near
        mov     ah,2
        sub     bh,bh
        sub     dx,dx                   ;set cursor to 0,0
        int     10h
        ret
q20curs endp
;-------------------------------------------------------------
codeseg ends
;*****************************************
        end     begin
```

图 6.17　人名排序的程序实现

可以看出，这是一个典型的使用子程序结构的程序，所有具有独立功能的部分都采用了子程序结构，因而主程序设计得使人一目了然。主程序调用 6 个子程序，它们的过程名和功能如下：

- Q10CLR　清屏。
- Q20CURS　置光标。
- B10READ　接收键入的人名存放在 NAMEPAR 中，并用空格符清除其后的单元。
- D10STOR　把人名从 NAMEPAR 传送到 NAMETAB 中，并用 NAMECTR 计数。
- G10SORT　用起泡排序算法对人名排序，并用 SWAPPED 作为交换标志控制循环的结束。其中调用子程序 H10XCHG 来交换两个人名串的位置，并设置 SWAPPED 标志。
- K10DISP　显示已经排序了的人名。

可见每个子程序的功能都是很明确而且独立的。其中 Q10CLR 和 Q20CURS 使用了 BIOS 调用，这方面的内容我们将在第九章说明，在这里我们只要知道它们的功能分别是清除屏幕和把光标移到起始位置就可以了。其它子程序都是易于阅读的。可以看出这样的程序结构是很清晰的，所以，读者在学完本章后应善于使用子程序结构来编制程序，这将会取得很好的效果。

本例中的所有主、子程序都在同一个源文件中，因而变量传送问题的处理是简单的，所有子程序都可访问数据段中的变量，而且也省略了寄存器的保存和恢复工作。

6.4 DOS 系统功能调用

系统功能调用是 DOS 为系统程序员及用户提供的一组常用子程序，在前面的章节中我们已经用过其中的某些功能，这里再简单作一说明。

DOS 规定用中断指令 INT 21H 进入各功能调用子程序的总入口，再为每个功能调用规定一个功能号以便进入相应各个子程序的入口。子程序的入口参数及出口参数在每个功能调用的说明(见附录三)中可以查到。

DOS 共提供了约 80 个功能调用。大致分为设备管理、文件管理和目录管理等几类。有关设备管理的功能调用将在第九章中说明，有关文件管理的功能调用将在第十二章说明。

这里简单说明 DOS 系统功能调用的使用方法，如下：

1) 在 AH 寄存器中存入所要调用功能的功能号；
2) 根据所调用功能的规定设置入口参数；
3) 用 INT 21H 指令转入子程序入口；
4) 相应的子程序运行完后，可以按规定取得出口参数。

习 题

6.1 下面的程序段有错吗？若有，请指出错误。

```
CRAY    PROC
        PUSH    AX
        ADD     AX,BX
        RET
ENDP    CRAY
```

6.2 已知堆栈寄存器 SS 的内容是 0F0A0H，堆栈指示器 SP 的内容是 00B0H，先执行两条把 8057H 和 0F79BH 分别入栈的 PUSH 指令，然后执行一条 POP 指令。试画出示意图说明堆栈及 SP 内容的变化过程。

6.3 分析图 6.18 的程序，画出堆栈最满时各单元的地址及内容。

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
s_seg   segment  at 1000h                ;define stack segment
        dw       200 dup(?)
  tos   label    word
s_seg   ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
c_seg   segment                          ;define code segment

        assume cs:c_seg,ss:s_seg

        mov      ax,s_seg
        mov      ss,ax
        mov      sp,offset tos
;
        push     ds
```

```
        mov     ax,0
        push    ax
          ⋮
        push    t_addr
        push    ax
        pushf
          ⋮
        popf
        pop     ax
        pop     t_addr
        ret
c_seg   ends                                    ;end of code segment
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        end     c_seg                           ;end assembly
```

图 6.18　6.3 题的程序

6.4　分析图 6.19 程序的功能，写出堆栈最满时各单元的地址及内容。

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
stack   segment at 500h
        dw      128  dup (?)
  tos   label   word
stack   ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
code    segment                                 ;define code segment
;-------------------------------------------------------------------
main    proc    far                             ;main part of program
        assume cs:code,ss:stack
start:                                          ;starting execution address
        mov     ax,stack
        mov     ss,ax
        mov     sp,offset tos
;
        push    ds
        sub     ax,ax
        push    ax
;MAIN PART OF PROGRAM GOES HERE
        mov     ax,4321h
        call    htoa
        ret                                     ;return to DOS
main    endp                                    ;end of main part of program
;-------------------------------------------------------------------
```

```
htoa        proc    near            ;define subprocedure htoa
            cmp     ax,15
            jle     b1
            push    ax
            push    bp
            mov     bp,sp
            mov     bx,[bp+2]
            and     bx,000fh
            mov     [bp+2],bx
            pop     bp
            mov     cl,4
            shr     ax,cl
            call    htoa
            pop     ax
b1:         add     al,30h
            cmp     al,3ah
            jl      printit
            add     al,7h
printit:    mov     dl,al
            mov     ah,2
            int     21h
            ret
htoa        endp                    ;end subprocedure
;-------------------------------------------------
code        ends                    ;end of code segment
;*****************************************
            end     start           ;end assembly
```

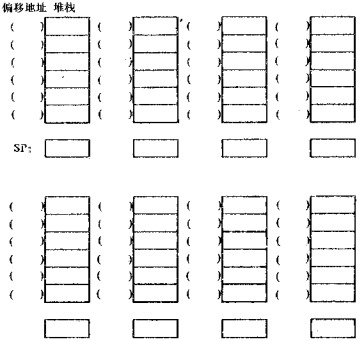
图 6.19　6.4 题的程序

6.5　图 6.20 是一个程序清单，请在下面的图中填入此程序执行过程中的堆栈变化。

```
                                ;***************
0000                            stacksg  segment
0000    20    [                         dw       32 dup  (?)
                ? ? ? ?
                                ]
0040                            stacksg  ends
                                ;***************
0000                            codesg   segment  para  'code'
                                ;-------------------------
0000                            begin    proc     far
                                         assume cs:codesg,ss:stacksg
0000    1E                               push     ds
```

```
0001  2B  C0                    sub     ax,ax
0003  50                       push    ax
0004  E8  0008  R              call    b10
                               ;--------------------------------
0007  CB                       ret
0008                     begin endp
                               ;--------------------------------
0008                     b10   proc
0008  E8  000C  R              call    c10
                               ;--------------------------------
000B  C3                       ret
000C                     b10   endp
                               ;--------------------------------
000C                     c10   proc
                               ;--------------------------------
000C  C3                       ret
000D                     c10   endp
                               ;--------------------------------
000D                     codeag ends
                               ;**********************
                               end     begin
```

图 6.20　6.5题的程序清单



6.6 下面是用 STRUC 伪操作定义的参数表 NAMELIST。

(1) 请用结构预置语句分配此结构的存储区；

(2) 编写一程序段，从键盘输入字符(用 DOS 功能调用)存入结构中，然后将输入的字符数送入 DISPFILE 单元中。

```
NAMELIST    STRUC
  MAXLEN    DB      100
  ACTLEN    DB      ?
  NAMEIN    DB      100  DUP (' ')
NAMELIST    ENDS
```

6.7 将学生的班级、姓名和成绩定义为一个结构，再用五个结构预置语句产生五个学生的成绩登记表。然后编写程序把成绩小于60分的学生情况显示出来。

6.8 写一段子程序 SKIPLINES，完成输出空行的功能。空出的行数在 AX 寄存器中。

6.9 设有10个学生的成绩分别是76,69,84,90,73,88,99,63,100,80分，试编制一个子程序统计60～69分，70～79分，80～89分，90～99分和100分的人数，分别存放到 S6,S7,S8,S9和 S10单元中。

6.10 编写一个有主程序和子程序结构的程序模块。子程序的参数是一个 N 字节数组的首地址 TABLE，数 N 及字符 CHAR，要求在 N 字节数组中查找字符 CHAR，并记录该字符的出现次数。主程序则要求从键盘接收一串字符以建立字节数组 TABLE，并逐个显示从键盘输入的每个字符 CHAR 以及它在 TABLE 数组中出现的次数。(为简化起见，假设出现次数≤15，可以用十六进制形式把它显示出来。)

6.11 编写一个子程序嵌套结构的程序模块，分别从键盘输入姓名及8个字符的电话号码，并以一定的格式显示出来。

主程序 TELIST：

- 显示提示符 INPUT NAME:；
- 调用子程序 INPUT_NAME 输入姓名；
- 显示提示符 INPUT A TELEPHONE NUMBER:；
- 调用子程序 INPHONE 输入电话号码；
- 调用子程序 PRINTLINE 显示姓名及电话号码。

子程序 INPUT_NAME：

- 调用键盘输入子程序 GETCHAR，把输入的姓名存放在 INBUF 缓冲区中；
- 把 INBUF 中的姓名移入输出行 OUTNAME。

子程序 INPHONE：

- 调用键盘输入子程序 GETCHAR，把输入的8位电话号码存放在 INBUF 缓冲区中；
- 把 INBUF 中的号码移入输出行 OUTPHONE。

子程序 PRINTLINE：

显示姓名及电话号码，格式为：

```
NAME              TEL.
×××               ×××
```

6.12 编写子程序嵌套结构的程序，把整数分别用二进制和八进制形式显示出来。

主程序 BANDO：把整数字变量 VAL1存入堆栈，并调用子程序 PAIRS；

子程序 PAIRS：从堆栈中取出 VAL1，调用二进制显示程序 OUTBIN 显示出与其等效的二进制数；输出8个空格；调用八进制显示程序 OUTOCT 显示出与其等效的八进制数；调用输出回车及换行符的子程序。

6.13 给定一个正数 N≥1 存放在 NUM 单元中，试编写一段递归子程序计算 FIB(N)，并将结果存入 RESULT 单元中。

Fibonacci 数的定义如下：

$$\begin{cases} FIB(1)=1 \\ FIB(2)=1 \\ FIB(n)=FIB(n-2)+FIB(n-1) & n>2 \end{cases}$$

6.14 编写一个递归程序，计算 ackermann 函数 ACK(m,n)的值。对于 m≥0和 n≥0的函数 ACK(m,n)由下式定义：

$$\begin{cases} ACK(0,n)=n+1 \\ ACK(m,0)=ACK(m-1,1) \\ ACK(m,n)=ACK(m-1,ACK(m,n-1)) \end{cases}$$

m 和 n 在主程序中从键盘输入，如果 m 或 n 小于零，则显示："Error in input data!"；如果 m 和 n 都大于零，则转递归子程序 ACK，在计算 ACK(m,n)函数值时，当 n 等于零则递归以降低 n 的值；当 m 等于零，则递归结束，求得函数值 n+1。

第七章　高级汇编语言技术

7.1　宏　汇　编

上一章介绍了子程序结构。我们已经了解到，使用子程序结构具有很多优点，可以节省存储空间及程序设计所化的时间，可提供模块化程序设计的条件，便于程序的调试及修改等。但是，使用子程序也有一些缺点：为转子及返回、保存及恢复寄存器以及参量的传送等都要增加程序的开销，这些操作所消耗的时间以及它们所占用的存储空间，都是为取得子程序结构使程序模块化的优点而增加的额外开销。因此，有时特别在子程序本身较短或者是需要传送的参量较多的情况下使用宏汇编就更加有利。

7.1.1　宏定义和宏调用

宏是源程序中一段有独立功能的程序代码。它只需要在源程序中定义一次，就可以多次调用它，调用时只需要用一个宏指令语句就可以了。

宏定义是用一组伪操作来实现的。其格式是：

```
macro name    MACRO    [dummy parameter list]
              ⋮        (宏定义体)
              ENDM
```

其中 MACRO 和 ENDM 是一对伪操作。这对伪操作之间是宏定义体，一是一组有独立功能的程序代码。宏指令名(macro name)给出该宏定义的名称，调用时就使用宏指令名来调用该宏定义。宏指令名的第一个符号必须是字母，其后可以跟字母、数字或下划线字符。其中哑元表(dummy parameter list)给出了宏定义中所用到的形式参数(或称虚参)，每个哑元之间用逗号隔开。

经宏定义定义后的宏指令就可以在源程序中调用。这种对宏指令的调用称为宏调用，宏调用的格式是：

```
macro name    [actual parameter list]
```

实元表(actual parameter list)中的每一项为实元，相互之间用逗号隔开。

当源程序被汇编时，汇编程序将对每个宏调用作宏展开。宏展开就是用宏定义体取代源程序中的宏指令名，而且用实元取代宏定义中的哑元。在取代时，实元和哑元是一一对应的，即第一个实元取代第一个哑元，第二个实元取代第二个哑元……依此类推。一般说来，实元的个数应该和哑元的个数相等，但汇编程序并不要求它们必须相等。若实元个数大于哑元个数，则多余的实元不予考虑；若实元个数小于哑元个数，则多余的哑元作"空"处理。另外，应该注意，宏展开后即用实元取代哑元后，所得到的语句应该是有效的，否则汇编程序将会指示出错。

下面我们用一个例子来说明宏定义、宏调用和宏展开的情况。

例 7.1　用宏指令定义两个字操作数相乘，得到一个 16 位的第三个操作数，作为结果。

宏定义：

```
MULTIPLY    MACRO    OPR1,OPR2,RESULT
```

```
        PUSH    DX
        PUSH    AX
        MOV     AX,OPR1
        IMUL    OPR2
        MOV     RESULT, AX
        POP     AX
        POP     DX
        ENDM
```

宏调用：

```
                :
        MULTIPLY CX,VAR, XYZ [BX]
                :
        MULTIPLY 240,BX,SAVE
                :
```

宏展开：

```
                :
    +   PUSH    DX
    +   PUSH    AX
    +   MOV     AX, CX
    +   IMUL    VAR
    +   MOV     XYZ [BX],AX
    +   POP     AX
    +   POP     DX
                :
    +   PUSH    DX
    +   PUSH    AX
    +   MOV     AX,240
    +   IMUL    BX
    +   MOV     SAVE, AX
    +   POP     AX
    +   POP     DX
                :
```

汇编程序在所展开的指令前加上'+'号以示区别。从上面的例子可以看出：由于宏指令可以带哑元，调用时可以用实元取代，这就避免了子程序因变量传送带来的麻烦，使宏汇编的使用增加了灵活性。而且实元可以是常数、寄存器、存储单元名以及用寻址方式能找到的地址或表达式等。从以后的例子中并可看到，实元还可以是指令的操作码或操作码的一部分等，宏汇编的这一特性是子程序所不及的。但是，宏调用的工作方式和子程序调用的工作方式是完全不同的，图 7.1 说明了两者的区别。可以看出，子程序是在程序执行期间由主程序调用的，它只占有它自身大小的一个空间；而宏调用则是在汇编期间展开的，每调用一次就把宏定义体展开一次，因而它占有的存储空间与调用次数有关，调用次数越多则占有的存储空间也越大。前面已经提到，用宏汇编可以免去执行时间上的额外开销，但如果宏调用次数较多的话，则其空间上的开销也是应该考虑的因素。因而，读者可根据具体情况来选择使用方案。一般说来，由于宏汇编可能占用较大的空间，所以代码较长的功能段往往使用子程序而不用宏汇编；但那些较

短的且变元较多的功能段，则使用宏汇编就更加合理了。下面通过例子来说明宏汇编的各种使用方法，同时说明一些在宏汇编中常用的伪操作。

7.1.2 宏指令举例

例 7.2 宏定义可以无变元

宏定义：

```
SAVEREG  MACRO
         PUSH    AX
         PUSH    BX
         PUSH    CX
         PUSH    DX
         PUSH    SI
         PUSH    DI
         ENDM
```

宏调用：

```
         SAVEREG
```

宏展开则将宏定义体的内容全部列出。

例 7.3 变元可以是操作码

宏定义：

```
FOO      MACRO    P1,P2,P3
         MOV      AX,P1
         P2       P3
         ENDM
```

宏调用：

```
         FOO      WORD_VAR,INC,AX
```

宏展开：

```
+        MOV      AX,WORD_VAR
+        INC      AX
```

例 7.4 变元可以是操作码的一部分，但在宏定义体中必须用 & 作为分隔符。

宏定义：

```
LEAP     MACRO    COND,LAB
         J&COND   LAB
         ENDM
```

宏调用：

```
         LEAP     Z,THERE
         LEAP     NZ,HERE
```

宏展开：

```
+        JZ       THERE
+        JNZ      HERE
```

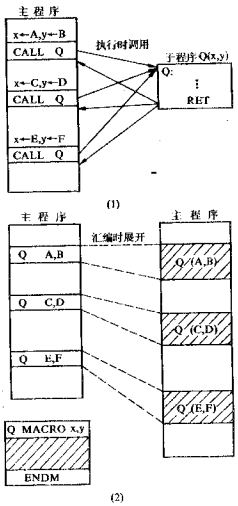


图 7.1 子程序调用和宏调用工作方式的区别

(1)子程序调用的工作方式； (2)宏调用的工作方式。

& 是一个操作符，它在宏定义体中可以作为哑元的前缀，展开时可以把 & 前后两个符号合并而形成一个符号，这个符号可以是操作码、操作数或是一个字符串。下面两个例子进一步具体说明这个问题。

例 7.5

宏定义：

```
FO      MACRO    P1
        JMP      TA&P1
        ENDM
```

宏调用：

```
        FO       WORD_VAR
```

宏展开：

```
+       JMP      TAWORD_VAR
```

在这里，如果宏定义写为

```
FO      MACRO    P1
        JMP      TAP1
        ENDM
```

则在展开时，汇编程序把 TAP1 看作一个独立的标号，并不把 TAP1 中的 P1 作为哑元看待，这样就不能得到预期的结果。

例 7.6 变元是 ASCII 串的情况

宏定义：

```
MSGGEN    MACRO     LAB, NUM, XYZ
          LAB&NUM   DB  'HELLO MR. &XYZ'
          ENDM
```

宏调用：

```
          MSGGEN    MSG,1,TAYLOR
```

宏展开：

```
+         MSG1      DB  'HELLO MR. TAYLOR'
```

例 7.7 宏指令名可以与指令助记符或伪操作名相同，在这种情况下，宏指令的优先级最高，而同名的指令或伪操作就失效了。伪操作 PURGE 可以用来在适当的时候取消宏定义，以便恢复指令的原始含义。本例说明 PURGE 的用法。

宏定义：

```
ADD       MACRO     OPR1,OPR2,RESULT
            ⋮
          ENDM
```

宏调用：

```
                    ⋮
          ADD       XX,YY ,ZZ
          PURGE     ADD
                    ⋮
```

在宏调用后，用 PURGE 伪操作取消宏定义，以便恢复 ADD 指令的原始含义，在 PURGE ADD 后面所用的 ADD 指令，则服从机器指令的定义。

PURGE 伪操作可同时取消多个宏定义，此时各宏指令名之间需用逗号隔开。

例 7.8 LOCAL 伪操作的使用。宏定义体内允许使用标号，如

宏定义：

```
ABSOL    MACRO    OPER
         CMP      OPER,0
         JGE      NEXT
         NEG      OPER
NEXT:
         ENDM
```

如果程序中多次调用该宏定义时，展开后会出现标号的多重定义，这是不能允许的。为此系统提供了 LOCAL 伪操作，其格式是

```
         LOCAL    list of local labels
```

其中局部标号表内的各标号之间用逗号隔开。汇编程序对 LOCAL 伪操作的局部标号表中的每一个局部标号建立唯一的符号(用??0000～??FFFF)以代替在展开中存在的每个局部标号。必须注意，LOCAL 伪操作只能用在宏定义体内，而且它必须是 MACRO 伪操作后的第一个语句，在 MACRO 和 LOCAL 伪操作之间还不允许有注释和分号标志。

本例中的 ABSOL 宏定义在考虑到有多次调用可能性的情况下，应定义为：

```
ABSOL    MACRO    OPER
         LOCAL    NEXT
         CMP      OPER,0
         JGE      NEXT
         NEG      OPER
NEXT:
         ENDM
```

宏调用：

```
           ⋮
         ABSOL    VAR
           ⋮
         ABSOL    BX
           ⋮
```

宏展开：

```
           ⋮
+        CMP      VAR,0
+        JGE      ?? 0000
+        NEG      VAR
+?? 0000:
           ⋮
+        CMP      BX,0
+        JGE      ?? 0001
+        NEG      BX
+?? 0001:
           ⋮
```

宏定义允许嵌套，下面两个例子说明宏嵌套的情况。

例 7.9 宏定义中允许使用宏调用，其限制条件是：必须先定义后调用。

宏定义：

```
DIF     MACRO   X,Y
        MOV     AX,X
        SUB     AX,Y
        ENDM
DIFSQR  MACRO   OPR1,OPR2,RESULT
        PUSH    DX
        PUSH    AX
        DIF     OPR1,OPR2
        IMUL    AX
        MOV     RESULT ,AX
        POP     AX
        POP     DX
        ENDM
```

宏调用：

```
        DIFSQR  VAR1,VAR2,VAR3
```

宏展开：

```
+       PUSH    DX
+       PUSH    AX
+       DIF     VAR1,VAR2
+       MOV     AX,VAR1
+       SUB     AX,VAR2
+       IMUL    AX
+       MOV     VAR3, AX
+       POP     AX
+       POP     DX
```

在汇编后形成的 LST 清单中，我们可以看到上述宏展开的结果。但是，其中的 DIF VAR1，VAR2 语句是为用户方便而提供的，它并不占有存储单元，也就是说，在 OBJ 文件中，这一语句是不存在的。

例 7.10 宏定义体内不仅可以使用宏调用，也可以包含宏定义。

宏定义：

```
DEFMAC    MACRO       MACNAM,OPERATOR
          MACNAM      MACRO       X,Y,Z
                      PUSH        AX
                      MOV         AX,X
                      OPERATOR    AX,Y
                      MOV         Z,AX
                      POP         AX
                      ENDM
          ENDM
```

其中 MACNAM 是内层的宏定义名，但又是外层宏定义的哑元，所以当调用 DEFMAC 时，就形成一个宏定义。

宏调用：

```
          DEFMAC    ADDITION,ADD
```

宏展开：

```
+       ADDITION   MACRO    X,Y,Z
                   PUSH     AX
                   MOV      AX,X
                   ADD      AX,Y
                   MOV      Z,AX
                   POP      AX
                   ENDM
```

形成加法宏定义 ADDITION。同样，宏调用

```
           DEFMAC    SUBTRACT,SUB
```

可形成减法的宏定义。宏调用

```
           DEFMAC    LOGOR,OR
```

可形成逻辑或的宏定义等。当然在形成这些宏定义后，就可以使用宏调用

```
           ADDITION  VAR1,VAR2,VAR3
```

而展开成：

```
+          PUSH      AX
+          MOV       AX,VAR1
+          ADD       AX,VAR2
+          MOV       VAR3,AX
+          POP       AX
```

例 7.11 这里再介绍一个宏定义的变元中使用的伪操作%，它的格式是：

```
%expression
```

汇编程序把跟在%之后的表达式的值转换成当前基数下的数，在展开期间，用这个数来取代哑元。

宏定义：

```
MSG        MACRO          COUNT,STRING
           MSG&COUNT      DB      STRING
           ENDM
ERRMSG     MACRO          TEXT
           CNTR=CNTR+1
           MSG            %   CNTR,TEXT
           ENDM
```

宏调用：

```
           ⋮
CNTR=0
ERRMSG     'SYNTAX ERROR'
           ⋮
ERRMSG     'INVALID OPERAND'
           ⋮
```

宏展开：

```
           ⋮
+          MSG1    DB     'SYNTAX ERROR'
           ⋮
```

```
+       MSG2    DB      'INVALID OPERAND'
        :
```

一般在 LST 清单中，使用隐含的.XALL 伪操作，即不产生代码的语句在清单中并不列出。为了能看到宏展开后所有的语句，可以在源程序中增加.LALL 语句，此时就能看到所有的语句了。实际上本例的展开可分为两层。第一层展开为：

```
        :
        CNTR=0
+       CNTR=CNTR+1
+       MSG     %   CNTR,'SYNTAX ERROR'
        :
+       CNTR=CNTR+1
+       MSG     %   CNTR,'INVALID OPERAND'
        :
```

第二层展开为：

```
        :
        CNTR=0
+       CNTR=CNTR+1
+       MSG1    DB      'SYNTAX ERROR'
        :
+       CNTR=CNTR+1
+       MSG2    DB      'INVALID OPERAND'
        :
```

例 7.12 为宏汇编使用举例，源程序如图 7.2 所示。其中有两个伪操作是我们还不熟悉的。一个是双分号;;;,这是 MASM 提供的一种伪操作，它用在宏定义或下一节要说明的重复语句中，前面用双分号的注释，在宏展开时将不予展开。另一个伪操作是 .SALL，它和.XALL 和.LALL 同样用来控制汇编清单中宏展开的列出情况，当使用.SALL 时，在汇编清单中将不列出任何展开后的信息。因此在本例中代码段的 LST 清单如图 7.3 所示。

```
;-------------------------------------------------------------
init2   macro       csname,dsname,ssname
        assume      cs:csname,ds:dsname,ss:ssname,es:dsname
        push        ds
        sub         ax,ax
        push        ax
        mov         ax,dsname
        mov         ds,ax
        mov         es,ax
        endm
;-------------------------------------------------------------
prompt  macro       message
;       This macro displays any message
;;      Generates code that links to DOS
          mov       ah,9                    ;request display
```

```
        lea     dx,message
        int     21h
        endm
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
stack   segment para stack 'stack'
        dw      32  dup(?)
stack   ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
data    segment para 'data'
  message1      db      'customer name?',13,10,'$'
  message2      db      'customer address?',13,10,'$'
data    ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
cseg    segment para 'code'
;-----------------------------------------------
begin   proc    far
        .sall
        init2   cseg,data,stack
        prompt  message1
        .lall
        prompt  message2
        ret
begin   endp
;-----------------------------------------------
cseg    ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        end     begin
```

图 7.2　例 7.12 的源程序

```
                        ;* * * * * * * * * * * * * * * * * * * * * * * * *
0000                    cseg    segment para 'code'
                        ;-------------------------------------------
0000                    begin   proc    far
                                .sall
                                init2   cseg,data,stack
                                prompt  message1
                                .lall
                                prompt  message2
                          +
                          +;    This macro displays any message
                          +
                          +
                          +
```

```
0013   B4  09              +     mov     ah,9        ;request display
                           +
0015   8D  16  0011  R     +     lea     dx,message2
0019   CD  21              +     int     21h
                           +
001B   CB                        ret
001C                       begin endp
                           ;--------------------------------
001C                       cseg  ends
                           ;**********************
                                 end     begin
```

图 7.3　例 7.12 代码段的 LST 清单

7.2 重复汇编

有时汇编语言程序需要连续地重复完成相同的或者几乎完全相同的一组代码，这时可使用重复汇编。

7.2.1 重复伪操作

其格式为：

```
REPT    expression
  ⋮       (重复块)
ENDM
```

其中表达式的值用来确定重复块的重复次数，表达式中如包含外部或未定义的项则汇编指示出错。

重复伪操作并不一定要用在宏定义体内。下面举例说明重复伪操作的使用方法。

例 7.13

```
X=0
        REPT  10
X=X+1
        DB    X
        ENDM
```

则汇编后产生

```
+       DB    1
+       DB    2
+       DB    3
        ⋮
+       DB    10
```

例 7.14　把字符 A 到 Z 的 ASCII 码填入数组 TABLE。

```
CHAR='A'
TABLE   LABEL   BYTE
        REPT    26
```

```
        DB      CHAR
CHAR=CHAR+1
        ENDM
```

经汇编产生

```
+       DB      61H
+       DB      62H
        ⋮
+       DB      7AH
```

例 7.15 用宏定义及重复伪操作把 TAB,TAB+1,TAB+2,...,TAB+16 的内容存入堆栈。

宏定义：

```
PUSH_TAB    MACRO   K
            PUSH    TAB+K
            ENDM
```

宏调用：

```
I=0
            REPT    17
            PUSH_TAB %I
I=I+1
            ENDM
```

宏展开：

```
+           PUSH    TAB+0
+           PUSH    TAB+1
            ⋮
+           PUSH    TAB+16
```

例 7.16 要求建立一个 100D 字的数组，其中每个字的内容是下一个字的地址，而最后一个字的内容是第一个字的地址。

```
ARRAY       LABEL   WORD
            REPT    99
            DW      $+2
            ENDM
            DW      ARRAY
```

经汇编后得

```
+           DW      $+2
+           DW      $+2
            ⋮                 } 99 个字
+           DW      $+2
            DW      ARRAY
```

7.2.2 不定重复伪操作

7.2.2.1 IRP 伪操作

格式是：

```
IRP     dummy,<argument list>
  ⋮       (重复块)
ENDM
```

汇编程序把重复块的代码重复几次，每次重复把重复块中的哑元用自变量表中的一项来取代，下一次取代下一项，重复次数由自变量表中的自变量个数来确定。自变量表必须用尖括号括起，它可以是常数、符号、字符串等。

IRP 和下面要讲到的 IRPC 都和 REPT 一样，不一定要用在宏定义中。下面用例子说明其使用方法。

例 7.17

```
IRP     X,<1,2,3,4,5,6,7,8,9,10>
DB      X
ENDM
```

汇编后得：

```
+       DB      1
+       DB      2
          ⋮
+       DB      10
```

例 7.18

```
IRP     REG,<AX,BX,CX,DX>
PUSH    REG
ENDM
```

汇编后得：

```
+       PUSH    AX
+       PUSH    BX
+       PUSH    CX
+       PUSH    DX
```

7.2.2.2 IRPC 伪操作

格式为：

```
IRPC    dummy, string (或<string>)
  ⋮       (重复块)
ENDM
```

IRPC 和 IRP 类似，但自变量表必须是字符串。重复次数由字符串中的字符个数确定，每次重复用字符串中的下一个字符取代重复块中的哑元。举例说明如下：

例 7.19

```
IRPC    X,0 1 2 3 4 5 6 7
DB      X+1
ENDM
```

汇编后得：

```
+       DB      1
+       DB      2
          ⋮
+       DB      8
```

例 7.20

```
        IRPC    K,A B C D
        PUSH    K & X
        ENDM
```

汇编后展开成：

```
+       PUSH    AX
+       PUSH    BX
+       PUSH    CX
+       PUSH    DX
```

7.3 条件汇编

汇编程序能根据条件把一段源程序包括在汇编语言程序内或者把它排除在外，这里就用到条件伪操作。条件伪操作的一般格式是：

```
IF××                    argument
 ⋮                    } 自变量满足给定条件汇编此块
[ELSE]
 ⋮                    } 自变量不满足给定条件汇编此块
ENDIF
```

自变量必须在汇编程序第一遍扫视后就成为确定的数值(有关汇编程序的工作情况我们将在第十三章中说明)。条件伪操作中的××表示条件如下：

IF expression	汇编程序求出表达式的值，如此值不为 0 则满足条件。
IFE expression	如求出表达式的值为 0 则满足条件。
IFDEF symbol	如符号已在程序中定义，或者已用 EXTRN 伪操作说明该符号是在外部定义的，则满足条件。
IFNDEF symbol	如符号未定义或未通过 EXTRN 说明为外部符号则满足条件。
IFB (argument)	如自变量为空则满足条件。
IFNB (argument)	如自变量不为空则满足条件。
IFIDN (arg—1),(arg—2)	如果字符串(arg—1)和字符串(arg—2)相同，则满足条件。
IFDIF (arg—1),(arg—2)	如果字符串(arg—1)和字符串(arg—2)不相同，则满足条件。

条件伪操作可以用在宏定义体内，也可以用在宏定义体外，也允许嵌套任意次。下面举例说明条件伪操作的使用方法。

例 7.21 宏指令 MAX 把三个变元中的最大值放在 AX 中，而且使变元数不同时产生不同的程序段。

宏定义：

```
MAX     MACRO   K,A,B,C
        LOCAL   NEXT,OUT
        MOV     AX,A
        IF      K—1
        IF      K—2
        CMP     C,AX
```

```
            JLE     NEXT
            MOV     AX,C
            ENDIF
NEXT:       CMP     B,AX
            JLE     OUT
            MOV     AX,B
            ENDIF
OUT:
            ENDM
```

宏调用：

```
            MAX     1,P
            MAX     2,P,Q
            MAX     3,P,Q,R
```

宏展开：

```
            MAX     1,P
+           MOV     AX,P
+?? 0001:
            MAX     2,P,Q
+           MOV     AX,P
+?? 0002:   CMP     Q,AX
+           JLE     ?? 0003
+           MOV     AX,Q
+?? 0003 :
            MAX     3,P,Q,R
+           MOV     AX,P
+           CMP     R,AX
+           JLE     ?? 0004
+           MOV     AX,R
+?? 0004:   CMP     Q,AX
+           JLE     ?? 0005
+           MOV     AX,Q
+?? 0005:
```

例 7.22 宏指令 GOTO L,X,REL,Y（其中 REL 可以是 Z,NZ,L,NL 等）可以根据不同情况产生无条件转移指令或比较和条件转移指令。

宏定义：

```
GOTO        MACRO   L,X,REL,Y
            IFB     <REL>
            JMP     L
            ELSE
            MOV     AX,X
            CMP     AX,Y
            J&REL   L
            ENDIF
            ENDM
```

宏调用：

```
                ⋮
        GOTO    LOOP,SUM,NZ,15
                ⋮
        GOTO    EXIT
                ⋮
```

宏展开：

```
                ⋮
+       MOV     AX,SUM
+       CMP     AX,15
+       JNZ     LOOP
                ⋮
+       JMP     EXIT
```

例 7.23 宏定义可允许递归调用，此时条件伪操作可用来结束宏递归。

宏指令 POWER 可以用来实现 X 和 2^N 相乘。这只需对 X 左移 N 次即可实现，可以设 COUNT 为递归次数的计数值，当该数与 N 相等时就可结束递归调用。

宏定义：

```
POWER   MACRO   X,N
        SAL     X,1
COUNT=COUNT+1
        IF      COUNT-N
        POWER   X,N
        ENDIF
        ENDM
```

宏调用：

```
COUNT=0
        POWER   AX,3
```

宏展开：

```
+       SAL     AX,1
+       SAL     AX,1
+       SAL     AX,1
```

例 7.24 宏指令 BRANCH 产生一条转向 X 的转移指令。当它相对于 X 的距离小于 128 字节时产生 JMP SHORT X；否则产生 JMP NEAR PTR X (X 必须位于该转移指令之后，即低地址区)。

宏定义：

```
BRANCH  MACRO   X
        IF      ($-X) LT 128
                JMP     SHORT X
        ELSE
                JMP     NEAR PTR X
        ENDIF
        ENDM
```

宏调用：

```
        BRANCH    X
```

宏展开：

如 X 与 BRANCH 指令间的距离小于 128 时产生

```
+       JMP       SHORT X
```

否则产生

```
+       JMP       NEAR PTR X
```

读者可以看出在本例的宏定义中使用了逻辑操作符，这在一些宏定义中经常用到，希望读者能逐步熟悉它的使用方法。

习　题

7.1　编写一条宏指令 CLRB，完成用空格符将一字符区中的字符取代的工作。字符区首地址及其长度为变元。

7.2　某工厂计算周工资的方法是每小时的工资率 RATE 乘以工作时间 HOUR，另外每工作满十小时加奖金 3 元，工资总数存放在 WAG 中。请将月工资的计算编写成一条宏指令 WAGES，并展开宏调用：

```
        WAGES     R1,42
```

7.3　给定宏定义如下：

```
DIF     MACRO     X,Y
        MOV       AX,X
        SUB       AX,Y
        ENDM
ABSDIF  MACRO     V1,V2,V3
        LOCAL     CONT
        PUSH      AX
        DIF       V1,V2
        CMP       AX,0
        JGE       CONT
        NEG       AX
CONT:   MOV       V3,AX
        POP       AX
        ENDM
```

试展开以下调用，并判定调用是否有效。

(1)　ABSDIF　P1,P2,DISTANCE

(2)　ABSDIF　[BX],[SI],X[DI],CX

(3)　ABSDIF　[BX] [SI],X[BX][SI],240H

(4)　ABSDIF　AX,AX,AX

7.4　试编制宏定义，要求把存储器中的一个用 EOT 字符结尾的字符串传送到另一个存储区去。

7.5　宏指令 BIN_SUB 完成多个字节数据连减的功能：

RESULT←(A-B-C-D-…)

要相减的字节数据顺序存放在首地址为 OPERAND 的数据区中，减数的个数存放在 COUNT 单元中，最后结果存入 RESULT 单元。请编写此宏指令。

7.6　请用宏指令定义一个可显示字符串 GOOD:'GOOD STUDENTS ' CLASSX NAME'，其中 X 和 NAME 在宏调用时给出。

7.7　下面的宏指令 CNT 和 INC 完成相继字存储。

```
CNT     MACRO     A,B
A&B     DW        ?
        ENDM
INC     MACRO     A,B
        CNT       A,%  B
B=B+1
        ENDM
```

请展开下列宏调用，并说明宏指令 INC 和机器指令 INC 的差别。

```
C=0
        INC       DATA,C
        INC       DATA,C
```

7.8 定义宏指令并展开宏调用。宏指令 JOE 把一串信息'MESSAGE NO. K'存入数据存储区 XK 中。宏调用为：

```
I=0
        JOE       TEXT,I
          ⋮
        JOE       TEXT,I
          ⋮
        JOE       TEXT,I
```

7.9 大多数DOS功能调用都需要在AH寄存器中存放不同的功能码。请将这种功能调用定义成宏指令DOS21。再定义宏指令DISP完成显示字符的功能，其中可使用已定义的DOS21。然后展开宏调用 DISP '*'。

7.10 宏指令 STORE 定义如下：

```
STORE   MACRO     X,N
        MOV       X+I,I
I=I+1
        IF        I-N
        STORE     X,N
        ENDIF
        ENDM
```

试展开下列调用：

```
I=0
        STORE     TAB,7
```

7.11 试编写非递归的宏指令，使其完成的工作与 7.10 题的 STORE 相同。

7.12 试编写一段程序完成以下功能，如给定名为 X 的字符串长度大于 5 时，下列指令将汇编 10 次。

```
        ADD       AX,AX
```

7.13 定义宏指令 FINSUM，比较两个数 X 和 Y，若 X>Y 则执行 SUM←X+2*Y；否则执行 SUM←2*X+Y

7.14 试编写一段程序完成以下功能：如变元 X='VT55'，则汇编 MOV TERMINAL,0；否则汇编 MOV TERMINAL,1。

7.15 对于 DOS 功能调用，所有的功能调用都需要在 AH 寄存器中存放功能码，而其中有一些功能需要在 DX 中放一个值。试定义宏指令 DOS21，要求只有在程序中定义了缓冲区时，汇编为：

```
        MOV       AH,DOSFUNC
        MOV       DX,OFFSET BUFF
```

```
        INT     21H
```

否则，无 MOV DX,OFFSET BUFF 指令。并展开以下宏调用：

```
        DOS21   01
        DOS21   0AH,IPFIELD
```

7.16 编写一段程序，使汇编程序根据 SIGN 中的内容分别产生不同的指令。

如果(SIGN)=0，则用字节变量 DIVD 中的无符号数除以字节变量 SCALE；如果(SIGN)=1，则用字节变量 DIVD 中的带符号数除以字节变量 SCALE，结果都存放在字节变量 RESULT 中。

第八章　输入/输出程序设计

在广泛使用的微型机系统中，外部设备是以实现人机交互和机间通讯为目的的一些机电设备。计算机系统通过硬件接口以及I/O控制程序对外部设备进行控制，使其能协调地、有效地完成输入输出工作。在对外部设备的控制过程中，主机不可避免地，有时甚至要很频繁地对设备接口进行联络和控制，因此能直接控制硬件的汇编语言就成了编写高性能I/O程序最有效的程序设计语言。本章我们将以一些常用的I/O设备为例，着重讨论I/O程序设计的几种方法。

8.1　I/O设备的数据传送方式

每种输入输出设备都要通过一个硬件接口或控制器和CPU相连。例如软磁盘通过软盘控制器和CPU连接起来；终端显示器通过数据接口和CPU连接起来。这些接口和控制器都能支持输入输出指令IN、OUT与外部设备交换信息。这些信息包括控制、状态和数据三种不同性质的信息，它们必须按不同的端口地址分别传送。

控制信息输出到I/O接口，告诉接口和设备要做什么工作；从接口输入的状态信息表示I/O设备当前的状态；数据信息则是I/O设备和CPU真正要交换的信息。不同的I/O设备要求传送的数据类型也是不同的，例如和终端显示器交换的数据必须是ASCII码，而不能是二进制表示的数。

IBM PC具有一系列简单而又灵活的输入/输出方式。在第三章已经提到一种用IN和OUT指令直接在端口级上处理输入输出的程序直接控制I/O的方式，另外还有中断传送方式和DMA方式。程序直接控制输入输出方式和中断传送方式我们在后面专门要讲到，而DMA方式因为主要由硬件DMA控制器实现其传送功能，所以只在这里做一点简单介绍。

DMA方式即直接存储器存取(Direct Memory Access)方式，也称为成组数据传送方式。主要适用于一些高速的I/O设备，如磁带、磁盘、模数转换器(A/D)等设备。这些设备传输字节或字的速率非常快，如磁盘的数据传输率约为每秒200,000字节，也就是说磁盘与存储器传送一个字节只需5微秒。对这类高速I/O设备，用执行输入输出指令的方法或完成一次次中断序列的方法来传输字节，将会造成数据的丢失，而DMA方式能使I/O设备直接和存储器进行成批数据的快速传送。每个字节一到达端口，就直接从接口送到存储器，同样，接口和它的DMA控制器也能直接从存储器取出字节并把它送到I/O设备中去。

DMA控制器或接口一般包括四个寄存器：状态控制寄存器，数据寄存器，地址寄存器和字节计数器，这些寄存器能在信息传送之前进行初始化。每个字节传送后，地址寄存器增1，字节计数器减1。

IBM PC机完成DMA传送的步骤如下：

1. DMA控制器向CPU发出HOLD信号请求使用总线。

2. CPU发出响应信号HLDA给DMA控制器，并将总线让出，这时CPU不再使用总线，而DMA控制器获得总线控制权。

3. 传输数据的存储器地址(在地址寄存器中)通过地址总线发出。
4. 传输的数据字节通过数据总线进行传送。
5. 地址寄存器增 1。
6. 字节计数器减 1。
7. 如字节计数器非 0,转向第 3 步。
8. 否则,DMA 控制器撤消总线请求信号 HOLD,传送结束。

8.2 程序直接控制 I/O 方式

8.2.1 I/O 端口

计算机的外部设备和大容量存储设备都是通过接口连接到系统上,每个接口由一组寄存器组成,这些寄存器都分配有一个称为 I/O 端口的地址编码。计算机的 CPU 和内存就是通过这些端口和外部设备进行通讯的。

I/O 接口中有用作数据缓冲的数据寄存器;有用做保存设备和接口的状态信息,供 CPU 对外设进行测试的状态寄存器;还有用来保存 CPU 发出的命令以控制接口和设备的操作的命令寄存器。它们都分配有各自的端口号,CPU 就是通过不同的端口号来选择外部设备的。

在 IBM PC 机中,I/O 端口编址在一个独立的地址空间中,这个 I/O 空间允许设置 64K (65536)个 8 位端口或 32K(32768)个 16 位端口,这些端口地址实际上只用了其中很小一部分,因为系统中一般只有十几个外部设备和大容量存储设备和 PC 机相连。对不同型号的计算机及其接口,I/O 端口的编号有时不完全相同。表 8.1 列出了 PC 机的部分端口地址(16 进制)。

表 8.1 IBM PC I/O 端口地址分配

00—0F	DMA 芯片 8237A
20—21	中断控制器 8259A
40—43	时钟/定时器
60—63	可编程序外围接口芯片(PPI)8255A
200—20F	游戏适配器
320—32F	硬磁盘控制器
378—37A	并行接口打印机适配器
3B0—3BF	单色显示和并行打印机适配器
3D0—3DF	彩色/图形适配器
3F0—3F7	软磁盘控制器
3F8—3FE	异步通讯适配器(Primary)
2F8—2FE	异步通讯适配器(Alternate)

8.2.2 I/O 指令

对于一个 I/O 和存储器分离的地址空间系统,IBM PC 有专门的 I/O 指令与端口进行通信。在指令系统一章,我们已介绍了 PC 机的 I/O 指令 IN 和 OUT,这两条指令既可以传送字节

也可以传送字，并且都有直接端口寻址和间接端口寻址两种方式。

```
IN   AL,PORT     (AL)←(PORT)
IN   AX,PORT     (AX)←(PORT+1：PORT)
IN   AL,DX       (AL)←((DX))
IN   AX,DX       (AX)←((DX)+1：(DX))

OUT PORT,AL      (PORT)←(AL)
OUT PORT,AX      (PORT+1：PORT)←(AX)
OUT DX,AL        ((DX))←(AL)
OUT DX,AX        ((DX)+1：(DX))←(AX)
```

以上 IN 和 OUT 指令的前两种方式是直接端口寻址方式，端口地址 PORT 是一个 8 位的立即数，其范围是 0—255。两组指令中的后两种格式是间接寻址方式，端口地址在 DX 中，其范围为 0—65535，这种方式通过对 DX 寄存器的增量可以处理几个连续端口地址的输入/输出。另外要注意的是，I/O 指令中使用的寄存器必须是 AL 或 AX。

用 IN 指令可以从一个数据寄存器输入数据或从状态寄存器输入接口和外设的状态。例如下面两条指令能把一个字从端口地址 0028 和 0029 传送到存储器的 DATA—WORD 单元中。

```
IN    AX,28H
MOV   DATA_WORD,AX
```

又例如，测试某状态寄存器(端口地址为 27H)的第 2 位是否为 1，若为 1，则转移到 ERROR 进行处理。其指令序列为：

```
IN    AL,27H
TEST  AL,00000100B
JNZ   ERROR
```

同样，OUT 指令用来输出数据或给一个指定的 I/O 端口传送命令信息。例如，某接口的命令寄存器(端口地址为 126H)的第 7 位控制成组数据传送，那么下面的指令序列将发出成组传送命令：

```
MOV   DX,126H
IN    AL,DX
OR    AL,80H
OUT   DX,AL
```

I/O 指令是与外部设备进行通信的最基本途径，即使使用 DOS 功能调用或 BIOS 例行程序，其例行程序本身也是用 IN 和 OUT 指令与外部设备进行数据交换的。例如，当程序请求从键盘输入字符时，程序中安排了一条中断指令 INT 16H，当执行这条指令时，系统将调用 ROM BIOS 的一个键盘例行程序，在这个例行程序中就有一条 IN 指令从端口 60H 输入一个字符到 AL 寄存器。

使用 I/O 指令对端口地址进行直接的输入或输出，比调用 DOS 功能或 BIOS 例行程序更能提高数据的传送速度和吞吐量，但同时也要求程序员对计算机的硬件结构有一定的了解，其程序对硬件的依赖性也大，因此，对于一般的程序设计，我们还是尽可能使用 DOS 或 BIOS 功能调用。

8.2.3 I/O 程序举例

下面我们通过几个 I/O 程序的例子，说明使用 I/O 指令直接在端口级上输入输出的方法。

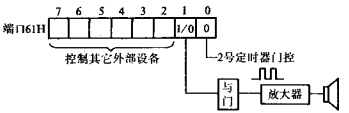


图 8.1 设备控制寄存器

例 8.1 Sound 程序

这是一个最基本的直接控制扬声器发出声音的程序。程序通过 I/O 指令使设备控制寄存器(I/O 端口地址为 61H)的第 1 位交替为 0 和 1,而端口 61H 的第 1 位和扬声器的脉冲门相连,当第 1 位由 0 变为 1,延迟一会又由 0 变为 1 时,脉冲门就先打开后关闭,产生了一个脉冲电流。这个脉冲电流被放大后送到扬声器使之发出了声音。61H 端口的第 0 位和一个振荡器(2 号定时器)相连,现在不用振荡器产生声音,所以把第 0 位置零。

图 8.2 是控制扬声器发声的 Sound 程序。

```
;sound--Makes a sound with the speaker
prognam   segment
main      proc      far
          assume cs:prognam,ds:prognam
          org       100h
start:    mov       dx, 100          ;turn on /off 100 times
          in        al, 61h          ;get port 61h
          and       al, 11111100b    ;AND off bits 0,1
sound:    xor       al, 2            ;toggle bit 1
          out       61h,al           ;output to port 61h
          mov       cx,140h          ;value of wait
wait1:    loop      wait1            ;delayed a while
          dec       dx               ;total 100 times
          jne       sound
          int       20h
main      endp
prognam   ends
          end       start
```

图 8.2 Sound 程序

程序中的第 2 条指令 IN AL,61H 取得设备控制寄存器的开关量,然后由第 3 条指令 AND 将第 0 位和第 1 位置零,2—7 位保持不变,XOR 指令将第 1 位置为 1,然后把这个开关量输出到 61H 端口以控制接通扬声器。在第二次循环执行 XOR 指令时,第 1 位又由 1 变为 0,也就是关闭了扬声器,这样在脉冲电流的驱动下,扬声器就发出了声音。

另外两条指令:

```
MOV  CX,140h
WAIT1: LOOP  WAIT1
```

是用来控制脉冲门开关的时间,这个时间值是可以改变的。这里送了一个固定的数值 140h,因此扬声器接通和关闭的时间间隔总是相同的,结果发出的声音是一个没有频率变化的纯音。

通常一个外设的数据端口是 8 位的,而状态与控制信息只需一位或两位,所以不同外设的

状态和控制位可共用一个端口。61H 端口的 0、1 位是控制扬声器的，2--7 位分别控制其它外部设备。

例 8.2 COMM 程序

这是一个关于 INS 8250 串行通讯口 I/O 的例子。它的数据寄存器的端口地址是 03F8H，状态寄存器的端口地址是 03FDH，其中 0 位是输入数据准备位，5 位是输出数据准备位。图 8.3 是串行口输入输出程序。

```
;-----------------------------------------
COM_IN    PROC    FAR
          PUSH    DX
          MOV     DX,03FDH          ;line states register
COM_IN1 :
          IN      AL,DX
          TEXT    AL,01             ;data ready ?
          JE      COM_IN1
          MOV     DX,03F8H          ;receive buffer register
          IN      AL,DX             ;receive a character
          POP     DX
          RET
COM_IN    ENDP
;-----------------------------------------
;-----------------------------------------
COM_OUT   PROC    FAR
          PUSH    DX
          PUSH    AX
          MOV     DX,03FDH          ;line status register
COM_OUT1 :
          IN      AL,DX
          TEST    AL,20H            ;holding reg empty ?
          JE      COM_OUT1
          POP     AX
          MOV     DX,03F8H          ;holding register
          OUT     DX,AL             ;serially transmit a char
          POP     DX
          RET
COM_OUT   ENDP
;-----------------------------------------
```

图 8.3 串行通讯口的查询程序

我们可以看到在这两段程序中，都使用了 TEST 指令对状态寄存器(端口地址为 03FDH)的 0 位或 5 位进行反复的测试。如果 0 位为 1，表示接收数据准备好，如果 0 位为 0，则说明数据还未准备好，需要再等待，这时由转移指令 JE 控制返回执行 TEST 指令，再测试这位的状态，直到 0 位变为 1，CPU 才从数据寄存器(I/O 端口地址为 03F8H)接收数据到 AL 寄存器。同样，输出数据到串行口，必须反复测试状态寄存器的 5 位，若为 0 就查询等待，直到 5 位由 0 变

1 时，就用 OUT 指令将 AL 寄存器中的数据送到串行口的数据寄存器。

这种 CPU 与外部设备交换信息的方式称为查询方式或等待方式。

造成 CPU 必须查询等待的主要原因是许多外设的工作速度比较低，如键盘、打印机等，它们通过按键或打印头的机械动作输入或输出一个数据，其速度是很慢的，而 CPU 执行指令的速度是它的几千倍乃至上万倍，所以 CPU 在接收或发送数据之前必须要了解外设的状态，看外设是否已经准备好。当外设还没有准备好以前，CPU 就要等待，不能做别的操作。为了提高 CPU 的工作效率，我们可采用中断方式传送数据。关于中断，我们将在下一节中做详细的介绍。

有时系统中同时有几个设备要求输入输出数据，那么对每个设备都编写一段执行输入输出数据的程序，然后轮流查询这些设备的准备位。当某一设备准备好允许输入或输出数据时，就调用这个设备的 I/O 程序完成数据传输，否则依次查询下一个设备是否准备好。

下面我们再看一个程序例子，CPU 要从 3 个设备轮流输入数据。PROC1，PROC2，PROC3 分别是设备 1，设备 2 和设备 3 的数据输入程序，它们的状态寄存器的端口地址分别用 STAT1，STAT2，STAT3 表示，这三个状态寄存器的第 5 位是输入准备位。

例 8.3 轮流查询三个数据输入设备的程序段。

```
;Round_robin polling
INPUT:  IN      AL,STAT1        ;check device 1
        TEST    AL,20H          ;if device 1 is ready
        JZ      DEV2            ;no,goto divice 2
        CALL    FAR PTR PROC1   ;yes,device 1 input data
DEV2:   IN      AL,STAT2        ;check device 2
        TEST    AL,20H          ;if device 2 is ready
        JZ      DEV3            ;no,goto device 3
        CALL    FAR PTR PROC2   ;yes,device 2 input data
DEV3:   IN      AL,STAT3        ;check device 3
        TEST    AL,20H          ;if device 3 is ready
        JZ      NO_INPUT        ;no,goto no input
        CALL    FAR PTR PROC3   ;yes,device 3 input data
NO_INPUT:
        ⋮
```

图 8.4 轮流查询多个设备的指令序列

查询方式的优点是，可以用程序安排几个输入/输出设备的先后优先次序，最先查询的设备，其工作的优先级也最高。修改程序中的查询次序，实际上也就修改了设备的优先级。查询方式的缺点就是前面提到的在查询过程中，浪费了 CPU 原本可执行大量指令的时间，而且由询问转向相应的处理程序的时间较长，尤其在设备比较多的情况下。

8.3 中断传送方式

中断是 CPU 和外部设备进行输入/输出的有效方法。这种输入/输出方式一直被大多数计算机所采用，它可以避免因反复查询外部设备的状态而浪费时间，从而提高了 CPU 的效率。

中断是一种使 CPU 中止正在执行的程序而转去处理特殊事件的操作。这些引起中断的事件称为中断源，它们可能是来自外设的输入输出请求，也可能是计算机的一些异常事故或其它内部原因。由外设控制器或协处理器(8087/80287)引起的中断一般称为外中断，由程序中安排的中断指令 INT 产生的中断，或由 CPU 的某些错误结果产生的中断称为内中断。8086/8088 的中断源如图 8.5 所示。图中引线端标示的数字为分配的中断类型号。

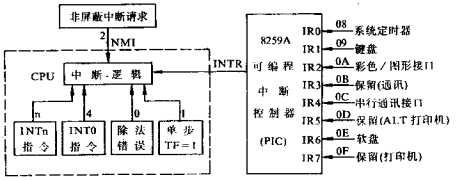


图 8.5 8086/8088 中断源

连到 CPU 的非屏蔽中断(NMI)是为电源错、内存或 I/O 总线的奇偶错等异常事件的中断保留的。外部设备的中断是通过 Intel 8259A 可编程中断控制器(PIC)连到主机上。CPU 通过一组 I/O 端口控制 8259A，而 8259A 则通过 INTR 管脚给 CPU 传送中断信号。多个 8259A PIC 可以树形结构联在一起，从而使大量的外部设备顺序连接到 CPU 的中断系统上。外部设备和 8259A PIC 的连法是由设计人员规定好的，这种外中断类型的分配由硬件连线实现，因而软件不能对其修改。图 8.5 中外设与 8259A PIC 的连法是 IBM PC/XT 的连法。内中断不是由连线接到硬件上的，中断 20H 到 3FH 用于调用 DOS 功能例行程序，其它中断号小于 20H 或大于 3FH 的中断，用于调用 IBM PC ROM BIOS 或一些应用软件，这些内容在以后的章节中将要陆续讲到。

8.3.1 中断向量表

我们给每种中断都安排一个中断类型号。IBM PC 中断系统能处理 256 种类型的中断，类型号为 0—0FFH。如图 8.5 所示的中断源，系统定时器的中断类型为 08，键盘为 09，内中断中的除法错误的中断类型为 0，等等。每种类型的中断都由相应的中断处理程序来处理，中断向量表就是各中断类型的处理程序的地址表。

我们知道存储器的低 1.5K 字节，地址从 0 到 5FFH 为系统占用，其中最低的 1K 字节，地址从 0 到 3FFH 存放中断向量。中断向量表中的 256 项中断向量对应 256 种中断类型，每项占用四个字节，其中两个字节存放中断处理程序的段地址(16 位)，另两个字节存放偏移地址(16 位)。因为各处理程序的段地

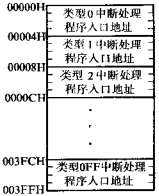


图 8.6 中断向量表

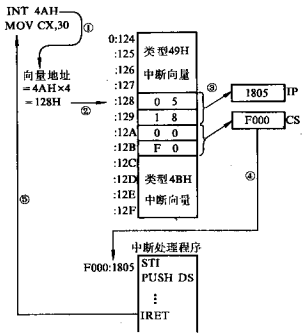


图 8.7 中断操作步骤

址和偏移地址在中断向量表中按中断类型号顺序存放(如图 8.6 所示),所以每类中断向量的地址可由中断类型号乘以 4 计算出来。例如,报警中断的中断类型为 4AH,它的中断向量地址为 4AH×4=128H,即 128H,129H 两字节存放的是报警中断处理程序的偏移地址,12AH,12BH 两字节存放的是报警中断处理程序的段地址,取出段地址和偏移地址,CPU 就可转入中断处理程序。图 8.7 以 BIOS 中断 INT 4AH 为例,表示出中断操作的 5 个步骤:

(1) 取中断类型号

(2) 计算中断向量地址

(3) 取中断向量,偏移地址送 IP,段地址送 CS

(4) 转入中断处理程序

(5) 中断返回到 INT 指令的下一条指令

采用向量中断的方法,大大加快了中断处理的速度,因为计算机可直接通过中断向量表转向相应的处理程序,而不需要 CPU 去逐个检测和确定中断原因。

8.3.2 设置中断向量

表 8.2 列出了 IBM PC 各类中断的地址分配。

表 8.2 中断向量分配

地址	中断		地址	中断	
0—7F	0—1F	BIOS 中断向量	1C0—1DF	70—77	I/O 中断向量
80—FF	20—3F	DOS 中断向量	1E0—1FF	78—7F	保留
100—17F	40—5F	扩充 BIOS 中断向量	200—3C3	80—FD	BASIC 中断向量
180—19F	60—67	用户中断向量	3C4—3FF	F1—FF	保留
1A0—1BF	68—6F	保留			

用户可以利用保留的中断类型号扩充自己需要的中断功能,对新增加的中断功能要在中断向量表中建立相应的中断向量。我们已经讨论了中断类型和中断向量地址的对应关系,下面我们用指令来为中断类型 N 设置中断向量:

```
MOV     AX,0
MOV     ES,AX              ;set to base of interrupt vectors
MOV     BX,N*4             ;offset of type N interrupt
```

```
        MOV     AX,OFFSET INTHAND
        MOV     ES : WORD PTR[BX],AX    ;set addr of INTHAND
        MOV     AX,SEG INTHAND
        MOV     ES : WORD PTR[BX+2],AX
        ⋮
INTHAND :                               ;interrupt processing routine
        ⋮
        IRET
```

如果新的中断功能只供自己使用，或用自己编写的中断处理程序代替系统中的中断处理功能时，要注意保存原中断向量，在设置自己的中断向量时，应先保存原中断向量再设置新的中断向量，在程序结束之前恢复原中断向量。

实际上，我们在检查或设置任何中断向量时，总是避免直接使用中断向量的绝对地址，而是使用 DOS 功能调用(21H)存取中断向量。

设置中断向量

把由 AL 指定的中断类型的中断向量

DS : DX 放置在中断向量表中

预置 : AH＝25H

AL＝中断类型号

DS : DX＝中断向量

执行 : INT 21H

取中断向量

把由 AL 指定的中断类型的中断向量

从中断向量表中取到 ES : BX 中

预置 : AH＝35H

AL＝中断类型号

执行 : INT 21H

返回时送 : ES : BX＝中断向量

例 8.4 使用 DOS 功能调用存取中断向量

```
        ⋮
        MOV     AL,N                ;type N interrupt
        MOV     AH,35H              ;get interrupt vector
        INT     21H
        PUSH    ES                  ;save the old base and
        PUSH    BX                  ; offset of interrupt N
        PUSH    DS
        MOV     AX,SEG INTHAND
        MOV     DS,AX               ;base of INTHAND in DS
        MOV     DX,OFFSET INTHAND   ;offset in DX
        MOV     AL,N                ;type N
```

```
        MOV     AH,25H              ;set interrupt vector
        INT     21H
        POP     DS
        ⋮
        POP     DX                  ;restore the old offset
        POP     DS                  ; and base of interrupt
        MOV     AL,N                ;type N
        MOV     AH,25H              ;set interrupt vector
        INT     21H
        RET                         ;return
;
INTHAND:                            ;interrupt processing routine
        ⋮
        IRET
```

8.3.3 中断过程

当中断发生时，由硬件自动完成下列动作：

1. 取中断类型号 N
2. 标志寄存器(PSW)内容入栈
3. 当前代码段寄存器(CS)内容入栈
4. 当前指令计数器(IP)内容入栈
5. 禁止外部中断和单步中断(IF=0,TF=0)
6. 从中断向量表中取 4×N 的字节内容送 IP，取 4×N+2 中的字节内容送 CS
7. 转中断处理程序

中断发生的过程很象我们所熟悉的子程序调用，不同的是在保护中断现场时，除了保存返回地址 CS：IP 之外，还保存了标志寄存器 PSW 的内容。因为标志寄存器记录了中断发生时，程序指令运行的结果特征，当 CPU 处理完中断请求返回原程序时，要保证原程序工作的连续性和正确性，所以中断发生时 PSW 的内容也要保存起来。另一个不同点是，在中断发生时，CPU 还自动清除了 IF 位和 TF 位，这样设计的目的是使 CPU 转入中断处理程序后，不允许再产生新的中断，如果在执行中断处理程序的过程中，还允许外部的中断，可通过 STI 指令再把 IF 置为 1。

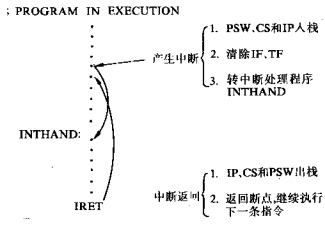


图 8.8 中断过程

编写中断处理程序和编写子程序一样，所使用的汇编语言指令没有特殊限制，只是中断处理程序返回时使用 IRET 指令。这条指令的工作步骤和中断发生时的

工作步骤正好相反。它首先把 IP、CS 和 PSW 的内容出栈，然后返回到中断发生时紧接着的下一条指令。图 8.8 为中断过程的示意图。

8.3.4 内中断

内中断又称为软件中断，它通常由三种情况引起：

1. 由中断指令 INT 引起。
2. 由于 CPU 的某些错误而引起。
3. 为调试程序(DEBUG)设置的中断。

下面分别介绍这几种内中断。

8.3.4.1 中断指令 INT 引起的内中断

CPU 执行完一条 INT n 指令后，会立即产生中断，并且调用系统中相应的中断处理程序来完成中断功能，中断指令的操作数 n 指出中断类型。

例如，我们要对存储器的容量进行测试，可在程序中安排一条中断指令：

```
INT   12H
```

当 CPU 执行这条指令时，立即产生了一个中断，并从中断向量表的 0：46H 开始的四个字节单元中取出段地址和偏移地址，然后转去执行相应的中断处理程序以完成对存储器的测试，返回调用程序后，AX 寄存器中的数据即为存储器的大小。

INT 指令可以指定 0—0FFH 中的任何类型号。除系统占用的类型号之外，用户还可利用为用户保留的类型号扩充新的中断处理功能。

8.3.4.2 处理 CPU 某些错误的中断

CPU 在执行程序时，还会发现一些运算中出现的错误，为了能及时处理这些错误，CPU 就以中断的方式中止正在运行的程序，待程序员改正错误后，重新运行程序。

1. 除法错中断

除法错的中断类型为 0。

在执行除法指令时，若发现除数为 0 或商超过了寄存器所能表达的范围，则立即产生一个类型为 0 的中断。

2. 溢出中断

如果溢出标志 OF 置 1，有一条专门的指令 INTO 来中断发生溢出的算术操作，如 OF 为 0，则 INTO 指令不产生中断，CPU 继续运行原程序。溢出中断处理程序的主要功能是打印出一个出错信息，在处理程序结束时，不返回原程序继续运行，而是把控制交给操作系统。

溢出中断的中断类型为 4，下面的指令用来测试加法的溢出：

```
ADD AX,VALUE
INTO
```

8.3.4.3 为调试程序(DEBUG)设置的中断

一个新的程序编制好以后，必须要上机调试才能正确可靠地工作。在调试程序时，为了检查中间结果或寻找程序中的问题，往往要在程序中设置断点或进行单步工作(一次只执行一条指令)，这些功能都是由中断系统来实现的。

1. 单步中断

单步是一种很有用的调试方法。当标志位 TF 置为 1 时，每条指令执行后，CPU 自动产生类型 1 的中断——单步中断。产生单步中断时，CPU 同样自动地将 PSW、CS 和 IP 的内容保存

入栈，然后清除 TF，IF，于是，当进入单步中断处理程序后，就不是处于单步方式了，它将按正常方式运行中断处理程序。在单步处理程序结束时，原来的 PSW 从栈中取回，又把 CPU 重新置成单步方式。

使用单步中断可以一条指令一条指令地跟踪程序的流程，观察 CPU 每执行一条指令后，各个寄存器及有关存储单元的变化，从而指出和确定产生错误的原因。

2. 断点中断

断点中断也是供 DEBUG 调试程序使用的，它的中断类型为 3。通常调试程序时，把程序按功能分成几段，然后每段设一个断点。当 CPU 执行到断点时便产生中断，这时程序员可以检查各寄存器及有关存储单元的内容。

断点可以设置在程序的任何地方，设置断点实际上是把一条断点指令 INT 3 插入程序中，CPU 每执行到断点处的 INT 3 指令便产生一个中断。

在上述内中断中，INT 指令和 INTO 指令产生的中断，以及除法错中断都不能被禁止，并且比任何外部中断的优先权都高。

8.3.5 外中断

外中断来自处理机的外部条件，如 I/O 设备或其它处理机等，以完全随机的方式中断现行程序而转向另一处理程序。

从图 8.5 可以看出，外部中断主要有两种来源，一种是非屏蔽中断(NMI)，另一种是来自各种外部设备的中断。由外部设备的请求引起的中断也称为可屏蔽中断。微型计算机配置的外部设备一般有硬磁盘(DISK)，软磁盘(FLOPPY DISK)，键盘(KEYBOARD)，显示器(CRT)和各种打印机(LINE PRINTER)等。这些外部设备通过 8259A 可编程中断控制器和 CPU 相连，8259A 可编程中断控制器可接收来自外设的中断请求信号，并把中断源的中断类型号送 CPU，如果 CPU 响应该外设的中断请求，就自动转入相应的中断处理程序。但是从外设发出中断请求到 CPU 响应中断，有两个控制条件是起决定性作用的，一是该外设的中断请求是否屏蔽，另一个是 CPU 是否允许响应中断。这两个条件分别由 8259A 的中断屏蔽寄存器(IMR)和标志寄存器(PSW)中的中断允许位 IF 控制。

中断屏蔽寄存器的 I/O 端口地址是 21H，它的 8 位对应控制 8 个外部设备(见图 8.9(a))，通过设置这个寄存器的某位为 0 或为 1 来允许或禁止某外部设备的中断。某位为 0 表示允许某种外设中断请求，某位为 1 表示某种外设的中断请求被屏蔽(禁止)。

例如，只允许键盘中断，可设置如下中断屏蔽字：

```
MOV     AL,11111101B
OUT     21H,AL
```

如果系统中要新增设键盘中断，则可用下列指令实现：

```
IN      AL,21H
AND     AL,11111101B
OUT     21H,AL
```

在编写中断程序时，应在主程序的初始化部分设置好中断屏蔽寄存器，以确定允许用中断方式工作的外部设备。

外部设备向 CPU 发出中断请求，CPU 是否响应还与标志寄存器中的中断标志位 IF 有关。如果 IF＝0，CPU 就禁止响应任何外设的中断，也就是说，CPU 将不会产生中断来处理外设的请求。如果 IF＝1，则允许 CPU 响应外设的中断请求，有两条指令能设置或清除 IF 位。

STI　设置中断允许位(IF=1)

CLI　清除中断允许位(IF=0)

允许 CPU 响应外设的中断请求(IF=1)也叫做开中断,反之叫做关中断(IF=0)。

我们已经知道,当任何类型的中断发生时,当前的 PSW 要保存入栈,然后清除 IF 位进入中断处理程序。如果允许在一个中断处理程序的执行过程中发生外中断,则必须用一条 STI 指令开中断。当执行到中断返回指令 IRET,又取出 PSW 先前的值,其中 IF 为 1,CPU 将允许外中断再次发生。

有一种特殊的外部中断和 IF 标志位无关,这就是非屏蔽中断,非屏蔽中断的类型号为 2,CPU 不能禁止非屏蔽中断,如果系统使用了这种类型的中断,那么 CPU 总会响应的,所以非屏蔽中断主要用于一些紧急的故障处理,如电源掉电等。另外计算机内部的实时钟希望能不停地计时,所以也可以把非屏蔽中断提供给实时钟。

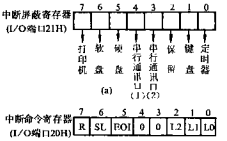


图 8.9　中断屏蔽寄存器和中断命令寄存器

在一次外中断处理结束之前,还应给 8259A 可编程中断控制器的中断命令寄存器发出中断结束命令(EOI)。中断命令寄存器的 I/O 端口地址为 20H(见图 8.9(b)),它的各控制位可动态地控制中断处理过程,其中 L2—L0 三位指定 IR0—IR7 中具有最低优先级的中断请求。6 位(Set Level)和 7 位(Rotate)控制 IR0—IR7 的中断优先级的顺序。5 位(EOI)是中断结束位。当 EOI 位为 1 时,当前正在处理的中断请求就被清除,所以在中断处理完成后,必须把中断结束位置为 1,否则以后将屏蔽掉对同级中断或低级中断的处理。当然在必要的时候,在中断处理程序中也可利用 EOI 命令清除当前的中断请求,使得在中断处理的过程中又能响应同级或低级中断。

结束外中断用下面的指令:

```
MOV     AL,20H
OUT     20H,AL
```

8.3.6　中断优先级和中断嵌套

在 IBM PC 机系统中,有内中断、外中断等多个中断源。当多个中断源同时向 CPU 请求中断时,CPU 应如何处理呢?我们可以给各种中断源事先安排一个中断优先级次序,当多个中断源同时申请中断时,CPU 先比较它们的优先级(Priority),然后从优先级高到优先级低的次序来依次处理各个中断源的中断请求。

IBM PC 规定中断的优先级次序为:

```
优先权高    内中断 (除法错,INTO,INT)
            非屏蔽中断 (NMI)
            可屏蔽中断 (INTR)
低          单步中断
```

可屏蔽中断的优先权又分为八级,在正常的优先级方式下,优先级次序是

IR0,IR1,IR2,IR3,IR4,IR5,IR6,IR7

也就是说，按图 8.5 的连接，定时器的优先权最高，键盘其次，打印机的优先权最低。

在前一节中我们已经提到 8259A 的中断命令寄存器的 6 位和 7 位能控制各中断请求端的优先级次序。在发出一个 EOI 命令时，7 位(R)和 6 位(SL)有四种组合，其含义如下：

R	SL	
0	0	正常优先级方式
0	1	清除由 L2—L0 指定的中断请求
1	0	各中断优先级依次左循环一个位置
1	1	各中断优先级依次循环到由 L2—L0 指定的中断请求到达最低优先级位置上。

外中断的优先级次序一般在正常优先级方式下(R=0,SL=0)，但在必要的情况下，设置中断命令寄存器能改变 IR0—IR7 的优先级次序。例如，IR0—IR7 原为正常的优先级次序，现在要使 IR4 成为最低级的中断请求，则给端口 21H 送命令码：11100100，即 R=1,SL=1，EOI=1,L2L1L0=100，这样，各中断优先级就依次循环到 IR4 为最低优先级的位置上：

IR5,IR6,IR7,IR0,IR1,IR2,IR3,IR4

如果再送一个命令码：10100000，则优先级次序再向左循环一个位置，成为：

IR6,IR7,IR0,IR1,IR2,IR3,IR4,IR5

在下面的章节中，如无特别说明，则中断优先级是指正常方式下的中断优先级。

正在运行的中断处理程序，又被其它中断源中断，这种情况叫做中断嵌套。IBM PC 机没有规定中断嵌套的深度(中断程序又被中断的层次)，但在实际使用时，多重的中断嵌套要受到堆栈容量的限制，所以在编写中断程序时，一定要考虑有足够的堆栈单元来保存多次中断的断点及各寄存器的内容。

一个正在执行的中断处理程序，在开中断(IF=1)的情况下，能被优先级高于它的中断源中断，但如果要被同级或低级的中断源中断，则必须发出 EOI 命令，清除正在执行的中断请求，才能响应同级或低级的中断。

下面举一个例子来说明在正常优先级方式下，优先中断和中断嵌套发生时的处理过程(图 8.10)。

假定在主程序的执行过程中，IR2 和 IR4 的中断请求同时发生，而后 IR1 的中断请求又到达，最后 IR3 的中断请求也到达。

首先，CPU 响应优先级高的 IR2，转去执行 IR2 的中断处理程序。进入 IR2 处理程序后，IF 被置为 1，当 IR1 的中断请求到达后，因 IR1 的优先级高于 IR2，CPU 就立即中断 IR2 的程序，转去执行 IR1 的处理程序。在 IR1 处理程序中，由指令发出了 EOI 命令，结束了 IR1 的中断请求。返回 IR2 处理程序后，同样由于发出 EOI 命令清除了 IR2 的中断请求，所以在较低级的中断请求 IR4 到达后，即转向处理 IR4 的中断请求。在 IR4 处理程序的执行过程中，IR3 的中断请求到达，当判断 IF 已被置为 1，则又中断了 IR4 的程序，转去执行 IR3 的程序。在 IR3 程序中，也发出了开中断指令(STI)和中断结束命令(EOI)，最后 IRET 指令使其返回到 IR4 程序，IR4 在返回 IR2 之前也发出了 EOI 命令，结束了 IR4 的中断请求。IR2 中断请求在前面已被清除，所以 IR4 执行完后，IR2 继续执行直到返回主程序。

在多级的 8259A 中断系统中，从属的 8259A 连接到主 8259A 的哪一端上，它就具有哪一端的中断优先级别，例如在图 8.11 的连接方式下，各中断请求端的优先级排列顺序如下：

最高优先级

Master：IR0

Slave1：IR0,IR1,IR2,IR3,IR4,IR5,IR6,IR7

Slave2：IR0,IR1,IR2,IR3,IR4,IR5,IR6,IR7

Master：IR3,IR4,IR5,IR6,IR7

最低优先级

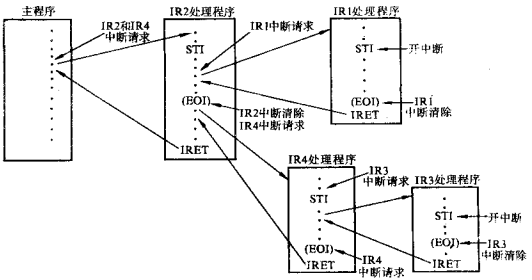


图 8.10　正常优先级方式下的典型中断序列

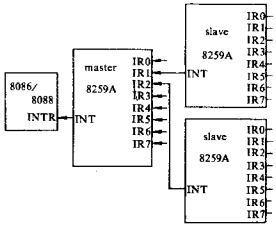


图 8.11　多级 8259A 中断系统

对于主 8259A 和从属 8259A 的中断屏蔽寄存器，禁止相应端中断请求的原理和单级 8259A 是一样的。

8.3.7 中断处理程序

通过前面几节对中断的介绍，我们对如何编写中断程序已经有一些了解了，现在我们把它们归纳在一起就更清楚了。

下面是主程序为中断所做的准备工作和硬件(包括 CPU 和外设接口)自动完成的动作：

主程序：
(1) 设置中断向量
(2) 设置设备的中断屏蔽位
(3) 设置 CPU 的中断允许位 IF(开中断)

硬件：
(4) 外设接口送中断请求给 CPU
(5) 当前指令执行完后，CPU 送响应信号给外设接口
(6) CPU 接收中断类型号
(7) 当前的 PSW，CS 和 IP 保存入栈
(8) 清除 IF、TF
(9) 中断向量送 IP 和 CS

至此，CS 和 IP 寄存器取得了中断处理程序的段地址和偏移地址，CPU 就把控制转给中断处理程序。这里要注意的是设备发到 CPU 的中断请求信号在时间上是随机的，只要未被屏蔽的设备本身的状态是准备好或空闲的，它就会向 CPU 请求中断，如果此时 CPU 正在执行一条指令，那么就要等这条指令执行完后，才响应中断。对加重复前缀的串指令(如 REP MOVSB)，要作为一条指令处理，但不是把串操作全部重复执行完，而是执行一次重复和串指令即可响应中断。对 MOV 指令和 POP 指令，如果处理对象为段寄存器时，那么本条指令执行完后，接着再执行一条指令才响应中断。对开中断指令 STI 和中断返回指令 IRET，也是要在 STI 或 IRET 指令执行完后，再执行一条指令才响应中断，以上是几种特殊情况，对一般指令，只要一条指令的执行周期结束即可响应中断。

中断处理程序的编写方法和标准子程序很类似，下面是编写中断处理子程序的步骤，请注意与子程序编写的一些不同之处。

(1) 保存寄存器内容
(2) 如允许中断嵌套，则开中断(STI)
(3) 处理中断
(4) 关中断
(5) 送中断结束命令(EOI)给中断命令寄存器
(6) 恢复寄存器内容
(7) 返回被中断的程序(IRET)

进入中断处理程序时，IF 和 TF 已被清除，这样在执行中断处理程序的过程中，将不再响应其它外设的中断请求，如果这个中断处理程序允许其它设备中断，则需用 STI 指令把 IF 位置 1。中断结束命令(EOI)在程序的什么地方发出，这要看程序员是否要求在其处理过程中允许同级或低级中断。一般设备希望一次中断的处理过程最好是完整的，所以只在中断处理结束之前发出 EOI 命令。

处理中断部分是中断处理程序的主体部分，它要完成的任务是各式各样的，这与实际的应用有关。如果它的任务是处理某种错误的，一般要显示输出一系列出错信息。如果它是对一个 I/O 设备进行服务的，就按其端口地址接收或发送一个单位(字节或字)的数据。要注意的是

CPU 产生一次中断，I/O 设备只完成一个字节(或字)的输入/输出，所以中断处理程序所用的指针变量或数据变量一般应设置存储器单元来保存。

8.3.8 中断程序举例

例 8.4 在系统定时器(中断类型为 8)的中断处理程序中，有一条中断指令 INT 1CH，时钟中断每发生一次(约每秒中断 18.2 次)，都要调用一次中断类型 1CH 的处理程序。在 ROM BIOS 中，1CH 的处理程序只有一条 IRET 指令，实际上它并没有做任何工作，只是为用户提供了一个中断类型号。如果用户有某种定时周期性的工作需要完成，就可利用系统定时器的中断间隔，用自己设计的处理程序来代替原有的 1CH 程序。

下面我们编写一个中断处理程序，要求在主程序运行的过程中，每隔 10 秒钟响铃一次，同时在屏幕上显示出信息"The bell is ring!"

1CH 为用户的中断类型，可能已被其它功能的程序所引用，所以在编写新的中断程序时，应做下述工作：

1. 在主程序的初始化部分，先保存当前中断向量表的内容，再置新的中断向量。
2. 在主程序的结束部分恢复保存的 1CH 向量。

```
TITLE   TIMER_INT -- EXAMPLE 8-1
;Sounds and display a message per 10 second
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
dseg    segment
count   dw   1
mess    db   'The bell is ring!',0dh, 0ah,'$'
dseg    ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
cseg    segment                    ;defint code segment
;-------------------------------------------------------------------
main    proc        far            ;main part of program
        assume cs : cseg, ds : dseg, es : dseg
start :
        push        ds             ;initialize
        sub         ax,ax
        push        ax
        mov         ax,dseg        ;datarea segment addr
        mov         ds,ax          ;  into DS register
        mov         al,1ch         ;get interrupt vector
        mov         ah,35h
        int         21h
        push        es             ;save interrupt
        push        bx             ;  vector in stack
        push        ds
        mov         dx,offset ring ;offset of ring
        mov         ax,seg ring    ;base of ring
        mov         ds,ax
```

```
        mov     al,1ch          ;set interrupt
        mov     ah,25h          ;  vector
        int     21h
        pop     ds

        in      al,21h          ;set interrupt
        and     al,11111110b    ;  mask bit
        out     21h,al
        sti

        mov     di,2000         ;main processor
delay : mov     si,3000
delay1 : dec    si
        jnz     delay1
        dec     di
        jnz     delay

        pop     dx              ;restore old
        pop     ds              ;  intrrupt vector
        mov     al,1ch
        mov     ah,25h
        int     21h
        ret
main    endp
;-------------------------------------------------
ring    proc    near
        push    ds              ;save the working regs.
        push    ax
        push    cx
        push    dx
        mov     ax,dseg
        mov     ds,ax
        sti
        dec     count           ;count is value of 10 sec.
        jnz     exit

        mov     dx,offset mess
        mov     ah,09h          ;print the message
        int     21h

        mov     dx,100          ;turn on/off 100 times
        in      al,61h          ;get port 61h
        and     al,0fch         ;AND off bits 0,1
sound : xor     al,02           ;toggle bit 1
        out     61h,al          ;output to port 61h
        mov     cx,140h         ;value of wait
wait1 : loop    wait1
        dec     dx              ;total 100 times
```

```
        jne     sound
        mov     count,182       ;value of 10 second
exit:   cli
        pop     dx              ;restore the regs
        pop     cx
        pop     ax
        pop     ds
        iret                    ;interrupt return
ring    endp
;-------------------------------------------
cseg    ends                    ;end of code segment
;*************************************
        end     start           ;end assembly
```

图 8.12 TIMER_INT 程序

例 8.5 在配置了键盘输入(中断类型 09)和打印机输出(中断类型 0FH)两种中断设备的 PC 机系统中,要求从键盘上接收字符,同时对 32 字节的输入缓冲区进行测试,如果缓冲区已满,则将键盘挂起(禁止键盘中断输入),由打印输出一个提示信息。键盘和打印机分别由中断屏蔽寄存器(21H)的 1 位和 7 位控制。键盘的输入寄存器端口地址为 60H,控制寄存器的端口地址为 61H。打印机输出寄存器的端口地址为 378H,打印机控制寄存器的端口地址为 37AH。

在这种特定情况下,只要求打印机在键盘输入缓冲区满了后,打印出提示信息,因此它可以在屏蔽键盘中断的同时,设置打印机的中断屏蔽位。另外在中断处理程序中用到的一些指针及计数值要保存在指定的存储单元中,每次进入中断,取出指针及计数值,退出中断时,再把修改后的指针及计数值保存起来。

这个程序包括以下几部分:

MAIN 初始化部分,保存 09 和 0AH 的原中断向量,设置新的中断向量。主程序用有限循环来模拟。主程序结束时,恢复原中断向量。

KBINT 键盘中断处理程序。接收按键的扫描码并保存在缓冲区中,如果输入字符大于 32,则屏蔽键盘中断,允许打印机中断,并调用初始化打印机子程序。

INTIP 初始化打印机,启动打印机适配器,发出选通信号。

PRINT 打印机中断处理程序。按照指针取出打印字符送到输出寄存器,发出选通信号。

DISP 用十六进制显示 AL 中的代码。

```
TITLE   KBPRT_INT -- EXAMPLE 8 - 2
;Keyboard input and print a message on the printer
;*************************************
dseg    segment                         ;defint data segment
addr_point  dw  ?
count       dw  ?
buffer      db  20h dup(' ')
prompt      db  'please enter the characters :'
            db  0dh,0ah,'$'
message     db  'buffer overflow',0dh,0ah
```

```
save_ip9    dw  ?
save_cs9    dw  ?
save_ipf    dw  ?
save_csf    dw  ?
dseg        ends
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
cseg        segment                 ;define code segment
;-----------------------------------------------------------
main        proc    far             ;main part of program
            assume cs : cseg, ds : dseg
start :
            push    ds              ;set up stack for return
            sub     ax,ax
            push    ax
            mov     ax,dseg         ;set DS to dseg segment
            mov     ds,ax
            mov     ax,offset buffer ;initialize
            mov     addr_point,ax
            mov     count,0
            mov     al,09h          ;save interrupt
            mov     ah,35h          ;  vector of
            int     21h             ;  type 09h
            mov     save_ip9,bx
            mov     save_cs9,es
            mov     dx,offset kbint ;set interrupt
            push    ds              ; vector of 09
            mov     ax,seg kbint
            mov     ds,ax
            mov     al,09h
            mov     ah,25h
            int     21h
            pop     ds
            mov     al,0fh          ;save interrupt
            mov     ah,35h          ;  vector of
            int     21h             ;  type 0Fh
            mov     save_ipf,bx
            mov     save_csf,es
            cli
            mov     dx,offset prtint ;set interrupt
            push    ds              ; vector of 0Fh
            mov     ax,seg prtint
            mov     ds,ax
```

```
        mov     al,0fh
        mov     ah,25h
        int     21h
        pop     ds

        in      al,21h          ;set KB interrupt
        and     al,0fdh         ;  mask bit
        out     21h,al

        mov     ah,09h          ;print prompt mess.
        lea     dx,prompt
        int     21h

        sti
        mov     di,2000
mainp:  mov     si,3000
mainp1: dec     si
        jnz     mainp1
        dec     di
        jnz     mainp

        mov     ah,2            ;'$' means end
        mov     dl,'$'          ; of main process
        int     21h

retn:   cli
        push    ds
        mov     dx,save_ip9     ;restore interrupt
        mov     ax,save_cs9     ; vector of 09h
        mov     ds,ax
        mov     al,09h
        mov     ah,25h
        int     21h
        pop     ds

        push    ds
        mov     dx,save_ipf     ;restore interrupt
        mov     ax,save_csf     ; vector of 0Fh
        mov     ds,ax
        mov     al,0fh
        mov     ah,25h
        int     21h
        pop     ds

        in      al,21h          ;enable KB interrupt
        and     al,0fdh
        out     21h,al
        sti
        ret                     ;return DOS
```

```
main    endp
;-------------------------------------------------
kbint   proc    near
        push    ax              ;save the working
        push    bx              ;  register
        cld                     ;forward direction

        in      al,60h          ;read in the char.
        push    ax              ;save it
        in      al,61h          ;get the control port
        mov     ah,al           ;save value
        or      al,80h          ;reset bit for kbd.
        out     61h,al          ;
        xchg    ah,al           ;get back org control
        out     61h,al          ;KB has been reset
        pop     ax              ;recover scan code

        test    al,80h          ;is press or release code?
        jnz     cont            ;don't process release code

        mov     bx,addr_point   ;buffer point
        mov     [bx],al         ;store in buffer
        call    disp            ;display scan code
        inc     bx              ;inc address
        inc     count           ;count characters
        mov     addr_point,bx   ;save the point
check:  cmp     count,32        ;is buffer overflow?
        jb      cont            ;no,return
        in      al,21h          ;yes, then
        or      al,02           ;  mask KB bit
        and     al,7fh          ;  enable PRT bit
        out     21h,al
        call    intip           ;initialize printer

cont:   cli
        mov     al,20h          ;end of interrupt
        out     20h,al
        pop     bx              ;restore register
        pop     ax
        iret
kbint   endp
;-------------------------------------------------
intip   proc    near
        push    ax              ;save working reg.
        push    bx
        push    dx
        cli
```

```
        mov     bx,offset message
        mov     addr_point,bx       ;save addr of mess

        mov     dx,378h             ;start printer adapter
        mov     al,0dh
        out     dx,al

        mov     dx,37ah             ;send a strobe to
        mov     al,1dh              ;   control byte
        out     dx,al
        jmp     $+2
        mov     al,1ch
        out     dx,al

        pop     dx                  ;restore registers
        pop     bx
        pop     ax
        ret
intip   endp
;-------------------------------------------------
prtint  proc    near
        push    ax                  ;save registers
        push    bx
        push    dx

        mov     bx,addr_point       ;message addr. in BX
        mov     al,[bx]

        mov     dx,378h             ;printer data port
        out     dx,al               ;output a charactet

        push    ax
        mov     dx,37ah             ;PRT control port
        mov     al,1dh              ;send control code
        out     dx,al               ;   and strobe
        jmp     $+2
        mov     al,1ch
        out     dx,al
        pop     ax

        inc     bx                  ;next character addr.
        mov     addr_point,bx       ;save it
        cmp     al,0ah              ;end of message ?
        jnz     return              ;no,return

        in      al,21h              ;disnable printer
        or      al,80h              ;   interrupt
        out     21h,al

return: mov     al,20h              ;end of interrupt
```

```
        out     20h,al
        pop     dx
        pop     bx
        pop     ax
        iret                    ;interrupt return
prtint  endp
;-------------------------------------------------
disp    proc    near            ;display with hex
        push    ax
        push    cx
        push    dx
        mov     ch,2            ;number of digits
        mov     cl,4
nextb:  rol     al,cl           ;highest 4 bits to lowest
        push    ax
        mov     dl,al
        and     dl,0fh          ;mask off left digit
        or      dl,30h          ;convert to ASCII
        cmp     dl,3ah
        jl      dispit
        add     dl,7h           ;digit is A to F
dispit: mov     ah,2            ;display character
        int     21h
        pop     ax
        dec     ch              ;done 2 digit?
        jnz     nextb           ;not yet
        mov     ah,2
        mov     dl,','          ;display ','
        int     21h
        pop     dx
        pop     cx
        pop     ax
        ret                     ;return form disp
disp    endp
;-------------------------------------------------
cseg    ends
        end     start           ;end assembly
```

图 8.13 KBPRT_INT 程序

例 8.6 除数为 0 时的内中断(类型 0)处理程序

此程序分成两个主要部分:初始化部分和中断处理部分。

初始化部分(Init)设置新的零型中断向量,显示一条信息,然后完成终止和驻留后退出 DOS。这种特殊的退出使用 INT 21H 的功能 31H,它将保留程序所占的内存,从而使这些内存单元不被以后的应用程序破坏。

中断处理程序(Zdiv)在发生一个被零除中断时接收控制。中断处理程序先保存有关寄存器,然后打印出信息询问用户是退出程序(Quit)还是继续(Continue)。在这个中断处理程序中,我们可使用 DOS 显示器和键盘 I/O 功能,这是因为在进入 Zdiv 中断处理时,用 STI 指令打开中断了。若键入"C"要求继续执行程序,则处理程序恢复所有寄存器并执行 IRET 返回主程序(显示一个标记符#),当然此时除法的操作结果应是无效的。若键入"Q"要求退出,则从处理程序直接退回 DOS(无标记符显示),这里退回 DOS,使用 INT 21H 的功能 4CH,该功能允许传送返回代码,同时它也是唯一不依赖于任何段寄存器内容的终止功能。图 8.14 是除数为 0 的处理程序清单。

```
TITLE ZERODIV.ASM—Divide by Zero Handler
;To assemble,link and conver into a com file :
;C>MASM ZERODIV
;C>LINK ZERODIV
;C>EXE2BIN ZERODIV.EXE ZERODIV.COM
;C>DEL ZERODIV.EXE
cseg   segment para public 'code'
       org   100h
       assume cs : cseg,es : cseg,ss : cseg
init   proc  near
       mov   dx,offset zdiv          ;reset interrupt
       mov   ax,seg zdiv             ;0 vector
       mov   ds,ax
       mov   al,0
       mov   ah,25h
       int   21h
       mov   dx,offset signon        ;print message
       mov   ah,9
       int   21h
       mov   ax,20                   ;assume the case
       mov   dl,0                    ; of divide by zero
       div   dl
       mov   ah,02                   ;if entered 'c',
       mov   dl,'#'                  ; return the main
       int   21h                     ;   program
       mov   ah,31h                  ;exit & stay residente
       mov   al,0                    ; return code = 0
       mov   dx,((pgm_len+15)/16)+10h
       int   21h                     ;DX=paragraphs of
                                     ; memory to reserve
init   endp
;-------------------------------------------------------
zdiv   proc  far                     ;this is the interrupt handler
```

```
                push    ax                      ;save general registers
                push    bx
                push    cx
                push    dx
                push    si
                push    di
                push    bp
                push    ds
                push    cs
                sti
zdiv0 :         mov     ax,cs                   ;print warning
                mov     ds,ax                   ;
                mov     dx,offset warn
                mov     ah,9
                int     21h
zdiv1 :         mov     ah,1                    ;read keyboard
                int     21h
                cmp     al,'c'                  ;is it c or q ?
                je      zdiv3                   ;jump,it's a c
                cmp     al,'q'
                je      zdiv2                   ;jump, it's a q
                mov     dx,offset bad           ;illegal entry
                mov     ah,09                   ;send a beep
                int     21h
                jmp     zdiv0                   ;try again
zdiv2 :         mov     ax,4cffh                ;exit
                int     21h
zdiv3 :         mov     dx,offset crlf          ;send CR and LF
                mov     ah,09
                int     21h
                cli
                pop     es                      ;restore registers
                pop     ds
                pop     bp
                pop     di
                pop     si
                pop     dx
                pop     cx
                pop     bx
                pop     ax
                iret                            ;interrupt return
zdiv            endp
;-------------------------------------------------------------------
```

```
signon    db   0dh,0ah,' * Divide by zero Interrupt '
          db   'Handler installed. *'
          db   0dh,0ah,'$'
warn      db   0dh,0ah,'Divide by zero detected : '
          db   0dh,0ah,'Continue or Quit (c/q)? $'
bad       db   07h,'$'
crlf      db   0dh,0ah,'$'
pgm_len   equ  $-init
;-------------------------------------------------------------
cseg      ends
          end       init
```

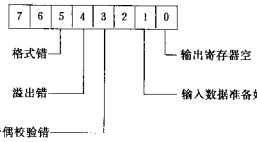
图 8.14 ZERODIV 程序

习 题

8.1 写出指令将一个字节数据输出到端口 25H。

8.2 写出指令将一个字数据从端口 1000H 输入。

8.3 假定串行通讯口的输入数据寄存器的端口地址为 50H,状态寄存器的端口地址为 51H,状态寄存器各位为 1 时含义如下:



请编写一程序:输入一串字符并存入缓冲区 BUFF,同时检验输入的正确性,如有错则转出错处理程序 ERROUT。

8.4 试编写程序,它轮流测试两个设备的状态寄存器,只要一个状态寄存器的第 0 位为 1,则与其相应的设备就输入一个字符;如果其中任一状态寄存器的第 3 位为 1,则整个输入过程结束。两个状态寄存器的端口地址分别是 0024 和 0036,与其相应的数据输入寄存器的端口则为 0026 和 0038,输入字符分别存入首地址为 BUFF1 和 BUFF2 的存储区中。

8.5 假设外部设备有一台硬币兑换器,其状态寄存器的端口地址为 0006,数据输入寄存器的端口地址为 0005,数据输出寄存器的端口地址为 0007。试用查询方式编制一程序,该程序作空闲循环等待纸币输入,当状态寄存器的第 2 位为 1 时,表示有纸币输入,此时可从数据输入寄存器输入的代码中测出纸币的品种,一角纸币的代码为 01,二角纸币为 02,五角纸币则为 03。然后程序在等待状态寄存器的第 3 位变为 1 后,把应兑换的五分硬币数(用 16 进制表示)从数据输出寄存器输出。

8.6 给定(SP)=0100,(SS)=0300,(PSW)=0240,以下存储单元的内容为(00020)=0040,(00022)=0100,在段地址为 0900 及偏移地址为 00A0 的单元中有一条中断指令 INT 8,试问执行 INT 8 指令后,SP,SS,IP,PSW 的内容是什么?栈顶的三个字是什么?

8.7 类型 14 的中断向量在存储器的哪些单元里?

8.8 假设中断类型 9 的中断处理程序的首地址为 INT_ROUT,试写出主程序中为建立这一中断向量而编制的程序段。

8.9　编写指令序列，使类型 1CH 的中断向量指向中断处理程序 SHOW_CLOCK。

8.10　如设备 D1，D2，D3，D4，D5 是按优先级次序排列的，设备 D1 的优先级最高。而中断请求的次序如下所示，试给出各设备的中断处理程序的运行次序。假设所有的中断处理程序开始后就有 STI 指令。

1）设备 3 和 4 同时发出中断请求

2）在设备 3 的中断处理程序完成之前，设备 2 发出中断请求。

3）在设备 4 的中断处理程序未发出中断结束命令(EOI)之前，设备 5 发出中断请求。

4）以上所有中断处理程序完成并返回主程序，设备 1，3，5 同时发出中断请求。

8.11　在 8.10 中假设所有的中断处理程序中都没有 STI 指令，而它们的 IRET 指令都可以由于 PSW 出栈而使 IF 置 1，则各设备的中断处理程序的运行次序应是怎样的？

8.12　试编制一程序，要求测出任意程序的运行时间，并把结果打印出来。

第九章　BIOS 和 DOS 中断

在存储器系统中，从地址 0FE000H 开始的 8K ROM(只读存储器)中装有 BIOS(Basic Input/Output System)例行程序。驻留在 ROM 中的 BIOS 提供了系统加电自检，引导装入，主要 I/O 设备的处理程序以及接口控制等功能模块来处理所有的系统中断。使用 BIOS 功能调用，给程序员编程带来很大方便，程序员不必了解硬件 I/O 接口的特性，可直接用指令设置参数，然后中断调用 BIOS 中的程序，所以利用 BIOS 功能编写的程序简洁，可读性好，而且易于移植。

DOS (Disk Operating System)是 IBM PC 机的磁盘操作系统，它是由软盘或硬盘提供的。它的两个 DOS 模块 IBMBIO. COM 和 IBMDOS. COM 使 BIOS 用起来更方便，因为 DOS 模块提供了更多更必要的测试，使 DOS 操作比使用相应功能的 BIOS 操作更简易，而且 DOS 对硬件的依赖性更少些。

IBMBIO. COM 是一个输入/输出设备处理程序，它提供了 DOS 到 ROM BIOS 的低级接口，它完成将数据从外设读入内存，或把数据从内存写到外设去的工作。

IBMDOS. COM 包括一个文件管理程序和一些处理程序，在 DOS 下运行的程序可以调用这些处理程序。为了完成 DOS 功能调用，IBMDOS. COM 把信息传送给 IBMBIO. COM，形成一个或多个 BIOS 调用。它们之间的关系如图 9. 1 所示。

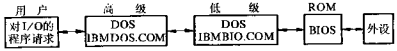


图 9. 1　DOS 模块和 ROM BIOS 的关系

在一些情况下，既能选择 DOS 中断也能选择 BIOS 中断来执行同样的功能。例如，打印机输出一个字符的功能，可用 DOS 中断 21H 的功能 5，也可用 BIOS 中断 17H 的功能 0。因为 BIOS 比 DOS 更靠近硬件，因此建议尽可能地使用 DOS 功能，但在少数情况下必须使用 BIOS 功能，例如，BIOS 中断 17H 的功能 2 为读打印机状态，它就没有等效的 DOS 功能。

DOS 中断能处理大多数的 I/O，但有一些功能还没有提供，如声音控制等，这就要考虑用 I/O 指令在端口级上编程，或使用高级语言编程。

表 9. 1 和表 9. 2 列出了 IBM PC 系统主要的 BIOS 中断类型和 DOS 中断类型。

表 9. 1　BIOS 中断类型

CPU 中断类型			
0	除法错	4	溢出
1	单步	5	打印屏幕
2	非屏蔽中断	6	保留
3	断点	7	保留

续表

8259 中断类型			
8	8254 系统定时器	C	保留(通讯)
9	键盘	D	保留(Alt 打印机)
A	保留	E	软盘
B	保留(通讯)	F	打印机
BIOS 中断类型			
10	显示器	16	键盘
11	设备检验	17	打印机
12	内存大小	18	驻留 BASIC
13	磁盘	19	引导
14	通讯	1A	时钟
15	I/O 系统扩充	40	软盘
用户应用程序			
1B	键盘 Break	1C	定时器
4A	报警		
数据表指针			
1D	显示器参量	41	1# 硬盘参量
1E	软盘参量	46	2# 硬盘参量
1F	图形字符扩充		

表 9.2 DOS 中断类型

20	程序结束	26	绝对盘写入
21	功能调用	27	结束并留在内存
22	结束地址	28 2E	保留给 DOS
23	Ctrl_Break 出口地址	2F	打印机
24	严重错处理	30-3F	保留给 DOS
25	绝对盘读取		

9.1 键盘 I/O

键盘提供了三种基本类型的键：

1. 字符键，如字母 A 到 Z，数字 0 到 9，%，$，# 等

2. 扩展功能键，如 Home，End，Backspace，Arrows，Return，Del，Ins，PgUp，PgDn 以及程序功能键等。

…其它键组合使用的控制键，如 Alt、Ctrl 和 Shift。

字符键给计算机传送一个 ASCII 码字符，而扩展功能键产生一个动作，如按下 Home 键能把光标移到屏幕的左上角，End 键使光标移到屏幕上文本的末尾。使用控制键能改变其它键所产生的字符码。

键盘是计算机最基本的一种输入设备，用以输入信息。本节将主要介绍 BIOS 和 DOS 的键盘操作。

9.1.1 字符码与扫描码

当我们在键盘上"按下"或"放开"一个键时，如果键盘中断是允许的(21H 端口第 1 位＝0)，就会产生一个类型 9 的中断，并转入到 BIOS 的键盘中断处理程序。该处理程序从 8255 可编程序外围接口芯片的输入端口 60H 读取一个字节，这个字节的低 7 位是键的扫描码，最高位为 0 或为 1，分别表示键是"按下"状态还是"放开"状态。按下时，取得的字节称为通码，放开时，取得的字节称为断码。如按下 Esc 键时产生一个通码为 01H，放开 Esc 键时产生一个断码为 81H。键盘上的每个键都对应一个扫描码，从 01(Esc)到 83(Del)，或从 01H 到 53H，所以根据扫描码就能唯一地确定哪一个键改变了状态。表 9.3 是键盘上每个键对应的扫描码(十六进制)。

表 9.3 IBM 键盘的扫描码表

键	扫描码	键	扫描码	键	扫描码	键	扫描码
Esc	01	O	18	X	2D	F8	42
1	02	P	19	C	2E	F9	43
2	03	[	1A	V	2F	F10	44
3	04	]	1B	B	30	NumLock	45
5	06	Enter	1C	N	31	ScrollLock	46
6	07	Ctrl	1D	M	32	Home	47
8	09	A	1E	,	33	↑	48
9	0A	S	1F	.	34	PgUp	49
0	0B	D	20	/	35	-	4A
—	0C	F	21	Shift(右)	36	←	4B
=	0D	G	22	Prtsc	37	5	4C
Backspace	0E	H	23	Alt	38	→	4D
Tab	0F	J	24	Space	39	+	4E
Q	10	K	25	CapsLock	3A	End	4F
W	11	L	26	F1	3B	↓	50
E	12	;	27	F2	3C	PgDn	51
R	13	'	28	F3	3D	Ins	52
T	14	`	29	F4	3E	Del	53
Y	15	Shift(左)	2A	F5	3F		
U	16	\	2B	F6	40		
I	17	Z	2C	F7	41		

BIOS 键盘处理程序将取得的扫描码转换成相应的字符码，大部分键的字符码是一个标准的 ASCII 码，没有相应 ASCII 码的键，如 Alt 和功能键(F1—F10)，字符码为 0，还有一些非

ASCII 码键产生一个指定的操作，如打印屏幕内容等。转换成的字符码以及扫描码存入 ROM BIOS 数据区的键盘缓冲区 KB_BUFFER 中。

```
0040：001A   BUFF_HEAD   DW ?              ;键盘缓冲区的首地址
0040：001C   BUFF_TAIL   DW ?              ;键盘缓冲区的末地址
0040：001E   KB_BUFFER   DW 16 DUP (?)  ;16 个输入量的空间
0040：003E   KB_BUFFER_END LABEL WORD
```

这个缓冲区是一个先进先出的循环队列，BUFF_HEAD 和 BUFF_TAIL 是缓冲区的两个地址指针。当 HEAD 指针和 TAIL 指针相等时，说明缓冲区空。当 CPU 想要得到键盘输入时，就调用 BIOS 键盘例行程序，它按其接收时的次序从缓冲区取出字符和扫描码，回送给 CPU。缓冲区的大小可适应最快的打字员，但如果缓冲区已满又按下了一个键，BIOS 不处理这个键，只发出"嘀"的响声。

我们可以用 BIOS 中断，也可以用 DOS 中断和键盘通讯，下面我们分别讨论这两种键盘中断。

9.1.2 BIOS 键盘中断

类型 16（键盘）中断提供了基本的键盘操作，类型 16 的中断处理程序包括 3 个不同的功能，分别根据 AH 寄存器的内容来选择（见表 9.4）

表 9.4 BIOS 键盘中断(INT 16H)

AH	功 能	返回参数
0	从键盘读一字符	AL＝字符码
		AH＝扫描码
1	读键盘缓冲区的字符	如 ZF＝0
		AL＝字符码
		AH＝扫描码
		如 ZF＝1，缓冲区空
2	取键盘状态字节	AL＝键盘状态字节

利用 INT 16H 调用键盘 I/O ROM 例行程序时，先在 AH 中放一个功能编号 0，1 或 2，例如我们要查看按键的扫描码和 ASCII 码，可以调用中断类型 16H 的 0 功能，该功能把扫描码回送到 AH 中，把 ASCII 码回送到 AL 中，然后调用二进制转换十六进制的子程序 BINIHEX，把 AH 和 AL 中的内容打印出来。其指令序列为：

```
MOV     AH,0          ;read character function
INT     16H           ;Keyboard I/O ROM call
MOV     BX,AX         ;move AX to BX
CALL    BINIHEX       ;Print scancode & char
```

前面我们已经提到 Shift、Ctrl、Alt、Num Lock、Scroll、Ins 和 Caps Lock 这些键不具有 ASCII 码，但按动了它们能改变其它键所产生的代码，那么如何能判断这些键按动与否呢？INT 16H 的 AH＝2 的功能可以把表示这些键状态的字节——键盘状态字节(KB_FLAG)回送到 AL 寄

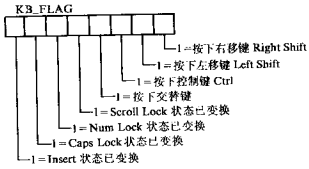


图 9.2　键盘状态字节

存器。图 9.2 标出了 KB_FLAG 各位表示的状态信息，其中高 4 位指出各种键盘方式(Ins、Caps Lock、Num Lock、Scroll)是 ON(1)还是 OFF(0)；低 4 位表示 Alt，Ctrl，Shift 键是否按动。这 8 个键有时又被称为变换键。

图 9.3 的程序可以读取 KB—FLAG 的内容，如果要显示出各位的状态，可调用 BINIHEX 子程序来显示 KB_FLAG 的十六进制内容。

```
AGAIN:
        MOV     AH,02H          ;shift status function
        INT     16H             ;call kbd ROM routine
        MOV     BX,AX           ;put result in BX
        CALL    BINIHEX         ;print out result
        MOV     DL,0DH          ;print CR
        MOV     AH,02H
        INT     21H
        JMP     AGAIN           ;repeat
;-----------------------------------------
```

图 9.3　读取变换键状态的指令序列

9.1.3　DOS 键盘功能调用

上节介绍了 BIOS 键盘中断(16H)，它能同时回送字符码和扫描码，这在使用功能键和变换键的程序中是很重要的。但对一般的键盘操作，我们最好使用适应能力更强的由 DOS 类型 21H 中断提供的键盘功能调用。

表 9.5 列出了与键盘输入有关的 DOS 21H 功能调用，它包括把单字符读入 AL 和把一个字符串读入存储器等功能。在编写程序时，你会感到使用 DOS 21H 键盘功能调用非常方便。

表 9.5 DOS 键盘操作(INT 21H)

AH	功能	调用参数	返回参数
1	从键盘输入一个字符并回显在屏幕上.		AL=字符
6	读键盘字符	DL=0FFH	AL=字符 (如果准备好) AL=0 (未准备好)
7	从键盘输入一个字符,不回显		AL=字符
8	从键盘输入一个字符,不回显,检测 Ctrl_Break		AL=字符
A	输入字符到缓冲区	DS:DX=缓冲区首址	
B	读键盘状态		AL=0FFH 有键入 AL=00 无键入
C	清除键盘缓冲区,并调用一种键盘功能	AL=键盘功能号(1,6,7,8 或 A)	

9.1.3.1 单字符输入

DOS 21H 中断的功能 1,7 和 8 都能从键盘读一字符送入 AL 寄存器。功能 1 能把字符显示出来并检验是否按下了 Ctrl_Break 键,如果按下了 Ctrl_Break 键,就自动调用中断 23H 并结束程序。21H 的功能 7 不能回打字符或检验 Ctrl_Break。21H 的功能 8 检验 Ctrl_Break,但是不回显。

在交互程序中常常需要用户对一个提示做出应答,或通过输入一个字母或数字对菜单的各项进行选择,这时就要用到 21H 的单字符输入功能。例如程序显示出一串信息,要求你回答 Y 或 N,回答 Y,程序将转入标号为 YES 的程序段,而 N 使程序转入标号为 NO 的程序段,按下其它键程序就等待。这样的工作由下面的程序段来完成:

```
GET_KEY:  MOV AH,1                ;Read a key with echo
          INT 21H
          CMP AL,'Y'              ;Is it Y?
          JE YES                  ;If so,jump to YES
          CMP AL,'N'              ;Is it N?
          JE NO                   ;If so ,jump to NO
          JNE GET_KEY             ;otherwise,wait for Y or N
```

测试 Y,N 或其它字母,数字和符号可直接把它们写在 CMP 指令中,用引号括起来。但是如果想检测 Enter(Return)键,就要在指令中写出它的 ASCII 码 0D(16 进制)或 13(十进制)。例如,要求程序在按下 Return 键后才继续运行,用下面的指令:

```
WAIT_HERE:  MOV AH,7              ;Wait for enter
            INT 21H
            CMP AL,0DH
            JNE WAIT_HERE
```

这里用 AH=7 代替 AH=1,差别只是不把按下的键显示出来,或不执行键的特定功能。

如果要求程序能接收功能键或数字组合键必须进行两次 DOS 调用,第一次回送 00,第二次回送扫描码。例如,程序显示出一个菜单,要求用户通过键入 F1,F2 或 F3 来选择 1,2 或 3

项，按其它键则产生错误信息。程序的应答检测部分如下：

```
        MOV     AH,7        ;Wait for key
        INT     21H
        CMP     AL,0        ;Is it a function key?
        JE      GET_EC      ;yes, read the scan code
        JMP     ERROR       ;No,display error message
GET_EC: MOV     AH,7
        INT     21H
        CMP     AL,3BH      ;F1 ?
        JE      OPTION1
        CMP     AL,3CH      ;F2?
        JE      OPTION2
        CMP     AL,3DH      ;F3?
        JE      OPTION3
        JMP     ERROR       ;Invalid key ,display
                            ;error message
```

9.1.3.2 输入字符串

在许多应用程序中，要求用户输入姓名、地址或其它字符串，21H 中断的功能 A 能从键盘读入一串字符并把它存入用户定义的缓冲区中。缓冲区的第一个字节保存最大字符数，这个最大字符数由用户程序给出。如果键入的字符数比此数大，那就会发出“嘟嘟”声，而且光标不再向右移动。

第二个字节是实际输入字符的个数，这个数据是由功能 A 填入的，而不是由用户填入。在这两个字节之后，字符串就按字节存入缓冲区，最后结束字符串的回车符 0DH 还要占用一个字节，因此整个缓冲区的字节空间应为最大字符数(包括 Return 在内)加 2。

例如，在数据区定义的字符缓冲区如下：

```
MAXLEN  DB  32
ACTLEN  DB  ?
STRING  DB  32 DUP (?)
```

输入字符串的指令如下：

```
LEA     DX, MAXLEN      ;Make DX point to buffer
MOV     AH,0AH          ;Input the string
INT     21H
```

如果我们键入字符串：

By brooks too broad for leaping↙

此时缓冲区 MAXLEN 的各存储单元图示如图 9.4。

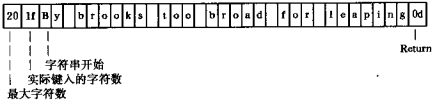


图 9.4 字符缓冲区

INT 21H 的功能 A 把实际字符数(不包括 Return)填入缓冲区的第二个字节，并保持 DS：

DX 指向缓冲区的第一个字节。如果想把实际字符数放入 CX 寄存器，并把指针(DS：DX)指向字符串的第一个字符，图 9.5 的程序可完成这个工作。

```
;Read a string from the keyboad ---
;  this procedure read up to 50 keys
;* * * * * * * * * * * * * * * * * * * * * *
dseg        segment
  user_string  db  50,0,50 dup(?)
dseg        ends
;-----------------------------------------
cseg        segment
            assume cs：cseg, ds：dseg
read_keys   proc    far
            push    ds                  ;set up for ret
            sub     ax,ax
            push    ax
            mov     ax,dseg             ;Initialize DS
            mov     ds,ax
            lea     dx,user_string      ;read the string
            mov     ax,0ah
            int     21h
            sub     ch,ch               ;read char count into CX
            mov     cl,user_string+1
            add     dx,2                ;make DX point to string
            ret                         ;return
read_keys   endp
;-----------------------------------------
cseg        ends
;* * * * * * * * * * * * * * * * * * * * *
            end
```

图 9.5　输入字符串程序

9.1.3.3　清除键盘缓冲区

从键盘输入的字符实际上先放在一个 16 字节的键盘缓冲区内，功能 1，7，8 和 0AH 实际上是从键盘缓冲区取得字符。

INT 21H 的功能 0CH 能清除键盘缓冲区，然后执行在 AL 中指定的功能，AL 指定的功能可以是 1，6，7，8 或 0AH，使用 0CH 可以使程序在输入一个字符之前，将以前键入的字符清除掉。

功能 0CH 的用法如下：

```
MOV   AH,0CH
MOV   AL,08H
INT   21H
```

这几条指令实际提供的输入功能是 8，它不回打，但要检测 Ctrl_Break。如果不想用 Ctrl_

Break，来结束程序，可以用功能 7 来代替功能 8。

9.1.3.4 检验键盘状态

DOS 21H 的功能 0BH 能检验一个键是否被按动，如果按下一个键，则在 AL 寄存器中放入 0FFH，如没有按下键，则在 AL 中放 00，无论哪种情况都将继续执行程序中的下一条指令。有时这是一种不可少的功能，例如希望程序保持运行状态，同时又检验键盘，看用户是否按下任意一个键来终止程序或退出循环。下面指令序列的特点是，在未按键之前，程序总是不断循环执行，只要按下任何一个键，程序就退出循环并返回。

```
SOUNDER:  ;                    ; Sound the tone
          ;
          ;
          MOV   AH, 0BH        ; get kbd status
          INT   21H            ; call DOS
          INC   AL             ; if AL not offh,then
          JNZ   SOUNDER        ; no key pressed
          RET                  ; key pressed return
```

9.2 显示器 I/O

显示器可通过两种适配器板连接到 IBM PC 机上。单色显示和并行打印机适配器连接 IBM 单色显示器和并行打印机。彩色/图形监视器适配器连接监视器或标准的电视机(彩色 TV 或黑白 TV)。下一章我们将专门介绍彩色/图形的应用。

单色适配器只能显示字符，只能用于黑白显示器。字符由标准字母、数字和符号组成，加上一些简单的图形，如菱形、矩形及笑脸等符号。

显示器的屏幕被分成 80 列 25 行，适配器就是在这个 2000 个(25×80)网格位置上显示字符。屏幕上的每个网格位置也称为一个"象素"。对应屏幕上的每个象素，存储器中都有一个相应单元，因此我们说屏幕是"存储器映象"的。这种存储器映象，使显示器电路很容易知道哪个单元的内容对应屏幕上的哪个位置。从图 9.6 中可以看到，屏幕的行号从 0 至 24，列号从 0 到 79。

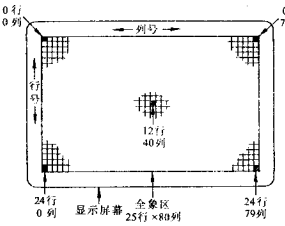


图 9.6 单色屏幕上的字符位置

9.2.1 属性

单色显示屏幕上的每个字符在存储器中由两个字节表示，一个字节保存字符的 ASCII 码，另一个字节保存字符的属性。字符的属性确定了每个要显示字符的特性，如字符是否闪烁显

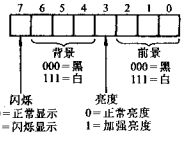


图 9.7　单色显示的属性字节

示，是否被加亮，是否反相显示。单色显示的属性字节各位分配如图 9.7。

属性可以有不同的组合，例如可以在屏幕上显示白底黑字(反相显示)代替通常的黑底白字。正常的属性是 07(二进制 00000111)，即背景为黑色(000)，前景为白色(111)，而闪烁位为正常(0)，加强亮度位也是正常(0)。为改变成反相显示，必须使背景为白色(111)，前景为黑色(000)，所以属性字节的值应为 70，即二进制 01110000。如果想要黑底白字及闪烁显示，属性值应为 87(10000111)。背景为 000，前景为 001，这种组合可产生下划线。

属性值可以任意组合，下面是一些典型的属性值。

二进制	十六进制	效　果
00000000	00	不显示
00000001	01	黑底白字，下划线
00000111	07	黑底白字，正常显示
00001111	0F	黑底白字，高亮度
01110000	70	白底黑字，反相显示
10000111	87	黑底白字，闪烁
11110000	F0	白底黑字，反相闪烁

屏幕上的字符可以按相同的属性显示，也可以按不同的属性显示，如果你设置的属性为 00H，字符就显示不出来。

单色屏幕有 2000d 个字符位置(25×80)，因每个字符需要两个字节表示，所以单色显示存储器容量为 4K 字节。单色显示存储器安排在段地址为 B000H 的区域，偏移地址从 0 到 0F9FH。图 9.8 表明了这部分存储器与屏幕上字符的对应关系。

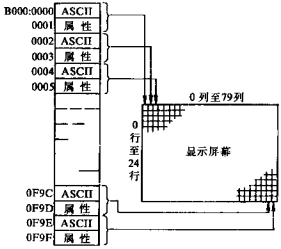


图 9.8　单色存储器与显示屏幕

9.2.2 BIOS 显示中断

表 9.6 列出了中断类型 10H 的部分显示操作及所用的寄存器。

表 9.6 类型 10H 的显示操作

AH	功 能	调用参数	返回参数
1	置光标类型	$(CH)_{5-3}$=光标开始行	
		$(CL)_{5-3}$=光标结束行	
2	置光标位置	BH=页号	
		DH=行	
		DL=列	
3	读光标位置	BH=页号	CH=光标开始行
			CL=光标结束行
			DH=行
			DL=列
6	屏幕初始化	AL=上卷行数	
	或上卷	AL=0 全屏幕为空白	
		BH=卷入行属性	
		CH=左上角行号	
		CL=左上角列号	
		DH=右下角行号	
		DL=右下角列号	
7	屏幕初始化	AL=下卷行数	
	或下卷	AL=0 全屏幕为空白	
		BH=卷入行属性	
		CH=左上角行号	
		CL=左上角列号	
		DH=右下角行号	
		DL=右下角列号	
8	读光标位置	BH=显示页	AH=属性
	的属性和字符		AL=字符
9	在光标位置	BH=显示页	
	显示字符及	AL=字符	
	其属性	BL=属性	
		CX=字符重复次数	
A	在光标位置	BH=显示页	
	只显示字符	AL=字符	
		CX=字符重复次数	

9.2.2.1 控制光标

光标不是ASCII字符表中的字符，计算机有专门的硬件来控制光标。我们熟悉的光标符一般是一个下划线或方块符。

利用INT 10H的功能1使光标显现或关闭。这个功能也控制光标行的开始和结束，也就是说控制光标的大小。表示光标行开始和结束的数据分别放在CH和CL的低4位(0～3)，当CH的第4位为1时，光标不显现出来(关闭)；当第4位为0时，光标在屏幕上显现出来。单色显示器的光标大小的范围从0到13。

INT 10H的功能2设置光标位置。光标新位置的行号设在DH寄存器中，列号设在DL中。行列设在(0,0)是屏幕的左上角，(24,79)是屏幕的右下角。BH中必须包含被输出的页号，对单色显示器来说，页号总是0。

例 9.1 置光标开始行为5，结束行为7，并把它设置到第5行第6列。

```
MOV    CH,5        ; Beginning of cursor and turn on
MOV    CL,7        ; End of cursor
MOV    AH,1        ; define cursor
INT    10H         ; call BIOS
MOV    DH,4        ; row 5
MOV    DL,5        ; column 6
MOV    BH,0        ; page 0
MOV    AH,2        ; place cursor
INT    10H         ; call BIOS routine
```

9.2.2.2 读光标位置

10H的功能3是读光标位置，页号必须在BH中指定。此功能把光标位置的行号回送给DH，列号回送给DL。光标大小的参数填入CH和CL。也就是说，在CH和CL中回送的是用功能1设置的光标参数。

例 9.2 读0页的当前光标位置。

```
MOV    AH,3
MOV    BH,0
INT    10H
```

9.2.2.3 清屏和卷屏

10H的功能6能使屏幕内容上卷指定的行，这个功能需要设置7个参数。如果屏幕的起始行列不为(0,0)，结束行列不为(24,79)，则屏幕只有指定的一部分具有上卷的功能，这个屏幕上的部分区域我们叫做窗口(Window)，象这样的窗口我们可以在屏幕上设置多个，这些窗口都可独立使用。如果上卷超过指定窗口的顶部，这些行的内容就被丢失，出现在窗口底部的新行被填为空格，其属性由BH寄存器决定。

如果AL=0，则实际完成的工作是清除屏幕的功能，它将按AL中的BLANK字符(0)使指定的窗口为空白。

10H的功能7和功能6类似，也能使屏幕(或窗口)初始化或使屏幕(或窗口)的内容下卷指定的行。它的其它参数的设置与功能6一样。请看下面的例9.3。

例 9.3 清除左上角为(0,0)，右下角为(24,39)的窗口，初始化为反相显示，该窗口相当于全屏幕的左半部分。

```
MOV    AH,7        ; scroll downward function
```

```
MOV   AL,0        ; code to blank screen
MOV   BH,70H      ; reverse video attribute
MOV   CH,0        ; upper left row
MOV   CL,0        ; upper left column
MOV   DH,24       ; lower right row
MOV   DL,39       ;lower right column
INT   10H         ; video ROM call
```

下面我们编写一个完整的程序(例 9.4)在 PC 机上运行,此程序在屏幕的中间建立一个 20 列宽和 9 行高的窗口,然后把键入的内容在这个窗口显示出来。键入的字符将被显示在窗口的最下面一行,每当输入 20 个字符,该行就向上卷动,9 行字符输入完后,顶端行的内容丢失。

例 9.4 在屏幕中心的小窗口显示字符。

```
; WINDOW—Demonstrates video window function
; keyboard writes into a window 20 chars wide
; and 9 chars high in middle of screen
;-----------------------------------------------
prognam   segment
          assume cs : prognam
          push    ds
          sub     ax,ax
          push    ax
;clear screen,using scroll up function
          mov     ah,6              ;scroll up function
          mov     al,0              ;code to blank screen
          mov     ch,0              ;upper left row
          mov     cl,0              ;upper left column
          mov     dh,24             ;lower right row
          mov     dl,79             ;lower right column
          mov     bh,7              ;blank line attribute
          int     10h               ;video ROM call
;position cursor at bottom of window
pos_curse:
          mov     ah,2              ;position cursor func.
          mov     dh,16             ;starting row
          mov     dl,30             ;starting column
          mov     bh,0              ;current page
          int     10h               ;video ROM call
;get characters from key board
          mov     cx,20             ;set count to 20
get_char:
          mov     ah,1              ;kbd input function
          int     21h               ;call DOS
          cmp     al,3              ;if char is ctrl_c
          jz      exit              ;then exit
```

```
                loop    get_char
; scroll up
                mov     ah,6                    ;scroll up function
                mov     al,1                    ;number of lines
                mov     ch,8                    ;upper left row
                mov     cl,30                   ;upper left column
                mov     dh,16                   ;lower right row
                mov     dl,50                   ;lower right column
                mov     bh,7                    ;normal attribute
                int     10h                     ;video ROM call
                jmp     pos_curse               ;go reset cursor
exit:           ret                             ;return
prognam         ends
;-----------------------------------------
                end                             ;end assembly
```

图 9.9 WINDOW 程序

此程序使用了几种 ROM 显示例行程序:清除屏幕,光标定位和上卷。如果在屏幕上同时有几个窗口工作,就要分别清除它们,这可通过设置不同的左上角坐标和右下角坐标来完成。

9.2.2.4 字符显示

10H 的功能 9 和功能 A 都能把一个字符传送到显示屏幕,然后光标返回到它的初始位置,所以在当前光标位置上写一字符之后,必须用 INT 10H 的功能 02 移动光标到下一个字符位置上。

这两种功能的区别是:AH=9 的功能把字符及其属性输出到当前光标位置上,而 AH=0AH 的功能只输出字符,它的属性值就是这一位置上先前已具有的属性。0AH 功能在使用黑白显示器时特别方便,因为此时我们很少改变显示字符的属性。

例 9.5 置光标到 0 显示页的(20,25)位置,并以正常属性显示一个星号'*'。

```
MOV   AH,2      ; set cursor position
MOV   BH,0      ; page 0
MOV   DH,20     ; row 20
MOV   DL,25     ; column 25
INT   10H       ; video ROM call
MOV   AH,9      ; write character
MOV   AL,'*'    ; character '*'
MOV   BH,0      ; page 0
MOV   BL,7      ; normal attribute
MOV   CX,1      ; number of repeat char.
INT   10H
```

10H 的功能 8 读取当前光标位置的字符及属性。

例 9.6 在 0 显示页的(11,0)位置读取字符和属性。

```
MOV   AH,2      ; set cursor position
MOV   BH,0      ; page 0
MOV   DH,11     ; row 11
MOV   DL,0      ; column 0
```

```
INT   10H          ; video ROM call
MOV   AH,8         ; read char and attr.
MOV   BH,0         ; page 0
INT   10H          ; video ROM call
```

9.2.3 DOS 显示功能调用

表 9.7 为 INT 21H 的显示操作，其中有两个是显示单字符的功能，另一个是显示字符串功能，这些功能都自动向前移动光标。

表 9.7 INT 21H 显示操作

AH	功能	调用参数	
2	显示一个字符 (检验 Ctrl_Break)	DL=字符	光标跟随字符移动
6	显示一个字符 (不检验 Ctrl.Break)	DL=字符	光标跟随字符移动
9	显示字符串	DS：DX=串地址	串必须以 $ 结束，光标跟随串移动

AH=9 的功能是显示字符串，它要求被显示输出的字符必须以 $ 字符(24h)作为定界符，此功能是用 $ 作为标记来计算串的长度的。有些 ASCII 码，如控制码，不能出现在该字符串中。显示字符串时，如果希望光标能自动换行，那么可在字符串结束之前加上回车和换行的 ASCII 码。

```
MESSAGE DB 'The sort operation is finished.',13,10,'$'
```

要显示输出的信息一般定义在数据段。输出该字符串的指令为：

```
MOV   AH,9
MOV   DX,SEG MESSAGE
MOV   DS,DX
MOV   DX,OFFSET  MESSAGE
INT   21H
```

使用赋值伪操作可以使程序的可读性更好，另外也可以根据显示格式的要求使用 TAB 符，TAB 的 ASCII 码为 09。

```
CR    EQU  13  (或 CR EQU 0DH)
LF    EQU  10  (或 LF EQU 0AH)
TAB   EQU 09
MESSAGE  DB  TAB,'The sort operation is finished.'
         DB  CR, LF, '$'
```

使用 INT 21H 显示字符串，一定要在显示串之后加上定界符 $，丢失定界符可能会在屏幕上引起意想不到的后果。

9.3 打印机 I/O

击打式的字符打印机大都实现全字符打印，即整个字符由一次打印动作完成，这种打印机

的字迹清晰，还能提供点划线、阴影、粗体字符等，速度在每分钟1000行左右。随着打印字符种类的增加，特别是打印图形和汉字的需要，发展了点阵式打印机。

点阵式打印机的字符是以点阵的形式构成的，一般是7×7或9×9的点阵。它可打印出点阵图形，斜体字，粗体双线字符以及笑脸、心形等字符。

激光打印机既具有点阵打印机的图形功能，也能打印出和字符打印机一样的高质量字符。

打印机的接口有并行接口，即一次从处理器接收8位代码，还有串行打印接口，即每次从处理器接收一位代码。IBM PC适用并行接口。

许多打印机都具有能存储几千个字符的缓存(组成缓存器)。一台打印机必须能识别并处理从处理器来的信号，如换页、换行或列表符(Tab)。处理器也必须能理解从打印机发来的表示忙或纸出界等信号。

不同类型的打印机可以响应从处理器来的不同的信号，这给打印机与接口的程序设计造成一些困难，所以在编写打印机程序之前，必须先了解连接在计算机上的打印机的型号，认真查阅打印机的技术手册。但是就打印机的处理过程而言，它比屏幕处理、磁盘处理都要简单，它只涉及到很少的一些操作，既能用DOS INT 21H来实现，也能用BIOS INT 17H来实现。表9.8是有关打印机I/O的中断操作。

表9.8 打印机I/O中断

INT	AH	功 能	调用参数	返回参数
21H	5	打印一个字符	DL=字符	
17H	0	打印一个字符 并回送状态字节	AL=字符 DX=打印机号	AH=状态字节
17H	1	初始化打印机 回送状态字节	DX=打印机号	AH=状态字节
17H	2	回送状态字节	DX=打印机号	AH=状态字节

9.3.1 DOS打印功能

INT 21H的功能5把一个字符送到打印机，字符必须放在DL寄存器中，这是唯一的DOS打印功能。如果需要回车，换行等打印功能，必须由汇编语言程序送出回车、换行等字符码。下面的功能段是送一个字符给打印机，为了连续打印，还指定了打印的字符数。当然也可以用指定的结束符来代替计数控制的方法。

```
TEXT    DB 'Hello,everybody!'
COUNT   EQU $-TEXT
        ⋮
        MOV   CX,COUNT
        MOV   BX,0
NEXT:   MOV   AH,5          ; request print function
        MOV   DL,TEXT [BX]  ; character to print
        INT   21H           ; call DOS
        INC   BX
```

```
        LOOP    NEXT
```

这些指令也适用于发送打印控制字符。例如 TEXT 字符串定义如下：

```
TEXT DB 0CH,'Hello,everybody!',0DH, 0AH, 0AH
```

字符串中的第一个字符是换页码(0CH)，最后两个字符是换行码(0AH)。用上面的指令把 TEXT 在打印机上输出，则字符串打印在新的一页的顶部，并与下文有两个空行的距离。下面我们来介绍打印机的标准控制字符和特殊控制字符。

9.3.2 打印机的控制字符

9.3.2.1 标准控制字符

打印机的标准控制字符如下：

十进制	十六进制	功　能
08	08	空格
09	09	水平 Tab (横表)
10	0A	换行
11	0B	垂直 Tab (纵表)
12	0C	换页
13	0D	回车

水平 Tab (09H)仅当打印机有此功能并被设置成打印机 Tab 状态时才能实现，否则打印机不执行此命令，或用打印多个空格来代替 Tab。

换行命令使打印机向前空走一行，若连续用两次换行命令，则会空出两行。

当打印机加电启动后，打印头在一页纸的顶部位置。打印机打印时，记下所打印的行数，并检查是否到了一页的最大行数(如 55 行/页)，如打印了最后一行，就执行一个换页命令(0CH)，然后再把行计数器置成 0。

一般显示器遇到显示文件中的 Tab 字符(09)，就把当前的光标位置移到 8,16,24,…等字符位置上。但许多打印机并不认识 Tab 字符，若要打印一个 ASCII 码文件(如汇编源程序)，就必须检验送到打印机的每个字符，若该字符是 Tab，就要插入空格，使下一个字符的位置在 8,16…等。形成 Tab 终止位置的方法由下面三个例子来说明。

当前打印的位置：	1	9	21
二 进 制 数：	00000001	00001001	00010101
清最右边的二位：	00000000	00001000	00010000
加 8 ：	00001000	00010000	00011000
新的 Tab 终止位置：	8	16	24

图 9.10 列出的程序是一个能打印 ASCII 文件的子程序(PRTASC)，它的基本功能是把已读到输入缓冲区(recarea)中的字符送到一个打印区(prtline)，并检查行尾、文件尾，处理换行符和 Tab 符。

```
;printing an ASCII file
dseg    segment
recarea db  512 dup(' ')    ;input area for ASCII file
prtline db  82  dup(' ')    ;print line
count   dw  0
dseg    ends
```

```
cseg    segment
        assume cs : cseg,ds : dseg,es : dseg
main    proc    far
        push    ds
        sub     ax,ax
        push    ax
        mov     ax,dseg
        mov     ds,ax
        mov     es,ax
        call    prtasc
        ret
main    endp
;
;Transfer data to print line;
prtasc  proc    near
        cld                             ;set left to right
        lea     si,recarea              ;initialize
p10:    lea     di,prtline
        mov     count,0
p20:    lea     dx,recarea+512
        cmp     si,dx                   ;end of record
        je      p70                     ;yes,exit
p30:    mov     bx,count
        cmp     bx,80                   ;at end of print line?
        jb      p40                     ;no,bypass
        mov     word ptr[di+bx],0d0ah   ;set CR/LF
        add     count,2
        call    subprt                  ;print this line
        lea     di,prtline              ;reinitialize
        mov     count,0
        mov     bx,0
p40:    lodsb                           ;[si] to al,inc si
        mov     [di+bx],al              ;char to print line
        inc     bx
        cmp     al,1ah                  ;end of file
        jne     p70
        cmp     al,0ah
        jne     p50
        call    subprt                  ;print a line
        jmp     p10
p50:    cmp     al,09                   ;Tab char ?
        jnc     p60
        dec     bx                      ;yes,reset BX
        mov     byte ptr [di+bx],20h    ;clear Tab to blank
```

```
                and     bx,0fff8h               ;clear rightmost 3 bits
                add     bx,08                   ; & add 8--->Tab stop
p60:            mov     count,bx
                jmp     p20
p70:            mov     bx,count                ;end of file
                mov     byte ptr [di+bx],0ch    ;form feed
                call    subprt                  ;print last line
                ret
prtasc          endp
;-------------------------------------------
subprt          proc    near
                mov     cx,count                ;length
                inc     cx
                mov     bx,0
next:           mov     ah,5                    ;request print
                mov     dl,prtline[bx]
                int     21h
                inc     bx
                loop    next
                mov     ax,2020h                ;clear print line
                mov     cx,41
                lea     di,prtline
                rep     stosw
                ret
subprt          endp
;-------------------------------------------
cseg            ends
                end     main
```

图 9.10　打印 ASCII 文件的程序

9.3.2.2　特殊的打印命令

我们已经讨论了打印机基本控制命令的使用。还有一些命令包括：

十进制	十六进制	功　能
15	0F	设置紧缩方式
14	0E	设置扩展方式
18	12	取消紧缩方式
20	14	取消扩展方式

一些命令需要和 Esc(1BH)字符一起使用，这些命令是：

1B	30	设置每英寸为 8 行
1B	32	设置每英寸为 6 行
1B	45	设置加重打印方式
1B	46	取消加重打印方式

我们可以用两种不同的方式把命令码发送给打印机。

1. 在数据区中定义命令码。下述数据区中的命令是设置紧缩方式，每寸 8 行，打印一个标题，并发送回车、换行字符。

```
HEAD    DB    0FH ,1BH,30H,'Title…', 0DH,0AH
```

2. 直接用指令方式

```
MOV   AH,05        ; 请求打印
MOV   DL,0FH       ; 设置紧缩方式
INT   21H
```

上面的指令使以后打印的字符都是以紧缩方式打印，只有当程序发送取消此方式的命令后，才变成正常的方式进行打印。

上述这些特殊命令并不适用于所有型号的打印机，这就需要查阅打印机的手册，看其是否具有执行这些特殊命令的功能。

9.3.3 BIOS 打印功能

BIOS 17H 中断指令提供了由 AH 寄存器指定的三种不同的操作。

BIOS 中断 17H 的功能 0 是打印一个字符的功能。要打印输出的字符放在 AL 中，打印机号放在 DX 中，BIOS 最多允许连接三台打印机，机号分别为 0，1 和 2。如果只有一台打印机，那么就是 0 号打印机，打印机的状态信息被回送到 AH 寄存器。

```
MOV   AH,0         ; request print
MOV   AL,char      ; character to be printed
MOV   DX,0         ; select printer 0#
INT   17H          ; call BIOS
```

17H 的功能 1 初始化打印机，并回送打印机状态到 AH 寄存器。如果把打印机开关关上然后又打开，打印机各部分就复位到初始值。此功能和打开打印机时的作用一样。在每个程序的初始化部分可以用 17H 的功能 1 来初始化打印机。

```
MOV   AH,01        ; request initialize printer
MOV   DX,0         ; select printer 0#
INT   17H          ; call BIOS
```

这个操作要发送一个换页符，因此这个操作能把打印头设置在一页的顶部。对于大多数打印机，只要一接通电源，就会自动地初始化打印机。

BIOS 17H 的功能 2 把状态字节读入 AH 寄存器。打印机的状态字节如图 9.11 所示。

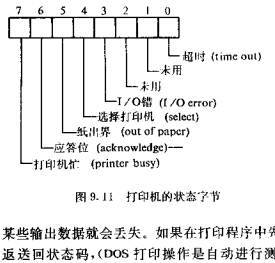


图 9.11 打印机的状态字节

打印机忙(Printer busy)表示打印机正在接收数据，或正在打印，或处于脱机状态。应答位(acknowledge)表示打印机已发出一个表明它已经接收到数据的信号。选择位(select)表示打印机是联机的。超时位(Time out)表示打印机发出忙信号很长一段时间了，系统将不再给它传送数据。表示打印机出错的是第 5 位(纸出界)或第 3 位(I/O 错)为 1。如果打印机没有接上电源，没有装上纸或没有联机，而打印程序已开始运行，这时显示器的指示光标会不停地闪烁，当接通打印机的电源后，某些输出数据就会丢失。如果在打印程序中先安排指令测试打印机的状态，则 BIOS 操作就会返送回状态码，(DOS 打印操作是自动进行测试的，但对各种情况都显示一个"纸出界"的信息)。当打印机接通电源后，即开始正常打印，而且不丢失任何数据。

下面我们应用前面介绍的 BIOS 和 DOS 功能调用，编写一个简单的打字程序(TYPER)。它要求把从键盘上接收的字符显示在屏幕上，并由打印机输出，在键盘上按下 Ctrl_C，即退出程序。

```
;* * * * * * * * * * * * * * * * * * * * * * * *
prog_stack  segment stack 'stack'
        db   64 dup ('stack...')
prog_stack  ends
;-----------------------------------------------
prog_data   segment 'data'
inchar  db      20 dup (?)
prog_data   ends
;-----------------------------------------------
prog_code   segment 'code'
typ_int     proc        far
            assume      cs:prog_code, ds:prog_data
            assume      ss:prog_stack,es:prog_data
            push        ds
            mov         ax,0
            push        ax
            mov         ax,prog_data
            mov         ds,ax
            mov         es,ax
            sti
            cld
            mov         ah,0                    ;Initializing printer
            mov         dx,0
            int         17h
            mov         ah,6                    ;clear screen
            mov         al,0
            mov         ch,0
            mov         cl,0
            mov         dh,24
            mov         dl,79
            mov         bh,7
            int         10h
            mov         ah,2                    ;display prompt char.
            mov         dl,10h
            int         21h
input_char:
            mov         ah,1                    ;input with echo & check
            int         21h
check_char:
            cmp         al,0                    ;if any non_ASCII code
            je          end_prog
output_char:
            mov         dl,al                   ;print a character
            mov         ah,5
            int         21h
```

```
        cmp     al,0dh                  ;check for CR
        jne     input_char
        mov     dl,0ah                  ;if CR and LF
        mov     ah,5
        int     21h
        mov     ah,2                    ;LF on screen also
        int     21h
        jmp     input_char
end_prog:
        ret
typ_int endp

prog_code ends
;*********************************
        end
```

图 9.12 TYPER 程序

9.4 串行通讯口 I/O

9.4.1 串行通讯接口

串行接口主要用作微机与微机，或微机与特定类型的设备之间的接口。由于计算机通讯的广泛应用，串行接口已成为个人计算机必备的部件，IBM PC 机内装有通讯适配器板，这使得 PC 机可以和其它配有串行通讯接口的计算机或设备进行通讯。串行接口每次由 CPU 得到 8 位的数据，然后串行地通过一条线路，每次发送一位将该数据发送出去。

最常用于个人计算机上的串行接口是标准的 RS－232 串行接口，它装于 PC 机内的通讯适配器板上。这个标准串行接口既可用于近程或远程的数据通讯，又可用来连接附加的一些外部设备，如具有串行接口的绘图仪，数字化仪，汉字或西文终端，各式打印机等。每个系统中可以有两个或多个串行控制器连接到不同的外设上，如 IBM PC 可连接两个串行接口（COM1 和 COM2）。但程序每次只能对其中一个接口进行存取。

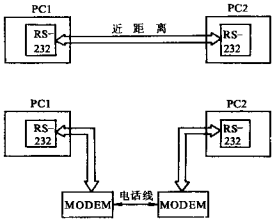


图 9.13 两台 PC 机串行通讯的连接方式

两台 PC 机或设备进行近距离通讯时，可直接将它们连接。当它们进行远距离通讯时，要使用调制解调器（MODEM）连接到电话线上，因为 RS－232 标准串行接口输出的是电压信号，不能直接接到电话线上，调制解调器把代表逻辑 1 和逻辑 0 的电压信号转换成能在电话线上传输的不同频率的信号。电话线另一端的调制解调器又把这些不同频率的信号转换成接口要求的电压信号。图 9.13 为两种连接方式的示意图。

与前面介绍的 DOS 对键盘、显示器和打印机的支持相比，DOS 对串行通讯口的支持

相对要弱一些，在这种情况下用户有时要编写一些对硬件依赖性较强的程序以获得较好的传输指标。

9.4.2 DOS 串行通讯口功能

DOS 手册中称串行通讯接口为辅助设备。INT 21H 的功能 03H 是从辅助设备(第一个串行口 COM1)读一个字符到寄存器 AL。功能 04H 将 DL 寄存器中的字符传送给串行设备，如果输出设备正忙，该功能调用等待，直到设备准备好接收字符。

表 9.9 DOS 串行通讯口功能

AH	调用参数	返回参数
3		AL=输入的 8 位数据
4	DL=输出的 8 位数据	

要注意的是，在多数 DOS 系统中，串行设备没有缓冲和中断，如果串行通讯口或其它辅助设备送的数据比程序处理数据快，字符可能丢失。在 PC 系统中，第一个串行口 COM1 被初始化为 2400 波特，无奇偶校验位，1 个终止位和 8 位数据。其它机器上的 DOS 实现可能有不同的初始化。

例 9.7 从串行通讯口输入一字符并存入 input_char 单元中。

```
        MOV   AH,3
        INT   21H
        MOV   INPUT_CHAR,AL
        :
INPUT_CHAR  DB  0
```

例 9.8 将字符串 hello 输出到串行通讯口。

```
        MOV   BX, SEG BUFFER        ;DS + BX=addr of string
        MOV   DS, BX
        MOV   BX, OFFSET BUFFER
        MOV   CX, BUF_LEN           ;CX=length of string
NEXT:   MOV   DL, [BX]              ;take the next char.
        MOV   AH,4                  ;AUX output
        INT   21H                   ;call DOS
        INC   BX                    ;inc pointer
        LOOP  NEXT
        :
BUFFER  DB    'HELLO'
BUF_LEN EQU   $-BUFFER
```

DOS 没有提供读辅助设备的状态和检测 I/O 错误(如丢失字符等)的功能，但 ROM 中 BIOS INT 14H 提供了这些功能。

9.4.3 BIOS 串行通讯口功能

IBM PC 及其兼容机提供了一种有较强的硬件依赖性，但却比较灵活的串行口 I/O 的方法，即通过 INT 14H 调用 ROM BIOS 串行通讯口例行程序。该例行程序包括将串行口初始化为指定的字节结构和传输速率，检查控制器的状态，读写字符等功能。

表 9.10 串行通讯口 BIOS 功能(INT 14H)

AH	功能	调用参数	返回参数
0	初始化串行通讯口	AL=初始化参数 DX=通讯口号 COM1=0 COM2=1	AH=通讯口状态 AL=调制解调器状态
1	向串行通讯口写字符	AL=所写字符 DX=通讯口号 COM1=0 COM2=1	写字符成功: $(AH)_7=0$ (AL)不变 写字符失败: $(AH)_7=1$ $(AH)_{0\text{-}6}$=通讯口状态
2	从串行口读字符	DX=通讯口号 COM1=0 COM2=1	读成功: $(AH)_7=0$ (AL)=字符 读失败: $(AH)_7=1$ $(AH)_{0\text{-}6}$=通讯口状态
3	取通讯口状态	DX=通讯口号 COM1=0 COM2=1	(AH)=通讯口状态 (AL)=调制解调器状态

INT 14H 的 AH=0 功能把指定的串行通讯口初始化为希望的波特率,奇偶性,字长和终止位的位数。这些初始化参数设置在 AL 寄存器中,其各位的含义如图 9.14。

例 9.9 要求 0 号通讯口的传输率为 2400 波特,字长为 8 位,1 位终止位,无奇偶校验。

```
MOV   AH,0      ; Initialize communication
MOV   AL,0A3H   ; 0A3H=10100011B
MOV   DX,0      ; COM1
INT   14H       ; call BIOS
```

返回参数中通讯口状态字节各位置 1 的含义如图 9.15。

在接收和发送过程,错误状态位(1,2,3,4 位)一旦被置为 1,则读入的接收数据已不是有效数据,所以在串行通讯应用程序中,应检测数据传输是否出错。

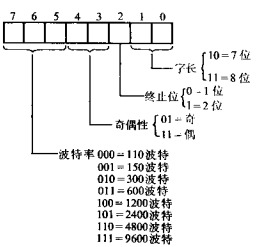


图 9.14 串行通讯口初始化参数

奇偶错:通信线上(尤其是用电话线传输时)的噪音引起某些数据位的改变,产生奇偶错。通常检测出奇偶错时,要求正在接收的数据至少应重新发送一段。

超越错:在上一个字符还未被处理机取走,又有字符要传送到数据寄存器里,则会引起超

越错。如果处理机处理字符的速度小于串行通讯口的波特率，则会产生这种错误。

帧格式错：当接收/发送器未接收到一个字符数据的停止位，则会引起帧格式错。这种错误可能是由于通信线上的噪音引起停止位的丢失，或者是由于接收方和发送方初始化不匹配。

间断：间断有时候并不能算是一个错误，而是为某些特殊的通讯环境设置的"空格"状态。当间断位为 1 时，说明接收的"空格"状态超过了一个完整的数据字传输时间。

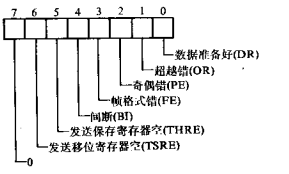


图 9.15　串行通讯口状态字节

例 9.10　从通讯口 0 读入字符并把它们显示出来，如果字符没有准备好则等待，如果传送有错则显示出错信息"?"。

```
CHECK:  MOV   AH,3        ; Read COMM port status
        MOV   DX,0        ; COM1
        INT   14H         ; call BIOS
        AND   AH,1        ; character ready?
        JZ    CHECK       ; no,check again
        MOV   AH,2        ; yes,read
        MOV   DX ,0       ; a character from COM1
        INT   14H
        TEST  AH ,80H     ; read availably?
        JNZ   ERROR       ; no,error
        AND   AL,7FH      ; yes,reserve character
        MOV   BX,0        ;
        MOV   AH,0EH      ; display function
        INT   10H         ; call BIOS
        JMP   CHECK       ; for next
ERROR:  MOV   AL,'?'      ; error message
        MOV   BX,0
        MOV   AH,0EH      ; display function
        INT   10H         ; call BIOS
```

调制解调器状态字节各位置 1 的含义如图 9.16。

调制解调器的状态字节的第 0—3 位，实际上是用来指出第 4—7 位所对应的 MODEM 的四个现行状态信号从 CPU 上次读过状态字节之后，状态又有了改变。

BIOS 提供的串行口例行程序没有采用中断方式传送数据，所以当传输速率大于 2400 波特时，BIOS 例行程序在传输时有可能丢失字符。如果要求 PC 机来承担快速文件的传送，程序员就要利用串行通讯口的 I/O 端口编写高性能的应用程序，并提供自己的中断处理程序。编写这样的程序需要对硬件接口有较多的了解，对 IBM PC 来说，就要了解 8250(或 8251)异步串行通讯接口和 8259 可编程中断控制器。下面是使用 8250 中断方式进行通讯时，程序初始化的几步工作：

(1) 在主程序中为串行输入、输出建立一个循环队列(一个先进先出的数据缓冲区)。

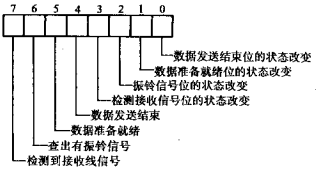


图 9.16 调制解调器状态字节

(2) 对 8259 中断控制器的中断屏蔽寄存器(I/O 端口 21H)初始化,即允许串行通讯口中断。

(3) 将 8250 中断子程序的入口地址填入中断向量表的向量单元 0CH×4

(4) 初始化串行通讯接口的中断允许寄存器(I/O 端口 3F9H),允许"接收数据就绪"中断(第 0 位置 1)。

(5) 当接收数据准备好,发生中断时,中断子程序将数据从接收器数据寄存器传送到循环队列,并把状态寄存器(I/O 端口 3FDH)反映出来的错误状态位返回给主程序。

习 题

9.1 INT 21H 的键盘输入功能 1 和功能 8 有什么区别?

9.2 编写一个程序,接收从键盘输入的 10 个十进制数字,输入回车符则停止输入,然后将这些数字加密后(用 XLAT 指令变换)存入内存缓冲区 BUFFER。加密表为:

输入数字: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

密码数字: 7, 5, 9, 1, 3, 6, 8, 0, 2, 4

9.3 对应黑白显示器屏幕上 40 列最下边一个象素的存储单元地址是什么?

9.4 写出把光标置在第 12 行,第 8 列的指令。

9.5 编写指令把 12 行 0 列到 22 行 79 列的屏面清除。

9.6 编写指令使其完成下列要求:

1) 读当前光标位置

2) 把光标移至屏底一行的开始

3) 在屏幕的左上角以正常属性显示一个字母 M

9.7 写一段程序,显示如下格式的信息:

Try again,you have n starfighters left.

其中 n 为 CX 寄存器中的 1—6 之间的二进制数。

9.8 从键盘上输入一行字符,如果这行字符比前一次输入的一行字符长度长,则保存该行字符,然后继续输入另一行字符;如果它比前一次输入的行短,则不保存这行字符。按下'$'输入结束,最后将最长的一行字符显示出来。

9.9 编写程序,它在屏幕上显示出信息"What is the date(mm/dd/yy)?"并响铃(响铃符为 07),然后从键盘接收数据,并按要求的格式保存在 date 存储区中。

9.10 用户从键盘输入一文件并在屏幕上回显出来。每输入一行(≤80 个字符),用户检查一遍,如果用户认为无须修改,则键入回车键,此时这行字符存入 BUFFER 缓冲区保存,同时打印机把这行字符打印出来并回车。

第十章　单色和彩色图形显示

编制图形程序是程序设计中非常有趣和有价值的工作之一。在图形领域中，汇编语言具有潜在的优点，因为显示屏幕上的一个图象由成千上万个元素组成，处理这些图象需要大量的指令。以速度而论，汇编语言远比高级语言快得多，最高级的图形技术，例如动画，只能以汇编语言设计才更逼真，更有效。

本章将介绍 IBM PC 的图形操作以及单色和彩色绘图所使用的一些常用方法。

10.1　显 示 方 式

IBM PC 的标准显示器适配器，一种是第九章介绍的单色显示和并行打印机适配器，它只能显示黑白字母、数字、符号和菱形、方块等简单的图形字符，很显然，它是以文本方式工作的。另一种是彩色/图形监视器适配器，它能以文本和图形两种方式工作，既可以显示黑白图形又可以显示有 16 种颜色的彩色图形。

采用文本方式时，彩色/图形适配器能产生 80×25 字符的高分辨率显示或 40×25 字符的低分辨率显示，因此，它具有 16K 字节的存储器作为显示缓冲区，能存储 4 页(0—3)80×25 字符的象素或 8 页(0—7)40×25 字符的象素。

表 10.1　INT 10H 设置显示方式功能

AH	调用参数	显 示 方 式	
00	AL=00	40×25	黑白文本方式
	AL=01	40×25	彩色文本方式
	AL=02	80×25	黑白文本方式
	AL=03	80×25	彩色文本方式
	AL=04	320×200	彩色图形方式
	AL=05	320×200	黑白图形方式
	AL=06	640×200	黑白图形方式

在图形方式中，彩色/图形适配器把屏幕分成 m×n 的点阵，每个点的坐标上的图象元素就是一个象素(Pels)。彩色/图形适配器提供了两种选择：320×200 中等分辨率和 640×200 高分辨率。对 320×200 中等分辨率，每个象素可以四种不同的颜色显示，背景可以有 16 种颜色。对高分辨率，只支持黑白显示器。

ROM BIOS 显示例行程序提供了设置各种显示方式的功能。表 10.1 列出了常用的几种显示方式。

从表 10.1 可以看出，AL＝0—3 时设置文本方式，缺省的方式是 AL＝0 的文本方式。AL＝4 以上为图形方式，一旦置为图形方式，光标即消失。

下例是将显示方式设置为彩色文本方式：

```
MOV   AH,00          ;set video mode
MOV   AL,03          ;80×25 color text
INT   10H            ;video ROM call
```

如果我们为未知的显示器写软件，可以先用 BIOS INT 11H 来确定配置到系统的设备。这个操作实际上是把设备标志(Equipment Flag)的值回送给 AX，其中第 5 位和第 4 位表示显示

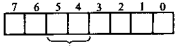


图 10.1　设备标志位

器的配置及初始的显示方式。

设备标志包括在“ROM BIOS Data Areas”部分，它的绝对地址为 0040:0010。当 PC 第一次加电时，ROM BIOS 的类型 11H 的中断例行程序读取外部硬件开关状态(on/off)保留在设备标志单元中，通过对其第 4 位和第 5 位的测试，设置所要采用的显示器类型，然后再设置显示方式。设备标志的其它位对应其它的硬件设置，当更换第 4、5 位的值时，其它各位的设置不可丢失。

如果 PC 机原来使用单色显示，但同时还装配有彩色/图形适配板，我们就可以设置所希望的显示方式。例 10.1 的 CHAMODE 程序向我们提供了三种选择：

键入“m”，对应“单色”，是正常的黑白显示。

键入“h”，对应“高分辨率”，是 640×200 黑白显示。

键入“c”，对应“彩色”，是 320×200 彩色显示。

当做出选择后，程序修改设备标志，并把系统设置成相应的显示方式。为了多样性和快速装入，此程序使用 COM 文件格式。

例 10.1　选择显示方式的程序

```
;CHAMODE -- Program to change screen modes
key_in   equ      1h          ;keyboard input
pstring  equ      9h          ;print string
doscall  equ      21h         ;DOS interrupt num.
; * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;Segment to contain equipment flag
rom_da   segment at 40h
         org   10h
eq_flag   dw   ?
rom_da   ends
; * * * * * * * * * * * * * * * * * * * * * * * * * * * *
codeseg  segment
          assume cs : codeseg
          assume ds : codeseg
          org 100h
start:
;print intro message
          mov       dx,offset mess          ;addr of mess
          mov       ah,pstring              ;print string
          int       doscall                 ;call DOS
;set data segment to equipment flag
          assume    ds : rom_da
          mov       ax,rom_da               ;set DS to
          mov       ds,ax                   ;equipment flag
;get input letter
          mov       ah,key_in               ;kbd input
```

```
        int     doscall             ;call DOS
        cmp     al,'m'              ;is it "m"?
        je      mono                ;monochrome
        cmp     al,'h'              ;is it "h"?
        je      hi_res              ;hi res b and w
        cmp     al,'c'              ;is it "c"?
        je      color               ;320 * 200 color
        jmp     start               ;unknown input
;set up for monochrome display
mono:
        mov     ax,eq_flag          ;get equipment flag
        and     ax,11001111b        ;mask off video bits
        or      ax,00110000b        ;monochrome bits
        mov     eq_flag,ax          ;back into flag
        mov     al,2                ;80 column B & W
        mov     ah,0                ;set mode func.
        int     10h                 ;call video BIOS
        jmp     exit
;set up for 640 * 200 BW
hi_res:
        mov     ax,eq_flag          ;get equipment flag
        and     ax,11001111b        ;mask off video bits
        or      ax,00100000b        ;80 * 25 color card
        mov     eq_flag,ax          ;back into flag
        mov     al,6                ;640 * 200 BW
        mov     ah,0                ;set mode function
        int     10h                 ;call video BIOS
        jmp     exit
;set up for 320 * 200 color mode
color:
        mov     ax,eq_flag          ;get equipment flag
        and     ax,11001111b        ;mask equipment flag
        or      ax,00100000b        ;mask off video bits
        mov     eq_flag,ax          ;back into flag
        mov     al,4                ;320 * 200 color
        mov     ah,0                ;set mode function
        int     10h                 ;call video BIOS
        jmp     exit
exit:   ret
;sign_on message
mess    db  13,10
        db  'Type "m" for 80 * 25 monochrome'
        db  13,10
        db  '       "h" for 640 * 200 b & w',13,10
```

```
        db   '    "c" for 320 * 200 color',13,10
        db   'selection: $'
codeseg ends
;*************************************
        end      start
```

图 10.2 CHAMODE 程序

这个程序相当简单，在打印提示信息之后，用一个 DOS 该键盘功能确定三种方式选择了哪一种，然后转移到相应的例行程序。在每个例行程序中都有相应的一位或若干位被"或"到设备标志上，而不会影响其它位。然后，根据 AL 所置的数值，设置成所选择的显示方式。

在引用设备标志之前，DS 寄存器必须装入段地址，这个段地址是 40h。但是，只有在打印信息之后才能装入此段地址，这是因为打印信息位于代码段中，为了访问打印信息，DS 必须包含代码段的地址。一旦信息打印完后，就可改变 DS 的内容。

10.2 文本方式

文本方式通常在屏幕上处理字母、数字以及一些字符图形。对应屏幕上的每个字符位置有两个存储器字节，一个是 ASCII 码字节，另一个是属性字节。

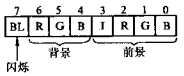


图 10.3 彩色属性字节

在文本方式下，属性字节对单色显示和彩色显示都是有效的。对单色文本方式属性提供了闪烁，反相和高亮度。对彩色文本，属性字节能使你选择前景（显示的字符）和背景的颜色，每个字符可以选择 16 种颜色中的一种，背景有 8 种颜色可以选择。图 10.3 是彩色文本方式显示的属性字节。

前景的 16 种颜色由位 0—3 组合，RGB 分别表示红、绿、兰，BL 表示闪烁，I 为亮度，闪烁和亮度只应用于前景。表 10.2 为彩色文本方式颜色的组合。

表 10.2 16 种颜色的组合

颜色	I R G B	颜色	I R G B	颜色	I R G B	颜色	I R G B
黑	0 0 0 0	灰	1 0 0 0	红	0 1 0 0	浅红	1 1 0 0
兰	0 0 0 1	浅兰	1 0 0 1	品红	0 1 0 1	浅品红	1 1 0 1
绿	0 0 1 0	浅绿	1 0 1 0	棕	0 1 1 0	黄	1 1 1 0
青	0 0 1 1	浅青	1 0 1 1	灰白	0 1 1 1	白	1 1 1 1

显示屏幕的背景颜色只能是表中 I 为 0 的 8 种颜色。如果前景和背景是同一种颜色，显示出的字符是看不见的，但属性字节中的位 7 可以使字符闪烁（BL=1）。

在编写字符显示程序时，彩色显示和单色显示类似。例如前一章介绍的 BIOS 10H 的 09 功能显示字符，对单色显示表示字符闪烁、反相及亮度的属性值置入 BL 寄存器，对彩色显示，BL 中的值为前景和背景的彩色属性值。

表 10.3 属性字节的典型组合

显示颜色 \ 位	7	6	5	4	3	2	1	0	
	BL	R	G	B	I	R	G	B	16 进制
黑底黑字	0	0	0	0	0	0	0	0	00
黑底兰字	0	0	0	0	0	0	0	1	01
兰底红字	0	0	0	1	0	1	0	0	14
绿底青字	0	0	1	0	0	0	1	1	23
灰白底浅品红字	0	1	1	1	1	1	0	1	7D
绿底灰字闪烁	1	0	1	0	1	0	0	0	A8

例 10.2 在品红背景下，显示五个浅绿色闪烁的星号。

```
MOV  AH,09          ;display a character & attr
MOV  AL,'*'         ;Asterisk
MOV  BH,0           ;page #0
MOV  BL,0DAH        ;color attribute
MOV  CX,05          ;five times
INT  10H            ;call BIOS
```

除第九章"显示器 I/O"一节中介绍的 BIOS 显示功能外，INT 10H 还有几种很有用的显示功能(见表 10.4)

表 10.4 INT 10H 显示功能

AH	功能	调用参数	注释
E	显示字符	AL=字符	光标跟随字符移动
	(光标前移)	BL=前景色	
13	显示字符串	ES：BP=串地址	
		CX=串长度	
		DH，DL=起始行列	
		BH=页号	
		AL=0，BL=属性	光标返回起始位置
		串：char，char，…	
		…，char	
		AL=1，BL=属性	光标跟随移动
		串：char，char，…	
		…，char	
		AL=2	
		串：char，attr，……	
		char，attr，char，attr	
		AL=3	光标跟随串移动
		串：char，attr，……	
		char，attr，char，attr	

使用 INT 10H 的 13H 功能显示字符串有 4 种方式，前两种方式(AL=0,1)要指定整个显示字符串的属性，后两种方式(AL=2,3)必须指定每个字符的属性，我们通过下面两个例子来了解它的用法。

例 10.3 在屏幕上以红底兰字显示字符串： "WORLD SCENERY"

```
STRING      DB      'WORLD SCENERY'
LEN_STR     DB      EQU     $-STRING
            :
            MOV     AL,3                    ;select 80 * 25 color text
            MOV     AH,0                    ;change the mode
            INT     10H
            MOV     BP,SEG STRING           ;base of string
            MOV     ES,BP
            MOV     BP,OFFSET STRING        ;offset of string
            MOV     CX,LEN_STR              ;character count
            MOV     DX,0                    ;start at top left corner
            MOV     BL,41H                  ;use blue_on_red lettering
            MOV     AL,0                    ;make cursor return
            MOV     AH,13H                  ;display the string
            INT     10H                     ;call BIOS
```

图 10.001

例 10.4 在屏幕上以红底兰字显示"WORLD"，然后分别以红底绿字和红底兰字相间地显示"SCENERY"。

```
STRING1     DB      'WORLD'
STRING2     DB      'S',42H,'C',41H,'E',42H,'N',41H
            DB      'E',42H,'R',41H,'Y',42H
LEN_STR2    EQU     $ STRING2
            :
            MOV     AL,3                    ;select 80 * 25 corlor text
            MOV     AH,0                    ;change the mode
            INT     10H
            MOV     BP,SEG STRING1          ;base of first string
            MOV     ES,BP
            MOV     BP,OFFSET STRING1       ;offset of first string
            MOV     CX,STRING2-STRING1
            MOV     DX,0
            MOV     BL,41H                  ;color is blue_on_red
            MOV     AL,1                    ;character string
            MOV     AH,13H                  ;display the string1
            INT     10H
            MOV     AH,3                    ;read the cursor positions
            INT     10H
```

```
MOV     BP,OFFSET STRING2    ;offset of string 2
MOV     CX,LEN_STR2          ;caracter count
MOV     AL,3                 ;display char and attr.
MOV     AH,13H               ;display the string 2
INT     10H                  ;call BIOS
```

图 10.002

10.3 字符图形

文本方式一般用来显示信息，然而利用字符集中的方块图形字符也能产生一些简单的图形。多个方块图形字符也能组装成一个较复杂的图形，显示的方法和显示一般字符一样，调用 BIOS 的字符显示功能，如 INT 10H 的 AH＝09H，AH＝0AH 等。

例 10.5 是一个单字符图形的显示程序，它能在屏幕上沿着一条斜线的轨迹显示出笑脸符(02)，第一个笑脸的位置是(0,0)，第 2 个是(1,1)最后一个是(24,24)。此程序用了 INT 10H 的四个功能：AH＝0FH，把显示页号读入 BH；AH＝0，选择 80×25 黑白文本方式；AH＝2，移动光标；AH＝0AH，显示一个字符。

例 10.5 用"笑脸"符画一条斜线。

```
;Draws a diagonal line of 'smile faces'
;  starting at the tope left corner of the screen
;-----------------------------------------------
code_seg  segment
          assume cs:code_seg
          push    ds                    ;for return
          mov     ax,0
          push    ax
          mov     ah,0fh                ;set BH to active
          int     10h                   ;  display page
          mov     ah,0                  ;select 80 x 25 BW
          mov     al,3                  ;  text mode
          int     10h
          mov     cx,1                  ;character count=1
          mov     dx,0                  ;start at row 0 col 0
set_crs:
          mov     ah,2                  ;set new cursor
          int     10h                   ;  position
          mov     al,2                  ;display char
          mov     ah,0ah                ;write char to screen
          int     10h
          inc     dh                    ;point to next row & col
          inc     dl
          cmp     dh,25                 ;bottom of screen ?
          jne     set_crs
```

```
        ret                                    ;yes,exit
code_seg    ends
;-------------------------------------------------------
        end
```

图 10.003

多字符组成的图形需要多次显示，一次显示一个字符，当图形是由许多字符组成时，应考虑把它们定义在一个字符图形表里。字符图形表包括每个字符的 ASCII 码、属性以及在显示图形中的相对位移量。相对位移量是指前一个字符和当前要显示字符之间的行距和列距。

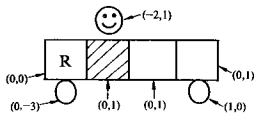


图 10.4 “汽车”各字符的相对位移量

例如图 10.4 中的汽车图形由 7 个文本字符组成，从左到右，车体是由字母 R(ASCII 码 52H)，1/2 阴影符(ASCII 码 B1H)组成和两个实心方块(ASCII 码 DBH)组成。头两个字符是反相属性显示，后两个字符是正常属性显示。两个车轮是字母 O(ASCII 码 4FH)，它和笑脸符(ASCII 码 02)都是以正常属性显示。

开始先显示车体，R 是显示的第一个字符，其相对位移量定为(0,0)，第二个阴影符在 R 的右边一列，所以相对 R 的位移量为(0,1)，两个实心方块都是在前一个字符的同一行的右边一列，所以位移量都为(0,1)。

前轮比最后一个方块符行数增加 1，但还在同一列上，所以它的相对位移量为(1,0)。后轮相对于前轮，行数不变，列 数减了 3，所以位移量应是(0,-3)。最后画出笑脸，它相对于后轮位移量应是(-2,1)。

每个字符由四个参数表示，所以由上述 7 个字符组成的汽车图形的字符图形表应定义如下：

```
CAR     DB      7               ;shape contains 7 char
        DB      52H,70H,0,0     ;rear character of body
        DB      0B1H,70H,0,0    ;driver's compartment
        DB      0DBH,7,0,1      ;next character of body
        DB      0DBH,7,0,1      ;front character of body
        DB      4FH,7,1,0       ;front wheel
        DB      4FH,7,0,-3      ;rear wheel
        DB      2,7,-2,1        ;smile face
```

图 10.004 的程序

字符图形表中第一个字节是组成图形的字符数，我们编写一个程序顺序将每个字符的 ASCII 码和属性以及显示的位置放入相应的寄存器中，然后发出显示命令，当 7 个字符都显示出来，我们就可以看到一个汽车图形，另外我们还应指定图形显示的初始位置，如汽车图形从 10 行 10 列处开始显示。

例 10.6 用字符构成一个“汽车”图形。

```
TITLE DSHAPE -- construct a shape from a shape table
;Constructs a shape on the screen using a shape table
```

```
data_seg    segment
car         db   7                    ;shape table
            db   52h,70h,0,0
            db   0b1h,70h,0,1
            db   0dbh,7,0,1
            db   0dbh,7,0,1
            db   4fh,7,1,0
            db   4fh,7,0,-3
            db   2,7,-2,1
data_seg    ends
;-------------------------------------------------
code_seg    segment
            assume cs : code_seg,ds : data_seg
main        proc   far
            push   ds                   ;for return
            sub    ax,ax
            push   ax
            mov    ax,data_seg          ;initialize DS
            mov    ds,ax
            call   clear_screen         ;clear the screen
            lea    di,car               ;DI points to car
                                        ;  shape table
            mov    dh,10                ;body is on line 10
            mov    dl,10                ;starting in col 10
            call   display_shape        ;display the car
            ret
main        endp
;-------------------------------------------------
clear_screen proc  near
            push   ax                   ;save register
            push   bx
            push   cx
            push   dx
            mov    ah,6                 ;scroll up func.
            mov    al,0                 ;code to blank screen
            mov    ch,0                 ;upper left row
            mov    cl,0                 ;upper left column
            mov    dh,24                ;lower right row
            mov    dl,79                ;lower right column
            mov    bh,7                 ;blank line attribute
            int    10h                  ;video ROM call
            pop    dx                   ;restore register
            pop    cx
            pop    bx
```

```
            pop     ax
            ret                     ;return to main program
clear_screen endp
;-----------------------------------------------------------
display_shape proc  near
            push    ax              ;save registers
            push    bx
            push    cx
            push    dx
            push    di
            mov     ah,0fh          ;set BH to active page
            int     10h
            sub     ch,ch           ;clear high byte of count
            mov     cl,[di]         ;Cl holds char count
            inc     di              ;DI points to first char
next_char:
            add     dh,[di+2]       ;update row pointer
            add     dl,[di+3]       ;and column pointer
            mov     ah,2            ;mov cursor
            int     10h
            mov     al,[di]         ;fetch char value
            mov     bl,[di+1]       ;  and attribute
            push    cx
            mov     cx,1
            mov     ah,09           ;write char to screen
            int     10h
            pop     cx
            add     di,4            ;DI points to next char
            loop    next_char
            pop     di              ;when  all char  are
            pop     dx              ;  displayed, restore
            pop     cx              ;     registers
            pop     bx
            pop     ax
            ret
display_shape endp
;-----------------------------------------------------------
code_seg    ends
            end     main
```

图 10.5 多字符图形显示程序 DSHAPE

此程序是针对黑白显示器编写的。如果 IBM PC 配置了彩色适配板和彩色显示器,我们也可以设计出彩色的字符图形,其各部分的颜色和背景的颜色通过属性字节设置。

10.4 动画显示的基础

在屏幕上显示出动画的效果并不困难，可以分 5 步进行：

1. 在屏幕上显示图形（单字符或多字符图形）。
2. 延迟一个时间周期，这样图形更清晰。
3. 清除图形
4. 改变图形的行列坐标
5. 返回第 1 步，重复上述过程。

时间的延迟可以通过指令的循环执行来实现。

从屏幕上清除图形可以通过清除部分屏幕来实现，也可以用空字符在原位置重画一次来实现。用空字符(ASCII 码为 0)重新画图，既可用 INT 10H 的 AH=9 来显示空字符，也可用 AH=0AH 的功能以黑底黑字(BL=0)的属性显示字符。

下面我们仍以汽车图形的例子编写程序(MSHAPE)，使“汽车”按水平方向从屏幕上“开”过去。

例 10.7 在屏幕上显示一个开动的汽车。

```
TITLE MSHAPE -- Animate a Multi_character shape
;Move a shape horizontally across the screen,
;       with a 1/4 second delay between moves
;-----------------------------------------------
data_seg  segment
car       db  7                    ;shape table
          db  52h,70h,0,0
          db  0b1h,70h,0,1
          db  0dbh,7,0,1
          db  0dbh,7,0,1
          db  4fh,7,1,0
          db  4fh,7,0, 3
          db  2,7,-2,1
char_cnt  dw  ?
pointer   dw  ?
line_on   db  ?
col_on    db  ?
data_seg  ends
;-----------------------------------------------
code_seg  segment
          assume cs : code_seg,ds : data_seg
main      proc      far
          push      ds                 ;for return
          sub       ax,ax
          push      ax
```

```
                mov     ax,data_seg             ;initialize DS
                mov     ds,ax
                call    clear_screen            ;clear the screen
                lea     di,car                  ;DI points to car
                                                ;   shape table.
                mov     dh,10                   ;body is on line 10
                mov     dl,10                   ;starting in col 10
                call    move_shape              ;move the car
                ret
main            endp
;-------------------------------------------------------------
clear_screen    proc    near
                push    ax                      ;save registers
                push    bx
                push    cx
                push    dx
                mov     ah,6                    ;scroll up func.
                mov     al,0                    ;code to blank screen
                mov     ch,0                    ;upper left row
                mov     cl,0                    ;upper left column
                mov     dh,24                   ;lower right row
                mov     dl,79                   ;lower right column
                mov     bh,7                    ;blank line attribute
                int     10h                     ;video ROM call
                pop     dx                      ;restore registers
                pop     cx
                pop     bx
                pop     ax
                ret                             ;return to main program
clear_screen    endp
;-------------------------------------------------------------
move_shape      proc    near
                push    ax                      ;save registers
                push    bx
                push    cx
                push    dx
                push    di
                mov     ah,0fh                  ;set BH to active page
                int     10h
                sub     ch,ch                   ;clear high byte of count
                mov     cl,[di]                 ;CI holds char count
                inc     di                      ;DI points to first char
                mov     char_cnt,cx             ;save character count
                mov     pointer,di              ;shape table pointer
```

```
                mov     line_on,dh          ;  and coordinates
                mov     col_on,dl
;These instructions plot the shape on the screen
plot_next:
                add     dh,[di+2]           ;update row pointer
                add     dl,[di+3]           ;  and column pointer
                cmp     dl,80               ;Is this char off the screen?
                jb      mov_crsr
                call    erase               ;If so,erase shape and exit
                pop     di
                pop     dx
                pop     cx
                pop     bx
                pop     ax
                ret
mov_crsr:
                mov     ah,2                ;move cursor
                int     10h
                mov     al,[di]             ;fetch char value
                mov     bl,[di+1]           ;and attribute
                push    cx
                mov     cx,1
                mov     ah,09               ;write char to screen
                int     10h
                pop     cx
                add     di,4                ;DI points to next char.
                loop    plot_next           ;
                call    dly_qrtr            ;wait a quarter second
                call    erase               ;then erase the shape
                jmp     short plot_next
move_shape      endp
;-----------------------------------------------------------------
;This procedure erases a shape by reploting it
;   with attribute = 0
erase           proc    near
                mov     cx,char_cnt         ;reload char count
                mov     di,pointer          ;shape table pointer
                mov     dh,line_on          ;line number
                mov     dl,col_on           ;and column number
erase_next:
                add     dh,[di+2]
                add     dl,[di+3]
                mov     ah,2
                int     10h
```

```
                mov     al,[di]             ;to erase a character
                mov     bl,0                ;use an attr = 0
                push    cx
                mov     cx,1
                mov     ah,9
                int     10h
                pop     cx
                add     di,4
                loop    erase_next
                mov     cx,char_cnt         ;reload char count
                mov     di,pointer          ;shape table point
                mov     dh,line_on          ;line number
                inc     col_on              ;point to next column
                mov     dl,col_on
                ret                         ;return to caller
erase           endp
;-------------------------------------------------------------
;This procedure generate a one quarter second delay
dly_qrtr        proc    near
                push    cx
                push    dx
                mov     dx,25
dl1:            mov     cx,2801
dl2:            loop    dl2
                dec     dx
                jnz     dl1
                pop     dx
                pop     cx
                ret
dly_qrtr        endp
;-------------------------------------------------------------
code_seg        ends
                end     main
```

图 10.6 MSHAPE 程序

这个程序是很容易理解的，产生图形的子程序 move_shape 和消除图形的子程序 erase 非常相似，同样是顺序取出字符图形表中的 7 个字符代码依次在屏幕上的显示出来，不同的只是在 erase 子程序中，字符显示的属性是黑底黑字(BL=0)，使人们看不见而已。

10.5 彩色图形

图形方式利用象素来生成彩色图形。

彩色/图形适配器(CGA)提供二种分辨率，中等分辨率可生成 200 行，每行 320 个象素。彩色图形存储器的每个字节表示 4 个象素。高分辨率可生成 200 行，每行 640 个象素。对只有

16K 字节的彩色图形存储器来说，每个象素只能有 1 位来表示，也就是说，存储器的 1 个字节表示 8 个象素。本节我们将主要介绍 320×200 中等分辨率彩色图形方式。表 10.5 是本节将用到的有关图形显示的 BISO 调用。

表 10.5 INT 10H 图形显示操作

AH	功 能	调用参数	返回参数
B	置彩色调板	BH=彩色调板 ID	
	(320×200 图形方式)	BL=和彩色 ID 配套使用的颜色	
C	写象素	DX=行 (0 199)	
		CX=列 (0—639)	
		AL=象素值*	
D	读象素	DX=行 (0—199)	AL=象素值
		CX=列 (0—639)	

* 对显示方式 04H 或 05H(中分辨率 4 色图形)，合法的象素值是 0—3。对显示方式 06H(高分辨率 2 色图形)，合法的象素值是 0 或 1。若 AL 的第 7 位为 1，新的象素值与当前象素值的内容异或(XOR)。

10.5.1 彩色图形存储器映象

中等分辨率彩色图形方式把屏幕分成 320×200=64000 个象素，16KB 的彩色图形存储器的每个字节要表示四个象素，如图 10.7 所示。

表示一个象素的两位有四种组合 00,01,10,11，分别表示四种不同的颜色。00 可选为和背景相同的颜色，背景色可以是 16 种颜色中的任一种。01,10,11 为调色板中的三种颜色。

象素值	调色板 0	调色板 1
00	背景色	背景色
01	绿	青
10	红	品红
11	棕	白

用 INT 10H 的 0BH 功能来选择调色板和背景色。如果选择背景色为黄色，调色板为 0，则屏幕上各象素的颜色可为黄、绿、红、棕。如果一个字节中的象素值为 10101010，则对应的四个象素都会显示红色。如果选择背景为蓝色以及调色板为 1，则有效的四种颜色为蓝、青、品红和白。如果一个字节中的象素值为 00110011，则对应的四个象素分别显示蓝、白、蓝、白，其显示效果象是一条白色的点线。

在彩色方式中，存储器和象素之间的对应关系和单色显示方式有些差别。在单色显示方式中，每个字符位置有两个存储器字节和它对应。而彩色方式中，每个象素只有 1/4 字节(两位)与其对应。除此而外，屏幕上的扫描行的对应关系也不同。在 IBM PC 中，彩色存储器被分成两部分：B800:0000 到 B800:1F3F 对应于偶数扫描行，另一半 B800:2000 到 B800:3F3F 对应于奇数扫描行。采取这样的排列是为了便于硬件线路向屏幕上绘图。其形式如图 10.8。

由此可见，要把一个区域变成整块的颜色，需要操作两次，一次向对应于偶数行的单元填入象

素值，另一次向对应于奇数行的单元填入象素值。如果用 DEBUG 命令来操作，应有以下几步：

```
A>debug
-r ds
DS 08F1
: b800 ←把数据段置成彩色存储器
-f 0 3f3f 0 ←清除整个屏幕
-f 0 400 55 ←使偶数行的 400h×4 个象素的值为 01(绿色)
-f 2000 2400 55 ←使奇数行的 400h×4 个象素值为 01(绿色)
```

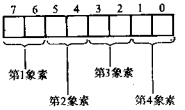


图 10.7　彩色图形存储器的一个字节

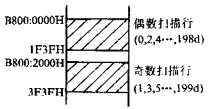


图 10.8　彩色存储器的两半部分

图 10.9 表明了彩色屏幕与显示存储器的关系。

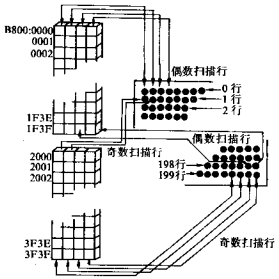


图 10.9　彩色存储器与显示屏幕

10.5.2　ROM BIOS 读/写象素例行程序

如果编写直接访问显示存储器，向对应的单元读/写象素值的程序，必须把象素所在的行号和列号转换成显示存储器中的存储地址。对应于奇数扫描行和偶数扫描行必须分别处理两个存储单元组。另外，由于每个字节中包括四个象素，因此程序不仅要断定读写单元的地址。还要断定这个字节中的四组两位象素值应读/写哪一组，这就导致了一些复杂的计算。ROM BIOS 读/写象素的例行程序实际上就要完成上述的转换过程。为了了解调用 BIOS 功能时，例行程序所做的工作，也为了给某些技术人员设计绘图的常用程序提供经验，我们以写象素(write dot)为例，把象素处理过程编写一个可调用的子程序，其输入参数由调用程序给出，并把附加段定义为彩色显示存储器，其段地址为 B800h。

例 10.8　在彩色屏幕上写象素的例行程序。

```
; SUBROUTINE TO PLOT A POINT ON SCREEN
; Medium res graphics mode 320 x 200 color
; Enter with:
; X — coordinate in CX (column number;0-319)
```

```
; Y -- coordinate in DX (row number; 0-199)
; color in AL (0=background, 1,2,3=colors)
; video address (B800h) is in ES
;-----------------------------------------------
plotsub     proc    far
            push    bx              ;save BX
            push    cx              ;save column
            push    dx              ;save row
            push    ax              ;save color
;multiply the row number by # of bytes per row 80
;   but since already mult by 2, use 40
            push    dx              ;save row for odd/even
            mov     al,40           ;bytes/row div by 2
            add     dx,0fch         ;mask off odd/even bit
            mul     dl              ;AX now is row address
;If odd row number, should add 2000h
;   for 2nd memory bank
            pop     dx              ;get original row #
            test    dl,1            ;test odd/even bit
            jz      not_odd         ;jump on even row
            add     ax,2000h        ;add to get 2nd bank
not_odd:
            mov     bx,ax           ;save row addr in BX
;add column address to row address
            push    cx              ;save column address
            shr     cx,1            ;shift it right to
            shr     cx,1            ;kill bit_pos bits
            add     bx,cx           ;add it to addr in BX
;use bit_pos bits to put color and mask
;   in the right position
            pop     cx              ;get original col #
            and     cx,3            ;save bit_pos bits
            inc     cx              ;get one free shift
            pop     ax              ;get color
            mov     dl,0fch         ;DL=mask : 11 11 11 00
shift:      ror     al,1            ;AL=color : 00 00 00 cc
            ror     al,1            ;shift color 2 bits right
            ror     dl,1            ;
            ror     dl,1            ;shift mask 2 bits right
            loop    shift           ;do it bit_pos times
;get contents of byte mask off all but color bits
;   or on color bits
            and     es:[bx],dl      ;mask off and
            or      es:[bx],al      ;or on color value
```

```
        pop     dx
        pop     cx
        pop     bx
        ret
plotsub endp
;-------------------------------
        end
```

图 10.10 PLOTSUB 例行程序

PLOTSUB 子程序分别利用 CX,DX 和 AL 寄存器保存列号、行号和象素值,其各位分配如图 10.11 所示。

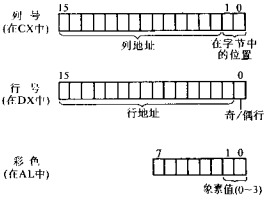


图 10.11 PLOTSUB 程序的参数寄存器

PLOTSUB 子程序所做的第一项工作是把 DX 中的屏幕行号转换成存储器地址,它是该行的起始地址,为此需把行号乘以每行的字节数 80d,(因为每行有 320 个象素,而每个字节表示 4 个象素,所以 320/4=80),但行号字节的最低位用来指明是奇数行还是偶数行,所以相当于行号左移了一位,即相当于已经乘以 2,故再乘以 40d 就得到了该行行首的偏移地址。

下一步检查是奇数行还是偶数行,如果是奇数行,则必须在该地址上加上 2000h,因为奇数行的象素处于第二组存储器,其开始地址为 2000h。

列地址(CX)的最低两位指出象素对应在字节中的位组,因此先右移两次暂时去掉这两位,然后把列地址加到已计算好的行地址(BX)中,现在 BX 的内容就是要操作的象素所在的字节地址。

最后根据 CX(列号)最低两位指出的象素在字节中的位置。将 AL 中的象素值移到对应的位组上,利用 AND 操作去掉 BX 所指示的存储单元中选定的位组(两位),并用 OR 操作给这两位加上象素值。瞬间,一个圆点就出现(或消失)在彩色屏幕上。

10.5.3 彩色绘图程序

在 ROM BIOS 显示 I/O 部分提供了读/写象素等显示功能,类似上节讲到的复杂计算由 BIOS 例行程序完成,用户只要用 INT 10H 来调用各种功能就行了,这使得在屏幕上绘图变得相当简单。

首先使用 INT 10H 的 AH=00 功能将显示方式置为图形方式:

```
MOV   AH,00          ;set graphics mode
MOV   AL,04          ;for CGA
INT   10H
```

第二步用 INT 10H 的 AH=0BH 功能置背景颜色和彩色调板:

```
MOV   AH,0BH         ;set color palette
MOV   BH,00          ;background
```

```
MOV   BL,cc          ;cc=0—F,one of 16 colors
INT   10H            ;call BIOS

MOV   AH,0BH         ;set color palette
MOV   BH,01          ;select palette
MOV   BL,00          ;palette  #0
INT   10H            ;call BIOS
```

如果想保持相同的调色板，只需设置一次，一旦要改变调色板，整个屏幕的颜色就会变成另外一套。如果在文本方式下用 0BH 功能，调色板的值只决定边界的颜色。

这些工作做完后，就可在指定的坐标位置上写象素点或读象素点：

```
MOV   AH,0CH         ; write pixel dot
MOV   AL,color       ; color of pixel (0—3)
MOV   CX,column      ; horizontal position (0—319)
MOV   DX,row         ; Vertical position (0—199)
INT   10H            ; call BIOS

MOV   AH,0DH         ; read pixel dot
MOV   CX,column      ; horizontal position (0—319)
MOV   DX,row         ; vertical position (0—199)
INT   10H            ; return the pixel color to AL
```

例 10.9　设置图形方式并显示彩条

选择背景色为篮色，调色板为 0，然后每行显示一种颜色，每 4 行重复一次，一直到整个屏幕都显示出彩条。

```
TITLE   GRAPHIX.COM
codeseg  segment
         assume cs : codeseg,ds : codeseg, ss : codeseg
         org     100h
main     proc    far
         mov     ah,0                ;set graphics mode
         mov     al,04               ;320 x 200 color
         int     10h

         mov     ah,0bh              ;set color palette
         mov     bh,0                ;background
         mov     bl,1                ;blue
         int     10h                 ;call BIOS

         mov     ah,0bh              ;select palette
         mov     bh,01
         mov     bl,0                ;palette 0#
         int     10h

         mov     bx,0                ;set initial color
         mov     cx,0                ;   column and
         mov     dx,0                ;      row
line:
```

```
        mov     ah,0ch                  ;write dot
        mov     al,bl                   ;set color
        int     10h
        inc     cx                      ;increment column
        cmp     cx,320                  ;column at 320 ?
        jne     line                    ;no, loop
        mov     cx,0                    ;yes, reset column
        inc     bl                      ;change color
        and     bl,03
        inc     dx                      ;increment row
        cmp     dx,200                  ;row at 200 ?
        jne     line                    ;no, loop
        ret                             ;yes, terminate
main    endp
codeseg ends
        end     main
```

图 10.12 GRAPHIX 程序

GRAPHIX 程序从 0 行 0 列开始画彩线，DX、CX 寄存器在此用来保留行号和列号，在画每条水平彩线时，DX(行号)保持不变，CX(列号)从 0 增加到 319。画下一条彩线时，DX(行号)加 1，CX(列号)又从 0 开始，而且每划一条水平彩线 BL 中的象素值增 1。第一条彩线是蓝色(背景色 00)，第二条彩线是绿色(01)，第三条是红色(02)，第四条是棕色(03)，以后反复显示这四种颜色。不难想象如果是要求画垂线时，应先保持列号(CX)不变，而行号(CX)从 0 开始每次增加一个象素直至最后一个象素 199。此程序使用了写象素点的功能 0CH，这使得编写绘图程序的工作简便多了。

例 10.10 按动光标控制键，在屏幕的上下左右任一方向上绘图。每画一点之前，由数字键 0—3 指定该点的彩色值。按动 ESC 键，绘图结束，返回 DOS。

```
;DRAW--Program to draw on screen with sursor arrows
; For 320 * 200 medium res color mode
up      equ     48h             ;scan code for up arrow
down    equ     50h             ;scan code for down arrow
right   equ     4dh             ;scan code for right arrow
left    equ     4bh             ;scan code for left arrow
escape  equ     1bh             ;"Esc" character
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
prognam         segment                 ;define code segment
;-----------------------------------------------------------
main    proc    far
        assume  cs : prognam
start:  push    ds                      ;set up stack for return
        sub     ax,ax
        push    ax
;clear screen by scrolling it,using ROM call
```

```
                mov     ah,6            ;scroll up function
                mov     al,0            ;code to blank screen
                mov     cx,0            ;upper left = 0,0
                mov     dl,79           ;lower right corner
                mov     dh,24           ; at 79,24
                int     10h             ;call video interrupt
;screen pointer will be in CX,DX registers
;row number ( 0 to 320d ) in DX
;column number ( 0 to 320d ) in CX
                mov     ah,0            ;set grapgics mode
                mov     al,04           ;320 * 200 color
                int     10h
                mov     ah,0bh          ;set color patette
                mov     bh,0            ;background
                mov     bl,0            ;black
                int     10h
                mov     ah,0bh          ;select palette
                mov     bh,01
                mov     bl,0            ;palette 0
                int     10h
                mov     dx,100d         ;set screen pointer to
                mov     cx,160d         ;  center of screen
;get character from keyboard
get_char:
                mov     ah,0            ;code for read char
                int     16h             ;keyboard I/O ROM call
                cmp     al,escape       ;is it Esc char ?
                jz      exit            ;yes
                cmp     al,33h          ;is it more than "3" ?
                jg      plot            ;yes,not a color
                cmp     al,30h          ;is it less than "0" ?
                jl      plot            ;yes,not a color
                mov     bl,al           ;save it in BL
                and     bl,03
                jmp     get_char        ;get next character
;figure out which way to go,and draw new line
plot:           mov     al,ah           ;put scan code in AL
                cmp     al,up           ;is it up arrow ?
                jnz     not_up          ;no
                dec     dx              ;yes,decrement row
not_up:         cmp     al,down         ;is it down arrow ?
                jnz     not_down        ;no
                inc     dx              ;yes,increment row
not_down:
```

```
                cmp     al,right        ;isit right arrow ?
                jnz     not_right       ;no
                inc     cx              ;yes,increment column
not_right:
                cmp     al,left         ;is it left arrow ?
                jnz     write           ;no
                dec     cx              ;yes,decrement column
;use ROM routine to write dot
;reguires row # in DX,col in CX,color in AL
write:          mov     al,bl           ;set color value
                mov     ah,0ch          ;write dot function
                int     10h             ;video BIOS routine
                jmp     get_char        ;get next color and arrow
exit:           ret                     ;return to DOS
main            endp
;-------------------------------------------------------------
prognam         ends
                end     start
```

图 10.13 绘图程序 DRAW

这个程序也很容易理解，它只使用了标绘象素点的 BIOS 功能，写象素点所需要的调用参数行变量和列变量分别保留在 DX 和 CX 寄存器中，一开始先把 DX 和 CX 置在屏幕的中央，当按动一个光标控制键时，寄存器 DX 或 CX 中的值会改变。当按动↑键时，DX 的值减 1，当按动↓键时，DX 的值增加 1，当按动←时，CX 减 1、当按动→键时，CX 的值加 1，这样由 CX 和 DX 的值决定了写点的坐标位置。写点的彩色值(象素值)由数字键 0—3 决定，其 0—3 的值保留在 BL 中，在调用写点功能时，再传送给 AL。在光标控制键的指引下，画一个汽车图形，我们会发现以这种方式画出的汽车图形比前面用字符图形组合成的汽车图形精致得多了。

这个程序运行后，仍返回在 320×200 的彩色图形方式下，如果要求退出后恢复原来的 80×25 文本方式，可在 ret 指令前再设置成原来的显示方式：

```
MOV   AH,00     ;设置显示方式
MOV   AL,02     ;80×25 黑白文本方式
INT   10H       ;调用 BIOS
```

习　题

10.1　在文本方式下，什么属性值产生以下各种显示特性？

1) 黑底白字

2) 黑底白字，下划线

3) 白底黑字

4) 蓝底红字

5) 蓝底红字、闪烁

10.2　写出指令序列：

1) 设置 80 列黑白方式

2) 把光标设置在第 5 行的开始

3) 上卷 10 行

4) 显示 10 个闪烁的 * 号。

10.3 编写指令，以文本方式在品红底显示 5 个浅绿色的笑脸符。

10.4 从键盘敲入一串字符，当输入字符是回车符时输入结束，并把光标置在 24 行 0 列。

10.5 写出指令：以图形方式选择背景为绿色。

10.6 写出指令：以图形方式从 12 行 13 列读出一个点。

10.7 编写程序使一只"鸟"飞过屏幕。飞鸟的动作可由小写字母 v(ASCII 码 76H)变为破折号(ASCII 码 0C4H)来模仿，这个字符先后交替在两列显示。鸟的开始位置是 0 列 20 行，每个字符显示 0.5 秒，然后消失。

10.8 游戏程序常常用随机数来控制其图形在屏幕上移动。请编写一程序，用随机数来控制笑脸符(ASCII 码 02)显示的位置。笑脸符每次显示的列号总是递增 1。而行的位置可能是前次的上一行，下一行或同一行，这根据随机数是 0，1 或 2 来决定，当行号变为 0、24 或列号变为 79 时显示结束。笑脸在每个位置上显示 1/4 秒。(提示：INT 1AH 的 AH=0 是读当前时间的功能调用，利用该功能返回的随时都在变化的时间值作为产生随机数的基数)。

第十一章　发声系统的程序设计

在前面两章，我们介绍了屏幕控制、键盘输入，打印机输出等 DOS 和 BIOS 的中断处理功能。但 DOS 和 BIOS 没有提供通过扬声器产生声音的工具，因此在本章我们将讨论如何编写利用 PC 发声系统产生各种声音和乐曲的汇编语言程序。为什么要计算机发出声音和乐曲呢？如果我们编写一个游戏程序，音乐和声音的效果会使游戏更加有趣。在演示表演程序时，音响会使表演更加丰富多彩。在大多数程序中，声音用来表示出错信息，或表示一个过程结束，或提醒用户输入信息，这时声音成为一种很有效的人-机通讯方式。

11.1　扬声器驱动系统

PC 机上的大多数输入/输出(I/O)都是由系统插件板上的 8255(或 8255A)可编程序外围接口芯片(PPI)管理的。PPI 包括三个 8 位寄存器，两个用于输入功能，一个用于输出功能。输入寄存器分配的 I/O 端口号为 60H 和 62H，输出寄存器分配的 I/O 端口号为 61H。由 PPI 的输出寄存器中的两位来选择扬声器的驱动方式(见图 11.1 扬声器驱动系统)。当输出寄存器(I/O 端口 61H)的第 0 位为 1 时，控制 8254 定时器来驱动扬声器，当第 1 位为 1 时，扬声器的门电路接通，并一直保持到位 1 变为 0 时关闭。

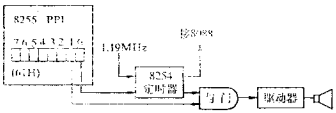


图 11.1　扬声器驱动系统

在第八章曾介绍过一个 Sound 程序，它是直接控制输出端口 61H 的第 1 位为 1 或为 0，使扬声器交替地接通与断开，从而推动扬声器的纸盆振动，发出不同音高和音长的声音。

另一种产生声音的方法是利用 8254 定时器来驱动扬声器。PC 机有三种不同的定时器。0 号定时器用来作为通用的日时钟，它每秒约发出 18 次中断以修正时间，1 号定时器用于 DMA(直接存储器存取)传送数据，连接到扬声器上的是 2 号定时器，它实际上是一个振荡电路，此电路向扬声器发送指定频率的脉冲，当输出端口 61H 控制扬声器为接通状态时，就发出了一定频率的声音。本章主要介绍这种利用定时器产生声音的技术。

11.2　通用发声程序

*ROM BIOS 中有个 BEEP 子程序，它能根据 BL 中给出的时间计数值控制 8254 定时器，产*生持续时间为 1 个或几个 0.5 秒，频率为 896Hz 的声音，这个子程序在 IBM PC 技术手册的 TEST4 部分。它的用途是当“加电自测”系统发现硬件错误后，由 ERROR_BEEP 处理程序调用 BEEP 子程序，使扬声器发出“嘟嘟”的信号。BL 中的时间计数值由 ERROR_BEEP 程序设置为

533H，BEEP 子程序将此计数值送给 8254 的定时器 2 来产生 896Hz 的声音，然后 BEEP 又将 AL 的第 0 位和第 1 位置 1，并把 AL 的值送到 8255 的输出寄存器(I/O 端口 61H)，使扬声器接通发出声音。

实际上 BEEP 是一个很好的通用发声程序，我们可以利用并修改 BEEP，使其产生任一频率的声音。为此我们需要做两点修改，首先，BEEP 程序只能产生 896Hz 的声音，我们的通用发声程序应能产生任一频率的声音。其次，BEEP 产生声音的持续时间(音长)只能是 0.5 秒的倍数，我们希望声音的持续时间更易于调整，例如可以是 10ms 的倍数。

我们知道 BEEP 是将计数值 533H 送给定时器 2 产生 896Hz 的声音的，那么产生其它频率声音的时间计数值应为：

533H × 896 ÷ 给定频率 = 123280H ÷ 给定频率

假定发声的频率存放在 DI 寄存器中，下面的指令在 AX 中得到送往定时器 2 的计数值：

```
MOV     DX,12H
MOV     AX,533H * 896
DIV     DI
```

10ms 的延迟时间可以简单地通过执行循环指令取得：

```
WAIT :  MOV     CX,2801
DELAY : LOOP    DELAY
```

如果要产生与 10ms 成倍数的延迟时间，可在 BX 寄存器中放入倍数。例如要产生 1 秒的持续时间，则在 BX 中放入 100，以控制 LOOP 指令执行 100×2801 次，也就是 10ms 的 100 倍。指令如下：

```
        MOV     BX,100
WAIT :  MOV     CX,2801
DELAY : LOOP    DELAY
        DEC     BX
        JNZ     WAIT
```

这样我们就能在修改后的 BEEP 程序的基础上编写一个任一频率(由 DI 指定)和任一持续时间(由 CX 和 BX 指定)的通用发声程序。此程序利用定时器产生声音比前面的 Sound 程序稍微复杂一些，它包括三个步骤：

1. 在 2 号定时器中的 43 端口送一个特定的数 0B6H(10110110B)，此数对定时器的方式寄存器进行初始化，使定时器 2 准备接收计时常数。

2. 在 2 号定时器中的 42H 端口装入一个 16 位的计时常数(533H×896/频率)，以建立将要产生的声音频率。

3. 把输出端口 61H 的 0、1 两位置 1，发出声音。

例 11.1 编写通用发声程序 GENSOUND，它能利用定时器发出指定频率的声音。

```
TITLE GENSOUND -- The speaker beeper
;Produces a tone of a specified frequency
;  and duration public gensound
public  gensound
cseg  segment  para  'code'
              assume cs : cseg
gensound      proc        far
```

```
                push    ax                      ;save registers
                push    bx
                push    cx
                push    dx
                push    di
                mov     al,0b6h                 ;write timer mode reg.
                out     43h,al
                mov     dx,12h                  ;timer divisor
                mov     ax,533h * 896           ; 533h * 896 / freq
                div     di
                out     42h,al                  ;write timer2 count low byte
                mov     al,ah
                out     42h,al                  ;write timer2 count hight byte
                in      al,61h                  ;get current port setting
                mov     ah,al                   ;and save it in AH
                or      al,3                    ;turn speaker on
                out     61h,al
wait1:          mov     cx,280l                 ;wait for specified interval
delay:          loop    delay
                dec     bx
                jnz     wait1
                mov     al,ah                   ;recover value of port
                out     61h,al
                pop     di                      ;restore the registers
                pop     dx
                pop     cx
                pop     bx
                pop     ax
                ret                             ;exit
gensound        endp
cseg            ends
                end
```

图 11.2 GENSOUND 程序

GENSOUND 程序能产生 19Hz 到 65535Hz 的声音，这个频率的下限 19Hz 是使除法不产生溢出的最小的 DI 值((DX)=12H=18d<19)。其上限 65535Hz 实际上是多余的，因为人们最高能听到的音频约为 20000Hz。这个程序在一开始用 public 伪操作定义成一个公用子程序。在后面的几节中，凡需要 PC 机发出指定频率(在 DI 中设定频率值)和指定延迟时间(在 BX 设定 10ms 的倍数)的声音时，可直接调用此通用发声程序。

11.3 乐曲程序

利用通用发声程序，可以编写演奏乐曲的程序。乐曲是按照一定的高低、长短和强弱关系组成的音调，在一首乐曲中，每个音符的音高和音长与频率和节拍有关。图 11.3 画出了两个音

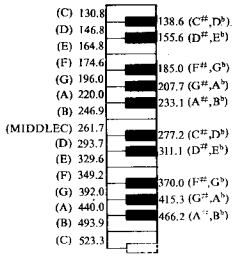


图 11.3 两个音阶的琴键

阶(一个音阶是 8 个音符)的钢琴键和每个键的音名及其频率(Hz)。低音阶从低 C(130.8Hz)到中 C(261.7Hz),高音阶从中 C 到高 C(523.3Hz)。白色键(A 到 G)演奏本位音符,黑色键演奏升降音符,黑键比它旁边的白键高半个音或低半个音。

组成乐曲的每个音符的频率值和持续时间是乐曲程序发声所需要的两个数据。音符的频率可以从图 11.3 中获得,音符的持续时间根据乐曲的速度及每个音符的节拍数来确定。在 4/4(四四拍)中,四分音符为一拍,每小节 4 拍,全音符持续 4 拍,二分音符持续 2 拍,四分音符持续 1 拍,八分音符持续半拍等。如果我们给全音符分配 1 秒的时间,则二分音符的持续时间为 0.5 秒(50×10ms),四分音符的持续时间为 0.25 秒(25×10ms),八分音符的持续时间为 0.125 秒(12.5×10ms)。

知道了音调与频率和时间的关系,我们就可以按照乐曲的曲谱将每个音符的频率和持续时间定义成两个数据表,然后编写程序依次取出表中的频率值和时间值,调用 GENSOUND 程序发出各种声音。

下面我们按照图 11.4 的乐谱编写一个演奏程序。

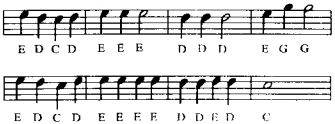


图 11.4 乐谱

编写乐曲程序可分为四个步骤:

1. 为演奏的乐曲定义一个频率表和一个节拍时间表。

```
MUS_FREQ    DW 330,294,262,294,3 DUP (330)      ;1,2 小节
            DW 294,294,294,330,392,392          ;3,4 小节
            DW 330,294,262,294,4 DUP(330)       ;5,6 小节
            DW 294,294,330,294,262,0FFFFH       ;7,8 小节

MUS_TIME    DW 6  DUP (25),50                   ;1,2 小节
            DW 2  DUP (25,25,50)                ;3,4 小节
            DW 12 DUP(25),100                   ;5—8 小节
```

2. 分别将两个表的偏移地址放入 SI 和 BP

```
        LEA   SI,MUS_FREQ
        LEA   BP,DS:MUS_TIME
```

3. 从表中取出音符的频率放入 DI，取出音符的持续时间(实际上是 10ms 的倍数)放入 BX。

```
        MOV   DI,[SI]
        MOV   BX,DS:[BP]
```

频率表中的最后一个数据 0FFFFH 作为乐曲的结束符，也可用 0 或其它特定值来代替。

4. 调用 GENSOUND 子程序发出音调。

例 11.2　演奏乐曲的程序 MUSIC

```
TITLE MUSIC -- A music of 'Mary had a little lamb'
;assemble with:masm music
;link with:link music+gensound
extrn gensound:far
stack segment para stack 'stack'
      db 64 dup ('stack...')
stack ends
;-----------------------------------------------
dseg segment para 'data'
mus_freq  dw  330,294,262,294,3 dup(330)   ;bars 1 & 2
          dw  3 dup(294),330,392,392       ;bars 3 & 4
          dw  330,294,262,294,4 dup(330)   ;bars 5 & 6
          dw  294,294,330,294,262,-1       ;bars 7 & 8
mus_time  dw  6 dup(25),50                 ;bars 1 & 2
          dw  2 dup(25,25,50)              ;bars 3 & 4
          dw  12 dup(25),100               ;bars 5 & 8
dseg  ends
;-----------------------------------------------
cseg segment para 'code'
        assume cs:cseg, ss:stack, ds:dseg
music   proc    far
        push    ds                  ;put return addr.
        sub     ax,ax               ;   on stack
        push    ax
        mov     ax,dseg             ;initialize DS
        mov     ds,ax
        lea     si,mus_freq         ;puts the freq table
                                    ; offset in SI
        lea     bp,ds:mus_time      ;puts the time table
                                    ; offset in BP
freq:   mov     di,[si]             ;read next frequency
        cmp     di,-1               ;end of tone ?
        je      end_mus             ;if so,exit
        mov     bx,ds:[bp]          ;otherwise,fetch the
```

```
                                        ;    duration
        call    gensound                ;play the note
        add     si,2                    ;update the table pointer
        add     bp,2
        jmp     freq                    ;go process next note
end_mus:
        ret                             ;return to DOS
music   endp
cseg    ends
;-------------------------------------------------------
        end   music
```

图 11.5　MUSIC 程序

这个程序是比较简单的，如果想演奏另一个乐曲，只需把 mus_freq 和 mus_time 两个表中的数据换成另一个乐曲的频率和节拍时间就可以了。例如我们把《太湖船》的曲调写入数据段，则此程序运行的结果是演奏了一首民乐《太湖船》。

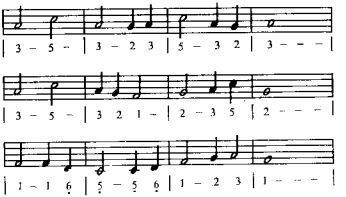


图 11.6　《太湖船》乐谱

```
DSEG    SEGMENT   PARA 'DATA'
MUS_FREQ  DW   330,392,330,294,330,392,330,294,330
          DW   330,392,330,294,262,294,330,392,294
          DW   262,262,220,196,196,220,262,294,330,262
          DW   -1
MUS_TIME  DW   3 DUP (50),25,25,50,25,25,100
          DW   2 DUP (50,50,25,25),100
          DW   3 DUP (50,25,25),100
DSEG      ENDS
```

11.4　键盘控制发声程序

音符和频率之间有一定的对应关系，如果计算机键盘上的某些键（例如数字键或字母键）和音符、频率也形成一种对应关系，则可以通过键盘控制扬声器发出各种音符的声音，这时计

算机键盘就变成了钢琴键盘，我们可以用它弹奏出简单的音乐。钢琴有 88 个音符，为了简单起见，我们先通过编写一个八度音程的钢琴程序，以了解键盘控制发声的原理，然后再设计一个如图 11.3 提供的两个音阶的本位音符以及升降音符(共 42 个音符)的钢琴程序。为了弹奏方便，我们让数字键 1～8 对应一个音阶的八个音符。

数字键	音符	频率(Hz)
1	C	262
2	D	294
3	E	330
4	F	347
5	G	392
6	A	440
7	B	494
8	C	523

弹奏时，对应乐谱上的 C 音符(简谱中为 1)，我们按下数字键"1"(ASCII 码为 31H)，程序的工作就是要接收这个键并将频率表中和它对应的频率值 262Hz 送给 GENSOUND 程序以发出 C 的音调。按下数字键"2"(ASCII 码为 32H)，程序就将 294Hz 的频率值送给 GENSOUND 程序从而发出 D 的音调。按下数字键"8"，将发出比第一个 C 高八度的音调，音符的频率表作为数据定义在数据段中。图 11.7 是一个八度的钢琴程序 PIANO 的框图。

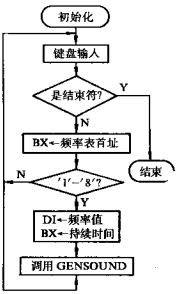


图 11.7 PIANO 程序框图

```
TITLE PIANO -- Use timer 2 to run speaker
;Keyboard number keys play notes of the scale
;assemble with : masm piano
```

```
;link with : link piano+gensound
;
        extrn gensound : far
stack segment para stack 'stack'
        db 64 dup('stack...')
stack ends
;-----------------------------------------------
dseg segment para 'data'
table   dw      262             ; C
        dw      294             ; D
        dw      330             ; E
        dw      349             ; F
        dw      392             ; G
        dw      440             ; A
        dw      494             ; B
        dw      523             ; C
dseg ends
;-----------------------------------------------
cseg segment para 'code'
main proc far
    assume cs : cseg,ds : dseg,ss : stack
        push    ds              ;set up stack for return
        sub     ax,ax
        push    ax
        mov     ax,dseg         ;set DS to current
        mov     ds,ax           ; data segment
new_note:
        mov     ah,0            ;read the keyboard
        int     16h
        cmp     al,0dh          ;is it return?
        je      exit            ;is so, exit
        mov     bx,offset table ;freq table address
        cmp     al,'1'          ;is'1'--'8'?
        jb      new_note        ;no, read the next key
        cmp     al,'8'
        ja      new_note
        and     al,0fh          ;mask off high 4 bits
        shl     ax,1            ; * by 2
        sub     ax,2            ;convert the ASCII to
        mov     si,ax           ; an index in SI
        mov     di,[bx][si]     ;look_up a frequency
        mov     bx,10           ;for 0.1 second
        call    gensound        ;play the note
        jmp     new_note        ;go get another note
```

```
exit:   ret                     ;return to DOS
main    endp
cseg    ends
;-------------------------------------------------
        end     main
```

图 11.8 PIANO 程序

从键盘输入一个键这里用的是 BIOS 16H 中断，AH 中回送的是键的 ASCII 码。AH 中回送键的扫描码。因为此程序没有用到功能键或控制键，所以完全可以用 DOS 的键盘输入功能。

```
MOV   AH,07             ;DOS Kbd function,no echo
INT   21H               ;call DOS
```

下面我们进一步设计一个音域较宽的电子钢琴程序 ELEPIANO。我们先考虑键和音符的对应关系。图 11.3 中的两个音阶，包括升降音共有 42 个音符，本位音，升音和降音都分为高音阶和低音阶两部分，所以在计算机键盘上按下一个数字键发出低音阶音符，按下 shift—数字键产生高音符音阶。为了产生升降音符，在按下数字键之前，再按下 b(降号)和 a(用 a 来代替升号#)。表 11.1 表明了按键和音符之间的关系。

表 11.1 按键和音符、音频之间的关系

音　符	按　键	频　率(Hz)
lower naturals : C — B	'1 — 7'	131,147,165,175,196,220,247
uppre naturals : C — B	shift '1 — 7'	262,294,330,349,392,440,494
lower sharps : C# — B#	'a' '1 — 7'	139,156,175,185,208,233,262
upper sharps : C# — B#	shift 'a' '1 — 7'	277,311,349,370,415,466,523
lower flats : Cb — Bb	'b' '1 — 7'	123,139,156,165,185,208,233
upper flats : Cb — Bb	shift 'b' '1 — 7'	247,277,311,330,370,415,466

图 11.001

乐谱上的升降号总是跟在音符之后，如 A#，D$ 等，但程序为了判断 '1—7' 的音符键是升音还是降音，应在按下数字键之前按下升号键 'a' 或降号键 'b'，确定音阶后再同时按下 shift 键。实际上按下 shift_a，机器接收的是 'A' 的 ASCII 码，按下 shift_b，机器接收的是 'B' 的 ASCII 码，较为复杂的是 shift_数字键(对应 upper_octave natural notes)，shift_1 为 '!' 字符，ASCII 码为 21H，shift_2 为 '*' 字符，ASCII 码为 2AH 等。它们的 ASCII 码没有规律性，所以判断 shift_'1—7' 的程序要稍微复杂一些。我们先把它们的对应关系列出来：

按　键	字符	ASCII 码
shift_1	!	21
shift_2	@	40
shift_3	#	23
shift_4	$	24
shift_5	%	25
shift_6	∧	5E
shift_7	&	26

和 PIANO 程序一样，42 个音符的频率值定义在数据段中：

```
l_nat   dw   131,147,165,175,196,220,247
u_nat   dw   262,294,330,349,392,440,494
l_sha   dw   139,156,175,185,208,233,262
u_sha   dw   277,311,349,370,415,466,523
l_fla   dw   123,139,156,165,185,208,233
u_fla   dw   247,277,311,330,370,425,466
```

程序首先要根据按键 a，b 或 shift_a，shift_b，或 shift 来选择频率表中的六组音频，然后再根据输入的'1—7'数字键来找出 7 个音频值中的一个，送 GENSOUND 程序发出相应的音调。图 11.9 是 ELEPIANO 程序的框图。

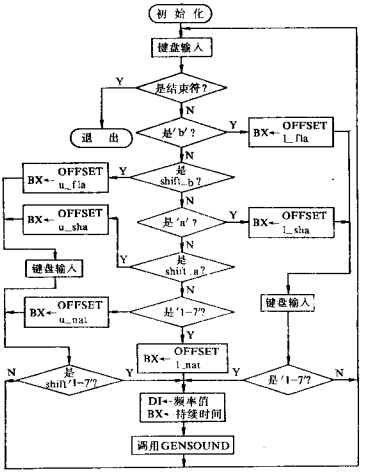


图 11.9 ELEPIANO 程序框图

```
TITLE ELEPIANO  - Electric piano
;This procedure makes the computer keyboard function
;Like a two_octave piano keyboad
```

```
;To play a lower_octave natural note,
;   press a number key (1-7)
;To play an upper_octave natural note,
;   hold shift down and press a number key (1_7)
;To play a sharp or flat,press the a or b key
;   and then the number key(1-7)
;To exit,press return
;-----------------------------------------------
        extrn gensound : far
stack segment para stack 'stack'
        db 64 dup('stack...')
stack ends
;-----------------------------------------------
dseg segment para 'data'
l_nat   dw  131,147,165,175,196,220,247
u_nat   dw  262,294,330,349,392,440,494
l_sha   dw  139,156,175,185,208,233,262
u_sha   dw  277,311,349,370,415,466,523
l_fla   dw  123,139,156,165,185,208,233
u_fla   dw  247,277,311,330,370,415,466
dseg ends
;-----------------------------------------------
cseg segment para 'code'
play proc far
        assume cs : cseg, ds : dseg, ss : stack
            push    ds                  ;cor return to DOS
            sub     ax,ax
            push    ax
            mov     ax,dseg             ;initialize the data
            mov     ds,ax               ; segment addr.
new_note:
            mov     ah,0                ;read the next key
            int     16h
            cmp     al,0dh              ;is it return?
            jne     lwr_flt
            ret                         ;exit
lwr_flt:    sub     si,si
            cmp     al,'b'
            jne     upr_flt
            lea     bx,l_fla            ;possible lower octave flat
get_lc:     mov     ah,0
            int     16h
            jmp     short chk_lo
```

```
upr_flt:  cmp     al,'B'
          jne     lwr_shp
          lea     bx,u_fla          ;possible upper octave flat
get_uc:   mov     al,0
          int     16h
          jmp     short chk_uo
lwr_shp:  cmp     al,'a'
          jne     upr_shp
          lea     bx,l_sha          ;possible lower octave sharp
          jmp     short get_lc
upr_shp:  cmp     al,'A'
          jne     lwr_nat
          lea     bx,u_sha          ;possible upper octave sharp
          jmp     short get_uc
lwr_nat:  cmp     al,'1'
          jb      upr_nat
          cmp     al,'7'
          ja      upr_nat
          lea     bx,l_nat
          jmp     short new_frq
upr_nat:  lea     bx,u_nat
          jmp     short chk_uo
;check whether the key selects lower octave(1-7)
chk_lo:   cmp     al,'1'
          jb      new_note
          cmp     al,'7'
          ja      new_note
          jna     new_frq
;check whether the key selects upper
;   octave (shift_'1'-'7')
chk_uo:   cmp     al,'!'
          jb      new_note
          cmp     al,'%'
          jna     new_frq
          cmp     al,'&'
          jne     again
          add     al,1              ;convert "&" to index 7
          jmp     new_frq
again:    cmp     al,'@'
          jne     next
          add     al,2              ;convert "@" to index 2
          jmp     new_frq
next:     cmp     al,'~'
```

```
        jne     new_note
;look_up a freq and call gensound to
;  play the note for 0.1 second
new_frq: and    ax,07h
        shl     ax,1
        sub     ax,2
        mov     si,ax           ;an index in SI
        mov     di,[bx][si]     ;get a freq to DI
        mov     bx,10           ;for 0.1 second
        call    gensound        ;play the note
        jmp     new_note        ;next key code
play    endp
cseg    ends
;-----------------------------------------------
        end     play
```

图 11.10 ELEPIANO 程序

11.5 报警程序

IBM PC 机的发声功能和定时功能相结合，还可以使 PC 机在规定的时间发出“警报”，好象是一只“闹钟”，在设定的时间发出铃声，以引起人们的注意。但是计算机的报警时钟比普通的闹钟用途更多，它实际上可以在报警时刻到来时，完成你在程序中要求它完成的任何事情，如发出简单的“嘟嘟”声或一支悦耳的乐曲声，同时在屏幕上显示出提示信息，也许还想让 PC 机在打印机上打印出当时的运行结果等，这些工作都是你自己编写的报警程序中要求完成的操作。

BIOS 中断的类型 1AH(AH=6、7)提供了使 PC/AT 进入或退出报警时钟的功能。报警程序的中断类型为 4AH。

为了使用报警功能，我们要做下列几件工作：

1. 编写执行报警操作的中断处理程序。
2. 把中断处理程序的地址保存在类型为 4AH 的中断向量单元中。
3. 把报警时间装入 CH(小时)、CL(分)和 DH(秒)。
4. 通过执行 INT 1AH 的 AH=6 功能，使计算机进入允许报警状态。

设置报警程序的一般格式如下：

```
;This program makes the PC execute
;  an interrupt service routine
;  called ALARM at a specified time
        MOV     AL,4AH          ;make alarm vector
        MOV     DX,SEG ALARM    ;  point to alarm
        MOV     DS,DX
        LEA     DX,ALARM
        MOV     AH,25H
```

```
        INT     21H
        MOV     CH,HOURS        ;specify the alarm time
        MOV     CL,MINUTES
        MOV     DH,SECONDS
        MOV     AH,6            ;set the alarm
        INT     1AH
        ...
        ...
ALARM   PROC
        ...                     ;instructions in alarm routine
        ...
        IRET
ALARM   ENDP
```

图 11.002

一旦 PC 机被设置为允许报警状态，实时钟在每次到达所设置的时间值时，PC 就产生一次中断，执行报警中断处理程序。为了解除或禁止报警，既可以重新启动一次计算机，也可以执行 INT 1AH 的 AH=7 的功能。重新启动计算机，使中断向量表中全部装入初始的中断向量。执行 INT 1AH，可以在中断向量表中保留用户所设置的中断向量，它只是告诉 BIOS 停止检测报警时间。

下面我们编写一个报警程序，这个程序比较长，但它划分成了几个功能子程序，所以还是比较容易读懂的。这个报警程序包括以下几项工作：

1. 清除报警。如果先前已经设置了报警，而现在不经清除又再次设置报警，则 DOS 只置“1”进位标志(CF)，而不做其它工作。
2. 使报警向量指向一个叫做 ALARM 的中断处理子程序，它的工作是打印出一个警呼号“!”并发出“嘟嘟”声。
3. 由用户输入报警时间(时、分、秒)，并把输入的时间值转换为压缩的 BCD 码。这部分工作由 GET_TIME 子程序完成。
4. 用户每输入一位时间值，即调用 CHECK 子程序，它检查输入的数字是否在 0 到 9 的范围之内，如果不是，则显示出错信息。
5. 用 INT 1AH 的 AH=6 功能设置报警。

```
TITLE SETALARM -- Set the alarm
;set the alarm based on user values
        extrn gensound : far
stack segment para stack 'stack'
        db 64 dup('stack...')
stack ends
;-----------------------------------------------
dseg    segment para public 'data'
bad_data        db  'Digits must be between 0 and 9'
                db  13,10,'$'
bad_hrs         db  'Hours must be between 0 and 23'
```

```
                db   13,10,'$'
bad_mins        db   'Minutes must be between 0 and 59'
                db   13,10,'$'
bad_secs        db   'seconds must be between 0 and 59'
                db   13,10,'$'
crlf            db   13,10,'$'
prompt          db   'Set the alarm time.',13,10
                db   'Enter values or press enter for'
                db   'zero',13,10,'$'
ask_hrs         db   'Hour (0 - 23) : $'
ask_mins        db   'Minute (0 - 59) : $'
ask_secs        db   'Second (0 - 59) : $'
user$           db   3,3 dup(?)              ;user's response
dseg            ends
;-------------------------------------------------------------------
cseg            segment para public 'code'
                assume cs : cseg, ss : stack, ds : dseg
entry           proc    far
                push    ds                   ;set up return address
                sub     ax,ax
                push    ax
                mov     al,4ah               ;read the type 4A vector
                mov     ah,35h
                int     21h
                push    es                   ;save it on the stack
                push    bx
                mov     ah,7                 ;reset the alarm
                int     1ah
                mov     al,4ah               ;make the 4A vector
                                             ;  point to alarm
                mov     dx,seg alarm
                mov     ds,dx
                mov     dx,offset alarm
                mov     ah,25h
                int     21h
                mov     ax,dseg              ;point to dseg segment
                mov     ds,ax
                lea     dx,prompt            ;ask for alarm time
                mov     ah,9
                int     21h
hour :          lea     dx,ask_hrs           ;ask for alarm hour
                mov     ah,9
```

```
        int     21h
        call    get_time        ;convert it to BCD
        jc      hour
        cmp     bl,23h          ;make sure it's < 24
        jna     hrs2ch
        lea     dx,bad_hrs
        mov     ah,9
        int     21h
        jmp     hour
hrs2ch: mov     ch,bl           ;and put it in CH
min:    lea     dx,ask_mins     ;ask for alarm minutes
        mov     ah,9
        int     21h
        call    get_time        ;convert it to BCD
        jc      min
        cmp     bl,59h          ;make sure it's < 60
        jna     min2cl
        lea     dx,bad_mins
        mov     ah,9
        int     21h
        jmp     min
min2cl: mov     cl,bl           ;and put it in CL
sec:    lea     dx,ask_secs     ;ask for alarm
        mov     ah,9            ;  seconds
        int     21h
        call    get_time        ;convert it to BCD
        jc      sec
        cmp     bl,59h          ;make sure it's < 60
        jna     sec2dh
        lea     dx,bad_secs
        mov     ah,9
        int     21h
        jmp     sec
sec2dh: mov     dh,bl           ;and put it in DH
        mov     ah,6            ;set the alarm
        int     1ah
        mov     dx,32000
mainp1: mov     cx,500
mainp2: loop    mainp2
        dec     dx
        jne     mainp1
res:    pop     dx              ;restore the 4ah vector
        pop     ds
        mov     al,4ah
```

```
                mov     ah,35h
                int     21h
                ret                             ;and exit
entry           endp
;following is the alarm routine
alarm           proc
                push    ax                      ;save the caller's reg.
                push    bx
                push    di
                sti
                mov     ah,2
                mov     dl,'!'
                int     21h
                mov     di,1000                 ;frequency
                mov     bx,100                  ;beep for one second
                call    gensound
                pop     di
                pop     bx
                pop     ax
                iret
alarm           endp
;This procedure reads up to two keys into
;  string buffer user$ ,then converts them into
;  a packed BCD number in BL
;  if either key is invalid, it prints an
;  error message and set CF to 1
get_time        proc
                lea     dx,user$                ;input user's response
                mov     ah,0ah
                int     21h
                lea     dx,crlf                 ;adrance to next line
                mov     ah,9
                int     21h
                cmp     user$+1,1               ;check character count
                jae     convert
                sub     bl,bl                   ;user pressed enter
                ret
convert:        mov     cl,4
                mov     al,user$+2              ;get first character
                call    check
                jc      leave                   ;if it is valid
                and     al,0fh                  ;convert it to BCD
                mov     bl,al
                cmp     user$+1,2               ;is there another char?
```

```
                jb      clr_cf
                shl     bl,cl           ;yes put first digit in high bits
                mov     al,user $ +3    ;get second character
                call    check           ;check this character
                jc      leave           ;if it is valid
                and     al,0fh          ;convert it to BCD
                or      bl,al
clr_cf :        clc
leave :         ret
get_time        endp
;this procedure checks whether an entry
;   value is between 0 and 9,if not ,it
;   displays an error message
;   and sets CF to 1
check           proc
                cmp     al,'0'          ;if entry is < 0
                jb      error
                cmp     al,'9'          ;or > 9
                ja      error           ;print message
                clc                     ;otherwise,clear CF
                ret                     ; and return
error :         lea     dx,bad_data     ;display error message
                mov     ah,9
                int     21h
                stc                     ;set CF
                ret                     ;and return
check           endp
cseg            ends
;-------------------------------------------------------
                end     entry
```

图 11.11 SETALARM 程序

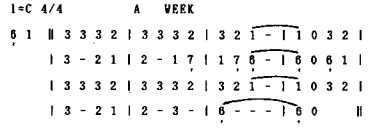
这个程序为我们提供了利用 PC 机发声功能的又一个实例，需要说明的是，INT 1AH 的 AH=2(读取实时钟)以及 AH=6(设置报警)，AH=7(清除报警)是 PC/AT 机处理日时钟的功能，所以 SETALARM 程序应在 AT 机上运行：

```
C)MASM   GENSOUND
C)MASM   SETALARM
C)LINK   SETALARM+GENSOUND
```

习　题

11.1　按下面的乐谱编写一个乐曲程序。

一个星期



11.2 编写用键盘选择计算机演奏歌曲的程序，程序首先在屏幕上显示出歌曲名单如下：

A music 1

B music 2

C music 3

当从键盘上输入歌曲的序号A，B或C时，计算机则演奏所选择的歌曲，当键盘上按下0键时，演奏结束。

第十二章　磁盘文件存取技术

外部设备一般分为两类，一类为字符设备，一类为大容量存储设备。前面几章介绍的键盘、显示器、打印机、串行通讯口等都是字符设备。目前微型机普遍使用的大容量存储设备是温彻斯特(Winchester)硬磁盘和 5-1/4 英寸软磁盘，这两种磁盘容量大，速度快，是微型机理想的外存储器。

IBM PC 使用的 5-1/4 英寸软磁盘是单密度或双密度的，每面有 40 个磁道，按每扇区 512 个字节格式化后，单面盘容量达 160KB 或 180KB，双面盘可达 320KB 或 360KB。IBM PC/XT(扩充型)与基本型的主要差别是增加了一台 10MB 的硬磁盘，这种盘片不可更换，所以也叫固定盘。对用户程序来说，软磁盘和硬磁盘的读写操作几乎是相同的，不同的是硬盘的驱动器指示符为 C，软盘为 A 或 B。

IBM PC 磁盘操作系统(DOS)提供了一组 DOS 磁盘存取功能，我们可以很方便地引用这组功能调用从磁盘上读取某个文件或把一个文件写入磁盘中去。所谓文件就是存放在磁盘上的程序或数据。DOS 1.00 和 1.10 为读写磁盘文件提供了三种方式：顺序存取方式，随机存取方式和随机分块存取方式。DOS2.00 以上的版本除了兼容这三种存取方式而外，还提供了一个更新的方法：文件代号式磁盘存取。在低一级的 DOS 功能中，中断 25H 和 26H 提供了按磁盘扇区号来绝对寻址的方法。最低级的是用 BIOS 中断 13H，此功能要求指定扇区号及磁道号来进行读写操作。本章我们将介绍一些有关磁盘存取的基本概念，并分别讨论 DOS 和 BIOS 所采用的几种磁盘存取方式，读者可根据具体的程序运行环境有选择地学习本章的内容。

12.1　利用文件控制块(FCB)的磁盘存取方式

不论采用什么存取方式读写磁盘文件，都有一些共同的问题需要解决。

首先，用户程序要通知操作系统将要存取哪个文件，为此专门开辟了一块内存区域作为用户程序和操作系统之间传递信息的空间。用户程序将文件名、扩展名、当前记录等有关信息保存在这个特定的内存区中，操作系统由此得知将要存取的文件的位置和大小。我们把这个特定的内存区称为文件控制块(File Control Block)。

其次，从磁盘读出的数据或要写入磁盘的数据也必须存放在一个约定的内存区中，这个内存区叫做数据传输区(Data Transfer Area)。

第三，在读写一个文件之前，必须先打开这个文件。打开文件就是通知操作系统寻找指定的文件，并在 FCB 中填入驱动器号，记录大小、文件大小等信息。

第四，在存取文件之后，特别是在写入文件之后，务必将此文件关闭。通过关闭文件使操作系统最后确定此文件的目录项。如果写入一个文件而忘记将其关闭，很可能导致文件的部分丢失或全部丢失。

为此，DOS 提供了一系列利用 FCB 进行磁盘文件管理的功能调用，很好地处理了以上几个问题。下面我们将分别介绍使用 FCB 的三种存取方式：顺序存取方式、随机存取方式和随机分块存取方式以及有关的 DOS 功能调用。

为了便于学习本节的内容，我们把所涉及到的 DOS 功能调用集中列于表 12.1

表 12.1 INT 21H 文件管理调用功能

AH	功能	调用参数	返回参数
0FH	打开文件	DS=FCB 的段地址	AL=00 文件被找到
		DX=FCB 的偏移地址	=FF 文件未找到
10H	关闭文件	DS=FCB 的段地址	AL=00 目录修改成功
		DX=FCB 的偏移地址	=FF 目录中未找到文件
16H	建立文件	DS=FCB 的段地址	AL=00 文件建立成功
		DX=FCB 的偏移地址	=FF 无磁盘空间
1AH	置 DTA 地址	DS=DTA 的段地址	无
		DX=DTA 的偏移地址	
14H	顺序读	DS=FCB 的段地址	AL=00 读成功
		DX=FCB 的偏移地址	=01 文件结束，记录中无数据
			=02 DTA 空间不够
			=03 文件结束，记录不完整
15H	顺序写	DS=FCB 的段地址	AL=00 写成功
		DX=FCB 的偏移地址	=01 盘满
			=02 DTA 空间不够
21H	随机读	DS=FCB 的段地址	AL=00 读成功
		DX=FCB 的偏移地址	=01 文件已结束
			=02 DAT 太小，传输结束
			=03 读到部分记录
22H	随机写	DS=FCB 的段地址	AL=00 写成功
		DX=FCB 的偏移地址	=01 盘满
			=02 DAT 太小，传输结束
27H	随机分块读	DS=FCB 的段地址	AL=00 全部记录读成功
		DX=FCB 的偏移地址	=01 文件结束，最后记录完整
		CX=读取的记录数	=02 DTA 太小，传输结束
			=03 文件结束，最后记录不完整
			CX=读取的实际记录数
28H	随机分块写	DS=FCB 的段地址	AL=00 全部记录写成功
		DX=FCB 的偏移地址	=01 盘满
		CX=写入的记录数	=02 DTA 空间满
			CX=写入的实际记录数
23H	测文件大小	DS=FCB 的段地址	AL=00 找到文件，记录
		DX=FCB 的偏移地址	数置入 FCB 随机记录域
			AL=FF 未找到文件

12.1.1 文件控制块 FCB

文件控制块 FCB 是在用户程序和操作系统之间传递有关磁盘文件信息的存储区，它一般定义在程序的数据段，共 36 个字节分为 10 个信息域。例如，我们为 NAMEFILE.DAT 文件在数据段定义的 FCB 如下：

```
MY_FCB    LABEL    BYTE
```

```
FILE_DRIVE        DB 0                ;set before open
FILE_NAME         DB 'NAMEFILE'       ;set before open
FILE_EXT          DB 'DAT'            ;set before open
FILE_CURR_BLOCK   DW 0                ; set by open,change as needed
FILE_REC_SIZE     DW 0                ;set by open,change if needed
FILE_SIZE         DW    2 DUP(?)      ;set by system
FILE_DATE         DW    ?             ;set by system
FILE_POSITION     DB 10 DUP(?)        ;set by system
FILE_CURR_REC     DB 0                ;set before sequence read
FILE_REL_REC      DW  2   DUP(?)      ; set before random read
```

下面我们按字节说明这 10 个信息域。

字节	信息域说明
0	磁盘驱动器 00=默认驱动器;01= 驱动器 A; 02=驱动器 B;03=驱动器 C,以此类推。
1—8	文件名 文件名不足 8 个字节时,用空格补足。
9—11	扩展名 扩展名少于 3 个字节时,用空格补充。 DOS 把文件名和扩展名存入目录中。
12—13	当前块号 一块包括 128 个记录。当前块号和当前记录号指定要读写的记录。
14—15	记录大小 打开文件操作把记录大小初始化为 128(80H),在打开文件之后,读写文件之前,可以把此项置为所需要的记录大小。
16—19	文件大小 当建立一个文件时,DOS 计算并把文件的大小(记录号×记录大小)存入目录中,打开文件操作,从目录中取出文件大小并把它存入此域。用户程序可以读这个域,但不能改变它。
20—21	日期 当建立一个文件或修正日期时,DOS 把年、月、日记录在目录中,打开文件操作把日期从目录中取出并存入此域。
22—31	由 DOS 填 入
32	当前记录号 此项是当前块中的当前记录号(0—127),通常当前记录号初始化为 0,但用户程序可把它设置为 0—127 中的任何数。
33—36	随机记录号 随机读写时,此项为随机记录号。若记录长度大于 64 字节时,仅使用该域的前三个字节,第 36 字节总为 0,这是因为文件的长度是有限制的(<1,073,741,824 字节)。

扩展的 FCB 在驱动器号之前又扩充 7 个字节，它可以使你用指定的属性来处理文件。这 7 个字节的第一个字节为 0FFH，表示该 FCB 为扩展的文件控制快，最后一个字节为属性字节，它为存取的磁盘文件规定了特殊的属性，如隐含文件，只读文件或系统文件等。中间的 5 个字节必须为 0。

12.1.2 顺序存取方式

顺序存取方式是最简单，也是最早用于存取磁盘内容的方式。这种方式把文件划分成记录(record)，然后从文件的第一个记录到最后一个记录顺序进行存取。

12.1.2.1 顺序写磁盘文件

把一个文件写入磁盘时，这个文件可能是一个新文件，也可能是一个已存在的文件。如果是一个新文件，在写操作之前，必须用建立文件功能(16H)来建立新文件。如果是一个已存在的文件，建立文件功能将覆盖掉原有的文件，并把文件长度清为 0。

```
MOV   BH,16H                ;Create disk file
MOV   DX,SEG MY_FCB         ; Addr of FCB
MOV   DS,DX
MOV   DX,OFFSET MY_FCB
INT   21H                   ;Call DOS
```

DOS 将在目录中查找 FCB 指定的文件，如果发现有相同的文件名，DOS 仍使用这个目录空间，如果没有发现，则寻找一个未用的目录空间。建立文件操作还检查可用的磁盘空间并在 AL 寄存器中回送信息。

AL=00　建文件成功

AL=FF　文件未建立，无磁盘空间。

写入磁盘的记录要保存在数据传送区 DTA，因为 FCB 中有记录大小的信息域，故 DTA 不需要指明记录尾。在写操作之前，用功能 1AH 来给 DOS 提供 DTA 的首地址。下面是初始化 DTA 地址的指令：

```
MOV   AH,1AH                ;set address of DTA
MOV   DX,SEG MY_DTA         ;address of DTA
MOV   DS,DX
MOV   DX,OFFSET MY_DTA
INT   21H                   ;Call DOS
```

如果程序只处理一个磁盘文件，那么在执行读写之前只需设置一次 DTA 地址。如果程序要处理多个文件，那么在读写每个文件之前必须设置相应的 DTA 地址。

顺序写磁盘文件时，调用 DOS 功能 15H：

```
MOV   AH,15H                ;write records sequentially
MOV   DX,SEG MY_FCB         ;addr of FCB
MOV   DS,DX
MOV   DX,OFFSET MY_FCB
INT   21H                   ; call DOS
```

在调用顺序写功能之前，文件已被打开或建立，当执行顺序写时，一个新记录就被写到文件中，并把记录号加 1。如果被写记录的长度不足以填满一个磁盘扇区，则由操作系统把这些记录放在缓冲区中，等到缓冲区中的记录足以填满一个扇区或文件被关闭时，这个缓冲区的记

录才一起被写入。例如，每个记录的长度是 128 字节，写操作将把 4 个记录存入缓冲区中(4×128=512)然后把它们一起写入一个磁盘扇区。

对于一次成功的写操作，DOS 将会递增 FCB 中的文件大小域并使当前记录号加 1。在当前记录号超过 128 时，此功能把当前记录号置为 0，并使 FCB 中的当前块号增 1，然后由 AL 寄存器回送代码：

AL =00 写成功
=01 磁盘满
=02 DTA 空间满

记录全部写入磁盘后，你可以接着写入一个文件结束符 1AH，然后调用功能 10H 来关闭文件：

```
MOV     AH,10H              ; close the file
MOV     DX,SEG MY_FCB       ; addr of FCB
MOV     DS,DX
MOV     DX,OFFSET MY_FCB
INT     21H                 ; call DOS
```

关闭文件操作将用日期和大小修改目录。下面的代码返回到 AL 寄存器：

AL =00 写成功
=FF 文件不在正确位置上，这可能是由于换盘引起的。

下面我们编写一个程序，它把由键盘输入的姓名作为记录顺序写入磁盘文件，文件控制块 FCBREC 定义在程序的数据段。整个程序由下列子程序组成：

BEGIN 初始化段寄存器，调用 CREATEF 来建立文件，并设置 DTA 地址，调用 INPUTF 从键盘上接收学生姓名。如果输入结束，CLSEF 关闭文件。

CREATEF 在目录中建立名单文件，把记录的长度设置为 32(20h)，并初始化 DTA 的地址。

INPUTF 提示并接收输入的姓名，调用 WRITEF 把名单写入磁盘。

DISP 处理卷屏和设置光标。

WRITEF 把记录写入磁盘文件

CLSEF 写入一个文件结束符并关闭文件

ERRM 建立或写入操作失败时显示出错信息。

例 12.1 顺序写磁盘文件程序

```
TITLE FCBWRIT.EXE---Use FCB to write file of names
;-------------------------------------------------
stack   segment  para stack  'stack'
        dw 80 dup(?)
stack   ends
;-------------------------------------------------
dseg    segment  para  'data'
reclen  equ      32
namepar label    byte              ;name parameter list
maxlen  db       reclen            ;max length of name
namelen db       ?                 ;act length of name
```

```
namedta  db   reclen dup(' ')      ; DTA

fcbrec   label byte                ;FCB for disk file

fcbdriv  db   0                    ;drive C
fcbname  db   'NAMEFILE'           ;file name
fcbext   db   'DAT'                ;extension
fcbblk   dw   0                    ;current block #
fcbrcsz  dw   ?                    ;record size
fcbflsz  dd   ?                    ;DOS file size
         dw   ?                    ;DOS date
         dt   ?                    ;DOS reserved
fcbsqrc  db   0                    ;current record #
         dd   ?                    ;relative record #


crlf     db   13,10,'$'
errcde   db   0
prompt   db   'name ? ',' $'
row      db   01
opnmsg   db   ' * * * open error * * * ',' $'
wrtmsg   db   ' * * * write error * * * ',' $'
dseg ends
;-------------------------------------------
cseg  segment  para  'code'
begin    proc far
         assume cs:cseg, ds:dseg , ss:slack, es:dseg
         push   ds
         sub    ax,ax
         push   ax
         mov    ax,dseg
         mov    ds,ax
         mov    es,ax
         mov    ah,6               ;scroll up func.
         mov    al,0
         call   scro               ;clear screen
         call   curs               ;set cursor
         call   createf            ;create file,set DTA
         cmp    errcde,0           ;available space?
         jz     contin             ;yes,continue
         ret                       ;no, return to DOS
contin:  call   inputf
         cmp    namelen,0          ;end of input?
         jne    contin             ;no,continue
         call   closef             ;yes,close
```

```
        ret                     ;return to DOS
begin   endp

;       open disk file :
;-------------------------------------------------
createf proc near
        mov     ah,16h          ;request create
        lea     dx,febrec
        int     21h
        cmp     al,0            ;available space?
        jnz     err1            ;no,error

        mov     febrcsz,reclen  ;record size (equ)
        lea     dx,namedta      ;set address of DTA
        mov     ah,1ah
        int     21h
        ret
err1:   lea     dx,opnmsg       ;error message
        call    errm
        ret
createf endp

;       accept  input:
;-------------------------------------------------
inputf  proc near
        mov     ah,09           ;request display
        lea     dx,prompt       ;display prompt
        int     21h

        mov     ah,0ah          ;request input
        lea     dx,namepar      ;accept name
        int     21h
        call    disp            ;handle scrolling
        cmp     namelen,0
        jnc     blank           ;yes, process
        ret                     ;no, exit
blank:  mov     bh,0
        mov     bl,namelen
        mov     namedta[bx],' ' ;store blank
        call    writef          ;call write routine

        cld
        lea     di,namedta      ;clear name field
```

```
        mov     cx,reclen/2
        mov     ax,2020h
        rep     stosw
        ret                     ;exit
inputf  endp

;       scroll,set cursor :
;-------------------------------------------------
disp    proc near
        mov     ah,09           ;request display
        lea     dx,crlf         ;CR / LF
        int     21h             ;call DOS
        cmp     row,18h         ;bottom of screen?
        jne     pass            ;yes,bypass
        inc     row             ;no,add to row
        ret
pass:   mov     ah,06           ;scroll one row
        mov     al,01
        call    scro
        call    curs            ;reset cursor
        ret
disp    endp

;       write disk record :
;-------------------------------------------------
writef  proc near
        mov     ah,15h          ;request write
        lea     dx,fcbrec
        int     21h
        cmp     al,0            ;valid write
        jz      valid           ;yes
        lea     dx,wrtmsg       ;no
        call    errm            ;call error routine
        mov     namelen,0
valid:  ret
writef  endp
;       close disk file :
;-------------------------------------------------
clsef   proc near
        mov     namedta,1ah     ;set EOF mark
        call    writef
        mov     ah,10h          ;request close
        lea     dx,fcbrec
```

```
        int     21h
        ret
clsef   endp

;       scroll screen :
;-------------------------------------------
scro    proc near               ;AX set on entry
        mov     bh,1eh          ;set yellow on blue
        mov     cx,0
        mov     dx,184fh
        int     10h             ;scroll
        ret
scro    endp
;       set cursor :
;-------------------------------------------
curs    proc near
        mov     ah,02
        mov     bh,0
        mov     dl,0
        mov     dh,row          ;set cursor
        int     10h
        ret
curs    endp
;       disk error routine :
;-------------------------------------------
errm    proc near
        mov     ah,09           ;DX contains
        int     21h             ; address of message
        mov     errcde,01       ;set error code
        ret
errm    endp
;-------------------------------------------
cseg    ends
        end begin
```

图 12.1 FCBWRIT 程序

程序中把每个记录定为32字节，这样写操作在缓冲区中存入16个记录，然后一起写入磁盘的一个扇区。如果输入25个姓名，记录数从1增加到25(19h)，因此文件大小为：

$$25 \times 32 \text{ 字节} = 800 \text{ 字节}(320\text{h})$$

关闭文件操作把缓冲区中其余的9个记录写入磁盘的第二个扇区并修改目录中的日期和文件大小。

为了使程序更加灵活，可以允许用户指定驱动器和文件名，这只要在程序的开始显示一个提示信息，接着从键盘输入驱动器号及文件名，并分别存入 FCB 的第一项(1 个字节)和第二项

(8个字节)中去。

12.1.2.2 顺序读磁盘文件

在读一个磁盘文件之前,必须先打开这个文件,打开文件的指令如下:

```
MOV   AH,  0FH                    ;open the file
MOV   DX,  SEG MY_FCB             ;addr of FCB
MOV   DS,  DX
MOV   DX,  OFFSET MY_FCB
INT   21H                         ; call DOS
```

打开文件操作用 FCB 中指定的文件名检查目录,如果目录中没有这个文件,就把 AL 置为 FFH。如果目录中有这个文件,就把 AL 置为00,并把 FCB 中的文件大小域置为实际的文件大小,把当前块号置为0,把 FCB 中的记录大小置为80H。文件打开后,可以重新设置记录大小。

读入的文件存放在 DTA 中,利用功能调用1AH 来设置 DTA 地址,这和写文件之前设置 DTA 地址是一样的。

利用功能调用14H 顺序读一个磁盘文件:

```
MOV   AH,  14H                    ; read record sequentially
MOV   DX,  SEG MY_FCB
MOV   DS,  DX
MOV   DX,  OFFSET MY_FCB      ;call DOS
```

读文件操作利用 FCB 中提供的信息把磁盘文件传送到 DTA,然后把 AL 寄存器设置如下:

AL =00　　读成功

=01　　文件结束,记录中无数据

=02　　DTA 中没有足够的空间读入一个记录

=03　　文件尾,已经读了用0填好的部分记录。

读操作把一个扇区的内容读入 DOS 缓冲区中,然后根据 FCB 中记录的大小从缓冲区中把记录顺序传送到 DTA,根据 FCB 的指示,读操作再确定下一个扇区并把它的内容读到 DOS 缓冲区中。

对于成功的读操作,DOS 会自动地把 FCB 中的当前记录号增1。在一个顺序读文件 的程序中,通常不必关闭输入文件,这是因为读操作不会改变文件目录,但如果程序要打开和读几个文件,则应在最后关闭它们,因为 DOS 对一次打开的文件数有限制。

例12.2 顺序读磁盘文件程序

此程序可读出由例12.1程序建立起来的学生名单文件(NAMEFILE),并在屏幕上将名单显示出来。例12.1和例12.2可共用同一个 FCB,虽然在这两个程序中 FCB 中各个域的各称可以不一样,但读写操作的文件名必须一致。

读磁盘文件程序(FCBREAD)由如下子程序组成:

BEGIN 初始化段寄存器,调用 OPENF 来打开文件并设置 DTA,调用 READF 来读磁盘记录,如果是文件尾则程序结束,如果不是文件尾,则用 DISP 显示记录。

OPENF 打开文件,设置记录大小为32,初始化 DTA 的地址。

READF 顺序读磁盘记录,读操作自动地递增 FCB 当前记录号。

DISP　显示读出的记录

ERRN　当打开操作或读操作失败时,显示一个出错信息。

```
TITLE FCBREAD. EXE -- Read records created by fcbwrite
```

```
stack   segment  para  stack  'stack'
        dw   80 dup(?)
stack   ends
;-------------------------------------------------------------
dseg    segment  para  'data'
febrec  label byte                ;FCB for disk file
febdriv db    0                   ;drive C
febname db    'namefile'          ;file name
febext  db    'dat'               ;extension
febblk  dw    0                   ;current block #
febrecsz dw   0                   ;record size
        dd    ?                   ;DOS file size
        dw    ?                   ;DOS date
        dt    ?                   ;DOS reserved
febsqrc db    0                   ;current record #
        dd    ?                   ;relative record #

reclen  equ   32
namefld db    reclen dup(' '),13,10,'$'
endcde  db    0
openmsg db    '* * * open error * * *','$'
readmsg db    '* * * read error * * *','$'
row     db    0
dseg    ends
;-------------------------------------------------------------
cseg    segment  para  'code'
begin   proc  far
        assume cs:cseg, ds:dseg , ss:stack, es:dseg
        push  ds
        sub   ax,ax
        push  ax
        mov   ax,dseg
        mov   ds,ax
        mov   es,ax
        mov   ah,06
        mov   al,00
        call  scro              ;clear screen
        call  curs              ;set cursor
        call  openf             ;open file,set DTA
        cmp   endcde,00         ;valid open?
        jnz   exit              ;no,terminate
loop1:  call  readf             ;read disk record
        cmp   endcde,00         ;normal read?
        jnz   exit              ;no,exit
```

```
        call    disp            ;display name
        jmp     loop1           ;continue
exit:   ret                     ;terminate
begin   endp
;       open disk file :
;-------------------------------------------------
openf   proc near
        lea     dx,fcbrec
        mov     ah,0fh          ;request open
        int     21h
        cmp     al,00           ;file found ?
        jnz     error           ;no,error
        mov     fcbrcsz,reclen  ;set record length
        mov     ah,1ah
        lea     dx,namefld      ;set address of DTA
        int     21h
        ret
error:  mov     endcde,01       ;error message
        lea     dx,openmsg
        call    errm
        ret
openf   endp
;       read disk record:
;-------------------------------------------------
readf   proc near
        mov     ah,14h          ;request read
        lea     dx,fcbrec
        int     21h
        cmp     namefld,1ah     ;EOF marker
        jne     tests           ;no
        mov     endcde,01       ;yes
        jmp     rend
tests:  cmp     al,0            ;normal read ?
        jz      rend            ;yes,exit
        mov     endcde,1        ;no
        cmp     al,1            ;end of file ?
        jz      rend            ;yes, exit
        lea     dx,readmsg      ;no,must be
        call    errm            ;       invalid read
rend:   ret
readf   endp
;       display record :
;-------------------------------------------------
disp    proc near
```

```
        mov     ah,09           ;request display
        lea     dx,namefld
        int     21h
        cmp     row,24          ;bottom of screen?
        jae     bottom          ;yes
        inc     row             ;no,inc row
        jmp     dend
bottom: mov     ah,06
        mov     al,01
        call    scro            ;scroll
        call    curs            ;set cursor
dend:   ret
disp    endp
;       scroll screen :
;-------------------------------------------------
scro    proc near               ;AX set on entry
        mov     bh,1eh          ;set color
        mov     cx,0
        mov     dx,184fh        ;request scroll
        int     10h
        ret
scro    endp
;       set cursor :
;-------------------------------------------------
curs    proc near
        mov     ah,02
        mov     bh,0
        mov     dh,row
        mov     dl,0
        int     10h
        ret
curs    endp
;       disk error routine :
;-------------------------------------------------
errm    proc near
        mov     ah,09           ;DX contains addr
        int     21h             ;  of message
        ret
errm    endp
;-------------------------------------------------
cseg    ends
        end     begin
```

图 12.2 FCBREAD 程序

打开文件操作检查目录中的文件名和扩展名，如果符合，则自动设置 FCB 中的文件大小、

记录大小。第一次读操作从当前记录号0开始存取磁盘并把一个扇区的内容(16个记录)读入 DOS 缓冲区。然后把第一个记录传送到 DTA,FCB 中的当前记录号也从00增为01。第二次执行的读操作不必存取磁盘,因为缓冲区中已经有所需要的记录,此时只需把当前记录号为01的记录从缓冲区传送到 DTA,并把当前记录号增为02。每次读操作都重复这样的动作,直到缓冲区中的16个记录都被传送到 DTA。

读当前记录号为16的记录时,实际上又把第二个扇区的内容读入缓冲区,接着连续地再把每个记录从缓冲区传送到 DTA。最后一个记录读取后,如果仍然继续读下一个记录,读操作就会把 AL 寄存器置为01,表明文件结束了。

12.1.3 随机存取方式

随机存取方式的特点是可以随机地存取文件中的任何记录,而不必象顺序存取方式那样必须从文件首开始读所有的记录,一直到所指定的记录。在文件处理中,我们往往需要修改文件中的数据,删除某些数据或在文件中添加一些数据,无疑随机存取方式更具有灵活性。

随机存取方式之所以能读写文件中的任何记录,原因之一是在 FCB 中构造了一个4字节的随机记录号(FCB 中的33－36字节)。为了找到所指定的记录,系统能自动地把这个随机记录号转换为当前块号(FCB 中的12－13字节)和当前记录号,然后执行读或写命令。

12.1.3.1 随机读

打开文件操作和 DTA 的设置对随机存取方式和顺序存取方式都是一样的。假如程序要直接读取随机记录号为05的记录,则在 FCB 中的33－36字节插入随机记录号,并调用 DOS 随机读功能(21H),如果读成功,记录的内容就被传送到 DTA。

```
MOV   AH,   21H
MOV   DX,   SEG MY_FCB
MOV   DS,   DX
MOV   DX,   OFFSET MY_FCB
INT   21H
```

随机读操作把随机记录号转换为当前块号和当前记录号,以此值确定所需要的磁盘记录,并把这个记录传输到 DTA。AL 寄存器中的返回值如下:

00　读成功
01　无有效数据
02　结束传送,因为 DTA 没有足够的空间。
03　已经读到部份用0填写的记录

在以上返回的参数中,没有文件结束的标记,对一个已存在的记录唯一的成功标记是0,当我们设置一个非法的随机记录号或不正确的 DTA 和 FCB 地址时,就会返回出错的标记,这些错误可根据 AL 中是非 0 码来加以判断。

当程序首次请求读取一个随机记录时,读操作利用目录确定记录所在的扇区,从磁盘中把这个扇区的内容都读到缓冲区,再把指定的记录传送到 DTA。例如,记录的长度定为128字节,则一个扇区存有4个记录。对随机记录号为23的存取请求可使下面的4个记录读入缓冲区:

|记录20|记录21|记录22|记录23|

当程序请求读取记录23时,随机读操作首先检查缓冲区,因为记录23已在缓冲区中,它就直接被送入 DTA。如果程序请求读取记录35,而缓冲区中没有记录35,则随机读操作又利用目录确定此记录的位置,并把它所在扇区的内容全部读入缓冲区,接着再把记录35送到 DTA。但是要

注意的是，当随机读操作读出一个记录时，随机记录号不会自动加1，如果不改变随机记录号而又再次请求随机读，则读出的仍是上次读出的记录。因此，在每次对磁盘进行随机存取(读或写)之前都要把正确的随机记录号插入 FCB 中。

例12.3 随机读磁盘文件

RANREAD 程序用随机读功能读取由例12.1程序(FCBWRIT)建立起来的文件。通过在文件范围内的随机记录号，我们能读取任何记录并显示在屏幕上。假如文件的长度为25个记录，那么合法的记录号应是从00到24。随机记录号由键盘输入，它是 ASCII 码形式的1位或两位数字。

随机读程序可由下面的子程序组成：

OPENR 打开文件，把记录大小置为32，设置 DTA 地址。

RECN 从键盘接收记录号，检查是否为0—24之间的合法记录号，然后转换为二进制数插入 FCB 的随机记录域。

READR 利用 FCB 中的随机记录号传送一个记录到 DTA。

DISPR 显示记录。

```
TITLE RANREAD.COM -- Random read records created by febwrit
;-------------------------------------------------------------
cseg    segment  para  'code'
        assume cs:cseg, ds:cseg
        assume ss:cseg, es:cseg
        org      100h
begin:  jmp      main
;-------------------------------------------------------------
febrec  label byte                ;FCB for disk file
febdriv db       1                ;drive A
febname db       'namefile'       ;file name
febext  db       'dat'            ;extension
febblk  dw       0                ;current block #
febrecsz dw      0                ;record size
        dd       ?                ;DOS file size
        dw       ?                ;DOS date
        dt       ?                ;DOS reserved
        db       0                ;current record #
febrnrc dd       0                ;random record #

reclen  equ      32               ;record length
readpar label byte                ;parameter list
maxlen  db       3
actlen  db       ?
readno  db       3 dup(' ')

namefld db       reclen dup(' '),13,10,'$' ;DTA
openmsg db       '*** open error ***',13,10,'$'
```

```
readmsg   db      '* * * read error * * *',13,10,'$'
col       db      0
prompt    db      'record number ? $'
row       db      0
endcde    db      0
;-------------------------------------------------
main      proc near
          mov     ax,cseg
          mov     ds,ax

          call    clrscn      ;clear screen
          call    curs        ;set cursor
          call    openr       ;open file,set DTA
          cmp     endcde,0    ;valid open?
          jz      contin      ;yes,continue
          mov     ax,4c00h    ;no,terminate
          int     21h
contin:   call    recn        ;request record #
          cmp     actlen,0    ;any more requests?
          je      a2          ;no,exit
          call    readr       ;random read
          cmp     endcde,0    ;normal read ?
          jnz     a1          ;no,  bypass
          call    dispr       ;call display routine
a1:       jmp     contin
a2:       mov     ax,4c00h    ;terminate
          int     21h
main      endp
;         open disk routine :
;-------------------------------------------------
openr     proc
          mov     ah,0fh      ;request open
          lea     dx,fcbrec
          int     21h
          cmp     al,0        ;valid open ?
          jnz     b1          ;no,error
          mov     fcbrecsz,reclen ;record size(equ)
          mov     ah,1ah
          lea     dx,namefld  ;set address of DTA
          int     21h
          ret
b1:       lea     dx,openmsg
          call    errm
          ret
```

```
openr   endp
;       get record number :
;-------------------------------------------------
recn    proc near
        mov     ah,09h          ;request display
        lea     dx,prompt
        int     21h
        mov     ah,0ah          ;request input
        lea     dx,recdpar
        int     21h
        cmp     actlen,01       ;check length 0,1,2
        jb      c3              ;length 0,terminate
        ja      c1
        sub     ah,ah           ;length  1
        mov     al,recdno
        jmp     c2
c1:     mov     ah,recdno       ;length 2
        mov     al,recdno+1
c2:     and     ax,0f0fh        ;clear ASCII 3's
        aad                     ;convert to binary
        mov     word ptr febrnrc,ax
c3:     mov     col,20
        call    curs            ;set cursor
        ret
recn    endp
;       read disk record :
;-------------------------------------------------
readr   proc near
        mov     endcde,0        ;clear indicator
        mov     ah,21h          ;random read
        lea     dx,febrec
        int     21h
        cmp     al,0            ;normal read ?
        jz      d1              ;yes,exit
        lea     dx,readmsg      ;no,invalid
        call    errm            ;call error routine
d1:     ret
readr   endp
;       display name :
;-------------------------------------------------
dispr   proc near
        mov     ah,09           ;request display
        lea     dx,namefld
        int     21h
```

```
        inc     row
        mov     col,0
        ret
dispr   endp
;       clear screen :
;-------------------------------------------
clrscn  proc near
        mov     ax,0600h        ;request scroll
        mov     bh,41h          ;color(07 for BW)
        mov     cx,0
        mov     dx,184fh
        int     10h
        ret
clrscn  endp
;       set cursor :
;-------------------------------------------
curs    proc near
        mov     ah,2            ;request set
        mov     bh,0            ;  cursor
        mov     dh,row
        mov     dl,col
        int     10h
        ret
curs    endp
;       disk error routine :
;-------------------------------------------
errm    proc near
        mov     ah,09           ;DX contains addr
        int     21h             ;  of message
        inc     row
        mov     endcde,01
        ret
errm    endp
;-------------------------------------------
cseg    ends
        end     begin
```

图 12.3 RANREAD 程序

子程序 RECN 从键盘接收一个随机记录号并检查输入的实际字符数。输入字符数有下面三种可能：

00 读取记录结束

01 输入一位数字，存放在 AL 中

02 输入两位数字，存放在 AX 中

AAD 指令把 AX 中未压缩的十进制数转换为二进制数，然后再把这二进制的记录号移到 FCB 中的随机记录域。为了清楚起见，我们再举个例子把这个转换过程说明一下。如果我们从键盘输入的记录号是12，这当然是 ASCII 码形式的数字，这样 AX 中的内容就为3132，AND 指令又把它变成0102，接着 AAD 指令又把它转换为000C，这就是二进制形式的记录号。最后按相反的顺序把它存入随机记录域：0C000000。

12.1.3.2 随机写磁盘文件

和顺序写操作一样，在对磁盘进行随机写操作之前，必须建立文件和设置 DTA 地址，然后初始化 FCB 中的随机记录号。调用随机写功能(22H)如下：

```
MOV   AH,    22H                ;request random write
MOV   DX,    SEG MY_FCB
MOV   DS,    DX
MOV   DX,    OFFSET MY_FCB
INT   21H
```

写操作在 AL 中设置的返回参数如下：

00　写成功

01　磁盘空间满

02　传输结束，因为 DTA 中没有足够的空间

FCB 中的随机记录号是一个4字节的域，对于一个小文件，只要初始化最低的一个或两个字节(33和34字节)就行了。如果是一个较大的文件，就要往第3 个和第4个字节中填入随机记录号，这时需要格外注意随机记录号存入时的顺序。

12.1.4 随机分块存取方式

如果程序中有足够的空间，一次随机分块操作能从 DTA 把整个文件写入磁盘，也能把整个文件从磁盘读到 DTA 中，这个功能对表的处理特别有用。

在使用随机分块存取功能时，仍首先要打开文件和初始化 DTA 地址，并设置适当的随机记录号和记录大小。

对于随机分块写，先把要写的记录个数放入 CX 寄存器，再把起始的记录号送入 FCB，然后调用随机写功能(AH=28H)：

```
MOV   AH,    28H                ;request random block write
MOV   CX,    N                  ;N=any number of records
MOV   DX,    SEG MY_FCB         ; addr of FCB
MOV   DS,    DX
MOV   DX,    OFFSET MY_FCB
INT   21H                       ;call DOS
```

随机分块写操作把 FCB 中的随机记录号转换为当前块号和当前记录号，由此确定记录块在磁盘上的起始位置。此操作把返回代码送入 AL 寄存器：

00　全部记录写成功

01　磁盘空间满

随机写操作还使 FCB 的随机记录号和当时块号/记录号指向下一记录，比如已经写00—24记录，紧接着就指向记录25(19H)。

随机分块读操作要求先在 CX 寄存器中放入要求读取的记录个数，然后调用27H 功能：

```
MOV   AH,    27H              ; request random block read
MOV   CX,    N                ; N=number of records
MOV   DX,    SEG MY_FCB       ;   addr of FCB
MOV   DS,    DX
MOV   DX,    OFFSET MY_FCB
INT   21H                     ; call DOS
```

该操作在 AL 中的返回代码为：

00　全部记录读成功

01　已经读到文件尾，最后一个记录完整。

02　DTA 中读入了尽可能多的记录

03　已读到文件尾，最后记录不完整。

随机读操作还把实际读取的记录数放入 CX 寄存器，并把 FCB 中的随机记录号和当前块号、当前记录号置向下一记录。

如果我们想把一个完整的文件从磁盘读出来，但又不知道确切的记录数，怎么办呢？一个简单的方法是定义一个很大的 DTA 数据传输区，如果 AL 中的回送值是01，表明所有的记录都完整地读取出来了，这时查看 CX 寄存器，就可知道文件的实际记录数。这个方法对小文件适用，对大文件就不太适用了。我们还有一个方法是利用 DOS 的文件大小测定功能(AH＝23H)来读取未知的文件大小。

调用测定文件大小的功能之前，应在 FCB 的记录大小域置适当的值，然后调用 DOS 功能23H：

```
MOV   AH,23H                  ; request file size func
MOV   DX,SEG MY_FCB           ; addr of FCB
MOV   DS,DX
MOV   DX,OFFSET MY_FCB
MOV   FILE_REC_SIZE,N         ; N=record size
INT   21H                     ; call DOS
```

测定文件大小的功能根据 FCB 中的文件名在目录中查找相应文件，如果找到则把文件的记录数填写到 FCB 的随机记录域。

下面我们编写一个随机分块读程序(RANBLOCK)将本章例12.1建立的学生名单文件分块读取并显示出来。程序把起始的随机记录号初始化为0。

随机分块读程序包括的子程序如下：

OPENB 打开文件，设置 FCB 记录大小域为32，设置 DTA 地址。

READB 记录数初始化为25并执行随机分块读。

DISPB 在屏幕上显示记录块。

例12.4　随机分块读磁盘文件

```
TITLE RANBLOCK.COM -- Random block read of name file
cseg    segment para 'code'
        assume cs:cseg, ds:cseg
        assume ss:cseg, es:cseg
        org     100h
begin:  jmp     main
;-------------------------------------------------
```

```
fcbrec  label   byte            ;FCB for disk file
fcbdriv db      0               ;drive C
fcbname db      'namefile'      ;file name
fcbext  db      'dat'           ;extension
fcbblk  dw      0               ;current block #
fcbrcsz dw      0               ;record size
fcbflz  dd      ?               ;DOS file size
        dw      ?               ;DOS date
        dt      ?               ;DOS reserved
        db      0               ;current record #
fcbrnrc dd      0               ;random record #

dskrecs db      1024 dup(?),'$' ;DTA for block rec.
endcode db      0
norecs  dw      25              ;number of records

openmsg db      '* * * open error * * *',13,10,'$'
readmsg db      '* * * read error * * *',13,10,'$'
rowctr  db      0
;-------------------------------------------------------------
main    proc    near
        mov             ax,cseg
        mov             ds,ax

        call            clrscn          ;clear screen
        call            curs            ;set cursor
        call            openb           ;open file,set DTA
        cmp             endcode,0       ;valid open?
        jnz             return          ;no,exit
        call            readb           ;read records
        call            dispb           ;call display
return: mov             ax,4c00h        ;no,terminate
        int             21h
main endp
;       open disk file :
;-------------------------------------------------------------
openb   proc near
        mov             ah,0fh          ;request open
        lea             dx,fcbrec
        int             21h
        cmp             al,0            ;valid open ?
        jnz             b1              ;no,error
        mov             fcbrcsz,20h     ;record size
```

```
        mov     ah,1ah
        lea     dx,dskrecs      ;set address of DTA
        int     21h
        ret
b1:     lea     dx,openmsg
        call    errm
        ret
openb   endp
;       read disk block :
;-------------------------------------------------
readb   proc near
        mov     ah,27h          ;random block read
        mov     cx,norecs       ;number of record
        lea     dx,fcbrec
        int     21h
        mov     endcode,al      ;save return condition
        ret
readb   endp
;       display disk block :
;-------------------------------------------------
dispb   proc near
        mov     ah,09           ;request display
        lea     dx,dskrecs
        int     21h
        ret
dispb   endp
;       clear screen routine :
;-------------------------------------------------
clrscn  proc near
        mov     ax,0600h        ;request scroll
        mov     bh,41h          ;color(07 for BW)
        mov     cx,0
        mov     dx,184fh
        int     10h
        ret
clrscn  endp
;       set cursor routine :
;-------------------------------------------------
curs    proc near
        mov     ah,2            ;request set
        mov     bh,0            ;  cursor
        mov     dh,rowctr
        mov     dl,0
        int     10h
```

```
        inc             rowctr
        ret
curs endp
;       disk error routine
;-------------------------------------------
errm proc near
        mov             ah,09           ;DX contains addr
        int             21h             ;  of message
        mov             endcode,01
        ret
errm endp
;-------------------------------------------
cseg ends
    end begin
```

图 12.4 RANBLOCK 程序

随机分块读操作把 FCB 中的随机记录号00转换为当前块号00和当前记录号00，读操作结束时，FCB 中的当前记录号为19H，随机记录号为19000000。

12.1.5 绝对磁盘 I/O

DOS INT 25H 和 INT 26H 是直接处理磁盘文件的绝对读写功能调用。在使用上比 INT 21H 的磁盘存取麻烦一些，也没有 INT 21H 所具有的目录处理和记录分块等优点，但此功能给系统程序员提供了一种直接处理磁盘扇区的手段。

绝对磁盘 I/O 操作把一个磁盘扇区作为一个记录，可以直接存取一个扇区或几个扇区，读写的磁盘位置是根据"逻辑记录号"(绝对扇区)来寻址的。在每个磁道分为9个扇区的双面软盘上，逻辑记录是从0磁道 1扇区开始编号的。

磁道	扇区	逻辑记录号
0	1	0
0	2	1
1	1	9
1	9	17
2	9	26

有一个简单的公式计算逻辑记录号：

$$\text{逻辑记录号}=(\text{磁道}\times 9)+(\text{扇区}-1)$$

例如，2磁道9扇区的逻辑记录号为：

$$(2\times 9)+(9-1)=18+8=26$$

调用绝对磁盘 I/O 功能的指令序列如下：

```
MOV  AL,  drive#            ;0 for A,1 for B,etc
MOV  BX,  SEG BUFF          ;transfer address
MOV  DS,  BX
MOV  BX,  OFFSET BUFF
MOV  CX,  N                 ; N=number of sectors to read /write
```

```
MOV   DX,   record #          ; beginning logical record no.
INT   25H   (or 26H)          ; DOS absolute read/write
```

绝对磁盘读写操作破坏了所有寄存器内容(除段寄存器外),并设置 CF 标志位来表示操作成功与否。CF=0操作成功,CF=1操作不成功。如果操作不成功,AL 给出错误码:

AL =80H 联接失败回答
=40H 查找操作失败
=20H 控制器失败
=10H 数据错(坏的 CRC)
=08H DMA 超越传送
=04H 请求的扇区未找到
=03H 企图在写保护磁盘上写
=02H 地址标志未找到
=01H 非法命令

绝对磁盘读写操作还把标志寄存器入栈。在返回 DOS 时,因为原来的标志位仍在堆栈中,所以在检验 CF 之后,要将标志寄存器从堆栈中取出。

12.2 文件代号式磁盘存取

在 DOS 2.0以上的版本中,为了支持层次结构,引用了树形结构目录,因此相应增加了一个新的存取方式,即文件代号式存取方式(file handles access)。这种方式不再采用文件控制块 FCB,有关文件的各种信息都包括在 DOS 中,对用户程序是透明的。在处理指定文件时,必须使用一个完整的路径名(path name),一旦文件的路径名被送入操作系统,就被赋予一个简单的文件代号(file handle),这个文件代号是一个16位的数。以后对该文件进行读写操作时,就用这个文件代号去查找相应的文件。对于每一个打开的文件,DOS 还为其管理一个读写指针(read/write pointer),读写指针总是指向下一次要存取的文件中的字节,这个读写指针可以移动到文件的任意位置,从而能满足随机存取的要求。

文件代号式存取方式对各种错误采取了更统一的处理方法。在操作过程中,AX 中回送错误代码,这些错误代码对所有代号式存取功能都是相同的,这样为用户进行分析提供了方便。

DOS2.0提供的有关代号式文件管理功能都包括在扩充 DOS 功能调用中。表12.2列出了其中常用的部分功能调用。

表12.2 代号式文件管理功能调用

AH 功 能	调用参数	返回参数
3CH 建文件	DS=ASCIZ 串的段地址	CF=0操作成功
	DX=ASCIZ 串的偏移地址	AX=文件代号
	CX=文件属性	CF=1操作出错
		AX=错误代码
3DH 打开文件	DS=ASCIZ 串的段地址	CF=0操作成功
	DX=ASCIZ 串的偏移地址	AX=文件代号
	AL=存取代码	CF=1操作出错
		AX=错误代码

AH	功 能	调用参数	返回参数
3EH	关闭文件	BX=文件代号	CF=0操作成功
			CF=1出现错误
			AX=错误代码
3FH	读文件或设备	DS=数据缓冲区段地址	CF=0读成功
		DX=数据缓冲区偏移地址	AX=实际读入的字节数
		BX=文件代号	
		CX=读取的字节数	AX=0文件结束
			CF=1读出错
			AX=错误代码
40H	写文件或设备	DS=数据缓冲区的段地址	CF=0操作成功
		DX=数据缓冲区的偏移地址	AX=实际写入的字节数
		BX=文件代号	CF=1出现错误
		CX=写入的字节数	AX=错误代码
42H	移动文件指针	CX=所需字节的偏移地址(高位)	CF=0操作成功
		DX=所需字节的偏移地址(低位)	DX+AX=新指针位置
		AL=方式码	CF=1操作失败
		BX=文件代号	AX=错误代码
43H	检验或改变文件属性	AL=0检验文件属性	CF=0操作成功
		AL=1置文件属性	AL=0
		CX=新属性	CX=属性
		DS=ASCIZ 串的段地址	CF=1操作失败
		DX=ASCIZ 串的偏移地址	AX=错误码

12.2.1 路径名和 ASCIZ 串

当使用扩展功能处理磁盘文件时，首先要告诉 DOS 一个 ASCIZ 串的地址。这个 ASCIZ 串包括文件的路径名和一个全0的字节。路径名说明文件的位置，包括磁盘驱动器、目录路径和文件名。下面是两个 ASCIZ 串。

```
PATHNM1  DB  'B:\TEST. ASM',0
PATHNM2  DB  'C:\UTILITY \NU. EXE',0
```

串中的后斜线，也可能是前斜线，起分割各项的作用。上面两个字符串都用一个0字节作为结束，所以叫做 ASCIZ 串(ASCII_ZERO)。路径名的最大长度允许63个字节，对于请求 ASCIZ 串的中断，要求把 ASCIZ 串的地址装在 DX 寄存器中，例如，可用指令把串的地址送 DX：

```
LEA DX,PATHNM1
```

12.2.2 文件代号和错误返回代码

存取文件要借助于文件代号，文件代号是由打开文件功能和建立文件功能传送到 AX 的一个16位数。当然标准设备不必打开就可直接使用它们的文件代号，因为 DOS 已经预定义了它们的文件代号：

0=标准输入设备

1=标准输出设备

2=标准错误输出设备

3=标准辅助设备

4=标准打印设备

我们建立或打开的文件，其代号从6开始顺序排列，在任一时刻我们最多只能同时打开5个文件，可见16位长的文件代号实际上并没有全部用上。

对于存取磁盘文件，首先用一个ASCIZ串指定文件并调用DOS功能3CH或3DH建立或打开文件。如果成功，操作置CF为0，并把文件代号传送到AX中，这时文件和代号建立了对应关系，所以要注意保存这个代号，例如把它保存在程序中的一个字数据项中，否则丢失了代号相当于丢失了文件。如果操作不成功，CF被置成1，AX中包含的是错误代码，这些错误代码都取自一个统一的错误信息表，如表12.3。

表12.3 错误返回代码

01	非法功能号
02	文件未找到
03	路径未找到
04	同时打开的文件太多
05	拒绝存取
06	非法文件代号
07	内存控制块被破坏
08	内存不够
09	非法存储块地址
10	非法环境
11	非法格式
12	非法存取代码
13	非法数据
14	(未用)
15	非法指定设备
16	试图删除当前的目录
17	设备不一致
18	已没有文件

12.2.3 文件属性

文件属性是一个说明文件特性的字节。比如一个文件被赋于的属性是"只读"，则这个文件只能由用户读取，而不能往文件中写入任何内容，这实际上是保护文件的一种方法。属性字节的各位含义如图12.5。

01—只读文件。该文件不能为写而打开

02—隐文件。用DIR查不到该文件

03—系统文件。用DIR查不到该文件

04—软盘的卷标号

10—子目录

20—已写入并关闭了文件(硬盘用)。

一个文件可以同时具有几种属性，也就是说可以把字节中的几位同时置1，如IBMBIO.

COM 和 IBMDOS. COM 文件既是只读文件，又是隐文件和系统文件。

对一般的程序，最有用的是只读位和隐文件位，因为有时文件不希望别人改动也不希望别人知道。然而大多数文件都不使用这些特征位，所以属性字节通常为00。属性字节只用上6位，但规定用一个16位的寄存器 CX 存放这个字节，这可能也是为了以后扩充方便而这样设置的。

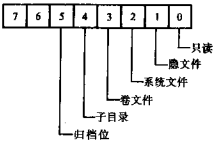


图 12.5 文件属性字节

使用改变文件属性功能(43H)可以改变现有文件的属性。

```
MOV   AH,  43H              ;check or change attribute
MOV   AL,  10H              ;set file attribute
MOV   CX,  01               ;read only
MOV   DX,  SEG FNAME        ;addr of ASCIZ string
MOV   DS,  DX
MOV   DX,  OFFSET FNAME
INT   21H                   ;call DOS
```

INT 21H 的功能43H 可以检验或改变目录中文件的属性，DX 中放 ASCIZ 串的地址，为了检验文件属性，AL 置为00，操作后当前的属性值返回到 CX 寄存器中。改变文件属性时，置 AL 为01，CX 放入新的属性，操作把这个新属性置入目录项。操作失败，AX 中返回的错误码为02、03或05。

12.2.4 写磁盘文件

写一个新文件或重写一个旧文件，首先要建立文件并赋给它一个属性。如果 DOS 发现要建立的文件已经存在，那么原来的文件就被破坏。

建立文件的功能调用是3CH，调用此功能时，在 DX 中装入 ASCIZ 串的地址，在 CX 中装入文件的属性。例如建立一个有正常属性文件的指令序列如下：

```
MOV   AH,  3CH              ;request create
MOV   CX,  00               ;normal file
LEA   DX,  PATHNM1          ;ASCIZ string
INT   21H                   ;call DOS
JC    ERROR                 ;exit if error
MOV   HANDLE1,  AX          ;save handle in DW
```

对于一次成功的操作，DOS 把指定的属性填入目录，清除进位位，并把文件代号回送到 AX 寄存器，以后的所有操作都使用这个代号。如果文件已经存在，操作将文件长度置为0。如果操作把 CF 置为1，则说明建立文件有错误，错误代码回送到 AX 寄存器，可能出现的错误为03、04或05。错误码05(拒绝存取)说明目录已经满了或所请求的文件是一个只读文件。

写磁盘文件是利用功能调用40H，在 BX 中装入文件代号，要写入的字节数放在 CX 中，输入缓冲区的地址放在 DX 中。下面的指令是把 OUTREC 数据区中的256个字节写入磁盘文件。

```
HANDLE1   DW    ?
OUTREC    DB    256DUP(?)
```

```
MOV     AH,  40H                ;request write
MOV     BX,  HANDLE1            ;handle
MOV     CX,  256                ;number of bytes
LEA     DX,  OUTREC             ;addr of output area
INT     21H                     ; call DOS
JC      ERROR2                  ; test for error
CMP     AX,  256                ;all bytes written?
JNE     ERROR3
```

把文件从内存写入磁盘的操作成功，则 CF 清0，并把实际写入的字节数送入 AX 寄存器。如果磁盘空间满，实际写入的字节数可能会和要求写入的字节数不同。操作失败的标志同样是 CF=1，并在 AX 中回送错误码05(拒绝存取)或06(非法文件代号)。

当写入文件操作完成后，必须用 DOS 功能调用3EH 来关闭文件，以确保操作系统将文件记录在磁盘上，这个操作只要求在 BX 中放入文件代号，关闭文件时，DOS 把内存缓冲区(等待输入字符满512个字节以填满一个扇区)中的数据写入磁盘，并修改目录和 FAT(文件分配表)。

```
MOV   AH,  3EH              ;request close
MOV   BX,  HANDLE1          ;file handle
INT   21H                   ; call DOS
```

关文件操作在 AX 中返回的错误码只可能是06(非法文件代号)。

例12.5 用文件代号建立文件

该程序从键盘接收一个由姓名组成的文件。它包括以下几个子程序：

CREATH 利用功能调用3CH 来建立文件，并把文件代号保存在 HANDLE 数据项中。

PROCH 从键盘接收输入并把输入缓冲区中其余的单元用空(blank)填入。

WRITH 利用功能调用40H 写文件。

CLSE 利用功能调用3EH 来关闭文件以建立相应的目录项。

```
TITLE HANCREAT. EXE -- Creat disk file of names
;-------------------------------
stack segment para stack 'stack'
      dw   80 dup(?)
stack ends
;-------------------------------------------
dseg segment para 'data'
namepar    label byte                  ;name parameter list
maxlen     db  30
namelen    db  ?
namerec    db  30 dup(' '),0dh,0ah     ;entered name
                                       ;CR/LF for writing
errcde     db  0
handle     dw  ?
pathnam    db  'A:/NAME. DAT',0
prompt     db  'name ?'
row        db  01
opnmsg     db  '* * * open error * * *',0dh,0ah
```

```
wrtmsg      db  ' * * *  write error  * * * ',0dh,0ah
dseg ends
;-------------------------------------------------
cseg    segment para 'code'
begin   proc far
        assume cs:cseg,ds:dseg,ss:stack,es:dseg


        push    ds
        sub     ax,ax
        push    ax
        mov     ax,dseg
        mov     ds,ax
        mov     es,ax
        mov     ax,0600h
        call    scren           ;clear screen
        call    curs            ;set cursor
        call    creath          ;create file,set DTA
        cmp     errcde,0        ;create error ?
        jz      contin          ;yes,continue
        ret                     ;no,return to DOS
contin: call    proch
        cmp     namelen,0       ;end of input
        jne     contin          ;no,continue
        call    clseh           ;yes,close
        ret                     ;return to DOS
begin   endp
;       create disk file
;-------------------------------------------------
creath  proc near
        mov     ah,3ch          ;request create
        mov     cx,0            ;normal attribute
        lea     dx,pathnam
        int     21h
        jc      a1              ;error ?
        mov     handle,ax       ;no,save handle
        ret
a1:     lea     dx,opnmsg       ;error message
        call    errm
        ret
creath  endp
;       accept input
;-------------------------------------------------
proch   proc near
        mov     ah,40h          ;request display
```

```
        mov     bx,01           ;01=output device
        mov     cx,06           ;length of prompt
        lea     dx,prompt       ;display prompt
        int     21h


        mov     ah,0ah          ;request input
        lea     dx,namepar      ;accept name
        int     21h
        cmp     namelen,0       ;is there a name ?
        jne     b1              ;yes
        ret                     ;no,exit
b1:     mov     al,20h          ;blank for storing
        sub     ch,ch
        mov     cl,namelen      ;length
        lea     di,namerec
        add     di,cx           ;address + length
        neg     cx              ;calculate remaining
        add     cx,30           ;  length
        rep     stosb           ;set to blank
        call    writh           ;write disk record
        call    scrl            ;check for scroll
        ret
proch   endp
;       check for scroll
;-------------------------------------------------------
scrl    proc near
        cmp     row,18h         ;bottom of screen ?
        jae     c1              ;yes,bypass
        inc     row             ;no,add to row
c1:     mov     ax,0601h        ;scroll one row
        call    scren
        call    curs            ;reset cursor
        ret
scrl    endp
;       write disk record
;-------------------------------------------------------
writh   proc near
        mov     ah,40h          ;request write
        mov     bx,handle
        mov     cx,32           ;for name and CR/LF
        lea     dx,namerec
        int     21h
        jnc     d1              ;valid write ?
        lea     dx,wrtmsg       ;no
```

```
        call    errm            ;call error routine
        mov     namelen,0
dl:     ret
writh   endp
;       close disk file
;-------------------------------------------------
clseh   proc near
        mov     namerec,1ah     ;set EOF mark
        call    writh
        mov     ah,3eh          ;request close
        mov     bx,handle
        int     21h
        ret
clseh   endp
;       scroll screen
;-------------------------------------------------
scren   proc near               ;AX set on entry
        mov     bh,1eh          ;set yellow on blue
        mov     cx,0
        mov     dx,184fh
        int     10h             ;scroll
        ret
scren   endp
;       set cursor
;-------------------------------------------------
curs    proc near
        mov     ah,02
        mov     bh,0
        mov     dh,row          ;set cursor
        mov     dl,0
        int     10h
        ret
curs    endp
;       disk error routine
;-------------------------------------------------
errm    proc near               ;DX contains address
        mov     ah,40h          ;  of message
        mov     bx,01           ;handle
        mov     cx,21           ;length
        int     21h
        mov     errcde,01       ;set error code
        ret
errm    endp
;-------------------------------------------------
```

```
cseg    ends
        end begin
```

图 12.6 HANCREAT 程序

输入缓冲区(NAMEREC)有30个字节,包括回车换行符共32个字节。此程序把32个字节作为一个固定长度的记录写入。在程序开始显示提示符和程序末尾显示错误信息都采用的是代号式输入/输出,显示器预先定义的文件代号是01,所以可以直接使用此文件代号向设备写入信息。在下一节我们将专门介绍字符设备的代号式输入/输出。

12.2.5 读磁盘文件

调用读文件或设备功能(3FH),要求先把文件打开取得文件代号,然后按照指定的字节数从磁盘中把文件读出,送入内存中预先定义好的数据缓冲区。如果读入的字节数大于缓冲区空间,那么这些多余的数据将送到程序所占空间之上的存储器中。

打开文件操作(3DH)要检查文件名是否合法,文件是否有效。文件名是一个 ASCIZ 串,其地址装入 DX 寄存器,并在 AL 中设置存取代码:

0——为读而打开文件

1——为写而打开文件

2——为读和写打开文件

这些存取代码告诉操作系统你打开文件的目的是什么,例如打开一个只读文件,而目的是为了向文件中写入内容(存取代码为01),则操作将回送一个错误码05(拒绝访问)。因此可以看出,文件属性和存取代码结合起来能防止非法读写文件。

下面是一个为读而打开文件的例子:

```
MOV     AH,3DH          ;request open
MOV     AL,00           ;read only
LEA     DX,PATHNM1      ;ASCIZ string
INT     21H             ; call DOS
JC      ERROR4          ;exit if error
MOV     HANDLE2,AX      ;save handle in DW
```

如果指定的文件存在,打开文件操作将把记录长度置为1,确定它的属性,清除 CF 标志位,并把文件代号放入 AX。

如果指定的文件不存在,操作置位 CF,并在 AX 中返回一个错误代码:02,04,05或12。所以在打开文件之后一定要检查 CF 位。

读文件调用 DOS 功能3FH,首先在 BX 中设置文件代号,在 CX 中装入要读取的字节数,在 DX 中放入输入数据缓冲区的地址,然后调用 DOS。

下面的指令读取一个512字节的记录:

```
HANDLE2     DW      ?
INPREC      DB      512 DUP(?)
            MOV     AH,3FH              ;request read
            MOV     BX,HANDLE2          ; handle
            MOV     CX,512              ;record length
            LEA     DX,INPREC           ;addr of input area
            INT     21H                 ;call DOS
```

```
        JC      ERROR5          ;test for error
        CMP     AX,0            ;zero bytes read
        JE      ENDFILE
```

如果读操作成功地把记录读入存储器，则清除 CF 位并把实际读入的字节数放入 AX。假如 AX 为0，表明试图从文件尾开始读。

如果读操作不成功，把 CF 位置1并在 AX 中返回错误码05(拒绝存取)或06(非法文件代号)。

因为 DOS 限制一次打开的文件数，所以在读取文件之后必须关闭这些文件。

一个文件分几次读取，取决于文件的大小和输入缓冲区的空间。如果文件比较小，又有足够的输入缓冲区，则由一次调用就能读进整个文件的内容。如果文件很大，程序中不能设置如此大的缓冲区，必须分几次重复调用读功能，直到整个文件结束。

例12.6 利用代号读取一个文件

读取的文件是由例12.5建立的 NAME. DAT 文件。程序调用 DOS 功能3DH 来打开文件，并把文件代号保存在 HANDLE 中，接着调用功能3FH 读取这个文件。

因为建立文件时，每个记录后面都跟着成对的回车/换行符，所以本程序在显示记录时不必移动光标。

```
TITLE HANREAD. EXE
; —— Read disk records created by hancreat
;-------------------------------------------
stack segment para stack 'stack'
        dw      80 dup(?)
stack ends
;-------------------------------------------
dseg    segment para 'data'
endede  db      0
handle  dw      ?
ioarea  db      32 dup(' ')
pathnam db      'A:/NAME. DAT',0
openmsg db      '* * * open error * * *',0dh,0ah
readmsg db      '* * * read error * * *',0dh,0ah
row     db      0
dseg    ends
;-------------------------------------------
cseg segment para 'code'
begin proc far
        assume cs:cseg,ds:dseg,ss:stack,es:dseg
        push    ds
        sub     ax,ax
        push    ax
        mov     ax,dseg
        mov     ds,ax
        mov     es,ax
        mov     ax,0600h
```

```
        call    scren           ;clear screen
        call    curs            ;set cursor
        call    openh           ;open file,set DTA
        cmp     endcde,0        ;valid open ?
        jnz     a1              ;no,terminate
contin: call    readh           ;read disk record
        cmp     endcde,0        ;normal read ?
        jnz     a1
        call    disph           ;yes,display name
        jmp     contin          ;continue
a1:     ret                     ;terminate
begin endp
;       open file
;-----------------------------------------------
openh proc near
        mov     ah,3dh          ;request open
        mov     cx,0
        lea     dx,pathnam
        int     21h
        jc      b1              ;error ?
        mov     handle,ax       ;no,save handle
        ret
b1:     mov     endcde,01       ;yes
        lea     dx,openmsg      ;error message
        call    errm
        ret
openh endp
;       read disk record
;-----------------------------------------------
readh proc near
        mov     ah,3fh          ;request read
        mov     bx,handle
        mov     cx,32           ;for name and CR/LF
        lea     dx,ioarea
        int     21h
        jc      c1              ;error on read ?
        cmp     ax,0            ;end of file ?
        je      c2
        cmp     ioarea,1ah      ;EOF marker ?
        je      c2
        ret
c1:     lea     dx,readmsg      ;no,invalid read
        call    errm
c2:     mov     endcde,01       ;force end
```

```
        ret
readh endp
;       display name
;-------------------------------------------------------------
disph proc near
        mov     ah,40h          ;request display
        mov     bx,01           ;set handle
        mov     cx,32           ;end length
        lea     dx,ioarea
        int     21h
        cmp     row,24          ;bottom of screen ?
        jae     d1              ;yes,bypass
        inc     row
        ret
d1:     mov     ax,0601h
        call    scren           ;scroll
        call    curs            ;set cursor
        ret
disph endp
;       scroll screen
;-------------------------------------------------------------
scren proc near                 ;AX set on entry
        mov     bh,1eh          ;set color
        mov     cx,0
        mov     dx,184fh        ;request scroll
        int     10h
        ret
scren endp
;       set cursor
;-------------------------------------------------------------
curs proc near
        mov     ah,2            ;request set cursor
        mov     bh,0
        mov     dh,row          ;row
        mov     dl,0            ;column
        int     10h
        ret
curs endp
;       disk error routine
;-------------------------------------------------------------
errm proc near
        mov     ah,40h          ;DX contains address
        mov     bx,01           ;handle
        mov     cx,20           ;length of message
```

```
        int     21h
        ret
errin endp
;-------------------------------------------
cseg ends
    end     begin
```

图 12.7 HANREAD 程序

12.2.6 移动读写指针

利用文件代号存取文件是以字节为存取单位的。一个文件被看作由许多字节组成，而不象顺序存取方式和随机(分块)存取方式是以记录为存取单位。但是用文件代号存取磁盘文件，尽管每次读写的字节数可任意指定，但一般还是为输入/输出缓冲区的大小所限制。所以一个比较大的文件总是要分几次读写。每次读写的字节我们也习惯地称为记录。另外象名单、零件表、账目等，每项具有同类型数据和同样格式的文件，按记录来进行处理更为方便。那么，在读写文件时，每次读写的记录是如何"拼接"起来的呢？原来操作系统对文件保存了一个称为读写指针(read/write pointer)的变量，由它确定应从文件的什么地方读出，或应往文件的什么地方写入。

为了存取文件中间某一特定的记录，首先要使读写指针指向这个记录。DOS 提供了移动读写指针功能42H，要求在 BX 中指定文件代号，由 AL 中的代码确定改变指针的三种方式。在每种方式中，由 CX 和 DX 指定一个双字长的偏移值，低位字在 DX 中，高位字在 CX 中，这个偏移值是一个带符号的整数，它可以是正数，也可以是负数。

绝对移动方式(AL=0)，偏移值从文件首开始计算。例如偏移值是182，则读写指针指向文件的第182字节。为了使指针指向文件首，可以在 CX、DX、AL 中都送入0，那么下面的读写就从文件首开始。

相对移动方式(AL=1)，当前的指针值加上偏移值作为新的指针值，也就是说，偏移值指出了从当前的读写位置起移动的字节数。根据偏移值的正负可正向或反向移动指针。

绝对倒移方式(AL=2)，新的指针位置通过把偏移值和文件尾的位置相加而确定。如果文件的记录长度是32字节，那么在 AL 中送入2，DX 中送入-32，CX 中送入0FFFFH(负号扩展到高位字)，则读写指针指向文件的最后一个记录。如果 CX 和 DX 为0，AL 为2，则指针将指向文件尾，以后的读操作读出的就是后面的新记录。

移动读写指针功能可能出现的错误码是01和06，错误码01说明 AL 中的方式值是不合法的，错误码06说明 BX 中的文件代号不合法。

如果指针移动成功，AX 和 DX 将是移动后的指针值，AX 中是低位字，DX 是高位字(调用之前，DX 是偏移值的低位字，CX 是偏移值的高位字)。

方式1和偏移值0能找出指针的当前值。方式2和偏移值0能找出文件的长度。

如果要在一个已存在的文件后面添加记录，则在写之前把指针指向文件尾：

```
MOV   AH,42H          ;request move pointer
MOV   AL,2            ;method 2
MOV   BX,HANDLE       ;handle
MOV   CX,0            ;offset
MOV   DX,0
```

```
        INT     21H             ;call DOS
        JC      ERROR           ;check ERROR
        ⋮
```

如果想从当前的指针位置向前或向后移N个字节，则可用方式1。假定偏移值N的范围在-32768到32767之间。

```
        MOV     BX,HANDLE       ;file handle
        MOV     CX,0            ;high_order word
        MOV     DX,N            ;low_order word
        CMP     DX,0            ;if N>0
        JGE     POINT           ;yes
        NOT     CX              ;extending the sign
POINT:  MOV     AL,1            ; method1
        MOV     AH,42H          ;request move pointer
        INT     21H             ;call DOS
        JC      ERROR           ;check error
        ⋮
```

这个功能使用起来很简单，一般程序中在打开文件之后，使用这个功能的某种方式将读写指针移到文件中需要的位置，以后的读写就从文件的这个地方开始，从而提供了在文件中随机存取的能力。

12.3 字符设备的文件代号式I/O

在前一节的程序例子中我们已使用了键盘、显示器等设备的文件代号式输入/输出功能，这是DOS 2.0为用户提供的另外一组独立于硬件的DOS功能。常用字符设备的文件代号都是由DOS预先定义好的。当一个用户程序得到控制权后，它就得到了五个已打开的文件代号，这五个文件代号是：

0000 输入设备，通常是键盘。

0001 输出设备，通常是显示器

0002 错误输出设备，总是显示器

0003 辅助设备，一般为通讯设备

0004 标准打印机(#0打印机)

设备和文件代号建立了对应关系，用户就可将这些设备视为文件。前两种设备的I/O功能，允许改向操作，比如，允许用户程序可以从键盘文件输入也可以从别的文件输入，可以向显示器文件输出也可以向别的文件输出，而且不必打开或关闭这些文件，这些设备文件也没有读写指针。键盘输入和显示器，打印机输出等设备使用代号式I/O是非常简单的。

例如，从键盘输入一行字符的指令序列为：

```
        MOV     AH,3FH          ;request read func
        MOV     BX,0            ;handle 0=keyboard
        MOV     CX,80           ;number of bytes
        MOV     DX,SEG BUFFER   ;addr of input area
        MOV     DS,DX
```

```
        MOV     DX,OFFSET BUFFER
        INT     21H                     ;call DOS
        JC      ERROR
        ⋮
BUFFER DB 80 DUP(' ')
```

从键盘实际读入的字符数返回到 AX 寄存器。

代号1和2可用于传送字符串并完成显示。例如可用功能40H(写文件或设备)将"hello"写到屏幕上。程序如下:

```
        MOV     AH,40H                  ;request write func
        MOV     BX,01                   ;handle 1=CRT
        MOV     CX,5                    ;length of string
        MOV     DX,SEG BUFFER           ;DS:DX =addr of string
        MOV     DS,DX
        MOV     DX,OFFSET BUFFER
        INT     21H                     ;call DOS
BUFFER DB 'hello'
```

该功能调用返回时,AX 中包含实际写的字符数。除去输出被改向为写磁盘文件而磁盘溢出的情况外,实际所写字节数应和要求写的字节数相等。

显示输出还可用错误输出设备(文件代号02),该文件代号总是指向控制台设备,并且它是不可改向的,所以使用02文件代号,一定是输出到了显示器,不可能输出到其它设备文件。

使用代号式 I/O 来完成打印输出比其它控制方式更为简单。打印机预先定义的文件代号为0004。用下列指令可将字符串'hello'输出到打印机:

```
        MOV     AH,40H                  ;request write func
        MOV     BX,04                   ;handle 04=printer
        MOV     CX,5                    ;number of string
        MOV     DX,SEG BUFFER           ;addr of output area
        MOV     DS,DX
        MOV     DX,OFFSET BUFFER
        INT     21H                     ; call DOS
        JC      ERROR                   ;CF=1 if error
        ⋮
BUFFER DB 'hello'
```

打印机输出功能返回时,AX 中包含实际写到打印机的字符数,正常情况下,AX 中的值应和所需写的字符长度一样,而且表示出错的 CF 标志位为0。但若输出数据中有文件结束标志(ctrl_2)时,输出结束,返回的 CF 位置1,表明程序有致命错误或是破坏了操作系统。

采用代号式 I/O,可以分别写几个列表设备(如 LPT1,LPT2),方法是由 INT 21H 的3DH 功能打开指定的设备,然后根据返回的文件代号用写功能40H 存取某一指定的打印机。

和其它字符设备一样,串行口通讯也可以通过文件代号式读写功能实现。作为辅助设备的串行接口,其预定义的代号为0003H。下面的程序可将 'hello' 写到串行口:

```
        MOV     AH,40H                  ; request write
        MOV     BX,03                   ;03=communication
        MOV     CX,5                    ;number of string
```

```
MOV   DX,SEG BUFFER        ;addr of output area
MOV   DS,DX
MOV   DX,OFFSET BUFFER
INT   21H                  ;call DOS
JC    ERROR                ;CF=1 if error
 ⋮
BUFFER DB 'hello'
```

如果系统中有多个串行口或辅助设备，仍要通过打开文件功能(3DH)使设备和文件代号建立关系，然后根据返回的文件代号去完成读写功能。

从以上的介绍可以看出，使用文件代号式 I/O，其优点是对不同设备并不考虑它们差异很大的硬件接口特性，只要指出不同的文件代号，在输入设备上读文件，在输出设备上写文件，其程序基本都是相同的，而且对错误的判断也有一致的标志(CF=1)，所以使用起来相当简便。如果用户不愿使用预定的标准设备，可先关闭这些文件代号(调用 INT 21H 的3EH 功能)，这样就为另一些文件或设备释放了文件代号。

下面我们再编写一个较为完整的程序例子。

例12.7 利用文件代号式 I/O，从键盘上输入文件并从打印机输出。

```
TITLE TYPER. EXE —— Using file handles I/O
;————————————————————————————
keyboard equ 0
crt      equ 1
printer  equ 4
;————————————————————————————
stack segment stack 'stack'
      db    64 dup(' ')
stack ends
;————————————————————————————
dseg segment 'data'
typebuff  db  130 dup(' ')
crrmsg    db  '* * * error * * *',0dh,0ah
dseg ends
;————————————————————————————
cseg segment 'code'
main  proc far
      assume cs:cseg,ds:dseg,ss:stack,es:dseg
      push    ds
      sub     ax,ax
      push    ax
      mov     ax,dseg
      mov     ds,ax
      mov     es,ax
      sti
      cld
      mov     ah,0          ;initializing printer
```

```
        mov     dx,0
        int     17h
input:  mov     bx,keyboard     ;handle 0
        mov     ah,3fh          ;request read func.
        mov     cx,130          ;number of bytes
        lea     dx,typebuff     ;addr of input area
        int     21h             ;call DOS
        jc      error
        cmp     typebuff,0
        je      exit            ;end of kbd input
output: mov     cx,ax           ;number of char.
        mov     bx,printer      ;handle 4
        mov     ah,40h          ;request write func
        int     21h             ;call DOS
        jc      error
        jmp     input
error:  mov     bx,crt          ;handle 1
        mov     ah,40h          ;output function
        mov     cx,15           ;length of string
        lea     dx,errmsg       ;addr of string
        int     21h             ;call DOS
exit:   ret
main    endp
cseg    ends
;-------------------------------------------------
        end main
```

图 12.8 TYPER 程序

12.4 BIOS 磁盘存取功能

我们可以直接在 BIOS 一级上编写磁盘处理程序，但 BIOS 不能像 DOS 功能那样自动地支持目录处理，文件结束操作和记录分块。BIOS 磁盘操作 INT 13H 处理的记录都是一个扇区的大小，都是以实际的磁道号和扇区号来寻址的。

读、写和检验磁盘文件之前，先把下列寄存器初始化：

AH　要执行的操作：读、写、检验或格式化

AL　扇区数

CH　磁道号

CL　起始的扇区号

DH　磁头号，对软盘是0或1

DL　驱动器号：0= 驱动器 A，1=驱动器 B，以此类推

ES:BX　数据区中 I/O 缓冲区的地址(除检验操作外)

12.4.1 BIOS 磁盘操作

BIOS INT 13H 要求在 AH 寄存器中指定操作。

AH=00 复位软盘系统

这个操作执行对软盘控制器的硬件复位。如果在其它磁盘操作之后调用这个功能，则返回一系列错误。

AH=01 读取软盘状态

这个操作在 AL 中返回最后一次软盘 I/O 操作之后的状态。

AH=02 读磁盘

这个操作把同一磁道上的若干个扇区中的数据读取到内存。扇区数一般是1、8或9。BX 中装入输入数据区的内存地址，但是注意 BX 中的偏移地址在附加段(ES)，这样输入数据区的地址应是 ES:BX。下面的例子把一个扇区的内容读入数据区 INSECT，这个数据区应足够大以接收一个扇区的所有内容。

```
MOV     AH,02          ;request read
MOV     AL,01          ;one sector
LEA     BX,INSECT      ;input buffer at ES:BX
MOV     CH,05          ;track 05
MOV     CL,03          ;sector 03
MOV     DH,00          ;head 00
MOV     DL,01          ;drive 01 (B)
INT     13H            ;call BIOS
```

调用返回时，AL 中是实际读取的扇区数，DS，BX，CX，和 DX 寄存器的内容不变。

在大多数情况下，程序只指定读一个扇区或一个磁道上的全部扇区。读操作顺序读取 CH 和 CL 指定的扇区内容，并递增 CH 和 CL 中的磁道号和扇区号。如果扇区号超过了磁道的最大扇区号，必须把扇区号重新置为01并把磁道号增1，或者把双面软盘的0面变为1面。

AL=03 写磁道

写磁盘操作把指定内存区中的数据写到一个扇区或几个扇区，这个内存区域一般为一个或几个512个字节。除了 AH=03外，其它寄存器的设置和读磁盘一样，调用返回时，AL 中是实际写入的扇区数，DS，BX，CX 和 DX 寄存器不变。

AL=04 检验磁盘扇区

这个操作只简单地检测指定的扇区是否能找到并且执行奇偶校验。有时为了更可靠的输出，可以在写操作之后，化费一些 I/O 时间来检验扇区的地址。这个操作因为没有数据传送所以不必设置 ES:BX。返回时，AL 中是实际检验的扇区数，DX，DS，BX 和 CX 寄存器不变。例如，判断在驱动器 A 中是否有格式化的盘时，用下面的指令序列：

```
MOV     AH ,04         ; request verify
MOV     DL,00          ;drive 0
MOV     DH,00          ;head 0
MOV     CH,00          ;track 0
MOV     CL,01          ;sector 1
MOV     AL,01          ;one sector
INT     13H            ;call BIOS
```

```
JC      ERROR           ;bad diskette
        ⋮
```

AH＝5 格式化盘磁道

用这个操作可以对一个或几个磁道进行格式化，IBM PC 对磁道格式化的标准大小是512。读写操作都要求装入一个指定扇区的格式化信息。格式化操作要求 ES:BX 寄存器指向一组磁道地址区，对磁道上的每一个扇区，必须有一个格式为 T/H/S/B 的四字节的数据项，这里

T＝磁道号

H＝磁头号

S＝扇区号

B＝每扇区的字节数(00＝128，01＝256，02＝512，03＝1024)

例如，把磁道3，磁头0格式化为每扇区512字节，那么1扇区的第一项内容就为03000102。

12.4.2 状态字节

对上面介绍的 BIOS 磁盘操作(AH＝02，03，04，05)如果操作成功，则 CF 和 AH 置为0；如果操作失败，CF 置为1，AH 中返回表示出错原因的状态代码。

AH	原　　因
01	给磁盘 I/O 传送了非法命令
02	磁盘上没有发现地址标记
03	试图往写保护盘上写
04	没有找到指定的扇区
08	DMA 超载运行
09	DMA 超过64K 的限制
10	读盘数据错(CRC)
20	软盘控制器出错
40	随机移动失败
80	回答失败

如果中断操作返回一个错误码，通常就把磁盘复位(AH＝00)，并且把这个操作重复执行三次。如果仍然得到一个错误码，则显示出错信息，以给用户掉换软盘的机会。

12.4.3 BIOS 磁盘操作举例

现在我们利用 BIOS 指令来编写一个读磁盘的程序 BIOREAD。这个程序和随机读程序(例12.3)的不同之处是：

1. 不用定义 FCB 和打开操作程序。

2. 程序要计算每一个磁盘地址，每次读操作之后，扇区号加1，当扇区号加到10，则重新置扇区号为01，如果盘面是1，则增加磁道号，然后改变盘面，或由0改变为1，或由1改变为0。

3. 数据项 CURADR 包含起始的磁道/扇面，ENDADR 包含结束的磁道/扇面，增强程序功能的一种方法是提示用户输入起始和结束的磁道/扇面地址。

```
TITLE BIOREAD.COM  -- Read disk sectors via BIOS

cseg segment para 'code'
```

```
        assume cs:cseg,ds:cseg,ss:cseg,es:cseg
        org 100h
begin:  jmp main
;-------------------------------------------------
recdin  db  512 dup(' ')        ;input area
endcde  db  0
curadr  dw  0304h               ;beginning track/sector
endadr  dw  0501h               ;ending track/sector
readmsg db  '* * * read error * * *$'
side    db  0
;-------------------------------------------------
main proc near
        mov     ax,cseg
        mov     ds,ax
        mov     es,ax
        mov     ax,0600h        ;request scroll
rept:   call    scren           ;clear screen
        call    curs            ;set cursor
        call    addrs           ;calculate disk addr.
        mov     cx,curadr
        mov     dx,endadr
        cmp     cx,dx           ;at end sector ?
        je      exit            ;yes,exit
        call    reads           ;read disk record
        cmp     endcde,0        ;normal read ?
        jnz     exit            ;no,exit
        call    disps           ;display sector
        jmp     rept            ;repeat
exit:   mov     ax,4c00h        ;terminate
        int     21h
main endp
;       calculate next disk address
;-------------------------------------------------
addrs proc near
        mov     cx,curadr       ;get track/sector
        cmp     cl,10           ;past last sector?
        jne     return          ;no,exit
        cmp     side,0          ;bypass if side 0
        je      chs
        inc     ch              ;increment track
chs:    xor     side,01         ;change side
        mov     cl,01           ;set sector to 1
        mov     curadr,cx
return: ret
```

```
addrs endp
;       read disk sector
;-------------------------------------------------
reads proc near
        mov     al,01           ;No. of sector
        mov     ah,02           ;request read
        lea     bx,reedin       ;addr of buffer
        mov     cx,curadr       ;track/sector
        mov     dh,side         ;side
        mov     dl,01           ;drive B
        int     13h
        cmp     ah,0            ;normal read ?
        jz      incrt           ;yes,exit
        mov     endcde,01       ;no
        call    errm            ;invalid read
incrt:  inc     curadr          ;increment sector
        ret
reads endp
;       display sector
;-------------------------------------------------
disps proc near
        mov     ah,40h          ;request display
        mov     bx,01           ;handle
        mov     cx,512          ;length
        lea     dx,reedin
        int     21h
        ret
disp sendp
;       clear screen
;-------------------------------------------------
scren proc near
        mov     ax,0600h        ;full screen
        mov     bh,1eh          ;set color
        mov     cx,0            ;request scroll
        mov     dx,184fh
        int     10h
        ret
scren endp
;       set cursor
;-------------------------------------------------
curs proc near
        mov     ah,02           ;request set cursor
        mov     bh,0
        mov     dx,0
```

```
        int     10h
        ret
curs endp
;       disk error routine
;-------------------------------------------------
errm proc near
        mov     ah,40h          ;request display
        mov     bx,01           ;handle
        mov     cx,18           ;length of message
        lea     dx,readmsg
        int     21h
        ret
errm    endp
;-------------------------------------------------
cseg    ends
        end     begin
```

图 12.9 BIOREAD 程序

建议在 DEBUG 下运行这个程序，用 G 命令执行这个程序并观察输入缓冲区的内容。

值得注意的是，如果用 DOS 建立了一个文件，那么这个文件可能插在几个扇区中，它们在磁盘上的位置不一定是相邻的，所以不能用 BIOS INT 13H 把这个文件顺序地读出来。

习 题

12.1 用 DOS 功能调用下面的操作：

(1) 建文件　　(2) 设置 DTA

(3) 顺序写　　(4) 打开文件

(5) 顺序读

12.2 编写建立并写入磁盘文件的程序，这个磁盘文件包括零件号(5个字符)，零件名称(12个字符)和单价(1个字)，程序允许用户从键盘输入这些值。(提示，单价应转换为二进制数)。

12.3 写一个程序读出并显示12.2题建立起来的文件。

12.4 写出随机记录号2652(10进制数)填入 FCB 的格式。

12.5 写出以下操作的功能调用：

(1) 随机写　　(2) 随机读

(3) 随机分块写　　(4) 随机分块读

12.6 写出确定文件记录数的指令，假定打开文件操作已经执行，FCB 中文件大小域为 FCBFLSZ，记录大小域为 FCBRSZ。

12.7 利用12.2题建立起来的文件，编写一个随机分块读程序，并按下面的格式在屏幕上显示出每个记录：

part #	price	Dscription
023	00315	assemblers
024	00430	linkages
027	00525	compilers
049	00920	compressors
114	11250	extractors

117	00630	haulers	
122	10520	lifters	
124	21335	processors	
127	00960	labtiers	
232	05635	bailers	
999	000		

12.8 “文件未发现”和“非法文件代号”的错误返回代码是什么？

12.9 写出一个名为 PATH1的 ASCIZ 串，磁盘驱动器为 C，文件名为 CUST. LST。

12.10 为12.9题的文件编写指令，要求：

(1) 为存放文件代号定义一个数据项 CUSTHAN。

(2) 建立这个文件。

(3) 从 CUSTOUT 缓冲区(128字节)，利用文件代号写一个记录。

(4) 关闭此文件，测试错误。

12.11 对于12.10题的磁盘文件，编写程序：

(1) 打开文件。

(2) 把记录读入 CUSTIN，测试错误。

12.12 在什么环境下，应当关闭只读文件？

第十三章　模块化程序设计

13.1　汇编程序概述

汇编程序的任务是把汇编语言源程序模块转换为二进制的目标模块。其输入是源文件(ASM)，而主要输出是 OBJ 文件和 LST 文件。汇编程序把源文件转换为目标文件的过程需要对源文件经过两遍扫视。第一遍扫视要确定源程序每一行的偏移地址，第一遍扫视后应提供一张符号表(或称标识符表)，它把源程序所定义符号的偏移地址记录下来。第二遍扫视则产生所要求的 OBJ、LST 和 CREF 文件。本节将简单介绍汇编程序的两遍扫视过程。

13.1.1　汇编程序的主要工具

汇编程序在两遍扫视的过程中用到的主要工具有：

13.1.1.1　地址计数器(Location counter)

又称单元计数器或位置计数器。当开始汇编或在每一段开始时，把地址计数器初始化为零，以后在汇编程序扫视源文件的过程中，每处理一条指令，地址计数器就增加一个值，此值为该指令所需要的字节数。所以，在汇编过程中，对于被汇编的每个段来说，地址计数器可以看作是动态地指向被汇编指令的相对位置的一个指针，也就是说，在汇编过程中，地址计数器的内容即当前正被汇编指令的偏移地址。

例 13.1　如图 13.1 所示。其中左部为源程序的一段，右部为汇编每个语句时地址计数器的值以及以字节为单位的语句长度。其中 ORG 伪操作使地址计数器置成其后表达式所规定的值。可以看出，在汇编的第一遍扫视的过程中，地址计数器的值可以用来确定每条指令的第一个字的偏移地址及数据段中变量名的值，这样就可以建立一张符号表。

```
                                        (location   length
;* * * * * * * * * * * * * * * * * *     counter) (byte)
data_seg        segment                   0          0
        org     10                        0          10
  num   db      -29                       10(0ah)    1
  array db      100  dup(?)               11(0bh)    100
  count dw      5                         111(6fh)   2
  masks db      82h,04h,2ah               113(71h)   3
data_seg        end                       116(74h)
;* * * * * * * * * * * * * * * * * *
code_seg        segment                   0          0
main    proc    far                       0          0
        assume  cs : code_seg,ds : data_seg 0        0
start : mov     ax,data_seg               0          3
        mov     ds,ax                     3(03h)     2
```

```
        mov     cx,count            5(05h)     4
repeat: dec     cx                  9(09h)     1
                                   10(0ah)
         :
        ret
main    endp
code_seg ends
;* * * * * * * * * * * * * * * * *
        end     start
```

图 13.1　例 13.1 的源程序及与其相应的地址计数器的值

13.1.1.2　符号表

在汇编程序对源程序第一遍扫视的过程中就建立了符号表，它把用户所定义的符号赋予当前地址计数器的值。在 LST 清单的后部，我们可以看到源文件汇编后的符号表。例 13.1 的符号表如图 13.2 所示。

```
Segments and groups:
            Name                    Size    align   combine class
CODE_SEG............                000B    PARA    NONE
DATA_SEG............                0074    PARA    NONE

Symbols:
            Name                    Type    Value   Attr
ARRAY..............                 L BYTE  000B    DATA_SEG
                          Length=0064
COUNT..............                 L WORD  006F    DATA_SEG
MAIN ..............                 F PROC  0000    CODE_SEG
                          Length=000B
MASKS..............                 L BYTE  0071    DATA_SEG
NUM................                 L BYTE  000A    DATA_SEG
REPEAT ............                 L NEAR  0009    CODE_SEG
START..............                 L NEAR  0000    CODE_SEG
```

图 13.2　例 13.1 的符号表

13.1.1.3　机器指令表

它是一张固定的表格，给出所有指令的助记符及与其对应的机器指令代码信息。

13.1.1.4　伪操作表

给出所有伪操作名及有关信息。

汇编程序在第一遍扫视后建立了符号表，在第二遍扫视的过程中，根据符号表、机器指令表和伪操作表把汇编语言指令翻译成机器语言指令，完成汇编任务。

13.1.2　汇编过程

两遍扫视的汇编过程如图 13.3 和 13.4 所示。这里所列出的只是汇编程序工作的最主要

部分，大部分伪操作及宏指令的处理过程以及大部分查错信息在这里均未表示出来，但这两张图已经清楚说明汇编程序所做的主要工作。

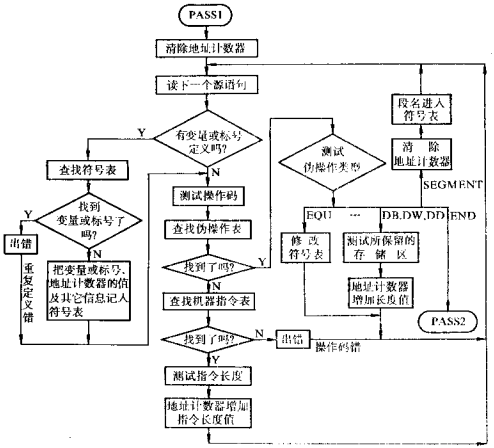


图 13.3 汇编程序第一遍扫描的主要流程图

13.1.3 几个问题

在了解了汇编的主要过程之后，有些问题还需要作进一步的说明。

13.1.3.1 有关"向前引用"问题

在汇编语句的操作数字段中出现变量或标号时，有两种不同的情况：如果出现在操作数字段的变量或标号是已经定义过的，则称为向后引用，否则称为向前引用。

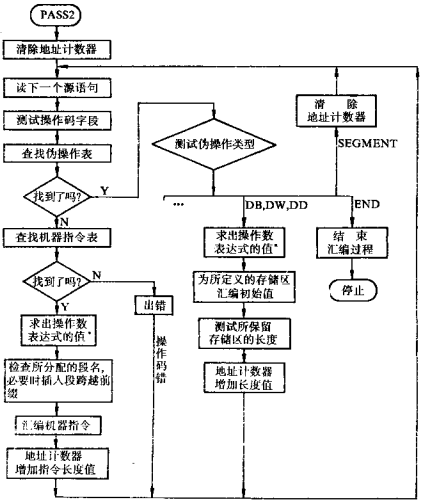
汇编程序在处理向前引用的情况时，有可能会发生困难，因为指令的长度与操作数类型有关，而操作数类型则与有关变量或标号的类型有关。在向后引用时这些都是确定值，但向前引用时就会发生困难了。例如，指令

```
JMP     EXIT
```

如 EXIT 是向前引用的，则汇编程序就不清楚它的类型属性应该是 SHORT、NEAR 还是 FAR，这样，指令长度就难于确定。汇编程序一般用 NEAR 属性作为缺省情况，当然，这不一定符合要求。因此，我们希望读者能注意，当你的程序中有类似的向前引用问题时，应该在程序中明确说明，如上例应写明

```
        JMP   SHORT EXIT
或      JMP   NEAR PTR EXIT
或      JMP   FAR PTR EXIT
```

以免发生错误。



* 如有表达式时

图 13.4 汇编程序第二遍扫视的主要流程图

此外，由于同一个原因，数据段定义应该先于代码段的定义，以保证当 指令引用到一些变量时变量已有了定义。EQU 语句则经常放在数据段和代码段之间，也有时放在数据段之前。应该注意的是：如果 EQU 语句使用了一个表达式，而表达式中的某项是由另一个 EQU 语句定义的，则应该使用向后引用。如：

```
I=0
I=I+1
```

13.1.3.2 “浮动”地址概念

我们知道，在汇编过程中每个段开始时地址计数器的值都置为零，因此，其后本段内所有偏移地址均为相对于起始的零地址而言的相对地址。在程序装入存储器时，并不是所有段都从物理的 0 地址开始的。实际上，由于物理的 0 地址区是由系统占用的，因而用户程序的各段都起始于某个非零地址，也就是说，段起始地址要在 0 地址的基础上“浮动”一个值，而此值要在连接时才能确定。从这个意义上说，变量和标号都是浮动地址，如果指令的操作数字段涉及到变量或标号，那末由汇编程序确定的指令字中的值为浮动值。在 LST 清单中，汇编程序在这样的二进制指令后记以 R。在图 13.5 所示的例 13.1 的 LST 清单中，由于 data_seg 是段地址，这是汇编程序不能确定的，因而，指令

```
mov  ax,data_seg
```

的机器语言值不定，后面记以符号 R。而

```
mov  cx,count
```

指令的机器语言虽然已有了一个值，但因 count 是浮动地址，因而在机器指令后仍记以符号 R，以表示这一指令中的地址是浮动地址。

读者也许会问，既然偏移地址是相对于段地址而言的相对地址，那么重新分配段地址值对偏移地址本身会有什么影响呢？确实，就一个程序模块本身来说，它确实没有什么影响。但是，在下一节里我们将介绍多个程序模块相连接的情况，在某些情况下，当连接程序把几个模块中的某些段连接在一起形成一个段时，地址的浮动就会直接影响机器指令代码，在这种情况下，连接程序将会修改汇编程序标以 R 的地址值以得到正确的机器指令。

```
                        ;* * * * * * * * * * * * * * * * * *
0000                    data_seg        segment
000A                            org     10
000A  E3                  num   db      -29
000B    64 [              array db      100  dup(?)
             ??
               ]


006F  0005                count dw      5
0071  82 04 2A            masks db      82h,04h,2ah
0074                    data_seg        ends
                        ;* * * * * * * * * * * * * * * * * *
0000                    code_seg        segment
0000                    main    proc    far
                                assume  cs : code_seg,ds : data_seg
0000  B8____ R          start : mov     ax,data_seg
0003  8E D8                     mov     ds,ax
0005  8B 0E 006F R              mov     cx,count
0009  49                repeat : dec    cx
                        ;       .
                        ;       .
```

```
                        ;          .
000A   CB                          ret
000B                    main       endp
000B                    code_seg              ends
                        ;* * * * * * * * * * * * * * * * * *
                                   end        start
```

图 13.5　例 13.1 的 LST 清单

13.2　连接程序及连接对程序设计的要求

13.2.1　连接程序的主要功能

经过汇编程序处理而产生的目标模块 OBJ 文件已经是二进制文件了，但它还不能直接上机运行，因为它还有两个问题并未得到解决。一个问题是目标模块中还有浮动地址，它必须在连接时才能确定下来。连接程序的这一工作称为再定位。另一个问题是，连接程序的更重要的作用是可以把多个程序模块连接起来形成一个装入模块，在这种情况下，每个程序模块中可能有一些外部符号(下节将专门说明)的值是汇编程序无法确定的，必须由连接程序来确定。因此连接程序的主要工作是：

1) 找到要连接的所有目标模块；

2) 对所有要连接的目标模块中的所有段分配存储单元，即确定所有段地址值；

3) 确定所有汇编程序所不能确定的偏移地址值(包括浮动地址及外部符号所对应的地址)；

4) 构成装入模块，并把它装入存储器。

在多个模块相连接时，各模块的连接次序是由用户在调用连接程序时指定的，调用方式可以是：

```
c>link↙
Object Modeles [.OBJ]：文件名[+文件名+……]↙
Run File [filename.EXE]：[文件名]↙
List File [NUL.MAP]：[文件名]↙
Labraries [.LIB]：[+]…]↙
```

连接程序就按目标模块行中用户所键入文件名的次序来实行连接。装入模块即可执行的 EXE 文件，对于这一文件，用户可以指定文件名，如不指定则连接程序就用第一个目标模块名作为装入模块名。另外，在多个目标模块相连接的情况下，只有主模块在模块结束的 END 伪操作后可以带有启动地址，如 END　　START 等，而其它模块只能用 END 结尾，不应该再带有任何启动地址。

13.2.2　连接对程序设计的要求

在模块化程序中，主程序以及多个子程序可以编制成不同的程序模块，各个模块在明确各自的功能和相互之间的连接约定以后，就可以独立编写并调试，各模块调试完后，再把它们连接起来形成一个完整的程序，这样就产生了怎么处理各模块之间的连接问题。例如，各个程序模块有各自的代码段和数据段，在模块相连接时它们是否需要分别连接在一起，又如何相连

接？又如，各程序模块之间会有一些共同使用的符号以及信息的传送应该怎么处理呢？下面我们来讨论这些问题。

13.2.2.1 多个模块组合时的连接情况

多个程序模块相连接时，并不一定要把所有的代码段或数据段分别连接在一起形成一个大的代码段或数据段。在很多情况下，各程序模块仍然有各自的分段，只是通过模块之间的调用来进行工作。但有时有些程序模块需要连接在同一段内，SEGMENT 伪操作的组合类型提供了这种可能性。在第四章里，我们已经介绍了 SEGMENT 的组合类型及类别的用法。在这里，我们再进一步说明其中的几种类型。

PUBLIC 可以把不同模块中的同名段在装入模块中连接而形成一个段，它们的连接次序按用户在调用 LINK 程序时指定的次序排列，每个段都从小段的边界开始，因此各模块原有的段之间可能存在小于 16 个字节的间隔。

COMMON 把不同模块中的同名段重叠而形成一个段。公共段的长度取决于各模块原有段中长度最大的一个，重叠部分的内容取决于排列在最后一段的内容。

STACK 把不同模块中的同名段组合而形成一个段，该段的长度为各原有段的总和，各原有段之间并无 PUBLIC 连接成段中的间隔，而且栈顶可自动指向连接后形成的大堆栈段的栈顶。

MEMORY 使该段放在装入模块的最高地区。如果连接时不止一个段有 MEMORY 组合类型说明，则遇到的第一个段作为 MEMORY 处理，其它段则作为 COMMON 处理。

下面举例说明组合类型的使用情况。

例 13.2 有两个源模块如图 13.6 所示。源模块经汇编和连接后形成的装入模块的存储区分配情况如图 13.7 所示，其中 data segment 为两个模块共用的数据区，它是两个模块数据段的覆盖区。code segment 则由两个模块的代码段连接而成。

例 13.3 两个源模块的堆栈段如图 13.8 所示。经汇编和连接后形成装入模块中的堆栈段如图 13.9 所示，可见各模块的堆栈段合并在一起且栈顶移至大堆栈段之前。

```
;       source module 1
data    segment common
        ⋮
data    ends
code    segment public
        ⋮
code    ends

;       source module 2
data    segment common
        ⋮
data    ends
code    segment public
        ⋮
code    ends
```

图 13.6 例 13.2 的源模块

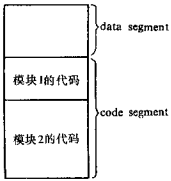


图 13.7　例 13.2 装入模块的存储区分配情况

```
;                source module 1
stack_seg        segments    stack
                 dw          20  dup(?)
  top_of_stack   label       word
stack_seg        ends
                 ⋮
;                source module 2
stack_seg        segment     stack
                 dw          30  dup (?)
stack_seg        ends
```

图 13.8　例13.3的源模块

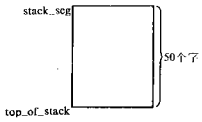


图13.9　例13.3连接后的堆栈段

装入模块中的各个源模块的排列情况可以在连接后的 MAP 文件中看到，这在第四章中已经介绍过，这里不再赘述。

此外，SEGMENT 伪操作中的类别说明并不能把相同类 别的段合并而形成一个段，但它能把同一类别的段的位置放在一起。看下面的例子。

例13.4 三个源模块如图13.10所示。对三个源模块分别汇编并连接后形成装入模块，图13.11是其MAP文件，可以看出，此时的装入模块是按连接时指定的源模块的次序排列而组成的。如果在上面的源模块中的代码段后均加上类别'code'，即源模块1的代码段为：

```
code segment  'code'
      ⋮
code  ends
```

源模块3中的代码段为：

```
code2 segment  'code'
      ⋮
code2  ends
```

则表示装入模块情况的MAP文件如图13.12所示，可见此时两个模块中的代码段并未合并成一个段，但在存储区中的位置却是靠在一起的了。

```
;              source module 1
stack_seg      segment  stack
                    dw        20     dup(?)
               top      label     word
stack_seg      ends
data           segment
  a            dw       65
  b            dw       78
data           ends
edata          segment
  c            dw       98
  d            dw       34
  e            dw       67
edata          ends
code           segment
               assume cs:code,ds:data,es:edata,ss:stack_seg
start:         push     ds
               sub      ax,ax
               push     ax
               ⋮
               ret
code           ends
               end      start


;              source module 2
stack_seg           segment stack
```

```
                    dw      20 dup(?)
                top label   word
stack_seg           ends
data2               segment
  x                 dw      9
  y                 dw      12
data2               ends
                    ⋮
                    end
;                   source module 3
stack_seg                   segment stack
                            dw      20 dup(?)
stack_seg                   ends
data3               segment
  r                 dw      76
  z                 dw      ?
data3               ends
code2               segment
                    assume cs:code2,ds:data3,ss:stack_seg
subroutine          proc far
                    push    ax
                    push    bx
                    push    cx
                    push    dx
                    ⋮
                    pop     dx
                    pop     cx
                    pop     bx
                    pop     ax
                    ret
subroutine          endp
code2               ends
                    end
```

图 13.10　例13.4的源模块

Start	Stop	Length	Name	Class
00000H	00004H	0005H	CODE	
00010H	00013H	0004H	DATA	
00020H	00025H	0006H	EDATA	
00030H	000A7H	0078H	STACK_SEG	
000B0H	000B3H	0004H	DATA2	
000C0H	000C8H	0009H	CODE2	
000D0H	000D3H	0004H	DATA3	

```
Origin     Group

Program entry point at 0000:0000
```

图 13.11　例13.4的 MAP 文件

Start	Stop	Length	Name	Class
00000H	00004H	0005H	CODE	CODE
00010H	00018H	0009H	CODE2	CODE
00020H	00023H	0004H	DATA	
00030H	00035H	0006H	EDATA	
00040H	000B7H	0078H	STACK_SEG	
000C0H	000C3H	0004H	DATA2	
000D0H	000D3H	0004H	DATA3	

```
Origin Group

Program entry point at 0000:0000
```

图 13.12　例13.4代码段加上同样的类别后形成的 MAP 文件

13.2.2.2　多个模块之间的变量传送问题

多个模块的程序相连接时，除段组合外还必然存在着变量传送问题，也就是过程通信问题。在第六章中，我们讨论过调用程序和子程序之间的变量传送问题，在这里我们要解决的是：如果调用程序和子程序不在同一个程序模块，那么怎样来进行变量传送呢？

1）外部符号

从连接的角度看，在源程序中用户定义的符号可以分为局部符号和外部符号两种。我们已经熟悉的在本模块中定义，又在本模块中引用的符号称为局部符号。而另一种在某一个模块中定义，而又在另一个模块中引用的符号称为外部符号。有两个伪操作与外部符号有关。

PUBLIC 伪操作，其格式是：

```
PUBLIC    symbol [,…]
```

在一个模块中定义的符号（包括变量、标号、过程名等）在提供给其它模块使用时必须要用 PUBLIC 定义该符号为外部符号。

EXTRN 伪操作，其格式是：

```
EXTRN   symbol name:type [,…]
```

在另一个模块中定义而要在本模块中使用的符号必须使用 EXTRN 伪操作。如符号为变量则类型应为 byte，word 或 dword；如符号为标号或过程名则类型应为 near 或 far。

有了这两个伪操作就提供了模块间相互访问的可能性。这两个伪操作的使用必须相匹配，连接程序的任务之一就是要检查每个模块中的 EXTRN 语句中的每个符号是否能和与其相连接的其它模块中的 PUBLIC 语句中的一个符号相匹配，如不相匹配则应给出出错信息，如相匹配就应给予确定值。下面的例子说明各模块中 PUBLIC 和 EXTRN 伪操作的匹配情况。

例13.5　三个源模块中的外部符号定义如图13.13所示，连接程序能检查出 var4是模块2需要使用的符号，但没有其它模块用 PUBLIC 来宣布其定义，因而连接将显示出错。在这个例子中，模块3用 PUBLIC 宣布了 lab3的外部定义，但其它模块均未使用该符号，这种不匹配情况由于不影响装入模块的建立，所以并不显示出错。此外，模块1和模块2都定义了局部符号 var3，由于局部符号是在汇编时就确定了其二进制值，所以并不影响模块的连接，因而不同模块中的

局部符号是允许重名的，但要连接模块的外部符号却不允许重名，如有重名，连接将显示出错。

```
;           source module 1
extrn       var2:word,lab2:far
public      var1,lab1
data1       segment
            var1    db      ?
            var3    dw      ?
            var4    dw      ?
data1       ends
code1       segment
            ⋮
lab1:
            ⋮
code1       ends
            ⋮
;           source module 2
extrn       var1:byte,var4:word
public      var2

data2       segment
            var2    dw      0
            var3    db      5    dup(?)
data2       ends
            ⋮
;           source module 3
extrn       lab1:far
public      lab2,lab3
            ⋮
lab2:
            ⋮
lab3:
            ⋮
```

图 13.13　例13.5的源模块

连接程序需要对目标模块作两遍扫视，第一遍扫视应对所有段分配段地址，并建立一张外部符号表(外部符号在汇编时是不可能确定其值的，LST 清单中对外部符号记以 E)；第二遍扫视才能把与这些外部符号有关指令的机器语言值确定下来。连接完成后建立了装入模块，再由装入程序把该模块装入内存等待执行。

2) 多个模块之间的变量传送方法

我们可以使用前面已经说明过的几种伪操作及其参量来解决变量传送问题。

例13.8　这一例子即第六章中的例6.4，这里说明当主程序和子程序不在同一模块时的变

量传送方法。图13.14给出了这一程序。

```
        ;       source module 1
extrn           proadd:far
;*********************
data            segment common          ;define data segment
  ary                   dw      100  dup(?)
  count                 dw      100
  sum                   dw      ?
data            ends
;*********************
code1           segment                 ;defint code segment
;-------------------
main            proc    far             ;main part of program
                assume cs:code1,ds:data
start:          push    ds              ;starting execution address
                sub     ax,ax
                push    ax
                mov     ax,data
                mov     ds,ax
                        :
                call    far ptr proadd
                        :
                ret
main            endp            ;end of main part of prog.
;-------------------
code1           ends
;*********************
                end     start


;               source  module 2
public          proadd
;*********************
data            segment common
  ary                   dw      100  dup(?)
  count                 dw      100
  sum                   dw      ?
data            ends
;*********************
code2           segment
proadd          proc    far             ;define subprocedure
                assume  cs:code2,ds:data
```

```
                mov     ax,data
                mov     ds,ax
;
                push    ax              ;save the registers
                push    cx
                push    si
                lea     si,ary          ;put addr of ary in si
                mov     cx,count        ;put count in cx
                xor     ax,ax           ;clear ax
next:           add     ax,[si]         ;add the elements of the
                add     si,2            ;  array to AX
                loop    next
                mov     sum,ax          ;store the result in sum
                pop     si              ;restore the registers
                pop     cx
                pop     ax
                ret                     ;return
proadd          endp                    ;end subprocedure
;-------------------------------------
code2           ends                    ;end of code segment
;*********************
                end                     ;end assembly
```

图 13.14 当主程序和子程序不在同一程序模块中时变量的传送方法之一

在这个例子中，data 段用 common 合并成为一个覆盖段，所以源模块2只引用了本模块中的变量，不必作特殊处理。整个程序的外部符号只有 proadd，处理比较简单。应该注意，由于主程序和子程序已经不在同一程序模块中，所以过程定义及调用都应该是 FAR 类型的，而不应该使用原来的 NEAR 类型。如果以上两个模块的 code 段都使用同一段名并加上 PUBLIC 说明，这样，连接时它们就可以合并为一个段，此时，过程和调用仍可使用 NEAR 属性。

使用公共数据段并不是唯一的办法，我们可以把变量也定义为外部符号，这样就允许其它模块引用在某一模块中定义的变量名。必须注意，我们在引用本模块中的局部变量前，在程序的一开始就用以下两条指令：

```
MOV   AX,DATA_SEG
MOV   DS,AX
```

把数据段地址放入 DS 寄存器中，这样才能保证对局部变量的正确引用。在引用外部符号时也必须把相应的段地址放入段寄存器中。如果程序中要访问的变量处于不同段时，就应动态地改变段寄存器的内容。让我们看下面的例子。

例13.7 有三个源模块如图13.15所示。其中模块1本身的局部变量都在 DS 段中，而外部变量则在 ES 段中，在程序中动态地改变 ES 寄存器的内容，以达到正确访问各外部变量的目的。如果源模块1本身使用 ES 段，或者外部变量较多，为避免动态改变段地址易产生的错误，也可以用下例所使用的方法。

```
;               source module 1
;*****************
```

```
extrn           var1:word,output:far
extrn           var2:word
public          exit
;* * * * * * * * * * * * * * * * * * *
local_data              segment
                var     dw      5
                        :
local_data              ends
;* * * * * * * * * * * * * * * * * * *
Code            segment                 ;define code segment
;-------------------------------
main            proc    far             ;main part of program
                assume cs:code,ds:local_data
start:                                  ;starting execution address
                push    ds              ;save old data segment
                sub     ax,ax           ;put zero in AX
                push    ax              ;save it on stack

                mov     ax,local_data   ;data segment addr
                mov     ds,ax           ; into DS register
                        :
                mov     bx,var
                mov     ax,seg var1
                mov     es,ax
                add     bx,es:var1
                        :
                mov     ax,seg var2
                mov     es,ax
                sub     es:var2,50
                        :
                jmp     output
                        :
exit:           ret

main            endp                    ;end of main part of program
;-------------------------------
code            ends                    ;end of code segment
;* * * * * * * * * * * * * * * * * * *
                end     start           ;end assembly

;               soure module 2
;* * * * * * * * * * * * * * * * * * *
```

```
public              var1
;* * * * * * * * * * * * * * * * *
extdata1                segment         ;define data segment
  var1              dw      10
                    :
extdata1                ends
;* * * * * * * * * * * * * * * * *
                    :
;* * * * * * * * * * * * * * * * *
                    end                 ;end assembly

;                   source module 3
;* * * * * * * * * * * * * * * * *
public              var2
extrn               exit:far
;* * * * * * * * * * * * * * * * *
extdata2                segment         ;define data segment
  var2                  dw      3
                        :
extdata2                ends
;* * * * * * * * * * * * * * * * *
public              output
;
prognam             segment             ;defint code segment
                    assume  cs:prognam,ds:extdata2
                            :
output:
                    jmp     exit
                            :
prognam             ends                ;end of code segment
;* * * * * * * * * * * * * * * * *
                    end
```

图 13.15　当主程序和子程序不在同一程序模块中时变量的传送方法之二

例13.8　有两个源模块如图13.16所示。

```
;                   source module 1
;* * * * * * * * * * * * * * * * * * * * * * * *
global              segment public
  extrn             var1:word,var2:word
global              ends
;* * * * * * * * * * * * * * * * * * * * * * * *
local_data              segment         ;define data segment
                    :
local_data              ends
```

```
;* * * * * * * * * * * * * * * * * * * * * * * *
code        segment                 ;defint code segment
;-----------------------------------------------
main        proc    far             ;main part of program
            assume cs:code,ds:local_data,es:global
start:
            push    ds
            sub     ax,ax
            push    ax

            mov     ax,local_data
            mov     ds,ax
            mov     ax,global
            mov     es,ax
            :
            mov     bx,es:var1
            add     es:var2,bx
            :
            ret
main        endp
;-----------------------------------------------
code        ends
;* * * * * * * * * * * * * * * * * * * * * * * *
            end     start
;           source module 2
;* * * * * * * * * * * * * * * * * * * * * * * *
global      segment public
    public          var1,var2
    var1            dw      ?
    var2            dw      ?
            :
global      ends
;* * * * * * * * * * * * * * * * * * * * * * * *
            :
;* * * * * * * * * * * * * * * * * * * * * * * *
            end
```

图 13.16　当主程序和子程序不在同一程序模块中时变量的传送方法之三

从以上几个例子可以看出，在掌握了有关外部符号的伪操作及 SEGMENT 伪操作的参数使用方法的情况下，读者可以灵活使用这些工具以编制出较好的程序模块来。

13.3 汇编语言程序与高级语言程序的连接

由于编写及调试汇编语言程序比高级语言复杂，所以高级语言比汇编语言的使用更为广泛。但是，汇编语言又有它自己的特点：占有的存储空间小，运行速度快，有直接控制硬件的能力等，因而在有些场合汇编语言是不可缺少的。经常会有这种情况：程序的大部分是用高级语言编写的，但在某些部分，如程序的关键部分，或是运行次数很多的部分，或是运行速度要求很高的部分，或是直接访问计算机硬件的部分等，则需要用汇编语言编写。这样就提出了汇编语言与高级语言的连接问题。在这一节里，我们以 PASCAL 和 BASIC 语言为例说明汇编语言和编译性语言及解释性语言的连接方法。实际上，不同的语言或同一种语言的不同版本在具体规定上会有所差别，但所要解决的问题却是相同的，所以这里只介绍一般方法，使用时请读者阅读有关手册。一般说来，连接中要解决以下三个问题。

1) 存储器分配问题

一般高级语言经编译后产生 OBJ 文件，而汇编语言经汇编后也产生 OBJ 文件，然后由连接程序把它们连接而形成可执行文件 EXE，并把它装入内存等待执行。因此存储器的分配是由连接程序解决的，不必由用户考虑。

对于使用解释程序的 BASIC 语言情况就要复杂一些，必须设法找到汇编语言程序在存储器中存放的位置，并把有关信息传送给 BASIC 程序才行。

2) 两种语言之间的控制传送问题

汇编语言程序一般作为高级语言的外部过程，由高级语言通过函数或过程来调用汇编语言程序。

3) 变量传送问题

高级语言和汇编语言程序之间也必然存在变量传送问题，这些变量一般用数值或地址的形式来表示，连接时必须解决这一问题。

下面分别以 PASCAL 和 BASIC 语言为例具体说明它们与汇编语言程序的连接方法。

13.3.1 PASCAL 语言与汇编语言程序的连接

汇编语言程序是作为 PASCAL 语言程序的一个外部过程的，所以汇编语言程序的写法应该与一般外部调用的过程相同，汇编语言程序的一开始就应有

```
PUBLIC  Procedure name
```

在段定义时应加上 public 组合型说明。过程结束时应该用 ret 指令返回，而且由于采用堆栈来传送变元，因而返回时往往采用带常数的返回指令以便跳过参数区。

PASCAL 语言程序中除对汇编语言过程的调用外，还应解决过程定义问题。可使用

```
$ include:'name'
```

语句，其中 name 为文件名。名为 name 的 PASCAL 文件用来定义所要调用的过程名或函数名，在编译时，编译程序遇到 include 语句就可以对名为 name 的 PASCAL 文件进行编译。

PASCAL 程序调用汇编语言过程时，按照指定参数的次序把变量逐个存入堆栈。由于系统堆栈是以字为单位的，因而变量也以字为单位存入堆栈，最后再保存返回的段地址和偏移地址。在转入汇编语言程序后，应该使用 BP 寄存器来取得 PASCAL 程序在堆栈中存放的参数。这和第六章中用堆栈传送子程序参数的方法是相同的。下面我们用一个例子来说明汇编语言

程序与 PASCAL 语言程序的连接方法。

例13.9 要求用 PASCAL 语言编写一个能发出声音的程序，由于 PASCAL 语言不能直接控制发声，因此其中发声部分调用汇编语言程序来完成。为完成这一功能，共建立三个程序如图13.17所示。其 PORTIO.PAS 是一个 PASCAL 语言程序，用来定义过程及参数。这里定义了一个函数 Portin 和一个过程 Portout，在这个例子里只用上了 Portout，Portin 用来说明函数及其参数的定义方式。可以看出：所有参数都应说明其类型，而且应该用 external 来说明过程(或函数)是外部过程。

顺便说明一下，如果要求传送的是数据的地址而不是数据本身，则应该在过程或函数定义中的数据类型说明之前加上 VAR。如下面的语句将使 data1 字类型变量的地址传送到汇编语言程序中去。

```
Function Whatsis (var data1:word):byte;external;
```

BLAISER.PAS 是程序的主体部分，程序先显示 'Press enter to fire!'，用户按回车键后就发声，这一过程可以重复。程序中四次调用 Portout 汇编语言程序：第一次是打开扬声器，第四次是关闭扬声器，中间两次则控制发声。

```
{
* * * * * * * * * * * * * * * * * * * *
PORTIO.PAS              Pascal port I/O external declarations
* * * * * * * * * * * * * * * * * * * *
}
Function Portin (port_addr:word):byte; external;
Procedure Portout (port_addr:word;data:byte); external;
```



```
{
* * * * * * * * * * * * * * * * * * * *
BLAISER.PAS             Blaiser Blast sound demo
* * * * * * * * * * * * * * * * * * * *
}
Program Blaiser_blast (input, output);
  Const
        speaker          = #61;
        timer            = #62;
        toggle_on        = #4f;
        toggle_off       = #4d;
        max_count        =250;
        scaler           =2;
  Var
        code,
        count            :word;
{$include:'portio.pas'}
  Begin
    Repeat
```

```
        Write ('press enter to fire!');
        Readln;
        Portout (speaker, toggle_on);
    For count := 0 to max_count do
        [
        code := Sqr(count) Div scaler;
        Portout (timer, lobyte (code));
        Portout (timer, hibyte (code));
        ];
        Portout (speaker, toggle_off);
    Until count = #ffff;
  End.
```

(2)

```
;* * * * * * * * * * * * * * * * * * * * *
;PORTOUT. ASM——Routine to send data byte to I/O Port
;       to be used with pascal programs
;* * * * * * * * * * * * * * * * * * * * *
coder   segment byte public ;define segment
        assume cs:coder
;-----------------------------------
public  portout
portout proc  far           ;define procedure
        push  bp            ;save old BP
        mov   bp,sp         ;load current SP into BP
        mov   dx,[bp+8] ;put port address in DX
        mov   al,[bp+6] ;put data byte in AL
        out   dx,al         ;output the byte
        pop   bp            ;restore old BP
        ret   4             ;return to Pascal prog
portout endp                ;end procedure
;-----------------------------------
coder   ends                ;end segment
;* * * * * * * * * * * * * * * * * * * * *
        end                 ;end assembly
```

(3)

图 13.17 例13.9的程序实现

(1) PORTIO. PAS (2) BLAISER. PAS (3) PORTOUT. ASM

PORTOUT 是汇编语言程序，其中有关发声的部分在第十一章已经说明，这里应该注意两个程序之间的变量传送问题。前面已经说过 PASCAL 程序在调用汇编语言子程序时把参数顺序存入堆栈，本例中每次调用汇编语言程序后的堆栈状态如图13.18所示，所以在程序中用[bp

+8]取得端口地址，用[bp+6]取得数据字节，这种取得参数的方法与第六章中用堆栈传送参数时子程序取得参数的方法是一样的，这里不再赘述。

上机时，先分别建立包括(1)、(2)两部分的 PASCAL 文件 BLAISER. PAS 和汇编语言程序 PORTOUT. ASM 然后分别编译、汇编后形成两个 OBJ 文件，再用 LINK 程序把它们连接起来形成 BLAISER. EXE 文件，就可以在机器上执行了。具体操作过程如下：

```
C>  pas 1 blaiser
C>  pas 2 blaiser
C>  asm portout
C>  link blaiser+portout
C>  blaiser
```

屏幕上显示出 press enter to fire!，按回车键后就会发出一次呼啸声。

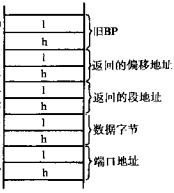


图13.18 例13.9调用 PORTOUT 时的堆栈状态

13.3.2 BASIC 语言与汇编语言程序的连接

由于 BASIC 语言是一种解释性语言，所以它与汇编语言程序的连接比 PASCAL 语言要复杂得多，一般可以有使用 USR 语句和 CALL 语句两种方法，下面分别加以说明。

13.3.2.1 用 USR 语句连接

使用 USR 语句来连接 BASIC 语言程序和汇编语言程序是一种常用的方式，但这种方式只允许传送及回送一个变量，这个变量存放在 BASIC 解释程序中的浮点累加器 FAC(Floating Point Accumulator)中，FAC 本来是为 BASIC 的浮点操作用的，在 BASIC 和汇编语言程序连接时用来作为传送变量的存放区域。FAC 是一个8字节的内存区域，它可以用来存放整型数(占有2个字节)、单精度浮点数(4个字节)和双精度浮点数(8个字节)。如果需要传送字符串，则因其长度不定而不通过 FAC 传送。在执行 BASIC 语言程序时把 FAC 的段地址放入 DS 寄存器，偏移地址放入 BX 寄存器，以便汇编语言程序取得变量。规定整型变量存放在 FAC 的第5个和第6个字节中，因而当变量为整型时，BASIC 直接把 FAC 的第5个偏移地址存放在 BX 寄存器中。为便于检查，对于所传送的变量规定一个类型值，由 BASIC 程序把它存放在 AL 寄存器中，汇编语言程序可以检查变量类型，变量与类型值的对应关系如表13.1所示。下面我们举例说明 BASIC 语言程序和汇编语言程序的连接。

表13.1 变量的类型值

变量类型	类型值
整型(2个字节)	2
字符串(3个字节的串描述符)	3
单精度浮点数(4个字节)	4
双精度浮点数(8个字节)	6

例13.10 DECIHEX 程序把从键盘输入的一个十进制数转换成二进制数，并在屏幕上以十六进制数的形式把它显示出来。这个程序的要求与例6.3完全相同，考虑到十进制数转换为二进制数的过程用 BASIC 语言实现的方便性，所以程序的前半部分用 BASIC 语言编写，而二进制数转换为十六进制数的部分，则由于 BASIC 语言不便于作位处理而用汇编语言编写，在

程序运行时把它们连接起来实现本题的要求。本例的程序实现如图13.19所示。

```
'    DECIHEX. BAS
10   DEFINT A--Z
20   DEF SEG=&H0
30   DEF USR0=0
40   INPUT "Decimal Number?",N
50   PRINT "Hex Equivalent is:"
60   D=USR0(N)
70   PRINT:PRINT
80   GOTO 40
```

(1)

```
; BINIHEX&.--Binary to hexadecimal converter
; Converts internal binary to hex on screen
; For use with BASIC programs

display equ     2h              ;video output function
doscall equ     21h             ;DOS interrupt number
; * * * * * * * * * * * * * * * * * * * * * *
binihex segment                 ;defint code segment
;-----------------------------------------------
main    proc    far             ;main part of program

        assume cs:binihex

; check that argment is really integer
        cmp     al,2            ; 2 is code for integer
        jne     exit            ;wrong argument
; get integer from Floating Point Accumulator
        mov     ax,[bx]         ;integer into AX
        mov     bx,ax           ; into BX for binihex
; print out binary number in BX on screen
        mov     ch,4            ;number of digits
rotate: mov     cl,4            ;set count to 4 bits
        rol     bx,cl           ;left digit to right
        mov     al,bl           ;move to AL
        and     al,0fh          ;mask off left digit
        add     al,30h          ;convert hex to ASCII
        cmp     al,3ah          ;is it>9?
        jl      printit         ;no,so 0 to 9 digit
        add     al,7h           ;yes,so A to F digit
printit:
        mov     dl,al           ;put ASCII char in DL
        mov     ah,display      ;display output funct
        int     doscall         ;call DOS
        dec     ch              ;done 4 digits?
        jnz     rotate          ;not yet
```

```
exit:
        ret                             ;return to DOS

main    endp                            ;end of main part of program
;-------------------------------------------------
binihex ends                            ;end of code segment
;*********************
        end     main                    ;end assembly
```

(2)

图 13.19 例13.10的程序实现

(1) BASIC 程序 DECIHEX.BAS (2) 汇编语言子程序 BINIHEX.ASM

DECIHEX.BAS 程序中，第10行定义变量 A~Z 为整型变量，亦即 N 定义为整型变量。第20和30行用来定义汇编语言子程序的段地址和偏移地址，在 BASIC 语言程序调用汇编语言子程序时，DS、ES 和 SS 寄存器自动设置成 BASIC 的数据空间地址，CS 寄存器则应置为由 DEF SEG 所指定的值，由于在程序编写时还不知道它们的值，所以在第20行和30行语句中都用0表示，下面我们还会介绍如何取得它们的正确值。第40行是 BASIC 的语句 INPUT，可以从键盘取得一个十进制数存放到变量 N 中。第50、60、70行用来完成把变量 N 中的二进制数以十六进制数的形式在屏幕上显示出来。其中第60行使用 USR 语句完成以变量 N 调用汇编语言子程序的任务。在这里，由于子程序并无回送给调用程序的值，因而 D 仅是由于格式上的需要而设的。

BINIHEX&.ASM 与例6.3中的 BINIHEX.ASM 的差别只是一进入子程序首先进行变量类型检查，即检查 AL 寄存器的内容，如发生类型错误则直接退出程序。然后，还需根据 BX 寄存器中的指针取出存放在 FAC 中的变量 N。应该注意，由于 BASIC 中的 USR 语句产生 FAR 调用，因而这里的过程必须定义为 FAR 类型的。

在程序编制完后，首先应确定在执行期间汇编语言子程序的存放区域。我们知道，BASIC 可以有64K 的工作区，如果机器的存储空间大于64K 则可把汇编语言程序存放在 BASIC 的工作区之外；但是如果机器的存储空间只有64K，那末，汇编语言程序就只能存放在 BASIC 的工作区之内了。如果要把汇编语言程序存放在 BASIC 的64K 工作区之内，则用户可以把汇编语言程序存放在 BASIC 工作区的最高地址区，为避免 BASIC 破坏这一区域，可以有两种办法来解决。

可以在 BASIC 语言程序内使用 CLEAR 语句，例如，如果需要在高地址区为汇编语言程序保留4K 的工作区则可用

```
10      CLEAR,&HF000
```

即限制 BASIC 的工作区为60K 字节。

或者，也可以在 DOS 系统下启动 BASIC 时使用选择项 M，如

```
C>BASIC/M:&HF000
```

可以达到同样的目的。

如果机器的存储空间大于64K，则可把汇编语言程序放在 BASIC 工作区之外，具体存放区域可以由 LINK 程序来确定。一般情况下，LINK 产生的 EXE 文件存放在存储器的最低可用空间，即常驻的操作系统之上，现在，这一空间已为 BASIC 程序所占用，因而我们可以使用调用 LINK 时的选择项 high，使汇编语言程序存放在可用的高地址区，即在 DOS 的转存部分之下。

在这种情况下，存储器空间的分配情况如图13.20所示。

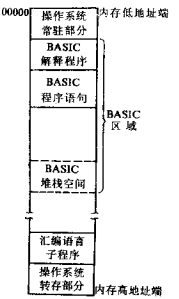


图 13.20 汇编语言程序存放在 BASIC 工作区以外时的存储空间分配情况

下面说明上机的运行步骤：

1）建立 BINIHEX&.. ASM 文件；

2）汇编该文件；

```
C>asm  binihex&.
```

3）用选择项 high 进行连接

```
C>link  binihex&./high
```

产生了 binihex&.. exe 文件；

4）用 DEBUG 装入 EXE 文件

```
C>debug binihex&.. exe
```

5）用 R 命令查看各寄存器的内容，以便确定该 EXE 文件在存储器中的存放区域。

```
C>debug binihex&.. exe
-r
AX=0000  BX=0000  CX=0025  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=18E4  ES=18E4  SS=9F94  CS=9F94  IP=0000  NV UP DI PL NZ NA PO NC
9F94:0000 3C02          CMP     AL,02
```

这样，我们就可以知道 binihex&. 程序的段地址是 CS 寄存器中的9f94，偏移地址是 IP 中的0。下面应该把这一信息传送给 BASIC 程序。

为了在 DEBUG 中装入 BASIC 程序，采用 N 和 L 命令，L(Load)命令可以把由 N(Name)命令指定的文件装入存储器中。N 命令则根据指定的文件名设置 FCB，以便下一步用 L 命令把该文件装入存储器，或用 W(Write)命令把存储器中的文件写入磁盘。因而

6）为装入 BASIC 程序使用 N 命令

```
—nbasic. com
```

7）用 L 命令装入 BASIC 程序

```
—l
```

运行 BASIC 程序

```
—g
```

这时屏幕应清除，然后显示 BASIC 准备好的信息及提示符 OK。为程序调试方便起见这里也可以设置断点，例如可以把断点设在汇编语言子程序的入口

```
—g 9f94:0
```

这样就可以在运行程序时停在汇编语言程序的入口，以便调试汇编语言程序。

9）装入（或键入）DECIHEX. BAS 程序

```
Ok
load "DECIHEX. BAS"
```

此时装入的 DECIHEX. BAS 程序即图13. 19(1)中的程序（或者，此时可全部键入程序），其中第20行和30行还没有确定的值，因此下面两步即设置此值。

10）把段地址放入 DEF SEG 语句中

```
20    DEF SEG =&H9f94
```

11）把偏移地址放入 DEF USR 语句中

```
30    DEF USR0=0
```

12）经修改后的 BASIC 语言程序应该重新保存起来，以免在出现故障时丢失。

```
Ok
save    "DECIHEX. BAS"
```

13）运行 BASIC 程序

```
Ok
run
Decimal Number?10←键入的十进制数
Hex Equivalent is:000A←显示出等价的十六进制数
Decimal Number?100
Hex Equivalent is :0064
Decimal Number?←键入 Ctrl Break
Break in 40
Ok←返回到 BASIC 的命令状态
```

为了以后运行该程序方便起见，我们应该把 BINIHEX&. 程序作为二进制文件保存起来，而且可以在 BASIC 程序中用 BLOAD 语句把该文件装入内存。下面两步就做这些工作。

14）在 BASIC 中用二进制形式保存汇编语言子程序；

```
Ok
def seg←&h9f94
Ok
bsave    "binihex&. bin",0,&h25
```

其中，0为文件的偏移地址，&h25为文件长度，它可以从汇编时产生的 LST 文件中查到。

15）修改 DECIHEX. BAS 文件，增加装入 BINIHEX&. BIN 文件的功能；

```
25    bload    "binihex&. bin",0
```

其中，0为文件的偏移地址。

16）保存最后的 DECIHEX. BAS 程序

```
Ok
save"DECIHEX. BAS"
```

程序如下：

```
 10  DEFINT  A—Z
 20  DEF SEG=&9F94
 25  BLOAD "BINIHEX&. BIN",0
30  DEF USR0=0
 40  INPUT "Decimal Number?";N
 50  PRINT "Hex Equivalent is:";
 60  D=USR0(N)
 70  PRINT PRINT
 80  GOTO 40
```

在完成以上所有步骤以后，可以在任何时候从 BASIC 执行 DECIHEX 程序了，此时汇编语言子程序是自动装入并调用的。

例13. 10中只反映了从 BASIC 语言程序传送一个变量到汇编语言程序的情况，下面再举一个从汇编语言程序回送一个变量给 BASIC 语言程序的情况。

例13. 11 HEXIDEC 程序从键盘接收一个十六进制数并以十进制形式打印出来。这一程序的前半部分即从键盘接收十六进制数并转换为二进制数的部分用汇编语言编写；后半部分用 BASIC 语言从 FAC 中取得该二进制数，并以十进制数的形式把它打印出来。编制程序如图13. 21所示。

```
;HEXIDEC. BAS
10  DEFINT A——Z
20  DEF SEG=&H9F94
30  DEF USR0=0
35  BLOAD "HEXIBIN&. BIN"
40  PRINT "Hex Number?";
50  N=USR0(D):PRINT
60  PRINT "Decimal Equivalent is";N
70  PRINT
80  GOTO 40
```

(1)

```
;HEXIBIN&——Hexadecimal to binary converter
;Converts hex from keyboard to binary
;For use with BASIC programs
;on entry.  BX holds FAC address

key_in      equ     1h          ;keyboard input
doscall     equ     21h         ;DOS interrupt number
;* * * * * * * * * * * * * * * * * * * * * * * *
hexibin     segment             ;definte code segment
;————————————————————————————————
```

```
main    proc    far             ;main part of program
        assume cs:hexibin
;get digit from keyboard convert to binary
        mov     dx,0            ;clear DX for number
newchar:
        mov     ah,key_in       ; keyboard input
        int     doscall         ;call DOS
        sub     al,30h          ;ASCII to binary
        jl      exit            ;jump if < 0
        cmp     al,10d          ;is it < 10d ?
        jl      add_to          ;yes,so it is digit
;not dec digit ( 0 to 9 ) may be letter ( A to F )
        sub     al,27h          ;convert ASCII to bin
        cmp     al,0ah          ;is it <0a hex?
        jl      exit            ;yes,not letter
        cmp     al,10h          ;is it >0f hex?
        jge     exit            ;yes,not letter
;is hex digit add to number in DX
add_to:
        mov     cl,4            ;set shift count
        shl     dx,cl           ;rotate DX 4 bits
        mov     ah,0            ;zero out AH
        add     dx,ax           ;add digit to number
        jmp     newchar         ;get next digit
;put number in Floating Point Accumulator
exit:
        mov     [bx],dx         ;number from DX
        ret                     ;return to BASIC
main    endp                    ;end subprocedure
;-----------------------------------------------
hexibin ends                    ;end of code segment
;**************************************
        end     main            ;end assembly
```

(2)

图 13.21 例13.11的程序实现

(1) BASIC 程序 HEXIDEC.BAS

(2) 汇编语言子程序 HEXIBIN&.ASM

可以看出，HEXIDEC.BAS 的写法和例 13.10 的 DECIHEX.BAS 的写法是类似的。HEXIBIN&.ASM 的写法和例6.8中的 HEXTBIN.ASM 也几乎相同，仅是把 BX 寄存器改为 DX 寄存器，这是由于 BASIC 把 FAC 的偏移地址存放在 BX 寄存器中，因而整个程序用 DX 寄存器来保存从键盘输入的数，直到形成十六位二进制数后，用 BX 寄存器间接寻址的方式把该变量回送到 FAC 中去，然后由 BASIC 语言程序取得该数并用十进制形式打印出来。

对于字符串类型变量的传送，BASIC 使用另一种处理方法。由于字符串的长度一般较长，

且长度是不确定的，因此 BASIC 把每个字符串的“串描述符”(string descriptor)存放在其变量表中。串描述符由三个字节组成：第一个字节为串长度，第二、三个字节为字符串在 BASIC 数据区内的偏移地址。这样，找到了串描述符就能找到整个字符串。BASIC 程序通过 DX 寄存器来传送串描述符的偏移地址，也就是说，在传送字符串变量时，BASIC 程序使 DX 寄存器指向串描述符的首地址，汇编语言子程序可通过 DX 寄存器找到串描述符，从而找到(或回送)字符串变量。下面的例子将说明字符串变量的传送方法。

例13.12 在某些情况下，用户需要把自己的文件进行加密处理，使用时经过解密就可以使文件恢复原状。本例要求由 BASIC 语言程序接收从键盘输入的一串字符，再由汇编语言子程序进行加密处理，然后由 BASIC 语言程序把它们打印出来。这里的加密处理采用最简单的对每个字符的 ASCII 值加一的办法。图13.22是本例的程序实现。

```
'CODE.BAS
10  DEFINT A--Z
20  DEF SEG=&H1f94
30  DEF USR0=0
35  BLOAD "CODESUB.BIN",0
40  PRINT "Type string to be coded:";
50  LINE INPUT ST$
60  CS$=USR0(ST$+" ")
70  PRINT "Original version:";ST$
80  PRINT "Coded version:";CS$
```

(1)

```
;CODESUB--Changes string to "coded" string
;Adds 1 to every character in the string
;For use with BASIC programs
;* * * * * * * * * * * * * * * * * * * * *
codesub     segment                 ;definte code segment
;-------------------------------------------
main        proc        far         ;main part of program
            assume cs:codesub
; check that argument is really a string
            cmp         al,3        ;3 is code for string
            jne         exit        ;wrong argument
;get address of string descriptor from DX
            mov         bx,dx       ;descriptor into BX
            mov         ch,0        ;clear hi half length
            mov         cl,[bx]     ;length of string
            cmp         cx,0        ;if length is zero
            je          exit        ; then exit
            mov         si,[bx+1]   ;address of string
;add 1 to every character in string
newchar:
```

```
        add     byte ptr[si],1   ;add 1 to char
        inc     si              ;point to next char
        loop    newchar         ;loop until done
exit:
        ret                     ;return to DOS
main    endp                    ;end of main part of program
;-------------------------------
codesub ends                    ;end of code segment
;*********************
        end     main            ;end assembly
```

(2)

图 13.22 例13.12的程序实现

(1) BASIC 语言程序 CODE.BAS (2) 汇编语言子程序 CODESUB.ASM

读者可以很容易地读懂这两个程序，但有两个问题需要说明一下。第一个问题是：CODE.BAS 中的第60行不是简单地置为

```
60    CS$=USR 0(ST$)
```

而是使用

```
60    CS$=USR0(ST$+" ")
```

对这两种格式的处理是有区别的。对于前者，BASIC 程序除将串描述符的地址送给汇编语言子程序外，还会修改 ST$ 中的内容。而对于后者，BASIC 程序认为字符串 ST$ 要进行算术运算，因而把它复制到一个工作区并进行指定的操作，即加上一个为空的字符串，当然结果与原来的字符串是一样的，但现在传送给汇编语言子程序的是工作区的串描述符，原来的 ST$ 将保持不变。

需要说明的另一个问题是：汇编语言子程序不允许改变串描述符中任一个字节的内容。也就是说，除串变量的地址不允许改变外，也不允许改变字符串的长度。汇编语言子程序可以改变字符串的内容，但不能改变字符串的长度，本例中修改了原来的字符串 ST$，产生并回送给 BASIC 一个新的字符串 CS$，但两者的长度是相等的。本程序运行举例如下：

```
Ok
run
Type string to be coded :Now is the time for all good men.
Or iginal version :Now is the time for all good men.
Coded version :Opx !jt !uif !ujnf !gps !bmm !hppe !nfo/!
Ok
```

13.3.2.2 用 CALL 语句连接

由于使用 USR 连接限制了可传送变量的个数，因此对于传送多个变量的场合可以使用 CALL 语句来连接。BASIC 语言中 CALL 语句的格式为：

```
CALL SUBR[(ARG1[,ARG2]…)]
```

其中，SUBR 是一个数值变量名，它的值是所要调用的汇编语言子程序入口的偏移地址，所以在 CALL 语句前应该定义其值，如下：

```
SUBR=0
CALL SUBR (ARG1,ARG2)
```

亦即把汇编语言子程序入口的偏移地址定义为0，ARG1、ARG2则是所要传送的变量，BASIC 在调用汇编语言子程序时把这些变量的地址顺序地存入堆栈，汇编语言子程序则从堆栈中取得它们，从而取得变量。这种方法与第六章中用堆栈传送变量的方法是一样的。下面我们来看一个例子。

例13.13 COUNT. BAS 程序首先接收键入的一个字符串和一个需要查找的字符，然后调用汇编语言子程序 COUNTER. ASM，找出这个字符在字符串中的出现次数，并把该数回送给 BASIC，由 BASIC 打印出结果。使用汇编语言子程序的原因在于可以加快查找速度。本例的程序实现如图13.23所示。

```
;COUNT.BAS
10  DEFINT A—Z
20  DEF SEG=&h9f94
25  BLOAD "COUNTER.BIN",0
30  COUNT=0
40  INPUT "String to be searched?";S$
50  INPUT "Character to be counted?";C$
60  AC=ASC(C$)
70  CALL COUNT (N,AC,S$)‡PRINT
80  PRINT "Character '";C$;"' occurs";N;"times in string"
90  PRINT S$
```

(1)

```
;COUNTER.ASM——Counts number of occurrences of a character
;       in a character in a string
;For use with BASIC program
;* * * * * * * * * * * * * * * * * * * * *
pro_nam   segment                    ;define code segment
;————————————————————————————
counter   proc   far

          assume cs:pro_nam

;Address of three variables are on the stack in the form

stack_form          struc

          save_bp          dw          ?

          save_cs_ip       dw          2 dup(?)

          strdes_addr      dw          ?

          char_addr        dw          ?

          count_addr       dw          ?

stack_form          ends

;MAIN PART OF PROGRAM GOES HERE

          push     bp                  ;save calling prog BP

          mov      bp,sp               ;establish new BP

;get address of string descriptor, then
```

```
;  length of string and address of string
        mov     bx,[bp]. strdes_addr ;addr of string desc
        mov     cl,[bx]              ;string length in CL
        mov     dx,[bx+1]            ;string addr in DX
;get addr of character,then character
        mov     bx,[bp]. char_addr ;addr of character
        mov     ax,[bx]              ;character in AL
;search for char in string,count matches
        mov     bx,dx                ;addr of string in BX
        mov     dx,0                 ;DX holds count
        mov     ch,0                 ;zero out hi half CX
next_char:
        cmp     al,[bx]              ;match?
        jne     not_equal            ;no
        inc     dx                   ;yes,increment count
not_equal:
        inc     bx                   ;increment pointer
        loop    next_char            ;done string?
;put count into address of count
        mov     bx,[bp]. count_addr ;addr of count in BX
        mov     [bx],dx              ;put count in address
restore base point,return to BASIC
        pop     bp                   ;restore BP reg
        ret     6                    ;return,pop 6 bytes
counter  endp                        ;end procedure
;--------  ----------  ---
pro_nam  ends                        ;end of code segment
;* * * * * * * * * * * * * * * * * * * *
        end                          ;end assembly
```

(2)

图 13.23　例13.13的程序实现

(1) BASIC语言程序 COUNT. BAS　(2) 汇编语言子程序 COUNTER. ASM

可以看出，用 CALL 语句调用汇编语言子程序与用 USR 语句调用有很多相同之处。如汇编语言子程序入口的处理方法，从汇编语言程序返回的方法以及用串描述符来处理串变量的方法都是一样的，只是变量的传送方法比使用 USR 语句的方法更加简单。本程序运行的例子如下：

```
Ok
run
String to be searched? She sells sea shells by the seashore.
Character to be counted? s
character's' occurs 7 times in string
She sells sea shells by seashore.
Ok
```

现在我们要结束有关 BASIC 语言与汇编语言程序连接问题的讨论了。读者可以选用上述两种方法中的任一种来解决你所遇到的问题。如果你的 BASIC 版本中没有提供 CALL 语句，那就只能使用 USR 语句了。有关这方面的内容请参考 IBM PC 机 BASIC 手册的附录 C，有些问题有更详细的说明。

13.4 模块化程序设计概述

在前面的章节中，我们已经说明了汇编语言程序设计技术，也说明了多个程序模块的连接技术。在这一节里，我们要讨论编制一个较复杂的大程序的总体设计方法。我们要站得更高，并从总体来讨论这一问题。正因为是从宏观来看，所以有的问题只能说明一些基本方法或处理原则，但这种设计思想可以帮助我们编制出更好的程序来。

模块化程序设计的步骤如下：

1) 正确地描述整个程序需要完成什么样的工作；

2) 把整个工作划分成多个任务，并画出层次图；

3) 确切地定义每个任务必须做什么事，它与其它任务之间如何进行通信，写出模块说明；

4) 把每个任务写成汇编语言程序模块，并进行调试；

5) 把各个模块连接在一起，经过调试形成一个完整的程序；

6) 把整个程序和它们的说明合在一起形成文件。

本节，我们将重点说明其中的2、3和4步，其它步骤读者应该已经熟悉了。

13.4.1 模块化程序设计

把一个程序分成具有多个明确任务的程序模块，分别编制、调试后再把它们连接在一起形成一个完整的程序，这样的程序设计方法称为模块化程序设计。其优点如下：

·单个程序模块易于编写、调试及修改。

·不同模块可以分配给不同的程序员来编写及调试，有利于加快工作进度。

·程序的易读性好。

·程序的修改可局部化。

·频繁使用的任务可以编制成模块存在库里供多个任务使用。

模块化程序设计的关键是模块的划分。层次图和模块说明是划分模块所使用的工具。

层次图是一个方块图，用来表示模块和子模块之间的关系。图13.24是一个层次图的例子，可以看出它表示了各模块之间的从属关系。主模块可以调用下一层的子模块A、B、C，而这些子模块又可以调用它下一层的模块 E、B 等。所以有了层次图各个模块之间的从属关系就很清楚了。层次图的顶端是主模块，它是一个总控模块，它直接控制位于其下一层的各个模块的执行过程，而各主要的子模块再去控制其下一层的子模块……，从图中也可以看出：一个子模块可以出现在层次图的多处，也可以从属于多个模块，如图13.24中的模块 B 和 D。

模块说明应简要地写出模块的功能、所用基本算法、模块的输入输出以及它们的数据结构等。应该考虑程序中哪些数据应该放在公共数据区可以由所有模块访问，哪些数据则应该在有直接从属关系的模块间传送等。一般说来，有了层次图和各个模块的模块说明后，划分模块的工作就完成了。

划分模块是一个从顶向下(top-down)的程序设计过程。主模块是一个总控模块，所以划分

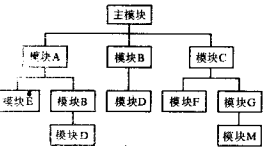


图 13.24 层次图举例

模块的第一步是要确定主要的子模块，也就是说要把总任务划分成几个主要的子任务。例如，一般说来，一个任务可以分成输入任务、输出任务和一个或几个进行处理或计算的子任务。在划分子模块的过程中应该弄清楚每个模块的功能、数据结构及相互之间的关系；第二步，对这些主要的子模块中的一些专门的子任务再划分给下一层的子模块去做，当然也要弄清楚它们相互的关系；第三步，重复上述过程，一直细分到程序已经分成易于理解和易于实现的小块为止。

那么，模块应该如何划分呢？或者说划分模块的原则应该是什么呢？由于模块的划分是很灵活的，所以这里只能说明一些指导原则。总的说来，每个模块应该具有独立的功能。此外，还应该考虑各个模块之间的联系，或者说它们之间的耦合关系。模块之间总有各种各样的耦合关系，其间的连接可归结为控制耦合(control coupling)和数据耦合(data coupling)两类。控制耦合是指模块应该在怎样的条件下进入和退出，以及它们是如何进入和退出的。数据耦合则是指模块之间的信息通信，信息量的多少以及信息通信方式等。当然，我们希望模块间的控制耦合简单以及数据耦合最小。下面给出划分模块时应该遵循的一些规则，供读者参考。

1) 模块之间的控制耦合应尽可能简单，应该尽量避免从多个入口点进入模块或从多个出口点退出。也就是说，一个模块应该只有一个入口点和一个出口点，这样的模块易于调试，也不容易出错。

2) 模块之间的数据耦合应最小，这包括数据传送量应少，或者数据传送方式应该是规则传送。如果两个模块之间传送的数据量较大，而且又是不规则传送的话，那么应该考虑把这两个任务放在同一个模块中；如果传送的数据量虽然很大，但它们可以放在公共数据区中，用同一种规律来传送信息(规则传送)的话，那还是可以考虑把这两个任务分在不同的模块中的。

3) 模块的长度适中。模块的长度可以作为划分模块的考虑因素之一，因为如果模块太长，理解和调试会发生困难，失去了模块化的优点；如果模块太短，则为该模块所做的连接、通信等工作的开销太大，又很不值得。所以，一般说来，一个模块的语句长度约在20～100的范围内比较合适。

下面我们用一个简单的例子来说明如何用层次图和模块说明来表示程序模块的划分。

例13.14 要求计算如下多项式：

$$Y \leftarrow A_n X^n + \ldots + A_1 X + A_0$$

程序的输入是数组 A 的系数集合、变量 X 和 n 的值，输出变量 Y 的值，它们都是16位非压缩 BCD 数。可以画出程序的层次图如图13.25所示。其中主模块 MAIN 调用子模块 INPUT 以输入所需要的数据，然后调用模块 EVALUATE 来执行计算，最后调用模块 OUTPUT 打印出 Y 的值。在用模块 EVALUATE 执行计算的过程中又调用模块 ADDITION 和 MULTIPLICATION 来执行加法和乘法运

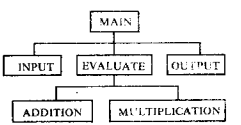


图 13.25 例13.14的层次图

算。可见用层次图很清晰地描述了程序的全貌。更加细微的信息则由以下的模块说明来给出。这里给出部分模块说明如下：

(1)INPUT

Name：INPUT

Inputs：From input terminal n，X and $A_n,\ldots,A_0$

Outputs：To MAIN through a parameter tables：

n，X and $A_n,\ldots,A_0$

Function：To input all of the data needed for evaluating

$$A_nX^n+\ldots+A_1X+A_0$$

(2) EVALUATE

Name：EVALUATE

Inputs：From MAIN through a parameter table：

n，X，$A_n,\ldots,A_0$

Output：To MAIN through a parameter table：

Value of the polynomial to Y

Function：To evaluate

$$Y \leftarrow\!- A_nX^n+\ldots+A_1X+A_0$$

using Horner's rule and the submodules ADDITION and MULTIPLICATION for performing the arithmetic operations on the 16_bit unpackedBCD numbers.

(3) ADDITION

Name：ADDITION

Inputs：From EVALUATE through a parameter tables：Operand to be added

Output：To EVALUATE through a parameter table：Sum

Function：To add two 16_bit unpacked BCD numbers.

可以看出，有了层次图和各模块的模块说明不仅完成了划分模块的工作，而且也能使人清楚地了解程序的总体情况。

13.4.2 结构化程序设计

在确定模块划分以后，下一步就是编制各个程序模块的工作了。本书前面所叙述的内容都是围绕如何编制汇编语言程序及程序设计技巧问题，所以这下一步的工作应该是读者都可以迎刃而解的了。在这里我们再介绍一下结构化程序设计方法，作为本书前面所述程序设计方法的一个总结。使用结构化程序设计方法能使编制的程序结构清晰，易于读懂，易于调试及修改，所以我们希望读者能用这种方法来进行程序设计。

结构化程序设计的思想是：程序的设计、编写和测试采用一种规定的组织方式(而不是随心所欲的)进行，在这种程序中只使用基本的逻辑结构，整个程序是各种基本结构的组合。基本的逻辑结构共有五种，每种结构都只有一个入口和一个出口。基本逻辑结构是：顺序，IF_THEN_ELSE，CASE，DO_WHILE 和 DO_UNTIL 五种，如图13.26所示。可以看出这五种结构是我们已经熟悉的顺序、分支和循环结构的几种结构形式。当然这些结构的汇编语言程序的编制方法也是我们已经很熟悉的了。

结构化程序设计要求每个程序都由这五个基本结构组合或嵌套而成，程序中每个方框都

可以用任一种基本结构来取代，当然每个方框也可以是一个子程序调用或是一次宏调用，使用基本结构并不排斥子程序结构的使用。图13.27是基本结构组合的一个例子，它是分支结构中

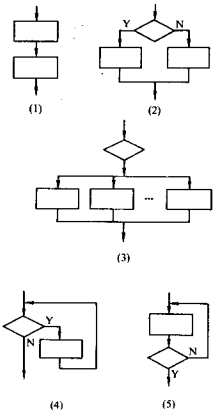


图 13.26 五种基本逻辑结构
(1) 顺序 (2) IF_THEN_ELSE
(3) CASE (4) DO_WHILE (5) DO_UNTIL

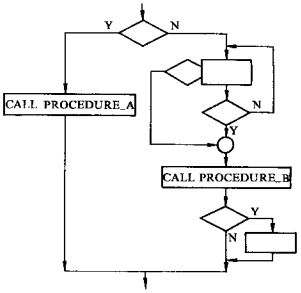


图 13.27 程序的基本结构组合举例之一

使用循环结构和分支结构的一个例子。其中右上部的循环结构增加了一个判断退出的条件，但对于这一基本结构仍然只有一个出口退出，这种结构仍然属于循环结构，在第五章里我们曾经有使用 LOOPZ 或 LOOPNZ 的例子都属于这种结构形式。图13.27的右下部为分支结构嵌套使用分支结构的情况，只是这里的一个分支不作任何处理就退出而已。图13.28是基本结构组合的另一个例子，这是循环结构中嵌套分支结构及多个子程序结构的例子。程序一开始把 ERROR_COUNT 置0，进入循环后把 FLAG 标志置0，在 PROCESS 子程序执行期间，如有某种错误发生则置 FLAG 标志为1，当从过程 PROCESS 返回后，马上测试 FLAG 标志，如 FLAG 为1则作出错处理并使 ERROR_COUNT 值加1。这里多个子程序结构的使用使总的程序结构更为清晰。

13.4.3 程序设计举例

最后让我们用一个例子来说明模块化程序设计方法。

例13.15 假设在磁盘中存有全体工作人员的情况记录及其目录，目录中存有每个工作人员记录在磁盘中的存放地址。现在需要为修改这些记录而编制一个程序，这个程序要能为新来的工作人员插入新的记录，要能为调走的工作人员删除记录，还要能更改当前记录中的内容。因此程序需要做以下的工作：

1）输入目录；

2）输入停止命令“S”，或者用字母“I”(插入)、“D”(删除)或“C”(更改)开始并用一组修改数据结束的命令；

3）如果是“S”命令，则程序在输出经修改的目录后结束，否则继续做第4步；

4）把出错码置0；

5）如果是“I”、“D”或“C”命令则做第6步，否则把出错码置1，然后做第7步；

6）按所要求的命令作修改，如修改的数据有错则置出错码为2；

7）如有错，则打印出相应的出错信息；

8）输入一组新的修改数据并做第3步。

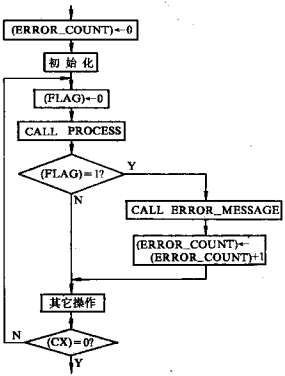


图 13.28 程序的基本结构组合举例之二

根据以上要求画出程序框图如图13.29所示，并画出层次图如图13.30所示。其中每个模块的功能如下：

MAIN：总控模块，也完成所有程序的初始化工作及结束工作。

INPUTM：从磁盘输入目录和全体工作人员的情况记录。

OUTPUTM：把目录和全体工作人员的情况记录存入磁盘。

INPUTU：输入为执行修改所需要的数据。

UPDATE：确定修改的类型并控制修改动作。

SEARCH：为给出的记录入口查找目录。

INSERT：把一个新记录入口插入目录中，并把新记录存入磁盘。

LOCATE：确定系统为存入新记录所需的信息。

DELETE：从目录中删除一个记录入口。

CHANGE：从磁盘输入一个记录，做所要求的更改，并把修改后的记录存入磁盘中。

ERROR：确定出错类型并打印出相应的出错信息。

假设目录采用链表结构，链表是一种数据结构，它是一组元素的序列，每个元素中都有一个字段，用来作为链表中下一个元素的指针，也就是说这一字段的内容是链表中下一个元素所在单元的地址。图13.31表示了链表结构的情况，在我们的例子中，链表的每个元素是一个工作人员的目录项，它包括三个字段，第一个字段是关键字，它是每个工作人员的编号，链表是按编号的上升次序排列的，所以编号就成为查找链表中元素的关键字。

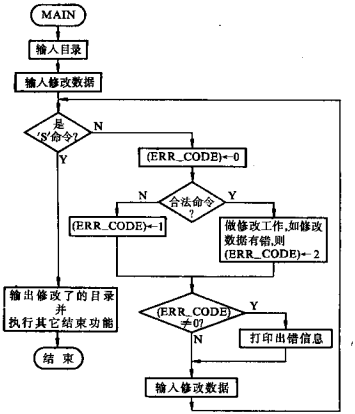


图 13.29 例13.15的程序框图

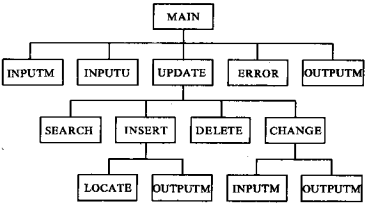


图 13.30 例13.15的层次图

规定工作人员的编号从1000～9999,第一个元素的关键字为1,这个元素用来指向第一个工作人员的入口,它是不允许删除的;末元素的关键字定为10000,这是一个结束标志,末元素也是不允许删除的。除两头外,链表的中间元素就是以编号上升的次序来排列的,显然它占有的存储区则并不是顺序排列的。每个元素中的第二个字段为信息字段,它占有三个字的存储空间,

用来保存每个工作人员记录在磁盘中存放的地址。所以根据给定的工作人员的编号(关键字)找到该元素后,就可从信息字段取得该工作人员在磁盘中记录的地址而取得这一记录。元素中的第三个字段即链表结构所需要的指针字段,它占有一个字,用来链接下一元素。在这个例子中,我们使用相对于目录首地址的位移量作为指针值,以使用基址变址寻址方式就可以方便地找到各元素。

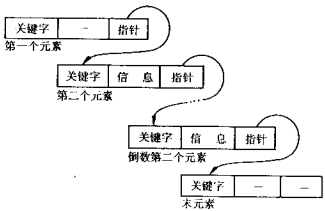


图 13.31 链表结构

采用有序链表结构对插入或删除元素的操作是很有利的,图13.32表示了链表结构中插入和删除一个元素的操作情况。图(1)中要求插入编号为5082的新元素,如原来链表中编号为5008的元素直接链向编号为5146的元素,现要在其间插入编号为5082的元素,只要把原来的链接删除(如图虚线所示),再建立新的链接(如图实线所示)就可以了。实际的操作是:把5008元素的指针字段从原来的4320改为4590,把5082元素的指针字段填入4320即可。图(2)中要求删除编号为5006的元素,也只需要把原来指向该元素的指针从原来的3830改为被删除元素的指针字段的值3010就可以了。从这里可以看出有序链表结构的优点是:它不需要像有序数组中插入或删除一个元素时需要移动其它元素的位置,只需要重新建立新的链接就可以了。再强调一下,这里的次序是按关键字排列的,而不是按存储单元的地址排列的,而关键字的大小与存储单元的地址是无关的。

现在,再来看本例中的模块组织情况。假设 INPUTM、OUTPUTM 和 LOCATE 均为系统程序库中的例行程序;MAIN、INPUTU、UPDATE 和 ERROR 均为独立的程序模块;而 SEARCH、INSERT、DELETE 和 CHANGE 则作为 UPDATE 源模块中的过程(即把这些模块组成一个源模块文件),这样它们可以使用同一个变量名以减少模块之间的变量传送。在这样的模块组织情况下,可以确定出多个模块访问的数据项,如下:

·DATA_BUFF(256个字节):用来存放当前处理的工作人员情况记录,其地址通过参数表传送。

·NO_OF_BYTES(1个字):由 INPUTM 输入或由 OUTPUTM 输出的字节数,其地址通过参数表传送。

·NEM_REC_INFO(3个字):目录中元素的信息部分,由 LOCATE 填入,其地址通过参数表传送。

·DIRECTORY(5000个字,1000个元素):全体工作人员情况记录的有序链表目录,在公共数据区。

·UPDATE_DATA(259个字节):由 INPUTU 取入的修改数据的存储区,在公共数据区。

·ERR_CODE(1个字节):用来表示是否发生过出错情况并表示出错类型的代码,在公共数据区。

·SRCH_CODE(1个字节):用来表示 SEARCH 查找结果的代码。如找到关键字为1,未找到

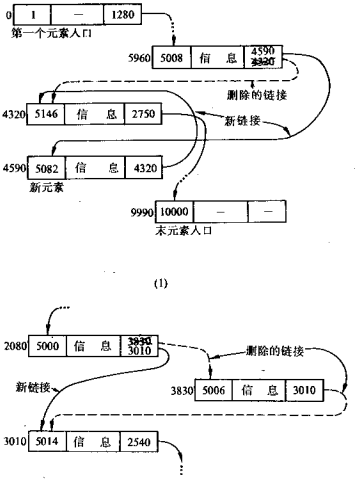


图 13.32 链表中插入和删除元素

(1) 插入一个元素 (2) 删除一个元素

关键字则为0。它是 UPDATE 及其子模块的局部变量。

·PRIOR_ELE(1个字):用来暂时存放查找过程中测试的前一个元素的下标,它是 UPDATE 及其子模块的局部变量。

·CUR_ELE(1个字):用来暂存查找过程中刚测过的元素的下标,它是 UPDATE 及其子模块的局部变量。

·KEY(1个字):用来存放用 SEARCH 查找的关键字,它是 UPDATE 及其子模块的局部变量。

在确定了模块组织及公共数据项之后,应该写出各模块的模块说明,然后编制各模块程序。在这里,我们只以部分程序及其模块说明为例来说明这一过程。下面给 出 UPDATE 及 SEARCH 的模块说明:

(1) UPDATE

Name: UPDATE

Common: Areas in common that may be affected or used DIRECTORY, UPDAT_DATA, ERR_CODE

Function: To determine the command code letter "I" (insert), "D" (delete) or "C" (change) and call the submodules SEARCH, INSERT, DELETE and CHANGE to perform the requested action. (A check for "S" (stop) is made in MAIN) It first puts 0 in ERR_CODE and then, if it cannot identify the command letter, it changes the content of ERR_CODE to 1 and returns to MAIN. Its submodules may change ERR_CODE to 2 if they detect an error.

(2) SEARCH

Name: SEARCH

Input: From UPDATE the local variable KEY

Output: To UPDATE the local variables PRIOR_ELE, CUR_ELE, SRCH_CODE

Common: Area in common that is used DIRECTORY, ERR_CODE

Function: It sets ERR_CODE to 2 and returns if (KEY) are outside the range 1000 to 9999. Otherwise it searches DIRCTORY for the key in KEY and returns to UPDATE the indices to the elements which are such that:

The key is the largest key less than (KEY).

The key is the smallest key greater than or equal to (KEY).

It puts 1 in SRCH_CODE if (KEY) are matched and 0 in SRCH_CODE if they are not matched.

UPDATE 模块的数据段以及 UPDATE 和 SEARCH 过程的程序如图13.33所示。我们虽然并未看到完整的程序，但是，从这个例子中，我们可以看到模块的划分以及把各个模块组成文件的方法。同时我们也看到了公共数据区的定义和使用方法。我们相信通过全书包括本节内容的学习，读者是能够编制出很好的汇编语言程序的。

```
;* * * * * * * * * * * * * * * * * * * *
        public      update
        extrn       locate:far,inputm:far,outputm:far
;* * * * * * * * * * * * * * * * * * * *
data_seg            segment
  srch_code         db      ?
  prior_ele         dw      ?
  cur_ele           dw      ?
  key               dw      ?
data_seg            ends
;* * * * * * * * * * * * * * * * * * * *
com_seg     segment common
        d_element           struc
          empnum            dw      ?
          info              dw      3 dup(?)
          pointer           dw      ?
        d_element           ends
;
```

```
u_data          struc
  command       db      ?
  com_key       dw      ?
  com_data      db      256     dup(?)
u_data          ends

;
  directory     d_element       1000    dup(<   >)
  err_code      db      ?
  update_data   u_data  <   >
com_seg ends
;* * * * * * * * * * * * * * * * * * *
update_seg      segment
;-------------------------------------------

        assume cs:update_seg,ds:data_seg,es:com seg

update  proc    far
        push    ax              ;save registers on stack
        push    bx
        push    cx
        push    dx
        push    si
        push    di
        mov     err_code,0      ;zero err_code
        mov     ax,update_data.com_key
                                ;move command key to AX
        mov     key,ax          ;  and key
        call    near ptr search ;search for key
        mov     dl,update data. command
                                ;put command in DL
        cmp     dl,'I'          ;if command letter is not
        jnz     not_i           ; 'I' branch to not_i
        call    near ptr insert ;otherwise,make insertion
        jmp     short exit      ;  and branch to exit
not_i:  cmp     dl,'D'          ;if command is not 'D'
        jnz     not_d           ;  branch to not_d
        call    near ptr delete ;otherwise,make delete
        jmp     short exit      ;  and branch to exit
not_d:  cmp     dl,'C'          ;if command is not 'C'
        jnz     not_c           ;  branch to not_c
        call    near ptr change ;otherwise,make change
        jmp     short exit      ;  and branch to exit
not_c:  mov     err_code,1      ;set err code to 1
exit:   pop     di              ;restore registers
        pop     si
```

```
            pop     dx
            pop     cx
            pop     bx
            pop     ax
            ret                             ;  and return
update      endp
;-------------------------------------------
            :
;-------------------------------------------
search      proc    near
            mov     srch_code,0             ;zero srch_code
            mov     ax,key                  ;put key in AX
            cmp     ax,1000                 ;check if key is
            jb      error                   ;  in the range 1000
            cmp     ax,9999                 ;     to 9999,if so
            ja      error                   ;        branch to ok
            jmp     short ok
error:      mov     err_code,2              ;else,put 2 in err_code
            jmp     short not_found         ;  and exit to not_found
ok:         lea     bx,directory.empnum     ;put base addr in BX
            mov     si,es:[bx+8]            ;put pointer in SI and
            mov     prior_ele,0             ;  prior index in prior_ele
again:      cmp     ax,es:[bx][si]          ;compare keys and
            je      found                   ;  branch to found if equal
            jl      greater                 ;if less than go to greater
            mov     prior_ele,si            ;else,update prior_ele
            mov     si,es:[bx][si+8]        ;  and SI
            jmp     short again
found:      mov     srch_code,1
greater:
            mov     cur_ele,si
not_found:
            ret
search      endp
;-------------------------------------------
            :
;-------------------------------------------
update_seg  ends                            ;end of code segment
;*************************
            end                             ;end assembly
```

图 13.33　例13.15中的 UPDATE 和 SEARCH 过程

习　题

13.1　有两个源模块如下：

```
;          source module  1
;
s_seg      segment stack
           dw      128  dup(?)
  top      label   word
s_seg      ends
;
d_seg      segment common
  var      db      50  dup(?)
d_seg      ends
;
e_seg      segment at 1000h
  area     dw      70  dup(0)
e_seg ends
;
c_seg      segment public
           .
           .       (500h bytes)
           .
c_seg      ends
;
           end
           source module 2
;
s_seg      segment stack
           dw      64  dup(?)
s_seg      ends
;
d_seg      segment common
  vect     dw      30
d_seg      ends
;
c_seg      segment public
           .
           .            (1000h bytes)
           .
c_seg      ends
;
           end
```

假设连接程序以 S_SEG，D_SEG 和 C_SEG 的次序建立段，堆栈的栈顶地址为20000，试画图说明各段在存储器中所占的地址区。

13.2　给出以下一组代码，试说明其中哪些段地址和偏移地址是由汇编程序确定的，哪些是由连接程序确定的，为什么？

```
extrn    cost:word,routine:far
public   begin
;
data_1   segment
         total   dw    50  dup(?)
         num     dw    ?
data_1   ends
;
data_2   segment
         part    dw    100  dup(?)
data_2   ends
;
code     segment
                 ⋮
         mov     ax,data_1
         mov     ds,ax
                 ⋮
         mov     ax,seg cost
         mov     es,ax
         mov     ax,total
         add     ax,es:cost
                 ⋮
         mov     ax,data_2
         mov     es,ax
         inc     es:part[si]
         mov     cx,num
         cmp     total[di],cx
         je      next
         jmp     far ptr routine
next:            ⋮
         ret
code     ends
;
         end
```

13.3 假定一个名为 MAINPRO 的程序要调用子程序 SUBPRO，试问：

(1) MAINPRO 中的什么指令告诉汇编程序 SUBPRO 是在外部定义的？

(2) SUBPRO 怎么知道 MAINPRO 要调用它？

13.4 假定程序 MAINPRO 和 SUBPRO 不在同一模块中，MAINPRO 中定义字节变量 QTY 和字变量 VALUE 和 PRICE，SUBPRO 程序要把 VALUE 除以 QTY，并把商存在 PRICE 中，试问：

(1) MAINPRO 怎么告诉汇编程序外部子程序要调用这三个变量？

(2) SUBPRO 怎么告诉汇编程序这三个变量是在另一个汇编语言程序中定义的？

13.5 假设：

(1) 在模 块1中定义了双字变量 VAR1，首地址为 VAR2的字节数组和 NEAR 标号 LAB1，它们将由模块2和

模块3所使用；

(2) 在模块2中定义了字变量 VAR3和 FAR 标号 LAB2，而模块1中要用到 VAR3，模块3中要用到 LAB2；

(3) 在模块3中定义了 FAR 标号 LAB3，而模块2中要用到它。

试对每个源模块给出必要的 EXTRN 和 PUBLIC 说明。

13.6 主程序 CALLMUL 定义堆栈段、数据段和代码段，并把段寄存器初始化；数据段中定义变量 QTY 和 PRICE；代码段中将 PRICE 装入 AX，QTY 装入 BX，然后调用子程序 SUBMUL，程序 SUBMUL 没有定义任何数据，它只简单地把 AX 中的内容(PRICE)乘以 BX 中的内容(QTY)，乘积放在 DX：AX 中。请编制这两个要连接起来的程序。

13.7 试编写一个执行以下计算的子程序 COMPUTE：

$$R \leftarrow X + Y - 3$$

其中 X，Y 及 R 均为字数组。假设 COMPUTE 与其调用程序都在同一代码段中，数据段 D_SEG 中包含 X 和 Y 数组，数据段 E_SEG 中包含 R 数组，同时写出主程序调用 COMPUTE 过程的部分。

如果主程序和 COMPUTE 在同一程序模块中，但不在同一代码段中，程序应如何修改？

如果主程序和 COMPUTE 不在同一程序模块中，程序应如何修改？

附　录

附录一　8086/8088 指令系统一览表

助记符	汇编语言格式	功　能	操 作 数	时钟周期数	字节数	标 志 位 ODITSZAPC
MOV	MOV dst,src	(dst)←(src)	mem,ac	10	3	_____
			ac,mem	10	3	
			reg,reg	2	2	
			reg,mem	8+EA	2—4	
			mem,reg	9+EA	2—4	
			reg,data	4	2—3	
			mem,data	10+EA	3—6	
			segreg,reg	2	2	
			segreg,mem	8+EA	2—4	
			reg,segreg	2	2	
			mem,segreg	9+EA	2—4	
PUSH	PUSH src	(SP)←(SP)−2	reg	11	1	_____
		((SP)+1,(SP))←(src)	segreg	10	1	
			mem	16+EA	2—4	
POP	POP dst	(dst)←((SP)+1,(SP))	reg	8	1	_____
		(SP)←(SP)+2	segreg	8	1	
			mem	17+EA	2—4	
XCHG	XCHG opr1,opr2	(opr1)↔(opr2)	reg,ac	3	1	_____
			reg,mem	17+EA	2—4	
			reg,reg	4	2	

IN	IN ac,port	(ac)←(port)		10	2	----------
	IN ac,DX	(ac)←((DX))		8	1	
OUT	OUT port,ac	(port)←(ac)		10	2	----------
	OUT DX,ac	((DX))←(ac)		8	1	
XLAT	XLAT			11	1	----------
LEA	LEA reg,src	(reg)←src	reg,mem	2+EA	2—4	----------
LDS	LDS reg,src	(reg)←(src) (DS)←(src+2)	reg,mem	16+EA	2—4	----------
LES	LES reg,src	(reg)←(src) (ES)←(src+2)	reg,mem	16+EA	2—4	----------
LAHF	LAHF	(AH)←(PSW 低字节)		4	1	----------
SAHF	SAHF	(PSW 低字节)←AH		4	1	----r r r r r
PUSHF	PUSHF	(SP)←(SP)—2 ((SP)+1,(SP))←(PSW)		10	1	----------
POPF	POPF	(PSW)←((SP)+1,(SP)) (SP)←(SP)+2		8	1	r r r r r r r r r
ADD	ADD dst,src	(dst)←(src)+(dst)	reg,reg	3	2	x ---x x x x x
			reg,mem	9+EA	2—4	
			mem,reg	16+EA	2—4	
			reg,data	4	3—4	
			mem,data	17+EA	3—6	
			ac,data	4	2—3	

续表

助记符	汇编语言格式	功　能	操作数	时钟周期数	字节数	标志位 ODITSZAPC
ADC	ADC dst,src	(dst)←(src)+(dst)+CF	reg,reg	3	2	x ———x x x x x
			reg,mem	9+EA	2—4	
			mem,reg	16+EA	2—4	
			reg,data	4	3—4	
			mem,data	17+EA	3—6	
			ac,data	4	2—3	
INC	INC opr	(opr)←(opr)+1	reg	2—3	1—2	x ———x x x x —
			mem	15+EA	2—4	
SUB	SUB dst,src	(dst)←(dst)—(src)	reg,reg	3	2	x ———x x x x x
			reg,mem	9+EA	2—4	
			mem,reg	16+EA	2—4	
			ac,data	4	2—3	
			reg,data	4	3—4	
			mem,data	17+EA	3—6	
SBB	SBB dst,src	(dst)←(dst)—(src)—CF	reg,reg	3	3	x ———x x x x x
			reg,mem	9+EA	2—4	
			mem,reg	16+EA	2—4	
			ac,data	4	2—3	
			reg,data	4	3—4	
			mem,data	17+EA	3—6	
DEC	DEC opr	(opr)←(opr)—1	reg	2—3	1—2	x ———x x x x —
			mem	15+EA	2—4	
NEG	NEG opr	(opr)←—(opr)	reg	3	2	x ———x x x x x
			mem	16+EA	2—4	

CMP	CMP opr1,opr2	(opr1)－(opr2)	reg,reg	3	2	x －－－－ x x x x x
			reg,mem	9＋EA	2－4	
			mem,reg	9＋EA	2－4	
			reg,data	4	3－4	
			mem,data	10＋EA	3－6	
			ac,data	4	2－3	
MUL	MUL src	(AX)←(AL)＊(src)	8 位 reg	70－77	2	x －－－－ u u u u x
			8 位 mem	(76－83)＋EA	2－4	
		(DX,AX)←(AX)＊(src)	16 位 reg	118－133	2	
			16 位 mem	(124－139)＋EA	2－4	
IMUL	IMUL src	(AX)←(AL)＊(src)	8 位 reg	80－98	2	x －－－－ u u u u x
			8 位 mem	(86－104)＋EA	2－4	
		(DX,AX)←(AX)＊(src)	16 位 reg	128－154	2	
			16 位 mem	(134－160)＋EA	2－4	
DIV	DIV src	(AL)←(AX)/(src)的商	8 位 reg	80－90	2	u －－－－ u u u u u
		(AH)←(AX)/(src)的余数	8 位 mem	(86－96)＋EA	2－4	
		(AX)←(DX,AX)/(src)的商	16 位 reg	144－162	2	
		(DX)←(DX,AX)/(src)的余数	16 位 mem	(150－168)＋EA	2－4	
IDIV	DIV src	(AL)←(AX)/(src)的商	8 位 reg	101－112	2	u －－－－ u u u u u
		(AH)←(AX)/(src)的余数	8 位 mem	(107－118)＋EA	2－4	
		(AX)←(DX,AX)/(src)的商	16 位 reg	165－184	2	
		(DX)←(DX,AX)/(src)的余数	16 位 mem	(171－190)＋EA	2－4	
DAA	DAA	(AL)←把 AL 中的和调整到压缩的 BCD 格式		4	1	u －－－－ x x x x x
DAS	DAS	(AL)←把 AL 中的差调整到压缩的 BCD 格式		4	1	u －－－－ x x x x x
AAA	AAA	(AL)←把 AL 中的和调整到非压缩的 BCD 格式		4	1	u －－－－ u u x u x
		(AH)←(AH)＋调整产生的进位值				

续表

助记符	汇编语言格式	功　能	操作数	时钟周期数	字节数	标志位 ODITSZAPC
AAS	AAS	(AL)←把 AL 中的差调整到非压缩的 BCD 格式 (AH)←(AH)－调整产生的借位值		4	1	u ———u u x u x
AAM	AAM	(AX)←把 AH 中的积调整到非压缩的 BCD 格式		83	2	u ———x x u x u
AAD	AAD	(AL)←10 * (AH)＋(AL) (AH)←0 实现除法的非压缩的 BCD 调整		60	2	u ———x x u x u
AND	AND dst,src	(dst)←(dst) ∧ (src)	reg,reg	3	2	0 ———x x u x 0
			reg,mem	9＋EA	2－4	
			mem,reg	16＋EA	2－4	
			reg,data	4	3－4	
			mem,data	17＋EA	3－6	
			ac,data	4	2－3	
OR	OR dst,src	(dst)←(dst) ∨ (src)	reg,reg	3	2	0 ———x x u x 0
			reg,mem	9＋EA	2－4	
			mem,reg	16＋EA	2－4	
			ac,data	4	2－3	
			reg,data	4	3－4	
			mem,data	17＋EA	3－6	
NOT	NOT opr	(opr)←($\overline{opr}$)	reg	3	2	————————
			mem	16＋EA	2－4	

XOR	XOR dst,src	(dst)←(dst)∀(src)	reg,reg	3	2	0 ———x x u x 0
			reg,mem	9+EA	2—4	
			mem,reg	16+EA	2—4	
			ac,data	4	2—3	
			reg,data	4	3—4	
			mem,data	17+EA	3—6	
TEST	TEST opr1,opr2	(opr1)∧(opr2)	reg,reg	3	2	0 ———x x u x 0
			reg,mem	9+EA	2—4	
			ac,data	4	2—3	
			reg,data	5	3—4	
			mem,data	11+EA	3—6	
SHL	SHL opr,1	逻辑左移	reg	2	2	x ———x x u x x
			mem	15+EA	2—4	
	SHL opr,CL		reg	8+4/位	2	
			mem	20+EA+4/位	2—4	
SAL	SAL opr,1	算术左移	reg	2	2	x ———x x u x x
			mem	15+EA	2—4	
	SAL opr,CL		reg	8+4/位	2	
			mem	20+EA+4/位	2—4	
SHR	SHR opr,1	逻辑右移	reg	2	2	x ———x x u x x
			mem	15+EA	2—4	
	SHR opr,CL		reg	8+4/位	2	
			mem	20+EA+4/位	2—4	
SAR	SAR opr,1	算术右移	reg	2	2	x ———x x u x x
			mem	15+EA	2—4	
	SAR opr,CL		reg	8+4/位	2	
			mem	20+EA+4/位	2—4	

续表

助记符	汇编语言格式	功　能	操作数	时钟周期数	字节数	标志位 ODITSZAPC
ROL	ROL opr,1	循环左移	reg	2	2	x ———— x
			mem	15+EA	2—4	
	ROL opr,CL		reg	8+4/位	2	
			mem	20+EA+4/位	2—4	
ROR	ROR opr,1	循环右移	reg	2	2	x ———— x
			mem	15+EA	2—4	
	ROR opr,CL		reg	8+4/位	2	
			mem	20+EA+4/位	2—4	
RCL	RCL opr,1	带进位循环左移	reg	2	2	x ———— x
			mem	15+EA	2—4	
	RCL opr,CL		reg	8+4/位	2	
			mem	20+EA+4/位	2—4	
RCR	RCR opr,1	带进位循环右移	reg	2	2	x ———— x
			mem	15+EA	2—4	
	RCR opr,CL		reg	8+4/位	2	
			mem	20+EA+4/位	2—4	
MOVS	MOVSB	((DI))←((SI))		不重复:18	1	————————
	MOVSW	(SI)←(SI)±1 或 2		重复:		
		(DI)←(DI)±1 或 2		9+17/rep		
STOS	STOSB	((DI))←(AC)		不重复:11	1	————————
	STOSW	(DI)←(DI)±1 或 2		重复:		
				9+10/rep		
LODS	LODSB	(AC)←((SI))		不重复:12	1	————————
	LODSW	(SI)←(SI)±1 或 2		重复:		
				9+13/rep		

REP	REP string primitive	当(CX)=0,退出重复;否则(CX)←(CX)−1,执行其后的串指令		2	1	————————
CMPS	CMPSB CMPSW	((SI))−((DI)) (SI)←(SI)±1 或 2 (DI)←(SI)±1 或 2		不重复:22 重复: 9+22/rep	1	x ———x x x x x
SCAS	SCASB SCASW	(AC)−((DI)) (DI)←(DI)±1 或 2		不重复:15 重复: 9+15/rep	1	x ———x x x x x
REPE 或 REPZ	REPE/REPZ string primitive	当(CX)=0 或 ZF=0 退出重复;否则,(CX)←(CX)−1,执行其后的串指令		2	1	————————
REPNE 或 REPNZ	REPNE/REPNZ string primitive	当(CX)=0 或 ZF=1 退出重复;否则,(CX)←(CX)−1,执行其后的串指令		2	1	————————
JMP	JMP short opr	无条件转移		15	2	————————
	JMP near ptr opr			15	3	
	JMP far ptr opr			15	5	
	JMP word ptr opr		reg	11	2	
			mem	18+EA	2−4	
	JMP dword ptr opr			24+EA	2−4	
JZ 或 JE	JZ/JE opr	ZF=1 则转移		16/4	2	————————
JNZ 或 JNE	JNZ/JNE opr	ZF=0 则转移		16/4	2	————————
JS	JS opr	SF=1 则转移		16/4	2	————————
JNS	JNS opr	SF=0 则转移		16/4	2	————————
JO	JO opr	OF=1 则转移		16/4	2	————————

续表

助记符	汇编语言格式	功　能	操作数	时钟周期数	字节数	标志位 ODITSZAPC
JNO	JNO opr	OF=0 则转移		16/4	2	----------
JP	JP/JPE opr	PF=1 则转移		16/4	2	----------
或 JPE						
JNP	JNP/JPO opr	PF=0 则转移		16/4	2	----------
或 JPO						
JC	JC/JB/JNAE opr	CF=1 则转移		16/4	2	----------
或 JB						
或 JNAE						
JNC	JNC/JNB/JAE opr	CF=0 则转移		16/4	2	----------
或 JNB						
或 JAE						
JBE	JBE/JNA opr	CF∨ZF=1 则转移		16/4	2	----------
或 JNA						
JNBE	JNBE/JA opr	CF∨ZF=0 则转移		16/4	2	----------
或 JA						
JL	JL/JNGE opr	SF∀OF=1 则转移		16/4	2	----------
或 JNGE						
JNL	JNL/JGE opr	SF∀OF=0 则转移		16/4	2	----------
或 JGE						
JLE	JLE/JNG opr	(SF∀OF)∨ZF=1 则转移		16/4	2	----------
或 JNG						
JNLE	JNLE/JG opr	(SF∀OF)∨ZF=0 则转移		16/4	2	----------
或 JG						
JCXZ	JCXZ opr	(CX)=0 则转移		18/6	2	----------
LOOP	LOOP opr	(CX)≠0 则循环		17/5	2	----------

LOOPZ 或 LOOPE	LOOPZ/LOOPE opr	ZF=1 且(CX)≠0 则循环	18/6	2	----------
LOOPNZ 或 LOOPNE	LOOPNZ/LOOPNE opr	ZF=0 且(CX)≠0 则循环	19/5	2	----------
CALL	CALL dst	段内直接:(SP)←(SP)−2 ((SP)+1,(SP))←(IP) (IP)←(IP)+D16	19	3	----------
		段内间接:(SP)←(SP)−2 reg ((SP)+1,(SP))←(IP) mem (IP)←EA	16 21+EA	2 2−4	
		段间直接:(SP)←(SP)−2 ((SP)+1,(SP))←(CS) (SP)←(SP)−2 ((SP)+1,(SP))←(IP) (IP)←转向偏移地址 (CS)←转向段地址	28	5	
		段间间接:(SP)←(SP)−2 ((SP)+1,(SP))←(CS) (SP)←(SP)−2 ((SP)+1,(SP))←(IP) (IP)←(EA) (CS)←(EA+2)	37+EA	2−4	
RET	RET	段内:(IP)←((SP)+1,(SP)) (SP)←(SP)+2	16	1	----------
		段间:(IP)←((SP)+1,(SP)) (SP)←(SP)+2 (CS)←((SP)+1,(SP)) (SP)←(SP)+2	24	1	

续表

助记符	汇编语言格式	功　能	操作数	时钟周期数	字节数	标志位 ODITSZAPC
	RET exp	段内：(IP)←((SP)+1,(SP)) (SP)←(SP)+2 (SP)←(SP)+D16		20	3	
		段间：(IP)←((SP)+1,(SP)) (SP)←(SP)+2 (CS)←((SP)+1,(SP)) (SP)←(SP)+2 (SP)←(SP)+D16		23	3	
INT	INT type	(SP)←(SP)−2	type≠3	52	1	−−0 0 −−−−−
	INT (当 type=3 时)	((SP)+1,(SP))←(PSW) (SP)←(SP)−2 ((SP)+1,(SP))←(CS) (SP)←(SP)−2 ((SP)+1,(SP))←(IP) (IP)←(type * 4) (CS)←(type * 4+2)	type=3	51	2	
INTO	INTO	若 OF=1，则 (SP)←(SP)−2 ((SP)+1,(SP))←(PSW) (SP)←(SP)−2 ((SP)+1,(SP))←(CS) (SP)←(SP)−2 ((SP)+1,(SP))←(IP) (IP)←(10H) (CS)←(12H)		53(OF=1) 4(OF=0)	1	−−0 0 −−−−−

IRET	IRET	(IP)←((SP)+1,(SP)) (SP)←(SP)+2 (CS)←((SP)+1,(SP)) (SP)←(SP)+2 (PSW)←((SP)+1,(SP)) (SP)←(SP)+2	24	1	r r r r r r r r r
CBW	CBW	(AL)符号扩展到(AH)	2	1	————————
CWD	CWD	(AX)符号扩展到(DX)	5	1	————————
CLC	CLC	进位位置 0	2	1	——————— 0
CMC	CMC	进位位求反	2	1	——————— x
STC	STC	进位位置 1	2	1	——————— 1
CLD	CLD	方向标志置 0	2	1	— 0 ———————
STD	STD	方向标志置 1	2	1	— 1 ———————
CLI	CLI	中断标志置 0	2	1	—— 0 ———————
STI	STI	中断标志置 1	2	1	—— 1 ———————
NOP	NOP	无操作	3	1	————————
HLT	HLT	停机	2	1	————————
WAIT	WAIT	等待	3 或更多	1	————————
ESC	ESC mem	换码	8+EA	2—4	————————
LOCK	LOCK	封锁	2	1	————————
	segreg :	段前缀	2	1	————————

注：符号说明如下： 0——置 0； 1——置 1； x——根据结果设置； ——不影响； u——无定义； r——恢复原先保存的值。

附录二　伪操作表

类型	伪操作名	格　　式	说　　明
数据	ASSUME	ASSUME　segreg,segment name[,...]	规定段所属的段寄存器
	COMMENT	COMMENT delimiter text delimiter	后跟注释,不必使用;
	DB	[variable name]　DB　operand[,...] 重复从句 repeat_count　DUP(operand[,...])	定义字节变量
	DD	[variable name]　DD　operand [,...]	定义双字变量
	DQ	[variable name]　DQ　operand[,...]	定义四字变量
	DT	[variable name]　DT　operand[,...]	定义十个字节的变量
	DW	[variable name]　DW　operand[,...]	定义字变量
	END	END　[label]	源程序结束
	EQU	expression_name　EQU　expression	赋值
	=	label=expression	赋值
	EVEN	EVEN	使地址计数器成为偶数
	EXTRN	EXTRN　name:type[,...]	说明用在本模块中的外部符号
	GROUP	name　GROUP　seg_name[,...]	可使指定的段都在 64K 的物理段内
	INCLUDE	INCLUDE　filespec	把另一个源文件放到当前的源文件中
	LABEL	name　LABEL　type type 可以是 BYTE,WORD,DWORD 或 NEAR,FAR	定义 name 的属性
	NAME	NAME module_name	为模块起名
	ORG	ORG　expression	地址计数器置成 expression 的值
	PROC	procedure_name　PROC　type ⋮ procedure_name　ENDP type 可以是 NEAR 或 FAR	包含过程块
	ENDP		
	PUBLIC	PUBLIC　symbol[,...]	说明在本模块中定义的外部符号
	.RADIX	.RADIX　expression expression 可以在 2—16 的范围内	可以改变当前的基数
	RECORD	record_name RECORD field specification[,...]	在字或字节内定义位模式

	field specification 的格式是：	length 为字段的位数
	field name,length [= preassignment]	preassign ment 为预置值
	记录预置语句的格式为：	
	variable record_name<preassignment specifications>	
SEGMENT	seg_name SEGMENT [align_type]	定义段
ENDS	[combine_type]	
	['class']	
	⋮	
	seg_name ENDS	
	align_type 可以是 PARA，BYTE，WORD 或	
	PAGE	
	combine_type 可以是 PUBLIC，COMMON，	
	AT expression，STACK 或 MEMORY	
	'class'为段组名	
STRUC*	structure_name STRUC	定义结构
	⋮	
	structure_name ENDS	
	结构预置语句的格式为	
	variable structure_name	
	<preassignment specifications>	

续表

类型	伪操作名	格　　式	说　　明
条件	IF	IF×× argument	条件伪操作格式
	ELSE	⋮	
	ENDIF	[ELSE]	
		⋮	
		ENDIF	
	IF	IF　expression	表达式不为零则为"真"
	IFE	IFE　expression	表达式为零则为"真"
	IF1	IF1	第一遍扫视为"真"
	IF2	IF2	第二遍扫视为"真"
	IFDEF*	IFDEF　symbol	符号已定义为"真"
	IFNDEF*	IFNDEF　symbol	符号未定义为"真"
	IFB*	IFB　<argument>	自变量为空则为"真"
	IFNB*	IFNB　<argument>	自变量不空则为"真"
	IFIDN	IFIDN　<arg－1><arg－2>	字符串<arg－1>和<arg－2>相同为"真"
	IFDIF	IFDIF　<arg－1><arg－2>	字符串<arg－1>和<arg－2>不同为"真"
宏	MACRO*	name　MACRO　[dummylist]	宏定义
	ENDM*	⋮	
		ENDM	
		宏调用：name　[paramilist]	
	PURGE*	PURGE　macro.name[,,,]	删去指定的宏定义
	LOCAL*	LOCAL　lists of local labels	汇编程序为指定的每个标号建立唯一的从??0001～??FFFF的符号
	REPT*	REPT　expression	REPT 和 ENDM 中的语句重复由表达式的值指定的次数
		⋮	
		ENDM	
	IRP*	IRP　dummy,<argument list>	重复 IRP 和 ENDM 中的语句，每次重复用自变量表中的一项取代语句中的哑元
		⋮	

		ENDM	
	IRPC*	IRPC dummy,string ⋮ ENDM	重复 IRPC 和 ENDM 中的语句，每次重复用字符串中的下一个字符取代语句中的哑元
	EXITM*	EXITM	可立即退出宏定义块或重复块
	&*	text & text	在展开宏定义时，可合并前后两个符号而形成一个符号
	;;*	;;text	展开时不产生其后的注释
	!*	! character	自变量中使用！则后面的字符直接输入
	%*	% expression	展开时把后面的表达式转换为由当前基数表示的数
列表	.CREF	.CREF	控制交叉引用文件信息的输出
	.XCREF	.XCREF	停止交叉引用文件信息的输出
	.LALL	.LALL	列出所有宏展开正文
	.SALL	.SALL	取消所有宏展开正文
	.XALL	.XALL	只列出产生目标代码的宏展开
	.LIST	.LIST	控制列表文件的输出
	.XLIST	.XLIST	不列出源和目标代码
	%OUT	%OUT text	在汇编期间遇到%OUT 时，在屏幕上显示出 text 项
	PAGE	PAGE [operand_1][operand—2]	控制列表文件的格式和打印，建立页的长度和宽度
	SUBTTL	SUBTTL text	在每页的标题后一行打印出一个子标题
	TITLE	TITLE text	指定每一页第一行上要打印的标题
	.LFCOND	.LFCOND	恢复对赋值为假条件块的列表
	.SFCOND	.SFCOND	取消对赋值为假条件块的列表
	.TFCOND	.TFCOND	置换控制假条件列表的现行设置

注：* 表示 ASM 不支持

附录三　中断向量地址一览表

一、8088 中断向量		
0—3	0	除以零
4—7	1	单步(用于 DEBUG)
8—B	2	非屏蔽中断
C—F	3	断点指令(用于 DEBUG)
10—13	4	溢出
14—17	5	打印屏幕
18—1F	6,7	保留
二、8259 中断向量		
20—23	8	定时器
24—27	9	键盘
28—2B	A	彩色/图形
2C—2F	B	异步通讯(secondary)
30—33	C	异步通讯(primary)
34—37	D	硬磁盘
38—3B	E	软磁盘
3C—3F	F	并行打印机
三、BIOS 中断		
40—43	10	屏幕显示
44—47	11	设备检验
48—4B	12	测定存储器容量
4C—4F	13	磁盘 I/O
50—53	14	串行通讯口 I/O
54—57	15	盒式磁带 I/O
58—5B	16	键盘输入
5C—5F	17	打印机输出
60—63	18	BASIC 入口代码
64—67	19	引导装入程序
68—6B	1A	日时钟
四、提供给用户的中断		
6C—6F	1B	Ctrl—Break 控制的软中断
70—73	1C	定时器控制的软中断
五、数据表指针		
74—77	1D	显示器参量表
78—7B	1E	软盘参量表
7C—7F	1F	图形表
六、DOS 中断		
80—83	20	程序结束
84—87	21	系统功能调用
88—8B	22	结束退出
8C—8F	23	Ctrl—Break 退出
90—93	24	严重错误处理
94—97	25	绝对磁盘读功能
98—9B	26	绝对磁盘写功能
9C—9F	27	驻留退出
A0—BB	28—2E	DOS 保留
BC—BF	2F	打印机
C0—FF	30—3F	DOS 保留
七、BASIC 中断		
100—17f	40—5F	保留
180—19F	60—67	用户软中断
1A0—1FF	68—7F	保留
200—217	80—85	由 BASIC 保留
218—3C3	86—F0	BASIC 中断
3C4—3FF	F1—FF	保留

附录四　DOS 功能调用

AH	功　能	调用参数	返回参数
00	程序终止	CS＝程序段前缀	
	(同 INT 20H)		
01	键盘输入并回显		AL＝输入字符
02	显示输出	DL＝输出字符	
03	异步通讯输入		AL＝输入数据
04	异步通讯输出	DL＝输出数据	
05	打印机输出	DL＝输出字符	
06	直接控制台 I/O	DL＝FF(输入)	AL＝输入字符
		DL＝字符(输出)	
07	键盘输入(无回显)		AL＝输入字符
08	键盘输入(无回显)		AL＝输入字符
	检测 Ctrl－Break		
09	显示字符串	DS：DX＝串地址	
		'$'结束字符串	
0A	键盘输入到缓冲区	DS：DX＝缓冲区首地址	
		(DS：DX)＝缓冲区最大字符数	(DS：DX+1)＝实际输入的字符数
0B	检验键盘状态		AL＝00 有输入
			AL＝FF 无输入
0C	清除输入缓冲区并请求指定的输入功能	AL＝输入功能号 (1,6,7,8,A)	
0D	磁盘复位		清除文件缓冲区
0E	指定当前缺省的磁盘驱动器	DL＝驱动器号 0＝A,1＝B,…	AL＝驱动器数
0F	打开文件	DS：DX＝FCB 首地址	AL＝00 文件找到
			AL＝FF 文件未找到
10	关闭文件	DS：DX＝FCB 首地址	AL＝00 目录修改成功
			AL＝FF 目录中未找到文件
11	查找第一个目录项	DS：DX＝FCB 首地址	AL＝00 找到
			AL＝FF 未找到
12	查找下一个目录项	DS：DX＝FCB 首地址	AL＝00 找到
		(文件名中带＊或?)	AL＝FF 未找到
13	删除文件	DS：DX＝FCB 首地址	AL＝00 删除成功
			AL＝FF 未找到
14	顺序读	DS：DX＝FCB 首地址	AL＝00 读成功
			＝01 文件结束,记录中无数据
			＝02 DTA 空间不够
			＝03 文件结束,记录不完整
15	顺序写	DS：DX＝FCB 首地址	AL＝00 写成功
			＝01 盘满
			＝02 DTA 空间不够

续表

AH	功　能	调用参数	返回参数
16	建文件	DS：DX=FCB首地址	AL=00 建立成功 =FF 无磁盘空间
17	文件改名	DS：DX=FCB首地址 (DS：DX+1)=旧文件名 (DS：DX+17)=新文件名	AL=00 成功 =FF 未成功
19	取当前缺省磁盘驱动器		AL=缺省的驱动器号 0=A，1=B，2=C，…
1A	置DTA地址	DS：DX=DTA地址	
1B	取缺省驱动器FAT信息		AL=每簇的扇区数 DS：BX=FTA标识字节 CX=物理扇区的大小 DX=缺省驱动器的簇数
1C	取任一驱动器FAT信息	DL=驱动器号	同上
21	随机读	DS：DX=FCB首地址	AL=00 读成功 =01 文件结束 =02 缓冲区溢出 =03 缓冲区不满
22	随机写	DS：DX=FCB首地址	AL=00 写成功 =01 盘满 =02 缓冲区溢出
23	测定文件大小	DS：DX=FCB首地址	AL=00 成功 文件长度填入FCB AL=FF 未找到
24	设置随机记录号	DS：DX=FCB首地址	
25	设置中断向量	DS：DX=中断向量 AL=中断类型号	
26	建立程序段前缀	DX=新的程序段的段前缀	
27	随机分块读	DS：DX=FCB首地址 CX=记录数	AL=00 读成功 =01 文件结束 =02 缓冲区太小，传输结束 =03 缓冲区不满 CX=读取的记录数
28	随机分块写	DS：DX=FCB首地址 CX=记录数	AL=00 写成功 AL=01 盘满 =02 缓冲区溢出
29	分析文件名	ES：DI=FCB首地址 DS：SI=ASCIIZ串 AL=控制分析标志	AL=00 标准文件 =01 多义文件 =FF 非法盘符
2A	取日期		CX=年 DH：DL=月：日(二进制)

续表

AH	功　能	调用参数	返回参数
2B	设置日期	CX：DH：DL＝年；月；日	AL＝00 成功 ＝FF 无效
2C	取时间		CH：CL＝时；分 DH：DL＝秒；1/100 秒
2D	设置时间	CH：CL＝时；分 DH：DL＝秒；1/100 秒	AL＝00 成功 AL＝FF 无效
2E	置磁盘自动读写标志	AL＝00 关闭标志 AL＝01 打开标志	
2F	取磁盘缓冲区的首址		ES：BX＝缓冲区首址
30	取 DOS 版本号		AH＝发行号，AL＝版号
31	结束并驻留	AL＝返回码 DX＝驻留区大小	
33	Ctrl—Break 检测	AL＝00 取状态 AL＝01 置状态(DL) DL＝00 关闭检测 ＝01 打开检测	DL＝00 关闭 Ctrl—Break 检测 ＝01 打开 Ctrl—Break 检测
35	取中断向量	AL＝中断类型	ES：BX＝中断向量
36	取空闲磁盘空间	DL＝驱动器号 0＝缺省，1＝A，2＝B…	成功：AX＝每簇扇区数 BX＝有效簇数 CX＝每扇区字节数 DX＝总簇数 失败：AX＝FFFF
38	置/取国家信息	DS：DX＝信息区首地址	BX＝国家码(国际电话前缀码) AX＝错误码
39	建立子目录(MKDIR)	DS：DX＝ASCIIZ 串地址	AX＝错误码
3A	删除子目录(RMDIR)	DS：DX＝ASCIIZ 串地址	AX＝错误码
3B	改变当前目录(CHDIR)	DS：DX＝ASCIIZ 串地址	AX＝错误码
3C	建立文件	DS：DX＝ASCIIZ 串地址 CX＝文件属性	成功：AX＝文件代号 失败：AX＝错误码
3D	打开文件	DS：DX＝ASCIIZ 串地址 AL＝0 读 ＝1 写 ＝2 读/写	成功：AX＝文件代号 失败：AX＝错误码
3E	关闭文件	BX＝文件号	失败：AX＝错误码
3F	读文件或设备	DS：DX＝数据缓冲区地址 BX＝文件代号 CX＝读取的字节数	读成功： AX＝实际读入的字节数 AX＝0 已到文件尾 读出错：　AX＝错误码
40	写文件或设备	DS：DX＝数据缓冲区地址 BX＝文件代号 CX＝写入的字节数	写成功： AX＝实际写入的字节数 写出错：AX＝错误码

续表

AH	功　能	调用参数	返回参数
41	删除文件	DS：DX＝ASCIIZ 串地址	成功：AX＝00 出错：AX＝错误码(2,5)
42	移动文件指针	BX＝文件代号 CX：DX＝位移量 AL＝移动方式(0,1,2)	成功：DX：AX＝新指针位置 出错：AX＝错误码
43	置/取文件属性	DS：DX＝ASCIIZ 串地址 AL＝0 取文件属性 AL＝1 置文件属性 CX＝文件属性	成功：CX＝文件属性 失败：AX＝错误码
44	设备文件 I/O 控制	BX＝文件代号 AL＝0 取状态 ＝1 置状态 DX ＝2 读数据 ＝3 写数据 ＝6 取输入状态 ＝7 取输出状态	DX－设备信息
45	复制文件代号	BX＝文件代号 1	成功：AX＝文件代号 2 失败：AX＝错误码
46	人工复制文件代号	BX＝文件代号 1 CX＝文件代号 2	失败：AX＝错误码
47	取当前目录路径名	DL＝驱动器号 DS：SI＝ASCIIZ 串地址	(DS：SI)＝ASCIIZ 串 失败：AX＝错误码
48	分配内存空间	BX＝ 申请内存容量	成功：AX＝分配内存首址 失败：BX＝最大可用空间
49	释放内存空间	ES＝内存起始段地址	失败：AX＝错误码
4A	调整已分配的存储块	ES＝原内存起始地址 BX＝再申请的容量	失败：BX＝最大可用空间 AX＝错误码
4B	装配/执行程序	DS：DX＝ASCIIZ 串地址 ES：BX＝参数区首地址 AL＝0 装入执行 AL＝3 装入不执行	失败：AX＝错误码
4C	带返回码结束	AL＝返回码	
4D	取返回代码		AX＝返回代码
4E	查找第一个匹配文件	DS：DX＝ASCIIZ 串地址 CX＝属性	AX＝出错代码(02,18)
4F	查找下一个匹配文件	DS：DX＝ASCIIZ 串地址 (文件名中带？或＊)	AX＝出错代码(18)
54	取盘自动读写标志		AL＝当前标志值
56	文件改名	DS：DX＝ASCIIZ 串(旧) ES：DI＝ASCIIZ 串(新)	AX＝出错码(03,05,17)

续表

AH	功　能	调用参数	返回参数
57	置/取文件日期和时间	BX＝文件代号 AL＝0 读取	DX：CX＝日期和时间
		AL＝1 设置(DX：CX)	失败：AX＝错误码
58	取/置分配策略码	AL＝0 取码	成功：AX＝策略码
		＝1 置码(BX)	失败：AX＝错误码
		BX＝策略码	
59	取扩充错误码		AX＝扩充错误码 BH＝错误类型 BL＝建议的操作 CH＝错误场所
5A	建立临时文件	CX＝文件属性 DS：DX＝ASCIIZ 串地址	成功：AX＝文件代号 失败：AX＝错误码
5B	建立新文件	CX＝文件属性 DS：DX＝ASCIIZ 串地址	成功：AX＝文件代号 失败：AX＝错误码
5C	控制文件存取	AL＝00 封锁 ＝01 开启 BX＝文件代号 CX：DX＝文件位移 SI：DI＝文件长度	失败：AX＝错误码
62	取程序段前缀地址		BX＝PSP 地址

* AH＝0 －2F 适用 DOS 1.0 以上版本；
AH＝2F－ 57 适用 DOS 2.0 以上版本；
AH＝58－ 62 适用 DOS 3.0 以上版本。

附录五　BIOS 中断

INT	AH	功　能	调 用 参 数	返 回 参 数
10	0	设置显示方式	AL=00 40×25 黑白方式	
			=01 40×25 彩色方式	
			=02 80×25 黑白方式	
			=03 80×25 彩色方式	
			=04 320×200 彩色图形方式	
			=05 320×200 黑白图形方式	
			=06 640×200 黑白图形方式	
			=07 80×25 单色文本方式	
			=08 160×200 16 色图形(PCjr)	
			=09 320×200 16 色图形(PCjr)	
			=0A 640×200 16 色图形(PCjr)	
			=0B 保留(EGA)	
			=0C 保留(EGA)	
			=0D 320×200 彩色图形(EGA)	
			=0E 640×200 彩色图形(EGA)	
			=0F 640×350 黑白图形(EGA)	
			=10 640×350 彩色图形(EGA)	
			=11 640×480 单色图形(EGA)	
			=12 640×480 16 色图形(EGA)	
			=13 320×200 256 色图形(EGA)	
			=40 80×30 彩色文本(CGE400)	
			=41 80×50 彩色文本(CGE400)	
			=42 640×400 彩色文本(CGE400)	
10	1	置光标类型	$(CH)_{5-3}$=光标起始行	
			$(CL)_{5-3}$=光标结束行	
10	2	置光标位置	BH=页号	
			DH,DL=行,列	
10	3	读光标位置	BH=页号	CH=光标起始行
				DH,DL=行,列
10	4	读光笔位置		AH=0 光笔未触发
				=1 光笔触发
				CH=象素行
				BX=象素列
				DH=字符行
				DL=字符列

续表

INT	AH	功　能	调 用 参 数	返 回 参 数
10	5	置显示页	AL=页号	
10	6	屏幕初始化	AL=上卷行数	
		或上卷	AL=0 整个窗口空白	
			BH=卷入行属性	
			CH=左上角行号	
			CL=左上角列号	
			CH=右下角行号	
			DL=右下角列号	
10	7	屏幕初始化	AL=下卷行数	
		或下卷	AL=0 整个窗口空白	
			BH=卷入行属性	
			CH=左上角行号	
			CL=左上角列号	
			DH=右下角行号	
			DL=右下角列号	
10	8	读光标位置的	BH=显示页	AH=属性
		字符和属性		AL=字符
10	9	在光标位置显示	BH=显示页	
		字符及其属性	AL=字符	
			BL=属性	
			CX=字符重复次数	
10	A	在光标位置	BH=显示页	
		显示字符	AL=字符	
			CX=字符重复次数	
10	B	置彩色调板	BH=彩色调板 ID	
		(320×200 图形)	BL=和 ID 配套使用的颜色	
10	C	写象素	DX=行(0 — 199)	
			CX=列(0 — 639)	
			AL=象素值	
10	D	读象素	DX=行(0 —199)	AL=象素值
			CX=列(0 — 639)	
10	E	显示字符	AL=字符	
		(光标前移)	BL=前景色	
10	F	取当前显示方式		AH=字符列数
				AL=显示方式
10	13	显示字符串	ES：BP=串地址	
		(适用 AT)	CX=串长度	
			DH,DL=起始行,列	
			BH=页号	
			AL=0,BL=属性	
			串,char,char,…	光标返回起始位置

续表

INT	AH	功　能	调 用 参 数	返 回 参 数
			AL=1,BL=属性	光标跟随移动
			串:char,char,…	
			AL=2	光标返回起始位置
			串:char,attr,char,attr,…	
			AL=3	光标跟随移动
			串:char,attr,char,attr,…	
11		设备检验		AX=返回值
				bit0=1,配有磁盘
				bit1=1,80287 协处理器
				bit4,5=01,40×25BW(彩色板)
				=10,80×25BW(彩色板)
				=11,80×25BW(黑白板)
				bit6,7=软盘驱动器号
				bit9,10,11=RS—232 板号
				bit12=游戏适配器
				bit13=串行打印机
				bit14,15=打印机号
12		测定存储器容量		AX=字节数(KB)
13	0	软盘系统复位		
13	1	读软盘状态		AL=状态字节
13	2	读磁盘	AL=扇区数	读成功:AH=0
			CH,CL=磁道号,扇区号	AL=读取的扇区数
			DH,DL=磁头号,驱动器号	读失败:
			ES:BX=数据缓冲区地址	AH=出错代码
13	3	写磁盘	同上	写成功:AH=0
				AL=写入的扇区数
				写失败:
				AH=出错代码
13	4	检验磁盘扇区	同上(ES:BX 不设置)	成功:AH=0
				AL=检验的扇区数
				失败:AH=出错代码
13	5	格式化盘磁道	ES:BX=磁道地址	成功:AH=0
				失败:AH=出错代码
14	0	初始化串行通讯口	AL=初始化参数	AH=通讯口状态
			DX=通讯口号(0,1)	AL=调制解调器状态
14	1	向串行通讯口写字符	AL=字符	写成功:$(AH)_7=0$
			DX=通讯口号(0,1)	写失败:$(AH)_7=1$
				$(AH)_{0-6}$=通讯口状态

续表

INT	AH	功　能	调用参数	返回参数
14	2	从串行通讯口读字符	DX=通讯口号(0,1)	读成功:$(AH)_7=0$ (AL)=字符 读失败:$(AH)_7=1$ $(AH)_{6\sim0}$=通讯口状态
14	3	取通讯口状态	DX=通讯口号(0,1)	AH=通讯口状态 AL=调制解调器状态
15	0	启动盒式磁带马达		
15	1	停止盒式磁带马达		
15	2	磁带分块读	ES：BX=数据传输区地址 CX=字节数	AH=状态字节 AH=00 读成功 =01 冗余检验错 =02 无数据传输 =04 无引导 =80 非法命令
15	3	磁带分块写	DS：BX=数据传输区地址 CX=字节数	AH=状态字节 (同上)
16	0	从键盘读字符		AL=字符码 AH=扫描码
16	1	读键盘缓冲区字符		ZF=0 AL=字符码 AH=扫描码 ZF=1 缓冲区空
16	2	取键盘状态字节		AL=键盘状态字节
17	0	打印字符，回送状态字节	AL=字符 DX=打印机号	AH=打印机状态字节
17	1	初始化打印机回送状态字节	DX=打印机号	AH=打印机状态字节
17	2	取状态字节	DX=打印机号	AH=打印机状态字节
1A	0	读时钟		CH：CL=时：分 DH：DL=秒：1/100 秒
1A	1	置时钟	CH：CL=时：分 DH：DL=秒：1/100 秒	
1A	2	读实时钟(适用 AT)		CH：CL=时：分(BCD) DH：DL=秒：1/100 秒(BCD)
1A	6	置报警时间(适用 AT)	CH：CL=时：分(BCD) DH：DL=秒：1/100 秒(BCD)	
1A	7	清除报警(适用 AT)		

附录六　DEBUG 主要命令

DEBUG 是为汇编语言设计的一种调试工具，它通过单步、设置断点等方式为汇编语言程序员提供了非常有效的调试手段。

6.1　DEBUG 程序的调用

在 DOS 的提示符下，可键入命令：

C>DEBUG[d：][path][filename[.ext]][parm1][parm2]

其中，文件名是被调试文件的名字。如用户键入文件名，则 DEBUG 将指定的文件装入存储器中，用户可对其进行调试。如果未键入文件名，则用户可以用当前存储器的内容工作，或者用 DEBUG 命令 N 和 L 把需要的文件装入存储器后再进行调试。命令中的 d 指定驱动器，path 为路径，parm1 和 parm2 则为运行被调试文件时所需要的命令参数。

在 DEBUG 程序调入后，将出现提示符 -，此时就可用 DEBUG 命令来调试程序。

6.2　DEBUG 的主要命令

1）显示存储单元的命令 D(DUMP)，格式为：

-D[address]或

-D[range]

例如，按指定范围显示存储单元内容的方法为：

```
-d100 120
18E4：0100  C7 06 04 02 38 01 C7 06—06 02 00 02 C7 06 08 02    G...8.G.....G...
18E4：0110  02 02 BB 04 02 E8 02 00—CD 20 50 51 56 57 8B 37    ..;..h..M PQVW.
7
18E4：0120  8B
```

其中 0100 至 0120 是 DEBUG 显示的单元内容。左边用十六进制表示每个字节，右边用 ASCII 字符表示每个字节，· 表示不可显示的字符。这里没有指定段地址，D 命令自动显示 DS 段的内容。如果只指定首地址，则显示从首地址开始的 80 个字节的内容。如果完全没有指定地址，则显示上一个 D 命令显示的最后一个单元后的内容。

2）修改存储单元内容的命令有两种。

· 输入命令 E(Enter)，有两种格式如下：

第一种格式可以用给定的内容表来替代指定范围的存储单元内容。命令格式为：

-E address　[list]

例如，-E DS：100　F3'XYZ'8D

其中 F3，'X'，'Y'，'Z' 和 8D 各占一个字节，该命令可以用这五个字节来替代存储单元 DS：0100 到 0104 的原先的内容。

第二种格式则是采用逐个单元相继修改的方法。命令格式为：

-E address

例如，-e cs：100

则可能显示为：

18E4：0100　89.-

如果需要把该单元的内容修改为 78，则用户可以直接键入 78，再按"空格"键可接着显示下一个单元的内容，如

下：

```
18E4 : 0100  89.78  1B._
```

这样，用户可以不断修改相继单元的内容，直到用 Enter 键结束该命令为止。

· 填写命令 F(Fill)，其格式为：

```
—F range  list
```

例如：—f 4BA : 0100 5 F3'XYZ'8D

使 04BA：0100～0104 单元包含指定的五个字节的内容。如果 list 中的字节数超过指定的范围，则忽略超过的项；如果 list 的字节数小于指定的范围，则重复使用 list 填入，直到填满指定的所有单元为止。

3）检查和修改寄存器内容的命令 R(Register)，它有三种格式如下：

· 显示 CPU 内所有寄存器内容和标志位状态，其格式为：

```
—R
```

例如，

```
—r
AX=0000  BX=0000  CX=010A  DX=0000  SP=FFFE  BP=0000  SI=0000  DI
=0000
DS=18E4  ES=18E4  SS=18E4  CS=18E4  IP=0100      NV UP DI PL NZ NA PO
NC
18E4 : 0100 C706040238 01  MOV     WORD PTR [0204],0138      DS : 0204=0000
```

其中标志位状态的含义可见第二章表 2.1。

· 显示和修改某个寄存器内容，其格式为：

```
—R register name
```

例如，键入

```
—r ax
```

系统将响应如下：

```
AX F1F4
:
```

即 AX 寄存器的当前内容为 F1F4，如不修改则按 Enter 键，否则，可键入欲修改的内容，如：

```
—r bx
BX 0369
: 059F
```

则把 BX 寄存器的内容修改为 059F。

· 显示和修改标志位状态，命令格式为：

```
—RF
```

系统将响应，如：

```
OV DN EI NG ZR AC PE CY—
```

此时，如不修改其内容可按 Enter 键，否则，可键入欲修改的内容，如：

```
OV DN EI NG ZR AC PE CY—PONZDINV
```

即可，可见键入的顺序可以是任意的。

4）运行命令 G(Go)，其格式为

```
—G [=address1 ][address2 [address3  ...]]
```

其中，地址 1 指定了运行的起始地址，如不指定则从当前的 CS：IP 开始运行。后面的地址均为断点地址，当指令执行到断点时，就停止执行并显示当前所有寄存器及标志位的内容，和下一条将要执行的指令。

5）跟踪命令 T(Trace)，有两种格式：

· 逐条指令跟踪

-T[=address]

从指定地址起执行一条指令后停下来，显示所有寄存器内容及标志位的值。如未指定地址则从当前的 CS：IP 开始执行。

· 多条指令跟踪

T[=address][value]

从指定地址起执行 n 条指令后停下来，n 由 value 指定。

6）汇编命令 A(Assemble)，其格式为：

-A[address]

该命令允许键入汇编语言语句，并能把它们汇编成机器代码，相继地存放在从指定地址开始的存储区中。必须注意，DEBUG 把键入的数字均看成十六进制数，所以如要键入十进制数，则其后应加以说明，如 100D。

7）反汇编命令 U(Unassemble)，有两种格式。

· 从指定地址开始，反汇编 32 个字节，其格式为：

-U[address]

例如：

```
-u100
18E4:0100  C70604023801  MOV   WORD PTR [0204],0138
18E4:0106  C70606020002  MOV   WORD PTR [0206],0200
18E4:010C  C70608020202  MOV   WORD PTR [0208],0202
18E4:0112  BB0402        MOV   BX,0204
18E4:0115  E80200        CALL  011A
18E4:0118  CD20          INT   20
18E4:011A  50            PUSH  AX
18E4:011B  51            PUSH  CX
18E4:011C  56            PUSH  SI
18E4:011D  57            PUSH  DI
18E4:011E  8B37          MOV   SI,[BX]
```

如果地址被省略，则从上一个 U 命令的最后一条指令的下一个单元开始显示 32 个字节。

· 对指定范围内的存储单元进行反汇编，格式为：

-U[range]

例如：

```
-u100 10c
18E4:0100  C70604023801  MOV   WORD  PTR  [0204],0138
18E4:0106  C70606020002  MOV   WORD  PTR  [0206],0200
18E4:010C  C70608020202  MOV   WORD  PTR  [0208],0202
```

或

```
-u100 112
18E4:0100  C70604023801  MOV   WORD  PTR  [0204],0138
18E4:0106  C70606020002  MOV   WORD  PTR  [0206],0200
18E4:010C  C70608020202  MOV   WORD  PTR  [0208],0202
```

可见这两种格式是等效的。

8）命名命令 N(Name)，其格式为：

-N filespecs [filespecs]

命令把两个文件标识符格式化在 CS：5CH 和 CS：6CH 的两个文件控制块中，以便在其后用 L 或 W 命令把文件装入或存盘。filespecs 的格式可以是：

[d：][path] filename[.ext]

例如，

—N myprog

—L

—

可把文件 myprog 装入存储器。

9）装入命令 L(Load)，有两种功能。

·把磁盘上指定扇区范围的内容装入到存储器从指定地址开始的区域中。其格式为：

—L[address [drive sector sector]

·装入指定文件，其格式为：

—L[address]

此命令装入已在 CS：5CH 中格式化了的文件控制块所指定的文件。如未指定地址，则装入 CS：0100 开始的存储区中。

10）写命令 W(Write)，有两种功能。

·把数据写入磁盘的指定扇区。其格式为：

—W address drive sector sector

·把数据写入指定的文件中。其格式为：

—W[address]

此命令把指定的存储区中的数据写入由 CS：5CH 处的文件控制块所指定的文件中。如未指定地址则数据从 CS：0100 开始。要写入文件的字节数应先放入 BX 和 CX 中。

11）退出 DEBUG 命令 Q(Quit)，其格式为：

—Q

它退出 DEBUG，返回 DOS。本命令并无存盘功能，如需存盘应先使用 W 命令。

附录七　汇编程序出错信息

编码	说　明
0	Block nesting error 嵌套过程、段、结构、宏指令、IRP、IRPC 或 REPT 不是正确结束。 如嵌套的外层已终止，而内层还是打开状态。
1	Extra characters on line 当一行上已接受了定义指令的足够信息，而又出现了多余的字符。
2	Register already defined 汇编内部出现逻辑错误。
3	Unknown symbol type 在符号语句的类型字段中，有些不能识别的东西。
4	Redefinition of symbol 在第二遍扫视时，接着又定义一个符号。
5	Symbol is multi-defined 重复定义一个符号。
6	Phase error between passes 程序中有模棱两可的指令，以至于在汇编程序的两次扫视中， 程序标号的位置在数值上改变了。
7	Already had ELSE clause 在 ELSE 从句中试图再定义 ELSE 从句。
8	Not in conditional block 在没有提供条件汇编指令的情况下，指定了 ENDIF 或 ELSE。
9	Symbol not defined 符号没有定义。
10	Syntax error 语句的语法与任何可识别的语法不匹配。
11	Type illegal in context 指定的类型在长度上不可接收。
12	Should have been group name 给出的组合不符合要求。
13	Must be declared in pass 1 得到的不是汇编程序所要求的常数值。例如，向前引用的向量长度。
14	Symbol type usage illegal PUBLIC 符号的使用不合法。
15	Symbol already different kind 企图定义与以前定义不同的符号。
16	Symbol is reserved word 企图非法使用一个汇编程序的保留字(例如，宣布 MOV 为一个变量)。
17	Forward reference is illegal 向前引用必须是在第一遍扫视中定义过的。

续表

编码	说　明
18	Must be register 希望寄存器作为操作数，但用户提供的是符号而不是寄存器。
19	Wrong type of register 指定的寄存器类型并不是指令中或伪操作中所要求的。例如 ASSUME AX。
20	Must be segment or group 希望给出段或组，而不是其它。
21	Symbol has no segment 想使用带有 SEG 的变量，而这个变量不能识别段。
22	Must be symbol type 必须是 WORD、DW、QW、BYTE 或 TB，但接收的是其它内容。
23	Already defined locally 试图定义一个符号作为 EXTERNAL，但这个符号已经在局部定义过了。
24	Segment parameters are changed SEGMENT 的自变量表与第一次使用这个段的情况不一样。
25	Not proper align/combine type SEGMENT 参数不正确。
26	Reference to mult defined 指令引用的内容已是多次定义过的。
27	Operand was expected 汇编程序需要的是操作数，但得到的却是其它内容。
28	Operator was expected 汇编程序需要的是操作符，但得到的却是其它内容。
29	Division by 0 or overflow 给出一个用 0 作除数的表达式。
30	Shift count is negative 移位表达式产生的移位计数值为负数。
31	Operand type must match 在自变量的长度或类型应该一致的情况下，汇编程序得到的并不一样。例如，交换。
32	Illegal use of external 用非法手段进行外部使用。
33	Must be record field name 需要的是记录字段名，但得到的是其它东西。
34	Must be record or field name 需要的是记录名或字段名，但得到的是其它东西。
35	Operand must have size 需要的是操作数的长度，但得到的是其它内容。
36	Must be var,label or constant 需要的是变量、标号或常数，但得到的是其它内容。
37	Must be structure field name 需要的是结构字段名，但得到的是其它内容。
38	Left operand must have segment 右操作数所用的某些东西要求左操作数必须有一个段。（例如，":"）

续 表

编码	说　明
39	One operand must be const
	这是加法指令的非法使用。
40	Operands must be same or 1 abs
	这是减法指令的非法使用。
41	Normal type operand expected
	当需要变量、标号时，得到的却是 STRUCT、FIELDS、NAMES、BYTE、WORD 或 DW。
42	Constant was expected
	需要的是一个常量，得到的却是另外的内容。
43	Operand must have segment
	SEG 伪操作使用不合法。
44	Must be associated with data
	有关项用的是代码，而这里需要的是数据，例如一个过程的 DS 取代。
45	Must be associated with code
	有关项用的是数据，而这里需要的是代码。
46	Already have base register
	试图重复基地址。
47	Already have index register
	试图重复变址地址。
48	Must be index or base register
	指令需要基址或变址寄存器，而指定的是其它寄存器。
49	Illegal use of register
	在指令中使用了 8088 指令中没有的寄存器。
50	Value is out of range
	数值大于需要使用的，例如将 DW 传送到寄存器中。
51	Operand not in IP segment
	由于操作数不在当前 IP 段中，因此不能存取。
52	Improper operand type
	使用的操作数不能产生操作码。
53	Relative jump out of range
	指定的转移超出了允许的范围(－128～＋127 字节)。
54	Index displ. must be constant
	试图使用脱离变址寄存器的变量位移量。位移量必须是常数。
55	Illegal register value
	指定的寄存器值不能放入"reg"字段中。(即"reg"字段大于 7)
56	No immediate mode
	指定的立即方式或操作码都不能接收立即数。例如：PUSH。
57	Illegal size for item
	引用的项的长度是非法的。例如，双字移位。
58	Byte register is illegal
	在上下文中，使用一个字节寄存器是非法的。例如：PUSH AL。
59	CS register illegal usage
	试图非法使用 CS 寄存器。例如：XCHG CS，AX

续表

编码	说　明
60	Must be AX or AL 某些指令只能用 AX 或 AL。例如，IN 指令。
61	Improper use of segment reg. 段寄存器使用不合法。例如，1 立即数传送到段寄存器。
62	No or unreachable CS 试图转移到不可到达的标号。
63	Operand combination illegal 在双操作数指令中，两个操作数的组合不合法。
64	Near JMP/CALL to different CS 企图在不同的代码段内执行 NEAR 转移或调用。
65	Label can't have seg override 非法使用段取代。
66	Must have opcode after prefix 使用前缀指令之后，没有正确的操作码说明。
67	Can't override ES segment 企图非法地在一条指令中取代 ES 寄存器。例如：存储字符串。
68	Can't reach with segment reg. 没有使变量可达到的 ASSUME 语句。
69	Must be in segment block 企图在段外产生代码。
70	Can't use EVEN on BYTE segment 被提出的是一个字节段，但试图使用 EVEN。
71	Forward needs override 目前不使用这个信息。
72	Illegal value for DUP count DUP 计数必须是常数，不能是 0 或负数。
73	Symbol already external 企图定义一个局部符号，但此符号已经是外部符号了。
74	DUP is too large for linker DUP 嵌套太长，以至于从连接程序不能得到所要的记录。
75	Usage of ? (indeterminate)bad "?"使用不合适。例如：? +5。
76	More values than defined with
77	Only initialize list legal
78	Directive illegal in STRUC
79	Override with DUP is illegal
80	Field cannot be overridden
81	Override is of wrong type
82	Register can't be farward ref
83	Circular chain of EQU aliases
84	Feature not supported by Small Assembler (ASM)

参 考 文 献

[1] Yu-cheng Liu Glenn A. Gibson:"Microcomputer System: The 8086/8088 Family—Architecture, Programming, and Design"Prentice—Hall INC.,1984.

[2] Leo J. Scanlon: "Assembly Language Programming with the IBM PC AT" Brady Communication Company, INC., 1986.

[3] Peter Abel: "IBM PC Assembler Language and Programming" Prentice—Hall, Inc.,1987.

[4] David C. Willen and Jeffrey I. Krantz:"8088 Assembler Language Programming: The IBM PC" Howard W. Sams & Co., Inc.,1983.

[5] Donna N. Tabler,"IBM PC Assembly Language" John Wiley & Sons, Inc.,1985.

[6] Thomas P. Skinner: "An Introduction to Assembly Language Programming for the 8086 Family". John Wiley & Sons, Inc.,1985.

[7] 沈美明、叶乃春、温冬婵、尹祚明、丁士元编译:"IBM PC [0520]汇编语言程序设计",清华大学出版社,1987。

[8]Ray Duncan 著,贺志强、李昌译:"DOS 磁盘操作系统——高级程序员指南",中国科学院计算所新技术发展公司,1988。

[General Information]
书名=IBM--PC 汇编语言程序设计
作者=沈美明等
页数=422
SS号=10011275
出版日期=