同济大学软件学院汇编课程资料
目 录

汇编语言教学提纲

一 学习目的

- 了解计算机体系结构（包括 CPU、寄存器、内存、I/O 设备等）
- 掌握计算机指令的执行过程。
- 学会使用调试工具分析内存数据并编写调试指令。
- 能编写一般的汇编程序。

二 教材

选用教材：IBM PC Assembly Language and Programming，4th Edition，Peter Abel 编着，美国 Prentice Hall，Inc 出版，清华大学出版社影印版，1998，30.00 元。
参考教材：IBM-PC 汇编语言程序设计，第 2 版，沈美明编着，清华大学出版社，34.80 元。
参考资料：（1）Intel 8086 微机原理，周明德编，清华大学出版社。
（2）DOS/BIOS 调用手册

三 学分

2 学分，30 节课堂教学，必修专业基础课。答疑每日均可，在学院教学部，或在 bbs.tongji.net 中学院课程答疑区。

四 实验环境及软件：

主要软件为：DOS，DEBUG，TASM/TLINK 或 MASM/LINK
常规指令可直接在 Windows 中调试和运行。但最好准备一张 DOS 启动盘，以便绕过 Windows 对执行某些指令的限制。

64 位 Windows 下建议使用 Dosbox 软件，详见“教你如何在 64 位 WIN7 系统下配置汇编软件并运行汇编程序.pdf”文档。

五 考试方式及成绩评定办法

（1）大项目 1 个：50%，考核方式为学期末演示与答辩结合。
（2）上机考试：40%，分 DEBUG 和写编程序两部分。考核方式为限定时间完成指定程序。
（3）平时作业、提问、知识掌握情况、知识灵活掌握能力及相关知识掌握程度：10%。

六 项目选题参考方案：

选题不限，难度越大得分越高。但建议优先考虑编写中断驻留程序，因为这是体现汇编语言作用的场合之一。以下是一些选题参考，但大家可完全不受此限制。

- 编写屏幕保护程序：利用时钟中断，在后台判断，若指定时间内无鼠标动作和键盘动作，则将屏幕关闭。按一键或鼠标动作后，打开屏幕。
- 利用时钟中断在屏幕右上角显示一个能走时的计时器。
- 编写定时程序：利用时钟中断，在后台判断，若已到某日某时某分演奏一只音乐或屏幕上提醒一件事情（清华书上有类似例子）。
- 编写键盘中断程序：按一指定键后，能将键盘当作一个简易钢琴，再按该键，键盘恢复正常功能。若写驻留程序有困难，写常规程序也可（清华书上有类似例子）。

- 编写串行通信中断程序（INT B 或 INT C）。
- 分析或编写 32 位保护方式汇编程序，或 Win32 汇编程序，例如将 CPU 在保护方式与虚拟 86 方式之间切换、编写 Win32 SDK 程序中的 WinMain/WndProc 函数。请参考 FAQ 中对 32 位汇编的解释。这是很好的选题。
- 编写程序，带两个参数 READ/WRITE 和文件名，能将 CMOS RAM 中的所有数据保存到指定文件中，或从指定文件恢复 CMOS RAM 中的内容。本程序较简单。
- 类似上题，保存或恢复硬盘主引导扇区（0 道 0 面 1 扇区）及 C 盘引导记录（C 盘 0 号逻辑扇区）中的内容。本程序较简单。
- 超过操作系统，根据文件名，直接通过分析软盘根目录表和 FAT 表，读取数据区扇区中的文件内容。本程序较复杂。
- 编写能将软盘特殊格式化的程序（如格式化成每扇区 1024 字节，格式化出第 81 道等），参见样板程序 FORTRACK.ASM（在 WAN.ZIP 中）。
- 编写一个通用程序，能为一个已存在的 COM 文件增加一些代码（如运行时先问口令，回答正确才能继续运行原 COM 程序）。
- 编写类似于 MEM 的程序，能遍历内存控制块，列出内存中的全部驻留程序，地址和长度。
- 为 SETINT1C 或其它 TSR 程序增加代码，当带指定参数时，能将其从内存中释放掉。
- 为 SETINT1C 或其它 TSR 程序增加代码，能防止其重复驻留。
- 用汇编为 C 语言编写一个带输入参数和返回值的函数，实现其全过程。
- 分析硬盘主引导扇区和分区表。解释其含义。
- 分析 C 语言生成的汇编指令。
- 分析 EXE 的文件头。
- 其它任何有一定难度的程序。

七 项目提交方式

若完成汇编项目作业，请在 10.60.41.1 服务器您个人汇编子目录中，新建一个 Project 子目录，将您的项目文件放到该文件夹中。除程序外，另请写一个简单的文本文件 Readme.txt，其中至少应包含如下内容：

姓名：
新学号：
专业：软件工程
年级：XX 级
项目所属课程名称：汇编语言及系统程序设计
项目名称：XXXXXXXXXX
项目完成日期：
项目源程序文件名：
源程序汇编与连接方式：
程序功能：
使用说明：
主要技术（若有）：
程序特色（若有）：
值得改进的地方（若有）：
项目经验或感想（若有）：例如，从项目体会中谈学习汇编语言的用处。
若您认为有更多的说明需写在 Readme.txt 文件中，也可写入。

第一部分 PC 硬件和软件的基本知识 （原版教材的 Part A）

第一节 常用数制

1） 2 进制数

表示方式，xxxxB

注：MSB/LSB（Most/Least Significant Bit）

与 10 进制的互换（除 2 取余法及 8421 速算法）

2） 16 进制数

表示方式，**H，0**H ，0x**

与 10 进制的互换

与 2 进制的互换

第二节 数据表示

1） 整数表示

整数分无符号数和有符号数。

- **无符号数的表示范围**

以 1B 的数据为例：

无符号数二进制表示范围 0000 0000B~1111 1111B

十进制表示范围 0~255

256 个数可以一一对应

n 位二进制数可表示的无符号数范围为 0～2^n－1

- **有符号数的表示范围**

数据最高位用来表示符号：

1 表示负，0 表示正。

以 1B 的数据为例：

二进制	十进制
0000 0000B：	+0
0000 0001B：	+1
0000 0010B：	+2
……	
0111 1111B：	+127
1000 0000B：	-0
1000 0001B：	-1
1000 0010B：	-2
……	
1111 1111B：	-127

8 位数据可以表示 256 种不同信息，但是这时却对应 255 个数据。为了解决这个问题，采用了**补码表示**的编码方式。

- ✓ **原码、反码、补码－－三种编码方式**

n 位码字：

原码：二进制码字

反码：原码按位取反

补码：反码＋1（2^n-原码）→ 求补

以 8 位码字为例：

原码	反码	补码
0000 0000	1111 1111	0000 0000
0000 0001	1111 1110	1111 1111
……	……	……
0111 1111	1000 0000	1000 0001
1000 0000	0111 1111	1000 0000
1000 0001	0111 1110	0111 1111
……	……	……
1111 1110	0000 0001	0000 0010
1111 1111	0000 0000	0000 0001

很容易发现补码有这个性质：[[原码]$_{补}$]$_{补}$＝原码

数的**补码表示**：

补码表示：正数的补码表示=原码

负数的补码表示=反码＋1（2^n-原码）

即：

- [X]补=X，if X >=0
- [X]补=2^n-|X|， if X<0。n=8,16,32…

求负数的补码可使用取反加 1 法或 2^n 减绝对值法。（符号位不变）

补码表示	十进制	补码表示	十进制
0000 0000B	0		
0000 0001B	+1	1111 1111B	-1
0000 0010B	+2	1111 1110B	-2
……			
0111 1111B	+127	1000 0001B	-127
		1000 0000B	?

我们很容易得出以下结论：

1、[[正数]补]反＋1－> [对应负数]补（用来求某一负数的补码）

如 求-1 的补码，可通过+1 的补码 0000 0001B，取反后加 1 得 1111 1111B。

2、[[负数]补]反＋1－>|负数| （用来求某一负数的补码具体表示的负数值）

如 -1 的补码 1111 1111B，取反后加 1 得 0000 0001B，它是+1 的补码。

那对于 1000 0000B 它代表什么呢？首先它是一个负数，我们不容易从一个负数的补码看出它所表示的具体数值，我们利用第二条结论。

1000 0000B 取反加 1 后为 1000 0000B，它应该为补码 1000 0000（128）所表示负数的绝对值，所以 1000 0000B 所表示的负数为-128。

这样 8 位二进制码字所表示的 256 种信息与-128~127 就形成了一一对应关系。

同理，**n 位码字能表示的有符号范围为-2^{n-1}～2^{n-1}-1**。

补码的**符号位扩展**，正数在前面补零，负数在前面补 1。

与 C 语言数据类型的对照

使用补码将减法转换为加法操作

X-Y=X+[Y]补

[X+Y]补=[X]补+[Y]补

[[X]补]补=X

看下面的例子，我们可以发现采用补码的方式纪录有符号数的好处。

如: MOV AL, 1000 0011B

ADD AL, 1

结果 AL：1000 0100B

1、 作无符号数处理：1000 0011B=131；1000 0100B=132

2、 作有符号数处理：1000 0011B=-125；1000 0100B=-124

我们看到对于计算机来说它执行的只是 add 指令，结果是由我们程序中定义的类型而定的。

如:

计算	补码表示
1	0000 0001
+(-5)	1111 1011
-4	1111 1100

2) 字符表示

a. ASCII 码 (ISO 8859-1 或 Latin-1)

约 60 年代出现。是西文字符的编码，8 位编码及其范围，基本 ASCII 码及奇偶检验位，扩展 ASCII 码。ASCII(American Standard Code for Information Interchange)—美国信息交换标准代码。

范围 ASCII 码表分区：

ASCII 非打印控制字符（00H～1FH）；

ASCII 打印字符（20H～7FH）；

ASCII 扩展打印字符（80H～FFH）。

常见 ASCII 码：00H（C 字符串结束标志）、07H（Bell 响铃），08H(Backspace)，09H(Tab)，0AH(LF 换行)，0CH(FF)，0DH(CR 回车)，1AH(EOF)，1BH(ESC)。20H（空格），30H（字符 0），41H（大写字符 A），61H（小写字符 a）。

一些公式：

■ 数字字符的 ASCII 码与数值本身的对照关系：

‘X’=X+30H，其中 X=0~9，‘X’表示数字 X 的 ASCII 码

如：数字‘1’的 ASCII 码为‘31H’=1H+30H

■ 大写字符的 ASCII 码与数值本身的对照关系：

‘X’=X+37H，其中 X=A~F，‘X’表示大写字符 X 的 ASCII 码

如：字符‘D’的 ASCII 码为‘44H’=DH+37H=44H

■ 小写字符与对应大写字符 ASCII 码的对照关系：

大写	十六进	二进制	小写	十六进制	二进制
A	41	0100 0001	a	61	0110 0001
B	42	0100 0010	b	62	0110 0010
C	43	0100 0011	c	63	0110 0011
D	44	0100 0100	d	64	0110 0100
E	45	0100 0101	e	65	0110 0101
F	46	0100 0110	f	66	0110 0110
G	47	0100 0111	g	67	0110 0111
……			……		

‘x’=‘X’+20H，其中 X=A~X，x=a~z

如：字符‘d’(64H)与‘D’(44H)的 ASCII 码的关系：‘d’=‘D’+20H

或者

将任一字母转为大写‘X’=‘N’ and ‘1101 1111’，

将任一字母转为小写‘x’=‘N’ or ‘0010 0000’

见附录 17：ASCII 表

与 C 语言的对照：

```
if (_getche()==0x1B) printf(“\7\n”);
```

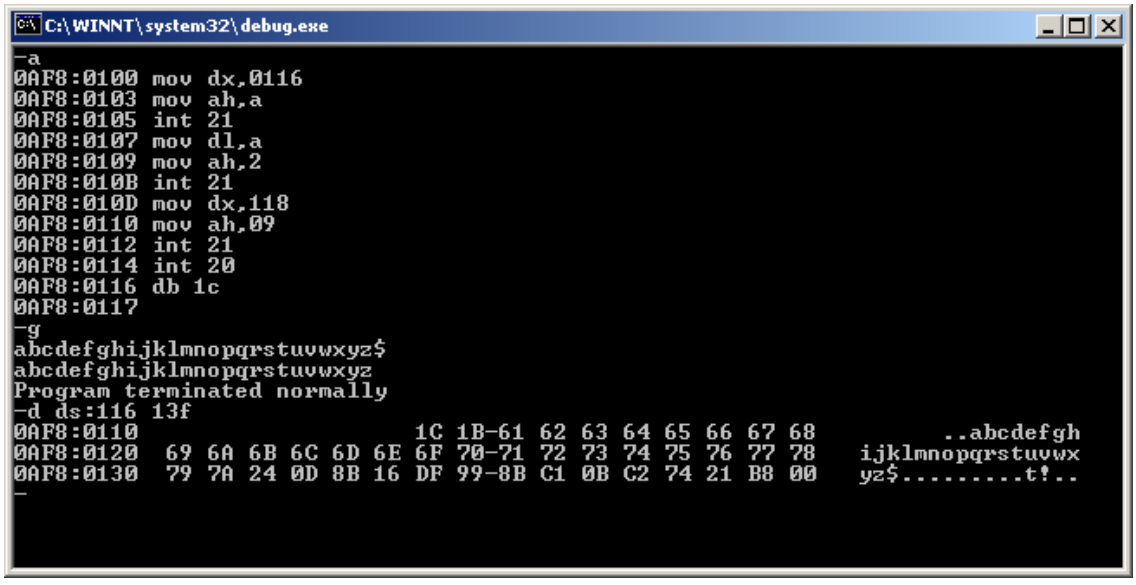
举例：键盘输入 2 和 5 两个字符，显示其和 7。

C>DEBUG↓参见　　附件 0
-A↓
xxxx:0100：MOV AH,1↓
xxxx:0102：INT 21↓　　;键盘输入数字 2 并回显，AL=32H（2 的 ASCII 码）

xxxx:0104：SUB AL,30↓　;AL=02H
xxxx:0106：MOV DL,AL↓　;DL=02H
xxxx:0108：INT 21↓　;因为 AH 始终为 1，键盘输入数字 5 并回显，AL=35H
xxxx:010A：SUB AL,30↓　;AL=05H
xxxx:010C：ADD DL,AL↓　　DL=07H
xxxx:010E：ADD DL,30↓　DL=37H

xxxx:0111：MOV AH,2↓
xxxx:0113：INT 21↓　　;显示 DL=37H 对应的 ASCII 码

xxxx:0115：INT 20↓　　;程序正常退出
xxxx:0117：↓
-t↓
-p↓
2
-t↓
……
注意执行 int 时在 debug 中使用 p 指令可不进入中断内部。
这里注意 debug 中-t,-p,-g 等执行指令的区别！
-Q↓
C>
作业 1:在 debug 环境下编写打印 ACSII 字符集的程序.
(说明:查看 21 号中断功能调用和 loop 循环指令的使用)
作业 2:分析下面的例子



本练习的中断(Interrupt)说明：
　　中断是一些子程序，事先存储在 BIOS 或操作系统中，能完成一些操作系统的基本功能，计算机启动后可直接使用。中断分软件中断（也称内中断，程序中用 INT 指令主动调用）和硬件中断（也称外中断，由硬件触发，会自动调用相应的中断程序）两类。不同类别的中断实现不同类别的

功能，例如 X86 系列共有 256 级中断，从 INT 0 到 INT FFH。INT 00-INT 1FH 称为 BIOS 中断，INT 20H 以后的称为 DOS 中断(粗略的划分)。

其中最常用的为 DOS 功能调用（即 INT 21H）。同一个中断，内部又分成许多子功能，使用时一般先需将子功能号放到 AH 寄存器中，然后根据该调用的说明设置其它的寄存器，然后执行 INT XX 指令，即可执行该中断，若有返回结果需查看指定的寄存器。

例如：从键盘上读一个字符的 INT 21 调用如下：
输入参数：AH=01
返回参数：AL=输入字符的 ASCII 码

又如：在屏幕上显示一个字符的 INT 21 调用如下：
输入参数：AH=02
DL=欲显示字符的 ASCII 码
返回参数：无

再如：DOS 中断返回，INT 20H
输入参数：无
返回参数：无

b. 汉字的 GB2312 编码

约 80 年制定。

（1）区位码：

《中华人民共和国汉字与图形字符集（GB2312）》规定，区位码表共分 94 区*94 位，包括一级汉字（最常用汉字，按拼音排序）和二级汉字（稍常用汉字，按部首排序），均为简体汉字，共 6763 个。区位码由**区号**（1-94/1H-5EH）和**位号**（1-94/1H-5EH）构成。

（2）国标码：为向 ASCII 码靠拢而制定的过渡性汉字编码。

国标码低 8 位=区位码低 8 位+20H
国标码高 8 位=区位码高 8 位+20H

国标码编码范围 2121H-7E7EH，请注意，ASCII 码中可显示西文字符范围也正好在 20H-7EH 之间，两者无法区分，所以国标码无法在计算机内使用。

（3）机内码（汉字内码）：在**计算机内实际使用的是机内码**，机内码是将国标码每个字节的最高位都置 1（即加上 80H）。

机内码低 8 位=国标码低 8 位+80H=区位码低 8 位+A0H;
机内码高 8 位=国标码高 8 位+80H=区位码高 8 位+A0H

显然，机内码是双字节编码，每个字节中的最高位肯定为 1，编码范围 A1A1H-FEFEH。与可显示的 ASCII 码可完全分开来，当然还会与扩展 ASCII 码冲突。

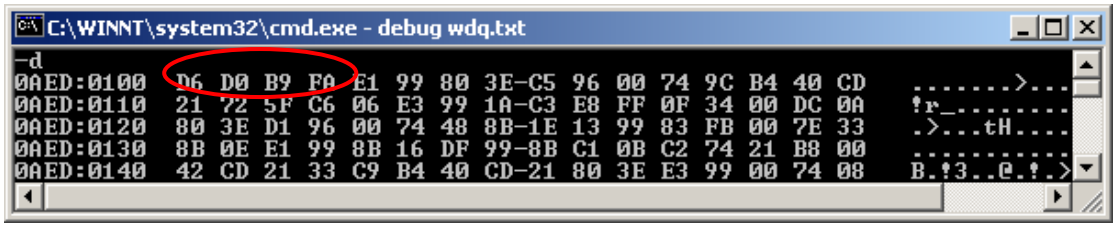
例如，查区位码表可查到汉字“万”位于 45 区 82 位，所以区位码为 4582（10 进制），写成 16 进制为 2D52H（区号和位号单独换算，45=2DH，82=52H）。

据此，可计算出汉字“万”的国标码为 4D72H（2DH+20H=4DH，52H+20H=72H），其机内码为 CDF2h（4D+80H=CDH,72H+80H=F2H），用 DEBUG 看到的都是汉字机内码。

例如：“中国”→汉字机内码＝？

汉字	区位码	汉字国标码	汉字机内码
中	5448(3630H)	5650H	D6D0H
国	2590(195AH)	397AH	B9FAH

用 debug 查看汉字机内码



GBK：扩展汉字码，新收录了 Unicode 增补汉字，与 GB2312-80 兼容。

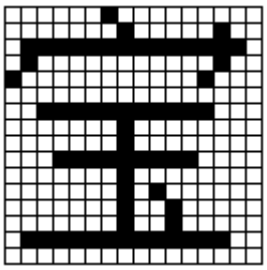
以上讲的都是基于 GB2312 的汉字编码，常称为 GB 内码，其它概念：Code Page、Big5 码（台湾用的汉字双字节内码，与 GB 码完全不兼容）、CJK。

但在目前主流操作系统中，Unicode 已占据了绝对主导地位，GB 汉字编码慢慢会成为一种历史。

(4) 点阵汉字字模

点阵汉字的字形（字模）都在汉字库中。在 GB2312 汉字库中的点阵汉字库分为 16*16，24*24，32*32 等几种点陈，例如若采用 16*16 的点阵字库，把一个方块横向和纵向都分为 16 格。若用 1 表示黑点，用 0 表示白点，则描述一个汉字的字模（一个汉字的形状）需要 16*16 个二进位，共 32 字节。

如：汉字“宝”的 16×16 点阵数字化信息：



02H 00H 01H 04H 7FH FEH 40H 04H
80H 08H 00H 00H 3FH F8H 01H 00H
01H 00H 1FH F0H 01H 00H 01H 40H
01H 20H 01H 20H 7FH FCH 00H 00H

到 16*16 点阵字库文件中取字模的公式为：

某汉字在 16*16 点阵字库文件中相对于文件首的字节偏移=[（区号-1)*94+(位号-1)）*32 到该偏移处直接取 32 字节，便可得到该汉字的二进制图形字模，然后写图形程序可将该汉字画在屏幕上用某个颜色打点，便可将点阵汉字显示在屏幕上，早期的 CCDOS 都是这样显示汉字的。现在外面的 LCD 大屏幕汉字显示很多还是这样做的。

见服务器 wan.zip 文件中 hzfd.c 程序（直接取字模放大汉字的源程序）。

c. Unicode

历史：约 90 年，ISO 制定了 ISO10646 ，也称 UCS(Universal Character Set 世界通用字符集)。UCS 用 4 字节通吃世界所有文字。实际操作上，UCS 因太肥使其无法实用。

Unicode 是一计算机行业组织，约 90 年开始着手制定全球文字统一编码，包括 CJK 所有简繁汉字在内，其制定的 Unicode 码为 2 字节编码。92 年起，Unicode 与 ISO 10646 使用相同的字库与编码（但仍为 2 字节编码）。Unicode 最终形成于 93 年。

重要性：目前及将来主流软件都使用 Unicode 为字符内部编码，Unicode 是 LINUX(UTF-8)、JAVA(UTF-8)、WindowsXP/2K(UTF-8)、Windows NT(UTF-16)等的内部编码。XML 规定所有的 XML Parser 必须支持 UTF-8 和 UTF-16。一般处理原理为：输入时按本地设置的编码标准（如 GB2312）->内部转换为 UTF-8->输出时再转换为本地设置的编码标准（如 GB2312）。Unicode 可很容易地与各种本地编码转换。

Unicode 编码范围：

U+0000~U+00FF 与 ISO 8859-1 一致，即 ASCII 码，当然 ASCII 码需前补 8 位 0。

U+4E00~U+9FFF：为 GB2312、Big5 汉字。

✓ **UTF8 和 UTF-16**

为了能在软件中使用 Unicode 成为可行，Unicode 推荐了 UTF-8 和 UTF-16 两种 Unicode 变换码。UTF 为 Unicode/UCS Transformation Format(通用字符集的转换码)的缩写。

UTF-16：同 Unicode 的 63000 多个编码，但通过代理机制可表示 1M 个编码。因 UTF-16 的西文字符为双字节，与现有大多使用单字节 ASCII 码的软件不兼容，很少用。

UTF-8：为最重要最主流的 Unicode 变种。可能采用 1-3 字节表示一个字符，规定必须根据所在的 Unicode 编码范围按下表选择编码的字节数并进行变换，并且必须选最短的字节。

U+0000~U+007F：0xxx xxxx，7 位编码。此部分即原基本 ASCII 码范围。

U+0080~U+07FF：110x xxxx 10xx xxxx，11 位编码。此部分为扩展 ASCII 码范围及部分 Latin 字符。

U+0800~U+FFFF：1110 xxxx 10xx xxxx，10xx xxxx ，16 位编码。汉字均在此范围中字符。

例如：“万”的 GB2312 码为 CDF2H，Unicode 码为 4E07H（即 UTF-16 码），其 UTF-8 码如下：

4E07： -> 0100 1110 0000 0111B

UTF-8：1110 0100 1011 1000 1000 0111 即 E4H，B8H，87H。

UTF-8 的好处：西文字符仍为单字节，原西文软件可用。由于每个字节中均有一些标字字节，不会出现原 UTF-8 中连续汉字中因单个汉字被错杀可能造成的后继汉字均混乱情况。缺点：汉字比原来宽 1.5 倍。

✓ 与 C 语言的对照

约 93 年，ANSI C 标准进行了修正，定义了新数据类型 wchar_t 表示宽字符（即 Unicode），例如 wchar_t *str=L“ABC”；表示“ABC”为 Unicode 串。所有的标准 C 函数也出现了宽字符版本，如 printf->wprintf 等。

VC++中的 TCHAR 类型也如此(please see TCHAR.H)：

```
#ifdef _UNICODE
typedef wchar_t     TCHAR;
#else
typedef char        TCHAR;
#endif
```

其中，W 表示宽串。宽串占 2 字节，比 char 宽了 1 倍。

在 VC++中，还可更灵活地定义字串，例如 TCHAR *s=_T(“ABC”)，则 VC++将根据本程序中先前是否定义过#define _UNICODE，灵活翻译_T()中的串。

d. BCD 码

BCD 编码是二—十进制编码，采用 4 位二进制数表示一位十进制数。有压缩 BCD 码格式和非压缩 BCD 码格式。如十进制 566，采用压缩 BCD 码为$(0000\ 0101\ 0110\ 0110)_{BCD}$占两个字节，采用非压缩 BCD 为$(0000\ 0101\ 0000\ 0110\ 0000\ 0110)_{BCD}$占用 3 个字节，Intel 处理器提供专门对 BCD 码形式的数字进行处理的指令。

✓ 用 DEBUG 观察补码和字符编码

观察补码：用 DEBUG 输入一条指令：MOV AL,-1，观察-1 的补码为 FFH。

观察 ASCII 码：编 DEBUG 程序：输入字符 2 和 3，然后显示其和 5。

观察汉字编码：用记事本建立一文本文件，内容包括：万你 AB13。分别以 ANSI（西文字符以 ASCII 码存储，汉字以 GB 内码存储）、Unicode（即 UTF16）、Unicode Big-endian（即 UTF-16）、UTF-8 存盘，然后用 DEBUG 观察其中的内容。

作业：查自己姓氏的 GB2312 码、UTF-16 和 UTF-8 码。

关于 Unicode 的参考文献：网上搜索文章《无废话 XML》。

第三节 x86 系列 PC 的基本结构

1) Intel 系列处理器

1．8086/8088：8088 是准 16 位微处理器，内部 16 位数据总线，外部 8 位数据总线，一个总线周期只能吞吐一个字节；8086 是 16 位微处理器，内部、外部都为 16 位数据总线，一个总线周期能吞吐一个字。8088 被 IBM 用于第一台微机上的处理器，两者指令系统、编码格式完全相同，在软件上兼容。

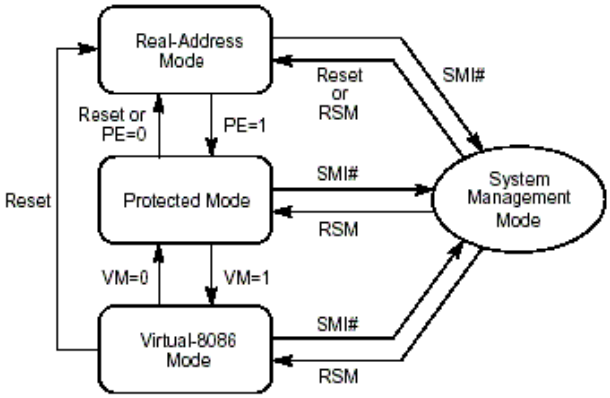
其实 8086 先推出来，但是由于当时的大多数外设只能进行字节传送所以后来又推出了 8088，虽然 8088 的性能较 8086 稍差，但是能搭配市场上原有的八位的周边装置，降低成本。（对应的机型叫 IBM PC/XT-extended technology）

此时就已经提出存储器的分段式管理机制，一直沿用至今。

2．80286：8086/8088 不具备实现完善的多任务操作系统的功能，于是 Intel 推出了 80286，CPU 可以工作在**保护模式**，保护模式下，系统可提供虚拟存储管理和多任务管理，但是它对 8086/8088 的兼容做的不好，这个产品在某种意义上是失败的。（对应的机型叫 IBM PC/AT-advanced technology）

3．80386：划时代产品，32 位处理器，硬件上支持存储器的分页管理，为保护模式下的虚拟内存管理提供了硬件支持。它有 4 种工作模式：

1) 实模式 real mode：工作方式相当于一个 8086；
2) 保护模式 protected mode：提供支持多任务环境的工作模式，建立保护机制；
3) **虚拟模式** virtual mode：可从保护模式切换至其中的一种 8086 工作方式；
4) **系统管理模式** SMM System Management Mode：出现于 Intel386SL 芯片。这个模式为 OS 实现平台指定的功能（比如电源管理或系统安全）提供了一种透明的机制。当外部的 SMM interruption（SMI#）被激活或者从 APIC（Advanced Programming Interrupt Controller）收到一个 SMI，处理器将进入 SMM。在 SMM 下，在保存当前正在运行程序的整个上下文（Context）时，处理器切换到一个分离的地址空间（原来的显存空间）。然后 SMM 指定的代码或许被透明的执行。当从 SMM 返回时，处理器将回到被系统管理中断之前的状态。



工作方式切换图

说明：PE 为控制寄存器 CR0 的保护方式使能位（Protection Enable）
VM 为标志寄存器中的虚拟机位（Virtual Machine）
RSM 指令用来返回被 SMI 中断的工作状态

4．80486：增加了高速 cache，使处理速度得到提高，另外将协处理器集成到芯片中提高浮点远算能力，增加了流水线执行方式。

5．Pentium：采用了多级流水线结构。486 之前的处理器采用单级流水线，使得处理器只有在完成一条指令后才能执行下一条指令。流水线结构是处理器将要执行的指令分解为多个利用不同资源的顺序步骤，这样可以使 CPU 并行运行多个操作。Pentium 为 5 级流水线，Pentium II 为 12 级流水线。

6．Itanium：Intel 联合 HP 推出的，是真正意义上的 64 位 CPU，面向高端市场，但是 Itanium 指令系统和 x86 指令系统不兼容。

总线宽度

年代	CPU	AB 宽度	寻址能力	DB 宽度	寄存器宽度

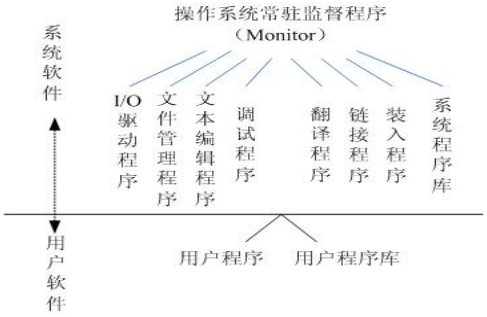
1979	8086/8088	20	1M	16/8	16
1982	80286	24	16M	16	16
1985	80386	32	4G	32	32
1989	80486	32	4G	32	32
1993	Pentium	32	4G	64	64
2001	Itanium	44	16T	64	64

8086 CPU 只有 20 根地址线，所以只能访问 1M 内存，但 80386 及以上的 CPU 有实模式(Real Mode)、保护模式(Protected Mode)两种工作方式，当其工作于实方式时，与 8086 完全兼容，只能访问 1M 以内的内存。但当工作于保护方式下，CPU 直接使用 32 位寄存器及 32 位内存地址访问内存，可直接访问 4G 内存地址空间。实方式与保护方式完全不兼容。在保护方式下不再有段地址与偏移地址之分。DOS 只能工作于实模式下，所以在 DOS 下只能访问 1M 内存（实际上 DOS 下的程序只能使用 640K 的基本内存）。Windows 工作于保护方式下。另外，80386 及以上的 CPU 在保护方式下还有一种虚拟 86 工作方式（V86 Mode），在 V86 方式下，一个 80386 可模拟多个 8086 虚拟机，使原 DOS 程序仍可使用。上位内存是给系统硬件使用的，无论 CPU 工作于何种方式下，用户程序都不能使用该块区域存储数据。(详见 8086 内存分布)

2) 计算机软件系统

系统软件：用户使用计算机时，为开发、调试、执行用户程序必备的。

用户软件：用户自行编制的程序。

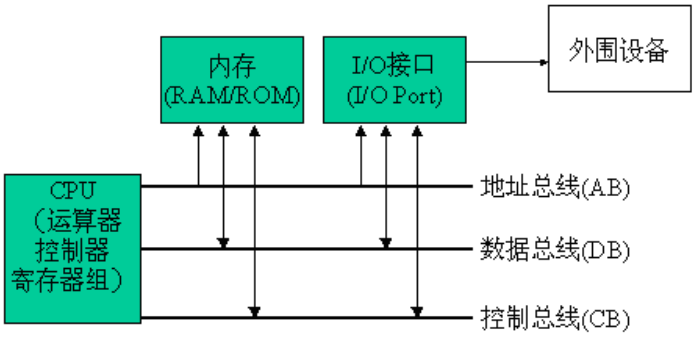


计算机软件层次

主要的系统软件：

- **操作系统：**系统软件的核心，主要作用是对系统软硬件资源进行管理，为用户创造方便、有效和可靠的计算机工作环境。其主要部分是常驻程序，一开机就驻留在内存中，用来接收用户指令并使 OS 执行相应动作。
- **I/O 驱动：**用来对 I/O 设备进行控制或管理。
- **文件管理程序：**用来处理存储在外存储器中的大量信息（外存储器中的信息是以文件的形式进行存在的），此时常与外存储器的设备驱动相连接使用。
- **文本编辑程序：**用来建立、输入或修改由字母、数字、符号等信息组成的文本。
- **翻译程序：**用来将汇编语言或其它高级语言转变为机器语言的工具，主要包括汇编程序、编译程序和解释程序等。
- **链接程序：**用来将目标文件和库文件或其它翻译好的子程序链接生成可执行文件。
- **装入程序：**用来将程序从外存储器装入内存以便程序执行。
- **调试程序：**系统提供给用户用于监控用户程序的工具。如 debug 程序。
- **系统/用户程序库：**各种标准程序、子程序和一些文件的集合。

3) 微机的基本结构



微机结构示意图

- **CPU**

  CPU 按功能分成 BIU 和 EU 两个单元：

  ✓ **执行部件（EU）：**

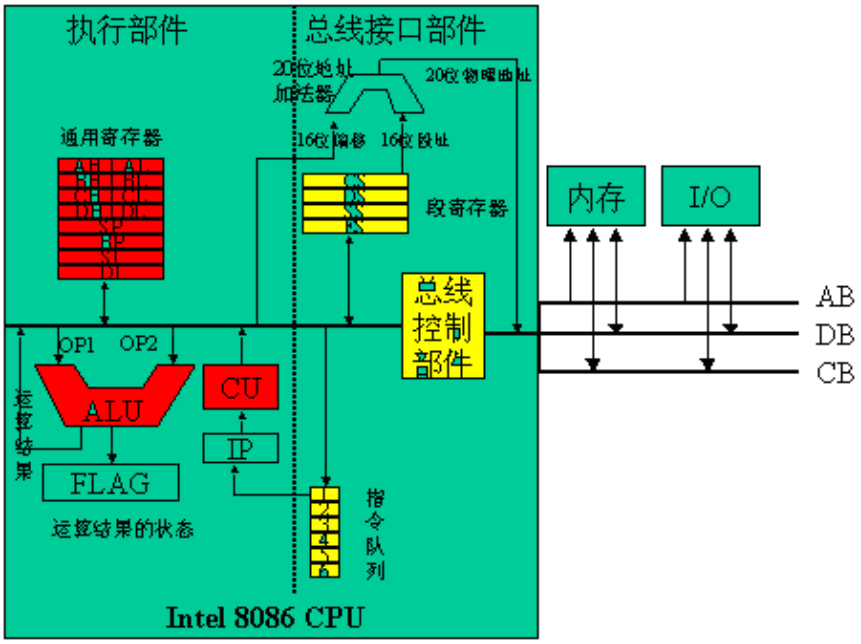
  执行指令，并进行算术逻辑运算，给出程序要访问的内存单元地址。

- ✧ 运算器（ALU，Arithmetic Logical Unit）：进行算术与逻辑运算；
- ✧ 控制器（CU，Control Unit）：取指、译码并执行指令；
- ✧ 寄存器组(Registers)：一些暂时存放数据的临时存储单元，位于 CPU 内部。

  ✓ **总线接口部件（BIU）：**

  负责 CPU 与存储器和外设间的信息传送，给 EU 提供数据和指令。

- ✧ 总线控制部件
- ✧ 段寄存器：存储器寻址
- ✧ 指令队列：



8086 CPU 的内部结构

比较寄存器与内存储器的不同：

物理位置的不同：寄存器位于 CPU 内，一旦 CPU 造好，不可再扩充；而内存是独立于 CPU 之外的独立芯片，可扩充。

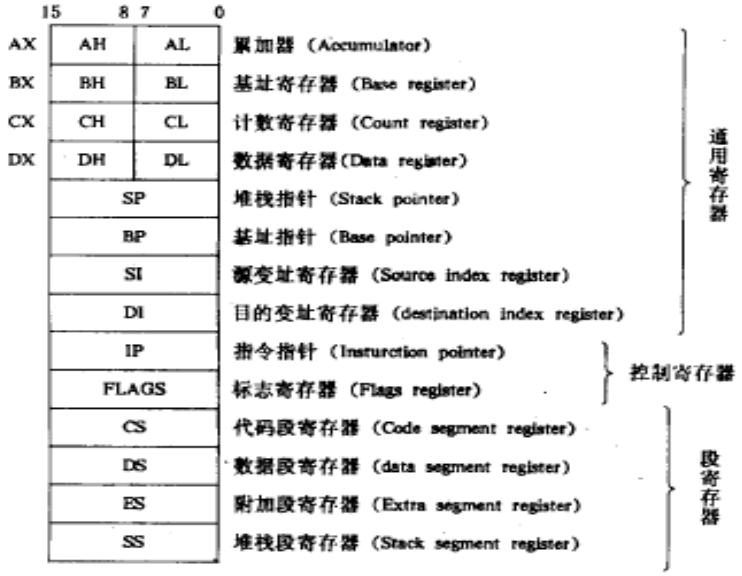
存储速度的不同：寄存器的存取速度极快，与 CPU 的工作速度相当；而内存的工作速度较慢。

容量的不同：寄存器数量较少，例如 8086CPU 内只有 14 个 16 位寄存器，只能存放少量数据；内存容量很大，适于存储大量数据。

使用方式的不同：寄存器通过名字使用，而内存通过地址访问。

数据存储顺序：寄存器中的数据按通常顺序存放，而内存中的数据依不同的 CPU 而定，例如 x86 系列 CPU 按倒字存放原则存放。（后有详细介绍）

✓ **8086 寄存器**



Intel 8086/8088 内部寄存器

熟悉寄存器的名字：

8086 共 14 个 16 位的寄存器：AX、BX、CX、DX、SI、DI、SP、BP、IP、CS、SS、DS、ES、PSW，其中 AX、BX、CX、DX 四个寄存器既可当作一个 16 位的寄存器使用，也可当作 2 个 8 位的寄存器使用（分别为 AH 和 AL，BH 和 BL，CH 和 CL，DH 和 DL）。对部分寄存器解释如下：

✓ **通用寄存器：**

AX：累加器(Accumulator)，作为算术运算的主要寄存器，在乘、除法等指令中用来存放操作数，另外在端口操作指令 I/O 中，只能使用 AX 或 AL 来与外部设备传送信息。

BX：基址寄存器（Base），常用来表示一块缓冲区(Buffer)的起始地址。

CX：记数寄存器（Counter），若用在移位指令、串处理指令和循环指令（LOOP）中，用来表示循环次数。

DX：数据寄存器(Data)，在双字乘、除法等指令中用来存放高位字，另外在端口操作指令 I/O 中，对于端口号大于 255 的端口操作，使用 DX 来存放端口地址。

SP：栈顶指针寄存器（Stack Pointer）。用来指示当前栈顶位置（即 SS:SP）。

BP：基址指针寄存器(Base Pointer)，常用来对堆栈中的数据进行寻址（栈段中的某个数据的偏移量，即 SS:BP）。而 SP 用来指示栈段的栈顶的位置（一个栈只有一个栈顶）。

SI：源变址寄存器(Source Index)，在字符串处理指令（如 MOVSB）中，用来表示预移动的源串的偏移地址（即 DS:SI）。

DI：目地变址寄存器(Destination Index)，在字符串处理指令（如 MOVSB）中，用来表示所移动的串的目标偏移地址（即 ES:DI）。在很多编译器中，SI 和 DI 常用来存放 C 语言中的寄存器变量。

注：在串操作指令中 SI，DI 内容可根据标志寄存器的 DF 位自动进行递减或递加操作。所以称 SI，DI 为变址寄存器。

DF=0 每次 SI，DI 递增 1，DF=1 每次 SI，DI 递减 1。

✓ **控制寄存器（专用寄存器）：**

IP：指令指针寄存器（Instruction Pointer），代码段偏移地址，用来指向下一条将被执行的指令的首地址（即 CS:IP）。

FLAG：标志寄存器，或称程序状态寄存器 PSW（Program Status Word）。其中的某些标志位反映指令运算结果的状态，某些标志位控制着指令的执行，某些标志位控制着系统的运行状况等。

不是所有的指令执行都影响标志位，如算术运算指令（如 ADD 等）和逻辑运算指令（如 AND）才可能改变 FLAG 中某些标志位的值（这些指令会自动修改 FLAG 中的某些标志位）。当然也可用某些指令（台 CLC、STC 等）强行修改 FLAG 中的某些标志位的值。

✓ **段寄存器：**详见 **8086 CPU 的内存分段**部分。

CS：代码段地址寄存器。
DS：数据段地址寄存器。
ES：附加段地址寄存器。
SS：堆栈段地址寄存器。

Intel 80x86 内部寄存器

寄存器的一般用法：

规则 1：若仅是存放数据用，可使用 AX、BX、CX、DX、BP、SI、DI 中的任何一个。不要使用其它寄存器存放数据，否则后果不可预测。在可能的情况下，可优先使用 AX（或 AH、AL）。在有些指令中，使用 AX 时指令的执行速度会快一些。

规则 2：若用寄存器表示内存**偏移地址**（此时须将内存地址写在[ ]中），**[ ]中只能使用 BX、BP、 SI、DI** 这四个寄存器。注在 386 以后的机型中可以使用所有通用寄存器进行偏移地址寻址。

例如：下列指令均合法：

```
MOV AX，[0100H]       ;ds:0100H
```

注：偏移地址也用立即数表示（在指令中能直接得到的数，叫立即数。常数即立即数）

```
MOV BX，0100H
```

```
MOV AX，[BX] ;ds:0100H
MOV DI，0100H
MOV [DI]，AX          ;ds:0100H
```

下列指令均非法：

```
MOV AX，[DX]
MOV [SP]，AX
```

规则 3：若[ ]中的偏移地址是经过运算得到的，则只能是如下模式：[基址±变址±disp]。其中 disp 是一个常数（即相对偏移,displacement）。[]中的任何一部分或两部分可省略（规则 2 实际上是规则 3 的特例）。**不允许出现同类性质寄存器的组合**，即[ ]中不允许出现[BX±BP±立即数] [SI±DI±立即数]这样的组合。

注：BX 和 BP 都是基址寄存器，属相同性质的寄存器；
SI 和 DI 都是变址寄存器，属相同性质的寄存器。
习惯上，[ ]中先写基址，然后写变址，最后写立即数。

例如：下列指令均合法：

```
MOV AX，[BX+DI]        ;ds:(BX+DI)或 ds:[BX+DI] 也可写做 MOV AX，[BX][DI]
MOV [BP+20H] ，AX      ;ss:(BP+20H)
MOV AX，[DI-20H]       ;ds:(DI-20H)
MOV AX，[BX+SI+20H]  MOV AX，20[BX][SI]       ;ds:(BX+SI+20H)
```

下列指令均非法：

```
MOV AX，[BX+BP+2]
MOV [SI-DI]，AX
```

规则 4：当表示内存偏移地址写在[ ]中时，BP 默认的是相对 SS 段寻址，而立即数、BX、SI、DI 默认的是相对 DS 段寻址。若[ ]中为一个寄存器加一个立即数，则按该寄存器的默认段地址寻址；若[ ]中为基址加变址的组合，则以基址为准寻址(与[ ]中书写顺序无关)。在指令中，可使用段超越前缀强行改变默认的段地址。
段超越前缀的表示方法是：段寄存器名：[偏移地址或寄存器名]。

例如，MOV AX,[BX]，操作数的物理地址为 DS:BX；
例如，MOV AX,[100H]，操作数的物理地址为 DS:0100H；
MOV AX,[BP]，操作数的物理地址为 SS:BP；
MOV AX,[BP+SI-2]，操作数的物理地址为 SS:BP+SI-2；
MOV AX,[SI-2]，操作数的物理地址为 DS:SI-2；
段超越前缀的例子：

```
MOV AX,SS:[BX]
MOV AX,ES:[BX]
MOV AX,DS:[BP+DI-2]
MOV AX,CS:[SI]
```

规则 5：下一条将被执行的指令永远位于 CS：IP 所指向的内存单元中。IP 永远相对于 CS 段寻址。

规则 6：当前的栈顶位置永远位于 SS：SP 所指向的内存单元中。SP 永远相对于 SS 段寻址。

规则 7：指令中永远不允许出现 IP 和 FLAG 两个寄存器的名字。

- **内存**

内存存放当前正在执行的程序和使用的数据，CPU 可以直接存取，它是相对于外存而言的，CPU 对外存(如软盘、硬盘或 CD-ROM、DVD-ROM 等)需要通过 I/O 接口访问。内存分只读存储器(ROM，Read Only Memory)和随机读写存储器(RAM，Random Access Memory)两种。

- **描述存储器容量的常用单位：**

1) 位 bit 0/1
2) 字节 Byte（通常所说的一个存储单元）

1 Byte=8 Bits

如：

位 7(MSB) 6 5 4 3 2 1 0(LSB)

0	1	0	1	0	0	0	1
0101 0001B=51H							

该字节内容：51H

3) 字 Word(x86 处理器中一个字的定义，其他处理器一个字不一定等于 2B)

1 Word=2 Bytes（高字节(High Order Byte)和低字节(Low Order Byte)）

如： MSB LSB

位 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	1	1	1	0	1	0	1	1	0	0	0	1	0	0	1
高字节：75H								低字节：89H							
0111 0101 1000 1001B=7589H															

4) 双字 Dword

1Dword = 2Word = 4Bytes

5) 千字节 KB、兆字节 MB、吉字节 GB

2^10B =1KiB= 1024 Bytes， 2^20B=1MiB=1024KiB=1048576 Bytes

2^30B =1GiB= 1024MiB，

2^40B =1TiB= 1024GiB

2^50B =1PiB= 1024TiB

2^60B =1EiB= 1024PiB

6) 其它：2^16B =64KiB =65536，2^32B=4GiB

- **8086 内存的存放次序**

字节顺序：

1) little endian：一个字的低有效字节存放在低地址空间，高有效字节存放在高地址空间；

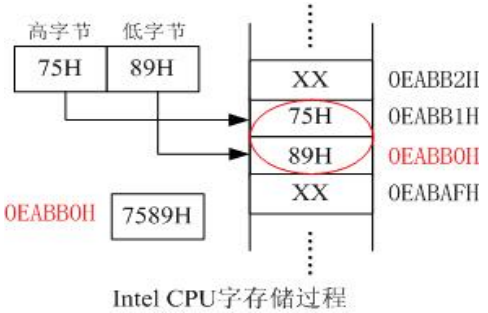
2) big endian：与之相反，采用 big endian 存放方式的 CPU 主要有 SPARC(主要用于 Sun 服务器中)、PowerPC(Motorola CPU)和 PARC(主要用在 HP 服务器中)，这种方式符合人的阅读习惯。

Intel 处理器采用的是 little endian 内存存放方式。

内存以字节为存放单元，对于字节信息按通常顺序依次的存放在内存中。对于字或双字，在该字（或双字）的内部，则按“倒字存放”原则存储（little_endian），具体如下：

对一个字，存储时先存放低字节，再存放高字节（即**低字节占低地址，高字节占高地址**）。**字的地址**是指其低字节的地址。

如将字 7589H 存放到内存地址 0EABB0H 地址中，其存放方式如下图所示：



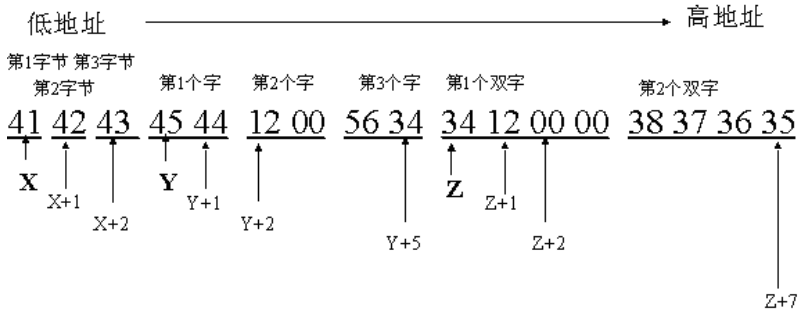
Intel CPU字存储过程

对一个双字，存储时先存放低字，再存放高字（即低字占低地址，高字占高地址）。**双字的地址**是指其低字的地址。在每个字的内部，同样按低字原则存放。

如将双字 14A8 F975H 存放到内存地址 0B75AH 处，内存中是如何存放的？此时读取 0B75AH 处一个字的内容应该是多少？

例如：

```
X     DB    “ABC”
Y     DW    “DE”，12H，3456H
Z     DD    1234H，“5678”
```



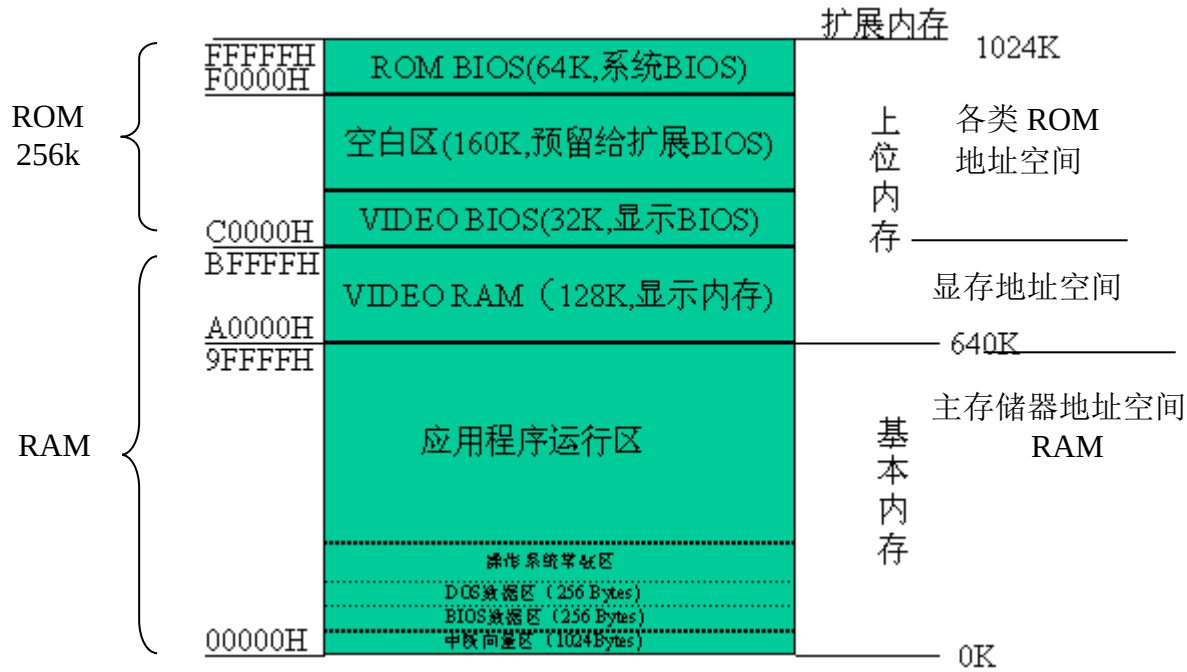
数在内存中的存放次序示意图

其中 X,Y,Z 为**符号地址**。

注意：无论是字节、字、双字，总体还是往上长的，所谓倒是指在每个字（或双字）的内部是倒的，但字与字（或双字与双字）之间还是顺序存放的。

在 Intel 体系中允许字或双字的存放采用非对准方式，即字地址不要求是偶数，双字地址不要求是 4 的倍数，但是非对准存放会影响数据存储效率，编程时尽量采用对准方式。而采用 Big Endian 的 CPU 体系中的存放方式只能采用对准方式，否则会发生存取异常。

8086 微机内存分布



系统硬件使用的内存位于地址区域的高端，从 A0000H～FFFFFH，其中 FFFF0H 地址是 PC 机启动时执行**第一条指令**，我们用 debug 工具查看，该处执行了一个无条件跳转指令，跳转到系统 BIOS 的初始化程序。

```
-u ffff: 0 0
FFFF:0000 EA5BE000F0     JMP      F000:E05B
```

CPU 通过内存地址访问指定的内存单元。以下是 PC 机中与内存相关的一些概念。

动态内存(DRAM)和静态(SRAM)：

RAM 主要有 DRAM 和 SRAM 两种类型。DRAM 的存储介质是一些小电容，需定时对小电容进行充电，以维持其中的数据不丢失，存在一个刷新周期的问题，只有当 DRAM 中的数据稳定后，CPU 才可读写 DRAM 中的数据，所以**DRAM 的工作速度很慢**。而 SRAM 是由一种称为触发器的逻辑电路构成，工作状态稳定，所以存取速度很快。但 SRAM 比 DRAM 价格高。另外，内存的工作速度一般以多少个纳秒（ns，10^{-9}s）来表示。表示时间的单位还有微秒（us，10^{-6}s），毫秒（ms，10^{-3}s）等。

CMOS RAM：

互补金属氧化物半导体存储器，简称 CMOS。是一种耗电极低的特殊 RAM。在 PC 机中一般为**128 字节**，用于保存 PC 的最重要参数，如时间、口令、硬盘类型等，启动时 BIOS 会自动读写 CMOS 中保存的参数。机器断电后，CMOS 中保存的数据可通过充电电池长期保存。PC 机的 BIOS 中有一段 CMOS 参数设置程序，可通过启动时按某个规定键（一般为 DEL 键）来运行 CMOS 参数设置程序。CMOS 中的数据也可通过端口地址进行读写。

Shadow RAM(影子内存)：

由于制造工艺上的原因，ROM 的工作速度较 RAM 慢很多，目前的 PC 一般在启动时自动将 ROM 中的内容复制到 RAM 中的某块特殊地方并对该块 RAM 自动进行写保护，称为 Shadow RAM。可在 CMOS 参数设置菜单中选择是否将 ROM BIOS 或 Video BIOS 或其它 BIOS 各部分进行 Shadow。

Cache（高速缓冲存储器）：

由于 RAM 相对于 CPU 来说，工作速度要慢很多，若 CPU 直接读写 RAM，CPU 要插入很多等待时间。为提高 CPU 的工作效率，目前的大多数 CPU 中，都在 CPU 与内存之间加了一层称为 Cache 的特殊存储器。Cache 的工作速度极高，与 CPU 的工作速度相当。位于 CPU 内部的 Cache 称为片内 Cache（Internal Cache，也叫一级 Cache 或 L1 Cache， 一般有几十 K）；位于 CPU 外部的 Cache 称为片外 Cache（External Cache，也叫二级 Cache 或 L2 Cache，一般有几百 K）。Cache 的工作原理是，每次 CPU 读写内存时，Cache 机构会自动将该内存地址相邻内存中的一批内容均读入到 Cache 中，这样，下次若 CPU 读写的是上次读写的相邻内存单元中的内容（称为预先读,Read Ahead），则 CPU 可直接从 Cache 中取得并返回。根据统计，从 Cache 中命中数据的概率是很高的。相反，CPU 写内存时，CPU 只要将数据写到 Cache 即可返回进行其它任务，由 Cache 机构在背后抽空将数据写入到 RAM 中（称为迟后写，Write Behind）。可在 CMOS 参数设置菜单中选择是否允许 L1 Cache 或 L2 Cache 工作。

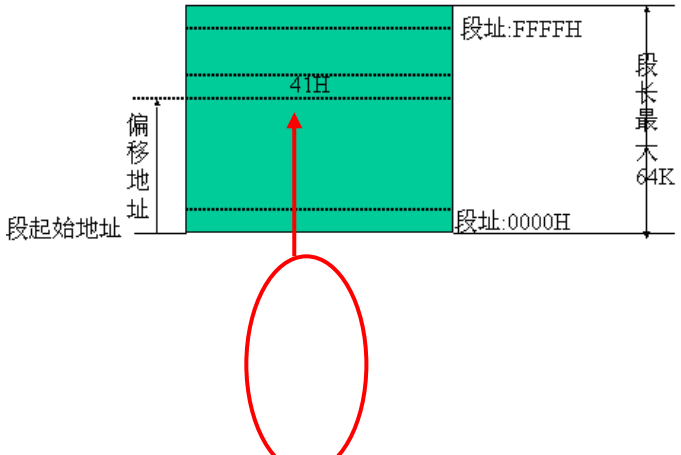
虚拟内存（Virtual Memory）：

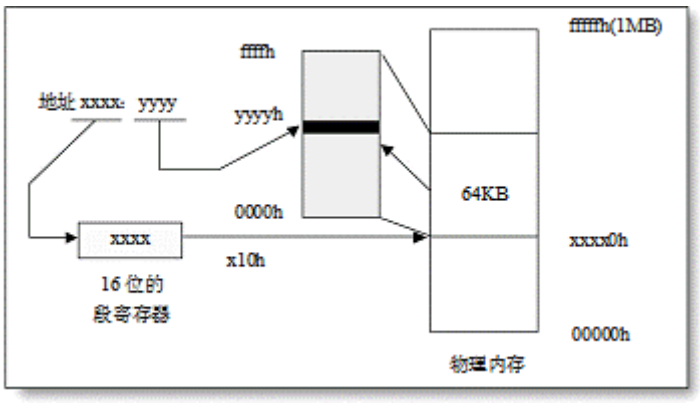
80386 以上的 CPU 可使用一部分硬盘空间模拟（或冒充）内存使用，称为虚拟内存。通过虚拟内存，可使用比实际内存大得多的内存。可在 Windows 控制面板/系统中设置允许使用的虚拟内存的大小。

8086 CPU 的内存寻址（实模式）（*重点）

✓ **段地址和偏移地址的概念：**

由于 8086 CPU 的寄存器只有 16 位，而地址线有 20 根，为使用 16 位的寄存器访问 20 位的内存地址（访问内存时，大多情况下需将内存地址放到寄存器中），8086 CPU 将 1M 内存分成多个段(Segment)。每个段有自己的段起始地址（简称段地址，16 位宽），每个段的最大长度为 64K。当我们访问一个内存单元时，必须告诉 CPU 所访问的内存单元的段地址以及该单元距段首的距离（多少字节），该距离称为段内偏移地址（简称偏移量，Offset，16 位宽）。离开段地址，孤立地谈偏移地址是没有任何意义的。在同一个段内，偏移量的最大变化范围是 0000H~FFFFH。





实模式下的内存寻址示意图

为表示方便，8086 规定，内存地址的表示方法为：**16 位段地址:16 位偏移地址**。其转换为 20 位物理地址的公式规定如下：

20 位物理地址=16 位段地址左移 4 位 + 16 位偏称地址

注：由于左移 4 位后，段地址的低 4 位一定是 0H，所以一个段的起始物理地址一定是 xxxx0H 的形式。

这里注意几个问题:

■ **同一物理地址的逻辑地址不唯一**。

例如逻辑地址 1A2B:5DF2 对应的 20 位物理地址是 200A2H：

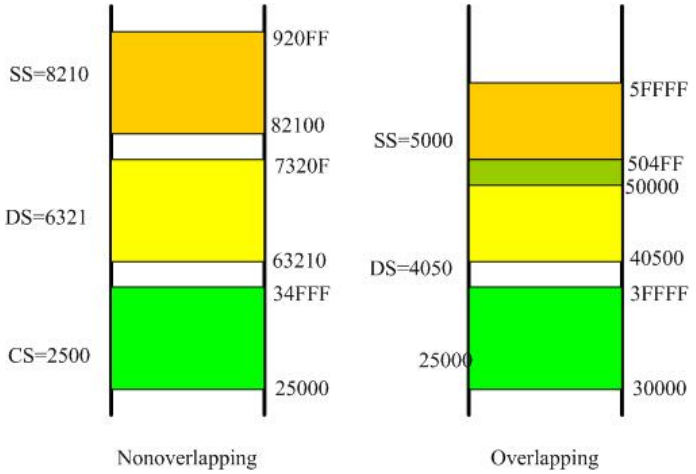
1A2BH->左移 4 个二进制位->1A2B0H

\+ 5DF2H

200A2H

显然，表示同一个物理地址时可以有近 64K 种段地址和偏移量的组合。例如 2000:00A2 和 200A:0002 都表示同一个物理地址 200A2H。

■ **端的最大长度为 64K,并不是每个段都能占用这么多空间**。



■ **段地址为 FF59 时，该段所能定位的空间范围是**?

一个应用程序可能使用 4 种类型的段，分别为：

代码段（Code Segment）：

用于保存程序代码的段。一个应用程序必须至少定义一个代码段。规定代码段的段起始地址必须放在代码段寄存器 CS 中，而下一条将被执行的指令的偏移量必须存放在指令寄存器 IP 中，所以下一条将被在执行的指令永远位于 CS:IP 所指向的内存单元中。

数据段(Data Segment)：

用于存放程序中用到的数据。一个应用程序的数据段是可选的。规定数据段的段起始地址必须放在数据段寄存器 DS 中。

栈段(Stack Segment)：见第一部分第五节

用于存放调用子程序主程序的返回地址等。一个应用程序可用的栈段是可选的。规定栈段的段起始地址必须放在栈段寄存器 SS 中，而当前栈顶的位置必须由栈顶指针寄存器 SP 表示。

附加段(Extra Segment)：

有时为程序设计方便（特别是字符串处理程序），可使用附加段。一个应用程序的附加段是可选的。规定附加段的段起始地址必须放在附加段寄存器 ES 中。

386 以后处理器增加了 FS、GS 附加段。

所有 x86 处理器在实模式下的最大寻址空间为 1MB。除专门指定，各段在存储器中的位置由 OS 负责，每段可独立占用 64KB 存储单元，但不是必须占用这么多。

寻址方式(Addressing Mode)，见原版教材的第 6 节。

保护模式下的内存寻址：

从 80386 开始，进入了真正的“保护模式”。386 是 32 位的 CPU，数据、地址总线都是 32 位，因此寻址空间为 4G。80386 中对内存的管理有两种，一个是段式内存管理，一个是页式内存管理。

先从段式内存管理说起。由于 Intel 是在 16 位 CPU 基础上设计 32 位 CPU 的，因此在 32 位处理器中，它继承了段寄存器。由于寻址空间的增大，16 位的段寄存器已不能够提供基地址了。这样就引入了一个数据结构来描述关于段的一些信息（即段描述符，它由段基址、界限、访问权、附加段构成）。当一个访存指令发出一个内存地址时，CPU 按照下面过程实现从指令中的 32 位逻辑地址到 32 位物理地址的转换：

1. 首先根据指令的性质来确定该使用哪一个段寄存器。
2. 根据段寄存器的内容(选择器—描述符的索引)，找到相应的“段描述符”。
3. 从“段描述符”中得到基地址。
4. 将指令中的地址作为位移，与段描述结构中规定的段长度相比，看是否越界；
5. 根据指令的性质和段描述符中的访问权限来确定是否越权；
6. 最后才将指令中的地址作为位移，与段基地址相加，得到物理地址。

同时，在上面过程中，由于有对访问权限的检查，就实现了保护。

80386 中有两个寄存器，分别是全局的段描述表寄存器(GDTR)和局部的段描述表寄存器(LDTR)，用来指向存储在内存中的某个段描述表。原段寄存器中的高 13 位指明某个段描述符在段描述表中的偏移，这个偏移加上 GDTR（或 LDTR）中段描述表的基地址，就得到段描述符的地址。最后从段描述符中得到段的 32 位基地址和其它的一些信息（这些信息包括关于越界和权限检查）。

段式内存管理只是 386 保护模式的一个部分，由于其效率的问题以及段是可变长度的，又发展出了页式内存管理。386 处理器中有一个寄存器 CR0，如果它的 PG 位为 1，就打开了页式内存管理。

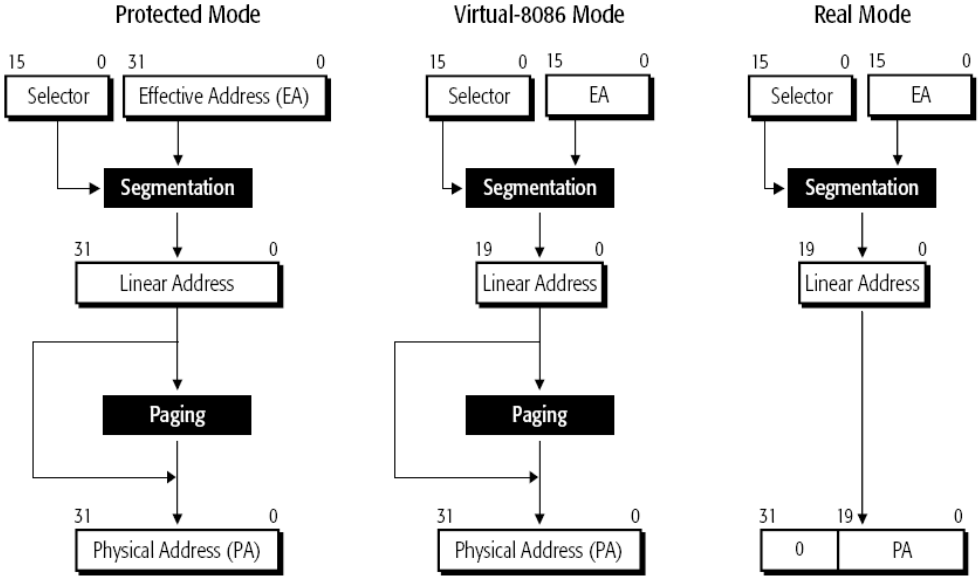
页式内存管理是在有段式内存管理形成的地址上再加上一层地址映射。此时由段式管理形成的地址就不再是物理地址了，而是“线性地址”。段式内存管理先把“逻辑地址”映射为“线性地址”，再由页式内存管理把“线性地址”映射为“物理地址”；当不使用页式内存管理时，“线性地址”就直接用作“物理地址”。

80386 把内存分为 4K 的页面，每一个页面被映射到物理内存中任一块 4K 字节大小的空间（边界必须与 4K 字节对齐）。需要注意的是，在段式管理中，连续的逻辑地址经映射后在线性空间还是连续的，但连续的线性地址经映射后在物理空间却不一定连续。

页式内存管理中，32 位的线性地址划分为三个部分：10 位的页目录表下标、10 位的页面表下标、12 位(4k)的页内地址偏移。CPU 增加了一个 CR3 寄存器存放指向当前页目录表的指针。寻址方式就改为：

1. 从 CR3 取得页目录表的基地址；
2. 根据 10 位页目录表下标和 1 中得到的基地址，取得相应页面表的基地址；
3. 根据 10 位页面表下标和 2 中得到的基地址，从页面表中取得相应的页面描述项；
4. 将页面描述项中的页面基地址和线性地址中的 12 位页内地址偏移相加，得到物理地址。

还有一个特例，就是所谓的“Flat（平坦）地址模式”，Linux 内核就是采用这种模式的。它是在段式内存管理的基础上，如果每个段寄存器都指向同一个段描述符，而此段描述符中把段的基地址设为 0，长度设为最大（4G），这样就形成了一个覆盖整个地址空间的巨大段。此时逻辑地址就和物理地址相同。形象的看，这样的地址就没有层次结构（段：偏移）了，所以叫做平坦模式，它是段式管理的特例。



三种模式下的内存寻址示意

- **总线（BUS）**

传输信息用的一组公共导线，其中每组导线中包含的导线根数称为总线宽度。一根导线对应一位二进制码。CPU 与存储器之间以及各外部设备之间都是通过这些总线进行通讯的。根据传送信息的不同，将总线分为：

- ✓ **地址总线 AB：**用来传送地址信号，地址总线是单向的。地址总线的宽度决定计算机的寻址能力（能访问内存的最大容量）。若地址总线宽度为 n，则该台计算机的寻址能力为 2^n 存储单元。
- ✓ **数据总线 DB：**用来传送数据，数据总线是双向的。数据总线宽度决定一台计算机传输数据的能力。若一台计算机数据总线的宽度为 n，则称该计算机为 n 位计算机。
- ✓ **控制总线 CB：**用来传输控制信号，控制总线是双向的。CPU 通过 CB 向内存及外设发出读写控制信号，相反，内存及外设通过 CB 向 CPU 报告自己的工作状态。

例如，指令 MOV AX，[0100H]的执行过程如下：
首先，CPU 将地址 0100H 放在 AB 上；
然后，CPU 通过 CB 向内存发读控制信号；
最后，内存将地址 0100H 处的数据放到 DB 上， CPU 便从 DB 上取数据到寄存器 AX 中。
再如，指令 MOV [0100H]，5 的执行过程如下：
首先，CPU 将地址 0100H 放在 AB 上；
然后，CPU 将数据 0005H 放到 DB 上；
最后，CPU 通过 CB 向内存发写控制信号，内存收到写控制信号后将 DB 上的数据放到 AB 上指定的的内存单元。

- **外部设备**

外部设备主要包括输入/输出设备和外部存储器等。外设与 CPU 和存储器的通讯是通过外部接口进行，每个接口有一组寄存器，寄存器组主要由以下三种组成：

- **数据寄存器：**用来存放通讯数据；
- **状态寄存器：**用来存放外设或接口的状态信息，使 CPU 了解外设工作状况；
- **命令寄存器：**用来存放 CPU 对外设或接口的控制命令。

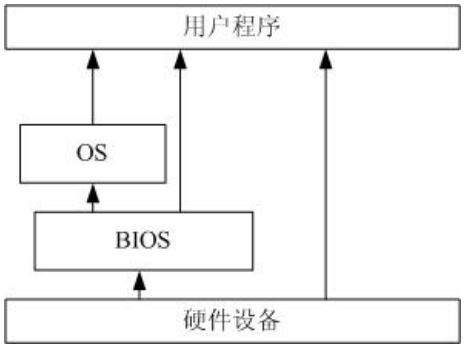
外设中的每一个寄存器被给予一个端口号。

端口(PORT)——I/O 地址空间

在计算机中，除了 CPU 中包含有寄存器外，其它芯片（外围芯片）中也含有大量的寄存器（有时也叫锁存器）。外围芯片中包含的寄存器称为端口。例如键盘控制器、中断控制器等芯片，以及位于插件卡（如显示卡）上的芯片中包含的寄存器等，都称为端口。端口也有编号，称为端口地址(I/O Base)。在 x86 系列 CPU 中，端口与内存地址都独立地分别从 0 开始编址，外部设备最多可有 65536 个端口，端口号为 0000～FFFFH。访问端口时需使用专门的的指令（IN/OUT）。有些 CPU（如 Motorola），端口与内存顺序编号，此时端口地址需占部分内存地址空间。

为了便于用户使用外设，80x86 提供了两种类型的例行程序可共用户调用，调用方式都是采用中断调用模式。

- BIOS 功能调用：存放在 ROM 中，可以看做硬件的组成部分，层次更低。
- DOS 功能调用：是 OS 的组成部分，开机时由磁盘装入存储器，它的例行程序可以调用 BIOS 来完成比 BIOS 更高级的功能。用户在使用外设时应尽量使用层次较高的 DOS 功能调用。



4) 机器指令的构成及指令的执行时间

- **机器语言**：计算机能够识别的语言，计算机是电子设备，他内部信号对应的是电平的高低和有如，所以机器语言是二进制的数，包括机器指令和数据。
- **汇编语言**：直接用机器语言编写程序是不现实的（全是 01 码字！），于是有了汇编语言，是机器语言的助记方式，是一种符号语言，便于理解和记忆。

那用汇编写的程序怎么让计算机执行呢？

将汇编指令翻译成机器指令的过程称为“**汇编**”（Assembly），其逆过程称为“**反汇编**”（Unassembly）。汇编一般通过工具 DEBUG（A 命令）或汇编器（如 TASM/MASM 等）进行，反汇编一般通过 DEBUG（U 命令）或专用工具（如 Sourcer 或 SoftICE 等）完成。即：

汇编源程序 xxx.asm(编辑)—>目标文件 xxx.obj(汇编)—>可执行文件 xxx.exe(链接)

在不同的机器指令中，不同的位表示不同的含义。例如可能有表示操作符码、操作方向码(d)、操作宽度码(w)、源操作数的寻址方式码(mod)、目的操作数和源操作数所用的寄存器码(reg)等。（见原版教材第 27 章“PC 指令集”或系统结构教程）。可查看 Complex/ Reduced Instruction Set Computer—CISC/RISC 相关知识。

与执行时间相关的概念：

- **时钟周期**：CPU 主频率的倒数，例如 8086 的主频为 4.77MHz（CPU 每秒工作 4.77 百万个时钟步），其时钟周期为 210 纳秒（$1ns=10^{-9}s$）。
- **指令周期**：指行一条指令所需要的时钟周期数（可查表得到）。最快的指令可能只需 1 个时钟期（如 INC AX），而最慢的指令需 180 多个时钟周期。对通常的数据传输指令（如 MOV），一般使用寄存器操作数最快，使用立即数操作数次之，使用内存操作数最慢。注：寄存器操作数和立即数操作数都来自 CPU 内部，立即数操作数由指令缓存缓冲器提供，而内存操作数来自 CPU 外部。

汇编语言发展至今，主要由以下 3 部分组成：

- 汇编指令：机器语言的助记符，编译后有对应的机器码

如汇编指令： mov ax,bx
机器码： 1000 1001 1101 1000(49D8H)
操作： 将寄存器 bx 的内容赋给寄存器 ax

- 伪指令：没有对应的机器码，给汇编器用来汇编使用的
  如 assume cs:code 是告诉汇编器 code 段地址与 cs 相关联
- 其它符号：如+,-,[,],;等，由汇编器识别，没有对应的机器码

5) 汇编指令的一般格式

➢ **（标号：） 操作符(操作码) 操作数 1，操作数 2 （；注释）**

标号：符号地址，代表该指令首字节地址；
操作符(operation):要执行的操作；
操作数(operand):执行对象，可以是操作数本身，也可以是操作数地址或地址的一部分。有些指令带 2 个操作数，如 ADD AX，10 （AX：目的操作数 10：源操作数）
有些指令带 1 个操作数，如 INC AX
有些指令带 0 个操作数，如 PUSHF
x86 为了避免指令过长，规定双操作数中**只能有一个使用内存寻址方式**，所以经常要先将操作数送到寄存器中。

➢ **操作数的类型：**

立即数操作数(data or immediate operand)：指在指令中可直接得到的操作数(即常数)。
寄存器操作数(reg)：指存放在寄存器中的操作数。
内存操作数(mem)：指存放在内存中的操作数。

使用符号地址（即变量名）表示内存操作数：

✓ **用 DB 定义字节类型的数:**表示其后的每个操作数占用一个字节.

var1 DB “ABCDE$”
分别定义了 6 个字节：41H、42H、43H、44H、45H、24H。符号地址 var1 代表第一个**字节**的起始地址。此写法与以下写法等价：
var1 DB “A”， “B”， “C”， “D”， “E”， “$”
var1 DB “ABC”， “D”， “E”， “$”
var1 DB “A”， “BC”， “D”， “E$”
var1 DB 41H，42H，43H，44H，45H，24H
var1 DB 41H，“BC”，44H，45H，“$”

✓ **用 DW 定义字类型的数:**表示其后的每个操作数占用一个字.

var2 DW 41H，4243H，“E”，“FG”
分别定义了四个字：0041H、4243H、0044H、4546H。符号地址 var2 代表第一个**字**的起始地址。
var2 DW “ABC”写法错误（超过一字宽）。

✓ **用 DD 定义双字类型的数:**表示其后的每个操作数占用一个双字.

var3 DD 41H，“E”，“FG”，12345678H
分别定义了四个双字：0000 0041H、0000 0045H、0000 4647H、1234 5678H。符号地址 var3 代表第一个**双字**的起始地址。
var3 DD“ABCDE”写法错误（超过一双字宽）。

✓ **用 DF 定义字类型的数:**表示其后的每个操作数占用 6 个字节.
✓ **用 DQ 定义字类型的数:**表示其后的每个操作数占用 8 个字节.
✓ **用 DT 定义字类型的数:**表示其后的每个操作数占用 10 个字节.

详见 P121 4.2.4 数据定义及存储器分配伪操作

6) 寻址方式

寻址方式是指在指令中找到操作数的方式。
8086 提供的主要寻址方式有：

➢ **立即数寻址方式**：指令中的操作数是一个立即数。如 MOV AX，5 中的源操作数 5。

立即数寻址只能用在源操作数，不能用在目的操作数；常被用来给寄存器赋值，但是不能给段寄存器赋值。

- **寄存器寻址方式**：指令中的操作数是一个寄存器操作数。如 MOV AX，5 中的目地操作数 AX。

以下寻址方式都是**关于内存**操作数（且在除代码段以外）的寻址方式：

- **直接寻址方式**：如 MOV AX，[0100H]中的源操作数[0100H]。
- **寄存器间接寻址方式**：如 MOV [BX]，AX 中的目地操作数。
- **直接变址寻址方式**：如 MOV [SI-0100H]，AX 中的目地操作数。
- **基址加变址寻址方式**：如 MOV [BX+SI]，AX 中的目地操作数。
- **相对基址加变址寻址方式**：如 MOV [BX+DI-2]，AX 中的目地操作数。

关于内存操作数的寻址，关键是确定操作数的偏移地址（或有效地址 effective address－EA）。

有效地址 EA＝基址＋变址×比列因子＋偏移量

注：比例因子是 386＋等序列的寻址方式术语，其值可为 1，2，4 或 8。

7) 指令中出现的符号地址的含义

符号地址是用符号形式表示的一个内存地址，在生成机器码后，符号地址会被用实际的偏移值替代。

例如：

MOV AX，X 等价于 MOV AX，[X]取地址为 X 的内存单元的值放到 AX 中。（生成的机器码类似于 MOV AX，[****]形式）。

MOV AX，X+1 等价于 MOV AX，[X+1]取地址为 X+1 的内存单元的值放到 AX 中。

MOV AX，[X+BX]取地址为 X+BX 的内存单元的值放到 AX 中。

MOV X，AX 等价于 MOV [X]，AX 将 AX 值存放到地址为 X 的内存单元中。

MOV X+1，AX 等价于 MOV [X+1]，AX 将 AX 值存放到地址为 X+1 的内存单元中。

INC X-20，等价于 INC [X-20]，将地址为 X-20 的内存单元的值加 1。

执行 MOV X+1，0ABH 后，X+1 处的值 42H 被替换成 ABH。

执行 MOV AX，X+2 后，AX=4543H

执行 MOV Y+3，1234H 后，Y+3 处的值 00H，56H 被替换成 34H，12H。

第四节 简单的数据传输指令(MOV 指令)

MOV 指令（原版教材 97 页）

格式：MOV 目的操作数，源操作数

1) 类型限制

- **源、目的操作数不可同时为内存操作数**

```
如：mov [100],[bx]错误，
需改为  mov ax,[bx]
        mov [100],ax
  或改为    push [bx]
            pop [100]
```

- **源、目操作数不可同时为段寄存器**

```
如：mov es,ds 错误，
需改为    mov ax,ds
          mov es,ax
 或改为   push ds
          pop es
```

- **不可将一个立即数直接传给段寄存器**

```
如：mov ds,100H 错误，
```

需改为 mov ax,100H
mov ds,ax

2) 宽度限制

- **若源、目双方宽度均明确，则宽度必须一致**

  如：mov es,al 错误，而 mov es,ax 正确。

- **若一方宽度明确，另一方宽度不明，则按宽度明确方进行传输**

宽度不明的操作数包括立即数和内存操作数（不包括用 DB、DW、DD 定义过类型的内存操作数）两种操作数，而寄存器操作数的宽度显然是明确的。

如： mov ax,12H 正确，相当于 mov ax,0012H。
mov [100H],AL 正确。

- **若双方宽度均不明，则出错**

此时可用 PTR 修饰符强行指明内存操作数的宽度（类似于 c 语言中的强制类型转换）。

PTR 修饰符的用法：

[BYTE|WORD|DWORD|SHORT|NEAR|FAR] PTR 内存操作数

其中，只有前三种类型能用于 MOV 指令中。

如：mov [100H],5 错误，

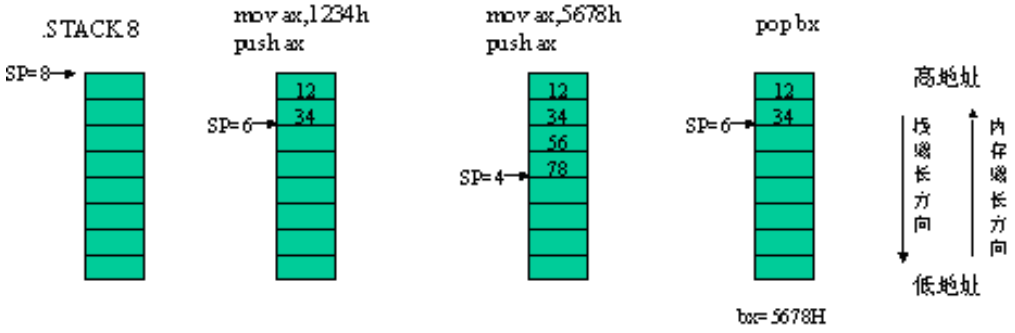
需改为 mov ax,5
mov [100H],ax

更好的改法为：mov word ptr [100H],5

第五节 栈(原版教材 23 页)

1) 栈结构

栈(Stack)是一块内存区，对该块内存区中的数据必须按**后进先出**（LIFO）原则进行存取。栈的一端是固定的，称为**栈底**(bottom)，栈的另一端是浮动的，称为**栈顶**(top)。在 x86 中，当前栈顶位置由 SP 寄存器指示。对栈的存取只能在栈顶进行。将一个数存入到栈顶的动作称为压栈操作（push），从栈顶取出一个操作数的操作称为弹栈（pop）。8086CPU 的栈操作都是以**字**为单位进行的。



8086CPU 只知道栈顶在何处(SS:SP)，而不知道栈空间的大小，用户必须自己控制栈的超界问题，即栈满时不能再执行 push，栈空时不能再执行 pop。

2) 栈的主要用处

1. 调用子程序时，保存主程序的返回地址。见下面对 CALL/RET 指令的解释。
2. 保护/恢复现场要

```
PUSHF
PUSH AX
PUSH BX
PUSH DS
PUSH ES
……
    CALL 子程序
```

……
POP ES
POP DS
POP BX
POP AX
POPF

3. 数据交换
如 MOV DS,CS 非法，可改为
PUSH CS
POP DS
又如 MOV [BX],[100H]非法，可改为
PUSH [100H]
POP [BX]

3) 栈操作的主要指令

1. PUSH 指令

格式：PUSH 源操作数

限制：源操作数必须是一个 16 位（机器字长）的操作数（压栈/弹栈只能以字为单位，并且不能是立即数）。

PUSH 指令执行过程：

第一步：SP-2 -> SP

第二步：源操作数入栈

例如：

PUSH AL 错，应改为 PUSH AX

PUSH 100 错，应改为 MOV AX，100
PUSH AX

作用：执行 PUSH 时，CPU 自动先将 SP 减 2，然后将源操作数放入到栈顶。即相当于：
SUB SP,2
MOV BP,SP
MOV [BP]，源操作数

注意，由于栈是内存的一块，所以压栈的字在栈中同样遵循倒字存放原则。

2. POP 指令

格式：POP 目地操作数

限制：目地操作数必须是一个 16 位的操作数，同样也不允许使用立即数，并且不允许使用 CS 寄存器。

POP 指令执行过程：

第一步：将栈顶一个字长数据弹出到目的操作数

第二步： SP+2 -> SP

例如：POP AL 错，应改为 POP AX

作用：执行 POP 时，CPU 先将栈顶的一个字存放到目地操作数中，然后自动将 SP+2，即相当于：
MOV BP，SP
MOV 目的操作数，[BP]
ADD SP，2

3. PUSHF 指令

格式：PUSHF

作用：将 SP-2，并将标志寄存器的当前值（16 位）压入栈中。

4. POPF 指令

格式：POPF
作用：将栈顶一个字弹到标志寄存器中，并将 SP+2。
例如：将标志寄存器的 PF 和 AF 复位（即清 0）（见原版教材 17 页）

```
PUSHF
POP AX
AND AX，0FFEBH （即 1111 1111 1110 1011B）
PUSH AX
POPF
```

4） 影响栈的指令

1．CALL/RET 指令对栈的影响 （原版教材 126 页）

子程序调用的一般格式：

```
主程序：
                ……
CALL [NEAR PTR 或 FAR PTR] 子程序
WWXX:YYZZ       ……

子程序：
        ……
        ……
        RET（或 RETF）
```

CALL 的执行过程如下：

当执行 CALL 指令，若 CALL 的是一个远过程，则 CPU 会**自动**将主程序的返回地址中的段地址（即 CALL 下一条指令的段地址 WWXX）压入栈中，然后将返回地址中的偏移地址（即偏移地址 YYZZ）压入栈中，然后跳转到子程序中执行。即相当于：

```
PUSH WWXX
PUSH YYZZ
```

远跳转指令也有类似操作，JMP FAR PTR 子程序

若 CALL 的是一个近过程，则只压偏移地址到栈中。CALL 不能实现短转移。

RET 的执行过程如下：

当执行到子程序中的 RETF 指令时，CPU 会自动将当前栈顶的一个字弹到 IP 中，然后再将栈顶一个字弹到 CS 中。这样就能达到返回到主程序中 CALL 下一条指令继续执行主程序剩余指令的目地。即相当于：

```
POP IP
POP CS
```

若为近返回（RETN），则只弹一个字到 IP 中。

2．INT/IRET 指令对栈的影响

中断服务程序最后必须以 IRET 结束。

INT n 指令相当于：

```
PUSHF
CALL DWORD PTR 0:[n*4]
```

IRET 指令相当于：

```
RETF
POPF
```

第六节 DEBUG（原版教材第 3 节，29 页）

Debug 是 DOS 操作系统提供的一个程序。

1) 主要功能

- 观察/修改寄存器中的值（**R 命令**）
- 往内存中输入/修改指令（**A 命令**）、显示内存中的指令（**U 命令**）
- 往内存中输入/修改信息（**E 或 F 命令**）、显示内存中的信息（**D 命令**）
- 在内存中对数据或指令进行移动（**M 命令**）或拷贝（**C 命令**）
- 在内存中单步执行指令（**T 或 P 命令**）、全速执行指令（**G 命令**）
- 将一个文件从磁盘上加载到内存中（**L 命令**）；
- 建立一个非 EXE 文件（**W 命令**）；
- 将磁盘的逻辑扇区读入到内存中（**L 命令**），或将内存中的数据写入到磁盘逻辑扇区中（**W 命令**）；
- 将从端口读数据（**I 命令**），或将数据写入到端口（**O 命令**）。

2) DEBUG 的一些约定

- 只能为 16 进制数，并且不带后缀 H（区别于会变源程序）；
- 不能出现伪指令或符号地址（若需使用变量名、标号、过程名，必须直接写其地址）。
- 若想中止当前的 DEBUG 操作，请按 **CTRL+C** 键
- 若命令中未指明段地址，默认为 DS 段。若命令中写明段地址，则只能是如下模式：

段寄存器名：偏移值，或 段值：偏移值。

其中偏移值只能是一个具体值，不能是寄存器名。

如-U 0 正确，-U CS:0 正确，-U ffff:0 也正确，但-DSS:SP 错误。

- 地址范围表示方式有两种：

方式一：起始地址 结束地址

方式二：起始地址 L 为字节长度

如-D 0:400 4FF 或 -D 0:400 L 100

都表示显示 0:400 到 0:4ff 内存单元内容。

- 表示地址范围时，只能在起始地址处写段值，结束地址不能再指定段值（即只能在同一个段中）

如-D 0:400 0:4FF 错，-D DS:400 DS:4FF 也错

- 刚进入 DEBUG 时，大多数命令默认的起始偏移地址均为 100H，以后输入的命令若未指定起始地址，默认均从上次结束地址处继续。若不想从上次结束处继续，应在命令后写明起始地址。

第二部分　汇编语言基础（Part B）

第七节 汇编程序上机的一般步骤（原版教材第 5 节）

第一步：编辑源程序。

可用记事本或 EDIT 等编辑软件建立.ASM 源程序。注意，汇编源程序的扩展名应为.ASM。这里假设我们建立的源程序名为 TEST.ASM。

第二步：汇编源程序（相当于 C 语言中的程序编译阶段）

Mcriosoft 的 MASM 汇编器各版本的区别：

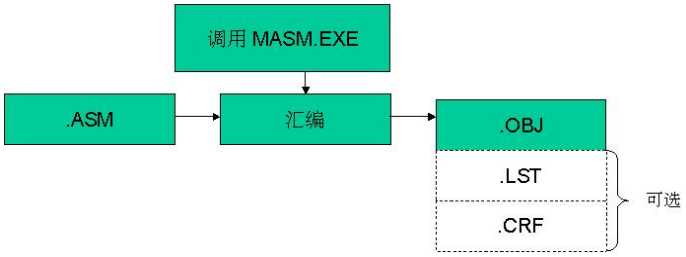
版　本	简　介
MASM 4.00	这是最先广泛使用的一个MASM版本，适用于DOS下的汇编编程。它很精巧，但使用起来不是很智能化，需要用户自己一板一眼地写出所有的东西。很多教科书上讲的8086汇编语法都是针对这个版本的，对程序员来说，它只比用Debug方便一点点
MASM 5.00	MASM 5.00比4.00在速度上快了很多，并将段定义的伪指令简化为类似 .code与.data之类的定义方式，同时增加了对80386处理器指令的支持，对4.00版本的兼容性很好
MASM 5.10	对程序员来说，这个版本最大的进步是增加了对@@标号的支持。这样，程序员可以不再为标号的起名花掉很多时间。另外，MASM 5.10增加了对OS/2 1.x的支持
MASM 5.10 B	1989年推出，比上一个版本更稳定、更快，它是传统的DOS汇编编译器中最完善的版本
MASM 6.00	1992年发布，有了很多的改进。编译器可以使用扩展内存，这样可以编译更大的文件，可执行文件名相应从Masm.exe改为Ml.exe。从这个版本开始可以在命令行上用*.asm同时编译多个源文件，源程序中数据结构的使用和命令行参数的语法也更像C的风格。最大的改进之一是开始支持 .if/.endif这样的高级语法，这样，使用复杂的条件分支时和用高级语言书写一样简单，可以做到几千行的代码中不定义一个标号；另外增加了invoke伪指令来简化带参数的子程序调用。这两个改进使汇编代码的风格越来越像C，可读性和可维护性提高了很多
MASM 6.00 A	未发售的版本
MASM 6.00 B	最后一个支持OS/2的MASM版本，修正了上一版本中的一些错误
MASM 6.10	修正了一些错误，同时增加了/Sc选项，可以在产生的list文件中列出每条指令使用的时钟周期数
MASM 6.10 A	1992年发布，修正了一些内存管理方面的问题
MASM 6.11	1993年11月发布，支持Windows NT，可以编写Win32程序，同时支持Pentium指令，但不支持MMX指令集
MASM 6.11 C	1994年发布，增加了对Windows 95 VxD的支持
MASM 6.12	1997年8月发布，增加 .686，.686P，.MMX声明和对相应指令的支持
MASM 6.13	1997年12月发布，增加了 .K3D声明，开始支持AMD处理器的3D指令
MASM 6.14	这是一个很完善的版本，它在 .XMM中增加了对Pentium III的SIMD指令集的支持，相应增加了OWORD（16字节）的变量类型
MASM 6.15	2000年4月发布

可使用 Borland 公司的汇编器 TASM 或 Microsoft 公司的宏汇编工具 MASM 完成此工作。命令格式如下：

TASM(或 MASM) [开关] 源文件[，目标文件] [，列表文件] [，交叉引用文件]

其中输入的源文件默认扩展名为.ASM，所产生的目标文件扩展名为.OBJ，所产生的列表文件扩展名为.LST（可选），所产生的交叉引用文件扩展名为.CRF 或.XRF（可选）。

若成功汇编，汇编器将产生目标文件(.OBJ)。若程序有错误，则汇编会报错，此时需修改源程序，然后重新进行汇编(但”No stack”属正常报警)。



汇编示意图

例如：

C>TASM↓ 获得帮助

C>TASM TEST↓ 产生 TEST.OBJ

C>TASM TEST,,↓ 或 TASM TEST /1↓产生 TEST.OBJ 和 TEST.LST

C>TASM TEST,,,↓产生 TEST.OBJ、TEST.LST 和 TEST.XRF

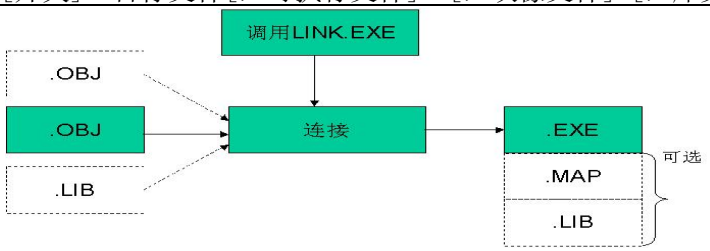
注：若使用 MASM，应先设置 C>PATH C:\MASM\BIN↓，否则可能会报错。并可在文件名后跟一个分号，以免出现提问，可直接得到 obj 文件。

- **目标文件**：目标文件只是一种半成品，虽然其中已包含机器码，但由于其中也包含一些尚无法确定最终值的符号（如段值、需重定位的项，如标号等），所以目标文件不能运行，需再经过连接才能生成可执行程序。
- **列表文件**：其中包含汇编指令与机器指令的对照表，以及段、符号的总结表。
- **交叉引用文件**：包含在汇编代码中使用的所有符号的列表。

第三步：生成可执行程序

可使用 TLINK（BORLAND）或 LINK（MICROSOFT）完成此过程。连接是一个装配过程，连接后将生成可执行程序.EXE 或.COM。命令格式如下：

TLINK(或 LINK) [开关] 目标文件[，可执行文件] [，映像文件] [，库文件]



链接示意图

其中目标文件和库文件为输入文件，生成的可执行文件默认为.EXE 文件，映像文件为生成的副产品（可选）。

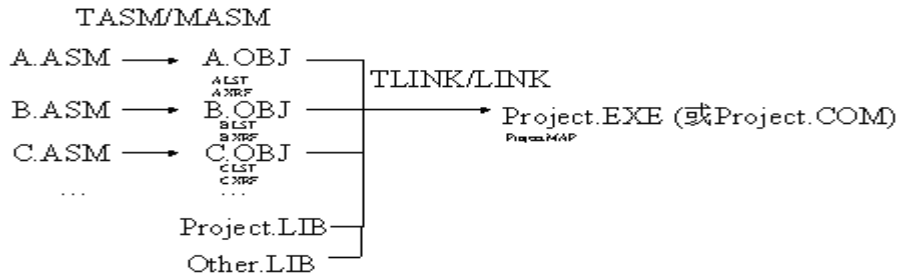
C>TLINK↓ 获得帮助

C>TLINK TEST↓ 产生 TEST.EXE

C>TLINK TEST /t ↓ 产生 TEST.COM（可能会报错）

C>TLINK TEST,,,↓或 TLINK TEST /m↓生成 TEST.EXE 和 TEST.MAP

注：若使用 LINK，可在命令最后跟一个分号，以免出现提问。



汇编与连接过程示意图

第四步：调试可执行程序

C>DEBUG TEST.EXE↓

一般先用 U 命令反汇编，用 P 或 T 命令跟踪单步执行，找程序的执行错误。

第五步：在命令行执行可执行程序

DEBUG 调试无错后，便可在命令行中执行。

第八节 汇编程序的结构（原版教材第 4 节）

1) 汇编程序风格

目前常用的有 Intel（本资料及课程使用该风格）和 AT&T 风格

- Intel 风格

  DOS 及微软等平台多用此风格；
- AT&T 风格

  LinuxS 及 Unix 等平台多用此风格；

  两种风格有很多细节差异，这里不赘述，可网补。

2) 程序格式

x86 汇编程序书写时，分经典格式与简化格式两种。

以下程序功能是在屏幕上显示字符串“Hello World”。我们以两种格式分别实现。

- **经典格式（Conventional Segment Directives）的样板程序**

  （假设程序名为 TEST.ASM）

```
STKSEG SEGMENT STACK
  DW 32 DUP(0)
STKSEG ENDS

DATASEG SEGMENT
  MSG DB”Hello World$”
DATASEG ENDS

CODESEG SEGMENT
  ASSUME CS:CODESEG, DS:DATASEG, SS:STKSEG
MAIN PROC FAR
  MOV AX,DATASEG
  MOV DS,AX
  MOV AH,9
  MOV DX,OFFSET MSG
```

```
        INT 21H
        MOV AX,4C00H
        INT 21H
    MAIN ENDP
    CODESEG ENDS
        END MAIN
```

substitute

- **简化格式(Simplified Segment Directives)的样板程序**

```
.MODEL SMALL
.STACK 64
.DATA
MSG DB”Hello World$”
.CODE
MAIN PROC FAR
        MOV AX,@DATA
        MOV DS,AX
        MOV AH,9
        MOV DX,OFFSET MSG
        INT 21H
        MOV AX,4C00H
        INT 21H
MAIN ENDP
        END MAIN
```

编辑完源程序后，用以下命令分别汇编和连接：

```
C>TASM  TEST↓
C>TLIKN TEST↓
```

即可生成 TEST.EXE。

3) 汇编程序的结构

- **程序由段组成**。程序至少必须有一个代码段，但数据段和栈段可有可无。
- **代码段中可由 0 个或多个过程（PROC）**。
- **程序最后必须有一条 END 伪指令，指示汇编器程序代码到此结束**。

  END 后可跟一个标号或过程名，表示程序的入口地址(Entry Point)，即程序从何处开始执行。

  格式：END [过程名|标号名]
- **大多数的简单程序可不写栈段，此时汇编时会警告“No Stack”（属正常现象）**。

  可根据程序中同时使用的 PUSH 个数，及子程序调用的深度决定定义的栈段的长度。
- **程序中定义的符号地址（变量）一般应放在数据段中**。

  数据段一般应放在代码段之前（变量先定义后使用）。

4) 对程序的解释

- **栈段的定义：**

对经典风格程序，凡 SEGMENT 后写有组合类型名 STACK 的段便自动成为栈段。由于栈段中的数据一般不通过符号地址访问，所以栈段中的变量可不必起名。对简化风格，若未指定栈段大小，**默认栈段大小为 1024 字节**。

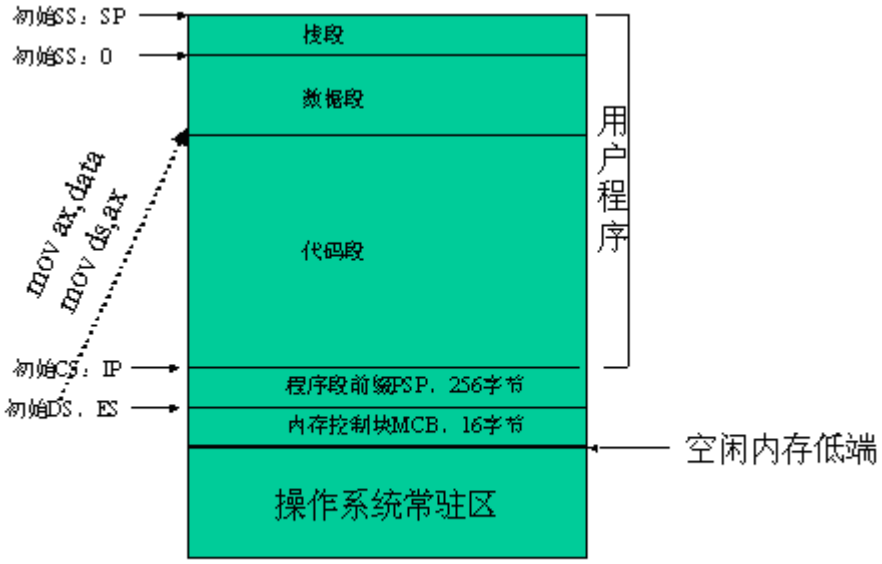
- **数据段的定义：**

特别注意，数据段除定义外，还需在程序中**显示地将数据段的段地址存放到 DS 寄存器中**。注意写法的不同：

经典风格：MOV AX，DATA

```
        MOV DS，AX
简化风格：MOV AX，@DATA
        MOV DS，AX
```

之所以需显示地将数据段地址用指令移到 DS 中，是因为 DOS 在加载一个可执行程序时，初始时会自动将 DS、CS 等的值设为**程序段前缀**的段地址（PSP，Program Segment Prefix，是一个 256 字节的重要数据结构，每个程序都有自己独立的 PSP，相当于进程标识符 PID，PSP 是在程序加载时由操作系统自动为应用程序建立的）。所以需用指令强行将被挪作他用的 DS 指向我们真正的数据段。



程序加载到内存中的映像

- **最常用的程序结束指令：**

```
1．使用 INT 21H
MOV AH，4CH
MOV AL，返回码          ；可选，也可不设返回码，正常返回一般设为“0”
INT 21H                 ；DOS 系统功能调用
```

相当于 C 语言的函数 exit(返回码)。

2．使用 INT 20H 结束

要求 CS 指向 PSP 的段基址，所以 INT 20H(DOS 中断返回)只能用于 COM 程序的结束或在 DEBUG 所写的指令。若将 INT 20H 用于 EXE 文件必须进行特殊处理，否则会死机。感兴趣的同学可以查看该连接 http://support.microsoft.com/kb/72848/en-us

3．程序作为过程被 DOS 调用时，此时需要在过程开始时将 DS 和 0 压栈，以便用 ret 指令返回 DOS。（其实就是 INT 20H 中断做的事情）如：

```
proname proc far
push ds
sub ax,ax ;xor ax,ax
push ax
……
ret
proname endp
```

PSP 的头两个字节是 INT 20H 指令，所以执行 ret 指令后将之前的 DS 和 00 分别给 CS：IP，及执行的就是 PSP 的头两条指令对应的 INT 20H。

- **代码段中也可不定义过程，而改用标号**。如下：

```
Start：MOV AX,@DATA
       MOV DS,AX
```

```
MOV AH,9
MOV DX,OFFSET MSG
INT 21H
MOV AX,4C00H
INT 21H
END Start
```

- **对于简化风格，汇编时会自动产生相应的段名(可参看 LST 文件（汇编时可选择生成）)：**
  - ✓ 栈段的段名为 STACK；
  - ✓ 代码段的段名为_TEXT；
  - ✓ 数据段的段名为_DATA；

对于某些内存模式（可能生成多个代码段或多个数据段的内存模式），生成的代码段或数据段段名前会自动加程序模块名，如 TEST_TEXT，TEST_DATA。

- **内存模式（Memory Model）及远/近指针**

如果目标地址不在当前段中（地址范围超过 FFFFH），则目标地址称为**远地址**（或远指针）。跨段称为远，同一个段中称为近。数据和代码有远近之分。若为近指针，则只要用 16 位的偏移地址即可表示目标地址，程序代码短，执行快；反之，若为远指针，则生成的代码长（目标地址需用**段：偏移**表示），执行慢。

对于简化模式，必须用 model 伪指令指出所采用的**内存模式**。

- ✓ **微模式(tiny)：**
  代码、数据、栈均在同一个段中，程序总长度小于 65536-256-2 字节，若想生成 COM 文件，必须选用 tiny 模式。
- ✓ **小模式(small) ：**
  数据占 1 个段，代码占 1 个段；（最常用的一种模式）
- ✓ **紧凑模式(compact) ：**
  数据占多个段（远数据），但代码只占 1 个段；
- ✓ **中模式(medium)：**
  数据只占 1 个段，但代码占多个段（远代码）；
- ✓ **大模式(large)：**
  数据和代码均占多个段；
- ✓ **巨模式(huge)：**
  数据和代码均占多个段，并且静态数组可超过 64K 字节（一个段）。
- ✓ **平坦式(flat)：**
  Win32 下的内存模式。

一般用 small 模式可适合绝大多数情况。

- **过程的远近问题**

若程序是以 INT 21H 的 4CH 号功能结束，则过程定义为远近均可。即 MAIN PROC FAR 中的 FAR 可省略（默认为近过程）。另外，MAIN 不是保留字。

- **其它解释**

INT 21H 的 9 号功能
功能：显示一字符串
输入参数：AH=9
DS:DX=欲显示字符串的段地址:偏移地址（该串以“$”结束）
返回参数：无

第九节 指令类型

指令分为（真）指令、伪指令、宏指令三种类型。

1) （真）指令

能产生机器码的指令叫（真）指令。一种 CPU 的（真）指令条数是一定的，如 MOV 等。学习汇编语言，主要是学习（真）指令。

2) 伪指令

不能产生机器码的指令，如 SEGMENT，PROC，DB，END 等。伪指令尽管不能产生机器码，但却是写汇编程序（.ASM）不能少的。伪指令的作用是指示汇编器（TASM 或 MASM）如何正确地翻译汇编源程序，以生成正确的机器码。

3） 宏指令

宏指令也是一种伪指令，只是将高级语言的某些特性融入到汇编程序中，如条件汇编、定义宏（可带输入参数）等。宏指令的目地是为了提高汇编程序的开发效率。

第十节 数据定义伪指令（原版教材 65 页）

可用 DB、DW、DD 伪指令分别定义**字节**类型、**字**类型、**双字**类型的变量。

一般用法：

符号地址 {DB|DW|DD} {? | 初值列表 | DUP 表达式 | 变量名}

- **格式 1：**？

作用：只预留内存空间，不赋初值。

例如：

```
X DB ?
Y DW ?,?,?
```

- **格式 2：**初值列表

作用：预留内存空间，并且赋初值。

例如：

```
X DB 2
Y DW 3,’A’,5
```

- **格式 3：**DUP 表达式，即：重复次数 DUP （初值）

作用：定义连续多个内存单元。

例如：

```
X  DB  20 DUP （’A’）
Y  DW  100H  DUP (?)
Z  DD  5  DUP  (10 DUP (41H,42H), 43H,44H)
```

- **格式 4：**变量名（在下节有说明）

符号地址 DW 变量名

符号地址 DD 变量名

作用：如果为 DW，表示取该变量的偏移地址作为符号地址的初值；如果为 DD，表示取该变量的段地址和偏移地址作为符号地址的初值（存储格式同中断向量，即偏移地址在前，段地址在后）。

例如：

```
X DB 2
我们假设 X 的地址为 1234H：5678H。
Y DW X （相当于 Y DW 5678H）
Z DD X （相当于 Z DD 12345678H）
```

例子，求 Fibonacci 数(斐波纳契数列：一种整数数列,其中每数等于前面两数之和)，例如，1,1,2,3,5,8 (See question 8-4, Page. 137)

```
COUNT EQU 10
.model small
.data
       Fib DB 1,1,COUNT dup (?)
.code
MAIN PROC
```

```
        mov ax,@data
        mov ds,ax

;求 10 个 Fibonacci 数
        MOV CX,COUNT
        MOV SI,0
ADDNEXT:
        MOV AL,[Fib+SI]
        ADD AL,[FIB+SI+1]
        MOV [FIB+SI+2],AL
        INC SI
        LOOP ADDNEXT

;以下为显示 Fibonacci 数的结果
        MOV CX,COUNT
        MOV SI,0
        LEA SI,Fib
DISPNEXT:
        MOV AL,[SI]
        CALL WRITE_AL
        MOV AH,2
        MOV DL,0DH ;输出“回车”
        INT 21H
        MOV DL,0AH ;输出“换行”
        INT 21H
        INC SI
        LOOP DISPNEXT

;程序结束
        MOV AX,4C00H
        INT 21H
MAIN ENDP

END MAIN
```

第十一节　　取得段地址和偏移地址的方法（原版教材第 6 节）

一、取偏移地址的方法：

方法 1：DW 伪指令后跟变量名（上一节提到过）

例如：

```
        DATA_1   DB 4,6,2,3,7,8,6,9
        Add_16   DW DATA_1
```

Add_16 存放的就是 DATA_1 的偏移地址

方法 2：使用 OFFSET 伪指令

格式：OFFSET　符号地址

这是最常用的取偏移方法。

例如：

```
    X DB ?
    MOV AX，OFFSET X （生成机器码后，为 MOV AX，0100H 格式）
```

方法 3：使用 LEA 指令

格式：LEA　　寄存器名，符号地址（内存单元）

例如：

X DB ?
LEA AX，X （取 X 的**偏移**放到 AX 中，机器码为 MOV AX，[0100H]格式）

注意：OFFSET 取的偏移地址为静态的，在汇编阶段便确定了，适于单模块程序，此外 OFFSET 后只能使用简单的符号地址，而不能使用[基址＋变址＋偏移]这种方式。而 LEA 取偏移是动态的，在运行时刻临时取偏移，适于多模块连接的程序。用 LEA 取偏移永远不会出错。一般全局变量：用 offset 伪指令取偏移；局部变量：用 lea 指令。

二、取段地址的方法：

汇编程序中出现的段名，在生成可执行程序后将变成一个常数（即相对段值，从 0000，0001，0002，…..），将来在运行程序时，段地址的具体值会重新修正（由操作系统根据当时内存使用情况和 EXE 文件头中保存的相对段值临时确定）。

方法 1：DD 伪指令后跟变量名（上一节提到过）

例如：

DATA_1 DB 4,6,2,3,7,8,6,9
Add_32 DD DATA_1

Add_32 存放的就是 DATA_1 的偏移地址和段地址

方法 2：直接写段名

例如：

MOV AX,DATA （传统风格）
MOV AX,@DATA （简化风格）

方法 3：使用 SEG 伪指令

格式：SEGMENT 符号地址

这是最常用的方法。注意 SEGMENT 后只能跟符号地址（变量名、标号、过程名），不可写段名。因为**段名是一个立即数**，不是符号地址。

例如：

X DB ?
MOV AX，SEG X ；获取 X 所在段的段名

方法 4：用 LDS 或 LES 指令

格式：

LDS 寄存器名，双字类型变量名
LES 寄存器名，双字类型变量名

作用：将指定变量处存储的内容作为一个双字看，前一个字作为偏移值放到指定寄存器中，后一个字作为段值放到 DS（对 LDS 指令）或 ES（对 LES 指令）中。

例如：

X DD 0FFFF0005H
LDS BX，X （执行后：DS=FFFFH，BX=0005H）
MOV AL，[BX]

注意与 LEA 指令的不同，LEA 是取变量本身的偏移量（不是变量处存放的值），而 LDS 和 LES 则是直接取变量处存放的值作为段地址和偏移的值。

三、AT、ORG 伪指令及$符号（原版教材第 199,200 页）

1．AT 伪指令

格式： AT 段地址 （AT 一般写在 SEGMENT 后面）

作用：强行指定该段的（物理段）段地址。（默认情况是，汇编器汇编时按段的出现顺序，顺序分配相对段值，从 0000H、0001H、0002H…等）

例如：

```
INTVECTSEG    SEGMENT AT 0000H
INTENTRY DD 256 DUP（？）
INTVECTSEG    ENDS
```

```
BIOSDATSEG   SEGMENT AT 0040H
COMADDR DW 4 DUP（？）
LPTADDR DW 3 DUP（？）
BIOSDATSEG   ENDS
```

2．ORG 伪指令和$符号

汇编时，同一个段都有自己独立的偏移地址计数器，每个段内部的偏移都从 0 开始顺序分配，当前的偏移值规定用$符号表示。可用 ORG 伪指令强行改变当前的偏移值（即$的值）。

用法：ORG 新偏移值

例如：

```
    DATA SEGMENT
    $=0 ->          Var1 DB “ABC”
    $=3 ->          Var2 DW 1,2
    $=7 ->          ORG 20  （Var3 与 Var2 之间共有 13 字节的未使用空间）
    $=20 ->         Var3  DB “XYZ”
    $=23 ->         ORG $+5 (预留 5 字节空间)
    $=28 ->         Var4 DD ？
    $=32 ->
    DATA ENDS
```

显然，DATA 段共长 32 字节。

第十二节 EQU、=及 LABEL 伪指令

一、EQU 伪指令和等号

用法：

```
      符号名  EQU  值
      符号名 =  值
```

作用：为增强程序的可读性和可维护性，定义一个符号，代表后面的值（与 C 语言中的#define 完全等价）。用 EQU 定义的符号以后不可再定义，但用等号定义的符号则可重新定义。EQU 或等号定义的符号不占用任何内存空间（**汇编时**会将符号出现的地方用值进行**替换**，注意与 C 语言中“=”的区别。），这与 DB、DW、DD 完全不同。

例如：

```
    MAX EQU 100
    COUNT=10
    MOV AX，MAX
    MOV CX，COUNT
    MAX EQU 200 （出错）
    COUNT=20 （正确）
```

又如：

```
.DATA
      MSG DB “Hello World”
      LENOFMSG EQU $-MSG   (LENOFMSG 为 MSG 中包含的字节数)
      ARRAY DW 1,2,5
      COUNTOFARRY EQU ($-ARRAY)/2 （COUNTOFARRY 为 ARRAY 中字的个数）
```

二、LABEL 伪指令

用法：

```
      符号  LABEL {BYTE|WORD|DWORD|SHORT|NEAR|FAR}
```

作用：使同一个内存地址可用不同类型的符号表示，方便程序设计。用 LABEL 定义的符号不占用任何内存空间。

例如：

```
    W LABEL WORD
```

D LABEL DWORD
X DB 100 DUP （? ）
则以下指令均正确：
MOV AL，X （取字节数据）
MOV AX，W （在同一地址取字数据）
LDS BX，D （得该地址的偏移地址和段地址）

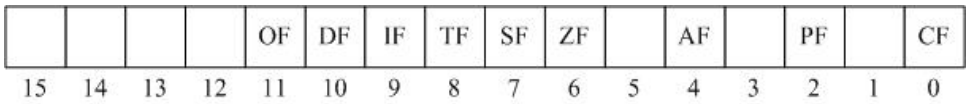
第十三节 标志寄存器（原版教材第 17，119 页）

FLAG 寄存器作用：

- 用来存储相关指令的某些执行结果
- 用来为 CPU 执行相关指令提供行为依据
- 用来控制 CPU 的相关工作方式

寄存器内存放的信息被称作**程序状态字 PSW**(Programm Status Word)。

8086/8088 的标志寄存器：



一、特殊标志位

1） 中断允许标志 IF：（bit 9）

IF=1 表示开放所有外部设备中断（EI），0 表示禁止外部中断（DI）。专用于中断服务程序中。

2） 陷阱标志 TF：（bit 8）

TF=1 表示使 CPU 工作于单步工作方式下，以便能使用 DEBUG 等调试软件。只有设计 DEBUG 等系统调试软件时才会用到 TF。

3） 方向标志 DF：（bit 10）

DF=0 表示按地址增加的方向移动字符串（UP），否则按地址减小的方向移动字符串（DN）。只对 MOVSB 等串处理指令有影响。

二、与运算相关的标志位

注意，只有运算指令才改变以下标志寄存器的值。

4） 全零标志 ZF：（bit 6）

ZF=1 表示本次运算结果为 0（ZR），否则为非 0（NZ）。

5） 符号标志位 SF：（bit 7）

SF 即运算结果的符号位（b7 或 b15）。SF=1 表示本次运算结果为负（NG），否则为正（PL）。

6） 辅助进位 AF：（bit 4）

AF=1 表示本次运算时，b3 向 b4 产生了进位（AC），即产生了**半字节**进位，否则无半字节进位（NA）。

7） 奇偶标志 PF：（bit 2）

PF=1 表示本次运算结果中有偶数个 1（PE），注意，不是运算结果为偶数的意思，否则有奇数个 1（PO）。

8） 进位标志 CF（bit 0）和溢出标志 OF（bit 11）

CF=1 在 DEBUG 中表示为 CY，0 表示为 NC。

OF=1 在 DEBUG 中表示为 OV，0 表示为 NV。

CF 描述的是**无符号数**运算结果的状态（当 b7 或 b15 向前产生进位/借位时，CF=1），而 OF 描述的是**有符号数**运算结果的状态。一个数既可当作无符号数，也可当作有符号数看待，CPU 不知道该当做什么类型的数，所以 CPU 只能同时将两种情况都反映出来，只有程序员才知道自己进行的是有符号数还是无符号数运算，**若是无符号数运算，则只应看 CF；若是有符号数运算，则只应看 OF**。只有无符号数才存在进位概念，只有有符号数才存在溢出概念，同时看 CF 和 OF 没有任何意义。

进位是正常现象，可通过指令将进位或借位找回来（如 ADC、SBB 指令等）；而溢出是一种错误，是程序员指令设计不当造成的（目的操作数宽度不够），只有两个同号数相加，或两个异号数相减少时才可能出现溢出（目的操作数放不下运行结果）。若 OF=1 表示有符号数的运算结果有

错，此时目的操作数中的运算结果无任何意义，并且无法找出正确结果。**进位和溢出两者之间无任何联系**。

例如：

MOV AL，81H（无符号数：129 或 有符号数：-127）

ADD AL，0FEH（无符号数：254 或 有符号数：-2 ）

执行后 AL=7FH，CF=1，OF=1。

本例 OF=1，所以若是有符号数运算，则 AL 中的结果无任何意义（错语）。

✓ **怎样判断 CF？**

此时，均应当作无符号数：

81H 即 129

FEH 即 254

相加结果为 383，超过一字节能表示的无符号数范围（0 ~ 255），所以 CF=1。

✓ **怎样判断 OF？**

此时，均应当作有符号数：

81H 即-127 （256-129=127）

FEH 即-2 （256-254=2）

相加结果为-129，超过一字节能表示的有符号数范围（-128 ~ +127），所以 OF=1。

又如：

```
MOV AL， 70H
ADD AL，0FEH
```

执行后 AL=6EH，CF=1（112+254=366），OF=0 （+112）+（-2）=（+110）。

此外，CF 位也可和移位指令配合使用。

9） 符号标志 SF：（bit 7）

纪录运算结果的符号，负为 1，正为 0。

三、与标志位有关的一些指令

指令：CLC

作用：使 CF=0

指令：STC

作用：使 CF=1

指令：CMC

作用：使 CF 取反

指令：CLD

作用：使 DF=0

指令：STD

作用：使 DF=1

指令：CLI

作用：使 IF=0 （关中断）

指令：STI

作用：使 IF=1 （开中断）

指令：LAHF

作用：将 FLAG 的低字节装入到 AH 中。

指令：SAHF

作用：将 AH 的值装入到 FLAG 的低字节中（修改 FLAG）。

指令：PUSHF

作用：将 FLAG 的值压入栈中。
指令：POPF
作用：将栈顶一个字弹到 FLAG 中（修改 FLAG）。

四、标志位符号表示

例如：
在 debug 中
-r
AX=0000 BX=0000 CX=0000 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=**** ES=**** SS=**** CS=**** IP=0100 **NV UP EI PL NZ NA PO NC**

NV UP EI PL NZ NA PO NC 都相应标志位的情况，具体可查下表：

标志位的符号表示

位	标志符号	标志位名称	标志为 1	标志为 0
0	CF	进位标志	CY	NC
2	PF	奇偶标志	PE	PO
4	AF	辅助进位标志	AC	NA
6	ZF	零标志	ZR	NZ
7	SF	符号标志	NG	PL
9	IF	中断标志	EI	DI
10	DF	方向标志	DN	UP
11	OF	溢出标志	OV	NV

五、用 DEBUG 观察标志寄存器

MOV AL，81H
ADD AL，0FEH
（用 P 或 T 命令执行 ADD 后：AL=7FH OV UP EI PL NZ NA PO CY）
SUB AL,AL
（用 P 或 T 命令执行执行 SUB 后：AL=00H NV UP EI PL ZR NA PE NC）

第十四节 算术运算指令（原版教材第 220 页）

一、加法指令

格式：
ADD 目，源
ADC 目，源
解释：ADC 称为带进位的加法，ADC=ADD+CF。一般使用 ADD 指令。若为多字节加法，则第一字节应使用 ADD，以后字节均应使用 ADC 相加，以便将前次产生的进位加到本次加法中。
例如：DX:AX+BX:CX=>DX:AX，正确指令为：

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

又如：

X DD 12345678，将 X 加 F000 的指令如下：具体操作时要进行内存数据的读写。

```
ADD WORD PTR X，0F000H
ADC WORD PTR X+2，0
```

二、减法指令

格式：
SUB 目，源
SBB 目，源

解释：SBB 称为带借位的减法，即 SBB=SUB-CF。一般使用 SUB 指令。若为多字节减法，则第一字节应使用 SUB，以后字节均应使用 SBB 相加，以便将前次产生的借位从本次减法中减去。

三、乘法指令

格式：
MUL 乘数　　（用于无符号数的乘法）
IMUL 乘数　　（用于有符号数的乘法）
解释：指令中给出的乘数可以是一个寄存器或内存单元，乘法指令按所给出的寄存器宽度，自动到 AL 或 AX 中找另一个乘数（按等宽原则）。乘法只能有以下两种模式：
8 位*8 位=16 位 （结果自动放于 AX 中）
16 位*16 位=32 位（结果自动放于 DX:AX 中）
例如：2*30H
```
MOV AL,2
MOV BL,30h
MUL BL
```
结果 AX=60H
又如：2H*9001H
```
MOV AX,2
MOV BX,9001H
MUL BX
```
结果 DX=0001H，AX=2002H

四、除法指令

格式：
DIV 除数　　（用于无符号数的除法）
IDIV 除数　　（用于有符号数的除法）
解释：指令中给出的除数必须是一个寄存器或内存单元，除法指令按所给出的寄存器宽度，自动到 AX 或 DX:AX 中找被除数（倍宽原则）。除法只能有以下两种模式：
16 位(AX)/8 位=8 位 （商放于 AL，余数放于 AH）
32 位((DX:AX)/16 位=16 位（商放于 AX 中，余数放于 DX 中）
例如：32H/3
```
MOV AX，32H
MOV BL,3
DIV BL
```
结果 AL=10H，AH=02H

对被除数的倍宽处理：对 8/8 或 16/16 的除法，应先对被除数进行位宽处理。对于无符号数除法，可直接将 AH（对 8/8）或 DX（对 16/16）清 0；但对有符号数，则应使用 CBW（对 8/8）或 CWD（对 16/16）对被除数进行补码的符号扩展，以保持倍宽后保持被除数的符号不变。

格式：CBW
作用：将 AL 的符号位扩展到 AH 的所有位。
例如：
```
MOV AL，92H
CBW
```
执行后 AX=11111111 10010010 B

格式：CWD
作用：用 AX 的符号位扩展到 DX 的所有位。
例如：
```
MOV AX，9001H
CWD
```

执行后 DX:AX=FFFF9001 H
例如：无符号数除法：101H/2
```
MOV AX,101H
MOV DX，0
MOV BX,2
DIV BX
```
执行后 AX=0080H，DX=0001H
又如：有符号数除法：0F1H/6 （即-15/6）
```
MOV AL,0F1H
CBW             ;AX=FFF1H
MOV BL,6
IDIV BL
```
执行后 AH=FDH（即-3），AL=FEH（即-2）

第十五节 逻辑运算（布尔运算）指令（原版教材第 127 页）

一、布尔运算指令

➢ **AND 指令（与）（类似 C 语言中的‘&’运算符）**

格式：AND 目，源
作用：

```
X AND 0=0
X AND 1=X
X AND X=X
```

（1）AND 主要用于将目的操作数中的某些位**强行清 0**（也叫复位 RESET，或称为屏蔽 MASK）。凡需清 0 的那些位用 0 去 AND，保持不变的那些位用 1 去 AND。

例如，设置中断屏蔽寄存器（端口地址为 21H），开放时钟中断和键盘中断。

中断屏蔽寄存器位于中断控制器（8259）中，为一个字节，其 b0 位对应时钟中断，b1 位对应键盘中断。1=禁止，0=允许

（在 Windows 中实验有效）

```
IN AL,21H
AND AL,11111100B
OUT 21H,AL
-D 40:6C （当前计时值）可通过查看该地址的信息，看时钟是否有走
```

（2）请分析下条指令的作用

```
AND AX，AX
JZ L
```

（3）拼数

➢ **OR 指令（或）（类似 C 语言中的‘|’运算符）**

格式：OR 目，源
作用：

```
X OR 0=X
X OR 1=1
X OR X=X
```

（1）OR 主要用于将目的操作数中的某些位**强行置 1**（也叫置位 SET）。凡需置 1 的那些位用 1 去 OR，保持不变的那些位用 0 去 OR。

例如，设置中断屏蔽寄存器，禁止时钟中断和键盘中断。

（在 Windows 中实验有效）

```
IN AL,21H
OR AL,00000011B
OUT 21H,AL （时钟被禁止，会发生什么？）
```

（2）请分析下条指令的作用

```
OR AX，AX
JZ L
```

（3）拼数

- **XOR 指令（异或）（类似 C 语言中的‘^’运算符）**

格式：XOR 目，源

作用：

X XOR 0-〉X 保持不变
X XOR 1-〉X 取反
X XOR X-〉X 清零

（1）XOR 主要用于将目的操作数中的某些位**强行置反**（翻转）。凡需翻转的那些位用 1 去 XOR，保持不变的那些位用 0 去 XOR。

例如，翻转大写指示灯的状态。

键盘状态字节位于 BIOS 数据区（地址 40：17H，详见附录 2），其 b6 位对应大写指示灯的状态(1=亮，0=灭)，见原版教材 186 页。（本代码只有在 DOS 中才有效）

```
MOV AX,40H
MOV DS,AX
XOR BYTE PTR [17H],01000000B
```

（2）快速清 0

```
XOR AX，AX
```

（3）字符串加密

设有明文 X，我们可选一合适的密钥 K，则执行 XOR X，K 后，X 将变成密文。若想解密，只要将密文再与同一个密钥 K 进行一次 XOR，则可恢复为明文 X。

（X XOR K）XOR K = X XOR （K XOR K）= X XOR 0 =X

示意性程序：

```
#define KEY 0x6AH
fp1=（FILE *） fopen(“InputFile”,“r”);
fp2=（FILE *） fopen(“OutputFile”,“w”);
while ((c=fgetc(fp1)!=EOF) fputc(fp2,c^KEY);
fclose(fp1);
fclose(fp2);
```

（4）在动画程序中，XOR_PUT 写模式可用于擦除前次遗留的画面。

- **NOT 指令（取反）（类似 C 语言中的‘~’运算符）**

格式：NOT 目

作用：

将目的操作数中的所有位均逐位取反。例如，NOT AL 与 XOR AL，0FFH 功能等价。

- **TEST 指令（测试）**

格式：TEST 目，源

作用：TEST 在内部执行时，模拟一次 AND 操作，但不真正改变目的操作数的值，只改变标志寄存器的值，可用于在不破坏目的操作数原值的情况下进行一些判断。TEST 专用于测试目的操作数中的某些位中是否至少有一位为 1（等价的说法：这些位不全为 0）。TEST 是通过截取目的操作数中的相应位来判断的。

例如，若左 Shift 键或右 Shift 键中的任何一个压下（地址 40：17H，详见附录 2），则跳转至 L。

```
MOV AX,40H
MOV DS,AX
TEST BYTE PTR [17H],00000011B
   JNZ L
```

问题 1：请写逻辑运算指令，将 AL 中的小写字符变为大写。
问题 2：请写逻辑运算指令，将 AL 中的大写字符变为小写。

二、基本移位指令

➢ 左移（类似 C 语言中的‘<<’运算符）

逻辑左移指令：SHL 目，移位次数
算术左移指令：SAL 目，移位次数
解释：对于左移，SHL 或 SAL 等价。被移掉的最高位将被自动放到 CF 中，最低位自动用 0 填补。当移位次数大于 1 时，必须事先将移位次数放于 CL 中。
用处举例：
（1）快速乘法：左移 n 位相当于乘以 2 的 n 次方。

```
例如：AL*10=>AL
SHL AL,1;AL*2
MOV BL,AL
MOV CL,2
SHL AL,CL ;AL*8
ADD AL,BL ;AL*10
```

（2）拼数

➢ 右移（类似 C 语言中的‘>>’运算符）

逻辑右移指令：SHR 目，移位次数
算术右移指令：SAR 目，移位次数
解释：对于逻辑右移，被移掉的最低位将被自动放到 CF 中，最高位自动用 0 填补。对于算术右移，被移掉的最低位将被自动放到 CF 中，最高位自动**用符号位填充**（以保持数的符号不变）。逻辑右移适于无符号数除法，算术右移适于有符号数除法。
用处举例：
（1）快速除法：右移 n 位相当于除以 2 的 n 次方。
（2）拼数

逻辑运算指令的综合例子：
1. 键盘输入两个 16 进制字符，要求将其合并到 AL 寄存器中。例如，若输入 ‘2’和‘5’两个字符，则使 AL=25H）。

```
READ_AL PROC
; 作用：键盘读入两个 0-9 之间的字符，返回在 AL 中
; 说明：本过程不会修改除 AL 以外的其它寄存器的原值
PUSHF
PUSH CX
PUSH DX
MOV DH，AH    ；保存 AH 的原值到 DH 中

MOV AH,1
INT 21H       ;AH=1:键盘输入并回显,键盘输入’2’后 AL=32H
MOV CL,4
SHL AL,CL     ;AL=20H
MOV DL,AL     ;DL=20H
INT 21H       :AH=1:键盘输入并回显,键盘输入’5’后 AL=35H
AND AL,0FH    ;AL=05H
OR AL,DL      ;AL=25H

MOV AH，DH    ；恢复刚进入本过程时的 AH 原值
POP DX
```

```
POP CX
POPF
RET
READ_AL ENDP
```

2. 在屏幕上显示 AL 的值。例如若 AL=25H，则显示 25。

```
WRITE_AL PROC
; 作用：显示 AL 中的值（假设在 0-9 之间）
; 说明：本过程不会破坏所有寄存器的原值
PUSH AX
PUSH CX
PUSH DX
PUSHF

PUSH AX
MOV CL,4
SHR AL,CL       ;AL=02H
OR AL,30H       ;AL=32H
MOV DL,AL       ;DL=32H
MOV AH,2
INT 21H         ; AH=2:显示 DL 对应字符 '2'
POP AX
AND AL,0FH      ;AL=05H
OR AL, 30H      ;AL=35H
MOV DL,AL       ;DL=35H
MOV AH,2        ; 请分析此句是否可不写?
INT 21H         ;显示' 5'

POPF
POP DX
POP CX
POP AX
RET
WRITE_AL ENDP
```

第十六节 循环指令（原版教材第 95 页）

➢ **基本的循环指令 LOOP**

循环程序一般结构：

```
        MOV CX，循环次数
标号：
        循环体
        LOOP 标号
```

解释：当 CPU 执行 LOOP 时，**先将 CX 减 1，然后判断 CX 是否为 0**，若 CX=0 则执行 LOOP 的下一条指令，否则跳转至指定标号处执行。

例如：

```
        MOV CX,5
        MOV AH,2
        MOV DL,' A'
L:      INT 21H
        LOOP L
```

分析：若 CX 的初值为 0，则会循环多少次？为此，产生了 JCXZ 指令。

用法：JCXZ 标号
作用：若 CX=0 时则跳至指定标号处执行。JCXZ 应放在循环体之前。

- **LOOPE 或 LOOPZ 指令（相等则循环）**

用法：同 LOOP
解释：LOOPE 和 LOOPZ 是两个完全相同的指令。其含义是，当 CX 非 0 **并且** ZF 为 1 时则循环（最多循环 CX 次，当遇到不等时则中止循环），主要用于查找某个缓冲区中的第一个不等于某个值的位置（如第一个非 0 值或第一个非空格等）。
例如：查找字符串 BUFFER 中的第一个非空格。

```
BUFFER DB “     BYE”
LEN EQU $-BUFFER ; 获取字符串字节长度
…
        MOV CX,LEN     ; 循环次数
        MOV BX,OFFSET BUFFER

L:      CMP BYTE PTR [BX],20H  ;space 的 ASCII 为 20H
        LOOPE L
        CMP CX,0
JNZ FOUND      ; 若已退出循环但 CX 不等于 0，说明已找到
        显示“未找到”
        …
FOUND:
        显示“已找到”
```

当然，我们只使用 LOOP 指令也完全可实现以上逻辑：

```
        MOV CX，LEN
        MOV BX,OFFSET BUFFER
L:      CMP BYTE PTR [BX]，20H
        JNZ FOUND
        INC BX
        LOOP L
        显示“未找到”
FOUND:
        显示“已找到”
```

- **LOOPNE 或 LOOPNZ 指令（不相等则循环）**

用法：同 LOOP
解释：与 LOOPE 相反。

第十七节 跳转指令

1. 无条件转移指令 JMP（原版教材第 114 页）

不同类型的 JMP 指令所能胜任的跳转距离不同，生成的机器码长度不同，执行的速度也不同。若 JMP 指令中未明确指明跳转类型，汇编器会根据情况，灵活决定生成何种类型的跳转指令（尽量短）。**跳转距离**是指 JMP 下一条指令与目标间的距离（跳转距离＝目标地址－jmp 下一条指令地址）。

格式：JMP [SHORT|NEAR|FAR|WORD|DOWRD] PTR 标号

- **短跳（段内直接跳）**：JMP SHORT 标号

解释：对应机器码为 2 字节：EBXX，其中 XX 为 1Byte 范围内跳转距离（用补码表示），跳转距离不能超过-128~+127 字节范围。
例如：
JMP SHORT L ；生成的机器码为 EB03H
MOV AX，0 ；该指令为 3 字节长
L：

- **近跳（段内直接跳）**：JMP NEAR PTR 标号 （最常用）

解释：对应机器码为 3 字节：E9XXXX，其中 XXXX 为 2Byte 范围内跳转距离（用补码表示），跳转范围显然在-32768~+32767 字节之间。
例如：L：JMP NEAR PTR L；生成的机器码为 E9FDFFH。（FFFD 为-3 的补码）

- **远跳（段间直接跳）**：JMP FAR PTR 标号 （很少用）

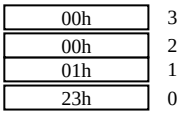
解释：对应机器码为 5 字节了：$EAX_1X_2X_3X_4Y_1Y_2Y_3Y_4$，其中 $X_3X_4X_1X_2$ 为目标偏移，$Y_3Y_4Y_1Y_2$ 为目标段地址，显然可跨段跳转。（注意小端的内存存放方式）

- **近跳（段内间接跳）**：JMP WORD PTR 字类型变量 （很少用）

解释：将指定变量处存放的一个字作为目标 IP 的值，CS 不变。
例如：
VAR DW 1234H
JMP WORD PTR VAR
将跳转至 CS：1234H 处执行

- **远跳（段间间接跳）**：JMP DWORD PTR 双字类型变量 （常用）

解释：将指定变量处存放的值作为双字看，其中前一个字（低地址）作为目标 IP 的值，后一个字（高地址）作为目标 CS。
例如：
BOOTENTRY DD 0FFFF0000H
JMP DWORD PTR BOOTENTRY
将跳转至 FFFF：0000H 处执行
又例如：
mov ax,0123h
mov ds:[0],ax
mov word ptr ds:[2],0
jmp dword ptr ds:[0]
将跳转至 0000：0123H 处执行



2. 条件转移指令 Jx （原版教材 120 页）
格式：
CMP 目，源 （有时也可不用 CMP 指令）
Jx 标号
解释：CMP 指令在内部模拟一次 SUB 运算，只改变标志寄存器的值，但不改变目的操作数的值，所以可作为后继 Jx 指令的判断条件。
注意：所有的**条件转移指令**均固定为 SHORT 跳(跳转距离小于 1Byte)。若跳转距离超过-128~+127 字节范围，必须用适当的 JMP 指令进行中间接力，否则会报错”Jmp out of rang”。
例如：若 AX 中的运算结果为 0，则跳至 L。

```
  CMP AX，0
  JZ L
  ……
L：
```

若范围超出+128 字节，可改为

```
   CMP AX，0
   JNZ  TMP
   JMP L
   TMP：
   ……
L：
```

（1）相等的比较（JZ）

- 相等的比较：使用 JZ 或 JE 指令
- 不相等的比较：使用 JNZ 或 JNE 指令

这些指令判断的依据是 **ZF** 标志位。

例如，

```
CMP AX，5
JZ  L
CMP AX，BX
JNZ  L
```

（2）无符号数高低的比较 （JA 和 JB）

- 高于等于的判断：使用 JAE 或 JNB 指令（高于等于或不低于）
- 低于的判断：使用 JB 或 JC 或 JNAE 指令（低于或不高于等于）

这些指令判断的依据是 **CF** 标志位。若目的操作数比源操作数低，则 CMP 相减时必然产生进位（CF=1），所以 JC 与 JB 是完全相同的指令。

- 低于等于的判断：使用 JBE 或 JNA 指令（低于等于或不高于）

测试条件：CFVZF＝1

- 高于的判断：使用 JA 或 JNBE 指令（高于或不低于等于）

测试条件：CFVZF＝0

例如：若 AX 高于 0，则跳至 L。

```
CMP AX，0
JA  L
```

（3）有符号数大小的比较 （JG 和 JL）

- 大于的判断：使用 JG 或 JNLE 指令（大于或不小于等于）
- 大于等于的判断：使用 JGE 或 JNL 指令
- 小于的判断：使用 JL 或 JNGE 指令
- 小于等于的判断：使用 JLE 或 JNG 指令

这些指令判断的依据比较复杂，此略。

例如：若 AX 为正数，则跳至 L。

```
CMP AX，0
JG  L
```

问题：下列指令 JB 和 JL 哪个条件能满足？

```
MOV AL，-1
CMP AL，0
JB     L1
JL     L2
```

（4）JC/JNC 指令

JC L 的作用是：IF **CF=1**，THEN JMP L

（5）JS/JNS 指令

JS L 的作用是：IF **SF=1**，THEN JMP L

可用于判断数的正负。

例如：若 AX 为负数，则跳至 L。

```
CMP AX，0
JS  L
```

（6）JP/JNP 指令：判断 PF 是否为 1

（7）JO/JNO 指令：判断 OF 是否为 1

常用的：

```
ja 大于时跳转
jae 大于等于
jb 小于
jbe 小于等于
je 相等
jna 不大于
jnae 不大于或者等于
jnb 不小于
jnbe 不小于或等于
jne 不等于
jg 大于(有符号)
jge 大于等于(有符号)
jl 小于(有符号)
jle 小于等于(有符号)
jng 不大于(有符号)
jnge 不大于等于(有符号)
jnl 不小于
jnle 不小于等于
jns 无符号
jnz 非零
js 如果带符号
jz 如果为零
```

- a: above
- e: equal
- b: below
- n: not
- g: greater
- l: lower
- s: signed
- z: zero

综合例子：

在屏幕上以小写形式显示 AL 的值。例如若 AL=2EH，则显示 2e。

```
WRITE_AL PROC
; 作用：以小写显示 AL 中的值
; 说明：本过程不会破坏所有寄存器的原值
        PUSH AX
        PUSH CX
        PUSH DX
        PUSHF

        PUSH AX
        MOV CL,4
        SHR AL,CL           ;取 AL 的高四位
        ADD AL,30H          ;得 2 的 ASCII 码 32H
        CMP AL,39H          ;判断数字是否在 0～9 以内
        JBE L1              ;若在数 0～9 内则跳转到 L1 处
        ADD AL,27H          ;若在数 A～F 内则修正，使其得到 a～f 的 ASCII 码
L1:     MOV DL,AL           ;DL 为 AL 第四位对应数的 ASCII 码
        MOV AH,2
        INT 21H             ;显示 AL 第四位对应 ASCII 码
        POP AX
        AND AL,0FH          ;处理高四位，与上同
```

```
        ADD AL,30H
        CMP AL,39H
        JBE L2
        ADD AL,27H
L2:     MOV DL,AL
        MOV AH,2
        INT 21H

        POPF
        POP DX
        POP CX
        POP AX
        RET
WRITE_AL ENDP
```

第十八节　子程序指令

- **call 指令**

  call [far|word|dword ptr] 标号|寄存器|内存单元

执行 call 指令时主要进行以下步骤：

1）　压栈

　　段内　push (IP)；

　　段间　push (CS)

　　push (IP)

2）　转移

　　类似跳转指令，更改 IP（段内跳转）或 IP 和 CS（段间跳转）的值，实现转移；

说明：用 call 指令没有短跳转，实现段内转移时的位移量为 2B（近跳转）。

- **ret 指令**　　pop IP
- **ret n 指令**　　pop IP

  pop CS

  add SP,n

说明：利用栈进行参数传递时，调用程序要向栈压入参数，子程序在返回时可用该指令修正栈指针调用前位置。

　　当标号为过程名时，跳转类型由过程属性决定，当过程为 near 时，为段内跳转；

　　过程为 far 时，为段间跳转。

如：

```
main proc far
  ……
  call sub1    ;call 为近转移
  ……
  ret          ;ret 为远转移
main endp
sub1 proc near
  ……
  ret          ;ret 为近转移
sub1 endp
```

- **强制远转移返回指令**

  retf 指令 pop IP

  pop CS

- **过程的参数传递**

  多使用寄存器或栈

下面是一个用栈传递参数的例子：

例子：

```
data segment
     a dw 1122h
     b dw 3344h
     sum dw 0
data ends

code segment
     assume   cs:code, ds:data
start:
     mov ax, data
     mov ds, ax

     push b                  ;参数入栈
     push a

     call calcsum
     mov sum, ax

     add sp, 4               ;由调用者清理栈 或者 过程中使用 ret n

     mov ah, 4ch
     int 21h

calcsum PROC
       PUSH BP
       MOV BP,SP

       mov ax, ss:[bp+4]
       add ax, ss:[bp+6]

       MOV SP, BP
       POP BP
       RET
calcsum  ENDP
code ends
       end start
```

第十九节　中断指令

- **int 中断指令**

格式：INT 中断类型号(不写默认为 3h — 断点中断)

执行：
```
push FLAGS/EFLAGS
IF← 0
TF← 0
Push CS
Push IP
(IP)← 中断类型号×4
(CS)← 中断类型号×4+2
```

- **into 溢出中断**　中断号为 4 的溢出中断，响应过程同上

- **iret/iretd 中断返回指令**

格式：iret/iretd
执行： pop IP/EIP
pop CS
pop FLAGS/EFLAGS

第二十节 串处理指令 （原版教材 203 页）

8086 CPU 提供了一些高效的串指令，主要包括：

指令	作用	隐含使用的寄存器	字节操作符	字操作符
MOVS	移动字符串	DS:SI->ES:DI	MOVSB	MOVSW
LODS	装载字符串	DS:SI->AL（或 AX）	LODSB	LODSW
STOS	存储字符串	AL（或 AX）->ES:DI	STOSB	STOSW
CMPS	比较字符串	DS:SI==ES：DI	CMPSB	CMPSW
SCAS	搜索字符串	AL（或 AX）==ES：DI	SCASB	SCASW

说明：

- 所有的串处理指令都用 **DF 标志**位决定字符串移动（或存储）的方向，在使用串指令前应使用 CLD 或 STD 指令设置 DF 的值，默认 DF=0（地址增加方向）。
  CLD 指令：使 DF=0，此时，将按地址增加的方向移动（或存储）字符串。
  STD 指令：使 DF=1，此时，将按地址减少的方向移动（或存储）字符串。

- 所有涉及 SI 的串指令在执行后，会自动将 SI 加 1（对字符串操作）或 2（对字操作）；类似地，所有涉及 DI 的串指令在执行后，会自动将 DI 加 1（对字符串操作）或 2（对字操作）。因此需注意，当执行完这些串指令后，SI 或 DI 并不指向最后处理的一个字节（或字），而是其**下一个**字节（或字）。

- 串指令前可使用以下前缀：
  REP：重复执行该指令，共 CX 次。
  REPE（或 REPZ）：重复执行该指令，最多 CX 次，当 ZF=0（即不等时）时则中止执行。用于 CMPS 和 SCAS 指令（搜索某个不等于某数的值）。
  REPNE（或 REPNZ）：与 REPE 相反。

例如：

```
.model small
.data
       buffe1 db “Hello World”
       len equ $-buffer1
       buffer2 db len dup (?)
.code
start:
       mov ax,@data
       mov ds,ax
       mov es,ax
       lea si,buffer1
       lea di,buffer2
       mov cx,len
       cld
       rep movsb
       mov ax,4c00h
       int 21h
       end start
```

以下是关于命令行参数的例子：

命令行参数位于程序段前缀 PSP 偏移量 80H 处开始的地方，其中，PSP:80H 处存放的是命令行字符个数（不含回车 0DH），PSP:81H 开始存放的是具体的命令行字符（以 0DH）结束。

例如，假设程序 READFILE 带参数 ABC 运行，即 C>READFILE ABC

则 PSP:80H 处的内容应是 04 20 41 42 43 0D，在纯 DOS 环境中，以及 Win2K 的 DEBUG 环境中，空格均不会被去掉，并且计入命令行字符个数中（就像我们上课讲的那样）。但在 Win2K 内部所提供的命令行提示符下，命令行字符的所有前导空格均会被自动删除，即 PSP:80H 处的内容是 03 41 42 43 0D。

有些同学在 Win2K 内的命令行中，若以 C>READFILE ABC 运行不能打开文件 ABC，但在 DEBUG 中却能打开文件 ABC。其原因如上。

其实，设计好的程序，不应直接到 PSP:82H 处取字符，而应该能自动跳过前导空格，这样设计的程序就不会出现以上那种在不同环境中运行结果不一样的情况。如下，是很好的代码：

```
.CODE
        XOR CX,CX       ;DS=ES=Segment of PSP
        MOV CL,ES:[80h]
        MOV DI,81H
        MOV AL,20H
        CLD
        REP SCASB       ;Skip the initial space
        DEC DI          ;let DI point to the first non-space char
```

其中 REP SCASB 是在 ES:DI 中自动搜索等于 AL 的字符，最多搜索 CX 次，当找到时便停止。由于 SCASB 每搜索一个字符都会自动将 DI 加 1（字节操作），所以最后应将 DI 倒回去 1 个字符。

不用 SCASB 的代码如下：

```
.CODE
        MOV DI,80H
SPACE:
        INC DI
        CMP BYTE PTR ES:[DI],20H
        JZ SPACE
```

以上执行完后 DI 均指向首个非空字符。

另外注意，最好按 ALT+ENTER 键将命令行窗口切换成全屏幕方式，否则有些信息可能不能正常显示。

第二十一节　换码指令 （教材 55 页）

格式：XLAT　　；al←ds:[bx+al]

将 BX 指定的缓冲区中、AL 指定的位移处的一个字节数据取出赋给 AL。

例子：

```
; 查表法实现一位 16 进制数转换为 ASCII 码的显示
        .model small
        .stack
        .data
ASCII   db 30h,31h,32h,33h,34h,35h,36h,37h,38h,39h      ;0～9 的 ASCII 码
        db 41h,42h,43h,44h,45h,46h                      ;A～F 的 ASCII 码
hex     db 0bh                      ;任意设定了一个待转换的一位 16 进制数
        .code
        .startup
        mov bx,offset ASCII         ;BX 指向 ASCII 码表
        mov al,hex                  ;AL 取得一位 16 进制数，正是 ASCII 码表中位移
        and al,0fh                  ;只有低 4 位是有效的，高 4 位清 0
```

```
xlat                        ;换码：AL←DS:[BX＋AL]
mov dl,al                   ;入口参数：DL←AL
mov ah,2                    ;02 号 DOS 功能调用
int 21h                     ;显示一个 ASCII 码字符

mov ax,4c00h
int 21h
end
```

第三部分 专题知识

第二十二节 COM 程序（原版教材 108 页）

常用的两种可执行程序。

一、EXE 程序

优点：程序大小没有限制，是可执行程序的主流形式。

缺点：由于 EXE 文件中包括重定位项（程序中出现的所有段值需在运行时刻由操作系统根据当时内存使用情况临时修正），所以 EXE 程序的启动速度较慢。另外，每个 EXE 文件的最前部都包含一个 512 字节的文件头（其中包含需修正的重定位项的位置，以及 IP、SP 等寄存器的初值），所以 EXE 文件不可能小于 512 字节。

EXE 文件组成：

文件头(512B)	可重定位程序的映像

EXE 程序初始时各寄存器的初值：

CS=代码段的段地址，DS=ES=程序段前缀（PSP）的段地址，SS=栈段的段地址，SP=栈段长度，IP=程序入口地址（由 END XX 决定）

EXE文件的装入执行

地址	内容	
XXXX:0000	PSP	← DS,ES
XXXX:0100	数据	
	程序代码	← CS:IP
	堆栈	← SS:SP

二、COM 程序

优点：程序长度可很短（大于 0 字节）。另外，COM 程序中不包含任何需重定位的项，从磁盘调入内存后可立即执行，启动速度很快。

缺点：不具代表性，属可执行程序的特殊形式。COM 文件的最大长度不能超过一个段。

COM 程序初始时各寄存器的初值：

CS=DS=SS=ES=PSP，IP=100H，SP=FFFEH

COM文件的装入执行

地址	内容	
XXXX:0000	PSP	← CS,DS,ES,SS
XXXX:0100	程序代码	← IP
	数据	
	堆栈	← SP

总结：

1. 对于.com 格式的可执行文件，加载后未执行前的寄存器状态：

CS = DS = ES = SS = PSP 的段地址

IP = 0100H

SP = FFFEH

AX 与 FCB 相关

其他未定义

2. 对于.exe 格式的可执行文件，加载后未执行前的寄存器状态：

DS = ES = PSP 的段地址

CS:IP = .exe 头中指出的执行开始位置（经过重定位）

SS:SP = .exe 头中指出的栈段和栈指针（经过重定位）

AX 与 FCB 相关

其他未定义

建立 COM 程序的特殊要求：

- 内存模型（.MODEL）只能为 tiny 或 small 两者之一。

- 程序中只能定义一个段，即代码段。程序中不能定义独立的数据段和栈段。
- 程序必须从 CS:100H 处开始执行（通过在第一条可执行指令前写 ORG 100H 伪指令来保证），并且在 CS:100H 之前不能有任何数据。
- 程序最大长度不能超过 65536-256-2 字节。
- 程序中不能出现需重定位的段（即程序中不能出现段名）。
- 程序除可用 INT 21H 的 AH=4CH 号功能结束外，还可以 INT 20H 结束。
- 生成 COM 时，若用 TLINK 连接，需带/t 开关。也可先生成 EXE，然后用 EXE2BIN.EXE 程序，将 EXE 转换成 COM 程序。例如：

```
TASM TEST
TLINK TEST /t
或
TLINK TEST
EXE2BIN  TEST.EXE  TEST.COM
```

显示字符串的 COM 程序例子：

1．经典风格

```
CODE SEGMENT
       ASSUME CS:CODE, DS:CODE
ORG 100H
START:
JMP L
       MSG  DB ”  HELLO$”
L:
       MOV AX,CS              ；这两句可不写
       MOV DS,AX
       LEA DX，MSG
       MOV AH，9
       INT 21H
       INT 20H
CODE ENDS
       END START
```

2．简化风格

```
.MODEL TINY
.CODE
ORG 100H
START:
JMP L
       MSG  DB ”  HELLO$”
L:
       MOV AX,CS              ；这两句可不写
       MOV DS,AX
       LEA DX，MSG
       MOV AH，9
       INT 21H
       MOV AH，4CH
       INT 21H
END START
```

三、BIN 程序

扩展名为BIN的程序主要用于DOS下的设备驱动程序，其书写要求，除程序入口地址在CS：0或CS：100H均可外，其余与COM相同。

第二十三节 文件读写（原版教材304页）

文件读写有两种方式：**句柄方式(handle)**和**文件控制块方式（FCB）**。

- **句柄方式**使用的是INT 21H的AH=3CH至42H号功能，是推荐使用的方式。
- **文件控制块方式**使用的是INT 21H的AH=0FH至28H号功能，是推荐淘汰的方式。

以下只介绍以句柄方式读写文件的INT 21H调用：

- AH=3CH号功能：建立一个新文件，对应标准C函数 **int creat(const char *filename, int mode);**
- AH=3DH号功能：打开一个已有文件，对应标准C函数 **FILE *fopen(const char *filename, const char *mode);**
- AH=3EH号功能：关闭一个文件，对应标准C函数 **int fclose(FILE *stream);**
- AH=3FH号功能：读文件或设备，对应标准C函数 **size_t fread(void *buffer, size_t size, size_t count, FILE *stream);**
- AH=40H号功能：写文件或设备，对应标准C函数 **size_t fwrite(void *buffer, size_t size, size_t count, FILE *stream);**
- AH=42H号功能：移动文件指针，对应标准C函数 **int fseek(FILE *stream, long offset, int origin);**

读写文件的一般过程如下：

（1）以ASCII字符串给出路径和文件名（文件名需以0H结束）；
（2）用3C号（新建）或3DH（已有文件）功能打开文件。这一步骤输入的是文件名，返回的是文件句柄。以后的操作将使用该文件句柄。
（3）（若需要）用42H号功能将文件指针设置到合适位置；
（4）用3FH号（读）或40H号（写）读写文件内容；
（5）用3EH号功能关闭文件。

执行以上任何一个功能时，**若出错**（如文件找不到，文件指针移动超界、读写文件出错等），都会**自动设置CF=1**。所以执行每个功能后，都应使用JC指令判断本步操作是否出错。

涉及的一些概念：

- **文件句柄(handle)：**

一个16位数(Windows95后为32位无符号整数)，代表一个已打开的文件，相当于标准C函数的FILE *stream。规定前5个句柄为系统保留，其中0代表stdin(标准输入设备，即键盘)，1代表stdout(标准输出设备，一般为屏幕)，2代表stderr(标准出错输出设备，即屏幕)，3代表stdaux(标准辅助设备，即串行口)，4代表stdprn（标准打印设备，即并行口）。

- **文件存取模式(acess mode)：**0=read only，1=write only，2=read/write
- **文件指针(file pointer)：**

每个已打开的文件，其内部都有一个文件指针，代表当前的读写位置，初始时位置为0。移动文件指针时，所给出的移动距离，有相对于文件首（SEEK_SET）、相对于当前位置(SEEK_CUR)、相对于文件尾（SEEK_END）之分。

例子：创建文件

```
.model small
.data
filename DB 'f:\wdq.asm',00
handle   DW ?
inf    DB 'Fault!',0dh,0ah,'$'
.code
START:        MOV AX,@DATA
```

```
                MOV DS,AX
                LEA DX,filename
                MOV CX,0
                MOV AH,3CH
                INT 21H
                JC ERROR
                MOV handle,AX

                MOV AX,4C00H
                INT 21h
ERROR:          LEA DX,inf
                MOV AH,9
                INT 21H

                MOV AX,4C00H
                INT 21h
    END  START。
```

第二十四节　端口(port)（原版教材 394 页）

一、端口的概念

端口是一种设备，用于连接 CPU 与其它部件。计算机中，除了 CPU 芯片外，还有许多其它芯片，这些外围芯片中也包含很多寄存器，外围芯片中的寄存器便称为端口。CPU 通过读写这些端口来控制各种设备的工作。为了能使 CPU 访问这些端口，必须对这些端口进行编号，称为端口号或输入输出端口地址（称为 I/O Base 或 I/O Port）。

在 PC 只用了 10 位地址线(A0-A9)对端口地址进行译码，故端口的寻址范围为 0H-3FFH，共有 1024 个 I/O 地址。这 1024 个地址中前半段(A9=0，范围为 0H-FFH)是属于主机板 I/O 译码，后半段(A9=1，范围为 100H-3FFH)则是用来扩展插槽上的 I/O 译码用。

在 PC 机中，I/O 地址与内存地址都是从 0 开始分别编址，但**访问端口只能通过 IN/OUT 指令**进行，所以通过指令就可区分使用的是 I/O 地址，还是内存地址。

以下是一些常用的端口号：

21H	中断屏蔽寄存器（IMR）
60H	键盘扫描码读入端口
61H	扬声器控制(Bit 0 and 1)
70-71H	CMOS RAM 端口
3F8-3FFH	COM1 串行口
2F8-2FFH	COM2 串行口
3E8-2EFH	COM3 串行口
2E8-2EFH	COM4 串行口
378-37FH	LPT1 并行口
3BC-3BFH	LPT2 并行口
278-27FH	LPT3 并行口
3C0-3CF	VGA 显示卡

二、有关端口读写的指令

1．读指令：IN 寄存器，端口号　　；从端口读数据

2．写指令：OUT 端口号，寄存器　　；向端口写数据

规定 1：寄存器只能为 AL（读写 1 个字节）或 AX（读写 1 个字）。

规定 2：若端口号能用 1 字节表示（00-FFH），可在指令中直接给出，但若为 1 个字（100H-3FFH），则必须事先将端口号放到 DX 寄存器中。

例如，从 3F8H 号端口读入 1 个字节：

```
MOV DX，3F8H
IN AL,DX
```

三、地址口和数据口

有些设备上的端口较多，但 PC 机只能为这些设备预留有限数量的端口。例如系统只为 VGA 显示卡分配了 3C0-3CFH 共 10 个口地址，但实际的 VGA 显示卡内部有 50 多个端口，随着显示卡功能的增强，显示卡内部随时可能增加更多的端口。如何通过有限个端口地址来访问所有这些端口，便成为一个问题。

解决方案是，在系统中，为每个设备分配一个地址口和数据口，然后将同一个设备内部的所有寄存器进行编号（从 0 开始，称为索引），访问时，只要先将要访问的寄存器的内部编号（即数据的地址）放到该设备对应的地址口中，然后通过该设备的数据口所读写的数据便是相应寄存器中的内容。通过这种方法，实际上一个设备只要在系统中占用地址口和数据口这两个口地址，就可通过这两个端口访问该设备内部的所有端口。

地址口：也叫索引口，存放要访问的寄存器的内部编号。

数据口：通过该端口读写索引口中指定的寄存器。

状态口：反映设备的工作状态，从访问方法看，状态口也可作为数据口对待。

各设备在系统中所占用的地址口和数据口，请查 PC 机硬件资料。至于各设备内部的寄存器编号，请查该设备的硬件编程资料。

以下是 CMOS RAM 读写的例子：

CMOS RAM 用于保存计算机的硬件配置和系统设置等重要数据，目前 PC 机中的 CMOS RAM 一般为 64 字节或 128 字节。CMOS RAM 实际上是一种端口，要读写 CMOS RAM 中的内容，必须用 I/O 指令。在 PC 机中，CMOS RAM 只占用 70H（地址口）和 71H（数据口）两个口地址，但 CMOS RAM 中有 64 个（或 128 个）寄存器，索引从 00-3FH（或 7FH）。

例如，读 CMOS RAM 中的第 19H 字节到 AL 中：

```
MOV AL,19H
OUT 70H,AL
IN AL,71H
```

将 1FH 写入 CMOS RAM 中的第 19H 字节中（请不要在 PC 机上进行此实验）：

```
MOV AL,19H
OUT 70H,AL
MOV AL,1FH
OUT 71H，AL
```

四、DEBUG 有关读写端口的命令

（1）读端口的命令:

用法：I port

例如：-I 60

显示 1C

（2）写端口的命令:

用法：O port value

例如：-O 21 1

五、延时程序

286 及后续处理器中，61H 端口的 D4 位每 15.085us 变换一次状态可用于设置程序延时，如下面是产生 0.5s 的延时程序：

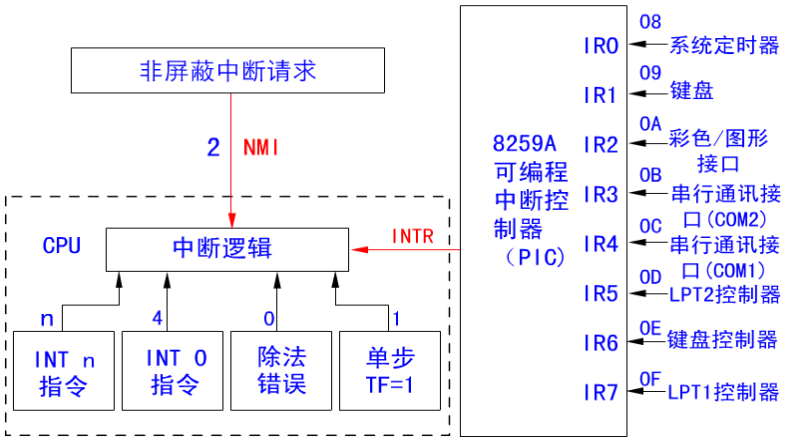
```
DELAY  PROC
       PUSH AX
       PUSH CX
       MOV CX,33144          ;delay=33144x15.085us=0.5sec
Wait:  IN AL,61H
```

```
        AND AL,00010000B      ;get D4 of the 61H port
        CMP AL,AH
        JE Wait
        MOV AH,AL
        LOOP Wait
        POP CX
        POP AX
        RET
DELAY   ENDP
```

第二十五节　硬中断

一、中断的类型

计算机中的中断根据信号产生的来源可分为：硬件中断和软件中断。硬件中断多由外围设备和计算机系统控制器触发，软件中断一般由软件命令产生(程序中写的 INT 指令)。在硬件中断中又有“可屏蔽中断”（INTR）和“不可屏蔽中断”（NMI）之分。顾名思义，可屏蔽中断可以由计算机根据系统的需要来决定是否进行接收处理或是延后处理（即屏蔽）；而不可屏蔽中断便是直接激活相应的中断处理程序，它不能也不会被延误（如内存出错、掉电等），我们常说的 IRQ 中断就是可屏蔽的硬件中断。



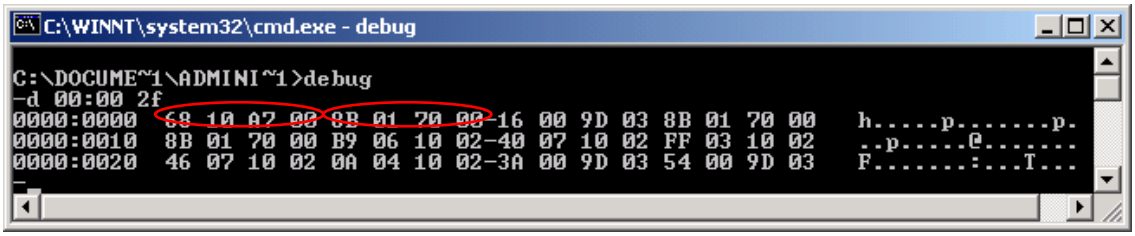
二、中断的执行过程

当有中断发生时，CPU 在内部执行如下动作：

- 将当前的标志寄存器压栈（PUASHF）；
- 清除标志寄存器中的陷阱标志位 TF（禁止 CPU 工作于单步模式）和中断允许位 IF（禁止 CPU 响应其它 IRQ 中断）；
- 将主程序中当前的 CS 和 IP 压栈;
- 到中断向量表（0：中断号*4）中取该相应中断服务程序的入口地址（前一个字作为目标 IP，后一个字作为目标 CS），然后跳转到中断服务程序中执行。

对应 IRQ 中断，中断服务程序一般是到该 IRQ 设备对应的端口中读写数据，进行合适的处理，以达到控制硬件工作的目的。中断服务程序内部必须以 IRET 结束（IRET=RETF+POPF），执行 IRET 后，CPU 便能返回到主程序中继续后续指令。

例如查看中断向量表：



INT 0 除零中断 ISR（Interrupt Service Routine）的逻辑地址：00A7：1068

三、IRQ 中断（外设中断）

IRQ 表示 Interrupt Request，即“中断请求”的意思。当外部设备需要 CPU 为其服务时，外设便通过相应的 IRQ 中断请求线向 CPU 提出中断请求。此时，若 CPU 响应该请求（可通过 cli/sti 指令设置），则 CPU 便执行相应的中断服务程序，读写相应的外设端口，为该外设服务。

在 PC 机中，所有 IRQ 请求线均接在中断控制器 Intel 8259A（PIC-可编程中断控制器，它的升级版 APIC 82093AA）上，而 8259 的 INT 引脚与 8086CPU 的中断请求引脚 INTR 相连。所以，所有的 IRQ 中断均需通过 8259 来控制。在 PC 机中，共有两片 8259 芯片，主片 8259 通过 IRQ2 引脚与从片 8259 的 IRQ9 引脚相连。每片 8259 可接 8 个外设 IRQ 线，由于主片和从片级连共占去 1 个引脚，所以实际只能接 15 个外设中断请求。

图1 IBM PC/AT机硬中断结构

IRQ编号	设备名称	用途
IRQ0	Time	电脑系统计时器
IRQ1	KeyBoard	键盘
IRQ2	Redirect IRQ9	与IRQ9相接，MPU-401 MDI使用该IRQ
IRQ3	COM2	串口设备
IRQ4	COM1	串口设备
IRQ5	LPT2	建议声卡使用该IRQ
IRQ6	FDD	软驱传输控制用
IRQ7	LPT1	打印机传输控制用
IRQ8	CMOS Alert	即时时钟
IRQ9	Redirect IRQ2	与IRQ2相接；可设定给其它硬件使用
IRQ10	Reversed	建议保留给网卡使用该IRQ
IRQ11	Reversed	建议保留给AGP显卡使用
IRQ12	PS/2Mouse	接PS/2鼠标，若无也可设定给其他硬件使用
IRQ13	FPU	协处理器用，例如FPU（浮点运算器）
IRQ14	Primary IDE	主硬盘传输控制用
IRQ15	Secondary Ide	从硬盘传输控制用

中断优先级：

8259 控制的外部中断优先级可通过编程设置(中断控制寄存器端口 20H)

中断编号：在 PC 机中，由 8259A 管理的 16 级中断均有规定的中断向量存储地址，主片中 IRQ0-IRQ7 分别对应 INT 08H-0FH，从片中 IRQ8-IRQ15 分别对应 INT 70H-77H。这些中断均属硬中断。

中断控制寄存器（ICR）和中断屏蔽寄存器（IMR）：

主片的中断控制寄存器 ICR 和中断屏蔽寄存器 IMR 的端口地址分别为 20H 和 21H，从片的相应寄存器口地址分别为 A0H 和 A1H。通过 ICR 可对中断控制器进行一些控制（详见“附件 11、Intel 8259 简介”）。通过 IMR 可控制中断控制器是禁止或开放来自某个 IRQ 的中断请求（0=Enable，1=Disable）。

四、IRQ 号的配置原则

IRQ 号、端口地址、DMA 通道号和硬件占用的内存地址是 4 种重要的硬件资源。在 PC 机中，一般任何两个外设不能在同一时刻占用同一个 IRQ 号，或相同的 I/O 地址，或 DMA 通道号，也不可共享内存地址，否则相应设备便不能正常工作。此时，可找出未用的那些 IRQ 号分配给冲突的设备。当前的 PC 机一般都具有 Plug and Play，能自动为外设分配合适的 IRQ 号。若仍出现冲突，只能手动设置。在 Windows 中，在“控制面板/系统/设备管理器/硬件”中，在“查看”菜单中选“依连接查看硬件资源”可显示该计算机的以上 4 种硬件资源占用情况。

关于 IRQ 中断能否共享问题，若中断控制设为边沿触发(edge trigger)，则不能共享；若为电平触发（level trigger），则可共享。在目前的 PC 机中，使用 ISA 总线的外设不能共享 IRQ，但使用 PCI 总线的外设可共享 IRQ。由于 PC 机中，ISA 和 PCI 设备混用，所以一般应按不能共享的原则设置。

五、编写 IRQ 中断服务程序的一些原则

进入中断服务程序前，CPU 已自动将标志寄存器中的中断允许位 IF 清 0，以防本中断程序执行时被高优先级的 IRQ 中断所打断。如果希望在执行低优先级的中断服务程序时，CPU 仍能响应高优先级的 IRQ 中断，则应在中断服务程序开始时，通过 STI 指令使 CPU 能响应其它 IRQ 中断。（STI 开中断，CLI 关中断）。注：CLI/STI 指令是告诉 CPU 是否响应来自中断控制器的 IRQ 中断请求（CLI/STI 只有写在 IRQ 中断程序中才有用，而且只影响 IRQ 中断，对软中断无效），而中断屏蔽寄存器则是控制中断控制器 8259 本身。一个 IRQ 中断要被 CPU 响应，首先要过 8259 的 IMR 一关，然后再过 CLI/STI 一关。

中断程序最后必须用 IRET 结束。而且在 IRET 结束前，应向 ICR（端口号为 20H 或 A0H）发送中断结束控制字 EOI（其代码为 20H），以便告诉 8259，本次中断响应已结束，8259 便可响应下一个中断。

中断程序内部不允许调用那些不可重入的中断。操作系统中断（中断号在 INT 20H 及以上）均属不可重入的中断。如果想在中断程序内部进行某些输入输出处理，只能使用 BIOS 中断（INT 20H 以下）或直接对硬件端口编程。例如，显示可使用 INT 10H 的 AH=0EH 号功能，键盘输入可使用 INT 16H 的 AH=00H 号功能，读写磁盘只能使用 INT 13H 的 AH=02/03H 号功能等。该原则适于编写所有的中断服务程序，不仅仅是对编写 IRQ 中断服务程序的要求。

中断程序的执行时间应尽量短。

六、时钟中断（INT 8 和 INT 1CH）

时钟中断 IRQ0 由定时器 Intel 8253 芯片所自动触发。在 PC 机中，BIOS 在启动时已将定时器（Intel 8253）初始化为每秒钟自动产生 18.2 次时钟中断（即 INT 8），即每 55ms 产生一次 INT 8。INT 8 中断程序内部的主要工作流程如下：

- 修改位于 ROM 数据区 40:6CH 处的一个双字（当前时间计数值）；
- 若软驱空闲时间较长，自动关闭软驱马达；

```
INT 1CH
IRET
```

INT 1CH 主要是保留给用户编写自己的定时服务程序使用。所以我们可编写 INT 1CH 中断程序来做我们自己的定时事情。

实验 1：查看当前计时值
-D 40:6C （当前计时值）（地址 40：6CH 处双字记录时钟值，详见附录 2）
-D 40:6C（此值在不断增加）
-o 21 1 （禁止时钟中断）
-D 40:6C（当前值）
-D 40:6C（计时不动了）
-o 21 0 （允许时钟中断）
-D 40:6C（当前值）
-D 40:6C（值又开始增加）

实验 2：编写最简单的 INT 1CH 程序，使每秒显示 18.2 个’A’ (可在 Windows 中实验)。
-A 200
XXYY:0200 MOV AX,E41 ;AH=E
XXYY:0203 INT 10
XXYY:0205 IRET
XXYY:0206
-E 0:70 00 02 YY XX
屏幕上立即开始每秒显示 18.2 个’A’。其中 0:70H 是 INT 1CH 中断服务程序入口地址在中断向量表中的保存位置（1CH*4=70H）。
-021 1 （禁止时钟中断，会立即停止显示）
-021 0 （开放时钟中断，会立即开始显示）

实验 3：对定时器 Intel 8253 编程，使每秒显示 72.8 个‘A’ (可在 Windows 下实验)
-A200
XXYY:0200 MOV AX,E41
XXYY:0203 INT 10
XXYY:0205 IRET
XXYY:0206
-A100
XXYY:0100 MOV AL,36 设置计数器 0
XXYY:0102 OUT 43,AL
XXYY:0104 MOV AL,00 计数值低字节
XXYY:0106 OUT 40,AL
XXYY:0108 MOV AL,40 计数值高字节
XXYY:010A OUT 40,AL
XXYY:010C
-P=100 6
-E0:70 00 02 YY XX
屏幕上立即开始每秒显示 72.8 个’A’。

实验 4：用定时器产生 0-65535 之间的随机数。
-A100
XXYY:0100 MOV AL,36 设置计数器 0
XXYY:0102 OUT 43,AL
XXYY:0104 IN AL,40H ；读入低字节
XXYY:0106 MOV AH,AL
XXYY:0108 IN AL,40H ；读入高字节
XXYY:0109 XCHG AL,AH; ；AX=当前计数值
-P=100 6
-P=100 6

从 40H 读入的为计数器 0 的当前计数值（初置为 FFFFH），以每秒 1.19M 次的速度递减，至 0 后重新置为 FFFFH。

实验 5：音乐程序（见附件 12：SETINT1C.ASM，可在 Windows 中运行）。

七、键盘中断（INT 9）

每当有击键动作，键盘控制器 Intel 8042 便向中断控制器提出 IRQ1 中断申请，进而触发 CPU 执行 INT 9 中断程序，处理按键。INT 9 主要作用是，读 60H 号端口，取得击键盘对应的键盘扫描码，然后查内部表得到键盘扫描码所对应的 ASCII 码，然后将键盘扫描码和 ASCII 码一同送入键盘缓冲区中（位于 ROM 数据区 40:1E-3EH 处，是一个 32 字节的循环队列），并修改队列的首尾指针。以后其它程序便可通过调用 INT 16H 读写键盘缓冲区中的键盘扫描码和 ASCII 码（INT 21 内部需调用 INT 16H 读键盘）。

实验（在 Windows 中实验均有效）：

```
-I60
```

显示 1C，为回车键对应的扫描码

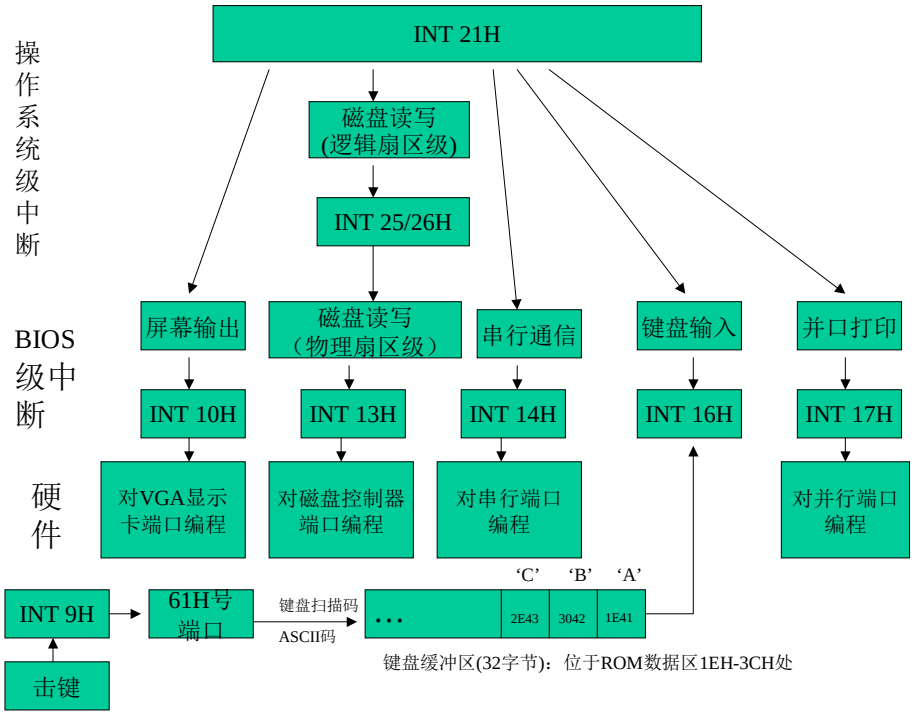
```
-D40:1A     显示键盘缓冲区中的内容，其中 1A 和 1CH 处的字分别为头/尾指针
0040:0010                                                   36 00 36 00 0D 1C
0040:0020  69 17 36 07 30 0B 0D 1C-44 20 34 05 30 0B 3A 27
0040:0030  31 02 41 1E 0D 1C 0D 1C-69 17 36 07 30 0B

-021 1
```

键盘按不动了

第二十六节　输入输出操作

下图列出了主要的 BIOS 中断，及其与操作系统中断的调用关系。



一 键盘 I/O

键盘缓冲区输入过程：

1、 获取扫描码－－INT 9H

键盘上“按下”或“放开”某键，产生 INT 9H 中断（中断允许时），该中断处理程序从端口 60H（8255（并行输入输出接口 PPI）的输入端口）获取一个字节的键盘扫描码，“按下”时最高位 bit7 为‘0’表示为该键“通码”；“放开”时 bit7 为‘1’表示“断码”）；

2、 写键盘缓冲区－－INT 9H

BIOS 中断处理程序将取得的扫描码转换成字符码（如果字符码存在），接着将该键对应的字符码和扫描码存储到 BIOS 数据区（见附录 2）的键盘缓冲区中；

地址	长度	意义
0000：041AH	字	指向键盘缓冲区首址（随机的，但键盘缓冲区地址范围内）
0000：041CH	字	指向键盘缓冲区尾址，当该值等于前一字的值时，说明缓冲区空。
0000：041EH	16 字	循环键盘缓冲区，它保存键盘键入的字符，直到程序可以接收这些字符为止，前两个字指向此缓冲区的当前是首和尾。

0040:17 单元中存储键盘状态字节，用来记录控制键和切换键的状态。

读键盘：

1、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 ZF=1，缓冲区空
2	取键盘状态字节	AL=键盘状态字节

键盘状态字用来反映组合功能键（交换键）状态的，详情见书上 P319.

2、DOS 键盘中断（INT 21H）

AH	功能	调用参数	返回参数
1	输入一个字符并显示 （检测 Ctrl-C 或 Ctrl-break，并调用 INT 23H 结束程序）		AL=字符
6	读键盘字符，不回显	DL=0FFH	有字符，AL=字符，ZF=0 无字符，AL=0 ，ZF=1
7	输入一个字符不显示 （不支持检测 Ctrl-C 或 Ctrl-break）		AL=字符
8	输入一个字符不显示 （检测 Ctrl-C 或 Ctrl-break，并调用 INT 23H 结束程序）		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=00H 无键入
C	清除键盘缓冲区 调用一种键盘功能	AL=键盘功能号 （1、6、7、8、A）	

二 显示器 I/O

对于所有的显示适配器，文本方式下字符显示的原理都类似，每个字符的 ASCII 码和属性码各占一个字节，所以每个字符占用显存一个字单元。所不同的是其显存的起始地址不同：

- MDA(Monochrome display adaptor)单色显示适配器
  显存起始地址：B000:0000
- CGA、EGA 和 VGA
  显存起始地址：B800:0000

以下只介绍有关输入输出的中断。

一、有关输出的中断

- INT 21H 有关输出的主要功能：

AH=02H：输出单个字符

AH=09H：输出一串字符（以’$’结束）

INT 10H 的主要功能（见原版教材 328 页）：

AH=00H：设置显示模式
AH=0EH：以 TTY 方式在当前光标处显示一个 ASCII 字符（电传打字方式，即显示一个字符后，光标自动前移）。
例如，在光标当前位置处显示一个‘A’：
MOV AH，0EH ；功能号
MOV AL，‘A’ ；欲显示字符的 ASCII 码
MOV BH，0 ；显示页的页号，一般应设为 0

- INT 10H

INT 10H 的其它功能请见教材，支持 VESA 扩展功能的显示卡还在 BIOS 中对 INT 10H 进行了很多扩展，这些扩展功能在教材中未列出。若对图形有兴趣，可参考 VGA 显示卡编程手册。

二、有关输入的中断

- INT 21H 有关输入的主要功能（P320）：

AH=01H：输入单个字符
AH=0AH：输入一串字符（以回车结束）

- INT 16H 的主要功能（见原版教材 P318 页）。

功能：AH=00H
作用：输入单个字符
返回：AH=扫描码，AL=ASCII 码

功能：AH=01H
作用：测试键盘有否输入（因其并不将字符读出，所以可用于提前测试按键为何字符。若需读出，需使用 AH=0 功能）
返回：
若 ZF=1，表示键盘无输入
若 ZF=0，表示键盘有输入，此时，AH=扫描码，AL=ASCII 码

功能：AH=02H
作用：返回键盘状态字节（即 40：17H 处的一个字节）。
返回：AL=返回键盘状态字节。

实际上，INT 16H 并不直接对键盘控制器硬件(Intel 8042)编程，而是到键盘缓冲区中读取字符。键盘缓冲区是一个循环队列，位于 ROM 数据区 40：1A-3CH 处，共 34 字节。其中，40：1EH-3CH，共 30 字节存放的是每个击键所对应的扫描码和 ASCII 码；而 40：1A-1BH 处存放的是该对列的头指针，INT 16H 到头指针指示的位置取字符；40：1C-1DH 处存放的是该对列的尾指针，键盘输入字符的扫描码和 ASCII 码由硬中断 INT 9 负责存储到尾指针所指示的位置。见原版教材 197 页。

扫描码的概念：大致表示一个键在键盘上的物理位置，是键盘控制器硬件使用的键盘编码，用 1 字节表示。原版教材 579 页列出了所有的扫描码和 ASCII 码。常规字符键盘既有 ASCII 码，又有扫描码，通过 ASCII 码就可判断是什么按键。但扩展功能键（光标键、INS、DEL、F1-F10，ALT 或 CTRL 与其它按键的组合键等）则只有扫描码，无 ASCII 码，这些键只有通过扫描码才能进行区分。例如：

```
MOV AH，0
INT 16H ；若按 Home 键，则返回 AX=4700H。
CMP AX，4700H
JZ Home_Pressed
```

每当有击键动作发生，键盘控制器会通过 IRQ1 中断请求线，向中断控制器提出中断请求，若 CPU 响应，则会自动触发硬件中断 INT 9。INT 9 中断服务程序在内部会到 60H 号端口读入所击键的扫描码，然后在内部查表得到该扫描码所对应的 ASCII 码，由 INT 9 负责将扫描码和 ASCII 码送入到键盘缓冲区中。

初始时，40：1A 处的值（头指针）和 40：1CH 处的值（尾指针）都是 001EH（相对于 ROM 数据区的起始位置 40：0 的字节距离）。当头指针等于尾指针时，表示缓冲区为空（键盘无输入）。假

设现在键盘输入一个字符‘A’，则 INT 9 会将‘A’的 ASCII 码 41H 存放到 40：1E 处，而‘A’的扫描码 1EH 则存放到 40：1F 处，并自动修改尾指针（即 40：1C 处）的值为 0020H；假设现在用 INT 16H 读入一个字符，INT 16H 会从头指针所指示的位置将‘A’的 ASCII 码和扫描码读出，然后修改头指针为 0020H。依此类推。

Int 16H 例子见“附件 14、演示热键激活的中断驻留程序”。

第二十七节 磁盘结构

1．磁盘的物理扇区号

硬盘最基本的组成部分是由坚硬金属材料制成的涂以磁性介质的**盘片**，不同容量硬盘的盘片数不等。每个盘片有两面，每个面都有一个**磁头**（Head），都可记录信息。盘片被分成许多扇形的区域，每个区域叫一个**扇区**(Sector)，每个扇区可存储 128×2 的 N 次方（N＝0，1，2，3）字节信息。在 PC 机中，每扇区存放 512 字节，盘片表面以盘片中心为圆心，不同半径的同心圆称为**磁道**(Track)。硬盘中，不同盘片相同半径的磁道所组成的圆柱称为**柱面**(Cylinder)。

扇区是驱动器存取磁盘数据的基本单位。要读写一个扇区，必须告诉磁盘控制器该扇区的扇区地址，即该扇区所在的磁道号、磁头号和扇区号，称为物理扇区号（或绝对扇区号，或 CHS 扇区）。

磁道号：最外圈的称为 0 道，依此住里编号。0 道通常用来存放磁盘的最重要参数。

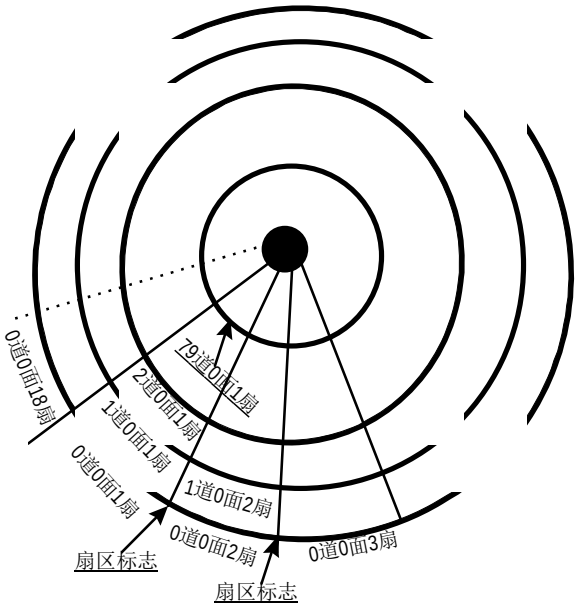
磁头号：从 0 开始编号，直至最大一个磁头号。软盘无标签的一面为 0 面。

扇区号：同一磁道上的扇区号从 1 开始编号，直到该磁道上的最大一个扇区号。

磁盘首扇区的编号是 0 道 0 面 1 扇区，0 道 0 面 0 扇区不存在。

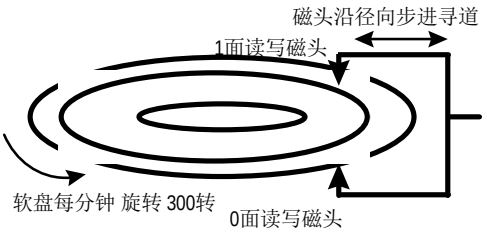
磁盘存储容量＝磁头数×柱面数×每道扇区数×每扇区 512 字节。

1.44M软盘空间划分示意图

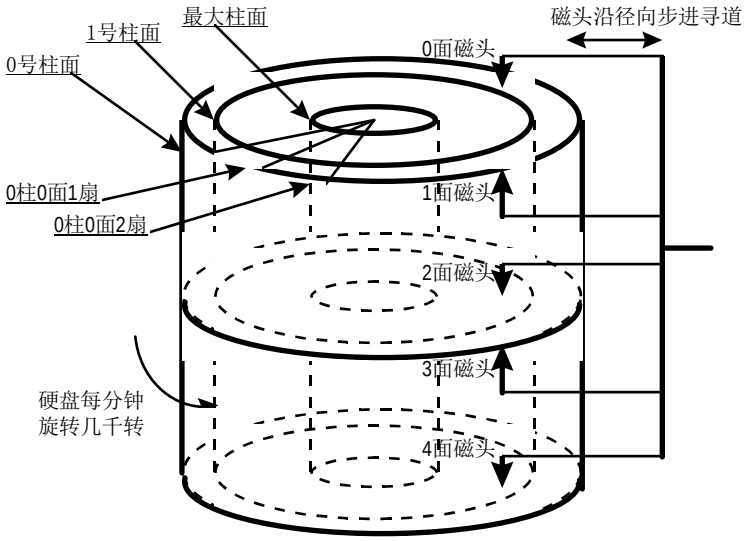


硬盘空间划分示意图

软盘的读写过程示意图

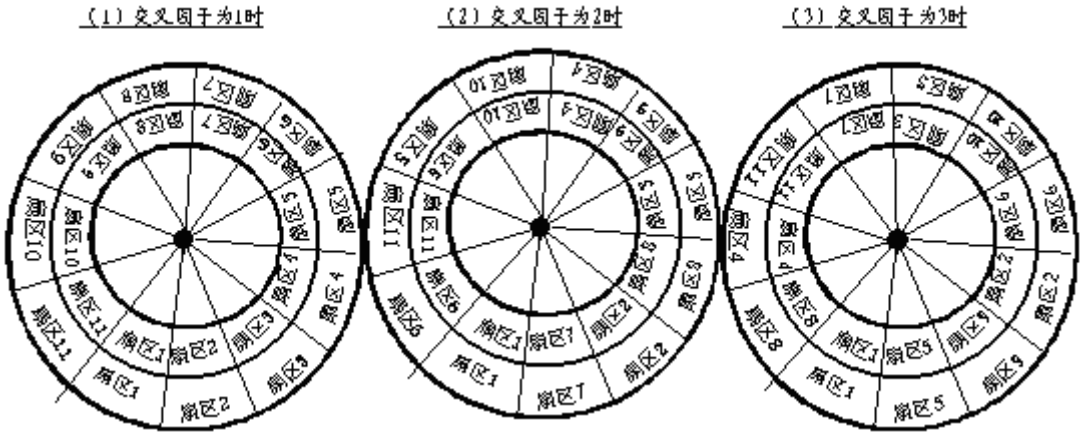


（硬盘相当于摞在一起的一叠软盘。硬盘一个柱面上有很多磁道）



不同交叉因子时的硬盘扇区号排列示意图

（假设每磁道有 11 个扇区）



2. 物理扇区的读写

CHS 扇区(CHS—Cylinder Head Sector)只能由 BIOS 中断 INT 13H 读写。在 INT 13H 中，柱面号用 10 位表示（0-1023），扇区号用 6 位表示（1-63），磁头号用 8 位表示（0-255）。INT 13H 的传统功能主要包括：

功能描述： 读/写物理扇区

输入参数： 功能号 AH=02（读）/03（写）

DL=物理磁盘号

（0=A，1=B，80H=第 1 物理硬盘，81H=第 2 物理硬盘）

CH=柱面号的低 8 位

CL=Bits 6-7 为柱面号的高 2 位，Bits 0-5 为扇区号

DH=磁头号

AL=扇区个数

ES：BX=读写数据的缓冲区首地址

返回参数： 若 CF=1，表示读扇区出错；CF=0，读写成功。

INT 13H 的 AH=05H 号功能可对磁盘格式化。

3、 硬盘分区小知识

硬盘分区是针对一个物理硬盘进行操作的，它可以分为：主分区、扩展分区、逻辑分区（也称为逻辑盘）。其中主分区可以是 1-3 个，扩展分区可以有 0-1 个，逻辑盘则没有什么限制。以下是一些可能的分区方式：

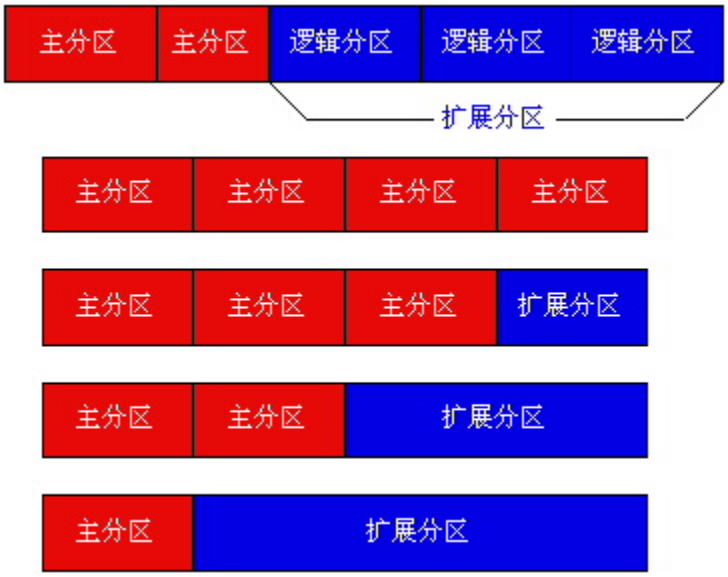


图 1-4-1：硬盘分区关系图

主分区与扩展分区是平级的，扩展分区本身无法用来存放数据，要使用它必须将其分成若干个（1-n 个）逻辑分区。一个硬盘能够分成 4 个主分区、3 个主分区 1 个扩展分区、2 个主分区 1 个扩展分区、1 个主分区 1 个扩展分区。当然，如果你愿望只分一个主分区、或两个主分区，没有扩展分区也是可以的。

也就是说，不管什么操作系统，能够直接使用的只有主分区、逻辑分区。不过不同的操作系统使用的文件系统格式不同，因此在用于不同操作系统后，分区又有了一些新名字：早版本 MSDOS 使用的 FAT16 分区、Windows 使用的 FAT32 分区、Windows NT 使用的 NTFS 分区、Linux 使用的 Ext2 分区及 Swap 分区……等等。

Windows 下的分区：

在 Windows 下，使用“盘符”，如 A、B、C、D、E……等还表示一个分区。这种方法使得硬盘分区这一东西变得十分简单。其中 A 和 B 是软驱，硬盘分区是从 C 开始编号的。下面我们看一看下面的例子(图 1-4-2)：

图 1-4-2：Windows 下分区编号

对于 Windows 而言，它只能够使用一个主分区（在 FDISK，称为主 DOS 分区），可以使用多个逻辑盘。硬盘盘符的编号如上图所示。

在 Windows 下，分区主要用微软的 FDISK 软件完成。也可使用 Partition Magic 等功能更为强大的分区工具完成。Linux 操作系统也有自己的 FDISK 分区软件。在对磁盘分区后，还要对磁盘格式化（FORMAT），格式化时会将文件系统写入到磁盘中。有了文件系统后，磁盘才可使用。

Linux 下的分区：

而在 Linux 下，则复杂一些了。首先，它对每一个设备进行了命名：IDE 设备：一台 PC 上可以有两个 IDE 接口（我将其称为第一硬盘 Primary IDE、第二硬盘 Secondary IDE），而每个 IDE 接口上可以接两个 IDE 设备（我将其称为 Master 主盘、Slave 从盘）。其中硬盘与光驱都是 IDE 设备。Linux 这样为其命名：

```
第一 IDE 的主盘：/dev/hda
第一 IDE 的从盘：/dev/hdb
第二 IDE 的主盘：/dev/hdc
第二 IDE 的从盘：/dev/hdd
```

SCSI 设备：它通常需要加上一块 SCSI 卡来驱动。第一块 SCSI 设备称为：/dev/sda、第二块就是 /dev/sdb … … 以此类推。接下来看一下 Linux 下的分区命名规则：

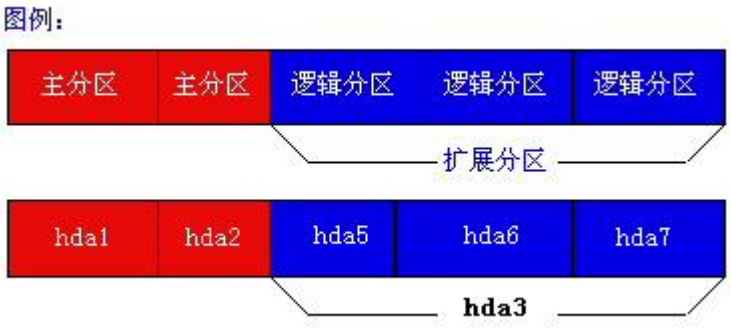


图 1-4-3：Linux 下的分区命名]

也就是主分区（或扩展分区）被命名为 hda1-hda4，如果没有，就跳过。而扩展分区中的逻辑分区则从 hda5 开始编号，以此类推。

注意了，这里是以第一 IDE 的主盘为例，如果是第二硬盘，就是 hdb1、hdb2、hdb5、hdb6、hdb7。

到此，我们可以发现，Windows 下不管有多少个 IDE 设备都是顺序地分配盘符，而在 Linux 下是认真区分对待每一个硬盘的。

为 Linux 划分分区要点：

大家都知道，Windows 下每一个分区都可利用于存放文件，而在 Linux 则除了存放文件的分区外，还需要一个“Swap（交换）分区”用来补充内存，因此通常需要两个分区：

1. 主分区：学习使用的话，建议 2G；

2. 换分区：只需 1-2 倍内存的大小，若 64-128M 内存，交换分区可为 128M，128-256M 内存，交换分区可为 256M。

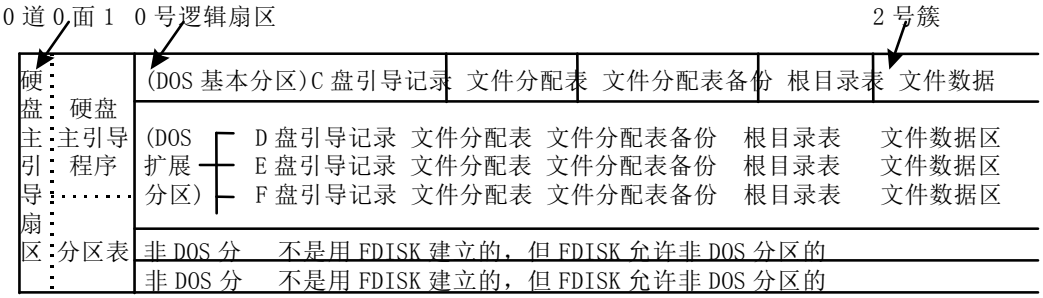
3. 提示 1：由于早期的 Linux 启动器 LILO 无法识别 8G 以外的硬盘分区，因此建议将 Linux 两个分区分在 8G 以内。

   提示 2：每个物理硬盘的 0 道 0 面 1 扇区中存放着 MBR（主引导记录），这里就包括了主分区和扩展分区信息，当机器启动后，将引导交给硬盘时，就首先执行 MBR 上的程序，然后找到活动分区，启动操作系统。象 LILO、OS Loader 等多引导工具都是通过改写 MBR 来实现的。

服务器\\10.40.55.59\public files\Documents & Course Materials (教学文档与学习资料)\Assembly Programming (万金友)\reference materials 中有一篇文章“讲义--《操作系统实践》：PC BIOS & Disk Paramater.htm”，详细详解了磁盘分区的操作方法，请阅读之。

4、硬盘的组织

1. 硬盘主引导扇区位于 0 道 0 面 1 扇区,没有对应的逻辑扇区
2. 每个逻辑盘中的第一个扇区对应的逻辑扇区号是 0,引导记录位于 0 号逻辑扇
   磁盘系统区(包括引导记录,FAT 表,根目录表三部分)只有逻辑扇区号,没有对应的簇号。项
3. 文件数据区用于存放文件的实际内容。簇号从文件数据区开始编起，数据区中的最小
   一个簇号为 2。



主引导扇区及引导记录的组成

名称	组成	作用	何时写入	备注
主引导扇区(主引导记录)	主引导程序	将活动分区的引导记录读入内存	低级格式化，以后可由 FDISK /MBR 重写	一个硬盘上只有一个主引导扇区
	分区表	标识各分区起止位置大小及活动分区	FDISK	
引导记录(引导扇区)	引导程序	读入 IO.SYS 和 MSDOS.SYS 启动操作系统	FORMAT	软盘及硬盘各逻辑盘都有自己独立的引导记录
	磁盘基本参数块(BPB)	标识逻辑盘类型及各种规格参数(如道面扇数、每簇含扇区数等)	FORMAT	

磁盘各部位的产生

磁盘部位	硬盘	软盘	是否可变
扇区标志	低级格式化程序写入	FORMAT 写入	固定
主引导程序	低级格式化程序写入（可由 FDISK/MBR 重写）	无	固定
分区表	FDISK	无	可变
引导记录	FORMAT 写入（可由 SYS 重写）	FORMAT 写入（可由 SYS 重写）	同一操作系统版本时固定
根目录表	初始时由 FORMAT 建立	初始时由 FORMAT 建立	可变
文件分配表	初始时由 FORMAT 建立	初始时由 FORMAT 建立	可变

对于软盘，整个软盘只相当于硬盘的一个逻辑盘（即从引导记录部分开始的部分），软盘的 0 道 0 面 1 扇即软盘的 0 号逻辑扇区。

5、PC 机的引导过程

RESET

↓

从 FFFF：0 处开始执行 ROM BIOS，执行 POST 过程(Power On Self Test)，按 CMOS 中的参数设置进行硬件自检，调用 Video BIOS 等附加 BIOS 程序，设置低端的中断向量和 ROM 数据区，最后调用 INT 19H 从磁盘启动操作系统。

↓

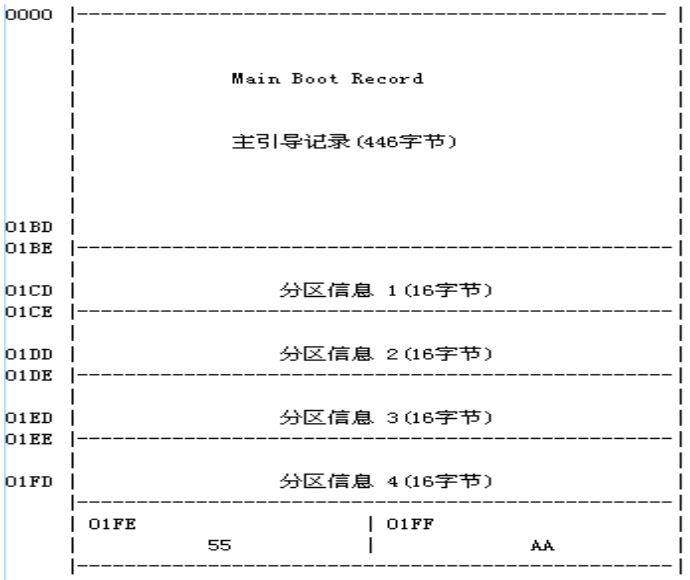
INT 19H 程序：
若软驱中有软盘，则将软盘 0 道 0 面 1 扇的引导记录（即 0 号逻辑扇区）读入到内存，跳转执行其中的引导程序；
否则，将硬盘 0 道 0 面 1 扇的主引导记录读入内存，跳转执行其中的主引导程序。
↓
软盘引导程序：根据引导记录中的磁盘参数表确定软盘格式，并在软盘根目录中查找有无 IO.SYS 和 MSDOS.SYS 文件。若找到，则将这两个文件调入内存，将控制权交给这两个文件，启动操作系统，出现 A>（若找不到，则显示“None System Disk”）。
硬盘主引导程序：根据主引导记录中的分区表确定各分区的起始结束位置（若不能识别分区信息，则显示“Invalid Partition”），然后将活动分区（通常为 C 盘）中的引导记录读入到内存 7C00：0000 处，并跳转至该处执行活动分区中的引导程序。余下过程类似于软盘启动，直到出现 C>。

可见，启动时各部分的执行顺序如下：
（1）软盘启动：ROM BIOS->软盘引导程序->IO.SYS->MSDOS.SYS->CONFIG.SYS->COMMAND.COM->AUTOEXEC.BAT。
（2）硬盘启动：ROM BIOS->主引导程序->活动分区中的引导程序->IO.SYS->MSDOS.SYS->CONFIG.SYS->COMMAND.COM->AUTOEXEC.BAT。

6、硬盘主引导扇区的结构

硬盘的主引导扇区，或叫主引导记录（MBR），不属于任何操作系统。MBR 中包含硬盘的主引导程序和硬盘分区表。其中的主引导程序部分可由 FDISK /MBR 命令重写。

分区表有 4 个分区记录区。记录区就是记录有关分区信息的一张表。它从主引导记录偏移地址 01BEH 处连续存放，每个分区记录区占 16 个字节。



分区表的格式如下：

分区表项的偏移	意义	占用字节数
00	引导指示符	1
01	起始磁头号	1
02	高 2 位为起始柱面号的高 2 位，低 6 位起始扇区号	1
03	起始柱面号的低 8 位	1
04	分区类型指示符	1
05	结束磁头号	1
06	高 2 位为结束柱面号的高 2 位，低 6 位结束扇区号	1
07	结束柱面号的低 8 位	1

08	分区前面的扇区数（隐藏扇区）	4（双字）
0C	分区中总的扇区数	4（双字）

其中，引导提示符：80=活动分区,00=非活动分区（只能有一个活动分区）。分区类型指示符：04=FAT16 分区，0B=FAT32 分区，07=NTFS 分区，82=Linux Native 分区，83=Linux Swap 分区等。
注 1：在用 INT 13H 读写时，可将分区表偏移 02、03 处的两个字节作为一个字看，直接放到 CX 寄存器中。
注 2：尽管主引导扇区只占用 0 道 0 面上的一个扇区，但分区软件一般会保留整个 0 道 0 面上的所有扇区，不分给任何操作系统（对分区 1 而言，就是隐藏扇区数）。所以，分区 1 起始于 0 道 1 面 1 扇，非 0 道 0 面 2 扇。
注 3 ，每个逻辑盘（D、E、F、…）前部也有自己的主引导扇区，其中也有一个相同的分区表（但无主引导程序），其中描述了本逻辑盘的位置和大小，以及下一个逻辑盘的位置和大小，为一链表结构，当下一个逻辑盘对应的 16 字节全为 0 时，表示逻辑盘结束。请参见 Wan.ZIP 中的 GETBOOTS.ASM 和 PUTBOOTS.ASM 程序。

硬盘主引导扇区极为重要，是病毒最易感染的部位之一。可用下列代码将主引导扇区保存到文件中：

```
A>DEBUG
-A100
    MOV AX,0201
    MOV BX,0200
    MOV CX,0001
    MOV DX,0080
    INT 13
    INT 3
-G=100
-RCX
    :200
-N MBR.DAT
-W200
```

以后，当主引导扇区被破坏时，可用下列代码将所保存的 MBR 恢复到主引导扇区中。

```
A>DEBUG
-N MBR.DAT
-L200
-A100
MOV AX,0301
MOV BX,0200
MOV CX,0001
MOV DX,0080
INT 13
INT 3
-G=100
```

主引导扇区中的主引导程序部分可用 FDISK /MBR 命令随时重写，可用于清除其中的病毒。

例 1：经典 CHS 模式硬盘的分区表举例。
以下是一个总容量为 42M（C=560，H=6，S=26 的硬盘）划分为 C 盘（32M）、D 盘（10M）的分区表：

```
                                                  80 01
 01 00 04 05 5A A3 1A 00-00 00 D6 FF 00 00 00 00
```

41 A4 05 05 9A 2E F0 FF-00 00 B4 54 00 00 00 00
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00-00 00 00 00 00 00 55 AA

	引导	起始头	起始扇	起始柱	类型	结束头	结束扇	结束柱	隐藏扇区数	扇区总数
分区 1	80	01	01，00		04	05	5A，A3		0000001A	0000FFD6
	启动	1	1	0	FAT16	05	26(1A)	419(1A3)	26	65494
分区 2	00	00	41，A4		05	05	9A,2E		0000FFF0	000054B4
	不引导	1	1	420(1A4)	FAT12	05	26(1A)	558(22E)	65520	21684

例 2：超过 7.875G 的现代 LBA 32 硬盘的分区表举例。
以下是我院 9 楼机房的硬盘第一个操作系统分区的分区表（装有硬盘保护卡），磁盘总容量 60G（装有硬盘保护卡）。其中 C 盘划为 16G，D 盘（数据盘）划分为 4G。

80 01
01 00 07 FE FF FF 3F 00-00 00 B9 11 F4 01 00 01
C1 FF 07 FE FF FF 30 13-F8 03 3F 04 7D 00 00 00
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00-00 00 00 00 00 00 55 AA

	引导	起始 H	起始 S	起始 C	类型	结束 H	结束 S	结束 C	隐藏扇区数	扇区总数
分区 1	80	01	01，00		07	FE	FF，FF		0000003F	01F411B9
	启动	1	1	0	NTFS	254	63(3F)	1023(3FF)	63	32772537
分区 2	00	01	C1，FF		07	FE	FF，FF		03F81330	007D043F
	不引导	1	1	1023(3FF)	NTFS	254	63(3F)	1023(3FF	66589488	8193087

其中分区表中的起始/结束 CHS 并不反映硬盘的实际参数，INT 13H Extension 不再根据 CHS，而是根据 LBA 扇区号（0~扇区总数-1）在内部进行扇区读写。

7、大容量硬盘的读写

最早期的 BIOS 中的 INT 13H，只支持 10 位柱面号（0-1023），4 位磁头号（0-15），6 位扇区号（1-63）。所以早期的 BIOS 只能识别的硬盘最大硬盘容量为 1024*16*63*512=528M 字节。
90 年代初，随着硬盘容量的增加，BIOS 相应进行了更新，开始支持 8 位磁头号（0-255）。这样的 BIOS 能识别的硬盘最大硬盘容量为 1024*256*63*512=8.4G 字节。
但大于 1G 容量的硬盘，在制造工艺上，主要是通过增加柱面数和每道扇区数的方式增加硬盘容量。实际的柱面数和每道扇区数均已突破了 BIOS 规定的 1024 道和每道 63 个扇区的限制，为了使 BIOS 能接受这些参数，人们借鉴了 SCSI 接口读写硬盘的方法，发明了称为 LBA 方式的硬盘读写方法（Logic Block Address，逻辑块地址）。其思想是，在保持硬盘扇区总数不变的前提下，提供 BIOS 一个 C/H/S 在 1024/256/63 范围的假想硬盘参数（逻辑参数），然后由支持 LBA 方式的 BIOS 在内部转换成硬盘的实际物理参数，进行硬盘的读写。例如，一个物理参数为 C/H/S=4096/16/63 的硬盘可转换为一个逻辑参数为 C/H/S=1024/64/63 的“合法”硬盘，即可欺骗 BIOS。所谓 LBA 是指用扇区的顺序号表示每个扇区的地址（LBA 号从 0 编号至最大扇区数减 1）。对于 CHS 扇区，对应的 LBA 扇区号为：
LBA = C * 磁头数* 每道扇区数 + H *每道扇区数 + (S - 1)

硬盘驱动器硬件根据 LBA 号，在内部进行变换（对 BIOS 不透明），即可找到相应的扇区。这种 LBA 扇区号是 24 位的。

对于超过 1G 容量的硬盘，必须通过 LBA 方式才能读写（在 CMOS 设置中硬盘读写有 CHS/LARGE/LBA 三种方式可选）。此时在 CMOS 设置中所看到的 C/H/S 参数都不代表硬盘的实际物理参数，只是表示硬盘的总容量而已。

以上讲的通过 LBA 方式读写硬盘，仍是通过传统的 INT 13H 读写，未能突破 CHS=1024/256/63 的总扇区个数限制，即 8.4G 的容量限制。90 年代中期，随着 10G 以上容量的硬盘出现（称为现代硬盘，或 EDD 硬盘，Enhanced Disk Driver），Microsoft / IBM / Phoenix 等公司联合对 BIOS 进行了改造，增加了 INT 13H Extension 部分(AH=4x)，专门支持对特大硬盘的读写，97 年以后制造的 BIOS 都支持 INT 13H Extension。INT 13H Extension 不再使用传统的 CHS 作为硬盘读写参数，而是使用 LBA 号作为所读写扇区的物理地址，由驱动器硬件在内部根据 LBA 号找到相应的扇区。

LBA 方式 是一种线性块地址表示方式。INT 13H Extension 规定 LBA 号用 64 位表示，能表示的扇区数有 2 的 64 次方个，最大容量为 8TG（天文数字）；目前大多数硬盘可接受的是 32 位的 LBA 号（Linux 中称为 LBA32），能表示的扇区总数为 2 的 32 次方个，即 4G 个，能识别的最大硬盘容量为 4G*512=2T。Phoenix 认为 LBA32 至少可用 15 年。

其实，在现代大容量硬盘的内部，每道扇区数已不再相同，从外圈到内圈，每道扇区数由高至低变化，与传统的 CHS 概念已不相同。对于现代大容量硬盘，出厂时已低级格式化过，若使用通用低级格式化程序对其进行低级格式化，虽不会造成硬盘损坏，但等于浪费时间 （驱动器内部会屏蔽这些低格操作）。若坚持低格，请到硬盘制造商的网站上下载其专用的低格软件，因为只有硬盘制造商自己才知道其硬盘的内部格式。“Burn down your house to get rid of the termites!”。

一台 PC 要能读写大容量的硬盘，除了 BIOS 要支持 INT 13H Extension（即 LBA32）外，操作系统使用的硬盘驱动程序也必须使用 INT 13H Extension 读写硬盘，Win95 及以后的 Windows 版本，及 Red Hat Linux 7 之后，均支持 INT 13H Extension。但很多操作系统，都要求操作系统内核本身必须安装在硬盘最前部的 8.4G 以内，这是因为启动时，这些操作系统仍使用的是传统的 INT 13H 读写硬盘，在识别出硬盘类型，装载硬盘驱动程序后，才能开始使用 INT 13H Extension 进行硬盘读写。这是一个”Chick and Egg”问题。

若想更全面了解 LBA32，在\\10.40.55.59\public files\Documents & Course Materials (教学文档与学习资料)\Assembly Programming (万金友)\reference\INT 13H Extension 子目录中，有几篇精彩的资料。

在 Red Hat Linux 8 安装时，可选是否采用 LBA32 的高级选项（对 Redhat 7，要修改 /etc/lilo.conf，将 linear 改为 lba32）。另外，在装有 Windows 的硬盘上，最好不要冒险将 Linux 引导程序(LILO)装在 MBR 中（高级选项），而是应装在分区的引导记录中，否则 Linux 装完后，你原来的 Windows 就可能就不能启动了。

8、逻辑扇区读写

DOS/Win 操作系统在内部使用的是逻辑扇区号，C 盘或逻辑盘每个盘内部的扇区均从 0 开始独立编号，直至该盘的最后一个扇区，该编号称为逻辑扇区号。逻辑扇区号需转换成 CHS 扇区号，才能被驱动器所接受。

软盘的道面扇数

(扇区总数=磁道数*每道扇区数*面数)

软盘	磁道数(编号)	每道扇区数(编号)	面数(编号)	扇区总数(逻辑扇区号)
1.2M	80(0～79)	15(1～15)	2(0～1)	2400(0～2399)
1.44M	80(0～79)	18(1～18)	2(0～1)	2880(0～2879)

1.44M 软盘物理扇区与逻辑扇区编号的对应关系

物理扇区号	0 道 0 面 1 扇	0 道 0 面 2 扇	…	0 道 0 面 18 扇
逻辑扇区号	0	1	…	17

物理扇区号	0 道 1 面 1 扇	0 道 1 面 2 扇	…	0 道 1 面 18 扇
逻辑扇区号	18	19	…	35
物理扇区号	1 道 0 面 1 扇	1 道 0 面 2 扇	…	1 道 0 面 18 扇
逻辑扇区号	36	37	…	53
…	…	…	…	…

（1）CHS 转换成逻辑扇区号 L 的公式为：
L= C * 磁头数* 每道扇区数 + H *每道扇区数 + (S - 1)
（2）逻辑扇区号 L 转换成 CHS 的公式为：
S=（L MOD 每道扇区数）+1
H=（L DIV 每道扇区数）MOD 磁头数
C=（L DIV 每道扇区数）DIV 磁头数
辑扇区号到 CHS 扇区号的转换是由操作系统在内部自动完成的。

读逻辑扇区只能使用 INT 25H（对应 DEBUG 的 L 命令），写逻辑扇区只能使用 INT 26H（对应 DEBUG 的 W 命令）。用法如下：

输入参数：
AL=逻辑盘号（0=A，1=B，2=C，3=D，4=E，…）
DX=起始逻辑扇区号
CX=扇区个数
DS:BX=读写缓冲区首地址
返回参数：
CF=0 成功
CF=1 出错，此时 AL=出错码

要注意，由于 INT 25/26H 中断程序内部不是用 IRET 结束，而是用 RETF 结束的，所以在调用 INT 25/26H 后，应立即跟一句 POPF 或 ADD SP，2 指令，以使栈指针恢复正常。

另外，在 Win2K 中，只能使用 INT 25/26/13H 对软盘进行读写。对硬盘读写只能在纯 DOS 环境中进行（用软盘启动）。

DEBUG 的 L 或 W 命令用法如下：
-L（或 W） 缓冲区首地址 逻辑盘号 起始逻辑扇区号 扇区个数

9、DOS 逻辑盘结构
DOS 逻辑盘的结构
每个逻辑盘内部，从 0 号逻辑扇区开始，依此是：
引导记录(1)、FAT(9)、备份 FAT(9)、根目录表(14)、磁盘数据区(2847)。
括号中为 1.44M 软盘各部分所占的扇区数。
其中磁盘系统区（引导记录、FAT 表、根目录表）只能根据逻辑扇区号读写（无对应簇号），用 INT 25/26H 读写。
磁盘引导记录(DBR)
DBR 位于逻辑盘中的 0 号逻辑扇区。DBR 分为两部分：DOS 引导程序和 BPB（BIOS 参数块）。其中 DOS 引导程序完成 DOS 系统文件（IO.SYS，MSDOS.SYS）的定位与装载，而 BPB 用来描述本 DOS 磁盘的规格参数，BPB 位于 DBR 偏移 0BH 处，共 25 字节。它包含逻辑格式化时（FORMAT）使用的参数，可供 DOS 计算磁盘上的文件分配表，目录区和数据区的起始地址，BPB 之后三个字提供物理格式化（低格）时采用的一些参数。引导程序或设备驱动程序根据这些信息将磁盘逻辑扇区号转换成 CHS 扇区号。

表：BPB 格式：

偏移	意义	1.44M 盘的参数例子
（00H-02H）	JMP 的机器码：跳至引导程序	EB 34 90
03H－0AH	OEM 名和版本	IBM 3.3
0BH－0CH	每扇区字节数	0200H
0DH	每簇扇区数	01H
0EH－0FH	保留扇区数	0001H（指引导扇区本身所占的扇区）
10H	FAT 表个数	02H
11H－12H	根目录项数（允许最大数）	00E0H（根最多 224 个文件或子目录）
13H－14H	扇区总数（从 0 号逻扇始）	0B40H（即 2880）
15H	磁盘介质说明符	F0H（表示是 1.44M 软盘）
16H－17H	每 FAT 所占的扇区数	0009H
18H－19H	每磁道扇区数	0012H
1AH－1BH	磁头数	0002H
1CH－1FH	隐藏扇区总数	见硬盘分区表，软盘为 0
20H－23H	扇区总数	见硬盘分区表，软盘为 0
24H－25H	物理驱动器数	
26H	扩展引导签证	
27H－2AH	卷系列号	
2BH－35H	卷标号	
36H－3DH	文件系统号	

对于 1.44M 软盘：
引导记录起始逻辑扇区号=0
（L 200 0 0 1）
FAT 起始逻辑扇区号=1（即保留扇区数）
（L 200 0 1 9）
备份 FAT 起始逻辑扇区号=保留扇区数+FAT1 占扇区数=1+9=10
（L 200 0 A 9）
根目录表起始逻辑扇区号=保留扇区数+FAT1 占扇区数*2=1+9*2=19
根目录表扇数=(根表项数*每表项 32 字节)/每扇 512 字节=（224*32）/512=14
（L 200 0 13 E）
数据区起始逻扇号=保留扇区数+每 FAT 扇数*2+根目录表扇数=1+9*2+14=33
（L 200 0 21 1）

磁盘数据区的簇号

文件中包含的实际内容及子目录中的文件目录列表，均存放磁盘数据区中。DOS 将磁盘数据区中的扇区划分成簇(Cluster 或 Allocation Unit)，每簇由几个扇区组成，规定最小的一个簇号为 2。例如 1.44M 软盘每簇占 1 个扇区，磁盘数据区中共有 2880-33=2847 个簇，簇号从 2 至 2848。簇是 DOS 为文件存储分配磁盘空间的基本单位。若一个簇中包含的扇区数较多，则文件尾簇将浪费较多空间；太少则需要较宽的 FAT 表位数表示簇号。

DOS 操作系统没有提供可直接使用簇号的中断可用，读簇中内容时，需将簇号转换成逻辑扇区号，然后用 INT 25/26H 读写对应的逻辑扇区。转换关系：

逻辑扇区号=数据区起始逻辑扇区号+（簇号-2）*每簇扇区数

对 1.44M 软盘，逻辑扇区号=33+(簇号-2)*1

文件分配表

FAT 表是 DOS 文件组织结构的主要组成部分，FAT 表反映硬盘上所有簇的使用情况，通过查文件分配表可以得知任一簇的使用情况。DOS 在给一个文件分配空间时总先扫描 FAT，找到第一个可用的空闲簇，将该空间分配给文件，并将该簇的簇号填到相应的 FAT 表表项中，作为该文件占用的上一个簇的后继簇，最终，每个文件所占用的数据空间在 FAT 表中会形成一个“簇号链”。FAT 就是记录文件簇号拉链的一张表。FAT 的头两个域为保留域，对 FAT12 来说是 3 个字节，FAT16 来说是 4 个字节。其中头一个字节是用来描述介质类型的，其余字节为 FFH。所以数据区的簇号从 2 开始编起。

FAT 表中每个表项中存放的是簇号，对于软盘和小于 10M 的硬盘，每个表项占 12 位宽（即簇号用 12 位表示），称为 FAT12 文件系统；容量在 10-512M 之间的硬盘使用 FAT16；容量在 512M-2G 之间的硬盘可选使用 FAT16 或 FAT32；容量在 2G 以上的硬盘只能使用 FAT32。

FAT 表中的表项内容的含义

12 位表项	16 位表项	含义
000H	0000H	本簇未被占用
FFFH	FFFFH	本簇已占用,且是某文件的最后一簇
FF7H	FFF7H	本簇是坏簇
其它(002H～FEFH)	其它(0002H～FFEFH)	本簇已占用,该值是某文件下一簇的簇号

一个文件所占用的首簇号（链表表头）并不存在 FAT 中，而是存放在文件目录表表项中（偏移 26、27 处的一个字），FAT 表中存放的是除该文件占用的首簇外其余簇的簇号，该链表最终在 FAT 表中以（F）FFFH 结束。

设文件的当前簇号为 n，在 FAT 表中查找其后继簇的方法如下：

FAT12

将 n*1.5 作偏移地址，从 FAT 中取出该偏移处的一个字。若当前簇号 n 为偶数，则取该字的低 12 位作为后继簇的簇号；否则取该字的高 12 位作为后继簇号。

将后继簇号作为当前簇号，重复上述过程，直到所取字为 FFFH 为止。

FAT16

将 n*2 作偏移地址，从 FAT 中取出该偏移处的一个字，即后继簇的簇号；

将后继簇号作为当前簇号，重复上述过程，直到所取字为 FFFFH 为止。

根目录表

文件目录表是 DOS 文件组织结构的又一重要组成部分。分根目录表和子目录表两类。根目录表存于磁盘系统区中，但子目录被当作一种特殊文件处理，所以子目录表存于磁盘数据区中。DOS 为每个文件目录表中的每个表项分配 32 字节，用于描述该目录中的每个文件或子目录的名字、属性等信息。

8.3 格式文件目录表中每个表项的结构

1 8	9 11	12	13 22	23 24	25 26	27 28	29 32
文件目录主名	扩展名	属性	保留	时间	日期	首簇号	文件长度

其中，文件主名特殊首字节：00H 代表未使用， E5H 代表此文件已被删除。另外，WINDOWS 的长文件名使用“保留”区域存放长名字符。

属性字节中的含义

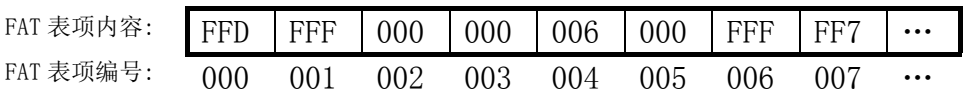
第 7 位	第 6 位	第 5 位	第 4 位	第 3 位	第 2 位	第 1 位	第 0 位
保留	保留	归档	子目录	卷标	系统	隐含	只读

（1）若子目录位为 1，表示该表项是一个子目录，0 表示是一个文件

（2）若第 1 位至第 4 位全为 1（即属性字节为 0FH），表示本 32 字节是一个 Win95 长名表项。

建立和修改一个文件时的内部处理过程图解

现准备在磁盘上新建一个名为 TEST.TXT 的文件。假设在该文件建立前，FAT 表（设表项为 12 位）中的部分内容如下图所示（FAT 表中前两个表项的内容 FFD 和 FFF 为系统内部所使用）：



1. 初始建立文件时

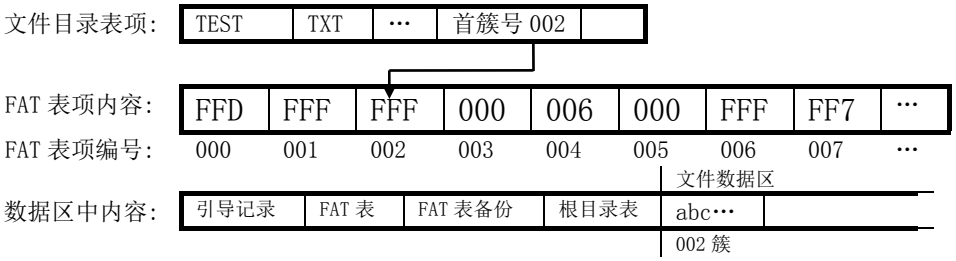
(1)首先为该文件在文件目录表中建立一个 32 字节的表项，并登记文件名等信息。

(2)然后在 FAT 表中查找内容为 0 的表项，假设第 002 个表项为所找到的第一个未用簇，则 002 号簇将作为该文件的首簇号。将首簇号登记在刚建立的文件目录表表项中。

(3)将文件首簇的实际内容(假设内容为“abc…”)写入数据区中 002 号簇所对应的各扇区中。

(4)修改 FAT 表，将第 002 号表项的内容改为 FFFH，表示 002 号簇是该文件的尾簇。

(5)将文件属性（带归档标志）、日期、时间、文件长度等登记在文件目录表相应表项中。



2. 文件内容增加，需再分配一个新簇时

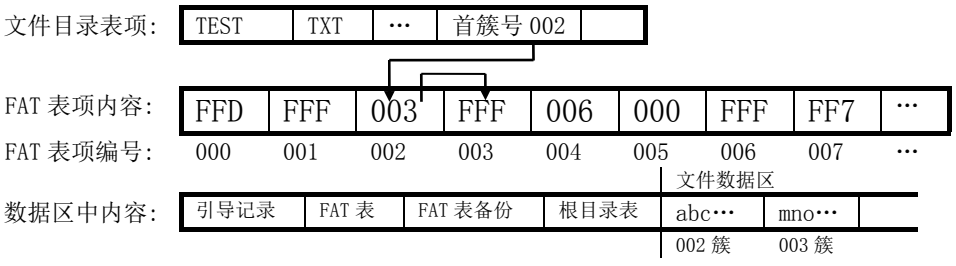
(1)在 FAT 表中查找内容为 0 的表项，本次第 003 个表项为所找到的第一个未用簇，则 003 号簇将作为该文件下一簇的簇号。

(2)修改 FAT 表：将第 002 个表项(文件原来的尾簇号)的内容改为 003，表示 002 号簇的后继簇号是 003。

(3)将文件的新增内容(假设内容为“mno…”)写入数据区中 003 号簇所对应的各扇区中。

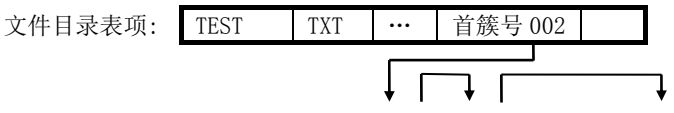
(4)修改 FAT 表：将第 003 号个表项的内容改为 FFFH，表示 003 号簇是文件的尾簇。

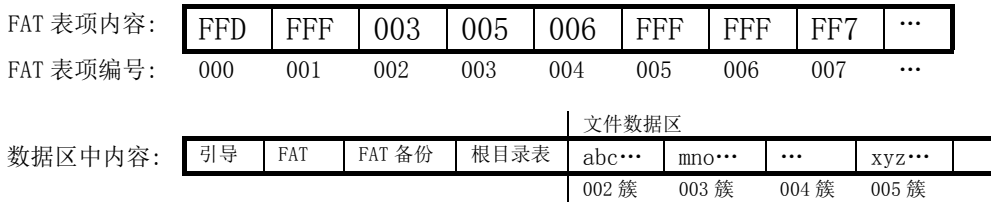
(5)将新的属性（带归档标志）、日期、时间、文件长度等信息登记在文件目录表相应表项中。



3. 文件内容继续增加，需再分配一个新簇时

本次在 FAT 表中找到的第一个未用簇位于第 005 个表项中，所以 005 号簇是该文件的下一簇的簇号。整个处理过程与对刚才分配 003 号簇时类似。如下图所示（假设本次新增内容为“xyz…”）：





4. 至此，该文件共占用簇号分别为 002,003,005 的三个簇。

其中 003 号簇与 005 号簇之间存在文件碎片（006 号簇先前已被其它文件占用）。

建立一个子目录时的内部处理过程图解

假设在根目录中新建一个名为\USR 的子目录。其处理过程与建立文件时基本类似。

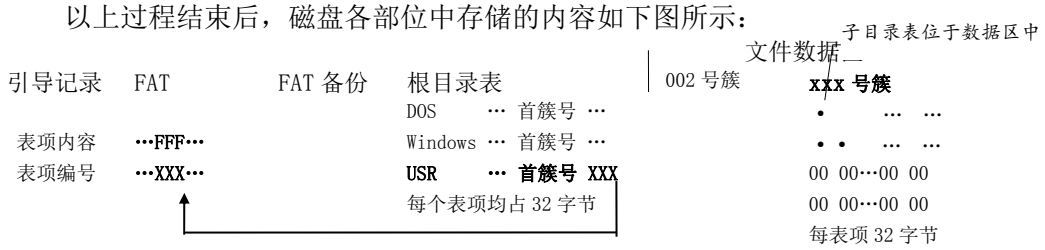
1. 首先在根目录表中为该子目录建立一个 32 字节的表项，将子目录名 USR 等登记在该表项中。

2. 然后在 FAT 表中查找内容为 0 的表项，假设第 XXX 个表项为所找到的第一个未用簇，则 XXX 号簇将作为该子目录的首簇号。将首簇号登记在根目录表中的相应表项中。

3. 在数据区中对刚分配的那个簇（即第 XXX 号簇）进行初始化，具体操作包括：将该簇各扇区中的内容全清为 0；在该簇首扇区中建立一个名为“.”的表项（表示当前目录）和一个名为“..”（表示上级目录）的表项，并在每个表项中登记日期、时间等信息，每个表项均占 32 字节。初始时，子目录中只有这两个固定表项，首簇中其余表项(内容全为 0)留给以后在该子目录中所建立的文件或下级子目录使用。

4. 修改 FAT 表，将第 XXX 号表项中的内容改为 FFFH，表示第 XXX 号簇是该 USR 子目录的尾簇。

以上过程结束后，磁盘各部位中存储的内容如下图所示：



第二十八节 编写中断服务程序（TSR 程序）

中断服务程序也叫 TSR 程序（Terminated but Stay Resident program，终止但保持驻留的程序）。设计中断服务程序，要编写**安装部分**（相当于运载火箭）和**常驻内存部分**（相当于运载火箭所发射的卫星）两部分程序。其中安装部分只运行一次，其任务是负责将常驻部分驻留到内存中。常驻部分即通常所说的中断服务程序，安装后便一直常驻在内存中，以后每次调用该中断，常驻部分都会被调用。设计常驻部分时要符合对中断服务程序的编写要求，如不能调用不可重入的中断等。

一、安装部分的编写

- **保存原中断程序入口地址到常驻部分中**
- **设置新的中断程序入口地址**
- **调用 INT 21H 的 AH=31H 号功能或用 INT 27H 将常驻部分驻留到内存中**

1. 如何保存原中断入口地址

事先须在常驻部分中定义一个双字类型的变量（如 old_int_vector），然后到中断向量表中取该原中断的入口地址保存到该变量中。

取原中断入口地址当然可直接到 0:INT_VECTOR*4 处取，但建议使用 INT 21H 的 AH=35H 号功能：

```
mov ah,35h            ；功能号 35H：取中断入口地址
mov al,INT_VECTOR     ；所取的中断号
int 21h
mov word ptr cs:old_int_vector,bx；返回 es:bx=中断程序的段地址：偏移
mov word ptr cs:old_int_vector+2,es
```

2. 如何设置新的中断入口地址

设置新中断入口地址可直接修改 0:INT_VECTOR*4 处的值，但建议使用 INT 21H 的 AH=25H 号功能：

```
mov ah,25h            ；功能号 25H：设置中断入口地址
mov al,INT_VECTOR     ；所设置的中断号
push cs                       ；使 ds:dx=新中断程序的段地址：偏移
pop ds
mov dx,offset new_int_proc
int 21h
```

3. 如何使程序驻留内存

常规程序使用 INT 21H 的 AH=4CH 号功能或 INT 20H（只对 COM 程序）结束，但中断程序必须使用 INT 21H 的 AH=31H 号功能或 INT 27H（只对 COM 驻留程序）结束，否则程序不能驻留内存。

- 使用 INT 21H 的 AH=31H 号功能使程序驻留内存（适于所有程序，推荐使用）

功能：使程序驻留内存
输入参数：
ah=31H
al=返回码 （可不设）
dx=驻留长度（从 PSP 算起，以小节为单位，1 小节=16 字节）
返回参数：无

例如：

```
mov ax,3100h
mov dx,offset INSTALL  ; INSTALL 以前为应驻留部分的代码
add dx,100h   ; 对 EXE 必须加 PSP 长度，因对 EXE，CS！=PSP
              ; 但对 COM，此句可省，因 COM 的偏移已包含了 PSP(CS=PSP)。
mov cl,4            ; 驻留长度/16，转换成小节
shr dx,cl
inc dx              ; 考虑到可能有余数，留一小节余量
int 21h             ; 使驻留
```

- 使用 INT 27H 使程序驻留内存（仅适于 COM 程序，推荐淘汰）

功能：使程序驻留内存
输入参数：
dx=驻留长度（从 PSP 算起，以字节为单位）

例如：

```
mov dx,offset INSTALL  ; INSTALL 以前为应驻留部分的代码
int 27h                ; 使驻留
```

二、常驻部分的编写

（1）第一种结构：先执行完原中断程序，再执行新增加的部分（如热键程序）。
此类中断程序的结构如下：

```
pushf
call dword ptr cs:old_int_vector
新增加的代码部分
iret
```

（2）第二种结构：先执行完新增加的部分，再执行原中断程序（如定时音乐程序）。
此类中断程序的结构如下：

```
新增加的代码部分
jmp dword ptr cs:old_int_vector
```

三、防止中断驻留程序重复驻留

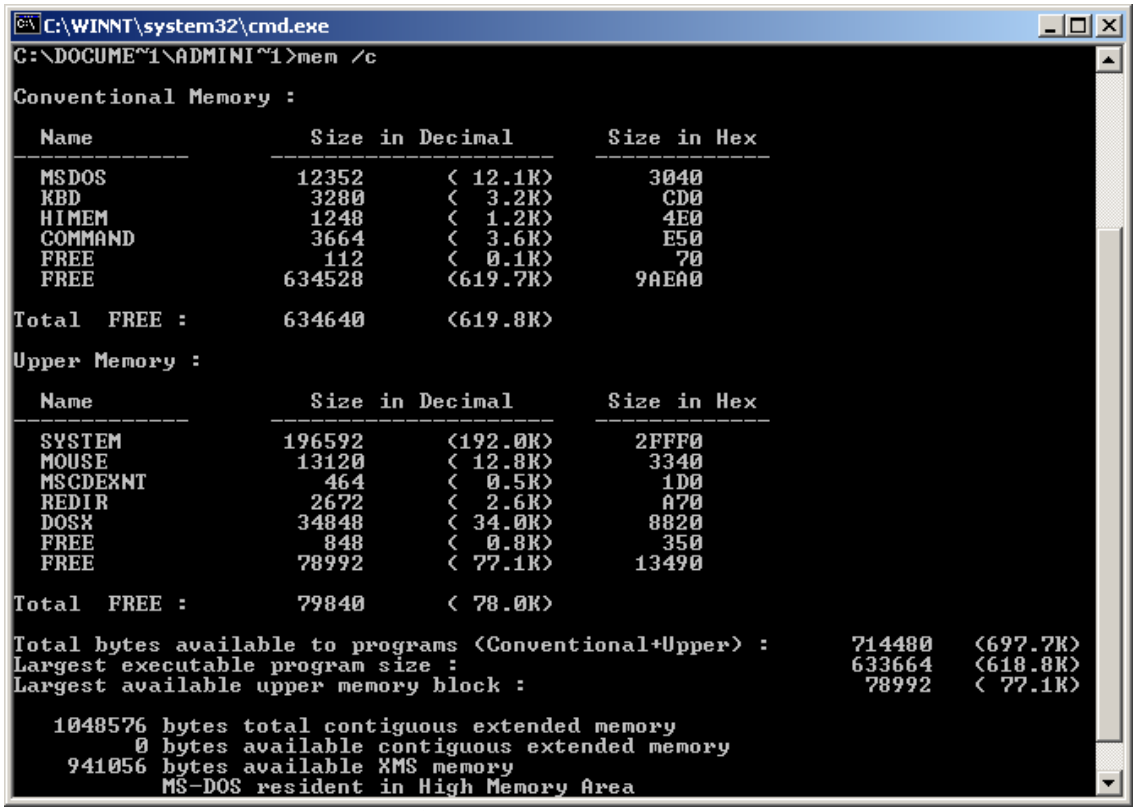
一个好的 TSR 程序应能防止重复驻留，并提供将本 TSR 从内存中卸除的功能。在附件的 SETINT1C.ASM 和 SETINT9.ASM 两个程序都没有判断先前该程序是否已驻留内存的代码。但 SETDOSVE.ASM 程序包含了这部分代码。其思想是，除将取版本号的 INT 21H 的 AH=30H 号功能驻留外，同时设置并驻留了一段专用于判断本程序是否已留过的代码（该例子设置的是 AX=F1F1H 这个 INT 21H 没有使用的功能）。在每次运行 SETDOSVE 程序时，安装部分首先以 AX=F1F1H 调用 INT 21H，若返回 AX=1F1FH，则表示我们的程序已经驻留过，不应再次驻留内存，否则说明未驻留过，这时可将本程序驻留。

四、TSR 程序举例

见附件 SETINT1C.ASM、SETINT9.ASM、SETDOSVE.ASM 程序。

五、查看内存中的中断驻留程序

可使用 MEM 命令（该程序位于\winnt\system32 目录中），列出内存中的驻留程序。
简单查看，可使用 MEM /C 命令：



更详细的情况，可使用 MEM /D 命令查看：

```
C>mem /d

  Address      Name           Size       Type
  -------      --------       ------     ------
  000000                      000400     Interrupt Vector
  000400                      000100     ROM Communication Area
  000500                      000200     DOS Communication Area

  000700       IO             000370     System Data
               CON                         System Device Driver
               AUX                         System Device Driver
               PRN                         System Device Driver
               CLOCK$                      System Device Driver
               COM1                        System Device Driver
               LPT1                        System Device Driver
               LPT2                        System Device Driver
               LPT3                        System Device Driver
               COM2                        System Device Driver
               COM3                        System Device Driver
               COM4                        System Device Driver

  000A70       MSDOS          001650     System Data

  0020C0       IO             002070     System Data
                 KBD          000CD0      System Program
                  HIMEM        0004E0       DEVICE=
                   XMSXXXX0                  Installed Device Driver
                                000490      FILES=
```

```
                        000090      FCBS=
                        000170      LASTDRIVE=
                        0007D0      STACKS=
  004140      COMMAND     000A20     Program
  004B70      MSDOS         000070     -- Free --
  004BF0      COMMAND     000500     Environment
  005100      REDIR           000A70     Program
  005B80      DOSX            0087A0     Program
  00E330      SETINT1C      000510     Environment
  00E850      SETINT1C      000200     Program
  00EA60      MEM             000510     Environment
  00EF80      MEM             017550     Program
  0264E0      MSDOS         079B00     -- Free --
  09FFF0      SYSTEM        03C000     System Program

  0DC000      IO                  003350     System Data
                MOUSE           003340      System Program
  0DF360      MSDOS         000520     -- Free --
  0DF890      MSCDEXNT   0001D0     Program
  0DFA70      DOSX            000080     Data
  0DFB00      MSDOS        0004F0     -- Free --


    655360 bytes total conventional memory
    655360 bytes available to MS-DOS
    594016 largest executable program size

   1048576 bytes total contiguous extended memory
         0 bytes available contiguous extended memory
    941056 bytes available XMS memory
           MS-DOS resident in High Memory Area
```

其中显示的以下两行表示 SETINT1C 占用的内存空间及地址：

```
  00E330      SETINT1C      000510     Environment
  00E850      SETINT1C      000200     Program
```

Environment 表示其环境块所占用的内存空间，环境块中存放的是本程序对应的环境变量（如 PATH 等设置）。若想观察该环境块，可使用 DEBUG 命令：

```
-D E33:0
0E33:0000  4D 86 0E 51 00 61 6E 6E-6F 74 20 72 75 6E 20 74   M..Q.annot run t
0E33:0010  43 4F 4D 53 50 45 43 3D-43 3A 5C 57 49 4E 4E 54   COMSPEC=C:\WINNT
0E33:0020  5C 53 59 53 54 45 4D 33-32 5C 43 4F 4D 4D 41 4E   \SYSTEM32\COMMAN
0E33:0030  44 2E 43 4F 4D 00 41 4C-4C 55 53 45 52 53 50 52   D.COM.ALLUSERSPR
0E33:0040  4F 46 49 4C 45 3D 43 3A-5C 44 4F 43 55 4D 45 7E   OFILE=C:\DOCUME~
0E33:0050  31 5C 41 4C 4C 55 53 45-7E 31 00 41 50 50 44 41   1\ALLUSE~1.APPDA
0E33:0060  54 41 3D 43 3A 5C 44 4F-43 55 4D 45 7E 31 5C 41   TA=C:\DOCUME~1\A
0E33:0070  44 4D 49 4E 49 7E 31 2E-53 43 2D 5C 41 50 50 4C   DMINI~1.SC-\APPL
```

Program 是程序本身所占用的内存空间。0E850H 是本程序对应的内存控制块 MCB 的段地址（MCB 共 16 字节），MCB 之后是 256 字节的程序段前缀 PSP，PSP 之后就是我们的代码段。可使用 DEBUG 命令观察：

```
-D E85:0
```

```
0E85:0000  4D 86 0E 20 00 40 00 8E-53 45 54 49 4E 54 31 43   M.. .@..SETINT1C
0E85:0010  CD 20 A6 0E 00 9A F0 FE-1D F0 DE 01 15 04 4B 01   . ............K.
0E85:0020  15 04 56 01 15 04 15 04-01 01 01 00 02 FF FF FF   ..V.............
0E85:0030  FF FF FF FF FF FF FF FF-FF FF FF FF 34 0E E6 FF   ............4...
0E85:0040  86 0E 14 00 18 00 86 0E-FF FF FF FF 00 00 00 00   ................
0E85:0050  05 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0E85:0060  CD 21 CB 00 00 00 00 00-00 00 00 00 00 20 20 20   .!...........
0E85:0070  20 20 20 20 20 20 20 20-00 00 00 00 00 20 20 20          .....
-D
-D
```

第二十九节　在 C 语言中调用汇编子程序（原版教材 437，440 页）

1）调用约定问题（calling convention）

混合语言编程，只能在生成的目标模块（OBJ）一级进行连接(link)。OBJ 中包含的是一定格式的汇编码。要实现混合语言的连接，必须明白语言的调用约定问题。程序设计语言的调用约定分 **c 约定**和 **pascal 约定**两大类。除 c 语言外，其它语言（包括汇编）都使用 pascal 约定。其区分如下：

- **大小写问题(Case-Sentive)**

在 c 约定生成的汇编码中，符号严格按 c 程序中的大小写生成，而 pascal 约定则将符合均转换成大写。

- **下划线问题(Underbar)**

c 约定生成的符号均自动加下划线，而 pascal 约定则不加下划线。

例如,变量名 i 和函数名 add 生成的汇编码如下：

c 约定：_i，_add

pascal 约定：I，ADD

- **参数压栈次序问题**

在调用函数时，c 约定是将函数参数倒序压栈（从右往左），而 pascal 约定则是正序压栈（从左到右）。

- **压栈参数谁负责处理**

当从函数调用返回时，c 约定规定由调用程序(caller)负责去除栈中的函数参数，而 pascal 约定则规定由被调用程序(callee)自行负责去除栈中的函数参数。

例如 add(int i,int j,int k)的生成的 c 约定代码如下(small 模式)：

调用程序：

```
push _k
push _j
push _i
call near ptr _add
add sp,6
```

被调用程序_add：

```
_add proc near
       …
       ret
_add endp
```

而生成的 p 约定代码如下:

调用程序：

```
push I
push J
push K
call near ptr ADD
…
```

被调用程序 ADD：

```
ADD proc near
        …
        ret 8
ADD endp
```

其中 ret n 的含义是，在 ret 弹栈的同时，再从栈中额外地多弹掉 n 个字节。

原因：因 c 语言中有些函数的参数个数可变（有些 c 函数最右边的参数可缺省，如 printf（）函数的参数个数可变），所以只有 caller 才搞得清，到底代入了几个参数，所以栈中的函数参数只能由 caller 负责去掉。其缺点是，若一个函数被反复调用，则每个 caller 都要带重复的清栈代码，生成的程序较长。采用 pascal 约定的语言，参数个数固定，所以可由 callee 负责统一清栈。所以 P 约定生成的程序短。

相应地，在 Win32 SDK 程序中，函数可有__pascal、__cdecl 两种类型，另外还有__stdcall 类型。其中__pascal 类型只有在 Win16（如 Windows 3x）中使用，在 Win32 中已经被完全淘汰。

__stdcall 约定是 c 约定和 p 约定的综合，除被调用函数(callee)负责清栈这一点同 p 约定外，其它方面均同 c 约定。__stadcall 约定规定由 callee 负责清栈的目的是为了使生成的 Win32 程序短些。Win32 SDK 内部的所有 API 均为__stdcall 类型。Win32 中的 CALLBACK 和 WINAPI 类型也均属__stdcall 类型。

__cdecl 约定即 c 约定，在 Win32 程序中，只有那些用 c 约定编写的函数或将被其它语言调用的 c 函数，才应被声明为__cdecl 类型，或用 extern “C”括起来说明，如：

```
// Sample.h
#if defined(__cplusplus)
extern "C"
{
#endif
     int   atoi( char *string );
     long atol( char *string );
#if defined(__cplusplus)
}
#endif
```

更详细地，请查 MSDN 的关键字__stdcall，__cdecl，WINAP，CALLBACK，extern “C”，__declspec 等。

2）关于 C 语言生成的汇编码

1．函数参数压栈时所占的字节数

- char,int 占 2 字节；
- long，float 占 4 字节；
- double 占 8 字节；
- 近指针占 2 字节（只有偏移）；
- 远指针占 4 字节（段地址和偏移）。

2．函数的返回值

若函数返回值为 char、int 或近指针，则返回值须放于 AX 寄存器中；若函数返回值为 long 或远指针，则返回值的低字部分（或远指针的偏移部分）须放于 AX 寄存器中，高字部分（或远指针的段地址部分）须放于 DX 寄存器中。

3．局部变量

最开始的两个局部变量会自动使用 SI 或 DI 寄存器存储，其余局部变量均存放在栈中。

4．全局变量

全局变量和静态变量均存放于数据段（也叫堆,heap）中，而不是栈中。

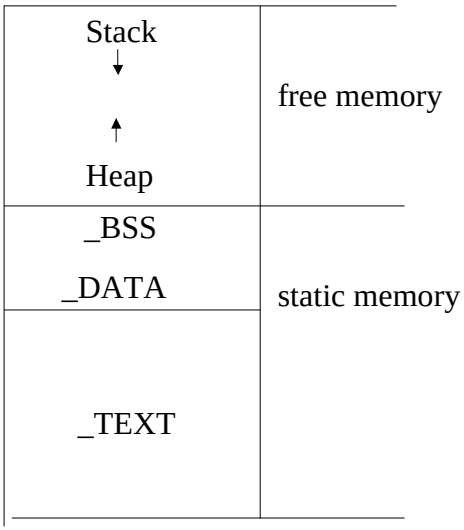
5．生成的段名

对只生成一个代码段的编译内存模型（tiny,small,compact）代码段的段名为**_TEXT**，其它内存模型（medium,large,huge）生成的代码段段名为**“模块名_TEXT”**。

数据段分**_DATA**段（存放已初始化的全局变量）和**_BSS**段（存放未初始化的全局变量）两种，这两种数据段最后会被用 group 伪指令强行合成一个段组，段组名为**DGROUP**。将所有未初始化的全局变量专门放在一个段中的目的是为了使生成的 EXE 长度短（BSS 段不占 EXE 长度，BSS 段只说明了要用到的内存大小，在程序加载时再为 BSS 段分配空间）。

BSS 意为 block starded by symbol，用于保存未初始化的符号。bss 是可执行文件中的一段。所谓的未初始化是说，此段仅包含了变量的符号和相关信息，由于变量具体的值还不知道，所以未给这些变量分配空间。在程序被加载时，bss 段被扩充(为变量分配空间)并映射到内存中的"未初始化数据区"，而实际上相应的值被统一初始化为 0（在高级语言的编译器中是这样做的）。注：如果 EXE 文件是被固化在 ROM 中，则 EXE 文件的长度就很重要了，仅当程序开始运行时，才需为 BSS 中的数据分配空间。BSS 节对应数据段里未初始化的数据部分。我们不想让未初始化的数据占用文件空间，但是进程映象必须保证能够分配足够的内存空间。

BSS 节位于数据段尾部，任何超过文件尺寸的定位都假设位于该节中。



_text: 代码段 _data 已初化的全局变量和静态变量 _bss 未初化的全局变量
Heap: malloc() / new动态内存分配区 Stack: 函数返回地址/局部变量等

C语言生成 程序的内存映像

6．声明被外部调用的过程

凡是本模块中定义的全局变量和函数都须用 public 伪指令声明，以便连接时，其它模块能使用这些全局变量和函数。凡是在本模块中使用的在其它模块中定义的全局变量和函数，均必须在本模块中用 extrn(extern)伪指令说明其类型。

7．可以自由使用的寄存器

在子程序中，可自由使用 BX、CX、ES 寄存器。若使用其它寄存器，必须进行保护。

8．编译内存模型问题

tiny,small 内存模型生成的代码和数据指针均为近指针；
compact 模型生成的代码为近指针，数据为远指针；
medium 模型生成的数据为近指针，代码为远指针；
large,huge 模型生成的代码和数据指针均为远指针。
生成的子程序的结构(Standard Stack Frame)
_subproc proc near (假设选择的是近代码)

```
        push bp
        mov bp,sp
        …
        …
        mov sp,bp
        pop bp
        ret
_subproc endp
```

怎样使 c 语言生成汇编码，以便进行分析

以 Turbo C 为例：

```
C>tcc -S  -ms  - v test.c
```

将产生对应的 test.asm。

其中，-S(要大写!)表示要生成汇编码，-ms 表示以 small 内存模型编译（其它模型：-mt,-mm,-mc,-ml,mh），-v 表示要生成调试信息。

怎样方便地分析 c 语言生成的汇编码

可使用 Turbo Debugger 进行 c 和汇编码的对照分析。

```
C>TD  test.exe
```

注：若想在 Visual C++ 6.0 中观察生成的汇编代码片断，可按 F10 或 F11 键，跟踪 c 语言程序，右击鼠标，选 Go to Disassembly，则出现汇编窗口（选中 byte code 还会同时显示机器码），可按 F10 单步跟踪执行。其中使用的都是 32 位寄存器，如 eax,esp,ebp 等，显示的是 32 位线性地址。char，int，long 等数据类型都分配 32 位内存空间。注意，char 类型数组，每个 char 仍占 1 字节。局部数组在栈中分配空间时，如 char s[20]，s[0]位于 20 字节栈空间的低端，s[19] 位于 20 字节栈空间的高端。跟踪时观察各寄存器值的方法与观察变量值的方法相同，即在右下角 Watch 窗口中的 Name 栏中输入“寄存器名[，显示格式]”，例如：

```
esp,0x
ebp
eax
s[0],02x
```

以下是 TC 2.0 生成的汇编码举例：

```
/*TEST.C*/
int sum;
main()
{
        sum=add(10,20);
}
int add(int a,int b)
{
        int t;
        t=a+b;
        return t;
}
```

进入_add 后，执行完 mov bp,sp 后的栈顶框架如下：

```
bp+8->20  (b)
bp+6->10  (a)
bp+4->ip  (返回的偏移)
bp+2->si  (push si)
bp+0->bp  (push bp,  mov bp,sp)
```

```
/*生成的 TEST.ASM 片段*/
_main proc near
       mov ax,20
       push ax
       mov ax,10
       push ax
       call near ptr _add
       pop cx
       pop cx
       mov word ptr dgroup:_sum,ax
       ret
_main endp
_add proc near
       push si
       push bp
       mov bp,sp
       mov si,word ptr [bp+6]
       add si,word ptr [bp+8]
       mov ax,si
       mov sp,bp
       pop bp
       pop si
       ret
_add endp
       …
       public _sum
       public _add
       public _main
       …
```

更全面的例子见附件 16。

3）C 语言调用汇编子程序的例子

```
/*C 语言程序 TEST.C*/:
extern int add(int,int);
int sum;
main()
{
       sum=add(2,3);
       printf(“sum=%d”,sum);
}


;;;;;汇编程序 TESTADD.ASM:
.model small
.code
public _add
_add proc near
        push bp
        mov bp,sp
        mov ax,word ptr [bp+4]
```

```
        add ax,word ptr [bp+6]
        pop bp
        ret
_add endp
    end
```

本汇编程序比自动生成的_add 进行了优化，执行完 mov bp,sp 后的栈顶框架如下：
bp+6->3
bp+4->2
bp+2->ip （返回的偏移）
bp+0->bp （push bp, mov bp,sp）

以上程序编译连接的步骤如下：
C>tcc -c -ms test.c
C>tasm testadd.asm
C>tlink lib\c0s.obj+test.obj+testadd.obj+lib\cs.lib,test.exe（“+”也可以使用空格）
生成 test.exe
C>test
显示 sum=5

5) Win32 下 C 和汇编的混合编程

混合编程的关键是两种语言的接口问题，解决方法：

A. 在 C 程序中直接嵌套汇编代码；

B. 由 C 主程序调用汇编子程序；（使用更广，功能更强）

当然汇编程序也可以调用 C 函数。

- **C 语言中直接嵌入汇编代码**

关键字__asm,为了向上兼容，也支持_asm。

格式：

```
__asm{
汇编语句
汇编语句
……
}
```

内嵌汇编中有一些限制，如不能使用 byte,word,Dword 等语句定义数据，操作码必须使用有效的 80x86 指令。内嵌汇编语句的操作数是：

A 寄存器；

B 局部变量、全局变量和函数参数；

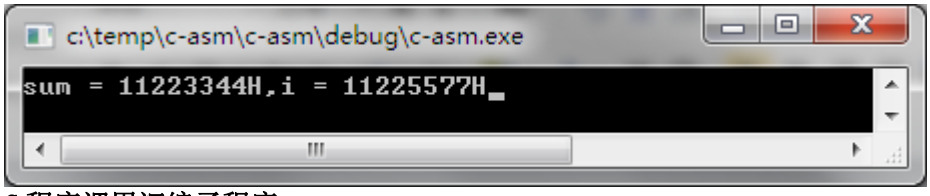
C 结构成员；（可以直接用“结构名.成员名”，如果结构的地址已存入寄存器中，可用“[寄存器].成员名”。）

例子：

```
#include<stdio.h>
int main()
{
    int sum = 0x11223344,i = 0;
    _asm{
          MOV EAX,sum          //EAX=11223344H
          MOV AX,5566h         //EAX=11225566H
          MOV AL,77h           //EAX=11225577H
          MOV i,EAX            //i=EAX
```

```
    }
    printf("sum = %xH,i = %xH",sum,i);
    return 0;
}
```

运行结果：



➢ **C 程序调用汇编子程序**

该种方法使用最广，功能最强，C 和汇编源程序各自分别编写，遵循各自的书写规则。C 模块可调用汇编模块的子程序和全局变量，反过来汇编模块可调用 C 模块的函数和全局变量。

a) C 模块使用汇编模块中的变量

C 模块中的 char、int 等数据类型和汇编模块中的数据类型的相互对应关系：

C 变量类型	汇编变量类型	大 小
char	sbyte	1B
short	sword	2B
int	sdword	4B
long	sdword	4B
unsigned char	byte	1B
unsigned short	word	2B
unsigned int	dword	4B
unsigned long	dword	4B
指针	dword	4B

如果 C 模块需要调用汇编模块中的全局变量，则在汇编模块中变量名前需加下划线开头。例如：

汇编模块中：

```
_strTitle sbyte “Tongji University”
_xVal sdword 100
```

用 public 声明允许外部模块访问该变量：

```
public _strTitle
public _xVal
```

C 模块中用 extern 声明变量来自外部模块：

```
extern char strTitle[];
extern int xVal;
```

之后，在 C 模块中就可以使用这些变量了，如

```
printf(“%s, x=%d\n”,strTitle,xVal);
```

注意：在 C 模块中变量前必须去掉下划线，因为在编译时，会自动给这些变量名前加下划线，这样在链接的时候才会一致。

b) 汇编模块使用 C 模块中的变量

在这里只要注意 C 模块中的变量在编译时会自动加一个下划线就 Ok 了。

C 模块中用 public 声明变量可被外部模块调用：

```
public int a，b，c;
```

汇编模块中要使用 C 模块中声明的变量，应使用 ENTRN 加以说明：

```
ENTRN _a:sdword, _b:sdword, _c:sdword
```

之后就可以在汇编语言中使用这些变量了，如

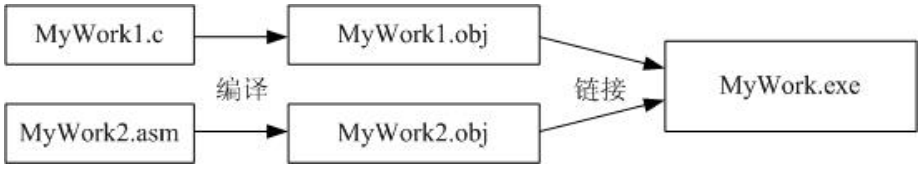
```
MOV EAX,_a
```

c) **C 模块调用汇编模块中的子程序**

```
汇编模块中子程序格式：
getNum        PROC   C       x:sdword
              MOV eax,x      ;参数 x 在堆栈中
              ……
              MOV eax,1      ;返回值
              RET
getNum        ENDP
```

```
C 模块中用 extern 声明函数：
extern int getNum(int x); //由于子程序的返回值为 eax，所以返
                          //回类型为 int
之后就可以在 C 模块中调用汇编子过程了。
int num = getNum(x);
```

d) **编译链接过程**



具体步骤如下：

C>cl /c MyWork1.c

C>ml /c /coff MyWork2.asm ;生成 coff 格式的 obj 文件，并不直接生成 exe 文件；

C>link MyWork1.obj MyWork2.obj /out:MyWork.exe /subsystem:console;表明程序为 Windows 下的控制台程序；

➢ **汇编模块调用 C 函数**

```
在汇编模块使用 PROTO 声明 C 函数的名称、调用方式、参数类型等，如：
function1     proto C px:ptr sdword, py:ptr sdword, pz:ptr sdword
function2     proto C x:dword, y:dword, z:dword
```

```
C 模块中用 extern 声明函数：
extern void function1(int *px,int *py,int *pz);
extern int function2(int x,int y,int z);
```

5）相关问题

1. 关于 END 问题

由于在 C 语言初始化模块 c0s.obj 中已指明程序的入口地址是 main（），所以在汇编程序中，程序结束的 END 伪指令后不能再指定程序的入口地址，否则连接时会出错。

2. 怎样连接 c 语言和汇编语言生成的 OBJ 模块

a) 编译&汇编

✓ 对 c 语言进行编译（不要进行连接）

C>tcc -ms -c test.c

将生成 test.obj。其中-c 表示 compile only。若要包括调试信息，请加-v 开关。

以上用菜单操作也可。

✓ 对汇编语言进行汇编

C>tasm add.asm
将生成 add.obj。若要包括调试信息，请加-zi 开关。

b) 对 OBJ 进行连接

C>tlink **c0s.obj**+test.obj+add.obj,project.exe,,cs.lib
将生成 project.exe。其中，

- ✓ c0s.obj 是 small 内存模型对应的 c 语言初始化模块，连接时必须作为第一个 OBJ 出现。其它模型对应的初始化模块分别为 c0t.obj, c0c.obj, c0m.obj, c0l.obj, c0h.obj。请查看 LIB 子目录中是否存在该模块文件。
- ✓ cs.lib 是 tiny 和 saml 内存模型对应的 c 语言标准库。其它模型对应的初始化模块分别为 cm.lib, cc.lib, cl.lib, ch.lib。请相看 LIB 子目录中是否存在对应的库文件。如果 c 语言中还用到了非标准库函数，需再加入相应的库（如 maths.lib,grahics.lib,emu.lib 或 fp87.lib 等）。

要注意，若以上的 c0s.obj 和 cs.lib 不在当前目录中（一般在 LIB 目录中），连接时，需指明其所在路径。例如：

C>tlink lib\c0s.obj+test.obj+add.obj+lib\cs.lib+emu.lib,project.exe
其它说明：

tasm 汇编后，默认将所有的符号全转换成大写，可用-mu 开关禁止 tasm 转换成大写。tlink 连接时，默认不区分符号的大小写。若要区分大小写，可使用-c 开关。这是以上我们在 tasm 时未用-mu 开关的原因。

另外，若想在 Turbo Debugger 中能得到汇编源程序与生成指令的完整对照（即使生成的 EXE 中包含 DEBUG 信息），必须在 TASM 汇编时带/Zi 开关，在 TLINK 时带/v 开关。

第三十节 AT&T 汇编

到目前为止我们用的都是 Intel 汇编格式，在 Linux 内核代码中，使用的是 AT&T 汇编（Linux 下常使用 gas 汇编器），它的语法风格与 Intel 汇编差异较大，主要如下：

1） 寄存器引用

引用寄存器要在寄存器号前加百分号%,如“movl %eax, %ebx”。

80386 有如下寄存器：

8 个 32-bit 寄存器 %eax，%ebx，%ecx，%edx，%edi，%esi，%ebp，%esp;

8 个 16-bit 寄存器 它们事实上是上面 8 个 32-bit 寄存器的低 16 位：

%ax，%bx，%cx，%dx，%di，%si，%bp，%sp;

8 个 8-bit 寄存器 %ah，%al，%bh，%bl，%ch，%cl，%dh，%dl。

它们事实上是寄存器%ax，%bx，%cx，%dx 的高 8 位和低 8 位;

6 个段寄存器 %cs(code)，%ds(data)，%ss(stack)，%es，%fs，%gs;

3 个控制寄存器 %cr0，%cr2，%cr3;

6 个 debug 寄存器 %db0，%db1，%db2，%db3，%db6，%db7;

2 个测试寄存器 %tr6，%tr7;

8 个浮点寄存器

栈 %st(0)，%st(1)，%st(2)，%st(3)，%st(4)，%st(5)，%st(6)，%st(7)。

2） 操作数顺序

操作数排列是从源（左）到目的（右），如“movl %eax(源)，%ebx(目的)”

3） 立即数

使用立即数，要在数前面加符号$

如“movl $0x04, %ebx”

或者：

```
para = 0x04
movl $para, %ebx
```

指令执行的结果是将立即数 04h 装入寄存器 ebx。

4） 符号常数

符号常数直接引用 如

```
value: .long 0x12a3f2de
movl value , %ebx
```

指令执行的结果是将常数 0x12a3f2de 装入寄存器 ebx。

引用符号地址在符号前加符号$, 如“movl $value, % ebx”则是将符号 value 的地址装入寄存器 ebx。

5） 操作数的长度

操作数的长度用加在指令后的符号表示 b(byte, 8-bit), w(word, 16-bits), l(long, 32-bits)，如“movb %al, %bl”，“movw %ax, %bx”，“movl %eax, %ebx ”。

如果没有指定操作数长度的话，编译器将按照目标操作数的长度来设置。比如指令“mov %ax, %bx”，由于目标操作数 bx 的长度为 word，那么编译器将把此指令等同于“movw %ax, %bx”。同样道理，指令“mov $4, %ebx”等同于指令“movl $4, %ebx”，“push %al”等同于“pushb %al”。对于没有指定操作数长度，但编译器又无法猜测的指令，编译器将会报错，比如指令“push $4”。

6） 符号扩展和零扩展指令

绝大多数面向 80386 的 AT&T 汇编指令与 Intel 格式的汇编指令都是相同的，符号扩展指令和零扩展指令则是仅有的不同格式指令。符号扩展指令和零扩展指令需要指定源操作数长度和目的操作数长度，即使在某些指令中这些操作数是隐含的。

在 AT&T 语法中，符号扩展和零扩展指令的格式为，基本部分"movs"和"movz"（对应 Intel 语法的 movsx 和 movzx），后面跟上源操作数长度和目的操作数长度。movsbl 意味着 movs （from）byte （to）long; movbw 意味着 movs （from）byte （to）word; movswl 意味着 movs （from）word （to）long。对于 movz 指令也一样。比如指令“movsbl %al, %edx”意味着将 al 寄存器的内容进行符号扩展后放置到 edx 寄存器中。

其它的 Intel 格式的符号扩展指令还有：

```
cbw -- sign-extend byte in %al to word in %ax;
cwde -- sign-extend word in %ax to long in %eax;
cwd -- sign-extend word in %ax to long in %dx:%ax;
cdq -- sign-extend dword in %eax to quad in %edx:%eax;
```

对应的 AT&T 语法的指令为 cbtw，cwtl，cwtd，cltd。

7） 调用和跳转指令

段内调用和跳转指令为"call"，"ret"和"jmp"，段间调用和跳转指令为:"lcall"，"lret"和"ljmp"。

段间调用和跳转指令的格式为“lcall/ljmp $SECTION, $OFFSET”，而段间返回指令则为 “lret $STACK-ADJUST”。

8） 前缀

操作码前缀被用在下列的情况：

字符串重复操作指令(rep,repne);
指定被操作的段(cs,ds,ss,es,fs,gs);
进行总线加锁(lock);
指定地址和操作的大小(data16,addr16);

在 AT&T 汇编语法中，操作码前缀通常被单独放在一行，后面不跟任何操作数。例如，对于重复 scas 指令，其写法为：

```
repne
scas
```

上述操作码前缀的意义和用法如下：

指定被操作的段前缀为 cs,ds,ss,es,fs,和 gs。在 AT&T 语法中，只需要按照 section:memory-operand 的格式就指定了相应的段前缀。比如：lcall %:realmode_swtch 操作数 / 地址大小前缀是“data16”和"addr16"，它们被用来在 32-bit 操作数 / 地址代码中指定 16-bit 的操作数 / 地址。

总线加锁前缀“lock”，它是为了在多处理器环境中，保证在当前指令执行期间禁止一切中断。这个前缀仅仅对 ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG,DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD,XCHG 指令有效，如果将 Lock 前缀用在其它指令之前，将会引起异常。

字符串重复操作前缀"rep","repe","repne"用来让字符串操作重复“%ecx”次。

9） 内存引用

Intel 语法的间接内存引用的格式为：

section:[base+index*scale+displacement]

而在 AT&T 语法中对应的形式为：

section:displacement(base,index,scale)

其中，base 和 index 是任意的 32-bit base 和 index 寄存器。scale 可以取值 1，2，4，8。如果不

指定 scale 值，则默认值为 1。section 可以指定任意的段寄存器作为段前缀，默认的段寄存器在

不同的情况下不一样。如果你在指令中指定了默认的段前缀，则编译器在目标代码中不会产生此

段前缀代码。

如果 call 和 jump 操作在操作数前指定前缀“*”，则表示是一个绝对地址调用/跳转，也就是说 jmp/call 指令指定的是一个绝对地址。如果没有指定"*"，则操作数是一个相对地址。

任何指令如果其操作数是一个内存操作，则指令必须指定它的操作尺寸(byte,word,long），也就是说必须带有指令后缀(b,w,l)。

第三十一节 汇编多模块程序实现

1） 例子：文件 module1.asm

```
TITLE MODULE 1
DATA SEGMENT
STRING DB 'CHARACTER DISPLAY $'
DATA ENDS

STACK1 SEGMENT PARA STACK
       DW 20H DUP(0)
STACK1 ENDS
PUBLIC STRING
EXTRN DISPLAY:FAR
CODEG segment
ASSUME CS:CODEG,SS:STACK
START:        MOV AX,DATA
              MOV DS,AX
              CALL DISPLAY

              MOV AX,4C00H
              INT 21h
CODEG ENDS
    END  START
```

文件 module2.asm

```
TITLE MODULE 2
EXTRN STRING:BYTE
PUBLIC DISPLAY

CODEG1 segment
DISPLAY   PROC FAR
              MOV DX,OFFSET STRING
```

```
                MOV AH,9
                INT 21h
                RET
DISPLAY    ENDP

CODEG1 ENDS
    END
```

注意两个文件间符号调用关系。
步骤一：分别对两个源文件进行汇编生成对应目标文件
如：masm ***.asm; 生成 module1.obj 和 module2.obj;
步骤二：链接生成可执行文件，注意 link 是用空格分割。
如：link module1.obj module2.obj; （默认生成与第一个 obj 同名的 exe 文件）

2） *第四部分 参考资料(网上 download 资料)*

附件 0. Debug 资料大全

Debug－PC 之开山老祖

Debug 原意是杀虫子。这里是机器调试工具。

其实，Debug 的由来，还有一则趣闻，在早期美国的一计算机房中，科学家正在紧张的工作。同时，许多台大型的计算机也在不停的运行着。大概是由于机器过热，引来许多的小虫子，以至于计算机无法正常运行。科学家们只好停下来捉虫子，虫子捉完了，计算机运行也正常了，后来，这个名词就沿用至今了。

虽然，现在好的软件越来越多了，但是有些，我们只需动手，用 Debug 就可解决，且快而好！接下来，我们就一起学习 Debug 在各方面的运用吧！（在杀毒、加解密、系统...）下面我和大家一起学习一些 Debug 的用法！

DEBUG 是为汇编语言设计的一种高度工具，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。

一、 DEBUG 程序的调用

在 DOS 的提示符下，可键入命令：

C:\DEBUG [D:][PATH][FILENAME[.EXE][PARM1][PARM2]

其中，文件名是被调试文件的名字。如用户键入文件，该文件必须为可执行文件（EXE），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。

二、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 80H 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,8d 和各占一个字节，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 DS：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。

3、填写命令 F(FILL)，其格式为：

-F range list

例如：-F 4BA:0100 104 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。

4、检查和修改寄存器内容的命令 R(register)，它有三种格式如下：

1）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 C70604023801 MOV WORD PTR [0204],0138 DS:0204=0000

2）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。

3）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。

5、运行命令 G，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。

6、跟踪命令 T(Trace)，有两种格式：

1）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。

2）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。

7、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。

8、反汇编命令 U(Unassemble)有两种格式。
1）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 C70606020202 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。
2）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 C70606020202 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 C70606020202 MOV WORD PTR[0208],0202
```
可见这两种格式是等效的。

9、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。

10、装入命令 L(Load)，有两种功能。
1）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]
2）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。

11、写命令 W(Write)，有两种功能。
1）把数据写入磁盘的指定扇区。其格式为：
-W address drive sector sector
2）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。

12、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。

－－－－－－－－－－－－－－－－－－－－
问题:初学者问一个低级问题,执行 debug-a 后,如果有一行输入错误,如何更改这一行?
回答:
加入进行如下输入：
D:\PWIN95\Desktop>debug
-a
2129:0100 movax,200
2129:0103 movbx,200
2129:0106 movcx,200
2129:0109
此时，发现 mov bx,200 一句错误，应为 mov bx,20，可以敲回车返回"-"状态，然后输入：
-a103
212

Debug 实战
1.查看主板的生产日期，版本
D ffff:05
D fe00:0e
2.模拟 Rest 键功能
A
:100 jmp ffff:0000
:105
g
3.快速格式化软盘
L 100 0 0 * 插入一张已格式化软盘
W 100 0 0 * 放入一张欲格式化软盘
注：* 分别为:720K e |1.2M id |1.44M 21
4.硬盘格式化两种方法
(1) G=c800:05
(2) A 100
mov ax,0703

```
mov cx,0001
mov dx,0080
int 13
int 3
g 100
```

5.加速键盘

```
A
mov ax,0305
mov bx,0000
int 16
int 20
rcx
10
n fast.com
w
q
```

6.关闭显示器（恢复时，按任意键）

```
A
mov ax,1201
mov bl,36
int 10
mov ah,0
int 16
mov ax,1200
int 10
rcx
10
n crtof.com
w
q
```

7.硬盘 DOS 引导记录的修复

在软驱中放入一张已格式化软盘

```
debug
-l 100 2 0 1
-w 100 0 50 1
```

把软盘放入故障机软驱中

```
debug
-l 100 0 50 1
-w 100 2 0 1
-q
```

8.清 coms 中 setup 口令

```
debug
-a
mov bx,0038
mov cx,0000
mov ax,bx
out 70,al
inc cx
cmp cx,0006
jnz 0106
int 20
```

```
-rcx
:20
-nclearpassword.com
-w
-q
```

注：以上适合 super 与 dtk 机，对于 ast 机，因为他的口令放在 coms 的 4ch-51h 地址处，只要将：mov bx,0038 改为： mov bx,004c 即可

9.将 coms 数据清为初始化

```
-o 70,10
-o 71,10
-g
-q
```

10.将硬盘主引导记录保存到文件中

```
debug
-a
mov ax,0201
mov bx,0200
mov cx,0001
mov dx,0080
int 13
int 3
-rcx
:200
-nboot.dat
-w
-q
```

11.调用中断实现重启计算机（可以成文件）

```
debug
-a
int 19
int 20
-rcx
:2
-nreset.com
-w
-q
```

表 1 DEBUG 中标志位的符号表示

标志名称	溢出 OF	方向 DF	中断 IF	负号 SF	零 ZF	辅助进位 AF	奇偶 PF	进位 CF
置位状态	OV	DN	EI	NG	ZR	AC	PE	CY
复位状态	NV	UP	DI	PL	NZ	NA	PO	NC

表 2 DEBUG 命令及其含义

命令格式	功能说明
[地址]	输入汇编指令
C [范围] 起始地址	对由“范围”指定的区域与“起始地址”指定的同大小区域进行比较，显示不相同的单元
D [范围]	显示指定范围内的内存单元内容

E	地址 字节值表	用值表中的值替换从“地址”开始的内存单元内容
F	范围 字节值表	用指定的字节值表来填充内存区域
G	[=起始地址] [断点地址]	从起点(或当前地点)开始执行，到终点结束
H	数值 1 数值 2	显示二个十六进制数值之和、差
	端口地址	从端口输入
L	[地址 [驱动器号 扇区 扇区数]]	从磁盘读
M	范围 地址	把“范围”内的字节值传送到从“地址”开始的单元
N	文件标识符 [文件标识符…]	指定文件名，为读/写文件做准备
O	端口地址 字节值	向端口输出
P	[=地址] [指令数]	按执行过程，但不进入子程序调用或软中断
Q		退出 DEBUG，不保存正在调试的文件
R	[寄存器名]	显示和修改寄存器内容
S	范围 字节值表	在内存区域内搜索指定的字节值表。如果找到，显示起始地址，否则，什么也不显示
T	[=地址] [指令数]	跟踪执行，从起点(或当前地点)执行若干条指令
U	[范围]	反汇编，显示机器码所对应的汇编指令
W	[地址 [驱动器号 扇区 扇区数]]	向磁盘写内容，(BX、CX)为写入字节数

附件 1. CMOS 参数结构

CMOS 实际上存放的是计算机的系统时钟和硬件配置方面的一些信息，供系统引导时读取；同时初始化计算机各个部件的状态，总共有 128 个字节，存放在 RAM 芯片中。具体内容如下(以 Award CMOS 为例)：

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00000000H	30	00	FF	00	39	00	FF	00	12	00	FF	00	01	00	18	00
	秒		秒报警		分		分报警		小时		时报警		星期		日	
00000010H	11	00	98	00	26	00	02	00	70	00	80	00	00	00	00	00
	月		年		寄存器 A		寄存器 B		寄存器 C		寄存器 D		诊断		下电	
00000020H	40	00	7E	00	F0	00	03	00	0F	00	80	00	02	00	00	00
	软驱		密码域		硬盘				设备		基本内存				扩充	
00000030H	7C	00	2E	00	00	00	7F	00	15	00	86	00	00	00	00	00
	内存		硬盘类型						密码数据位							
	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00000040H	00	00	00	00	00	00	00	00	00	00	00	00	E2	00	22	00
	未知															
00000050H	0F	00	FF	00	FF	00	E1	00	22	00	3F	00	08	00	59	00
	未知															
00000060H	00	00	7C	00	19	00	80	00	FF	00	FF	00	FF	00	FF	00
					世纪值>		未知									
00000070H	7D	00	81	00	AA	00	0F	00	39	00	9B	00	E8	00	19	00
	未知															

上述的内容参考了其他资料，所以不一定完全正确。在 38H-3BH 这四个字节中，由于 39H 和 3BH 这两个字节一直为 00H，所以就略过，那么 CMOS 密码的关键就集中到了 38H 和 3AH 这两个字节上。先介绍一点 Award 的密码规则，Award 允许一位至八位密码，每一个字符的范围由 20H-7FH，也就是由空格到 ASCII 码的 127 号。想必大家已经发现了，八个字符要放到两个字节中去，好象不压缩一下是不行的。的确，Award 是将其压缩了，但是不是普通的压缩方法，我想 Award 另有将其加密的想法，因为在 CMOS 中空位还很多，要想放八个字节看来是没有什么问题的，不过这么裸露的密码就更加没有什么用处了。通常的压缩方式有无损压缩，如 zip,arj 等，或者是有损压缩，象 mpeg,jpeg 等。但是对这么几个字节，这些方法就没有什么用武之地了，而且压缩过的东西，应该是可以还原的，否则压来压去就没有什么意义了。不过 Award 的方法就不同了，他不仅仅进行了超级的有损压缩用的是 HASH 算法，而且这种压缩是不可还原的，下面就给出他的加密压缩方法（以下数值，运算均基于 16 进制）：假如有一密码，八位，记为：ABCDEFGH（每一位的取值范围为 20H-7FH），将其按下列公式运算：

H+4*G+10*F+40*E+100*D+400*C+1000*B+4000*A

将结果按由低到高保存到：H1,H2,H3,字节中，然后将 H2 保存到地址：3AH 中，将 H1 和 H3 的和保存到 38H 中。如果密码不足八位，以此类推。

下面举一实例：我的密码为：r*vte，ASCII 码为：72H、2AH、76H、74H、65H，按公式运算得：72*100 + 2A*40 + 76*10 + 74*4 + 65=8615，于是 H1=00H，H2=86H，H3=15H，所以 3AH 的值

为 86H，38H 的值为 15H。看来密码就这么简单，在你每次输入密码的时候，BIOS 将其算算，再与 CMOS 中的值比较一下，如果一样就放行，否则免谈。

在 CMOS SETUP 中的“BIOS FEATURES SETUP”选项中有“SECURITY OPTION”一栏。若把它设为“SETUP”状态，则只有在进入 CMOS SETUP 时才询问密码。若把它设为“SYSTEM”(或 ALWAYS)状态，则无论正常引导或进入 CMOS SETUP 都要询问密码。

当接通电源时，首先被执行的是 BIOS 中的加电自检程序 POST 对整个系统进行检测，包括对 CMOS RAM 中的配置信息作累加和测试。该累加和与原来的存贮结果进行比较，当两者相吻合时 CMOS RAM 中的配置有效，自检继续进行；当两者不相等时，系统报告错误，要求重新配置，并自动取 BIOS 的默认值设置，原有口令被忽略，此时可进入 BIOS SETUP 界面。

因此，当口令保护被设为 Setup 级时可以利用这一点往 CMOS RAM 中的任一单元写入一个数，破坏 CMOS 的累加测试值，即可达到清除口令的目的。

CMOS 在 DOS 系统中的访问端口为：地址端口 70h，数据端口 71h。

附件 2. BIOS 数据区说明：(V1.0)

本网页内容，对于普通 DIY 来说是没有意义的；但对于某些“高手”可是很重要的。本页只是让大家了解一下 BIOS 各数据区的内容说明。

段地址：00H

偏移	类型	内　　容
0000H	256 双字	中断向量表。
0300H	256 字节	在自检和引导时作为缓冲区使用。
400H	字	计算机上 0 号 RS232-1 适配器的基地址，通常为 3F8H。
402H	字	计算机上 1 号 RS232-1 适配器的基地址，通常为 2F8H。
404H	字	计算机上 2 号 RS232-1 适配器的基地址。
406H	字	计算机上 3 号 RS232-1 适配器的基地址。
408H	字	计算机上 0 号并行打印机适配器的基地址，通常为 378H。
40AH	字	计算机上 1 号并行打印机适配器的基地址。
40CH	字	计算机上 2 号并行打印机适配器的基地址。
40EH	字	计算机上 3 号并行打印机适配器的基地址。(PS2 型此值为扩展 BIOS 数据区段地址)
410H	字	该字保存与计算机连接的设备编码表，BIOS 中断 11H(设备测定)可返回此信息。
	位	
	0	软驱安装标志，此位为 0 表示没有软驱。
	1	数字协处理器安装标志，此位为 0 表示未安装协处理器。
	3-2	系统板 RAM 的大小，适用于一些旧机型，PS2 型未使用。00=16K，01=32K，10=48K，11=64K)。
	5-4	初始显示方式(00=AG，01=CGA-40，10=CGA-80，11=MDA-80)。
	7-6	软驱的数量，公当位 0 为 1 时有效，00=1，01=2，10=3，11=4
	8	DMA 标志
	9-11	所连 RS232 适配器数
	12	连有游戏 I/O
	13	不用(PS2 型为内置 MODEM 安装标志，此位为 0 表示没有安装)
	14-15	所连打印机适配器数
412H	字节	初始测试标志(红外线键盘连接错误单元/？)。
413H	字	该字给出打印机可用 RAM 的容量，基本内存容量为 0-10K，以千字节为单位。BIOS 中断 12H(内存大小测定)可返回此信息。
415H	字	I/O 通道的存储器容量(PS2 型，BIOS 控制标志)。
417H	字节	这是第一个键盘状态字，通过编码，使每位均有特定的含义，具体格式如下：
	位	
	0	表示键盘右边的 Shift 键当前是否被按下(1 表示按下，0 表示未按下)。
	1	表示键盘左边的 Shift 键当前是否被按下(1 表示按下，0 表示未按下)。
	2	表明 Ctrl 键当前是否按下(1 表示按下，0 表示未按下)。
	3	表明 Alt 键当前是否按下(1 表示按下，0 表示未按下)。
	4	屏幕(Scroll)锁定开关键状态(1 表示屏幕锁定处于开，0 表示关)。
	5	数字(Num Lock)锁定开关键状态(1 表示数字锁定处于开，0 表示关)。
	6	大写字母(Caps Lock)开关键状态(1 表示 Caps Lock 处于开，0 表示关)。
	7	插入状态，它表明 Ins 键是否已按下，以使计算机进入“插入”方式，1 表示插入状态正工作，0 表明未动作。
418H	字节	这是第二个键盘状态字，其格式如下：
	位	
	0	表示键盘左边 Ctrl 键当前是否被按下(1 表示按下，0 表示未按下)。
	1	表示键盘左边 Alt 键当前是否被按下(1 表示按下，0 表示未按下)。
	2	如按下 Ctrl+Alt+Del 键，则该位为 1。
	3	如果系统键(Ctrl 和 Num Lock)接下且保持住，则该位为 1，当这个系统键依次按下时，

		BIOS 暂停处理，直至下键按下为止。但它仍响应中断。
	4	表明屏幕(Scrool)锁定键当前是否按下(1 表示按下，0 表示未按下)。
	5	表明数字(Num Lock)锁定键当前是否按下(1 表示按下，0 表示未按下)。
	6	表明大写字母(Caps Lock)锁定键当前是否按下(1 表示按下，0 表示未按下)。
	7	表明 Ins 键当前是否按下(1 表示按下，0 表示未按下)。
419H	字节	为 Alt 和数字键盘键入的数而保留。(按住 ALT+数字，可直接得到相应的 ASCII 码)
41AH	字	指向键盘缓冲区首址
41CH	字	指向键盘缓冲区尾址，当该值等于前一字的值时，说明缓冲区满。
41EH	32 字节	循环键盘缓冲区，它保存键盘键入的字符，直到程序可以接收这些字符为止，前两个字指向此缓冲区的当前是首和尾。
43EH	字节	表示磁盘驱动器的搜索状态，0-3 位分别对应于驱动器。如果这些位中有一位为 0，则表示在搜索磁道之前，必须重新校准相应的驱动器。位 4-6 未使用，位 7 为中断标志位，为 1 表示中断发生。
43FH	字节	表示磁盘驱动器的马达状态，0-3 位分别对应于驱动器 0-3，如果某位被置为 1，则相应驱动器的马达正在转动。位 4-6 未使用，位 7 为 1 表示现行操作是写。
440H	字节	保存一个表明驱动器马达接通多长时间的计数，每个时钟节拍，计数减 1，当计数为 0 明马达停转(根据 INT8 计时)。
441H	字节	表明磁盘工作状态，它被编码，通过使相应位置 1 来表示一个特定的状态，格式如下：
	值	
	00H	正确。
	01H	送给磁盘控制器的是无效命令。
	02H	在盘上未找到地址标记。
	03H	试图在有写保护的盘上写操作。
	04H	所请求扇区未找到。
	08H	驱动器 DMA 错。
	09H	试图使 DMA 对 64KB 存储体进行存取。
	10H	循环冗余校验(CRC)错。
	20H	NEC 磁盘控制器片出现错误。
	40H	无效的查找操作。
	80H	延时，没有响应。
442H	7 字节	从 NEC 磁盘驱动器返回的七个字节状态信息(参见 FDC)。
449H	字节	指明当前视频方式，参见 INT 10H。
44AH	字	指明显示屏幕的当前列数。
44CH	字	指明一个显示页面的字节数，它随时视频方式的不同而变化。80*25 方式=1000H 字节，40*25 方式=800H 字节，图形方式=4000H 字节
44EH	字	指明当前显示页面的地址，即显示在当前显示屏幕的显示页面。
450H	8 字	每个字均表示有关显示页面内当前光标的位置，每个字的第一字节表示列，第二字节表示行(改变这个字节并不能立刻改变显示)。
460H	字节	表明光标的形状，此字节表示光标字符点阵的最下一行的行号，10H 功能调用 1 设置此光标形状(不要直接更改此字节)。
461H	字节	此字节表示光标字符点阵的最上一行的行号。10H 功能调用 1 设置此光标形状(不要直接更改此字节)。
462H	字节	表明工作显示页面号，由 10H 功能调用 5 设置。
463H	字	表明当前工作显示板的口地址。3BCH=单色，3D4H=彩色。
465H	字节	表明 6845 芯片的方式寄存器的当前值(端口：3X8H)。
466H	字节	表示当前显示控制面板的设置。10H 功能调用 0BH 可设置当前面板(端口：3D9H)。
467H	5 字节	PC 中，这 5 个字节用以表示磁带控制的定时计数字、CRC 寄存器字和最后输入数值字节，在 AT 中，这 5 个字节作为端口使用，从 467H 开始的双字长是一个指针，它指向 BIOS 开关使 80X86 由保护虚地址方式转到实地址方式时控制返回的位置。
46CH	双字	这是 BIOS 作为时钟计数器的一个双字单元，时钟第步进一次，此值增加一次，其值为 0，表示一天开始(午夜)，当此计数器达到一天结束的值时，计数器清 0，且字节 470H 置 1。中断 1AH 功能调用 0 可从此双字单元中读取一天的时间。

470H	字节	这是一个时钟翻转字节。当时钟计数器达到一天结束且复位时，此字节置 1 以表明新的一天开始。中断 1AH 功能调用 0 在读取这一天的时间后，将此字节复位。
471H	字节	位 7 为 1 表示 BREAK 键按下(INT 9 设置此标志)。
472H	字	由软件设置复位功能标志或直接跳转 FFFF:0 重启动。
	值	
	1234H	热启动
	5678H	系统中止
	9ABCH	在制造商检测时使用。
474H	字节	硬盘状态。
	值	
	00H	正确
	01H	送给磁盘控制器的是无效命令或参数。
	02H	在盘上未找到地址标记
	03H	试图在有写保护的盘上进行写操作。
	04H	所请求扇区未找到。
	05H	重新复位失败。
	07H	操作失效。
	08H	DMA 错
	09H	试图使 DMA 对 64K 存储体进行存取。
	0AH	坏的扇区标志。
	0BH	坏磁道已清除。
	0DH	扇区号、格式错。
	0EH	控制数据地址已清除。
	0FH	DMA 超出限制。
	10H	循环冗余校验 CRC 错。
	11H	ECC 数据错。
	20H	NEC 磁盘控制器片出现错误。
	40H	无效的查找操作。
	80H	延时，没有响应。
	AAH	没准备好。
	BBH	发生错误，定义不正确。
	CCH	写错误。
	E0H	寄存器错误。
	FFH	磁盘检测失败。
475H	字节	硬盘设备数。
476H	字节	磁盘适配器控制。
477H	字节	硬盘适配器端口。
478H	字节	测试打印机 0 的超时值。
479H	字节	测试打印机 1 的超时值。
47AH	字节	测试打印机 2 的超时值。
47BH	字节	测试打印机 3 的超时值(PS2 型除外)。
47CH	字节	测试 0 号 RS232 超时值。
47DH	字节	测试 1 号 RS232 超时值。
47EH	字节	测试 2 号 RS232 超时值。
47FH	字节	测试 3 号 RS232 超时值。
480H	字	指向存放键盘输入字符的循环缓冲区首址。
482H	字	指向存放键盘输入字符的循环缓冲区尾址。
484H	字节	显示字符的列数。其值为显示字符的列数减 1(EGA 以上有效)。
485H	字	每个字符高度(EGA 以上有效)。
487H	字节	显示控制状态(EGA 以上有效)1。

	位	
	0	光标仿真模式状态(1 为开启)。
	1	单色显示系统状态(1 为启用)。
	2	保留。
	3	显示系统空闲状态(1 为空闲)。
	4	保留。
	6-5	显存容量(00=64K，01=128K，10=192K，11=256K)。
	7	显示模式可用状态。
488H	字节	显示控制状态 2(EGA 以上有效)。
	位	
	0	SW1(1=关闭)
	1	SW2(1=关闭)
	2	SW3(1=关闭)
	3	SW4(1=关闭)
	4	?
	5	?
	6	?
	7	?
489H	字节	显示控制状态 3(MCGA 或 VGA 有效)。
	位	
	0	VGA 模式状态
	1	灰度模式状态
	2	单色显示状态
	3	使用默认模式
	4	--
	5	保留
	6	显示状态开关
	7	--
	值	
	位 7 位 4	
	0 0	350 线模式
	0 1	400 线模式
	1 0	200 线模式
	1 1	保留
48AH	字节	显示适配器 DCC 索引。
48BH	字节	最后磁盘数据率。
	位	
	3-0	保留。
	5-4	步进时间。
	7-6	数据传输率。
48CH	字节	硬盘状态。
48DH	字节	硬盘错误。
48EH	字节	硬盘中断标志。
48FH	字节	位 0 为 1，表示硬盘和软盘使用一个控制卡。
490H	字节	驱动器 0 介质状态。
491H	字节	驱动器 1 介质状态。
492H	字节	驱动器 0 的起始状态。
493H	字节	驱动器 2 的起始状态。
494H	字节	驱动器 0 磁道数。
495H	字节	驱动器 1 磁道数。

496H	字节	键盘类型和方式，各位含义为：
	位	
	0	E1H 隐含码最后。
	1	E0H 隐含码最后。
	2	右 Ctrl 键按下。
	3	右 Alt 键按下。
	4	101/102 键盘
	5	若读标识和键盘，则强置 Num Lock。
	6	最后的字符是第一个 ID 字符。
	7	读键盘的 ID。
497H	字节	键盘标志。
	位	
	0-2	LED 状态位。
	3	保留。
	4	收到消息。
	5	重发接收标志。
	6	方式指示器更新。
	7	键盘传送错误标志。
498H	双字	等待完成标志的偏移地址。
49AH	双字	用户等待计数(低位字)，以微秒为单位。
49EH	字	用户等待计数(高位字)，以微秒为单位。
4A0H	字节	RTC 等待激活标志。80 表示等待时间已过。
4A1H	7 字节	这 7 个字节用于局域网。
4A8H	双字	这双字指向保存视频系统的指针表。指针表格式为：
		偏移值 类型 指向 00H DD 视频参数 04H DD 参数保存区 08H DD 字母字符集 0CH DD 图形字符集 10H DD 第二个保存指针表 14H DD 保留 18H DD 保留 第二个指针表格式为： 偏移值 类型 功能或指向 00H DW 这个表的字节 02H DD 组合码表 06H DD 第二个字母字符集 0AH DD 用户调色板表 0EH DD 保留 12H DD 保留 16H DD 保留
4ACH	8 字节	保留。
4B4H	字节	键盘 NMI 控制标志(可变)。
4B5H	双字	键盘中断中标志(可变)。
4B9H	字节	端口 60 单字节队列(可变)。
4BAH	字节	最后的键盘扫描码(可变)。
4BBH	字节	NMI 缓冲头位置(可变)。
4BCH	字节	NMI 缓冲头位置(可变)。
4BDH	16 字节	NMI 扫描码缓冲(可变)。
4CEH	字	日期计数(可变)。
4F0H	16 字节	?

附件 3. PC I/O 地址分配

I/O 地址	功能、用途
0	DMA 信道 0，内存地址寄存器（DMA 控制器 1(8237)）
1	DMA 信道 0, 传输计数寄存器
2	DMA 信道 1，内存地址寄存器
3	DMA 信道 1, 传输计数寄存器
4	DMA 信道 2，内存地址寄存器
5	DMA 信道 2, 传输计数寄存器
6	DMA 信道 3，内存地址寄存器
7	DMA 信道 3, 传输计数寄存器
8	DMA 信道 0-3 的状态寄存器
AH	DMA 信道 0-3 的屏蔽寄存器
BH	DMA 信道 0-3 的方式寄存器
CH	DMA 清除字节指针
DH	DMA 主清除字节
EH	DMA 信道 0-3 的清屏蔽寄存器
FH	DMA 信道 0-3 的写屏蔽寄存器
19H	DMA 起始寄存器
20H-3FH	可编程中断控制器 1(8259)使用
40H	可编程中断计时器(8253)使用，读/写计数器 0
41H	可编程中断计时器寄存器
42H	可编程中断计时器杂项寄存器
43H	可编程中断计时器,控制字寄存器
44H	可编程中断计时器,杂项寄存器（AT）
47H	可编程中断计时器,计数器 0 的控制字寄存器
48H-5FH	可编程中断计时器使用
60H-61H	键盘输入数据缓冲区
61H	AT:8042 键盘控制寄存器/XT:8255 输出寄存器
62H	8255 输入寄存器
63H	8255 命令方式寄存器
64H	8042 键盘输入缓冲区/8042 状态
65H-6FH	8255/8042 专用
70H	CMOS RAM 地址寄存器
71H	CMOS RAM 数据寄存器
80H	生产测试端口
81H	DMA 信道 2,页表地址寄存器
82H	DMA 信道 3,页表地址寄存器
83H	DMA 信道 1,页表地址寄存器
87H	DMA 信道 0,页表地址寄存器
89H	DMA 信道 6,页表地址寄存器
8AH	DMA 信道 7,页表地址寄存器
8BH	DMA 信道 5,页表地址寄存器
8FH	DMA 信道 4,页表地址寄存器
93H-9FH	DMA 控制器专用
A0H	NM1 屏蔽寄存器/可编程中断控制器 2
A1H	可编程中断控制器 2 屏蔽
C0H	DMA 信道 0，内存地址寄存器（DMA 控制器 2(8237)）
C2H	DMA 信道 0, 传输计数寄存器
C4H	DMA 信道 1，内存地址寄存器
C6H	DMA 信道 1, 传输计数寄存器
C8H	DMA 信道 2，内存地址寄存器
CAH	DMA 信道 2, 传输计数寄存器
CCH	DMA 信道 3，内存地址寄存器
CEH	DMA 信道 3, 传输计数寄存器
D0H	DMA 状态寄存器
D2H	DMA 写请求寄存器
D4H	DMA 屏蔽寄存器
D6H	DMA 方式寄存器

D8H DMA 清除字节指针
DAH DMA 主清
DCH DMA 清屏蔽寄存器
DEH DMA 写屏蔽寄存器
DFH-EFH 保留
F0H-FFH 协处理器使用
100H-16FH 保留
170H 1 号硬盘数据寄存器
171H 1 号硬盘错误寄存器
172H 1 号硬盘数据扇区计数
173H 1 号硬盘扇区数
174H 1 号硬盘柱面（低字节）
175H 1 号硬盘柱面（高字节）
176H 1 号硬盘驱动器/磁头寄存器
177H 1 号硬盘状态寄存器
1F0H 0 号硬盘数据寄存器
1F1H 0 号硬盘错误寄存器
1F2H 0 号硬盘数据扇区计数
1F3H 0 号硬盘扇区数
1F4H 0 号硬盘柱面（低字节）
1F5H 0 号硬盘柱面（高字节）
1F6H 0 号硬盘驱动器/磁头寄存器
1F7H 0 号硬盘状态寄存器
1F9H-1FFH 保留
200H-20FH 游戏控制端口
210H-21FH 扩展单元
278H 3 号并行口，数据端口
279H 3 号并行口，状态端口
27AH 3 号并行口，控制端口
2B0H-2DFH 保留
2E0H EGA/VGA 使用
2E1H GPIP(0 号适配器)
2E2H 数据获取(0 号适配器)
2E3H 数据获取(1 号适配器)
2E4H-2F7H 保留
2F8H 2 号串行口，发送/保持寄存器(RS232 接口卡 2)
2F9H 2 号串行口，中断有效寄存器
2FAH 2 号串行口，中断 ID 寄存器
2FBH 2 号串行口，线控制寄存器
2FCH 2 号串行口，调制解调控制寄存器
2FDH 2 号串行口，线状态寄存器
2FEH 2 号串行口，调制解调状态寄存器
2FFH 保留
300H-31FH 原形卡
320H 硬盘适配器寄存器
322H 硬盘适配器控制/状态寄存器
324H 硬盘适配器提示/中断状态寄存器
325H-347H 保留
348H-357H DCA3278
366H-36FH PC 网络
372H 软盘适配器数据输出/状态寄存器
375H-376H 软盘适配器数据寄存器
377H 软盘适配器数据输入寄存器
378H 2 号并行口，数据端口
379H 2 号并行口，状态端口
37AH 2 号并行口，控制端口
380H-38FH SDLC 及 BSC 通讯
390H-393H Cluster 适配器 0
3A0H-3AFH BSC 通讯

3B0H-3B H	MDA 视频寄存器
3BCH	1 号并行口，数据端口
3BDH	1 号并行口，状态端口
3BEH	1 号并行口，控制端口
3C0H-3CFH	EGA/VGA 视频寄存器
3D0H-3D7H	CGA 视频寄存器
3F0H-3F7H	软盘控制器寄存器
3F8H	1 号串行口，发送/保持寄存器(RS232 接口卡 1)
3F9H	1 号串行口，中断有效寄存器
3FAH	1 号串行口，中断 ID 寄存器
3FBH	1 号串行口，线控制寄存器
3FCH	1 号串行口，调制解调控制寄存器
3FDH	1 号串行口，线状态寄存器
3FEH	1 号串行口，调制解调状态寄存器
3FFH	保留

附件 4. 全面了解 IRQ

对于大多数 DIY 高手来说，电脑的 IRQ 设置是“轻而易举”的事情。但对于一些初涉此道的“菜鸟”朋友，如何熟悉、掌握它们，还要经过一个了解→熟悉→掌握的过程。我们这篇文章就是希望能够帮助您认识 IRQ 并且熟练地掌握它们。

什么是 IRQ

IRQ 全称为 Interrupt Request，即“中断请求”的意思（以下使用 IRQ 称呼）。IRQ 的作用就是在我们所用的电脑中，执行硬件中断请求的动作，用来停止其相关硬件的工作状态。

我们可以举一个日常事例来说明，假如你正在给朋友写 E-mail，突然电话铃响了，那么你就需要放下手中的笔去接电话，通话完毕后再继续写信。这个例子就体现了中断及其处理的过程：电话铃声使你暂时中止当前的工作，而去处理更为急需处理的事情——接电话，当接完电话后，再回过头来继续原来的工作。在这个例子中，电话铃声就可称为“中断请求”，而你暂停写信去接电话的动作就是“中断响应”，接电话的过程则是“中断处理”。

在使用电脑的过程中，每当我们按一下键盘上的按键，就会产生一个键盘中断信号，CPU 就要停下正在处理的工作来处理这个信号，记录下刚才是哪个键被按下了，如果按下的这个键对应于某一个操作，就要优先进行这个操作，然后再处理按键前的工作。如果这时键盘同其它硬件设备的中断有冲突（即键盘和另外一个硬件设备共享一个中断，这种情况好比家中有两部电话放在一起，但其振铃声调却完全一样，这样，必然造成接电话时不知该接哪一部的混乱状态），那么计算机就无法判断刚刚到达的信号是来自键盘的还是来自其它硬件设备的，因此就可能会引发出一些问题。可见“IRQ”在计算机应用中的重要性，因此将中断合理分配，让它们之间没有冲突，是保证电脑稳定运行的关键问题之一。

认识 IRQ

计算机中的中断根据信号产生的来源可分为：硬件中断和软件中断。硬件中断多由外围设备和计算机系统控制器发出，软件中断一般由软件命令产生。在硬件中断中又有“可屏蔽中断”和“不可屏蔽中断”之分。顾名思义，可屏蔽中断可以由计算机根据系统的需要来决定是否进行接收处理或是延后处理（即屏蔽）；而不可屏蔽中断便是直接激活相应的中断处理程序，它不能也不会被延误，我们**常说的 IRQ 中断就是可屏蔽的硬件中断**。

irq 分配表

中断 0 系统计时器
中断 1 键盘
中断 2 可编程中断控制器
中断 3 com2
中断 4 com1
中断 6 软盘控制器
中断 7 并口 1
中断 8 系统 cmos/时钟
中断 12 ps/2 鼠标
中断 13 数学协处理器
中断 14 第一 ide 控制器
中断 15 第二 ide 控制器

在以前的电脑系统中，各硬件的 IRQ 是由一个中断控制器 8259 或是 8259A 的芯片（现在此芯片大都集成到其它的芯片内）来进行控制的。目前共有 16 组 IRQ，去掉其中用来做桥接的一组 IRQ，实际上只有 15 组 IRQ 可供硬件调用。而这些 IRQ 都有自己建议的配置（见 IRQ 分配表）。

从上表中我们可以看到，只有 5、9、10、11 共 4 个中断没有被占用，可以给用户用做新添硬件设备使用，其中显卡要独占一个中断，声卡一般来说也会占用两个中断，它们分别用于 MIDI 接口和 WAVE 的播放。

现在的 Windows 操作系统已经运用 PNP 技术，这种“即插即用”的功能可以将 IRQ 进行自动分配，大大简化了用户的操作。不过这种 PNP 技术也有它的弱点，就是如果操作系统不能正确识别出

要安装的新设备，那幺自动分配 IRQ 时就可能会和其它硬件设备产生冲突。遇到这种情况，只要将新旧两个硬件的 IRQ 配置手动调开就可以解决了。

附件 5. 操作系统实践经验（定时器）

首先我们来了解一下 PC 机的时钟是如何工作的，PC 机采用一块 8253 定时器芯片计算系统时钟的脉冲，若干个系统时钟周期转换成一个脉冲，这些脉冲序列可以用以计时，也可以送入计算机的扬声器产生特定频率的声音。8253 定时器芯片独立于 CPU 运行，它可以象实时时钟那样，CPU 的工作状态对它没有任何影响。
8253 芯片有三个独立的信道，每个信道的功能各不相同，三个信道的功能如下：信道 0 ：为系统时钟所用，在启动时由 BIOS 置入初值，每秒钟约发出 18.2 个脉冲，脉冲的计数值存放在 BIOS 数据区的 0040:006c 存储单元中（注意，这个单元的内容对我们非常有用！），信道 0 的输出脉冲作为申请定时器中断的请求信号，还用于磁盘的某些定时操作，如果改变了信道 0 的计数值，必须确保在 CPU 每次访问磁盘以前恢复原来的读数，否则将使磁盘读写产生错误。信道 1 ：用于控制计算机的动态 RAM 刷新速率，一般情况下不要去改变它。信道 2 ：连接计算机的扬声器，产生单一的方波信号控制扬声器发声。8253 定时器芯片的每一个信道含有 3 个寄存器，CPU 通过访问 3 个端口（信道 0 为 40h，信道 1 为 41h，信道 2 为 42h）来访问各个端口的 3 个寄存器，8253 每个端口有 6 种工作模式，当信道 0 用于定时或信道 2 用于定时或发声时，一般用模式 3 。在模式 3 下，计数值被置入锁存器后立即复制到计数器，计数器在每次系统时钟到来时减 1 ，减至 0 后一方面马上从锁存器中重新读取计数值，另一方面向 CPU 发出一个中断请求（INT 1CH 中断，很有用），如此循环在输出线上高低电平的时间各占计数时间的一半，从而产生方波输出。
对 8253 定时器芯片编程是通过命令端口寄存器 43h 来实现，它决定选用的信道、工作模式、送入锁存器的计数值是一字节还是两字节、是二进制码还是 BCD 码等工作参数，端口 43h 各位的组合形式如下：
位 0 ____若为 1 则采用二进制表示，否则用 BCD 码表示计数值。
位 3－1 ____工作模式号，其值（0-5）对应 6 种模式。
位 5－4 ____操作的类型： 0 0 ：把计数值送入锁存器；
0 1 ：读写高字节；
1 0 ：读写低字节；
1 1 ：先读写高字节，再读写低字节；
位 7－6 ____决定选用的信道号，其值为 0－2 。
对 8253 芯片编程的三个步骤：
1. 设置命令端口 43h，
2. 向端口发送一个工作状态字节，
3. 确定定时器的工作方式；
若是信道 2 ，给端口 61h 的第 0 位和第 1 位置数，启动时钟信号，当第 1 位置 1 时，信道 2 驱动扬声器，置 0 时用于定时操作；将一个字的计数值，按先低字节后高字节的顺序送入信道的 I/O 端口寄存器（信道 0 为 40h，信道 1 为 41h，信道 2 为 42h）。当第 3 个步骤完成后被编程的信道马上在新的状态下开始工作。由于 8253 的三个信道都独立于 CPU 运行，所以在程序结束以前要恢复各信道的正常状态值。
在游戏中我们只使用信道 0 ，怎样对其进行编程呢？首先得确定放入锁存器的 16 位的计数值：
16 位的计数值=1.19318MB/希望的频率
其中 1.19318MB 是系统振荡器的频率。
由上式可知，计数器所能产生的值是 18.2Hz-1.19318MHz，这已经足够了，可以满足我们游戏的要求。
以下是对 8253 定时器芯片信道 0 编程的函数：

```
#define T60HZ 0x4dae
#define T50HZ 0x5d37
#define T40HZ 0x7468
#define T30HZ 0x965c
#define T20HZ 0xe90b
#define T18HZ 0xffff
#define LOW_BYTE(n) (n&0x00ff)
#define HI_BYTE(n) ((n>>8)&0x00ff)
int far *clk=(int far *)0x0000046c;
```

1. 改变时间定时器值的函数：

```
void ChangTime(unsigned cnt)
{
outportb(0x43,0x3c);
outportb(0x40,LOW_BYTE(cnt));
outportb(0x40,HI_BYTE(cnt));
}
```

2. 延时函数：

```
void Delay(int d)
{
```

```
int tm=*clk;
while(*clk-tm<d);
}
```

上面定义的指针*clk 指向 BIOS 数据区的 0040:006c 存储单元，该单元中存放着定时器的计数值，我们可以根据该单元的内容计算差值来达到延时的目的。

需要注意的是，由于改变定时器计数值的操作会与保护模式下的代码发生冲突，所以不能在 WINDOWS 的 DOS 仿真环境下改变定时器，必须在纯 DOS 环境里使用。改变了定时器值以后，程序结束以前必须恢复原来的值，如果不恢复原来的 18．2 次的计数值，在读写磁盘操作时将引起读写错误甚至死机。

我们将时钟设置成每秒 60HZ 这样调用函数：ChangTime(T60HZ)；

在游戏中延时一定时间这样调用函数：Delay(10)；

附件 6. 认识中断请求 IRQ(图)

一、 了解 IRQ 家族

IRQ 全称为 Interrupt Request，即是“中断请求”的意思（以下使用 IRQ 称呼）。IRQ 的作用就是在我们所用的计算机中，执行硬件中断请求的动作，用来停止其相关硬件的工作状态，比如我们在打印一份图片，在打印结束时就需要由系统对打印机提出相应的中断请求，来以此结束这个打印的操作。在每台电脑的系统中，是由一个中断控制器 8259 或是 8259A 的芯片（现在此芯片大都集成到其它的芯片内）来控制系统中每个硬件的中断控制。目前共有 16 组 IRQ，去掉其中用来作桥接的一组 IRQ，实际上只有 15 组 IRQ 可供硬件调用。这 16 组 IRQ 的主要用途如下表：

IRQ编号	设备名称	用途
IRQ0	Time	电脑系统计时器
IRQ1	KeyBoard	键盘
IRQ2	Redirect IRQ9	与IRQ9相接，MPU-401 MDI使用该IRQ
IRQ3	COM2	串口设备
IRQ4	COM1	串口设备
IRQ5	LPT2	建议声卡使用该IRQ
IRQ6	FDD	软驱传输控制用
IRQ7	LPT1	打印机传输控制用
IRQ8	CMOS Alert	即时时钟
IRQ9	Redirect IRQ2	与IRQ2相接；可设定给其它硬件使用
IRQ10	Reversed	建议保留给网卡使用该IRQ
IRQ11	Reversed	建议保留给AGP显卡使用
IRQ12	PS/2Mouse	接PS/2鼠标，若无也可设定给其他硬件使用
IRQ13	FPU	协处理器用，例如FPU（浮点运算器）
IRQ14	Primary IDE	主硬盘传输控制用
IRQ15	Secondary Ide	从硬盘传输控制用

二、 掌握 IRQ 家族的相处之道

现在的 windows 操作系统已经运用 PNP 技术，这种“即插即用”的功能可以将中断进行自动分配，大大简化了用户的操作。不过这种 PNP 技术也有它的弱点，那就是如果不能认出要安装的新设备，那幺自动分配中断时就会产生冲突。我们日常所用的，对于 IRQ 的设置也不尽相同，所以在安装新硬件的时候，系统往往并不能自动检测正确的 IRQ 来分配给所用调用的硬件，这就会造成此硬件设备或是原来的旧硬件出现不能正常工作的现像。现在新的硬件产品层出不穷，各种产品又相互兼容，功能类似，这就导致了操作系统常常不能正确检测出新设备，中断冲突也就不可避免了。其实这是因为系统自动将该硬件的 IRQ 分配给了其它与此 IRQ 相同的硬件上，从而发生冲突使硬件不能正常工作。一般如果遇到这种情况，只要将新旧两个硬件的 IRQ 配置手动调开就可以解决了。手动配置 IRQ 时，最好检查有无保留中断（IRQ），不要让其它设备使用该中断号，以免引起新的中断冲突，造成系统死机。

以下使用目前比较受欢迎的 KT266A 主板 Epox 8KHA+为例，介绍 IRQ 家族的和平相处之道。我从一些外国网站的论坛知道有关 Epox 8KHA+在 Bios 将 Set PnP OS 选项设定为 NO 的时候，IRQ 的自动配置情况：

插槽	INT--A	INT—B	INT—C	INT—D
PCI插槽1	共享			
PCI插槽2				
PCI插槽3			共享	
PCI插槽4				共享
PCI插槽5			共享	
PCI插槽6				共享
AGP插槽	共享			
主板声卡			共享	
USB控制器				共享

从以上默认的 IRQ 自动配置可以得出以下配件最佳安装方法一览表：

配件	安装位置	默认的IRQ自动配置
AGP显卡	AGP插槽	与PCI 插槽 1共享IRQ
PCI显卡	PCI 插槽1	与AGP插槽共享IRQ
IDE RAID 卡	PCI 插槽2	独立使用IRQ（不共享IRQ）
声卡	PCI 插槽 3或插槽 5	与主板上的AC97声卡共享IRQ
网卡、内置猫	PCI 插槽 4或插槽 6	与主板上的USB控制器共享IRQ

只要我们把配件安装正确，BIOS 按照出厂时的设置，这时系统会自动设置 IRQ，使得各个 IRQ 合理分配，使系统工作正常。

三、 解决 IRQ 冲突

常见的 IRQ 冲突现像有系统不能正确检测出新设备、有些硬件工作不正常（如声卡不发声），严重的会出现死机。这往往没有正确安装硬件或手动调整 IRQ 不当引起的。

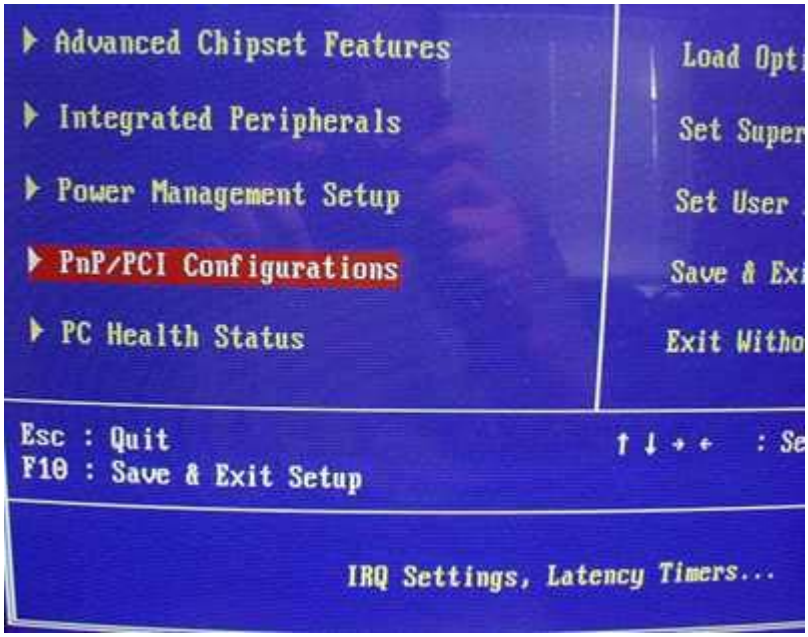
要解决中断冲突，首先我们要知道系统中冲突的设备，做法是在控制面板中双击“系统”图标，查看设备管理器中的各设备。一般有“？”和“！”的设备要注意了，有问题的设备就是它们了。解决方法有分两步做：

第一步、先删去有“？”和“！”的设备，然单击刷新，让计算器自己再认一遍这些设备。这样做是因为部分有“？”和“！”的设备可能是驱动程序安装有误，再重装一遍或升级驱动程序可解决问题。

第二步、如果上面一步还是不能解决问题，现在多半是中断冲突了，那我们只能手动调整来解决中断冲突。在系统＝＞设备管理器＝＞属性 中我们可以看到系统资源分配的情况，通过查看此项就可从中了解到哪些系统资源被占用，哪些系统资源还没有用，用户做相应的调整即可（通常换另外一条插槽再手动配置 IRQ，问题就解决了）。

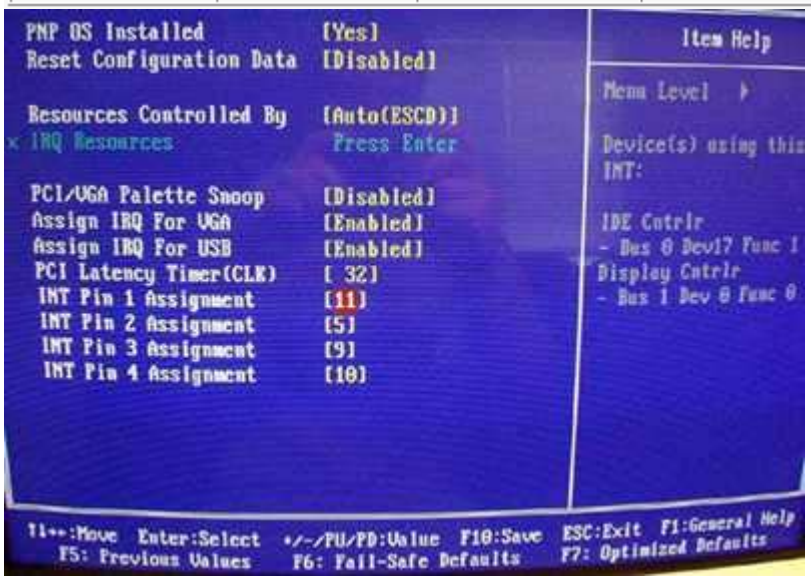
以下使用 Epox 8KHA+ KT266A 主板为例，介绍手动配置硬件 IRQ 时的安装方法和最优的设置方法：

1、开机，进入 CMOS 设置界面，它是 AWARD 公司的 BIOS，进入“PNP/PCI CONFIGURATION” （见图一）



2、将“PNP OS Installed”改为Yes，将“Resources controlled By”改为Auto，利用方向箭头和+，-符号键来设置INT Pin X（x=1,2,3,4）。在菜单左边的INT Pin X的新设置值（红色部分），在右边显示设置的设备。例如INT Pin 1设定为11，对应IRQ为11的设备（显卡）在右边帮助栏显示出来（见图二）。完成对INT Pin X的设置后保存（按F10）后重激活。激活后计算器检测正常，Windows的PNP功能会找到并且安装新硬件。.以下是手动配置硬件IRQ时的安装方法和最优IRQ值的设置一览表：

配件	安装位置	BIOS设置项	设置IRQ值	备注
AGP显卡	AGP插槽	INT-Pin1	11	
PCI显卡	PCI 插槽1	INT-Pin1	11	不能与AGP显卡共同使用
声卡	PCI 插槽 3或插槽 5	INT-Pin2	5	不要忘记把在bios菜单中屏蔽主板自带的声卡
USB		INT-Pin3	9	
网卡、内置modem	PCI 插槽 4或插槽 6	INT-Pin4	10或9	采用IRQ10比采用IRQ9快



四、设置 IRQ 时注意的问题

笔者的计算机在 bios 的设定如下：

PnP OS -> NO
Modem use IRQ -> N/A
Unika GeForce Mx200-> IRQ 11
SB live! Value -> IRQ 5
Star Internal Modem ->IRQ 10
onboard sound ->DISABLED

game port -> DISABLED

midi port -> DISABLED

但事实上在 windows 的系统信息中显示以下配置：

Unika GeForce Mx200 -> 11
SB Live! Value -> IRQ 10
Star Internal Modem -> IRQ 3
USB controller -> IRQ 3
ACPI -> IRQ 9

为什么会出现这种情况呢？原来如果手动配置 INT Pin 4（控制 PCI 插槽 4 和 6，并且控制主板上的 USB）分配断点 9，那么当你安装支持 ACPI 的 Windows 操作系统时， PCI 插槽 4 和 6 和主板上的 USB 控制器的实际断点会被分配其它空闲的中断资源。因为安装时 Windows 默认打开 ACPI 功能，并且会占用 IRQ 9 或 7 或 11 其中一个 IRQ，通常 Windows 操作系统的 ACPI 断点默认是 9。如果关闭 ACPI，你的系统会减少一点发热并可以提高 3 D 的性能。但如果在打开 ACPI 功能时一切运作正常，我建议不要改动这项设置。因为如果关闭 ACPI，就不能使用即插即用功能，这时会出现 Windows 不能探测任何新安装的硬件，也可能会出现多个设备一起使用同一个 IRQ（例如声卡，内置 modem，显卡都使用 IRQ 11）。

一些 VIA KT133A 芯片主板在安装支持 ACPI 的 Windows 2000 或 Win XP 时，会使主板自带的 modem（或内置 modem）的 IRQ 自动设为 9，和 ACPI 功能共享中断。如果你不使用主板自带的 modem 或内置 modem， 那么要将 Modem Use IRQ" 设为 N/A（默认中断请求是 3 ）。你将"Integrated Peripheral \ Super IO Device\Onboard Serial Port 2"设为 DISABLED 并且在 Power Management Setup\Modem Use IRQ 设为 DISABLED 来屏蔽 com1 通信口 （com1 的默认中断请求是 3）， 这样 Epox 8KHA+主板（其它 VIA KT266A 芯片主板也一样）会根据 bios 里的设置而自动合理分配各个中断。这时 PCI 插槽 4 和 6 和主板上的 USB 控制器的实际断点自动分配为 3 。

另外创新的一些声卡如 SBLive 是需要 2 个 IRQ，其中一个对当前的声卡的支持(IRQ 10)，而另一个 IRQ (IRQ 5)用于对那些仍然需要声霸卡（Sound Blaster）兼容模式的老游戏的支持。用户可以在设备管理器中，展开声音视频游戏控制器中看见 Legacy Audio Drivers，双击进入看它是否占用任何 IRQ，如果用户不玩老游戏（如比较旧的 DOS 游戏）你可以在 Legacy Audio Drivers 的属性框中选择禁用该设备。

五、其它技巧

1.删除设备的驱动程序，关机后将外设拔出，置重新安装，让系统重新检测。

2.如果你使用内置调制解调器，可以在 bois 菜单中关闭 com2,这可以节省出 IRQ3,供内置调制解调器使用。

3.屏蔽那些暂时不需要使用的硬件，例如 USB 控制器，节省出 IRQ 以供其它急需使用的重要设备。

附件 7. PC 机高号中断编程 8259 初始化及中断服务程序处理

8259A Initialization and Interrupt Serve Program Processing for Higher IRQ in IBM PC

作者 龚建伟 J.W. Gong

摘 要 本文对 PC 机中高号中断（IRQ8~IRQ15）编程时如何初始化 8259 可编程中断控制器和中断服务程序处理进行了说明，给出了 Turbo C++3.0 编写的 8259 初始化程序和中断服务程序实例。

关键词 8259A 可编程中断控制器； IBM PC 中断编程

在 IBM PC 及其兼容机中，通过 CPU 的 NMI（非屏蔽中断）和两个 8259A 可编程中断控制器芯片为系统提供了 16 级中断，硬件中断结构如图 1 所示 1，两片 8259A 构成主从式级联控制结构，与 CPU 相连的称为主片，下一层的称为从片，从片中断请求信号 INT 与主片的 IRQ2 相连。IBM PC 机中保留给用户可随意编程的中断号有 IRQ10、IRQ11、IRQ12 和 IRQ15，这些中断信号都在 8259A 从片上。8259A 的详细资料请参阅有关手册，本文仅列出大多数 PC 硬件手册中未提及的编程资料，然后说明 PC 机中 8259A 中断控制器的编程初始化过程和如何处理中断服务程序。

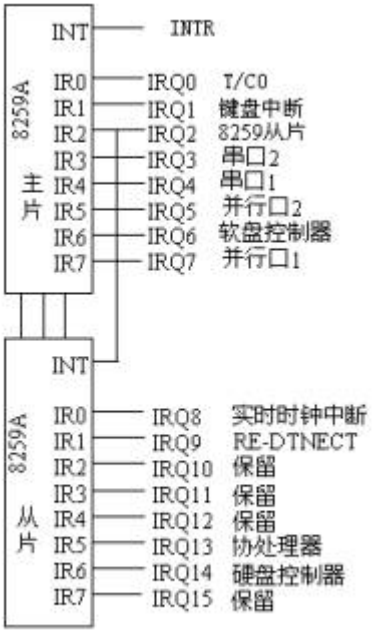


图1 IBM PC/AT机硬中断结构

图 1

IBM PC 机中由 8259A 管理的 16 级中断均有规定的中断向量存储地址，主片中 IRQ0-IRQ7 分别对应 08H-0FH，从片中 IRQ8-IRQ15 分别对应 70H-77H。主片的中断控制寄存器 ICR 和中断屏蔽寄存器 IMR 的口地址分别为 20H 和 21H，从片的相应寄存器口地址分别为 A0H 和 A1H。

中断初始化编程时，当用主片中 IRQ0-IRQ7 时，只须在屏蔽寄存器中打开相应中断，在中断服务程序中，中断结束后，发一次中断结束命令 EOI；而涉及从片中 IRQ7-IRQ15 高号中断时，除在从片中的屏蔽寄存器中打开相对应的中断，还须打开主片中的 IRQ2，且在中断服务程序中中断结束时，要发两次 EOI 命令，分别使主片和从片执行中断结束命令。下面就 IRQ11 的初始化和中断服务程序处理给出 Turbo C 的源代码编程说明。

首先说明一个中断指针 oldvect 以保存原来的中断向量，在中断服务程序 ser_program()结束后，分别向主片和从片的中断控制寄存器 ICR 送中断结束信号 EOI。

```
void interrupt(*oldvect)(...); //设置原中断向量保存指针
void interrupt ser_program(...)
{ {... ...} //中断服务程序代码
outportb(0xA0,0x20); //向从片 ICR 送 EOI 命令
outportb(0x20,0x20); //向主片 ICR 送 EOI 命令
}
```

中断初始化时要先保存 IRQ11 对应的地址 73H 存储的原中断向量，然后将自己的中断服务程序入口地址装入，再分别打开主片 IRQ2 和从片 IRQ11。

```
void Interrupt_Enable(void)
{ int temp;
{... ...} //其它初始化代码
oldvect = getvect(0x73); //保存原中断向量
setvect(0x73,ser_program); //装入中断服务程序入口地址
temp = inportb(0x21) & 0xFB;//打开主片 IRQ2
outportb(0x21, temp);
temp = inportb(0xA1) & 0xF7;//打开从片 IRQ11
outportb(0xA1, temp);
}
```

最后,不要忘记在程序关闭前关中断和恢复原中断向量。

```
void Interrupt_Disable(void)
{ int temp;
setvect(0x73, oldvect); //恢复原中断向量
temp = inportb(0x21) | ~（0xFB）;//关主片 IRQ2
outportb(0x21, temp);
temp = inportb(0xA1) | ~（0xF7）;//关从片 IRQ11
outportb(0xA1, temp);
}
```

以上仅给出了初始化和中断服务程序处理的必要源代码，如感兴趣，可与作者联系索要结合多串口中断收发数据的 Turbo C++源程序(在 Turbo C++3.0 下编译),本人在做 PC104 时用到了多串口。

参考文献

1 Thom Hojam(美). PC 软硬件技术资料大全. 清华大学出版社，1990.

附件 8. 中断的作用及其资源检测

一、 中 断 的 作 用 及 其 资 源 检 测

---- 操 作 系 统 一 般 最 多 允 许 16 种 不 同 的 设 备 向 CPU 提 出 中 断 申 请。 每 一 种 中 断 称 为 硬 件 中 断 或 者 IRQ 级 别。 通 常， 一 个 IRQ 级 别 只 能 配 置 给 一 个 设 备， 但 在 具 有 MicroChannel、 纯 EISA 或 者 PCI 总 线 结 构 的 机 器 中 可 以 共 享 IRQ。 然 而 在 大 多 数 情 况 下， EISA 和 PCI 又 经 常 与 老 式 的 ISA 总 线 插 槽 混 合 使 用， 因 此 目 前 大 多 数 主 板 实 际 上 无 法 实 现 中 断 共 享。 通 常 的 IRQ 设 置 如 表 一 所 示:

表 1 通 常 的 IRQ 设 置

IRQ 级别	用途	说明
0	定时器	主板上的硬布线，不可更改
1	键盘	主板上的硬布线，不可更改
2	级联到第二个中断控制器	是否可用取决于主板设计，最好不用
3	COM2 or COM4	
4	COM1 or COM3	
5	LPT2	第二个并口一般不用,其他设备可以用 IRQ5
6	软盘控制器	
7	LPT1	第一个并口
8	实时时钟	主板上的硬布线，不可更改
9	未用	
10	未用	
11	未用	
12	PS/2, Inport 鼠标	显示卡
13	算术协处理器	用预处理算术协处理其错误
14	硬盘控制器	
15	未用	

---- 计 算 机 中 断 资 源 的 分 配 和 占 用 情 况 的 报 告 在 MS-DOS 下， 可 以 通 过 运 行 MSD 程 序 获 得。 在 Windows 95 下， 可 以 通 过 系 统 诊 断 功 能 方 便 获 得， 当 然 也 可 以 采 用 第 三 方 系 统 测 试 和 诊 断 软 件 如 WinBench97 等。 以 Windows 95 中 文 版 为 例， 具 体 方 法 如 下：

---- 在“ 我 的 电 脑” 上 单 击 鼠 标 右 键， 选 择“ 属 性” 项， 显 示 “ 系 统 属 性” 窗 口， 在“ 设 备 管 理” 标 签 上 单 击 左 键， 出 现 本 机 器 配 置 硬 件 设 备 一 览 表。 在“ 计 算 机” 上 双 机 鼠 标， 进 入 如 下 图 1 所 似 的 窗 口 设 置：

图1 Win95 下的计算机中断资源的配置与管理

---- 不同硬件配置的机器中断分配和占用情况不同，比如某计算机中断占用情况如下：

中断级别	基本用途	使用该中断的硬件
00	定时器	系统时钟
01	键盘	标准 101 键盘或者 Microsoft 自然键盘
02	级联到第二中断控制器	可编程控制器
03	COM2 or COM4	通讯端口 COM4,内置 Modem
03	COM2 or COM4	通讯端口 COM2,主板内置
04	COM1 or COM3	通讯端口 COM1,主板内置
05	LPT2	Creative Labs Sound Blaster 16 or AWE -32
06	软盘控制器	标准软盘控制器，1.44M
07	LPT1	打印端口(LPT1)
08	实时时钟	系统 CMOS/实时钟
09	未用	未用
10	未用	NE2000 Compatible(ISA)
11	未用	未用
12	PS/2, Inport 鼠标	IRQ Holder for PCI Steering
12	显示卡	S3 Vision 868 PCI
13	算术协处理器	数值数据处理器
14	硬盘控制器 1	Intel 82371FB PCI Bus Master IDE Controller
14		Primary IDE Controller(dual fifo)
15	硬盘控制器 2	Intel 82371FB PCI Bus Master IDE Controller
15		Primary IDE Controller(dual fifo)

二、 中断级别的选择与设置

---- 我们以网卡为例，描述硬件中断级别的选择与设置方法。

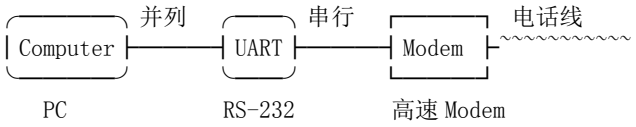
---- 网卡是通过IRQ中断通知CPU响应LAN网卡请求的。LAN网卡有权这样操作CPU主要是因为网卡的缓冲区容量有限，如果CPU不能尽快取得这些数据，那么当后续数据进入网卡缓冲区时，就会覆盖当前缓冲器的数据并刷新冲区。

---- 配置LAN网卡时，要为网卡设置一个IRQ等级。对于某些旧式网卡,可通过卡上的开关或者跳线设置其IRQ；而对于新式网卡，则通过运行设置程序，改变Flash Rom内的设定信息的方式完成。该程序首先对用户系统进行检测，寻找一个合适的IRQ（IO地址也一样），然后建议用户采用这些设置。

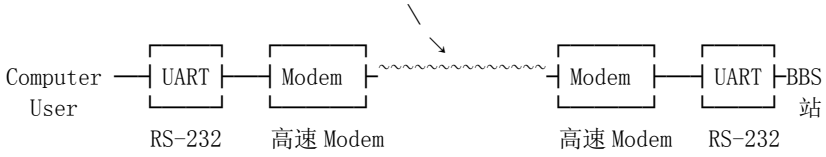
---- 有些人把他们的网卡中断级别设置为IRQ2，这里有潜在的隐患。原因还得从8位PC/XT机器说起。PC/XT上的IRQ2可以随意使用，但在高档PC机器上却有特殊作用。
---- 早期的PC/XT机器只有一个中断控制器，也就是Intel8259芯片。8259最多支持8个中断，PC/XT机器把通道0和通道1分别连接到系统定时器（每秒计数18.2次的时钟电路）和8255键盘控制器。
---- 由于IBM希望保证键盘和定时器具有最高优先级，因此将这两个中断进行固化。在8259中，两个中断同时发生时，中断级别越低的优先级越高。中断3，4分别用作Com2和Com1，因为Com2支持Modem，而Com1支持打印机，显然Modem的优先级要高一些。
---- 中断5、6和7分别被硬盘控制器、软盘控制器和并行口占用。
---- 随着第一台16位PC兼容机—IBM/At的推出。市场上扩展设备的势头很旺，8个中断级已明显不足。于是，IBM决定增加一个8259芯片，但强行在主板上增加一个额外的8259会产生向下兼容的问题，因此IBM决定采取迂回的方法，悄悄加入8259。
---- IBM的做法是：首先获取新的IRQ8-IRQ15，通过其输出端1NT统一发送，1NT又与IRQ2相连。一旦有IRQ8到IRQ15的请求信号，1NT便发出请求信号，这就使IRQ2好象是由1NT产生的中断。而PC BIOS却知道只要IRQ2中断产生，就要去检查第二个8259，找出真正发出中断请求的IRQ。同时，IBM还同时放弃了IRQ5的占用，因此AT以及以后的机器的硬盘控制器不再占用IRQ5。
---- 因此，不要占用IRQ2，因为IRQ2已经作为连接IRQ8-IRQ15的大门。一定要使用IRQ2时候，要通知软件已经将网卡设置为1NT。由于IRQ2与1NT连接在一起，因此在系统中其功能相当。
---- 由于IRQ8-IRQ15是通过IRQ2连入主机的，它们实际上继承了IRQ2的优先级。即IRQ8-IRQ15的优先级高于IRQ3-IRQ7。所以，安全的IRQ是5、10、11和15，应避免使用其他的IRQ。用户可将这些IRQ用于下列硬件设备。
声霸卡，对于8位声霸卡，只能选择IRQ5
LAN网卡
SCSI主适配器
三、合理而有效地利用中断资源
---- 1、安装多个网卡
---- 中断09，10，11可以为网卡所用。一个机器甚至可以安装三个以上的网卡，不过实际上两个网卡足够了，留下一个高中断可以安装PCI的声卡，以便空缺出中断5，留作下文所述的用途。此外，最好选择PnP的PCI网卡。因为如今流行的的Pentium或者PII主板上的ISA扩展槽最多不超过4个，PCI为4个，逐步流行5个，因此ISA只留下3个扩展槽。
---- 2、安装多个Modem
---- 我还没有发现超过8位的Modem，更不可能为PCI方式了，除非Cable Modem或者CDSL Modem传输带宽超过了50M。至少目前，内置Modem都采用ISA插槽，16位声卡，或ISA网卡服务也要占用这些插槽，因此需要合理安排，巧妙使用。实现在一个机器上安装两个以上的Modem，可以节约多串口卡或终端服务器的的投资，对于建立在NT和Win9x系统上的中小型远程信息服务系统来说，这是经济可行的方案。
---- 具体方法如下:
---- 选择PS/2鼠标，Com1和Com2可以选择连接内置或者外置的Modem，分别占用中断3和4。第三个Modem，只能是内置的并带有中断选择调线的，端口可以选择Com3或者Com4，中断则通过调线设定为5。这样通过3条电话线路，利用NT/95，就可以设置一个中小型的远程信息服务系统，同时实现拨入/拨出服务或者3人同时在线，且几乎不需额外的设备投资。
四、综述
----清楚地了解计算机中断资源的配置情况，再结合I/O地址分析，不仅可以避免并排除一些系统升级和扩充时所出现的“冲突”，也可以灵活地改变一些外设的中断响应级别，并充分利用有限的系统资源，最大限度发挥计算机性能的目的。欢迎email切磋、交流。

附件 9. Intel 16550 简介

16550 这是一颗足以取代 8250 的 UART (Universal Asynchronous Receiver Transmitter).
UART 是用来转换 并行传输的资料 与 串行传输的资料,
应此他是 Modem 与 计算机间的重要桥梁.
当使用高速 Modem 接收资料时, 资料大量涌入, CPU 的工作又十分忙碌, 一个好的 UART 便能适时的负起协调的工作.



高速 Modem : 只要是超过 4800 bps 的 Modem 都称之.
我以下解释的是外接式 Modem, 非内接(插卡的) 式.
现在市售的高速 Modem 有 9600 ,14.4K, 16.8K bps, 预计明年会有新的 28.8K bps, 这些速率指的是 Modem 与 Modem 间的最高速率.



或许你会觉的奇怪, Modem 与 Modem 间不就是电话线吗! 电话的频宽怎会有 16.8K 甚至 28.8K 这幺宽! 当然这需要使用其它的电子技术, 尤其是全双工 (Full-duplex) 时.

bps : 指的是 每秒(S-econd) 传输多少 位(B-it).

相信你使用过 ZIP ARJ LHA 这类的程序, Modem 当然也可以有自己的压缩能力(MNP5 与 V.42bis 便是). 高速 Modem (以 14.4K bps 为例) 已经够快了, 再加上这些特异功能,传输的资料量(有可能大于 14.4K bps) 可想而知, 记得上次说的桥梁吧 (UART) ! UART 必须承受这幺大的传输量, 万一 CPU 来不及从 UART 那里取走 Data, Modem 又把 Data 死命的塞向 UART , 那就 ....

传输量到底有多大! 要看 MNP5 或 V.42bis 发挥多少.
MNP5 : 最大压缩量 2 倍.
V.42bis : 最大压缩量 4 倍.
这也就是为何在高速 Modem 下 Telix 里要把 Boud 设到 38400 或 57600 bps.

或许你会说, 利用控制讯号不就得了! 实际上可没那幺容易, 因此 16550 便诞生了。它有 16 Bytes 的 Buffer 可以缓冲 Modem 与 Computer 间的资料流量(有点像 水库 的功能) .
说了半天, 到底要不要使用 16550. 我再举个例, 若一条河流量适当而稳定, 下游的人取水种田, 快乐的不得了, 那你要不要花钱盖水库? 若是不盖, 大雨来临, 河水泛滥, 辛苦种植付之一炬. 若是盖了, 说不定风调雨顺, 老死了都没遇上狂风暴雨, 那钱不是白花了.

你只要把计算机中 RS-232 适配卡上的 UART(8250 或 16450) 换成 16550 便可.

就是把旧的芯片拔起来, 再把新的插上去, 够容易了吧!
8250 : 是 PC-XT 使用的 UART, 有 40 只接脚, 只有一个 Receiver Buffer, 可能的编号有 8250 , 8250A , 82450 , 82C450 ...
16450 : 是 PC-AT 使用的 UART, 较 8250 快, 有 40 只接脚, 只有一个

Receiver Buffer, 可能的编号有 16450 , 16450N , 86450 ..
16550 : 是 PC-AT 使用的 UART, 有 40 只接脚, 兼容于 8250 和 16450
有 16 个 Recriver Buffer ,可能的编号有 16550 , 16550A ,
16550AN , 16550AFN , 16C550CP ...
Reciver Buffer : UART 从 Modem 收到串行资料时, 必须将其组成并

ps:据 GSZ 程序说, 又有一颗有 32bytes 的 16650ic

题要：16550 简介

近来 MODEM 区有很多人在讨论 16550 这颗 CHIP.但现在的 RS-232 卡′大多′是 MULTI CARD (即集合 2S/1P/1G/AT BUS)) 并无 40 脚的插座可放 16550。此时您就得去买一个单纯的 2S/1P 卡 (约 $300) 把上面的 86C450 弄下来再装上 16550 即可;千万不要买到 16550A 这颗 CHIP 而是 NS16550AFN,因为 16550A 有 Bug。
★最近又出了一颗新的其名字有所变更为 PC16550AFN。

UART 是 PC port 或 RS-232C 适配卡中的重要组件(UART-Universal Asynchronous Receiver Transmitter),早期用的 UART 是 8250,接着是 16450,后为 16550。这三个芯片都是 40-pin ,且接脚都兼容。

8250 是 IBM PC 时代用的异步通讯接口上的 UART ,为 8-bit chip,而 16550 是配合 AT 级 PC 或更快的 BUS 速度,为 16-bit chip、而 16450 是 8250 的快速兼容品罢了!如你目前使用高速 MODEM 的朋友,最好是换装由 National Semiconductor 制造的 16550AF 也就是 NS16550AFN。==>PC16550AFN

8250	8-bit	1-byte	FIFO	buffer	(先进后出缓冲区)
16450	16-bit	1-byte	FIFO	buffer	(先进后出缓冲区)

也就是 NS16550AFN。==>PC16550AFN

8250	8-bit	1-byte	FIFO	buffer	(先进后出缓冲区)
16450	16-bit	1-byte	FIFO	buffer	(先进后出缓冲区)
16550	16-bit	16-byte	FIFO	buffer	(先进先出缓冲区)

基本上 UART 是透过中断与 CPU 沟通,在接收资料过程中,UART 负责收集传入的资料位,并暂存在 buffer。当 buffer full 时通知 CPU 将资料取走。

支持 16550AFN 的通讯程序有,Telix 3.15、CrossTalk、...。Telix 使用时必须将设定中的 16550 buffer 开启即可。

多任务环境下,NS16550AFN 更是必备的,Windows 3.1 以经支持 57600 bps,而 Windows 3.0 只支持到 19200bps。

附件 10. Intel 8253 简介

8253 和 8254 均为 PC 使用之定时器，两者除了外接的频率限制不同外（前者最多只能接 2MHz，后者可接至 8MHz，8254-2 更可接至 10MHz），其余部份均兼容。但以 PC 定时器外接频率才 1.19MHz 来看，两者在使用上并没有什么差别，因此以下之定时器，我们均以 8253 来称呼。

8253 具有三个 16 位的计数器，可分别加以规划。每个计数器均具有一个输入的频率、一个控制闸（GATE）和一个输出端，其中控制闸便是用来控制输出端是否输出结果。以下为 PC 对于这三个 16 位计数器的布线方式及作用：

计数器 0
　　作用　　= 系统定时器
　　输入频率 = 1,193,180Hz
　　GATE0　 = 接至+5V
　　输出　　= 第一个 8259 之 IRQ0
计数器 1
　　作用　　= DRAM 更新要求产生器
　　输入频率 = 1,193,180Hz
　　GATE1　 = 接至+5V
　　输出　　= 更新要求周期
计数器 2
　　作用　　= 喇叭音调频率
　　输入频率 = 1,193,180Hz
　　GATE2　 = 接至 Port 61h 位 0
　　输出　　= 喇叭

由上面的资料我们可以知道，在 PC 上，除了 GATE2（也就是 Port 61h 的位 0）可以控制计数器 2 是否有输出外（控制喇叭是否发声），其余两个计数器我们均无法禁止其计数结果的输出，顶多是更改其输出频率和输出波形而已。对于三个计数器的输出规划及计数值读写，首先必须写出控制字组到控制缓存器中，这个缓存器在 PC 上是经由 Port 43h 写入的，之后再分别由 Port 40h～42h 读写计数器 0～2 的值。控制字组的格式为：

PORTS 43H:
位 7～6 = 计数器选择（0～2）
位 5～4 = 操作方式
　　00 = Latch 住目前的计数值
　　01 = 读写计数值的高字节
　　10 = 读写计数值的低字节
　　11 = 依次读写计数值的低、高字节
位 3～1 = 模式，各值意义为
　　000 - 模式 0
　　001 - 模式 1
　　x10 - 模式 2
　　x11 - 模式 3
　　100 - 模式 4
　　101 - 模式 5
位 0 = 0 表二进制计数，1 表 BCD 制计数

以下为各输出模式的意义：

1.模式 0（Interrupt on Terminal Count）
　　计数时输出为 Low，当计数完毕后，输出升至 High，然后重新计数，并将输出再还原为 Low。

2.模式 1（Programmable One-Shot）

当 GATE 由 Low 升成 High 时开始计数，计数时输出为 Low，当计数完毕后，输出升至 High 并结束计数。

3.模式 2（Rate Generator）

当 GATE 为 High 时开始计数，计数时输出为 High，当计数完毕后，输出降为 Low 一个 Clock 后升为 High，然后继续重新计数。

4.模式 3（Square Wave Generator）

当 GATE 为 High 时开始计数，且以 2 递减，当计数完毕后，输出高低互变，并继续重新计数。

5.模式 4（Software Triggerd Strobe）

计数时输出为 High，当计数完毕后，输出降为 Low 一个 Clock 后升为 High，并停止计数。若 GATE 为 Low 时停止计数。

6.模式 5（Hardware Triggerd Strobe）

当 GATE 由 Low 升成 High 时开始计数，计数时输出为 High，当计数完毕后，输出降为 Low 一个 Clock 后升为 High，并停止计数。

由于在 PC 上，各计数器之用途及外接线路均已定死，因此**计数器 0 必须采用模式 3；计数器 1 必须采用模式 2；计数器 2 则必须采用模式 3**，否则输出之波形将无法与线路配合，而产生奇怪的现象。以下，作者再进一步介绍各计数器的计数值在 PC 上的规划方式：

1.计数器 0（PORT 40H）

计数器 0 在 PC 上是做为时序之控制，以 18.2Hz 定频触发 INT 8h。其初值为 0，也就是每计数 65536 中断一次 INT 8h，因此 1,193,180/65536=18.2Hz。通常这个计数值是不加以更动的，以免影响到系统时间的运作，但是若欲做为短时间定频中断之特殊运用时，更动也是允许的。不过相同功能也可以运用 146818（INT 70h）来完成，因此更改计数器 0 的计数值便不是那幺的需要。以下为本计数器的计数值写入方式：

```
out 43h,36h
out 40h,计数值低字节
out 40h,计数值高字节
```

2.计数器 1（PORT 41H）

计数器 1 在 PC 上是做为 DRAM 定时更新使用，在 XT 上是接到 8237 DMA 通道 0 上，但在 AT 上已不再使用 DMA 式更新了。这个计数器的初值为 18，也就是每隔 15us 更新 DRAM 一次。由于 DRAM 的更新频率对于程序运用上较无关，因此这个计数器的计数值通常我们不会加以改变。以下为本计数器的计数值写入方式：

```
out 43h,74h
out 41h,计数值低字节
out 41h,计数值高字节
```

3.计数器 2（PORT 42H）

计数器 2 在 PC 上是用来控制喇叭发声的频率，应写入的计数值为 1,193,180/频率。以下为本计数器的计数值写入方式：

```
out 43h,0B6h
out 42h,计数值低字节
out 42h,计数值高字节
```

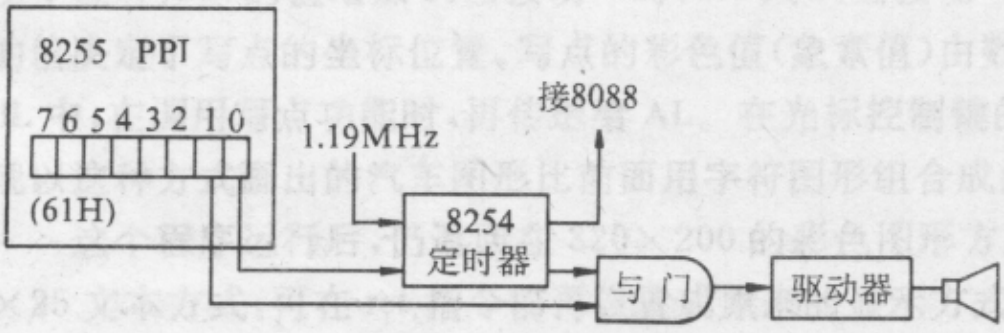
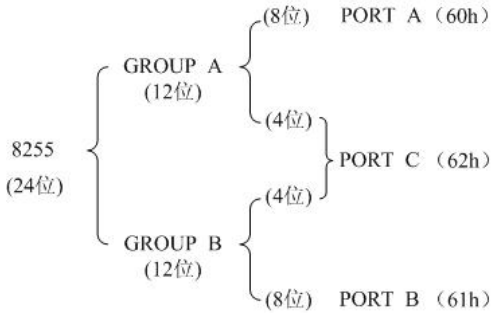


图 11.1 扬声器驱动系统

附件 11. Intel 8255 简介

Intel 8255 为可编程的并行输入输出接口（PPI-programmable peripheral interface）。8255 为可程序化的外围界面，只用在 XT 上，AT 则已取消使用这个组件，因此对于 8255，我们只说明和 XT 相关且较重要的内容，至于详细的内容，请参阅其它相关书籍。8255 具有 24 位的 I/O 线，并区分为两组：Group A 和 Group B，分别占用 12 位。而这 24 位的 I/O 线又可区分为三个输出入埠，其中 Port A 和 Port B 系由 Group A 和 Group B 中的 8 个位 I/O 线所组成，Port C 则分为高四位和低四位，分别由 Group A 和 Group B 剩下的四位元所组成。因此在规划上，Port C 可针对高四位和低四位分别程序化成不同的输出入方式和处理模式，Port A 和 Port B 则不行。



8255 对于这 24 位 I/O 线的规划方式，系经由其内部控制字组来完成的，端口号 63H，这个控制字组的意义如下：

1. 模式定义格式

位 7 = 必须固定为 1
位 6～5 = Group A 模式，当位 6 为 1 时，一律为模式 2
位 4 = Port A 输出入方式，0 表输出，1 表输入
位 3 = Port C 高四位输出入方式，0 表输出，1 表输入
位 2 = Group B 模式
位 1 = Port B 输出入方式，0 表输出，1 表输入
位 0 = Port C 低四位输出入方式，0 表输出，1 表输入

说明：XT 的设定值为 99h，即 Port A、Port C 为输入，Port B 为输出，采用的模式均为模式 0。至于 8255 可规划的三种模式名称分别为：

模式 0 = Basic Input/Output
模式 1 = Strobed Input/Output
模式 2 = Bi-Directional Bus

2. 位重置格式

位 7 = 必须固定为 0
位 6～4 = 不使用
位 3～1 = 选择的位
位 0 = 设定值

说明：位重置格式只用在模式 1 和模式 2 上，用来控制中断讯号产生的禁能和允能。

在 XT 上，控制字组是在 I/O Port 63h，而 Port A、Port B、Port C 则分别在 I/O Port 60h、61h、62h。其中 Port A 除了用来做为键盘资料的输入外，亦可做为 DIP 开关 1 的输入值，这项差别由 Port B 位 7 来控制。Port B、Port C 各位的意义如下：

1. Port B(61h)

位 7 = 0 表启用键盘，允许键盘中断的发生，Port A 将读取键盘扫描码；1 表键盘禁能，并清除键盘的输入，Port A 将读取 DIP 关闭 1 的值。
位 6 = 0 表强迫键盘时基于低电位，1 表键盘时基正常作用
位 5 = 0 表启用 I/O 信道检查

位 4 = 0 表启用 RAM 同位检查，286 及后续处理器中，每 15.085us 变换一次状态
位 3 = 控制录音机马达，0 表运转
位 2 = 值 1 表 PC0 读取 DIP 开关 2 的位 0，0 表读取位 4
位 1 = 喇叭资料输入控制
位 0 = 定时器 Gate 2 控制

说明：喇叭的资料输入系接至定时器 2，而定时器 2 的输出与否由 Gate 2 控制。在输至喇叭前还有一个 AND 闸控制，因此欲使喇叭发声，必须使得位 1 和位 0 均成为 1，此时喇叭便会以定时器 2 的输出频率发声。关于定时器的部份，请参阅 8237 的说明。

2.Port C(62h)

位 7 = 1 表 RAM 同位检查错误
位 6 = 1 表 I/O 信道检查错误
位 5 = 定时器 2 输出讯号
位 4 = 录音机资料输入
位 3～0 = DIP 开关 2 读取值。

在 AT 上，8255 已不再使用。原 Port 60h 改接至 8042 键盘控制器，仍做为键盘资料读取使用。Port 61h 则改用 ALS175 来取代，并集结了 Port B 和 Port C 中几个较重要的位。至于 Port 62h 和 63h，则均舍弃不用了。

以下为 AT 中 Port 61h 各位的意义，其中高 4 位为仅读位：

位 7 = 1 表 RAM 同位检查错误
位 6 = 1 表 I/O 信道检查错误
位 5 = 定时器 2 输出讯号
位 4 = RAM Reflesh 讯号
位 3 = 1 表启用 I/O 信道检查
位 2 = 1 表启用 RAM 同位检查
位 1 = 喇叭资料输入控制
位 0 = 定时器 Gate 2 控制

备注：以上关于 8255 各 I/O 位在 XT 上的作用，作者参阅多本 XT 书籍后，发现所着内容均有所出入。虽然作者针对相异的部份，尽量和 BIOS 程序及线路图搭配来做判断，但因找不到 XT 机器做实际测试，因此不敢保证内容完全正确，但至少错误的机会应已较少。

附件 12. Intel 8259 简介

- **Overview**

我们已经提到，中断的来源除了来自于硬件自身的 NMI 中断和来自于软件的 INTn 指令造成的软件中断之外，还有来自于外部硬件设备的中断，这些中断是可屏蔽的。这些中断也都通过 PIC(Programmable Interrupt Controller)进行控制，并传递给 CPU。在 IBM PC 极其兼容机上所使用的 PIC 是 Intel 8259A 芯片。8259A 芯片的功能非常强大，但在 IBMPC 上，我们只用到比较简单的功能。我们本节也只讨论其在 PC 上的使用。

一个 8259A 芯片的可以接最多 8 个中断源，但由于可以将 2 个或多个 8259A 芯片级连（cascade），并且最多可以级连到 9 个，所以最多可以接 64 个中断源。早期，IBMPC/XT 只有 1 个 8259A，. 但设计师们马上意识到这是不够的，于是到了 IBMPC/AT，8259A 被增加到 2 个以适应更多外部设备的需要，其中一个被称作 Master，另外一个被称作 Slave，Slave 以级连的方式连接在 Master 上。如今绝大多数的 PC 都拥有两个 8259A，这样最多可以接收 15 个中断源。

通过 8259A 可以对单个中断源进行屏蔽。

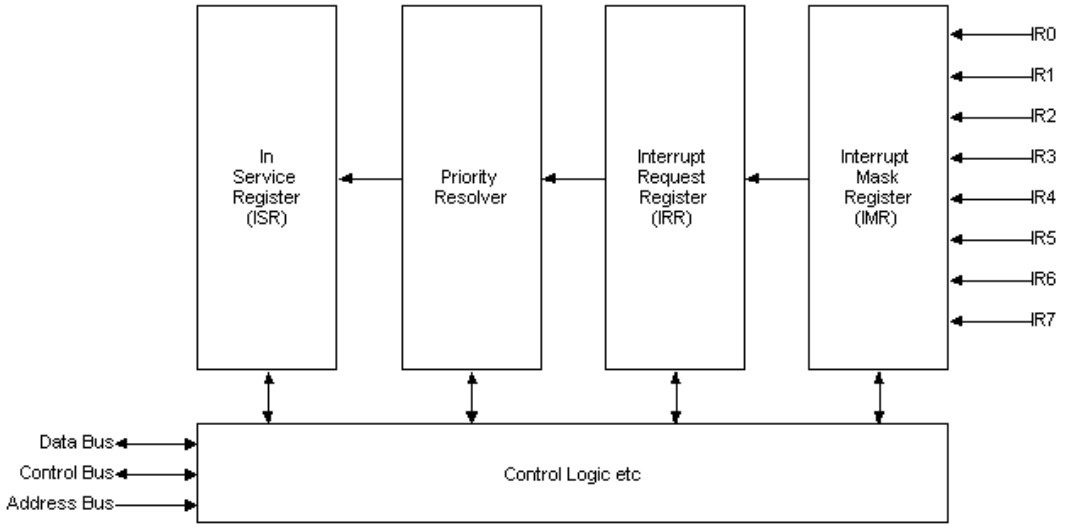
- **Principle**

在一个 8259A 芯片有如下几个内部寄存器：

- Interrupt Mask Register (IMR)
- Interrupt Request Register (IRR)
- InSevice Register (ISR)

IMR 被用作过滤被屏蔽的中断；IRR 被用作暂时放置未被进一步处理的 Interrupt；当一个 Interrupt 正在被 CPU 处理时，此中断被放置在 ISR 中。

除了这几个寄存器之外，8259A 还有一个单元叫做 Priority Resolver，当多个中断同时发生时，PriorityResolver 根据它们的优先级，将高优先级者优先传递给 CPU。



当一个中断请求从 IR0 到 IR7 中的某根线到达 IMR 时，IMR 首先判断此 IR 是否被屏蔽，如果被屏蔽，则此中断请求被丢弃；否则，则将其放入 IRR 中。

在此中断请求不能进行下一步处理之前，它一直被放在 IRR 中。一旦发现处理中断的时机已到，PriorityResolver 将从所有被放置于 IRR 中的中断中挑选出一个优先级最高的中断，将其传递给 CPU 去处理。IR 号越低的中断优先级别越高，比如 IR0 的优先级别是最高的。

8259A 通过发送一个 INTR(InterruptRequest)信号给 CPU，通知 CPU 有一个中断到达。如果 CPU 的标志寄存器 IF 位为 1，允许响应可屏蔽中断请求，此时 CPU 收到这个信号后，会暂停执行下一条指令，然后发送一个 INTA(InterruptAcknowledge)信号给 8259A。8259A 收到这个信号之后，马上将 ISR 中对应此中断请求的 Bit 设置，同时 IRR 中相应的 bit 会被 reset。比如，如果当前的

中断请求是 IR3 的话，那么 ISR 中的 bit-3 就会被设置，IRR 中 IR3 对应的 bit 就会被 reset。这表示此中断请求正在被 CPU 处理，而不是正在等待 CPU 处理。

随后，CPU 会再次发送一个 INTA 信号给 8259A，要求它告诉 CPU 此中断请求的中断向量是什么，这是一个从 0 到 255 的一个数。8259A 根据被设置的起始向量号（起始向量号通过中断控制字 ICW2 被初始化）加上中断请求号计算出中断向量号，并将其放置在 DataBus 上。比如被初始化的起始向量号为 8，当前的中断请求为 IR3，则计算出的中断向量为 8+3=11。

CPU 从 DataBus 上得到这个中断向量之后，就去 IDT 中找到相应的中断服务程序 ISR，并调用它。如果 8259A 的 End of Interrupt(EOI)通知被设定为人工模式，那么当 ISR 处理完该处理的事情之后，应该发送一个 EOI 给 8259A。

8259A 得到 EOI 通知之后，ISR 寄存器中对应于此中断请求的 Bit 会被 Reset。如果 8259A 的 End of Interrupt(EOI)通知被设定位自动模式，那么在第 2 个 INTA 信号收到后，8259AISR 寄存器中对应于此中断请求的 Bit 就会被 Reset。

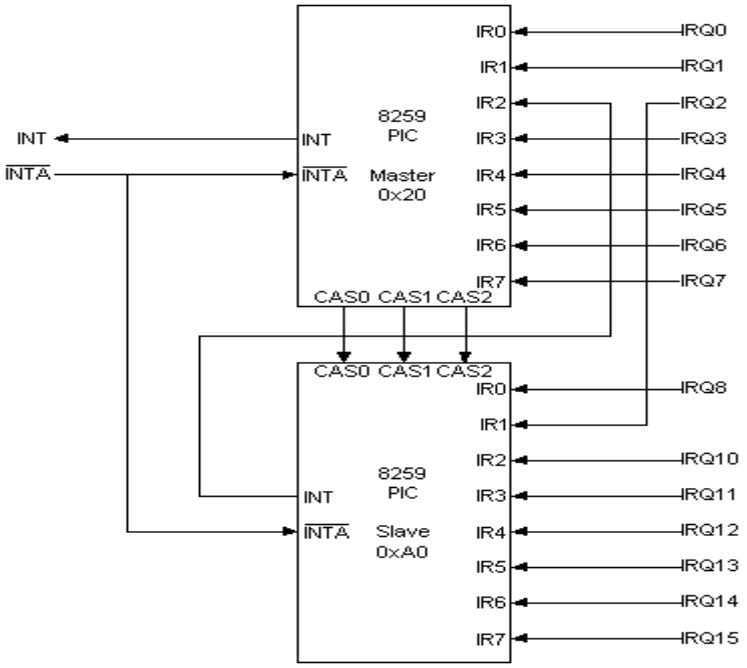
在此期间，如果又有新的中断请求到达，并被放置于 IRR 中，如果这些新的中断请求中有比在 ISR 寄存中放置的所有中断优先级别还高的话，那么这些高优先级别的中断请求将会被马上按照上述过程进行处理；否则，这些中断将会被放在 IRR 中，直到 ISR 中高优先级别的中断被处理结束，也就是说知道 ISR 寄存器中高优先级别的 bit 被 Reset 为止。

- **IRQ2/IRQ9 Redirection**

为什么要将 IRQ2 重定向到 IRQ9 上？这仍然是由于兼容性问题造成的。

早期的 IBMPC/XT 只有一个 8259A，这样就只能处理 8 种 IRQ。但很快就发现这根本不能满足需求。所以到了 IBMPC/AT，又以级连的方式增加了一个 8259A，这样就可以多处理 7 种 IRQ。原来的 8259A 被称作 Master PIC，新增的被称作 SlavePIC。但由于 CPU 只有 1 根中断线，Slave PIC 不得不级连在 Master PIC 上，占用了 IRQ2，那么在 IBMPC/XT 上使用 IRQ2 的设备将无法再使用它；但新的系统又必须和原有系统保持兼容，怎么办？

由于新增加的 Slave PIC 在原有系统中不存在，所以，设计者从 SlavePIC 的 IRQ 中挑出 IRQ9，要求软件设计者将原来的 IRQ2 重定向到 IRQ9 上，也就是说 IRQ9 的中断服务程序需要去掉用 IRQ2 的中断服务程序。这样，将原来接在 IRQ2 上的设备现在接在 IRQ9 上，在软件上只需要增加 IRQ9 的中断服务程序，由它调用 IRQ2 的中断服务程序，就可以和原有系统保持兼容。而在当时，增加的 IRQ9 中断服务程序是由 PC 开发商开发的 BIOS 提供的，所以就从根本上保证了兼容。



- **Programmingthe8259As**

每一个 8259A 芯片都有两个 I/O ports，程序员可以通过它们对 8259A 进行编程。

Master8259A 的端口地址是 0x20，0x21；Slave 8259A 的端口地址是 0xA0，0xA1。
程序员可以向 8259A 写两种命令字：

1. *Initialization Command Word(ICW)；* 这种命令字被用作对 8259A 芯片的初始化。
2. *Operation Command Word(OCW)：* 这种命令被用来向 8259A 发布命令，以对其进行控制。OCW 可以在 8259A 被初始化之后的任何时候被使用。

下表的内容是 Master8259A 的 I/O 端口地址，以及通过它们所能操作的寄存器。

Address	Read/Write	Function
0x20	Write	Initialization Command Word 1(ICW1)
	Write	Operation Command Word 2(OCW2)
	Write	Operation Command Word 3(OCW3)
	Read	Interrupt Request Register(IRR)
	Read	In-Service Register (ISR)
0x21	Write	Initialization Command Word 2(ICW2)
	Write	Initialization Command Word 3(ICW3)
	Write	Initialization Command Word 4(ICW4)
	Read/Write	Interrupt Mask Register(IMR)

Addresses/Registers for Master 8259A

下表的内容是 Slave8259A 的 I/O 端口地址，以及通过它们所能操作的寄存器。

Address	Read/Write	Function
0xA0	Write	Initialization Command Word 1(ICW1)
	Write	Operation Command Word 2(OCW2)
	Write	Operation Command Word 3(OCW3)
	Read	Interrupt Request Register(IRR)
	Read	In-Service Register (ISR)
0xA1	Write	Initialization Command Word 2(ICW2)
	Write	Initialization Command Word 3(ICW3)
	Write	Initialization Command Word 4(ICW4)
	Read/Write	Interrupt Mask Register(IMR)

Addresses/Registers forSlave8259A

由于 8259A 芯片不仅能够用于 IBMPC/X86，也可以被用作 MCS-80/85，对于这两者，在操作模式上有一些不一样，对于某些寄存器的设置也有所不同。我们后面仅仅讨论 X86 模式相关的内容。

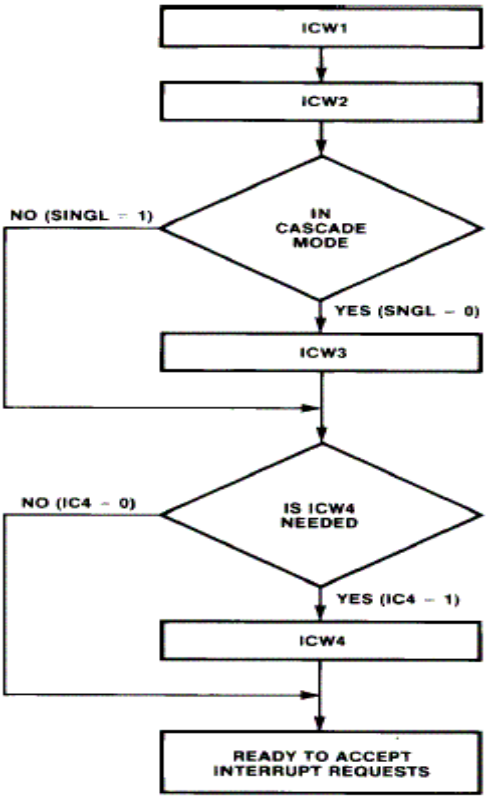
- Initialization

当主机 Power-on 或 Reset 之后，必须对两个 8259A 都进行初始化。事实上，BIOS 已经这么做了。但不幸的是，BIOS 对其进行的初始化的结果并非我们所需要。比如，我们要开发 ProtectedMode 下 OS，我们要设置自己的 IDT（中断描述符表），那么我们就不能使用 BIOS 设置的 IVT（中断向量表），而在对 8259A 初始化操作中，我们需要告诉 8259A，其相关中断请求的起始向量号，而我们对 IDT 的中断向量布局和 BIOS 设置的 IVT 的中断向量布局可以是不一样的。这样，我们也需要对两个 8259A 进行初始化。

任何时候，只要向某一个 8259A 的第一个端口(0x20 for Master, and 0xA0forSlave)写入的命令的 bit-4(从 0 算起)为 1，那么这个 8259A 就认为这是一个 ICW1；而一旦一个 8259A 收到一个

ICW1，它就认为一个初始化序列开始了。你可以通过对照上边的表和后面的表，第一端口可写的有ICW1，OCW2 和 OCW3。而 ICW1 的 bit-4 要求必须是 1，但 OCW2 和 OCW3 的 bit-4 要求必须是 0。

8259A 的初始化流程协议如下图所示，程序员对其进行初始化时必须遵守此协议：



ICW1

Bit(s)		Function
7:5		Interrupt Vector Addresses forMCS-80/85 Mode.
4		Must be set to 1 forICW1
3	1	Level Triggered Interrupts
	0	Edge Triggered Interrupts
2	1	Call Address Interval of 4
	0	Call Address Interval of 8
1 (SINGL)	1	Single PIC
	0	Cascaded PICs
0 (IC4)	1	Will be Sending ICW4
	0	Don't needICW4

Initialization Command Word 1 (ICW1)

对于 X86，bit-0 必须被设置为 1；由于当今的 IBMPC 上都有两个级连的 8259A，所以 bit-1 应该被设置为 0；由于 bit-2 是为 MCS-80/85 服务的，我们将其设置为 0；bit-3 也设置为 0；bit-4 被要求必须设置为 1；bit5:7 是为 MCS-80/85 服务的，对于 X86，应将全部将其设为 0。

所以，在 X86 系统上，ICW1 应该被设置为二进制 00010001 =0x11。

ICW2

Bit	80x86Mode
7	I7

6	I6
5	I5
4	I4
3	I3
2	0
1	0
0	0

Initialization Command Word 2 (ICW2)

ICW2 被用作指定本 8259A 中的中断请求的起始中断向量，bit0:3 必须被设为 0；所以，其起始中断向量必须是 8 的倍数。比如，我们的 OS 的设计讲来自于 Master8259A 的 8 个中断请求放在 IDT 的第 32 (从 0 开始计)个位置到第 39 个位置，则我们应该将 ICW2 设为 0x20。

这样，当将来此 8259A 上接收到一个 IRQ 时，其低 3 位会被自动填充为 IRQ 号。比如，其收到一个 IRQ6，将 6 自动填充到后 3 位，则生成的向量号为 0x26。8259A 会在收到 CPU 发来的第二个 INTA 信号之后，将生成的向量号放到 DataBus 上。

ICW3

Master 8259A 和 Slave8259A 有不同的 ICW3 格式。

Bit	Function
7	IR7 is connected to a Slave
6	IR6 is connected to a Slave
5	IR5 is connected to a Slave
4	IR4 is connected to a Slave
3	IR3 is connected to a Slave
2	IR2 is connected to a Slave
1	IR1 is connected to a Slave
0	IR0 is connected to aSlave

Initialization Command Word 3 for Master8259A (ICW3)

Slave 8259A 被接在 Master8259A 的那个 IRQ 上，则相应的位就被设置为 1，其余的位都被设置为 0。在 IBM PC 上，Slave 8259A 被接在 Master8259A 的 IRQ2 上，则此 ICW3 的值应该被设置为二进制 00000100 = 0x04。

Bit(s)	Function	
7	Reserved. Set to0	
6	Reserved. Set to0	
5	Reserved. Set to0	
4	Reserved. Set to0	
3	Reserved. Set to0	
2:0	*Slave ID*	
	000	Slave 0
	001	Slave 1
	010	Slave 2
	011	Slave 3
	100	Slave 4

	101	Slave 5
	110	Slave 6
	111	Slave7

Initialization Command Word 3 for Slaves(ICW3)

Slave8259A 的 ICW3 的 bit3:7 被保留，必须被设为 0；而 bit0:2 被设置为此 Slave 8259A 被接在 Master8259A 的哪个 IRQ 上。比如，在 IBM PC 上，Slave 8259A 被接在 Master8259A 的 IRQ2 上，则此 ICW3 应被设为 0x02。

ICW4

Bit(s)		Function
7		Reserved. Set to0
6		Reserved. Set to0
5		Reserved. Set to0
4	1	Special Fully Nested Mode
	0	Not Special Fully Nested Mode
3:2	0x	Non - Buffered Mode
	10	Buffered Mode - Slave
	11	Buffered Mode - Master
1	1	Auto EOI
	0	Normal EOI
0	1	8086/8080 Mode
	0	MCS-80/85

Initialization Command Word 4 (ICW4)

在 80x86 模式下，我们不需要使用 8259A 的特殊功能，因此我们将 bit1:4 都设为 0，这意味使用默认的 Full NestedMode，不使用 Buffer，以及手动 EOI 模式；我们只需要将 bit-0 设为 1，这也正是我们 ICW0 处提到的我们为什么必须要 ICW4 的原因。所以 ICW4 的值应该被设为 0x01。

所以我们可以用下列代码初始化 2 个 8259A 芯片：

```
inline voidinit_8259a(void)
{
 /* icw1 */
 outb( 0x11, 0x20 ); /* masterport A */
 outb( 0x11, 0xA0 ); /* slave port A */
 /* icw2 */
 outb(0x20, 0x21 ); /* master offset of 0x20 in the IDT */
 outb( 0x28, 0xA1 );/*slave offset of 0x28 in the IDT */
 /* icw3 */
 outb(0x04, 0x21 ); /* slaves attached to IR line 2 */
 outb( 0x02, 0xA1 ); /*this slave in IR line 2 of master */
 /*icw4 */
 outb(0x01, 0x21 ); /* set as master */
 outb( 0x01, 0xA1 ); /*set as slave*/
}
```

- **Operation**

一旦按照初始化协议初始化完成之后，程序员就可以在任何时候，以任何顺序向 8259A 发送操作控制字 OCW 了。

OCW1

Bit	PIC 2	PIC 1
7	Mask IRQ15	Mask IRQ7
6	Mask IRQ14	Mask IRQ6
5	Mask IRQ13	Mask IRQ5
4	Mask IRQ12	Mask IRQ4
3	Mask IRQ11	Mask IRQ3
2	Mask IRQ10	Mask IRQ2
1	Mask IRQ9	Mask IRQ1
0	Mask IRQ8	MaskIRQ0

Operation Control Word 1 (OCW1)

OCW1 是用来做中断请求屏蔽用的操作控制字。如果你想屏蔽那个 IRQ，只需要对照上表将相应的 Bit 置为 1，然后发送给相应的 8259A 就可以了。比如我想屏蔽 IRQ10，我只需要将 0x0A 写到端口 0xA1。对应代码如下：

```
outb(0x0A, 0xA1);
```

OCW2

Bit(s)	Function	
7:5	000	Rotate in Auto EOI Mode(Clear)
	001	Non Specific EOI
	010	Reserved
	011	Specific EOI
	100	Rotate in Auto EOI Mode(Set)
	101	Rotate on Non-Specific EOI
	110	Set Priority Command (Use Bits 2:0)
	111	Rotate on Specific EOI (Use Bits2:0)
4	Must be set to 0	
3	Must be set to 0	
2:0	000	Act on IRQ 0 or 8
	001	Act on IRQ 1 or 9
	010	Act on IRQ 2 or 10
	011	Act on IRQ 3 or 11
	100	Act on IRQ 4 or 12
	101	Act on IRQ 5 or 13
	110	Act on IRQ 6 or 14
	111	Act on IRQ 7 or15

OperationControl Word 2 (OCW2)

通过将 bit3:4 设置为 0，以说明这是一个 OCW2。如果 bit-6 被设为 1，则 bit0:2 有效，其操作则是面向某个 IRQ 的；否则将 bit0:2 设为 0，其操作是面向整个 8259A 的所有 IRQ 的。我们一般只会用到 NoSpecific EOI——因为我们在初始化 8259A 时，制定的 EOIMode 为手动模式，所以当每次对应某个 8259A 芯片的 IRQ 的中断服务程序 ISR 执行结束后，都需要向 8259A 发送一个

EOI，其对应的 OCW2 的值为 0x20。需要注意的是，由于 IBMPC 有 2 个级连的 8259A，所以我们每次必须分别给两个都发一个。

比如下面示例代码用来向两个 8259A 芯片发送 EOI，它需要在针对来自于两个 8259A 芯片的中断的服务程序 ISR 末尾处被调用：

```
inline void send_eoi(void)
{
 /*Send EOI to both master and slave */
 outb( 0x20, 0x20 ); /* master PIC*/
 outb( 0x20, 0xA0 ); /* slave PIC */
}
```

OCW3

Bit(s)	Function	
7	Must be set to 0	
6:5	00	Reserved
	01	Reserved
	10	Reset Special Mask
	11	Set Special Mask
4	Must be set to 0	
3	Must be set to 1	
2	1	Poll Command
	0	No Poll Command
1:0	00	Reserved
	01	Reserved
	10	Next Read Returns Interrupt Request Register
	11	Next Read Returns In-ServiceRegister

Operation Control Word 3 (OCW3)

通过将 Bit-3 设为 1，Bit-4 设为 0，以让 8259A 知道这是一个 OCW3。OCW3 中对我们最有意义的位是 bit0:1，我们可以通过将 bit-1 设为 1 来通知 8259A，下一个读端口的动作将要读取 IRR 或 ISR 寄存器的内容。

比如下面示例 C++代码用来读取 Master8259A 的 IRR 寄存器内容到__irr 变量中：

```
void read_irr(unsigned char&__irr)
{
 outb(0x02, 0x20);
 inb(&__irr,0x20);
}
```

- **Full Nested Mode**

为了让我们更加理解 8259A 的中断控制机理，我们需要说明一下 Full NestedMode。在我们初始化时，只需要将 ICW4 的 bit-4 设为 0，我们就选择了 Full NestedMode。

Full NestedMode 其实就是实现按照中断请求的优先级别进行抢断处理的机制——如果当前一个 IRQ 正在被 CPU 处理，也就是说，当前 CPU 正在调用其中断服务程序 ISR；这时 8259A 又接到了新的 IRQ，如果此 IRQ 的优先级大于正在处理的 IRQ，那么，此 IRQ 就会被提交给 CPU 以优先处理；否则此 IRQ 则被放置在 IRR 中，直到所有的高优先级中断被处理结束为止。

其处理过程大致如下：

在 ISR 寄存器中有一个 8-bit 的字节，范围为 bit[0,7]；每一个 bit 对应一个 IRQ（IRQ0-IRQ7 对应 bit[0,7]）。当一个 IRQ 被提交给 CPU 之后（收到来自于 CPU 的第一个 INTA 信号之

后），其对应的 bit 会被设置为 1。比如 IRQ6 被提交给 CPU 之后，ISRegister 的 bit-6 会被设置为 1。当此 8259A 收到一个 EOI 之后（对于手动模式，这意味着一个优先级别最高的中断请求被处理结束），会将 ISRegister 中被设置的最高优先级 IRQ 的对应的 bit 清为 0。比如在收到一个 EOI 时，发现 IS Register 的 bit-3, bit-5,bit-6 被设置，那么被清除的则是 bit-3（越小优先级别越高）。在清除优先级最高的 bit 之后，8259A 会到 IRR 中察看是否有优先级别高于当前正在处理的 IRQ 中优先级别最高的 IRQ，如果有，则将此 IRQ 提交给 CPU 处理，同时设置相应的 bit。还以上面的例子为例，当 bit-3 被清除之后，如果发现在 IRR 中有一个 IRQ4 等待被处理，则将其提交给 CPU，在收到来自于 CPU 的第一个 INTA 信号之后，则将 ISRegister 的 bit-4 置为 1。

在此过程中，如果 8259A 接到更高优先级别的 IRQ，则将其立即提交给 CPU。比如，当前正在处理的 IRQ 为 IRQ3，IRQ5，那么 ISRegister 中被设置的 bit 为 bit-3，bit-5；如果此时接到一个 IRQ1，则立即将其提交给 CPU，在收到来自于 CPU 的第一个 INTA 信号之后，则将 ISRegister 的 bit-1 置为 1。

由此过程我们也可以看出，为了实现这种优先级机制，必须将 EOI 设为手动模式，也就是说必须将 ICW4 的 bit-1 设为 0。因为，对于自动 EOI 模式，8259A 会在收到来自于 CPU 的第 2 个 INTA 信号之后，就自动将 ISRegister 中此 IRQ 对应的 bit 清 0，而事实上，这个时候此 IRQ 对应的中断服务程序还没有被 CPU 调用，也就是说此 IRQ 还没有被处理结束，而由于此 IRQ 对应的 bit 已经被清除，如果此 IRQ 是一个优先级很高的话，那么此 IRQ 的处理完全可以被一个优先级别更低的 IRQ 所中断。这不是我们所需要的。

另一篇：

8259 为一种可程序化的优先序中断控制器，所有 PC 外围硬件的中断均必须透过这个组件来触发所对应的中断处理程序。以下我们先说明 8259 的硬件特性。8259 共分成四个**启始命令字组 ICW** 和三个**操作命令字组 OCW**，前者做为初始化时使用，后者则可供程序自行使用。8259 初始化的过程为：

1 送出 ICW1
2 送出 ICW2
3 if 串接模式 then 送出 ICW3
4 if 需要 ICW4 then 送出 ICW4

各启始命令字组的意义为：
1.ICW1
位 7～5 = 中断向量地址（只用在 MCS-80 模式）
位 4 = 固定为 1
位 3 = 0 为边缘触发（edge trigger）
1 为位准触发（level trigger）
位 2 = 呼叫地址间隔（只用在 MCS-80 模式）
0 为呼叫地址间隔 8
1 为呼叫地址间隔 4
位 1 = 0 为串接模式（cascade mode）
1 为单一模式（single mode）
位 0 = 0 不需要 ICW4
1 需要 ICW4
说明：8259 除了可供 8088 系列使用外，也可供 MSC-80 系列使用，因此当任何位若标明只使用在 MSC-80 模式时，PC 上均不需使用。另外，由于 PC 上使用的是 8088 系列的 CPU，必须利用 ICW4 指定为 8088 模式，因此本启始命令的位 4、1 均必须固定为 1，位 7～5、2 则可为 0 可为 1。至于采用何种触发方式，则依电路特性而定，PC 上通常采用边缘触发，因此位 3 也大致固定为 0。比较有争议的是位 1 的值。在旧型的 PC/XT 上，由于只使用一个 8259 做为中断控制用，因此必须使用单一模式，故它的初值为 13h。但到了 PC/AT 时，由于系统功能加强，原来的一个 8259 已不敷使用，于是便扩增为两个 8259
串接在一起，因此这两个 8259 便都必须使用初值 11h。

2.ICW2
(1) MCS-80 模式
位 7～0 = 中断向量地址
(2) 8088 模式
位 7～3 = 中断向量地址
位 2～0 = 固定为 0
说明：这个命令是用来设定硬件中断时，所要执行的中断向量启始值。PC/AT 的第一个 8259 触发的中断在 08h～0Fh，因此初值为 08h；第二个 8259 触发的中断在 70h～77h，因此初值为 70h。这些初值若在不影响系统的中断功能下，是可以随意改变的。

3.ICW3
(1) 主控制器（master device）
位 7～0 = 各位分别表示 IRQx 是否有串接另一个 8259 控制器，0 表示没有，1 表示有。
(2) 次控制器（slave device）
位 7～3 = 固定为 0
位 2～0 = 次控制器的识别码。
说明：这个初始命令必须有两个以上的 8259 串接在一起时才使用的，因此 PC/XT 并不使用这个初始命令。至于 PC/AT 上的两个 8259，由于是串接在 IRQ2 上的，因此主控制器（也就是第一个

8259）的初值便必须为 04h。至于次控制器，由于必须配合所串接的 IRQ 来做编号，因此次控制器的初值便必须为 02h，否则无法正常工作。

4.ICW4

位 7～5 = 固定为 0
位 4 = 0 为非特别全巢状模式
1 为特别全巢状模式
位 3～2 = 0x 不带缓冲模式
10 主控制器带缓冲模式
11 次控制器带缓冲模式
位 1 = 0 表非自动结束中断（EOI）
1 表自动结束中断
位 0 = 0 表 MSC-80 模式
1 表 8088 模式

说明：特别全巢状模式（special fully nested mode），主要用在大系统中使用多个 8259 串接模式时，以便规划不同的中断权限，详细资料请参阅相关书籍。通常我们均使用非特别全巢状模式，也就是初始化时以 IRQ0 优先权最高，IRQ7 最低（这个优先权是可以改变的）。而在某个中断发生处理时，优先权更高的中断允许再发生，以中断目前正在处理的中断，而相同或较低优先权的中断则不允许发生。至于是否要规划成缓冲模式，必须视所连接系统之 Data Bus 线路而定。对于是否使用自动结束中断，则由操作系统或中断处理程序的设置程序来决定。一般使用自动结束中断对于中断处理程序会比较方便些，不会因忘了送出 EOI 命令（参阅 OCW2 命令）而使得系统功能停止或当机。但是它最大的缺点是，如果相同的 IRQ 中断一再发生，而该中断程序又使用了 STI 命令允许中断发生时（为了让更高优先权的中断产生），则可能造成中断程序再被中断，导致重入问题，以及中断顺序更动等问题。因此在绝大部份的情况下，大都会采用非自动中断结束，在 PC 上亦然。不过若使用非自动 EOI 的结束方式时，中断处理程序在处理完中断后必须送出一个 EOI 命令给 8259，表示中断已处理完，否则 8259 便视为该中断未结束而不再产生下个中断了。
备注：关于本初始命令的初值，由 XT 的 BIOS 查得为 09h，由 286 AT 的 BIOS 查得为 01h，也就是前者采用缓冲模式，后者并不采用缓冲模式。经查阅有关 8259 零件的手册，其中说明缓冲模式主要用在具有 BUS 驱动缓冲区的 Data Bus，由 8259 的 SP/EN 讯号来控制缓冲区。当不是缓冲模式时，这个讯号接脚在主控制器上必须接 Vcc，而在次控制器上则必须接地。再查阅 86 XT 和 286、386 AT 的线路图，发现确实 XT 的设计电路是使用缓冲模式，而 286、386 AT 的设计电路则为非缓冲模式。因此在初始化时，必须依 PC 的种类而有不同的初始化值（事实上，XT 只有一个 8259，在初始化过程方面便一定会有不太一样的地方，必须与 AT 分别处理），如果未与设计的电路做好匹配的规划，则有当机之虞（但并不一定会当机。据作者实际测试过，某些 AT 上将 8259 规划成缓冲模式会当机，而在某些 AT 上则不管是否规划成缓冲模式却仍都正常工作，这和主机电路板的线路设计有关）。

以下为各操作命令字组的意义：

1.OCW1

位 7～0 = 分别表示 IRQx 是否激活，0 表示允能（enable），1 表示
禁能（disable）。

说明：本命令也就是设定中断的罩遮值（IMR，Interrupt Mask Register）。在使用时，通常先读取原值后，再设置所要处理的中断所对应的位元，以避免影响到其它中断。

2.OCW2

位 7～5 = 000 取消自动 EOI 时旋转优先权模式
001 不特别指定 IRQx 方式结束中断
010 无作用
011 指定 IRQx 方式结束中断
100 设定自动 EOI 时旋转优先权模式

101 不特别指定 IRQx 结束中断时旋转优先权
110 设定优先权
111 指定 IRQx 结束中断时旋转优先权

位 4～3 = 固定为 0
位 2～0 = 指定的 IRQx

说明：本命令是用来指定中断优先权的处理方式（请参阅相关书籍），以及指示中断是否处理完毕。一般在 PC 上大都使用 20h 做不特别指定 RQx 方式来结束 8259 的中断。所谓不特别指定 IRQx 方式，指的是要处理的 IRQ 编号并不由位 2～0 来指定，而以目前已发生且优先权最高的中断为准。

3.OCW3

位 7 = 固定为 0
位 6～5 = 0x 无作用
10 取消特殊罩值（reset special mask）
11 设定特殊罩值（set special mask）
位 4 = 固定为 0
位 3 = 固定为 1
位 2 = 0 表使用中断方式服务
1 表不使用中断方式服务（poll command）
位 1～0 = 0x 无作用
10 读取中断需求缓存器（IRR）
11 读取内部服务缓存器（ISR）

说明：特殊罩值功能，主要是提供给需要任意改变中断的禁能、允能，而又不希望因未送出 EOI 命令，而导致其它较低优先权的中断无法发生的程序使用。当设定了特殊罩值后，则每次在 OCW1 重设罩值时，均会将所有未被罩遮掉的 IRQ 重新启用。IRR 及 ISR 这两个缓存器的值可用来辨别各中断的产生与否。IRR（Interrupt Request Register）记录了硬件已产生中断需求，但 8259 尚未发出中断讯号或 CPU 尚未响应确认讯号的中断；ISR（Interrupt Service Register）则记录了 CPU 已接收到中断讯号但处理程序尚未响应 EOI 命令的中断。当要读取这两个缓存器值时，首先必须送出 OCW3 指定所要读取的暂存器，然后再由相同的输出入埠将之读取。
备注：不使用中断服务的方式（poll command），依 8259 之零件说明为 RD 讯号会读取 Priority Level 值，其值意义如下：

位 7 = 1 表有中断发生
位 6～3 = 保留
位 2～0 = 中断发生的编号

又说明 Poll Command 是使用在 8259 的中断输出讯号不使用，或 CPU 线路已将中断讯号输入禁能的情况。据作者实际测试结果，在 PC 上将 OCW3 修改成 Poll Command 形态，并不影响到原中断的执行，读取而得的值也并非 Priority Level 的值。至于为何如此，作者遍寻相关书籍，也问过许多硬件设计从业人员，竟找不到答案。作者猜测可能是因 PC CPU 仍然接收并使用 8259 的中断输出讯号，使得 8259 的中断确认讯号并非由 Poll Command 的 RD 讯号确认所致，至于真正原因，则不得而知了。

当 8259 重新初始化后，IMR 的值均会重设为 00h（也就是允许所有的中断产生），并取消特殊罩值功能。因此当我们要重新初始化 8259 时，必须记录原来的 IMR 值，并在初始化后将之设回，以免初始化后某些原已被禁能的中断再次产生而发生问题。当然，初始化后视情况也必须重新设定 OCW3 的值。

上述所有命令字组在 PC 上均透过 I/O Port 20h、21h（第一个 8259）和 A0h、A1h（第二个 8259）来读写。I/O Port 20h 和 A0h 做为 ICW1、OCW2 和 OCW3 命令的输入，以及 IRR、ISR 缓存器值的输出。I/O Port 21h 和 A1h 则做为 ICW2、ICW3、ICW4 和 OCW1 命令的输入，以及 IMR 缓存器值的输出。

至于 8259 是如何去分辨送来的是那个命令字组，这倒不必我们费心，因为这些命令字组中大都已安排好某些位均固定为 0 或 1。由这些固定位，8259 便能轻易地辨别出所输入的命令种类。以下是目前在 PC/AT 中两个 8259 所处理的 16 个硬件中断内容：

```
                ┌ IRQ0 计时中断
                │ IRQ1 键盘中断
                │ IRQ2 串接第二个 8259
第一个 8259 ──┤ IRQ3 次串行通讯端口（COM2）
                │ IRQ4 主串行通讯端口（COM1）
                │ IRQ5 打印机中断（LPT2）
                │ IRQ6 软盘中断
                └ IRQ7 打印机中断（LPT1）
                ┌ IRQ0 实时时脉中断
                │ IRQ1 保留
                │ IRQ2 保留
第二个 8259 ──┤ IRQ3 保留
                │ IRQ4 保留
                │ IRQ5 数学处理单元中断
                │ IRQ6 硬盘中断
                └ IRQ7 保留
```

以下为各中断在本附录中所对应的硬件说明：

```
计时中断          = 8253、8254
键盘中断          = 8042、8255
实时时脉中断      = 146818
```

其余之硬件中断由于和本书内容无关，因此不予列入。

Intel 8042 简介

8042 为 AT 使用的键盘控制器，内部有一个状态缓存器，供外界检测键盘状态；一个输出埠和一个输入埠，用来做为与外界的控制讯号使用；一个输入缓冲器和一个输出缓冲器，用来存放与外界的界面资料，包括命令和资料等。以下为这些组件的意义。

1.状态缓存器

```
位 7 = 同位错误
位 6 = 接收逾时错误
位 5 = 传送逾时错误
位 4 = 键盘启用开关，0 表禁能，1 表允能
位 3 = 命令/资料指示位，0 表资料，1 表命令
位 2 = 系统旗标，由外界设定
位 1 = 输入缓冲器已满，但尚未被 8042 读走
位 0 = 输出缓冲器已满，但尚未被外界读走
```

说明：状态缓存器为一仅读缓存器，各位反应了键盘的某项状态。例如 8042 所使用的同位位为奇同位，因此如果取得的键盘资料侦测为偶同位的话，位 7 的值便会设立成 1，以便反应出同位错误。位 3 的值由线路所控制，反应出输入缓冲器内的资料型态（即命令或资料），以供 8042 决定处理的方式。在 AT 上，命令的写入是经由 I/O Port 64h，而资料的写入则经由 I/O Port 60h。因此若输入缓冲器内的资料是经前者（Port 64h）写入者，位 3 的值便会设立成 1，若是经由后者（Port 60h）写入者，则位 3 的值便会设立成 0。位元 2 的值由外界所送入的命令所控制，请参阅后面所述之命令。位

元 1 和位 0 是用来指示外界是否可对 8042 的缓冲器写入或读取资料。当位 1 为 0 时，外界才能再将资料送到 8042 的输入缓冲器；当位 0 为 1 时，外界才能从 8042 的输出缓冲器读取资料。当不是上述的情况时，外界程序便必须一直等待到上述的情况成立，否则不能对 8042 的缓冲器进行 I/O 动作。

备注：位 4 的值根据其它书籍所言为用来反应键盘抑制开关的状态，但经作者试用所有抑制键盘的命令均无法使之变成 0。猜想可能这个位值系接至输入端口位 7，但由于无法找到 8042 内部线路图，因此作者也无法确定。

2.输入埠

位 7 = 键盘抑制开关，0 为抑制，1 为开启
位 6 = 显示器型态选择，0 为彩色，1 为单色
位 5 = 0 表已装设制造时跳线
位 4 = 1 表激活系统电路板上的第二个 256KB RAM
位 3～0 = 保留

说明：输入端口的资料仅能读取，且必须透过命令方式读取。输入端口各位的信息均来自外界，其值因机器而异，因此便略而不提。位 7 接至一个 5 支脚的 Jumper 的第 4 脚，这支脚的讯号同时也控制着外界 RESET 讯号的进入，若能改变这支脚的值成为 0（也就是低电位），则不但键盘不能作用，外界 RESET 也无法重新激活计算机，必须整个关机重开，只可惜这个 Jumper 是做死的，无法利用软件来控制（作者曾在某本书中看到这个位值可以用软件改变，但该书中所写的 I/O Port 为 64h，似乎不太可能，再查阅各相关书籍及 AT 线路，已确认该书内容是错误的）。

备注：由于 8042 输入埠的值均依连接线路而定，并无法由软件控制或改变，因此大都无实用价值。作者曾在不同厂牌 PC 上读取输入埠的值（均使用彩色屏幕），结果所得到的值由 FFh、FBh、FCh、E7h 等不定，其中位 6 的值并不正确，但查阅线路图和各参考书籍均指出位 6 为接地时为彩色显示器，再由 AT BIOS 程序追踪发现显示器单彩色的判断并非由这个位来决定，因此这个输入埠的值并不是很可靠，最好不要使用。

3.输出埠

位 7 = 键盘资料线
位 6 = 键盘时脉
位 5 = 输入缓冲器已满
位 4 = 输出缓冲器已满
位 3～2 = 保留
位 1 = A20 地址线控制
位 0 = 系统重置

说明：输出端口的资料是可以读写的，但必须透过命令方式读写（注：请参阅 D1h 输出埠写入命令的说明）。键盘资料线传送的便是由 8042 送出的资料，如扫描码或响应资料等，其传送的数据结构和 RS232 相似：

第 1 位 = 开始位
第 2～9 = 资料，由低位先送
第 10 = 奇同位位
第 11 = 停止位

而键盘时脉便是用来分辨键盘资料线各位的时序的。因此这两个位的值并不固定，端视读取时的状态而定。位 1 是用来控制 CPU

的 A20 地址线输出，当其值为 0 时，将强迫 A20 地址线输出为 0，此时系统将无法读写 HMA 上的资料，也就是地址 FFFF:10h 将存取到 0000:0 地址。当其值为 1 时，A20 地址线才由 CPU 的输出决定。位 0 用来重置系统，当其值为 0 时，整个系统将会重置（也就是重新开机）。由于位 0 和位 1 关系到整个系统的运作，因此最好不要加以随意更动。

4.输出缓冲器

用来供外界读取按键的扫描码，或是键盘送出的响应数据。这个缓冲器必须在状态缓存器位 0 值为 1 时才可以读取。

5.输入缓冲器

用来供外界输入命令或资料使用，必须在状态缓存器位 1 值为 0 时才可以写入。

在 AT 上，8042 状态缓存器的读取，必须经由 I/O Port 64h，输出缓冲器的读取，则经由 I/O Port 60h。输入缓冲器则分成 60h 和 64h 两个 Port，分别用来输入资料和命令。至于 8042 的输入埠和输出埠的值，则必须籍由命令来取得或设定。送出命令给 8042 的方式，是先将命令写入 Port 64h，再依需要将资料写入 Port 60h，如此命令便算送出完毕。之后 8042 根据命令的处理结果，可能会回送一些资料回来，此时由 Port 60h 便可读回这些响应资料。以下为一些 8042 常用的命令：

20h = 读取 8042 命令字节。

60h = 写入 8042 命令字节，各位意义如下：

位 7 = 保留
位 6 = 0 表双断开码，1 窗体断开码
位 5 = 1 表不做同位检查
位 4 = 1 表抑制键盘
位 3 = 1 表不理会键盘抑制
位 2 = 系统旗标
位 1 = 保留
位 0 = 启用中断输入（IRQ1）

位 6 用来控制断开码的格式，也就是按键在放开时 8042 所送出的资料格式。当值为 0 时，表示采用双断开码格式，首先送出 F0h，然后后面跟着放开键的扫描码。当值为 1 时，则采用单断开码格式，也就是将扫描码的位 7 设成 1，用以表示这是个断开码。由于 PC 的 BIOS 程序均采用单断开码方式解译 8042 送来的资料，因此这个位大都设成 1。位 5 的值较有争议，在有些书上写明必须设成 1，做为 IBM PC 的兼容模式，但有的书上则只说明这是用来检查同位位的而已。经作者实际在多种 PC 上测试，结果这个位的初值有的为 0，有的为 1，而且不管这个位的值设成 0 或 1，系统均仍正常运作。因此作者建议还是将之设成 0，启用同位检查比较好些（原某书上标明 IBM PC 相容模式应为错误的）。位 4 用来抑制键盘，当值设成 1 时，将使得时脉线路成为低电位，使得键盘界面失效。位 3 根据各相关书籍所述为使键盘的抑制功能失效，但经作者试过，即使将这个位设成 1，各种键盘抑制命令仍然会抑制键盘的功能，因此它的实际用途不明。位 2 用来设定状态缓存器位 2，也就是系统旗标的值，这个旗标并没什么作用，一般在 PC 均设成 1。位 0 用来控制是否采用中断输入，一般均设成 1。若将之

设成 0，则必须使用 Poll 方式处理，较为麻烦。位 7 和位 1 为保留值，可任意写入 0 或 1。

AAh = 自我测试，OK 时输出 55h。注意这个命令会重设 8042 命令位组的值，因此在命令之后应重设 8042 命令字节的值，否则键盘功能将会被抑制而失去作用。

ABh = 接口测试，传回结果：
- 00 = 正常
- 01 = 键盘时脉保持在低电位
- 02 = 键盘时脉保持在高电位
- 03 = 键盘资料线保持在低电位
- 04 = 键盘资料线保持在高电位

ACh = 诊断倾印（diagonstic dump），将 16 字节的控制器内存、输入埠目前状态、输出埠目前状态、以及控制器的程序状态字组送至系统。事实上，经作者在各种 PC 上测试的结果发现，实际送出的资料差异性很大，有的完全不送资料，有的只送一个 byte，而有的一送便是十几个 byte。其资料的意义找遍各书籍均未说明，因此此部份的说明也只好省略了。

ADh = 键盘失效（即将 8042 命令字节之位 4 设成 1）。

AEh = 启用键盘（即将 8042 命令字节之位 4 设成 0）。

C0h = 读取输入埠，必须输出缓冲器有空的时才可使用。

D0h = 读取输出埠，必须输出缓冲器有空的时才可使用。

D1h = 写入输出埠。但是这个命令经作者实际测试过，并不会将给定的资料写入输出端口。因此若而改变输出埠的输出值，应使用后面会提到的 Fxh 命令，但这个命令只能改变输出端口位 3～0 的值。

E0h = 读取测试输入端口，各位值意义如下：
- 位 7～2 = 固定为 0
- 位 1 = 键盘资料线
- 位 0 = 键盘时脉

其中位 0～1 系由输出端口位 7～6 所回送的输入资料，以做为键盘自我测试时使用。同样的，这个命令所读出的值也因输出埠的状态而异，请参阅输出埠中的说明。

Fxh = 脉波输出，x 中 4 个位分别对应到输出端口的位 3～0，命令中被选到的位（以 0 选定）会有 6us 的低电位脉冲（也就是逻辑值 0）。请参阅输出埠中的说明。

以下则为 8042 的系统命令。系统命令和一般命令的送出方式不同，必须经由资料端口，也就是 I/O Port 60h 送出。8042 后收到后，除了 EEh 回音命令以外，均会回送 ACK 讯号 FAh（收到命令或资料均会回送）。以下为各系统命令的意义：

EDh = 设定模式指示器，输入值意义为：
- 位 7～3 = 固定为 0
- 位 2 = Caps Lock 指示器
- 位 1 = Numeric Lock 指示器
- 位 0 = Scroll Lock 指示器

设定之后，键盘上对应的 LED 便会亮起来。

EEh = 回音，键盘以 EEh 响应。

F3h = 设定键重复率及延迟，送入值意义为：
- 位 7 = 固定为 0。
- 位 6～5 = 延迟参数，延迟(1+x)*250ms。

位 4～0 = 重复参数，假设位 2～0 值为 A，位 4～3 值为 B，则重复周期为(8+A)*2^B*4.17ms。

F4h = 系统启用，并清除输出缓冲器。

F5h = 使用键盘默认值、清除输出缓冲器，最后将系统失效，等待进一步指令。若要再启用必须使用系统命令 F4h。

F6h = 使用键盘默认值设定重复率及延迟率，并清除输出缓冲器。

FEh = 重送命令，键盘回送上次的输出。

FFh = 重置命令，键盘接收后清除输出缓冲器，并使用默认值，最后开始自我测试，完毕回送 AAh 表正常，其它值表有误（注意，此命令会先送出 ACK 后再送出 AAh）。

EFh～F2h、F7h～FDh = 保留命令，不运作（NOP）。但事实上经作者测试之后，发现其中有些命令在某些机器上似乎有作用，有些会使键盘失效，有些则会回送一些额外的值。由于这部份的资讯无法确定，因此只能建议读者不要随便使用这些保留命令。

以下为一些键盘可能回送的资料或命令：

00h = 覆盖（Overrun），当输出缓冲器已满而且又有输出时，便会送出此一资料。

AAh = 基本测试完成码。

EEh = 回音响应。

F0h = 断开码第一字节，用来指示某个键已被放开。断开码由两个字节组成，第一字节便是 F0h，第二字节则为键盘扫描码。

FAh = ACK 讯号，用以响应系统命令使用。

FEh = 重送命令，当键盘接到一个无效或错误的输入时，均会发出此命令要求外界重送。

FDh = 诊断错误，当键盘周性自我诊断时发现错误，便发出此资料。

注意在对 8042 进行命令控制时，必须将中断禁能，以免外界键盘输入中断影响正常之键盘命令传送。

附件 13. “两支老虎跑得快“中断驻留程序

（本程序可在 Windows 中运行）

```
;(C) WAN JINYOU , 1992
;Function: the two tigers will sing every 60 seconds at back end.
;Sing Proc 和 Two_tiger Proc 两个过程为唱歌部分。
;请参考“附件 9、Intel 8253 简介”和“附件 10、Intel 8255 简介”。

INT_VECTOR=1Ch
TIME = 2*545 ; 60 second.  545 * 55ms = 30,000 ms == 30 seconds

code segment
 assume cs:code,ds:code
ORG 100h      ;this line could be deleted for EXE, only for COM
START:
    JMP INSTALL
 my_int proc near
                ;       push ax
                ;       pushf
                ;       call dword ptr cs:save_int_vector
                ;       pop ax
                ;       iret
       dec  cs:count
       jz @ok
       jmp dword ptr cs:save_int_vector
@ok:
       mov cs:count,TIME
       push ax
       push bx
       push cx
       push dx
       push si
       push di
       push ds
       push es
      call sing

       pop es
       pop ds
       pop di
       pop si
       pop dx
       pop cx
       pop bx
       pop ax
exit:
   jmp dword ptr cs:save_int_vector
```

```
save_int_vector dd 0
count dw TIME
sing proc near
lea si,cs:freq
lea bp,cs:times
repeat:
    mov di,cs:[si]
    cmp di,0
    je @sing_end
    add di,di  ; these two instructions could be deleted
    add di,di  ;

    mov bx,cs:[bp]
    call two_tiger
    inc si
    inc si
    inc bp
    inc bp
    jmp repeat
@sing_end:
    ret
sing endp

two_tiger proc near
      mov al,0b6h ;set Intel 8253 to counter 3 mode
      out 43h,al
      mov dx,12h  ;1234dch=1193180 is the input external frequency of 8253
      mov ax,34dch
      div di            ;di=The required song tone frequncy
out 42h,al  ;write to counter 3, use the quotient AX as the counter value
      mov al,ah
      out 42h,al
      in al,61h
      mov ah,al   ;save the original value of 61h
      or al,3           ;Set bit 0, 1 of Intel 8255 to turn on the speaker
      out 61h,al

      SLOW EQU 800     ; SLOW=delay factor,test song speed by yourself
                      ; for 8086 SLOW=1, for P4/1.8G CPU SLOW=800.

delay0:
      mov cx,SLOW ;external loop
delay1:
```

```
      push cx
      mov cx,2801h      ;inner loop
dl10ms:
      loop dl10ms ;delay for 10 ms
      pop cx
      loop delay1
      dec bx
      jnz delay0  ;total delay loops of times*SLOW*2801h

      mov al,ah   ;restore the original value of 61h
      out 61h,al
      ret

;;;;;;;;;;;;;;;;Two Tigers;;;;;;;;;;;;;
      freq dw 2 dup (262,294,330,262)
           dw 2 dup (330,349,392)
           dw 2 dup (392,440,392,349,330,262)
           dw 2 dup (294,196,262),0
      times dw 10 dup (25),50,25,25,50
           dw 2 dup (12,12,12,12,25,25)
           dw 2 dup (25,25,50)

;;;;;;;;;;;;;;;;Boots on Taihu Lake;;;;;;;;;;;;;
;      freq dw 330,392,330,294,330,392,330,294,330
;           dw 330,392,330,294,262,294,330,392,294
;           dw 262,262,220,196,196,220,262,294,330,262,0
;     times 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

two_tiger endp
my_int endp

INSTALL proc near
      call save_int
      call set_int
      mov ax,3100h
      mov dx,offset INSTALL
      add dx,100h ;must include PSP of EXE, as CS!=PSP for EXE
                ;but this line could be deleted for COM
      mov cl,4
      shr dx,cl       ;dx/16: convert byte to paragraph
      inc dx           ;consideration of the remainder
      int 21h
```

```
                ;mov dx,offset install
                ;int 27h
INSTALL endp
save_int proc near
             xor ax,ax
             mov ds,ax
             mov si,INT_VECTOR*4
             mov bx,[si]
             mov es,[si+2]
             mov word ptr cs:save_int_vector,bx
             mov word ptr cs:save_int_vector+2,es
             ret

           ;  mov ah,35h
           ;  mov al,INT_VECTOR
           ;  int 21h
           ;  mov word ptr cs:save_int_vector,bx
           ;  mov word ptr cs:save_int_vector+2,es
           ;  ret
   save_int endp
   set_int proc near
             xor ax,ax
             mov es,ax
             mov ax,offset my_int
             mov es:[INT_VECTOR*4],ax
             mov es:[INT_VECTOR*4+2],cs
             ret

                          ;           push cs
                          ;           pop ds
                          ;           mov dx,offset my_int
                          ;           mov ah,25h
                          ;           mov al,INT_VECTOR
                          ;           int 21h
                          ;           ret
   set_int endp
code ends
      end start
```

附件 14. 演示热键激活的中断驻留程序

（本程序可在 Windows 中运行）

```
;(C) WAN JINYOU, 1992
;Function：Whenever Shift+Alt+2 is pressed, dispay "OK" at cursor (12,40)
INT_VECTOR=9h
cseg segment
     assume cs:cseg,ds:cseg
     org 100h     ;this line could be deleted for EXE, only for COM
 begin:
       jmp install
 kbsave dd ?
newint9 proc near
      sti         ; 开中断, enable high level IRQ interrupt
      push ax
      push bx
      push cx
      push dx
      push si
      push di
      push ds
      push es
      pushf
      call dword ptr cs:kbsave  ;EXCUTE OLD INT 9
      mov ax,0040h
      mov ds,ax
      test byte ptr ds:[17h],00000011B
                  ;BIOS parameter at 0040:0017h is kbflag
                  ;Test if L_shift Or R_shift is pressed
      jz exit
      mov ah,1      ;Test whether KB buffer is empty
      int 16h
      jz exit   ;KB buffer is empty
      cmp al,0
      jnz exit  ;not function key
      cmp ah,79h  ;Alt+2
      jne exit
      mov ah,0    ;Kick out hot key from KB buffer
      int 16h
      mov ah,2
      mov dh,12   ;set cursor to (12,40)
      mov dl,40
      mov bh,0    ;page
      int 10h
```

```
      mov ah,0Eh   ;beep
      mov al,07h
      int 10h
      mov al,'O'
      int 10h
      mov al,'K'
      int 10h
exit:
      pop es
      pop ds
      pop di
      pop si
      pop dx
      pop cx
      pop bx
      pop ax
      iret
newint9 endp

install proc near
    xor ax,ax
    mov es,ax
    mov ax,word ptr es:[INT_VECTOR*4]   ;save old int 9 vector
    mov word ptr cs:kbsave,ax           ;also can use int 21h(ah=35h)
    mov ax,word ptr es:[INT_VECTOR*4+2]
    mov word ptr cs:kbsave+2,ax

    xor ax,ax
    mov es,ax
    mov ax,offset newint9               ;set new int 9 vector
    mov word ptr es:[INT_VECTOR*4],ax   ;also can use int 21h(ah=25h)
    mov word ptr es:[INT_VECTOR*4+2],cs

    MOV AX,3100H                        ;
    LEA DX,install
    ADD DX,100h   ;must include PSP for EXE, as PSP!=CS for EXE
                  ;but this line could be deleted for COM
    MOV CL,4
    SHR DX,CL     ;convert byte to paragraph
    INC  DX       ;consideration of the remainder
    INT 21h
        ;mov DX,OFFSET INSTALL
        ;INT 27H
install endp
```

```
cseg ends
     end begin
```

附件 15.设置 DOS 版本号的中断驻留程序

(注：Windows 不允许修改 INT 21H，故本程序只能在 DOS 下运行)

```
;(C) WAN JINYOU, 1995
;Function: Set the current DOS version to any version you want
;So that some old programs still could run on any version of DOS.
; Usage:setdosve 6.20
;            setdosve 3.31
      ;Use the VER command to display the current DOS version.

INT_VECTOR=21H
DISP MACRO MES
     MOV AX,CS
     MOV DS,AX
     MOV AH,9
     MOV DX,OFFSET MES
     INT 21H
ENDM
CODE SEGMENT
   ASSUME CS:CODE,DS:CODE
    ORG 100H      ; this line could be deleted for EXE
START:
         JMP INSTALL
NEWINT21 PROC
         CMP AX,0F1F1H
         JNZ L
         MOV AX,CS
         MOV ES,AX
         MOV BX,OFFSET NEWVER
         MOV AX,1F1FH
         IRET
     L:
         CMP AH,30H
         JNZ EXIT
         MOV AX,CS:NEWVER
         IRET
     EXIT:
         JMP DWORD PTR CS:OLDINT21
     OLDINT21 DD ?
     NEWVER DW ?
NEWINT21 ENDP
INSTALL PROC
 XOR CX,CX        ;DS=ES=Segment of PSP
         MOV CL,ES:[80h]      ;char number of command line
```

```
        MOV DI,81H
        MOV AL,20h
        CLD
        REP SCASB            ;Skip the initial space
        DEC DI               ;let DI point to the first non-space char

;     MOV DI,80H
;SPACE:
;     INC DI
;     CMP BYTE PTR ES:[DI],20H
;     JZ SPACE


        CMP BYTE PTR ES:[DI+4],0DH
        JZ OK
        DISP MES1
        MOV AX,4C01H
        INT 21H
    OK: MOV AL,ES:[DI]    ;major VERSION
        AND AL,0FH
        MOV BYTE PTR NEWVER,AL

        MOV AL,ES:[DI+2]    ;minor VERSION
        AND AL,0FH
        MOV CL,10D
        MUL CL
        MOV CL,ES:[DI+3]
        AND CL,0FH
        ADD AL,CL
        MOV BYTE PTR NEWVER+1,AL

        MOV AX,0F1F1H
        INT 21H
        CMP AX,1F1FH
        JNZ INST
        MOV AX,NEWVER
        MOV ES:[BX],AX
        DISP MES3
        MOV AX,4C00H
        INT 21H
INST:
        XOR AX,AX
        MOV DS,AX
        MOV BX,DS:[INT_VECTOR*4]
```

```
        MOV ES,DS:[INT_VECTOR*4+2]
        MOV WORD PTR CS:OLDINT21,BX
        MOV WORD PTR CS:OLDINT21+2,ES

        MOV WORD PTR DS:[INT_VECTOR*4],OFFSET NEWINT21
        MOV DS:[INT_VECTOR*4+2],CS

        DISP MES2
        MOV AX,3100H
        MOV DX,OFFSET INSTALL
        ADD DX,100H ;must include 100h for PSP, as CS!=PSP for EXE
                    ;but this line could be deleeted for COM
        MOV CL,4
        SHR DX,CL       ;convert byte to paragraph
        INC DX          ;consideration of the remainder
        INT 21H
    MES1 DB 'Set DOS version utility 1.00',0DH,0AH
         DB 'Usage example:SETDOSVER 6.20',0DH,0AH
         DB '               SETDOSVER 3.31',0DH,0AH
         DB 'Copywrite (C) Wan Jinyou,Tongji
University,1995.1',0DH,0AH,0DH,0AH,'$'
    MES2 DB 'OK,set successful !',0DH,0AH,'$'
    MES3 DB 'OK,reset successful !',0DH,0AH,'$'
INSTALL ENDP
CODE ENDS
END START
```

附件 16. C 语言程序生成的汇编程序

TEST.C 源程序：

```
001:   int g1=3;
002:   int g2;
003:
004:   main()
005:   {
006:          int i,j,k;
007:          char *s="the reruslt is";  /*注意：在栈中，s 在上，k 在下*/
008:          g1=-1;
009:          g2=-2;
010:          i=2;
011:          j=3;
012:          k=add(i,j);
013:          printf("%s %d\n",s,k);
014:   }
015:
016:   int add(int i,int j)
017:   {
018:          int var1=i,var2=j,var3;
019:          var3=var1+var2;
020:          return var3;
021:   }
```

_main 中的栈顶框架如下：

```
(mov bp,sp)
bp-2->offset s （局部变量 s 的偏移）
bp-4-> k （局部变量 k）
```

进入_add 后，栈顶框架如下：

```
bp+6->3        （参数 j）
bp+4->2        （参数 i）
bp+2->ip       （返回的偏移）
bp+0->bp       （push bp, mov bp,sp）
bp-2->var3     （局部变量）
```

用 C>tcc -S -ms -c -v TEST.C 命令产生的汇编程序 TEST.ASM：

```
       ifndef ??version
?debug macro
       endm
       endif
       ?debug S "test.c"
_TEXT  segment       byte public 'CODE'
DGROUP group  _DATA,_BSS
       assume cs:_TEXT,ds:DGROUP,ss:DGROUP
_TEXT  ends
_DATA  segment word public 'DATA'
d@     label  byte
```

```
d@w     label  word
_DATA   ends
_BSS    segment word public 'BSS'
b@      label  byte
b@w     label  word
        ?debug C E9098EB12E06746573742E63
_BSS    ends
_DATA   segment word public 'DATA'
_g1     label  word
        dw     3
_DATA   ends
_TEXT   segment       byte public 'CODE'
;       ?debug L 4
_main   proc   near
        push   bp
        mov    bp,sp
        sub    sp,4
        push   si
        push   di
;       ?debug L 7
        mov    word ptr [bp-2],offset DGROUP:s@
;       ?debug L 8
        mov    word ptr DGROUP:_g1,-1
;       ?debug L 9
        mov    word ptr DGROUP:_g2,-2
;       ?debug L 10
        mov    si,2
;       ?debug L 11
        mov    di,3
;       ?debug L 12
        push   di
        push   si
        call   near ptr _add
        pop    cx
        pop    cx
        mov    word ptr [bp-4],ax
;       ?debug L 13
        push   word ptr [bp-4]
        push   word ptr [bp-2]
        mov    ax,offset DGROUP:s@+15
        push   ax
        call   near ptr _printf
        add    sp,6
@1:
;       ?debug L 14
        pop    di
        pop    si
        mov    sp,bp
        pop    bp
        ret
_main   endp
```

```
;       ?debug L 16
_add    proc    near
        push    bp
        mov     bp,sp
        sub     sp,2
        push    si
        push    di
;       ?debug L 18
        mov     si,word ptr [bp+4]
;       ?debug L 18
        mov     di,word ptr [bp+6]
;       ?debug L 19
        mov     ax,si
        add     ax,di
        mov     word ptr [bp-2],ax
;       ?debug L 20
        mov     ax,word ptr [bp-2]
        jmp     short @2
@2:
;       ?debug L 21
        pop     di
        pop     si
        mov     sp,bp
        pop     bp
        ret
_add    endp
_TEXT   ends
_BSS    segment word public 'BSS'
_g2     label   word
        db      2 dup (?)
_BSS    ends
        ?debug C E9
_DATA   segment word public 'DATA'
s@      label   byte
        db      116
        db      104
        db      101
        db      32
        db      114
        db      101
        db      114
        db      117
        db      115
        db      108
        db      116
        db      32
        db      105
        db      115
        db      0
        db      37
        db      115
```

```
        db      32
        db      37
        db      100
        db      10
        db      0
_DATA   ends
_TEXT   segment       byte public 'CODE'
        extrn  _printf:near
_TEXT   ends
        public _main
        public _add
        public _g2
        public _g1
        end
```

附录 17. ASCII 表

ASCII 非打印控制字符（0～31） ASCII 打印字符（32～127） ASCII 扩展打印字符（128～255）

十进制	16进制	字符	十进制	16进制	字符	十进制	16进制	字符	十进制	16进制	字符	十进制	16进制	字符	十进制	16进制	字符	十进制	16进制	字符
0	0	空	32	20	space	64	40	@	96	60	`	128	80	Ç	160	A0	á	192	C0	└	224	E0	α
1	1	头标开始	33	21	!	65	41	A	97	61	a	129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
2	2	正文开始	34	22	″	66	42	B	98	62	b	130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
3	3	正文结束	35	23	#	67	43	C	99	63	c	131	83	â	163	A3	ú	195	C3	├	227	E3	π
4	4	传输结束	36	24	$	68	44	D	100	64	d	132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
5	5	查询	37	25	%	69	45	E	101	65	e	133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
6	6	确认	38	26	&	70	46	F	102	66	f	134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
7	7	震铃	39	27	’	71	47	G	103	67	g	135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
8	8	backspace	40	28	(	72	48	H	104	68	h	136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
9	9	水平制表符	41	29	)	73	49	I	105	69	i	137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
10	A	换行/新行	42	2A	*	74	4A	J	106	6A	j	138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
11	B	竖直制表符	43	2B	+	75	4B	K	107	6B	k	139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
12	C	换页/新页	44	2C	,	76	4C	L	108	6C	l	140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
13	D	回车	45	2D	-	77	4D	M	109	6D	m	141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
14	E	移出	46	2E	.	78	4E	N	110	6E	n	142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
15	F	移入	47	2F	/	79	4F	O	111	6F	o	143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
16	10	数据链路转义	48	30	0	80	50	P	112	70	p	144	90	É	176	B0	░	208	D0	╨	240	F0	≡
17	11	设备控制 1	49	31	1	81	51	Q	113	71	q	145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
18	12	设备控制 2	50	32	2	82	52	R	114	72	r	146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
19	13	设备控制 3	51	33	3	83	53	S	115	73	s	147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
20	14	设备控制 4	52	34	4	84	54	T	116	74	t	148	94	ö	180	B4	┤	212	D4	Ô	244	F4	⌠
21	15	反确认	53	35	5	85	55	U	117	75	u	149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
22	16	同步空闲	54	36	6	86	56	V	118	76	v	150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
23	17	传输块结束	55	37	7	87	57	w	119	77	w	151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
24	18	取消	56	38	8	88	58	X	120	78	x	152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	≈
25	19	媒体结束	57	39	9	89	59	Y	121	79	y	153	99	Ö	185	B9	╣	217	D9	┘	249	F9	·
26	1A	替换	58	3A	:	90	5A	Z	122	7A	z	154	9A	Ü	186	BA	║	218	DA	┌	250	FA	•
27	1B	转意	59	3B	;	91	5B	[	123	7B	{	155	9B	¢	187	BB	╗	219	DB	■	251	FB	√
28	1C	文件分隔符	60	3C	<	92	5C	\	124	7C	|	156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
29	1D	组分隔符	61	3D	=	93	5D	]	125	7D	}	157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
30	1E	记录分隔符	62	3E	>	94	5E	^	126	7E	~	158	9E	Pts	190	BE	╛	222	DE	▐	254	FE	■
31	1F	单元分隔符	63	3F	?	95	5F	_	127	7F	DEL	159	9F	*f*	191	BF	┐	223	DF	▀	255	FF	

附录 18.汇编程序出错信息

序号	显 示	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，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	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	PUBLEC 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 references is illegal	向前引用必须是在第一遍扫视中定义过的
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的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:":")
39	One operand must be const	一个操作数必须是常数
40	Operands must be same or labs	操作符必须相同或是编号
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 code	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).
54	Index displ. must be cinstant	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	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.
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 定义太大
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 forward ref.	
83	Circular chain of EQU aliases.	
84	Feature not supported by Small Assembler(ASM)	

FAQ、同学提问集锦

以下是一些同学发 EMAIL 来问的一些典型问题，我将回答整理如下，供参考：

一、关于 32 位汇编的问题

(1)在原版教材中,有些例子将 8086 指令和 386 指令(386 以后 CPU 新增加的指令)混在一起使用,例如 PUSHA,MOVSX 等 386 指令.对这种程序,必须在程序中写汇编指示.386,告诉汇编器，我们要使用和生成 386 指令，否则汇编时会报告其为非法指令。

```
.model small
.386
...
```

注意，.386 请写在.model small 之后一行，否则汇编器会只使用 32 位段格式，并将所有的寄存器和指令全转换成 32 位，程序可能会乱。在 32 位模式下，只有一种内存模型，即.model flat（平坦模型）。在 386 模式下，内存不再有段地址和偏移量之分，而是代之以 32 位线性地址空间。所谓 tiny/small/mediu,/...等内存模型问题只有在 16 位方式下才存在。以下是 Microsoft 的一段解释：

If the .MODEL directive is preceded by the .386 or .486 directive(or their privileged counterparts), the assembler uses 32-bit segments. If the .386 or .486 directive follows the .MODEL directive, the default 16-bit segments are used. The flat memory model uses 32-bit segments and must be preceded by a .386 or .486 directive.32-bit segments are available only on the 80386/486 processors.

更详细地请见\\10.40.55.59\public files\Documents & Course Materials (教学文档与学习资料)\Assembly Programming (万金友)\tools\Help of Masm611\LANG.HLP 和 ML.HLP 文件。

另外，在\\10.40.55.59\public files\Documents & Course Materials (教学文档与学习资料)\Assembly Programming (万金友)\tools 中有 Microsoft 公司的最后版汇编器 MASM611.ZIP，可使用之。

(2)对包含 386 指令的 EXE 程序，不能用 DEBUG 调试（DEBUG 只能使用和识别 8086 指令），可使用 Borland 公司的 Turbo Debugger(TD.EXE)或 Microsoft 公司的 CodeView(CV.EXE)软件调试，TD 和 CV 均是 DEBUG 的升级版。TD 和 CV 均已放在我们的 tools 目录中，其中 CV 包含在 MASM611 软件包中。

(3)在 32 位 Windows 中的最强大 DEBUG 软件是美国 Numega 公司的 SOFTICE，非常有名，特别是在编写调试设备驱动程序时，离不开 SOFTICE。在 Windows 中，ICE 安装后，在任何时刻我们可按 CTRL+D 键将 ICE 激活，适合动态分析系统。reference 目录中有其用法，该软件以后我会放在服务器上。

(4)关于 386 保护方式下的汇编，reference 目录中有一位很好的文章“386 保护模式简介”可看看。关于 Win32 汇编，可阅读该目录中的文件“Win32 汇编语言讲座.txt”。Win32 汇编专指用汇编语言编写 Windows 应用程序，而 386 汇编则是指针对 386 以上 CPU 的编程（与 Windows 无关）。另外，请参考 Microsoft 中 MASM611 中随带的帮助文档。

```
>Subject: 怎么不能用 EAX,EBX,ECX,EDX 等寄存器
>Date: Sat, 24 May 2003 10:13:47 -0000
>万老师：
>        怎么不能用 EAX,EBX,ECX,EDX 等寄存器
```

答：请在 ASM 程序中写汇编指示.386,注意请写在.model small 行后,.code 行前,例如 ：

```
.model small
.386
.code
```

```
mov eax,80h
push ebx
...
```

汇编连接后,请用 Turbo Debugger(TD.EXE)或 CodeView(CV.EXE)进行调试,DEBUG 不能识别 32 位寄存器及指令. 在 TD 或 CV 中可直接使用 386 指令和 32 位寄存器.

二、其它问题

(1)ASM 程序中的 MOV AX,[82H]汇编后,被翻译成 MOV AX,82H,内存操作数被译成了立即数(在 DEBUG 中不会出现这样现象).这是汇编器的问题,因为在 ASM 程序中直接在指令中给出内存操作数地址的用法(即立即寻址)是很罕见的,汇编器误以为这是你写错了,所以转换成了立即数.请将 MOV AX,[82H]写成 MOV AX,DS:[82H],汇编器就不会进行这种转换.

(2)在 DEBUG 中可以写 JMP FFFF:0 这样的绝对跳转指令，但在 ASM 程序中不允许这样写。在 ASM 程序中要实现程序的绝对跳转，可以两种方式实现：

方式一、

```
.DATA
BOOTENTRY DB 0FFFF0000H
.CODE
MOV AX,@DATA
MOV DS,AX
JMP DWORD PTR BOOTENTRY
```

方式二、

```
.CODE
MOV AX,0FFFFH
PUSH AX
MOV AX,0
PUSH AX
RETF
```

因为 RETF 将栈一个字弹到 IP 中，然后再弹一个字到 CS 中，所以 RETF 执行后 CS:IP=FFFF:0000。

(3)若想在 Turbo Debugger 中能得到汇编源程序与生成指令的完整对照（即使生成的 EXE 中包含 DEBUG 信息），必须在 TASM 汇编时带/Zi 开关，在 TLINK 时带/v 开关。

(4)很多同学问“显示命令行指定文件内容”作业，在 DEBUG 中可正常运行，但在命令行方式下不能打开文件。

现解释如下：命令行参数位于程序段前缀 PSP 偏移量 80H 处开始的地方，其中，PSP:80H 处存放的是命令行字符个数（不含回车 0DH），PSP:81H 开始存放的是具体的命令行字符（以 0DH）结束。

例如，假设程序 READFILE 带参数 ABC 运行，即 C>READFILE ABC

则 PSP:80H 处的内容应是 04 20 41 42 43 0D，在纯 DOS 环境中，以及 Win2K 的 DEBUG 环境中，空格均不会被去掉，并且计入命令行字符个数中（就像我们上课讲的那样）。

但在 Win2K 内部所提供的命令行提示符下，命令行字符的所有前导空格均会被自动删除，即 PSP:80H 处的内容是 03 41 42 43 0D。有些同学在 Win2K 内的命令行中，若以 C>READFILE ABC 运行不能打开文件，但在 DEBUG 中却能打开文件。其原因如上。

其实，设计好的程序，不应直接到 PSP:82H 处取字符，而应该能自动跳过前导空格，这样设计的程序就不会出现以上那种在不同环境中运行结果不一样的情况。如下，是很好的代码：

```
.CODE
        XOR CX,CX           ;DS=ES=Segment of PSP
```

```
                MOV CL,ES:[80h]
                MOV DI,81H
                MOV AL,20H
                CLD
                REP SCASB           ;Skip the initial space
                DEC DI                      ;let DI point to the first non-space char
```

其中 REP SCASB 是在 ES:DI 中自动搜索等于 AL 的字符，最多搜索 CX 次，当找到时便停止。

由于 SCASB 每搜索一个字符都会自动将 DI 加 1，所以最后应将 DI 倒回去 1 个字符。

不用 SCASB 的代码如下：

```
.CODE
                MOV DI,80H
SPACE:            INC DI
           CMP BYTE PTR ES:[DI],20H
           JZ SPACE
```

以上执行完后 DI 均指向首个非空字符。

另外注意，最好按 ALT+ENTER 键将命令行窗口切换成全屏幕方式，否则有些信息可能不能正常显示。

学汇编语言，最讨厌的就是 Windows，在 Windows 环境中，不仅很多汇编指令不让执行，而且经常变来变去。最好用一张能启动的 WIN98 软盘启动，在纯 DOS 下进行，保证什么都能做，而且不会有怪现象出现。

(5) 这个程序似乎不起作用（我在命令提示行下试的），软驱有反应，但却不执行 format。why?

By >the way, 硬盘格式化我没试过，能行吗？全部格掉还是只格系统盘？

```
// 快速格式化软盘
L 100 0 0 * 插入一张已格式化软盘
W 100 0 0 * 放入一张欲格式化软盘
注：* 分别为:720K e |1.2M id |1.44M 21
答：    L/W 内存起始地址    盘号    起始逻辑扇区号      扇区个数
```

其中，盘号 0=A,1=B,2=C,3=D,4=E,5=F,6=G...,

L/W 分别对应 INT 26H 和 INT 25H，是在逻辑扇区一级的读写（较低级，但仍属操作系统级），最低级的是 INT 13H(BIOS 级)，INT 13H 使用的是物理扇区号（道号、磁头号、扇区号）。在讲义上“主要 BIOS 中断”一节中讲了其调用关系。INT 21H 使用的是文件名，在 OS 内部要转换成逻辑扇区号读写（INT 25/26H），逻辑扇区号最后还要转换成对物理扇区的读写（INT 13H），才能被驱动器硬件所接受。逻辑扇区号只有一个数，每个逻辑盘的逻辑扇区号均从 0 开始顺序编号至最后一个扇区，如 A、C、D、E、F、G 等盘每个盘的逻辑扇区号均从 0 开始，所以用 W 格式化（即 INT 25H）只能格式化一个逻辑盘（即用 FDISK 分区时划出的每个盘），不能格式化整个物理硬盘。要格式化整个物理硬盘，必须用 INT 13H（称为低级格式化）。不过我们 9 楼机器安装了硬盘保护卡，能俘获直接的 INT 13H 调用，所以低级格式化也无效。

以上操作格式化软盘当然有效，问题是您的目标软盘必须先前已用 FORMAT 经格式化过的软盘，所谓快速格式化只对已格式化过的磁盘才能使用，不能用于新盘。未格式化过的磁盘无扇区地址标记，不能用 W 命令格式化。其实上面的用 W 命令格式化软盘，只是将源盘的系统区，包括引导记录（占 1 扇区）、FAT 表（每个 FAT 表占 9 扇区*2 个 FAT 表）和根目录表（占 14 个扇区）这三部分取下来（以上均以 1.44M 软盘为例），然后原封不动地写入到目标软盘中。所以实际上是将目标盘的系统区快速度清空，但扇区标记还是以前的。若源盘根目录上有文件名，目标盘也将存在完全相同的文件名。当然由于文件中包含的数据（位于磁盘数据区）并没有复制到目标盘中，所以目标盘上的文件并不能使用。

另外，在 Windows NT/2K/XP 中，只能对软盘操作使用 INT 25H（W）/26H（L）/13H，对硬盘无效。因为这些中断都会越过 Windows 操作系统对硬盘文件系统的管理，所以会被 Windows 俘获并

禁止。若要对硬盘做实验，必须不进入 Windows，在纯 DOS 下做实验，当然这是极其危险的操作，稍有不慎，硬盘数据全丢。只应在无可奈何的情况下才使用。

（6）INT 1 称为单步中断，INT 3 称为断点中断。其中 INT 3 对调试程序很有用，当 DEBUG 执行到 INT 3 时，便会暂停，INT 3 专用在 DEBUG 中设置断点用。可在程序任何需要暂停的地方插入 INT 3，其好处是不必知道断点地址，就可在想停的地方停止。例如：

```
-A100
MOV AH,2
MOV DL,41H
INT 21H
INT 3
-G
```

会停在 INT 3 处。

(7)Assembling programming 似乎更像一种过程化的语言，有滥用 Jx 之嫌。其结构化不是很明显，am I right?

答：maybe u r right. 但汇编语言是面向机器的，不同于任何高级语言，汇编主要是用来讲计算机工作原理和进行低级控制的，不会用来开发大型应用软件，所以汇编语言较少考虑程序的结构化问题。那是高级语言的任务。但任何高级语言不管结构化设计得如何好，编译生成机器码后，还是全靠 Jx 之类指令来实现的，因为 CPU 只能识别和执行 Jx 之类指令。例如高级语言 IF X>2 THEN Y=1，编译后生成的机器码是如下形式：

```
CMP [X], 2
JG L
MOV [Y], 1
L:
```

食品包装越来越漂亮，但吃到肚里还是变成了蛋白质。学汇编就是在直接吃蛋白质。您可用 Turbo C 或 Borland C 强行生成汇编码：

C>TCC(或 BCC) -S TEST.C 也可在菜单中设置

查看生成的 TEST.ASM，原型毕露。或用 CODEVIEW 查看 C 生成的汇编码也可。

(8) 怎样在 VC++中设置命令行参数？请在菜单 Project/settings/Debug 中设置 Working Directory 为您要打开的文件所在的目录(例如 d:\wan），在 Program arguments 中填要打开的文件名(例如 input.txt)。或者只在 Program arguments 中填 d:\wan\input.txt 也可。Program arguments 是专门在环境中设置命令行参数的。

(9)不要用 C 语言的思想理解汇编语言的过程

```
start proc near
       ...
       ...
       mov ax,4cxxh
       int 21h
start endp
sub proc
       ...
       ...
       ret
sub proc
```

以上程序，若 start proc 中未写 mov ax,4c00h/int 21h，则 CPU 会继续执行，会冲到 sub proc 中继续执行，直到死机为止。这与 C 语言不同，C 语言中的每个函数都会被系统自动加 return 指令,main（）过程还会被加入 mov ax,4cxxh/int 21h 之类的指令。所以 C 的函数不会跑到下一个相邻函数中执行。

（10）关于程序调试问题
分号；后面的内容是注释，可不必输入。汇编程序中只能使用英文的逗号和分号。我建议您先自己编一些小程序，等知道错误处理办法后，再试大程序。
以下是您程序汇编时出的错，这些错误都是很简单的错误。可找一个能显示行号的编辑软件如EDIT 或 WORD 等，以便根据出错信息找出对应行的错误。
Error CHENYING.ASM(19) Undefined symbol: B10INPUT
第 19 行：B10INPUT 未定义，您后面程序中定义的过程名为 B10INPU PROC NEAR ，名字写错了。
Warning CHENYING.ASM(25) Reserved word used as symbol: NEAR
第 25 行：使用了保留字。 该行应去掉。
Error CHENYING.ASM(43) Unmatched ENDP: A10MAIN
第 43 行：无与 ENDP 匹配的过程定义，由第 25 行错误引起的。
Error CHENYING.ASM(52) Too few operands to instruction
第 52 行：指令中的操作符太少。您不应使用中文（全角）逗号“，”，请改为英文（半角）的“,”。
Error CHENYING.ASM(53) Too few operands to instruction
同 52 行
Error CHENYING.ASM(60) Unknown character
同 52 行，注释应以英文分号;开始，不是中文的分号
Warning CHENYING.ASM(61) Reserved word used as symbol: AH
Error CHENYING.ASM(61) Extra characters on line
Error CHENYING.ASM(62) Too few operands to instruction
Error CHENYING.ASM(63) Too few operands to instruction
Error CHENYING.ASM(68) Unknown character
以上出错原因均相同，不该用中文分号；
Error CHENYING.ASM(70) Illegal instruction for currently selected processor(s)
第 70 行，对当前处理器使用了非法指令。原因：PUSHA 指令只能在 386 以上 CPU 使用，请在.code 一行前加一行.386，告诉汇编器我们使用了指令。
Error CHENYING.ASM(71) Too few operands to instruction
中文逗号问题
Error CHENYING.ASM(73) Too few operands to instruction
同上
Error CHENYING.ASM(74) Illegal instruction for currently selected processor(s)
同 70 行
Error CHENYING.ASM(79) Unknown character
Error CHENYING.ASM(80) Illegal instruction
Warning CHENYING.ASM(81) Reserved word used as symbol: AX
Error CHENYING.ASM(81) Extra characters on line
Error CHENYING.ASM(82) Too few operands to instruction
Error CHENYING.ASM(83) Too few operands to instruction
Warning CHENYING.ASM(87) Open procedure: A10MAIN
以上均为中文逗号或分号问题引起的。
其它错误：54 行变量名 PRALIST 写错了。

（11）ASM 程序中的[]中直接给出内存地址问题：
问题：这段代码的目的是为了检验大小写状态，(大写输出 BA,小写输出 A)，可是为什么无论我键盘状态是大写还是小写，输出的结果都是 A ？

```
.MODEL SMALL
.CODE
S:
MOV AX,40H
```

```
MOV DS,AX
MOV AL, [17H]
AND AL,01000000b
JZ L
MOV AH,2H
MOV DL,42H
INT 21H
L:
MOV AH,2H
MOV DL,41H
INT 21H
MOV AX,4C00H
INT 21H
END S
```

答案：我在纯 DOS 下试过，一切正常。WINDOWS 中的 DOS 方式是不可靠的。但您的程序有严重问题，在汇编时有以下警告：*Warning* TEST.ASM(6) [Constant] assumed to mean immediate constant，其意思是 MOV AL, [17H]一行将被汇编器翻译成 MOV AL, 17H（您 DEBUG 生成的 EXE 就知道了）。您应改写为 MOV AL, DS:[17H]，这样就不会将[17H]当作 17H 常数了。因为 ASM 源程序中直接寻址的用法是很罕见的（即[]中直接写出地址），汇编器器认为您不会是这种用法，因此将其转换为常数。一定要注意警告信息。