C 高级实用程序设计

王士元 编著

清华大学出版社

(京)新登字 158 号

内 容 简 介

本书针对目前应用程序设计的热点，如中、英文菜单设计，画图，动画，中断程序，程序的驻留，屏幕图形的存取、打印，C 程序汉字显示技术，C 语言与汇编语言的混合编程，C 语言与 FoxBASE 的混合编程等进行设计示范，附有大量示例程序和注释。本书也用了部分篇幅对高级程序设计涉及的硬件及 C 中的文件、指针、内存分配、图形适配器等内容进行分析，并简单介绍了实用程序编程方法。本书适用于理工科本科生、研究生和广大计算机应用人员。

TS200/18

图书在版编目(CIP)数据

C 高级实用程序设计/王士元编著. —北京：清华大学出版社，1996.3
ISBN 7-302-02063-9

Ⅰ.C… Ⅱ.王… Ⅲ.C 语言-程序设计 Ⅳ.TP312C

中国版本图书馆 CIP 数据核字(95)第 23619 号

C 高级实用程序设计

出版者：清华大学出版社(北京清华大学校内，邮编 100084)
印刷者：北京大中印刷厂
发行者：新华书店总店北京科技发行所
开　本：787×1092　1/16　**印张**：32　**字数**：793 千字
版　次：1996 年 6 月第 1 版　1997 年 2 月第 2 次印刷
书　号：ISBN 7-302-02063-9/TP・959
印　数：6001—12000
定　价：32.00 元

C 高级实用程序设计

王士元 编著

清华大学出版社

前言

C语言使用愈来愈普及，愈来愈深层次，许多读者，由于对知识的渴望或实际工作的需要，希望能更深入的了解一些C语言与PC机硬件的联系，希望能对C语言编程时遇到的一些技术难点给予帮助，这些难点诸如对PC机内存的分配与管理，80x86 CPU内部寄存器的使用，BIOS和DOS的调用，指针的分类与使用，堆和栈的结构及使用，文件的操作，对PC机输入输出接口的编程，中断处理程序的编写，驻留程序的设计，各种图形的绘制技术，中英文菜单程序设计，屏幕图形的存取与打印技术，C程序中的汉字使用及显示技术，C语言与汇编语言的混合编程，C语言与FoxBASE(dBASE)的接口技术等，这些问题也是目前C程序应用设计中的一些热点，它们涉及到PC机的硬件结构，中英文DOS系统和编程时积累的实践经验。

由于目前较多地看到的是一些有关C语言的教科书和参考书，对于上面提到的程序设计中的热点问题(它们应用于科研、教学、工程项目和软件产品中)的书较少见，这也是目前广大读者急需的。由于这方面的程序设计往往和PC机的系统结构联系在一起，属于C程序深层次的应用，故而称为高级实用程序设计。

笔者作为一个教师，一个科研工作者，一个较早搞计算机设计与应用且目前又直接讲授语言的人，了解一些学生及搞计算机应用开发的人们急需的要求，为此将自己的一些实践、体会、心得及参考到的新内容写到这本书中，奉献给读者，以期望能对进行C程序设计及软硬件开发人员能有所帮助。

由于Turbo C 2.0和Microsoft 5.0～7.0是目前大家较熟悉且又比较流行的C语言编译系统，而Turbo C有较良好的集成开发环境，所以本书以Turbo C 2.0为开发环境，书中的程序均在此环境中进行了调试，源程序集中在一张高密盘上，这些程序也可以方便地移植到别的编译系统上。

由于作者水平所限，加之时间仓促，书中错误及不妥之处在所难免，希望广大读者及专家批评指正。

作者在此对引入我出书的引路者——王家骅教授表示感谢，乔晓原、乔圆圆、王津涛、刘振实各位老师给予了不少帮助，李乐天、朱恩愈教授审阅了本书，提出了宝贵意见，在此深表谢意。

作　者

1994年12月于南开大学

目 录

第 1 章

概　述

C 语言是当前最流行的程序设计语言，它像其它高级语言一样，面向用户，面向解题的过程，编程者不必熟悉具体的计算机内部结构和指令；C 语言又像汇编语言一样，可以对机器硬件进行操作，如进行端口 I/O 操作、位操作、地址操作，并可内嵌汇编指令，将汇编指令当作它的语句一样。我们知道，汇编语言将涉及计算机硬件，所以 C 语言又像低级语言一样，可以对计算机硬件进行控制，因此人们把它称为介于高级语言与低级语言之间一种中级语言。

由于 C 语言的这种特点，因而它不但用于编一般应用程序，而且许多大的操作系统、编译系统也是由 C 语言来写的，甚至可以说 C 原来就是写系统软件的。因为它是和 UNIX 操作系统同时发展起来的，它最初用来写 UNIX 操作系统，由于 UNIX 的不断移植和推广，C 语言也就不断得到发展和普及，像后来的 PC-DOS，WORDSTAR，DBASE Ⅲ，PLUS 等都是由 C 语言和汇编语言相结合写成的。

1.1　C 语言的发展状况

1970 年美国贝尔实验室的 Ken Thompson 和 Dennis Ritchie 完成了 UNIX 的初版，与此同时，他们还改写了由 Martin Richards 开发的 BCPC 语言，形成了一种称为 B 的语言，此后 B 语言又进一步被进行了改进和完善，形成了称之为 C 的语言。

C 语言大约形成于 1972 年，1973 年 Dennis Ritchie 把 UNIX 系统中的 90%又用 C 语言进行了改写，并在 PDP-11 小型机上完成了用 C 语言及汇编语言编写的 UNIX 操作系统的调试，并将其投入运行，因而随着 UNIX 的移植、推广，C 语言也得到移植和推广，DOS 支持下的，甚至 Windows 支持下的 C 语言都相继出现，如我们目前广泛使用的在 PC 微机上的 MS-DOS 支持下的 Turbo C 和 Micrsoft C 就是典型的 C 语言版本。

由于众多的适用于不同机种、不同字长的 C 编译系统的出现(达几十种)，给 C 语言的统一性和兼容性带来了困难，虽然它们基本遵循在 UNIX V 操作系统上配备的 C 语言标准，但仍有许多各自的特点，使得相互移植困难，不利于推广，为此美国国家标准局(ANSI)语言标准化委员会公布了一个 C 语言标准草案(83 ANSI C)，该委员会又进一步修改完善该草案，终于在 1987 年公布了 C 语言标准，这就是目前投入商业运营的各种 C 编译系统所遵循的 87 ANSI C 语言标准，Turbo C、Microsoft C 均遵守这个标准。由于 PC 机的快速发展，原来的 C 标准已不能满足对 C 的功能要求，因而在不同的 C 语言版本中又增加了许多新的功能，它们可看作是对 C 标准的扩充和增强，如最明显的是 C 中增加了对屏幕分辨率的定义和一些图形功能的实现。

C 语言是一种结构程序设计的好语言，但随着软件处理对象从简单的数字和字符串发展到目前的图、文、声、象，信息量愈来愈大，愈来愈复杂，因而对程序的设计方法提出了更高的要求。随之在 80 年代出现了一种崭新的程序设计方法——面向对象的程序设计方法，结构程序设计的基本单位是模块，而面向对象的程序设计基本单位则是对象，因此 C 又进一步发展和充实，又出现了支持面向对象的程序设计语言C++，它实际上是 C 的超级集，其基本核心仍是 C。

例如美国 BorLand 公司 1989 年推出了 Turbo C 2.0，它又在继承和发扬 Turbo C 2.0 的集成开发环境的基础上，推出了面向对象的程序设计语言 Turbo C++，它实际上是 Turbo C 的一个超集，Turbo C++包含了所有 Turbo C 的内容，因而学好、用好 C，仍是面向对象程序设计的基础和前提，它甚至也可体现出面向对象设计的一些方法。

1.2 C 语言的编程格式

C 语言程序一般用小写字母，而仅在一些宏定义中，将常量名用大写字母表示，或对一些有特殊含义的变量，偶而也用大写表示，C 语言中对大小写字母是视作两个不同的量。

在 C 语言程序中没有程序行的概念，即在一行中可以任意书写多个语句，只要每个语句用分号作为结尾即可，多个语句还可用大括号{}括起来，形成一个如同单独语句一样的复合语句。一般情况下，为了层次清楚，一行只写一个语句，对复合语句也是按组成的语句依次写在不同的行中。

1.2.1 C 语言程序的结构特点

1. 一个 C 语言程序，通常由带有＃号的编译预处理语句开始，如＃include ＜文件名＞它表示，在编译源程序前，用指明的文件取代该预处理语句，通常文件名是指带有后缀为 .h 的磁盘文件，程序编译时，它将从磁盘中读出并插入到原来的预处理语句处，即预处理语句被所指明的包含文件(头文件)代替。

头文件通常是在程序中被调用函数的说明语句和该函数用到的一些符号常量的宏定义等，如在程序中要调用一些标准库函数时，系统提供了相应的头文件，它们其中的一些内容是对该函数的说明及该函数用到的符号常量的宏定义等，如对 fgets( )的说明放在 stdio.h 头文件中，在该文件中，包含了对 fgets( )函数的说明：

```
char *fgets(cbar *,int,file *)
```

对符号常量 NULL 的宏定义：

```
#define NULL 0
```

当然还包含对其它一些标准 I/O 函数的说明和宏定义等。

用户也可根据需要建立自己的头文件，如将程序中用到的符号常量组成一个单独的头文件，在程序开头用＃include 进行包含预处理，使得程序简捷方便。如在进行按键判别时，常要对键值(字符键值对应于 ASCII 码，功能键则为扩充的 ASCII 码)进行分析判别，以确定何键按下，键值为两字节的整型数，若高字节为 0，则表示低字节是代表相应的普通键的 ASCII 码；若低字节为 0，则高字节代表相应的功能键的扩充 ASCII 码。如在菜单程序设计中常用到一些功能键，为此可将它们的宏定义集合成一个文本文件，起名为 menu.h 头文

件：

```
#define    INSERT    0x5200
#define    ESC       0x0016
#define    TAB       0x0f09
#define    RETURN    0x000d
#define    RIGHT     0x4d00
#define    LEFT      0x4b00
#define    UP        0x4800
#define    DOWN      0x5000
#define    B         0x0e08
#define    HOME      0x4700
#define    END       0x4f00
#define    PGUP      0x4900
#define    PGDN      0x5100
#define    DEL       0x5300
#define    F1        0x3b00
#define    F2        0x3c00
#define    F3        0x3d00
#define    F4        0x3e00
#define    F5        0x3f00
#define    F6        0x4000
#define    F7        0x4100
#define    F8        0x4200
#define    F9        0x4300
#define    F10       0x4400
```

在程序开头用#include "menu.b"进行包含预处理后，在程序中就可用这些大写的键名，以代替难懂的键值，这样程序便显得明快易懂。不易因键值写错而导致程序有错。

当在模块化程序设计中，一个大的程序由多个文件组成，若这多个文件共同使用一些函数或常量时，可将这些共用的函数说明和符号常量的定义写在一个头文件中，然后在每个文件的开头用#include 进行包含说明，这样使得程序简捷、可靠，可维护性能提高。一般包含预处理有两种形式，即：

#include <文件名>

和 #include "文件名"

它们的区别在于对包含文件的查找路径不同，前者表示在安装编译系统时，在系统设定的标准目录中去查找，由于由系统提供的头文件一般放在 include 子目录下，所以常使用这种形式。

第二种形式表示在当前目录下去查找包含文件，若找不到，再到系统设定的目录中去查找。这种形式常用于用户自己写的包含文件，因为它们通常放在和用户程序同一个目录下。当然在这种形式中，文件名也可用文件路径名代替，这时，将在指定的文件路径中去查找文件。

文件包含也可用在程序中，如：

```
main ( )
  {
    ⋮
  #include "con.c"
    ⋮
  }
```

其中 con.c 是一个文本文件，编译时，它将从磁盘中调出，插入到该位置处。

一些在程序中用到的符号常量也常用编译预处理命令 #define 来进行定义，如在进行真假判断时，常用符号常量 TURE 和 FALSE 表示真假，这时可以定义：

```
#define   TURE 1
#define   FALSE 0
```

写在程序开头，这样在编译时在程序中出现 TURE 的地方，就表示为 1，即用 1 代替，出现 FALSE 的地方则用 0 代替。

当程序中经常要用到同一形式的表达式时，也可将该表达式定义为一个带参数的宏，使用时，如同调用函数一样，宏中写上参数即可。例如

求 x 和 y 中较大的一个：

```
#define max(x,y) (x>y ? x : y)
```

程序中经常要用到输出语句 printf("%d\n",x)，可定义为：

```
#define PRINTX printf("%d\n",x)
```

这样，只要在程序中需要求两个变量中的大者(如 a 和 b)，或需要上述格式的输出语句的地方，写上 max(a,b)或 PRINTX 即可。

2. 一个完整的 C 程序，总是由 main()函数开始，它像一个大型乐曲的引子，由此引出许多乐章——执行不同功能的函数；main()函数又像一个大型建筑的框架，它显示了要完成这个建筑的轮廓，这些轮廓就是由一个个的函数调用勾画出来的。因此可以说一个 C 程序是由一个个模块堆砌起来的，这个模块的最小单位是一个函数。当然模块也可以是一个源程序，它又由许多函数组成。所以 C 程序的设计，是一种模块化的设计，是许多函数的堆砌。因此在应用程序设计中，应将一个个的功能用一个个的函数来实现。

3. 函数的使用

Turbo C 提供了 400 多个标准库函数，每个函数都完成一定的功能，当程序需要执行这些功能时，只要调用这些函数即可，用户不用自己再去定义。这些函数包括输入输出函数、数学函数、字符串处理函数、内存函数、与 BIOS 和 DOS 调用有关的函数、屏幕和图形功能函数、过程控制函数、目录函数等。

当标准库函数中没有用户要用的功能函数时，就必须自己设计，设计函数的好方法是：

① 一个函数不宜处理太多的功能，要保持函数的小型化，功能单一化。

② 一个函数要保持自己的独立性，如同一个黑匣子一样，单进单出，即如下图如示：

这样一个函数编译后，其内部定义的程序代码与数据和另一函数的程序代码和数据不会相互影响，为此在函数中要使用局部变量，即它的生存期只维持在调用该函数并执行时，也就是说函数被激活时。要尽量少用或不用全局变量，它将破坏函数的独立性。函数的这种设计方法类似于面向对象设计方法中的封装性。

③ 可以在函数中多使用复合语句，使得函数也具有结构化，且可提高运行效率和节省存储空间。如假设一个函数可以划分成若干个逻辑上独立的程序段，将每个独立的程序段用复合语句表示之，然后用开关语句来控制它们的执行，即：

```
{
  ⋮
  switch (x) {
        case A : {…;break;}
        case B : {…;break;}
              ⋮
        default : {…;}
```

使程序显得清晰明快。

又如一个函数要对三个整型数组 a[300]、b[400]、c[500]分别进行同一处理后进行存盘。为了节省空间，可用复合语句来设计函数结构：

```
array() {
    {int a[300];…;}        /*分程序1*/
    {int b[400];…;}        /*分程序2*/
    {int c[500];…;}        /*分程序3*/
  }
```

这样相当于三个分程序，当执行完第一个分程序后，原分配的存储空间 300×2 字节将被收回，依次类推。实际上该函数仅用到了 500×2 个字节的空间，它比用形如 array(){int a[300],b[400],c[500],…}定义的结构节省了更多的空间。

在复合语句中定义一个局部变量的方法也可推广应用在其它程序的设计中。由于这种局部变量的生存期仅存在于复合语句的生命期中，复合语句执行完，它即失去作用，其占用的存储空间被系统收回，因而可节约存储空间，使程序层次清楚。

① 在主函数前，要罗列出所有使用的自定义函数的原型说明，这有利于在大程序设计中追踪要调用的函数设置是否正确，如参数传递对否？返回值一致吗？在函数定义时，采用现代定义风格。例如：一函数说明为

```
int average (int x,int y)
main()
  {
  int x
  char c;
  register result
  x=20;
  c='0'
```

```
  result=average (x,c);
  printf ("\The average is %d",result);
}
int average (int x,int y)
  {
    return ((x+y)/2);
  }
```

这样编译时，将会告诉你函数参数类型不匹配，并将参数 c(字符型)转换为 2 字节的整型数放入堆栈中。若没有程序前的函数说明，它将按一个字节的字符放入堆栈中。所以使用函数说明，可保证参数类型的正确性。如在主函数中用如下调用：

```
ch=average (x,r,p);
```

其中 ch 定义为字符，r 为一实数，检查该函数时马上会发现返回值和传递的参数个数和函数原说明不一致，立即会发现错误，编译时，也会立即报错。

当编制大型程序，不同的源程序模块共用一些函数时，可将这些函数的说明放在一个头文件中，每个源程序用 #include 进行包含说明，这是程序设计中常用的方法。

⑤ 在程序中适当的地方用/*…*/加入注释，这便于程序的阅读和调试。一个较大程序，经过数月后，再读它，可能许多关键的地方已记不起了，加注释可帮助你恢复记忆，调试时，也易于找错。

⑥ 采用层次的书写程序格式，按程序的不同功能分层，对 for、while、do_while、if_else、switch_case 等一类控制语句或它们的多重嵌套，采用缩格方式，可用 TAB 键进行控制，如设 TAB 键为每按一次为后移 4 个格(缺省方式为 8 个格，当要改变此值时，可在 Turbo C 集成环境下选择 Options 项下的环境子菜单 Environment，然后用光标再移到该子菜单的 Tah Size 项下，选择缩格的格数)。例如：

```
# include <…>
# define M 450
void fun1(int a,int b)
int fun2(char *p,int c)
void main()
  {int x,y;
    ⋮
   for (i=0;i<M;i++)
    {
      while (…)
       {⋮
        if (x>100)
           {
             ⋮
           }
         else
           {
             ⋮
```

```
            }
        }
    }
    ⋮
  printf(…);
}
```

1.2.2 模块化的程序设计

C 语言作为结构化的程序设计语言，易采用自顶向下的设计方法，即开始暂不涉及问题的实质和具体解决的步骤，而只是从问题的全局出发，给出一个概括性的抽象的描述。例如编写一个信号处理程序，它要求对信号数据经过数字处理后进行图形显示并存盘，因而程序大轮廓应是：

① 信号数据的输入
② 信号预处理
③ 信号进行数字处理
④ 进行显示
⑤ 进行存盘。

接着对各功能进行细分，例如对于信号的输入，又可分为：

① 通过 COM1 口由 RS-232 接口输入
② 由磁盘数据文件输入

对信号预处理又可分为：

① 对信号输入进行反序排列
② 用窗函数预处理

对数字处理又可分为：

① 求快速付立叶变换
② 求功率谱

对用窗函数预处理，又可分为：

① 海明窗处理
② 汉宁窗处理
③ 布拉格曼窗处理

其它功能类推。

在此细化的基础上再进行细化，以至于成为一个个单独的功能，因而可用一个个函数来实现。

设计一个个具体函数，就进入实质性阶段。要定义变量，要选取标准函数，要确定算法，这就是构造程序的基本单元。当一个个函数都设计完后，便可将这些函数按照调用的先后次序在主函数中堆砌起来，并用主函数作为总控程序，完成对它们的参数传递，控制选择对这些函数的调用，形成一个完整的实用的信号处理程序。

如要求数组 a[]的各元素和，奇数下标的元素的和，各元素中的最大值，各元素的平均值。设计程序时，可将要求实现的各项功能用函数实现，每个函数单独完成一个功能，数组传

递用一个指向该数组的指针 p 和指明元素项数 n 来完成，这样，数组元素个数的变化，不会影响函数的结构和逻辑，只要将 N 的定义和数组 a[]的各元素值改变一下，就又可适应新的数组了。

下面就是这样的一个程序，主函数 main()是一个总控程序，它对各函数进行了说明，然后依次调用完成各功能的函数，函数调用完，则整个要求实现的功能就完成了。

函数 arr add()单独完成对数组各元素求和，odd_add( )函数完成求奇数下标(即第 1 个元素看作是第一个奇数下标)元素的和，arr_max()完成求元素最大值的任务，arr aver()函数则完成求各元素平均值的功能。整个主程序是由一个个函数调用组成，如同一个个大砌块堆砌而成一样，显得结构明快，调试也方便。

下面是一个示例程序：

```
#define N 12
main()
{
  float a[]={1.5,4.6,7.8,9.2,7.4,6.5,7.0,9.8,8.7,
       21.7,15.6,34.3};
  void arr_add(float *p,int n);
  void odd_add(float *p,int n);
  void arr_max(float *p,int n);
  void arr_aver(float *p,int n);
  int n=N;
  float *pt=a;
  arr_add(pt,n);
  odd_add(pt,n);
  arr_max(pt,n);
  arr_aver(pt,n);
}
void arr_add(float *p,int n)
  {
   int i;
   float sum=0;
   for(i=0;i<n;i++,p++)
    sum=sum+*p;
   printf("the sum of %d elements is : ",n);
   printf("%8.2f\n",sum);
  }
void odd_add(float *p,int n)
  {
   int i;
   float sum=0;
   for(i=0;i<n;i=i+2,p=p+2)
    sum=sum+*p;
   printf("the sum of odd delements is : ",n);
```

```
    printf("%8.2f\n",sum);
  }
void arr_max(float *p,int n)
  {
   int i;
   float max;
   max=*p;
   for(i=0;i<n;i++,p++)
    if(*p>max)max=*p;
   printf("the maximum of %d elements is : ",n);
   printf("%8.2f\n",max);
  }
 void arr_aver(float *p,int n)
 {
    int i;
    float sum=0,ave;
    for(i=0;i<n;i++,p++)
     sum=sum+*p;
    ave=sum/n;
    printf("the average of %d elements is : ",n);
    printf("%8.2f\n",ave);
  }
```

1.2.3 大程序的设计风格

当一个程序较大时，可将一个程序分成几个部分，每个部分可单独成为一个源文件，这些源文件可进行单独编译，形成.OBJ 文件，然后将这些文件组合成一个大的实用程序，通常可采用如下的方法：

① include 方法

例如一个程序分成两个源文件，即由 A1.c 和 A2.c 两个源程序组成，这时可将 A1.c 写成：

```
#include<stdio.h>
#include"A2.c"
main()
{
    ⋮
  strcpy(s1,s2)
  A2 ();
    ⋮
}
```

而 A2.c 则可写成

```
#include<string.h>
```

```
void A2 ()
    {
     ⋮
    }
```

② 制作 project 文件

制作一个内容为：

A1.c

A2.c

的 project 文件(工程文件)，设名为 A1.prj，其中文件中各文件的后缀.c 可省略，先后顺序也无关，它仅影响编译时的顺序。这可在 Turbo C 的编辑状态下写成，并用 F2 或 write to 写成一个名为 A1.prj 的文件而存盘。

然后用 ALT_P 选择 Project 菜单中的 Project_name 项，填入生成的 A1.prj 文件名，然后按 F9 键，即可生成 A1.exe 执行文件。

当要用汇编语言和 C 语言混合编程时，则要将汇编语言子程序单独编译生成.OBJ 文件，然后制作成工程文件，再进行对 C 程序的编译和进行两者的连接，关于这方面的内容可参阅第 15 章中的有关介绍。

若程序还需一些其它的被编译的程序或库文件，这些文件是 C 语言标准库不能提供的，则也可将它们的名字放在 project 文件中，如：

```
mymain
myfunc
special.obj
other.lib
```

当用 F9 进行编译连接时，对后缀为.obj 的文件只进行连接，对后缀为.lib 的库文件不会进行编译，只是进行连接，这样当进行外部调用时，就会对库进行检索。

当多个源文件制作成 project 文件时，一个.c 的源文件依赖于其它.c 源文件，若它们之间用定义的一个头文件来进行接口，这时应用括号将这些头文件括起来(头文件之间可用逗号、空格或分号分隔)，这样一旦头文件改变时，它们将被重新编译，例如有一主程序名为 mymain.c，它包含头文件 myfuncs.h，而另一文件是 myfuncs.c，它也包含头文件 myfuncs.h，这样当 project 文件内容写成如下形式时：

```
mymain.c    (myfuncs.h)
myfuncs.c   (myfuncs.h)
```

若一旦 myfuncs.h 被修改，则对该 project 文件进行编译时，mymain.c 及 myfuncs.h 将被重新编译。

1.3 Turbo C 2.0 的程序设计开发过程

C 语言程序必须经历编辑—编译—连接过程，才能变成一个可单独执行的程序，即变成一个由机器指令代码组成的，能被硬件执行的程序。

Turbo C 提供了两种环境来实现这种过程，一种是大家熟悉的集编辑、编译、连接、运行

和调试为一体的集成开发环境，它使得上述过程可一气呵成；另一种就是所谓命令行编译，即在 DOS 下，用 Turbo C 的命令行编译程序对源文件进行编译、连接，最后生成一个执行文件。下面简单予以介绍：

1.3.1 Turbo C 集成开发环境

在 TC 目录下调用 TC 出现主菜单后，按 ALT-e 键即进入编辑，编辑完后按 F2 或选 FILE 菜单中的 write to 即可存入磁盘，然后按 Ctrl-F9 键即可完成编译—连接—运行三个过程，当编译—连接过程中不出现严重的语法错误(error)时，便生成. obj 和. exe 文件而存盘。

当然在 Turbo C 集成开发环境下，也可像传统的方法一样，分步进行，即：

按 ALT-c 键，选择 compile to obj 项，就会将文件编译成. obj 文件。

再选择 compile 中的 Link exe file 项，即可在规定的输出目录下，生成可执行文件. exe。

然后按 Ctrl-F9 或按 ALT-r 键后选择 Run 菜单项下的 Run 即可运行，按 ALT-F5 键可得到输出结果。

也可先编译连接，然后运行，这种方法像 UNIX 下的 CC 命令和 MS-C 系统中的 CL 命令，将编译连接合成一步，方法是选 compile 菜单项中的 Make exe file 项，这样，就在指定的目录中生成了. exe 文件。

1.3.2 Turbo C 的命令行编译连接

所谓命令行编译是指在 DOS 下，调用 Turbo C 的 TCC. exe 程序，来完成对源程序的编译连接工作。当选择对. asm 文件编译时，TCC 还要调用 TASM 后才能对后缀为. asm 的文件进行编译，这种方式适合于 C 程序与汇编语言的混合编程的编译连接，当 C 程序中嵌入汇编指令行时，也必须用此方法进行编译连接，如第 15 章中介绍的那样。

命令行编译程序 TCC. exe 的使用格式为：

tcc [选择项 1　选择项 2…]文件名 1　文件命 2…

其中选择项是指对后面给出的文件进行编译或连接时的选择项，可选的常用选择项如表 1.1 所示，每个选择项前均带有“—”减号，且大小写是有区别的。

文件名是指源文件. c 或目标文件. obj 或库文件. lib。

当不指定只编译不连接时，TCC 将完成编译和连接两个步骤，对. LIB 库只进行形式上的连接，标准库用户不用进行连接。

例如：

tcc —ib：\include —lb：\lib —etest start. c body. obj myc

当执行该命令时，表示将 start. c 源文件和 body. obj 目标文件及 myc. c(命令行中该文件无后缀)，分别进行编译(对 body. obj 不再编译)，然后连接生成名为 test 的执行文件 test. exe(由—etest 指出)。

—ib：\include　表示包含文件的路径是 b：\include

—lb：\lib　表示库文件的路径是 b：\lib

又例如：

tcc —ms —efile —lc：\tc\lib file1 file2. obj graphies. lib

其中－ms 表示选择小内存模式进行编译，它也是 Turbo C 缺省的编译模式。将 file1 进行编译，然后和 file2.obj 及 graphies.lib 进行连接，生成名为 file.exe 的执行文件。其中 graphies.lib 库的路径为 c:\tc\lib，即意为在 c:\tc\lib 目录下去寻找 graphies.lib 库文件。

当进行混合编程时，如已有汇编源程序 S3.asm，其命令行可写为：

tcc ic：\include －lc：\tc\lib －mm S1 S2 S3.asm mylib.lib

表示用中模式(－mm)编译源文件 S1.c 和 S2.c，调用 TASM 对 S3.asm 进行编译，然后连接生成执行文件 S1.exe，编译时，到 c：\include 目录中去寻找包含文件，在 c：\tc\lib 目录中去寻找库文件 mylib.lib。

表 1.1 给出命令行选择项及其含义。

表 1.1 命令行选择项

选择项	含义	选择项	含义
－C	只编译.OBJ 文件	－mT	极小内存编译模式
－B	编译带有内嵌汇编指令行的程序	－mm	中内存编译模式
－f	使用浮点仿真	－mL	大内存编译模式
－L	指定库文件路径	－W	显示警告错误
－I	指定包含文件路径	－W－	不显示警告错误
－S	输出一个汇编模块格式	－n×××	指定输出的目录(×××)
－C	只编译不连接	e×××	指定生成的执行文件名(×××)
－mS	小内存编译模式		

第 2 章

PC 机存储器结构及变量存储方式

2.1 PC 机存储器结构

PC 机的 8088 处理器，地址总线为 20 位，故直接寻址能力为 $2^{20}=1048576=1024K=1M$，系统使用其中的 384K 作为固定存储器 ROM 和随机存储器 RAM 的地址，用作硬件和开机使用，另 640K 地址用作程序和数据区。同属一个系列的 80286 和 80386 及 80486 寻址能力大大增加，如 80286 可达 16M，而 80386(80486)可达 4G(即 4 千兆)，由于 PC 操作系统的延续性，又考虑到和大量已存在的软硬件的兼容性，因此 286，386(和 486)的系统存储器仍为 1024K，即 1M，将大于此地址范围的存储器称为扩展存储器(Extended Memory)和扩充存储器(Expanded Memory)，现就其结构作以下简单介绍。

2.1.1 系统存储器(System Memory)

其结构如图 2.1 所示，它可分成两个部分，即低地址部分(0～640K)和高地址部分(641K～1024K)。

其中高地址部分又称为高端存储器、上位存储器，低地址部分又称为基本内存或常规内存。基本内存是 PC 机得以工作的最低内存，不能再少于此，这部分内存地址使用情况如图 2.2 所示，其中主要由 DOS 和用户的应用程序及驻留程序 TSR 占用，DOS 所占用的内存量要视其版本而异，如 DOS 3.3 约占 50K，DOS 4.0 约占 60K，DOS 5.0 约占 55K(当不用扩展内存和扩充内存时)。用户的应用程序所占地址取决于程序长短，这部分空间是自由空间，当回到系统时，该部分空间即被释放，有部分 DOS 占用的空间也是暂住部分，如常用的外部命令，一旦执行时，即调入该空间，执行完后，该空间又留作调用其它 DOS 命令，TSR 驻留部分将是受保护的空间，系统只要不重新启动，它的内容将始终存在，这部分主要由 DOS 的常驻部分占据，用户的驻留程序也存在此。

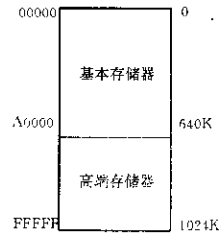


图 2.1 系统存储器

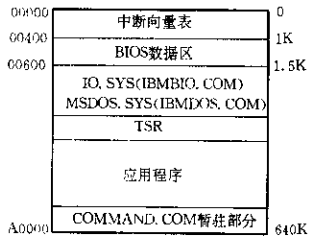


图 2.2 基本存储器

高端存储器占 384K，它的地址使用情况如图 2.3 所示，其中 A0000 到 BFFFF 用作显示缓冲区，其中 CGA 显示缓冲区开始于 B8000，单色显示缓冲区开始于 B0000，而 VGA，EGA 显示缓冲区开始于 A0000，这个显示缓冲区将在后面要用到，即我们将要在 Turbo C 作图部分用到的那部分 VRAM，C0000 到 DFFFF 部分为 ROM 扩充区，它用作存视频显示器适配器和磁盘的 BIOS，一些网络控制板，I/O 接口板，EMS 分页帧（它是 64K 区，用作存取扩充内存页面的窗口）也要用到此部分地址，E0000 到 EFFFF 为保留区，IBM PS/2 机用作附加系统 ROM。F0000 到 FFFFF 是系统 ROM，该部分装有系统引导程序（上电开机，即执行该引导程序，关于这部分的作用将在下一章 DOS 的启动中介绍），还有系统基本输入输出系统（BIOS）。

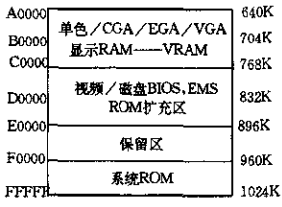


图 2.3 高端存储器

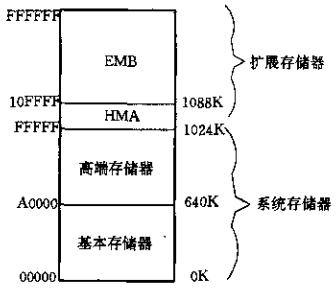


图 2.4 存储器地址分配

2.1.2 扩展存储器(Extended Memory)

扩展存储器指高于 FFFFFH 或 1M 的那部分存储器。如图 2.4 所示，它只能由 80286，80386，或 80486 直接去访问，80286 有 24 位寻址能力，故可寻址到 16MB，而 80386 和 80486 有 32 位寻址能力，可直接寻址 4GB，这些处理器对系统存储器采用实地址方式访问，而对扩展存储器采用保护虚地址方式，因而在实地址方式下运行 DOS 时（这就是我们通常执行的方式），无法对扩展存储器进行寻址，而必须进行方式切换，即由实方式换到保护方式，对扩展存储器操作完后，再切换到实地址方式下，恢复 DOS 的操作，扩展存储器同系统存储器一样，一般安装在系统板上，以内存条形式插在内存条插座上。

由于在实地址方式下，当采用段：偏移地址进行寻址时，地址可达 FFFF：FFFF，这已到达 1MB 内存之上的 64KB 区域了，而这部区域已属于扩展内存，我们又将这一 64K 可在实地址方式进行直接存取的扩展存储器称为高内存区 HMA(High Memory Area)，当使 CPU 的第 21 条地址线(A20)变高时，可在实地址方式下对其进行存取，关于 HMA 的使用，将在下面进行介绍。

2.1.3 关于 HMA(高端存储器)的使用

用段地址加偏移地址的方法可寻址到 FFFF：FFFF，这恰是 1MB 上面的 64KB 内存段的末地址，它们又称为 HMA(High Memory Area)，一般应用程序很少会用到这 64KB 的内

存，为了能用到这段区域，DOS 5.0 以上版本提供了管理 640KB 以上内存的 XMS 内存管理程序 HIMEM.SYS，这样只要在 CONFIG.SYS 文件中加入下面两行内容：

```
DEVICE=C：\DOS\HIMEM.SYS
DOS=HIGH
```

用 C 盘上的 DOS 启动时，IO.SYS 便在 C 盘上查找 CONFIG.SYS 文件，按照这个文件中登录的内容最新配置操作系统，即第一行表示将 C：\DOS 目录内的 HIMEM.SYS 程序安装到 DOS 系统中，这样 DOS 就可以控制 640KB 以上的内存使用，第二行命令表示将 DOS 放到 HMA 内存去，这样就可使 DOS 占用的基本存储器仅为几个 KB，增加了用户可用的区域达 50KB 左右。

对于 CONFIG.SYS(configuration file)文件有必要解释一下，它是一个告诉 DOS 系统配置的文件，用户可以通过改变该文件中命令行的内容，使 DOS 的运行环境和驱动设备的能力发生变化，因 DOS 开始启动时，首先要查找 CONFIG.SYS 文件，如我们在该文件中写上 DEVICE=C：\DOS\HIMEM.SYS，DOS 就会把 HIMEM.SYS 程序增加到 DOS 中去，而看到 DOS=HIGH，就将由 HIMEM.SYS 负责将 DOS 装到 HMA 去(假设 CONFIG.SYS 文件中未有其它程序使用扩充内存或扩展内存的命令)，当安装成功时，便显示信息：

High Memory Area Already In Use

这样就为用户增加约 50KB 的基本内存空间，若别的应用程序要使用 HMA，则可在 CONFIG.SYS 中去掉 DOS=HIGH 即可。

2.1.4 扩充存储器(Expanded Memory)

扩充存储器是指 PC 处理器寻址范围之外的物理存储器，它也被称为 EMS 存储器，厂家通常以扩充内存板的形式提供，使用时可插在 PC 机的扩充插槽上。

扩充存储器使用时，用高端存储器中的一段地址分配成扩充存储器的分页帧(page frame)，用页面映射到 EMS 某一区域，使得应用程序可以访问 EMS 存储器，不断改变分页帧对 EMS 的映射区域，即可访问 EMS 的所有存储区，所以对 EMS 的访问是通过占用系统存储器中的高端部分地址来换取对大量 EMS 存储区的访问，即以小换大。具体作法是在

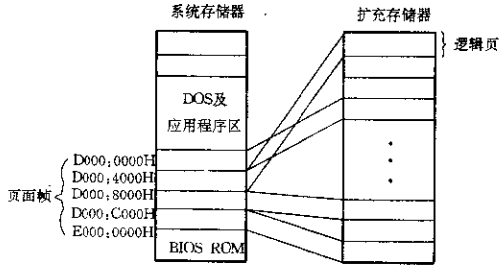


图 2.5 系统存储器与 EMS 的映射关系

A0000H 以上的高端存储器中，选取 DOS 系统没有使用的连续 64KB 地址空间作为分页帧，这 64KB 地址空间又均分成 4 个物理页，这 4 个页相当于 4 个窗口，用它来映射 EMS 的逻辑页，EMS 可分成许多逻辑页，每一逻辑页也为 16KB 大小，这样一个物理窗口可映射一个 EMS 中的逻辑页，4 个物理窗口可同时映射 4 个逻辑页，即 64KB 空间，若不断改变 4 个物理窗口的映射关系，则可访问完 EMS 的所有存储区，对于一个应用程序来说每次最多可访问 EMS 的 64KB 区域，即 4 个逻辑页，系统存储器与 EMS 的映射关系如图 2.5 所示。

2.2 存储器的分段与物理地址的形成

80286 和 80386 有两种地址操作方式，即实地址方式和保护虚地址方式，实地址方式为了和 8086/8088 处理器相兼容而设置的，在此方式下，物理寻址空间为 1M，Turbo C 就工作在实地址方式下，为了对扩展存储器进行寻址，就要采用保护虚地址方式，在该方式下可使 80286，80386 寻址存储空间达 1000M（多任务情况下），对单个任务 80286 可提供 16M 的地址空间，80386，80486 可提供 4GB 的地址空间。

现将实地址方式下的存储器段地址和偏移作以介绍：

传统的微处理器总是用 16 位地址字来表示。所以在内存中能选取的最多地址是 65536，即 64K。随着微型计算机变得愈来愈复杂，因而对存储器的容量要求愈来愈大，8088 仍采用 16 位地址字，但是它通过使用段地址，使存储器容量增加到 1 兆字节（1024K）。

地址若是 1 兆字节，就必须要用 0～1048576 范围的数来表示，也就是需要 20 位地址字，8088 对真实的存储器是用 20 位地址字，1048576 个不同地址集合，就表示了 1 兆地址空间。但实际在程序中仍采用 16 位地址，但通过地址分段的方法使 16 位地址可变成 20 位地址字来进行直接寻址。如何分段呢？它是将存储器地址分成许多块，称为段，每一个段内包

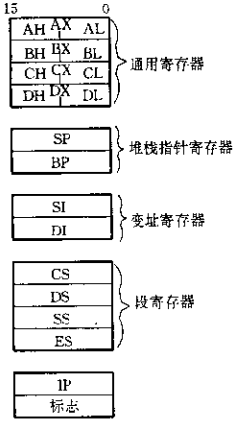


图 2.6 8088 寄存器组

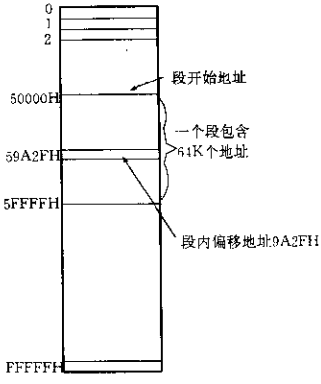


图 2.7 1M 地址空间的 64K 段

含有 64K 个地址，因而在段内可以用标准的 16 位地址字，这种地址又称为偏移地址，它仅表示了离该段开始处的距离，如图 2.7 所示。

一个段可以从 1M 地址空间的每一个地址为 16 倍数的边界开始，如图 2.8 所示。注意！有 64K 个不同的段开始地址，每一个开始地址其低四位全是零，每个段不一定包含有 64K 个数据，64K 仅是允许最大的可能值，一个段可以包含一个、百个、千个字节……，另外段还可以相互重叠，即同一个字节数据，可以从一个或多个段的开始地址存取。

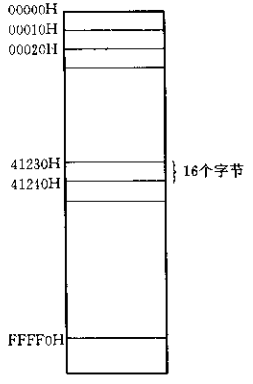


图 2.8 可能的段开始地址

8088 是怎样使用段地址呢？（见图 2.9），当 8088 对存储器进行存取时，选择一个段寄存器值作为该段的开始地址，将其值左移 4 位，再加上偏移地址，就得到 20 位物理地址，即存储器的真实地址。这个地址被用来作为对存储器进行存取的真正地址。

图 2.10 表明了四个段寄存器，并指出处理器何时选择它们来作地址计算。段寄存器寄存段的开始地址，CS 段寄存器总是定义成当前的代码段地址，该段内含有处理器要执行的指令。以前我们指出过 16 位 IP 寄存器指示了要执行的指令地址，实际上，当处理器要取指令时，是从 IP 寄存器中找到偏移地址，从 CS 寄存器中得到段地址的。

DS 寄存器值定义为数据段的开始地址，它用在通用数据的存取，以及字串操作指令的源操作数。SS 寄存器定义为堆栈段的开始地址，它用于所有的堆栈操作，最后的 ES 寄存器用来定义附加段地址，该段用作第二个通用数据区，附加段在字串处理指令中作为放置目的操作数而被存取。

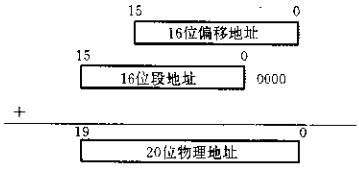


图 2.9 物理地址的计算

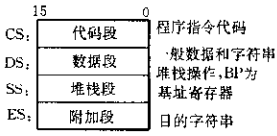


图 2.10 8088 段寄存器

四个段寄存器允许 8088 编程者同时对四个不同的地址空间存取，每一个空间可以包含多达 64K 字节的数据，图 2.11 给出了如何设置段寄存器的例子。在图 2.11(a)中，各段寄存器被设置成允许同时存取存储器的可能最大值，在这种配置下，可以有 64K 字节的指令，64K 字节的堆栈空间，两个 64K 的数据空间。图 2.11(b)给出了一种更实际的配置，在代码段，有 8K 程序(2000H)，在数据段中有可存取的 2K(80H)数据和可以使用的 256 个字节的堆栈。由于要求存储的数据很少，因而附加段不需要，ES 寄存器便被设置成和数据段重叠。

因为段寄存器在程序控制下可以被读和被写，段的地址和大小可动态地改变，因而在程序设计时有可能设计出最好的配置。

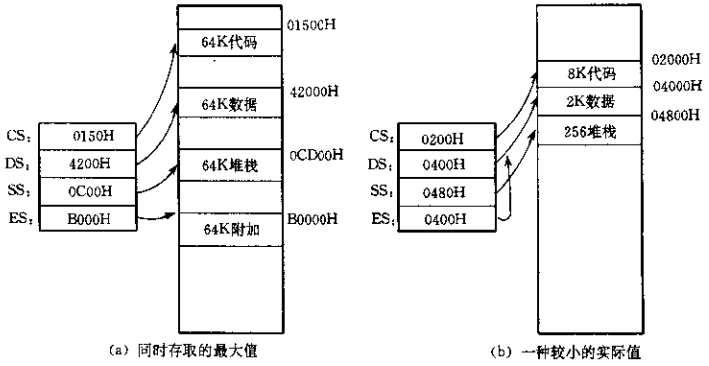


图 2.11 典型的段寄存器配置

2.3 与地址操作有关的几个宏

Turbo C 定义了几个与地址操作有关的宏，它们均在 dos.h 头文件中，使用时，要将该文件包含进去，即要在程序开头写上 #include<dos.h>，所谓宏是指定义了一组指令(用于完成某一特定操作)用一个宏名代替，当程序进行编译时，该宏便将代表的一组指令插入宏名处，即宏展开。

如从一个地址中(指针)分离出段地址和偏移地址的宏：

unsigned FP_SEG (void far * p)

unsigned FP_OFF (void far * p)

它们将分别从远地址指针中得到该指针表示的段地址和偏移地址。

如果有了段地址和偏移地址，还可将它们组合成一个实际物理地址，即一个远指针，这可由宏：

void far * MK_FP(unsigned segment,unsigned offset)来完成。

假设我们知道了一个地址单元(即知道了段地址和偏移地址如 ob000：0020)，用下面程序进行这些函数的演示：

```
#include<stdio.h>
#include<dos.h>
main()
  {  char far * p;
     unsigned seg,off;
     p=MK_FP(0xb000,0x20);
     seg=FP_SEG(p);
```

```
    off=FP_OFF(p);
    printf("far p %fp,segment%04x,offset%0x\n",p,
    seg,off);
}
```

最后输出 far p b000：0020,segment b000,offset 0020

当知道了某地址单元的段地址和偏移地址后，也可从内存的该地址中取出一个字节或从相邻的两个单元中取出一个字来，或者写入一个字节或一个字，这可用下面的宏来实现：

```
char peekb(unsigned segment,unsigned offset)
int peek(unsigned segment,unsigned offset)
```

它们将分别从给出的地址单元取出一个字节或从相邻两个单元取出一个字来。

下面的宏：

```
char pokeb(unsigned segment,unsigned offset)char value
int poke(unsigned segment,unsigned offset)intvalue
```

它们将写入一个字节或一个字到所指出的地址中。

注意！当没有用#include<dos.h>，这时程序中出现的这些宏名则被认为是一些调用函数，若用#include<dos.h>进行包含，而在程序某处又用#undef 进行结束，则这以后若使用这些宏也被认为调用了函数，所以在程序中使用这些与地址操作有关的宏时，必须事先要用#include<dos.h>进行包含说明，且不可中途用#undef 进行结束。

2.4 寄存器与伪变量

2.4.1 80x86 的内部寄存器

80x86 CPU 有 14 个基本寄存器，它们用于进行运算，控制指令的执行，处理内存寻址等，在 8088,80286 CPU 中，它们是 16 位长，但在 386,486 CPU 中，它们扩展为 32 位长(段寄存器除外)，如图 2.12 和图 2.13 所示，虽然 386,486 CPU 又增加了一些特殊的内部寄存器，但 Turbo C 只支持这 14 个基本寄存器，且长度认为是 16 位。即按照 8088 的 14 个内部寄存器来进行操作。

这 14 个寄存器按其功能可分为六类：

通用寄存器

地址指针寄存器

变址寄存器

段寄存器

指令指针寄存器

标志寄存器

1. 通用寄存器

即可作 8 位又可作 16 位数据寄存器用。这取决于是字节操作还是字操作。作 16 位用时，称为 AX、BX、CX、DX，它们又可分成高字节部分和低字节部分来分别被寻址，即高字节部分对应于 AH、BH、CH、DH，低字节部分对应于 AL、BL、CL、DL，这样四个 16 位寄存器可看作八个 8 位寄存器来使用。

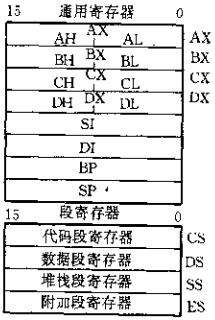


图 2.12 8088 内部寄存器

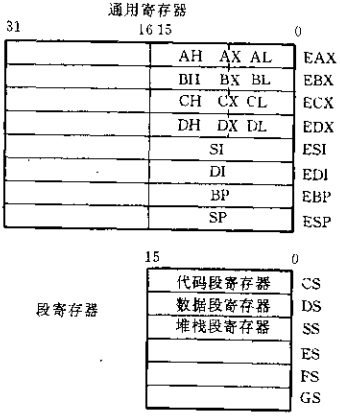


图 2.13 80386 的基本寄存器

(1) AX 寄存器

又称为累加器，用于所有的输入/输出操作。某些字串操作以及算术运算，如乘法、除法，某些翻译指令也使用 AX 寄存器。

(2) BX 寄存器

又称为基址寄存器，用于扩展寻址，起变址作用。

(3) CX 寄存器

又称为计数寄存器，在循环操作和移位操作中用作计数器。

(4) DX 寄存器

又称为数据寄存器，用于字乘法和除法，还用来提供输入输出操作中的口地址。

2. 段寄存器

在寻址内存单元时，最多可分成四个段，每个段最大范围可寻址 64K 字节，这些段的首地址保存在四个段寄存器中(省略了地址的低 4 位，即每段从内存段界开始，低 4 位认为是 0)，这些段寄存器是：

(1) CS 寄存器

代码段寄存器。它含有当前执行程序所在段的首地址，这个段寄存器内容左移 4 位(即乘 16)再加上指令指针寄存器(IP)的内容，就是下一条要执行的指令地址。

(2) DS 寄存器

数据段寄存器。它含有当前数据段的首地址，通常数据段用来存放数据和变量，数据段寄存器内容左移 4 位加上指令中的偏移值，即为对数据段指定单元操作的地址。

(3) SS 寄存器

栈段寄存器。它保存当前堆栈的首地址，堆栈是存储器中设置的一种数据结构，用于存放数据和地址，有先进后出的特点，在调用子程序时，保留返回主程序的地址，也用来保留进入子程序将要改变其值的各寄存器的内容等。

(4) ES 寄存器

附加段寄存器。附加段是在进行字符串操作时，作为目的段地址使用，这个附加段是附加的一个数据段，如用 DS 作为数据段段地址，SI 为源地址偏移，将一个字串从源传送到目的区去，这个目的区可设在附加段中，即 ES 装附加段地址，DI 寄存目的区地址的偏移。如果要使用附加段，用户程序必须对 ES 初始化，即设置值，否则 ES＝DS，即数据段和附加段重合。

在讲到的四个段，其地址可以重叠，也可以不重叠。

3. 指针寄存器

(1) SP 寄存器

栈指针寄存器。在访问堆栈时，段地址在 SS 中，SP 则表示偏移地址。

(2) BP 寄存器

基址指针寄存器。在通过堆栈传通数据和地址时，段地址在 SS 中，BP 则存放要传递的数据和地址的偏移地址。BP 也可作为通用寄存器用。

4. 变址寄存器

(1) SI 寄存器

源变址寄存器。在字串操作中，常用 SI 表示字串的源地址，段地址在 DS 中。

(2) DI 寄存器

目的变址寄存器。在字串操作中，常用 DI 表示操作的目的地址，它常和附加段寄存器 ES 相关连，如前述的字符串传送操作，DI 就表示了传送的目的地址偏移。

5. 指令指针寄存器 IP

IP 寄存器寄存要执行的下条指令的偏移地址，在用户程序中不需要使用该寄存器，但可用 DEBUG 程序改变其值，以改变程序执行方向，用于调试程序。IP 的内容可以进栈或退栈。

表 2.1　各寄存器配合使用情况

存储器操作类型	约定的段寄存器	另外可使用的段寄存器	偏移地址
取指令	CS	无	IP
堆栈操作	SS	无	SP
取数据或变量	DS	CS、ES、SS	有效地址
串操作中取源串	DS	CS、ES、SS	SI
串操作中写目的串	ES	无	DI
BP 被作为基地址	SS	CS、ES、SS	有效地址

6. 标志寄存器

程序有时需根据上次指令执行的结果，判断以决定执行的方向，为此 8088 设有一标志寄存器，如图 2.6 所示。它是 16 位寄存器，其中 6 位作为指令执行结果的状态标志，3 位用作控制标志，即 8088 的标志寄存器不仅用来标志结果的状态，还可用作控制某些操作。现就各位的标志介绍如下：

(1) 状态标志位

CF　第 0 位，进位标志。如果算术指令执行完后，最高位产生进位或借位，则 CF＝1，否

则CF＝0。CF 还可保存移位或循环移位时移出的一位值，也可给出比较操作的结果，也可作为乘法结果的指示器。

PF　第 2 位，校验标志。当操作结果含有偶数个 1 时，PF＝1，否则 PF＝0，这个标志多用于数据传输中。

AF　第 4 位，辅助进位标志。当操作数第 3 位产生进位或借位时，AF＝1，否则 AF＝0，这个标志多用于压缩的十进制数操作。

ZF　第 6 位，零标志。当运算结果为零时，ZF＝1，结果非零，则 ZF＝0

SF　第 7 位，符号标志。对带符号的数操作时，若产生一个负的结果，则 SF＝1，否则 SF＝0，当算术、逻辑、移位或循环移位操作时，都将影响此位。

OF　第 11 位，溢出标志。当带符号数算术运算时，高位溢出，则 OF＝1，否则为 0，它用来作错误指示标志。

(2) 下面的 3 位为控制标志位：

TF　第 8 位，陷阱标志。当置为 1 时，则 8088 处于单步执行指令方式，每执行一条指令，自动产生陷阱。

IF　第 9 位，中断允许标志。当用指令置为 1 时，则允许 8088 响应中断请求，若为 0 时，则禁止响应中断请求。

DF　第 10 位，方向标志。可用指令预置。当 DF＝0 时，执行串操作指令后，变址寄存器自动递增，当 DF＝1 时，则自动递减。即该标志位可控制地址朝增加的方向或减少的方向改变。

2.4.2　伪变量

Turbo C 程序设计中，在进行与硬件有关的操作时，经常要用到上述的那些寄存器，但 Turbo C 函数又无法直接对它们进行操作，因而它采用了与这些寄存器同名的变量来代表它们，由于这些变量只和这些寄存器有关，不能作为它用，因面称为伪变量，且名字用代表寄存器名的大写来代替，并前面有一下划线，这样在 Turbo C 中使用这些伪变量就如同对这些同名的寄存器操作一样，它们和 C 语言的一些低级操作符及函数配合，就可实现如同用汇编语言对机器硬件的操作控制功能。

表 2.2 列出了这些伪变量和其代表的寄存器名及其所属的变量类型。

表 2.2　伪变量与代表的寄存器及类型

寄存器	类型	伪变量名	寄存器	类型	伪变量名
AX	unsigned int	_AX	DL	unsigned char	_DL
AH	unsigned char	_AH	CS	unsigned int	_CS
AL	unsigned char	AL	DS	unsigned int	_DS
BX	unsigned int	BX	SS	unsigned int	_SS
BH	unsigned char	_BH	ES	unsigned int	_ES
BL	unsigned char	_BL	SP	unsigned int	_SP
CX	unsigned int	_CX	BP	unsigned int	_BP

续表

寄存器	类型	伪变量名	寄存器	类型	伪变量名
CH	unsigned char	_CH	DI	unsigned int	_DI
CL	unsigned char	_CL	SI	unsigned int	_SI
DX	unsigned int	_DX	FLAG	unsigned int	_FLAGS
DH	unsigned char	_DH			

由于指令指针寄存器IP，用于程序执行过程中指出下条要执行指令的偏移地址，用户程序中不需要使用该寄存器，因而Turbo C中没有与其对应的伪变量。

2.4.3 伪变量的使用

伪变量代表一个特定的寄存器，它在使用上又可像普通变量一样，在表达式中出现，例如：

```
unsigned char S;
unsigned int B,C;
   ⋮
B=_BX
_DX=B;
C=_CX;
_AL=S;
   ⋮
```

但在使用时又要注意如下的一些问题：

1. 不能对伪变量用地址运算符&，因寄存器是没有地址的。

2. 由于寄存器在程序执行时被系统频繁的使用，用户无法预料在执行哪个语句或函数时，要用到哪些寄存器，因而一些寄存器的值将是不确定的，即是你给它赋了值，以后再读出时，其值可能已变化。

3. 将Turbo C程序经过编译连接，变成.EXE执行文件，当运行时，CS段寄存器用来存放代码段的段地址，DS用来存放数据段的段地址，这些数据包括全局变量和静态变量的值，而堆栈段的段地址则存放在SS段寄存器中，堆栈段主要存放局部变量，函数的参数，以及函数返回地址等，通过读取这些段寄存器，可以得到相应的段值，例如：

```
⋮
unsigned int SP,SS;
unsigned char st[0x1000];                    /*定义堆栈*/
⋮
SS=_SS;
SP= SP;
_SS=_DS;                                     /*数据段和堆栈段重合*/
_SP=(unsigned)&st[0x1000-2];                 /*设置堆栈指针*/
⋮
_SS=SS;                                      /*恢复原堆栈段地址*/
_SP=SP;                                      /*恢复原堆栈指针*/
```

⋮

上面的程序段经常在中断调用时使用，当改写原系统的中断调用，为了在完成被改写的中断调用后，恢复原系统的中断调用时的堆栈设置，则必须保存原中断调用时堆栈段地址和堆栈指针，在调用功能完成后，再恢复它。

4. 在程序的代码段，数据段，堆栈段均不超过 64K 字节时，程序运行的自始至终，CS，DS 和 SS 的值不变，即代码段、数据段、堆栈段的段地址均不变，因面若用户在程序中改变这些寄存器对应的伪变量值，将可能导致程序破坏。在编译模式为最小模式(Tiny)，小模式(small)时，属这种情况。

5. 由于不能预期那些寄存器何时使用，有可能这时的寄存器值，就是下一个语句或函数要使用的值，因而在程序中任意修改某些寄存器的值，将可能破坏程序的运行，这一点要引起注意。

2.5 保护虚地址方式下的段和偏移

对扩展存储器寻址，只能采用保护虚地址方式，我们知道在编写程序时，地址是逻辑地址，不是真实的物理地址，即可看作是虚拟的，当它变成执行程序时，才使虚拟地址变成真实的物理地址。在保护虚地址方式下，80286 和 80386、80486 可提高存储空间达 1000M，即对每个任务可分别提供 16M 的存储空间。这些芯片是为多任务、多用户环境设计的，但目前它们都是在单任务的 DOS 下工作，较早的 DOS 并没有提供使用这种方式的方法，新的 DOS，如 5.0 版以上提供对其简单的使用。

在保护虚地址方式下，为了提供较大的虚似存储空间，物理地址的生成与实地址方式下完全不同，这主要体现在对指令给出的 16 位"段地址"的解释不同。在实地址方式下，把它作为段地址同偏移地址相加而生成物理地址，但在保护虚地址方式下，这 16 位的字称为段选择字，它指出了段描述符在描述表中的索引，由此可找到段描述符，而由段描述符再查到段地址，由此段地址和指令中的偏移地址相加，最后得到物理地址，其产生物理地址的方法如图 2.14 所示。现再解释一下关于段选择字、段描述符和描述符表。

1. 段描述符

在 80286 和 80386 保护虚地址方式下，仍用段来分配存储器，每段最长为 64K 字节，每个使用的段(如代码段，数据段，堆栈段等)都有一个段描述符，它描述了该段的属性，如段基地址和段限等，80386 的段描述符由 8 个字节组成，如图 2.14 所示。

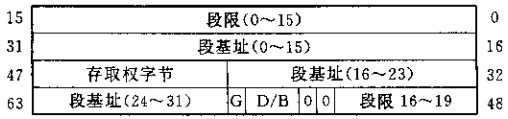


图 2.14 代码段和数据的段描述结构

图中 0—1 字节为相应段的长度，即段限长，为 16 位。2—4 字节为相应段的开始地址，称段基址，为 24 位。5 字节为存取权特性字节，对代码段和数据段，该字节有两位含义不同，

其它位同，如表 2.3 所示。

表 2.3 数据段和代码段存取权特性字节各位含义

位	名称	功能
7	P 存在位	P=1 该段在内存 P=0 该段不在内存
6—5	DPL 存取权限	分 0～3 共 4 级即 4 个特权级别
4	S 段描述符位	S=1 代码段或数据段描述符 S=0 非段描述符
3	E 段描述符分类	E=0 数据段段描述符 E=1 代码段段描述符
2	ED 延伸位(对数据段)	ED=0 向上延伸段，偏移量必须≤限长 ED=1 向下延伸段，偏移量必须＞限长
	C 验证特权位(对代码段)	C=0 当 CPL≥DPL 时，代码段只能被执行
1	W 可否写入位(对数据段)	W=0 数据段不能写入 W=1 数据段可写入
	R 可否读出位(对代码段)	R=0 代码段不能被读 R=1 代码段可以被读
0	A 访问位	A=0 未被访问 A=1 已被访问过，即段地址装入寄存器或已被段选择字指令使用过

上面提到的特权级别是指：每个任务都有一个特权级，处于就绪或封锁状态的任务，其特权级别在相应代码段段描述符中已给出，记为 DPL。只有正在执行的任务，其特权级别才在 CS 寄存器中低 2 位给出，称为当前特权级，记为 CPL。CPL 为 00 时，为 0 级特权，最高，一般操作系统核心程序定为该级，CPL 为 01 时，为次高特权，操作系统的程序可定为该级。10 为二级特权，各种语言的编译程序及应用软件可定为该级。11 为第三特权级，用户程序常为此级。

6—7 字节是扩展部分，对 80286，这 2 个字节未使用，可看作 0，对 80386，G 取值 1，表示段长度以页为单位，取值 0，表示段长度以字节为单位。D/B 是隐含长度位，当 D/B=1 时，为 32 位，当为 0 时，表示为 16 位。

2. 段描述符表和段描述符表寄存器

系统中每一个程序访问的代码段及数据段的段描述符组成了一个表，称为全局描述符表(GDT)，约占 8K 字节，用户每个任务使用的段描述符组成了局部描述符表(LDT)，每一种类型的描述符表都相应有一个寄存器，称为描述符表寄存器，全局描述符表寄存器(GDTR)含有全局描述符表的起始地址及大小，因此根据 GDTR 的值，就可确定 GDT 的地址，局部描述符表寄存器(LDTR)则含有 LDT 的起始地址和大小。

3. 选择字和段寄存器

在保护虚地址方式下，段寄存器中存放的不是段地址，而是相应段的段描述符在描述符表中的索引，通常称为选择字(16 位)，其格式如下：

索引号	TI	RPL
15 3	2	1 0

其中，RPL 占两位，为申请的权限级别，TI 占一位，是段描述符表指针。当 TI=0 时，表示使用全局描述符表，TI=1 时，表示使用局部描述符表，其余 3—15 位为段描述符表的索引号，它指出了相应的段描述符在段描述符表中的位置。

每个段寄存器都对应有一个 48 位的段描述缓冲寄存器，用来存放段地址。段限及有关的其它段属性。当一个选择字装入段寄存器时，在 GDT 或 LDT 中取段描述符，这时 80286 或 80386 便自动地从描述符表寄存器中得到段描述符表的地址，再按照选择字中的索引号，得到段描述符，而将其段属性包括段地址和段限装入到相应段的段描述缓冲寄存器中，然后由段描述缓冲寄存器中的 24 位地址与指令中的偏移地址相加而得到物理地址，这些操作均是由 CPU 自动完成的。

4. 逻辑地址转换成物理地址

上面已讲了怎样将指令中的逻辑地址转换成实际的物理地址，现用图 2.15 加以简单总结。

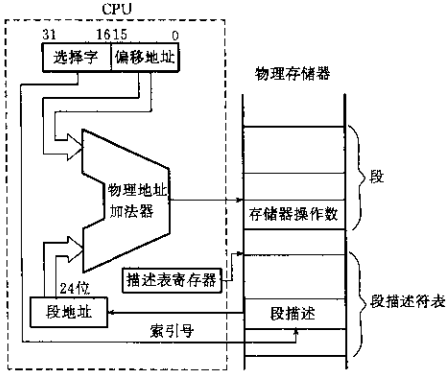


图 2.15 逻辑地址转换成物理地址

(1) 从段寄存器中得到选择字，从而得到段描述符在段描述表中的索引号。

(2) 由段描述符表寄存器得到段描述符表的地址。

(3) 根据段描述符表的地址和索引号找到段描述符。

(4) 得到的段描述符中的段地址和段限被装入到相应段的段描述缓冲寄存器中。

(5) 由段描述缓冲寄存器中的 24 位段地址与指令的偏移地址相加，而得到物理地址。

因此，在保护虚地址方式下，源程序中给出的地址并不是真正的物理地址，它由操作系统进行内存分配和管理，这个过程是动态进行的，即相应转化成的物理地址也是在变化的，并不固定。保护虚地址方式下段寄存器中的段选择字与操作系统有关，不能由程序来计算，用户想直接计算段值和内存地址是不可能的。

例如：在实方式下 53c2:107A 表示一个 20 位的物理地址，它是由段地址乘 16，然后再

加上偏移地址而得出一个真正的物理地址，即：

```
  53c20 H    段地址乘 10H
+ 107A H     偏移地址
-----------
  54c9A H    物理地址
```

而在保护虚地址方式下，53c2:107A 是可移动的，它的具体物理地址值是由操作系统通过前面所述的方法来给出的，用户无法控制。

对编程者来说，实地址和保护虚地址方式无多大差别，因为他们并不需要处理许多任务（进程），而这是由操作系统来管理的，再者，两方式下数据的存取方式都是一样的。遗憾的是，现在我们用的仍是单任务的 DOS 系统，因而保护虚地址方式下可同时运行多个程序，内存一些段受到保护不会被其它程序破坏等能力，并不能发挥出来。

正如前所述，用户在 DOS 下，若要使用扩展存储器，可以安装一个独立的扩展内存驱动程序，如 HIMEM. SYS 或 QEMM. SYS 等，用来管理系统或应用程序对扩展内存的访问，HIMEM. SYS 在 DOS 5.0 以上版本的系统盘上均有提供。也可在 CONFIG. SYS 文件中加入 HIMEM. SYS，使得 DOS 启动时，自动将大部分 DOS 安装在 HMA 中。若用户要直接进行对扩展存储器的控制使用，可用 BIOS 中断调用。BIOS 中断号为 0x15，功能号为 0x87 的中断调用，可以进行常规内存和扩展存储器之间相互传递数据，它把 CPU 设置成保护虚地址方式，这样可以直接存取 16MB 的地址，当数据传输完毕后，再将其恢复成实地址方式。

上面已介绍过，在保护虚地址方式下，每个程序的运行，首先要将其段描述符表加载到物理存储器中，根据段描述符才能将存储器中的段分配给程序，以运行程序，因此保护虚地址方式下的软件不仅有程序，还必须包含段描述表。

因此上述的 BIOS INT 15H 功能为 87H 的中断调用首先要建立一个段描述符表，其格式见图 2.16。

00H	伪描述符
08H	段描述符表的描述符
10H	源段描述符
18H	目的段描述符
20H	BIOS 代码段描述符
28H	栈段描述符

图 2.16　段描述符表

段限(0～15)					
段基址(0～15)					
存取权字节	段基址(16～23)				
段基址(24～31)	G	D/B	0	0	段限(16～19)

图 2.17　段描述符

在描述符表中，用户只需填入源段描述符和目的段描述符，其它则由 BIOS 完成，其段描述符格式如图 2.17 所示，其中 0～1 字节为源段的段限，即源数据区所占字节数，此时可按要传送的数据字节数来设置段限，第 2～4 字节为源数据段的 24 位地址，其中 2～3 字节存放 0～15 位，第 4 字节存放 16～24 位。

第 5 字节存取权字节，可置为 93H。

第 6,7 字节，置为零。

目的段描述符如上述的一样，建立好段描述符表以后，可将此表所在段及偏移地址传给 BIOS，以便找到此表，按此表得到物理地址而完成数据的传送，即：

INT 15H 中断调用，入口参数 AH＝87H，CX＝传送的数据块长度（字），ES：SI＝描述

符表段地址及偏移地址(20 位),出口参数 AH=0 调用成功,若 AH=1～3,出错。

执行上述调用时,24 位地址可选在 0～640K 基本存储器中(0～A0000H),也可选在 1～16M 即扩展存储器中(FFFFFH～FFFFFFH)。

2.6 扩展存储器的使用实例

2.6.1 程序 1

下面的一个程序是将内存中的缓冲区 buff 中内容(比如是图象信息,本程序采用装入 19 个 A 其后为 NULL 内容)送入扩展内存(1M 处),然后清 buff 缓冲区,再将传到扩展内存的内容送回 buff 中去。

在该程序中,定义了一个检查内存容量的函数 check_memsize(),在该函数中,首先调用库函数 hiosmemory(),它实际是调用 BIOS 的 0x12 中断,即检查基本存储器的容量,单位是 K 字节,由于 Turbo C 中没有检查扩展内存容量的函数,因而采用 BIOS 的 0x15 中断调用,功能号为 0x88,该调用将在 AX 中返回扩展内存的容量。在该函数中使用了 REGPACK 结构,该结构在 dos.h 头文件中进行了定义:

```
struct REGPACK{
       unsigned r_ax,r_bx,r_cx,r_dx;
       unsigned r_bp,r_si,r_di,r_ds,r_es,r_flags;
      }
```

因 intr()的说明为:

```
void intr(int n,struct REGPACK  *2preg);
```

所以将 BIOS 调用函数(intr)中的参数 r 说明成 struct REGPACK 型的。

对在保护虚地址方式下的段描述符表 LDT,该程序中定义了一个全局结构变量 moveld 来表示它,其结构类型为 LDT,即该结构中各元素对应于段描述符表中的各段。moveld 长度为 48 个字节,其中源段限为 src_size,源地址为 src_base_0_15,src_base_16_23;源存取权为 src_attr;目的段限为 dest_size,目的地址为 dest_base_0_15,dest_base_16_24,目的存取权为 dest_attr,在使用该结构前,用 memset()函数将其置零,然后设置存取权,函数 init_moveld()完成这些工作。

源地址和目的地址(24 位地址)按下式进行了计算(只举例源地址,目的地址的计算相同):

```
base_24=((long)src_seg*16)+(long)src_off;
```

然后将其放入占三个字节的 moveld 结构变量中的元素中,即:

```
moveld.src_base_0_15=(base_24&0xffff);
moveld.src_base_16_23=(base_24>>16);
```

将缓冲区内容传送到扩展内存由函数 basemem_to_extmem()来实现。

面将扩展内存中的内容再送回基本内存中的缓冲区 buff 用 extmem_to_basemem()函数来实现。

```
#include<dos.h>
```

```
#include<stdio.h>
#include<mem.h>
  struct LDT{                                    /*段描述符表结构*/
     char a[16];
     unsigned int src_size;
     unsigned int src_base_0_15;
     unsigned char src_base_16_23;
     char src_attr;
     char b[2];
     unsigned int dest_size;
     unsigned int dest_base_0_15;
     unsigned char dest_hase_16_23;
     char dest_attr;
     char c[18];
     }moveld;
main()
{
 int size;
 char buff[32000];
 size=check_memsize();                          /*检查内存*/
 printf("\n\nExtended memory=%dKB",size);
 memset(buff,'A',19);
 buff[19]='\0';
 printf("\nBUFF=%s",buff);
 if(basemem_to_extmem(FP_SEG(buff),FP_OFF(buff),0x100000l,16000)==
    -1) printf("\nFailure in moving to extended memory !");
 else printf("\nSucceeding in moving to extended memory!");
 memset(buff,0,20);
 printf("\nBUFF=%s",buff);
 if(extmem_to_basemem(0x100000l,FP_SEG(buff),FP_OFF(buff),16000)==
    -1) printf("\nFailure in moving back to extended memory !");
 else printf("\nSucceeding in moving back to extended memory!");
 printf("\nBUFF=%s",buff);
 getch();
 }
 void init_moveld(void)                         /*初始化描述符表*/
  {
     memset(moveld.a,0,48);
     moveld.src_attr=0x93;
     moveld.dest_attr=0x93;
  }

basemem  to_extmem(unsigned int src_seg,unsigned int src_off,
  long int ext_base,unsigned int size_w)
```

```
{
 union REGS r;
 struct SREGS segs;
 long int base_24;                          /* 24 位地址指针 */
 if(ext_base>16777215l)                     /* 16MB 范围检查 */
   printf("%c\nExtended Memory ERROR! Base_pointr out of range.");
 init_moveld();                             /* 初始化描述符表 */
 base_24=((long)src_seg * 16)+(long)src_off;
 moveld.src_base_0_15=(base_24&0xffff);
 moveld.src_base_16_23=(base_24>>16);
 moveld.src_size=size_w * 2;
 moveld.dest_base_0_15=(ext_base&0xffff);
 moveld.dest_base_16_23=(ext_base>>16);
 moveld.dest_size=size_w * 2;               /* 数据块长度 */
 r.h.ah=0x87;
 r.x.cx=size_w;
 segs.cs=FP_SEG(&moveld);
 r.x.si=FP_OFF(&moveld);
 int86x(0x15,&r,&r,&segs);
 if(r.h.ah! =0)
 return(-1);
 return(0);
 }
extmem_to_basemem(unsigned int src_seg,unsigned int src_off,
 long int ext_base,unsigned int size_w)
{
 union REGS r;
 struct SREGS segs;
 long int base_24;
 if(ext_base>16777215l)
   printf("%c\nExtended Memory ERROR! Base_pointr out of range.");
 init_moveld();
 base_24=((long)src_seg * 16)+(long)src_off;
 moveld.src_base_0_15=(ext_base&0xffff);
 moveld.src_base_16_23=(ext_base>>16);
 moveld.src_size=size_w * 2;
 moveld.dest_base_0_15=(base_24&0xffff);
 moveld.dest_base_16_23=(base_24>>16);
 moveld.dest_size=size_w * 2;
 r.h.ah=0x87;                               /* 功能号 */
 r.x.cx=size_w;                             /* 数据块长度 */
 segs.cs=FP_SEG(&moveld);                   /* 描述表首地址段 */
 r.x.si=FP_OFF(&moveld);                    /* 偏移地址 */
 int86x(0x15,&r,&r,&segs);                  /* BIOS 15H 中断调用 */
```

```
if(r.h.ah!=0)
return(-1);
return(0);
}
int check_memsize(void)
{
  int basesize;
  struct REGPACK r;
  basesize=biosmemory();
  printf("Base memory = %dkb",basesize);
  r.r_ds=_DS;
  r.r_es=_ES;
  r.r_ax=0x8800;
  intr(0x15,&r);
  return r.r_ax;
}
```

2.6.2 程序 2

下面的程序是将 VRAM 中(B800：0000H 开始的 16K 字节)中的内容传送到扩展内存，清屏后再从扩展内存传送到 VRAM 相同地址区域去，即复原原来的内容(即又恢复原来的显示)和前一个程序不同的是将段描述符表定义为一个 48 字节的无符号字符型数组 LD[48]。

```
#include<dos.h>
#include<stdio.h>
unsigned char LD[48];      /*段描述符表*/
main()
{
 unsigned int i,j,n;
 clrscr();
 puts("Base memory to Extended memory\n");
 for(j=0;j<60;j++)
  for(i=64;i<90;i++)
    putch(i);
 for(i=0;i<48;i++)
  LD[i]=0;                 /*初始化段描述符表*/
 i=0xb;j=0x8000;           /*B8000H 地址高位为 0BH,低位为 8000H*/
 LD[16]=255;
 LD[17]=255;               /*段限为 FFFFH*/
 LD[18]=j%256;
 LD[19]=j/256;
 LD[20]=i;                 /*源地址*/
 LD[21]=0x93;
```

```
 LD[24]=255;
 LD[25]=255;
 LD[26]=0;
 LD[27]=0;
 LD[28]=18;              /*目的地址*/
 LD[29]=0x93;
_CX=0x2000;              /*传送的数据块长度为16K字节即2000H个字*/
_ES=FP_SEG(LD);
_SI=FP_OFF(LD);
_AX=0x8700;
 geninterrupt(0x15);     /*15H中断*/
 getch();
 clrscr();
 puts("Extanded memory to Base memory");
 getch();
 LD[16]=255;
 LD[17]=255;
 LD[18]=0;
 LD[19]=0;
 LD[20]=18;
 LD[21]=0x93;
 LD[22]=0;
 LD[23]=0;
 LD[24]=255;
 LD[25]=255;
 LD[26]=j%256;
 LD[27]=j/256;
 LD[28]=i;
 LD[29]=0x93;
 LD[30]=0;
 LD[31]=0;
_CX=0x2000;
_ES=FP_SEG(LD);
_SI=FP_OFF(LD);
_AX=0x8700;
 geninterrupt(0x15);
 getch();
}
```

2.7 变量的存储方式

在C语言中,有全局变量,它能被程序中的各函数共同使用,它的生命期和程序的等同,这种变量也可叫作外部变量。另一种变量是局部变量,它仅在定义它的函数内部使用,它

的生命期仅存在于调用定义它的函数期间，当该函数执行结束后，局部变量也就消失了，因此它也称为内部变量，但对变量而言，除了依靠它在程序中的定义位置确定其变量性质之外，变量还有两个与存储有关的属性，即类型与存储种类。下面分别予以说明：

2.7.1 数据类型与存储字节数

C 语言中，对变量的类型有确定的字节数规定，由于 PC 机中规定一个地址单元存一个字节，因而不同的类型变量，为其分配的地址单元数也不一样，对于 6 种基本数据类型，其分配的字节数如下：

1. char 型

对字符(char)型变量，其长度规定为一个字节，即存放一个字符的 ASCII 码，其数值范围是－128～127。即为它分配一个地址单元。

2. short 型

对短(short)整型变量，其长度规定为 2 个字节，即可以存放一个 16 位的整型数，其数据范围为－32768～＋32767，即为它分配相邻的两个地址单元。

3. int 型

整(int)型变量，其长度按计算机本身的字长来确定，如对 CPU 为 6800 的计算机，其字长为 32 位，因而在该类计算机上运行的 C 程序，将对 int 型变量认为是 32 位长。对我们使用的 8086 系列微机，如 8088，80286，80386，80486 等微机，字长为 16 位，因而 int 型变量，其数据长度为 16 位，同 short 型，其数值范围为－32768～＋32767，占据相邻的两个地址单元。

4. long 型

长(long)型变量，其长度为 4 个字节，数据长度为 32 位，其数值范围为－2147483648～＋2147483647，它将占据相邻的 4 个地址单元。

5. float 型

对浮点型(float)，其长度为 4 个字节，数据长度为 32 位，其实数值可表示为 $(-1)^S * 2^{E-127} * M'$，其中 S 代表尾数的符号位，E 代表指数部分(占 8 位)，M 代表尾数部分(占 23 位)，这里的 $M'=1.M$。当用十进制表示时，其数值范围为±3.4E－38～3.4E＋38，相当于 6 位精度，它将占据相邻 4 个地址单元。

6. double 型

双(double)精度型浮点数变量，其长度为 8 个字节，实数值可表示为：$(-1)^S * 2^{E-1023} * M'$，其中 S 代表尾数的位，E 代表指数部分(占 11 位)，M 代表尾数部分(占 52 位)，它具有十进制的 16 位精度，其数值范围为±1.7E－308～1.7E＋308，它将占据相邻的 8 个地址单元。

对于扩充的数据类型

1. unsigned char 型

无符号字符(unsigned char)型变量，其长度为一个字节，其数值范围为 0～255，主要用于存图形字符和显示字的字模，它只占一个地址单元。

2. unsigned short (int)型

无符号短(整)型数，专门用于存放正整数，即无符号整数，其整值范围为 0～65535，它

占据相邻的两个地址单元。

3. unsigned long 型

无符号长整型，专门用于存放大的正整数，其数值范围为 0～4294967295，它占据相邻的 4 个地址单元。

4. long float 型

长型浮点数，它与 double 型有相同精度。

2.7.2 变量的存储类型

变量的数据类型确定了它存储的字节数，变量的另一属性，即存储类型决定了它在存储器中存储的位置或确切地说该变量存储的区域，C 语言规定变量的存储类型有 4 种，下面分别予以介绍：

1. auto 存储型

自动(auto)存储型变量又称为自动变量，它是最常用的一种变量的存储类型，在函数内部或复合语句内部定义的局部变量。只要存储类型是缺省的，均为自动变量，它的特点是其生命期与定义它的函数或复合语句的执行期同长，且有效范围仅在定义它的函数内或复合语句内，即该函数被调用时，或该复合语句执行时，定义的自动变量被激活，当函数执行结束后，或复合语句执行结束后，在它们之中定义的自动变量的值消失，即该变量在程序中不再起作用，它们占据的存储单元，将被别的激活的局部变量占用，在程序占据的内存区中，自动变量被分配在堆栈段中，该段是一种先进后出的存储结构，它随时在进行着压入变量值和弹出变量值的操作，即其所占内存区域在不断的被局部变量或函数传递的参数所重复占用，故所存的值均是暂时性的，关于堆栈的结构请参阅后面的栈结构一段。

自动变量在函数开始位置或复合语句开始位量定义，一般均采用缺省存储类型说明，例如：

```
    ⋮
void fun(void)
{
int a,b;
char c;
   ⋮
  { char d;
    int i,j,k;
      ⋮
  }
   ⋮
}
```

表示在函数 fun 中定义了自动变量 a,b,c,在该函数的复合语句中又定义了自动变量 d,i,j,k,其中 a,b,c 生命期与 fun 函数调用期同长，面 d,i,j,k 生命期仅存在于该复合语句执行期，这些变量的定义同下面的显示存储类型定义一样：

```
    ⋮
```

```
void fun(void)
{ auto int a,b;
  auto char c;
     ⋮
  { auto char d;
    auto int i,j,k;
     ⋮
  }
     ⋮
}
```

2. register 存储型

寄存器(register)存储型变量一般存储在计算机 CPU 的通用寄存器中,因而定义的这种类型变量存取速度快,适合于频繁使用的变量,可加快程序的运行速度,由于 CPU 中通用寄存器的数目有限,且每次可供C语言使用的通用寄存器数更有限,因而在程序中不宜大量使用这种存储类型的变量,以二三个为宜,当然若超过可用的寄存器数,也不会出错,编译程序将会将超过可用寄存器数的寄存器型变量当作 auto 变量处理,一般将般频繁使用的变量定义成 register 型的,如循环语句中的循环变量,函数的参数等,定义了寄存器变量后应马上使用,即变量的定义与变量的使用愈靠近,愈有效,这样就能保证该变量的寄存器马上得到使用,否则夜长梦多,可能被分配了寄存器的变量,在后面真正要使用时,该寄存器要作它用,而被编译程序收回,使该变量变为自动变量,下面是定义了一个寄存器变量:

```
⋮
register int i;
for (i=0;i<100,i++)
  {
     ⋮
  }
⋮
```

3. static 存储型

静态(static)型存储变量是分配在存储器中C程序占据的数据段内,对运行的C程序而言,这是一个程序所用的固定内存区域,因而静态变量的存储地址在整个程序的运行执行期间均保留,不会被别的变量占据。

静态变量可以定义成全局变量或局部变量,当定义成全局变量时,它在定义它的整个程序(或单独的一个模块)执行期间均存在,其原来的存储位置不会变化。

当定义成局部变量时,虽然在定义它的函数内或复合语句中有效,但在执行完该函数或复合语句后,静态变量最后取得的值仍保存,不会消失(因它所占存储地址不会被别的变量占用),这样当程序再次调用该函数或执行该复合语句时,该静态变量当前值就是再次进入该函数或复合语句的初始值,例如:

```
main()
{ void inc(void);
```

```
  inc();
  inc();
}
void inc(void)
{
  static int x=1;
  x++;
  printf ("%d\n",x);
}
```

其结果为:2
3

这是因为第一次调用 inc 时,静态变量 x 被初始化为 1,当该次调用结束时,该变量值变为 2,即原来内存存放该变量的值由 1 变为 2,当第二次调用该函数时,x 的初始值就是 2,而结束时,变为 3,从以上现象看来,静态变量的值是在变化的,可以这样来理解,当程序进行编译连接时,静态变量根据在函数中(或复合语句)的定义和给定的初始值被分配了存储地址和赋了初始值,当在该函数或复合语句被执行时(此时编译程序已不再起作用),若静态变量改变了其初始值,它的存储地址中的值也相应改变,这样,再次调用该函数或复合语句时,静态变量的值将是改变后的值,最初编译程序初始化的值已不存在。利用静态变量这种值的继承性,可将本次调用时的初始值采用上次调用时的结果值,如后面的链表程序中,指针的传递就利用了这种静态变量的继承特性。

由于静态变量在程序装入并执行时,自始至终占据分配给它的内存空间,因而过多地使用这种变量,会造成内存空间的浪费。当静态变量未赋初始值时,编译程序将其初始化为 0。

4. extern 存储型

外部(extern)存储型变量是指已在别的函数或模块中定义过的全局变量,而在本函数或模块中要使用它。因一个全局变量仅能定义一次,否则 C 编译程序将会报错:"duplicate variable name.",故在使用已定义过的全局变量时,在使用该变量的函数或模块前用 extern 说明该变量是外部变量(即在外面已定义过)就可以了,如有两个模块:

模块 1

```
int x,y;          /*全局变量*/
char c;           /*全局变量*/
main()
 {
   ⋮
  x=40;
 }
fun1()
 {
   ⋮
  y=60;
 }
```

模块 2

```
extern x,y;       /*外部变量*/
extern char c;    /*外部变量*/
fun 3 ()
 {
   ⋮
  x=y+10;
 }
fun4()
 {
   ⋮
  y=x/100;
 }
```

在模块 1 中，x,y,c 已定义为全局变量，且其类型已确定，而在模块 2 中，也要用到变量 x,y,c，因它们已定义过，故仅需在使用它的模块前用 extern 说明就可以了。这样将这两个模块编译连接后，生成的执行程序就可正常运行，而外部变量作为定义或说明它的各模块或函数间的公用参数而被传递使用。外部变量在编译时，其存储区域被分配在 C 程序的数据段内，这个区域在程序运行的整个期间都存在，因而外部变量的值可以被保留，传递，和重新赋值。当外部变量未初始化时，则编译程序将其初始化为 0。

注意！当定义一个外部变量为静态(static)外部变量时，它仅限制在定义该变量后的所有函数中使用，在其前的函数即使使用 extern 进行说明，也无法使用该函数，另外不同函数定义了同名的静态外部变量，则同名静态外部变量不会冲突，它们各自在定义自己的函数中起作用，而互不影响，例如：

```
模块 1                    模块 2
int a                     extern int a
static int b;             static int b;
fun1()                    fun2()
{                         {
  ⋮                         ⋮
 fun2();                   fun1()
  ⋮                        ⋮
}                         }
```

此时模块 1 中定义 a 为全局变量，因而模块 2 中要使用它而说明成 extern 型，在模块 1 中定义 b 为静态整型变量，故它的使用范围仅局限于模块 1 中，在模块 2 中又定义了一个静态全局变量 b，它和模块 1 中定义的虽然同名同类型，但决不是同一个变量，它的作用范围仅局限于模块 2 中。

一个函数也可说明成 extern 型，它的含义是，该函数在其它模块中已定义，本模块将调用它，这样对各模块编译连接后，将会生成正确的执行文件，若一个函数在一个模块中已说明成 static 型的，则它仅在本模块中可以调用，在其它模块中不能被调用。这使得不同编程者可分别编一个大程序的不同模块，不会因定义的函数同名而导致混乱。

另外，当一个大程序的不同模块使用相同的几个全局变量时，一般作法是在第一个模块中定义这些全局变量，在其它使用它的模块中用 extern 进行说明，当然外部变量太多易造成混乱且易带来副作用，因而在大的程序中应尽量少用。

第 3 章

关于 DOS 的说明及 BIOS 和 DOS 调用

3.1 关于 DOS 的说明

MS-DOS(Microsoft Disk Operating System)是一种不断发展的操作系统(现在已有 6.2 版),它由美国 Microsoft 公司研制,当 IBM 公司将其选用于 PC 机时,又称其为 PC-DOS,由于 IBM 在研制大型计算机软硬件方面的丰富经验,因而它独立地推出了 PC 机上的 DOS 4.0,IBM 推出的 PC-DOS 4.0 版由于和以前的 MS-DOS 版本和内存使用方式有些地方不同,且占用内存太多,因而没有在市场上广泛推销出来,Microsoft 为了适应 IBM 的要求,也仿照 DOS4.0 推出了 MS-DOS 4.0,同样也不成功,为此 Microsoft 又开发了新的版本,称为 MS-DOS 5,DOS 5 比 DOS 4 小得多,速度也比 DOS 4 快,且新版本增加了一些实用程序。DOS 3.3 一般认为是最好的 DOS 版本,但由于 386,486 处理器的出现,为了能充分利用它们的性能,为此开发出的 DOS 5 提供了一个图形界面及一个任务切换程序,并能将设备驱动程序,驻留程序(TSR)及 DOS 本身的大部分软件从基本存储器移到 640KB 以上的内存去。由于 DOS 5.0 许多优点,有人称它为真正的 DOS,Microsoft 于 1993 年 4 月正式推出了 MS-DOS 6.0,它在压缩存储,内存管理,防病毒方面更具特色,紧接着又推出了 6.1 版,它对 6.0 版变化不大,接着又推出了 6.2 版,它在保护磁盘数据安全性,加速对磁盘存取速度及其它方面又作了改进。

MS-DOS 是目前 PC 微机上广泛采用的一种单用户操作系统,许多 C 语言编译系统均可在它的支持下运行,Turbo C 便是其中的一种,MS-DOS 由以下五个部分组成,其中驻留在内存的三个程序分别称为 IO.SYS、MSDOS.SYS 和 COMMAND.COM,而在 PC-DOS 中则称为 IBMBIO.COM、IBMDOS.COM,但 COMMAND.COM 名不变,下面分别简单介绍一下这五个部分。

3.1.1 DOS 的基本组成

1. 引导程序

当对磁盘格式化时,FORMAT 命令将该引导程序复制到软盘的 0 道 1 扇区(对硬盘是在 DOS 区段的第一柱面的第一扇区)。

系统启动时,由 ROM 中自举程序将此引导程序从磁盘装入内存的某一地址,然后由该程序再把 DOS 的其余部分从磁盘装入内存。

2. IO.SYS(IBMBIO.COM)

它是和系统 ROM 中 BIOS(基本输入输出系统)的低级接口程序,它管理 BIOS 的活动。如果 DOS 或其它程序要使用 ROM 中子程序,则由 IO.SYS 判断它要做什么,然后调用

相应 ROM 中的子程序执行，IO. SYS 处理主存和外部设备（监视器、软盘）之间的输入输出。

3. MSDOS. SYS(IBMDOS. COM)

它是 DOS 的核心，它提供一系列系统功能子程序（可以被用户调用），如文件和目录管理、内存管理、对实时钟的访问，字符设备输入输出等，这些系统功能，只需将某一功能的特定参数装入特定的寄存器，然后通过一个调用，或软中断，就可由程序调用这些功能。

MSDOS. SYS 在系统启动时，被引导程序装入内存。

4. COMMAND. COM

它可以看作 MS-DOS 的外壳，是用户与操作系统的接口，该部分程序承担分析和执行用户键入的各种命令，并运行相应的程序。

COMMAND. COM 由三部分组成：

(1) 常驻内存部分

系统被引导时，该部分被装入内存底部，在 MS-DOS. SYS（及其数据区）之后，它包含的子程序用来处理如 Ctrl-c 和 Ctrl-Break 命令，处理磁盘输入输出错误和其它暂存程序执行的终止和退出。如发出的出错信息提示：Abort、Retry、Ignare?（放弃，再试，硬性执行），就是 COMMAND. COM 常驻部分查错而发出的错误信息。

常驻部分还包括必要时要重新装入 COMMAND. COM 暂存部分的指令。

(2) 初始化部分

在系统被引导时，该部分被装在常驻部分之后，当有 AUTOEXEC. BAT 批文件存在时，则对该文件进行处理，AUTOEXEC. BAT 文件是用户所列出的一系列命令，在系统被引导时，首先被执行，执行时这个部分即被覆盖。

(3) 暂存部分

COMMAND. COM 的暂存部分装在内存的高端，这部分内存可被其它应用程序覆盖，这部分用来显示提示符如 A>，由键盘或批文件读入命令，并使这些命令得到执行，它包含有把 COM 和 EXE 文件从磁盘装入内存并执行的程序。

COMMAND. COM 有一个再定位装入程序，它决定用户程序装在内存的何处执行。当执行用户程序时，暂存部分在 COMMAND. COM 常驻部分之后构造一个所谓的程序段前缀 PSP(Program Segment Prefix)，然后将用户的可执行程序（COM 文件）装在 PSP 的下边，离 PSP 起始位置偏移 100H 处（即 PSP 是 256 个字节长），接着将系统控制权交给用户程序，当用户程序执行完后，便由用户程序中的一条 RET 指令，使执行返回到 COMMAND. COM。若装入的是用户的. EXE 文件，则 PSP 可看作一个特殊的附加段，而. EXE 程序放在由 CS 指出的段开始处，并不紧接着在 PSP 的下边。

COMMAND. COM 接收并处理用户的三种命令：

a. 内部命令

它是 DOS 系统固有的命令，如 COPY、REN、DIR、DEL 等，这些命令的处理程序在 COMMAND. COM 的暂存部分，因此可以立即执行。

b. 外部命令

外部命令是以可执行文件的形式将程序存于磁盘上，如 PRINT、CHKDSK、BACKUP 等，这些命令被执行前，必须要将程序文件调入程序的暂存区，然后才执行，一旦外部命令被

执行完，这些程序就被丢弃，因此每次执行外部命令时，就要重新从磁盘调入到内存。

DOS 也将用户的可执行应用程序看作一条外部命令，当需执行时，则将此命令对应的程序调入内存，然后执行，完后再返回 DOS。

c、批文件命令

批文件是一个文本程序，是一个或多个 DOS 的内部命令、外部命令或批处理命令的集合，且具有扩展名 BAT，它可由用户用编辑程序如 EDLIN 或 COPY 命令建立。

5. 外部命令程序

外部命令程序是指 DOS 将由 COMMAND. COM 接收并执行的那些较长的命令程序，以可执行文件形式提供在 DOS 盘上，我们用列目录方式，可以看到这些命令程序的存在(命令名就是这些程序的文件名)。如上面提到的命令所对应的程序 PRINT. EXE，CHKDSK. EXE，BACKUP. EXE，FORMAT. COM 等。

3.1.2 关于 DOS 的启动

启动 DOS 可以用冷启动或热启动，下面分别予以介绍：

1. 冷启动

当加电开机后，若装有 DOS 的软盘在 A 驱动器中，当机器电源供电正常时，便发出电源好(POWER GOOD)信号，此时由机内时钟发生器产生一复位信号 RESET，将代码段寄存器 CS 置为 FFFF，指令指针 IP 置为 0000，于是由系统存储器 FFFF0 单元开始执行程序，即从高端存储器中的系统 ROM FFFF0 单元开始执行，从该单元开始的 5 个单元存有一条段间直接转移指令，即转入固化的初始化程序(BIOS)，该程序对系统各硬件进行检查，若有致命错误则停机，若无错，则将引导盘上的 DOS，这时 A 驱动器指示灯亮，引导过程实际上是将盘上预先定义的一个区域的内容，读到系统存储器的一个区域，接着将控制权交给该区域，该区域实际上装的就是磁盘引导程序，它将把 BIO. COM 和 DOS. COM 装入内存，这时 DOS 的 BIO. COM 将得到控制权，至此 DOS 启动完成，屏上出现要输入日期的信息，当输入完后，便出现 A 盘提示符，A>_。若 DOS 系统装在硬盘上时，不插入 A 盘，则其 DOS 启动过程类似，这时则由 C 盘启动 DOS。

2. 热启动

热启动是机器在工作时，或出现死机时，当按 RESET(机箱上的)或按 Ctrl，ALT，Del 键时重新引导 DOS 的过程，按 Ctrl，Alt，Del 热启动时将不进行系统存储器的检查，其余过程同冷启动。

3.1.3 关于 BIOS

在高端系统存储器的系统 ROM 区，将许多子程序固化在只读存储器 ROM 中，这些子程序主要用来管理各种输入和输出设备，如键盘、打印机、异步通讯，时钟等，固化管理这些设备子程序的 ROM 一般安装在 PC 机系统板上，称为系统 BIOS(即系统基本输入输出系统)，其中 BIOS 是 Basic Input Output System 的缩写。而管理磁盘、网络、图形显示器的基本输入输出系统——即磁盘 BIOS、网络 BIOS 和显示 BIOS 则一般固化在相应设备的硬件控制卡上，如一般称为多功能卡(2 串一并、软硬盘控制集于一卡)、网络卡和图形适配器卡(如称为 8900 卡和 9000 卡)，这些卡上的 ROM 所占地址空间，仍属于系统存储器，如图 2.3

所示的 ROM 扩充区，只不过器件安装位置不在系统板上罢了。

当系统加电启动时，系统 BIOS 检查这些控制卡上的 BIOS 是否存在，若有的话，则启动它们，使其执行必要的初始化操作，如设置中断向量。BIOS 在高端存储器中的地址分配情况如图 3.1 所示：

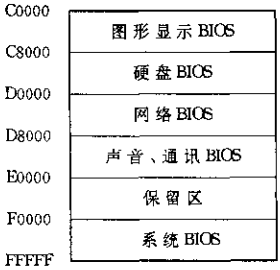


图 3.1 BIOS 地址分配

BIOS 为控制外部设备提供了一些控制的子程序，这样，用户若要控制这些设备执行某些功能时，就不用自己编写这些控制子程序，只要在自己的程序中调用相应的子程序就可以了，这就是所谓的 BIOS 中断调用，关于 BIOS 中断将在 BIOS 调用中进行介绍。另外 BIOS 为使用类型相同、但生产厂家不同的外部设备创造了接口条件，即只要它们的 BIOS 功能可以兼容，则涉及到该外部设备的软件可以在不同参数的相同类型外部设备上运行。

BIOS 的调用比较简单，即每个外部设备均有一个软中断调用号，控制该设备的不同功能，即调用不同功能的子程序，则由送入 AH 和 AL 寄存器的不同功能号来进行区别，关于这方面的内容请参阅 BIOS 调用。

3.1.4 中断向量表

它共占用 1024 个字节，处于存储器地址最低处。(0～3FFH)，存放着 0～255 号的中断向量，即 256 个中断处理程序的入口地址(段:偏移)，每个中断向量占据 4 个字节地址，系统启动后，系统的 ROM BIOS 立即把有关的中断向量读入中断向量表，有些系统保留的中断或用户可使用的中断，则可由用户进行设置，有关中断的问题请参阅中断服务程序的编写。

3.2 BIOS

基本输入输出系统(BIOS—Basic Input Output System)是操作系统的基本组成部分，如前所述它是由对系统进行硬件测试和加载操作系统的程序及对系统的 I/O 部件(如键盘、显示器、硬盘驱动器、软盘驱动器、实时钟、串行口、并行口)进行驱动服务的一些子程序组成，PC 机将 BIOS 固化在系统板上的 ROM 中，即软件固化，这样使得硬件系统板可独立于操作系统，进行改进和发展，使得不致于因为硬件的改变，而需要修改操作系统，这样硬件和操作系统各自独立，可以独立发展，只要保持 ROM BIOS 的兼容即可，因面也可以说

BIOS 是系统硬件和操作系统之间的一个软接口。

为 EGA、VGA 视频显示适配器提供服务的视频 BIOS 是在相应的视频显示卡的 ROM 中。

3.2.1 ROM BIOS 的使用

大部分 ROM BIOS 的 I/O 驱动程序是为操作系统和应用程序提供低层设备服务的，它们不同功能的 I/O 驱动程序，是通过软件中断调用来实现的，这也是我们主要讨论的部分。

还有一小部分 ROM BIOS 的 I/O 服务程序是专门用来对系统硬件进行服务的，它们是通过硬件中断调用来实现，如我们使用计算机时，进行按键动作，便产生对 BIOS 为键盘服务的功能调用。

3.2.2 视频 BIOS

PC/XT/AT 机使用单色显示器(MDA)或彩色显示器 CGA，它们的驱动服务程序固化在主机系统板上的 ROM BIOS 中，随着 EGA、VGA 显示器的出现，并进一步的有更高分辨率和显示颜色数的显示器出现，显示器显示功能愈来愈多，因而在现代的 286，386，486 等微机上，则将其驱动显示器的相应服务程序固化在显示适配器卡上的 ROM 中，这就称为视频 BIOS，当这样配置后，主机不用变化，只要插入相应显示器的适配器卡，便可实现对该类型显示器的显示控制和视频 BIOS 的功能调用。

由于视频 BIOS 和系统板上的 ROM BIOS 同属于系统 BIOS，故其功能调用形式是一致的。

3.2.3 BIOS 的调用

BIOS 中为 I/O 设备服务的程序均由一个中断服务调用号来标识，每个调用中有多个功能时，则又由功能指定的入口参数(功能号)送 AH 寄存器来确定要调用的功能，当又有一些子功能时，则可通过送 AL 或 BL 寄存器的参数值来确定，当调用时，若该功能还需一些运行的参数，则可通过对一些寄存器赋值来实现。

例如用汇编语言实现对显示器显示方式的视频 BIOS 调用，其调用号为 10H，功能号为 0H，由于该功能又包括了许多子功能(因显示方式有多种，如文本方式，图形方式，在每种方式下又有多种模式，如 40×25 文本模式，80×25 文本模式，640×350 图形模式，640×480 图形模式等等)，因而可通过送 AL 不同参数值来确定，如选定 640×480，16 色图形方式，可用如下汇编指令实现：

```
MOV   AH,0          ;功能号送 AH 寄存器
MOV   AL,12H        ;子功能参数送 AL 寄存器
INT   10H           ;进行 BIOS 视频显示方式功能调用
```

若用 Turbo C 伪变量形式可写成：

```
_AH=0;
_AL=0x12;
geninterrupt(0x10);
```

当进行 BIOS 调用后，有些调用可能还要产生一些返回信息，如调用的结果值，调用后状态变化，调用是否成功，若失败则又给出一些错误代码以提供用户分析错误原因的信息，这些均称为出口参数。如通过 BIOS 调用，读出当前光标位置上的字符及其显示属性：

```
MOV   AH,8              ;送功能号
MOV   BH,page           ;送显示的页号
INT   10H               ;进行读当前光标位置字符 BIOS 调用
MOV   BH,AH             ;读出字符的显示属性送 BH
MOV   BL,AL             ;读出的字符(ASCII 码)送 BL
```

若用 Turbo C 实现，可写成：

```
_AH=8;
_BH=page;
geninterrapt(0x10);
_BH=_AH;
_BL=_AL;
```

若定义了两个指针如 point_ch 和 point_att 取字符属性和取字符，则可写成：

```
*point_att=_AH;
*point_ch=_AL;
```

这样只要取该两个地址的内容，即可得到字符属性和字符。

3.3 DOS 调 用

PC-DOS 操作系统中有一个核心程序 IBM DOS.COM，它提供了一系列为系统服务的各种功能子程序，这些子程序分别用于设备管理(键盘、显示器、打印机、磁盘等)、文件管理(文件的建立、关闭、读、写、删除、复制、装入等)、目录操作(查找目录项、测文件大小、查找文件、文件改名、子目录删除、置/取文件属性、置/取日期时间，取当前目录路径等)及其它一些功能操作(如退出用户程序返回 DOS、终止用户程序并驻留内存、置中断向量、取中断向量、取/置日期、时间、分配内存空间、释放内存空间、取 DOS 版本号等)，这些为系统服务的功能子程序可以被用户调用，如同 BIOS 调用一样，但 BIOS 没有提供对文件、目录和内存的管理等，而文件管理是 DOS 调用的主要内容，可以说 BIOS 提供了一些最基本的输入输出服务，而 DOS 调用则提供了一些以文件形式进行输入输出服务的高层功能，如 BIOS 可以直接对磁盘扇区进行读写，而 DOS 系统调用却用文件形式对磁盘进行读写，用户无需告诉也无需知道对磁盘扇区如何操作。

有些 DOS 调用看来和 BIOS 调用有同样的功能，如 BIOS 的 INT 10H 的 0AH 功能调用和 DOS INT 21H 的 02H 和 09H 系统功能调用，同样都能在屏幕上显示字符，但 BIOS 仅限于在文本方式下，而 DOS 的显示调用在图形方式下也同样可以。再如 DOS 也提供了一些标准的输入输出服务所用的设备，但这些设备可以重定向，即通过参数可重新指定设备，BIOS 却是固定的，不可改变的，所以 DOS 调用比 BIOS 调用更灵活，功能更强，可移植性好，但速度较慢。

实际上有些 DOS 调用本身又调用一些 BIOS 的功能调用来最终完成自己的功能，好像

DOS 调用是用户应用程序和 BIOS 之间的一个软件接口一样，所以 DOS 调用执行速度要低于 BIOS 调用，因而若想提高执行速度，可直接采用 BIOS 调用。由于 BIOS 与机器硬件关系密切，而 DOS 调用是 DOS 操作系统的组成部分，因而它与 DOS 有关，与硬件关系不密切，因而具有 DOS 调用的程序在相同 DOS 系统的机器上运行不会出现不兼容的问题，但 BIOS 调用有时可能会出兼容性问题。

提供 DOS 调用功能的 IBM DOS.COM 在系统启动时，就被装入内存，而被装入内存的 IBM BIO.COM，则对 BIOS 进行管理，若 DOS 调用中要调用 BIOS 中的功能，它便根据功能要求而调用相应的 BIOS 功能。

下面示例用汇编语言进行 DOS 调用，如向打印机输出字符系统调用 INT 21H，功能号为 05，要输出的字符必须送 DL，即

```
MOV AH,05        ;打印字符功能号送 AH
MOV DL,'字符'    ;要打印的字符送 DL
INT  21H         ;进行系统功能调用
```

如用 Turbo C 语言进行 DOS 调用，可写成：

```
_AH=05;
_DL='字符';
geninterrupt(0x21);
```

3.4 BIOS 和 DOS 系统调用函数

C 语言中有些库函数(函数名前有 bios 或 dos 字样)，实际上就是调用了一些 BIOS 或 DOS 系统功能子程序，如 biosmemory()函数(得到系统基本存储器容量)实际上就是 BIOS 的 INT 12H，dosexterr()函数(得到 DOS 调用出错否的扩展信息)实际上就是功能号为 0x59 的 DOS INT 21H 系统调用，而更多的库函数则是其中用到了(隐含着)许多 BIOS 或 DOS 调用。

为了方便用户在自己的程序中进行 BIOS 或 DOS 调用，C 语言函数中提供了一些直接对它们调用的函数，现作以介绍：

3.4.1 int86()函数

大多 C 编译系统都提供了对 BIOS 调用的标准函数，其中 int86()函数就是其中之一(它也可用于 DOS 调用)，由于它们适用于以 80x86 系列处理器为 CPU 的微机，故称为 int86()，其函数说明格式如下：

```
int int86(int intr_num,union REGS * inregs,union REGS * outregs);
```

这个函数的三个参数如上述的 BIOS 功能调用一样，其中第一个参数 intr_num 表示 BIOS 调用类型号，相当于 int n 调用的中断类型号 n，第二个参数表示是指向联合类型 REGS 的指针，它用于接收调用的功能号及其它一些指定的入口参数，以便传给相应的寄存器，第三个参数也是一个指向联合类型 REGS 的指针，它用于接收功能调用后的返回值，即出口参数，如调用的结果，状态信息，这些值从相关寄存器中得到。

联合类型 REGS 在 dos.h 头文件中定义，因而使用该函数时，应将 dos.h 文件包括在程

序中，在 dos.h 中 REGS 定义如下：

```
Struct WORDREGS
  {
    unsigned int ax,bx,cx,dx,si,di,cflag,flags;
  };
Struct BYTEREGS
  {
    unsigned char al,ah,bl,bh,cl,ch,dl,dh;
  };
union REGS
  {
    Struct WORDREGS x;
    Struct BYTEREGS h;
  };
```

它表示 REGS 是一个联合类型名，该联合的成员由结构类型的 x 和 h 组成，其中 x 代表 16 位寄存器变量，它的成员用 8088 CPU 的相关寄存器名表示(注意！它们并不是真正的物理寄存器)。其中 cflag 代表标志寄存器的进位标志位，一般调用后，从该标志位是 0 还是 1 可知调用成功还是失败，结构类型 h 代表 8 位寄存器变量，即长度为字节的结构变量，它的成员与相关的 8088 的 8 位寄存器同名。union REGS 表示两个成员 x,h 共用一个内存区域，即可以是 16 位长，也可以是 8 位长，我们可以用这个内存区向有关的寄存器赋值，或取得有关寄存器的返回值，这样就和 BIOS 调用格式一致起来了，如定义一个 ax 变量并赋值如 2，可写作：

```
union REGS regs;
regs.x.ax=2;
```

它表示 regs 变量是一个 REGS 联合类型，取其 16 位的结构成员 x 中的 ax 并赋值 2。

若要对 8 位结构成员 h 中的 ah 赋值(如 2)，则可写作 regs.h.ah=2;

上述的定义与赋值过程，实际上就完成了对该变量代表的寄存器的赋值，这样就可以和用汇编语言实现的调用格式对应起来。

下面示例 INT 10H，功能号为 2 的显示中断调用，它的功能是设置光标位置，其中 DH 为光标所在行号，DL 为光标所在列号，BH 为显示光标的页号，选 0。出口参数无，下面的示例程序将在屏的 35 行第 10 列处显示出 Hello 字样，即在光标处开始进行显示。

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#define VIDEO 0x10
void movetoxy(int x,int y)
{
union REGS regs;
regs.h.ah=2;
regs.h.dh=y;
```

```
regs.h.dl=x;
regs.h.bh=0;
int86(VIDEO,&regs,&regs);
}
int main(void)
{
clrscr();
movtoxy(35,10);
printf("Hello\n");
return 0;
}
```

因 int86()函数中的第二、三个参数是联合类型的指针，故而在上面的程序中用 &regs 表示，因不要求出口参数，因而用两个同名参数表示。

当 int86()调用完成后，光标将出现在(35,10)处，因而执行 printf 显示 Hello 将在该光标处。调用完成后将返回 AX 的值，若进位标志寄存器 outregs→x.cflag 被置位，则说明调用出错。

在进行 BIOS 或 DOS 功能调用时，若需要改变段地址，即使用远指针，进行跨段调用时，int86()函数就无法使用了。在中模式以上的编译方式中，因使用远指针，会涉及到段地址的变化，为此 Turbo C 又提供了一个 int86x()函数，可解决这个问题。

3.4.2 int86x()函数

该函数的调用格式是：

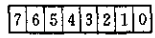
```
int int86x(int intno,union REGS * inregs,union REGS * outregs,struct SREGS * segregs);
```

函数中的前三个参数同 int86()函数中的，即第一个参数表示功能调用的类型号，第二个参数将接收功能号和入口参数，第三个参数接收调用后的出口参数，第四个参数是 int86x()函数特有的，它是一个结构 SREGS 类型的指针，结构 SREGS 类型在 dos.h 头文件中是这样定义的：

```
struct SREGS
        {unsigned int es,cs,ss,ds;
        };
```

定义该结构是为了设置和保存段寄存器的值，在 int86x()函数中，只用了 SREGS 结构中的 ds 和 es，它对应于数据段寄存器 DS 和附加段寄存器 ES，当进行 int86x 调用时，首先要设置这两个段寄存器值为要改变的值。当调用时，该函数自动将原先未改变的值保存，然后使用改变的 DS 和 ES 的值，当调用后，该函数又自动恢复原先的值。下面示例用该函数读取一文件属性。这实际上是进行了 DOS 系统功能调用。

DOS 为每一个磁盘文件建立一个目录，在建立的文件目录各项中，其中有一项是设置了该文件的文件属性字节，它的各位定义如下：



位0——当为1时，表示是只读文件
位1——当为1时，表示是隐含文件
位2——当为1时，表示是系统文件
位3——卷标
位4——子目录
位5——更改标志
位6——0
位7——0

当各位为0时，表示可读可写，利用属性字节，可以把某些文件变成只读或隐含文件，以便能进行保护性的保存。

DOS INT 21H 系统功能调用，当功能号 AH＝43H，入口参数 AL＝0 时，即为读文件属性，若调用成功，则文件属性字节在 CL 中(CH＝0)，若文件路径名非法，或文件不存在，则 AX 中返回错误代码。读文件属性时，由于要知道该文件路径名的存放地址，即段和偏移 DS：DX 的值，因此要对其进行设置，即要有第四个参数，而该参数的实际值，可通过宏 FP_SEG 和 FP_OFF 分别得到，下面是一个读文件属性的示例程序

```
#include <dos.h>
#include <process.h>
#include <stdio.h>
int main (void)
{char filename[80];
union REGS inregs,outregs;
struct SREGS segregs;
printf("Enter filename : ");
gets(filename);
inregs.h.ah=0x43;
inregs.h.al=0;
inregs.x.dx=FP_OFF(filename);
segregs.ds=FP_SEG(filename);
int86x(0x21,&inregs,&outregs,&segregs);
printf("File attribute : %x\n",outregs.x.cx);
return 0;
}
```

DOS 调用大部分是 INT 21H 系统功能调用，Turbo C 中有一个对应于 INT 21H 系统功能调用的库函数，名为 intdos(它隐含调用号 21H)。

3.4.3 intdos()函数

它的说明格式是：

int intdos(unio REGS * inregs,union REGS * outregs);

该函数的第一个参数是表示相应功能号的入口参数，第二个参数是出口参数，调用后，返回值在 outregs 的 AX 寄存器中，并将设置标志寄存器 flags，若进位标志寄存器 cflag 被置位，

则表示调用出错，此时 AX 中为错误代码，若为零，则表示成功。

由于该函数的参数中没有涉及到有关段寄存器，因而当进行要改变段的一些调用时，该函数就不能用了。

下面的示例程序是利用该函数进行 DOS 系统功能调用，删除一个当前目录下的文件（不需要改变段值）：

```
#include <stdio.h>
#include <dos.h>
int delete_file(char near * filename)
{
 union REGS regs;
 int ret;
 regs.h.ah=0x41;
 regs.x.dx=(unsigned)filename;
 ret=intdos(&regs,&regs);
 return(regs.x.cflag? ret : 0);
}
int main(void)
{
 char filename[80];
 int err;
 printf("Enter filename : ");
 gets(filename);
 err=delete_file(filename);
 if(! err)
    printf("Able to delete ",filename);
 else
    printf("NOT able to delete, this file not exist\n");
 return 0;
 }
```

3.4.4 intdosx()函数

当系统功能调用时，需改变数据段 ds 和附加段 es 的值时，就得用 intdosx()函数，它的说明格式是：

```
int intdosx(union REGS * inregs,union REGS * outregs,struct SREGS * segregs);
```

例如要删除一个文件，但该文件不在当前的 ds 段下，为此就得修改 ds 和 dx 的值，这时就可用 intdosx()函数，示例程序如下：

```
#include <stdio.h>
#include <dos.h>
int delete_file(char far * filename)
{
 union REGS regs;struct SREGS segregs;
```

```
int ret;
regs.h.ah=0x41;
regs.x.dx=FP_OFF(filename);
segregs.ds=FP_SEG(filename);
ret=intdosx(&regs,&regs,&segregs);
return(regs.x.cflag? ret : 0);
}
int main(void)
{
 char filename[80];
 int err;
 printf("Enter filename : ");
 gets(filename);
 err=delete_file(filename);
 if(! err)
    printf("Able to delete ",filename);
 else
    printf("NOT able to delete, this file not exist\n");
 return 0;
 }
```

还有一个类似于 int86 和 intdos 的库函数 intr，但它又有自己的特点。

3.4.5. intr()函数

它的说明格式是：

```
void intr(int intr_num,struct REGPACK * preg);
```

其中 intr_num 是调用号，preg 是一个 REGPACK 结构型的指针，REGPACK 结构在 dos.h 中定义为：

```
struct REGPACK
{
 unsigned r_ax,r_bx,r_cx,r_dx;
 unsigned r_bp,r_si,r_di,r_ds,r_es,r_flags;
}
```

由于用该函数进行 BIOS 或 DOS 调用时，仅用一个参数 preg 来既作入口参数又作出口参数使用，仿佛容易造成混乱，实际上进行调用时，将相应的入口参数值赋给结构变量的相应寄存器，该寄存器值又真实地赋给相应的物理寄存器，当调用结束后，当时各物理寄存器的值又赋给结构变量中的各寄存器，因此检查相应寄存器的值，便可得到调用后的出口参数值。

例如 BIOS 0x15 调用，当功能号为 0x88 时，便可从出口参数中(在 AX 寄存器中)得到当前机器的扩展内存(即 1MB 以后的内存)的字节数(以 1024 即 K 为单位)，用 intr 函数实现时，可如下编程：

```
#include <dos.h>
unsigned tast_extmem(void)
{
 struct REGPACK p;
 p.r_ds=_DS;
 p.r_es=_ES;
 p.r_ax=0x8800;
 intr(0x15,&p);
 return p.r_ax;
}
```

第 4 章

指针、函数

指针的大量使用是C语言的一大特色，使用指针可使程序简捷、明快，可大大提高程序运行速度。由于一大群数据(连续存放，比如数组)可用一指针来代表，因而处理它们就变得容易了，它给计算也带来了方便，使得对数据的寻址可以像汇编语言那样进行。由于指针就是指向的内存地址，因而使用不当，比如指针指向了内存的系统存储区，若对它们进行写操作，将会导致系统区被破坏而死机。

指针实际上就代表内存的地址，当指针初始化后，它就有了确定的代表地址，以这个地址为首地址，可能装有某种类型的数据。若定义指针为整型的，表示以此指针指向的地址为首地址，在连续的两个地址单元中存放着一个整数；若指针定义成字符型的，则表示该指针指向的地址存放一个 8 位字符型数，即 ASCII 码。

如图 4.1 所示，设定义了一个整型指针：

```
int *p=2000;
p++;
```

表示 p 指针指向 2000 单元，p++后，指针指向的地址变为 2002，即地址增量为两个地址单元。

若定义一个字符型指针：

```
char *a=2020;
a++;
```

当 a++后，指针指向的地址变成 2021，即地址增量为一个地址单元。

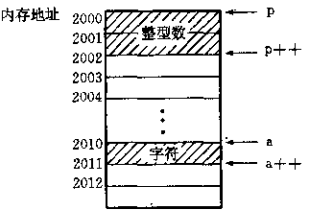


图 4.1 指针与其增量

由于我们仅将地址限于一个段(segment)中，即在中、小编译模式下，因而指针仅表示了在段内的偏移地址，段地址不再出现。

仅就指针而言，它也是一种变量，编译时也要为其分配内存地址，对近指针(near)型而言，它占据两个单元，用于存放它代表的变量的偏移地址，上述的指针就属于此种，对于远指针(far 型)则需 4 个单元，用于存放指向变量存放的段地址和偏移地址。关于这方面的内容可参阅指针的分类。

4.1 指针的赋值

在定义指针之后，只意味着在内存中为该指针变量分配了地址，但在该地址中存放着什么内容，还是未知的，即该指针指向何地址，还是未知的，因而未给指针赋值前，就使用它，尤其进行写操作，有可能导致系统环境破坏而出现死机现象，因为这时，代表指针变量的地址

中存放的数是随机的，这个数也可能恰好是系统使用的某地址。因此在使用指针前，必须赋给它正确的地址值，以指向在该地址中装的变量，由于指针指向的对象不同，因而对指针指向的方法也不同，下面简述之。

4.1.1 指向一个简单变量的指针

当指针指向一个简单变量时，这时用地址运算符 & 为指针赋值，如：

```
int x, *px;
px=&x;
```

它表示将变量 x 的存放地址赋给指针 px，因 px 定义为指向整型变量的，x 也确是整型变量。

又如 y[10]为字符型变量，若定义为：

```
char y[10], *py;
py=y;
```

也是正确的，当用该指针作为字符串地址，接收字符串时，若用 scanf()函数，可写成：

```
scanf("%s",py);
```

另外也可直接将一字符串赋给一字符型指针变量(这种方法带有不安全性，有可能放在别的变量地址中，而将原内容破坏)。

```
char *py;
py="ABCD";
```

它表示字符指针指向字符串"ABCD"，即存放该字符串的首地址赋给了 py，这样 *py，*(py+1)则分别表示存放的字符"A","B"，指针前加一"*"号，表示在指针代表的地址中的内容，由于字符串结束标志是"\0"，所以若以逐个地址输出该字符串，可以用如下形式：

```
while(*py! ='\0')
  printf("%c",*py++);
```

也可采用连续输出的形式：

```
printf("%s",py);
```

注意此时 py 前不带 * 号，实际含义是将该指针指向的字符串连续输出，直到遇到串终止符为止。

指针在函数的开始部分赋值时，称为对指针的初始化，任何类型的指针均可初始化。

4.1.2 指向数组的指针

1. 指向一维数组元素的指针

数组是一些类型相同的数据集合，每一个数组元素均占有相同长度的内存区，对数组元素的存取，可用数组下标的方法，也可用地址法或指针法，如有数组 a[6]，我们知道，数组名实际上就是代表该数组在内存中开始存放的地址，即对数组 a[6]来说，a 实际是就是存放该数组的地址代码，因此若用下标法，a[0]，代表第一个元素值；若用地址法，*(a)就是代表 a 地址中的内容，因而也就是 a[0]；若用指针法，设 p 为指针，若它指向该数组，则 *p 就是 a[0]。可用程序验证：

```
main()
```

```
{int a[6]={1,3,5,7,9,11}
 int i,*p;  ;p=a;
 for (i=0;i<5;i++)
      printf("%d",a[i]);
 printf("\n");
 for (i=0;i<5;i++)
      printf("%d",*(a+i));
      printf("\n");
 for (i=0;i<5;i++)
      printf("%d",*p++);
}
```

运行结果

```
1 3 5 7 9 11
1 3 5 7 9 11
1 3 5 7 9 11
```

在上面程序中，下标、地址、指针的关系可用图 4.2 表示：即装 a[0]的内存地址为 a，a[1]的内存地址为 a+1…。指针变量 p 中装的是 a[0]的地址，因而 *p，就表示了 p 指针代表的地址中的内容，也就是 a[0]，而 *(p+1)，就表示了 a[1]…依次类推。所以若对上面已定义的 a[]数组的第四个元素进行存取时，用指针法表示该元素可写成：

*(p+3)；

用数组下标法可写成：

a[3]；

若用地址法时，该元素可写成：

*(a+3)

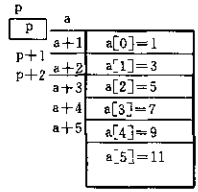


图 4.2　下标、地址、指针关系

用指针法顺序存取数组元素要比下标法和地址法快，特别是指针 p 的增量用 p++表示时更快，要注意的是数组名虽然代表一个地址，但它是一个常量，即恒定的代表一个数组的首地址，因面进行诸如 a++，a--，a+=n…的操作都是错误的。

由于数组类型不同，因而每个元素占的存储单元(即字节数)也不同，如整型数组每个元素占两个字节，实型数组每个元素占三个字节，字符型数组每个元素占一个字节，因面指针类型不同，它增 1 或减 1 时，并不意味着地址加 1 或减 1。由于数组元素都是同类型的数，因而每个数组元素均占有相同长度字节数，或相同数量的地址单元，因而指针每增加 1 或减 1，其地址相当于增加或减少数组元素所占的字节数，图 4.2 显示了指针与一维数组元素。

2. 指向二维数组的指针

由于二维数组是一个具有行列下标的数组，如 int a[3][4]，它表示有 3 行 4 列的整型数组元素，也可以认为它是由 3 个含有 4 个整型元素的一维数组组成，它们在内存的存放形式则按行存放。如图 4.3 所示，即可以将二维 a 数组看作是由三个一维数组 a[0]，a[1]和 a[2]组成，而 a[0]，a[1]和 a[2]分别又是由 4 个元素组成，这样 a[0]则代表 4 个元素 a[0][0]~a[0][3]的首地址，而 a[1]则代表 a[1][0]~a[1][3]的首地址…，因此 a 数组名代表了二维

数组的首地址，也即一维数组 a[0]的首地址，而 a+1 则代表了 a[1]数组的首地址，依次类推。若从行列上来区别，即 a 代表了第一行的首地址，a+1 代表了第二行的首地址，a+2 代表了第三行的首地址。因而若要寻址第二行第一列的元素值 a[1][0]，则其地址可写成 *(a+1)+0，若要寻址第三行第四列的元素值 a[2][3]，其地址可写成

```
*(a+2)+3
```

它等价于：

```
&a[0][0]+4*2+3
```

若用通式表示可写成：

```
&a[0][0]+4*i+j
```

它的含义是 &a[0][0]是数组首地址，当进行编译时，为数组 a[3][4]，分配了以 &a[0][0]为首地址的大小为 12 个整型数(3×4)的区间，其 a[i][j]元素的地址可用公式 &a[0][0]+4*i+j 计算，因而其值可表示为：

```
*(&a[0][0]+4*i+j)
```

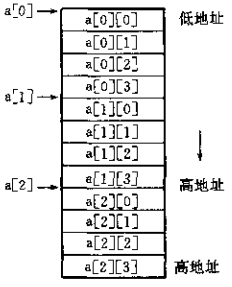


图 4.3 二维数组在内存的存放

若用指针法，则可用如下定义：

```
int a[3][4];
int (*p)[4];
p=a;
```

这样 p 指针就表示了是一个指向二维数组的指针，该数组每行有 4 个元素，若定义成如下形式：

```
int (*p)[6];
```

就表示 p 指针指向了每行有 6 个列元素的二维数组。

这样，对二维数组元素的存取也可用下标法、地址法或指针法，如对二维数组 a[][]，若要表示其第 i 行第 j 列的元素，则 a[i][j]的地址可表示成

```
*(p+i)+j;
```

而 a[i][j]的值可表示成

```
*(*(p+i)+j);
```

它表示第 i 行第 j 列的元素值。

类似的，对于指向三维数组的指针，可作如下说明：

```
int a[3][4][2];
int *p[4][2];
p=a;
```

这样当要存取 a[i][j][k]元素值时，可用指针法表示为：

```
*(*((*(p+i)+j)+k))
```

当用指针法采用 scanf()函数对二维数组等赋值时，对于整型数组，如下面的程序段：

```
int a[3][4];
int (*p)[4];
int i,j;
```

```
for (i=0;i<3;i++)
  for (j=0;j<4;j++)
scanf("%d",  *(p+i)+j);
```

当用 printf()函数输出其各元素的值时，可用如下方法：

```
for (i=0;i<3;i++)
  for (j=0;j<4;j++)
    printf("%d\n",*(*(p+i)+j));
```

但对实型二维数组采用上述方法时，程序运行时常常出错，例如有：

```
float b[4][3];
float (*p)[3];
int i,j;p=b;
for (i=0;i<4;i++)
  for (j=0;j<3;j++)
    scanf("%f",*(p+i)+j);
```

结果出现 scanf：floating point formas nat linked Abnormal program termination。

如果用一般指针则可解决此问题，如：

```
float *p,b[4][3];
int i,j;
p=b;
for (i=0;i<12;i++)
  scanf("%f",p+i);
```

这是因为二维数组在内存存放时，按行、列顺序存放，因而指针 p+i，实际上就表示了第 i 个元素的地址，若 b[0][0]为第 0 个元素，则 b[0][3]为第 3 个元素，而 b[3][2]则代表第 12 个元素，即按行列排列的最后一个元素。

用同样方法也可输出各元素的值：

```
for (i=0;i<12;i++)
  printf("%f\n",*(p+i));
```

4.1.3 指向函数的指针

函数是具有执行特定功能的子程序，编译后，它的执行代码分配在代码段，而其参数及变量则在堆栈段，因而主程序调用函数时，实际上就是将程序执行地址转移为函数在代码段的入口地址去执行，即每个函数都有一个在代码段的确定入口地址，依此程序执行，当遇到返回指令时(表示该函数结束)，程序便返回到该函数调用者的断点程序处，又继续执行，既然函数有确定的入口地址(实际上函数名就代表了它的入口地址)，因而可以用指针指向它，这个指针又称为函数指针，例如：

```
main()
{int func();          /*说明 func 是一个返回整型数的函数*/
 char *p*
```

```
p=(char *)func;  /*将函数的地址赋给指针*/
```

说明了 p 是一个指向字符型数据的指针，而

```
p=(char *)func;
```

则将 func 的地址(强制转换为字符型)赋给指针 p，即将段内偏移地址赋给了指针 p，当然更一般的形式是采用函数指针的说明形式，如：

```
main()
{int func();
 int (*p)();
 int c;
 p=func;
    ⋮
```

其中 int (*p)()表示 p 是一个指向整型函数的指针，而 p=func，则将函数地址赋给该指针。

注意！要将函数指针的说明形式和 int *p()相区别，该形式表示 p()函数是一个返回整型数据指针的函数，即函数返回一个指向整型数据的指针。

当调用 func 函数时，若用指针方式可写作

```
c=(*p)();
```

其中第一个括号表示执行函数入口地址中存放代码，并继续往下执行之，第二个括号表示调用时的参数，若无参，则括号中是空的，若有参数，则应将参数写在第二个括号中，例如若有参数 a，b，则应写作

```
c=(*p)(a,b);
```

下面是一个采用指针形式调用函数的例子，在程序执行过程中，有时根据当时的条件要调用不同的函数时，可以设计一个总控函数，其中设定一个函数参数为函数的指针，这样根据当时条件，只要将要调用函数的指针传给总控函数，它即可以执行选中的函数，并完成附加的操作。

例：该程序检查输入的两串数或字符的相等性，当判断输入的是两串数时，便将比较两数相等性的函数指针传给控制函数 check()，否则便将比较两字符串的函数(这是系统库函数)的指针传给控制函数。

```
#include <string.h>
#include <stdio.h>
main()
 { int strcmp();
   int numcmp();
   void check();
   char S1[80],S2[80];
printf("input string 1 : \n");
gees(S1);
printf("input string 2 : \n");
gets(S2);
```

```
if(isalpha(*S1))
    check(S1,S2,strcmp);
  else
    check(S1,S2,numcmp);
}
check(a,b,cmp)
char *a,*b;
int (*cmp)();
 }
  printf("testing for equality\n");
  if(*cmp)(a,b)
      printf("equality\n");
      else
      printf("not equal");
 }
int numcmp(a,b,cmp)
char *a,*b;
int (*cmp)();
  { if (atoi(a)==atoi(b))
    return 0;
    else
    return 1;
  }
```

4.1.4 指向结构的指针

指向结构的指针又称为结构指针，它表示指向结构的首地址，结构指针的增量，是定义的结构的数据长度。结构指针在程序中的说明形式是：存储类型 struct 结构名 *结构指针名

如 info 是一结构变量

```
struct records                    /*定义结构变量*/
    { char name[40];
      char street[40];
      char city[20];
    }info;
struct records *pi;               /*定义结构指针*/
pi=info;                          /*结构指针赋值*/
```

当要对结构变量的某成员项进行存取时，可用指针表示如下：

pi→name　　即指针指向 info.name 项

等效于(*pi).name

pi→street　　即指针指向 info.street 项

等效于(*pi).street

pi→street，实际上相当于 pi 指针值再增加 40 个地址单元，同样，pi→city 即表示指针指向 info. city 项，也可表示为：

```
(*pi).city
```

它相当 pi 指针再增加 40 个地址，即比原始的值再增加 80 个地址单元，如图 4.4 所示。

当结构变量成员项中又包含指针变量时，例如：

```
struct address
    { char name[20];
      char sex;
      int age;
      char *addr;
    }info;
struct addrees *p;
p=&info; info.addr="Street peace 237";
```

在结构成员项中又出现指针项 char *addr，此时当用指针 p 表示该项时，可写成：

```
p→addr
```

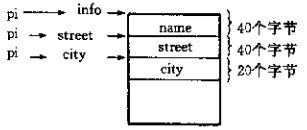


图 4.4 结构的指针

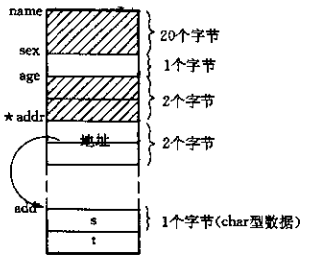


图 4.5 结构成员中的指针指向的数据

此时得到的是一指针，当要得到该指针指向的 char 型数据时，可写作：

```
(*(p→addr))
```

如图 4.5 所示。

还有一种动态数据，它们是根据需要而在程序中设置存储区的，并不是在程序开始时预留的，它占据的内存空间，则由内存分配函数 malloc()或 calloc()分配，关于这方面内容可看动态存储例——链表。如定义一个结构：

```
struct add{
    char name[40];
    char street[40];
    char city[20]
    char zip[10]
    }info;
struct add *pointer;      /*定义一个指向该结构的指针*/
```

再用内存分配函数 malloc 分配一个区域，并将首地址赋给指针

```
pointer=(struct add *) malloc(sizeof(info));
```

这样 info 结构的信息，则可按指针 pointer 指出的首地址开始顺序存入该内存区，如

```
pointer→name="Epsilon"
pointer→street="Beacon"
pointer→city="Boston"
pointer→zip="2770"
```

若再由 malloc()函数分配一次，则可由它给出的地址开始，存入第二个结构的各项信息。

4.2 指针数组

指针数组指一系列指针变量的集合，正如通常的数组是一系列有序的同类数据的集合一样，指针数组中的数组元素是指针变量，它们指向同类型的数据，定义指针数组与定义普通数组一样，如定义两个不同类型的指针数组：

```
int *a[2];
char *p[4];
```

前者表示定义了一个指向整型数的指针数组，它由两个指针变量组成。后者表示定义了一个指向字符型数据的指针数组，它由 4 个指针变量组成，如图 4.6 所示。又假设有两个二维数组：

```
int b[2][3];
char c[4][3];
```

由于二维数组可以看作由一维数组所组成，如 h[2][3]，可看作由两个一维数组 b[0]，和 b[1]组成，其中 b[0]和 b[1]要看作是一维数组名，它们分别又由三个数组元素所组成，即对 b[0]一维数组来说，它由 b[0][0]，b[0][1]，b[0][2]所组成，而对数组名为 b[1]的一维数组而言，它是由 b[1][0]，b[1][1]和 b[1][2]三个数组元素组成。

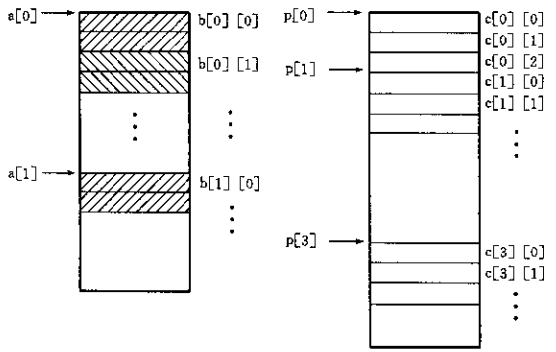


图 4.6 整型和字符型指针数组的指针变量

对字符型数组 c[4][3]，可以同样看作由四个一维数组所组成，即 c[0]，c[1]，c[2]，c[3]，它们分别又由三个数组元素组成。

由于数组名就是存放该数组的首地址，因而可以用它代表地址，赋给指针变量。由于这些指针变量均指向同类型的相关的数组元素，因而它们可看作是属于一个指针数组，例如将上述的 b[2][3]二维数组让其和指针数组 a[2]相互关联起来，即让 a 指针数组各元素指向相应的一维数组，这些一维数组又组成了二维数组，即：

a[0]=b[0]；或 a[0]=&b[0][0]；

a[1]=b[1]；或 a[1]=&b[1][0]；

将 p 指针数组指向 c[][]字符数组相应元素，即

p[0]=c[0]　或 p[0]=&c[0][0]；

p[1]=c[1]　或 p[1]=&c[1][0]；

⋮

p[3]=c[3]　或 p[3]=&c[3][0]；

这样 a[]和 p[]指针数组中各元素代表了指针，它们分别指向了 a[][]和 p[][]数组的各行（即代表各行首地址）。

指针数组常用于指向一组长度不同的字符串，这些字符串可能是一组信息，当对不同下标的指针数组元素进行存取时，便可得到相应指向的字符串，如定义一个函数：

```
MSG(num)
int num;
{static char *m[]={
        "Operation error\n",
        "Maching failure\n",
        "Illage Data\n",
        "Value out of range\n"
        };
printf("%s",m[num]);
```

其中参数 num 是由主程序中根据程序判断得出的信息号码，当调用该函数时，便可根据信息号码显示出不同长度的提示信息来，这比将信息变成相同长度的字符串放在二维的字符串数组中要方便多了（这不但节省了内存，也不用将字符串用填空格的办法变成同样长度）。

4.3 二级指针

一个指针指向了另一个指针，则这个指针称为指针的指针，或称二级指针。这类似于汇编语言的间接寻址一样，即二级指针所指向的地址中装的仍是一个地址值（又一指针），它指向另一地址，而指向的这个地址中才装的是数据。

例如：

```
#define NULL 0
main()
{                              /*定义一个二级指针*/
```

```
    char * *p;
    static char *m[]={"Operation error\n",
                      "Mach failure\n",
                      "Illage data\n",
                      "Value out of range\n",
                      }";"
p=m;
while(* *p! =NULL)
 printf("%s",*p++);
}
```

该程序将输出各字符串，直到 p+4 后，因该指向的地址单元中装的是空字符，故输出停止。二级指针 p 的指向情况如图 4.7 所示。

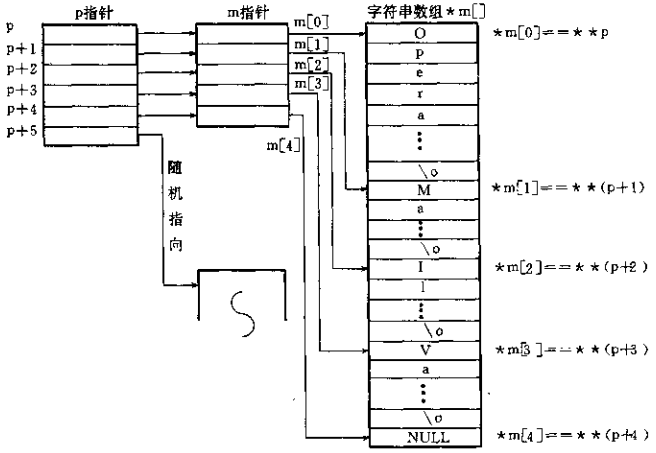


图 4.7 二级指针指向图

在上面定义的字符串数组中，常给最后一个字符串赋以空串，这是 C 程序中常用的手法，用它可以控制读字符串数组时结束的条件，尤其在使用指针数组，而数组元素个数又不定时，如上例中用了 while(* *p! =NULL)来控制输出字符串数组内容。有人认为在存放字符串数组的连续内存空间中，字符串数组结束后的区域中一定存放的是空(NULL)字符，即 0。实际上这不正确，它的内容是随机的，这是因为编译系统是根据数组实际大小分配连续的空间，接着这存储空间之外的地址单元中的内容是随机的，不定的，用户也是不可知的(当然可以用地址法，将其读出，但可以发现，每运行程序后，其内容总是在变的)，还有一个原因，就是若字符串数组中最后不放一个空串，即数组元素个数少 1，则编译系统为指向它的二级指针分配存放地址的空间时，仅可用到 p+3，面对于 p+4 开始存放指针值的地址单元，存放的值也是不定的，它并不指向指针数组最后一个指针地址的下一个地址(因这个地址内容也是随机的)，因而不可能一定出现 * *p++==NULL 的现象，故而无法控制程

序的结束，图 4.7 示出了当 p+5 后，该二级指针的指向将是随机的，并不指向指针数组 m[] 的后面地址。

在用二级指针输出字符串数组时，可用 printf("%s", * py)，其中 py 是定义的二级指针，而 * py 表示指针数组的首指针元素地址，它表示从该地址开始输出字符串，它和一个指针数组各指针元素指向一字符串数组各元素的表示方法一样，如下例中程序所演示的。但在输出指向数值类型的数据时，则要用二级指针取数据的形式，即 printf("%d", ** p)，p 为指向一个整型指针数组的二级指针，而指针数组又指向一个整型数组，故 ** p 表示指针数组指针元素指向的地址中的内容，即整型数组元素值。

```
#define NULL 0
main()
{
 char **py, *y[]={"operation","data",ascii"," "};
 int i;
 i=0;
 py=y;
 while(**py! =NULL)
  {
  printf("%s\n", *py++);
  printf("%s\n",y[i++]);
  }
 }
```

采用二级指针可大大加快程序运行速度，节约内存空间，比如上面指针数组所指向的四个字符串，若不采用指针的方法，则必须用一个二维字符串数组来表示这四个字符串，即有 m[4][20]，可看成有四个一维数组 m[4]，每个一维数组由 20 个字符组成（因每个数组元素均应有同样长度的字符串，因而该数组元素字符串长度以最长的字符串为准，故选 20，其它不够长的，存储时，将自动被填以空格），这样对于较大字符串数组而言，内存浪费将是很大的，若采用指针数组，将大大节省内存空间，因这时字符串存放时，将连续存放（如图 4.7 所示），字符串并不要求定长。

二级指针常常应用于二维数组中，如上例所述，可将一个二维数组用一个指针数组指向它，然后再由一个二级指针指向该指针数组，这样以后对二维数组的使用、处理，只要用定义的二级指针表示其实体即可，因而除了上述可节约内存外，还使得程序简捷，不易出错，且处理是直接对存放元素的地址单元中的内容而言，因面节省了大量的寻找数组元素的时间。

4.4 指针型函数

当一个函数的返回数据是指针时，我们称这种函数为指针型函数，由于指针指向的数据类型不同，因而指针型函数也有不同类型，如有指针函数说明：

```
int *f(int a,int b);
```

它说明带有两个参数的函数 f 将返回一个指向整型数的指针。

```
char *c();
```

它说明函数C将返回一个指向字符型数据的指针。指针函数通过return语句返回一个地址值，即指针。

例如有两个字符串，用字符串数组表示为s1[]，s2[]，要求将s2[]接在s1[]的后面，实现这种功能的函数在C语言标准库函数中有，名为strcat()，现在我们编一个这样的指针函数，

```
char *strcat(char *s1,char *s2)
 {
   char *p;
   for (p=s1; *p! ='\0'; p++);
     do
      {*p++=*s2++;
      }while(*s2! ='\0');
   *p='\0';
   return s1;
 }
main()
 {
   char d[20],s[20];
   char *result;
   strcpy(d,"This a");
   strcpy(s,"String");
   result=strcat(d,s);
   printf("strcat(): %s\n",result);
 }
```

该程序中两个字符串地址作为参数传给函数strcat()，而该函数执行结果返回一个指针，最后printf输出该指针指向的字符串。

C语言中许多库函数都是指针函数，它们执行结果返回一个指针。我们在编写指针型函数时要注意不能将函数内部使用的局部变量的地址作为指针返回，因为局部变量的寿命与定义它的函数生存期同长，当函数结束后，它的局部变量也消失，因而其地址将被释放，它所存内容将是随机的，因此返回这个地址是无意义的。

4.5 指针的分类

由于PC微机的存储器地址是由段地址和偏移地址组合而成的，每个段不能超过64K字节地址，因而同一个段内的地址存取，仅用偏移地址就可实现，因段地址寄存器所存的段地址是不变的(自始至终指向该段)，当用指针时，只16位就够了(即仅表示偏移地址)，这类指针，称为近程指针，是near型的。当要在另一个段内取数据时，就要跨越段，即要指明存取段的段地址和偏移地址，这时段地址寄存器所存段地址要改变，因此在使用指针指向另一个段内地址时，就要用32位表示(即段地址：偏移地址)，这类指针，称为远程指针，是far型的。另外还有一种巨型指针(huge)，当地址超过64K的段容量时，自动修改段地址，使其指

向另外的段。由于我们编程序时，只关心数据的存取，代码的分配和执行由编译程序根据内存模式进行了管理，用户一般不需干预，因而我们下面介绍三类指针时，只介绍指向数据的指针。

4.5.1 近程指针(near 型)

近程指针是16位的指针，它只表示段内的偏移地址，因而用近指针只能对64K字节的数据段内地址进行存取，这时段地址是固定存放在DS段寄存器内。近指针可用near进行说明，如定义一个指向字符型数据的近程指针p，可写作：

```
char near * p;
```

near型指针最大值不能超过64K(即65535)，例如：

```
p=(char near * )0x10000;
```

由于p的最大值只能到0xffff，即65535，当加1变成65536，即0x10000时，编译便出错，因它只能是16位指针，地址位不能大于16，如用下面方式：

```
char near * p;
p=(char near * )0xffff;
p++
printf("%u",p);
```

虽然p赋值为0xffff，编译不会出错，但p++后，结果为0，这种现象叫做折回，即超过64K的地址，重新又从0开始算起。

用近指针进行存取操作时，因无需进行段地址的计算，因而运算速度快，在小数据内存模式下，由于数据仅在一个数据段内存取，因而采用near型指针，定义指针时，可不加说明。

4.5.2 远程指针(far 型)

远程指针是32位的指针，它表示段地址：偏移地址，如定义远程指针p指向B500段的2号地址，即B500：0002，则可写作：

```
char far * p=(char far * )0xB5000002;
```

因而用远程数据指针可以指向任何的数据段内地址，在大数据内存模式下，当跨越段进行数据存取时(即存取超出64K的数据)，可以采用远程数据指针。当使用far型指针时，因要进行跨段寻址，数据段寄存器的值要更换，因面加长了程序的执行时间。

另外远程指针的值在进行加减时，只有偏移地址部分进行运算，段地址不参加运算，即段地址不变，因而也会出现折回现象。在进行far指针操作时，特别要注意的是，由于PC机内存独特的段结构形式，可以将一个实际的物理存储地址划归在不同的段内，因为在缺省段边界形式设置时，段地址是以1M内存空间内的每一个16倍数值开始作为边界的，即1M空间可以划分出64个段来，如23B0：0004，23A1：00F4，2300：0B04，…，分属在不同的段内，但其实均代表同一个物理地址23604。因要得到绝对物理地址，需将段地址左移4位(对二进制而言)，然后再和偏移地址相加即可。虽然不同的远程指针均可指向同一个物理地址，但由于这些指针的值不同，因而在进行指针的比较运算时，它们不能认为是相等的，如设上面所述的三个指针分别定义为a,b,c，因均指向同一个实实在在的物理地址，但它们的值不相等，因而在进行关系运算时，如进行相等比较：

a==b;

b==c;

a==c;

结果其值均为假，这是因为在进行相等(==)或不相等(!=)运算时，是将指针的 32 位值作为 unsigned long 数据来处理，而不是将其换算成地址再比较，而当进行大小比较时：

a<b;

b<c;

a<c;

则为真，这是因为在进行>，>=，<，<=等关系运算时，只对指针的偏移值进行比较运算，段地址部分并不参加运算。

4.5.3 巨指针(huge)

巨指针如同远指针一样，也为 32 位，它也表示段地址：偏移地址，但与 far 指针不同之处是，当 huge 指针的偏移地址超过 64K 时，并不会折回，而是重新改变段地址，这说明 huge 指针可寻址内存内的任何地址。huge 指针又称为规格化的指针，所谓规格化，是这样规定的，即将指针值转换为 20 位地址(二进制)，其中左边的 16 位作为段地址，右边的低 4 位作为偏移地址，如 2F84：0532 将其转换为 20 位地址(现以 16 进制表示)为 2FD72，这样高 16 位为 2FD7，它便是段地址，低 4 位为 2，这就是偏移地址，因而规格化的指针地址可表示为 2FD7：0002，它是由下式算出的：

```
段地址左移 4 位   0010  1111  1000  0100  0000=2F840H
偏移地址          +     0000  0101  0011  0010=0532H
                  ------------------------------------
物理地址          0010  1111  1101  0111  0010=2FD72H
```

再比如有 514D：1235 指针的值，它表明段地址为 514D，偏移地址为 1235，huge 指针将其规格化后，就变为 5271：0005。可以看出，所谓规格化，实际上就是统统化成物理地址，即真正代表的内存绝对地址。又如在 far 指针中介绍的例子，23B0：0004，23A1：00F4，2300：0B04 均代表同一物理地址，但当用 far 指针表示后，对其指针值进行关系运算，就会得出不相等的结论，而用 huge 指针表示这三个地址，进行指针值关系运算时，会得出相等的结论，因 huge 指针将其规格化后，结果均变为 23B0：0004。因此可以用 buge 指针进行==和!=的关系运算，也可进行>，>=，<和<=的关系运算(因 huge 指针在这些关系运算时，将指针值规格化后，采用 32 位比较)。

huge 指针的规格化是基于这样的事实，如上面提及，PC 微机 1M 存储器的段结构形式，在缺省方式下进行分段时，可以从二进制 16 倍数为段边界的，即每隔 16 个字节为一个新段的开始，因而对相同物理地址的指针，规格化后，地址值是相同的。

下面定义了一个 huge 指针并赋了值：

```
char huge *p;
p=(char huge *)0x 235B0012;
```

由于 huge 指针要进行规格化，即要调用库中的专用程序，因而其运算速度也比 far 指针慢，更比 near 指针慢。

4.6 函　数

函数是C语言程序结构中的基本构件。一个大的C程序，可以由许多函数构成，即一个C程序可由许多砌块——函数堆砌而成，用这种方法构造的程序易读易调试，适合集体编制。

C程序最少由一个main()主函数构成，当需要多种功能时，可以将主函数设计成总控函数，而将实现的各种功能分别由定义的许多函数来完成，每个函数具有独立的一种功能或多种功能，这样，程序运行时，根据需要，由总控函数来调用相应的函数，完成要求的功能，当不断调用各函数的最后结果，也就是整个程序的最后结果。

如要求求出一个数组a[]的各元素和，各奇数下标元素和(第一个元素a[0]认为是第一个奇下标)，数组元素中的最大者，数组元素的平均值，编程时，可将各要求的功能分别用函数实现，如这些功能要求可用arr_add()函数求和，odd_add()函数求奇数元素和，arr_max()函数求元素中的最大值，而arr_aver()函数则用来求各元素的平均值，下面所列这个程序，就可完成这些要求。

在该程序中，main()函数成为总控函数，它分别一次次调用相应的函数，调用完毕，各相应要求的功能也一一实现。

```
#define N 12
main()
 {
  float a[]={1.5,4.6,7.8,9.2,7.4,6.5,7.0,9.8,8.7,
       21.7,15.6,34.3};
  void arr_add(float *p,int n);
  void odd_add(float *p,int n);
  void arr_max(float *p,int n);
  void arr_aver(float *p,int n);
  int n=N;
  float *pt=a;
  arr_add(pt,n);
  odd_add(pt,n);
  arr_max(pt,n);
  arr_aver(pt,n);
 }
 void arr_add(float *p,int n)
  {
   int i;
   float sum=0;
   for(i=0;i<n;i++,p++)
    sum=sum+*p;
   printf("the sum of %d elements is : ",n);
   printf("%8.2f\n",sum);
```

```
}
void odd_add(float *p,int n)
{
 int i;
 float sum=0;
 for(i=0;i<n;i=i+2,p=p+2)
  sum=sum+*p;
 printf("the sum of odd delements is : ",n);
 printf("%8.2f\n",sum);
}
void arr_max(float *p,int n)
{
 int i;
 float max;
 max=*p;
 for(i=0;i<n;i++,p++)
  if(*p>max)max=*p;
 printf("the maximum of %d elements is : ",n);
 printf("%8.2f\n",max);
}
 void arr_aver(float *p,int n)
 {
  int i;
  float sum=0,ave;
  for(i=0;i<n;i++,p++)
   sum=sum+*p;
  ave=sum/n;
  printf("the average of %d elements is : ",n);
  printf("%8.2f\n",ave);
 }
```

大的应用程序，尤其商品程序，除了有美丽的外包装之外(即程序开始运行时，出现的广告画面)，还应有良好的人机界面，通常有一个主菜单，它由 main()函数控制，主菜单项下可有多种功能可供选择，每种功能可能又分成许多子功能，常以子菜单形式提供，每个子功能可能就是由一个函数来完成的，这样用户选择了子菜单项，实际就调用了相应函数来执行。比如 TC 集成开发环境，就是一个很好的例证，只不过它们是由汇编语言编的。

对于自顶向下的模块化程序设计，函数是最小的单位，即将目标要求逐次分解，由粗到精，由大到小，最终还是落实到各个函数的定义上。

4.6.1 函数的说明

函数在使用之前要进行说明，如同变量在使用之前也要进行说明一样，虽然对于返回值为整型或字符型的函数，可以在调用函数中不加说明，但为了程序易读，易调试，还是作说明为宜，一般习惯，总是将主函数放在前，函数在主函数后分别一一定义，但更好的方法是：可

将函数说明集中放在整个程序的开头(即主函数前),这样就不必在每个调用的函数中,对被调函数进行说明了。

函数说明形式是:

函数类型　函数名(数据类型 形参,类据类型 形参…);这是ANSI规定形式,有的书称为现代定义方式,对于有些TC的库函数,由于它们均在与其有关的头文件中进行了说明,所以虽然在程序中可以对其不加说明,但必须要将其有关的头文件用#include进行包含,例如标准输入输出函数的说明均在stdio.h头文件中,因而使用它们时(printf,scanf除外),必须在程序的开头用#include<stdio.h>或#include"stdio.h"进行包含,前者表示仅在系统设定的标准目录中去寻找头文件,后者表示首先在当前的目录下去寻找头文件,若没有时,再到系统设定的标准目录中去寻找。

下面是一个典型的函数说明例:

```
#include<stdio.h>
int prn(int x,int y,int color,char * pt);
void scr(int i,int j,int color);
float add(float a, float b);
main()
{
  ⋮
}
```

4.6.2　函数的类型与定义

函数类型指函数执行后的返回值的类型。一般来说函数有三种,一种是对由函数参数传过来的实参值进行运算处理,并返回处理的结果,它们可以定义成各种基本数据类型,也可以是结果的指针。

另一种函数是执行某些特定的操作,如磁盘操作。该函数返回操作后的信息(如成功否),常用代码或提示信息表示,它们也是有类型的。

还有一种函数没有返回值,它们常常用来执行一种过程,因而类型说明为void型(空)。

函数的类型说明就是指该函数执行后,将返回什么值。

函数的定义是指该函数要求的形参和其具体执行的功能,它由函数头和函数体组成,即

函数类型　函数名(数据类型 形参1,数据类型 形参2,…);

```
{
  函数体
}
```

函数定义可以放在程序的开头(即main()之前),也可放在其后,一般多放在main()函数之后.函数体是由语句、命令、库函数、函数调用组成,它又是一个小程序,但函数中不能再含有另外的函数定义,任何一个函数,都可以被程序中的另外函数调用,甚至也可再被自己调用,调用与函数在程序中的排列位置无关。

一个函数也可没有函数体,这常用于调试,这样的函数称为空函数,即

函数名()

```
{
}
```

它仅起到了在调用函数中占据一个位置。尤其在模块设计时，调试单独模块常采用这个手段，这可减少调试点，即假设在该处调用函数不会有错。

函数的返回值一般用 return 语句返回，该语句也使函数的执行结束，若无返回值，则靠程序执行到函数的末尾"}"处而结束。函数的结束，意味着要进行堆栈的弹出操作，即弹出程序因调用该函数而被中断的地址，当指令指针寄存器接收到中断了的地址时，程序便马上接着在中断点开始往下执行。

若函数有返回值，它是通过送到 AX 寄存器而带回到被调用处的，若返回值为 4 个字节长度，则用 DX：AX 返回。超过 4 个字节的，如 struct 型的，则放在静态变量中，而返回这个变量的地址—即指针。对于要返回多个值的情况，也可由函数返回指针来实现，有关这方面的内容将在参数传递中介绍。

对于返回指针的函数要特别加以说明，因为指针是指存储某个类型数据的地址，因而对这类函数类型说明，必须说明成指针指向数据的类型，以便在进行指针运算时，能得出正确的地址值，如指针是指向一个 char 型数据的，则指针加 1，就意味着地址加 1，因 char 型数据只占一个字节地址，而若指针说明是 int 型的，即它指向一个整型数，则指针加 1，就意味着地址加 2，因 int 型数据占 2 个字节地址。所以当我们对返回指针的函数进行了正确的说明，即函数类型指出的是返回指针的类型，这样编译程序才能正确地编译出指针的运算结果。

例如，下面的函数将返回一个指针，这个指针指向返回的一个字符，这个字符是要在已知的一个字符串中查找的，故必须定义此函数是 char 型指针函数。

```
char *match(char c,char *s);
 {
  int count;
  count=0;
  while(c! =s[count]&&s[count]! =NULL)
  count++;
  return(&s(count));
 }
```

当调用它时，主函数可设计成：

```
char *match();
main()
 {
  char s[80], *p,ch;
  printf("please input string");
  gets(s);                          /*输入一个字符串*/
  printf("please input character");
  ch=getchar();                     /*得到要查找的字符*/
  p=match(ch,s);                    /*调用指针函数*/
  if(*p)
   printf("%s",p);
```

```
    else
      printf("no match found");
  }
```

由于该函数正确地说明成字符指针型的函数，因而在编译时，count++实际上被编译成 s 串的地址加 1，因一个地址装一个字符。这样若找到这个字符的地址，将打印出以找到字符为开始处的字符串，否则打印出 no match found。

4.6.3 函数的调用

1. 通常的函数调用

函数是通过函数调用来执行的，通常当需要函数的计算结果值时，往往将其作为表达式中的一个成份，当仅作为一个执行过程，或传回某个信息，常将其写成一个独立调用语句。

前者如：

```
c=3*max(a,b);
```

设 max(a,b)是调用求 a,b 两者找出大者的函数，该赋值语句右边的表达式，实际上就调用了 max()函数，又如：

```
hzprint(x0,y0,RED);
```

设 hzprint() 是一显示汉字的函数，它只表示执行一种显示汉字的过程，因而可写成一独立的调用语句。

2. 用函数指针调用函数

函数地址如同数组的地址一样，其名字就代表了其地址。函数指针是通过定义一个函数类型的指针而得到，将函数地址赋给它，即可用该函数指针去调用该函数，如定义了一个函数：

```
double mult(double x)
  {
    double a=2.0, b=-3.0, c=5.0;
    return ((x*a)*x+h)*x+c;
    }
```

当用主函数调用它时，可用函数指针去调用它，即在主函数中定义一个函数指针：

```
double (*f)();
```

然后将函数地址赋给这个指针，即：

```
f=mult;
```

这样只要在 main() 函数中写入

```
f(x0);
```

即表示了函数调用，其中 x0 为实参，主函数如下：

```
#include <stdio.h>
main()
  {
    double x0;
    double delta=1.0;
```

```
    double first=0.0;
    double last=10.0;
    double (*f)();
    double mult(double x);
    f=mult;
    x0=first;
    while(x0<=last){
        printf("f(%Lf)=%Lf\n",x0,f(x0));
        x+=delta;
    }
}
```

有时也可直接用函数指针去调用，例如有一交换两个变量值的函数：

```
void swap(int *x,int *y)
    {
      int temp;
      temp=*x;
      *x=*y;
      *y=temp;
    }
```

用主函数调用

```
void swap(int *x,int *y)
main()
  {  int x,y;
     x=10;
     y=25;
     (*swap)(&x,&y);          /*通过函数指针调用函数*/
     printf("x=%d,y=%d",x,y);
  }
```

有时候，在程序中要根据当时的情况，从多个相同类型且参数也相同的函数中选择一个去执行，这时，若采用函数指针调用法就显得简单明了。例如下面的程序，将指针定义为函数指针数组，数组中的每一个指针，指向一个函数，如设有三个数学函数：quad，sqrt 和 log，它们都是 double 型的，且都只有一个 double 型的形参，定义 double(*f[M])()为一函数指针数组，然后用三个函数名作为函数指针数组的三个指针值(设 M=3)，因面可根据需要通过其中的一个指针来调用所指的函数：

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#define M 3;
main()
{
```

```
    double x;
    double delta=1.0;
    double first=0.0;
    double last=10.0;
    double (*f[M])();
    int i;
    char ch;
    double quad(double);
    f[0]=quad;
    f[1]=sqrt;
    f[2]=log;
    for (i=0;i<M;i++)
      {
       x=first;
       while (x<=last)
       {
       printf("f(%lf)=%lf\n",x,f[i](x));
       x+=delta;
       }
       printf("press any Key to continue");
       ch=getchar());
       clrscr();
      }
  }
double quad(double x)
    {
      double a=1.0,b=-3.0,c=5.0;
      return((x*a)*x+b)*x+c;
    }
```

三个函数中的 sqrt 和 log 是库函数，它们在头文件 math.h 中进行了说明，为此用户无需定义它。

4.6.4 函数的参数传送

当调用函数时，函数中的参数传送有两种方法，即通常的参数值传送，将函数中的形式参数用实参代替，这是对形参是属于基本数据类型时的情况，正如前面堆栈操作所述，当调用函数时，将依照参数的排列顺序，将参数反序压入堆栈(先进后出)，被调用函数通过将参数从栈中弹出，而取得实参，以代替函数中的形参。如图 4.8 所示：这种调用对主函数实参向堆栈传送是单向的，而函数中的形参从堆栈取值却是双向的，函数中的，这样即使在

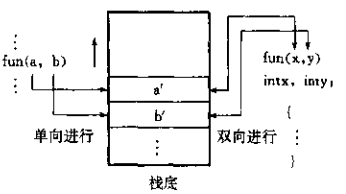


图 4.8 函数调用时的值传送

函数中对传递的实参值进行了改变，它也不会影响到原调用函数中的实际参数值，因为虽然在函数中改变了实参值，它也将改变在堆栈中形参的值，但调用函数的实参地址和形参地址不同，因而不会影响被调用函数中使用的实参值。图 4.8 表明，函数被调用时，实参值单向地压入堆栈，函数执行时，从栈中取出实参值来代替形参，当执行过程中实参值发生变化，则也相应的将变化后的值压入堆栈中的原先位置处，但被调用函数的实参地址不在这儿，因而不会影响到调用函数中的实参值。例如下面的程序：

```
main()
{
  int i=15;
  printf("%d%d",sqr(i),i);
}
sqr(x)
int x;
{
  x=x*x;
  return(x);
}
```

这是通过值调用，虽然在 sqr(x)函数中，当用实参 i 代替形参时，它的值变为 225 了，但它并不影响主函数中的 i 值。

C 程序中另一种参数传送方法是通过地址传送，即将实参的地址传给被调用函数，被调用函数从传过来的地址，取出其所存内容而进行操作，如图 4.9 所示。调用函数时，压入堆栈

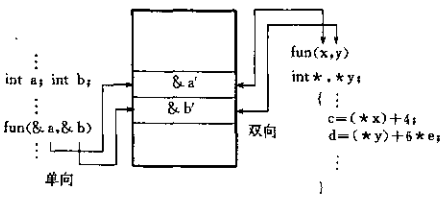


图 4.9　函数调用时的地址传送

的是地址值，即实参为地址值，函数得到的实参也是地址值，只有在函数中需要该实参地址中的内容时，才从相应地址中取出，例如下面的程序：

```
#include <stdio.h>
void sub(int *,int *);
main()
  {
    int i,j;
    printf("i,j=?");
    scanf("%d,%d",&i,&j);
```

```
    printf("Before calling\n");
    printf("i=%d j=%d i*j=%d\n",i,j,i*j);
    sub(&i,&j);
    printf("After calling\n");
    printf("i=%d j=%d i*j=%d\n",i,j,i*j);
    getch();
  }
void sub(int *i,int *j)
  {
  *i=*i+2;          /*取出i地址中的值加2后送回*/
  *j=*i-*j;         /*取出i地址和j地址中的值相减送j地址*/
  printf("In sub after calling\n");
  printf("i=%d j=%d i*j=%d",*i,*j,(*i)*(*j));
  }
```

由于这种通过地址的传送在函数中取操作数时，是从传过来的地址中去取。而写入时，也是写到该地址中去，当其值发生变化时，传过来的地址中内容也发生变化。因而当在调用函数处取该地址中的内容时，将得到变化后的内容。这种特点是和通过值调用不同。也常利用通过地址参数调用，使函数执行结束后，可取得多个返回结果值，不像用 return 语句仅返回一个结果，这样在调用函数中，只要调用函数后，再对传送的多个地址参数进行读取，这时得到的值就是函数中返回的多个值。

若要求通过地址传送，调用函数后，并不改变原来地址中的值，只能在函数中不再对这些地址重写。如上例中，将函数改写为：

```
void sub(int *i,int *j)
{
  int a,b;
  a=*i+2;
  b=*i-*j;
  printf("In sub after calling\n");
  printf("a=%d b=%d a*b=%d,",a,b,a*b));
}
```

这时，i 和 j 地址中的数在函数中没有重写，因而没有发生变化。

下面就常用的几种函数参数传送现象进行一下讨论：

1. 用数组作为向函数传送的参数

这时只能将数组的地址传送给函数，而不是将整个数组。所谓将数组的地址传送给函数，是指将数组的第一个元素的地址传给函数。

一般可采用三种方法来定义函数中的形参以便接收数组地址的传送：

① 将函数中的形参也定义为和实参同样大小，同样类型的数组，因而调用时，只将实参数组名作为参数传送即可，例如：

```
#include <stdio.h>
void disp();
```

```
main()
  {
    int n[10],i;
    for (i=0;i<10;i++)
      n[i]=i;
    disp(n);
    getch();
  }
void disp (int m[10])
  {
  int i;
  for (i=0;i<10;i++)
    printf("%d",m[i]);
  }
```

② 将函数中的形参定义为不定长的数组，它的长度将由调用函数中的实参数组决定，如

```
void disp (int m[])
  {int i;
   for (i=0;i<10;i++)
     printf("%d",m[i]);
  }
```

③ 将函数中的形参定义为一个指针，调用时，将数组名作为实参(即指针)进行调用即可，这是普遍使用的一种方法，如

```
void disp(int * p)
  {int i;
    for (i=0;i<10;i++)
      printf("%d", * (p++))
  }
```

若要传送数组中的某个元素，只要将函数中的形参定义为和数组同类型变量，然后调用时，将数组元素作为实参传送即可，例如：

```
#include <stdio.h>
void display();
main()
{
  int n[10],i;
  for (i=0;i<10;i++)
  {n[i]=i;
   display(n[i]);
  }
getch();
```

```
}
void display (int m)
    {
      printf("%d",m);
    }
```

对于多维数组的传送，实际上和一维数组传送完全类似，不过若定义形参为指针时，要采用规定的形式。如传送的数组为 int m[2][2]，则形参的指针要说明成 int (*p)[2]的形式，或者为 int p[][2]的形式，例如一个二维数组作为实参向函数传送：

```
#include <stdio.h>
main()
  {
  int a[3][3],i,j;
  void b();
  for (i=0;i<3;i++)
    for (j=0;j<3;j++)
      a[i][j]=0;
    b(a);                          /*传送参数为二维数组地址*/
    for (i=0;i<3;i++)
      {
        for (j=0;j<3;j++)
          printf("%d",a[i][j])
        printf("\n");
      }
  }
void b(int(*p)[3])                 /*函数中形参说明成一个二维数组的指针*/
  {int i;
    for (i=0;i<3;i++)
      a[i][j]=1;
  }
```

要注意的是，当对二维数组作为参数传送时，对应它的函数形参中的二维数组指针，必须要指明它的第二维的长度，因为只有说明了它后，当指针增量时，才能正确地知道相应地址要增加多少，否则便是一个未知数，而使编译程序无法正确编译。

对于多维数组当作为参数传送给函数时，除第一维外，其它维也必须加以说明，比如将数组 m 定义为三维：

```
int m[4][3][5];
```

那么接收 m 的函数形参(设为 d)则应说明成如下形式：

```
int d[][3][5];
```

或者用指针形式表示为：

```
int (*p)[3][5];
```

对于多维数组作为参数进行传送时，将使计算机耗费大量时间计算下标，因而使得存取数组元素的时间增加。多维数组操作也不方便，一般在程序中对多维数组采取动态分配内存

的函数进行处理。

数组传送的另一种方法，是将数组定义为全局变量。这样数组既可在主函数中使用，也可在函数中使用，因它属于全局的，所以可以不用作为参数来传送，例如下面程序：

```
#include <stdio.h>
void disp(void);
int n[10];
main()
  { int i;
    for (i=0;i<0;i++)
      h[i]=i;
      disp();
      getch();
  }
void disp(void)
  { int i;
    for (i=0;i<10;i++)
      printf("%d",n[i]);
  }
```

使用全局变量将使得函数和主函数间相互依附，独立性变差，函数作为独立单位被别的函数调用就比较困难，且全局变量占用的内存在整个程序运行期间不会被释放，这对于较大数组将占用的存储单元太多，因而造成资源浪费太大。

2. 将结构作为向函数传送的参数

在调用函数时，可以将结构变量或结构数组作为参数，传送给函数，当传送结构变量的一个元素时，可以像传送一个简单变量一样进行，也可以将整个结构传送给函数，如同数组传送一样，也可将多个结构组成的数组在函数中传送，也可用结构指针进行传送，现分别讨论：

① 将结构中的元素作为参数进行传送

这种传送如同简单变量传送一样，多采用值传送的方法，此时形参和实参类型当然要一致，例如定义一个结构和函数：

```
#include <stdio.h>
struct data
  {
    int a;
    int h;
    int c[10];
  };
main()
  { int i;
    int func(int x);
    struct data arg;
    arg.a=15;
```

```
    arg.b=4;
    for (i=0;i<10;i++)
    { arg.c[i]=i;
    arg.c[i]=func(arg.c[i]);
    }
  printf ("arg.a=%d arg.b=%d",arg.a,arg.b);
  for (i=0;i<10;i++)
    printf("arg.c[%d]=%d",i,arg.c[i]);
  }
int func(int x)
  { if x>5
   x=x*100;
   else
   x=x*200;
  return(x);
  }
```

上面结构变量元素像简单变量一样进行了传送，当然若用元素的地址传送也可以，如可写成func(&age.c[i])，表示将c[1]数组元素的地址传给函数，func(&.age.a)，表示将结构元素age.a的地址传给函数，不过此时形参应定义为指针变量。

② 将结构作为参数向函数传送

将结构作为参数向函数传送，可以采用值传送的方法，也可采用地址传送的方法，如采用值传送的方法，可以将结构元素全部传送给函数，这可以用局部变量的方法，即在被调用函数中，将形参也说明成同样结构类型的结构变量。当然在调用函数中作为参数要传送的结构也被定义成一局部的结构变量，在这种调用方法中，函数将不可能返回对传送结构处理后的结构。若希望函数返回值仍是处理过的结构，则宜采用将结构定义成全局变量，而进行传送。例如下面是将结构定义成局部变量进行传送：

```
main()
  {
  struct{
        int a,b;
        char ch;
        }arg;
        arg.a=100;
          f(arg);
  }
f(parm)
    struct{
        int x,y;
        char ch;
        }parm;
    {
      printf("%d",parm.x);
```

```
}
```

主函数将 arg 结构传给函数 f，f 函数形参 parm 也定义成和 arg 同样的结构，因而显示 parm.x，实际就显示了 arg.a 的值，即 100。

更一般的方法是，定义一个全局的结构，令实参和形参均为这一结构类型，同样也保持了形参和实参类型的一致，这种定义方法显得更方便一些。例如，上面程序可改写为：

```
struct str_type
  {
    int a,b;
    char ch;
  };
main()
  {
   struct str_type arg;
   arg.a=100;
   f(arg);
  }
f(struct str_type parm)
  {
   printf("%d",parm.a);
  }
```

下面的程序将一个结构作为全局变量，进行传送。

```
#include <stdio.h>
#include <stdlib.h>
struct stud_type
  {
    char name[20];
    long num;
    int age;
    char sex;
    float score;
  }student[30];
  int n=0;
main()
  {
    char ch;
    int flag=1;
    while (flag)
      {
        printf("\n type 'E' or 'e' to enter new record");
        printf("type 'L' or 'l' to list all records : ")
        ch=getchar();getchar();
```

```
        swit ch(ch)
        {cose 'e' :
        cose 'E' : new_record();
                   ;break;
          cose 'l' :
          cose 'L' : listall();
                     ;break;
          defanlt : flag=0;
        }
    }
}
new_record(void)
{
    char numstr[20];
    printf("\n record %d : \n enter name : ",n+1);
    gets (student[n]. name);
    printf("\n enter number : ");
    gets (numstr);
    student[n]. num=atoi(numstr);
    printf("\n enter age");
    gets (numstr);
    student[n]. age=atoi(numstr);
    printf("\n enter sex : ");
    student[n]. sex=getchar();getchar();
    printf("\n enter score : ");
    gets(numstr);
    student[n]. score=atoi(numstr);
    n++;
}
listall(void)
{
int i;
    if(n<1)
      printf("\n empty list. \n");
    for (i=0;i<n;i++)
      {
       printf("\n record number %d\n",i+1);
       printf("name : %s\n",student[i]. name);
       printf("num : %ld\n",student[i]. num);
       printf("age : %d\n",student[i]. age);
       printf("sex : %c\n",student[i]. sex);
       printf("score : %6. 2f\n",student[i]. score);
      }
}
```

该程序定义了一个 student[30]数组，该数组是一个结构为 struct stud_type 型的，是全局变量，因而各函数均可共同使用这个数组，它存放在内存的数据，不像局部变量那样要通过堆栈来存取，因此在使用该数组的函数中，没有形参，因所有函数均可对该数组进行操作，所以无需传送。

该程序运行时，将显示：

type 'E'or 'e' to enter new record,type 'L'or 'e' to list all record;

这样打 E 或 e 时，可通过键盘一一输入每个学生的记录，即结构的各元素。当打 L 或 e 时，则列出输入的所有学生的记录，即结构的各个元素项。

③ 将结构指针作为参数进行传送

正如前面所述，使用全局变量对大程序来说会带来许多问题，所以较方便的方法是采用结构指针，即用结构变量的地址作为参数来进行传送。使用时函数的形参定义成和要传送的实参具有相同结构的一个指针变量，调用时将结构地址传给它就可以了。例如定义了一个结构，设为：

```
struct date
  {
    inat a;
   int b;
   cha ch;
   }
main()
  {
    struct date arg;
    void subf(struct data *p);
         ⋮
    subf(&arg);
         ⋮
  }
void subf(struct data *p)
  {
    ⋮
  }
```

main()主函数中的结构变量以地址形式作为实参传给函数，而函数中的形参定义为指向和实参同样结构的一个指针。

下面用一个具体的例子来说明用结构指针传送的例子。假设定义了一个结构，在 main()主函数中建立了一个这样结构的元素组成的数组，即结构数组，可以通过指针，由函数接收一个传送过来的数组元素。即一个结构可以被传送过来。

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#define MAX 10
```

```
struct addr
  {
    char name[20];
    long num;
    int age;
    char sex;
    float score;
  };
main()
{
  struct addr student[MAX];
  int i;
  int input(struct addr *p);
  void display(struct addr *p,int);
  for(i=0;i<MAX;i++)
    if(input(&student[i])==0)break;
  printf("\n");
  display(student,i);
}
int input(struct addr *p)
  {
    cbar numstr[20];
    printf("\n enter name : ");
    gets(p->name);
    if(p->name[0]=='0')return(0);
    printf("\n enter number : ");
    gets(numstr);
    p->num=atoi(numstr);              /*将字符串转换成整数*/
    printf("\n enter age : ");
    gets(numstr);
    p->age=atoi(numstr);
    printf("\n enter sex : ");
    p->sex=getchar();getchar();       /*接收一个字符,并接收一个回车符*/
    printf("\n enter score : ");
    gets(numstr);
    p->score=atof(numstr);            /*将字符串转换成实数*/
    return(1);
  }
void display(struct addr *pd,int n)
{
  int i;
  printf(" name \t\t      num    age   sex   score \n");
  printf("-------------------------------------------------------\n");
  for(i=0;i<n;i++,pd++)
```

```
    printf("%-15s%81d %6d %3c %8.2f",pd->name,pd->num,pd->age,pd
        ->sex,pd->score);
}
```

上面程序由三个函数组成，其中 input 函数用于对结构的各元素输入数据，它的形参 p 定义成结构指针，用来接收传过来的结构地址，在 main()主函数中通过 for 循环不断调用该函数，而实参为一确定的结构地址，即 &student[i]，此时 i 已有确定值，这样结构地址送到 input 函数中，在该函数中给各结构元素赋值，若输入过程中，对 name 元素输入 0 时，则整个输入过程结束。

输入过程均用 gets 接收键盘输入，对于整数型的元素，作为字符串输入时，则接收的字符串用 atoi()函数转化成整数型的，其中使用字符数组变量 numstr[20]作为输入串的暂存处。由于定义 sex 为字符型的，所以用 getchar()函数，其中第一个用来接收性别，第二个 getchar()函数则用来接收回车符，若不将回车符接收掉，它将导致 gets()函数接收。对于 score，由于已定义为 float 型，故用 atof()函数，将输入的字符串转换为实数，该函数使用时，一定要将 stdlib.h 和 math.h 头文件包含在程序中，否则该函数将不能正确执行。

输入数据后，调用 display()函数将输入的所有结构数组元素显示出来，此时将整个结构数组(实际上是它的第一个元素地址)传给了此函数，而在函数中，通过形参指针 pd 得到指向该结构数组的指针后，通过 pd++，则可将所有的结构数组元素的值取出而输出出来。

为了得到正确的输出格式，对输出项的定位很重要，在 printf()中，'\t'表示回到下一个输出区位，而第二个 printf()语句中的"%-15s"表示输出的 name 左对齐。其它输出项的输出长度是经过实验确定的。用上面的格式恰能将各输出项的值对号入座在对应输出项下。

3. 将函数作为一个参数传送给另一个函数。

在这种情况下，被调函数的形参中必须要有一个被说明成函数指针。这样在调用时，只要将形参函数指针换成相应的函数名或它的指针即可。例如

```
#include<stdio.h>
main()
  {
    double x;
    double delta=0.01;
    double first=0.0;
    double last=10.0;
    double (*fx)();
    double quad(double);
    double largest(double,double,double,double(*fx)());
    fx=quad;
    printf("The Largest Volue in the range %Lf->%Lf",first,last);
    printf("is%Lf\n",lagest(first,last,delta,fx));
  }
double quad(double x)
  {
    double a=1.0,b=-3.0,c=5.0;
```

```
    return ((x*a)*x+b)*x+c;
  }
double largest(double a,double b,double step,double(*fx)())
  {
    double x=a,big=(*fx)(a);
    while (x<=b){
      if(big<(*fx)(x))
          big=(*fx)(x);
      x+=step;
    }
    return big;
  }
```

在该程序中，主函数 main 通过调用 largest 函数来求最大值，而通过函数指针 fx，把函数 quad 作为参数传给 largest 函数。quad 函数用来计算 $a*x^3+b*x+c$ 的值。在调用 largest 函数时，仅是把 quad 函数地址传来了，当在被调用函数中作为实参使用时，它的参数值，则由使用函数对其再进行传送。如在 largest 函数中，就把形参 a 的值作为它第一次调用时的参数，以后再调用时，又使用 a 按 step 步长增量后的值。largest 函数根据设定的计算法则，求出某一范围内的最大值。

总结：当函数的参数为基本类型时，一般采用值传送方式，对于像数组、结构这些复合类型的参数，一般多采用指针即地址传送方式。

4.6.5 main()函数中的参数

main()函数可以带有两个参数，即可以写成 main(int argc,char *argv[])，其中第一个参数 argc 是一个整型变量，第二个参数 argv 是一个字符型指针数组，其每一个指针元素指向一个字符型数据。这两个参数的具体值是由 DOS 下的命令行参数值传送来的，由于 main()函数是一个主函数，它不能被别的函数调用，故参数的传递仅可从命令行来。

当用 Turbo C 编程(在 main()函数中带有参数)并将源程序编译连接成可执行文件存盘后，便生成后缀为 .EXE 的可执行文件了。在 DOS 下执行这个文件，就是执行一条 DOS 的外部命令，它是可以带有参数的，而这些参数值便可相应的传给 main()函数中的参数，从而带进程序中去。其中第一个参数 argc 表示了外部命令中带的参数个数，它至少应为 1(这个参数用户无需输入，程序自动生成)，因程序名规定是第一个参数。第二个参数 argv[]是一个不定长的指针数组，其中 argv[0]指向程序名，argv[1]指向命令行中的第一个参数，argv[2]指向第二个参数，…依此类推，若仅带两个参数，则仅有 argv[1]和 argv[2]。例如生成的可执行程序名为 myfile.exe，当在 DOS 下执行它时，若写成：

```
c> myfile ab↙
```

则 myfile 程序中的 main()函数中的两参数 argc 将等于 2，argv[0]指向程序名，argv[1]指向参数 ab，如图 4.10 所示，即 argv[0]为存放程序名的地址，argv[1]是存放第一个参数的地址。

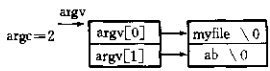


图 4.10 main 函数中的参数

例如下面的程序是一个倒计数程序，当用户输入该程序名并后面带计数值参数时，当减

计数到 0 时，便发出嘟嘟声，若在命令行中带有第二个参数 display 时，则将不断显示减计数值直到发嘟嘟声为止，若该程序名为 count.exe，则在 DOS 下，输入：

```
count 100 display↙
```

便会出现上述的现象，要注意的是：输入的参数间只能用空格或制表符分隔，不能用逗号、分号及其它符号分隔，下面就是该程序的源程序清单：

```
main (int argc,char *argv[])
  {
    int disp,n;
    if (argc<2)
      {
        printf("Your must enter the length of the count\n");
        printf("On the command line : Try again. \n");
        exit(0);
      }
    if (argc==3 && ! strcmp(argv[2],"display"))
      disp=1;
      else
      disp=0;
    for (n=atoi(argv[1]);n>0;--n)
      if(disp)
        printf("%d\n",n);
    sound(700);
    delay(1000);
    nosound();
  }
```

从理论上讲，从命令行中可以传递 32767 个参数给 argv[]，但实际上 DOS 仅允许带几个参数，而一般实际使用时，也仅带几个就够了。

利用 main()函数可带参数这一特点，可以在 DOS 命令行中随机地改变参数以适应当时条件的需要。这在编制应用文件时特别需要，它使得运行程序变得更灵活了。

4.7 函数的递归调用

Turbo C 允许函数自己再调用自己，这就是所谓的递归，采用递归可使程序紧凑，但由于函数调用要使用堆栈，因面运行速度变慢且占用较多内存。

下面是一个快速排序程序，其中在 sort()函数中就采用它自己的递归调用。

```
#include<stdio.h>
main()
{
  static int i,x[]={4,7,9,5,2,5,9,2,1,9,-5,-3};
  int find_pivot(),partition();
```

```
    void sort();
    for(i=0;i<12;i++)
      printf("%4d",x[i]);
    printf("\n");
    sort(x,12);
    for(i=0;i<12;i++)
      printf("%4d",x[i]);
    printf("\n");
}
void sort(a,n)
  int a[],n;
  {
    int k,pivot;
    if(find_pivot(a,n,&pivot)!=0)
    {
      k=partition(a,n,pivot);
      sort(a,k);                          /*递归调用*/
      sort(a+k,n-k);
    }
  }
  int find_pivot(a,n,pivot_ptr)           /*找大者*/
    int a[],n,*pivot_ptr;
    {
      int i;
      for(i=1;i<n;++i)
        if(a[0]!=a[i])
        {
          *pivot_ptr=(a[0]>a[i])? a[0]:a[i];
          return(1);
        }
      return(0);
    }
  int partition(a,n,pivot)
    int a[],n,pivot;
    {
      int i=0,j=n-1;
      void swap();
      while(i<=j)
      {
        while (a[i]<pivot) ++i;
        while (a[j]>=pivot) --j;
        if(i<j)swap(&a[i++],&a[j--]);
      }
    return(i);
```

```
}
void swap(p,q)                          /*交换元素*/
  int *p,*q;
  {
    int temp;
    temp=*p;
    *p=*q;
    *q=temp;
  }
```

4.8 函数的组织

对结构化程序设计，将程序要实现的功能逐步分解，最后化成一个个函数来实现，如同一个大楼用一块块的构件砌起来一样，对于不大的程序，主函数和函数可放在一个程序中。对很大的程序，可能有众多的函数，可以将那些通用的函数组织成一个函数库，这样当需要某个函数时，可将其包含进来。另一种方法是将这些函数组成一个文件，进行编译，然后和要用这些函数的程序进行连接，这样将使得程序变大，因为这时文件中的全部函数均连接到程序中来了，也许文件中的许多程序并未用到。

第 5 章

内存模式与动态存储结构

Turbo C 编译程序对源程序进行编译时，将函数中命令、语句编译成相应序列的机器指令代码放在代码段中，将已初始化的数据（如已赋值的全局变量、静态局部变量等）放在数据段内，将未初始化的数据放在 BSS 段内，对临时性数据，如函数调用时传递的参数、局部变量、返回调用时的地址等则放在栈段内，而对一些动态变化的数据，如在程序执行中建立的一些数据结构，如链表，动态数组等，则放在堆结构中（Heap）。关于堆（Heap）和栈（Stack）的使用，下面还要介绍（注意上述所指变量包括数组和结构）。

按照编译时选用的内存模式不同，这些段的命名和段的组合情况，及堆、栈的分配情况均有所不同，下面将分别进行介绍。

Turbo C 提供了六种编译的内存模式，也有人称为寻址模式，它们是微小模式（Tiny）、小模式（Small）、中模式（Medium）、紧凑模式（Compact）、大模式（Large）、巨模式（Huge）。编译前，在 TC 集成开发环境下，可在 Compiler 项下的 Model 子菜单项下进行选择。

5.1 内存模式（编译模式）

内存模式是指如何在内存中放置程序代码及数据、如何分配堆栈，它们允许占用的内存大小，及如何存取它们，当指定好内存模式后（又称编译模式），语言编译程序将按事先选择好的内存模式编译组织程序，Turbo C 提供了六种编译模式（即内存模式），用户可按自己的程序大小及需要进行选择，这六种模式是：

5.1.1 微小模式（Tiny）

程序中的数据及代码均放在同一段内，即它们不超过 64K，在该模式下代码段、数据段、堆栈段的段地址均相同，即 CS＝DS＝SS＝ES，在该模式下，指针都是近程（near）型的。一般小程序可采用此编译模式进行编译。还可用 DOS 中的 EXE2BIN 转换程序将 exe 程序转换成 .COM 程序，在该模式下，内存的分配格式如图 5.1 所示。从图中可以看出，代码段、数据段、堆栈段均在同一段内，对它们进行寻址时，均以同一段地址为地址偏移的参考点，具有这种特点的段我们又称为属于同一段组（DGROUP），从图中可以看出，栈是向上生长的，即每压栈一次，栈指针 SP 减 2，即向减少地址的方向移动，它开始的初始值指向栈底，即 0xffff（64K）。堆是向下生成的，即向增加地址的方向改变，可以看出，图中，堆和栈地址相向生长，当两者未相遇时，便出现了自由空间。一般程序均是这种状态，当占用栈地址较多时，两者可能重合并覆盖部分堆空间。

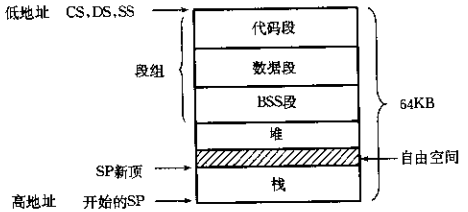


图 5.1 微小模式内存分配

5.1.2 小模式(Small)

在该模式下,程序中的代码放在 64K 的代码段内,数据放在 64K 的数据段内。栈段,附加数据段和数据段均指向同一地址,它们合三为一,即 DS=SS=ES。在该模式下,指针都是近程的(near),一般程序均采用此模式编译,在该模式下,内存分配图如图 5.2 所示。可以看出数据段、堆栈段、附加段为同一段组,即它们的偏移地址均以同一段地址为参考点。

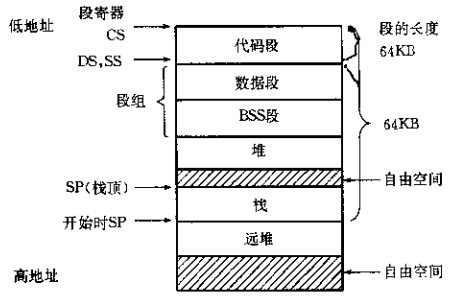


图 5.2 小模式内存分配

5.1.3 中模式(Medium)

在该模式下,所有数据放在 64K 的数据段内,因而数据段内使用近程指针(near)。代码量可大于 64K(允许达到 1M),因而可以在不同的代码段内,代码段使用远(far)指针。该种编译模式适用于大代码量,小数据量的大程序。该模式下的内存分配如图 5.3 所示。

5.1.4 紧凑模式(Compact)

在该模式下,数据量可超过 64K,因而可放在多个数据段,数据段内的指针是远程的(far)。代码量不超过 64K,在一个段内,因而代码段内指针是近程的(near)。但在该模式下,静态数据仍不能超过 64K,堆用 far 指针来存取,该模式下内存结构如图 5.4 所示。

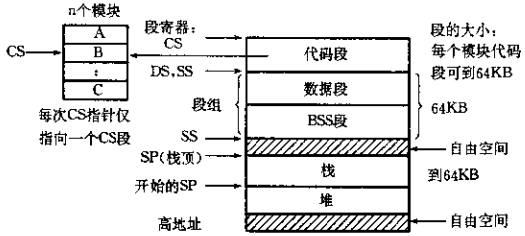


图 5.3　紧凑模式内存分配

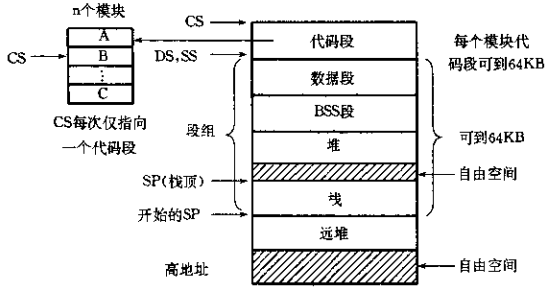


图 5.4　中模式内存分配

5.1.5　大模式(Large)

在该模式下,代码和数据均可采用 far 指针,两者均可达 1MB,但静态数据仍如同紧凑模式一样,不能超过 64K 字节。该模式下的内存结构如图 5.5 所示。

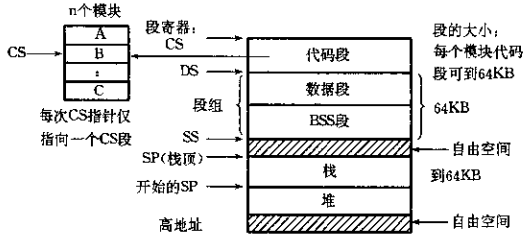


图 5.5　大模式内存分配

5.1.6 巨模式(Huge)

在该模式下,代码段和数据段内均用 far 指针,代码分布在不同的代码段内,数据也分布在不同的数据段内,它们来自于不同的源程序,但堆栈只有一个。Turbo C 一般限制静态数据不超过 64K,但巨模式允许超过 64K。该模式下内存结构如图 5.6 所示。

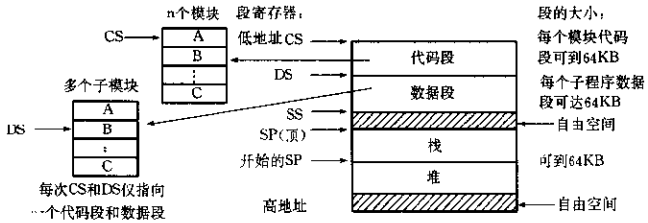


图 5.6 巨模式内存分配

紧凑模式、大模式、巨模式均允许数据区超过 64K,即可以用数据 far 指针对不同数据段内的数据进行存取,它们统称为大数据存储模式。但有一点不同的是:紧凑模式、大模式按 Turbo C 的规定,其静态数据,即如数组,结构或其它类型的数据被定义为静态类型时,其数据量不能超过 64K 字节,而只有巨模式才允许超过 64K。在大数据存储模式下,堆和栈分别在不同段内,所以动态数据和局部变量等不受 64K 的限制,栈的增长不会影响堆的空间。

无论采用哪一种编译模式,单个的 Turbo C 源程序其编译生成的代码和数据量均不能超过 64K 字节,对于超过的源程序,可以视代码或数据多少将其分解成两个或多个程序分别编译。但编译模式对大代码量程序要选用大代码编译模式(中模式、大模式、巨模式),对大数据量程序,选用大数据编译模式(紧凑模式、大模式、巨模式),这样编译生成的.OBJ 文件,将会带给连接程序信息将代码和数据安排在不同段内。这样生成的.EXE 文件在加载时将告诉 DOS 该程序应如何装入代码段和数据段、如何初始化各段寄存器。这样,就可实现在不同编译模式下,开辟数据区和代码区的大小,即大于 64K,或不超过 64K。

5.2 各内存模式下缺省段名和堆的分配

按照编译时选择的内存模式不同,Turbo C 对各个段的命名和段的组合情况,及堆栈的分配情况有所不同。当和汇编语言混合编程时,如用 C 编的模块和用汇编语言编的模块进行连接时,就需知道这些段名和类型,因为只有使汇编语言编的模块各段名同 C 编译的模块相应段同名时,才能正确地连接在一起。当然各段名可以由用户统一命名,但更方便的方法是采用缺省名。

当我们对 TC 编的源程序进行编译时,可使用段名控制选择项以便由用户命名这些段。(如在 TC 集成编译环境下,选 Options 菜单项的 Names 子菜单项时,便可由用户命名代码段、数据段、BSS 段及 Group 和类型(class)的名字。在 TCC 命令行编译器的 -Z 选择项下,

各段改名的选择项也有同样功能),但一般均不使用这些改名的子菜单或选择项,而是采用其缺省名,即使用 Turbo C 内定的各段名、段的类型及段组名。如在微小模式下,生成的代码段名为_TEXT,段的类型为 CODE 段,生成的数据段名为_DATA,段的类型为 DATA 段,生成的未初始化数据段名为_BSS,段的类型为 BSS,生成的堆栈段名为 STACK,段的类型为 STACK,这四个段组成一个段组,段组名为 DGROUP,即它们的段地址相同,其偏移地址均相对于同一段地址而言,对段寄存器有 CS=DS=SS,因而这些段合起来不能超过 64K。在各内存模式下缺省的段名及类型如表 5.1 所示:

表 5.1 各内存模式下的缺省段名及类型

编译模式	缺省段名	段的类型	段组名	段寄存器
微小模式	_TEXT	CODE	DGROUP	CS=DS=SS
	_DATA	DATA	DGROUP	CS=DS=SS
	_BSS	BSS	DGROUP	CS=DS=SS
	STACK	STACK	DGROOP	CS=DS=SS
	(堆和栈同属于 STACK 段)			
小模式	_TEXT	CODE		CS
	_DATA	DATA	DGROUP	DS=SS
	_BSS	BSS	DGROUP	DS=SS
	STACK	STACK	DGROUP	DS=SS
	(近堆和栈同属于 STACK 段,还有一个远堆)			
紧凑模式	_TEXT	CODE		CS
	_DATA	DATA	DGROUP	DS
	_BSS	BSS	DGROUP	DS
	STACK	STACK		SS
	(有一远堆)			
中模式	模块名 1_TEXT	CODE		模块 1 的 CS
	⋮			⋮
	模块名 n_TEXT	CODE		模块 n 的 CS
	_DATA	DATA	DGROUP	DS=SS
	_BSS	BSS	DGROUP	DS=SS
	STACK	STACK	DGROUP	DS=SS
	(近堆和 STACK 在同一段内)			
	(有一远堆)			
大模式	模块名 1_TEXT	CODE		模块 1 的 CS
	⋮			⋮
	模块名 n_TEXT	CODE		模块 n 的 CS
	_DATA	DATA	DGROUP	DS
	_BSS	BSS	DGROUP	DS
	STACK	STACK		SS
	(有一远堆)			
巨模式	模块名 1_TEXT	CODE		模块 1 的 CS
	⋮			⋮
	模块名 n_TEXT	CODE		模块 n 的 CS
	模块名 1_DATA	DATA		模块 1 的 DS
	⋮			⋮
	模块名 n_DATA	DATA		模块 n 的 DS
	STACK	STACK		SS
	(有一远堆)			

从各内存模式的内存分配图中可以看出，小数据模式(微小、小、中模式)的堆均夹在_BSS和 STACK 之间，它实质上和数据段、STACK 段、BSS 段同在一段组内。小模式，中模式除了近堆，还有远堆，它们可以通过设置远指针来进行存取。

对大程序模式(中模式、大模式、巨模式)，程序由许多模块连接而成，对每一模块都有自己的代码段，因而缺省的段名要在_TEXT 前再加上模块名，即模块名_TEXT。除巨模式外，所有模块，所有已初始化的全局变量均在数据段，未初始化的全局变量均在 BSS 段，生成的堆栈均为 STACK 段，且中模式的 DATA、BSS、STACK 均在同一段组(DGROUP)中。大模式的 DATA 和 BSS 为同一段组。对于巨模式，每个模块均有自己独立的代码段、数据段，没有 BSS 段。

程序进行编译时，将按照选择的内存模式，分别按图 5.1～图 5.6 的形式将其放入内存。

5.3 栈的结构

栈是一种先进后出的数据存储结构，它总是驻留在堆栈段内，SS 寄存器用来作为堆栈段(STACK)的段寄存器，堆栈指针寄存器 SP 作为堆栈段栈顶偏移地址寄存器，它用来给出堆栈内的偏移地址。例如在汇编语言中，将 AX 寄存器的值(如 2107H)压入堆栈(注意！堆栈操作用压入指令 PUSH 和弹出指令 POP，它们一次均对一个字即 16 位进行操作，即栈操作没有 8 位的，当然对 80386 还有 32 位的栈操作指令)，和将堆栈中的数弹出到 AX 寄存器，其堆栈段内数据的存放及 SP 的变化情况如图 5.7 和图 5.8 所示。

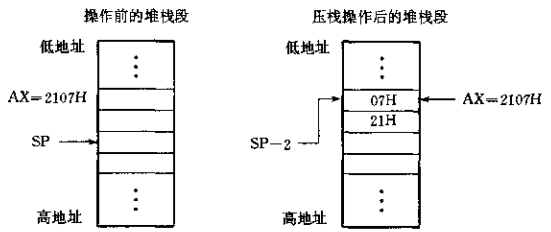


图 5.7 push AX

图 5.7 表示在未进行堆栈操作前，SP 指向栈中的某一地址，当压栈操作后，将 AX 中的 16 位数压入堆栈，此时 SP 的值减 2，即指向下一个低地址，以准备接收下一个要压栈的数据。当进行栈弹出操作时，则将 SP 的值加 2，表示前一地址内的数字已弹出到 AX，因而 SP 恢复为原来的值。

由于堆栈先进后出的特点，堆栈主要用在临时数据的存取，如函数调用时参数的传递，局部变量，调用函数时的返回地址等。例如设有一函数 addup 是对输入的 3 个参数进行处理的函数，其定义如下：

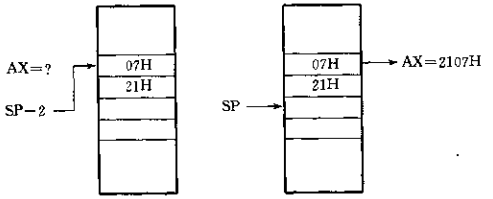


图 5.8 pop AX

```
int addup(int x,int y,int z)
  { int d;
    char f;
     ⋮
```

当调用该函数时，如：

```
main( )
{
  int a=3;
  int b=4;
  int c=6;
   ⋮
  addup(a,b,c)
   ⋮
```

调用函数时首先按三个形参的相反顺序将其对应的实参压入堆栈，如图 5.9 所示，然后根据调用函数是 near 型还是 far 型，将要返回的地址再压入栈(若是 near 型的，即调用函数和被调用函数的代码段均在同一段内，故仅需压入 2 个字节的偏移地址。若是 far 型的，即调用函数和被调用函数处于不同的两个代码段内，则返回地址应包括段地址和偏移地址，即应是 4 个字节：两个字节段地址和两个字节偏移地址)，当程序的返回地址压入栈后，接着把 BP 寄存器的值再压入堆栈，因为对栈中地址寻址时，80x86 系列规定，用 SS 作为堆栈段寄存器，而用 BP 寄存器作栈内偏移地址寻址寄存器。当将 SP 的值送 BP 后，SP 又可用作被调用函数使用的指针又去调另一函数。被调用函数则可通过[BP+4]、[BP+6]、[BP+8]将传过来的三个参数 a,b,c 的值取出使用。当被调用函数结束后，BP 恢复成原来的值，SP 指向返回地址，当执行返回指令后，参数将被清除，即执行 ADD SP,6，SP 恢复成原来调用前的值。

对于被调用函数中的局部变量，则是在将 BP 原来的值压入堆栈后，再按出现的相反次序为这些局部变量分配堆栈空间，如图 5.10 所示。

当被调用的函数返回时，若带有返回值，编译程序利用 AX 寄存器传递返回值(不论是 int 型或 char 型都当作 int 型返回)，若返回值为 4 个字节长度，则用 DX：AX 返回，超过 4 个字节的，如 struct 型的，则放在静态变量中，返回这个变量的指针。

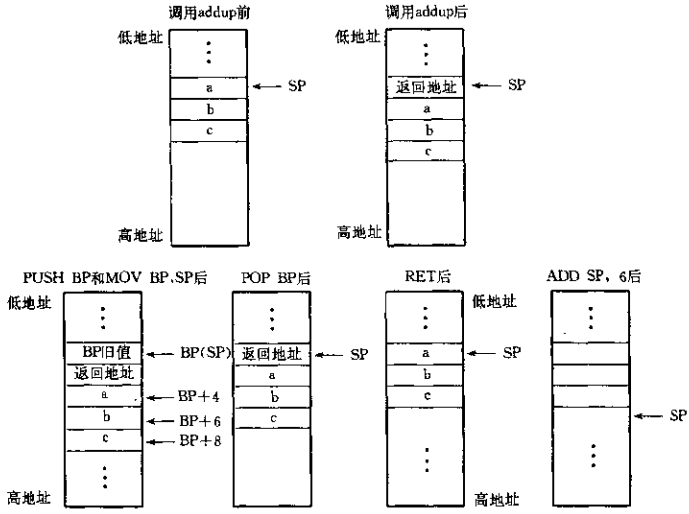


图 5.9　函数参数在栈中的变化

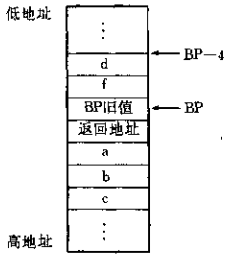


图 5.10　栈中的局部变量

5.4　堆的结构

堆是一种动态存储结构，实际上就是数据段中的自由存储区，它是 C 语言中使用的一种名称，常常用于动态数据的存储分配。堆中存入一数据，总是以 2 字节的整数倍进行分配，地址向增加方向变动。堆可以不断进行分配直到没有堆空间为止，也可以随时进行释放、再分配，不存在次序问题。

所谓动态数据是指在程序运行期间确定其大小的，如常用到的动态数组，它们是在程序执行过程中动态进行变化的，即在程序开始部分没有说明其名称和大小，只有在程序运行期间用堆的分配函数为其分配存储空间，分配的大小可根据需要而设定，这些数据使用过后，可释放它们占用的堆空间，并可进行再分配。动态数据最典型的有动态数组，即它的大小在程序执行过程中被动态地确定。与其相对应的静态数组，是指这些数组(如变量数组、指针数组、结构数组等)它们在程序的开始说明了名字和大小，甚至数值。它们由系统固定地分配了空间，且占据一连续的内存空间，该空间大小不能改变，它们的位置不在堆空间中，而在数据段内，这部分空间不能再使用。

堆的分配情况如图 5.1～图 5.6 所示，对小数据存储模式，堆在未初始化的数据段 BSS 与堆栈段之间(称为近堆，即 near 型的)。堆和栈在使用时相向生长，栈向上生长，即向小地址方向生长，而堆向下增长，即向大地址方向，其间剩余部分是自由空间。使用过程中要防止增长过度，而导致覆盖。对于大数据模式，堆栈以下直至物理存储器末端均可作为堆空间，这种堆可看作是一种远堆，即 far 型。

对堆进行操作，要使用几个堆管理函数，介绍如下：

5.5 堆管理函数

1. 得到堆和栈之间自由空间(即未用的空间)大小的函数

它的说明原型是：

对于小数据内存模式(微小型，小型，中型)为：

```
unsigned coreleft(void);
```

对于大数据内存模式为：

```
unsigned Long coreleft (void);
```

它们将返回自由空间的字节数，对于小数据内存模式，coreleft 返回的是自由空间再减去 256 个字节，这 256 字节为 DOS 所用。当对远堆时，可用 farcoreleft()函数返回自由空间数。

2. 分配一个堆空间的函数

它的说明原型是：

```
void malloc (unsigned size);
```

该函数将分配一块大小为 size 字节的堆空间，并返回一个指向这个空间的指针。由于这个指针是 void 类型的，因此当将它赋给其它类型的指针时，必须对该指针进行强制类型转换。如已说明 info 是一个结构类型的指针，即：

```
struct addr *info;
```

则将由 malloc()函数返回的指针赋给 info 时，必须进行类型转换：

```
info=(struct addr *) malloc (sizeof (record));
```

malloc()函数所分配的堆空间将不进行初始化。在调用 malloc()函数时，若当时没有可用的内存空间，该函数便返回 NULL 指针。若使用大数据内存模式，要建立远堆，则可用 farmalloc 函数。

3. 分配一个堆空间，其大小为能容纳几个元素，每个元素长度为 size 的函数：

它的说明原型为：

void calloc (unsigned n, unsigned size);

该函数将分配一个容量为 n * size 大小的堆空间，并用 0 初始化分配的空间，即每个地址装 0。该函数将返回一个指向分配空间的指针。没有空间可用时，则返回 NULL 指针。若在大数据模式下建立远堆，则可用 farcalloc 函数。

4. 重新分配堆空间函数

它的说明原型为：

void *realloc(void *ptr,unsigned newsize);

该函数将对由 ptr 指向的堆空间重新分配，大小变为 newsize。

5. 释放堆空间的函数

void free (void *ptr);

该函数将释放由 ptr 指向的堆空间，这样释放的堆空间将可重新被利用。

这些函数的原型均在 stdlib.h 和 alloc.h 中。

5.5.1 示例 1——使用 coreleft()和 malloc()函数例

下面的程序使用了 coreleft()函数和 malloc()函数。首先，程序运行时，用 coreleft()函数显示出可用的内存大小，在小数据内存模式下和大数据内存模式下进行编译时，显示的内存空间显然不一样。然后用 malloc(512)分配一个 512 字节的堆空间，其堆指针赋给了 mem_ptr，接着打印该指针和剩余的自由空间。若 mem_ptr 指针不为 NULL 时，则再为 struct cust_tp 结构分配一个堆空间，并打印出 cust_ptr 指针的值和所剩空间。程序最后释放了两个指针指向的堆空间，并显示了此时的剩余空间。该程序是一个演示程序，我们分析一下显示的结果数据。

可以看出，在小内存模式下，可使用内存空间为 63772。当分配一个 512 字节的堆空间后，剩余空间变为 63252，实际用去了 520 字节空间，与设置的 512 字节相比，多出了 8 个字节，这是因为 Turbo C 对每个变量分配的堆空间，在其前部放有一个头，头中放有两个信息，标识其长度和下一个要分配的堆指针，共占八个字节。对于下一个分配的 cust_tp 结构共占用 104 个字节，加上一个头，共占用 112 个字节，因而显示剩余空间为 63140。当将这两个分配的堆空间释放后，剩余空间又变为 63772。有时用函数 coreleft 反映的剩余空间并不准确，它只是给出了堆中的最后一个变量后可用的空间，在变量上面可能还有许多空隙没有被使用。

值得注意的是 malloc()函数将返回一个 void 类型的指针，但它实际是代表堆空间的以字节为单位的地址指针，相当于字符性指针。为了使程序可读性强，移植性好，可将其说明成字符性指针，应该在程序开头进行说明，即用：

char *malloc();

这样编译程序将知道该函数返回的是一个字符指针。另一种方法是对其返回值进行强制性转换。如下例中，若开头不将 malloc()说明成字符型的，则可在程序中用 mem_ptr=(char *)malloc(512)进行强制性转换后，将该指针值赋给 mem_ptr。

当要用 malloc()返回的指针指向一个结构类型的数据时，则要将其强制转换为该结构类型，如程序中用了下面的形式：

cust_ptr=(struct cust_tp *) malloc(sizeof(struct cust_tp));

若在程序中要多处用到 malloc()函数，而它们返回的指针又要指向不同类型的数据，可不在程序开头对 malloc()函数进行类型说明，而在用到它的地方进行强制转换即可。

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
  char *mem_ptr, *malloc();
  struct cust_tp
  {
     char name[30],addr[30],city[20],state[20];
     float balance;
  } *cust_ptr;
  printf("coreleft=%u\n",coreleft());
  mem_ptr=malloc(512);
  printf("mem_ptr=malloc(512)=%p; ",mem_ptr);
  printf("coreleft=%u\n",coreleft());
  if(mem_ptr==NULL)
    printf("Not enough memory for the array");
  cust_ptr=(struct cust_tp *)malloc(sizeof(struct cust_tp));
  printf("cust_ptr=malloc()=%p;",cust_ptr);
  printf("coreleft=%u\n",coreleft());
  if(cust_ptr! =NULL)
    {
      strcpy(cust_ptr->name,"my_name");
      cust_ptr->balance=0.0;
    }
  else
    printf("Not enough memory for the structure");
  free(mem_ptr);
  free(cust_ptr);
  printf("cust_ptr=malloc()=%p;",cust_ptr);
  printf("coreleft=%u\n",coreleft());
}
```

运行结果：

```
coreleft=63772
mem_ptr=malloc(512)=0598;coreleft=63252
cust_ptr=malloc()=07A0;coreleft=63140
cust_ptr=malloc()=07A0;coreleft=63772
```

5.5.2 示例 2—— 使用 farcalloc()函数例

该程序演示了用 farcalloc()函数开辟了一个远堆空间，容量为 3000×8=24KB(即 n *

8bety),程序中用(double huge *)farcalloc()强制分配的远堆指针为 huge 型的。当其指针为 NULL 时,表明无内存空间,否则给其分配在堆空间的动态数组 a[i]赋值,并求和,最后再显示出来。

```
#include <stdio.h>
#include <stdlib.h>
main()
 {
  double huge * a,sum=0;
  int i,n=3000;
  if(a=(double huge * )farcalloc(n,sizeof(double))==NULL)
    {
     printf("coreleft=%u\n",farcoreleft());
     printf("Not enough memory ");
     exit(1);
    }
  else
    for (i=0;i<n;i++)
      {
       a[i]=i;
       sum=sum+a[i];
      }
    for (i=0;i<n;i++)
      printf("a[%d]=%d\n",i,a[i]);
    printf("sum of all a[i]=%8.0f\n",sum);
 }
```

5.5.3 示例 3——堆和栈操作演示例

下面的程序演示了堆和栈的数据存放和弹出,在主程序中用 malloc(50 * sizeof(int))分配了一个 100 个字节的堆空间,它返回的指针用(int *)转化成一个整型数的指针赋给指针变量 pi,这样 pi 就是一个这个堆的指针,即代表这个堆空间的第一个地址(最低的地址)。若 pi 为 NULL 时,则表示没有可用的堆空间了,则显示 allocation failure 字样。当正确地返回一个堆指针后,我们又让其代表一个栈顶的指针,演示一下栈的压入和弹出操作,即 tos=pi,意味着堆和栈重叠。当我们输入一个整数时,则存入堆中(即 pi++)又可看作压入栈中。当输入超过 50 个数时,则显示 stack overflow。当输入 0 时,便将存于栈中的数弹出,即(pi--)。当 pi=tos 时,表示已到栈顶,再 pop,便出现 stack underflow 字样,表示已无数据可弹了。这个过程也演示了栈的特点,即先进后出。这个程序是我们自己定义了一个堆和栈,并认为它们相互重合,来完成堆和栈的压入和弹出操作,而实际上程序中的函数参数传送、局部变量、返回调用的地址等均是由编译程序进行编译完成的,它们的操作过程,就如同下面程序的演示一样。

```
char  * malloc();
int  * pi, * tos;
```

```
main()
 {
 int v;
  pi=(int * )malloc(5 * sizeof(int));
  if(! pi)
    {
     printf("allocation failure\n");
     return;
    }
  tos=pi;
   do{
       printf("pleas input value,push it(enter 0 then pop)\n");
       scanf("%d",&v);
       if(v! =0)
        push(v);
       else
        printf("pop this is it %d\n",pop());
       }while(v! =-1);
  }
  push(i)
   int i;
   {
    pi++;
    if(pi==(tos+5))
     {
      printf("stack overflow");
      exit();
     }
     * pi=i;
    }
  pop()
   {
    if((pi)==tos)
       {
       printf("stack underflow");
       exit();
       }
     pi--;
     return * (pi+1);
    }
```

5.5.4 示例 4——使用堆空间指针例

这个程序打印出了 N^2,N^3,N^4 的数值表,其中N 从 1 到 10。与一般程序不同的是,在该

程序中各数组的存储分配不是由编译程序安排的，而是由函数 malloc 将其分配在堆空间。该函数返回了一个 200 字节的堆空间指针 pi，当在主函数中调用 table()和 show()函数时，该堆指针作为参数传给了被调函数，因而在调用的函数中 p[][]变成了一个动态数组，p 数组的大小由函数自行决定。用这种方法可以加快程序运行速度，因参数传递时仅传送一个地址指针即可，而在函数中数组大小可随机进行确定。

```
#include<stdio.h>
#include<stdlib.h>
main()
 {
  int *pi;
  pi=(int *)malloc(100*sizeof(int));
  if(! pi)
   {
    printf("Not enough memore ");
    exit(1);
   }
  table(pi);
  show(pi);
 }
 table(p)
  int p[4][10];
  {
   int i,j;
   for (j=1;j<11;j--+)
    for(i=1;i<5;i++)p[i][j]=pwr(j,i);
  }
  show(p)
  int p[4][10];
  {
   int i,j;
   printf("%10s %10s %10s %10s\n","N","N^ 2","N^ 3","N^ 4");
   for(j=1;j<11;j++)
    {
     for(i=1;i<5;i++)
      printf("%10d",p[i][j]);
     printf("\n");
    }
 }
 pwr(a,b)
  int a,b;
  {
   int t=1;
```

```
for(;b;b--)
 t=t*a;
return t;
}
```

运行结果

N	N^2	N^3	N^4
1	1	1	1
2	4	8	16
3	9	27	81
4	16	64	256
5	25	125	625
6	36	216	1296
7	49	343	2401
8	64	512	4096
9	81	729	6561
10	100	1000	10000

5.6 动态存储例——链表

5.6.1 单链表结构

前面已经介绍过，堆是用来存储动态数据的，动态数据最典型的例子是链表。它的结构是这样的：形象地说，将若干个数据项按一定原则前后链接起来，每个数据项都有一个指向下一个数据项的指针，即这些数据项靠指针链成了一个表，如图 5.11 所示，设数据项包括三项：姓名、地址和指向下一个数据项的指针。每个数据项称为一个节点，链尾数据项的指针是空的，即 NULL，被定义为 0，即不指向任何地址。上述的链表称为单链表，可以看出链表放在存储器中，并不一定象数组一样，连续存放，也可以分开存放。由于链的各节点均带有指向下一个节点的地址，因而要找到某个节点，必须要找到上一个节点，如此上推，则可由第一个节点出发找到目的节点。链表在数据库建立和管理中用得比较普遍，如建立一个许多人的通信录，个人的通信数据再加一个指向下个人通信数据的指针即形成了链表，也就是一个通信数据库，在这个库中可以随意增加或删除某个人的通信数据项。由于数据项的多少是不能预知的，它根据当时情况增加或减少，这就是一种典型的动态数据结构。它无法一次分配好存储空间，只有用堆存储结构才能对这样的动态数据分配内存。

图 5.11 单链表

链表中的每个节点都具有相同的结构类型，它们由两部分组成，即数据部分（它们包含着一些有用信息，如通信录链表中的姓名、地址等），另一部分就是链的指针。下面就是定义的一个通信链节点的数据结构：

```
struct address
 {
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  char zip[10];
  struct address * next; /* pointer to next entry */
 list_entry;
}
```

该结构中前五个成员是该节点的信息部分，最后一个成员是指向同一结构类型的指针，即next 又指向一个同样结构类型的结点。

5.6.2 链表的建立

建立链表时，首先要将第一个节点的内容存入堆中，为此要将堆中能存入该节点内容的内存区域的首地址赋给一个指针。如 info 为一指针，存上述节点 list_entry 的区域可用 malloc()函数来分配，即：

```
info=(struct addr * )malloc(sizeof(list_entry));
```

当第一个节点存入由 info 指出的内存区后，再执行该函数，便得到下一个节点的存储地址 info，此时将该 info 赋给上一节点的 next，并将该节点内容存入 info 指出的内存区，这样两个节点就链接起来了，此过程反复多次，就可不断的将节点加入链表的尾端。根据节点多少可随机决定申请堆空间大小，即控制上述过程循环次数。例如用下面的函数就可实现这种功能。

```
/* enter name and address */
void enter()
 {
  struct address * info, * dls_store();
  for(;;)
   {
    info=(struct address * )malloc(sizeof(list_entry));
    if(!info)
     {
      printf("\n ont of memory");
      return;
     }

    inputs("enter name : ",info->name,30);
```

```
    if(! info->name[0])
    break; /* stop entering */;
    inputs("enter street : ",info->street,40);
    inputs("enter city : ",info->city,20);
    inputs("enter state : ",info->state,3);
    inputs("enter zip : ",info->zip,10);
    start=dls_store(info);
   }/* enter loop */
}
```

该函数中 inputs()函数用来接收节点信息并将其存入分配给该节点的堆空间中，而 dls_store()函数用来将新节点地址赋给上一节点的 next 指针，即起到链接作用。

下面是 inputs 函数，其中 prompt 参数代表输入节点信息时的说明提示，而 s 则是信息项要存入的内存地址，count 参数表示输入项的长度(字符个数)，局部变量字符数组 p[]是一个暂存信息的数组，它用来暂存输入的各项信息，当检查其长度不超过该信息项规定长度 count 后，便拷贝到相应的链表域中。

```
inputs (prompt,s,count) /* this function will input a string up to the length in
        count. This will prevent the string from overrunning its space and display a
        prompt message. */
char *prompt;
char *s;
int count;
{
 char p[255];
 do
  {
   printf(prompt);
   gets(p);
   if(strlen(p)>count)
    printf("\n too long\n");
   }while(strlen(p)>count);
   strcpy(s,p)
}
```

下面是 dls_store()函数，它将输入的节点的地址写到上一节点的 next 指针项去。

```
void dls_store(i)
struct address *i;
{
 static struct adderss *last=NULL;
 if(! last)                          /*若是空表，则输入的是首节点*/
 last=i;
 else
  last->next=i;
```

```
    i->next=NULL;
    Last=i;
}
```

其中定义的结构指针 last 是一个静态变量，初始值为 0(即 NULL)，这意味着在编译时将为该变量分配一个固定的存储空间以存放其值。因初始值为 0，这样在第一次调用该函数时，由于它代表一个空指针，因而把由 malloc()分配的第一个节点的地址赋给它，使 last 指向该节点(在程序中用 if(!last)last=i;来实现)，第二次调用该函数时，静态变量 last 已指向第一个节点的地址，故程序执行

```
last->next=i;
i->next=NULL;
last=i;
```

即将第二个节点的地址赋给了第一节点的 next 指针，如此反复调用，便建立起了由 n 次调用产生的 n 个节点的链了。

5.6.3 按升序排列的单链表

当链表各节点的数据项按升序排列时，可能出现三种情况：

① 即链表还未建立，是一个空链表，因而新输入的节点为链表的第一个节点，如图 5.12 所示。

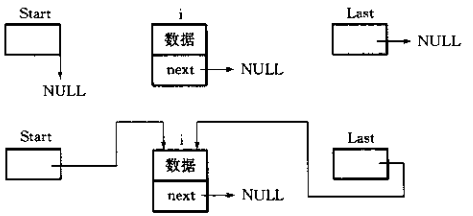


图 5.12 在空表中加入第一个节点

② 按升序原则，新输入的节点可能排在链表的中间某一位置，如图 5.13 所示。

③ 按升序原则，新输入的节点必须排在链表的末尾，如图 5.14 所示。

我们用一个链表头指针 start 表示链表最前一个节点的地址，i 表示新输入节点的地址，用 last 静态的结构地址指针表示链表的尾指针。这样若用 dlS_store(i,top)函数来实现节点插入功能时，第一次调用，last=0，表示是空链，故输入的节点是链表的头，即可写成：

```
if(! last)
 { i->next=NULL;
   last=i;
   return;
 }
```

以后的调用，它的值将发生变化。

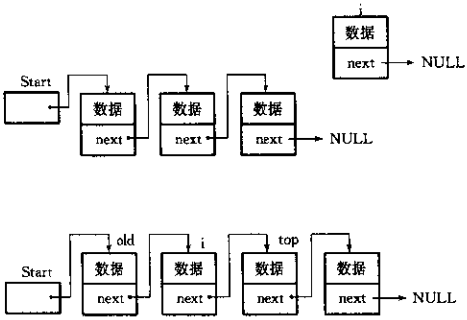


图 5.13　在链表中插入一个节点

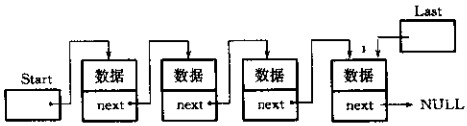


图 5.14　在链表尾加入一个节点

在一个有许多节点组成的单链表中，假设 top 指向第一个节点，将 top→name 与 i→name 进行比较，若 top→name<i→name，则将 top 指针后移到下一个节点，且将移前的 top 值暂存在 old 指针中。如此循环比较，直到找到一个 top，使得 top→name≥i→name，此时 old 指针就是上次 top 的值。这就意味着找到了一个插入点，即该 i 节点应该插在 old 指针所指向的节点与现时 top 指针指向的节点之间，即：

```
old=NULL;
while(top){
  if(strcmp(top→name,i→name)<0)
   {old=top;
    top=top→next;
   }
   else
    {if(old)                    /* 在链表中插入新节点 */
     {old→next=i;
      i→next=top;
      return start;             /* 首指针不变 */
     }
     i→next=top;                /* 若新节点的 name 小于首节点的 name 时
     return i;                  新节点变为首节点 */
    }
```

```
    }
```

当比较到最后一个节点为止，所有节点的 name 项均小于 i 节点的 name 项时，则该 i 节点应接在链表最后，即：

```
last→next=i;
i→next=NULL;
last=i;
return start;  /* 返回链表头指针 */
```

5.6.4 单链表的输出及节点删除

1. 单链表的输出

单链表输出比较简单，即将 top 指向链表中的第一个节点，输出节点信息，然后 top 再指向下一个节点，输出该节点信息，如此循环进行，直至 top=NULL 时为止，如输出节点的名字项可用下面函数表示：

```
void display(top)
struct address *top
{
  while(top)
  {
   printf(top→name);
   top=top→next;
  }
}
```

2. 单链表中一个节点的删除

删除一个节点，实际上是将链表中这个节点从链中断开，而将该节点前后的两个节点再接起来。断开的这个节点所占堆空间可以释放，这样就真正删除了。关键要在链表中找到这个节点，才能进行删除，这可能有四种情况：

① 删除的是第一个节点

② 删除的是中间某节点

③ 删除的是尾部节点

④ 链表中无此节点

为此先定义一个函数查寻要删除节点的指针值，设 top 的初值为链表的开始指针，n 为要找的名字，则有：

```
struct address *search(top,n)
struct address *top;
char *n;
 {
   while(top)
    {if (! strcmp(n,top→name))
       return top;              /* 找到要删除的节点指针 */
```

```
        top=top->next,          /* 继续找 */
    }
    printf("name not found\n")
    return NULL;                /* 没有找到 */
}
```

然后对找到的节点进行删除，用 delete()函数。设 info 是要删除的节点指针，old 是前一节点指针，

```
delete(info old)
struct address *info, *old;
{
 if(info)
   {if(start==info)             /* 若删除的是第一个节点 */
      {start=info->next;}       /* 将删除节点后面的节点地址赋给头指针 */
      else
       {
        old->next=info->next;   /* 被删除节点前的指针项指向下一节点 */
        last=old;               /* 若节点是链表尾，则该节点前的节点指针项指向空地址 */
       }
      free(info);               /* 释放删除节点占用的空间 */
   }
}
```

5.6.5 单链表程序例

下面是个可以建立通信录的单链表，可进行节点的删除、查找，基本构思正如上面所述，但在具体实现上，并不完全相同。程序中用 menu_select()函数进行程序功能的选择，按照用户键入的号码转去执行相应的函数，而完成要求的操作，程序中用 head 存储第一个节点的指针，它就是头指针。

下面就各函数的作用分述如下：

该程序产生一个简单的菜单，它是由主函数调用 menu_select 函数来实现的：

```
main()
        {char s[80],choice;
         struct addr *info;
         start=last=NULL;
         for(;;)
               switch(menu_select())
                     {case 1 : enter();break;
                      case 2 : delete(); break;
                      case 3 : search();break;
                      case 4 : exit(0);
                     }
        }
```

```
int menu_select()
        {char s[80];
         int c;
         printf("1--Enter a name\n");
         printf("2--Delete a record\n");
         printf("3--Search\n");
         printf("4--Quit\n");
         do
                 {printf("\nEnter your choice : ");
                  gets(s);
                  c=atoi(s);
                 }while(c<1 || c>4);
        return(c);
        }
```

链表的建立是通过 enter 函数来实现的，在该函数中用 malloc 函数给节点分配一个缓冲区，指针为 info。若 info 不空时，便调用 inputs 函数进行节点各项的输入：

```
void enter()
       {struct addr  * info, * des_store();
        void inputs(char * ,char * ,int);
        int n;
        for(n=0 ;;n++)
                {info=(struct addr  * )malloc(sizeof(record));
                 if(info ==NULL)
                         {printf("\n out of memory");
                          return;
                         }
                 inputs("enter name : ",info->name,30);
                 if(info->name[0]=='0')
                  break;
                 else
                          {inputs("enter street : ",info->street,40);
                           inputs("enter city : ",info->city,20);
                           inputs("enter zip : ",info->zip,10);
                           start=des_store(info,start);
                           if(n==0)
                           head=start;
                           }
                }
       }
```

inputs 函数通过实参传过来节点输入项的名称、节点各项元素值的存放地址和该项输入的最大字符个数(形参分别用 p1,s,count 表示)，马上在屏幕提示输入的项名，然后等待输入。当输入该项字符个数不超过限制时，便将该内容拷贝到该节点相应项中去，inputs 函

数如下：

```
void inputs(p1,s,count)
        char *p1,*s;
        int count;
        {char p[40];
         do
                    {printf("%s",p1);
                     gets(p);
                     if(strlen(p)>count)printf("\n too big\n");
                    }while(strlen(p)>count);
         strcpy(s,p);
        }
```

当节点各项内容输入后，最后要将该节点的指针 info 写到上一节点的 next 指针项中去，以便将节点链接起来，节点指针的写入，是用函数 des_store()来实现的。该程序是按节点输入的先后次序来链接的。若 last 为空时，表示是首节点，因而使 last=i，下次输入节点时，便会在其后链接了。否则使前一节点的指针 next=i 而使该节点的 next 为空，如此反复调用，将会把输入的节点一个个链接起来。

```
struct addr *des_store(i,top)
        struct addr *i,*top;
        {if(! last)
                {last=i;return(i);}
        else
                {top->next=i;i->next=NULL;last=i;return(i);}
        }
```

该程序中删除和查找节点，都是通过 find 函数来实现的，而 find 函数则根据要找的名字，从链表头指针指向的首节点开始，一个个顺着链表往后查找，若找到该节点，便把它的指针返回，若找不到，则返回一个空指针。而查找函数 search 和删除函数则根据返回的指针 info 显示该节点信息或删除该节点，find 函数如下：

```
struct addr *find(name)
        char *name;
        {struct addr *info;
nfo=head;
           while(info)
                 {if(! strcmp(name,info->name))
                         return(info);
                  else
                         info=info->next;
                  }
           return(info);
           }
```

查找函数 search 根据用户输入的查找名字调用 find 函数，当返回的指针为空时，表示没找到，否则显示找到的节点各项内容(它由 display 函数完成)。

```
void search()
      {char name[40];
       struct addr *info, *find();
       printf("enter name to find : ");
       gets(name);
       if((info=find(name))==NULL)
            printf("not found\n");
       else
            display(info);
      }
```

删除函数 delete，也调用 find 函数在链表中查找要删除的节点。若查找后返回的是首节点，则将链表头指针修改为指向下一个节点，即 head=info→next，并释放该节点占的内存空间。若不是首节点，则保存前一节点的指针于 p2 中，而 p1 为找到的节点指针，这时，将前一节点的指针项指向删除节点的下一个节点即可(p2→next=p1→next)。该函数如下：

```
void delete()
     {char s[80];
      struct addr *p1, *p2, *info;
      printf("enter name : ");
      gets(s);
      info=find(s);
      if(info! =NULL)
       {
       if(head==info)
        {
          head=info->next;
          printf("deleted : %s\n",info->name);
          free(info);
        }
       else
        {
          p1=head->next;
          while(info! =p1)
           {p2=p1;p1=p1->next;}
          p2->next=p1->next;
          printf("deleted : %s\n",info->name);
          free(info);
        }
       }
          else
           printf("%s not found! \n",info->name);
```

```
}
```

下面便是上述各函数有机组织后的程序清单。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct addr
       {char name[30];
        char street[40];
        char city[20];
        char ZIP[10];
        struct addr *next;
       }record;
struct addr *head, *start;
struct addr *last;
int menu_select();
void enter();
void delete();
void search();
main ()
         {char s[80],choice;
          struct addr *info;
          start=last=NULL;
          for(;;)
                 switch(menu_select())
                        {case 1 : enter();break;
                         case 2 : delete();break;
                         case 3 : search();break;
                         case 4 : exit(0);
                        }
         }
int menu_select()
         {char s[80];
          int c;
          printf("1--Enter a name\n");
          printf("2--Delete a record\n");
          printf("3--Search\n");
          printf("4--Quit\n");
          do
                {printf("\nEnter your choice:");
                 gets(s);
                 c=atoi(s);
                }while(c<1 || c>4);
          return(c);                          /*返回选择的项*/
```

```
        }
void enter()
        {struct addr *info, *des_store();
         void inputs(char *,char *,int);
         int n;
         for(n=0 ;;n++)
                {info=(struct addr *)malloc(sizeof(record));
                 if(info ==NULL)
                        {printf("\n Out of memory");
                         return;
                        }
                 inputs("enter name : ",info->name,30);
                 if(info->name[0]=='0')
                  break;
                 else
                        {inputs("enter street : ",info->street,40);
                         inputs("enter city : ",info->city,20);
                         inputs("enter ZIP : ",info->ZIP,10);
                         start=des_store(info,start);
                         if(n==0)              /*若输入的是第一个结点*/
                         head=start;           /*头指针指向第一个结点*/
                        }
                }
        }
void inputs(p1,s,count)
        char *p1, *s;
        int count;
        {char p[40];
         do
                 {printf("%s",p1);
                  gets(p);
                  if(strlen(p)>count) printf("\n too big\n");
                 }while (strlen(p)>count);
         strcpy(s,p);        /*将结点的输入项拷贝到指向的地址中去*/
        }
struct addr *des_store(i,top)
        struct addr *i, *top;
        {if(! last)
                {last=i;return(i);}      /*若是首结点时*/
        else
                {top->next=i;i->next=NULL;last=i;return(i);}
                        /*上一结点的指针项指向新结点*/
        }
void display(info)
```

```
        struct addr *info;
        {printf("%s\n",info->name);
         printf("%s\n",info->street);
         printf("%s\n",info->city);
         printf("%s\n",info->ZIP);
         printf("\n\n");
        }
void search()
        {char name[40];
         struct addr *info, *find();
         printf("enter name to find : ");
         gets(name);
         if((info=find(name))==NULL)
               printf("not found\n");
         else
               display(info);                    /*显示找到的结点*/
        }
struct addr *find(name)
        char *name;
        {struct addr *info;
         info=head;
         while(info)
               {if(! strcmp(name,info->name))
                     return(info);
                else
                     info=info->next;
               }
         return(info);
        }
void delete()
        {char s[80];
         struct addr *p1, *p2, *info;
         printf("enter name : ");
         gets(s);
         info=find(s);                          /*表示已找到要删去的结点*/
         if(info! =NULL)
          {
           if(head==info)                       /*若是首结点*/
            {
              head=info->next;
              printf("deleted:%s\n",info->name);
              free(info);                       /*释放该结点占用的缓冲区*/
            }
          else
```

```
    {                                          /* 若删除的不是第一个节点 */
      p1=head->next;
      while(info! =p1)
       {p2=p1;p1=p1->next;}
      p2->next=p1->next;
      printf("deleted : %s\n",info->name);
      free(info);
      }
  }
  else
    printf("%s not found! \n",info->name);   /* 若 info 是空,表示未找到 */
}
```

5.6.6 双链表

单链表有一缺点,就是无法反向操作,若某一链因破坏导致断裂,则整个链就被破坏,而无法恢复。但双链表可以弥补这一缺点,所谓双链表是指每个节点有两个指针项,一个指针指向其前面的节点,而另一个指针指向其后面的节点,如图 5.15 所示:

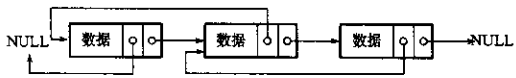


图 5.15 双链表

构造双链表的每个节点要有两个指针,以构成双链,因而双链表的节点除了数据项外,还须有存放两个指针的项,现仍以通信录为例,每个节点的结构如下:

```
struct address
{char name[40];
 char street[40];
 char city[20];
 char state[3];
 char zip[10];
 struct address *next;      /* 指向后面节点的指针 */
 struct address *prior;     /* 指向前面节点的指针 */
}info;
```

对于在双链表中插入、删除、输出链表各节点的指定项等,要涉及到两个链,但原理和单链表相同,因而和单链表的操作基本一样,不过要增加对每个节点中向前指向的指针处理,如构造一个双链表,可用下面的函数实现:

```
void dlstore(i)
struct address *i;
{
```

```
    static struct address *last=NULL;
    if(Last==NULL)              /*若是空链*/
    Last=i;                     /*i为第一个节点*/
    else Last->next=i;
    i->next=NULL;
    i->prior=Last;
    Last=i;                     /*Last为最后的一个节点*/
}
```

该函数将把输入的节点按输入的先后次序双向链接起来。

示例：下面是一个通信录程序，它们构成一个双链表，该程序中的几个函数的作用如同在单链表中介绍的那样。该程序增加了对链表的存盘功能(函数save())和将存在盘上的链表装入内存的功能(函数load())。

程序开始调用菜单选择函数menu_select()，返回一个用户选择的功能号以通过开关语句switch(menu_select)转向不同的功能。它们分别是：输入名字(即建立链表)、删去名字(即删去节点)、列出链表、检索、保存链表、从盘上装入链表。请参阅下面的清单。

```
#include <stdio.h>
struct address
 {
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  char zip[10];
  struct address *next;  /* pointer to next entry */
  struct address *prior; /* pointer to previous record */
 }list_entry;

struct address *start;      /* pointer to first in list */
struct address *last;       /* pointer to last entry */
void enter();
void display();
void search();
void save();
void load();

main()
  {
   char s[80],choice;
   struct address *info;
   start=last=NULL;  /* zero length list */
   for(;;)
    {
     switch(menu_select())
```

```
    {
        case 1 : enter();
          break;
        case 2 : delete();
          break;
        case 3 : list();
          break;
        case 4 : search();
          break;
        case 5 : save();
          break;
        case 6 : load();
          hreak;
        case 7 : exit();
      }
    }
  }
  /* select an operation */
menu_select()
 {
  char s[80];
  int c;
  printf("1, Enter a name\n");
  printf("2, Delete a name\n");
  printf("3, List the file\n");
  printf("4, Search\n");
  printf("5, Save the file\n");
  printf("6, Load the file\n");
  printf("7, Quit\n");
  do _
   {
    printf("\nEnter your choice : ");
    gets(s);
    c=atoi(s);
   }while(c<0 || c>7);
  return c;
  }
  /* enter name and address */
  void enter()
   {
    struct address *info, *dls_store();
    for(;;)
     {
```

```
        info=(struct address *)malloc(sizeof(list_entry));
        if(! info)
         {
          printf("\n ont of memory");
          return;
        }
        inputs("enter name : ",info->name,30);
        if(! info->name[0])
        break; /* stop entering */;
        inputs("enter street : ",info->street,40);
        inputs("enter city : ",info->city,20);
        inputs("enter state : ",info->state,3);
        inputs("enter zip : ",info->zip,10);
        start=dls_store(info,start);
        }/* enter loop */
}
inputs(prompt,s,count) /* this function will input a string up to the length in
        count. This will prevent the string from overrunning its space and display a
        prompt message. */
char *prompt;
char *s;
int count;
{
 char p[255];
 do
  {
   printf(prompt);
   gets(p);
   if(strlen(p)>count)
    printf("\n too long\n");
   }while(strlen(p)>count);
   strcpy(s,p);
}
/* Create a double linked list in sorted order.
   A pointer to the first element is returned because it is possible that a new ele-
   ment will be inserted at the start of the list. */
struct address *dls_store(i,top)   /* store in sorted order */
struct address *i;                 /* new element */
struct address *top;
/* first element in list */
 {
  struct address *old,*p;
  if(last==NULL){                  /* first at element in list */
```

```
    i->next=NULL;
    i->prior=NULL;
    last=i;
    return i;
   }
   p=top;  /* strat at top of list */
   old=NULL;
   while(p){
   if(strcmp(p->name,i->name)<0){
    old=p;
   p=p->next;
  }
  else{
   if(p->prior){
    p->prior->next=i;
     i->next=p;
     i->prior=p->prior;
     p->prior=i;
     return top;
    }
    i->next=p;
    i->prior=NULL;
    p->prior=i;
    return i;
    }
  }
  old->next=i;        /* put on end */
  i->next=NULL;
  i->prior=old;
  last=i;
  return start;
}
/* remove an element from the list */
delete()
 {
  struct address *info, *find();
  char s[80];
  printf("enter name : ");
  gets(s);
  info=find(s);
  if(info){
   if(start==info)
    {
```

```
        start=info->next;
        if(start)
         start->prior=NULL;
         else
          last=NULL;
        }
       else
        {
         info->prior->next=info->next;
          if (info! =last)
            info->next->prior=info->prior;
          else
            last=info->prior;
        }
        free(info); /* return memory to system */
      }
    }
  struct address *find(name)
  char *name;
  {
   struct address *info;
   info=start;
   while(info)
    {
     if(! strcmp(name,info->name))
      return info;
      info=info->next; /* get next address */
    }
    printf("name not found\n");
    return NULL; /* not found */
  }
  list()
   {
    register int t;
    struct address *info;
    info=start;
    while(info)
     {
      display(info);
      info=info->next;/* get next address */
     }
    printf("\n\n");
  }
```

```
void display(info)
 struct address *info;
 {
  printf("%s\n",info->name);
  printf("%s\n",info->street);
  printf("%s\n",info->city);
  printf("%s\n",info->state);
  printf("%s\n",info->zip);
  printf("\n\n");
 }
void search()
 {
  char name[40];
  struct address *info, *find();
  printf("enter name to find: ");
  gets(name);
  if(! (info=find(name)))
  printf("not found\n");
  else
  display(info);
 }
void save()
 {
  register int t;
  struct address *info;
  FILE *fp;
  if((fp=fopen("mlist","wb"))==NULL)
   {
    printf("cannot open file\n");
    exit(1);
   }
  printf("\n saving file\n");
  info=start;
  while(info)
   {
    fwrite(info,sizeof(struct address),1,fp);
    info=info->next; /* get next address */
   }
   fclose(fp);
 }
void load()
 {
  register int t;
```

```
struct address *info, *temp=NULL;
FILE *fp;
if((fp=fopen("mlist","rb"))==NULL)
{
 printf("cannot open file\n");
 exit(1);
}
while(start)
 {
  info=start->next;
  free(info);
  start=info;
 }
printf("\n loading file\n");
start=(struct address *)malloc(sizeof(struct address));
if(! start)
 {
  printf("out of memory\n");
  return;
 }
 info=start;
 while(! feof(fp))
  {
   if(1! =fread(info,sizeof(struct address),1,fp))
    break;
 /* get memory for next */
 info->next=(struct address *)malloc(sizeof(struct address));
 if(! info->next)
  {
   printf("out of memory\n");
   return;
  }
  info->prior=temp;
  temp=info;
  info=info->next;
 }
temp->next=NULL;  /* last entry */
last=temp;
start->prior=NULL;
fclose(fp);
}
```

5.7 关于联合的说明

联合是类似于结构的复合数据类型，联合中的各成员共同使用同一内存区域，该区域长度以最长的成员长度为准，各域员在使用该区域时，以相互覆盖的方法分时使用，后者覆盖前者的值。即在该内存区，一次仅允许驻留一个联合的成员值，如定义一个联合：

```
union data
    {int a;
     float b;
     char c;
    }x,y;
```

此时对联合变量 x 和 y 来说，为它分配的内存共占 4 个字节，即以 float b 的长度为准。为 x,y 分配的内存区如图 5.16 所示。有的人往往以为 x,y 也共占同一内存区，这是错误的。例如下面的程序是一个模拟学校人员数据管理的程序，在程序中定义了一个结构类型，它的成员中又含有一个联合。假设学校中有 10 个人，每个人的信息可看作一个结构，我们用数组 person[10]来表示，这样可定义为：

```
struct
  {
    char name[20];
    char sex;
    char job;
    union
      { int class;
        char group[20];
        }category;
  }person[10];
```

在内存中将开辟 10 个区域，用来装 10 个人员的信息，每个区域结构均如图 5.17 所示一样，其中联合所占的区域为 20 个字节，当其成员为 class 时，则仅使用 2 个字节，否则使用 20 个字节。

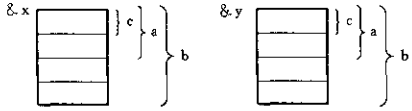


图 5.16 联合变量 x,y 各分配一内存区

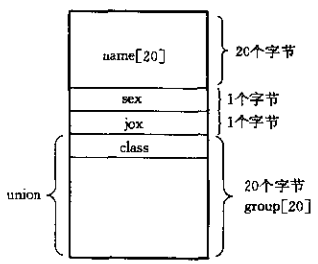


图 5.17 含有联合变量的结构变量内存图

下面的程序演示对联合的内存区使用情况。程序根据人员的工作状况 person[i].job 是学生't'还是教师's',而将所在班级或教研室装入联合的内存区中,因此凡是学生,其使用的联合内存区存放 person[i].category.class,而属于教师的,则在所使用的联合内存区存放 person[i].category.group。由于共分配了 10 个这样的区域,当在程序中 10 次调用 enter()函数后,则相应的这 10 个联合内存区均驻留了相应的联合成员,故当调用 print()函数时,则可把它们一一输出出来。enter()程序中字符串输入用 gets,而字符输入用 getchar()函数。由于使用该函数时,输入后要按回车键,此回车符仍保留在键盘缓冲区,并未由 getchar 取走,为了不影响下一个输入(取得 person[i].job 值),所以又加了一个 getchar(),为的是把上次遗留的回车符取走。

当在程序中用到大量数据时,采用联合可大大节省内存,当然也可使程序中一些复合变量的成员意义明确,提高可读性,并使调试方便。

联合内存区虽然不属于动态分配,但可理解该区域为动态使用区。

```
#include<stdio.h>
struct
  {
    char name[20];
    char sex;
    char job;
    union
     {
       int class;
       char group[20];
     }category;
   }person[10];
 void enter();
 void print();
 main()
  {
    int n,i;
    for(i=0;i<10;i++)
      enter();
    print();
  }
void enter()
 {
   int i;
   char p[10];
   printf("enter name ");
   gets(person[i].name);
   printf("enter sex ");
   person[i].sex=getchar();
```

```
    getchar();                                  /*取回车符*/
    printf("enter job ");
    person[i].job=getchar();
    getchar();                                  /*取回车符*/
        if(person[i].job=='s')
         {
           printf("enter class ");
           gets(p);
           person[i].category.class=atoi(p);
         }
        else
         {
           if(person[i].job=='t')
            {
              printf("enter group ");
              gets(person[i].category.group);
            }
           else
            printf("input error!");
            printf("\n\n");
         }
    }
void print()
  {
    int i;
    printf("\n");
    printf("Name            Sex Job class/group\n");
    for(i=0;i<10;i++)
     {
       if(person[i].job=='s')
         printf("%-20s%-5c%-5c%-6d\n",person[i].name,
                person[i].sex,person[i].job,person[i].category.class);
        else
         printf("%-20s%-5c%-5c%-20s\n",person[i].name,person[i].sex,
                person[i].job,person[i].category.group);
     }
  }
```

第 6 章

文 件

在我们的心目中，文件是指存放在磁盘上的一些信息的集合，它具有一个唯一的名字，通过名字可以对文件进行存取、修改、删除等操作。用户也可以建立自己的文件，用不同的名字和后缀(或叫扩展名)，这实际上就是磁盘文件。但从广义上来看，许多外部设备也可看作是一种文件，因为也可给它们取一个唯一的名字，对它们的操作也可用对磁盘文件相同的操作来实现，如在 DOS 中定义打印机为名字是 PRN 的文件，当向该文件写信息时，实际上就是打印输出；定义键盘为名字是 CON 的文件，当从该文件读信息时，实际上就是从键盘接收键入的字符数字。将物理设备看作是一种逻辑文件，用和磁盘文件操作一样的命令，可以简化设计、方便用户。C 语言中也采取了类似的作法，因而从广义上说，文件是指信息输入和输出的对象，磁盘文件是，打印机是，键盘是，显示器也是…。

由于程序中经常有大量对文件的输入输出操作，它经常构成了程序的主要部分，因而 C 语言提供许多输入输出操作函数，它们分别用于两种类型文件输入输出系统：即由 ANSI 标准定义的缓冲文件系统，或称标准文件(流)输入输出(I/O)系统；另一类是 ANSI 标准中没有定义的非缓冲文件系统或称非标准文件输入输出(I/O)系统。下面我们将分别介绍这两种文件系统的特点及其相应的 I/O 函数。

由于我们已经熟悉了通过键盘和显示器进行输入输出的一些函数如，printf、scanf 等，这些通过控制台(键盘、显示器等)进行 I/O 的操作，可以看作标准文件输入输出系统的一些特例，实际上在标准输入输出系统中的一些函数中，有关文件的参数(即文件结构指针或称流指针)，只要用标准设备的流指针代替，这些标准输入输出函数即成为控制台 I/O 函数。任何程序执行时，C 系统都定义了 5 个标准设备文件可供使用，因对文件进行 I/O 操作时，必须首先要打开或建立文件，这时将返回该文件结构的一个指针或文件代号(非标准文件 I/O 系统)，以后的 I/O 操作函数中，被操作的文件仅以该文件结构指针或文件代号形式出现。自动打开的五个标准设备文件的文件结构指针(在标准 I/O 系统中)和文件代号(当用在非标准文件 I/O 系统中时)将有一个规定值，即：

设备	标准文件 I/O 系统中的流指针名	非标准文件 I/O 系统中的文件代号
键盘(标准输入)	stdin	0
显示器(标准输出)	stdout	1
显示器(标准错误)	stderr	2
串行口(标准辅助)	stdoux	3
打印机(标准打印)	stdprn	4

这样，不论在标准文件 I/O 系统还是非标准文件 I/O 系统中，文件结构指针只要用上述的流指针或文件代号代替，则这些函数也均适用于控制台设备。但对用户来说，这两个文件

I/O系统中，仍然主要是用它们对磁盘文件进行 I/O 操作，因而后而将主要讲述对磁盘文件的 I/O 操作。

对于大量的信号采集和控制，需要使用接口设备。这些设备可作为 I/O 的一个部分，因而 C 也提供了几个专门用于接口输入输出的函数，我们也将进行介绍。

C 语言中提供的 I/O 函数是非常丰富的，概括起来有：控制台 I/O 函数、标准文件 I/O 函数（它们也适于控制台设备）、非标准文件 I/O 系统中的输入输出函数、接口输入输出函数。由于专门的控制台 I/O 控制函数大家已经熟悉，因而本书将不作介绍。

6.1 文本流和二进制流

在 C 中引入了流(stream)的概念。它将数据的输入输出看作是数据的流入流出，这样不管是磁盘文件或者物理设备(打印机、显示器、键盘等)，都看作是一种流的源或目的，视它们为同一种东西，而不管其具体的物理结构，即对它们的操作，就是数据的流入和流出。这种把数据的输入输出操作对象，抽象化为一种流，而不管它的源或目的具体结构的方法很有利于编程，而涉及流的输出操作函数可用于各种对象，与其具体的实体无关，即具有通用性。

在 C 中流可分成两类，即文本流(text stream)和二进制流(binary stream)。所谓文本流是指在流中流动的数据以字符形式出现，由于文本有行的限制，因而一行流完后，必须有行结束符，C 规定为“\n”，它代表了回车换行，因而在文本流中，当流入(文件)时“\n”被换成回车 CR 和换行 LF 代码 ODH 和 OAH(因变成 DOS 文件时，DOS 规定用回车换行符作为一行的结束)。而当流出时(从文件)，则 ODH 和 OAH 被换成“\n”符号(即符合 C 中规定的一行的行结束符)，例如下面有一个简单程序：

```
main()
 {
  printf("\n This is a program");
 }
```

当将这个程序存盘时，以文本流方式，其流的形式为：

m	a	i	n	(	)	\n	{	\n	p	r	i	n	t	f	(	"	\n	T	h	i	s		i	s		a		p	r	o	g	r	a	m	"	)	;	\n	}

当将其存盘后，将变成：

m	a	i	n	(	)	0d	0a	{	0d	0a	p	r	i	n	t	f	(	"	0d	0a	T	h	i	s		i	s		a		p	r	o	g	r	a	m	"	)	;	0d	0a	}

若将该文件读出时，即流出时，0d 和 0a 将转换成\n 符号。

二进制流是指流动着的是二进制数字序列，若流中有字符，则用一个字节的二进制 ASCII 码表示，若是数字，则用一个字节的二进制数表示。在流入流出时，对\n 符号不进行变换，因而流中写入的字节数和读出的字节数相同，例如 2001 这个数，在文本流中用其 ASCII 码表示为：

'2'	'0'	'0'	'1'
50	48	48	49

4 个字节

即将数字也表示成对应的字符流，而在二进制流中，则表示成相应的二进制数字：

00011111010001

若用 16 进制表示，则为：

07	D1

即两个字节。

这样对这个流读出时，与写入时的字节数相等，因为不再进行字符转换。按 C 语言规定，二进制流中对整型数用两个字节表示，长整型用 4 个字节，浮点数用 4 个字节表示。而双精度数则用 8 个字节表示。由此可以看出，二进制流比文本流节省空间，且不用进行对\n 的转换，这样可以大大加快流的速度，提高效率，因而对于含有大量数字信息的数字流，可用二进制流方式，对于含有大量字符信息的流，可采用文本流方式。

6.2 流与文件

在 C 语言中流就是一种文件形式，它实际上就表示一个文件或设备（从广义上说，设备也可看作是一种文件）。把流当作文件总觉不习惯，因而有人称这种和流等同的文件为流式文件。我们将它们等同起来，流的输入输出操作也称为文件的输入输出操作。当流到磁盘而成为文件时，意味着要启动磁盘写入操作，这样流入一个字符（文本流）或流入一个字节（二进制流）均要启动磁盘操作，将会大大降低传输效率（因磁盘是慢速设备），且降低磁盘驱动器使用寿命。为此使用了缓冲技术，即在内存为输入的磁盘文件开辟了一个缓冲区（缺省时为 512 个字节），当流到该缓冲区装满后，再启动磁盘一次，将缓冲区内容装到磁盘文件中去。这如同一个水桶（缓冲区），装满后再倒入一个大容器，而不必用小杯一次次的去接和倒。图 6.1 示出了这种缓冲技术。而从磁盘文件读出时，情形是类似的。

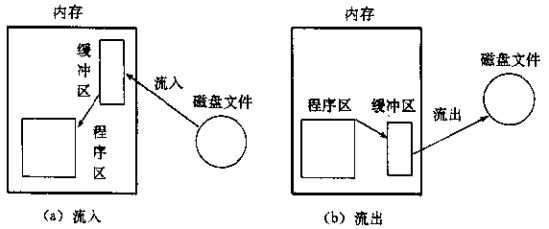


图 6.1 流的输入输出

在 C 语言中将此种文件输入输出操作称为标准输入输出，或称流式输入输出（因这种输入输出操作是 ANSI C 推荐的标准）。还有一种不带缓冲的文件输入输出，称为非标准文件输入输出或称为低级输入输出，它将由 DOS 直接管理，关于这两种输入输出文件系统下面将分别进行介绍。

6.3 标准文件的输入输出操作

C 语言提供了 4 种标准读写文件的方法，它们相应的也有 4 种函数，即：

1. fgetc 和 fputc 函数——读写一个字符；
2. fgets 和 fputs 函数——读写一个字符串；
3. fscanf 和 fprintf 函数——格式化读写；
4. fread 和 fwrite 函数——读写数据块。

这些函数将分别予以介绍。

在对文件进行读写之前必须用 fopen 函数将其打开，读写完后必须用函数 fclose 将其关闭。在程序中要进行磁盘文件的读写操作时，DOS 要求必须将文件名放入一指定存储区——文件控制块 FCB 中，并分配传输缓冲区，这可由 DOS 功能调用完成。而要实现这些操作，必须用打开文件的函数 fopen 实现。

当对文件操作完成后，必须进行关闭，因文件操作完后，应将缓冲区的信息写入磁盘文件中去并释放缓冲区，这样就保证了文件信息的完整性。关闭操作还起到了保护磁盘文件的作用，这样只要关闭的磁盘文件没有打开，任何对该磁盘文件的操作均不能进行。

DOS 的文件结构如图 6.2 所示。存放在磁盘上的文件被划分成一个个记录(RECORD)，一个记录可以包含一个或一个以上字节的内容，一个记录所含字节数称为记录长度，对一个指定文件，记录长度是一定的。记录以数据块的形式存放在磁盘中，每个数据块最多可包含 128 个记录。这些记录的编号从 0 到 127，数据块编号从 0 依次增加。要找到一个文件中的某记录，必须要知道数据块的编号和在此块中记录的编号。每个文件的最后一块不一定填满记录，根据对磁盘文件的存取情况，DOS 提供了四种存取方式：

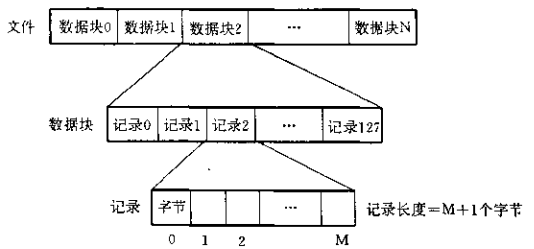


图 6.2 DOS 的文件结构

① 顺序存取方式

当对磁盘文件按这种方式存取时，是对文件的记录按顺序从头至尾进行读或写，一次只能顺序存取一个记录。

② 随机存取方式

当要对文件中的某个特定记录进行存取时，可采用此种方式。在这种方式下，一次只能存取一个任意指定的记录。

③ 随机块存取方式

当要对整个文件一次完成存取操作，或指定一个相对记录号，从该记录开始对指定数目的记录一次完成存取操作的方式，称为随机块存取方式，因为要存取的这些记录是分布在不同的块中，所以称为随机块存取方式。

④ 文件代号存取方式

当在给定的路径目录下建立或打开这种文件时，DOS 就回送一个该文件的代号，有人称为句柄(HANDLE)，以后就可以用该代号对文件进行存取了。

对前三种文件存取方式，C 语言将其用标准文件输入输出方式代替。而文件代号存取方式下，C 语言用相应的低级(非缓冲)的文件输入输出代替，后面将分别介绍这些存取方式。

6.3.1 标准文件输入输出

在 C 程序中当建立或调用一个磁盘文件时，必须要知道与该文件对应的内存缓冲区的地址、文件当前的读写位置、文件的操作方式等信息，即是文本文件还是二进制文件，是读还是写等，这些信息存放在一种叫 FILE 的数据结构中。每当我们打开一个文件时(若没有，则表示要建立，若已存在，则打开就意味着要对该文件进行读或写)，C 语言就在内存中建立了一个与该文件对应的 FILE 结构，并返回这个结构的指针(地址)。这样，对该文件的读写操作，都以该指针为参考。读者无需对这个结构的内容进行控制。如对一个文件进行字符输入，只要在文件字符输入函数中指明该文件对应的 FILE 结构指针 fp 即可，即 fputc(ch,fp)执行时，就可对由 fp 指向文件输入一个字符。至于内存缓冲区在哪，文件读写位置如何变化等一系列问题，用户均无需过问，这样就解脱了用户许多烦琐的工作。当要用汇编语言实现类似的功能时，用户要改变的参数就多了。后面讲到的一些标准的文件输入输出操作函数均类似于这种形式，我们将一一介绍。

6.3.2 关于 FILE 数据结构

前面说的 FILE 这个数据结构，其定义在 stdio.h 头文件中，定义如下：

```
typedef struct{
        short level;
        unsigned flags;
        char fd;
        unsigned char hold;
        short bsize;
        unsigned char *huffer;
        unsigned char *curp;
        unsigned istemp;
        short token;
    }FILE;
```

这是 Turbo C 中使用的定义，不同的 C 编译程序，可能使用不同的定义，但基本含义变化不会太大，因为它最终都要通过 DOS 去控制这些文件。

FILE 结构中，flags 字段是一个 10 位的标志字，其各位含义如下：

位	代表符号	含义
0	_F_READ	读
1	F_WRIT	写
2	_F_BUF	由 fclose 释放缓冲区
3	_F_LBUF	行缓冲
4	_F_ERR	出错标志
5	_F_EOF	EOF 文件尾标志
6	_F_BIN	二进制方式
7	_F_IN	在进行输入
8	_F_OUT	在进行输出
9	_F_TERM	文件是一个终端

其中_F_BUF 是 Turbo C 为每个文件提供的缓冲区，缓冲区大小在 bsize 字段内，其缺省值为 512 个字节，但可以由用户用 setbuf 函数重新进行设置，一旦进行了设置，_F_BUF 将被置 1。当关闭该文件时(用 fclose)，将释放开辟的缓冲区。文件缓冲区的指针则存放在 FILE 结构的 buffer 字段内，FILE 结构中的 level 和 curp 字段用来存放有关缓冲的一些信息。

flags 字段中的_F_LBUF 位为 1 时，表示为行缓冲。即当文件是一个终端设备时，缓冲区中所存信息遇到一个回车符时，便将回车符前的这一行信息送到终端(当输出时)，或将该行信息送到程序区(当输入时)。当_F_LBUF 为 0 时，表示是一个磁盘文件，此时缓冲区将装满一个扇区的内容(512 个字节)，然后送往磁盘文件(当输出时)，或从磁盘文件读入一个扇区的内容，然后送往程序区，送完后，再次启动读入或写磁盘。

FILE 结构中的 hold 字段用来存储用户调用 ungetc 函数时返回到输入流的一个字符，istemp 字段表示该文件是否为临时文件，临时文件是由 tmpfile 函数建立的，关于这方面内容可参阅 Turbo C 参考手册。

值得一提的是 FILE 结构中的 fd 字段，它实际上是文件的代号，如在下面要介绍的低级(非缓冲)文件输入输出中的文件代号(文件句柄)。DOS 为每个文件代号设置了有关的一些数据结构来存储与文件有关的信息，如文件指针(它表示当前打开文件的开始读写位置)，文件指针是否到文件尾，文件建立日期及时间，文件长度、属性，文件在哪一个磁盘驱动器上等一系列信息，因而我们用一些标准的文件输入输出函数可以对文件中的字符、字符串、块、记录等进行存取，这都是通过控制文件指针来实现的。注意！不要把文件指针和 FILE 结构指针(流指针)混为一谈，它们代表两个不同的地址。文件指针指出了对文件当前要读、写的数据位置，而 FILE 结构指针是指出了打开文件所对应的 FILE 结构在内存的地址，这个指针又称为流指针，实际上它本身也包含了文件指针的信息。流指针中的各字段是供 C 语言内部使用，用户不应该存取它的任何字段。

6.3.3 标准文件打开函数 fopen()

文件的打开操作(用 fopen 函数)表示将给用户指定的文件在内存分配一个 FILE 结构区，并将该结构的指针返回给用户程序，以后用户程序就可用此 FILE 指针来实现对指定文件的存取操作。当使用打开函数时，必须给出文件名、文件操作方式(读、写或读写)，如果该文件名不存在，就意味着要建立(只对写文件而言，对读文件则出错)，并将文件指针指向文件开头。若已有一个同名文件存在，则删除该文件，若无同名文件，则建立该文件，并将文件

指针指向文件开头。

该函数的格式为：

fopen(char *filename,char *type);

其中 *filename 是要打开的文件名指针，一般用双引号括起来的文件名表示，也可使用双反斜线隔开的路径名。而 *type 参数表示了对打开文件的操作方式，其可采用的操作方式如表 6.1 所示：

表 6.1 文件操作方式

方式	含义
"r"	打开，只读
"w"	打开，文件指针指到头，只写
"a"	打开，指向文件尾，在已存在的文件中追加
"rb"	打开一个二进制文件，只读
"wb"	打开一个二进制文件，只写
"ab"	打开一个二进制文件，进行追加
"r+"	以读/写方式打开一个已存在的文件
"w+"	为读/写建立一个新的文本文件
"a+"	为读/写打开一个文本文件进行追加
"rb+"	为读/写打开一个二进制文件
"wb+"	为读/写建立一个新的二进制文件
"ab+"	为读/写打开一个二进制文件进行追加

从表中可以看出，r 和 r+是为了存取一个已存在的文件。若要建立一个新文件，则应用“w”。而“a”方式，则是向一个已存在的文件尾进行追加。“r+”，“w+”和“a+”则既可读也可写。当对文件读写交替进行时，要注意文件指针的位置。可以用 fseek 或 rewind 函数正确地调整文件指针位置，后面将要介绍。

当操作方式中没有用 b 对文件类型说明时，一般指文本文件(它由在 fentl.h 头文件中定义的全局变量 fmode 来决定，其缺省值为文本方式，但为了更清楚，建议使用 t 进行说明，使用方式如同用 b 一样，如“rt”表示只读文本文件)。

指定操作文件类型还可用定义全局变量_fmode 来实现如上所述_fmode 在 Turbo C 的头文件 fentl.h 中定义，其缺省值是文本方式，我们也可以重新定义它，这样在程序中打开的文件均表示为该种类型，除非另行定义。比如我们在自己程序的开头定义两个常量：

#define O_TEXT 0x4000

#define O_BINARY 0x8000

若需要使程序某处以后打开的文件均为二进制文件时，可使全局变量_fmode 取值为 O_BINARY，即

_fmode=O_BINARY;

若又要在某处定义随后打开的文件为文本文件时，可使全局变量_fmode 取值为 O_TEXT，

即

```
_fmode=O_TEXT;
```

当用 fopen()成功的打开了一个文件时，该函数将返回一个 FILE 指针，如果文件打开不成功，将返回一个 NULL 空指针。如想打开 test 文件，进行写，可用下面的方法：

```
FILE *fp;
 if((fp=fopen("test","w"))==NULL)
 { printf("File cannot be opened\n");
   exit();
 }
 else
  printf("Fill opened for writing \n");
     ⋮
  fclose(fp);
```

下面是打开一个由路径名指明的文件：

```
#include<stdio.h>
 main()
  { FLLE *fp;
    if((fp=fopen("c:\\mydir\\myfile.dat","rb+"))=NULL)
    {
      printf("FILE cannat be opeened\n");
      exit(1);
    }
    else
      {
        printf("FILE opened for reading and writing/n");
        ⋮
        fclose(fp);
        ⋮
          }
```

上面程序中使用 exit()函数返回操作系统，该函数也将关闭所有打开的文件。一般使用时 exit(0)表示程序正常返回，若函数参数为非零值，表示出错返回，如 exit(1)等。

DOS 操作系统对同时打开的文件数目是有限制的，当 DOS 启动时，要查看系统配置文件 CONFIG.SYS，我们可以改变登录在该文件中的命令行来改变 DOS 的运行环境，比如在用户的程序中要打开超过 10 个的文件，再加 DOS 必须要打开的文件数，假设总数不超过 20，则可在 CONFIG.SYS 文件中写上：

```
files=20
```

这样的命令行，可以命令 DOS 同时打开 20 个文件。若没有这条命令，DOS 缺省打开的文件数只允许同时打开 5 个文件。

另外还要注意的是，当执行一个 Turbo C 程序时，自动地打开了 5 个文件(设备文件)，并给出了对应的 5 个 FILE 指针，这 5 个文件的 FILE 指针是：

stdin　　标准输入(键盘)
stdout　　标准输出(显示器)
stdaux　　标准辅助输入输出(异步串行口)
stdprn　　标准打印(打印机)
stderr　　标准错误输出(显示器)

因此使用这些设备时(从流的概念上来看,也是一种文件)不必再进行打开和关闭,它们由C编译程序自动完成,用户可以任意使用。

例如将1到10共十个自然数从打印机输出,则可指定用标准打印设备文件stdprn(因它已自动打开,故无需再打开),用fprintf函数向它输出即可。关于fprintf将在后面介绍。

```
#include<stdio.h>
main()
{
  int i;
  for(i=0;i<=10;i++)
fprintf(stdprn,"%d\n",i);
}
```

为了能在屏幕和打印机上同时输出结果,可采用如下程序。因printf函数定为向stdout输出,即屏幕,用fprintf()定向向打印机输出,下面的程序将把键盘输入的字符同时在屏幕和打印机上输出,键入#号时,程序结束,程序中第一个getchar()函数用来接收字符。第二个getchar()函数用来接收回车符,以免第二次输入时,将回车符当作新的输入。

```
#include<stdio.h>
main()
{
  char ch;
    while(ch! ='#')
    {
      ch=getchar();
      getchar();
      printf("%c",ch);
      fprintf(stdprn,"%c",ch);
    }
}
```

对于这些预先打开的设备文件,可以由用户用freopen()函数重定向,即改变原来这些设备的FILE指针的指向,使其对应新的文件,freopen()函数的说明是:

FILE *freopen(char *filename,char *type,FILE *stream);

它将用指定的文件名代替原来的流指针对应的设备文件,即原来已打开的设备文件的FILE指针,指向了新文件,这样原来的设备文件将被关闭。type参数含义同fopen()函数中的type。

这个函数常常用来关闭标准的输入输出流文件,而进行输入输出的重定向。

例如在用 printf 函数时，规定向 stdout 标准输出设备输出，即输出到屏幕。为了向打印机输出，则必须关闭 stdout 设备，而用 PRN 设备(DOS 规定的打印机文件名)来代替，故可用 freopen()函数来完成这个工作。如将 10 个自然数(1 到 10)用 printf 函数在打印机上输出，可用下面的程序来实现，当然也可如上面示例程序中那样，用 fprintf()函数输出到打印机，即输出文件指明为 stdprn 即可。

```
#include<stdio.h>
main()
{
  FILE *stream;
  int i;
  stream=freopen("PRN","w",stdout);
  for(i=0;i<=10;i++)
  printf("%d\n",i);
```

6.3.4 标准文件关闭函数 fclose()

文件操作完成后，必须要用 fclose()函数进行关闭，这是因为对打开的文件进行写入时，若文件缓冲区的空间未被写入的内容填满，这些内容不会写到打开的文件中去而丢失。只有对打开的文件进行关闭操作时，停留在文件缓冲区的内容才能写到该文件中去，从而使文件完整。再者一旦关闭了文件，该文件对应的 FILE 结构将被释放，从而使被关闭的文件得到保护，因为这时对该文件的存取操作将不会进行。文件的关闭也意味着释放了该文件的缓冲区。

fclose()函数的说明如下：

```
int fclose(FILE *stream);
```

它表示该函数将关闭 FILE 指针(流指针)对应的文件，并返回一个整数值。若成功地关闭了文件，则返回一个 0 值，否则返回一个非零值。常用下面方法进行测试

```
if(fclose(fp)!=0)
   {
     printf("\n FILE cannot be closed");
     exit(1);
   }
 else
   printf("\n FILE is now closed");
```

当打开了多个文件进行操作，而又要同时关闭时，可采用 fcloseall 函数，它将关闭所有在程序中已打开的文件(stdin,stdout,stdaux,stdprn,stderr 除外)，其函数说明如下：

```
int fcloseall(void);
```

该函数将关闭所有已打开的文件，将各文件缓冲区未装满的内容写到相应的文件中去，接着释放这些缓冲区，并返回关闭文件的数目，如关闭了 4 个文件，则当执行

```
n=fcloseall();
```

时，n 应为 4。

6.3.5 标准文件的读写

1. 读写文件中字符的函数

这类函数执行一次，仅能读写文件中的一个字符，主要有：

```
int fgetc(FILE *stream);
int fgetchar(void);
int fputc(int ch,FILE *stream);
int fputchar(int ch);
int getc(FILE *stream);
int putc(int ch,FILE *stream);
```

其中 fgetc()函数将把由流指针指向的文件中的一个字符读出，例如：

```
ch=fgetc(fp);
```

将把由流指针 fp 指向的文件中的一个字符读出，并赋给 ch，当执行 fgetc()函数时，若当时文件指针(注意不要和流指针混为一谈)指到文件尾时，即遇到文件结束符 EOF(其对应值定义为-1)，该函数返回一个-1 给 ch，在程序中常用检查该函数返回值是否为-1 而判断已读到文件尾否，从而决定是否继续读。

例如有下面的程序

```
#include "stdio.h"
main()
 {
   FILE *fp;
   char ch;
   if((fp=fopen("myfile. txt","r"))=NULL)
     {printf("file cannot be opened \n");
        exit(1);
     }
   while((ch=fgetc(fp))! =EOF)
   fputc(ch,stdout);
   fclose(fp);
 }
```

该程序以只读方式打开 myfile. txt 文件，在执行 while 循环时，文件指针每循环一次后移一个字符位置。用 fgetc()函数将文件指针指定的字符读到 ch 变量中，然后用 fputc()函数在屏幕显示，当读到 ch 为文件结束标志-1 时(即 EOF)，便关闭该文件。

上面程序中用到了 fputc()函数，该函数将字符变量 ch 的值写到流指针指定的文件中去，由于流指针用的是标准输出(显示器)的 FILE 指针 stdout，故读出的字符将在显示器上显示。又比如：

```
fputc(ch,fp);
```

该函数执行结果，将把 ch 表示的字符送到流指针 fp 指向的文件中去。

在 Turbo C 中，putc 是函数 fputc 的宏，而 getc 是函数 fgetc 的宏，即：

#define putc(ch,fp),fputc(ch,fp)

#define getc(fp),fgetc(fp)

因而 putc()的功能和 fputc()相同,getc()和 fgetc()相同,大家熟悉的 putchar 和 getchar 函数其实也是 fputc 和 fgetc 的宏,这时流指针定义为标准输出 stdout 和标准输入 stdin,即:

#define putchar(c)　　fputc(c,stdout)

#define getchar()　　fgetc(stdin)

fgetchar 是一个 fgetc(stdin)的宏,即 fgetc()函数中流指针参数定义为标准输入 stdin。

fputchar 是一个 fputc(stdout)的宏,即 fputc()函数中流指针参数定义为标准输出 stdout。

在使用上述函数时,有一个要注意的问题是在使用接收字符的变量 ch 时,要把它定义为整型的,即:

int ch

而没有定义为字符型的,这是因为用到该变量的函数自动将其变换为无符号的字符,而将其整型数高位字节忽略,即实际得到的仍是一个字符,另一原因是,当函数返回文件尾的信息 EOF 时,它并不是一个字符,而代表-1 值,故若定义 ch 为字符型,这个值便和字符代码不同(因没有任何一个字符的 ASCII 码会取-1)。如若定义

char ch;

当执行从文件中循环读字符并检查是否到文件尾时,即:

while ((ch=fgetc(fp))! =EOF)

该循环将无休止,因永远不会有 ASCII 码为-1 的字符。

2. 读写文件中字符串的函数

这些函数将从指定的文件中读写指定长度的字符串,主要有:

```
char *fgets(char *string,int n,FILE *stream);
char *gets(char *s);
int fprintf(FILE *stream,char *format,<variable-list>);
int fputs(char *string,FILE *stream);
int fscanf(FILE *stream,char *format,<variable-list>);
```

其中 fgets()函数将把由流指针指定的文件中的 n-1 个字符,读到由指针 stream 指向的字符数组中去,例如:

fgets(buffer,9,fp);

将把 fp 指向的文件中的 8 个字符读到 buffer 内存区去,buffer 可以是定义的字符数组,也可以是动态分配的内存区。

fgets()函数读完指定的 n-1 个字符后返回,若在读到一个换行符('\n')时,即使没有读够 n-1 个字符,也将停止,此时将在读入的字符串后加一个串结束符("\0")。这就是为什么 fgets()函数只能读 n-1 个字符的原因,原来最后一个字符位置是为填加'\0'而留。如想读取的字符串是"main()\n",这时存入 buffer 中的字符串将是

m	a	i	n	(	)	\n	\0

即没有读够 8 个字符就遇到了换行符,故停止,最后加一个空字符。

fgets()函数执行完后，返回一个指向该串的指针。如果读到文件尾或出错，则均返回一个空指针 NULL，所以只有用 ferror()函数或 feof()函数来测定是出错还是到了文件尾，例如下面的程序用 fgets()函数读 test.txt 文件中的第一行字符并显示出来：

```
#include "stdio.h"
main()
{ FILE *fp;
  char str[128];
  if((fp=fopen("test.txt","r"))==NULL)
    { printf("cannot open file\n");
      exit(1);
    }
    while(! feof(fp))
      {
       if(fgets(str,126,fp)! =NULL)
          printf("%s",str);
      }
      fclose(fp);
    }
```

函数 gets()从标准输入 stdin 中(即键盘)读字符并把它们存入由 s 指向的存储区，当读到换行符或遇到文件结束标记 EOF 时，该过程结束。注意，读到的换行符将被转换成'\0'写到字符串最后，作为字符串的终止符号。

gets()函数执行时，只要未遇到换行符或文件结束标记，将一直读下去。因此读到什么时候为止，这要由用户进行控制，否则可能造成存储区的溢出。

fputs()函数向指定文件写入一个由 string 指向的字符串，该字符串必须是以空字符"\0"结束的串，该空字符将不写到文件中去。注意!string 指向的字符串也可用数组名代替，也可用双引号括起来的字符串代替。

该函数正确执行后，将返回写入的字符数。当出错时，将返回－1。例如：

```
c=fputs("Hello\0",fp);
```

将把"Hello"字符串写到由 fp 指定的文件中去。

fprintf()和 fscanf()函数和 printf()及 scanf()函数类似，不同之处是 printf()函数是向 stdout 标准输出设备(显示器)输出，scanf()是从 stdin 标准输入设备(键盘)输入，fprintf()和 fscanf()函数则是向流指针指定的文件输出或由流指针指定的文件输入。当将流指针 stream 定义为 stdout 和 stdin 时，这两个函数的功能就和 printf()和 scanf()相同了。这两个函数中的参数 char * format 和<variable_list>也同 printf()和 scanf()函数中的一样，表示输出或输入格式及输出的序列及输入的序列(存储单元)。例如：

```
fprintf(fp,"%s %d %f",name,num,score);
```

它将按规定的格式"%s %d %f"将 name,num,score 变量值输出到由 fp 指定的文件中去。

6.3.6 清除和设置文件缓冲区的函数

Turbo C 提供了二个清除文件缓冲区的函数和两个设置文件缓冲区的函数，现作以下

介绍：

1. 清除文件缓冲区函数

```
int fflush(FILE * stream);
int flushall(void);
```

fflush()函数将清除由 stream 指向的文件缓冲区的内容，常用于写完一些数据后，立即用该函数清除缓冲区，以免误操作时，破坏原来的数据。

flushall(void)函数将清除所有打开的文件所对应的文件缓冲区。

2. 设置文件缓冲区函数

```
void setbuf(FILE  * stream,char * buf);
int setvbuf(FILE  * stream,char  * buf,int type,unsigned size);
```

这两个函数将使得打开文件后，用户可建自己的文件缓冲区(由指针 buf 给出地址)，而不使用 fopen()函数打开文件时设定的缓冲区。

对于 setbuf 函数，buf 指出的缓冲区长度由头文件 stdio.h 中定义的宏 BUFSIZE 的值决定，缺省值为 512 字节。当选定 buf 为空时，setbuf 函数将使得文件 I/O 不带缓冲。而对 setvbuf 函数，则由 malloc 函数来分配缓冲区。参数 size 指明了缓冲区的长度(必须大于 0)，而参数 type 则表示了缓冲的类型，其值可取如下值：

type 值	含义
_IOFBF	文件全部缓冲，即缓冲区装满后，才能对文件进行写入或读出。
_IOLBF	文件行缓冲，即缓冲区接收到一个换行符时，才进行写入或读出。
_IONBF	文件不缓冲，此时参数 buf、size 均被忽略，直接写文件或读文件，不再经过缓冲区进行缓冲。

当 setvbuf 成功调用后，将返回 0，否则将返回非零值，这可能是 buf、size 参数设定错或由 malloc 分配存储空间时，没有足够的缓冲区。setbuf 无返回值。

使用这两个函数时要注意，它们必须用在打开文件之后或调用 fseek 之后，否则文件缓冲区将出现两个，使数据混乱。

6.3.7 应用例

下面的程序用于读一个文件(file1.c)，然后将文件内容反序写到另一个文件(file2.c)中去，程序中用到了前面叙述过的几种函数。在程序中定义了一个 2000 字节的缓冲区 buffer，注意！这不是读文件时由 DOS 或 Turbo C 开辟的那个内存缓冲区，这个 buffer 用于存放从 file1.c 文件中读出的字符。当定义的长度不够装下读出的文件时，便显示 beffer 不够的信息而退出。

```
#include<stdio.h>                          /* 包含有关文件的读写函数定义 */
main()
 {
   FILE  *fp1, *fp2;                       /* 定义两个 FILE 结构指针 */
   char buffer[2000];                      /* 定义存放读出字符的缓冲区 */
   int ch,i;
   if (! (fp1=fopen("file1.c","r")))       /* 为读打开一个文本文件 */
```

```
    { printf("File cannot be opened:file1.c\n");
      exit(1);
    }
  if (! (fp2=fopen("file2.c","w")))          /*为写打开一个文本文件*/
    { printf("File cannot be opened:file2.c\n");
      exit(1);
    }
 i=0;
 while((ch=fgetc(fp1))! =EOF)                /*检查是否读到文件末尾*/
  {  buffer[i++]=ch;                         /*顺序送入缓冲区*/
     if (i>=2000)
     {printf("beffer not enough");           /*若i=2000仍未到文件尾,则报错*/
        exit(1);}
  }
 while(--i>=0)
  fputc(buffer[i],fp2);                      /*反序写入打开的文件*/
 fclose(fp1);
 fclose(fp2);                                /*关闭文件*/
 }
```

下面的程序用于对一个数据文件(fnum1.dat 文件中全是数据)进行从小到大的排序，并将排的结果写到另一数据文件 fnum2.dat 中去。该程序中用到 Turbo C 中的一个排序函数 qsort()，它的定义形式是：

```
void qsort(void  *base,int nelem,int width,int (*fcmp)());
```

其中参数 *base 表示待排序表中的第一个数据的地址，nelem 是待排序数据的个数，width 则表示以字节为单位时数据的长度。而 fcmp 则是用户自己定义的比较函数的指针。在本程序中定义为 compi，qsort 函数自动地将两个顺序排列元素送到用户定义的比较函数中，根据比较结果(返回一整数)，将其位置进行调整，当反复调用比较函数，所有待排列的数据均排序完后，qsort()函数执行结束。

```
 # include <stdio.h>
 # include <stdlih.h>
int huffer[200];
main()
 { FILE  *fp1,*fp2;
   int i,sz,nch;
   int compi();
   if((fp1=fopen("fnum1.dat","r"))==NULL)
     { printf("File cannot be opened:fnum1.dat\n");
       exit(1);
     }
   if((fp2=fopen("fnum2.dat","w"))==NULL)
     { printf("File cannot be opened:fnum2.dat\n");
       exit(1);
```

```
    }
    sz=0;
    while(! feof(fp1))                          /* 检查是否到文件尾,否则顺序读文件 */
      { fscanf(fp1,"%d",&nch);
        buffer[sz++]=nch;                       /* 存缓冲区 */
      }
    qsort(buffer,sz,sizeof(int),compi);         /* 进行排序 */
    for(i=0;i<=sz;i++)
      fprintf(fp2,"%d ",buffer[i]);             /* 顺序输出到 fp2 指向的文件中去 */
    fclose(fp1);
    fclose(fp2);
  }
int compi(int *n1,int *n2)
{
  return(*n1-*n2);                              /* 返回两个数比较结果 */
}
```

6.3.8 文件的随机读写函数

前面介绍的文件的字符/字符串读写,均是进行文件的顺序读写,即总是从文件的开头开始进行读写。当文件打开时,其文件指针均指向文件的第一个字符或数,随后顺序读写下一个数,使文件指针移动到下个数的位置。下次再读写时,只能接在其后继续读写。若能从文件的任意位置开始读写,即可任意移动文件指针,则可克服顺序读写的限制。C 语言提供了移动文件指针和随机读写的函数,它们是:

① 移动文件指针的函数

```
long ftell(FILE *stream);
int rewind(FILE *stream);
fseek(FILE *stream,long oifset,int origin);
```

函数 ftell()用来得到文件指针离文件开头的偏移量(即偏移的字节数)。当返回值是-1L时表示出错,例如:

```
Long i;
if((i=ftell(fp))==-1L) printf("A file error has occurred\n");
```

rewind()函数用于把文件指针移到文件的开头,当移动成功时,返回 0,否则返回一个非零值。下面的程序是将一个文本文件 fstr1.txt 的字符串进行排序,然后再写回该文件去。由于用 fscanf()函数将该文件顺序读出并放入用户定义的缓冲区后,文件指针已指到 fstr1.txt 的文件尾了,所以若通过排序后,再将 buffer 中排好序的字符串写回原文件时将出错,故而采用 rewind()函数将文件指针重移到文件头,然后再写。这样产生的 fstr1.txt 文件便是排好序的字符串文件了。

为了验证该程序,特此附了一个写 fstr1.txt 文件的程序,它将 10 个城市名写入该文件中,假设字符串长度为 10。当用排序程序将该文件内容排序后,若用 DOS 的 type 命令在屏幕上显示出来时,可以看出,它们已按字母的顺序进行了排序。

排序程序

```
# include <stdio.h>
# include <string.h>
char *buffer[200];
main()
 {
  FILE *fp;
  int i,j,sz;
  char str[200][80];                    /*接收从文件中读出字符的数组*/
  char *tmpstr;                         /*临时字符串的地址指针*/
  if(! (fp=fopen("fstr1.txt","r+")))
   { printf("File cannot be opened : fstr1.txt\n");
     exit(1);
   }
  sz=0;
  while(! (feof(fp)))
   { fscanf(fp,"%10s",str[sz]);         /*读字符串到str[]中*/
     buffer[sz]=str[sz++];              /*再写到buffer中*/
   }
  for (i=0;i<sz-1;i++)
   for (j=i+1;j<sz;j++)                 /*进行排序*/
    if(strcmp(buffer[i],buffer[j])>0)
     { tmpstr=huffer[i];
       buffer[i]=huffer[j];
       buffer[j]=tmpstr;
     }
  sz--;
  rewind(fp);                           /*移文件指针到文件头*/
  for (i=0;(i<=sz);i++)
   fprintf(fp,"%10s",buffer[i]);
   fclose(fp);
  }
```

建立一个待排序的文件

```
#include<stdio.h>
FILE *fp;
char a[][10]={"London","Paris","Bon","Rome","Tokyo", "Detroit","Moscow",
     "Jerusalem","Bombey","Bejing"};
main()
  {
    int i;
    fp=fopen("fstrl.txt","w");
    for(i=0;i<=9;i++)
```

```
        fprintf(fp,"%10s",a[i]);
      fclose(fp);
    }
```

fseek()函数用于把文件指针以 origin 为起点移动 offset 个字节，其中 origin 指出的位置可有如下表几种选择：

Origin	数值	代表的具体位置
SEEK_SET	0	文件开头
SEEK_CUR	1	文件指针当前位置
SEEK_END	2	文件尾

例如：

fseek(fp,10L,SEEK_SET);

把文件指针从文件开头移到第 10 个字节处，由于 offset 参数要求是长整型数，故其数后带 L。

fseek(fp,-15L,SEEK_END);

把文件指针从文件尾向前移动 15 个字节。

fseek()函数调用成功，返回一个 0，否则返回一个非零值。

利用 fseek()函数随意移动文件指针后，可以使用前面讲过的文件的读取函数，进行顺序读写，但顺序读写的开始位置，可以通过 fseek()来改变，而不一定从头开始。但对于以追加方式打开的文件是个例外，因进行追加操作时，总是将文件指针指到文件尾，而不管对文件指针进行了如何移动。

下面的程序使用了 fseek()函数进行读：

```
#include<stdio.h>
main()
 { FILE *fp;
   if((fp=fopen("fnum.dat","rb"))==NULL)   /*打开二进制文件读*/
    { printf("File cannot be opened : fnum.dat\n");
      exit(1);
    }
   fseek(fp,8L,SEEK_SET);                  /*文件指针定位到第8个字节*/
   fgetc(fp);                              /*读这个字符*/
   fclose(fp);
}
```

这个程序用 fseek()函数，将文件指针移到第 8 个字节处，接着用顺序读字符函数 fgetc()将其读出。

② 文件随机读写函数

```
int fread(void *ptr,int size,int nitems,FILE *stream);
int fwrite(void *ptr,int size,int nitems,FILE *stream);
```

fread()函数从流指针指定的文件中读取 nitems 个数据项，每个数据项的长度为 size 个字节，读取的 nitems 数据项存入由 ptr 指针指向的内存缓冲区中，在执行 fread()函数时，文件指针随着读取的字节数而向后移动，最后移动结束的位置等于实际读出的字节数。该函数执行结束后，将返回实际读出的数据项数，这个数据项数不一定等于设置的 nitems，因为若文件中没有足够多的数据项，或读中间出错，都会导致返回的数据项数少于设置的 nitems。当返回数不等于 nitems 时，可以用 feof()或 ferror()函数进行检查，看是到文件尾还是读出错。

fwrite()函数从 ptr 指向的缓冲区中取出长度为 size 字节的 nitems 个数据项，写入到流指针 stream 指向的文件中，执行该操作后，文件指针将向后移动，移动的字节数等于写入文件的字节数目。该函数操作完成后，也将返回写入的数据项数。当返回值比设置参数 nitimes 小时，可能是 ptr 指向的缓冲区数据不够，或写出错。

例

下列程序使用 fread()函数将 PC 机硬盘第一个扇区的内容由缓冲区(它由函数 biosdisk()从第一扇区读入)读到 B 盘的 part.dat 文件中进行保护，也可由 fwrit()函数将其再写入到缓冲区 buffer 中，再由 biosdisk()函数恢复到硬盘的第一扇区中。

硬盘第一扇区的内容是硬盘分区表(Hard disk parttion table)。要了解分区表，必须要知道硬盘分区是怎么回事，现作以下说明：

由于目前 PC 机上配备的硬盘容量很大，因而为了使用方便，可以安装多种操作系统，再者 DOS 3.3 以前版本管理硬盘的容量有限，不能全部管理大容量硬盘。而对硬盘采用分区的办法(目前人们在使用高版本 DOS 时仍多采用分区)，就好像将其又分成了几个独立的区域，每个区域仿佛是一个独立的硬盘一样，DOS 在访问每个分区的扇区时，使用各分区相对的逻辑扇区号，就像是一个独立的硬盘一样。由于有了分区，因而硬盘必须要有一个硬盘分区表，在该表中说明了每个分区从哪里开始、它的长度、是否为 DOS 分区，如果是，则文件分配表(FAT)是什么类型，以及是否为活动分区等(活动分区指可以直接启动引导系统的分区，硬盘中只有一个)。

分区表位于硬盘的 0 面 0 道 1 扇区，该扇区的内容如图 6.3 所示。

000H	主引导记录(240 字节)
0F0H	全 0(206 字节)
1BEH 1CEH	第一分区表(16 个字节)
1DEH	第二分区表(16 个字节)
1EEH	第三分区表(16 个字节)
	第四分区表(16 个字节)
	55H AAH

图 6.3 硬盘的 0 面 0 道 1 扇区的内容

分区表从第 446(1BEH)个字节处开始，共 64 个字节长，最后两个字节为 55H，AAH，每个分区的表为 16 个字节长，这 4 个 16 个字节长的块中各字节含义都是相同的，如表 6.2 所

列：

表 6.2 分区表各字节含义

字节	含义
0	引导标志，80H 表示活动分区，0 表示其它分区，每次只能有一个活动分区
1	分区开始的面
2	低 6 位是分区开始的扇区，头两位是分区开始的磁道的头两位。这是 BIOS 调用时使用的格式
3	分区开始的磁道的低 8 位
4	系统标志，若为 4，则说明该分区为 DOS 分区，使用的是 16 位的 FAT，若为 1，则说明该分区为使用 12 位的 FAT 的 DOS 分区，其它分区为 0
5	分区结束标志
6	低 6 位为分区结束的扇区，头两位为结束磁道号的头两位
7	分区结束磁道号的低 8 位
8～11	该双字包含了分区前的扇区数，低位字节在前
12～15	该双字包含了分区的扇区数，低位字节在前

分区的内容由 FDISK 在进行磁盘分区时设置的，当进行 FORMAT 时，有时会改变它。

硬盘分区表完整与否，直接关系到机器能否正常运行，有些计算机病毒经常会破坏分区表，而导致系统瘫痪，因此我们可以采取预防措施，预先将硬盘分区表（即第一扇区的内容）存到软盘上。当硬盘分区表出现故障时，可从软盘将其调入内存（可开辟一个 512 字节的缓冲区），然后从缓冲区用 biosdisk()函数将其写回到硬盘的第一扇区去），从而恢复被破坏的分区表。

上面的程序可实现上面所述的存分区表到软盘和将软盘所存分区表（左文件 part.dat）再恢复到硬盘第一扇区去的功能。

该程序中使用的对磁盘进行操作的函数说明如下：

```
char biosdisk(int cmd,int drive,int head,int track, int sector,int nsects,void *
        buffer);
```

该函数执行时，将调用中断 0x13，对磁盘的各种操作将直接由 BIOS 来完成。该函数中的参数 cmd 表示要对磁盘执行的操作，具体的主要内容如表 6.3 所示，第二个参数 drive 表示磁盘驱动器号：0 代表第一个软驱，1 表示第二个软驱，2 表示第三个软驱等。当 drive 的值为 0x80 时，表示第一个硬驱，为 0x81 时表示第二个硬区，0x82 时，为第三个硬驱等，head 表示磁头号，track 表示磁道号，sector 表示扇区号，nsects 表示扇区数，buffer 为缓冲区指针。当该函数操作成功后返回一个数，0 表示成功，其它数表示操作失败，关于失败的具体原因，可查阅 Turbo C 参考手册。

表 6.3 cmd 主要取值及其含义

cmd	含义
0	复位磁盘系统，即强制硬复位（此时其它参数没有）
1	返回最后一次磁盘操作状态（此时其它参数不用）
2	读一个或多个扇区内容到缓冲区 buffer
3	将缓冲区 buffer 中的内容写入磁盘的一个或多个扇区中去
4	验证一个或多个扇区
5	格式化一个磁道

关于cmd的其它值可参阅Turbo C参考手册。

下面是该函数调用例：

result=biosdisk(2,0x80,0,0,1,1,buffer);

它将把硬盘C中0号磁头0磁道第一扇区共一个扇区内容读到buffer中去。

设下面的程序进行编译连接后生成名为SAVEPART.EXE的执行文件，当要将磁盘第一扇区内容写到B盘上进行保护时，可在DOS下输入如下命令行：

C>SAVEPART B ↙

它将把第一扇区内容存到B盘上的part.dat文件中，当键入的命令行为：

C>SAVEPART C ↙

它将从B盘上将第一扇区内容写到硬盘第一扇区去。

```
# include <stdio.h>
# include <bios.h>
# include <fcntl.h>
# include <sys\types.h>
# include <sys\stat.h>
void helpmsg(void);
main(int argc,char *argv[])
{
  FILE *fp;
  int result;
  char buffer[512];
  if(argc==1) helpmsg();
  if(*argv[1]=='b'||*argv[1]=='B')
   {
     result=biosdisk(2,0x80,0,0,1,1,buffer);
     if(! result)
      {
      printf("reding bard disk partition table ok! \n");
      if((fp=fopen("b:part.dat","wb+"))==NULL)
     { printf("File cannot be opened:b:part.dat\n");
       exit(1);
     }
     fwrite(buffer,1,512,fp);
  fclose(fp);
  printf(" saveing hard disk partition table to B:. ok ! \n");
  return 0;
}
else
 { fprintf(stderr,"reading partition table failure");
   exit(1);
 }
}
```

```
if( *argv[1]=='c'|| *argv[1]=='C')
 {
  if((fp=fopen("b:part.dat","rb+"))==NULL)
   {
    fprintf(stderr,"File cannot be opend");
    exit(1);
   }
  fread(buffer,1,512,fp);
  result=biosdisk(3,0x80,0,0,1,1,buffer);
  if(! result)
  {
   printf("Hard disk partition table is restored");
   fclose(fp);
   return 0;
   }
  else{
   fprintf(stderr,"restoretion of hard disk partition table failed");
   fclose(fp);
   exit(1);
   ;
  }
  return 0;
  }
  void belpmsg(void)
  {
   puts("program format is SAVEPART B or SAVEPART C");
   puts("B------save bard disk partition table to driver B");
   puts("C------restore hard disk partition table to driver C");
   exit(0);
   }
```

6.4 非标准的文件输入输出操作

前面讲述的标准文件输入输出操作是指这些操作函数符合C语言的普通标准(即ANSI标准),因而各种机器上使用的C语言中这些函数具有兼容性,这样便于移植,且用户编程使用也简单方便,这是因为编程者对这种文件操作介入很少,因而操作显得比较简单,这是ANSI C标准推荐的,故称标准的。

非标准文件输入输出操作是指这些操作函数不是ANSI C标准,它们最早用于UNIX操作系统。对PC机来说,在MS-DOS中提供了对数据文件的读写功能,而非标准文件输入输出函数实际上就是调用了DOS的这些文件读写功能,因而用这些函数对文件进行读写更直接,速度更快,它们特别适合于编写系统软件和数据采集、自动控制等要求速度快的应用程序。由于其兼容性差,因而移植不方便。由于这类文件操作不具有自动缓冲功能,因而文

件缓冲区的设置要编程者自己设计。再者，由于该类函数中没有格式化输入输出函数，因而也带来使用上的不方便，由于它更接近于操作系统，因而人们称这类文件操作为低级文件输入输出操作，或非标准文件输入输出操作，也有称系统级输入输出操作的。

非标准文件读写函数在 Turbo C 中提供了四种，它们是：

```
int _read(int handle,void *block,int n);
int read(int handle,void *block, int n);
int _write(int handle,void *block,int n);
int write(int handle,void *block,int n);
```

我们将分别予以介绍。

上述函数中都有一个参数 handle，它代表了要进行操作的文件，有人称之为文件句柄。该参数代替了在标准文件操作函数中的流指针(FILE *stream)，当我们对一个文件用非标准输入输出函数操作时，也必须用 open()函数打开它，它将返回代表这个文件的一个代号(handle)，以后对该文件的操作，均用该代号进行，文件名将不再出现。如果对一个文件写时，而这个文件不存在，则必须首先用函数 creat()建立这个文件，它也返回建立的这个文件的代号，因而可用 write()函数对它写。对文件操作结束时，必须用 close()函数对它进行关闭，以便收回文件代号。

6.4.1 建立一个新文件或重写一个已有文件的函数 creat()，_creat()，creatnew()

当要对一个文件进行写时，若该文件不存在，或虽然存在，但要对其内容进行重写，这时可采用函数 creat()，它的定义形式是(原型在 io.h 中)：

```
int creat(char *filename,int permiss);
```

该函数将根据 filename 指定的文件名建立一个文件，若该文件已存在，则表示要进行重写，其中参数 permiss 可以取下列值：

permiss	含义
S_IWRITE	允许写
S_IREAD	允许读
S_IRER | S_IWRITE	允许读写

若文件已存在，并已设置属性为只写，则文件属性将不会改变。若文件属性为只读，则 creat()返回一个错误代码。当建立成功，该函数将返回一个文件代号，以后就可用该文件代号对其进行读写了。建立的文件是文本方式还是二进制方式，则由全程变量 fmode 来指定，缺省时为文本方式。

另一个创建文件的函数是：

```
creat(char *filename,int attrib);
```

该函数将按照设置的文件属性(attrib)来建立一个由 fmode 指定形式的文件，名为 filename 的文件，参数 attrih 可以是下列常量之一(它们在 dos.h 中被定义)：

attrib	代码	含义
FA_RDONLY	1	只读文件
FA_HZDDEN	2	隐含文件
FA_SYSTEM	4	系统文件

这些属性也可以或起来，如 FA_RDONLY | FA_HZDDEN。

例如：

```
fh = _creat("Myfile.dat",3);
```

将建立一个名为 Myfile 的文件，该文件将成为只读的隐含文件。当该文件建立成功后，将返回一个该文件的代号。

还有一个建立文件的函数：

```
int creatnew (char *filename,int attrib);
```

其功能和参数含义同_creat()函数，不同的是若已有一个与 filename 指定文件同名的文件存在，则函数返回错误代码。而 creat()和_creat()则将把这个已存在的文件打开并清除其所含内容。

用文件打开函数也可以建立文件，如下所述。

6.4.2 非标准的文件打开函数

除了上述的 creat()，_creat()，creatnew()等函数将建立一个新文件外，若要对已存在文件进行读写，也必须先打开它，当正确打开并得到文件代号后，便可对其进行读写操作。Turbo C 中定义了 open()和_open()函数来打开文件，它也可用来建立新文件。

打开文件函数的说明如下：

```
int open(char *pathname, int access[,int permiss]);
int _open(char *pathname,int access);
```

这两个函数将打开由 pathname 指定的文件，patbname 既可以是简单的文件名，也可以是文件的路径名。参数 access 表示了打开文件的存取代码，各代码符号与其含义如表 6.4 所列：

表 6.4 文件存取代码符号及含义

access(代码符号)	含义
O_RDONLY	打开文件用于只读
O_WRONLY	打开文件用于只写
O_RDWR	打开文件用于读写
O_APPEND	打开文件用于追加
O_CREAT	若文件不存在，则建立，否则无效
O_TRUNC	将存在的文件长度裁为 0
O_EXCL	和 O_CREAT 一起使用，若文件已存在，则返回错误信息
O_BINARY	以二进制方式打开文件
O_TEXT	以文本方式打开文件

这些代码符号还可或起来作为 access 参数，例如：fh=open("Myfile.txt",O_RDONLY | O_TEXT)将把 Myfile.txt 文件打开，用文本方式对其进行读。此时文件指针将指向文件头。若打开成功，则返回该文件的代号。若在参数中没有给出是二进制方式还是文本方式，则按_fmode 设置的方式。

在 open()函数中的可选参数 permiss 用在当 access 参数为 O_CREAT 时，它可以取下列值：

permiss	含义
S_IWRITE	允许写
S_IREAD	允许读
S_IREAD | IWRITE	允许读写

实际上它就是 creat()函数中的 permiss 参数。

例如：

fh=open("Myfile1.txt",O_RDONLY | O_CREAT,S_IREAD),

它将以只读方式打开 Myfile1.txt 文件，若该文件不存在，则建立该文件(用于读)，操作成功，则返回该文件的代号给 fn。

_open()函数不允许使用除 O_RDONLY,O_WRONLY 和 O_RDWR 以外的 access 参数，对于 MS_DOS 3.x 的版本，access 还可使用另外一些参数(O_DENYxxx)，关于它们的说明可参阅有关手册。由于它们不常用，因而不作介绍。

上述两个文件打开函数，若打开成功，均返回打开文件的代号(整数)，并将文件指针指向文件头。对于 open()函数，若打开是为了追加(O_APPEND)，则将文件指针指向文件尾。当打开失败时，则返回错误代码－1，并将全局变量 errno 置为下列值之一：

errno 值	含义
ENOENT	路径名或文件名没有找到
EMFILE	打开的文件太多
EACCES	无此存取权限
EINVACC	无效的存取代码

我们可以读取 errno 的值，以了解打开失败的原因，例如：

```
if((fh=open("Myfile",O_RDONLY))==-1)
  {
   switch(errno)
     {
      case ENOENT:
              printf("\n path or filename not found");
              break;
      case EMFILE:
              printf("\n Too many open files");
              break;
      case EACCES:
              printf("\n permission denied");
              break;
```

```
        case EINVACC;
                printf("\n Invalid access code");
    }
}
```

6.4.3 关于系统标准输入输出设备的文件代号

在标准文件输入输出操作时，Turbo C 自动打开了 5 个设备文件，并给它们设置了相应的流指针(如标准文件的打开函数中所述)。在非标准文件输入输出操作时，同样自动地打开了这 5 个文件，并给了它们不同的文件代号。因而用户无需再用函数来打开它们。下面是各设备对应的文件代号：

设备名	DOS 赋予的文件名	文件代号
标准输入(指键盘)	CON	0
标准输出(指显示器)	CON	1
标准错误(指显示器)	CON	2
标准辅助(指异步串行口 1)	AUX	3
标准打印(指打印机)	PRN	4

由于实际需要，这些标准设备的输入输出可以重定向。例如为了串行通讯的需要，可以将标准打印重定向到串行口，将标准输入输出重定向到磁盘文件上。

6.4.4 非标准文件的关闭函数 close()

虽然程序结束后，打开的文件自动关闭，但一个程序中能打开文件的数目是有限的，因而用完一个文件后，必须进行关闭，以使退出来的文件代号能被别的文件使用，更主要的是使磁盘缓冲区的内容写到文件中去。非标准文件关闭函数定义如下：

```
int close(int handle);
int _close(int handle);
```

它们都将关闭由文件代号(handle)代表的文件。要注意的是，这两个函数执行后，并不会给文件尾添一个文件结束符号 ctrl-Z 字符，以表示文件结束。故当文件写结束时，用户自己要加上这个符号。

若文件正确关闭，则两个函数返回 0，否则返回 −1。其中错误信息可以通过读 errno 变量的值可知。如 handle 是一个不正确的文件代号，则 errno 的值为 EBADF，即非法的文件号。

6.4.5 非标准文件的读写函数

Turbo C 提供了两对读写文件的函数，它们的定义均在 io.h 中，它们是：

```
int read(int handle,void  * buf,int nhyte);
int _read(int handle,void  * buf,int nbyte);
int write (int handle,void * huf,int nbyte);
int _write(int handle,void  * huf,int nbyte);
```

下面分别进行介绍：

① 读文件函数

有两个读文件函数：

int read(int handle,void *buf,int nbyte);

int _read(int handle,void *buf,int nbyte);

这两个函数将从由文件代号 handle 代表的文件中，读出 nbyte 个字节到 buf 指出的缓冲区中去。若成功，函数将返回读出的字节数。对于文本文件，若读到文件结束标志 ctrl-Z 时，返回的字节数将不包含回车符和 ctrl-Z 字符。因而当从文件尾开始读时，返回值为 0。不过这两个函数开始读时，文件指针总是指向文件开头位置，即总是从文件头开始读起，除非用函数 lseek()移动文件指针，使开始读的位置不从文件头开始。

当读出错时，两个函数返回－1，根据 errno 变量的值，可以得到出错原因，如：errno 为

EACCES　　无此存取权限

EBADF　　无效的文件代号

两个函数中设置的读出的字节数 nbyte 最大不能超过 65534，因 65535 的值(0xFFFF)同返回错误标志－1 相同。

对于缓冲区 buf 的设置，可以定义一个数组，可以动态分配一个缓冲区(用 malloc())，也可以静态定义缓冲区变量，例如：

```
char buff[4096];
    ⋮
read(fh,buff,4096);
    ⋮
```

又如：

```
float x;
 ⋮
read(fh,&x,4);
    ⋮
```

动态分配一个缓冲区：

```
buffer=(char *) malloc(16);
 ⋮
read(fh,hufler,16);
 ⋮
```

read()函数和_read()函数的差别是，后者读文本文件时，不滤除回车符，读到文件尾时，不报告文件结束信息。

② 写文件函数

有两个写文件函数：

int write(int hamdle,void *buf,int nbyte);

int _write(int handle,void *buf, int nbyte);

这两个函数把 buf 缓冲区中的 nbyte 个字节写到由文件代号 handle 指出的文件中去。若成功，函数将返回写入的字节数。对文本文件，write()返回的字节数中不包括回车符。当

出错时，这两个函数均返回－1。通过全局变量 errno，可知错误原因，如 errno 为下列值时，可知其错误原因：

EACCES　　无此权限

EBADF　　无此文件代号

write()和_write()的区别是，对文本文件，前者将把写入的回车符转换成回车换行符，在返回的写入字节数中，将不包含回车符，也不包含文件结束符(ctrl_Z)。当读到文件结束时，write()函数会给出文件结束信息，而_write()函数则不进行上面的转换和给出文件结束信息，因而_write()函数用于二进制文件。

当用 open()函数打开文件时，文件指针总是指向文件头，因而若接着进行读或写操作，总是从文件头开始。例外的情况是，当用 O_APPEND 选择项来打开文件，用 write()函数写时，文件指针在文件尾，因而写入的内容将追加在原文件尾后面。

下面的程序是一个文件复制程序，它如同 DOS 的 copy 命令一样，在该程序中复制的源文件必须存在。故而为读打开它，并返回文件代号 fhi。复制的目的文件由于原先不存在，故必须先建立并返回文件代号 fho。由 read()函数将 fhi 代表的文件读到 buffer 中，并返回读出的字节数 i_size，再由 write()函数将 buffer 中的内容写到 fh2 代表的文件中去，写的字节数也为 i_size。当写完后，返回写入的字节数 o_size。当 i_size＝o_size 时，表示完成复制。当在 DOS 下输入命令行，执行该程序时，命令行格式应是 FILECOPY 源文件名目的文件名，(设该程序执行文件名为 FILECOPY. EXE)，当命令行输入参数错时，则调用 belpmsg()函数，提示帮助信息，以正确输入。

```
# include <stdio. h>
# include <io. h>
# include <fcntl. h>
# include <sys\types. h>
# include <sys\stat. h>
# define COUNT 512
void helpmsg(void);
main(int argc,char *argv[])
{
  int fhi,fho;
  int i_size,o_size;
  char buffer[COUNT];
  if(argc! =3) helpmsg();
  if((fhi=open(argv[1],O_RDONLY))==EOF)
   {
     printf("Source file missing \n");
     exit(1);
   }
  if((fho=creat(argv[2],S_IWRITE))==EOF)
    { printf("Destination file cannot be created \n");
      close(fhi);
      exit(1);
```

```
    }
  do
   {
     i_size=read(fhi,buffer,COUNT);
     if(i_size>0)
      {
        o_size=write(fho,buffer,i_size);
        if(o_size! =i_size)
         {
           printf("file write error———stoped.\n\n");
           close(fhi);
           close(fho);
           exit(0);
           }
        }
    }while(i_size==COUNT);
   close(fhi);
   close(fho);
   printf("file copy complete.\n");
}
void helpmsg(void)
   {
     puts("command format is FILECOPY source_file destination_file");
     exit(0);
   }
```

6.4.6 删除文件函数 unlink()

若想从目录中删除一个文件，可使用 unlink()函数，该函数在 dos.h 中定义为：

```
int unlink (char *filename);
```

该函数将删掉由 filename 指定的文件，filename 可以是路径名或文件名，若文件是只读的，则用此函数不能删除。若删除成功，该函数返回 0，出错时返回－1，出错原因可从 errno 变量的值来了解。errno 值对应的错误原因如下：

ENOENT	路径或文件名没有找到
EACCES	无此权限

6.4.7 移动文件指针的函数 lseek()

上面所述的函数进行读写时，文件指针开始均指向文件头。而当用 O_APPEND 追加方式打开文件进行写时，文件指针指向文件尾，因而想进行随机读写是不可能的。若通过 lseek()函数移动文件指针到指定的位置，然后再用读写函数进行读写，则可实现随机读写，该函数定义(在 io.h)为：

```
long lseek(int handle,long offset, int fromwhere);
```

该函数把由 handle 指定文件的文件指针，移到由 fromwhere 所指位置再加上 offset 偏移量的地方，fromwhere 的值可以是 0,1,2，它们分别代表了三个符号常量的值(在 stdio.h 中定义)，即：

fromwhere	值	含义
SEEK_SET	0	文件开始位置
SEEK_CUR	1	当前文件指针位置
SEEK_END	2	文件尾

offset 是指移动的字节数。当 lseek()函数移动文件指针成功后，该函数将返回移动的 offset 值。若失败则返回－1，失败原因可从全局变量 erron 中的内容得知：

EBADF　　无效的文件代号

EINVAL　　无效的参数

若在移动文件指针时，不知道当前文件指针的位置，可用 tell()函数。它将返回当前文件指针的位置，即返回从文件开头到当前文件指针所在处的字符数。该函数在 io.h 中定义为：

Long tell(int handle);

该函数将返回 handle 指定文件的当前文件指针位置。若返回为－1，则表示出错，出错原因可从 erron 全局变量中得知：

EBADF　　无效的文件代号

下面的程序使用 lseek()函数。该程序定义的为了读而打开的文件中每 128 个字节为一扇区单位(sector)，当屏幕出现 buffer 字样时，用户可输入要读第几个扇区的扇区号，然后将该串由 atoi(s)函数转换成数字，从而由(long) sector * BUF_SIZE 得出从文件的多少个字节处开始读(此时由 lseek()函数将文件指针移到此处)，然后由 read 函数从该处开始读出 128 个字节内容到 buf 中去，然后再显示到屏幕上(printf(buf))。

设该程序的执行文件名为 SEEK.EXE，拟要显示内容的文件名为 filename，则要执行时命令行应写成：

SEEK filename ↙

下面是源程序：

```
#include<stdio.h>
#define BUF_SIZE 128

long int lseek();
main(int argc,char *argv[])
 {
   char buf[BUF_SIZE+1],s[10];
   int fh,sector;
   buf[BUF_SIZE+1]='\0';
   if((fh=open(argv[1],0))==-1)
    {
      printf("cannot open file\n");
      exit(0);
```

```
  }
 do
  {
   printf("buffer: ");
   gets(s);
   sector=atoi(s);
   if(lseek(fh,(long)sector * BUF_SIZE,0)==-1L)
    printf("seek error\n");
   if(read(fh,buf,BUF_SIZE)==0)
    {
     printf("sector out range\n");
    }
    else
     printf(buf);
   }while(sector>0);
   close(fh);
 }
```

下面的程序用标准设备打印机(prn)来打印文件,要打印的文件定义为只读文本文件,它的文件代号为 fh_1,并用文件代号 $fh_2=4$ 表示打印机,它是由 DOS 定义的标准文件 prn(即打印机)的代号。这样由 fh_1 代表的文件中读出 128 个字节的内容到 buf 中,然后由 buf 将其内容再输出到打印机上,如此反复(用 while(! eof(fh1))控制),直到文件尾为止。若执行该程序的命令行中还有要打印的第二、三个文件,则该程序也将一一打印输出,它是用循环语句:

```
while(i<argc)
 { ⋮
  i++;
 }
```

来实现的。

若执行该程序的命令行中第一个参数不是 prn,则规定 $fb_2=1$,即在屏幕显示。

设该程序的执行程序为 writeprn.exe,则命令行应写为:

writeprn prn filename1 filename2…其中 filename1,filename2…为分别要打印的文件。当 prn 写错时,将在显示器上一一显示各文件内容而不打印。

```
#include<io.h>
#include<fcntl.h>
#include<string.h>
#define BUF  SIZE 128

main(int argc,char *argv[])
 {
  char buf[BUF_SIZE];
  int fh1,fh2,i,bytes;
```

```
if(argc<2)
 {
   printf("\nInvalid of arguments");
   printf("\nCommand format : flist filename filename...");
   exit(0);
 }
i=1;
if(strcmp(argv[1],"prn")==0)
 {
   fh2=4;
   i++;
 }
else
 fh2=1;
while(i<argc)
    {
      if((fh1=open(argv[i],O_RDONLY | O_TEXT))<0)
        {
          printf("cannot open file\n",argv[i]);
        }
      else{
       while(! eof(fh1))
       {
         bytes=read(fh1,buf,BUF_SIZE);
         write(fh2,buf,bytes);
       }
      close(fh1);
      }
      i++;
 }
}
```

第 7 章

I/O 接口的输入输出

由于PC机软硬件资源丰富，它的使用已遍及各行各业。当使用PC机对物理、化学等过程进行数据采集与数据处理、进行实时控制以及对一个现场用PC机控制或管理时，均需要将数据输入到计算机中，又要将计算机处理的结果输出出来，或作为控制量，或作为图形与数据打印显示，称之为数据输出。这些为计算机输入数据或接收计算机输出数据的设备又叫作输入输出设备，或称为I/O设备，也有称为外围设备的。由于这些设备向计算机输入数据或接收计算机输出数据的速度和计算机不匹配，甚至两者数据格式可能不一样，电路工作的逻辑时序也可能不一样，因此必须在计算机与I/O设备间有一个媒介，对上述的矛盾进行协调，这就是要制作I/O接口电路，即用该电路将双方连接起来。为此PC机提供了几个扩充插槽，它们安装在系统板上，这些插槽的相应编号的接脚含义均有统一的规定，其中62针的插槽称为XT总线，还有一36针插槽是后来增加的，它们在286、386、486、586等微机上出现，这62针插槽和36针插槽合起来称为AT总线，AT总线又称为ISA总线(工业标准结构总线)，如图7.1所示。表7.1、7.2分别为36线和62线的各引脚信号。这些插槽中可插入各种接口电路卡，由于各插槽相同编号的引脚都有相同的含义，因而接口电路卡可插在任何一个AT总线槽内，引脚编号是从装焊元件面的靠近机箱后部的引脚开始，为A1，A2，…A31，背面为B1，B2，…B31。对应的36针为C1，C2，…C18，背面为D1，D2，…D18。另外PC系统板上还有EISA总线或VESA总线，由于C不支持，故不作介绍。

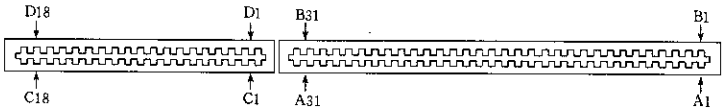


图 7.1　PC与AT总线

由于不同槽上插有不同功能的接口电路卡，为了区别它们，制作者赋与它们一接口地址，这样计算机就可如同访问存储单元一样按地址访问这些接口电路卡。接口卡上也许还有数据寄存器、状态寄存器及命令寄存器等。为了区别它们，制作者也给以它们不同的地址，以便计算机能正确地找到它们，对其进行操作。接口卡上也许有不同的电路，要区别它们，启动需要的电路工作，也可给它们赋以不同的地址。为了将这些地址和存储器地址相区别，我们称它为接口地址。按接口地址去寻址，也就是接口地址译码问题。I/O接口的寻址方式一般有两种，下面分别予以介绍：

表 7.1　36 线各引脚信号

信号名称	输入/输出	引脚号		输入/输出	信号名称
−MEM CS16	O	D1	C1	O	SBHE
−I/O CS16	O	D2	C2	I/O	LA23
IRQ 10	O	D3	C3	I/O	LA22
IRQ 11	O	D4	C4	I/O	LA21
IRQ 12	O	D5	C5	I/O	LA20
IRQ 15	O	D6	C6	I/O	LA19
IRQ 14	O	D7	C7	I/O	LA18
−DACK0	I	D8	C8	I/O	LA17
DRQ0	O	D9	C9	O	−MEMR
−DACK5	I	D10	C10	O	−MEMW
DRQ5	O	D11	C11	I/O	SD08
−DACK6	I	D12	C12	I/O	SD09
DRQ6	O	D13	C13	I/O	SD10
− DACK7	I	D14	C14	I/O	SD11
DRQ7	O	D15	C15	I/O	SD12
+5 Volts		D16	C16	I/O	SD13
−MASTER	O	D17	C17	I/O	SD14
Ground		D18	C18	I/O	SD15

表 7.2　62 线各引脚信号

信号名称	输入/输出	引脚号		输入/输出	信号名称
Ground	I	B1	A1	I	−I/O CH CK
Reset	I	B2	A2	I/O	SD7
+5 Volts	I	B3	A3	I/O	SD6
IRQ2	O	B4	A4	I	SD5
−5 Volts	I	B5	A5	I/O	SD4
DRQ2	O	B6	A6	I	SD3
−12 Volts	I	B7	A7	I/O	SD2
−OWS	I/O	B8	A8	I/O	SD1
+12 Volts	I	B9	A9	I/O	SD0
GND Ground	I	B10	A10	I	I/O CH RDY
−SMEMW	I	B11	A11	I/O	AEN
−SMEMR	I	B12	A12	I/O	SA19

续表

信号名称	输入/输出	引脚号		输入/输出	信号名称
−IOW	I/O	B13	A13	I/O	SA18
−IOR	I/O	B14	A14	I/O	SA17
−DACK3	I/O	B15	A15	I/O	SA16
DRQ3	I/O	B16	A16	I/O	SA15
−DACK1	I/O	B17	A17	I/O	SA14
DRQ1	I/O	B18	A18	I/O	SA13
−DACK0	I/O	B19	A19	I/O	SA12
CLK	I/O	B20	A20	I/O	SA11
IRQ7	I/O	B21	A21	I/O	SA10
IRQ6	I/O	B22	A22	I/O	SA9
IRQ5	I/O	B23	A23	I/O	SA8
IRQ4	I/O	B24	A24	I/O	SA7
IRQ3	I/O	B25	A25	I/O	SA6
−DACK2	I/O	B26	A26	I/O	SA5
T/C	I	B27	A27	I/O	SA4
BALE	I	B28	A28	I/O	SA3
+5 Volts	I	B29	A29	I/O	SA2
OSC	I	B30	A30	I/O	SA1
Ground	I	B31	A31	I/O	SA0

7.1 I/O 接口的寻址方式

I/O 接口的寻址方式一般有两种：一种是将接口地址和存储器地址统一编址；一种是 I/O 接口地址和存储器地址分别独立编址。

接口和存储器统一编址，是将 I/O 接口当作存储单元一样，赋给它地址，这些地址是存储器地址空间的一部分。因此在其指令系统中，没有专门的 I/O 指令，共用访问存储器的指令，如 APPLE 机中的 6502 微处理器等就属此类。

PC 机中的 80x86 CPU 采用 I/O 独立编址方式，采用专门的 I/O 指令来对接口地址进行操作。这样，存储器地址和 I/O 接口地址可以重叠。然而由于两者需采用不同的指令进行读写操作，所以不会由于地址相同而混淆。这种编址方式的优点是不占用存储器地址，因而不会减少存储器容量。由于有专门的 IN 和 OUT 指令，因此比用存储器读写指令执行速度快。但这种方式的缺点是在硬件电路上要对这两种存取进行区别。另外，专门的 I/O 指令功能简单，要完成某些操作还须和某些指令配合。

PC 机中仅使用 A_9～A_0 地址位来表示 I/O 口地址，即可有 1024 个口地址。前 512 个供

系统电路板使用，后512个供扩充插槽使用。即当$A_9=0$时表示为系统板上I/O口地址；$A_9=1$时，表示为扩充插槽接口卡上的口地址。因此用户要制作接口电路卡时，其口地址要保证$A_9=1$。

在1024个口地址中，有许多已被IBM或其它厂商制作的各种与主机配套的接口卡占用，有些保留有待今后继续开发。PC口地址分配图如表7.3和7.4所示。

表7.3 系统板I/O口地址使用情况

口地址范围(16进制)	用途	口地址范围(16进制)	用途
0000—001F	8237A SDMA控制器1	00A0—00BF	8259A中断控制器2
0020—003F	8559A中断控制器1	00C0—00DF	8237A 5 DMA控制器2
0040—005F	8253/8254定时/计数器	00F0	清除协处理器总线
0060—006F	8042键盘控制器(AT)	00F1	设置协处理器总线
0070—007F	CMOS RAM与NMI屏蔽寄存器(AT)	00F8—0FF	协处理器
0080—009F	DMA页寄存器	01F0—01F8	硬盘

表7.4 扩充插槽I/O口地址使用情况

口地址范围(16进制)	用途	口地址范围(16进制)	用途
200—207	游戏卡I/O口	378—37F	并行打印机1
210—217	扩展部件(仅XT用)	380—38F	SDLC通信及同步通信1
220—24F	保留	3A0—3AF	同步通信2
278—27F	并行口打印机2	3B0—3BF	MDA单色显示器
2F0—2F7	保留	3C0—3CF	保留
2F8—2FF	串行口2	3D0—3DF	彩色图形适配器
300—31F	试验卡	3F0—3F7	软盘适配器
320—32F	硬盘适配器	3F8—3FF	串行口1
360—36F	保留		

200～03FF地址范围为扩充插槽使用的口地址，因此接口卡口地址一般局限在此范围内，已被一些专用接口卡占用的口地址不能再使用，除非这些卡你暂时不使用，而将其拿掉，或将来不用，总之接口卡使用时，不能同时有和别的卡相同的口地址。

当执行I/O指令时，只能对选中的口地址进行读写操作。具有唯一口地址的I/O接口，如何知道计算机选中了自己，要和自己进行数据交换呢？这就是口地址识别问题，或称为口地址译码。当对送来的地址码进行译码，得出就是自己的口地址时，该接口电路便按设计的逻辑动作，完成预定的功能。因此任何一个接口电路卡的设计，都必须要有口地址译码部分。

7.2 I/O接口的输入输出函数

Turbo C提供了专门对I/O接口进行输入输出操作的几个函数，它们是：

```
int inp(int portid);
int inport(int portid);
int inportb(int portid);
void outp(int portid,int value);
void outport(int portid,int value);
void outportb(int portid,int value);
```

这些函数的原型在 dos.h 中。其中,inp 是等价于 inport 的一个宏,outp 是等价于 outport 的一个宏,因而我们仅需了解其它几个函数就可以了。

7.2.1 接口输入函数

```
int inport(int portid)
int inportb(int portid)
```

inport 函数从指定的接口地址 portid 中读入一个字(即 16 位二进制数),而 inportb 则从指定的接口地址 portid 中读入一个字节(8 位二进制数)。如 I/O 接口的寻址方式中所述,当执行这两个函数后,它们均返回各自从接口地址所对应的输入设备中得到的 16 位或 8 位二进制数。由于 PC 机数据总线是 8 位的,故 inportb()比较常用,而 inport(portid)实际上执行了两次 inportb(),即 inportb(portid)和 inportb(portid+1)。例如:

```
unsigned char p;
p=inportb(0x2F0);
```

它将从 2F0H 接口地址中得到一个字节的数,并赋给无符号变量 P。

```
unsigned int c;
c=inport(0x2F0);
```

它将从 2F0H 和 2F1H 接口地址中分别得到一个字节的数,然后组合成一个 16 位的二进制数赋给变量 C,其中 2F0H 接口中得到的数为低字节,2F1H 接口中得到的数为高字节。

7.2.2 接口输出函数

```
int outport(int portid, int value);
int outportb(int portid unsigned char value);
```

outport 函数把一个 16 位二进制数 value 发送到口地址为 portid 的接口中去。对于 PC 机,实际上是将低字节数送到口地址为 portid 接口中去,将高字节送到口地址为 portid+1 的接口中去。

outporth 函数是将一个字节的数 value 送到口地址为 portid 的接口中去。例如:

```
outportb(0x2F0,385);
```

它将把整数 385 送往口地址为 2F0H 的接口中去。

```
outport(0x2F2,4095);
```

它将把 4095 送到口地址为 2F2H 和 2F3H 的接口中去,(化成二进制数的低 8 位送 2F2H,高 8 位送 2F3H)。

7.3 应 用 例

7.3.1 发声程序

在 PC 机的系统板上装有定时与计数器 8253 芯片，还有 8255 可编程并行接口芯片，由它们组成的硬件电路可用来产生 PC 机内扬声器的声音，对于 286、386、486、586 等 PC 微机，由于采用了超大规模集成电路，因而看不到这些芯片，它们均集成在外围电路芯片上了。

当我们操作计算机时，常常听到的发声，就是由软件控制这些电路而产生的，声音的长短，音调的高低，均可由程序进行控制。下面就是一个控制扬声器发音程序。系统板上控制发音的电路可用图 7.2 表示，其中定时器电路产生一个方波信号，其频率可以编程设定，它决定了扬声器发音的频率，当对定时器电路进行频率设定时，首先要对其命令寄存器(口地址为 0x43)写命令字，如写入 0x66，则表示选择该定时器的第二个通道，计数频率先送低 8 位(二进制)，后送高 8 位，这是用 outportb(0x43,0x66)；来实现的。接着用口地址 0x42 送频率计数值，先送低 8 位，后送高 8 位，即用 outportb(0x42,count.c[0])和 outportb(0x42),count.c[1])来实现。该计数值减为一半时，定时器产生低电平，减为 0 时，又变高电平并又重新开始减计数(仍从最初设置的计数值开始)，因而产生一系列方波信号，此信号是否能推动扬声器发音，还要看由 8255 产生的门控信号和送数信号是否为 1。它们也可编程。其口地址为 0x61，当 8255 的 PB 口的 0 位和 1 位为 1 时，就表示允许发声。为 0 时，便禁止发声。因而用 outportb(0x61,bits | 3)来允许发声，而用 outportb(0x61,bits&0xfc)来禁止发声，且不改变 8255 其它位原来的值，关于这方面的详细内容可以参阅本人写的 IBM/PC XT 接口技术及其应用一书有关内容。由于方波信号可看作由许多频率的正弦信号合成(按富里叶信号分析)，对于高次谐波产生的声音，扬声器无法响应，故实际上产生我们可听见的声音的波形可用基频的正弦波表示。如图 7.3 所示。

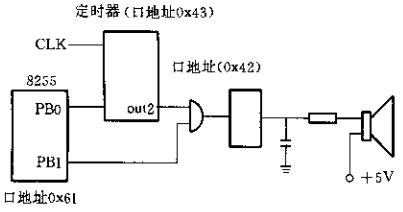


图 7.2 扬声器电路

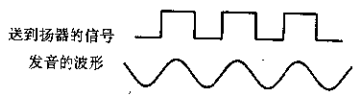


图 7.3 扬声器信号

程序中由 rand()函数产生一个 0～32767 之间的伪随机整数，为使声音柔和，在 freq<=5000 时，退出循环而调用 sound 发声函数，该函数由传来的参数 freq 来设定频率计数值 count .divisor 为 119328/freq(占用两个地址单元)，由于采用了联合结构，共用一个存储区，故联合结构成员 count.c[0]和 count.c[1]取该数的低 8 位和高 8 位(二进制位)，用于初始化定时器的频率计数值。

运行该程序后，将会听到美妙的声音，当要停止时，可按任意键而由 kbhit()函数控制

do 循环结束来实现。

由于 PC 机主频不同，发声效果将不同，选取 freq＞1000，将会在 386 PC 机上(主频为 33M)发出清脆的声音，当主频小于 33M 时，可适当增加此数。

```
void sound(unsigned int freq);
main()
 {
    unsigned int freq;
    do {
       do{
          freq=rand();
          }while(freq>1000);
          sound(freq);
        }while(! kbhit());
    }
 void sound(unsigned int freq)
 {
   unsigned i;
   union {
           long divisor;
           unsigned char c[2];
         }count;
   unsigned char bits;
   count.divisor=119328/freq;
   outportb(0x43,0xb6);
   outportb(0x42,count.c[0]);
   outportb(0x42,count.c[1]);
   bits=inportb(0x61);
   outportb(0x61,bits | 3);
   for(i=0;i<20000;i++);
   outportb(0x61,hits&0xfc);
   }
```

7.3.2 信号采集程序

这是一个信号采集程序，它由作者研制的 NKAD12-2 12 位模数(AD)转换卡将外界的信号采集到 PC 286、386、486 微机内存中去，从而对信号进行数字处理，提取有用信息由计算机处理。该电路原理如图 7.4 所示。

该电路板可插于 PC 机的任一插槽中，可外接 16 路信号，但每次仅能接通一路信号去进行 A/D 转换，因此在进行信号采集时，必须首先选择采集的通道，分配给选择通道号的接口地址为 300H，故若选择 0 通道，仅需执行 outportb(0x300,0)即可。当信号送到 A/D 转换器时，必须进行启动，才能对信号进行转换，分配给启动 A/D 转换的口地址为 301H，由于启动仅是发送一个低电平信号，因而只要执行 outporth(0x301,0)即可。

当转换完成，将模拟信号变成数字信号后才能提取，并将其存入内存。因而必须对 A/D

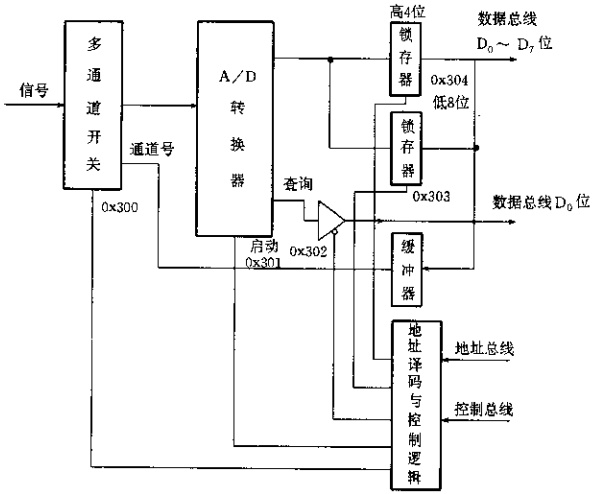


图 7.4 A/D 电路框图

转换完成否进行查询，分配给查询的口地址为 302H，故只要执行：

```
p=inportb(0x302);
```

当得知 p 为 1 时，即表示转换完成，否则继续查询。

当查询得知转换完成后，便将转换后的 12 位数据取走，用口地址 303H 取低 8 位数据，用口地址 304H 取高 4 位数据，最后合成为 12 位数据（并转换成十进制数），即

```
a=inportb(0x303);
b=inportb(0x304);
data=(b&0x0f)*256+a;
```

该程序中 A/D 转换部分，以函数 adin(int ch)形式提供，在主程序中由用户输入选择的信号通道号 ch 传给 adin，以进行转换，转换完成后将转换结果回传给主程序，由主程序将其转换为画图象素的 y 坐标，并画出其象点来，再存到数组 ad[]中。这样一个画面共 640 个点，周而复始，清除掉旧的，画出新的点，屏幕上将动态地显示采样点构成的波形，直到按任意键为止。

程序清单如下：

```
int adin(int ch)
 {
   int a,b,data;
   outportb(0x300,ch);
   outportb(0x301,0);
   begin :
```

```
    a=inportb(0x302);
    a=a&0x01;
    if(a!  =0)goto begin;
    else
    { a=inportb(0x303);
      b=inportb(0x304);
      data=(b&0x0f) * 256+a;
      return(deta);
      }
    }
    #include "graphics.h"
    #include <process.h>
    #include <stdlib.h>
main()
 {
    int m,n,y,i,ch;float da;
    static int ad[640];
    int graphdriver=DETECT,graphmode;
    initgraph(&graphdriver,&graphmode," ");
    if (graphdriver==VGA) n=480,m=2;
    else n=199,m=1;
    printf("%s","          Please input A/D channel number(0-15)");
    scanf("%d",&ch);
     do
     {  for(i=1;i<640;i++)
        { da=adin(ch);
          y=n-(int)(da * n/4095);
          putpixel(i,ad[i],0);
          ad[i]=y;
          putpixel(i,y,m);
          };
     }while(! kbhit());
    closegraph();
    }
```

第 8 章

中断服务程序的编写

所谓中断，是指 CPU 在正常运行程序时，由于程序的预先安排或内外部事件，引起 CPU 中断正在运行的程序，而转到为预先安排的事件或内外部事件服务的程序中去，这些引起程序中断的事件称为中断源。预先安排的事件是指 PC 机的中断指令，执行到此，立即转相应的服务程序去执行。内部事件是指系统板上出现的一些事件信号，中断指令也可看作内部事件，外部事件是指某些接口设备所发出的请求中断程序执行的信号，这些信号称为中断请求信号。中断请求信号何时发生是不能预知的，然而，它们一旦请求中断，则会向 CPU 的接收中断信号的引脚发出电信号，因此这些信号 CPU 是马上可以知道的。这样 CPU 就无需花大量的时间去查询这些信号是否产生。因为中断请求信号一旦产生，便会马上通知 CPU。如键盘，何时有键按下，是随机的，因而 CPU 可以对键盘不加理睬，而去执行其它程序，一旦有键按下，键盘马上产生中断请求信号，CPU 得知这信号后，便立即去执行为键盘服务的中断程序，服务完后，CPU 又恢复执行被中断了的程序。中断服务程序执行完，返回原来执行程序的中断处（称为断点）继续往下执行，称为中断返回。有时中断请求信号（即中断源）可能有好几个，因此 CPU 响应这些中断就得有先后次序，这称为中断的优先级。CPU 首先响应优先级高的中断，优先级低的中断，暂不响应，称为挂起。有些中断源产生的中断，可以用编程的办法使 CPU 不予理睬，这叫中断的屏蔽。CPU 响应中断，转去执行中断服务程序前，需将被中断程序的现场信息保存下来，以便执行完中断服务程序后，接着从被中断程序的断点处继续往下执行。现场信息是指程序计数器的内容、CPU 的状态信息、执行指令后的结果特征和一些通用寄存器的内容，有些信息的保存和程序计数器的内容等由机器硬件预先安排完成，称为中断处理的隐操作。有些信息保存是在中断服务程序中预先安排。CPU 响应中断时，由中断源提供地址信息，引导程序转移到中断服务程序中去执行。这个地址信息称为中断向量，它一般是和中断源相对应的，PC 机采用类型号来标识中断源。

中断方式以其执行速度快，可实时处理，不占用 CPU 过多的时间等优点，在一些高级应用场合中较多地被采用。PC 机中断系统不仅具备一般中断系统的特点，而且有所创新，比如，中断不仅可由外部事件引起，也可由预先安排的事件，或称为内部的事件引起，这些内部事件是指中断指令和执行一些指令引起的特殊事件等。本章将对 PC 机的中断系统加以描述，对中断向量表的填充和中断服务程序的编制及应用分别加以讨论。

8.1 PC 机的中断类型

PC 机有两种类型的中断：由执行某些指令引起的软中断（也可称为内中断）和接口设备引起的硬中断（也可称为外中断），这些类型的中断，均有中断类型号相对应，现就软硬中

断分述如下：

8.1.1 软中断

执行下述指令时，将产生或者可能产生中断，这些中断称为软中断。

1. DIV(除)或IDIV(整除)指令

当执行这些除法指令时，若除数为0或商溢出，则产生中断，这类中断称为0型中断。

2. INT 指令

当执行中断指令 INT n 时，则产生 n 型中断。Turbo C 用库函数 geninterrupt()

3. INTO 指令

若指令序列执行过程中，上条指令执行的结果，使溢出标志位 O=1，接着若执行的是 INTO 指令，则引起内部中断，称为4型中断，若溢出标志位 O=0，该指令将不起作用。

4. 单步执行

当标志位 T=1 时，每执行一条指令，则引起一次中断，使得指令的执行，成为单步执行方式，这种方式用于程序的调试，如 DEBUG 中的跟踪命令 T，就是将标志位 T 置1，进而去执行一个单步中断服务程序，单步执行为1型中断。

8.1.2 硬中断

80X86 CPU 有两条中断请求线：非屏蔽中断 NMI 和可屏蔽中断 INTR 线，当这两条线上收到中断请求信号而引起的中断，称为硬中断。现分述如下：

1. 非屏蔽中断

当 NMI 线上出现一个由低上跳的高电平中断请求信号后(持续时间需大于两个时钟周期)，不管标志寄存器 I 位的状态如何，当前指令执行完后，80X86 CPU 马上转入中断处理。

此种类型的中断有三种来源：系统板上随机存储器 RAM 产生奇偶错，协处理器插座上来的中断请求(仅 XT 机)，I/O 通道检查出错等，这种中断是在 PC 机的系统板上，用户一般是不能用的。

2. 可屏蔽中断

当 INTR 线上出现高电平的中断请求信号时(必须保持到当前执行的指令结束为止)，80X86 是否响应该中断，取决于标志寄存器 I 位的状态，若 I=1，则 CPU 处于开中断，因而可以响应，若 I=0，则 CPU 不响应。I 的状态，可以由汇编设置，若执行开中断指令 STI，则 I 位被置1，Turbo C 有此功能的是库函数 enable()，若执行关中断指令 CLI，则 I 位被清零。Turbo C 有此功能的库函数是 disable()。

在 PC 机上 80X86 的 INTR 中断请求线接 8259A 中断控制器，该中断控制器有进行优先级排队的八条中断请求线，这样，就将 80X86 的 INTR 扩展成八条中断请求线，因而使得 PC 机具有八级中断，其中 0,1,3,4,5,6,7 编号的中断已为系统配置的 I/O 设备所占用，当然用户若不用对应的设备时，则可借用该中断号，不过相应的中断向量表和中断服务程序要改写。

8.1.3 中断向量表

80X86 在内存的前 1024 个字节(即地址为 00000—003FFH)建立了一个中断向量表，

可存储 256 个中断向量，每个中断向量占用 4 个字节，前两个字节为中断服务程序的入口地址偏移量，后两个字节装入了段地址。使用时，将这两个字节分别装入 IP 及 CS 中，以转入中断服务程序，表 8.1 示出了中断向量表，每个中断向量用类型号加以区别，当执行中断时，CPU 根据类型号再乘 4 后，得到中断向量地址，进而得到 IP 及 CS 的值，它就是中断服务程序的入口地址，程序由此转入中断服务程序去执行。中断向量表示例图如图 8.1 所示：

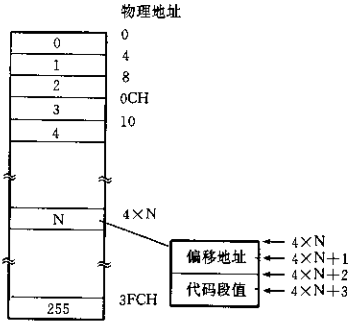


图 8.1　中断向量表

8.1.4　中断向量表的填入

系统开机后，BIOS 立即把有关的中断向量内容填入中断向量表中，包括 0—7FH 一段。待 BIOS 引导 DOS 到内存后，DOS 进行初始化，它又将某些中断向量填入中断向量表中，它还要修改由 BIOS 装入的某些中断向量。有些中断向量，用户是可以用在自己的程序中的，如 60—67H 号中断。绝对地址为 180—19FH 的一段，是专为用户保留的，它可用作用户的软中断。如想使用中断号为 60H 的软中断，用户只要在 4 * 60H＝180H 和 181H 处填入中断服务程序的偏移地址，在 182H—183H 单元填入段地址即可。这样在程序中执行到 geninterrupt(0x60)时，便执行这个中断服务程序。如何用 Turbo C 填入中断向量，将在下面介绍。

对于硬件中断，若要通过 IRQ_i 线产生硬中断，用户必须制作接口电路，以能产生由低到高的中断请求信号，并通过插头接到微机的扩充槽相应的 IRQ_i 脚上，用户可用的是 IRQ_2、IRQ_{10}、IRQ_{11}、IRQ_{12}和 IRQ_{15}，它们的中断向量号分别是 10、72、73、74 和 77，因此可以在中断向量表中填入硬中断服务程序的段和偏移值。上述的硬件中断向量号若不用作硬件中断时，也可将其当作软件中断，不过产生中断要靠发软件中断命令，此时相应的 IRQ_i 脚就无作用了。

8.2　用 Turbo C 编写中断程序的方法

用 Turbo C 实现编写中断程序的方法可用三部分来实现：即编写中断服务程序、安装

中断服务程序、激活中断服务程序，下面分述之。

8.2.1 编写中断服务程序

我们的任务是，当产生中断后，脱离被中断的程序，使系统执行中断服务的程序，它必须打断当前执行的程序，急需完成一些特定操作，因此该程序中应包括一些能完成这些操作的语句和函数，因涉及 DOS 的重入问题(请看 TSR 程序设计中关于 DOS 重入问题的说明)，因而不应该有与 DOS 系统调用有关的库函数，如 printf()，sprint()等。

由于产生中断时，必须保留被中断程序中断时的一些现场数据，即保存断点，这些值都在寄存器中(若不保存，当中断服务程序用到这些寄存器时，将改变它的值)，以便恢复中断时，使这些值复原，以继续执行原来中断了的程序。Turbo C 为此提供了一种新的函数类型 interrupt，它将保存由该类型函数参数指出的各寄存器的值，而在退出该函数，即中断恢复时，再复原这些寄存器的值，因而用户的中断服务程序必须定义成这种类型的函数。如中断服务程序名定为 myp，则必须将这个函数说明成这样：

```
void interrupt myp (unsigned bp,unsigned di, unsigned si, unsigned ds,unsigned es,
                    unsigned dx,unsigned cx,unsigned bx,unsigned ax,unsigned ip,
                    unsigned cs,unsigned flags)
```

若是在小模式下的程序，只有一个段，在中断服务程序中用户就可以像用无符号整数变量一样，使用这些寄存器。

若中断服务程序中不使用上述的寄存器，也就不会改变这些寄存器原来的值，因而也就不需保存它们，这样在定义这种中断类型的函数时，可不写这些寄存器参数，如可写成：

```
void interrupt myp()
  {
  ⋮
  }
```

对于硬中断，则在中断服务程序结束前要送中断结束命令字给系统的中断控制寄存器，其口地址为 0x20，中断结束命令字也为 0x20，即

```
outportb(0x20,0x20);
```

在中断服务程序中，若不允许别的优先级较高的中断打断它，则要禁止中断，可用函数 disable()来关闭中断。若允许中断，则可用开中断函数 enable()来开放中断。

8.2.2 安装中断服务程序

定义了中断服务函数后，还需将这个函数的入口地址填入中断向量表中，以便产生中断时程序能转入中断服务程序去执行。为了防止正在改写中断向量表时，又产生别的中断而导致程序混乱，可以关闭中断，当改写完毕后，再开放中断。一般的，常定义一个安装函数来实现这些操作，如：

```
void install(void interrupt (*faddr)(),int inum)
    {disable();
     setvect(inum,faddr);
```

```
    enable();
}
```

其中 faddr 是中断服务程序的入口地址，其函数名就代表了入口地址，而 inum 表示中断类型号，setvect()函数就是设置中断向量的函数，上述定义的 install()函数，将完成把中断服务程序入口地址填入中断向量 inum 中去。

8.2.3 中断服务程序的激活

当中断服务程序安装完后，如何产生中断，从而执行这个中断服务程序呢？如前所述，对硬件中断，就要在相应的中断请求线(IRQ_i，i=0,1,2,…,7)产生一个由低到高的中断请求电平，这个过程必须由接口电路来实现，但如何激励它产生这个电平呢？这可以用程序来控制实现，如发命令(outportb(口地址，命令))。然后主程序等待中断，当中断产生时，便去执行中断。

对于软中断，有几种调用方法。由于中断类型的函数不同于用户定义的一般函数，因此也不能用调用一般函数的方法来调用它，一般软中断调用可用如下方法：

1. 使用库函数 geninterrupt(中断类型号)

在主函数中适当的地方，用 setvect 函数将中断服务程序的地址写入中断向量表中，然后在需要调用的地方用 geninterrupt()函数调用。

2. 直接调用

如已用 setvect(类型号，myp)设置了中断向量值，则可用 myp()；直接调用，或用指向地址的方法调用：

```
(*myp)();
```

3. 也可用在 Turbo C 程序中插入汇编语句的方法来调用，如：

```
setvect (inum,myp);
        ⋮
asm int inum;
        ⋮
```

不过用这种方法的程序生成执行程序稍麻烦点。

通常上述的调用可定义成一个中断激活函数来完成，该函数中可附加一些别的操作，主程序在适当的地方调用它就可以了。

4. 恢复被修改的中断向量

这一步视情况而定，当用户采用系统已定义过的中断向量，并且将其中断服务程序进行了改写，或用新的中断服务程序代替了原来的中断服务程序，为了在主程序结束后，恢复原来的中断向量以指向原中断服务程序，可以在主程序开始时，存下原中断向量的内容，这可以用取中断向量函数 getvect()来实现，如 j=(char *)getvect(0x1c)，这样 j 指针变量中将是 0x1c 中断服务程序的入口地址，由于 DOS 已定义了 0x1c 中断的服务程序入口地址，但它是一条无作用的中断服务，因而我们可以利用 0x1c 中断来完成一些用户想执行的一些操作，实际上就是用户自己的中断服务程序代替了原来的。当主程序要结束时，为了保持系统的完整性，我们可以恢复原来的中断服务入口地址，如可用

```
setvect(0x1c,j)
```

也可以再调用 install()函数再一次进行安装。

一般情况下可以不加这一步。

PC XT 机上由 8259A 中断控制器来处理 8 个外中断，如图 8.2 所示，当某个外中断信号 IRQ_i(i=0—7)产生中断请求时，8259A 接收到此信号(当多个同时请求时，则根据优先级，如表 8.1 所列)再向 8088 发中断请求信号，当 8088 认可后，便发确认信号$\overline{INTA}$而产生中断，程序转向由中断向量指出的中断服务程序去执行。中断服务程序的入口地址装于中断向量表中。XT 机分配给 IRQ_0—IRQ_7 的中断向量类型号为 8—15，该号再乘 4 即为中断向量地址(它占 4 个字节)，在该地址中装入该型中断的中断服务程序入口偏移地址，后两个字节装入段地址，XT 机中仅保留了 IRQ_2 给用户，所以一般仅可用 IRQ_2 硬中断。在 PC/XT 机的扩充插槽上(称为 PC 总线)引出了 IRQ_2—IRQ_7。

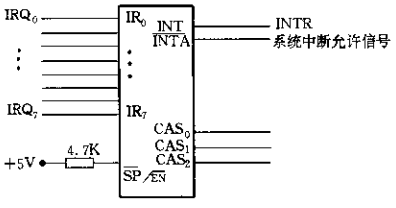


图 8.2 XT 的中断控制器

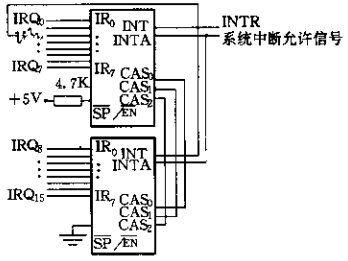


图 8.3 级联的中断控制器

286、386、486PC 机使用了两个级联的 8259A 中断控制器，如图 8.3 所示。由于采用了超大规模集成电路，因而该类微机将两个 8259A 和其它外设控制芯片集成在一块外设控制器芯片上了，所以我们在系统板上看不到 8259A 芯片。这两个级联的 8259A 有主从之分，SP/$\overline{EN}$脚接高电平者为主片，接地者为从片。当从片接收到外界中断请求信号后，便向主片发出中断请求信号 INT(而不是向 CPU)，这个 INT 又作为主片的 IRQ_2 中断请求信号，主片收到 IRQ_2 中断请求信号后，又向 CPU 发出中断请求信号 INT，当 80x86 CPU 发出中断认可信号$\overline{INTA}$时，程序便进入中断服务程序去执行。由于在 BIOS 对主片 8259A 初始化时，已设定 IRQ_2 线上连有级联的从片 8259A，这样主片从 CAS0—CAS2 端发出识别码给从片，由从片 8259A 通过 CAS0—CAS2 端接收并确认一致后，即由其数据输出端(即 D_0—D_7，它们和主片的 D_0—D_7 端并接，图中未画出)将中断类型码送到 80x86，再由其乘 4 后，作为中断向量的地址，再将该地址中的内容装 IP 及 CS 寄存器中，从而执行相应的中断服务程序。

对 PC286、386、486 等微机由于采用了级联 8259A 中断结构，因而扩充了中断请求信号线。在它的扩充插槽上(称为 AT 总线)引出了 IRQ_2—IRQ_7 和扩充的 IRQ_{10}、IRQ_{11}、IRQ_{12}、IRQ_{14}、IRQ_{15} 中断请求信号线，而对 IRQ_9 仍当作 IRQ_2 来处理。即当在该中断请求线上产生中断请求时，处理程序将其仍认为是 IRQ_2 中断，即中断向量立即被改为 0A 型中断，由于这条线未引出，所以硬件上无法使用，但我们可用其中断号 71 作为软中断。但执行时，该中断

立即又转向 0A 型中断。

PC286、386、486 上使用的硬中断请求信号和对应的中断向量号与外接的设备如表 8.1 所示，其硬中断系统结构如图 8.4 所示。

表 8.1 PCX86 硬中断请求信号与中断向量号及外设的关系

中断请求信号	中断向量号	使用的设备
IRQ_0	8	系统定时器
IRQ_1	9	键盘
IRQ_2	10	对 XT 机保留，AT 总线扩充为 IRQ8—15
IRQ_3	11	RS-232C(COM2)
IRQ_4	12	RS-232C(COM1)
IRQ_5	13	硬盘中断
IRQ_6	14	软盘中断
IRQ_7	15	打印机中断
IRQ_8	70	实时钟中断
IRQ_9	71	软中断方式重新指向 IRQ_2
IRQ_{10}	72	保留
IRQ_{11}	73	保留
IRQ_{12}	74	保留
IRQ_{13}	75	协处理器中断
IRQ_{14}	76	硬盘控制器
IRQ_{15}	77	保留

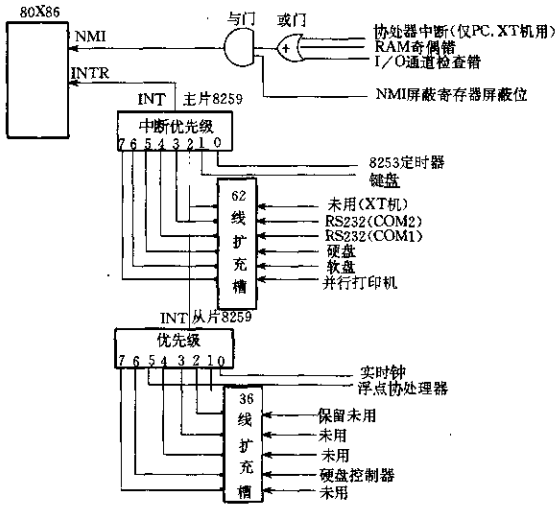


图 8.4 硬中断系统

中断请求信号的优先级

从图 8.4 可以看出，中断优先级从高到低排列顺序为：IRQ_0，IRQ_1，IRQ_2，IRQ_8，IRQ_9，IRQ_{10}，IRQ_{11}，IRQ_{12}，IRQ_{13}，IRQ_{14}，IRQ_{15}，IRQ_3，IRQ_4，IRQ_5，IRQ_6，IRQ_7。

8.3 中断服务程序例

8.3.1 硬中断演示程序——秒表

这个程序是一个硬中断程序，我们知道 PC 系统以每秒 18.2 次的频率进行时钟硬中断（使用中断请求 IRQ_0），即执行 8 号中断。这个中断周而复始的在进行，在它的中断服务程序中除了进行日时钟计数和磁盘驱动器超时检测控制外，接着又进行 0x1c 的软中断调用，0x1c 软中断只有一条返回指令，它不作什么事情，因而我们可以改写它的内容，使其变为一个有用的软中断服务程序。在这个例子中，我们利用每秒 18.2 次的定时硬中断每秒要调用 18.2 次的软中断 0x1c，将中断 0x1c 中断服务程序改写为对进入该中断的次数进行计数的程序，每到 18 次时，在屏幕的右上角开一个窗口（window(50,1,54,3)，在窗口的中间位置显示 0—9 十个数字中的一个，频率接近于秒表数（不过只显示十个数）。由于这是一个硬中断演示程序，计时并不准确，若要精确计时，则应 91 次 0x1c 中断为 5 秒。

该程序中用 programsize 表示程序长度，设置为 400 节，由于在小模式下，Turbo C 使用 64K 的一个段，其中段的使用分配如图 5.2 所示。

```
#include<dos.h>
#include<conio.h>
#define programsize 400
#define TRUE 1
void interrupt(*oldtimer)();
void interrupt newtimer();
void install();
static struct SREGS seg;
int b=0;int b1=0;
unsigned intsp,intss;
unsigned myss,stack,x0,y0;
int busy=0;
void on_timer();
void goxy();
void rexy();
void prt();
main()
  {
    char ch;
    char *p;
    p=(char *)newtimer;
    on_timer(p);
    while(TRUE)
```

```
    {
      ch=getch();                          /*键盘输入 0、1 或 q 进行功能选择*/
      switch(ch){
      case '0':
        install( oldtimer);
        break;
      case '1':
        b1=47;
        install(newtimer);
        break;
      case 'q':
        install(oldtimer);
        exit(1);
      default:                             /*若是其它字符,则打印出来*/
        printf("%c",ch);
      }
    }
  }
void on_timer(p)
  int (*p)();                              /*指向中断函数*/
  {
    segread(&seg);
    stack=programsize*16;                  /*设置堆栈*/
    myss=_SS;
    oldtimer=getvect(0x1c);                /*得到原 0x1c 中断向量*/
  }
void install(void interrupt (*faddr)())
  {disable();
    setvect(0x1c,faddr);                   /*设置 0x1c 新中断向量*/
  }enable();
void interrupt newtimer()
  {
    (*oldtimer)();                         /*指向旧中断程序*/
    if(busy==0){
      busy=1;
      disable();
      intsp=_SP;                           /*保存 SP,SS 值*/
      intss=_SS;
      _SP=stack;                           /*设置 SP,SS 为新值*/
      _SS=myss;
      enable();
      rexy();                              /*得到当前光标位置*/
      clrscr();
      window(50,1,54,3);
```

```
        textcolor(YELLOW);
        textbackground(RED);
        b+=1;
        if(b==18){                          /*如果中断18次,则打印秒计数值*/
        gotoxy(3,2);
        b1++;
        b=0;
        if(b1>57)b1=48;                     /*计数值超过9,则令b1为0*/
          prt(b1);                          /*显示计数值*/
          }
        goxy(x0,y0);                        /*光标返回原处*/
        disable();
        _SP=intsp;                          /*恢复SP,SS的原值*/
        _SS=intss;
        enable();
        busy=0;
      }
    }
  void goxy(int x,int y)                    /*光标回到x,y处*/
    {
      union REGS rg;
      rg.h.ah=2;
      rg.h.dl=y;
      rg.h.dh=x;
      rg.h.bh=0;
      int86(0x10,&rg,&rg);
    }
  void rexy()                               /*得到光标坐标*/
    {
      union REGS rg;
      rg.h.ab=3;
      rg.h.bh=0;
      int86(0x10,&rg,&rg);
      y0=rg.h.dl;
      x0=rg.h.dh;
    }
  void prt(p)                               /*显示字符*/
    {
      union REGS rg;
      rg.h.al=p;
      rg.b.ah=14;
      int86(0x10,&rg,&rg);
    }
```

8.3.2 用 geninterrupt()函数产生中断

该程序采用 geninterrupt(inum)函数来调用中断函数，相当于产生一个软中断，即程序执行到该函数处，自动转到由中断号(inum)所指出的中断服务程序去执行。此处中断号为0x0a，它的中断向量指向 beep 中断服务程序，即喇叭发声函数。中断向量的设置是通过 install 函数，为了防止设置时，又响应别的中断，导致设置失败，因而用 disable()函数来禁止中断，设置完后，再用 enable()开放中断。

对于扬声器发声函数 beep 的说明，要涉及到一些对系统板上的硬件电路的控制问题。关于 PC 机上的扬声器发声原理，已在 I/O 接口的输入输出函数中的示例程序中作了说明，可参阅该段的示例程序。

由于硬中断 IRQ_2 规定为用户所用的中断，当在 PC 插槽中没有插入使用该中断的接口卡时(这种接口卡不是 PC 系统的标准配备)，可用 IRQ_2 中断所对应的中断号 0x0a。该程序借用该中断号来作软中断使用，即 0x0a 中断不是由于接口卡上的中断请求信号 IRQ_2 产生的，而是执行一条软中断指令 INT 10 产生的，该指令在 C 程序中又是由 geninterrupt(10)函数来产生。

```
#include<dos.b>
main()
  {
    char ch;
    void interrupt beep();
    void install();
    void testbeep();
    install(beep,0x0a);
    testbeep(0x0a);
  }
  /* interrupt processing */
  void interrupt heep (unsigned bp,unsigned di,unsigned si,
                       unsigned ds,unsigned es,unsigned dx,
                       unsigned cx,unsigned bx,unsigned ax)
    {
      int i,j;
      char originalbits,bits;
      /* get the current control port setting */
      bits=originalbits=inportb(0x61);
      /* close the speaker */
      outportb(0x61,bits&0xfc);
      /* set timer command */
      outportb(0x43,0xb6);
      /* set  frequency of the speaker signal */
      outportb(0x42,989);
      outportb(0x42,5);
```

```
    /* open the speaker */
    outportb(0x61,bits | 3);
    getch();
    /* close speaker */
    outportb(0x61,bits&0xfc);
    /* restore the control port setting */
    outportb(0x61,originalbits);
  }
 /* set interrupt vector */
void install(void interrupt(*faddr)(),int inum)
  {
    disable();
    setvect(inum,faddr);
    enable();
  }
void testbeep( int inum )
  {
    clrscr();
    printf("press any key to exit");
    geninterrupt(inum);
    }
```

8.3.3 扬声器唱歌程序

这个程序是利用扬声器唱歌程序，发声原理同上一例类似，当运行完该程序回到 DOS 提示符下，表示已将 0x1b 中断向量改写，指向了 newint 中断服务程序，当我们接着再按 CTRL_BREAK 两键时，便响起了“友谊地久天长”歌，即执行了 newint 中断服务程序。结束后，屏幕出现 A>C，然后又回到 DOS 提示符下。现来看一下这是如何实现的。

我们知道，系统启动时，BIOS 给 0x1b 中断向量设置指向了一个空处理程序，接着引导 DOS 时，DOS 又将该中断向量改写，指向了一个新处理程序，该程序只是增加了设置 CTRL_BREAK 标记的功能，再无什么操作。因而用户可以改写这个中断服务程序。我们再来看一下是如何产生 INT 1BH 调用的。当我们按 CTRL_BREAK 两键时，便产生 INT9 键盘中断，该程序然后判断，若是 CTRL_BRAK 两键按下，便调用 0x1b 中断，而该中断执行的结果，则在屏幕上显示 A>C，然后结束中断，回到系统。

这个例子程序改写了 0x1b 中断服务程序，起名为 newint()。该函数首先保存了原中断服务程序的 SP 和 SS，然后设置了自己的新值。由于发声的数据以数组形式存放(存在 freq[],dely[]中)，因而可将 SS 设置为数据段 DS，这样以免运行程序结束后，破坏原数据。当该程序结束时，又调用原来的 0x1b 中断，并恢复原先的 SP 和 SS 的值。

若要恢复原来的 0x1b 中断，只有重新引导 DOS。

```
#include<stdio.h>
#include<stddef.h>
#include<dos.h>
```

```
#include<stdlib.h>
void interrupt(*oldint)();
void interrupt newint();
unsigned char stack[0x1000];
unsigned intsp,intss;
unsigned freg1[87] frep[87]={
    196,262,262,262,330,294,262,294,330,294,262,
    330,394,440,440,394,330,330,262,294,262,294,
    330,294,262,230,230,196,262,440,394,330,330,
    262,294,262,294,440,394,330,330,394,440,523,
    394,330,330,262,294,262,294,330,294,262,230,
    230,196,262,440,394,330,330,262,294,262,294,
    440,394,330,330,394,440,523,394,330,330,262,
    294,262,294,330,294,262,230,230,196,262};
int dely[87]={
    25,38,12,25,25,38,12,25,12,12,56,25,25,50,25,
    38,12,12,12,38,12,25,12,12,38,12,25,25,100,25,
    38,12,12,12,38,12,25,25,38,12,25,25,100,25,38,
    12,12,12,38,12,25,12,12,38,12,25,25,100,25,38,
    12,12,12,38,12,25,25,38,12,25,25,100,25,38,12,
    12,12,38,12,25,12,12,38,12,25,25,100};
main()
  {
    oldint=getvect(0x1b);
    setvect(0x1b,newint);
  }
void interrupt newint (bp,di,si,ds,es,dx,cx,bx,ax,ip,cs,flags)
{
  int i;
  char orignalbits, bits;
    disable();
    intsp=_SP;
    intss=_SS;
    _SP=(unsigned)&stack[0x1000-2];
     SS=_DS;                          /*数据段和堆栈段同*/
    enable();
  for(i=0;i<87;i++)
    { outportb(0x43,0xb6);
      freq1[i]=0x1234dc/freq[i];
      outportb(0x42,freq1[i]&0x00ff);
      freq1[i]=freq1[i]>>8;
      outportb(0x42,freq1[i]);
      bits=orignalbits=inportb(0x61);
      outportb(0x61,bits | 3);
```

```
        delay(dely[i] * 25);
        outportb(0x61,orignalbits);
        }
        oldint();                          /* 调用原 0x1b 中断 */
        disable();
        _SP=intsp;
        _SS=intss;
        enable();
      }

    #include<stdio.h>
    #include<stddef.h>
    #include<dos.h>
    #include<stdlib.h>
  void interrupt far music()
  {
    int i;
    unsigned freq[87]={
      196,262,262,262,330,294,262,294,330,294,262,
      330,394,440,440,394,330,330,262,294,262,294,
      330,294,262,230,230,196,262,440,394,330,330,
      262,294,262,294,440,394,330,330,394,440,523,
      394,330,330,262,294,262,294,330,294,262,230,
      230,196,262,440,394,330,330,262,294,262,294,
      440,394,330,330,394,440,523,394,330,330,262,
      294,262,294,330,294,262,230,230,196,262};
    int dely[87]={
      25,38,12,25,25,38,12,25,12,12,56,25,25,50,25,
      38,12,12,12,38,12,25,12,12,38,12,25,25,100,25,
      38,12,12,12,38,12,25,25,38,12,25,25,100,25,38,
      12,12,12,38,12,25,12,12,38,12,25,25,100,25,38,
      12,12,12,38,12,25,25,38,12,25,25,100,25,38,12,
      12,12,38,12,25,12,12,38,12,25,25,100};
    char originalbits,bits;
    system("cls");
    gotoxy(4,12);
    printf("sing singing\n");
    for(i=0;i<87;i++)
      { outportb(0x43,0xb6);
        freq[i]=0x1234dc/freq[i];
        outportb(0x42,freq[i]&0x00ff);
        freq[i]=freq[i]>>8;
        outportb(0x42,freq[i]);
        bits=originalbits=inportb(0x61);
```

```
        outportb(0x61,bits | 3);
        delay(dely[i] * 25);
        outportb(0x61,originalbits);
      }
    }
main()
{ void(interrupt far *intp)();
  intp=getvect(0x1b);
  setvect(0x1b,music);
  intp();
}
```

8.3.4 采用中断方式的信号采集程序

该程序是作者研制的光隔离 A/D 电路的信号采集示范程序。它采用硬中断方式，共有 48 路信号输入通道，可采集电压信号，0—10mA 电流信号，4—20mA 电流信号，该电路在模拟转换部分和 PC 机数字输入部分之间有一光隔离带，使得 PC 微机和信号现场部分进行了隔离，使得它们之间不发生电的联系，而通过光作为媒介来传递转换后的信号，这样可有效地防止一些现场电的干扰，并保护微机不受偶然事故的破坏。图 8.5 示出了这个电路的示意图，图中的光隔块就表示了光隔离带。它的工作过程是这样的，首先清 A/D 中断寄存器，使其变低，当 A/D 转换完成后，利用转换完成信号(EOF)将其置 1，从而产生 IRQ_2 中断请求信号，该寄存器口地址设为 0x303，利用 outportb(0x303,0)可将其清 0，这样就为采样作好了准备，接着选择信号输入通道，其口地址为 0x300。若用 ch 表示输入通道号(0—47)，则可用 outportb(0x300,ch)来选择输入通道，接下去便启动 A/D 开始转换，其口地址为 0x301。即 outportb(0x301,1)使数模转换芯片开始进行转换。函数 enairq9()完成这些任务。一旦转

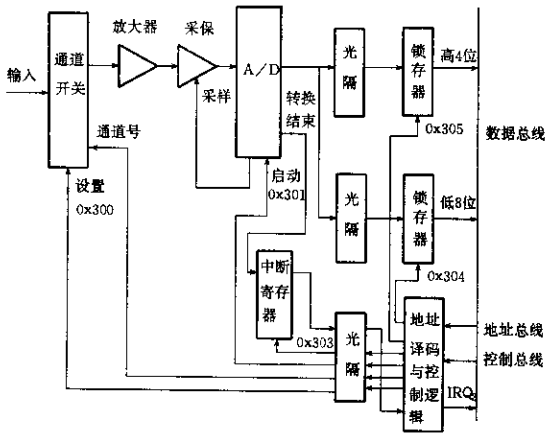


图 8.5 光隔离 12 位 A/D 转换电路

换完成，它便发出 EOF 信号，该信号将 AD 中断寄存器置成高电平输出，因而产生 IRQ_2 中断请求信号。当 PC 机接到该中断后，便进入中断服务程序 int9()。该程序首先禁止中断，以防破坏取数。由于光隔离的存在，将光变成 PC 微机一侧的数据有时间延迟，因而程序中用了无语句的 for 循环来进行延迟，用 n 控制延迟时间。由于使用 PC 的 8 位数据总线，因而对于 12 位 A/D 转换电路，转换后的数据要分两次取走，先取低 8 位，后取高 4 位（实际上取了高 8 位，因而用位的与运算将高 8 位的后 4 位取出）。它们用了口地址 0x304 和 0x305。取数用 inportb 函数。12 位数据取走后，中断服务程序就完成了任务，因而必须最后发送硬中断结束命令：outportb(0x20,0x20)，为下次中断作好准备，并开中断。

由于硬中断 IRQ_2 的中断类型号为 0x0a，on_intr9()，函数用来设置中断入口地址，将中断服务程序 int9 的入口地址置入 0x0a 指出的偏移地址中。

该程序中 choice_channel()函数用来接收用户键入的要打开的输入通道号。因该电路板规定为 48 个通道（通道号为 0—47），故该函数检测键入的通道号是否在 0—47 之内，若不是则发声并要求重新输入。若正确则返回输入的通道号。该函数用到了库函数 isdigit()，它用来判断输入的通道号是否为数字，该库函数包含在头文件 ctype.h 中。

```
#define TRUE   1
#define FALSE  0
#define END_OF_INT 0x20
#define N 80

#include <process.h>
#include <stdlib.h>
#include <ctype.h>
#include <dos.h>
int start,ch,flag;
void interrupt int9();
void on_intr9();
void enairq9();
void disirq9();
int ad[641];
int i;
char *ab;

main()
  {
    char c;
    clrscr();
    printf("          Please input A/D channel number(0—47) ");
    ch=choice_channel();
    on_intr9();
    i=0;
    do{
        clrscr();
```

```
        flag=0;
        enairq9();
        to;
        if(flag==0)goto to;
        i++;
        ad[i]=ab;
        if(i==640){
          for(i=1;i<=640;i++)
            printf(" %d ",ad[i]);
          i=0;
          printf("press q key to quit");
          c=getch();
          if (c=='q')break;
          continue;};
          }while(1);
         disirq9();
         printf("\n\r");
         exit(0);

  }

int choice_channel()                    /* 选择 A/D 输入通道 */
  {int c;
  static char tem[4]={0,0,0,0};
  i=0;
  do
    { tem[i]=getch();
      if(isdigit(tem[i])&&(i<2))        /* 若是数字且少于 2 位 */
        { i++;
          gotoxy(48+i,1);
          printf("%c",tem[i-1]);        /* 打印出来 */
          continue;
        }
      else
        if(tem[i]=0x0d)                 /* 若是回车 */
        {
        c=atoi(tem);                    /* 转换成整数 */
        if(c>=0 && (c<=47))             /* 若不是 0--47 通道号 */
          break;                        /* 重新输入 */
        }
        i=0;
        sound(1000);
        delay(200);
        nosound();
```

```
        }while(1);
        return c;
        }
void on_intr9()
  {
    disable();
    setvect(0x0a,int9);
    enable();
  }
void interrupt int9(bp,di,si,ds,es,dx,cx,bx,ax,ip,cs,flags)
   {
     int j,a,b;
     disable();
     if (start)
       {
        start=FALSE;
          for(j=0;j<N;j++);                    /*时间延迟*/
       a=inportb(0x304);                       /*取低8位数*/
       for(j=0;j<N;j++);
       b=inportb(0x305);                       /*取高4位数*/
       ah=(b&0x0f)*256+a;                      /*得到12位A/D转换后的数*/
       outportb(0x20,END_OF_INT);              /*发硬中断结束命令*/
       enable();
       flag=1;                                 /*中断标志置1*/
        }
       }
  void enairq9()
   {
    int a;
    enable();
    start=TRUE;
    outportb(0x303,0);                         /*清A/D中断寄存器*/
    for (a=0;a<N;a++);
      outportb(0x300,ch);                      /*送A/D通道号*/
    for (a=0;a<N;a++);
      outportb(0x301,1);                       /*启动A/D转换*/
   }
  void disirq9()
   {
    start=FALSE;
    outportb(0x303,0x00);                      /*A/D中断寄存器*/
   }
```

8.3.5 定时中断程序

图 8.6 示出了一个采用 8253 定时器和可编程的外围接口芯片 8255A 构成的一个定时中断的电路，中断时由 8255A 的 A 口和 B 口送出开关量，以控制相应的设备（该部分未画出）。

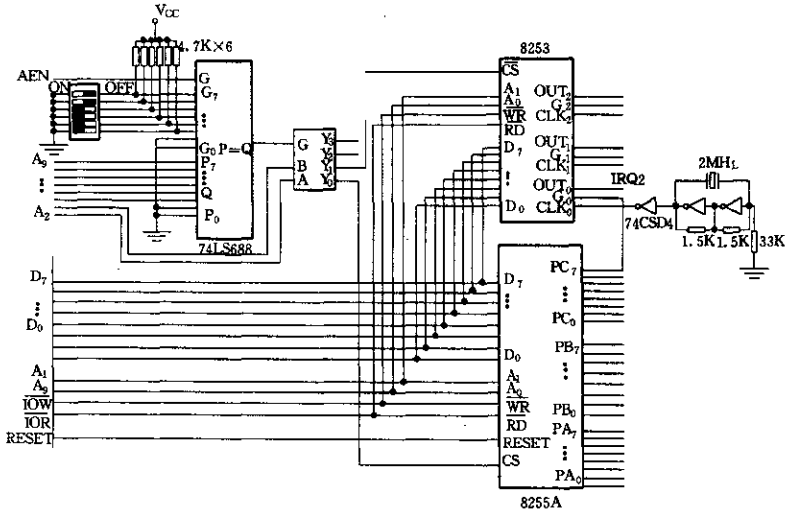


图 8.6 产生定时中断的电路

I/O 口地址译码部分采用 74LS688 比较器，通过 DIP 开关可设置不同的 I/O 口地址范围，图中已设置成 300H—30FH 口地址范围。由于 8253 或 8255 仅需 4 个口地址，故对低位地址由 74LS139 进一步译码，从而译出 4 个地址范围，$\overline{Y}_0$ 用到 8255，再和 A_0 和 A_1 地址码配合，可译出 300H—303H 4 个口地址。同理 8253 用 Y_1，因而 8253 使用 304H—307H 4 个口地址。

8253 的 0 号计数器的时钟信号 CLK 采用由 2M 石英晶体构成的脉冲发生器来产生，它编程工作在模式 3 下，即形成一个方波频率发生器。当装入计数值 N 后，对 $GATE_0$ 端加上高电平，8253 便开始减计数（以 2MHz 频率）。若 N 为偶数，则减到 N/2 时，OUT_0 输出端变低。重复上述的过程，OUT_0 端将输出宽度为 N/2 个时钟周期的方波，其频率为 2M/N。用它来作为 IRQ_2 中断请求信号，则将发生频率为 2M/N Hz 的中断请求。图中 8253 的 $GATE_0$ 端由 8255 的 PC7 来控制。

8255 的 A 口和 B 口及 C 口均编程为输出口，当 8253 的 OUT_0 输出定时脉冲产生中断时（OUT_0 信号作为 IRQ_2），则由 8255 的 A 口和 B 口产生开关量输出，以控制相应设备动作（该部分未画出）。

用 Turbo C 编程序控制该电路由 8255 定时地送出开关量。控制外部设备开和关的中断程序分成三个部分，即设置中断向量并启动产生中断请求信号的电路启动部分；CPU 接收中断后，执行中断服务的程序部分；当需要停止中断时，进行停止中断的程序部分。它们分别用函数来实现，这三个函数分别是 enairq9()、int9()和 disirq9()，该程序名为 intq.c。现分别作以介绍。

enairq9()函数用来设置中断向量，由于该电路采用 IRQ_9 中断，它实际上就是 IRQ_2 中断，故中断号为 0AH，用 setvect(0x0a,int9)就设置好了该中断向量。该函数还设置 8255 和 8253 工作模式。我们知道 8255 是一个可编程外围接口芯片，它具有三个并行输入输出端口(A 口，B 口和 C 口)，能在三种方式下工作，C 口还可作为控制口用于输出控制信号和状态信号。在何种方式下工作，C 口作为什么口来使用，这都可以向 8255 的控制寄存器写控制命令字来进行设置。如何设置可参阅有关书籍(如本人写的 IBM PC/XT 接口技术及其应用一书)。在本函数中用

```
outportb(P8255_CTRL,0x80);
```

设置 8255 工作在方式 0 下，即 A 口和 B 口及 C 口均为输出口，且 C 口具有按位进行置 1 和清 0 的功能。这样可以给 A 口和 B 口各送 8 位数据作为开关量输出，给 C 口第 7 位(即 PC7)送 1 或 0(即置位或复位)来控制 8253 的 $GATE_0$ 端。

8253 是一个有三个定时器/计数器的外围电路芯片，可编程使其具有定时或计数功能，它含有三个 16 位的二进制计数器，8253 可有六种工作方式，当对控制寄存器写入相应的控制字时，则可使其工作在规定的方式下，本函数设定计数器 0 在方式 3 下，即方波频率发生器方式，用 outportb(P8253_CTRL,0x37)来设置，设置频率为 960Hz，计数值相当于 0x2083，故根据控制字的设置，装计数值时，先装低字节，后装高字节，这样因函数开始已通过 8255 的 C 口第 7 位将 $GATE_0$ 置高，故 8253 将周而复始的产生方波，即产生中断请求，其周期大约 1ms。

函数中用 outportb(P8255_C,0x80)使 $GATE_0$ 变高，且只有装入计数值后才能开始计数，故函数后面立即设置了计数值，当计数值装入后，8253 就开始自动减计数操作。

int9()是中断服务程序，这个函数名字前面必须加上关键字"interrupt"来进行说明，且按规定，中断服务程序必须先于设置中断向量的程序说明，这样，编译时，先知道中断服务程序的入口地址，因而才能正确地设置中断向量。

disirq9()函数是禁止中断的函数，它将等待按键，当任一键按下后，它便使 8255 的 A 口和 B 口输出为零，且在下一个中断来时，使 8255 的 C 口也为 0，因而停止中断信号的产生。

test.c 是主程序，它对要输出的开关量进行设置。当每产生一次中断时，输出一组开关量，1ms 后(下一个中断)再清零，因而开关量持续时间 1ms，8 次中断一个循环，周而复始。当按下任一键时，便退出。程序中 while 命令的目的实际上就是等中断。

```
/*
  File name :int9.c
  C version :Turbo C v2.0
  Funcation :int9().....interrupt IRQ9 service routine
            :enairq9()...start 8253 timer and enable IRQ9
```

```
                ;disirq9()...stop 8253 timer and disable IRQ9
*/
#difine P8255_CTRL          0x303
#difine P8255_A             0x300
#difine P8255_B             0x301
#difine P8255_C             0x302
#difine P8253_CTRL          0x307
#difine P8253_0             0x304
#difine TRUE                1
#difine FALSE               0
#difine END_OF_INT          0x20

#include <dos.h>
int buffer[2][8];
int send=0;
int start,stop;

void interrupt int9()
{
    if (start)
    {
      outportb(P8255_A,0);
      outportb(P8255_B,0);
      outportb(P8255_C,0x80);
      outportb(P8255_A,buffer[0][send];
      outportb(P8255_B,buffer[1][send];
      send=send+1 & 0x07;
      stop=False;
    }
    else
    {
      outportb(P8255_c,0x00);
      send=0;
      stop=TRUE;
    }
    outportb(0x20,END_OF_INT);
}

void enairq9()
{
    /* IRQ9....type 10 interrupt */
    setvect(0x0a,int9)
    /* 8255A mode setting:
```

```
        port A and port B. . . . output port
        port C . . cntral port,PC7 start/stop 8253 timer 0
      * /
     outportb(P8255_CTRL,0x80);

     / * start 8253 timer 0 * /
     outportb(P8255_c,0x80);

     start=TRUE;
     stop=FALSE,

     / * 8253 timer 0 mode setting * /
     outportb(P8253_CTRL,0x37);

     / * count value=(2,000,000/(16 * 60))=2083 * /
     / * send frequency:960Hz * /
     outportb(P8253_0,0x83);
     outprotb(P8253_0,0x20);
   }
   void disirq9()
   {
     start=FALSE;
     while (stop ! =TRUE);
     outportb(P8255_A,0x00);
     outportb(P8255_B,0x00);
   }
C> type test.c
/ *
   File name   :test.c
   Function    :test interrupt 9
   * /
   #define TRUE    1
   #define FALSE   0
   #include <dos.h>
   #include <coio.h>
   #include <process.h>
   extern int buffer[2][8];
   main()
   {
     int Y;
     int a[8]={0x80,0x00,0xe0,0x00,0xf0,0x00,0xc0,0x00};
     int b[8]={0x01,0x00,0x03,0x00,0x05,0x00,0x07,0x00};
```

```
    enairq9();
    for (Y=0;Y<8;Y++)
    {
      buffer[0][Y]=a[Y];
      buffer[1][Y]=b[Y];
    }
    while(TRUE)
    {
      if(kbhit())
      {
      disirq9();
      getch();
      exit(0);
      }
    }
}
```

8.3.6 能被 Turbo C 程序调用的汇编语言中断程序

当要求快速响应和完成中断服务时，可用汇编语言编写，主程序则可使用 C 语言来编写。因为中断服务程序的执行时间要小于两次中断的时间间隔。尤其在定时中断时，要严格遵循此原则，因此采用汇编来编写中断程序是最佳的选择。

这个例子完全是上面程序的翻版，它也是针对图 8.5 所示电路编程的，它也由三个过程来实现前一例子中相应的三个函数 int9、enairq9()和 disirq9()，即中断服务程序部分，中断向量设置和启动部分及中断停止部分。不过用汇编语言编写时，它们的结构要符合 C 语言的要求，下面简述之。

由于中断服务程序是独立执行的，故应是一个远过程(far)，用大写字母命名为 INT9(它不受大小写字母和前面要有下划线的限制)。由于所编 C 语言程序定为工作在小模式下，因而可被它调用的_enairq9()和_disirq9()应定义成近过程(near)，其过程名必须用小写字母，且前面必须要有一条下划线，这样就相当于 C 语言中的 enairq9()和 disirq9()。由于用 C 所编主程序中用到二维数组 buffer[2][8]，在汇编中只能用一维来处理，即写成_buffer[16]，而分别用前后两部分_buffer[BX]和_buffer[BX+8]来表示(BX—0—7)，且前面有下划线。这样该程序和 C 程序连接时，可由 C 语言程序存取该变量内的值。

在汇编程序中设置_BSS 段来存未设初值的变量，在_DATA 段中则存放有初值的变量。当在汇编程序中定义了变量和过程时，必须用 PUBLIC 进行说明，以表示为公用变量或过程，它可以被别的程序调用。而在调用它程序中必须说明成外部的，即汇编语言中用 EXTERN 说明，C 语言中用 extern 说明。

中断服务程序开头必须用 PUSH 指令来保存所有要用到的寄存器，而中断返回前用 POP 指令来恢复它们原来的值。该程序结尾用：

```
MOV   AL,20H
OUT   20H,AL
```

它是中断结束信号，这样就为下次响应中断作好了准备。

下面是 Turbo C 所编主程序 test.c 和汇编中断程序 INT9.ASM。

```
/*
  File name   :test.c
  Function    :test interrupt 9
  */
  #define TRUE    1
  #define FALSE   0
  #include <dos.h>
  #include <conio.h>
  #include <process.h>
  extern int buffer[2][8];
  extern enairq9();
  extern disirq9();
  main()
  {
    int Y;
    int a[8]={0x80,0x00,0xe0,0x00,0xf0,0x00,0xc0,0x00};
    int b[8]={0x01,0x00,0x03,0x00,0x05,0x00,0x07,0x00};
    enairq9();
    for (Y=0;Y<8;Y++)
    {
      buffer[0][Y]=a[Y];
      buffer[1][Y]=b[Y];
    }
    while(TRUE)
    {
      if (kbbit())
      {
      disirq9();
      getch();
      exit(0);
      }
    }
}

;IRQ(AT)? IRQ@(XT) INTERRUPT SERVICE ROUTINE

;_enairq9__START 8253 TIMER TO ENABLE IRQ9
;_disirq9__STOP 8253 TIMER TO DISABLE IRQ9
;          INTERRUPT FREQUENCY:960Hz
;
```

```
;
P8255_CTRL      EQU  303H
P8255_A         EQU  300H
P8255_B         EQU  301H
P8255_C         EQU  302H
P8253_CTRL      EQU  307H
P8253_0         EQU  304H
TRUE            EQU  1
FALSE           EQU  0
END_OF_INT      EQU  20H
;
;
        DGROUP      GROUP    _BSS
_BSS    SEGMENT WORD PUBLIC 'BSS'
        PUBLIC      _buffer
_buffer         DB      16 DUP(?)
STOP            DW      ?
START           DW      ?
SEND            DB      0
_BSS            ENDS
_TEXT       SEGMENT  BYTE  PUBLIC 'CODE'
            PUBLIC   _enairq9
            PUBLIC   _disirq9

            ASSUME CS:_TEXT,DS:DGROUP,SS:DGROUP
INT9        PROC     FAR            ;ISR FOR IRQ2(XT) OR IRQ9(AT)
            PUSH     AX
            PUSH     BX
            PUSH     DX
            PUSH     DS
            MOV      AX,DGROUP
            MOV      DS,AX
            CMP      START,FALSE
            JE       START_IS_FALSE
START_IS_TRUE:
            MOV      AL,SEND
            OR       AL,80H
            MOV      DX,P8255_C
            OUT      DX,AL
            MOV      BL,SEND
            MOV      BH,0
            MOV      AL,_buffer[BX]
            MOV      DX,P8255_A
```

```
            OUT     DX,AL
            MOV     AL,_buffer[BX+8]
            MOV     DX,P8255_B
            OUT     DX,AL
            INC     SEND
            AND     SEND,07H
            MOV     STOP,FALSE
            JMP     SEND_EOI
START_IS_FALSE:
            MOV     AL,0
            MOV     DX,P8255_C
            OUT     DX,AL
            MOV     STOP,TRUE
SEND_EOI:
            MOV     AL,END_OF_INT
            OUT     20H,AL
            POP     DS
            POP     DX
            POP     BX
            POP     AX
            IRET
INT9        ENDP
;
;
_enairq9    PROC    NEAR
            PUSH    DS
;
            PUSH    CS              ;DS:DX--->POINTER TO INT9
            POP     DS
            MOV     DX,OFFSET INT9
            MOV     AH,25H          ;DOS FUNCTION CALL
            MOV     AL,0AH          ;SET INTERRUPT VECTOR
            INT     21H
            POP     DS
;8255 MODE SETTING
;PORT A---OUTPUT (LOW BYTE DATA)
;PORT B---OUTPUT (HIGH BYTE DATA)
;PORT C---OUTPUT,PC7---START/STOP 8253 TIMER

            MOV     AL,80H
            MOV     DX,P8255_CTRL
            OUT     DX,AL
            MOV     AL,80H
            MOV     DX,P8255_C
```

```
                OUT     DX,AL
                MOV     STOP,FALSE
                MOV     START,TRUE
                MOV     AL,37H          ;8253 TIMER 0 MODE
                MOV     DX,P8253_CTRL
                OUT     DX,AL
                MOV     DX,P8253_0      ;LOAD COUNT VALUE
                MOV     AL,83H
                OUT     DX,AL
                MOV     AL,20H
                OUT     DX,AL
                RET
_enairq9        ENDP
;
;
;
_disirq9        PROC    NEAR
                MOV     START,FALSE
WAIT_FOR_STOP_IS_NOT_TRUE:
                CMP     STOP,TRUE
                JNE     WAIT_FOR_STOP_IS_NOT_TRUE
                MOV     AL,0
                MOV     DX,P8255_A
                OUT     DX,AL
                MOV     DX,P8255_B
                OUT     DX,AL
                RET
_disirq9        ENDP
_TEXT           ENDS
                END
```

第 9 章

驻留程序的设计

由于 PC DOS 是一个单任务单用户的操作系统，简单地说：就是一次仅能执行一个任务（一个程序）。由于在许多实际的应用程序设计中，往往希望能同时执行多个程序，比如在信号采集并实时显示的过程中，当发现有用信号希望立即将该信号存盘或打印出来，但信号采集程序希望仍在运行（仅在存盘或打印时暂停执行）。类似这样的问题可以采用中断的方法，如当用户通过键盘发送某一个信息，当 PC 机接收到这个信息后，立即让程序转去执行能完成这个任务的中断程序，中断完后，又回到原程序断点处去执行。又如许多作图程序，可在屏幕上作出各种图形来，我们希望根据需要将某屏幕图形保存或打印出来，但这个保存图形或打印图形的程序又不是作图程序的组成部分，只是按照用户的需要，才让它出现，执行其功能。这都可以通过驻留在内存的一个程序来实现，当然执行该程序是采用中断方式，即中断当时的程序，而去执行驻留在内存的这个程序。这种装入内存，执行后仍保存在内存，而且可以再次执行的程序，称为 TSR（Terminate and stay resident），即终止并驻留的程序。TSR 技术过去仅用宏汇编语言来实现，现用 C 语言也可以实现，人们采用这种技术愈来愈多（注意，当我们在 DOS 下调入某个可执行程序并执行后，又回到 DOS 下时，这个程序就在内存中消失了（被覆盖），要执行它，必须在 DOS 下再次调入，因而这些程序是终止非驻留的）。

在一般程序设计中，用户不必了解 DOS 对程序的管理，但若要编制出高水平的软件来，有必要了解它。尤其在编制驻留程序时，更需了解它，否则有可能使编制的程序无法使用。

每当 DOS 装载一个程序到内存并运行时，它都要为该程序建立一个长度为 100H 个字节的程序段前缀 PSP，然后根据程序的类型（EXE 或 COM）来对程序进行定位，如 COM 文件则和 PSP 为同一个段，代码放在 PSP 后，EXE 文件将 PSP 放在附加的一个特殊段内，开始时 DS、ES 指向该段，而代码则放在由 CS 指出的段开始处，然后 DOS 将控制权交给该程序。为此有必要了解程序段前缀 PSP（program segment prefix）。

9.1 几个特殊区域

9.1.1 程序段前缀 PSP 和 DTA

1. PSP

当一个可执行程序由 DOS 装入内存并执行，即执行一条相当于 DOS 的外部命令时，首先由 DOS 的 command.com 命令处理程序在内存构造一个 256 个字节的程序段前缀 PSP，然后才调入程序并开始执行，PSP 开始地址分配在可用内存空间的最低端，它是用来作为 DOS 和被装入的执行程序间相互通讯的一块内存区。在程序装入而未执行前，DS 寄存器中装的就是 PSP 的段地址，但在程序开始执行后，则由用户程序将 DS 设置为数据段地址，

PSP 结构如图 9.1 所示，各地址中的内容如下：(以偏移地址计)

0～01　装 INT 20H 指令，这条软中断指令的功能就是退出当前执行的程序，并返回 DOS，INT 20H 中断的服务程序是由 DOS 提供的，执行程序开始用指令(如前所述)将 PSP 的段值和偏移 0 入栈；程序执行完，遇到结束指令，便通过出栈操作，将原存 PSP 的段地址送到 CS，IP 取得 0 偏移，程序便转至 PSP 的开始处，此处恰好为 INT 20H 指令，于是执行该指令，程序便返回 DOS。

02～03　内存容量，以 16 字节为单位，它实际上指出的是自由可用内存空间的高端段地址。所以程序长度也可这样计算，即将此段地址中的内容(可用空间高端段址)减去 PSP 的段址，再乘 16，即为用户程序的长度(总的字节数)。

04～09　FAR JMP 到 DOS 的功能调用入口，在第一个字节存放远程调用指令(CALL)，后四个字节存放 INT 21H 的入口地址。

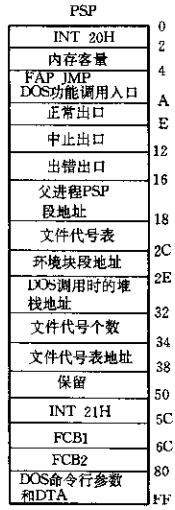


图 9.1　程序段前缀 PSP

0A～0D　程序结束时调用 INT 22H 中断向量。

0E～11　当 Ctrl-Break 或 Ctrl-C 时调用 INT 23H 中断向量。

12～15　当发生诸如读不存在的磁盘操作等严重错误时调用 INT 24H 中断向量。

16～17　父进程的 PSP 段地址。

18～2B　文件代号(文件句柄)表，它可以存放 20 个文件代号，因此在用户程序中同时可打开的文件不能超过 20 个，由于用户程序一旦运行，必然自动打开 5 个标准文件，所以实际用户可以打开 15 个文件。

2C～2D　程序环境块段地址，DOS 给每个装入的程序都建立一个程序环境块，所以通过这两个地址，可访问程序环境块。

2E～31　DOS 调用时的堆栈地址，当用户程序进行 DOS 调用时，DOS 将用户的堆栈地址和指针保存在此段地址中，不许改写。

32～33　文件代号的最大个数，即用户程序可以使用的最大文件数，通常也为 20，但可以改写，使其大于 20，但此时必须重新建立文件代号表。

34～37　表示存放各文件代号表的段地址和偏移地址，若用户要建立自己的文件代号表，必须要修改这些地址的内容，以表示用户建立的文件代号表的段地址和偏移。

38～4F　保留

50～52　这段地址含有 INT 21H 和 RETF 指令，供 DOS 功能调用和返回用。

5C～6B　文件控制块 1(FCB1)，当对文件进行存取时(不是文件代号式的存取方式)，必须首先将存取的文件名放到一个指定的存储区——文件控制块 FCB，在此块中，还存放磁盘驱动器号、当前块号，记录长度，即当前要读写的记录号等。

6C～7B　文件控制块 2(FCB2)。

当使用文件代号式存取方式时，这两个文件控制块(FCB1,FCB2)将不用。

80～FF　命令行参数和 DTA。当 DOS 以外部命令形式装入用户程序时，80H 处装命令行长度，81～FFH 装命令行的参数。当要进行磁盘操作时，这一段地址又作为程序和磁盘之间进行文件传送时的数据传输区(DTA—DISK Transfer Area)。所以当要在磁盘操作后需要使用命令行参数时，则应在磁盘操作前，将该段地址的内容保存起来。

2. DTA

数据传输区 DTA 是程序和磁盘之间进行文件传送时必须开辟的数据暂存区。写文件时，程序将文件记录送到 DTA，然后再由 DTA 送到磁盘。读文件时，将文件记录由磁盘读到 DTA，再由 DTA 送到程序，由程序进行处理。由于顺序或随机对文件进行存取时每次只传一个记录，因而 DTA 长度为一个记录长度，即 128 个字节(缺省值)，当由用户程序进行设置时，不一定为此长度。

9.1.2　DOS 环境块

当 DOS 执行外部命令时，它总是到当前目录下去找可执行文件，但可用 DOS 命令如 PATH 改变它，使它到指定路径去找，这就是说可以人为地改变 DOS 的当前执行环境。在 MS-DOS 中将那些可以改变程序的开发与执行的变量称为环境变量，即它们的不同设置，可以构成 DOS 不同的工作环境，通常 DOS 在内存中开辟了一块区域，用于存放这些环境信息，环境信息由环境字符串组成，环境字符串的结构是：

环境变量名＝环境变量值

各串之间用 00H 分隔，而在最后的环境串后用两个 00H 结尾，它表示环境信息到此结束。

用户可以自己定义、修改或删除环境变量，DOS 定义有三个标准的环境变量，它们是：COMSPEC、PATH 和 PROMPT。

COMSPEC(COMMAND SPECIFICATION)，用于设置 DOS 的 COMMAND.COM 命令处理程序所在的驱动器、路径、文件名和扩展名，当 DOS 启动时，该变量被赋值为“x:\COMMAND.COM”，其中 x 表示启动盘符，如 A 盘或 C 盘。

PATH 环境变量用于引导 DOS 到哪些目录中去找后缀为 COM、EXE 或 BAT 的文件，当输入 DOS 命令时，DOS 首先检查是内部命令还是外部命令，当是外部命令时，则按 PATH 指出的路径依次去查找该命令文件。

PROMPT 环境变量用于设置 DOS 的系统提示符，它的参数可为一个或多个按规定定义的组合字符串，因而系统提示符在屏幕上出现时，可有许多式样。

环境变量有专门的 DOS 命令可以设置，如 SET 命令等。具体的命令可参阅有关的 DOS 手册。

除了这些环境变量外，用户也可自己进行设置和删除环境变量。

在 DOS 下，DOS 被引导时，首先对 COMMAND.COM 建立一个环境块，称为主环境块。装载运行程序时，DOS 将为该运行程序建立一个环境块，该环境块是主环境块的副本，并可添加一些本程序使用的环境信息，若该程序运行时，发生程序嵌套，调入嵌套的内层程序时，DOS 又为该层程序设置环境块，该环境块又是上层程序环境块的继承(副本)，并可添加本层程序的一些环境信息，所以外层环境块被修改时，将会传给嵌套的内层环境块，而内

层环境块的某些改变，将不会影响到外层环境块。

例如，在对 Turbo C 进行安装时，已经对编辑、编译及连接程序要求的环境变量进行了设置，并对源文件、目标文件和可执行文件在什么目录下进行存取，都确定了缺省值，这就是 TC 的工作环境。它们保存在 TCCONFIG.TC 文件中，这样每次启动 TC 时，它总是执行 TCCONFIG.TC，将 TC 的工作环境确定下来，这样我们在开发程序时，include.lib 等就可从确定的路径去找。

程序环境块的段地址和偏移在该程序 PSP 位移＋2CH 处。一般来说程序环境块长度是不固定的，但对驻留程序的环境块只允许 128 个字节长，它应不会受任何添加或修改环境信息的影响，因此在 TSR 程序驻留前，应将先前建立的程序块释放掉，这样也可节省内存空间。

9.2 TSR 程序设计

TSR 程序一般按照进驻内存方式，分为两大部分，即实现 TSR 中断服务功能的常驻部分和用于加载常驻部分的暂住部分。当 TSR 程序在 DOS 下装载时，实际上将运行暂住部分，由这部分将 TSR 的执行部分装入内存，至此 TSR 的加载便完成，且回到 DOS 下。这时 TSR 常驻部分只是驻留在内存，当加载别的程序时，不会覆盖掉它，而它还没有得到控制权，即还不能使机器的执行依照它的执行代码去执行。只有当激活它时，即令机器的执行进入 TSR 常驻部分时，常驻部分才能执行，即得到对机器的控制权。

要设计 TSR 程序，从程序结构上来说一般应包括三个部分。第一是初始化部分，这部分程序主要完成将 TSR 程序的入口地址放入中断向量表中相应的中断号处，以便当用户用某个键执行该 TSR 程序时，便产生相应中断号的中断，DOS 在向量表中取出该程序入口地址，而去执行 TSR 程序。

第二部分是执行功能部分。该部分是 TSR 程序要完成特定功能的程序，或称中断服务程序部分（它实际上就是常驻部分）。一般应包括一个弹出窗口，进行提示说明，当得到用户响应后即去执行，并最后恢复屏幕的原来状态。

第三部分，将 TSR 程序按给定的长度（占内存的字节数）装入内存，并能保护它，不被别的程序破坏，即所谓终止并驻留内存，这可通过 DOS 或 BIOS 调用来实现，也可用 keep()函数。

第二部分编程的几个要点必须要注意，否则将导致该程序不能执行，即在功能程序部分，由于执行时实际上中断了原来的程序执行，即它的运行环境暂时脱离了原来程序的环境，这有可能涉及到 DOS 的重入问题。我们知道 DOS 是不可重入的，因为在执行 DOS 系统调用时，DOS 把一些必要的信息存在它的全局变量中，如果在执行过程中又被中断打断，而中断程序中又有 DOS 的系统调用，这样原全局变量内容就被改写，因而程序的执行就被破坏。如在执行 DOS INT21H 系统调用时，又被用户定义的某中断程序打断，而用户的中断程序中又进行了 INT 21H 系统调用，如同进行了递归调用 INT 21H 一样，这样原 INT 21H 系统调用的工作数据因未保存而导致破坏，因而将回不到原先断点时的状态去，导致系统死机。因而这部分程序中不应有涉及 DOS 系统调用的一些 Turbo C 函数，如 malloc()，printf()，sprintf()等。另外还有一个 TSR 程序重入的问题，即当我们按下热键，执行驻留在内存

的 TRS 程序过程中，若又按下热键，则应不加理睬，否则在执行 TRS 的过程中又执行 TSR 程序，形成递归，这将导致程序的破坏。这可通过在该程序部分设立标志来判别是否重入了，若是，则不执行这次的中断调用。

关于 DOS 的重入问题，可以这样理解，当 DOS 系统调用时，将使用规定的系统专用区（系统堆栈），如 INT 21H 的输入输出功能调用时，将使用 I/O 堆栈（功能号为 1～0C），当使用有关磁盘的操作的功能调用时，将使用磁盘堆栈，这样当系统正在某堆栈工作时，突然被另一中断打断，原来工作的数据没有保存，因而将回不到原来状态，造成系统工作环境破坏。

关于解决 DOS 重入的一般方法，将在关于 DOS 重入问题的解决方法一节中详加论述。

9.2.1 TSR 的中断服务部分

这部分程序应定义为 interrupt 类型的函数，这样一旦调用该函数时，它会自动保留中断时各寄存器的内容，当中断完成后返回时，自动恢复原来的值。另外为了解决该程序的重入问题，可以用设置一个全局变量（如 busy）来解决。设其初始值为 0，当进入该程序时，将其变为 1，该程序结束时，再将其恢复为 0，这样，进入该程序前，首先检查 busy 变量，若为 1，表示程序已在执行，不能再第二次执行，如可以这样编写：

```
void interrupt far tsr_ap()
  {
    if (! busy)
      {
        busy=1;
        中断服务部分
              :
        busy=0;
      }
  }
```

当然除了防止程序重入的问题外，还应考虑 DOS 重入的问题，即若将这两个问题都进行考虑，应有这样的内容：

```
if(TSR 程序未激活 &&DOS 没有重入 &&.有激活键按下)
      中断服务部分
          :
```

关于如何判别 DOS 没有重入，将在下面进行详细讨论。

9.2.2 程序的驻留

这可以通过 BIOS 或 DOS 功能调用来实现，如可用 DOS 中断 0x21，功能号为 49，设定义驻留函数 res()

```
res(size)
unsigned size;
  {
    union REGS rg;
```

```
    rg.h.ah=49;
    rg.h.al=0;
    rg.x.dx=size;
    int 86(0x21,&rg,&rg);
}
```

其中size表示要驻留的程序长度，单位为节，一节代表16个字节。用高级语言编的程序要计算出所占字节数一般很难，且其执行文件装入内存时并不分配在一个连续的内存空间，因而简直无法计算。但可以估计，如已知其.EXE文件所占字节数(可看目录)，用16去除它，然后将其值加倍，这个数字可以做为驻留程序的长度。

也可用Turbo C的库函数keep()来实现程序驻留，该函数的说明原型为：

Void Keep(int status,int size);

此函数将把当前程序驻留在内存中，驻留内存的长度为size个节(每节等于16个字节)。当该函数执行后，出口状态信息保存在status中，它实际上也是执行了DOS 21H的49号功能调用。

DOS INT 27H调用也可实现程序驻留，调用时DX装程序的末地址，而CS装程序段前缀(PSP)的地址，它仅限小于64K的驻留程序，功能号为49的INT 21H调用，则不局限于64K的长度。

程序长度除了用上面所述的估计法来估算外，也可采用程序开始地址和可能结束地址的差来求得。_PSP全局变量是程序段前缀的段地址，PSP可看作程序的一部分，故视作头，而程序若将使用的堆栈也包括进去，在小内存模式下，则堆栈底可看作驻留程序的结束，故驻留长度(以节为单位)应为：

size=_ss+_sp/16-_psp

其中_ss为变量存放堆栈段地址，_sp为堆栈指针，若已定义了堆栈长度，设为stack_size，则应为size=_ss+_(sp+stack_size)>>4-_psp；当然上述计算也是一种估算，永远得不出精确的结果。

还有一种方法是，由于PSP的偏移地址为2处，记录了该程序装入后，剩余内存空间的开始段地址，因此用它和PSP的开始地址相减，也可得到程序驻留的长度。

9.2.3 关于DOS重入问题的解决方法

我们可以利用系统运行时，不可能发生DOS重入问题的时间间隔来激活驻留程序，可分几种特殊情况。

1. DOS不忙时，即没有发生DOS系统调用时，激活驻留程序是安全的。怎样得知这个时间呢？这可以用DOS的INT 21H的34H功能调用来得到。当执行该调用时，便得到一个指针ES：BX，它指向一个忙或闲标志单元，当其值为1时，表示DOS处于忙状态，即正在执行系统调用。若为0，则表示空闲，因而这时若调用驻留程序是安全的。

2. 当DOS处于等待状态，即处于等待用户输入命令状态(如：C>提示符下)，这时若想通过热键激活驻留程序，应该说是安全的。但此时采用查询DOS是否忙时，仍得出忙的标志，表示不能调用驻留程序，这是因为在DOS等待状态下，在不停的调用INT 28H，因此若用INT 21H的34H功能调用来查询，它总是处于忙状态，但实际上INT 28H调用不进行任

何功能操作，是一条空调用，因而当我们检测到是 INT 28H 系统调用时，即使 DOS 34H 功能调用得到忙标志，此时若有热键按下，也可立即激活驻留程序，这时是安全的。

3. 当进行磁盘操作时，若突然激活驻留程序，这意味着要突然停止磁头的寻道、读、写等操作，这将发生磁盘错，由于磁盘操作时，总是执行 INT 13H 调用，因此我们可以改写这个中断调用程序，即在该中断调用程序中，设立一标志，当调用该中断时，该标志被置 1，调用完后再清 0。因而在有热键按下时，我们可以检查该标志，若为 0，则可激活驻留程序，这样就可保证有磁盘操作功能调用时，只有该功能执行完后，才执行驻留程序。即保持了安全。

4. 系统总是以每秒 18.2 次的频率在进行定时中断(INT 8)，该中断又调用功能号为 1CH 的系统调用，而该调用又是一条空操作调用，什么事也没作，因而当我们得知在进行 INT 8 调用时，此时有热键按下，便可激活驻留程序。

利用每秒 18.2 次的硬时钟中断也可作为激活驻留程序的信号。例如在信号采集程序中，可利用它定时采样。采样程序驻留在内存。又比如在需要时间的场合，在运行用户程序的时候，定时的在屏幕上显示当时的时间，这个计时程序可驻留在内存，它们都是用 INT 8 中断定时激活驻留程序，且运行驻留程序时无需考虑 DOS 的重入问题，当然它总是安全的。不过一般使用中，多用热键，由用户随机地激活驻留程序。

5. 当 DOS 遇到紧急错误时，如读写软磁盘时驱动器的门没有关闭，DOS 检测到这种错误时，便调用紧急错误中断 INT 24H，此时屏上便出现 Abort，Retry or Ingore 信息，若在执行程序时又激活 TSR 程序，此时，若出现 INT 24H 调用，当用户用 A 回答后，DOS 将放弃 TSR，恢复到原先执行的程序中去，此时由于原先执行程序的 PSP 还未恢复，还使用的是 TSR 的 PSP，故无法正确返回到原来的执行程序中去。因此在 TSR 程序中可以检测 INT 24H 中断，当发生该中断时，一般仅在 AL 中设置一个返回信息，除此之外，再不作任何操作，如可在 AL 中设置为 03H，它表示放弃失败的功能调用，这样等于 INT 24H 中断的主要功能被屏蔽，这实际上就是将 INT 24H 中断进行改写(用新的中断向量代替旧的)。

6. 当在 TSR 程序运行中按了 CTRL-C 键，DOS 检测到后，执行 INT 23H 中断调用，通常是中断程序的运行，而把控制权交给 command.com，这样 TSR 的中断向量将得不到恢复，出现系统错误。为了防止 CTRL-C 中止 TSR 程序，可在 TSR 程序中设置新的 INT 23H 中断程序，一旦发生 INT 23H 中断，便什么事也不作。这样可避免出现由于 INT 23H 中断造成系统错误。

7. 当在 TSR 程序运行时，按了 CTRL-BREAK 键，将产生 INT 1BH 中断，这也将发生如 CTRL-C 那样的错误。为此在 TSR 程序中也可修改 INT IBH 中断，一旦发生 INT 1BH 中断，也什么事不作。

9.2.4 TSR 程序设计中另外的几个问题

1. 关于 PSP

DOS 是通过当前程序的段前缀 PSP 来和当前程序进行通讯的。当 TSR 被激活而中断了当前程序时，TSR 的 PSP 并未被置为当前的 PSP，这样当 TSR 程序中想使用 DOS 的文件管理或内存申请功能时，DOS 将认为是原先被中断程序的，而 TSR 中的许多函数执行也将被 DOS 认为是被中断程序的函数的执行，因而对原来的程序将产生影响，为此必须用 TSR 的 PSP 替换掉原执行程序的 PSP，这可用 INT 21H 的 51H 号功能调用，即 AH=51H，

令 BX=PSP 的段地址(TSR 的),进行 INT 21H 调用后,即将 TSR 的 PSP 变为当前的,这样执行 TSR 中的函数时,将不会影响原先的执行程序,TSR 中使用 DOS 的文件管理和内存申请功能将不会出现错误,当然 TSR 结束后,PSP 又要换回。

2. 关于 DAT

由于 PSP 的后 128 个字节是缺省的磁盘数据交换区,因而 TSR 也应有自己的 DTA,故 PSP 变换时,DTA 也需变换,但由于用文件代号式读写磁盘文件时,并不用这个 DTA,故此时 DTA 变换无甚作用。若实行了变换,TSR 运行结束后,应恢复原来的 DTA。

3. 关于堆栈

当由于 TSR 被激活而中断了原来的执行程序时,此时堆栈段寄存器的堆栈指针仍指向被中断程序所使用的堆栈,因此 TSR 使用堆栈时,仍被指向该堆栈,这可能会由于栈不够用而溢出,因而 TSR 必须设置自己的堆栈,并保存原先的栈地址,以便 TSR 运行结束时,恢复成原先的。由于 C 语言中直接控制各寄存器比较困难,因而可用汇编语言来实现。

4. 被中断程序扩展错误信息的恢复

当被中断程序由于调用 DOS 而出现失败时,该错误信息应被记载,所以当 TSR 程序激活时,该信息应被保留,当 TSR 程序结束后,应将该错误信息送回。这可在执行 TSR 时,用 Turbo C 的 dosexterr()函数将错误信息保留,而在 TSR 程序结束时,用 DOS INT 21H 的 59H 功能调用,将错误信息恢复,以便 DS:DX(段地址:偏移地址)指向存放错误信息一个结构,从而可得到它。

9.2.5 几个有关的库函数说明

在 TSR 程序中,有可能遇到几个以前未曾介绍过的函数,现特作以下介绍:

1. 得到程序段前缀 PSP 的函数 getpsp()

该函数将返回当前进程 PSP 的段地址,所谓当前进程是指当前由 DOS 控制的正在运行的程序,该函数说明为:

```
unsigned getpsp (void);
```

2. 得到磁盘传输区 DTA 地址的函数 getdta()

该函数将得到指向 DTA 的指针,在小型和中型编译模式时,DTA 的段地址在 DS 中。该函数说明为:

```
void getdta(char far *dta);
```

3. 设置磁盘传输区 DTA 的函数 setdta()

该函数将设置一个 DTA,其地址由该函数的参数指针指出,原来的 DTA 将被放弃。该函数说明为:

```
void setdta(char far *dta);
```

4. 得到错误信息的函数 dosexterr()

该函数的说明为:

```
int dosexterr(struct DOSERR *dblkp);
```

当 DOS 功能调用失败后,该函数将在由 dblkp 指向的 DOSERR 结构中,填入扩展的错误信息,DOSERR 结构如下:

```
struct DOSERR{
```

```
    int exterror;          /* 扩展错误信息代码 */
    char class;            /* 错误类别 */
    char action;           /* 建议的行动 */
    char locus;            /* 错误位置 */
}
```

当 DOS 调用未出错时，exterror 的值为 0，该函数的返回值为 exterror 的值。

9.3 用户激活驻留程序 TSR 的方法

用户激活 TSR 程序的方法有多种，如上述用 INT 8 定时中断，它每秒钟产生 18.2 次中断，每次中断又调用功能号为 1CH 的系统调用，该调用为一空调用，用户可以改写它，使之激活 TSR 程序。

也可用 INT 28H 中断调用，当 DOS 处于等待状态时，如在屏幕显示 C＞提示符下，DOS 在不停地进行 INT 28H 功能调用，它也是一条空调用，因而可在这时激活 TSR 程序。

但上述两种方法均有局限性，即无法由用户控制激活的时间，所以更一般的方法是采用按某特定键并和上面出现的功能调用相与，即当两种条件均出现时，便激活 TSR 程序。

用户按特定键激活 TSR 程序时，这特定键称为热键(hot key)。如最简单的方法是用 Print-Screen 键(即屏幕打印键)，不过要改写由于按该键而引起的屏幕打印中断调用(即 INT 5)，即要用用户的驻留程序去置换原来的屏幕打印中断程序，这只要将按该键引起的 INT 5 的中断向量指向用户的驻留程序即可。我们知道中断 5 的向量地址在 5 * 4＝20 处，设用户自己的中断服务程序名为 tsr-ap，则主程序可以如下编制：

```
void interrupt tsr-ap();
  main()
  { struct address{
    char far *p;
  };
  struct address far *addr=(struct address far *)20;
  addr->=(char far *)tsr_ap;
   ⋮
```

即定义 addr 为一结构指针且指向 20，它就是存放中断 5 向量的地址，该向量又指向 trs. ap()的入口地址，这样就将原来中断 5 的中断向量指向的地址改为用户自己的中断服务程序地址了，因而一旦按 Print-Screen 键后，并不执行原来的屏幕打印中断程序，而去执行用户的中断程序 tsr-ap()。

当然利用 Turbo C 提供的设置中断向量的函数 setvect(inum,faddr)也可以，如：

```
void interrupt tsr-ap();
main()
  {
    setvect(0x5,tsr-ap);
          ⋮
```

利用按 Print_Screen 键的方法来激活 TSR 程序虽然比较简单，因为原来的驻留程序(打印屏幕程序)已解决了 DOS 重入问题，但用该种方法来激活 TSR 驻留程序有几个缺点，即系统中只能允许一个驻留程序，但需要实现多个功能就要有多个驻留程序，该方法就不能用了；再者，系统原来的屏幕打印功能就不能用了，对于别的用户，由于用惯了原来该键的屏幕打印功能，再改用该键完成其它功能，就觉得不习惯，甚至感到莫明其妙，以为该键出了毛病。所以更一般的情况是利用 INT 9 中断，因为按键时，均产生中断 9。

当采用 INT 9 来激活驻留程序时，要对 INT 9 中断程序进行改写，即要保留原来的 ITN 9 中断驻留程序，以便接收按下键的扫描码。要增加对按下键的判别，如常用 Fi 键(i 为 1～12)作为热键，这样当对按下的键码进行判别，并确认为热键时，便可设置标志，从而激活相应的驻留程序。为了更好地使用热键，了解 INT 9 中断，必须对键盘缓冲区和识别键的扫描码有明确的认识。下面将进行介绍。

还有一种激活驻留程序的方法，就是采用鼠标器上的按键。由于 DOS 操作系统本身并不支持鼠标的使用，但用户一旦购置了鼠标器，便带有鼠标驱动程序，使用鼠标前可以安装该程序，并对它进行初始化，这样就可以打开鼠标，用户操作鼠标器的各种动作就由鼠标驱动程序来管理。我们可以通过 INT 33H 的 12 号功能调用(它由 Microsoft 公司鼠标驱动程序提供)，来设置利用鼠标调用的中断服务程序，这样一旦设置的条件出现，比如某钮按下，驻留的服务程序即被激活而执行。由于 Turbo C 2.0 没有提供与鼠标接口的函数，因而可以通过 INT 33H 调用来实现与鼠标的接口，有关参数分别放在 ax，bx，cx，dx 寄存器中，后面将要专门介绍。

上面介绍的用热键的方法中，键盘或鼠标器仅是一个激活 TSR 的动作，是否能激活，还要判断 DOS 重入与否，当这些条件均满足时，才真正激活 TSR。这样既可保证 TSR 的正确执行，又可保证不破坏系统工作的环境。所以最完整的方法是，设置几个标志，使得当按下热键时，恰好产生 INT 8 中断，便可激活 TSR，或当按下热键时，恰好为 INT 28H 中断，便可激活 TSR。再要判别 DOS 此时是否处于忙状态，可用 INT 21H 的 34H 功能调用来得到 DOS 的安全标志。当 DOS 不忙时，即安全时，则可激活 TSR。另外还要判断是否正在进行磁盘读写操作，即是否在进行 INT 13H 中断。若没有，可激活 TSR。另外还要判断的是：DOS 是否正在处理遇到的紧急错误，如磁盘驱动器的门未关，DOS 会调用 INT 24H 中断来处理这种错误，因此当没有 INT 24H 中断，且有上述条件满足时，便在热键按下时激活 TSR，所以一个完善的 TSR 程序被激活的条件，应包括上述各种条件，即要考虑到各种可能造成 DOS 重入的情况。例如当使用定时中断 INT 8H 来激活 TSR 时，可以改写 INT 8H 中断服务程序，使该程序除了保存原有的中断程序的内容外，还增加是否激活 TSR 程序的部分，新的中断程序可写成如下的内容：

```
void interrupt far newint 8(void)
  {
    (*oldint8)();                    /*执行原来的 int8 程序*/
    if (TSR 未被激活 && 热键已按 && DOS 不忙 && DOS 没有处理致命错误 && 没有执行 INT
       13H 中断)
      {
        激活 TSR 部分
      }
```

当产生 INT 28H 中断时，激活 TSR 可改写为：

```
void interrpt far newint28(void)
  {
    if (TSR 未被激活 && 热键已按 && DOS 没有处理致命错误 && 没有执行 INT 13H 中断)
      {
        激活 TSR 部分
      }
      (*oldint28)();
  }
```

关于这方面的示例程序，将在 9.6 中进行介绍。一般较简单，但也可以用的(即仅考虑到几种 DOS 重入可能性)例子也将作介绍。

9.4 键盘编码

当我们按下键盘上某键时，系统如何知道某键被按下呢？它的奥妙在于在键盘内有一个微处理器，它用来扫描和检测每个键的按下和抬起状态。当检测到某键被按下或松开时，就产生一个中断请求信号(IRQ1)，即 INT 9。然后输出一个字节的扫描码给系统，扫描码的 0～6 位标识了每个键在键盘上的位置，而其最高位(7 位)标识了对应该键是被按下(若为 0)或松开(为 1)，即当时状态，各键扫描码如表 9.2 所列，一些特殊键如 Print-Screer 等将不产生扫描码，而直接引起中断调用。

由于扫描码仅能区别键的位置和键的按下与松开，它并不能区别大小写字母(因它们用同一个键表示，如 A 和 a 均为同一个键)，因而每当某键按下或松开时，便产生 INT 9 中断，以调用 ROM 中 BIOS 中的键盘中断处理程序，它的作用是将得到的键扫描码(当第七位为 0 时)翻译成对应的 ASCII 码，这实际上使用了一个扫描码→ASCII 码的对照表，至于字母是大小写或上键、下符，则是参照 Caps Lock 键的状态来进行转换的。

由于 ASCII 码仅能有 256 个(2^7)，它不能将 PC 键盘上的键全部包括，因此有些控制键如 CTRL，ALT，END，HOME，DEL…等用扩充的 ASCII 码表示，扩充码用两个字节的数表示。第一个字节是 0，第二个字节是 0～255 的数，键盘中断处理程序将把转换后的扩充码存放在 AX 寄存器中，存放格式如表 9.1 所示：

表 9.1 键盘扫描码

键名	AH	AL
字符键	扩充码=ASCII 码	ASCII 码
功能键/组合键	扩充码	0

对字符键，其扩充码就是其 ASCII 码。

表 9.2 键盘标准扫描码

扫描码	基本键	Shift	Ctrl	Alt
29H	`	~	无定义	无定义
2H	1	!	无定义	扩充码
3H	2	@	扩充码	扩充码
4H	3	#	无定义	扩充码
5H	4	$	无定义	扩充码
6H	5	%	无定义	扩充码
7H	6	^	RS	扩充码
8H	7	&	无定义	扩充码
9H	8	*	无定义	扩充码
AH	9	(	无定义	扩充码
BH	0	)	无定义	扩充码
CH	-	_	US	扩充码
DH	=	+	无定义	扩充码
2BH	\	|	FS	无定义
EH	BackSpace	BackSpace	DEL	无定义
FH	Tab	扩充码	无定义	无定义
10H	q	Q	DC1	扩充码
11H	w	W	ETB	扩充码
12H	e	E	ENQ	扩充码
13H	r	R	DC2	扩充码
14H	t	T	DC4	扩充码
15H	y	Y	EM	扩充码
16H	u	U	NAK	扩充码
17H	i	I	HT	扩充码
18H	o	O	SI	扩充码
19H	p	P	DLE	扩充码
1AH	[	{	Esc	扩充码
1BH	]	}	GS	无定义
1DH	无定义(Ctrl)	无定义	无定义	无定义
1EH	a	A	SOH	扩充码
1FH	s	S	DC3	扩充码

续表

扫描码	基本键	Shift	Ctrl	Alt
20H	d	D	EOT	扩充码
21H	f	F	ACK	扩充码
22H	g	G	BEL	扩充码
23H	h	H	BS	扩充码
24H	j	J	LF	扩充码
25H	k	K	VT	扩充码
26H	l	L	FF	扩充码
27H	;	:	无定义	无定义
28H	'	"	无定义	无定义
1CH	CR(Enter)	CR	LF	无定义
2AH	无定义(Left Shift)	无定义	无定义	无定义
2CH	z	Z	SUB	扩充码
2DH	x	X	CAN	扩充码
2EH	c	C	ETX	扩充码
2FH	v	V	SYN	扩充码
30H	b	B	STX	扩充码
31H	n	N	SO	扩充码
32H	m	M	CR	扩充码
33H	,	<	无定义	无定义
34H	.	>	无定义	无定义
35H	/	?	无定义	无定义
36H	无定义(Right Shift)	无定义	无定义	无定义
38H	无定义(Alt)	无定义	无定义	无定义
39H	SP(Space)	SP	SP	SP
3AH	无定义(Caps Lock)	无定义	无定义	无定义
3CH	扩充码(F2)	扩充码	扩充码	扩充码
3EH	扩充码(F4)	扩充码	扩充码	扩充码
0H	扩充码(F6)	扩充码	扩充码	扩充码
42H	扩充码(F8)	扩充码	扩充码	扩充码
44H	扩充码(F10)	扩充码	扩充码	扩充码
3BH	扩充码(F1)	扩充码	扩充码	扩充码

续表

扫描码	基本键	Shift	Ctrl	Alt
3DH	扩充码(F3)	扩充码	扩充码	扩充码
3FH	扩充码(F5)	扩充码	扩充码	扩充码
41H	扩充码(F7)	扩充码	扩充码	扩充码
43H	扩充码(F9)	扩充码	扩充码	扩充码
1H	Esc	Esc	Esc	无定义
45H	无定义(Num Lock)	无定义	特殊组合键	无定义
46H	无定义(Scroll)	无定义	特殊组合键	无定义
54H	特殊组合键(Sys Req)	无定义	无定义	无定义
37H	*	特殊组合键	扩充码	无定义

由表 9.2 可看出，组合键 shift_d 表示 D，其扫描码与 d 的同，可以看出并非对于任意组合键 BIOS 都承认。

有些组合键是专门用于执行特殊功能的，它们将不产生扫描码：

(1) Alt-Ctrl-Del：系统热启动。

(2) Ctrl-Break：产生 INT 1BH 中断。

(3) Ctrl-NumLock：使系统处于暂停一项操作的状态，按除本键之外的任意键恢复。

9.5 键盘缓冲区

由于某键按下产生中断和当时运行的程序接收键盘输入不能同时进行，即是异步工作的。因键盘输入有可能随时产生，但当时运行的程序不能马上将其进行的工作停下来，它必须要保存好当时现场信息，才去接收键盘输入，但键盘输入的信息是即刻要消失的，因而 PC 系统在内存确定的位置定义了一个可存 15 个键的扩充 ASCII 码的 32 个单元的键盘缓冲区(KB_BUFFER)，有效使用的是 30 个单元，每个键扩充码用 2 个单元，当存满时，若再存，扬声器将会响，键盘缓冲区在内存的分配图和对应指针如图 9.2 所示。键盘缓冲区实际上是一个先进先出的循环队列，它使用两个指针 BUFFER1 和 BUFFER2 来指使队列中的数据，前者指向开始的数据，后者指向结束的数据，当键入的键扫描码被中断程序转换成扩充 ASCII 码

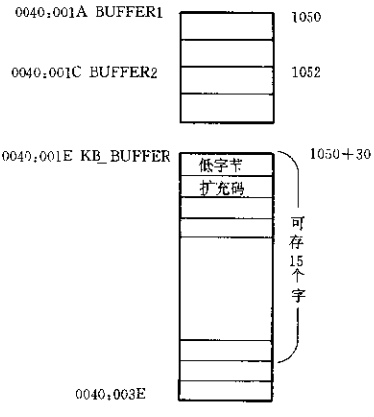


图 9.2 键盘缓冲区

后，就放入由 BUFFER2 指出的两个相邻单元内，BUFFER2 中的值接着增 2，以指向接着要接收的扩充码存放地址，当队列中由 BUFFER1 指向的数据被取走后，则 * BUFFER1 又增 2，指向下一个要取走的数据，当 * BUFFER1 = = * BUFFER2 时，表明队列中无数据了，即键盘缓冲区空了，关于键盘缓冲区循环队列操作如图 9.3 所示。图中标记 0，1，2，…是两个字节为一单位。X 表示 X 的扩充扫描码。

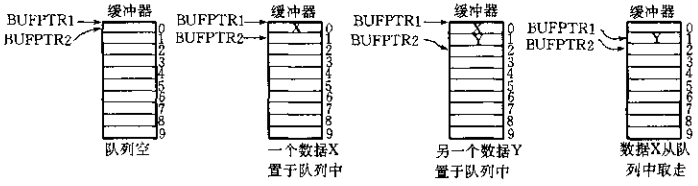


图 9.3　循环队列

从键盘缓冲区在内存的分配图来看，BUFFER1 指针值存于 0040：001A 地址处，即十进制 1050 处；BUFFER2 指针值在 0040：001C 地址处，即十进制 1052 处。由于 BUFFER1 和 BUFFER2 中的值实际上是指的循环队列中数的地址，因而若要变成存放扩充码的地址序号，必须要减 30，即减去键盘缓冲区首址的偏移地址 001E，这样，就可用 BUFFER1 指针变量来代表键盘缓冲区的数据地址。如我们定义 F1 为热键，用它来激活驻留程序，判断是否有 F1 热键按下，可定义 t1 和 t2 两个数据指针，即相当(BUFFER 1 和 BUFFER 2)，可如下编程：

```
        ⋮
char far  * t1=(char far * ) 1050;      /* BUFFER 1 指针 */
char far  * t2=(char far * )1052;      /* BUFFER 2 指针 */
  ( * old  key)();调用原键盘中断程序(即 INT9);
  if ( * t1!  = * t2)                  /* 表示键盘缓冲区不空，即有键按下 */
    {
      t1+= * t1-30+5                   /* 键盘缓冲区第二个单元指针，该单元存扩充码 */
      if( * t1==59)                    /* 表示有 F1 热键按下 */
        key=1                          /* 设置有 F1 键按下标志 */
       * (t2+1)= * t2                  /* 清键盘缓冲区 */
    }
        ⋮
```

上面编程示例中，清键盘缓冲区目的是将热键扩充码清除，以防键盘缓冲区的热键扩充码被系统取走。

9.6　键盘操作函数 bioskey()

是否有键按下，何键按下，也可采用 Turbo C 提供的键盘操作函数 bioskey()来识别，该

函数说明原型为：

int bioskey (int cmd)；

它在 bios.h 头文件中进行了说明，该函数实际上调用了 BIOS 的 INT 16H 中断，参数 cmd 用来确定 bioskey()如何操作：

cmd　操作

0　返回按键的键值，该值是 2 个字节的整型数，若没有键按下，则该函数一直等待，直到有键按下。当按下时，若返回值的低 8 位为非零，则表示为普通键，其值代表该键的 ASCII 码。若返回值的低 8 位为 0，则高 8 位表示为扩展的 ASCII 码，表示按下的是特殊功能键。

1　此时，该函数则用来查询是否有键按下。若返回非 0 值，则表示有键按下，若为 0，表示没键按下。

2　此时，该函数将返回一些控制键是否被按过，按过的状态由该函数返回的低 8 位的各位值来表示：

字节位	对应的 16 进制数	含　义
0	0x01	右边的 shift 键被按下
1	0x02	左边的 shift 键被按下
2	0x04	Ctrl 键被按下
3	0x08	Alt 键被按下
4	0x10	Scroll Lock 已打开
5	0x20	Num Lock 已打开
6	0x40	Caps Lock 已打开
7	0x80	Inset 已打开

当某位为 1 时，表示相应的键已按，或相应的控制功能已有效，如选参数 cmd 为 2，若有：

key=bioskey(2)；

若 key 值为 0x09，则表示右边的 shift 键被按，同时又按了 Alt 键。

下面是一个利用该函数来判断是否有热键按下，从而激活 TSR 的中断程序，即新的 INT 9 程序。由于该中断程序用到了有关键盘硬件接口电路，故作以下简介：

PC 机键盘接口电路原理图如图 9.4 所示，它由一个串变并电路 U_1 和一个中断请求触发器 U_2、控制电路 U_3 和并行接口电路 U_4 组成，从键盘得到的数据是依扫描得到的一位值，当扫描 8 次后，便得到一个扫描码，因此串变并电路就是将依次串行得到的 1 位值经过移位

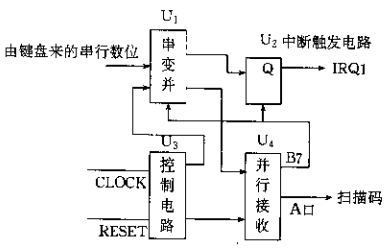


图 9.4　键盘接口电路

变成一个8位并行的数据送U_4的A口(它实际是一个8255电路),因而可从A口得到8位扫描码(A口的口地址为0x60)。当接收8位扫描码后,中断触发器U_1的Q端变高,即产生IRQ_1硬中断,这时中断服务程序便可将扫描码取走,即可用intportb(0x60)进行读取,读取后应将中断触发器U_2清零,并将串变并电路U_1也清零,以准备下次的键盘输入,U_1和U_2的清零可通过U_4B口的第7位,即B7产生一个由1到0的跳变来实现。B口的口地址为0x61,为了不改变B口的其它位原来的状况,可用key=intportb(0x61)来读取B口各位,因这时B7已变零。然后再将原数送回,即outportb(0x61,key);这样就将U_1和U_2清零,为下次键盘输入作好准备。

下面是用上述原理设计的一个新INT9中断程序,当按键后,即产生INT9中断。然后若没有激活TSR程序,便通过0x60口读扫描码,是否为52,若是,则表示小数点键或是ALT键按过,到底是小数点键还是ALT键,则通过由bioskey(2)返回的值是否为0x08来确定。若是,则表示按下的是ALT键。因而若没有INT 13调用(即若无磁盘读写),且DOS不忙,便可激活TSR。

```
void interrupt far newint9(void)
  {
    if(! active){
      if(inportb(0x60)==52){
        if((bioskey(2)&0x08)==0x08){
          key=inportb(0x61);
          outportb(0x61,key);
          outportb(0x20,0x20);      /*硬中断结束*/
          if(! int13_active)
            {
              dosbusy=0;            /*DOS不忙*/
              active=1;             /*表示TSR已将激活*/
              (*oldint9)();         /*执行原int9中断程序*/
              TSR程序;
              active=0              /*表示TSR已执行结束,程序可再入*/
            }
          else{
              dosbusy=1;            /*否则DOS忙*/
              (*oldint9)();
            }
          }
        else (*oldint9)();
      }
      else(*oldint9)();
      }
      else(*oldint9)();
  }
```

在该中断程序中,不管各种判断的结果如何,最终都要执行一下原先的INT9中断服务

程序(*oldint9)(),这是因为 int9 中断程序的具体内容也应该是 newint9 中断程序的一部分,这样才能和 DOS 正确接口。而不必花费精力将其具体内容写到 newint9 中。

程序中 int13_active 是 int13 激活的标志,而 active 是 TSR 是否被激活的标志,dos-busy 表示 DOS 忙否的标志,TSR 程序代表 TSR 的具体程序内容。

9.7 程 序 例

下面将介绍三个例子,即分别用 print_screen 和 ctrl_break 键激活驻留程序和用上述所介绍的方法写成的比较完整正规的一个程序。

9.7.1 用 int 5 激活驻留程序

由于当按下 Print_Screen 键时,便产生中断 5,这是由 DOS 管理的屏幕打印程序,关子 DOS 重入问题和热键激活 TSR 程序问题,已经由 DOS 解决,为此我们只要将原来的中断服务程序替换成我们自己的驻留程序即可,这样一旦按下 Print_Screen 键,即可激活我们的驻留程序。不驻留的该程序说明请看 14.7.4 用 32 级灰度的抖动法打印图象例。

```
#include <bios.h>
#include <dos.h>
#include <stdio.h>
unsigned char stack[0x1000];
float grade_value[16];
void read_grade(void);
void interrupt(*oldint5)(void);
void interrupt newint5(void);
int line1;
unsigned sp,ss;
int set_print(int col,int row,unsigned char *buf);
unsigned char prn_data[3841];                    /*定义打印缓冲区*/
void convert_data(int line);
unsigned char shake_grade[32][8]={{0,0,0,0,0,0,0,0},
    {0,0x40,0,0,0,4,0,0},
    {0,0x44,0,0,0,0x44,0,0,},
    {0,8,0,0x22,0,0x80,0,0x22},
    {0x88,0,0x22,0,0x88,0,0x22,0},
    {0,0x44,0,0x92,0,0x44,0,0x29},
    {0x55,0,0x44,0,0x55,0,0x44,0},
    {0x55,0,0x49,0,0x55,0,0x49,0},
    {0x55,0,0x55,0,0x55,0,0x55,0},
    {0x55,0,0x55,0x20,0x55,0,0x55,2},
    {0x55,0x20,0x55,0x2,0x55,0x20,0x55,2},
    {0x55,0x20,0x55,0x88,0x55,2,0x55,0x88},
    {0x55,0x22,0x55,0x88,0x55,0x22,0x55,0x88},
```

```
    {0x55,0x49,0x55,0x88,0x55,0x49,0x55,0x88},
    {0x55,0x49,0x55,0x92,0x55,0x49,0x55,0x92},
    {0x55,0x49,0x55,0xaa,0x55,0x49,0x55,0xaa},
    {0x55,0xaa,0x55,0xaa,0x55,0xaa,0x55,0xaa},
    {0xaa,0xb6,0xaa,0x6d,0xaa,0xb6,0xaa,0x6d},
    {0xaa,0xb6,0xaa,0x77,0xaa,0xb6,0xaa,0x77},
    {0xaa,0xdd,0xaa,0x77,0xaa,0xdd,0xaa,0x77},
    {0xaa,0xdf,0xaa,0x77,0xaa,0xfd,0xaa,0x77},
    {0xaa,0xdf,0xaa,0xfd,0xaa,0xdf,0xaa,0xfd},
    {0xaa,0xff,0xaa,0xdf,0xaa,0xff,0xaa,0xfd},
    {0xaa,0xff,0xaa,0xff,0xaa,0xff,0xaa,0xff},
    {0xaa,0xff,0xb6,0xff,0xaa,0xff,0xb6,0xff},
    {0xaa,0xff,0xbb,0xff,0xaa,0xff,0xbb,0xff},
    {0xff,0xbb,0xff,0x6d,0xff,0xbb,0xff,0xd6},
    {0x77,0xff,0xdd,0xff,0x77,0xff,0xdd,0xff},
    {0xff,0xf7,0xff,0xdd,0xff,0x7f,0xff,0xdd},
    {0xff,0xbb,0xff,0xff,0xff,0xbb,0xff,0xff},
    {0xff,0xbf,0xff,0xff,0xff,0xfb,0xff,0xff},
    {0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff}};   /* 32 级抖动矩阵 */
main()
  {
    oldint5=getvect(5);                          /* 保存旧的中断向量 */
    setvect(5,newint5);                          /* 设置新的中断向量 */
    keep(0,(_SS+(_SP/16)-_psp));                 /* 将程序驻留内存 */
    return(0);
  }
void interrupt newint5(void)                     /* 新的中断 5 服务程序 */
    {
     disable();
     ss=_SS;                                     /* 保存原中断前的寄存器值 */
     sp=_SP;
     _SS=_DS;                                    /* 令堆栈段和数据段为同一段组 */
     _SP=(unsigned)&stack[0x1000-2];             /* 设置堆栈指针 */
     enable();                                   /* 开中断 */
     read_grade();                               /* 读调色板灰度值 */
     for(linel=0;linel<20;linel++)
       {convert_data(linel);                     /* 转换后的数据送打印机打印 */
       set_print(640,1,prn_data);
       }
       oldint5();                                /* 恢复原中断 5 */
       disable();
       _SS=ss;                                   /* 恢复原堆栈 */
       _SP=sp;
```

```
    enable();
    }
void read_grade(void)
  {
   union REGS r;
   float *grade_p;
   int k,m,c,i,red,green,blue;
   grade_p=grade_value;
   for(k=0;k<=15;k++)                          /*得到颜色灰度值*/
    {
     r.h.ah=0x10;
     r.h.al=7;
     r.h.bl=k;
     int86(0x10,&r,&r);                        /*读颜色寄存器*/
     m=r.h.bh;
     r.h.ah=0x10;
     r.h.al=7;
     r.h.bl=0x10;
     int86(0x10,&r,&r);                        /*读模式控制寄存器*/
     c=r.b.bh;
     i=c&0x80;
     r.h.ah=0x10;
     r.h.al=7;
     r.h.bl=0x14;
     int86(0x10,&r,&r);                        /*读颜色选择寄存器*/
     c=r.h.bh;
     c<<=4;
     if(i!=0)
       c+=(m&0xf);
       else{
       c&=0xc0;
       m&=0x3f;
       c+=m;
        };
     r.h.ah=0x10;
     r.h.al=0x15;
     r.x.bx=c;
     int86(0x10,&r,&r);                        /*读颜色值*/
     red=r.h.dh;
     green=r.h.ch;
     blue=r.h.cl;
      *grade_p++=(0.3*(float)red+0.59*(float)green+
     0.11*(float)blue)/63.0;
     }
```

```
}
void convert_data(int line)                    /*将颜色转换为灰度值并打印*/
{
int i,j,colum,row,k;
int color;
float *grade_p;
union REGS r;
unsigned char grade,grade_1;
for(i=0;i<=3841;i++)
prn_data[i]='\0';                              /*清打印缓冲区*/
for(i=0;i<640;i++)
{
for(j=0;j<24;j++)
{
grade_p=grade_value;
r.h.ah=0xd;
r.h.bh=0;
r.x.cx=i;
r.x.dx=j+line*24;
int86(0x10,&r,&r);                             /*读屏幕上一象素颜色值*/
color=r.h.al;
grade_p+=color;
grade=(unsigned char)(31.0**grade_p);          /*转换成灰度*/
row=j;k=j;colum=i;
row%=8;colum%=8;k/=8;
grade_1=shake_grade[grade][row];
grade_1>>=colum;
grade_1&=1;                                    /*得到抖动矩阵元素值*/
prn_data[i*3+k]=(grade_1<<(7-row));            /*送打印缓冲区*/
}
}
}
int set_print(int col,int row,unsigned char *buf)   /*打印函数*/
{
 unsigned n1,n2,i,j,hang;
 if(biosprint(2,0,0)=(0x90);
 {
biosprint(0,27,0);                             /*设行距*/
hiosprint(0,'3',0);
biosprint(0,24,0);
n1=col%256;n2=col/256;
for(hang=0;hang<row;hang++)                    /*打印机打印一行*/
 {
  hiosprint(0,27,0);
```

```
        biosprint(0,'*',0);
        biosprint(0,33,0);
        biosprint(0,n1,0);
        biosprint(0,n2,0);
        for(i=0;i<col;i++)
         {
         for(j=0;j<3;j++){
           biosprint(0,*(buf+bang*3*col+3*i+j),0);
          }
         }
         biosprint(0,13,0);                    /*回车换行*/
         biosprint(0,27,0);
         biosprint(0,43,0);
         biosprint(0,10,0);
        }
    }
return 0;
}
```

9.7.2 用 Ctrl_Break 激活驻留程序

这个程序是利用扬声器唱歌，发声原理同 8.3.3 中的例子。当执行该中断程序时将清屏并显示 sing singing，然后唱"友谊天长日久"歌。主程序中用 daint 保存原 0x1b 中断向量，然后将 music 中断服务程序地址填入 0x1b 中断向量中。接着调用原来 0x1b 中断向量指向的函数。由于当我们通常按 CTRL_BREAK 两键时，系统首先调用 int9 键盘中断，然后判断若是 CTRL_BREAK 键时，便调用 0x1b 中断，该中断服务程序仅是设置一个 CTRL_BREAK 标志，即显示 ^C，然后退出。因系统启动时，BIOS 给 0x1b 设置了一条空处理程序指令，当又引导 DOS 后，DOS 则改写了该中断向量，使其指向了它的 0x1b 中断处理程序。该程序只是增加了设置 CTRL_BREAK 标志的功能，再无什么操作。因而用户可以改写这个中断服务程序，本程序则改写了这个程序，我们将该程序编译连接后，在 DOS 提示符下运行该程序，则在屏上显示 ^C 然后结束。这便是调用原 0x1b 中断程序的结果。但这时 0x1b 中断向量中已是用户改写的 Music 中断服务程序的地址了。第二次在 DOS 提示符下运行该程序，这时 oldint 中则是 music 的地址了，因而又执行 oldint ()；即调用 music，这时将显示 Singing a Song，然后奏友谊天长日久歌曲，然后结束。若将程序中 old_intp()中加入(*Music)()；则在 DOS 下第一次运行，即奏歌曲。从上述程序看来，可以用多种方法来调用中断程序。

本程序运行后，已将 0x1b 中断程序改写，所以要恢复原来的 0x1b 中断程序，只有再重新引导 DOS。

```
#include<stdio.h>
#include<stddef.h>
#include<dos.h>
```

```
#include<stdlib.h>
void interrupt(*oldint)();
void interrupt music();
unsigned char stack[0x1000];
unsigned intsp,intss;
unsigned freq1[87],freq[87]={
    196,262,262,262,330,294,262,294,330,294,262,
    330,394,440,440,394,330,330,262,294,262,294,
    330,294,262,230,230,196,262,440,394,330,330,
    262,294,262,294,440,394,330,330,394,440,523,
    394,330,330,262,294,262,294,330,294,262,230,
    230,196,262,440,394,330,330,262,294,262,294,
    440,394,330,330,394,440,523,394,330,330,262,
    294,262,294,330,294,262,230,230,196,262};
int dely[87]={
    25,38,12,25,25,38,12,25,12,12,56,25,25,50,25,
    38,12,12,12,38,12,25,12,12,38,12,25,25,100,25,
    38,12,12,12,38,12,25,25,38,12,25,25,100,25,38,
    12,12,12,38,12,25,12,12,38,12,25,25,100,25,38,
    12,12,12,38,12,25,25,38,12,25,25,100,25,38,12,
    12,12,38,12,25,12,12,38,12,25,25,100};
main()
  {
    oldint=getvect(0x1b);
    setvect(0x1b,music);
    keep(0,(_SS+(_SP/16)-_psp));
  }
void interrupt music (bp,di,si,ds,es,dx,cx,bx,ax,ip,cs,flags)
{
  int i;
  char orignalbits,bits;
    disable();
    intsp=_SP;
    intss=_SS;
    _SP=(unsigned)&stack[0x1000-2];
    _SS=_DS;
    enable();
    clrscr();
    printf("Singing a Song");
    for(i=0;i<87;i++)
    { outportb(0x43,0xb6);
      freq1[i]=0x1234dc/freq[i];
      outportb(0x42,freq1[i]&0x00ff);
      freq1[i]=freq1[i]>>8;
```

```
        outportb(0x42,freq1[i]);
        bits=orignalbits=inportb(0x61);
        outportb(0x61,bits | 3);
        delay(dely[i] * 25);
        outportb(0x61,orignalbits);
        }
      oldint();
      disable();
      _SP=intsp;
      _SS=intss;
      enable();
}
```

9.7.3 一个完整的驻留程序

这是一个较完整的 TSR 驻留程序，它考虑到了许多因素，使用了前面叙述过的解决 DOS 重入的各种措施。TSR 程序被激活时，将 TSR 的 PSP 设成当前的 PSP，将原堆栈换成 TSR 自己的堆栈，并将 DTA 设置成 TSR 的 DTA。当从 TSR 退出时，又恢复原先的 PSP、DTA 和堆栈及各中断向量。该程序可作为一个驻留程序的标准框架，用户只需在驻留部分填上自己的内容即可。

激活 TSR 而不发生 DOS 重入的条件是：可以在热键按下时；当发生 INT 08H 时钟中断时；或发生 INT 28H 中断时；此时若 DOS 不忙或没有 INT 13H 中断；DOS 没有处理紧急错误时；且不会发生 TSR 重入时(即未激活 TSR 时)，则可以激活 TSR 程序，因而当发生三种中断时，INT 9H 中断(当有键按下时必然产生)，INT 08H 中断和 INT 28H 中断都有可能激活 TSR，为此必须改写这三个中断服务程序，使得当发生该类中断时，再根据其它一些条件来判断可否激活 TSR。当允许激活时，便运行 TSR 程序，这些改写过的三个中断程序分别如下：

1. INT 08 时钟中断

```
void interrupt far newint 8(void)
  {
    (*oldint8)()                    /*执行旧中断*/
    if(! active&&hot_dosbusy&&! dosbusy()&&! int13_active)
      {
        hot_dosbusy=0;              /*将热键按下，且 DOS 忙标志置 0*/
        active=1;                   /*标明 TSR 已将激活*/
        enable();
        tsr_active();               /*激活 tsr_active()程序*/
        active=0;
      }
  }
```

即当 TSR 未被激活时，热键按下，且 DOS 安全不忙和没有 INT 13H 调用时激活 TSR(即

tsr_active())。

2. INT 9 键盘中断

```
void interrupt far newint9(void);
```

该中断程序,已在 9.3 激活驻留程序的方法中作了介绍,可参阅。

3. INT 28 中断

```
void interrupt newint28(void)
{
  int28_active++;
  if(not_dosbusy&&(! intdosbusy())&&! active&&! int13_active)
  {
    active=1;
    tsr_active();
    active=0
  }
  int28_active--;
  (*oldint28)();
}
```

激活 TSR 的条件中当 intdosbusy()函数测得 DOS 安全标志为 1,即访问 DOS 功能是安全的,和 DOS 没有处理紧急错误时,则返回 0。int28_active++表示 int28 已被激活,该标志将限制它再次激活 TSR。当 TSR 服务程序执行完后,则可 int28_active--使其再次作为允许 TSR 激活的一个条件。

当按下 CTRL_BREAK(即产生 INT 1BH 中断)和按下 CTRL_C 键时(即产生 INT 23H 中断)应不能中断停止 TSR 程序的运行,即应使产生的这两种中断无效,否则如前所述,将不能恢复被修改过的一些中断向量,导致系统不能恢复。

故当发生这两种中断时,使其空操作即可,即定义为:

```
void interrupt far newint1b(void)
{
  }
```

即可。INT 23H 同理。

当产生 INT 13H 中断时,应执行原中断内容,并设置正在执行该中断的标志,以免激活 TSR,即 int13_active++;并保存各寄存器结果,即

```
void interrupt far newint 13(bp,di,si,ds,es,dx,cx,bx,ax,ip,cs,flags)
unsigned bp,di,si,ds,es,dx,cx,bx,ax,ip,cs,flags;
  {
    int13_active++;
    (*oldint13)();
    ax=_AX;
    cx=_CX;
    dx=_DX;
```

```
    flags=_FLAGS;
    --int 13_active;
  }
```

当产生严重错误，而 DOS 进行 INT 24H 调用时，可告诉 DOS 忽略该错误，即改写为：

```
void interrupt far newint24(bp,di,si,ds,es,dx,cx,bx,ax,ip,cs,flags)
unsigned bp,di,si,ds,es,dx,cx,bx,ax,ip,cs,flags;
{
  ax=0x03        /* 令 DOS 忽略该错误 */
}
```

当进行 PSP 交换时，采用了 INT 21H 的 51H 号功能调用，由 getpsp 函数得到当前进程的 PSP 段地址，然后用 INT 21H 的 51 号功能调用来设置 PSP 为当前的 PSP，即：

```
void setpsp(unsigned segpsp)
  {
    union REGS regs;
    if(! crit_err_ptr)
      return;
    *crit_err_ptr=0xff;              /* 致命错误标志总指向 PSP 最后 */
    regs.h.ah=0x51;
    regs.x.bx=segpsp;
    intdos(&regs,&regs);
    *crit_err_ptr=0;                 /* 致命错误标志指向 PSP 开头 */
  }
```

其中！crit_err_ptr 表示当发生致命错误时返回，否则设置当前的 PSP。

当 TSR 程序被激活后，即在本程序中 tsr_active()函数被执行，该 TSR 程序首先设置自己的堆栈，并检查若 dosbusy 和没有 INT28H 中断，只是设置 not_dosbusy=1，为下一次 INT28H 激活 TSR 准备条件，否则便激活 TSR。这时，保存 INT 1BH、INT23H 和 INT 24H 的原中断向量，并设置新的 INT 1BH、INT 23H 和 INT 24H 中断向量。然后进行 PSP 及 DTA 的替换，保存原先的扩展错误信息，然后执行 TSR 的真正内容。本例用简单的输出字符串来代替。执行完后，则恢复原先的 DTA、PSP 和 INT 1BH、INT 23H 及 INT 24H 各中断向量，并恢复原先的堆栈，从而完成一次 TSR 的激活操作。

该程序还有一个功能是撤消 TSR 的驻留，即当在 DOS 下，键入 TSR D↙时，表示撤消 TSR 的驻留。此时将调用 do_deinstall()函数，它将把程序内容显示在屏幕上，并调用 deinstall()函数。该函数再调用 communicate()函数，使之执行 tsr_exit()函数，完成恢复 PSP，并回到原先的系统下。tsr_exit()函数要通过调用 unlinkveet()函数来确定 INT 8，INT 9，INT 28H，INT 13H，INT 62H 中断向量是否被改变，即若其中有一个不是原先的(即激活 TSR 后设置的)，便可能又有别的驻留程序，因而将不改变当前的 PSP。

do_deinstall()函数将根据执行 dinstall()汇编子程序的结果，给出 TSR 程序撤消与否的信息，即没有移走、已撤消和发生了错误三种情况。

当在 DOS 下执行该程序(设程序名为 MYTSR)，即 MYTSR↙，将由 main()函数完成

驻留 MYTSR，若已经驻留，则在屏幕上显示：TSR program already installed 信息，不再进行驻留操作。当进行驻留前，该程序首先将准备改写的各中断向量保存下来，并用新的中断向量值代替原先的，然后计算驻留空间大小，并完成驻留。此时机器将处于等待状态，还可进行别的操作。一旦有热键按下（即 ALT），再和各种激活 TSR 的条件相与，便可激活 TSR。

当在 DOS 下键入 MYTSR d↙，将由主函数 main()完成如前所述的撤消驻留的操作。

若调用该程序的命令行，不是这两种形式，屏幕将出现帮助信息，即执行 help_message()。

最后解释一下在 main()函数中程序驻留的实现方法，和进行堆栈切换的问题：

TSR 驻留程序长度计算是这样进行的，由于在小模式下内存分配图已在图 5.2 中画出，现简化后用图 9.5 表示，程序 PSP 由 DOS 分配，它的段地址在 DS 中（当未执行程序时），故用节为单位时，其长度可表示为

size=_DS+_SP/16-_PSP

故可用 keep 函数完成驻留

keep(0,size);

为了将要求的空间由 DOS 进行分配，给出一个由指针连在一起的段，所以用 setblock()函数进行调整，然后用 keep()函数进行驻留。

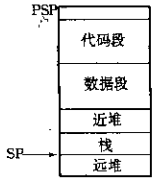


图 9.5 小模式下的内存分配

由于栈的交换要涉及对许多寄存器的操作，故用汇编语言实现。关于 C 语言程序和汇编语言的混合编程，可参阅第十四章有关内容。

将该程序和栈切换的汇编程序写成 object 文件，进行编译连接，即可生成可执行程序。

程序清单如下：

```
#include<dos.h>
#include<stdlib.h>
#include<stdio.h>
#include<stddef.h>
#include<conio.h>
#include<bios.h>
#define STACK_SIZE 8192                 /*设置栈的容量*/
#define KEYBOARD 0x60                   /*键盘数据输出口*/
#define PSP_TERMINATE 0x0a              /*PSP 中的结束地址*/
#define PSP_PARENT_PSP  0x16            /*PSP 中的父进程的 PSP*/
#define PSP_ENV_PSP 0x2c                /*PSP 的环境块地址*/
#define SCANCODE 52                     /*小数点键的 ASCII 码*/
#define PARAGRAPHS(x)((FP_OFF(x)+15)>>4)
#define KEYMASK 8                       /*ALT 键的扩充 ASCII 码*/
char far *indos_ptr=0;                  /*DOS 安全标志的指针*/
char far *crit_err_ptr=0;               /*致命错误标志的指针*/
char far *stack_ptr;                    /*TSR 栈指针*/
char far *dta1;                         /*前台进程的磁盘传输区地址*/
char far *dta2;
```

```
unsigned ss_save;                    /* 保存栈段寄存器变量 */
unsigned sp_save;                    /* 保存栈指针寄存器变量 */

static union REGS rg;
static struct SREGS seg;
static struct DOSERROR doserr;
static unsigned foreground_psp;      /* 前台进程的 PSP */
static unsigned long terminateaddr;  /* 取掉 TSR 时用 */
static int active=0;                 /* 表示 TSR 未激活标志 */
static int hot_doshusy=0;            /* DOS 不忙 */
static int int28_active=0;           /* int 28 未激活 */
static int int13_active=0;           /* int 13 未激活 */
static int key;                      /* 读键盘端口时用 */
extern int sign;                     /* 删除 TSR 时,汇编子程序使用的返回标志 */
typedef struct{
unsigned bp,di,si,ds,es,dx,cx,bx,ax,ip,cs,flags;
}INTERRUPT_REGS;
static void interrupt(*old8)();
static void interrupt(*old9)();
static void interrupt(*old1b)();
static void interrupt(*old23)();
static void interrupt(*old24)();
static void interrupt(*old28)();
static void interrupt(*old62)();
static void interrupt(*old13)();
void interrupt far new8();
void interrupt far new9();
void interrupt far new1b();
void interrupt far new23();
void interrupt far new24(INTERRUPT_REGS p);
void interrupt far new13(INTERRUPT_REGS p);
void interrupt far new28();
void interrupt far communicate(INTERRUPT_REGS p);
void help_message(void);
void initindos(void);
void tsr_active(void);
void tsr_exit(void);
void do_deinstall(char *program);
extern void set_stack(void);
extern void restore_stack(void);
void setpsp(unsigned segpsp);
void setexterr(struct DOSERROR near *err);
int dosbusy(void);                   /* DOS 忙时,该函数返回 0 */
int int28dosbusy(void);              /* 若 DOS 使用安全,且致命错误标志非零,该函
```

```
                                        数返回零*/
extern int deinstall(void);                    /*判TSR是否删除的函数*/
int change_vect(int vect,void interrupt(*newint)(),void interrupt(*oldint)());
main(int argc,char *argv[])
{
   union REGS rg1,rg2;
   struct SREGS segs;
   unsigned memory;                            /*驻留内存大小的变量*/
   unsigned far *fp;                           /*指向环境块段址的指针*/
   initindos();                                /*初始化DOS安全标志和致命错误标志*/
   switch(argc){
    case 1:
     if(getvect(0x62)!=NULL){
       puts("TSR program already installed");
       exit(1);
     }
       break;
    case 2:
     if(toupper(argv[1][0])=='D'&&(getvect(0x62)!=NULL))
       do_deinstall(argv[0]);
     else
       help_message();
       break;
    default:
     help_message();
   }
   if((stack_ptr=malloc(STACK_SIZE))==NULL){
    printf("Not enough mrmory\n");
    exit(1);
   }                                           /*设置TSR使用的栈*/
   stack_ptr+=STACK_SIZE;
   old8=getvect(0x08);
   old9=getvect(0x09);
   old13=getvect(0x13);
   old28=getvect(0x28);
   old62=getvect(0x62);
   setvect(0x08,new8);
   setvect(0x09,new9);
   setvect(0x13,new13);
   setvect(0x28,new28);
   setvect(0x62,communicate);
   fp=MK_FP(_psp,PSP_ENV_PSP);
   freemem(*fp);                               /*释放环境块*/
   segread(&segs);
```

```
    memory=segs.ds+PARAGRAPHS(stack_ptr)-_psp;
    setblock(_psp,memory);                    /*调整内存空间*/
    keep(0,memory);                           /*进行驻留*/
    }
void interrupt far new8(void)
{
  (*old8)();
  if(! active&&hot_dosbusy&&! dosbusy()&&! int13_active)
   {
    hot_dosbusy=0;
    active=1;
    enable();
    tsr_active();
    active=0;
    }
   }
void interrupt far new9(void)
{
  if(! active){
   if(inportb(0x60)==SCANCODE){    /*从端口读键扫描码*/
    if((bioskey(2)&KEYMASK)==KEYMASK){
     key=inportb(0x61);                       /*清键盘缓冲区*/
     outportb(0x61,key);
     outportb(0x20,0x20);                     /*键盘硬中断结束*/
      if(! int13_active){                     /*如果没有磁盘读、写,可弹出 TSR 程序*/
       hot_dosbusy=0;
       active=1;
       (*old9)();
       tsr_active();
       active=0;
       }
      else{
       hot_dosbusy=1;
       (*old9)();
       }
      else(*old9)();
      }
    else(*old9)();
    }
   else(*old9)();
  }
void interrupt far new13(INTERRUPT_REGS p)
{
```

```
  int13_active++;
  (*old13)();
  p.ax=_AX;
  p.cx=_CX;
  p.dx=_DX;
  p.flags=_FLAGS;
  --int13_active;
}
void interrupt far new1b(void)
{
  /* Do nothing */
}
void interrupt far new23(void)
{
  /* do nothing */
}
void interrupt far new24(INTERRUPT_REGS p)
{
  p.ax=0x03;                              /*忽略 DOS 错误*/
}
void interrupt far new28(void)
{
  int28_active++;
  if(hot_doshusy&&(! int28dosbusy())&&! active&&! int13_active){
                                          /*判 TSR 能否弹出*/
   active=1;
   tsr_active();
   active=0;
   }
int28_active--;
(*old28)();
}
void interrupt far communicate(INTERRUPT_REGS p)   /*通讯中断函数*/
{
   terminateaddr=((long)p.bx<<16)+p.dx;
   if(! active){
     enable();
     tsr_exit();                          /*退出驻留*/
     active=1;
     }
   }
void help_message(void)                   /*帮助信息*/
{
  printf("The correct way of using this TSR program is\n");
```

```
  printf("TSR[D]\n");
  printf("TSR D stands for deinstalling the TSR program\n");
  printf("TSR stands for installing the tar program\n");
  exit(1);
}
void tsr_active(void)                    /* TSR 激活函数 */
{
  set_stack();                           /* 设置自己的栈 */
  if(doshusy()&&! int28_active)
   hot_dosbusy=1;
  else{
   hot_dosbusy=0;
   old1b=getvect(0x1b);
   old23=getvect(0x23);
   old24=getvect(0x24);
   setvect(0x1b,new1b);
   setvect(0x23,new23);
   setvect(0x24,new24);
   foreground_psp=getpsp();              /* PSP 切换 */
   setpsp(_psp);
   dta1=getdta();                        /* DTA 切换 */
   dta2=MK_FP(_psp,0x80);
  setdta(dta2);
   dosexterr(&doserr);                   /* 保存前台进程的扩展错误 */
   printf("TSR program has been install! \n");  /* 驻留的应用程序内容 */
   printf("This program is a easy example! \n");
   setexterr((struct DOSERROR near *)&doserr);
   setdta(dta1);                         /* 恢复前台的 DTA */
   setpsp(foreground_psp);               /* 恢复前台的 PSP */
   setvect(0x1b,old1b);
   setvect(0x23,old23);
   setvect(0x24,old24);
  }
  restore_stack();                       /* 恢复栈 */
}
void setpsp(unsigned segpsp)             /* 设置 PSP 的函数 */
  {
   union REGS regs1;
   if(! crit_err_ptr)
   return;
   *crit_err_ptr=0xFF;
  regs1.h.ah=0x50;
  regs1.x.bx=segpsp;
  intdos(&regs1,&regs1);
```

```
    * crit_err_ptr=0;
}
void setexterr(struct DOSERROR near * err)   /* 设置扩展错误信息的函数 */
{
   rg.x.ax=0x5d0a;
   rg.x.bx=0x00;
   segread(&seg);
   rg.x.dx=(int)err;
   intdosx(&rg,&rg,&seg);
}
void initindos(void)
{
   union REGS regs;
   struct SREGS sregs;
   regs.h.ah=0x34;
   intdosx(&regs,&regs,&sregs);
   indos_ptr=MK_FP(sregs.es,regs.x.bx);
   crit_err_ptr=MK_FP(sregs.ds,regs.x.si);
}
int dosbusy(void)                          /* 判 DOS 忙函数 */
{
   if(indos_ptr&&crit_err_ptr)
    return(*crit_err_ptr || *indos_ptr);
   else
    return 0xffff;
}
int int28dosbusy(void)
{
   if(indos_ptr&&crit_err_ptr)
    return(*crit_err_ptr || (*indos_ptr>1));
    else
    return 0xffff;
}
int change_vect(int vect,void interrupt(*newint)(),void interrupt(*oldint)())
{
   if(newint==getvect(vect)){
    setvect(vect,oldint);
    return 0;
    }
   return 1;
}
void tsr_exit(void)
{
   set_stack();                            /* 设置自己的栈空间 */
```

```
  if(! (change_vect(8,new8,old8)|| change_vect(9,new9,old9)
   || change  vect(0x28,new28,old28)|| change_vect(0x13,new13,old13)||
   change_vect(0x62,communicate,old62)))
     {/*把当前活动进程的 PSP 设置为 TSR 进程 PSP 中的父本 PSP*/
      *(int far *)(((long)_psp<<16)+PSP_PARENT_PSP)=getpsp();
      *(long far *)(((long)_psp<<16)+PSP_TERMINATE)=terminateaddr;
      setpsp(_psp);        /*设置当前 TSR 进程的 PSP 为当前活动进程的 PSP*/
      bdos(0x4c,0,0);      /*代出口码的程序结束,退到父程序*/
      }
     }
void do_deinstall(char *program)
{
  fputs(program,stdout);
  deinstall();
  switch(sign){              /*判断汇编子程序返回标志*/
   case 1:
     puts("deactivated but not removed");
     break;
   case 2:
   puts("Deinstalled");
     break;
   default:
     puts("Unexpected errors");
     break;
   }
  exit(0);
}


_TEXT SEGMENT BYTE PUBLIC 'CODE'
_TEXT ENDS
_DATA SEGMENT WORD PUBLIC 'DATA'
_DATA ENDS
CONST SEGMENT WORD PUBLIC 'CONST'
CONST ENDS
_BSS SEGMENT WORD PUBLIC 'BSS'
_BSS ENDS
DGROUP GROUP _DATA,CONST,_BSS
  ASSUME CS:_TEXT,DS:DGROUP,SS:DGROUP
_DATA SEGMENT
  public  sign
  _sign dw 0
_DATA ENDS                          ;返回到C程序中的标志变量
```

```
  public _deinstall
  extrn _ss _save:near
  extrn _sp _save:near
_TEXT SEGMENT
_deinstall proc far
  push si
  push di
  push bp
  mov word ptr _ss _save,ss          ;保存主程序的栈段地址
  mov word ptr _sp _save,sp
  mov cs:_ds _save,ds                ;保存数据段地址
  mov hx,cs
  mov dx,offset terminateaddr        ;BX:DX 为程序的结束地址
  inc _sign
  int 62h                            ;调用 communicate 函数
  jmp short noterminate
terminateaddr:
  mov ax,cs:_ds _save
  mov ds,ax
  inc _sign
  mov ss,word ptr _ss _save          ;恢复栈
  mov sp,word ptr _sp _save
noterminate:
  pop bp
  pop di
  pop si
  ret
_deinstall endp
_ds _save dw 0
_TEXT ENDS
END

C:\TC>type mtsr2.asm
_TEXT SEGMENT BYTE PUBLIC 'CODE'
_TEXT ENDS
_DATA SEGMENT WORD PUBLIC 'DATA';
_DATA ENDS
CONST SEGMENT WORD PUBLIC 'CONST'
CONST ENDS
_BSS SEGMENT WORD PUBLIC 'BSS'
_BSS ENDS
DGROUP GROUP _DATA,CONST,_BSS
  ASSUME CS:_TEXT,DS:DGROUP,SS:DGROUP
```

```
    public _set_stack
    public _restore_stack
    extrn _ss_save:near
    extrn _sp_save:near
    extrn _stack_ptr:near
_TEXT SEGMENT
  _set_stack proc far
    pop ax                              ;保存返回值的偏移地址
    pop bx                              ;保存返回值的段地址
    mov word ptr _ss_save,ss            ;保存前台程序的栈
    mov word ptr _sp_save,sp
    mov ss,word ptr _stack_ptr+2        ;设置 TSR 程序的栈
    mov sp,word ptr _stack_ptr
    push bx                             ;正常返回
    push ax
    ret
_set _stack endp
_restore _stack proc far
    pop cx
    pop bx
    mov word ptr _stack _ptr+2,ss
    mov word ptr _stack _ptr,sp
    mov ss,word ptr _ss _save
    mov sp,word ptr _sp _save
    push bx
    push cx
    ret
_restore _stack endp
_TEXT ENDS
END
```

9.8 鼠 标 器

鼠标器是目前广泛使用的一种人机交互工具，在当前流行的许多软件中，都已大量使用，如 WINDOWS 和一些商品软件都用鼠标器进行菜单选择和光标定位，它在许多地方可以代替键盘操作，且有它独到之处。

9.8.1 鼠标器工作原理简介

鼠标器有一、二或三个按钮，现普遍使用三个按钮的。

目前主要有两种形式的鼠标器，第一种是机械式的，第二种是光电式的。在机械式鼠标器中使用一个转动球，随着鼠标器移动，转动球在旋转，它的移动使得传感器将这些移动变成移动光标的方向信息。光电式鼠标器使用二个发光二极管 LED 和两个光电晶体管来检测

移动。一个 LED 发红光，而另一个则产生紫外光，光电鼠标器用一个特殊的焊盘来改变 LED 的光强。这个焊盘有两个方向线，当鼠标器向一个方向移动时，吸收红光，向另一方向移动时，吸收紫外光，光间断的颜色和数目决定了鼠标器移动的方向和距离，它将这些信息变成数字信号向计算机发送。

鼠标器通过 RS232 异步串行口向计算机发送这些移动和移动方向及距离多少的信息。对于 PC 机，它常接在 COM1 口或 COM2 口上，使用时主机用硬件中断 INT 0BH 接收串行口送来的数据信息(对 COM 2 口)，若使用 COM 1 口，则使用 INT 0CH 中断来接收 COM 1 传来的信息，如同用键盘中断 INT 9 接收键盘信息一样。所以从实质上说，鼠标器如同键盘一样，是向计算机送信息的一个输入设备，它通常以 1200 波特率的传送速度和数据传送格式为 8 个数据位的方式向主机发送数据，这些数据代表着光标的移动和按钮的状态，一般定义 5 个字节为一组数据，各字节含义如下(设鼠标器底开关置于 3)：

第一字节：80H～87H 是鼠标的三个按钮按下或松开的 8 种组合值，其中单独按下左、中、右各钮的值分别为 83H、85H、86H，三个按钮不按时，其值为 87H。

第二字节：0 没有上下方向移动，1＋表示以 1 为起点递增，数值越大，表示向右移动速度愈快。255－，表示向左移动，即以 255 为起点向左移动，移动速度愈快，则数值递减越快。

第三字节：0 表示没有左右方向的移动，1＋表示向上移动，增加越快，表示移动越快，255－表示向下移动，移动越快，数值递减越快。

第四和第五字节含义与第二第三字节相似。

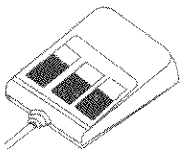


图 9.6　鼠标器图

鼠标在移动时，其光标并没有直接显示在屏幕上，而是在一个假想的虚拟屏幕上。它以象素为单位进行计数，然后再将其映射到显示屏幕上。由于显示模式不同，即分辨率不同，满屏的象素个数不同，因而单位长度上的象素个数也不同，即鼠标计数的个数也不同。这些映射过程，都是通过鼠标器的驱动程序来完成。

一般鼠标驱动程序首先将 RS232 进行初始化，然后采用中断方式接收鼠标数据，规定采用 INT 33H 中断，即中断地址为 0000：30H～33H。鼠标驱动程序由生产鼠标的厂家提供，该程序提供了许多功能，通过设置不同的入口参数，通过 INT 33H 鼠标中断调用，来使用这些功能，如 Microsoft 的 INT 33H 中断调用，提供了 35 个功能调用。

9.8.2　鼠标器的 INT 33H 功能调用

DOS 操作系统和 Turbo C 2.0 并不支持鼠标器的操作，因而要使用鼠标器，必须首先要安装其相应的驱动程序，通常使用的方法是在 CONFIG.SYS 文件中加入一行信息：DEVICE＝MOUSE.SYS，使得在 DOS 启动时，将 Mouse 的驱动程序也装入内存，或者也可直接运行 mouse.com 文件，使其驻留在内存。

当安装好了鼠标器的驱动程序，并进行了初始化后，就可使用鼠标器的驱动程序来管理鼠标器的各种操作。鼠标器驱动程序将 INT 33H 中断作为鼠标器的操作中断，这样每当移动一下鼠标器，或者按动一下鼠标器的按钮，就将产生一次 INT 33H 中断。而鼠标驱动程序将按照中断时的入口参数，调用不同的功能处理程序，来完成中断服务。

对于 Microsoft 鼠标驱动程序，共提供了 30 多个功能调用，用户可以通过 INT 33H 中断调用，选用不同的入口参数，来实现相应的功能调用，这些功能号和对应的功能如下所列：

功能号	功能简介	功能号	功能简介
0	鼠标复位及取状态	20	交换中断程序
1	显示鼠标光标	21	取驱动程序存储要求
2	鼠标光标不显示	22	保存驱动程序状态
3	取按钮状态和鼠标位置	23	恢复驱动程序状态
4	设置鼠标光标位置	24	设辅助程序掩码和地址
5	取按钮压下状态	25	取用户程序地址
6	取按钮松开状态	26	设置分辨率
7	设置水平位置最大值	27	取分辨率
8	设置垂直位置最大值	28	设置中断速度
9	设置图形光标	29	设置显示器显示的页号
10	设置文本光标	30	取显示器显示的页号
11	取鼠标器移动的方向和距离	31	关闭驱动程序
12	设中断程序掩码和地址	32	打开驱动程序
13	打开光笔模拟	33	软件重置
14	关闭光笔模拟	34	选择语言
15	设置 Mickey 与象素比	35	取语言编号
16	条件关闭	36	取版本号及鼠标类型
19	设置速度加倍的下限		

表 9.3 列出了一些上述功能号中的基本的功能调用和相应的入口参数，及调用后的出口参数，这些调用是一般常用的：

表 9.3 鼠标驱动程序(INT 33H)常用功能码及参数

功能码	功能	入口参数	出口参数
0	鼠标复位及取状态	m=0 (AX=0)	m1=−1 鼠标安装成功 (AX=−1) m1=0 鼠标安装失败 (AX=0) m2= 鼠标按钮的数目 (BX=)
1	显示鼠标光标	m1=1 (AX=1)	无
2	不显示鼠标光标	m1=2 (AX=2)	无
3	取按钮状态和鼠标位置	m1=3 (AX=3)	m2=各按钮状态 (BX=)见注释

续表

功能码	功　能	入口参数	出口参数
4	设置鼠标光标位置	m1＝4 (AX＝4) m3＝光标 x 坐标 (CX＝) m4＝光标 y 坐标 (DX＝)	
5	取按钮压下状态	m1＝5 (AX＝5) m2＝按钮号 (BX＝) 0 为左按钮 1 为右按钮	m1＝各按钮状态 (AX＝)见注释 m2＝自上次调用以来该按钮按下的次数 (BX＝) m3＝最后一次按下时鼠标的 x 坐标 (CX＝) m4＝最后一次按下时鼠标的 y 坐标 (DX＝)
6	取按钮松开状态	m1＝6 (AX＝) m2＝按钮号 (BX＝)	m1＝各按钮状态 (AX＝)见注释 m2＝自上一次调用以来，该按钮被释放的次数 m3＝最后一次释放时鼠标的 x 坐标 (CX＝) m4＝最后一次释放时鼠标的 y 坐标 (DX＝)
7	设置水平位置最大值	m1＝7 (AX＝) m3＝x 坐标的最小值 m4＝x 坐标的最大值	
8	设置垂直位置最大值	m1＝8 (AX＝)， m3＝y 坐标的最小值 m4＝y 坐标的最大值	
11	取鼠标器移动的方向和距离	m1＝11 (AX＝)	m3＝x 方向移动距离 m4＝y 方向移动距离
12	设中断程序掩码和地址	m1＝12 m3＝调用掩码，见注释 m4＝程序地址	

注释 1：鼠标各按钮的状态，由下面信息格式提供：

位	等于 0 时	等于 1 时
0	左按钮未按下	左按钮正在按下

1	右按钮未按下	右按钮正在按下
2	中按钮未按下	中按钮正在按下

注释 2：当用功能 12 设置用户的鼠标中断服务程序时，其入口参数 m3＝调用掩码，该掩码表示在哪种条件发生时，产生中断，即执行用户定义的中断服务程序。其掩码位与中断产生条件如下：（该掩码位置 1 时，便产生中断）

掩码位	中断的条件
0	光标位置移动
1	左按钮按下
2	左按钮松开
3	右按钮按下
4	右按钮松开

在上表中，各入口参数和出口参数中，m1，m2，m3，m4 分别存放在 AX，BX，CX，DX 各寄存器中。表中用括号标示出了。当用汇编语言调用上表中的功能时，只要将相应入口参数赋给相应的 AX，BX，CX，DX 寄存器即可，出口参数则从上述的寄存器中得到。

9.8.3 用鼠标器作图

下面的程序就是通过调用 INT 33H 软中断的不同功能实现画图。该程序演示了如何调用这些功能，为了使程序便于理解，将作图时要用到的一些功能调用以函数形式实现，这些函数有：

init(int xm1,int xma,int ymi,int yma);

该函数将通过调用 int 33H 的 0 号功能调用对鼠标器进行初始化，调用 7 号和 8 号功能，设置 x 和 y 位置的最小和最大值。这就为鼠标器移动进行了初始化准备。由于 0 号功能调用是测试鼠标驱动程序是否安装，因此在运行该程序前必须首先执行鼠标驱动程序 mouse.com，若使用的是键盘连接的球形鼠标器，则必须首先执行驱动程序 ball.com，若调用该函数执行了 0 号功能调用，当返回值为 0 时（即返回参数为 0），表示未安装成功，这可能是鼠标器或驱动程序未安装。这时程序将显示 Mouse or Mouse Driver Absent，并回到系统。

read (int *mx,int *my,int *mbutt)

该函数将通过调用 int 33H 的 3 号功能调用，读鼠标的位置和按钮状态。鼠标的 x、y 位置值将由指针 mx 和 my 给出，而按钮状态则由 mbutt 指针给出。

cursor(int x,int y);

该函数将用画线函数 line()画出一个十字形光标。

newxy(int *mx,int *my,int *mbutt);

该函数将通过调用 read()函数来判断是否有按钮按下，若按下，则调用 cursor()函数在新位置画出一十字光标。

光标的移动是通过将原位置光标用背景色再画而使其消失，然后在新位置处重新画一个光标，从而实现光标移动的动感。

当光标移动到 quit 上时，程序便终止，并回到系统。

运行该程序前，要将鼠标器安装在 COM1 口，并在当前目录下运行 mouse 驱动程序，然

后运行该程序，按住鼠标器的任意键并移动，十字光标将随鼠标而移动，并用白色线条在蓝色背景上画出移动的效果来。

```
#include <dos.h>
#include <stdio.h>
#include <graphics.h>
union REGS regs;
int init();
int read();
void cursor(),newxy();
int xmin,xmax,ymin,ymax,x_max=639,y_max=479;
main()
  {
    int buttons,xm,ym,x0,y0,x,y;
    char str[100];
    int driver=VGA;
    int mode=VGAHI;
    initgraph(&driver,&mode,'');
    clrscr();
    rectangle(0,0,x_max,y_max);
    setfillstyle(SOLID_FILL,BLUE);
    bar(1,1,x_max-1,y_max-1);
    outtextxy(3,15,"move mouse using any button.");
    outtextxy(285,15,"quit");
    xmin=2;
    xmax=x_max-1;
    ymin=8;
    ymax=y_max-2;
    setwritemode(XOR_PUT);
    if(init(xmin,xmax,ymin,ymax)==0)  /*调用 init 函数对鼠标器初始化*/
     {
      printf("Mouse or Mouse Driver Absent,Please Install!");
      delay(5000);
      exit(1);
     }
    x=320;y=240;
    cursor(x,y);              /*置十字光标在屏幕中心*/
    for(;;){
      newxy(&x,&y,&buttons);
      if(x>=280&&x<=330&&y>=12&&y<=33&&buttons)
                              /*十字光标移到 quit 处时*/
      {
       cleardevice();
       exit(0);               /*回到系统*/
```

```
        }
      }
    }
void cursor(int x,int y)              /*画十字光标函数*/
  {
   int x1,x2,y1,y2;
   x1=x-4;
   x2=x+4;
   y1=y-3;
   y2=y+3;
   line(x1,y,x2,y);
   line(x,y1,x,y2);
  }
int init(int xmi,int xma,int ymi,int yma)     /*鼠标器初始化函数*/
  {
   int retcode;
   regs.x.ax=0;
   int86(51,&regs,&regs);
   retcode=regs.x.ax;
   if(retcode==0)
     return 0;                  /*返回0表示鼠标或鼠标驱动程序未安装*/
   regs.x.ax=7;
   regs.x.cx=xmi;
   regs.x.dx=xma;
   int86(51,&regs,&regs);
   regs.x.ax=8;
   regs.x.cx=ymi;
   regs.x.dx=yma;
   int86(51,&regs,&regs);  /*表示鼠标器和驱动程序已安装*/
   return retcode;
  }
int read(int *mx,int *my,int *mbutt)  /*读鼠标的位置和按钮状态函数*/
  {
   int xx0=*mx,yy0=*my,buto=0;
   int xnew,ynew;
   do{
     regs.x.ax=3;
     int86(51,&regs,&regs);
     xnew=regs.x.cx;
     ynew=regs.x.dx;
     *mbutt=regs.x.bx;
     }while(xnew==xx0&&ynew==yy0&&*mbutt==buto);
     *mx=xnew;
     *my=ynew;
```

```
    if( * mbutt){
     * mx=xnew;
     * my=ynew;
     return -1;
     }
     else{
      * mx=xnew;
      * my=ynew;
      return 1;
      }
    }
void newxy(int * mx,int * my,int * mbutt)   /* 是否有按钮按下,若按下,在新位置画十字 */
  {
   int ch,xx0= * mx,yy0= * my,x,y;
   int xm,ym;
   ch=read(&xm,&ym,mbutt);
   if(ch>0)
     {
      cursor(xx0,yy0);
      cursor(xm,ym);
     }
    else
     {
      cursor(xx0,yy0);
      cursor(xm,ym);
      putpixel(xm,ym,7);
     }
    * mx=xm;
    * my=ym;
   }
```

9.8.4 用鼠标器热键激活 TSR 程序

INT 33H 的功能号为 12 的功能调用可以设置用户按鼠标器按钮而引起中断，并转入由用户设置的中断服务程序去执行，这样我们若设置了产生中断的条件(鼠标某按钮按下)，又将中断服务程序入口设置为 TSR 程序的入口，并将该程序驻留在内存，当我们按下对鼠标器定义的某按钮热键时，便可立即激活 TSR 程序。

可以作一个小程序进行示范，例如有一清屏程序，该程序是在屏幕文本方式下使屏的显示颜色进行改变，程序中 * vsg 表示显示存储器的首地址，该函数对每行、列对应的单元进行写入，以改变原来的值。

```
void main()
{
```

```
  int j,k;
  static int colr=0;
  char far *vsg=0xb8000000;
  for (j=0;j<50;j++)
    for(k=0;k<80;k++)
      *(vsg+j*80*2+k*2+1)=colr&0x7f;
    colr++;
}
```

现将该程序驻留在内存，并定义鼠标器右按钮按下时作为热键激活该程序，程序如下：

```
#include<dos.h>
#include<stdio.h>
#include<alloc.h>
static union REGS rg;
static struct SREGS reg;
unsigned size=600;
void mouse_key();
main()
{
  int mess;
  rg.x.ax=0;
  int86(0x33,&rg,&rg);                    /*对鼠标器初始化*/
  mess=rg.x.ax;
  if(mess==0){                            /*返回值为0,表示失败!*/
    printf("Mouse or Mouse Driver Absent\n");
    delay(1000);
    exit(1);
    }
  rg.x.ax=12;
  rg.x.cx=8;
  rg.x.dx=(int)mouse_key;
  reg.es=((long)mouse_key)>>16;
  int86x(0x33,&rg,&rg,&reg);
  rg.x.ax=0x3100;
  rg.x.dx=size;
  intdos(&rg,&rg);                        /*将程序驻留内存*/
  }
void mouse_key()
{
  int j,k;
  static int colr=0;
  char far *vsg=0xb8000000
  for (j=0;j<50;j++)
   for(k=0;k<80;k++)
```

```
        *(vsg+j*80*2+k*2+1)=colr&0x7f;
        colr++;
    }
```

该程序首先调用 0 号功能，对鼠标器复位并检查，若出口参数为 0，则表示鼠标器或鼠标驱动程序未安装(即出现信息：Mouse or Mouse Driver Absent)，并返回系统。若成功，则调用 12 号功能，设置中断条件，并设中断服务程序地址。接着调用 DOS int 31H 中断，将程序驻留内存，设程序长度 size 等于 600，这样每当按下鼠标器右按钮，便产生中断，使屏幕改变颜色。

若考虑到关于 DOS 重入问题，可采用如下的程序来避免鼠标器热键按下时引起 DOS 的重入。在该程序中判断当 DOS 不忙和没有磁盘操作，且进入 1CH 定时器中断后按下鼠标器键时，便产生中断。DOS 的忙标志地址由 DOS 的 34H 功能调用得到，故在 1CH 中断中用 peekb()来取得忙标志，以判别 DOS 忙否。当按鼠标器键时，便产生 0CH 中断。故当产生 0CH 中断时，若有键按下，便置按键标志 press=1。为了鼠标热键不与其它程序冲突，该程序使用了特殊的热键激活方法，即当按下鼠标键 1 秒后，才激活 TSR。因 1CH 中断每秒产生 18.2 次，故当执行 counter>20 时，才激活 TSR。

```
#include<dos.h>
#include<stdio.h>
#include<alloc.h>
static union REGS rg;
static struct SREGS seg;
unsigned size=600;
unsigned osseg;
static unsigned busy;
static void interrupt(*old1c)();
static void interrupt(*oldcom)();
static void interrupt(*old13)();
void interrupt new1c();
void interrupt newcom();
void interrupt new13();
void mouse_key();
static int tsr=0;
static int diskbusy=0;
main()
{
  int mess;
  rg.x.ax=0;
  int86(51,&rg,&rg);                    /*鼠标器初始化调用*/
  mess=rg.x.ax;
  if(mess==0){
    printf("Mouse or Mouse Driver Absent\n");
    delay(1000);
```

```
    exit(1);
    }
  init();
  rg.x.ax=0x3100;
  rg.x.dx=size;
  intdos(&rg,&rg);                    /*程序驻留调用*/
  }
void mouse_key()
{
  int j,k;
  static int colr=0;
  char far *vsg=0xb8000000;           /*显示存储器首地址*/
  for (j=0;j<50;j++)
   for(k=0;k<80;k++)
     *(vsg+j*80*2+k*2+1)=colr&0x7f;
    colr++;
}
  int hot=0;
  int press=0;
  int counter=0;
void interrupt newcom()               /*RS232通信中断*/
  {
   (*oldcom)();                       /*调用旧的RS232中断*/
    rg.x.ax=3;
    int86(0x33,&rg,&rg);
    if(rg.x.bx==2)
     {press=1;
      counter=0;
      return;
     }
    if(rg.x.bx==0&&press)
    if(counter>20)
     {
      press=0;
      if(!tsr)
      hot=1;
     }
    }
  void interrupt new1c()              /*新的1C中断*/
   {
    (*old1c)();
    counter++;
    if(hot&&peekb(osseg,busy)!=0)
    if(diskbusy==0)
```

```
    {
     outportb(0x20,0x20);
     hot=0;
     mouse_key();
    }
   }
init()                                  /*设置新的中断向量,保存旧的*/
 {
  segread(&seg);
  rg.h.ah=0x34;
  intdos(&rg,&rg);
  osseg=_ES;
  busy=rg.x.hx;
  old1c=getvect(0x1c);
  old13=getvect(0x13);
  oldcom=getvect(0x0c);
  setvect(0x1c,new1c);
  setvect(0x13,new13);
  setvect(0x0c,newcom);
 }
 void interrupt new13(bp,di,si,ds,es,dx,cx,bx,ax,ip,cs,flgs)
                                        /*得到磁盘是否忙信息中断*/
 {
  diskbusy++;
  (*old13)();
  --diskbusy;
 }
```

除了利用 INT 33H 的功能调用对应的鼠标动作中断来激活 TSR 程序外，还可以利用 INT 0BH 或 INT 0CH 中断来用鼠标动作激活 TSR 程序。因当鼠标器和 COM1 相接，每当鼠标器操作时，便产生 0CH 中断，即 RS232 向主机传送一次数据。当鼠标器和 COM2 相接时，产生 0BH 中断，因此用户可以设置自己的 0CH 中断或 0BH 中断服务程序，当产生该中断时，检查鼠标按钮操作，一旦有规定的按钮动作，立即激活 TSR 驻留程序。这种激活 TSR 程序的方法比较单一，若在驻留程序中再使用鼠标器操作，就不允许。所以最好的方法还是使用 INT 33H 的 12、20、24 等有中断程序功能的调用，而最普遍使用的是 12 号功能调用。

第 10 章

Turbo C 作图

计算机图形程序设计是程序设计中较难且又最吸引人的部分。为了用户设计图形程序方便，不同版本和公司出的 C 编译系统都提供了许多画图的库函数，用户设计图形程序时，只要在需要的地方，设置相应的参数对其调用即可(这些画图库函数不是 C 语言标准所要求的，所以一般 C 语言程序设计课本中都没作介绍)。

Turbo C 为用户提供了一个功能很强的画图软件库，它又称为 BorLand 图形接口(BGI)，它包括图形库文件(graphics.lib)，图形头文件(graphics.h)和许多图形显示器(图形终端)的驱动程序(如 CGA.BGI、EGAVGA.BGI 等)。还有一些字符集的字体驱动程序(如 goth.chr 黑体字符集等)。编写图形程序时用到的一些图形库函数均在 graphics.lib 中，执行这些函数时，所需的有关信息(如宏定义等)则包含在 graphics.h 头文件中。因此用户在自己的画图源程序中必须包括 graphics.h 头文件，在进行目标程序连接时，要将 graphics.lib 连接到自己的目标程序中去，关于这方面的内容，请看图形程序运行的条件一节。

由于计算机画图涉及到显示器和驱动它们工作的图形适配器(卡)等许多硬件知识和图形的一些基本概念、定义等，因而有必要简单地介绍一下：

10.1 图形显示的坐标和象素

10.1.1 图形显示的坐标

显示器的屏幕如同一张坐标纸，在其上显示图形时，图形上任一点的位置均有确定的坐标，即可用 x，y 坐标值来表示。显示屏的坐标系统如图 10.1 所示，定义屏幕的左上角为其原点，正 x 轴右延伸，正 y 轴向下延伸，如同一个倒置的直角坐标系，其 x 和 y 均为≥0 的整数值，其最大值则由显示器的类型和显示方式来确定，这将在后面讲述。这种显示坐标我们称为屏幕显示的物理坐标或绝对坐标，以便和后面将要提到的图视窗口(图视口)坐标相区别，图视窗口是指在物理坐标区间又开辟一个或多个区间，在这些区间又可定义一个相对坐标，以后画图均可在此区间进行，以相对坐标来定义位置。如图 10.2 所示，当定义了一个左上角坐标为(200,50)，右下角坐标为(400,150)的一个区域为图视口，则以后处理图形时，就以其左上角为坐标原点(0,0)，右下角为坐标(200,100)的坐标系来定位图形上各点位置，关于这方面内容请看图视口设置函数。

10.1.2 象素

我们看电视时，屏幕上显示的画面，在走近看时，会发现均由一些圆点组成(其亮度、颜色不同)，这些点称为象素(或称象点)，它们是组成图形的最小单位，显示器显示的图形也是

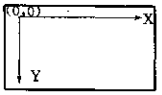


图 10.1 显示屏的坐标系统

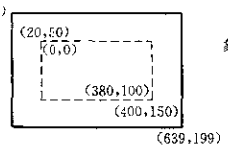


图 10.2 相对坐标

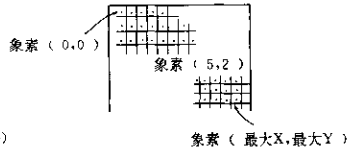


图 10.3 不同位置象素的坐标

由象素组成，不过象素的大小可以通过设置不同的显示方式来改变。象素在屏上的位置则可由其所在的 x,y 坐标来决定，例如在图 10.2 所示的显示方式下，x,y 最大坐标是(639,199)，即满屏显示的象素个数为 640×200。图 10.3 示出了不同位置象素的坐标，其最大的 x,y 值(即行和列值)由程序设置的显示方式来决定。满屏显示象素多少，则决定了显示的分辨率高低，可以看出象素越小(或个数越多)，则显示的分辨率越高。

10.2 图形显示器与适配器

计算机中要显示的字符和图形均以数字形式存储在存储器中，而显示器接收的应是模拟信号，例如常用的显示器有三条模拟红绿兰颜色的模拟信号输入线，每条输入线的电压决定了颜色的亮度，只要能产生出可区分的电压来，它们不同的组合，便可使显示器显示出不同的颜色来。插在 PC 微机插槽中的图形卡(即适配器)，其作用就是将要显示的字符和图形以数字形式存储在卡上的视频存储器 VRAM 中，再将其变成视频模拟信号送往相应适配的显示器进行显示，也即适配器在计算机主机和显示器之间起到了信息转换和视频发送作用，由于计算机配有的显示器种类不同，因而适配器种类不同，图 10.4 示出了一般 386,486 等 PC 微机中适配器、主机、显示器之间的关系图。

图 10.4 适配器、主机、显示器的关系

由于 PC 微机配有的显示器种类不同，因而适配器也就不同，也就是说不同的适配器配不同的显示器，反过来说也行。现就目前常用的几种适配器作以下介绍：

1. 单色显示适配器(MDA)

仅显示一种颜色，仅支持 80×25 行的字符显示。

2. 彩色图形适配器(CGA)

这是 PC/XT 等微机配用的显示器图形卡，它可以产生单色或彩色字符和图形。

在图形方式下，Turbo C 支持两种分辨率供选择，一种为高分辨方式(CGAHI)，象素数

为 640×200，这时背景色是黑的(当然也可重新设置)，前景色可供选择，但前景色只是同一种，因而图形只显示两色。

另一种为中分辨显示方式，象素数为 320×200，其背景色和前景色均可由用户选择，但仅能显示四种颜色。在该显示方式下，可有四种模式供选择，即 CGAC0，CGAC1，CGAC2，CGAC3，它们的区别是显示的 4 种颜色不同，更详细的信息，请看图形系统的初始化。

3. 增强型图形适配器(EGA)

该适配器与之配接的相应显示器，除支持 CGA 的四种显示模式外，还增加了分辨率为 640×200 的 16 色显示方式，，Turbo C 称为 EGALO(EGA 低分辨显示方式)和 640×350 的 EGA 高分辨显示方式(EGAHI)，也可显示 16 色。

4. 视频图形阵列适配器(VGA)

它是目前流行的 PC 微机显示标准，它支持 CGA、EGA 的所有显示方式，但自己还有 640×480 的高分辨显示方式(VGAHI)、640×350 的中分辨显示方式(VGAMED)和 640×200 的低分辨显示方式(VGALO)，它们均可有 16 种显示颜色可供选择。

5. TVGA

它是目前市面上最流行的 PC 286，386，486，586 微机配的显示器标准，在图形方式下，它可有 640×400，640×480，800×600，1024×768 及 768×1024 等分辨率，可选颜色达 256 种，在文本方式下，可支持 25、30、43、60 行，132 列的字符显示。它也兼容 CGA、EGA、VGA 的显示方式，由于 Turbo C 早于该产品出现，因而上述增强了的显示方式均不支持，但对于高级编程者，可通过对 TVGA 各专用寄存器直接编程来实现这些功能。

6. PVGA

国内有部分高档微机(如 ALR 公司，SUN 公司，香港 Superking 等公司生产的 386，486 微机)配有称 PVGA 的显示适配器，它是 paradise VGA 的缩写，它由 Western Digital 公司生产，有的叫 SVGA(香港 Surperking 公司生产的 386 微机上使用)。还有称为 EVGA(Extended VGA)，它们和 PVGA 在软硬件上均是兼容的。PVGA 完全兼容 CGA、EGA、VGA 的所有工作方式，它还有自己一些特有的工作模式，与 TVGA 类似。

PVGA 结构及显示原理与 TVGA 相似。

7. XGA

XGA 是 IBM 公司新推出的一种增强性图形适配器，它是 VGA 的换代产品，具有更高的分辨率和性能，它有和 CGA，EGA，VGA 全兼容的工作方式，但显示速度却比 VGA 快一倍。

8. CVGA

这是国内研制的带有显示汉字功能的 VGA 图形适配器，它带有汉字字库，该适配器提供了和 CGA、EGA、VGA 全兼容的工作方式，还提供了在文本工作方式下的高分辨图形显示功能。在它的 ROM BIOS 中提供了 INT 10H 的功能号为 30H—38H 的功能调用，因而提供了方便的汉字处理功能。由于该图形适配卡上又带有汉卡功能，因而使用汉字时，不必从 CCDOS 或 CCBIOS 2.13 以及 UCDOS 中再调字库到内存去，这样大大节省了内存和提高了速度。

众多生产厂家推出了许多性能优于 VGA 但名字各异的图形显示系统，美国标准协会制定了这样的系统应具有的主要性能标准，常将属于这类的显示适配卡统称为 SVGA(即

Super VGA)。

10.3 显示器工作方式

显示器有两种工作方式，即文本方式或称字符显示方式和图形显示方式，它们的主要差别是显示存储器(VRAM)中存的信息不同。字符方式时，VRAM 存放要显示字符的 ASCII 码，用它作为地址，取出字符发生器 ROM(固定存储器)中存放的相应字符的图象(又称字模)，变成视频信号在显示器屏上进行显示。而当选择图形方式时，则要显示的图形的图象直接存在 VRAM 中，VRAM 中某地址单元存放的数就表示了相应屏幕上某行和列上的象素及颜色。如在 CGA 中分辨图形方式下，每字节代表 4 个象素，即每 2 位表示一个象素及颜色。字符方式和 CGA 中分辨图形方式显示的示意图如图 10.5 所示。

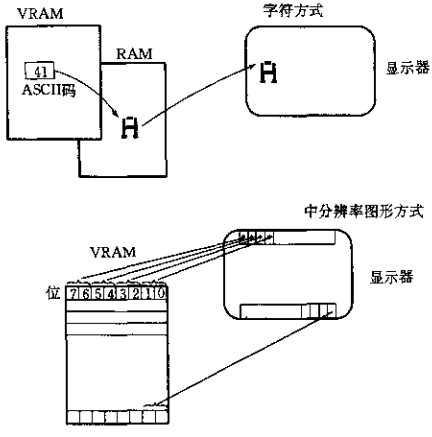


图 10.5 VRAM 和字符及图形显示关系

在文本方式下，EGA、VGA 可以使用几种字符集，如 EGA 下有三种字符集，VGA 有五种字符集，它们都存在 EGA、VGA 的 BIOS(EGA、VGA 基本输入输出系统)的 ROM 中。使用时，由 BIOS 将字符集调入 VRAM 指定的区域中，并驻留在该区域中，当要显示字符时，则由该字符的 ASCII 码作为地址，再找到 VRAM 中存放该字符的字模(图象)进行显示，图 10.6 显示了 C 的 8×8 点阵字模，关于字模的问题，可参阅第十四章。

10.4 Turbo C 支持的适配器和图形模式

Turbo C 支持的各种显示器适配器和图形模式如表 10.1 所示，其中几种适配器前面未作介绍，这是因为国内用户较少，使用而不大之故，现简单予以介绍：

1. MCGA(多色图形阵列)

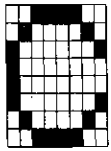


图 10.6　C 的 8×8 点阵字模

它和 VGA 是 PS/2 系列微机的主要适配器，MCGA 功能上同 CGA 相似，它和 CGA 是部分兼容的，但有些功能又和 VGA 类似，可有 640×480 的二色模式。

该适配器可以和单色适配器同插于微机中。

PS/2 是 IBM 公司 1987 年推出的一种微机系统，PS/2(Personal System/2)意为个人系统。

2. IBM8514

该适配器可插入 PS/2 微机中，它有较强的图形功能，最高分辨率可达 1024×768，可有 256 种颜色。

3. HERC

大力神公司生产的一种单色适配器，是 PC 系列微机早期的第三种显示标准(其它两种是 MDA 和 CGA)，它采用 720×348 高分辨单色显示模式。

4. EGAMONO

EGA 单色适配器，它只有一种 640×350 单色显示模式。

其它的几种适配器不多见，Turbo C 支持的适配器及其显示模式可参阅表 10.1。

表 10.1　Turbo C 支持的适配器和图形模式

适配器 Driver	模式 Mode	分辨率	颜色数	页数	标示符
CGA	0	320×200	4	1	CGAC0
	1	320×200	4	1	CGAC1
	2	320×200	4	1	CGAC2
	3	320×200	4	1	CGAC3
	4	640×200	2	1	CGAHI
EGA	0	640×200	16	4	EGALO
1	640×350	16	2	EGAHI	
EGA64	0	640×200	16	1	EGA64LO
	1	640×350	4	1	EGA64NI
EGAMONO	0	640×350	2	1	EGAMONOHI
VGA	0	640×200	16	2	VGALO
	1	640×350	16	2	VGAMED
	2	640×480	16	1	VGAHI
MCGA	0	320×200	4	1	MCGA0

续表

适配器 Driver	模式 Mode	分辨率	颜色数	页数	标示符
	1	320×200	4	1	MCGA1
	2	320×200	4	1	MCGA2
	3	320×200	4	1	MCGA3
	4	640×200	2	1	MCGAMED
	5	640×480	2	1	MCGAHI
HERC	0	720×348	2	1	HERCMONOHI
ATT400	0	320×200	4	1	ATT400C0
	1	320×200	4	1	ATT400C1
	2	320×200	4	1	ATT400C2
	3	320×200	4	1	ATT400C3
	4	640×200	2	1	ATT400MED
	5	640×400	2	1	ATT400HI
PC3270	0	720×350	2	1	PC3270HI
IBM8514	0	640×480	256		IBM8514LO
	1	1024×768	256		IBM8514HI

上面介绍了一些编制图形程序的预备知识，下面将介绍编制图形程序的各种库函数和相应的一些知识。

10.5 图形系统的初始化

在编制图形程序时，进入图形方式前，首先要在程序中对使用的图形系统进行初始化，即要用什么类型的图形显示适配器的驱动程序，采用什么模式的图形方式(也就是相应程序的入口地址)，以及该适配器驱动程序的寻找路径名。所用系统的显示适配器一定要支持所选用的显示模式，否则将出错。Turbo C 提供了一个图形系统初始化函数 initgraph 可完成这些功能。

10.5.1 图形系统的初始化函数

它的说明原型是：

```
void far initgraph(int far *driver,int far *mode, char far path-for-driver);
```

当我们使用的存储模式为 tiny(微型)、small(小型)或 medium(中型)时，不需要远指针，因而可以将初始化函数调用格式写成如下形式(该说明适用于后面所述的任一函数)：

```
initgraph(&graphdriver,&graphmode," ");
```

其中驱动程序目录路径为空字符" "时，表示就在当前目录下，参数 graphmode 用如表 10.1 所示的模式号或标示符来定义。参数 graphdriver 是一个枚举变量，它属于显示器驱动程序的枚举类型：

```
enum graphics_ driver {DETECT, CGA, MCGA, EGA, EGA64, EGAMONO, IBM
```

8514,HERCMONO,ATT400,VGA,PC3270};

其中枚举成员的值顺序为：DETECT 为 0,CGA 为 1,依次类推。除 DETECT 外,其它几种对应的适配器我们已作了介绍。当我们不知道所用显示适配器名称时,可将 graphdriver 设成 DETECT,它将自动检测所用显示适配器类型,并将相应的驱动程序装入,并将其最高的显示模式作为当前显示模式,如下面所列：

检测到的适配器	选中的显示模式
CGA	4(640×200,2 色即 CGAHI)
EGA	1(640×350,16 色,即 EGAHI)
VGA	2(640×480,16 色,即 VGAHI)

一旦执行了初始化,显示器即被设置成相应模式的图形方式。下面是某画图程序的开始部分,它包括对图形系统的初始化：

```
#include<graphics.h>
main()
 { int graphdriver=DETECT;
   int graphmode;
   initgraph(& graphdriver, & graphmode,"");
             ⋮
```

上面初始化过程中,将由 DETECT 检测所用适配器类型,并将当前目录下相应的驱动程序装入,并采用最高分辨率显示模式作为 graphmode 的值。如检测到为 CGA 适配器时,则 graphmode 等于 4 或为 CGAHI,若检测到 VGA 适配器,则 graphmode 等于 2 或为 VGAHI。

若已知所用图形适配器为 VGA 时,想采用 640×480 的高分辨显示模式 VGAHI,则图形初始化部分可写成：

```
int graphdriver=VGA;
int graphmode=VGAHI;
initgraph(&graphdriver,&graphmode,"");
      ⋮
```

其中参数(路径名)""表示就在当前目录下。

10.5.2 图形系统检测函数

当 graphdriver=DETECT 时,实际上 initgraph 函数又调用了图形系统检测函数 detectgraph,它完成对适配器的检查并得到显示器类型号和相应的最高分辨率模式,若所设适配器不是规定的那些类型,则返回−2,表示适配器不存在,该函数的原型说明是：

```
void far detectgraph(int far *graphdriver,int far * graphmode);
```

当想检测所用的适配器类型,但并不想用其最高分辨率显示模式,而想由自己进行控制使用时,可采用这个函数来实现。例如：

```
detectgraph(&graphdriver,&graphmode);
switch(graphdriver){
```

```
    case CGA: graphmode=1;   /*设置成低分辨模式*/
              break;
    case EGA: graphmode=0;   /*设置成低分辨模式*/
              break;
    case VGA: graphmode=1;   /*设置成中分辨模式*/
              break;
    case -2:  printf("\n Graphics adapter not installed");
              exit(1);
    default:  printf("\n Graphics adapter not is CGA, EGA, or VGA");
    }
    initgraph(&graphdriver,&graphmode,"");
          ⋮
```

调用 detectgraph 时，该函数将把检测到的适配器类型赋予 graphdriver，再把该类型适配器支持的最高分辨率模式赋给 graphmode。

当图形系统初始化后，后面将要进行的画图操作均采用缺省值作为参数的当前值，如画图屏幕为全屏，当前开始画图坐标为(0,0)(又称当前画笔位置，虽然这个笔是无形的)，采用画图的背景颜色和前景颜色，图形的填充方式，以及可以采用的字符集(字库)等均为缺省值。当在程序某处需要改变时，可采用相应的函数进行重新设置相应的参数，后面将分别要进行介绍。

10.5.3 清屏和恢复显示方式的函数

1. 清屏函数

画图前一般需清除屏幕，使得屏幕如同一张白纸，好画最新最美的图画，因而必须使用清屏函数。它的说明原型是：

```
void far cleardevice(void);
```

该函数作用范围为整个屏幕，如果用函数 setviewport 定义一个图视窗口，则可用清除图视口函数，它仅清除图视口区域内的内容，该函数的说明原型是：

```
void far clearviewport(void);
```

2. 恢复显示方式函数

当画图程序结束，回到文本方式时，要关闭图形系统，回到文本方式，该函数的说明原型是：

```
void far closegraph(void);
```

由于进入 C 环境进行编程时，即进入文本方式，因而为了在画图程序结束后恢复原来的最初状况，一般在画图程序结束前调用该函数，使其恢复到文本方式。

为了不关闭图形系统，使相应适配器的驱动程序和字符集(字库)仍驻留在内存，但又回到原来所设置的模式，则可用恢复工作模式函数，它也同时进行清屏操作，它的说明原型是：

```
void far restorecrtmode(void);
```

该函数常和另一设置图形工作模式函数 setgraphmode 交互使用，使得显示器工作方式在图形和文本方式之间来回切换，这在编制菜单程序和说明程序时很有用处。

10.6 基本图形函数

图形由点、线、面组成，Turbo C 提供了一些函数，以完成这些操作，而所谓面则可由对一封闭图形填上颜色来实现。

10.6.1 画点函数

该函数的说明原型是：

void far putpixel(int x,int y,int color);

它表示在指定的 x,y 位置画一点，点的显示颜色由设置的 color 值决定，关于颜色的设置，将在设置颜色函数中介绍。

该函数的相对应函数是取象素值函数，它的说明原型是：

int far getpixel(int x,int y);

该函数将得到在(x,y)点位置上的象素的颜色值。

例：

下面是一个画点的程序，它将在 y=20 的恒定位置上，沿 x 方向从 x=200 开始，连续画两个点(间距为 4 个象素位置)，又间隔 16 个点位置，再画两个点，如此循环，直到 x=300 为止，每画出的两个点中的第一个由 putpixel(x,20,1)所画，第二个则由 putpixel(x+4,20,2)画出，颜色值分别设为 1 和 2，它的含义将在颜色设置函数中介绍。

```
#include<graphics.h>
main()
  {
    int graphdriver=CGA;
    int graphmode=CGACO,x;
    initgraph(&graphdriver,&graphmode,"");
    cleardevice();
    for (x=20,x<=300;x+=16)
     {
       putpixel(x,20,1);
       putpixel(x+4,20,2);
     }
      getch();
      closegraph();
  }
```

为了观查点方便，设置了适配器类型为 CGA，CGACO 中分辨显示模式。由于 VGA 和它兼容，因此若显示器为 VGA，此时它即以此兼容的仿真形式显示。第一个点显示颜色为绿，第二个点为红。

10.6.2 有关画图坐标位置的函数

在屏幕上画线时，如同在纸上画线一样。画笔要放在开始画图的位置，并经常要抬笔移

动，以便到另一位置再画。我们也可想象在屏上画图时，有一无形的画笔，可以控制它的定位、移动(不画)，也可知道它能移动的最大位置限制等。完成这些功能的函数是：

① 移动画笔到指定的(x,y)位置，移动过程不画：

void far moveto(int x,int y)；

② 画笔从现行位置(x,y)处移到一位置增量处(x+dx,y+dx)，移动过程不画：

void far moverel(int dx,int dy)；

③ 得到当前画笔所在位置

int far getx(void)；

得到当前画笔的 x 位置

int far gety(void)；

得到当前画笔的 y 位置

10.6.3 画线函数

这类函数提供了从一个点到另一个点用设定的颜色画一条直线的功能，起始点的设定方法不同，因而有下面不同的画线函数：

① 两点之间画线函数

void far line(int x0,int y0,int x1,int y1)；

从(x0,y0)点到(x1,y1)点画一直线。

② 从现行画笔位置到某点画线函数

void far lineto(int x,int y)；

将从现行画笔位置到(x,y)点画一直线。

③ 从现行画笔位置到一增量位置画线函数

void far linerel(int dx,int dy)；

将从现行画笔位置(x,y)到位置增量处(x+dx,y+dy)画一直线。

下面的程序将用 moveto 函数将画笔移到(100,20)处，然后从(100,20)到(200,20)用 lineto 函数画一直线。再将画笔移到(200,20)处，用 lineto 画一直线到(100,80)处，再用 line 函数在(100,90)到(200,90)间连一直线。接着又从上次 lineto 画线结束位置开始(它是当前画笔的位置)，即从(100,80)点开始到 x 增量为 0，y 增量为 20 的点(100,100)为止用 linerel 函数画一直线。moverel(－100,0)将使画笔从上次用 linerel(0,20)画直线时的结束位置(100,100)处开始移到(100－100,100－0)，然后用 linerel(30,20)从(0,100)处再画直线至(0+30,100+20)处。

注意！用 line 函数画直线时，将不考虑画笔位置，它也不影响画笔原来的位置，lineto 和 linerel 要求画笔位置，画线起点从此位置开始，而结束位置就是画笔画线完后停留的位置，故这两个函数将改变画笔的位置。

```
#include<graphics.h>
main()
  {
    int graphdriver=VGA;
    int graphmode=VGAHI;
```

```
    initgraph(&graphdriver,&graphmode,"");
    cleardevice();
    moveto(100,20);
    lineto(100,80);
    moveto(200,20);
    lineto(100,80);
    line(100,90,200,90);
    linerel(0,20);
    moverel(-100,0);
    linerel(30,20);
    getch();
    closegraph();
}
```

10.6.4 画矩形和条形图函数

画矩形函数 rectangle 将画出一个矩形框，而画条形函数 bar 将以给定的填充模式和填充颜色画出一个条形图来，而不是一个条形框，关于填充模式和颜色将在后面介绍。

① 画矩形函数

void far rectangle(int x1,int y1,int x2,int y2);

该函数将以(x1,y1)为左上角，(x2,y2)为右下角画一矩形框。

② 画条形图函数

void bar(int x1,int y1,int x2,int y2);

该函数将以(x1,y1)为左上角，(x2,y2)为右下角画一实形条状图，没有边框，图的颜色和填充模式可以设定。若没有设定，则使用缺省模式。

下面的程序将由 rectangle 函数以(100,20)为左上角，(200,50)为右下角画一矩形，接着又由 bar 函数以(100,80)为左上角，(150,180)为右下角画一实形条状图，用缺省颜色(白色)填充。

```
#include<graphics.h>
main()
  {
    int graphdriver=DETECT;
    int graphmode,x;
    initgraph(&graphdriver,&graphmode,"");
    cleardevice();
    rectangle(100,20,200,50);
    bar(100,80,150,180);
    getch();
    closegraph();
  }
```

10.6.5 画椭圆、圆和扇形图函数

这些函数将用于画椭圆、圆和扇形图。

① 画椭圆函数

void ellipse(int x,int y,int stangle,int endangel, int xradius,int yradius);

该函数将以(x,y)为中心,以 xradius 和 yradius 为 x 轴和 y 轴半径,从起始角 stangle 开始到 endangle 角结束,画一椭圆线。当 stangle=0,endangle=360 时,则画出的是一个完整的椭圆,否则画出的将是椭圆弧。关于起始角和终止角规定如图 10.7 所示,下面将要解释。

② 画圆函数

void far circle(int x,int y,int radius);

该函数将以(x,y)为圆心,radius 为半径画一个圆。

③ 画圆弧函数

void far arc(int x, int y, int stangle,int endangle, int radius);

该函数将以(x,y)为圆心,radius 为半径,从 stangle 为起始角开始,到 endangle 为结束角,画一圆弧。

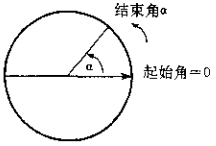


图 10.7 起始角和终止角

上面提到的画椭圆和画圆弧函数中以及以后的函数中,有关于角的概念。在 Turbo C 中是这样规定的:屏的 x 轴方向为 0 度,当半径从此处逆时针方向旋转时,则依次是 90 度、180 度、270 度,当 360 度时,则和 x 轴正向重合,即旋转了一周。如图 10.7 所示。

④ 画扇形图函数

void far pieslice(int x, int y, int stangle, int endangle,int radius);

该函数将以(x,y)为圆心,radius 为半径,从 stangle 为起始角,endangle 为结束角,画一扇形图,扇形图的填充模式和填充颜色可以事先设定,否则以缺省模式进行。

例:

该程序将用 ellipse 函数画椭圆,从中心为(320,100),起始角为 0 度,终止角为 360 度,x 轴半径为 75,y 轴半径为 50 画一椭圆,接着用 circle 函数以(320,220)为圆心,以半径为 50 画圆。然后分别用 pieslice 和 ellipse 及 arc 函数在下方画出了一扇形图和椭圆弧及圆弧。

```
#include<graphics.h>
main()
  {
    int graphdriver=DETECT;
    int graphmode,x;
    initgraph(&graphdriver,&grapbmode,"");
    cleardevice();
    ellipse(320,100,0,360,75,50);
    circle(320,220,50);
    pieslice(320,340,30,150,50);
```

```
    ellipse(320,400,0,180,100,35);
    arc(320,400,180,360,50);
    getch();
    closegraph();
}
```

10.7 颜色控制函数

象素的显示颜色，或者说画线、填充面的颜色都可以用一些函数来设置，否则将采用缺省的值，显示点、线、面的颜色，称为前景色，而衬托它们的背景，称为背景色。按照CGA、EGA、VGA图形适配器的硬件结构，颜色可以通过对其内部相应的寄存器进行编程来改变，但不编程时，则取缺省值，这就是我们上面所列举程序的情况。

为了能形象地说明颜色的设置，一般用所谓调色板来进行描述，它实际上对应一些硬件的寄存器，确有实体存在。从C语言的角度看，调色板实际上就是一张颜色索引表，对CGA显示器，在中分辨显示方式下，有4种显示模式，每一种模式对应有一个调色板，可用调色板号区别。每个调色板有4种颜色可以选择，颜色可以用颜色值0、1、2、3来进行选择，由于CGA有四个调色板，一旦显示模式确定后，调色板即确定，如选CGAC0模式，则选0号调色板，但选调色板的哪种颜色则可由用户根据需要从0、1、2和3中进行选择，表10.2就列出了调色板与对应的颜色值。

表10.2 CGA的调色板号与对应的颜色值

模式	调色板号	颜色值			
		0	1	2	3
CGAC0	0	背景色	绿	红	黄
CGAC1	1	背景色	青	洋红	白
CGAC2	2	背景色	淡绿	淡红	棕
CGAC3	3	背景色	淡青	淡洋红	淡灰

表中所指若选调色板的颜色值为0，表示此时选择的颜色和当时的背景色一样，即前景、背景同为一色。图10.8表示了当调色板确定后，选颜色值，这时象素即以此颜色进行显示。

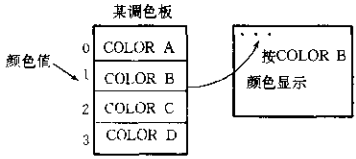


图10.8 CGA中分辨模式显示

10.7.1 颜色设置函数

Turbo C 提供了一个前景颜色设置函数 setcolor

① 颜色设置函数

该函数的原说明为：

void far setcolor(int color)；

该函数将使得前景以所选 color 颜色进行显示，对 CGA，当为中分辨模式时只能选 0，1，2，3。

显示图形的背景色可用 setbkcolor 函数来选择，其颜色可从 16 种中选 1。

② 选择背景颜色的函数

void far setbkcolor(int color)

该函数将使得背景色按所选16种中的一种color颜色进行显示，表10.3列出了颜色值color对应的颜色，此函数使用时，color既可用值表示，也可用相应的大写颜色名来表示。

表 10.3 背景色值与对应的颜色名

颜色值	颜色名	颜色	颜色值	颜色名	颜色
0	BLACK	黑	8	DARKGRAY	深灰
1	BLUE	蓝	9	LIGHTBLUE	淡蓝
2	GREEN	绿	10	LIGHTGREEN	淡绿
3	CYAN	青	11	LIGHTCYAN	淡青
4	RED	红	12	LIGHTRED	淡红
5	MAGENTA	洋红	13	LIGHTMAGENTA	淡洋红
6	BROWN	棕	14	YELLOW	黄
7	LIGHTGRAY	浅灰	15	WHITE	白

例：

下面的程序用 initgraph 设置为 CGA 显示器，显示模式为 CGAC0，再用 setcolor 选择显示颜色，由于 color 选为 1，这样图形将选用 0 号调色板的绿色显示，因而用 line 函数将画出一条绿色直线来，背景色由于没设置，故为缺省值，即黑色。当按任一键后，执行 setbkcolor，设置背景色为蓝色(也可用 1)，这时用 line 画出的(20,40)到(150,150)的线仍为绿色，但背景色变为蓝色。当再按一键后，程序往下执行，这时用 setcolor 又设显示前景颜色，颜色号为 0，表示画线选背景色，此时用 line(60,120,220,220)画出的线将显不出来，因前景、背景色一样，混为一体。

```
#include<graphics.h>
main()
  {
    int graphdriver=CGA;
    int graphmode=CGAC0;
    initgraph(&graphdriver,&grapbmode,"");
```

```
    cleardevice();
    setcolor(1);
    line(0,0,100,100);
    getch();
    setbkcolor(BLUE);
    line(20,40,150,150);
    getch();
    setcolor(0);
    line(60,120,220,220);
    getch();
    closegraph();
}
```

应该提起注意的是，setbkcolor 函数的参数 color 和 setcolor 函数中的 color 不是同一个含义，前者只能选表 10.3 的 16 色之一，而 setcolor 只能选表 10.2 的颜色值，该值对于不同的调色板所表示的颜色不同。可以这样理解，当用 setbkcolor 选了 16 种之一的颜色作背景色后，该颜色就放到调色板的颜色值为 0 处，即改变了调色板颜色值为 0 时代表的颜色。

对于 640×200 高分辨显示模式 CGAHI，颜色只能选 0 或 1，当选其它值时，仍作为 1，即两色显示，当然背景色可选 16 种之一，前景色为白色。若用 EGA 或 VGA 显示器仿真 CGA，上述的说法是正确的。即在上述显示器上选择 CGA 兼容方式，令 graphdriver＝CGA，graphmode＝CGAHI，是正确的。但用在真正的 CGA 显示器上，如 PC/XT 机的显示器上，设置的背景色被用作前景色，而用 setcolor 设置的前景色却用作背景色，正好相反，这是由于两种显示器硬件结构不同，而 Turbo C 设置函数时，只考虑到 EGA、VGA 的颜色设置正确，没有兼顾到 CGA 硬件结构特点，因此我们使用不同显示器时，对 CGA 方式的 CGAHI 显示模式的编程要注意到这点。

10.7.2 调色板颜色的设置

① 调色板颜色的设置函数

void far setpalette(int index,int actual_color)

该函数用来对调色板进行颜色设置，由于 CGA 显示器在中分辨方式下，4 个调色板对应的 4 个颜色值是固定的，即从硬件角度看，是固定不变的，因而这个函数一般不用在 CGA 上。但若令 index 为 0，则表示改变 0 号颜色、即背景色。对于 index 的其它值，该函数将无意义。该函数一般用在 EGA、VGA 显示方式上

对 EGA、VGA 显示器，只有一个调色板，但这个调色板有 16 个调色板寄存器，它们存的内容对 EGA 和 VGA 含义不同，下面分别予以介绍：

对 EGA 显示器，调色板即 16 个调色板寄存器是一个颜色索引表，它存有 16 种颜色，VRAM 中的每个象素值（是 4 位）实际上代表一个颜色索引号。由该值即上述函数的参数 index 可知道选中哪个调色板寄存器，而每个调色板寄存器寄存了一种颜色。由于调色板寄存器为 6 位，故可用 6 位二进制数表示一个颜色，因而可有 $2^6=64$ 种颜色供选择。该颜色值即是上述参数 octual_color，图 10.9 所示就是每个调色板寄存器代表的颜色。为 0，则闭断该颜色，为 1 则接通。因而它们不同的组合可产生 64 种颜色。对 EGA 图形系统初始化时，

16 个调色板寄存器已装入确定的颜色，如表 10.4 所列，也称为标准色。显示模式不同，所装颜色值也不同，表 10.4 是指 EGAHI 或 VGAHI 模式下的标准色。

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
X	X	R′	G′	B′	R	G	B
		淡红	淡绿	淡蓝	红	绿	蓝

图 10.9　调色板寄存器六位数代表的颜色

表 10.4　16 个调色板寄存器对应的标准色和值

调色板寄存器号	颜色名	值	调色板寄存器号	颜色名	值
0	EGA_BLACK	0	8	EGA_DARKGRAY	56
1	EGA_BLUE	1	9	EGA_LIGHTBLUE	57
2	EGA_GREEN	2	10	EGA_LIGHTGREEN	58
3	EGA_CYAN	3	11	EGA_LIGHTCYAN	59
4	EGA_RED	4	12	EGA_LIGHTRED	60
5	EGA_MAGENTA	5	13	EGA_LIGHTMAGENTA	61
6	EGA_BROW	20	14	EGA_YELLOW	62
7	EGA_LIGHTGRAY	7	15	EGA_WHITE	63

图 10.10 表示了 EGA 中象素显色过程，VRAM 中 3 个象素值分别为 1001、0000 和 1111，即 index 值(或调色板寄存器号)分别为 9、0 和 15。它们所存颜色分别为黄、黑和白，因而三个象素点在显示屏上分别显示亮蓝、黑和白，实际上第二个总不显示(因和背景色同，前题是，背景色也是黑的)。

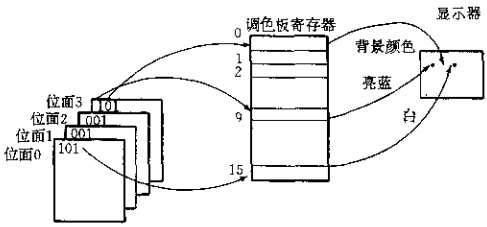


图 10.10　EGA 中象素显色过程

当编制动画或菜单等高级程序时，系统图形初始化时常需要改变每个调色板寄存器的颜色设置，这时就可用 setpalette 函数来重新对某一个调色板寄存器颜色进行再设置。

对于 VGA 显示器，也只有一个调色板，对应 16 个调色板寄存器。但这些寄存器装的内容和 EGA 的不同，它们装的又是一个颜色寄存器表的索引，而这些颜色寄存器才寄存要显示的颜色，它们字长为 18 位，因而可选 256K 种颜色，共有 256 个颜色寄存器，因而可一次同时显示 256 种颜色。VGA 的调色板寄存器是 6 位，而要寻址 256 个颜色寄存器需

有 8 位。它是这样寻址的，即还要通过一个所谓模式控制寄存器的最高位(即第 7 位)的值来决定，若为 0(对于 640×480 16 色显示是这样)，则低 6 位由调色板寄存器来给出，高两位由颜色选择寄存器给出，从而组合出 8 位地址码。因此它的象素显示过程是：由 VRAM 提供调色板寄存器索引号(0～15)，再由检索到的调色板寄存器的内容同颜色选择寄存器配合，检索到颜色寄存器，再由颜色寄存器存的颜色值而令显示器显示，其检索过程如图 10.11 所示。

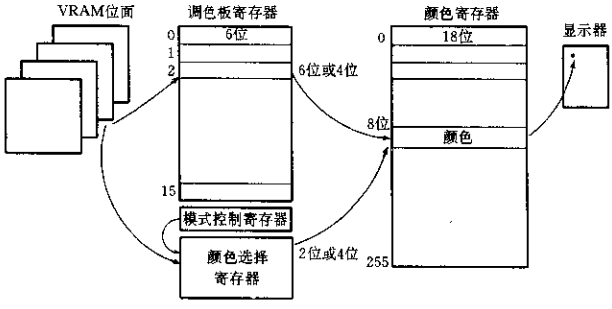


图 10.11　VGA 的象素显色过程

当模式寄存器最高位为 1 时，则调色板寄存器给出低 4 位的 4 位地址码，而由颜色选择寄存器给出高 4 位的 4 位地址码，来组合成 8 位地址码对颜色寄存器寻址而得出颜色值。

上述的调色板寄存器，颜色选择寄存器，模式控制寄存器和颜色寄存器均属于 VGA 显示器中的属性控制器。

由于 Turbo C 中没有支持 VGA 的 256 色的图形模式，只有 16 色方式，因而 16 个颜色寄存器寄存了 16 个颜色寄存器索引号，它们代表的颜色如表 10.4 所列，所显示的颜色和 CGA 下选背景色的顺序一样。注意！EGA 和 VGA 的调色板寄存器装的值虽然一样(当图形系统初始化时，指缺省值)，但含义不同，前者装的是颜色值，后者装的是颜色寄存器索引号，不过它们最终表示的颜色是一致的，因而当用 setpalette(index actual_color)对 index 指出的某个调色板寄存器重新设置颜色时，actual_color 可用表 10.4 所指的颜色值，也可用大写名，如 EGA_BLACK，EGA_BLUE 等。在缺省情况下，和 CGA 上 16 色顺序一样，当使用 setpalette 函数时，index 只能取 0～15，而 actual_color 若其值是表 10.4 所列的值，则调色板颜色保持不变，即调色板寄存器值不变。

② 改变调色板 16 种颜色的函数

```
void far setallpalette(struct palettetype far *palette);
```

其中结构 palettetype 定义如下：

```
#define MAXCOLORS 15
struct palattetype{
    unsigned char size;
    signed char colors[MAXCOLORS+1];
```

};

该定义在头文件graphics.h中。size元素由适配器类型和当前模式下调色板的颜色数决定，即调色板寄存器数。colors是个数组，它实际上代表调色板寄存器，每个数组元素的值就表示相应调色板寄存器的颜色值。对VGA的VGAHI模式，size=16，缺省的colors的各元素值就相当于表10.4所列值。

③ 得到调色板颜色数和颜色值的函数

与上述两个函数对应的是如下两个函数：

void far getpalette(struct palettetype far *palette);

void far getpalettesize(void);

前者将得到调色板的颜色数(即调色板寄存器个数)和装的颜色值，后者将得出调色板颜色数。getpalette函数将把得到信息存入由palette指向的结构中，其结构palettetype定义如上所述。

下面的一个程序演示了调色板颜色设置函数的作用，首先程序用setcolor(1)，设置了前景颜色，其中参数1表示用1号调色板寄存器中的颜色，即缺省值为蓝色，这样用rectangle将画出一个蓝色的方框，然后按任一键，这时用setpalette(1,i)函数将分别设置一号调色板寄存器为绿、青、红……黄、亮白。每按一键，方框颜色改变一次，直到方框变成亮白为止。因为调色板相应的调色板寄存器所装的颜色一旦改变，用setcolor(调色板寄存器号)设置的颜色也立即改变，可以这样形象地理解，setcolor函数相当于把代表红、绿、蓝的三条模拟电压线接通，由于它们电压高低不尽相同，所以显示器混合出来的颜色也不相同，当调成蓝色时，三条模拟电压线代表红绿的电压近似为零，因而这时方框便是蓝色的。然而我们用函数setpallette再调三条模拟电压线的电压比例(如同用电位器调一样)，因而原来画好的框，其颜色将跟着立即改变。

该程序中将palette定义成palettetype类型结构，palettetype结构如前所述在graphics.h头文件中已有定义，这样用getpalette(&pallette)函数得到图形系统初始化的各调色板寄存器的颜色值(即缺省值)，然后在程序快结束时，用setallpalette(&palette)恢复各调色板寄存器的原来值。

可以做实验，将第一个调色板寄存器的颜色值分别置成6,8,9,…,15时产生的颜色并不是标准的BROWN、DARKGRAY、LIGHTBLUE，即和置成20,56,57…63的颜色不一样，这也证明了在EGA/VGA 16色显示图形方式下，16个调色板寄存器装的颜色值(缺省值)不同于CGA的16色值，但其对应的颜色却是一致的，即两者代表同一颜色的颜色值不同，可参阅表10.4。

```
#include<graphics.h>
main()
 {
   int graphdriver=DETECT,graphmode;
   struct palettetype palette;
   int i,j;
   initgraph(&graphdriver,&graphmode,"");
   getpalette(&palette);
```

```
    setcolor(1);
    rectangle(200,200,300,320);
    getch();
    i=2;j=2;
    do
      { printf("color=%d",j);
        setpalette(1,i);
        getch();
        i++;j++;
        if(i==6)i=20;
        if(i==21)i=7;
        if(i==8)i=56;
         }while (j<16);
    getch();
    setallpalette(&palette);
    closegraph();
}
```

10.8 画线的线型函数

10.8.1 设定线型函数

当我们前述例子中用来画线，画圆，画框时，就会发现线的宽度都是一样的，但 Turbo C 也提供了改变线的宽度、类型的函数，其线的宽度当不设定时，取缺省值，即一个象素宽，当设定为 3 时，可取三个象素宽，如表 10.5 所列。当线的形状不设定时，取缺省值，即实线。设定时，可有 5 种选择如表 10.6 所列，它们可用设定线型函数来进行设置：

```
void far setlinestyle(int linestyle,unsigned upattern,int thickness);
```

表 10.5 线宽(thickness)

符号名	值	含义
NORM_WIDTH	1	一个象素宽
THICK_WIDTH	3	二个象素宽

表 10.6 线的形状(linestyle)

符号名	值	含义
SOLID_LINE	0	实线
DOTTED_LINE	1	点线
CENTER_LINE	2	中心线
DASHED_LINE	3	点画线
USERBIT_LINE	4	用户自定义线

其中线型参数(linestyle)和线的宽度参数(thickness)可取的值或大写的符号名如表10.6和表10.5所列。upattern参数只有在linestyle取4或USERBIT_LINE时才有意义，它表示用户自定义线型时，该参数才有用。该参数若表示成16位二进制数，则每位代表一个象素。是1的位，代表的象素用前景色显示，是0的位，代表的象素用背景色显示(实际没有显示)，如图10.12所示。当用setlinestyle(4,0xf3D0)选线型时，它表示linestyle为4，即用户定义线型(USERBIT_LINE)，upattern为16进制数f3D0，图10.12的16位2进制数有斜划线的位为1，表示对应的这些象素将用颜色显示，紧接着的2个象素为背景色不显示，又显示4个象素，最后6个不显示。这些象素即构成一个16个象素长的线段，线宽为1个象素宽，当lineseyle不是USERBIT_LINE时，upattern取0值。

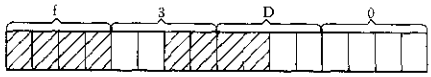


图10.12 用setline style(4,0xf3D0,1)设置的线型

该程序首先在屏中间以屏中心为圆心，半径为98画出一个绿色的圆框，由于没有设置画线的线型和线宽，故取缺省值为1个象素宽的实线。接着用setcolor(12)设置前景色为淡红色。程序进入for循环，而画出线宽交替为1个和3个象素宽的15个矩形框来，框由小到大，一个套一个，颜色为淡红色。程序的下一个for循环将用2、3、4和5颜色，即用绿、青、红、洋红分别画出通过屏幕中心的4条线，线型分别是实线、点线、中心线和点划线，线宽为3个象素，如此重复，共画出5组。

程序最后用setcolor(EGA_WHITE)设置画线颜色为白色，将用自定线型(0x1001)在原先画出的绿色圆框中，标出一个十字线，线的形状为4个白点，8个不显示点，又4个白点，接着又重复这个线段模式，直到画至圆周上而终止。由于人的视觉分辨能力，我们将4个白点看成了一个点，因此出现了屏幕上的那种现象。

```
#include<graphics.h>
main()
 {
   int graphdriver=VGA,graphmode=VGAHI;
   int i,j,x1,y1,x2,y2;
   initgraph(&graphdriver,&graphmode,"");
   setbkcolor(EGA_BLUE);
   cleardevice();
   setcolor(EGA_GREEN);
   circle(320,240,98);                    /*画出一个绿色圆**/
   setcolor(12);                          /*设置颜色为淡红色*/
   j=0;
   for(i=0;i<=90;i=i+6)                   /*画出一个套一个的矩形框*/
     {setlinestyle(0,0,j);
      x1=440-i;y1=280-i;
```

```
    x2=440+i;y2=280+i;
    rectangle(x1,y1,x2,y2);
    j=j+3;
    if(j>4)j=0;
   }
 j=0;
 for(i=0;i<=180;i=i+16)        /*画出通过屏幕中心的4种线型的4色线*/
   {
    if(j>3)j=0;
    setcolor(j+2);
    setlinestyle(j,0,3);
    j++;
    x1=0;y1=i;
    x2=640;y2=480-i;
    line(x1,y1,x2,y2);
   }
 setcolor(EGA_WHITE);
 setlinestyle(4,0x1001,1);        /*用户定义线型,1个象素宽*/
 line(220,240,420,240);           /*画出通过圆心的y线*/
 line(320,140,320,340);           /*画出通过圆心的x线*/
 getch();
 closegraph();
}
```

10.8.2 得到当前画线信息的函数

与设定线型函数 setlinestyle 相对应的是得到当前有关线的信息的函数：

void far getlinesettings(struct linesettingstype far *lineinfo);

该函数将把当前有关线的信息存放到由 lineinfo 指向的结构中，其结构 linesettingstype 定义如下：

```
struct linesettingstype{
        int linestyle;
        unsigned upattern;
        int thickness;
              };
```

10.9 封闭图形的填色函数及有关画图函数

Turbo C 提供了一些画基本图形的函数，如我们前面介绍过的画条形图函数 bar 和将要介绍的一些函数，它们首先画出一个封闭的轮廓，然后再按设定的颜色和模式进行填充，设定颜色和模式有特定的函数，介绍如下：

10.9.1 填色函数

void far setfilestyle(int pattern,int color);

该函数将用设定的 color 颜色和 pattern 图模式对后面画出的轮廓图进行填充，这些图轮廓是由特定函数画出的，color 实际上就是调色板寄存器索引号，对 VGAHI 方式为 0～15 即 16 色，pattern 表示填充模式，可用表 10.7 中的值或符号名表示。

表 10.7 填充模式(pattern)的规定

符号名	值	含义
EMPTY_FILL	0	用背景色填充
SOLID_FILL	1	用单色实填充
LINE_FILL	2	用"—"线填充
LTSLASH_FILL	3	用"///"线填充
SLASH_FILL	4	用粗"///"线填充
BKSLASH_FILL	5	用粗"\\\"线填充
LTBKSLASH FILL	6	用"\\\"线填充
HATCH_FILL	7	用方网格线填充
XHATCH_FILL	8	用斜网格线填充
INTTERLEAVE_FILL	9	用间隔点填充
WIDE DOT_FILL	10	用稀疏点填充
CLOSE_DOT_FILL	11	用密集点填充
USER_FILL	12	用用户定义样式填充

当 pattern 选用 USER_FILL 用户自定义样式填充时，setfillstyle 函数对填充的模式和颜色不起任何作用，若要选用 USER FILL 样式填充时，可选用下面的函数：

10.9.2 用户自定义填充函数

void far setfillpattern(char *upattern,int color);

该函数设置用户自定义可填充模式，以 color 指出的颜色对封闭图形进行填充。这里的 color 实际上就是调色板寄存器号，也可用颜色名代替。参数 upattern 是一个指向 8 个字节存储区的指针，这 8 个字节表示了一个 8×8 象素点阵组成的填充图模，它是由用户自定义的，它将用来对封闭图形填充。8 个字节的图模是这样形成的：每个字节代表一行，而每个字节的每一个二进制位代表该行的对应列上的象素。是 1，则用 color 显示，是 0 则不显示。如定义一个字符数组：char gray50[]={0xaa,0x22,0x01,0xff,0xaa,0x22,0x01,0xff};

1 0 1 0　1 0 1 0
0 0 1 0　0 0 1 0
0 0 0 0　0 0 0 1
1 1 1 1　1 1 1 1
1 0 1 0　1 0 1 0
0 0 1 0　0 0 1 0
0 0 0 0　0 0 0 1
1 1 1 1　1 1 1 1

图 10.13 图模

当用 setfillpattern(gray50,RED)设置填充模式后，后面画出的封闭图将用红色按图 10.13 图案进行连续的填充，直到填满为止。图中是 1 的位置，表示用红色显示一个点，是 0 的位置将不显示。由于在屏幕上两象素间距离太近，肉眼难于分辨，结果好像一个点一样。

下面的程序演示了如何用 setfillstyle(SOLID_FILL,color)对用 bar 生成的条状图进行填充。对 VGA 显示器，由于可显示 16 色，因而通过 getpalette(&palette)函数，可得出 palette.size 为 16。这样 for 循环，将使得用 bar 函数生成的 16 个小条分别填充上调色板上的 16 种颜色，其顺序为缺省时(即标准的)调色板各寄存器的顺序颜色。

do 循环又利用随机函数 random(palette.size)随机产生 0～palette.size 的整数，对调色板各寄存器的颜色重新进行设置，这样一旦 setpalette 函数对某调色板寄存器进行了颜色的重新设置，则代表相应号调色板寄存器的小条颜色立即变成新设的颜色。若随机遇到 random 产生一个 0，则 setpalette 立即对 0 号调色板用第二次 random 产生的数进行颜色设置，因而此时整个屏幕显示的背景色立即发生变化，颜色为第二次 random 产生的颜色。关于调色板的设置问题，可参阅前面已介绍过的调色板设置函数。

```
#include<graphics.h>
#include<stdio.h>
#include<stdlib.h>
main()
 {
   int graphdriver=DETECT,graphmode;
   struct palettetype palette;
   int color;
   initgraph(&graphdriver,&graphmode,"");      /*若是 VGA*/
   getpalette(&palette);
   for (color=0;color<palette.size;color++)    /*对 16 个小条用 16 种颜色填充*/
    {
      setfillstyle(SOLID_FILL,color);
      bar(20*(color-1),0,20*color,20);
    };
   if (palette.size>1)
    {
      do                                       /*对调色板 16 种颜色重新进行设置*/
        setpalette(random(palette.size),random(palette.size));
      while (! kbhit());
      getch();
    }
   setallpalette(&palette);
   closegraph();
 }
```

例

该程序演示了用不同填充图模(pattern)对由 bar 和 pieslice 函数产生的条状和扇形图进行颜色填充。运行程序，可以看出第 1 个 bar(0,0,100,100)产生的方条将由蓝色的斜线

填充，即以 LTSLASH_FILL(3)图模填充。接着将由红色的网格(HATCH_FILL,RED)图模填充一个扇形。由于缺省时，前景颜色为白色，故该扇形将用白色边框画出，接着用户自定义填充模式，因而用 bar(100,100,200,200)画出的方条，将用用户定义的图模(用字符数组 gray50[]表示的图模)，用黄色进行填充。

```
#include<graphics.h>
main()
 {
   int graphdriver=VGA,graphmode=VGAHI;
   struct fillsettingstype save;
   char savepattern[8];
   char gray50[]={0xff,0x00,0x00,0x00,0x00,0x00,0x00,0x81};
   initgraph(&graphdriver,&graphmode,"");
   getfillsettings(&save);                    /*得到初始化时填充模式*/
   if(save.pattern! =USER_FILL)
   setfillstyle(3,BLUE);
   bar(0,0,100,100);
   setfillstyle(HATCH_FILL,RED);
   pieslice(200,300,90,180,90);
   setfillpattern(gray50,YELLOW);             /*设定用户自定义图模进行填充*/
   bar(100,100,200,200);
   if(save.pattern==USER_FILL)
     setfillpattern(savepattern,save.color);
   else
     setfillstyle(save.pattern,save.color);   /*恢复原来的填充模式*/
   getch();
   closegraph();
 }
```

和上述两个函数相对应的是如下函数：

10.9.3 得到填充模式和颜色的函数

```
void far fillsettings(struct fillsettingstype far *fillinfo)
```

它将得到当前的填充模式和颜色，这些信息存在结构指针变量 fillinfo 指出的结构中，该结构定义是：

```
struct fillsettingstype{
        int pattern;        /*当前填充模式*/
        int color;          /*填充颜色*/
       };
```

```
void far getfillpattern(char *upattern);
```

该函数将把用户自定义的填充模式和颜色存入由 upattern 指向的内存区域中。

10.9.4 与填充函数有关的作图函数

前面我们已经介绍了画条形图函数 bar 和画扇形函数 pieslise，它们需要用 setfillstyle 函数设置填充模式和颜色，否则按缺省方式。另外还有一些画图形的函数，也要用到填充函数，现作以下介绍：

① 画三维立体直方图函数

void far bar3d(int x1,int y1,int x2,int y2,int depth, int topflag);

该函数参数名定义如图 10.14 所示。当 topflag 非 0 时，画出三维顶，否则将不画出三维顶，depth 决定了三维直方图的长度.

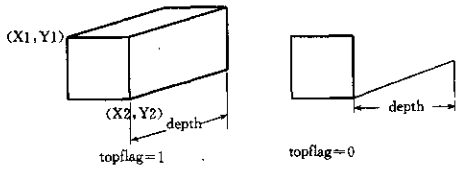


图 10.14 画三维立体直方图函数的参数名定义

② 画椭圆扇形函数

viod far sector(int x,int y,int stangle,int endangle,int xradius,int yradius);

该函数将以(x,y)为圆心，以 xradius 和 yradius 为 x 轴和 y 轴半径，从起始角 stangle 开始到 endangle 角结束，画一椭圆扇形图，并按设置的填充模式和颜色填充。当 stangle 为 0，endangle 为 360 时，则画出一完整的椭圆图。

③ 画椭圆图函数

void far fillellipse(int x,int y,int xradius,int yradius);

该函数将以(x,y)为圆心，以 xradius 和 yradius 为 x 轴和 y 轴半径，画一椭圆图，并以设定或缺省模式和颜色填充。

④ 画多边形图函数

void far fillpoly(int numpoints,int far *palypoints)

该函数将画出一个顶点数为 numpoints，各顶点坐标由 polypoints 给出的多边形，也即边数为 polypoints－1，当为一封闭图形时，numpoints 应为多边形的顶点数加 1，并且第一个顶点坐标应和最后一个顶点的坐标相同。

下面程序用 bar3d 函数画出了一个立方图，并且画面用蓝色斜线填充，接着由第二个 bar3d 函数又在相邻位置画出一个没有顶的三维图，画面用红色方格填充。该函数的 topflag＝0。在屏幕下方，由 sector 函数画出了一个不完整的椭圆，并用绿色填充，可以看出差 120 度就是一个完整的椭圆了。在其相邻位置则是由 fillellipse 函数画出的一个椭园，它用淡红色填充，屏幕的右上半是由 fillpoly 函数画出的一个六边图形，被填以洋红色，由于最初顶点坐标和最后一个顶点坐标相同(同为(420,20))，所以是一个封闭的图形。

```
#include<graphics.h>
```

```
main()
 {
   int graphdriver=VGA,graphmode=VGAHI;
   struct fillsettingstype save;
   char savepattern[8];
   int d[]={420,20,330,45,330,145,420,120,510,145,510,55,420,20};
   initgraph(&graphdriver,&graphmode,"");
   getfillsettings(&save);
   setfillstyle(3,BLUE);
   bar3d(100,50,150,120,30,1);
   setfillstyle(HATCH_FILL,RED);
   bar3d(200,50,250,120,30,0);
   setfillstyle(1,GREEN);
   sector(200,300,0,250,100,40);
   setfillstyle(1,LIGHTRED);
   fillellipse(420,300,100,40);
   setfillstyle(1,5);
   fillpoly(7,d);
   getch();
   setfillstyle(save.pattern,save.color);
   closegraph();
 }
```

10.9.5 可对任意封闭图形填充的函数

前面介绍的填充函数，只能对由上述特定函数产生的图形进行颜色填充，对任意封闭图形均可进行填充的还有一函数，其原型说明为：

```
void far floodfill(int x,int y,int border);
```

该函数将对一封闭图形进行填充，其颜色和模式将由设定的或缺省的图模与颜色决定。其中参数(x,y)为封闭图形中的任一点，border 是封闭图形的边框颜色。编程时该函数位于画图形的函数之后，即要填充该图形。需要注意的是：

a) 若(x,y)点位于封闭图形边界上，该函数将不进行填充。

b) 若对不是封闭的图形进行填充，则会填到别的地方，即会溢出。

c) 若(x,y)点在封闭图形之外，将对封闭图形外进行填充。

d) 由参数 border 指出的颜色必须与封闭图形的轮廓线的颜色一致，否则会填到别的地方去。

下面的程序首先用白色线画出一个长方体，并用设定的亮红色(LIGHTRED)和实填充模式填充该长方体的正面，然后使用两个 floodfile 函数用同样的模式和颜色填充该长方体能看得见的另两面，然后将画线颜色由 setcolor(LIGHTGREEN)设置为亮绿色，并由 setfillstyle 设置填充模式为棕色和用平直线填充，由于该函数对 rectangle 画出的矩形框不起作用，所以并不执行填充，当画好矩形框后，其后的 floodfile 函数将完成对该矩形框的填充，即以棕色的平直线进行填充。可以作实验，当将 floodfile 中的 x,y 参数设在被填图形框外

时，结果会将该框外的所有区域填充。当 floodfile 中的 border 指定的颜色和画图框的颜色不符时，也会将颜色填到图外边去。

```
#include<graphics.h>
main()
 {
   int graphdriver=VGA,graphmode=VGAHI;
   initgraph(&graphdriver,&graphmode,"");
   setbkcolor(BLUE);
   setcolor(WHITE);                    /*用白色画线*/
   setfillstyle(1,LIGHTRED);           /*设填充模式和颜色*/
   bar3d(100,200,400,350,100,1);       /*画长方体并填正面*/
   floodfill(450,300,WHITE);           /*填侧面*/
   floodfill(250,150,WHITE);           /*填顶部*/
   setcolor(LIGHTGREEN);
   setfillstyle(2,BROWN);
   rectangle(450,400,500,450);         /*画矩形*/
   floodfill(470,420,LIGHTGREEN);      /*填矩形*/
   getch();
   closegraph();
 }
```

10.10 屏幕操作函数

这类函数主要完成对屏幕图象的操作，除了本章开始部分讲的清屏函数 cleardevice() 外，主要还有如下几个：

10.10.1 屏幕图象存储和显示函数

① 存屏幕图象到内存区

void far getimage(int x1,int y1,int x2,int y2,void far *bitmap);

该函数将把屏幕左上角为(x1,y1)，右下角为(x2,y2)矩形区内的图象保存到指针 bitmap 指向的内存区去。为了能开辟一个内存缓冲区，使它恰能存下所指矩形区中的图象，则必须首先要知道所存图象占多少字节，则内存缓冲区也可设这样多的字节，这可用下面的函数：

② 测定图象所占字节数的函数

unsigned far imagesize(int x1,int y1,int x2,int y2);

该函数将得到屏幕上左上角为(x1,y1)，右下角为(x2,y2)矩形区内图象所占的字节数。

③ 将所存图象显示函数

void far putimage(int x1,int y1,void far *bitmap,int op);

该函数将把指针 bitmap 指向的内存区中所装图象，与屏上现有左上角为(x1,y1)的矩

形区内图象进行 op 规定的操作后显示在屏上，关于 op 操作如表 10.8 所列：

表 10.8　op 规定值及操作

符号名	值	含义
COPY_PUT	0	复制
XOR_PUT	1	进行异或操作
OR_PUT	2	进行或操作
AND_PUT	3	进行与操作
NOT_PUT	4	进行非操作

该函数进行各种图象的逻辑操作如同二进制操作一样。设取图象一行中的 4 个象素，1 为象素显示，0 为不显示，即和背景色一样，下面的算式表示取屏幕上 4 个象素，再取内存缓冲区中同一位置的 4 个象素进行逻辑操作：

```
    1011        1011        1011
XOR 0101    OR 0101    AND 0101    NOT 0101    COPY 0101
    1110        1111        0001        1010        0101
```

0101 表示内存中的图象，1011 表示屏上的某一位置图象，下面的便是经过相应逻辑操作后的图象。XOR(异或)表示两位相异时才为 1。OR(或)表示两位中只要有一位是 1，结果便是 1。AND(与)运算表示两位值相同的位，与的结果仍为原来的数。NOT(非)是将内存中图象值取反，COPY(复制)则是重新复制。后两种运算均将覆盖掉原来屏上的图象，下面图 10.15 表示了两图象逻辑运算后的图象：

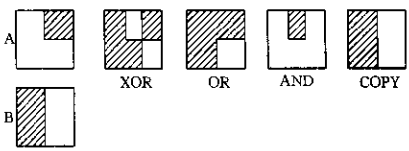


图 10.15　两个图象的逻辑运算

下面的程序演示了图的各种逻辑操作，for 循环用来在屏上方产生连续的五个方框，方框中套一用洋红色(5)填充的小方块，五个图象全一样。循环结束后，又在屏下方画出两个框，小框用洋红色填充并在大框内。程序运行后，立即在屏上显示出上述图案，当按任一键后，则由函数 imagesize 得到屏下方大框套一填充框区域内图象所占字节数，然后由 malloc 函数按字节数分配一内存缓冲区 buffer，再由 getimage 函数将图象存到 buffer 中，然后复制到屏上方左边第一个框位置。按任一键后，又将 buffer 中图象和第二个框图象进行与操作后显示，再按任一键，buffer 中的图象又和第三个方框内图象进行或操作并显示，如此重复，则可将 5 种逻辑操作结果均显示在屏上。注意 COPY 和 NOT 操作将与原来屏上的图象无关，buffer 中图象经过这两种操作，将覆盖掉原屏上图象，并将结果进行显示。

```
#include<graphics.h>
main()
{
  int i,j,graphdriver,graphmode,size;
  void *buffer;
  graphdriver=DETECT;
  initgraph(&graphdriver,&graphmode,"");
  setbkcolor(BLUE);
  cleardevice();
  setcolor(YELLOW);
  setlinestyle(0,0,1);
  setfillstyle(1,5);
  for(i=0;i<5;i++)                          /*产生连续的五个方框中套小框的图*/
   {
     j=i*110;
     rectangle(80+j,100,130+j,150);         /*产生小框且用洋红色填充*/
     floodfill(110+j,140,YELLOW);
     rectangle(50+j,100,130+j,180);         /*画大框*/
   }
  rectangle(50,340,100,420);                /*产生一个小框*/
  floodfill(80,360,YELLOW);                 /*用洋红色填充*/
  rectangle(50,340,130,420);                /*产生一个大框*/
  getch();
  size=imagesize(40,300,132,430);           /*取得(40,300)右下角(132,430)区域图
                                              象字节数*/
  buffer=malloc(size);                      /*分配缓冲区(按字节数)*/
  getimage(40,300,132,430,buffer);          /*存图象*/
  putimage(40,60,buffer,COPY_PUT);          /*重新复制*/
  getch();
  j=110;
  putimage(40+j,60,buffer,AND_PUT);         /*和屏上的图与操作*/
  getch();
  putimage(40+2*j,60,buffer,OR_PUT);
  getch();
  putimage(40+3*j,60,buffer,XOR_PUT);
  getch();
  putimage(40+4*j,60,buffer,NOT_PUT);
  getch();
  closegraph();
}
```

10.10.2 设置显示页函数

存储在显示适配器上的图象存储器 VRAM 中的一满屏图象信息称为一页。每个页一

般为 64K 字节，VRAM 可以存储要显示的图象几个页(视 VRAM 容量而定，最大可达 8 页)，Turbo C 支持页的功能有限，按在图形方式下显示的模式最多支持 4 页(EGALO 显示方式)，一般为两页(注意对 CGA，仅有一页)，因存储图象的页显示时，一次只能显示一页，因此必须设定某页为当前显示的页(又称可视页)，缺省时定为 0 页，如图 10.16 所示。

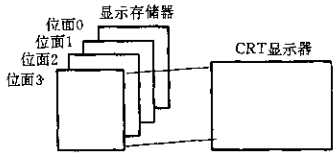


图 10.16　VRAM 中显示页与显示的关系

正在由用户编辑图形的页称为当前编辑页(又称激活的页)，这个页不等于显示页，即若用户不设定该页为当前显示页时，在该页上编辑的图形将不会在屏上显示出来。缺省时，设定 0 页为当前编辑页，即若不用下述的页设置函数进行设置，就认定 0 页既是编辑页，又是当前显示页。

设置激活页和显示页的函数如下：

void far setactivepage(int pagenum)；

void far setvisualpage(int pagenum)；

这两个函数只能用于 EGA、VGA 等显示适配器。前者设置由 pegenum 指出的页为激活的页，后者设置可显示的页。当设定了激活的页，即编辑页后，则程序中其后的画图操作均在该页进行，若它不定为显示页，则其上的图象信息并不会在屏上显示出来。关于图形模式和可以设定的页数，可参阅表 10.1。

下面的程序演示了设置显示页函数的应用。首先用 setactivepage(1)设置 1 页为编辑页，在上面画出一个用洋红色填充的方块，此图并不显示出来(因缺省时，定义 0 页为可视页)。接着又定义 0 页为编辑页并清屏(即清 0 页)，也定义 0 页为可视页，并在其上画出一个红色边框，用淡绿色填充的圆，该圆将在屏上显示出来。接着进入 do 循环，设置 1 页为可视页，因而其上的洋红色方块便在屏上显示出来，圆的图象消失，用 delay(2000)将方块图象保持 2000 毫秒即 2 秒，当不按键时，下一次循环又将 0 页设为可视页，因面圆的图象显示出来，方块图象又消失。保持 2 秒后，又重复刚开始的过程。这样我们就会看到：屏上同一位置淡绿色方块和洋红色圆交替出现，若将 delay 时间变少，将会出现动画的效果。

```
#include<graphics.h>
main()
{
  int i,graphdriver,graphmode,size,page;
  graphdriver=DETECT;
  initgraph(&graphdriver,&graphmode,"");
  cleardevice();
  setcolor(YELLOW);
```

```
setactivepage(1);                    /* 设置 1 页为编辑页 */
setfillstyle(1,5);
bar(100,210,160,270);                /* 画方块并填充洋红色 */
setactivepage(0);                    /* 设置 0 页为编辑页 */
cleardevice();                       /* 清 0 页 */
setvisualpage(0);                    /* 设置 0 页为可视页 */
setbkcolor(BLUE);
setcolor(RED);
setfillstyle(1,10);
circle(130,210,30);                  /* 画圆 */
floodfill(130,210,4);                /* 用淡绿色填充圆 */
page=1;
do
 {
  setvisualpage(page);               /* 显示设定页的图象 */
  delay(2000);                       /* 延迟 2000ms */
  page=page-1;
  if (page<0)
     page=1;
  }while(! kbhit());
 getch();
 closegraph();
}
```

10.11 图视口操作函数

10.11.1 图视口设置函数

在图形方式下可以在屏幕上某一区域设置一个窗口，这样以后的画图操作均在这个窗口内进行，且使用的坐标以此窗口顶左上角为(0,0)作参考，而不再用物理屏幕坐标(屏左角为(0,0)点)。在图视口内画的图形将显示出来，超出图视口的部分可以不让其显示出来，也可以让其显示出来(不剪断)，该函数原型说明为：

void far setviewport(int x1,int y1,int x2,int y2,clipflag);

其中(x1,y1)为图视口的左上角坐标，(x2,y2)为所设置的图视口右下角坐标，它们都是以原屏幕物理坐标为参考的。clipflag 参数若为非 0，则所画图形超出图视口的部分将被切除而不显示出来。若 clipflag 为 0，则超出图视口的图形部分仍将显示出来。

10.11.2 图视口清除与取信息函数

1. 图视口清除函数

void far clearviewport(void)

该函数将清除图视口内的图象。

2. 取图视口信息函数

void far getviewsettings(struct viewport type far * viewport);

该函数将取得当前设置的图视口的信息，它存于由结构 viewporttype 定义的结构变量 viewport 中，结构 viewporttype 定义如下：

```
struct viewporttype{
        int left,top,right,bottom;
        int clipflag;};
```

使用图视口设置函数 setviewport，可以在屏上设置不同的图视口——窗口，甚至部分可以重叠，然而最近一次设置的窗口才是当前窗口，后面的图形操作都视为在此窗口中进行，其它窗口均无效。若不清除那些窗口的内容，则它们仍在屏上保持，当要对它们处理时，可再一次设置那个窗口一次，这样它就又变成当前窗口了。

使用 setbkcolor 设置背景色时，对整个屏幕背景起作用，它不能只改变图视口内的背景，在用 setcolor 设置前景色时，它对图视口内画图起作用。若下一次设置的图视口没有设置颜色，那么上次在另一图视口内设置的颜色在本次设置的图视口内仍起作用。

下面程序首先用 setviewport 函数设置了一个左上角(0,0)，右下角为(639,199)的图视窗口，并设置 clipflag 参数为 1，即超出图视口的图形将被切除，接着画了一个方框和一个边与方框一边相切的圆，它将完整地显示出来。按任意键后，开一个图视口，用棕色画了和第一个窗口相同的方框和圆，超出图视口的部分被剪切。由于在开此窗口前，没有用 clarviewport()清除上次的窗口内容，所以上次画的洋红色的方框和圆仍保留，当再按任一键后，调用了 clarviewport()函数，因而将窗口中的棕色方框和圆清除了。接着又建立了一个图视窗口，该窗口(50,50,200,125)和最初窗口(0,0,639,199)有部分重合，因而重合部分的内容将在当前窗口中显示出来。新画的方框和圆也显示出来(窗外部分内容仍保留)，由于选择了 clipflag=1，所以超出窗口的方框和圆的部分被剪切，不显示出来，但再按任一键后，由于用了 clearviewport，所以窗口中内容被清除并又在同一位置开辟了同样大小的窗口，但由于 clipflag=0，所以超出的部分没被剪掉，因而可看见。

```
#include<graphics.h>
main()
{
  int i,graphdriver,graphmode,size,page;
  graphdriver=VGA;
  graphmode=VGAHI;
  initgraph(&graphdriver,&graphmode,"");
  cleardevice();
  setviewport(0,0,639,199,1);                    /*设图视口*/
  setcolor(5);
  rectangle(50,50,125,100);                      /*方框*/
  circle(100,75,50);                             /*圆*/
   getch();
  setviewport(150,150,639,239,1);                /*又设一图视口*/
  setcolor(6);
  rectangle(50,50,125,100);                      /*方框超出部分被切掉*/
```

```
 circle(100,75,50);                          /* 圆超出部分被切掉 */
  getch();
 clearviewport();                            /* 清窗口 */
 setviewport(50,50,200,125,1);
 rectangle(50,50,125,100);
 circle(100,75,50);
  getch();
 clearviewport();
 setviewport(50,50,200,125,0);               /* 开窗口,超出部分不切除 */
 rectangle(50,50,125,100);
 circle(100,75,50);
  getch();
 clearviewport();
 closegraph();
}
```

利用图视口设置技术，可以实现图视口动画效果，例如可在不同图视口中设置同样的图象，而让图视口沿 x 轴方向移动设置，这次出现前要清除上次图视口的内容，这样就会出现图象沿 x 轴移动的效果。下面的程序就是这样作的，不断的沿 x 轴开辟图视窗口，就像一个大小一样的窗口沿 x 轴在移动，由于总有 clearviewport 函数清除上次窗口的相同立方体，因面视觉效果上，就像一个立方体从左向右移动一样。程序中定义的 movebar 函数作用是开辟一个图视窗口，并画一个填色的立方体，保留 250 毫秒然后清除它，主程序不断调用它，因每次顶点 x 坐标在增加，因而效果是立方体沿 x 轴从左向右在运动。

```
#include<graphics.h>
main()
 {
  int i,graphdriver,graphmode;
  graphdriver=VGA;
  graphmode=VGAHI;
  initgraph(&graphdriver,&graphmode,"  ");
  for(i=0;i<11;i++)
    {setfillstyle(1,i);
     movebar(i*50);                    /* 调用函数 */
    }
  closegraph();
 }
movebar(int xorig)                     /* 设窗口并画填色小立方体 */
 {
    setviewport(xorig,0,639,199,1);
    setcolor(5);
    bar3d(10,120,60,150,40,1);
    floodfill(70,130,5);
    floodfill(30,110,5);
```

```
    delay(250);
    clearviewport();
}
```

10.12 图形方式下的文本输出函数

在图形方式下，虽然也可以用 printf()，puts()，putchar()函数输出文本，但只能在屏上用白色显示，无法选择输出的颜色，尤其想在屏上定位输出文本，更是困难，且输出格式也是不能变的 80 列×25 行形式。

Turbo C 提供了一些专门用在图形方式下的文本输出函数，它们可以用来选择输出位置，输出字型、大小，输出方向等。

为在图形方式下输出文本，Turbo C 提供一个 8×8 点阵的字库，它们是英文字母和一些常用符号的字模，是用 8×8 点阵表示出其图象的字形库。该库嵌入在图形系统中，我们一旦在 Turbo C 下对系统进行了图形系统初始化(即 initgraph())，该字库即被调入内存，这种字库也是在缺省情况下，图形方式中输出文本时采用的字形。另外 Turbo C 的图形接口软件(BGI)还提供 4 种向量字库，又称笔划字库。该字库中，字符用一组向量表示，这些向量表示如何画字符，4 个向量字库在磁盘上用后缀为.chr 文件名存放，它们是 itrip.chr(三倍笔划体字库)、litt.chr(小笔划字库)，sans.chr(无衬笔划字库)，goth.chr(黑体笔划字库)。当文本输出选择这些字库中的一个时，如用 settextstyle()函数，则相应的笔划字库就被调入内存。这些字形常被用来显示放大的字，因 8×8 点阵字放大时，明显地看出字形不好看，仿佛是用小方块堆砌成的。

下面介绍有关在图形方式下输出文本的函数：

10.12.1 文本输出函数

1. 当前位置文本输出函数

```
void far outtex(char far * textstring);
```

该函数将在当前位置在屏上输出由字符串指针 textsering 指出的文本字符串。该函数没有定位参数，只能在当前位置输出字符串。

2. 定位文本输出函数

```
void far outtextxy(int x,int y,char far * textstring);
```

该函数将在指定的(x,y)位置输出字符串，(x,y)位置如何确定，还需要用位置确定函数 settextjustify()来确定，选用何种字形显示，字体大小，及横向或纵向显示还需用 settextstyle()函数来确定，这些均要在文本输出函数之前确定。若没有使用函数确定，则输出采用缺省方式，即字形采用 8×8 点阵字库，横向输出，其(x,y)位置表示输出字符串的第一个字符的左上角位置，字体 1∶1。如当执行函数

```
outtextxy(10,10,"Turbo C");
```

时，Turbo C 将采用缺省方式显示在如图 10.17 所示位置，其"T"字的左上角位置为(10,10)，字形为 8×8 点阵(8 个象素宽，8 个象素高)，尺寸 1∶1，即和字库中的字同大。

3. 文本输出位置函数

void far settextjustify(int horiz,int vert);

该函数将确定输出字位串时，如何定位(x,y)。即当用outtext(x,y,"字符串")或outtextxy(x,y"字符串")输出字符串时，(x,y)点是定位在字符串的哪个位置，horiz 将决定(x,y) 点的水平位置相对于输出字符串如何确定，vert 参数将决定(x,y)点的垂直位置相对于输出字符串如何确定。这两个参数的取值和相应的符号名如表 10.9 和表 10.10 所示，如若 horiz 取 LEFT_TEXT(或取 0)，则(x,y)点是以输出的第一个字符的左边为开始位置，即(x,y)定位于此。但是以第一个字符左边的顶部、中部，还是底部定位(x,y)，还不能确定，也就是说，(x,y)点是指输出字符串第一个字符的左边位置，但是在第一个字符左边的垂直方向上的位置还须由 vert 参数决定。如 vert 取 TOP_TEXT(即 2)，则(x,y)在垂直方向定位于第一个字符左边位置的顶部，如图 10.18 所示。

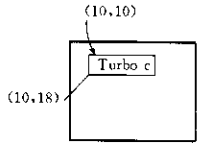


图 10.17 outtextx.y(10,10,"Turbo C")输出

表 10.9 参数 horiz 的取值

符号名	值	含义
LEFT_TEXT	0	输出左对齐
CENTER_TEXT	1	输出以字串中心对齐
RIGHI_TEXT	2	输出右对齐

表 10.10 参数 vert 的取值

符号名	值	含义
BOTTOM_TEXT	0	底部对齐
CENTER_TEXT	1	中心对齐
TOP_TEXT	2	顶部对齐

```
settextjustify(LEFT_TEXT,0);          settextjustify(0,BOTTOM_TEXT);
outtextxy(120,20,"Turbo C");          outtextxy(220,120,"Turboc");
settextjustify(CENTER_TEXT,0);        settextjustify(0,CENTER_TEXT);
outtextxy(120,40,"Turbo C");          outtextxy(240,120,"Turboc");
settextjustify(RIGHT TEXT,0);         settextjustify(0,Top-TEXT);
outtextxy(120,60,"Turbo C");          outtextxy(260,120,"Turboc");
```

图 10.18 是由后面所列程序产生，该程序首先用 settextjustify()函数中的 horiz 参数，设置不同的值，如表 10.9 所列顺序，在屏上产生 x 坐标不同定位的 Turbo C 字符串输出，可以看出第一行是以左对齐输出，即以(120,20)点为坐标横向输出。第二行则以(120,40)点坐标为中点，左右输出。第三行则以(120,60)点坐标为原点向左输出。其所用语句如上面左部所列。

程序的后半部则演示了 settextjustify()函数中 vert 参数的作用，首先用 settextstyle()函数设置了字符串垂直输出(该函数下面将介绍)，然后用 settextjustify()函数，将参数 vert

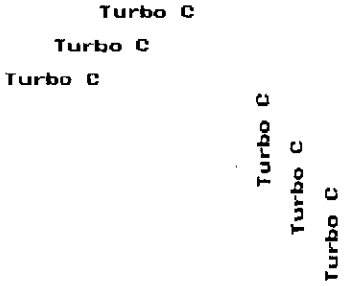


图 10.18 horiz 和 verb 的作用

分别设置成如表 10.10 所取的值，然后在不同的 x 位置输出 Turbo C 字符串。可以看出当垂直输出字符串时，y 方向坐标如何定位，当 vert 选 BOTTOM_TEXT 时，则字符串底部以所定坐标为准纵向输出，当 vert 选 CENTER_TEXT 时，则以所定坐标为中点上下输出，而当选 TOP_TEXT 时，则以顶部为准，纵向输出。

```
#include<graphics.h>
main()
{
   int i,graphdriver,graphmode,size,page;
   char s[30];
   graphdriver=DETECT;
   initgraph(&graphdriver,&graphmode,"");
   cleardevice();
   sprintf(s,"Turbo C");
   settextjustify(LEFT_TEXT,0);
   outtextxy(120,20,s);
   settextjustify(CENTER_TEXT,0);
   outtextxy(120,40,s);
   settextjustify(RIGHT_TEXT,0);
   outtextxy(120,60,s);
   settextstyle(0,VERT_DIR,1);
   settextjustify(0,BOTTOM_TEXT);
   outtextxy(220,120,s);
   settextjustify(0,CENTER_TEXT);
   outtextxy(240,120,s);
   settextjustify(0,TOP_TEXT);
   outtextxy(260,120,s);
   getch();
   closegraph();
}
```

10.12.2 定义文本字型函数

void far settextstyle(int font,int direction,int char size);

该函数用来设置文本输出的字形、大小和方向。其中参数 font 用来设置字形，其取值如表 10.11 所列，参数 direction 取值如表 10.12 所列，charsize 的取值如表 10.13 所列。

表 10.11 font 的取值

符号名	值	含义
DEFAULT_FONT	0	8×8 字符点阵(缺省值)
TRIPLEX_FONT	1	三倍笔划体字
SMALL_FONT	2	小字笔划体字
SANSSERIF FONT	3	无衬线笔划体字
GOTHIC_FONT	4	黑体笔划体字

表 10.12 direction 的取值

符号名	值	含义
HORIZ_DIR	0	水平输出
VERT_DIR	1	垂直输出

font 可选的字体如图 10.13 所示。

表 10.13 char size 的取值

符号名或值	含义	符号名或值	含义
1	8×8 点阵	7	56×56 点阵
2	16×16 点阵	8	64×64 点阵
3	24×24 点阵	9	72×72 点阵
4	32×32 点阵	10	80×80 点阵
5	40×40 点阵	USER_CHAR_SIZE=0	用户自定义字符大小
6	48×48 点阵		

图 10.19 中的 font 可选字体是由下面的程序输出到屏上的，该程序用 settextstyle()函数，按表 10.11 所列的 font 参数可取值，分别在屏上不同位置输出了用该字体符号名作为字符串的不同字形，输出方向为水平输出(即 direction 取 HORIZ_DIR)，而输出字符的点阵，即 char size 参数则取 2，为 16×16 点阵。

```
#include<graphics.h>
main()
{
```

```
int i,graphdriver,graphmode,size,page;
char s[30];
graphdriver=DETECT;
initgraph(&graphdriver,&graphmode,"");
cleardevice();
settextstyle(DEFAULT_FONT,HORIZ_DIR,2);
settextjustify(LEFT_TEXT,0);
outtextxy(220,20,"Defaut font");
settextstyle(TRIPLEX_FONT,HORIZ_DIR,2);
settextjustify(LEFT_TEXT,0);
outtextxy(220,50,"Triplex font");
settextstyle(SMALL_FONT,HORIZ_DIR,2);
settextjustify(LEFT_TEXT,0);
outtextxy(220,80,"Small font");
settextstyle(SANS SERIF_FONT,HORIZ_DIR,2);
settextjustify(LEFT_TEXT,0);
outtextxy(220,110,"Sans serif font");
settextstyle(GOTHIC_FONT,HORIZ_DIR,2);
settextjustify(LEFT_TEXT,0);
outtextxy(220,140,"Gothic font");
getch();
closegraph();
}
```

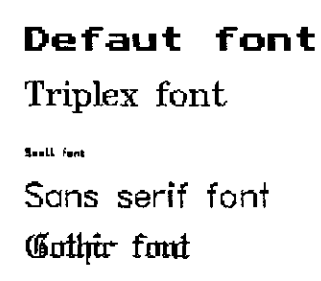


图 10.19 font 可选字体例

10.12.3 文本输出字符串函数

int sprintf(char *string,char *format[,argument,…])

该函数将把变量值 argument,…按 format 指定的格式,输出到由指针 string 指定的字符串中去,该字符串就代表了其输出。这个函数虽然不是图形专用函数,但它在图形方式下的文本输出中很有用,因为用 outtext()或 outtextxy()函数输出时,输出量是文本字符串,当我们要输出数值时不太方便。因而可用 sprintf 函数将数值输出到一个字符数组中,再让

文本输出函数输出这个字符数组中的字符串即可。

下面的程序首先用 setviewport 开了一个对角坐标为(40,40)和(600,440)的图视窗口，其后的图形操作则在此窗口中进行，超过图视口的部分将被剪切，接着用黄色画了一个矩形框，并用绿色填充。下面的 rectangle()函数又在其框内再画一个矩形框，并用淡洋红色填充，这样，就在屏上出现了带有绿色边框(用黄线勾出了边框四周)的一矩形。然后在其内用蓝色写出"Welcom your"字样，由于用函数 settextstyle(1,0,6)设置使用字体为 TRIPLEX_FONT 的笔划字形，且水平放置(direction=0)，字体大小取 6 倍(char size=6)。因而 welcom your 字符串字形将不同于我们通常看到的印刷字体。接着又用 setviewport(100,200,540,380,0)开辟了又一个窗口，且以后操作时窗口外的图形将不被切除(这种方法常用于调正显示图形在图视口中的位置，以知道显示图形超出了图视口多大距离，便于调正)。由于又设置了一个图视窗口，因而先前建立的那个图视口就不起作用了(由于没使用 clearviewport()函数，所以原窗口内的内容仍保留)，又以此窗口为坐标系，在此内画了一个黄色边框的矩形框。由于用 setfillstyle(1,12)进行了设置，因而其内用淡红色进行了填充，接着用 sprintf 函数将要显示的字符串存在 s[]字符数组内(因要多次用到该字符串，为了方便，故这样操作)，执行完函数 outtextxy(60,40,s)后，又用 9 倍的小号笔划体(settextstyle(2,0,9)的设置)，将"Let's study Turbo C"显示在淡红色框内，接着用 settextstyle(4,0,3)设置，将该字符串用黑体笔划字放大 3 倍用蓝色显示在该框的下边部分。

```
#include<graphics.h>
main()
{
   int i,graphdriver,graphmode,size,page;
   char s[30];
   graphdriver=DETECT;
   initgraph(&graphdriver,&graphmode,"");
   cleardevice();
   setbkcolor(BLUE);
   setviewport(40,40,600,440,1);              /*开图视口*/
   setfillstyle(1,2);
   setcolor(YELLOW);
   rectangle(0,0,560,400);
   floodfill(50,50,14);                       /*用绿色填画出的矩形框*/
   rectangle(20,20,540,380);
   setfillstyle(1,13);
   floodfill(21,300,14);                      /*用淡洋红色填画出的矩形框*/
   setcolor(BLACK);
   settextstyle(1,0,6);                       /*设要显示字符串的字形方向,尺寸*/
   outtextxy(100,60,"Welcom Your");
   setviewport(100,200,540,380,0);            /*又开一窗口*/
   setcolor(14);
   setfillstyle(1,12);
   rectangle(20,20,420,120);
```

```
    settextstyle(2,0,9);
    floodfill(21,100,14);                    /*用深蓝色填充*/
    sprintf(s,"Let's study Turbo C");        /*将字符串存到s字符数组内*/
    setcolor(YELLOW);
    outtextxy(60,40,s);                      /*用黄色显示*/
    setcolor(1);
    settextstyle(4,0,3);                     /*设选用字形4,放大3倍,水平设置*/
    outtextxy(110,80,s);                     /*显示s字符串*/
    getch();
    closegraph();
}
```

下面的程序示例了在图形方式下,当用 settextjustify(horiz,vert)时,怎样定义输出字符串的位置,即当用 outtext(x,y,"字符串"),或 outtextxy(x,y,"字符串")显示字符串时,(x,y)点用字符串的什么位置作为参考。程序中开始由于没有用 settextstyle()函数,因而选用缺省值,显示 Nankai University,(x,y)点确定为左对齐(因用 settextjustify(LEFT_TEXT,0)进行设置),接着用 settextjustify(CENTER_TEXT,0)设置为中心对齐,并用 4 倍的 8×8 点阵字体(即缺省字体)显示字符串。可以看出,由于 x 值相同,但是显示时该串中心恰对上次串的左边,其后是用 3 倍字体垂直显示该字符串(用 sertextstyle(DEFAULT_FONT,VERT_DIR,3);)。(x,y)定位采用顶对齐方式(用 settextjustify(0,TOP_TEXT)函数进行了设置),程序最后用 6 倍的 TRIPLEX_FONT 字体,水平用淡红色显示了该字符串。当按任何键后,程序停止。

```
#include<graphics.h>
main()
{
   int i,graphdriver,graphmode,size,page;
   char s[30];
   graphdriver=DETECT;
   initgraph(&graphdriver,&graphmode,"");
   cleardevice();
   sprintf(s,"Nankai University");
   setbkcolor(WHITE);
   setcolor(RED);
   settextjustify(LEFT_TEXT,0);              /*用左对齐方式*/
   outtextxy(320,120,s);/*"Nankai University");*/
   setcolor(5);
   settextjustify(CENTER_TEXT,0);            /*用中心对齐方式*/
   settextstyle(DEFAULT_FONT,HORIZ_DIR,4);
   outtextxy(320,260,s);/*"Nankai University");*/
   settextstyle(DEFAULT_FONT,VERT_DIR,3);
   settextjustify(0,TOP_TEXT);               /*垂直显示用顶对齐方式定位*/
   setcolor(BLUE);
   outtextxy(80,115,s);
```

```
    settextstyle(TRIPLEX_FONT,HORIZ_DIR,6);  /* 设置用三倍笔划体,水平放置,
                                                 放大6倍*/
    setcolor(LIGHTRED);
    outtextxy(120,320,s);
    getch();
    closegraph();
}
```

10.13 综合应用例

本节给出了几个使用画图函数的综合应用。比如用自定义图模进行填充的例,可以形象地看出各种填充图模的图象,可为自己设置图形程序进行填充提供图案。另外在办公室自动化,产生各种统计报表时,若用彩色图显示一些枯燥的数字将是很有趣的,既直观又好看。为此示例了画饼状图、直方图的程序。在实验室中对一些实验测试点的描述中,经常要画出其分布图,连线图,为此举例了画点线图的程序。一个实用软件,往往要有一个使用说明,操作步骤,各功能实现的命令,这就是和用户的一个接口——菜单的功能。它的直观上的好坏如同一个商品的外包装,人们力求做的好看、醒目,容易操作,为此举了一个菜单程序例,说明一个菜单的制作过程。

由于屏幕上的图形往往希望打印出来,以便保存,仔细研究,因而最后给出一个打印屏幕图形的程序例。

10.13.1 用户自定义图模填充长方框图象

该程序演示了由用户自定义13种图模填充各长方框的图象,该程序首先用for循环连续在屏的上下各画了七个长方框。接着又用for循环用7种定义模式分别对上面的7个长方框进行了填充。可以看出:第一个长方框未填充,这是因为填图模放在二维数组fillpattern[0][]中,它的内容为{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},即用8×8点阵组成的小方块连续填充。其方块颜色为背景色。故第一个长方框是黑的,即填充颜色就是原背景色。第二个长方框用fillpattern[1][]表示的图模填充,即8×8点阵出现1的位置上的象素用颜色2(即绿色)显示。如此循环,直到第7个填充完。亮的地方用颜色7(即浅灰色)显示,该循环结束后,又进入第三个for循环,开始对下面六个长方条进行填充,直到下面第6个用淡洋红填充完为止。最后一个长方条未填充,用来和前面13个经过填充的长方条进行比较。该循环结束后,用setfillstyle(save,pattern,save.color);恢复原填充模式,当按任一键后,程序结束。

用这种自定义模式填充,常用在图画设计中,如模仿壁纸,喷涂模糊的水平线等各种色调和图模的设计。

```
#include<graphics.h>
main()
 {
    int graphdriver=VGA,graphmode=VGAHI,x,color;
    struct fillsettingstype save;
```

```
    char savepattern[8];
    char fillpattern[13][40]={{0x09,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
                              {0x00,0x20,0x00,0x00,0x00,0x20,0x00,0x00},
                              {0x20,0x00,0x20,0x00,0x80,0x00,0x08,0x00},
                              {0x20,0x02,0x80,0x08,0x20,0x02,0x80,0x08},
                              {0x44,0x11,0x44,0x11,0x44,0x11,0x44,0x11},
                              {0xaa,0x44,0xaa,0x11,0xaa,0x44,0xaa,0x11},
                              {0x55,0xaa,0x55,0xaa,0x55,0xaa,0x55,0x44},
                              {0x55,0xbb,0x55,0xee,0x55,0xbh,0x55,0xee},
                              {0xbb,0xee,0xbb,0xee,0xbb,0xee,0xbb,0xee},
                              {0xdf,0xff,0x7f,0xf7,0xdf,0xfd,0x7f,0xf7},
                              {0xdf,0xff,0xfd,0xff,0x7f,0xff,0xf7,0xff},
                              {0xff,0xdf,0xff,0xff,0xff,0xfd,0xff,0xff},
                              {0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff}};
    initgraph(&graphdriver,&graphmode,"");
    getfillsettings(&save);
    setcolor(WHITE);
    rectangle(0,0,539,399);
    for(x=15;x<476;x=x+75)
      {
       rectangle(x,20,x+60,190);          /*画屏上方的框*/
       rectangle(x,210,x+60,380);         /*画屏下方的框*/
      }
    color=-1;
    for(x=16;x<476;x=x+75)
     {
      color=color-1;
      setfillpattern(fillpattern[color],color+1);
      floodfill(x,21,WHITE);             /*对上方7个框用相应图模和颜色填充*/
     }
    for(x=16;x<401;x=x+75)
     {
      color=color+1;
      setfillpattern(fillpattern[color],color+1);
      floodfill(x,220,WHITE);            /*对下方7个框用相应图模和颜色填充*/
     }
    setfillstyle(save.pattern,save.color);   /*恢复原填充模式和颜色*/
    getch();
    closegraph();
}
```

图 10.20 便是该程序画出的图模，它是用打印机将屏幕图形复制下来的。

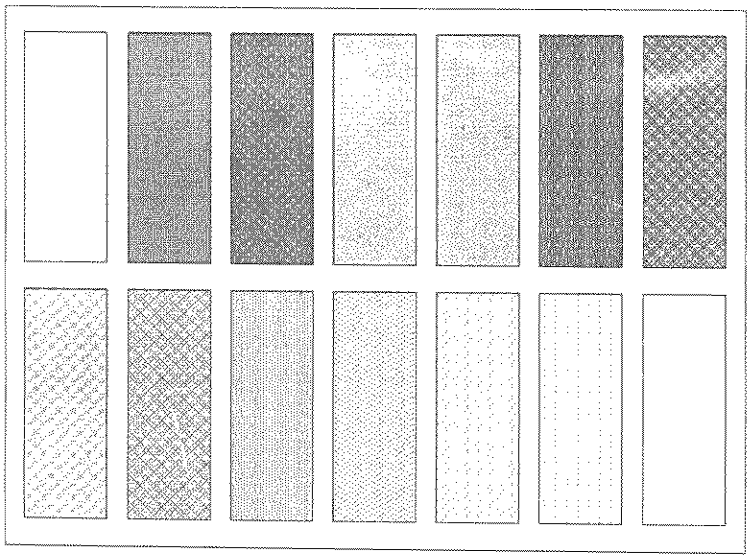


图 10.20 图模

10.13.2 画圆饼图程序

经常需要用图表表示各个量在总量中所占的百分比，如工厂中各个车间的产值占全厂总产值的百分比，百货公司下属各个商店在全年公司总销售额中所占的份额，学校教职员工和学生占学校总人数的百分比，诸如此种颇为多见，若用带有各种颜色的扇形组成的一个饼状图表示，则异常醒目，一目了然。整个圆饼代表总量，每一个扇形代表各个单位所占份额（即百分数），后面程序就画出了一种这样的圆饼图。

程序中，values[]数组代表了 10 个要表示的量，它们在总量中所占比重用一个扇形面积来表示（圆代表总量），画扇形用 pieslice 函数，10 个扇形用同一个圆心（300，240），同一个半径（radius＝140），10 个扇形代表 10 个量（即 10 个 values[]数组元素），它们围起来即形成一个圆（它代表总量）。每个扇形的起始角 begangle 总是前一个扇形的结束角 endangle，结束角可用圆周角 360 度除以该扇形所占总量的比再加上开始角面得到，为了标出各个扇形所占百分比，要知道该扇形在所开图视窗口中的位置，这可以用其半径在 x 轴和 y 轴上的投影而得到，即 x＝rcos(α)，y＝rsin(α)。由于 Turbo C 中 cos 和 sin 函数角用弧度，因而程序中将 begangle 和 endangle 换算成弧度 bega 和 enda。为了在扇形外标出其所占百分比，各乘了 1.5 和 1.2。再由于圆心为（300，240），所以 x，y 要依此参考点。故所标百分比的位置（x，y）如程序中 x，y 的表达式所示（y 中用负号，为了将 y 坐标反过来）。画扇形时，在 pieslice 函数中 endangle 参数加 4，是为了由于计算过程中舍入误差而加的调正量，使得最后扇

形和最初扇形能合并成一完整圆。程序的最后，又在图视窗口的右上角开了一个条形框，里面又画了代表10个扇形颜色的相同颜色条，并标有相应的扇形号，以说明哪个颜色扇形代表哪个顺序号。该号当然也可表示为名称，如一车间，二车间，或一公司、二公司…等等，也就是说哪个扇形代表哪个名称所具有的百分量。

```
#include<math.h>
#include<graphics.h>
#define PI 3.1416
main()
{
   char s[30];
   float values[10]={12.0,16.0,22.0,8.0,10.0,13.0,20.0,14.0,9.0,19.0};
   char *categories[10]={"1","2","3","4","5","6","7","8","9","10"};
   double x,y,bega,enda,midangle;
   float total;
   int radius,begangle,endangle;
   int i,graphdriver,graphmode;
   graphdriver=VGA;
   graphmode=VGAHI;
   initgraph(&graphdriver,&graphmode,"");
   cleardevice();
   setviewport(10,10,639,479,1);                    /*开一个图视窗口*/
   setcolor(3);
   rectangle(20,20,600,460);                        /*画一矩形框*/
   total=0;
   for (i=0;i<=9; i++)
      total=total+values[i];                        /*得到总量*/
   begangle=0;                                      /*开始角为0*/
   radius=140;                                      /*半径*/
   rectangle(530,40,590,180);
   for (i=0;i<=9;i++)
      {
       endangle=360*values[i]/total+begangle;       /*画扇形的结束角*/
       hega=hegangle*PI/180;                        /*换成弧度*/
       enda=hegangle*PI/180;
       midangle=(hega+enda)/2;                      /*得出扇形的中间角*/
       x=300+cos(midangle)*radius*1.5;
       y=240-sin(midangle)*radius*1.2;
       sprintf(s,"%3.2f%",values[i]/total*100);
       setcolor(WHITE);
       outtextxy(x,y,s);
       setfillstyle(1,i+2);
       pieslice(300,240,begangle,endangle+4,radius);
       rectangle(540,55+12*i,560,60+12*i);          /*画代表各扇形的颜色条*/
```

```
        floodfill(550,57+12*i,WHITE);
        outtextxy(565,57+12*i,categories[i]);      /*标上号*/
        begangle=endangle;                         /*上一个扇形的结束角,就是下一
                                                     个扇形的开始角*/
      }
   getch();
   closegraph();
}
```

图 10.21 便是该程序产生的圆饼图。

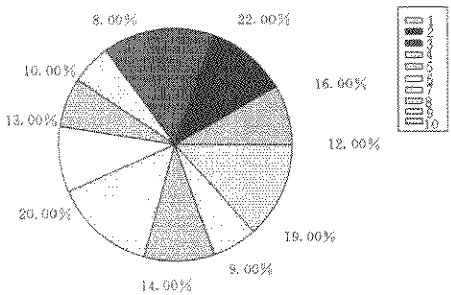


图 10.21 一个圆饼图

10.13.3 画条形图程序

日常生活中,常常需要用条形图表示某种量的随时间变化情况,或某个时间具有的某种事物的量,如各年的生产量、同一种产品各个厂家同期的生产量等,用条形图表示,对比明显、表示形象,使枯燥的数字变得悦目。例如下面所示程序完成了 20 个要用条形图表示的量(量值放在数组 a[]中),其中指针数组 categories[]代表了 x 轴坐标的 20 个坐标量类别的指针,程序中用函数 rectangle 画条形,使用蓝色边框,用相应条形顺序号的颜色进行了填充(当超过 16 时则取模)。为了画出 20 个矩形条,用了 for 循环,每个矩形条宽度为 ddx,每个矩形条间距为 dx-ddx,高度为 dy=a[i]*1.5(乘 1.5 是为了拉长条形,使看起来与窗口比例适中,且更好看)。第二个 for 循环画出了 x 坐标,并标了值,第三个 for 循环画出了 y 坐标,并标出了值。程序最后用 outtextxy 写出 x 所代表的含义,并用 settextstyle(0,1,1)定义了用纵向 8×8 点阵字形写出了 y 坐标的含义。

当需要时,用条形图表示的量 a[],可以用键盘随机输入,这样可以适应各种需要,当然 x 坐标也可用同样方法处理。

```
#include<graphics.h>
main()
{
   char *categories[]={"1","2","3","4","5","6","7","8","9","10","11","12","13","14",
```

```
                    "15","16","17","18","19","20"};
    float a[]={3.9,5.3,7.2,9.6,12.9,17.0,23.2,31.4,39.8,50.2,62.9,76.0,92.
               0,105.7,122.8,131.7,150.7,179.3,203.2,211.0};
    char s[10];
    int graphdriver=VGA;
    int graphmode=VGAHI,i,j,x,n,dx,ddx,y,dy;
    initgraph(&graphdriver,&graphmode,"");
    cleardevice();
    setviewport(20,20,570,450,1);
    setcolor(1);
    setbkcolor(7);
    n=20;
    dx=n;ddx=0.8*dx;y=390;                    /*有20个要用条表示的值*/
    for(i=0;i<=n-1;  i++)                     /*画20个条形图*/
    {x=dx*i+100;
    dy=a[i]*1.5;
    setfillstyle(1,i);
    rectangle(x,y,x+ddx,y-dy);                /*画条形*/
    floodfill(x+1,y-dy+1,1);}
    setcolor(WHITE);
    rectangle(80,390,x+ddx+20,15);            /*画包围图形的矩形框*/
    j=0;
    for(i=108;i<=x+ddx;i=i+20)                /*x轴坐标*/
    {line(i,390,i,400);
     outtextxy(i-4,405,categories[j]);        /*标x坐标值*/
     j++;
     }
     sprintf(s,"%d",j);
     for(j=0;j<=300;j=j+50)
          {
           line(70,390-1.5*j,80,390-1.5*j);   /*y坐标*/
           sprintf(s,"%d",j);
           outtextxy(45,390-1.5*j-3,s);       /*标y坐标值*/
          }
     outtextxy(150,420,"Every year1990-2000");/*标x坐标含义*/
     settextstyle(0,1,1);
     outtextxy(30,40,"Production");           /*标y坐标含义*/
     getch();
     closegraph();
}
```

图 10.22 便是该程序产生的条形图。

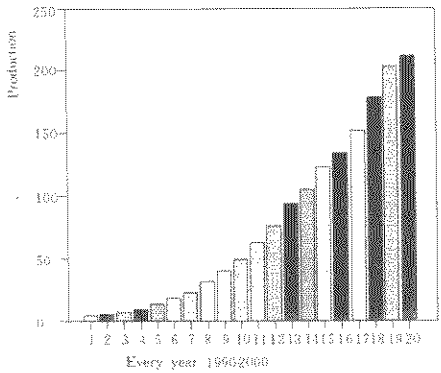


图 10.22 一个条形图

10.13.4 画函数曲线

科学实验中经常需要将一些实验数据点连成曲线，以反映变化规律。有时可根据曲线的趋势，进行曲线模拟。在用 A/D 进行数据采样时，为了醒目地看到现场采得的数据，也经常需要用点线图来描述采集的情况。有时为了演示函数曲线，也将曲线画出来，本程序将存于数组 a[]中的数据用蓝色小圆圈表示了出来，并用洋红色的线将其连接了起来，背景色采用淡灰色，以加强对比。坐标用白线标出，并标有 x 轴和 y 轴含义的说明，实际需要时，可将 a[]用键盘随机输入的方法赋值，也可用函数公式表示。

程序中的第一个 for 循环用于画连线和小圆圈，由于画线时，这个点和那个点是前后有联系的，因而用 lineto 函数，将数据点画小圆圈后(即用 circle 函数)，将无形的画笔坐标移走。为此要用 moveto 函数将其移回到小圆圈的圆心位置，以便和画的下一个点能用 lineto 将其连接起来。画这类图时，要注意 x，y 坐标和实验数据的对应关系，使它们一致，否则将会张冠 李戴。程序中，在第一个 for 循环前首先用 moveto 将画笔移到表示 a[0]的位置，因而 for 循环从 i=1 开始，以便用 lineto 将 a[0]和 a[1]点连起来。由于 y 坐标的分度为 1.5 * J，也即 1.5 * 50，故而实验数据 a[i]也要乘 1.5 倍，以便和纵坐标比例一致。坐标原点相对于开辟的图视窗口为(80,390)，故实验数据值 a[i]对应的(x,y)坐标分别要加上偏移 80 和 100。

程序中第二个 for 循环用于画 x 轴坐标并标值，第三个 for 循环用于画 y 轴坐标，并标上坐标值。程序的最后在 x 轴和 y 轴写上相应的含义。

图 10.23 便是该程序产生的曲线图。

```
#include<graphics.h>
main()
  {
    char *categories[]={"1","2","3","4","5","6","7","8","9","10","11","12","13",
```

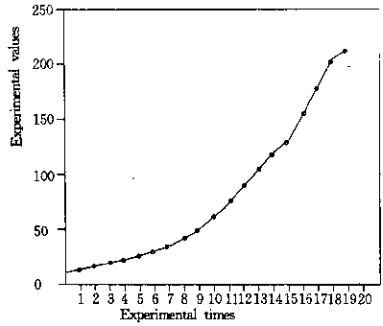


图 10.23　一个函数曲线图

```
                "14","15","16","17","18","19","20"};
    float a[]={3.9,5.3,7.2,9.6,12.9,17.0,23.2,31.4,39.8,50.2,62.9,76.0,92.
               0,105.7,122.8,131.7,150.7,179.3,203.2,211.0};
    char s[10];
    int graphdriver=VGA;
    int graphmode=VGAHI,i,j,x,n,dx,ddx,y,dy;
    initgraph(&graphdriver,&graphmode,"");
    cleardevice();
    setviewport(20,20,570,450,1);
    setbkcolor(7);
    n=20;
    y=390;
    moveto(80,y-a[0]*1.5);
    for(i=1;i<=n-1;i++)
      {
       x=n*i+80;
       dy=a[i]*1.5;
       setcolor(12);
       lineto(x,y-dy);
       setcolor(1);
       circle(x,y-dy,2);
       moveto(x,y-dy);
      }
    setcolor(WHITE);
    rectangle(80,390,100+n*20,15);
    j=0;
    for(i=100;i<=80+n*20;i=i+20)
      {
```

```
        line(i,390,i,400);
        outtextxy(i-4,405,categories[j]);
        j++;
       }
     sprintf(s,"%d",j);
     for(j=0;j<=300;j=j+50)
       {
        line(70,390-1.5*j,80,390-1.5*j);
        sprintf(s,"%d",j);
        outtextxy(45,390-1.5*j-3,s);
       }
     outtextxy(150,420,"Experimental times");
     settextstyle(0,1,1);
     outtextxy(30,40,"Experimental values");
     getch();
     closegraph();
}
```

10.14 图形程序运行的条件

由于图形程序运行并显示图象直接与显示器有关，而如何控制驱动显示器进行显示，Tubro C 并没有向用户提供这种技术，而这也是不必要的，因它与显示器硬件结构息息相关，编程者并不需要知道这些东西，否则太复杂了！但用户的图形程序要能运行并显示，则必须要包含有驱动显示器的这种程序。不同种类的显示器因硬件结构不同，因而驱动程序也不同，这些驱动程序一般由生产厂家提供，对一般标准的显示器，如 CGA、EGA，VGA 等，其驱动程序已经在 Turbo C 系统盘上提供。在用户的图形程序中，进行图形系统初始化时，即执行函数

initgraph(&graphdriver,&graphmode,char path_for_driver);

时，程序就按照 path_for_driver 所指的路径将图形驱动程序装入内存。这样，以后的图形功能才能被支持。若在所指路径下找不到相应显示器的驱动程序，或没有对驱动程序进行装入操作，则运行图形程序时，就会在屏上显示出错信息：

BGI Error: Graphics not initialized (use"initgraph")

当 Turbo C 2.0 安装在软盘上时，没有安装上图形驱动程序（如 CGA.BGI，EGAVGA.BGI等），所以必须在工作盘上复制上这些文件，否则图形程序就无法运行，而出现上述的错误信息。这些图形驱动程序以 BGI.ARC 名存在第五张 Turbo C 系统盘上，它是压缩形式的，因而还必须用该盘上的解压缩程序文件 UNPACK.COM 将其展开，作法是将 5 号系统盘插入 A 盘中，用户工作盘置于 B 盘中，然后用命令：

UNPACK BGI.ARC B: ↙

这样在 B 盘上就生成了各种显示器的图形驱动程序：EGAVGA.BGI，CGA.BGI，HERC.BGI，IBM 3514.BGI，PC3270.BGI 和 BGIOBJ.EXE 等，可以将使用的显示器相应类型驱

动程序保留在自己的工作盘上，其他可以删去。

若 Turbo C 已安装在硬盘上，则这些程序已展开在硬盘上，故而重复直接将其拷贝到自己的工作盘即可，这样在集成环境下，生成的.EXE 文件，就可以在 DOS 下直接运行了。

生成能在 DOS 下直接运行的图形程序的另一种方法是：用 UNPACK BGI.ARC 命令将 BGI.ARC 展开后，用生成的 BGIOBJ.EXE 程序将相应显示器的驱动程序(如 CGA.BGI，EGAVGA.BGI 等)转换成.OBJ 文件，即在 DOS 下用命令(对 EGAVGA 显示器)：

B>A:BGIOBJ A:EGAVGA ↵ (A 盘为工作盘)

这样就在 B 盘上(即用户的库盘上)生成了 EGAVGA.OBJ 文件，然后将该文件连到用户的库盘上 LIB 子目录下的 GRAPHICS.LIB 库中，即用 Turbo C 系统 3 号盘上的 TLIB.EXE 文件，将 EGAVGA.OBJ 和 GRPHICS.LIB 进行连接，可将 3 号盘插入 A 中，用命令

B>A:TLIB LIB\GRAPHICS.LIB+EGAVGA.OBJ

这样就在用户的库盘上的 GRAPHICS.LIB 中连入了 EGAVGA 驱动程序。

若在硬盘上已安装了 Turbo C，则变得简单了，只需键入命令

TLIB LIB\GRAPHICS.LIB+EGAVGA.OBJ ↵

即可，因这些程序均存在于 TC 目录下。

通过以上处理，无论是安装在硬盘上的 Turbo C 或者软盘上的 Turbo C，均在图形库中连入了相应显示器的驱动程序。

最后，在用户程序中的 initgraph()函数调用前要用 registerbgidriver()进行登记，表示相应的驱动程序在用户程序进行连接时，已连接，因而执行 initgraph()时，不必再装入相应的 BGI 程序了，即用(对 EGA，VGA 显示器)：

```
registerbgidriver(EGAVGA_driver);
```

这样执行 initgraph()时，就不用再按该函数的第三个参数 patch for driver 去寻找驱动程序了。

registerbgidriver(*driver(void))中的参数名和相应的图形驱动程序(.BGI)对应关系如下：

图形驱动程序	registerbgidriver 驱动器 driver 参数
CGA.BGI	CGA_driver
EGA.VGA.BGI	EGAVGA drvier
HERC.BGI	HERC_driver
ATT.BGI	ATT_driver
PC3270.BGI	PC3270_driver
IBM8514.BGI	IBM8514_driver

对所有的 Turbo C 图形程序，在 initgraph()函数前，加上上述的函数后(当然对 GRAPHICS.LIB 还要进行如上述的连接操作)，编译连接后生成的 EXE 文件，存放在软盘上后，就变成了一个可独立运行的执行程序了，它不需 Turbo C 环境的支持，可在 DOS 下直接运行。

10.15 图形方式下字型输出的条件

当用文本字型定义函数

settextstyle(int font,int direction,int char size);设置文本输出字形时,由 font 参数给出的字型库必须装入内存,这些字型是 TRIPLEX_FONT、SMALL_FONI、SANSSERIF_FONT 和 GOTHIC_FONT,它们对应的字型库是 trip.chr、litt.chr、sans.chr 和 goth.chr,这些库是由 UNPACK 程序将 BGI 程序扩展后得到的,当在程序中选中了某字形时,便将相应字库调入内存。当我们要使用这些字体输出的图形程序,在脱离 Turho C 集成环境下,仍能独立运行,则必须将相应的字库存在图形程序盘上,其作法如同显示驱动程序一样。即将需要的程序如 trip.chr 等拷贝到程序盘上。另外也可以用 BGIOBJ 程序将相应的字库驱动程序 goth.chr、litt.chr、sans.chr 和 trip.chr 转换成 OBJ 文件。

GOTH.OBJ、LITT.OBJ、SANS.OBJ 和 TRIP.OBJ,如可用命令:BGIOBJ TRIP ↵,这样 trip.chr 就转换成 TRIP.OBJ ↵。然后用 TLIB 程序将其连接到 GRAPHICS.LIB 中,如使用 TRLPLEX_FONT 字形时,可用命令:

C:\TC>TLIB LIB\GRAPHICS.LIB+TRIP.OBJ ↵

这样,GRAPHICS.LIB 中就连接上了相应的字形驱动程序。

当完成上述步骤后,还要在图形初始化之前,即 initgraph()前用函数 registerhgifont(void(*font)(void))进行说明,说明相应 font 的字形库已经连到 CRAPHICS.LIB 中,即连到了图形程序中。如若已装入 TRIP 库,则可写作 registerbgifont(triplex_font)。

上述的这些作法和装显示驱动程序完全一样。其中 registerbgifont()函数中的字形参数和相应的字形驱动程序(.chr)的对应关系如下:

字形驱动程序	registerbgifont()参数名
trip.chr	TRIPLEX_FONT
litt.chr	SMALL_FONT
sans.chr	SANSSERIF_FONT
goth.chr	GOTHIC_FONT

若不经过上述过程,则含有笔划字体的图形函数独立运行时,显示的字体都以 8×8 点阵的缺省字符显示,原来设置使用的字体将不会出现。

第 11 章

菜单设计与动画技术

11.1 菜　单

所谓菜单，故名思义，就是像饭店的菜单，由顾客根据菜单上罗列的菜名来进行点菜，然后店家根据顾客点的菜，进行加工制作，最后送到顾客面前，计算机的菜单也如同上述菜单的作用，因一个实用软件，往往有多种操作功能，可完成多种任务，因此为了方便顾客，在程序开始运行时，首先在屏幕上开一个窗口，显示许多有关的信息，这些信息分解成若干项，这就是通常说的菜单窗口，这些信息项就叫作菜单项，用户可以通过光标移动键，或其他键，选择合适的菜单项，当再一次确认后，程序便执行相应项的功能。一个高质量的菜单，如同一张彩色的美丽画面，不仅使人视觉上有美的感受，更重要的是给用户使用实用软件带来方便，使得操作简化，避免了误操作带来的不良后果，且能迅速得到帮助信息。Turbo C 运行时，屏幕上出现的图象，就是一个典型的菜单，它提示了在 Turbo C 集成环境下，各种选择项，由用户选择和确认，并由 Turbo C 执行。

菜单式样有长条形、矩形。在屏上出现方向有纵向、横向、屏幕滚动(菜单较长，一屏显示不完)，如图 11.1 所示：

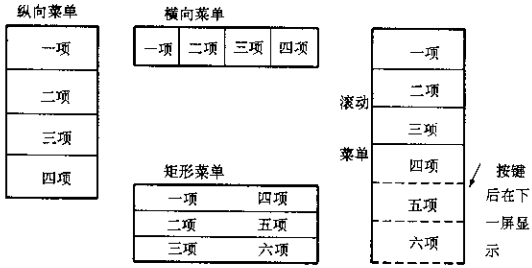


图 11.1　菜单式样

由于显示器有文本显示方式和图形显示方式之分，因面菜单也有文本方式下制作的菜单和图形方式下制作的菜单。就菜单生成的方式面言，又有固定式菜单，弹出式菜单和下拉式菜单之分。固定式菜单，就是程序运行一开始就出现在屏幕上的一种菜单，它仅存在一次，或始终停留在画面上，或功能选择完成后就消失了，不再出现。这种画面，生成简单。所谓弹出式菜单，实际上就是一种动态变化的菜单，如出现菜单后，当选中某项后，菜单立即消失，

出现选中菜单项的功能程序运行，若又需要另外和用户的接口功能，则又弹出一个菜单，供用户选择，一般地说，弹出式菜单仅用在一级深度的选择中，即在该菜单中选中某菜单项后，无需再进行第二次选择，也就是说，选中的该菜单项再不会带有子项要选择。

下拉式菜单则不同于弹出式菜单，它是在选中菜单中某项后，接着在其下面又出现选中项的子项菜单，又要再一次进行选择，也许还要再深度地进行选择，几个下拉式菜单可以同时出现在屏幕上，Turbo C 开始时呈现在用户面前的菜单，就是一种典型的下拉式菜单，这个菜单反映了 Turbo C 集编辑、编译、连接、执行为一体的一个集成化的环境，首先屏上显示一横向主菜单，当选中某一项后，就在此项下面又出现一纵向子菜单。可以用光标移动键向下移动光条，来选择该项的子项，这个菜单就称下拉式菜单。

已经有许多商业软件可用来编制各种形式的菜单，这样使得用户自己编制菜单程序简单方便了，如 Turbo C Tools 就提供这种功能的库函数，但用 C 语言编制菜单更直接，菜单提供的速度更快，需要支持的外部软件更少，不过编程难度增加了。

11.2 菜单设计要点

菜单设计一般要有如下几个部分：

1. 菜单窗口图象的存储和重放

当弹出一个窗口时，原窗口区域内的内容就被窗口覆盖了。为了在弹出的窗口消失后，能恢复原被覆盖的内容，必须事先将原来的内容保存起来，这可以用 getimage()函数，保存区域所占字节数，可以用 imagesize()函数来测试，并由 malloc()函数来根据测得的字节数分配显示存储器的缓冲区。

当要重现被覆盖的内容时，可以用 putimage()函数将保存在缓冲区的原图象通过 OP 操作(一般用 COPY－PUT)重现在原区域。

在文本下，可以使用 gettext()和 puttext()函数来存放某一缓冲区的文本，而存放的缓冲区，则由区域所占的行数和列数乘 2 来决定，这可事先定义一个字符数组来实现，如：

```
char buf[行数 * 列数 * 2];
```

2. 菜单窗口和菜单项的生成

按照用户的按键生成一个相应的设计好的菜单图象，并将事先存放在字符指针数组中的菜单各项内容，填入相应的图象位置中，并用颜色(一般是红色)标出相应项选择对应的热键(即选择该项时，按下的键)。

3. 光条的生成

在菜单项上要压上光条，用户按 Up 或 Down 键，使该光条应在各菜单项上移动，以标明要选择的菜单项。当按回车键或热键后，相应菜单项功能被实现，这可通过对相应菜单项的图象存取和菜单项图象改变背景色后的重放来实现光条。

4. 键识别

当按下菜单所示的功能键(或称热键)或光标移动键时，要得到这些键的键盘扫描码，才能得知何键按下。可设计一个键分析程序将扫描码返回，菜单根据键值，转入相应的功能处理。一般采用 DOS 的 int 86()功能调用，即：如定义取键扫描码函数：

```
int get_key()
```

```
{ union REGS rg;
  rg.h.ah=0;
  int86 (0x16,&rg,&rg);
  return rg.h.ah;
}
```

该函数将返回按键的扫描码，也可用键盘操作函数 bioskey()而得到键的扫描码，它将得到一般键的 ASCII 码，若为特殊键时，将会得到扩展的 ASCII 码，此时低 8 位为 0，高 8 位代表具扩展的 ASCII 码，即扫描码。

5. 菜单的连接

当一个菜单有多展次的子菜单又要选择时，则应有相互连接程序。

6. 功能执行

菜单中应有根据用户的菜单项选择，转入相应的功能程序去执行部分，这个部分可以嵌入到菜单程序中去，对较大功能程序，可以做成功能模块，在菜单中由键分析程序，根据按键直接调用。

11.3 两个菜单程序

11.3.1 菜单程序 1

下面是一个菜单程序，是作者为自己研制的光隔离 A/D 板配的演示程序中简化了的菜单，如图 11.2 所示：

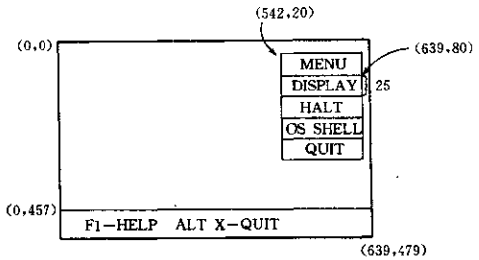


图 11.2 菜单和各特殊坐标

在屏的下方，有一白色长条，显示 F1 和 ALT X 键的作用，屏右上方出现一菜单(显示 MENU)，第一项选择项为 DISPLAY，表明选择该项后，将在屏上显示采样的实时信号图(用象素点显示)。第二项为 HALT，选择该项后，将暂停采样，因而该瞬间的采样图将在屏上保持。第三项为 OS SHELL，当选择该项后，程序将退到 DOS 的 OS SHELL 下(即操作系统的外壳下，实际上就是 COMMAND.COM，它承担分析和执行用户键入的各种命令)，该菜单程序仍保留在内存，若在 TC 集成环境下运行该程序，则在 OS SHELL 下，键入 EXIT，则重新又回到 TC 的集成环境下，第四项为 QUIT，选择该项后，程序将回到 DOS 下(或 TC 下)。

菜单的选择项，可以通过 Down 和 Up 键控制一光条在菜单中上下移动，而选择某项。按该项的第一个字母，便表示确认了该项(第一个字母用红色表示)，程序马上执行实现该项的功能，对第一项 display，当确认后，在屏下方白条框中显示出选择 A/D 通道号(0～15)并按 ENTER 键到转换的字样(pleace input A/D channel number(0～15) and press Enter to conversion)，当输入通道号后，马上又显示"A/D channel number is 通道号"和" press Enter to again choice channel! "，这时屏上将动态地显示实时采样的信号图。

程序开始定义了 10 个键做为功能键，这些键名代表了后面所列出的扫描码，程序中共定义了 4 个函数：get_key()，它用来检查按键后键的值，以确定是哪个功能键，这个函数实际上是一个 BIOS 功能调用(INT16H，功能号为 0)，即键盘中断调用，用来读键盘打入的字符，关于 int86()函数可参阅 Turbo C DOS 调用一章。get_item()函数用来将用户选择的菜单项加上光条并得到选择确认的菜单项，以便在主程序中执行相应的菜单项功能。adin()函数用按输入的通道号启动 12 位 A/D 进行采样，将采得的 12 位二进制数(换算成十进制)返回，以便在主程序中显示。choice_channel()函数，用来接收并检查用户输入的 A/D 通道号，限定为输入 0～15，并以回车符(0x0d)结束，若不是 0～15 的数字或其他字符，则以发声提示输入错，并显示"channel number error! again"信息，要求重新输入，当正确输入后，则返回输入的通道号。

主程序设计了一光条 bar(0,0,90,22)，颜色采用淡洋红，并将该图象存入由指针 buf_curse 指出的内存区中，该内存缓冲区大小恰能容纳下该光条，该光条大小也恰能盖住菜单项，这样将该光条取出并放到选中的某菜单项，且进行异或操作时，选中的菜单项则变成洋红色，即用 putimage (543, 91, row * 25, buf_curse, XOR_PUT)，其中 row 为选中菜单项的行号(有 0,1,2,3 四行)。若对有光条的菜单项，再一次进行 putimage(543, 91, row * 25, buf_curse, XOR_PUT)操作时，则恢复原来的颜色。主程序中/ * display menu * / 后面的程序段用来产生一个大矩形框和在该框下面再产生一个横向的长条。大框就代表显示采样数据点的窗口，下条框显示提示信息，紧跟着的 for 循环用来产生一个菜单，并在各矩形条中显示相应代表的功能信息，并用红色加亮信息的第一个字母，接着将淡洋红色光条放在菜单项上，用 do 循环根据选择的菜单项实现相应的功能。它利用 getitem()函数，得到功能项号(row)，再用 switch(row)语句，转向相应的功能程序段去执行。

在函数 get_item()中，当按 F1 键时应显示帮助信息，由于篇幅关系，未将具体内容写入。实际应用时，只要将帮助信息写入此处即可。图 11.3 为该程序产生的菜单，它是用打印机复制下的图，程序如下：

```
/ * MENU program on graph mode * /
#include <graphics.h>
#include <process.h>
#include <stdlih.h>
#include <dos.h>
#include <ctype.h>

#define TRUE 1
#define FALSE 0
#define KB_S_N_DOWN     80              / * 定义各功能键的扫描码 * /
```

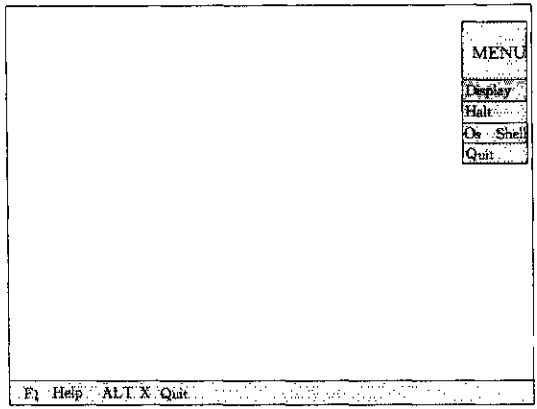


图 11.3 程序产生的菜单画面

```
#define KB_S_N_UP       72
#define KB_S_N_HOME     71
#define KB_S_N_END      79
#define KB_S_N_ENTER    28
#define KB_S_N_F1       59
#define KB_S_N_D        32
#define KB_S_N_H        35
#define KB_S_N_O        24
#define KB_S_N_Q        16
#define KB_S_N_X        45

/* DEFINE FUCTION */
int get_key();                          /*取键值函数*/
int choice_channel(int x, int y);       /*取通道号函数*/
int get_item(int,int);                  /*取菜单项函数*/
int adin(int ch);                       /*12位A/D采样函数*/
void *buf_curse;                        /*缓冲区指针*/

main()
 {
  int driver,mode;
  int ad[640];
  int row,item_num,i,ch,y,n,data,done,k;
  float da;
  unsigned size;
  MENU;
  driver=VGA;
```

```
mode=VGAHI;
initgraph(&driver,&mode,"");
size=imagesize(0,0,90,22);
if (size! =1)buf_curse=malloc(size);
/* get ligh bar */
setfillstyle(1,13);
bar(0,0,90,22);
getimage(0,0,90,22,buf_curse);
setfillstyle(1,BLACK);
bar(0,0,91,23);
/* display menu */
setcolor(YELLOW);
rectangle(0,0,639,479);
rectangle(0,457,639,479);
setfillstyle(1,LIGHTGRAY);
bar(1,458,638,478);
setcolor(BLACK);
outtextxy(10,467,"F1_Help ALT_X-Quit");
setcolor(RED);
outtextxy(10,467,"F1 ALT_X");
setfillstyle(1,BLUE);
rectangle(542,20,635,89);
bar(543,21,634,88);
item_num=4;
setcolor(YELLOW);
   for (k=0;k<item_num;k++)
   { rectangle(542,90+k*25,635,115+k*25);
     floodfill(544,92+k*25,YELLOW);}
     settextstyle(1,HORIZ_DIR,4);
     setcolor(LIGHTRED);
     outtextxy(548,40,"MENU");
     settextstyle(0,HORIZ_DIR,1);
     setcolor(WHITE);
     outtextxy(548,100,"Display");
     outtextxy(548,125,"Halt");
     outtextxy(548,150,"Os shell");
     outtextxy(548,175,"Quit");
   /* added hote key */
     setcolor(LIGHTRED);
     outtextxy(548,100,"D");
     outtextxy(548,125,"H");
     outtextxy(548,150,"O");
     outtextxy(548,175,"Q");
   /* display light bar on farst menu item */
```

```
putimage(543,91,buf_curse,XOR_PUT);
setcolor(WHITE);
row=0;
done=FALSE;
do
  {
   row=get_item(row,item_num);
   switch(row)
    {
      case 0:
      /* do function */
       setcolor(LIGHTRED);
       bar(1,458,638,478);
       outtextxy(5,467,"Please input A/D channel number(0-15)");
       ch=choice_channel(189,467);
       setcolor(14);
       outtextxy(220,467,"Prees Enter to again choice channel!");
       n=450;
       AD:
       do
        {
         for(i=0;i<450;i++)
          {
           da=adin(ch);
           y=n-(int)(da * n/4095);
           putpixel(i+2,ad[i],0);      /* 盖掉旧的显示点 */
           ad[i]=y;
           putpixel(i+2,y,2);          /* 用点显示采样值 */
          }
        }while(! kbhit());
        break;
    case 1:
      putimage(543,91+row * 25,buf_curse,XOR_PUT); /* 光条下移 */
      row=0;
      putimage(543,91+row * 25,buf_curse,XOR_PUT); /* 恢复原来的
                                                      颜色 */
      break;
    case 2:
    /* hait program */
    /* enter Os shell */
       free(buf_curse);
       done=TRUE;
  restorecrtmode();
     system(" ");
```

```
                goto MENU;
              case 3:
              /* enter DOS */
                 done=TRUE;
                 closegraph();
                 exit(0);
               }
            }while(! done);
            restorecrtmode();          /*恢复原来屏幕设置的模式*/
 }
int choice_channel(int x,int y)
  {
   int i,c;
   char num[3]={0,0,0};
   i=0;
   do  {
        num[i]=getch();
        if(isdigit(num[i])&&(i<2))
          {
           i++;
           setcolor(WHITE);
           bar(1,458,638,478);
           outtextxy(5,467,"A/D channel number is");
           setcolor(RED);
           outtextxy(180,467,num);
           continue;
          }
        else
         if(num[i]==0x0d)
            {
             num[i]=0;/* mask Enter key */
             c=atoi(num);
             if((c>=0)&&(c<=15))
              {bar(1,458,170,478);
               outtextxy(5,467,"Sampling channel is");
               break;
              }
            }
            sound(800);
            delay(100);
            nosound();
            outtextxy(210,467,"channel number error! again");
            i=0;
```

```
            strcpy(num,"");
           }while(1);
           return c;
  }
/* take function of menu item */
int get_item(int row,int item_num)
{
 int key,done;
 done=FALSE;
 do
    {
     key=get_key();
     switch (key)
           {case KB_S_N_DOWN:
            /* restore down key */
            putimage(543,91+row*25,huf_curse,XOR_PUT);
            if(row==item_num-1)row=0;
            else row+=1;
            /* display light bar */
            putimage(543,91+row*25,buf_curse,XOR_PUT);
            break;
             case KB_S_N_UP:/* UP KEY */
               putimage(543,91+row*25,buf_curse,XOR_PUT);
               if(row==0)row=item_num-1;
               else row-=1;
               putimage(543,91+row*25,buf_curse,XOR_PUT);
               hreak;
             case KB_S_N_X:
               row=item_num-1;
               done=TRUE;
               break;
             case KB_S_N_ENTER:
               done=TRUE;
               break;
             case KB_S_N_F1:
             /* display help */              /*此处帮助信息可由用户填入*/
               break;
             case KB_S_N_HOME:
               if(row!=0)
                {
                 putimage(543,91+row*25,buf_curse,XOR_PUT);
                 row=0;
                 putimage(543,91+row*25,buf_curse,XOR_PUT); /*光条返回
```

```
    }
    break;
case KB_S_N_END:
  if(row! =item_num-1)
  {
   putimage(543,91+row * 25,buf_curse,XOR_PUT);
   row=item_num-1;
   putimage(543,91+row * 25,buf_curse,XOR_PUT);
  }
  break;
case KB_S_N_D:/ * display key * /
  if(row! =0)
  {
   putimage(543,91+row * 25,buf_curse,XOR_PUT);
   row=0;
   putimage(543,91+row * 25,buf_curse,XOR_PUT);
  }
  done=TRUE;
  break;
case KB_S_N_H:/ * halt key * /
  if(row! =1)
  {
    putimage(543,91+row * 25,buf_curse,XOR_PUT);
    row=1;
    putimage(543,91+row * 25,buf_curse,XOR_PUT);
  }
  done=TRUE;
  break;
case KB_S_N_O:/ * Os shell * /
  if(row! =2)
  {
   putimage(543,91+row * 25,buf_curse,XOR_PUT);
   row=2;
   putimage(543,91+row * 25,buf_curse,XOR_PUT);
  }
  done=TRUE;
  break;
case KB_S_N_Q:/ * quite key * /
  if(row! =3)
  {
   putimage(543,91+row * 25,buf_curse,XOR_PUT);
   row=3;
   putimage(543,91+row * 25,buf_curse,XOR_PUT);
```

```
            }
            done=TRUE;
            break;
          default:
            break;
          }
        }while(! done);
        return row;
      }
                    /* read chare on key,return 16 bit scane code */
                    int get_key()
                    {union REGS rg;
                     rg.h.ah=0;
                     int86(0x16,&rg,&rg);
                     return rg.h.ah;
                     }
int adin(int ch)
{int a,b,c,data;
 outportb(0x308,ch);
 outportb(0x309,0);
 do
 {
  a=inportb(0x30a);
  a=a&0x01;
 }while(a! =0);
 a=inportb(0x309);
 b=inportb(0x30b);
 c=inportb(0x30c);
 data=(c&0x0f) * 256+b;
 return(data);
 }
```

11.3.2 菜单程序 2

作为一个菜单示范程序，现介绍一个实用的下拉式弹出菜单，该菜单在图形方式下使用，在该菜单中，主菜单项用 File、Menul、Menu2、Menu3、Menu4 和 Quit 来表示，实际使用时，可换成符合用户要求的菜单项名，该菜单程序产生的菜单形式，类似于 Turbo C 2.0 的集成开发环境下的菜单。

运行该程序的执行程序，首先在屏上产生一个黑背景下的淡红色框，在框的上部用白色框填上青色，并画出各主菜单项，在主菜单项第一项 File 上盖有一淡红色光条，File 用黄色显示，其它项用黑色显示，当按 ENTER 键或 ↓ 键时，表示选中该项，于是立即弹出一个下拉式子菜单，可以用 ↓ 和 ↑ 键进行子菜单项的选择(此时将会出现一淡红色光条在子菜单项上向下或向上移动)，当选中每一子菜单项后，按回车，表示开始执行该子菜单项的功能。本程

序只是进行了模拟，即马上弹出一个青色窗口，并用白色显示提示，下面有以淡红为底显示黄色的 OK 字样，它表示了该子菜单项已完成了其执行的功能，这时，若再按 ENTER 键，则将又回到主菜单去。当然，按 ESC 键也回到主菜单，一般它表示了中途停止，放弃继续执行而回到主菜单去。比如选中主菜单项中的 Menu1 项（此时淡红色光条盖在此项上），然后按回车，立即在该项下而出现了一个下拉式子菜单，这时再移动光条到 Draw 子菜单项下，表示选中该项，当按 ENTER 键时，便开始执行该项功能，此时在窗口内便随机地画出了各种颜色的、大小不同的方条来，并动态的在变化着，它表示了程序在不断地执行，当按任意键后此过程结束，控制又回到子菜单，若按 ESC 键，则又返回主菜单，若用户此时选择了主菜单项中的 Quit 项，当按 ENTEN 键后，便响喇叭又回到系统，并在屏顶部显示 Good Bye…等字样。

为了便于修改程序及增加程序的易读性，特编了一个头文件 menu3. h，在该头文件中定义了一些常用的控制键名，用它们代表该键对应的扫描码，还定义一个代表 16 色的一颜色字符数组和菜单中将要使用的颜色，另外它还定义了主菜单和子菜单项的项名，若用户想增加主菜单或子菜单项的项名。只要在头文件中的主菜单或子菜单项的指针数组中，增加相应的项名即可。因指针数组并未限定数组元素的个数，比如，若想增加 Menu5 一项在主菜单中，可先在 * MainMenuItem[]指针数组中增加“Menu5”一项（应放在“Menu4”之后），然后再增加一相应子菜单项的指针数组，即可写上：

Static char * SubMenuItem 5[]={……,0}，其大括号中的内容为相应的子菜单项。这样，当对程序编译连接后，将会自动生成新的主菜单项和子菜单项，因面可为不同用户对菜单项的不同要求提供可修改的条件，因面该菜单程序具有实用性和适应性。

这个头文件中还定义了 一个结构，并用 MENUTYPE 命名为新的类型。

主程序调用了许多定义的函数，其中有设置屏幕为 VGAHI 模式的图形初始化函数 InitialGraphics()，当调用该函数对屏幕初始化不成功时（如不是 VGA 显示器等），则返回错误代码（即不是 grok 时），此时等待用户按键，当按任意键后，便调用 GoodBye()函数而响喇叭退出，返回系统。若初始化成功，则程序调用 LoadMainMenu()函数，该函数首先给用 MENUTYPE 类型的 MainMenu 变量中的元素 MainMenu. coor[i]分别赋值，以确定主菜单框的对角坐标，并接着调用 getitemcount()函数而得到主菜单项中最长项的长度和项数，从而计算出两项之间的长度（maxlen），并计算出每个主菜单项长条的坐标[(x_1,y_1),(x_2,y_2)]，还有在其上显示菜单项字符的坐标(x,y)。i 代表菜单项号，j 为每个菜单项长条起始 x 坐标的倍数因子，即前面项的长度和项间隔长度累加和，当乘字符宽度后（宽度 textw=8），即变成 x1 坐标。

LoadSubMenu()函数则用来计算每个主菜单项所对应的子菜单框的对角坐标和子菜单中各项在框中显示的位置坐标。

函数 ManagemainMenu(void)是主菜单管理函数，它首先调用 DisplayMainMenu()函数来画出主菜单，然后根据调用得到按键扫描码的函数 GetKey()得知用户按了什么键（←→）面激活相应的主菜单项（即调用激活主菜单项函数 In_ActiveMainMenuItem()），使淡红色光条出现在选中的主菜单项上，当按↓或 ENTEN 键时，便弹出一个下拉式子菜单，即调用子菜单的管理函数 ManageSubMenu()，若选中的是子菜单项 Quit，则立即返回系统。

ManageSubMenu()是子菜单管理函数。它首先调用画子菜单的函数 DisplaySubMenu

()，该函数定义了一个子菜单框面积大小的图形区域的缓冲区，用远堆分配函数(因该程序在大内存模式下工作)farmalloc()，在内存中设置了一缓冲区，并得到它的指针，然后用getimage()库函数将图形保存到该缓冲区中，接着画出选中的子菜单框，并在框内填上子菜单项。子菜单管理函数根据 Getkey()函数得到的键扫描码，决定程序的走向，当按 ESC 键时，则调用 Exitmenu()函数，恢复原子菜单占用区域的内容(即子菜单消失)，并释放缓冲区，回到主菜单，若是按上下(↑、↓)键，则淡红色光条移到选中的子菜单项上，即调用 In_ActiveSubmenuItem()函数，当按 ENTER 键时，便抹去子菜单而执行相应子菜单项的功能。

FuncProc()函数用来执行子菜单项的功能。它按照指示代码ID，转向不同的功能，一般这些功能在实用程序中用模块代替，即执行由许多函数组成的子菜单项功能模块，本菜单程序仅作了一些模拟，如若选中 Quit 项，此时 ID 为 6，即调用 Goodbye 模拟(仅由一个函数组成)，显示结束信息，并回到系统，而当选中主菜单第二项对应的子菜单，且又选中子菜单的第二项时，此时 ID=101，它代表 Draw 子菜单项，于是程序执行该项功能，即调用 Draw 功能模块，该功能模块将画出一个方条，它们大小、颜色均不等，是在随机地变化着。若再按 ENTER 或 ESC 键，程序则又回到主菜单项，此时若选第一项，并选相应子菜单中的第一项，则将调用 MessageBox()函数，它将显示 This is example! 信息。由子程序中仅安排了三个子菜单中的项执行相应功能，即第二项主菜单中子菜单 Draw 项，第四个主菜单项中的 MoveText 子项，第五个主菜单中的 About 子项，其它项均用同一种操作来模拟。所以除了这三个子项外，其它项的执行结果均将画出一个用青色填充的长方块并带有阴影。然后调用 MessageBox()函数，在其上写上 Tbis is an example! 信息。实用时用户根据自己的需要，换成真正的功能函数即可。

当执行子菜单项功能后，若此时再按任意键，便恢复信息框内原来的内容(即恢复为蓝背景色)，并释放缓冲区，然后返回主菜单。

功能模块中，About()将在屏中部画出一长方形兰框，并画出其黑色的投影，在框中写上 MENU EXAMPLE IN GRAPHICS MODE 6—1994 等字样，并等待按键，当接收到按键后，便释放占用的缓冲区。

MoveText()功能函数将实现一个黄色的 Menu exmple! 字样从屏的右边向左边移动的图象。

这个菜单程序较大，因而要求在大内存模式下(lage)进行编译。

所附图 11.4 是该程序产生的主菜单图，当时表示已选中了主菜单项 Menu1，因而该项下面出现了一个下拉子菜单，图 11.5 表示选择了子菜单项 Item21 后，执行该项的图示。这两幅图均是用打印机复制屏幕的图。

```
#define   INSERT      0x5200
#define   ESC         0x001b
#define   TAB         0x0f09
#define   RETURN      0x000d
#define   RIGHT       0x4d00
#define   LEFT        0x4b00
#define   UP          0x4800
```

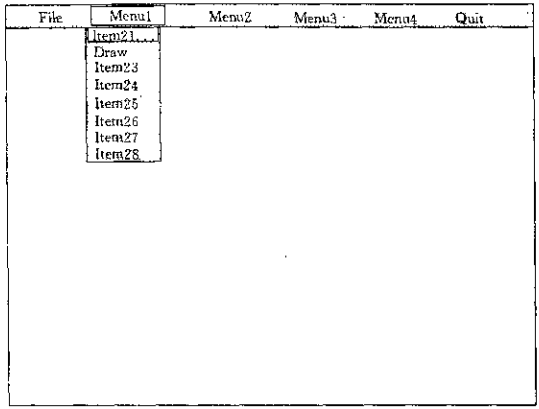


图 11.4　选中一个主菜单项后的菜单图

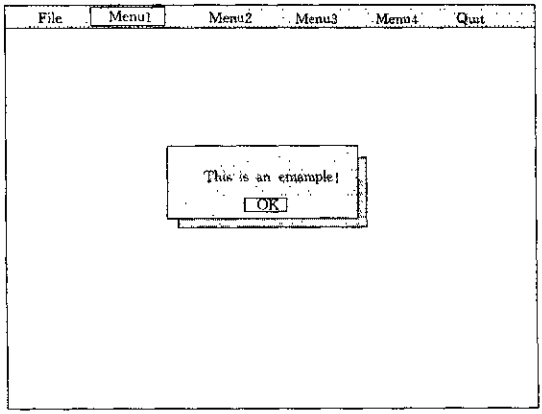


图 11.5　选中下拉子菜单中第一项后的图

```
#define  DOWN    0x5000
#define  BS      0x0e08
#define  HOME    0x4700
#define  END     0x4f00
#define  PGUP    0x4900
#define  PGDN    0x5100
#define  DEL     0x5300
```

```
#define  F1                0x3b00
#define  F2                0x3c00
#define  F3                0x3d00
#define  F4                0x3e00
#define  F5                0x3f00
#define  F6                0x4000
#define  F7                0x4100
#define  F8                0x4200
#define  F9                0x4300
#define  F10               0x4400

#define  MenuInGround          1
#define  MenuActiveGround      2
#define  MenuActiveItem        3
#define  MenuInItem            4
#define  MenuEdge              5
#define  WindowEdge            6
#define  WindowGround          7
#define  PopWindow             8

unsigned char AllColors[20]={0,CYAN,LIGHTRED,YELLOW,BLACK,WHITE
            ,LIGHTRED,BLUE,CYAN,9,10,11,12,13,14,15};

#define INUM      10          /* maximum 10 items in each MENU */
typedef struct   _menu
{
 int  coor[4];                /* The menu's area */
 int  itemcoor[4*INUM];       /* max 10 item, each with x1,y1,x2,y2 */
 int  itemdispxy[2*INUM];     /* Actually disp item name's X coor */
 char select;                 /* selected item's ord */
 char itemnum;                /* The number of items in this menu */
 char **itemname;             /* item's name(point to static data) */
 int  COMMAND_ID[INUM];       /* Key number */
}MENUTYPE;

static char *MainMenuItem[]= {"File","Menu1","Menu2","Menu3","Menu4",
                              "Quit",0};
static char *SubMenuItem1[]= {"Item11","Item12...","Item13", "Item14",
                              "Item15","Item16","Quit",0};
static char *SuhMenuItem2[]= {"Item21...","Draw","Item23", "Item24","Item25",
                              "Item26","Item27", "Item28", 0};
static char *SuhMenuItem3[]= {"Item31...","Item32...","Item add.....",
                              "Item33",0};
static char *SubMenuItem4[]= {"Item41...","Item42...","Moving Text",
                              "Item44","Item45","itime**",0};
static char *SuhMenuItem5[]= {"Item51...","About...","Item53", 0};
```

```
static char * SubMenuItem6[]= {0};
#include <graphics.h>
#include <alloc.h>
#include <process.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <dos.h>
#include <bios.h>
#include <conio.h>
#include <ctype.h>
#include "menu3.h"

#define  WindowMinX 1
#define  WindowMinY 25
#define  WindowMaxX 638
#define  WindowMaxY 478

void getitemcount(char * * ,int * ,int * ,int * );
void InitialGraphics(void);
void LoadMainMenu(void);
void LoadSubMenu(char * *name, int ord);
int  ManageSubMenu(void);
void ManageMainMenu(void);
void DisplayMainMenu(void);
int  DisplaySuhMenu(int ord);
void Help_page_show(void){return;};
void Exitmenu(int);
void In_ActiveMainMenuItem(int select, char In_Active);
void In_ActiveSubMenuItem(int ord, int select, char In_Active);
int  GetKey(void);
void FuncProc(int ID);
void Draw(void);
void MoveText(void);
void About(void);
void MessageBox(char *Message);
void GoodBye(char *pcInf1, char *pcInf2);

MENUTYPE  MainMenu, SubMenu[INUM];
int  maxx, texth, textw;
void *MenuImageBuf;

void main(void)
{
  InitialGraphics();                    /*设置图形方式*/
```

```
    maxx = getmaxx();
    texth = textheight("text");
    textw = textwidth("t");

    LoadMainMenu();
    LoadSubMenu(SubMenuItem1, 0);
    LoadSubMenu(SubMenuItem2, 1);
    LoadSubMenu(SubMenuItem3, 2);
    LoadSubMenu(SubMenuItem4, 3);
    LoadSubMenu(SubMenuItem5, 4);
    LoadSubMenu(SubMenuItem6, 5);

    ManageMainMenu();
}

void ManageMainMenu(void)        /* 主菜单管理 */
{
    unsigned key;
    int ID;

    DisplayMainMenu();

    for(key=0 ; ; )
     {
      key=GetKey();

      switch(key)
       {
        case LEFT:
         In  ActiveMainMenuItem(MainMenu.select,0);
         if(MainMenu.select<1) /* Already leftest item in mainmenu */
              MainMenu.select=MainMenu.itemnum-1;
           else MainMenu.select--;
           In_ActiveMainMenuItem(MainMenu.select,1); /* Light bar */
           break;
        case RIGHT:
         In_ActiveMainMenuItem(MainMenu.select, 0);
         if(MainMenu.select>MainMenu.itemnum-2) /*already rightest */
           MainMenu.select=0;
         else MainMenu.select++;
        In_ActiveMainMenuItem(MainMenu.select,1);
        break;
       case DOWN:
       case RETURN:
```

```
        switch(MainMenu.COMMAND_ID[MainMenu.select])
        {
        case 5:  /* Exit to DOS */
          if (key==DOWN)
            break;
          closegraph();
          GoodBye("Goodbye from ","the MENU DEMO in graphics mode.");
          break;
        default:
          ID = ManageSubMenu();
          if (ID >=0)
            FuncProc(ID);
           break;
           }

       }
     }
}

void FuncProc(int ID)        /*执行选中的子菜单项功能程序*/
{
  switch ( ID )
  {
  case 6: /* 006 File|Quit */
    closegraph();
    GoodBye("Goodbye from " , "the MENU DEMO in graphics mode.");
    break;
  case 101;                  /*101表示主菜单第二项所对应的子菜单中的第一项*/
    Draw();
    break;
  case 302:
    MoveText();
    break;
  case 401:
    About();
    break;
  default:
      /*Clear window*/
    setfillstyle(SOLID_FILL, AllColors[WindowGround]);
    bar(WindowMinX,WindowMinY,WindowMaxX, WindowMaxY);
    MessageBox("This is an example!");
    hreak;

  }
```

```
}

/* ManageSubMenu() return an int value that is          */
/*  ==-2: If there is a error                           */
/*  ==-1: Return to main menu, nothing to do            */
/*  ==0 or >0: the COMMAND_ID of a menu item            */
int ManageSubMenu(void)          /*子菜单管理函数*/
{
  MENUTYPE   *mn=&SubMenu[MainMenu.select];

  unsigned key;

  if (DisplaySubMenu(MainMenu.select)) /* Out of memory */
    return -2;

    for(key=0 ; ; )
     {
      key=GetKey();

      switch(key)
      {
       case ESC: /* return to MainMenu */
         Exitmenu(MainMenu.select);
         return-1;
       case UP:
         In_ActiveSubMenuItem(MainMenu.select, mn->select,0);
         if(mn->select>0)
           mn->select--;
         else mn->select=mn->itemnum-1;
           In_ActiveSubMenuItem(MainMenu.select, mn->select,1);
         hreak;
        case DOWN:
          In_ActiveSubMenuItem(MainMenu.select, mn->select,0);
          if(mn->select<mn->itemnum-1)
            mn->select++;
          else mn->select=0;
            In_ActiveSubMenuItem(MainMenu.select, mn->select,1);
          hreak;
        case LEFT:
          Exitmenu(MainMenu.select);
        In_ActiveMainMenuItem(MainMenu.select,0);
          if(MainMenu.select<1)/*Already leftest item in mainmenu*/
            MainMenu.select=MainMenu.itemnum-1;
          else MainMenu.select--;
```

```
            In_ActiveMainMenuItem(MainMenu.select,1);/*Light bar*/
            if(DisplaySubMenu(MainMenu.select))/*Out of memory*/
              return -2;
            mn=&SubMenu[MainMenu.select];
            break;
          case RIGHT:
            Exitmenu(MainMenu.select);
            In_ActiveMainMenuItem(MainMenu.select, 0);
            if(MainMenu.select>MainMenu.itemnum-2)/*already rightest*/
              MainMenu.select=0;
            else MainMenu.select++;
            In_ActiveMainMenuItem(MainMenu.select,1);
            if (DisplaySubMenu(MainMenu.select)) /* Out of memory */
              return -2;
            mn=&SubMenu[MainMenu.select];
            break;
          case RETURN:
            Exitmenu(MainMenu.select);
            return mn->COMMAND_ID[mn->select];
        }
    }
}

void LoadMainMenu()
{
int count,totallen,maxlen,i,j=1;

MainMenu.coor[0]=0;      MainMenu.coor[1]=0;
MainMenu.coor[2]=maxx; MainMenu.coor[3]=3*texth;   /*主菜单框的两对角坐
                                                     标*/
getitemcount(MainMenuItem,&count,&totallen,&maxlen);
MainMenu.itemnum=count;

maxlen=(maxx/textw-totallen)/count;/*length(char) between 2 items(horiz.)
                                    */
for(i=0,j=0;i<count;i++)
  {
   MainMenu.itemcoor[i*4]  = j*textw;                /*x1*/
   MainMenu.itemcoor[i*4+1] = MainMenu.coor[1];    /*y1*/
   MainMenu.itemcoor[i*4+2]=textw*(j+maxlen+strlen(MainMenuItem
                                        [i]));       /*x2*/
   MainMenu.itemcoor[i*4+3] = MainMenu.coor[3];

   MainMenu.itemdispxy[i*2] = textw*(j+0.5*maxlen); /*actuall disp X*/
```

```
    MainMenu.itemdispxy[i*2+1] = texth;      /* actuall disp Y */

    MainMenu.COMMAND_ID[i] = i;

    j+=maxlen+strlen(MainMenuItem[i]);
      }
    MainMenu.itemname=MainMenuItem;
}

void LoadSubMenu(char **name,int ord)
{
    int count,i,j,maxlen;

    getitemcount(name,&count,&i,&maxlen);
    SubMenu[ord].itemname=name;

    SubMenu[ord].itemnum=count;

    j=textw*(maxlen+2);
    if(j+MainMenu.itemcoor[4*ord]<maxx)/*left text justification is OK*/
      {
       SubMenu[ord].coor[0]=MainMenu.itemcoor[4*(ord)]; /*item's x1*/
       SubMenu[ord].coor[2]=SubMenu[ord].coor[0]+j;
       }
      else      /* Use right justification because of space */
       {SubMenu[ord].coor[2]=maxx;      /* left just use X2 */
        SubMenu[ord].coor[0]=SubMenu[ord].coor[2]-j;
       }
     SubMenu[ord].coor[1]=MainMenu.itemcoor[4*ord+3]; /* The item's y2
                                                        */
     SubMenu[ord].coor[3]=SubMenu[ord].coor[1] + count*(texth+6);

        /* 6-pixel between items of vert. menu */
     for(i=0;i<count;i++)
       {
        SubMenu[ord].itemcoor[i*4]=SubMenu[ord].coor[0];
        SubMenu[ord].itemcoor[i*4+1]=SubMenu[ord].coor[1]+i*(6+texth);
        SubMenu[ord].itemcoor[i*4+2]=SubMenu[ord].coor[2];
        SubMenu[ord].itemcoor[i*4+3]=SubMenu[ord].itemcoor[i*4+1]+texth
                                     +6;

        SubMenu[ord].itemdispxy[i*2]=SubMenu[ord].itemcoor[i*4]+textw;
        SubMenu[ord].itemdispxy[i*2+1]=SubMenu[ord].itemcoor[i*4+1]+3;
```

```
    SubMenu[ord].COMMAND_ID[i] = ord * 100+i;
  }
}


void DisplayMainMenu(void)          /* 显示主菜单 */
{
unsigned i;

setcolor(AllColors[WindowEdge]);
rectangle(0,0,getmaxx(), getmaxy());

setcolor(AllColors[MenuEdge]);
setfillstyle(SOLID_FILL,AllColors[MenuInGround]);
bar3d(MainMenu.coor[0],MainMenu.coor[1],MainMenu.coor[2],
                                MainMenu.coor[3],0,0);
setcolor(AllColors[2]);
for(i=0;i<MainMenu.itemnum;i++)
  {
  if (MainMenu.select==i)
   In_ActiveMainMenuItem(i,1);
   else setcolor(AllColors[MenuInItem]);
  outtextxy((
  MainMenu.itemdispxy[i*2]),MainMenu.itemdispxy[i*2+1],
                              MainMenu.itemname[i]);
  }
}

int  DisplaySubMenu(int ord)      /* 显示子菜单 */
{  unsigned        size,i;
   MENUTYPE       *mn=&SubMenu[ord];

   size=imagesize(mn->coor[0],mn->coor[1],
                          mn->coor[2],mn->coor[3]);
  if(NULL==(MenuImageBuf=(unsigned char *)farmalloc(size)))
   {
    clearviewport();
    outtextxy(100, 100,"Out of memroy!");
    outtextxy(100, 110, "Press any key...");
    getch();
    return 1;
   }
getimage(mn->coor[0],mn->coor[1],mn->coor[2],mn->coor[3],MenuIm-
ageBuf);

setcolor(AllColors[MenuEdge]);
```

```
setfillstyle(SOLID_FILL,AllColors[MenuInGround]);
bar3d(mn->coor[0],mn->coor[1],mn->coor[2],mn->coor[3],0,0);
for(i=0;i<mn->itemnum;i++)
   {
    if(i==mn->select)
      In_ActiveSubMenuItem(ord, i, 1);
    else
     {
      setcolor(AllColors[MenuInItem]);
       outtextxy(mn->itemdispxy[i*2], mn->itemdispxy[i*2+1], mn->
itemname[i]);
     }
    }
  return 0;
}

static void  Exitmenu(int ord)  /*释放缓冲区,返主菜单*/
{  MENUTYPE *mn=&SubMenu[ord];
  if ( MenuImageBuf )
    {
      putimage(mn->coor[0],mn->coor[1],MenuImageBuf,COPY_PUT);
      farfree(MenuImageBuf);
     }
}

void In_ActiveMainMenuItem(int select, char In_Active)
{                          /*激活主菜单项*/
   MENUTYPE *tp = &MainMenu;
   int  start,end,bot,top;
   start = tp->itemcoor[select*4]+8;
   top = tp->itemcoor[select*4+1]+2;
   end = tp->itemcoor[select*4+2]-8;
   bot = tp->itemcoor[select*4+3]-2;
   if (In_Active==0)     /* off */
    {
     setfillstyle(SOLID_FILL,AllColors[MenuInGround]);
     bar(start,top,end,bot);
     setcolor(AllColors[MenuInItem]);
     outtextxy(
     tp->itemdispxy[select*2],tp->itemdispxy[select*2+1],
                              tp->itemname[select]);
    }
    else
    {
```

```
    setfillstyle(SOLID_FILL,AllColors[MenuActiveGround]);
    bar(start,top, end,bot);
    setcolor(AllColors[MenuActiveItem]);
    outtextxy(
    tp->itemdispxy[select * 2],tp->itemdispxy[select * 2+1],
                              tp->itemname[select]);
    }
}

void In_ActiveSubMenuItem(int ord, int select, char In_Active)
{                              /* 激活子菜单项 */
  MENUTYPE *tp=&SubMenu[ord];
  int  start,end,bot,top;

  start = tp->itemcoor[select * 4]+2;
  top = tp->itemcoor[select * 4+1]+1;
  end = tp->itemcoor[select * 4+2]-2;
  bot = tp->itemcoor[select * 4+3]-1;
   if (In_Active==0)   /* off */
    {
    setfillstyle(SOLID_FILL,AllColors[MenuInGround]);
    bar(start,top,end,bot);
    setcolor(AllColors[MenuInItem]);
    outtextxy(
    tp->itemdispxy[select * 2],tp->itemdispxy[select * 2+1],
                              tp->itemname[select]);
    }
    else
    {
    setfillstyle(SOLID_FILL,AllColors[MenuActiveGround]);
    bar(start,top,end,bot);
    setcolor(AllColors[MenuActiveItem]);
    outtextxy(
    tp->itemdispxy[select * 2],tp->itemdispxy[select * 2+1],
                               tp->itemname[select]);
    }
}

void InitialGraphics(void)     /* 图形方式初始化 */
{
  int driver=VGA, mode=VGAHI;
  int errorcode;

  initgraph(&driver,&mode,"c:\\tc");
```

```
  /* read result of initialization */
  errorcode = graphresult();
  if (errorcode != grOk)  /* an error occurred */
    {
     printf("Graphics error: %s\n", grapherrormsg(errorcode));
     printf("Press any key to halt:");
     getch();
     GoodBye("Graphics error. ", "Try again!");
    }
}

int  GetKey(void)              /*得到按键的扫描码*/
{

  int key;

  key=bioskey(0);
  if(key==F1)                  /*若是F1键,则显示帮助信息*/
   Help_page_show();
  else if (key<<8)             /*若键值低8位非零,便是ASCII码*/
   {
    key = key&0x00ff;          /*取得ASCII码*/
    if (isalpha(key))          /*是字母否*/
      key = toupper(key);      /*变成大写字母*/
    }
    return key;
}

static void getitemcount(char **s,int *count,int *len,int *maxlen)
{              /*计算主菜单的项数和最长项长度*/
  int i,j;

  for((*maxlen)=0,(*len)=0,i=0; s[i]!=NULL; i++)
   {
    j=strlen(s[i]);
    (*len)+=j;
    if(j>(*maxlen)) (*maxlen)=j;
   }
  (*count)=i;
}

void Draw(void)
{
  int x, x1, x2, y1, y2, ch;
```

```
  randomize();  /*对随机函数发生器初始化*/
  do
   {
    x1 = random(WindowMaxX-WindowMinX)+WindowMinX;
    y1 = random(WindowMaxY-WindowMinY)+WindowMinY;
    x2 = random(WindowMaxX-WindowMinX)+WindowMinX;
    y2 = random(WindowMaxY-WindowMinY)+WindowMinY;
    ch = random(16);
    setfillstyle(SOLID_FILL, ch);
    if (x1>x2)
     {
      x = x1; x1= x2; x2=x;
     }
    if (y1>y2)
     {
      x = y1; y1 = y2; y2 = x;
     }
      bar(x1, y1, x2, y2);  /*随机产生位于菜单窗口中的各种颜色条*/
      delay(100);
     }while(! bioskey(1));  /*当按任意键时结束*/
}

void About(void)
{
  int MidX, MidY, x1, y1, x2, y2, size;
  void *Buffer;

  MidX = (WindowMaxX - WindowMinX)/2;
  MidY = (WindowMaxY - WindowMinY)/2;

  x1 = MidX-90; x2=MidX+90; y1=MidY-100; y2=MidY+100;
  size = imagesize(x1, y1, x2, y2);  /*取设定窗口中包含的字节数*/
/* Clear window */
  setfillstyle(SOLID_FILL, CYAN);
  bar(WindowMinX,WindowMinY,WindowMaxX-1, WindowMaxY-1);
  if ((Buffer=malloc(size))==NULL)  /*若没有存储区可用*/
   {
    outtextxy(100,100,"Out of memorry! Press any key to continue.");
    getch();
   }
  else
   {
    getimage(x1,y1, x2,y2, Buffer);    /*保存设定区域中的内容*/
```

```
    setfillstyle(SOLID_FILL, BLUE);
    bar(x1, y1, x2-9, y2-9);          /*画一蓝色长方形*/
    setfillstyle(SOLID_FILL, BLACK);
    bar(x1+8, y2-8, x2, y2);          /*画出黑色的阴影*/
    bar(x2-8, y1+8, x2, y2);
    setcolor(WHITE);
    settextjustify(CENTER_TEXT, CENTER_TEXT);
    outtextxy(MidX-4, MidY-40, "MENU EXAMPLE");  /*写上说明*/
    outtextxy(MidX-4, MidY-20, "IN GRAPHICS MODE");
    outtextxy(MidX-4, MidY+25, "6-1994 by");
    outtextxy(MidX-4, MidY+45, "X. Y. CAO");
    getch();

    putimage(x1,y1,Buffer, COPY_PUT);  /*恢复原设定区域中的内容*/
    free(Buffer);                      /*释放缓冲区*/
    settextjustify(LEFT_TEXT, TOP_TEXT);
   }

}

void MessageBox(char *Message)          /*产生信息框*/
{
  int MidX, MidY, x1, y1, x2, y2, size, Len;
  void *Buffer;

  MidX = (WindowMaxX - WindowMinX)/2;
  MidY = (WindowMaxY - WindowMinY)/2;
  Len = 8*(strlen(Message)+1)/2;
  x1 = MidX-Len-5*8; x2=MidX+Len+5*8; y1=MidY-50; y2=MidY+
50;

  size = imagesize(x1, y1, x2, y2);
  if ((Buffer=malloc(size))==NULL)
   {
    outtextxy(100,100,"Out of memorry! Press any key to continue.");
    getch();
   }
  else
   {
    getimage(x1,y1, x2,y2, Buffer);
    setfillstyle(SOLID_FILL, AllColors[PopWindow]);
    bar(x1, y1, x2-13, y2-13);  /*画长的深灰色填充条*/
    setfillstyle(SOLID_FILL, BLACK);
    bar(x1+12, y2-12, x2, y2);  /*画阴影*/
```

```
    bar(x2-12, y1+12, x2, y2);
    setcolor(WHITE);
    settextjustify(CENTER_TEXT, CENTER_TEXT);
    outtextxy(MidX, MidY-20, Message);      /*写信息*/
    setfillstyle(SOLID_FILL, LIGHTRED);
    bar(MidX-3*8, MidY+16, MidX+3*8, MidY+34);  /*画一淡红色小条*/
    setcolor(YELLOW);
    outtextxy(MidX, MidY+26, "OK");             /*画上黄色 OK*/
    getch();

    putimage(x1,y1,Buffer, COPY_PUT);           /*恢复原内容*/
    free(Buffer);
    settextjustify(LEFT_TEXT, TOP_TEXT);
  }
}

void MoveText(void)          /*产生从右向左移动的 Menu example 字样*/
{
  int i;

/* Clear window */
  setfillstyle(SOLID_FILL, BLUE);
  bar(WindowMinX,WindowMinY,WindowMaxX, WindowMaxY);

  setcolor(YELLOW);
  for ( i=300; i>1; i--=2)
   {
    outtextxy(i, 100, "Menu example!");
    delay(50);
    bar(i, 100, i+104, 108);
   }
}

void GoodBye(char *pcInf1, char *pcInf2)  /*产生 Goodbye…字样并响嘀叭回到系统
                                            */
{
  int f;

  clrscr();
  window(1,1,80,1);
  textbackground(LIGHTBLUE);
  clrscr();

  highvideo();
```

```
    textcolor(WHITE);
    cprintf(pcInf1);
    textcolor(YELLOW);
    cprintf(pcInf2);
    window(1,1,80, 25);
    printf("\n");
    for(f=400; f<800; f+=100)
     {
      sound(f);
      delay(100);
     }
    nosound();
    exit(0);
}
```

11.4 动画技术

利用人的视觉暂留这一生理特点，即对动态的图象变化，仅能分辨出时间间隔为 25 毫秒左右的变化，若太快，则分辨不出了，因而可以用类似电影的方法，将一个图象分解成不同时间出现的图象，然后一张张快速呈现在屏幕上，从视觉效果上看，就如同这些画面在连续变化一样，因而给人以动的视觉感觉。在屏幕上制作动画，这也是目前热门的题目。动画技术是计算机图形学中的一个内容，它可用于游戏娱乐，辅助教学，科学实验模拟、论证、仿真等计算机辅助设计(CAD)。Turbo C 提供的一些图形处理函数可用于动画设计，前面已有简短的例子作了演示。总结起来，实现动画，可用如下方法来实现：

11.4.1 利用动态开辟图视口方法

如第 10 章 10.10.2 节中的程序例所示，在位置动态变化，但大小不变的图视口中(用 setviewpot()函数)，设置固定图形(也可是微小变化的图象)，这样呈现在观察者面前的是当前图视口位置在动态变化，因而在屏上看到的图象就好像在动态变化一样。采用这种方式对较复杂图形不宜，因在图视口内画这种图形要占较长时间，这样图视口位置切换的时间就变得较长，因而动画效果就变差。

11.4.2 利用显示页和编辑页交替变化

如 10.10.2 节中的程序例所示，将当前显示页和编辑页分开(用 setvisualpage()和 setactivepage()函数)，在编辑页上画好图形后，立即令该页变为显示页显示，然后在上次的显示页上(现在变为编辑页)进行画图，画好后，又再次交换，如此编辑页和显示页反复地交换，在观察者的视觉上，就出现了动画的效果。要让页的交替速度快，唯一的办法是缩短在页上的画图时间(即采用优化的画法)。

下面例子程序演示了利用这种方法产生的动画效果。

该程序利用显示页和编辑页交替变化，在视觉效果上产生了动画的形象。程序中用

welcome 作画面，在两个页面上用不同颜色交替写上该字，并交替将写上字的画面变成显示页，并逐渐在依次的页面上放大字体，这样看起来有一个从小到大，从远到近的 welcome 动态地出现在屏中间，好像是一个特写镜头。

```
#include<graphics.h>
main()
  {
    int graphdriver=VGA;
    int graphmode=VGAMED;
    int i,height,width;
    unsigned char *temp="Welcome";
    initgraph(&graphdriver,&graphmode,"");
    settextjustify(LEFT_TEXT,TOP_TEXT);
    cleardevice();
    for(i=1;i<11;i++)
     {
      setvisualpage(0);
      setactivepage(1);
      cleardevice();
      setcolor(12);
      sethkcolor(BLUE);
      settextstyle(TRIPLEX_FONT,HORIZ_DIR,i);
      width=textwidth(temp);
      height=textheigh(temp);
      outtextxy((639-width)/2,175-height/2,temp);
      setvisualpage(1);
      setactivepage(0);
      cleardevice();
      setcolor(10);
      settextstyle(TRIPLEX_FONT,HORIZ_DIR,i++);
      width=textwidth(temp);
      height=textheight(temp);
      outtextxy((639-width)/2,175-height/2,temp)
     }
    getch();
    closegraph();
}
```

11.4.3 利用画面存储再重放的方法

如下例所示，同制作幻灯片一样，将整个动画过程变成一个个片断，然后存到显示缓冲区内，当把它们按顺序重放到屏幕上时，就出现了动画效果，这可以用 getimage()和 Putimage()函数来实现，这种方法较前两种都快，因它已事先将要重放的画面画好了。余下的问题，就是计算应在什么位置重放的问题了。

该程序中演示了利用这种方法产生的动画效果。程序将用 circle 函数画出的并用洋红色填充圆的图象用 getimage 函数存到内存 buffer 中，然后再用 putimage 函数将该圆放到屏的原来圆的相对位置，接着 do 循环不断地用 putimage 函数复制两个小圆(注意！每复制一次，由于存的图象面积覆盖了原来的圆，故只显示一个面)，其方向是左边圆沿 x 轴正向复制，右边圆沿 x 轴负向复制，由于复制速度较快，因而动态地看到小球相向运动，直至碰撞(即两个小球圆心相距 60 时，或者说 i=184 时)，接着 for 循环又不断复制两球，直至球碰到屏的尽端。当不按键时，又重复 do 循环，如此反复，就看到了两个洋红色小球碰撞、弹回、又碰撞，……当按任一键时，此过程结束。

```
#include<graphics.h>
main()
{
 int i,graphdriver,graphmode,size;
 void *buffer;
 graphdriver=DETECT;
 initgraph(&graphdriver,&graphmode,"");
 setbkcolor(BLUE);
 cleardevice();
 setcolor(YELLOW);
 setlinestyle(0,0,1);                    /*用细实线*/
 setfillstyle(1,5);                      /*用洋红实填充*/
 circle(100,200,30);
 floodfill(100,200,YELLOW);              /*填充圆*/
 size=imagesize(69,169,131,231);         /*指定图象占字节数*/
 buffer=malloc(size);                    /*为其分配存储区*/
 getimage(69,169,131,231,buffer);        /*保存圆球图*/
 putimage(500,169,buffer,COPY_PUT);      /*在另一区域重新显示*/
 do
  {
   for (i=0;i<185;i++)
    {
     putimage(70+i,170,buffer,COPY_PUT);      /*左边球向右运动*/
     putimage(500-i,170,buffer,COPY_PUT);     /*右边球向左运动*/
    }                                         /*两球相撞后循环结束*/
   for(i=0;i<185;i++)
    {
     putimage(255-i,170,buffer,COPY_PUT);     /*左边球向左运动*/
     putimage(315+i,170,buffer,COPY_PUT);     /*右边球向右运动*/
    }
  }while(! kbhit());                          /*当不按键时重复上述过程*/
 getch();
 closegraph();
}
```

11.4.4 直接对图象动态存储器进行操作

利用显示适配器上控制图象显示的各种寄存器和图象存储器 VRAM，对其进行直接操作和控制，从而可以高效快速的实现动画效果，这可以用汇编语言，直接进行 BIOS 调用（适配器上的 EGA/VGA ROM BIOS 一视频基本输入输出系统调用）来实现，但这涉及到许多硬件结构，有一定难度，这儿就不作介绍了。

11.5 一个动画例子

该程序将产生一个在繁星背景下的一个由经纬线组成的蓝色地球，并围绕有一红色光环，它们看起来都在转动，然后一蓝色宇宙飞船从左到右缓缓飞过，周而复始，给人一种遨游太空的神秘感，屏的下方写出了 AROUND THE WORLD 字样。

该程序在图形系统初始化前，用 registebgidriver(EGAVGA_driver)进行了登记，表示已将 EGAVGA，BGI 驱动程序连接到 graphics. lib 中了，即用户程序中已有了相应的显示器驱动程序了。

程序首先用 outtextxy 函数在屏下方显示了 AROUND THE WORLD 字样，然后调用 draw-image(x,y)函数画出尾部带有三个天线的飞船，用 imagsize()函数求出了该图形所占字节数，然后用 pt_addr 指针指向存放该图形的缓冲区，并将飞船图形存在该缓冲区，接着调用 putstar()函数画星，该函数用了初始化随机数发生器函数 srand()，和随机数发生器 random(r)，srand 使得 random()每次重新产生新起点的随机数，该随机数为 0～r－1，这样就将在画面上随机地产生由小圆点和象素点构成的夜空小星星画面。

在 while 循环中，当不按键时，就反复产生一个红色光环，接着又是黑色光环，这实际上使得产生的红色光环时隐时现，因面给人以动的感觉，接着的 for 循环则用来产生地球的经纬线，它们实际上是由不同长短半径的椭圆组成，给人以立体感。为了造成动的感觉，而使经线当 i 等于偶数时，为浅蓝色，当 i 为奇数时，为黑色，给人在视觉上好像经线在动一样，这样看起来就像地球自西向东在转动一样。用 ellipse 画出的纬线则不动。接着下面用 putimage ()将飞船图象以每次 x 方向增加 6 复现在屏幕上，看起来它自西向东在遨游一样，当 x 达到最大边界 maxx 时，便重新又从 x＝r 处开始。这 while 循环中的第一个 putimage()将飞船画面与原来的画面进行异或操作，从而实现原画面的恢复工作，第二个 putimage()将在新位置(x 增 6)让飞船出现，下一轮循环时，将由第一个 putimage 将其盖掉并恢复原屏幕图象，这样就实现了飞船的飞行。

while 循环结束后(当按任意键)，则由 free()函数释放所占缓冲区空间，并回到原先的显示器工作状态。

该程序由于用 registerbgidriver(EGAVGA_driver)进行说明，表示图形显示驱动程序已经连在程序中了，又用 registerbgifont(triplex_font)函数进行了说明，表示 TRIPLEX_FONT 字形库已经连在程序中了，所以生成的可执行程序 EXE 可脱离 Turbo C 集成环境，在任何 DOS 环境下可独立运行(如同用汇编程序生成一样)。注意在 registerbgifont()函数中的字形库参数用小写。

在该程序中，有几个与作图有关的函数未介绍过，如 getmaxx()和 getmaxy()，它们将

得到当前图形显示模式下，允许的 x 和 y 的最大值，而 getmaxcolor()则将取得当前模式下允许的最大颜色数。有关函数的详细介绍，可参阅 Turbo C 2.0 参考手册。

图 11.6 便是该程序某一时刻产生的图，用打印机复制下来的。

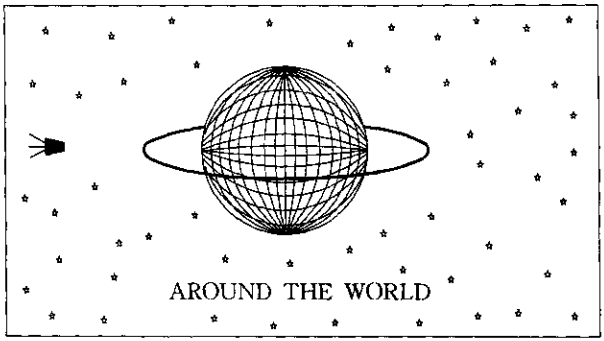


图 11.6 人造卫星环绕地球运行图

```
#include<graphics.h>
#include<stdlib.h>
#include<stdio.h>
#include<conio.h>
#define IMAGE_SIZE 10
void draw_image(int x,int y);
void Putstar(void);

main()
  {
    int graphdriver=DETECT;
    int graphmode,color;
    void *pt_addr;
    int x,y,maxx,maxy,midy,midx,i;
    unsigned int size;
    registerbgidriver(EGAVGA_driver);
    registerbgifont(TRIPLEX_FONT);
    initgraph(&graphdriver,&graphmode,"");
    maxx=getmaxx();          /*取允许的最大x值*/
    maxy=getmaxy();          /*取允许的最大y值*/
    midx=maxx/2;
    x=0;
    midy=y=maxy/2;
    setcolor(YELLOW);
```

```
settextstyle(TRIPLEX_FONT,HORIZ_DIR,4);
settextjustify(CENTER_TEXT,CENTER_TEXT);
outtextxy(midx,400,"AROUND THE WORLD");
setbkcolor(BLACK);
setcolor(RED);
setlinestyle(SOLID_LINE,0,THICK_WIDTH);
ellipse(midx,midy,130,50,160,30);
setlinestyle(SOLID_LINE,0,NORM_WIDTH);
draw_image(x,y);                    /*画飞船*/
size=imagesize(x,y-IMAGE_SIZE,x+(4*IMAGE_SIZE),y+IMAGE_
                SIZE);
pt_addr=malloc(size);
getimage(x,y-IMAGE_SIZE,x+(4*IMAGE_SIZE),y+IMAGE_SIZE,pt_
        addr);
Putstar();                          /*画星*/
setcolor(WHITE);
setlinestyle(SOLID_LINE,0,NORM_WIDTH);
rectangle(0,0,maxx,maxy);           /*画方框*/
while(! kbhit())
  { Putstar();
    setcolor(RED);
    setlinestyle(SOLID_LINE,0,THICK_WIDTH);
    ellipse(midx,midy,130,50,160,30);
    setcolor(BLACK);                /*画一个围绕地球的光环*/
    ellipse(midx,midy,130,50,160,30);
    for (i=0;i<=13;i++)
          {
            setcolor(i%2==0? LIGHTBLUE:BLACK);
            ellipse(midx,midy,0,360,100,100-8*i);  /*画地球*/
            setcolor(LIGHTBLUE);
            ellipse(midx,midy,0,360,100-8*i,100);
          }

          putimage(x,y-IMAGE_SIZE,pt_addr,XOR_PUT);  /*恢复原画面
    */
          x=x>=maxx? 0:x+6;
          putimage(x,y-IMAGE_SIZE,pt_addr,XOR_PUT);  /*在另一位置
                                                      显示飞船
                                                      */
  }
  free(pt_addr);
  closegraph();                                  /*释放缓冲区*/
  return;
}
```

```
void draw_image(int x,int y)                    /*画飞船*/
 {
  int arw[11];
  arw[0]=x+10;arw[1]=y-10;arw[2]=x+34;arw[3]=y-6;
  arw[4]=x+34;arw[5]=y+6;arw[6]=x+10;arw[7]=y+10;
  arw[9]=x+10;arw[10]=y-10;
  moveto(x+10,y-4);
  setcolor(14);
  setfillstyle(1,4);
  linerel(-3*10,-2*8);                          /*画尾部天线*/
  moveto(x+10,y+4);
  linerel(-3*10,+2*8);
  moveto(x+10,y);
  linerel(-3*10,0);
  setcolor(3);
  setfillstyle(1,LIGHTBLUE);
  fillpoly(4,arw);                              /*画飞船本体*/
 }
 void Putstar(void)                             /*画星*/
  {
   int seed=1858;
   int i,dotx,doty,h,w,color,maxcolor;
   maxcolor=getmaxcolor();                      /*得到当前模式和最多颜色数*/
   w=getmaxx();
   h=getmaxy();
   srand(seed);
   for(i=0;i<250;++i)
   {
   dotx=i+random(w-1);
   doty=1+random(h-1);
   color=random(maxcolor);
   setcolor(color);
   putpixel(dotx,doty,color);                   /*用点表示小星*/
   circle(dotx+1,doty+1,1);                     /*用圆表示大星*/
   }
 srand(seed);
}
```

第 12 章

文本的屏幕输出

显示器显示方式有文本方式和图形方式两种，第 10 章已讲了图形方式，已建立起了有关显示器类型和作图的概念及方法，本章讲述文本方式下如何控制屏幕的输出，介绍一些有关屏幕处理的函数，在 Turbo C 集成环境下，我们进行的源程序编辑、编译、连接，均是在文本方式下，它们都是以文本方式进行工作，即每屏 80 列 25 行蓝底白字，输出为整个屏幕，但 Turbo C 提供了一些屏幕处理函数，它们可以改变屏幕输出的方式，可以开设窗口，使文本输出在该窗口中进行，这些函数的有关信息（如宏定义等）均包含在 conio.h 头文件中，因此在用户程序中使用这些函数时，必须用 include 将 conio.h 包含进程序中。

12.1 文本方式的控制

12.1.1 文本方式控制函数

故名思义，文本方式就是显示文本的，它的显示单位是字符而不是图形方式下的象素，因而在屏幕上显示字符的位置坐标就用行和列进行表示。如缺省方式下，每屏为 80 列 25 行，Turbo C 规定屏的左上角为 1 行 1 列，屏的右下角为 25 行 80 列，如图 12.1 所示。Turbo C 支持的文本显示方式有 5 种，它们可以用文本显示方式设置函数来进行设置，该函数原型为

```
void textmode(int newmode)
```

该函数中的 newmode 参数可以选用如表 12.1 所示的任一种方式，可以用表中指出的方式代码，也可以用大写的方式名，该函数将清除屏幕，以整个屏幕为当前窗口，并移光标到屏的左上角。

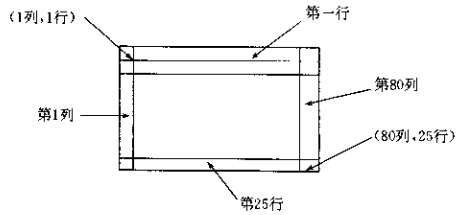


图 12.1 屏幕文本显示坐标

表 12.1 文本显示方式

方式	方式名	显示列 * 行数和颜色
0	BW40	40×25 黑白显示
1	C40	40×25 彩色显示
2	BW80	80×25 黑白显示
3	C80	80×25 彩色显示
7	MONO	80×25 单色显示
−1	LASTMODE	上一次的显示方式

LASTMODE 方式指上一次设置的文本显示方式，它常用于在图形方式到文本方式的切换，关于颜色，将在文本颜色设置函数中介绍。MONO 方式用在 MGA 显示器上。

上述的文本显示方式仅是 CGA 显示器特有的，它并不支持 EGA. VGA 显示器的 80 列 * 43 行文本方式，更不支持 TVGA 的 132 列 * 60 行的文本方式，所以文本显示方式时，EGA，VGA，TVGA 均工作在 CGA 仿真方式下。

12.1.2 文本方式颜色控颜函数

为了控制文本显示的前景和背景颜色，Turbo C 提供控制颜色的函数

① 文本颜色设置函数

void textcolor (int color);

该函数将设置显示的前景色，即字符显示的颜色，其颜色 color 可用表 12.2 中所列的值或大写的颜色名表示，该函数只能用在能进行彩色显示的方式下。

表 12.2 颜色表

颜色名	值	显示色	用处
BLACK	0	黑	前景、背景色
BLUE	1	蓝	前景、背景色
GREEN	2	绿	前景、背景色
CYAN	3	青	前景、背景色
RED	4	红	前景、背景色
MAGENTA	5	洋红	前景、背景色
BROWN	6	棕	前景、背景色
LIGHTGRAY	7	淡灰	用于前景色
DARKGRAY	8	深灰	用于前景色
LIGHTBLUE	9	淡蓝	用于前景色
LIGHTGREEN	10	淡绿	用于前景色
LIGHTCYAN	11	淡青	用于前景色

续表

颜色名	值	显示色	用处
LIGHTRED	12	淡红	用于前景色
LIGHTMAGENTA	13	淡洋红	用于前景色
YELLOW	14	黄	用于前景色
WHITE	15	白	用于前景色
BLINK	128	闪烁	用于前景色

② 文本背景颜色设置函数

void textbackground (int color);

该函数将设置文本显示的背景颜色，其参数 color 仅能选择表 12.2 中的前 8 种颜色，即值为 0～7。

③ 文本属性设置函数

void textattr (int attr);

该函数将设置文本显示的属性，所谓显示的属性指字符显示的颜色，背景色，字将显示是否闪烁，显示属性参数 attr 可用一个字节即 8 位来描述，各位含义如图 12.2 所示：

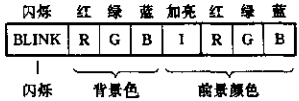


图 12.2 属性字节各位含义

其中低四位用来设置字符显示颜色(对应颜色值 0～15)，4～6 位用来设置显示背景色(颜色值为 0～7)，第 7 位最高位用来设置显示的字将是否闪烁，示例如下：

要求在蓝色背景下显示红色的字符，可用此函数设置：

textattr(RED+(BLUE<<4));

要求在白色背景下显示闪烁的蓝色字符，可设置为：

textattr((WHITE<<4)+BLUE+BLIKK);

或者：

textattr(128+1+(15<<4));

其中 WHITE<<4 表示左移四位，即其对应的二进制值左移四位，变成 4～6 位，这恰是设置背景色的位，也就是用 15 的二进制表示的数左移四位。用十六进制表示可写成 textattr (0xf1)

12.1.3 字符显示亮度控制函数

void highvideo(void);

该函数将设置用高亮度显示字符。

void lowvideo (void);

该函数将设置用低亮度显示字符。

void normvideo (void);

该函数将设置通常亮度显示字符。

例：该程序首先定义了一个字符指针数组 S()指向可用背景和前景显示的七种颜色，然后用 for 循环连续开辟七个窗口，每个窗口背景色同指针 s[i]指向的颜色一样，每个窗口上显示大写的颜色名(由 cputs(s[i]))，其颜色为窗口颜色的下一顺序颜色，偶数窗口颜色名用高亮度显示，奇数窗口用低亮度显示。如图 12.3 所示，该程序中用了设置窗口的函数 Window()和窗口输出函数 cputs()，这些函数将在下一节讲述，可参阅这些函数使用的说明。

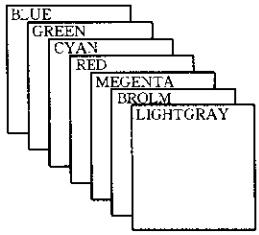


图 12.3 用高低亮度字符标示偶奇序号的窗口

```
#include <conio.h>
main()
 {
   int i;
   char *s[ ]={"BLACK","BLUE","GREEN","CYAN","RED",
               "MAGENTA","BROWN","LIGHTGRAY"};
   textmode(C80);                /*设置显示文本方式为C80*/
   texthackground(0);            /*设置背景色*/
   clrscr();                     /*清屏*/
   for (i=1;i<8;i++)
     {
       window(10+i*5,5+i,30+i*5,15+i);/*开窗口*/
       textbackground(i);
       clrscr();
       textcolor(7+i);           /*设置窗口内字符显示颜色*/
       if(i%2==0)
       highvideo();              /*i为偶数的窗口字符高亮度显示*/
       else
       lowvideo();               /*奇数窗口字符低亮度显示*/
       cputs(s[i]);              /*输出字符到窗口内*/
     }
   getch();
}
```

12.2 窗口设置和文本输出函数

在文本方式下，没进行窗口设置时，即窗口设置缺省时，认为整个屏幕为显示窗口，但

Turbo C 也提供了窗口设置函数 Window(),可由用户根据自己的需要来重新设定显示窗口,当设定后,以后的控制台 I/O 操作(即文本输入输出),就可均在此窗口中进行,现对文本方式下的窗口设置函数和控制台输出函数(即文本输出函数)作以下介绍:

12.2.1 窗口设置函数

```
void window(int x1,int y1,int x2, int y2);
```

其中(x1,y1)为窗口的左上角坐标,(x2,y2)为窗口的右下角坐标,这些坐标是以整个屏幕为参考系,如图 12.1 所示,当定义窗口时,若坐标超出屏幕坐标界限,则该窗口将不会建立。

利用窗口函数可以在屏幕上定义多个不同窗口,以显示不同的信息,但最后一次定义的窗口为当前窗口,与窗口有关的操作函数,均在此窗口内进行,如定义了一个窗口后,前面讲过的函数 textcolor, textbackground, textattr 将仅对此窗口起作用,窗口外不会受到影响。

12.2.2 控制台文本输出函数

我们以前介绍过的 printf(),putc(),puts(),putchar()和输出函数以整个屏幕为窗口的,它们不受由 Window 设置的窗口限制,也无法用函数控制它们输出的位置,但 Turbo C 提供了三个文本输出函数,它们受窗口的控制,窗口内显示光标的位置,就是它开始输出的位置。当输出行右边超过窗口右边界时,自动移到窗口内的下一行开始输出,当输出到窗口底部边界时,窗口内的内容将自动产生上卷,直到完全输出完为止,这三个函数均受当前光标的控制,每输出一个字符光标后移一个字符位置。这三个输出函数是:

```
int cprintf(char *format,...);
int cputs(cbar *str);
int putch(int ch);
```

它们的使用格式同 printf(),puts()和 putc(),其中 cprintf()是将按格式化串定义的字符串或数据输出到定义的窗口中,其输出格式串同 printf 函数,不过它的输出受当前光标控制,且输出特点如上所述,cputs 同 puts,是在定义的窗口中输出一个字符串,而 putch()则是输出一个字符到窗口,它实际上是函数 putc 的一个宏定义,即将输出定向到屏幕。

12.3 清屏和光标操作函数

12.3.1 清屏函数

```
void clrscr(void);
```

该函数将清除窗口中的文本,并将光标移到当前窗口的左上角,即(1,1)处。

```
void clreol(void);
```

该函数将清除当前窗口中从光标位置开始到本行结尾的所有字符,但不改变光标原来的位置。

```
void delline (void);
```

该函数将删除一行字符,该行是光标所在行。

12.3.2 光标操作函数

void gotoxy(int x, int y);

该函数将把光标移到窗口内的(x,y)处,x,y 坐标是相对窗口而言。它多和 cprintf 函数配合,以指定输出开始位置。

下面的程序演示了上述函数的作用,本程序由于没有用 window 函数设置窗口,因而用缺省值,即全屏幕为一个窗口,程序开始设置 40 列×25 行文本显示方式(C40),背景色为蓝色,前景色为红色,因此屏上显示出蓝底红字的 prees any key to continue,可以看出整个屏(即窗口)并没有显示蓝色,当按任一键后,经 clrscr 函数清屏后,设置的背景色才使屏背景变蓝,说明只有用 clrscr 清屏后,其前面的设置背景色函数才起作用。gotoxy(10,10)使光标移到第 10 行 10 列,因而接着 cprintf 函数将 Welcom your 字符串从此位置开始输出,而下一个 cprintf 的输出则被定位在 14 行 10 列位置输出,接着 gotoxy(17,10)将光标移到 10 行 17 列位置,当按任一键后,则由 clreol 函数将该行从 17 列位置开始到行尾的所有字符删去,即将 your 删掉了,接着 gotoxy(17,14)又将光标移至 14 行 17 列位置,按任一键后,则显示在该行的 Let's study Turbo C 全被删除,说明 delline 函数仅受光标所在行的控制,列号不起作用,即将删除光标所在行的一行字符。

```
#include <conio.h>
 main()
{
  int i;
  textmode(C40);                    /*设置文本显示方式为C40*/
  textbackground(BLUE);             /*设置背景色为蓝*/
  textcolor(RED);                   /*设置前景色为红色*/
  cprintf("%s","Prees any key to continue.");
  getch();
  clrscr();
  gotoxy(10,10);                    /*光标移到(10,10)处*/
  cprintf("%s","Welcom Your");      /*在(10,10)处开始显示字符串*/
  gotoxy(10,14);
  cprintf("%s","Let's study Turbo C.");
  gotoxy(17,10);
  getch();
  clreol();                         /*删去从第17列开始到行尾字符*/
  gotoxy(17,14);
  getch();
  delline();                        /*删去第14行*/
  getch();
 }
```

程序运行时，首先显蓝底红字的字符串：
Prees any key to continue.
按任一键后：显示蓝屏红字的字符串：
Welcom Your

Let's study Turbo C.

12.4 程 序 例

下面的程序开始设置文本方式为C40，接着用黑底红字显示出Welcom You字样，跟着for循环开出7个窗口，第一个窗口背景为蓝色，写出绿色的"please press any key to continue"字样，由于to字的t字母已经到窗口的右边界，故o及以后的字符均从第二行输出，cprintf输出完后，闪烁的光标停在continue字符串最后，表示若以后有窗口内的输出操作，则从此位置开始。当按任一键后，又出现第二个窗口，颜色为绿色，并显示和第一个窗口内相同的字符串，且颜色为青色，即为窗口背景色加1(即i+1)，当再按任一键后，第二个窗口变青，又按键后，窗口颜色变红，如此重复，直到最后窗口变灰白为止，这是因为从第三个窗口开始，(该窗口的行列坐标是window(25,7 45,16))，窗口的右下角列坐标已超出，故而以后的window函数不起作用，所以仅是textbackground()和textcolor()函数再起作用。当for循环结束后，屏幕又被设置成C80文本方式，接着for循环又生成由蓝、绿……灰组成的七个窗口，每个窗口上而显示着闪烁的Window字样，这是因为用textattr(i+1+(i<<4)+BLINK)函数，设置了前景色值为i+1，背景色为i<<4，即前景色值的二进制数左移4位及属性字节最高位为BLINK(即128)，所以显示的字符是闪烁的，且颜色是窗口背景值加1。

```
#include <conio.h>
main()
 {
  int i;
  textmode(C40);
  textbackground(0);
  textcolor(RED);
  clrscr();
  cprintf("%s","Welcom You");
  for (i=1;i<8;i++)
   {
    window(10+i*5,1+2*i,30+i*5,10+2*i);
    textbackground(i);
    textcolor(i+1);
    clrscr();
    cprintf("%s","Please Press any key to continue");
```

```
    getch();
  }
  textmode(C80);                    /* 设置 C80 文本显示方式 */
  textbackground(0);                /* 设置背景色为黑 */
  clrscr();                         /* 清屏 */
  for (i=1;i<8;i++)                 /* 连续产生 7 个窗口 */
   {
    window(10+i*5,1+2*i,30+i*5,10+2*i);
    textattr(i+1+(i<<4)+BLINK);
    clrscr();
    cprintf("Window%d",i);
   }
  getch();
}
```

程序运行时，将出现如图 12.4 所示的屏幕形象，但最后出现的七个颜色窗口(即该屏幕形象消失后)并显示闪烁的 Window 字样的形象未画出。

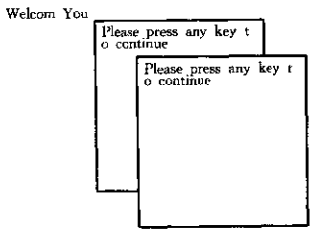


图 12.4 每按一次键后出现的窗口图形

12.5 屏幕文本移动与存取函数

12.5.1 屏幕文本移动函数

void movetext(int x1,int y1,int x2,int y2,int x3,int y3);

该函数将把屏幕上左上角为(x1,y1)，右下角为(x2,y2)的矩形内文本拷贝到左上角为(x3,y3)的一个新矩形区内。这里 x,y 坐标是以整个屏幕为窗口坐标系，即屏左上角为(1,1)。该函数与开设的窗口无关，且原来矩形区文本不变。

12.5.2 屏幕文本存取函数

① 存文本函数

void gettext(int x1, int y1, int x2, int y2, void * buffer);

该函数将把左上角为(x1,y1)，右下角为(x2,y2)的矩形区内的文本存到由指针 buffer

指向的一个内存缓冲区内，这个缓冲区大小可以这样来计算，因一个在屏幕上显示的字符需占显示存储器 VRAM 的两个字节，即第一个字节是该字符的 ASCII 码，第二个字节为属性字节，即表示其显示的前景、背景色及是否闪烁，如图 12.2 所示，所以矩形区内的文本所占字节总数可以这样来计算：

字节总数＝矩形内行数 * 每行列数 * 2

其中：

矩形内行数＝y_2-y_{1+1}

每行列数＝x_2-x_{1+1}

每行列数是指矩形内每行的列数。

矩形内文本字符在缓冲区内存放的次序是从左到右，从上到下，每个字符占连续两个字节并依次存放。

② 取文本函数

void puttext(int x1, int y1, int x2, int y2, void * buffer);

该函数将把由 buffer 指针指向的缓冲区内所存文本复制到屏幕上一矩形区内，该矩形区左上角为(x1,y1)，右下角为(x2,y2)。

下面的程序首先定义了一个字符数组，下标为 64，表示用来存四行八列的文本，然后在(10,10)开始位置显示 L：load，接着在下面三行相同的列位置显示另外三条信息，13 行 10 列显示的 E：exit 后面带有回车换行符，为的是将光标移到下一行开始处，好显示 Press any key to continue。当按任一键后，gettext 函数将(10,10,18,13)矩形区的内容存到 ch 缓存区内 och 即上述的四行八列信息，接着设置一个窗口，并纵向写上 1,2,3,4，然后用 movetext()将此窗口内容复制到另一区域，由于此区域包括背景色和显示的字符，所以被复制到另一区域的内容也是相同的背景色和文本。当按任一键后，又出现提示信息，再按键，则存在 ch 缓冲区内的文本由 puttext()又复制到开设的窗口内了，注意上述的函数 movetext()，gettext()，puttext()均与开设的窗口内坐标无关，而是以整个屏幕为参考系的。

```
#include <conio.h>
main()
{
  int i;
  char ch[4 * 8 * 2];                    /* 定义 ch 字符串数组作为缓存区 */
  textmode(C80);
  textbackground(0);
  textcolor(RED);
  clrscr();
  gotoxy(10,10);
  cprintf("L : load");
  gotoxy(10,11);
  cprintf("S : save");
  gotoxy(10,12);
  cprintf("D : delete");
```

```
    gotoxy(10,13);
    cprintf("E : exit\r\n");
    cprintf("Press any key to continue");
    getch();
    gettext(10,10,18,13,ch);                    /* 存矩形区文存到 ch 缓存区 */
    clrscr();
    textbackground(1);
    textcolor(3);
    window(20,9,34,14);                         /* 开一个窗口 */
    clrscr();
    cprintf("1. \r\n 2. \r\n 3. \r\n 4. \r\n"); /* 纵向写 1,2,3,4 */
    movetext(20,9,34,14,40,10);/* 将矩形区文本复制到另一区域 */
    puts("Hit any key");
    getch();
    clrscr();
    cprintf("Press any key to put text");
    getch();
    clrscr();
    puttext(23,10,31,13,ch);/* 将 ch 缓存区所存文本在屏上显示 */
    getch();
}
```

12.6 状态查询函数

有时需要知道当前屏幕的显示方式，当前窗口的坐标、当前光标的位置，文本的显示属性等，Turbo C 提供了一些函数。

12.6.1 得到得幕文本显示有关信息的函数

```
void gettextinfo(struct text_info *f);
```

这里的 text_info 是在 conio.h 头文件中定义的一个结构，该结构的定义是：

```
struct text_info{
    unsigned char winleft;          /* 窗口左上角 x 坐标 */
    unsigned char wintop;           /* 窗口左上角 y 坐标 */
    unsigned char winright;         /* 窗口右下角 x 坐标 */
    unsigned char winbottom         /* 窗口左下角 y 坐标 */
    unsigned char attributes;       /* 文本属性 */
    unsigned char normattr;         /* 通常属性 */
    unsigned char currmode;         /* 当前文本方式 */
    unsigned char screenheight;     /* 屏高 */
```

```
    unsigned char screenwidth;      /*屏宽*/
    unsigned char curx;             /*当前光标的x值*/
    unsigned char cury;             /*当前光标的y值*/
    };
```

下面的程序将屏幕设置成80列彩色文本方式，并开了一个window(1,5,70,20)的窗口，在窗口中显示了current information of window，然后用gettextinfo函数得到当前窗口的信息，后面的cprintf()函数将分别显示出结构text_info各分量的数值来，即：

current information of window
Left corner of window is 1,5 Right corner of window is 70,20 Text window attribute is 29 Text window normal attribute is 29 Current video mode is 3 Window height and width is 25,80 Row cursor pos is 2,Column pos 1

程序清单如下：

```
#include <conio.h>
main()
 {
   struct text_info current;
   textmode(C80);
   textbackground(1);
   textcolor(13);
   window(1,5,70,20);
   clrscr();
   cputs("current information of window\r\n");
   gettextinfo(&current);
   cprintf("Left corner of window is %d,%d ",
           current.winleft,current.wintop);
   cprintf("Right corner of window is %d,%d ",
           current.winright,current.winbottom);
   cprintf("Text window attribute is %d ",
           current.attribute);
   cprintf("Text window normal attribute is %d ",
           current.normattr);
   cprintf("Current video mode is %d",current.currmode);
   cprintf("Window height and width is %d,%d ",
           current.screenheight,current.screenwidth);
   cprintf("Row cursor pos is %d,Column pos %d",
           current.cury,current.curx);
 }
```

12.6.2 得到当前光标位置的函数

```
int wherex(void);
int wherey(void);
```

这两个函数将分别得到当前窗口中光标的 x 和 y 坐标。

例如用下面的调用，就可以打出当前光标的 x，y 坐标：

```
printf("The cursor is at %d,%d\n",wherex(),wherey());
```

12.7 综合应用例

12.7.1 一个弹出式菜单

这个程序是一个文本方式下的菜单程序，它生成一个弹出式菜单，如图 12.5 所示。程序运行时，首先弹出一个带洋红色框的蓝底菜单，并有一红色光条压在第一项上，当按 E 键时，程序回到系统，压 DOW 键时，光条移到第二项，压 A 时，则列出当前目录下的各文件目录，再压任意键后则又弹出菜单，当压 B 键时，则列出目录，当屏满后，暂停，按键后，又继续列出，它实际上就是执行 dir/p 命令，当按回车键后，又出现菜单，当按 C 键后，便执行 dir/w dos 命令，以宽行格式列出目录，按回车键后又回到菜单用 UP 键可使光条上移。

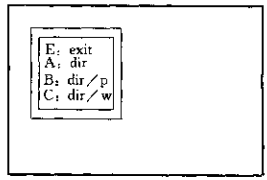


图 12.5 弹出式菜单

程序开始时，首先用 do 循环反复运行下面的程序，直到按 E 键后为止，window(7,8,19,15)定义了一个用洋红色填充的窗口，接着又定义一个蓝色窗口，它套在洋红色窗口内，因而形成一个带洋红色粗边框的蓝色窗口，在窗口内写上各菜单项，接着又调用光条上移函数 upbar(y－1)，使红色光条压在第一菜单项上，下面的 do 循环则用来构成一个反复检查按键，并转去执行相应的功能操作，do 循环内的第一个 switch (ky)语句用来判断按的是何键，按键值赋给 y 的相应值，并使 ky=key－enter，以结束 do 循环，跟着第二个 switch(y)语句则按 y 值转去执行相应的菜单项功能。只有当菜单项功能键是 E 时，才使得外层的 do 循环结束面回到系统。

int key()是读键函数，它将返回按键的扫描码在 ah 中。upbar(int y)函数是产生一个上移光条(用 gettext()函数)，实际上它第一次被调用时，则用 gettext(i,y,i,y,&t)连续 8 次将菜单项 8 个字符长的区域(蓝底无字)存入 &t 中，用蓝底白字又放回(用 puttext()函数)，而第 2 个 gettext()函数则将下移一行的 8 个字符长区域(即第一个菜单项，存入 &t 中，又以白字红底放回原处，即产生一个红色光条压在第一菜单项上。以后的调用(当按 UP 键时)，则随 y 值不同，而红色光条压在不同菜单项上。

downbar()函数产生下移光条，当按 Down 键时，便调用该函数将红底白字的光条压在下一条菜单项上。

```
#include <process.h>
#include <stdlib.h>
#include <stdio.h>
```

```
#include <dos.h>
#include <conio.h>
/*define Keys scan code */                    /*定义各键的扫描码*/
#define Key_DOWN  80
#define Key_UP    72
#define Key_A     30
#define Key_B     48
#define Key_C     46
#define Key_E     18
#define Key_ENTER 28

int key();

main()
 {
  int ky,y;
  char ch;
  textbackground(0);
  clrscr();
  do
    {
     textmode(C80);
     textbackground(13);
     textcolor(RED);
     window(7,8,19,15);                    /*开一个窗口*/
     clrscr();
     textbackground(1);
     textcolor(RED);
     window(8,9,18,14);                    /*再开一个当前窗口,套在上一窗口之中*/
     clrscr();
     gotoxy(3,3);cprintf("E:exit\r\n");     /*窗口中写上红色的菜单项*/
     gotoxy(3,4);cprintf("A:dir\r\n");
     gotoxy(3,5);cprintf("B:dir/p\r\n");
     gotoxy(3,6);cprintf("C:dir/w\r\n");
     y=10;
     upbar(y-1);                           /*调用光条上移函数*/
     do
      {
       ky=key();                           /*得到按键的扫描码*/
      switch(ky)
```

```
{
  case Key_A:/*A and a key*/
     {y=12,ky=Key_ENTER;};
      break;
  case Key_B:/*B and b key*/
     {y=13,ky=Key_ENTER;};
      break;
  case Key_C:/*C and c key*/
     {y=14,ky=Key_ENTER;};
      break;
  case Key_E:/*E and e key*/
     {y=11,ky=Key_ENTER;};
      break;
  case Key_DOWN:/*Cursor down key*/
     if(y<13){upbar(y);y++;};
      break;
  case KB_S_N_UP:/*Cursor up key*/
    if(y>10){downbar(y);y--;};
      break;
 }
  }while(ky! =Key_ENTER);      /* Enter key */
textcolor(WHITE);              /*设置显示文本用白色*/
   switch(y)
    {
     case 11:
       ch='%';                 /*返回系统*/
       break;
     case 12:
       {system("dir");getch();};   /*列目录后等待按键*/
       break;
     case 13:
       {system("dir/p");getch();};/*列目录屏满后暂停,按任意键继续*/
       break;
     case 14:
       {system("dir/w");getch();};/*用宽行格式列目录*/
       break;};
    if (ch=='%') break;
     }while(1);
    clrscr();                  /*清屏*/
```

```
        }
    /* read char on key,return 16 bit scan code */
int key()                                   /*读键函数*/
   { union REGS rg;
     rg.h.ah=0;
     int86(0x16,&rg,&rg);
     return rg.h.ah;
   }
    /*red bar down */
upbar(int y)                                /*光条上移函数*/
   {int i;
    typedef struct texel_struct{unsigned char ch;
                unsigned char attr;}texel;
    texel t;
      for(i=9;i<=17;i++)
       {gettext(i,y,i,y,&t);
        t.attr=0x1f;                        /*字符为白色,背景蓝色*/
        puttext(i,y,i,y,&t);
        gettext(i,y+1,i,y+1,&t);
        t.attr=0x4f;                        /*字符为白色,背景为红色*/
        puttext(i,y+1,i,y+1,&t);};
        gotoxy(3,y+1);
    return;
   }
    /* red har up */
  downbar(int y)                            /*光条下移函数*/
    {int i;
     typedef struct texel_struct{unsigned char ch;
                unsigned char attr;}texel;
     texel t;
     for(i=9;i<=17;i++)
       {gettext(i,y,i,y,&t);
        t.attr=0x1f;                        /*字为白色,背景为蓝色*/
        puttext(i,y,i,y,&t);
        gettext(i,y-1,i,y-1,&t);
        t.attr=0x4f;                        /*字符为白色,背景为红色*/
        puttext(i,y-1,i,y-1,&t);};
        gotoxy(3,y-1);
    return;
   }
```

12.7.2 一个下拉式菜单

该程序在文本方式下产生一个下拉式菜单，程序运行时首先在屏幕顶行产生一个白底黑字的主菜单，各菜单项的第一个字母加红，表示为热键，如图 12.6 所示，当选择主菜单第一项，即按 ALT_F 时，便产生一个下拉式子菜单，可用 UP 和 DOWN 键使压在第一个子菜单项上的黑色光条上下移动，当压在某子菜单项上，且按回车后，程序便转去执行相应子菜单项的内容，由于篇幅关系，该程序仅是一个演示程序，只作了第一个主菜单项和对应的子菜单，且子菜单项对应的操作只在程序相应处作了说明，并无具体内容，对主菜单项其它各项未作选它时相应的子菜单，但作法和第一项 File 的相同，故不赘述。

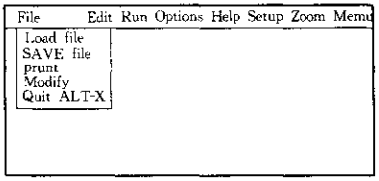


图 12.6 下拉式菜单

程序中用指针数组 munu[ ]存放主菜单各项，red[ ]存放各项的热键字符(即主菜单项各项的第一个字母)，f[ ]存放主菜单第一项 file 的子菜单各项。定义字符数组 buf 存放原子菜单所占区域的内容，buf1 存放一个子菜单项区域内容，由于一个字符占两个字节，故所占列数均乘了 2。第一个 for 循环在开辟的窗口内(window(1,1,80,2))，用白底黑字依次显示出主菜单各项，并用测菜单项字符个数的办法，使光标回移到该项的第一个字母处，然后用红色字母再重现它，当进行 while(1)循环时，等待按键，用键盘管理函数 bioskey()，当有键按下时，则由 get_key()函数取回按键的扫描码，然后进行判断，当按 ALT_X 键时，则退出，若按 ALT_F 键时，则执行弹出子菜单的操作，首先要将子菜单的区域内容保存到 buf 缓冲区内(用 gettext(4,2,19,12,buf))，当子菜单项消失时，用它来恢复原区域的内容。然后设置一个作子菜单的白色窗口(window(4,2,19,9))，调用作框函数 box，在白色底上画出一个矩形框来，接着的 for 循环则在框内显示出子菜单各项内容，并等待按键，当是 ALT、X、ENTER 或 ESC 各键时，则转去作相应的处理，不是这些键是 UP 或 DOWN 键时，则产生黑色光条的上下移动，当光条在第一项上时，若再按 UP 键，则光条移到最后一项，若光条原来就在最后一项，再按 DOWN 键，则光条退回到第一子菜单项去，这由 y=y==2? 6:y-1 和 y=y==6? 2:y+1 来实现，当光条压在某子菜单项上，且按回车后，程序则转去执行相应的子菜单项指明的操作，它们由 switch(y-1)语句来实现，当光条压在第一子菜单项上，且按回车后，则执行 cas 1 后的操作，由于是示范程序，具体操作没有指出，要变为实用菜单，则需在此处填上操作内容，当按 ESC 键时，则恢复原子菜单所占的区域内容，并回到主菜单下。

用画框子程序 box()画框，实际上是由符号┌(ASCII 为 0xdε)，─，┐，└，|，┘拼成了一个矩形框，这是和图形方式下画框不同之处，图形方式下画框调用库函数 ractangle()

即可。

```
#include <process.h>
#include <stdlib.h>
#include <stdio.h>
#include <dos.h>
#include <conio.h>

#define Key_DOWN     80
#define Key_UP       72
#define Key_ESC      1
#define Key_ALT_F    33
#define Key_ALT_X    45
#define Key_ENTER    28

int get_key();
void box(int startx,int starty,int high,int widtb);

main()
{
  int i,key,x,y,l;
  char *menu[]={"File","Edit","Run","Options","Help", /*主菜单各项*/
                "Setup","Zoom","Menu"};
  char *red[]={"F","E","R","O","H","S","Z","M"};    /*加上红色热键*/
  char *f[]={"Load     file",                        /*File项的子菜单*/
             "Save     file",
             "Print        ",
             "Modify       ",
             "Quit   Alt_X",};
  char buf[16*10*2],buf1[16*2];        /*定义保存文本的缓冲区*/
  textbackground(1);
  clrscr();
/*   textmode(C80);*/
  window(1,1,80,1);
  textbackground(15);
  textcolor(0);
  clrscr();
  window(1,1,80,2);                    /*定义显示菜单的窗口*/
  for(i=0,l=0;i<8;i++)
   { x=wherex();                       /*得到当前光标的坐标*/
```

```
    y=wherey();
    cprintf("    %s",menu[i]);          /*显示各菜单项*/
    l=strlen(menu[i]);                  /*得到菜单项的长度*/
    gotoxy(x,y);
    textcolor(RED);
    cprintf("    %s",red[i]);           /*在主菜单项各头字符写上红字符*/
    x=x+l+4;
    gotoxy(x,y);
    textcolor(BLACK);}                  /*为显示下一个菜单项移动光标*/
while(1)
  {
  key=0;
  while(bioskey(1)==0);                 /*等待按键*/
  key=get_key();                        /*取键扫描码*/
  if(key==Key_ALT_X)exit(0);            /*ALT-X则退出*/
  if(key==Key_ALT_F)                    /*若ALT_F,则弹出子菜单*/
    {
    textbackground(0);
    textcolor(15);
    gotoxy(4,1);
    cprintf("%s",menu[0]);              /*加黑File项*/
    gettext(4,2,19,12,buf);             /*保存窗口原来的文本*/
    window(4,2,19,8);
    textbackground(15);
    textcolor(0);
    clrscr();
    window(4,2,19,9);                   /*设置作矩形框的窗口*/
    box(1,1,7,16);                      /*调用作框函数*/
    for(i=2;i<7;i++)                    /*显示子菜单各项*/
      {
      gotoxy(2,i);
      cprintf("%s",f[i-2]);
      }
    gettext(2,2,18,3,buf1);
    textbackground(0);
    textcolor(15);
    gotoxy(2,2);
    cprintf("%s",f[0]);
    y=2;
```

```
key=get_key();                                    /*等待按键*/
while(key!=Key_ALT_X&&key!=Key_ENTER&&key!=Key_ESC)
                                                  /*是ALT_X,回车和Esc键时退出*/
 {
  if(key==Key_UP || key==Key_DOWN)/*是UP或Down键时*/
  {
   puttext(2,y,18,y+1,buf1);                      /*恢复原先的项*/
   if(key==Key_UP)y=y==2? 6:y-1;
   if(key==Key_DOWN)y=y==6? 2:y+1;
   gettext(2,y,18,y+1,buf1);                      /*保存要压上光条的子菜单项*/
   textbackground(0);
   textcolor(15);
   gotoxy(2,y);
   cprintf("%s",f[y-2]);                          /*产生黑条压在所选项上*/
  }
  key=get_key();                                  /*等待按键*/
  }
if(key==Key_ALT_X)exit(0);
if(key==Key_ENTER)                    /*若是回车键,判断是哪一子菜单按的回车*/
 {
  switch(y-1)
   {
  case 1:                                         /*是子菜单项第一项*/
   /*Load file */
       break;
  case 2:
   /*save file*/
      break;
  case 3:
   /*print */
      break;
  case 4:
   /*modify */
      break;
  case 5:
      exit(0);
  default:
      break;
   }
 }
```

```
    else                                          /*是 Esc 键,返回主菜单*/
      {
      window(1,1,80,2);
      puttext(4,2,19,12,buf);
      textbackground(15);
      textcolor(0);
      gotoxy(4,1);
      cprintf("%s",menu[0]);
      }
     }
   }
}
int get_key()                                     /*取键扫描码函数*/
  { union REGS rg;
   rg.h.ah=0;
   int86(0x16,&rg,&rg);
   return rg.h.ah;
   }
void box(int startx,int starty,int high,int width)   /*画矩形框函数*/
   {  int i;
   gotoxy(startx,starty);
   putch(0xda);                                   /*画┌*/
   for(i=startx+1;i<width;i++)putch(0xc4);        /*画─*/
   putch(0xbf);                                   /*画┐*/
   for(i=starty+1;i<high;i++)
     {
      gotoxy(startx,i);putch(0xb3);               /*画|*/
      gotoxy(width,i);putch(0xb3);
     }
   gotoxy(startx+1,width);
   putch(0xc0);                                   /*画└*/
   for(i=startx+1;i<width;i++) putch(0xc4);
   putch(0xd9);                                   /*画┘*/
   return;
   }
```

只能在文本方式下显示光标,当显示为标准文本方式时(80 列 25 行英文字符),可有 16 色显示),可有 8 个显示页,第 0 页 VRAM 地址为 B800:0000 开始,共占 4096 个字节地址,实际用了 4000 个字节地址。

字符属性指要显示一个字符时,不仅要有该字符的 ASCII 码,还要指出显示字符的前景色和背景色,它占用一个字节,该字节各位的含义如图 12.2 所示。

第 13 章

屏幕图形的存取

在图形方式下，将屏幕显示的图形用文件形式存在磁盘上，需要时将该图形文件重新显示在屏幕上，这种技术在开发 C 语言实用程序时，非常重要，然而 Turbo C 没有提供这方面的库函数，由于这种技术和图形适配器硬件结构息息相关，因而要编这类程序，必须对与屏幕存取有关的硬件有个大致了解，在第十章开始已经介绍了一些与作图有关的一些概念和知识，但为了能对屏幕图形存取，还需更深入地了解一些图形适配器的结构，它以接口卡的形式插在 PC 机的扩展槽中。现简单加以介绍。

13.1 屏幕图形与 VRAM 地址的关系

当 PC 机进行显示时，要显示的字符代码和图象信息均以二进制形式存储在视频存储器中，简称 VRAM，它们在图形适配器上。VRAM 中的信息再由适配器上的其它部件，将其变成串行的模拟信息流，以控制显示器将其显示出来。VRAM 是 PC 微机系统存储器(或称内存)中的一部分，它占用系统的部分地址区间，即是同系统存储器统一编址的，同受 CPU 控制。

对 CGA 显示器，VRAM 仅有 16K 字节，在中分辨显示方式下，每屏可有 320×200 象素，VRAM 中每个字节地址存放 4 个象素点，故每行相当占 80 个字节，即一个象素点可用 2 位表示，2 位二进制代码可有 4 种组合，因而若用这种组合代表颜色，可有四种，这就是为什么在 CGA 方式下，中分辨显示时，仅能显示四种颜色的原因，这些象素在 VRAM 中存放的特点是：屏幕上显示的象素点从左到右，从上到下依次存在 VRAM 从 0 号地址开始的连续地址中，但对 CGA 方式，还有一个例外，即屏幕上显示的偶数行象素存在 16K VRAM 的上半部(即 0～7K 地址中)，而奇数行则存在 VRAM 的下半部(即 8K～16K 地址中)即以地址 0x1000 即 8192 开始，由于 VRAM 地址和系统内存地址统一编址，从统一编址的角度来看，VRAM 段地址为 0xB800，则 VRAM 的 0 地址就相当于 0xB800:0009，而 VRAM 下半部开始地址 8K 就相当于 0xB800:0x1000，如图 13.1 所示。所以在 CGA 方式下，存中分辨图象时，只要将图象显示的偶数行对应的 VRAM 地址中内容顺序存到磁盘文件中，接着将 VRAM 下半部对应奇数行的地址中的内容再存到磁盘中。这样交替把 VRAM 中的信息存到磁盘文件中，当存够 16000 个地址时，便将一屏图象存完(320×200＝64000 点，然后除 4 得 16000 个字节，即占 16000 个字节地址)。

对 CGA 高分辨方式(640×200)，VRAM 中每个字节可表示 8 个象素，因二进制 1 位仅能有 0,1 组合，故可表示两色，这样存够 128000 个点，即 16000 个字节，就可把一屏图象存完，图 13.2 表示了 VRAM 与屏幕图象关系。

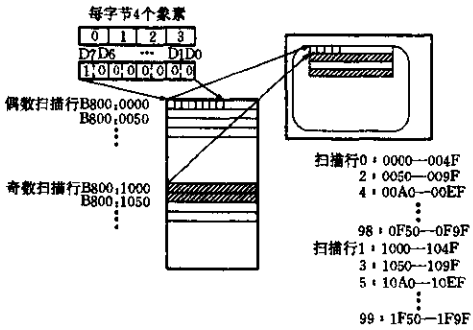


图 13.1 CGACX 方式(x=0,1,2,3)

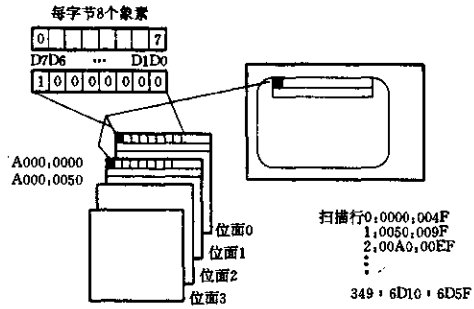


图 13.2 CGAHI 方式

13.2 存取屏幕图象时地址指针的设置

由于 turbo c 中远程指针是一个 32 位的地址指针，它前面 16 位存的是段地址，后面 16 位存段的偏移地址，故存取对应屏幕图象 VRAM 中的信息时，其开始地址指针可这样设置：

```
char far *per=(char far *)0xB8000000;
```

若已定义 fp 为存图象的文件指针，即

```
FILE *fp;
```

则存一屏图象，用如下 for 循环即可：

```
for (i=0;i<8192;i++)
  { putc(*ptr,fp);          /*存偶字节*/
    putc(*(ptr+8192),fp);/*存奇字节*/
```

```
    ptr++;
  }
```

当要将存在文件中的满屏图象信息显示到屏幕上时，只要将文件中的二进制信息依序再存入 VRAM 中即可，存的过程，就是显示的过程。

```
char far *per=(char far *)0xB8000000L;
    ⋮
for (i=0;i<8192;i++)
{ *per=get(fp);              /* 加载偶字节 */
  *(per+8192)=get(fp)        /* 加载奇字节 */
 per++
}
```

13.3 VRAM 的位面结构和对它的读写操作

13.3.1 VRAM 的位面结构

对 EGA，VGA 适配器，其结构大不同于 CGA，它的 VRAM 采用位面结构，VRAM 可达 256K 字节，它分成独立的四个 64K 字节部分，每个 64K 字节部分称为一个位面，这四个位面占有同一系统存储区间，即用同一的 64K 个地址，也就是说不同位面相应位置字节采用相同地址，这个问题可以这样来理解，即将 EGA，VGA 的 VRAM 想像成一个三维结构，一个位面压在另一位面上，互相重叠，如图 13.3 所示，这些位面依次称为 0，1，2，3，位面，因此对每个地址，实际上代表四个位面上相并列的 4 个字节，由于习惯上认为一个地址仅能代表一个字节，故这个代表字节的每一位实际上就代表了四个位面上的同一位置的位，由于 4 位可有 16 种 0，1 的组合，它可代表 16 种颜色，因此，若用 VRAM 地址代表字节，则该字节地址和字节中所处的位，就代表了象素在屏上所处的位置，而该位又代表了四个位面上同一位的值，这个值又决定了该象素的颜色，因此又可称位面为颜色位面，图 13.3 和图 13.4 表示 VRAM 的位面结构和 VGAMED 和 VGAHI 显示方式下 VRAM 和屏上象素的对映图。

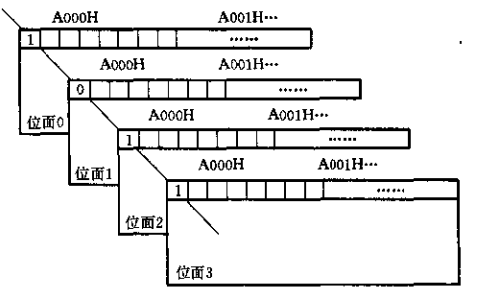


图 13.3 VRAM 的位面结构

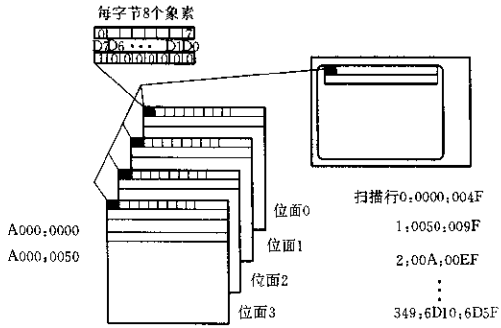


图 13.4 VGAHI 方式下 VRAM

13.3.2 将 VRAM 位面信息存入文件

由于位面结构，因此如何将四个位面的信息提取出来，依序存到磁盘文件中去，这是EGA，VGA 方式下存屏幕图象的关键，在 EGA/VGA 图形适配器上有图形控制器，它含有许多的内部寄存器，其中有一读位面选择寄存器，它和几个其它用处的寄存器共用一个口地址(0x3cf)，因此若要选择某一用处的寄存器，还有一个索引寄存器，它的口地址是 0x3ce，读位面选择寄存器的索引号为 4，这样只要如下编程：

```
outportb(0x3ce,4);   /*将索引号 4 送索引寄存器*/
outportb(0x3cf,0);   /*允许位面 0 可读*/
```

即将读位面选择寄存器索引号 4 送索引寄存器，表示下面用到的口地址 0x3cf 代表的是读位面选择寄存器(这样以后对口地址 0x3cf 的操作就代表对读位面选择寄存器的操作)，然后将选择的位面号 0 送 0x3cf，即送到读位面选择寄存器，这样操作后，再进行读，就可将选中位面的信息读到所指的文件中，由于系统统一编址时，EGA，VGA 的 VRAM 开始段地址分配为 0xA000，所以 VRAM 开始单元地址为 0xa000:0x0000，这样假设已定义 FILE *fp；并且定义 VRAM 开始地址指针为：ptr=(char far *)0xa0000000L，然后用 for 循环：

```
for (k=0;k<4;k++)          /*K 表示 4 个位面号*/
   { outport(0x3ce,4);
     outport(0x3cf,k);
     per=(char far *)0xA0000000L;
     for(i=0;i<38400L;i++)
       {
          putc(*per,fp);
          per++;
       }
   }
```

即可将 VGAHI 方式下屏幕图象保存到 fp 所指的文件中去。对 EGAHI 方式，由于分辨率是 640×350，故占用 VRAM 长度为 28000，所以上面的 i 应小于 28000L，它的计算方法是，由于每个字节代表 8 个象素，故对一行象素，可用 80 个字节表示，对于 VGAHI 480 行方式，占 VRAM 38400 个字节地址，对于 EGAHI，350 行方式，占 VRAM 28000 个字节地址（注意：在 EGA，VGA 方式下，象素依次存放在 VRAM 中，故无奇偶行象素存 VRAM 下、上部之说）。

13.3.3 将文件图象信息写入 VRAM 位面

当要将存在文件中的图象信息重新显示在屏上时，则只要将其二进制信息依序重新装到 VRAM 原来的 4 个位面上的字节中去即可。

由于要将文件写到四个位面上去，且一次只能对一个位面进行写操作，所以在 EGA，VGA 图形适配器上有一个定序器，它可管理写的顺序，作法是先给它的索引寄存器（口地址为 0x3c5）送位面写允许寄存器的索引号 2，表示选中位面写允许寄存器，接着给寄存器（它的口地址为 0x3c5）送位面号，然后将文件中顺序存放的信息写到指定位面的指定地址中去，写的过程就是显示的过程，由于写的过程快于显示的过程，所以显示的中间过程将看不到，而看到的仅是将文件全部装完到 VRAM 后的图象，上述过程可用如下方法编程：

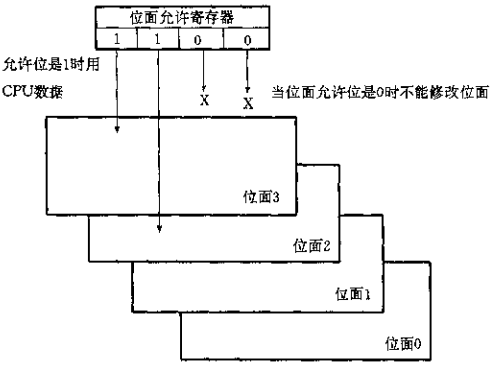


图 13.5 颜色位面写允许操作

```
for (i=0;i<4;i++)
  { outportb(0x3c4,2);          /* 选位面写允许寄存器 */
    outportb(0x3c5,k)           /* 送位面号到位面写允许寄存器 */
    ptr=(char far * )0xa0000000L;
    for(j=0;j<38400L;j++)
      { * ptr=getc(fp);
        per++;
```

```
    }
     k * =2;
    }
```

上面的 k 指选择要写的位面，由于颜色位面写允许寄存器的低 4 位，每位代表一个位面，即 D_0 位代表位面 0，D_1 位代表位面 1，依次类推，故每次选下一位面时，k 要乘 2，即 k * =2；颜色位面写允许寄存器允许写位面的情况如图 13.5 所示。

当某位为 1 时，表示允许对代表的该位面进行写操作，为 0 时则表示禁止对所指的位写操作，即该位面被屏蔽。

13.4 程 序 例

13.4.1 将屏幕图形存入文件的程序

该程序将产生占满屏的各色圆和方框图形并存到了文件名为 pic.dat 的文件中去，主程序产生不同半径和颜色的圆，又产生各种大小不同的方框，然后调用存图象函数 save_pic ("pic.dat")；将其图象保存，由于 for 循环中 i<32 时终止，实际上仅有 16 种颜色，因此超过 16 时，颜色号将取 16 的模。

在函数 save_pic(char * fname)中，当是 EGAHI 模式时，取 j<28000L，若是 CGA 方式，则要将该程序改造，如前面所述。该函数最后使用 fclose(fp)函数，用于关闭存图形的文件，关闭就意味着将数据缓冲区未写到盘文件中的剩余信息全部写完，由于系统初始化时，读位面选择寄存器设置为位面 0 选中，故最后用 outportb(0x3cf,0)恢复原系统初始化设置。

```
#include<stdio.h>
#include<graphics.h>
void save_pic();

main()
{
  int i,k,graphdriver,graphmode;
  graphdriver=DETECT;
  initgraph(&graphdriver,&graphmode,"");
  for (i=0;i<32;i++)
   {
    setcolor(i+1);
    circle(320,240,(i+1) * 15);                    /* 画充满屏的圆 */
    k=i * 10;
    rectangle(320-k,240-k,320+k,240+k); /* 画充满屏的方框 */
   }
  /* save video graphics display */
```

```
    save_pic("pic.dat");
  getch();
  closegraph();
}
/* save part of the video graphics display */
void save_pic(char *fname)
  {
   FILE *fp;
   int i;
   register long j;
   char far *ptr;
   fp=fopen(fname,"wb");                    /*打开二进制写文件*/
   for(i=0;i<4;i++)
    {
     outportb(0x3ce,4);        /*索引寄存器送索引号4,即选中读位面选择寄存器*/
     outporth(0x3cf,i);                     /*给位面选择寄存器送选中的位面*/
     ptr=(char far *)0xa0000000l;           /*VGA的VRAM指针*/
     for(j=0;j<38400l;j++)                  /*38400为满屏图象字节数*/
      {
       putc(*ptr,fp);
       ptr++;
      }
    }
   fclose(fp);                              /*关文件*/
   outportb(0x3cf,0);                       /*恢复原来的设置*/
}
```

13.4.2 将图形文件显示到屏幕上的程序

该程序用于将装在文件名为 pic.dat 中的图形重新显示在 VGA 屏幕上，需要说明的是，由于 EGA，VGA 图形适配器中，通过定序器控制送往显示器的信息流所以在定序器中由颜色位面写允许寄存器控制，将图象颜色信息写到 VRAM 的颜色位面，该寄存器的低 4 位，每位代表一个位面，故顺序写位面时，要 K=K*2，K 为位面号，由于系统初始化时，该寄存器编程为四个位面全允许写，故程序最后用 outportb(0x3c5,0xf)；以恢复原来状态。

```
#include<stdio.h>
#include<graphics.h>
void load_pic();

main()
{
```

```
  int graphdriver,graphmode;
  graphdriver=DETECT;
  initgraph(&graphdriver,&graphmode,"");
  /*load image from file */
  load_pic("pic.dat");
  getch();
  closegraph();
}
/*load the video graphics desplay*/
void load_pic(char *fname)
  {
   FILE *fp;
   int k=1;
   register int i;
   register long j;
   char far *ptr;
   fp=fopen(fname,"rb");                    /*打开只读二进制文件*/
   for(i=0;i<4;i++)
    {
     outportb(0x3c4,2);      /*送索引寄存器索引号2,即选中颜色位面写允许寄存器*/
     outporth(0x3c5,k);                     /*颜色位面写允许寄存器写入位面号*/
     ptr=(char far *)0xa0000000l;           /*VGA显示存储器指针*/
     for(j=0;j<38400l;j++)                  /*读满屏信息字节数*/
      {
       *ptr=getc(fp);        /*从文件中将图形信息字节写到VRAM中,即显示出来*/
       ptr--;                               /*字节地址加1*/
      }
     k*=2;
    }
     fclose(fp);
     outportb(0x3c5,0xf);                   /*恢复位面允许写寄存器的缺省值*/
  }
```

13.4.3 存多幅图形的程序

当需要存多幅图形，当然文件名要有多个，这在一些实际应用中很有必要，如多次实验的结果图，实验条件不同，结果当然一般不同，为此这些图要区别放置在不同图形文件中，当在显示图形时，又要输入要存该图形的文件名，它势必要占据屏幕的某些位置，因而屏幕图象就不完全了，下面的程序解决了这个问题，它首先开辟了一个二维字符数组 char buf[4][1120]，用于存一个原屏幕 14 行象素的信息(每行 640 个象素，相当于占用 80 个字节)，当

将该 14 行信息存完后，这 14 行位置便变成空白，以接收用户输入的文件名，当键入文件名后，再将存于 buf 中的信息，连同屏幕 14 行后的图象信息存于所指的文件中，这样一屏图象就可完整的保留下来。buf[4] [1120]中的 4，表示要存四个位面的同一地址字节值，save_pic()函数中的第一个 for 循环用来设置要读的位面，其中嵌套的 for，则用来读出将要占用的字节内容并保存在 buf 中，即

```
buf[k] [i]= * temp;
```

然后将该字节清零，如此循环 1120 次，将开辟出一个 14 行的空白区，以输入各文件名，程序中接下去的 for 循环，则当文件名输入完后，将保存的原屏幕 14 行内容又重写回 VRAM 去，程序最后将 VRAM 中的满屏图象信息再写到用户命名的文件中去。

另一个保存 14 行原屏幕信息的方法是用 getimage()函数，将该 14 行原屏幕内容通过该函数保存在预先定义的一个缓冲区内(如可起名为 buffer)，当保存屏幕图象文件名输入完后，可用 putimage()函数用 COPY_PUT 的方法将原信息恢复，这样以后的存屏幕图象操作，将会完整的将全屏图象保存到文件中去。由于这种方法简单易行，不用举例，读者就会编程实现。

```
#include<stdio.h>
#include<grapbics.h>
void save_pic();

main()
{
  int i,k,graphdriver,graphmode;
  graphdriver=DETECT;
  initgraph(&graphdriver,&graphmode,"");
  for (i=0;i<32;i++)
   {
    setcolor(i+1);
    circle(320,240,(i+1) * 15);                    /*画出充满屏幕的圆*/
    k=10 * i;
    rectangle(320-k,240-k,320+k,240+k); /*画出充满屏幕的方框*/
    }
  /* save the video graphics display */
  save_pic();
  closegraph();
}
/* save part of the video graphics display */
void save_pic()
  {
   char fname[80];                                 /*存文件名字符数组*/
```

```
FILE *fp;
register long i,j;
int k;
char far *ptr;
char far *temp;
unsigned char buf[4][1120];          /*保存四个位面的屏顶部内容的字符数组*/
for(k=0;k<4;k++)
  {outportb(0x3ce,4);
   outportb(0x3cf,k);
   ptr=(char far *)0xa0000000l;      /*VGA 显示存储器指针*/
   temp=ptr;
   for(i=0;i<1120;i++)               /*保存屏顶部 14 行的信息*/
    {
     buf[k][i]=*temp;                /*保存该字节值*/
     *temp=0;                        /*然后该字节地址清 0*/
     temp++;                         /*地址加 1*/
    }
  }
printf("filename: ");
gets(fname);
if(! (fp=fopen(fname,"wb")))
  { printf("cannot open file\n");
    return;
  }
k=1;
for(j=0;j<4;j++)
  { outportb(0x3c4,2);     /*索引寄存器送索引号 2,即选位面写允许寄存器*/
    outportb(0x3c5,k);                /*位面写允许寄存器写入要选择写的位面*/
    ptr=(char far *)0xa0000000l;
    temp=ptr;
    for(i=0;i<1120;i++)
     {*temp=buf[j][i];
    temp++;
     }
    k*=2;
  }
  outportb(0x3c5,0xf);
  for(k=0;k<4;k++)
    {outportb(0x3ce,4);     /*向图形控制器索引寄存器送索引号 4,即选中读位面选择
```

```
                                寄存器*/
        outportb(0x3cf,k);              /*位面选择寄存器送选中的位面*/
        ptr=(char far *)0xa0000000l;
        for(i=0;i<38400l;i++)
        {
        putc(*ptr,fp);                  /*读位面数据到文件中*/
        ptr++;                          /*选下一个字节地址*/
        }
      }
    fclose(fp);                         /*关闭文件*/
    outportb(0x3cf,0);                  /*恢复原始状态*/
}
```

13.4.4 显示图象程序例

该程序将把以二进制形式存储的满屏图象文件重新显示在屏幕上，load_pic 函数首先将屏幕 14 行信息暂存起来，然后等待接收输入的文件名，若该文件打不开时，便恢复原先的 14 行信息，然后返回主函数，若该文件能打开，则将该文件装入以 0xa000：0000 地址开始的 VRAM 中，即显示该图象。

```
#include<stdio.h>
#include<graphics.h>
void load_pic();

main()
{
  int graphdriver,graphmode;
  graphdriver=DETECT;
  initgraph(&graphdriver,&graphmode,"");
  /* load image from file */
  load_pic();
  getch();
  closegraph();
}
/* load the video graphics display */
void load_pic()
  {
   char fname[80];                  /*存文件名字符数组*/
   FILE *fp;
   register long i,j;
   int k;
```

```
char far *ptr;
char far *temp;
unsigned char buf[4][1120];        /*存顶部14行象素信息数组*/
for(k=0;k<4;k++)
  {outportb(0x3ce,4);
   outportb(0x3cf,k);
   ptr=(char far *)0xa0000000l;
   temp=ptr;
   for(i=0;i<1120;i++)             /*存顶部14行信息*/
    {
     buf[k][i]=*temp;              /*存该字节信息*/
     *temp=0;
     temp++;                       /*清零该字节*/
    }
  }
printf("filename:");
gets(fname);                       /*要求输入文件名*/
if(!(fp=fopen(fname,"rb")))
  {printf("cannot open file\n");
   k=1;
   for(j=0;j<4;j++)
    {outporth(0x3c4,2);
     outportb(0x3c5,k);
     ptr=(char far *)0xa0000000l;
   temp=ptr;
   for(i=0;i<1120;i++)             /*恢复原来顶部14行信息*/
    {*temp=buf[j][i];
     temp++;
    }
    k*=2;
   }
   return;                         /*选下一位面*/
  }
 k=1;
  for(i=0;i<4;i++)
    {outportb(0x3c4,2);
     outportb(0x3c5,k);
     ptr=(char far *)0xa0000000l;
     for(j=0;j<38400l;j++)
```

```
        {
         *ptr=getc(fp);
         ptr++;
        }
       k*=2;
      }
    fclose(fp);                    /*关闭文件*/
    outportb(0x3c5,0xf);           /*恢复位面写允许寄存器为全允许写*/
}
```

13.5 屏幕图形和图形文件的打印输出

屏幕上显示的图形或以二进制形式存储的图形文件常需要用打印机打印出来，打印机打印文本很方便，然而要将以象素组成的图形打印出来，却不那么方便。虽然DOS提供了一个常驻内存的外部命令GRAPHICS.COM，当运行该程序后，只要按下prtscr键，就可将屏幕图形打印出来，但它只能打印低分辨率的图形(如640×200，320×200)，对于EGA，VGA高分辨图形就打印不出来了，且打印无法进行控制，因此用Turbo C语言编程实现EGA，VGA高分辨图形的打印输出，实用性很大，下面就这个问题，介绍用程序实现打印的方法，由于现在普遍使用点阵式打印机，则以此类型打印机为例，现让我们看一下打印机适配器和点阵打印机的打印原理。

13.5.1 打印机适配器及其寄存器结构

由于PC机高速的数据处理功能和受机械制约的慢速打印机输出之间的矛盾，因而两者之间必须要有一个缓冲接口，这就是打印机适配器接口，它和接显示器的图形适配器一样，都起到了PC主机和所接打印设备之间数据传输的缓冲作用，主机向它们输出数据和进行控制的命令由该适配器接口作出转换，以使打印机能够接收。在PC286、386、486、586微机中打印机适配器和RS232串行口做在一块卡上，通称两串一并卡，这一并就是指接打印机的适配器，因打印机接的是并行口，即8位PC数据总线一次并行将8位二进制数据送打印机打印，打印机适配器上有三个寄存器，用来控制打印机和接收CPU输出打印数据到打印机，现简要作以介绍：

1. 数据输出寄存器

它用来寄存PC主机送来打印的数据，使得在高速主机与慢速打印机之间起到缓冲作用，当主机送来打印的八位数据，而打印机接收但未打印完时，暂停止接收下面的数据，即主机暂停止发送，待打印完时，再继续下一次打印的数据接收，该寄存器通常占用口地址0x378。

2. 状态寄存器

该寄存器提供打印机当时的状态，如打印机忙否，即正在打印的一个数据是否已打印完，若打印完，则把该寄存器(8位)第7位置1，表示不忙，否则为0表示忙，这时数据寄存器将不再接收新数据，因而我们编写打印机驱动程序时，便要查询此信号，若该位为1，便可再

向数据寄存器发送下一个要打印的数据。该寄存器各位可由 Turbo C 的 biosprint 函数读出，如 byte=biosprint(2,0,0)；表示读 1 号打印机的状态，读出的状态字在 byte 变量中，这个变量值就是状态寄存器各位的组合值通过或运算分离出来的，其反映打印机的状态如下：

第 0 位即 0x01　　设备超时，表示设备硬件故障或连接错误。

第 3 位即 0x08　　打印机输入输出错。

第 4 位即 0x10　　打印机已联机。

第 5 位即 0x20　　打印机无纸。

第 6 位即 0x40　　打印机已确认并收到信号。

第 7 位即 0x80　　打印机不忙。

3. 控制寄存器

该寄存器用来接收主机的命令，初始化打印机，如打印机打印模式的设置，打印机打印动作的控制等，其口地址为 0x37A，各位含义如下：

第 0 位=0 静态时的状态。

=1 使打印机读取主机送来的数据后，立即复位。

第 1 位=0 不自动换行。

=1 打印一个回车动作后自动换行。

第 2 位=0 对打印机初始化。然后被置 1。

=1 静态时的状态。

第 3 位=0 使打印机脱机。

=1 打印机联机。

第 4 位=0 禁止打印机中断。

=1 允许打印机中断，并产生 IRQ7 中断请求信号。

第 5—7 位 未用。

一般在使用打印机时，编写驱动程序，常采用查询方式，一般检查打印机适配器状态寄存器第 7 位，当其为 1 时，即打印机不忙时，向其发送打印数据，若为 0，即等待，直到为 1 时止。

对 9 针打印机，收到一个 8 位数据后，便立即进行打印，对 24 针打印机，要待收到三个 8 位数据乘每行列数后，才开始打印，即一个打印行的数据全收到后，立即开始打印。一个打印行共有 3 个 8 位数据乘每行列数，因 24 针击打一次，需 3 个 8 位数据，故要列数乘 3。

另一种驱动打印机的方法是采用中断方式，即使打印机适配器控制寄存器第 4 位置 1，即允许打印机中断，这样打印机每打印输出一个字符后，立即向 CPU 发出 IRQ7 中断请求信号，要求发送下一个字符，这样 CPU 不必查询打印机忙否，立即输送要打印的数据，关于这方面的内容将不作详细介绍，我们编写打印图形的驱动程序将采用查询方式。

关于打印机打印图形的原理，将在下面进行介绍。

13.5.2　点阵式打印机打印图形的原理

点阵式打印机通常有 9 针和 24 针之分，这些针垂直排列，当带有这些针的打印头移过纸面时，某些针被电流脉冲击发而撞击色带，在纸上留下一个色点，对 9 针打印机来说，字符一般可用 8 根针打印出来，对于长的字母如 g，y 则用第 9 针打印其下部。文本方式下当打印

字符时，由其对应的 ASCII 码找到打印机存储器中存的该字符字模，控制针击打，随着打印头移动，则可打印出其字符来。对于 24 针，则将其分成 3 组，每组 8 针，字符对应的字模点阵将变成 24×24，打印原理同 9 针。

打印屏幕图形时，由于图形由象素组成，这些象素点恰好对应打印机针头的针，当屏幕上某点象素值为非零时（该值代表了该象素的颜色），便由对应的针打印出来，象素值为零时，则不打印，打印机这种工作方式不同于打印字符，这种工作方式称为图形方式，对于 9 针打印机，使用其中的 8 针，它对应一列上的 8 个象素，恰为 8 位（一个字节），位 0 对应于第 1 个针（打印头上最低的针），位 7 对应于第 8 个针，如图 13.6a 所示，图 13.6b 示出了输入字符值为 0x34 和 0x80 时打印出来的点。对于 24 针打印机，24 针分成了三组，每组 8 个针，对应一个字节，24 针对应一列上的 24 个象素，当打印机接收到 3 个字节数据（即 24 个象素值）时，便打出一列图形，先接收到的 8 位数据，对应上面的一组 8 个针，依此类推，图 13.6c 示出这种对应关系。

对于 VGAHI 方式，640×480 分辨率的情况下，对 24 针打印机，一行宽度为 24 个针，即对应 24 个纵排的象素，若要纵向打印则要打印 20 行，将 480 个纵向（行）排列的象素打印完，又横向为 640（即有 640 列），故 24 个针需横向移动（x 方向）640 次，这样就打印出 640×480 个象素，为了保持打印出的象素点纵向间距都是相同的，每行最后 24 针打完后，要由用户程序发送回车（CR，ASCII 码为 13），换行（LF，ASCII 码为 10），并控制打印机下一行与上一行的间距为 24 针（即上一行最下面第 24 针位置和本行最上面第 1 针的位置的间距应和针头之间的间距一样，这样整个屏幕的象素点输出在纸上，才能显得均匀，在图形方式下，这个间距可以通过给打印机发送相应的控制命令进行控制。

以常用的 LQ-1600K 打印机为例，这种打印机在水平方向打印出点的间距 可有多种，如单密度时，每英寸可打印出 60 点，双密度时，每英寸可打 120 个点，对宽度为 8 英寸的窄行打印纸，单密度下每行最多可打 480 列，双密度下则可打 960 列，还有三倍密度（每英寸可打 180 个点），六倍密度等，由于其针间距为 1/180 英寸，故行间距应设置为 24/180 英寸，LQ-1600K 提供了相应的打印机控制命令，现简述如下：

1. 图象方式指令

ASCII 码	ESC	*	m	n1	n2	data
十进制数码	27	42	m	n1	n2	data

ESC 为命令前缀，* 表示设置打印机为图形方式，数字 m 表示图象选择，就是前面所说的单密度、双密度、三倍密度、六倍密度打印形式（其对应的 m 分别为 32，33，39，40），n1 n2 变相的表示了每行打印的列数，如一行要打印 640 列，则 n1＋256×n2＝640，即 n1＝640 mod256，n2＝INT（640/256），这样 n1＝128，n2＝2，后面的 data，表示发送的要打印的字节数据。

2. 设置行间距命令

ASCII 码	ESC	3	n
十进制码	27	51	n

该命令设置 n/180 英寸行距进行换行，n 为 0～255 的任意值。

3. 设置走纸 n/180 英寸

ASCII 码	ESC	J	n

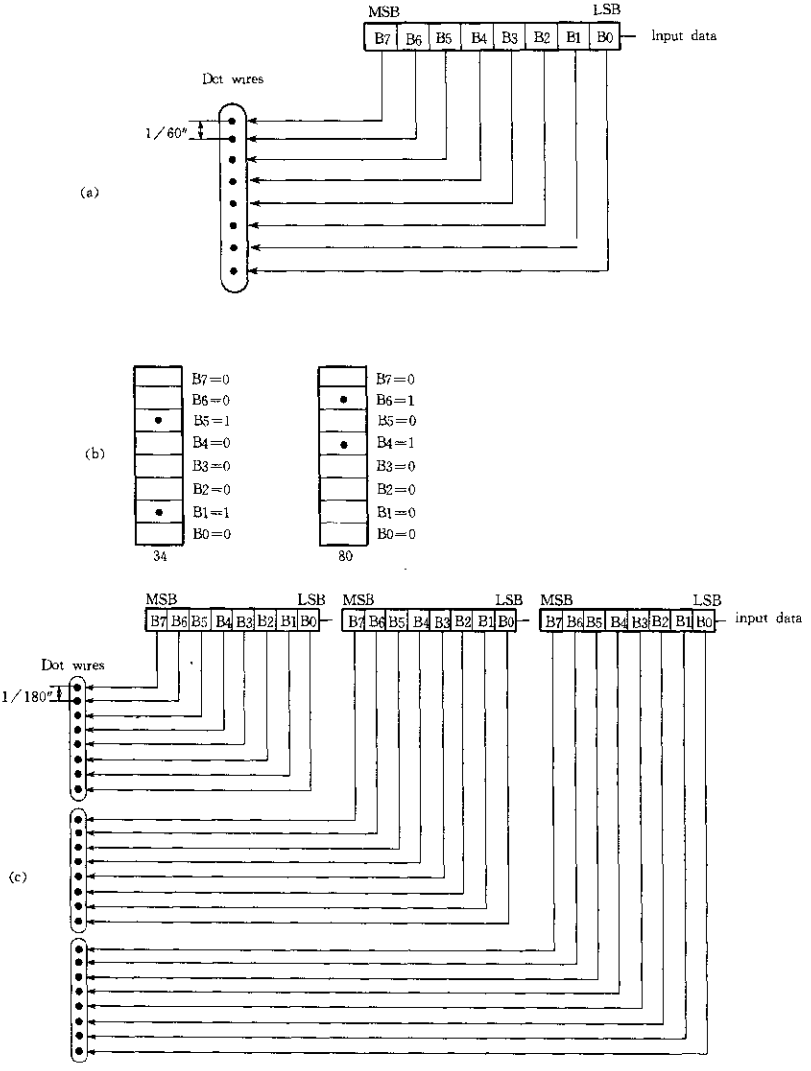


图 13.6　打印针与接收的打印字节的对应关系

十进制码　　27　　74　n

该命令将使得打印机产生走 n/180 英寸纸的动作，可以看作是产生了一次换行，如当 n=24 时，便产生 24/180 英寸的向前走纸。

4. 打印机初始化命令

ASCII 码　ESC　@

十进制码　27　64

该命令将清除前面设置的打印机命令，恢复打印机缺省的工作状态，即复原。

上面命令格式中，十进制码表示对应 ASCII 码字符的十进制表示。

当打印机型号不同时，上述命令形式有差别，但控制功能大同小异，一般均按所谓 EPSON 标准。

打印机控制命令有许多，一般打印机用户手册均有介绍。

13.6　屏幕输出打印编程要求

在 Turbo C 图形方式下，可以用 getpixel()函数得到指定位置的象素颜色值，当用普通打印机时，则仅能打印黑白两色的图形，或最多打印出灰度有差别的黑白两色图形，因此若要用单色打印机打印彩色象素，则要将得到的象素值经过变换，如打印黑白图形时，当得到的象素点为非零值时，则令一个字节对应的该位为 1 值，否则为 0(即该象素点不显示为背景色)，这样用 y 方向的 8 个象素值可构成一个字节，24 个象素构成控制 24 针击打的三字节，当打完后，x 加 1，再取 24 个同 y 向象素值，再完成 24 针的击打，反复如此，当 x 增加到 640 时表示该行象素(共有 24×640)打完，接着 y 再增加 24，此时用打印机控制命令走纸 24/180 英寸，接着重复上述的打印过程，对 480 行的 VGAHI 显示方式，要重复 20 次即打印 20 行，可将一屏图形打印完。

13.6.1　打印屏幕图形例

该程序将把由主程序产生的充满屏的圆和方框组成的图形在 LQ-1600K 打印机上打印出来。函数 printscr 的功能是用来控制打印机打图，该函数首先设置了一个缓冲区 buffer，用来保存一个将要被提示信息占用的原屏幕上的一小框内图象，以便要打印屏幕时，将该小方框用 putimage(0,0,buffer,COPY-PUT)恢复，接着开一小窗口显示提示信息，若打回车，则该函数又调用读屏函数 readscr()。readscr()函数用 getpixel(x,i * 24+j * 8+y)来读确定位置的象素值，若象素值不为 0(即不是背景色)，则令 K=1，否则为 0，这个 K 值代表了打印针是否击打色带，具体是对应哪个针，或者说具体对应一个字节的哪一位，则由函数 shlbit()来完成，它将对应 K 值的那个二进制位置成相应的 0 或 1，对应关系是字节最高位对应低地址象素点，字节最低位代表 8 个象素地址中最高的那一个象素点，这通过由最低位左移 0 或 1 来实现。

```
#include<stdio.h>
#include<graphics.h>
void printscr();

main()
{
  int i,k,grapbdriver,graphmode;
```

```
  graphdriver=DETECT;
  initgraph(&graphdriver,&graphmode,"");
  for (i=0;i<32;i++)
   {
    setcolor(i+1);
    circle(320,240,(i+1)*15);
    k=10*i;
    rectangle(320-k,240-k,320+k,240+k);
   }
  /*print the video graphics display*/
  printscr();
  closegraph();
}
/* print part of the video graphics display */
void printscr()
  {
   #include<alloc.h>
   int size,i;
   char ch;
   void *buffer;
   size=imagesize(0,0,163,33);
   buffer=malloc(size);
   getimage(0,0,163,33,buffer);              /*保存矩形框内信息*/
   setcolor(10);
   window(0,0,160,30);
   rectangle(0,0,163,33);
   bar(3,3,160,30);
   setfillstyle(1,9);
   setcolor(12);
   outtextxy(10,6,"<CR>----print");          /*在矩形框内显示 print*/
   outtextxy(10,18,"<ESC>_quit");
   while(ch!=27&&ch!=13)
   ch=getch();                               /*等待按键*/
   putimage(0,0,buffer,COPY_PUT);            /*恢复原来占用的信息框内原信息*/
   free(buffer);                             /*释放占用的缓冲区*/
   if(ch==13)                                /*读屏幕*/
     readscr();
   return;
   }
```

```
readscr()
  {
   #include<stdio.h>
   char c,ch;
   int x,y,i,j,k,dot;
   unsigned char data[3][640];
   for(i=0;i<20;i++)                          /*共打印 20 行,相当 480 象素行*/
    {
     for(x=0;x<640;x++)                       /*x 方向 640 个象素点*/
      for(j=0;j<=2;j++)                       /*y 方向 3 个字节*/
       data[j][x]=0x00;                       /*清 0*/
       for(x=0;x<640;x++)
      for(j=0;j<=2;j++)
       for(y=0;y<=7;y++)                      /*每个字节有 8 位*/
         {
          dot=getpixel(x,i*24+j*8+y);         /*得到象素颜色值*/
          if(dot! =0)                         /*若象素不是背景色*/
            k=1;                              /*K 加 1*/
          else                                /*否则 K 为 0*/
            k=0;
          data[j][x]=shlbit(data[j][x],k,y);
         }
     fprintf(stdprn,"%c%c%c%c%c",27,42,39,128,2);/*发 ESC'*' n1 n2 命令*/
     for(x=0;x<640;x++)
       for(j=0;j<=2;j++)
         biosprint(0,data[j][x],0);           /*向打印机送一个字节数打印*/
     fprintf(stdprn,"%c%c%c",27,51,24);       /*发 ESC 3 n 命令定行距*/
     fprintf(stdprn,"%c%c",13,10);            /*给打印发回车换行命令*/
    }
  }
  int shlbit(unsigned char c,int m,int n)      /*将象素值转换成对应字节中的位值*/
    {
     if(n==7)
       {
        if(m==0)
        return(c&0xfe);
        return(c | 0x01);
       }
        else
```

```
    {
    if(m==0)
      return((c&0xfe)<<1);
    return((c | 0x01)<<1);
    }
}
```

13.6.2 打印图形文件例

对于存于文件中的图形象素点文件，可以用下面程序打印出来，不过打印出来的图形与原屏幕位置不同，该图形旋转 90°，在程序中 readscr()函数用来将前面介绍过的用 Save_pic 程序产生的保存屏幕图形的文件 pic.dat，用打印机打印出来，该函数首先设置打印机行距 23/180 英寸，然后从打开的 pic.dat 文件中取出图象信息，以字节形式存在 c0[j][k]字符数组中，它存储形式是以 j 为行号，每行存 80 个字节，这恰相当于 480×640 分辨率形式，即存了 480 个行的信息，每个行共 80 个字节。接着用 for 循环，采用 DOS 中断(功能号为 5)，发送字符到打印机，发送的字符是原屏幕上每行相同位置的一个字节，共取三个字节打一次，然后再换一行再取相同位置的三字节打印，这样取了 480 行的相同位置三字节后，便实际上在打印机上打印了一行，所以设置打印机图形方式命令中的列数为 480，即 n1＝480mod256＝224，n2＝INT(480/256)＝1，然后换一行(对屏幕图形来说实际上是再过 3 列，又重复 480 次循环，即打印出第二行来。因而这样打出的图形实际上进行了旋转，即列变成了行，且本末倒置，由于图形文件中存了 4 个位面的信息，即共有 4 个 38400 字节信息，该程序中实际上仅读了 0 号位面的 38400 字节信息，实际仅将颜色号为×××1 的象素点打印出来了，即 7 种颜色对应的象点被打印出来，对于颜色号为偶数的，则视为背景色。若要将 16 种色的点均打印出来，则需将 4 个位面存的信息均读出来，且相同字节的位进行运算(如或运算)，然后构成新的 38400 个字节值，进行打印即可。

利用彩色打印机可以打印出彩色图形，如 Ax-1900 型号的彩色打印机，其控制命令也采用 EPSON 标准，黑白打印控制命令同 LQ-1600，当进行彩色打印时，要使用控制彩色色带的命令，因该打印机色带由多彩色的宽色带组成，色带架由步进电机带动可变换俯仰角，从而可使被选中颜色的色带条对准打印头，当打印针击打时，便打印出相应的彩色来，色带控制命令为：

ASCII 码　ESC　r　n；

十进制码　27　114　n；

n：	0	1	2	3	4	5	6
对应颜色：	黑	品红	青蓝	紫	黄	白	绿

为了实现七色打印，在同一行必须重复打印七次(即 7 次行距均为 0)，打印颜色可由用 getpixel()函数读到的象素颜色号选择相一致的 n 值，由于象素颜色号有 16 种，为了对应，可将读出的第 3 位面的位值不要，即仅用 3 个位面值来和 n 呼应，这样就仅能打印出前 7 种颜色来。由于彩色打印机使用不普遍，因而不在此列举程序，若手头有彩色打印机，则遵循上述原则，参照黑白打印程序，不难编出彩色打印屏幕图形的程序来。

```
#include<stdio.h>
#include<dos.h>
#include<graphics.h>

main()
{
  /*print the video graphics display*/
   char ch;
   int driver,mode;
   driver=DETECT;
   initgraph(&driver,&mode,"");
   setcolor(10);
   window(0,0,160,30);
   rectangle(0,0,163,33);
   bar(3,3,160,30);
   setfillstyle(1,9);
   setcolor(12);
   outtextxy(10,6,"<CR>----print");
   outtextxy(10,18,"<ESC>__quit");
   while(ch!=27&&ch!=13)
   ch=getch();
   if(ch==13)
     readscr();
  closegraph();
   }
readscr()
   {
    #include<stdio.h>
   FILE *fp;
   register i,j,k;
   unsigned char c0[480][81];
   fprintf(stdprn,"%c%c",13,10);          /*回车换行*/
   fprintf(stdprn,"%c%c%c",27,51,23);     /*设置打印机行间距23/180英寸*/
   fp=fopen("pic.dat","rb");
   for(j=0;j<480;j++)
     for(k=0;k<80;k++)
       c0[j][k]=getc(fp);                 /*取C号位面的象素点位值*/
   for(i=0;i<480;i++)
     c0[i][80]=0;
```

```
for(k=0;k<81;k=k+3)
  {
   fprintf(stdprn,"%c%c%c%c%c",27,42,39,224,1);   /*设置打印机图形方式
                                                    及行长度*/
   for(j=0;j<480;j++)                  /*进行行打印*/
     {
      out((int)c0[j][k]);
      out((int)c0[j][k+1]);
      out((int)c0[j][k+2]);
     }
   fprintf(stdprn,"%c%c",13,10);       /*每行完回车换行*/
  }
 fclose(fp);
}
out(int x)                             /*DOS功能调用*/
  {
   union REGS rg;
   rg.h.ah=05;
   rg.h.dl=x;
   intdos(&rg,&rg);
  }
```

13.7 用单色打印机打印彩色图象的方法

单色打印机只能打印黑白图象，当打印彩色图象时，我们只是作了一些近似，如前面打印屏幕图形的一些程序例子，将象素颜色值大于1的打印出来，而象素颜色值为0的，则不打印，这种打印方法称为阈值法，也称为两值法，即将象素颜色值设定一个阈值，大于或等于阈值的象素点打印出来，小于阈值的象素点则不打印出来，即最后将象素颜色值规一为0、1两种值，当然也可反相打印，这是一种最简单的打印方法，除此之外，还有灰度打印方法，它能将不同颜色用不同的灰度表示出来，因而真实感强，有层次，能较好地反映彩色图象。所谓灰度是这样定义的，我们知道亮度方程为：

$$L=0.3r+0.59g+0.11b$$

即根据红、绿、蓝三基色原理，光是由这三种颜色混合的，上面方程中的r,g,b则代表这三色，L代表亮度，方程后面的项则代表了这三种色的比例，可以看出r,g,b均为1时，亮度最强，此时L=1，表示最高级灰度，当均为0时，则L=0，表示最暗，灰度最低，因而当r,g,b所占成分不同时，便可得到不同的灰度级别，当然可根据需要，划分成不同间隔的灰度级别来。

象素的灰度值的确定，可用多种方法，对于划分级别少的，我们可以将颜色分成几组，如对640×480分辨率的16色显示(VGHI)，可将16色分成四组，形成四种灰度，当得知象素

为某一颜色组时，便用对应的灰度将其打印出来，更精细的划分，则可按上面的亮度方程计算，如对于 VGA 显示器，其象素显色过程如在调色板颜色设置函数中所叙述的那样，象素的颜色由颜色寄存器中的 18 位值所决定，当某象素显色时，由其颜色值找到调色板寄存器（即索引到 16 个之 1 的调色板寄存器），然后由模式控制寄存器的第 7 位（最高位）是 1 或 0，来决定如何再寻址颜色寄存器，当是 1 时，则由调色板寄存器的 4 位作为低 4 位，再由颜色选择寄存器的 4 位作为高 4 位，组合成 8 位地址来寻址 0～255 个颜色寄存器，对于模式控制寄存器第 7 位为 0 时（VGAHI 显示方式属此情况），则由调色板寄存器的 6 位和颜色选择寄存器的 2 位（作为高位）组合成 8 位来寻址颜色寄存器，当找到相应编号的颜色寄存器后，其 18 位值则分别表示了该象素颜色的灰度，对其读时须三次，第一次读出的 6 位数是红色的亮度，第二次读出的 6 位数是绿色的亮度，第三次读时，读出的是蓝色的亮度，当然用显示器 BIOS 的 int 10H 调用，也可知颜色的亮度，该调用入口参数和调用形式如下：

```
r.h.ah=0x10;
r.h.al=0x15;
r.x.bx=颜色寄存器地址;
int86(0x10,&r,&r);
```

调用后，便可由 dh 中的值得到红色亮度值，ch 的值为绿色亮度值，cl 中的值为蓝色亮度值，可知最亮值为 r=111111B=63，g=63，b=63，若 L=1，则 r=读出的红色值/63，g，b 同样，这样，只要从颜色寄存器中读出各色的颜色值。再分别除以 63，从而可得到 r，g，b 值，将其代入亮度方程，即可得到该象素对应的灰度。

按照灰度打印，通常采用两种方法，下面分别介绍：

13.7.1 模式法

是将屏幕上的一个象素，用 4＊4 的小方块代替，即每个小方块等分 16 块，各块进行填色或不填，这样可有 17 种方式，它们可代表 17 种灰度，将象素的灰度重新归类，也使其有 17 种，这样当得到一个象素的灰度值时，便用相应代表该灰度的小方块代替，并打印出来，这样就可打印出不同颜色用不同灰度表示的图形了，不过这种方法将放大了原来的图形尺寸，即放大 4 倍，例如用 2×2 矩阵代替，即每个小方块分成 4 块，其填色情况可分成四种，如图 13.7 所示，它可代表四种灰度，若将象素颜色灰度分成四个等级，则可打印出四种灰度的图来，不过图则放大了 1 倍。

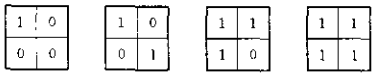


图 13.7 代表四种灰度的 2×2 矩阵

13.7.2 抖动法

由于点阵打印只能打点或不打点，对于单色打印，打印灰度是通过一小区域打印点的不同分布来得到的，如该区域所有象素点均打印点，则表示最亮，若全不打点，则表示最暗，对其它分布情况，则可定义出不同级别的灰度来，屏幕象素的颜色值可通过一种所谓抖动处

理，而得到打印点的分布，从而使打印机打出与象素点对应的灰度来，使得颜色图象通过灰度的不同，用黑白图象来打印出来。

颜色抖动处理是图象处理中的一种技术，对黑白打印图象，抖动可通过抖动矩阵来实现，最简单的办法是采用 8×8 的 0、1 矩阵，在矩阵中，使矩阵元素为 0 或 1 的分布不同，从而得到不同灰度的矩阵，这种原理如同模式法所述的矩阵一样，如若定义矩阵元素全为 1，表示最亮，即灰度为 1，若矩阵元素全为 0，则最暗，表示灰度为 0。介于两者之间的分布，则可区分出不同等级的灰度来。

与模式法不同的是，抖动法不是将一个象素对应一个矩阵，而是将屏幕上的一个象素颜色和一个代表该颜色灰度的抖动矩阵对应起来，用对应抖动矩阵相应位置元素的值代替该象素，来决定打点或不打点(即 0 或 1)，一般常用于将屏幕上 8×8 点阵区域各点值用各点对应灰度矩阵相应位置元素值代替，重构成打印的矩阵，这样在打印时，就会打印出视觉上感到有深浅和花样不同的图，用它们就表示出了不同的颜色或不同的灰度，不过用这种方法打印线段时，有时失真。

13.7.3 用模式法打印图象例

该程序是用模式法进行图象按灰度打印的例子，它采用如图 13.7 所示 2×2 矩阵代表一种灰度的象素，将象素颜色分成 4 级灰度，该程序演示了在 VGAHI 方式下，即 16 色 640×480 分辨率情况下，如何按 4 种灰度进行打印，程序首先在屏上画出三种不同颜色的线，再画出方框，用红色填充，又画出用亮红色填充的一矩形条和用绿色实线填充的一长条，当按任意键后接着调用打印函数 set_print()，该函数首先检测打印机忙否(该程序适于 LQ-1500 或 LQ-1600)，即若等于 0x90(说明打印机状态寄存器第 7 位为 1，表示不忙，第 4 位为 1，表示打印机已联机)，则向打印机发送行间距命令，接着进入 for 循环，y 变量表示行，由于打印头为 24 针，即可打印出 24 个点来，由于该模式法采用一个象素代表纵向打印的二个点，故 y 增量取 12，即取 12 个象素点，但可打印出 24 个针点。如此循环，直到 y 等于 480，则可打印出所有行来，for 循环一开始便设置打印机为图形方式，下面内层循环中，x 变量表示列，当 x=640 时便可打印出一行来(共打印出 2×24×640 个点表示的打印行)，再下面的循环则表示打印行上的一列 12 个象素时，打印机应打印的 2 列点的值，当循环变量 m=1 时，表示打印代表灰度的矩阵左半部，m=2 时，表示打印矩阵的右半部，J 变量表示 24 针对应的打印字节，J=0 表示 24 针的头 8 针对应的字节，依次类推，这样循环三次，即向打印机发送三个字节的值后，便打印出 24 针来，即打印出 12 个象素来，又按 m 是 1 还是 2，若 m=2，则再按模式附加 24 个点值，循环中的两个 switch()语句，便是按读出的象素值，当 m=1 时，按灰度模式矩阵，给出该点在一个字节中应代表的值，当 m=2 时，便在打印行的奇数列构造三个字节(按灰度模式矩阵)，如此循环便可打印出横向，纵向均是原屏幕象素 2 倍的图来。

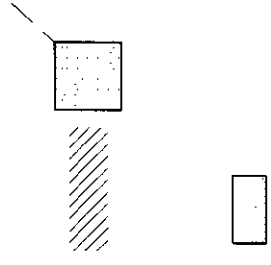


图 13.8 用模式法打印的图

用该函数打印的图如图 13.8 所示。

```
#include <bios.h>
#include <dos.h>
#include <stdio.h>
#include <graphics.h>
define   Row   480
define   COLUM   640
void set_print();

main()
  {
     int driver=VGA;
     int mode=VGAHI;
     initgraph(&driver,&mode,'');
     setcolor(1);
     line(0,0,10,10);
     setcolor(2);
     line(15,15,20,20);
     setcolor(3);
     line(25,25,30,30);
     setfillstyle(1,BLUE);
     setcolor(RED);
     rectangle(20,20,60,60);
     floodfill(25,40,RED);
     setfillstyle(1,LIGHTRED);
     setcolor(WHITE);
     rectangle(130,100,150,140);
     floodfill(140,120,WHITE);
     setcolor(BROWN);
     setfillstyle(3,GREEN);
     bar(30,70,50,140);
     setch();
     set  print();
   }
  void set_print()
    {
     int n1,n2,i,j,x,y,ny,m,ty,hyte,color,n;
     if(biosprint(2,0,0)==0x90)          /*打印机不忙且已联机*/
      {
       biosprint(0,27,0);                /*设置打印行间距*/
```

```
biosprint(0,'3',0);
biosprint(0,24,0);
n1=(2*COLUM)%256;n2=(2*COLUM)/256;
for(y=0;y<ROW;y+=12)
 {
 biosprint(0,27,0);
 biosprint(0,'*',0);
 biosprint(0,39,0);                    /*设置三倍密度打印*/
 biosprint(0,n1,0);
 biosprint(0,n2,0);
 for(x=0;x<COLUM;x++)
   for(m=1,m<3;m++)
    {ty=y;
     for(j=0;j<3;j++)
      {byte=0;
       for(ny=0;ny<4;ny++)
        {if(ty+ny<ROW)
         {
          color=getpixel(x,ty+ny);
          if(color)
           switch(color/4+1)
           {
            case   1:if(m==1)n=0;
                       else n=3;break;
            case   2:if(m==1)n=0;
                       else n=2;break;
            case   3:if(m==1)n=0;
                       else n=1;break;
            case   4:n=1;break;
           }
           switch(n)
           {
            case   0:byte+=1<<(7-2*ny);
                       break;
            case   1:byte+=1<<(7-2*ny)+1<<(7-2*ny-1);
                       break;
            case   2:byte+=1<<(7-2*ny-1);
                       break;
            case   3:byte+=0;
```

```
                    break;
                }
              }
            }
          }
          biosprint(0,byte,0);ty+=4;
        }
      }
    biosprint(0,13,0);
    biosprint(0,10,0);
  }
 }
}
```

13.7.4 用32级灰度抖动法打印图象例

该程序采用抖动法进行图象打印，首先定义了代表32级灰度的一个抖动矩阵数组，每个行元素代表一个8×8的抖动矩阵，共有32行，8×8矩阵中各元素值非零即1，用16进制数将8个元素合起来表示。程序中read-grade()函数用来将VGAHI方式下对应的16个调色板寄存器对应的颜色转换成相应的灰度值，其原理如同在用单色打印机打印彩色图象的方法中所述，在我们用setcolor(颜色号)设置颜色时，其颜色号即代表了相应索引号的调色板寄存器，而该寄存器中值并不最终代表某种颜色，该值还要视颜色选择寄存器的值而定，在VGAHI方式下，当读模式控制寄存器时，其第七位为零，它表示调色板寄存器的2位值

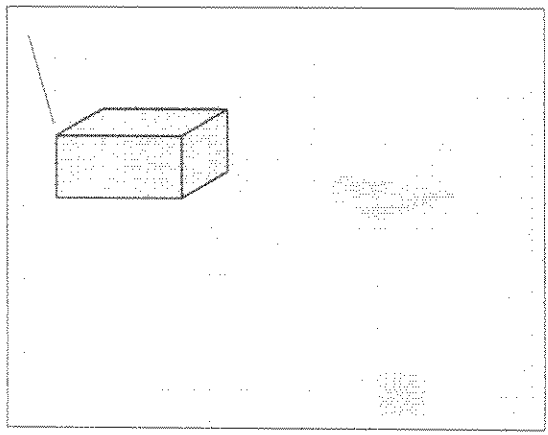


图 13.9 用32级灰度矩阵打印的图

(代表高 2 位)组成 8 位值,选址 0～255 个颜色寄存器,选中的颜色寄存器的值才是最终的颜色,它的值代表了灰度。该程序中三次 BIOS 调用就是为此目的,这样就将 16 个调色板寄存器代表的颜色确定了相应的灰度级别,并将其存到了用 grade_p 指针指出的区域中。

程序中的 convert_data()函数将读出的象素点转换成对应灰度矩阵中的相应点的 0 或 1 值,由于用 24 针打印,故两个 for 循环完成了一个打印行所占有的象素 24×640 点的转换,接着用 set_print()函数完成一行的打印,如此循环,便可完成 20 个打印行(即象素行 480)的打印。

当然用这种方法理论上很理想,实际上由于打印机打印点的密度若不是有太大的差异,则肉眼难于区别,几乎视作相同,尤其有时打线段时,可能打不上,简单的办法是定义 16 个抖动矩阵,对应于 16 个调色板寄存器的索引号,当读到某点象素值时(即索引号),即用相应抖动矩阵中的某元素代替,13.7.5 列出的程序就是这样一个程序,它仅定义了 16 级灰度,即用 16 种抖动矩阵。代替 16 级灰度的象素矩阵。

图 13.9 是采用 32 级灰度矩阵打印出的图。下面就是相应的程序:

```
#include <bios.h>
#include <dos.h>
#include <stdio.h>
#include <graphics.h>
float grade_value[16];
void read_grade(void);
int linel;
int set_print(int col,int row,unsigned char *buf);
unsigned char prn_data[1920];
void read_grade(void);
void convert_data(int line);
unsigned char color_grade[32][8]={{0,0,0,0,0,0,0,0},/*32 级灰度抖动矩阵数组*/
    {0,0x40,0,0,0,4,0,0},
    {0,0x44,0,0,0,0x44,0,0,},
    {0,8,0,0x22,0,0x80,0,0x22},
    {0x88,0,0x22,0,0x88,0,0x22,0},
    {0,0x44,0,0x92,0,0x44,0,0x29},
    {0x55,0,0x44,0,0x55,0,0x44,0},
    {0x55,0,0x49,0,0x55,0,0x49,0},
    {0x55,0,0x55,0,0x55,0,0x55,0},
    {0x55,0,0x55,0x20,0x55,0,0x55,2},
    {0x55,0x20,0x55,0x2,0x55,0x20,0x55,2},
    {0x55,0x20,0x55,0x88,0x55,2,0x55,0x88},
    {0x55,0x22,0x55,0x88,0x55,0x22,0x55,0x88},
    {0x55,0x49,0x55,0x88,0x55,0x49,0x55,0x88},
    {0x55,0x49,0x55,0x92,0x55,0x49,0x55,0x92},
```

```
    {0x55,0x49,0x55,0xaa,0x55,0x49,0x55,0xaa},
    {0x55,0xaa,0x55,0xaa,0x55,0xaa,0x55,0xaa},
    {0xaa,0xb6,0xaa,0x6d,0xaa,0xb6,0xaa,0x6d},
    {0xaa,0xb6,0xaa,0x77,0xaa,0xb6,0xaa,0x77},
    {0xaa,0xdd,0xaa,0x77,0xaa,0xdd,0xaa,0x77},
    {0xaa,0xdf,0xaa,0x77,0xaa,0xfd,0xaa,0x77},
    {0xaa,0xdf,0xaa,0xfd,0xaa,0xdf,0xaa,0xfd},
    {0xaa,0xff,0xaa,0xdf,0xaa,0xff,0xaa,0xfd},
    {0xaa,0xff,0xaa,0xff,0xaa,0xff,0xaa,0xff},
    {0xaa,0xff,0xb6,0xff,0xaa,0xff,0xb6,0xff},
    {0xaa,0xff,0xbb,0xff,0xaa,0xff,0xbb,0xff},
    {0xff,0xbb,0xff,0x6d,0xff,0xbb,0xff,0xd6},
    {0x77,0xff,0xdd,0xff,0x77,0xff,0xdd,0xff},
    {0xff,0xf7,0xff,0xdd,0xff,0x7f,0xff,0xdd},
    {0xff,0xbb,0xff,0xff,0xff,0xbb,0xff,0xff},
    {0xff,0xbf,0xff,0xff,0xff,0x1b,0xff,0xff},
    {0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff}};

main()
  {
    int driver=VGA;
    int mode=VGAHI;
    initgraph(&driver,&mode,'');
    setbkcolor(BLUE);
    setcolor(WHITE);
    setfillstyle(1,RED);
    bar3d(50,100,200,180,50,1);
    floodfill(140,90,WHITE);
    floodfill(220,120,WHITE);
    line(15,15,40,80);
    setcolor(RED);
    setfillstyle(1,DARKGRAY);
    sector(550,200,150,230,200,50);
    setfillstyle(1,GREEN);
    rectangle(450,400,500,450);
    floodfill(470,420,RED);
    setfillstyle(3,MAGENTA);
    bar(100,330,400,420);
    getch();
```

```
    read_grade();
    for(linel=0;linel<20;linel++)
      {convert_data(linel);
       set_print(640,1,prn_data);
      }
  }
void read_grade(void)
  {
   union REGS r;
   float *grade_p;
   int k,m,c,i,red,green,blue;
   grade_p=grade_value;
   for(k=0;k<=15;k++)
    {
     r.h.ah=0x10;
     r.h.al=7;
     r.h.bl=k;
     int86(0x10,&r,&r);              /*得到相应于K号的调色板寄存器中的值*/
     m=r.h.bh;
     r.h.ah=0x10;
     r.h.al=7;
     r.h.bl=0x10;
     int86(0x10,&r,&r);              /*读模式控制寄存器*/
     c=r.h.bh;
     i=c&0x80;
     r.h.ah=0x10;
     r.h.al=7;
     r.h.bl=0x14;
     int86(0x10,&r,&r);              /*读颜色选择寄存器*/
     c=r.h.bh;
     c<<=4;
     if(i!=0)
    c+=(m&0xf);
    else{
    c&=0xc0;
    m&=0x3f;
    c+=m;
     }
     r.h.ah=0x10;
```

```
    r.h.al=0x15;
    r.x.bx=c;
    int86(0x10,&r,&r);              /*读颜色寄存器*/
    red=r.h.dh;
    green=r.h.ch;
    blue=r.h.cl;
     *grade_p++=(0.3*(float)red+0.59*(float)green+
    0.11*(float)blue)/63.0;         /*得到相应颜色的灰度值*/
    }
  }
void convert_data(int line)
  {
    int i,j,colum,row,k;
    int color;
    float *grade_p;
    union REGS r;
    unsigned char grade,grade_1;
    for(i=0;i<=1920;i++)            /*清打印缓冲区*/
    prn_data[i]='\0';
    for(i=0;i<640;i++)
     {
    for(j=0;j<24;j++)
     {
      grade_p=grade_value;
      r.h.ah=0xd;
      r.h.bh=0;
      r.x.cx=i;
      r.x.dx=j+line*24;
      int86(0x10,&r,&r);            /*读象素点*/
      color=r.b.al;
      grade_p+=color;               /*得到相应点的灰度值*/
      grade=(unsigned char)(31.0**grade_p);    /*转换成相应灰度矩阵数组的
                                                 行号*/
      row=j;k=j;colum=i;
      row%=8;colum%=8;k/=8;
      grade_1=color_grade[grade][row];
      grade_1>>=colum;
      grade_1&=1;
      prn_data[i*3+k]+=(grade_1<<(7-row));
```

```
    }
    }
  }
int set_print(int col,int row,unsigned char *buf)
  {
   unsigned n1,n2,i,j,k;
   if(biosprint(2,0,0)==0x90)
    {
    biosprint(0,27,0);              /*设置打印行距*/
    biosprint(0,'3',0);
    biosprint(0,23,0);
    n1=col%256;n2=col/256;          /*打印一行*/
    for(k=0;k<row;k++)
     {
      biosprint(0,27,0);            /*设置打印机为图形方式*/
      biosprint(0,'*',0);
      biosprint(0,39,0);            /*设置三倍密度打印*/
      biosprint(0,n1,0);
      biosprint(0,n2,0);
      for(i=0;i<col;i++)
       {
        for(j=0;j<3;j++)
          biosprint(0,*(buf+k*3*col+3*i+j),0);
       }
        biosprint(0,13,0);          /*回车换行*/
        biosprint(0,27,0);
        biosprint(0,43,0);
        biosprint(0,10,0);
      }
    }
 return 0;
 }
```

13.7.5 用16级灰度抖动法打印图象例

该程序采用16个抖动矩阵来对应15种灰度的颜色，该程序比上一程序简化了许多，但实际得出的打印效果无多大差别，因控制打印机打印出不同的灰度来，并不是控制打印颜色的深浅(打印机进行打印时，颜色深度是无法用计算机进行控制的)，而是相当于用不同的图模代替颜色灰度，因打印针点太密，因而从效果上来看许多图模一打印出来，几乎无法区别，所以分的灰度级别再多，似无多大意义。

```
#include <bios.h>
#include <dos.h>
#include <stdio.h>
#include <graphics.h>
int line1;
int set_print(int col,int row,unsigned char *buf);
unsigned char prn_data[1920];
void convert_data(int line);
unsigned char color_grade[16][8]={{0,0,0,0,0,0,0,0},
    {0x88,0,0x22,0,0x88,0,0x22,0},
    {0x55,0,0x49,0,0x55,0,0x49,0},
    {0x55,0x20,0x55,0x2,0x55,0x20,0x55,2},
    {0x55,0x49,0x55,0x88,0x55,0x49,0x55,0x88},
    {0x55,0x49,0x55,0xaa,0x55,0x49,0x55,0xaa},
    {0x55,0xaa,0x55,0xaa,0x55,0xaa,0x55,0xaa},
    {0xaa,0xb6,0xaa,0x6d,0xaa,0xb6,0xaa,0x6d},
    {0xaa,0xb6,0xaa,0x77,0xaa,0xb6,0xaa,0x77},
    {0xaa,0xdf,0xaa,0xfd,0xaa,0xdf,0xaa,0xfd},
    {0xaa,0xff,0xaa,0xff,0xaa,0xff,0xaa,0xff},
    {0xaa,0xff,0xbb,0xff,0xaa,0xff,0xbb,0xff},
    {0xff,0xbb,0xff,0x6d,0xff,0xbb,0xff,0xd6},
    {0x77,0xff,0xdd,0xff,0x77,0xff,0xdd,0xff},
    {0xff,0xbf,0xff,0xff,0xff,0xfb,0xff,0xff},
    {0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff}};

main()
  {
    int driver=VGA;
    int mode=VGAHI;
    initgraph(&driver,&mode,'');
    setcolor(WHITE);
    setfillstyle(1,RED);
    bar3d(50,100,200,180,50,1);
    floodfill(140,90,WHITE);
    floodfill(220,120,WHITE);
    line(15,15,40,80);
    setcolor(RED);
    setfillstyle(1,DARKGRAY);
    sector(550,200,150,230,200,50);
```

```
    setfillstyle(1,GREEN);
    rectangle(450,400,500,450);
    floodfill(470,420,RED);
    setfillstyle(3,MAGENTA);
    bar(100,330,400,420);
    getch();
    for(linel=0;linel<20;linel++)
      {convert_data(linel);
       set_print(640,1,prn_data);
      }
  }
void convert_data(int line)
  {
    int i,j,colum,row,k;
    int color;
    union REGS r;
    unsigned char grade_1;
    for(i=0;i<=1920;i++)
    prn_data[i]='\0';
    for(i=0;i<640;i++)
     {
        for(j=0;j<24;j++)
         {
          r.h.ah=0xd;
          r.h.bh=0;
          r.x.cx=i;
          r.x.dx=j+line*24;
          int86(0x10,&r,&r);
          color=r.h.al;                    /*得到象素颜色*/
          row=j;k=j;colum=i;
          row%=8;colum%=8;k/=8;
          grade_1=color_grade[color][row];
          grade_1>>=colum;
          grade_1&=1;
          prn_data[i*3+k]+=(grade_1<<(7-row));
         }
     }
  }
int set_print(int col,int row,unsigned char *buf)
```

```
    {
      unsigned n1,n2,i,j,k;
      if(biosprint(2,0,0)==0x90)              /*打印机忙和联机否*/
       {
      biosprint(0,27,0);                      /*设置打印行间距*/
      biosprint(0,'3',0);
      biosprint(0,23,0);
      n1=col%256;n2=col/256;
      for(k=0;k<row;k++)
       {
        biosprint(0,27,0);                    /*设置打印机为图形模式*/
        biosprint(0,'*',0);
        biosprint(0,39,0);                    /*三倍密度打印*/
        biosprint(0,n1,0);
        biosprint(0,n2,0);
        for(i=0;i<col;i++)
         {
          for(j=0;j<3;j++)
            biosprint(0,*(buf+k*3*col+3*i+j),0);     /*打印数据*/
         }
          biosprint(0,13,0);
          biosprint(0,27,0);
          biosprint(0,43,0);
          biosprint(0,10,0);
        }
      }
  return 0;
}
```

第 14 章

C 程序中汉字显示技术

在许多应用软件设计中，经常需要用汉字进行提示和用于人机交互，因而一个面向不同层次用户的应用软件中将会用到许多汉字，以便进行显示，为此没有汉化的 Turbo C 集成开发环境，如何编制能显示汉字的程序，这是用户比较关心的问题，本章将讲述这方面的技术。

由于在中文操作系统下，屏幕显示将处于图形显示方式下，但是常用的一些输入和输出函数仍可使用，这些函数是 printf()，puts()，putchar()，scanf()，gets()，getche()等，由于这些函数在执行时，实际上进行了一些 BIOS 功能调用，如 printf()，putchar，puts()等则是进行 INT 10H 中断调用来显示字符，而中文 DOS 对 INT 10H 功能调用进行了保留并加以扩充，因而具有这些函数的 C 程序当然也可在中文 DOS 下运行，如何编制带有输入输出汉字的程序呢，下面进行介绍。

14.1 可在中文 DOS 下显示汉字的程序编制

一、在中文 DOS 下，运行中文编辑软件，如 WPS，CCED、中文 WS、中文 PE 等，编辑带有汉字的源程序(在 ASCII 码方式下输入程序，当有汉字的地方，用中文输入方式输入汉字)。注意在装有 TC 时不能用 WPS！因 WPS 程序较大，无法在它驻留内存的情况下，再调用 TC 集成开发环境(否则将出现程序太大，无法装入内存的错误信息：program too big to fit in memory!)，更主要的是它的文件格式也不符合。

当然使用 WPS 也很方便，因为大家比较熟悉，方法是将 C 源文件在 WPS 下进行编辑，即在有汉字的地方，加入汉字，当汉字加入完后，再将编辑过的文件用 WPS 的文本转换功能，把 WPS 格式转换为文本格式，然后存盘，此时重新引导系统，调入 TC 进行编译连接，生成执行程序。

二、在 TC 集成开发环境下编译连接带有汉字的源程序，从而生成执行程序。

三、在中文 DOS 下，运行带有汉字的执行程序，这样在显示汉字的地方，将会正确显示汉字。

当然上述过程，也可用下面过程代替，即：

一、在西文 DOS 下，调用 Turbo C，利用它的集成开发环境，编辑带有汉字的源程序，由于在此环境下，无法输入汉字，因此我们暂可用英文代替汉字，将编好的源程序编译连接，直到没有错误。

二、调用中文 DOS，并用上述的中文编辑软件对编好的源程序进行修改，凡要用汉字的地方将原来代替汉字的英文换成中文。

三、在西文 DOS 下，调 TC，在 TC 集成开发环境下编译连接已进行汉字替换的源程序，并生成可执行程序。

四、在中文 DOS 下，运行生成的执行程序。这样就可实现汉字的显示。

例如：

```
main()
  {
   char *a,*s;
   a="          ";
   s="请输入汉字";
   printf("%s\n",s);
   scanf("%s",a);
   printf("%s\n",a);
  }
```

当在中文 DOS 下运行该执行程序时，屏上出现中文提示"请输入汉字"，当用户输入汉字时，便马上又显示出来。

上述工作方式，仅在较早的汉字操作系统下，用于实现在 C 中汉字的显示，显然这种方法过于落后了，那种汉字操作系统，将西文的输入(ASCII 码方式)和中文的输入(汉字内码)截然分开。好在现在出的一些汉字系统已有重大改进，如超想全字符型汉字系统(6.0 版)，USDOS 3.0 以上版本，联想 DOS 2.13 I 以上版本，均支持中西文混合输入，当用户输入西文时，便是 ASCII 码输入，当是汉字输入时便转成机内码输入，这样当在这些汉字操作系统下，调入 Turbo C 的集成环境，那么编辑一个带汉字的 C 语言程序，输入可在集成环境下进行，不用如上面所说的，再转到汉字 DOS 下，输入汉字，然后回到西文 DOS 下进行 TC 环境下的编译连接，再转到中文 DOS 下运行。在这些新版本的中文 DOS 支持下，在 C 程序中使用汉字更加方便，若采用高版本 DOS(如 5.0 以上)，将 DOS 的一些部分放入高端内存，将使得有更多自由空间，因而编辑一个较大的使用汉字的应用 C 程序就成为可能。

例如使用超想全字符型汉字系统 6.0 版，可制作一个自动批处理文件，设名为 CXDOS.BAT，其内容如下：

```
echo off
cd cxdos
cxvrk%2
cxvga
py
cd..
```

在 DOS 下运行该自动文件时，将运行 cxdos(超想 DOS)，并调入超想的矢量汉字库，采用 VGA 显示驱动程序，并调入拼音汉字输入方式。这样机器就在上述环境的 cxdos 控制下了，这时再调入 TC，那么 Turbo C 集成环境就在 cxdos 支持下了，因而就可用上述的混合编程法来输入带汉字的 C 语言程序了，当要输入汉字时，打 Ctrl-Alt Q 使外挂输入变成汉字拼音输入，汉字输入完后，再用此开关回到外挂输入，用这种输入方法，可编辑较多汉字的 C

程序，当在 TC 下编完后，可按 F9 进行编译连接，执行文件可在该环境下运行，这就为编制带大量汉字的实用程序带来了方便。因而用户若有新版本的中文 DOS 或新推出的汉字系统，就可采用此法，而不必采用前述的老方法了。

在中文 DOS 下时，由于屏幕处于图形方式，故可以使用图形函数(但必须要对屏幕进行图形初始化设置)，有些文本方式下的输入输出函数也可使用，如前面已示例用过的 printf()，scanf()，除此之外 puts()，putchar()，gets()，getche()，也可以使用，也可用 window()函数定义一个窗口，用 gotoxy()函数对汉字输入输出定位，用 textbackground()设置背景色，textcolor()函数设置汉字的颜色，但有在窗口中输入输出的函数不能使用，gettext()和 puttext()函数也不能用。

为了能在设置窗口的情况下，也能使用窗口中的输出函数，如 cprintf()，cputs()等，必须对 Turbo C 头文件 conio.h 中的 directvideo 变量进行重新设置，在编译时，缺省情况下该变量取 true，它表示当调用 cprintf()，cputs()等输出函数显示输出的字符串时，控制显示器直接从显示存储器 VRAM 中取字符显示(可参阅 Turbo C 图形显示一章有关段)，由于在 VRAM 中取显示的西文字符与显示的汉字过程不同，故该函数不能用于显示汉字，但当变量 directvideo 取 false 值(0 值)时，这两个函数执行时，将通过 INT 10H 中断调用进行显示，而不直接通过 VRAM，因而可以在中文 DOS 下进行使用。

下面程序中定义了一个清屏函数 cls()，因若使用系统的清屏库函数 clrscr()，它将把屏幕显示方式置为文本方式，因而用 printf，cprintf 和 cputs 均不能显示汉字，故而自定义了一个清屏函数，使得调用它时，不会将图形方式改变为文本方式。

```
#include<dos.h>
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
void cls();
main()
{
   cls();
   directvideo=0;
   textattr(0x1e);
   printf("计算机\n");
   cprintf("计算机\n");
   textattr(0x16);
   cputs("计算机\n");
   getch();
}
void cls(void)
{
  union REGS regs;
  regs.h.ah=6;
```

```
    regs.h.al=0;
    regs.h.ch=0;
    regs.h.cl=0;
    regs.h.dh=24;
    regs.h.dl=79;
    regs.h.bh=7;
  int86(0x10,&regs,&regs);
  }
```

显示汉字最方便的方法是，在图形方式下，调用汉字字库中要显示汉字的字模，然后将其图象显示出来，这个字库，可以使用中文 DOS 中的汉字点阵字库，也可以自己建一个小字库，这样，显示汉字的程序不仅在中文状态下可以运行，在西文状态下也可运行，更主要的是一些 C 的库函数的使用将不会受影响。

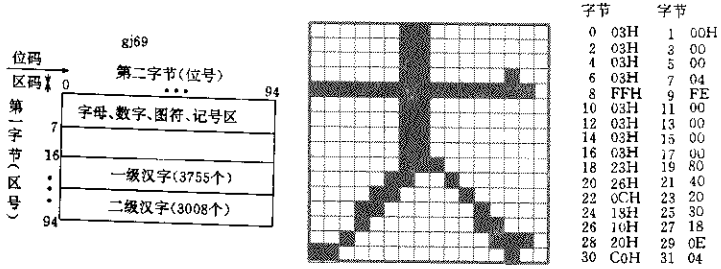
14.2 在西文 DOS 下 C 程序显示汉字技术

由于前面介绍的在中文状态下运行 C 程序，必须将汉字库装入内存，这样对大的应用程序有可能造成无法加载，为此下面的内容将介绍各种在西文状态下显示汉字的方法，而将中文 DOS 中的汉字库作为一个磁盘的二进制数据文件，用时进行打开，并找到相应汉字的字模，将其取出面显示，也可以为了专用和加快程序速度，建立自己的专用字库，它可作为程序的一部分，也可作为一个文件存在盘上，用时连接即可。

为了后面程序的易读，读者必须对有关汉字的一些有关字库的知识有所了解，为此进行简单的介绍：

14.2.1 国标汉字字符集与区位码

根据对汉字使用频繁程度的研究，可把汉字分成高频字（约 100 个），常用字（约 3000 个），次常用字（约 4000 个），罕见字（约 8000 个）和死字（约 45000 个），即正常使用的汉字达 15000 个。我国 1981 年公布了《通讯用汉字字符集（基本集）及其交换码标准》GB2312-80 方案，把高频字、常用字、和次常用字集合成汉字基本字符集（共 6763 个），在该集中按汉字使用的频度，又将其分为一级汉字 3755 个（按拼音排序）、二级汉字 3008 个（按部首排序），再加上西文字母、数字、图形符号等 700 个。国家标准的汉字字符集（GB2312-80）在汉字操作系统中是以汉字库的形式提供的。汉字库结构作了统一规定，如图 14.1(a)所示，即将字库分成 94 个区，每个区有 94 个汉字（以位作区别）每一个汉字在汉字库中有确定的区和位编号（用两个字节），这就是所谓的区位码（区位码的第一个字节表示区号，第二个字节表示位号，因面只要知道了区位码，就可知道该汉字在字库中的地址，每个汉字在字库中是以点阵字模形式存储的，如一般采用 16×16 点阵形式，每个点用一个二进位表示，存 1 的点，当显示时，可以在屏上显示一个亮点，存 0 的点，则在屏上不显示，这样把存某字的 16×16 点阵信息直接用来在显示器上按上述原则显示，则将出现对应的汉字，如一个大字的 16×16 点阵字模如图 14.1(h)所示，当用存储单元存储该字模信息时，将需 32 个字节地址，图 14.1(b)右边写出了该字模对应的字节值。



(a) 国标(GB2312-80)汉字字符集　　(b) "大"字的字模(1, 15)

图 14.1

如"大"的区位码为 2083,它表示该字字模在字符集的第 20 区的第 84 个位置。

14.2.2 汉字的内码

我们知道在计算机内英文字符是用一个字节的 ASCII 代码表示,该字节最高位一般用作奇偶校验,故实际是用 7 位码来代表 128 个字符的,但对于众多的汉字,只有用两个字节才能代表,这样用两个字节的代码表示一个汉字的代码体制,国家制定了统一标准,称为国标码。国标码规定,组成两字节代码的各字节的最高位均为 0,即每个字节仅只使用 7 位,这样在机器内使用时,由于英文的 ASCII 码也在使用,可能将国标码看成两个 ASCII 码,因而规定用国标码在机内表示汉字时,将每个字节的最高位置 1,以表示该码表示的是汉字,这些国标码两字节最高位加 1 后的代码称为机器内的汉字代码,简称内码,以"大"字为例:

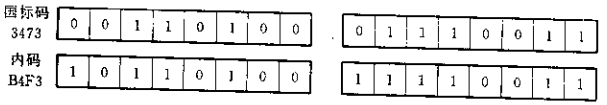


图 14.2 "大"的国标码与内码

国标码与区位码存在一种转换关系,即将 16 进制的区位码,两个字节各加 80H 后,即成国标码,后面将要介绍。

在 PC 机中,与汉字输入与输出有关的 ROM BIOS 中断调用程序有:

INT 10H 显示器驱动程序

INT 5H 屏幕打印程序

INT 16H 键盘输入程序

INT 17H 打印机输出驱动程序

由于它们是用于西文的,因而对汉字就不适用了,因而中文 DOS 将这些中断调用进行了改写,为了保留原来西文的上述程序,即保留原 ROM BIOS 系统,中文 DOS 将改写后的 BIOS (CCBIOS)以在内存驻留的形式存放在 PC 机的 RAM 中(当调用中文 DOS 或用中文 DOS

启动时，CCBIOS 即驻留内存)，这样机器在 MS-DOS 控制下时，使用 ROM BIOS，当在中文 DOS 控制下时，使用 CCBIOS，若在中文 DOS 下，而使用西文输入方式时，也仍使用 ROM BIOS，这样就使得 PC 机可以使用西文，也可使用汉字。

当用键盘进行汉字输入时，则由键盘管理模块将通过键盘输入的汉字输入码转换为机内码，从而再由内码转换成区位码找到汉字库的汉字，进行显示。

上面已提到了汉字在字库中的排列，按汉字在字库中所处的区和位的代码表示一个汉字的码称为区位码，由于区位码和内码存在着一种对应关系，因而知道了某汉字的内码，即可确定出对应的区位码，知道了区位码，就可找出该汉字字模在字库中存放的地址，由此地址调出该汉字的 32 个字节内容(字模)进行显示，就可显示出 16×16 点阵组成的该汉字。

由于在西文 DOS 下不能用本章开始叙述的方法直接运行能显示汉字的程序，若在中文 DOS 下用 TC 编制有汉字的程序，则许多 TC 的库函数又不能正常运行，因此可以在西文状态，对有汉字的 C 程序，读出汉字的内码(如同读字符的 ASCII 码一样)，将其转换成字库的区位码，然后直接到字库中去找出汉字字模，再用有关的位操作和循环语句，对每个字节的每一位进行判断，如同过滤一样，如果某位是 1，则按设置的颜色在屏幕的相应位置画点，若某位为 0，则不画点，这样就可按预先设置的颜色在相应位置显示出该汉字来。

14.2.3 内码转换为区位码与字模显示技术

由于在中文 DOS 下，输入汉字时，其相应的内码即已在程序中存在，(即内码已在原汉字的位置)，如同在西文 DOS 下，输入英文字符时，其对应的 ASCII 也在程序中存在一样，因而得知汉字的内码，将其转换为区位码，再从字库中找到对应的汉字，将其字模直接显示即可。

汉字内码与区位码有固定转换关系，即若汉字内码为十六进制数 aaff，则区号 qh 和位号 wh 分别为：

```
qh=aa-0xa0;
wh=ff-0xa0;
```

若用十进制表示内码为 c_1 c_2，则

```
qh=c1-160;
wh=c2-160;
```

即区位码 qw 为：

```
qw=100*(c1-160)+(c2-160);
```

反过来，若已经知道了区位码 qw，则也可求得区号和位号：

```
qh=qw/100;
wh=qw-100*qh;
```

因而该汉字在汉字库中离起点的偏移位置(以字节为单位)，可计算为：

```
offset=(94*(qh-1)+(wh-1))*32l;
```

注意：在上面的描述中，假设已在 C 程序中定义 int qh，wh，qw，long offset，字库中每个区有 94 个字符。

因此当定义了一个汉字库文件的代号如 hzk p，并用 open()函数打开字库(假设用 2.13H 的简体字库 HZK16)。

```
hzk_p=open("C:\\213\\HZK16",O_BINARY | O_RDONLY);
```

再用要求显示汉字在库中的偏移地址 offset 在库中找出对应的字模：即用

```
lseek(hzk_p,offset,SEEK_SET)
```

来定位文件指针确定读位置，然后用 read()函数，读出其字模(共 32 个字节)，即

```
read(hzk_p,bytes,32);
```

其中 bytes 已定义为—char bytes[32]数组的数组名。这样 bytes 数组中则存了要显示汉字的 16×16 点阵字模，然后将字模按行扫描的办法，(通过循环)用 putpixel()函数在屏幕设定位置显示出象点，因而组合成一个显示的汉字。

14.2.4 程序例

下面的程序就是用这种方法来显示汉字的，当通过 get_hz()函数取得汉字字模后，(在 mat[]中)用 dis hz()函数显示汉字时，用予先定义的一个数组 mask[]中的元素去和字模的相应位去相与，若结果为大于 0，则用 putpixel(x,y,color)，在 x,y 处用 color 颜色显示、若为 0，则不显示。MASK[]数组中存了一个字节的每位权值，这样用 mask[j%8]&mat[pos+j/8]即得到了第[pos+j/8]字节的第[j%8]位的值，因而用该值便可控制显示。

当然也可用多重循环的方法来判断 mat[]中各字节的各位是否为 1，如：

```
for(i=0;i<16;i++)                    /*字模的 16 个行*/
 for(j=0;j<2;j++)                    /*每个行的两个字节*/
   for(k=0;k<8;k++)                  /*每个字节的 8 位*/
     if(getbit(mat[i*2+j],7-k))
      putpixel(x,y,color);
          ⋮
int getbit(unsigned char c, int n)
      {
         return((c>>n)&1);           /*将字节的某位移到最低位并屏蔽其它 7 位*/
      }
```

在该程序中，用指针变量 s 作为要显示汉字串的地址指针，因而在取出汉字字模显示后，s 必须加 2(因一个汉字占两个字节)，以便再取第二个汉字，由于在设定的图形方式下，x 最大值可取 639，故当该行 x 值未到 640 时，并且未到汉字串尾时，继续在该行显示汉字，否则令 x=20，s+=2，在下一行开始显示汉字，这是由两重的 while 循环结构实现的。

在编辑源程序时，当在西文方式下用 TC 集成环境编完后，再回到中文 DOS 下，比如用中文 WS 进行再编辑，将 s 指向的字符串，用所需的汉字填充，填汉字时，汉字间的空格一定要用区位码或汉字输入方式下的图符输入方式选空格(为 1)，若用空格键输入，将读出的仍是前一个汉字，不会出现空格，这一点要注意；填完汉字的程序，再在西文的 TC 下进行编译，连接，这样生成的执行程序即可在西文状态下运行。

本程序中汉字库选用 213H 汉字系统下的 16*16 点阵汉字字库 hzk16，它存放在 C 盘 213 子目录下，故用 open 打开时，指明其路径为：

```
"C:\\213\\hzk16"
```

程序如下

```
#include<stdio.h>
#include<graphics.h>
#include<fcntl.h>
#include<io.h>
int hzk_p;
void open_hzk(void);
void get_hz(char incode[],char bytes[]);
void dishz(int x,int y,char cade[],int color);
main()
{
  int x=20;
  int y=100;
  char *s="春眠不觉晓  处处闻啼鸟  夜来风雨声  花落知多少";
  int driver=DETECT;
  int mode=0;
  initgraph(&driver,&mode,"c:\\tc");
  open_hzk();
  while(*s!=NULL)
   {
     while(x<640&&(*s!=NULL))   /*汉字串未到结尾和x未到640时循环*/
      {
       dishz(x,y,s,LIGHTRED);        /*用亮红显示一个汉字*/
       x+=20;                        /*列增加20*/
       s+=2;                         /*汉字字符字节地址加2*/
      }
     x=20;y+=20;                     /*若一行显示完,则右下行又开始显示*/
    }
   getch();
   close(hzk_p);
   closegraph();
   }
void open_hzk()
{
  hzk_p=open("c:\\213\\hzk16",O_BINARY | O_RDONLY);  /*以二进制方式
                                                       打开字库*/
  if(hzk_p==-1)
    {
     printf("The file HZK16 not exist! ENTER to system\n");
```

```
    getch();
    closegraph();
    exit(1);
   }
}
void get_hz(char incode[],char bytes[])
{
  unsigned char qh,wh;
  unsigned long offset;
  qh=incode[0]-0xa0;                          /*得到区号*/
  wh=incode[1]-0xa0;                          /*得到位号*/
  offset=(94*(qh-1)+(wh-1))*32L;              /*得到偏移位置*/
  lseek(hzk_p,offset,SEEK_SET);               /*移文件指针到要读取的汉字字模处*/
  read(hzk_p,bytes,32);                       /*读取32个字节的汉字字模*/
  }
void dishz(int x0,int y0,char code[],int color)
{
  unsigned char mask[]={0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};
                                              /*屏蔽字模每行各位的数组*/
  register int i,j,x,y,pos;
  char mat[32];
  get_hz(code,mat);
  y=y0;
  for(i=0;i<16;++i)
   {
    x=x0;
    pos=2*i;
    for(j=0;j<16;++j)
     {
      if((mask[j%8]&mat[pos+j/8])!=NULL)      /*屏蔽出汉字模的一个位(即一个
                                                点)*/
       putpixel(x,y,color);                   /*是1则显示,否则不显示该点*/
      ++x;
     }
   ++y;
  }
}
```

14.3 在西文 DOS 下，24×24 点阵汉字的显示与放大

上面已经讨论了在西文 DOS 下，显示 16×16 汉字点阵的方法和程序，但若用此种规模的点阵字体进行放大或打印，则明显地出现线条的锯齿状，字体不好看，因此用 16×16 点阵字体打印，效果不好，再加打印汉字和屏幕显示汉字，对字模的扫描方式不同，因而为了打印，中文 DOS 系统为此另外提供了 24×24 点阵的汉字打印字库，如 UCDOS2.0 提供了 24×24 点阵字模的宋体字库 CLIBS，和楷体字库 CLIBK 及仿宋字库 CLIBF、黑体字库 CLIBH 等，213 汉字系统下的 24×24 点阵汉字字库 hzk24 等。一个 24×24 点阵的汉字字模由 72 个字节组成，对应字模的各个字节在库中的排列顺序方式与 16×16 点阵不同，这是因为对于打印和显示，对字模各字节的扫描方式不同，由于显示扫描是对字模从左到右，从上到下进行，因而各字节序号是从左到右从上到下，依次为 0,1,2……31，而对打印机来说，由于打印头的 24 针是纵向排列，一次垂直打印 24 点，即三个字节，然后再打印第二列的 24 点，依次打 24 次，即完成一个 24×24 点阵汉字的打印，所以说它对字模的扫描方式为从左到右进行，不再有从上到下的扫描动作，因而对字模各字节的提取方式为纵向提取，即从上到下依次顺序排列为 0,1,2……31(对 16×16 点阵而言)，对 24×24 点阵，则从上到下为 0,1,2，然后第二列为 3,4,5……到最后一列为 69,70,71。如图 15.2 所示，关于打印的原理，请参阅本书关于屏幕打印的部分。

由于国标对汉字的内码是唯一的，当然区位码也是唯一的，它和汉字字模是何种点阵无关，所以使用 24×24 点阵的汉字，同使用 16×16 点阵汉字的方法完全一样，不同的只是对 24×24 点阵字库取一个字模时，要 72 个字节才能取得一个汉字字模，因而存放字模的数组要定为有 72 个数组元素，比如 char mat[72]，每取一个汉字字模，24×24 汉字库的字模指针要偏移 72 个字节，由内码转换为区位码的过程则同 16×16 点阵的一样。

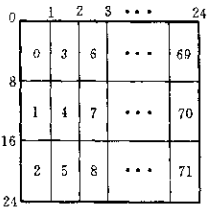


图 14.3 24×24 字模点阵的打印字节提取

最大的区别是：由于 24×24 点阵字库是专为打印而设计的，字节顺序是纵向排列，因而显示时，采用 putpixel()函数显示一个象点，则也是纵向排列，从上到下进行，即若用 i 代表纵列序号，j 代表纵向排列的 3 个字节序号，k 表示每个字节位号，当某位为 1 时，用定义的颜色画点，为 0 时不画，则可用下面的一个含有 for 三重循环的函数来实现

```
void dishz(int x0,int y0,char code[],int color)
{unsigned char mask[]={0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};
register int i,j,k;
char mat [72];
get_hz(code,mat);
for(i=0,i<24,i++)
```

```
    for(j=0;j<=2;j++)
      for(k=0;k<8;k++)
        {if((mask[k%8]&mat[3*i+j])!=NULL)
         putpixel(x0+i,y0+j*8+k,color);
        }
    }
```

该函数中 get_hz(code,mat)是由内码得到字模的函数,和 16×16 点阵汉字的存取过程基本相同,只不过将 32 字节换成 72 字节就可以了。

程序例:

下面是一个使用 USDOS 中的 24×24 点阵仿宋字库 clibs.dot 并显示汉字的程序,其过程同上面显示 16×16 点阵汉字一样,不同的是该程序可将显示的汉字放大一倍,即纵向和横向(放大倍数分别用 N,M 表示)同放大一倍,实现的方法是每个点纵向、横向再重复显示一次,即放大了一倍。

```
#include<stdio.h>
#include<graphics.h>
#include<fcntl.h>
#include<io.h>
#define M 2
#define N 2
int hzk_p;
void open_hzk(void);
void get_hz(char incode[],char hytes[]);
void dishz(int x,int y,char cade[],int color);
main()
{
  int x=10;
  int y=100;
  register i;
  char *s="白日依山尽 黄河入海流 欲穷千里目 更上一层楼";
  int driver=DETECT;
  int mode=0;
  initgraph(&driver,&mode,"c:\\tc");
  open_hzk();
  while(*s!=NULL)
   {
    while((640-x)>(24*M)&&*s!=NULL)
     {
      dishz(x,y,s,LIGHTRED);
```

```
        x+=24*M;
        s+=2;
        }
      x=10;y+=32*N;
      }
     getch();
     close(hzk_p);
     closegraph();
}
void open_hzk()
{
  hzk_p=open("c:\\puc\\clibs.dot",O_BINARY|O_RDONLY);
  if(hzk_p==-1)
     {
      printf("The file clibs.dot not exist! ENTER to system\n");
      getch();
      closegraph();
      exit(1);
      }
}
void get_hz(char incode[],char bytes[])
{
  unsigned char qh,wh;
  unsigned long offset;
  qh=incode[0]-0xa0;
  wh=incode[1]-0xa0;
  offset=(94*(qh-1)+(wh-1))*72L;
  lseek(hzk_p,offset,SEEK_SET);
  read(hzk_p,bytes,72);
}
void dishz(int x0,int y0,char code[],int color)
{  unsigned char mask[]
       ={0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};        /*屏蔽字模的数组*/
  register int i,j,k,q,r;
  char mat[72];
  get_hz(code,mat);
  for(i=0;i<24*M;i=i+M)
   for(q=0;q<M;q++)
     for(j=0;j<=2;j++)
```

```
    for(k=0;k<8;k++)
    {
     if((mask[k%8]&mat[3*i/M+j])!=NULL)
      for(r=0;r<N;r++)
       putpixel(x0+i+M,y0+j*N*8+k*N+r,color);
    }
}
```

14.4 用直接写显示存储器 VRAM 的方法显示汉字

用显示象点的函数 putpixel()调用来显示汉字字模点阵，或采用 BIOS 显示象点的功能调用来显示汉字点阵，显示速度都较慢，还有一种显示速度最快的方法，就是直接写屏法，实际上是将要显示的汉字点阵信息在图形方式下直接存入显示存储器 VRAM 中，于是马上在屏上显示出汉字来。下面简要介绍其原理，更详细内容可参阅有关书籍（如本人写的 IBM PC 286，386 汇编语言与外设编程一书，和本书的 Turbo C 作图一章）。

EGA 和 VGA 有 256 个字节的显示存储器(VRAM)(EGA 可有 64K 字节，128K 字节和 256K 字节三种配置)，256K 字节 VRAM 被分成独立的 4 个 64K 字节部分，称为颜色位面。这 4 个颜色位面占有同一个系统存储空间，即 64K 地址，这个问题可以这样来理解，即将 VRAM 想像成一个三维结构，一个位面压在另一个位面上，相互重叠，如图 14.4 所示，这些位面依次称为 0，1，2，3 位面，因此对每个地址(共 64K 个地址)，实际代表 4 个并列的字节，这样位就用 4 个位来表示，这些位是 4 个位面同一位置的位，若用 4 个位的不同组合来代表某种颜色，则可决定 16 种颜色(2^4)，实际上这 4 个位面上的同一位 4 个不同值的组合，代表了对调色板寄存器的索引，而调色板寄存器寄存了 16 种显示颜色，如图 14.5 所示。因此用在字节中所处的位和该字节的地址来决定该象素在屏上的位置，而用该位的 4 个位值来表示该象素的颜色，这样就可较容易地编程实现画图。从图 14.4 中可看出第一个象素的颜色位面值是 1101，EGA 和 VGA 工作在 12H 模式时，VRAM 便是这种结构，这样，一个地址表示依次的 8 个象素，即一个字节的每一位代表一个象素，而该位对应的 4 个位值代表

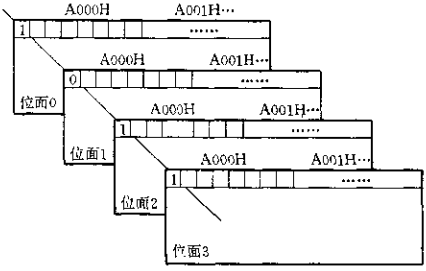


图 14.4 VRAM 的位面结构

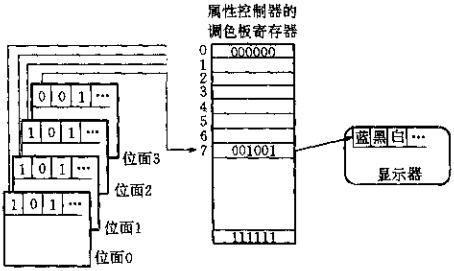


图 14.5　典型的图形方式

素的颜色。这样，通过一次写操作，可同时改变(或有选择地)8 个象素的颜色，因而可实现快速地填色或画图。位面具体使用方法因工作模式不同而异，所以我们只要将要显示的象素色值(即调色板颜色的索引值)写入相应位置的 4 个位面中(分成位面 0，位面 1，位面 2，位面 3)，即可立即显示出该象点来，(如若 4 个位面同一地址的相应位均为 0，即 0000，则象点显示黑色(即背景色)，如均为 1，即 1111，则显示白色，依次类推)，关于 EGA 与 VGA 显示象素过程可参阅本书第 10 章 10.1 节。

显示存储器 VRAM 中的字节地址和显示器屏上显示象素的位置 x，y 有对应的关系，即屏上每行可显示 640 个象素，对应 VRAM 的 80 个字节内容(每个位面的同一个位代表一个象点，故一个字节对应 8 个象点，即 4 个位面的同一字节地址代表要显示的 8 个象点，这样知道了在屏上象点的显示位置(x，y)，则在 VRAM 中的字节偏移地址为(即距 A0000H 的偏移)：

$$80 * y + \mathrm{int}(x/8)$$

$80 * y$ 表示象点所在行的偏移值，$\mathrm{int}(x/8)$表示象点离该行起点的字节偏移。若将一个 16 * 16 的汉字字模存入 VRAM 的 32 个字节地址中，其字节地址排序如图 14.6 所示：

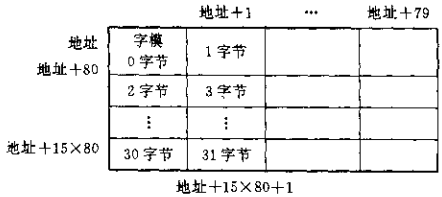


图 14.6　16×16 点阵字模在 VRAM 中的地址排列

这样字模 0 字节的最高位 D_7 对应于屏上显示的左边象点，依次排列，字节的 D_0 位对应于显示最右边的象点，结果将在屏上显示一个 16 * 16 的汉字，若沿行方向顺序按第一个字模装入模式装入 40 个汉字字模，则在 16 个扫描行上显示 40 个汉字(若地址取 A0000H，则显示在屏上的头 16 个扫描行上)。

14.4.1 将汉字字模装入 VRAM 的方法一

将显示的汉字字模按照上述地址分配装入 VRAM，有多种方法。其中一种是通过 EGA/VGA 适配器上的图形控制器的两个寄存器，即方式寄存器(索引号为 5)和位屏蔽寄存器(索引号为 8)，因为图形控制器中共有 9 个内部寄存器，为了减少 I/O 口地址数量，它们共用 1 个口地址，这些口地址可重复使用。为此，又设立了一个索引寄存器(其口地址为 3CEH)，9 个内部寄存器分别有自己的索引号，它们共用一个口地址(3CFH)。当给索引寄存器写入一个索引号后，其后对 3CFH 口地址的读或写操作就是对由索引号指出的那个寄存器进行的操作，这就实现了一个口地址的复用。索引寄存器口地址为 3CEH，9 个寄存器的口地址为 3CFH。

这些寄存器的名称和索引号如下：

索引号	寄存器名称
00	设置/清除寄存器
01	设置/清除允许寄存器
02	颜色比较寄存器
03	数据旋转移动与功能选择寄存器
04	读颜色位面选择寄存器
05	方式寄存器
06	混合寄存器
07	颜色无关寄存器
08	位屏蔽寄存器

我们将用到其中的两个寄存器，即方式寄存器和位屏蔽寄存器，其作用简要说明如下：

方式寄存器(索引号为 5)

方式寄存器提供了 3 种写数据到 VRAM 的方法和 2 种读数据到处理器的方法，它们分别由位 1、位 0 和位 3 决定，我们首先来看写方式位。

D1 D0 写方式

0 1 用锁存器的内容写

1 0 用位屏蔽寄存器来设置相应的位，将处理器写数据的 n 位写到相应的颜色位面 n(0—3)去，即用第 n 位填充位面 n

1 1 不用

下面介绍读方式位：

D3 读方式位。

0 当处理器读时，读出的一个字节内容对应于由颜色位面选择寄存器指出的颜色位面一个字节值。

1 读出的结果是颜色位面和颜色比较寄存器值比较的结果，当使用颜色比较寄存器时，必须使该位为 1。

D4 奇偶方式位。

在文本方式时，必须设置该位为 1，以便偶地址和偶数位面对应，奇地址和奇数位面对应。

D5 移位寄存器方式位。

用于 CGA 模式 4 和 5，当该位面为 1 时，操作数被串行移位输出，偶位被写入偶数位面，奇位被

写入奇数位面，以形成 4 种颜色的图形。

D6　256 种颜色方式位(仅 VGA 有)。

当为 1 时，按照 VGA 256 种颜色模式修改 VGA 属性控制器的操作。

D7　不用。

BIOS 内定写方式为直接处理器写(写方式 0)和读方式 0(D3=0)。

在本节介绍的这种方法中，我们采用写方式 2，即用位屏蔽寄存器来设置相应的位，将要写的 n 位(0～3)数据，通过 CPU 处理器写到相应的 VRAM 的位面去，第 0 位写到 0 位面，第 1 位写到 1 位面，依此类推，如图 14.7 所示：

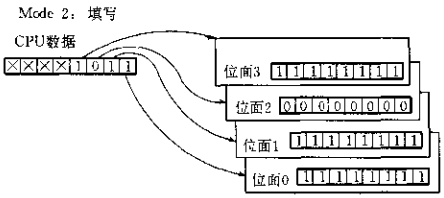


图 14.7　写方式 2

位屏蔽寄存器(索引号为 8)

该寄存器用来对处理器写入 VRAM 各位面的数据位进行屏蔽，若某位为 1，则允许处理器的数据写入到颜色位面，否则写入的是锁存器相应的数(即该位不改变)。它们规定是 D7 位用于第 7 位的屏蔽，D6 位用于第 6 位的屏蔽，依次类推，图 14.8 示出了这种屏蔽操作。

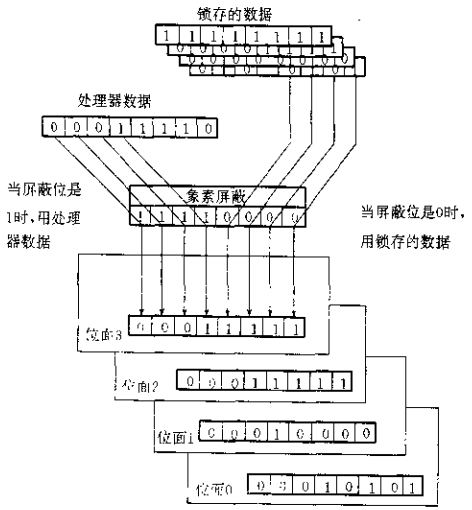


图 14.8　位屏蔽寄存器操作

因此当选择写方式 2 时，可如下编程：

```
outportb(0x3c,0x05);  /*选择方式寄存器*/
outportb(0x3e,0x02);  /*采用写方式 2*/
```

对于字模的写入，是将要显示的字模的一个字节信息读入位屏蔽寄存器，再将要显示的颜色值由 CPU 送入 VRAM 相应的字节地址(即相同地址的 4 个位面)，由于是模式 2 写入，故位屏蔽寄存器是 0 的位(即字模 1 个字节中是 0 的位)相应的各位面的位将不能写入颜色值，这样 VRAM 中一个字节地址(4 个位面同一地址)中对应字模是 1 的位，便写入颜色值，是 0 的位则被屏蔽而不能写入颜色值，而仍为原来的背景颜色。若要重新设置背景色，则只要将字模的字节信息取反，再写入位屏蔽寄存器，这样再将设置的背景颜色值写入 VRAM 的相应字节地址中，因而原来字模字节中是 0 的位，将被写入背景色，原来是 1 的位，将被屏蔽，而不能写入背景色，即保持原来的颜色值，通过这样操作，因而立即出现用设置的颜色和背景色显示的汉字来。

编程如下：

```
outportb(0x3ce,0x08);  /*选择位屏蔽寄存器*/
outportb(0x3cf,bytes[2*i+j])  /*将第 i 行第 j 个字节的字模写入位屏蔽寄存器*/
*(p+80*i+j)=color;  /*颜色值写入 VRAM*/
```

这样一个字节地址中，字模点阵中一个字节中是 1 的位，对应的 VRAM 各位面相应位写入颜色值，否则保持原来的背景值，若要设置自己的背景值，可如下编程：

```
outportb(0x3ce,0x08);
outportb(0x3cf,~bytes[2*i+j]);  /*字模取反写入位屏蔽寄存器*/
*(p+80*i+j)=bkcolor;/*VRAM 中写入背景色*/
```

若将上述过程写成一个函数形式，清单如下：

```
disph2(int x, int y, unsigned char * bytes, int color, int bkcolor)
    { char far *;
      int i,j,k;
      p=(char far *(0xa0000000+80*y+x/8);
      for (i=0;i<16;i++)
         for(j=0;j<2;j++)
         {outportb(0x3ce,0x05);
          outportb(0x3cf,0x02);
          outportb(0x3ce,0x08);
          outportb(0x3cf,bytes[2*i+j]);
          *(p+80*i+j)=color;
          k=*(p+80*i+j);
          outportb(0x3ce,0x08);
          outportb(0x3cf,~bytes[2*i+j]);
          *(p+80*i+j)=bkcolor;
```

```
    }
    outport(0x3ce,0x0005);      /*恢复为缺省的写方式 0*/
    outport(0x3ce,0xff08);      /*位屏蔽寄存器不再屏蔽*/
```

14.4.2 程序例

下面是直接显示汉字的程序例：

在该程序中，整个程序的结构同上一程序，只是在显示字模 x 位置增量时要注意，因在决定显示象点列位置时，列由 x/8 来决定，这样当 x 增量后，不能被 8 整除时，将有舍去余数的操作，因而将会出现显示的汉字不等距现象发生，故在 main()函数中采用 x+=24。

在 dishz 函数中，取 p 指针为一远程指针：

$P=(char\ far\ *)(0xa0000000+80*y_0+x_0/8);$

它决定了开始计算 8 个象点字节地址时，其对应在 VRAM 中的参考地址，当(P+80*i+j)时，该地址才是字模相应 8 个象点对应的绝对地址。

```
#include<stdio.h>
#include<graphics.h>
#include<fcntl.h>
#include<io.h>
int hzk_p;
void open_hzk(void);
void get_hz(char incode[],char bytes[]);
void dishz(int x,int y,char hz_mat[],int color,int bkcolor);
main()
{
  int x=20;
  int y=100;
  char mat[32];
  char *s="白日依山尽  黄河入海流  欲穷千里目  更上一层楼";
  int driver=DETECT;
  int mode=0;
  initgraph(&driver,&mode,"c:\\tc");
  open_hzk();
  while(*s!=NULL)
   {
    while(x<640&&(*s!=NULL))
     {
      get_hz(s,mat);
      dishz(x,y,mat,LIGHTRED,BLUE);
      x+=24;
```

```
        s+=2;
        }
      x=24;y+=20;
     }
   getch();
   close(hzk_p);
   closegraph();
   }
void open_hzk()
{
  hzk  p=open("c:\\213\\hzk16",O_BINARY | O_RDONLY);
   if(hzk_p==-1)
     {
      printf("The file HZK16 not exist! ENTER to system\n");
      getch();
      closegraph();
      exit(1);
     }
}
void get_hz(char incode[],char hytes[])
{
   unsigned char qh,wh;
   unsigned long offset;
   qh=incode[0]-0xa0;
   wh=incode[1]-0xa0;
   offset=(94*(qh-1)+(wh-1))*32L;
   lseek(hzk_p,offset,SEEK_SET);
   read(hzk_p,bytes,32);
    }
void dishz(int x0,int y0,unsigned char hz_mat[],int color,int bkcolor)
   { char far *p;
   register int i,j,k;  char mat[72];
   p=(char far *)(0xa0000000+80*y0+x0/8);
   for(i=0;i<16;++i)
    {
     for(j=0;j<2;++j)
      {
       outportb(0x3ce,0x05);
       outportb(0x3cf,0x02);
```

```
            ou:portb(0x3ce,0x08);
            ou:portb(0x3cf,hz_mat[2*i+j]);
            *(p+80*i+j)=color;
            k=*(p+80*i+j);
            outporth(0x3ce,0x08);
            outportb(0x3cf,~hz_mat[2*i+j]);
            *(p+80*i+j)=bkcolor;
          }
        }
            outportb(0x3ce,0x05);
            outportb(0x3ce,0xff08);
}
```

14.4.3 将汉字字模装入 VRAM 的方法二

直接写屏的第二种方法是利用 EGA/VGA 图形适配器结构中的定序器。

定序器控制所有 EGA 功能的时序，完成对存储器地址的译码，VRAM 或图形控制器到属性控制器的数据流均由它控制。它的控制是通过 5 个寄存器来进行的。这 5 个寄存器在 EGA 中只能写，在 VGA 中则允许读，它们使用 2 个 I/O 口地址(3C4H 和 3C5H)，3C4H 作为索引寄存器的口地址，用于选择要编程的寄存器，3C5H 则用于要选择编程的寄存器，表 14.1 列出了这些寄存器。

表 14.1 定序器的寄存器

索引号(索引寄存器口地址为 3C4H)	定序器寄存器(口地址为 3C5H)
0	复位
1	时钟模式
2	颜色位面写允许
3	字符发生器选择
4	存储器模式

这些寄存器中，汉字显示中用到的颜色位面写允许寄存器介绍如下：

颜色位面写允许寄存器(索引号为 2)

在画图操作中，这是一个很重要的寄存器，它控制处理器，使之允许或禁止写到指定的颜色位面。当该寄存器某位为 1 时(D_3—D_0)，该位对应的颜色位面允许被修改(写)，若是 0，则禁止写，各位与颜色位面对应关系是 D_3 若为 1，则允许颜色位面 3 可写，同理，D_2 对应颜色位面 2，D_1 对应颜色位面 1，D_0 对应颜色位面 0，D_7～D_4 不用，为 0。

颜色位面写允许寄存器的作用如图 14.9 所示，图中 D_1，D_0 为 0，即禁止对颜色位面 0 和 1 写，故它们保持原来的字节值，而由于 D_3、D_2 为 1，即允许对颜色位面 3 和 2 进行写，这样，在处理器对颜色位面写操作时，位面 3 和 2 相应字节就变成处理器写入的数了。

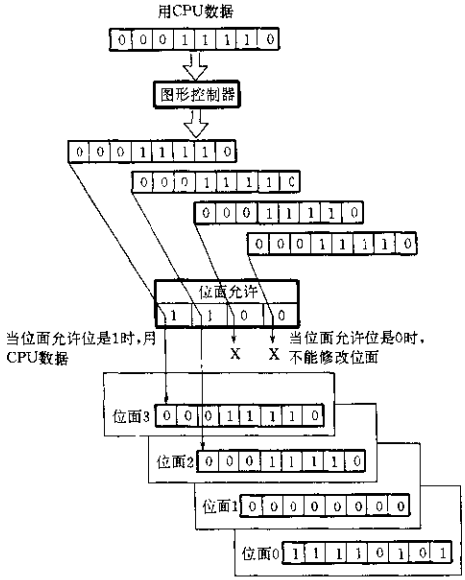


图 14.9 颜色位面写允许操作

编程如下：

```
outportb(0x3c4,0x2);          /*选择颜色位面写允许寄存器*/
outportb(0x3c5,color);        /*将颜色值写入该寄存器*/
*(p+80*i+j)=betes[2*i+j];/*将字模第i行j列字节信息写到VRAM*/
```

当执行上述程序运行时，将要显示的颜色值写入了颜色位面写允许寄存器，这样当欲显示汉字字模的第 i 行的第 j 个字节(j 为 0 或 1)时，写入 VRAM 的相应字节地址，由于受颜色位面写允许寄存器的控制，即字模该字节中是 1 的位，受颜色位面写允许寄存器控制，相当于对应的颜色位面写入了颜色值，而字模中字节为 0 的位，不能写入颜色值(仍为背景色值)，这样在写的过程中，汉字便用相应设置的颜色显示出来了，下面是用此种方法显示汉字的子程序。

```
disphz(unsigned char *bytes,int x, int y, int color);
  {
  char far *p;
  int i,j,k;
  p=(char far *(0xa0000000+80*y+x/8);
  for(i=0;i<16;i++)
    for(j=0; j<2; j++)
```

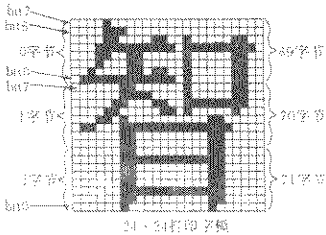
```
    {outportb(0x3c4,0x02);
     outportb(0x3c5,color);
      *(p+80*i+j)=betes[2*i+j];
    }
  outport(0x3c4,0xff02);
}
```

用该种方法写屏的程序，请看 14.4 中程序例。

14.4.4 24×24 打印字模转换为 24×24 点阵显示字模

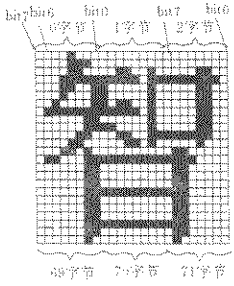
当使用 24×24 点阵打印字库进行汉字直接屏幕显示时，关键是将打印字模进行转换，即转变为按显示顺序排列的字模点阵。此时字节按横向排列，也就是将打印字模中的 0,3,6,9,12,15,18,21 各字节中的最高位 bit7 取出，依次变成一个新字节的 bit7 位、bit 6 位、bit5、bit 4、bit3、…bit0 位，这样就组成了一个新转置字模的第 0 字节，将 24,27,30…45 字节的 bit7 位取出，依次变成显示点阵的第一字节的 bit7,bit6…bit0 位，将 48,51,……69 字节的 bit7 位取出，依次变成显示点阵的第二字节的 bit7、bit6…bit0 位，然后又将打印字模中的 0,3,6,9…21 字节中的 bit6 位取出，依次变为显示点阵的第三字节的 bit7、bit6…bit0 位，将 24、27、30…45 字节的 bit6 位取出，组成第 4 字节的 bit7,bit6…bit0 位，依次类推，通过这样转置便又生成按横向排列的 24×24 点阵的汉字显示字模，其打印字模与显示字模字节排列关系如图 14.10 所示(该字模以智字为例按上述转置原则，将 24×24 点阵打印字模转换成 24×24 点阵显示字模。可用下面的函数来实现，该函数中字符数组 msk[]，用来屏蔽出某字节的 bit7 位，bit6 位…bit0 位，在该函数中 m[j*3+i]为按纵向字节排列的 24×24 点阵的打印字模，其中 j 表示打印字模点阵的列，i 为排列的字节序号，l 表示纵向排列字节的位号(0～7)，n[x]中则为转置后字节按横向排列的显示字模。该函数用三重 for 循环完成，其中里层两重循环实现了将打印字模的一排字节(有 24 个)的同一位取出，即通过屏蔽 m[j*3+i]&msk[l];，然后通过对不同字节同一位的左移，不移或右移操作，组成一个新的显示字节，如此循环 8 次，即 l 从 0 变到 7，则将一排字节的 8 位变成了 24 个字节，位序经过重新排列的(横向排列)显示字模，再如此循环三次(i 从 0 到 2)，则将 72 个字节的打印字模全转换成了显示字模。函数如下：

```
void get_hz(cbar m[],char n[])
 {
  unsigned char msk[]={0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};
  unsigned cbar a,b=0;
  int i,j,k,l,x;
  x=k=0;
  for (i=0;i<=2;i++)
   for(l=0;l<8;l++)
     for (j=0;j<24;j++)
      {
       a=m[j*3+i]&msk[l];
```



24×24打印字模

(a) 智字的24×24打印字模



24×24显示字模

(b) 智字的显示字模

图 14.10 智字的字模转换

```
        if(k-1>0)
         a=a>>k-1;
         if(k-1<0)
          a=a<<1-k;
         b=b+a;
         k++;
         if(((j+1)%8)==0)
         {n[x++]=b;
          k=b=0;
         }
    }
}
```

14.4.5 程序例

该程序就是采用上法，将 24×24 点阵的打印字模转换为 24×24 点阵的显示字模，并用上面介绍的第二种直接写屏的方法，通过 EGA/VGA 适配器中的颜色位面写允许寄存器，将转置成 24×24 点阵的显示字模 dis_mat[]通过颜色位面写允许寄存器的控制放入

VRAM 相应的地址中，即颜色位面写允许寄存器被设置成 color 颜色值，当显示点阵某位为 1 时，该颜色值便被写入相应位面的位中，并在屏上相应位置显示出来，某位为 0 时，则写入 0 不显示，这样用两重循环，即可很快地在屏上显示出该汉字来。

```
#include<stdio.h>
#include<graphics.h>
#include<fcntl.h>
#include<io.h>
int hzk_p;
void open_hzk(void);
void get_prhz(char incode[],char bytes[]);
void get_hz(char m[],char n[]);
void dishz(int x,int y,char cade[],int color);
main()
{
  int x=20;
  int y=100;
  register i;
  char *s="白日依山尽 黄河入海流 欲穷千里目 更上一层楼";
  int driver=DETECT;
  int mode=0;
  initgraph(&driver,&mode,"c:\\tc");
  open_hzk();
  while(*s!=NULL)
   {
     while(x<640&&*s!=NULL)
       {
        dishz(x,y,s,LIGHTRED);
        x+=24;
        s+=2;
       }
     y+=24;x=24;
    }
   close(hzk_p);
   getch();
   closegraph();
}
void open_hzk()
{
```

```
  hzk_p=open("c:\\puc\\clibs.dot",O_BINARY | O_RDONLY);
  if(hzk_p==-1)
    {
     printf("The file clib.dot not exit! ENTER to system\n");
     getch();
     closegraph();
     exit(1);
    }
}
void get_prhz(char incode[],char bytes[])
{
  unsigned char qh,wh;
  unsigned long offset;
  qh=incode[0]-0xa0;
  wh=incode[1]-0xa0;
  offset=(94*(qh-1)+(wh-1))*72L;
  lseek(hzk_p,offset,SEEK_SET);
  read(hzk_p,bytes,72);
}

void dishz(int x0,int y0,char code[],int color)
{

  register int i,j;
  char far *p;
  char pr_mat[72];

  char dis_mat[72];

  p=(char far *)(0xa0000000+80*y0+x0/8);
  get_prhz(code,pr_mat);
  get_hz(pr_mat,dis_mat);
  for(i=0;i<=2;i++)
   for(j=0;j<24;j++)
   { outportb(0x3c4,0x02);
     outportb(0x3c5,color);
     *(p+80*j+i)=dis_mat[3*j+i];
    }
}
```

```
void get_hz(char m[],char n[])
{
  unsigned char msk[]={0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};
  unsigned char a,h=0;
  int i,j,k,l,x;
  x=k=0;
  for (i=0;i<=2;i++)
   for(l=0;l<8;l++)
     for (j=0;j<24;j++)
       {
        a=m[j*3+i]&msk[l];
        if(k-l>0)
       a=a>>k-l;
        if(k-l<0)
       a=a<<l-k;
        b=b+a;
        k++;
        if(((j+1)%8)==0)
        {n[x++]=b;
       k=b=0;
        }
      }
}
```

14.5 汉字的任意倍数放大

在应用程序设计中，为了得到一个醒目的人机界面，常常需要将部分汉字按不同倍数放大进行显示，下面的一个自定义函数 fdhz()，可以按用户要求的放大倍数，对 2.13H 汉字系统中的 24×24 点阵字库中取出的汉字作横向或纵向放大，并可设置字间距离(以象素为单位)，也可水平或垂直显示，汉字字体可选 2.13H 字库中的 hzk24f(仿宋字库)，hzk24s(宋体字库)，hzk24k(楷体字库)和 hzk24h(黑体字库)，本函数在中西文状态下均可正确运行。

在该函数中，x0，y0 为开始显示汉字的坐标，mx，my 为横向、纵向放大汉字的倍数，dis 为字间距，h_vdir 为水平或垂直显示标志，当为 0 时，表示水平显示，当为 1 时，为垂直显示，color 为显示汉字的颜色，s 为要显示汉字串的指针，lih name 为要调用的 2.13H 汉字系统中的汉字库名，如上所述，lib_name 可以是各种字体的 24×24 点阵字库。上面提到的这些参数也就是 fdhz()函数中的形参，因而若调用该函数，则调用形式为：

```
fdhz(int x0,int y0,int mx, int my, int dis,
     int h_vdir, int color, char *s, char *lib_name);
```

在该函数中，首先以只读方式打开用户指定的二进制文件——24 * 24 点阵的字库，然后根据要显示的汉字串个数开辟一个缓冲区 buffer，将要显示的汉字串字模从指定的字库中一一取出，并放入缓冲区中，它用 while((temp= * s++)! =0)循环来实现，由于读出的汉字内码为两字节的，当标志 id=0 时，表示读入第一个字节，即 byte1=temp；它对应于区码部分，当 id=1 时，表示读第二字节，即 byte2=temp，它对应于位码部分，由此可以算出该汉字内码所对应的区位号和在字库的偏移地址 offset，从而得到字模，如此循环，直到所有要显示的汉字字模取出为止。

下面的 while(total--)循环，则根据放大倍数和字间距及是水平或垂直来显示汉字，其中显示字模的各象点是用将对应各字节的位均左移到最高位，然后用 mask=0x80 进行屏蔽，从而得知为 0 还是 1，若是 1 用 putpixel()函数用所设 color 进行显示该点，其中 switch()语句则根据是水平显示还是垂直显示，从而调整方向，进行汉字的显示。

最后释放缓冲区并关闭打开的字库。

该程序中，main()函数用来演示该函数的调用，它将按设定的方向和放大倍数及颜色，在给定的起始位置显示计算机三个字。

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<stdlib.h>
#include<string.h>
#define SIZE 72
void fdhz(int x0,int y0,int mx,int my,int dis,
        int h_vdir,int color,char *s,char *lib_name)
  {
    int x,y;
    FILE *fp;
    unsigned long offset;
    unsigned mxx,myy;
    unsigned int qh,wh;
    unsigned char *buf,*buffer;
    unsigned mask=0x80,id=0;
    unsigned count=0,total=0;
    unsigned byte1,byte2,temp;
    if(h_vdir)h_vdir=1;
    mxx=getmaxx();
    myy=getmaxy();
    if(! (fp=fopen(lib_name,"r+b")))printf("The file not exist");exit(1);}
      buf=buffer=(unsigned char *)malloc(strlen(s)*36+1);
    while((temp= *s++)! =0)      /*将放大的汉字串的字模一一取出放入缓冲区*/
     {
```

```
  if(temp>0xa0)
   {
  if(! id)
   {
     id=1;byte1=temp;
     continue;
     }
     else{
      id=0;
      byte2=temp;
      total++;
      }
   qh=byte1-0xa0;
   wh=byte2-0xa0;
   if(qh>15)qh=8;
    else if(qh=9)qh=6;
   offset=(long)((qh-1)*94+wh-1-658)*SIZE;
   if(fseek(fp,offset,SEEK_SET)==NULL)
   if(fread(buf,1,SIZE,fp)! =SIZE){printf("read error!");exit(1);}
   buf+=SIZE;
   id=0;
 }
}
 while(total--)      /*汉字字模的各字节的各位分离并放大显示*/
  {
   int i,j,k,xt,yt,bit;
   x=x0;
   y=y0;
   for(i=0;i<SIZE;i+=3,count+=3)
    for(xt=0;xt<mx;xt++)
    {
    for(k=0;k<3;k++,count++)
    {
      for(j=0;j<8;j++)
      {
       bit=buffer[count]&mask;
       for(yt=0;yt<my;yt++)
         {
          if(bit)
```

```
                putpixel(x,y,color);
                y++;
                }
             mask>>=1;
             }
          mask=0x80;
          }
          y=y0;
          count-=3;
          x++;
          }
      switch(dis)          /*显示下一个汉字时的字间距与换行处理*/
      {
       case 0:{
          x0+=24*mx+dis;
          if((x0+24*mx)>mxx)
          {
          x0=0;
          y0+=24*my+dis;
          }
          break;
          }
       case 1:{
          y0+=24*my+dis;
          if((y0+24*my)>myy)
          {
          y0=0;
          x0+=24*mx+dis;
          }
          break;
          }
        }
       }
      free(buffer);
      fclose(fp);
     }

    main()          /*演示放大汉字*/
    {
```

```
int driver=DETECT,mode;
initgraph(&driver,&mode,"计算机");
fdhz(200,50,2,2,6,0,14,"计算机","hzk24h");
getch();
fdhz(40,60,3,2,4,1,4,"","hzk24s");
getch();
closegraph();
}
```

14.6 利用 BIOS 中断调用显示汉字

为了更清楚地理解用中文 DOS 的 BIOS 10H 中断调用显示汉字的方法，必须首先了解西文 DOS 下，在文本方式时，显示字符的原理：

14.6.1 文本方式下显示汉字的原理

在文本方式时，当一屏可显示 25 行，每行为 40 或 80 个字符数字时(标准文本方式)，需要占用 VRAM 2000 或 4000 个字节，即对 40 列方式需显示 1000 个字符(每页)，对 80 列方式需显示 2000 个字符；所对应的 VRAM，一个字节表示要显示的字符的 ASCII 代码，一个字节表示该字符的显示属性。因此，对 80 列方式需 4000 个字节为一页，实际长度为 4096 个字节，后面 96 个字节不用，偶地址存放要显示字符的 ASCII 代码，奇地址存放显示字符的显示属性，此时若用位面结构来理解，则位面 0 表示要显示字符的 ASCII 代码，位面 1 表示显示属性(注意！此时和上述图形方式下位面结构的含义不同)。这种结构如图 14.12 所示，其属性字节各位含义如图 14.11 所示，其中位 0，位 1，位 2 表示前景色，即选择要显示的字符颜色，位 4，位 5 和位 6 表示背景色，即要显示字符位置的背景颜色，位 3 用来对前景色进行加亮控制，即选择 8 到 16 号颜色(加亮颜色)，这些颜色规定如下表所示。

属性位(前景颜色位或背景颜色位)	颜色	加亮颜色(当位 3 为 1 时)
000	黑	灰
001	蓝	浅蓝
010	绿	浅绿
011	青	浅青
100	红	浅红
101	紫红	浅紫红
110	棕	黄
111	灰	白

当第 7 位选 1 时，则显示的字符闪烁。

当显示字符时，由位面 1 的一个属性字节作为对调色板寄存器的索引，可决定字符显示

图 14.11 显示属性字节

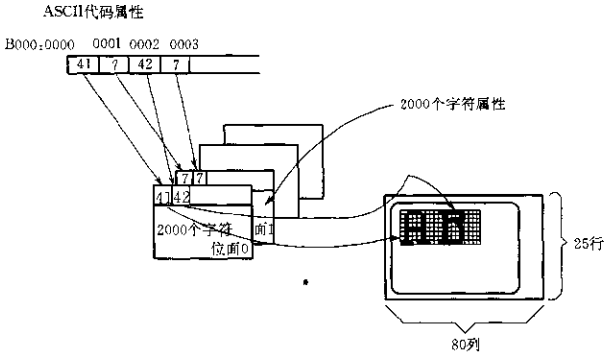


图 14.12 处理器的地址与对应的位面

的前景色与背景色，如图 14.13 所示。

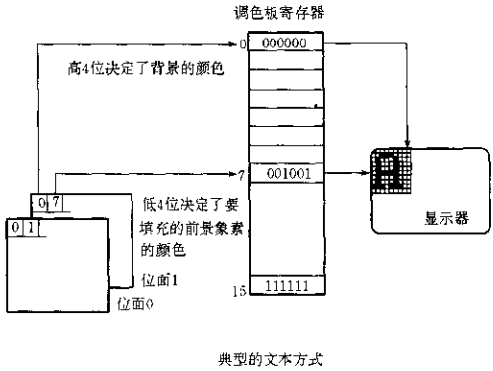


图 14.13 文本方式下显示字符

对 PC/XT 上使用的彩色适配器 CGA，在文本方式时显示的字符是由 VRAM 中偶字节的 ASCII 代码作为地址，从字符发生器(字符点阵固化在 ROM 中)相应的单元取出字符点阵的一排点显示，而接着又取出下一排点，直至这一行显示完。但是在 EGA 和 VGA 中没

有使用 ROM，当在文本方式时，可以由用户用 BIOS 调用将指定的字符集装入 VRAM 的位面 2(或者内定字符集自动装入)，对 EGA 一次可装入 4 种字符集(每个集为 256 字节)，对 VGA 一次可同时装入 8 种字符集。这些字符集的差别是，每个字符点阵的宽度均为 8 个象素点，但它们的高度所占象素点不同，标准的字符点阵是 8×8(如 CGA 中所用)，增强的彩色字符集是 8×14。当用户用 BIOS 选择了一种文本操作模式时，便由 BIOS 自动地将其中一种装入 VRAM 的位面 2。对单色文本模式将装入 8×14 增强的彩色字符点阵集，当然其它的字符集可以用 BIOS 调用来指定装入。一次只能激活(使用)两种字符集(即在一屏上可同时显示 512 种字符)，若使用两种字符集时，图 14.11 所示的字符属性字节的位 3 则用来选择一种字符集。

图 14.14 示出了从位面 0 所存的 ASCII 码作为地址索引，从位面 2 所存字符集中取出相应字符的点阵信息，接着在屏上显示。

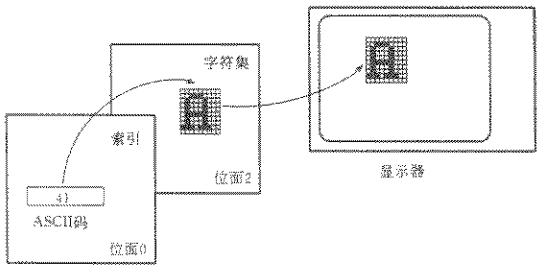


图 14.14　文本方式下位面 0 和位面 2 的使用

所以在文本方式时，由 VRAM 的位面 0 的一个字节给出要显示的一个字符的 ASCII 码，由对应的位面 1 的一个字节给出要显示字符的颜色和背景色，而由位面 0 的 ASCII 字节作为索引，在位面 2 上找出要显示字符的点阵信息进行显示。

中文 DOS 对 BIOS 10H 中断调用进行了修改，使原来只能显示英文字符功能变成也可显示汉字，对显示汉字要用的 10H 中断调用要用到两个功能，即指明要显示汉字时的光标位置设置用 10H 中断的 2 号功能调用，在设置的光标处显示汉字的 10H 中断的 9 号功能调用，各入口参数如下：

2 号功能调用(设置光标位置)

入口参数：AH＝2

DH＝行号

DL＝列号

BH＝要显示的页号

9 号功能调用(在光标位置上显示字符及属性)

入口参数：AH＝9

BH＝当前光标所在的页号

AL＝要显示的字符的 ASCII 码

BL=字符属性

CX=要显示的字符数

若将这两个功能调用结合起来编成一个显示汉字的函数，设函数名为 out_hz；

```
void out_hz(int x, int y, char * hz_P,int attrib)
  {
     union REGS regs;
     while( * hz_P){
        regs.h.ah=2;
        regs.h.dl=x++;
        regs.h.dh=y;
        int86(0x10,&regs,&regs);      /* 设置光标位置调用 */
        regs.h.ah=9;
        regs.h.bh=0;
        regs.x,cx=1;                  /* 显示一个字符 */
        regs.h.al=hz_P++;
        regs.h.bl=attrib;             /* 属性字节 */
        int 86(0x10,&regs,&regs);     /* 显示字符调用 */
     }
  }
```

则可用它显示由 hz_P 指针指出的汉字，直到将由 hz_P 指向的装汉字的缓冲区内容显示完为止。

14.6.2 中文菜单

下面程序就是采用类似的函数编制的一个中文的信号采集菜单程序，当在中文 DOS 下运行该程序时，屏上出现一个主菜单，它由四个彩色矩形框组成，中间标有要采集的内容，用 ← →左右箭头可选菜单项，选中的项其框便成为红色框和显示红色汉字，当按回车或向下箭头 ↓ 时，便在该项下面出现一个白色框的下拉子菜单，其间有相应的黄色子菜单项，如选中温度项，当按回车或 ↓ 时，便出现如图 14.15 所示子菜单，然后可按 ↓ 或 ↑ 选中一个子菜单项(选中的子菜单项此时变成红色)，当按回车时，便执行采集该项值的采集程序 Acquir ()并回填采集的现时值，程序中该子程序省略。

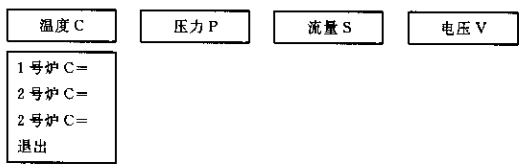


图 14.15 中文菜单

在该程序中，首先用 DOS 命令 CLS 清屏，接着用 nocsr 函数清光标，它也是用了 BIOS 调用，该程序中 bolawin()函数用来画菜单框，它实际上是用在汉字状态下，用区位码选中的图符“┌”,“—”,“┐”,“|”,“└”,“┘”来组成一个菜单框。清汉字函数 clr_hz()是用背景色写空格符来达到清除汉字的作用，键盘扫描函数 key()利用 bioskey()键盘操作函数来判断是否按键并返回按键的 ASCII 码或扩充的 ASCII 码，其格式为：

int bioskey (int cmd);

它包含在 bios.h 头文件中。当 cmd=1 时，则该函数用来查询是否有键按下，有，则返回非零值，否则返回 0。当 cmd=0 时，则返回按键的键值，该值是一个两字节的 16 位二进制数，若按下的是普通键，则低 8 位为其 ASCII 码值，若是一些特殊键，则是扩充的 ASCII 码，其低 8 位为 0，高 8 位为扩充码，故在 key()函数中，利用 key1=key1&0xff? key&0xff:key1>>8;来得到 ASCII 码或扩充的 ASCII 码。

将该程序在 TC 下进行编辑，然后在中文 WS 下输入用到的汉字，用区位码输入作菜单框的图符“┌”,“—”,“┐”,……等，然后在 TC 下编译带汉字的源程序(要删去由 WS 编辑后源文件带来的几个多余符号)并连接，在中文 DOS 下运行生成的执行程序，即可产生出如上所述的中文菜单，用上述的 BIOS INT 16H 调用，只能在中文状态下，才能正确地显示汉字，在西文状态下将会运行不正常，因而这种方法显示汉字带有一定的局限性。

```
#include <bios.h>
#include <dos.h>
#define ENTER 13
#define ESC 27
int menu(),submenu(),acquire();
int j[4]={0};
char *f[]={"温度 C","压力 P","流量 S","电压 V"};
char *f1[]={"1 号炉 C=","2 号炉 C=","3 号炉 C=","退出"};
char *f2[]={"1 号炉 P=","2 号炉 P=","3 号炉 P="};
char *f3[]={"水流 S=","气流 S=","原料流 S="};
char *f4[]={"1 号炉电压 V=","2 号炉电压 V=","3 号炉电压 V="};
main()
{
  void nocsr(),out_hz();
  int x=0,v=0;
  system("cls");          /*清屏*/
  nocsr();                /*清光标*/
  boldwin(1,1,3,17,13); /*写主菜单框*/
  out_hz(2,7,f[0],12);  /*写汉字温度*/
  for(x=1;x<4;x++)
   {
    boldwin(1,x*18+1,3,x*18+17,9);
```

```
    out_hz(2,x*18+7,f[x],14);                    /*在框内分别写压力,流量,电压汉字*/
    }
    x=0;
    menu(x,v,0);
  }
  int menu(int x,int v,int o)
    {
    do{
      v=key();                                   /*检查按键*/
     do{
     if(v==75 || v==77)                          /*是否为左右箭头*/
       {
        boldwin(1,x*18+1,3,x*18+17,9);
        out_hz(2,x*18+7,f[x],14);                /*恢复原色框和字*/
       if(v==75)x=(x==0)? 3:(x-1);
       if(v==77)x=(x==3)? 0:(x+1);               /*左右箭头处理*/
        holdwin(1,x*18+1,3,x*18+17,13);
        out_hz(2,x*18+7,f[x],12);                /*选中处用红色画框和汉字*/
       if(o==1) submenu(x,v);
        }
        if(v==80 || v==ENTER)                    /*若是向下箭头或回车键,则调出下拉子菜单*/
          {
           submenu(x,v);
           o=1;
          }
           }while(o==1);
             }while((v==key())! =45);            /*当按 ALT_X 时,退出循环*/
           exit(0);
          }
int submenu(int x,int v1)                        /*下拉子菜单*/
  {
   int i,l;                                      /*选中子菜单的相应内容*/
   char *fx[5];                                  /*将选中子菜单内容存入 fx[]中*/
   switch(x)
    {
     case 0:l=4;
          for(i=0;i<l;i++)
          fx[i]=f1[i];
          break;
```

```
  case 1:l=3;
         for(i=0;i<l;i++)
         fx[i]=f2[i];
         break;
  case 2:l=3;
         for(i=0;i<l;i++)
         fx[i]=f3[i];
         break;
  case 3:l=3;
         for(i=0;i<l;i++)
         fx[i]=f4[i];
         break;
  }
for(i=0;i<l;i++)
 out_hz(i*2+5,x*18+3,fx[i],14);      /*写子菜单*/
 i=j[x];
 out_hz(i*2+5,x*18+3,fx[i],12);
 boldwin(4,x*18-1,2*l+5,x*18+17,7);
  do
  {
   if(v1==72 || v1==80)      /*判断上下箭头*/
    {
   out_hz(i*2+5,x*18+3,fx[i],14);
   if(v1==72)i=(i<=0)? (l-1):(i-1);
   if(v1==80)i=(i>=(l-1))? 0:(i+1);
    out_hz(i*2+5,x*18+3,fx[i],12);  /*依照箭头上下交替写*/
   j[x]=i;
    }
    if(v1==75 || v1==77)      /*若左右箭头则擦和写*/
    {
      clr_hz(l,x);
      menu(x,v1,l);
    }
      if(v1==ENTER)
       {
        if(i==3&&x==0)
         {
         system("cls");
         exit(0);
```

```
            }
          /* acquire(x,i); */      /*调用所选项参数的采集程序*/
            }
           }while((v1=key())! =ESC);      /*按 ESC 退出*/
           clr_hz(1,x);      /*清除下拉菜单*/
           menu(x,v1,0);
           }                  /*调主菜单*/
int clr_hz(int l1,int xx)      /*清除汉字*/
 {
  int i,j;
  for(i=0;i<=(l1*2+1);i++)
   for(j=0;j<=18;j++)
 } out_hz(i+4,xx*18+j," ",0);
 int key(void)          /*键扫描函数*
  {
   int key1;
    while(bioskey(1)==0);
    key1=bioskey(0);
    key1=key1&0xff? key1&0xff:key1>>8;
    return(key1);
   }
   int boldwin(int x1,int y1,int x2,int y2,int att)   /*写框函数*/
    {
     int i;
     char *bc[6]={"┌","─","┐","│","└","┘"};
      out_hz(x1,y1,bc[0],att);
     for(i=y1+2;i<y2;i+=2)
      out_hz(x1,i,bc[1],att);
      out_hz(x1,i,bc[2],att);
     for(i=x1+1;i<x2;i++)
      {
      out_hz(i,y1,bc[3],att);
      out_hz(i,y2,bc[3],att);
      }
     out_hz(x2,y1,bc[4],att);
      for(i=y1+2;i<y2;i+=2)
     out_hz(x2,i,bc[1],att);
     out_hz(x2,i,bc[5],att);
      }
```

```
void out_hz(int x,int y,char *hz_p,int attrih)
{
  union REGS regs;
  register int i;
   for(i=y;*hz_p;i++)
   {
     regs.h.ah=2;
     regs.h.dl=i;
     regs.h.dh=x;
     regs.h.bh=0;
     int86(0x10,&regs,&regs);      /*光标定位*/
     regs.h.ah=9;
     regs.h.bh=0;
     regs.x.cx=1;
     regs.h.al=*hz_p++;
     regs.h.bl=attrib;
     int86(0x10,&regs,&regs);      /*写汉字*/
     }
}
void nocsr()                       /*清光标*/
{
     union REGS regs;
     regs.h.ah=19;
     regs.h.al=0;
     int86(0x10,&regs,&regs);
   }
```

14.7 建立一个小型专用汉字库

对于一些应用程序，特别是实时性强的应用程序，可以将菜单中或程序提示中的汉字放在一个专用汉字库中，而不必带有中文 DOS 系统中标准汉字库，从而可大大节省程序开销，提高运行速度。由于用到的汉字数量很少，所以建一个小型汉字库是容易的。

专用字库的结构形式可根据情况而定，如一般可设计成如图 14.16 所示的结构形式，即有一项为库中汉字个数 N，另有 N 项为汉字机内码与对应的 16*16 点阵的字模，机内码在库中按升序即从小到大排列，若用结构定义，可表示如下：

汉字个数 N	
内码 1	字模 1
内码 2	字模 2
⋮	⋮
内码 N	字模 N

图 14.16 专用字库的结构

```
struct hz_mat{
        unsigned incode;
        char mat[32];
        };
struct hzlib{
        int n;
        struct hz_mat Lib[N]
        } clib;
```

14.7.1 库的建立方法

可以将用到的汉字，在中文 WS 下编成一个中文的文本文件，设为 hz.tex，然后可以编一个程序，从中文 DOS 的标准字库中，读出其对应的内码和 16×16 点阵字模，并将其写到库结构中相应的项中去，这样如此循环，可将所用的 N 个汉字均填入小型汉字库中，然后再用排序程序，将其按内码升序排列，至此一个小型专用数据库即建成，该库可作为一个头文件，用包含形式，将其和自己的应用程序联起来，这样当在应用程序中用到库中的汉字时，便可立即在库中找到对应的字模，并将其显示出来。

更快速的方法是用定义静态结构数组的方法，将内码与对应的字模一一赋给各数组成员，即相当于对全局变量初始化，然后将该结构数组作为全局变量使用，这样若要在程序某处显示汉字时，只要根据汉字的内码即可快速在结构数组中找到其对应的字模，从而显示出来。

也可用程序生成一个小型库文件，在应用程序中用打开文件的方式去使用它。

14.7.2 建立小型汉字库的程序例

设需要建立一个由如下汉字组成的小型字库：

春晓

孟浩然

春眠不觉晓

处处闻啼鸟

夜来风雨声

花落知多少

这可用中文 WS 编辑，设形成的文本文件名为 hz.txt。

这样运行下面的程序，就可建立一个如图 14.16 所示结构形式的专用字库，它包括上面诗词中用到的全部汉字，该字库名为 myclib.h。

程序中首先定义了两个结构，即结构 hz_mat 和 hzlib，而全局的结构变量 CLib 中的成员即数组 lib[N]则是 hz_mat 结构类型的，即 lib[N]中每个元素是由内码 incode 和对应的字模 mat[32]组成的一个结构。

函数 creatclib()用来建立由 C 盘 WPS 子目录下的 hz.txt 文本中的汉字组成的一个字库，该函数首先打开该文本文件进行读，由于在中文状态下用 WS 编辑的该文件是由国标汉字内码组成的，故用 fread(code,2,1,fp)，用 2 个字节为单位，将汉字内码读入由 code 指针

指出的内存中，然后查找 clib 中的 lib[i].incode 是否有该内码，若没有，则将该内码赋给它，并调用得到该内码对应汉字字模的函数 get_mat()，将得到的字模 bytes[32]再赋给 lib[n]中的 mat[]成分，如此循环，直到读到 hz.txt 的文件尾为止。这样就建立起了包括 bz.txt 中所有汉字的 clib 汉字库。

值得注意的是，该函数中定义了 unsigned 型的指针变量 code，因而它指向无符号的 16 位数据，即由 fread()函数读出的内码(2 个字节)放入由它指向的两个地址单元，如春字的内码为 0xbab4，b4 放低字节，ba 放高字节，当进行 * code 操作时，便可一次读出 16 位的内码。而在调用取得该内码对应的字模函数 get_mat()中，形参 char 型的 hz_code[]为两个字符型元素的数组，这样用 code 指针代替时，它将把其传递的内码看成为两个字节，即 hz_code[0]和 hz_code[1]。如春字，内码 0xbab4，经过 get_mat(code,bytes)调用后，则 hz_code[0]=0xb4，hz_code[1]=0xba；因而便可正确的得到其对应的区号(qh)和位号(wh)，因而得到其字模。

函数 writeclib()用来将得到的 clib 以定义的结构形式存入磁盘中，其路径名为：

c:\tt\myclib.b，

还需要说明的是为了对生成的库检索方便，提高效率，故用排序函数 sort()将 clib 中的结构数组 lib[N]按内码从小到大进行了排序，排序算法采用最简单的冒泡法，由于数据较少，用此法还是比较快的，对于其它排序法，如选择排序，插入排序，快速排序并不一定效率高，尤其快速排序对于大于 100 个以上的数据，才能显示出优越性来。

该程序以 myclib.h 头文件存盘，为的是在应用程序中，用#include"myclib.h"可将该库包括到应用程序中去，下面的程序例就采用了这种形式。

```
#include<stdio.h>
#include<graphics.h>
#include<fcntl.h>
#include<io.b>
#define N 30
int hzk_p;
void open_hzk(void);
void get_mat(char hz_code[],unsigned char buff[]);
void creatclib(char *name);
void sort();
void writeclib(char *name);
unsigned char bytes[32];

struct hz_mat{
   unsigned incode;
   unsigned char mat[32];
   };
struct hzlib{
   int n;
```

```
    struct hz_mat lib[N];
    }clib;
main()
{
  int x,y;
  unsigned ccode;
  open_hzk();
  creatclib("c:\\wps\\hz.txt");
  writeclib("c:\\tc\\myclib.h");

  }
void creatclib(char *name)                    /*建立一个汉字的字库*/
   {
    register i,j,found,n;
    unsigned *code[2];
    FILE *fp;
    n=0;
    fp=fopen("c:\\wps\\hz.txt","r");
    if(fp==NULL)
     {printf("hz.txt not exist! ENTER to system");
      exit(1);
     }
   while(! feof(fp))
    {
     fread(code,2,1,fp);                      /*从 hz.txt 文件中一次读两个字节内码*/
     found=0;
     for(i=0;i<N&&! found;i++)
       found=clih.lib[i].incode==*code;
       if(! found){
        clib.lib[n].incode=*code;             /*将内码写入 incode 中*/
        get mat(code,bytes);                  /*得到对应的字模*/
        for(j=0;j<32;j++)
        clib.lib[n].mat[j]=bytes[j];          /*赋给 mat[]*/
        n++;
       }
    }
  clib.n=n;
  fclose(fp);
  sort();
```

```
}
void open_hzk()      /*打开 213 中文系统中的 hzk16 字库*/
{
 hzk_p=open("c:\\213\\hzk16",O_BINARY | O_RDONLY);
  if(hzk_p==-1)
   {
    printf("the hzk16 not exit! ENTER to system\n");
    getch();
    exit(1);
    }
}
void get_mat(char hz_code[],unsigned char buff[])
 {int i;
  unsigned char qh,wh;
  unsigned long offset;
  qh=hz_code[0]-0xa0;
  wh=hz_code[1]-0xa0;
  offset=(94*(qh-1)+(wh-1))*32L;
  lseek(hzk_p,offset,SEEK_SET);
  read(hzk_p,buff,32);        /*得到字模*/
 }
void writeclib(char *name)      /*在磁盘上输出库*/
 {
  FILE *fp;
  int i,j;
  fp=fopen(name,"w");
  fprintf(fp,"struct hz_mat{\n");
  fprintf(fp,"unsigned incode;\n");
  fprintf("unsigned char mat[32];\n");
  fprintf(fp,"};\n");
  fprintf(fp,"struct hzlib {\n");
  fprintf(fp,"int n;\n");
  fprintf(fp,"struct hz mat lib[%d];\n",clib.n);
  fprintf(fp,"}clib={%d,\n{\n",clib.n);
  j=0;
  while(j<clib.n){
   fprintf(fp,"{0x%x,",clib.lib[j].incode);
   for(i=0;i<32;i++)
    {
```

```
        fprintf(fp,"0x%x",(unsigned char) clib.lib[j].mat[i]);
        if(i! =31)
         fprintf(fp,",");
         if((i+1)%8==0&&i! =31)
        fprintf(fp,"\n");
        }
        j++;
        if(j! =clib.n)
         fprintf(fp,"\n},\n");
         else
           fprintf(fp,"}\n");
        }
    fprintf(fp,"}\n};\n");
    fclose(fp);
   }
void sort(void)          /*对库按内码升序进行排序*/
   {
    register int a,b,i,m;
    unsigned t;
    char buffer[32];
    m=clib.n;
    for(a=1;a<m;++a)
     for(b=m-1;b>=a;--b)
      {
       if(clib.lib[b-1].incode>clib.lib[b].incode){
       t=clib.lib[b-1].incode;
       clib.lih[b-1].incode=clib.lib[b].incode;
       clib.lib[b].incode=t;
       for(i=0;i<32;i++)
       {
        buffer[i]=clib.lib[b-1].mat[i];
        clib.lib[b-1].mat[i]=clib.lib[b].mat[i];
        clib.lib[b].mat[i]=buffer[i];
        }
      }
    }
 }
```

下列程序段就是该程序生成的 myclib.h 头文件开始的一部分，可以看出，汉字内码及其对应的汉字字模已按升序排列，第一个是处字的内码 0xa6b4，它处于第 20 区的第 6 位，

其后是它的字模，第二个是花字的机内码 0xa8bb，它处于第 27 区第 8 位，其后是内码，第三个是知字内码 0xaad6，它处于第 48 区第 10 位，其后是字模……，该库中共有 22 个这样的结构。

```
struct hz_mat{
unsigned incode;
unsigned char mat[32];
};
struct hzlib {
int n;
struct hz_mat lih [22];
}clib={22,
{
{0xa6b4,0x0,0x40,0x10,0x40,0x10,0x40,0x10,0x40,   /*"处"字的区位码和字模*/
0x1e,0x40,0x22,0x60,0x22,0x50,0x22,0x4c,
0x54,0x44,0x94,0x40,0x8,0x40,0x14,0x40,
0x14,0x40,0x22,0x6,0x41,0xfc,0x80,0x0
},
{0xa8bb,0x8,0x20,0x8,0x24,0xff,0xfe,0x8,0x20,     /*"花"字的区位码和字模*/
0x8,0x20,0x8,0x80,0x10,0x88,0x10,0x98,
0x30,0xa0,0x50,0xc0,0x90,0x80,0x11,0x80,
0x12,0x82,0x14,0x82,0x10,0x7e,0x10,0x0
},
{0xaad6,0x20,0x0,0x20,0x0,0x22,0x4,0x3f,0x7e,     /*"知"的区位码和字模*/
0x28,0x44,0x48,0x44,0x88,0x44,0x9,0x44,
0xff,0xc4,0x8,0x44,0x8,0x44,0x14,0x44,
0x12,0x44,0x22,0x7c,0x40,0x44,0x80,0x0
},
{0xb4c0,0x1,0x0,0x1,0x0,0x1,0x8,0x7f,0xfc,
0x1,0x0,0x21,0x10,0x19,0x30,0x9,0x44,
0xff,0xfe,0x3,0x80,0x5,0x40,0x9,0x30,
0x31,0x1e,0xc1,0x4,0x1,0x0,0x1,0x0
},
{0xb9d2,0x2,0x0,0x1,0x4,0xff,0xfe,0x8,0x80,
0x8,0x80,0x10,0xf8,0x11,0x8,0x32,0x88,
0x56,0x50,0x99,0x10,0x10,0xa0,0x10,0x40,
0x10,0xa0,0x11,0x10,0x12,0xe,0x1c,0x4
},
```

14.7.3 利用小型汉字库显示汉字程序例

该程序使用了上面程序生成的小型汉字库，它将其以头文件形式包含到本文件中，这样后面当查找该字时，便以全局变量的形式对该库进行查找。在进行文件包含时，要写成这种形式：

```
#include"myclib.h"
```

而不能写成：

```
#include<myclib.h>
```

因后者将在系统设定的子目录 include 下查找包含文件 myclib.h，因而将不会找到，这样编译时将出现打不开该文件的错误信息，前者将在当前目录下查找 myclib.h，这正是我们所存 myclib.h 文件所在的目录。

下面程序将显示"处处花落"四个汉字，它由主函数中的指针变量 ccode 指明所存的地址，由于汉字内码为两字节，故定义指针变量为 unsigned 型，当进入 while 两重循环时，内层循环显示一个汉字，外层循环显示一行汉字(直到遇到字符串结束符为止)。

dis_hz()是在屏幕指定位置显示汉字的函数，它调用了得到显示汉字字模的函数 get_dismat()和将字模显示到屏上的函数 dis_scr()。

get dismat()函数是根据汉字内码，在小型字库 clib 中找出其对应的字模，该函数搜索方法采用折半法，即先从库的中间去找，若未找到，再看查找的内码是大于还是小于库中间的内码，从而可知要找的内码在库中是前半部还是后半部，然后再折半，如此反复，最后可找到要显示的内码及其对应的字模，从而可以用它进行显示。

dis_scr()函数则根据字模指针和要显示的 x，y 位置及颜色，在屏上显示出该汉字来，该函数的显示过程，前面的程序已经进行过解释，不再赘述。

为了使该程序和头文件 myclib.h 有显式的依赖关系，可以定义一个 project 文件：

```
clibsor(myclib.h)
```

这样当 myclib.h 中的内容改变了，如增加了汉字，则该程序将被重新编译。

```
#include<stdio.h>
#include<graphics.h>
#include<fcntl.h>
#include<io.h>
#include"myclib.h"                    /*将建立的小型字库包括进来*/
void init(void);
void dis_hz(int x0,int y0,unsigned code);
void dis_scr(int x,int y,unsigned char *pt,int color);
unsigned char *get dismat(unsigned code);
unsigned char bytes[32];

main()
{
```

```
int i,x,y;
unsigned *ccode="处处花落";      /*要显示的汉字*/
init();
x=20;
y=100;
while(*ccode! =NULL)
  {
   while(x<640&&(*ccode! =NULL))
    {
     dis_hz(x,y,*ccode);          /*取内码并调用函数显示*/
     x+=20;
     ccode+=1;
    }
   x=20;
   y+=20;
  }
 getch();
 closegraph();
}
unsigned char *get_dismat(unsigned code)  /*从小型字库中寻找要显示的汉字字模*/
 {
  int down=0;
  int up=clib.n-1,mid;
  int found=0;
  while (down<=up&&! found)
   {
    mid=(up+down)/2;
     if(code==clib.lib[mid].incode)
      found=1;
     else if(code>clib.lib[mid].incode)
      down=mid+1;
      else
        up=mid-1;
    }
    return(found? clib.lib[mid].mat:NULL);
 }
void dis_hz(int x0,int y0,unsigned code)
 { int i;    unsigned char far *p,mat[32];
   if(p=get_dismat(code))
```

```
    dis_scr(x0,y0,p,LIGHTRED);
  }
void init(void)                          /*屏幕图形方式初始化*/
  {
  int driver=DETECT;
  int mode=0;
  initgraph(&driver,&mode,"");
  }
void dis_scr(int x,int y,unsigned char *pt,int color)  /*在屏上显示汉字*/
  {
    unsigned char mask[]={0x80,0x40,0x20,0x10,0x08,0x04,
                          0x02,0x01};
    register int i,j,x0,y0,pos;
    unsigned char mat[32];
    for(i=0;i<32;i++)
    mat[i]=*(pt++);
    y0=y;
    for(i=0;i<16;i++)
    {
     x0=x;
     pos=2*i;
     for(j=0;j<16;j++)
       {
        if((mask[j%8]&mat[pos+j/8])!=NULL)
       putpixel(x0,y0,color);
       ++x0;
       }
     ++y0;
    }
  }
```

第 15 章

C 语言与汇编语言的混合编程

虽然 C 语言可以完成许多由汇编语言完成的工作，人们说它是一种介于高级语言与低级语言之间的一种中级语言，但在有些要对硬件和操作系统直接操作的场合，在有些要求执行速度快的场合，仍要用到汇编语言的程序，这可由 C 程序直接调用汇编程序来实现，这种调用可用两种方法进行，即在 C 程序中嵌入汇编代码，这是一种当汇编程序较短时可采用的方法，另一种是直接调用汇编语言子程序，这是一种较广泛采用的方法，我们将在后面分别进行介绍。

15.1 汇编语言子程序使用的场合

由于汇编语言编程比较困难，因此在编一个较大的实用程序时，编程语言的选择很重要，由于 C 的结构化设计特点，简洁、明快的特色，丰富的库函数和较好的调试手段，因而较多地选 C 语言来编程，但在有些场合仍需要用汇编语言编程，而 C 语言也提供了这种混合编程的方法，它支持调用汇编语言的子程序，也支持将汇编指令当作它的语句而内嵌于 C 程序中，那么什么时候选用汇编语言编子程序好呢？一般来说内嵌汇编指令，仅适于极短的用少量汇编指令即可完成的操作中，而要完成一种特定操作功能，又需要较短时间，则可用汇编语言编成子程序形式来实现，比如在信号采集中，外界的信号要通过模/数转换电路将模拟信号变成数字量化的二进制信号，送到计算机去处理，而模数电路(即 A/D 电路)是由一些硬件组成，我们知道根据信号采样定理(Nyqust)，采样频率应大于信号频率的两倍以上，对采得的信号进行处理时，才不会失真，当对信号进行幅度分析时要准确记录其波形，，则应该有十几倍于信号频率的采样速度，因而这时对 A/D 电路的初始化，启动，采样，则应选择用汇编语言编制子程序来实现，加快采样速度。

当某个突发事件产生时，如定时到，条件越界(湿度过高，电压过高等)，常用中断处理程序来处理这些事件，由于这些事件一般中断级别较高，要求中断处理快速，因而常选用汇编语言的中断处理程序，而没有中断事件发生的大量信息处理，则可用 C 语言程序来完成。

对于有些需要直接对 PC 机硬件进行操作的功能，为了直接，编程方便，也常采用汇编子程序的方法。比如对 PC 机系统板上的某些接口芯片的编程，和某些端口的存取，对 VGA 适配器的编程，常可选用汇编子程序，编起来简单明了，便于查错，且操作快速。

当在编程时，若某种特定操作重复次数过多，为了提高程序的执行速度，也可选用汇编语言编程的方法，比如：

```
main()
  {
```

```
register int i;
init();
for (i=0;i<1000;++i)
    {
      phase1();
      phase2();
  if(i==10)phase3();
  }
  byebye();
}
```

该程序中 phase1()和 phase2()函数将要执行 1000 次，因而若要提高速度，可考虑用汇编语言编这二个函数，而 phase3()和 byebye()仅执行一次，故可用 C 语言编的函数，至于 init()函数，若是对硬件初始化的函数，若考虑到让其立即完成初始化，则也应用汇编子程序来实现。

对于用汇编语言编程的人，要有 PC 硬件方面的知识和汇编语言方面的知识，为了方便读者，特将 Microsoft 公司的 MASM 宏汇编(3.0 以上)的汇编程序基本结构和框架，及用到的一些有关伪指令作简单介绍，MASM 5.0 以上版本推出了宏汇编程序简化段的编程方法，它对和高级语言混合编程带来了更多的方便，因此后面也将作以介绍。

15.2 汇编语言程序的结构

15.2.1 常规的汇编语言程序结构

一个完整的汇编语言程序由三个段组成，即堆栈段、数据段和代码段，在汇编程序中，用段定义伪指令 SEGMENT 和 ENDS 将源程序划分成若干段，以便于产生目标代码和为它们分配存储区，在进行连接时，将同名段进行组合。

1. 段的定义格式

一个完整的段定义格式如下所列：

```
段名        SEGMENT    边界类型 结合类型 USE'类型'
            语句
             ⋮
段名        ENDS
```

其中段名为定义段的名称，该名字可以是唯一的，也可以和程序中其它的段同名，这时在同一程序中同名的段就可看作是同一个段。这种方式常用在模块化程序结构设计中，同一段的不同部分，分别放置在不同的子模块中，在各子模块中，这些不同部分具有同一名字，表示的是同一个段。

边界类型、结合类型、USE 和'类型'都是可选项，即可有可无，它们用来告诉汇编和连接程序如何对段进行组合，这些选择项的排列次序不能变更，当然其中一些可省略，即取缺省值。

注意！这些选择项不是伪指令，它们可看作 SEGMENT 语句的参数，PAGE 和 PUBLIC 伪指令和上面同名的选择项是两个不同的东西，要注意区别。

现将各选择项加以说明。

(1) 边界类型

边界类型表示段开始地址位于何处，可有五种选择：

BYTE——表示段开始地址可以位于任何地址上，即字节地址。

WORD——表示段开始地址为偶数地址，即字地址。

DWORD——表示段开始地址为 4 的倍数，即 4 个字节(双字)，一般用于 80386 的 32 位段中。

PARA——段开始地址为 16 的倍数，它是缺省的类型，即若不指明这个选择项时，段地址便属于该类型。

PAGE——表示段地址为 255 的倍数，因 256 字节为一页，所以实际上是用页为段地址单位，即页型地址位于页边界上。

各边界类型的 20 位地址示意如下：

××× ×× ××××××××××××××××	BYTE 型段地址
××× ×× ××××××××××××××××0	WORD 型段地址
××× ×× ××××××××××××××××00	DWORD 型段地址
××× ×× ××××××××××××××0000	PARA 型段地址
××× ×× ××××××××00000000	PAGE 型段地址

在较早的一些 MASM 版本中(4.0 以下)，没有 DWORD 型，它是为了适应 80386 处理器而新增加的。

边界类型表示了在存储区如何连续地存放各个段，段间有无间隙，该项一般省略，因而边界类型是 PARA 型的。

(2) 结合类型

结合类型是告诉连接程序，该段和其它段的结合关系，连接程序可以将不同模块的同名段进行结合，根据结合类型，可将各段连接在一起，或重叠在一起，结合类型有：

NONE——表明本段与其它段逻辑上不发生关系，当结合类型项省略时，便指定为这一结合类型。

PUBLIC——表示将所有该类型的同名段连接成一个连续的段，公用一个段地址，所有的原各分段内的偏移量都转变成相对于连续段的开始地址的偏移量，运行时装入同一物理段中。

STACK——表示该段为堆栈段，当进行连接时同名的堆栈段连接成一个连续段，连接方式同 PUBLIC，但不同的是连续段的段地址是放在 SS 段寄存器中，当段用 STACK 类型说明后，SS 就自动初始化为堆栈段的段地址。

COMMON——表示所有该类型的同名段都有相同的段地址，这些同名段可相互覆盖，段长度为其中最长的同名段的长度。利用这种同名段的连接法，可使不同模块的变量或标号使用同一存储区域，便于模块间通信和信息交换。

MEMORY——将所有该类型的同名段连接成一个连续段，其处理同 PUBLIC 段。虽然连接程序不单独区分 MEMORY 类型，但 MASM 仍允许使用该类型，以使得能和 INTEL 公司汇编程序的 MEMORY 类型兼容。

AT 地址表达式——表示该段地址以地址表达式的值为段地址，段内标号和变量地址均以此地址来进行确定。这是一种由用户给段定义地址，但这种方式不能用于代码段。

(3) USE 类型

USE 类型说明了 80386 的段字长，段字长指缺省的操作数和段地址的长．USE 可以是 USE 16 或 USE 32，16 位的段可含 64K 字节，32 位的段可含 4M 字节，但当前版本的 DOS 将段限制在 64K 内，当在程序开头使用了．386 伪指令后，缺省类型为 32 位段字长。

在禁止用 80386 的情况下，使用 USE 选择项将导致出错。该选择项只有使用 MASM 5.0 以上版本时，才允许有。

若结合类型这一项省略，则该段是独立段(NONE 型)，和其它段没有联系。

(4) "类型"

是用单引号括起来的字符串，以表示该段的类型，如代码段(CODE)、数据段(DATA)、堆栈段(STACK)等，当然也允许用户在类别中用其它的名，这样进行连接时，连接程序便将同类型的段(但不一定同名)放在连续的存储区内。

上述段的组合类型适于多个模块连接时用，若程序仅有一个模块，只包括代码段、数据段和堆栈段时，为了和其它段相区别，除了堆栈段用 STACK 说明外，其它段的结合类型，类型均可缺省。例如有两个模块：

```
  模块 1
STACK       SEGMENT     STACK
            DW          20DUP(?)
STACK       ENDS
DATA        SEGMENT     COMMON
              ⋮
DATA        ENDS
CODE        SEGMENT     PUBLIC
              ⋮
CODE        ENDS
            END
  模块 2
STACK       SEGMENT     STACK
            DW          30DUP(?)
STACK       ENDS
DATA        SEGMENT     COMMON
              ⋮
DATA        ENDS
CODE        SEGMENT     PUBLIC
              ⋮
CODE        ENDS
            END
```

当汇编连接后，存储区映象如图 15.1 所示，(假设模块 2 的数据段长于模块 1 的)此时由于代码段是同名段，且用 PUBLIC 说明，故连接时，变成一个物理段，数据段也同名，但用 COMMON 说明，故连接时，被重叠放在一起，其长度是同名段中最长者，堆栈段具有同名，

因此被连接成一个较大堆栈，供两模块共享。

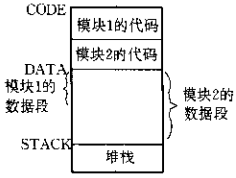


图 15.1　装入模块地址图

为了告知汇编程序哪一个段寄存器是该段的段地址寄存器，以便对使用变量或标号的指令汇编出正确的目标代码，必须用 ASSUME 伪指令，其格式为

ASSUME　段寄存器：段名[，段寄存器：段名，…]

其中段寄存器只能是：CS、DS、ES、SS。

如在上例中：CODE 段为代码段，DATA 段为数据段，STACK 段为堆栈段，则应在 CODE 段开头加入这条伪指令，以告诉汇编程序相应段的段地址存于何段地址寄存器中。由于 ASSUME 伪指令只是设相应段地址存于何段地址寄存器中，并没有执行把段地址放入该寄存器的操作，因而段地址寄存器装入的真实段地址，还要通过汇编指令来完成，而且必须完成这一步，例如可写成：

```
CODE        SEGMENT
            ASSUMEN     CS: CODE,DS: DATA,SS: STACK
            MOV         AX,DATA
            MOV         DS,AX
             ⋮
CODE        ENDS
```

当程序运行时由于 DOS 的装入程序负责把 CS 初始化成正确的代码段地址，SS 初始化为正确的堆栈段地址，因此用户不必设置。但装入程序已将 DS 当作它用，因此编程者必须用两条 MOV 指令对 DS 初始化，以装入数据段地址，当使用附加段时，也要用 MOV 指令给 ES 赋段地址。代码段可以看作由许多过程组成，过程的定义有一定的格式。

2. 过程的定义格式

在汇编程序设计中，可将具有一定功能的程序段看成一个过程（相当子程序或函数）。它可以被别的程序调用也可由本程序顺序执行，也可作为中断服务程序，中断响应后，转此执行。过程由伪指令 PROC 和 ENDP 来定义，其格式为：

```
过程名   PROC     [类型]
          ⋮    过程体
         RET
          ⋮
过程名   ENDP
```

其中过程名是为过程起的名称，不能省略，过程的类型由 FAR 和 NEAR 来确定，ENDP 表示过程结束，注意！过程体内应至少有一条 RET 指令，以便返回被调用处，过程可以嵌套，即过程可以调用别的过程，过程也可递归使用，即过程可以调用过程本身。

过程类型 FAR 或 NEAR 的含义是：

NEAR　近过程，即该过程仅在本段内调用，RET 表示是段内返回指令。返回时，仅 IP 改变（即 16 位字退栈）。当过程缺省类型这一项时，该过程便是一个 NEAR 过程。

例如一个延时 100ms 的子程序，其过程定义是

```
SOFTDLY    PROC
           MOV       BL,10
;内循环延时 10ms
DELAY:     MOV       CX,2801
WAIT:      LOOP      WAIT
           DEC       BL
           JNZ       DELAY
           RET
SOFTDLY    ENDP
```

类似这样的程序，在接口程序设计中，经常用于软件延时，改变装入 BL 或内循环使用的 CX 中值，就可改变延时的时间。

FAR　远过程，可由外段调用，RET 表示是段间返回，返回时，IP 和 CS 均改变(即两个 16 位字退栈)。

如果--个过程是由 DOS 直接调入内存执行的，则该过程一定要定义成 FAR 型，否则 RET 将返回不到 DOS 去。

示例如下：

没有两个段，其结构如下：

```
CODE1      SEGMENT
           ASSUME    CS: CODE1
            ⋮
FARPROC    PROC      FAR
            ⋮
           RET
            ⋮
FARPROC    ENOP
CODE1      ENDS
CODE2      SEGMENT
           ASSUME    CS: CODE2
            ⋮
           CALL      FARPROC
            ⋮
           CALL      NEARPROC
            ⋮
NEARPROC   PROC      NEAR
            ⋮
           RET
NEARPROC   NEDP
CODE2      ENDS
```

由于 CODE1 段中的 FARPROC 过程被另一段 CODE2 所调用，故定义为远过程，CODE2 段内的 NEARPROC 仅被本段调用，故定义为近过程。

3. 一个标准的汇编语言程序结构

一个标准的汇编语言程序结构是由三个段组成的，其程序中按排的段序为 STACK、DATA 然后 CODE，这个段序决定了汇编系统进行编译连接时，将按段序(即段排列的先后次序)来分配存储区，这也是 Microsoft 汇编所要求的次序，因按此段序，先分配了变量和数据的存储区，代码段才能得知其操作数的地址。

代码段可由许多过程组成，下面就是一个标准的汇编语言程序的框架，编程时，只要填入相应的内容即可，当然 STACK 段也可增大。

```
STACK    SEGMMENT   STACK
         DB         32  DUP(?)
STACK    ENDS
DATA     SEGMENT
         ⋮          各种数据项
DATA     ENDS
CODE     SEGMENT
MAIN     PROC       FAR
         ASSUME     CS:CODE,DS:DATA,ES:DATA,SS:STACK
         PUSH       DS                ;DS 值入栈
         MOV        AX,0
         PUSH       AX                ;0 入栈
         MOV        AX,DATA
         MOV        DS,AX             ;数据段地址送 DS
         MOV        ES,AX             ;数据段地址送 ES
         ⋮
         RET                          ;返回 DOS
MAIN     ENDP                         ;过程结束
CODE     ENDS                         ;代码段结束
         END        MAIN              ;程序结束，启动地址为 MAIN
```

15.2.2 用简化段定义的程序结构

若要汇编 80386 等新增加的指令或增加了原指令功能的指令，需使用近几年推出的汇编系统，如 Microsoft 公司的 MASM 5.0 或以上版本，该版本引入了简化段定义的伪指令，用它可简化段的定义。

一个段可以用简化段定义的伪指令来定义，当然也可用以前所用的段定义方式来定义。例如一个程序，用常规的段定义方式定义了三个段：

```
STACK    SEGMENT    PARA STACK STACK
         DB         100H DUP(?)
STACK    ENDS
_DATA    SEGMENT    WORD PUBLIC DATA
ivariable DB        5
iarray   DW         50 DUP(5)
string   DB         'This is a sering'
iarrag   DW         50 DUP(?)
```

```
_DATA     ENDS
_TEXT     SEGMENT     WORD PUBLIC 'CODE'
Start:    ASSUME      CS:_TEXT,DS:_DATA,SS:STACK
          PUSH        DS
          MOV         AX,0
          :
_TEXT     ENDS
          END         Start
```

若用简化的段定义可写成如下格式：

```
          .MODEL      SMALL
          .STACK      100H
          .DATA
ivariable DB          5
iarray    DW          50 DUP(5)
string    DB          'This is a string'
iarray    DW          50 DUP(?)
          .CODE
Start:    PUSH        DS
          MOV         AX,0
          :
          END         Start
```

上例中使用了简化段定义的伪指令 .STACK,.DATA 和 .CODE，它们分别表示了一个段的开始，也表示了前一段的结束，而且自动地产生了 ASSUME 语句的功能。可以看出，使用简化段的定义伪指令起到了简化程序设计的作用，如果编写的是 MICROSOFT 的高级语言调用的子程序，那么使用简化段定义的伪指令将保证程序相互兼容和一些定义的一致性。

15.2.3 简化段定义的伪指令

简化段定义的伪指令有 .CODE,.DATA,.DATA?,.FARDATA,.FARDATA?,.CONST 和 .STACK，它们都表示了一个段的开始，同时也说明了前一段的结束，若这个段是程序中的最后一个段，则该段以 END 伪指令结束。这些伪指令使用的格式如下：

.STACK 长度

.CODE 名字

.DATA

.DATA?

.FARDATA 名字

.FARDATA? 名字

.CONST

上述伪指令中带有名字的项，可以省略，若省略，则使用缺省名，可看表 15.1。

STACK 伪指令中的参数——长度，用于定义栈的长度，若省略，则使用缺省值 1K 字节。

段中的数据若不确定，则以？和 DUP 来定义，表示数据不确定，可任意。如 DATA？表示数据段中数据不确定，可任意，可这样定义：

```
        .DATA?
VAR     DB          100 DUP(?)
```

用伪指令 .STACK、.CONST、.DATA 或 .DATA？定义的段中的数据存放在一个叫 DGROUP 的段组中，所谓段组是指这些段共用一个起始地址，各个段内的偏移地址均以这个起始地址为起点，而不依本段内的段地址为起点，关于段组后面说明。

用伪指令 .FARDATA 或 .FARDATA？定义的段中数据不放在任何段组中，它们属于 FAR(远程的)。

上述的 .DATA、.DATA？.CONST、.FARDATA、.FARDATA？实际上都是数据段，它们用于存放不同类型的数据。

15.2.4 段组定义伪指令

不同的段，使用同一个起始地址作为其段内偏移地址的参考点，这些段就称为一个段组，它们可以用同一个段寄存器来访问，定义段组伪指令的格式是：

名字　GROUP　段名 1,段名 2,……

名字表示定义的段组名，它也代表了这个段组的起始地址，段名表示已定义的段的名字。

段组中各段间不一定是连续的，也可能中间插有不是该段组的其它段。段组中第一个段的首字节到最后一个段的最末字节其距离不能超过 65535 个字节，即不能超过 64K 字节。

例如：

```
DGROUP    GROUP     ASEG,CSEG
          ASSUME    DS:DGROUP
ASEG      SEGMENT   WORD PUBLIC 'DATA'
            ⋮
asym:
ASEG      ENDS
BSEG      SEGMENT   WORD PUBLIC 'DATA'
bsym:       ⋮
BSEG      ENDS
CSEG      SEGMENT   WORD PUBLIC 'DATA'
csym:       ⋮
CSEG      ENDS
          END
```

上例中共定义了三个段，其中 ASEG 和 CSEG 属于同一段组 DGROUP 中，而 BSEG 段则是独立的一个段，段内地址的计算如图 15.2 所示。由于 ASEG 和 CSEG 属于同一段组，故共用同一起始地址为参考点，这时就以 ASEG 段的起始地址为参考点，因而标号 asym 和 csym 的偏移量都是相对于 ASEG 段起始段地址而言的。由于 BSEG 是一个独立段，因而 bsym 标号的偏移地址量是相对它所在的段 BSEG 的段地址而言的。

15.2.5 定义内存模式伪指令

使用简化段定义伪指令时，必须事先说明用户程序要使用的内存模式，内存模式是指编译时用户程序的数据及代码存放的格式及占用内存的大小，汇编程序中的内存模式和C中规定的内存模式是一致的，Turbo C规定的内存模式我们已在第5章的5.1内存模式(编译模式)中作了介绍，共有六种，从汇编程序的角度看，可作如下的叙述：

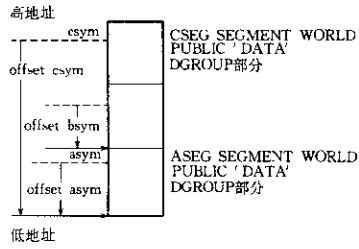


图 15.2 段组 DGROUP 和独立段偏移地址

微小模式(Tiny)——程序中的数据及代码均放在同一段内，这也就是后缀为.COM的程序。它不能用简化段定义。

小模式(Small)——程序中的数据放在64K的数据段内，代码放在64K的代码段内，因而对代码和数据的访问可通过近程(NEAR)调用来实现。

中模式(Medium)——所有数据放在64K的一个数据段内，而代码量大于64K，因而可在不同段内，这样数据是近程的，而代码是远程的。

紧凑模式(Compact)——所有代码放在一个64K的代码段内，而数据区可以大于64K，这样对代码的访问是近程的，而数据是远程的。

大模式(Large)——数据和代码均大于64K，因而都是远程的，

巨模式(Huge)——代码和数据都大于64K，并且数据数组也可以大于64K，这样代码、数据和数组指针都是远程的，它和大模式的差别是大模式数据数组不能大于64K，但两者的段结构是一样的。

一个独立的汇编语言程序可以采用上述的任一模式，一般用小模式就够了，有时独立的汇编语言程序可以通过对过程及变量的近程、远程说明而得到一种混合模式。

当C语言程序调用汇编语言子程序时，该子程序的内存模式一般应和C语言程序的内存模式保持一致。

伪指令.MODEL用于定义内存模式，它应放在其它段定义的伪指令之前，其格式是：

.MODEL 内存模式

其中内存模式是指上述的TINY SMALL、MEDIUM、COMPACT、LARGE或HUGE。例如：

.MODEL SMALL

15.2.6 段名的缺省名

使用简化段伪指令时，每个段都有一个缺省名。当整个程序都用简化段定义时，段的名字没有必要知道。若完整的段定义和简化段定义在同一个程序中都出现，则可能需要知道每个段的名字，表15.1列出了各个段在不同内存模式下的缺省段名(NAME)、边界类型(ALIGN)、结合类型(COMBINE)、类型(CLASS)、及段组名(GROUP)。从表中可以看出，在中内存模式和大内存模式时.CODE伪指令表示的缺省段名为name_TEXT，即name是这个段名可变的部分，当程序模块有一个具体名字时，name就表示这个名字，.FARDATA

和.FARDATA? 伪指令使用的缺省名在各种模式下,可以替换。

表中列出的段组名,表示了段所属的段组名,我们前面已提及:在用简化段定义时,实际上就完成了 ASSUME 语句和 GROUP 语句,即在小内存模式和紧凑模式下,完成了如下的语句:

ASSUME CS: TEXT,DS: DGROUP.SS: DGROUP

在中、大和巨模式下,完成如下的语句:

ASSUME CS: name_TEXT,DS: DGROUP SS: DGROUP

当一个程序中用到了所有段时,则完成了如下的 GROUP 语句:

DGROUP GROUP_DATA,CONST, BSS,STACK

表 15.1 各种内存模式下的缺省段名及类型

Model	Directive	Name	Allgn	Combine	Class	Group
Small	.CODE	TEXT	WORD	PUBLIC	'CODE'	
	.DATA	DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP
Medium	.CODE	name TEXT	WORD	PUBLIC	'CODE'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP
Compact	.CODE	TEXT	WORD	PUBLIC	'CODE'	
	.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
	.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
	.DATA	DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP
Large or huge	.CODE	name TEXT	WORD	PUBLIC	'CODE'	
	.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
	.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
	.DATA	DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP

从表 15.1 中可以看出,各内存模式下的缺省段名及类型和 Trubo C 编译模式下的缺省段名及类型是一致的,这就为两种语言的混合编程带来了方便。

15.2.7 定义段次序

MASM 可以按照源程序中各段出现的次序来排列目标文件中各段的先后次序,也可以按照段名的字母顺序来排列次序,缺省情况是按段出现的顺序来排列,编程者也可以用伪指

令来定义段序。这些伪指令是：

1. .ALPHA——按字母序对段排序。

2. .SEQ——按段出现的顺序排序。

3. .DOSSEG——按 DOS 定义的段序来排序段，DOS 定义的段序如下所列，其排列顺序是：

① 类型名为'CODE'的段；

② 不是'CODE'的段，且不是 DGROUP 段组中的独立段；

③ 属于 DGROUP 中的段，它们的次序是：a）类型为 BEGDATA 的段（这是 MICROSOFT 保留的段类型名），b）不是 BEGDATA、BSS、STACK 的段，c）BSS 类型段，d）STACK 段。

Microsoft 高级语言的编译程序都按此约定来进行段的排序，Turbo C 的编译系统也按此约定来进行段的排序。

伪指令 DOSSEG 优先级高于任何定序伪指令，当在源程序开头使用这条伪指令时，对源文件中段的次序安排就不必按照一定要求的次序来编了。对于大多数独立的汇编语言程序，段的次序并不重要，因而只要写上 DOSSEG 伪指令就可以了。

15.2.8 一个用简化段定义的汇编程序标准框架

下面的标准程序框架和 15.2.2 中的类似，在第三行采用 .386 伪指令使得可对 80386，80286 指令进行汇编并在 386 机上运行。当然程序中若没有 80286，80386 增加的指令或增强了功能的指令，也不影响在 8088 机上运行。

```
        DOSSEG
        .MODEL SMALL
        .386
        .STACK 100H
        .DATA
          ⋮
        .CODE
main    PROC FAR
        PUSH    DS
        SUB     AX,AX
        PUSH    AX
        MOV     AX,@DATA
        MOV     DS,AX
          ·
          ·
        RET
main    ENDP
        END main
```

在上面的框架中，DOSSEG 伪指令告诉汇编程序用 DOS 定义的段次序来设置段序，由于用 .MODEL 伪指令定义了采用小型内存模式进行编译，因而可用简化段定义伪指令来

定义段，框架中按习惯用 .STACK 定义了一个 256 字节的堆栈，一个用 .DATA 定义的数据段(段中的数据可按需要填上)，一个用 .CODE 定义的代码段，由于用了简化段，所以实际上就完成了 ASSUME 语句和 GROUP 语句。段中定义了一个远过程，保存了 DS，并将数据段地址送入了 DS，其中@DATA 是 DATA 段的等价名，实际上它就是 DATA 段的段地址。

15.3 能被 C 程序调用的一个汇编子程序框架

由于 C 编译系统要求约定的段序，要求规定的段组结合，故要编制能被 C 调用的汇编子程序，则要严格按照 C 的约定来设计程序的结构，否则将不能被正确调用。

15.3.1 可被 C 调用的一般程序结构

下面是可被 C 调用的一般程序结构

```
<code>    SEGMENT     BYTE PUBLIC 'CODE'
          ASSUME      CS:<code>,DS:<dseg>
            ⋮
          代码
            ⋮
<code>    ENDS
<dseg>    GROUP       _DATA,_BSS
<data>    SEGMENT     WORD PUBLIC 'DATA'
            ⋮
          初始化的数据
            ⋮
<data>    ENDS
_BSS      SEGMENT     WORD PUBLIC 'BSS'
            ⋮
          未初始化的数据
_BSS      ENDS
          END
```

在该结构中<code>，<data>，<dseg>要根据存储模式，换成相应的名字，按 Turbo C 规定，必须按如下约定进行替换：

内存模式	替换名字
微、小、紧凑模式	<code>→_TEXT
	<data>→_DATA
	<dseg>→DGROUP
中、大模式	<code>→文件名_TEXT
	<data>→_DATA
	<dseg>→DGROUP
巨模式	<code>→文件名_TEXT
	<data>→文件名_DATA
	<dseg>→文件名_DATA

上述这种结构是基于如下原因：

1. C 编译系统将不同类型的变量保存在不同的几个段内，即

_BSS 段——存放未初始化的静态变量(除去在源文件中说明为 far 或 huge 的数据)

_DATA 段——存放所有未初始化的全局变量和已初始化的全局变量或静态变量。

FAR_DATA——存放已初始化且被说明为 far 的全局或静态变量。

FAR_BSS——存放未初始化的且被说明为 far 的全局或静态变量。

STACK——存放局部变量和函数参数

CONST——只读的常数。

_TEXT——存放程序的执行代码。

又将 DATA、_CONST、_BSS 和 STACK 段组合成一个 DGROUP 段组。

2. C 编译系统对不同的段的边界类型、结合类型及类型规定了统一的命名：如表 15.2 所列：

表 15.2 各内存模式下段名、边界类型及结合类型和类型

编译模式	段名	边界类型	结合类型	类型
小/中	_TEXT/name _TEXT	WORD	PUBLIC	'CODE'
	_DATA	WORD	PUBLIC	'DATA'
	CONST	WORD	PUBLIC	'CONST'
	_BSS	WORD	PUBLIC	'BSS'
	STACK	PARA	STACK	'STACK'
紧凑/大/巨	_TEXT/name _TEXT	WORD	PUBLIC	'CODE'
	FAR_DATA	PARA	PRIVATE	'FAR_DATA'
	FAR_BSS	PARA	PRIVATE	'FAR_BSS'
	_DATA	WORD	PUBLIC	'DATA'
	CONST	WORD	PUBLIC	'CONST'
	_BSS	WORD	PUBLIC	'BSS'
	STACK	PARA	STACK	'STACK'

所以在上述的可被 C 调用的汇编程序结构中段名、段的边界类型，结合类型及类型均和 15.2 表中所列一致，且将 DATA，BSS 定义为同一个段组，程序中的段序也和 C 编译要求的段序一致。

在该结构中未定义 CONST 段和 STACK 段，因 C 调用汇编子程序时，常量和局部变量及参数均放在 C 程序的 CONST 段和 STACK 段中，故在汇编子程序中无需再设置 CONST 和 STACK 段了，若需要时，当然也可设置，如驻留程序设计中，为了进行堆栈切换，便设置了堆栈段和 CONST 段。

实际上汇编语言子程序可根据需要设置段，仅有一个代码段也是可以的。

15.3.2 按 C 编译要求段序的一个汇编子程序框架

还有一种可采用的格式是：汇编程序开始按 C 编译要求的段序进行段描述，然后进行段组描述，接着便是按照传统的汇编语言程序格式进行程序设计，如：

```
TEXT    SEGMENT    BYTE PUBLIC 'CODE'    ;段描述
```

```
_TEXT    ENDS
_DATA    SEGMENT    WORD PUBLIC 'DATA'
 DATA    ENDS
CONST    SEGMENT    WORD PUBLIC 'CONST'
CONST    ENDS
_BSS     SEGMENT    WORD PUBLIC 'BSS'
_BSS     ENDS
DGROUP   GROUP        DATA,CONST,_BSS;  ;组描述
         ASSUME    CS:_TEXT,DS:DGROUP,SS:DGROUP
STACK    SEGMENT                              ;传统格式
          ⋮
STACK    ENDS
_DATA    SEGMENT
          ⋮
_DATA    ENDS
_BSS     SEGMENT
_BSS     ENDS
_TEXT    SEGMENT
          ⋮
_TEXT    ENDS
         END
```

15.4 由 Turbo C 自动产生的一种汇编语言程序结构框架

由 Turbo C(也可以是 Turbo C++、BorLand C++)可以自动产生可被其调用的汇编程序结构框架,用户只要在框架中有"@1:"处填入该过程需要的汇编指令、进退栈及参数传递指令即可。

产生该框架的方法是用命令行编译程序 TCC.EXE 的设置开关-S 来产生,具体步骤如下:

首先用 Turbo C 写一个要由 C 调用的函数,该函数是空的,该函数名实际上就是要由 C 调用的汇编语言子程序的名字,如:

```
subproc()
    {
    }
```

然后压 F2 键将其存盘,退出 TC 集成环境。

接着用 TCC 进行编译,在 DOS 下打入命令行:

TCC -S subproc ↵

编译后便自动生成一个 subproc.asm 的汇编程序,实际上它也是空的,仅是一个结构空框架而已,其框架如下:

```
        ifndef    ?? version
```

```
?_debug   macro
          endm
          endif
          ? debug  S "subproc.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 E9388877090973756270726F632E63
_BSS      ends
_TEXT     segment  byte public 'CODE'
;         ? debug  L  1
_main     proc     near
@1:
;         ? debug  L  3
          ret
_main     endp
_TEXT     ends
          ? debug  C E9
_DATA     segment  word public 'DATA'
s@        label    byte
_DATA     ends
_TEXT     segment  byte public 'CODE'
_TEXT     ends
          public   _main
          end
```

若用 Turbo C 写的调用函数带有参数，则此时用 TCC 进行编译后，将产生如下所示的程序框架，设该函数说明为：

```
int asm2(int a, int b);
```

并在 TC 下建立了一个空函数，用 TCC 命令行进行编译

```
TCC -s asm2 ↵
```

便生成如下的框架，可以看出它在 asm2 过程中增加了 push bp、mov bp,sp 和 pop bp 的操作，因为该过程取参数将通过 bp 寄存器进行，所以要将它的原值保存起来。其它部分和不带参数的框架程序一样，即只要在@1 处填入具体内容即可。

```
        ifndef  ?? version.
? debug macro
        endm
        endif
        ? debug  S "asm2.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 E9CF567C1D0661736D322E63
_BSS    ends
_TEXT   segment  byte public 'CODE'
;       ? debug  L 1
 asm2   proc     near
        pusb     bp
        mov      bp,sp
@1:
;       ? debug  L 3
        pop      bp
        ret
_asm2   endp
 TEXT   ends
        ? debug  C E9
_DATA   segment  word public 'DATA'
s@      label    byte
_DATA   ends
_TEXT   segment  byte public 'CODE'
_TEXT   ends
        public    asm2
        end
```

15.5 用简化段定义的汇编语言子程序

若用 Microsoft 公司的 MASM 5.0 编译系统，则可用简化段定义的形式，构造一个可被 C 程序调用的汇编子程序，此时有几点必须注意：

1. 不需要说明段序，它的段序可按照 C 程序要求的段序进行设置，当然也可用 DOSSEG 说明，而用传统的段序编程。

2. 必须定义存储模式，用 .model 伪指令。

3. 可以不定义栈段(STACK)，使用 C 语言程序编译后的栈段。

```
.Model Small
.Code
   ⋮
代码
.data
   ⋮
初始化数据
   ⋮
.data?
   ⋮
未初始化数据
```

其中各个段的缺省名字，将按照内存编译模式自动设定，它将和 C 中相应内存模式的各缺省段名保持一致，各个段的段组 DGROUP 的设定也是自动进行，同时也自动完成了 ASSUME 语句进行段寄存器对应关系的设置。这些都符合 C 的要求。

现在有人把传统的汇编语言程序框架称为过时的格式，而将用简化段定义的汇编语言框架，称为现代的汇编语言程序格式。

在编写一个具体的能被 C 程序调用的汇编语言子程序时，要严格遵守 C 程序编译规则的约定，除了汇编子程序框架正确外，还应注意，在 C 程序和汇编子程序相互间调用函数和变量时，如何约定：名字是如何规定的；当在汇编子程序中要取得由 C 程序传递过来的参数，是如何进行堆栈操作的；当由汇编子程序传回结果值时，这个值是如何返回给 C 程序；另外在汇编子程序中如何使用 8086 各寄存器，又不会影响 C 程序的正常运行，这都需要按约定进行，下面将一一进行介绍。

15.6 编写汇编语言子程序的几个问题

15.6.1 变量和函数的相互调用

C 程序可以调用汇编语言编的子程序(或称过程或函数)和在汇编语言中定义的变量，汇编语言也可调用 C 语言编的函数和定义的变量，但要注意的是，由于 C 编译后的目标文件自动地在函数名和变量名前加了一个下划线，这是因为编译系统为了防止和它自己使用的内部函数和变量名发生混淆而造成错误，所以在汇编语言中调用 C 语言的函数和变量时，应在函数名和变量名前加一下划线，在汇编语言程序的开始部分，应对调用的函数和变量用 EXTERN 加以说明，其格式为：

```
extern  _函数名：函数类型
extern  _变量名：变量类型
```

其中函数类型指明该函数是一个近程函数或是一个远程函数(即处在另一个段中)，分

别可表示为 near 型或 far 型，而变量类型指该变量的数据类型，它和 C 语言中的数据类型具有如下表 15.3 所列的对应关系：

表 15.3 C 语言和汇编语言数据类型对应关系

数据类型		数据长度(字节数)
C 语言	汇编语言	
Char	DB BYTE	1
int	DW WORD	2
Long	DD DWORD	4
float double	DQ DWORD	8

如调用 C 程序中名为 myfunction()的函数，和变量 sign，它们在 C 程序开始说明部分为：

```
int myfunction(void);
    int sign, jarray[10];
    char ch;
    long result;
```

在调用它的汇编程序中则在程序开始说明为：

```
extern _myfunction:near
extern sign: WORD, _jarray: WORD,ch: BYTE,result: DWORD
```

若 C 程序调用汇编语言中的过程(函数)和变量，则汇编语言中应用 public 进行说明，且函数名和变量名前应带有下划线，即函数名和变量名的第一个符号应是一下划线(可看作它是函数名或变量名的一部分)，如：

```
public _myproc
public _ac
   ⋮
```

而在 C 语言中则应将其说明为 extern，即

```
extern myproc(void);
extern ac;
```

注意在函数名和变量名前不应有下划线。由于 Turbo C 对大小写是加以区别的，而在 C 中一般用小写书写程序，因而在汇编语言中，被 C 调用的函数名和变量名应尽量用小写，若要让 C 取消对大小写的区别，则应在 C 程序与汇编语言程序进行编译连接前(按 F9 前)，选中集成开发环境中的 options 选择项中的 Linker 子菜单项，将该子菜单项中的大小写敏感开关 Case-Sensitive Link 置成 off 状态，这样，以后编译连接时，C 将对大小写不加区别，汇编中的大写将被视作和小写一样。

在汇编语言子程序中，若该子程序(过程)无返回值或是整型的，则在 C 中对该函数的

类型说明可省略，如上面的说明 extern ac;extern myproc(void)。

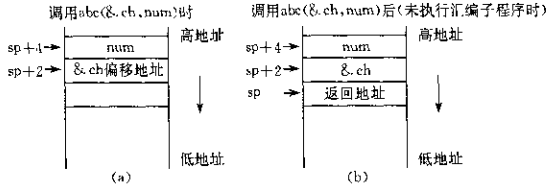


图 15.3 在小内存模式下栈的变化情况

15.6.2 参数的传递原则

C 程序调用汇编子程序时，参数是通过堆栈传给汇编程序的，如在 C 程序中说明了一个函数(用汇编程序写成)：

```
void abc(char *p1,int p2);
```

假设是在小内存模式下进行编译的，当在 C 程序中调用它时，如写成

```
abc(&ch,num);
```

则首先将 num 压入堆栈，接着将 &ch 压入堆栈，即参数是按从右到左的顺序入栈的，当开始执行调用的汇编子程序前还要将返回地址压入栈中。由于栈是向下生长的，即向地址减小的方向将数据入栈，故每压栈一次，栈指针 sp 减 2，首先压入第二个参数 num，然后压入第一个参数 &ch，接着压入要返回的地址，此时 sp 指向它。由于是小内存模式下，故入栈的地址为偏移地址(段地址同程序的段地址)，栈中变化情况如图 15.4 所示。

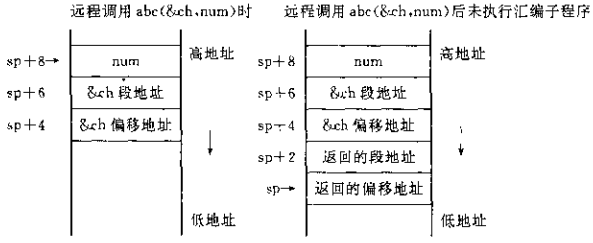


图 15.4 在大内存模式下栈的变化情况

当在大内存模式时，地址值应包括段地址和偏移地址，故地址入栈时，要压栈两次，即先压入段地址，然后压入偏移地址，当调用汇编子程序时，要说明成远程的，如写成：

```
void far abc(char far *p1,int p2);
```

这样调用时：

```
abc(&ch,num);
```

将首先压 num，然后分两次压入 &ch，即先压入段地址，然后压入偏移地址，但调用后(此时还未执行汇编子程序，即未将 BP 入栈)还要将返回地址压入栈，也是先压入段地址，然后压

入偏移地址，这个返回地址是指调用汇编子程序后，执行完该子程序应返回到C程序执行的地址。远程调用栈中地址变化情况如图15.4所示。

当汇编语言子程序(如上述的abc())要取得C程序中传递来的参数时，便用bp寄存器作为基地址寄存器，用它加上不同的偏移量来对栈中所存数据进行存取操作，由于一般C程序和调用的子程序共用一个堆栈，因此在汇编语言子程序中开始必须执行两条指令，即：

```
push bp
pop bp,sp
```

即将bp寄存器原来的值压入堆栈保存，然后将堆栈指针值sp赋给bp，用bp寄存器的值作为地址指针，以它为基地址，加上不同的偏移量来取得压入堆栈的参数，压入寄存器原来的旧值后，堆栈内容变化情况如图15.5所示：

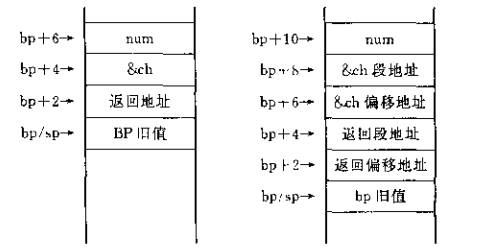


图15.5 进行push bp操作后栈的变化

例如下面这个程序例子：C程序调用一个汇编子程序power2()，传送两个整型参数x，y，然后由汇编语言子程序完成$x+2^y$计算，并将计算结果返回。程序中先执行：

```
push bp
mov bp,sp
```

由于是近程调用，这样通过下面指令：

```
mov ax,[bp+6]
mov cx,[bp+6]
```

则可将由C程序中传过来的两个参数3和5取走，因此取参数时，实际上从左到右进行。

当在汇编子程序结束时，接排一条pop bp，则可恢复bp的旧值，然后通过最后的RET语句，返回到调用处的下条语句去执行。

由于C编译程序自动进行栈的调整，所以在用RET指令返回时，不像汇编程序中要求的那样，为了调整堆栈指针，RET后面要带参数。

下面是这个程序的清单：汇编语言子程序采用简化段格式。

```
main()
```

```
{
printf("3 times 2 to the power of 5 is %d\n",power2(3,5));
}
```

汇编子程序

```
.model small
.code
        public _power2
_power2 proc
        push bp
        mov bp,sp
        mov ax,[bp+4]
        mov cx,[bp+6]
        shl ax,cl
        pop bp
        ret
_power2 ENDP
        END
```

当传递的参数是某些类型时，则首先进行转换，然后再压入堆栈，如字符型(char)参数转换成整型(int)，无符号字符型(unsigned char)转换成无符号整型(unsigned int)，浮点型(float)转换成双精度(double)，结构(structure)是整个压入堆栈的(也是按照结构成员的相反顺序)，对于数组，是将指向该数组的指针压入堆栈的，近指针为 16 位，即压入偏移地址，远指针为 32 位，先压入段寄存器值，后压入偏移地址。表 15.4 是压入参数对，转换后的类型及在栈中所占字节数。

表 15.4 参数转换后的类型及在栈中所占字节数

类型	转换后的类型	在栈中所占字节数
char	int	2
unsigned	unsigned int	2
short		2
unsigned short		2
int		2
unsigned int		2
long		4
unsigned long		4
float	double	8
near pointer		2
far pointer		4
array	指向 array 的指针	4 或 2
structure		n

15.6.3 汇编语言子程序的返回值

当被调用汇编语言子程序有值返回调用它的 C 程序时，这个值是通过 AX 和 DX 寄存

器进行传递的，即若返回值小于或等于 16 位二进制值，则该值放在 AX 寄存器中，若返回值是 32 位，则高 16 位存放在 DX 寄存器中，低 16 位存放在 AX 寄存器中。若返回值大于 32 位（如结构、实数或双精度数），则存放在静态变量存储区，而这个存储区的指针则存放在 AX 寄存器中，若指针是远程的，即是 32 位的。则 DX 存放指针的段地址，AX 存放偏移地址。

表 15.5 是返回不同的数据类型时，存入的寄存器情况：

表 15.5 返回到 Turbo C 的数据类型与存放的寄存器

数据类型	寄存器
char	AX
unsigned char	AX
int	AX
unsigned int	AX
long	高位字节在 DX 中，低字节在 AX 中
unsigned long	高字节在 DX 中，低字节在 AX 中
float	高字节在 DX 中，低字节在 AX 中
double	值的地址在 AX 中
struct 和 union	地址在 AX 中
near 指针	AX
far 指针	DX 中为段地址，AX 为偏移地址

15.6.4 汇编语言子程序中寄存器的使用

由于汇编语言中要用到 80x86 CPU 中的许多寄存器，而调用它的 C 语言程序中也要用到一些寄存器，因而这些寄存器的作用和保护则应引起注意，否则将导致程序失败。

汇编语言子程序中首先要用到的是 BP 寄存器，在 C 程序中它是作为参数和自动变量的基地址，而在汇编语言子程序中，它也是作为取参数的基地址，故在进入汇编子程序时，必须通过压栈操作将其值进行保护，汇编子程序结束时，必须通过弹出操作，恢复其原来在 C 程序中的值。即如前所说：

```
        ⋮
过程名  PROC
        PUSH BP
        MOV BS,SP
          ⋮
          ⋮
        POP BP
        RET
过程名  ENDP
        END
```

SI 和 DI 寄存器在 Turbo C 中用作存放寄存器变量的，当在汇编语言子程序中用它们时，应在用之前用压栈操作将它们的值保存起来，在返回调用者前，应将其原值恢复（通过弹

出堆栈操作),若在C程序编译前,选择了-r编译选择项,此时寄存器变量的使用被关闭,因而就可以在汇编中随意使用SI和DI寄存器了。

BX和CX寄存器可以在汇编语言子程序中任意使用,AX和DX用作存放汇编子程序的返回值,可在汇编语言子程序中使用,但若有返回值时,则要保证在子程序返回时,它们存有要返回的值。

对于CS、SS、DS、SP、IP由于用作段寄存器和堆栈指针,在程序中不能随意使用,根据编译的内存模式或far,near调用,它们可能和C程序在同一个段内,用高一个段寄存器,或不同的段内。因而可根据不同情况,有时要改变它们中的值,以得到正确段地址,但它们不能用于其它用处。

15.7 混合编程的编译和连接

若已完成了C语言程序和调用的汇编语言子程序的编程,若要成为一个能执行的程序,还需进行对它们的分别编译和连接,最后生成一个可执行文件。编译连接有两种方法可以采用:

15.7.1 在Turbo C集成环境下进行编译和连接

在TC下,进行编译和连接的过程如下:

1. 在DOS下,用宏汇编(MASM.EXE)将汇编语言子程序进行编译,即生成后缀为.OBJ的目标文件。

2. 在TC下,选择主菜单中的project项,然后选中该项中的project项中的project name子项,写入一个后缀为.PRJ的工程文件,该文件包括需编译连接的C程序和它调用的汇编语言子程序的目标文件名,该工程文件可在file菜单项下,用编辑完成,然后用F2或Write to,将该文件用后缀为.PRJ的文件名存盘。例如设两个文件分别为:

```
myprogram
mysubp.obj
```

然后用名为myprogram.prj的文件存盘,即生成了名为myprogram.prj的工程文件。

3. 用F9键对工程文件进行编译连接,由于汇编子程序已是.OBJ文件了,故仅对C程序进行编译,然后再将两个目标文件进行连接,生成一个.EXE执行文件。

编译连接时,最好关闭对大小写敏感的开关,即options中的选项Linker中的Case-sensitive Link置成off。

15.7.2 用Turbo C命令行编译程序TCC进行编译连接

在DOS下,运用传统的命令行编译方式,对源文件进行编译连接,Turbo C提供的TCC.EXE调用格式如下:

TCC -mx -I\include -L,\Lib -efile1 file2 file3…

式中-m选择项指定内存模式,其后字母x应用s、m、c、l或h替换,它们分别代表五种存储模式。

-I选择项指出了连接时所需初始化模块COX.OBJ的路径。

-L 选择项指定连接所需的库文件路径

-e 选择项指定了生成的可执行文件名，若本项缺省，则生成的执行文件名和第一个 C 源程序名相同。

file1，file2，file3 分别表示要顺序编译和连接的文件名，它们可以是汇编程序名或目标文件名，也可以是 C 源程序名(.C 后缀可省略但若是后缀为 .ASM 或 .OBJ 的汇编程序，或是目标文件名则后缀不能省)。

当 TCC 进行编译时，发现有后缀为 .ASM 的汇编程序，便立即调用 TASM.EXE 汇编程序对 .ASM 文件进行汇编，因而 TASM.EXE 必须在当前目录下，当没有时，可将 MASM.EXE 改名为该文件即可。也可事先将后缀为 ASM 的文件先用 MASM 编译生成 .OBJ 文件，然后再用 TCC 进行连接。

一般采用第一种方法，它较为方便。

15.8 混合编程实例

15.8.1 程序 1——同为小内存模式的混合编程

下面是一个从 5 个数中找出其中最小数的混合编程例，该程序调用汇编语言子程序 min_num()来完成找寻小数的工作，C 程序调用它时，共传递六个参数，即数的个数和 5 个数，由于在汇编子程序开始时进行压栈操作 push bp，所以 SP 此时指向栈内的 BP 值，SP+2 指向返回地址值，因而 SP+4 指向传递来的第一个参数(count 值)，其余依次类推，在汇编子程序中，首先将比较的个数送入 CX 中，若个数为 0，则不比较而返回，否则将第一个比较的数从[bp+6]中取出，再和[bp+8]中的第二个数比较；若小于，则再取出和第三个数比较；否则将第二个数送 AX 寄存器，用它再和第三个数比较，依此类推，直到 CX 减计数到 0 为止(这由 Loop comp 来完成)。

该程序在小内存模式下编译。作法是将汇编子程序用 TASM 编译成 .OBJ 文件，设名为 min_num.OBJ，然后作成一个 .PRJ 文件，设 C 程序名为 min.c，即制成的 .PRJ 文件名为 min.prj，其内容为

```
min.c
min  num.obj
```

将选择项 project 项的 project name 中写入该工程文件名，然后将 Linker 选择项中的大小写敏感开关置成 off，按 F9 编译连接即成。

```
#include<stdio.h>
int extern min_num(int count,int v1,int v2,int v3, int v4,int v5);

main()
{
  int i;
  i=min_num(5,7,0,6,1,12);        /* 调汇编子程序 */
  printf("The minimum of five is %d\n",i);
}
```

```
            min_num.asm     源程序如下:
            .model    small
            .code
             public _min_num
_min num    proc   near
            push   bp
            mov    bp,sp
            mov    ax,0
            mov    cx,[bp+4]          /* count 出栈 */
            cmp    cx,ax
            jle    exit
            mov    ax,[bp+6]          /* v1 出栈 */
            jmp    ltest              /* 若不是 0 */
comp:       cmp    ax,[bp+6]          /* 同下一个数比较 */
            jle    ltest
            mov    ax,[bp+6]
ltest:      add    bp,2               /* bp+2 */
            loop   comp               /* 循环进行比较 */
exit:       pop    bp
            ret
_min_num    endp
            end
```

15.8.2 程序 2——C 程序和汇编子程序是不同的内存模式

有时,也可以 C 程序和被调用的汇编语言子程序的内存模式不同,如要编一个通用的汇编子程序,可将其在大内存模式下编译,这样若 C 程序在小内存模式下时,也可进行连接,不过此时在 C 程序中要将调用的汇编子程序说明成一个远程函数,若有函数参数为指针类型时,则要说明成 far,即它包含段地址和偏移地址两部分。

下面的程序是,C 程序从键盘接收输入的整数(用 getchar()取回车),然后调用汇编子程序对接收的数进行检查,若是 0~9 之间的数,则将其转换为 ASCII 码(即程序中的 add ax,30h),并将其存在由 C 程序传递来的地址指针 &ch 指出的地址中去,并将 0 送 AX (MOV AX,0)作为返回参数。若从程序中键盘接收的是字符,则汇编子程序返回非零值,因而 C 程序继续作 do_while 循环,直到接收到的是数字为止,然后从 &ch 地址中取出显示并返回 DOS。

由于对汇编子程序是远程调用,故在汇编子程序中,取第一个传递来的参数是在堆栈[bp+6]地址([bp]是存 bp 的原值,[bp+2]是返回地址的段地址,[bp+4]是返回地址的偏移值),由于该参数是指针,故[bp+6]为指针的偏移地址部分,而[bp+8]存指针的段地址部分。因而第二个参数存在[bp+10]地址中。

```
#include <stdio.h>
int far num(char far *ch,int k);
main()
```

```
{
  int in;
  char ch;
  do
   {
     printf("\npleas input");
     scanf("%d",&in);
     getchar();
    }while(num(&ch,in));
  printf("\n%c",ch);
}
```

num.asm　　源程序如下：

```
          .model  large                    /* 大内存模式 */
           public _num
          .code
_num      proc    far
          push    bp
          mov     bp,sp
          mov     ax,[bp+8]                /* 取指针段值 */
          mov     es,ax
          mov     bx,[bp+6]                /* 取指针偏移地址 */
          mov     ax,[bp+10]
          cmp     ax,9
          ja      exit
          add     ax,30h                   /* 变成 ASCII 码 */
          mov     es:[bx],ax               /* 存放在指针指向的地址中 */
          mov     ax,0
exit:     pop     bp
          ret
 num      endp
          end
```

15.8.3　程序 3——中内存模式下的混合编程

下面的程序是 C 程序将一数组传递给汇编子程序，然后由汇编子程序采用冒泡法，对该数组从小到大排序，传递数组一般采用传地址法，即将数组的地址传递给汇编子程序，由于该程序是中内存模式下编译连接的，故如上例一样，数组指针应包括段地址和偏移地址两部分，共 4 个字节。在汇编子程序中通过[bp+6]地址取得偏移地址，而通过[bp+8]取得段地址值。在调用汇编子程序时，传递来两个参数，第一个是远程地址指针(即数组地址)，第二个参数是数组元素个数。在汇编子程序中用了 bp,ds,si,bx，故将它们压栈保存原值，返回时再弹出以恢复其原来的值。程序中将数组的段地址送 ds 数据段寄存器，而将偏移地址送 SI 源地址寄存器，这样以后对[si]地址的存取操作，均认为在同一段地址 ds 下进行。

汇编子程序返回时，返回值也是一 far 型地址指针，即包括段地址(存在 DX 寄存器中)

和偏移地址(存在AX寄存器中),该地址指针指向排序好的数组,所以在C程序中用printf("%d",*b++);依次将排序好的数组元素一一输出。

汇编子程序采用简化段的程序框架。采用一般格式的子程序在其后也附了它的程序清单。

```
            .model medium                    /* 中模式 */
             public _sort_num
            .code
_sort_num   proc  far
            push  bp
            mov   bp,sp
            push  ds
            push  si
            push  bx
            mov   ds,[bp-8]                  /* 排序数组段地址 */
            mov   bl,0ffh                    /* 排序结束标志 ffh 送 bl */
a1:         cmp   bl,0
            je    a4
            xor   hl,hl
            mov   cx,word ptr[bp+10]         /* 排序数组元素个数送 cx */
            dec   cx
            mov   si,word ptr[bp+6]          /* 排序数组偏移地址 */
a2:         mov   ax,word ptr[si]
            cmp   ax,word ptr[si+2]          /* 两元素进行比较 */
            jbe   a3                         /* 若等于或不高于则不交换 */
            xchg  word ptr[si+2],ax          /* 交换 */
            mov   word ptr[si],ax
            mov   bl,0ffh
a3:         inc   si
            inc   si
            loop  a2                         /* 下一个元素地址 */
            jmp   a1
a4:         pop   bx
            pop   si
            mov   dx,word ptr[bp+8]          /* 返回的排序数组段地址送 DX */
            mov   ax,word ptr[bp+6]          /* 返回的排序数组偏移送 AX */
            pop   ds
            mov   sp,bp
            pop   bp
            ret
_sort_num   endp
            end
    #include<stdio.h>
    int far *sort_num(int far *,int);
```

```
void main(void)
 {
  register int i;
  int a[10];
  int far *b;
  for (i=0;i<10;i++)
   a[i]=10-i;                          /* 产生待排序的数组 */
  for(i=0;i<10;i++)
   printf("%d",a[i]);
  b=sort_num((int far *)a,10);  /* 调用汇编子程序并返回排序数组地址 */
  printf("\n");
  for(i=0;i<10;i++)
   printf("%d",*b++);
  b--;
 }
```

下面是采用一般格式的汇编子程序：

```
_text     segment byte public 'code'
_text     ends
_data     segment word public 'data'
_data     ends
const     segment word public 'const'
const     ends
_bss      segment word public 'bss'
_bss      ends
dgroup    group _data,const,_bss
          assume cs:_text,ds:dgroup,ss:dgroup
_text     segment
          public _sort_num
_sort_num proc  far
          push  bp
          mov   bp,sp
          push  ds
          push  si
          push  bx
          mov   ds,[bp+8]
          mov   bl,0ffh
a1:       cmp   bl,0
          je    a4
          xor   b1,b1
          mov   cx,word ptr[bp+10]
          dec   cx
          mov   si,word ptr[bp+6]
```

```
a2:         mov   ax,word ptr[si]
            cmp   ax,word ptr[si+2]
            jbe   a3
            xchg  word ptr[si+2],ax
            mov   word ptr[si],ax
            mov   bl,0ffh
a3:         inc   si
            inc   si
            loop  a2
            jmp   a1
a4:         pop   bx
            pop   si
            mov   dx,word ptr[bp+8]
            mov   ax,word ptr[bp+6]
            pop   ds
            mov   sp,bp
            pop   bp
            ret
_sort_num   endp
_text       ends
            end
```

15.9 Turbo C 程序中内嵌汇编指令行

C 程序中还有一种和汇编语言混合编程的方法，即可在C 程序中嵌入汇编语言的指令，它就好像 C 程序的语句一样，当 C 语言程序中想直接控制硬件或加快运行速度，但可用较短的汇编程序实现时，可采用这种方法，它实现的方法是：

1. 在嵌入的汇编指令前，必须用关键字 asm 说明，其格式为：

asm <操作码> <操作数> <;或换行>

其中操作码是有效的 8086 指令及某些汇编伪指令 db，dw，dd 及 extern。操作数可以是 C 语言中的常数、变量和标号，也可以是操作码可以接收的数，内嵌的汇编指令用分号或换行作为结束。

注意！不能用像 MASM 中规定的那样，用分号作为一条注释的开始，必须用/*…*/来标识注释，一条汇编指令不能跨越两行，但允许一个文本行中可有多条汇编指令。

例如：

```
    ⋮
asm mov ax,ds; /*DS→AX*/
asm pop ax;
asm pop ds;
asm iret;
asm push ds;
```

```
    ⋮
```

又例如：

```
myfunc()
  {
  int i,x;
  if(i>0)
  asm mov x,4
  else i=7;
  }
```

在这个例子中，汇编指令 mov 前用 asm 关键字作了说明，因而该语句被看作一条 C 语句来对待，其后以换行作为语句的结束。

前面已介绍过可被 C 语言程序调用的汇编语言子程序的编制方法，若子程序不太大时，也可完全用内嵌汇编指令的 C 程序函数来实现它，例如求两个整数中的小者，可编成 C 语言的函数形式：

```
int min(int v1,int v2)
    {
      asm mov ax,v1
      asm cmp ax,v2
      asm jle minexit
      asm mov ax,v2
     minexit:
       return( ax);
     }
当要初始化系统板上的一个端口时也可写成函数形式
init _port()
  {
    printf("InitiaLizing port \n");
   asm out 26,255
   asm out 26,0
  }
```

当写成 C 程序函数形式时，它可以在各种编译模式下通过，如同用 C 语句编写的函数一样。

汇编指令集可内嵌在函数体中，如上面的 min()函数那样，此时编译时，它将被放入代码段内，也可作为外部说明放在函数体外，这样，编译时，它们将被放入数据段内。

2. 内嵌式汇编指令中的操作数

内嵌式汇编指令中的操作数可以是 C 语言程序中的符号，当编译时，它会被自动转换为适当的操作数，当然在那些能使用寄存器的指令中，才能用寄存器变量，在那些要用到地址操作的指令中，才能用于地址有关的符号。这里所说的 C 程序中用到的符号指自动变量，寄存器变量，函数参数等。

当汇编指令中使用寄存器名时，则大小写不分，并只能是 8086 的寄存器名。

在C语言程序中，DI和SI两个寄存器常被用作寄存器变量，而只能是short，int(或unsigned)型的变量能被定义成寄存器变量，若过多的用register定义寄存器变量，或将register放在不适合作寄存器变量的数据类型前，则多出的变量不会成为寄存器变量，或者用register定义将不起作用。若在函数中没有指定寄存器变量，则在汇编指令中可任意使用SI和DI，因为在C程序中调用这个函数和退出这个函数时，SI和DI的值会自动被保护和恢复的。若在函数内用了寄存器变量(即进行了寄存器变量的定义)，若再次使用SI和DI，将可能改变寄存器变量的值，这一点要注意。

3. 汇编指令操作数也可以是结构数据

内嵌式汇编指令的操作数可以是C程序中的结构成员，其书写形式可以是：

结构变量．结构成员

例如有一结构：

```
struct mystruct {
    int a;
    int b;
    int c;
    } myA;
```

如在函数myfunc()中的汇编指令用其结构成员作为操作数：

```
myfunc()
  {
    ⋮
  asm mov ax,myA.b
  asm mov bx,[di].c
    ⋮
  }
```

其中第一条内嵌汇编指令将myA.b赋给了ax寄存器，第二条内嵌汇编指令把DI中装的数作为地址再加上myA结构中的c成员对myfunc结构的位移量形成一个新的地址作为操作数。

当汇编指令操作数使用结构的成员时，也可用各成员相对该结构的位移量作为操作数，如同一个常量操作数一样，如上面的汇编指令可写成：

```
myfunc()
  {
    ⋮
    asm mov ax,DGROUP:myA+2
    asm mov bx,[di+4]
    ⋮
  }
```

由于结构中各成员项为整型的，故b成员位移为2，c成员位移量为4。

若将mystruct结构的地址放入di中，则可直接访问结构的成员(即直接对其所在地址进行访问)，如[di-4]，就表示将该成员(c)的值取出。

4. 转移指令的执行

内嵌汇编指令可以使用任何有条件或无条件转移汇编指令，Loop 循环指令。但它们只能在函数体内有效，不允许进行段间转移。由于 asm 语句中不能给出标号，因而转移指令只能使用 C 语言程序中 GOTO 语句使用的标号，如：

```
int fun1()
  { int a;
     ⋮
   a:
     ⋮
    asm jmp a
     ⋮
    }
```

当执行到第一个内嵌汇编语句时，程序将转到 a：标号指向的语句去执行。

转移目标也可以是间接转移，如：

```
int fun()
   { int a;
      ⋮
    a:
      ⋮
     asm jmp[a]
      ⋮
   }
```

此时当执行到第一个内嵌式汇编语句时，程序将无条件转移到由 a 整型变量的值作为地址的目标去。

15.10 内嵌汇编指令的 C 程序编译连接方法

内嵌汇编指令的 C 程序只能采用 TCC 命令行的编译连接方法，用 TCC 命令行实现编译连接的方式是：

TCC -B-L：\LIB　文件名　库文件名

其中-L 选择项指定了连接所需的库文件路径，文件名指有内嵌汇编指令的 C 程序名，库文件指程序中要用到的库函数所在的库文件(Turbo C 标准库可省略)。

有内嵌汇编指令的 C 程序进行编译时，必须要有-B 选择项，否则编译时，一旦遇到汇编代码，便立即给出警告信息，并以-B 选择项重新进行编译，若在 C 程序中加上 #program inline 语句，则可省掉-B 选择项。

由于汇编时，TCC 要调用 TASM.EXE 程序，若无此程序，可将 MASM(3.0 以上)改名为 TASM.EXE 以代替之。

15.11 嵌入汇编指令行的程序例

1. 下面是一个从两数中选小数的程序，选小数的操作用函数 min()来实现。在 TC 集成环境下编此程序，然后退出 TC，在 DOS 下(注意不是 TC 的 DOS SHELL 下)用命令行编辑程序编译连接该程序，即输入命令：

c:\TC> TCC -B ASMC1 ↵

即生成 ASMC1.EXE 执行文件，由于 lib 库、C 程序均在当前 TC 子目录下，故 TCC 后的选项仅留-B

```
#include<stdio.h>
int extern min(int v1,int v2);
main()
 {
  int i;
  i=min(6,8);
  printf("The minimum of two is %d\n",i);
 }
int min(int v1,int v2)
 {
  asm mov ax,v1;
  asm cmp ax,v2;
  asm jle minexit;          /*若小于转移*/
  asm mov ax,v2;
  minexit:
  return(_AX);              /*返回小数*/
 }
```

该程序也可改写为将函数操作变成主函数中的语句行，不再调用函数，这更直接，对 PC 机的硬件控制操作往往采用这种方式，如下面是将上述程序改写后的一个程序，当用户输入比较的两个数后，立即给出最小数，这个程序中汇编指令 mov i,ax，是将 ax 中的数送给 i 变量，因它们都是 16 位的数，其中一个操作数是寄存器，i 是整型变量，若采用 mov i,v1 将不会被编译，因汇编指令格式中 mov 指令的两个操作数不允许均是变量。

```
main()
{
  int i,v1,v2;
  printf("Pleas input tow degit ");
  scanf("%d %d",&v1,&v2);
  asm mov ax,v1;
  asm cmp ax,v2;
  asm jle minexit;        /* 若 v1<=v2 则转移到 minexit 去执行 */
  asm mov ax,v2;
  minexit:
```

```
  asm mov i,ax;          /* i为最小数 */
  printf("The minimum of two is %d\n",i);
}
```

2. 下面是两个整数相乘的程序，其中用到了无符号乘汇编指令 mul，它是一个隐含目的操作数与一个源操作数(如在 CL 中)相乘，对 16 位目的操作数则隐含在 AX 寄存器中，结果积在 DX：AX 中，对 8 位目的的操作数，则隐含在 AL 寄存器，而积在 AX 中，在这个程序中，假设积不超过 16 位，故执行 mov i,ax 后，i 变量中便存放的是积了。

```
main()
{
  int i,v1,v2;
  printf("Pleas input tow degit ");
  scanf("%d %d",&v1,&v2);
  asm mov ax,v1;
  asm mov cl,v2;
  asm mul cl;        /* 做AX*CL→AX运算 */
  asm mov i,ax;      /* 乘积送i */
  printf("The mul of two is %d\n",i);
}
```

3. 下面是一个有关图形的程序，程序开始将屏幕设置成 VGA：640×480. 16 color 方式，并用 bar3d 函数画出一个立体方块来，然后用汇编指令进行 BIOS 调用，将屏幕设置成 320×200 分辨率的 256 种颜色方式，并画出 256 种不同颜色的竖线，然后调用函数 fun1() 通过 BIOS 调用，将屏幕设置成 VGA 640×480 的 16 色方式，并在屏幕中央画出一个红方块来。

程序中用到的 sleep(s)函数是用来延时，以使图形暂时保留 s 秒。

```
#include <dos.h>
#include <graphics.h>

main()
  {
      int i=256;
      int driver=VGA;
      int mode=VGAHI;
      initgraph(&driver,&mode,"");
      setbkcolor(BLUE);
      setcolor(WHITE);
      setfillstyle(1,RED);
      bar3d(100,200,200,300,100,1);
      printf("In Turbo C VGA:640*480,16 color");
      sleep(3);              /* 延时3秒 */
      asm mov bl,i;          /* BIOS功能调用,设置屏幕图形方式 */
      asm mov ah,0;
```

```
        asm mov al,13h;
        asm int 10h;
        asm mov cx,290;          /* 画 256 条各种颜色线 */
lop:    asm mov dx,199;
loop:   asm mov ah,0ch;
        asm mov al,bl;
        asm mov bh,0;
        asm int 10h;
        asm dec dx;
        asm cmp dx,0;
        asm jne loop;
        asm dec cx;
        asm dec bl;
        asm cmp bl,0;
        asm jne lop;
        printf("Assemhly VGA:320 * 200,256 color");
        sleep(3);
        funl();
        sleep(3);
        closegraph();
        printf("The end");
        sleep(1);
    }
 funl()
   {
        asm mov ah,0;
        asm mov al,12h;
        asm int 10h;
        asm mov dx,240;
 lop1:  asm mov cx,440;          /* 画方块 */
 lop2:  asm mov ah,0ch;
        asm mov al,4;
        asm mov bh,0;
        asm int 10h;
        asm dec cx;
        asm cmp cx,200;
        asm jne lop2;
        asm dec dx;
        asm cmp dx,120;
        asm jne lop1;
        printf("Asemmbly VGA:640 * 480,16 color");
      }
```

15.12 汇编语言程序调用C函数

有时需要用汇编语言程序调用C函数，以便利用C语言许多特点，有时混合编程时，由C程序启动程序运行，然后调用汇编语言子程序，在执行汇编语言子程序过程中，又调用C函数。

在汇编语言程序调用C函数时，也要按有关约定编程：

1. 由于C程序编译后，在所有变量名和函数名前加一下划线，所以在汇编语言程序中，对使用C语言的函数和变量在其名字前均应加下划线，如调用一个C函数cfunction()，则应写作：

```
call _cfunction
```

2. 要对调用的C函数用关键字EXTERN进行说明。

若调用的C函数为NEAR型，EXTERN说明可放在代码段中，若C函数为FAR型，EXTERN说明要放在所有段之外。如：

```
EXTERN _cfunction: NEAR
```

3. 对汇编语言中使用的C函数中变量也要用EXTERN进行说明，说明格式是：

```
EXTERN 变量名:size
```

SIZE指变量的字节数，它们与Turbo C中数据类型长度的变换关系如表15.3中所示，如在C中定义：

```
int i,array [10];
char cb;
```

若在汇编程序中使用这些变量时，应说明为：

```
EXTERN _i:WORD, _array: WORD,_ch: BYTE
```

当使用巨模式编译时，EXTERN说明必须放在所有的段外。

4. 参数传递

可用两种方法传递参数，即在C程序中定义变量，在汇编语言程序中将它说明成EXTERN型，在程序中将值传给该变量，然后C也可使用该变量，此时该变量具有的值就是汇编程序中传给的值，如在C程序中已说明了一个变量：

```
int n;
```

在汇编中被说明为

```
extern _n:WORD
```

并赋值

```
mov -n,3
```

这样，若C中使用变量n时，它便具有值3了。

另一种传递参数是采用堆栈，如前所述，C程序调用函数时，函数参数是通过堆栈，按参数从右到左的顺序压入堆栈的，而函数取得这些参数是从左到右进行，这也正是由于堆栈有先进后出的出栈原则，所以C调用函数传递参数采用堆栈。因此在汇编程序中要向C函数传递参数时，要将最后取走的参数先入栈，把最先要取走的参数最后入栈，即入栈顺序应按C函数参数序列的反序进行，即从右到左顺序。在将长型参数(如指针等)压栈时先压高位部

分(如段地址),再压入低位部分(如偏移地址)。

在调用C函数完成后,应该对栈指针进行调整,以恢复原指针的值.如:

add sp,size

这里,size的值为:

size=程序返回地址所占字节数+向C函数传送参数所占的字节数

15.13 汇编语言调用C函数例

下面是两个示例程序,它们都是以C语言程序开始执行,因而实际上混合程序的初始化工作由C程序完成,接着调用汇编语言子程序,在汇编语言子程序中完成某些处理后,再调用C语言编的函数继续完成某些功能,程序最后在汇编语言子程序中结束。前一个程序,不传递参数,后一个程序由汇编子程序用参数调用C语言函数。

15.13.1 程序1——无参调用

该程序首先由C程序启动运行,在屏上显示调用的信息,然后调用汇编子程序acal(),该子程序执行时,显示printed by assmbler:字样,然后对数字4进行处理,并显示处理结果,当处理后超过9时,便显示回车换行,再调用C函数accl()。由该函数返回后,用add sp,2对栈指针进行调整,因返回地址为2个字节(为偏移地址,因采用小内存模式),故sp+2。程序最后以RET结束。

```
extern acal();
main()
{
  printf("C calling MASM,then MASM call C\n");
  acal();                    /* 调用汇编子程序 */
}
accl()
{
  int i;
  printf("follows are printed by c language:\n");
  for(i=0;i<5;i++)
   printf("i=%-4d",i);
  printf("\n");
}
```

acl.asm 源程序如下:

```
        extrn _accl:far
        .model small
        .data
string  db      "printed by assembler:",0dh,0ah,'$'
        .code
        public  _acal
_acal   proc
```

```
        mov     dx,offset string        ;显示字符串调用
        mov     ah,9
        int     21h
        mov     bl,4
        push    bx
again:  mov     dl,32                   ;空格 ASCII 码
        mov     ah,2                    ;显示字符调用
        int     21h
        pop     bx
        mov     al,bl
        inc     al
        push    ax
        daa                             ;十进制调整
        pop     cx
        cmp     cl,9
        jg      stop                    ;大于 9 则转 STOP
        and     al,0fh
        mov     bl,al
        push    bx
        or      al,30h
        mov     dl,al
        mov     ah,2
        int     21h
        jmp     short again
stop:   mov     dl,0dh                  ;显示回车
        mov     ah,2
        int     21h
        mov     dl,0ah
        mov     ah,2                    ;显示换行
        int     21h
        call    _accl                   ;调用 C 函数
        add     sp,2
        ret                             ;堆栈恢复
_acal   endp
        end
```

15.13.2 程序 2——有参调用

该程序和上面的程序类似，只不过当调用 C 语言函数时，传递了 5 个参数，这 5 个参数由调用的 C 语言函数 acc2()进行显示。在汇编子程序中按调用 C 函数中参数的顺序，反序将各参数依次入栈，即先入栈 4，然后 3、2、1、0，这样当调用 C 语言函数 acc2(n0,n1,n2,n3,n4,n5)时，acc2 函数中的参数 n1 将取 0，n2 取 1，依次类推，n5 取 4，而参数 n0 实际上是故意设的虚变量，它实际上将取得返回地址，因 C 函数中不用它，所以实际上目的是使指针跳

过堆栈中返回地址的 2 个字节，而能正确取得接着存放的 5 个参数。

由于参数传递完后栈指针已减 12(因采用小内存模式)，故恢复栈指针时，应 add sp，12，栈中存放的内容如图 15.6 所示。

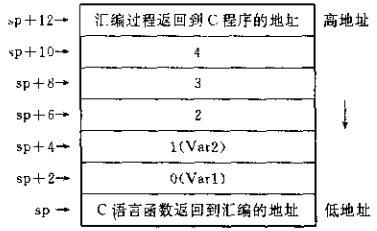


图 15.6　汇编调用 C 函数时的堆栈

```
extern aca2();
main()
{
  printf("C calling MASM,then MASM calls C\n");
  printf("Parameters are passed by MASM to C\n");
  aca2();
}
void acc2(int n0,int n1,int n2,int n3,int n4,int n5)
{
  printf("follows are printed by c language:\n");
  printf("n1=%-4d n2=%-4d n3=%-4d",n1,n2,n3);
  printf("n4=%-4d n5=%-4d\n",n4,n5);
}
```

aca.asm　源程序如下：

```
            extrn _acc2:far
            .model small
            .data
string      db      "printed by assembler:",0dh,0ah,'$'
var1        equ     0
var2        equ     1
            .code
            public _aca2
_aca2       proc
            mov     ax,4
            push    ax;                 ;4 入栈
            mov     ax,3
            push    ax                  ;3 入栈
            mov     ax,2
            push    ax                  ;2 入栈
```

```
        mov     ax,var2
        push    ax                      ;1入栈
        mov     ax, var1
        push    ax                      ;0入栈
        mov     dx,offset string
        mov     ah,9
        int     21h                     ;显示字符串调用
        mov     bl,4
        push    bx
again:  mov     dl,32
        mov     ah,2
        int     21h
        pop     bx
        mov     al,bl
        inc     al
        push    ax
        daa
        pop     cx
        cmp     cl,9
        jg      stop
        and     al,0fh
        mov     bl,al
        push    bx
        or      al,30h
        mov     dl,al
        mov     ah,2
        int     21h
        jmp     short again
stop:   mov     dl,0dh
        mov     ah,2
        int     21h
        mov     dl,0ah
        mov     ah,2
        int     21h
        call    _acc2                   ;调用C函数
        add     sp,12                   ;恢复堆栈
        ret
_acc2   endp
        end
```

第 16 章

C 与 FoxBASE(dBASE)的接口技术

在实际应用中，C 往往需要与数据库语言混合进行编程，这是因为关系数据库(FoxBASE 或 dBASE Ⅲ)有很强的数据处理能力，通过这些数据库语言，很容易建立起数据库和对数据进行处理及操作，而 C 语言具有较强的计算功能和作图能力，通过它们的混合编程，可取长补短，完成实用性很强的应用程序。

当 C 语言程序与数据库语言混合编程时，往往是要对数据库进行读写操作，因此必须首先了解 FoxBASE 或 dBASE Ⅲ 数据库文件的结构，下面进行介绍：

16.1 C 程序直接读取 FoxBASE 数据库中的数据

16.1.1 FoxBASE(dBASE)数据库文件结构

FoxBASE(dBASE)数据库文件的结构可分为两部分，第一部分为库本身结构说明部分，第二部分才是数据库本身的数据内容。

库结构说明部分又分为两个部分，前一部分(共 32 个字节)是关于整个数据库的结构说明，从第 33 个字节开始为各个字段结构的说明，库结构示意如图 16.1 所示，其中库说明部分的前 32 个字节是库结构说明，其具体内容与含义如表 16.1 所列。

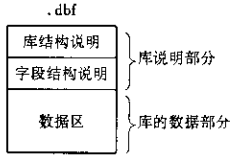


图 16.1 库结构

表 16.1 库说明部分的具体内容

字节	内容	含义
1	83H 或 03H	DBF 的标志，若含 Memory 字段则为 80H，否则为 03H
2～4	日期	文件建立或修改的最后日期(年、月、日)
5～8	数据库记录个数	数据库所含记录个数，低位在前，高位在后。
*9～10	库说明部分的长度	指出了库说明部分的实际长度。
**11～12	记录长度	每条记录的总长度。
13～32	0	未用

* 库说明部分的实际长度=(第 1 字节的值) * 256+(第 9 字节的值)

** 记录长度=(第 12 字节的值) * 256+(第 11 字节的值)。

库说明部分从第 33 个字节开始，依次存放每个字段结构的说明，每 32 个字节描述一个字段，其中有用字节的具体内容如表 16.2 所列：

表 16.2 字段结构说明

字节	内容	含义
1～1	字段名	以 ASCII 表示的字段名，当字段名不足 10 个字节时，后补空格
11	0	保留未用
12	字段类型标志	用字母 C、D、L、N、M 等的 ASCII 码表示的该字段类型。
13～14	偏移地址	首记录中该字段对应内存的偏移地址
15～16	段地址	首记录中该字段对应内存的段地址
17	字段长度	表示该字段的总长度(以字节为单位，最多不超过 256 字节)
18	小数位数	表示数字型(N 型)字段的小数部分的位数，否则为 0
19～32	0	保留部分(未用)

字段结构说明部分结束后存放一个结束符 0DH，表示库说明部分结束，下面是数据库的具体数据内容，对 dBASE Ⅲ，则是用 2 个字节存放 0DH 和 00H 以表示库说明部分结束，这也是区别 dBASE 数据库和 FoxBASE 数据库的标志。

紧接库说明结束标志 0DH 之后(对 dBASE Ⅲ 为 0DH、00H 后)便是数据区，每条记录按字段顺序依次存放，每条记录的第一个字节存放删除标志，若该记录已被删除，则该字节存放 2AH(“*”的 ASCII 码)，否则存放 20H(空格的 ASCII 码)，其后是本记录各字段的 ASCII 码，各个记录都是定长的，每个记录的字段之间无分隔符，每个记录也无终止符。

数据区结束符为 1AH 标志，这也是整个数据库结束的标志。

下面以 Score.dbf 为例，具体看一下其文件结构。

用 FoxBASE 的 List 命令列出该数据库的内容如下：

```
. use score1
. list
记录号#  姓名     数学  物理  英语  化学  平均成绩
      1  张国有   83.0  76.5  93.4  80.5         .
      2  王玉香   90.3  85.2  91.5  85.5         .
      3  李卫军   73.4  65.8  80.5  82.6         .
```

为了容易看懂库结构，我们将上述库进行复制，不过各字段名用英文表示(因为若是汉字，则在机器内用内码表示)，这样在用 DEBUG 程序看库结构说明部分时，则可看到相应的 ASCII 码表示的字段名，这个复制的库名为 Score4.dbf，列出其内容如下：

```
. use score4
. list
记录号#  NAME     MATH  PHYS  ENGL  CHEM  AVER
      1  张国有   83.0  76.5  93.4  80.5    .0
      2  王玉香   90.3  85.2  91.5  85.5     .
      3  李卫军   73.4  65.8  80.5  82.6     .
```

再用 List stru 命令，看一下库结构形式：

```
. use a:score4.dbf
. list stru
Structure for database  : A: \SCORE4.DBF
Number of data records  :       3
Date of last update     : 02/25/95
Field  Field Name  Type        Width      Dec
    1  NAME        Character       8
    2  MATH        Numeric         4        1
    3  PHYS        Numeric         4        1
    4  ENGL        Numeric         4        1
    5  CHEM        Numeric         4        1
    6  AVER        Numeric         4        1
** Total **                       29
```

为了验证上面所述的库结构形式，我们退出 FoxBASE，在 DOS 下，用 DEBUG.COM 程序将 Score4.dbf 数据装入内存。DEBUG.COM 程序是 DOS 提供的一个用来显示、修改或执行文件的常用程序，当然目前有许多新的工具软件可完全和方便地代替这个程序，用户也可以用它们进行验证。

下面是用 DEBUG 来显示该库结构的操作过程：

首先用 DEBUG 将 Score4.dbf 从盘上调入内存：

```
c> debug score4.dbf
```

这时出现 debug 状态的提示符-，然后用 debug 的 R 命令看一下装入 Score4.dbf 的内存地址，即知道了段寄存器 DS 的值，和 IP 寄存器的值，即可知道装入的开始地址，装入的长度则可由 CX 寄存器的值得知，即

```
-r ↵
```

则显示的各寄存器内容如下：

```
AX=0000  BX=0000  CX=0139  DX=0000  SP=CFDE  BP=0000  SI=0000
   DI=0000
DS=136A  ES=136A  SS=136A  CS=136A  IP=0100   NV UP DI PL
   NZ NA PO NC
136A:0100  035F02            ADD     BX,[BX+02]     DS:0002=9F53
```

可以看出，装入地址为 136A:0100，装入长度为 139H 个字节，然后再用 D 命令看一下将 Score4.dbf 装入内存中的具体内容：

```
-d100l139
```

该命令中，100 表示显示内存的开始地址，L 后面的数表示显示的字节数，执行该命令后，显示结果如下：

```
136A:0100  03 5F 02 19 03 00 00 00-E1 00 1D 00 00 00 00 00  ._.......a.......
136A:0110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
136A:0120  4E 41 4D 45 00 00 00 00-00 00 00 43 01 00 00 00  NAME.......C....
136A:0130  08 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
136A:0140  4D 41 54 48 00 00 00 00-00 00 00 4E 09 00 00 00  MATH.......N....
136A:0150  04 01 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

```
136A:0160  50 48 59 53 00 00 00 00-00 00 00 4E 0D 00 00 00  PHYS.......N....
136A:0170  04 01 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
136A:0180  45 4E 47 4C 00 00 00 00-00 00 00 4E 11 00 00 00  ENGL.......N....
136A:0190  04 01 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
136A:01A0  43 48 45 4D 00 00 00 00-00 00 00 4E 15 00 00 00  CHEM.......N....
136A:01B0  04 01 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
136A:01C0  41 56 45 52 00 00 00 00-00 00 00 4E 19 00 00 00  AVER.......N....
136A:01D0  04 01 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
136A:01E0  0D 20 D5 C5 B9 FA D3 D0-20 20 38 33 2E 30 37 36  . UE9zSP  83.076
136A:01F0  2E 35 39 33 2E 34 38 30-2E 35 20 20 2E 30 20 CD  .593.480.5  .0 M
136A:0200  F5 D3 F1 CF E3 20 20 39-30 2E 33 38 35 2E 32 39  uSqOc  90.385.29
136A:0210  31 2E 35 38 35 2E 35 20-20 2E 20 20 C0 EE CE C0  1.585.5  .  @nN@
136A:0220  BE FC 20 20 37 33 2E 34-36 35 2E 38 38 30 2E 35  >¦  73.465.880.5
136A:0230  38 32 2E 36 20 20 2E 20-1A                       82.6  .  .
```

用 debug 的 D 命令显示内存内容时，每个字节的内容用 16 进制表示(其中第一列表示起始的内存地址)，右边是各字节内容的 ASCII 码显示，不可见字符则用(·)表示。

16.1.2 对 FoxBASE 数据库中数据的读取

由上面的显示可知，若要读取数据库中的记录，只要跳过库结构的描述部分，将文件指针指到所要读的记录开始处即可，不过在 C 程序中要定义一个结构，使结构中的每一个成员对应于数据库中的一个字段，由于数据库的记录内容均以 ASCII 码形式存放，因而结构中的各成员均要定义成字符型数组，其数组大小均和字段宽度一致(注意！数据库的结构描述是以二进制方式存放，而不是用 ASCII 码)。这样，当要读取记录时，则要用 fopen()函数按文本文件方式打开数据库文件。

文件指针的定位，可由读取结构说明部分的第 9-10 字节(若用序号则为 8-9 字节)内容而得知结构描述部分的长度，将文件指针跳过此长度，即定位到记录开始处，因而可将其读出而放入预先定义的结构中，从而可实现 C 程序对记录数据的处理。

记录个数可由 5～8 字节内容得知(若用序号则是 4～7 字节)，若已知记录长度(可从.dbf 库的各字段长度得知)，则可任意定位到要读的记录处，进行读取。

如由 fopen()函数打开了 score.dbf 库文件，

```
fp=fopen("score.dbf","rb");
```

要将文件指针定位到 8 字节而得知结构描述部分长度，可用如下语句：

```
fseek(fp,8,0);
fscanf(fp,"%c %c",&a,&b);
filehead=a+b*0x100      /* 文件结构描述部分长度 */
```

其中 a,b 为字符型变量。

若要得知文件记录个数，可用如下语句

```
fseek(fp,4,0);
fscanf(fp,"%c %c",&a,&b);
reccount=a+b*0x100;      /* 记录个数 */
```

上面假设记录个数的数目不超过 512 个，否则将要继续读取下面的字节，进行计算。

假设已在C程序中定义了一个结构rec，其指针为*s，它的每个成员对应于记录中的相应字段，则要顺序读出各条记录，可用如下的语句：

```
for(i=0;i<reccount;i++)
  {
    fseek(fp,filehead+i*reclen,0);
    fread(s,sizeof(rec),fp);
          ⋮
  }
```

若要随机地对记录进行读取，则可用如下语句：

```
printf("请输入读取的记录号");
scanf("%d\n",&readnum);
while(readnum<reccount)
  {
    fseek(fp,filehead+(readnum-1)*reclen,0);
    fread(s,sizeof(rec),fp)
          ⋮
    scanf("%d\n",&readnum);
  }
```

16.1.3 程序例

下面的程序将实现按输入的数据库文件名，跳过该文件结构说明部分的字节，把其后的记录内容写到用户指定的一个数据文件中去。

在程序中定义了一个结构rec，使其成员和数据库的记录字段一致，由于在库中每个记录之间有一空格，因而在结构中多定义了一个字符变量成员sign(该成员也可能取*号，因在库中一条记录被删除后，第一个字节将不是空格而是*号)。

在主程序中定义了两个字符数组file1[12]和file2[12]，用来存放用户输入的FoxBASE数据库文件名和将记录要存入的文件名，它们的名字长度(包括后缀)可控制在12个字符长度以内。输入文件名时，不要输入后缀，程序自动用strcat()函数将后缀加上了。

子程序rw用来将数据库中的记录读出并写到用户命令的文件中去，在该程序中，首先由库的第8～9字节得到库结构说明部分的长度(存入变量filehead中)，并由4,5字节得到该库中存放的记录数reccount，用该数控制循环，将数据库中的记录写到指定的文件中去。

程序中disp()函数是用来显示字符串，它是用EGA/VGA的BIOS功能调用来实现的。

函数cls()用来清屏，它是用行上滚来实现清屏的，也是采用EGA/VGA的BIOS调用来实现的。

函数goto_xy用来移动光标，也是由EGA/VGA的BIOS调用来实现的。

在上述调用中，将x看作是行，y看作是列，因而在读程序中，要注意和通常命名的差别。

```
#include<stdio.h>
```

```
#include<stdlib.h>
#include<string.h>
#include<dos.h>
#include<bios.h>
#include<ctype.h>
#include<conio.h>
void goto_xy(),cls(),disp();
void rw();
struct rec{
  char sign;
  char name[8];
  char math[4];
  char phys[4];
  char engl[4];
  char chem[4];
  char aver[4];
   }record;                    /* 结构形式和 score1.dhf 数据库记录对应 */
main()
{
char file1[12],file2[12];
char *fp1,*fp2;
fp1=file1;
fp2=file2;
cls(0,24,0,79,7);
disp(11,10,"pleas input foxhase data file name:",27);
goto_xy(11,45);
gets(fp1);
disp(12,10,"pleas input c data file name:",27);
goto_xy(12,40);
gets(fp2);                     /* 给输入的数据库名加上后缀 */
strcat(fp1,".dbf");
strcat(fp2,".dbc");
rw(fp1,fp2);
cls(0,24,0,79,27);
goto_xy(1,1);
}
void rw(char *f1,char *f2)
  {
   unsigned int filehead,reccount,reclen,recnum;
   int i,j=0;
   unsigned int count=0;
   unsigned char a,b;
   FILE *fpin,*fpout;
   if((fpin=fopen(f1,"rh"))==NULL)
```

```
    {
     disp(22,12,"Can't open file",27);
     exit(0);
    }
    reclen=sizeof(struct rec);          /* 得到记录长度 */
    fpout=fopen(f2,"w");
    fseek(fpin,8,0);
    fscanf(fpin,"%c%c",&a,&b);
    filehead=a+b*256;                   /* 得到库结构说明部分长度 */
    fseek(fpin,4,0);
    fscanf(fpin,"%c%c",&a,&b);
    reccount=a+b*256;                   /* 得到记录个数 */
  for(i=0;i<reccount;i++)
    {
     fseek(fpin,filehead+i*reclen,0);
     if(fread(&record,sizeof(struct rec),1,fpin)!=NULL);     /* 将记录读出 */
     {
       fseek(fpout,sizeof(struct rec)*count,0);
       fwrite(&record,sizeof(struct rec),1,fpout);     /* 写到另一文件中去 */
       count+=1;                                       /* 记录号加 1 */
     }
    }
    fclose(fpout);
    fclose(fpin);
    }
void disp(x,y,p,attr)                   /* 显示带有颜色属性的字符串 */
int x,y;
char *p;
int attr;
{
  union REGS r;
  register int i,j;
  for(i=y;*p;i++)
   {
    goto_xy(x,i);
    r.b.ah=9;
    r.h.bh=0;
    r.x.cx=1;
    r.h.al=*p++;
    r.h.bl=attr;
    int86(0x10,&r,&r);
   }
}
void cls(int x1,int x2,int y1,int y2,int color)     /* 清部分屏幕 */
```

```
{
 union REGS r;
 r.h.ah=6;
 r.h.al=x2-x1+1;
 r.h.ch=x1;
 r.h.cl=y1;
 r.h.dh=x2;
 r.h.dl=y2;
 r.h.hh=color;
 int86(0x10,&r,&r);
}
void goto_xy(int x,int y)                    /* 光标移至 x,y */
{
 union REGS r;
 r.h.ah=2;
 r.h.dl=y;
 r.h.dh=x;
 r.h.bh=0;
 int86(0x10,&r,&r);
}
```

16.1.4 C 程序读取数据库中 MEMO 字段

我们知道，FoxBASE(dBASE)数据库中，字符型、数字型、日期型字段在内存中所占字节数和其字段长度一样，且以 ASCII 码存放，但唯独 MEMO 字段规定占 10 个字节，这 10 个字节中的内容并不是该字段的具体内容，而是存放其具体内容的地址。MEMO 字段的具体内容是存放在 .DBT 文件中，在 .DBT 文件中，是以块的形式记载着 MEMO 字段的内容，块以 512 个字节为单位，在 .DBT 文件中，最先的两个块存的是头信息和结构描述等，从第三个块开始，才是 MEMO 字段的内容，且以 ASCII 码存放，每个 MEMO 的内容占有完整的块数，不会因为前一个 MEMO 内容未占满一个块，而将新的一个 MEMO 内容再放入该块中，一个 MEMO 字段内容结束标志为 0x1a，所以 MEMO 字段 10 个字节中，实际存放的是该字段具体内容在 DBT 文件中的位置，即起始逻辑块号。

因此，C 程序要读取 MEMO 字段内容时，可以按如下步骤进行：

1. 读取数据库结构说明部分的第 9～10 字节内容，以得到存放记录的开始位置，即将文件指针定位于此。

2. 根据库结构其它字段类型长度定义，找到记录中 MEMO 字段的开始位置，从而可读出其内容，进而得知其具体内容在 DBT 文件中存放的逻辑块号。

3. 读取 DBT 文件中相应块号的内容，直至遇到 0x1a 为止，这样一个 MEMO 字段代表的具体内容便被读出了

下面是一个示例程序，假设 drssjk.dbf 为一个数据库文件，而 drssjk.dbt 为 MEMO 字段对应的说明文件，drssjk.dat 文件是保存读出的 MEMO 字段对应的实际内容，它是一个 C 程序可用的文件，程序中定义了一个结构，其各成员对应于 .dbf 库一条记录的各条字段，

并定义了该结构类型的指针 record，程序中用 malloc()函数分配了可装下一个记录的内存空间，然后得知数据库结构说明部分的长度，从而将文件指针定位到第一条记录处；读出该记录的 MEMO 字段内容到 buff 中(用 strncpy(buff,&record[0].nr[10],10))，共 10 个字节，然后将其用 ASCII 码表示的内容转换为数字，从而得出具体内容在 .dbt 文件中的逻辑块号。

程序中用 fwrite()函数将记录和 MEMO 字段对应的明细文件中的内容写入到 dat 文件中去，并用＃号作为分隔标记，当读写完后，再最后加入 0x1a 结束标志。

```
#include<stdio.h>
main()
{
  FILE *fp;
  FILE *fp1;
  FILE *fp2;
  int bio_seq;
  char filehead,a,b,get_char,eof_flag;
  char *buff, *footer;
  struct file_title{
    char flag[1];
    char ygly[40],ygrq[6],fsdd[16],sjfw[18];
    char syzt[40],gjzl[10],syfw[18],gjbh[9];
    char me[64],nr[10],mark[1];
     } *record;                          /* 定义记录结构 */
  fp1=fopen("drssjk.dbf","rb");          /* 打开 dbf 文件只读 */
  fp2=fopen("drssjk.dbt","rb");
  fp=fopen("drssjk.dat","rb");
  record=malloc(sizeof(struct file_title));  /* 分配内存空间 */
  if(fp1=='\0')                          /* 若返回指针为 0,表示打开失败 */
   {puts("Can't open the file");
    exit(1);
   }
  eof_flag=0x1a;
  footer='#';                            /* 分隔标记 */
  buff="";
  fseek(fp1,8,0);
  fscanf(fp1,"%c%c",&a,&b);
  filehead=a+b*0x1a;                     /* 得到结构说明部分长度 */
  fseek(fp1,filehead,0);
  while(fread(&record[0],sizeof(struct file_title),1,fp1)!=NULL)
   {
    strncpy(buff,&record[0].nr[0],10);
    buff[10]='\0';
    bio_seq=atoi(buff);                  /* 得到 MEMO 的块号 */
```

```
    fwrite(&record[0].ygly[0],sizeof(struct file_title)-1,1,fp);
    fwrite(footer,1,1,fp);                    /* 写入分隔标记 */
    while(bio_seq! =0)
     {
      fseek(fp2,bio_seq * 256,0);             /* 指向 MEMO 的具体内容 */
      get_char=getc(fp2);
      while(get_char! =0x1a)
       {
        putc(get_char,fp);
        get_char=getc(fp2);
       }                                      /* 将 MEMO 字段对应的内容写到 .dat 文件中 */
      fwrite(footer,1,1,fp);
      break;
      }
    }
    fwrite(&eof_flag,1,1,fp);                 /* 写入结束标志 */
    fclose(fp1);
    fclose(fp2);
    fclose(fp);
    return(0);
}
```

16.1.5 自定义的几个对 FoxBASE 操作的函数

由于 C 读取 FoxBASE 数据库，不管库大小和复杂如何，其操作过程具有共同性，因而可以将它们的操作写成类似的标准函数形式，使用时，如同库函数，进行调用即可。

首先编一个头文件，设名为 fox.h，用于定义一些操作中常要用到的一些常数和外部变量，还有定义的一些函数说明，内容如下所列：

```
#define MAXFIELD 128                          /* 记录字段的最大长度 */
unsigned long recordnum;                      /* 记录号 */
unsigned recordlen,structlen,filednum;        /* 记录长，库结构长，字段数 */
FILE * fp;
struct {
  char name[11];                              /* 字段名 */
  char type;                                  /* 字段类型 */
  int length;                                 /* 字段长度 */
  int decim;                                  /* 小数位数 */
  }field[MAXFIELD];
union {
  long i;                                     /* 存整型字段 */
  double f;                                   /* 存实型字段 */
  char s[255];                                /* 存字符型字段 */
  }f_value[MAXFIELD];
```

```
void clear();
void get_head();
void go();
void duse();
void readdata();
void dispdata();
void closedata();
```

其中结构数组 field[MAXFIELD]用来存放记录的各字段的说明信息，这些信息包括：字段名，字段类型，字段长度，数值型字段数据的小数位数。联合数组 f_Value [MAXFIELD]用来存放记录中各字段的值，不同类型的成员将存放于同一区域内。在上述的结构和联合中，都使用了 FoxBASE 所允许的参数最大值，这样可适用于任何规模的数据库结构。

在头文件中，有七个库函数说明，它们分别是：

void clear()，用来设置屏幕为图形方式，以便用来读数据库数据后进行画图。

Void get_head()，用来读取数据库的结构信息，以便能找到数据库的数据。

Void go()，用来移动记录指针，以便读取相应的记录。

Void duse()，用来打开数据库，以便得到其指针，以对该库进行存取操作。

Void readdata()，用来读取数据库的一个记录。

Void dispdata()，用来显示数据库的某一个记录。

Void dclosedata()，该函数用来关闭打开的数据库。

这些函数的清单如下所列：

```
#include<graphics.h>
void clear()
{
  int graphdriver=DETECT;
  int graphmode;
  initgraph(&graphdriver,&graphmode,"");
}

void get_head(FILE *fp)
{
  unsigned char ch[9];
  int i;
  fseek(fp,41,0);
  fgets(ch,9,fp);
  recordnum=ch[0]+256*ch[1]+4096*ch[2]+65535*ch[3];
  recordlen=ch[6]+256*ch[7];
  structlen=ch[4]+256*ch[5];
  filednum=(structlen-32-1)/32;
  for(i=0;i<filednum;i++)
   {
    fseek(fp,(i+1)*32,0);
    fgets(field[i].name,11,fp);
```

```
    fseek(fp,(i+1)*32+11,0);
    filed[i].type=fgetc(fp);
    fseek(fp,(i+1)*32+16,0);
    ch[0]=fgetc(fp);
    filed[i].length=ch[0];
    fseek(fp,(i+1)*32+17,0);
    fgets(ch,2,fp);
    filed[i].decim=ch[0];
    }
}

void go(FILE *fp,unsigned long i)
  {
  if(fp==NULL)
    {
    printf("file not open\n";
    getch();
    return;
    }
   fseek(fp,structlen+(i-1)*recordlen,0);
  }

void duse(char *filename)
  {
  if((fp=fopen(filename,"rwb"))==NULL)
    {
    printf("file con't open\n");
    getch();
    exit(1);
    }

  void readdata(FILE *fp,unsigned long rnum)
   {
    int j;
    unsigned lim;
    char ch;
    double ftemp;
    if(rnum>recordnum)
     {
      printf("parameter error\n");
      getch();
      return;
     }
    go(fp,rnum);
```

```
ch=fgetc(fp);
if(ch==' * ')
 {
  printf("record had been deleter");
  return;
 }
printf("filednum=%d\n",filednum);
j=0;
lim=fieldnum;
while(j<lim)
 {
  fgets(f_value[j].s,filed[j].length+1,fp);
  if(field[j].type=='N')
   {
  ftemp=atof(f_value[j].s);
  if (field[j].decim>0)
   {
    f_value[j].f=ftemp;
   }
   else
    f_valye[j].i=ftemp;
    }
  j++;
   }
}

void dispdata(FILE *fp,unsigned long rnum)
{
 int j;
 unsigned lim;
 lim=filednum;
 readdata(fp,rnum);
 clear();
 j=0;
 while(j<lim)
  {
   printf("%-10s:",field[i].name);
   if(field[i].type=='N')
    {
   if(field[i].decim>0)
    printf("%f\n",f_value[j].f);
   else
    printf("%d\n".f_value[j],i);
    }
```

```
        }
        else
          printf("%s\n",f_value[j].s);
        j++;
        }
      getch();
    }

void closedata(FILE *fp)
{
   if(fclose(fp))
     printf("database close error\n");
     exit(1);
   }
```

16.2 通过 FoxBASE 索引文件读取数据

我们知道在索引文件中，是把数据库的关键字按升序排好，再把每一个关键字的值指向该值在数据库中的记录。要按关键字查询数据库中的某一记录，只要在索引文件中找到该关键字即可找到该记录，由于索引文件采用了一种类似 B^+ 树的稠密索引结构，且索引文件较原库小得多，所以用它进行索引查询，效率是很高的，若要用 C 程序按关键字对大的数据库进行索引查询，那么通过索引文件，可高效完成。为此必须要了解索引文件的结构。

FoxBASE 的每个索引文件(后缀为.idx)由索引文件头和索引文件体组成，索引文件逻辑上体现的索引树的各索引结点是以相对物理块号标识的，每个物理块号为 512 个字节，所以索引文件头占一个块，相对于索引文件的物理零块号，该文件头描述索引文件的组织信息，包括索引树的根结点位置，索引关键字表达式及索引关键字长度。我们可用 DEBUG 程序将一索引文件 teacher.idx 装入内存，看其结构，首先来看一下索引文件头的内容。下面列出了一个示例的数据库 teacher.dbf，它由 4 个记录组成，为了将其索引文件装入内存时，容易看懂关键字，我们将各字段用英文名代替，这样，就形成了一个新库 teacher1.dbf，而 tearcher1.idx 索引文件就是对该库按 address(地址)字段进行索引的文件，这两个库的内容如下所列：

teacher.dbf

记录号#	姓　名	性别	年龄	地　　址	工　资	婚否	出生年月	工作成绩	职称
1	杨金棋	男	34	友谊路 21 号	287.68	已	09/18/61	备注	工程师
2	俞家键	男	31	鞍山道 12 号	287.90	已	09/18/62	备注	讲师
3	俞媛媛	女	60	重庆道 32 号	450.00	已	05/27/35	备注	副教授
4	毛进程	女	32	友谊路 12 号	170.00	未	08/13/73	备注	助教

teacher1.dbf

记录号#	NAME	SEX	AGE	ADDRESS	SALARY	MARITAL	BIRTH	LEVEL	OCCUPATION
1	杨金棋	男	34	友谊路 21 号	287.68	已	09/18/61	备注	工程师
2	俞家键	男	31	鞍山道 12 号	287.90	已	09/18/62	备注	讲师

```
  3  俞媛媛  女    60  重庆道 32 号 450.00      已      05/27/35  备注    副教授
  4  毛进程  女    32  友谊路 12 号 170.00      未      08/13/75  备注    助教
数据库结构：                    A:\TEACHER1.DBF
数据记录数：                    4
最新更改日期                    ：03/10/95
字段    字段名          类型        宽度        小数
   1    NAME            字符         6
   2    SEX             字符         2
   3    AGE             数值         3
   4    ADDRESS         字符        10
   5    SALARY          数值         6          2
   6    MARITAL         字符         2
   7    BIRTH           日期         8
   8    LEVEL           备注        10
   9    OCCUPATION      字符         8
** 总和 **                          56
```

用 DEBUG 程序将 teacher1.idx 装入内存，可以看出共占用 400H 字节（由 CX 中的值可知）。

```
-r
AX=0000  BX=0000  CX=0400  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3E99  ES=3E99  SS=3E99  CS=3E99  IP=0100   NV UP EI PL NZ NA PO NC
3E99:0100 0002                ADD     [BP+SI],AL                SS:0000=CD
```

装入地址为 3E99:0100

显示一下文件头：

```
-d100130
3E99:0100  00 02 00 00 FF FF FF FF-00 04 00 00 0A 00 00 00  ................
3E99:0110  41 44 44 52 45 53 53 00-00 00 00 00 00 00 00 00  ADDRESS.........
3E99:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

可以看出，从 3E99:120 到 3E99:2FF 全为 00H，故不再列出。在这 512 个字节的文件头中，有用字节的含义如表 16.3 所列：

表 16.3　索引文件的文件头各字节含义

字　节	含　义
1～4	标识根结点所在块号
5～8	保留，全为 FFH
9～12	索引文件总的块数
13	索引关键字长度
17～	用 ASCII 码表示的索引关键字表达式

索引文件体从索引文件的相对物理块号为 1 的块开始，文件体的每块（也就是索引树的一个结点）又分为两个部分：前 13 个字节标识本块的属性，后面为按索引关键字非递减顺序排列的索引项，每个索引项结构如下：

索引关键字 | 指针域(4字节)　索引项所占字节数，即索引项长度，计算公式为：

索引项长度＝索引关键字长度＋指针域长度

当是叶结点，指针域用来存放该索引关键字所对应的记录号，为固定4个字节。索引树的索引结点所能存放的索引项数最多为：

M＝[(512－12/索引项长度]

当索引文件中的记录数不大于M时，索引树只有一个结点，根、叶合一，我们示例的teacher.idx就属于这种。用DEBUG装入该索引文件，显示一下其文件体，即

D300L50

文件体显示结果如下：

```
3E99:0300   03 00 04 00 FF FF FF FF-FF FF FF FF B0 B0 C9 BD ................
3E99:0310   B5 C0 31 32 BA C5 00 00-00 02 D3 D1 D2 EA C2 B7 ..12..........
3E99:0320   31 32 BA C5 00 00 00 04-D3 D1 D2 EA C2 B7 32 31 12..........12
3E99:0330   BA C5 00 00 00 01 D6 D8-C7 EC B5 C0 33 32 BA C5 ..........32..
3E99:0340   00 00 00 03 00 00 00 00-00 00 00 00 00 00 00 00 ................
```

可以看出从3E99:300到3E99.4FF为一文件体，由于teacher1.idx中的记录数比较少，小于前面提到的M值，即只有一个结点，且根、叶重合，故只有一个块。

文件体的前13个字节标识本块的属性，后面则是按索引关键字存放的索引项。

前13个字节具体的内容含义如表16.4所列。

表16.4　文件体前13个字节含义

字　节	含　义
1	块属性标记，若为00H，表示为非叶结点和根结点，01H，表示为根结点，02H表示为叶结点，03H表示根叶结点合1。
2	00H
3～4	块内索引项总数
5～8	同一层中前继结点块号或4个FF
9～12	同一层中后继结点块号或4个FF
13～	按索引关键字升序排列的索引项，当是叶结点时，则该索引项中的指针域指出的是该记录在数据文件中的记录号，否则指针域指出的是索引树下层相应结点。

对teacher1.idx文件，由于只包含少量个记录(<M)，故索引树只有一个结点，可以看出：第1个字节为03，第3个字节为索引项数，由于一个结点最多可有[(512－12)/(1＋4)]＝100个索引项，而最少可有[(512－12)/(100＋4)]＝4个索引项，其中分母中的1和100，分别表示索引关键字可能最小或最大的长度，而4则表示指针域长度(固定为4个字节)。从内存显示图中，可以看出tearcher1.idx有4个索引项。第5～12个字节为8个FF，表示为双向链表，有两个表尾结点。第13字节是按索引关键字升序排列的的索引项，由于汉字在机器内用内码表示，所以在DEBUG显示内存图中，相应内容用对应的ASCII码表示，若没对应的，则用点表示，所以看不出具体的汉字内容，但可以看出由指针域指出的该记录在数据库文件(teacher.dbf)中的对应记录号，如按升序分别为02,04,01,03号记录。

了解了索引文件的结构，我们就可以通过索引文件迅速找到要索引查询的记录号，进而

从数据库文件中读出该记录进行处理。

下面是一个示例程序，按照要查询的索引关键字“友谊路 21 号”，在 teacher1.idx 中找到相应的记录号，使用该程序时，索引关键字长度一定要按照在数据库中该字段长度来定义，且要找的关键字一定也要按此长度输入，当长度不够时，后面要补以空格。该程序只能查找字符型字段，由于数值型字段，在.idx 文件中用特殊码存放，故当用数值型输入索引关键字进行查询，本程序是不适宜的。

该程序流程图如图 16.2 所示：

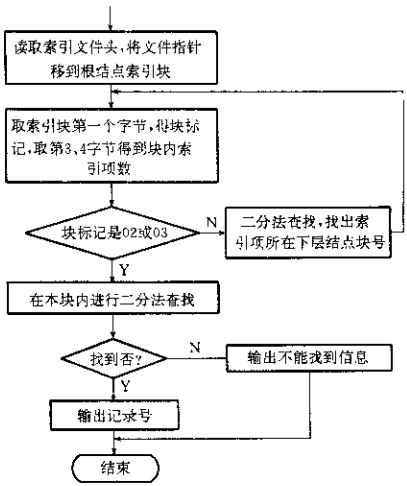


图 16.2 Search1.c 流程图

在该程序中，int comp()函数是进行字符串比较，按比较结果而返回不同的值。该函数也可用库函数 strcmp()代替。

binsearch1()函数用来当块标记不是 02 或 03 时，进行二分法查找，以找出下层结点的块号，而 binsearch2()函数则是在块标记为 02 或 03 时，在当前块中进行查找，查出记录号。

程序中关键字长度 KEYLEN 定义为 10 个字节，这是因为 address 字段为 10 个字符宽，故索引项长度 ITEMLEN＝KEYLEN＋4＝14，其中 4 表示指针域固定为 4 个字节宽。

对索引文件要查找的索引关键字是“友谊路 21 号”，运行该程序，最后执行结果是：

```
The record number is 1
#define KEYLEN 10                          /*索引关键字长度 */
#define ITEMLEN (KEYLEN+4)                 /*索引项长度*/
#include<stdio.h>
char headblock, *keyword="友谊路 21 号"      /*要查找的索引关键字*/
FILE *fp;
int binsearch1(char *key,char *base,int nelem);
```

```
int comp(char *a,char *b);
int binsearch2(char *key,char *base,int nelem);
int index(char *key);
main()
{
  unsigned long recordnum;
  if ((fp=fopen("a:teacher1.idx","rb"))==NULL)
   {
    printf("Can't open the file");
    exit(1);
   }
  fseek(fp,1l,0);
  fscanf(fp,"%c",&headblock);
  recordnum=index(keyword);
  if(recordnum>0)
    printf("The record number is :%u\n",recordnum);
   else
    printf("Can't find the record containing key %s\n",keyword);
}
int comp(char *a,char *b)
  {
   int i;
   for (i=0;i<KEYLEN;i++)
    {
     if(*(a+i)>*(b-i))
       return(1);
     else if(*(a+i)<*(b+i))
       return(-1);
     }
    return(0);
  }

int binsearch1(char *key,char *base,int nelem)        /*nelem 是索引项数*/
  {
   int bnelem,unfound,before,after;
   unfound=1;
   if(comp(key,base)<=0)
   return(*(base+KEYLEN+2));
   else
    if(comp(key,base+(nelem-1)*ITEMLEN)>0)
    return(0);
     else
       {
        while(unfound){
```

```
        bnelem=nelem/2;
        before=comp(key,base+(bnelem-1) * ITEMLEN);
        after=comp(key,base+bnelem * ITEMLEN);
        if(before<=0)
        nelem=bnelem;
       else
        if((before>0)&&(after<=0))
          {
           unfound=0;
           base+=bnelem * ITEMLEN+KEYLEN+2;
           return( * base+( * (base+1)) * 0x100);
          }
        else
          if(after>0)
           {
            base=base+bnelem * ITEMLEN;
            nelem-=bnelem;
           }
       }
      }
    return(0);
   }

  int binsearch2(char  * key,char  * base,int nelem)
    {
    int bnelem,unfound;
    unfound=1;
    while((nelem>0)&&unfound){
     bnelem=(nelem+1)/2;
     switch(comp(key,base+(bnelem-1) * ITEMLEN))
       {
        case 1;
          {
           base+=hnelem * ITEMLEN;
           nelem/=2;break;
          }
        case -1;
        {
          nelem=bnelem-1;break;
        }
        case 0;
        {
          unfound=0;
          base+=(bnelem-1) * ITEMLEN+KEYLEN;
```

```
            return((*base*0x1000000)*(*(base+1)*0x10000)
              *(*(base+2)*0x100)+(*(base+3)));
           }
          }
        }
       return(0);
      }

int index(char *key)
  {
   cbar buffer[512];
   long fileoffset;
   int i;
   fileoffset=headblock*0x100;
   fseek(fp,fileoffset,0);
   fread(buffer,512,1,fp);
   while((buffer[0]!=0x2)&&(buffer[0]!=0x3))
   {
    fileoffset=binsearch1(key,&buffer[12],buffer[2])*0x100;
    fseek(fp,fileoffset,0);
    fscanf(fp,"%512c",buffer);
   }
  return(binsearch2(key,&buffer[12],buffer[2]));
 }
```

16.3 从数据库的.MEM 文件中读取数据

FoxBASE 中的内存变量是在内存中定义的临时工作单元，它存储变量的类型有字符(C)、数值(N)、逻辑(L)和日期(D)，当将它们存入磁盘后，就形成了.MEM 文件，即内存变量文件，在内存变量文件中，只有字符型变量的值是以 ASCII 码形式存放。其它类型则用特殊的编码，我们也可以用 C 程序将字符型变量的.MEM 文件各变量的值读出，然后利用 C 程序的数值处理或丰富的作图功能，再对这些变量进行处理，为了能读取内存变量文件中的字符变量，我们必须了解其文件结构，为此，如同上面采用的过程一样，我们将一示例的.MEM文件，cbar.mem 用 DEBUG 装入内存，cbar.mem 文件列出来，如下所示：

```
-list memo
A               Pub     C   "15"
B               Pub     C   "35"
C               Pub     C   "46"
D               Pub     C   "74"
      4 variables defined,          36 bytes used
    252 variables available,      5964 bytes available
```

当用 DEBUG 装入内存后，看一下寄存器情况：

```
-r
AX=0000  BX=0000  CX=008D  DX=0000  SP=CFDE  BP=0000  SI=0000  DI=0000
DS=3DDC  ES=3DDC  SS=3DDC  CS=3DDC  IP=0100   NV UP DI PL NZ NA PO NC
3DDC:0100 41              INC     CX
```

可以看出，装入地址为 3DDC:0100，长度为 8DH，将其内容显示出来：

```
-d100l8d
3DDC:0100   41 00 00 00 00 00 00 00-00 00 00 43 00 00 00 00   A..........C....
3DDC:0110   03 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
3DDC:0120   31 35 00 42 00 00 00 00-00 00 00 00 00 00 43 00   15.B..........C.
3DDC:0130   00 00 00 03 00 00 00 00-00 00 00 00 00 00 00 00   ................
3DDC:0140   00 00 00 33 35 00 43 00-00 00 00 00 00 00 00 00   ...35.C.........
3DDC:0150   00 43 00 00 00 00 03 00-00 00 00 00 00 00 00 00   .C..............
3DDC:0160   00 00 00 00 00 00 34 36-00 44 00 00 00 00 00 00   ......46.D......
3DDC:0170   00 00 00 00 43 00 00 00-00 03 00 00 00 00 00 00   ....C...........
3DDC:0180   00 00 00 00 00 00 00 00-00 37 34 00 1A            .........74..
```

从而可知内存变量文件的一般结构如表 16.5 所示：

表 16.5 .MEM 文件中一个内存变量存放结构

字　节	含　义
1～10	变量名称
11	保留为 0
12	变量类型(用 C、N、D、L 的 ASCII 码表示)
13～14	偏移地址
15～16	段地址
17	字符串的长度加 1
18～32	保留为 0
33～	字符串的 ASCII 码，顺序存放，最后补 0

以后的其它变量均按此格式放置，即前面的变量字符串值补以一字节 0 后，紧跟着又是重复前面的排列，而给出另一内存变量名、类型、地址、长度，和其代表的字符串值。由于有此规律，因而我们可以方便地读出该文件中各内存变量的值(字符串)，进行处理，对于非字符型变量，我们可以在 FoxBASE 程序中将其转换为字符型变量存储后，也可将 .MEM 文件变成这种形式。

下面是一个示例程序，该程序首先将屏幕置成高分辨图形方式，然后以只读方式打开内存变量文件 char.mem。当该文件被打开后，将文件指针移到第 17 个字节处(即序号为 16 字节处)，然后读出两个字节的内容，这就是第一个变量的字符串长度值，用这个长度值，就可控制读出的字节数，因而当 l[0]=charlen[0]，再将指针后移 15 个字节后，便用 l[0]值控制读出的字节数，即可正确得到该变量的字符串值(即字符串的 ASCII 码表示，它存于 char[0][]中)，然后又将指针后移 16 个字节，移到第二个变量的字符串长度值处，将其读出后，再将指针后移 15 个字节，用字符串长度数控制读出的字节数，这样就将第二个变量的字符串值读到 char[1][]中了，如此循环，直到指针指到文件尾为止，可以将所有的内存变量值

读出。

程序中定义字符数组 ch[4][3]来接收读出的 4 个变量的字符串值，数组 charlen[2]接收字符串的长度值，数组 y[4]用来存储 4 个变量的字符串转换为数字后的值，然后根据 y[i]的值，画出不同边框颜色和填充模式的四个立体的长方体，这就是可以形象的看出四个变量值的直观大小的直方图，这仅是一个示意程序，当然也可定义一个较大 ch[][30]数组，来接收未知数目的变量值，当程序中 while 循环结束时，由最后的 i−1 值，可知变量个数，再用 for 循环，将其字符串值用 atoi()函数转换成数值型赋给 y[i]，然后根据 i 值及 y[i]值，画出不同的直方块，以形成 i−1 个直方图。

```
#include<stdio.h>
#include<graphics.h>
#include<stdio.h>
main()
{
int i=0,l[30],y[4];
char ch[4][30],charlen[2];
FILE *fp;
int graphdriver=VGA;
int graphmode=VGAHI;
initgraph(&graphdriver,&graphmode,"");
cleardevice();
if((fp=fopen("char.mem","rb"))==NULL)
  {
   printf("Can't open the file");
   exit(1);
  }
while(! feof(fp))
  {
   fseek(fp,16,1);
   fgets(charlen,2,fp);
   l[i]=charlen[0];
   fseek(fp,15,1);
   fgets(ch[i],l[i]+1,fp);
   i++;
  }
setbkcolor(BLUE);              /* 设置背景色为蓝色 */
setcolor(LIGHTRED);            /* 设置前景色 */
setfillstyle(8,10);
y[0]=200-atoi(ch[0]);          /* 转换第一个变量值 */
bar3d(160,y[0],205,300,20,1);          /* 画出立体直方图 */
setcolor(YELLOW);
setfillstyle(10,12);           /* 填充模式 */
y[1]=200-atoi(ch[1]);
```

```
bar3d(205,y[1],245,300,20,1);
setcolor(GREEN);
setfillstyle(3,13);
y[2]=200-atoi(ch[2]);
bar3d(245,y[2],285,300,20,1);
setcolor(WHITE);
setfillstyle(11,10);
y[3]=200-atoi(ch[3]);
bar3d(285,y[3],325,300,20,1);
getch();
closegraph();
}
```

16.4 C 程序间接读取数据库的.DBF 文件

我们知道 FoxBASE(或 dBASE)数据库中的数据是以 ASCII 码的形式存放,但在实际的数据前面存放着一些库结构信息,若要直接读取,正如前面所述,则必须跳过这些信息,将读取指针指向数据项,其读取方法前面已作了较详细的介绍。还有一种间接读取数据库的方法,就是将数据库变成文本文件,而 C 语言中,正如在 6.1 中介绍的那样,将数据的输入和输出,看作流的流入和流出,而流有两种,即文本流(textstream)和二进制流(binary stream),文本流是指流动的数据以字符形式出现,即流动着的是字符流,一行流完后必须有结束符(回车换行),Turho C 提供了许多对流式文件输入输出的操作函数,因而若将数据库文件变成 C 要求的文本流文件,则可用 C 的流式文件输入输出函数对其进行处理了,而 FoxBASE(dBASE)恰好提供了将.DBF 文件转换成文本流文件的方法,下面进行介绍:

16.4.1 用 COPY TYPE 命令将.DBF 转换成文本文件

我们知道 FoxBASE 中的 COPY TYPE 命令的一般格式为:

COPY TO <文本文件名>[FIELDS<字段名表>][FOR<条件>][WHILE<条件>]TYPE<文件类型>

其中文件类型可以为:

SDF——系统数据格式(System Data Format)文件,也称为标准格式文件。此文件由 ASCII 码组成,数据项之间没有分隔符,每行记录等长,且均以回车接行结尾,文件后缀缺省为 TXT。

DELIMITED[WITH<定界符>/BLANK]——通用格式文件,也称紧凑格式文件,它也由 ASCII 码组成,数据项之间可以由用户选择分隔符,若选择 BLANK,则数据项之间则以空格分隔,若选择了<定界符>,则数据项之间用逗号分隔,字符型数据项则用指定的定界符括起来,若没有选择 WITH,则数据项之间用逗号隔开,字符型数据项用双引号括起来。

例如对 score1.dbf 文件用如下 copy 命令进行转接:

```
.copy to score1 deli with blank
```

则生成紧凑格式文本文件 score1.tex，且数据项间用空格分隔，如在 FoxBASE 下列出该文件也可看出：

```
. run type score1.txt
张国有  83.0 76.5 93.4 80.5 .0
王玉香  90.3 85.2 91.5 85.5 .
李卫军  73.4 65.8 80.5 82.6 .
```

C 语言中从文本流中进行格式化输入的函数 fscanf，可用于从转换后的紧凑格式文本文件中读取字段，它读取字段时，可以用空格作为读字段的结束符，而在读取字符型数据时，不受字符型变量定义长度的限制，直到读到空格字符时为止，所以它很适合读取由 FoxBASE(dBASE)的.DBF 文件转换成紧凑格式的.tex 文件。

下面是以 score1.dbf 数据库文件为例，C 语言读取用 dBASE Ⅱ 的 copy 命令转换的以空格为分隔符的紧凑格式文本文件 score1.txt 的程序，该程序将读取 score1.txt 文件中的各字段，并计算每个学生的平均成绩。

```
#include<stdio.h>
main()
{
static float a[3][4];
float c,a1,a2,a3,a4;
char xm[3][8];
int i;
FILE *fp;
if ((fp=fopen("a:score1.txt","r"))==NULL)
  {printf("Can't open file");
   exit(0);
  }
for(i=0;i<3;i++)
  {
  fscanf(fp,"%s %f %f %f %f",&xm[i],&a1,&a2,&a3,&a4);
  a[i][0]=a1;a[i][1]=a2;
  a[i][2]=a3;a[i][3]=a4;
  }
printf("姓名    数学  物理  英语  化学  平均成绩      \n");
for(i=0;i<3;i++)
  {
  c=(a[i][0]+a[i][1]+a[i][2]+a[i][3]+a[i][4])/4;
  printf("%8s%4.1f%4.1f%4.1f%4.1f%4.1f\n",&xm[i],&a[i][0],&a[i][1],
   &a[i][2],&a[i][3],&a[i][4],&c);
  }
  fclose(fp);
}
```

程序中定义了一个静态实型数组 a[3][4]，用来准备存放从 score1.txt 文件中读取的

三个学生的4门课成绩，而字符型数组xm[3][8]，则用来存放每个学生的名字，每个名字定为8个字符长。定义的实型变量a1,a2,a3,a4用来接收读入的4个成绩，C则用来得到平均成绩。

程序中用fopen函数以只读方式打开score1.txt文件，并得到文件指针赋给fp，然后用fscanf函数分别将姓名和各字段的内容输入到xm[i]和a1,a2,a3,a4中去，共循环三次，然后再循环三次计算出每个学生的平均成绩，并输出学生成绩表，最后关闭fp指向的文件。

对于标准格式文件(即SDF)，每条记录是等长的，由于一般在定义数据库结构时，字段宽度往往是大于实际数据的宽度，因而在转换成SDF的.txt文件时，实际上各字段间均有空格，因而上述读取的方法，也适用于SDF格式的.txt文件。

16.4.2 C程序间接传送数据给.DBF文件

当由C程序将数据传送给FoxBASE(dBASE)的.DBF文件中去时，必须将这些数据写成和前面所述的SDF或紧凑格式文件相同结构的C语言文本文件(.txt)，然后再用FoxBASE(dBASE)的APPEND命令，将这些数据添加到.dbf文件中去。

我们知道FoxBASE(dBASE)的APPEND命令格式为：

APPEND FROM<文本文件名>[FOR<条件>]TYPE<文件类型>

其中文件类型可以是SDF标准格式文件或紧凑格式文件，与COPY命令中的文件类型相同。

该命令将数据从左开始，向指定的字段追加数据，当选择BLANK定界时，则遇到一个空格，便完成一个字段的追加，当遇到一个回车换行时，便完成一个记录的追加，如此循环，直到遇到文件结束符为止。

下面的程序是将三个学生的考试成绩，按紧凑格式写到一个C程序的score2.txt文件中去，然后将系统转换到FoxBASE(dBASE)状态下，用APPEND命令将score2.txt中的记录追加到score1.dbf数据库文件中去。

C程序中如同上节中的例子一样，将追加的三个记录中的姓名用xm[3][8]二维字符数组存放，每个人姓名为8个字符长，4门课成绩用a[3][4]存放。

用fopen()函数以只写方式打开score2.txt文件，然后用fpinrf()函数用指定的格式，写入到a数组和xm数组中去，其中写到xm中的字符串长度定为8，以和score.dbf库结构定义姓名项长度相同，各课成绩均以5.1f形式写入，该宽度比score1.dbf库中各字段宽度多了一位，为的是使各字段间有空格，各字段的排列顺序也和库结构的一致，这样才能使用APPEND命令将score2.txt文件中的各记录追加到score1.dbf数据库中去。

当然也可定义一个结构，使其成员对应于一条记录的各字段，如可用如下的定义：

```
struct score
      {char  *name;
       float a[4];
      }stud[3]={{"赵阿力",85.0,72.5,93.4,84.5,}
                {"张  香",92.3,83.2,78.5,75.5,}
                {"陆华荣",83.4,85.8,84.5,86.6}};
```

这样用如下的格式输出语句：

```
fprintf(fp,%85%5.1f%5.1f%5.1f%5.1f%5.1f\n",stud[i].name,stud[i].a[0],
stud[i].a[1],stud[i].a[2],stud[i].a[3]);
```

可以将记录写到 score1.txt 文件中去，程序内容和上面的基本相同，因而将不列出其程序清单。

三个学生的考试成绩的程序清单：

```
#include<stdio.h>
main()
{
static char xm[3][8]={"赵阿力 ","  张  香  ","  陆华荣  "};
static float a[3][4]={{85.0,72.5,93.4,84.5},
   {92.3,83.2,78.5,75.5},{83.4,85.8,84.5,86.6}};
int i;
FILE *fp;
if((fp=fopen("a:score2.txt","w"))=NULL)
  {printf("Can't open file");
   exit(0);
  }
for (i=0;i<3;i++)
 {
  fprintf(fp,"%8s%5.1f%5.1f%5.1f%5.1f\n",xm[i],a[i][0],a[i][1],
   a[i][2],a[i][3]);
 }
 fclose(fp);
}
```

在 FoxBASE 状态下，进行追加：

```
.use a:score1
.append from a:score2 deli with blank
```

这样就实现了将 C 程序写成的 score2.txt 文本文件内容追加到 FoxBASE 的 score1.dbf 数据库中了。

16.5 关于 C 和 FoxBASE 的交替使用问题

当用 C 程序对 FoxBASE 数据库的记录进行写入、修改、删除等操作时，显得很不方便，这是因为不仅对数据库的记录进行操作，还要直接修改该库文件的结构说明部分及库结束部分的标志。所以若在应用程序中有对库管理的部分，则应回到 FoxBASE 环境下去执行 FoxBASE 程序，当需对库中的数据进行处理、画图时，再回到 Turbo C 环境下，执行 C 程序。这可用 Turbo C 提供的 system()函数来实现，即在 C 程序需要调用 FoxBASE 处写上：

```
system(foxbase filename);
```

即可。其中 filename 是 FoxBASE 的命令文件，但用这种方法将带来致命的问题，因执行该

函数时，Turbo C 环境仍存在，即它没有被覆盖，但由于 FoxBASE 本身又要占较大内存，因而当 C 程序较大时，调用 FoxBASE 便没有内存空间了，故 system()将无法执行。虽然 dBASE 占用内存比 FoxBASE 少，但这个问题仍得不到根本解决。若在汉字操作系统下使用 FoxBASE，汉字操作系统又占用较大系统空间，使得内存不够的问题更显得严重，实际上，用上述方法将行不通，虽然有些人使用了扩展内存，但仍然解决不了根本问题。

一种可以采用的方法是用 DOS 的批处理命令，建立一个自动文件，按照需要，建立一种执行该需要的环境，在执行完后，释放该环境，又为下一种操作创造条件。这种环境就是某种语言系统。批处理文件，实际上起了总控作用。

例如下面是一个示例的批处理文件，用户可以自己编一个接收键盘输入的小程序，设名为 INKEY.COM，它由两个 DOS 中断调用组成，即接收键盘输入的 int 21H 和程序中止的 int 20H 调用，它主要用在批处理文件中接收用户对菜单选择项的选择，该小程序可用 DEBUG 程序来建立，即

```
-a100
136A:0100 mov ah,1
136A:0102 int 21
136A:0104 int 20
136A:0106 ^C

-rcx
CX 0000
:07
-ninkey.com          ;起名字
-w
Writing 0007 bytes   ;写入磁盘
-q
```

在批处理文件中，利用批处理命令 IF 中的这种格式：

IF ERRORLEVEL number

即由 COMMAND.COM 执行了前一程序之后，传回的代码大于或等于 number 时，条件才成立，根据条件再由 GOTO 命令进行执行的转向，因而可用批命令编制一个菜单，如用显示命令 ECHO：

```
ECHO
ECHO      1 ----- 数据管理
ECHO      2 ----- 图形显示
ECHO      3 ----- 退出
ECHO
ECHO    请按 1,2,3 进行选择：
```

然后用 INKEY 程序接收键盘输入的选择号，再用

IF ERROLEVEL number GOTO 命令人口

根据键盘选择，转入相应环境去执行。

下面的示例程序，将根据用户选择，转入不同环境去执行，如选择 1，则进入 FOX 子目录，并调用 FoxBASE 环境，执行 FoxBASE 命令程序 SJCL，然后返回菜单处，若选择 2，则回到根目录下，执行 C 语言编的执行程序 datamem. exe，此时 FoxBASE 环境已脱离，若选择 3，则清屏返回。

```
echo off
:start
cls
echo  ------------------------
echo   1.....fox
echo   2.....turbo c
echo   3.....exit
echo  ------------------------
:loop
inkey
if errorlevel 51 goto p3
if errorlevel 50 goto p2
if errorlevel 49 goto p1
:p1
cd fox
foxbase sjcl
goto start
:p2
cd..
datamem
goto start
:p3
cls
echo  -------------------
echo   Good hye
echo  -------------------
echo on
```

除了上述的方法外，还可用 C 语言编一个菜单程序，当用户选择数据管理时，用 switch 语句控制转向，此时将用写文件的形式，把调用 FoxBASE 环境及执行其命令文件的批命令写到一个批文件中去，然后执行它，当执行完 FoxBASE 的命令文件后（该文件最后一条命令必须是 quit），便返回 C 环境，然后再用运行菜单程序，进行新的选择。

如假设已编制了一个同用户界面能起到很好交互的菜单程序 int menu()，则可编制一个主程序完成调用菜单和写批处理文件并执行的功能，现列出如同上面批处理文件 C—fox. bat 同样功能的主程序：

```
.
.
.
```

```
while(1)
 {
  c=menu();                          /* 调用菜单程序 */
  fp=fopen("c_fox.bat","w");         /* 建立一个批处理文件 */
  switch(c)
   {
    case 1:
     fputs("echo Pleas waite! \n",fp);   /* 写文件 */
     fputs("cd fox\n",fp);               /* 进入 fox 子目录 */
     fputs("foxbase sjcl\n",fp);         /* 执行 FoxBASE 命令文件 */
     fputs("cd..\n",fp);                 /* 返出子目录 */
     fputs("menu",fp);                   /* 运行菜单程序 */
     break;
    case 2:                              /* 当在菜单程序中选 2 时 */
     fputs("datamem\n",fp);              /* 执行 C 的程序 */
     fputs("menu",fp);
     break;
    case 3:
     exit(0);                            /* 退出系统 */
    default:
     break;
    }
  fclose(fp);
  .
  .
  .
```

参 考 文 献

[1] 林学焦,刘力,晓夏. Tyubo c 2.0 用户手册. 北京: 学苑出版社,1993
[2] 王士元,吴芝芳. IBM PC/XT(长城 0520)接口技术及其应用. 天津: 南开大学出版社,1990
[3] 王士元. IBM PC/XT、286、386 微机汇编语言与外设编程. 天津: 南开大学出版社,1993
[4] 尹彦芝. C 语言高级实用教程. 北京: 清华大学出版社,1992
[5] 郭兴社,戴建鹏. C 语言大全. 北京: 电子工业出版社,1990
[6] 王治宇. dBASE Ⅲ程序设计与实用技术. 北京: 电子工业出版社,1991