目　录

第一章　MS–DOS环境下高水平程序设计
必备知识……………………………………………… (1)
1.1 MS–DOS 的装入过程 ……………………………… (1)
1.2 两类基本的 MS–DOS 程序 ………………………… (5)
1.2.1 使用.COM 格式 ……………………………………… (6)
1.2.2 EXE 程序结构 ……………………………………… (10)
1.3 内存管理基础 ……………………………………… (14)
1.3.1 使用内存管理功能调用 …………………………… (16)
1.3.2 内存控制块 ………………………………………… (19)
1.3.3 内存图 ……………………………………………… (20)
1.4 MS–DOS EXEC 功能 ……………………………… (21)
1.4.1 获得可用内存 ……………………………………… (23)
1.4.2 请求 EXEC 功能 …………………………………… (23)
1.4.3 实例程序 SHELL.C 及 SHELL.ASM ……………… (30)
1.5 MS–DOS 文件与记录操作 ………………………… (34)
1.5.1 使用 FCB 功能 ……………………………………… (35)
1.5.2 使用句柄文件和记录功能 ………………………… (46)
1.5.3 MS–DOS 出错代码 ………………………………… (52)
第二章　中断与DOS功能……………………………… (56)
2.1 中断机构……………………………………………… (57)
2.2 从汇编程序中访问软中断程序 …………………… (60)
2.3 从C语言中访问软中断程序 ……………………… (61)
2.4 选择中断功能………………………………………… (64)
2.5 MS–DOS 功能 ……………………………………… (65)

2.5.1 MS-DOS 功能的一般用法 …………………………… (66)
2.5.2 DOS 服务功能分类 ………………………………… (69)
第三章 BIOS 功能 ……………………………………… (91)
3.1 BIOS 功能的一般用法 ……………………………… (92)
3.2 BIOS 功能分类 ……………………………………… (93)
3.2.1 视频服务功能:中断 10h ……………………………… (93)
3.2.2 磁盘服务功能:中断 13h ……………………………… (101)
3.2.3 串行端口服务功能:中断 14h ………………………… (103)
3.2.4 盒式磁带服务功能:中断 15h ………………………… (105)
3.2.5 AT 机上的扩展服务功能:中断 15h …………………… (105)
3.2.6 键盘服务功能:中断 16h ……………………………… (105)
3.2.7 打印服务功能：中断 17h ……………………………… (107)
3.2.8 时间和日期服务功能:中断 1Ah ……………………… (110)
3.2.9 其他服务功能 ………………………………………… (110)
第四章 MS-DOS 机器的其他资源 ………………………… (113)
4.1 程序段前缀(PSP) …………………………………… (113)
4.1.1 从汇编语言中访问 PSP ……………………………… (115)
4.1.2 从 C 语言中访问 PSP ………………………………… (116)
4.1.3 重要的 PSP 域 ……………………………………… (117)
4.2 低内存地址的数据区 ………………………………… (126)
4.3 硬件产生的中断 ……………………………………… (129)
4.3.1 外部硬件中断 ………………………………………… (130)
4.3.2 硬件中断服务程序 …………………………………… (131)
4.4 其他 ………………………………………………… (138)
4.4.1 数据中断向量 ………………………………………… (138)
4.4.2 端口 …………………………………………………… (140)
4.4.3 可安装设备驱动程序 ………………………………… (142)

4.4.4 Ctrl-C 处理程序 …… (142)
4.4.5 Ctrl-Break 处理程序 …… (145)
4.4.6 致命错误处理程序 …… (152)
第五章 兼容性的理论与测试 …… (159)
5.1 一般的兼容性准则 …… (160)
5.2 确认计算机环境 …… (162)
5.2.1 资源表 …… (162)
5.2.2 动态测试 …… (165)
5.2.3 用户安装程序的使用 …… (176)
5.3 使用可提供的资源 …… (177)
5.3.1 使用特定资源 …… (177)
5.3.2 使用与机器类型有关的信息 …… (177)
5.4 MS-DOS 各版本之间的差别及其兼容性问题 …… (179)
5.4.1 版本兼容性的一般概念 …… (180)
5.4.2 高级语言的考虑与 MS-DOS 中断 …… (183)
5.4.3 功能调用 …… (184)
5.4.4 错误代码 …… (190)
5.4.5 磁盘格式 …… (195)
5.4.6 文件操作 …… (196)
5.4.7 MS-DOS 及 IBM PC 机系列 …… (200)
5.5 与其他操作系统的兼容性 …… (203)
5.5.1 CP/M-80 …… (204)
5.5.2 CP/M-86 及 Concurrent CP/M-86 …… (206)
5.5.3 Concurrent CP-DOS 和 Concurrent DOS-286 …… (206)
5.5.4 Xenix 和 UNIX …… (207)
5.6 "规距"的 MS-DOS 应用程序 …… (207)
5.6.1 基本准则 …… (208)

5.6.2 与硬件有关的 IBM–PC 应用程序 …………………… (209)
第六章 快速字符显示的程序实现 …………………… (212)
6.1 通过 DOS / ANSI.SYS 的视频显示 ………… (213)
6.2 通过 BIOS 和属性代码的视频显示 ………… (215)
6.3 通过直接对视频内存区写的视频显示 ………… (219)
6.3.1 字符串函数 …………………… (221)
6.3.2 窗口函数 …………………… (225)
6.4 基准测试(Benchmark) …………………… (232)
6.5 屏幕生成程序例 …………………… (236)
6.6 在 C 程序中使用视频显示子程序 ………… (258)
第七章 内存驻留程序设计 …………………… (266)
7.1 编写 TSR 时需注意的问题 …………………… (267)
7.1.1 与其他 TSR 共存 …………………… (267)
7.1.2 与 MS–DOS 共存 …………………… (273)
7.1.3 与前台程序共存 …………………… (278)
7.1.4 与 BIOS 磁盘活动共存 …………………… (282)
7.1.5 与中断处理程序共存 …………………… (283)
7.1.6 可重新进入的问题 …………………… (283)
7.1.7 Microsoft 标准 …………………… (285)
7.2 实现 C 语言程序的内存驻留 …………………… (288)
7.2.1 在 C 程序中使用 tsr 函数 …………………… (295)
7.2.2 汇编语言子程序的实现 …………………… (298)
7.2.3 待改进的若干功能 …………………… (304)
第八章 扩充内存及其 C 语言接口 …………………… (307)
8.1 扩充内存规范(EMS)概述 …………………… (308)
8.2 EMS 的 C 语言程序接口 …………………… (310)
8.2.1 错误码说明 …………………… (317)

8.2.2 接口功能函数……………………………………………… (318)
8.2.3 可实现的功能增加……………………………………………… (323)
8.3 从C语言中使用扩充内存…………………………… (323)
8.3.1 临时应用程序 …………………………………………… (323)
8.3.2 内存驻留应用程序 …………………………………………… (328)
8.4 Lotus / Intel / Microsoft 扩充内存规范
参考手册…………………………………………………… (329)
第九章 Intel 8087 / 80287 数学协处理器编程 ………… (345)
9.1 程序员看 8087………………………………………… (346)
9.1.1 8087 中的数据寄存器 ………………………………… (346)
9.1.2 8087 中的浮点实数表示 ……………………………… (347)
9.1.3 8087 使用的其他数据格式 …………………………… (349)
9.1.4 数据类型小结 ………………………………………… (352)
9.1.5 8087 指令集 …………………………………………… (353)
9.1.6 FWAIT 前缀……………………………………………… (354)
9.1.7 8087 的寻址方式 ……………………………………… (359)
9.1.8 FINIT 和 FFREE 指令 ………………………………… (361)
9.1.9 控制 8087 ……………………………………………… (361)
9.2 对 8087 使用 MS-DOS 工具 ……………………… (366)
9.2.1 对8087使用MASM …………………………………… (366)
9.2.2 MASM 的 8087 开关——/r和/e ……………………… (368)
9.2.3 MASM中的8087数据类型 …………………………… (368)
9.2.4 对 8087 使用 DEBUG ………………………………… (370)
9.3 用 MASM 对 8087 编程的例子………………………… (372)
9.3.1 FWAIT 和 FINIT 指令 ………………………………… (372)
9.3.2 DUMP87 子程序 ……………………………………… (372)
9.3.3 使用 8087 实现二一十进制变换 ……………………… (383)

第一章　MS-DOS环境下高水平程序设计必备知识

1.1 MS-DOS的装入过程

系统加电或重置，程序在地址0FFFF0H处开始执行。这是8086系列微处理器的特点，与MS-DOS无关。用这类处理器进行系统设计时应将地址0FFFF0H置于ROM区域中，并且其内容是一条跳转机器指令，使控制转给系统测试程序及ROM自举（bootstrap）程序。

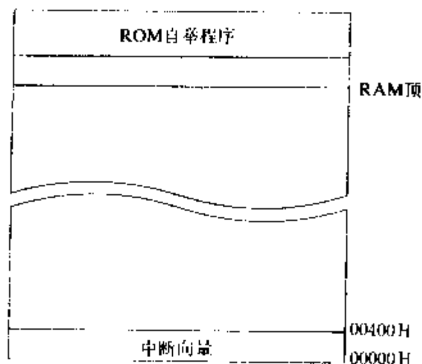


图1-1　典型的用8086／8088的微计算机系统加电后，在0FFFF0H处开始执行，该处为一条jump指令将程序控制转给ROM自举程序。

ROM自举程序从磁盘的第一个扇区（boot扇区）中读入磁盘自举程序至内存中某一任意地址，然后将控制移交给它（此boot扇区中还包含一个与磁盘格式有关的表）。

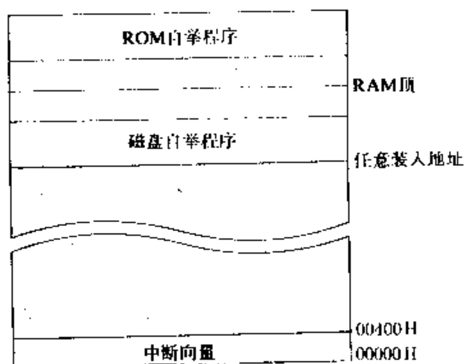


图 1-2　ROM自举程序将磁盘自举程序从系统盘第一个扇区中装入内存，然后将控制移交给它。

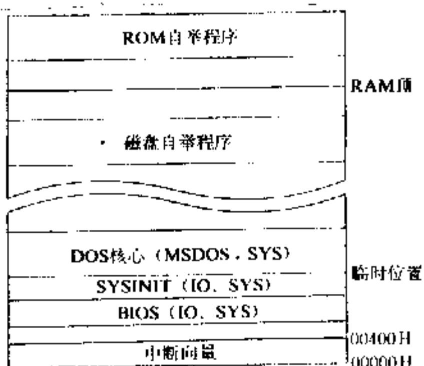


图 1-3　磁盘自举程序将文件IO.SYS读入内存 此文件包含MS-DOS，BIOS（驻留设备驱动程序）及 SYSINIT 模块，磁盘自举程序或 BIOS 再将 DOS 核心从 MSDOS.SYS 中读入内存.

磁盘自举程序查看该磁盘是否包含 MS-DOS 的拷贝，此时将由读入根目录的第一个扇区来确定前两个文件是否为 IO.SYS 及 MSDOS.SYS（顺序不能错），如果这两个文件

不存在，它便提示操作人员更换磁盘并可打入任意键重新开始。如果查到了这两个系统文件，磁盘自举程序即将它们读入内存，控制将转到 IO.SYS 的初始进入点（有些 MS-DOS 实现磁盘自举程序时只将 IO.SYS 读入内存，再由 IO.SYS 负责去装入 MSDOS.SYS 文件）。

从磁盘装入的 IO.SYS 文件，实际上由两个单独的模块组成。第一个模块是 BIOS，其内容是互相链接的一组用于控制台、辅助端口、打印机及数据分块设备的驻留设备驱动程序，再加上一些在系统启动时间运行的与硬件有关的初始化程序。第二个模块称为 SYSINIT，由 Microsoft 公司提供，并由计算机制造商随 BIOS 链接到 IO.SYS 文件中。

SYSINIT由制造商的BIOS初始化代码调用。在确定了系统中存在的相连（contiguous）内存的量之后，它再将自己重新安排到高内存区，然后将 DOS 核心 MSDOS.SYS 从原始装入位置移至内存最后位置，覆盖掉从原始的 SYSINIT 程序及包含在 IO.SYS 文件中的某些其他可扩展的初始化代码。

以下过程就是 SYSINIT 调用 MSDOS.SYS 中的初始化代码。DOS 核心初始化其内部的各种表和工作区，建立中断向量 20H 至 2FH，查找出链接后的驻留设备驱动程序清单，并为每个设备驱动程序调用初始化功能。这些驱动程序功能负责确定设备的状态并实现必需的硬件的初始化，同时为驱动程序所服务的外部硬件中断设置向量。

作为初始化序列的一部分，DOS 核心检查，由驻留块设备驱动程序返回的磁盘参数块确定系统所用的最大扇区尺寸，建立某些驱动器参数块，分配一个磁盘扇区缓冲区，然后显示出 MS-DOS 版权信息，将控制返回给 SYSINIT。

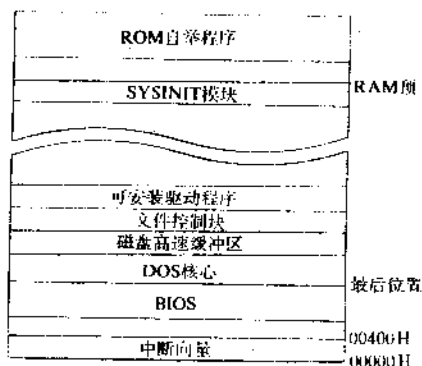


图 1-4　SYSINIT 将自身移至高内存区并重定位DOS核心.MSDOS.SYS 至最后地址.分配 MS-DOS 磁盘高速缓冲区及文件控制块区域.

至此，DOS 核心已被初始化，所有驻留的设备驱动程序已可使用。SYSINIT 可调用普通的 MS-DOS 文件服务功能去打开 CONFIG.SYS。此文件包含各种命令，使用户可配置 MS-DOS 环境。例如，用户可指定附加的硬件设备驱动程序、磁盘缓冲区数量、可同时打开文件的最大数量、命令处理程序的文件名等。

如果找到了 CONFIG.SYS，便将其装入内存进行处理：所有小写字符都转换为大写字符，一次一行地解释文件并作为命令来处理，为磁盘高速缓冲区及内部文件控制块或文件和记录系统功能用的句柄（Handle）分配内存，把 CONFIG.SYS 文件中给出的设备驱动程序顺序地装入内存，由调用其 init 模块进行初始化，并连接到设备驱动程序表中。每个驱动程序的 init 功能通知 SYSINIT 应为该驱动程序保留多少内存。

装入所有可安装的设备驱动程序后，SYSINIT关闭所有文件句柄并重新打开控制台（CON）、打印机（PRN）、辅

助设备（AUX）作为标准输入、标准输出、标准出错、标准列表及标准辅助设备。它使用户安装的字符设备可覆盖掉 BIOS 中用于标准设备的驻留驱动程序。

最后，SYSINIT 调用 MS-DOS 的 EXEC 功能以装入命令解释程序（系统中默认的命令解释程序是 COMMAND.COM，但亦可通过 CONFIG.SYS 文件换成另一个解释程序）。一旦装入命令解释程序，便显示出提示信息并等待用户打入命令，此时 SYSINIT 模块被撤销。典型系统的 MS-DOS 启动过程的最后结果见图 1-5。

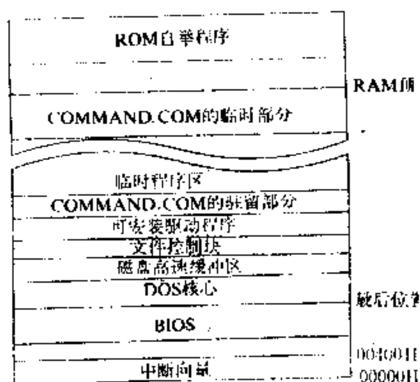


图 1-5 COMMAND.COM 驻留部分处于低内存地址区。临时部分包含批文件处理程序和内部命令，处于高内存区，该区可被运行在临时程序区中的外部命令和应用程序所覆盖。

1.2 两类基本的 MS-DOS 程序

在 MS-DOS 下运行的程序可分成两种基本类型，一类是 COM 程序，最大不超过 64K，另一类是 EXE 程序，可大至可用内存的容量。按照 Intel8086 的术语，COM 程序符合小模式（Small model），在此情况下所有段寄存器含

有同样的值，即代码与数据混合在一起。EXE 程序符合中或大模式（Medium，Large Model），段寄存器含有不同的值，即代码、数据、堆栈驻留在不同的段中，甚至可有多个代码和数据段。EXE 程序使用长调用及操作数据寄存器(DS) 来寻址。

驻留在磁盘上的 COM 类程序是绝对的内存映像，文件扩展名为.COM。该文件既无文件头也没有任何其他的内部标识信息。而.EXE 程序驻留在磁盘上，它是一种特殊的文件类型，带有一个文件头、一个重定位图、一个检查和以及 MS-DOS 使用的其他信息。

COM 及 EXE 程序进入内存执行所采用的是相同的机构：MS-DOS装入时用 EXEC 功能。EXEC 可用 COMMAND.COM 装入程序之文件名来调用，亦可由其他的命令解释程序或用户接口，或由以前用 EXEC 装入的另一个程序来调用。如果在临时程序区有足够的自由内存可用，EXEC 便分配一个内存块以容纳新的程序，在其基础上建立程序段前缀，然后将程序读入内存中 PSP 的直接上方。最后，EXEC 设置段寄存器和堆栈，并将控制传送给程序。调用 EXEC 时，可提供其他有关的信息，如命令 tail、文件控制块及环境块的地址。这些信息都将被传送给新程序。

COM 及 EXE 程序通常被称为"临时程序"。一个临时程序"拥有"已分配给它的内存块并在其执行过程中几乎完全获得对系统全部资源的控制。程序结束时，释放该内存块(因此称其为"临时"程序)，以供下一个装入的程序使用。

1.2.1 使用.COM 格式

COM 文件是内存中程序的准确映像。它是一类较小、

装入较快且较简单的程序．虽然 Microsoft 正在取消 COM 文件，但目前它仍是用汇编语言编写小型、快速、实用程序的较好格式．COM 文件亦可作为一个大型应用程序的前导程序.用户可键入一个小型COM文件的名字，然后由它来装入主程序．

但必须看到，COM 文件有两个重要限制:

(1)代码和初始化后的数据总量不可超过 64K，如果超过了此限制，将无法用 EXE2BIN 实用程序把 EXE 文件转换成 COM 文件．

(2)DOS 装入程序不能为 COM 文件执行段重定位．而对于 EXE 文件，该装入程序将使实际的运行时间段值赋予程序代码中要求的地址．此特点使程序员可将数据装入段寄存器，例如:

```
mov ds, dataseg
```

关于 COM 文件还有几点说明．其一是通常 COM 文件不可具有一个以上的段，其二是它不能包含堆栈段．下面的程序清单提供了生成一个较特殊的、多段的．COM 文件的基本框架 (shell)．对代码和数据使用不同的段是一个好办法．

此框架开始时对三个段:stackseg.codeseg 和 dataseg.加以说明．说明 stackseg 段仅用于连接程序以免在其找不到堆栈段时会显示出错信息，此段必须为空段且不被引用．

然后说明代码和数据段，迫使连接程序将代码段放在数据段前面．装入次序由首先引用的段次序来决定．在 COM 文件中，代码段必须位于文件的最前面（堆栈段实际上在最前面，但它是空的)．

然后将代码段和数据段分配给称为 groupseg 的组段．考虑下面的数据引用指令:

```
page 50,130

;File:    COMFMT.ASM

;Format for a multisegment .COM file
;
;    To generate .COM file from TEST.ASM:
;         MASM TEST;
;         LINK TEST;
;         EXE2BIN TEST.EXE TEST.COM

                                        ;Declare segments to force
                                        ;loading order.

stackseg   segment stack               ;Placate linker.  This segment must
stackseg   ends                        ;remain empty.

codeseg    segment
codeseg    ends

dataseg    segment                     ;Place all data in this segment.
num_var    db      ?
message    db      'hello from the data segment',10,13,'$'
buffer     label   byte
dataseg    ends

groupseg   group   codeseg,dataseg
           assume  cs:groupseg
           assume  ds:groupseg
           assume  es:groupseg

codeseg    segment

           org     5ch                 ;Template for Program Segment Prefix.
parm1      db      12 dup (?)          ;Parameters from command line.
           org     6ch
parm2      db      12 dup (?)
           org     80h
parmlen    db      ?                   ;Length of command line.
parmline   db      127 dup (?)         ;Full command line.

           org     100h
main       proc    near
                                             ;Main code.
           mov     ah, 09h                   ;Access data segment.
           mov     dx, offset groupseg:message
           int     21h                       ;Print message with DOS.

           mov     ah, 4ch                   ;Exit w/ errorlevel 0.
           mov     al, 0
           int     21h

main       endp

subr1      proc    near
                                             ;Subroutine code.
           ret
subr1      endp

codeseg    ends

           end     main
```

mov ah, num－var

正常情况下，象 num－var 这样的数据项被此类指令引用时，汇编程序从包含它的段的开始产生变量的偏移量，此情况下应是 0. 但由于这是一个 COM 文件，数据段寄存器指向代码段的开始（所有段寄存器都一样），而不是指向数据段的开始，因此，要求有从代码段开始的 num－var 的偏移量．将数据段分配成一个组，则此类指令的偏移量自动从组中第一个段的开始产生，此时正好就是代码段。

虽然数据引用指令可自动地正确处理，但偏移量操作数必须使用组段前缀，如

mov dx, offset groupseg:message

无此前缀，所产生的偏移量将从数据段开始而不是从组开始。

文件中的下一项是实际的数据段 dataseg．注意，因为数据段出现在源文件代码段之前，故所有数据项均在文件头部列在一起．

要记住，在最后的可执行文件中，数据段在代码段之后，这种安排比将数据放在代码段前部，然后跳过它的那种常用技术好．因为如需用一个大的未初始化的缓冲区，此缓冲区可作为一个标号（如清单中的 buffer）置于数据段的尾部，而无保留空间（注意，在 COM 文件尾部前的所有用户内存在装入时均自动分配给了该程序）。如果此缓冲区在代码段开始处进行了说明，将保留下空间，但增加了磁盘上 COM 文件的长度（增加部分为缓冲区容量）．记住，堆栈指针初始化时指向 64K 程序段的顶部，如果会影响缓冲区，便应重定位该堆栈。

清单的最后一部分是代码段．在 org 100h 前面出现的

标号用于访问程序段前缀。当一个 EXE 文件转换成 COM 文件时，删除开始的 100h 个字节（在由 end 语句指定程序进入点之前的所有代码或数据全被删除），故可执行文件中出现的第一个字节将是主程序的第一条指令。

下面的批处理文件 A2C.BAT 自动生成一个 COM 文件

```
if exist %1 goto end
masm %1.asm
if not error level 1 link %1;
if not errorlevel 1 exe2bin %1.exe %1.com
del %1.obj
del %1.exe
:end
```

欲处理 TEST.ASM,可打入:

A2C TEST

1.2.2 EXE 程序结构

EXE文件的格式比COM文件的更灵活,适合于超过64K的程序,更易与将来的操作系统环境兼容,而且只有EXE文件方可用 Microsoft Codeview 检错程序进行符号 debug。下面的清单提供了一个样板，可用于从汇编语言生成 EXE 文件，并且适合于作符号 debug。

该清单说明EXE格式与 COM 格式有相当的三个段,但也有一些重要差别:

· 与 COM 文件不同，这里堆栈段是有用的，并且为堆栈保留空间。装入程序自动初始化堆栈段寄存器以指向此段的基底，而堆栈指针指向顶部。

· 在 EXE 文件中，数据段和附加段寄存器初始化后指

```
page 50,130

;File:    EXEFMT.ASM

;Format for generating an .EXE file
;
;    To generate .EXE file from EXEFMT.ASM:
;         MASM EXEFMT;
;         LINK EXEFMT;

public     message                      ;Declare all symbols 'public'
public     main                         ;for CodeView debugger.

codeseg    segment 'CODE'               ;Declare code segment to force it
codeseg    ends                         ;to load first.

dataseg    segment 'DATA'               ;Place all data in this segment.
message    db      'hello from an .EXE file',10,13,'$'
dataseg    ends

stackseg   segment stack 'STACK'        ;Set up 1K stack.
           db      128 dup ('stack   ')
stackseg   ends

           assume  cs:codeseg
           assume  ds:dataseg
           assume  es:dataseg
           assume  ss:stackseg

codeseg    segment 'CODE'

main       proc    far

           mov     ax, dataseg                  ;Set up segment registers.
           mov     ds, ax
           mov     es, ax

           mov     ah, 09h                      ;Access data segment.
           mov     dx, offset message
           int     21h                          ;Print message with DOS.

           mov     ah, 4ch                      ;Exit w/ errorlevel 0.
           mov     al, 0
           int     21h

main       endp

codeseg    ends

           end     main
```

向一个单独的段 dataseg，由于装人程序为 EXE 文件执行重定位，故将 dataseg 分配给这些寄存器是可能的。又因为数据段寄存器指向一个单独的段，故不必再去说明一个段组，并有可能使用整整 64K 的数据和整整 64K 的代码（如果还说明了其他段，则还可用得更多）。

· 数据指令与 COM 文件中的相同，但偏移量算子不再使用组前缀。

· 程序的大小可任意。如果代码段超过了 64K，只需

再生成一个代码段（在段间用长调用分支）即可。类似地可说明另外的数据段，但数据段寄存器必须始终包含当前被访问段的基地址。

EXE 程序总是由 MS-DOS 装入程序装入内存中，直接处于程序段前缀上方，虽然代码、数据及堆栈段的次序是可以改变的。

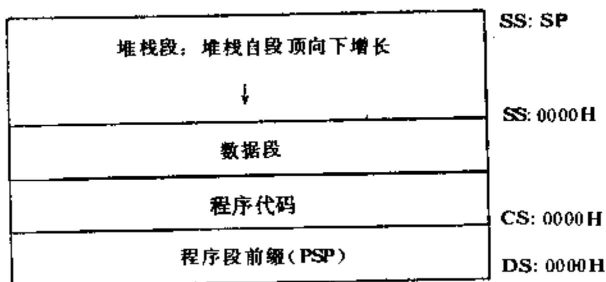


图 1-6 典型 EXE 类型程序刚装入后的内存映象。EXE 文件的内容被重定位，并被装入内存中 PSP 上方。代码，数据和堆栈驻留在不同的内存中，次序不必与图中相同。程序进入点可在代码段中任意位置，由主模块中 END 语句指定。程序接受控制后，DS 和 ES 寄存器指向PSP，程序通常保存此值，然后重置 DS，ES 寄存器以指向数据区。

EXE 文件有一个文件头（headcr），或称为控制信息块，它具有带特征的格式。此文件头的大小随在装入时需重定位的程序指令数而变，但总是 512 字节的倍数。

MS-DOS 将控制传送给程序之前，代码段（CS）寄存器和指令指针（IP）寄存器的初始值根据 EXE 文件头中进入点信息及程序的装入地址计算后得到。此信息是由源代码中的 END 语句推算出的。数据段（DS）及附加段（ES）

寄存器指向程序段前缀，故程序可访问环境块指针、命令tail以及PSP中包含的其他有用信息。

字节偏移量	
0000H	EXE文件标识第一部分(4DH)
0001H	EXE文件标识第二部分(5AH)
0002H	模512的文件长度
0004H	包括文件头在内的,以512字节页计的文件大小
0006H	重定位表项数
0008H	以段落(16字节)计的文件头大小
000AH	程序以上所需最小段落数
000CH	程序以上所需最大段落数
000EH	堆栈模块的段位移
0010H	SP寄存器进入时的内容
0012H	检查和字
0014H	IP寄存器进入时的内容
0016H	代码模块的段位移
0018H	文件中第一个重定位项的偏移量
001AH	覆盖号(程序驻留部分为0)
001BH	可变保留空间
	重定位表
	可变保留空间
	程序和数据段
	堆栈段

图1—7 EXE装入模块的格式

堆栈段(SS)及堆栈指针(SP)寄存器的初始内容来自于文件头。此信息根据程序源代码中某处的属性STACK段的说明推算得到。分配给堆栈的内存空间可被初始化或不

初始化，这取决于堆栈段的定义，许多程序员喜欢使用某个统一的数据来初始化堆栈内存，以后可方便地检查内存转储情况并确定程序实际使用了多少堆栈空间。

一个 EXE 程序结束处理后，应通过 Int21H 功能 4CH 将控制返回给 MS-DOS，此功能还可向调用它的程序、命令解释程序或批处理文件返回一个代码．通过 Int20H,Int21H 功能 0，及一个 NEAR RETURN 退出 (因为在进入时一个字 0 被推入堆栈，故 NEAR RETURN 使传送到 PSP:0000,该处即为Int20H指令)时，均要求 CS 寄存器指向程序段前缀，这不是一种好办法，并且会与将来的 MS-DOS 版本不兼容。

对于 EXE 类型程序的链接程序 (Linker) 可有许多单独的目标模块．每个模块可使用唯一的代码段名字，过程的属性既可以是 NEAR 的，也可以是 FAR 的，这取决于命名惯例及可执行代码的大小．程序员必须注意链接在一起的模块只能包含一个带 STACK 属性的段，以及一个由 END 汇编伪指令定义的进入点。Linker 的输出是一个带.EXE 扩展名的文件，此文件可直接执行。

1.3 内存管理基础

当前版本的 MS-DOS 可以管理多达 1MB 的邻接 RAM．在 IBMPC 及其兼容机上，由 MS-DOS 和其他程序所占的内存在地址 0000H 处开始，并可达地址 09FFFFH，此 640KB 的 RAM 区域有时称为"常规内存区"。此地址以上的内存区是为 ROM 硬件驱动程序、视频刷新缓冲区等所保留的。非 IBM 兼容机有可能使用其他的内存区段。

在 MS-DOS 控制下的 RAM 区可分成两个主要部分:

· 操作系统区域

· 临时程序区域

操作系统区域在地址 0000H 处开始，即它占用 RAM 的最低部分。该处有中断向量表、操作系统模块及其所拥有的表（table）和缓冲区，以及 CONFIG.SYS 文件中指定的附加可安装设备驱动程序及 COMMAND.COM 命令解释程序的驻留部分。操作系统区所占的内存量随 MS-DOS 版本、磁盘缓冲区数、可安装设备驱动程序个数与大小而改变。

临时程序区为 RAM 中所存操作系统区上面的剩余部分，是动态可分配的内存。MS-DOS 为临时程序区（TPA）中所分配的每个内存块（chunk）维护一个特殊的控制块，这些块是链接在一起的。有三个 MS-DOS 功能可被调用来从 TPA 分配和释放内存块，它们是：

功能	作用
48H	分配内存块
49H	释放内存块
4AH	

这些功能当用 COMMANDCOM 请求从磁盘上装入一个程序或外部命令时由 MS-DOS 本身使用。MS-DOS 的程序装入器EXEC功能，可调用功能48H，为被装入程序的环境分配一个内存块，另一内存块分配给程序本身及其程序段前缀，然后从磁盘上将程序读入已分配的内存区。程序结束后，MS-DOS 调用功能 49H 释放这两个内存块，然后将控制返回给命令解释程序。

MS-DOS 内存管理功能亦可由临时程序用来动态管理 TPA 中可用的内存。正确地使用这些功能是实现 MS-DOS

下"规矩"程序设计最重要的准则之一.

1.3.1 使用内存管理功能调用

内存分配功能以两种常见方式使用:

(1)压缩程序的内存分配，故可在其控制下有足够的空间装人和执行另一个程序.

(2)动态分配程序所需要的附加内存，不需要时将这些内存再释放掉.

▲压缩内存分配

许多 MS-DOS 应用程序通常假定它们拥有所有内存.此假定源于 CP/M，该操作系统在任意给定时刻只支持唯一的一个活动进程."规矩"的 MS-DOS 程序仅分配给自己实际使用的内存，而释放掉不需要的内存.

但在当前版本的 MS-DOS 下，一个程序将拥有的内存量难以预先确定.一个程序第一次装人时分配给它的内存量取决于两个因素:

- 所装人程序的文件类型;
- TPA 中可用内存量.

装人 COM（内存映像）文件的程序总是分配掉所有的 TPA.因为 COM 程序没有文件头可向 MS-DOS 传送段和内存使用的信息，故 MS-DOS 假定了最恶劣情况，并将所有内存供该程序使用.只要 TPA 中所剩空间够文件长度上用于 PSP 的 256 个字节及用于堆栈的 2 个字节，MS-DOS 便装人该程序.接受控制后，由 COM 程序负责确定是否有足够的内存可供完成其功能.

从 EXE 文件装人的程序在分配内存时遵照更为复杂的规则.首先，TPA 应具有足够的空间以容纳代码、数据和堆栈段.此外，在 EXE 文件头中有两个由 Linker 设立的域

用于通知 MS-DOS 有关该程序的内存需求。第一个域称为 MIN-ALLOC，定义了除用于代码、数据及堆栈段以外，该程序所需要的最小段落数。另一个叫做 MAX-ALLOC，定义了该程序将使用附加内存的最大段落数。

装入一个 EXE 文件时，MS-DOS 首先试图按照 MAC-ALLOC 段落数加上程序本身所需的段落数来分配内存。如果没有那么多内存可供使用，则只要所剩内存够 MIN-ALLOC 指定的量加上映像文件大小，则 MS-DOS 便分配掉所有自由内存。如果上述条件得不到满足，该程序便不能执行。

一旦装入并运行了 COM 或 EXE 程序，便可用 SETBLOCK（修改内存块功能）去释放不立即使用的所有内存。在程序从 MS-DOS 接受控制后，可通过调用 Int21H 功能 4AH 来实现这一目的，此时应在寄存器 ES 中置入程序段前缀的段地址，在 BX 中置入该程序所需要的段落数。

```
        main  qroc  far         ;DOS进入点
                                ;COM和EXE文件接受控制后，DS和
                                 ES寄存器均指向程序前缀
        mov  sp,  offset stk    ;COM程序应将其堆栈移至安全区
        mov ah, 4ah             ;功能 4 AH = 修改内存块
        mov bx, 400h            ;保留400H个段落
        int  21h                ;传送给DOS
        jc  error               ;如果功能失败则跳转
error:
        dw  46  dup (? )
        stk   equ   $           ;新堆栈基址
```

▲动态分配附加内存

如果程序需要附加内存空间，例如用于I/O缓冲区或中间结果数组，可调用 Int21H 功能 48H（分配内存块）分配给所要求的段落数。如果有足够大的未分配内存块可用，MS-DOS 将返回分配给区域的段地址并清除标志寄存器(0)，表示此功能已成功见下面动态内存分配示例。

如果没有足够大的未分配内存块可用，MS-DOS 将设置寄存器（1），在 AX 寄存器中返回出错代码，在 BX 寄存器中返回最大可用块的大小（以段落计）.

```
mov ah, 48h              ;功能 48H = 分配内存块
mov bx,0800h             ;800H 段落 = 32K 字节
int 21h                  ;传送给 DOS
jc error                 ;如果分配失败,则跳转
mov  buff-seg.ax         ;保有分配块的段地址
 :
mov es,buff-seg          ;ES.DI = 块地址
xor di,di                ;
mov  cx, 08000h          ;储存 32768 字节
mov  al,offh             ;以-1 填充缓冲区
cld                      ;
rep stosb                ;执行快速填充
 :
mov cx,08000h            ;以字节计的长度
mov bx,handle            ;以前打开文件的句柄
push  ds                 ;保存数据段
mov ds,buff seg          ;使 DS:DX = 缓冲区地址
mov dx,o                 ;
mov ah,40h               ;功能 40H = 写
```

```
int-21h                    ;传送给 DOS
pop  ds                    ;恢复数据段
jc  error                  ;如果写失败则跳转
:
mov es,buff seg            ;ES＝以前分配块的段
mov ah,49h                 ;功能 49H＝释放内存块
int 21h                    ;传送给 DOS
jc  error                  ;如果释放失败则跳转
:
error:
:
handle  dw  o
buff seg dw  o
```

程序可使用 BX 寄存器中返回的值确定是否能够以“降级”方式工作以使用更少的内存。如果可以，需再次调用功能 48H 以分配较小的内存块。

程序完成后应使用功能 49H 去释放该内存块。

1.3.2 内存控制块

Microsoft 迄今尚未公布内存控制块的内部结构，其目的可能是要阻止程序员直接去操作内存分配，而不使用已有的 MS-DOS 调用。

内存控制块的内部结构（按 UNIX／XENIX 术语称为界标—arena header）对于 MS-DOS 版本 2 与 3 是一样的，长度均为 16 字节（一个段落），直接处于所控制内存区之前部，如图1－8。

一个界标——arena header 包括：

• 一个字节表示此为整个标题（header）链中的一个成员或是最后一个登记项。

• 一个字表示所控制的区域是可供使用的还是已分配给了某个程序（如果是后者，则字指针指向程序的 PSP）。

• 一个字含有受控内存区域(内存arena)的大小(以段落计)。

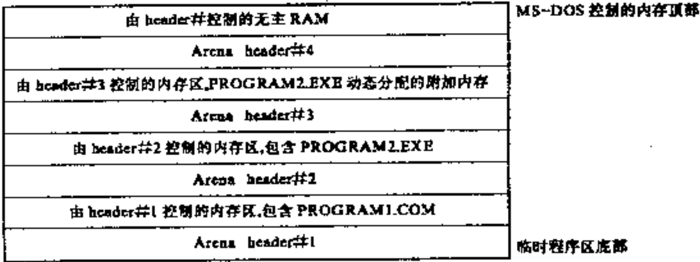


图 1－8 MS-DOS arena header（内存控制块）与临时程序区结构图.

为简明起见,图中未带有环境块及其相关的 header.

只要对内存块有分配、修改或释放功能被请求，或一个程序被 EXEC 调用或结束，MS-DOS 均检查该 arena header 链。如果有一个块表现出已被破坏或链条被破坏，MS-DOS 将显示出信息:

Memory allocation error

而且系统死机。在上例中,PROGRAM1.COM 由 COMMAND.COM 装入 TPA,由于它是一个 COM 文件,故分配掉所有 TPA, 由 arena header# 1 控制．PROGRAM1.COM 接着使用功能 4AH （修改内存块）将其内存分配量压缩到运行时实际需要的量，然后再使用 EXEC 功能（Int 21H 功能 4BH）装入并执行 PROGRAM2.EXE。EXEC 功能获得一定量的由 arena header#2 控制的内存,并将 PRGORAM2.EXE 装入其中。继而 PROGRAM2.EXE 需要一些附加内存存贮某些中间结果，故其调用功能 48H（分配内存块）以获得由 arena header#3 控制的区域。最高的 arena header#4 控制尚未分配给任何程序的所有剩余的 TPA。

1.3.3 内存图

下表为典型 IBM-PC 兼容系统中内存惯用的轮廓。段及偏移量两项都给出了所述每个区域的起始地址，说明大小之项也给出了以字节计时所占用的内存量。

段	偏移量	大小	说明
0000h	0000h	1024	中断向量表
	0400h	172	BIOS 通信区
	04ACh	68	IBM 保留
	04F0h	16	用户通信区
	0500h	256	DOS 通信区
	0600h	变化	操作系统(下列文件名适用于PC-DOS) ▲IBM BIO.COM(DOS与BIOS的接口) ▲IBM DOS.COM(DOS中断处理程序及中断 21h 服务程序) ▲DOS缓冲区，控制区及设备驱动程序 ▲COMMAND.COM(驻留部分) ▲中断22h，23h，24h处理程序，以及用于重新装人 COMMAND.COM 临时部分的程序。 ▲内存驻留程序 ▲临时应用程序

1.4 MS-DOS EXEC 功能

MC-DOS EXEC 功能（4BH）允许一个程序（称为父进程）从存储设备中装人另一个程序（称为子进程）并执行它，子进程结束后再获得控制。

			▲COMMAND.COM 的临时部分
9000h	FFFF		用户内存区最高地址
A000h	0000	128k	ROM 保留区(段 A0000h 及 B000h)或 EGA 区域开始
B000h	0000h	4000	单显视频内存
	0800h	16k	彩色图形适配器视频内存
C000h	0000h	192k	ROM 扩展及控制区(段 C000h,D000h,E000h)
	8000h		硬盘 ROM 区
D000h			ROM 扩展及控制区
E000h			ROM 扩展及控制区
F000h	0000h	16k	ROM 保留
	4000h	40k	ROM BASIC
	E000h	8k	ROM BIOS

父进程可从命令行上或默认文件控制块、或一组称为环境块的字符串中将信息传送给子进程。使用句柄扩充文件管理调用打开的父进程的所有文件或设备在新生成的子任务中全被复制，即子进程继承父任务的所有活动句柄。子进程对这些句柄的任何文件操作（如文件 I／O），也影响到与父进程句柄有关的文件指针。

子进程结束工作后，可向父进程传送退出代码，指示是否遇到了任何错误，亦可装入其他程序，并可有多级控制，直至系统内存用尽。

MS-DOS 版本 2 和 3，不支持多任务，在子进程结束前，父进程的执行简单地“挂起”。但设计 EXEC 功能及其他MS-DOS功能时在已考虑到避免与地址和时钟信号相关。

MS-DOS 命令解释程序 COMMAND.COM 使用 EXEC 功能运行其外部命令及其他应用程序。许多常用的商用程序，如数据库管理程序、字处理程序等，使用 EXEC

运行其他程序或装入COMMAND.COM的第二个拷贝，从而使用户能够列出目录、拷贝文件、换名文件，而用不着关闭应用程序文件，也不必停止进行中的主要工作。

1.4.1 获得可用内存

父进程欲使用 EXEC 功能装入子进程，则必须在临时程序区中有足够的未分配内存可用。父进程本身装入时，根据原始文件类型是 COM 还是 EXE，以及由装入程序提供的其他信息分配一个变化的内存量。由于操作系统尚无可靠的方法能够预测某一程序所需内存量的大小，通常分配给程序的内存量要比实际需要多得多。因此，一个好的父进程第一位工作应是向 MS-DOS 释放其所拥有的多余内存。父进程使用 Int21H 功能 4AH（修改内存块）实现不需用内存的释放。在此情况下，应使寄存器 ES 指向欲释放内存之程序的程序段前缀，而使寄存器 BX 包含该程序所要保留的内存段落数，再发出功能 4AH 调用。

注意:COM 程序必须确保使其堆栈移至安全区域，方可将其内存分配减少至 64kb 以下。

1.4.2 请求 EXEC 功能

一旦获得了足够的自由内存，父进程便可由调用 EXEC 系统服务功能（Int21H 功能 4BH）来装入和执行一个子进程。寄存器 AL 中如为子功能代码 0，则为在未分配内存中装入和执行一个程序;如为子功能代码 3，则为在已属于调用程序的特定地址处装入一个覆盖模块。

调用 EXEC 功能时:

DS:DX 指向欲运行程序的路径名;

ES:BX 指向一个参数块。

该参数块包含 EXEC 功能所需的其他信息。

1.4.2.1 程序名

由调用程序提供给EXEC功能的欲被运行的程序名，必须是一个明确的文件说明，不可使用通配符（Wildcard characters）且必须具有 COM 或 EXE 扩展名。如果在程序名中未提供路径和磁盘驱动器，便使用当前子目录和默认的磁盘驱动器。在 PATH 变量所列出的目录中按顺序搜寻 COM，EXE 及 BAT 文件，虽然不是 EXEC 的功能，但 COMMAND.COM 内部是按此处理的。

不可以直接地执行一个批处理文件，必须首先执行一个 COMMAND.COM 的拷贝，然后将批处理文件名传送命令 tail，并伴随以／C 开关。

1.4.2.2 参数块

参数块包含如下四个数据结构：

- 环境块
- 命令 tail
- 两个命令控制块

参数块为环境块地址所保留的空间长仅 2 个字节，包含一个段地址，其余三个地址均是双字地址，长 4 个字节，前二个字节是偏移量，后二个字节是段地址。

▲环境块

由 EXEC 功能装入的程序均从其父进程继承一个称为环境块的数据结构。在程序段前缀中偏移量 002CH 处可找到指向该块段地址的指针。环境块装有系统命令解释程序使用的某些信息，也可能装有临时程序使用的某些信息。它们对操作系统的正常工作无影响。

如果 EXEC 参数块中的环境块指针为 0，则子进程将获得父进程环境块的拷贝，亦可由一个段指针指向一组不同

的或扩展的字符串．环境块最大可为 32 kb，这意味着借助于此机构可以在程序之间传送非常大量的信息．

对于任何给定的程序，环境块都是静态的．这意味着如果在 RAM 中驻留有多代子进程，每个子进程均有自己独立的环境块拷贝，也就是说，内存驻留程序的环境块不会被后面的 PATH 和 SET 命令所更改．

环境块总是"段对齐"的（即开始地址总是 16 字节的整倍数），包含一串ASCIIZ字符(以二进制0 结尾的ASCII 字符串)．每个字符串形如：

NAME＝PARAMETER

用附加的 0 字节表示整组字符串的结尾．在 MS-DOS 版本 3.X 下，环境块字符串及附加 0 字节后跟一个字计数，及 EXEC 装入该程序所用的完整的驱动器、路径、文件与扩展名．

正常条件下，一个程序继承的环境块至少包含三个字符串：

COMSPEC＝变量

PATH＝变量

PROMPT＝变量

系统初始化时这三个字符串在解释 CONFIG.SYS 及 AUTOEXEC.BAT 文件时置入环境块中．它们通知 MS-DOS:命令解释程序 COMMAND.COM 可执行文件的地址（用于重新装入临时部分）;去何处搜寻可执行外部命令或程序文件;以及用户提示的格式等等．

在环境块中可加入其他字符串，既可在批处理文件中也可交互地使用 SET 命令加入．它们仅可由临时程序作为传递信息使用．例如，Microsoft C 编译程序在环境块中查找

INCLUDE，LIB,TMP 字符串以确认去何处找到它所要用的 include 文件及库文件，以及到何处建立其临时工作文件等。

▲命令 tail

命令 tail 被拷贝到子进程 PSP 偏移量 0080H 处。其形式为一个计数字节后跟一个 ASCII 字符串，以回车结尾(该回车不包括在计数中)。

命令 tail 可包括文件名、开关或其他参数。从子进程的观点看，如果在 MS-DOS 提示符下直接由用户命令运行程序，则命令 tail 应提供同样的信息。EXEC 将忽略命令 tail 中的任何 I／O 重定向参数，标准设备的重定向必须由父进程在发出 EXEC 调用之前提供。

▲默认省缺的文件控制块

由 EXEC 参数块指向的两个默认文件控制块拷贝到子进程 PSP 偏移量 005CH 及 006CH 处。如果不需要默认文件控制块 FCB，可由在参数块的 FCB 指针处置-1 (0FFFFH) 来省略之。

但从子进程的观点看，如要真正仿真 COM——MAND.COM 的功能，应使用功能 29H（系统文件名语法检查服务功能）在调用 EXEC 功能之前对命令 tail 中默认文件控制块中的前两个参数进行语法检查。文件控制块在 MS-DOS 版本 2.x 和 3.x 下用得不多，因为它们不支持树状文件结构，但有些应用程序还是用检查文件控制块来作为获得命令 tail 中前两个开关或其他参数的一种快速方法。

1.4.2.3 从 EXEC 功能返回

与多数其他的 MS-DOS 功能调用不同，EXEC 将破坏

除码段寄存器 CS 及指令指针 IP 以外的所有寄存器内容。因此在发出 EXEC 调用之前，父进程必须将其他重要的寄存器内容推入堆栈，然后将堆栈段寄存器 SS 及堆栈指针 SP 保存在码段内可访问到的变量中。当成功的 EXEC 调用返回时（即子进程结束执行），父进程应把所保存的变量重新装入SS和SP，然后再将其他保存的寄存器从堆栈中弹出。

最后，父进程可使用 Int21H 功能 4DH（得到返回代码）获得由子进程传回的成功或失败的代码。

如果出现以下情况，则 EXEC 功能调用失败：

·无足够的未分配内存可用于装入和执行所请求的程序文件；

·所请求的文件在磁盘上未找到；

·RAM 最高区中的 COMMAND.COM 临时部分（包含实际的装入程序）已被破坏，且已无足够的自由内存重新装入它（仅适用于 MS-DOS 版本 2）。

下表总结了功能 4BH 的调用惯例。

调用	
AH	＝4BH
AL	＝功能类型
	00＝装入并执行程序
	03＝装入覆盖模块
ES:BX	＝参数块的段:偏移量
DS:DX	＝程序参数的段:偏移量

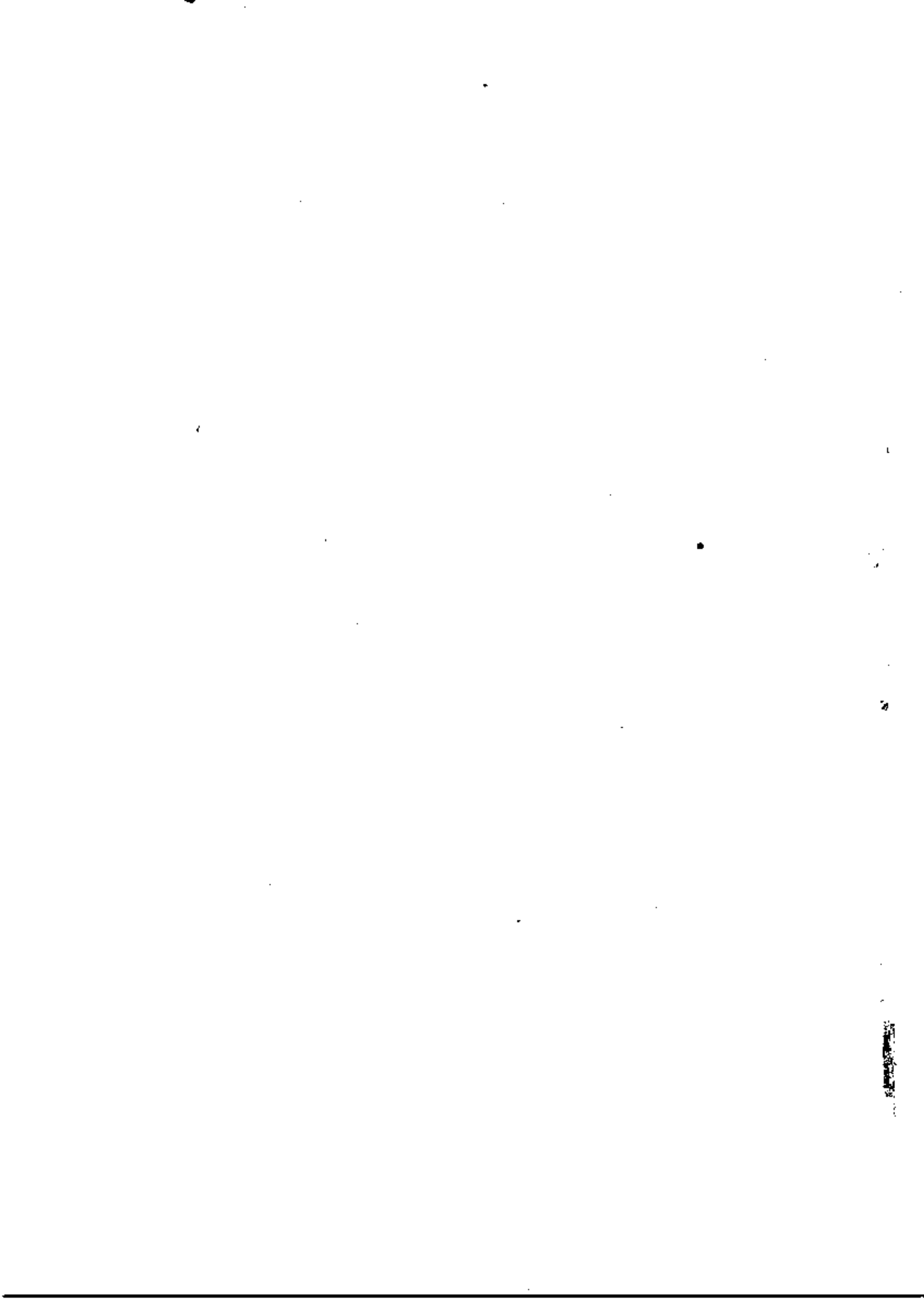
返回

·如果调用成功：

进位标志清0，除CS,IP外的所有其他寄存器（包括堆栈指针）均被破坏。

·如果调用失败：

进位标志置位且AX＝出错代码。



```
par_blk         dw      envir                   ;segment address of
                                                ;environment descriptor
                dw      offset cmd_line         ;address of Command Line
                dw      seg cmd_line            ;to be passed to program
                dw      offset fcb1             ;address of default File
                dw      seg fcb1                ;Control Block #1
                dw      offset fcb2             ;address of default File
                dw      seg fcb2                ;Control Block #2

cmd_line        db      4,' *.*',cr             ;actual command line to
                                                ;be passed to EXEC'd pgm
fcb1            db      0                       ;File Control Block #1
                db      11 dup ('?')
                db      25 dup (0)
fcb2            db      0                       ;File Control Block #2
                db      11 dup (' ')
                db      25 dup (0)

envir           segment    para 'ENVIR'         ;Environment Descriptor

                db      'PATH=',0               ;empty search path
                                                ;location of COMMAND.COM
                db      'COMSPEC=A:\COMMAND.COM',0
                db      0                       ;end of environment block
envir           ends
```

此例已包括了所有必需的变量和命令块，注意：保存寄存器ＳＳ和ＳＰ所用的两个变量必须在码段中，其它数据项可置于数据段中。

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F 0123456789ABCDEF
0000 43 4F 4D 53 50 45 43 3D 43 3A 5C 43 4F 4D 4D 41 COMSPEC=C:\COMMA
0010 4E 44 2E 43 4F 4D 00 50 52 4F 4D 50 54 3D 24 70 ND.COM.PROMPT=$p
0020 24 5F 24 64 20 20 20 24 74 24 68 24 68 24 68 24 $_$d   $t$h$h$h$
0030 68 24 68 24 68 20 24 71 24 71 24 67 00 50 41 54 h$h$h $q$q$g.PAT
0040 48 3D 43 3A 5C 53 59 53 54 45 4D 3B 43 3A 5C 41 H=C:\SYSTEM;C:\A
0050 53 4D 3B 43 3A 5C 57 53 3B 43 3A 5C 45 54 48 45 SM;C:\WS;C:\ETHE
0060 52 4E 45 54 3B 43 3A 5C 46 4F 52 54 48 5C 50 43 RNET;C:\FORTH\PC
0070 33 31 3B 00 00 01 00 43 3A 5C 46 4F 52 54 48 5C 31;....C:\FORTH\
0080 50 43 33 31 5C 46 4F 52 54 48 2E 43 4F 4D 00 20 PC31\FORTH.COM.
```

上图是MS-DOS 版本 3 下典型的环境块。本例中包含有默认的COMSPEC参数和两个复杂的PATH和 PROMRT命令。注意可执行程序的文件标识符跟在偏移量0073H处两个0之后，而0073 H表示环境块结束。

1.4.3 实例程序 SHELL.C 及 SHELL·ASM

作为 MS-DOS EXEC 功能的使用例子，下面给出一个称为 SHELL 的命令解释程序（简单表格驱动的命令解释程序），源程序用 Microsoft C 编译。

SHELL 程序由表格驱动，不难扩充成几乎可为任何应用程序提供的强有力的用户接口。SHELL 获得对系统的控制后，显示出提示:sh:，并等待用户输入。用户打入一行后，以回车结束，SHELL 在其内部命令表中试找第一个匹配项。如果未找到匹配项，则调用 MS-DOS EXEC 功能并将用户输入，使用/ C开关传送给 COMMAND.COM，在其控制下将 COMMAND.COM 作为临时命令处理程序。

本清单中的 SHELL 只认识下面三条内部命令：

命令	作用
CLS	使用 ANSI 标准控制序列清单显示屏并使光标归 0
DOS	运行 COMMAND.COM 的拷贝
EXIT	退出 SHELL,将系统控制返回给下一级命令解释程序

SHELL 源程序中，可以容易地加入新的内部命令。程序员只需编一段合适的程序，并将该程序的名字，随同定义该命令的文本字符串插入名为 COMMANDS 的表中。此外 SHELL 还可简单地阻止某些“危险”命令的传送（如 MKDIR 或 ERASE），只需将命令名字排除出内部命令表并打印出错误信息即可。

```
/*
        SHELL.C         a simple user-extendable command
                        interpreter for MS-DOS 2.X and 3.X

        Copyright (C) 1985 Ray Duncan

        To compile with Microsoft C 3.0 and
        link into the executable file SHELL.EXE:

                        C>MSC SHELL;
                        C>LINK SHELL;
*/

#include <stdio.h>
#include <process.h>
#include <stdlib.h>
#include <signal.h>

#define dim(x) (sizeof(x)/sizeof(x[0])) /* macro to return the number of
                                           elements in a structure */

int break_handler();                    /* handler routine for Ctrl-C */

int cls_cmd(), dos_cmd(), exit_cmd();   /* declare intrinsic processors
                                           for use in command table. */

struct cmd_table {                      /* table of intrinsic commands */
           char *cmd_name;              /* command name entered by user */
           int  (*cmd_fxn)();           /* corresponding fxn to execute */
           }  commands[] =
              {  "CLS",   cls_cmd,
                 "DOS",   dos_cmd,
                 "EXIT",  exit_cmd,
              };

static char com_spec[64];               /* filespec of COMMAND.COM from
                                           Environment Block placed here */


main(argc, argv)
int   argc;
char  *argv[];

{   char inp_buf[80];                   /* operator's command placed here */

    get_comspec(com_spec);              /* get filespec for COMMAND.COM. */

                                        /* take over Break Int 23H
                                           so shell won't lose control. */

if (signal(SIGINT,break_handler) == (int(*)()) -1)
    {   fputs("Can't capture Ctrl-C Interrupt",stderr);
        exit(1);
    }
```

```
    while(1)                            /* main command interpreter loop */
        {   get_cmd(inp_buf);           /* get a command. */
            if (! intrinsic(inp_buf) )  /* if it's an intrinsic command,
                                           run its subroutine. */
                extrinsic(inp_buf);     /* else pass it to COMMAND.COM. */
        }
}


/*
        Try to match user's command with intrinsic command table.
        If a match is found, run the associated routine and return
        a True flag, else return a False flag.
*/

intrinsic(input_string)
char *input_string;
{   int i, j;
                                        /* scan off leading blanks. */
    while( *input_string == '\x20') input_string++ ;
                                        /* search the command table. */
    for (i=0; i < dim(commands); i++)
        {   j = strcmp( commands[i].cmd_name,input_string );
            if  (j == 0)                /* if match found, run routine */
                {   ( *commands[i].cmd_fxn )() ;
                    return(1);          /* and return a True flag. */
                }
        }
    return(0);                          /* no match, return False flag. */
}


/*
        Process an extrinsic command by passing it
        to an EXEC'd copy of COMMAND.COM.
*/

extrinsic(input_string)
char *input_string;
{   int status;
    status = system(input_string);      /* call EXEC function. */
    if (status)                         /* if failed, print error message. */
        fputs("\nEXEC of COMMAND.COM failed\n",stderr);
}

/*
        Issue prompt, get command line from the Standard Input Device
        as a null-terminated (ASCIIZ) string, fold it to uppercase.
*/

get_cmd(buffer)
char *buffer;
{   printf("\nsh: ");                  /* display prompt. */
    gets(buffer);                       /* get line from Standard Input. */
    strupr(buffer);                     /* fold it to uppercase. */
}
```

```
/*
        Get the full path and file specification for COMMAND.COM
        from the "COMSPEC=" variable in the Environment Block
*/

get_comspec(buffer)
char *buffer;
{   strcpy( buffer, getenv("COMSPEC") );
    if  ( buffer[0] == NULL )
        {   fputs("\nNo COMSPEC variable in Environment\n",stderr);
            exit(1);
        }
}


/*
        This handler for Int 23H signal keeps our shell from
        losing control when Ctrl-C is entered.  Just re-issues
        the prompt and returns.
*/

break_handler()
{   signal(SIGINT,break_handler);          /* reset handler address. */
    printf("\nsh: ");                      /* display prompt. */
}


/*
        These are the subroutines for the various intrinsic commands.
*/

cls_cmd()                 /* routine for intrinsic CLS command */
{   printf("\033[2J");  /* this is ANSI clear screen string. */
}


dos_cmd()                 /* routine for intrinsic DOS command */
{   int status;
    status = spawnlp(P_WAIT,com_spec,com_spec,NULL);
    if (status) fputs("\nEXEC of COMMAND.COM failed\n",stderr);
}

exit_cmd()                /* routine for intrinsic EXIT command */
{   exit(0);
}
```

SHELL 程序的基本流程可归纳如下:

(1) 调用 MS-DOS Int21H 功能 4AH（修改内存块）以压缩 SHELL 的内存分配，使得 COMMAND.COM 在作为一个覆盖模块运行时有尽可能大的空间可用（在汇编语言版本中这一点尤其明显）。

(2) 搜寻环境去查找变量 COMSPEC。该变量定义了 COMMAND.COM 可执行拷贝的地址。如果未能找到 COMSPEC 变量，SHELL 打印出错误信息并退出。

(3) SHELL 将其句柄的地址放入 Ctrl-C 向量 (Int23H)，使得用户打入 Ctrl-Break 时不致丢失控制。

(4) 对标准输出设备发出一个提示。

(5) 从标准输入设备读入缓冲的一行，以获得用户的命令。

(6) 命令行中第一个以空格分界的项（token）与 SHELL 内部命令表相匹配。如果找到了匹配项，便执行有关的过程。

(7) 如果未在内部命令表中找到匹配项，便可把用户输入加到 / C 开关上来合成命令行 tail，然后执行 COMMAND.COM 的拷贝，在 EXEC 参数块中传送合成命令 tail 的地址。

(8) *重复步骤 4 到 7，直到用户打入命令 EXIT，此条*内部命令使 SHELL 结束执行。

目前版本中，SHELL 使 COMMAND.COM 继承当前环境的完整拷贝。但在某些应用程序中传送经修改过的环境块拷贝可能更有帮助或更为安全。

1.5 MS-DOS 文件与记录操作

MS-DOS 同时考虑到与 UNIX／XENIX 及 CP／M 两个操作系统的兼容性。

与 CP／M 兼容的文件、记录功能组称为 FCB 功能。这些功能依赖于叫做文件控制块 FCB 的数据结构，以维护有关打开文件的信息。此结构驻留在应用程序的内存空间中。FCB 功能可使程序员能够生成、打开、关闭和删除文件，并在这种文件的任意记录处读写记录。这些功能不支持 MS-DOS 版本 2.0 导出的树状文件结构，这意味着它们只能用于访问给定磁盘驱动器当前子目录中的文件。

与 UNIX／XENIX 兼容的文件、记录功能组称为句柄功能。使用这些功能，可给 MS-DOS 传送一个空结尾字符串来打开或生成文件，该字符串给出了文件在树状文件结构中的位置、文件名、扩展名（叫做路径）。MS-DOS 返回一个 16 比特的句柄，同时由应用程序保存并在后面的操作中用来指定文件。

使用句柄功能时，数据结构由操作系统在其拥有的内存空间中维护，应用程序不可访问。句柄功能完全支持树状文件结构，允许程序员在任意磁盘驱动器中的任何子目录中生成、关闭、打开文件，并在这类文件中任意字节处读、写任意大小的记录。

1.5.1 使用 FCB 功能

了解文件控制块的结构是成功使用 FCB 族文件和记录功能的关键。FCB 是在应用程序内存空间中分配的一个 37 字节长的数据结构，并把其分成如图 1 — 9 的许多域。

如用另外的记录大小，必须在打开或生成功能后自己重填。

字节偏移量		
00H	驱动器标识	注 1,3
01H	文件名(8 个字符)	注 2
09H	扩展名(3 个字符)	注 2
0CH	当前块号	注 8
1EH	记录大小	注 9
10H	文件大小(4 字节)	注 3,6
14H	生成／更新时的日期	注 7
16H	生成／更新时的时间	
18H	保留	
20H	当前记录号	注 8
21H	随机记录号(4 字节)	注 5

注1:驱动器标识为二进制数:00＝默认驱动器,01＝驱动器A:, 02＝驱动器B:, 等等。如果应用程序给的驱动器代码为 0（省去，即默认），则当成功打开或生成调用后，在 MS-DOS 中所填写的代码为当前所用的磁盘驱动器。

注 2:文件和扩展名需左对齐，并用间隔隔开。

注3:标准型FCB中，字节0000H—000FH和0020H—0024H都要由用户设置。MS-DOS 使用 0010H—001FH 字节，而且在应用中不应更改。

注 4:所有字段都要用有效字节存在低位地址处。

注5:如记录大小少于64字节，则随机记录段作为4字节;否则只用本段中前3个字节。

注6:文件大小段以相同格式放在目录之中，用少量有效字放在低位地址处。

注7:数据段映象在目录之中，看成16位的字（如装入寄存器后本身存在时）。

注 8:按顺序读写时所用的当前块和当前记录数。

注9:记录大小段在打开(OFH)和生成(16H)功能时设置到128字节，以同CP／M兼容。

图 1-9　标准文件控制块，总长度为 37 字节(25H 字节)

程序初始化 FCB 时给出驱动器代码、文件名、扩展名(从而可由 Int21H 功能 29H 实现文件名语法检查服务功能)，然后将 FCB 的地址传送给 MS-DOS 以打开或生成文件。如果文件被成功地打开或生成， MS-DOS 就根据磁盘当前文件登记项填写 FCB 的某些域。这些信息包括以字节计的文件准确大小、文件生成或最后一次更新时的日期。在 FCB 的保留区中有时也存放某些其他信息，但此区域是由操作系统用于自身目的的，并随 MS-DOS 的不同版本而变。此保留区不能由应用程序修改。

```
mov  dx, seg my__fcb        ;文件名在"my-fcb"中已作过语法检查
mov  ds, dx                 ;DS:DX = 文件控制块地址
mov  dx,offset  my__fcb     ;
mov  ah,0fh                 ; 功能 0FH = 文件打开
int  21h
or  al,al                   ;成功打开否?
jnz  wrror                  ;否则跳转出错处理程序

my = fcb  db  37dup (0)
```

上面是 FCB 文件操作示例，此段代码试图打开一个文件，其名字以前已在名为my-fcb的 FCB 中作过语法检查。

为与 CP / M 兼容，MS-DOS 自动地将 FCB 的记录大小域设置为 128 字节。如果程序不打算使用此默认的记录大小，必须在记录大小域中置人以字节计的所要求的大小。以后当程序需从文件中读、写记录时，必须将 FCB 地址传送给 MS-DOS，MS-DOS 根据更新了信息后的 FCB 了解文件之大小及文件指针的当前位置。如果应用程序打算执行随机记录访问，必须于发出每个功能调用之前在文件控制块中设置记录号，由 MS-DOS 来维护 FCB，应用程序无需对其有特殊的干涉。

一般来说，使用文件控制块的 MS-DOS 在寄存器

DS:DX 中接受 FCB 的完整地址，并将返回代码传送回寄存器 AL。

对于文件管理调用（打开、关闭、生成、删除）如果功能调用成功则返回码为 0，如果功能失败，则返回码为 FFH．对于 FCB 类型的记录读、写功能，寄存器 AL 中返回成功代码仍是 0，但失败的代码却有几种。在 MS-D0S 版本 3.0 以上，可由在 FCB 功能调用失败后调用功能 59H（获得扩展错）来得到更详细的出错报告．

在 MS-DOS 下装入一个程序时，操作系统在程序段前缀偏移量 005CH 及 006CH 处建立两个文件控制块．通常称为有默认的 FCB，当时是用来提供与 CP／M 的兼容性的．启动该程序的命令行上前两个参数（不包括重定向符号）由 MS-DSO 经语法检查后送入默认的 FCB，前提假定为这两个参数是文件标识符．由应用程序负责确定它们是否真的是文件名．此外由于默认 FCB 是重叠的，而且尤其对于 EXE 程序不约定的地点，故为了使用时安全，应被拷贝到某处。

应注意，CP／M 的FCB结构和MS-DOS下的FCB结构是不同的.主要差别是在FCB的保留区，这种区域在任何情况下均不涉及应用程序，故“规矩”的 CP／M 程序员使用了某些提高性能的特殊技巧，就可能在 MS-DOS 下产生严重问题，尤其是在网络环境下．

FCB 的一个特殊变种称为“扩展控制块”，可用于生成或访问带有特殊属性（如隐式文件或只读文件）、卷标及子目录的文件。扩展 FCB 有一个 7 字节的标头，后面跟 37 字节的普通 FCB 结构。

注 1,2,3,4,5,6,7,8,9，之内容参见图 1-9．

字节偏移量		
00H	OFFH	注10
01H	保留(5字节,必须为0)	
06H	属性字节	注11
07H	驻动器标识符	注1,3
08H	文件名(8字符)	注2
10H	扩展名(3字符)	注2
13H	当前块呆	注8
15H	记录大小	注9
17H	文件大小(4字节)	注3,6
1BH	生成/更新日期	注7
1DH	生成/更新时间	
1FH	保留	
27H	当前记录号	注8
28H	随机记录号(4字节)	注5

注10:该结构的第一个字节0FFH(255)表示为扩充文件控制块．接受普通FCB的任何调用都可使用扩充FCB.

注11:扩充FCB的属性字节可以利用专门特征隐式、系统或只读方式来访向文件．扩充FCB也可用来读文档标题和专门子目录文件的内容.

图1-10 扩展文件控制块．总长度为44字节(2CH字节).

第一个字节为0FFH，通知MS-DOS使用的是扩展FCB。下面5个字节保留，当前版本的MS-DOS未使用。第七个字节为当前访问特殊文件类型的属性。使用普通FCB的MS-DOS功能也能使用扩展的FCB。

对于文件控制块和扩展文件控制块有以下几点需要说明:

(1) 驱动器标识符为二进制数:00＝默认驱动器;01＝驱动器 A;02＝驱动器 B 等等．如果应用程序提供的驱动代码是 0（默认驱动器），则在成功地打开或生成调用之后由 MS–DOS 填写实际的当前磁盘驱动器．

(2) 文件及扩展名必须左对齐并填充以空格．

(3) 普通 FCB 中，字节 0000H 到 000FH 及 0020H 至 0024H 由用户设置．字节 0010H 至 001FH 由 MS–DOS 使用且不可被应用程序修改．

(4) 所有字域在最低地址处储存最低位字节．

(5) 随机记录域作为 4 字节处理，条件是记录长度小于 64 字节，否则只使用此域的前 3 个字节．

(6) 文件大小域与目录的格式相同，最低地址处存放最低位字．

(7) 日期域作为 16 比特字看待（就象装入寄存器后所表现的那样)，该域分解为:

FE DC BA 9	8 7 6 5	4 3 2 1 0
年	月	日

其中年是相对于 1980 而言的．

(8) 当前块与当前记录号与顺序读、写联用，模仿 CP／M 的做法．

(9) 打开文件(OFH)及生成文件(16H)功能将记录大小域设置成 128 字节，以提供与 CP／M 的兼容性．如果打算使用另外的记录大小，必须在打开或生成文件功能之后重填．

(10) 扩展 FCB 中的属性字节允许访问带特殊特性（如隐式、系统或只读）的文件．扩展 FCB 还可用于读卷标及

特殊的子目录文件之内容。

如欲更好地了解 FCB 文件和记录管理调用，可将它们按如下归类：

功能	作用
普通的 FCB 文件操作	
OFH0	打开文件
10H	关闭文件
16H	生成文件
普通的 FCB 记录操作	
14H	顺序读
15H	顺序写
21H	随机读
22H	随机写
27H	随机块读
28H	随机块写
其他重要的 FCB 操作	
1AH	建立磁盘传送地址
29H	文件名语法检查
不太常用的 FCB 文件操作	
13H	删除文件
17H	文件更名
不太常用的 FCB 记录操作	
23H	获得文件大小
24H	建立随机记录号

其中有几项功能具有很特殊的性质。例如，功能 27H (随机块读)、28H (随机块写) 可用于读、写任意大小的多个记录，还可用于自动地更新随机记录域 (与功能 21H 及 22H 不同)。功能 28H 可用于将文件“裁”至任意要求的大小，使用扩展 FCB 的功能 17H 是改变卷标或给子目录换名的唯一方法。

▲FCB 文件访问概述

使用 FCB 访问文件的典型程序序列为:

(1) 清除将要用的文件控制块。

(2) 从用户、默认的文件控制块或程序段前缀中的命令 tail 中获得文件名。

(3) 如果没能从默认的文件控制块中获得文件名，便使用功能 29H 将文件名送入新的文件控制块。

(4) 打开文件 (功能 0FH) 操作生成 0 长度的文件或将同样名字的现存文件“裁”至 0 长度 (功能 16H)。

(5) 如不使用默认的记录大小，应在 FCB 中建立记录大小域. 在成功地打开或生成操作之后此项很重要.

(6) 如果执行随机记录 I/O，在 FCB 中建立记录号域。

(7) 使用功能 1AH 建立磁盘传送区，除非自上次调用此功能后缓冲地址未发生变化。如果此应用程序未执行过建立磁盘传送区操作，则磁盘传送区的默认地址位于 PSP 中偏移量 0080H 处。

(8) 请求读、写记录的操作(功能 14H——顺序读，15H——顺序写，21H——随机读，22H——随机写，27H——随机块读，28H——随机块写)。

(9) 如果未完成，转去步骤(6)，否则关闭文件 (功能

10H)．如果文件只用于作读操作，早期版本的 MS-DOS 可能不执行此关闭文件操作．但这一缺点在 MS-DOS 版本 3.0 以上便会产生问题，尤其是在通过网络访问文件时．

```
recsize   equ   1024              ;file record size
          :
          mov   ah,29h            ;parse input filename.
          mov   al,1              ;skip leading blanks.
          mov   si,offset fnamel  ;address of filename
          mov   di,offset fcbl    ;address of FCB
          int   21h
          or    al,al             ;jump if name
          jnz   name__err         ;was bad.
          :
          mov   ah,29h            ;parse output filename.
          mov   al,1              ;skip leading blanks.
          mov   si,offset fname2  ;address of filename
          mov   di,offset fcb2    ;address of FCB
          int   21h
          or    al.al             ;jump if name
          jnz   name__err         ;was bad.
          :
          mov   ah,ofh            ;open input file.
          mov   dx,offset fcbl
          int   21h
          or    al,al             ;open successful?
          jnz   no__file          ;no, jump.

          ⋮
          mov   ah,16h            ;create and open
          mov   dx,offset fcb2    ;output file.
          int   21h
          or    al,al             ;create successful?
          jnz   disk__full        ;no jump.
          ⋮
                                  ;set record sizes.
          mov   word ptr fcb1+0eh, recsize
          mov   word ptr fcb2+0ch, recsize
```

```
        mov   ah, 1ah              ;set disk transfer
        mov   dx,offset buffer     ;address for reads
        int   21h                  ;and weites.
next:   :                          ;process next record.
        mov   ah,14h               ;sequential read from
        mov   dx,offset fcb1       ;input file
        int   21h
        cmp   al,01                ;check for end file.
        je    file_end             ;jumo if end of file.
        cmp   al,03
        je    filo_end             ;jump if end of file
        or    al,al                ;other read fault?
        jnz   bad_read             ;jump if bad read.
        :
        mov   ah,15h               ;sequential write to
        mov   dx,offset fcb2       ;output file
        int   21h
        or    al,al                ;write successful?
        juz   bad_write            ;jump if write failed.
        :
        jmp   next                 ;process next record.
file_end: :                        ;reached end of input
        mov   ah, 10h              ;close input file.
        mov   dx,offset fcb1
        int   21h
              :
        mov   ah,10h               ;close output file.
        mov   dx,offset fcb2
        int   21h
        mov   ac,4c00h             ;exit with return
        int   21h                  ;code of zerc.
        :
fname1  db    'OLDFILE.DAT',0      ;name of input file
fnemc2  db    'NEWFILE.DAT',0      ;name of output file
fcb1    db    37 dup(0)            ;FCB for input file
fcb2    db    37 dup(0)            ;FCB for output file
buffer  db    recsize  dup(?)      ;buffer for file I/O
```

上面是使用FCB族功能调用执行文件和记录I/O的汇编语言程序例。

▲FCB 功能操作的优点

· 在 MS-DOS 版本 1,2 下,使用 FCB 可并发打开的文件数不限（在版本 3 下不是这样,尤其是在网络软件运行时）.

· 使用 FCB 的文件访问方法非常适于具有 CP / M 基础的程序员,规矩的 CP / M 应用程序欲在 MS-DOS下运行，无需大的改动.

· 文件的大小与日期在打开后由文件控制块提供，并可由调用程序来检查.

▲FCB 功能操作的缺点

· 文件控制块占用用户内存空间.

· FCB 不支持树状文件结构(不可访问当前目录以外的文件).

· FCB 不能提供网络环境下的文件锁定 / 共享或记录锁定功能.

· 使用 FCB 的文件读、写需要操作文件控制块以设置记录大小和记录号，除读、写调用本身以外，还要加上单独的 MS-DOS 功能调用去建立 DTA 地址.

· 对于包含可变长度记录的文件，使用 FCB 作随机记录 I / O 非常不方便.

· 用于访问或生成带特殊属性（如隐式、只读或系统）之文件的扩展文件控制块与 CP / M 不兼容.

· FCB 文件功能出错报告能力很差．在 MS-DOS 版本 3 中部分地改进了这种情况，因为在 FCB 功能失败后可

	打开之前的 FCB	FCB 内容	打开之后的 FCB
00H	00	驱动器	03
01H	4D	文件名	4D
02H	59		59
03H	46		46
04H	49		49
05H	4C		4C
06H	45		45
07H	20		20
08H	20		20
09H	44	扩展名	44
0AH	41		41
0BH	54		54
0CH	00	当前块	00
0DH	00		00
0EH	00	记录大小	80
0FH	00		00
10H	00	文件大小	80
11H	00		3D
12H	00		00
13H	00		00
14H	00	文件日期	43
15H	00		0B
16H	00	文件时间	A1
17H	00		52
18H	00	保留	03
19H	00		02
1AH	00		42
1BH	00		73
1CH	00		00
1DH	00		01
1EH	00		35
1FH	00		0F
20H	00	当前记录	00
21H	00	随机记录号	00
22H	00		00
23H	00		00
24H	00		00

图 1-11　在成功地打开调用(Int21H 功能 0 FH)之前与之后的典型文件控制块

调用附加的功能 59H（获得扩展错）.

· Microsoft 不推荐使用类 CP／M 的调用.

1.5.2 使用句柄文件和记录功能

用于访问文件的句柄文件和记录管理功能，在方式上类似于 UNIX／XENIX 操作系统中所用的方式. 文件由一个 ASCIIZ 字符串（以空或 0 字节结尾的 ASCII 字符串）表

示，可包含驱动器标识符、路径、文件名和扩展名。例如，文件标识符 C:\SYSTEM\COMMAND.COM 以如下字节序列：

433A5C595354454D5C434F4D4D414E442E434F4D00

一个程序如欲打开或生成一个文件，标识此文件的 ASCIIZ 字符串地址在寄存器 DS:DX 中传送给 MS-DOS。如果操作成功，MS-DOS 在 AX 中将一个 16 比特的句柄返回给程序。此句柄保留，以便将来引用。

以后有该文件的操作请求时，在调用 MS-DOS 之前常将此句柄置于寄存器 BX 中。如果操作成功，则进程标志被清 0（无论何种句柄功能）。如果操作失败，则进位标志置位，此时寄存器 AX 包含出错代码。

可同时激活的句柄数（即可使用句柄族功能调用并发打开的文件及设备数）受以下两个因素制约：

・对于所有活动进程总计应为系统中可并发打开文件的最大数，该数字要在 CONFIG.SYS 文件中由 FILES＝nn指定，这样也就确定了在"系统文件打开表"中所分配的项数。在 MS-DOS3.0 版本下，默认值为 8，最大值为 255。一旦 MS-DOS 启动并运行，就没有办法再扩充此表以增加可打开文件的总数，否则必须使用编辑程序修改 CONFIG.SYS 文件，然后重新启动系统。

・单个进程可并发打开文件的最大数是 20，前提是系统文件打开表中尚有充足的项数可用。程序装人时，这 20 个句柄中的 5 个预先分配给了标准设备。每一次进程发出打开或生成功能调用，便分配给其一个句柄，直至用完所有句柄或系统文件打开表满。

```
          mov   ah,3dh              ;function 3DH = open
          mov   al,2                ;mode 2 = read / write
          mov   dx,seg filename     ;address of ASCIIZ
          mov   ds,dx               ;file specification
          mov   dx,offset filename
          int   21h                 ;request open from DOS.
          jc    error               ;jump if open failed.
          mov   handle,ax           ;save file handle.
          :
filename  db    'c:\MYDIR\MYFILE.DAT',0
handle    dw    0
```

上面是典型的句柄文件操作例，欲打开文件由一个ASCIIZ字符串表示，其地址在寄存器DS:DX中传送给MS－DOS。

句柄类文件和记录管理调用可大致分类如下：

功能	作用
常用句柄文件操作	
3CH	生成文件(需 ASCIIZ 字符串)
3DH	打开文件(需 ASCIIZ 字符串)
3EH	关闭文件
常用句柄记录操作	
42H	设置文件指针(也用于确定文件大小)
3FH	读记录
40H	写记录
不大常用的句柄文件操作	
41H	删除文件
43H	获得或修改文件属性

56H	文件换名
57H	获得或设置日期和时间
5AH	生成临时文件(仅用于版本 3)
5BH	生成文件(如果已存在则失败,仅用于版本 3)

▲句柄文件访问的一般步骤

使用句柄（或类 UNIX / XENIX）族功能调用访问文件的典型程序顺序如下:

(1) 通过带缓冲的输入服务功能（Int21H 功能 OAH）或通过 MS-DOS 在程序段前缀中提供的命令 tail 获得用户给出的文件名。

(2) 在文件标识符后加一个 0,以形成一个 ASCIIZ 字符串。

(3) 使用 Int21H 功能 3DH 及方式 2（读 / 写访问）打开文件，或使用功能 3CH 生成文件（确保将寄存器 CX 设置为 0，从而避免不小心所生成的一个带有特殊属性的文件）。保存返回的句柄。

(4) 使用 Int21H 功能 42H 建立文件指针。可相对三个位置之一来建立文件指针的位置:文件头，当前指针位置及文件尾。如果执行的是顺序记录 I / O，通常可跳过这一步，因为 MS-DOS 将自动地维护文件指针。

(5) 读文件(功能 3FH) 或写文件(功能 40H)。这两个功能要求 BX 包含文件的句柄，CX 包含记录长度，DS:DX 指向被传送数据的内存地址。实际传送的字节数在 AX 中返回。

作读操作时，如果读入的字节数少于请求的字节数，说明已达到文件尾。当作写操作时，如果所写的字节数少于请

求的字节数，说明包含该文件的磁盘满。这两种情况并不作为错误代码返回，不设置进位标志。

(6) 如果未结束，转第(4)步，否则，关闭文件（功能3EH）。程序的正常退出（除功能31H——结束并驻留）均关闭所有活动的句柄。

```
recsize     equ     1024                    ;file record size
            :
            mov     ah,3dh                  ;open input file.
            mov     al,0                    ;mode = read only
            mov     dx.offset fname1        ;name of input file
            int     21h
            jc      no__file                ;jump if no file.
            mov     handlel.ax              ;save token for file.
            :
            mov     ah,3ch                  ;create output file.
            mov     cx,0                    ;attribute = normal
            mov     dx.offset  fname2       ;name of output file
            int     21h
            jc      disk__full              ;jump if create fails.
            mov     handle2,ax              ;save token for file.
next:                                       ;process next record.
            :
            mov     ah,3fh                  ;sequential read from
            mov     bx,handle1              ;input file
            mov     cx,recsize
            mov     dx,offset buffer
            int     21h
            jc      bad__read               ;jump if read  error.
            or      ax,ax                   ;check bytes transferred.
            jz      file__end               ;jump if end of file.
            :
            mov     ah,40h                  ;s quential write to
            mov     bx,handle2              ;o tput file
            mov     cx,recsize
            mov     dx,offset buffer
            int     21h
            jc      bed__write              ;j mp if write error.
```

```
        cmp   ax,recsize             ;whole record written?
        jne   disk__full             ;jump if disk is full.
        :
        jmp   next                   ;process next record.
file__end:  :                        ;reached end of input
        mov   ah,3eh                 ;close input file.
        mov   bx,handle1
        int   21h
        :
        mov   ah,3eh                 ;close output file.
        mov   bx,hadle2
        int   21h
        :
        mov   ax, 4c00h              ;exit with return
        int   21h                    ;code of zero.
        :
fname1    db   'OLDFILE.DAT',0       ;name of input file
fname2    db   'NEWFILE.DAT',0       ;name of output file
handle1   dw   0                     ;token for input file
handle2   dw   0                     ;token for output file
buffer    db   recsize dup(?)        ;buffer for file I/O
```

上面的汇编语言程序例对一个输入文件执行顺序操作，并将结果写到一个输出文件，使用句柄文件和记录功能。本程序假定DS和ES已设置成指向包含缓冲区和文件名的段。

▲优点

・句柄调用对于标准输入、输出设备提出I/O重定向和管道的直接支持，其方式在功能上与UNIX/XENIX所用的类似。

・句柄功能为子目录(树状文件结构)及特殊文件属性提供直接支持。

・句柄调用支持网络环境下的文件共享/锁定和记录锁定。

・使用句柄功能，程序员可打开字符设备通道并将它们

作为文件处理。

・句柄调用使得随机记录访问非常方便。

当前文件指针可移至相对于文件头、文件尾或当前指针位置任意字节偏移量处。任意长度的记录（直至整段 65535 字节）可用一次操作读入内存中任意地址。

・句柄功能在 MS-DOS 下有较好的出错报告能力，在版本 3 下又得到进一步加强。

・Microsoft 公司大力推荐使用句柄族功能调用，以便与将来的 MS-DOS 环境提供向上兼容性。

▲缺点

・可并发打开的文件数受限制（这些限制在 MS-DOS 版本 3 下使用 FCB 打开文件时也存在）。

・句柄功能调用的实现尚不太完善。例如还需使用扩展 FCB 去访问卷标和实现子目录的特殊文件内容。

1.5.3 MS-DOS 出错代码

句柄功能失败，则进位标志置"1";或在 FCB 功能失败后调用功能 59H（获得扩展错），可能返回如下的出错代码。

代码	意义
版本 2 的 文件功能错误	
01	非法功能号
02	文件未找到
03	路径未找到
04	打开文件过多（未剩下句柄）
05	拒绝访问
06	非法句柄

07	内存控制块破坏
08	内存不够用
09	非法内存块地址
10	非常环境
11	非法格式
12	非法访问代码
13	非法数据
14	保留
15	非法磁盘驱动器
16	试图删除当前目录
17	非同一设备
18	无更多文件

映照为致命错误处理程序

19	磁盘写保护
20	无此磁盘设备
21	驱动器未准备好
22	无此命令
23	数据出错(CRC)
24	请求结构长度坏
25	寻道错
26	无此介质类型
27	扇区未找到
28	打印机无纸
29	写失败
30	读失败
31	总体失败

版本 3 附加的出错代码

32	共享失败
33	加锁失败
34	非法磁盘改变
35	不可使用 FCB
36	共享缓冲区溢出
37–49	保留
50	不支持网络请求
51	远程计算机不在线
52	网络上有重名
53	网络名未找到
54	网络忙
55	网络设备不再存在
56	已超过网络 BIOS 命令限制
57	网络适配器硬件出错
58	网络响应不正确
59	不希望有的网络错误
60	远程适配器不兼容
61	打印队列满
62	打印列队不满
63	打印文件已删除(空间不够)
64	已删除网络名
65	拒绝访问
66	网络设备类型不正确
67	未找到网络名
68	已超过网络名限制
69	已超过网络 BIOS 对话限制

70	暂停
71	不接受网络请求
72	打印或磁盘重定向暂停
73—79	保留
80	文件已存在
81	保留
82	不能生成目录项
83	Int24H 失败
84	重定向过多
85	重定向重复
86	非法通行保密字
87	非法参数
88	网络设备失败

在MS-DOS 版本 3 下，功能59H可用于获得有关出错的其他信息，如出错地点及推荐的恢复动作。

第二章　中断和 DOS 功能

当编程工具选定后，下一步工作就是开始利用这些工具开发利用计算机系统的资源。一种高级语言，如 C 语言，与它的各种函数库一起，是开发可靠而简捷的计算机程序的重要工具。然而，高级语言是为一般的、抽象的开发环境而设计的，并不依赖于机器的内部细节。因而，它不可能完全利用任一特定计算机的功能。所以，生成一个高性能的程序，经常需要超越高级语言和它的功能库的边界而直接访问系统的基础资源。

例如，建立视频输出有许多种方法，可以经常使用 printf，接受由编译器选择的基础机构。但这种选择对一种特殊的应用系统和一种特殊的计算机系统来说，不一定是最好的。然而，如果熟悉了系统中可以使用的软件和硬件资源，就可以采用一种更成熟的决定，并选择一种更为有效的方案。

在 MS-DOS 机器中，一种最容易的可被访问的重要资源是由操作系统和 BIOS 所提供的基本程序（基本输入/输出系统，包含在硬件提供的只读存储器中）。所有这些功能都有一个共同的特点，它们都是通过中断机构而被访问的。本章从对中断机构的总体解释开始，然后叙述用汇编语言和 C 程序访问这些中断的方法。最后一节是对 MS-DOS 功能的概括说明——一个程序员指南，解释这些功能是怎样组织的，讨论选择最佳程序的策略，并给出了一些有趣的例子。

第五章将继续讨论并说明 BIOS 服务程序.

2.1 中断机构

中断就是使处理器暂时中止当前的工作并分支指向内存中某处的服务程序. 这些服务程序在完成了该做的工作后, 就返回到被中止的过程. MS－DOS 机器上的中断, 按它们产生的方法可分为三类:内部硬件中断、外部硬件中断和软中断.

▲内部硬中断　当一条指令执行完以后, 如果下列条件中的一条存在时, 8086／88 处理器立即产生一个内部中断:

· 上一条指令是一个除法 (div 或 Idiv), 并且所得的商大于一个确定的终点 (也叫做除 0 错).

· 陷井标志 (TF) 被置位. 这一标志在 debugger 的单步跟踪方式下被置位, 在每一条指令后产生一个中断 (除非修改了段寄存器).

· 上一条指令是 into (中断溢出), 并且溢出标志 (OF) 被置位.

在这些情况下, 中断的结果使处理器分支而指向一个适当的程序来处理出错或特殊的状态. 80286 处理器有许多附加的内部中断.

▲外部硬中断　处理器外面的硬件设备, 可以由处理器的 INTR 或 NMI 脚向一个信号来触发一个中断. 例如, 在键盘上按下一个键时产生的硬中断, 使程序分支指向 BIOS 中读字符并将它放入缓存区的服务程序. 这种类型的中断将在第六章中加以讨论.

▲软中断　机器指令 int 也产生一个中断. 这种类型的中断, 利用操作系统和用户程序去逼近系统的效能. 它是这一章的主要课题.

在接受一个特殊中断时，处理器是怎样知道程序分支转向何处呢?无论三种机构的哪一种（内部、外部或软）所触发的中断，都必须提供一个字节的数字，叫做中断类型．这一数字不是服务程序的物理地址（这是一非常硬性的机构），而是将一张包含实际程序地址的索引表保持在低区．这些地址被称为中断向量．因为它们指向中断服务程序．

要了解组成中断的准确次序,就必须考虑下列中断指令．

Int 21h

数字 21h 是中断类型.它告诉处理器: 目标程序的地址放置在中断向量表入口的 21h 处．中断向量表中的每一入口包含以偏移量形式给出的服务程序的一个双字（4 字节）地址段内容．中断向量表中的一个单个的人口有如图 2–1 所示的结构．表以 0 偏移量开始，因此一个入口的实际内存中的偏移量可用中断类型乘以 4 得到．中断 21h 的偏移量为 21h×4 等于 84h．存储在这一位置的地址可用 DEBUG 发出下列命令时看到:

–D 0:84L4

如果响应后得到

0000:0080 08E4 8023

服务程序的地址以段:偏移量的形式表示，即是 238D:E408、在接受 Int21h 指令后，处理器将:

- 标志寄存器压入堆栈．
- 用清中断标志 (IF) 和陷井标志 (TF),禁止硬件中断．
- 当前段寄存器码压入堆栈．
- 当前指令指针压入堆栈．
- 转向内存地址 238D E40B。

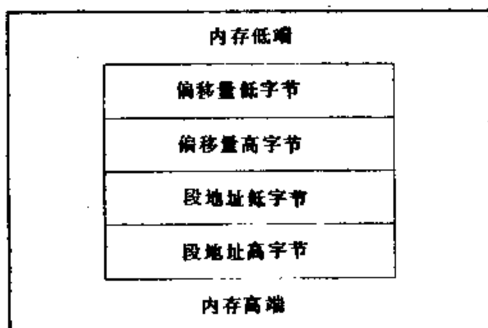


图 2–1　中断向量表的入口

然后服务程序通过 Iret 指令，从中断程序中返回，使处理器将:

- 指令指针弹出堆栈。
- 段寄存器码值弹出堆栈。
- 标志弹出堆栈。

中断类型是一单字节，它只能定义 256 个可能的入口。所以表的整个尺寸是 1024 字节。一些表的入口在系统启动时被 BIOS 和 MS－DOS 初始化。另一些入口可在其后由用户请求装入的内存映象程序和其他中断处理器初始化。一些中断类型为内部硬中断保留，一些为外部硬中断所保留，其他为软中断所保留。那些为软中断所使用的服务程序存在于硬件所加的属于 BIOS 或属于适配卡的只读存储器 (ROM)、MS－DOS 的核心、由用户安装的设备驱动程序、内存映象程序和其他中断服务程序。

中断指令是一种灵活的、用途广泛的机构，可以引入一

切可能获得的服务。当申请一个中断服务时，所有程序，包括操作系统本身，必须使用一个未知位置上的地址。所以，一个给定中断的全部属性可以用替换指向其他程序的地址的方式很容易地修改掉。例如，实际上通过中断类型 16h 读键盘的所有程序，就是一般地指向 BIOS 中的一个服务程序。所以 在简单的情况下，要一边有效地监视键盘的一切活动，一边通过另一过程重新转回到这一向量，使系统响应键盘的读请求。

2.2 从汇编语言中访问软中断程序

用汇编语言申请一个标准 DOS 或 BIOS 中断程序是非常简单的。通过寄存器（偶尔也通过内存控制块）进行调用程序与中断服务程序之间的标准的信息交换。对于每一种 DOS 和 BIOS 功能，在中断指令之前要先装入明确的状态值，并通过寄存器返回一些值。作为一个例子，研究一下 DOS 为获得一个给定磁盘设备所有空闲空间的功能。这一功能可通过中断 21h 实现，并且给定下列寄存器的使用：

进入

AH	36h	(DOS 功能号)
DL	驱动器说明	(0=缺省，1=A)

返回

AX	扇区/每束 或FFFFh如果驱动器说明不合法
BX	可得到的束
CX	字节/每扇区
DX	束数/驱动器

下面的汇编程序码说明了怎样使用这一功能获取驱动器 A 上的空闲空间：

```
mov   ah, 36h        ; DOS功能号
mov   dl, 1          ; 驱动器A上的空闲空间
int   21h            ; 申请中断
cmp ax, offfh        ; 错误检查
je err_rtn           ; 指定驱动器不合法
mov   sec-cl-,ax     ; 存入每束扇区数
mov   avl-clv,bx     ; 存入可得到的束数
mov   byt-sec,cx     ; 存入每扇区字节数
```

该例说明了使用汇编程序申请一个中断程序时所需的标准步骤:

1.在寄存器中装人指定值;

2.申请中断(intn);

3.检查返回码 (cmp ax, offffh),如果指出有错则跳转到出错处理程序 (je err_rtn)。其他情况下存入返回值并继续执行程序.

2.3 从C语言中访问软中断程序

C 的函数库提供了一些引入软中断的函数: bdos, int86, lnt86x, lntdos, lntdosx. 功能 bdos, lntdos 和 lntdosx 是以中断 21h 为限 (MS-DOS 功能调用入口点原理), 而且使用它们优点并不多. 然而 lnt86 和 lnt86x 可以引人任何中断. 如果必须通过 ES 或 DS 寄存器传递或返回值, 则必须使用 int86x, 其他情况下才有可能使用 lnt86. 在所有这几种选择中, lnt86 是最通用的, 它可在任何情况下使用.

图 2-2 中的 C 程序, 说明使用 lnt86 引入中断 21h 的 36h 号功能 (与前面汇编程序例子中所使用的 DOS 功能相同). 调用 DSO 后, C 程序中的 free-space 功能利用返回

值计算所有空闲空间的字节数．即用可获得的束数（在 BX 中）乘上每束的扇区数（在 AX 中），再乘上每扇区的字节数（在 CX 中），然后 free－space 功能将每个值赋给一个长整型字以避免溢出，并返回乘积．

lnt86 功能提供了三个自变量：

・中断号（一个整型数）；

・保存功能引人时装人寄存器的值的地址域（union REGS 型）；

・接收从功能返回寄存器的值的地址域（union REGS 型）。由于一般不需要将中断功能用过的值保存起来，所以通常只用一个 union REGS 变量就可以了．例如：

```
union REGS reg;
lnt86 (ox21 & reg,&veg)
```

所需的集合类型（REGS）由文件的 DOS.H.REGS 定义，包括通过.h 与集合单元（AH，AL，BH，BL，CH，CL，DH 和 DL）引出的一个 8 位寄存器和通过.x 与单元（AX，BX，CX，DX，SI，DI 和标志寄存器）引出的一个 16 位寄存器．

有些中断服务需要通过寄存器交换数据，但并不包括在 union REGS 中．对于这些中断，必须使用 int86x．这一功能所需的前3个参数与 int86中用过的一样．但第 4 个参数是一附加参数，即指出 strnct SREGS 的一系列类型．struct SREGS 也是用 DOS.H 定义的，虽然它包括 ES，CS，SS 和 DS 段寄存器的整型字段．int86x 只使用 ES 和 DS．在即将申请中断时，int86x 从变量 SREGS 复制适当的值到 ES 和 DS 寄存器；从中断返回时，再把 ES 和 DS 寄存器的值复制到变量 SREGS 中，并恢复 DOS 的原始内

容.

MS—DOS 的35h号功能,返回当前指定的中断向量的值,是一个使用段寄存器的服务的一个例子.手册给出了下列约定:

引入

AH　35h (DOS功能号)

AL　中断类型

返回

ES　中断向量段地址

BX　中断向量偏移量地址

图 2—3 的 C 程序说明了功能 int86x 的使用.功能 getvcc 指定第1个参数取当前中断向量的值,第2个自变量给出结构中可获得的地址.

不论是 lnt86 和 lnt86x,均用从中断返回的带有标志的值装入 REGS 中的 cflag.许多 DOS 中断功能都用置标志的方法标定一个错误,并通过 AX 寄存器返回错误码.如果标志被设置,错误码可以下面三种方式之一给出:

1. C功能int86和lnt86x均用AX返回值,立即给出错误码.

2. 可以通过 outreg.s.ax 引出 AX.

3. DOS 错误码分配给－doserror 的所有变量,并在功能返回时立即检查.

例如,一个错误条件可以按下面方式进行检查:

```
lnt86x(0x21,& lnregs, &outregs, &segregs)
        if (outregs.x.cflag)      / * 检查错误的出现 * /
        switch C—doserror)    / * 按错误码进行跳转 * /
```

```
long free_space (int);

void main ()
   {
   printf ("The free space on drive A is: %ld bytes.\n", free_space (1));
   }

#include <dos.h>

long free_space (drive)
int drive;
   {
   union REGS inreg, outreg; /* Variables for passing register values. */

   inreg.h.ah = (unsigned char)0x36;         /* DOS function number. */
   inreg.h.dl = (unsigned char)drive;          /* Disk drive. */
   int86 (0x21, &inreg, &outreg);             /* Call DOS. */
         /* Calculate number of free bytes, and return results. */
   return (long)outreg.x.ax * (long)outreg.x.bx * (long)outreg.x.cx;
   }
```

图 2-2　访问中断服务功能的 C 程序

2.4 选择中断功能

前面一节的例子，说明了从汇编语言和 C 语言访问 MS—DOS 机器中可以使用的中断功能是很简单的。除了需要使用寄存器外，这种中断功能的引入并不比调用库程序难多少。然而，在现有的可使用的功能迷宫中选出适当的程序，却可能是困难的。由于这一节的内容会在很大程度上影响软件的性能，所以本章与第五章一样，也把重点集中在选择最恰当的程序的策略上。

为什么选择最恰当的中断功能会成为一件难事呢？原因之一是：MS－DOS机器中现有的一系列功能并不是依照主计划设计的，而是开发成一个层次分明、并经常排列着多余的接力程序的有机体。许多在 BIOS 水平上可使用的功能与 MS－DOS 提供的功能很相似，有许多集中在MS－DOS的服务程序中，它们完成基本的双重功能。然而，这些双重功能在特点、性能和兼容性方面有着细微的差别。

难于选择最合适的程序的另一个原因是文档。首先，最

基本的参考资料分散在两本手册中——BIOS功能的PC或AT的技术参考手册，及MS-DOS功能的DOS技术参考手册．同样，BIOS的文档淹没在一大长串汇编语言源程序中．DOS功能只提供简单的数字化命令，比它们实际所能做的少得多（缺乏逻辑的数学化命令）．

下面一节的主要内容与第三章一样，是帮助读者选择最合适的中断程序．所以，书中的这部分内容是根据程序的服务功能，强调它们之间的区别，并讨论作最佳选择的策略来组织的．接下去的一节，也是用程序列表来说明有关的中断功能的实际应用，并从中获得更多的启发．

```
struct vector        /* Structure for holding interrupt vector address. */
   {
   unsigned int segment;
   unsigned int offset;
   };

void getvec (int, struct vector *);

void main ()
   {
   struct vector v;        /* Declare variable for containing results. */

   getvec (0x21, &v);           /* Get contents of interrupt type 21h. */
                                /* Print results stored in structure. */
   printf ("Segment:Offset of int 21h is %x:%x\n", v.segment, v.offset);
   }

#include <dos.h>                 /* Contains structure definitions. */

void getvec (type, vecptr)
int type;
struct vector *vecptr;
   {
   union REGS regs;                          /* Defined in DOS.H */
   struct SREGS segregs;

   regs.h.ah = 0x35;                        /* DOS function number. */
   regs.h.al = (unsigned char)type;            /* Interrupt type. */
   int86x (0x21, &regs, &regs, &segregs);     /* Call DOS service. */
   vecptr->offset = regs.x.bx;                  /* Store results. */
   vecptr->segment = segregs.es;
   }
```

图 2-3 使用int86x的C程序

2.5 MS-DOS功能

DOS中断实用程序虽然对高性能软件并非很根本，但

却提供了一组综合性的、可靠且简捷的功能，这对于磁盘文件操作尤其有用。DOS 功能所采用的方法和起点对于 IBM 兼容计算机已很陈旧了，但 C 编译还是一有可能便使用 DOS 功能。

什么情况下采用 DOS 服务功能，什么情况下选择更低层的方法呢?最好的使用 DOS 领域之一的是磁盘和文件管理。与其他好的操作系统一样， DOS 将本质上很难对付的硬件资源——磁盘驱动器处理后,生成为容易管理的逻辑资源—— 文件系统。另外,DOS 的视频显示程序也是很成熟的资源。

按 IBM 的说法，DOS 服务功能分成"功能调用和中断调用"。这两类服务功能中有些是重复的，也有不少差别。但一般来说，IBM 建议只要有可能便使用功能调用。

DOS 服务功能清单是很混杂的，因为新的操作系统版本会引入新的功能，又要维持老的功能以保持向上兼容性。因此常常有两个以上功能上类似的服务功能。决定使用其中哪一个，必须考虑到兼容性与最大能力两个方面。

DOS 技术参考手册将涉及到 DOS1.X 的功能（范围从 0 到 2Eh，除 1ch 以外）称为是"传统的"。此类中的文件管理功能使用的结构称为文件控制台。由DOS2.X引出的功能通常称为"是扩展的"(范围从 2Fh 到 57h，加上 1Ch)，扩展文件服务功能使用文件句柄。版本 3.X 还带来了一组新的功能（范围从 58h 到 62h）。

2.5.1 MS-DOS 功能的一般用法

在讲述单个 MS-DOS 功能之前，本节先介绍错误处理、寄存器保存以及建立合适堆栈的一般准则。

▲错误处理

MS-DOS 功能的出错处理相当混乱，因为不同版本使用不同的方法。MS—DOS 在报告出错条件时使用四个基本方法。

第一种方法由 DOS1.X 传统功能（范围从 0 到 2Eh）使用，尤其是文件管理部分，使用 AL 寄存器作错误报告。如果 AL 等于 0，则无错误发生。如果 AL 不等于 0，便含有出错代码。这些代码没有标准的意义，对每个功能都单独地给出代码意义。多数功能使用值 FFh 报告一个出错条件，其他功能返回 01h，02h,03h等等指出少量可能的错误之一。

出错处理的第二种方法是由 DOS 版本 2.X 和 3.X 所引出的一组功能所用。这些功能由设置进位标志并在 AX 寄存器中放置出错代码来报告出错。此组功能为:38h 到 4Bh，4Eh 和 4Fh，56h 和 57h，5Ah 到 5ch，5Eh 和 5Fh。

AX 中返回的值称为扩展出错代码。与传统功能所报告的代码不同，这些出错代码是标准的，并在技术参考手册中的一张图中列出。出错代码共有88个，但 DOS 版本 2.X 功能仅使用前 18 个。这 18 个代码表示多数普通的错误并列在表 2.1 中。图 2-4 中所列宏 printn（以十进制打印出寄存器内容）在显示这些出错信息时有用。

表 2.1　DOS版本 2.X的出错代码

代码	意义	使用此代码的功能
1	非法功能	所有
2	文件未收到	38,3D,41,43,4B,4E,56
3	路径未找到	39,3A,3B,3C,3D,41,43,44,4B,4E,56

4	无更多句柄	3C,3D,45,46
5	拒绝访问	39,3A,3C,3D,3F,40,41,43,44,4B,56
6	非法句柄	3E,3F,40,42,44,45,46,57
7	内存控制块被破坏	48,49,4A
8	内存不够用	48,4A,4B
9	非法内存块	49,4A
10	非法环境	4B
11	非法格式	4B
12	非法访问代码	3D
13	非法数据	
14	保留	
15	非法驱动器标识符	3A
16	试图删除当前目录	3A
17	无相同设备	56
18	无更多文件	4E,4F

DOS 使用的第三种出错处理方法是由调用中断 24h 并将一个出错代码传送给该中断处理程序来响应所谓的"致命错误".

DOS3.0 引出了处理出错条件的第四种机构:功能 59h,由它来返回扩展的出错信息.

DOS 技术参考手册建议,在每个DOS21h功能之后和在中断 24h 致命错误处理程序开始处调用功能 59h. 甚至在传统的 FCB 调用之后也可调用此功能, 此时所报告的在技术参考手册中描述的是最接近于特定功能的扩展错误号.

注意, 除了这四种出错处理的基本方法以外, 由中断 25h,26h,2Fh 所用的特殊出错代码在技术参考手册中描述.

▲寄存器保存

作为一般性准则,在中断功能调用过程中，除寄存器AX以外应保存所有寄存器。此准则有以下几个例外:

· 中断25h，26h破坏除段寄存器以外的所有寄存器。

· 中断59h破坏寄存器AX,BX,CX,DX,SI,DI,ES和DS。

· 中断4Bh破坏所有寄存器,包括SP和SS。

SP和SS必须在码段内存储（可由CS寄存器来寻址），从而可在功能返回时恢复。

▲用户的堆栈需求

DOS中断服务功能使用自身的内部堆栈，但技术参考手册建议用户程序应具有至少200字节的自由堆栈空间以适应中断机构。

2.5.2 DOS服务功能分类

2.5.2.1 传统的字符设备输入／输出

传统的字符设备I／O现在看来有些过时了，Microsoft公司也不鼓励其使用，但在编写致命错误处理程序或设备驱动程序时，欲执行I／O只允许使用这些功能。

从键盘读入或向控制台写的本组功能均导向标准输入和输出，可由用户重定向。这些功能有些看起来有重复，实际上均有细微的差别，例如是否检查ctrl－c，是否将输入字符回送到屏幕上，输入是否被缓冲等。如果键盘输入功能返回0值，表示此键没有对应的ASCII值（例如功能键）。所读入的下一个字符将作为"扩展ASCII代码"，表示按下的实际键或组合键。

本组中的程序可作为显示简短信息（不必象句柄功能那样要求对字符串长度计数）或转储寄存器内容的快速、简单方法。下面列出的三个宏指令对于出错处理特别有用。

例如，首先使用 prints 显示一条简短出错信息，然后使用 printn 显示出任意寄存器的内容（十进制格式有利于给错误编号），或使用十六进制格式显示的 printh（有利于寻址）。

```
;File:     DISPLAY.MAC

;Assembler Display Macros:
;
;     PRINTS s        Sends string s to standard output.
;
;     PRINTN r        Prints contents of register r in decimal.
;
;     PRINTH r        Hex dump of register r.
;
;------------------------------------------------------------------------

prints    macro   s                          ;;Displays string 's' at
          local   a01, str                   ;;current cursor position.
          jmp     a01                        ;;Sample usage:
str       db      s                          ;;     prints "hello world"
          db      '$'                        ;;Saves all registers.
a01:      push    ax
          push    dx
          push    ds
          mov     ax, cs                     ;;DS:DX must contain address
          mov     ds, ax                     ;;of '$' terminated string.
          mov     ah, 09h                    ;;DOS print string function.
          mov     dx, offset str
          int     21h
          pop     ds
          pop     dx
          pop     ax
          endm

;------------------------------------------------------------------------

printn    macro   reg                        ;;Prints contents of register 'reg'
          local   a01                        ;;in unsigned decimal format.
                                             ;;Sample usage:
                                             ;;     printn si
          push    ax                         ;;Saves all register.
          push    bx
          push    cx
          push    dx
          push    si

          mov     ax, reg                    ;;Place register value in AX.
          mov     cx, 5                      ;;Will print 5 digits.
          mov     bx, 10000                  ;;= 2710h.
a01:      xor     dx, dx                     ;;Zero out upper word of dividend.
          div     bx                         ;;Obtain highest decimal digit.
          mov     si, dx                     ;;Save remainder in SI.
          mov     dl, al                     ;;Move digit to DL.
          add     dl, 30h                    ;;Convert binary to decimal.
          mov     ah, 02h                    ;;DOS display character function.
          int     21h                        ;;Print it.
          mov     ax, bx                     ;;Divide divisor (BX) by 10.

          mov     bx, 10
          xor     dx, dx                     ;;Zero out upper word of dividend.
          div     bx
          mov     bx, ax
          mov     ax, si                     ;;Restore remainder.
          loop    a01                        ;;Divide remainder to get next digit.
```

```
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax

        endm
;-----------------------------------------------------------------------

printh  macro       reg             ;;Prints contents of register 'reg'
local   a01, a02                    ;;in hexadecimal.
                                    ;;Sample usage:
        push    ax                  ;;    printh si
        push    bx                  ;;Saves all registers.
        push    cx
        push    dx

        mov     bx, reg
        mov     ch, 4               ;;Number of digits.
a01:    mov     cl, 4               ;;Set count to 4 digits.
        rol     bx, cl              ;;Left digit to right.
        mov     al, bl              ;;Move to AL.
        and     al, 0fh             ;;Mask off left digit.
        add     al, 30h             ;;Convert hex to ASCII.
        cmp     al, 3ah             ;;Is it-> 9?
        jl      a02                 ;;Jump if digit = 0 to 9.
        add     al, 7h              ;;Digit is a to f.
a02:    mov     dl, al              ;;Put ASCII char in DL.
        mov     ah, 2               ;;Display output function.
        int     21h                 ;;Call DOS.
        dec     ch                  ;;Done 4 digits?
        jnz     a01                 ;;Not yet.
        pop     dx
        pop     cx
        pop     bx
        pop     ax

        endm
;-----------------------------------------------------------------------
```

宏 prints 显示字符串时，将其直接置入码段，跳过它，然后调用显示字符串功能 0Ah，宏 printn 将指定寄存器的内容转换为十进制格式，并使用功能 02h 显示出每个字符。宏 primth 也使用功能 02h，但只是以十六进制格式转储寄存器的内容。

下面的功能从标准输入读入数据：

功能	执行的服务
01h	键盘输入，回送，检查 ctrl—c。
06h	直接控制台I/O，无ctrl—c检查。由于此功能无论字符是否准

备好均有返回，可用于测试键盘状态，无 ctrl－c 检查.

07h　控制台输入，无回送或 ctrl—c 检查.

08h　控制台输入，回送，无 ctrl—c 检查.

0Ah　带缓冲的键盘输入.

0Ch　清键盘缓冲区并调用输入功能.

0Bh　检查标准输入状态.

下面这组功能完成向标准输出写操作:

功能	执行的服务
02h	显示输出，检查 ctrl—c.
06h	直接控制台 I／O．无 ctrl—c 检查.
09h	打印字符串.

下面这组功能管理打印机输出:

功能	执行的服务
05h	向打印机送字符.

下面功能管理串行端口 I／O:

功能	执行的服务
03h	辅助输入
04	辅助输出

2.5.2.2 磁盘管理

磁盘级功能将磁盘设备作为一个整体来管理，它不涉及特定的文件.

curr_drv使用功能19h返回包含当前驱动器的字符串.

下面给出磁盘级管理功能．前两个功能可用于读磁盘时

启动扇区或文件分配表。

功能	执行的服务
int25h	绝对磁盘读。
int26h	绝对磁盘写。
0Dh	磁盘重置。清所有缓冲区，此功能应被Ctrl－C处理程序调用，但并不更新目录登记项(必须由关闭文件来更新目录登记项)。
19h	获得当前驱动器。
0Eh	设置当前驱动器。
1Bh	从当前驱动器获得文件分配表(FAT)信息。
1Ch	从指定设备获得FAT信息。此功能等同于1Bh，但可任意指定驱动器。
2Fh	获得磁盘传送区(DTA)。
1Ah	设置磁盘传送区(DTA)。
54h	获得验证开关。
2Eh	设置／重置验证开关。
36h	获得磁盘自由空间。

```
/*
     File:     CURDRV.C

     A C function that uses DOS function call 19h to obtain the current drive.
*/

main ()
     {
     char *curr_drv (void);

     printf ("The current drive is %s.\n", curr_drv());
     }

#include <dos.h>

char *curr_drv ()                        /* Returns a string containing the */
     {                                   /* current drive designation.      */
     static char *diskname [] = {"A:","B:","C:","D:","E:"};
     union REGS intregs;

     intregs.h.ah = 0x19;                /* Note that 'int86' returns the */
                                         /* value of the AX register.     */
     return (diskname [int86(0x21,&intregs,&intregs) & 0xff]);
     }
```

上面是使用DOS功能19 h返回当前驱动器的C函数。

2.5.2.3 目录管理

由 DOS2.x 推出的目录功能涉及到树状目录结构. 例如下面的汇编程序使用功能 47h 将当前目录路径装入全局变量 cur__dir 中 (注:此程序假定 cur_ dir 位于作为 groupseg 成员的一个段中, 且 DS 已含有此组的基址).

```
ah,47h                                  ;获得当前目录
mov     si,offset groupseg:cur__dir;
mov     di,o                            ;指定默认驱动器
int     21h
```

下面的 DOS 功能用来管理目录。

功能	执行的服务
39h	生成目录
34h	删除目录
47h	获得当前目录
3Bh	设置当前目录

2.5.2.4 文件管理

当前 DOS 提供两组无关的功能:传统的文件控制块 (FCB) 功能与文件句柄功能用于管理单个文件.

▲传统的 FCB 功能

功能	执行的服务
0Fh	打开文件
16h	生成文件
10h	关闭文件
11h	找出第一个匹配文件

12h　　找出下一个匹配文件

13h　　删除文件

17h　　文件换名

23h　　获得文件大小

29h　　文件名语法检查

```
page 50,130
;File:    RENDIR.ASM
;.COM format for a program that renames a directory
;
;    usage:
;          RENDIR [drive:]oldname newname
;
;    To generate .COM file from RENDIR.ASM:
;          MASM RENDIR;
;          LINK RENDIR;
;          EXE2BIN RENDIR.EXE RENDIR.COM
;
include \masm\display.mac              ;Contains macro 'prints', Figure 4.1.

stackseg    segment stack
stackseg    ends

codeseg     segment
codeseg     ends

dataseg     segment

fcb         db      0ffh               ;Extended file control block.
            db       5 dup (0)         ;Reserved.
            db      10h                ;10h is code for directory.
d_old       db      12 dup (?)         ;Old drive and directory name.
            db       5 dup (?)         ;Reserved.
n_new       db      20 dup (?)         ;New name (not including drive) &
                                       ;remainder of FCB.
dataseg     ends

groupseg    group   codeseg,dataseg
            assume  cs:groupseg
            assume  ds:groupseg
            assume  es:groupseg

codeseg     segment
                                       ;PSP values entered by user.
            org     5ch
dir1        db      12 dup (?)         ;Old drive and directory name.
            org     6dh
name2       db      11 dup (?)         ;New name only (skips drive at 6Ch).

            org     100h
main        proc    near

            mov     cx, 12             ;Move old drive and directory into FCB.
            mov     si, offset dir1
            mov     di, offset groupseg:d_old
            cld
            rep     movsb
```

```
mov     cx, 11                      ;Move new name into FCB.
mov     si, offset name2
mov     di, offset groupseg:n_new
cld
rep     movsb

mov     ah, 17h                     ;FCB function 17h renames files,
mov     dx, offset groupseg:fcb     ;directories, or volume labels.
int     21h
or      al, al
jz      m01                         ;AL = FFh => error.
prints  "no such directory or duplicate name"
mov     ax, 4c01h                   ;Exit w/ errorlevel 1.
int     21h
m01:        mov     ax, 4c00h                   ;Exit w/ errorlevel 0.
            int     21h

main        endp

codeseg     ends

            end     main
```

上面是使用DOS功能17h给目录换名的COM程序。

▲句柄功能

五个标准句柄如下:

句柄	名字	设备
0000	标准输入	CON
0001	标准输出	CON
0002	标准错误	CON
0003	标准辅助设备	AUX
0004	标准打印机设备	PRN

前两个标准句柄可由用户重定向。例如，如果程序UNWS 使用如下命令行:

UNWS<FILE.DOC>FILE.TXT

则从标准输入句柄 (0000) 读人的内容来自文件FILE.DOC，而写到标准输出句柄 (0001) 的内容将引导向文件 FILE.TXT. 有时不希望用户重定向控制台 I/O，因为某个程序可能必须与控制台通信. 欲产生向屏幕的输出(不准许重定向)，可使用标准错误 (0002) 来代替标准输出

(标准错误典型地用于向用户发送重要信息，因此是不可重定向的)。

避免控制台输入、输出重定向的另一种方法是使用不同的句柄打开 CON 设备。例如，下面的汇编程序打开控制台设备并保存返回后的句柄。后面有关此句柄的读、写不可被重定向。

```
datascg   segment
fname   db'CON', 0 ;控制台设备的 ASCIIZ 名字
console dw?         ;控制台句柄
dataseg  ends
mo    ah, 3dh       ;句柄文件打开
mov   dx,offset group seg:fname;
mov   al,2           ;打开作读、写
int   21h
jc   error           ;跳至出错处理程序
mov console,ax       ;否则保存句柄
```

注意，在 C 中使用 stream 文件函数时，标准句柄可由使用下列预定义的 stream（流）之一来直接访问（由 FILE 指针）：

DOS 标准句柄	C 预定义的 stream
标准输入	stdin
标准输出	stdout
标准错误	stderr
标准辅助设备	stdaux
标准打印机	stdprn

某些 C 函数，如 getchar 及 putchar，自动地使用标准

输入和标准输出。

下面是用于管理文件的句柄功能:

功能	执行的服务
3Ch	生成文件。
5Bh	生成新文件。
5Ah	生成唯一文件。
3Dh	打开文件。
3Eh	关闭文件。
41h	删除文件。
43h	改变文件方式。
45h	获得新的重复文件句柄。此功能可用于周期性地为某个文件更新目录登记项，而无需关闭并重新打开原始的句柄。为同一文件简单地请求并关闭一个附加句柄。
46h	强使重复现存的句柄。此功能有助于重定向标准设备句柄。
4Eh	找到第一个匹配文件。
4Fh	找到下一个匹配文件。
56h	文件换名。
57h	获得／设置文件日期和时间。

```
page 50,130

;File:    NOBAK.ASM

;.COM format for a program that switches off the file archive bit
;
;     Usage:
;          NOBAK filespec                 (filespec can include wildcards)
;
;     To generate .COM file from NOBAK.ASM:
;          MASM NOBAK;
;          LINK NOBAK;
;          EXE2BIN NOBAK.EXE NOBAK.COM

include    \masm\display.mac              ;Contains 'printn' and 'prints' macros
                                          ;(Figure 4.1).
```

```
stackseg    segment stack
stackseg    ends

            assume  cs:codeseg
            assume  ds:codeseg

codeseg     segment

            org     80h
dta         label   byte                    ;The default disk transfer area (DTA).
parmlen     db      ?                       ;Length of parameter line.
parmline    db      127 dup (?)             ;Parameter line starts here (includes
                                            ;leading blank).
            org     100h
main        proc    near

            xor     bh, bh                      ;Make parameter ASCIIZ (i.e.
            mov     bl, parmlen                 ;place 0 after last byte).
            mov     parmline[bx], 0

            mov     ah, 4eh                     ;Find first matching file.
            mov     dx, offset parmline+1       ;NB: assumes single blank!
            mov     cx, 0007h                   ;File attribute for search.
            int     21h
            jnc     m01
            prints  'no files found'            ;Error: print message &
            mov     ax, 4c01h                   ;bail out w/ errorlevel 1.
            int     21h

m01:        mov     ah, 43h                     ;Chmod function.
            mov     dx, offset dta + 30         ;Position of found file name.
            mov     cl, dta[21]                 ;Place existing attribute in cl.
            and     cx, 00dfh                   ;Mask off archive bit.
            mov     al,1                        ;'Set' subfunction.
            int     21h
            jnc     m02
            printn  ax                          ;Error: print error code &
            prints  ' chmod error'              ;quit w/ errorlevel 1.
            mov     ax, 4c01h
            int     21h

m02:        mov     ah,4fh                      ;Find next file.
            int     21h
            jc      m03                         ;Not an error, just no more files.
            jmp     m01

m03:        mov     ax,4c00h                    ;Exit w/ errorlevel 0.
            int     21h

main        endp

codeseg     ends

            end     main
```

上面是使文件档案比特复位的COM程序。

2.5.2.5 文件的记录／字节级管理

一个文件一旦被打开（通过某个打开或生成功能，或使用了五个预定义的设备处理程序之一），本节所列的功能便提供了传送单个字节或是整块数据的一种方法。

▲传统的 FCB 功能

下面给出执行文件 I／O 的传统 FCB 功能：

功能	执行的服务
14h	顺序读
15h	顺序写
24h	设置有关的记录域
21h	随机读，单条记录
22h	随机写，单条记录
27h	随机块读，多条记录
28h	随机块写，多条记录

▲句柄功能

如前所述，使用句柄功能读、写基本字符设备时可由实施功能 44h（设备的 I／O 控制）将设备方式从"加工过的"改变为"未加工的"而进行优化。改变方式还可用于控制输入是否带有回送等。在加工过的 I／O 方式下（设备的默认方式，不可用于磁盘文件）：

- 对每个字符检查是否为 Ctrl−S，Ctrl−P 及 Ctrl−C。
- 输入字符回送到标准输出。
- Tabs 扩展成为空格。
- 使用常用的 DOS 功能键编辑输入数据。

在未加工方式下，所有上述特点一律消失，从而改进了性能。注意，标准输入、标准输出或标准错误的方式改变后，三种句柄的方式也自动改变，因为所引用的实际设备是相同的。用IBM技术参考手册中更准确的语言说是"经加工的"方式，而"未加工的"方式便是二进制方式。

下面给出了功能 40h，44h 及 47h 的例子。

```
page 50,130

;File:   CURDIR.ASM

;.COM file format for a program that obtains the current directory, and
;writes it to the standard output, after changing device mode to "raw"
;
;    To generate .COM file from CURDIR.ASM:
;           MASM CURDIR;
;           LINK CURDIR;
;           EXE2BIN CURDIR.EXE CURDIR.COM

stackseg    segment stack
stackseg    ends

codeseg     segment
codeseg     ends

dataseg     segment

cur_dir     db      64 dup (?)          ;Buffer for holding current directory.

dataseg     ends

groupseg    group   codeseg,dataseg
            assume  cs:groupseg
            assume  ds:groupseg
            assume  es:groupseg

codeseg     segment

            org     100h
main        proc    near

            mov     ah, 47h                 ;Use DOS function 47h to load
                                            ;current directory into 'cur_dir'.
            mov     si, offset groupseg:cur_dir
            mov     dl, 0                   ;Specifies the default drive.
            int     21h

            call    rawmode                 ;Change stdin/out/err to "raw"
                                            ;mode to improve performance.

                                            ;Print 0-terminated contents
                                            ;of 'cur_dir'.
            mov     dx, offset groupseg:cur_dir
            call    printz

            call    cookmode                ;Restore stdin/out/err to default
                                            ;"cooked" mode.

            mov     ah, 4ch                 ;Exit w/ errorlevel 0.
            mov     al, 0
            int     21h

main        endp

rawmode     proc    near            ;Places standard input/output/error
                                    ;in "raw" mode.
                                    ;Alters registers: AX,BX,DX.

            mov     ax, 4400h       ;Read "device information" using
                                    ;function 44h, subfunction 00h.
            xor     bx, bx          ;Handle for standard input.
            int     21h             ;Places device information in DX.

            xor     dh, dh          ;Zero out upper byte of DX.
            or      dl, 20h         ;Turn on "raw" bit, #5.
            mov     ax, 4401h       ;Write back "device information" using
                                    ;function 44h, subfunction 01h.

            xor     bx, bx          ;Handle for standard input.
            int     21h
```

```
        ret
rawmode endp

cookmode proc   near            ;Places standard input/output/error
                                ;in "raw" mode.
                                ;Alters registers: AX,BX,DX.

        mov     ax, 4400h       ;Read "device information" using
                                ;function 44h, subfunction 00h.
        xor     bx, bx          ;Handle for standard input.
        int     21h             ;Places device information in DX.

        xor     dh, dh          ;Zero out upper byte of DX.
        and     dl, 0dfh        ;Turn off "raw" bit, #5.
        mov     ax, 4401h       ;Write back "device information" using
                                ;function 44h, subfunction 01h.
        xor     bx, bx          ;Handle for standard input.
        int     21h

        ret

cookmode endp

printz  proc    near            ;Prints a zero terminated string on
                                ;standard output.
                                ;On entry:
                                ;       DS:DX = address of string.
                                ;Alters registers: AX,BX,CX,DI,ES.

        mov     ax, ds          ;Calculate length of string and place
        mov     es, ax          ;in register CX.
        mov     di, dx
        xor     al, al
        cld
p01:    scasb
        jnz     p01
        sub     di, dx
        mov     cx, di

        mov     ah, 40h         ;Handle write function.
        mov     bx, 0001        ;Standard output handle.
        int     21h             ;Transfer to DOS.

        ret

printz  endp

codeseg ends

        end     main
```

此程序首先将当前目录装入变量 cur_dir（使用功能 47h，自动将 0 追加到字符串末尾)，然后调用 rawmode，此子程序使用功能 44h 将标准输出改变为未加工方式．改变为未加工方式的目的是说明一种改进性能的方法，但在实际应用中，除非要显示的数据量非常可观，一般并不进行方式改变。一旦改变了方式，便调用 pritz 将 cur-dir 的内容通过功能 40h 写到标准输出．函数 printz 对写以 0 结尾的

字符串很有用，因为它能自动计算出长度。最后调用 cookmode 将方式恢复为默认的“经加工”方式。(否则，后面的程序会出现意想不到的结果，因为在未加工方式下 tab 字符不再扩展成为 8 个空格，而是作为一个小圆圈显示)。

下面给出了函数 readc，它使用句柄函数 3Fh 将一个字符读入寄存器 AL。此函数说明了如何将范围在 01h—0Ch 间的传统字符设备功能用更多的标准句柄功能来仿模。如果标准输入在未加工方式下调用此函数，其结果很象是功能 07h，不回送也不检查该字符，也不等待回车便立即返回。

```
page 50,130

;File:    READC.ASM

include   DISPLAY.MAC                    ;Contains macros 'prints' & 'println',
                                         ;Figure 6.1.

readc     proc     near                  ;Reads a single character from stdin
                                         ;into register AL, using function 3Fh.
                                         ;Alters registers: AX, BX, CX, DX.
                                         ;Returns -1 in AL for end of file.
          jmp      r01

charbuf   db       ?                     ;Function 3Fh requires a buffer.

r01:      push     ds
          mov      ax, cs                ;Set DS to segment of 'charbuf' (in CS).
          mov      ds, ax
          mov      ah, 3fh               ;Read from file or device function.
          mov      bx, 00h               ;Handle for standard input.
          mov      dx, offset charbuf    ;Address of buffer.
          mov      cx, 1                 ;Read single character.
          int      21h
          jnc      r02                   ;Test for error.
          prints   "dos function 3f error "
          println  ax
          mov      ax, 4c01h             ;Quit with errorlevel 1.
          int      21h

r02:      or       ax, ax                ;Test for end of file.
          jnz      r03
          mov      al, 0ffh              ;Return -1 (ffh) in AL for eof.
          pop      ds
          ret

r03:      mov      al, charbuf           ;Not end of file.
          pop      ds
          ret

readc     endp
```

下面是执行文件 I/O 的 DOS 句柄功能:

功能	执行的服务
44h	设备 I/O 控制
3Fh	自文件/设备读
40h	自文件/设备写
42h	移动读写指针

2.5.2.6 网络管理功能

DOS 版本 3.0 第一次提供了支持网络的功能。注意，功能 5Eh 及 5Fh 只可用于 DOS 版本 3.1 以后，需装入 IBM—PC Network 程序。

下面为网络管理功能清单:

功能	执行的服务
5Ch	对文件访问加锁/解锁
5Eh	AH=00h 获得机器名
	AH=02h 建立打印机设置
	AH=03h 获得打印机设置
5Fh	AH=02h 获得重定向表登记项
	AH=03h 设备重定向
	AH=04h 撤消重定向

2.5.2.7 内存及进程管理

DOS2.0 推出了一组用于分配和解除分配系统内存块的函数。还推出了一个从父进程中运行一个子进程的功能，因此有可能使一个程序去运行一个独立的 EXE 或 COM 文件，然后再返回到以前正在进行的任务。内存和进程管理功

能分成一组是因为使用内存管理服务功能的主要目的是释放内存以运行子进程. 装入并运行另一个程序所需的步骤有些复杂，最好的办法是通过实例来介绍.

下面的汇编语言程序说明了使用 DOS 功能 4Ah（修改已分配内存块）和 4Bh（执行程序）来装入和运行一个子进程. 该程序还使用了功能 46h（强行复制句柄）对标准输出和标准错误进行重定向. 如前所述，标准错误是不能够从命令行上重定向的，此程序介绍了如何从父进程内为子进程进行重定向. 此实用程序装入并运行 MASM.EXE，将它传送给用户打入的命令行. 从 MASM 输出的重定向到命令行所指定源文件的同名文件，但扩展名为 ERR. 此实用程序可提供给 Microsoft MASM 版本 4.0 用户，该版本不允许"错误输出"重定向到带 > 命令行符号的文件. 基本步骤如下:

(1) 重新分配堆栈. 设置堆栈指针以指向数据段中上一次定义之标号上方 1024 字节，从而使 1K 的堆栈空间处于程序末尾，且不使用 db 进行保存（那样会不必要地加长磁盘上的 COM 文件）.

(2) 释放程序上方已分配的内存. 因为初始化时将所有自由内存均分配给了 COM 文件，故使用功能 4Ah 去释放堆栈末尾上方的内存，为子进程提供尽可能多的内存. 注意，即使对于不执行子进程的程序，释放不使用的内存也是有必要的,至少可使该程序上多任务环境下能够增加兼容机会.

(3) 由将 ERR 追加到命令行上源文件名之后来建立 fname("错误文件"名).

(4) 打开"错误文件"fname,将文件句柄置于 handle中.

(5) 使用功能 46h（强行复制句柄）将标准输出句柄

(0001)、标准错误句柄（0002）强行复制成同一个句柄 fhandle。这样一来，以后的标准输出或标准错误的输出便重定向到该错误文件。由于子进程继承父进程所有已打开的句柄，因此在子进程执行过程中重定向保持有效。

(6) 将寄存器 SS,SP 保存在码段可寻址的内存中，并在功能返回后立即恢复它们。

(7) 使用功能 4Bh 运行子进程，必须为此功能建立参数块（且程序清单）。

(8) 从功能4Bh返回时，关闭"错误文件"并结束父进程。

```
page 50,130

;File:    RMASM.ASM

;.COM format for program to run MASM.EXE as a child process after redirecting
;all output to an error file
;
;     Usage:
;          RMASM sourcefile, ...         (same command line as used with MASM)
;
;     To generate .COM file from RMASM.ASM:
;          MASM RMASM;
;          LINK RMASM;
;          EXE2BIN RMASM.EXE RMASM.COM

include     DISPLAY.MAC                    ;Contains 'printn' and 'prints',
                                           ;Figure 4.1.

STD         equ     1                      ;1 => redirect standard output.
ERR         equ     1                      ;1 => redirect standard error.

stackseg    segment stack                  ;Fake stack segment to appease linker.
stackseg    ends

codeseg     segment
codeseg     ends

dataseg     segment

fname       db      64 dup (?)             ;Build up error file name here.
fext        db      '.err', '$'            ;File extension for error file.
fhandle     dw      ?                      ;File handle for error file.
                                           ;Name of program to run:
masm        db      'c:\masm\masm.exe',0

parmblk     label   word                   ;Parameter block for function 4Bh.
environ     dw      0                      ;0 => inherit parent's environment.
comm_off    dw      80h                    ;Offset of command line.
comm_seg    dw      ?                      ;Com. line segment: fill in at runtime.
fcb1        dd      -1                     ;File control blocks to copy.
fcb2        dd      -1                     ;-1 => don't copy FCB.

s_seg       dw      ?                      ;Save area for SS and SP.
s_ptr       dw      ?
```

```
stackstart label byte                   ;1024 k stack begins here, but is not
                                        ;"db'ed" to save disk space.

dataseg     ends

groupseg    group   codeseg, dataseg
            assume  cs:groupseg
            assume  ds:groupseg
            assume  es:groupseg

codeseg     segment

            org     80h
parmlen     db      ?                   ;Length of command line.
parm        db      127 dup (?)         ;Full command line.

            org     100h
main        proc    near
                                        ;Lower the stack.
            mov     sp, offset groupseg:stackstart + 1024

            mov     ah, 4ah             ;Reduce memory allocation to make room
                                        ;for child process.
                                        ;Load new memory size into bx, add 15
                                        ;and shr by 4 to convert to paragraphs.
            mov     bx, offset groupseg:stackstart + 1024 + 15

            mov     cl, 4
            shr     bx, cl
            int     21h                 ;Reduce memory block.
            jnc     m01
            printn  ax                  ;Error: print message and quit.
            prints  ' error on function 4Ah'
            mov     ax, 4c01h
            int     21h

m01:                                    ;Build up error file 'fname'.
            xor     bh, bh              ;Mark end of parameters w/ FFh.
            mov     bl, parmlen
            mov     parm[bx], 0ffh
                                        ;Initialize SI, DI, & DF for transfer.
            mov     si, offset groupseg:parm
            mov     di, offset groupseg:fname
            cld
m02:        cmp     byte ptr[si], ' '   ;Scan past leading blanks in parm.
            jne     m03
            inc     si
            jmp     m02
m03:        cmp     byte ptr[si], '.'           ;End of source file name is
            je      m04                         ;indicated by one of these
            cmp     byte ptr[si], ':'           ;characters, therefore STOP
            je      m04                         ;copying from 'parm' to 'fname'.
            cmp     byte ptr[si], ' '
            je      m04
            cmp     byte ptr[si], '/'
            je      m04
            cmp     byte ptr[si], ','
            je      m04
            cmp     byte ptr[si], 0ffh
            je      m04
            movsb                       ;Copy a character from parm to fname.
            jmp     m03                 ;Go back for another character.

m04:        mov     cx, 5               ;Copy 5 byte 'fext' onto end of 'fname'.
            mov     si, offset groupseg:fext
            rep     movsb

            mov     ah, 3ch             ;Open error file.
            mov     dx, offset groupseg:fname
            mov     cx, 0               ;Normal file attribute.
            int     21h
            jnc     m05
            printn  ax                  ;Error: print message and quit.
            prints  ' error on function 3Ch'
            mov     ax, 4c01h
            int     21h
```

```
m05:
if STD
        mov     fhandle, ax             ;Redirect standard out -- i.e. force
        mov     ah, 46h                 ;standard output file handle to refer
        mov     bx, fhandle             ;to the error file.
        mov     cx, 0001
        int     21h
        jnc     m06
        printn  ax
        prints  ' error on function 46h'
        mov     ax, 4c01h
        int     21h
endif

m06:
if ERR
        mov     ah, 46h                 ;Redirect standard error.
        mov     bx, fhandle
        mov     cx, 0002

        int     21h
        jnc     m07
        printn  ax
        prints  ' error on function 46h'
        mov     ax, 4c01h
        int     21h
endif
                                        ;Execute program function.
m07:    mov     ah, 4bh                 ;Pointer to program name.
        mov     dx, offset groupseg:masm ;Fill in segment of command line.
        mov     comm_seg, cs            ;Point BX to parameter block.

        mov     bx, offset groupseg:paramblk
        mov     al, 00                  ;0 => EXEC subfunction.

        push    ds                      ;Save any registers here.
        mov     s_seg, ss               ;Save SS & SP in memory addressible
        mov     s_ptr, sp               ;from CS.
        int     21h                     ;Branch to child process here.
        mov     ss, cs:s_seg            ;Restore SS & SP FIRST.

        mov     sp, cs:s_ptr
        pop     ds                      ;Restore any other registers here.

        mov     ah, 3eh                 ;Close the error file.
        mov     bx, fhandle
        int     21h
        jnc     m08
        printn  ax
        prints  ' error on function 3eh'
        mov     ax, 4c01h
        int     21h

m08:    mov     ax, 4c00h               ;Exit w/ errorlevel 0.
        int     21h

main    endp

codeseg ends

        end     main
```

下面是管理内存和进程的 DOS 功能。

功能	执行的服务
26h	生成新的程序段前缀。注意，此方法是过时的，建议使用功能 4Bh。

48h　　分配内存.

49h　　释放已分配的内存.

4Ah　　修改已分配的内存块.

4Bh　　装人并执行程序.

4Dh　　获得子进程的返回代码. 为使用此功能, 子进程必须通过 DOS 功能 4Ch 来结束.

2.5.2.7　程序结束

MS-DOS 提供了两种结束程序的基本方法:临时应用程序结束后释放所占用的内存, 程序结束但将代码驻留在内存中.

▲简单结束

可由发出 ret 指令简单地结束程序, 但堆栈必须包含正确的值（指令 ret 的结果是指向程序段前缀的第一个字节, 该处含有一条 int20h 指令）. 更安全的办法是使用 DOS 结束功能之一. 较推荐的功能是 4Ch, 因为它能够返回一个出错代码（此代码可在批文件中通过 errorlevel 来测试, 或在父进程中通过功能 4Dh 测试）. 功能 4Ch 是唯一不需要 CS 寄存器包含程序段前缀段地址的结束功能. 注意, 虽然这些功能执行某些扫尾工作并关闭所有打开的文件句柄, 但改变了长度的文件必须由应用程序本身来关闭, 以保存它们的目录登记项能够正确地被更新.

下面给出简单结束功能.

功能	执行的服务
int 20h	程序结束.
00h	程序结束.
4Ch	结束一进程. 此为推荐的方法.

▲结束并驻留

功能 31h 是结束一个内存驻留程序的推荐的方法，因为它返回一个出错代码（与功能 4Ch 一样），并且允许程序能保持多于 64K 的内存。

下面给出结束且驻留功能。

功能	执行的服务
int27h	结束并驻留。注意此功能不允许程序返回出错代码，并限制驻留程序不得大于64K．
31h	结束并驻留。此为推荐的方法。

2.5.2.8 其他功能

功能	执行的服务
int2Fh	打印机假脱机程序访问与多路中断。
35h	获得中断向量。
25h	设置中断向量。
2Ah	获得日期。
2Bh	设置时间。
2Ch	获得时间。
2Dh	设置时间。
30h	获得 DOS 版本号。
33h	获得／设置Ctrl-Break检查状态。
38h	获得／设置与国家有关的信息。
44h	设备 I／O 控制。
59h	获得扩展错误信息。

第三章 BIOS 功能

IBM–PC BIOS(基本输入输出系统)包含在随硬件的只读存储器中,为 MS–DOS 程序员提供了另一组有用的功能。与 DOS 功能相比,这些功能处于更低的层次上。这可以从两个方面来看:

首先,遵循操作系统的经典设计方法,MS–DOS 是以层次方式建立的。较高的层次是使用由较低层次提供的资源实现的。应用程序通常调用 DOS 功能, DOS 功能再去调用 BIOS 功能。按照经典的设计方法,某个较高层次亦可越过下一个较低层次而直接去访问最低层次。因此程序员也可以越过 DOS 直接去访问 BIOS,这样可省去两个软件层次之一而获得更高的性能。

其次,BIOS 功能控制了许多与 IBM–PC 体系结构有关的功能。DOS 提供了许多通用的服务功能, 而 BIOS 提供了许多与硬件有关的功能,如设置视频显示方式和属性,显示图形,格式化磁盘扇区,获得可用设备清单等。因此 BIOS 是获得最优软件性能的主要工具。

但应注意到 BIOS 并不是系统中的最低层次,更低的层次是特定的内存地址和与硬件相关的端口。BIOS 设计时考虑到相当程度的硬件独立性,这样即使修改了硬件也不致改变其服务程序接口。

与 DOS 提供的功能相比较,BIOS 功能有两个主要优点,即更高的性能及更多的 DOS 未提供的功能。文本和图形视

频显示的管理是 BIOS 大大优于 DOS 的领域，当然还包括磁盘驱动器、打印机等的控制。

BIOS 也有一些缺点，例如缺乏对于非 IBM 兼容的 MS-DOS 机器的移植性，并且缺乏象文件系统那样的由 DOS 提供的可管理的逻辑资源。BIOS 提供的服务程序可用于实行磁盘 I／O，但仅针对单个扇区。文件的概念仅在 MS-DOS 级存在，故要编写高性能的程序，对于磁盘操作系统，通常使用 DOS 服务程序更好(不管怎么说 DOS 毕竟是磁盘操作系统)。例如，使用 DOSCOPY 命令，较之直接调用 BIOS 的汇编程序速度更快。

3.1 BIOS 功能的一般用法

本节介绍使用 BIOS 中断功能的一般准则。注意，PC／AT 技术参考手册中的文本只适用于 IBM-PC／AT 母板上 ROM 的 BIOS 代码，并在段 FOOOh 处开始。而部分 BIOS 程序代码可能处于其他地址代码，用以提供扩展的或径修改的服务功能。系统启动时，BIOS 程序扫描内存，查找合法的、可能处于各种适配器板上的 ROM 代码。合法的代码块被调用来执行初始化，并有可能去修改 BIOS 中断向量以指向它们自己的新程序。例如，如果安装了硬盘控制器，便可找到指向 C800h 段的一个程序的 BIOS 中断 13h(磁盘服务程序)向量 (而不是指向 F000h 段的原始程序)。又如安装了 EGA 适配器，视频服务向量 10h 可能指向 C000h 块。因此如果编写程序去使用某些适配器，便应参考相应的技术文件。

有些软件程序亦可能截获 BIOS 向量，并修改或扩展所提供的服务程序(例如 Topview 通过中断 10h 提供了附加的视频服务)。这类系统所提供的文档可以覆盖掉 PC／AT 技

术参考手册中相应的内容.

下面给出使用 BIOS 编程的一般准则:

- 对于给定的中断号,功能号放在寄存器 AH 中.
- 一般来说, BIOS 服务程序保存除 AX、标志寄存器以及文本中特别指定要返回值的其他寄存器以外的所有寄存器.
- BIOS 代码程序总是通过正式的中断号来访问,而绝不要通过可能改变的绝对地址等.

3.2 BIOS 功能分类

BIOS 功能的组织比 MS-DOS 功能的组织简单得多.它不但重复很少,且各种服务已很好地根据中断类型分了类. 每种中断类型控制一种设备. 但 IBM BIOS 的文档是远不能满足要求的. 本节后面将给出每种服务功能的调用步骤.

3.2.1 视频服务功能:中断 10h

ROM BIOS 视频服务最突出的优点是,几乎所有程序均可由用户软件调用而不用考虑所用的显示控制器或视频显示方式——单显适配器、彩色/图形适配器或是 EGA 适配器. 视频服务程序确定当前的显示类型并执行所有必须的地址翻译,例如程序打算在图形方式下显示文字,则应从一张驻留表中提取比特图形.

调用序列:

视频驱动程序的普通调用序列如下:

```
mov  ah, function          ;AH 包含功能号
:                          ;其他寄存器中装人
                           ;与调用有关的参数
unt 10h                    ;传送给 ROM  BIOS
```

虽然每个 ROM BIOS 视频功能具有不同的变元并返回不同的值,但均遵守一定的规则:

• 寄存器 AH 中包含调用号. 如果寄存器 AH 中的号不在视频驱动程序合法功能号范围内,就不发生任何动作(合法功能号随机器类型及显示适配器而变).

• 欲写的字符或象素值通常在寄存器 AL 中传送.

• 在所有视频驱动程序调用中,均要保存 BX,CX, DX 寄存器和段寄存器。其他寄存器尤其是 SI 和 DI 的内容有可能被破坏.

• X 坐标(列号)在图形功能下通过 CX 传送,在文本功能下通过 DL 传送。

• Y 坐标(行号)在图形功能下通过 DX 传送,在文本功能下通过 DH 传送。

• 显示页通过寄存器 BH 传送.

```
/*
     File:      VIDEO.C
     This file contains a set of routines that use the BIOS video services
     interrupt (10h) to get and set the video mode and cursor parameters
*/

struct cursor                                        /* Cursor parameters. */
     {
     unsigned char row;
     unsigned char col;
     unsigned char startline;
     unsigned char stopline;
     };

struct crtmode                                       /* Video mode parameters. */
     {
     unsigned char videomode;
     unsigned char columns;
     unsigned char videopage;
     };
                                   /* Forward declarations for type checking. */
int getcur (struct cursor *, int);
int setcur (struct cursor *, int);
int getmode (struct crtmode *);
int setmode (struct crtmode *);

main ()
     {
     struct cursor oldcur;                 /* Structures for getting & setting */
     struct cursor newcur;                 /* cursor parameters & for getting  */
     struct crtmode oldmode;               /* & setting video mode parameters. */
     struct crtmode newmode;
```

```
                              /* Save current video state and initilize new state. */

    getmode (&oldmode);              /* Save current video mode parameters. */

    getcur (&oldcur, oldmode.videopage); /* Save current cursor parameters.*/

    if (oldmode.videomode != 7)
         {                               /* If not mono, set mode for 80 column */
         newmode.videomode = 3;          /* color text and page 0.              */
         newmode.videopage = 0;
         setmode (&newmode);
         }
                                    /* Use setcur to hide the cursor.          */
    newcur.row = 30;                /* Place cursor beyond possible position. */
    newcur.col = 80;
    newcur.startline = 32;          /* Turning on bit 5 zaps the cursor.      */
    newcur.stopline = 0;
    setcur (&newcur, 0);            /* Set cursor parmeters for video page 0. */

                                         /* Place main application here ... */


                              /* Before exit, restore previous video state. */
    setmode (&oldmode);
    setcur (&oldcur, oldmode.videopage);

    } /* end main */


#include <dos.h>                    /* These two lines are necessary for all of */
union REGS reg;                     /* following functions.                    */


getcur (curptr, page)               /* This function obtains the current cursor */
struct cursor *curptr;              /* position and 'type' (start and stop row) */
int page;                           /* for the specified video 'page'. Rows and */
     {                              /* columns are numbered starting from 0.    */

     reg.h.ah = 3;
     reg.h.bh = page;
     int86 (0x10, &reg, &reg);
     curptr->row = reg.h.dh;
     curptr->col = reg.h.dl;
     curptr->startline = reg.h.ch;
     curptr->stopline = reg.h.cl;

     } /* end getcur */


setcur (curptr, page)    /* This function sets all of the cursor parameters; */
struct cursor *curptr;   /* rows and columns are numbered starting from 0.   */
int page;
     {
     reg.h.ah = 2;
     reg.h.dh = curptr->row;
     reg.h.dl = curptr->col;
     reg.h.bh = page;
     int86 (0x10, &reg, &reg);

     reg.h.ah = 1;                  /* Note: sets cursor style for all pages. */
     reg.h.ch = curptr->startline;
     reg.h.cl = curptr->stopline;
     int86 (0x10, &reg, &reg);

     } /* end setcur */


getmode (modeptr)                 /* This function gets the current video mode, */
struct crtmode *modeptr;          /* number of display columns, and video page. */
```

```
     {
     reg.h.ah = 15;
     int86 (0x10, &reg, &reg);
     modeptr->videomode = reg.h.al;
     modeptr->columns = reg.h.ah;
     modeptr->videopage = reg.h.bh;

     } /* end getmode */


setmode (modeptr)                  /* This function sets the video mode and  */
struct crtmode *modeptr;           /* display page;  note that the field     */
                                   /* 'columns' is not used, since this value */
     {                             /* is implicit in the mode.               */
     reg.h.ah = 0;
     reg.h.al = modeptr->videomode;
     int86 (0x10, &reg, &reg);

     reg.h.ah = 5;
     reg.h.al = modeptr->videopage;
     int86 (0x10, &reg, &reg);

     } /* end setmode */
```

上面是一组使用BIOS的视频显示程序。它用一个C程序演示了BIOS的视频服务程序的使用.这个程序包含四个过程：getmode,setmode,getcur和setcur,并且用它们来管理基本的视频参数。这些C过程都利用了中断10h服务程序的功能。大致可以概括如下:

getmode过程使用了15号功能,以获得:

- 当前的视频方式。
- 显示的列数。虽然这一数值隐含在当前视频方式中,在使用06或07号功能清屏以前可以直接方便地得到它。
- 当前的显示页。

setmode过程使用了:

- 0号功能以设置视频方式。
- 5号功能以设置显示页。

过程getcur使用了3号功能,以获得:

- 当前光标位置。注意,行数和列数均从0开始,每一显示页具有独立的光标位置。

· 光标"样式",即通常产生光标的起始行和终止行. 对于单色系统,它在 0 到 13 范围内(缺省值起始行=12,终止行=13). 对于彩色系统(CGA 和 EGA),它在 0 到 7 范围里(缺省值起始行=6,终止行=7). 一个分散的光标可以用规定起始位置大于终止位置的方法获得.

过程 setcur 使用了:

· 2 号功能以设置光标位置. 光标位置是为一个特定的显示页面设置的.

· 1 号功能用以设置光标类型. 与设置光标位置不同,光标类型同时是为所有显示页面设置的.

注意,BIOS 要求使用一个单独的功能块以获得当前的视频方式或光标的参数,但可以用两个功能块去设置同样的参数,C程序提供了一个更趋均衡的视频介面,例如管理视频方式. 一个单独的过程得到所有参数;一个单独的过程设置所有参数. 使用相同的数据结构(struct crtmode)可以调换两个过程的所有参数值.

任何用于修改视频方式或光标类型的程序必须能够保存和重现原有的参数值. 这一点对于用最高效能显示一个屏幕窗口的内存驻留方法来说尤为关键,它必须能准确地恢复视频状态,包括光标位置. 上面程序中的main过程给出了使用这些 C 视频功能的可能的步骤和次序:

· 过程 getmode 保存当前的视频方式参数.

· 一旦当前视频显示页通过调用 getmode 而得知,就能用 getcur 保存这一页的光标参数.

· 如果不是单色显示(方式 7),则将使用 setmode 过程把屏幕设置为 80 列彩色正文(方式 3)和 0 页显示状态.

·通过调用 setcur 过程,用使光标消失的方式设置光标参数。光标的隐藏是通过设置 startline 状态字的第 5 位来实现的。当隐藏工作完成后,光标位置被设置在屏幕下方一英寸的位置上。

·紧接着是应用的主体。

·在退出之前,要指出程序开始时用这些功能调用保存的视频方式和光标参数值,通过调用 setmode 和 setcur 过程予以恢复。

注意,上面的视频显示程序没有为置方式和写屏以前而保留当前屏幕数据所作好的准备。保留和恢复屏幕数据的过程将在第六章提供。使用一个存储器驻留程序显示一个具有前景要求的窗口对所遇到的问题,将在后面加以讨论。

使用 BIOS 服务程序的另一个例子是 printa 过程("print with attributes"),见图 3—1。printa 要求具有更好的视频显示控制,包括对视频显示属性和颜色的规范要求、在屏幕上显示字符串时的位置(比简单地在当前位置上显示要复杂得多)。同样,正象第八章将要演示的,这一功能比使用 DOS 程序进行显示,速度大大地加快了。

printa 过程使用 02 号功能块进行光标定位,使用 09 号功能块进行具有特定的颜色或单色属性的字符的写入工作。注意,09号功能块的作用不是校正光标位置。当每个字符显示之后,光标必须用02号功能块进行重新定位。09功能块程序主要是在进行窗口设计时,在某些特定点显示短信息或少量数据时有用,显示到最后一列时,正确地滚动到下一行。但是,它却不适合用于大量的不定长信息的显示。因为有些特殊的字符,象退格和回车,不能被自动处理(这些字符以图形符号进行显示。例如 CR 是一个音乐的音符)。同样地,当最后一行

装满后,屏幕不再滚动了(以后的字符自然就消失了).

下表概括了 BIOS 的中断 10h 的视频功能."功能"是指在访问中断 10h 时,放置于寄存器 AH 中的具体服务程序的号码.

功能	服务程序的执行
00	置视频方式. 注意,黑白方式0,2和6只抑制合成监视器的彩色信号,但不抑制标准的 RGB 或 EGA 监视器的彩色信号.
01	置光标类型
02	置光标位置
03	读光标位置和类型
04	读光笔位置
05	设置有效显示页. 这一服务程序只对CGA 和EGA的正文方式有效. 注意,数据可往任何页里写,光标也可移动,不论它是不是当前正在显示的那一页. 这样,数据可以在一个暂不显示的页里慢慢地建立起来,然后使用这一功能,立即将这一页显示出来.
06	页面向上滚动
07	页面向下滚动. 06和07服务程序使屏幕在一个特定区域内进行滚动,因此可以进行窗口的管理. 再者,可以通过将 AL 设置为 0,对整个窗口或屏幕进行清除 (且可同时装入特定的属性). 对于CGA和EGA,只影响当前有效显示页的滚动.
08	在光标处读字符和属性. 这一功能允许字符直接从屏幕读出,无论是工作于正文或图形方式(在正文方式时加入属性).
09	在光标处写字符和属性. 见图3-2的例子. 注意. 即使这一功能调用并不推进光标, 当众多相同的字符重复显示时,(CX > 1),这些字符在屏幕上被顺序地显示出来,并停在一行的结尾.

10　　只在光标处写字符．服务程序10与09不一样，它不修改屏幕上写有字符的点的现有属性．

11　　设置彩色色调．用于彩色图形方式．

12　　写点阵

13　　读点阵

14　　写电信打印字符．与09号功能不同,这一功能具有(a)自动地移动每个显示字符的光标；(b)正确地响应ASCII的07h(响铃),08h(退格),0Ah(换行)和0Dh(回车);(c)当显示到最后一列时自动换行,当显示完最后一行时自动滚动．然而,字符属性或彩色却不能用此项功能调用进行设置．(但如果先用09号功能写入一个带正确属性的空格,用这一功能写入的后继字符也将携带这一属性．)

15　　得到视频方式、显示的列数和有效显示页．

16　　只适用于EGA和IBM-PCjr:置彩色调色寄存器．(见附录G引用的EGA手册中有关这部分的详细说明以及EGA的下列功能．)

17　　只适用于EGA,字符生成程序．

18　　只适用于EGA,"交替选择"功能．如果BL＝10h,这一功能选择EGA参数．如果BL＝20h,这一功能选择交替打印屏幕程序(已知EGA屏幕尺寸是可变的)．

19　　只适用于AT和EGA,写字符串．虽然只可用于AT和EGA适配器,但它有一个优点,即将视频服务程序的09,10和14号功能联合起来了,这样只用一次单独的调用即可完成一个完整的字符串的写入．可以设置显示属性,CR．LF,退格(08)和响铃(07)都按命令而不作为可打印字符来对待．

```
/*
    File:    PRINTA.C

    A C Function that uses a BIOS interrupt to display a string
*/

void printa (char *, int, int, int);

void main ()
{
    printa ("red",0x4,1,1);
    printa ("white",0xf,2,1);
    printa ("blue",0x1,3,1);
}


#include <dos.h>

void printa (s, a, r, c)        /* Position cursor and write a string with */
char *s;                        /* specified color or video attribute.     */
int a, r, c;
/*
    s    : String to be displayed on screen.
    a    : Display attribute.
    r,c  : Starting row and column (0 .. 24, 0 .. 79) (ul corner = 0,0).
*/
    {
    union REGS cur_regs, write_regs;

    while (*s)                      /* Continue until null at end of string. */
        {
        cur_regs.h.ah = 2;          /* Update position of cursor using     */
        cur_regs.h.dh = r;          /* BIOS set cursor position function. */
        cur_regs.h.bh = 0;
        cur_regs.h.dl = c++;
        int86 (0x10, &cur_regs, &cur_regs);

        write_regs.h.ah = 9;        /* Write char & attribute at current */
        write_regs.h.bh = 0;        /* cursor position using BIOS.        */
        write_regs.x.cx = 1;
        write_regs.h.bl = a;
        write_regs.h.al = *s++;
        int86 (0x10, &write_regs, &write_regs);
        }

    } /* end printa */
```

图 3-1 使用 BIOS 写字符串的 C 函数

3.2.2 磁盘服务功能:中断 13h

IBM PC 在 PC-DOS 下提供了整套磁盘访问服务功能，从功能最强且与硬件无关的直到较原始或与硬件相关的都有,包括:

(1) 使用句柄和路径名的扩展文件操作服务功能。PC-DOS 版本 2.0 以上由 DOS 核心的 Int21H 功能 3CH 至 46H 提供此类服务。

(2) 与 CP／M 兼容的传统文件操作服务功能,由 DOS

核心提供,通过 Int21H 的功能 12H 至 24H,及 27H 至 29H 实现。

(3) 与设备无关的绝对磁盘读、写功能,通过 Int25H 及 Int26H,由 MS-DOS BIOS 提供。

(4) 通过 Int13H 提供的、与设备有关的对软件磁盘的绝对读、写,是 ROM BIOS 的一种服务功能。

(5) 直接访问磁盘控制器芯片的软磁盘 I/O。许多反拷贝软件使用此方法来读非标准的扇区布置。

使用 Int13H 是可移植性与硬件相关性之间的一种折衷方案。虽然在 PC 系列所有型号间均可使用它,但可能无法移植至其他运行 MS-DOS 的 8086/8088 微机上。

调用序列

通过 ROM BIOS Int13H 访问软磁盘总共可有六个不同功能可用。一般的调用序列是:

```
mov   ah,function          ;AH包含有功能代码
  :                        ;与功能有关的值装入其他寄存器
int13h                     ;传送给ROM驱动程序
```

段寄存器及 BX,CX,DX,SI,DI,BP 寄存器均保留。寄存器 AX 用于返回结果或状态。

中断 13h 提供的是一系列在扇区水平上操作的磁盘控制功能。注意,原始的 ROM BIOS 所提供的功能只适用于 5.25 英寸软盘驱动器。然而,象以前提到的,如果安装了一个硬盘控制器,中断 13h 也许立即转向装在段地址 C800h 处的程序(装在控制卡的 ROM 里)。一个硬盘控制器 BIOS 提供了与标准 ROM BIOS 的中断 13h 相同的基本功能。然而在使用时,有许多不同之处。而 15 个附加功能也是一样。

象以前提到的一样,这些服务程序可以被 MS-DOS 所

提供的一个很精巧的逻辑源文件所旁路。这些旁路平时是很少用到的,除非是在已有拷贝保护的磁盘上进行写操作,进行磁盘的增效修改或者其他不法活动发生的时候会用到。

如果磁盘服务程序得到一个出错报告,首先要使用 00 号功能复位磁盘系统。磁盘马达从起动到全速运转需要一定的时间,由此而引起的错误,要按规定重做 3 次。下表概括了 BIOS 中断 13h 的磁盘功能。

功能	服务程序的执行
00	复位磁盘系统。这一功能必须在出错后调用。下面的磁盘操作将从读/写头的校准开始。
01	读最后一次磁盘操作后的状态。即使是使用DOS进行磁盘I/O控制,万一出现错误,调用此功能仍然可提供极为详细的错误信息,这也许正是一个危险错误处理程序要用到的(见第六章)。
02	读扇区。
03	写扇区。
04	扇区检验。通过计算CRC(周期冗余检验),检查给定扇区里数据的完整性。
05	格式化磁道。
06-20	在这一范围里,有15个由硬盘控制器提供的附加服务程序。

3.2.3 串行端口服务功能:中断 14h

通过 Int14H 提供的四种功能可访问串行通信端口控制器。一般的调用序列为:

```
mov ah, 功能                ;AH 包含功能类型
mov  dx,portnumber          ;DX 选择通信端口
                            ;在其他寄存器中装入与
                            功能有关的值
```

```
int14h                    ;调用 ROM BIOS
```

段寄存器及寄存器 BX,CX,DX,SI,DI,BP 保留。寄存器 AX 用于返回结果或状态。

注意,Int14H 选用的通信端口号从 0 开始,而在 MS-DOS 级上是从 1 开始编号的(COM1,COM2…)。

Int14H 的程序负责管理通讯接口的比特流的输入/输出。然而,这些程序严格限制了高性能通讯的运用。通常,一个通讯服务程序必须能作到在同一时间里进行:读到达串行口的字符,并在屏幕上显示出来;从键盘读入字符,显示在屏幕上并将它们送出串行口。

如果使用了 BIOS 服务程序,程序必须在执行其他任务的同时,定期地打断其进程来检查串行口是否有数据要接收。由于字符是异步到达串行口的,所以极有可能丢失字符。解决的办法是旁路 BIOS,另写一个中断驱动服务程序,使其具备以下功能:

- 当一个字符到达串行口时,产生一个硬中断。
- 中断处理程序直接从输入口读入字符或保存在一个缓存区里(这很象 BIOS 的键盘中断处理程序)。
- 当主程序需要一个字符时,就从缓冲区里取出来。

编写的方法见下一章写硬中断处理程序的有关内容。

下表概括了 BIOS 的中断 14h 的串行口功能:

功能	服务程序的执行
00	通讯口初始化
01	向通讯口写字符
02	当字符准备好时读入
03	返回通讯口的状态

3.2.4 盒式磁带服务功能:中断 15h

下表概括了 BIOS 的中断 15h 的盒带服务程序的功能:

功能	服务程序的执行
00	启动马达
01	关闭马达
02	磁带块读入
03	磁带块写

3.2.5 AT机上的扩展服务功能:中断 15h

当使用与调用盒式磁带服务程序相同的中断号进行调用时,AT 机可有 12 个扩展的服务程序,以适应特殊的硬件特性. 详见 AT 机技术参考手册.

3.2.6 键盘服务功能:中断 16h

键盘服务程序也许是 BIOS 中第二位有用的功能块. 它们既简单,功能又很强. 如果使用了 BIOS 的视频服务程序,没有理由不把它们卷入.

举一个例子,图 3–3 中的 getkey 过程使用了 BIOS 的 0 号功能块,从键盘读入一个键. 0 号功能是接受键盘数据的极好方式. 它具有一些有用的特性:

- 当一个键没有按下时,0 号功能可以等待其输入. (然而,要事先调用 01 号功能来检查是否有字符已准备好.)
- 当一个键按下后,此功能立即返回. 因此,用户不用打回车键,而让程序继续执行. 按键后就无需再理会了.
- 此功能可以返回一个键的 ASCII 值. 如果返回的是 0,则无 ASCII 值(例如功能键).
- 此功能可以返回键的扩展码. 每一个键都分配了一个扩展码(例如组合键 Alt–F1). 所以,可以定义无 ASCII 值的键(即返回一个 ASCII 为 0). 注意,与 DOS 服务程序不同,一

个单独的功能调用可返回 ASCII 值和扩展码.

扩展码与扫描码不同. IBM-PC 标准键盘的 83 个物理键的每一个键都有一个唯一的扫描码, 从 1 到 83(在键按下时, 由硬件给出这个值). 如果一个键有一个 ASCII 值, 则返回的扩展码与扫描码相等, 但是, 对于组合键(如 shift-F1),扩展码就不是扫描码,而是比 83 大的一些数值. 所以,不同键的组合可以用一个单独的数值加以区分.

例如,F1 键没有 ASCII 值,它的扫描码为 59. 各种组合键所返回的扩展码为:

键	扩展码
F1	59
Ctrl-F1	94
shift-F1	84
Alt-F1	104

注意,具有 ASCII 值的键通常返回一个与扫描码相等的扩展码. 例如,不论打 a , A或 Ctrl-A, 都会产生一个扩展码 30,即这个键的扫描码. 所以要区分任意一个键的正确过程是首先检查返回的 ASCII 码. 如果不是 0,则在 ASCII 值的基础上处理这一键;如果是 0,则检查扩展码, 以决定是哪一个键或是组合键被按下. PC 技术参考手册(在 BIOS 章节里)和 BASIC 参考手册的附录,列出了所有键及各种组合键的 ASCII 码和扩展码.

图 3-2给出了 0 号功能的一些基本应用. BIOS 返回的 ASCII 值存放在 AL 寄存器中,扩展码放在 AH 寄存器中. getkey 的作用是将两个寄存器中的值变为一个整数. 因此,在进行单一数据的检验时, C 程序必须进行移位和屏蔽操

作．main 的作用,首先是为 getkey 提供了两项基本应用:建立一个暂停和读人功能键．第三个作用是简单地将已按下键的返回值打印出来．如果对键编码有疑问,这个检查程序可给出明确的答案。

01 号功能块检查一个键是否已准备好,以将其读入(例如在 BIOS 键盘缓冲区里),并立即返回调用程序．这种功能可以消除等待从键盘输入所需的空闲时间,提高了程序的效率(例如,00号功能要等输入了一个键以后才返回)．相比之下,许多有用的工作可以在等待的时间里完成．用伪码表示的这一过程就是:

```
report
      call function 01 to see character is ready
      if character is ready then
            call function 00 to read oharacter
                                        process character
        else
          perform arother task (eg send characters
                                              to printer)
        until(ending condition)
```

下表概括了 BIOS 的中断 16h 的键盘处理功能。

功能	服务程序的执行
00	读字符
01	检查字符是否准备好、不间断
02	返回shift状态。

3.2.7 打印服务功能:中断 17h

中断 17h 为打印机提供了一些基本的功能调用．02 号功能是一个很有用的功能块．它返回打印机的当前状态．图

```
/*
    File:    GETKEY.C

    A C function that uses a BIOS interrupt to read the keyboard
*/

main ()
    {
    int ch;

    /* demo 1 */
    printf ("press any key to continue ...");      /* Use unbuffered input */
    getkey ();                                     /* to generate pause.   */


    /* demo 2 */
    printf ("\n\nenter another key ...");
    ch = getkey ();
    if ((ch & 0x00ff) == 0)                /* Mask off upper byte to look at  */
                                           /* ASCII code.  If ASCII code = 0, */
                                           /* must look at extended code.     */
         switch ((ch & 0xff00) >> 8)       /* Branch on extended code.        */
              {
              case 59 : {
                        printf ("\nF1 pressed");
                        break;
                        }
              case 60 : {
                        printf ("\nF2 pressed");
                        break;
                        }
              /* etc., etc.  */
              }

    /* demo 3 */
    printf ("\n\nenter a series of keys to see codes, <esc> ends");
    printf ("\n\nASCII code       extended code");
    do
         {
         ch = getkey ();
         printf ("\n%5d%16d", (ch & 0x00ff),
                (ch & 0xff00) >> 8);
         }
    while ((ch & 0x00ff) != 0x1b);                          /* Test for <esc>. */

    } /* end main */

#include <dos.h>

int getkey ()
/*

    This function waits for a key to be entered.  Input is unbuffered, so
    that the function returns immediately after the key is pressed (the
    user does not have to press (R).  Returns an integer containing:
         low order byte:   the ASCII key code (if available)
         high order byte:  the "extended" key code
*/
    {
    union REGS reg;

    reg.h.ah = 0;
    int86 (0x16, &reg, &reg);
    return (reg.x.ax);

    }  /* end getkey */
```

图 3-2 函数 getkey,使用 BIOS 键盘服务中断(16h)从键盘上读人一个键

3-3中,prnread这一C程序的作用,就是利用该功能调用来确定联接打印机时,是否存在I/O错误。当发现向一个不是连接状态的打印机输出数据时,这一程序就产生一个不优雅的程序结束("Abort,Retry,Ignorc?")。

```
/*
     File:     PRNREADY.C

     A C function using BIOS printer services interrupt 17h to read the
     status of the printer
*/

main ()
     {
     while (!prnready() )
          {
          printf ("\nready printer and press any key to continue ...");
          getkey ();                                          /* Pause. */
          }
     printf ("\nprinter is now online");

     } /* end main */


#include <dos.h>
union REGS reg;

int prnready ()
/*
     This function returns false if there is an i/o error associated with
     LPT1, and true otherwise.
*/
     {
     reg.h.ah = 2;                            /* Printer status function. */
     reg.x.dx = 0;                            /* Printer 0 is LPT1.       */
     int86 (0x17, &reg, &reg);
     return (! (reg.h.ah & 8));               /* Mask i/o error bit, #3.  */

     } /* end prnready */


int getkey ()
     {
     reg.h.ah = 0;
     int86 (0x16, &reg, &reg);
     return (reg.x.ax);

     } /* end getkey */
```

图3—3 使用BIOS读打印机状态的C函数

下表概括了BIOS中断17h打印机服务程序的功能。

功能	服务程序的执行
00	打印字符。

01	打印机初始化．这一功能调用可用于清除打印机当前的控制码，并使打印机回到初始状态．得到打印机状态．

3.2.8 时间和日期服务功能:中断 1Ah

这些服务程序允许读出或修改保存在 BIOS 中的时间记录．下表概括了 BIOS 中断 1Ah 功能调用．

功能	服务程序的执行
00	读时钟．
01	置时钟．00和01号服务程序报告的是从午夜发生的"嘀嗒声"(每秒钟发出 18.2 次)．要得到实际的小时数、分钟数和秒数，还需要进行计算．
02	AT机专用．读实际时间时钟与00和01号服务程序不同，02 和 03 号程序直接用小时数、分钟数和秒数设置和取得时间．
04	AT机专用．读数据．
05	AT机专用．置数据．
06	AT机专用．置警报．
07	AT机专用．警报复位．

3.2.9 其他服务功能

3.2.9.1 中断 05h:屏幕打印服务功能

按下 Shift-Prtsc 键便执行此程序，应用程序中只有通过中断 05h 来实现该功能．

3.2.9.2 中断 11h:设备列表服务功能

进入:

无．

返回:

AX 在 PC 机中，此寄存器的比特表示所安装的设备如下

(AT 的编号有微小差别,详见 AT 技术参考手册):

比特	意义
15,14	所连打印机数
13	未用
12	所连接游戏 I/O 数
11,10,9	RS232 适配器数
8	未用
7,6	磁盘驱动器数. 下列代码仅当比 特 0 的值是 1 时给出磁盘驱动器 数 编码　驱动器数 00　1 01　2 10　3 11　4
5,4	初始的视频显示方式 值　方式 00　未用 01　40×25　CGH 卡 10　80×25　CGA 卡 11　80×24 单显卡
3,2	母板上 RAM 编码　RAM 量 00　16K 01　32K

	10	48K
	11	64K
1	未用	
0	如果比特=1表示存在一个以上的软盘驱动器(实际个数见比特7,6)	

此功能返回的设备标志字存储在地址 0000h:0410h 处，由 BIOS 根据诊断结果在系统起动时设置。

3.2.9.3 中断 12h:内存测试服务功能

进入:

无。

返回:

AX 连续的 1KB内存块数。

此功能根据母板上开关设置获得有关 PC/XT 内存量与存储在地址 0000h: 0413h 处的内存量值相同。

3.2.9.4 中断 18h:装入 ROM BASIC

进入:

无。

返回:

无。

此程序激活包含在 ROM 中的 BASIC,并可用于覆盖磁盘 BASIC 或 BASICA。

3.2.9.5 中断 19h:自举装入程序

此程序使系统重新启动,但越过计算机加电冷启动过程中执行的冗长的诊断，并且当按下Ctrl-Alt-Del 时不再重置存储在段 0400h 中的系统数据。

第四章　MS-DOS 机器的其他资源

除了可通过中断机构获得的大量子程序外,MS-DOS 机器中还有许多其他资源可用于加强软件性能. 这些资源由硬件、固件(BIOS)和软件(MS-DOS)提供,可用于:

- 获得有关系统配置的信息
- 与其他应用程序通信
- 修改 DOS 和 BIOS 的功能
- 绕过 DOS 和 BIOS 去加强性能
- 使用硬件信号去同步子程序
- 为外部设备开发用户接口

本章介绍的部分资源可广泛应用于 MS-DOS 机器,另一些资源是与机器相关的,只能用于 IBM PC AT 或高兼容性的机器.

本章所讨论的资源包括:程序段前缀、低内存地址区存贮的系统数据、Ctrl-Break 和致命错误中断处理程序以及硬件产生的中断等.

4.1 程序段前缀(PSP)

MS-DOS 装入程序(Loader)为一程序的执行作准备时,在内存中装入程序之前建立一个100h的数据块.此数据块称为程序段前缀(PSP),其组成部分为:

- DOS 用于管理进程所贮存的参数
- 装入程序使用时要求 DOS 提供的信息
- 管理文件输入/输出及交换文件数据的区域

下面给出 PSP 的总结构。

偏移量	长度	意义
00h	2	int20h 指令（结束程序）
02h	2	分配内存顶部的段地址
04h	1	DOS 保留
05h	5	对DOS功能调度程序的长调用指令
06h	2	段中字节数（此域用于两个目的，它亦是5 字节对 DOS 长调用的偏移量地址部分）
0Ah	4	结束地址（向量 22h 存贮的数值）
0Eh	4	Ctrl-Break地址(向量23h存贮的数值)
12h	4	致命错误地址（向量24h存贮的数值）
16h	2	DOS 保留
18h	20	DOS 文件表(未写入文档)
2Ch	2	环境段地址
2Eh	34	DOS 保留
50h	3	Int21h,retf指令(一种调用DOS功能的不推荐的方法)
53h	2	DOS 保留
55h	7	第一扩展 FCB 区域
5Ch	9	第一 FCB
65h	7	第二扩展 FCB 区域
6Ch	20	第二 FCB
80h	1	命令行长度，也是128字节的默认磁盘传送区(DTA) 起点
81h	127	命令行(以程序名后第一个空格开始)

后面将详细介绍对程序员较为有用的几个域。

4.1.1 从汇编程序中访问 PSP

从汇编程序中访问 PSP 的方法取决于该程序是写成了.COM 文件还是.EXE 文件.

▲.COM 文件

访问.COM 文件的 PSP 较为简单. 操作系统装入一个.COM 文件后,所有段寄存器均指向 PSP 的起点. 由于.COM 程序一般不改变这些寄存器的值,故可用简单的数据参考指令访问 PSP 中的域.例如下面的指令是将PSP中偏移量 80h 处的一字节域移入寄存器 CL:

```
mov  cl,  ds:[80h]
```

因为 DS 是数据基址指令的默认段寄存器,故从理论上说不必指定 DS 段. 但如果不写此段地址,汇编程序将会把 80h 作为一个立即值而不是指针来解释了.

一个.COM 文件只含有可执行代码,并且是该程序内存中的准确映像.在代码段的开头,需要被访问的PSP的每个域都由一个指出偏移量的 org 语句、给出的一个名字、数据类型和域本身大小的 db 语句来表示. 例如,偏移量 80h 包含命令行长度,表示为:

```
org  80
parmlen  db  ?      ; 单字节命令行长度
```

欲将此值装入寄存器CL, 只需使用

```
mov  cl,  parmlen
```

这些 db 指令看起来与普通的变量定义一样,但实际上并无助于赋给初始值,因为在进入点偏移量 100H 之前,所有一切已在该文件被转换成一个.COM 格式文件前被 EXE2BWIN"砍"掉了. 这些定义只是作为一个临时的样板,以使汇编程序能够生成正确的偏移量数值,因而不会增加

.COM文件在磁盘上或内存中的大小.

▲.EXE 文件

从.EXE 文件中访问 PSP 与从.COM 文件中进行访问主要有两点不同:

首先是程序装入时,只有寄存器 DS 和 ES 指向 PSP,而这些段寄存器通常在程序开始时被重新初始化以指向数据段. 因此 PSP 的段地址应保存在某个内存变量中以备将来使用. 例如可使用下面的代码:

```
mov  ax,dataseg
mov  ds,ax
mov  psp,es  ;存入内存变量 psp 中
mov  es.ax
  :
psp dw?    ;psp 变量在数据段中定义
```

如果在程序运行中要将程序段前缀偏移量 80h 处的一个字节装入寄存器 CL,使用下面的两条指令:

```
mov  cs,psp
mov  cl,cs:[80h]
```

第二点不同是EXE 文件代码段开头的 PSP 偏移量用来定义指令即将插入可用的代码空间,且不必扩展文件大小. 因此最好使用立即值表示的偏移量(如上例中的 80h). 如果使用立即数,则当汇编程序无其他语句涉及到该操作数大小时,可能需增加一条 ptr 指令,例如:

```
mov cs,psp
cmp byte ptr es:[80h],127     ;比较一个字节
```

4.1.2 从 C 语言中访问 PSP

Microsoft 提供了一个很方便的全局变量 _PSP,它包含

程序段前缀的段地址．此变量是在文件 STDLIB.H 中作为 unsigned int 声明的．下面的代码说明了如何使用此变量去访问包含命令行参数长度的偏移量 80h 处的单字节域:

```
#include<stdlib.h>        /*用于 PSP*/
#include<dos.h>           /*用于 FP_OFF 及 FP_SEG*/
  :
unsigneg char far * plptr    /*单个字节指针*/
FP OFF(plptr)=0×80       /*PSP 中偏移量*/
FP SEG(plptr)= PSP       /*PSP 的段值*/
Printf("参数行长度=%d/n", * plptr);
```

此例说明了下面几个重要点:

• 该值是通过一个远程指针访问的,因为它处在默认的数据段之外．

• 此情况下,该远程指针指向一个 unsigned char,因为打算读的是一个字节．

• 预先定义了的宏 FP_OFF 和 FP_SEG 用于将偏移量值和段值赋给此远程指针(该远程指针长 4 个字节)．

• PSP 域的实际值从 * plptr 中推演出．

最后要提及的是,如果使用的是 DOS3.0 或更高的版本,功能调用 62H 将 PSP 的段地址返回给当前执行的进程．

4.1.3 重要的 PSP 域

本节讨论程序段前缀各域中对于程序员来说特别重要的一个子集的结构和使用:内存顶部地址、段大小、DOS 文件表、环境地址、文件控制块、命令行以及默认的磁盘传送地址．

(1) 偏移量 02h:内存顶部

此域内容是当前已分配给程序的内存块顶部的地址。该段偏移量 0 处的字节实际上便是未分配内存的第一个字节。注意,此域不一定包含"用户内存"的顶部地址(所谓用户内存指的是系统所安装的内存总量).此域的重要性取决于该程序是一个.COM 文件还是一个.EXE 文件。

如果是.COM文件,在它装入时分配掉所有可能的内存,此时"内存顶部"域实际上包含的是用户内存顶部的地址(除非该程序有意的释放过内存)。从而在此地址之上已不能再获得内存。

.EXE文件不一定分配完所有的内存,故可使用 DOS 功能 48H(内存分配)获得内存顶部地址上面的额外内存。初始分配给.EXE 文件的内存量取决于文件头最大段落(maximum paragraphs)域中的数值。修改此值有两种方法,一种是在连接时使用/CP标志,另种是使用带 /MAX 选择的 Microsoft EXEMOD 实用程序修改现存的.EXE 文件。

BIOS 的中断 12h 返回用户内存的总 K 数(根据 IBM PC 的 DIP 开关设置)。但要注意,有些 RAM 磁盘等实用程序将自身装入高内存区,这些程序调低了由 DOS 管理的用户内存地址的一端,但并不影响 BIOS 返回的值,故此可用内存值大了一些。下面提供一种一般性的方法来可靠地确定.COM 文件或.EXE 文件已分配了的及未分配的总的可用内存量:

· 检查 PSP 偏移量 02 处的域以确定已分配给该程序的内存量。由于.COM 文件无更多内存可供使用,故到此为止。

· 对于.EXE 文件,可由 DOS 中断 21h 功能 48h(分配内

存)来确定可供分配的附加内存,调用时在寄存器 BX 中装入 FFFFh. 该功能调用返回时,进位标志置位(因为 FFFFh 的内存请求是无法满足的), 寄存器 BX 包含最大的可用内存块 (以段落——16 字节计).

(2) 偏移量 06h:段大小

对于.COM 文件,此域包含该程序段的可用字节数,通常此值相当接近于64K(最大的段大小);但如果没有足够的自由内存,此值可能会小些. 因此用到有关该程序段可用内存的知识时最好检查一下此域.

(3) 偏移量 18h-2Bh:DOS 文件表

PSP 的这个 20 字节的区域本来是保留的. 故下面的信息未进文档,因而在操作系统的将来版本中有可能会发生变化. 使用时应小心。

单个进程可打开的文件句柄数通常限制为 20(假定在配置文件 Config.sys 中,FILES=行上给出的数值大于、等于 20). 但由直接控制 PSP 中的 DOS 文件表实际上可同时打开多达 255 个句柄.

① DOS 文件表的结构和使用

DOS 对于一条"打开"(open)命令返回的文件句柄为 0 到 19 之间的一个数,并在 PSP 偏移量 18h 处的 20 字节文件表中作一标引. 该表中每个单字节的登记项如果包含的是 FFh,说明此表登记项未使用;如果数值为 0 到 FEh,则为 DOS 内部使用的表示引用一个实际文件或设备的数码. 此数码被称为"DOS 内部数码"以区别于文件句柄. 一个文件句柄传送给 DOS 后,DOS 并不使用该文件句柄去标识实际的文件,而使用的是从文件表中得到的内部 DOS 数码. 内部 DOS 数码具有以下数值:

内部 DOS 数码	文件/设备
00	AUX
01	CON
02	PRN
03 及以上	由程序打开的文件/设备

一个程序装人时,该文件表具有下面的初始值:

偏移量	句柄名称	值
0	标准输入	01
1	标准输出	01
2	标准错误	01
3	标准辅助设备	00
4	标准打印机设备	02
5 及以上	自由句柄	FF

前五个登记项对应于标准的预先打开的文件句柄。应注意,标准输入、标准输出及标准错误全都指的是控制台设备。

此方法的优点是,DOS 可由简单地改变该文件表中的对应登记项来重定向一个文件句柄。例如,偏移量 1 处的表登记项(对应于标准输出文件句柄),通常包含用于控制台(01)的内部 DOS 数码,如果操作系统由一个关于磁盘文件的内部 DOS 数码替换此登记项,则后面所有对标准输出的写操作均写至该磁盘文件,

默认情况下,DOS 为 8 个内部文件号保留空间。但如果

配置文件中含有 FILES=nn, 则 DOS 为给出的文件数保留空间， 此数可大至 255. 如果 Config.sys 中包含 FILES=255, DOS 内部将为 255 个打开文件保留空间,编号从 0 到 FEh, 20 个文件的限制起因于 PSP 中的句柄表只有 20 个字节长,并且当没有更多的自由登记项(即标记为 FFh 的登记项)可用时,DOS 无法进一步打开文件,将返回出错代码 4.

② 突破系统限制

图4-1的C程序清单说明了如何同时打开31个文件,可用同样的方法打破 DOS只有20 个句柄的限制,可同时打开多达 255 个文件. 基本技术依据不是由内部 DOS 数码, 而是由文件句柄来引用一个文件. 其步骤如下:

(A)在 CONFIG.SYS 文件中写上 FILES=31(欲打开更多的文件就写入更大的数,限制是 255),然后重新启动系统.

(B)变量 ftptr 作为一个远程指针声明,初始化时指向句柄 5 的文件表登记项. 由此完整保留了五个预打开的设备句柄,以后的所有文件将都使用文件句柄号 5 来打开和关闭.

(C)数组 dosnum 将为欲被打开的 26 个新文件保存 26 个内部 DOS 数码.

(D)每个文件按如下顺序打开:

(a)为句柄 5 的文件表登记项赋给 FFh 以标记它是可用的. 句柄不再保存,因已经知道此号码将是 5(DOS 使用的第一个可供使用的句柄).

(b)内部 DOS 数码保存在 dosnum 数组中,以后被用于访问该文件.

(E)每个文件的关闭使用下列方法,也说明了应如何引用文件.

(a)句柄 5 的文件表登记项被赋给关闭文件的内部 DOS 号。

(b)使用 close 函数关闭文件句柄 5。

```
/*
     File:     FILES31.C

     A C demonstration that uses handle functions and has 31 files open at
     the same time:  the 5 standard handles plus 26 new files named
     'FILEA' ... 'FILEZ'
*/

#include <stdlib.h>
#include <dos.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>

main ()
     {
     unsigned char far *ftptr;          /* Far pointer to file table entry.  */
     unsigned char dosnum[26];          /* Storage for internal DOS numbers. */
     static char filename[6] = "testx";
     int i;

     FP_SEG (ftptr) = _psp;             /* Beginning of the PSP.             */
     FP_OFF (ftptr) = 0x1d;             /* Entry in file table for handle 5; */
                                        /* leave 5 standard handles alone.   */

      for (i=0;i<26;++i)                   /* Open 26 new files.             */
          {
          *ftptr = 0xff;                   /* Mark handle 5 free.            */
          filename[4] = 'a'+i;             /* Create unique file name.       */
          creat (filename, S_IWRITE);      /* Open file, don't save handle   */
                                           /* since we know the handle is 5. */
          dosnum[i] = *ftptr;              /* Save the internal DOS number.  */
          }

                                           /* 31 files are now open!         */

      for (i=0;i<26;++i)                   /* Close the 26 new files.        */
          {
          *ftptr = dosnum[i];              /* Replace internal DOS number.   */
          close (5);                       /* All files use handle 5.        */
          }

      } /* end main */
```

图 4-1 使用文件句柄功能但可同时打开 31 个文件的 C 程序

(4)偏移量 2Ch:环境段地址

环境(environment)是内存中的一个区域,主要用于存贮由 DOS 的 PATH, PROMPT, SET 命令设置的参数。它是一系列以0结束的字符串(称为ASCIIZ字符串——其最后一个字节是二进制0,而不是0字符的ASCII代码),名字字符串

的格式为:

名字=参数

全部字符串集合完后再以另外一个0字节结束. 注意,对于DOS3.X在结尾0后面是下列诸域:

· 一个字(word)作为后面字符串的计数.

· 一个包含该程序全部路径名的ASCIIZ字符串,故可找到该程序被装入时使用的名字和目录. 在C语言中,对于DOS版本3.X,argv[0]包含本程序名,但对于更早的DOS版本,argv[0]中仅包含字符C.

· 可能存在的该程序可用的附加字符串.

每个进程均在其PSP前面的内存块中有自己的环境备份.环境从一个进程拷贝到另一个进程的方式如下: 命令处理程序维护自己的环境备份, 该环境包含一个表明COMMAND.COM位置的登记项,例如,COMSPEC=C:\加上其他由PATH,PROMPT及SET命令产生的附加登记项. 一个程序装入时,它接收命令处理程序之环境的备份. 该程序可以修改此环境备份,但并不影响由命令处理程序维护的环境,也不会传送给其他程序. 如果该程序调用了一个子进程,可将自己的环境传送给该进程,也可以传送另一个完全不同的环境. 故可以说,环境是一个从父进程到子进程通信的单向通道. 内存驻留进程将维持其环境备份,但后面即使改变了DOS的环境拷贝也无法更新此区域.

程序段前缀偏移量2Ch处的域包含属于该程序的两字节的环境备份段地址,第一个字节处于此段的偏移量0处. 汇编程序使用此地址去访问环境与访问PSP的方法类似,但要求间接进行. 例如下面的代码将环境的第一个字节送入寄存器AL:

```
mov  es.psp            ;保存 PSP 段地址
mov  ax,es:[2ch]       ;将环境段地址送入 AX
mov  es,ax  9          ;现在 ES 中有了环境段地址
mov  al,es:[0]         ;访问环境的第一个字节
```

C 程序可通过预先已定义了的_PSP 变量来访问环境，要使用远程指针以及若干层间接寻址。Microsoft C 提供了另外三种方法:

① 库函数 getenv 可从环境中检索出字符串,函数 putenv 将字符串加入到环境中。注意,putenv 可能引起环境被移至另外的地址。

② C 维护着一个预定义了的变量 environ,该变量总是指向环境(即使环境被移动了),并可被用于访问环境字符串。此变量是对于组成了该程序环境的字符串的一个指针数组，以 char * environ[]说明。

③ 除了 argc,argv 外,还有第三个参数 envp 被传送给 main 函数。此参数也是一个对于参数表的指针(char * envp[]),但与 environ 不同,如果环境被移动后,它并不自动更新,因此使用 environ 更安全些。

(5) 偏移量 5Ch 和 6Ch:文件控制块

如果命令行上给出前两个参数是程序运行时的合法文件名(不包含目录路径),它们将出现在偏移量 5Ch 和 6Ch 处,并处理成下面所列标准格式:

位置	内容
字节 0	驱动器代码(0＝默认,1＝A:,2＝B:,等)
字节 1–8	文件名
字节 9–11	文件扩展名

程序员如何才能确定这些域是否含有合法的文件名呢?

如果第一个参数是一个合法文件名,则当该程序开始时,寄存器 AL 将被设置成 00h(FFh 表示是一个非法名)。如果第二个参数是一个合法文件名,则寄存器 AH 被设置为 00H(否则是 FFh)。程序在打开文件控制块时 DOS 提供此服务,此法的主要限制是不能支持目录。既使不使用文件控制块 FCB 方式的程序,也可以用上这些区域。

另外,偏移量 5Ch 标志 PSP 中非重要部分的开始。PSP 中从 5Ch 到结尾,DOS 在执行正常工作时都用不着,因此这些区域可以安全地用于其他目的。

(6) 偏移量 80h:命令行长度/默认磁盘传送区

偏移量 80h 处的一字节域开始时包含命令行长度,而命令行存储在偏移量 81h 后。

此区域有二重目的。它亦是默认磁盘传送区(DTA)的开始,因此在调用任何使用到 DTA 的子程序之前应先保存命令行长度和命令行。磁盘传送区 DTA 是文件控制块命令,用于在文件之间交换数据的缓冲区。默认 DTA 是 PSP 中的一个 128 字节的缓冲区,在偏移量 80h 处开始。对于使用句柄功能调用的用户,此区域用处不大,因为句柄功能调用的输入/输出发生在由程序员指定的缓冲区中,通常是在数据段中。但有一个例外,句柄功能调用 4Eh 和 4Fh 在搜寻匹配文件时使用 DTA 去存贮和返回信息。下面程序代码说明了功能调用 4Eh 和 4Fh 使用默认 DTA 作文件搜寻。

(7) 偏移量 81h:命令行

偏移量 81h 处的域包含的是无格式的命令行拷贝,以程序名后打入的第一个空格开始。此区域最多可包含 127 个字节(延至 PSP 结尾处),且为该命令行的严格拷贝,只有一个例外,即重定向符(<,>)、管道符(|)以及与其相关的文件名均

被命令处理程序删除了. 由于重定向操作和管道操作都是由操作系统进行的,而不是由应用程序进行的,故此信息对于程序是透明的。

4.2 低内存地址的数据区

低地址 0000:0400h 至 0000:05FFh 的内存块保留作为通信区,BIOS,DOS 及用户的应用程序使用此区域中的特定域以存贮参数、标志及其他数据. 程序员可直接访问这些区域以获得有关系统状态的内部信息,修改系统的性能以及与其他应用程序交换信息.

下面(1)到(4)的地址以段 0000h 中的十六进制偏移量给出. 先看一个小例子来说明如何访问这些特定内存区域中的一个, 此例将偏移量 500h 处的值装入寄存器 AL:

```
xor  ax,a              ;快速清 AX
mov  cs,ax             ;段=0000h
mov  ax,es:[500h]      ;AX 获得 0000h:0500h 处的值
```

(1) 4000h–4ABh:BIOS 通信区

BIOS 使用偏移量 400h 至 4ABh 处的内存区域存贮众多的参数和状态标志. 这些域在 PC/AT 技术参考手册的汇编语言清单开头给出了定义. 这些值中的大多数可通过更好的 BIOS 功能调用方法来操作, 但也有一些需使用直接的内存访问做读、写. 下面给出此区域的基本结构,重点介绍对程序员有实际意义的若干域(偏移量均以十六进制计):

偏移量	内容
400–407	RS–232 适配器地址
408–40F	打印机地址
410–411	设备标志. 此处的值与中断11h所返回的值一样. 这些值通常仅在启动时设置. 但如果程序切换显示器适

配器(或在多显示适配器卡上切换方式),则必须如下正确地设置该字的比特 5:4。

00:EGA

01:CGA 卡 40 列

10:CAA 卡 80 列

11:单显卡

412	初始化标志
413—414	内存大小，由中断 12h 返回。
415—416	I/O 通道中的内存量。
417—43D	键盘数据。此区域包含键盘状态标志及一个15字节的键盘缓冲区。通常是通过中断 16h 来执行键盘 I/O,以及确定键盘状态的。但有些特殊的实用程序,例如某些内存驻留程序或是用于加快键盘重复输入率(某键按下后重复的速率)的实用程序，将绕过 BIOS,而对此缓冲区作直接读、写。
43E—448	软磁盘数据。
449—466	显示器参数。通过中断10h对这些数值进行读出或设置更为方便。
467—46B	磁带机数据。
46C—470	时间数据。BIOS 的时钟信息通过中断 1Ah 来管理。
471	Break 标志。如果按下 Break 键,则此标志置为 1。
472—473	重置标志。如果此字标志设置成1234,则在段FFFFh处开始的 BIOS 初始化子程序将跳过通常冷启动时执行的冗长的内存检测。下面代码给出了一种产生快速热启动的简单方法:

```
bios    segment at offffh
reboot  label far
```

```
bios    ends
:
xov ax,ax
mov es,ax
mov word ptr des:[472h],1234h
jmp reboot
```

474–477　　固定盘数据区

418--47F　　用于IBM PC j r 的打印机和串行端口的超时计数器.

480–483　　额外的键盘数据区。键盘缓冲区起始和停止偏移量的Word 数值。

484–4AB　　仅用于EGA系统的附加显示器参数.

(2) 4ACh–4EFh:保留

此区域为 BIOS 所保留

(3) 4F0h–4FFh:用户通信区

区域 4F0h 到 4FFh 称为"应用程序内部通信区",可由任何程序自由使用. 例如,若干独立但有关联的应用程序可通过此区域交换数据;也可以是某个程序在此存贮数据,每次运行时都使用它. 但由于任何应用程序都可向这个内存区域作写操作,故信息常常不可靠. 建立全局可用配置信息的更完全的方法是通过 DOS 的环境,但环境数据必须由用户在命令级设置并且是不可由程序赋给的(程序只可访问临时、私用的环境拷贝,由 COMMAND.COM 管理全局环境).

(4) 500h–5FFh:DOS 通信区

500h 到 5FFh 之间的部分地址是为 DOS 保留的,而另一部分是为 BASIC 所保留的. 其中较重要的一个域是:

5000　　屏幕打印状态标志

此一个字节的域用于记录屏幕打印功能的状态:

0 屏幕打印不工作/已成功结束

1 正在进行屏幕打印

255 屏幕打印过程中出错

在此域中置入 1,可使系统认为打印过程正在进行而达到撤消屏幕打印功能的目的.

4.3 硬件产生的中断

前面已介绍过 IBM PC 的中断机构分成软件中断、内部硬件中断和外部硬件中断. 外部硬件中断可作为加强程序有效性的一种重要资源.

软件中断与硬件中断有许多重要差别,其中之一是:软件中断必须由一个进程在逻辑流程的适当一点显式地调用,因此它是在调用程序控制下的一个同步事件,在这层意义上很象是一种子程序调用. 但硬件中断是由某个自主设备产生的,它是由另一个进程初始化的异步事件,也不在被中断的程序的控制下.这对于高水平编程有重要意义,即可借用硬件中断机构去生产一种有限多任务能力以加强程序效能.

例如,一个字符到达串行端口时,通信程序必须很快响应. 如果该程序不得不频繁地使用功能调用去确定是否有准备好的字符,则必然会浪费掉许多本来可用于其他任务的时间. 但如果使用一个硬件中断去通知该程序,一个字符已到达,则不会浪费时间,响应时间也最短. 硬件中断机构可作为一个并行进程,用于增加总体效能.

另一个例子是执行某种冗长内部处理,例如排序、编译、数值计算等程序, 这类程序无法对键盘快速响应, 除非频繁使用功能调用去查询键盘状态. 但如果由硬件中断来通知程序何时有键按下,在代码中就不要再使用键盘功能调用了.

最后一个例子是显示常值状态信息,例如时间或键的档位状态 (如 Caps-Lock 键等) 的程序. 使用来自时钟或键盘的硬件中断,这些应用程序可不必作频繁的功能调用即更新显示.

4.3.1 外部硬件中断

在讲述硬件中断处理程序设计之前,先介绍此类中断在 MS-DOS 机器上的几个基本特点. 外部硬件中断分为两大类:不可屏蔽的及可屏蔽的.

(1)不可屏蔽的硬件中断

只有一个中断类型 02h 是不可屏蔽的,即该中断不可由软件指令 cli 使之失效. 此中断是由处理器 NMI 脚上的信号产生的,通常用于报告内存奇偶校验错. 另外某些 debugger(检错程序)使用此中断,在程序"崩溃"及正常中断被失效时重获控制. 此种 debug 机构由两部分组成:

直接选通 NMI 的外部按钮,当按下按钮后接收控制,并继续该 debugger 工作的中断 02h 处理程序.

(2) 可屏蔽硬件中断

可屏蔽硬件中断是由处理器的 INTR 脚上的信号产生的。虽然只有一个脚,但此类中断有若干来源,故设备的中断请求是通过可编程中断控制器(8259A PIC)来安排的. 此芯片控制 8 条中断请求线,标记为 IRQ0 到 IRQ7. 按照优先级来解决请求的冲突,IRQ0 具有最高优先级,而 IRQ7 为最低优先级. IRQ0 和 IRQ1 线来源于母板,而 IRQ2 到 IRQ7 来自 I/O 通道上的适配卡. 下表给出了典型 MS-DOS 机器上这些中断的配置.

表 4－1 可屏蔽硬件中断

PIC 线	起源*	中断号	中断用法
IRQ0	8253-5 时钟	08h	BIOS 时间功能,每秒钟产生 18.2 次中断
IRQ1	键盘	09h	通知键盘输入
IRQ2	保留		(AT 机上为控制器 1 的门路 gate)
IRQ3	COM2	0Bh	通信(次级)
IRQ4	COM1	0Ch	通信(主)
IRQ5	硬盘	0Dh	来自硬盘控制器的信号
IRQ6	软磁盘	0Eh	来自软盘控制器的信号
IRQ7	打印机	0Fh	来自打印机控制器的信号

PIC 由向处理器发送一个 INTR 信号服务于最高优先级的中断线。处理器证实了此请求后,PIC 将此中断号置于数据总线上,处理器然后产生此中断,实际的中断号是将 IRQ 线上的数值加上 8 得到的。由清除 PIC 端口 21h 处的对应比特来使 IRQ 线有效。例如欲使 IRQ3 有效,读入端口 21h 处的值,关掉比特 3，再将此值写回到端口 21h。系统启动时，IRQ3 和 IRQ4 均无效。

注意，AT 机使用了两个 8259A 控制器,故有 15 条可用的线。这两个控制器以串联方式连接在一条输入线上，IRQ2 用于接收来自第二个 8259 的输出。

4.3.2 硬件中断服务程序

下面的 C 程序作为安装和使用硬件中断服务程序的一个例子,它利用了键盘中的中断(09h)处理程序。安装子程序 (ini _ kbrk)和中断处理程序 (kb _ brk) 包含在后面的汇编语言文件中。

此中断处理程序的目的是加强程序的有效性并简化其编码。此 C 程序的基本结构对于多数应用程序是共同的,用一个主菜单来作出各种选择。在一个实际的应用程序中,一个选择功能可以是很长的,用户应能在任何时候返回到主菜单。

为使用户能够中断一个进程,查询键盘的功能调用通常必须被插入代码中。这使得编程复杂化,降低了效率,且响应不迅速。下面写出的 C 程序配合汇编程序所给出的另一种方法对用户的输入提供了即时的响应,且在程序中无需置入键盘功能调用。

```
/*
     File:      KBRK.C

     A C program demonstrating the use of a keyboard interrupt handler

     This program must be linked with the assembler module KBRKA.ASM
     (Figure 6.7).  To generate an .EXE file from KBRK.C and KBRKA.ASM:
          MSC KBRK;
          MASM KBRKA;
          LINK KBRK+KBRKA
*/

#include <setjmp.h>
#include <dos.h>

jmp_buf mark;
int  menu (void);                                  /* C function declarations. */
void demo1 (void);
void demo2 (void);
void printa (char *,int,int,int);
void cls (int,int,int,int);

void init_kbrk (void);                                  /* Assembler function. */

main ()
     {
     int choice;

     while (setjmp (mark))              /* Sets return point for 'longjmp ()'. */
          ;
     choice = menu ();                  /* Display main menu, get choice.      */
     switch (choice)
          {
          case 1:
               init_kbrk ();     /* Activate the keyboard interrupt routine. */
               demo1 ();
               break;
          case 2:
               init_kbrk ();     /* Activate the keyboard interrupt routine. */
               demo2 ();
               break;
          case 3:
```

```
            cls (0,0,24,79);
            exit (0);
        } /* end switch */

    } /* end main */


int menu ()
    {
    int ch;
    cls (0,0,24,79);
    printa ("--------------------------------------------------------",64,7,15);
    printa ("|                         MAIN MENU                    |",64,8,15);
    printa ("|    (1)  DEMO ONE                                     |",64,9,15);
    printa ("|                                                      |",64,10,15);
    printa ("|    (2)  DEMO TWO              ENTER CHOICE:          |",64,11,15);
    printa ("|                                                      |",64,12,15);
    printa ("|    (3)  EXIT TO DOS                                  |",64,13,15);
    printa ("--------------------------------------------------------",64,14,15);
    printa (" ",64,11,60);
    while ((ch = getch()) < '1' || ch > '3')
        ;
    return (ch - 48);

    } /* end menu */

void demo1 ()
    {
    cls (0,0,24,79);                             /* Clear the whole screen. */

    printa ("--------------------------------------------------------",64,7,15);
    printa ("|                                                      |",64,8,15);
    printa ("|                   THIS IS DEMO ONE                   |",64,9,15);
    printa ("|                                                      |",64,10,15);
    printa ("|              PRESS ANY KEY TO INTERRUPT              |",64,11,15);
    printa ("|                                                      |",64,12,15);
    printa ("--------------------------------------------------------",64,13,15);

    for (;;)                                     /* This is forever. */
        ;

    } /* end demo1 */

void demo2 ()
    {
    cls (0,0,24,79);                             /* Clear the whole screen. */

    printa ("--------------------------------------------------------",64,7,15);
    printa ("|                                                      |",64,8,15);
    printa ("|                   THIS IS DEMO TWO                   |",64,9,15);
    printa ("|                                                      |",64,10,15);
    printa ("|              PRESS ANY KEY TO INTERRUPT              |",64,11,15);
    printa ("|                                                      |",64,12,15);
    printa ("--------------------------------------------------------",64,13,15);

    for (;;)                                     /* This is forever. */
        ;

    } /* end demo2 */

void brkdisp ()    /* Display function called by keyboard interrupt handler. */
    {
    printa ("--------------------------------------------------------",30,22,15);
    printa ("| CONTINUE (C) OR RETURN TO MAIN MENU (R) ?            |",30,23,15);
    printa ("--------------------------------------------------------",30,24,15);
    printa (" ",30,23,61);                       /* Place cursor for reply. */

    } /* end brkdisp */

void printa (s, a, r, c)          /* Position cursor and write a string with */
char *s;                          /* specified color or video attribute.     */
```

```
int a, r, c;
/*
     s    : String to be displayed on screen.
     a    : Display attribute.
     r,c  : Starting row and column (0 .. 24, 0 .. 79) (ul corner = 0,0).
*/
     {
     union REGS cur_regs, write_regs;

     while (*s)                          /* Continue until null at end of string. */
          {
          cur_regs.h.ah = 2;              /* Update position of cursor using     */
          cur_regs.h.dh = r;              /* BIOS set cursor position function. */
          cur_regs.h.bh = 0;
          cur_regs.h.dl = c++;
          int86 (0x10, &cur_regs, &cur_regs);

          write_regs.h.ah = 9;            /* Write char & attribute at current  */
          write_regs.h.bh = 0;            /* cursor position using BIOS.        */
          write_regs.x.cx = 1;
          write_regs.h.bl = a;
          write_regs.h.al = *s++;
          int86 (0x10, &write_regs, &write_regs);
          }

     } /* end prints */

void cls (row1,col1,row2,col2)       /* This function uses BIOS interrupt 10h  */
int row1,col1,row2,col2;             /* to clear the specified area of screen. */
/*
     row1, col1 : Upper left corner of window to be cleared.
     row2, col2 : Lower right corner of window to be cleared.
*/
     {
     union REGS reg;

     reg.h.ah = 6;
     reg.h.al = 0;                              /* Blank the whole window. */
     reg.h.ch = row1;
     reg.h.cl = col1;
     reg.h.dh = row2;
     reg.h.dl = col2;
     reg.h.bh = 7;

int86 (0x10, &reg, &reg);

} /* end cls */
```

上面是使用键盘中断处理程序使用的C程序。

```
page 50,130

;File:     KBRKA.ASM

;Keyboard interrupt routines callable from a C program
;          Called from the C program KBRK.C

public     _init_kbrk
public     c_ds
public     int09_off
public     int09_seg
public     kb_act
public     kb_brk
public     k00, k01, k02, k03, k04, k05
public     acls

_data      segment word public 'DATA'
extrn      _mark:word
_data      ends

dgroup     group    _data
assume     cs:_text, ds:dgroup
```

```
_text       segment byte public 'CODE'

extrn       _brkdisp:near               ;List of all C functions called.
extrn       _longjmp:near

_init_kbrk proc     near                ;** Initializes keyboard interrupt handler.
                                        ;void init_kbrk ();
            assume  cs:_text

            mov     cs:c_ds, ds         ;Save C data segment for use by kb_brk.

            mov     ah, 35h             ;Save old int 09h vector, called by
            mov     al, 09h             ;kb_brk.
            int     21h
            mov     cs:int09_off, bx
            mov     cs:int09_seg, es

            mov     ah, 25h             ;Set int 09h to point to kb_brk.
            push    cs
            pop     ds
            mov     dx, offset kb_brk
            mov     al, 09h
            int     21h

            mov     ds, cs:c_ds         ;Restore data segment.
            ret                         ;Return to C program.

_init_kbrk endp

                                        ;** Code segment data. **
c_ds        dw      ?                   ;Stores C data segment.
int09_off   dw      ?                   ;Stores old int 09 vector.
int09_seg   dw      ?
kb_act      db      0                   ;Flag: 1 => kb_brk already active.

kb_brk      proc    far                 ;** Keyboard interrupt handler. **
            assume  cs:_text

k00:        cmp     cs:kb_act, 0        ;Test if routine already active.
            je      k01                         ;If active:
            jmp     dword ptr cs:int09_off      ;branch to original vector.
k01:        mov     cs:kb_act, 1        ;Set active flag.
            pushf                       ;Simulate interrupt to original vector.
            call    dword ptr cs:int09_off
            push    ax
            mov     ah, 01h             ;Test if key available.
            int     16h
            jnz     k02
            pop     ax                  ;No key available -- restore & go back.
            mov     cs:kb_act, 0
            iret

k02:                                    ;Key is available.
             mov     ah, 0              ;Read it.
             int     16h
             sti                        ;Enable interrupts.
             push    bx                 ;Save more registers.
             push    cx
             push    dx
             call    _brkdisp           ;Display message.
k03:         mov     ah, 0              ;Read user's response.
             int     16h
             cmp     al, 'C'            ;Test response.
             jz      k04
             cmp     al, 'c'
             jz      k04
             cmp     al, 'R'
             jz      k05
             cmp     al, 'r'
             jz      k05
             jmp     k03                ;Invalid key entered, go back.
k04:                                    ;** Continue. **
             call    acls               ;Assembler version of clear screen.
             pop     dx                 ;Restore registers.
             pop     cx
```

```
        pop     bx
        pop     ax
        cli                             ;Prevent recursive call.
        mov     cs:kb_act, 0            ;Indicate routine not active.
        iret                            ;Go back.
k05:                                    ;** Return to main menu. **
        mov     ah, 25h                 ;Restore interrupt 09 vector.
        lds     dx, dword ptr cs:int09_off
        mov     al, 09h
        int     21h
                                        ;Call 'longjmp'.
        mov     ax, 1                   ;Second parm is 1.
        push    ax
                                        ;First parm is address of 'mark' array.
        mov     ax, offset dgroup:_mark
        push    ax
        mov     ds, c_ds                ;Restore C data segment.
        mov     cs:kb_act, 0            ;Indicate routine not active.
        call    _longjmp                ;Back to instruction after 'setjmp'.

kb_brk  endp

scls    proc    near                    ;Routine which uses BIOS interrupt 10h,
        assume  cs:_text                ;function 6, to clear last three rows
                                        ;of the screen.
        mov     ah, 6                   ;Scroll active page up function.
        mov     al, 0                   ;0 => blank entire window.
        mov     ch, 22                  ;UL row.
        mov     cl, 0                   ;UL column.
        mov     dh, 24                  ;LR row.
        mov     dl, 79                  ;LR column.
        mov     bh, 7                   ;Normal/b & w attribute.
        int     10h                     ;BIOS video services.
        ret

scls    endp

_text   ends                            ;End of code segment.
        end
```

上面是可用C程序调用的键盘中断程序.

(1) 中断处理程序的基本特点

本节介绍上面的C程序和汇编程序是如何工作的.

C的库函数setjmp及Longjmp用于使程序可从代码的任意点(甚至从嵌入很深的功能调用中)跳回main的开头.函数setjmp在外部数组mark中保存寄存器和堆栈的当前状态,以后当调用Longjmp时,恢复全部机器状态,程序从setjmp后面的一条指令继续执行.

可从主菜单选择两个主要程序部分demo1和demo2之一.在程序分支之前,由调用汇编子程序init_kbrk初始化键盘中断处理程序.init_Kbrk保存中断09h向量的旧值,并重置此向量指向kb_brk.以后由键盘动作产生的中断将分

支到 kb _ brk。

C 程序的演示(demo)子程序调用后,显示出信息,然后简单地进入一个无穷尽的回路以模似一个冗长、无交互的进程。当用户按下一个键时,kb _ brk 立即接收控制。此函数执行下列步骤:

· 测试标志 kb _ act 以查看是否已进入 kb _ brk;如果已进入,立即跳至原来的中断程序以避免递归调用。

· 设置 kb _ act 标志以防止递归调用。

· 调用原来的中断程序使其可处理来自键盘的扫描码。

· 返回时,调用 BIOS(功能 16h)以查看是否有键盘输入在等待读入。

· 如果无键入,恢复 AX,重置 kb _ act,并将控制返回给演示程序。

· 如果有键入,读入,使中断有效,提示用户打入 C 以继续此演示程序,或打入 R 返回到主菜单。

· 如果用户选择"继续",则恢复状态并发出 iret。

· 如果用户决定返回主菜单,则恢复中断 09h 向量,使主程序可执行普通的键盘输入。最后,调用 Longjmp,控制立即返回到 main 的开头。注意,不必将寄存器"弹"出堆栈,因为 Longjmp 恢复了原始堆栈。

(2) 硬件中断处理程序的一般导引

上面的 C 程序及汇编程序说明了编写硬件中断处理程序的几个重要导引。首先,一个中断处理程序应链接回(chain back)原始的中断子程序以避免妨碍系统的其他部分。故保存了原始的中断 09h 向量,且在此地址处的子程序可由任何中断调用。

链接回原始中断 09h 子程序的另一个重要原因为硬件

中断是通过 8259A PIC 安排的. 在一个中断信号结束的信息被送至此芯片之前,所有低于当前中断优先级的中断均无效(且不管该中断标志的状态如何). 此功能对一般的原始 BIOS 程序都可执行. 但如果无法链接上一个能通知 PIC 的子程序,可使用下面的端口输出语句通知中断的结束:

```
mov  al,20h            ;PIC 端口为 20h
out  20h,al            ;20h 通知中断已结束
```

另外,一个中断程序接收控制后,中断均使无效,故应尽可能快地用 sti 指令使中断功能有效,否则会妨碍其他进程,例如,时钟中断 08h,调用了 BIOS 的时间跟踪功能.

设计一个中断处理程序时,应仔细考虑使中断有效的位置及使其无效的位置. 这一点与管理多任务系统中的异步进程所遇到的问题很类似. 一般来说,中断在代码的一个重要位置处暂时地使之无效以阻止另一个进程获取控制或干扰当前的任务,或破坏当前正在被修改的数据. 例如上面汇编程序中,在标号 k05 上面,使中断无效以防止 kb_acl 信标(semaphore)被重置为 0 时重入该代码段.

4.4 其他

MS-DOS 机器还有几个资源应当提及. 包括数据中断向量、硬件端口、显示内存区及可安装设备驱动程序.

4.4.1 数据中断向量

并非所有中断向量均包含可执行程序的地址,其中有几个包含的是存贮着 BIOS 所用输入/输出参数表的内存地址. BIOS 将其部分数据的地址放置在中断向量表中的原因是,此种机构使得程序员可由其他的数据表来置换此内存地址中的内容. 正如系统中断子程序可由改变中断向量表中的地址来替换或捕获,亦可由置换某些内部数据来修改系统的

性能．推荐的步骤是,首先将当前数据拷贝到程序中的某个区域,做完修改后,重置相应的中断向量以指向经修改后的表．下面描述 BIOS 使用的数据中断向量．

▲1Dh:显示器参数

向量 1Dh 通常指向用于初始化显示控制器上 6845 芯片的参数表．系统启动时,该向量指向 ROM BIOS 中包含的数据(在 PC 中为 F000h:F0A4h,其内容见 BIOS 清单)．但比向量可被修改以指向另一个表．例如,EGA 控制器的 BIOS 可调整此向量以指向包含在 C000h 段的一个修改过的表．

▲1Eh：磁盘基本参数

系统启动时,向量 1Eh 指向标准 BIOS 中的一个数据表(在 PC 中为 F000h:EFC7h,其内容见 BIOS 清单)．但其可由经修改过的磁盘参数块的地址来替换．MS-DOS 装入时,它重置此向量以指向它自己的一张表(对于版本 2.1，在地址 0000f:0522h 处)．MS-DOS 动态地修改这些参数以加强磁盘性能,此技术亦可由注意性能的程序员用于编写非 DOS 的磁盘子程序．修改下面两个域对磁盘性能的改进尤其重要:

(1) 偏移量 9h:磁头稳定时间

PC／XT 的默认值是 25 毫秒,在读操作时暂时将此值设置为 0 可改进磁盘访问速度(写操作时仍使其为 25 毫秒)．

(2) 偏移量 Ah:电机启动时间

默认值为 4(＊1／8 秒)．可经实验来修改此值,数值小可改进性能,但太小会产生错误．

▲1Fh:扩展的图形字符表

向量 1Fh 指向一个比特映照数据的表,用于在图形方式下产生 ASCII 代码 128 到 255 的字符．ROMBIOS 中的数据用于产生前 128 个字符(在 PC 机中为 F000h:FA6Eh 处),

但不包含高 128 个的数据。系统启动时,此向量设置为 F000h:0000h,表示无数据可用。程序员必须生成一张新表,并重置此向量以指向它。参见 BIOS 清单可产生字符所需的过程(PC 机中,位于偏移量 FA6Eh 处)。

▲41h:固定盘参数

向量 41h 指向一张用于控制固定磁盘的数据表。硬盘控制器 ROM 一般将其设置为 C800h 段的某个地址。

▲EGA 图形显示器参数

安装有 EGA 控制器后,向量 43h 指向一张包含显示器初始化参数的表。系统启动时,该向量被设置为指向 EGA ROM 中段 C000h 处包含的默认参数表(见 EGA 手册中的 ROM 清单)。

▲EGA 图形字符表

安装有 EGA 控制器后,向量 43h 指向一张包含显示器初始化参数的表。系统启动时,该向量被设置为指向 EGA ROM 中段 C000h 处包含的默认参数表(见 EGA 手册中的 ROM 清单)。

▲EGA 图形字符表

安装有 EGA 控制器后,向量 44h 指向一张含有用于产生图形字符的 dot 图形的表。显示方式在 4,5,6 下,该表用于前 128 个字符(用于高 128 个字符的表由向量 1Fh 指向)。但在所有其他的 EGA 图形方式中,此表用于全部 256 个字符。

4.4.2 端口

8086 系列的处理器能够寻址 64K个 I/O 端口。为专门端口所编写的程序,从本质上说是不可移植的.一般来说,I/O 可通过 DOS 或 BIOS 来控制,但有些特殊目的功能如

不进行端口访问便无法获得足够好的性能。某些端口接口在各种兼容机中以及以后的硬件版本下都不改变。

下表列出 IBM PC/AT 兼容机的当前端口分配。IBM 可能在将来会使用未列出I/O地址中的任何一个。

I/O 地址范围	用途
0000h–000Fh	PC:DMA 芯片(8237A–5)
0000h–001Fh	AT:DMA 芯片 1(8237A–5)
0020h–0021h	PC:中断控制器(8259A)
0020h–003Fh	AT:中断控制器 1(8259A)
0040h–0043h	PC:可编程时钟(8253–5)
0040h–005Fh	PC:可编程时钟(8254)
0060h–0063h	PC:可编程外设接口(82554–5)
0060h–006Fh	AT:键盘
0070h–007Fh	AT:不可屏蔽中断(NMI)参考寄存器
0080h–0083h	PC:DMA 页寄存器
0080h–009Fh	AT:DMA 页寄存器
00A0h	PC:不可屏蔽中断(NMI)参考寄存器
00A0h–00BFh	AT:中断控制器 2(8259A)
00C0h–00DFh	AT:DMA 芯片 2(8237A–5)
00F0h–00FFh	AT:数学协处理器
01F0h–01F8h	AT:硬盘控制器
0200h–0207h	AT:游戏控制
0200h–020Fh	PC:游戏控制
0210h–0217h	PC:扩展单元
0278h–027Fh	AT:第二并行打印机
02F8h–02FFh	第二串行口

0300h–031Fh	样机研制卡
0320h–032Fh	PC:硬盘控制器
0378h–037Fh	第一并行打印机
0380h–038ch	SDLC 通信
0380h–0389h	第二二进制同步通信
0390h–0393h	盘束
03A0h–03A9h	第一二进制同步通信
03B0h–03BFh	单色显示器适配器／打印机
03D0h–03DFh	彩色图形适配器(CGA)
03F0h–03F7h	软磁盘控制器
03F8h–03FFh	第一串行口
0790h–2393h	盘束适配器 1 到 4

4.4.3 可安装设备驱动程序

MS–DOS2.0 以上版本所提供的最后一个重要资源是可安装的设备驱动程序。其内容用一本单独的书来讲述。

4.4.4 *Ctrl–C* 处理程序

MS–DOS 检测到键盘或其他输入流(stream)有一个 Ctrl–x(03H)在等待时,便执行Int23H向量所存贮地址的子程序。一般情况下,此向量所指向的程序仅仅终止当前活动的进程并将控制返回给其父进程——通常即是 MS–DOS 命令解释程序。

也就是说,如果一个程序正在执行,你不小心按下了 Ctrl–C,该程序便会简单地流产。使用文件控制块打开的所有文件均不能正常关闭;经你改掉的中断向量也无法正常恢复;如果此时正在执行某种直接的输入／输出操作(例如,程序中有一个用于串行端口的中断驱动程序),则可能发生各种意料不到的结果。

你可在程序中采用几种方法来避免遇到 Ctrl-C 后会失控。首先介绍一点预备知识:

MS-DOS 环境下的键盘输入技术分为两大类。一种是通过 MS-DOS 功能调用(Int21H)来实现与硬件无关且与其他操作系统具有兼容性的键盘字符输入方法。另一种方法依赖于机器固件(如 ROM BIOS)或直接访问键盘控制器,使用这种方法的程序移植差,并且在多任务环境会出问题。

第一种键盘输入方法采用"句柄流式 I/O"功能(Handle stream I/O),即某个应用程序接收控制后,已被赋给了五个句柄(或通道号),这些句柄是为以下的五个字符设备打开的:

句柄	名字	别名
0	标准输入设备	CON
1	标准输出设备	CON
2	标准出错设备	CON
3	标准辅助设备	AUX
4	标准列表设备	PRN

这些句柄可直接使用在相关的逻辑设备上执行读、写操作。

对字符设备作读、写操作又可分为两大方式:

• 已处理(cooked)方式:

操作系统对接收或发送的每一个字符进行检测,找到一些特殊字符后便执行相应的特殊动作。此时的字符流是已"过滤"的。

• 未处理(raw)方式:

操作系统不对任何字符作任何检测。

所有字符设备在默认状态下均采用"已处理"方式,当然需要时也可由程序选择"未处理"方式。如果标准输入处于"已处理"方式,则 MS-DOS 所用的内部 128 字节缓冲区要填入由键盘读入的字符。用户可借助 Backspace 等特殊功能键对此输入进行编辑,也将检测到 Ctrl-C,一旦用户按下回车键,输入的字符数从内部缓冲区拷贝到调用程序缓冲区,直至 Return 作为结束。

如果标准输入为"未处理"方式,读入字符数时是不考虑 Return,Ctrl-C 等控制字符的。实际读入的字符数总是在寄存器 AX 中返回。

与机器无关的键盘输入方法常用以下几个功能调用:

功能	动作	Ctrl-C 检查
01H	带回送的键盘输入	yes
06H	直接控制台输入/输出	no
07H	无回送的键盘输入	no
08H	无回送的键盘输入	yes
0AH	读入经缓冲的一行	yes
0BH	读输入状态	yes
0CH	读输入缓冲区和输入	随情况变

避免输入 Ctrl-C 后程序失控的方法可以是:

(1) 通过功能调用 06H 和 07H 执行所有键盘输入和状态检查,使控制台驱动程序采用"未处理"方式。

(2) 通过功能调用 06H(直接控制台 I/O)或直接利用

ROM BIOS 甚至显示控制器刷新缓冲区来执行所有的显示输出。

(3) 将其他字符设备(AUX,PRN)等设置成"未处理方式"。

(4) 使用功能调用 33H(获得或设置 Ctrl—Break 标志)使 Ctrl—C 检测失效。

但这四种方法都不够理想,更好的办法是让 Ctrl—C 检测发生,但应换成自己写的 Ctrl—C 处理程序,此处理程序可以什么都不做,也可以按照应用程序的要求动作。

下面的例子是某应用程序采用了一个什么都不做的 Ctrl—C 处理程序。程序代码的第一部分用于改变 Int23H 向量的内容,在应用程序的初始化部分执行。只要 MS—DOS 在键盘处或在某个文件或设备字符流处检测到一个 Ctrl—C,则名为 Brk—Routine 的处理程序将获得控制。本例中的处理程序只是作立即 Interrupt Return,故 Ctrl—C 仍留键盘输入流中,并将应用程序下一次从键盘请求字符时传送过去(屏幕上显示为^C):

```
    mov  ah, 25h                    ;功能 25H＝设置中断
    mov  al, 23h                    ;Int 23H 是 Ctrl—C 处理程序中断
                                    向量
    mov dx,seg  Brk__Rtine          ;使 DS:DX＝处理程序地址
    mov  ds,dx
    mov  dx,offset  Brk__Routine
    int 21h                         ;调用 DOS 改变此向量
    Brk—Roution:                    ;此为 DOS 检测到 Ctrl—C 后
    iret                            ;调用的处理程序
```

该应用程序结束后,MS—DOS 根据程序段前缀(PSP)中

保存的信息自动恢复 Int23H 向量以前的内容。

4.4.5 Ctrl–Break 处理程序

IBMPC 及其兼容机还有一个由 ROM BIOS 键盘驱动程序调用的中断处理程序,用于检测到特殊键组合 Ctrl–Break 后进行处理。此处理程序的地址保留在 Int1BH 向量中。在 MS–DOS 下,此向量通常指向一个什么也不做只是设置一个标志然后执行一个 INTERRUPT RETURN 的中断处理程序。占用此中断向量非常有用,但有时有点风险。因为键盘是中断驱动的,故按下 Ctrl–Break 几乎在任何场合(甚至在程序已"崩溃"或进入"死循环")下都可获得控制。

一般来说,不能把为 Int23H 编写的处理程序用于 Int1BH,Int1BH使用起来更受限制,因为它是作为一个硬件中断的结果被调用的,而 MS–DOS 在发出此中断时有可能正在执行一段非常重要的代码段。另外,除 CS:IP 以外的所有寄存器都正处于未知状态,故在该中断处理程序可执行前,可能需先保存然后才修改。还有在 Int1BH 处理程序被调用时堆栈的深度也不得而知,而如果此处理程序打算执行的操作频繁使用堆栈,应保存原有堆栈段和堆栈指针,并切换到有足够深度的新堆栈。

可利用 Int1BH 获得对系统的控制,并直接将应用程序在某一点上分支,但在使用此技术以前必须考虑到以下几点:

· 由于硬件中断自动使其他所有中断失效,故要尽可能早地执行指令 sti,使那些中断重新可用。

· 有些兼容机(例如使用 ROM 版本 C 的 Compaq 公司的便携机),其 ROM 键盘驱动程序中有 bug, 在 Int1BH 处理程序获得控制后将使系统死机,解决办法是使应用程序执

行下面代码段:

```
in al, 61h
or al, 80h
out 61b, al
and al, 7fh
oit 61h, al
```

• 因为此中断是由硬件产生的,故处理程序必须向 8259A 中断控制器发出一条 E0I(中断结束),不然便会出现死机。对于 IBM PC 及其兼容机,由下面的代码完成:

```
mov al, 20h
out 20h, al
```

• MS-DOS 的 IBM 实现(PC-DOS)在应用程序结束后并不自动恢复 Int1BH 向量的内容,因为此向量属于 ROM BIOS 而不属于 MS-DOS. 如果打算为 Int1BH 写一个处理程序,则必须首先保存该向量的以前状态才能修改它,在你的程序退出前再恢复至原始状态.

下面介绍的例子 BREAK.ASM 是一个可与 Microsoft C 程序连接的 Ctrl-Break 处理程序. 后面的一个简短的 C 程序将说明此处理程序如何使用.

```
1           page    55,132
2           title   Ctrl-Break handler for Microsoft C programs
3           name    break
4
5   ;
6   ; Ctrl-Break Interrupt Handler for Microsoft C programs
7   ; running on IBM PCs (and ROM BIOS compatibles)
8   ;
9   ; Ray Duncan, May 1985
10  ;
11  ; This module allows C programs running on the IBM PC
12  ; to retain control when the user enters a Ctrl-Break
13  ; or Ctrl-C.  This is accomplished by taking over the
14  ; Int 23H (MS-DOS Ctrl-C) and Int 1BH (IBM PC
15  ; ROM BIOS Keyboard Driver Ctrl-Break) Interrupt
16  ; vectors.  The interrupt handler sets an internal
17  ; flag (which must be declared STATIC INT) to TRUE within
```

```
18    ; the C program; the C program can poll or ignore this
19    ; flag as it wishes.
20    ;
21    ; The module follows the Microsoft C parameter passing conventions.
22    ;
23    ; The Int 23H Ctrl-C handler is a function of MS-DOS
24    ; and is present on all MS-DOS machines; however, the Int 1BH
25    ; handler is a function of the IBM PC ROM BIOS and will not
26    ; necessarily be present on other machines.
27    ;
28
29    args    equ     4               ;offset of arguments, small model
30
31    cr      equ     0dh             ;ASCII carriage return
32    lf      equ     0ah             ;ASCII line feed
33
34
35    _TEXT   segment byte public 'CODE'
36
37            assume cs:_TEXT
38
39            public  _capture,_release       ;function names for C
40
41            page
42    ;
43    ; The function CAPTURE is called by the C program to
44    ; take over the MS-DOS and keyboard driver Ctrl-
45    ; Break interrupts (1BH and 23H).  It is passed the
46    ; address of a flag within the C program which is set
47    ; to TRUE whenever a Ctrl-Break or Ctrl-C
48    ; is detected.  The function is used in the form:
49    ;
50    ;               static int flag;
51    ;               capture(&flag);
52
53    _capture proc   near            ;take over Ctrl-Break.
54
55            push    bp              ;interrupt vectors
56            mov     bp,sp
57            push    ds              ;save registers.
58            push    di
59            push    si
60
61            mov     ax,word ptr [bp+args]
62            mov     cs:flag,ax      ;save address of integer
63            mov     cs:flag+2,ds    ;flag variable in C program.
64
65                                    ;pick up original vector content
66            mov     ax,3523h        ;for interrupt 23H (MS-DOS
67            int     21h             ;Ctrl-C handler).
68            mov     cs:int23,bx
69            mov     cs:int23+2,es
70
71            mov     ax,351bh        ;and interrupt 1BH
72            int     21h             ;(IBM PC ROM BIOS keyboard drive
73            mov     cs:int1b,bx     ;Ctrl-Break interrupt handler).
```

```
74          mov     cs:int1b+2,es
75
76          push    cs              ;set address of new handler
77          pop     ds
78          mov     dx,offset ctrlbrk
79          mov     ax,02523h       ;for interrupt 23H
80          int     21h
81          mov     ax,0251bh       ;and interrupt 1BH.
82          int     21h
83
84          pop     si
85          pop     di
86          pop     ds              ;restore registers and
87          pop     bp              ;return to C program.
88          ret
89
90  _capture endp
91          page
92  ;
93  ; The function RELEASE is called by the C program to
94  ; return the MS-DOS and keyboard driver Ctrl-Break
95  ; interrupt vectors to their original state.  Int 23H is
96  ; also automatically restored by MS-DOS upon the termination
97  ; of a process; however, calling RELEASE allows the C
98  ; program to restore the default action of a Ctrl-C
99  ; without terminating.  The function is used in the form:
100 ;
101 ;               release();
102 ;
103
104 _release proc   near            ;restore Ctrl-Break interrupt
105                                 ;vectors to their original state.
106         push    bp
107         mov     bp,sp
108         push    ds              ;save registers.
109         push    di
110         push    si
111
112         mov     dx,cs:int1b     ;set interrupt 1BH
113         mov     ds,cs:int1b+2   ;(IBM PC ROM BIOS keyboard driver
114         mov     ax,251bh        ;Ctrl-Break interrupt handler).
115         int     21h
116
117         mov     dx,cs:int23     ;set interrupt 23h
118         mov     ds,cs:int23+2   ;(MS-DOS Ctrl-C
119         mov     ax,2523h        ;interrupt handler).
120         int     21h
121
122         pop     si
123         pop     di
124         pop     ds              ;restore registers and
125         pop     bp              ;return to C program.
126         ret
127
128 _release endp
129
```

```
130          page
131     ;
132     ; This is the actual interrupt handler which is called by
133     ; the ROM BIOS keyboard driver or by MS-DOS when a Ctrl-C
134     ; or Ctrl-Break is detected.  Since the interrupt handler
135     ; may be called asynchronously by the keyboard driver, it
136     ; is severely restricted in what it may do without crashing
137     ; the system (e.g. no calls on DOS allowed).  In this
138     ; version, it simply sets a flag within the C program to
139     ; TRUE to indicate that a Ctrl-C or Ctrl-Break has
140     ; been detected; the address of this flag was passed
141     ; by the C program during the call to the CAPTURE function.
142     ;
143
144     ctrlbrk proc   far                  ;Ctrl-Break interrupt handler
145
146             push   bx                   ;save affected registers
147             push   ds
148
149             mov    bx,cs:flag           ;set flag within C program
150             mov    ds,cs:flag+2         ;to "True"
151             mov    word ptr ds:[bx],-1
152
153             pop    ds                   ;restore registers and exit
154             pop    bx
155
156             iret
157
158     ctrlbrk endp
159
160
161     flag    dw     0,0                  ;long address of C program's
162                                         ;Ctrl-Break detected flag
163
164     int23   dw     0,0                  ;original contents of MS-DOS
165                                         ;Ctrl-C interrupt 23H
166                                         ;vector
167
168     int1b   dw     0,0                  ;original contents of ROM BIOS
169                                         ;keyboard driver Ctrl-Break
170                                         ;interrupt 1BH vector
171
172     _TEXT   ends
173
174             end
```

上面是可与Microsoft C程序连接的ctrl－C及Ctrl－Break 中断处理程序。

```
/*
        TRYBREAK.C

        Try Microsoft C Ctrl-Break interrupt handler
```

```
        Ray Duncan, May 1985

    */

    #include <stdio.h>

    main(argc, argv)
        int   argc;
        char  *argv[];
{   int hit = 0;                        /* flag for keypress */
    int c = 0;                          /* character from keyboard */
    static int flag = 0;                /* true if Ctrl-Break or
                                           Ctrl-C detected */

    puts("\n*** TRYBREAK.C running ***\n");
    puts("Press Ctrl-C or Ctrl-Break to test handler,");
    puts("Press the Esc key to exit TRYBREAK.\n");

    capture(&flag);                     /* pass address of flag */

    puts("TRYBREAK has CAPTUREd interrupt vectors.\n");

    while ( (c&127) != 27 )             /* watch for Esc key */
    {   hit = kbhit();                  /* check for keypress */
        if (flag != 0)
                { puts("\nCtrl-Break detected.\n");
                  flag=0;
                }
        if (hit != 0)                   /* read key if ready */
         {  c=getch();
            putch(c);                   /* and display it */
         }
  }
  release();
  puts("\nTRYBREAK has RELEASEd interrupt vectors.");
```

上面是说明如何使用中断处理程序BREAK.ASM的简单Microsoft C程序。

名为capture的函数在C程序中某个整型变量的地址处调用。它保存该变量的地址,并将Int1BH和Int23H向量指向新的中断处理程序,然后返回。

检测到Ctrl-C或Ctrl-Break后,中断处理程序将C程序中的整型变量设置为True(1),然后返回。C程序一有空就去查询此变量(如果该程序打算检测到一个以上的Ctrl-C,必须将此变量重置为00)。名为releade的函数简单地将

Int1BH 和 Int23H 向量恢复为原始值,从而也使该中断处理程序失效.

虽然本例中 Int23H 向量也是由 release 函数恢复的,但这并不是非常必要的,因为任何应用程序结束后,MS-DOS 将自动恢复中断向量. 但 Int1BH 是一个与 IBM-PC 机器有关的中断处理程序,MS-DOS 对其一无所知,故应用程序如果修改过它,则在退出前必须正确地恢复此向量. 否则该向量将仍然指向下一个程序运行时的某个随机区域,下一次用户按下 Ctrl-Break,必然是系统死机.

4.4.6 致命错误处理程序

所谓致命错误,指的是诸如试图访问文件时磁盘门打在,试图打印一个字符时打印机不在线等硬错误. MS-DOS 遇到致命错误后所执行的动作之一便是调用中断 24h. 初始化时,此向量指向一个打印出"Abort,Retry,Ignore?"信息的程序,然后向 DOS 返回一个代码通知其应执行这个动作中的哪一个. 问题是如果用户选择了 Abort(退出),程序便没有机会作准备,故有可能丢失数据. 编写自己的中断 24h 处理程序便可控制致命错误处理的过程. 自编致命错误中断处理程序的优点如下:

· 有机会打印出更详细、更有帮助的出错信息.

· 提供与屏幕设计一致的出错信息显示能力(DOS 只在当前光标位置上重写屏幕,给用户的映象是该程序已丢失了控制).

· 在用户选择 Abort 时,有机会关闭文件并执行最后清理.

介绍安装致命错误处理程序的方法最好通过举例. Microsoft C 没有可提供方便的库函数用于安装致命错误处

理程序．因此程序例由汇编语言的致命错误处理程序及可与C程序连接的安装过程组成．图4-6所给的C程序用于安装致命错误处理程序并测试其工作．图4-7为包含安装子程序cc_init和致命错误处理程序ce_hand本身的汇编语言模块．

```
/*
     File:     CRITERR.C

     A C program demonstrating a critical-error handler

     This program must be linked with the assembler module CRITERRA.ASM
     (Figure 6.5).  To generate program from CRITERR.C and CRITERRA.ASM:
          MSC CRITERR;
          MASM CRITERRA;
          LINK CRITERR+CRITERRA;
*/

#include <stdio.h>

main ()
     {
     FILE *testf;                      /* File to test if data is preserved.   */
     void lastrites (void);            /* Declare function for passing as parm. */
     void ce_init (void (*)());        /* Assembler routine: initializes       */
                                       /* error handler.                       */

                                /* Test file to see if stream closed properly. */
     testf = fopen ("testfile","w");
     fprintf (testf,"this data will not get recorded in the file's ");
     fprintf (testf,"directory entry if the program terminates abnormally\n");

     ce_init (lastrites); /* Initialize crit. error handler & pass address  */
                          /* of function to prepare program for termination.*/

     fprintf (stdprn,"crash");              /* Note: take printer off line.    */
     fopen ("a:crash","w");                 /* Note: leave drive A: door open. */

     fclose (testf);
     printf ("\nprogram continued normally ... not aborted");

     } /* end main */


void lastrites ()          /* This function should perform the minimum set of */
                           /* tasks required to cleanup and save data before  */
                           /* termination due to control-c or critical error. */
     {
     fcloseall ();                             /* Closes all open streams. */
     }
```

上面是安装并测试致命错误处理程序的C程序。

```
page 58,130

;File:      CRITERRA.ASM

;Assembler routines for critical error handler
;           Called from the C program CRITERR.C

public      _ce_init                        ;Called from C.
public      ce_hand                         ;Declare other symbols for CodeView.
public      terminate
public      c01, c02, c03, c04

_data       segment word public 'DATA'
                                            ;Error message table for 'ce_hand':

mess_table db       10,13,'critical error: write-protected diskette    $'
mess_len    equ     $ - mess_table
            db      10,13,'critical error: unknown unit                $'
            db      10,13,'critical error: drive not ready             $'
            db      10,13,'critical error: unknown command             $'
            db      10,13,'critical error: data error (CRC)            $'
            db      10,13,'critical error: bad request structure length$'
            db      10,13,'critical error: seek error                  $'
            db      10,13,'critical error: unknown media type          $'
            db      10,13,'critical error: sector not found            $'
            db      10,13,'critical error: printer out of paper        $'
            db      10,13,'critical error: write fault                 $'
            db      10,13,'critical error: read fault                  $'
            db      10,13,'critical error: general failure             $'

mess01      db      10,13,'abort, ignore, retry? $'
terminate   dw      ?                       ;Address of C terminate routine.

_data       ends

dgroup      group   _data
assume      cs:_text, ds:dgroup

_text       segment byte public 'CODE'

_ce_init    proc    near                    ;** Initilizes Critical Error Handler **
                                            ;void ce_init (func);
                                            ;void (*func)();

sframe      struc                           ;Template to access stack frame.
bptr        dw      ?                       ;Position of saved BP register.
ret_ad      dw      ?
param1      dw      ?                       ;Address of C function called on crit.err.
sframe      ends

frame       equ     [bp - bptr]             ;Base for accessing stack frame.

            push    bp                      ;Standard module initialization.
            mov     bp, sp
            sub     sp, bptr

            push    di
            push    si

            mov     ax, frame.param1        ;Save address of terminate routine to
            mov     terminate, ax           ;be called by 'ce_hand'.

            mov     cs:dataseg, ds          ;Save the C data segment for 'ce_hand'.

            push    ds                      ;Initialize interrupt 24h to point to
            mov     ah, 25h                 ;'ce_hand'.
            push    cs
            pop     ds
            mov     dx, offset ce_hand
            mov     al, 24h
            int     21h
            pop     ds
            pop     si                      ;Restore registers.
            pop     di
            mov     sp, bp
            pop     bp
```

```
        ret                             ;Return to C program.

_ce_init endp

ce_hand proc    near                    ;The critical error handler --
                                        ;activated by interrupt 24h.

        sti                             ;Turn interrupts back on.
        push    ax                      ;Save registers.
        pushf

        push    bx                      ;Save registers for writing messages.
        push    ds
        push    dx

        mov     ax, di                  ;Write error message using code passed
        mov     bx, mess_len            ;in DI as pointer to message table.
        mul     bx
        mov     ds, cs:dataseg
        mov     dx, offset dgroup:mess_table
        add     dx, ax
        mov     ah, 09h
        int     21h

        mov     ah, 09                  ;Prompt user for decision.
        mov     dx, offset dgroup:mess01
        int     21h

        pop     dx                      ;Done with messages: restore registers.
        pop     ds
        pop     bx

c01:    mov     ah, 07h                 ;Read user response.

        int     21h
        cmp     al, 'a'                 ;Branch on response value.
        je      c02
        cmp     al, 'A'
        je      c02
        cmp     al, 'r'
        je      c03
        cmp     al, 'R'
        je      c03
        cmp     al, 'i'
        je      c04
        cmp     al, 'I'
        je      c04
        jmp     c01                     ;Go back if invalid response.

c02:    popf                            ;Abort.
        pop     ax
        mov     ds, cs:dataseg          ;Restore C data segment value.
        call    word ptr cs:terminate   ;Call C terminate function.

        mov     ax, 4C01h               ;Exit w/ err level 1.
        int     21h

c03:    popf                            ;[illegible]
        pop     ax
        mov     al, 1                   ;1 => retry.
        iret                            ;Back to DOS.

c04:    popf                            ;Ignore.
        pop     ax
        mov     al, 0                   ;0 => ignore.
        iret                            ;Back to DOS.
dataseg dw      ?                       ;Store C DS value where addressible
                                        ;by 'ce_hand'.
ce_hand endp

_text   ends                            ;End of code segment.
        end
```

上面是包含安装子程序ce-init和致命错误处理程序ce-

hand 本身的汇编语言模块。

为安装并测试致命错误处理程序,该 C 程序首先打开一个数据文件并写一行数据. 然后调用汇编语言程序 ce_init 安装致命错误处理程序. 此函数只取一个变元: 如果用户选择Abort,则在程序结束前即是被调用的函数地址.C程序传送 Lastrites 地址,并由一个 C 函数关闭所有已打开的 stream. 一旦致命错误处理程序安装完, 程序试图首先打开驱动器 A 上的文件,然后试图向打印机写来测试其工作. 在运行此程序之前, 应使驱动器 A门打开着,并使打印机离线以构成致命错误的条件.

上面的过程 _ce_init 安装该致命错误处理程序. 它首先保存作为变元传送的函数地址, 从而可在程序流产之前由 _cc_hand 来调用此函数. 然后保存 C 数据段的值为以后由 _cc_hand 使用. 因为致命错误处理程序是由 DOS 调用的,故 DS 寄存器内容并不是欲访问 C 数据组的正确值,因此 DS 必须被保存,设置成正确的值,然后由 _ce_ hand 恢复以允许访问 C 数据. 最后,使用 DOS 功能 25h 设置中断向量 24h 以指向 _cc_hand.

过程 _cc_hand 按如下方式控制致命错误: DOS 调用中断 24h 后,寄存器 DI 包含范围从 0h 到 Ch 的出错代码. 这些出错代码对应于 mess_table 中所列的信息. 该过程由将 DI 值乘以单条出错信息长度(注意,所有信息长度都相同)打印出出错信息,然后将此值加到 mess_table 的偏移量上,结果得到的偏移量传送给 DOS 打印字符串功能 09h.

过程 _ce_hand 然后根据用户的决定分支程序走向. 如果用户选择 Abort,则调用程序员的结束子程序(本例中,实际调用的程序是 C 函数 Lastrites),程序通过 DOS 功能

4Ch(errorlevel 值为 1)来结束。如果用户选择 retry,则给寄存器 AL 赋值 1,此代码指示 DOS 重试此操作,并通过一个中断返回将控制返回给 DOS。如果用户选择 ignore,则 AL 中所赋值为 0,并发出一个中断返回。

通过 AL 寄存器可返回给 DOS 另外两个值:

AL=2 和 AL=3

AL=2 通过中断 23h 结束程序,本例中的过程 ce_hand 不采用这种方法,因为还可能也安装了 Ctrl-Break 处理程序,那样会使情况复杂化。

AL=3 使正在进行的系统调用失败,换言之,DOS 返回到应用程序并且不完成其服务功能而是返回一条出错信息。过程 ce_hand 不使用此选择的另一个原因是因为在 DOS3.X 下可能不允许 fail 请求,而是将其自动转换成 Abort。跳过结束程序从而破坏了本处理程序的目的。但有些应用程序可能要求即使错误条件不能被改正也要继续运行,此场合下,使 AL=3 可能是最好的选择。

致命错误处理程序将执行必要的动作以处理好出错情况,但要遵循两条重要原则:

- 保存所有寄存器的内容;
- 只许使用 DOS 功能调用 01h 至 0Ch,或 59h。

致命错误处理程序打印出简单的出错信息,所执行的任务也很简单.但在此基础上可设计出更高级的错误处理程序.大量的出错信息可供中断 24h 处理程序使用,出错信息包括:

- 寄存器 DI 包含简单的出错代码。这些出错范围自 0h 至 Ch,准确地对应于标准"扩展出错代码"19(十进制)至 31(十进制)。

·寄存器 BP:SI 包含该设备标题控制块的地址。此标题(header)包含涉及到该错误之设备的信息。

·寄存器 AH 中的比特 7 设置为 0 表示磁盘错误。AH 的比特 7 为 1 表示非磁盘错误(如果设备标题控制块指定是一块设备则表示是与磁盘有关的 FAT 内存映像错)。

如果 AH 的比特 7 为 0,表示为磁盘出错,但仍可使用如下的附加信息:

·寄存器 AL 包含出错的驱动器号,0 表示驱动器 A,等等。

·AH 比特 0 为 0 表示读操作,为 1 表示写操作。

·AH 比特 2 和 1 表示受影响的磁盘区域:

00 DOS区域

01 FAT

10 目录

11 数据区域

·AH 比特 3 为 0 则不允许 fail,为1 则允许 fail。

·AH 比特 4 为 0 则不允许 retry,为 1 则允许 retry。

·AH比特5 为 0,则不允许 ignore,为 1则允许 ignore。

第五章　兼容性的理论与测试

兼容性是注意性能的程序员常常关心的．为一部特殊的目标机器所编写的程序，当然可使用该机器任何可用的功能．但如果此程序设计时考虑到要有广阔的市场，并期望有较长的生命期，那就必须与一大类目前及将来的硬、软件环境相兼容．然而实际上许多最高性能的技术均有如下的不兼容问题：

- 与某些版本的操作系统不兼容
- 与非 IBM 兼容的 MS-DOS 机器不兼容
- 与新开发出来的外部设备不兼容
- 与网络系统不兼容
- 与 Windows，Topview 等多任务环境不兼容

这是否意味着，要编写一个可移植应用程序就必须放弃高性能，并且程序只能最低限度地使用最普通的资源呢?绝对不是．实际上，性能与兼容性并不互斥．经过对运行时间计算环境所存在资源的分析，有可能选出最高性能的技术且与此环境相兼容．

首先介绍开发与目前及将来的机器和操作系统相兼容的应用软件的一般准则，不难遵守这些准则又不失软件性能；然后描述确定所运行硬件与软件环境的程序方法；最后讨论环境特性一旦建立，如何最佳地使用当前资源．

本章介绍的方法为在高性能程序中保持兼容性提供了一般性的处理．但这里提供的信息只是一个起点，随着新的处

理硬件、外设及操作系统的大量涌现，需进行不断的研究以保证该应用程序保持其兼容性，同时又可使用可提供的最先进功能。

5.1 一般的兼容性准则

有关编写兼容性应用程序的规则的 MS-DOS 文献有许多。本节介绍通常不影响软件性能的一般规则，故可用于多数应用程序。下一节讨论限制了软件性能的规则（例如不能寻址绝对内存地址等）。

随着多任务环境及内存驻留实用程序的出现，一个程序不能再假定它是内存中的唯一进程，因此使用正规的通道并保持 DOS 随时了解程序动作很重要。只要使用 MS-DOS 就要尽可能遵守下列规则：

(1) 只要可能便使用 MS-DOS 文件系统。

文件系统是 MS-DOS 的强点之一，通过文件系统实现磁盘 I／O 比使用 DOS 中断 25h，26h，BIOS 中断 13h 或程序员编写的子程序对特定扇区作直接操作要好得多。

(2) 使用 DOS 句柄功能调用，不使用 FCB 功能或传统的字符设备 I／O 功能 01h 到 0Ch。

(3) 用 DOS 管理进程。

使用 EXEC 功能装入程序覆盖模块并执行子进程，允许 DOS 去控制重定位及其他程序加载的细节。使用 DOS 功能跟踪有关当前进程的系统信息。操作系统保持着当前进程的一个内部记录，该记录通过其程序段前缀的地址来标识。

(4) 由 DOS 管理内存分配。

通过功能 48h，49h 及 4Ah 分配和释放内存，不要使用未分配给该进程的内存。

(5) 释放不使用的内存。

在多任务环境下，一个程序释放掉由装入程序分配的但未使用的内存十分重要。注意，COM 文件总是分配掉所有自由用户内存，必须通过 DOS 功能 4Ah 释放内存。但 .EXE 文件按照其文件头中的指令分配程序以上的附加内存。在默认情况下，连接程序产生的.EXE 文件头要求所有可用的内存，使用/cp:n LINK 命令行选择可减小所需内存量，其中 n 为所需要附加内存的 16 字节段落数。将此值设置为 1，将使分配的内存减至最小（在默认情况下，此值设计为 65535，表示最大内存分配）。

(6) 使用较新的 DOS 退出功能 4Ch（终止）及 31h（结束并驻留），不采用老的功能 00h（终止）、中断 20h（等同于功能 00h）、及中断 27h（终止并驻留）。

(7) 使用中断时，通过 DOS 功能 25h 和 35h 获得并设置中断向量，不使用直接对内存写的方法。

使用这些功能将把你的意图通知 DOS，且在该向量处于转换状态时使中断失效。应存储任何经修改中断向量以前的内容，并在程序结束前恢复其值。此规则的例外是中断 23h 和 24h，它们由 DOS 根据存储在程序段前缀中的值自动恢复。

(8) 使用 MS-DOS 环境实现用户与应用程序的通信（例如配置文件、覆盖程序的地点等）。

(9) 使用系统时间功能（DOS 功能 2Ch 或 BIOS 中断 1Ah）或 8253 可编程定时器，不使用软件定时回路（其周期取决于处理器速度）。

(10) 通过有文档的中断向量访问 BIOS 功能，而不要通过绝对内存地址进行访问。

5.2 确认计算机环境

获得尽可能高的性能的第一步，是研究出一种系统化的、可靠的方法用以确定和记录运行时所存在的硬件和软件环境．一旦获得了此信息，便可贯彻在程序中以实现各种方法间的最优选择．本书所推荐的方法是建立起一张资源表，然后通过动态的运行时间测试或是用户安装步骤来填充其中的值．

5.2.1 资源表

资源表是一个数据结构，用于存贮有关当前计算环境的信息，并且其形式应有助于以后对其诸域的测试．从下面C程序中第41行开始的结构resources便是一个简单资源表的例子．多数域作为unsigned char而不是int来定义．按照所存贮的内容有两种基本类型的域：

第一类域表示的是在一组互斥可能性中选一,例如处理器类型(cpu)、机器名字(machine)域等,为了简化对这些域的设置和测试，为每个可能的内容均定义了常数（见下面程序的19–33行），这些常数在数值上是有顺序的．

第二类域表示可以同时存在的一个或多个资源的存在，例如视频功能域（video）．为此种域所赋的常数在数值上并不是有顺序的，而是1，2，4，8，使得每个资源都有自己的比特．这样就可用一个字节来表示多个功能，而一个系统中可能有若干功能共存．例如，一个系统可以具有正常的彩色图形能力（VCOLOR）及加强的图形能力（VEGA），见36–39行．

```
 1:   /*
 2:
 3:        File:     RESRC.C
 4:
 5:        A C program that obtains and stores the resources of the current
 6:        computing environment
 7:
 8:        This program must be linked with the assembler module RESRCA.ASM.
 9:        To generate the executable program from RESRC.C and RESRCA.ASM:
10:             MSC RESRC;
11:             MASM RESRCA;
12:             LINK RESRC+RESRCA;
13:   */
14:
15:   #include <stdio.h>
16:
17:                                  /* Manifest constants for "resources" structure. */
18:                                            /* resources.cpu                        */
19:   #define    P86        0                  /* Intel 8086 or 8088                   */
20:   #define    P186       1                  /* Intel 80186 or 80188                 */
21:   #define    P286       2                  /* Intel 80286                          */
22:   #define    PNEC       3                  /* NEC V20 or V30                       */
23:
24:                                            /* resources.machine                    */
25:   #define    MUNK       0                  /* Unknown machine                      */
26:   #define    MIBMPC     1                  /* IBM-PC                               */
27:   #define    MIBMXT     2                  /* IBM-XT or Portable PC                */
28:   #define    MIBMJR     3                  /* IBM-PCjunior                         */
29:   #define    MIBMAT     4                  /* IBM-AT                               */
30:   #define    MIBMCONV   5                  /* IBM-CONVERTIBLE                      */
31:   #define    MIBMUN     6                  /* IBM-unknown model                    */
32:   #define    MCOMPAQ    7                  /* COMPAQ                               */
33:   #define    MATT       8                  /* AT&T                                 */
34:
35:                                            /* resources.video                      */
36:   #define    VMONO      1                  /* MDA, HGC, or EGA                     */
37:   #define    VHGC       2                  /* Hercules Graphics Card               */
38:   #define    VCOLOR     4                  /* CGA, EGA, or PGA                     */
39:   #define    VEGA       8                  /* Enhanced Graphics Adapter            */
40:
41:   struct       /* Table of resources present in current computing environment. */
42:        {
43:        unsigned char cpu;              /* Processor type.                          */
44:        unsigned char machine;          /* Brand of machine.                        */
45:        struct dos                      /* Version of MS-DOS.                       */
46:             {
47:             unsigned char major;       /* Major version number = 1, 2, 3, etc.     */
48:             unsigned char minor;       /* Minor version number                     */
49:             }
50:        dos;
51:        unsigned char video;            /* Active display adapter features.         */
52:        unsigned char ncp;              /* Boolean: numeric coprocessor present?    */
53:        unsigned char gameport;         /* Boolean: game port attached?             */
54:        unsigned int memsiz;            /* Total system memory in kilobytes.        */
55:        unsigned char numprn;           /* Number of printers attached.             */
56:        unsigned char numserial;        /* Number of serial ports attached.         */
57:        unsigned char numdisk;          /* Number of diskette drives present.       */
58:        }
59:   resources;
60:
61:                                              /* Assembler functions.              */
62:   unsigned char cur_cpu (void);              /* Returns resources.cpu             */
63:   unsigned char cur_mach (void);             /* Returns resources.machine         */
64:   void dos_ver (struct dos *);               /* Sets resources.dos                */
65:   unsigned char cur_video (void);            /* Returns resources.video           */
66:   unsigned char cur_ncp (void);              /* Returns resources.ncp             */
67:   unsigned char cur_gameport (void);         /* Returns resources.gameport        */
68:   unsigned int cur_memsiz (void);            /* Returns resources.memsiz          */
69:   unsigned char cur_numprn (void);           /* Returns resources.numprn          */
70:   unsigned char cur_numser (void);           /* Returns resources.numserial       */
71:   unsigned char cur_numdisk (void);          /* Returns resources.numdisk         */
```

```
 72:
 73:
 74:  void main (argc, argv)
 75:  int argc;
 76:  char *argv[];
 77:        {
 78:        static char *cpuid [] =  {"Intel 8086/88","Intel 80186/8","Intel 80286",
 79:                                 "NEC V20/V30"};
 80:        static char *machid [] = {"unknown","IBM-PC","IBM-XT","IBM JUNIOR",
 81:                                 "IBM-AT","IBM CONVERTIBLE","IBM-unknown model",
 82:                                 "COMPAQ","AT&T"};
 83:        unsigned char ibmcom;        /* Boolean variable => IBM-compatibility. */
 84:
 85:        resources.cpu = cur_cpu ();                        /* Set resource table. */
 86:        resources.machine = cur_mach ();
 87:        dos_ver (&resources.dos);
 88:
 89:        ibmcom = (resources.machine != MUNK  /* Set IBM-compatibility variable.*/
 90:                 || *argv[1] == '$'                      /* Allow user override. */
 91:                 || *argv[1] == '!');
 92:
 93:                                     /* The following tests use BIOS,          */
 94:                                     /* therefore must have IBM-compatibility. */
 95:        if (ibmcom)
 96:            {
 97:            resources.video = cur_video ();
 98:            resources.ncp = cur_ncp ();
 99:            resources.gameport = cur_gameport ();
100:            resources.memsiz = cur_memsiz ();
101:            resources.numprn = cur_numprn ();
102:            resources.numserial = cur_numser ();
103:            resources.numdisk = cur_numdisk ();
104:            }
105:          else       /* Can't perform tests, therefore must assume the worst. */
106:            {
107:            resources.video = 0;
108:            resources.ncp = 0;
109:            resources.gameport = 0;
110:            resources.memsiz = 0;
111:            resources.numprn = 0;
112:            resources.numserial = 0;
113:            resources.numdisk = 0;
114:            }
115:
116:                                              /* Print out resource table. */
117:        printf ("\ncurrent computing environment\n");
118:        printf ("------- --------- -----------\n");
119:        printf ("current processor:      %s\n",cpuid [resources.cpu]);
120:        printf ("machine make:           %s\n",machid [resources.machine]);
121:        printf ("MS-DOS version:         %1d.%2d\n",
122:                resources.dos.major,
123:                resources.dos.minor);
124:        if (ibmcom)                           /* The following have meaningful  */
125:            {                                 /* values only if IBM-compatible. */
126:            printf ("active display:          %s%s%s%s\n",
127:                    resources.video&VMONO?" MONOCHROME":"",
128:                    resources.video&VHGC?" HERCULES":"",
129:                    resources.video&VCOLOR?" COLOR":"",
130:                    resources.video&VEGA?" EGA":"");
131:            printf ("numeric coprocessor?:   %s\n", resources.ncp?"yes":"no");
132:            printf ("gameport attached?:     %s\n",
133:                    resources.gameport?"yes":"no");
134:            printf ("memory size:            %d kilobytes\n",resources.memsiz);
135:            printf ("number of printers:     %d\n",resources.numprn);
136:            printf ("number serial devices:  %d\n",resources.numserial);
137:            printf ("number diskette drives: %d\n",resources.numdisk);
138:            }
139:
140:        } /* end main */
```

上面是测试当前计算机环境的C程序。

应当设计一个自己版本的资源表，以包含非常适合于特定应用程序的信息。一旦此表中的诸域设置完成，便可使用其值简单地选择出最优子程序。例如：

```
if (resource.machine9==MUNK)  /*机器类型未知*/
                              /*不能使用BIOS*/
printf ("plain message");     /*只有使用慢且笨的prinf*/
else                          /*IBM兼容机可使用*/
printa ("attractive message", 112, 24, 0, 7);/*快速、简单
                                               的BIOS调用*/
```

设置资源表中的数值有两种基本的处理办法:运行时间一是由程序进行动态测试，另一是用户安装。

5.2.2 动态测试

实行测试程序本身可确定大量有关运行计算机环境的信息。此种设置资源表的方法为用户提供了极大的方便，下面的汇编程序是一个很好的例子。

```
page 50,130

;File:      RESRCA.ASM

;Assembler routines that test the current computer environment
;           Called from the C program RESRC.C

                                        ;Manifest constants for resource table.
                                        ;resources.cpu
P86         equ     00                  ;Intel 8086 or 8088.
P186        equ     01                  ;Intel 80186 or 80188.
P286        equ     02                  ;Intel 80286.
PNEC        equ     03                  ;NEC V20 or V30.

                                        ;resouces.machine
MUNK        equ     00                  ;Unknown machine.
MIBMPC      equ     01                  ;IBM-PC.
MIBMXT      equ     02                  ;IBM-XT or Portable PC.
MIBMJR      equ     03                  ;IBM-PCjunior.
MIBMAT      equ     04                  ;IBM-AT.
MIBMCONV    equ     05                  ;IBM-CONVERTIBLE.
MIBMUN      equ     06                  ;IBM-unknown model.
MCOMPAQ     equ     07                  ;COMPAQ.
MATT        equ     08                  ;AT&T.
```

```
                                                ;resources.video
VMONO       equ     01                          ;MDA, HGC, or EGA.
VHGC        equ     02                          ;Hercules Graphics Card.
VCOLOR      equ     04                          ;CGA, EGA, or PGA.
VEGA        equ     08                          ;Enhanced Graphics Adapter.

public      _cur_cpu
public      a01,a02,a03
public      _cur_mach
public      b00,b01,b02,b03,b04,b05,b06,b07,b08,b09
public      _dos_ver
public      c01,c02
public      _cur_video
public      d01,d02,d03,d04,d05
public      _cur_ncp
public      _cur_gameport
public      _cur_memsiz
public      _cur_numprn
public      _cur_numser
public      _cur_numdisk
public      ibm, lenibm, compaq, lencompaq, att, lenatt

                                                ;Masks for BIOS interrupt 11h.
equipment   record  prn:2,r1:1,game:1,ser:3,r2:1,drv:2,mode:2,ram:2,ncp:1,ipl:1

_data       segment word public 'DATA'

ibm         db      "IBM"
lenibm      equ      $-ibm
compaq      db      "COMPAQ"
lencompaq   equ      $-compaq
att         db      "OLIVETTI"
lenatt      equ      $-att
video       dw      ?                           ;Build up video attributes.

_data       ends

dgroup      group   _data
            assume  cs:_text, ds:dgroup

_text       segment byte public 'CODE'


_cur_cpu    proc    near                        ;** returns resources.cpu **
                                                ;unsigned char cur_cpu ();
            assume  cs:_text

            push    sp                          ;** Test for 80286. **
            pop     ax                          ;80286 writes to stack first,
            cmp     ax, sp                      ;then decrements SP.
            jne     a01
            mov     ax, P286
            ret
a01:                                            ;** Test for 80186. **
            mov     cl, 21h                     ;80186+ mask upper 3 bits of
            mov     al, 0ffh                    ;CL for a shift operation.
            shl     al, cl
            jz      a02                         ;0 => upper bits of CL not masked.
            mov     ax, P186
            ret
a02:                                            ;** Test for NEC V20 or V30. **
                                                ;NEC correctly restarts 'rep' instruction
                                                ;with segment override on an interrupt.
            sti                                 ;Make sure interrupts are happening.
            mov     cx, 0ffffh                  ;Allow the maximum time.
            push    si                          ;Set up 'rep' with segment override.
            rep     lods byte ptr es:[si]
            pop     si
            or      cx, cx                      ;If CX=0 then instruction was restarted
            jnz     a03                         ;correctly after an interrupt.
            mov     ax, PNEC
            ret
```

```
a03:
          mov     ax, P86                 ;It's the old 8086/88.
          ret

_cur_cpu  endp

_cur_mach proc    near                    ;** Returns resources.machine. **
                                          ;unsigned char cur_mach ();
          assume  cs:_text

          push    si
          push    di
          push    es
          mov     si, offset dgroup:ibm        ;** Test for IBM logo.
          mov     ax, 0f000h
          mov     es, ax
          mov     di, 0e00eh
          mov     cx, lenibm
          repe    cmpsb
          je      b00
          jmp     b06                     ;Not IBM, go to next test.

b00:                                      ;** Test for PC. **
          cmp     byte ptr es:[0fffeh], 0ffh
          jne     b01
          mov     ax, MIBMPC
          jmp     b09
b01:                                      ;** Test for XT/Portable. **
          cmp     byte ptr es:[0fffeh], 0feh
          jne     b02
          mov     ax, MIBMXT
          jmp     b09
b02:                                      ;** Test for Junior. **
          cmp     byte ptr es:[0fffeh], 0fdh
          jne     b03
          mov     ax, MIBMJR
          jmp     b09
b03:                                      ;** Test for AT. **
          cmp     byte ptr es:[0fffeh], 0fch
          jne     b04
          mov     ax, MIBMAT
          jmp     b09
b04:                                      ;** Test for Convertible. **
          cmp     byte ptr es:[0fffeh], 0f9h
          jne     b05
          mov     ax, MIBMCONV
          jmp     b09
b05:                                      ;** Unknown IBM model. **
          mov     ax, MIBMUN
          jmp     b09
b06:                                      ;** Test for COMPAQ. **
          mov     si, offset dgroup:compaq
          mov     di, 0ffeah
          mov     cx, lencompaq
          repe    cmpsb
          jne     b07
          mov     ax, MCOMPAQ
          jmp     b09
b07:                                      ;** Test for AT&T. **
          mov     si, offset dgroup:att
          mov     di, 0c050h
          mov     cx, lenatt
          repe    cmpsb
          jne     b08
          mov     ax, MATT
          jmp     b09
b08:                                      ;** Unidentified machine. **
          mov     ax, MUNK
b09:
          pop     es
          pop     di
          pop     si
          ret
```

```
_cur_mach   endp


_dos_ver    proc     near                 ;** Assigns values to 'resources.dos', **
                                          ;void dos_ver (version);
                                          ;struct dos *version;
            assume   cs:_text

sframe      struc                         ;Template to access stack frame.
bptr        dw       ?
ret_ad      dw       ?
parm1       dw       ?
sframe      ends
frame       equu     [bp - bptr]          ;Base for accessing stack frame.

            push     bp                   ;Set up base pointer to access frame.
            mov      bp, sp

            mov      ah, 30h              ;Call dos to get version number.
            int      21h
            mov      bx, frame.parm1      ;Move 'dos' structure address into BX.
            cmp      al, 0
            jne      c01
            mov      byte ptr [bx], 1     ;Use BX as a pointer to access the
            mov      byte ptr [bx+1], 0   ;actual 'dos' structure fields.
            jmp      c02
c01:        mov      byte ptr [bx], al
            mov      byte ptr [bx+1], ah
c02:
            pop      bp
            ret

_dos_ver    endp


_cur_video  proc     near                 ;** Returns resources.video **
                                          ;unsigned char cur_video ();
                                          ;Note: this function uses the BIOS and
                                          ;therefore should be called only if
                                          ;IBM compatibility is established.
            assume   cs:_text

            push     si
            push     di
            mov      video, 0             ;Initialize video flag w/ no attributes.

                                          ;** Test MONO vs. COLOR. **
            mov      ah, 15               ;Get current mode function,
            int      10h                  ;BIOS video services.
            cmp      al, 7                ;7 => monochrome.
            je       d01
            or       video, VCOLOR        ;Color display (= CGA, EGA, or PGA).
            jmp      d04                  ;Go to test for EGA.
d01:
            or       video, VMONO         ;Mono display (= MDA, HGC, or EGA).

                                          ;** Test for HERCULES. **
                                          ;This routine supplied courtesy of.
                                          ;Hercules Computer Technology.
            mov      dx, 03bah            ;Display status port.
            xor      bl, bl               ;Clear counter.
            in       al, dx               ;Read port.
            and      al, 80h              ;Mask off all bits except 7.
            mov      ah, al               ;Save bit 7 in AH.
            mov      cx, 8000h            ;Set loop counter.
d02:
            in       al, dx               ;Read port again.
            and      al, 80h              ;Mask out bit 7.
            cmp      al, ah               ;Test if bit has changed.
            je       d03                  ;Bit not yet changed.
            inc      bl                   ;Bit changed, increment counter.
            cmp      bl, 10               ;Want to see it change 10 times.
            jb       d03                  ;Need to see more changes.
            or       video, VHGC          ;Yes, it is a HCG.
            jmp      d04                  ;Go on to EGA test.
d03:
```

```
        loop    d02                     ;Continue testing for changes.

d04:                                    ;** Test for EGA. **
        mov     bh, 0ffh                ;Impossible return value.
        mov     cl, 0fh                 ;Impossible return value.
        mov     ah, 12h                 ;BIOS alternate select function.
        mov     bl, 10h                 ;Request EGA information.
        int     10h                     ;BIOS video services interrupt.

                                        ;Three part test for EGA.
        cmp     bh, 01h                 ;BH should have 0 or 1 for color/mono.
        ja      d05                     ;If invalid value, not EGA.
        cmp     cl, 0fh                 ;Maximum switch setting.
        ja      d05
        or      video, VEGA             ;If control reaches here, must be EGA.
d05:
        pop     di
        pop     si
        mov     ax, video
        ret

_cur_video endp


_cur_ncp   proc    near                 ;** Returns resources.ncp **
                                        ;unsigned char cur_ncp ();
                                        ;Note: this function uses the BIOS and
                                        ;therefore should be called only if
                                        ;IBM compatibility is established.
        assume  cs:_text
        int     11h                     ;BIOS equipment service --
        and     ax, mask ncp            ;     bit 1 = 0 : no num. coprocessor
        mov     cl, ncp                 ;     bit 1 = 1 : num. coprocessor found
        shr     ax, cl                  ;Move significant bit to rightmost position.
        ret

_cur_ncp   endp


_cur_gameport proc near                 ;** Returns 'resources.gameport'. **
                                        ;unsigned char cur_gameport ();

                                        ;Note: this function uses the BIOS and
                                        ;therefore should be called only if
                                        ;IBM compatibility is established.
        assume  cs:_text

        int     11h                     ;BIOS equipment service --
        and     ax, mask game           ;     bit 12 = 0 : no game port
        mov     cl, game                ;     bit 12 = 1 : game port
        shr     ax, cl                  ;Move significant bit to rightmost position.
        ret

_cur_gameport endp


_cur_memsiz proc near                   ;** Returns 'resources.memsiz' **
                                        ;unsigned int cur_memsiz ();
                                        ;Note: this function uses the BIOS and
                                        ;therefore should be called only if
                                        ;IBM compatibility is established.
        assume  cs:_text

        int     12h                     ;BIOS memory size service.
        ret                             ;Returns kilobytes of memory in AX.

_cur_memsiz endp


_cur_numprn proc near                   ;** Returns 'resources.numprn' **
                                        ;unsigned char cur_numprn ();
                                        ;Note: this function uses the BIOS and
                                        ;therefore should be called only if
                                        ;IBM compatibility is established.

        assume  cs:_text
```

```
        int     11h                     ;BIOS equipment service.
        and     ax, mask prn            ;Bits 15:14 = number of printers.
        mov     cl, prn
        shr     ax, cl
        ret

_cur_numprn endp

_cur_numser proc near                   ;** Returns 'resources.numserial' **
                                        ;unsigned char cur_numser ();
                                        ;Note: this function uses the BIOS and
                                        ;therefore should be called only if
                                        ;IBM compatibility is established.
        assume  cs:_text

        int     11h                     ;BIOS equipment service --
        and     ax, mask ser            ;       bits 11:10:9 = num serial devices.
        mov     cl, ser
        shr     ax, cl
        ret

_cur_numser endp

_cur_numdisk proc near                  ;** Returns 'resources.numdisk' **
                                        ;unsigned char cur_numdisk ();
                                        ;Note: this function uses the BIOS and
                                        ;therefore should be called only if
                                        ;IBM compatibility is established.
        assume  cs:_text

        int     11h                     ;BIOS equipment service --
        test    ax, mask ipl            ;       bit 0 => diskette drive(s) present
        jnz     j01                     ;       bits 7:6 = num diskette drives
        xor     ax, ax                  ;               00 = 1
        ret                             ;               01 = 2
j01:    and     ax, mask drv            ;               10 = 3
        mov     cl, drv                 ;               11 = 4
        shr     ax, cl
        inc     ax

        ret

_cur_numdisk endp

_text   ends                            ;End of code segment.
        end
```

上面是测试当前计算机环境的汇编程序。

程序开始时首先调用不使用BIOS服务，并能在任何MS-DOS 机器上运行的那些汇编程序函数：cur_cpu，cur_mach，dos_ver（85-87行）。

注意：如果第一个命令行参数以 i 或 I 开头，则设置 IBM 兼容标志的表达式设置为真（true）。此特点使用户可不使用由程序确定的兼容度类型。为用户提供决定权是很重要的，因为由本例所能确定的机器类型毕竟很有限（见 90 和 91 行）。

一旦域 resources.machine 被设置（由调用 cur_mach），程序便可用其值去设置有关 IBM 兼容性的标志ibmcom。本例中 resources.machine 可表明非 IBM 兼容的唯一值是 MUNK（未知机器类型，见 89 行）。

一旦确认了 IBM 兼容性，便可调用使用 BIOS 功能的测试函数。如果这些函数不能被调用（且无其他方法可用于找出信息），则程序假定判断错误，并将所有资源域设置为 0（95—114 行）。最后程序打印出结果:机器中找到资源的清单。上面给出了可被 C 程序调用的汇编函数清单，此文件开始的定义与 C 文件中的常数定义一样。下面介绍这些函数。

▲ _cur_cpu

函数 _cur_cpu 返回当前处理器的识别码，并用于设置 resources.cpu 的值。它使用下面的一系列测试在一组可能的处理器间进行识别。由于新的处理器（例如 80386）用得已很普遍，故应加进一些其他测试以检测它们是否存在。

首先测试 80286。80286 处理器在实现 push 指令时使用了与以前的 Intel 模式有点不同的机构。这类处理器先将数值写入内存，然后减堆栈指针。

接着测试 80186。80186 处理器在执行移位（chift）操作时屏蔽掉 CL 寄存器的上 3 个比特（可能是为避免不合理数字的无意义移位操作）。如果检测到此机构，便假定处理

器是 80186.

然后测试 NEC V20/V30. 此测试基于其字符串操作，如 Lods，是带有 rep 前缀及段优生（ES:）的. 与 Intel8086 及 8088 不同，NEX 处理器在中断后会正确恢复此指令的执行（8086/88 采用段优先重新开始此指令，但忽略了 rep 前缀，会使进程过早结束，而 CX 中仍留下非 0 值）.

如果上述测试均失败，此函数便假定处理器是 Intel8086 或 8088.

此例仅用于说明识别处理器的一种基本方法. 其实不但可加入对新处理器（如 80386）的测试，即使 8086 与 8088 之间亦有差别，如果这对程序的工作很重要，便可增加进一步的测试功能.

▲ _cur _mach

函数 _cur _mach 返回有关计算机的识别码，用于设置 resources.machine. 它只是简单地测试 ROM 中包含机器标识的地址. 如果在正确地址处（F000h:E00Eh）找到了 IBM 标识，测试另一个字节（在 F000h:FFFEh 处）以确定 IBM 机型. 如果未找到这些标识，返回 MUNK 表示机器类型未知. 本例中包括几种机器，实际的应用程序应进行更多的测试. 随着新的 IBM 机型的推出，此段程序应更新.

▲ _dos _ver

过程 _dos _ver 使用了 DOS 功能 30h 返回操作系统的当前版本. 由于此 DOS 功能对 DOS1.X 版本不成立，故对 1.X 版本不适用，但此时会返回 0. 注意，_dos _ver 并不返回一个数值，而是直接去改变其地址作为参数传递的一个

结构。由于该结构的两个域均是一个字节的，可假定它们在内存中是连续的。但由于C语言常在结构元素之间有间隙，故不要使此假定成为一般情况。

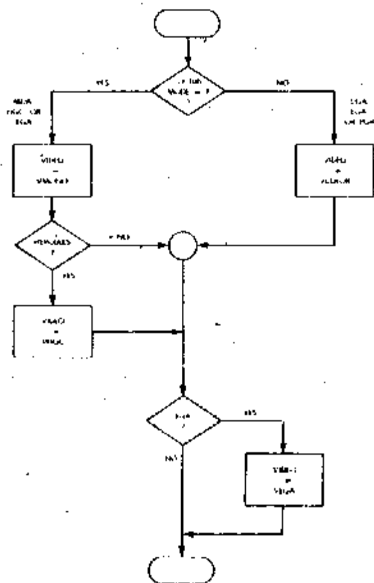


图 5-1 测试活动的显示适配器

▲_cur_video

函数_cur_video 返回当前活动的显示适配器，用于设置 resource．video。没有再去检测是否存在有第二个不活动的适配器。此测试的主要目的是确定程序当前可用的视频显示方式，以及视频刷新缓冲区在内存中的位置。图 5-1 是测试的逻辑流程图。第一个测试使用 BIOS 显示服务中断

10h，功能15，以确定当前显示方式是否为7。如果方式为7，则单色比特作用（使用VMONO对属性字节作OR操作），否则彩色比特起作用（使用VCOLOR作OR操作）。

单色比特（VMONO）指示该系统可以支持视频文本显式方式7，并将显示单色属性（如反相显示等）。如果单色比特作用，适配器可能是：标准单色显示适配器（MDA）、Hercules图形卡（HGC）、接在单色显示器上的EGA。

彩色比特（VCOLOR）表示系统能够使用文本和图形方式0到6。如果彩色比特作用，适配器可能是：标准彩色图形适配器CGA、EGA或专用图形适配器PGA。

第二个测试用于Hercules图形卡HGC。如果系统是单色的（VMONO为ON），再测试看是否是Hercules卡。测试时查看"显示方式状态"端口3BAh的比特7在回路执行期间（8000h次重复）是否至少变化10次。对于Hercules卡，此比特每次垂直回扫时变成0，而IBM卡不使用此比特。

第三个测试用于EGA。EGA测试调用于EGA系统的BIOS中断服务程序：中断10h，功能12h（"alternate select function"），子功能10h（"return EGA information"）。然后检查返回的数据是否合法，此功能在BH中返回0和1，分别表示彩色或单色方式；CL中的数值范围0到Fh表示当前的开关设置。这两个寄存器初始化为FFh，功能被调用后检查这些寄存器看其值是否在合法范围内。这两个寄存器中重新赋给合法数据，则有足够理由认为存在EGA适配器。

▲ _cur_ncp

函数_cur_ncp用于检测是否存在有数学协处理器，

返回 1 说明存在有 8087 或 80287 协处理器，返回 0 说明不存在。它用于设置 resource.ncp.此功能简单地调用 BIOS 设备列表服务（中断 11h）。寄存器 AX 返回值的比特 1 表示数学协处理器存在与否。此比特移位至最右边并返回其值。对于 IBM-PC／XT，此测试有可能出错，因为其 BIOS 直接根据开关设置获得此值，而开关有可能是拨错了的。使用 X87 指令的方法更高级。

▲ _cur_gameport

如果游戏适配器已安装，则函数_cur_gameport 返回 1，否则返回 0。其工作原理与_cur_ncp 一样。

▲ _cur_memsiz

函数_cur_memsiz 使用 BIOS 中断 12h 内存测试服务程序。返回的值以 K 计，直接根据开关设置得到（对于 PC／XT）。亦可根据段前缀来确定可用内存量。

▲ _cur_numprn

函数_cur_numprn 亦使用 BIOS 中断 11h 返回当前安装的打印机数量。只报告接在独立并行端口的打印机，共享一个端口的或使用了切换设备的不算。

▲ _cur_mumser

函数_cur_numser 返回已安装的串行设备数，例如 modem 调制解调器及串行打印机。使用方法与_cur_numprn 一样。

▲ _cur_numdisk

函数_cur_numdisk 返回已安装的软盘驱动器数（不包括硬盘或 RAM）。使用的基本方法与前两个函数相同。

5.2.3 用户安装程序的使用

设置资源表内容的另一种方法是由用户提供必要的信

息. 许多应用程序使用此方法，它有几个优点. 首先，此方法可以简化软件开发，不需要研究、设计测试程序. 另外在某些场合下用户安装更为可靠，例如对于新型的 IBM 兼容机,或是有些难以检测的因素，例如打算在 Windows 或其他多任务环境下运行的程序. 最后，用户安装程序由于不需要测试代码段，故程序更小、运行更快.

用户安装方法也有几个缺点. 首先，此方法将重担交给了用户，而很多用户并无很多技术知识. 另外只要改变了机器或外设，就必须重新安装此程序. 例如改变了显示适配器后，用户必须记住哪些程序需要人工重新安装，没有进行重新安装的程序会造成"死机".

许多场合下，最好的办法是包括动态的运行时间测试，但也允许用户指定一部分功能. 这种方法在多数情况下可自动工作，但仍允许用户对程序行为有最终的控制权.

无论是安装功能还是取消程序设置，从用户获得信息的方法包括:程序运行时使用命令行开关;环境变量;作为主应用程序一部分或是一个独立程序的安装子程序（访问命令行及环境的信息见有关程序段前缀的章节).

一旦从用户处收集到信息，就可用几种不同的办法来存储和处理. 首先，可使一个"补丁"程序，将取得的值直接写入可执行文件，例如直接写人数据段中的资源表. 其次，可借助获得的信息选择几种不同版本程序之一，将其拷贝到工作磁盘上. 第三种方法是将获得的信息写入一个配置文件，程序每次运行时均要读此文件. 最后一种方法是将与机器有关的代码段单独写成一个小的、独立的设备驱动程序,给每种设备提供一个单独的驱动程序. 用户选中了一种特殊设备（例如 Hercules 显示适配器），则拷贝相应的驱动程序即

可。

5.3 使用可提供的资源

一个程序获得了有关运行时环境的信息后，软件兼容性的许多规则成为相对而不是绝对的——取决于提供的一组资源。如何使用已收集到的有关资源的信息，此问题涉及到许多方面，例如，特定的运行时间资源、所用的机器类型、操作系统版本以及如 Microsoft Windows 等操作环境的存在与否等等。

5.3.1 使用特定资源

资源表中各域涉及到特定的资源，如果该资源存在，便可以使用,如果不存在，便不能使用。例如，如果资源表中域 resources.cpu 表明的是 8086／88 处理器，便不能使用专用于 80286 的指令（例如 pusha）。如果域 resources.video 表明的是单色系统，则为所有视频输出只可以赋给显示属性而不是颜色。resource 结构的其他域有 ncp，gameport，memsiz，numprn，numserial，numdisk。

5.3.2 使用与机器类型有关的信息

资源表的机器类型域 machine 的含义不那么明显。特定的机器牌子有特定的资源，为简单化起见，可将机器归为两大类:非 IBM 兼容机和 IBM 兼容机。

5.3.2.1 非 IBM 兼容机

如果程序无法识别机器，或是已知该机器是非 IBM 兼容机，且你又没有关于其资源的其他信息，则只好将程序限制于可用在任何 MS-DOS 机器上的功能，且不能使用下列功能:

- IBM 扩展字符集（ASCII 代码 127-255）。
- 光标键或功能键（应由程序提供另外的键）。

· BIOS 中断服务程序.

· 存贮在低内存地址区的系统数据.

· 通过 I/O 端口作直接的硬件访问.

· 直接向视频内存区输出.

· 比特映照的图形.

所有输入/输出均应通过 DOS 功能执行，因为这是唯一可用的资源.

这些限制与高性能程序设计是矛盾的，但随着更多高兼容度机器的出现，这样的限制越来越不必要.

5.3.2.2 IBM 兼容机

一旦确认是 IBM 兼容机，便有许多高性能资源可用. 确切的资源清单取决于特定的机器型号. IBM 建议（但并不保证）在 IBM 计算机系列中将保持下面功能. 使用这些功能的应用程序可保持 IBM 兼容性. 因此下面所列内容应由 IBM 兼容机提供.

· BIOS 中断服务程序. 但应通过中断向量表而不是由绝对地址来访问.

· 中断向量表. 一个向量可由指向另一个子程序来替换. 但该处理程序应保存原始地址.

· BIOS 数据变量存贮在 400h—48Fh 的低内存地址. 多数域应保持其当前的意义. 但只要有可能，最好通过 BIOS 服务程序来访问这些数值.

· 视频内存区 B0000h 用于文本方式 7，B8000h 用于彩色图形适配器方式（文本或图形）. 一般来说，给定显示方式的缓冲区地址应保持相同，如果使用了新地址，应与新的方式号相关联（例如 EGA 提供了新的方式）.

·下列I/O端口:

021h	8259A 中断控制器参考寄存器
040h,042h, 043h	8253 定时器,通道 0 和 2. 程序块不能修改端口 41h. 将维持不变的输入频率 1.19MHz
061h	声音控制 8255A-5
201h	游戏控制
3BAh 及 3DAh	单色及彩色视频状态寄存器
2F8h-2FFh 3F8h-3FFh	2F8h 和 3F8h 开始的异步通信适配器端口可编程后成为中断驱动的通信,但仍使用 BIOS 服务程序设置通信参数

5.4 MS-DOS 各版本之间的差别及其兼容性问题

自1981 年 MS-DOS 第一版本问世以来, 操作系统在适应硬件环境、 修补缺陷以及一般性操作性能的改进等方面都得到了提高. 这些提高虽使其能力大大地增加了, 但同时也带来了一些新问题. 因为某些新的功能与 MS-DOS 的旧版本不兼容. 为了使改进后的新版本具有较理想的前景, 本节指在帮助读者在使用某一特定的 MS-DOS 版本开发应用程序时, 能对 MS-DOS 不同版本之间的兼容性有所考虑. 本节内容对使用汇编语言开发应用程序尤其有用.

除为程序员设计的工具命令 (如 DEBUG, LINK) 外, 新改进的 MS-DOS 命令相对说来对程序员没有什么作用. 对程序员有特殊意义的改进有 MS-DOS 中断、 功能调用、错误代码、软件磁盘格式以及文件控制等. 这些领域的问题之所以常常涉及到, 是由于象功能调用这类的问题在某给定的 MS-DOS 版本的所有实现中都存在.

而其它一些范围的问题(如内存映照)一般说来就不大涉

及，因为它们通常随 MS-DOS 目标硬件环境的变化而变化。例如对于 IBM PC 及其高度兼容机器来说，硬件体系结构完全不同的系统，其MS-DOS实现的内存映照体制也不相同，甚至在某些相对来说较"标准"的领域（如中断等）里也存在着关键性的差别。因此，在开发应用程序时程序员需懂得做什么和不做什么。如果正在开发的程序要求尽可能大的适应环境，则了解这些差别特别重要。记住，存在着不同的与机器相关的 MS-DOS 版本，并且有许多机器是有不同的硬件体系结构与不同的 MS-DOS 实现。如果开发的程序打算在所有的 MS-DOS 实现下运行，则仅遵照《MS-DOS 技术手册》是不够的。

本节内容并无取代《MS-DOS 手册》之意，而是通过对 MS-DOS 各版本之间的差别进行概述来补充 MS-DOS 所有版本的技术手册。本节内容是按提供的 MS-DOS（从 1.0 到 3.1）所有版本之间的差别来划分专题的。其内容包括对程序员有关的技术信息以及建议步骤、要避免的事宜等技巧。

5.4.1 版本兼容性的一般概念

程序员可利用不同程度的兼容性，多数情况下总希望获得全兼容性。然而，由于一般都喜欢设计精巧的程序，我们常常利用建立在 MS-DOS 实现中新改进过的功能，例如很漂亮的屏幕显示功能或特殊用途的中断等，并且常常忘记不兼容所造成的后果。经常采用折衷的方法来选择兼容度。下列规则对获得全兼容性很有用。

(1) 除那些作为 MS-DOS 中断而设置的中断指令外，在任何情况下都不要采用 8086 系列的 INT（中断）指令。

(2) 不向任何绝对内存地址写数据，应让 MS-DOS 去处理内存的使用.

(3) 不采用 8086 系列的 I/O 指令.

(4) 避免采用那些仅由 80188、80186 和 80286 微处理机提供的指令. 这些指令如下:

PUSH immediate (push immediate)

PUSHA (push all registers)

POPA (pop all registers)

SHR>1 (shift right with immediate value greater than 1)

SHL>1 (shigt Left with immediate value greater than 1)

IMUL dest_reg, source immdiate (multiply iom-mediate signed integer)

INS source-string, port (in string)

OUTS port, dest_string (out string)

ENTER (enter procedure)

LEAVE (Leave procedure)

BOUND (detect value out of range)

避免使用指令 POP CS, 因为此指令仅对 8088 和 8086 微处理器有用. 注意, 8086 系列的不同处理器之间在操作上的其他差别.

(5) 如果在开发程序时用的机器中,ROM内储存有子程序, 不要调用它们, 甚至不要试图去读它们.

(6) 不要使用仅由 MS-DOS1.0 版本以后的版本所支持的 MS-DOS 功能调用.

(7) 永远确保向屏幕写的信息仅由标准的 ASCII 字符

(00 到 7F,十六进制）组成。避免使用任何其它字符，如 IBM PC 及兼容的扩展字符集。避免使用比特映照的图形。

当必须打破前五条规则时，最好连同第六条一起打破。因为第一个选择很可能是为一个不兼容的机器写一个设备驱动程序，以及当可安装的设备驱动程序仅由 MS-DOS2.0 版本及其更新版本支持时，就会发现程序所使用的功能调用已不是由 MS-DOS1.0，1.1 版本支持的了。当需要（或想要）打破第七条规则时，应为目标机器或"通用"安装程序（该程序可为应用程序适配各种不同终端和监视器）写一个设备驱动程序。但安装程序至少必须遵守第七条规则。

由于设备驱动程序可能是解决不兼容性问题的办法之一，我们发现我们自己已经打破了第六条规则（此规则提供了另一种需要考虑的兼容水平）。由于不是所有的 MS-DOS 版本都提供了所希望或需要的特定的功能调用，所以，在许多情况下，你都倾向于想打破第六条规则。例如，当应用程序广泛地采用了树结构目录时，就很可能要使用 39h 至 3Bh 功能调用。在这种情况下，兼容水平就会被限制在 MS-DOS2.0 或更新的版本之中，而被排斥在 1.0 和 1.1 版本之外。与之相类似，如果程序需要利用 MS-DOS3.1 版本所支持的网络功能时，该程序就不能与 MS-DOS1.0 至 2.1 版本相兼容。

别忘记在源代码或文档中要说明程序的兼容度（最好在两者中都说明）。如果是商用程序，那么在软件包和广告中都要清楚地说明其兼容限度（或没有限制）。

如果开发的程序打算运行在任何一个 MS-DOS 版本下（但其中包含一些可以有选择地在某特定的 MS-DOS 版本

下执行的子程序），使用功能 30h（获得 DOS 版本号）来控制某个子程序是否被执行．尽管此功能只有 MS-DOS2.0 或更新的版本才能提供，但只要按照《MS-DOS 手册》的行使 DOS 功能中所提到的预备措施步骤，此功能也可在使用 1.0 和 1.1 版本时无副作用地执行．

使用此功能时，将 30h 装入 AH 寄存器中．当执行 INT21h 时，主要版本号在 AL 寄存器中返回，次要版本号在 AH 寄存器中返回．如果 AL 包含 00，就可推断 MS-DOS 的版本不是 1.0 就是 1.1．AL 中的号码表示版本号．例如，当使用 MS-DOS2.00 版本时，在 AL 中为 0.2．AH 中为 0.0．当使用 MS-DOS3.10 版本时,在AL 中为 03，AH 中为 10．甚至当不需要控制某子程序的选择执行时，而用户又试图在一个不兼容的 MS-DOS 版本下运行该程序，此功能仍允许显示出一条用户友好的信息．

5.4.2 高级语言的考虑与MS－DOS中断

如果用高级语言写程序，必须弄明白所用的特定的编译程序或解释程序的说明．如果产品说明书指出编译程序或解释程序仅在某特定的 MS-DOS 版本下运行时，则经过编译或解释的程序可能不能在某个更早的版本下工作．BASIC 解释程序（比如 Microsoft／IBM BASIC 和 GWBASIC）尤其如此．因为这些解释程序的新版本常常伴随 MS-DOS 的新版本推出之．

▲ MS-DOS 中断

为 MS-DOS 使用而定义的软件中断，除中断 2Fh 以外，对所有版本都是一致的，而中断 2Fh 是在版本 3.0 后加入的．表 5.1 列出了这些中断．

表5.1 MS-DOS中断

<table>
<tr><th colspan="2">中断</th><th colspan="6">MS-DOS版本</th></tr>
<tr><th>INT</th><th>说明</th><th>1.0</th><th>1.1</th><th>2.0</th><th>2.1</th><th>3.0</th><th>3.1</th></tr>
<tr><td>20</td><td>程序结束</td><td colspan="6" rowspan="8">有</td></tr>
<tr><td>21</td><td>功能请求</td></tr>
<tr><td>22</td><td>结束地址</td></tr>
<tr><td>23</td><td>Ctrl_BREAK 退出地址</td></tr>
<tr><td>24</td><td>致命错误处理程序向量</td></tr>
<tr><td>25</td><td>绝对磁盘读</td></tr>
<tr><td>26</td><td>绝对磁盘写</td></tr>
<tr><td>27</td><td>结束但驻留</td></tr>
<tr><td>28</td><td>未使用</td><td colspan="6">MS-DOS 内部使用</td></tr>
<tr><td>29</td><td></td><td colspan="6">(保留)</td></tr>
<tr><td>2A</td><td>MS-NET 访问</td><td colspan="3">无</td><td colspan="3">有</td></tr>
<tr><td>2B
至
2E</td><td>未使用</td><td colspan="6">(保留)</td></tr>
<tr><td rowspan="2">2F</td><td>打印机</td><td colspan="2" rowspan="2">无</td><td>有</td><td colspan="3">///</td></tr>
<tr><td>多路中断</td><td>xxx</td><td colspan="3">有</td></tr>
<tr><td>30
至
3F</td><td>未使用</td><td colspan="6">(保留)</td></tr>
</table>

许多机器还有几个中断没有列在上表中．这些中断为特殊用途而定义，如访问 BIOS 子程序或串行通信端口通信的中断．不要把这些中断与 MS-DOS 使用的中断相混淆．只有那些《MS-DOS 技术手册》中介绍的中断才是真正的 MS-DOS 中断．为了保持与 MS-DOS 所有实现的兼容性，要避免使用任何非真正的 MS-DOS 中断．

5.4.3 功能调用

用汇编语言编程时，使用功能调用也许是兼容性中最重要的因素．因为几乎所有通常由 MS-DOA 执行的操作事件均可由功能调用来做，可以避免使用中断（除 INT21 外）及 BIOS 调用．通过使用 MS-DOS 功能调用，还可以

消除包括在程序中的某些类型（如文件操作用）的子程序。如果程序的快速执行不是至关重要的，就可让 MS–DOS 通过功能调用的办法来执行所有的标准操作。MS–DOS 执行功能调用的速度在大多数情况下是足够的了。

5.4.3.1 执行功能调用的标准方法

MS–DOS 第一版本问世时提供了两种执行功能调用的方法.第一种方法适用于采用所有 MS–DOS 版本的,其步骤如下:

(1)把 AX、BX、CX 和 DX 寄存器的内容存入堆栈。

(2)把功能号记入 AH 寄存器中。

(3)根据需要将其它数据置入特定功能指定的寄存器中。

(4)执行 INT21h 指令。以后可由程序读人或使用的变量数据在指定的寄存器中返回。有些功能不能返回任何信息。

(5)如果需要，根据对刚执行的功能返回的数据再执行某些需要的操作。

(6)恢复原来寄存器的初始内容。

以上介绍的方法是针对所有 MS–DOS 版本的。第二种方法将在下节介绍。

5.4.3.2 在兼容方式下执行功能调用

MS–DOS 提供的、与其它操作系统相兼容的第二种方法，尤其适合于 CP／M–80 和 CP／M–86。不过此方法实际上并不能提供在 MS–DOS 下运行 CP／M 程序的能力。它仅仅简化了 CP／M 程序向 MS–DOS 程序的变换。然而，这样也许必须重改很多功能号。此方法只在 MS–DOS 功能 0 至 24h 中起作用。在执行某些功能调用时，还可能遇到某些寄存器用法上的困难。因此，除在某个程序完全转换前要对它测试外,当然应避免采用此方法。MS–DOS 要求

采用第二种方法的功能调用需遵守以下步骤:

(1)通过把 AX，BX，CX 和 DX 寄存器推入堆栈来保存其内容。

(2)将功能号置入 CL 寄存器中（仅可使用功能号 0至24h）。

(3)按照欲执行的特定功能；将其它数据置入指定的寄存器中。

(4)执行对当前代码段地址 5 处的段内调用，此地址包含对 MS-DOS 调度程序的一个长调用。

(5)根据所执行的功能，以后将由应用程序读入或使用的变量数据在指定的寄存器中返回。有些功能不能返回任何信息。

注意，此方法总是抹掉 AX 寄存器中的内容。其它所有寄存器所受的影响与采用标准功能调用方法时所受的相同。

(6)恢复这些寄存器的初始内容。

5.4.3.3 新的功能调用方法（用于 MS-DOS2.00 以及更新的版本）

MS-DOS2.00 版本介绍了发出功能调用的第二种方法。此方法还可用于更新的（2.00 版本以上)版本中,但不能用于以前（2.00 版本以下）的版本中。按照下列方式可实现第三种方法:

(1)通过将 AX，BX，CX 和 DX 寄存器推入堆栈的方式保存其内容。

(2)将功能号置入 AH 寄存器中。

(3)将其它数据置入欲执行特定功能所指定的寄存器中。

(4)向程序段前缀偏移 50h 处发出一个长调用。

(5)根据所执行的功能，以后应用程序可读人或使用的变量数据在指定的寄存器中返回.某些功能不返回任何信息.

(6)由堆栈弹出操作恢复寄存器的原始内容.

如 MS-DOS3.10 版本的发布所言，Microsoft 和 IBM 都建议不采用此方法.既然如此，为何还介绍它呢?此方法的一种可能用途也许可作为该问题的解释. PSP（程序段前缀）中的偏移 50h 通常包含一个 INT21h 指令.通过使用以上介绍的方法，程序员可通过一个地址实现对所有 MS-DOS 功能代码的访问（不包括其他中断).通过改变位于偏移 50h 中的指令，可重定向所有程序的 MS-DOS 访问. Microsoft 为实现多任务而放弃这种方法的企图只有 Microsoft 公司自己知道.

5.4.3.4 由不同版本支持的功能

MS-DSO 功能在新版本中是有增补的.其功能范围可分成多个"功能"组.恰巧这些"功能"组具有能确定 MS-DOS 各版本的倾向（但也不总是如此).这些"功能"组在下面介绍.

(1)程序终止组

0 是本组中唯一的功能.此功能与 INT12h 中断几乎一样.尽管 INT20h 在所有的 MS-DOS 实现中被定义为程序终止，但仍应使用功能 0 以避免 INT 指令的使用.另一个需要注意的问题是,MS-DOS2.0 或更新版本的手册中,建议将功能 4Ch（终止一个过程，也叫 EXIT）作为终止程序的更好方法来使用.但是，此功能在 2.00 以前的各版本中不存在.

按该手册中的建议来终止程序是个好主意.我们十分恳切地建议,在终止MS-DOS2.00 以及更新版本的程序时使用

功能 4Ch。如果希望程序在所有的版本下运行，用 Get DOS 版本功能（30h）来确定使用哪个程序终止代码：使用 MS-DOS1.0 及 1.1 版本的功能 0，或者使用功能 4Ch 于所有其它版本。

(2)标准字符设备输入／输出组（01h—0Ch）

本组包括功能 01 至 0Ch。它们用于从键盘输入，输出到控制台显示器及打印机上，以及从辅助（逻辑的）设备输入或输出。这些功能以相同的方式在 MS-DOS 的所有版本中使用，并且在本质上与 CP／M 中同等范围的功能相似。

(3)标准文件管理组（0Dh—24h，27h—29h）

本组包括功能 0Dh 至 24h 以及 27 至 29h。使用这些功能来管理文件可与 MS-DOS 所有的版本兼容。其中某些功能还与 CP／M 中使用的同等范围的功能相似。尽管在 MS-DOS2.00（下文将介绍）版本中介绍了一些较为新颖的文件管理功能，在使用它们时仍需要仔细考虑其中所涉及的兼容问题。

(4)标准非设备功能组（25h，26h，2Ah—2Eh）

本组功能包括 25h，26h 以及 2Ah 至 2Eh。注意，功能 2Eh 是由 MS-DOS2.00 以前的各版本所支持的最高功能。这些功能执行与设备无关的各种不同的任务：检查和设置当前时间及日期，设置中断向量，生成新的程序段以及设置和重新设置验证开关。所有这些功能是 MS-DOS 专用的，在 CP／M 中无等同的功能。它们在 MS-DOS 所有的版本中都能很好地工作。但需特别注意 25h（设置中断向量），在执行此功能前需做两件事：中断处理子程序的地址必须装入 DX 寄存器和数据段中（DS:DX），中断号必须装入 AL 寄存器中。由于此功能与中断有关，故在使用时需小心，否则

将导致应用程序与其他的 MS-DOS 实现以及硬件环境不兼容。

(5) 扩展的(一般性)功能组(2Fh-38h,4Ch-4Fh,54h-57h, 59h-5Fh, 62h)

本组功能跨出了 MS-DOS2.00 及 3.10 版本的界线。功能 59h 至 5Ch 以及 62h 仅存在于 3.00 及其更新的版本中。功能 5Eh 至 5Fh 仅存在于 3.10 及其更新的版本中。这些功能都不能用于 MS-DOS2.00 以前的各种版本。此外，功能 32h, 34h, 37h, 50h 到 53h, 55h, 58h, 5Dh, 60h 以及 61h 对于 MS-DOS3.10 版本来说是保留的（未作使用定义）。存在于所有版本中的功能使用方法是一样的。但以下情况例外。

• 功能 38h（与国家有关的信息）。在 MS-DOS3.00 以及更新的版本下，此功能可用来设置并检索与国家有关的信息。但是，在版本 2.00 至 3.00 中，此功能则只能用来检索信息。

• 功能 44h（设备[IOCTRL]的 I/O 控制）在 MS-DOS3.00 版本中有两个用来支持设备驱动程序的新的辅助参数(AL=08h 用于检查可移动介质，AL=0Bh 用于改变在块设备的共享重试计数)。在 MS-DOS3.10 版本中增加了两个用来检查网络重定向的参数（AL=09h 检查设备，而 AL=0Ah 检查文件或设备句柄）。

• 功能 5Eh 和 5Fh 仅被 3.1 及更新的版本所支持。它们仅在网络环境中使用。它们各自又可细分成几个子功能：作为 4 位十六进制（16 比特）功能号装入 AX 寄存器中，其中最后两位数字代表特别功能(或子功能)。功能 5E00h 用来检索连接与发出此功能调用的同一网络上的某机器的名

字。功能 5E02h 用于对网络上由若干计算机共享的打印机进行初始化。功能 5F02h 至 5F04h 用于控制网络上的数据重定向：5F03h 重定向一个设备；5F02h 检索重定向的信息;5F04h 撤销重定向。

(6) 目录组（39h—3Bh，47h）

由功能 39h 至 3Bh 以及 47h 组成的这组功能，是由 MS—DOS2.00 以及更新的版本提供的。它们各自行使子目录命令:39h 生成子目录（MKDIR 或 MD）;3Ah 删除子目录（RMDIR 或 RD）;3Bh 更改当前目录（CHDIR 或 CD）。功能 47h 用于捡索当前目录信息（如同使用无任何参数的 CD 命令）。

(7) 内存／进程管理组（48h—4Bh）

MS—DOS2.00 版本上附加的几个功能可用于进程和内存管理。本组中的多数功能都涉及内存分配控制。最后一个功能 4Bh 对调用和装入其它程序或覆盖模块很有用。注意，功能 4Ch(终止进程[EXIT])应总是在由功能 4Bh 调用和装入的程序中使用。

总要保持完全的或理想的兼容度可能会很复杂，甚至无益。但事先决定所希望获得的兼容度则是很有益的。然后选出在应用中使用的 MS—DOS 功能号。

5.4.4 错误代码

MS—DOS 产生的错误、错误类型及其处理方法与 MS—DOS 早期的版本相比，已在很大程度上改变了。较新的版本不仅介绍了新的错误代码，还介绍了新的报道错误的机制。下文阐述了 MS—DOS 各版本处理错误的差别。

5.4.4.1 致命或硬错误代码（通过 INT24h）

在 MS—DOS1.0 版本中，返回错误代码完全是由

INT24h 中断向量来处理的。这些错误代码表示与硬件相关的，以及被认为其性质是很严重或是至关重要的错误。尽管 MS-DOS2.0 版本又介绍了一些新的代码，上述相同的代码及其报道机制却是被所有较新的版本支持的。

如果某应用程序有意要响应此错误报道机制，该程序的初始化代码应保存在 INT24h 向量中，并从某个指向程序出错处的子程序来取代该向量。该程序终止前，原始的 INT24h 向量应恢复到其初始状态。在 MS-DOS 版本下，经过此机制被返回的代码最多可达 7 个;在 2.0 版本下可达 13 个，在 3.0 及更新的版本下可达 16 个。

致命错误代码还可通过 MS-DOS2.0 版本所介绍的另一错误报道机制检索到。在 2.0 版本下，当出现错误条件时，某些功能调用将返回错误代码。

5.4.4.2 功能调用错误返回代码(MS-DOS2.0 及更新版本)

从 2.0 版本开始，所有的 MS-DOS 版本中，一旦该功能执行后出现错误结果，某些功能调用将在特定的寄存器中返回错误代码。如果发生错误就会设置进位标志，并且在相应的寄存器中检查错误代码（如果功能支持）。如果进位标志未置位，就可判断为无错误发生。前述的致命性错误或硬错误（由 INT24h 机制确定）也是通过此机制给出的，不过采用了不同的代码值。在 MS-DOS2.0 至 3.1 的所有版本下，如果在下列功能执行后设置了进位标志，这些功能则将在 AX 寄存器中返回错误代码。这些功能是:38h 至 4Bh，4Eh，4Fh，56h，57h，5Ah 至 5Ch，以及 5E00h 至 5E04h。由于某些功能在 AH 中返回了其它信息，所以应在 AX 的 AL 那一半中检查错误代码。对所有这些功能而言，

AL 中为 0 则表示无错误发生。

注意，错误代码 19 至 31 与 INT24h 类型错误代码 0 至 Ch 对应。而错误代码 34 与 INT24h 类型错误代码 Fh 相对应。

5.4.4.3 功能调用扩展错误信息(MS-DOS3.0 及更新版本)

因为要考虑 MS-DOS 不同版本之间的兼容性问题，所以在较新的版本中几乎不可能为所有新的和现存的功能调用增加错误返回信息处理。为了加强 MS-DSO 的错误处理能力，MS-DOS3.0 版本介绍了一种称为扩展错误代码的新方法。在 3.0 及其所有后继的版本中，当一个功能执行后，或是进位标志置位，或是 AL 寄存器中为 FFh。通过将 0 装入 BX 寄存器，然后发出功能调用 59h（获得扩展错误），就可立刻检索到补充的详细的错误信息。所返回的信息如下表所示。

扩展的错误返回信息

寄存器	内容
AX	错误代码
BH	错误类别
BL	建议的动作
CH	地点

· 错误代码

在 AX 寄存器中返回的错误代码可以是列入上表中的任何一个，关键取决于 MS-DOS 版本。

· 错误类别

表 5.2 中的某个值在 BX 寄存器中返回，用于表示错误的一般类别。出于不同的原因，同一错误代码可能出现两次，所以它有助于确定错误的真正原因。

表 5.2 错误类别

数值	定义
1	缺少资源（不再有空间、通道等）
2	暂存状况（问题可能消除，如文件加锁）
3	合法性（拒绝访问）
4	内部（MS-DOS确认错误由内部的bug引起而不是由用户或系统引起）
5	硬件失灵（问题不是由用户引起）
6	系统失灵（系统软件严重失灵，尽管可能不是用户程序直接造成的错误，如失去或弄错配置文件）
7	应用程序错误（如要求不一致）
8	未找到（文件或其它项目没找到）
9	错误格式（文件或其它项目格式不正确）
10	锁住（文件或其它项目相互锁住）
11	介质（介质失灵，如不正确的磁盘，CRC错误驱动器中不正确的磁盘或被损坏的介质表面）
12	已存在（与现在的项目如文件名、机器名等相冲突）
13	未知（未分类或分类不恰当的错误）

· 建议的动作

列入表 5.3 中的值是在 BL 寄存器中返回的，它指出了摆脱错误状态的动作进程。

表 5.3　纠正错误的建议动作

数值	定义
1	重试（重试数次，如继续失败，提示用户确认程序是否应继续进行或终止）
2	延迟重试（与重试相同，但首先暂停以确认错误是否已被纠正）
3	用户（提示用户重新输入，可能已打入了不正确的文字）
4	（通常在清除后终止程序）
5	立刻退出（不正常地跳过清除而终止程序）
6	忽略（可以忽略的错误）
7	用户介入后重试（用户作用后继续工作，如更换磁盘）

· 地点

列入表 5.4 的值是在 CH 寄存器中返回的，它提供了关于问题的地点方面的补充情况。

表 5.4　错误地点

数值	定义
1	未知（无特点的或无合适的）
2	块装置（与磁盘存储介质有关）
3	网络（仅MS-DOS3.1版本[与网络有关的错误]）
4	串行设备（与串行连接或设备有关的错误）
5	内存（与 ROM 内存有关的错误）

由于在 MS-DOS 较新的版本中，有关错误处理方面的内容有所更改，因此程序员将面临困难的选择。十分显然，新的"扩展错误"信息技术对于程序中设计错误俘获子程序是非常有用的，代价是失去兼容性。如果一定要在子程序中采用此技术，并且还要保持与 MS-DOS 较早版本的某些较低水平的一致性，那么，Get MS-DOS Version 子程序（本节前面已有介绍）可能是很有用的。对于 MS-DOS2.0 以前的各版本，需要加强错误处理的能力，并提供对更多的错误代码的检查;对于 3.0 以及更新的版本，使用 Get Extended Error（获得扩展错误）信息功能调用甚至可进一步加强错误处理的能力。

5.4.5 磁盘格式

由 MS-DOS 不同版本所支持的磁盘格式有好几种。表 5. 5 提供了由 MS-DOS 各版本（最新的到版本 3.1）所支持的全部 5.25 英寸和 8 英寸软盘的说明概要。

尽管有些磁盘格式和类型（如 3.5 英寸磁盘）是由 MS-DOS 实现支持的，但表 5. 5 所列的那些格式却是正式由 MS-DOS 支持的。类似的情况还有没报道磁盘的说明，因为许多变种是产品专用或系统专用的，所以未得到 MS-DOS 的正式支持。唯一可被认作标准的硬盘格式是那些在 IBM XT（10MB）和 IBM AT（20MB）中使用的。许多其它作为 IBM PC 系列机附件而设计的硬盘即使不完全相同，也都很相似。仅通过增加一个设备驱动程序或新的 ROM BIOS，就可在 MS-DOS 机器上运行容量更高的硬盘。但随之而来又会出现硬盘不同尺寸和类型的问题，如 3.5 英寸、5.5 英寸、8 英寸的温盘、可移动的 3.5 或 5.25 英寸刚硬磁盘以及 8 英寸 Bernoulli cartridges。所有这些硬盘

表5.5　MS-DOS软磁盘格式

	5.25英寸					8英寸		
MS-DOS版本	1.00	1.10	2.00	2.10	3.00			
格式描述字体	FFE	FFF	FFC	FFD	FF9	FFE	FFD	FFE
面数	1	2	1	2	2	1	1	2
每面道数	40	40	40	40	80	77	77	77
每道扇区数	8	8	9	9	15	26	26	8
每扇区字节数	512	512	512	512	512	128	128	1024
每束扇区数	1	2	1	2	1	4	4	1
保留扇区	1	1	1	1	1	1	4	1
每FAT扇区数	1	1	2	2	7	6	6	2
FAT数	2	2	2	2	2	2	2	2
根目录扇区数	4	7	4	7	14	17	17	6
根目录登记项	64	112	64	112	224	68	68	192
扇区总数	320	640	360	720	2400	2002	2002	1232
可用扇区总数	313	630	351	708	2371	1972	1968	1221
盘束总数	313	315	351	354	2371	493	492	1221
总容量	160K	320K	180K	360K	1.2M	250K	250K	1.232M
总的可用容量	156.5K	315K	175.5K	354K	1.1855K	246K	2101K	1.221M

在其追加到运行MS-DOS的机器上时，都要求增加一个设备驱动程序或使用经修改的BIOS程序。

5.4.6 文件操作

涉及到MS-DOS不同版本时，要考虑程序中文件处理的方式。首次公布的MS-DOS所提供的文件处理能力与CP／M操作系统所提供的相似。此相似性是有意识做的，因这为程序员从CP／M向MS-DOS的8比特和16比特程序的转换提供了一种较易接受的方法。为保持兼容性，所有的MS-DOS版本（最新至3.1）都具有同样的文件处理能力。然而，在2.0版本中介绍了一种新方法，它是背离CP／M文件处理方式的主要标志。此方法与Xenix操作系

统使用的文件处理方法极为相似。不过，尽管新方法使用起来要容易得多，但它与旧方法不兼容，因此需要特别注意。下文将阐述这两种方法的差别。

5.4.6.1 使用文件控制块（FCB）

MS-DOS 第一版中介绍的功能调用 0Fh 至 29h 与称为 FCB（文件控制块）的结合起来用以生成、修改和删除文件。FCB 是写在内存中的一个代码段，它定义了程序控制的文件参数。MS-DOS 和应用程序采用 FCB 参数来查明该文件的位置、名字、大小以及其它有关信息。然而，由于在真正生成一个完整的 FCB 时没有提供任何功能调用，所以，该 FCB 必须在程序使用任何一个与该文件有关的功能调用前已作了定义。在所有情况下，每个与文件有关的功能调用（0Fh 至 29h）都要求在执行该功能前，将内存中 FCB 的位置装入 DS:DX 寄存器对。这意味着应用程序必须首先生成 FCB，然后将它装入内存数据段或者内存代码段的数据区（由程序初始时定义）中的某个已知位置。

当 MS-DOS 装入一个程序时，系统就在该程序的 PSP (程序段前缀）中生成两个 FCB 并使之格式化。有关 PSP 中的这些 FCB 位置以及其访问 PSP 的方式的内容见《MS-DOS 开发者指南》一书的第三章。在程序进入时，文件名字域由在命令行上打出的信息来填写。然而，如果文件说明中含有路径名字，那么，只有 FCB 中的驱动器号有效。而且，在 FCB 中不能出现重新定向的指令。最后注意，如果程序打开 PSP 中的第一个 FCB，第二个 FCB 则会被关闭。

表 5.6 MS-DOS FCB 格式

偏移点	字节大小	定义	修改者
-7	1	0FF hex	程序
-6	6	保留(必须为 0)	程序
-1	1	文件属性	程序 / MS-DOS
0	1	驱动器号(0 到 16)	程序 / MS-DOS
1	8	文件或设备名字	程序
9	3	文件扩展或类型	程序
12	2	当前块	程序
14	2	以字节计的记录大小	程序
16	4	以字节计的文件大小	MS-DOS
20	2	日期	MS-DOS
22	10	保留	MS-DOS
32	1	当前记录	程序 / MS-DOS
33	4	随机记录号	程序 / MS-DOS

注:偏移点和大小的值都是十进制数.

表 5. 6 所示的是 FCB 的结构以及 FCB 中每个参数在内存里的大小和偏移点位置. 注意, 不是所有 FCB 中的参数都由程序控制的.其中一些仅可由MS-DOS 本身来修改, 另一些 MS-DOS 和程序均可修改. 在任何一种情况下, 当生成 FCB 时, 必须给所有的参数分配空间.

表 5. 6 中凡带有负偏移点的都是在 MS-DOS2.0 或更新的版本中用来使 FCB 变换成称为"扩展"FCB 的."扩展" FCB 允许在偏移点-1 中使用文件属性参数. 0FFh 必须出现在偏移点-7 处, 以表示 FCB 是"扩展"的 FCB.

5.4.6.2 MS-DOS 文件句柄

MS-DOS2.0 版本提供了一种容易得多的文件控制方法.仅需指定能描述整个文件说明的单个ASCII字符串并用 0 终止之, 就可以使用好几个功能调用, 而不必在文件生成

或打开时都得千方百计地去定义和生成 FCB。此称之为 ASCIZ 字符串，可长达 64 字节以便容纳长的路径名字，它遵循与正常文件说明相同的句法：

驱动器:路径＼文件名.扩展名

当执行功能调用 3Ch（生成文件）、或是功能调用 3Dh（打开文件）时，MS-DOS 以包含在该 ASCIZ 字符串中的信息为基础生成所谓的文件句柄。功能调用 3Ch 至 57h 都是涉及到文件句柄使用的、与文件有关的功能调用。它们包括 MS-DOS3.0 版本中介绍的三种新功能（5Ah 至 5Ch）。

由于 MS-DOS 生成并控制文件句柄，应用程序就不必再跟踪掌握文件信息在内存中的位置。仅参考该 ASCIZ 字符串就足以通报 MS-DOS 了。此内部装备还有一个好处：几个文件句柄可同时存在，因为 MS-DOS 始终掌握着它们在内存中所处的位置。

文件句柄唯一的缺点是，MS-DOS2.0 以前的版本不支持它们。因此，如果程序必须与 MS-DOS 的所有版本兼容时，则避免使用文件句柄。注意，由于文件句柄的引入以及许多其他特征，MS-DOS2.0 至 3.1 版本已被证明是区分较细的操作系统（如 CP／M）与较先进的 Xenix 操作系统的分水岭。

几乎所有与新的MS-DOS文件有关的功能调用及其他特征，如路径名字、树结构目录、重新定向等，都是直接与 Xenix 兼容的。因此，向上的兼容性也应考虑，尤其认识到 Xenix 的当前版本不支持旧的文件处理的 FCB 方法时更是如此。在 MS-DOS 将来的更高级的版本中，传统的 FCB 文件处理方法可能会从操作系统中完全取消。

5.4.7 MS-DOS及IBM PC机系列

IBM PC 机在所有安装了 MS-DOS 的计算机中无疑是最流行的. 的确, MS-DOS 之如此受欢迎应归因于 IBM PC 机系列以及兼容机史无前例的成功. 在阅读 IBM PC机的 MS-DOS 手册 (在此手册中 MS-DOS 称为DOS 或 IBM PX DOS) 以及由 Microsoft 出版的 MS-DOS 手册时, 你会注意到这两者的相似之处和明显区别. 涉及到 MS-DOS 部分的相似之处是那些对于所有 MS-DOS 实现而言都是标准的或一般的. 差别是: 特定实现 MS-DOS 时的特殊功能是唯一的. 本书之目的是从普通观点出发来揭示在 MS-DOS 环境中的编程问题, 因此其重点在对所有 MS-DOS 实现都适用的编程上. 然而, 由于 IBM PC 机上的 MS-DOS 是最流行的实现, 所以, 必须弄清楚其相似性和差异性. 这些情况有助于为程序的关键性兼容设计作出决策.

5.4.7.1 相似性

以下 MS-DOS"通用"方面的情况对于 MS-DOS 各版本的实现都一样.

· DOS (磁盘操作系统) 程序

该程序, 即 MS-DOS, 储存在启动磁盘上的一个隐式文件中. 此文件在 IBM PC 机上叫做 IBM DOS.COM. 尽管它在别在机器上可能称谓不同, 但对于某给定的版本它是一样的, 并且可被分解为以下几部分:

(1) 操作系统执行程序

(2) 功能调用

(3) 内存管理 (不是内存构造) 可达 640K

(4) BIOS 接口 (不是 BIOS 本身)

・BIOS 接口程序

BIOS（基本输入／输出系统）接口程序在 MS-DOS 和 BIOS 之间起接口或翻译器的作用。在 IBM PC 机上，此接口储存在启动磁盘上称为 IBM BIO.COM 的隐式文件中。程序的输入部分对任何版本的 MS-DOS 都一样。但其输出部分则取决于特定的机器（IBM PC，IBM PCj r，IBM PC Portable，IBM PC XT 或 IBM PC AT）．IBM PC 兼容机的 DOS 具有一个类似的文件，不过名称可能不同。在某些 MS-DOS 实现中（如 MS-PRO 以及 Comupro（Viasyn）的 PC-PRO 计算机）此文件被 BIOS 全部取代。

・命令解释程序（COMMAND.COM）

该非隐式文件存在于所有启动磁盘中。通常情况下，它对于所有的实现都一样，偶尔也会碰到不同情况。它提供了 MS-DOS 与计算机用户、显示提示之间的"接口"，以及内部命令和函数，如 DIR,COPY,RENAM，ERASE 及重新定向。

・外部命令

一组外部命令对所有的 MS-DOS 实现都是标准的。但仍常增加一些对特定的 MS-DOS 实现来说是唯一的外部命令。比如，COMP 和 DISKCOMP 命令就是专用于 IBM PC 机系列的。许多其他 MS-DOS 实现都有等价命令，不过它们有些微小的差别，并且名称常常不同。

5.4.7.2　差别

以下各部分的 MS-DOS 是实现所专用的。

・BIOS

在 IBM PC 系列机以及几乎所有的 IBM PC 兼容机

上，BIOS 都储存在 ROM 中．BIOS 包含多个子程序，作为 MS-DOS 的扩展来控制硬件．由于该硬件总是以计算机厂商的特殊设计为基础，BIOS 的设计也必须是特殊的，除非它是从别的厂商那里购买来的．以下的 BIOS 通用部分常常是与机器有关的：

(1) 硬、软件中断处理程序

(2) 磁盘控制器和磁盘驱动器的子程序

(3) 控制台、打印机及通信端口的子程序

(4) 其他各种功能，如图形控制器和游戏适配器等

· BIOS 接口程序

在任何具有 BIOS 接口文件（如 IBM PC 机系列上的 IBM BIO.COM）的机器上，为了接收来自 MS-DOS 操作系统的"通用"数据，该程序的输入部分是一样的．然而，文件的输出部分却不一样，因为它必须能够与特殊的 BIOS 联系．

· 设备驱动程序

为了控制该系统硬件的某些独特部分，现在许多系统都拥有作为 MS-DOS 部分的设备驱动程序．在 IBM PC 机系列中，叫作 ANSI.SYS 的设备驱动程序将扩展的功能增加到监示器系统上．某些 IBM PC 兼容机也给出了一个相似文件，但非 IBM PC 兼容机则很少给出，

· 外部命令

在 MS-DOS 实现中常常包含特别的非标准外部命令．

一般说来，MS-DOS 实现之间最重要的差别与 BIOS 本身有关，因为 BIOS 含有机器特殊硬件要求的子程序．(如磁盘控制器、监示器或终端以及键盘)．因此，在作出关于程序的设计决定时，应认真考虑所期望的程序兼容水平．

如果要使程序与所有的 MS-DOS 实现兼容，千万别直接访问 BIOS，也别使用如中断这样的系统专用功能。如果既需要系统专用功能，又要求较高的兼容性，这类功能就应该或者由设备驱动程序来处理，或者在该程序自身中处理。后者要求有一个能作为与机器有关参数修改的安装程序相配合。

甚至在 IBM PC 系列中也存在兼容性问题。比如，储存在 ROM 中的 BIOS 程序的兼容性，在 IBM PC，IBM PC XT 以及 IBM PC AT 里都是有变化的。尽管在 IBM PC XT 中也存在着 IBM PC 中的 BIOS 功能，但 XT 提供补充功能。IBM PC XT 与 IBM PC AT 之间存在着明显的差别。如果对这些差别不清楚，可参见各机器的《IBM 技术参考（硬件）手册》。每种手册都有完整的 BIOS 清单。

5.5 与其它操作系统的兼容性

如本节前面所说，MS-DOS 在许多方面都与其他操作系统相似。MS-DOS 第一版本，无论从程序员还是从用户的观点来看，都与 CP／M 操作系统相似。尽管 MS-DOS 的许多特点在 CP／M 中都不存在，但其基本结构和命令使用（如 DOS A>提示及 COM 命令文件）实际上是相同的。然而，MS-DOS2.00 版本中介绍的几种特点和功能则是从高级得多的 Xenix 操作系统以及 Microsoft 发展来的。（Xenix 是流行的小型计算机与大型机操作系统 UNIX 的一个变种。）其许多功能，如文件、设备重新定向、管道、设备驱动程序以及文件句柄等都是从 Xenix 提供的相似功能派生出来的。由于现在可供使用的 MS-DOS 版本有好几个，某些较新的操作系统便提供了 MS-DOS 的兼容性。最

著名的例子也许是 Digital Research.Inc（CP／M 原始的设计者）的 Concurrent PC DOS 和 Concurrent DOS 286. 下面将阐述 MS-DOS 与这些兼容机及伪兼容操作系统的相似点与不同点.

5.5.1 CP／M-80

检查了 MS-DOS 的构造和能力后，就会发现其设计思想来自以 8080，8085，Z80 等微处理器为基础机器使用的 CP／M 操作系统. IBM PC 在推出 MS-DOS 以前，CP／M 被认为是微机的工业标准操作系统. CP／M 现在仍然是 8 位机最流行的操作系统. 在计算机制造商们开始考虑采用当时从 Intel 引进的 8086 微处理机来设计 16 位机的计划时，由于那时无法获得 16 位的 CP／M 版本（现称为 CP／M-86），许多制造商便不得不等待. 然而，当时 Seattle Computer Products 公司却走在了前面，设计出了它们自己的称之为 QDOS（Quick'n Dirty Operating System）的操作系统. 该系统经过几次改进后又重新命名为 86-DOS.

86-DOS 的构造与 CP／M 的很相似. 但 Seattle Computer Products 公司改进了许多已有的功能，并增加了一些新功能. 然后将它卖给了 Microsoft 公司并被重新命名为 MS-DOS. IBM 在它们的新产品 IBM PC 机上采纳了 MS-DOS 第一版（该版基本上是未更改的 86-DOS）. 然后 Microsoft 又对 MS-DOS 做了一些改进，其成果就是 MS-DOS2.00 版本. 该版本保留了第一版的许多功能，因此，也就保留了与 CP／M 的相似性. 这样对于程序员来说是有好处的，因为多数的 CP／M 程序可以很容易地转换成 MS-DOS. 从程序员的观点来看，以下的相似性是十分重

要的。

・功能调用

MS-DOS第一版的多数功能调用，尤其是那些与文件功能有关的，都与CP／M2.2和3.0版本所提供的功能调用相似。尽管8位8080／Z80和16位8086系列的微处理机之间在寄存器的使用上很不相同，但其建立功能和返回信息的方式却非常相似，甚至某些功能调用号都一样。实际上与CP／M相同的MS-DOS功能调用包括功能0至24h。这些功能及其操作在MS-DOS后来的版本(最新到3.1版本)中仍保留着。

・FCB文件控制块

MS-DOS第一版生成、打开、更改或删除文件的唯一方式就是采用FCB。MS-DOS下的FCB格式及其建立方式与CP／M下的FCB的用法几乎一样。因为在许多仅以DOS为基础的操作系统中，文件处理是很关键的，所以，在CP／M和MS-DOS中的FCB用法之相似性，对于程序员来说就非常重要了。尽管在MS-DOS2.00版本中介绍了一种新的文件处理方法，但为保持兼容性，所有以后的版本（直至3.1版本）都保留了FCB的"旧"方法。

・命令

在这两种操作系统中，内部命令和外部命令的使用都非常相似。CP／M有其自己的内部命令，称之为CCP（控制台命令处理程序)，作为操作系统的一部分被装入内存。MS-DOS采用十分相似的方法来处理内部命令，但不同的是其命令处理程序存于COMMAND.COM磁盘文件上。对于外部命令，MS-DOS也有8位兼容方式。其处理COM文件的方式与CP／M采用的处理方式几乎相同。在

MS-DOS 中，COM 文件仅占用一个 64KB 的内存段，由此来仿真 8080 或 Z80 微处理机基础系统的内存使用方式。但是，在 MS-DOS 中，EXE 命令格式仅用于有 8086 系列微处理机的机器上，因此，它与 CP／M 不兼容.

5.5.2 CP／M-86 及 Concurrent CP／M-86

CP／M 操作系统（16 位）相当于微处理机 8086 系列最早的 CP／M。它从 8 位 CP／M 版本中继承过来的许多特征都与 MS-DOS 相似。比如，在 CP／M-86 中所采用的 FCB 及文件相关功能调用（除文件句柄外）与 MS-DOS 所采用的非常想象.

CP／M-86 问世不久，很快又出现了一和叫做 Concurrent CP／M-86 的新版本，它比 CP／M-86 增加了多任务和开窗口的特点。这两种操作系统的特殊版本都是为 IBM PC 发布的。CP／M-86 的多数功能都被 Concurrent CP／M-86 所继承，但其中许多功能又被后者的多任务特点进一步地复杂化了.

5.5.3 Concurrent CP-DOS 和 Concurrent DOS-286

随着 MS-DOS 作为 16 位 8086 系列为基础的微处理机的工业标准操作系统的出现，CP／M 的制造商意识到由于 MS-DOS 如此庞大的用户基础，他们必须提供某种与 MS-DOS 相兼容的形式。Digital Research Inc 公司公布了一种 Concurrent CP／M-86 的改进型版本，叫做 Concurrent PC DOS。在其首次公布时就提供了与 MS-DOS1.0 版本的兼容性。改进了的 Concurrent PC DOS 版本 3 再次保持了与 MS-DOS2.00 版本的兼容性。该操作系统能够并发地运行 CP／M-86 和 MS-DOS 程序，并且接受 MD-DOS 等价版本所支持的所有功能调用.

Concurrent PC DOS 的另一种变种叫 Concurrent DOS 286. 它是为具有 Intel80286 微处理机的机器配备的. 该操作系统是为使用"Virtual"(又称为 Protected)形式的 80286 处理机设计的. 它提供了可寻址范围为 16 兆字节内存量的能力. 它还具有在"real"形式和"virtual"形式下并发运行的能力. 这样 MS-DOS 和 CP/M-86 的程序都可运行了. Concurrent DOS 286提供了与 Concurrent PC DOS 相同的 MS-DOS 兼容特点. 但 Concurrent DOS 286在某些版本的80286处理机上运行有一定困难.当涉及到该操作系统的兼容性时应特别注意, 因为其正确操作在很大程度上有赖于该系统所使用的 80286 处理机的版本. (该处理机的较早版本在"virtual"和"real"形式之间的转换及通信方面存在问题.)

5.5.4 Xenix 和 UNIX

如前所述, MS-DOS 的较新版本(从 2.0 开始)吸收了 Xenix(此为另一个 Microsoft 的操作系统)的某些特征. MS-DOS2.0 版本中介绍的设备驱动程序、重新定向、管道及文件句柄等许多特征, 都是以 Xcnix 的特征为基础功能的, 而 Xenix 又是以 AT 和 T 的 UNIX 操作系统为基础的. 因此, 尽管向下的兼容性问题(MS-DOS 与 CP/M)应予以注意,向上的兼容性问题(MS-DOS 与 UNIX)也同样应该加以考虑, 因为 MS-DOS 中 Xenix 式的特征代表了 MS-DOS 未来版本的发展方向.

5.6 "规距"的 MS-DOS 应用程序

Microsoft 公司与 IBM 公司共同建立过一些生成所谓"规矩"(Well Behaved)的 MS-DOS 和 PC-DOS 应用程序的准则. 遵循这些准则开发出来的应用程序可能在将来的

多任务或网络版本的 MS-DOS 或 PC-DOS 环境下仍能继续正常运行，即使在当前版本的操作系统环境下，亦有助于减少与其他程序之间的互相干扰.

5.6.1 基本准则

规距应用程序的基本准则如下:

· 使用新的句柄（类 UNIX）的文件系统调用（功能 2FH 至 5CH），尽量不使用功能较差、老版本的 FCB（类 CP／M）的文件功能调用.

· 如果不得不使用 FCB 方式，则完成后必须将其关闭，打开时不要移来调去. 不要再次打开已经打开的 FCB，也不要再次关闭已经关闭的 FCB. 这些看起来似乎无害的操作，在网络环境下可能会出问题.

· 使用环境块检查程序的覆盖模块或数据文件的路径.

· 装入覆盖模块或其他程序时使用 EXEC 功能调用 (4BH)，这样可使调用者与程序结构及重装配需求无关.

· 一定要释放程序不用的内存. 对于 COM 类型的程序，此点尤为重要.

· 不要触及不属于你的程序内存空间. 欲建立或查证中断向量，使用 Int21H 功能 25H 及 35H.

· 如果改变了某些中断向量的内容，则应将原来的内容保存起来，并在该程序退出之前重新恢复.

· 避免使用与硬件有关的定时回路，当程序中需要延时的时候，使用 Int21H 的功能 2CH（获得系统时间功能）.

· 只要有可能就使用带缓冲的输入／输出， MS-DOS 版本 2.0 及以上的设备驱动程序可处理长达 64 字节的字符串，使用新的、大些的记录比使用许多短小记录对系统性能更好.

· 尽可能使用 MS-DOS 版本 3.X 所提供的扩展错误报告功能（功能调用 59H）。

· 通过 Int21H 功能 4CH（带返回码程序结束）或功能 31H（带返回码程序结束并驻留）来退出程序。通常的惯例是 0 返回码表示正常退出，非 0 返回码表示退出时具有某种错误或意料之外的情况。然后便可用批文件或父进程检查这些返回代码。应避免使用过时的、通过 Int20H 或 Int21H 功能 0 的退出方法。

5.6.2 与硬件有关的 IBM PC 应用程序

许多程序员在为 IBM PC 系列编写高质量、高性能应用程序时常感到不大可能使程序与硬件完全无关。IBM 公司已注意到这种需求，并决定在 IBM PC 系列机中保持某些硬件接口的稳定性。

(1) 硬件

· 通过端口 61H 控制声音。

· 8253-5 定时器芯片通道 0 到 2 为端口 40H，42H，43H。端口 41H 用于控制动态 RAM 刷新。对于 8253-5 的输入频率保持为 1.9MHz，它与系统其他部分的时钟速率无关。

· 游戏棒适配器位于端口 201H。

· 端口 3BAH 和 3DAH 为垂直和水平扫描间隔状态比特。

· 通过端口 21H 处的 8259A 参考寄存器控制中断系统。

· 端口 03F8H 至 03FFH 处的 INS8250 异步通信控制器。

(2)内存分配

・中断向量——同一中断号的ROM BIOS服务程序总是可提供的，并且总是向上兼容的。如果替换一个中断，应与该中断以前的所有者相链接。

・用于基本显示方式（0到7）的显示器刷新缓冲区处于0B0000H及0B8000H处。

・ROM BIOS及数据区在00400H处。

注意，采用了与硬件有关信息的程序在非IBM PC兼容的机器上，即使是MS-DOS操作系统环境下也无法正常工作，并且在诸如Microsoft Windows，GEM及Topview等多任务环境下可能会产生错误，甚至引起系统"崩溃"。在多任务环境下，最容易出问题的是音调发生和对于8253-5定时器或8259A中断控制器的直接编程。

(3) 机器标识

地址0FFFFEH（F000:FFFE）处的一个字节确定了IBM PC系列中的机器状态：

数值	方式
OFFH	PC
OFEH	PC/XT
OFDH	PCjr
OFCH	PC/AT
OF9H	PC Convertible

对于IBM PC兼容机，此位置的内容变化很大，例如，使用修改版C的ROM的Compaq Portable机器，此位置内容为0，而对于Compaq 286 Portable，此字节为0FCH，这是由于该机器是完全模仿IBM PC/AT的。

一般来说，当试图确定一部非IBM机器的标识时，还

应采取其他方法．例如，如果要确定该机器是否为 Compaq 机，还可对 ROM 空间进行扫描，以查找 Compqg Corporation 的版权记号等（见如下汇编程序）．这种查找工作有时很有用，例如由于 Compaq 的显示适配器没有"雪花"效应，故在字符字母显示方式下可不考虑水平方向的扫描间隔．

```
Compaq    proc  near          ;测试宿主机是否为 Compaq 机器
                              ;如果是 Compaq,AX =-1
                              ;如果不是 Compaq,AX = 0
          mov ax,0f000h       ;在 ROM BIOS 中搜寻
          mov es,ax           ;Compaq 的版权标记
          mov di,0a000h
          mov cx,05fffh
Compoq1:  mov al,'C           ;查找前导 C
          repnz scasb
          jnz compaqr         ;查尽 ROM,未找到字符串"COMPAQ"
          push di             ;存当前 ROM 指针
          push cx
          push si
          mov cx,6            ;找到 C,试着找到字符串"COMPAQ"
          mov si, offset compaq_ name;
          dec di
          repz copsb
          pop si              ;恢复 ROM 指针
          pop cx
          pop di
          jnz compaq 1        ;跳转,字符串不匹配
          mov  ax,-1          ;返回 Compaq = True
          ret
Compaq2:  mov ax,0            ;返回 Compaq = False
                    ret
           Compaq     endp
           Compaq_name db 'COMPAQ'
```

第六章 快速字符显示的程序实现

IBM PC 机的字符显示方式比图形显示方式的处理速度更快,而且数据存贮的容量要求却更小. 最简单的字符显示方式称为"电传式输出"(teletype output),即将字符"流"使用默认属性或颜色写在当前光标位置处. 到达行末后,下一个字符写在下一行的第一列上,到达最后一行后,整个屏幕上滚,采用此种方法常见的例子有:

· C 语言中 printf,fprintf 等写向标准输出或 CON 设备(如控制台)的操作;

· MS-DOS 中写向标准输出或 CON 设备的功能调用 09h 与 40h.

· BIOS 中断 10h,功能写 14.

IBM PC 兼容机对于显示输出有更高级的控制方法:

· 显示 254 种不同字符. 除标准 ASCII 字符外,还有各种外国文字、数学、图形等符号.

· 直接将给定字符安放在 25 行乘 80 列屏幕矩阵的任意位置上.

· 以各种属性在单显系统上、或以特定的后台与前台彩色在彩显系统上显示字符.

· 为生成及管理窗口,可对屏幕上选择的部分保存、覆盖、恢复、清除或滚动.

这些增加的显示能力有三种基本方法可实现,即 ANSI SYS,BIOS 及直接显示写.

6.1 通过 DOS / ANSI.SYS 的视频显示

产生可控制视频输出最高层次上的方法是使用 MS−DOS 功能调用配合以可安装设备驱动程序 ANSI.SYS，这些均是随操作系统提供的. MS−DOS 功能通常产生的是简单电传机式的视频输出,但如果安装了 ANSI.SYS,则对视频输出(以及键盘输入)可提供更高级的控制.

欲使用 ANSI.SYS 的视频显示功能,需执行三个步骤:

第一步,在启动磁盘根目录中的 CONFIG.SYS 文件中应有一行:

DEVICE = ANSI.SYS

如有需要,在 ANSI.SYS 应冠以完整的路径名.

第二步是使用下列的 MS−DOS 功能调用于视频输出:

- 功能 02h,06h,09,自动向标准输出设备写;
- FCB 功能 15h, 22h, 28h, 向名为 CON 的已打开的文件写;
- 句柄功能 40h,向标准输出写(句柄 01),向标准出错设备写(句柄 02),向名为 CON 的已打开文件写;

C 库函数产生的控制台输出均采用上述各组中的 MS−DOS 功能,故与 ANSI.SYS 可配合使用.

第三步是在输出流中嵌入适当的控制序列. ANSI.SYS 控制代码是一些 escape(转义)序列，均以下面两个字符打头 ESC [，在十六进制表示下，为两个字节 1Bh 5Bh.

详细使用方法见清华大学出版社 1987 年 9 月出版的夏东涛等编译的"IBM PC DOS 3.X 版本技术参考手册"的第三章.

ANSI.SYS 可实现下面类型的视频控制:

▲光标定位与移动

光标可被置于屏幕上任何绝对位置处,或从当前位置在任意方向上移动一指定距离. 当前光标位置可被保存及恢复.

▲抹除

可抹除整个屏幕或某一行.

▲显示属性

可指定写到屏幕上所有字符所用的彩色或单色显示属性. 在单色系统上,可选择下列属性:

- 高亮度
- 加下划线
- 闪烁
- 反相显示
- 不可见字符
- 正常显示(除去所有特殊属性)

在彩色系统上,可选用下列特点:

- 16 种前景色及 8 种背景色
- 闪烁
- 正常显示(黑底白字,无闪烁)

用于上述属性的数字代号与 BIOS 所用的及视频控制硬件所用的不同.

▲显示方式

可设置显示方式.

下面的代码说明了如何从一个 C 程序中使用 ANSI.SYS 驱动程序. 因为 printf 使用 MS-DOS 功能调用产生输出,故只需简单地将正确的 escape 序列直接嵌入欲显示的字符串中,例如:

```
printf("\x1b[41;30m");          /* 黑色前景,红色背景 */
```

```
printf("\x1b[10;5H");              /* 光标定位于 10 行,5 列 */
printf("ROW 10,Column5,black on red");
printf("lx1b[0m");                 /* 恢复正常显示. 黑底白字 */
```

使用 ANSI.SYS 有几个优点. 主要优点是移植性很好. 因为所有 MS-DOS 机器都可提供 ANSI.SYS,故它提供了一个方便的与设备无关的显示接口. 另外由于 ANSI.SYS 方法使用的是 MS-DOS 功能,故显示输出只要被发送到标准输出,便可被重定向. 最后,使用 ANSI.SYS 可与多任务环境兼容. 例如,通过 ANSI.SYS escape 代码控制显示输出的应用程序可在 Microsoft Windows 的一个窗口中正常地运行(但 Windows 可能在解释某些特定颜色的指令时与 ANSI,SYS 有所不同).

ANSI.SYS 方法亦有两个重要缺点. 首先它的使用要求修改用户的配置文件,且在应用程序可正常运行之前需重新启动系统.其次ANSI,SYS 相当慢,尤其在默认的"加工后(looked)"方式下更是如此.

因此 ANSI.SYS 对于高水平程序设计不是最佳选择. 但如果使用它,转换为"未加工"方式可使输出速度明显提高. 记住,在程序结束前应恢复原来的设备方式.

6.2 通过 BIOS 和属性代码的视频显示

BIOS 中断 10h 充分发挥了 IBM-PC 兼容机的显示能力,比 ANSI.SYS 所用的 MS-DOS 功能更有效. 下面的程序例子 printa 说明了如何在 C 程序中使用 BIOS 视频显示功能,并表明了可获得的控制程序.

此方法用于显示少量信息及管理窗口很方便,但当要显示整屏数据时,它比6.3 中介绍的直接视频写方法效果差

很多。

BIOS 视频功能及直接视频写方法使用同样的一字节代码指定视频显示属性(此为传送给 printa 函数第二个参数的值)。PC/AT 技术参考手册详尽列出了彩显及单显系统的256 个可能的代码值及其意义。这些代码的每一比特都有一个特定的意义。因此便有可能不用在技术手册中查找每一个值便使用几种不变的定义产生所需要的效果。某个特定代码的解释取决于系统使用的是单色还是彩色显示适配器。

▲彩色系统

彩色系统属性代码比特的意义如下:

7	6	5	4	3	2	1	0
b	R	G	B	i	R	G	B

背景	前景
彩色	彩色

b=闪烁

R=红

G=绿

B=蓝

i=高光度

3 个背景比特允许 8 种可能的背景色,4 个前景比特允许 16 种可能的前景色。闪烁比特使各种颜色的组合开始闪烁。如果懂得红、绿、蓝三种主要颜色组合的方式,下面的常数定义可用于简化指定颜色代码的过程(记住与高光度比特作"或(OR)"操作可使所选颜色鲜亮):

```
#define  BLINK   0×80   /* 闪烁                */
```

```
#define BG-R     0×40  /*背景红、绿、蓝      */
#define BG-G     0×20
#define BG-B     0×10
#define FG-I     0×80  /*前景高亮度比特     */
#define FG-G     0×40  /*前景红、绿、蓝     */
#define FG-G     0×20
#define FG-B     0×01
```

所要求的前景和背景颜色可由与适当的常数作"或"(OR)操作而产生,例:

```
promta("Red  on  black",FG _ R,10,0);
printa("Cyan  on  black",FG-G|FG _ B,11,0);
printa("Yellow  on  magenta",FG-I|FG-R|FG-G|
BG-R|BG-B,12,0);
```

▲单色系统

下面给出了单色系统属性代码各比特的意义.

7	6	5	4	3	2	1	0
b	B	B	B	i	F	F	U

背景	前景

b=闪烁

B=背景比特

i=高亮度

F=前景比特

U=下划线

所有可能的属性组合可由定义五个常数来指定:

```
#define BLINK        0×80     /*闪烁          */
```

```
#define BG          0×70     /*打开背景       */
#define INTENSE     0×08     /*高亮度字符     */
#define FG-NORMAL 0×7       /*转向常规前景   */
#define FG__UL      0×01     /*转向带下划线前景 */
```

但要遵守下面的规则:

· 属性的组合由与上面的常数作"或"操作得到. 例如,欲产生带闪烁的常规前景,使用:

FG_NORMAL|BLINK

· 有一种类型的背景BG, 两种类型的前景FG_NORMAL 及 FG_UL. 只可选择一种类型的前景,取决于是否需要下划线.

· 如果既没有选择背景 BG, 又没有选择前景(FG_NORMAL 及 FG_UL), 则字符将为黑低黑字（不可见).

· 如果仅选择前景,则字符将是黑低白字.

· 如果仅选择背景,则字符将是白底黑字(即反相显示). 与反相显示结合选择高亮度的比特对某些适配器可产生可视差别.

· 如果同时选择了背景和前景,则前景选择占先,字符为黑底白字. 此时不可能产生反相显示和带下划线字符,因为这必须同时选择背景和前景.

如果遵守上述规则,对任何单显系统均可预测到结果. 有些系统还允许其他的属性组合(如白色背景上高亮度白色前景,即 0×7f;或高亮度反相显示,即 0×78). 但在不同的系统之间这些特殊的组合将不产生可预见的结果,如果寄存器设置被改变,即使在同一个系统上结果也会不同.

单色常数的用法与彩色常数的一样．例如:

printa("row10,column5,reverse video,blinking",
BLINK|BG,10,5);

6.3 通过直接对视频内存区写的视频显示

最有效的屏幕输出方法是直接对视频内存区写．视频内存区是 RAM 中的一个保留区域,用于映照显示在屏幕上的字符和属性(在图形方式下用于映照象素)。程序对此区域的读、写与对 RAM 中其他区域的读、写一样．视频控制器硬件可直接访问该内存区,并不断对其扫描,故写入此区的任何东西立即会在屏幕上显示出来．

在 80 列文本方式下,视频内存区由 4000 个字节组成．第一个字节内容是屏幕左上角(0 行,0 列)显示字符的 ASCII 值,第二个字节为该字符的显示属性,第三个字节为 0 行 1 列字符的 ASCII 值,第四个字节为其显示属性．此模式完成全部 25 行 80 列屏幕的 2000 个字符．次序是以行为主:某行上所有字符在下一行字符前面．图 6 — 1 中字符mn 意为 m 行，n 列上的字符。显示属性值(无论是单色,还是彩色)均与 BIOS 显示方式所述一致．

字符00	属性00	字符01	属性01	字符03	属性03…
字符10	属性10	字符11	属性11	字符13	属性13…
字符20	属性20	字符21	属性21	字符23	属性23…

图 6-1 视频显示内存区结构

单色系统与彩色的视频内存区是有差别的．使用单色适配器(标准 MDA 单显适配器,字符方式下的 Hercules 卡,接在单色显示器系统上并工作于方式 7 的 EGA 卡等等)时,视频内存区位于B000h 段,为4000字第一页,即第一个ASCII 值

位于地址 B000h:0000h 处．但如使用彩色适配器(标准 CGA 或文本方式下的 EGA),80 列文本方式(mode2 或 3),有 4 个 4000 字节的视频 RAM,在下列段地址处开始:

0 页　B800h

1 页　B900h

2 页　BA00h

3 页　BB00h

数据可直接写入任意页(使用 BIOS 功能亦可对任意页作数据读、写,或获取、设置与任意页有关的光标位置)．但在一给定时刻仅显示一页(此页叫"活动显示页"),默认情况下，此为第 0 页,但可通过 BIOS 视频功能 5 来改变活动显示页．借助此机构可由程序先行在一个非活动页中积累数据，然后再使其转为活动页而一下子显示出整页．

注意,视频内存区实际上位于显示适配卡中,而不是 640K 用户内存区(地址 0 到 9FFFFh)的一部分(只有 IBM PCjr 例外,其视频内存区是用户内存区的一部分,但由于采用了对读／写请求的硬件重定向,故可通过标准的 CGA 地址来访问)．

直接访问视频内存区有许多优点．首先,此方法特别快,从后面的标准测试(Benchmark)可看出．由于每个字符均有其相关的属性值,故只有此方法可实现对视频显示的最彻底控制．最后,无论对于高级语言还是低级语言,此方法均最易实现．在视频内存区中写入一个字符及其属性无非是向一个变量赋值．可制作一个以视频内存区为样板的数据结构,然后可由标准赋值及数据移动指令来产生复杂的屏幕操作,如屏幕的保存与恢复、窗口的压栈、出栈与移动等．

使用直接访问视频内存区的方法亦有几个不利之处．首

先，非 IBM 兼容的体系结构可能会使用不同的视频显示方案，故此法会失灵．第二个问题更为严重,即直接对视频内存区写限制了程序在多任务环境(如 Microsoft Windows 及 BI, Topview 等)下的使用．这类程序欲在 Microsoft Window 下显示，就必须“挂起”多任务，然后接收屏幕的全部控制．第三个问题是直接视频内存区访问方法用于彩色图形适配器(CGA)时可能产生“雪花”效应．当程序和控制器同时去访问视频内存时便会产生雪花效应．但只有标准的 CGA 控制器才有此问题，使用 EGA，ATT 机器，Compaq 等均无问题．后面的例子说明了如何在视频内存区水平、垂直回扫周期内，川暂时停止显示的同步方法来避免出现雪花效应(但此法大大影响了性能)．

下面介绍了两类直接视频显示函数:一类用于单个字符串写,一类用于生成窗口．

6.3.1 字符串函数

下面清单给出了两个直接对视频内存区写字符串的函数：一个函数用于 CGA 显示,另一个函数可用于其他各种显示．这两个函数功能上类似于 Printa(用 BIOS 中断功能实现),传送的参数也相同,即:

- 欲显示的、以空字符结尾字符的地址
- 显示属性
- 起始列

此函数叫 Printv,其功能类似于 Printa,但速度要快近乎 92 倍!

▲Printv

函数 Printv 仅用于不产生雪花效应的控制器．所执行的主要步骤如下:首先设置 SI 以指向源字符串的开始(DS 已

包含合适的段地址);然后用检查地址 0000h:0440h 处的 BIOS 视频方式标志;以向 ES 赋给视频内存区的正确段地址．然后设置 DI 指向视频段内该字符串欲被显示的偏移量处．此偏移量值根据作为参数传递给的起始行、列计算得到(其中使用了一个计算用子程序以避免使用速度慢的 mu (指令)．最后使用字符串原语 Lodsb 及 stosw 将源字符串与有关的属性一起写入视频内存区。

```
page 50,130

;File:     PRINTV.ASM

;Assembler routines for displaying a string by direct writes to video memory
;These routines may be called from a C program.

public     _printv
public     a01,a02,a03,a04
public     _printvcga
public     c01,c02

_data      segment word public 'DATA'
_data      ends

dgroup     group   _data
assume     cs:_text, ds:dgroup

_text      segment byte public 'CODE'

_printv    proc    near                 ;** Displays a string on screen. **

comment    /*
           This procedure writes a null terminated string directly to video
           memory, and should be used only with IBM-compatible systems with
           controllers that do not produce "snow."

           void printv (str,att,row,col);
           char *str;
           int att;
           int row;
           int col;

           */

sframe     struc                        ;Stack frame template.
bptr       dw      ?
ret_ad     dw      ?
str_ad     dw      ?                    ;String address.
att        dw      ?                    ;Display attribute.
row        dw      ?                    ;Starting row.
col        dw      ?                    ;Starting column.
sframe     ends

frame      equ     [bp]

           push    bp
           mov     bp, sp
           push    di
           push    si
```

```
          mov     si, frame.str_ad      ;DS:SI point to start of source string.

          xor     ax, ax                ;Set ES to correct video buffer segment.
          mov     es, ax
          cmp     byte ptr es:[449h], 7     ;Test video mode.
          je      a01
          mov     ax, 0b800h            ;Color buffer.
          jmp     a02
a01:      mov     ax, 0b000h            ;Monochrome buffer.
a02:      mov     es, ax

                                        ;Starting offset in video memory =
                                        ;       (row * 160) + (col * 2)
          mov     di, frame.row         ;Place row in DI and multiply by 160.
          mov     ax, di
          mov     cl, 7
          shl     di, cl
          mov     cl, 5
          shl     ax, cl
          add     di, ax
          mov     ax, frame.col         ;Multiply col by 2.
          shl     ax, 1
          add     di, ax                ;Add (col * 2) to DI.

                                        ;Attribute in AH.
          mov     ah, byte ptr frame.att
          cld
a03:      cmp     byte ptr [si], 0      ;Test for null termination.
          je      a04
          lodsb                         ;Load a single character from source.
          stosw                         ;Write a char AND attribute to dest.
          jmp     a03

a04:      pop     si                    ;Restore registers.
          pop     di
          mov     sp, bp
          pop     bp
          ret                           ;Return to C program.

_printv   endp


h_retrace macro                         ;Pauses until the beginning of a
          local   b01, b02              ;horizontal retrace.
b01:      in      al, dx                ;Loop to complete any retrace period
          test    al, 1                 ;in progress.
          jnz     b01
b02:      in      al, dx                ;Loop until start of a new horizontal
          test    al, 1                 ;retrace.
          jz      b02
          endm


_printvcga proc   near                  ;** Displays a string on screen. **

comment   /*
          This procedure is a special version of '_printv' designed especially
          for color graphics adapters that produce "snow" when writing to
          video memory.  Same parameters as '_printv'.

          */

          push    bp
          mov     bp, sp
          push    di
          push    si

          mov     si, frame.str_ad      ;DS:SI point to start of source string.

          mov     ax, 0b800h            ;Set ES to CGA buffer.
          mov     es, ax
                                        ;Starting offset =
                                        ;       (row * 160) + (col * 2)
```

```
        mov     di, frame.row          ;Place row in DI and multiply by 160.
        mov     ax, di
        mov     cl, 7
        shl     di, cl
        mov     cl, 5
        shl     ax, cl
        add     di, ax

        mov     ax, frame.col          ;Multiply col by 2.
        shl     ax, 1
        add     di, ax                 ;Add (col * 2) to DI.

                                       ;Attribute in BH.
        mov     bh, byte ptr frame.att
        mov     dx, 03dah              ;CRT status register.
c01:    cmp     byte ptr [si], 0       ;Test for null termination.
        je      c02
        mov     bl, [si]               ;Move next character to BL.
        inc     si
        cli                            ;Disable interrupts.
        h_retrace                      ;Wait for horizontal retrace.
        mov     es:[di], bl            ;Move ASCII.
        inc     di
        h_retrace                      ;Wait again.
        mov     es:[di], bh            ;Move ATTRIBUTE.
        sti                            ;Restore interrupts.
        inc     di
        jmp     c01                    ;Go back for another character.

c02:    pop     si                     ;Restore registers.
        pop     di
        mov     sp, bp
        pop     bp
        ret                            ;Return to C program.

_printvcga endp

_test   ends                           ;End of code segment.
        end
```

上面是直接向视频显示内存区写字符串的汇编程序。

▲Printvcga

函数 Printvcha 是 Printv 的一个特殊形式,仅在产生雪花效应(即 CGA)的情况下用于当前显示控制器。此函数效能远比 printv 低,因此除非用不可时,不要用它(可先用测量技术探知显示类型),并应允许用户取代此子程序的使用。此函数与 printv 函数有两点不同。首先,由于此函数的仅用于 CGA 卡,故视频 RMA 的段地址假定为 B800h,并且不再作显示方式测试。其次,写每个字节之前,此函数等待水平回扫周期的起始。

水平回扫为电子束关闭的时刻(此时电子束返回屏幕的另一边以开始新的水平扫描)。此期间,视频状态端口 3ADh 的比特 0 置 1。宏 h _ retrace 产生在水平回扫开始前的延

迟。注意,必须有两个测试 Loop:

· 第一个 Loop 等到水平回扫的结束(此时当前正在进行水平回扫)。如果无此 Loop,程序便可能在一个回扫的中间进入,其间的时间可能不足从将一个字节传送到视频内存区。

· 第二个 Loop 等到新的水平回扫开始。

在水平回扫期间,反有足够的时间将一个字节传送到视频内存区。故 ASCII 值及其属性必须分别传送,用于延迟的宏 h__rertrace 两次被调用。

6.3.2 窗口函数

字符串显示函数只适于将少量数据写到屏幕上的指定位置,且每个字符串均以相同的一个属性显示。欲显示整个屏幕或大块数据,函数 printw(窗口显示)及 printwcga(用于 CGA 的特殊形式)提供了更有效、方便的方法。

```
page 50,130

;File:      PRINTW.ASM

;Assembler routines for displaying windows
;These routines may be called from a C program.

public      _printw
public      a01,a02,a03
public      _printwcga
public      b01,b02,b03,b04,b05,b06

_data       segment word public 'DATA'
_data       ends

dgroup      group   _data
assume      cs:_text, ds:dgroup

_text       segment byte public 'CODE'

_printw     proc    near                  ;** Displays a window. **
Comment     /*
            This procedure transfers data from a program buffer directly to
            video memory, and should be used only with IBM-compatible systems
            with controllers that do not produce "snow."

            void printw (buf,ulr_s,ulc_s,lrr_s,ulr_d,ulc_d);
            char *buf;
            int ulr_s;
            int ulc_s;
            int lrr_s;
            int lrc_s;
            int ulr_d;
            int ulc_d;
            */
```

```
sframe      struc                           ;Stack frame template.
bptr        dw      ?
ret_ad      dw      ?
buf_ad      dw      ?                       ;Address of source buffer.
ulr_s       dw      ?                       ;Upper-left row source.
ulc_s       dw      ?                       ;Upper-left column source.
lrr_s       dw      ?                       ;Lower-right row source.
lrc_s       dw      ?                       ;Lower-right column source.
ulr_d       dw      ?                       ;Upper-left row destination.
ulc_d       dw      ?                       ;Upper-left column destination.
sframe      ends

frame       equ     [bp]                    ;Base for accessing stack frame.

            push    bp                      ;Set up base pointer to access frame.
            mov     bp, sp
            push    di
            push    si

            mov     si, frame.buf_ad        ;Assign SI start of source buffer.

                                            ;Calculate source buffer offset in BX
                                            ;     = (row * 160) + (col * 2)
            mov     ax, frame.ulr_s         ;Multiply starting row by 160.
            mov     bx, ax
            mov     cl, 7
            shl     ax, cl
            mov     cl, 5
            shl     bx, cl
            add     bx, ax

            mov     ax, frame.ulc_s         ;Multiply starting column by 2.
            shl     ax, 1
            add     bx, ax                  ;Total offset in BX.

            add     si, bx                  ;Add offset to SI.
                                            ;SI now contains offset in source buffer.
            xor     ax, ax                  ;Set ES to correct video buffer segment.
            mov     es, ax
            cmp     byte ptr es:[449h], 7   ;Test video mode.
            je      a01
            mov     ax, 0b800h              ;Color buffer.
            jmp     a02
a01:        mov     ax, 0b000h              ;Monochrome buffer.
a02:        mov     es, ax

                                            ;Calculate video memory offset
                                            ;     = (row * 160) + (col * 2)
            mov     di, frame.ulr_d         ;Place row in DI and multiply by 160.
            mov     ax, di
            mov     cl, 7
            shl     di, cl
            mov     cl, 5
            shl     ax, cl
            add     di, ax

            mov     ax, frame.ulc_d         ;Multiply col by 2.
            shl     ax, 1
            add     di, ax                  ;Add (col * 2) to DI.
                                            ;DI now contains video memory offset.

                                            ;Calculate number of rows in BL.
            mov     bl, byte ptr frame.lrr_s
            sub     bl, byte ptr frame.ulr_s
            inc     bl

            mov     dx, frame.lrc_s         ;Calculate number of bytes/row in DX.
            sub     dx, frame.ulc_s
            inc     dx
            shl     dx, 1
            mov     ax, 160                 ;Calculate SI, DI increment in AX.
            sub     ax, dx

            cld
a03:        mov     cx, dx                  ;Load(reload) bytes/row into CX.
            rep     movsb                   ;Move one row.
            add     si, ax                  ;Adjust SI, DI to start of next row.
            add     di, ax
            dec     bl                      ;Decrement row counter.
```

```
            jnz     a03                     ;If more rows, move another one.

            pop     si                      ;Restore registers.
            pop     di
            mov     sp, bp
            pop     bp
            ret                             ;Return to C program.

_printw     endp

_printwcga  proc    near                    ;** Displays a window. **
comment     /*
            This procedure is a special version of '_printw' designed especially
            for color graphics adapters that produce "snow" when writing
            to video memory.  Same parameters as '_printw'.
            */

            push    bp                      ;Set up base pointer to access frame.
            mov     bp, sp
            push    di
            push    si

            mov     si, frame.buf_ad        ;Assign SI start of source buffer.

                                            ;Calculate source buffer offset in BX
                                            ;     = (row * 160) + (col * 2)
            mov     ax, frame.ulr_s         ;Multiply starting row by 160.
            mov     bx, ax
            mov     cl, 7
            shl     ax, cl
            mov     cl, 5
            shl     bx, cl
            add     bx, ax
            mov     ax, frame.ulc_s         ;Multiply starting column by 2.
            shl     ax, 1
            add     bx, ax                  ;Total offset in BX.

            add     si, bx                  ;Add offset to SI.
                                            ;SI now contains offset in source buffer.

            mov     ax, 0b800h              ;Set ES to CGA video buffer.
            mov     es, ax

                                            ;Calculate video memory offset
                                            ;     = (row * 160) + (col * 2)
            mov     di, frame.ulr_d         ;Place row in DI and multiply by 160.
            mov     ax, di
            mov     cl, 7
            shl     di, cl
            mov     cl, 5
            shl     ax, cl
            add     di, ax

            mov     ax, frame.ulc_d         ;Multiply col by 2.
            shl     ax, 1
            add     di, ax                  ;Add (col * 2) to DI.
                                            ;DI now contains video memory offset.

                                            ;Calculate number of rows in BL.
            mov     bl, byte ptr frame.lrr_s
            sub     bl, byte ptr frame.ulr_s
            inc     bl
                                            ;Calculate number of bytes/row in BH.
            mov     bh, byte ptr frame.lrc_s
            sub     bh, byte ptr frame.ulc_s
            inc     bh
            shl     bh, 1

            mov     ah, 160                 ;Calculate SI, DI increment in AH.
            sub     ah, bh

            mov     dx, 3dah                ;Keep crt status register in DX.

            cld
            mov     ch, 0                   ;Only need LSB of CX.
```

```
b01:        mov     cl, bh              ;Load bytes/row into CX.

                                        ;Pause until the beginning of a
                                        ;vertical retrace.
b02:        in      al, dx              ;Loop to complete any retrace period
            test    al, 8               ;in progress.
            jnz     b02
b03:        in      al, dx              ;Loop until start of a new vertical
            test    al, 8               ;retrace.
            jz      b03

            rep     movsb               ;Move entire row during vertical retrace.
            mov     cl, ah              ;AH stores SI, DI increment value.
            add     si, cx              ;Adjust SI, DI to start of next row.
            add     di, cx
            dec     bl                  ;Decrement row counter.
            jz      b06                 ;Quit if no more rows.

                                        ;Move another row, byte by byte, while
                                        ;waiting for next vertical retrace.
                                        ;Pause until the beginning of a
                                        ;horizontal retrace.
            mov     cl, bh              ;Loop to complete any retrace period
b04:        in      al, dx              ;in progress.

            test    al, 1
            jnz     b04
b05:        in      al, dx              ;Loop until start of a new horizontal
            test    al, 1               ;retrace.
            jz      b05
            movsb                       ;Move one byte.
            loop    b04                 ;Loop until entire row is moved.
            mov     cl, ah
            add     si, cx              ;Adjust SI, DI to start of next row.
            add     di, cx
            dec     bl                  ;Decrement row counter.
            jz      b06                 ;Quit if no more rows.
            jmp     b01                 ;Go back to move another row during the
                                        ;next vertical retrace.

b06:        pop     si                  ;Restore registers.
            pop     di
            pop     bp
            ret                         ;Return to C program.

_printwcga endp

_text       ends
            end
```

上面是C程序用于显示窗口的汇编子程序。

窗口函数工作原理如下:程序在一个4000字节的数组中存储产生整个屏幕的数据。此数组为视频内存区的准确映像(由ASCII代码和属性值交替地组成)。然后窗口函数printw或printwcga将一个连续的数据块从程序数组直接传送到视频内存区,产生所有字符及其显示属性的立即显示。下面的经初始化的C数组可用于存储视频数据(注意,经初始化的数组必须外部进行声明,或函数内部作为static数组):

char screen[4000]={'h',15,'e'15,'l',15,'l', 5'o',…etc};

该缓冲区以黑底白字 hello 开始．在实际程序中,应初始化用于整个屏幕的所有 4000 个字节．然后可使用 printw 或 printwcga 在屏幕任意位置显示此数组的任意部分．例如,欲显示出整个数组,使用下面的函数调用:

printw(screen,0,0,24,79,0,0);

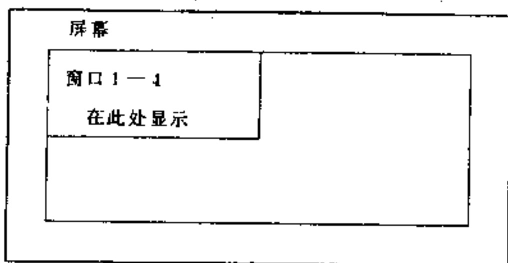
此方法要求在源代码文件中对整个 4000 字节的数组初始化,这很不实际．解决办法是使用一个交互式的屏幕设计程序来设计整个屏幕,然后由其自动生成初始化此 4000 字节数组所必须的 C 源代码．后文给出了此类应用程序的一个简单版本．使用一个屏幕设计程序以产生视频数据,然后用 printw 或 printwcga 来显示它．这有三大优点：一是屏幕显示非常快．所有屏幕数据以其最终格式存储在可执行程序中,故可不经内部处理很快地传送到视频内存区．二是此数组将每个字符的属性单独存贮,故屏幕设计时可具有最大的灵活性．三是使用交互式设计程序很容易设计出一个漂亮的屏幕,所看到的便是将来你可以显示的．仅由发出一连串单个字符串的命令很难实现完美的屏幕设计．

这便出现了另一个问题,即欲产生多个屏幕就必须编译并存储大量数据．运行时每屏需使用 4000 字节数据空间,而一个源代码文件需数倍于此量．这可通过以下几个步骤来帮助克服此困难:

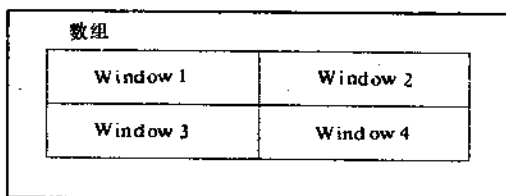
首先,将所有屏幕数据安放在另一个源代码文件中,以后在连接时将其并入主应用程序．当主代码部分每次编译时,此数据不必再处理．

其次,当设计的显示窗口比 25×80 屏幕小时，要尽可能将多个窗口压缩到一个数组中．例如,

某应用程序将四个窗口均放在屏幕的左上角矩形中显示:



在此情况下,四个窗口可放置在一个数组中,如下图:



有助于减少存储需求的另一个办法,是将屏幕数据存储在磁盘的独立文件中,每个文件包含一个或多个屏幕。该程序可保留一个或多个 4000 字节的数组,运行时可读入屏幕数据,然后再用 printw 或 printwcga 将其显示出来。显然,此方法将减慢程序的执行,除非明显缺少 RAM 空间一般不用此法。但数据一旦被读入,窗口的实际显示仍是立即的。第八章介绍了如何使用扩展内存(Lotus-Intel-Microsoft 标准)在 640KDOS 限制以上存储屏幕数据。

虽然屏幕上有许多重复的字符(特别是空格字符),但该数组本身并不包含连接的重复字符,因为属性代码与 ASCII 值交替出现。因此无法使用压缩技术(如 Microsoft 的 EXEPACK 实用程序)来消除连续重复的字符。

▲printw

函数 printw 将一个数据块从程序数组中直接传送到视频内存区,且仅当显示适配器不是 CGA 时使用. 所需数据块由传送给源数组中的左上角和右下角行、列来指定. 该数据块可在实际屏幕上任意位置上显示,仅需给出开始的行与列即可. 此函数传递以下七个参数:

· 源数组地址

· 源数组中欲被显示数据块的左上角行(0–24)

· 该数据块的左上角列(0–79)

· 该数据块的右下角行

· 该数据块的右下角列

· 屏幕上目的地的起始行

· 屏幕上目的地的起始列

例如欲在屏幕左上角显示上图中的窗口 4,使用下面的函数调用:

printw(screen,12,40,24,79,0,0);

函数 printw 执行以下主要步骤: 首先使用前两个参数计算 screen 数组中源数据块开始的偏移量,将结果放置在寄存器 DI 中; 然后用测试地址 0000h:0440h 处由 BIOS 维护的视频方式标志,并将得到的视频内存区段地址赋给寄存器 ES;再使用最后两个参数计算出视频内存区的起始偏移量,将此结果置于寄存器 DI 中; 接着该函数使用前 4 个参数计算出欲移动的行数(此值置于寄存器 BL 中)与每行的字节数(此值置于寄存器 DX 中); 最后,printw 使用 rep movs 指令重复地移动每一行. 由于欲被传送的数据不一定是连续的(仅当数据块为80 列宽时,数据才是连续的),故不可使用单块移动的方法.

▲printwcga

函数 printwcga 是 printw 用于 CGA 控制器的一种特殊形式。象函数 printvcga 一样,使用回扫周期作写同步。但 printwcga 同时使用水平与垂直回扫周期,数据传送分成两个阶段:

(1) 第一行传送前,程序暂停等待垂直回扫开始(由端口地址 3DAh 处状态寄存器比特 3 置 1 表示)。垂直回扫等待的过程基本上与以前讲述的水平回扫的过程相同。但垂直回扫是一个时间长得多的事件,故有可能在垂直回扫期间传送多达 400 字节(而水平回扫期间只能传送一个字节)。一行的最大尺寸是 160 字节,故可安全地传送完。

(2) 程序在等待下一个垂直回扫时,有时间传送多达 150 个字节,传送每个字节前均暂停以等待一个水平回扫的开始。在此期间程序还可以传送另一行数据。

然后程序循环回来再次在传送第三行之前等待一个垂直回扫。此过程(在垂直回扫期间传送整整一行,在水平回扫期间一个字节一个字节地传送下一行)连续执行直至所有行被移动完毕。

6.4 基准测试(Benchmark)

下面的程序可获得各种在屏幕上显示数据方法之效率的定量比较.测试了七种不同的方法,用每种方法写十整屏数据.每一屏由给定属性的2000个 x 字符组成。在屏幕显示前、后立即调用函数 gettime 实现计时,此时间经计算存贮在浮点数组 time 中。函数 gettime 使用 DOS 功能 2Ch 获得接近 1/100 秒的系统时间。

```c
     File:     BMARK.C

     A C program comparing methods for generating video output

     This program must be linked with object modules containing the external
     functions called.  All of these functions are listed in other figures.

     The program prints 10 screens of data for each of 7 different methods
     of video output, measures the elapsed times, and displays the results.
*/

#include <stdio.h>

extern char scr[4000];                                 /* Contains screen data. */

                                      /* Declarations of external functions. */
void rawmode   (void);                                         /* Figure 4.8 */
void cookmode  (void);                                         /* Figure 4.8 */
void printa    (char *, int, int, int);                        /* Figure 5.2 */
void cls       (int, int, int, int);                           /* Figure 6.6 */
void printv    (char *, int, int, int);                        /* Figure 8.4 */
void printvcga (char *, int, int, int);                        /* Figure 8.4 */
void printw    (char *, int, int, int, int, int, int);         /* Figure 8.5 */
void printwcga (char *, int, int, int, int, int, int);         /* Figure 8.5 */

void gettime   (int *,int *,int *,int *);                /* Internal function. */

void main ()
     {
     register int s, r;                           /* Loop counters.          */
     float time[7];                               /* Stores elapsed times.   */
     int h1, h2, m1, m2, s1, s2, c1, c2;          /* Temporary variables to  */
     float timebefore, timeafter;                 /* calculate times.        */
     static char *mess [] =
          {                                     /* Method description table. */
          "dos/ansi.sys, cooked:",
          "dos/ansi.sys, raw:   ",
          "bios:                ",
          "direct video:        ",
          "direct video, CGA:   ",
          "block move:          ",
          "block move, CGA:     "
          };

     gettime (&h1,&m1,&s1,&c1);   /* DOS/ANSI.SYS method, cooked file mode. */
     for (s=1;s<=10;++s)          /* Write 10 screens of 'x' in red.        */
          for (r=1;r<=25;++r)     /* Write 25 lines.                        */
               {                  /* Set color to red and position cursor.  */
               printf ("\x1b[31m\x1b[%d;1H",r);
               printf ("xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx\
xxxxxxxxxxxxxxxxxxxxxxxx");
               }
     gettime (&h2,&m2,&s2,&c2);

     timebefore = (float)(h1*3600.0 + m1*60.0 + s1 + c1/100.0);
     timeafter  = (float)(h2*3600.0 + m2*60.0 + s2 + c2/100.0);
     time[0] = timeafter - timebefore;
     printf ("\x1b[0m");                     /* Restore display attributes.    */
     cls (0,0,24,79);                        /* Clear screen.                  */

     rawmode ();                             /* DOS/ANSI.SYS method, raw file mode. */
     gettime (&h1,&m1,&s1,&c1);
     for (s=1;s<=10;++s)                     /* Write 10 screens of 'x' in magenta. */
          for (r=1;r<=25;++r)                /* Write 25 lines.                */
               {                             /* Set color to magenta and position */
                                             /* cursor.                        */
               printf ("\x1b[35m\x1b[%d;1H",r);
               printf ("xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx\
xxxxxxxxxxxxxxxxxxxxxxxx");
               }
     gettime (&h2,&m2,&s2,&c2);
     timebefore = (float)(h1*3600.0 + m1*60.0 + s1 + c1/100.0);
     timeafter  = (float)(h2*3600.0 + m2*60.0 + s2 + c2/100.0);
     time[1] = timeafter - timebefore;
     printf ("\x1b[0m");                        /* Restore display attributes.  */
     cookmode ();                               /* Restore file mode.           */
     cls (0,0,24,79);                           /* Clear screen.                */
```

```
     gettime (&h1,&m1,&s1,&c1);            /* BIOS method.                       */
     for (s=1;s<=10;++s)                   /* Write 10 screens of 'x' in green.  */
          for (r=0;r<=24;++r)              /* Write 25 lines.                    */
               printa ("xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx\
xxxxxxxxxxxxxxxxxxxxxxxxxxxx",02,r,0);
     gettime (&h2,&m2,&s2,&c2);
     timebefore = (float)(h1*3600.0 + m1*60.0 + s1 + c1/100.0);
     timeafter  = (float)(h2*3600.0 + m2*60.0 + s2 + c2/100.0);
     time[2] = timeafter - timebefore;
     cls (0,0,24,79);

     gettime (&h1,&m1,&s1,&c1);            /* Direct video string writes.        */
     for (s=1;s<=10;++s)                   /* Write 10 screens of 'x' in blue.   */
          for (r=0;r<=24;++r)              /* Write 25 lines.                    */
               printv ("xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx\
xxxxxxxxxxxxxxxxxxxxxxxxxxxx",01,r,0);
     gettime (&h2,&m2,&s2,&c2);
     timebefore = (float)(h1*3600.0 + m1*60.0 + s1 + c1/100.0);
     timeafter  = (float)(h2*3600.0 + m2*60.0 + s2 + c2/100.0);
     time[3] = timeafter - timebefore;
     cls (0,0,24,79);

     gettime (&h1,&m1,&s1,&c1);            /* Direct video string writes         */
                                           /* using "nosnow" version.            */
     for (s=1;s<=10;++s)                   /* Write 10 screens of 'x' in cyan.   */
          for (r=0;r<=24;++r)              /* Write 25 lines.                    */
               printvcga ("xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx\
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",03,r,0);
     gettime (&h2,&m2,&s2,&c2);
     timebefore = (float)(h1*3600.0 + m1*60.0 + s1 + c1/100.0);
     timeafter  = (float)(h2*3600.0 + m2*60.0 + s2 + c2/100.0);

     time[4] = timeafter - timebefore;
     cls (0,0,24,79);

     gettime (&h1,&m1,&s1,&c1);            /* Direct video block move.           */
     for (s=1;s<=10;++s)                   /* Write 10 complete text screens.    */
          printw (scr,0,0,24,79,0,0);
     gettime (&h2,&m2,&s2,&c2);
     timebefore = (float)(h1*3600.0 + m1*60.0 + s1 + c1/100.0);
     timeafter  = (float)(h2*3600.0 + m2*60.0 + s2 + c2/100.0);
     time[5] = timeafter - timebefore;
     cls (0,0,24,79);

     gettime (&h1,&m1,&s1,&c1);            /* Direct video block move using      */
                                           /* "nosnow" version.                  */
     for (s=1;s<=10;++s)                   /* Write 10 complete text screens.    */
          printwcga (scr,0,0,24,79,0,0);
     gettime (&h2,&m2,&s2,&c2);
     timebefore = (float)(h1*3600.0 + m1*60.0 + s1 + c1/100.0);
     timeafter  = (float)(h2*3600.0 + m2*60.0 + s2 + c2/100.0);
     time[6] = timeafter - timebefore;
     cls (0,0,24,79);

     printf ("\nResults of Benchmark\n");            /* Display final results. */
     printf ("------- -- ---------\n");
     printf ( "method                          time (seconds)\n");
     printf ( "------                          --------------\n");
     for (s=0;s<=6;++s)
          printf ("%s%8.2f\n", mess[s],time[s]);

     } /* end main */

#include <dos.h>

void gettime (h,m,s,c)                     /* This function uses MS-DOS to */
int *h,*m,*s,*c;                           /* obtain the system time.      */
/*
     h : hours
     m : minutes
     s : seconds
     c : hundredths ("centi-seconds")

     {
     union REGS reg;

     reg.h.ah = 0x2c;
     int86 (0x21,&reg,&reg);
```

```
    *h = reg.h.ch;
    *m = reg.h.cl;
    *s = reg.h.dh;
    *c = reg.h.dl;

} /* end gettime */
```

上面是对各种视频输出方法作基准测试的C程序。

注意,上例并未包括整个程序,屏幕数据数组及除main,gettime之外的所有函数已假定存在于其他目标模块中. 屏幕数据可使用屏幕设计程序产生,并放置于另一个C源文件中,其他所有必须的函数均可在本书中找到.

下表给出了此程序在标准IBM-PC(4.77MHz)上采用EGA显示时得到的结果.

方法	函数	时间(秒)
DOS / ANSI.SYS,加工后	printf	52.95
DOS / ANSI.SYS,原始	printf	34.49
BIOS	ptinta	35.15
直接视频显示	printv	0.38
直接视频显示,CGA	printvcga	2.53
块移动	printw	0.22
块移动,CGA	printwcga	2.19

从这些数据中可以看出:

(1) 对于DOS功能调用方法,当控制台设备设置为未加工方式(原始方式)而不是默认的经加工方式时,速度有明显改进。

(2) 使用BIOS功能的函数printa效果不大好. 原因之一是,使用BIOS产生控制后的视频输出每个字符显示需调用两次中断功能;原因之二是,printa函数是用C语言写的,而直接视频函数是用汇编语言写的,通过C函数int86产生中断比汇编语言的直接中断调用要慢,因此影响了printa的速

度(这使得比较有些不公平)。

(3) CGA 方法比不需使用回扫周期进行同步的方法明显慢。

(4) 在直接对视频内存区写的各种方法中,块传输方法只比字符串写方法快一点。但采用块传送方法,设计和显示整屏或窗口式的数据更方便。

(5) 直接视频方法比使用操作系统或 BIOS 方法快两个数量级,且直接视频传送更灵活,实现也更简单。

6.5 屏幕生成程序例

下面为一个交互式屏幕设计实用程序的完整汇编语言源代码。此程序为使用了系统级资源的一个较长的汇编程序。下面介绍该程序的作用、原理以及改进建议。

```
page 50,130

;File:      SM.ASM

;An .EXE format for an interactive screen designing utility
;
;     To generate .EXE file from SM.ASM:
;           MASM SM;
;           LINK SM;

INITATTR    equ     0fh                         ;Initial attribute.

codeseg     segment 'CODE'
codeseg     ends

stackseg    segment stack 'STACK'
            db      128 dup ('stack   ')
stackseg    ends

dataseg     segment 'DATA'

sw_rec      record  color:1=0                   ;Global flags.
switches    sw_rec  <>
                                                ;Messages.
f2_1        db      'foreground',0
f2_2        db      '+  toggle blinking',0
f2_3        db      'SELECT COLOR ATTRIBUTE',0
f2_4        db      27,26,' select foreground',0
f2_5        db      24,25,' select background',0
f2_6        db      'SELECT MONOCHROME ATTRIBUTE',0
f2_7        db      27,26,' select attribute',0
f3_1        db      'name of file to read:',0
f3_2        db      49 dup (' '),0
f3_3        db      'file not found, press a key to continue ...',0
f4_1        db      'name of file to write:',0
f4_2        db      'cannot write file, press a key to continue ...',0
f10_1       db      'terminate program? (y/n): ',0

                                                ;Table of monochrome attributes
                                                ;used by 'f2'.
attrtab     db      07h                         ;Normal.
            db      0Fh                         ;High intensity.
            db      01h                         ;Underlined.
```

```
        db      09h                     ;High intensity and underlined.
        db      70h                     ;Reverse video.
        db      7fh                     ;White background/intense foreground
                                        ;(visible only on some systems).
TABLEN  equ     $-attrtab

filename db     50 dup (?)              ;Used by 'f3' & 'f4'.
handle  dw      ?

oldcur  dw      ?                       ;Used by 'hidecur' & 'showcur'.
curpos  dw      ?                       ;Used by 'savecur' & 'restcur'.

                                        ;Data for 'pushscr' & 'popscr'.
stkptr  dw      offset scrstack         ;Pointer for screen stack.
scrstack label  byte                    ;Stack space for saving screens --
                                        ;must be LAST in data segment to avoid
                                        ;reserving space and bloating .EXE file.

dataseg ends

        assume  cs:codeseg
        assume  ds:dataseg
        assume  es:dataseg
        assume  ss:stackseg

codeseg segment 'CODE'
                                        ;** Macros **
cursor  macro                           ;Positions cursor at row/col
        push    ax                      ;in DH/DL -- uses BIOS.
        push    bx
        mov     ah,02h
        mov     bh,00h
        int     10h
        pop     bx
        pop     ax
        endm

putchar macro                           ;Writes character & attribute
                                        ;directly to video memory.
        push    ax
        push    bx                      ;Entry:
                                        ;       CX = attribute/character
        xor     ah, ah                  ;       DX = row/column
        mov     al, dh                  ;       ES = segment address of
        xor     bh, bh                  ;            video memory
        mov     bl, dh
        push    cx                      ;Calculates offset in video memory =
        mov     cl, 7                   ;       (row * 160) + (2 * column)
        shl     ax, cl                  ;Places result in BX.
        mov     cl, 5
        shl     bx, cl
        pop     cx
        add     ax, bx
        xor     bh, bh
        mov     bl, dl
        shl     bx, 1
        add     bx, ax
        mov     word ptr es:[bx], cx    ;Write character & attribute.

        pop     bx
        pop     ax
        endm

putattr macro                           ;Writes attribute only directly
                                        ;to video memory.
        push    ax

        push    bx                      ;Entry:
                                        ;       CH = attribute
        xor     ah, ah                  ;       DX = row/column
        mov     al, dh                  ;       ES = segment address of
        xor     bh, bh                  ;            video memory
        mov     bl, dh
        push    cx                      ;Calculates offset in video memory =
        mov     cl, 7                   ;       (row * 160) + (2 * column)
        shl     ax, cl                  ;Places result in BX.
        mov     cl, 5
        shl     bx, cl
        pop     cx
        add     ax, bx
        xor     bh, bh
```

```
        mov     bl, dl
        shl     bx, 1
        add     bx, ax
        mov     byte ptr es:[bx+1], ch     ;Write attribute to video memory.

        pop     bx
        pop     ax
        endm

getchar macro                       ;Gets character & attribute.
                                    ;Directly from video memory.
        push    ax                  ;Alters register: CX.
        push    bx                  ;Entry:
                                    ;       BX = row/column
        xor     ah, ah              ;       ES = segment address of
        mov     al, dh              ;            video memory
        xor     bh, bh              ;Returns:
        mov     bl, dh              ;       CX = attribute/character
        mov     cl, 7               ;            at row/col in BX.
        shl     ax, cl
        mov     cl, 5               ;Calculates offset in video memory =
        shl     bx, cl              ;       (row * 160) + (2 * column).
        add     ax, bx              ;Places result in BX.
        xor     bh, bh
        mov     bl, dl
        shl     bx, 1
        add     bx, ax
        mov     cx, word ptr es:[bx]       ;Read character and attribute.
        pop     bx
        pop     ax
        endm

cls     macro   ulr, ulc, lrr, lrc  ;Clears the screen within the specified
                                    ;dimensions -- uses BIOS.
        push    ax
        push    bx
        push    cx
        push    dx
        mov     ah, 6               ;Scroll window up function.
        xor     al, al              ;0 => clear entire window.

        mov     ch, ulr             ;Specify the window.
        mov     cl, ulc
        mov     dh, lrr
        mov     dl, lrc
        mov     bh, 0fh             ;Attribute to use for blanked area
        int     10h                 ;(= white on black/high intensity).
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        endm


main    proc    far                 ;** Main program. **

        mov     ax, dataseg         ;Initialize data segment.
        mov     ds, ax

                                    ;** Initialization. **
        xor     ax, ax              ;Set ES to correct video buffer segment.
        mov     es, ax
        cmp     byte ptr es:[449h], 7      ;Test video mode.
        je      a01
        mov     ax, 0b800h          ;Color buffer.
        or      switches, mask_color       ;Set color flag.
        jmp     a02
a01:    mov     ax, 0b000h          ;Monochrome buffer.
a02:    mov     es, ax
        cls     0,0,24,79           ;Macro: clear entire screen.
        xor     dx, dx              ;Initialize cursor at 0,0.
        cursor                      ;Macro.
        mov     ch, INITATTR        ;Initialize attribute.
        mov     cl, 254             ;Initialize character to '■'.
```

```
                                                ;** Begin keyboard read loop. **
a03:        mov     ah, 0                       ;Invoke BIOS keyboard read service.
            int     16h

            cmp     ah, 01                      ;Test for escape.
            je      a03                         ;Return to top of loop.

            cmp     ah, 3bh                     ;Test F1: select character.
            jne     a04
            push    cx
            push    dx
            call    f1
            pop     dx
            pop     cx
            mov     cl, al                      ;Character returned in AL.
            jmp     a03                         ;Return to top of loop.

a04:        cmp     ah, 3ch                     ;Test F2: select attribute.
            jne     a05
            push    cx
            push    dx

            call    f2
            pop     dx
            pop     cx
            mov     ch, ah                      ;Attribute returned in AH.
            jmp     a03                         ;Return to top of loop.
a05:        cmp     ah, 3dh                     ;Test F3: read file.
            jne     a06
            push    cx
            push    dx
            call    f3
            pop     dx
            pop     cx
            jmp     a03                         ;Return to top of loop.

a06:        cmp     ah, 3eh                     ;Test F4: write file.
            jne     a07
            push    cx
            push    dx
            call    f4
            pop     dx
            pop     cx
            jmp     a03                         ;Return to top of loop.

a07:        cmp     ah, 44h                     ;Test F10: quit.
            jne     a08
            push    cx
            push    dx
            call    f10
            pop     dx
            pop     cx
            jmp     a03                         ;Return to top of loop.

a08:        cmp     ah, 48h                     ;Test up arrow.
            jne     a11
            cmp     al, 38h                     ;Test up-shift.
            je      a09                         ;Skip 'putchar' if shift.
            putchar                             ;Macro.
a09:        cmp     dh, 0                       ;Test if at top limit.
            je      a10                         ;Skip 'cursor' if movement not possible.
            dec     dh
            cursor                              ;Macro.
a10:        jmp     a03                         ;Return to top of loop.

a11:        cmp     ah, 50h                     ;Test down arrow.
            jne     a14
            cmp     al, 32h                     ;Test down-shift.
            je      a12                         ;Skip 'putchar' if shift.
            putchar                             ;Macro.
a12:        cmp     dh, 24                      ;Test if at bottom limit.
            je      a13                         ;Skip 'cursor' if movement not possible.
            inc     dh
            cursor                              ;Macro.
a13:        jmp     a03                         ;Return to top of loop.

a14:        cmp     ah, 4bh                     ;Test left arrow.
            jne     a17
            cmp     al, 34h                     ;Test left-shift.
            je      a15                         ;Skip 'putchar' if shift.
            putchar                             ;Macro.
```

```
a15:    cmp     dl, 0           ;Test if at left limit.
        je      a16             ;Skip 'cursor' if movement not possible.
        dec     dl
        cursor                  ;Macro.
a16:    jmp     a03             ;Return to top of loop.

a17:    cmp     ah, 4dh         ;Test right arrow.
        jne     a20
        cmp     al, 36h         ;Test right-shift.
        je      a18             ;Skip 'putchar' if shift.
        putchar                 ;Macro.
a18:    cmp     dl, 79          ;Test if at right limit.
        je      a19             ;Skip 'cursor' if movement not possible.
        inc     dl
        cursor                  ;Macro.
a19:    jmp     a03             ;Return to top of loop.
a20:    cmp     ah, 0eh         ;Test backspace.
        jne     a22
        cmp     dl, 0           ;Move cursor back if possible.
        je      a21
        dec     dl
        cursor                  ;Macro.
        push    cx              ;Write a space.
        mov     cl, 32
        putchar                 ;Macro.
        pop     cx
a21:    jmp     a03             ;Return to top of loop.

a22:    cmp     al, 0           ;Test for no ASCII value.
        je      a23             ;Go back if no ASCII value.
        jmp     a24
a23:    jmp     a03             ;Return to top of loop.

a24:    push    cx              ;Write out ASCII character.
        mov     cl, al
        putchar                 ;Macro.
        pop     cx
        cmp     dl, 79          ;Move cursor right if possible.
        je      a25
        inc     dl
        cursor                  ;Macro.
a25:    jmp     a03             ;** End keyboard read loop. **

main    endp

                                ;** Begin subroutines. **
frame   equ     [bp]


f1      proc    near            ;** F1: select character. **
                                ;Selected character returned in AL.

        push    cx              ;Save current attribute/character.

        call    pushscr         ;Save old screen.
        call    hidecur         ;Hide the cursor.

        mov     ax, 36          ;Draw box 0,0 to 24,36.
        push    ax
        mov     ax, 24
        push    ax
        xor     ax, ax
        push    ax
        push    ax
        call    box
        add     sp, 8

        cls     1,1,23,35       ;Macro: clear inside of box.

        mov     ch, INITATTR    ;Write characters.
        mov     cl, 1           ;Start with ASCII 1.
        mov     dh, 1           ;Start w/ row 1.
        mov     dl, 3           ;Start w/ column 3.

        pop     ax              ;Restore current att/char in AX.

                                ;Start of loop.
b01:    putchar                 ;Macro.
```

```
        cmp     cl, al          ;Test for current character.
        jne     b02
        mov     bx, dx          ;Save row/col of current char. in BX.
b02:    inc     cl              ;Step through all ASCII characters.
        cmp     cl, 32          ;Skip space.
        jne     b03
        inc     cl
b03:    cmp     cl, 255         ;Test for last character.
        je      b05
        inc     dh              ;Go to next row.
        cmp     dh, 24          ;Test for last row.
        je      b04
        jmp     b01
b04:    inc     dl              ;Increment column.
        inc     dl
        inc     dl
        mov     dh, 1           ;Start over with row 1.
        jmp     b01             ;Return to top of loop.
b05:    mov     dx, bx          ;Start by highlighting current char.
        mov     ch, 112         ;Reverse video attribute.
        putattr                 ;Macro.
b06:    mov     ah, 0           ;Begin main read loop.
        int     16h
        cmp     ah, 48h         ;Test up arrow.
        jne     b07
        cmp     dh, 1           ;See if movement up is possible.
        je      b06             ;Return to top of loop if not possible.
        mov     ch, INITATTR    ;Restore attribute.
        putattr                 ;Macro.
        dec     dh              ;Move row counter up.
        mov     ch, 112         ;Display highlight at new position.
        putattr                 ;Macro.
        jmp     b06             ;Return to top of loop.
b07:    cmp     ah, 50h         ;Test down arrow.
        jne     b09
        cmp     dh, 23          ;See if movement down is possible.
        je      b08             ;Return to top of loop if not possible.
        mov     ch, INITATTR    ;Restore attribute.
        putattr                 ;Macro.
        inc     dh              ;Move row counter down.
        mov     ch, 112         ;Display highlight at new position.
        putattr                 ;Macro.
b08:    jmp     b06             ;Return to top of loop.
b09:    cmp     ah, 4bh         ;Test left arrow.
        jne     b11
        cmp     dl, 3           ;See if movement left is possible.
        je      b10             ;Return to top of loop if not possible.
        mov     ch, INITATTR    ;Restore attribute.
        putattr                 ;Macro.
        sub     dl, 3           ;Move column counter left.
        mov     ch, 112         ;Display highlight at new position.
        putattr                 ;Macro.
b10:    jmp     b06             ;Return to top of loop.
b11:    cmp     ah, 4dh         ;Right arrow.
        jne     b13
        cmp     dl, 33          ;Test if movement right is possible.
        je      b12             ;Return to top of loop if not possible.
        mov     ch, INITATTR    ;Restore attribute.
        putattr                 ;Macro.
        add     dl, 3           ;Move column counter right.
        mov     ch, 112         ;Display highlight at new position.
        putattr                 ;Macro.
b12:    jmp     b06             ;Return to top of loop.
b13:    cmp     ah, 01          ;Test for escape.
        jne     b14
        getchar                 ;Macro.
        push    ax              ;Save returned attribute/character.
        call    popscr          ;Restore screen.
        call    showcur         ;Restore cursor.
        pop     ax              ;Retrieve attribute/character into AX.
        ret
b14:    jmp     b06             ;End of main keyboard loop.
```

```
f1          endp

f2          proc    near                ;** F2: Select attribute. **
                                        ;Selected attribute returned in AH.

                                        ;Template for attribute byte.
attr_rec    record  blink:1, bckgnd:3, forgnd:4

            push    cx                  ;Save current attribute.
            call    pushscr             ;Save old screen.
            call    hidecur             ;Hide the cursor.

            mov     ax, 39              ;Draw outer box:  0,0 to 14,39.
            push    ax
            mov     ax, 14
            push    ax
            xor     ax, ax
            push    ax
            push    ax
            call    box
            add     sp, 8

            cls     1,1,13,38           ;Macro: clear inside of box.

            mov     ax, 30              ;Draw inner box:  4,9 to 6,30.
            push    ax
            mov     ax, 6
            push    ax
            mov     ax, 9
            push    ax
            mov     ax, 4
            push    ax
            call    box
            add     sp, 8
                                        ;Message 1.
            mov     ax, 15              ;Column.
            push    ax
            mov     ax, 5               ;Row.
            push    ax
            mov     ax, 0fh             ;Attribute =
            push    ax                  ;white on black/high intensity.
            mov     ax, offset f2_1
            push    ax
            call    printv
            add     sp, 8               ;Restore stack.

                                        ;Message 2.
            mov     ax, 10              ;Column.
            push    ax
            mov     ax, 11              ;Row.
            push    ax
            mov     ax, 0fh             ;Attribute =
            push    ax                  ;white on black/high intensity.
            mov     ax, offset f2_2
            push    ax
            call    printv
            add     sp, 8               ;Restore stack.

                                        ;Color and mono differ from this point.
            test    switches, mask color
            jnz     c01
            jmp     c17
                                        ;** Beginning of color routine. **

                                        ;Message 3.
c01:        mov     ax, 9               ;Column.
            push    ax
            mov     ax, 2               ;Row.
            push    ax
            mov     ax, 70h             ;Attribute =
            push    ax                  ;black on white.
            mov     ax, offset f2_3
            push    ax
            call    printv
            add     sp, 8               ;Restore stack.

                                        ;Message 4.
            mov     ax, 10              ;Column.
            push    ax
```

```
        mov     ax, 9               ;Row.
        push    ax
        mov     ax, 0fh             ;Attribute =
        push    ax                  ;white on black.
        mov     ax, offset f2_4
        push    ax
        call    printv
        add     sp, 8               ;Restore stack.

                                    ;Message 5.
        mov     ax, 10              ;Column.
        push    ax
        mov     ax, 10              ;Row.
        push    ax
        mov     ax, 0fh             ;Attribute =
        push    ax                  ;white on black.
        mov     ax, offset f2_5
        push    ax
        call    printv
        add     sp, 8               ;Restore stack.

        pop     cx                  ;Restore current attribute.

c02:                                ;Begin display/read/modify attr. loop.

        mov     dh, 5               ;Display the current attribute in the
        mov     dl, 10              ;inner box.
c03:    putattr
        inc     dl
        cmp     dl, 30
        je      c04
        jmp     c03

c04:    mov     ah, 0               ;Read keyboard.
        int     16h

        cmp     ah, 48h             ;Test for up arrow.
        jne     c05
        mov     bh, ch              ;Increment background attribute.
        and     bh, mask bckgnd
        mov     cl, bckgnd
        shr     bh, cl
        inc     bh
        shl     bh, cl
        and     bh, mask bckgnd
        and     ch, not mask bckgnd
        or      ch, bh
        jmp     c02                 ;Return to top of loop.

c05:    cmp     ah, 50h             ;Test down arrow.
        jne     c06
        mov     bh, ch              ;Decrement background attribute.
        and     bh, mask bckgnd
        mov     cl, bckgnd
        shr     bh, cl
        dec     bh
        shl     bh, cl
        and     bh, mask bckgnd
        and     ch, not mask bckgnd
        or      ch, bh
        jmp     c02                 ;Return to top of loop.

c06:    cmp     ah, 4bh             ;Test left arrow.
        jne     c07
        mov     bh, ch              ;Decrement foreground attribute.
        and     bh, mask forgnd
        dec     bh
        and     bh, mask forgnd
        and     ch, not mask forgnd
        or      ch, bh
        jmp     c02                 ;Return to top of loop.

c07:    cmp     ah, 4dh             ;Test right arrow.
        jne     c08
        mov     bh, ch              ;Increment foreground attribute.
        and     bh, mask forgnd
        inc     bh
        and     bh, mask forgnd
        and     ch, not mask forgnd
        or      ch, bh
        jmp     c02                 ;Return to top of loop.
```

```
c08:    cmp     ah, 4eh                 ;Test grey +.
        jne     c09
        xor     ch, mask blink          ;Toggle blink bit.
        jmp     c02                     ;Return to top of loop.

c09:    cmp     ah, 1                   ;Test for escape.
        jne     c10
        jmp     c23                     ;Go to quit on escape.

c10:    jmp     c02                     ;Return to start of loop.
                                        ;End of main loop.

                                        ;** Beginning of monochrome routine. **

c11:                                    ;Message 6.
        mov     ax, 6                   ;Column.
        push    ax
        mov     ax, 2                   ;Row.
        push    ax
        mov     ax, 70h                 ;Attribute =
        push    ax                      ;reverse video.
        mov     ax, offset f2_6
        push    ax
        call    printv
        add     sp, 8                   ;Restore stack.

                                        ;Message 7.
        mov     ax, 10                  ;Column.
        push    ax
        mov     ax, 10                  ;Row.
        push    ax
        mov     ax, 0fh                 ;Attribute =
        push    ax                      ;white on black.
        mov     ax, offset f2_7
        push    ax
        call    printv
        add     sp, 8                   ;Restore stack.

        pop     cx                      ;Restore current attribute.
        mov     bx, offset attrtab + 1      ;Initialize attribute table ptr.

c12:                                    ;Begin display/read/modify attr. loop.

        mov     dh, 5                   ;Display the current attribute in the
        mov     dl, 10                  ;inner box.
c13:    putattr
        inc     dl
        cmp     dl, 30
        je      c14
        jmp     c13

c14:    mov     ah, 0                   ;Read keyboard.
        int     16h

        cmp     ah, 4bh                 ;Test left arrow.
        jne     c17
c15:    dec     bx
        cmp     bx, offset attrtab      ;If gone below bottom of table, wrap
        jae     c16                     ;around to the top.
        add     bx, TABLEN
c16:    cmp     ch, [bx]                ;Prevent repeat of same attribute on
        je      c15                     ;first time thru loop.
        mov     ch, [bx]                ;Assign CH the attribute from table.
        jmp     c12                     ;Return to top of loop.

c17:    cmp     ah, 4dh                 ;Test right arrow.
        jne     c20
c18:    inc     bx
        cmp     bx, offset attrtab + TABLEN - 1
        jbe     c19                     ;If gone past end of table, go to start.
        mov     bx, offset attrtab
c19:    cmp     ch, [bx]                ;Prevent repeat of same attribute on
        je      c18                     ;first time thru loop.
        mov     ch, [bx]                ;Assign CH the attribute from the table.
        jmp     c12                     ;Return to top of loop.

c20:    cmp     ah, 4eh                 ;Test grey +.
        jne     c21
        xor     ch, mask blink          ;Toggle blink bit.
        jmp     c12                     ;Return to top of loop.

c21:    cmp     ah, 1                   ;Test escape.
        jne     c22
        jmp     c23                     ;Go to quit on escape.
```

```
c22:    jmp     c12                     ;Return to start of loop.
                                        ;End of loop.

                                        ;** Quit routine. **
c23:    push    cx                      ;Save selected attribute.
        call    popscr                  ;Restore screen.
        call    showcur                 ;Restore cursor.
        pop     ax                      ;Restore selected attribute into AX.
        ret

f2      endp

f3      proc    near                    ;** F3: read file. **

        call    pushscr                 ;Save old screen.
        call    savecur                 ;Save old cursor position.

        mov     ax, 79                  ;Draw box: 0,0 to 4,79.
        push    ax
        mov     ax, 4
        push    ax
        xor     ax, ax
        push    ax
        push    ax
        call    box
        add     sp, 8

d00:    cls     1,1,3,78                ;Macro: clear inside of box.

                                        ;Message 1.
        mov     ax, 3                   ;Column.
        push    ax
        mov     ax, 2                   ;Row.
        push    ax
        mov     ax, 0fh                 ;Attribute =
        push    ax                      ;white on black.
        mov     ax, offset f3_1
        push    ax
        call    printv
        add     sp, 8                   ;Restore stack.

                                        ;Message 2: reverse video blanks.
        mov     ax, 26                  ;Column.
        push    ax
        mov     ax, 2                   ;Row.
        push    ax
        mov     ax, 70h                 ;Attribute: black on white/reverse video.
        push    ax
        mov     ax, offset f3_2
        push    ax
        call    printv
        add     sp, 8                   ;Restore stack.

                                        ;Read file name.
        mov     ax, 50                  ;Buffer size.
        push    ax
        mov     ax, 26                  ;Column.
        push    ax
        mov     ax, 2                   ;Row.
        push    ax
        mov     ax, 70h                 ;Attribute: black on white/reverse video.
        push    ax
        mov     ax, offset filename     ;Buffer address.
        push    ax
        call    getstr
        add     sp, 10
        cmp     ah, 0                   ;Test if operation completed.
        je      d01
        call    popscr
        jmp     d04                     ;Operation aborted by user: go to quit.

d01:    mov     ah, 3dh                 ;Open file.
        mov     dx, offset filename     ;ASCIIZ file name.
        mov     al, 0                   ;Read only.
        int     21h
        jc      d02
        mov     handle, ax              ;Save file handle.
        jmp     d03
d02:                                    ;Begin error routine.
        cls     1,1,3,78                ;Macro: clear inside of box.

                                        ;Message 3.

        mov     ax, 3                   ;Column.
        push    ax
        mov     ax, 2                   ;Row.
```

```
        push    ax
        mov     ax, 8fh             ;Attribute =
        push    ax                  ;blinking white on black.
        mov     ax, offset f3_3
        push    ax
        call    printv
        add     sp, 8               ;Restore stack.

        mov     dh, 2               ;Pause until user enters key.
        mov     dl, 46
        cursor                      ;Macro.
        mov     ah, 0
        int     16h

        jmp     d00                 ;Go back and try again.

                                    ;Read file right into video memory!
d03:    call    popscr
        mov     ah, 3fh             ;DOS read file function.
        mov     bx, handle
        mov     cx, 4000            ;Number of bytes to read.
        push    ds                  ;Place ES into DS.
        push    es
        pop     ds
        mov     dx, 0
        int     21h
        pop     ds                  ;Restore DS.

        mov     ah, 3eh             ;Close file.
        mov     bx, handle
        int     21h

d04:    call    restcur             ;Restore the cursor.
        ret

f3      endp

f4      proc    near                ;** F4: write file. **

        call    pushscr             ;Save old screen.
        call    savecur             ;Save old cursor position.

        mov     ax, 79              ;Draw box:  0,0 to 4,79.
        push    ax
        mov     ax, 4
        push    ax
        xor     ax, ax
        push    ax
        push    ax
        call    box
        add     sp, 8

e00:    cls     1,1,3,78            ;Macro: clear inside of box.

                                    ;Message f4_1.
        mov     ax, 3               ;Column.
        push    ax
        mov     ax, 2               ;Row.
        push    ax
        mov     ax, 0fh             ;Attribute =
        push    ax                  ;white on black.
        mov     ax, offset f4_1
        push    ax
        call    printv
        add     sp, 8               ;Restore stack.

                                    ;Message f3_2: reverse video blanks.
        mov     ax, 26              ;Column.
        push    ax
        mov     ax, 2               ;Row.
        push    ax
        mov     ax, 70h             ;Attribute:black on white/reverse video.
        push    ax
        mov     ax, offset f3_2
        push    ax
        call    printv
        add     sp, 8               ;Restore stack.

                                    ;Read file name.
        mov     ax, 50              ;Buffer size.
        push    ax
        mov     ax, 26              ;Column.
        push    ax
        mov     ax, 2               ;Row.
```

```
        push    ax
        mov     ax, 70h             ;Attribute:black on white/reverse video.
        push    ax
        mov     ax, offset filename ;Buffer address.
        push    ax
        call    getstr
        add     sp, 10
        cmp     ah, 0               ;Test if operation completed.
        je      e01
        call    popscr
        jmp     e04                 ;Operation aborted by user: go to quit.

e01:    mov     ah, 3ch             ;"CREAT" file.
        mov     dx, offset filename ;ASCIIZ file name.
        mov     cx, 0               ;No special attributes.
        int     21h
        jc      e02
        mov     handle, ax          ;Save file handle.
        jmp     e03
e02:                                ;Begin error routine.
        cls     1,1,3,78            ;Macro: clear inside of box.
                                    ;Message f4_2.
        mov     ax, 3               ;Column.
        push    ax
        mov     ax, 2               ;Row.
        push    ax
        mov     ax, 8fh             ;Attribute =
        push    ax                  ;blinking white on black.
        mov     ax, offset f4_2
        push    ax
        call    printv
        add     sp, 8               ;Restore stack.

        mov     dh, 2               ;Pause until user enters key.
        mov     dl, 49
        cursor                      ;Macro.
        mov     ah, 0
        int     16h

        jmp     e00                 ;Go back and try again.

                                    ;Write file right from video memory!
e03:    call    popscr
        mov     ah, 40h             ;DOS write file function.
        mov     bx, handle
        mov     cx, 4000            ;Number of bytes to write.
        push    ds
        push    es
        pop     ds
        mov     dx, 0
        int     21h
        pop     ds

        mov     ah, 3eh             ;Close file.
        mov     bx, handle
        int     21h

e04:    call    restcur             ;Restore cursor.
        ret

f4      endp


f10     proc    near                ;** F10: quit. **

        call    pushscr             ;Save the screen.
        call    savecur             ;Save old cursor position.

        mov     ax, 49              ;Draw box:  4,10 to 8,49.
        push    ax
        mov     ax, 8
        push    ax
        mov     ax, 10
        push    ax
        mov     ax, 4
        push    ax
        call    box
        add     sp, 8

        cls     5,11,7,48           ;Clear inside of box.

                                    ;Display message 1.
        mov     ax, 16              ;Column.
        push    ax
        mov     ax, 6               ;Row.
```

```
            push    ax
            mov     ax, 8fh                 ;Attribute =
            push    ax                      ;intense blinking.
            mov     ax, offset f10_1
            push    ax
            call    printv
            add     sp, 8                   ;Restore stack.
            mov     dh, 6                   ;Position cursor.
            mov     dl, 43
            cursor                          ;Macro.

            mov     ah, 0                   ;Read user response.
            int     16h
            cmp     al, 'y'
            je      f01
            cmp     al, 'Y'
            je      f01
            call    restcur                 ;If "yes", restore cursor and screen
            call    popscr                  ;before returning.
            ret
                                            ;** Terminate program routine. **
f01:        xor     dx, dx                  ;Home cursor and clear screen.
            cursor                          ;Macro.
            cls     0,0,24,79

            mov     ah, 4ch                 ;Exit w/ errorlevel 0.
            mov     al, 0
            int     21h

f10         endp


getstr      proc    near                    ;** Reads a string from keyboard. **
comment     /*
            Reads characters until CR or until one less than the specified
            number of characters has been read.  Terminates the string with
            a null.  Returns AH = 0 if operation completed, AH = 1 if operation
            aborted by user entering <ESC>.

            Alters registers:  AX, BX, CX, DX, SI
            */

gframe      struc                           ;Stack frame template.
gbptr       dw      ?
gret_ad     dw      ?

gbuf_ad     dw      ?                       ;Buffer address.
gatt        dw      ?                       ;Display attribute.
grow        dw      ?                       ;Starting row.
gcol        dw      ?                       ;Starting column.
gnum        dw      ?                       ;Number of characters to read (+ 1 for
                                            ;null at end).
gframe      ends

            push    bp
            mov     bp, sp

            mov     bx, frame.gbuf_ad           ;BX is buffer pointer.
            mov     si, bx                      ;SI points to last buffer position.
            add     si, frame.gnum
            dec     si
            mov     ch, byte ptr frame.gatt     ;Initialize attribute.
            mov     dh, byte ptr frame.grow     ;Initialize row/column.
            mov     dl, byte ptr frame.gcol
            cursor                              ;Place cursor at first position.

g01:        mov     ah, 0                   ;Start keyboard read loop.
            int     16h

            cmp     ah, 1                   ;Test for escape => abort operation.
            jne     g02
            jmp     g08                     ;Go to quit routine.

g02:        cmp     al, 13                  ;Test for CR.
            jne     g03
            jmp     g07                     ;Go to end of loop on CR.

g03:        cmp     al, 8                   ;Test for backspace.
            jne     g04
            cmp     dl, byte ptr frame.gcol     ;Test if backspace possible.
```

```
        je      g01                     ;Return to top of loop if not possible.
        dec     dl                      ;Decrement column & move cursor back.
        cursor                          ;Macro.
        mov     cl, 32                  ;Write space.
        putchar                         ;Macro.
        dec     bx                      ;Decrement buffer pointer.
        jmp     g01                     ;Read another character.

g04:    cmp     bx, si                  ;Test to see if at end of buffer.
        jne     g05
        jmp     g01                     ;Return to top of loop if at end (can
                                        ;act on BS or CR only).

g05:    cmp     al, 0                   ;Test for non-ASCII key.
        jne     g06                     ;Don't move cursor on non-ASCII value.
        jmp     g01                     ;Return to top of loop.

g06:    mov     cl, al                  ;Write the entered character.
        putchar                         ;Macro.
        inc     dl                      ;Move to next column.
        cursor                          ;Macro.

        mov     [bx], al                ;Write the character to the buffer.
        inc     bx                      ;Increment buffer pointer.
        jmp     g01                     ;Return to top of loop.
                                        ;End of loop.

g07:    mov     byte ptr [bx], 0        ;Null termination.
        mov     ah, 0                   ;Indicates operation completed.

g08:    pop     bp                      ;Branch directly to this point if
        ret                             ;operation aborted.

getstr  endp

printv  proc    near                    ;** Displays a string on screen. **
comment /*
        This procedure writes a null terminated string directly to video
        memory, and should be used only with IBM-compatible systems with
        controllers that do not produce "snow."

        Alters registers:  AX, CX, DI, SI, DI
*/

hframe  struc                           ;Stack frame template.
hbptr   dw      ?
hret_ad dw      ?
hstr_ad dw      ?                       ;String address.
hatt    dw      ?                       ;Display attribute.
hrow    dw      ?                       ;Starting row.
hcol    dw      ?                       ;Starting column.
hframe  ends

        push    bp
        mov     bp, sp

        mov     si, frame.hstr_ad       ;DS:SI point to start of source string.

                                        ;Starting offset in video memory =
                                        ;       (row * 160) + (col * 2)
        mov     di, frame.hrow          ;Place row in DI and multiply by 160.
        mov     ax, di
        mov     cl, 7
        shl     di, cl
        mov     cl, 5
        shl     ax, cl
        add     di, ax

        mov     ax, frame.hcol          ;Multiply col by 2.
        shl     ax, 1
        add     di, ax                  ;Add (col * 2) to DI.

                                        ;Attribute in AH.
        mov     ah, byte ptr frame.hatt
        cld
h01:    cmp     byte ptr [si], 0        ;Test for null termination.
        je      h02
        lodsb                           ;Load a single character from source.
        stosw                           ;Write a char AND attribute to dest.
        jmp     h01                     ;Return for another character.
```

```
h02:     pop     bp
         ret

printv   endp


pushscr  proc    near                   ;** Push current video display. **
                                        ;Alters registers: CX, SI, DI.

         push    es                     ;Exchange ES and DS.
         push    ds
         pop     es
         pop     ds

         xor     si, si                 ;Initialize SI, DI, CX.
         mov     di, es:stkptr
         mov     cx, 4000               ;Number of bytes to transfer.
         cld
         rep     movsb                  ;Move entire screen.

         push    es                     ;Restore ES and DS.
         push    ds
         pop     es
         pop     ds

         add     stkptr, 4000           ;Increment screen stack pointer.

         ret

pushscr  endp


popscr   proc    near                   ;** Pop a pushed video display. **
                                        ;Alters registers: CX, SI, DI.

                                        ;Decrement stack pointer if not
                                        ;already at bottom.
         cmp     stkptr, offset scrstack
         jbe     j01
         sub     stkptr, 4000           ;Decrement screen stack pointer.
j01:     mov     si, stkptr             ;Initialize SI, DI, CX.
         xor     di, di
         mov     cx, 4000               ;Number of bytes to transfer.
         cld
         rep     movsb                  ;Move entire screen.

         ret

popscr   endp

box      proc    near                   ;** Display box on screen. **
                                        ;Alters registers: AX, BX, CX, DX.

kframe   struc                          ;Stack frame template.
kbptr    dw      ?
kret_ad  dw      ?
kulr     dw      ?                      ;Upper left row.
kulc     dw      ?                      ;Upper left column.
klrr     dw      ?                      ;Lower right row.
klrc     dw      ?                      ;Lower right column.
kframe   ends

         push    bp
         mov     bp, sp

         mov     ah, byte ptr frame.kulr   ;Store upper/lower rows in AH/AL.
         mov     al, byte ptr frame.klrr

         mov     ch, INITATTR           ;Use default attribute.
         mov     cl, 201                ;Print upper left corner.
         mov     dh, ah
         mov     dl, byte ptr frame.kulc
         putchar
                                        ;Macro.

         mov     cl, 200                ;Print lower left corner.
         mov     dh, al
         putchar                        ;Macro.
```

```
                                        ;Calculate number of horizontal
                                        ;line characters; store in BL.
        mov     bl, byte ptr frame.klrc
        sub     bl, dl                  ;Subtract ulc.
        dec     bl

        mov     cl, 205                 ;Print top and bottom lines.
k01:    inc     dl                      ;Loop for printing all horizontal lines.
        mov     dh, ah                  ;An upper line.
        putchar                         ;Macro.
        mov     dh, al                  ;A lower line.
        putchar                         ;Macro.
        dec     bl
        jz      k02
        jmp     k01

k02:    mov     cl, 188                 ;Print lower right corner.
        inc     dl
        putchar                         ;Macro.

        mov     cl, 187                 ;Print upper right corner.
        mov     dh, ah
        putchar                         ;Macro.

        mov     bl, al                  ;Calculate number of vertical line
        sub     bl, ah                  ;characters; store in BL.
        dec     bl
        mov     ah, byte ptr frame.kulc     ;Store left/right cols in AH/AL.
        mov     al, dl

        mov     cl, 186                 ;Print vertical lines.
k03:    inc     dh                      ;Loop for printing all vertical lines.
        mov     dl, ah                  ;A left line.
        putchar                         ;Macro.
        mov     dl, al                  ;A right line.
        putchar                         ;Macro.
        dec     bl
        jz      k04
        jmp     k03

k04:    pop     bp
        ret

box     endp


hidecur proc    near                    ;** Makes the cursor disappear. **
                                        ;Alters registers: AX, BX, CX, DX.

        mov     ah, 3                   ;BIOS read cursor function.
        mov     bh, 0
        int     10h                     ;BIOS video services.
        mov     oldcur, cx              ;Save the old cursor style.

        mov     ah, 1
        mov     ch, 32                  ;1 in bit 5 makes the cursor disappear.
        mov     cl, 0
        int     10h
        ret

hidecur endp


showcur proc    near                    ;** Restores the cursor. **
                                        ;Alters registers: AX, CX.

        mov     ah, 1                   ;BIOS set cursor type function.
        mov     cx, oldcur
        int     10h                     ;BIOS video services.

        ret

showcur endp


savecur proc    near                    ;** Saves current cursor position. **
                                        ;Alters registers: AX, BX, CX, DX.
```

```
        mov     ah, 3               ;BIOS read cursor position function.
        mov     bh, 0

        int     10h                 ;BIOS video services.
        mov     curpos, dx

        ret

savecur endp

restcur proc    near                ;** Restores saved cursor position. **
                                    ;Alters registers: AH, BX, DX.

        mov     ah, 2               ;BIOS set cursor position function.
        mov     bh, 0
        mov     dx, curpos
        int     10h                 ;BIOS video services.

        ret

restcur endp

codeseg ends

        end     main
```

上面是汇编语言编制的交互式屏幕设计程序。

▲程序的作用

该程序首先打开一个空屏幕。用户可直接向屏幕打入任意序列的 ASCII 字符。该程序的多数基本操作都是通过功能键实现的。

F1 功能键——显示出一个窗口,其中列出了扩展 IBM 集中所有 253 可显示字符。使用箭头可选择某个字符。所需字符选中后,按 Esc 退出该窗口,恢复原始屏幕。下面只要按下四个箭头键之一,便可在相应方向上重复出现选中的字符,此功能可用于画线、框以及其他非键盘字符。但当按下 shift 键或 NumLock 为 ON 时,箭头键只移动光标,而不重复选中的字符,此为所谓的"抬笔"状态。

F2功能键——如果用彩色显示器,则此功能用于颜色选择;如果用单显系统,则用于选择单色属性。使用箭头键选择所需的属性,并以 ESC 退出屏幕。键+为闪烁属性的跟头式开关。选择完毕后,后面输入的字符均以选中的属性显示。

F3 功能键—读人并显示一个屏幕数据文件,从而可对以

前保存的文件进行编辑。注意,此时当前的屏幕数据将丢失。按 ESC 键退出此操作。

F4 功能键——将当前屏幕内容写入一个4000字节的数据文件(2000 字符和 2000 个属性为视频内存区的准确转储)。按 ESC 键退出此操作。

F10 功能键—— 终止此程序。

一旦包含整屏数据的文件生成并存入磁盘,就可有两种方式来使用它。第一种方式,该文件置于应用程序的同一张磁盘上,应用程序可将数据直接读到一个内部缓冲区中,并使用 printw 这样的函数来显示它。此方法减小了程序大小,但降低了程序的执行速度。另一种方法是使用原始的屏幕数据产生一个源代码。此段源代码在编译时用于初始化一个程序中的数组。如果屏幕数不多,此方法较好,因为它提供了尽可能快的显示时间,并且无需在工作磁盘上安放补充的数据文件。

如何才能将原始的屏幕数据翻译成适当的源代码?下面的清单为一个简单的 C 实用程序,它将一个 4000 字节的屏幕数据文件翻译成C或汇编源代码,然后用于数组的初始化。此程序并不产生变量名或声明,仅是用逗号分开的十进制表示4000 ASCII 值的一个清单(对于汇编程序,每行首部为一个 db)。故在 C 程序中,屏幕数组 scr 可声明如下:

```
char scr[4000] =
{
#include "SCT.DAT"   /* 包含 4000 个十进制数 */
};
```

如前所述,将此数据定义放在与主应用程序分开的另一个文件中以避免减慢编译时间。如果使用了编译开关/W2,

在编译此文件时应注意去除此开关,否则当每个整数转换成字符时都会给出警告信息(共 4000 个警告信息)。

```
        File:     CODER.C

        A C utility that converts a screen data file into C or assembler source

        This program reads a 4000 byte file which is an image of video memory,
        such as generated by SM.ASM (Figure B.9), and converts it to the source
        code required to initialize a C or Assembler array.
*/

#include "stdio.h"
#include "conio.h"

FILE *infile,*outfile;

void main ()
        {
        char infilename[128];
        char outfilename[128];
        int  ok = 1;
        int  ch;
        int  col;
        int  asm;
                                                /* Select C or Assembler format. */
        printf ("language for include file:\n");
        printf ("  (1) C\n");
        printf ("  (2) assembler\n");
        printf ("enter 1 or 2: ");
        do

       {
       asm = getch () - 49;
       }
while (asm < 0 || asm > 1);

do                                              /* Input file: get name and open. */
       {
       ok = 1;
       printf ("\n\nenter name of screen data file: ");
       scanf ("%s",infilename);                         /* Read file name. */
       if ((infile = fopen (infilename,"rb")) == NULL)
              {
              printf ("can't open %s\n",infilename);
              ok = 0;
              }
       }
while (!ok);

do                                              /* Output file: get name and open. */
       {
       ok = 1;
       printf ("enter name of include file to create: ");
       scanf ("%s",outfilename);                        /* Read file name. */
       if ((outfile = fopen (outfilename,"w")) == NULL)
              {
              printf ("can't open %s\n",outfilename);
              ok = 0;
              }
       }
while (!ok);

if (asm)
       fprintf (outfile,"db ");

col = asm?3:0;                                  /* Initialize starting column. */

while ((ch = fgetc (infile)) != EOF)            /* Read and translate file.    */
       {
       if ((col > 3) | (col > 0 && !asm))
              if (col <= 75)
                     fprintf (outfile,",");
```

```
            else
                 {
                 fprintf (outfile,asm?"\ndb ":",\n");
                 col = asm?3:0;
                 }
        fprintf (outfile,"%1d",ch);
        if (ch < 10)
            col += 2;
        else if (ch < 100)
            col += 3;
        else
            col += 4;
        } /* end while */

    fclose (infile);
    fclose (outfile);

    } /* end main */
```

上面是将屏幕数据文件转换成C或汇编源程序的C实用程序。

▲程序的工作原理

程序采用标准的.EXE格式,过程 main 设置ES寄存器以指向正确的视频内存区段地址,并由设置一个全局标志以标明当前活动的显示是彩色的还是单色的。将重复使用的值存储在寄存器中可加强性能.下列寄存器用于在程序中存储重要数值(某些子程序只是临时为其他目的所使用,其中的数值便不宜存储在寄存器中).

ES 视频内存区段地址

CH 当前选择的显示属性

CL 当前选择的字符

DH 当前行

DL 当前列

在以速度为主要的代码区域(如用于从键盘读并对视频内存区写字符的主回路),使用宏(macros)而不是子程序可加强性能。

过程 main 由执行下面一系列步骤的回路组成:

(1) 使用 BIOS 中断 16h 读键盘.

(2) 如果按下 ESC 键则立即返回(此键用于退出子程序,应被主程序忽略).

(3) 测试功能键 F1,F2,F3,F4 及 F10. 如果按下其中一个,便调用相应的子程序.

(4) 测试箭头键. 如果按下其中一个,便移动光标,且如果 shift 状态不工作则重复选中的字符.

(5) 测试退格键. 如果测到,便后移光标并抹去左边的字符(除非当前列为 0) 。

(6) 如果输入了非上述的任何一个控制键,便返回到回路的开头,即象 F6 这样的键不起作用,也不使光标移动.

(7) 如果检测到上述键之一,程序执行所要求的服务,然后返回到回路的开头. 因此,如果到达了回路的底部,字符必须具有正常的 ASCII 值,并与选中的显示属性一起直接写到视频内存区,当前光标位置被更新.

通过功能键选中的五个子程序之任一个都将在当前屏幕的顶部显示出一个窗口. 为保存当前显示,这些功能均调用过程 pushscr 及 popscr. pushscr 和 popscr 使用标准的堆栈数据结构去存储屏幕,工作方式类似于机器指令 push 和 pop,故可以"后进先出"的次序存储和恢复一连串的屏幕(堆栈指针 stkptr 为每次调用增量 4000). 并不使用 db 指令为屏幕堆栈保留一个固定大小的内存(因为预先可能无法知道准确大小,并且保留内存增加了磁盘上.EXE 文件的大小),本程序

使用了一点小技巧:

说明段时使数据段处在内存中最高处,堆栈(scrstack)作为数据段底部的一个标号说明。因此堆栈位于程序结尾的前面,处于内存在过程中装入的自动分配给该过程的区域中(默认情况下,.EXE 文件头将请求所有用户内存;但如果除应用程序外无足够内存可用,则并不发出警告信息。更安全的方法是预先测定实际可用的自由内存大小,或使用扩展内存来实现此目的)。

▲可进一步加强的功能

虽然此屏幕设计程序的源文件已达 50K,该程序还只是一个起码的子集,但已有了一个很好的框架,能借以实现更多的功能。例如:

(1) 使用窗口子程序

在此程序中屏幕是一个字符一个字符生成的,而不是简单、快速地使用 prontw 从经初始化数据变来的。不使用 printw,是为了避免人工输入成千的数字来初始化数组,一旦此程序运行,它可用来为自己有效地生成屏幕。

(2) 联机帮助功能

可生成某种联机帮助屏幕用于显示有关程序运行、使用方法的信息。

(3) 光标状态

该程序应可选地显示光标的当前行、列位置。一旦屏幕设计完毕,便可使用 shift-prtsc 将其打印出来,记录下屏幕上重要点的行、列坐标以便以后引用。

(4) 防止雪花效应

如果检测到 CGA 卡,则向视频内存区写时应采用无雪花效应的版本。

(5) 显示方式

在彩色系统上,应保存并恢复当前的显示方式,并将显示方式切换成 80×25 彩色文本方式(方式 3). 如果程序开始时处于图形方式或 40 列显示方式,当前的程序版本不能正确工作.

(6) 数据丢失报警

当前没有向用户发数据丢失的报警. 屏幕被修改时应设置一个标志,并当屏幕被存贮时重置此标志. 在读人新屏幕或程序结束时,应向用户报警"所作修改将丢失!"。

(7) 抹除数据

应有更方便的从屏幕上抹除数据的方法,例如,在可选字符中加入空格字符,或采取其他方法使光标在任意方向移动时能抹除字符.

(8) 画框方式

可提供更方便的画框字符. 例如可以选择框的类型,然后由光标来画线,并在适当的点上自动产生框角等.

(9) 附加的编辑功能

可增加更多的编辑功能,例如,块的移动和拷贝,用某个字符和属性填充指定区域,自动对中等.

(10)直接代码生成

可直接从程序产生 C 或汇编语言代码,而不用运行单独的实用程序.

6.6 在 C 程序中使用视频显示子程序

本节介绍 C 程序员如何利用本书介绍的各种视频函数和实用程序来设计漂亮的显示屏幕,快速显示这些屏幕,以及收集用户的输入. 编写一个 C 程序时,可借助本书中的函数来实现下面的基本视频显示操作:

·交互式地设计屏幕显示,并在C数据段内存储结果数据。

·使用一个数据堆栈来存储和恢复当前显示,使窗口可在当前屏幕顶部显示出来,覆盖其他窗口,并从屏幕上移去。

·在物理屏幕任意位置处显示屏幕数据数组的任意部分。

·使用任意显示属性在屏幕上任意位置写字符串,故可使动态信息与屏幕设计不变部分相结合。

·从屏幕的任意位置处读数据,以任意显示属性回送字符,并将回送输出约束在符合屏幕设计的指定区域内。

下面程序清单给出了这一函数集并说明了它们在C程序中的使用。假定屏幕是利用前面的屏幕设计程序设计的,并且数据已被变换成C源代码用于初始化外部数组scr。此数组在另一独立文件中定义并与应用程序相链接。该程序还需与包含所有调用的外部函数的目标模块相链接(从清单中的说明可见这些函数的位置)。

```
        File:     WINTEST.C

        A C program demonstrating the use of window management functions

        This program uses a set of routines to manage the display of windows.
        The window data is contained in an initialized array, as produced by
        a screen designing utility (such as SM.ASM in Figure 8.9).  Note that
        the program must be linked with:
             -   The C file containing the definition and initial data for the
                 array 'scr'.
             -   Assembler and C modules containing the external functions called
                 (the sources are given in the declarations below).

        For example,

             MSC WINTEST;                   This file.
             MSC SCREEN;                    Contains screen data ('scr') and 'cls ()'.
             MASM WINPROC;                  Contains assembler functions listed below.
             LINK WINTEST+SCREEN+WINPROC;
*/

#include <stdio.h>
#include <conio.h>

extern char scr[4000];  /* Initialized array containing screen data;        */
                        /* located in a separate file to save compile time. */

                                                      /* Assembler functions.*/
void printv (char *,int,int,int);                     /* Figure 8.4          */
```

```
void printw (char *,int,int,int,int,int,int);          /* Figure 8.5          */
void pushscr (void);                                    /* Figure 8.12         */
void popscr (void);                                     /* Figure 8.12         */
void getstr (char *,int,int,int,int);                   /* Figure 8.12         */

                                                        /* External C function.*/
void cls (int,int,int,int);                             /* Figure 6.6          */

void cursor (int,int);                                  /* Internal function.  */

void main ()
     {
     char buf [15];                                /* Buffer to read in string.  */

     cls (0,0,24,79);                              /* Start with a clear screen. */
     cursor (0,0);                                 /* Home the cursor.           */

     printf ("this is normal teletype display\n");
     printf ("press any key for window demo ... ");
     getch ();                                     /* Create a pause.            */

     pushscr ();                                   /* Save program screen.       */
     printw (scr,0,0,11,39,0,0);                   /* Display window 1.          */
                                  /* Substitute 'printwcga' for CGA adapter.     */

     printv ("please enter your name:",0x0f,5,7);
                                  /* Display dynamic message; substitute         */
                                  /* printvcga for CGA adapter.                  */

     printv ("                ",0x70,6,7);
                                  /* Display reverse video area for input.       */

     getstr (buf,0x70,6,7,15);                     /* Input string in reverse video. */

     pushscr ();                                   /* Save screen with window 1. */
     printw (scr,0,40,11,79,4,12);                 /* Display window 2.          */
     printv (buf,0x70,8,18);                       /* Display dynamic messages.  */
     printv ("press any key for next window ...",0x0f,10,16);
     cursor (10,48);                               /* Pause for input.           */
     getch ();

     pushscr ();                                   /* Save screen with window 2. */
     printw (scr,12,0,23,39,9,25);                 /* Display window 3.          */
     printv ("press any key to remove window ...",0x0f,13,27);
     cursor (13,62);
     getch ();

                                         /* Remove window 3, expose window 2. */
     popscr();
     printv ("press any key to remove window ...",0x0f,10,16);
     cursor (10,49);
     getch ();

                                         /* Remove window 2, expose window 1. */
     popscr ();
     printv ("press any key to remove final window",0x0f,5,2);
     printv ("...                          ",0x0f,6,2);
     cursor (6,6);
     getch ();

     popscr ();                    /* remove window 1, expose original screen */
     cursor (10,0);
     printf ("back to teletype output ...\n");
     printf ("demonstration complete\n");

     } /* end main */


#include <dos.h>

void cursor (row, col)          /* Positions cursor at specified row/column. */
int row, col;
     {
     union REGS reg;

     reg.h.ah = 2;                           /* BIOS set cursor position function. */
     reg.h.dh = row;
     reg.h.dl = col;
     reg.h.bh = 0;                                              /* Page 0. */
     int86 (0x10,&reg,&reg);                  /* BIOS video services interrupt. */
     } /* end cursor */
```

上面是从C程序中使用管理窗口函数的示范程序。

此演示程序首先清屏并定位光标于初始位置．接着使用printf生成某种标准的屏幕输出．然后将当前屏幕推入堆栈并在顶部显示出窗口1．再后显示出两个相互重叠的窗口等待用户输入．最后将从前屏幕弹出堆栈一次一个地去除窗口，直到恢复原始显示．

该程序还说明了在窗口内使用printv打印动态信息,使用getstr从窗口读数据．

下面是由C演示程序调用的三个汇编语言函数．这些函数类似于屏幕设计程序中出现的同名函数,修改后可由一个C程序来调用．

欲使这些过程可被C调用,需为屏幕堆栈scrstack分配一定量的内存．本例保留了12,000个字节,足以在任何时间存储三屏(亦可由C的动态内存分配函数malloc分配可变量的内存)．还须在每次调用该函数时重新初始化ES寄存器，以指向正确的视频内存区段地址．

```
page 50,130

;Figure:    8.17
;File:      WINDOW.ASM

;This file contains functions for saving and restoring screens on a stack,
;and a function for inputting data from the screen.  Note that these are
;versions of procedures contained in the screen designer (SR.ASM, Figure 8.9),
;modified to be callable from a C program.

public      stkptr, scrstack
public      _pushscr
public      _popscr
public      _getstr

_data       segment word public 'DATA'

stkptr      dw      offset dgroup:scrstack   ;Points to beginning of next
                                             ;free block of 4,000 bytes on stack.

scrstack    db      12000 dup (?)            ;Reserve data area for 3 screens.

_data       ends

dgroup      group   _data

_text       segment byte public 'CODE'

assume      cs:_text, ds:dgroup
```

```
include     window.mac                      ;Contains the macros:
                                            ;       'cursor'
                                            ;       'putchar'
                                            ;These macros are listed in Figure 8.9.
frame       equ     [bp]

_pushscr    proc    near                    ;** Push current video display. **    a
                                            ;       void pushscr ();
assume      cs:_text, ds:dgroup

            push    si
            push    di
            push    es
            push    ds

            xor     ax, ax                  ;Set AX to current video memory segment.
            mov     es, ax
            cmp     byte ptr es:[449h], 7      ;Test video mode.
            je      a01
            mov     ax, 0b800h              ;Color buffer.
            jmp     a02
a01:        mov     ax, 0b000h              ;Monochrome buffer.

a02:        push    ds
            pop     es                      ;ES now points to C data segment.

            xor     si, si                  ;Initialize SI, DI, CX.
            mov     di, stkptr
            mov     cx, 4000                ;Number of bytes to transfer.
            cld
            mov     ds, ax                  ;Move AX into DS (segment of video mem).
            rep     movsb                   ;Move entire screen.

            pop     ds
            pop     es
            pop     di
            pop     si

            add     stkptr, 4000            ;Increment screen stack pointer.

            ret

_pushscr    endp

_popscr     proc    near                    ;** Pop a pushed video display. **    b
                                            ;       void popscr ();
assume      cs:_text, ds:dgroup

            push    si
            push    di
            push    es

            xor     ax, ax                  ;Set ES to current video memory segment.
            mov     es, ax
            cmp     byte ptr es:[449h], 7
            je      b01
            mov     ax, 0b800h
            jmp     b02
b01:        mov     ax, 0b000h
b02:        mov     es, ax
                                            ;Decrement screen stack pointer if not
                                            ;already at bottom.
            cmp     stkptr, offset dgroup:scrstack
            jbe     b03
            sub     stkptr, 4000            ;Decrement screen stack pointer.
b03:        mov     si, stkptr              ;Initialize SI, DI, CX.
            xor     di, di
            mov     cx, 4000                ;Number of bytes to transfer.
            cld
            rep     movsb                   ;Move entire screen.

            pop     es
            pop     di
            pop     si

            ret

_popscr     endp
```

```
_getstr     proc    near                    ;** Reads a string from keyboard. ** c
Comment     /*
            Reads characters until <CR>, or until number read is one less than
            the specified buffer size.  Terminates the string with a null.
            Returns AX = 0 if operation completed, AX = 1 if operation aborted
            by user entering <ESC>.  Terminating <CR> is not placed in the
            buffer.

            void getstr (buf, att, row, col, num);
            char *buf;
            int att;
            int row;
            int col;
            int num;
            */

assume      cs:_text, ds:dgroup

cframe      struc                           ;Stack frame template.
cbptr       dw      ?
cret_ad     dw      ?
cbuf_ad     dw      ?                       ;Buffer address.
catt        dw      ?                       ;Display attribute.
crow        dw      ?                       ;Starting row.
ccol        dw      ?                       ;Starting column.
cnum        dw      ?                       ;Number of bytes in receiving buffer.

cframe      ends

            push    bp
            mov     bp, sp

            push    si
            push    di
            push    es

            xor     ax, ax                  ;Set ES to current video memory segment.
            mov     es, ax
            cmp     byte ptr es:[449h], 7
            je      c01
            mov     ax, 0b800h
            jmp     c02
c01:        mov     ax, 0b000h
c02:        mov     es, ax

            mov     bx, frame.cbuf_ad           ;BX is buffer pointer.
            mov     si, bx                      ;SI points to last buffer position.
            add     si, frame.cnum
            dec     si
            mov     ch, byte ptr frame.catt     ;Initialize attribute.
            mov     dh, byte ptr frame.crow     ;Initialize row/column.
            mov     dl, byte ptr frame.ccol
            cursor                              ;Place cursor at first position.

c03:        mov     ah, 0                   ;Start keyboard read loop.
            int     16h

            cmp     ah, 1                   ;Test for escape => abort operation.
            jne     c04
            mov     ax, 1                   ;1 => abort.
            jmp     c10                     ;Go to quit routine.

c04:        cmp     al, 13                  ;Test for CR.
            jne     c05
            jmp     c09                     ;Go to end of loop on CR.

c05:        cmp     al, 8                   ;Test for backspace.
            jne     c06
            cmp     dl, byte ptr frame.ccol   ;Test if backspace possible.
            je      c03                     ;Return to top of loop if not possible.
            dec     dl                      ;Decrement column & move cursor back.
            cursor                          ;Macro.
            mov     cl, 32                  ;Write space.
            putchar                         ;Macro.
            dec     bx                      ;Decrement buffer pointer.
            jmp     c03                     ;Read another character.
```

```
c06:        cmp     bx, si               ;Test to see if at end of buffer.
            jne     c07
            jmp     c03                  ;Return to top of loop if at end (can
                                         ;act on BS or CR only).

c07:        cmp     al, 0                ;Test for non-ASCII key.
            jne     c08                  ;don't move cursor on non-ASCII value.
            jmp     c03                  ;Return to top of loop.

c08:        mov     cl, al               ;Write the entered character.
            putchar                      ;Macro.
            inc     dl                   ;Move to next column.
            cursor                       ;Macro.
            mov     [bx], al             ;Write the character to the buffer.
            inc     bx                   ;Increment buffer pointer.
            jmp     c03                  ;Return to top of loop.
                                         ;End of loop.

c09:        mov     ax, 0                ;Indicates operation completed.

c10:        mov     byte ptr [bx],0      ;Null termination -- branch directly
            pop     es                   ;to this point if operation aborted.
            pop     di
            pop     si
            pop     bp
            ret

_getstr     endp

_text       ends
            end
```

上面是从C程序中管理窗口所用的汇编函数。

在正常的堆栈机构下,当最后一项被移出后继续 pop 会引起出错. popscr 有所不同,它可连续恢复推入的第一个窗口,甚至在堆栈已空后仍可执行. 此功能有时很有用,例如可用重复恢复一个基础屏幕使窗口在屏幕上移动.

Getstr 是一个有用的通用函数,可在窗口环境下从用户读入数据. 与 C 库函数 scanf 和 gets 相比, 它有几个优点. 首先,它可用光标在屏幕上指定地点定值,并以与屏幕设计时相符合的指定属性回送字符. 其次,它只允许固定数量的字符被输入,并将回送输出限制在指定区域中,故用户不至于搞乱屏幕(使用 scanf 及其他 DOS 输入字符串的函数时很易搞乱屏幕). 此函数可用于输入各种类型的数据,因为使用 C 库函数 atoi 及 atof 等不难将字符中数据转换成整数或浮点数.

函数 getstr 读入字符直至下述三个条件之一发生:

• 按下回车.

· 接收缓冲中已进入了最大字符数减 1。一旦达到此阶段,该函数并不自动退出,而是等待回车或是退格键输入.

· 用户输入了 Esc. 此情况下函数立即终止并向调用程序返回 1,表明输入操作已失败(返回 0 值表明该操作完全正常).

终止回车不放入缓冲区,该缓冲区总是以一个空字符结束. 当前由 egtstr 提供的唯一编辑功能是破坏性退格,它虽然简单,但提供了下面的附加功能:

· 使用箭头键移动光标.

· 插入和覆盖方式.

· Home 与 End 移动光标至行首或行末.

此函数经修改后可提供另外一些有用的选择,可由传送给特殊参数来激发。例如:

· 只允许数字性字符.

· 自动变换成大写.

· 指定小数点、日期分隔符等特殊格式.

· 输入了最大数目的字符后自动退出等.

第七章　内存驻留程序设计

MS-DOS(直到版本 3.3)的主要限制是缺乏对多任务的支持．程序结束并驻留实用程序(TSR)是一种很巧妙的机构，至少可以部分地解决受限制的问题．TSR 是这样一类程序，它把自己装入内存后将控制返回给 DOS,然后"潜伏"在后台．用户按下指定的组合键(称为"热键"hotkey),TSR 被激活,立即中断当前运行的应用程序,并允许立即访问它所提供的服务．使用 TSR 取代从 DOS 命令行上装入的传统程序，有几大优点:

- 速度：旧式实用程序在使用时要先退出当前应用程序，打入一条 DOS 命令,然后重新启动原来的应用程序．但采用 TSR 便可实现快速访问．如果前台运行的是一个大且复杂的应用程序,提高速度便至关重要．

- 数据交换：许多 TSR,尤其是日记编辑程序、宏处理程序等允许在实用程序与前台应用程序之间输入、输出数据．因此它为不相似程序、不同格式文件之间交换文本及图形数据提供了方便的通道．

- 模块化：装入一组经选择的驻留实用程序,而不是运行一个集成软件包,使得用户有可能设计一个优化的计算环境．例如可在前台运行一个字处理程序,在后台配合以内存驻留的词典软件、拼音检查程序、日记程序等．

Microsoft　Windows 等多任务环境提供了 TSR 机构的一种替代方案．但由于这种系统存在着不兼容的问题,并且

在普通 IBM PC/XT 上运行速度很慢,故从目前看,还是内存驻留程序能提供更多的灵活性及更高的性能.

MS-DOS 机器有一些特点有利于研制 TSR 程序,但也存在着一些问题影响 TSR. 一种有用的资源是 DOS 有一个中断服务程序(功能号 31h)允许一个程序结束后驻留在内存中. 另一个重要特点是软、硬件中断通过一个公用的向量表可建立联系,故用替换中断子程序、驻留程序可监测硬件事件和软件服务请求.

研制开发 TSR 时,MS-DOS 的主要问题是"臭名昭著"的所谓 DOS 代码是"不可重入(nonreentrant)"的,意即操作系统核心不可在任意点处中断后使其代码为另一个进程重新使用. MS-DOS 操作系统设计时便没有考虑多个进程的处理问题.

本章第一节讨论在 MS-DOS 下实现内存驻留程序时会遇到的问题,以及一些有助于这些问题解决的准则. 第二部分讲述一个实现 TSR 的实际方法,给出一组汇编语言过程可使 C 程序在内存驻留使用. 这些过程详细说明了如何处理内存驻留问题,尤其是允许内存驻留代码自由地调用 MS-DOS 功能. 这些汇编模块与 C 程序连接后,一个简单的函数调用可将传统的 C 程序转换为 TSR.

7.1 编写 TSR 时需注意的问题

编写内存驻留程序时遇到的主要问题是要能够与其他 TSR 程序、与操作系统、与中断了的前台应用程序、与 BIOS 磁盘活动、以及与中断处理程序"和平共处". 本节讲述这些问题和重进入问题,并给出编写 TSR 的建议标准.

7.1.1 与其他 TSR 共存

内存驻留机构的一个主要优点是若干独立的 TSR 可组

合在一起使用。但是由于此时的环境有些对立,这些程序常常表现出"敌对"行为,故设计一个新的 TSR 的主要问题之一便是存在着运行时间与其他内存驻留实用程序冲突的可能性.

冲突的发生并不奇怪,因为内存驻留程序必须共享硬、软件资源且相互之间有干扰,更因为目前尚无标准协议或一组正规的准则用于它们的开发. 这里介绍的准则将增加应用 STR 成功运行的机会,也将使其他 TSR 的运行更容易,但并不能保证不遵守同样一组规则的其他程序不产生困难. 冲突可能发生的范畴包括:hotkcy 重复定义、可利用的内存的竞争、TSR 不能正确地与当前已安装了的其他内存驻留程序链接。

(1) 避免 hotkey 冲突

另一个 TSR 程序可能因为与已选的 TSR 的键相同而激活;还有对所选择的 hotkey 亦可能是某个前台应用程序所用的控制键,此时激活的是哪个 TSR 通常取决于两个程序的装入次序. 为避免此困难,选择一个很少用的默认组合键,并使用户有权将 hotkey 改变为另一个.

(2)节省内存

随着应用程序的大小与复杂性的增加,使用多个 TSR 越来越常见,因此内存资源显得越来越缺乏. 前台应用程序(又叫"临时程序transicnt program")可使用限制内所有可用内存,但由于 TSR 永久性地获取了它所占用的内存,故应使用尽可能少的内存。下列方法有助于减少对可用内存的争夺:

▲只要有可能,应允许用户在装入 TSR 时选择所占用的内存量.

例如,对于"日记"程序,用户可以选择分配给数据的缓冲

区太小(从而设置了可接纳的最大文件大小)。又如,如果一个TSR包含一个以上的功能模块(如Sidekick等办公软件包括"日记"程序、日历、计算器、电话号码薄等功能),应让用户可选择将任意的模块组合装入内存。

▲使用磁盘文件作为代码覆盖模块或作为交换数据的临时保存区以减少内存需求。

但应注意,访问磁盘会大大降低性能,尤其是使用软磁盘驱动器的用户更不方便。故此方法用在带硬盘的系统上。

▲如果内存约束得很厉害,应考虑全部使用汇编语言来编写代码。

汇编语言不仅可产生更紧凑的代码,而且允许初始化程序所占用的内存在执行完其功能后能够释放掉。借助于下节介绍的汇编模块,用C编写TSR,可使程序的开发简单化。但是C编译程序通常产生较之手写汇编程序更大的代码,尤其在调用了C的库程序时。而且C语言编写的TSR不能释放其初始化代码(理由见后)。

▲只要存在扩展内存便使用它。

扩展内存(Expanded memory)非常适合于TSR,TSR程序可用它来存贮代码和数据,从而将传统的RAM区留给前台应用程序及其他的TSE程序。下面一系列步骤说明了一种内存驻留程序能够最优使用扩展内存的方法。

①用户在DOS提示符下装入TSR。初始化代码则为代码和数据分配所有需要的扩展内存。

②初始化代码在扩展内存页内存贮几乎所有程序代码和数据。但实际的中断处理程序必须留在传统内存区中,因为中断向量无法指向扩展内存页面内的地址。

③当一个TSR中断处理程序被激活时,它保存了当前的

页映照关系,并且自己的页映照到页面(page frame)内。该程序有两个选择.第一个选择是简单地将控制传送给页面内的代码,并且完全在扩展内存中执行(但堆栈仍应留在传统 RAM 中)。由于页面仅为 64K 字节,故如果需要与其他页的代码及数据进行交换,该程序须仔细地设计成覆盖模块式系统。第二种方法是,存贮在扩展内存中的所有代码和数据可被交换进入传统内存,然后正常执行该程序,无须使用覆盖模块。此时最好使用传统 RAM 的高端,因为此区域不大可能包含其他 TSR(如果你的代码临时取代了内存中的 TSR,且此时那个 TSR 的 hotky 被按下,则会得到令人莫明其妙的效果)。

④该 TSR 结束后,恢复页映照关系。

⑤如果卸掉该 TSR,则应释放所有分配的扩展内存。

▲与其他 TSR 链接

如果控制的导向不正确,则一个有错的 TSR 将使其后面的内存驻留程序全部失效。在理想情况下,如果有多个 TSR 被装入内存,应将它们链接成一条链子,控制以一个程序传送给另一个程序,不超过任何模块。例如,考虑由一系列 TSR 取代现存的中断 16h 向量(BIOS)键盘服务功能),此中断几乎所有的程序都要调用用于从键盘读入一个键,并常常为 TSR"截获"以检测何时按下了 hotkey 或处理来自键盘的数据。装入的第一个 TSR 替换了指向原始 BIOS 代码的向量,使该向量指向它自己。由于它保存了原始向量,当被激活时,首先调用原始的子程序,在原始子程序返回后执行它自己的任务。后面装入的程序以类似方式将自己插入到此链中,得到的结构如下图所示。

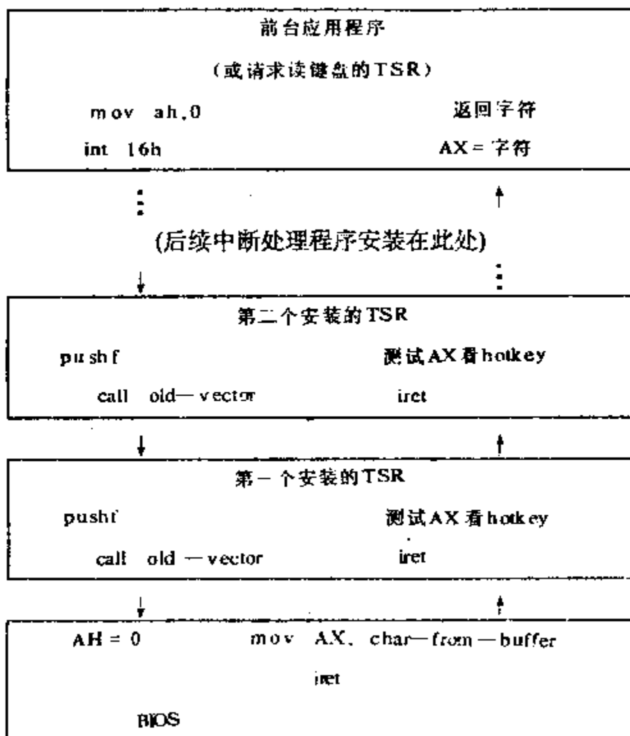


图7-1 关于中断16h的TSR链

本例中,一个临时应用程序(或一个 TSR)通过 BIOS 中断 16h 请求键盘读。该请求通过链往下传送,逐步到达 BIOS 代码。返回的字符通过 TSR 链传送回来,每个 TSR 均要测试其 hotkey 或执行某些其他处理(例如可用键盘宏实用程序进行字符翻译)。如果所有 TSR 都遵守此图的调用协议,则返回字符以程序安装的次序从一个程序传送给另一个程序。如果两个 TSR 发生了 hotkey 冲突,先安装的那个首先接收

到该键,第二个便失效。因此,象键盘宏处理器这样的程序通常应首先安装(在链之 BIOS 端的上方),以便能够正确地翻译键码,然后传送给其他 TSR。

为避免打断此链条,并允许多个 TSR 一起工作,互相不干扰,应遵守两个重要规则。首先,如果一个驻留程序替换了一个中断服务程序,总应保留以前程序的地址,在每次激活的开始调用此程序。从而附近的代码总可加到由该中断触发的一系列事件后,而现存的服务程序也没有被删除。此规则防止了丢弃在当前活动的 TSR 之前安装的 TSR。

第二个规则是当一个驻留程序本身请求一个中断服务时,应根据号码来调用该中断,而不是简单地调用链中的下一个程序。例如,如果上图中标为"第一个安装的 TSR"程序打算读键盘,可以简单地调用 old_vector,从其下面的程序直接获得所需要的字符。但是这种方法完全丢弃了"第二个安装的 TSR"及其后安装的所有 TSR,它们再无机会去监视键盘输出并对相应的键响应。而如果调用是原始中断 16h,则请求和返回字符均通过链条上的所有 TSR。此规则防止了丢弃在当前活动之 TSR 之后安装的 TSR。

实际情况可能更复杂,问题之一是有些 TSR 并不使用中断 16h 去监视其 hotkey,而是去截取硬件中断 09h,该中断是由键盘活动直接产生的。一个 TSR 程序还可能截取若干其他中断向量。例如 Sidekick 至少替换了 13 个不同的向量。但无论使用哪个中断向量,上面的两个规则总是应当遵守的。

为避免内存驻留程序与命令行上装入的其他 TSR 发生冲突,可以设备驱动程序的方式编写此代码使其由 MS-DOS 在启动时装入。

7.1.2 与 MS-DOS 共存

设计 TSR 时遇到的下一个问题是使之与 MS-DOS 操作系统兼容,从而使内存驻留程序可以自由地使用 DOS 中断服务程序. 在 TSR 内使用 MS-DOS 服务功能出现问题的根源有两个:一是 TSR 的激活是一个异步事件;二是 MS-DOS 代码是不可重入的.

▲异步激活问题

所谓异步事件指的是,可在任意时刻发生而不管机器的当前状态或是目前的活动进程. 使用 TSR 的一个主要目的便是能够中断任何程序,并在 hotkey 被按下时跳到前台. 通过硬件键盘中断(09h)激活的 TSR 可在任何时刻发生,只要中断系统可工作. 有些 TSR 是通过软件键盘中断(16h)激活的,仅当程序在读键盘时可被激活. 由于中断 16h 频繁发出,故此事件实质上也可看作是异步的.

▲不可重入的 DOS 代码

由于 TSR 是一种异步事件,因此,MS-DOS 正在执行时,或在其他进程正在执行期间,均可激活 TSR. 这样可能造成如下的问题:当一个 MS-DOS 程序开始执行时,它首先将当前的 SS:SP 值存入内存变量中,然后重新设置 SS:SP 以指向一个内部 DOS 堆栈的顶部. 在执行过程种,该堆栈当然会用于临时存储数据. 设想下面的情况:用户按下 hotkey,一个 TSR 中断 DOS 并获得控制,TSR 本身又调用一个 DOS 服务程序,DOS 程序再次切换到内部堆栈的顶部,很有可能是已被用于包含临时数据的同一个堆栈(DOS 功能使用仅有的三个内部堆栈之一). DOS 功能正常执行后,将控制返回给 TSR. 但当 TSR 将控制返回给 DOS 时,由于内部堆栈已被扰乱,系统后面的行为总的来说是不可预料的(常常引起系统

失败)。

如何才能在一个 TSR 内部使用 DOS 呢?有两种可能的方法。

第一种方法是在内存驻留代码内部干脆不使 DOS 功能。此时必须借助 BIOS 服务以及其他一些低层上的技术编写所有子程序。这种方法受很大限制,且损失了 DOS 文件服务功能的主要优点。也很难用 C 语言编程,因为许多重要的 C 库函数使用 MS-DOS 功能。

更巧妙的办法是将 TSR 的激活推迟到 MS-DOS 不活动之后。一旦得知 DOS 已不执行,则可激活 TSR 并可不成问题地自由使用所有 DOS 功能调用。如何才能确定 DOS 当前是否正在执行呢?MS-DOS 本身就提供了一个未写入文档但很著名的办法,即使用中断 21h,功能号 34h。此中断返回一个指向由 DOS 维护其标志的指针,用以表明 DOS 代码当前是否在活动。此 indos 标志实际上是一个计数器,表示已进入 DOS 代码的递归进入次数 (DOS 可在控制条件下调用自身)。如果indos为0,表示 DOS 无活动;如果它大于 0, 表示 DOS 在活动且其代码已进入了一次以上。功能 34h 的调用规则如下:

进入:

AH = 34H

返回:

ES　indos 段

BX　indos 偏移量

此功能应在安装 TSR 的初始化代码中调用,指向 indos 标志的双字指针应被保存,以便以后由中断处理程序来访问。中断处理程序可依据保存的指针很容易地测试 indos,而

不必再次调用中断(由于此亦为一个 DOS 功能,故此时再调用它也是不大安全的)。

可考虑下列经简化的事件序列:中断处理程序(中断 09h 或 16h)检测到按下了 hotkey,便通过保存在代码段中的双字指针测试 indos 标志.如果此标志等于0,中断处理程序激活 TSR 主体(现在 TSR 可自由地调用 DOS 了)。如果此标志大于 0,则不激活 TSR,此中断处理程序可有以下几种选择:

第一种是简单地忽略此 hotkey 并返回,这种方法简单,但会使用户觉得此 TSR 有问题 (由于此方法很简单, 故本章的例子中用此方法)。

第二种方法是用某种信息通知用户当前 TSR 无法被激活,有些办公室实用程序由发出怪调声音来表示这种情况.

第三种方法是由设置一个内部标志(activate 标志)来表示只要安全便尽快激活 TSR. 如果此标志为 ON,则每次中断处理程序被调用,以检查 DOS 是否空. 如果 DOS 空,便激活此 TSR. 程序员亦可为频繁调用的中断(如时针信号中断 08h)安装一个处理程序,如果该激活标志为 ON 且 DOS 不忙激活 TSR. 这种技术的最大缺点是用户按下 hotkey 后可能会有一段明显的延迟时间才能最后激活TSR.还有一个严重的问题,即当COMMAND.COM显示其提示信息并等待用户输入时,indos 标志是设置成 1 的(因为 COMMAND.COM 是使用 DOS 服务来读键盘的)。因此上述保护 TSR 的方法虽然很安全但同时也阻止了 TSR 在 DOS 提示符下被激活。然而事实上此时中断 DOS 还是较安全的,故此限制可不被接受.现在要求的是有一种方法能使当indos标志大于0时可激活TSR,但此时DOS必须处于提示符下等待用户输入。

MS-DOS 本身还提供了一个解决办法. 只要 DOS 处

于空闲状态,在提示符下等待用户输入,便连续地调用 int28h,此中断用于激活假脱机打印一类的后台进程. 对于 TSR 程序设计者,这表明此时去中断 DOS 是相对安全的. 借助于安装一个中断 28h 处理程序,整个"策略"如下:当主 TSR 中断处理程序(几乎可在任意时刻调用)检测到 hotkey 时,仅当 indos 为 0 才激活 TSR. 如果中断 28h 处理程序(仅当 DOS 空闲时调用) 检测到 hotkey. 它便激活 TSR 而不管 indos 标志的状态.

还有一个警告要提出. 通过中断 28h 来中断 DOS 只是相对安全的,但如果 TSR 去调用 DOS 功能 01h 到 0Ch 仍是有危险的,因为在提示符下中断 DOS 时,这些功能调用将会使用 DOS 正在使用的(相同的)堆栈. 而且这一组功能亦被 C 库函数(如 getch,getche 等)所使用. 为了能够使用这些子程序,TSR 代码需要采取最后一个重要措施:必须在 TSR 的本身堆栈中保存当前的 DOS 堆栈,然后在结束时恢复该堆栈. 本章介绍的 TSR 外壳程序提供了保护 DOS 堆栈的快捷方法. 有些读者会想到,如果保存整个 DOS 堆栈,不是可在任何时刻去中断 DOS,而且也用不着去检测 indos 了吗?但毕竟是当 DOS 空闲时去中断它最安全,例如在磁盘操作期间, DOS 很易受到损害.

TSR 与 MS-DOS 和平共处还要求采取最后几个措施. DOS 装入一个程序时,便假定在被调用以结束该程序之前将连续运行. 而当该程序正在运行中,DOS 维护着有关此进程的一些内部信息. 如果 TSR 异步地中断了当前程序,而没有通知 DOS 去作相应的改变,则某些内部信息便会成为错误的. 由 DOS 维护的两个值是程序段前缀 PSP 和磁盘传送区 DAT.

▲程序段前缀 PSP

DOS 维护着当前执行进程之程序段前缀段地址的一个内部记录．PSP 用于维护与相关过程有关的信息．当 TSR 中断了一个程序时,DOS 将继续使用属于被中断程序的 PSP,而不是属于 TSR 的 PSP．欲获得 PSP 值,DOS3.x 提供了一条文档上记载了的功能 62h;DOS2.X 和 3.X 还提供了未写入文档的 get PSP(51h)和 set PSP(50h)。

功能 51h 调用如下:

功能 51h:获得当前 PSP(未写入文档)

进入:

AH 51h

返回:

BX 当前 PSP 的段地址

此功能调用的例子有:

```
mov  ah,51h
int  21h
mov  cs:old_psp,bx
```

功能 50h:设置 PSP(未写入文档)

进入:

AH 50h

BX 新 PSP 段地址

返回:

无。

例子:

```
mov  ah,50h
mov  bx,_PSP        ;为 C 程序建立 PSP
int  21h
```

本章给出 TSR 外壳程序未考虑改变 DOS 的 PSP 记录. 所以如果使用了 PSP,则 TSR 本身可能会借用属于其他进程的 PSP. 特别是,如果打开了文件,DOS 有可能使用属于另一个进程的文件句柄表,而该表可能已没有足够的自由文件句柄可用. 因此,TSR 本身应尽可能少打开文件并在结束前一定要关闭所有文件,最好是一次只打开一个文件,然后立即关闭它。

▲数据传送区 DAT

磁盘传送区是 DOS 用于为文件控制块功能传送文件数据的内存缓冲区,也用于查找匹配文件功能调用 4Eh 和 4Fh. DOS 维护当前 DAT 段:偏移量地址的一条内部记录,如果 TSR 使用 DAT 于某种操作,则当前 DAT 将被临时地改变(以前的值将被保存并恢复). 可通过 DOS 功能调用 2Fh 和 1Ah 来获得并设置 DAT.

最后一点要提及的是,给出的 TSR 例子是不能与版本 1.X 兼容的,因此初始化程序检查操作系统版本(通过功能调用 30h),如果版本是 2x 之前的,则程序应打印出错信息并退出.

注意,前面所讨论的只是与 MS-DOS 兼容的问题. 后面还要讨论其他因素,它们在 TSR 代码主体程序被激活之前还应加以测试.

至此已给出的准则集中在 TSR 与其他内存驻留程序以及与操作系统和平共处的问题上.TSR还必须与系统中其他几个基元:包括被中断的前台程序、BIOS 磁盘活动及中断处理程序等有较好的相互作用关系.

7.1.3 与前台程序共存

由于 TSR 的激活是一个异步事件,故 TSR 可在任意时

刻中断前台程序,但必须保存已存在的机器状态。有三部分机器状态必须被保存,即寄存器、视频显示以及 DOS 参数。

▲保存寄存器

必须强制性地将 TSR 所用的全部寄存器保存并恢复。唯一不须在 TSR 激活程序中保存的寄存器是指令寄存器 IP、码段寄存器 CS 及标志寄存器。因为这些寄存器已由机器指令 int 和 iret 自动地保存并恢复。

▲保存视频显示

如果 TSR 对屏幕写,它必须保存现存的视频状态,将显示设置为适当的视频方式,然后在退出之前完全恢复视频状态。视频状态有三个基元必须被保存:视频显示方式、光标参数以及屏幕数据。

①保存视频方式及显示页

当前视频方式及显示页可由 BIOS 中断 10h 功能 15h 找到。此任务用本书前面 C 函数 getmode 实现。在 TSR 终止前,可使用中断 10h 功能 0 来恢复视频方式,且由功能 5 来恢复显示页。C 函数 setmode 用于恢复这些值。

②保存光标参数

TSR 必须保存光标位置和光标类型,即用于产生光标的顶线与底线。BIOS 中断 10h 功能 3 可获得这两个值,功能 2 用于设置光标位置,功能 1 设置光标类型。可使用函数 getcur 和 setcur 来保存和恢复这组参数。注意,光标参数是根据指定视频页得到的,因此有必要首先获得当前活动的显示页。

③保存屏幕数据

保存屏幕数据的方法取决于被中断程序的当前视频方式是文本方式还是图形方式。

如果发现被中断程序的显示方式是标准文本方式之一(方式 0,1,2,3 和 7) ,则保存和恢复屏幕数据便没有多大困难. 有可能只保存被 TSR 实际使用的屏幕部分,但如果此区域超过了几行,就不如干脆保存整个屏幕. 有两种方法可将数据读出屏幕,然后再恢复原屏幕.

第一种方法是使用 BIOS 中断 10 功能 8,用于在当前光标位置处读入字符及其属性. 让光标在打算保存的每一个位置走一遍,读入并存储每个字符和属性. 类似地,在程序结束前使用功能 9(在当前光标位置与一个字符及其属性)恢复该显示. 此法适用于保存小范围的显示,如果用于在全屏幕上移动光标则实在太慢了.

保存屏幕的第二种方法是采用对视频内存区直接写. 此方法相当快. 保存及恢复整个屏幕的简单方法是调用汇编函数 pushscr 和 popscr,这两个函数可用 C 或汇编语言程序来调用.

如果视频显示方式是图形方式之一 (4, 5, 6, 8, 9, 10, 13, 14, 15, 16), 屏幕数据的保存便不那么简单. 文本方式使用的视频内存区仅由 4000 个字节组成. 而对于 CGA 卡图形方式的内存映照大小便为 32K 字节,EGA 适配器则可用多达 256K 字节的视频内存区. 因为当离开图形方式时, 欲保存全屏幕数据可能要求有大量内存(但 TSR 应尽可 能少用内存).

如果 TSR 只使用文本输出,似乎只有需要保存与恢复实际被 TSR 占用的视频 RAM 的 4000 字节区域.但问题是,当调用CGABIOS将此显示切换成文本方式时,它排除当前的屏幕数据. 下面介绍在一个图形方式程序上显示一个 TSR 的三种可能办法.

第一种方法是，只要 TSR 在某个图形方式程序下"弹出"，便牺牲原来的显示，即使用 BIOS 功能保存当前图形方式，在 TSR 开头切换到文本方式，然后在退出前恢复图形方式．这种方法意味着，用户激活该 TSR 时，排除现存的屏幕，在返回主应用程序后要重新生成屏幕．

第二种方法是仅保存实际要被 TSR 重写的视频内存区部分．如果显示使用文本方式，则保存段地址 B800h 处开始的 4000 个字节．使用 BIOS 以确定当前方式，但当切换到文本方式后，写程序要绕过 BIOS 并直接对视频控制器芯片编写程序(对于 CGA，此芯片为 6845)．这样一个程序应包括 BIOS 方式设置过程所执行的所有任务(除去排除屏幕数据以外)．此程序必须做到：

· 由程序将"颜色选择寄存器"(CGA 的此端口为地址 3D8h 处的 I／O)设置成适当方式．

· 按照 BIOS 视频初始化表中的参数设置所有寄存器值(表的地址由中断向量 10h 指向)．

· 更新所有 BIOS 视频参数(这些参数在地址 0400h:0049h 处开始)．

切换方式不同，此步骤应有相应改变．在编写这种程序之前，应先参考 BIOS 源程序清单(在 PC 机中，改变方式的过程标记为 EST_MODE，在偏移量 FOFCh 处开始)，并在《Options and Adapters Techical Reference》一书的第 2 卷中查找 CGA 部分的寄存器信息．

第三种保存当前视频内存区内容的方法仅能用于 EGA BIOS．首先保存将被 TSR 以文本方式重写 4000 字节的视频内存区，然后当调用中断 10h 功能 0 去改变方式时，设置 AL 寄存器的高阶比特，这将使 EGA 程序去保存当前视频内

存数据.

如果有不同的图形方式，情况就更为复杂．例如，Hercules 图形方式既不由标准的 BIOS 视频参数指明,也不由获得当前视频方式的 BIOS 功能(功能 15)报告．为了维护与这种图形标准的兼容性,必须使用专门测试以检测显示方式并对特定端口编写程序以切换方式．在非标准的图形方式下(尤其在使用兼容的适配卡时),TSR 的激活常常失败．

下面为保存和恢复完全视频状态所需基本步骤做一个小结.

· 保存当前视频方式和活动显示页.

· 为该活动显示页保存当前光标位置和类型.

· 保存屏幕数据(注意,必须在方式切换前执行).

· 切换成适当的视频方式．如果必要,该应用程序可自由地向屏幕写并修改光标位置及类型.

· 退出前,恢复以前的视频方式.

· 恢复光标位置和类型.

· 恢复屏幕数据.

▲保存 DOS 参数

必须被保存和恢复的机器状态之最后基元,是由当前进程维护的两个 DOS 变量:程序段前缀 PSP 段地址和磁盘传送地址 DAT 的段 偏量地址．这两个变量在"与 MS-DOS 共存"一节中已经讨论.

7.1.4 与 BIOS 磁盘活动共存

磁盘驱动器是不可共享的资源—— 一个时刻只有一个进程能使用它,并且要求同步工作.因此当BIOS进行磁盘服务时绝不要去激活 TSR．所有 BIOS 磁盘服务均是通过中断 13h 访问的．通常应用程序通 MS-DOS 与磁盘通信,然后由

DOS 调用低层的 BIOS 程序. 如果遵守这种步骤,则前述避免中断 DOS 的技术就足以防止破坏磁盘活动. 但有些应用程序通过中断 13h 直接访问 BIOS 磁盘服务. 因此需有一个单独的机构用于在中断 13h 进行过程中防止 TSR 被激活. 简单的方法是捕获此中断,并当 BIOS 忙时设置一个标志.

7.1.5 与中断处理程序共存

由硬件中断通知的事件通常被提示性地服务,例如系统时钟信号或串行端口到达一个字符. 但 TSR 能够"阻塞"硬件中断任意长的时间,因此应使用下面两条准则以减少 TSR 对这些中断处理的时间延迟.

首先尽可能快地发出机器指令 sti. 一个中断程序接收控制后,硬件中断失效,必须通过sti指令使之生效(也可以通过cli 指令使之失效). 记住,硬件中断生效后,TSR便可能被其他的中断处理程序异步地"挂起". 因此仍需在代码的关键部分(如何能破坏数据的分支子程序等)使硬件中断失效.

其次,如有必要,则向 8259A 中断控制器发送一个中断结束信号. 此准则仅适用于通过硬件中断激活的 TSR(例如键盘中断 09h).

7.1.6 可重新进入的问题

TSR 的设计已超出了简单顺序程序设计的命题. 由于存在着硬件中断机构,故有可能递归地进入本身内存驻留代码. 为说明这种可能性,需考虑两个激活 TSR 的常用方法:通过硬件键盘中断 09h 及通过软件键盘中断 16h.

通过硬件键盘中断激活时,用户按下 hotkey,产生中断 09h 并使程序进入你的中断处理程序,由其检测此 hotkey 并激活 TSR 的主体. TSR 激活后,用户可以再按同样的 hotkey,使当前执行的 TSR"挂起"(只要中断系统有效),并从

头第二次执行该 TSR 代码.

使用软件键盘中断时,TSR 本身被安装在中断 16h 处理程序链上. TSR 激活后,有两种可能的方法可使该 TSR 递归地进入. 第一种方法是,程序本身可发出中断 16h 以读取键盘,如果用户再次按动该 hotkey,则 TSR 从开始重新进入. 第二种方法是,即使程序本身不发出中断 16h,只要中断系统当前有效,硬件中断可以发生(例如中断 08h 时针信号),便可使程序分支到以后通过中断 16h 读取键盘的程序中. 此时亦实现了你的代码的递归重进入.

设计时可以正确处理递归调用的代码属于可重新进入的. 如果代码不是可重新进入的,则会因数据存储在固定的内存地点而产生严重问题. 例如,如果程序本身在内存变量中存储临时值,递归调用有可能破坏这些数值. 另外如果程序切换到自身堆栈中的一个固定地址,则递归调用亦可能破坏以前所存储的数值. 注意,由于 MS-DOS 本身是不可重入的,故常常产生上述的问题.

使内存驻留程序具有可重入性有两种基本办法. 第一种并且是最简单的办法是,只要代码当前活动,便设置一个 busy 标志. 激活 TSR 的中断处理程序必须首先检查此标志. 如果此标志置位,则程序立即返回,防止了递归地激活;如果此标志未置位,则继续其他测试并有可能激活 TSR. 后面的 TSR 实例便采用这种简单的办法.

第二种产生可重新进入代码的方法是允许此代码递归地进入,但要避免在固定内存地址中存储临时数值. 防止破坏局部数据有两个要紧的警告:第一是应将局部值存储在寄存器中而不是内存变量中,存储并恢复所有寄存器的原始内容. 第二是使用下列方法之一防止自身堆栈的数据被破坏.

· 不要使用自身堆栈(local stack)。这便防止了 TSR 破坏自己的堆栈,但仅使用属于被中断进程的堆栈,有时可能不够大。只有 TSR 很小时才会使用很小的堆栈,此刻方可使用这种不用自身堆栈的技术。

· 使用一系列自身堆栈,并维护一个指针以指向下一个可用堆栈区域的顶部。在代码开始处,切换到下一个自由堆栈(而不总是切换到一个单独堆栈顶部)。

· 仅当代码段处于中断无效的期间或是该区域已设置了阻止中断进程重进入标志时,才临时切换到固定的自身堆栈。

7.1.7 Microsoft 标准

为避免内存驻留程序编写中的混乱,Mircosoft 与其他软件开发公司一起正在制定一组编写 TSR 的标准规则。目前已公布的规则只是很基本的,很不完善。在完成最后文件并为众多软件者接受之前,标准的许多潜在好处还无法实现,但仍存在着一些建议的规则对于产生可靠的 TSR 很有用。这些规则大致分为两类:关于良好表现的规则及关于标准接口的定义。

▲关于良好表现(Good Conduct)的规则

这些可选的规则有助于生成可靠的 TSR。在采取标准之前,这些规则可立即带来好处。应记住,一个 TSR 通常可在任意时刻被另一个内存驻留程序中断,遵守以下几条规则便可实现这一点。

首先,一个内存驻留程序只应在安装时一次性地替换中断向量.有些TSR设置一个时钟、为后面的 TSR 安装后重新占用中断向量。这种方法可用于防御某些有错误的程序,但有失于"粗暴".一个 TSR 程序应假定有一个不断的控制链,并

知道自己在链中的位置.

第二个规则是,只要有可能改变视频显示方式或光标参数便使用 BIOS. 如果基于某种理由有必要直接改变视频控制器的端口设置,TSR 应更新在低内存地址处维护的 BIOS 数据,以随时通知系统和应用程序所做的修改. 这些参数处于段 0040h 偏移量 0049h～0066h,对于 EGA,还有附加偏移量 0084h～00ABh. 特别要注意,对于方式控制寄存器(单色系统的 3B8h,CGA 系统的 3D8h)的任何改变应由在地址 0040h:0065h 处 CRT_MODE_SET 域的相应变化来反映. 如果程序使用自己的软件光标而不使用硬件光标,应连续更新地址 0040h:0050h 处 CURSOR_POSN 域中的光标位置(这使TSR能够识别当前的光标位置).这些域的详细定义可查看 PC/AT 技术参考手册的 BIOS 清单与 EGA 手册.

第三是使用致命错误处理程序(中断 24h)来对硬件错误执行下列动作:将 DOS 返回的出错代码保存在一个全局标志中,以便以后可用程序来检查;给寄存器 AL 赋 0 值以通知 DOS 忽略此错误,然后发出中断返回指令. 不要直接跳回程序,这样做将使 DOS 无法清其堆栈.

▲标准接口

这部分规范可实现比和平共处更高级的性能,它定义一组数据记录和调用协议,使 TSR 和一组标准服务程序的工作特性均可为其他程序所用. 软件开发者不必急于实现此接口, 因为最后的格式尚未定义,且还有待被更多的软件编写者接受. 但其最本质的特点还是值得提及的,包括两个基本部分: 一个程序识别记录和一组标准函数.

程序识别记录(program identification record)很象设

备驱动程序的设备标题．它是一条包含在 TSR 中的标准记录,用于为其他程序提供基本信息．在理想情况下,如果系统中的每个 TSR 都包含这样一条记录,则任何程序均有可能确定当前运行环境下所安装的所有 TSR 的标识及工作特性,然后在必要时或检测到冲突时修正自己的行为．该记录有若干字段,其一是一个指示一个合法程序标识记录的 ID,其二是一个版本号,其三是指向被替换的所有中断的数据的一个指针,然后指向 TSR 使用的 hotkey 的指针以及一个程序 ID 字符串．

每个中断的数据包含在一条中断信息记录(interrupt information record)中．此记录中的一个域是指向以前中断程序的一个指针,用它可管理一个给定中断的整个程序链．中断信息记录中还包含一个优先级字段,给出了该服务程序在此中断链中的要求位置．例如,最高优先级的 TSR 应最接近于链条底部的 BIOS．这样某个 TSR 便有可能搜索已安装在给定中断上的处理程序序列,并将本身安装在链条的合适位置(理想情况下,清单的顺序应按照优先及字段中的值来安排)．此功能实现了规范的一个重要目标:所有 TSR 都应当正常工作而不管装入顺序．

如何查找给定中断号的中断信息记录和程序标识记录呢？规范的另一个建议是在每个中断过程(即由该中断向量指向的代码)前置一条记录．该记录包含相应中断信息记录的指针．中断信息记录又包含指向程序标识记录的指针．由此可见记录连接的复杂。进一步的细节请参见1986 年 2 月1日 Microsoft 发布的一个文件:《 Programming Guidlines for MS-DOS Terminate and Story Resident Programs》．

接口的第二部分是 TSR 提供的一组标准服务程序,可由

任何其他程序通过中断15h(选中此向量是因为它很少用)来访问. 目前,绝大多数TSR是仅当用户按下hotkey时才激活. 如果采用了建议的功能,则另一个程序便可激活一个TSR,就象现在已实现的一个程序可调用操作系统服务程序或装入另一个应用程序一样. 标准中含有可获得有关特定TSR信息、激活或解除特定TSR功能以及使整个TSR工作或失效的各种函数. 这样在程序中便能够运行各种TSR功能. 这种方式还提供了程序临时部分与驻留部分通信的标准通道.

标准还包括一个解除TSR的标准机制用于解除代码的运行并释放内存,以及包括TSR使用扩展内存的标准机制.

7.2 实现C语言程序的内存驻留

本节介绍如何用调用汇编程序TSR将一个C程序变换为TSR. 下面两个程序是说明性的C程序和用于初始化并激活TSR的汇编程序. 提供这些源程序的目的是为了说明一个内存驻留程序的重要特点,给出TSR设计中遇到主要问题的解决办法,并提供一个使用C语言生成TSR的简单且快捷的方法.

```
/*
     Figure:    11.2
     File:      TSR.C

     A demonstration C program that terminates, stays resident, and is
     activated by a hotkey

     This program must be linked with the assembler module TSRA.ASM
     (Figure 11.3) and the object module(s) containing the other external
     routines listed below:
*/

#include <stdio.h>

/********  External Routines that must be linked with this program.  ********/

int tsr (void (*)(), int);                    /* Figure 11.3           */
void pushscr (void);                          /* Figure 8.12           */
void printv (char *, int, int, int);          /* Figure 8.4            */
void popscr (void);                           /* Figure 8.12           */

/********************  Internal Routine Declarations  ********************/
```

```
void demo (void);
int getkey (void);

/******************************  Main Program  ********************************/

void main ()
     {
     int error;                        /* Error code.                           */

                                       /* Print message -- also perform any     */
                                       /* initializations at this point.        */
     printf ("Installing TSR Demo ... \n");
     printf ("Press <alt>-<left shift> to activate.\n");

     error = tsr (demo, 0x000A);       /* Install program as TSR, with hotkey   */
                                       /* <alt>-<left-shift> activating 'demo', */
                                       /* and terminate program.                */

     printf ("Error %d installing TSR.\n", error);
                                       /* If code reaches this point, an error  */
                                       /* has occurred.                         */
     } /* end main */

/***********************  Routine Activated by Hotkey  ***********************/

void demo ()                           /* C entry point when hotkey is pressed. */
     {
     int row = 3;
     int col = 10;

     pushscr ();                       /* Save screen of interrupted program.   */

     printv ("--------------------------------------------",15,row++,col);
     printv ("|                                          |",15,row++,col);
     printv ("|           T  S  R    D  E  M  O          |",15,row++,col);
     printv ("|                                          |",15,row++,col);
     printv ("|                                          |",15,row++,col);
     printv ("|                                          |",15,row++,col);
     printv ("|        press any key to continue ...     |",15,row++,col);
     printv ("|                                          |",15,row++,col);
     printv ("|                                          |",15,row++,col);
     printv ("--------------------------------------------",15,row++,col);

     getkey ();
     popscr ();                        /* Restore screen of interrupted program.*/

     } /* end demo */

#include <dos.h>

int getkey ()                          /* Reads a key using BIOS interrupt 16h. */
                                       /* See Figure 5.3.                       */
     {
     union REGS reg;

     reg.h.ah = 0;
     int86 (0x16, &reg, &reg);
     return (reg.x.ax);

     }  /* end getkey */
```

上面是用C语言编写的TSR程序例.

```
1:    page 50,130
2:
3:
4:    ;File:     TSRA.ASM
5:
6:    ;Routines for generating a terminate and stay-resident C program
7:
8:    public     _tsr                          ;Procedure called by C program.
9:
10:   extrn      __psp:word                    ;C global variable containing the
11:                                            ;segment address of the PSP.
12:
13:   HEAPSIZE   equ     64                    ;Size of heap in 16 byte paragraphs;
14:                                            ;must increase this value on a DOS
15:                                            ;Allocation Error.
16:
17:   _data      segment word public 'DATA'
18:   _data      ends
19:
20:   dgroup     group   _data
21:
22:   _text      segment byte public 'CODE'
23:
24:   extrn      _sbrk:near                    ;C library function which returns
25:                                            ;offset (w.r.t. DS) of break address.
26:
27:   assume     cs:_text, ds:dgroup
28:
29:   ;*********************** Code segment data. ****************************
30:
31:   c_ss       dw      ?                     ;C Stack Segment.
32:   c_sp       dw      ?                     ;C Stack Pointer.
33:   c_ds       dw      ?                     ;C Data Segment.
34:   c_es       dw      ?                     ;C Extra Segment.
35:
36:   fun_ptr    dw      ?                     ;Address of main TSR C function.
37:
38:   hot_key    dw      ?                     ;Hotkey keyboard shift-status mask.
39:
40:   c_dta_off  dw      ?                     ;C Disk Transfer Address.
41:   c_dta_seg  dw      ?
42:   d_dta_off  dw      ?                     ;Disk Transfer Address of interrupted
43:   d_dta_seg  dw      ?                     ;program.
44:
45:   indos_ptr  label   dword                 ;Pointer to DOS "indos" flag.
46:   indos_off  dw      ?
47:   indos_seg  dw      ?
48:
49:   int28_vec  label   dword                 ;Old interrupt 28h vector.
50:   int28_off  dw      ?
51:   int28_seg  dw      ?
52:
53:   int13_vec  label   dword                 ;Old interrupt 13h vector.
54:   int13_off  dw      ?
55:   int13_seg  dw      ?
56:
57:   int09_vec  label   dword                 ;Old interrupt 09h vector.
58:   int09_off  dw      ?
59:   int09_seg  dw      ?
60:
61:   break_off  dw      ?                     ;Save heap break offset.
62:
63:   busy       db      0                     ;Flag to prevent recursive TSR calls.
64:   in_bios    db      0                     ;Flag to indicate int 13h activity.
65:
66:   dos_ss     dw      ?                     ;Saves SS of interrupted program.
67:   dos_sp     dw      ?                     ;Saves SP of interrupted program.
68:
69:   ;********************** Installation procedure *************************
70:
71:   _tsr       proc    near                  ;Makes program resident, installs
72:                                            ;hotkey, & terminates.
73:   comment    /*
74:              This function terminates a C program, leaving the code resident in
75:              memory.  After the program terminates, the specified C function
```

```
76:             will be activated by the selected hotkey.  Normally, this routine
77:             never returns to the C program;  if it does return, an error
78:             occurred and one of the following codes is passed back:
79:
80:             Error Codes:
81:
82:             1       insufficient memory
83:             2       pre DOS 2.0 version
84:             3       TSR already installed      (not currently implemented)
85:
86:             int tsr (funptr, hotkey);
87:             void (*)() funptr;              Pointer to function to activate.
88:             int hotkey;                     Mask for hotkey which activates function.
89:
90:             */
91:
92:  aframe     struc                           ;Stack frame template.
93:  abptr      dw      ?
94:  aret_ad    dw      ?
95:  afun_ptr   dw      ?                       ;Pointer to C TSR function to activate.
96:  ahot_key   dw      ?                       ;Hotkey keyboard status mask.
97:  aframe     ends
98:
99:             push    bp
100:            mov     bp, sp
101:            push    es
102:
103:            mov     ah, 30h                 ;Test DOS version.
104:            int     21h
105:            cmp     al, 0                   ;Major version returned in AL.
106:            jg      a00                     ;2.x or greater.
107:            mov     ax, 2                   ;Version 1.x, set error code.
108:            jmp     a02                     ;Quit.
109: a00:
110:            mov     cs:c_ss, ss             ;Save C SS.
111:            mov     cs:c_sp, sp             ;Save C SP.
112:            add     cs:c_sp, 10             ;Adjust saved C stack pointer to its
113:                                            ;value before 'tsr_init' was called.
114:            mov     cs:c_ds, ds             ;Save C DS.
115:            mov     cs:c_es, es             ;Save C ES.
116:
117:            mov     ax, [bp].afun_ptr       ;Save pointer to C TSR function.
118:            mov     cs:fun_ptr, ax
119:            mov     ax, [bp].ahot_key       ;Save hotkey mask.
120:            mov     cs:hot_key, ax
121:
122:            mov     ah, 2fh                 ;Get C Disk Transfer Address.
123:            int     21h
124:            mov     cs:c_dta_off, bx        ;Save it.
125:            mov     cs:c_dta_seg, es
126:
127:            mov     ah, 34h                 ;Call DOS to retrieve pointer to
128:            int     21h                     ;"indos" flag.
129:            mov     cs:indos_off, bx        ;Save dword pointer to "indos" flag.
130:            mov     cs:indos_seg, es
131:
132:            mov     ah, 35h                 ;Get old interrupt 28h vector.
133:            mov     al, 28h
134:            int     21h
135:            mov     cs:int28_off, bx        ;Save old vector.
136:            mov     cs:int28_seg, es
137:
138:            mov     ah, 35h                 ;Get old interrupt 13h vector.
139:            mov     al, 13h
140:            int     21h
141:            mov     cs:int13_off, bx        ;Save old vector.
142:            mov     cs:int13_seg, es
143:
144:            mov     ah, 35h                 ;Get old interrupt 09h vector.
145:            mov     al, 09h
146:            int     21h
147:            mov     cs:int09_off, bx        ;Save old vector.
148:            mov     cs:int09_seg, es
```

```
149:
150:            push    ds
151:            mov     ax, cs              ;Set new interrupt 28h vector.
152:            mov     ds, ax
153:            mov     ah, 25h
154:            mov     al, 28h
155:            mov     dx, offset int28
156:            int     21h
157:
158:            mov     ax, cs              ;Set new interrupt 13h vector.
159:            mov     ds, ax
160:            mov     ah, 25h
161:            mov     al, 13h
162:            mov     dx, offset int13
163:            int     21h
164:
165:            mov     ax, cs              ;Set new interrupt 09h vector.
166:            mov     ds, ax
167:            mov     ah, 25h
168:            mov     al, 09h
169:            mov     dx, offset int09
170:            int     21h
171:            pop     ds
172:                                        ;Calculate number of paragraphs to
173:                                        ;retain in memory.
174:            mov     ax, HEAPSIZE        ;Size of heap in 16 byte paragraphs.
175:            mov     bl, 16              ;Convert to bytes.
176:            mul     bl
177:            push    ax                  ;Attempt to move break address.
178:            call    _sbrk               ;'Sbrk' returns offset (wrt DS) of
179:            add     sp, 2               ;former break address.
180:            mov     cs:break_off, ax    ;Save break offset address.
181:
182:            cmp     ax, -1              ;'Sbrk' returns -1 if insufficient
183:            jne     a01                 ;memory is available for heap.
184:            mov     ax, 1               ;Set error code.
185:            jmp     a02                 ;Return to C.
186:    a01:
187:            mov     ax, HEAPSIZE        ;Restore C heap.
188:            mov     bl, 16
189:            mul     bl
190:            neg     ax
191:            push    ax
192:            call    _sbrk
193:            add     sp, 2
194:
195:            mov     dx, cs:break_off    ;Convert offset of break address to
196:            add     dx, 15              ;absolute segment (paragraph) value.
197:            mov     cl, 4               ;Convert offset to paragraphs,
198:            shr     dx, cl              ;rounding up.
199:
200:            mov     ax, ds              ;Add DS.
201:            add     dx, ax              ;DX now has segment value of current
202:                                        ;break address.
203:
204:            mov     ax, __psp           ;AX now has segment address of PSP.
205:
206:            sub     dx, ax              ;DX now has number of paragraphs of C
207:                                        ;code and data.
208:
209:            add     dx, HEAPSIZE        ;Leave room for Heap.
210:
211:                                        ;Call DOS Terminate and Stay Resident
212:            mov     ah, 31h             ;service.
213:            mov     al, 0               ;Set errorlevel to 0.
214:            int     21h
215:
216:    a02:    pop     es                  ;Branch here if an error occurs.
217:            pop     bp
218:            ret                         ;Return to C program on error only.
219:
220:    _tsr    endp
221:
```

```
222:    ;*************************  Interrupt Handlers ********************************
223:
224:    int28       proc    near                ;Patches into interrupt 28h.  Similar
225:                                            ;to 'int09' except the 'indos' flag
226:                                            ;is not tested.
227:
228:                pushf                       ;Chain to prior installed int 28 handler.
229:                call    cs:int28_vec
230:
231:                cli                         ;Make sure interrupts are disabled.
232:                cmp     cs:busy, 0          ;Test 'busy' flag to prevent recursive
233:                je      b01                 ;calls.
234:                iret                        ;TSR busy.
235:    b01:                                    ;TSR not busy.
236:                cmp     cs:in_bios, 0       ;Test for BIOS disk activity.
237:                je      b02
238:                iret                        ;BIOS disk services active.
239:    b02:                                    ;BIOS disk services NOT active.
240:                push    ax                  ;Test if hot key is pressed.
241:                push    es
242:                xor     ax, ax
243:                mov     es, ax              ;Read BIOS keyboard flag at 0000:0417h.
244:                mov     ax, word ptr es:[417h]
245:                and     ax, cs:hot_key      ;Test if keyboard flag has all bits on
246:                cmp     ax, cs:hot_key      ;that are on in 'hot_key'.
247:                pop     es
248:                pop     ax
249:                je      b03
250:                iret                        ;Hot key not pressed.
251:    b03:                                    ;Hot key pressed.
252:                call    activate            ;Activate TSR.
253:                iret                        ;Return to interrupted process.
254:
255:    int28       endp
256:
257:
258:    int13       proc    far                 ;Patches into interrupt 13h to set
259:                                            ;flag when disk services are active.
260:
261:                mov     cs:in_bios, 1       ;Turn on BIOS active flag.
262:
263:                pushf                       ;Invoke original interrupt.
264:                call    int13_vec
265:
266:                mov     cs:in_bios, 0       ;Turn off BIOS active flag.
267:
268:                ret     2                   ;Return from interrupt, saving flags.
269:
270:    int13       endp
271:
272:
273:    int09       proc    near                ;Patches into interrupt 09h.
274:
275:                pushf                       ;Chain to prior installed int 09h handler.
276:                call    cs:int09_vec
277:
278:                cli                         ;Disable interrupts for tests.
279:                cmp     cs:busy, 0          ;Test 'busy' flag to prevent recursive
280:                je      c01                 ;calls.
281:                iret                        ;TSR busy.
282:    c01:                                    ;TSR not busy.
283:                cmp     cs:in_bios, 0       ;Test for BIOS disk activity.
284:                je      c02
285:                iret                        ;BIOS disk services active.
286:    c02:                                    ;BIOS disk services not active.
287:                push    ax                  ;Test if hot key is pressed.
288:                push    es
289:                xor     ax, ax
290:                mov     es, ax              ;Read BIOS keyboard flag at 0000:0417h.
291:                mov     ax, word ptr es:[417h]
292:                and     ax, cs:hot_key      ;Test if keyboard flag has all bits on
293:                cmp     ax, cs:hot_key      ;which are on in 'hot_key'.
294:                pop     es
```

```
295:            pop     ax
296:            je      c03
297:            iret                        ;Hot key not pressed.
298:    c03:                                ;Not busy & hot key pressed.
299:            push    ds                  ;Test if in DOS.
300:            push    bx
301:            lds     bx, cs:indos_ptr    ;Load pointer to 'indos' flag.
302:            cmp     byte ptr ds:[bx], 0 ;Test 'indos' flag.
303:            pop     bx
304:            pop     ds
305:            je      c04
306:            iret                        ;Are in DOS.
307:    c04:                                ;Not busy, hot key pressed, & not in DOS.
308:            call    activate            ;Activate TSR.
309:            iret                        ;Return to interrupted process.
310:
311:    int09   endp
312:
313:
314:    activate proc   near                ;Activates the C TSR function.
315:
316:    comment /*
317:            This procedure performs a context switch which saves the current
318:            machine state, initializes the runtime environment for C, calls
319:            the C TSR function, and then restores the former state.  It is
320:            called by either 'int28' or 'int09' when the hotkey is pressed,
321:            and it is safe to interrupt the current program.
322:            */
323:
324:            mov     cs:busy, 1          ;Set busy flag to prevent recursive calls.
325:
326:                                        ;Switch to C stack.
327:            mov     cs:dos_ss, ss       ;Save current SS.
328:            mov     cs:dos_sp, sp       ;Save current SP.
329:
330:            mov     ss, cs:c_ss         ;Set up C stack segment.
331:            mov     sp, cs:c_sp         ;Set up C stack pointer.
332:
333:            push    ax                  ;Save machine state on C stack.
334:            push    bx
335:            push    cx
336:            push    dx
337:            push    bp
338:            push    si
339:            push    di
340:            push    ds
341:            push    es
342:
343:                                        ;Save the DOS stack.
344:            mov     cx, 64              ;Set counter for 64 words.
345:            mov     es, cs:dos_ss       ;Point ES:SI to top of DOS stack.
346:            mov     si, cs:dos_sp
347:    d01:
348:            push    word ptr es:[si]    ;Loop to save 64 words of DOS stack.
349:            inc     si
350:            inc     si
351:            loop    d01
352:
353:            mov     ah, 2fh             ;Save DTA for interrupted program.
354:            int     21h
355:            mov     cs:d_dta_off, bx
356:            mov     cs:d_dta_seg, es
357:
358:            mov     ah, 1ah             ;Set up C DTA.
359:            mov     dx, cs:c_dta_off
360:            mov     ds, cs:c_dta_seg
361:            int     21h
362:
363:            mov     ds, cs:c_ds         ;Set up C segment registers.
364:            mov     es, cs:c_es
365:
366:            sti                         ;Set interrupts back on.
367:            call    cs:fun_ptr          ;Call the C TSR function.
```

```
368:             cli                              ;Turn interrupts back off.
369:
370:             mov     ah, 1ah                  ;Restore DTA of interrupted program.
371:             mov     dx, d_dta_off
372:             mov     ds, d_dta_seg
373:             int     21h
374:
375:                                              ;Restore DOS stack.
376:             mov     cx, 64                   ;Set counter for 64 words.
377:             mov     es, cs:dos_ss
378:             mov     si, cs:dos_sp
379:             add     si, 128
380:   d02:                                       ;Loop to restore 64 words of DOS stack.
381:             dec     si
382:             dec     si
383:             pop     word ptr es:[si]
384:             loop    d02
385:
386:             pop     es                       ;Restore machine state.
387:             pop     ds
388:             pop     di
389:             pop     si
390:             pop     bp
391:             pop     dx
392:             pop     cx
393:             pop     bx
394:             pop     ax
395:
396:             mov     ss, cs:dos_ss            ;Restore stack of interrupted process.
397:             mov     sp, cs:dos_sp
398:
399:             mov     cs:busy, 0               ;Reset busy flag.
400:             ret                              ;Return to 'int28' or 'int09'.
401:
402:   activate  endp
403:
404:   _text     ends
405:             end
406:
407:
```

上面是将C程序转换为TSR的汇编语言模块。

开发一个可应付各种情况,并可在一大类软、硬件环境下工作的完善的TSR需要很大的程序量。本文介绍的TSR过程的主要设计目标之一,是尽可能地简化代码,以突出主要的程序部分并为增加自信的功能提供一个容易"加工"的起点。首先介绍如何在C程序中使用汇编模块来生成一个TSR,然后描述该汇编语言的实现,重点放在设计方案的选择上,最后讨论如何强化该汇编程序功能。

7.2.1 在C程序中使用 tsr 函数

使用tsr函数可将一个普通的C程序很简单地变换成一个TSR,在低层次上的绝大多数实现都包括在汇编程序模块中。如上节中用C语言编写的TSR程序所示,C程序首先向

用户显示出一些信息执行必要的初始化,这是执行初始化的最后机会,因为下一次程序是在通过hotkey激活后才接受控制.然后程序调用TSR函数,传递给它两个参数.第一个参数是通过hotkey激活的C函数的地址,即main是程序从命令行上运行以装入和初始化TSR的C程序进入点,而其地址传送给TSR的函数是按下hotkey后C程序的进入点.第二个参数是用于激活该程序的hotkey. hotkey不是一个ASCII字符或特殊字符(例如一个功能键),而是由BIOS用于表示特殊"上档键(shift key)"组合的特征码(mask).本例使用特征码000Ah,表示同时按下Alt及Left-shift键,即只要用户同时按下这两个键,便能激活TSR. 下面是用于"上档键"的BIOS代码:

比特	上档键
0	按下 Right-Shift
1	按下 Left-Shift
2	按下 Ctrl-Shift
3	按下 Alt-Shift
4	ScrollLock 工作
5	NumLock 工作
6	CapsLock 工作
7	Ins 工作
8-10	未用
11	Ctrl-Numlock 工作
12	按下 ScrollLock
13	按下 NumLock
14	按下 CapsLock
15	按下 Insert

其中所谓"工作(active)"表示该键为一跟头式开关,不是

ON 就是 OFF. 如果该比特为 ON,则相应状态便是当前工作的(此时该键按不按下是没有关系的). 因为 hotkey 特征码是作为一个参数传送给 tsr 的,因此用户欲修改所选的 hotkey 并不难(从而可避免与其他程序冲突).

tsr 函数存储激活 TSR 的汇编程序所需要的数据,然后安装中断处理程序并结束该程序,将代码和数据留在内存中. 如果一切正常,此程序不再返回,并且躲开了正常的 C 结束代码. 如果出现了错误条件,tsr 返回下列出错代码之一:

1　内存不够

2　DOS 版本错(1,X)

3　已安装了 TSR(本例中尚未实现此功能)

因为本例中的 TSR 没有检查是否已经安装了 TSR,故实际上此版本程序并不返回第三个出错代码.如果发生错误且 tsr 返回,则 main 显示出错误信息并正常结束,即代码未留在内存中.

如前所述,按下 hotkey 后执行的实际代码处于一个函数中而不是 main 中(本例中,此段代码包含在 demo 函数中). 由于 TSR 进入点是一个普通 C 函数,故可调用任意个其他函数,但仍有几个特殊的限制.

首先,C 语言的 TSR 进入函数必须保存和恢复当前视频状态. 本例使用 pushscr 和 popscr 保存和恢复屏幕数据,但更完善的实现还应保存光标参数和视频方式. 汇编函数自动保存寄存器和 DOS 磁盘传送区.

此外,有几个 C 的库函数在 TSR 环境下不能正常工作,它们是 system,alloc,exit 等. 函数 system 需要在该程序上面有大块已分配了的内存,当从 TSR 中调用时将返回"内存不够用"信息. 函数 alloc 从 C 启动代码初始化时保留的

heap 空间分配内存。但 tsr 将此内存区的绝大部分返回给了系统,实际可用的只是由常数 HEAPSIZE 指定的量,但 alloc 函数没有意识到已减少了的 heap 大小。如果所用内存超过了实际分配给该进程的量,就会产生 DOS 内存分配错误。TSR 最好使用 alloca,它从堆栈空间分配内存(堆栈空间安全地驻留在 TSR 的内存中)。下面将讨论如何计算内存驻留代码的大小。任何作为 TSR 一部分的 C 函数均不要使用 exit 调用。程序必须使用 return,因此控制通过此汇编模块返回,汇编模块要求正确地恢复机器状态。在使用此系统的过程中,人们还会发现其他会出现问题的 C 库函数。

7.2.2 汇编语言子程序的实现

本节介绍将 C 程序作为一个 TSR 安装并激活的汇编语言程序。尽管编写灵活、紧凑 TSR 的最好办法是用汇编语言编写全部代码,但对于内存大小不是那么要紧的程序来说,使用 C 语言编程带来的方便远胜过其缺点。C 语言的接口问题限制了程序员对代码和数据准确位置以及运行时间环境建立的控制。例如,正常情况下,一个完全用汇编语言编写的 TSR,将初始化代码安排在程序结束处,从而可释放初始化代码占用的内存。但 C 编译程序将所有代码均放在数据段之前,从而阻止了初始化代码的释放。

注意:为了与所给汇编模块兼容,C程序必须使用"小内存模式(Small-memory model)"进行编译。本节介绍该汇编模块的三个主要部分:初始化代码、中断处理程序以及激活子程序。

▲初始化代码

代码段 _text(31～67 行)的开始用于声明用中断处理程序访问的变量。一个中断处理程序接受控制后,DS 寄存器不

指向 C 数据段,因此将这些数据置于__text 段比置于__data 段更为方便. 注意,这些变量的所有引用均使用 CS:段头.

103～108 行测试 DOS 版本是否为 2.0 以后的版本,因为更早的版本无法支持某些必要的功能. 109～115 行保存 C 运行时间环境中的若干重要的寄存器值(SS,SP,DS 和 ES),故以后在激活该 TSR 程序之前可立即恢复这些值. 下面,TSR 保存两个传送给它的参数:C 语言 TSR 进入点地址和 hotkey 特征码(117～120 行),使它们可为中断处理程序所用. 下面几行(127～130 行)使用了未写入文档的 DOS 功能 34h 以检索对于 indos 标志的双字指针. 如前所述,中断处理程序以后可通过此指针直接访问 indos,而无需重新发出中断.

下面一段代码初始化中断 09h,13h 和 28h 的处理程序. 首先保存旧值,使处理程序可链接到以前的程序 (132～148 行) 上. 然后,设置向量以指向汇编模块内的相应子程序(150～172 行). 注意,一旦安装了键盘中断处理程序 09h,则它任何键盘动作都可将其立即激活. 因此安装此向量之前,一定要确保完成所有必要的初始化.

tsr 的最后一段(174～214 行)计算将留在内存中的 16 字节段落(paragraph)的数量,调用 DOS 的结果并驻留功能 31h. 从 C 程序中计算出正确的段落数需要有点技巧. 图 7-2 给出了"小内存模式"C 程序所用的内存机构.

注意,数据堆栈及 heap 均包含在一个 64K 块中,它们可通过数据寄存器来寻址(SS 寄存器与 DS 寄存器具有相同的值). C 启动程序释放掉 heap 以上的所有内存,即使对于非常小的程序,也要分配掉整个 64K 的内存块(从而产生很大的 heap 区). 但一个 TSR 必须要减少内存的使用,因此在堆栈上方不远的某一点上释放内存,仅给 heap 留下最小的内存

量。

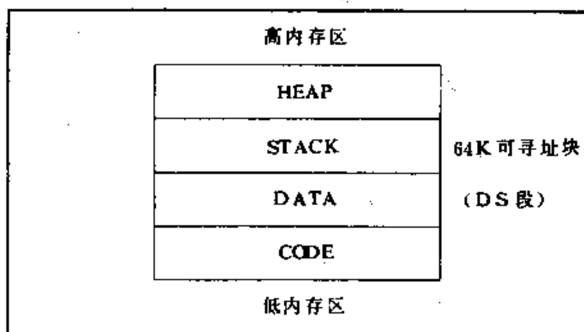


图 7-2 "小内存模式"C 程序的内存机构

这种技术由使用 C 库函数 sbrk 开始。此函数通常用于从 heap 分配内存的低层次方法。传送给它一个参数,该参数表示欲移去的当前 heap break 地址字节数(正的或负的),这里的所谓 break 地址是相对于第一个以前的 break 地址的数据段寄存器的偏移量,而不是 heap 空间中未分配内存的全部字节数。此函数为 heap 空间返回 64K 内存块的剩余部分,而 TSR 只保留由常数 HEAPSIZE 表示的 heap 空间量。为计算必须驻留在内存中的段落总数,tsr 应执行下述的步骤:

首先调用 _sbrk 根据 heap 所需要的字节数(由 HEAPSIZE 乘以 16 得到)在内存中向上移动 break 地址。如果 64K 内存块无法满足 heap 的内存要求,_sbrk 返回-1,tsr 向 C 程序返回一条出错信息。_sbrk 仍返回以前的 break 地址,并把它保存在变量 break_off 中(此地址通常直接在堆栈上方,除非以前的 C 代码已分配了 heap 内存)。注意,这步的执行仅要求保证有足够的自由内存可供 heap 使用,由 C 库程序维护的 berak 值必须以下列步骤恢复(174-185 行):

tsr 使用一个负数来调用_sbrk 从而得到步骤 1 的反效果,它将 heap break 地址恢复到其原始值,从而 C 分配函数开始根据 break_off 中包含的偏移量来分配 heap 空间.

接着 tsr 将 break_off 中包含的 break 偏移量变换为 16 字节段落,并将此值与 DS 寄存器中的值相加,得到的结果总数便是 heap 开始的段地址(195-201 行).

然后,tsr 从 heap 开始的段地址中减去程序段前缀开始的段地址(包含在全局 C 变量 _PSP 中)(204-206 行).

最后,tsr 将运行时间 heap 所需要的段落数加到常数 HEAPSIZE 中. 这个结果便是必须驻留在内存中的总段落数(209 行).

一旦计算出总的驻留代码和数据的大小,tsr 便调用 DOS功能31h以结束此过程,并将所请求的段落数驻留在内存中.

如前所述,C 程序应避免调用内存分配函数 malloc 及 calloc,它们从 heap 中分配内存,但现在情况是 heap 空间已很不足. 但有些 C 库函数(如 fprintf)亦从 heap 中分配内存,因此必须提供少量的 heap 空间. 如果 C 程序分配的 heap 内存超过了 DOS 为 TSR 保留的内存块,就显示出如下信息:

Memory Allocation Error ▲System Halted

解决办法是增加分配给 HEAPSIZE 的值,直到此信息不再显示.

tsr 的最后几行(216-218 行)仅当阻止 TSR 安装发生而出错时用于接受控制. 这几行恢复寄存器值并将错误返回给 C 程序,显示以适当的出错信息.

▲中断处理程序

该汇编模块包含三个中断处理程序:

用于硬件键盘中断(09h)的int09、用于 DOS 空闲中断(28h)的 int28 以及用于 BIOS 磁盘服务中断(13h)的 int13。在其他编写的 TSR 中还可能安装另一些中断的处理程序,例如时钟信号(clock tick)(08h)及软件键盘中断(16h)。

硬件中断处理程序 int09(273-311 行)用于检测何时按下 hotkey。任意时刻按下或释放某个键盘中断(16h)来检测 hotkey,该中断仅当程序在读键盘时产生。

int09 首先调用前面已安装的中断处理程序。从以前程序返回时,要执行一系列测试以确定 TSR 应否被激活。如果测试中任一项失败,则立即发出 iret 指令以返回到被中断的程序。实行测试有(279-306 行):

· 测试 busy 标志,查看该 TSR 是否已被活动(该 TSR 程序设计时是不允许递归调用的)。

· 如果 in_bios 标志等于 1,则 BIOS 磁盘服务中断 13h 当前正在活动,本中断处理程序返回。

· 接着,中断处理程序读地址 000h:0417h 处的一个字,查看 BIOS Shift 键标志,以确定 hotkey 组合键当前是否被按下。

· 如果通过双字指针 indos_ptr 访问的 indos 标志大于 0,则 DOS 不应被中断,故该处理程序简单地返回。

如果所有上述测试都通过,就调用过程 activate 以触发 TSR 主代码部分。注意,尽管按下 hotkey,但测试表明调用 DOS 不安全,程序自会忽视该请求并返回。

此实现的另一个简化特点是通过读 shift 键标志而不是读一个字符来检测 hotkey。虽然使用 shift 键而不使用字符键允许的可能组合更少一些,但省去了要处理所有键入的麻烦。例如,假如中断 09h 处理程序要检测 AltF10 hotkey,就

必须通过键盘缓冲区来搜索此键，如果找到了还得小心地去掉它以防它传送给另一个程序。检测 shift 键组合简单并且快捷，因为只需测试状态标志。

注意，在中断 16h 处理程序内是不能使用 shift 状态标志的。BIOS 中断 16h 服务在没有输入一个实际字符之前是不会返回控制的。如果按下了shift 键，BIOS便更新其状态标志，但并不将控制返回给调用程序，这样 TSR 便无法检测到已改变了的 shift 状态。

如前所述，安装中断处理程序 int28 (224–255 行)允许 TSR 在 DOS 提示符下被激活，即使此时 indos 标志为非 0。原因是发出空闲中断28h期间，中断 DOS 是安全的。因此，该中断处理程序执行与 int09 相同的一系列测试，只是不测试 indos 标志。

中断 13h 处理程序(258–270 行)用于不同的目的。在 BIOS 磁盘服务程序当前活动时，它为设置 in–bios 标志而截获该中断。注意，该处理程序使用指令 ret2 而不是 iret 来返回。这是因为此过程是作为 far 过程声明的，ret2 产生 far 返回并根据原始的 int 指令(其推入了标志)来恢复堆栈。这种方法省去了标志弹出并保存了当前标志值，由于 BIOS 用设置进位标志来通知出错，故该方法很重要(BIOS 代码本身便是以 ret2 返回的)。

▲激活子程序

一旦确定 TST 应被激活，则 int09 和 int28 便调用 activate(314–402 行)。函数 activate 的作用是保存当前的机器状态，为 C 建立运行时间环境，调用 C 语言 TSR 函数，然后恢复保存的机器状态。

函数 activate 首先将 busy 标志设置成 1，以阻止代码的

递归进入,然后把保存 SS 和 SP 的当前值切换到一个自用堆栈,再设置 SS 和 SP 到初始化代码保存的值。程序在保存当前机器状态以确保有足够的堆栈空间之前,先切换到自用堆栈。333--341 行保存所有当前的寄存器。

该程序然后从属于被中断程序的堆栈顶部保留 64 个字(344-351 行)。此措施为的是避免在 indos 标志设置时中断 MS-DOS(通过中断 28h,此时 DOS 处于空闲状态)而引起的堆栈破坏。注意,数字 64 是随便给的,但一般还是够大的。可试着去减小此数,如果在使用 DOS 功能 01h-0Ch 然后返回 DOS 之后出现了问题,则必须增加此值。

在 353-361 行处,activate 保存了旧的磁盘传送地址并将 DTA 设置为初始化代码所保留的 C 程序的值。在调用 C 语言 TSR 进入点之前,activate 亦将 DS 和 ES 设置成初始化代码保存的 C 的值,并使硬件中断生效。367 行最后将控制传送给 C 语言 TSR 函数。

从 C 函数返回时,activate 使中断失效并以下列次序恢复被中断程序的全部机器状态:

- 磁盘传送区 DTA
- 自堆栈顶部保存的 64 个字
- 寄存器值
- SS 和 SP(通过切换回原始堆栈)

在返回之前,将 busy 标志再设置为 0 (注意,此程序仍是不允许递归重进入的,因为在程序结束前,中断系统是无效的)。

7.2.3 待改进的若干功能

为了尽可能地使代码简化,若干有用的功能被丢掉了,其中包括 Ctrl--Break 和致命错误处理程序、重复安装的

阻止以及解除安装的措施等.

▲Ctrl－Break 和致命错误处理程序

为使程序更有生命力,应添加的一个重要功能是用于 Ctrl—Break(中断 23h)和致命错误(中断 24h)的中断处理程序. 如果实现了这些中断处理程序,则每次调用 activate 都必须保存旧的中断向量值并设置这两个向量以指向新的中断处理程序,然后在退出之前恢复原来的值.

▲防止重复安装

本书给出的 TSR 不能阻止代码的重复安装,但安装程序检测是否已安装过 TSR 是件很要紧的事情. 不能简单地用查看用于激活 TSR 之中断向量所指向的程序代码来确定,因为其他 TSR 也可能在后面又取代了这些向量.

解决的办法较复杂,例如可通过 DOS 分配的内存块来跟踪、搜索特别的识别标记. 较简单的办法是使用一个用户中断向量,如范围 60h 至 67h 之间. 使用一个用户中断向量,如范围 60h 至 67h 之间. 安装程序搜寻此范围内第一个具有 0 值的中断(可保证此中断未被其他程序所用), 然后将此向量设置为一个识别代码,或可赋给它在驻留代码中的一个标记的地址. 只要运行了初始化代码,便扫描向量 60h 至 67h 以查看识别标记,如果找到了,便使安装失败并返回一个出错代码.

▲解除安装的措施

TSR 的一个可选功能是采用从内存中除去代码并将内存返回给 DOS 的机构. 如果 TSR 解除安装,应使用 DOS 功能 49h 释放从程序段前缀开始的内存块以及包括环境的内存块.

使 TSR 能自行释放可能会产生问题,因为 TSR 必须是

最后一个安装到内存中的,否则在DOS维护的已分配内存块中会留下一个大的空隙.另外,后面安装的 TSR 可能已经保存了指向已释放代码的中断向量,如果此 TSR 使用保存的地址链接到前一个程序,将程序转入已释放掉的内存区就会造成严重问题.

释放并重新装人 TSR 最常见的办法是,使用一个独立的实用程序管理来自不同厂家的多个内存驻留程序,并以正确次序从内存中将它们释放. 目前我们可提供两个这样的实用程序 MRAK.COM 和 RELEASE.COM. MARK.COM 在打算释放的 TSR 装入内存之前运行,在内存中作个记号,以后在使用 RELEASE.COM 释放此 TSR 时,程序会自动根据上次由 MARK.COM 作的记号释放该 TSR 上方的全部内存空间. MARK 与 RELEASE 配对使用,次数不限,可用于标记与释放多个 TSR. 本章中的汇编模块源程序及经汇编后生成目标库程序、演示用的 C 源程序以及 MARK.COM 与 RELEASE.COM 实用程序已制作在一张盘上. 读者如需要,可来函索要.

联系地址:北京市西城区三里河四区六栋十号

联系人:夏东涛

第八章　扩充内存及其C语言接口

在设计高性能应用程序时，总要考虑程序的长度与执行速度的关系．速度快的程序往往需要使用更多的内存．例如，C的编译程序重点考虑的是优化程序的速度．而且要将一个较大的应用程序全部装入内存可大大提高效率，但比将覆盖模块存储在磁盘上的程序需要使用多得多的内存．另外，把满屏的视频数据存储在一个经初始化后的数组中，可产生非常快的显示，但它比一个按要求从磁盘文件中读屏幕数据的程序所用内存要多得多．

显然，应用程序可用的内存与高性能使用的可能性之间有直接关系．在DOS3.3版本下可用的内存只有640kb，这一点是对大应用程序的主要限制，尤其是在使用了内存驻留程序或是多任务操作系统环境(如Microsoft Windows)时，更是如此．

避开DOS内存限制的最好办法之一就是使用扩充内存规范EMS(Expanded Memory Specification)．由于这是Lotus、Intel和Microsoft三个公司合作开发的，因此也叫"LIM-EMS"．扩充内存卡按此标准，允许在DOS控制上用分页方式，可访问的内存空间为8MB．

注意，"expanded"(扩充)内存与"extended"(扩展)内存是有区别的．后者指的是由80286处理器可寻址16MB内存之高15MB(这种内存，只是在"保护(protected)"方式时是可寻址的；DOS运行于real方式，仿真8086寻址空间，因此

标准的 DOS 应用程序是不能直接访问"extended"内存的)。

除了标准的 EMS 外，还有一种增强型的 EMS 板，即 EEMS 板。它是由 AST Research，Quadram 和 Ashton-Tate 联合开发的，是EMS的一个扩充集。对遵循EMS规则的应用程序来讲，使用 EEMS 板及其控制软件时亦是兼容的。

本章所介绍的适用于 EMS3.2 版本。

8.1 扩充内存规范(EMS)概述

MS-DOS 工作在由 8086／88 可寻址的 1 兆字节内存中。这个地址空间的前 640KB 定义为用户内存。它安装在母板和内存卡上，可由操作系统、内存驻留及暂驻应用程序使用。640KB 以上保留的内存用于多种目的，如视频适配器，磁盘控制器，网络控制器以及系统 ROM BIOS(参看内存图)。

在适配板上安装一些内存芯片是非常容易的，但要对 640K 以上的内存寻址并非简单，要考虑到不能与操作系统所保留的空间发生冲突，不能超过 8088／86 和以仿真方式运行的增强型处理机固有的 1 兆字节的限制。实际上，在一台计算机系统中，不是 640K 以上内存的所有部分都要被使用，至少可以找到 64K 相连内存没有被使用(这个未用的内存地址取决于已存在的其他选件，并且是在装扩充内存卡时，通过设置开关指定的)。64K 的相连内存叫做"页面结构(page frame)"，用它作为一个窗口，可访问扩充内存的 8 兆字节空间。在页面结构中可通过标准的读、写操作获得对扩充内存的存取(访问)。然而，通过一个给定地址访问扩充内存中的实际地址取决于程序已请求过的当前的映照关系(mapping)。对页面结构内部内存地址的读／写请求自动地由硬件重新引向当前映照关系指定的扩充内存区域。基本的

方法如下:

· 应用程序保留若干扩充内存的16K逻辑页.根据已安装扩充内存的数量以及事先保留了多少页来确定最大页数.

· 64K 的页面结构分成 4 组相连 16K 的物理页面，这些页面是可以直接寻址的．程序可指定扩充内存的哪 4 个逻辑页面映照成页面结构里的 4 个物理页面．这样，在一个给定的时间内，应用程序就可以对扩充内存的 64K 空间进行直接读／写访问了．

· 为了能够访问其它扩充内存空间，程序必须请求其他是映照关系，作为把其他的扩充内存逻辑页面转换成 64K 页面结构的窗口。这个过程叫做"切换(Bank Switching)"或"分页(paging)"。

当前页面结构映照关系是由在扩充内存板上设置硬件页面寄存器(称为 bankswitch)来确定的．应用程序不能直接访问这些寄存器,而只能通过名叫 EMM (Expanded Memory Manager) 随适配器板提供的扩充内存管理程序对扩充内存进行控制．使用符合规范的 EMM 而不是直接由硬件控制，可防止可能出现的兼容性问题．EMM 是作为一个字符设备驱动程序驻留在内存中．任何类似的设备驱动程序都不能与其同时运行.因为EMM不是通过DOS文件系统进行访问的，而是直接用的中断 67h(注意,由于EMM没有链接到其他中断处理程序上,故其他程序不可再使用 67h).

EMM 提供了使用扩充内存的所有功能．这些服务功能是通过在寄存器之间交换值，调用中断 67h 来提供的．寄存器的使用方法类似于请求 MS-DOS 和 BIOS 完成某项功能的方法．但在所有情况下，寄存器 AH 均被程序用于指定功能代码，被 EMM 用于返回一个状态码．下面是分类

的 EMM 功能:

功能	用途
▲由临时应用程序使用:	
1	获得 EMS 状态
2	获得"分页结构"的段地址
3	获得未分配逻辑页面计数
4	分配逻辑页面地址
5	将逻辑页面转换为物理页面
6	释放逻辑页面
7	获得 EMM 版本号
▲由内存驻留程序使用:	
8	保存当前页面变映照关系
9	恢复已存的页面映照关系
▲保留	
10	
11	
▲由低层内存管理程序使用:	
12	获得 EMM 句柄数
13	获得分配给指定句柄的页面号
14	获得分配给所有句柄的页面号
▲由多任务系统使用:	
15	向一个程序陈列或从这个程序中设置或获得页面映照关系

8.2 EMS 的 C 语言程序接口

下面是一组汇编功能函数，可以在一个 C 程序中调用它，这组功能函数构成了一个对扩充内存的高层接口。这些功能函数组成一个完整的函数集:在各单独程序之间共享

数据，每个模块有自己的一组错误信息。因此，这个完整的文件应与某个 C 程序相连接，而不应与其他直接访问 EMM 的函数联用。这些函数的设计思路是:

· 提供一个完整的功能函数集，用于管理扩充内存，使C可以容易地访问扩充内存。这些功能函数对一般的用途是足够了。

· 用尽可能少的功能函数，使其与 C 库中的标准内存分配函数相类似。

· 使调用 C 程序时涉及到的低层细节对用户透明。这些细节包括确定扩充内存是否存在及管理 EMM 句柄(分配一个扩充内存块时，EMM 返回一个句柄，用来作为在随后发生的函数功能调用的一个标识符。但 C 程序都不必知道 EMM 句柄)。

· 提供一个简单的错误码集，以排除一些可能发生的错误。如传一个无效的功能号或用了无效的 EMM 句柄等。

C 接口所做的简化之一是消除了多重的内存分配。EMM 允许一个进程多次调用“分配功能函数(04)”，以获得几组扩充内存块，每一组有它的标识句柄。通常，程序只是在开头部分分配一个扩充内存块，在结束时释放掉它。为简便起见，接口只存储一个句柄，如果程序第二次用 calloc 分配内存，此接口程序将返回一个错误码(05，说明 EMS 内存已被分配)。

```
 1:    page 60,130
 2:
 3:    ;Figure,    10
 4:    ;File:      EMS.ASM
 5:
 6:    ;* C Interface to LIM/EMS Expanded Memory.
 7:
 8:    public      emm_name, emm_alloc, emm_handle
 9:    public      _emmavail, a01, a02, a03, a04
10:    public      _ealloc, b00, b01, b02, b03
11:    public      _efree, c01, c02, c03
12:    public      _emap, d01, d02, d03, d04, d05
13:    public      _esave, e01, e02, e03
14:    public      _erestore, f01, f02, f03
15:
```

```
16:  _data        segment word public 'DATA'
17:
18:  extrn        _ems_err:word               ;Global error code.
19:
20:  emm_name     db      'EMMXXXX0'          ;Expected device name.
21:  ems_alloc    db      0                   ;Flag => EMS memory allocated/handle valid.
22:  ems_handle   dw      ?                   ;Handle for EMM.
23:
24:  _data        ends
25:
26:  dgroup       group   _data
27:  assume       cs:_text, ds:dgroup
28:
29:  _text        segment byte public 'CODE'
30:
31:  frame        equ     [bp]
32:
33:
34:  _emsavail proc   near                    ;Returns number of free EMS pages.
35:
36:  comment      /*
37:               This function returns the number of free 16K pages of LIM-EMS
38:               expanded memory.  If 0 is returned, 'ems_err' contains one of the
39:               following error codes:
40:
41:               01      Expanded memory not installed.
42:               02      No free expanded memory pages available.
43:               03      Software malfunction.
44:               04      Hardware malfunction.
45:
46:               int emsavail ();
47:
48:               */
49:
50:               push    di
51:               push    si
52:               push    es
53:                                           ;Test for presence of expanded memory.
54:               mov     ah, 35h             ;Get vector to EMM.
55:               mov     al, 67h
56:               int     21h
57:                                           ;Test for correct device name.
58:               mov     di, 000Ah           ;Offset of device name in header.
59:                                           ;Compare with expected name:
60:               mov     si, offset dgroup:emm_name
61:               mov     cx, 8               ;Compare 8 bytes.
62:               cld
63:               repe    cmpsb
64:               je      a01                 ;EMM present.
65:               xor     ax, ax              ;EMM not present, therefore 0 pages.
66:               mov     _ems_err, 01        ;Set error code.
67:               jmp     a04                 ;Go to end.
68:  a01:                                     ;Call EMM for number of free pages.
69:               mov     ah, 42h
70:               int     67h                 ;EMM service interrupt.
71:               cmp     ah, 0               ;Test for EMM error.
72:               je      a02                 ;No error.
73:               call    err_rtn             ;Set EMM error code.
74:               xor     ax, ax              ;EMM error, therefore 0 pages.
75:               jmp     a04                 ;Go to end.
76:  a02:
77:               mov     ax, bx              ;BX contains free pages.
78:               cmp     bx, 0               ;Test for 0 pages free.
79:               jne     a03
80:               mov     _ems_err, 02        ;Set error code.
81:               jmp     a04                 ;Go to end.
82:  a03:         mov     _ems_err, 00        ;Set error code: no errors.
83:  a04:
84:               pop     es
85:               pop     si
86:               pop     di
87:               ret                         ;Return to C program
88:
89:  _emsavail endp
```

```
90:
91:
92:  _ealloc   proc    near                    ;Allocates LIM-EMS expanded memory.
93:
94:  comment   /*
95:            This function allocates the requested number of 16K pages of
96:            expanded memory.  It returns a far pointer to the beginning of
97:            the page frame where memory will be mapped.  If NULL (= 0) is
98:            returned, then 'ems_err' contains one of the following error codes:
99:
100:           03      Software malfunction.
101:           04      Hardware malfunction.
102:           05      EMS memory already allocated by this process.
103:           06      No EMM handles free.
104:           07      Insufficient EMS pages to fill request.
105:           08      Attempt to allocate 0 pages.
106:
107:           char far * ealloc (pages);
108:           int pages;                       Requested number of EMS pages.
109:
110:           */
111:
112:           assume cs:_text
113:
114: bframe    struc                            ;Stack frame template.
115: bbptr     dw      ?                        ;Base pointer.
116: bret_ad   dw      ?                        ;Return address.
117: bnum      dw      ?                        ;Number of pages requested.
118: bframe    ends
119:
120:           push    bp
121:           mov     bp, sp
122:
123:           cmp     emm_alloc, 0             ;Test if memory already allocated.
124:           je      b00
125:           xor     dx, dx                   ;Already allocated, return NULL.
126:           mov     _ems_err, 05             ;Set error code.
127:           jmp     b03                      ;Go to end.
128: b00:
129:           mov     ah, 43h                  ;Call EMM to allocate pages.
130:           mov     bx, frame.bnum           ;Load requested number of pages.
131:           int     67h
132:           cmp     ah, 0                    ;Test for EMM error.
133:           je      b01                      ;No error.
134:           call    err_rtn                  ;Set EMM error code.
135:           xor     dx, dx                   ;EMM error, return NULL.
136:           jmp     b03                      ;Go to end.
137: b01:                                       ;Allocation successful.
138:           mov     emm_alloc, 1             ;Set flag => mem.allocated/valid handle.
139:           mov     emm_handle, dx           ;Save EMM handle.
140:
141:           mov     ah, 41h                  ;Call EMM to get frame address.
142:           int     67h
143:           cmp     ah, 0                    ;Test for EMM error.
144:           je      b02                      ;No error.
145:           call    err_rtn                  ;Set EMM error code.
146:           xor     dx, dx                   ;Error, return NULL.
147:           jmp     b03                      ;Go to end.
148: b02:      mov     dx, bx                   ;Return segment in DX.
149:           mov     _ems_err, 00             ;Set error code: no errors.
150: b03:
151:           xor     ax, ax                   ;AX returns offset, always 0.
152:           pop     bp
153:           ret                              ;Return to C program.
154:
155: _ealloc   endp
156:
157:
158: _efree    proc    near                     ;Frees expanded memory.
159:
160: comment   /*
161:           This function frees all LIM-EMS memory allocated to the current
162:           process.  It returns 0 if no error occurred, otherwise -1 is
```

```
163:            returned and 'ems_err' contains one of the following error codes:
164:
165:            03      Software malfunction.
166:            04      Hardware malfunction.
167:            09      EMS memory not allocated.
168:            10      'esave' called without subsequent 'erestore'.
169:
170:            int efree ();
171:
172:            */
173:
174:            cmp     emm_alloc, 0            ;Test if memory has been allocated.
175:            jne     c01
176:            mov     ax, -1                  ;Error: no memory to deallocate!
177:            mov     _ems_err, 09            ;Set error code.
178:            jmp     c03                     ;Go to end.
179:    c01:
180:            mov     ah, 45h                 ;Call EMM to deallocate pages.
181:            mov     dx, emm_handle
182:            int     67h
183:            cmp     ah, 0                   ;Test for EMM error.
184:            je      c02
185:            call    err_rtn                 ;Set EMM error code.
186:            mov     ax, -1                  ;Error on deallocation.
187:            jmp     c03                     ;Go to end.
188:    c02:
189:            mov     emm_alloc, 0            ;Reset allocation flag.
190:            xor     ax, ax                  ;Return value for no errors.
191:            mov     _ems_err, 00            ;Set error code: no errors.
192:    c03:
193:            ret                             ;Return to C program.
194:
195:    _efree  endp
196:
197:
198:    _emap   proc    near                    ;Maps EMS pages.
199:
200:    comment /*
201:            This function maps logical EMS pages onto the four physical pages
202:            of the EMS page frame.  It returns 0 if no error occurred.
203:            If -1 is returned, 'ems_err' contains one of the following error
204:            codes:
205:
206:            03      Software malfunction.
207:            04      Hardware malfunction.
208:            09      EMS memory not allocated.
209:            11      Logical page out of range allocated to process.
210:
211:            int emap (p0, p1, p2, p3);
212:            int p0;                         logical page for physical page 0
213:            int p1;                         logical page for physical page 1
214:            int p2;                         logical page for physical page 2
215:            int p3;                         logical page for physical page 3
216:
217:            Note: p0 through p3 contain the logical page numbers to be mapped
218:            onto physical pages 0 through 3.  If any parameter = -1, then NO
219:            mapping is done to that physical page.
220:
221:            */
222:
223:    dframe  struc                           ;Stack frame template.
224:    dbptr   dw      ?                       ;Base pointer.
225:    dret_ad dw      ?                       ;Return address.
226:    log0    dw      ?                       ;Logical page -> physical page 0.
227:    log1    dw      ?                       ;Logical page -> physical page 1.
228:    log2    dw      ?                       ;Logical page -> physical page 2.
229:    log3    dw      ?                       ;Logical page -> physical page 3.
230:    dframe  ends
231:
232:            push    bp
233:            mov     bp, sp
234:
235:            push    si
236:
237:            cmp     emm_alloc, 0            ;Test if memory has been allocated.
238:            jne     d01
```

```
239:            mov     ax, -1                  ;Error, no memory allocated!
240:            mov     _ems_err, 09            ;Set error code.
241:            jmp     d05
242:    d01:                                    ;Map 4 logical pages -> 4 physical pages.
243:            xor     cl, cl                  ;Physical page counter.
244:            mov     si, 0                   ;Index for accessing all 4 parameters.
245:    d02:
246:            mov     ah, 44h                 ;Call EMM to map page.
247:            mov     al, cl                  ;Physical page -> AL.
248:            mov     bx, ss:[bp+log0+si] ;Logical page -> BX.
249:            cmp     bx, -1                  ;Test for dummy parameter.
250:            je      d03                     ;If parm = -1, go on to next one.
251:            mov     dx, emm_handle
252:            int     67h
253:            cmp     ah, 0                   ;Test for EMM error.
254:            je      d03
255:            call    err_rtn                 ;Set EMM error code.
256:            mov     ax, -1                  ;EMM error.
257:            jmp     d05                     ;Bail out.
258:    d03:
259:            inc     si                      ;Move to next parameter.
260:            inc     si
261:            inc     cl                      ;Next physical page.
262:            cmp     cl, 4                   ;Test for last physical page.
263:            jge     d04                     ;Last physical page, quit loop.
264:            jmp     d02                     ;Back for more mappings.
265:    d04:
266:            xor     ax, ax                  ;No error.
267:            mov     _ems_err, 00            ;Set error code: no errors.
268:    d05:
269:            pop     si
270:            pop     bp
271:            ret                             ;Return to C program.
272:
273:    _emap   endp
274:
275:
276:    _esave  proc    near                    ;Save page map.
277:
278:    comment /*
279:            This function saves the current EMS state -- the values of
280:            registers on all EMS boards.  It returns 0 if successful.  If an
281:            error occurred, -1 is returned and 'ems_err' is set to one of the
282:            following values:
283:
284:            03      Software malfunction.
285:            04      Hardware malfunction.
286:            09      EMS memory not allocated.
287:            12      No room in save area.
288:            13      'Save' already called by this process.
289:
290:            int esave ();
291:
292:            */
293:
294:            cmp     emm_alloc, 0            ;Test if memory has been allocated.
295:            jne     e01
296:            mov     ax, -1                  ;Error, no memory allocated!
297:            mov     _ems_err, 09            ;Set error code.
298:            jmp     e03                     ;Go to end.
299:    e01:                                    ;Call EEM to save page map.
300:            mov     ah, 47h
301:            mov     dx, emm_handle
302:            int     67h
303:            cmp     ah, 0                   ;Test for EMM error.
304:            je      e02
305:            call    err_rtn                 ;Set EMM error code.
306:            mov     ax, -1                  ;EMM error.
307:            jmp     e03                     ;Go to end.
308:    e02:
309:            xor     ax, ax                  ;No error.
310:            mov     _ems_err, 00            ;Set error code: no errors.
311:    e03:    ret                             ;Return to C program.
312:
313:    _esave  endp
```

```
314:
315:
316:  _erestore  proc    near                    ;Restores page map saved by 'esave'.
317:
318:  comment    /*
319:             This function restores the EMS state -- the values of registers
320:             on all EMS boards -- which was previously saved by 'esave.'  It
321:             returns 0 if successful.  If an error occurred, -1 is returned
322:             and 'ems_err' is set to one of the following values:
323:
324:             03      Software malfunction.
325:             04      Hardware malfunction.
326:             09      EMS memory not allocated.
327:             14      'erestore' called without prior 'esave'.
328:
329:             int erestore ();
330:
331:             */
332:
333:             cmp     emm_alloc, 0            ;Test if memory has been allocated.
334:             jne     f01
335:             mov     ax, -1                  ;Error, no memory allocated!
336:             mov     _ems_err, 09            ;Set error code.
337:             jmp     f03                     ;Go to end.
338:  f01:                                       ;Call EEM to restore page map.
339:             mov     ah, 48h
340:             mov     dx, emm_handle
341:             int     67h
342:             cmp     ah, 0                   ;Test for EMM error.
343:             je      f02
344:             call    err_rtn                 ;Set EMM error code.
345:             mov     ax, -1                  ;EMM error.
346:             jmp     f03                     ;Go to end.
347:  f02:
348:             xor     ax, ax                  ;No error.
349:             mov     _ems_err, 00            ;Set error code: no errors.
350:  f03:       ret
351:
352:  _erestore  endp
353:
354:
355:  err_table  db      03                      ;80h  Software malfunction.
356:             db      04                      ;81h  Hardware malfunction.
357:             db      15                      ;82h  Unidentified error.
358:             db      15                      ;83h  Unidentified error.
359:             db      15                      ;84h  Unidentified error.
360:             db      06                      ;85h  No free handles.
361:             db      10                      ;86h  'Esave', no 'Erestore'.
362:             db      07                      ;87h  Insufficient EMS pages.
363:             db      07                      ;88h  Insufficient EMS pages.
364:             db      08                      ;89h  0 page request.
365:             db      11                      ;8Ah  Log. page out of range.
366:             db      15                      ;8Bh  Unidentified error.
367:             db      12                      ;8Ch  No save room.
368:             db      13                      ;8Dh  'Save' already called.
369:             db      14                      ;8Eh  'Erestore' w/out 'esave'.
370:
371:  MAX_ERR    equ     $ - err_table           ;Largest error number.
372:
373:
374:  err_rtn    proc    near                    ;Sets 'ems_err' to current EMM error.
375:
376:  comment    /*
377:             This function translates an EMM error into the appropriate error
378:             code for the C Interface, and assigns the value to the global error
379:             variable 'ems_err'.   It is for internal use only and is not to be
380:             called from C.
381:
382:             */
383:
384:             push    ax
385:             push    bx
386:
```

```
387:            cmp     ah, 0               ;Test for "no error."
388:            jne     g00
389:            mov     _ems_err, 0         ;Assign code for "no error."
390:            jmp     g02                 ;Go to exit.
391:
392:  g00:      xor     bh, bh              ;0 upper order byte of BX.
393:            mov     bl, ah              ;Place error code in BL.
394:            and     bl, 7fh             ;Mask off upper bit of error code.
395:            cmp     bl, MAX_ERR         ;Test if error code within defined range.
396:            jle     g01                 ;Range OK.
397:            mov     _ems_err, 15        ;Unidentified error.
398:            jmp     g02                 ;Go to end.
399:  g01:                                  ;Use BX as index to error table.
400:            mov     al, cs:err_table [bx]
401:            cbw                         ;'ems_err' is a word.
402:            mov     _ems_err, ax        ;Assign 'ems_err'.
403:  g02:
404:            pop     bx
405:            pop     ax
406:            ret
407:
408:  err_rtn   endp
409:
410:
411:  _text     ends
412:            end
413:
414:
```

上面是C程序使用扩充内存的接口。

8.2.1 错误码说明

所有 EMM 功能每次都把状态码返回到 AH 寄存器中。出于两种原因，图 8-1 的接口程序提供自己的一组错误码。第一，因为有些错误码 EMM 没有提供，如，"没有安装扩充内存"。一个好的接口程序可以避免一些 EMM 发生的错误，并可省掉一些相对应的错误码，如"无效的 EMM 句柄(83h)"和"无效的功能码(84h)"。

第二，在错误出现时(例如 0，NULL,或-1),图 8-2 的每一个功能函数都返回一个指定的值给 C 程序，并送一个错误码给外部变量 ems-err。这个变量是在 C 程序中定义的。在 C 程序检测发生错误时，要检查这个全局变量以准确地确定错误。在每个功能函数的起始注释部分都定义了一些可能由功能返回的错误码。完整的接口错误码和相对应的 EMM 错误由表 8-1说明。

错误分为两类:

(1)由接口本身检测出的错误，例如"扩充内存没有安

装”或“EMS内存已由这个过程分配过了”。在这种情况下，功能函数简单地送适当的错误码给 ems_err。

(2)由 EMM 检测出的错误．在 EMM 功能调用时，发现返回到寄存器 AH 中的是非零值．这种情况，接口功能函数调用 err_rtn(374 行)，将 EMM 错误转换成适当的接口错误，然后指定一个值给 ems_err．这个转换是用从 355 行开始的变址错误表(error_table)来完成的．注意，AH 中所有返回的 EMM 状态码的高位被设置(除非“无错”时，它是 00)；在用一个值作为标引放入错误表之前，err_rtn 必须屏蔽掉高位。

表 8.1 C 接口程序的错误码

	接口错误码	相应的 EMM 错误
00	无错	
01	未安装扩充内存	
02	没有未用的扩充内存页面	
03	软件故障	80
04	硬件故障	81
05	EMS 内存已由这个过程分配了	
06	无 EMM 句柄释放	85
07	EMS 页面不够使用	87／88
08	分配 0 页面	89
09	没有分配 EMS 内存	
10	esave 后面没有调用 erestore	86
11	逻辑页面超出范围	8A
12	在保存的空间中没有余剩单元了	8C
13	过程已调用了 save	8D
14	erestore 没有在 esave 之前调用	8E
15	无标识错误	

8.2.2 接口功能函数

这部分介绍 C 接口提供的每个功能函数。

▲ememavail(34～89 行)

ememavail 将 EMM 逻辑页面号返给程序. 可能分配给 emm_err 的错误码有 00，01,02,03 和 04.

- 没有安装 EMM(emm_err＝01)
- 已分配了扩充内存的所有页面地址(emm_err＝02)
- 软或硬件出错(emm_err＝03 或 04)

ememavail 包括两个主要部分:第一是检测部分，确定扩充内存是否安装上了(54～67 行). 实现它有两种方法. 一种方法是把 EMM 作为一个设备(是作为一个字符设备驱动程序安装的)打开. 这种方法不能由内存驻留代码程序(如设备驱动程序)使用，因为设备驱动程序不能随便调用 DOS. 第二种方法是用中断向量 67h；如果 EMM 已安装，这个向量的段部分就会含有 EMM device header 的段地址，可在偏量0Ab处找到设备名EMMXXXXO.这种方法任何程序都可使用.后者比前者简单.ememavail使用的就是第二种方法。

第二部分，如果确定扩充内存已安装，调用功能 3 来确定空间逻辑页面的号码.

EMM 功能 3 使用如下的约定:

入口:	AH	42h
返回:	AH	错误码
	BX	空闭页面号
	DX	扩充内存的所有页面号

▲ealloc(92～155 行)

ealloc 分配扩充内存到调用处理程序. 它得到一个整数参数，此数包含逻辑页面的请求号，并将一个远程(far)指示字返回到扩充内存“页面结构”的第一个字节. 返回值

NULL(=0)指出一个错误；可能分配给 ems_err 的错误码有 00,03,04,05,06,07 和 08。

ealloc 运行分四步。第一步检查标志 emm_alloc,看当前是否将扩充内存分配到调用程序了。如果已分,就不再分了，除非紧接着要调用 efree(因为它是释放 emm_alloc)。

第二步(129～136行)调用 EMM 功能 4 实现分配：

EMM 功能 4: 分配逻辑页面

入口:	AH	43h
	BX	分配逻辑页面号
返回:	AH	错误码
	DX	EMM 句柄

第三步，一旦扩充内存成功地分配了，标志 emm_alloc 置为 1，句柄存入变量 emm-handle，(138,139 行)。emm_alloc 等于 1 时，其他接口功能函数就知道 emm_handle 保存了一个有效句柄，可将它安全地送到 EMM。

最后一步(141～147 行)调用 EMM 功能 2，获得扩充内存"页面结构"的段地址。

EMM 功能 2: 获得"页面结构"的段地址

入口:	AH	41h
返回:	AH	错误码
	BX	"页面结构"的段地址

注意，这个功能与 C 接口是在寄存器 DX:AX 中放一个段:偏移量的值将一个远程指示字返回的。在这种情况下，段是从 EMM 中获得的，偏移量为 0。将 NULL 指示字返回，DX 和 AX 为 0。

▲efree(158～195 行)

功能 efree 释放所有的由 ealloc 分配的扩充内存页面。无错时返 0，出错时返-1。可能分配给 ems-err 的错误码有 00,03,04,09 和 10。

efree 主要完成三个任务。第一(174～178 行),检查全局标志 emm__alloc 以确认 alloc 已被调用和已获得一个有效的句柄。

第二(180～187 行)，调用 EMM 功能 6 释放内存:

EMM 功能 6:释放已分配的逻辑页面

入口:	AH	45h
	DX	EMM 句柄
返回:	AH	错误码

第三步(189 行)，标志 ems _ alloc 置为 0。这个值将告诉 ealloc 现在可以重新分配内存了，也告诉其他的功能(如 emap),emm _ handle 不包含有效的句柄了。

▲emap(198～273 行)

emap 只是一个接口功能，在程序执行过程中多次被调用，用来将扩充内存的逻辑页面转换为"页面结构"的 4 个物理页面。它接收 4 个整数参数，参数含有逻辑页面号。如果参数中有-1，那么相应的物理页面就不改变了，例如:

emap (3,-1,4,5);

转换逻辑页面 3 为物理页面 0，逻辑页面 4 为物理页面 2，逻辑页面 5 为物理页面 3，物理页面 1 不改变。

如无错，功能返回 0，有错返回-1。可能分配给 ems _ err 的错误码有 00,03,04,09 和 11。

emap 完成两个任务。第一(237～241 行),检查标志 emm _ alloc，以确认是否已成功地将扩充内存分配给程序。程序的其余部分有一个循环，调用 EMM 功能 5。

EMM 功能 5:转换逻辑页面号为物理页号

入口:	AH	44h
	AL	物理页号
	BX	逻辑页号
	DX	EMM 句柄
返回:	AH	错误码

▲esave(276~313 行)

esave 使 EMM 把当前变址表(所有 EMS 寄存器的值)存在一个与传送的句柄有联系的内部空间里。用相同的句柄来调用 erestore 时，保存的值恢复到寄存器中。这一对功能对内存驻留程序非常有用，因为这些程序可以中断其他的正在使用扩充内存的程序。在改变变址表前，用 esave 保存当前的状态，以后在返回到中断的程序之前时再用 erestore 恢复它。

注意，在调用 efree 时(文件句柄不再有效以后)，如果 esave 已经被调用，而 erestore 没有调用，就会返回一个错误码 10，因为在 efree(这时 EMM 句柄仍有效)之前必须先调用 erestore。

如果成功，功能返回 0，出错返回-1。可能分配给 ems_err 的错误码有 00,03,04,09,12 和 13。

esave 完成两个任务。第一(294~298 行)，检查emm_alloc，证实扩充内存已成功地被分配和 emm_handle 已有一个有效的句柄。其余的部分就是调用 EMM 功能 8，使 EMM 保存页面变址表。

EMM 功能 8: 保存当前页面变址表

入口:	AH	47h
	DX	EMM 句柄

返回: AH 错误码

▲erestore(316～352 行)

erestore 和 esave 相对应．恢复由 esave 保存的页面变址表．如果成功，返回 0，出错返回－1．可能分配给 emm_err 的错误码有 00,03,04,09,14．

与 esave 相同,erestore 也完成两个任务．333～337 行完成检查 emm_alloc 的第一个任务．第二，调用 EMM 功能 9，恢复页面变址表:

EMM 功能 9:恢复已存的页面变址表

入口:	AH	48h
	DX	EMM句柄，与存时用的相同
返回:	AH	错误码

8.2.3 可实现的功能增强

可将一个附加的功能加到 C 接口，即返回被安装的 EMM 版本号．如果应用程序要知道正在使用的 EMM 与其是否兼容，可用 EMM 功能 7．

EMM 功能 7: 获得 EMM 版本号

入口:	AH	46h
返回:	AH	错误码
	AL	版本号:高数四位包括十进制小数点之前的数，低数四位包括十进制小数点之后的数．

8.3 从 C 程序中使用扩充内存

8.3.1 临时应用程序

下面是一个典型的从临时应用程序中使用扩充内存的例子．这个程序在完成运行后，放弃所占用的内存．下面

介绍它完成任务的步骤。

首先，用功能 ememavail 来确定扩充内存是否被安装了。如果已安装，再确定能够分配的未用页面数。如果没有足够的扩充内存可用，demo 程序退出。

```
/*
    Figure:   10.2
    File:     EMSTEST.C

    A C Program demonstrating the use of the LIM-EMS C interface

    This program must be compiled using the COMPACT memory model, and must
    be linked with the assembler EMS interface EMS.ASM (Figure 10.1).
    To generate program from EMSTEST.C and EMS.ASM:

        MSC /AC EMSTEST.C             use COMPACT memory model
        MASM EMS;
        LINK EMSTEST+EMS;
*/

#include <stdio.h>
#include <string.h>
                                /* Assembler EMS Interface functions    */
int ememavail (void);           /* (listed in Figure 10.1).             */
char far * ealloc (int);
int efree (void);
int emap (int, int, int, int);
int esave (void);
int erestore (void);

void exit (int);                /* Other functions.                     */
char *err_desc (int);

int ems_err = 0;                /* Global EMS error code.               */


#define PAGE0 0                 /* Offsets of physical pages in EMS     */
#define PAGE1 0x4000            /* page frame.                          */
#define PAGE2 0x8000
#define PAGE3 0xC000

void main ()
    {
    char *emsbase;              /* Pointer to base of page frame.       */

    /***********************************************************************
        (1)  Test if sufficient EMS memory is available to run program.
    ***********************************************************************/

    if (ememavail () < 4)
        {
        if (ems_err == 0)           /* No error, but memory insufficient.   */
            printf ("insufficient EMS memory to run demo\n");
        else                        /* 0 pages available, or error.         */
            printf ("%s\n", err_desc (ems_err));
        exit (1);                   /* Bail out with errorlevel 1.          */
        }

    /***********************************************************************
        (2)  Allocate EMS memory.
    ***********************************************************************/

                                /* Allocate 4 logical pages of EMS      */
                                /* memory and assign pointer to base of */
                                /* page frame.                          */
    if ((emsbase = ealloc (4)) == NULL)
        {
```

```
        printf ("%s\n", err_desc (ems_err));
        exit (1);                    /* Bail out with errorlevel 1.          */
        }

/****************************************************************************
        (3)  Access EMS memory.
****************************************************************************/

                                /* Map logical pages to physical pages.  */
                                /* Maps first 4 logical pages to first    */
                                /* 4 physical pages.                      */
    if (emap (0, 1, 2, 3) == -1)
        {
        printf ("%s\n", err_desc (ems_err));
        exit (1);                    /* Bail out with errorlevel 1.          */
        }

                                /* Write a string to each page.           */
    strcpy (emsbase+PAGE0, "logical page zero");
    strcpy (emsbase+PAGE1, "logical page one");
    strcpy (emsbase+PAGE2, "logical page two");
    strcpy (emsbase+PAGE3, "logical page three");

                                /* Remap logical pages to physical pages.*/
    if (emap (3, 2, 1, 0) == -1)
        {
        printf ("%s\n", err_desc (ems_err));
        exit (1);                    /* Bail out with errorlevel 1.          */
        }

                                /* Print contents of 4 physical pages.    */
    printf ("%s\n",emsbase+PAGE0);
    printf ("%s\n",emsbase+PAGE1);
    printf ("%s\n",emsbase+PAGE2);
    printf ("%s\n",emsbase+PAGE3);

/****************************************************************************
        (4)  Deallocate EMS memory.
****************************************************************************/

    if (efree () == -1)
        {
        printf ("%s\n", err_desc (ems_err));
        exit (1);                    /* Bail out with errorlevel 1.          */

        }

    } /* end main */

char *err_desc (code)       /* Returns a pointer to string containing the      */
int code;                   /* description of error corresponding to 'code'.   */
    {
    static char *err_list [] =
        {
        "No error.",
        "Expanded memory not installed.",
        "No free expanded memory pages.",
        "Software malfunction.",
        "Hardware malfunction.",
        "EMS memory already allocated by this process.",
        "No EMM handles free.",
        "Insufficient EMS pages to fill request.",
        "Attempt to allocate 0 pages.",
        "EMS memory not allocated.",
        "'esave' called without subsequent 'erestore'.",
        "Logical page out of range allocated to process.",
        "No room in save area.",
        "'Save' already called by process.",
        "'erestore' called without prior 'esave.'",
        "Unidentified error."
        };
```

```
return (err_list [code]);
} /* end print_err */
```

上面是使用扩充内存接口的说明性C程序。

第二步，调用 calloc 分配所需要的扩充内存页面数。与标准的 C 内存分配功能不同(如 malloc，在程序中要反复地调用它)，扩充内存只需用一个请求分配所有需要的页面，然后在程序结束时，再用一个功能调用释放所有的扩充内存。

第三步，首先调用 emap 将逻辑页面映照到"页面结构"，然后用由 calloc 返回的指针，加上需要的偏移量，对在"页面结构"中的地址执行普通的(far)读／写操作来访问扩充内存。emap 是程序运行过程中唯一应被反复调用的扩充内存函数。对扩充内存的典型访问包括:一系列映照、读／写、重新映照的循环。

注意，扩充内存可用于存储数据和代码，只有程序堆栈不能存在扩充内存里。DOS EXEC 功能 48h 不能将程序装入"页面结构"占有的内存空间，因此必须由程序本身将代码装入扩充内存。

demo 程序举例说明了映照、读／写和重新映照这三个过程。首先将逻辑页面 0～3 映照成物理页面 0～3，然后将一串有标志的字符写到每个逻辑页面的开始处，再调用 emap 重新把逻辑页面 0～3 映照为 4 个物理页面，这次按相反的次序进行。最后从每个物理页面的开始处将有标志的字符串打出来，顺序是 3,2,1,0。这说明逻辑页面在内存中已被反过来了。

最后一步是调用 cfree 释放扩充内存。注意，在程序终

止时 EMS 内存不能自动地释放。为了使其他应用程序能够使用扩充内存，在程序结束时一定要调用 efree 功能。

C demo 程序处理错误是打印相应的错误信息，用全局错误码 ems_err 作为索引放在由功能 err_desc 保留的错误信息表里，然后错误级设置为 1，终止程序。在正常的情况下，程序都要对特定的错误进行检查并适当地处理它。对每个可能发生的错误一般用分支转移的办法(即 switch 结构)进行处理。

注意，demo 程序的编译用的是压缩内存模式，对代码用 near 地址(16 位)，对数据用 far 地址(32 位)。此种模式的优点是 ealloc 把一个 far 指针返回给"页面结构"，对扩充内存的数据寻址需要 far 指针。这种方式，所有的指针(如 39 行 emsbase)自动地 far，库功能函数(如 strcpy)接收 far 指针的参数。

可能的话，也可用小内存模式编译程序，不需要对汇编接口子程序进行修改(这些子程序只对中、大和巨大的模块进行修改)。

在三种情况下，需对 C 程序进行改动。

第一，指针 emsbase 被分配给一个由 ealloc 返回的"页面结构"的起始地址，接着又要用这个指针去访问扩充内存，必须明确表示这是一个 far 指针：

```
char    far    * emsbase
```

第二，在没有用 strcpy、printf 和其他需要 near 数据指针参数的 C 库函数直接访问扩充内存时，必须要有替换程序，或首先要将数据拷贝到 near 缓冲器中。例如，把一个字符串拷贝到扩充内存的起始位置：

```
char  * source;             / * Near source,   * /
```

```
char far  * destination;   / * but far destination. * /
...
source = "string to copy to expanded memory";
destination = emsbase;
while( * destination++ =  * source++)
        ;
```

第三，在小内存模式时，有些C库函数可用来访问扩充内存，它们接受两个值，即段和偏移量地址值。例如，movedata，这是一个向扩充内存传送、接收大批量数据的函数(在压缩模式时用memcpy)。

8.3.2 内存驻留应用程序

内存驻留实用程序或设备驱动程序也可以使用扩充内存。实现的方法与临时应用程序稍有不同。内存驻留程序通过以下几步管理扩充内存:

(1) 初始化程序(仅在应用程序装入内存时调用它)调用ememavail来确定扩充内存的可用性，然后调用ealloc保留所需的页数。这时程序应分配扩充内存。

(2) 每次启动内存驻留程序时，首先要调用esave保存当前EMM页面映照状态。这一步不可缺少，因为由内存驻留程序中断的应用程序可能正在使用扩充内存。

(3) 当前页面映照关系保存后，内存驻留代码程序就可按照标准的映照(emap)、读／写、重新映照(emap)步骤对扩充内存进行访问了。

(4) 在内存驻留程序用完扩充内存后(终止之前)，必须调用erestore恢复保存的EMM映照关系。

(5) 如不准备再用扩充内存，需要释放程序占用的内存，这时可用efree。

8.4 Lotus / Intel / Microsoft 扩充内存规范参考手册

Lotus / Intel / Microsoft 扩充内存规范(EMS)定义了运行 MS-DOS 之基于 8086 / 8088 / 80286 微机的一类硬 / 软件子系统，使用户应用程序可访问多达 8Mb 的切换 RAM.

一旦确定了存在扩充内存管理程序 EMM，应用程序便可直接通过软件中断与此管理程序进行通信. 该管理程序的调用序列为:

```
mov   ah,function      ;AH contains the function number
.                      ;other registers are loaded with
.                      ;function-specific arguments.
.
int   67h              ;transfer to Expanded Memory Manager
```

如果 EMM 调用成功，在寄存器 AH 中返回 0 值，否则 AH 中为出错代码.

由应用程序使用类似于打开和关闭文件的过程来获得及释放扩充内存资源，程序所拥有的内存资源由句柄来引用.

本节的材料取自 1985 年 9 月的 Lotus / Intel / Microsoft EMS 文本版本 3.2.

▲EMS 功能 01H——获得管理程序状态

测试扩充内存硬件是否起作用.

调用时: AH＝40H

返回时: AH＝状态

00H 如果功能成功

80H 如果 EMM 软件内部出错

81H 如果 EMS 硬件出故障

84H　如果应用程序请求的功能未定义

注意，此调用仅可用于应用程序已确认确实存在扩充内存管理程序之后。

例:确定扩充内存硬件是否存在且正常工作。

```
        mov   ah,40h        ;40h = EMM Function 1
        int   67h           ;transfer to Manager.
        or    ah,ah         ;test status.
        jnz   emin__error   ;AH<>0 if error
              .
emm_error     .
              .
```

▲EMS 功能 02H——获得页面(page frame)段

获得EMM所用页面的段地址。

调用时:　AH=41H

返回时:　如果功能成功

AH=00H

BX=页面的段地址

如果功能失败

AH=出错代码

80H　EMM 软件内部出错

81H　扩充内存硬件故障

84H　应用程序请求的功能未定义

注意:

· 页面分成 4 个 16K 字节的页,用于将逻辑的扩充内存页映射为 CPU 的物理内存空间。

· 应用程序使用此功能时不必获得 EMM 句柄。

例:获得此系统中 EMS 所用页面的段地址。

```
        mov   ah,40h          ;41H = EMM function 2
        int   67h             ;transfer to Manager.
        or    ah,ah           ;test status.
        jnz   emm_error       ;AH <>0 if error
        mov   page_frame,bx   ;save segment of page frame.
        .
emm_error: .
page_frame dw   0
```

▲EMS 功能 03H——获得页数

获得系统中存在的逻辑扩充内存页总数，以及这些页中未分配的个数。

调用时:　　AH＝42H

返回时:　　如果功能成功:

AH＝00H

BX＝未分配的页

DX＝系统中页的总数

如果功能失败:

AH＝出错代码

80H　EMM 软件内部出错

81H　扩充内存硬件故障

84H　应用程序所请求的功能未定义

注意:应用程序在使用此功能前不必获得一个EMM句柄。

例:获得逻辑 EMS 页总数及剩下的可用页,保存数值为以后引用。

```
        mov   ah,42h          ;42H = EMM Function 3
        int   67h             ;transfer to Manager.
        or    ah,ah           ;test status.
```

```
        jnz     emm_error       ;AH <>0 if error
        mov     total_pages,dx  ;save total EMM pages.
        mov     avail_pages,bx  ;save pages available
        .
emm_error: .
        .
total_pages dw  0
avail_pages dw  0
```

▲EMS 功能 04H——获得句柄及分配内存

通知 EMM 该程序将使用扩充内存，获得句柄，分配该句柄所控制的一定量的扩充内存逻辑页。

调用时: AH＝43H

BX＝欲分配的逻辑页数

返回时: 如果功能成功

AH＝00H

DX＝句柄

如果功能失败

AH＝出错代码

80H EMM 软件内部出错

81H 扩充内存硬件故障

84H 应用程序请求的功能未定义

85H 无更多的句柄可用

87H 请求分配的逻辑页多于实际系统可供使用的页，不分配页。

88H 请求分配的逻辑页多于当前系统中可供使用的页（请求未超过实际存在的物理页，但有一部分已分配给了其

他句柄)．不分配页

89H 如果请求 0 页

注意：

·此为 EMM 中打开文件功能的等价功能．返回的句柄模仿一个文件句柄，并拥有一定量的 EMM 页．以后用于映照内存的每一条请求都必须使用此句柄，应用程序结束后必须由关闭操作来释放．

·如果无句柄可供分配或无足够的扩充内存逻辑页可供使用，则功能 04H 失败．在后一种情况下，可由应用程序调用功能 03H，以确定实际可供使用的页数．

·EMS 的 Intel Abore Board 实现中，句柄号分配顺序为 FF00H,FE01H,FD02H 等．

例:试图打开扩充内存管理程序用于后面的内存页操作，并分配 10 个扩充内存逻辑页(每页 16K 字节)．

```
             mov   ah,43h          ;43H = EMM Fnuction 4
             mov   bx,10           ;BX = pages to allocate
             int   67h             ;transfer to Manager.
             or    ah,ah           ;test status.
             jnz   emm_error       ;AH <>0 if error
             mov   handle,dx       ;allocation successful.
             .                     ;save handle.
emm_error:   .
             .
handle       dw    0               ;save EMM handle.
```

▲EMS 功能 05H——内存映照

将分配给一个句柄之扩充内存逻辑页之一映照为 EMM 页面内 4 个物理页之一．

调用时:　　AH=44H

AL＝物理页号(0–3)

BX＝逻辑页号

DX＝句柄

返回时:　　AH＝状态

00H　功能成功

80H　EMM 软件内部出错

81H　扩充内存硬件故障

83H　非法句柄

84H　应用程序所请求的功能未定义

8AH　所请求的逻辑页映照到分配给句柄之逻辑页外面

8BH　在映照请求中的非法物理页号

注意:

• 此功能中使用的句柄是由功能 04H 返回的。

• 逻辑页号必须处于范围{0…n−1}中，其中 n 为以前用功能 04H 分配给 EMM 句柄的逻辑页数.

• 为了访问已映照为物理页的内存区，应用程序需要知道 EMM 页面的段地址. 此段地址可通过在应用程序初始化过程中调用 EMM 功能 02H 来获得.

例:将分配给 EMM 句柄的扩充内存逻辑页 4 映照为物理页 2，然后将此页的第一个字节装入寄存器 AL.

```
mov   ah,44h        ;44H = EMM Function 5
mov   al,2          ;AL = physical page number
mov   bx,4          ;BX = logical page number
mov   dx,handle     ;DX = handle owning page
int   67h           ;transfer to Manager.
or    ah,ah         ;test status.
jnz   emm_error     ;AH <> 0 if error
```

```
        mov    es,page_frame;get segment of page frame.
        mov    bx,8000h       ;page 2 is at offset 2 * 16K.
        mov    al,es:[bx]     ;read byte from EMM page.
        .
emm_error: .
        .
handle       dw     0         ;EMM handle from Function 4
page_frame   dw     0         ;page Frame segment from
                              ;EMM Function 2
```

▲EMS 功能 06H——释放句柄和内存

释放当前分配给某个句柄的扩充内存逻辑页，然后释放句柄本身。

调用时: AH＝45H

DX＝EMM 句柄

返回时: AH＝状态

00H 功能成功

80H EMM 软件内部出错

81H 扩充内存硬件故障

83H 非法句柄

84H 应用程序请求的功能未定义

86H 保存或恢复映照关系时出错

注意:

• 此功能相当于文件的关闭操作. 它通知 EMM 应用程序不再使用保存在扩充内存页中的数据. 此功能应在应用程序退出前调用.

• 由调用功能 04H 获得 EMM 句柄.

• 如果返回出错条件, 应用程序不应无视. 例如, 如果

返回"busy"出错条件，则分配给该应用程序的逻辑页不可释放，如果该任务在成功前不反复重新调用功能 06H，则这些页将丢失，系统必须复位。

例:释放分配给正在执行程序的 EMM 逻辑页和 EMM 句柄。

```
            mov   ah,45h        ;45H = EMM Function 6
            mov   dx,handle     ;DX = EMM handle to release
            int   67h           ;transfer to Manager.
            or    ah,ah         ;test status.
            jnz   emm_error     ;AH <> 0 if error
            .                   ;otherwise close succeeded.
emm_error:  .
            .
handle      dw    0
```

▲EMS 功能 07H——获得 EMM 版本

返回扩充内存管理程序(EMM)的版本号。

调用时: AH=46H

返回时: 如果功能成功

AH=00H

AL=EMM 版本号

如果功能失败

AH=错误代码

80H EMM 软件内部出错

81H 扩充内存硬件故障

84H 应用程序所请求的功能未定义

注意:

· 此版本号为 EMS 驱动程序软件的版本号作为 BCD 码返回,版本号的整数部分在 AL 的高 4 位，分数部分在低 4

位。

·应用程序应检查 EMM 版本号以保证要用的所有 EMM 功能均可使用。

例:获得 EMM 版本号并保存之。

```
            mov   ah,46h              ;46H = EMM Function 7
            int   67h                 ;transfer to Manager.
            or    ah,ah               ;test status.
            jnz   emm_error           ;AH <> 0 if error
            mov   emm_version,al      ;otherwise save version.
            .
emm_error:  .
            .
emm_version db    0
```

▲EMS 功能 08H——保存映照关系

保存扩充内存板上扩充内存页映照寄存器中的内容以及与指定 EMM 句柄有关的内容。

调用时: AH=47H

DX=句柄

返回时: AX=状态

00H 功能成功

80H EMM 软件内部出错

81H 扩充内存硬件故障

83H 非法句柄

84H 应用程序请求的功能未定义

8CH 页映照硬件状态保存区域满

8DH 映照关系保存失败；保存区已包含与所请求句柄有关的内容

注意:

・由调用功能 04H 获得 EMM 句柄。

・此功能为必须访问扩充内存的中断处理程序或驱动程序所设计．提供给该功能的句柄是在中断处理程序初始化过程中分配给它的句柄，而不是被中断程序的句柄。

・用后面调用 EMM 功能 09H 来恢复映照关系。

例:保存当前执行程序的 EMS 映照状态。

```
        mov   ah,47h          ;47H = EMM Function 8
        mov   dx,handle       ;EMM handle from Function 4
        int   67h             ;transfer to Manager.
        or    ah,ah           ;test status.
        jnz   emm_error       ;AH <> 0 if error

emm_error

handle        dw    0         ;EMM handle from function 4.
```

▲EMS 功能 09H——恢复映照关系

将所有扩充内存硬件页映照寄存器的内容恢复到由以前的功能 08H(保存映照关系)赋给句柄的值。

调用时: AH＝48H

DX＝EMM 句柄

返回时:

AH＝状态

00H 功能成功

80H EMM 软件内部出错

81H 扩充内存硬件故障

83H 非法句柄

84H　应用程序请求的功能未定义

8EH　映照关系恢复失败；保留区域中没有所请求句柄的内容

注意:该功能的使用必须与 EMM 功能08H配对,用于将使用了扩充内存的中断处理程序或驱动程序之映照关系恢复为中断时的状态。

例:为当前执行程序恢复 EMS 映照关系。

```
        mov   ah,48h        ;48H = EMM Function 9
        mov   dx,handle     ;DX = handle from Function 4
        int   67h           ;transfer to Manager.
        or    ah,ah         ;test status.
        jnz   emm_error     ;AH <> 0 if error
        .
emm_error: .
        .
handle      dw    0
```

▲EMS 功能 0CH——获得 EMM 句柄数

获得当前活动的扩充内存句柄数.

调用时:　AH＝4BH

返回时:　如果功能功能:

AH＝00H

BX＝EMM 句柄数

如果功能失败:

AH＝出错代码

80H　EMM 软件内部出错

81H　扩充内存硬件故障

83H　非法句柄

84H　应用程序所请求的功能未定义

注意：

・如果返回的 EMM 句柄数为 0,EMM 为空闲，无扩充内存被使用。

・由此功能返回的值并不是使用了扩充内存的当前活动程序的个数。因为一个程序可实行若干次分配请求，从而拥有几个 EMM 句柄。

・当前活动的 EMM 句柄数不可超过 255.

例:获得当前活动的 EMM 句柄数。

```
            mov    ah,4BH          ;4BH = EMM Function 12
            int    67h             ;transfer to Manager.
            or     ah,ah           ;test status.
            jnz    emm_error       ;AH <> 0 if error
            mov    actives,bl      ;save active handles.
            .
emm_error:  .
            .
actives     db     0
```

▲EMS 功能 0DH——获得句柄拥有的页返回分配给指定 EMM 句柄的逻辑扩充内存页数。

调用时:　　AH＝4CH

　　　　　　DX＝EMM 句柄

返回时:　　如果功能成功:

　　　　　　AH＝00H

　　　　　　BX＝逻辑页数

　　　　　　如果功能失败:

　　　　　　AH＝出错代码

80H EMM 软件内部出错

81H 扩充内存硬件故障

83H 非法句柄

84H 应用程序所请求的功能未定义

注 意:所返回的页数范围是 1到512(如果功能成功).不可能为 EMM 句柄分配给 0 页内存.

例:获得与当前执行程序之 EMM 句柄有关的逻辑 EMS 页数。

```
            mov   ah,4CH        ;4CH = EMM Function 13
            mov   dx,handle     ;DX = handle from Function 4
            int   67h           ;transfer to Manager.
            or    ah,ah         ;test status.
            jnz   emm__error    ;AH <> 0 if error
            mov   peges,bx      ;save assigned pages.
                  .
emm__error:       .
                  .
handle      dw    0             ;EMM handle from function 4
pages       dw    0             ;number of pages assigned
```

▲EMS 功能 OEH——获得所有句柄的页

返回一个数组，包含所有的活动句柄以及与每个句柄有关的逻辑扩充内存页数.

调用时: AH=4DH

ES:DI=接收信息之数组的段:偏移量

返回时: 如果功能成功:

AH=00H

BX=活动 EMM 句柄数

如果功能失败:

AH＝出错代码

80H　EMM 软件内部出错

81H　扩充内存硬件故障

84H　应用程序所请求的功能未定义

注意:

· 该数组以 2 字长的项来填充. 每一项的第 1 个字包含一个句柄, 下一个字包含与该 EMM 句柄有关的页数. 在 BX 中返回的值给出了数组中合法的 2 字长项的个数.

· 因为活动 EMM 句柄的最大数是 255, 故该数组不必大于 1024 字节.

例:获得描述活动 EMM 句柄及其有关 EMS 逻辑页的数组.

```
        mov  ah,4Dh                 ;4DH = EMM Function 14
                                    ;ES:DI = address of array
                                    ;to receive information

        mov  di,segment proc_array
        mov  es,di
        mov  di,offset proc_array
        int  67h                    ;transfer to Manager.
        or   ah,ah                  ;test status.
        jnz  emm_error              ;AH <> 0 if error
        mov  actives,bx             ;save no. of active handles.

emm_error .

actives     dw   0                  ;number of active EMM handles
```

```
proc_arry  dw   512 dup (0)          ;array of EMM information:
                                     ;EMM handles in even words,
                                     ;number of allocated pages
                                     ;in odd words.
```

▲EMS 功能 OFH—— 获得或设置页映照关系

保存或设置扩充内存板上 EMS 页映照关系寄存器的内容。

调用时:　AH＝4EH

AL＝00H　将映照关系寄存器存入数组

01H　根据数组设置映照关系寄存器

02H　一次操作完成获得和设置映照关系寄存器

03H　返回页映照关系数组的大小

DS:SI＝保存信息时(子功能01H，02H)数组的段:偏移量

DS:DI＝接收信息时(子功能00H，02H)数组的段:偏移量

返回时:　如果功能成功:

AH＝00H

AL＝页映照关系数组中的字节数(仅用于子功能 03H)

由 ES:DI 所指向的数组接收映照关系信息(子功能 00H 和 02H)

如果功能失败:

AH＝出错代码

80H　EMM 软件内部出错

81H　扩充内存硬件故障

84H　应用程序所请求的功能未定义

8FH　子功能参数未定义

注意:

· 此功能是在 EMS 版本 3.2 时加入的,设计时考虑为多任务操作系统使用. 通常不由标准的应用程序使用.

· 用户必须保证在数组被 EMM 访问时不发生段的重叠.

· 数组的内容由硬件和 EMM 软件所决定. 除了映照关系寄存器本身的内容外, 该数组还包含其他有用的信息, 可用于将扩充内存子系统还原为以前的状态.

第九章 Intel 8087／80287 数学协处理器编程

MS-DOS 世界是完全属于 Intel 的．这一事实为 MS-DOS 用户带来了两个好处:第一个是为 MS-DOS 系统编写的程序一般来说即使是在目标代码级也是可移植的;第二个是绝大多数 MS-DOS 系统均有能力使用 Intel8087 数学协处理器芯片．

8087 用于为 8086 系统提供快速执行浮点计算的能力．8087 为系统提供的指令包括数值变换、基本算术运算以及某些超越函数,如 sin,cos,及 log．

8087 的好处不仅限于速度方面．由于提供了浮点数学运算子程序库,故程序员可不再编写功能类似的子程序,这样便加快了编程过程．此外由于这些子程序是包含在 8087 芯片中而不是在程序的内存中,故使用 8087 后可使程序减小,从而降低了软件成本．

与早期的数学处理器 Intel8231A 和 8232 等不同,8087 使用在汇编语言程序看来象是机器语言指令那样的转义(escape)序列来访问．8087 不需要安装其他的软件或硬件(只要 8088 或 8086 芯片配置成"max 方式"即可),也不需要对 I／O 或 DMA 编程来进行访问．

由于 8087 与建议的 IEEE 浮点运算标准完全兼容,故可提供一个很大的高级数值运算软件包．对于没有时间编制复

杂数值运算子程序的程序员来说,这一基础软件包是非常有用的。

使用 8087 不仅限于 iAPX8086 和 iAPX8088 处理器,8087 还可用于 iAPX186 和 iIAP188 处理器. 对于 iAPX286 处理器的使用者,Intel 提供了 80287 数值处理器.

9.1 程序员看 8087

9.1.1 8087 中的数据寄存器

虽然 8087 指令看起来象是主处理器指令集的一个部分,但 8087 并不能访问主 CPU 的寄存器. 它有自己的一组寄存器并通过公共内存区与主 CPU 通信. 由于主 CPU 的寄存器不很适合于实数,故此举并非无益. 主 CPU 用的是 16 比特寄存器,而 8087 使用了 8 个 80 比特寄存器,从而可容纳更多的信息. 这些寄存器如图 9-1 所示.

图 9-1 8087 中的寄存器结构

注意,与主 CPU 不同,8087 的数据寄存器并不具有唯一名,但在堆栈中(如 ST(1))中被作为建了索引的登记项存在. 将数值推入此堆栈即将它们装入了 8087. 许多 8087 指令只在推栈顶部工作,而其他多数指令的默认工作状态也是在堆栈顶部.

8087 将其寄存器作为堆栈来寻址的事实是非常重要的，因为所有寄存器的地址都是相对于堆栈顶部而言的。例如，包含在寄存器 i 中的值在堆栈做弹出操作时便包含在了寄存器 i–1 中,而当新的登记项被推入堆栈时,便包含在了寄存器 i+1 中。

对 8087 编程时,必须密切注意堆栈的情况。千万不可认为将某个值装入一个寄存器后,该值仍在同样的地方。

9.1.2 8087 中的浮点实数表示

8087 中的寄存器与主 CPU 的寄存器还有一个不同是，它们只可以保存一种类型的数,即浮点实数(在 Intel 的术语中, 称为临时实数)。图 9–2 最上面的格式给出了浮点实数在 8087 寄存器中的表示。从图中可见,该寄存器分成三个域：符号比特、位移指数(biased exponent)15 比特及有效数字域 64 比特。每一部分均作为无符号二进制整数出现,但合起来可表示一个很大的数字。

下面仔细观察浮点实数的每一部分。最左边(比特 79)是符号比特。此比特为 0 时,则此数为正,反之为负。与 8086CPU 所用的带补数的二进制整数不同,此浮点实数的正数与负数一样多。其结果是此数制系统有两种类型的 0,也就是说 0 可分为正的和负的两种,两个 0 不一定是相等的。8087 注意到这个问题,在试图与 8086 使用的实数作比较时尤其小心。

图 9–2 的右边是有效数值(比特 0 到 63)。此域给出数的有效数字部分。由于每个登记项不是正的就是负的,故两种登记项的范围大小相同。还应注意,比特 63(最高比特)的内容是 1。这是因为8087通常以规范化格式存储数字,这意味着 8087 会在二进制数中找到最左边的 1,然后进行移位使此 1

位于比特 63(一个 1 也没有的数便是 0)．数字 10 的表示为:

十进制: 10

十六进制: A

二进制 64 比特整数: 000000000000000···00000001010

8087 64 比特实数: 101000000000000···00000000000

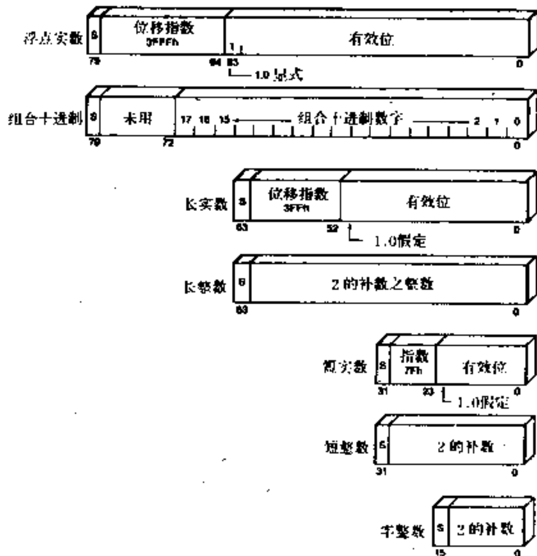


图 9–2 8087 中的数字表示

让我们看看,8087 是如何将数移到左边的．将数移到左边使得有更多的空间用来表示更大的数字,如 10.1,10.12,等等.现在还剩下的问题是,在 8087 中表示的数不再是 10,而是 10×2^{60}．8087 怎么知道这个数实际上刚好是 10 呢?这里它使用了一个称为指数的域(位 64 到 78).

8087 总是假设有效位处于数在 1 和 2 之间．上例中的

有效位便是二进制的 1.01 或十进制的 1.25（括号这是因为在分数中的每一个二进制数字是前一个二进制数字的 1/2，所以在二进制数中小数点右边位置分别是 1/2，1/4，1/8，1/16 等等）。8087 在指数域中记录了原数移动了多少位。如 10,8087 移动小数点三位，从 1010.0（二进制）移为 1.0100（二进制）。3 这个值存入指数域。在 8087 数的存储上还有另外的技巧。由于阶是作为无符号整数存入的，如果 8087 只是将原始的阶存入阶域，那么无法存储小于 1 的数。不存在负指数意味着无法表示小于 2^0 或 1 的数。所以 8087 采用偏移指数的方法，即给指数加上一个偏移量。8087 中用到的偏移量为 3FFFh 或 16383（十进制）。以数 10 的存储为例，它的偏移指数为 3 乘 3FFFh 或 4002h。

图 9-3 示出了数 10 在 8087 内部的表示方法。

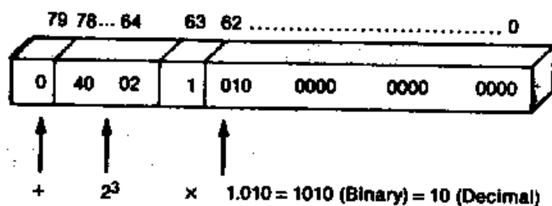


图 9-3 数字 10 在 8087 中的表示

为什么必须了解在 8087 中数是如何存储的呢?因为,在调试程序过程中你可能想检查 8087 寄存器的内容以及为了了解某些 8087 更高级的指令的用法和限制,这时你必须首先知道所控制的数据的类型。

9.1.3 8087 使用的其他数据格式

除内部 80 位浮点实数格式外,图 9-2 中还包括其他 6 种

数据格式。这些表示方法用来做什么呢?除 80 位实数外,这些数据格式是 8087 能用来在内存中读写数据的。如果数据是以其中一种格式表示的,那么 8087 就认识它。

三种基本类型如图 9–2 所示,它们是实数、整数和组合式十进制数。

▲短实数和长实数格式

短实数(32 位)以及长实数(64 位)的格式与刚讨论的 80 位浮点实数非常相似。

这些数同样有表示浮点实数的能力,只是表示范围更小,精度更低。它们之间的差别总结于表 9 · 1.

表 9.1　实数格式之间的差别

数据类型	有效位	指数位	指数偏置	最左边一位
80 位实数	64	15	3FFF(16383)	显式
64 位实数	52	11	3FF(1023)	假定
32 位实数	23	8	7F(127)	假定

除了它们的大小外,短、长实数格式与 80 位实数还有差别。在短、长实数中,最高有效位实数实际上并不出现。因为空间的限制,它们总是假设最左边的位置上为 1,而实际上并不放 1,这样就获得了另一个数字位。

▲单字整数、短整数和长整数数据格式

这些型式是 8086CPU 用来存储 2 的补码整型数,虽然 8086 不能使用 8 字节的整数格式。这些数的表示范围如下:

64位 –9, 223, 372, 036, 854, 775, 808 to 9 223, 372, 036, 854, 775, 807

32 位: –2,147,483,648 to 2,147,483,647

16 位: -32,768 to 32,767

与实数不同,从这种型式取出的整数的值是其确切的表示。注意,虽然这些数是有符号数,并且最高有效位反映数的符号,但它们仍然是 2 的补码数。

▲组合式二一十进制(BCD)格式

8087 的最后一种型式称为组合式 BCD(二一十进制)数。什么是组合式 BCD 数?在二一十进制计数法中,每 4 位 1 组,可为 0 至 9 的单独数字。整个数除表示一个数字串外,别无其它实在意义。在这方面,这种数更象 ASCII 串。在图 9-4 中,我们以数 256 为例,给出了它的一般二进制和二一十进制表示型式。图中还附加了一点有关的计算,这些计算是十进制的。

从图 9-4 中,你可以看到,我们用二一十进制写数,就好象这个数是16进制一样,每个数字用4位表示,但我们把它作为十进制解释。但是,为什么这种数据型式如此重要呢?因为它是 ASCII 和组合式 BCD 之间转换的桥梁。图 9-5 表示 BCD 转换为 ASCII,你只需把这些数字拆开,每组拆为一个数字,放入字节中并加上十六进制的 30 就构成了 ASCII 字符 0 到 9(十六进制 30 到 39)。反之,每个字符减去 30h,按每个字节两个数,把它装配组合好。

这种数据型式在 8087 中只用来存取数据。没有任何一条算法指令能用于组合式 BCD 型式。甚至有了这种限制,8087 的组合式 BCD 数存取指令仍为它所拥有的指令中最有用的两条。这是因为,如果没有把结果告诉用户的方法,计算能力也是毫无价值的,而且大多数人使用标准十进制计数法表示浮点数。

8087 提供了从以 10 为基到以 2 为基的换算,反之亦行。

程序员只需关心 ASCII 串和组合式 BCD 间的转换,并且正确地标定小数点。我们将在有关十进制与二进制浮点转换一节中讨论这个问题,其余的由 8087 自动考虑。

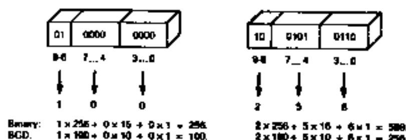


图 9–4　二进制编码十进制数的表示

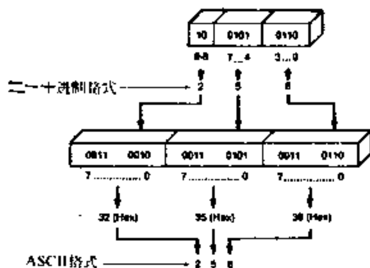


图 9–5　ASCII 与 BCD 数字之间的变换

9.1.4 数据类型小结

在表 9.2 中,我们总结了每一个数据类型能表示的数的大小和每个数据类型支持的近似十进制精度,即有效数字表示的数。按实际应用,我们建议:为了将 ASCII 转换为浮点实数以及作相反的转换,请用组合式二一十进制数。对于所有计算和 MASM 中的实常数,请使用浮点实数。使用整数时,请用能适合 MASM 中整常数的最小整数型式。按照下面的指南,你能通过使用尽可能短的整数型式,获得尽可能高的精度,同时又能节省存储空间。

表 9．2　8087 数据类型的范围与精度.

数据种类	二进数位	十进制数字	近似范围
浮点实数	80	19	$3.4\times10^{-4932}\leqslant N\leqslant 1.2\times10^{4932}$
组合十进制数	80	18	$-10^{18}-1\leqslant N\leqslant10^{18}-1$
长实数	64	15—16	$4.19\times10^{-307}\leqslant N\leqslant1.67\times10^{306}$
长整数	64	18	$-9\times10^{18}\leqslant N\leqslant9\times10^{18}$
短实数	32	6—7	$8.43\times10^{-37}\leqslant N\leqslant0.37\times10^{38}$
短整数	32	9	$-2\times10^{9}\leqslant N\leqslant+2\times10^{9}$
字整数	16	4	$-32.768\leqslant N\leqslant+32,767$

8087 中数的表示范围如图 9—6 所示．注意,8087 内部存储的数的精度(80 位实数)大于存取 8087 寄存器时,其精度也按通常用的数的精度考虑(长实数)．这为计算留下了额外的精度余地．同时还要看到,可以唯一地表示的两数之间的空隔,即 8087 可精确表示的两个相邻数间的距离,由任一边朝零的方向,这个空隔是减小的,而朝正负无穷大的方向则增大．这种数表示的密度分布，意味着 8087 处理很小数的精度比处理大数时更高．

9.1.5 8087 指令集

8087 在工业界中以丰富的指令集而知名．这并不一定意味着 8087 有许多指令 (它有 69 条不同的指令)，而是这个指令集很适于 8087 所希望的操作类型．在 8087中,几乎每一个用途都有一条指令,从而大大减少了一个较小的数学协处理器可能遇到的与编程有关的困难．

这 69 条指令列于表 9•3．这张表是以操作为顺序组织的,而不是以字为顺序．因为你更有可能想通过类型而不是

名字查找一条指令．表 9•3 中有两个标记需要说明．首先是与一些指令相联的(P)标记．这个标记标志着与之相关联的指令可能会以 POP 型式用到,如FopP。POP 型式告诉 8087 使栈指针增大,并且将原栈顶标为空,这样实质上是抛出栈了．这些问题在下文中将进一步讲解．

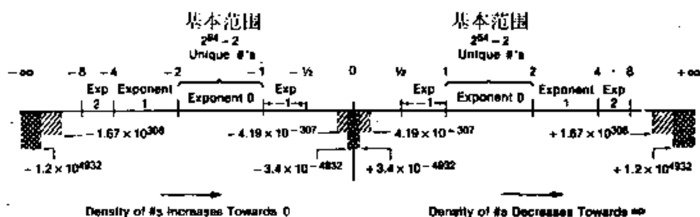


图 9—6　　8087可表示的范围

9.1.6 FWAIT 前缀

表 9•3 中第二个标记是(N)标记．(N)标记表示与之相关联指令可能以不等待的型式使用;如 FNop．通常,MASM 汇编程序为每一条 8087 指令产生一个 FWAIT 前缀．"不等待"型式告诉 MASM 汇编程序不必产生一个 FWAIT 前缀．什么是 FWAIT 前缀呢?

一般来说,8087 接受一条新指令必须等待现行指令完成．这是由 FWAIT 操作码前缀(9Bh)完成的．9Bh 实际上是一个 8086 操作码．当 8086 执行这条指令时,8086CPU 等待,直到 8086／8087 接口上的 TEST 脚有效．当 8087 已经完成执行,并已准备执行下一条指令时,TEST 脚有效．8086CPU 再开始执行,下一条 8087 指令被取出,开始了另一

个循环。

用 FWAIT 作为前缀为的是使 8086 仅在它要送 8087 的另一条指令时才等待。一旦一条 8087 指令送出,8086 和 8087 就能同时处理了.当 8086 再次需要 8087 时,8086 必须检测,直到 8087 准备好.

在另一种情况下,8086 也必须使用 FWAIT 指令。每当 8086 需要从 8087 读取数据时,8086 将导致 8087 用合适的指令把数据存入存储器。8086 必须等待数据可用,这也是通过 FWAIT 指令实现的。在这种情况下,程序员必须明确地写下 8087 指令 FWAIT,因为 MASM 不知道是 8086 而不是 8087 正在等待指令完成。

既然已清楚了(P)和(N)的含义,那么请看表 9.3

表 9.3 8087 指令及寻地格式清单

注释	指令形式	寻址方式	指令名称
			数据传递指令(9)
	FXCH	/ /	d Exchange Registers
(P)	FLD	s	Load Real
	FST	d	Store Real
(P)	FILD	s	Load Integer
	FIST	d	Store Integer
	FBLD	s	Load Packed BCD
	FBSTP	d	Store Packed BCD

			常数指令(7)
	FLDZ		Load+0.0
	FLD1		Load+1.0
	FLDP1		Load Pi
	FLDL2T		Load $\log_2 10$
	FLDL2E		Load $\log_2 e$
	FLDLG2		Load $\log_{10} 2$
	FLDLN2		Load $\log_e 2$
			超越函数指令(5)
	FPTAN		Partial Tangent
	FPATAN		Partial arctangent
	F2XM1		2^X-1
	FYL2X		$Y \times \log_2 X$
	FYL2XP1		$Y \times \log_2 (X+1)$
			比较指令
(P)	FCOM	/ / s	Compare Real
(P)	FICOM	s'	Compare Integer
	FCOMPP		Compare and POP Twice
	FTST		Test Stack Top
	FXAM		Examine Stack Top
			算术指令(25)

(P)	FADD	*	Add Real
	FIADD	s	Add Integer
			算术指令(25)
(P)	FSUB	*	Subtract Real
	FISUB	s	Subtract Integer
(P)	FSUBR	*	Subtract Real(reversed)
	FTSUBR	s	Subtract Integer(reversed)
(P)	FMUL	*	Multiply Real
	FIMUL	s	Multiply Integer
(P)	FDIV	*	Divide Real
	FIDIVT	s	Divide Integer
(P)	FDIVR	*	Divide Real(reversed)
	FIDIVR	s	Divide Integer(reversed)
	FSQRT		Square Root
	FSCALS		Scale
	FPREM		Partial Remainder
	FRNDINT		Round to Integer
	FXTRACT		Extract Exponent and Significand
	FABS		Absolute Value
	FCHS		Change Sign
			进程控制指令(16)
(N)	FINIT		Initialize Processor
	FLDCW	s	Load Control Word

(N)	FSTCW	d	Store Control Word
(N)	FSTSW	d	Store Status Word
+(N)	FSTENV	d	Store Environment
	FLDENV	s	Load Environment
+(N)	FSAVE	d	Save State
	FRSTOR	s	Restore State
	FINCSTP		Increment SP
	FDECSTP		Decrement SP
	FFREE	d	Free Register
	FNOP		No Operation
	FWAIT		CPU Wait
(N)	FDISI		Disable Interrupts
(N)	FENI		Enable Interrupts
(N)	FCLEX		Clear Exceptions

---

* Instruction operand frms for FADD,FSUB,FSUBR,FMUL,FDIV,FDIVR

: F 〈op〉 ···generates F 〈op〉 P ST(1),ST

: F 〈op〉 s ···generates F 〈op〉 ST, 〈memory〉

: F 〈op〉 d,s ···d,s registers only

: F 〈op〉 P d,s···d,s registers only

(P) F 〈op〉 or F 〈op〉 P forms

(N) F 〈op〉 or FN 〈op〉 forms

s source

d destination

/ / s none or source

/ / d none or destination

+ Instuction not self-synchronizing

9.1.7 8087 的寻址方式

8087 的寻址方式反映了处理器的堆栈结构。所有 8087 的数字操作码 与控制操作码不同,它们至少使用栈顶作为一个操作数。一些指令只对栈顶操作 ,如 FSQRT 和 FABS。另一些指令对栈顶及其下面一个寄存器操作,如 FSCALE 和 FZXMI。剩下的双操作数指令的寻址方式是根据类型变化的。某些指令从另一个堆栈寄存器中取得第二个操作数,另一些指令从内存中取得第二个操作数。

表9 .4 表示各种可行的 8087 指令和操作数寻址的联系。注意,虽然一些数学和比较指令可能使用存储器操作数作为源操作数,但是除存储指令外,如 FST<P>,FIST<P> 和 FBSTP,存储器操作数可能从不用作目标操作数。注意,任何整数指令(FIop)的源操作数必须是存储器操作数,因为 8087 的寄存器中存放的总是实数。

表 9.4　8087 数字指令所允许的类型

8087 指令例	字	双字	四字	十字节	8087 寄存器	算术比较指令
FLDsource		YES	YES	FID	YES	实数
FSTdest		YES	YES	FSTP		无
FILDsource	YES	YES	YES			INT
FISTdest	YES	YES	YES			无
FBLDsource				YES		无
FBSTPdest				YES		无

8087 访问它的操作数或许仍存在一些易混淆的问题。下面的小例子能帮助我们搞清楚问题。让我们看看 3 个 8087

操作码的操作。

FLD	<变量 1>	；从内存中取第一个变量
FLD	<变量 2>	；从内存中取第二个变量
FADD		；编码 FADDP ST(1),ST
FSTP	<结果>	；把结果存入内存

这个操作用 FLD 读两个存储器操作数到 8087 寄存器堆栈中,然后用 FADD 的"典型"型式把它们相加,用 FSTP 存储结果。注意，当用基本算法指令（如 FADD, FSUB, FMUL 和 FDIV）自身编程时，MASM 产生"典型的"堆栈操作,弹栈,用栈项(ST)作为源操作数,下一个栈元素(ST(1))作为目标操作数。

上面四条指令的操作示于图 9-7。我们已将 FADD 指令分为两部分,以便你能更好地了解弹栈的作用。注意这些操作,你可看到 8087 依照指令含义完成操作的算法部分(将结果存于 ST(1)),然后弹栈将结果移到栈顶 ST 或 ST(0)。

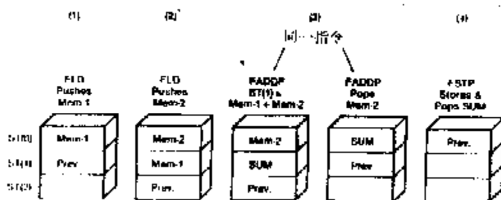


图 9-7 8087 堆栈操作例

在小范例的结尾,栈与我们刚进入时完全一样。如果栈中有存放另外变量的空间,那么结果正是如此。然而,如果栈

中没有足够的空间容纳新数据,由于栈溢出,8087将发出一个错误操作异常信息。在下文中,我们会谈到异常事件。所以,甚至在运行这个小范例前,也必须先肯定8087能接受数据。有两种方法能完成这个工作。

9.1.8 FINIT和FFREE指令

为操作准备8087的最简单方法是通过FINIT指令。每当一个新程序运行时,它应该是送给8087的第一条指令。FINIT指令就象进行硬启动一样初始化8087,这意味着它清出所有寄存器和异常且为程序员工作提供一个清楚的环境。

保证8087有空余寄存器的另一个方法是使用FFREE指令.FFREE标明指定寄存器为空,且允许程序员利用此寄存器作后读计算。注意,不必清出栈顶寄存器。如果栈底(ST(T))有足够空间,那么每当取出一个新值时,上面的寄存器会下压到栈内。

9.1.9 控制8087

▲控制字和状态字

除8个数据寄存器外,8087还有4个程序员可访问的寄存器。从图9—1中,我们可以看到,它们分别是状态字、控制字、操作数和指令指针。8087还有另一个称为标志字的寄存器,但它只能用于8087内部。标志字是8087用来标志其寄存器为空、零还是非数字用的。操作数和指令指针仅在外部错误处理过程中才有用,我们将在下面讨论这个问题。剩下的就是控制字和状态字。你必须了解这两个寄存器,这样才能有效地利用8087。

▲8087控制字

这16位字确定8087处理不同异常条件的方式以及它如何查看它所用到的数字系统。图9—8示出了控制字的各个

域和它们的作用。控制字基本上包括了个控制域和 7 个用于表示异常的标志。我们先介绍异常标志。

下面,我们将使用尽可能多的 8087 附加设备。这样做的原因在于利于 8087 的内部异常处理能力。你可以看到,8087 完全能独立处理可能发生的大多数错误,或者设置一个它所设置的最好的数,或者返回一个称为非数值的特殊值。因为我们自己不易处理这些错误,所以让 8087 做这些工作。我们是通过标记异常事件达到这个目的的,并且是用在控制字中设置异常标记来实现的。所有的异常标记,外加主中断允许标记都包含在控制字的低字节中。

为使 8087 能利用其内部的错误处理,我们将此低字节置为 BFh,用取控制字指令 FLDCW 实现。在 8086 内存中,我们简单地定义一个字,其低字节含值 BFh,然后象下面那样将它取出:

```
:        :        :
CW87     dw       03BFh      ;8087 控制字值
         :        :          :
         FLDCW    CW87       ;取 8087 控制字
```

为什么在控制字高字节中用 3 这个值呢?高字节包含确定 8087 所用数据模式的域。这三个同样示于图 9-8。用值 3 与图 9-8 对照,你可看出,我们选择了 64 位精度,且取整为最接近的整数。这些值是 Intel 公司推荐的,同时也为 8087 作为默认值。如果你想变换这些设置,图 9-8 可以告诉你用什么值。

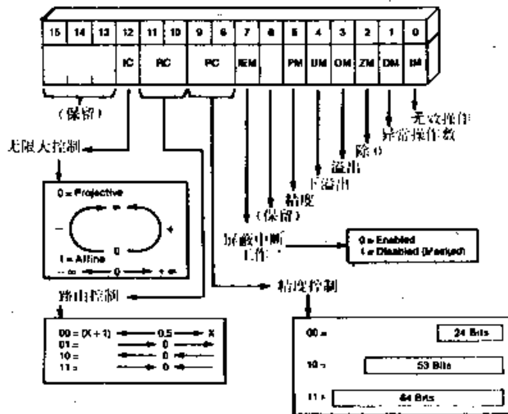


图 9-8　控制字及其对 8087 操作的影响

▲8087 状态字

8087 状态字包含四类信息:(1)忙指示项；(2)栈顶指针；(3)反映 FCOM,FTST 和 FXAM 指令结果的条件码；(4)异常指示项,它标志可能发生的任何错误。图 9-9 给出了状态字中不同指示项的位置。忙指示项指示 8087 现在是否正处理指令。这个指示项实际上对我们用处不大,因为直到 8087 指出它完成了存储状态字的工作,状态字内容才能用。这时,你知道 8087 空闲,因为 FWAIT 指令完成。从第 11 到 13 位是栈顶指针。它对于那些编写按顺序完成连续操作,并且在 8087 栈中存储许多值的复杂程序的程序员很有用。在这种情况下,为了保证下一个操作有足够空间可用,程序操作前先检查栈的深度。如果栈没有足够空间支持这个操作,某些或者所有的寄存器必须存入内存,这使程序能安全地执行。

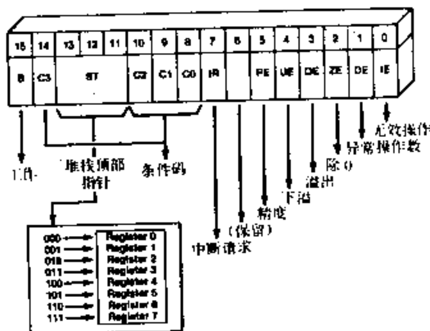


图 9—9　8087 的状态字

栈指针由 FINIT 初始化,指到 000(0),并且每一个成功的取数操作将使之减少,循环回 111(T),直到最后它为 001(1)为止．栈指针可能也为 FINCSTP(使栈指针增加)和 FDECSTP(使栈指针减小)指令所操作．然而,由于这些操作未将寄存器标为空,所以使用 FDECSTP 和 FINCSTP 通过栈项指示项检查寄存器是否为空的方法无效．

在程序分叉点,条件码经常用来决定采取什么动作．在程序设计范例一节中,我们将看到一些使用条件码的例子．简而言之,为了检查条件码,首先用 FSTSW 指令存入状态字,然后通过 8086 检查条件码．当我们为 8086 检查条件码而存储 8087 状态信息时要记住在存储指令后增加一条 FWAIT 指令．

```
    :        :        :
SW87    dw        ?        ;8087 状态字空间
        :                          :
        FCOM      ST(1)    ;检查 ST 和 ST(1)的关系
```

```
FSTSW    SW87        ;存贮8087状态字
FWAIT                ;等待8087完成
  test   SW87,4000h  ;操作数相等吗?
  je     are-equal   ;相等
  :                  :
```

由各种比较指令赋给的条件码的含义见表9．5.注意,条件码并不出现在一组内,而是被栈指针隔开．同样要注意FCOM和FTST指令的返回条件码也被条件位C1隔开,这一位是不用的．还要注意,NAN表示"非数值".

▲8087中的异常处理

状态字的低字节包含一个异常标志．这种标志对应于控制字中的异常码．异常发生时,8087设定相应标志,然后检查该异常事件是否已标记,由于多数操作使用带标记的响应(8087的内部出错处理程序)．应注意周期性地检查异常事件，以保证结果是正确的．如果发生了异常事件,设置适当的标志,并在使用初始化8087(FINIT)指令或清除异常指令FCLEX之前,此标志保持置位．由于此标志始终置位,它们为处理过程中所发生的错误提供了一种累积性的记录.

处理异常事件的另一种方法是,当8087检测到异常情况时,发出一个中断并请求8086去处理此异常情况．但并不需要将8087连接到8036的中断请求线上,而要用一个外部中断处理电路产生来自8087的中断请求．如果系统不支持8087外部中断就不要使用此方法.

如果所用系统支持外部中断,就必须为8087中断8086

时提供异常处理程序. 8086程序读入8087状态字以确定问题的性质. 如果需要,也可由你的异常处理程序检查8087的指令和操作数指针来确定造成问题的指令和操作数. 欲获得此信息,该异常处理程序必须发出8087指令FSTENV或FSAVE. 这些指令至少将五个8087控制寄存器的内容写到8086内存中. 这五个8087寄存器为状态字、控制字、特征(tag)字、指令指针和操作数指针. 异常处理程序可以从内存中检索出这些信息并加以处理.

9.2 对8087使用MS-DOS工具

使用或不使用8087编写程序的唯一差别是,如果带有协处理器,则可提供更多的指令用于数字运算. 由于此差别仅在指令级是可觉察的,所以需了解8087的MS-DOS工具有MASM和DEBUG。其他工具,如LINK,LIB及CREF等与8087的存在无关.

9.2.1 对8087使用MASM

对8087使用MASM时,程序员只需以与8086指令的相同方式输入8087指令. 8087指令与8086指令有相同的域:标号、操作符、操作数及注释. 两种指令的唯一差别是8087操作数只能是8087寄存器或内存,而8086操作数只能是8086寄存器或内存. 在内存操作情况下,两种格式没什么不同. 8087指令可使用下述五种基本内存格式之一.

–Displacement	FSTSW	mem word
–Base或Index	FIADD	word ptr[bx]
–Displacement+Base或Index	FSTP	base [di]
–Base+Index	FLDCW	[bp] [si]
–Displacement+Base+Index	FILD	[bp]table[di]

警告:

MASM 版本 1.25 中有一个错误会造成操作符FSUB与 FSUBR 或 FDIV与FDIVR错误交换(如果上述操作符使用时未指定操作数). 如果使用的是老版本的MASM, 应为这些指令指定操作数及类型:

```
FSUBP    ST(1),ST
FDIVRP   ST(1),ST
```

注意,此种格式总是使用这些指令的弹出格式.

表9.5 由 FCOM,FTST 和 FXAM 指令设置的状态条件

	条件码				结果
	C3	C2	C1	C0	
F	0	0	D	0	ST>源码
C	0	0	o	1	ST<源码
O	1	0	N	0	ST=源码
M	1	1	T	1	ST?源码
F	0	0	C	0	ST>0.0
T	0	0	A	1	ST<0.0
S	1	0	R	0	ST=0.0
T	1	1	E	1	ST?0.0
F	0	0	0	0	+异常
X	0	0	0	1	+无
A	0	0	1	0	−异常
M	0	0	1	1	−无
	0	1	0	0	+正常
	0	1	0	1	+∞
	0	1	1	1	−∞
	1	0	0	0	+0
	1	0	0	1	空
	1	0	1	0	−0
	1	0	1	1	空
	1	1	0	0	+异常
	1	1	0	1	空
	1	1	1	0	−异常
	1	1	1	1	空

9.2.2 MASM 的 8087 开关—/r 和/e

一旦程序输入到文件中,就必须用 MASM 来汇编这个程序。如果使用标准的 MASM 命令型式,那么每遇到一条 8087 指令就产生一个语法错误。这是因为在通常的操作模式下,MASM 不知道有关 8087 的任何信息。为了准确汇编 8087 指令,就必须在命令行告诉 MASM,源文件中包含的 8087 指令要通过用命令行开关"/r"(实模式)的方法才可达到目的。如以下命令:

A:>masm test. asm test.obj test,lst test.crf/r

这使得 MASM 知道正被汇编的程序将在实实在在的 8087 上执行。那么,MASM 产生合适的 8087 操作码,用 FWAIT 操作加前缀,除非用了一条 FN<op>指令。请注意,虽然 8087 的空操作指令 FNop 也是由 FN 开始,它同样也产生一个 FWAIT 前缀。

MASM 还有一个指挥它汇编 8087 指令的开关,这就是"/e"开关(模拟模式)。/e 模式开关的功能除"不等待"指令(FN<op>)不汇编外,其余与实模式开关功能相似。这个开关是给那些拥有能用 8086CALL 模拟子程序替换 8087 操作码的模拟库的用户用的。因为 MASM 不提供这样一个模拟库,如果你确实有一个 8087,也就没有使用这个库的必要。

9.2.3 MASM 中的 8087 数据类型

现在已知道 8087 支持七种不同的数据类型:字、短和长整数、短和长实数、组合式二一十进制数以及浮点实数。为了利用这些类型,必须在内存中定义合适的存储单元。8087 的数据类型以及 MASM 中定义和说明其方式的对应关系,请见表 9·6。

存储单元是用 MASM 数据定义命令(dw,dd,dq,dt)后面

跟一个问号(?)分配的．这种格式告诉 MASM 保留空间而不是初始化．为了初始化保留的单元为特定的实数值,MASM 提供了三种不同的型式:不带指数的科学计数法、带指数的科学计数法以及实(R)型式．使用其中每一种型式都可带任何更大的"数据定义命令",如下例．

```
double   dd  3.14159                    ;不带指数的科学计数法
quad     dq  1.23456E+03                ;带指数的科学计数法
tenbyt   dt  0123456789ABCDEF0123R      ;实型式
```

表 9·6　比较 8087 与 MASM 的数据类型

8087 数据类型	8086 数据类型	MASM 字节数	伪指令	操作数名	8087 兼容度
Word integer	Word	2	dw	word ptr	Yes
Short integer	double word	4	dd	dword ptr	Yes
Short real	double word	4	dd	dword ptr	No
Long integer	quad word	8	dq	qword ptr	Yes
Long real	quad word	8	dq	qword ptr	No
Packed BCD	ten byte	10	dt	tbyte ptr	"R"form
2Floating real	ten byte	10	dt	tbyte ptr	Yes

用定义字节(db)或定义字(dw)命令定义实数是不可能的．实数可能只被初始化为整数值．

既然为了便于实数表示法的十六进制表示能与它的定义相对应而采用按每组一位为基准,那么,科学计数法就定义为浮点数格式,其中包括符号、指数和有效数．

注意,虽然 MASM 有用 4 和 8 字节长定义实数的能力,但用来初始化这些数的格式与 8087 不兼容!图 9-10 表示

Microsoft 公司是如何实现这些规格的实数的. 通过与图 9-2 的比较,可以看出它们之间有很大差别. 如果一定要用这些格式,又由于考虑到与已有软件的兼容性,你可以自己编制将其由一种格式转换为其他格式的转换程序.

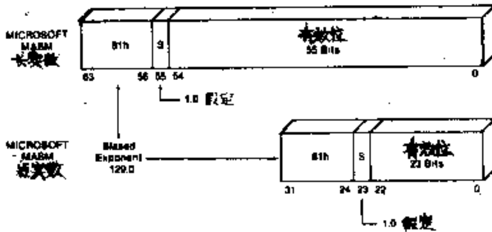


图 9-10　Microsoft MASM 实数格式

9.2.4 对 8087 使用 DEBUG

DEBUG 总是懂得 8087 指令的. 这可以解释为什么有时当你想"反汇编"内存时,DEBUG 列出奇怪的指令. 用十六进制字 DEAD 填充未用的存储单元, 是调试过程中常用的技巧. 这种易于识别的型式使程序员能快速查看哪些内存单元已改变. 然而,DEBUG 将其反汇编为 FISUBR WORD PTR[DI+ADDE].

虽然 DEBUG 总是在 8087 模式下,但它并不能识别所有的 8087 指令 .DEBUG 既不显示,也不允许你汇编任何 FN<op>型式的指令. 其原因是 DEBUG 将 FWAIT 从 8087 操作码中分离出来作为单独一条指令识别,实际上它也是单独一条指令. 所以,DEBUG 将 FN<op>指令作为一条不出现 FWAIF 前缀的标准指令译码.

与 MASM 相反,DEBUG 并不自动在标准的 8087 指令

前插入 FWAIT 前缀．当使用 DEBUG 输入 8087 指令时，必须记住人工送入 FWAIT 指令。

同时还应该记住 DEBUG 说明存储器操作数时,必须用下面型式告诉 DEBUG 这个操作数的规格．方括号是告诉 DEBUG,这个数是地址而不是立即值．

▲debug 8087 的寄存器

DEBUG 无法做到的事情之一是显示 8087 的状态或其寄存器的内容。如果打算检查某个 8087 寄存器,首先必须使 8087 将数据写到公用的内存中．

下节中所提供的子程序 dump87 可用于 debug8087 程序．该子程序使用 FSAVE 指令存储 8087 的全部状态,然后在控制台显示屏上以更易了解的格式将其显示出来．该子程序可放入一个库中或在需检查 8087 计算的状态时调用．

▲指令的编码格式

阅读十六进制 dump 时,如果出现 FWAIT 操作码(9B)或 escape 代码 D8 到 DF,则说明出现了 8087 指令．图 9—11 给出了 8087 指令可能取的各种格式,但所有指令均以比特图形 11011 开始．

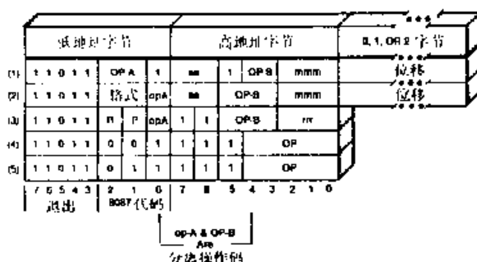


图 9—11　指令的编码格式

9.3 用 MASM 对 8087 编程的例子

欲对 8087 进行编程,必须深入了解 8087,得到 Intel 的指令参考手册,还必须阅读一些现成程序以积累经验。下面我们举例来说明 8087 的运行方式,并为你建立自己的 8087 程序库提供一个起点。

9.3.1 FWAIT 和 FINIT 指令

如果 8086CPU 打算使用来自 8087 的结果,首先必须保证 8087 是由发出 FWAIT 指令结束的。

还有一点必须提及,即 8087 必须在程序开头处用 FINIT 指令进行初始化。在继续运行操作之前,必须使 8087 进入一个已知的状态,这一点尤其重要。

9.3.2 DUMP87 子程序

前面已讲到,DEBUG 程序不能用于检查 8087 的内容或状态,下面提供一个子程序以显示出 8087 的内容,然后进行检查。此程序如下:

```
        page    60,132
;*****************************************************************
;
;     L I B R A R Y   I M P L E M E N T A T I O N
;
;
        public  dump87          ; DEFINED LIBRARY ROUTINE
code    segment para public 'code'
        assume cs:code,ds:code,es:code,ss:code
        extrn   bin2hex:near    ; CALLED LIBRARY ROUTINE
;*****************************************************************
;
;    D U M P 8 7 -- 8 0 8 7   D E B U G G I N G   T O O L
;
; This procedure dumps the entire state of the 8087 onto the
; stack, then formats and outputs said state to the terminal.
;
; Setup Requirements: NONE
; Stack Requirements: 108 bytes free on the stack
;
; ... wd--Word Defines for bit fields within various words.
; The defined structures take advantage of the fact that the
;      SW and CW interrupt structures match.
;
;*****************************************************************
```

```
;
;          M A C R O   D E F I N I T I O N S
;
;; Display a character (from DL)
dis_chr macro   char
        push    ax
        push    dx
        mov     dl,&char
        mov     ah,02h
        int     21h
        pop     dx
        pop     ax
        endm
;; Display a String by Label
dis_str macro   string
        push    ax
        push    dx
        mov     dx,offset &string
        mov     ah,09h
        int     21h
        pop     dx
        pop     ax
        endm
;; Display a String (from DS:DX)
display macro
        mov     ah,09h
        int     21h
        endm
;*****************************************************************
;
;       S T R U C T U R E   D E F I N I T I O N S
;
intrpt  record  master:1,nul0:1,pr:1,un:1,ov:1,zd:1,de:1,inv op:1
control record  infc:1,rndc:2,prec:2
status  record  busy:1,c3:1,stp:3,c2:1,c1:1,c0:1
tag     record  onetag:2
ipwd    record  ipseg:4,nul2:1,opcode:11; opcode & instruction pointer
opwd    record  opseg:4,nul3:12         ; operand pointer segment
expwd   record  sign:1,exp:15           ; sign & exponent
; Basic Environment Structure
enviro struc
cw87    dw      ?                       ; control word
sw87    dw      ?                       ; status word
tw87    dw      ?                       ; tag word
ipo87   dw      ?                       ; instruction pointer offset
ips87   dw      ?                       ; IP segment & opcode
opo87   dw      ?                       ; operand pointer offset
ops87   dw      ?                       ; OP segment
enviro  ends
; Register Structure
fltreg  struc
man87   dq      ?                       ; mantissa (significand)
exp87   dw      ?                       ; exponent & sign
fltreg  ends
; Entire State Save Structure
state87 struc
        db      size enviro dup (?)     ; environment header
reg87   db      size fltreg * 8 dup (?) ; 8 data registers
state87 ends
dump87s struc                   ; stack format for dump 87
```

```
rec87   db      size state87 dup (?)     ; space for 8087 state
;oldbp  dw      ?                        ; entry base pointer
dump87s ends
base    equ     [bp - size dump87s]      ; structure index
;*****************************************************************
;
;       B E G I N   P R O G R A M   C O D E
;
dump87 proc     near
        push    bp                      ; save entry BP
        pushf                           ; save caller's flags
        push    ds                      ; save caller's data segment
        mov     bp,sp                   ; and set up index
        sub     sp,size dump87s         ; allocate space for local store
        push    ax                      ; save caller's registers
        push    bx
        push    cx
        push    dx
        push    di
        push    si
        mov     ax,cs                   ; set DS to point to this routine's
        mov     ds,ax                   ; ... data area.
; Get copy of the 8087's internal state
        pushf                           ; save caller's interrupt state
        cli                             ; don't allow interrupts while
                                        ; saving
        FSAVE   base.rec87              ; save state of 8087
        FRSTOR  base.rec87              ; restore state that was just saved
        FWAIT                           ; wait to complete restore
        popf                            ; reenable interrupts ?
; Now that we have a copy of the 8087's state, decode it and present
; it to the user on the terminal.
;
;       Presentation consists of the following items:
;
;       ==================== 8087 DUMP =======================
;       Infinity:  Affine     Round....... near    Precision: 64
;       Inst Addr: x:xxxx     Oper Addr: x:xxxx    Opcode:  Dxxx
;
;               INT PRE UND OVR ZER DEN IOP             C3 C2 C1 C0
;       Enable: x   x   x   x   x   x   x               x  x  x  x
;       Signal: x   x   x   x   x   x   x   x means unmasked
;                                             or signaled
;               exponent        significand
;       ST(x)   + xxxx    xxxx xxxx xxxx xxxx #0 tag
;         :        :        :    :    :    :    :  :
;       --------------------------------------------------------
;
;
; Infinity, Rounding, and Precision Control
        dis_str line_1                  ; start display
        mov     al,byte ptr base.cw87+1 ; get control word
        and     al,mask infc            ;       infinity control
        mov     cl,infc
        shr     al,cl                   ;       condition #
        mul     inf_siz                 ;       condition offset
        add     ax,offset inf_cnd       ;       condition address
        mov     dx,ax
        display
        dis_str rnd_lab
```

```
        mov     al,byte ptr base.cw87+1 ; get control word
        and     al,mask rndc            ;       rounding control
        mov     cl,rndc
        shr     al,cl                   ;       condition #
        mul     rnd_siz                 ;       condition offset
        add     ax,offset rnd_cnd       ;       condition address
        mov     dx,ax
        display
        dis_str pre_lab
        mov     al,byte ptr base.cw87+1 ; get control word
        and     al,mask prec            ;       precision control
        mov     cl,prec
        shr     al,cl                   ;       condition #
        mul     pre_siz                 ;       condition offset
        add     ax,offset pre_cnd       ;       condition address
        mov     dx,ax
        display
; Instruction & Operand Pointers, and Opcode
        dis_str line_2                  ; next line
        mov     ax,base.ips87           ;       instruction pntr.
        and     ax,mask ipseg           ;               segment
        mov     cl,ipseg
        shr     ax,cl                   ;               digit
        mov     ch,1                    ;               display 1
        call    bin2hex
        dis_chr ':'
        mov     ax,base.ipo87           ;       instruction pntr.
        mov     ch,4                    ;               offset
        call    bin2hex
        dis_str opadr                   ; operand pointer
        mov     ax,base.ops87           ;               segment
        and     ax,mask opseg
        mov     cl,opseg
        shr     ax,cl                   ;               digit
        mov     ch,1                    ;               display 1
        call    bin2hex
        dis_chr ':'
        mov     ax,base.opo87           ;       operand pntr.
        mov     ch,4                    ;               offset
        call    bin2hex
        dis_str ocode                   ; opcode
        mov     ax,base.ips87
        and     ax,mask opcode
        or      ax,0800h                ; add OPCODE assumed bit
        mov     ch,3                    ;               3 digits
        call    bin2hex                 ;               display
; Interrupt/Exception--Enable Flags
        dis_str line_3                  ; next line
        mov     al,byte ptr base.cw87   ; exception enable flags
        call    exception_flags         ; show status
; Condition Codes
        dis_str space10
        mov     ah,byte ptr base.sw87+1 ; condition codes
        push    ax                      ;       (save codes)
        mov     al,30h                  ;       (ASCII "0")
        and     ah,mask c3              ;       C3
        sub     ah,mask c3              ;       0 -> CY, 1 -> NC
        cmc                             ;       0 -> NC, 1 -> CY
```

```
        adc     al,0                    ;       0 -> "0", 1-> "1"
        dis_chr al                      ;       display
        pop     ax                      ;       (save codes)
        mov     ch,c2 + 1               ; # of codes to display
next_cc:
        dis_str space2
        mov     al,30h                  ;       (ASCII "0")
        and     ah,mask c2 + mask c1 + mask c0
        sub     ah,mask c2              ;       0 -> CY, 1 -> NC
        cmc                             ;       0 -> NC, 1 -> CY
        adc     al,0                    ;       0 -> "0", 1-> "1"
        dis_chr al                      ;       display
        shl     ah,1                    ; next code
        dec     ch                      ; 1 less to go ...
        jnz     next_cc                 ; ... until all done
; Interrupt/Exception--Status Flags
        dis_str line_6
        mov     al,byte ptr base.sw87   ; exception signal flags
        call    exception_flags         ; show status
; Data Register Display
        dis_str crlf
        mov     dh,8                    ; # of reg. to display
        mov     si,0                    ; start with reg #0
register_display:
        dis_str line_8                  ; registers status
        push    dx                      ; save count
        mov     al,8                    ; calculate register #
        sub     al,dh
        add     al,30h                  ;       convert to ASCII
        dis_chr al                      ;       and display
        pop     dx
; Sign of Data Register
        dis_str paren                   ; sign comes next
        mov     ax,word ptr base.reg87[si].exp87
        test    ax,mask sign            ; what is it?
        jnz     sign_minus
        dis_str plus
        jmp     show_exponent
sign_minus:
        dis_str minus
; Exponent Portion of Data Register
show_exponent:
        and     ax,mask exp             ; obtain exponent
        xor     cx,cx                   ;       four characters
        call    bin2hex                 ;       and display
        dis_str space3
        mov     di,si                   ; base of register
        add     di,offset exp87         ; location of mantissa
        mov     dl,4                    ; 4 words per register
; Display Significand Portion of Data Register
show_significand:
        sub     di,2                    ; point at word start
        mov     ax,word ptr base.reg87[di]
        call    bin2hex                 ;       and display
        dis_str space1
        dec     dl                      ; another word gone
        jnz     show_significand
; True Register Number
```

```
        dis_str truenum
        mov     al,byte ptr base.sw87+1 ; get stack pointer
        and     al,mask stp
        mov     cl,stp
        shr     al,cl                   ; have stack pointer
        mov     cl,8                    ; convert counter to ...
        sub     cl,dh                   ; ... 0 through 7
        add     al,cl                   ; current reg. #
        and     al,07H
        push    ax                      ; save register number
        add     al,30h                  ;       convert to ASCII
        dis_chr al                      ;       and display
        dis_str space2                  ; now for the TAG field
; Tag Word Status
        mov     ax,base.tw87            ; get tag word
        pop     cx                      ; get register number in CL
        shl     cl,1                    ; multiply by 2
        shr     ax,cl                   ; and get proper tag word
        and     ax,mask tag
        push    dx
        mul     tag_siz                 ;       condition offset
        add     ax,offset tag_cnd       ;       condition address
        mov     dx,ax
        display                         ;       show tag status
        pop     dx
; All Done for That Register!
        add     si,size fltreg          ; next register
        dec     dh                      ; 1 less
        jz      finished
        jmp     register_display        ; until all gone
; All Done for All Registers!
finished:
        dis_str line 9                  ; all done!
; Restore the 8086 to the way it was and return
; Start w/ saved registers
        pop     si                      ; restore caller's registers
        pop     di
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        mov     sp,bp                   ; restore stack
        pop     ds                      ; restore data segment
        popf                            ; restore caller's flags
        pop     bp                      ; restore entry BP
        ret                             ; return when finished
;*****************************************************************
; Display subroutine for displaying MASK & SIGNAL status of
; exceptions.
; Test byte in AL for bits corresponding to exception flags
;
exception_flags         proc    near
        test    al,mask master          ;       master control
        call    mark_it
        mov     cl,pr                   ; next is PR flag
        ror     al,cl                   ; move to 1's position
        inc     cl                      ; count 1 > than bit #
test_exception:
```

```
        test    al,1                    ; is flag set?
        call    mark_it
        rol     al,1                    ; next flag
        dec     cl                      ; keep track of count
        jnz     test_exception          ; continue until done.
        ret
;*********************************************
; Mark result according to flags set on entry
;
mark_it         proc    near
        jz      mark_space
        dis_str marky
        ret
mark_space:
        dis_str markn
        ret
mark_it         endp
exception_flags         endp
;****************************************************************
;
; D U M P 8 7   L O C A L   C O N S T A N T   S T O R A G E
;
;           ----- this section read only -----
;
; "_lab"--label for section
; "_cnd"--condition for label
; "_siz"--number of bytes in condition
cret    macro                           ;; new line macro
        db      0Dh,0Ah
        endm
line_1  equ     $
        cret
        db '===================== 8087 DUMP ======================'
        cret
        db      'Infinity:  $'
rnd_lab db      '     Round:....... $'   ; Label
pre_lab db      '     Precision: $'      ; Label
inf_siz db      7
inf_cnd db      'Proj. $'               ;       infinity state
        db      'Affine$'               ;       infinity state
rnd_siz db      5
rnd_cnd db      'near$'                 ;       round state
        db      'down$'                 ;       round state
        db      'up  $'                 ;       round state
        db      'chop$'                 ;       round state
pre_siz db      3
pre_cnd db      '24$'                   ; "ret" precision state
        db      '**$'                   ; "ret" precision state
        db      '53$'                   ; "ret" precision state
        db      '64$'                   ; "ret" precision state
line_2  equ     $
        cret
        db      'Inst Addr: $'          ; "x:xxxx"
opadr   db      '     Oper Addr: $'     ; "x:xxxx"
ocode   db      '     Opcode:  D$'      ; "xxx","ret","ret"
line_3  equ     $
        cret
        cret
```

```
        db      '       INT PRE UND OVR ZER DEN IOP'
        db      '          C3 C2 C1 C0'
        cret
        db      'Masked:$'
;               condition codes                    "ret"
line_6  equ     $
        cret
        db      'Signal:$'                        ; "ret"
marky   db      ' x  $'
markn   db      '    $'
line_8  equ     $
        cret
        db      'ST($'                            ; "x"
paren   db      ')   $'
plus    db      '+ $'
minus   db      '- $'                             ; "xxxx"
space10 db      '          '                      ; 10 space
space2  equ     $ + 1                             ; 2 space
space1  equ     $ + 2                             ; 1 space
space3  db      '   $'                            ; 3 space
                                                  ; "xxxx " 4 times
truenum db      ' #$'                             ; " #x", then " "tag
tag_siz db      6
tag_cnd db      'Valid$'                          ;       tag state
        db      'Zero $'                          ;       tag state
        db      'Spec.$'                          ;       tag state
        db      'Empty$'                          ;       tag state
line_9  equ     $
        cret
        db '-----------------------------------------------------'
crlf    equ     $
        cret
        db      '$'
dump87  endp
;*****************************************************************
code    ends
        end
```

DUMP87 显示的信息是使用 8087 的 FSAVE 指令时所产生的．此指令以图 9-12 所示的 94 字节格式保存 8087 的整个状态．如果执行了 FINIT 指令,FSAVE 也对 8087 进行初始化．从而可用一个子程序去保存 8087 的状态,然后用一条指令将其初始化．因为我们打算不中断地继续处理,所以必须在 FSAVE 指令后再使用 FRSTOR 指令。该指令根据保存的信息重新装入 8087。

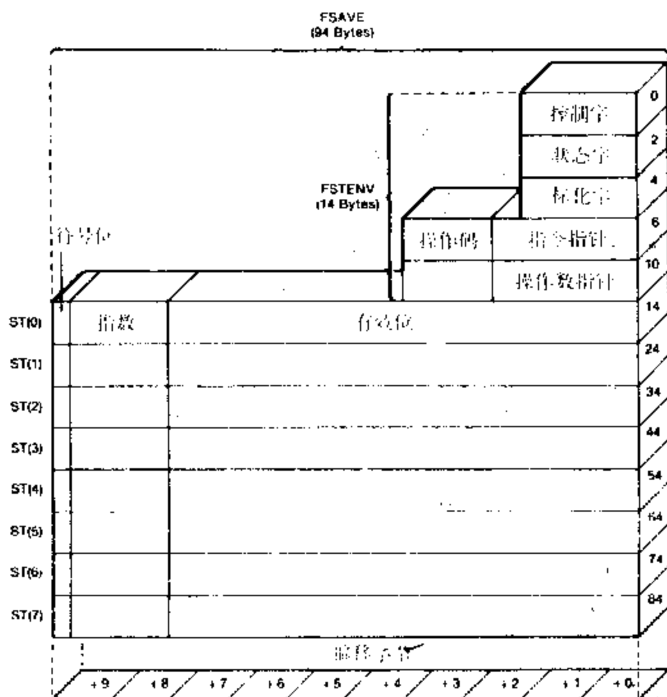


图 9—12　FSAVE 和 FSTENV 的内存结构

从图中可看到所保存信息的前 14 个字节与 FSTENV 指令(存储环境)所保存的完全一样。FSTENV 指令虽然不重新初始化 8087,但它允许程序员访问某些特殊情况下所需要的信息:状态字、指令、操作数指针。与 FSAVE 相似，FSTENV 也有一条称为 FLDENV 的配套指令,可用于根据存储的信息重新装入环境。

▲使用 DUMP87 子程序

程序的剩余部分与 8087 无关．程序后面使用 MASM 结构和记录定义分解 FSAVE 所返回的信息并显示给用户．信息的显示格式在该子程序标题部分进行了说明．所给清单适合于汇编程序及作为一个库文件．如果遵照此步骤，可由将 DUM87 作外部过程进行声明，由提供外部程序 BIN2HEX，并由匹配 DUM87 的段和类（class）名称，而将其包含在任意其他文件中．使用 DUMP87 的一种变种方法如下：

```
cod     segment  parapublic'code'       ;library  segment
        assume   cs:code,ds:code,es:code;ss:code
        extrn    dump87:near            ;LIBRARY  ROUTINE
        ORG      0100h
main    proc     far
start:
        FINIT                           ;initialize  8087
          :               :                        :
        call     dump 87                ;analyze  8087
          :               :                        :
```

DUMP87 需要超过 CPU 堆栈 120 字节．返回时，该程序不使用任何数据存储单元．DUMP87 要使用称为 BIN2HEX 的子程序如下：

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
;BIN2HEX—BINary to HEXadecimal conversion. Displays the
;contents of the AX register on the screen as a hexadecimal
;number. CH register contains count of the number of characters
;to display.
```

```
bin2hex         proc        near
        push    bx
        push    cx
        push    dx
        mov     bx,ax       ; use BX as temporary holding
        cmp     ch,0        ; count already set?
        jne     align_left
        mov     ch,4        ; no, set character count
    ; Align the number on the leftmost side of the AX
    ; (rotate left by (4-CH) * 4bit positions
align_left:
        mov     cl,4        ; find number of digits to shift
        sub     cl,ch
        shl     cl,1        ; multiply by 4
        shl     cl,1
        rol     bx,cl       ; align on left side
        mov     cl,4        ; and set minor rotate count
    ;Main Loop—repeat N times···Print the Leftmost digit
more_dec:
        rol     bx,cl       ; left digit to right
        mov     al,bl       ; move to AL
        and     al,0fh      ; right digit only
        add     al,90h      ; sneaky conversion
        daa                 ; to ASCII hex characters
        adc     al,40h
        daa
; Print digit
```

```
        dis_chr al
        dec     ch           ; digits count-1
        jnz     more_dec     ; continue if more
        pop     dx
        pop     cx
        pop     bx
        ret
bin2hex endp
;**********************************
```

9.3.3 使用8087实现二—十进制变换

下面介绍几种更高级的程序设计方法、首先介绍以十进制表示对 8087 进行数据输入、输出的方法。

▲整数运算

由于有了 FBLD,FBSTP 组合式 BCD 装入和存储指令,故在 8087 上执行整数到二进制变换是十分容易的。唯一需要的是使用一个简单的 8086 程序使 ASCII 字符串压缩并复原为 BCD 数字。欲从十进制变换为二进制,使用 FBLD 装入该十进制数,并使用 FIST 将其作为一个二进制整数存储。反方向转换则需使用 FILD 指令,后跟十进制存储指令 FBSTP。

注意,只要该被变换的数字小得足以在一个 16 位寄存器中容下,使用 8087 从十进制转换为二进制便不值得。压缩成数字及执行 FBLD—FIST 序列的有关开销比下述标准"移位乘法"变换子程序更大。

```
;Assume number being accumuloted is in AX and
          the new cigit is in the CH register.
shr      ax,1      ;eisting  number×2
mov      bx,ax     ;save
shr      ax,1      ;number×4
shr      ax,1      ;number×8
add      ax,bx     ;(#×8)+(#×2)=#×10
xor      ch,ch     ;prepare  for  16-bit  add
add      ax,cx     ;next  digit  added  in
```

对于小的数字(1到3位十进制数),8087将十进制变换成二进制的时间要长两倍.

如果数字大于16个二进制位,8086开始减慢,因为它必须连续地检查进位标志及可能的溢出等. 在16到64二进制位范围内,8087确实能使变换大大加快.

只要该数字不大于18个二进制数字(通常不可能超过),无需比装入和存储指令更多的8087指令. 一旦超过了18个数字,必须进行规一化.

▲浮点运算

十进制与二进制数浮点变换处理主要归结为规一化的问题,即可使用FBLD和FBSTP指令从8087获得或送入基本数字,然后使用十的幂来调整此数. 需使用到如下的算术恒等式变换.

1. $10^X=2^X \cdot \log_2 10$
2. $E^X=2^X \cdot \log_2 E$
3. $Y^X=2^X \cdot \log_2 Y$

4. $\log_{10} X = \log_{10} 2 \cdot \log_2 X$

5. $\log_E X = \log_E 2 \cdot \log_2 X$

需使用下面的指令完成各种变换:

A. F2XM1 calculates2^x-1

B. FLDL2T constant $\log_2 X$

C. FLDL2E constant $\log_2 E$

D. FYL2X calculates $Y \times \log_2 X$

E. FLDLG2 constant $\log_{10} 2$

F. FLDLN2 constant $\log_E 2$

一旦装入一个整数,或是以正的以十为底的指数乘它,或是以负的以十为底的指数除它.

▲ 2^x 计算

通常可通过简单的移位操作使 2 升幂,8087 即使用此来实现 FSCALE 指令. 但由于 10 的整数幂不对应于 2 的整数幂,需计算 2 的幂的分数部分. 这便是 8087 指令 F2XM1 的作用.

F2XM1 可用于计算 2 到 X 幂,X 值从 0.0 到 0.5. 给出任意数 X,可由计算如下表达式将其分为整数部分和分数部分.

integer(X) = FRNDINT(X)

fractional(X) = FSUBX−integer(X)

X 的整数部分由 FSCALE 用于将 2 升至某个整数幂,而分数部分成为 F2XM1 的输入. 我们可连续使用两次操作,因为有:

$$2^{(Y+Z)} = 2^Y * 2^Z$$

X 分数部分的绝对值在 0.0 至 0.5 中,可保证 8087 的程序控制最紧密,从而保证了最大分数为 0.5.

然后使用 F2XM1 来计算总结果。必须注意,如果分数部分是负的,可使用其绝对值并使用恒等式:

$$2^{(Y-Z)}=2^Y/2^Z$$

▲10^x 计算

根据运算规则,有:

$10^x=2^{x*\log_2 10}$

如果利用 2^x 计算子程序,则只需计算出值:

$X * \log_2 10$

而 8087 可计算出以 2 为底的 log10。从而计算 10^x 可选运算 FLDL2T,然后用乘法 FMUL,最后调用 EXP10。仿照上面的方法,亦可计算出 e^x, Y^x 等。

▲十进制数到实数的规一化函数

计算出了 10^x,便可在 8087 中使用科学表示法。给出指数用的组合式 BCD 数及字整数 X,便可将其变换成浮点实数,方法是使用 FBLD 装入组合式 BCD 有效部分,计算出 10^x,然后进行组合,使用子程序 FMUL(用于正的 X)或 FDIV(用于负指数)作乘或除法。整个过程由子程序 DEC2FLT 完成。

产生的子程序包(包括 EXP2,EXP10,DEL2FLT)接收一个可由 8086 产生的两部分数字(组合式 BCD 有效数及整型指数),然后将其变换为 8087 内部所用的浮点实数。

▲实数到十进制数的规一化函数

获得 8087 内使用的数字后,便可开始计算。如果无空间可用,可将其作为临时实数存储在公共内存区中(FSTP 指令用于此)。

将一个数字作为一个组合式 BCD 字符串存储时,FBSTP 指令首先将其四舍五入至其最接近的整数。如果此

数字过大,无法用一个 BCD 字符串表示,8087 便不能存储该数。如果该数字太小,作四舍五入时将损失有效精度。故在使用 FBSTP 指令之前,必须首先确认存储在寄存器中的数字是否处于合理范围内。对于打算要其精确到 10 位十进制数字的一个数，其二进制小数点最好处于比特 32 处,大约处于整个 64 比特浮点数的中间。

如果数字的指数不在此范围中,则应改变指数。第一步是确定指数是多少。使用 FXTRACT 指令,将 8087 数据寄存器分为两个,一个保存有效位(ST),另一个保存原数的指数(ST①)。我们所感兴趣的部分在 ST(1)中。

计算的第一步是确定要求的指数与当前指数之间的距离。可使用 FSUB 来获知。

一旦获得了距离,还不能马上用其作为规一化因子施加于原指数上,因为显示此数时,应以科学表示法来告知用户:

+1.234560000E+00

而如果指数是 2 的幂次就无法实现。办法是使 8087 产生一个整数,然后算出将其变成一个整数过程中移动了多少次 10 的幂。

我们要做的是,将当前以 2 的幂表示的距离变成以 10 的整数幂表示的距离。这两个值之间的关系由下述规则表示:

$$2^x = 10^{X \cdot \log10\ 2}$$

或 $$2^X = 10^{X / \log2\ 10}$$

从该恒等式得到的第二个关系是:

$$\log_a b = 1 / \log_b a$$

无论用哪种方法进行计算,均可确定用于生成正确规一化因子所需的 X 值(对于 10 到 X 的表达式)。可通过 FLDLG2 后跟 FMUL 或通过 FLDL2T 后跟 FDIV 来生成

此因子.由于我们需要最接近的整数,所以再使用FRNDINT对该数四舍五入,得到以10为底的指数.

给出此指数后,便可由 EXP10 计算出 10 到 X 的规一化因子,从而将实数变换成整数(使用 FMUL). 10 的指数通过 FIST(存储整数)返回,有效部分通过 FBSTP(存储组合式BCD)返回.

一旦组合式 BCD 数存储在内存中,便可使用二进制到十六进制显示程序(如 BIN2HEX)显示出数字,因为它们看起来与十六进制数字一样.

```
        page    60,132                  ; wide listing
        public  dec2flt                 ; DECLARE LIBRARY ROUTINE
        public  flt2dec                 ; DECLARE LIBRARY ROUTINE
        public  exp10                   ; DECLARE LIBRARY ROUTINE 10**x
        public  expE                    ; DECLARE LIBRARY ROUTINE e**x
        public  expY                    ; DECLARE LIBRARY ROUTINE y**x
        public  exp2                    ; DECLARE LIBRARY ROUTINE 2**x
;-----------------------------------------------------------------------
;       I M P L E M E N T A T I O N
;
code    segment para public 'code'
        assume cs:code,ds:code,es:code,ss:code
;***********************************************************************
;
; DEC2FLT--Convert decimal integer with exponent to floating
;          point real number. Accept exponent and pointer to
;          packed BCD string on stack. Return result in ST(0)
;
; Use:     push        offset (tbyte ptr packed BCD)
;          push        exponent
;          call        dec2flt
;
; Requirements:        3 stack locations
; Notation:     N ...... exponent for 10**N
;               S ...... significand portion of loading real
;
d2fltd  struc
d2fltbp dw      ?               ; old base pointer
        dw      ?               ; return address
d2fltex dw      ?               ; exponent
d2fltpd dw      ?               ; pointer to packed BCD
d2fltd  ends
dec2flt proc    near
        push    bp
        mov     bp,sp                   ; address parameters
        cmp     word ptr [bp].d2fltex,0 ; check sign of exponent
        jz      d2flt_nxp               ; if zero, no 10**N needed
```

```
        pushf                           ; save sign of exponent
        jg      d2flt_pos               ; if positive, start 10**N
        neg     word ptr [bp].d2fltex   ; ... else make exp positive
d2flt_pos:
        FILD    word ptr [bp].d2fltex   ; get exponent of 10
        call    exp10                   ; calculate 10**N
d2flt_nxp:                              ; enter here if exp is 0
        push    si
        mov     si,[bp].d2fltpd         ; get pointer to packed BCD
        FBLD    tbyte ptr [si]          ; ST => S; ST(1) = 10**N
        pop     si
        popf                            ; restore exponent's sign
        jz      d2flt_end               ; done if exp is 0
        jl      d2flt_neg               ; if negative, do divide
        FMUL                            ; ST => significand * 10**N
        jmp     d2flt_end               ; and done
d2flt_neg:
        FDIVR                           ; ST =>
d2flt_end:
        pop     bp                      ; restore bp
        ret     4
dec2flt endp
;***********************************************************
;
; FLT2DEC--Convert floating real to decimal integer with exponent.
;         ST(0) contains number to be converted. Stack contains
;         number of binary digits desired and pointer to 10's
;         exponent location.
;         Returns with ST(0) converted to an integer and writes
;         the 10's exponent to the designated location.
;
; Use:    push        sig_digits
;         push        offset (word ptr to exponent)
;         call        flt2dec
;
; Requirements:       4 stack locations
; Notation:     R ...... Real number to display
;               N ...... Exponent of 10 to convert R to integer
;               I ...... Integer portion of resultant number
;               n(N) ... nearest integer of N
;
f2decd  struc
f2deccw dw      ?               ; original control word
f2decbp dw      ?               ; old base pointer
        dw      ?               ; return address
f2decex dw      ?               ; pointer to exponent
f2decsd dw      ?               ; number of significant binary digits
f2decd ends
; *** check rounding control at this point--use other ?? ***
f2decct equ     03BFh           ; new control word--round nearest
flt2dec proc    near
;       Set up the 87's control word and open storage on the stack
;
        push    bp              ; save old base pointer
        sub     sp,(f2decbp-f2decd)     ; make storage on the stack
        mov     bp,sp           ; address new structure
        push    ax              ; save AX
        mov     ax,f2decct      ; push new control word on stack
```

```
        push    ax
        FSTCW   word ptr [bp].f2deccw
        FLDCW   word ptr [bp-4] ; set to round to nearest INT
        pop     ax              ; clean up stack
        pop     ax              ; restore AX
;       Find N for 10**N to convert to integer
;
        FLD     ST(0)           ; duplicate R (preserve until end)
        FXTRACT                 ; ST(1) => exponent portion of R
        FSTP    ST(0)           ; ST => exponent portion of R
        FISUBR word ptr [bp].f2decsd ;sigdig - exp=# of scale digits
        FLDL2T                  ; ST => log2 (10), ST(1)=>scale
        FDIV                    ; ST => scale/log2 (10)=N
        FRNDINT                 ; ST => n(N)
;       Store nint (N) as exponent & calculate 10**nint(N)
        push    si
        mov     si,[bp].f2decex   ; get pointer to exponent
        FIST    word ptr [si]     ; store base 10 scale
        FWAIT
        neg     word ptr [si]     ; direction to move dec. point
        pop     si
        call    exp10             ; calculate 10**N (scale)
;       ST(1) now has R (the original real #)--scale it
;
        FMUL                            ; ST => R*10**N=Integer
        FLDCW   word ptr [bp].f2deccw   ; restore control word
        add     sp,(f2decbp-f2decd)     ; resize stack to original
        pop     bp                      ; restore BP
        ret     4                       ; clear stack on return
flt2dec endp
;***********************************************************
;
; EXP10--Calculate 10 to the power of ST(0)
;         Return result in ST(0)
;
; Uses formula: 10**N = 2**(N*log2(10))
;
; CALLS:        EXP2
;
; Requirements:         3 stack locations
; Notation:     N ...... exponent for 10**N
;               X ...... equivalent exponent for 2**X
;               n(x) ... nearest integer of X
;               f(x) ... fractional part of X
;
exp10   proc    near
        FLDL2T                  ; ST > log2 (10); ST(1) => N
        FMUL                    ; ST => N * log2 (10) => X
        call    exp2            ; raise 2 to ST power ...
        ret                     ; for 10 ** N
exp10   endp
;***********************************************************
;
; EXPE--Calculate E to the power of ST(0)
;       Return result in ST(0)
;
; Uses formula: E**N = 2**(N*log2(E))
;
; CALLS:        EXP2
```

```
;
; Requirements:         3 stack locations
; Notation:     N ...... exponent for E**N
;               X ...... equivalent exponent for 2**X
;               n(x) ... nearest integer of X
;               f(x) ... fractional part of X
;
expe    proc    near
        FLDL2E                  ; ST > log2 (e); ST(1) => N
        FMUL                    ; ST => N * log2 (e) => X
        call    exp2            ; raise 2 to ST power ...
        ret                     ; for E ** N
expe    endp
;************************************************************
;
; EXPY--Calculate Y [ST(0)] to the power of N [ST(1)]
;        Return result in ST(0)
;        ST(1) (value of N) is lost!
;
; Uses formula: Y**N = 2**(N*log2(Y))
;
;       **** Y MUST BE POSITIVE ****
;
; CALLS:        EXP2
;
; Requirements:         3 stack locations
; Notation:     N ...... exponent for Y**N
;               X ...... equivalent exponent for 2**X
;               n(x) ... nearest integer of X
;               f(x) ... fractional part of X
;
expy    proc    near
        FYL2X                   ; ST => N * log2 (Y) => X
        call    exp2            ; raise 2 to ST power ...
        ret                     ; for Y**N
expy    endp
;************************************************************
;
; EXP2--Calculate 2 to the power of ST(0)
;        Return result in ST(0)
;
; Requirements:         3 stack locations
; Notation:     X ...... exponent for 2**X
;               n(x) ... nearest integer of X
;               f(x) ... fractional part of X
;
exp2d  struc
exp2cc dw       ?               ; condition codes
exp2cw dw       ?               ; original control word
exp2bp dw       ?               ; old base pointer
       dw       ?               ; return address
exp2d  ends
exp2ct equ      03BFh           ; new control word--round nearest
exp2   proc     near
;       Set up the 87's control word and open storage on the stack
;
        push    bp              ; save old base pointer
        sub     sp,(exp2bp-exp2d)       ; make storage on the stack
        mov     bp,sp           ; address new structure
        push    ax              ; save AX
```

```
        mov     ax,exp2ct          ; push new control word on stack
        push    ax
        FSTCW   word ptr [bp].exp2cw
        FLDCW   word ptr [bp - 4] ; set to round to nearest INT
        pop     ax                 ; clean up stack
        pop     ax                 ; restore AX
        Start processing the number now.
        FLD     ST(0)              ; ST => ST(1) => X for 2**X
        FRNDINT                    ; ST => n(X); ST(1) => X
        FXCH                       ; ST => X; ST(1) => n(X)
        FSUB    ST,ST(1)           ; ST => f(X); ST(1) = n(X)
        FTST                       ; set condition codes
        FSTSW   word ptr [bp].exp2cc      ; store CC's
        FWAIT
        and     byte ptr [bp + 1].exp2cc,45h     ; mask all but CC's

        cmp     byte ptr [bp + 1].exp2cc,1        ; test for negative
        ja      exp2_err                 ; NAN or infinity => error
        je      exp2_neg                 ; fractional part is minus
;
        F2XM1           ; ST => (2**f(X)) - 1; ST(1)=n(X)
        FLD1            ; ST => 1; ST(1) => (2**f(X)) - 1; ST(2)=n(X)
        FADD            ; ST => 2**f(X); ST(1)=>n(x)
        FLD1            ; ST => 1; ST(1) => (2**f(X)) - 1; ST(2)=n(X)
        FADD            ; ST => 2**f(X); ST(1) => n(X)
        FSCALE          ; ST => 2**(X) => 2**(N*log2(?)) => ?**N
        FSTP    ST(1) ; ST => ?**N; ST(1) => restored
        jmp     exp2_mer ; merge
;
exp2 neg:
        FABS            ; ST => 1 - f(x); ST(1)=n(X) + 1
        F2XM1           ; ST => (2**(1 - f(x))) - 1; ST(1)=n(X) + 1
        FLD1            ; ST => 1; ST(1) => (2**(1 - f(x))) - 1
        FADD            ; ST => 2**(1 - f(x)); ST(1) => n(X) + 1
        FXCH            ; ST => n(X) + 1; ST(1) => 2**(1 - f(x))
        FLD1            ; ST => 1; ST(1)=n(X) + 1
        FSCALE          ; ST => 2**(n(x) + 1); ST(2) => 2**(1 - (f(x))
        FDIVRP ST(2),ST; ST(1) => 2**(n(X) + 1)/2**(1 - f(x))
        FSTP    ST(0) ; ST => 2**(n(x) + 1 - 1 + f(x) => 2**(X)
;
exp2_mer:
        clc                              ; no errors
exp2_out:
        FLDCW   word ptr [bp].exp2cw     ; restore control word
        add     sp,(exp2bp-exp2d)        ; resize stack to original
        pop     bp                       ; restore BP
        ret
exp2_err:
        stc                              ; errors occurred
        jmp     exp2_out
exp2    endp
;**********************************************************
code    ends                             ; end code segment
        end
```

上面提供的DEC2FLT.FLT2DEC及指数计算程序EXP2，EXP10，EXPE，EXPY用于 debug和 I/O 的程序,不仅能使读者了解 8087 如何工作,还能帮助用户开发使用 8087 的应用程序．当使用了 8087 的强大功能后,傅里叶变换、三角函数分析等问题都是不难解决的．

［Ｇｅｎｅｒａｌ　Ｉｎｆｏｒｍａｔｉｏｎ］
书名＝ＭＳ－－ＤＯＳ高水平程序设计
作者＝ＢＥＸＰ
页数＝６３５
下载位置＝ｈｔｔｐ：／／ｂｏｏｋ７．ｓｓｒｅａｄｅｒ．ｃｏｍ／ｄｉｓｋｊｓｊ／０２／１ｂ４４／！００００１．ｐｄｇ