DOS 设备驱动程序

计算机早期的程序员直接在硬件水平上编写各类程序。每个程序都必须直接涉及卡的读写器、打印机和其它一些与计算机相联系的设备。于是程序员不得不掌握有关处理错误类型、处理正确输入和输出等方面的各种知识。

随着计算机的发展，程序员发现这种重复工作所花的时间完全是没有必要的。外部设备的处理程序渐渐成为加到程序上的标准项目。不久，这些处理程序被选进了原始操作系统之中，在这个系统内，所有程序都可以使用设备处理程序或驱动程序的相同配置。

在最早的操作系统中，不同的设备驱动程序编码组成了该系统的整个部分，并以复杂的方式与系统其它部分之间发挥作用。要使设备驱动程序更加独立，系统程序员可在启动过程中，把设备驱动程序作为必要的部分来装入。

尽管大多数重要的操作系统都有一定的灵活性，但只有 DOS 才为用户提供了最大程度的灵活性，可方便地安装各类驱动程序。许多微机操作系统都需要冗长的修补来完成提前编写的驱动程序和系统配置文件所能完成的事情。

对于大多数用 DOS 进行工作的人来说，他们与驱动程序的唯一联系就是从发布磁盘来安装它们，并在 CONFIG.SYS 文件中创建所需项（稍后将介绍这些页）。用户则只需遵循程序所给出的指导，一行一行地进行修改。在有些系统中，甚至不需要这种联系，因为安装程序可以自己完成这种改变。

最后大多数程序员开始对他们自己的技术相当信赖并决定编写一个设备驱动程序。富有经验的汇编语言程序员发现这个任务相对较容易—他们只需遵循一个公式般的约定。如果能正确遵守此约定，驱动程序就能正常运行。许多程序员没有成功，因为他们没有坚持严格的规则，即执行驱动程序必须做的事情，以及必须以某种方式使用驱动程序。

本章的目的就是要说明编写设备驱动程序的方法。

为了获得最高速度和编程的便利性，在 PC 机上的设备驱动程序通常用汇编语言来编写。尽管设备驱动程序的一些部分能用 C 语言编写，但在获得正确结构及减少额外开销方面会产生问题。设备驱动程序有严格的结构，而用 C 编译的模型只能提供功能性的服务。若使用 C 来为驱动程序建立功能，就必须从汇编语言部分开始，该部分获得初始控制，然后调用所需的 C 例程。不能调用 C 的库函数，因为其中的许多都是 DOS 功能（不允许驱动程序调用 DOS 服务。本章的“驱动程序初始化”一节将更多地介绍这方面的内容）。

用汇编语言编写整个驱动程序有 3 个理由：

- 设备驱动程序是所有的设备访问的核心，所以必须把它紧凑地编程以节约执行时间和内存空间。
- 驱动程序布局是严格定义的；只有汇编语言才能提供所需的布局控制。
- 必须在特定的时间去操纵特定的 CPU 寄存器，从 C 中很难做到这一点。

用 C 来编写设备驱动程序在 UNIX 世界中是很普遍的，但接口要求却不一样。汇编语言前端链接并控制了操作系统。用 C 编写驱动程序则可能很好笑（并且是失败的）。

在建立设备驱动程序之前，应该知道设备驱动程序的使用方法和它的工作方式，然后才能建立该程序。

知道了驱动程序，就要建立驱动程序的外壳程序，在其中增加另外的程序来为真实设备编写真实驱动程序。首先让我们看看驱动程序的类型以及它们的工作方式。

12.1 驱动程序的类型

设备驱动程序有两个基本类型——字符设备和块设备，二者是根本不同的。现在先看看

它们的差异。

12.1.1 字符设备驱动程序

字符设备是面向字节的设备，如打印机端口或串行端口。所有与设备的通信都发生在一个个字符的基础之上。

字符设备的输入和输出在两个基本模式之一内进行：处理过的和原始模式。在处理过的模式中，DOS 从驱动程序每次要求一个字符并在内部缓冲输入。特殊的键组合如 Ctrl-C 和回车键由 DOS 处理。在原始模式中，DOS 不缓冲数据，也不寻找和对 Ctrl-C 或回车键产生反应。但 DOS 请求输入固定数目的字符，该请求直接传递给驱动程序；回答则是由驱动程序所读取的字符组成的。

字符设备名（如 CON、AUX 和 LPT）能达到 8 个字符那么长（像文件名一样）。8 个字符的限制是因为设备头的驱动程序名只有 8 个字符（“设备头”一节将讨论头）。这个限制在 DOS 中是故意设计成这样的，目的是使用相同的 I/O 例程来处理命名的文件和设备。它也使访问与系统中已存在的驱动程序具有相同名字的文件成为不可能;如果遇上了与设备同名的文件，就由这些例程来访问驱动程序。

12.1.2 块设备驱动程序

块设备处理磁带和磁盘上的数据块。每次访问块设备都把数据以合适的块大小进行传递。在块设备中，没有与字符设备驱动程序那样的处理过的模式和原始模式。

驱动器字母分配给了块设备，块设备变成了一个或多个系统逻辑驱动器（如 A、B 和 c）。单个块设备驱动程序能操纵多个硬件单元或把一个硬件单元映射成多个逻辑驱动器。每个逻辑驱动器都有基本文件系统这样的结构，它包含一个 FAT（文件分配表）和根目录（这些结构的其它情况，请看第 2 章和第 8 章）。

12.2 设备驱动程序的工作方式

应用程序调用 DOS 的 Int 21h 来执行任何 I／O 功能时，设备驱动程序会涉及几乎每种情况（除非这些系统功能用作扩展出错处理）。让我们看一个实例，在其中我们要向磁盘上的文件里输入内容。

不论是把文件写操作当作 DOS 调用，还是使用函数库中的函数，应用程序都要设置寄存器，并调用 Int 21h（DOS 服务例程）来操纵磁盘 I/O。服务例程获得控制时，它反过来设置并调用 Int 26h（绝对的磁盘写）。Int 26h 在内存的保留区域设立请求头（驱动程序的命令缓冲区），并调用操纵磁盘的设备驱动程序策略例程。该例程只是保存请求头的地址，并把控制返回到中断处理程序。

然后，DOS 调用驱动程序的中断部分（它的名字，像策略例程一样并不反映它的功能）。这个部分读请求头。并确定请求的内容。接着，它把控制传递给合适的内部例程，并调用 BIOS 磁盘写功能—Int 13h来执行磁盘写操作。完成磁盘写操作后，控制就从链中返回到应用程序，并调整状态码来反映每个调用例程所希望的状态。

图 12．1 显示了这些事件的系列。每一步都涉及把控制传递给更低水平的例程，直到发生磁盘写操作。

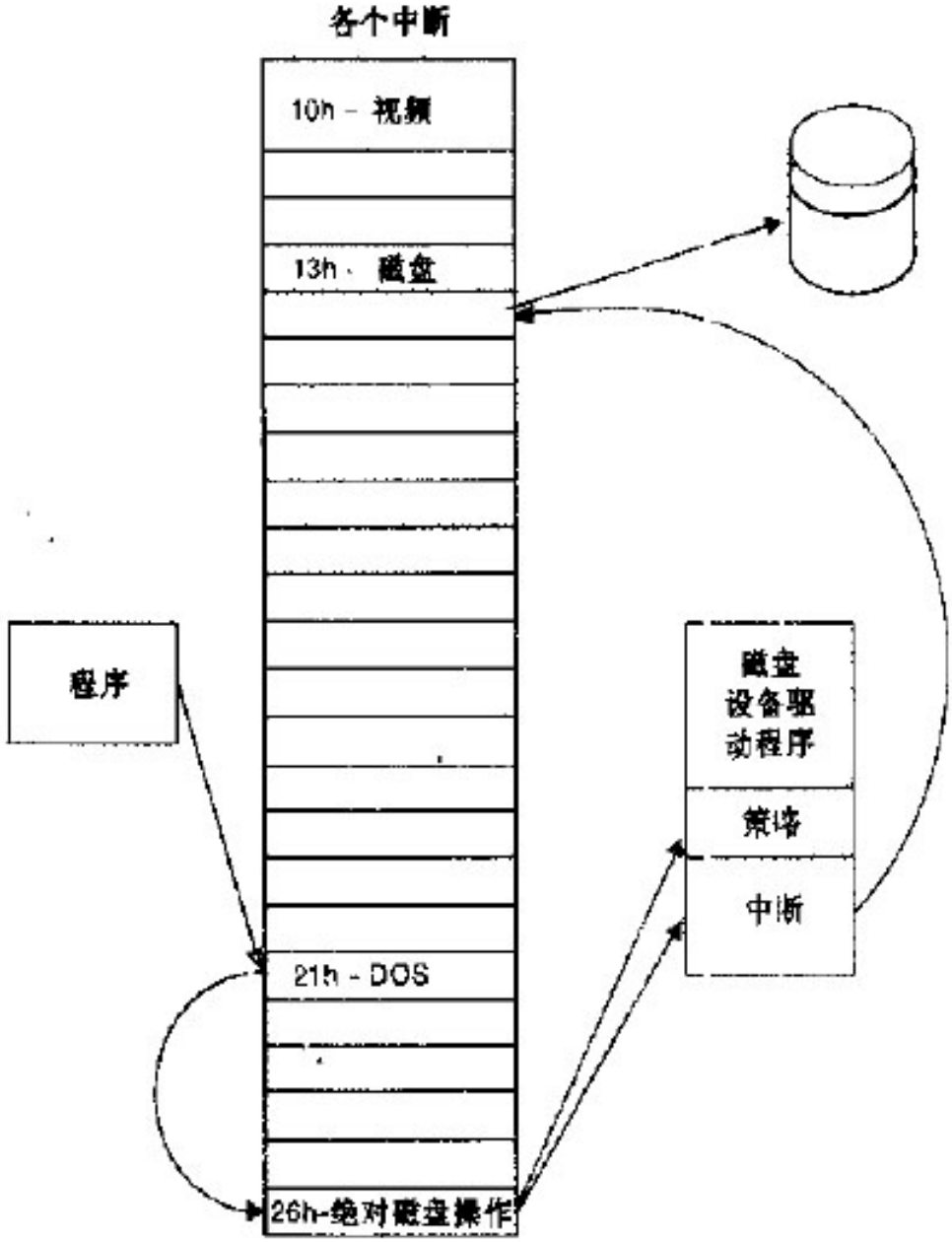


图 12．1 请求磁盘写

这些步骤都代表了磁盘的单一操作；这样对驱动程序的调用可能很多。若在文件调用中使用高级语言源代码，可能需要对磁盘进行另外的访问来在磁盘上阅读FAT、分配空间并更新参数。这样的话，驱动程序可能极忙。

尽管这个例子很复杂，但可以用对 BIOS 的调用来操纵它。不必担心“地下又黑又暗”的接口细节（硬件操作）。在每个 PC 机和兼容机上，都假定 BIOS 能够使所有设备看起来像一套标准的设备。但加上一个自定义的片段又怎么样呢？图 12．2 显示了所发生的事情。驱动程序必须直接操纵新的硬件，因为没有 BIOS 来处理接口的细节。

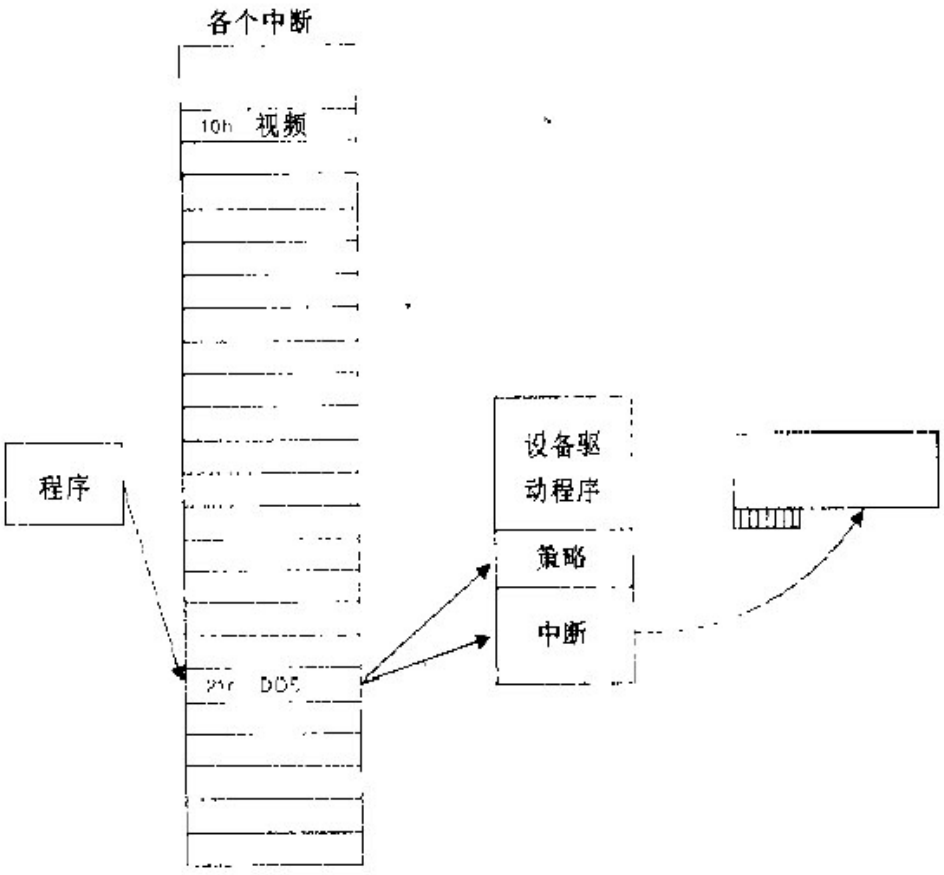


图 12．2 常规设备驱动程序

向系统添加一块插板，即加上一项新功能的话，必须加上一个新的设备驱动程序。如加上 CD-ROM 驱动器、鼠标、局部网络卡或音乐合成器一类的硬件，则加上的硬件就是设计原始 PC 系统软件时未曾考虑过的设备。MS-DOS 没有软件来处理鼠标设备。与鼠标一起所带的驱动程序必须直接与硬件打交道。复杂性便从这里开始。要把这个硬件与系统连接起来，就需要驱动程序。

通过驱动程序进行磁盘访问的实例（参看图 12．1）隐藏了一个重要事实：BIOS 已经注意到硬件的细节，并处理所有的定时、位处理等操作。对于一个自定义的添加设备驱动程序，必须直接处理有关硬件的细节问题。

向系统添加硬件的新片段，并为它编写驱动程序时，必须考虑到所有的接口细节。例如，如果是添加一个模拟数字转换器，以便读取某个仪器的数据，就必须在硬件水平上服务于芯片。如果要观察定时限制或处理其它问题就必须了解它们。

有关特殊插板的工作接口的介绍超出了本书的范围。要编写好一个成功的接口，必须具有想运行的硬件的详细知识。有一点考虑不到也是不可接受的。本书提供了一个框架能让你的驱动程序运行起来，利用它，既可以为加进系统中的硬件新片段编写驱动程序也可以为已存在的硬件编写驱动程序。

12.3 设备驱动程序的结构

设备驱动程序由三个部分组成：设备头、策略例程和中断例程。在 DOS 2．0 版中驱动程序必须是没有起始地点（ORG0，或没有语句）的内存映象（或 COM 程序）。而且它必

须编写成一个 FAR 程序。EXE2BIN 程序可把汇编过的驱动程序变成一个映象文件，而且系统在引导操作过程中安装该映象。常规情况下，驱动程序常带有扩展名.SYS（或者有时是.BIN）。文件扩展名变成.SYS，可防止人们偶尔把驱动程序当作一个程序来执行。

在 DOS3.0 版和更高的版本中，驱动程序以 EXE 格式而成为目标文件。操作系统同样地能正确安装它们。但要保持与 DOS 2.0 版的向下兼容性，大多数编写驱动程序的人都用 COM 格式来工作（DOS 1.0 版没有准备可安装的驱动程序）。本章的实例都是 COM 类型的驱动程序。

本节讨论驱动程序的结构。首先看看设备头——驱动程序的最重要部分。

表 12.1 驱动程序属性字

位	意　义
FEDCBA98 76543210	
....1	标准输入设备
....0	非标准输入设备
....1.	字符设备：标准输出设备
	块设备：能处理 32 位扇区数（V4 独有）
....0.	字符设备：非标准输出设备
	块设备：不能处理 32 位扇区数（V4 独有）
....1..	NUL 设备
....0..	非 NUL 设备
....1...	时钟设备
....0...	非时钟设备
....1....	驱动程序服务 Int 29h
.....000 000.....	在 V3．2 之前保留（设置成 0）
....0.....	在 V3．2 和更高版本中保留（设置成 0）
.... .1......	驱动程序支持通用的 IOCTL（V3．2 和更高版本）
.... .0......	驱动程序不支持通用的 IOCTL(V3．2 和更高版本)
.....000 0.......	在 V3．2 和更高版本中保留（设置成 0）
....0...	支持 OPEN/CLOsE / 可移动的介质（V3 和更高版本）
....1...	支持 OPEN/CLOSE / 可移动的介质（v3 和更高版本）
...0....	保留（设置成 0）
..1.....	字符设备：设备不支持“输出直到忙”操作
	块操作：IBM 块格式
..0.....	字符设备:设备支持“出直到忙”操作
	块操作：非 IBM 块格式
.1......	支持 IOCTL
.0......	不支持 IOCTL
1.......	字符设备
0.......	块设备

12.3.1 设备头

设备头是一个 18 个字节的区域，被划分成五个域（见图 12．3）。

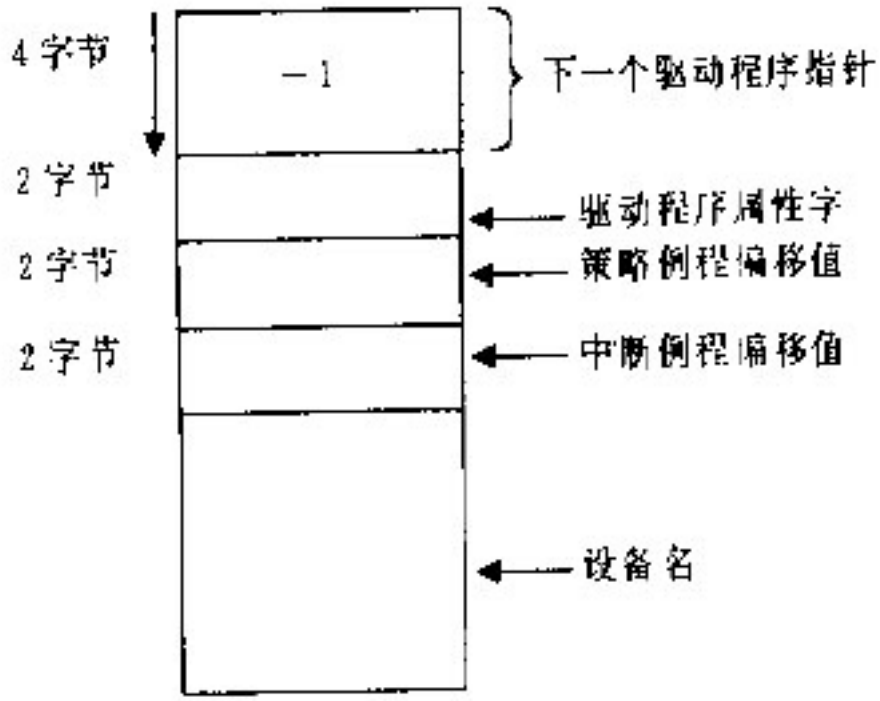


图 12.3 设备头的组成

下面介绍这五个域的含义。

- 下一个驱动程序指针。4 个字节初始化为-1（FFFFFFFFh）。DOS 用这个域装入驱动程序表中下一个驱动程序的指针。表中的最后一个驱动程序标记为-1（只有该指针的偏移值部分必须是-1；而段值部分则可以为 0）。
- 驱动程序属性字。这两个字节定义了驱动程序特性（见表 12.1）。
- 策略例程偏移值。它是驱动程序内策略例程的两字节偏移值。
- 中断例程偏移值。它是驱动程序内中断例程的两个字节偏移值。
- 设备名。若设备为字符设备，后面就会出现 8 个字符、向左边对齐、不足部分补空格的设备名。如果该设备名与已存在的设备的名字相同，新的驱动程序就会取代已存在的设备。如果该设备为块设备，本域的第一字节就是与驱动程序相关的逻辑驱动程序器号；其它字节被忽略（DOS 的一些版本还包括可引导块设备的特殊信息）。

DOS 在初始化自己后，便建立了标准设备驱动程序的链表；在每个驱动程序中的下一个驱动程序的指针提供了链表中下一个驱动程序的地址（见图 12.4）。链中的最后一个驱动程序有一个-1 指针来指示链的结束。

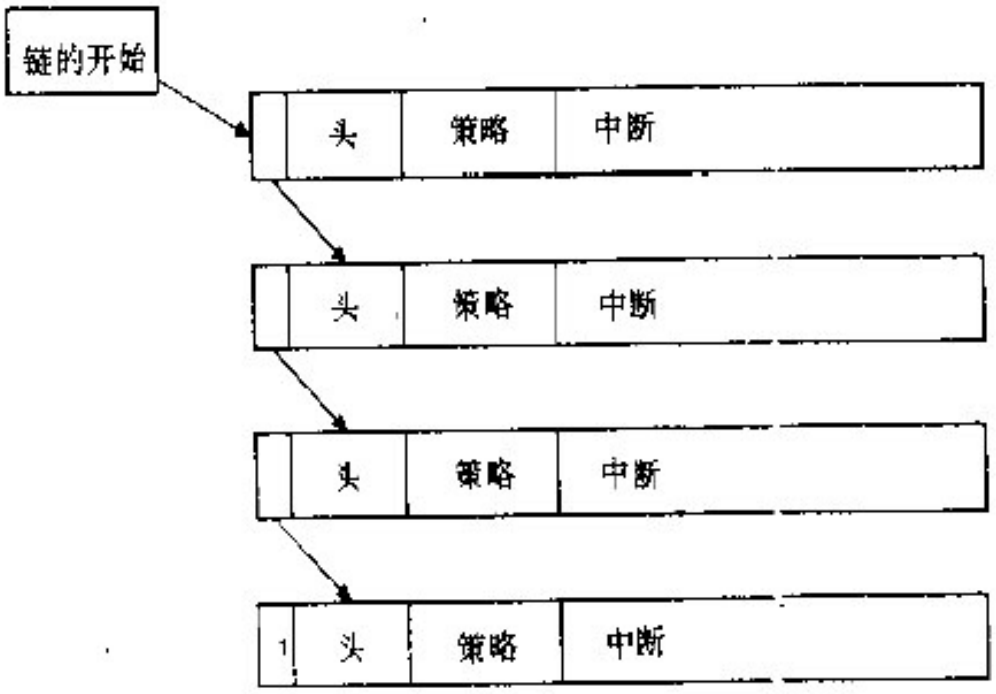


图 12.4 驱动程序链

DOS 在最后读取 CONFIG.SYS 文件时，已建立起了一条驱动程序链。新的驱动程序会加到链头上（见图 12.5）。

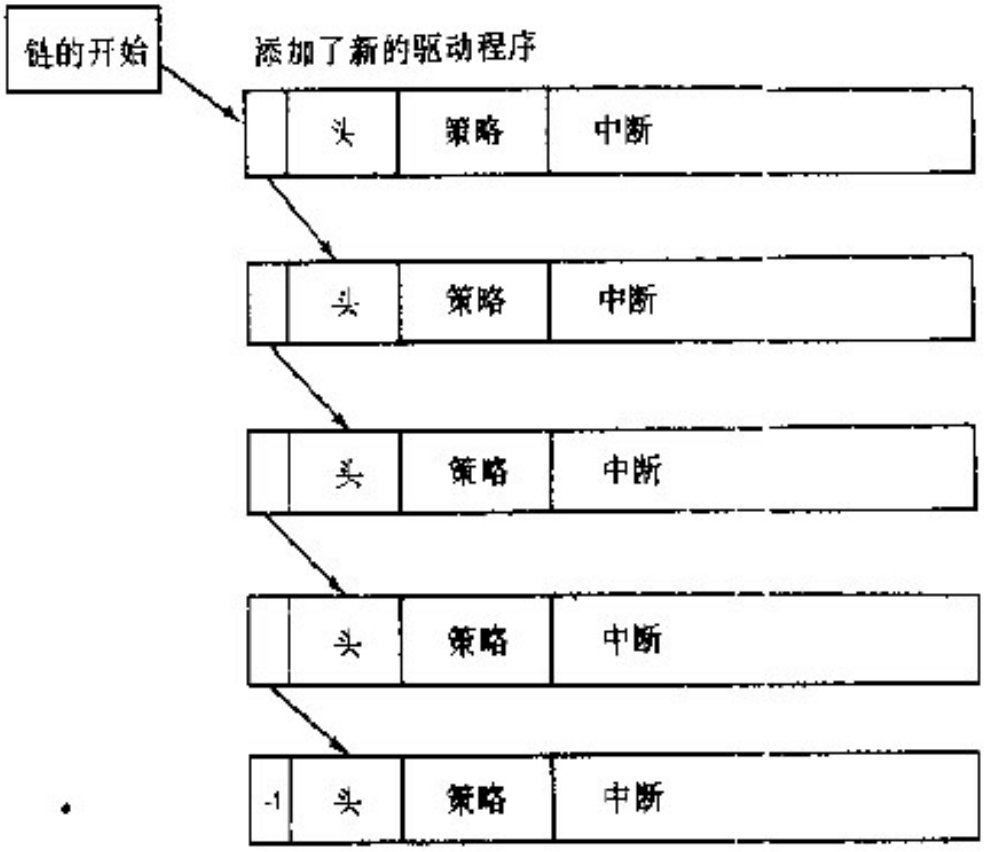


图 12.5 向驱动程序链添加一个新的驱动程序

DOS 先寻找字符设备驱动程序，它搜索驱动程序链找到与所请求的名字相匹配的驱动程序名。 DOS 从驱动程序链表的起点开始检查驱动程序名字看看是否与所请求的名字匹配。若不是，则查寻下一个驱动程序指针域来找到表中的下一个驱动程序，并且 DOS 也在那里检查。DOS 在链中检查每个驱动程序（它跳过块设备驱动程序）直到发现所需的驱动程序或链表的末端（在下一个驱动程序指针域中标记为-1）

新的驱动程序总是加到链表的开始（参见看图 12.5）。然后，DOS 搜寻驱动程序时，它首先检查新的驱动程序。如果加上的驱动程序，其名字与已存在的驱动程序名字相同（如加上一个新驱动程序名字为 PRN——与打印机驱动程序名字相同），则新的驱动程序将“取代”已存在的那一个，因为一次搜索总会导致使用新的驱动程序，而不是旧的同名的驱动程序。

ANSI. SYS 驱动程序包含在 CONFIG.SYS 文件中时都是以这种方式来工作的。它与控制台驱动程序有相同的名字（CON）;把 ANSI. SYS 驱动程序加到驱动程序链上时，无论何时进行控制台操作，都会首先找到该驱动程序。然后所有控制台操作都通过 ANSI. SYS 驱动程序而不是通常的 DOS 控制台驱动程序来进行。

要说明建立驱动程序及用它进行工作的方式，让我们建立一个实际例子（或至少是真实驱动程序的工作外壳程序）。列表 12.1 就是一个叫做 drvr.asm 的真实驱动程序头。在本章后面将介绍把这个驱动程序汇编成实际上不做任何事情的驱动程序。该驱动程序是一个可添加更多实际应用程序的外壳程序。

列表 12. 1

```
CR              EQU     0Dh             ;Carriage return
```

```
LF          EQU     0Ah             ;Line feed
MAXCMD      EQU     16              ;DOS3. 0，12DOS2. 0
ERROR       EQU     8000h           ;Set error bit
BUSY        EQU     0200h           ;Set busy bit
DONE        EQU     0100h           ;Set completion bit
UNKNOWN     EQU     8003h           ;Set unknown status
cseg        segment public'code'    ;Start the code segment
            org     0               ;Zero origin
            assume cs:cseg, ds:cseg, es:cseg
```

该驱动程序的第一部分由指示汇编程序的伪指令组成。首先，为获得程序的便利性，要定义许多常量如 CR（回车）、LF（换行）、MAXCMD（最大命令号：16 用于 V3.0 和 V3.1，12 用于(V2.x)，等等。这些定义可在继续工作时简化编程。

我们在前面已提到，列表 12.1 已定义成 0 起点（ORG 0）的代码段。所有段寄存器都设置成与代码段相同，以便能把驱动程序作为一个二进制映象（COM 格式）文件来汇编。

影响内存的驱动程序的第一部分是以标号 drvr 开始的，在这里将它声明为一个 FAR 程序。这一点很重要，因为 DOS 要用一个 FAR 子例程请求来调用所有的驱动程序。FAR 子例程请求能跨越段界限——把返回地址（段和偏移值地址）放到堆栈上作为调用的一部分。声明它是一个 FAR 程序以后，就能保证汇编程序使用 FAR 返回（它能获得离开堆栈的段和偏移值地址）来把控制返回给 DOS。

头的第一个域（一个双字，或 8 个字节）原始值被声明为-1。DOS 在驱动程序链中把这个字设置成下一个驱动程序的地址。然后把属性字设置成 8000h，指示该字符设备驱动程序没有特殊功能(参看表 12.1)。这后面是驱动程序的策略指针（只有偏移值）和中断程序，然后是驱动程序的 8 个字节的名字。

驱动程序头对于正常驱动程序操作是很关键的。DOS 在需要某个驱动程序时，它先检查属性字节看看该驱动程序能做的事情，然后再用策略和中断指针来定位这些例程。如果驱动程序头不正确，驱动程序在启动之前就会失败。因为驱动程序头是登记式的，且汇编程序进行登记工作，所以还是让我们先看看策略例程。

12.3.2 策略例程

策略例程与通常所提到的“策略”没有什么关系——它不是试图找到最好的途径来驱动设备或诸如此类的其它事情。可以只用 5 行程序来编写策略例程；它的目的就是要“记住”操作系统在内存中分配给设备的请求头（RH）的位置。RH 有以下两种功能:

- 它是 DOS 内部操作的数据区。
- 它是一个通信区，在其中，DOS 告诉驱动程序要做的事情，而驱动程序则要汇报操作的结果。

请求驱动程序输出数据时，数据地址由 RH 提供。驱动程序则执行输出任务，然后在请求头中设置标志或状态字节未指示完成。

DOS 要调用驱动程序时，请求头就建立在内存的保存区域中，其地址传递给 ES:BX 中的策略例程。尽管对驱动程序的每次调用可能都有一个新地址,但实际上这些地址都是相同的，策略例程把这个值保存起来以备驱动程序的中断例程将来使用。

请求头的长度会改变，但却总有一个固定的 13 字节的请求头（有时叫作请求头的“静态部分”）。请求头的结构见表 12.2。

表 12.2 请求头的开始部分

字书偏移值	域长度	意　　义
00h	字节	请求头的长度
01h	字节	单元代码：块设备的设备号。
02h	字节	命令代码：送给驱动程序的最近命令号。
03h		状态：每次调用后驱动程序设置的状态代码 。如果设置了15 位，则错误代码在低顺序的 8 个位中。0 状态代码意味着已成功地结束。
05h	8 个字节	保留起来以备 DOS 使用
0Dh	变量	驱动程序请求的数据

请求头中的大多数域已在表中解释清楚。但状态字（字节 03-4h）需要说明一下。

状态字把请求的完成状态传递回 DOS 所用格式见表 12.3。

表 12.3 请求头的状态字

位	意义
FEDCBA98 76543210	
.... 00000000	写保护违反错误
.... 00000001	未知的单位错误
.... 00000010	驱动程序没准备好错误
.... 00000011	未知
.... 00000100	CRC 错误
.... 00000101	不正确的驱动程序请求结构长度错误
.... 00000110	寻道错误
.... 00000111	未知的介质错误
.... 00001000	未找到扇区错误
.... 00001001	打印缺纸错误
.... 00001010	写失败
.... 00001011	读失败
.... 00001100	严重故障
.... 00001101	保留
.... 00001110	保留
.... 00001111	无效的磁盘改变
.......x	已完成
......x.	忙碌
.xxxxx..	保留
0.......	无错误

在状态字中设置出错位是为了指示驱动程序操作中发生的出错类型。出错码返回到状态字的较低 8 个位中。如未设置出错位，则出错码应设置为 0 以表示操作的圆满完成。

设置忙碌位是指示调用设备时，该设备正在忙碌。驱动程序完成操作时应设置已完成位。驱动程序在请求操作时设置以上那些位来指示状态;所有功能都要设置已完成位来指示已完成。

用于样本驱动程序的策略例程（drvr.asm）如列表 12．2 所示。

列表 12．2

```
rh_seg      dw      ?       ;RH segment address
rh_off      dw      ?       ;RH offset address
```

```
strategy:
        mov cs:rh_seg, es
        mov cs:rh_off, bx
        ret
```

列表 12.2 分配空间来保存请求头的段和偏移值。整个策略例程，仅由保存请求头指针（段地址在 ES 寄存器中，偏移值地址在 bx 寄存器中）的操作组成。为什么不做更多的事情？一个更突出的问题可能是“为什么有两个入口点？”为什么不把指针传递给 ES:BX 中的中断例程并用它来完成工作？

答案涉及操作系统的兼容性和内部机制。驱动程序结构设计成希望能与将来的扩展版兼容，该扩展版打算把 DOS 变成多任务的结构。操作系统运行多任务时，在单个请求被处理之前可能会输送多个请求给某个特定的驱动程序。换言之，这些请求可能不得不排队。

例如，如果要请求磁盘扇区的读操作，在第一个请求得到满足之前可能到达了多个请求。特别是如果所请求的扇区远离磁盘的当前位置，则更有可能会出现以上的情况。可以改进策略例程使它把多个请求排序，减少磁头移动，从而优化对磁盘设备的访问。但这些功能无一能用于 DOS V4.01 及更高版本。

DOS 是一个单用户、单任务的系统，所以它没有多个进程访问驱动程序的潜在能力但其结构却允许在这个方向上进行扩展（如果这类扩展确实是必需的）。

安全地保存了请求头的地址后，可以返回到 DOS 并等着调用中断例程：在单任务系统中它立即就会调用。

12.3.3 中断例程

驱动程序的重要部分叫做中断例程，由它来完成所有工作。采用这个名字并不恰当，因为它并不像一个中断，而且它以 RET 终止而不是以 IRET 终止。但这个名字却反映了：没有具体化的计划；它的意图就是中断处理程序要处理那些排成队列的请求。当某个设备要处理下一个任务时，它会中断 DOS，然后处理程序会把控制指向中断例程。但像策略例程的设计一样，这一想法在 DOS 上同样未具体实现。

中断例程包括 DOS 系统所需的 21 个功能那么多的代码（DOS2.0 版上为 13，DOS3.0 版则为 20，DOS5.0 版为 21）。无论何时调用设备驱动器程序，客观上都会获得请求头的地址并去查看其中的偏移值为 02h 处的字节，以便找到命令代码，该代码指示驱动程序所要执行的功能。

大多数驱动程序都会创建一个含有驱动程序功能指针的表。命令代码用作该表的索引，以定位所需的功能。列表 12.3 显示了样本驱动程序的调度表。

列表 12. 3

```
d_tDl:
    dw s_init           ;Initialization
    dw s_mchk           ;Media check
    dw s_bpb            ;BIOS parameter block
    dw s_ird            ;IOCTL read
    dw s_read           ;Read
    dw s_nrd            ;Nondestructive read
    dw s_inst           ;Current input Status
    dw s_infl           ;Flush input buffer
    dw s_write          ;write
    dw s_vwrite         ;Write with verify
```

```
    dw s_ostat          ;Current output status
    dw s_oflush         ;Flush output buffers
    dw s_iwrt           ;IOCTL write
    dw s_open           ;Open
    dw s_close          ;Close
    dw s_media          ;Removable media
    dw s_busy           ;Output until busy
```

这个表特别易于设计,因为汇编程序跟踪这些功能并自动把正确的偏移值地址插入该表中。这个驱动程序不支持DOS V3.0之后产生的特殊功能：通用的IOCTL；Get LogicalDevice（获取逻辑设备）;Set Logical Device（设置逻辑设备）和IOCTL Query（查询）。但这不是问题，因为大多数运行DOS的DOS旧版本的程序并不使用依赖于这些功能的调用。

中断例程的主体决定了要服务的请求的本质。它通过调度表来分别走向相应的功能。列表12.4显示了中断例程余下的部分。

列表12. 4

```
interrupt:
    cld                 ;Save machine state
    push es             ;Save all registers
    push ds
    push ax
    push bx
    push cx
    push dx
    push si
    push di
    push bp
    mov dx, cs:rh_ seg
    mov es, dx
    mov bx, cs:rh_off
    mov al, es:[bx]+2               ;Command Code
    xor ah, ah
    cmp ax, MAXCMD                  ;Legal command?
    jle ok                          ;Jump if okay
    mov ax, UNKNOWN                 ;Unknown command
    jmp finish
ok:
    shl ax, 1                       ;Multiply by 2
    mov bx, ax
    jmp word ptr [bx+d_tbl]
finish:
    mov dx, cs:rh_seg
    mov es, dx
    mov bx, cs:rh_off
    or ax, DONE                     ;Set the DONE bit
    mov es:[bx] + 3, ax
```

```
    pop bp                          ;Restore the registers
    pop di
    pop si
    pop dx
    pop cx
    pop bx
    pop ax
    pop dS
    pop es
    ret                             ;Back to DOS
```

中断例程开始时，先在堆栈中保存当前的机器状态。然后它从策略例程所保存的请求头指针的地方获得该请求头的指针。中断例程接着查看请求头中的偏移值02h处的字节，决定要做的情况。该例程接着检查一下，确定该命令是否是合法的：如果是，该例程就执行分支转列它完成该功能的地方。一个非法的命令（比最大的命令号还大）会导致驱动程序返回一个出错标志，设置这个标志即表示它是一个未知的命令。

当命令号小于 MAXCMD（这是驱动程序所支持的命令个数）时，驱动程序就把命令号乘以2（借助左移，这与乘以之是相同的）来在调度表中获得命令代码的偏移值。这一移位操作必不可少，因为在调度表中为每个表项保留了两个字节。然后把偏移值加到调度表的基地址上，然后驱动程序跳到指定的例程上。

功能结束时，驱动程序重新获得请求头的指针并在状态字中设置已完成位，表示操作已经完成。接下来，保存在驱动程序开始处的寄存器重新恢复，控制返回到DOS内核。

在查看每个独立的驱动程序功能之前，让我们先看看该样本驱动程序的余下部分代码。在任何驱动程序中，只有一些功能是必须实现的。在不需要某个功能的情况下，驱动程序可以只返回一个状态代码而不必做任何事情。一些人主张返回一个0代码来指示圆满的操作。其它人则建议返回一个错误代码 3（命令未知）。如果要编写自己的驱动程序来使用自己的软件，就要进行自己的选择。但是，如果编写一个驱动程序来代替已有的驱动程序，新返回的代码应该与原来的驱动程序返回的代码相同。

列表12.5提供了drvr.asm剩下的部分。

列表12.5

```
s_mchk:                 ;Media check
s_bpb:                  ;BIOS parameter block
s_ird:                  ;IOCTL  read
s_read:                 ;Read
s_nrd:                  ;Nondestructive read
s_inst:                 ;Current input status
s_infl:                 ;Flush input buffers
s_vwrite:               ;Current output status
s_ostat:                ;Current output status
s_oflush:               ;Flush output buffers
s_iwrt:                 ;IOCTL write
s_open:                 ;Open
s_close:                ;Close
s_media:                ;Removable media
s_ busy:                ;Output until busy
```

```
    MOV AX, UNKNOWN         ;Set error bits
    jmp finish
ident:
    db CR, LF
    db 'Sample Device Driver- - Version '
    db '0.0'
    db CR, LF, LF, '$'
S_init:
    mov ah, 9                       ;Print String
    mov dx, offset ident
    int 21h
    ;Retrieve the rh pointer
    mov dx, CS:rh_seg
    mov es, dX
    mov bx, CS:rh_off
    lea ax, end_driver              ;Get end of driver address
    mov es:[bX] + 14, ax
    mov es:[bX] + 16, CS
    xor ax, ax                      ;Zero the AX register
    jmp finish
s_write:
    xor ax, ax                      ;Zero the AX register
    jmp finish
end_driver:
drvr endp
cseg ends
end
```

编写驱动程序时，可以忽略那些在完成你的工作时不需要的功能。例如，在 drvr.asm 中，只有初始化功能和写功能才起作用。所有剩下的功能都由同一个部分来处理，该部分也只返回一个代码告诉 DOS 所请求的功能是未知的。只需读和写的驱动程序可以只提供这些功能。

唯一必须包含在所有驱动程序内的功能是初始化功能:它必须把驱动程序的最后地址放进请求头的偏移值 0Eh 处，以便操作系统的初始化程序使用它。如果该地址为 0，对于一个块设备驱动程序来说，整个驱动程序头就在安装过程中从内存中移走了。如果在初始化时遇到了致命的错误，那么这种移动是可取的；一个大胆的开发者用这个特点来开发了一个在屏幕偶尔显示一下数据然后立即消失的驱动程序。但是字符设备驱动程序做不到这一点；它的“结束”地址必须大到最少能容纳整个驱动程序头。如果它不这样做，则所有后面的字符设备都会失效。

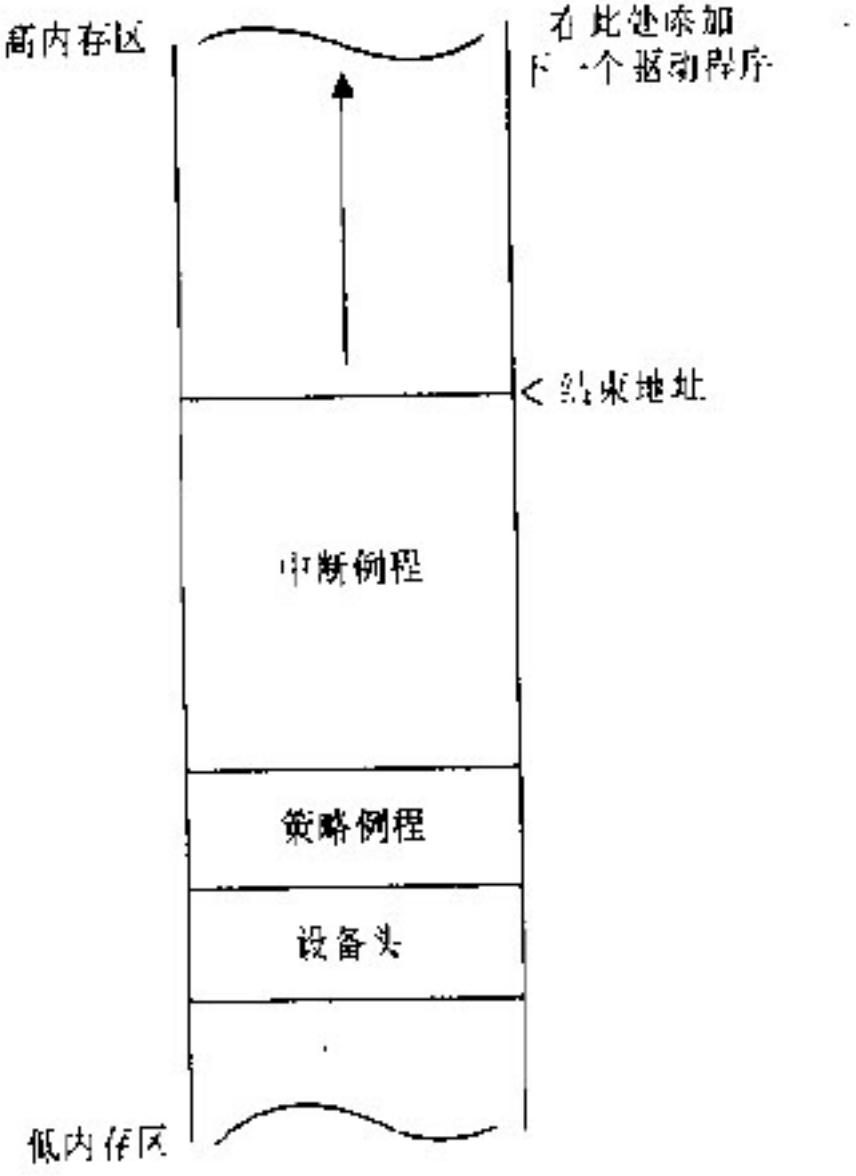


图 12．6 下一个驱动程序添加在当前驱动程序的结束地址之后

要忽略这些功能，只需返回一个代码告知你不知道DOS 在请求什么。然后跳到关闭操作的中断例程的部分，并在请求头的状态字（偏移值 03h）中设置已完成位（第 8 位）。然后是初始化工作。打印一个字符串给屏幕（以便知道驱动程序确实在那里），接着确定驱动程序的最后地址（标号 end_driver 处就是该例程的末尾）。这个地址必须保存在请求头中，这样 DOS 才知道装入下一个驱动程序的位置。由于驱动程序是从低内存到高内存进行装入的，所以下一个驱动程序在当前驱动程序的结束地址之后（见图 12． 6）。

最后写例程把 AX 寄存器设置为 0（驱动程序功能的返回状态）来指示驱动程序的操作中没有出错。这类简单的功能设置对于大多数驱动程序来说是典型的。大多数驱动程序都只需要几个功能来完成其操作。其中一些功能只对某类驱动程序才有意义。例如 BIOS 参数块功能对涉及键盘的驱动程序来说就毫无意义。

表 12．4 列举了设备驱动程序的功能，并指出哪些能用于 DOS 的特定版本。该表还详细说明了每个功能的工作方式。

一、驱动程序初始化

初始化功能是必须存在于所有驱动程序之中的一个功能。它为驱动程序执行所有必要的设置。它有一项任务对 DOS 来说是必不可少的：它必须把驱动程序的结束地址设置到请求头的偏移 0Eh 字节处。如果驱动程序正在驱动一个硬盘系统，初始化功能将检查磁盘的存在及它的正确操作，对磁盘参数进行初始化，等等。对于串行端口，它应该对端口进行初始化并建立缺省值设置。

驱动程序的初始化部分是唯一能合法调用 DOS 的部分。其它部分都不许调用 DOS（记住 DOS 是不能重入的。当程序处在驱动程序里面时，它就是 DOS!）只有功能 01h 到 0Ch（有限的控制台 I／O）和 30h（获得 DOS 版本号）才是可用的。当驱动程序进行初始化时，DOS 的其它部分还未进行初始化，所以对磁盘驱动器之类的调用（等等）会失败并使系统死锁。

要允许参数通过 CONFIG. SYS 命令地来传递给驱动程序，就要传递一个命令行指针给驱动程序。该指针指向“DEVICE＝”行中等号后的第一个字符；只允许初始化程序去读数据，不允许改变它。

表 12。4 设备驱动程序的各个功能

功能	意　义	DOS 版本
00h	驱动程序初始化	2，3，4，5，6
01h	介质检查	2，3，4，5，6
02h	建立 BIOS 参数块	2，3，4，5，6
03h	I/O 控制读	2，3，4，5，6
04h	读	2，3，4，5，6
05h	非破坏性的读	2，3，4，5，6
06h	输入状态	2，3，4，5，6
07h	清空输入缓冲区	2，3，4，5，6
08h	写	2，3，4，5，6
09h	带校验的写	2，3，4，5，6
0Ah	输出状态	2，3，4，5，6
08h	刷新输出缓冲区	2，3，4，5，6
0Ch	I／O 控制的写	2，3，4，5，6
0Dh	打开	3，4，5，6
0Eh	设备关闭	3，4，5，6
0Fh	可移动的介质	3，4，5，6
10h	输出直到忙碌	3，4，5，6
11h	通用 IOCTL	3．2，3．3
13h	通用 IOCTL	4，5，6
17h	获得逻辑设备	3．2，3．3，4，5，6
18h	设置逻辑设备	3．2，3．3，4，5，6
19h	IICTL 查询	5，6

为了保持 DOS 4.0 版（及更高版本）中改进的错误报告，已经在初始化功能的 RH 格式中加上一个标志，用于指示显示 Error in CONFIG. SYS 信息。在装入不成功的情况下要想显示这条信息，可在这个 16 位域中传递任何非零的值；零值则会避免显示，其操作与较早的 DOS 版本中的操作相同。

初始化过程在加上下列信息后，就能更新请求头。

字节偏移值	内容
03h	返回状态
0Dh	单元号（用于块设备）

0Eh　　驱动程序之上第一个自由内存的地址

12h　　BIOS 参数块指针（块设备）

图 12．7 显示了进入初始化功能时的请求头;图 12．8 则显示了从功能出来时的请求头。

因为初始化功能只被调用一次，所以许多程序员都把它放在驱动程序模块的最后，并且将初始化功能的开始处设置第一个自由内存的地址。有较大初始化部分的大驱动程序，能通过这条途径获得大量空间。但本章中的实例没有这么复杂。它只是用程序段已定义的地址来定义驱动程序的。

二、介质检查

DOS 在请求头中为介质检查功能提供了下列信息：

字节偏移值　　内容

01h　　单元码

02h　　命令码（介质检查为 1）

0Dh　　介质描述字节

检查功能将检查块设备上的磁盘介质是否已在最后一次访问后发生了变化。对于字符设备，该例程应该总是返回 DONE(请求状态字设置成 0100h；设置已完成位，即第 8 位；其它都为零）。这就意味着“完成”（见列表 12．1 中的定义）。对于一个固定的块设备（如硬盘），该例程应该总是指示出介质没有改变,它是通过把请求头中的字节偏移值 0Eh 处的内容置为 1 来指示的。但是，怎样才能知道一个可移动的磁盘是否已发生了改变呢？

许多程序员在与这个问题作斗争。但因为在 DOS 世界里从没有什么标准(在 4.0 版之前）来维持卷标等，所以还没有人能提供真正可靠的答案。下面是一些尝试的想法：

请求头偏移值		内容
00h	00	长度
01h	01	单元号
02h	02	命令代码
03h	03	
04h	04	返回状态
05h	05	保留用于 DOS
06h	06	
07h	07	
08h	08	
09h	09	
0Ah	10	
0Bh	11	
0Ch	12	
0Dh	13	
0Eh	14	
0Fh	15	
10h	16	
11h	17	
12h	18	CONFIG.SYS 命令行的偏移值
13h	19	
14h	20	CONFIG.SYS 命令行的段值
15h	21	
16h	22	第一单元号
17h	23	CONFIG.SYS 错误信息控制标志字(仅用于 4.0 版)
18h	24	
19h	25	
1Ah	26	
1Bh	27	
1Ch	28	
1Dh	29	
1Eh	30	
1Fh	31	

图 12．7 进入初始化功能时的请求头

请求头偏移值		内容
00h	00	长度
01h	01	单元号
02h	02	命令代码
03h	03	返回状态
04h	04	
05h	05	保留用于 DOS
06h	06	
07h	07	
08h	08	
09h	09	
0Ah	10	
0Bh	11	
0Ch	12	
0Dh	13	单元个数
0Eh	14	自由内存的偏移值
0Fh	15	
10h	16	自由内存的段值
11h	17	
12h	18	BPB 的偏移值
13h	19	
14h	20	BPB 的段值
15h	21	
16h	22	
17h	23	
18h	24	
19h	25	
1Ah	26	
1Bh	27	
1Ch	28	
1Dh	29	
1Eh	30	
1Fh	31	

图 12．8 从初始化例程返回时的请求头

- 快速回来进行另一次磁盘访问，以致磁盘还来不及移走。但没有人知道怎样的快速才是足够的（或许奥运会将产生一项新的项目——快速打开磁盘驱动器门比赛）；4.0 版的 IBM 手册指定 2 秒为使用的时间。
- 如果能感觉到驱动器门已经打开，并且如果确实已经打开，就可以假定介质已经改变（但是如果偶尔打开了门然后又关上了，那又怎么样呢？）。
- 如果自最后的一次磁盘访问以来，卷 ID 已发生了改变，那么可以知道介质是不相同的。遗憾的是，这句话反过来却是假的（因为有磁盘有相同的卷 ID）。

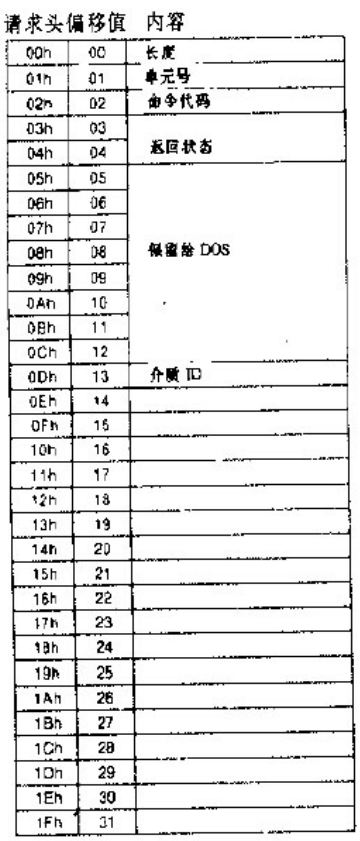
如果正用 DOS 4.0版及更高版本，并且已改变了卷号，那么可以肯定磁盘卷已发生了改变。这个方法最可靠，但它只在 4.0 版或更高版本中才有用，较早的版本没有卷号。

很清楚，在 DOS 所有版本中没有一个统一的很简单的方法来确定磁盘是否已经改变了。在 Macintosh 上，只有通过操作系统才能改变磁盘。在不涉及操作系统的情况下，没有办法（除了用纸张按格来进行系列操纵外）移动磁盘。PC 并不在此限制之中，所以 DOS 程序员丢下了这个包袱。通常，并不能真正地肯定我们向它写的磁盘就是我们打算写的那张盘。

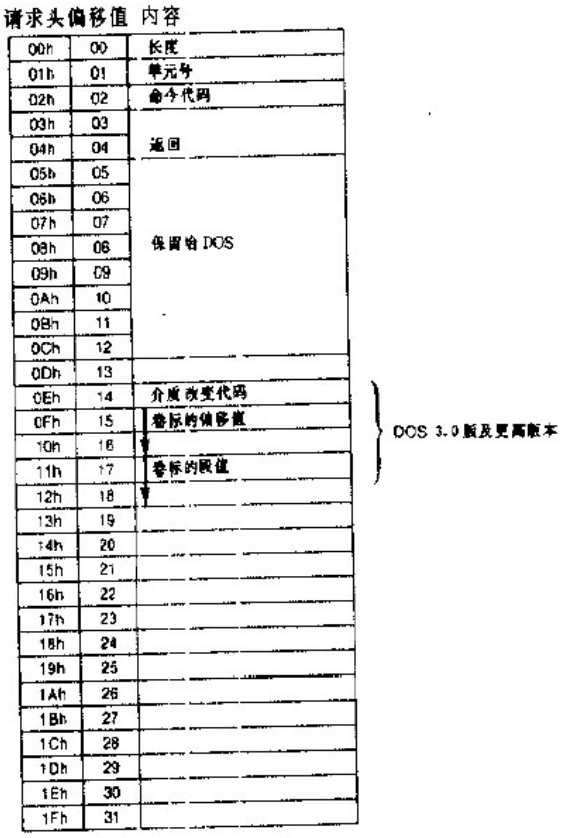
DOS 的反应依赖于放在请求头的字节偏移值 0Eh 上的返回代码。如果值为 1，DOS 就假定设备没变并且在不从磁盘上再次读入 FAT 的情况下接着进行写。—1 的代码告诉 DOS 已改变了磁盘，并迫使 DOS 转储所有写缓冲区，然后从设备重新读 FAT 和目录。DOS 转储缓冲区时，它们中的信息就丢掉了。

0 代码则意味着“也许”（驱动器程序无法告知是否已改变）。在这种情况下，DOS 假定每件事情都正常，并且会直接把所有缓冲区清空到磁盘上。如果这些缓冲区是空的，DOS 就重新读入 FAT 和目录来肯定这些缓冲区确实是空的。通常这个反应是最安全的。如果有什么怀疑，DOS 就会试着保存这些信息（如果改变了磁盘，那么这个过程会导致磁盘扇区被覆盖掉）。

从 DOS 3. 0 开始，该功能还返回一个指向磁盘读的最后卷 ID 的指针。图 12. 9 显示了含有检查功能入口的请求头内容；图 12. 10 则显示了返回的请求头。



请求头偏移值		内容
00h	00	长度
01h	01	单元号
02h	02	命令代码
03h	03	返回状态
04h	04	
05h	05	保留给 DOS
06h	06	
07h	07	
08h	08	
09h	09	
0Ah	10	
0Bh	11	
0Ch	12	
0Dh	13	介质 ID
0Eh	14	
0Fh	15	
10h	16	
11h	17	
12h	18	
13h	19	
14h	20	
15h	21	
16h	22	
17h	23	
18h	24	
19h	25	
1Ah	26	
1Bh	27	
1Ch	28	
1Dh	29	
1Eh	30	
1Fh	31	



请求头偏移值		内容
00h	00	长度
01h	01	单元号
02h	02	命令代码
03h	03	返回
04h	04	
05h	05	保留给 DOS
06h	06	
07h	07	
08h	08	
09h	09	
0Ah	10	
0Bh	11	
0Ch	12	
0Dh	13	
0Eh	14	介质改变代码
0Fh	15	卷标的偏移值
10h	16	
11h	17	卷标的段值
12h	18	
13h	19	
14h	20	
15h	21	
16h	22	
17h	23	
18h	24	
19h	25	
1Ah	26	
1Bh	27	
1Ch	28	
1Dh	29	
1Eh	30	
1Fh	31	

DOS 3.0版及更高版本

图 12．9 进入介质检查功能时的请求头 图 12．10 检查功能返回的请求头

磁盘驱动器行为的安全过程，应该是硬盘返回 NOT CHANGED(请求头中的字节偏移值 0Eh 上的值为 1），而软盘则是 DON't KNOW（那个值为 0）。

三、建立 BIOS 参数块

只有块设备才建立 BIOS 参数块（BPB）功能，并把它们的指针返回到请求头中的偏移值 12h 上。字符设备应该只返回 DONE。在 DOS 3．0 及后来的系统中，该例程应该也读入并保存设备卷 ID 以备后来的检查功能使用，该功能必须在 DOS 3.0 和更高的版本中返回指向卷标 ID 的指针（参看图 12．10）。

在建立 BPB 功能的入口处，请求头包含下列信息：

字节偏移值	意义
01h	单元代码
02h	命令代码（2）
03h	介质描述字节
0Dh	缓冲区地址

传递一个 1 扇区的缓冲区给该例程。若设备属性字中的非 IBM 格式位为 0，缓冲区就包含 FAT 的第一个扇区并且不应改变该缓冲区。如果设置了这个位，该缓冲区可用作一个临时区域，在其中建立 BPB。

无论介质检查功能何时探测到一个磁盘变化或“认为”该磁盘可能已经改变时，DOS 都会从磁盘中调用 BPB 例程来重新建立 BPB。表 12.5 显示了 BPB 的布局。在 3.2 版中扩展了原来的 BPB 结构，以便允许使用需要 32 位存储的扇区号，但在 4.0 版之前，扩展的结构并未应用于通常的使用场所，只是把原来的 BPB 用于少于 64K 扇区的驱动器中，而把扩展结构用于超出此大小的驱动器。

表 12．5BIOS 参数块（BPB）的布局

字节偏移值	域长度	意　　义
00h	字	每扇区的字节数
02h	字节	每簇的扇区数
03h	字	从 0 扇区开始的保留扇区数
05h	字节	FAT 的数
06h	字	根目录项的最大数目
08h	字	扇区总数（在 3.0 及更高的版本中若大于 65535 则为 0）
0Ah	字节	介质描述符
0Bh	字	每个 FAT 的扇区数
0Dh	字	每磁道的扇区数（DOS3.0 和更高版本）
0Fh	字	磁头数（DOS3.0 和更高版本）
11h	双字	隐藏扇区数（DOS3.0 和更高版本）
15h	双字	08h 上的字为零时的总扇区（DOS 3．0 或更高版本）
19h	7 个字节	保留

图 12.11 显示了该功能入口处的请求头；图 12.12 则显示了返回时的情况。

四、I／O 控制读

I／O 控制（IOCTL）读功能使程序能借助 IOCTL 调用直接访问设备。只有在设备头的属性字中设置了 IOCTL 位时，才能调用该功能。其请求头包括下列信息：

字节偏移值	意义
01h	单代码
02h	命令代码
0dh	介质描述字节
0Eh	传输地址
12h	字节／扇区记数

请求头偏移值 内容

00h	00	长度
01h	01	单元号
02h	02	命令代码
03h	03	返回状态
04h	04	
05h	05	保留给 DOS
06h	06	
07h	07	
08h	08	
09h	09	
0Ah	10	
0Bh	11	
0Ch	12	
0Dh	13	介质 ID
0Eh	14	FAT 缓冲区偏移值
0Fh	15	
10h	16	FAT 缓冲区段值
11h	17	
12h	18	
13h	19	
14h	20	
15h	21	
16h	22	
17h	23	
18h	24	
19h	25	
1Ah	26	
1Bh	27	
1Ch	28	
1Dh	29	
1Eh	30	
1Fh	31	

图 12.11 在建立 BPB 功能入口处的请求头

请求头偏移值 内容

00h	00	长度
01h	01	单元号
02h	02	命令代码
03h	03	返回状态
04h	04	
05h	05	保留给 DOS
06h	06	
07h	07	
08h	08	
09h	09	
0Ah	10	
0Bh	11	
0Ch	12	
0Dh	13	
0Eh	14	
0Fh	15	
10h	16	
11h	17	
12h	18	BPB 的偏移值
13h	19	
14h	20	BPB 的段值
15h	21	
16h	22	
17h	23	
18h	24	
19h	25	
1Ah	26	
1Bh	27	
1Ch	28	
1Dh	29	
1Eh	30	
1Fh	31	

图 12.12 从建立 BPB 功能返回的功能请求头

该例程应该返回偏移值 03h 上的状态字以及偏移值 12h 上的传输字节个数。 DOS 在调用中不进行错误检查。

所有 IOCTL 调用（读、写和加进 DOS 3.2版本中的普通调用）与驱动程序通信，而不与设备通信。程序调用这些调用来告诉驱动程序要做的事情及安装它自己的方法。例如，对于一个串行口驱动程序，可以用 IOCTL 写来设置波特率、字长度、停止位和奇偶性；用 IOCTL 读来确定当前设置。问题是 IOCTL 调用与驱动程序的关系是极为密切的（不能通用）。

IOCTL 命令没有 DOS 限定的结构。命令（以应用程序给它的任何形式）保存在传输地址上。如果程序想配置一个串行端口，它可能把下列字符串：

WORD＝8, BAUD＝1200, STOP＝1, PARITY＝N

放在传输地址处，表示 8 个位，1200 的波特率，一个停止位以及没有奇偶性（有关讨论见第 7 章“串行设备”），但不能保证驱动程序会理解这个控制字符串。

如果想在样本驱动程序中试试 IOCTL 调用（参见列表 12.5），驱动程序会忽略任何传递的信息而 IOCTL 调用就会失败。 IOCTL（调用从不会到达驱动程序，因为 IOCTL 支持的位（表 12.1）中设备属性字的第 14 位）还没设置。进一步说，即使能到达驱动程序，也会从 IOCTL 读和写功能返回未知的功能错误。我们没有编写样本驱动程序去对 IOCTL（调用产生反应）。

在大多数书籍中，有关 IOCTL 的情况是笼统的或未定义的。已定义的 IOCTL 的情况见“DOS 参考手册”一节。

五、读

读功能从设备读入数据并把它们返回给指定的缓冲区。该功能也返回一个完成代码和输的字节或扇区数，即使发生了错误，也必须传递这些信息。在 DOS（和更高版本中），如果返回了错误 0Fh，那么驱动程序也必须返回一个指向卷 ID 的指针。

读功能从设备读入并使所读的内容能让调用设备驱动程序的程序得到，该功能以这些方法来与设备通信。在功能入口上，请求头包含有下列信息：

字节偏移值	意义
01h	单元代码
02h	命令代码（3）
0Dh	介质描述字节
0Eh	传输地址
12h	字节 / 扇区记数
14h	起始扇区号（块设备）;若在 V4 中为-1，如果驱动程序处理 32 位扇区数，则为 16h 上的值
16h	32 位起始扇区号（只在 4.0 版本中有）

也可调用读功能来读字符设备和块设备。当推测要进行单个字符读时，就给驱动程序字节计数为 1。忽略那些对于驱动程序毫无意义的参数。有关读操作的原则也适用于写调用。

六、非破坏性读

非破坏性读是只用于字符设备的预先准备方式的读。它对于块驱动程序毫无意义；它们应返回 DONE。在入口上，请求头只在 02h 上包括命令代码（05h） 见图 12.13）。驱动程序应读下个字符，但把它留在输入缓冲区中，以便调用读功能时，为该功能所用。字符应返回到请求头的位置 0Dh 处，见图 12.14

在键盘操作过程中，DOS 用非破坏性读功能来进行预先准备的读。DOS 还用此功能在键盘输入流中寻找 Ctrl-C 字符。

七、输入状态

DOS 用输入状态功能来检查字符是否正在字符设备的输入缓冲区中等待着。块设备自动为该例程返回 DONE。字符设备把它们的状态返回到请求头的位置 03h 上。该功能不像非破坏性读功能，它不读任何字符；它只是返回设备的忙碌状态。

在该功能可使用时，DOS 就用它来检查设备在试图读之前是否忙碌。在入口上，请求头只包含位置 02h 上的命令代码（06h）。

请求头偏移值		内容
00h	00	长度
01h	01	单元号
02h	02	命令代码
03h	03	
04h	04	返回状态
05h	05	
06h	06	
07h	07	
08h	08	保留给 DOS
09h	09	
0Ah	10	
0Bh	11	
0Ch	12	
0Dh	13	
0Eh	14	
0Fh	15	
10h	16	
11h	17	
12h	18	
13h	19	
14h	20	
15h	21	
16h	22	
17h	23	
18h	24	
19h	25	
1Ah	26	
1Bh	27	
1Ch	28	
1Dh	29	
1Eh	30	
1Fh	31	

图 12.13 非破坏性读功能入口处的请求头

请求头偏移值		内容
00h	00	长度
01h	01	单元号
02h	02	命令代码
03h	03	
04h	04	返回状态
05h	05	
06h	06	
07h	07	
08h	08	保留给 DOS
09h	09	
0Ah	10	
0Bh	11	
0Ch	12	
0Dh	13	字符
0Eh	14	
0Fh	15	
10h	16	
11h	17	
12h	18	
13h	19	
14h	20	
15h	21	
16h	22	
17h	23	
18h	24	
19h	25	
1Ah	26	
1Bh	27	
1Ch	28	
1Dh	29	
1Eh	30	
1Fh	31	

图 12.14 从非破坏性功能返回的请求头

八、清空输入缓冲区

该命令代码（07h,在该功能入口处请求头的位置 02h 上）告诉驱动程序去转储任何等待从设备输入的字符。它应该把返回状态码返回到请求头中的位置 03h 处。块驱动程序总是返回 DONE。

九、写

写功能（命令代码 08h 在请求头的位置 02h 处）把字符从与请求头一起传递的缓冲区中拿出来并把它们输出给设备。如果已返回 0Fh 错误代码，它就返回状态（在位置 03h 上）、所传输的字节或扇区数（在位置 02h 上）及（在 DOS 3.0 版和更高版本中）一个指向卷 ID（在位置 16h 上）的指针。必须返回错误状态和传输的字节数。

其请求头的布局与读功能及带校验的写功能是一样的。

十、带校验的写

该功能命令代码 09h，在请求头的位置 02h 上，其格式与写功能一样，它还应该验证已完成的写操作。验证过程包括假装的检查体（即，没有验证，就像一般的 CON 驱动程序所做的那样），它以此来执行数据字节与字节之间的比较，该验证是写给那些在写后又读回来的数据的。标准的块设备只执行数据的一次循环冗余检查（CRC）。

任何不能执行验证的设备都应该以非验证和书写行为来对该请求产生反应 。如果 DOS 验证标志为 ON，那么所有写请求都会自动转换成带有验证的写；所以支持该功能是很重要的。

十一、输出状态

输出状态功能（命令代码 0Ah，在请求头的位置 02h 上）返回字符设备的状态。要确定设备是否忙碌，DOS 可在输出给设备之前调用该功能。将要打印的数据传递给打印机驱动程序时，该驱动程序会看到这个调用。返回状态字（请求头中的字节 03-04h 用于返回 设备的状态。如果较低的 8 个位为零，那么设备就准备好了。如果未准备好，其状态就用表 12．3 中的标准代码进行编码。

十二、清空输出缓冲区

该功能（命令代码 0Bh 在段头的位置 02h 上）转储输出缓冲区的内容。像清空输入功能一样，它只能用于字符设备。块设备应返回 DONE。

十三、I／O 控制写

像 I／O 控制读功能一样，I／O 控制（IOCTL）写功能（命令代码 0Ch 在请示头的位置 02h 上)也直接访问设备。只有在设备头的属性字中设置了 IOCTL 位，才能调用该功能。IOCTL 读功能的所有说明（除命令代码以外）都可用在这里，但却是反过来——例如 IOCTL 写功能把信息传递给驱动程序。因为与读功能一样，驱动程序和应用程序必须在发送内容及其格式上达成一致。

十四、打开

如果在设备属性字中设置了 OPEN/CLOSE/RM 位，那么在设备上要进行打开时可调用打开功能（命令代码 0Dh，在请求头的位置 02h 上）。对于块设备，可用这个调用来跟踪设备上的已打开文件数。遗憾的是，FCB 功能调用避开了这个计数器，因为用 FCB 打开的文件不能关闭。用句柄，DOS 就能在过程结束时自动关闭文件。但是用 FCB 功能，就没有可调用的 FCB CLOSE 功能，除非由进程来关闭它。在字符设备上，事情要简单一些，因为这个调用常用于传递特殊的启动字符串（如打印机初始化字符串）给设备，或者对多个进程的同时访问。

在该功能的入口处，请求头包括单元码（在位置 01h 处）和命令码（0Dh，在位置 02h 处）。从此功能返回时，将要传递的状态字放在 03h 处。

十五、设备关闭

设备关闭功能（0Eh，在位置 02h 处）有助于跟踪查看设备在当前是否向一个或多个进程开放。如果打开功能（0Dh）在它被调用时增加了内部计数器，关闭功能就能减少计数并在计数到达 0 时清除缓冲区。但 FCB 打开功能所带来的问题依然存在。终止字符串（如最后的换行符）可送给字符设备。该功能的项参数就像打开功能的那些参数：在 01h 处的单元代码以及 02h 处的命令代码（0Eh）。

十六、可移动的介质

如果在设备头中设置了 OPEN/CLOSE/RM 位，那么可在 DOS3.0 及更高版本中使用可移动的介质调用（0Fh，在位置 02h 处）来确定设备是否有可移动的介质。如果没有，DOS 能优化其策略来处理该设备，把磁盘参数表装入到内存中以获得更快的访问速度。字符设备应该只返回 DONE。从表 12．3 中选择状态代码来返回到请求头 03h 上的状态字上。

如果可能，该调用就返回一个 0 到 8 状态字的 BUSY 位上；1 则表示不能移动的介质。

十七、输出直到忙

该功能（10h，在位置 02h 上）原来是为打印卷纸（折叠纸）而提供的，一些设备，有大量的内部缓冲区或独立的打印机缓冲区的打印机能以极高的速率来接收字符 ——其速度比计算机送出字符的速度还要快。如果允许这样，驱动程序就能持续不断地把大量字节传输到设备中，而不使设备变得忙碌。这就是这个功能的目的。

调用该垢霉时，它以尽可能快的速度把字节传递给设备。它尽可能多地传递字节，直到设备变得忙碌起来或者直到要传输的字节已经传输完毕 。在功能的入口处，请求头在 02h 上放有命令代码，在 0Eh 上有传输地址（这里有要写给设备的字节）,而字节计数则放在 12h 处。

从功能返回时，请求头必须在 03h 上有返回状态，在 12h 上有要传输的字节数。如果已传输的字节数少于要传输的字节数，那就没有出错。块设备应为该功能返回 DONE。

十八、通用 IOCTL

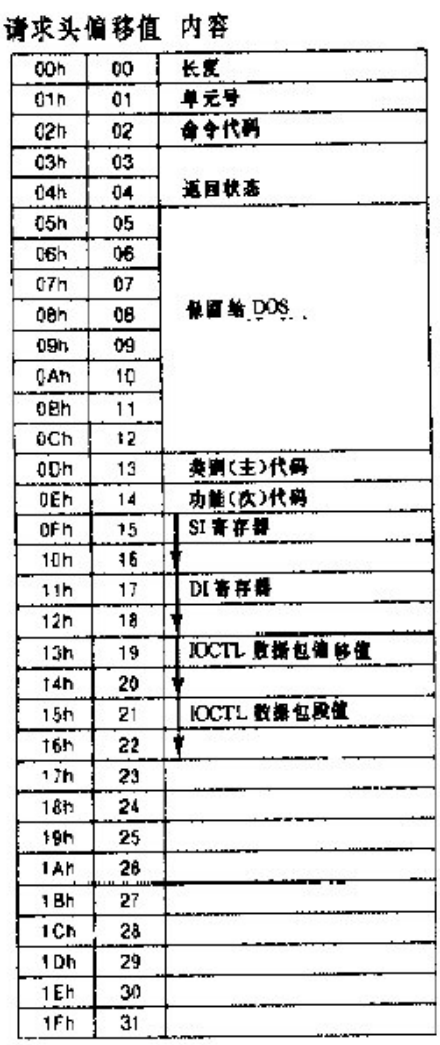
像 IOCTL 读和 IOCTL 写功能一样，通用 IOCTL 功能（3.0 版中命令代码为 11h，在请求头的位置 02h 上，4.0 和更高版本中，命令代码为 13h，在相同的位置上）依赖于驱动程序和应用程序之间的一致性信号的使用。因为没有什么规则，所以该功能常为编写他们自己的驱动程序的程序员工作得最好。通用 IOCTL 支持 DOS 3．3 带来的一些新的 IOCTL 功能（见“DOS 参考手册”一节）。

图 12.15 显示了该功能入口处请求头的布局；图 12.16 则显示了返回时的布局。

十九、获得并设置逻辑设备

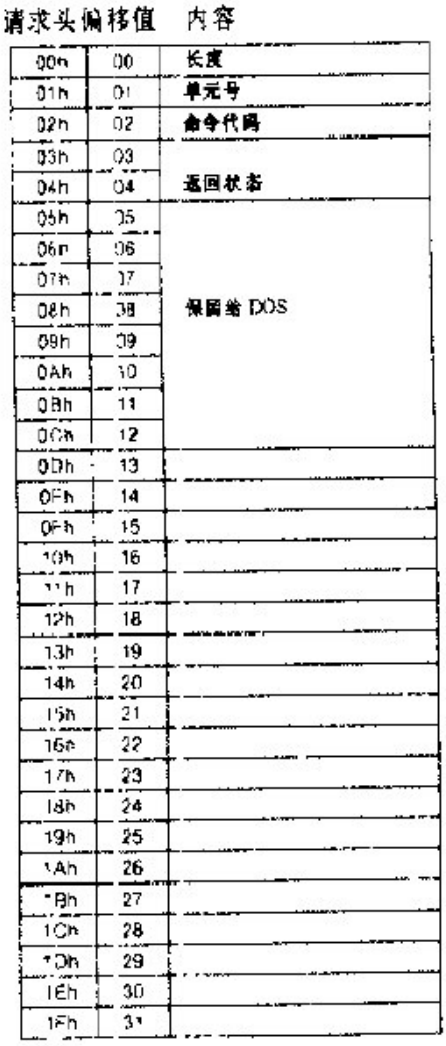
这些功能（命令代码 12h 和 13h（3.0 版中）或 17h 和 18h（4.0 版或更高版本中）,在请求头的位置 02h 处）支持 Int 21h，功能 44h（子功能 0Eh 和 0Fh）的操作。它们确定所给设备的最后一个块设备名并告诉驱动程序下一个设备名。有关这些功能的其它情况，可参(见“DOS 参考手册”一章）。

可用请求头的位置 01h 处的单元号及位置 02h 上的命令码来调用这些功能。反过来，最后的设备单元码会返回到位置 01h 处，而设备状态则返回到 03h 处。



请求头偏移值		内容
00h	00	长度
01h	01	单元号
02h	02	命令代码
03h	03	返回状态
04h	04	
05h	05	保留给 DOS
06h	06	
07h	07	
08h	08	
09h	09	
0Ah	10	
0Bh	11	
0Ch	12	
0Dh	13	类别(主)代码
0Eh	14	功能(次)代码
0Fh	15	SI 寄存器
10h	16	
11h	17	DI 寄存器
12h	18	
13h	19	IOCTL 数据包偏移值
14h	20	
15h	21	IOCTL 数据包段值
16h	22	
17h	23	
18h	24	
19h	25	
1Ah	26	
1Bh	27	
1Ch	28	
1Dh	29	
1Eh	30	
1Fh	31	

图 12.15 通用 IOCTL 入口处的请求头



请求头偏移值		内容
00h	00	长度
01h	01	单元号
02h	02	命令代码
03h	03	返回状态
04h	04	
05h	05	保留给 DOS
06h	06	
07h	07	
08h	08	
09h	09	
0Ah	10	
0Bh	11	
0Ch	12	
0Dh	13	
0Eh	14	
0Fh	15	
10h	16	
11h	17	
12h	18	
13h	19	
14h	20	
15h	21	
16h	22	
17h	23	
18h	24	
19h	25	
1Ah	26	
1Bh	27	
1Ch	28	
1Dh	29	
1Eh	30	
1Fh	31	

图 12.16 从通用 IOCTL 返回的请求头

12．4 完整的驱动程序

列表 12．6 列举了完整的驱动程序 drvr.asm。根据前面一节中的指导来改写并汇编这个程序,就能按需要产生一个能工作的驱动程序(尽管它不能做很多工作,但这已是个起点）。但要小心:虽然已测试过这个驱动程序，但是如果打错一些内容或不清楚的地方隐藏了重要步骤，或者步骤没有正确地完成（并错过了它）,那么就会使程序变成一团糟。无论何时测试驱动程序，一定要在软盘而不是主系统上操作。一定要遵守下一节中列举的警告。

列表 12．6

```
; drvr.asm
CR          EQU 0Dh                 ;Carriage return
LF          EQU 0Ah                 ;Line feed
MAXCMD      EQU 16                  ;DOS 3.0, 12 DOS 2.0
ERROR       EQU 8000h               ;Set error bit
BUSY        EQU 0200h               ;Set busy bit
DONE        EQU 0100h               ;Set completion bit
UNKNOWN     EQU 8003h               ;Set unknown status
cseg        segment public 'code'   ;Start the code segment
            org 0                   ;Zero origin
```

```
            assume cs:cseg, ds:cseg, es:cseg
; ========================================================
drvr        proc far                ;FAR procedure
    dd  -1                          ;Next driver pointer
    dw 8000h                        ;Attribute
    dw strategy                     ;Pointer to strategy
    dw interrupt                    ;Pointer to interrupt
    db 'DRVR '                      ;Device name
    ; ========================================================
    rh_seg dw ?                     ;RH segment address
    rh_off dw ?                     ;RH offset address
strategy:
    mov cs:rh_seg, es
    mov cs:rh_off, bx
    ret
; ========================================================
; dispatch table
; ============================================================
d_tbl:
    dw s_init                       ;Initialization
    dw s_mchk                       ;Media Check
    dw s_bpb                        ;BIOS parameter block
    dw s_ird                        ;IOCTL read
    dw s_read                       ;Read
    dw s_nrd                        ;Nondestructive read
    dw s_inst                       ;Current input status
    dw s_infl                       ;Flush input buffer
    dw s_write                      ;Write
    dw s_vwrite                     ;Write with verify
    dw s_ostat                      ;Current output status
    dw s_oflush                     ;Flush output buffers
    dw s_iwrt                       ;IOCTL write
    dw s_open                       ;open
    dw s_close                      ;Close
    dw s_media                      ;Removable media
    dw s_busy                       ;output until busy
; ============================================================
; interrupt routine
; ============================================================
interrupt:
    cld                             ;Save machine state
    push es                         ;Save all registers
    push ds
    push ax
```

```
    push bx
    push cx
    push dx
    push si
    push di
    push bp
    mov dx, cs:rh_seg
    mov es, dx
    mov  bx, cs:rh_off
    mov al, es:[bx]+2               ;Command code
    xor ah, ah
    cmp ax, MAXCMD                  ;Legal command?
    jle ok                          ;Jump if OK
    mov ax,UNKNOWN                  ;Unknown command
    jmp finish
ok:
    shl aX, 1                       ;Multiply by 2
    mov bx, ax
    jmp word ptr[bx + d_tbl]
finish:
    mov dx, cs:rh_seg
    mov es, dx
    mov bx, cs:rh_off
    or ax, DONE                     ;Set the DONE bit
    mov es:[bx]+3, ax
    pop bp                          ;Restore the registers
    pop di
    pop Si
    pop dx
    pop cx
    pop bx
    pop ax
    pop ds
    pop es
    ret                             ;Back to DOS
;=================================================================
; main body of driver
;=================================================================
s_mchk:                         ;Media check
s_bpb:                          ;BIOS parameter block
S_ird:                          ;IOCTL read
S_read:                         ;Read
s_nrd:                          ;Nondestructive read
s_inst:                         ;Current input status
```

```
s_infl:                          ;Flush input buffers
S_vwrite:                        ;Current output status
S_ostat:                         ;Current Output Status
s_oflush:                        ;Flush output buffers
S_iwrt:                          ;IOCTL write
s_open:                          ;open
s_close:                         ;Close
s_media:                         ;Removable media
s_busy:                          ;output until busy
    MOV AX, UNKNOWN              ;Set error bits
    jmp finish
ident:
    db CR,LF
    db 'Sample Device Driver..Version'
    db '0,0'
    db CR,LF,LF,'$'
s_init:
    mov ah,9                     ;Print String
    mov dx, offset ident
    int 21h
    ;Retrieve the rh pointer
    mov dx,CS:rh_seg
    mov es，dx
    mov bx, cs:rh_off
    lea ax, end_driver          ;Get end of driver address
    mov es：[bx]+14, ax
    mov es：[bx]+16, cx
    xor ax, ax                   ;Zero the AX register
    jmp finish
s_write：
    xor ax, ax                   ;Zero the AX register
    jmp finish
end_driver：
drvr endp
cseg ends
end
;================================================
```

12.4.1 汇编驱动程序

要汇编驱动程序，需要运行宏汇编程序和连接程序MASM 6. 0 中的ml，然后用EXE2BIN程序产生驱动程序为二进制的映象。这些步骤已联合进一个标准批处理文件，该文件还把驱动程序拷贝到A 驱动器上（防止使用者忘记这样操作）。列表 12.7 是 MAKEDRVE.BAT 文件，用于执行上述的各项操作。

列表 12.7

```
;=======================================
```

```
@echo off
ml /Fo%1%1.asm
exe2bin %1 %1.sys
copy %1.sys a:
```

该批处理文件能汇编驱动程序。如果汇编正确，驱动程序就能连接起来。连接成功时，就执行 EXE2BIN 来把驱动程序转换成内存映象格式。程序结束时，驱动程序就被拷贝到 A 驱动器上。

大多数程序员在驱动程序上工作时总是反复汇编。批处理文件有助于消除不断失败而带来的紧张，看不到任何内容以及没有明白的途径来获得输出就会导致不断的失败。如果编写一个驱动程序，就要预备它在用户获得正确操作之前可能会几次死锁系统。

要使用批处理文件，必须给它驱动程序的源文件名（没有扩展名）。想执行 drvr.asm 程序时，可敲入下面一行：

C：＞makedrvr drvr

批处理文件会自动明白输出的是一个可加进 CONFIG.SYS 文件里的文件（名为 DRVR.SYS）。

12.4.2 安装驱动程序

操作系统在系统引导过程中处理CONFIG.SYS 文件时会安装此驱动程序。若 DRVR.SYS(我们的驱动程序）保存在 A 驱动器的根目录中，可以编辑 CONFIG.SYS 文件，使之包含下列行：

DEVICE＝A:DRVR.SYS

从而把 DRVR.SYS 文件加到系统中。

12.4.3 调试驱动程序。

如果驱动程序是第一次就能运行，那么情况要比预料的好一些。就是最好的程序员也要不断测试驱动程序直到它们工作正常。有时并不是程序员做错了什么；可能是程序员没有完全理解设备。

要调试驱动程序，必须完成大量广泛的头擦除工作。初始化过程中地址里的一个很小的错误也会死锁 DOS 系统，如果该 DOS 系统正在寻找的指针存在于内存的其它地方的话。对驱动程序的调用会在黑洞中消失，再也不会返回。应用程序可以给功能调用附加上错误的反应，因为驱动程序返回了错误号。

从驱动程序内部进行打印并不容易，所以获取出错信息是件讨厌的事情。有时当错误是时间为关键性时，只用把它放进调试码中就能使驱动程序运行正常。为 UNIX 系统工作的驱动程序不会在 DOS 下工作，除非在驱动程序中间用去了一些数量的不确定的时间。尽管许多系统程序员已用此功能进行了大量操作，还是存在一些延时。DOS 能以相同的途径来操纵用户。

要调试驱动程序，请记住下列指导原则：

- 不要在硬盘上测试新驱动程序 。构造一张可引导的软盘并把驱动程序和 CON-FIG.SYS 文件拷贝到软盘上以备测试。若在硬盘上测试并且驱动程序未进行初始化，就不能直接引导硬盘。这时就不得不用软盘引导来改变硬盘上的 CONFIG.SYS。
- 如果有系统没有硬盘，那么就在那里进行测试。就是一个简单的问题也能在驱动程序水平上破坏结果。 如果驱动程序破坏了硬盘 FAT，那又怎么样呢？）
- 用 BIOS 调用在关键点上打印驱动程序的状态。如果想了解输出，小心不要包含过多的调试输出以致不能阅读完。
- 测试过程中任何记录屏幕显示及能在低速度中回行显示的内容都能有所帮助。小计算机、系统打印机或甚至录象带磁带，如果它们能配置来记录所发生的事情都会有

所帮助。

12.5 编一个实用的驱动程序

向样本驱动程序加上一些东西,可使它更实用。加上写功能后能使驱动程序获得字符时,便能向打印机写。如果想让驱动程序代替缺省的打印机驱动程序,就可以在设备头中改变它的名字（改成 PRN）。

drvr. asm 中的一切应保持不变，但扩大了写功能（见列表 12. 8)

列表 12. 8

```
        mov cx, es:[bx]+12h         ;Number of bytes to print
        mov di, es:[bx]+0eh         ;Offset of data buffer
        mov ax, es:[bx]+10h         ;Segment address of data buffer
        mov es, ax
        mov dl, 0                   ;Printer 0
        mov bX, 0                   ;Count 0 bytes printed
  s_prt1:
        cmp bx, cx                  ;printed all characters yet?
        je s_done                   ;All done
        mov al, es:[di]             ;Get a character
        inc di                      ;Point to the next one
        mov ah, 2                   ;Check printer status
        int 17h
        test ah, 80h                ;Busy?
        jne s_prtch                 ;Print it
        jmp s_err                   ;Busy device, exit
    s_prtch
        test al, Lf                 ;Is the character a line feed?
        je s_bxinc                  ;Skip it
        mov ah, 0                   ;Print character
        int 17h
        test ah, 09h                ;I/O error?
        jne s_err                   ;Handle it
    S_bxinc:
        inc bx                      ;Count one printed
        jmp s_prt1
    S_err:
        mov ax, 800ch               ;General failure error
        jmp s_end
    s_done:
        mov ax, bx                  ;Save count
        mov bx, cs:rh_off           ;Get req.hdr
        mov es:[bx]+12h, ax         ;Store byte count
        xor ax, ax                  ;Zero AX register
    S_end:
        jmp finish
```

新代码向打印机打印字符并忽略换行字符。如果想用那些回车和换行作为一个“回车和换行"对，并向老式打印机打印时，这个功能就很有用。

该功能很简单。它一开始就把请求头指针定位到数据缓冲区（偏移值 0Eh 和 01h）中，并把字节数，定位到传输区（偏移值 12h）中。然后它检查打印机状态，若为忙碌状态就返回一个错误。如果不忙，就打印字符。在这个简单例程中，任何阻止用户向设备写字符的错误都会使该功能在 AX 寄存器（设备返回状态）中设置出错码。

可以扩大这个功能，使之进行更复杂的错误处理——如认识到哪些错误并返回合适的代码——但基本功能还是原样。

12.6 使用设备驱动程序

要测试驱动程序，就必须使用它。从程序或直接从命令行符处，都有许多简单的方式来使用它。

该样本驱动程序名为 DRVR（见列表 12．6）。用名字调用它时，它就被激活。要把一些内容重新定向给有正常驱动程序的打印机（PRN），可打入下面一行：

C：＼＞type autoexec.bat＞prn：

这一行指导把 AUTOEXEC.BAT 文件拷贝到打印机上。要使用驱动程序，可敲入下面一行：

C：\:type autoexec.bat＞drvr；

用户想向 drvr 写时，DOS 就检查驱动程序链来定位 DRVR 这个名字，并使用该功能驱动程序来打印数据。

像打开文件那样，打开设备，就可以从程序中访问驱动程序。可以用句柄功能调用来打开设备（见列表 12．9）

列表 12．9

```
/* example.C
    Listing 12. 9 of DOS Programmer's Reference*/
union REGS regs;
regs.h.ah=0x3d;                  /*Open-file function*/
regs.h.al＝0x01;                 /*Write access*/
regs.x.dx=(int)”drvr”;           /*Creates pointer to string*/
intdos(&regs, &regs);            /*Call DOS function int*/
handle=regs.x.ax;                /*Save the file handle*/
```

不管选择哪种方式来访问驱动程序，都能根据其设计来测试其操作。在 drvr 中，能测试它是否能向打印机打印字符以及是否能从字符流中消除换行。

12.7 小　　结

本章讨论了程序员们所认为的 DOS 编程中最困难的部分：创建驱动程序。遵循下列驱动程序的标准就能创建设备驱动程序。

所有驱动程序都由 3 个基本部分组成：

- 设备头
- 策略例程
- 中断例程

每部分都有它自己的结构。设备头包括驱动程序名和在系统驱动程序链中下一个驱动程序的指针。策略例程只记住系统请求头内存的地址以备驱动程序与操作系统之间通信时使用。

大多数驱动程序都包含在构成中断例程的各个功能（有 21 个）之中。所给的驱动程序都只能执行几个功能而忽略其它功能，如果向驱动程序提出这样的请求，就会返回一个合适的完成码。