内 容 提 要

本书共有 18 章和两个附录，系统而详尽地讲述了有关 PC 机的 I/O、CPU 和固定内存区等硬件的知识和许多未公开发表的鲜为人知的内部技术和相关资料，并提供了大量的源代码。该书原著出版后大受欢迎，被广大读者认定为“是有关 PC 机硬件的详尽的终极资源”、“PC 硬件发烧友的利器”、“硬件核心技术的宝库”，还被 PC 杂志评论为“是系统级程序员的圣经”。

本书对于从事 PC 机硬件和软件设计的技术人员有极重要的参考价值，也可供关心 PC 技术的爱好者们参考。

图书在版编目（CIP）数据

PC 技术内幕/（美）奇鲁威著；精英科技译.-北京：中国电力出版社，2000

ISBN 7-5083-0503-5

Ⅰ.P… Ⅱ.①奇…②精… Ⅲ.个人计算机-基本知识 Ⅳ.TP368.3

中国版本图书馆 CIP 数据核字（2000）第 86338 号

著作权合同登记号 图字：01-2000-1695

本书英文版原名：The Undocumented PC

中国电力出版社出版、发行

（北京三里河路 6 号 100044 http://www.infopower.com.cn）

实验小学印刷厂印刷

各地新华书店经售

*

2001 年 4 月第一版 2001 年 4 月北京第一次印刷

787 毫米×1092 毫米 16 开本 65 印张 1486 千字

定价 99.00 元

前　言

第二版《PC 技术内幕》在对第一版作了较大修改的基础上，提供了广泛而全新的资料。我仍然对过去两年来的变化感到惊讶:MCA 系统被无情的淘汰了,与 PC 发展的最初 10 年相比，出现了更多的新型处理器和更多的新型操作系统。

本书有什么新内容?

本书第二版对每一章都作了修改。某些章只是简单地作了些更新，例如加入了一些最新的 Windows95 键盘键。另外一些章节,比如硬盘系统这一章,看起来就像每页都作了修改。我还加入了大量的资料来处理某些新型的 CPU,这些 CPU 在第一版中尚未涉及。其中包括 Intel 的 Pentium Pro 和 MMX 系列，AMD 的 5x86、Cyrix5x86 和 6x86，以及那些过时了的 NexGen CPU。

我还对许多在第一版中出现的程序加以改进和更新。跟第一版一样，我们公开了全部的源代码（感兴趣的读者可以访问 www.infopower.com.cn——编者注）。这些新特征包括：检测 PCI 及其相关信息、BIOS 供应商和日期、以及一个详细显示和描述 BIOS 数据的工具。键盘视图程序也作了修改，以适应更多的工作环境和用任意键来显示未译扫描码和译毕扫描码。

CPU 检测程序将检测是否带有 MMX，并标识真正的指令集(这些指令集通常和你所想象的并不一样)。我还进一步详细地查阅了许多与 CPU 供应商相关的资料，以获取 CPU 速度及其内部信息。型号专用寄存器程序按名词顺序描述了许多尚未公布的寄存器，并且实现了对隐藏寄存器的快速访问。

为什么将这本书取名为《PC 技术内幕》?

如果你是第一次阅读此书，可能会被书名《PC 技术内幕》所吸引。但是我希望你和我一样对此持有怀疑的态度，因为过去有许多技术书籍给人以很大的希望，但实际上仅仅只是一个老调重弹的作品，比如一张和以前相同的老式表，谈论一些关于中断、OS 命令，ASCII 码表等的信息。但是你不会对这本书感到失望的。

你会发现,这本书与现在的其他书籍很少有相同之处。在一些相关的问题上,我加入许多信息和代码例,而不是仅仅作一些比较深入的阐释。

那么,这本书到底有什么地方吸引人呢？我并非无所不知，在本书所涉及的内容之外还有许多 PC 尚未说明的领域。我仅仅只是关注那些非常有用的领域，这些领域以前尚未说明或者阐明得不够，但是对目前许多开发项目非常有用。

在每一个新系统的设计当中，生产商的目的似乎是要将它做成一个最不可靠的平台，这个平台可能存在于一个复杂软件的区域内。我为何说出如此难以置信的话？因为这些制造商及其协会所提供的说明文档是令人难以置信的。对于早期的 PC 机，至少你可以获得完整的原理图和 BIOS 列表，当说明文档缺乏关键的信息时，你却可能会用到这些信息。但是即使如此，如果你对读懂原理图和关键的汇编代码感到困难，你可能很难用好这些信息。

今天，说明文档提供的信息少得可怜。显然，准备这些文档时他们很少考虑到软件和固件开发人员的实际需要。这些粗劣的文档的价格也变得越来越昂贵，简直令人望而却步。例如，如果你对支持基于 EISA 和 MCA 总线的机器感兴趣，那么你就不得不花几百美元来购买有关 EISA 和 MAC 的说明书。这也是为什么这些开发平台很少成功的一个原因。即使你对整个技术工作比较熟悉，你还是会发现许多地方不够精确、信息缺损，或者很多地方标有“保留”而隐藏了相关信息。

我努力保证本书高度精确。为了做到这一点，我开发了一些工作程序来检验这本书中的有关信息。另外，我还花了大量的时间来确认最低层的信息，包括仔细检查原理图、隐藏的 BIOS 列表和 IC 源清单。

或许你刚刚才开始理解本书所讨论的问题，但是你仍然马上可以使用这些信息。除此之外，我还汇集了一些信息，涉及那些过时的和现行的系统标准，包括 PC、XT、AT、ISA、MCA、EISA 和 PCI 系统。其中许多过时的系统，如 MCA 机，现在仍然在使用，并且不费多大劲就可以轻易地获得支持。

通过操作系统或者一些详细说明的标准功能，通常可以高效地处理软件任务，适宜的时候使用接口不失为明智之举。当这些接口没有提供任务所需要的属性和工作时，有必要深入底层，充分发挥那些制造商特意隐藏起来的系统潜在功能。

现在，我给那些制造商们一些喘息的机会，他们创造了一台多功能的机器，出于竞争的考虑，他们隐藏了一些信息是可以理解的。同时从一个实用的软件视角来解释操作也不是一件容易的事情。如果某些功能设计没有加以说明，那么更新它们就显得容易些，因为这样的话，理论上没有人会使用他们。今天的市场要求开放的界面，大多数供应商声称他们的系统是透明的，但是不要期望能够从他们那里获得非常有用而廉价的信息。

本书的结论会令你感到满意的，对于这一点我充满了信心。我知道每天有许多新的设计推向市场，这些设计都有相应的说明文档。但是我相信，这本书的确为程序员们提供了详尽的信息，而这些信息被长久的忽略了。

Frank van Gilluwe (74000.635@Compusurve.com)

目　录

前　言

第 1 章　简介 …… 1
资料的来源 …… 2
系统类型 …… 3
程序员的系统框图 …… 3

第 2 章　开发 PC 内幕 …… 6
简介 …… 6
反汇编 …… 8
反汇编 BIOS …… 21
IOSPY-I/O 端口监视器 TSR …… 23
UNPC-I/O 端口浏览器 …… 26

第 3 章　CPU 和内幕指令 …… 28
基本输入输出块 …… 28
从端口输入 …… 30
警告 …… 32
指令定时 …… 34
定时方面的难题 …… 35
与 I/O 有关的 CPU 模式 …… 39
通过 C 和 C++访问硬件 …… 40
CPU 系列归纳 …… 46
内幕指令 …… 56
使用 LOADALL …… 82
寄存器细节 …… 93
隐藏的地址空间 …… 109
内电路模拟 …… 118
CPU 重启 …… 118

第 4 章　系统与设备检测 …… 123
简单的方法 …… 123
系统检测 …… 125
CPU 信息 …… 139

第 5 章　适配卡的开发 …… 212
ROM 表头和初始化 …… 212
MCA ROM 扫描 …… 213
设置 ROM 大小和开始地址 …… 213
ROM 代码 …… 214
获得必要 RAM 的决窍 …… 215
选择 I/O 端口号 …… 219
很多端口 …… 219
隐去 ROM 和 RAM …… 221
开关与跳线 …… 222
即插即用 …… 222

第 6 章　BIOS 数据和其他固定数据区 …… 223
BIOS 数据区 …… 223
扩展 BIOS 数据区 …… 256
显示器内存 …… 263
适配器 ROM 和 UMB 内存 …… 264

第 7 章　中断向量表 …… 265
中断向量表与数据描述 …… 269

第 8 章　键盘系统 …… 292
基本操作 …… 293
AT 上的一个典型的按键操作 …… 294
PC/XT 上一个典型的按键操作 …… 295
控制器通信 …… 295
键盘到主板的数据 …… 295
AT 上主板到键盘的数据 …… 297
低级键盘 BIOS …… 298
键盘 BIOS ——中级 …… 299
键盘 BIOS 数据区 …… 310
热键及访问未定义键 …… 312
扫描码 …… 313
国外的键盘 …… 319
扩展内存的 A20 访问 …… 322
警告 …… 326
键盘的连接和信号 …… 328
端口归纳 …… 349
端口细节 …… 349

第 9 章　视频系统 …… 370
简介 …… 370
视频适配器标准 …… 371
BIOS 服务 …… 374
其他与视频系统相关的中断 …… 442
重定位屏幕接口规范（RSIS） …… 443
环境是如何提供 RSIS 支持的 …… 452
端口归纳 …… 459
未公开的视频 I/O 端口的细节 …… 461
第 10 章　软盘系统 …… 464
简介 …… 464
软盘驱动器媒质表 …… 466
软盘数据格式化 …… 466
软盘参数表 …… 467
BIOS 初始化 …… 470
软盘 BIOS …… 471
软盘 BIOS 数据 …… 479
软盘控制器的常见类型 …… 482
向软盘控制器发送命令 …… 483
端口归纳 …… 488
端口细节 …… 488
第 11 章　硬盘系统 …… 516
简　介 …… 516
是否会给出实际的硬盘大小 …… 518
接口标准和控制器 …… 519
驱动器操作 …… 521
大型的 IDE 驱动器 …… 522
磁盘参数表 …… 523
驱动器类型表 …… 525
BIOS 初始化 …… 528
硬盘 BIOS …… 529
磁盘 BIOS 数据 …… 568
向磁盘控制器发送命令 …… 569
一个典型的读扇区操作 …… 569
警告 …… 574
端口归纳 …… 583

端口细节 …… 585
命令细节 …… 595

第 12 章　串行口 …… 638
简介 …… 638
BIOS 初始化 …… 640
串行口 BIOS …… 641
串行帧 …… 646
控制/调制解调器信号 …… 646
事件顺序——串行传送 …… 647
事件顺序——串行接收 …… 648
回环（Loopback）操作 …… 648
波特率 …… 649
中断控制 …… 650
FIFO 模式 …… 651
BIOS 数据区 …… 653
调试 …… 654
UART 类型归纳 …… 655
串行连接器 …… 655
警告 …… 656
端口归纳 …… 681
UART 寄存器细节 …… 683

第 13 章　系统功能 …… 694
BIOS 服务 …… 694
端口归纳 …… 748
端口细节 …… 749

第 14 章　并行口和屏幕打印 …… 789
简介 …… 789
BIOS 初始化 …… 790
一个系统可以有第四个并行口吗？ …… 791
打印机 BIOS …… 791
屏幕打印 …… 793
BIOS 数据区 …… 794
并行口定时 …… 796
并行口连接器 …… 797
快速并行口 …… 798
警告 …… 799

端口总结 …… 808
端口细节 …… 809

第 15 章　CMOS 内存和实时时钟 …… 813
简介 …… 813
实时时钟（RTC）的常用信息 …… 813
实时时钟 BIOS …… 814
EISA 系统的不同之处 …… 819
系统数据区 …… 827
扩展 CMOS 寄存器 …… 827
警告 …… 827
端口归纳 …… 836
端口详述 …… 837

第 16 章　系统时钟 …… 896
简介 …… 896
操作模式 …… 897
时钟 0——系统定时 …… 903
时钟 1——DRAM 刷新 …… 903
时钟 2——一般用途和扬声器 …… 903
时钟 3——看门狗（仅 MCA） …… 903
时钟 3——看门狗（仅 EISA） …… 904
时钟 4——未使用（仅 EISA） …… 905
时钟 5——CPU 速度控制（仅 EISA） …… 905
典型的时钟设置和操作 …… 905
典型用途 …… 906
访问 …… 906
警告 …… 906
端口归纳 …… 918
端口细节 …… 919

第 17 章　中断控制和 NMI …… 938
简介 …… 938
典型的中断过程 …… 939
边沿/电平控制 …… 940
NMI——不可屏蔽中断 …… 941
浮点协处理器和 NMI …… 941
MCA 系统的不同之处 …… 942
EISA 系统的不同之处 …… 943

PCI 系统的不同之处 …… 943
典型使用 …… 943
中断数据区 …… 945
警告 …… 945
端口归纳 …… 950
端口细节 …… 951

第 18 章　DMA 服务和 DRAM 刷新 …… 970
简介 …… 970
澄清模糊认识 …… 972
一个典型的 I/O 到内存的传送 …… 973
内存到内存 DMA …… 974
操作模式 …… 975
MCA 系统的不同之处 …… 976
EISA 系统的不同之处 …… 977
虚拟 DMA 服务（VDS） …… 981
典型用途 …… 982
DMA BIOS 数据区 …… 982
警告 …… 983
端口归纳 …… 986
端口细节 …… 988

附录 A　软件包中的程序 …… 1019
可执行程序归纳 …… 1019
有趣的子程序 …… 1020
可执行程序的详细解释 …… 1021

附录 B　术语表 …… 1025
常用的缩写形式 …… 1025
常见芯片的编号和功能 …… 1028

第1章

简　介

内幕信息可能会非常有趣并激起读者的兴趣，但是我只会关注那些软件和硬件开发人员初次接触的 PC 领域。可能你想知道，有关 PC 的技术书籍何止成百上千，还会留下什么内幕！不错，在过去的几十年中，似乎有数不尽的重要领域都涉及到了 PC，但是它们往往未加说明或者阐明得很不够。许多重要的区域，例如系统 BIOS 和输入/输出端口，就很少被详细阐述，所以人们常常难以真正理解和使用系统这些重要的领域。

特别值得指出的是，输入/输出端口往往是系统环境中阐述得最不清楚的地方。我尽力阐明每一个端口及其定义位，这一点与那些提及 I/O 的用户手册的单线描述有很大不同。在许多情况下，我还提供了关于其用法的例子和可能出现的问题。

在为新的硬件选择端口位置时，这本参考书对开发人员的作用是无法估量的，有一整章专门从软件的视角讨论了一些与适配卡有关的问题。

你可能会问，这本《PC 技术内幕》完整吗？我曾经编写了一个流行的反汇编程序，Sourcer,在它的帮助下，过去的十年中我一直在收集有用的资料，以便充实我所写的这本内幕书籍。作为一个反汇编器，Sourcer 将一些可执行文件和 BIOS ROM 转化成可读的汇编代码，并对中断、I/O 端口等做出注释。为了保证 Sourcer 能够跟上时代，我仔细检查了说明文档和列表，并深入理解了 BIOS 功能和 I/O 端口的使用方法及原理。

综上所述，本书对于系统的每个功能块都有一章专门论述。每章都从最底层解释了每个 BIOS 功能，并对相关 I/O 端作出了详细的阐述。许多章节还提供了一些有趣的程序，来说明如何访问这些功能。这些程序包含有完整的源代码，因此将它们改成你自己的程序语言并不难。

最有趣的论述还包括如何开发第 2 章所论述的技术内幕。第 3 章讲述了大量尚未说明的指令。第 5、6、7 章阐释了适配卡的开发、BIOS 数据区以及中断向量表。剩下的章节则对每一个子系统作出详细说明。

资料的来源

在我开始写这本书时，我知道现在有大量的信息要么论述得不够充分，要么就是缺损的，或者根本就从未阐释过。为了获得本书的相关资料，我做了大量非同寻常、细致深入的工作。大多数情况下，我首先回顾了一下制造商为子系统提供的 IC 数据源清单，然后仔细察看这些芯片在标准的主板上是如何具体连接的。做到这一点需要用到系统的原理图，在某些情况下，还要查看系统的电路图。我也仔细研究了不同制造商提供的反汇编 BIOS 代码，以便在这些较低的层次上考察它们与子系统的联系。我还生成了一些测试程序来检验某些子系统的操作。最后，我才去看那些“正式”的文档，包括 IBM 的技术参考资料，当然它也是许多其他技术书籍的资料来源。

IBM 的技术参考资料对可靠的底层信息来说，是一个差劲的资料来源，对于这一点我并不感到惊讶。显然，IBM 的说明文档直接源于各种 IC 数据清单中的程序注解。令我感到吃惊的是，在实际应用中这些信息常常要么是错误的，要么就容易引起误解。例如，在硬件具体执行时，某些芯片的性能根本不可能得到发挥，然而某些说明文档却对此加以详细地阐述，似乎有人真的实际应用过它们。在许多情况下，对功能的解释过于简单，对程序员毫无用处。

当然，我并不是说，IBM 的技术参考资料一点用处也没有。其实，他们也提供了许多有用的信息。当然，这带来了你一定会遇到的第二个问题。这些说明文档对开发各种系统很有必要，例如开发 PC、XT、AT、ISA、MCA、EISA，许多说明文档都谈到了 PCI 系统。为了易于掌握，我列出了每个 BIOS 功能和端口描述的差别。这些正是程序员所需要的。

许多技术参考资料喜欢用“保留”一词，来避免谈及一些具体细节，然而，“保留”的真正含义并不太清楚。我认为，供应商指的是下面所列的某个意思：

- 现在未用到，但是将来可能用到
- 现在未使用，也可能从未使用过
- 在一些功能中用到，但 PC 结构不支持这个功能
- 在一些隐藏的功能中用到
- 在某些功能中用到，但这些功能会产生意想不到的效果
- 在老式的 PC 中用到，但现在过时了
- 供应商也不知道它的用处。

本书中我尽量不使用“保留”一词。但是坦白的说，我也不能讲清楚某些保留功能有什么用处。大多数情况下，你可以认为，在我的分析中“保留”指的就是“未使用”。

系统类型

在本书的许多地方，我特别列出一些机器类型的缩写形式，如 AT，EISA 等等。它们代表了一个特定的系统家族系列，这些系列的硬件设计相同、BIOS 功能类似。一个加号（+）表明该家族系列以及其后的家族系列都支持该特定的功能和 I/O 端口。例如：AT+表示所有的 AT 机器都支持该功能，同时，EISA、PCE 和过时了的 MCA 机器也都支持。

类 型	描 述
PC	早期的基于 8088 的计算机。
XT	早期的基于 8088 的计算机，带有硬盘，但没有内置的 CMOS 时钟和 CMOS 配置内存。
AT	基于 80286、386、486 以及一些 Pentium+ CPU 的系统，工业标准结构（ISA）总线。
MCA	基于 80286、386、486 以及一些 Pentium+ CPU 的系统，带有微通道结构（MCA）总线。所有的 IBM PS/2 50 型以及更高级的机器都使用 MCA 总线。
EISA	基于 386、486 和 Pentium+ CPU 的系统，带有扩展工业标准结构（EISA）总线。
PCI	基于 486、Pentium 以及 PentiumPro CPU 的系统，带有外围部件互联（PCI）总线。

程序员的系统框图

尽管你可能对基本系统有深入的理解，但是我认为从编程的角度来考虑系统而不去考虑实际的处理器和总线设计仍是非常有用的。虽然许多程序设计依赖于 CPU 和总线结构，但是仍然有许多编程系统并不依赖于这些独特之处。许多硬件性能，例如总线宽度、高速缓存内存、局部总线，与编程一点关系也没有，也没有编程接口。

图 1-1 被我称为程序员的系统框图，它列出了系统的所有硬件块，并指出了彼此之间的连接关系，同时还标出了这个硬件块在系统内的中断和 I/O 端口。每个方框中还指出了对这个子系统作出深入阐述的章节。

记住，不同的供应商和平台提供的硬件的连接方法有所不同，但是功能性的软件总是相同的。任意编程上的差别都在子系统所用到的中断和端口中加以了注明。如果一个供应商偏离了基本标准，那么众多的软件就不能在该机器上正常运行。早期，由于供应商偏离了基本标准，从而阻碍了兼容 PC 的发展。那些遵守基本标准，并且真正兼容的机器的供应商今天依然存在，而其他的已成为历史。

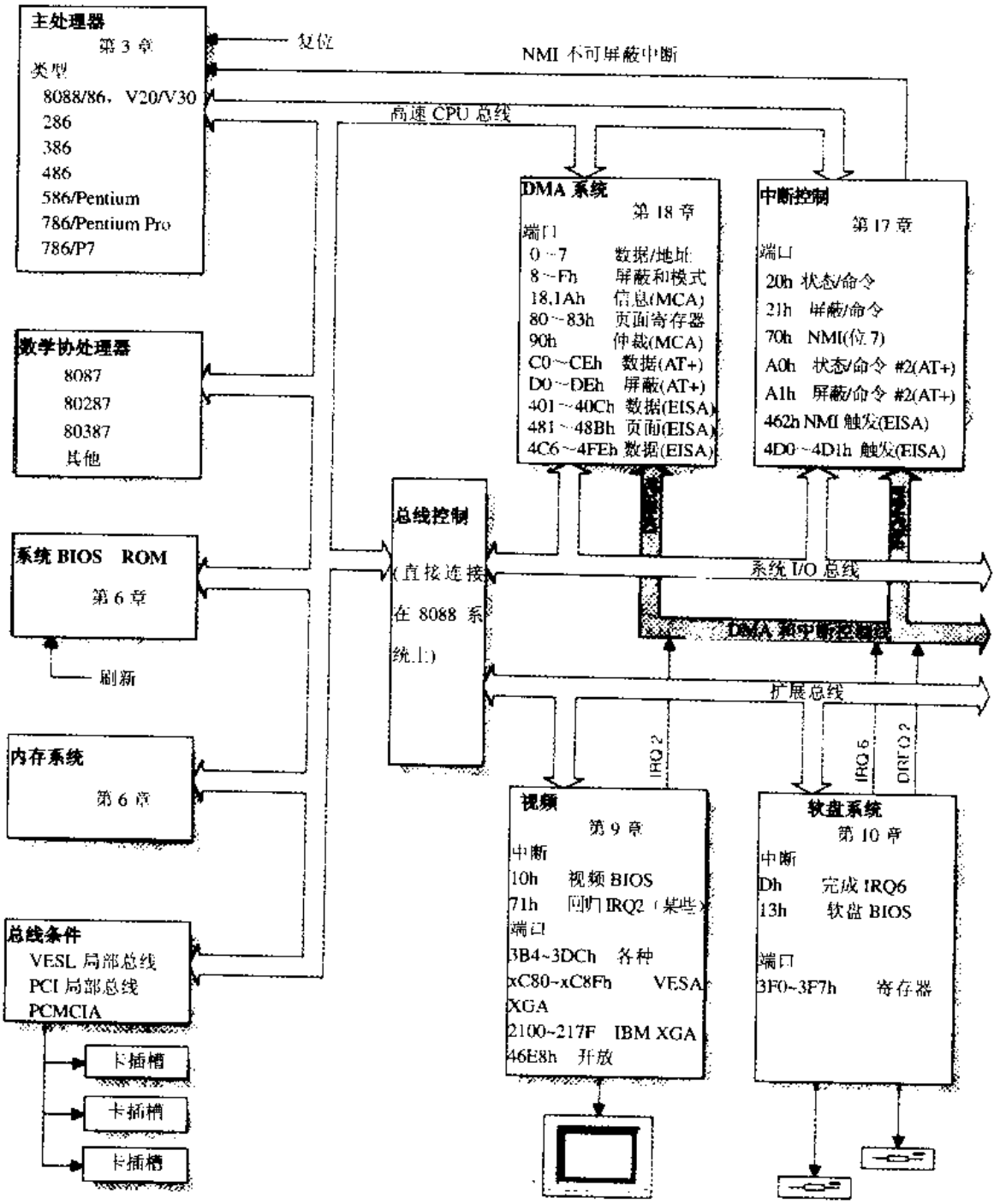
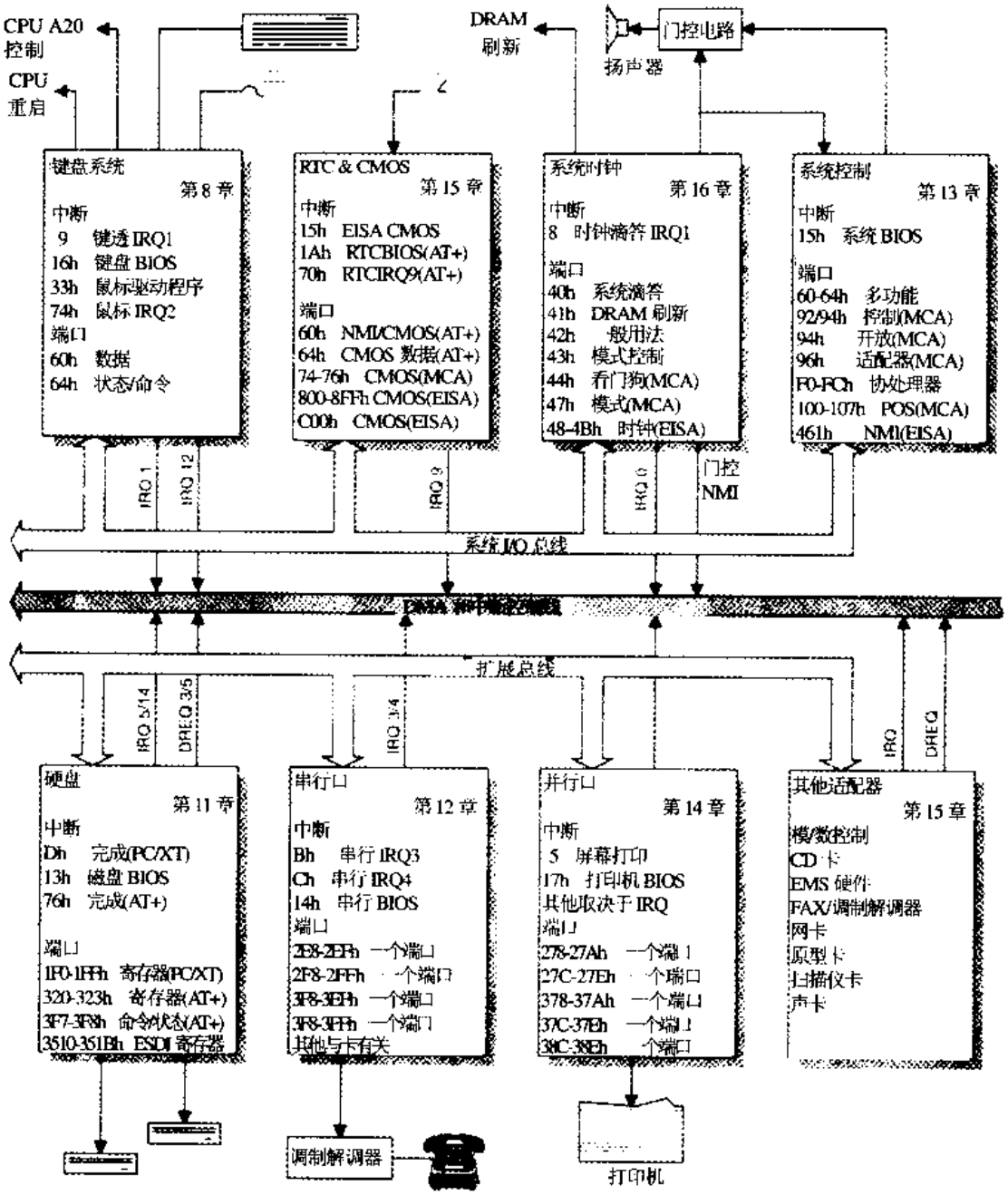


图 1-1 程序员的系统框图


CPU A20
控制
CPU
重启
DRAM
刷新
门控电路
扬声器
键盘系统
第8章
中断
9 键透 IRQ1
16h 键盘 BIOS
33h 鼠标驱动程序
74h 鼠标 IRQ2
端口
60h 数据
64h 状态/命令
RTC & CMOS
第15章
中断
15h EISA CMOS
1Ah RTCBIOS(AT+)
70h RTCIRQ9(AT+)
端口
60h NMI/CMOS(AT+)
64h CMOS 数据(AT+)
74-76h CMOS(MCA)
800-8FFh CMOS(EISA)
C00h CMOS(EISA)
系统时钟
第16章
中断
8 时钟滴答 IRQ1
端口
40h 系统滴答
41h DRAM 刷新
42h 一般用法
43h 模式控制
44h 看门狗(MCA)
47h 模式(MCA)
48-4Bh 时钟(EISA)
系统控制
第13章
中断
15h 系统 BIOS
端口
60-64h 多功能
92/94h 控制(MCA)
94h 开放(MCA)
96h 适配器(MCA)
F0-FCh 协处理器
100-107h POS(MCA)
461h NMI(EISA)
IRQ 1
IRQ 12
IRQ 9
IRQ 0
门控
NMI
系统 I/O 总线
DMA 和中断控制
扩展总线
IRQ 5/14
DREQ 3/5
IRQ 3/4
IRQ
DREQ
硬盘
第11章
中断
Dh 完成(PC/XT)
13h 磁盘 BIOS
76h 完成(AT+)
端口
1F0-1FFh 寄存器(PC/XT)
320-323h 寄存器(AT+)
3F7-3F8h 命令/状态(AT+)
3510-351Bh ESDI 寄存器
串行口
第12章
中断
Bh 串行 IRQ3
Ch 串行 IRQ4
14h 串行 BIOS
端口
2E8-2EFh 一个端口
2F8-2FFh 一个端口
3E8-3EFh 一个端口
3F8-3FFh 一个端口
其他与卡有关
并行口
第14章
中断
5 屏幕打印
17h 打印机 BIOS
其他取决于 IRQ
端口
278-27Ah 一个端口
27C-27Eh 一个端口
378-37Ah 一个端口
37C-37Eh 一个端口
38C-38Eh 一个端口
其他适配器
第15章
模/数控制
CD 卡
EMS 硬件
FAX/调制解调器
网卡
原型卡
扫描仪卡
声卡
调制解调器
打印机

第2章

开发PC内幕

简　介

生产商总是对许多新的 PC 领域未加说明，当然，这并一定是故意的，但是相对于迅速发布的系统适配器或软件而言，有关的文档总是相对落后。特别是当要从底层实现设备功能而需要这些文档时更是如此。即使提供了相关文档，对专业化软件来说也常常是毫无用处的。

通过我在这里阐述的一些例子，你可以拨开制造商提供文档的面纱，加深对关键接口、操作和数据区的理解。我也会告诉你如何避免落入充满了误解的陷井和使用一些并不精确的文档信息。

要诀#1：找到最初的源文档

针对你所感兴趣的子系统请选择那些最初的源文档作参考，其中包括原理图、IC 使用说明清单、IC 应用注释以及 IC 程序员参考手册。如果手头有这些文档，并且你对硬件和软件有所了解，那么你就能透彻地理解大多数子系统的本质。记住，每台 PC 中都使用了许多复杂的 IC。在许多情况下，它们拥有一些特性和功能，然而这些特性和功能在 PC 设计中却是不可能实现的。例如，8254 可编程时钟 IC 虽有许多模块供每个计数器编程使用，但却因为时钟 IC 与 PC 硬件连接方式，许多模式不可能实现或者没有任何意义。彻底理解 IC，同时仔细查看具体的原理图，将有助于消除这些限制。当然我已经为 PC 子系统完成了这项工作，然而你仍然可以仔细研究一些新型的设计和适配卡。

通常，许多技术参考资料只是从一本 IC 用户手册中取出一些资料而并不考虑其实际应用效果。经常会出现这种情况：许多技术手册中列出的功能和模式在 IC 手册中也有，然而对任意一个 PC 的设计并不适用。

不幸的是，在将这些技术应用到子系统时存在着许多问题。很少有制造商提供其内部的原理图。IBM 最近一次在《技术参考资料—AT 个人计算机》中公布原理图是在 1985 年。即使在这本参考资料中，IBM 也删去了大量的信息，并且使用了许多难解的术语，使这本书越发难以理解。许多程序员发现一个普通的原理框图都很难理解，更不用说那些重要信

息的缺失了。

对于那些最新的 PCI、MCJ 和 EISA 系统，在最低层次上理解子系统几乎是不可能的。有时原理图并未提供，并且大多数最新的系统使用传统的 LSIIC，因此也很少提供数据，甚至根本就没有。这时，必须采用另外的方法来理解所关注系统的性能。

要诀#2：有没有其他的人已经做过这方面的工作？

我希望你能在这本书中找到你所需要的所有的答案。但是，正如早就指出的那样，总是不断地有新的适配卡、子系统甚至是新系统设计被创造出来。有许多子系统已经被仔细地研究过，并且也已作出了非常好的论述说明。VGA 显示器系统就属于这一类。目前有许多出色的技术参考手册从不同的角度谈到 VGA 适配卡的编程问题。

上面所述看起来像是说，但是请你看看那些适配器或子系统的制造商是否为程序员准备了所需要的技术参考资料吧！小公司比较乐意发布它们产品的信息，以便使他们的产品成为一种标准，而大公司通常提供一些信息，但是往往很难找到合适的文档，即使找到了，它们通常也比较贵。

如果你正在寻找那些直接从硬件设计中归纳出来的文档，那么请记住，通常它们也提供了一些额外的性能。这些额外的性能之所以透露出来，往往是由于某些别有用心的供应商需要保护其竞争力，或者是由于这种性能用处有限，或者是技术资料的纂写者没有从软件角度充分理解该硬件。

要诀#3：看看其他人是怎么做这项工作的？

即使你有幸获得供应商所提供的源文档，它们也很少指出如何编写程序来应用所关注的子系统。仔细研究 BIOS 是如何同硬件子系统交互的，这将极大地促进你对该子系统的了解。BIOS（基本输入/输出系统）控制着程序访问系统内硬件的方式。

一种信息来源是 BIOS 对 IBM AT 给出的源列表。最近一次正式的 BIOS 列表出版在 IBM 的《技术参考资料——个人机 A》手册中。在理解 AT BIOS 源程序的内部工作原理方面，它是一个相当出色的信息来源。尽管如此，它仍然也已经过时了。这些列表由许多小程序生成，很难理解其中一系列的跳转和列表调用。它也没有提供前后对照，并且，由于磁盘上没有列表，搜索数据和子程序是徒劳无益的。

其他的信息来源包括微软公司提供的 Windows 和 NT 设备驱动工具条（DDK，Device Driver Kit）。这些程序包含有大量的汇编源码，指出了 Windows 和 NT 是如何实现与 PC 硬件的连接的。在某些情况下，这些程序绕过了 BIOS 执行程序。DDK 包括了一些程序来替代一部分 BIOS，这部分 BIOS 用于键盘、软盘和一些硬盘 BIOS 服务。DDK 还包括了一些代码，以实现和串行口、鼠标以及许多视频标准的接口。

最后一种方法是反汇编 BIOS 或者驱动程序来看看其他人是如何使用硬件的。我开发

了一个商业化的产品，带有 BIOS 预处理器的程序 Sourcer，来完成这项工作。正是由于我开发了这个工具，所以我能发现许多尚未说明的 PC 硬件和软件功能特性。

通过反汇编，有可能确切知道制造商是如何编写软件的。反汇编也指示了如何使用那些尚未说明的中断、I/O 端口、指令、以及其他的技巧，供应商通常在其软件中用到这些技巧。在下面一部分中我们将详细地研究反汇编的全过程。

反汇编

对反汇编的需求

在软件开发过程中，如果需要源代码或代码文档，但是手头又没有，这是一件很令人沮丧的事情。对可执行代码进行反汇编是唯一可行的解决方法。由于很多原因，出现缺乏源代码的机会远远比人们所能想像的要多得多。

许多开发人员发现，获得源代码列表也有助于理解其他的产品。如果必须开发一个新的程序来实现或加入到一个现存的产品中去，那么透彻理解接口，对于成功实现这一点就非常重要了。一旦弄清楚了接口，你的应用开发就可以继续进行下去。

反汇编的条件选项

每台销售的 DOS 电脑包含有一个基本的调试工具，称作 DEBUG，它对于解决问题很有帮助。DEBUG 可反汇编可执行程序并显示汇编代码。不幸的是，DEBUG 在此方面的能力非常有限。

更先进的产品，例如 Borland 的 Turbo Debugger，也能反汇编，但是它的注意力主要集中在开发过程中程序的调试，在这种情况下，已经存在有源代码。由于它们一次只显示一个屏幕的代码，因此这些调试器以及其他交互式反汇编工具的结果极其有限。输出不容易写到一个文件或打印机上，代码也没有汇编向导，并且要理解它们也很不容易。这些产品在将代码分开方面无能为力，并且很容易出现排列错误，需要做大量工作将它们拉回到正常的轨道上。

为什么反汇编这么难？

8088 微处理器系列的指令集最复杂，所以很难被正确地反汇编。这些 CPU 带有一个庞大而复杂的指令集，并且还有许多小小的变化。一个最大的困难出现在内存定位方面。某列表的某一行如图 2-1 所示。

```
0492    A1   20C2          mov          ax，Memory_spot
```

图 2-1 源代码列表中的内存定位

由于编译器知道 Memory_spot 所在的段址和偏移量，所以这个简单的移动指令很容易写和汇编。当处理器执行这条指令时，通过这个二进制指令中的值 20C2h 和 DS 段寄存器中的当前值，处理器可以计算出 Memory-spot 的地址。仅仅以二进制指令进行反汇编是不够的，因为不知道段寄存器的值。如果仅仅给出二进制指令“A1 20C2”，一个反汇编器就会有 65,536 种可能的地址供数据驻留内存。当没有提供其他的源代码时，图 2-2 展示了调试器是如何显示同一指令的。除非程序恰恰执行到这一位置，否则无法知道当前的段值。

```
22B1：0492    A1   20C2          MOV      AX,[20C2]
```

图 2-2 调试器反汇编

处理偏移量甚至是一个更难的问题。在图 2-3 列出的第一行，在 SI 寄存器中装入了 string_data 二进制形式的偏移量。String_data 的段址并不重要，除非需要再次使用 SI。在这个例子中，第三条指令改变了 ES 的段值，第四行指令用 ES 段寄存器和 SI 寄存器作为内存地址参考。

```
011C   BE   4E02          MOV     si,offset String_data
011F   B8   6000          MOV     ax,data_seg
0122   BE   C0            MOV     es,ax
0124   26:8B   14         MOV     dx,es:[si]
```

图 2-3 来自源列表中的偏移操作

调试器一般不作任意分析，仅仅只是显示指令的编码值。图 2-4 给出了 DEBUG 对同样代码的输出。

DEBUG 输出				注释
5C22:011C	BE024E	MOV	SI, 4E02	无偏移量
5C22:011F	B8C260	MOV	AX,60C2	无段址
5C22:0122	BEC0	MOV	ES,AX	
5C22:0124	26	ES:		段覆盖在不正确的线上
5C22:0125	8B14	MOV	DX,[SI]	

图 2-4 来自 DEBUG 的代码

尽管从技术上讲正确，DEBUG 没有什么问题，但是 DEBUG 反汇编还是使程序非常难

以理解。DEBUG 也使程序不容易修改，因为所有的偏移量和段都是 16 进制值的。

带注释反汇编

大约 10 年前，我曾开发一个专用的反汇编程序，来解决 DEBUG 和其他反汇编器遇到的问题。我希望看到反汇编输出能够带有注释。我也希望这个输出同那些出色的程序员所编写的汇编程序类似,以避免留下 DEBUG“机器”翻译的痕迹。这几年来，我所编写的反汇编程序 Sourcer，被 V 通信公司购买并加以了改进。从一些可执行文件中,Sourcer 可以生成汇编源代码,并对复杂的指令、I/O 端口用法、中断子功能作出注释。Sourcer 可以对程序进行全面分析，并使用多种途径来生成完整的汇编源代码。

一次性扫描整个程序，并且执行一次代码的内部模拟，将极大地提高输出的效果。这样，理解代码就会更容易。Sourcer 还插入了必要的汇编指示来帮助输出的源代码在重新汇编时尽可能简便。

参看下面的一个 932 字节的小文件，KEYID.EXE。这个程序可以让用户从三个选项中选择一个，并且还可以显示 ASCII 码、所读入的代码或者数字锁定状态。尽管这个程序的用处不大，但是 KEYID 仍是一个很好的起步程序，可以用来说明反汇编的复杂性。

首先，我使用 DEBUG 进行代码反汇编，得到的结果如图 2-5 所示。很难说清楚这个程序在做些什么以及它是如何实现的这些功能的。这是互动式的调试和反汇编的一个典型的例子。如果将程序扩大十倍，那么反汇编任务将会不可控制。我已经对一些很重要的问题在旁边作了注释。

DEBUG				DEBUG 注释
C:> DEBUG KEYID.EXE				运行 DEBUG;
-U 0 L1A8				从 0 开始反汇编者按 1A8h 个字节，该区域实际上是数据，但是 DEBUG 不能区别代码和数据，或者指出数据的类型;
22E8:0000	4B	DEC	BX	
22E8:0001	45	INC	BP	
22E8:0002	59	POP	CX	
22E8:0003	49	DEC	CX	
22E8:0004	44	INC	SP	
22E8:0005	207631	AND	[BP+31],DH	
22E8:0008	2E	CS:		
22E8:0009	3030	XOR	[BX+SI],DH	
22E8:000B	2028	AND	[BX+SI],CH	
22E8:000D	63	DB	63	DEBUG 进一步出错
22E8:000E	2920	SUB	[BX+SI],SP	
22E8:0010	3139	XOR	[BX+DI],DI	
22E8:0012	3933	CMP	[BP+DI],SI	

```
22E8:0014  204656          AND   [BP+56],AL
22E8:0017  47              INC   DI
22E8:0018  B8F422          MOV   AX,22F4          这是代码的超始位置，但是并不明显
22E8:0018  8ED8            MOV   DS,AX
22E8:001B  B409            MOV   AH,09
22E8:001D  BA0E00          MOV   DX,000E
22E8:0022  CD21            INT   21               这条语句干什么？混淆了越界点
22E8:0024  26              ES:
22E8:0025  803E800002      CMP   BYTE PTR [0080],02
22E8:002A  7509            JNZ   0035             后没有标号
22E8:002C  26              ES:
22E8:002D  A18100          MOV   AX,[0081]        数据在哪个段中？
22E8:0030  86C4            XCHG  AL,AH
22E8:0032  EB0C            JMP   0040             没有标号
22E8:0034  90              NOP                    可执行吗？
22E8:0035  B409            MOV   AH,09
22E8:0037  BA2000          MOV   DX,0020          偏移量的值？
22E8:003A  CD21            INT   21               这条语句做何用处？
22E8:003C  FECC            DEC   AH
22E8:003E  CD21            INT   21
22E8:0040  2C31            SUB   AL,31
22E8:0042  77EF            JA    0035
22E8:0046  32E4            XOR   AH,AH
22E8:0048  D0E0            SHL   AL,1
22E8:004A  8BDB            MOV   BX,AX
22E8:004C  FFA7CE00        JMP   [BX+00CE]        没有提示这在做什么
22E8:0050  B409            MOV   AH,09            跳到什么地方？怎么执行这条语句？
     ... skipping over confusing code
22E8:00A6  B44C            MOV   AH,4C
22E8:00A8  CD21            INT   21
22E8:00AA  0000            ADD   [BX+SI],AL       似乎非常不可能（实际上这是堆栈，在 Soucer
22E8:00AE  0000            ADD   [BX+SI],AL       的输出中显示为一条语句“db 30 dup(0)）
22E8:00B0  0000            ADD   [BX+SI],AL
22E8:00B2  0000            ADD   [BX+SI],AL
22E8:00B4  0000            ADD   [BX+SI],AL
22E8:00B6  0000            ADD   [BX+SI],AL
     ... skipping over confusing code
22E8:00CA  0000            ADD   [BX+SI],AL
```

```
22E8:00CC 0000          ADD     [BX+SI],AL
22E8:00CE 4B            DEC     BX              没有任意提示表明这是一个新段，并且再
22E8:00CF 45            INC     BP              次将数据错当成了代码
22E8:00D0 59            POP     CX
22E8:00D1 20494E        AND     [BX+DI+4E],CL
22E8:00D4 44            INC     SP
22E8:00D5 45            INC     BP
... skipping over confusing code
22E8:019A 36            SS:                     实际上它是数据串，而不是代码。
22E8:019B 37            AAA
22E8:019C 3839          CMP     [BX+DI],BH
22E8:019E 41            INC     CX
22E8:019F 42            INC     DX
22E8:01A0 43            INC     BX
22E8:01A1 44            INC     SP
22E8:01A2 45            INC     BP
22E8:01A3 46            INC     SI
22E8:01A4 86C4          XCHG    AL,AH           这已经越出文件末尾以外了。
22E8:01A6 90            NOP
22E8:01A7 B44F          MOV     AH,4F
-Q                                              足够了，退出吧。
```

图 2-5 使用 DEBUG 反汇编 KEYID.EXE

对于同样一个 KEYID 程序，图 2-6 显示了由 Sourcer 自动生成的源代码。请注意，这个输出更像是一个程序员编写的源代码，而不像是由机器生成的。Sourcer 在汇编中插入了注释，并且标出了程序中的每个段和过程。另外，列表中的注释也是由 Sourcer 自动生成的。

依据上下文，Sourcer 能区分代码和数据以及不同的数据类型集合。区分代码和数据是反汇编最为复杂的一项工作，因为二进制文件对数据和代码并未提供直接的区分信息。也正是主要由于这个原因，许多不带源代码的调试器的反汇编结果相当低劣。

```
PAGE 59,132

;
```

```
;
;
;                              KEYID
;
;       Created:    28-Apr-96
;       Version:    1.00
;       Passes:     5                   Analysis Options on: none
;       (c) 1996    FVG
;
;

keybd_flags_1       equ     17h
psp_cmd_size        equ     80h
psp_cmd_tail        equ     81h

;-------------------------------------------------- seg_a ----
seg_a               segment byte public
                    assume cs:seg_a,ds:seg_a,ss:stack_seg_b

                    db          'KEYID   v1.00'
copyright           db          '(c) 1996   FVG'
;
;
;                           Program Entry Point
;
;
keyid   proc    far

start:
        mov     ax,seg_c
        mov     ds,ax
        mov     ah,9
        mov     dx,offset data_12   ;('KEY IDENTIFIER   ')
        int     21h                 ;DOS 服务,ah=功能 09h
                                    ;显示位 ds:dx 处的字符串
        cmp     byte ptr es:psp_cmd_size,2
```

```
        jne     loc_1                   ; 如果不相等, 则跳转
        mov     ax,es:psp_cmd_tail
        xchg    al,ah
        jmp     short loc_2
        db      90h
loc_1:
        mov     ah,9
        mov     dx,offset data_10   ;(' 按 1 显示 ASCII 字符 ')
        int     21h                 ; DOS 服务, ah=功能 09h
                                    ; 显示位于 ds:dx 处的字符串

        dec     ah                  ; DOS 服务, ah=功能 08h
        int     21h                 ; 获取键盘字符 al, 但不显示
loc_2:
        sub     al,31h              ; '1'
        cmp     al,2
        ja      loc_1               ; 如果大于, 则跳转
        xor     ah,ah               ; 清零寄存器
        shl     al,1                ; 左移一位, 用零填充
        mov     bx,ax
        assume  ds:seg_c
        jmp     word ptr data_15[bx];3 个入口
;---------------Indexed Entry Point-----------------------------------------------------

loc_3:
        mov     ah,9
        mov     dx,offset data_11   ; (' 按一键显示 ASCII 字符')
        int     21h                 ; DOS 服务, ah=功能 07h
                                    ; 显示位于 ds:dx 处的字符串
        mov     ah,1
        int     21h                 ; DOS 服务, ah 功能=01h
                                    ; 显示位于 ds:dx 处的字符串
        jmp     short loc_7
        db      90h
; ------------Indexed Entry Point-----------------------------------------------------
loc_4:
```

```
        mov     ah,9
        mov     dx,offset       data_12     ;(' 按一键显示 0')
        int     21h                         ;DOS 服务,ah=功能 09h
                                            ; 显示位于 ds:dx 处的字符串
        xor     ah,ah                       ; 清零寄存器
        int     16h                         ; 键盘 I/O    ah=功能 00h
                                            ; 获取键盘字符，并保存在 al 中,ah=扫描码
        mov     bx,offset data_18
        mov     al,ah
        and     ax,0F00Fh

        xlat                                ;al=[al+[bx]]表
        mov     dl,al
        mov     al,ah
        mov     cl,4
        ror     al,cl                       ; 翻转
        xlat                                ;al =[al+[bx]]表
        mov     ah,0Eh
        xor     bh,bh                       ; 清零寄存器
        int     10h                         ; 视频显示，ah=功能 0Eh
                                            ;  写字符 al, 电传模式
        mov     al,dl
        int     10h                         ; 视频显示，a=功能 0Eh
                                            ; 写字符 al，电传模式
        mov     al,68h
        int     10h                         ;
                                            ;
        jmp     short loc_7
        db      90h

; ---------------------- Indexed Entry Point ----------------------------------------------

loc_5:
        mov     ax,40h
        mov     es,ax
        test    byte ptr es:keybd_flags_1,20h   ;'  '
```

```
        jz          loc_6                       ; 如果为零, 则跳转
        mov         byte ptr data_13+14h,31h    ; ('0') '1'
        nop
loc_6:
        mov         ah,9
        mov         dx,offset data_13           ; ('  Numlock BIOS flag=0 ')
        int         21h                         ; DOS 服务, ah=功能 09h
                                                ; 在 ds:dx 处显示字符串
loc_7:
        mov         ah,4Ch
        int         21h                         ; DOS 服务, ah=功能 4Ch
                                                ;   结束, al=返回码
        db          0,0,0,0,0,0
keyid   endp
seg_a   ends

; ---------------------------------------------------------- stack_seg_b ----------------
stack_seg_b     segment word stack 'STACK'

        db      30 dup (0)

stack_seg_b     ends

; ---------------------------------------------------------- seg_c ----------------------
seg_c   segment byte public
        assume cs:seg_c , ds:seg_c , ss:stack_seg_b

        db      14 dup (0)
data_9  db      'KEY IDENTIFIER , 0Dh, 0Ah, '$'
data_10 db      '  Press 1 for ascii char', 0Dh,0Ah
        db      '  Press 2 for scan code', 0Dh,0Ah
        db      '  Press 3 for numlock flag', 0Dh
        db      0Ah, 0Dh, 0Ah, '$'
data_11 db      '  Press key for ascii char - $'
data_12 db      '  Press key for display of scan '
        db      'code - $'
```

```
data_13   db       '  Numlock BIOS flag=0$'
data_15   dw       offset loc_3          ; 数据表(索引访问)
data_16   dw       offset loc_4
data_17   dw       offset loc_5
data_18   db       '0123456789ABCDEF'

seg_c     ends
          end      start
```

图 2-6 Sourcer 反汇编 KEYID.EXE

Sourcer 可以输出类似于 ASM 类型的代码，这种代码很适合汇编。除此之外，Source 还可以生成一个列表，这点类似于 MASM 和 TASM。只要定义了标号或数据项，Source 就会加入一些前后对照信息。

另外，这个工具还生成了一个简表，列出了中断和 I/O 端口的使用情况。图 2-7 给出了 Source 对程序 BEEPA 的输出列表。这个程序借助于时钟来发出声音，并持续两秒钟。

```
                    PAGE    59.132

                    ;
                    ;
                    ;                    BEEPA
                    ;
                    ;       Created:   29-Apr-96
                    ;       Passes:    5     Analysis Options on: QRS
                    ;
                    ;

                    seg_a   segment byte public
                            assume cs:seg_a, ds:seg_a

                            org     100h

                    beepa   proc    far

9382:0100           start:
9382:0100  B0 B6            mov     al,0B6h
9382:0102  E6 43            out     43h,al          ; 端口 43h,8253 时钟控制
9382:0104  B8 00EE          mov     ax,0EEh
9382:0107  EB 00            jmp     short $+2       ; I/O 延时
```

```
9382:0109  EB  00          jmp     short $+2       ; I/O 延时
9382:010B  E6  42          out     42h,al          ; 端口 42h,8253 时钟 2 扬声器
9382:010D  8A  C4          mov     al,ah
9382:010F  EB  00          jmp     short $+2       ; I/O 延时
9382:0111  EB  00          jmp     short $+2       ; I/O 延时
9382:0113  E6  42          out     42h,al          ; 端口 42h,8253 时钟 2 扬声器
9382:0115  E4  61          in      al,61h          ; 端口 61h,8255 端口 B,读
9382:0117  0C  03          or      al,3
9382:0119  EB  00          jmp     short $+2       ; I/O 延时
9382:011B  EB  00          jmp     short $+2       ; I/O 延时
9382:011D  E6  61          out     61h,al          ; 端口 61h,8255 B—扬声器,等
9382:011F  B4  86          mov     ah,86h
9382:0121  B9  0008        mov     cx,8
9382:0124  BA  0000        mov     dx,0
9382:0127  CD  15          int     15h             ; 通用服务,ah=功能 86h
                                                   ; 等待 cx:dx 微秒
9382:0129  E4  61          in      al,61h          ; 端口 61h,8255 端口 B,读
9382:012B  24  FC          and     al,0FCh
9382:012D  EB  00          jmp     short $+2       ; I/O 延时
9382:012F  EB  00          jmp     short $+2       ; I/O 延时
9382:0131  E6  61          out     61h,al          ; 端口 61h,8255 B—扬声器,等
                                                   ;  ah = 0, 关闭了扬声器
9382:0133  B4  4C          mov     ah,4Ch
9382:0135  CD  21          int     21h             ; DOS 服务,ah=功能 4Ch
                                                   ; 结束,ah=返回码
                   beepa   endp

                   seg_a   ends

                           end     start
```

CROSS REFERENCE—KEY ENTRY POINTS

```
           seg:off      type          label
           ---- -----   --------  ----------------------------------
           9382:0100    far       start
```

中断用法概要

中断 15h: 通用服务, ah=功能 xxh
中断 15h: ah=86h, 等待 cx:dx 微秒
中断 21h: DOS 服务, ah=功能 xxh
中断 21h: ah=4Ch, 结束, al=返回码

I/O 端口用法概要

端口 42h: 8253 时钟 2 扬声器

端口 43h: 8253 时钟控制器
端口 61h: 8255 端口 B,读
端口 61h: 8255B—扬声器,等
端口 61h: al=0, 关闭扬声器

图 2-7 Souice 反汇编 BEEPA.COM 生成的列表

看一看 Source 输出列表的结尾，你就会很快掌握某个程序的主题思想。你还会找到一些中断和 I/O 端口使用情况的简表。在 BEEPA 结尾处的列表中，注意到中断 15h 用作等待功能，I/O 端口作为时钟和扬声器的门。这个程序需做大量修改，才可以运行于非 IBM 兼容的系统上，比如 NEC9800 系列机，这种机器在日本很流行。

列表的其他部分常常有助于快速浏览反汇编程序。回头看看前面的 KEYID 反汇编程序，在程序的开头很少像 Source 那样生成一些等价定义式，在图 2-8 中展示了这些等价定义式，它们来自 Source 对 KEYID 的反汇编列表。

```
keybd_flags_1    equ    17h        ; (0040:0017=0)
psp_cmd_size     equ    80h        ; (81EF:0080=0)
psp_cmd_tail     equ    81h        ; (81EF:0081=0)
```

图 2-8 KEYID 反汇编中的等价定义式

这些由 Source 生成的等价定义式，指明了程序以外的的数据参考地址。Source 带有一个内部数据库，里面有关于 BIOS、PSP 和其他区域里的通用参考地址。

在第一个等价定义式中，程序使用 BIOS 数据区 40:17h 内的键盘标志器。当程序运行在一个非 IBM 兼容系统上时，BIOS 参考会出现问题。

KEYID 程序还以偏移量 80 和 81h 访问了 PSP 中的数据。很明显，程序以命令行的形式读取信息，这些信息保存在 PSP 偏移 81h 的 DOS 中。在程序中查找关键字 *psp_cmd_size*,

将有助于标识处理命令行输入的代码区域。

另一个引起读者兴趣的重要位置是列表结果带有前后参照特性。前后参照简表指示了程序的主入口点和中断的入口点。图 2-9 列出了系统 BIOS 反汇编的前后参照部分。这部分信息将帮助你迅速定位到程序的关键位置。

CROSS REFERENCE - KEY ENTRY POINTS

seg:off	type	label
0000:7C00	far	run_boot_sector
F000:00F0	near	ROM_exception
F000:0100	near	system_reset
F000:0F1F	near	int_19h_bootup
F000:15F7	near	int_2_NMI
F000:3CCC	near	int_13h_floppy
F000:4A41	near	int_13h_hrd_dsk
F000:529B	near	int_10h_video

图 2-9 一部分前后参照,Source 分析 BIOS 的结果

通过仔细察看所有的中断调用功能 21h，25h,Source 还标明了中断的入口地址。这是用来挂起一个中断的 DOS 功能。通过 Source 的模拟操作，通常可以标识出被挂起的服务程序的入口地址，并且给它分配一个描述性的标号，比如 int_33h_entry。

回头重新浏览一遍代码不失为明智之举。在可能出麻烦的地方，Source 会在该行放入一个星号注释 "; *" 以示警告。例如，在 Source 的 KEYID 列表中，带索引跳转指令就带有这样一个警告星号。在图 2-10 中重新列出了该行。

```
81FF:004C     jmp    word  ptr  data_15[bx] ; *(820B:00CE=50h) 3 个入口
```

图 2-10 带警告星号的例子

在运行这个复杂的指令之前，Source 需要做一些假定。在一个开始于 data_15 偏移量的列表中，Source 标出了三个入口。然后让每个偏移量与程序中的一个特定地址相连。在这种情况下，Source 执行起任务来比较完美。带索引跳转通常出现在 BIOS 中。Source 能够自动生成偏移量表，这一点有助于理解 BIOS 程序。

如果需要做一些改动，Source 还带有一个定义集来提供对输出结果的完全控制。在这个集中，你可以控制段的类型为 16 位或 32 位，或者加入你自己的描述表和注释，以便定义特殊的数据类型。它还允许对 Source 的控制格式选项加以设置，比如大、小写，标号风格以及其他选项。

在生成汇编代码时，Source 还允许选择目标汇编器。对于相同的代码，每个汇编器生

成的二进制代码会有细微差别。这主要是由于内部指令集造成的，这条指令集会造成许多种不同的编码结果。Source 可以“调节”其输出以适应特定汇编器版本，这一点有利于重新汇编该代码。在大多数情况下，汇编结果代码能被重新汇编成与原二进制文件相同的字节形式。

Source 能处理众多的输入格式，包括多段 EXE 文件、COM 文件、设备驱动文件(SYS 和 EXE 类型)、链式设备驱动文件。从 ROM 或 RAM 得到反汇编文件，以及 Windows 的 32 位 VxD 文件。Source 还带有一个可任选的预处理器,Windows 资源，来处理 Windows 的 EXE 文件、VxD 文件、设备驱动文件、资源、WIN32S、OS/2 的 32 位程序以及其他多种格式。

任意人都可以进行反汇编，但是掌握一些汇编语言对于理解结果来说是很有必要的。Source 极大地简化了这项工作。只有有了像 Source 这样的高级工具来处理问题，理解接口的内幕才会成为可行的。

反汇编 BIOS

反汇编 BIOS 远比反汇编普通的 EXE 文件或设备驱动文件复杂。BIOS 由一系列的独立的程序和数据表组成。每个 BIOS 执行的方法各不同，所以程序和数据也位于完全不同的位置。相同制造商生产的不同版本之间也有所差别。

不同的 BIOS 提供不尽相同的功能。BIOS 系统目前有五种主要的类型：PC、XT、AT（ISA 和 PCI）、EISA 以及 MCA。系统 BIOS 控制电源测试、基本磁盘控制、简单的视频控制、打印机、串行口以及其他重要功能。至于视频卡，则有数不清的 BIOS。一些比较常见的适配卡 BIOS 包括硬盘、视频以及网卡。

几年前，我开发了 Source 的一个配件——BIOS 预处理器，来自动实现对一个 BIOS 的反汇编。这个 BIOS 预处理同 Source 一起生成该 BIOS 的全部列表文件，并带有详细的注释、数据区驻留位置、数据区的大小以及有关这个 BIOS 的其他关键信息。关键参考是用英文写的，而不是一个很随意的标号（或者在使用一个调试器时没有标号）。例如，视频服务程序入口标号是 int_10h_video。许多数据参考在代码中也是描述性的，比如 baud_rate_tbl 和 hdsk_cylinders。

最终生成的 BIOS 列表文件的可读性极强。有一些客户在比较了制造商所提供的 BIOS 源代码和 Source 生成的源代码后，都说用 Source 生成的代码更清晰明了。图 2-11 给出了打印机 BIOS 程序处理三个功能。通过一个带索引的调用指令来处理每个功能。在这种情况下，Source 为三个子程序的偏移量建立了一个表，然后标识出起点为 sub_75 的三个子程序。

```
mbios.lst       SYSTEM BIOS.          Sourcer Listing          Page 35

F000:08E5  0910      data_75 dw       offset sub_75      ; 数据表(索引访问)
                                                         ; 地址参考 F000:0903
```

```
F000:08E7   091D      data_76 dw         offset sub_76          ; 地址参考 F000:0903
F000:08E9   092F      data_77 dw         offset sub_77          ; 地址参考 F000:0903

                      ;
                      ;                                    int 17h
                      ;
                      ; 打印机服务
                      ;
                      ; 调用: ah=功能码
                      ;       dx=打印机 0~2,(可能支持 0~3)
                      ;
                      ; 返回: ah=状态位
                      ;            7    6    5    4    3     2     1    0
                      ;            不   确   无   选   I/O   未    无
                      ;            忙   认   纸   中   错误  使用  响应
                      ;            ----------- 来自打印机 ----------
                      ;
                      ; 功能:
                      ;      ah = 0   向打印机送字符,al=字符
                      ;      ah = 1   打印机端口初始化
                      ;      ah = 2   获取打印机状态,并存入 ah

F000:08EB             int_17h_printer proc   far                 ; 地址参考  F000:0504
F000:08EB   53                 push    bx
F000:08EC   51                 push    cx
F000:08ED   1E                 push    ds
F000:08EE   06                 push    es
F000:08EF   80 FC 03           cmp     ah,3
F000:08F2   73 15              jae     loc_135                   ; 大于等于则跳转
F000:08F4   E8 FFE4            call    sud_74
F000:08F7   8B FA              mov     di,dx
F000:08F9   D1 E7              shl     di,1                      ; 左移一位,用零填充
F000:08FB   26:8B 55 08        mov     dx,es:@prn_port_1[di]; (0040:0008=378h)
F000:08FF   8A DC              mov     bl,ah
F000:0901   32 FF              xor     bh,bh                     ; 清零寄存器
F000:0903   FF 97 08E5         call    word ptr data_75[bx]  ; * (F000:08E5=910h)
                                                                 ;  3 个入口
F000:0907   EB 02              jmp     short loc_136
F000:0909             loc_135:                                   ; 地址参考 F000:08F2
F000:0909   B4 09              mov     ah,9
```

```
F000:090B                loc_136:                                ;  地址参考 F000:0907
F000:090B  07                    pop     es
F000:090C  1F                    pop     ds
F000:090D  59                    pop     cx
F000:090E  5B                    pop     bx
F000:090F  CF                    iret                            ;  中断返回
                         int_17h_printer endp
                          ;
                          ;                    子程序
                          ;
                          ;                 调用自: F000:08E5,0903
                          ;
                          ;

                          sub_75  proc    near
F000:0910  EE                     out     dx,al                  ;  端口 378h, 打印机数据
F000:0911                 loc_137:                                ;  地址参考 F000:091A
F000:0911  E8 0033                call    sub_85
F000:0914  E8 0018                call    sub_77
F000:0917  F6 C4 80               test    ah,80h
F000:091A  75 F5                  jnz     loc_137                ;  非零则跳转
F000:091C  C3                     ret
                         sub_75  endp
```

图 2-11 Source 反汇编 BIOS 的部分列表文件

IOSPY-I/O 端口监视器 TSR

在许多情况下，很难在运行或调试其他程序时看到 I/O 端口的状态。IOSPY 是一个内存驻留程序（Terminate and Stay Resident utility,TSR），它能够持续监视指定端口的当前状态，并将这个当前状态显示在屏幕上。IOSPY 有一系列的选项，可以用来处理几乎所有的 I/O 端口。

使用 IOSPY 时，只要运行 IOSPY 并指明端口号就行了。一次可以监视不多于 8 个的端口。一系列的全局或局部选项对 IOSPY 操作进行控制。当激活它（敲击 Scroll-Lock 键）后，将为每一个指定的端口显示一个监视点。监视点的信息包括端口号、从端口中读出的当前值、上一次读出的不同值、以及一个简短的文本标识符。一个监视点在屏幕上是这样显示的：

```
port 27Ah=FFh          last=0Eh        Printer
```

全局选项是与特定 I/O 端口无关的选项，它们包括：

-a 激活常开模式。缺省情况下，IOSPY 只有在 Scroll-Lock 开时才可以被激活。

-c 除所有端口，但是仍保持驻留状态（在选-c 项之前如果没有端口号显示出来，就会发生这种情况）。

-d 该选项显示彩色 VGA 系统普通的 25 行屏幕之外的监视点信息。VGA 系统控制器被设置 31 行文本模式，端口的状态信息在普通的 25 行屏幕下面显示出来，而不妨碍 DOS 或某个正在执行的程序。DOS 的屏幕显示仍然是 80 列 25 行，并且大多数运行正常的程序只会显示 25 行窗口下的信息。这种特殊的模式在图形程序或有普通视频控制的程序中并不适用，一台膝上型电脑的屏幕可能也不会支持这种性能，因为 IOSPY 会将 VGA 控制器设置为 500 线扫描模式。

-m 在一个双监视器系统中，该选项将端口信息显示在一个单色适配卡上，而不管是否真的存在单色适配卡。

-n 不改变背景色。如果没有设置这个选项时，背景色随着端口值的改变而改变。

-u 卸载。在 IOSPY 挂起了中断 1Ch 之后，如果没有装入其他的 TSR 程序，那么就可以删除 IOSPY。即使 ISOPY 被装入高端内存，仍然可以执行卸载。如果前面使用了-d 选项，IOSPY 将 VGA 屏幕恢复为正常的 80x25 模式。

局部选项适用于某个特定的端口。每个被监视的 I/O 端口可以有不同的局部选项。缺省情况下，没有激活任意局部选项。指定了一个端口号后，出现在该端口号后而后选项就与该端口联系在了一起。

-b 在指定的 I/O 端口的值发生改变时发声。IOSPY 会使用硬件时钟 2 来进行发声。如果没有其他程序使用时钟 2，发声程序会改写时钟 2 模式和定时功能。

-c 从监视器的 IOSPY 端口列表中，清除这个指定端口。

-f 对输入数据过滤。当数据变化太快而无法看时，该选项很有用。依然显示快变数据，但是不再显示最后的改变值，而是显示一个过滤值，每 400ms 更新一次。

-hXX 允许 16 进制预写模式。当 I/O 端口使用两个端口访问信息时，该选项将 16 进制值 XX 写到指定的端口，然后从端口号加 1 后指向的那个端口读取状态信息（如果是 16 位端口，则加 2）。这一点对于查看 CMOS 寄存器和许多 VGA 寄存器很有用。

-S 选择端口时按 16 位 I/0 操作，而不是按缺省的 8 位操作（大多数 I/O 端口是 8 位的）

-W 回写端口前面读取的值。当读取端口会造成这个寄存器变化时会用到该选项。例如，在读取一些串行口状态寄存器后会发生清位。在这个读和写操作之间不允许中断。

'text' 覆盖由 IOSPY 提供的缺省文本串。文本串长度不超过 10 个字符，在 16 位模式下只显示前 8 个字符。

可以一次指定多个端口，在 IOSPY 装入后还可以加入新的端口。在装入一个端口后，IOSPY 可以带新选项重新运行相同的端口号，来更新驻留的 IOSPY。例如，为了监视端口 279h 的打印机状态和端口 27Ah 的打印机控制，可以输入下面的命令行。

例如要监视端口 279h 处的打印机端口状态和端口 27Ah 处的打印机端口控制，可以输入下面的命令行。开启 Scroll-Lock 键来观察状态信息。

```
iospy  279  27A
```

在屏幕的右上角会有两行显示这个例子的监视结果。监视点显示如下：

```
port   279h=FFh              printer
port   27Ah=FFh              printer
```

在一个更复杂的例子中，使用了全局常开选项(-a)，并且在 80x25 屏幕下显示端口信息(-d)。该例子监视 CMOS 设备字节数 14h（端口 70 和 71h），如果 CMOS 设备字节的值发生了改变，则会发声(-b)。文本 'CMOS equip' 将和端口对 70/71h 相关联。这个例子也监视串行口线状态寄存器 3FBh。由于线状态寄存器在读取时会清位，所以必须回写端口值，这样一个要用到这些错误标志的程序才可以得到正确的信息。

```
isopy  -a  -d  70  -h14  -b  'CMOS  equip'  3FB  -w
```

在 25 行显示屏幕下，这个例子的监视点信息如下：

```
---------------------------------------- isopy ----------
port  71h=0Fh      CMOS  equip      port  3FBh=02h      RS232  line
```

记住，某些端口是只写的，读取该端口不会获得任意有效的信息。由于 IOSPY 在每个时间片内监视端口一次(每 54ms 一次)，所以显示的信息总是有效的。但是也有可能在 IOSPY 监视间歇内端口值已经改变。

IOSPY TSR 可以在高端内存和低端内存中可靠地工作。当装入到高端内存时，IOSPY 大约需要 6K 字节，包括需要 1.3K 的驻留空间。最少驻留空间取决于它在汇编语言中的具体应用情况。

最后说明一下对串行口方面使用 IOSPY 的情况。记住，如果你从一个串行口读取输入数据，该数据会显示在屏幕上，但是对使用该串行口的软件会发生数据丢失现象。由于写串行口意味着告诉 UART 要传送这个字节。所以回写选项不会起作用。对于监视串行口状态来说，一些寄存器在读取之后需要回写数据。使用程序 SSPY 可以监视串行口状态。它是 IOSPY 程序的一个子集，特意为监视串行口状态而设计。参考第 12 章，获得有关 SSPY

的进一步信息。

尽管在软盘上没有包含 IOSPY 的源代码，但是从本质上讲它和 SSPY 很相似，而软盘包含了 SSPY。IOSPY 工作时要挂起中断 8 时钟中断。驻留部分每 54ms 调用一次，来读取指定端口的当前状态，它采用直接写屏幕的方法更新屏幕信息。

UNPC-I/O 端口浏览器

有时候希望知道是否正在使用某个指定的端口，或者该端口的值是多少。我设计了一个 I/O 端口浏览器程序，来显示任意一组连续的 256 个端口的内容。要运行这个浏览器程序，只要运行 UNPC，选择 I/O port viewer，然后选择 view I/O ports。

UNPC 会连续读取并显示前 256 个 I/O 端口。PgUp 和 PgDn 键用来在不同组端口间切换。按 F1 键获得详尽信息。图 2-12 展示了 I/O 端口浏览器程序是如何在屏幕上显示信息的。

PC 内幕

连续 I/O 端口视图

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0000	001	001	001	001	001	001	001	001	00	00	00	00	00	00	00	00	DMA 控制器 1
0010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0020	00	A8	00	A8	00	A8	00	A8	00	A8	00	A8	00	A8	00	A8	中断控制 1
0030	00	A8	00	A8	00	A8	00	A8	00	A8	00	A8	00	A8	00	A8	
0040	1C	01	7F	00	63	0E	04	00	48	07	7F	00	5B	03	04	00	时钟控制器
0050	6E	09	7F	00	53	05	04	00	98	0F	7F	00	4B	0B	04	00	
0060	2A	20	FF	FF	1C	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	键盘和标志
0070	FF	00	FF	00	FF	00	FF	00	FF	00	FF	00	FF	00	FF	00	RIC 和 CMOS 寄存器
0080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	DMA 页面寄存器
0090	00	EX	00	00	00	00	00	00	00	00	00	00	00	00	00	00	系统控制
00A0	00	00	0D	00	0D	00	0D	00	0D	00	0D	00	0D	00	0D	00	中断控制 2
00B0	0D	00	0D	00	0D	00	0D	00	0D	00	0D	00	0D	00	0D	00	
00C0	0D	001	00	001	00	001	00	001	00	001	00	001	00	001	00	001	DMA 控制器 2
00D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	DMA 控制器 2
00E0	00	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	系统控制
00F0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	数学协处理器

使用 PgUp 或 PgDn 进行选择　　F1 帮助　　Esc 退出

图 2-12　I/O 浏览器

记住，系统端口和许多适配卡并不会对所有 I/O 端口进行完全译码，而使它们出现在

多个地址中。例如，一个端口地址为 279h 的打印机端口可能出现在每隔 400h 的端口地址中。这意味着，读取端口 279h 与读取端口 679h,A79h,E79h 等端口的结果一样。一些系统端口，比如中断控制器端口 20h 与 21h，可能会每两个端口重复一次，从 22h 和 23h 一直到 3Eh 和 3Fh。“这种端口重复”现象是由主板或适配器的供应商造成的。从逻辑上讲，他们节省了一点钱，并且简单地对 I/O 地址进行译码。较新的系统很少出现这种情况。EISA 系统还有专门设计来避免这个问题发生。

正如我在讨论 IOSPY 时提到的那样，一些 I/O 端口需要做特别处理。首先，一些 8 位端口必须读两次来获得一个 16 位的结果。用一个空格来切换那些需要读取两遍的端口的高位部分和低位部分。一些端口在读取时会产生系统问题。这些端口已经被排除在外了，并且在屏幕上显示为红色的 EX。最后，一些端口在读取时会丢失内容，读后必须回写。回写端口用蓝色显示。

所有的指定端口必须一个一个地改变。在起始菜单中（如果你在 I/O 端口浏览器中，请按 Esc），你可以选择排除、回写、读两次。以上每一个选择都可以为端口设置各自的选项。用空格键来切换光标指示的端口状态。Insert 和 Delete 键支持激活或删除一组端口，或者重新设置缺省的指示标志。在退出 UNPC 之时，会自动保存所有的选项变更。如果你希望退出时不保存任意变更，只需要按下 Ctrl+C 键。

I/O 端口浏览器只是简单地读取每个 I/O 端口，并在屏幕上尽可能快地将结果给显示出来。一旦读取完了当前组中所有的 256 个端口，会重复这个过程。更新的速度取决于 CPU 速度、扩展总线速度以及视频显示卡的运行情况。即使工作在图形模式下，但是由于使用了用户自定义字体，所以实际上仍是使用文本模式显示，该字体是借自（经允许）V 通信公司的。这一点保证了屏幕的更新速度尽可能快。

第 3 章

CPU 和内幕指令

大多数的 80x86 CPU 在 CPU 用户手册和其他有关书籍中有较为详细的说明。既然本书重点介绍未公开的端口及其用法，所以本章第一部分先回顾一下 80x86 CPU 系列有关输入输出的问题。这一部分没有多少新的尚未公开的东西，但是如果你对如何使用端口去控制和访问 PC 机不清楚也不熟悉，那么这些信息会对你有用。

接下来的一部分更能引起读者的兴趣，它涉及 I/O 端口的定时问题。如果 I/O 端口之间的访问需要延迟，那么定时问题就显得重要了。这里提供了一个程序范例来确保端口的定时不发生重叠。

对于那些不喜欢汇编代码的人来说，他们会发现这里有一部分讲述如何通过高级语言，例如 C、C++，来访问 CPU 硬件。当然也包括访问寄存器、中断以及 I/O 端口。这里提供了一个程序范例来比较用 C 和用汇编代码实现的区别。

我还对同种 CPU 的各种版本以及不同公司提供的 CPU 的差别作出了解释，其中包括 AMD/NexGen、Cyrix、IBM、Intel、Nec、Texas 仪器公司提供的相关资料。

中间部分最有意思，它涉及各种 CPU 尚未公开的资料，其中包括 LOADALL 的详尽细节，以及其他鲜为人知的未公开信息。

本章论述关于 CPU 的专项主题，并对隐存、挂起/恢复、内电路模拟和重新启动作出了解释。这里还提供了一些程序来创建一个可靠的系统启动环境，并在 CRTL+ALT+DEL 重启之前捕获控制。

基本输入输出块

处理器的输入输出指的是其与外部设备通信的能力。例如，当处理器向打印机发送一个字符时，CPU 输出该字符。反之，处理器就接受输入。当敲击键盘的一个键时，该字符就被输入到了处理器。如图 3-1 所示。

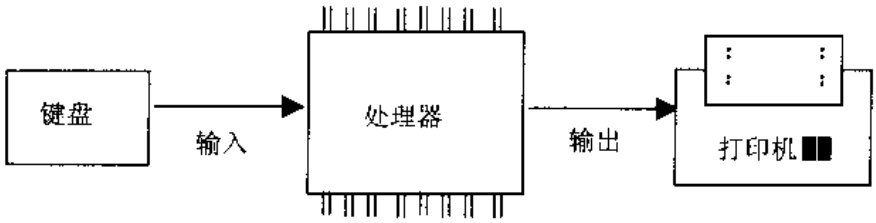


图 3-1 处理器的输入和输出

其他的 CPU 输入设备还有鼠标、数字表、滚动球、调制解调器、网络、磁盘驱动器、磁带驱动器。输出设备包括显示器、绘图仪、调制解调器、网络、磁盘驱动器、磁带驱动器。一个存储设备，比如磁盘驱动器，可用来保存处理器的输出，也可以从磁盘驱动器读取数据来作为处理器的输入。输入输出通常是按字节或字传送的，近来还可以按双字传送。这意味着处理器可以一次传送 8 位、16 位或者 32 位信息。

Intel 80x86 处理器系列提供了两种方法访问输入输出设备。一种方法是，由处理器提供专门的指令来访问专用的 I/O 寻址空间。利用 I/O 寻址空间是 PC 机处理输入输出最常用的方法。另外，硬件设计人员可以将 I/O 定义为内存空间中的地址，这种方法通常被称为内存映射 I/O。现在很少使用这种方法，以避免出现内存冲突。

I/O 寻址

80x86 系列能访问 65,536 个独立的 8 位端口。两个 8 位端口组成一个 16 位端口。在 386 或者更高级的处理器上，四个 8 位端口可以组成一个 32 位端口。通过直接访问一个指定了端口号的立即数，可以访问前 256 个端口，而寄存器 DX 常常被用来指定 65,536 个端口中的任意一个。表 3-1 归纳了不同的寻址形式。

表 3-1 I/O 寻址类型

端口号定义形式	端口数	建议的对齐方式	数据大小	端口编号
立即数	256 个	字节	8 位	0,1,2,3,...,255
立即数	128 个	字	16 位	0,2,4,6,....254
立即数	64 个	双字	32 位	0,4,8,16,...,252
DX 寄存器	256 个	字节	8 位	0,1,2,3,...,65535
DX 寄存器	128 个	字	16 位	0,2,4,6,...,65534
DX 寄存器	64 个	双字	32 位	0,4,8,16,...,65532

依据端口大小调整好其边界地址，以确保在同一个周期内传送完所有位。也就是说，16 位的端口应该用偶地址，例如 0、2、4，一个 32 位的端口应该用可被 4 整除的地址，例如 0、4、8。CPU 允许不按照这种方法来排列地址，但是这样会导致访问需占用更多的总线周期，从而减慢了访问该端口的速度。

I/O 指令

所有的 8088 和后来的处理器都提供四种基本的 I/O 指令。在 80188 还加入了两种指令来处理用一条指令完成乘法的问题。表 3-2 简要列出了基本的指令。

表 3-2 I/O 端口指令简集

指　令	操 作 数	描　述
IN	寄存器，立即数	输入，寄存器=AL/AX/EAX
IN	寄存器，DX	输入，寄存器=AL/AX/EAX
INS		输入保存于内存 ES：[DI]中的数据
OUT	立即数，寄存器	输出，寄存器=AL/AX/EAX
OUT	DX，寄存器	输出，寄存器=AL/AX/EAX
OUTS		输出保存于内存 ES：[DI]中的数据

从端口输入

IN 指令将一个字节、字或双字数据从指定的端口传送到指定的寄存器 AL、AX 或 EAX 中，这种简单直接的指令形式适用于从任意一个前 256 个端口，0~0FFh，中读取数据。DX 指令格式用于从任意一个 0~0FFFFh 的端口读取数据。

IN	AL，PORT#	将一个字节从立即数指向的端口输入到 AL（仅对端口号 0~0FFh 有效）
IN	AX，PORT#	将一个字从立即数指向的端口输入到 AX（仅对端口号 0~0FFh 有效）
IN	EAX，PORT#	将一个双字从立即数指向的端口输入到 EAX（仅对端口号 0~0FFh、386 和更高级的 CPU 有效）
IN	AL，DX	从端口 DX 传送一个字节到 AL
IN	AX，DX	从端口 DX 传送一个字到 AX
IN	EAX，DX	从端口 DX 传送一个双字到 EAX（仅对 386 和更高级的 CPU 有效）

代码举例

```
in      al,  60h
in      ax,  20h
```

```
in      eax,  42h
mov     dx,   60h
in      al,   dx
mov     dx,   3E27h
in      ax,   dx
mov     dx,   177Ch
in      eax,  dx
```

输入串

INS（输入串）指令能从DX寄存器指定的端口传送一个字节、字或双字。当工作在16位寻址模式下时，传送的目标地址是ES：[DI]，当工作在32位寻址模式下时，传送的目标地址是ES：[EDI]。ES段寄存器一般总是被用到，并且不允许出现段重叠现象。

传送结束后，DI/EDI寄存器值根据传送对象的大小自动更新。清除方向位标志（源于以前的CLD指令）会造成DI/EDI寄存器值增加，设置方向位标志（源于以前的STD指令）会造成DI/EDI寄存器值减小。对于字节传送而言，DI/EDI寄存器值逐个增加/减少，而字传送按2改变，双字则按4改变。

REP（重复repeat）前缀在INS指令中通常用来输入一个长为CX的串。INS在8088/8086处理器中无效，而INSD（双字）以及所有的32位寻址形式都只在80386和更高级的处理器中有效。

16位寻址方式

INSB　从端口DX向ES：[DI]输入字节，然后逐个增加/减少DI

INSW　从端口DX向ES：[DI]输入字，然后按2增加/减少DI

INSD　从端口DX向ES：[DI]输入双字，然后按4增加/减少DI（仅对386或更高级CPU有效）

32位寻址方式

INSB　从端口DX向ES：[EDI]输入字节，然后逐个增加/减少EDI

INSW　从端口DX向ES：[EDI]输入字，然后按2增加/减少EDI

INSD　从端口DX向ES：[EDI]输入双字，然后按4增加/减少EDI（仅对386或更高级CPU有效）

代码举例

```
mov     es,   seg   in_buffer       ; es：di = 指向缓冲区的指针
mov     di,   offset  in_buffer
```

```
mov     dx, 123h                ; dx = 端口号 123h
cld                             ; 清除方向标志
insb                            ; 向 in_buffer 中输入一个缓冲区字节
                                ; DI 指向 in_buffer+1

mov     es, seg in_buffer       ; es: di = 指向缓冲区的指针
mov     di, offset in_buffer
mov     dx, 60h                 ; dx = 端口号 60h
mov     cx, 30                  ; cx = 输入的次数
cld                             ; 清除方向标志
rep     insw                    ; 向 in_buffer 中输入 30 个字,
                                ; 完成时 DI 指向 in_buffer+60
```

警　告

若 CPU 运行在 386 或更高级处理器的 8086 虚模式下时，跟踪执行 INS 指令会出现问题。不幸的是，跟踪过程不会重新启动最初由 INS 指令输入的字节/字。尽管考虑到执行效果需要避免跟踪单独的端口，但是有时在保护模式下某些管理程序不得不这样做以保证系统的可靠性。跟踪执行 DMA 端口就是一个简单的例子。

保护模式下的环境包括所有的内存管理程序，比如 EMM386、WINDOWS 增强模式、较新的操作系统如 OS/2、WINDOWS95 和 NT。当软件运行在这些环境下时，我极力建议你不要在代码中使用 INS。

向端口输出

OUT	PORT# , AL	将一个存放在 AL 中的字节输出到立即数指向的端口（仅对端口号 0~0FFh 有效）
OUT	PORT#, AX	将一个存放在 AX 中的字输出到立即数指向的端口（仅对端口号 0~0FFh 有效）
OUT	PORT#, EAX	将一个存放在 EAX 中的双字输出到立即数指向的端口（仅对端口号 0~0FFh，386 和更高级的 CPU 有效）
OUT	DX, AL	向端口 DX 传送一个存放在 AL 中的字节
OUT	DX, AX	向端口 DX 传送一个存放在 AX 中的字
OUT	DX, EAX	向端口 DX 传送一个存放在 EAX 中的双字（仅对 386 和更高级的 CPU 有效）

代码举例

```
out     60h,  al
out     20h,  ax
out     42h,  eax
mov     dx,   60h
out     dx,   al
mov     dx,   3E27h
out     dx,   ax
mov     dx,   177Ch
out     dx,   eax
```

输出串

OUTS（输出串）指令能从源寄存器向 DX 寄存器指定的端口传送一个字节、字或双字。当工作在 16 位寻址模式下时，传送的源地址是 DS：[SI]，当工作在 32 位寻址模式下时，传送的源地址是 DS：[ESI]。DS 是缺省的段寄存器，并且允许出现段重叠现象。

传送结束后，SI/ESI 寄存器值根据传送对象的大小自动更新。清除方向位标志（源于以前的 CLD 指令）会造成 SI/ESI 寄存器值增加，设置方向位标志（源于以前的 STD 指令）会造成 SI/ESI 寄存器值减小。对于字节传送而言，SI/ESI 寄存器值逐个增加/减少，而字传送按 2 改变，双字则按 4 改变。

REP（重复 repeat）前缀通常用在 OUTS 指令中来输入一个长为 CX 的串。OUTS 在 8088/8086 处理器中无效，而 OUTSD（双字）以及所有的 32 位寻址形式都只在 80386 和更高级的处理器中有效。

16 位寻址方式

OUTS	DX，BYTE PTR DS：[SI]	从端口 DS：[SI]向 DX 输出字节，然后逐个增加/减少 SI，任意的段寄存器都可以代替 DS。
OUTS	DX，WORD PTR DS：[SI]	从端口 DS：[SI]向 DX 输出字，然后按 2 增加/减少 SI，任意的段寄存器都可以代替 DS。
OUTS	DX，DWORD PTR DS：[SI]	从端口 DS：[SI]向 DX 输出双字，然后按 4 增加/减少 SI，任意的段寄存器都可以代替 DS，仅对 386 或更高级 CPU 有效。
OUTSB		从端口 DS：[SI]向 DX 输出字节，然后逐个增加/减少 SI。
OUTSW		从端口 DS：[SI]向 DX 输出字，然后按 2 增加/减少 SI。

OUTSD	从端口 DS：[SI]向 DX 输出双字，然后按 4 增加/减少 SI，仅对 386 或更高级 CPU 有效。

32 位寻址方式（仅对 386 或更高级 CPU 有效）

OUTS	DX，BYTE PTR DS：[ESI]	从端口 DS：[ESI]向 DX 输出字节，然后逐个增加/减少 ESI，任意的段寄存器都可以代替 DS。
OUTS	DX，WORD PTR DS：[ESI]	从端口 DS：[ESI]向 DX 输出字，然后按 2 增加/减少 ESI，任意的段寄存器都可以代替 DS。
OUTS	DX，DWORD PTR DS：[ESI]	从端口 DS：[ESI]向 DX 输出双字，然后按 4 增加/减少 ESI，任意的段寄存器都可以代替 DS，仅对 386 或更高级 CPU 有效。
OUTSB		从端口 DS：[ESI]向 DX 输出字节，然后逐个增加/减少 ESI。
OUTSW		从端口 DS：[ESI]向 DX 输出字，然后按 2 增加/减少 ESI。
OUTSD		从端口 DS：[ESI]向 DX 输出双字，然后按 4 增加/减少 ESI，仅对 386 或更高级 CPU 有效。

代码举例：

```
mov     gs,   seg out_buffer   ; gs：si = 指向缓冲区的指针
mov     si,   offset out_buffer
mov     dx,   234h             ; dx = 端口号 234h
cld                            ; 清除方向标志
outsb   dx,   gs:[si]          ; 从 out_buffer 中输出一个缓冲区字节
                               ; SI 指向 out_buffer+1
mov     ds,   seg out_buffer   ; ds：si = 指向缓冲区的指针
mov     si,   offset out_buffer
mov     dx,   234h             ; dx = 端口号
mov     cx,   20               ; cx = 输入的次数
cld                            ; 清除方向标志
rep     outsw                  ; 从 out_buffer 中输出 20 个字，
                               ; 完成时 SI 指向 out_buffer+40
```

指令定时

下表 3-3 列出了每条指令所需的处理器时钟周期数，前提是已经取出了每条指令并正

准备执行、没有等待、没有异常处理、双字对齐方式。这意味着你不可能获得这些最佳情况下的周期数。

386到Pentium的保护模式（Protect Mode，PM）列出了两个定时选择，前一个表示当前的优先级不高于I/O的优先级（IOPL），而后一个表示当前的优先级高于I/O的优先级。

在EFLAGS寄存器中，还定义了使用I/O指令的权限，Intel并未公开Pentium Pro的指令定时性能。

表3-3 I/O指令定时

				---- 80386 ----		---- 80486 ----			---- Pentium ----		
指　令	操作码	8088	80286	实模式	PM	实模式	VM	PM	实模式	VM	PM
IN　AL,imm8	E4 ib	10	5	12	6/26	14	27	8/28	7	19	4/21
IN　AX,imm8	E5 ib	10	5	12	6/26	14	27	8/28	7	19	4/21
IN　EAX,imm8	E5 ib			12	6/26	14	27	8/28	7	19	4/21
IN　AL，DX	EC	8	5	13	7/27	14	27	8/28	7	19	4/21
IN　AX，DX	ED	8	5	13	7/27	14	27	8/28	7	19	4/21
IN　EAX，DX	ED			13	7/27	14	27	8/28	7	19	4/21
INSB	6C		5	15	9/29	17	30	10/32	9	22	6/24
INSW	6D		5	15	9/29	17	30	10/32	9	22	6/24
INSD	6D			15	9/29	17	30	10/32	9	22	6/24
OUT　imm8,AL	E6 ib	10	3	10	4/24	16	29	11/31	12	24	9/26
OUT　imm8,AX	E7 ib	10	3	10	4/24	16	29	11/31	12	24	9/26
OUT　imm8,EAX	E7 ib			10	4/24	16	29	11/31	12	24	9/26
OUT　DX,AL	EE	8	3	10	5/25	16	29	11/31	12	24	9/26
OUT　DX,AX	EF	8	3	10	5/25	16	29	11/31	12	24	9/26
OUT　DX,EAX	EF			10	5/25	16	29	11/31	12	24	9/26
OUTSB	6E		5	14	8/28	17	30	10/32	13	25	10/27
OUTSW	6F		5	14	8/28	17	30	10/32	13	25	10/27
OUTSD	6F			14	8/28	17	30	10/32	13	25	10/27

注：imm8代表一个立即数端口号。

定时方面的难题

如果连续两次访问相同的I/O设备，就可能在第二次访问时得到一个错误的结果。当

系统的某个适配器总线周期快于 I/O 设备的接收周期时，就会发生这种情况。某些新型的系统在设计时就要求 I/O 操作不能覆盖一个附加的 I/O 设备。一切现有系统都符合这个范畴（EISA、PCI、VLB），并且在访问同一个 I/O 设备时不需要额外延迟。

为了放慢较老系统中代码的运行速度，程序特意包括了一个短跳转。当系统为 20MHz 386 时，下面的方法在对同一设备操作时产生了 1 微秒的延迟。

```
out     dx,     al          ; 输出
jmp     short   $+2         ; 延迟
jmp     short   $+2         ; 延迟
in      al,     dx          ; 输入
```

由于 CPU 的速度越来越快，那么大约需要多长的延迟才比较合适呢？对于一个 8MHz 的底板总线，系统能在 75ns 内访问同一个 I/O 端口两次。这已经相当快了。建议在连续访问同一设备时，对于那些运行情况未知的 I/O 设备采用大约 1 微秒的延迟。这可以保证 I/O 操作彼此不会发生覆盖。当然，如果某些特殊的硬件带有固件，在硬件满速处理 I/O 时就不必有延迟。

同时使用一串跳转指令是在快速系统中形成短延迟的最简单的方法。表 3-4 列出了不同系统中延迟指令需要的跳转次数。记住，执行一个短跳转指令的周期数因 CPU 而异，软件应提供系统最大预期速度下的延迟。

表 3-4　带有跳转指令的 I/O 延迟

CPU	CPU 速度	CPU 周期	每次跳转的周期数	跳转的次数
8088	5MHz	200ns	15	0
80286	8MHz	125ns	9	1
80286	16MHz	63ns	9	2
80386	20MHz	50ns	9	2
80386	40MHz	25ns	9	4
80486	25MHz	40ns	5	5
80486	66MHz	15ns	5	14
Pentium	66MHz	15ns	1	67
Pentium	100MHz	10ns	1	100

主板设计者通常给每个 I/O 访问加入额外的等待状态，来让以往的软件能在快速系统下可靠地运行。这些等待状态可以防止 CPU 过快地访问 I/O 设备。通常在这种情况下，快速系统只要有少数几个短跳转指令就够了。下面的代码例 3-1 指出了一种可供选择的方法

来确保总有一定的延迟。

代码例 3-1 I/O 延迟

设计下面的程序使为了确保无论 CPU 或者总线的速度多大都有足够的 I/O 延迟。第一次调用 ID_DELAY 时，程序测量 CPU 运行的时间并将之保存为一个调节因子。如果有实时时钟（RTC），ID_DELAY 就利用它来测量 5ms 内的指令执行速度。如果没有 RTC，ID_DELAY 就假定系统是一台 PC 或 XT，并采用最小延迟。

下面的程序提供至少 1 微秒的延迟，由变量 IO_COUNT 控制，它在第一次调用程序时经计算得到。程序在第一次调用时，会花 5ms 来标准化时间设置。如果需要，可以在使用 ID_DELAY 之前在标准化初始设置中调用它一次。

在一个像 5MHz 8088 那样很慢的系统中，将 IO_COUNT 设置为 1 的最短延迟常常会产生一个大约 16 微秒的延迟，其中包括子程序的近调用。对于快速 CPU 和时钟速度来说，这些就显得微不足道了。

在本书提供的其他程序中，我用到了一个称为 IODELAY 的宏，它就是利用两个短跳转指令来插入一个短延迟。使用这个宏也可以改为调用 IO_DELAY。

```
;--------------------------------------------------------------------------
;    I/O DELAY
;        生成延迟，以保证在一个快速系统上对同一个设备
;        的连续的I/O操作不会过快  第一次调用 iodelay
;        时，会出现一个 5 ms 的延迟，来校准延迟循环周
;        期 。
;
;        调用:    无
;
;        返回:    1 到 16 μs 的延迟
;
;        Regs used:        none (flags altered)

iodelay   proc    near
          push    cx
          mov     cx, cs:[io_count]
          jcxz    io_init              ; 仅当cx=0（第一次）
io_delay_loop:
          loop    io_delay_loop        ; 在一台386上，每循环有 13 个周期
```

```
        pop     cx
        ret

; 第一次运行，所以要确定初始化值，并保存

io_init:
        push    ax
        push    bx
        push    dx
        push    es

        mov     ax, cs
        mov     es, ax
        mov     ax, 8300h           ; 设置等待时间间隔
        mov     cx, 0
        mov     dx, 5000            ; 延迟 5 ms 直到设置了等待
        mov     bx, offset io_flag  ; 标志位
        int     15h                 ; 开始 5 ms 的延迟
        jc      io_failed_15        ; 没有执行或者正在执行

; 启动了 RTC 时钟，如果5 ms 期满，就会设置等待标志的第7位。
; 同时，软件循环进行计数，在慢速系统上计数慢，而对于快速系
; 统上计数也快。在一台 386 的CPU上，这个循环是30个周期。
; 其他的 CPU 的情况会有点不同

        dec     cx                  ; cx = FFFFh

io_delay_loop2:
        test    cs:[io_flag], 80h   ; 5 ms 后设置标志
        jnz     io_5ms_done         ; 如果到了 5 ms 则跳转
        jmp     short $+2
        loop    io_delay_loop2

        mov     ax, 100             ; 如果CPU速度大于400 MHz ，则到这里
        jmp     io_set              ; 这样，使用的就一定是大数
```

```
; cx = 从 FFFF 开始递减计数

io_5ms_done:
        mov     ax, 0FFFFh
        sub     ax, cx                  ; 获得循环次数
        mov     bx, 1500                ; 调整因子
        xor     dx, dx                  ; dx 变成零，供除法用
        div     bx                      ; ax= dx:ax/cx
        cmp     ax, 0
        je      io_1_delay              ; 设置至少1次延迟
        jmp     io_set                  ; ax = 延迟值

io_failed_15:
        or      ah, ah                  ; 支持中断15吗？中断忙吗？
        jz      io_exit                 ; 如果忙，则跳转，以后在试验
                                        ; 只有慢速机器才不支持中断 15h
io_1_delay:
        mov     ax, 1                   ; 最小延迟周期
io_set:
        mov     cs:[io_count], ax       ; 装入延迟值
io_exit:
        pop     es
        pop     dx
        pop     bx
        pop     ax
        popf
        pop     cx
        ret
io_delay endp

io_count dw     0                       ; 0 = 非初始化计数
io_flag  db     0                       ;在 5ms 等待结尾 位7=1
```

与 I/O 有关的 CPU 模式

从 386 到 Pentium Pro，处理器提供了三种操作模式，它们分别是实模式、保护模式和

虚 386 模式。实模式在概念上类似于一个 8088 处理器。所有的资源，包括 I/O 端口，都可以被软件用户所访问，访问 I/O 地址不受限制。

其他两种处理器模式，保护模式和虚 86 模式，会限制对 I/O 的访问。这意味着像 Windows 或 OS/2 那样高级的操作系统可以控制谁被允许读/写 I/O 端口。在 DOS 环境下，内存管理器运行在保护模式下，以便在运行于虚 8086 模式下的 DOS 的顶部提供一系列的内存服务。在某些情况下，内存管理器需要对访问进行控制，或者“陷入”指定的 I/O 地址。

例如，对 DMA（Direct Memory Access，直接内存访问）控制器硬件的访问就需要进行控制，这些硬件驻留在每个 PC 机中。DMA 使一部分内存和 I/O 总线不需要借助 CPU 就可以进行数据传送，并且在实模式下能良好地运行。软件可以通过 I/O 端口向 DMA 控制器通信，并指示传送数据的来源、走向和大小。

在 DOS 环境下，内存管理器的任务之一就是为 EMS 服务和高载 TSR 而移动扩展内存。在处理器翻译了地址之后，软件就可以监视经过内存管理器映射过来的扩展内存。尽管 DMA 硬件操作物理内存，但是控制 DMA 的软件却只理解虚拟内存。如果软件向内存的一个虚拟地址传送一些字节，DMA 就会将信息传送到一个内存中不存在的“孔”里。

为了解决这个问题，内存管理器会“陷入”DMA 控制器的 I/O，如果试图访问 DMA 的 I/O 端口，内存管理器就会在 I/O 操作之前接管控制。这样管理器就可以监视程序所涉及的 DMA 控制器的地址。如果这些地址指向一个虚拟的区域，内存管理器就会更改送到 DMA 控制器的值，以便使内存访问指向存在于物理映射中的实内存。

操作系统或者其他一些低级的资源管理程序提供了许多陷入 I/O 操作的方法。陷入操作允许多任务软件实现程序间的保护，它还提供了一种方法来模拟那些甚至并不存在的各种硬件。

通过 C 和 C++访问硬件

到目前为止，我们给出了一些具体的二进制指令和一些用来访问 PC 机的 I/O 和中断的汇编指令的例子。许多高级语言也提供了对硬件 I/O 和中断的直接访问。尽管其中有些是通过在线汇编实现的，实际上它们也提供一些高级指令。

寄存器

许多中断要求在执行之前将值传递给寄存器，同时可能还会将值返回到某些寄存器。如果用 C 编写的代码访问 CPU 寄存器，那么就可以使用 DOS.H 提供的结构 REGS，它定义了一系列的寄存器。例如，可以在调用 int86x 功能来触发一个中断之前，将 70h 装入 DX：

```
regs.x.dx = 0x70;                    // 在汇编语言中：  mov  dx, 70h
```

大多数编译器在80386和更高级的CPU中支持所有的标准8位和16位寄存器，但是只有最新的版本才会支持32位寄存器和FS、GS寄存器。通常，不支持IP（指令指针）寄存器。表3-5列出了支持使用的标准伪变量。

表3-5 C中的CPU伪变量

汇　编	变　量*	类　型	REGS
ah	_AH	unsigned char	regs.h.ah
al	_AL	unsigned char	regs.h.al
ax	_AX	unsigned int	regs.x.ax
bh	_BH	unsigned char	regs.h.bh
bl	_BL	unsigned char	regs.h.bl
bx	_BX	unsigned int	regs.x.bx
ch	_CH	unsigned char	regs.h.ch
cl	_CL	unsigned char	regs.h.cl
cx	_CX	unsigned int	regs.x.cx
dh	_DH	unsigned char	regs.h.dh
dl	_DL	unsigned char	regs.h.dl
dx	_DX	unsigned int	regs.x dx
bp	_BP	unsigned int	regs.x.bp
di	_DI	unsigned int	regs.x.di
si	_SI	unsigned int	regs.x.si
sp	_SP	unsigned int	regs.x.sp
cs	_CS	unsigned int	regs.x.cs
ds	_DS	unsigned int	regs.x.ds
es	_ES	unsigned int	regs.x.es
ss	_SS	unsigned int	regs.x.ss
(标志位)	_FLAGS	unsigned int	regs.x.flags

*并非所有的编译器都支持（Borland包括了这个变量）。

中　断

大多数带有编译器的库提供了调用PC系统中断程序的简易方法。应用程序调用一个中断功能并将一系列CPU寄存器传递给中断，同时从中断返回一系列寄存器。

注意这些C函数在PC环境下有诸多限制。它不支持32位寄存器，也不支持SS、FS

或 GS 段寄存器。如果中断需要这些寄存器，那么你就必须使用汇编语言。我还没有发现任意一个标准函数用到这些寄存器。

表 3-6 列出了 Borland 和 Microsoft 用到的通用函数，其他语言的供应商也有类似的函数变量。Borland 和 Microsoft 还提供了其他一些函数来访问 DOS 21h 中断。另外，一些供应商还提供了一些独特的函数来解决一些特殊问题。

表 3-6 通用的中断函数

函 数	描 述
int86	触发一个中断。要求用户指明一系列的输入输出寄存器。中断结束后该函数返回 AX 的值 int int86(int interupt_#, &inregs, &outregs);
int86x	触发一个中断。要求用户指明一系列的输入输出寄存器以及 DS 和 ES 段寄存器。中断结束后该函数返回 AX 的值 int int86x(int interupt_#, &inregs, &outregs, &segregs);

这些通用函数使用起来比较容易，但是并不总像汇编代码那样有效和迅速。即使有时中断并不使用某些或者大多数寄存器，大多数函数在进入中断之前仍然会先更新所有的寄存器。同样地，它们通常会保存所有的寄存器，即使中断只是改变其中少数几个寄存器。

对于需要一个远指针或返回一个远指针的中断函数而言，有必要在 C 的远指针和 CPU 所需要的 段：偏移量 之间进行转换。表 3-7 给出了三个实现这种转换的宏。

表 3-7 指针转换

函 数	描 述
FP_OFF	获取一个 C 远指针的偏移部分，语法如下： Unsigned FP_OFF(void far *p);
FP_SEG	获取一个 C 远指针的段部分，语法如下： Unsigned FP_SEG(void far *p);
MK_FP	将一个段及其偏移量转换为一个 C 远指针，只有 Borland 提供了这个函数，也不同于 Microsoft C 7.0 中的这个函数。语法如下： void far *MK_FP(unsigned segment, unsigned offset);

中断控制

某些例子需要禁止或开放中断，为此提供了两个函数。在 Borland 和 Microsoft 编译器中，他们是_disable 和_enable 函数。_disable 实现汇编指令中 CLI 的功能，即对处理器的中断指示器清零。_enable 实现汇编指令中 STI 的功能，即设置处理器的中断指示器。

I/O 端口

四个基本库函数提供了对 I/O 设备的直接访问，分字节传送和字传送两种，但是要记住，大多数端口按字节访问。表 3-8 和 3-9 列出了适用于 Borland 和 Microsoft 的四个可行的组合。

表 3-8 Borland I/O 端口函数

函 数	描 述
inportb	从指定的端口输入一个字节，语法如下： unsigned char inportb(int portid);
inport	从指定的端口输入一个字，语法如下： unsigned char inport(int portid);
outportb	向指定的端口输出一个字节，语法如下： void outportb(int portid,unsigned char value);
outport	向指定的端口输出一个字，语法如下： void outport(int portid,unsigned char value);

表 3-9 Microsoft I/O 端口函数

函 数	描 述
inp	从指定的端口输入一个字节，语法如下： int _inp(int portid);
inpw	从指定的端口输入一个字，语法如下： unsigned char _inpw(int portid);
outp	向指定的端口输出一个字节，语法如下： int _outp(unsigned portid,int value);
outpw	向指定的端口输出一个字，语法如下： unsigned _outpw(unsigned portid, unsigned char value);

C 语言及汇编代码例

为了比较汇编语言 Borland C 和 Microsoft C，我编写了一个子程序来让扬声器发声两秒钟。这个程序展示了输出、输入和中断操作。代码例中的第一个例子展示了怎样用汇编语言编写这个程序代码。源代码文件是 BEEPA.ASM。对于 Borland C 和 Microsoft C，参看代码例 3-3，源代码文件是 BEEPB.C。对于 MicrosOft C，参看代码例 3-4，源代码文件是 BBEPM.C。

代码例 3-2 BEEPA 的汇编代码

```
;-------------------------------------------------------------------------
;                              Two Second Beep
;-------------------------------------------------------------------------
include   undocpc.inc
cseg      segment   para public
          assume    cs:cseg, ds:cseg
          org       100h              ; 作为 CMOS 程序创建
; 首先将时钟 2 设置为 5KHz
beepa     proc      near
          mov       al, 0B6h          ; 时钟 2 模式设置
          out       43h, al           ; 设置模式
          mov       ax, 0EEh          ; 对于 5KHz: 1190/5 = 0EEh
          IODELAY
          out       42h, al           ; 设置计数器的 LSB
          mov       al, ah
          IODELAY
          out       42h, al           ; 设置计数器的 MSB
; 激活扬声器
          in        al, 61h
          or        al, 3             ; 将扬声器连到时钟 2
          IODELAY
          out       61h, al
; 等待 2 秒
          mov       ah, 86h           ; 系统等待功能
          mov       cx, 1Eh           ; cx:dx = 1E8480h,等待 2 秒
          mov       dx, 8480h         ;    (1E8480h = 2,000,000  μS)
          int       15h               ; 等待 2 秒
; 关闭扬声器
          in        al, 61h
          and       al, 0FCh          ; 关闭扬声器
          IODELAY
          out       61h, al
; 退出并返回到 DOS 操作系统
```

```
        mov     ah, 4Ch
        int     21h
beepa   endp
cseg    ends
        end
```

代码例 3-3 BEEPA 的 Borland C 代码

```
/*--------------------------------------------------------------------------
                        Two Second Beep - Borland C
  --------------------------------------------------------------------------*/
#include "dos.h"
int main(void)
{
        static union REGS ourregs;
        // 首先将时钟 2 设置成 5KHz 信号
        outportb(0x43, 0xB6);          // 时钟 2 模式设置
                                       // 对于 5kHz: 1190/5 = 0xEE
        outportb(0x42, 0xEE);          // 设置计数器的 LSB
        outportb(0x42, 0);             // 设置计数器的 MSB
        // 仅设置端口 0x61 的第 0 和
        // 1 位，来激活扬声器，并与
        // 时钟相连。其他位保持不变。
        outportb(0x61, (inportb(0x61) | 0x03));
        // 使用中断 15 功能 86 等待 2 秒
        ourregs.h.ah = 0x86;
        ourregs.x.cx = 0x001E;         //  0x1E8480 = 2,000,000 μs
        ourregs.x.dx = 0x8480;
        int86(0x15, &ourregs, &ourregs);  //  触发中断 15
        // 关闭扬声器
        outportb(0x61, (inportb(0x61) & 0xFC));
        return 0;
}
```

代码例 3-4 BEEPA 的 Microsoft C 代码

```
/*--------------------------------------------------------------------------------
                        Two Second Beep - Microsoft C
--------------------------------------------------------------------------------*/
#include "dos.h"
int main(void)
{
        static union _REGS ourregs;
        // 先将时钟 2 设置定为 5kHz 的时钟信号
        _outp(0x43, 0xB6);          // 时钟 2 模式设置
                                    // 对于 5kHz: 1190/5 = 0xEE
        _outp(0x42, 0xEE);          // 设置计数器的 LSB
        _outp(0x42, 0);             // 设置计数器的 MSB。
        // 仅设置端口 0x61 的第 0 和
        //  1 位，来激活扬声器，
        //  并与时钟相连，其他位保持不变。
        _outp(0x61, (_inp(0x61) | 0x03));
        // 使用中断 15 功能 86 等待 2 秒
        ourregs.h.ah = 0x86;
        ourregs.x.cx = 0x1E;        //  0x1E8480 = 2,000,000 μs
        ourregs.x.dx = 0x8480;
        _int86(0x15, &ourregs, &ourregs);  //  触发中断 15
        // 关闭扬声器
        _outp(0x61, (_inp(0x61) & 0xFC));
}
```

CPU 系列归纳

表 3-10 列出了今天 PC 机所使用的主要的处理器家族。由于 80188/80186 的设计对 PC 应用不兼容，所以没有列出它们。80188/80186 CPU 成功地进入了嵌入式控制器市场，当时，许多集成外部芯片在价格和大小上都优于 8088。新型的 386 和 486 变型也进入了嵌入控制器市场，但在这里也没有把它们包括进来。

表 3-10 Intel 处理器家族及其特征

CPU	Intel 发布日期	内部总线大小	外部总线大小	速 度	最大寻址能力
8088	1978.6	16	8	5MHz	1MB
80286	1982.2	16	16	6MHz	16MB
80386	1985.10	32	32	16MHz	4GB
80486	1989.8	32	32	20MHz	4GB
Pentium	1993.5	64	64	66MHz	4GB
Pentium Pro	1995.4	64	64	150MHz	4GB
P55C(MMX)	1997 年春*	64	64	200MHz	4GB

* Pentium，带 MMX，1996 年春天开始 Beta 测试，发布日期 1997 年春。

接下来的一部分指出了最常用的处理器，供应商，以及一个系列内部不同型号间的差别。有必要指出，从 386 开始每个系列有许多种处理器。为了区别这些不同的种类，供应商采用了一种令人迷惑的后缀形式，比如 DX、DX2、DX4、SX、SL、SLC、SLC2、DLC 等等。三种常见的 Intel 类型是 386SX、486SX、和 486SL。它们有相似的后缀，然而 386SL 的外部数据总线只有 16 位，而 486SX 和 486SL 使用 32 位的总线。386SX 和 486SX 没有内置的浮点数学协处理器，但是 486SL 含有协处理器。有意义的两个字母是 D 和 L。“D”表示是一个完全的 32 位外部总线。一些没有“D”后缀的类型可能有，也可能没有 32 位的外部总线。“L”后缀表示 CPU 提供某种类型的低功率模式。CPU 类型号后面的数字“2”或“4”表示 CPU 使用时钟双倍或三倍技术，以实现更快的内部处理。其他所有的后缀都毫无任何意义，只是用来标识它是一种类型的变型。所列出的处理器列表按照近似的发布日期排列。每个系列中对应于同一个供应商的所有类型都放在一起。

类型	供应商	家族系列
8088	Intel 和其他	8086

IBM 为其 PC 系列选用了 8088，目前这种类型早已经过时了。8088 提供了一个 16 位的内部 CPU，带有 8 位的外部数据，可以选择使用浮点处理器，8087。

类型	供应商	家族系列
8086	Intel 和其他	8086

从软件的视角来看，它和 8088 相同，但是它提供了一个 16 位的外部总线，以及一个更大的预取队列。这样做的结果是 8086 的运行速度要比 8088 快百分之 10 到 20。只有几个 PC 供应商使用 8086，例如 AT&T 就使用 8086。这种芯片在嵌入式系统市场更为成功。

类型	供应商	家族系列
V20	NEC	8086

V20 可以一个引脚对应一个引脚地替代 8088，而同时提高了 10%~15%的执行速度。V20 还提供了一些指令和功能来处理老式的 8080 代码。许多希望提高速度的 8088 用户都急于用 V20 替代他们的 8088。除了无用的 POP CS 指令（所有后来的 CPU 系列都放弃了这个指令）外，V20 可以执行 8088 上的每条指令。V20 不能使用 8088 上几条未公开的指令。

类型	供应商	家族系列
80286	Intel 和其他	286

80286 是第一个和 8088 兼容，但是又极大地提高了系统的执行速度、性能和内存寻址能力的 CPU。速度从 8088 上的 5MHz 提高到 8MHz。另外，寻址范围从 8088 上的 1MB 扩展到了 16MB。

由于完成 286 设计的时间和 PC 引进的时间大致相同，所以大多数的 286 的实模式 DOS 软件不能执行在保护模式下的新特性。IBM 和 Intel 这时是背道而驰的。后来，当 OS/2 和 Windows 的第一个版本出现在市场上时，大多数用户觉得 286 能力太弱，而不能应用于这些操作系统环境。这一点推动了对更快 CPU 的需求。

当需要额外的数学运算时，可使用供选的 80287 浮点协处理器。Intel 生产了一种高性能的数学协处理器，80287XLT。

类型	供应商	家族系列
80386DX	Intel 和 AMD	386

借助 386，Intel 极大地提高了 CPU 的功率、可使用的性能以及寻址空间。这个第一次采用 32 位的处理器与 80286 的运行速度相比，提高了四倍。当然推出该类型之时，几乎所有的软件产品仍然是为 8088 编写的。这一点会稍稍降低最大可能的收益。

最大的突破是新引入的虚 86 模式。这一点允许供应商在软件中加入各种新功能。这些特性包括软件的扩展内存管理、高载 TSR 和设备驱动程序、Windows 和 OS/2 之类的多任务操作系统、多个 DOS 窗口等等。80386 成为 90 年代早期大多数用户最低的常见配置。

另一个主要的进步包括，通过减少大多数指令的时钟数来提高运算速度。它还扩大内置调试属性的范围，将内存的寻址能力提高到 4GB。

许多公司试图仿制 80386，但是都在 Intel 的法律大战面前败下阵来。唯一的仿制品是 AMD 的 Am386。大多数 1993 年生产的 AMD 类型，都仿制 Intel 的屏蔽类型和微指令。奇怪的是，AMD 更快，达 40MHz。IBM 有一段时间也生产过一种准 386 类型，但那是很晚的事了。

在 286 的指令集上又加入许多指令，同时加入了大量新的 32 位寻址模式。其中大多数都已经被详细地加以说明。它放弃了未说明的 286 的 LOADALL 指令，同时加入一种全新

的带有不同操作码的386 LOADALL。参看“内幕指令”，来获得有关LOADALL的更多信息。

80386DX带有一个供选的数学协处理器，80387DX。起初由于推迟了这种类型的发布，一些供应商就选用便宜的16位80287数学协处理器，这也是当时唯一可能的解决方法。

类型	供应商	家族系列
80386SX	Intel	386

内部功能完全等同于80386DX，但是使用16位外部数据总线和24位地址总线。这意味着CPU需要两倍周期来同外界传送数据和代码。另外，386SX最大寻址能力为16MB。

由于与386DX相比，它的价格更便宜，所以很多计算机都使用386SX。16位的外部总线也简化了主板的设计，有助于进一步降低成本。

80386SX的浮点处理器是80387SX。387SX采用16位外部总线，从软件的视角来看，它与80387DX相同。

类型	供应商	家族系列
386SL	Intel	386

膝上型电脑使用这种Intel的低功耗386类型。386SL与80386完全兼容，低功率模式能够挂起和恢复运行，并且集成了一个内存与高速缓存控制器。

内部功能与80386DX完全一致，但是只提供16位外部数据总线和24位的地址总线。这意味着外部数据和代码的传送需要两倍于同速386DX的周期。另外，386SL最大只能寻址16MB。与大多数其他Intel家族系列的CPU不同，该类型只用于静态逻辑。在长时间的休眠期间通过停止CPU时钟来节省功耗，并且有助于用户控制和恢复运行。

配套的浮点处理器是80287SL，它可以和387SL一起低功耗运行。然而，387使用16位的外部数据总线。

类型	供应商	家族系列
386SXL/DXL	AMD	386

这些类型类似于标准的386DX和386SX，但是提供了低功耗模式下的静态运行。与Intel的386SL不同的是，386SXL/DXL不提供挂起和恢复特性。

类型	供应商	家族系列
386SXLV	AMD	386

该类型类似于Intel的80386XL，提供低功率模式以及挂起和恢复特性。挂起和恢复运行的操作不同于Intel类型，它需要单独的代码。

类型	供应商	家族系列
386DXLV	AMD	386

这是唯一一种提供了 32 位外部地址总线、低功率模式、能挂起和恢复运行的 386 类型。AMD386DXLV 提供了膝上型电脑的快速解决方案。

类型	供应商	家族系列
386SLC	IBM	386

我称这个 386SLC 是一个伪 386。仔细看看就会发现该类型实际是个 486，就好像一匹“披着羊皮的狼”。它拥有一个 486SL 所有的东西，只是对 PC 不兼容罢了。该 386SLC 有一个完全的 8KB 内部高速缓存，支持所有的 80486SX 指令，并且该芯片采用与 80486 相同的时钟速度来改进指令运行速度。386SLC 还有一些额外指令来实现类似于 80486SL 的节能运行。

这一点似乎好得难以令人置信！从总体而言确实如此，然而 IBM 的这个类型也有一个局限。它采用 16 位外部数据总线和 24 位地址总线，这一点与 Intel 的 386SX 相似。这一点不可以与 80486SX 相混淆，80486SX 采用的是真正的 32 位外部数据和地址总线。

那为什么 IBM 不称之为 486 呢？有两个原因，但是我认为这完全是一种手段，首先，他们有一个更好的类型，称为 486SLC2，它带有一个 16KB 的内存高速缓存。标号为 386SLC 有助于区别于这两种类型。其次，我认为更有可能的原因，就是为了省钱。我猜测依据 IBM 同 Intel 达成的版税协议，IBM 将为 386 类型付出比 486 类型更低的版税。

尽管有点不太可能，486SLC 仍然可以算是个带有半个 16KB 高速缓存的 486SLC。可能是模板通过生产测试的方式，决定了生产出来的 CPU 是被称为 486 还是 386。这将提高总产量，同时还可以省钱。我没能说服一些朋友拿掉芯片上的盖子，来看看 IBM 是否使用相同的模板。当然，这样会损坏芯片，所以我能理解他们为什么不情愿这么做。

IMB 的 80386SLC 有什么缺憾呢？所有的 486 CPU 提供了对齐检查性能，在数据没有对齐时会生成一个中断。我从未看到有人使用过这个性能，很可能是由于对齐检查会触发中断 11h，这一点会与系统的 BIOS 的设备配置功能发生直接冲突。对齐检查由 IFLAGS 的第 18 位控制。可以在 486 类型上修改这一位，但是不能在 386 上修改。多年来，许多开发人员就采用这一技巧来识别出 486 类型。

386SLC 还支持其他一些有趣的指令，包括 LOADALL、RDMSR 以及 WRMSR。在下一部分，内幕指令中将对这些指令作出解释。公平地讲，IBM 公开了这些指令，但是很少有程序员知道这些信息。

类型	供应商	家族系列
80486DX	Intel	486

80486DX 是 Intel 处理器家族系列中的又一个巨大的进步。往回看，与 80386 的第一次发布相比，80486 似乎更全面。80486 可以看成带有重大改进的 80386。486DX 包括一个集成的 8KB 高速缓存，和一个内置于 CPU 的完整的浮点处理器。当然，在大多数的 386 系统中可以外带这两条。另外，80486 的硬件和微指令相对于前面的设计有进一步的改进，以提高运行速度。将所有这些置于一个芯片内就是一个巨大的成就。该类型兼容所有以前的处理器。

80486 也有许多变型，它们提供一系列的速度和低功率形式，是 Intel 竞争对手的早期针对的主要目标。

最近的变型采用了双倍时钟或三倍时钟技术，使内部 CPU 时钟比外部总线时钟快得多。Intel 在其后加入 DX2、DX4 后缀。（DX4 指的是三倍时钟而不是四倍！）这进一步提高了运行速度而不用进行复杂的高速主板设计。许多情况下，时钟倍频的 CPU 可以安装在较老的 486 系统上，但是不会提供更快的运行速度。

新加入的指令包括 BSWAP、CMPXCHG（它可以交换 A 步和 B 步之间的操作码）、INVD、INVLPG、WBINVD 以及 XADD。当然,我也详细地阐明了所有上述的这些指令，而删除了未公开的 386 LOADALL 指令。

类型	供应商	家族系列
80486SX	Intel	486

除了没有内置的浮点处理器（Floating-point Processor,FPU）之外，80486SX 与 80486DX 完全一样。实际上，早期的 SX 类型就被误认为是不带 FPU 的 DX 类型。这个特点有助于在提高增加产量的同时降低价格。与 386SX 不同的，486SX 支持 32 位的内部和外部总线。

采用下面方法可以查出一个 80486 是 DX 类型还是 SX 类型：一个 DX 类型的控制寄存器 CR0 的第 4 位是不能被修改的，表明这个 FPU 是内置式的。而在 SX 类型上，这一位可以被修改，即使它带有一个外部的 FPU。

相配套的数学协处理器是 80487SX FPU。从软件的视角来看,它等同于 80486DX 的 FPU。

类型	供应产	家族系列
80486SL	Intel	486

与 486SX 不同的，80486SL 提供了 486DX 的所有的功能，并且带有浮点处理器。另外，486SL 能够为低功率设计、时钟控制、总线以及内置式内存控制器提供电源管理功能。它特别适合于与外部配套芯片 82360 一起使用。这两项构成了一个完整的 AT 系统的基本组成。

类型	供应商	家族系列
486SLC2	IBM	486

这是 486SX 与 386SX 的混合体。从编程的角度来看，这个类型除了拥有 80486SX 的全部指令外，还附带了一些额外的指令。这意味着它支持 80486 指令集。当然，它没有内置的浮点数学协处理器。IBM 的 486 也为膝上型电脑设计提供了功率管理功能。与 386SX 一样，486SLC2 有 16 位外部数据总线、24 位地址总线。与其他 386/486 CPU 不同的是，IBM 的 486SLC 带有一个更大的 16KB 的内部高速缓存。该芯片支持时钟双倍周期，因此内部 CPU 运行的速度是外部总线的两倍。基于上述改进，IBM 声称，与具有相同外部总线速度的 386SX 相比，该类型的运行速度提高了 3 倍。

486SXC3 是作为一种更新类型在 1993 年被推出来的。这种变型使用 3 倍时钟技术，因此 CPU 速度是外部总线的 3 倍。486SL3 可以提供 75MHz 到 99MHz 的内部速度。

其他为 486SLC 所支持的有趣的方面包括：LOADALL、RDMSR 和 WRMSR。每一个指令将在下一节内幕指令中加以解释。正如我在 IBM 386SLC 中讲到的那样，IBM 公布了这些指令。

类型	供应商	家族系列
486SLC	Cyrix	486

这是第一个没有得到 Intel 公司授权的 486 复制芯片。在同 Intel 打了几年的官司之后，Cyrix 赢得了这场争论。

经过测试，我发现 Cyrix 在仿制 486 CPU 方面做得相当出色。

对于相同的时钟速度，486SLC 显著地提高了大多数指令的执行速度，但是却只使用了 1KB 的内置高速缓存。Cyrix 也为膝上型电脑提供了低功率选项。同 486SX 一样，它不带内置的浮点处理器。但与 486SX 不同的是，486SLC 只含有 16 位的数据总线以及 24 位的地址总线。在这一点上它更接于 386SX。由于它同 Intel 的类型存在太多硬件上的差别，所以很难说清楚谁好谁坏。

Cyrix 类型指令唯一不同之处在于，它不包含尚未公开的 UMOV 指令，这个指令主要用在模拟器中。这一点并不重要，主要是因为 Intel 从 Pentium 开始就删除了 UMOV。

Cyrix 从 1995 年年底开始停止生产这种 486 类型，以便集中生产更高速度的类型。

类型	供应商	家族系列
486DLC	Cyrix	486

Cyrix 486DLC 完全等价于 Intel 的 486SX。它提供完整的 32 位数据和地址总线。另外，它还修改了许多指令以减少运行周期和提高运行速度。与 Intel 的 8KB 高速缓存相比，486DLC 只有一个 2KB 的内部高速缓存。同它的小弟弟 486SLC 一样，486DLC 有一系列的低功率模式，以适应膝上型电脑的设计需要。

Cyrix 类型指令唯一的特别之处在于，它不含有未公开的 UMOV 指令，这个指令主要用在模拟器中。这一点并不重要，主要是因为 Intel 从 Pentium 开始就删除了 UMOV

指令。

Cyrix 从 1995 年年底开始停止生产这种 486 类型。

类型	供应商	家族系列
486SLC	Texas 仪器公司	486

这个类型与 Cyrix 对应的类型相同。Cyrix 开发了这种类型，TI 是 Cyrix 的制造商之一。TI 和 Intel 有广泛的互用协议。这样做是为了避免同 Intel 的法律纠纷。TI 已不再生产这种类型。

类型	供应商	家族系列
486DLC	Texas 仪器公司	486

这个类型与 Cyrix 对应的类型相同，详见上条说明。

类型	供应商	家族系列
486SXLV	AMD	486

这种类型类似于 Intel 的 486SX，但是，它的功耗更少。AMD 的 486SXLV 包括了额外的低功率模式，支持挂起和恢复操作。同 486SX 一样，486SXLV 提供了完全的 32 位地址总线和 8K 的内部高速缓存。它没有提供内部数学协处理器。

类型	供应商	家族系列
486DXLV	AMD	486

该类型类似于 Intel 的 486DX，但是功耗较少。AMD 的 486DXLV 包括了额外的低功率模式，支持挂起和恢复操作。486DXLV 提供完全的 32 位地址总线，8K 内部高速缓存和一个内置的数学协处理器。新式的 DX2 与 DX4 系列采用时钟双倍和三倍技术，以提高运行速度。AMD 可以生产出 120MHz 的 486 系列，它是 486 系列中最快的类型。DX2 和 DX4 类型支持 CPUID。最新的版本改进了对回写高速缓存的设计。

类型	供应商	家族系列
Am5x86	AMD	486

这个 5x86 实际上是一个快速 486 芯片（133MHz，DX4），由于其运行速度接近于 75MHz 的 Pentium，所以 AMD 将之称为“586”。它没有任何 Pentium 所特有的指令，也没有采用额外的逻辑手段来提高运行速度。因为它实际上是一个 486，所以下一章中的 CPU 标识程序将它标识为一个快速 486 类型。

类型	供应商	家族系列
Pentium	Intel	586

从软件的角度来看，Pentium 酷似 80486，只是加入了一些额外的指令而已。从硬件的角度来看，其设计是一种全新思维的新方法，这种全新思维的新方法带来了运行速度的极

大提高。Pentium 也可以运行在系统管理模式下，特别是可用在那些要求省功率和可以保存和恢复的设计中。

从硬件的角度来看，Pentium 作了一系列很大的改进。它包括了一个 8KB 的数据高速缓存和一个独立的 8KB 指令高速缓存，一个 64 位的内部和外部数据总线，一个 256 位的内部指令总线，带回写的双向 64 字节预取队列。双重指令流可以实现同时执行两条指令，一个分支表用来预测最近的 256 个分支。另外，内置了一个改进型的浮点协处理器，从而将浮点运算的速度提高了 10 倍。

新的并且详细说明的 Pentium 指令包括 CMPXCHG8B、CPUID、MOV 一直到 CR4。其他未说明的或论述不详的指令包括 RDMSR、RDTSC、RCM 和 WRMSR，在下一章内幕指令中会详细阐释这些指令。Pentium 从测试寄存器指令中删除了所有的 MOV 指令，还删除了所有的未公开指令 UMOV，Intel 386 和 486 类型提供了这个未公开指令。

Pentium 还做了一些新改进，但 Intel 最初并未向程序员说明这些改进。Pentium 用户手册中有许多“保留”的功能和领域，要了解它们可以参考一个秘密文档，附录 H。1995 年，Intel 公开表示将公开所有这些信息，并且在最近的 Pentium 手册中会公开一些以前隐藏的信息。不幸的是，最新的手册（购自 Intel6/96）仍然隐藏了一些信息。它仍然有一个不可得到的附录 H，来收集许多高级性能的信息。极其荒唐的是，大部分的“隐藏信息”看起来又像是已经包含在了标准的 Pentium Pro 手册中。真是活见鬼！

可以通过控制寄存器 4（CR4）中的两位来控制这些新的“未公开”性能。保护模式下的虚中断（PVI）位有助于减少中断开销。虚 8086 模式扩展（VME）能够消除对 CLI 和 STI 中断禁止和开放指令的陷入，编程时经常要使用这两个指令。

定义了两个新的状态标志位来处理这些新特性。在 EFLAGS 寄存器中，虚中断挂起标志（Virtual Interupt Pending,VIP）和虚中断标志（Virtual Interupt Flag,VIF）一起用来虚拟化中断标志。

类型	供应商	家族系列
5x86	Cyrix	586

这个 5x86 可以和 Intel 的 Pentium 匹敌，但是要求它使用更高一些的时钟频率才能同 Pentium 的运行速度相似。同 Pentium 一样，它提供了 64 位的内部数据通道。5x86 不处理 64 位外部数据，它只处理 32 位的外部数据。它包括一个 16KB 的内部高速缓存和一个 80 位的浮点单元。其他高级性能包括：能同时处理多条指令、带有分支预测逻辑以及功率管理性能。与 Pentium 不同的是，5x86 的引线插脚和总线设计和 486 兼容，它不支持大多数 Pentium 所特有的指令。这使它成为 486 系统升级的一种廉价的方案，它对低档的 Pentium 处理器很有竞争力。

IBM 和 SGS 也出售这种类型的机器。

类型	供应商	家族系列
Nx586	NexGen	586

Nx586 曾经是 Intel 的 Pentium 有力的竞争对手，它的运行速度最接近于后者。使用基准程序进行测试的结果通常指示，对于相同的处理器速度来说，Nx586 更快些。它提供完全的 64 位内部和外部总线、分支预测、能够同时执行多条指令。

它使用一个 32KB 的 L1 高速缓存（同 Pentium Pro 一样），在这一点上它优于 Pentium。它不带内置的浮点处理器，但是 NexGen 也可以提供 FPU。Nx586 的接脚和主板设计与任意的 Intel 类型兼容。

从一个软件的视角来看，Nx586 是一个快速 486。它没有提供 Pentium 指令，不支持 486 的对齐检查特性（即使用到也很少见）。大多数 CPU 标识程序利用对齐测试来确定一个 CPU 是 486 还是 386，这时所有的测试程序都显示，NxGen 是一个 386。下一章我们所讨论的 CPU 标识测试程序将把 Nx586 标识为一个带有 486 指令集的 586。NexGen 曾打算在一个修正版中使用 CPUID 指令，但据我所知，这一点从未正式公开过。

AMD 公司 1995 年年底收购了 NexGen 公司，所以 NexGen 系列不再生产。AMD 实际上想要的是 NexGen 的技术和 NexGen 正在开发中的 686 CPU。

Nx586 不同于 Cyrix5x86、AMD 的 5x86 和 5k86。

类型	供应商	家族系列
5k86	AMD	586

5k86 可以与 Intel 的 Pentium 匹敌。它支持 Pentium 的全部指令和功能。与 Pentium 不同的是，它使用 32 位的内部和外部数据通道。这一点保证了它和 486 兼容。它带有一个 16KB 的内部高速缓存，以及一个 80 位的浮点单元。它具备其他 Pentium 的高级性能，例如：分支预测逻辑和能够同时处理多条指令。

类型	供应商	家族系列
Pentium Pro	Intel	686

Pentium Pro 在目前的 86 家族中运行速度最高。从软件的视角来看，它几乎等同于 Pentium，只是增加了一个额外的指令而已。从硬件的角度来看，它作了进一步的改进来提高运行速度。Pentium Pro 带有一个内部的 16KB L1 高速缓存和一个内部的 256KB 或 512KB 的 L2 高速缓存。它还从硬件上支持多个处理器，同时包含一个带有长流水线的乱序执行装置。它对 32 位软件提供了最大支持。早期的 16 位软件并没有意识到，相对于相同时钟速度下的 Pentium，将会有巨大的改进。当软件（主要是 16 位的）的代码段寄存器作一些改动时就会出问题。每次修改代码段都要求清除指令流，这一点不利于乱序执行。

新的详细说明的 Pentium Pro 指令包括 CMOV、FCMOV、FCOMI 和 RDPMC。读/写型号专用寄存器已经作出了阐述，并在 Pentium 基础上作了进一步扩展，它还包括了新的内存管理特性。

类型	供应商	家族系列
6x86	Cyrix	686

6x86 可以与 Intel 的 Pentium 和 Pentium Pro 处理器匹敌。与 Cyrix 的 5x86 不同的是，它有完全的 64 位外部接口。其他性能同 5x86 类似，例如有一个 16KB 的内部高速缓存和一个 80 位的浮点单元。其他的高级特性还包括：能同时处理多条指令、分支预测逻辑、功率管理特性。对软件透明而有助于提高运行速度的独有特性包括：寄存器重命名、乱序执行和数据正向处理（forwarding）。该类型支持某些 Pentium 的特有指令，但不支持全部。

IBM 和 SGS 也出售这种类型的机器。

类型	供应商	家族系列
P55C(Pentium MMX)	Intel	586

MMX 是一些新指令，提供它们是为了提高数学运算的速度，特别是多媒体环境下的数学运算速度。MMX 代表的是多媒体扩展（Multi-Media eXtentions）还是矩阵数学扩展(Matrix Maths eXtentions)，这要看你是在和谁交谈了。目前 Intel 认为，MMX 不代表任意东西，更确切的讲应该用“MMX 技术”。在 Intel 网址（www.Intel.com）有这些指令和操作的详细说明。设计 MMX 指令，于是能够在单独的一次操作中处理 64 位数据，该 64 位数据可能是一组字节、字或双字，这样就可以提高运行速度。当 CPUID 属性位 23 是 1 时，表示 CPU 支持 MMX 指令。带 MMX 指令的处理器在 1997 年初发布。Intel 还在 1997 年年底发布了带 MMX 的 Pentium 版本。

P55C 类似于标准的 Pentium，但是它也做了很大的改进，包括加入了 MMX 的指令。这些改进还包括一个更大的 32KB L1 高速缓存（16KB 给数据，16KB 给代码），以及新的分支预测逻辑。该 Pentium 也可执行读运行监视计数器（Read Performance Monitoring Counter,RDPMC）指令，Pentium Pro 手册对这个指令做出了说明。

内幕指令

你可能想知道，我是如何定义一个内幕指令的，在我们的面前就好象是一团雾水。我所指的内幕指令是所有现在尚未在 Intel 出版的处理器文档中阐释的那些指令。大多数内幕指令不是很有用。一些内幕可以找到替代的公开指令来完成同样的功能。容易引起读者兴趣的是，每个 80x88 系列的 CPU 都有内幕指令。在一些情况下，内幕指令可以提供一种秘密的途径，来实现处理器的特殊性能。

尽管有时必须使用这些指令，但在大多数情况下，应避免使用它们，除非绝对必要。当指令不被后来的版本所支持时，做好保护性措施以防指令错误，不失为明智之举。至少，应该使用一个用户可设置的选项来避免使用任意的内幕指令。

为了查看你的 CPU 是否能正常执行我说的内幕指令，请运行 CPUTYPE。为了测试其他隐藏的 CPU 内幕指令，请运行 CPUUNDOC。这个 CPUUNDOC 程序测试所有的指令组合，在以前没有指出这些指令组合。它忽略了那些我已经标出了的内幕指令。在对许多 CPU

的测试中，我还没有发现其他的内幕指令。必须在实模式下运行这项测试（也就是说，没有运行内存管理程序、不是运行在 Windows95/NT 或 OS/2 的 DOS 界面下。CPUUNDOC 不会找出任意需要在保护模式下进行操作的指令，也不会找到那些需要专门寄存器值或专门的 mm-reg-r/m 区域的指令。

很少有汇编器支持内幕指令。我已经以宏的形式描述了许多内幕指令，可以在当前所有的汇编器中使用这些宏。软盘文件 UNDOCPC.INC 中包含这些宏，只需在你的汇编程序开头简单的包含文件 UNDOCPC.INC 就可以了：

```
include    undocpc.inc
```

内幕指令归纳

指令	描述	平台
AAD	在 BCD 除法之前调整 AX	所有，除了 V20
AAM	在 BCD 除法之后调整 AX	所有，除了 V20
CPUID	CPU 标识	某些 486+
IBTS	插入位串	Intel 80386 A step
ICEBP	内电路模拟断点（Break-Point）	大部分 386+
IN/OUT	用端口 22h/23h 实现 CPU 功能	大部分 Cyrix CPU
LOADALL	装入处理器所有的寄存器	某些 286-486
POP CS	从堆栈中装入 CS	8088/8086
RDMSR	读型号专用寄存器	某些 386+
RDTSC	读时间标志计数器	Pentium+
RSM	从系统管理模式下恢复	某些 386+
SETALC	按进位标志设置 AL 的值	所有，除了 V20
SHL	按计数值向左移位	80188+
SHL	左移 1 位	所有
SMI	系统管理中断入口	AMD386 DXLV/SXDLV
TEST	带立即数测试计数器	所有
UMOV	用户移动寄存器指令	某些 386/486
WRMSR	写型号专用寄存器	某些 386+
XBTS	取出位串	Intel 80386 级别 A

内幕指令详述

指令		描述	处理器
AAD		在 BCD 除法之前调整 AX	所有，除了 V20
D5h,	imm	aad	；将基数转化为二进制
D5h,	imm	aad	；将基数转化为二进制

对 AAD 指令的标准形式已经有了很清楚的阐释，所有的处理器都可以运行它。缺省形式下，它取出 AH 的值，乘以 10，在加上 AL，结果放在 AX 中。它也用于将两个 BCD 码转化为二进制码。例如，如果 AX 是 0907h（BCD 码 97），那么 AAD 将它转化为（9*10）+7=0061h。对于 BCD 码的除法和其他功能，这一点很有必要。

在公开的形式下，汇编器总是让立即数操作码等于字节 0Ah。其实，它也可以将其他进制（比如 16 进制或者 8 进制）的数转化为二进制数，而这一点并未公开。对指令的进制进行手动编码将可以实现任意进制的转化，而不是让汇编器将第二个操作码强行设置为 0Ah（10 进制）。在操作码字节 D5h 后面接立即数 N，CPU 就会执行下面操作：

```
AL = (AH*N) + AL
AH = 0
```

最常见的用法是将 16 进制和 8 进制转化为二进制，只用一条指令就可以了！使用 D5h，10h 可以将两个 16 进制数字转化二进制数，CPU 执行下面的操作：

```
AL = (AH*10h) + AL
AH = 0
```

使用 D5h，08h 可以将两个 8 进制数字转化为二进制数，CPU 执行下面的操作：

```
AL = (AH*8) + AL
AH = 0
```

商业上早就开始应用 AAD 的这种功能，它适用于除 V20 外的所有的 80x86 CPU。如果 N 值已经存放在某个寄存器中，那么在 8088 上 8 位乘法会显得略快，而在后来的 CPU 上显得略慢。

指令	描述	处理器
AAM	在 BCD 乘法之后调整 AX	所有，除了 V20

```
D4h,    imm         aam                 ; 将基数转化为二进制
```

对 AAM 指令的标准形式已有了很清楚的阐释，所有的处理器都支持它。缺省形式下，它将 AL 除以 10，结果放在 AH 中，余数放在 AL 中。将两个 BCD 码数字相乘后，标准的 AAM 指令将 AX 中的结果转化为 BCD 码数字。

在公开的形式下，汇编器总是让立即数操作码等于 0Ah。使用另外的立即数操作码可以实现除以其他数的特性。对指令进行手动编码可以做到这一点，而不是让汇编器将第二

个操作码强行设置为 0Ah（10 进制）。在操作码字节 D4h 后面接立即数 N，CPU 就会执行下面的操作：

```
AH = AL/N
AL = AL MOD N
```

为了分离一个 16 进制数，以便将其高半部分放在 AH 中，低半部分放在 AL 中，可以输入操作码 D4h，10h。CPU 执行下面的操作：

```
AH = AL/10h
AL = AL MOD 10h
```

如果在 AL 中放入值 7Fh，那么执行的结果就是 AX=070Fh。AAM 适用于除 NEC V20 之外的所有 80x86 CPU。

指令	描述	处理器
CPUID	在标识了处理器之后调整 AX	某些 486+，Pentium+

0Fh, A2h cpuid

CPUID 用来标识 CPU 的类型、供应商以及其他 CPU 所提供的特殊属性。虽然已有文档对这条指令做出了说明，它返回的信息会因供应商的不同而有所差别，老式的类型不支持这条指令。Intel 在 Pentium 处理器中引入了这条指令，也把它加入 Pentium 发布后的新的 486 系列中。较早的 AMD、Cyrix 和 NexGen 类型不带 CPUID 指令，但最新的 AMD 5k86 支持这条指令。

为了知道 CPU 是否支持 CPUID 指令，你可以试图修改 EFLAGS 的第 21 位。例如，如果你的 CPU 至少是 386，那么你可以用下面的代码来检测你的 CPU 是否支持 CPUID 指令：

```
cli                         ; 禁止中断
pushdf                      ; 标志位入栈
pop     eax                 ; 获得标志位
mov     ebx, eax            ; 保存，供后面使用
xor     eax, 200000h        ; 对第21位求反
push    eax
popdf                       ; 往CPU中装入修改后的标志位
pushdf                      ; 标志位入栈
```

```
pop     eax             ; 获取新标志位的值
push    ebx             ; 原来的标志位入栈
sti                     ; 开放中断
xor     eax, ebx        ; 检查位是否改变
jnz     CPUID_ok        ; 如果支持CPUID则跳转
jmp     no_CPUID        ; 没有CPUID指令
```

在使用 CPUID 指令之前，首先应在 EAX 中装入功能号，然后执行 CPUID 指令。结果会显示在一系列寄存器中，具体是哪些寄存器由功能号决定。

获取供应商信息串，EAX = 0

返回: EAX = CPUID 指令所支持的最大功能值，大部分 CPU 返回值 1，而 Pentium Pro 返回值 2。

EBX,ECX,EDX = ASCII 码形式的供应商信息串。ASCII 码字节开始于 EBX 的低位字节（0~7 位），结束于 EDX 的高位字节（31~24 位）。现有的供应商信息串包括:

"AuthenticAMD"	（现在仍在使用）
"Cyrix Inxtead"	（现在仍在使用）
"Genuin Intel"	（现在仍在使用）
"NexGen Driven"	（不清楚有没有用过）
"UMC UMC UMC"	（不再生产？）

获取版本和属性信息，EAX = 1

返回: EAX = 版本信息:

31~14 位	保留，以供将来使用（零）
13~12 位	处理器类型
	00 = 原始的 OEM 处理器
	01 = 向后引脚兼容系列（Intel OverDrive）
	10 = 双 Pentium 处理器配置
	11 = 保留，以供将来使用
11~8 位	家族系列
	3 = 386
	4 = 486
	5 = 586 和 Pentium
	6 = 686 和 PentiumPro
	7 = P7 类
7~4 位	型号
3~0	级别号（Stepping number）

EBX 和 ECX = 零（保留，以供将来使用）

EDX = 属性信息（在 Intel 的 Pentium Pro 手册中有以下说明）

31~24 位	保留，或者未予公开
23	CPU 支持 MMX
22~16	保留，或者未予公开
15	支持 XMOV 指令
14	支持机器全局功能检查
13	支持全局页面开放标志
12	支持内存类型范围寄存器
11~10	保留，或者未予公开
9	高级可编程中断控制器（APIC）驻留在芯片上，并开放了该控制器
8	支持 CMPXCH8B 指令
7	支持机器检查异常
6	支持 36 位物理寻址扩展
5	支持型号专用寄存器
4	支持时间标志计数器
3	支持 4MB 页面大小扩展
2	支持调试扩展
1	支持虚 8086 模式增强型
0	浮点单元驻留在 CPU 芯片内部

高速缓存和 TLB 信息，EAX = 2（现在仅适用于 Pentium Pro+）

该功能显示 CPU 中的高速缓存类型

返回： EAX = 高速缓存信息

31 位	0—如果寄存器内的信息有效
30~24 位	高速缓存类型
23~16 位	高速缓存类型
15~8 位	高速缓存类型
7~0 位	为获得所有高速缓存信息而触发 CPUID 的 2 号功能的次数。Pentium Pro 第一次发布时的返回值为 1

EBX，ECX，EDX = 高速缓存信息

31 位	0—如果寄存器内的信息有效
30~24 位	高速缓存类型
23~16 位	高速缓存类型
15~8 位	高速缓存类型
7~0 位	高速缓存类型

高速缓存类型： 0 = 空描述符

1 = 指令翻译备用缓冲区（Translate 备用 Buffer，TLB）：4KB 页面，4 路设置相关的高速缓存，64 个入口
2 = 指令翻译备用缓冲区（Translate 备用 Buffer，TLB）：4KB 页面，4 路设置相关的高速缓存，4 个入口
3 = 指令翻译备用缓冲区（Translate 备用 Buffer，TLB）：4KB 页面，4 路设置相关的高速缓存，64 个入口
4 = 指令翻译备用缓冲区（Translate 备用 Buffer，TLB）：4KB 页面，4 路设置相关的高速缓存，8 个入口
6 = 指令高速缓存：8KB 页面，4 路设置相关，32 字节线大小
0Ah= 数据高速缓存：8KB 页面，2 路设置相关，32 字节线大小
41h= 未定义的高速缓存：128KB 页面，4 路设置相关，32 字节线大小
42h= 未定义的高速缓存：256KB 页面，4 路设置相关，32 字节线大小
43h= 未定义的高速缓存：512KB 页面，4 路设置相关，32 字节线大小

指令	描述	处理器
IBTS	插入位串	Intel 80386 规范 A

0Fh, A7h, r/m　　ibts　　regW/memW, ax, cl, reg

插入位串指令从第二个操作数中取出一个位串，并把它放入到第一个操作数中。只有最早的 80386，即“规范 A”（A step）才包括了 IBTS 指令，在后面的规范中就删除了这条指令，以便为微指令留出空间。其他竞争者也没有用到这条指令，因为它代表着一种非常古老而古怪的 80386 芯片。

早期 80486 中的 CMPXCHG 指令也可以生成相同的指令码。据说，某些写于 80486 之前的应用程序在检查了有没有 IBTS 指令之后，错误地将 80486 当成了早期的 80386 A step。于是，Intel 在 80486 B step 中更新了 SMPXCHG 指令操作码。这一点使得大多数的开发人员难以区分新旧操作码，所以他们干脆不用 IBTS 或者 CMPXCHG 指令。参看本章后面的 XBTS 指令。

这条指令的格式可以是字格式，也可以是双字格式，这取决于当前的模式（16 位还是 32 位），还取决于是否使用了越界前缀。

指令	描述	处理器
ICEBP	内电路模拟器断点	大多数的386+

F1h, icebp 断点或中断1

这条指令有几种可任选模式。它可以作为一个单字节中断使用，类似于INT 3，只是它会调用中断1。在其他模式下，可以在一个内电路模拟器（ICE）或软件中使用它，来中断当前的进程，获得对系统的控制权。

ICEBP的操作形式依赖于芯片供应商提供的CPU类型。信息**“非常不明确”**指的是某些信息并非适用所有的供应商，或者同种类型的不同规范之间差别很大。可以在AMD386类型上使用相同的F1h操作码试一试SMI指令。

中断模式

在硬件重启之后的80386及后来的CPU上，ICEBP指令将以缺省模式按单字节访问中断1。只要调试寄存器7的第12位（一个未公开位）保持为0，该模式就将维持下去。在IBM的386和486 CPU上，型号专用寄存器1000h的第5位，也必须保持为0（参看WRMSR指令，以了解修改这一位的方法）。

通常情况下，中断1作为单步中断而保留下来。但是在这种模式下由ICEBP指令激活中断1后，不会被设置陷入标志（FLAGS寄存器的第8位）。当中断1的处理完成时，指令将继续执行下去，这一点不同于单步模式。

内电路模拟器模式

某些特殊版的Intel 80386和80486，以及所有的IBM 386SLC和486SLC CPU，都可以使用模拟器。Intel为安装ICE的连接版附加了一些线路来连接模拟器硬件。IBM类型也包含必要的连接。如果要利用这种属性，所有类型的CPU必须附加必要的硬件。

如果将调试寄存器7的第12位设置为1，就可以将ICEBP的指令模式改为内电路模拟器模式。开始执行ICEBP指令时，CPU从普通用户内存切换到隐存。模拟器断点处理程序获得控制。在本章的内电路模拟一节中将详细阐述隐存。

如果CPU没有附带与隐存的必要连接手段，而ICEBP又工作在内电路模拟器模式下，那么将会挂起系统：CPU将一直试图切换到隐存状态下，可是在非连接芯片上不存在这个隐存，它也不大可能存在于带有IBM CPU的主板上。系统必须带有隐存，正如一个ICE系统所提供的那样，否则ICEBP指令会试图执行一些无意义的数据。当运行在内电路模拟器状态下时，如果没有隐存，另外两种情况也会造成问题。如果软件通过标准中断指令INT 1激活了中断1，或者试图运行在单步模式下（这也会激活1号中断），系

统就会发生冲突。在这些情况下，不管是谁激活了中断 1，CPU 都将会试图切换到隐存状态下。

在模拟器模式下，ICGBP 可用来在你的程序中指定的地点中止模拟器。许多情况下，你可能不知道（或留意）从何处装入你的程序。在你的源代码中某个感兴趣的地方插入 ICEBP，就会在该点中止模拟器。在触发 ICE 命令“GO TIL 0”之后，调试寄存器的第 14 位会被设置为 1。只要开始执行 ICEBP 命令，模拟器都会停止下来。在 Intel 的模拟器中，屏幕上会显示出一条警告：“Unknown break point at adress xxxx:xxxx:xxxxxxxx”（xxxx:xxxx:xxxxxxxx 处的未知断点）。

ICEBP 可以工作在所有模式下，包括实模式，V86 模式以及保护模式，甚至在支持分页时也可以正常工作。

寄存器存储模式

IBM 和其他一些供应商还提供了另外一种 ICEBP 模式，它可以切换到隐存状态并保存 CPU 中每个寄存器的值。同内电路模拟器模式一样，调试寄存器 7 的第 12 位必须设置为 1。另外，还要将型号专用寄存器 1000h 的第 15 位设置为 1，来开放寄存器存储模式。如何设置这一位参看 WRMSR 指令。在开放了这个模式时，一旦执行了 ICEBP 指令，系统就会切换到隐存状态，并保存所有寄存器，然后切换到实模式，重新设置 CPU 中的所有寄存器，跳到 ICE 中或者中止代码。同时，所有的代码和数据都不通过内部 CPU 高速缓存，所以高速缓存的内容不会改变。

至于膝上型电脑的挂起/恢复属性，ICEBP 会将它当作功率中断 CPWI 来处理。可以从本章后面有关挂起/恢复模式的部分获得更多信息。

一个内电路模拟器可以显示或改变 ICEBP 保存在隐存中的寄存器。如果要继续执行程序，可以执行 LOADALL 指令，来装入所有的 CPU 寄存器，退出隐存，并从 CS：IP 值指向的地址开始执行。参看 LOADALL 指令中所有寄存器的有关定义，它们排列的顺序和 ICEBP 保存它们的一样。

指令	描述	处理器
IN/OUT	端口 22h/23h 访问 CPU	大多数的 Cyrix CPU

```
E4h,  2xh        in   al, 2xh        ; 输入端口
E6h,  2xh        out 2xh, al         ; 输出端口
```

Cyrix CPU 通过端口 22h 和 23h 来访问内部机器寄存器。为了访问这些寄存器，有必要将寄存器 AL 中的值输出到端口 22h。如果该值在一个有效的范围之内，那么在 6 个时钟的延迟之后（或者更多），就可以通过端口 23h 来读写指定的寄存器。在每次读或写之前，你必须将这个寄存器的值写到端口 22h。无效的访问（队列出错或者超出了寄存器值的范

围）将会向外部总线发出I/O请求。

提供了下列寄存器：

寄 存 器	描 述	Cyrix处理器
20h	执行速度控制寄存器	5x86
C0h	配置控制寄存器0	486SLC/DLC,6x86
C1h	配置控制寄存器1	486SLC/DLC,5x86,6x86
C2h	配置控制寄存器2	486S/S2/D/D2/DX/DX2/DX4,5x86，6x86
C3h	配置控制寄存器3	486S2/D2/DX/DX2/DX4,5x86,6x86
C4h	地址区域0，位31~24	486DLC，6x86
C5h	地址区域0，位23~16	486SLC/DLC，6x86
C6h	地址区域0，位15~12	486SLC/DLC，6x86
C7h	地址区域1，位31~24	486DLC，6x86
C8h	地址区域1，位23~16	486SLC/DLC，6x86
C9h	地址区域1，位15~12	486SLC/DLC，6x86
CAh	地址区域2，位31~24	486DLC，6x86
CBh	地址区域2，位23~16	486SLC/DLC，6x86
CCh	地址区域2，位15~12	486SLC/DLC，6x86
CDh	地址区域3，位31~24	486DLC，6x86
CDh	SMM区域起点，位31~24	486S/S2/D/D2/DX/DX2/DX4,5x86
CEh	地址区域3，位23~16	486SLC/DLC，6x86
CEh	SMM区域起点，位23~16	486S/S2/D/D2/DX/DX2/DX4,5x86
CFh	地址区域3，位15~12	486SLC/DLC，6x86
CFh	SMM区域起点，位15~12	486S/S2/D/D2/DX/DX2/DX4,5x86
D0h	地址区域4，位31~24	6x86
D1h	地址区域4，位23~16	6x86
D2h	地址区域4，位15~12	6x86
D3h	地址区域5，位31~24	6x86
D4h	地址区域5，位23~16	6x86
D5h	地址区域5，位15~12	6x86
D6h	地址区域6，位31~24	6x86
D7h	地址区域6，位23~16	6x86

续表

寄 存 器	描 述	Cyrix 处理器
D8h	地址区域 6，位 15~12	6x86
D9h	地址区域 7，位 31~24	6x86
DAh	地址区域 7，位 23~16	6x86
DBh	地址区域 7，位 15~12	6x86
DCh	区域控制 0	6x86
DDh	区域控制 1	6x86
DEh	区域控制 2	6x86
DFh	区域控制 3	6x86
E0h	区域控制 4	6x86
E1h	区域控制 5	6x86
E2h	区域控制 6	6x86
E3h	区域控制 7	6x86
E8h	配置控制寄存器 4	5x86,6x86
E9h	配置控制寄存器 5	6x86
F0h	电源管理	5x86
FEh	设备标识寄存器 0	486S2/D2/DX/DX2/DX4,5x86,6x86
FFh	设备标识寄存器 1	486S2/D2/DX/DX2/DX4,5x86,6x86

Cyrix CPU 的 I/O 端口寄存器细节

注意：如果某个 CPU 类型的位没有标出，则表示这一位被这个 CPU 保留。

寄存器	描述	Cyrix 处理器
20h	执行速度控制寄存器	5x86

PCR0	位	7=0	可以重新安排内存的读和写操作，来获得最佳运行速度
		6=x	保留
		5=x	保留
		4=x	保留
		3=x	保留
		2=1	提高了运行的速度，采用的方法是，如果跳转目的

已经位于这个队列中，那么就不使用预取队列。

1=1 开放了分支预测功能。

0=1 提高了运行速度，采用的方法是，开放了堆栈返回。RET 指令将按照期望的那样执行 CALL 调用指令后面的指令。

寄存器	描述	Cyrix 处理器
C0h	配置控制寄存器 0	486SLC/DLC、6x86

CCR0	位		
		7=1	开放挂起特性（SUSP 输入和 SUSPA 输出引脚）*
		6=0	高速缓存设置为 2 路相连方式*
		1	直接映射高速缓存*
		5=1	在开始 HOLD 状态之前清空内部高速缓存*
		4=1	开放清洗输入引脚*
		3=1	开放 KEN 输入引脚*
		2=1	开放 A20 输入引脚*
		1=0	可以对地址区域 640KB 到 1MB 进行高速缓存。
		0=1	1 MB 地址的前 64KB 不可以进行高速缓存*

*在 6x86 上保留了这些值，可能用作其他用途或者未使用。

寄存器	描述	Cyrix 处理器
C1h	配置控制寄存器 1	486SLC/DLC、5x86、6x86

CCR1	位		
		7=1	地址区域 3 设计成 SMM 地址空间（仅 6x86）*
		6=x	保留
		5=x	保留
		4=0	在总线周期内对 LOCK 引脚不求反，以提高运行速度（仅 6x86）
		3=1	在一个 SMI 服务中，所有的数据访问获得的是系统内存而不是 SMI 内存（仅 5x86）
		2=1	测试 SMI 内存——在一个 SMI 地址区域内访问 SMI 内存而不是访问系统内存（仅 5x86 和 6x86）*
		1=1	开放 SMI 特性的 SMI#I/O 引脚和 SMIACT#输出引脚（仅 5x86 和 6x86）*
		0=1	开放输出引脚 RPLSET 和 RPLVAL#（仅 486）

*只有在 CCR3（寄存器 C3h）的位 0 开放时，才可以对这些位进行写访问。

寄存器	描述	Cyrix 处理器
C2h	配置控制寄存器 2	486S/S2/D/D2/DX/DX2/DX4、5x86、6x86

CCR2	位	7=1	开放挂起特性的 SUSP#输入引脚和 SUSPA#输出引脚
		6=1	允许使用 16 字节的脉冲回写周期
		5=1	在请求 HOLD 和确认 HLDA 之前，通过回写所有的垃圾高速缓存数据来开放高速缓存的一致性校验。
		4=1	写保护区域 1，内存地址 640KB 到 1MB（6x86）。所有对内存地址范围 640Kb 到 1MB，并命中了内部高速缓存的写操作，也会触发外部总线（仅 486 和 5x86）
		3=1	在执行 HLT 指令时，CPU 进入挂起模式
		2=0	可以改变 CR0 的位 29（内部高速缓存的回写模式）
		1=1	允许回写高速缓存接口。开放 INVAL 和 WM_RST 输入引脚（仅 486 和 5x86）、高速缓存#和 HITM#输出引脚（仅 5x86），以及 HITM#引脚（仅 486）。
		0=1	保留

寄存器	描述	Cyrix 处理器
C3h	配置控制寄存器 3	486S2/D2/DX/DX2/DX4、5x86、6x86

CCR3	位	7=x	控制对端口 22h/23h 的访问（仅 5x86 和 6x86）
		6=x	0 = 只有配置寄存器 C0 到 CFh、FEh 和 FFh 可以访问。
		5=x	1 = 所有的寄存器都可以访问。
		4=x	
		3=0	SMM 引脚的功能和 Intel 的 486SL CPU 定义相同（仅 5x86）
		1	SMM 引脚的功能和标准的 Cyrix SMM 模式定义相同（仅 5x86）
		2=0	线性地址脉冲周期使用“1+4”Pentium 兼容排序（仅 5x86 和 6x86）
		1	线性地址脉冲周期使用线性排序（仅 5x86 和 6x86）
		1=0	在 SMM 模式期间不识别不可屏蔽中断
		0=1	写保护 SMM 配置寄存器位 CCR0 的位 1、2、3；CCR3 的位 1、3；以及所有 SMM 地址区域寄存器。在 SMM 模式下这些位和寄存器总是可写的。设置了这一位后，只有 CPU 重启就可以清除这一位。这些位在 SMM 模式下都是可写的，而不论这一位的状态如何。只有在 BIOS 中才可以一次性设置 SMM 功能，并且从此以后就不能再修改。

寄存器	描述	Cyrix 处理器
C4h	地址区域 0，位 31~24	486DLC、6x86

指定某个内存区域的地址。内存地址的位 31~12 出现在三个连续的 8 位寄存器中。在第三个寄存器中指定这个区域的大小。在 486DLC 上不可以高速缓存内存区域。在 6x86 上，与之相应的区域控制寄存器定义了区域的类型。

寄存器	描述	Cyrix 处理器
C5h	地址区域 0，位 23~16	486SLC/DLC、6x86

区域内存地址的位 23 到 16。参看寄存器 C4h 了解更多的细节。

寄存器	描述	Cyrix 处理器
C6h	地址区域 0，位 15~12	486SLC/DLC、6x86

这个寄存器的高四位是区域内存地址的位 23 到 16。低四位指示了其大小。参看寄存器 C4h 了解更多的细节。

位				
	7=x	区域起始地址的位 15		
	6=x	区域起始地址的位 14		
	5=x	区域起始地址的位 13		
	4=x	区域起始地址的位 12		
	3=x	区域的大小*	区域 0~6	区域 7
	2=x	0=	禁止	禁止
	1=x	1=	4KB	256KB
	0=x	2=	8KB	512KB
		3=	16KB	1MB
		4=	32KB	2MB
		5=	64KB	4MB
		6=	128KB	8MB
		7=	256KB	16MB
		8=	512KB	32MB
		9=	1MB	64MB
		A=	2MB	128MB
		B=	4MB	256MB
		C=	8MB	512MB
		D=	16MB	1GB
		E=	32MB	2GB
		F=	4GB	4GB

*在 486 上仅区域 0~3 可用。

寄存器	描述	Cyrix 处理器
C7h	地址区域 1，位 31~24	486DLC、6x86

参看区域 0，寄存器 C4h 了解相关细节。

寄存器	描述	Cyrix 处理器
C8h	地址区域 1，位 23~16	486SLC/DLC、6x86

参看区域 0，寄存器 C5h 了解相关细节。

寄存器	描述	Cyrix 处理器
C9h	地址区域 1，位 15~12	486SLC/DLC、6x86

参看区域 0，寄存器 C6h 了解相关细节。

寄存器	描述	Cyrix 处理器
CAh	地址区域 2，位 31~24	486DLC、6x86

参看区域 0，寄存器 C4h 了解相关细节。

寄存器	描述	Cyrix 处理器
CBh	地址区域 2，位 23~16	486SLC/DLC、6x86

参看区域 0，寄存器 C5h 了解相关细节。

寄存器	描述	Cyrix 处理器
CCh	地址区域 2，位 15~12	486SLC/DLC、6x86

参看区域 0，寄存器 C6h 了解相关细节。

寄存器	描述	Cyrix 处理器
CDh	地址区域 3，位 31~24	486DLC、6x86

参看区域 0，寄存器 C4h 了解相关细节。

寄存器	描述	Cyrix 处理器
CDh	SMM 区域起点，位 31~24	486S/S2/D/D2/DX/DX2/DX4、5x86

指定内存的 SMM 区域的地址。内存地址的位 31~12 出现在三个连续的 8 位寄存器中。在第三个寄存器中指定这个区域的大小。

寄存器	描述	Cyrix 处理器
CEh	地址区域 3，位 23~16	486SLC/DLC、6x86

参看区域 0，寄存器 C5h 了解相关细节。

寄存器	描述	Cyrix 处理器
CEh	SMM区域起点，位23~16	486S/S2/D/D2/DX/DX2/DX4、5x86

SMM 内存地址的位 23 到 16。参看寄存器 CDh 了解更多的细节。

寄存器	描述	Cyrix 处理器
CFh	地址区域3，位15~12	486SLC/DLC、6x86

参看区域 0，寄存器 C6h 了解相关细节。

寄存器	描述	Cyrix 处理器
CFh	SMM区域起点，位15~12	486S/S2/D/D2/DX/DX2/DX4、5x86

SMM 内存地址的位 15~12。参看寄存器 C6h 了解更多的细节。

寄存器	描述	Cyrix 处理器
D0h	地址区域4，位31~24	6x86

参看区域 0，寄存器 C4h 了解相关细节。

寄存器	描述	Cyrix 处理器
D1h	地址区域4，位23~16	6x86

参看区域 0，寄存器 C5h 了解相关细节。

寄存器	描述	Cyrix 处理器
D2h	地址区域4，位15~12	6x86

参看区域 0，寄存器 C6h 了解相关细节。

寄存器	描述	Cyrix 处理器
D3h	地址区域5，位31~24	6x86

参看区域 0，寄存器 C4h 了解相关细节。

寄存器	描述	Cyrix 处理器
D4h	地址区域5，位23~16	6x86

参看区域 0，寄存器 C5h 了解相关细节。

寄存器	描述	Cyrix 处理器
D5h	地址区域 5，位 15~12	6x86

参看区域 0，寄存器 C6h 了解相关细节。

寄存器	描述	Cyrix 处理器
D6h	地址区域 6，位 31~24	6x86

参看区域 0，寄存器 C4h 了解相关细节。

寄存器	描述	Cyrix 处理器
D7h	地址区域 6，位 23~16	6x86

参看区域 0，寄存器 C5h 了解相关细节。

寄存器	描述	Cyrix 处理器
D8h	地址区域 6，位 15~12	6x86

参看区域 0，寄存器 C6h 了解相关细节。

寄存器	描述	Cyrix 处理器
D9h	地址区域 7，位 31~24	6x86

参看区域 0，寄存器 C4h 了解相关细节。

寄存器	描述	Cyrix 处理器
DAh	地址区域 7，位 23~16	6x86

参看区域 0，寄存器 C5h 了解相关细节。

寄存器	描述	Cyrix 处理器
DBh	地址区域 7，位 15~12	6x86

参看区域 0，寄存器 C6h 了解相关细节。

寄存器	描述	Cyrix 处理器
DCh	区域控制 0	6x86

定义相关地址区域的类型和控制。

位		
	7=x	保留
	6=x	保留
	5=1	访问相关区域时，没有确认LBA#输出引脚
	4=1	地址区域写通
	3=1	开放地址区域的写收集
	2=1	开放地址区域的弱锁功能
	1=1	开放地址区域的弱写排序
	0=1	禁止区域的高速缓存（仅区域0~6）
	1	开放区域的高速缓存（仅区域7）

寄存器	描述	Cyrix处理器
DDh	区域控制1	6x86

参看区域控制0，寄存器DCh了解相关细节。

寄存器	描述	Cyrix处理器
DEh	区域控制2	6x86

参看区域控制0，寄存器DCh了解相关细节。

寄存器	描述	Cyrix处理器
DFh	区域控制3	6x86

参看区域控制0，寄存器DCh了解相关细节。

寄存器	描述	Cyrix处理器
E0h	区域控制4	6x86

参看区域控制0，寄存器DCh了解相关细节。

寄存器	描述	Cyrix处理器
E1h	区域控制5	6x86

参看区域控制0，寄存器DCh了解相关细节。

寄存器	描述	Cyrix处理器
E2h	区域控制6	6x86

参看区域控制0，寄存器DCh了解相关细节。

寄存器	描述	Cyrix 处理器
E3h	区域控制 7	6x86

参看区域控制 0，寄存器 DCh 了解相关细节。

寄存器	描述	Cyrix 处理器
E8h	配置控制寄存器 4	5x86、6x86

CCR4	位	7=1	支持 CPUID 指令，同时也支持 EFLAGS 的位 21 的可写特性（如果这一位可写，那么支持 CPUID 指令）
		6=x	保留
		5=x	保留
		4=1	允许目录表入口（Directory Table Entry）高速缓存
		3=1	允许绕过内存读（仅 5x86）
		2=x 1=x 0=x	I/O 访问间的最小的总线时钟周期数 0= 无时钟延迟 1= 2 个时钟的延迟 2= 4 个时钟的延迟 3= 8 个时钟的延迟 4= 16 个时钟的延迟 5= 32 个时钟的延迟 6= 64 个时钟的延迟 7= 128 个时钟的延迟

寄存器	描述	Cyrix 处理器
E9h	配置控制寄存器 5	6x86

CCR5	位	7=x	保留
		6=x	保留
		5=1	开放所有的地址区域寄存器（C4h~DBh）
		4=1	确认 LBA#输出引脚，所有的内存访问可以访问区域 640KB~1MB）
		3=x	保留
		2=x	保留
		1=x	保留
		0= 0	对丢失的读和写分配新的高速缓存线
		1	仅对丢失的读分配新的高速缓存线

寄存器	描述	Cyrix 处理器
F0h	电源管理	5x86

PMR	位	7=x	保留
		6=x	保留
		5=x	保留
		4=x	保留
		3=x	保留
		2=1	半速时钟——CPU 运行的速度是外部总线速度的一半，并忽略位 0 和 1。在进行外部总线传送时，内部时钟会增加传送周期的频率，并在完成时回到半速时钟状态。
		1=x 0=x	内部时钟/总线的时钟比（参看位 2） 0=1：1 1=2：1（上电时如果 CLKMUL 引脚=0，则这是缺省值） 2=保留 3=3：1（上电时如果 CLKMUL 引脚=1，则这是缺省值）

寄存器	描述	Cyrix 处理器
FEh	设备标识寄存器 0	486S2/D2/DX/DX2/DX4、5x86、6x86

CPU 类型的只读寄存器。参看寄存器 FFh 了解相关细节。

寄存器	描述	Cyrix 处理器
FFh	设备标识寄存器 1	486S2/D2/DX/DX2/DX4、6x86

只读寄存器，指示 CPU 级别（位 7~4）、版本（位 3~0）。下面列出了一部分标识：

FEh 寄存器	FFh 寄存器	类　型　名
0	n/a	Cx486SLC
1	n/a	Cx486DLC
2	n/a	Cx486SLC2
3	n/a	Cx486DLC2
4	n/a	Cx486SRx （Cx486SLC 的零售升级品）
5	n/a	Cx486DRx （Cx486DLC 的零售升级品）
6	n/a	Cx486SRx2（Cx486SLC2 的零售升级品）
7	n/a	Cx486DRx2（Cx486DLC2 的零售升级品）
10h	n/a	Cx486S(B step)
11h	n/a	Cx486D2 (B step)

续表

FEh 寄存器	FFh 寄存器	类　型　名
12h	n/a	Cx486Sc(B step)
13h	n/a	Cx486S2e(B step)
1Ah	5	Cx486DX-40
1Bh	8	Cx486DX2-50
1Bh	Bh	Cx486DX2-66
1Bh	31h	Cx486DX2-v80
1Fh	36h	Cx486DX4-v100
28h	n/a	5x86 1xs
29h	n/a	5x86 2xs
2Ah	n/a	5x86 1xp
2Bh	n/a	5x86 2xp
2Ch	n/a	5x86 4xs
2Dh	n/a	5x86 3xs
2Eh	n/a	5x86 4xp
2Fh	n/a	5x86 3xp
30h	?	6x86 1xs
31h	?	6x86 2xs
32h	?	6x86 1xp
33h	?	6x86 2xp
34h	?	6x86 4xs
35h	?	6x86 3xs
36h	?	6x86 4xp
37h	?	6x86 3xp

注：① n/a—— 在这个类型上没有提供这个寄存器；
②? ——寄存器值未知。

指令	描述	处理器
LOADALL	装入所有的处理器寄存器	某些 286--486
0Fh,05h	loadall	；从 0:800h 处装入寄存器（仅 286）
0Fh,07h	loadall	；从 es:edi 处装入寄存器（386--486）

最初使用 LOADALL 指令的芯片类型是 80286，它使用这条指令来实现在一次操作中装入所有的 CPU 寄存器，这些寄存器包括大量隐藏的、不可访问的寄存器。80386

对操作码做了大量的修改，而 Intel 的 80486 则完全抛弃了这条指令。令人感到奇怪的是，某些其他的 CPU 制造商，例如 IBM 和 AMD，仍然在它们生产的 486 系列中包含了这条指令。

Intel 使用 LOADALL 指令来测试已经生产好的 CPU 芯片，同时将它作为内电路模拟器（In-Circuit-Emulator，ICE）的一种手段，来实现在一次操作中装入所有的 CPU 寄存器。Intel 从来没有想过要在程序中使用 LOADALL 指令，因此也就没有公开这条指令。但是一些聪明的程序员还是发现了这条指令，他们使用这条指令实现了不用进入保护模式就可以快速访问扩展内存（1MB 以上）。

如果要访问系统扩展内存，可以替代 LOADALL 的唯一的方案是进入保护模式。但是在 286 进入保护模式需要花费很长的时间。这个进程（可能会使用到 XMS 功能）必须先切换到保护模式，将扩展内存的内容传递到期望的主存区，然后重新启动 CPU 回到实模式。利用一个特殊的 BIOS 模式，BIOS 能在重启 CPU 后恢复运行用户代码而不用重新执行上电自检程序。这里还提供了几个 BIOS 子程序来访问扩展内存，但是这些子程序的运行速度非常缓慢。

在实模式下使用 LOADALL 指令时，程序还可以装入隐藏的描述符高速缓存寄存器，而保持系统的实模式状态不变。CPU 使用隐藏的描述符高速缓存寄存器来限制对内存的访问，设计描述符高速缓存寄存器的目的就是为了限制程序的访问和保护操作系统以及多任务操作系统环境下的其他程序。在 BIOS 初始化后，缺省的情况是，设置这些寄存器的状态，使它们模拟 8088 环境而只可以访问前 1MB 的内存空间。

在 80286 上，LOADALL 指令是可以真正访问扩展内存的唯一可行的方案。要使 80286 上的内存管理器真正有用，就必须使用这条指令。你还会发现，即使是 Microsoft，也在 HIMEM.SYS 中使用它来提供 XMS 服务，这个服务随 DOS 和 Windows 环境一起发售。

另外一个容易引起读者兴趣的是，大多数的 80386 BIOS 会挂起坏操作码中断 6。如果是 80286 的 LOADALL 指令造成坏操作码中断，中断 6 就会更改数据而执行 80386 的 LOADALL 指令。

在模拟器断点或挂起操作之后，模拟器和膝上型电脑的休眠模式也可以使用 LOADALL 指令来重启 CPU，以便进入一个指定的 CPU 状态。在这种情况下，就要求已经保存过了所有的寄存器。为了获得所有的内部寄存器，一些新式的芯片针对低功耗休眠模式提供了一些特殊的增强功能。

在使用 LOADALL 指令时，不会检查所装入的任意数据。如果装入的数据不合法，CPU 就可能产生意想不到的操作结果。在使这条指令执行时，你必须熟悉保护模式下的编程技术以及相关的寄存器和描述符表。如果在执行这条指令期间出现了 CPU 错误，就会挂起处理器。页面错误也会导致 CPU 挂起。只有在优先级 0 下才可以实行这条指令。AMD 的 486 DXLV 只允许在系统管理模式下运行这条指令，当试图在其他处理器状态下执行这条指令时，会产生非法操作码中断。

80286 上的 LOADALL 指令

执行 LOADALL 指令时，CPU 从 0：800h 地址处的物理内存中读取 51 个字，并将这 51 个字装入到寄存器和各种描述符中。表 3-11 列出了某个 80286 的 LOADALL 指令表。表 3-12 列出了六种描述符的结构层次。

表 3-11　LOADALL 字表，80286

内存地址	字　　数	装入的寄存器
800h	3	无
806h	1	MSW——机器状态字
808h	7	无
816h	1	TR——任务寄存器
818h	1	Flags
81Ah	1	IP
81Ch	1	LDTR——逻辑描述符表寄存器
81Eh	1	DS
820h	1	SS
822h	1	CS
824h	1	ES
826h	1	DI
828h	1	SI
82Ah	1	BP
82Ch	1	SP
82Eh	1	BX
830h	1	DX
832h	1	CX
834h	1	AX
836h	3	ES 描述符
83Ch	3	CS 描述符
842h	3	SS 描述符
848h	3	DS 描述符
84Eh	3	GDT 描述符（Global Descriptor Table，全局描述符表）
854h	3	LDT 描述符（Local Descriptor Table，局部描述符表）
85Ah	3	IDT 描述符（Interupt Descriptor Table，中断描述符表）
860h	3	TSS 描述符（Task State Segment，任务状态段）

表 3-12 LOADALL 描述符字节，80286

偏移量	字节数	描述	
0	3	24 位的物理基地址	
3	1	对于 IDT 和 GDT，设置为 0	
		段访问权限字节——参看 Intel CPU 手册了解其他取值的详细描述	
		对于 CS：	9Bh=只可以执行
			93h=可执行、可读、可写
		对于 DS/ES/SS：	93h=可读、可写
		对于 LDT：	82h=指定它是一个 LDT
		对于 TSS：	89h=任务状态可以切换
4	2	16 位的段址限制，单位是字节	
		使用 FFFFh 可以访问一个段内的所有 64K 字节	

80386/80486 上的 LOADALL 指令

执行 LOADALL 指令时，CPU 从 ES：EDI+100h 指向的物理内存读取 10 个双字，然后再从 ES：EDI 所指向的地址读取 51 个双字。即使系统工作在保护模式下，ES 也必须是一个真正的模式段值，而不是一个选择符。

同 80286 的 LOADALL 指令的功能相似，这 61 个双字被装入到寄存器和各种描述符中，表 3-13 列出了 80386 的基本的 LOADALL 表。表 3-16 列出了描述符的基本层次结构。某些 AMD 类型（386 和 486）所使用的表与 Intel 和 IBM 所提供的表有所不同。

Intel 没有在 80486 和后来的 CPU 上提供 LOADALL 指令，这只是我个人的看法。我的测试也证明了这一点，但是 Intel 也有可能将这个指令和某些特殊的寄存器或者费解的方案设计一起隐藏起来。

表 3-13 LOADALL 双字表，386/486

ES：EDI 偏移量	双字数	装入的寄存器
00h	1	CR0
04h	1	EFLAGS
08h	1	EIP
0Ch	1	EDI
10h	1	ESI
14h	1	EBP
18h	1	ESP
1Ch	1	EBX

续表

ES：EDI 偏移量	双字数	装入的寄存器
20h	1	EDX
24h	1	ECX
28h	1	EAX
2Ch	1	DR6
30h	1	DR7
34h	1	TR（任务状态选择符寄存器）
38h	1	LDTR（局部描述符表寄器）
3Ch*	1	GS
40h*	1	FS
44h*	1	DS
48h*	1	SS
4Ch*	1	CS
50h*	1	ES
54h	3	TSS 描述符（任务状态段）
60h	3	IDT 描述符（中断描述符表）
6Ch	3	GDI 描述符（全局描述符表）
78h	3	LDT 描述符（局部描述符表）
84h	3	GS 描述符
90h	3	FS 描述符
9Ch	3	DS 描述符
A8h	3	SS 描述符
B4h	3	CS 描述符
C0h	3	ES 描述符
CCh**	1	表的长度
D0h**	12	没有使用这些字，LOADALL 也不会装入这些字
100h	1	临时寄存器 TST
104h	1	临时寄存器 IDX
108h	1	临时寄存器 H
10Ch	1	临时寄存器 G
110h	1	临时寄存器 F
114h	1	临时寄存器 E
118h	1	临时寄存器 D
11Ch	1	临时寄存器 C
120h**	1	临时寄存器 B
124h **	1	临时寄存器 A

*双字的高字节总是为零；

**AMD 486 的 LOADALL 指令会忽略这些值。

IBM的386和486芯片都支持LOADALL指令。这个类型还可以选择是装入普通的61个字呢还是装入IBM所有的CPU寄存器，后面一种选择共要装入73个双字。如果设置了型号专用寄存器1000h的第5位，就会装入73个双字。表3-14列出了保存在前61个双字后面的附加信息。

表3-14 IBM的386/486上可选的LOADALL双字数*

ES：EDI偏移量	双字数	装入的寄存器
128h	1	CR2
12Ch	1	CR3
130h	1	MSR——型号专用寄存器1001h，位0~31
134h	1	MSR——型号专用寄存器1001h，位32~63
138h	1	MSR——型号专用寄存器1000h，位0~14
13Ch	1	DR0
140h	1	DR1
144h	1	DR2
148h	1	DR3
14Ch**	1	PEIP——前续隐存空间指令指针

*只有在型号专用寄存器1000h设置了其ICE开放位（位14）之后，才会装入这些寄存器（参看指令WRMSR）。

**LOADALL从来没有将前续隐存空间指令指针装入到CPU中，但是如果从CPU中转储这个寄存器，就会显示出这个寄存器。参看本章有关挂起和恢复一节，来进一步了解有关隐存空间的内容。

表3-15 AMD的486DXLV上其他的LOADALL双字数

ES：EDI偏移量	双字数	装入的寄存器
128h *	1	PEIP——前续隐存空间指令指针
12Ch**	7	未使用
148h	2	未使用，SMI不会装入它
150h	22	浮点内部寄存器

*LOADALL从来没有将前续隐存空间指令指针装入到CPU中，但是如果从CPU中转储这个寄存器，就会显示出这个寄存器。参看本章有关挂起和恢复一节，来进一步了解有关隐存空间的内容。

**AMD486的LOADALL（RES4）指令会忽略这个值。

表 3-16 LOADALL 描述符字节，386/486

偏 移 量	字 节 数	描 述	
0	0	未使用，设置为 0	
1	1	对于 IDT 和 GDT，设置为 0	
		对于其他所有——重新定义了访问权限字节的位 7（当前位），将它定义成一个合法位。位值为 0 表示描述符无效，如果这时使用描述符，就会产生一个一般保护性错误（General Protection Fault）中断（中断 Dh）	
		段访问权限字节——参看 Intel CPU 手册了解其他取值的详细描述：	
		对于 CS：	9Bh=只可以执行
			93h=可执行、可读、可写
		对于 DS/ES/FS/GS/SS:	93h=可读、可写
		对于 LDT：	82h=指定它是一个 LDT
		对于 TSS：	89h=任务状态可以切换
2	2	未使用，设置为 0	
4	4	32 位物理基地址	
8	4	32 位限制	

使用 LOADALL

正如我在前面所指出的那样，大多数人使用 LOADALL 指令，是为了实现在实模式下访问所有的内存。这一点需要将 CPU 的优先级设置为 0，带有安全级别的任意操作系统都不会让一个用户程序运行在优先级 0 下。这些操作系统包括 OS/2，NT 等等。当然 DOS 不带有任意的安全级别，在没有执行内存管理程序时，它就运行在实模式下，因此允许执行 LOADALL 指令。

我必须先解释一下描述符。在访问一个 B8006h 的内存地址（该地址是显示器的缓冲区）时，段寄存器中装入 B800h，然后使用指令访问 DS 段中的第 6 个字节。应用程序看不到的是 DS 段的描述符。为了确定实际的内存地址，CPU 先从隐藏的描述符寄存器中取出一个“基值”，并把它加到 DS 段中，然而加上偏移量。图 3-2 显示了这个过程。

	值	乘 数	结 果	控制的范围
内存偏移量	=0006h	1	6h	64KB
DS 段值	=B800h	10h	B8000h	1MB
描述符基地址	=000000h	1	0	16MB*

生成的内存地址			B0006h	

*4GB，如果是 32 位基地址（386+）；16MB，如果是 24 位基地址（286）。

图 3-2 计算内存地址，前 1MB

重启 CPU 后，描述符的基地址自动设置为 0。这样，应用程序就可以访问第 1 个 1MB 内存中的任意字节（在此，我忽略了 HMA 可以通过段 FFFF 访问额外的 64K-16 字节）。现在，如果我们希望访问开始于物理地址 200000h 的第 3 个 1MB，我们可以改变描述符的基地址，如下面图 3-3 所示。

	值	乘 数	结 果	控制的范围
内存偏移量	=0000h	1	0h	64KB
DS 段值	=0000h	10h	00000h	1MB
描述符基地址	=200000h	1	200000h	16MB*

生成的内存地址			200000h	

*4GB，如果是 32 位基地址（386+）；16MB，如果是 24 位基地址（286）。

图 3-3 计算内存地址，第三个 1MB

程序也可以在 DS 中装入从 0~FFFF 的值，然后访问第二兆字节中的任意内存。但是，利用 DS 段值，不可以访问 DS 段 1MB 以外的内存。除非在实模式下，否则没有其他的方法（不包括 LOADALL）可以改变描述符的值，这一点到目前为止已经非常明了。我所做的一些简化也使理解这一点变得非常容易。如果你对这些还不熟悉，那么请参看《Intel Pentium 处理器用户手册》，它将有助你熟悉描述符以及相关主题。

所以，在实模式下访问 1MB 以外内存的诀窍就是，使用 LOADALL 改变某个段的描述符基地址。如果你需要将第一兆字节中的内存数据传送到第二兆字节，可以将 DS 描述符基地址设置为 0，而将 ES 描述符基地址设置为 200000h。当然，也必须设置好其他所有的寄存器。执行完 LOASALL 之后，DS 段寄存器将允许访问任意的第一兆字节空间，而 ES 将允许访问任意的第二兆字节空间。利用一系列不同的指令可以容易地实现内存传送。例如，可以使用 MOVSB 将字节数据从 DS:[SI]传送到 ES:[DI]。

使用 LOADALL 实现实模式下的分页

80386 和 80486 可以将任意一个 4K 字节的物理块重新映射到一个不同的逻辑地址，这就是所谓的“分页”。分页后，软件就只使用逻辑地址。这一点有助于实现将 BIOS ROM 重新映射到快速的 32 位 RAM、提供 EMS 模拟、使内存出现在 PC 的上位内存块（UMB）中，以及其他一切技巧。Intel 分页设计的目的是为了供保护模式的程序来使用，这些程序包括操作系统和内存管理程序。不幸的是，要获得一个可以运行 DOS 程序的保护模式系统相当困难。大多数情况下，保护模式会使系统速度减慢 5%~10%，在中断频繁时速度减慢尤为显著。

在 80386 以及后来支持 LOADALL 的 CPU 上，LOADALL 可用来解决保护模式下运行速度的限制问题。LOADALL 可用来创建一个特殊的（可能不兼容）未公开模式，称为“分页模式”或者“大实模式”。显然，保护模式比实模式提供更广泛的附加性能，但是，如果保护模式只需要分页性能，那么 LOADALL 是实现这一点的唯一方法。

CR0 控制系统何时进入保护模式以及何时分页。只要一个移动指令就可以装入 CR0，并且在系统进入保护模式之前，CPU 不允许设置分页位。正如我们在前面所指出的那样，对 LOADALL 装入的值不会有任何的检查。如果设置了 CR0 的分页位，同时又进入了保护模式，那么 CPU 就会进入未公开的分页实模式。

在为 LOADALL 设置 CR0 时，所有的保留位必须置为 0。这一点真有点令人感到难以理解，因为一旦读取了 CR0 后，所有的保留位都会被设置为 1。

从系统管理模式“LOADALL”中恢复

AMD、Cyrix 以及 Intel 提供一种新的指令，用来替代 LOADALL 实现从某些 PC 系统管理模式中恢复过来。这条指令本质上和 LOADALL 一样，但是操作码有点不同。主要的不同在于，在某个特殊的系统管理模式下，CPU 逻辑只允许 RSM 指令。本章后面有关挂起和恢复的部分将详细讨论 RSM。

令读者感兴趣的是，Pentium 的 RSM 指令可对装入 CPU 的值作某些检查。特别地，在设置保护模式之前，它不允许设置 CR0 的分页位。这意味着它不支持前面提到的分页实模式。

指令	描述	平台
POP CS	从堆栈中装入 CS	8088/8086

```
0Fh          pop cs          ; 从堆栈中装入 CS
```

所有的 8088 和 8086 CPU 都可以执行 POP CS 指令。在执行这条指令时，当前栈顶的数据被装入到当前指令段，CS 寄存器中。这条指令并不太有用。Return Far 指令可以从堆栈栈中同时更新 CS 和 IP 寄存器。我还没有发现 POP CS 的任何有用之处。

我猜测，设计它可能是一个微指令失误，或者留下它仅仅只是为了保存芯片逻辑的少数某些位。如果仔细看一下指令映射表，你会发现这儿有四个 POP 段寄存器指令，它们都使用相同的位模式，000nn111。nn 指的就是段寄存器。

指令	nn	操作码
POP ES	00	07h
POP CS	01	0Fh

POP SS	10	17h
POP DS	11	1Fh

所有后来的处理器，包括NEC V20/V30 CPU，都重新使用POP CS操作码作为新的一组指令的起始字节。

指令	描述	平台
RDMSR	读型号专用寄存器	某些386+
0Fh，32h	**rdmsr**	**；读取型号专用寄存器ecx,并保存在edx:eax中**

这条指令读取型号专用寄存器，由ECX指定要读取的寄存器。返回时，EDX:EAX中装入64位寄存器值。参看指令“写型号专用寄存器”（WRMSR）获得全面的寄存器细节信息。

只有优先级为0的实模式和保护模式才允许执行RDMSR指令，否则，会产生一般保护性错误。

参看本章结尾的程序例（CPURDMSR），以获得可能的未公开寄存器。我们已在Intel 80486上运行了这个程序，没有发现任何的型号专用寄存器。Pentium有许多未公开的寄存器，它们定义高速缓存信息，翻译备用缓冲区（Translation Lookaside Buffer，TLB）等等。其他的CPU可能也隐藏了寄存器。需要记住的是，存在CPURDMSR无法标识的隐藏寄存器。如果在访问一些未公开的寄存器时需要设置一些未公开位，或者除了ECX外其他寄存器都需要一些特殊值来允许这种访问，那么就会发生这种情况。

指令	描述	平台
RDTSC	读时间标志计数器	Pentium+
0Fh，31h	**rdtsc**	**；读取时钟标志寄存器,并保存在edx:eax中**

这条指令是Pentium新推出的，用来访问内部时间标志计数器（Time Stamp Counter）的当前内容。重启后这个64位的时间标志计数器被设置为0，并按CPU速度开始记数。对于200MHz的时钟，这意味着每秒计数200,000,000次。在Pentium手册中Intel对它做了详细说明。

如果当前的优先级为0，那么随时可以使用RDTSC指令。当优先级非零时，只有在CR4中的TSD位2被清零之后，才可以读RDTSC。

也可以使用型号专用寄存器命令利用寄存器10h来访问时间标志计数器。借助于WRMSR，可以改变时间标志计数器值。

这是一种不用考虑其他使用者就可以精确计时的好方法。在事件开始时，读入并保存时间标志计数器的值，在事件结束后再次读入并保存该值。两者的差值就是该事件的持续

时间。由于该时钟是 64 位的，所以只有连续执行超过 8,000 年的事件才会让时钟循环一次，我们一生中也不可能遇到这种事件。

测量极短或极长时间的事件的方法相当简单，只须 6 条指令就可以获得事件所持续的时间。不幸的是，它只适用于 586+的 CPU，并且你必须知道时钟频率以确定时间误差。参看程序 CPUTYPE 中的子程序 CPUSTEED，可以获得 CPU 的速度信息。

```
    Rdtsc                          ; 获取时间标志
    mov     [count_hi],edx         ; 保存值
    mov     [count_lo],eax         ;
<<< 在这里放入要计时的事件或代码 >>>>
    rdtsc                          ; 获取秒时间标志
    sub     eax,[count_lo] ;计算差别
    sbb     edx,[count_hi] ;
                                   ; edx:eax=事件持续的时间
```

你可以做到测量尽可能地精确，除了取决于你所测量的对象之外，你还必须关闭所有的中断和 NMI，这一点可让中断不至于影响测量结果。在前面这个例子之前添加一条伪指令和两条指令，将会提高一点测量精度。

```
    ALIGN 4
    mov     [count_lo],eax ;保存从前一步开始的事件持续时间值
    mov     [count_hi],edx
```

它确保所有的数据存放在 L1 高速缓存中，否则，第一次使用 COUNT_HI 和 COUNT_LO 时，可能会产生高速缓存冲突，造成一些不确定的循环。

若一些循环会造成影响，那么可用下面的代码段来删除执行 RDTSC 所需的少量时间，并且保存寄存器。在代码计算了时间差之后，应马上执行该代码段。

记住，对齐代码和数据也会占用一些周期，所以用 ALIGN 4 来指示所有的数据都占据偶字地址。必须在数据定义之前使用另外的 ALIGN 4（没有举例）。

```
    mov     [temp_lo],eax      ;保存从前一步事件的持续时间值
    mov     [temp_hi],edx
    rdtsc                      ;获取时间标志
    mov     [count_hi],edx     ;保存值
    mov     [count_lo],eax     ;
    rdtsc                      ;获取秒时间标志
```

```
sub     eax,[count_lo]      ;计算差别
sbb     edx,[count_hi]      ;edx:eax 是事件的时间开销
sub     [temp_lo],eax
sbb     [temp_hi],edx
mov     eax,[temp_lo]
mov     edx,[temp_hi]       ;edx:eax 中保存了调整后的事件持续时间
```

除了 Pentium Pro，所有的处理器都没有确保会在 RDTSC 之前执行所有上述指令，在读操作之前就执行了 RDTSC 之后的指令也是有可能的。由于这两个优先指令一定会在 RDTSC 之前执行，而 RDTSC 之后的两个指令不可能在 RDTSC 之前执行，所以就我们前面的这个例子而言，不会发生这种问题。

现在 Pentium Pro 手册对 RDTSC 指令作了说明，但是 1995 年以前的手册并未公开这条指令。

指令	描述	平台
RSM	从系统管理模式下恢复	某些 386+
0Fh，AAh	rsm	；从 SMI 中恢复

RSM 会装入以前由系统管理中断保存的所有的 CPU 寄存器。它不装入现有的任意浮点寄存器，也不改变 Pentium 及其兼容机上的型号专用寄存器。程序将从上次系统管理中断发生的地点恢复执行下去。

其功能类似于 LOADALL，只是它只能在系统管理模式下执行。在其他任意模式下，例如实模式、保护模式或 V86 模式，试图执行 RSM 时会产生一个坏操作码错误并触发中断 6。

请参看本章挂起和恢复模式一节以获得详细信息。该节中表 3-14 列出了保存在隐存中的 CPU 寄存器，可以通过 RSM 指令恢复它们。

Intel 的 80386SL、80486SL 以及 Pentium 处理器都支持 RSM。Cyrix/TI 486SLC/e 也用到了这条指令，但是这种类型恢复的寄存器要少得多。某些 Intel 的 CPU 用户手册也阐述了这条指令。在《Cyrix SMM 程序员指南》中也有论述。

指令	描述	平台
SETALC	按进位标志设置 AL 的值	所有，除了 V20
D6h	setalc	；将进位标志的值转移到 AL 中

这条指令将 AL 中的所有位设置为进位标志的值。如果设置进位标志失败，则 AL 变成零；如果设置进位标志成功，则 AL 就变成 FFh。这条指令不影响任意标志位。

经测试后知，除 NEC 的 V20 外几乎所有的供应商的所有芯片都支持这种替代指令形式。我还发现某些老式的商业应用程序使用了这条指令。可以预见，如果所有将来的 CPU 想同以前的代码兼容，那么他们都会支持指令 SETALC。

下面的代码例可以实现与这条未公开指令一样的功能：

```
        pushf                   ;保存标志位
        mov         al,0        ;清除 al
        jnc         not_set     ;如果没有进位，则跳转
        dec         al          ;设置为 FFh
not_set:
        popf                    ;恢复标志位
```

指令	描述		平台
SHL	按计数值左移		80188+
C0h，r/m	shl	regB/memB, imm	；按立即数左移字节
C1h，r/m	shl	regW/memW, imm	；按立即数左移字
C1h，r/m	shl	regD/memD, imm	；按立即数左移双字

从 80188 开始，按立即数左移指令就成为了一个标准的指令。但是却没有公开另外一种执行相同操作的操作码格式。

标准格式和未公开格式之间的区别体现在 MM-REG-R/M 字节的 REG 域中。公开的操作格式有 3 位 REG 域，且固定为 100。而未公开的格式有 3 位 110 的 REG 域。

这种可替代的格式没有多大的用处。我曾经看到有一些有胆量的人就在一些由共件汇编程序生成的代码中使用这种格式。我想，这一点可以告诉我们，是否会有人在未获得汇编程序许可的情况下就使用汇编程序生成了一些代码。

经测试，所有供应商的所有芯片都支持这种替代格式。由于有一些代码使用到这种格式，所以在一段较长的时间内仍将支持这条指令。

指令	描述		平台
SHL	左移 1 位		所有
D0h，r/m	shl	regB/memB，1	；左移字节 1 位
D1h，r/m	shl	regW/memW，1	；左移字 1 位
D1h，r/m	shl	regD/memD，1	；左移双字 1 位

从 8088 开始，所有的处理器都支持左移 1 位操作。但是却没有公开另外一种执行相同操作的操作码格式。

标准格式和未公开格式之间的区别体现在 MM-REG-R/M 字节的 REG 域中。公开的操作格式有 3 位 REG 域，且固定为 100。而未公开的格式有 3 位 110 的 REG 域。

这种可替代的格式没有多大的用处。同未公开的按计数值左移格式一样，在一些由共享软件汇编程序生成的代码中我曾经看到过这种未公开的格式。

经测试，所有供应商的所有芯片都支持该替代格式。由于有一些代码用到这种格式，所以在一段较长的时间内仍将支持这条指令。

指令	描述	平台
SMI	系统管理模式入口	AMD 386SXLV、DXLV、486DXLV
F1h	smi	；调用 SMI 中断处理程序

这条指令为 AMD 的低功率类型所独有。它提供了一种进入系统管理模式的软件方法，在系统管理模式下可以访问隐存。大多数其他类型的处理器将 F1h 当作 ICE 断点。参看 ICEBP 指令。同 IBM 类型一样，LOADALL 也被用来从系统管理模式下恢复普通的操作。UMOV 用来在系统管理模式下访问用户内存。在挂起和恢复一节中将详细讨论系统管理模式。

AMD 新式的 DX2 和 DX4 486 系列以及 Am5k86 不支持 SMI 指令，而仅仅只可以使用硬件触发。

指令	描述	平台
TEST	带立即数测试寄存器	所有
F6h，r/m	test regB/memB，imm	；将字节和一个立即数相“与”
F7h，r/m	test regW/memW，imm	；将字和一个立即数相“与”
F7h，r/m	test regD/memD，imm	；将双字和一个立即数相“与”

从 8088 开始，每个处理器都支持带立即数指令的测试操作，但是没有公开另外一种执行相同操作的操作码。

标准格式和未公开格式之间的区别体现在 MM-REG-R/M 字节的 REG 域中。公开的操作格式有 3 位 REG 域，且固定为 100。而未公开的格式有 3 位 110 的 REG 域。

这种可替代的格式没有多大的用处。同未公开的左移格式一样，在一些由共享软件汇编程序生成的代码中我曾经看到过这种未公开的格式。

经测试，所有供应商的所有芯片都支持该替代格式。由于有一些代码用到这种格式，所以在一段较长的时间内仍将支持这条指令。

指令	描述	平台
UMOV	用户移动寄存器指令	某些 386/486
0Fh，10h，r/m	mov regB1/memB，regB2	；将字节从寄存器 2 移动到

```
                                                    ; 寄存器 1 或内存
0Fh, 11h, r/m    mov    regW1/memW, regW2           ; 将字从寄存器 2 移动到
                                                    ; 寄存器 1 或内存
0Fh, 12h, r/m    mov    regB1, regB2/memB           ; 将字节从寄存器 2 或内存
                                                    ; 移动到寄存器 1
0Fh, 13h, r/m    mov    regW1, regW2/memW           ; 将字从寄存器 2 或内存
                                                    ; 移动到寄存器 1
```

要理解用户移动指令，你必须先理解隐存，在挂起和恢复一节以及内电路模拟一节中详细阐述了它们。当激活了 CPU 的隐存后，通常不可访问用户内存区。当未激活隐存时，UMOV 执行与普通 MOV 相同的操作，操作码从 88h 到 8Bh。

除了不需要隐存的 Cyrix CPU 外，所有的 386 和 486 处理器都支持这条指令。在 Cyrix 上，UMOV 指令的功能类似于一个两字节指令 NOP，它不生成错误信息。Pentium 以及后来的 Intel 处理器指令集删去了 UMOV 指令。

UMOV 主要用于模拟器访问用户空间的数据和指令信息，这时，隐存处于激活状态。例如，在出现了 ICE 断点后，就会激活隐存。为了显示断点指令处的反汇编结果，必须从断点处的用户内存读取代码字节。UMOV 之所以是实现这一点的唯一方法，是由于在激活断点后，普通的 MOV 指令（以及几乎其他所有的指令）只对隐存进行操作。

UMOV 也可以用来区分 Cyrix486CPU 和其他类型的 CPU。一旦确定了某一个 CPU 是 Pentium 级的，就可以检查它是否提供 UMOV 指令。必须先挂起坏码中断 6，以免 CPU 执行这条命令。一旦 CPU 被标识为 386 或 486，就可以执行 UMOV 指令。接着检查是出现了 UMOV 传送指定数据的操作，还是触发了坏码中断。如果两种情况都不发生，那么 CPU 一定是 Cyrix 的。当然，这不能保证将来推出的 CPU 不会出现新的情况，参看 CPUTYPE 程序中的 CPUVEDOR 子程序，了解如何使用 UMOV 指令来帮助确定 CPU 的供应商信息。

这条指令格式是字还是双字，取决于当前模式是 16 位还是 32 位，以及是否使用一个大小覆盖前缀。这条指令不改变任何标志位。

指令	描述	平台
WRMSR	写型号专用寄存器	某些 386+

```
0Fh, 30h    wrmsr    ; 将 edx:eax 写到型号专用寄存器 ecx 中
```

许多 CPU 芯片都包括了许多新的寄存器来支持其特有的功能。这些专用寄存器在不同类型的 CPU 上不可能都获得相同的支持。现在，只有 Intel 的 Pentium CPU、AMD 的 Am5k86 以及 IBM 的 386 和 486 CPU 才支持型号专用寄存器。

在一般情况下，仅系统 BIOS 才使用型号专用寄存器，而一般的应用程序不使用型号专用寄存器。在一个应用程序中改变型号专用寄存器会产生意想不到的结果，具体会产生哪些结果就取决于硬件系统的执行情况以及 BIOS 是如何设置这些标志的。

在写一个型号专用寄存器时，在 ECX 中装入这个寄存器的编号，并将一个 64 位值装入到 EDX：EAX 中。然后使用 WRMSR 指令。只有在优先级为 0 时，在实模式和保护模式下才可以使用这条命令。否则，将出现一般保护性错误。

如果要读一个型号专用寄存器的内容，请参看读型号专用寄存器指令 RDMSR。在改变某个型号专用寄存器的任意一位之前，必须先读取型号专用寄存器的当前值。请只改变那些需要修改的位，而不要改变那些不需要改变的位。然后，新的双字寄存器值就会写回到型号专用寄存器中。例如：

```
mov     ecx, 1000h
rdmsr                          ; 读取寄存器 1000h
or      eax,20h                ; 设定 edx:eax 中的第 5 位为 1
wrmsr                          ; 写入寄存器 1000h
```

寄存器归纳

寄 存 器	描 述	处 理 器
0	机器检查地址	Pentium、Pentium Pro、Am5k86
1	机器检查类型	Pentium、Pentium Pro、Am5k86
2	奇偶请反寄存器（TR1）	Pentium
4	结束位测试寄存器（TR2）	Pentium
5	高速缓存数据测试寄存器（TR3）	Pentium
6	高速缓存状态测试寄存器（TR4）	Pentium
7	高速缓存控制测试寄存器（TR5）	Pentium
8	TLB 命令测试寄存器（TR6）	Pentium
9	TLB 数据测试寄存器（TR7）	Pentium
Bh	BTB 标识测试寄存器（TR9）	Pentium
Ch	BTB 目标测试寄存器（TR10）	Pentium
Dh	BTB 命令测试寄存器（TR11）	Pentium
Eh	新特性控制（TR12）	Pentium
10h	时间标志计数器	Pentium、Pentium Pro、Am5k86
11h	控制/事件选择寄存器	Pentium
12h	计数器 0	Pentium
13h	计数器 1	Pentium
1Bh	APICBASE	Pentium Pro

续表

寄 存 器	描 述	处 理 器
2Ah	上电功能	Pentium Pro
79h	BIOS 更新触发器寄存器	Pentium Pro
82h	阵列访问寄存器	Am5k86
83h	硬件配置寄存器	Am5k86
8Bh	BIOS 更新署名寄存器	Pentium Pro
C1h	执行计数器 0 控制	Pentium Pro
C2h	执行计数器 1 控制	Pentium Pro
FEh	内存类型范围寄存器	Pentium Pro
179h	机器检查全局功能	Pentium Pro
17Ah	机器检查全局状态	Pentium Pro
17Bh	机器检查全局控制	Pentium Pro
186h	事件选择 0 寄存器	Pentium Pro
187h	事件选择 1 寄存器	Pentium Pro
1D9h	调试控制 MSR 寄存器	Pentium Pro
1DBh	上一个从 IP 起的分支	Pentium Pro
1DCh	上一个到 IP 的分支	Pentium Pro
1DDh	上一个从 IP 起的中断	Pentium Pro
1DEh	上一个到 IP 的中断	Pentium Pro
1E0h	ROB_CR_BKUPTMPDR6	Pentium Pro
200h~20Fh	内存类型范围寄存器的物理基地址和屏蔽（每一个有 8 个寄存器）	Pentium Pro
250h	内存类型范围寄存器 64K_0	Pentium Pro
258h~259h	内存类型范围寄存器 16K (2 个寄存器)	Pentium Pro
268h~26Fh	内存类型范围寄存器 4K (8 个寄存器)	Pentium Pro
2FFh	内存类型范围寄存器	Pentium Pro
400h	机器检查 0 控制	Pentium Pro
401h	机器检查 0 状态	Pentium Pro
402h	机器检查 0 地址	Pentium Pro
403h	机器检查 0 多种	为将来的 Pentium 保留
404h	机器检查 1 控制	Pentium Pro
405h	机器检查 1 状态	Pentium Pro
406h	机器检查 1 地址	Pentium Pro

续表

寄存器	描述	处理器
407h	机器检查1多种	为将来的Pentium而保留
408h	机器检查2控制	Pentium Pro
409h	机器检查2状态	Pentium Pro
40Ah	机器检查2地址	Pentium Pro
40Bh	机器检查2多种	为将来的Pentium而保留
40Ch	机器检查4控制	Pentium Pro
40Dh	机器检查4状态	Pentium Pro
40Eh	机器检查4地址	Pentium Pro
40Fh	机器检查4多种	为将来的Pentium而保留
410h	机器检查3控制	Pentium Pro
411h	机器检查3状态	Pentium Pro
412h	机器检查3地址	Pentium Pro
413h	机器检查3多种	为将来的Pentium而保留
1000h	处理器操作寄存器	IBM 386/486SLC
1001h	高速缓存区域控制寄存器	IBM 386/486SLC
1002h	处理器操作寄存器2	IBM 486SLC2
1004h	处理器控制寄存器	IBM 486SBL3

*Pentium Pro手册在某种程度上公开了所有的Pentium Pro MSR，但是这里没有有关这些寄存器的内容的进一步说明。使用CPURDMSR程序来查看任意指定的寄存器的当前内容。

在我的测试中，如果不支持这个寄存器，那么就会触发一个双重错误CPU中断8。

寄存器细节

寄存器0　　机器检查地址　　Pentium，Pentium Pro，Am5k86

机器检查特性用来记录数据读取奇偶性错误或者总线错误。无论发生其中哪种错误，错误地点的64位地址信息就会保存到这个寄存器中。这种错误的其他相关信息保存在型号专用寄存器1中。发生错误时，若开放了机器检查中断，那么就会触发中断12h。尽管实际上允许写寄存器0操作，但是我还没有发现这么做的任何用处，。

Pentium Pro将这个寄存器作为Pentium的兼容寄存器使用。习惯上先检测是否为Pentium Pro，然后使用机器检查指令MSR所提供的详细信息。这些信息开始于400h，这一点在Pentium Pro手册中有详细论述。

寄存器 1　　机器检查类型　　Pentium、Pentium Pro、Am5k86

当机器检查地址中保存了一个错误的地址时，这个寄存器中的字节就是一个错误字节。这个寄存器第 0 位设置为 1，表明这个寄存器和寄存器 0 都有效。当第 0 位是 0 时，这个寄存器的其他位无意义，寄存器 0 也没有意义。

这个寄存器会清除第 0 位，使寄存器无意义。所以比较麻烦的是，CPU 要求先读取并保存机器检查地址寄存器 0，然后读取机器检查地址寄存器 1，看看前面读取的地址是否有效。

我还没有发现写寄存器 1 的任何用处，尽管事实上允许这个操作。

Pentium Pro 将这个寄存器作为 Pentium 的兼容寄存器使用。习惯上先检测是否为 Pentium Pro，然后使用机器检查命令 MSR 所提供的详细信息。这些信息开始于 400h。这一点在 Pentium Pro 手册中有详细论述。

位		
	63 ~ 5=0	保留
	4=1	错误出现时维持 LOCK 硬件线
	3=0	I/O 周期中出现
	1	内存周期中出现
	2=0	数据周期中出现
	1	取代码周期中出现
	1=0	写周期中出现
	1	读周期中出现
	0=0	不进行错误锁存，其他位无意义（参见文中解释）
	1	进行错误锁存

寄存器 2　　奇偶求反寄存器（TR1）　　Pentium

这个寄存器用来测试 Pentium 内部各区域的奇偶逻辑。当对指定的内部存储类型使用这个寄存器时，写逻辑会对奇偶生成位求反，（在测试微指令时是读逻辑），只有在奇偶求反情况下，读与写才是针对相关的对象。必须生成一个奇偶修正位来表明逻辑正常。

奇偶修正位会激活 IERR#引脚，并关闭 CPU。可以将第 1 位设置为 1 来避免关闭 CPU。为了确认生成了奇偶修正位，必须读取这个寄存器的第 0 位（如果奇偶出错，应该是 1）。所有的位都可写，但是只有第 0 位可读。

位		
	63 ~ 14=x	保留（未使用或者是隐藏的位）
	13=1	对读取的微代码奇偶求反
	12=1	对数据 TLB（翻译备用缓冲区）的数据奇偶求反
	11=1	对数据 TLB（翻译备用缓冲区）的标识奇偶求反
	10=1	对数据高速缓存的数据奇偶求反
	9=1	对数据高速缓存的标识奇偶求反
	8=1	对代码 TLB（翻译备用缓冲区）的数据奇偶求反

7=1	对代码TLB（翻译备用缓冲区）的标识奇偶求反
6=1	对代码高速缓存的数据奇偶求反（奇数位 255、253、……、131、129）
5=1	对代码高速缓存的数据奇偶求反（偶数位 254、252、……、130、128）
4=1	对代码高速缓存的数据奇偶求反（奇数位 127、125、……、3、1）
3=1	对代码高速缓存的数据奇偶求反（偶数位 126、124、……、2、0）
2=1	对代码高速缓存的标识奇偶求反
1=0	奇偶修正位设置位0=1，确认IERR#线，并关闭CPU
1	奇偶修正位只设置位0=1，确认IERR#线
0=1	出现了奇偶错误

寄存器4　　结束位测试寄存器（TR2）　　Pentium

寄存器4到7（TR2到TR5）用来测试芯片上的高速缓存。80486上的MOV TRx指令也提供了类似的功能，但是Pentium不包含这些指令。在普通的操作进程中写高速缓存测试寄存器会产生意想不到的结果。在试图测试高速缓存之前，你必须彻底理解高速缓存的相关操作及术语。

为了确保正确的测试结果，必须限制外部查询的周期。要做到这一点，可以设置CR0高速缓存禁止位（第30位）和CR0的非写通位（第29位）。在测试过程中，不可以使用INVD、WBINVD以及INVLPG指令（测试前后可以使用）。

通过测试寄存器写高速缓存的步骤：

1) 禁止外部查询（CR0的第30位和第29位置1）
2) 对于每个4字节通道（记住，一个高速缓存线需要8个4字节通道）：
   a) 将周期的地址写入到TR5（WRMSR寄存器7），TR5的第0位和第1位置为“0”。
   b) 将数据写到TR3（WRMSR寄存器5）
   c) 若是写指令高速缓存，那么还要设置TR2的结束位（WRMSR寄存器4）。
3) 将期望的标签，LRU以及有效位写入到TR2（WRMSR寄存器6）。
4) 装入TR5（WRMSR寄存器7），注意TR5的控制位第0位置“0”而第0位置1，来进行写测试。

通过测试寄存器读Cache入口点的步骤：

1) 禁止外部查询（CR0的第30位和第29位置1）
2) 对于每个4字节通道（记住一个高速缓存线需8个4字节通道）：
   a) 将期望的地址中写入到TR5（WRMSR寄存器7），TR5的第0位置“1”和第1位置“0”。

b) 从 TR3 中读取数据（RDMSR 寄存器 5）。

c) 如果是读指令高速缓存，那么还有读取 TR2 的结束位（RDMSR 寄存器 4）。

d) 从 TR4 中读标签 LRU 以及有效位（RDMSR 寄存器 6）。

使所有代码高速缓存无效的步骤：

写 TR5（WRMSR 寄存器 7），要求 TR5 的 CD 位第 13 位设置“0”，TR5 的控制位第 1 位和第 0 位置“1”。

所有数据高速缓存无效的步骤（不回写修改线）：

写 TR5（WRMSR 寄存器 7），要求 TR5 的 CD 位第 13 位设置“1”，TR5 的 WB 位设置为“0”，TR5 的控制位第 1 位和第 0 位置“1”。

使一个高速缓存线无效的步骤（修改后回写）：

写 TR5（WRMSR 寄存器 7），要求 TR5 的 CD 位第 13 位置“1”，TR5 的 WB 位置“1”，TR5 的控制位第 1 位和第 0 位置“1”，TR5 中也要标明高速缓存线的地址。

结束位测试寄存器（TR2）仅适用于处理指令高速缓存时，第 4 位与 TR3 中的第 4 个字节对应。当 TR3 中的字节是一条指令的最后一个字节时，这一位就置为“1”。这被用来对一个时钟周期内的两条指令进行译码。不正确的结束位值会自动被修正，但是该坏结束位会阻止在一个时钟内执行两条指令。

位	63 ~ 4=x	未使用
	3=1	TR3 的字节 3（位 31~24）保存了一条指令的最后一个字节
	2=1	TR3 的字节 2（位 23~16）保存了一条指令的最后一个字节
	1=1	TR3 的字节 1（位 15~8）保存了一条指令的最后一个字节
	0=1	TR3 的字节 0（位 7~0）保存了一条指令的最后一个字节

寄存器 5　　高速缓存数据测试寄存器（TR3）　　Pentium

这个寄存器含有要写到或读自一条高速缓存线的 4 个字节。对照型号专用寄存器 4（结束位测试寄存器），以了解如何使用这个寄存器。

位	63~32=x	未使用
	31~24=x	字节 3 数据
	23~16=x	字节 2 数据
	15~8=x	字节 1 数据
	7~0=x	字节 0 数据

寄存器6　　高速缓存状态测试寄存器（TR4）　　Pentium

这个寄存器为高速缓存操作保存状态信息。在看型号专用寄存器4（结束位测试寄存器），以了解如何使用这个寄存器。

位					
	63~32=x	未使用			
	31~8=x	标签			
	7~3=x	未使用			
	2=0	指向 WAY 0			
	1	指向 WAY 1			
	1=x	高速缓存线状态			
	0=x	如果是数据高速缓存（TR5 的位 13=1）		如果是代码高速缓存（TR5 的位 13=0）	
		位 1	位 0	位 1	位 0
		0	0=无效的状态	0	0=无效的状态
		0	1=共享状态	0	1=有效状态
		1	0=位于 E 状态	1	0=无效的状态
		1	1=位于 M 状态	1	1=有效状态

寄存器7　　高速缓存状态测试寄存器（TR5）　　Pentium

这个寄存器设置高速缓存地址、类型以及读写控制。参看型号专用寄存器4（结束位测试寄存器），以了解如何使用这个寄存器。

位			
	63~15=x	未使用	
	14=0	高速缓存写通（仅数据高速缓存）	
	1	高速缓存写回（仅数据高速缓存）	
	13=0	代码高速缓存	
	1	数据高速缓存	
	12=0	入口是 WAY 0	
	1	入口是 WAY 1	
	11~5=x	高速缓存地址（选择高速缓存线，128 个 32 字节集中的一个）	
	4~2=x	高速缓存地址（在一条高速缓存线上从八个 4 字节区域中选择一个，通过 TR2 访问这条线）	
	1=x	测试控制	
	0=x	位 1	位 0
		0	0 = 普通操作
		0	1 = 写测试性
		1	0 = 读测试性
		1	1 = 清空高速缓存

寄存器 8　　TLB 命令测试寄存器（TR6）　　Pentium

寄存器 8 和 9（TR6 和 TR7）被用来测试翻译备用缓冲区（TLB）。在普通的操作过程中写任意一个 TLB 测试寄存器，都会带来意想不到的结果。在试图测试 TLB 之前，你必须彻底理解 TLB 的相关操作和术语。表 3-17 列出 Pentium 用到的三个 TLB。

表 3-17　翻译备用缓冲区

TLB 类型	入口地址数	页面大小
数据	64	4KB
数据	8	4MB
代码	32	4KB 或 4MB

通过测试寄存器向 TLB 写入口地址的步骤：

1) 向 TR7 中写入物理地址、相关位，并将第 4 位（选中位）置“1”。
2) 向 TR7 中写入线性地址、相关位、页面大小并将第 0 位置“0”（写 TLB）。

通过测试寄存器从 TLB 读取入口地址的步骤：

1) 向 TR6 中写入线性地址、相关位，页面大小，并将第 0 位置“1”（读 TLB）。
2) 读 TR7，并检查第 4 位，选中位的状态。
   a) 如果选中位是 0，那么读取失败，物理地址无意义。
   b) 如果选中位是 1，那么读取成功，读自 TR7 的翻译过来的物理地址和相关位有效。接着读 TR6 获得 4 位，第 10 位~第 8 位（有效、垃圾、用户以及可写位）。

注意：

作为有效的代码 TLB 测试的一部分，读周期将自动清除 TR6 中的线性地址位，第 31 位~第 12 位。但数据 TLB 测试不会清除这些位。

TLB 命令测试寄存器（TR6）定义了线性地址以及相关的控制及状态位。

位		
	63~32=x	未使用
	31~12=x	线性地址
	11=0	TLB 入口无效
	1	TLB 入口有效
	10=0	没有向相关的内存页写
	1	向相关的内存页写
	9=0	用户可以在任何优先级（0~3）下访问内存页
	1	用户只有在优先级 0 下才可以访问内存页
	8=0	关闭了可写模式，页面只可读

1	页面可写	
7~3=x	保留	
2=x	TLB选择	
1=x	位2	位1
	x	0= 代码TLB（忽略大小位）
	0	1= 数据TLB 4KB页面大小
	1	1= 数据TLB 4KB页面大小
0=0	TLB写	
1	TLB读	

寄存器9　　TLB数据测试寄存器（TR7）　　Pentium

这个寄存器用来测试翻译备用缓冲区（TLB）。详细说明参看寄存器8（TR6）。

位	63~32=x	未使用
	31~12=x	物理地址
	11=x	页面等级高速缓存禁止
	10=x	页面等级写通
	9=x	LRU位2
	8=x	LRU位1
	7=x	LRU位0
	6=x	保留
	5=x	保留
	4=0	前面的读操作错过
	1	前面的读操作命中
	3=x	入口指针（四路之一）
	2=x	
	1=x	保留
	0=x	保留

寄存器Bh　　BTB标签测试寄存器（TR9）　　Pentium

寄存器B到D（TR9到TR11）用来测试分支目标缓冲区（Branch Target Buffer，BTB）。为了确保正确的测试结果，必须禁止分支预测逻辑。先将型号专用寄存器F（TR12）的第0位设置为1，可以做到这一点。在正常的操作期间写任意一个BTB测试寄存器都会得到意想不到的结果。

通过测试寄存器向BTB写入口地址的步骤：

1）禁止分支预测（TR12的第0位置1）。

2）将标签地址和历史写入TR9（WRMSR寄存器0Bh）。

3）将标签地址写入 TR10（WRMSR 寄存器 0Ch）。

4）将选好的值、入口地址以及写测试（第 1 位=0，第 0 位=1）写入 TR11（WRMSR 寄存器 0Dh）。

通过测试寄存器从 BTB 读取入口地址的步骤：

1）将选好的值、入口地址以及读测试（第 1 位=1，第 0 位=0）写入 TR11（WRMSR 寄存器 0Dh）。

2）从 TR9 中读取（RDMSR 寄存器 0Bh）标签地址和历史。

3）从 TR10 中读取（RDMSR 寄存器 0Ch）标签地址。

BTB 标签测试寄存器（TR9）定义了标签地址和历史位。

位		
	63~32=x	未用
	31~6=x	标签地址
	5~2=x	保留
	1~0=x	历史

寄存器 Ch　　BTB 目标测试寄存器（TR10）　　Pentium

这个寄存器用来测试分支目标缓冲区（BTB）。详细说明参见型号专用寄存器 B（TR9）。

位		
	63~32=x	未用
	31~0=x	标签地址

寄存器 Dh　　BTB 目标测试寄存器（TR11）　　Pentium

这个寄存器用来测试分支目标缓冲区（BTB）。详细说明参见型号专用寄存器 B（TR9）。

位		
	63~12=x	未使用
	11~6=x	集选择（从 64 个中选择一个）
	5=x	保留
	4=x	保留
	3~2=x	入口（从四个中选择一个）
	1=x 0=x	控制位

位 1	位 0
0	0 = 普通操作
0	1 = 写测试性
1	0 = 读测试性
1	1 = 清空

寄存器Eh　　BTB目标测试寄存器（TR12）　　Pentium

这个寄存器控制在Pentium CPU中引入的新特性。

位		
	63~10=x	保留（要么未使用，要么是附加的隐藏位）
	9=x	IO陷入重启。作为SMM（系统管理模式）的一部分，重新启动被SMM陷入的I/O通道
	8=0	当允许分支跟踪时（参看下面第1位），触发一个快速分支跟踪信息总线周期（这种特性只适用于较新的Pentium系列）
	1	当允许分支跟踪时（参看下面第1位），生成普通的分支跟踪信息总线周期
	7=x	保留（未公开的功能）
	6=1	禁止自动停机（所有的Pentium系列都不支持）
	5=x	保留（未公开的功能）
	4=1	禁止内部APIC（对于带有集成APIC的Pentium而言）
	3=0	普通的高速缓存
	1	禁止L1高速缓存（内部数据和代码），但是不清除它。它不影响任意存在的L2高速缓存
	2=0	正常的多指令管线操作（允许U和V管线）
	1	禁止V指令管线
	1=1	允许跟踪。这一位允许模式器跟踪所有的分支。当允许跟踪，并且CPU分支时，目标地址就在一个特殊周期内输出到总线上。然后模拟器在其跟踪缓冲区中记录下该地址。这种情况下的分支包括跳转、调用、返回、中断以及其他一些特殊的情形。例如装入段描述符。这种特性只用于测试和模拟器，但程序操作不产生影响
	0=0	允许正常的分支预测
	1	禁止向分支预测缓冲区中加入新的入口地址，已经存在的入口地址将继续被执行

寄存器10h　　时间标志计数器　　Pentium,Pentium Pro,Am5k86

在未公开的RDTSC指令下详细描述了时间标志计数器。这个寄存器提供了一种可替代的方法来读取时间标志计数器，并提供了一种手段来改变这个寄存器的值。

寄存器11h　　运行监控和事件选择　　Pentium

Pentium 处理器提供两个 40 位的计数器，用来对各种内部事件或者一次事件中的时钟数目进行记数。这个寄存器用来控制每个计数器，并指定要监视的事件。计数器保存在寄存器 12h 和 13h 中。在使用之前，必须将它们设置为一已知的值（通常为零）。另外，在切换到一个新事件之前，必须关闭计数器，并重新设置初始值。

为了使用事件计数器来执行监视操作，可以使用时间标志计数器。这时，可对寄存器 10h 使用 RDMSR 指令，或使用 RDTSC 指令。在事件开始之前，应保存时间标志值，在所监视的事件结束之后，也要保存这个值。它们之间的差是一个测量非常精确的时钟周期。

位		
	63~32=0	未使用
	31~26=x	保留（未使用或者隐藏位）
	25=x	计数器 1（在寄存器 13h 中）的 PM1 处理器引脚控制：0=增加计数器时，在外部 PM1 引脚上发出信号；1=计数器溢出时，在外部 PM1 引脚上发出信号
	24=x	计数器 1 控制——事件计数时钟或者事件的数目（参看指定的事件来了解如何设置这一位） 0=对事件的数目计数；1=在事件发生期间对时钟数进行计数
	23=x 22=x	计数器 1 的优先级 位 23 位 22 0 0=禁止计数器 0 1=在位于优先级 0、1 或 2 时计数 1 0=在位于优先级 3 时计数 1 1=在位于任何优先级时计数
	21=x 20=x	计数器 1 要监视的事件（项目的计数都是基于发生的次数的，但是那些标有星号“*”的项目都是对这件事件发生期间的时钟数进行计数）
	19=x 18=x 17=x 16=x	0=读数据 1=写数据 2=数据高速缓存 TLB（翻译备用缓冲区）丢失 3=数据读丢失 4=数据写丢失 5=写命中了数据高速缓存中的 M 或 E 状态 6=数据高速缓存写回 7=外部检测 8=数据高速缓存检测命中 9=同时在两个管线中访问内存 Ah=组冲突 Bh=数据没有对齐的内存或 I/O 参考 Ch=读代码 Dh=代码 TLB（翻译备用缓冲区）丢失

Eh=代码高速缓存丢失
Fh=装入任意段的寄存器
12h=实际的分支（包括跳转、调用、返回、中断等等）
13h=BTB（分支目标缓冲区）命中
14h=取分支或者BTB命中
15h=管线清空
16h=指令执行
17h=指令在V型管上执行
18h=一个总线周期内读时钟*
19h=由于写缓冲区满，所以时钟计数停止*
1Ah=由于等待读取数据内存，所以停止了指令管线*
1Bh=停止了向数据高速缓存内的E或M状态的写操作
1Ch=锁住了总线周期
1Dh=I/O读或写周期
1Eh=内存读操作不可以高速缓存
1Fh=由于地址生成发生了互锁，所有停止了指令管线*
22h=FLOP（浮点操作）
23h=DR0上的断点匹配
24h=DR1上的断点匹配
25h=DR2上的断点匹配
26h=DR3上的断点匹配
27h=硬件中断
28h=数据读或写
29h=数据读丢失或写丢失

15~10=x　保留（未使用或隐藏位）

9=x　计数器0（在寄存器12h中）的PM0处理器引脚控制：0=增加计数器时，在外部PM0引脚上发出信号；1=计数器溢出时，在外部PM0引脚上发出信号

8=x　计数器0控制——事件计数时钟或者事件的数目（参看指定的事件来了解如何设置这一位）：0=对事件的数目计数；1=在事件发生期间对时钟数进行计数

7=x
6=x　计数器0的优先级

位7	位6
0	0=禁止计数器
0	1=在位于优先级0、1或2时计数
1	0=在位于优先级3时计数
1	1=在位于任何优先级时计数

5~0=x　计数器0要监视的事件（参看位21~16了解事件的类型）

*作为一个操作码字节（8个之一），而不是双字。

寄存器 12h　　事件计数器 0　　Pentium

该 40 位的计数器用来对各种内部事件或者一次事件中的时钟数目进行计数。有关事件类型和其他相关细节，参看寄存器 11h。

寄存器 13h　　事件计数器 1　　Pentium

该 40 位的计数器用来对各种内部事件或一次事件中的时钟数目进行计数。有关事件类型和其他相关细节，参看寄存器 11h。

寄存器 82h　　阵列访问寄存器　　Am5k86

该功能允许测试驻留在 Am5k86 中的各种高速缓存内存。它包括 8KB 数据高速缓存,以及 4KB 和 4MB 的两个 TLB 高速缓存。执行后，WRMSR 不改变 EDX 中的测试类型。对寄存器 82h 执行 RDMSR 也不会改变 EDX。因此，读和写数据就不用重新装入 EDX。先用 WRMSR 写入测试数据，然后用 RDMSR 检查数据是否写得正确。

表 3-18 列出了每一种测试类型下的寄存器内容。“Way”代表了测试的列，“Set”代表了测试的行。在某些情况下，还使用 3 个位来定义使用哪个双字（从 8 个中选）。阵列 ID 总是 EDX 的低字节（第 7~0 位）。所有未指出的位必须设置为 0。

表 3-18　列阵寄存器内容

……在 EDX 中……					
所访问的阵列	WAY 位	SET 位	双字位	阵列 ID	在 EAX 中的有效位
数据高速缓存：数据	29~28	18~13	12~10	E0h	31~0
数据高速缓存：线性标签	29~28	18~13	无	E1h	27~0
数据高速缓存：物理标签	29~28	18~13	无	ECh	22~0
指令高速缓存：指令	29~28	19~12	11~9*	E4h	25~0
指令高速缓存：线性标签	29~28	19~12	无	E5h	19~0
指令高速缓存：物理标签	29~28	19~12	无	EDh	20~0
指令高速缓存：有效位	29~28	19~12	无	E6h	18~0
指令高速缓存：分支位	29~28	19~12	无	E7h	18~0
4KB TLB：页面	29~28	12~8	无	E8h	21~0
4KB TLB：线性标签	29~28	12~8	无	E9h	19~0
4MB TLB：页面	29~28**	无	无	EAh	11~0
4MB TLB：线性标签	29~28**	无	无	EBh	14~0

*作为一个操作码字节（8 个之 1），而不是双字；
**作为一个入口（4 个之 1），而不是一条路线。

位	63~32	阵列指针（edx）
	31~0	阵列数据（eax）

寄存器 83h　　硬件配置寄存器　　Am5k86

这个寄存器用来控制高速缓存、分支跟踪以及时钟控制功能。

位	63~8	保留（WRMSR 要将它们设置为 0）
	7=0	开放数据高速缓存
	1	禁止数据高速缓存
	6=0	开放指令高速缓存
	1	禁止指令高速缓存
	5=0	开放分支预测
	1	禁止分支预测
	3~1=x	调试控制（保留的非指定组合位）

位 3	位 2	位 1
0	0	0=禁止调试控制
0	0	1=开放分支跟踪信息
1	0	0=在调试陷入时激活刺探模式

0=0	允许在 HALT 和停止确认状态下停止 CPU 时钟
1	禁止在 HALT 和停止确认状态下停止 CPU 时钟

寄存器 1000h　　处理器操作寄存器　　IBM 386/486SLC

这个寄存器用来控制一系列用户自定义性能，只有 IBM 的芯片才支持这些自定义性能。486SLC 用到第 15 位~第 17 位，而 386SLC 却将它们予以保留。EDX:EAX 指定了如下功能位：

位	63~19	保留，不妨假定没有使用
	18=1	低功率 PLA 模式——在低功率停止状态时，关掉了 CPU 附加动态部件的电源。（386SLC 不支持或未公开）
	17=1	即使开放了高速缓存，也强制所有的数据和代码都读自外存。它用于工厂的 CPU 测试（386SLC 不支持或未公开）
	16=1	将内部高速缓存从奇校验切换到偶校验。这将强制生成一个内部高速缓存奇偶修正位。它用于工厂的 CPU 测试（386SLC 不支持或未公开）。
	15=1	允许浮点操作码读自高速缓存。如果使用了外部 Intel

FPU，则必须将这一位置为零。如果使用的是 Cyrix 的 FPU，设置该位将提高操作码的传送速度。

14=0 允许 ERROR 输入引脚兼容 Intel 的 ERROR。所有的 PC 兼容机都不使用 ERROR 线。Intel 特意设计它来做为数学 FPU 错误的指示信号。

1 将 ERROR 引脚切换到隐存地址选通输出线上。如果主板支持挂起或模拟操作，那么这可用来在挂起或模拟期间访问隐存。参看挂起和恢复部分获得有关隐存的详细信息。这一位只能在硬件重启后第 1 次执行 WRMSR 1000h 时设置。其他时候将忽略对值所做的任何修改以避免 ERROR 输入和选通输出相冲突。

13=1 能低功率停止模式。一个 HALT 指令将使 CPU 停止其内部时钟以节约功耗。

12=1 等待输出后的准备好状态。触了一个输出指令后，处理器将一直等待，直到 CPU 的 READY 线被激活，然后执行下一条指令。它允许执行那些可被关闭电源的设备，这些设备需要额外的时间返回到联机状态。

11=1 高速缓存重新装入状态位——当出现重新装入内部高速缓存的操作时，CPU 会设置这一位。

10=0 内部高速缓存开放与否由外部硬件连线的输入决定。

1 内部高速缓存开放与否由 CPU 决定，同时受到型号专用寄存器 1001h 的限制。它一般用作输入的引脚（当标志位为 0 时），有时也作为输出，用来指示某个内存周期是否是一个高速缓存周期。这一位只能在硬件重启后第 1 次执行 WRMSR 1000h 时设置。在其他任何时候都将忽略对值所做的任何修改。

9=1 禁止高速缓存锁存模式——允许 CPU 识别一个锁存的读-改-写周期，但不允许通过高速缓存实现这个周期。

8=x 保留作未知功能，或未使用。

7=1 开放内部高速缓存——功能类似于 Intel CPU 的 EELAGS 寄存器的高速缓存开放位。

6=1 对 E0000~E0FFFh 的内存区域不使用高速缓存。该 4K 空间用作日文系统的双字节字符与支持（DBCS），当使用 DBCS 时不能使用高速缓存。

5=1 开放功率中断 PWI——允许 PWI 引脚控制挂起模式。参看挂起和恢复模式部分。它也控制保存与恢复的字节数。参看 LOADALL 以获取额外的信息。

4=1	允许清空检测。它用于专用的主板设计。当处理器位于 HOLD 状态以及 CPU 的信号线处于激活状态时，它清洗内部 CPU 高速缓存。只有在第 3 位为零时才可以使用它。
3=1	开放检测输入——当 CPU 因硬件原因而处于 HOLD 状态时，CPU 仍然能够监视总线上的数据。如果因其他 CPU 或设备而造成的写内存地址也存在于高速缓存中，那么高速缓存中的项就会失效。
2=1	开放 A20 屏蔽——AT+系统必须包括一些逻辑，通常是 CPU 的外部逻辑，来屏蔽 CPU 地址线 A20。这一点用来模拟 8088 寻址空间。IBM CPU 包含一个 CPU 内部逻辑来完成同样的功能。当这一位开放时，将一直禁止地址线 A20，除非激活分页（在 CR0 的第 31 位中设是分页）特性。当关闭 A20 屏蔽位时，控制要么是外部的，要么是全部的可访问地址区域。
1=1	高速缓存奇偶校验开放——开放 CPU 内部高速缓存的奇偶校验。如果出现了一个奇偶错误，就会清洗高速缓存，并禁止高速缓存（第 7 位置 0），设置奇偶错误标志位（第 0 位置 1），并禁止奇偶校验（第 1 位置 0）。然后，调用 NMI 处理程序来处理奇偶错误。
0=1	奇偶错误发生在内部高速缓存存储器中——写零以清除这个标志位，参见第 1 位以获得详细信息。即使是禁止了奇偶校验，奇偶校验也会设置该标志位，但是这时不会采取其他行动。

寄存器 1001h　　高速缓存区域控制寄存器　　IBM386/486SLC

当允许使用高速缓存时，可以设置这个寄存器来允许物理内存指定的区域使用高速缓存读取。高速缓存也可用于任意 ROM 存储区域，只是高速缓存会忽略写 ROM 空间的努力。

一般情况下，由 BIOS POST 操作基于系统中内存的总数和类型来设置这些寄存器。EDX:EAX 中的 64 位内容如下所示：

位	63~40	保留
	39~32	扩展内存高速缓存限——指定了连续的 64K 块的数目，这些 64K 块开始于 1MB 边界，并且可以使用高速缓存。例如，值 0Fh 表示有 15M 字节可以使用高速缓存。
	31~16	只读高速缓存块——每一位都代表了一个 64K 的内存区域，这些内存位于第 1M 字节的内存空间，该

		空间含有 ROM 存储器。设置 31 位表示段 F000h 处的最后 64K 是 ROM，而设置 30 位表示段 E000h 处的 64K 是 ROM，依此类推。向设置为 ROM 的区域写不会更新高速缓存存储器。
	15~0	首兆字节可以使用高速缓存——每位代表一个 64K 的首兆字节内存区域可以使用高速缓存。设置第 15 位表示段 F000h 处的最后 64K 可以使用高速缓存，而设置第 0 位表示第 1 个 64K 可以使用高速缓存。

寄存器 1002h　　处理器操作寄存器 2　　IBM 486SLC2

这个寄存器控制不同的时钟模式，以使内部 CPU 时钟运行速度是外部时钟的两倍。它也可以改变外部时钟的频率。

因为内部时钟的运行频率可能是外部时钟频率的两倍，在 turbo 模式下并不能马上改变这个寄存器。你必须告诉 CPU 你请求改变时钟频率。可以设置一个软件位或者从芯片外部引脚来提出该要求。当 CPU 接受改变后的时钟速度时，它会在硬件线上作出相应的反映。一旦改变了输入时钟的频率，就必须删除相应的软件或硬件要求，这样 CPU 可以恢复到普通的时钟状态。

位	63~30	保留		
	29=1	开放外部动态频率交换——该选项允许主板控制 CPU 时钟频率交换。只有系统的 BIOS 才可以控制该功能，因为它依赖于主板设计。		
	28=1	动态频率交换准备好——当通过硬件或设置第 27 位来请求交换频率时，或者当 CPU 准备好频率交换时，就会设置该标志。		
	27=1	动态频率请求——将这一位设置为 1 将会提出请求，要求 CPU 准备改变输入的频率。如果设置了第 28 位，就会改变输入时钟。		
	26=x	时钟模式		
	25=x	第 26 位	第 25 位	第 24 位
		0	0	0 = 将输入时钟除以 2（与 386SX 相同）
	24=x	0	1	1 = 使用输入时钟，不执行除法，以获得双倍的 CPU 速度
		1	0	0 = 3:1 时钟模式（尚未证实）
	23~0	保留		

寄存器 1004h　　处理器控制寄存器　　IBM 486 SBL3

这个寄存器控制 IBM 486L3，蓝光 CPU 的各种选项（尚未证实）。

位	63~24	保留
	23=0	DD1 硬件
	1	DD0 硬件（对于 OS/2）
	22=0	MOV CR0 为 DD0、DD1A、DD1B 和 DD1D 硬件译码
	1	MOR CR0 为 DD1C 硬件译码
	21=x	未知
	20=0	高速缓存保持通
	1	不使用时关闭高速缓存（低功率模式）
	19=x	未知
	18=0	NOP 指令周期（DD0 用两个周期，DD1 用三个周期）
	1	NOP 指令周期（DD0 用三个周期，DD1 用两个周期）
	17~0=x	未知

指令	描述	处理器
XBTS	读出位流	Intel 80386 A step
0Fh,A6h,r/m	xbts	regW,regW/memW,ax,cl

取出位流指令从第 1 个操作数中取出一串位并放入第 2 个操作数中。

这条指令只存在于第一代 80386 上，即 A step。接下来的 CPU 都删除了这条指令，以便给其他的位指令提供空间。其他所有的竞争者都没有这条指令，因为它代表了一种很古老也很古怪的 80386 芯片。

早期的 80486 的 CMPXCHG 指令和它有相同的操作码。据发现，一些写出 80486 之前的指令先检测 XBTS 指令，然后错误地把 80486 当作了古老的 80386 A step 系列。所以在 Intel 80486 B step 上，分给了 CMPXCHG 指令一个新的操作码。而实际应用的结果是，大多数程序员发现区分上述问题相当麻烦，于是就干脆不使用 CMPSCHG 指令。请参看本章前面的 IBTS 指令。

这条指令采用字格式还是双字格式，依赖于当前的模式（是 16 位还是 32 位），以及是否使用了大小覆盖前缀。

隐藏的地址空间

某些 386 版本以及后来的 CPU 提供了一些特性来支持挂起模式和支持内电路模拟。这种特性的获得，部分是由于使用了隐藏的或“替代”的内存，在通常情况下程序不可访问这些内存。只有当 CPU 支持隐存以及主板上的硬件支持隐存时，才可以使用这种特性。支持这种特性的 CPU 有 IBM 生产的 CPU（386SLC 与 486SLC），Intel 的 80386SL、80486SL

和 Pentium 系列以及一些 AMD、Cyrix 和 TI 产品。记住，许多 IBM 计算机使用非 IBM 生产的 CPU，但是这些 CPU 并不支持这种特性。另外，IBM 也向其他供应商提供 IBM 主板，其中包括了 IBM 生产的 CPU。

大多数情况下，有三条尚未公开的指令用来挂起和模拟操作，它们包括内电路模拟器断点（In-Circuit-Emulator breakpoint，ICEBP）、用户移动寄存器（User Move Register，UMOV）以及装入所有寄存器（LOADALL）。另外，读写型号专用寄存器指令（RDMSR 和 WRMSR）也可以控制这种特性，还包括 RSM 指令。前面的章节中对这些内幕指令有详细的阐释。

挂起与恢复模式

通过按下某个按钮或关闭某个按钮，许多膝上型电脑能够执行挂起操作。一般情况下会保存所有的寄存器状态，并降低 CPU 的功率。不久，又会提高 CPU 的功率并恢复所有的寄存器。操作系统和应用程序从挂起的地方继续执行，完全不理会中断。

早期的处理器，比如 80286 和 80386 DX/SX，在生成可靠的挂起和恢复特性时，会产生严重的问题。由于某些寄存器是不可访问的，所以全部捕获 CPU 的状态很成问题。把挂起操作当作不希望的“挂起系统”的现象相当普遍。

四个主要的 80x86CPU 制造商，AMD、Cyrix、IBM 以及 Intel 提供了各自不同的方案来解决这个问题。表 3-19 总结了各供应商解决挂起和恢复操作的不同之处。从它不难看出，可以激活系统管理模式（SMM）来挂起和恢复，并且列出了激活 SMM 时所保存的字节数。

表 3-19　系统管理设计

	由　谁　激　活：				
供应商类型	硬件	软件	保存的字节数	隐存保存区	隐藏 SMM 代码区
AMD 368SXLC	是	SMI	228	6000:0h	FFFFFFF0h（重启）
386DXLC	是	SMI	228	6000:0h	FFFFFFF0h（重启）
486DXLC	是	SMI	364	6000:0h	FFFFFFF0h（重启）
486DX2	是	否	512***	3000:FE00h	3000：8000h
486DX4	是	否	512***	3000:FE00h	3000：8000h
Am5k86	是	否	512***	3000:FE00h	3000：8000h
Cyrix 486SLC	是	HALT	无	无	无
486DLC	是	HALT	无	无	无
486SLC/e*	是	否	35	用户定义	用户定义

续表

由谁激活：					
供应商类型	硬件	软件	保存的字节数	隐存保存区	隐藏SMM代码区
6x86	是	SMINT	48	用户定义	用户定义
IBM 386SLC	是	ICEBP	284	6000：0h	FFFFFFF0h（重启）
486SLC	是	ICEBP	284	6000：0h	FFFFFFF0h（重启）
Intel 386SL	是	时钟**	512***	3000:FE00h	3000：8000h
486SL	是	时钟**	512**	3000:FE00h	3000：8000h
Pentium	是	否	512***	3000:FE00h	3000：8000h
Pentium Pro	是	否	512***	3000:FE00h	3000：8000h

*该系列的A step类型不支持SMM。

**82360SL配套芯片提供了一个时钟，可以在一个软件设置时钟周期激活系统管理模式。

***供应商只是指出了保存的最大可能的字节数。

挂起和恢复特性会造成一个难以解决的问题。可以在执行一个重复串指令中激活这种特性。最糟糕的情况是在执行REP INS过程中激活这种特性，而这时正在进行输入传送。每个制造商都有方案来解决这个问题，尽管它要求大量的软件来使其可靠地工作。

下面一部分将介绍各供应商是如何实现挂起和恢复的。许多供应商也对这种特性使用其他的称谓，比如系统管理模式或者功率管理中断。你需要供应商提供该模式的详细说明，和获得使用该模式的全部细节。许多情况下，供应商的IC数据手册中并没有挂起和恢复的详细说明，但是一些分散的档案中对某些操作还是有详细的说明。它们一般都列举在目录文献中。

AMD 386SXLC、386DXLC、486DXLC

与CPU相连的信号中有一条特殊的系统管理中断（System Management Interrupt，SMI）线，它指示CPU输入挂起特性下的代码。SMI是一个不可屏蔽中断，它的优先级比任何其他的中断都要高，包括中断NMI。另外，一条特殊的指令SMI可以启动相同的操作。

当激活SMI后，CPU会采取一系列不同于其他任意中断的措施。首先，它在开始于6000:0h的隐存中保存所有的CPU寄存器信息。隐存只有在这种状态下才可以访问，并且和相同地址的普通用户内存共存。主板必须设计成支持隐存形式。图3-4给出了隐存的层次图。隐存的某些部分可能会是RAM，但是降低功率的挂起代码一定是ROM形式。取决于具体的应用，只有很少一部分隐藏地址空间有实际的内存。

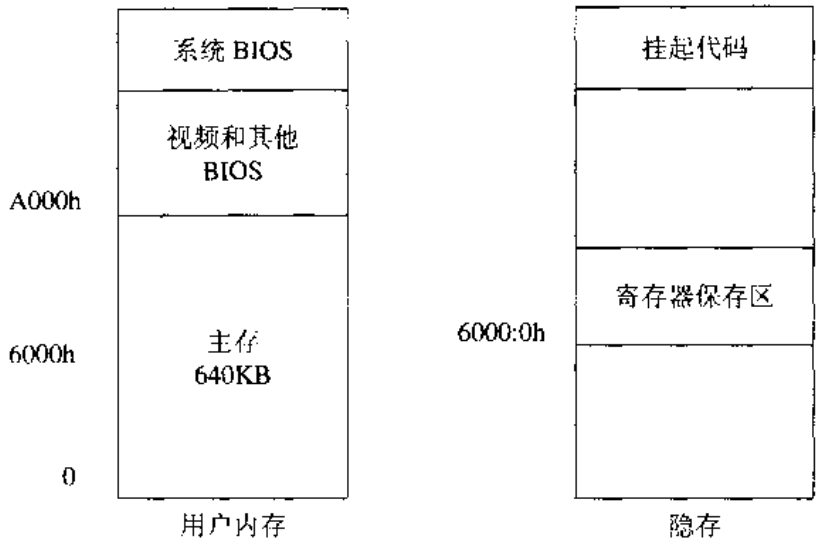


图 3-4 AMD486DXLC 的隐存

一旦保存了所有的寄存器，CPU 会在功率恢复时恢复几乎所有的寄存器。如果 CPU 有内部高速缓存内存，激活 SMI 不会改变 CPU 的高速缓存内容。即使以前处于保护模式或虚 86 形式，CPU 也会被强制到实模式。NMI 期间禁止所有中断。

在 16 位实模式下，CPU 然后从隐存 FFFFF0h 处开始执行。可以使用覆盖来执行 32 位代码。执行隐存中的代码时，不使用内部高速缓存，所以不会改变高速缓存内容。低功率的挂起代码然后执行一些额外的必要操作，例如写 I/O 端口来关闭各种外设的电源。在激活隐存后如果需要从用户内存中写或读数据，可能会使用 UMOV 指令。

如果要恢复运行，就必须执行 LOADALL 指令，AMD 将它称为 RES4 指令，但是它和 386 的 LOADALL 实现相同的功能并且有相同的操作码。将 ES:EDI 设置为 6000:0h,然后执行 LOADALL，这时就可以从出现 SMI 的地点恢复运行。

Cyrix 和 TI 的 486 SLC、486 DLC

从一个软件的视角来看，Cyrix/TI 类型设计相当简单。它们不需要任何特殊的软件。Cyrix/TI 类型带有一个硬件控制的低功率模式。激活后，CPU 会先完成当前的指令。如果 CPU 带有数学协处理器，那么 CPU 会等待协处理器完成其最后一条指令。然后 CPU 进入低功率模式，并保持所有寄存器内容有效。这时，硬件可以停止时钟。如果处于备用模式下，CPU 就会使用 100 微安电流：该电流足够小，几乎可以忽略的。大多数电池可以为 CPU 供电很长一段时间。

如果要从硬件激活的挂起状态中恢复过来，可以开启时钟并释放 CPU 的挂起线。然后 CPU 从激活挂起的地方恢复运行。

触发 HLT（停止）指令，也可以从软件中进入挂起模式。当软件活动很少或根本没有活动，以及需要省功时，这条指令就显得非常有用了。HLT 指令使 CPU 进入备用模式，而

这时仍然开着 CPU 时钟。这时，CPU 需要的功耗大约为 5 毫安。

触发 HALT 指令后，CPU 处于备用模式，这时一条 NMI 指令或任意的真正的硬件中断都会使 CPU 恢复运行。CPU 将进入上次中断的中断服务处理程序中开始执行。

Cyrix 和 TI 的 486 SLC/e

这种 486 新系列提供了一种更常规的挂起和恢复特性，但同时也包括一些有趣的花样。同其他供应商一样，触发一个 SMI 中断引脚可以引发系统管理模式。该系统没有触发 SMM 的软件方法。触发之后，只保存起来很少一部分的 CPU 寄存器。这意味着，从 SMM 入口到出口的时间大约只有其他 CPU 的五分之一。这一点对某些特殊用途会有用，但是对于标准的挂起和恢复特性来说并不很重要。

与其他供应商不同的是，保存 CPU 寄存器的地址和 SMM 代码空间可以由软件定义。这就提供了独特的灵活性。分配给 SMM 的空间可以自由定义为 4KB~32MB。

在系统管理模式下，有必要保存另外一些寄存器。Cyrix 定义了许多独特的指令来保存其他有必要保存的寄存器。当 CPU 的优先级为 0，且 CPU 的 SMI 引脚为激活低电平时，才可以使用这些额外的指令。如果 SMI 未被激活，而 CPU 在 0 优先级下，同时设置了配置寄存器 1 的 SMAC 位，那么这条指令仍然有效。只有 Cyrix/TI 类型才提供这些配置寄存器，可以通过端口 22h 和 23h 访问它们。如果上述条件不满足，那么这些新指令将会触发坏操作码中断。表 3-20 归纳了这些新指令。在 SMM 程序员指南中，Cyrix 详细说明了这些指令。

表 3-20 Cyrix/TI 的特殊指令

基础指令	操作码	描述
RSDC	0F79	恢复段寄存器以及描述符
RSLDT	0F7B	恢复局域描述符表寄存器及其描述符
RSM	0FAA	从系统管理模式下恢复（与 Intel 同）
RSTS	0F7D	恢复任务状态寄存器及其描述符
SVDC	0F78	保存段及其描述符
SVLDT	0F7A	保存局域描述符表寄存器及其描述符
SVTS	0F7C	保存任务状态寄存器及其描述符

为了从 SMM 下恢复正常操作，程序必须首先使用标准指令和表 3-20 列出的特殊指令来恢复任意一个改变后的寄存器。然后，执行 RSM 指令来恢复在进行系统管理模式之前被保存起来的寄存器。执行完 RSM 后，将从出现 SMI 的地方恢复执行。

Cyrix 6x86

6x86 非常类似于以前谈到的 486SLC/e。主要的不同在于 6x86 加入了 SMINT 指令来从软件中触发 SMM。在触发 SMM 时也会保存另外一些寄存器。

IBM 386SLC 和 486SLC

CPU 有一条到其内部的特殊功率中断（PWI）线，来标志 CPU 进入挂起状态。PWI 是一条特殊的真正的中断，其优先级高于包括 NMI 在内的所有的其他中断。设置型号专用寄存器 1000h（参看 WRMSR 指令）的第 5 位开放 PWI 位，可以开放这种特性。

激活 PWI 时，CPU 执行一些不同于其他中断的操作。首先，它将所有的 CPU 寄存器保存到开始于 6000:0h 的隐存中。该操作酷似早期的 486 AMD 类型。在此强调只有在这种特殊的状态下才可以访问隐存，并且隐存同普通用户内存以相同的地址共存。主板必须支持隐存。图 3-5 列出了隐存的层次结构。某些隐存可能是 RAM，但是低功率挂起代码一定是 ROM 形式的。依赖于基本操作，只有很少一部分隐存地址空间有实际的内存。

一旦保存了所有寄存器之后，CPU 会在下次恢复功率状态时恢复所有的寄存器。PWI 模式不改变 CPU 高速缓存内存及型号专用寄存器的内容。这意味着 CPU 将进入实模式，即使以前是位于保护模式或 V86 模式。禁止包括 NMI 在内的所有中断。

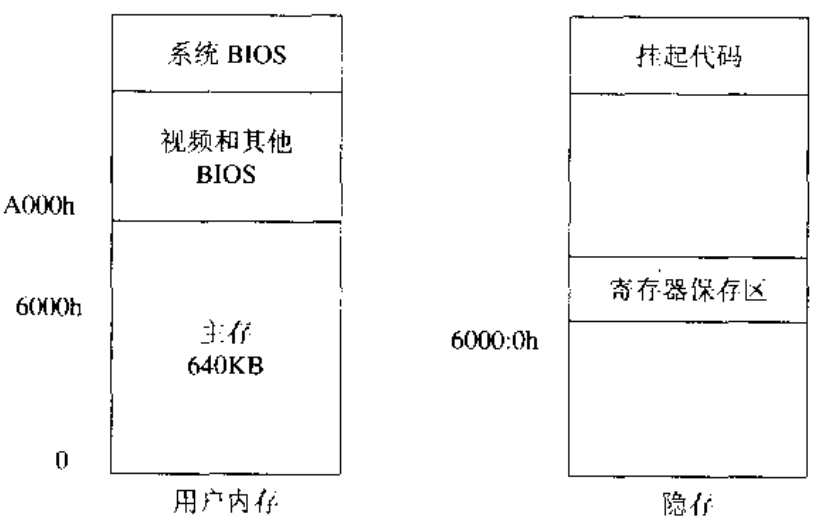


图 3-5　IBM 386 上的隐存

然后，在 16 位实模式下，CPU 从 FFFFF0h 的隐存处开始执行。仍然可以使用覆盖来执行 32 位代码。执行隐存中的代码时，不使用内部高速缓存，所以不会改变高速缓存的内容。低功率挂起代码然后执行一些额外必要的操作，例如写 I/O 端口来关闭各种外设的电源。激活隐存后如果需要从用户内存中写或读数据，可以使用 UMOV 指令。

掉电下要恢复运行，可以重启CPU，以便让BIOS接管对用户内存的控制。系统BIOS必须检查重启是否来自于以前的挂起操作。如果是，处理器会开放PWI（型号专用寄存器的第5位），将ES:EDI设置为6000:0h，然后使用LOADALL指令。这条指令会从隐存中取出所有的寄存器，并将它们装入到CPU中。完成LOADALL之后，系统将从发生PWI的地点恢复执行。

Intel 80386SL、80486SL、Pentium、Pentium Pro、AMD 5k86

IBM和Intel/AMD有类似的设计，但也有很大的区别。一条系统管理中断（SMI）线指示CPU进入挂起状态。SMI是一条特殊的真正的中断，其优先级高于包括NMI在内的所有其他中断。

激活SWI时，CPU执行一系列不同于其他中断的操作。首先，它将所有的CPU寄存器保存到开始于3000:FE00h的隐存中。只有在这种特殊的状态下才可以访问隐存，并且隐存同普通用户内存以相同的地址共存。主板必须支持隐存。图3-6列出了隐存的层次结构。当激活了隐藏模式后，通常只有开始于3000:8000h的32K内存才是RAM。

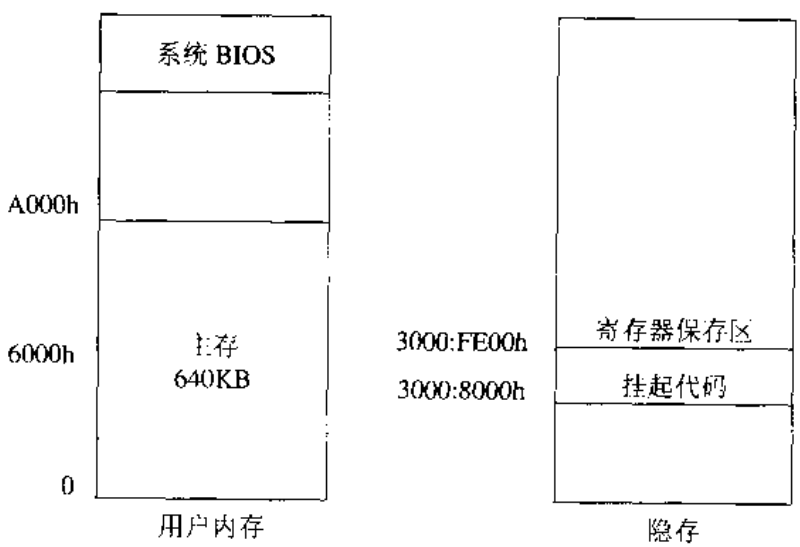


图3-6 Intel CPU中的隐存

表3-21列出了装入隐存的寄存器。如果需要，可以修改大多数寄存器，但是一些保留地址不得修改，否则处理器会出故障。保留内存包括供将来使用的空间以及供应商不想公布的一些寄存器。保存在保留区域中的专用寄存器包括CR1、CR2、CE4和所有的描述符。没有保存的寄存器包括DR0~DR7以及所有的浮点寄存器STn、FCS、FSW、标签字、FIP和浮点操作码及操作数指针。

表 3-21 保存在隐存中的寄存器

内存偏移量	字 数	装 入 的 寄 存 器
FE00	124	保留
FEF8	2	状态清空的基地址——它保存着挂起代码段和这个标的保持段的内部寄存器值。缺省时这个值是 3000h。如果改变了这个值，那么它必须对齐 32K（800h）边界。第一次执行 RSM 时，会将这个值装入到 CPU 的内部清空基地址寄存器中。后面的 SMI 中断寄存器保存和挂起代码的执行要用到这个新的基段址
FEFC	1	系统管理模式位 位 0=1 CPU 支持 I/O 陷入重启 位 1=1 CPU 支持改变状态清空基地址
FEFE	1	系统管理模式的改进版号
FF00	1	I/O 陷入重启——在保存时总将它设置为零。参看 RSM 指令了解如果挂起代码将它设置为 FFFFh 时会发生什么动作
FF02	1	暂停后自动重启——在 CPU 执行 HALT 指令时，如果出现了 SMI，就会设置这一位。否则，就将它设置为零。参看 RSM 指令了解如果这个字的状态不同会有那些动作
FF04	66	保留
FF88	2	GDT 基地址
FF8C	4	保留
FF94	2	IDT 基地址
FF98	8	保留
FFA8	1	ES
FFAA	1	保留
FFAC	1	CS
FFAE	1	保留
FFB0	1	SS
FFB2	1	保留
FFB4	1	DS
FFB6	1	保留
FFB8	1	FS
FFBA	1	保留
FFBC	1	GS
FFBE	1	保留

续表

内存偏移量	字 数	装 入 的 寄 存 器
FFC0	1	LDTR——局部描述符表寄存器
FFC2	1	保留
FFC4	1	TR——任务寄存器
FFC6	1	保留
FFC8	2	DR7
FFCC	2	DT6
FFD0	2	EAX
FFD4	2	ECX
FFD8	2	EDX
FFDC	2	EBX
FFE0	2	ESP
FFE4	2	EBP
FFE8	2	ESI
FFEC	2	EDI
FFF0	2	EIP
FFF4	2	EFLAGS
FFF8	2	CR3
FFFC	2	CR0

注：上述偏移量与 Intel Pentium 手册中的一致，但与 Pentium Pro 手册中的不一致。Intel Pentium Pro 手册上的偏移量信息自相矛盾，在一个地方与 Pentium 一致，但在同一页中其他地方又给出不同的偏移量。

一旦保存了所有的寄存器后，CPU 将在下一次恢复功率状态时恢复所有的寄存器。SMI 模式不改变 CPU 的高速缓存内存。SMI 模式也不影响 Pentium 的型号专用寄存器。如果在 SMM 模式中开放高速缓存，那么可能会改变高速缓存的内容。CPU 将工作在实模式下，即使以前位于保护模式或 V86 模式。

然后，CPU 从隐存 3000:8000h 处开始执行。如果必要，挂起代码可以执行一些额外的操作，例如减少或停止 CPU 时钟以节省功率。

系统管理模式不改变浮点器件的状态。由于没有自动保存，所以在挂起模式下应避免使用浮点指令。

为了从系统管理模式下恢复执行，可以使用 RSM 指令。这条指令将以前保存于 3000:FF00h 处的所有寄存器恢复到 CPU 中。只有在系统管理模式下才可以使用 RSM 指令。执行完 RSM 后，将从用户内存中恢复正常的运行。其他细节请看 RSM 指令。

内电路模拟

进行模拟操作时，会出现一个特殊的断点，并将 CPU 的所有状态保存到内存中。于是，模拟器可以显示断点处的 CPU 状态。为了恢复断点处的指令，模拟器会恢复源指令，并重新装入被保存的 CPU 状态。

模拟类似于上面谈到的 Intel 和 IBM 的挂起和恢复模式。在这种情况，隐存中保存的是 ICE 代码而不是挂起代码。更多细节参看 UMOV、LOADALL 以及 ICEBP 指令。

CPU 重启

通过激活硬件重启线，CPU 被设置为初次上电状态。这条硬件重启线在开启电源时被触发，在某些系统中有一个重启按钮也可以激活它。在 AT 以及后来的机器上，键盘控制器也可以产生硬件重启（参看第 8 章，键盘系统）。

重启时，CPU 将自己设置为一个已知状态，实模式，并且禁止中断，以及将关键的寄存器设置为一个已知状态。如果在重启期间系统没有掉电，硬件重启不改变系统内存的内容。在 386 以及后来的处理器上，会运行自检程序。如果自检获得通过，AX 会返回 0。DH 会设置为家族系列类型，数字 3 代表 386，4 代表 486，5 代表 Pentium，6 代表 Pentium Pro。某些供应商会在 DH 的上半部分返回一个数，来表示子系列，比如 386SL。DL 中将保存版本号，通常指芯片是第几代的。

重启会使处理器跳到系统 BIOS 并开始执行代码。在 8088 上该地址是 F000:FFF0。对于 80286 以及后来的处理器来说，该地址位于可寻址内存的顶部，至少有 16 字节。在一个 32 位地址总线的 486 上，物理地址将是 FFFFFFF0h。主板通常可以对系统 BIOS 进行双重映射，可以是映射到寻址范围的顶部，也可以是 1MB 区域的顶部。访问 F000:FFF0 实际上就是访问物理地址 FFFFFFF0h。

现在，286+ CPU 跨出了非凡的一步。当第一次远跳转或远调用时，CPU 将仅限于对首兆字节内的内存进行访问。大多数系统 BIOS 生产商在 FFFFFFF0h 处带有一个远跳转，以开始 F000 段内的初始化代码。这将指示 CPU 将高端地址线设置为 0。在这一点上系统酷似早期的 8088，仅限于 1MB 的寻址空间。系统 BIOS 将负责完成设置寄存器、描述符，以及其他一些操作要求，来匹配正在运行的系统。

系统 BIOS 获得控制之后，AT+系统会从 CMOS 读取关闭寄存器字节 Fh。这个字节告诉系统 BIOS 进一步作什么。可能仅仅只是处理一下普通的 BIOS 初始化，或者直接跳到引导自举程序，或者跳到预先设定的程序中。如果提供了这种控制，80286 就可以从保护模式下切换回实模式。CMOS 关闭寄存器 Fh 将在第 15 章（CMOS 内存与实时时钟）中讨论。

重启 CPU

似乎有许多方法可以从软件重启 CPU，但是只有少数几种方法比较可靠。有一些方法可以在一台机器上可行，但在另一台机器上无效。

错误认识#1：跳到引导自举向量中 BIOS 有一个中断可以从硬盘或软驱装入引导自举装入程序，只要触发引导自举中断向量 19h 就可以做到这一点。应用程序永远也不应该使用这种方法。这可以绕过这部分 BIOS，所以它似乎是一种重新启动系统的快速方法。真正出错的地方是，TSR 和设备驱动程序会挂起一系列中断。系统 BIOS 可以从驱动器向固定地址 0:7C00h 中装入引导自举程序，而根本不管是否有 TSR 或驱动文件仍在相应的地址运行。一旦运行引导自举程序，它会向内存中装入很多东西，这可能会破坏其他的 TSR 和驱动文件。这时，许多中断向量会指向一些内存中不再存在的 TSR 和驱动文件！当触发指向不存在的 TSR 和驱动文件的中断时，系统会发生冲突。

错误认识#2：跳到重启开始地址 F000:FFFF 当处理器执行硬件重启时，CPU 从 F000:FFFF 处开始执行。自然会想到，要跳到 F000:FFFF 似乎就可以进行系统 BIOS 的初始化。大多数情况，这一点在 8088 上运行良好，对于后来的 CPU 而言，在某些情况下仍然可行，但是它也极有可能会失败并挂起系统。设计 BIOS 时，都假定 CPU 完全重启，实模式，并且重要的寄存器处于已知状态。直接跳到系统 BIOS 并不保证上述前提。系统可能处于虚 86 模式，调试程序的断点可能仍处于激活状态，中断可能仍处于开放状态，以及其他的一系列潜在问题会挂起系统 BIOS。

重启的三种方法 要实现 8088 PC 上的硬件重启，唯一的方法是跳到地址 F000:FFF0。正如前面所指出的，这种方法不可以用于带有 286 或后来处理器的系统。

一个 AT 系统及其更晚的系列，可以实现软件控制下的处理器重启。向键盘控制器微控制器发送一条命令，来激活重启线，可以做到这一点。第 8 章，键盘系统，提供了代码例 8-4，来显示如何实现这一点。

在 MCA 以及某些 EISA 系统上，提供了一种更快捷的方法。系统控制端口定义了一位用来快速重启，而不用通过键盘控制器。这种重启更快，在从保护模式切换到实模式时应该采用这种方法。其代码如下：

```
in      al, 92h          ; 获得系统控制端口
IODELAY
or      al, 1            ; 设置位 0，来快速重启
out     92h, al          ; 执行
hlt                      ; 这里不应该使用 halt
```

你可以回顾一下系统控制端口（92h）的其他各位，它们可以用于其他功能，比如禁止

A20 地址线。

热启动和冷启动 当系统首次上电时，BIOS 会检测 RAM 中的某个值，来辨别系统是处于热启动状态（即：已经上电）还是处于冷启动状态。大多数 BIOS 会在冷启动状态下测试所有的内存，并且采取一些必要的措施。热启动状态意味着不必测试内存，系统启动会更快。

当你按下 Ctrl-Alt-Del 时，BIOS 会将热启动的 RAM 值设置到 1234h 中，以指示这是一次热启动。表 3-22 中列出了一些系统所用到的指示值。热启标志驻留于 BIOS 40:72h 的字中。如果通过软件重启，也许将热启标志设置为相应的启动类型。冷启动，必须将热启标志清零。

表 3-22 启动标志值

值	功 能
0000h	冷启——检查内存，所有内存内容丢失
1234h	热启——忽略内存检查，所有内存内容可能丢失
4321h	热启——忽略内存检查，RAM 保持不变
0064h	BIOS 循环测试（某些供应商提供）

重启之前捕获控制

某些情况下，常常很希望能在重启之前捕获控制。例如，一个磁盘高速缓存在 RAM 中存有信息但还未写回磁盘，在重启之前，磁盘高速缓存程序需要清洗这些缓冲区。如果不更新它们，磁盘文件，甚至是磁盘结构本身，会不完整。

在 8088 系统中，禁止重启唯一的方法是替换键盘 BIOS 中断 9h。由于各供应商提供的系统各不相同，做到这一点很难。如果替换了中断 9，那么也将禁止组合 Ctrl-Alt-Del（重启组合）。

AT 以及后来的系统都为 Ctrl-Alt-Del 命令提供了一个断点。这种情况下，中断 15h 作为一个特殊的处理程序处理。该处理程序会搜索 AH。AH 这时被设置为 4Fh，表示按下了某个键，但键盘 BIOS 的中断 9 还未处理它。如果找到是 4Fh，就会检查进位标志以确定其被设置好。如果还未设置进位标志，该键就会被忽略，也没有必要作进一步的检查。如果设置了进位标志，就会检查 AL 寄存器，看看它是不是 DEL 的扫描码 71h。如果是，接着会检查 40:17h 处的 BIOS 键盘标志，看看现在是否按下了 Ctrl 和 Alt 键。如果按下了 Ctrl 和 Alt 键，那么第 2 位和第 3 位都会被设置为 1。这时，用户一定是按下了 Ctrl-Alt-Del，并且在重启之前将采取所有期望的措施。

从 15h 中断返回时，注意不要改变 AX，并且保持进位标志的设置。这样，其他处理程序也可以使用 15h 中断来执行必要的操作。这时，键盘处理程序重新从中断 15h 获得控

制，然后会重新启动。

该机制不保护下面的重启：直接向键盘控制器或系统控制端口写来实现重启。如果程序运行在保护模式下，例如 Windows 或者内存管理器同时使用的是 386 或更晚的 CPU，那么就有可能会陷入 I/O 端口 60h，64h 和 92h。这些端口用来监视是否有人试图按这种方法重启系统。如果将要发生重启，就必须采取必要的措施。

代码例 3-5 是某程序的一部分，它说明了中断 15h 处理程序是如何探获 Ctrl-Alt-Del 的。前提是中断 15h 已经被指向了 int_15h_hook，并且以前的中断 15h 指针保存在了 old_int_15h 中。

代码例 3-5 驻留的重启处理程序

```
;--------------------------------------------------------------------------------
; 驻留的处理程序，检测是否按下了 CTRL-ALT-DEL

int_15h_hook    proc    far
        pushf
        push    ax
        push    ds
        cmp     ah, 4Fh                 ; 中断 9 键盘功能?
        jne     skip_finished;          ; 如果不是，则跳转

        cmp     al, 71h                 ; 按下了 delete 键?
        jne     skip_finished;          ; 如果不是，则跳转
        mov     ax, 40h
        mov     ds, ax                  ; 指向 BIOS 数据区
        test    byte ptr ds:[17h],4     ; 按下了 ALT 键?
        jz      skip_finished;          ; 如果不是，则跳转
        test    byte ptr ds:[17h],2     ; 按下了 Ctrl 键?
        jz      skip_finished;          ; 如果不是，则跳转

; 用户按下了 CTRL-ALT-DEL
   <<<在这里放用户代码>>>

; 完成，现在给其他的挂起了中断 15h 的 TSR 和驱动程序一个机会，
; 来检测 CTRL-ALT-DEL
```

```
skip_finished:
        pop     ds
        pop     ax
        popf
        jmp     dword ptr cs:old_int_15h;处理老式的 int_15h

int_15h_hook    endp

old_int_15h     dd      0               ; 在这里保存以前的指针
```

第 4 章

系统与设备检测

本章将告诉你如何检测一系列系统的类型，并列举了 CPU 的详细信息。据我所知在《技术内幕》之前，还没人发布过这些信息。检测功能很容易作为一个子程序插入到其他程序中去。我举出了许多例子来说明每个功能是如何工作的。

有一个程序用来检测系统的总线类型：PC、ISA、EISA、MCA 和 PCI。其他大量的程序用来检测从 8088 到 Pentium Pro 的 CPU 类型、该处理器的真正指令集（也可能与该类型不匹配！）、浮点处理器类型、CPU 供应商的身份，内部 CPU 速度分析以及确定内部 CPU 高速缓存的大小。另外，还有许多程序用来检查我所指的内幕指令的功能，并进一步搜寻其他的内幕指令。

其他章节讨论了一些特殊的硬件，包括用某些程序来检测显示器适配卡的类型、软盘驱动器的类型、硬盘驱动器类型和容量、串行口类型以及并行口类型。请参看相关章节以获取进一步信息。在本书后面的附录 A 中列出了所有的这些程序。每个程序的完整源代码也包括在所提供的软盘中。

有必要提供可靠的方法来检查系统提供哪些可用的资源。由于和处理器、适配卡以及大多数软件兼容的系统有很多种，所以实现这一点相当复杂。有许多很容易获得的相关信息往往很容易产生误解。剩下的工作也需要巧妙地探测系统隐藏的配置。

总之，虽然硬件开发商简单地阐述了硬件的功能、版本以及所作的修改，但是这些怪里怪气的阐述是极不可信的，我们必须越过这些阐述以获取真正的信息。在某些情况下，设备检测程序必须依赖于内幕技术才能得到必须的信息。但是我们总是避免这样做。

简单的方法

IBM 确实做了很多事情，来使我们更方便地利用 BIOS 功能来获知系统的信息：中断 11h（设备确定）和中断 12h（内存大小确定）。它们在早期的 PC 上非常有用。尽管它们提供了很有用的信息，甚至在今天的计算机上也依然如此。但是，它们仍然无法提供系统的完整图解。请记住，那时我们只能选择一种处理器，8088，我们还只能使用 360K 的 5.25 英寸的软盘驱动器，并且显卡也只有两种选择，CGA 或者单显。

中断	描述	平台
11h	设备确定	所有

这个中断在 AX 中返回描述系统的标志。这个中断只是简单地从 40:10h 读取 BIOS 设备字。该设备字在上电（POST）自检期间由 BIOS 初始化设置。

由于在运行 POST 之后，不再更新该字，所以其他设备驱动文件或程序可以加入到设备列表中而不用更新该字。设备确定仅用于获得系统的最小配置。

在 AX 中的返回值：

位		
	15=x	经自检 POST 检测到的打印机并行口数（0~3）
	14=x	
	13=0	未用（大多数系统），或者未安装调制解调器
	=1	安装了内部调制解调器（某些系统）
	12=0	未用（所有 AT+系统），或者未安装游戏端口
	=1	安装了游戏端口（PC 与 XT）
	11=x	串行口数（0~4）
	10=x	串行口数（0~4）
	9=x	串行口数（0~4）
	8=0	未用
	7=x	软驱数（如果第 0 位是 1）
	6=x	第 7 位　第 6 位
		0　0=1 个驱动器
		0　1=2 个驱动器
		1　0=3 个驱动器（仅 PC 与 XT）
		1　1=4 个驱动器（仅 PC 与 XT）
	5=x	初始视频模式类型
	4=x	第 5 位　第 4 位
		0　0=EGA/VGA 或后来的显卡
		0　1=CGA 彩色 40 列 25 行
		1　0= CGA 彩色 80 列 25 行
		1　1=单色 80 列 25 行
	3=0	未用（参看下面的文字）
	2=1	系统板上安装了鼠标（参看下面的文字）
	1=1	安装了数学协处理器
	0=1	安装了软驱

该设备也有许多局限。首先，许多系统支持 4 个并行口，而该设备字不能表示第 4 个串行口的存在。当设置为 1 时，内部调制解调器标志位才有效。许多系统带有两个以上的驱动器，但是只有前两个驱动器（a:和 b:）被计入软驱数中。

早期的 PC 用第 3 位和第 2 位来表示主板上带有多少的系统内存（16K、32K、48K 还是 64K）。如果你的软件永远不会运行在早期的 PC 上，你就不必考虑第 2 位的冲突。系统板上安装了鼠标端口的 AT+系统通常设置了第 2 位。当然，这一位并不表示是否支持鼠标。因为在应用程序使用鼠标之前，必须先安装驱动文件。

在安装了数学协处理器之后可能会设置数学协处理器标志位。当系统是带有内置数学协处理器的 486 DX、Pentium 及以后的 CPU 时，也可能会是设置这一位。注意，代码例 4-4 虽然没有使用中断 11h 来确定是否存在协处理器，但是实际上也检查了有效的数学协处理器响应。

中断	描述	支持平台
12h	内存大小确定	所有

使用这个中断时，AX 中将返回系统全部主存的大小。中断只是简单地返回保存在 BIOS 40:13h 处的字。对于一个不超过 640K 的系统来说，返回在 AX 中的值将是 640（280h），1,024 字节块的数目。

某些系统支持容量扩展，以提供额外的主存。在 386 或更晚的 CPU 上，一些内存管理程序会重新安排内存。

记住，内存大小值代表的是主存的全部大小，而不是可用主存的大小。CPU 和 BIOS 会用掉底部的一部分主存。而操作系统也会用掉底部的另外一部分，这部分刚好在 BIOS 数据的上面。另外，某些 BIOS 和一些适配卡也会占用一部分顶部主存，以供自己使用。在这些情况下，全部主存容量会减少 1 到 4K。该区域通常作为扩展的 BIOS 数据区。

许多病毒，例如 Michelangelo，也会占用系统的顶部内存。该病毒程序先将自己装入主存的顶部，然后减少 40:13h 处的值，就好像系统的内存现在减少了一样。实际上，该部分内存依然存在，只不过被病毒代码占据着。

系统检测

在使用某平台的任意特性之前，检测该系统非常重要。这些系统包括早期的 PC、XT、AT、EISA、MCA 以 PCI 类型的系统。针对大多数的 AT 系统以及更晚的系统，IBM 提供了专门服务以获取一个指针，该指针指向有关系统细节的 ROM。早期的系统不支持中断 15h 服务，也不标识 EISA 或 PCI 系统。更多的信息参看第 13 章，系统功能，中断 15h，功能 C0h。

代码例 4-1 提供了一种可靠的方法来确立该软件所运行的系统的类型。这个程序可以和其他检测程序一起使用以获得该系统的完整描述。

代码例 4-1 系统检测

为了检测大多数现有系统的系统类型，可以使用中断 15h 的 C0h 功能，它返回一个指针，指向系统 BIOS 的关键系统数据，包括型号及子型号类型。它直接从 BIOS 的尾部复制这个数据，并用 1 位来表示该系统是否使用微通道（Micro Channel，MCA）总线。

BIOS 的倒数第二个字节，即 F000:FFFE 表示了型号类型，下一位就是子型号类型。一些内存管理程序会改变这些字节的内容，所以避免直接从内存读取这些字节就显得非常重要了。今天大多数制造商都符合 IBM 的惯例，但是许多古老的 286 以及更早的仿制系统在型号及子型号方面会出现这样那样的问题，所以不要完全相信这些字节。该检测程序还检查处理器类型的型号信息。

一个 EISA 系统在 F000:FFD9 处有文串“EISA”。因为 EISA 系统总是 32 位系统，所以一个 EISA 系统一定有一个 386 或更晚的处理器。

要看子程序 SYSVALUE 在你的系统上的运行结果，只须运行程序 SYSTYPE。在某个系统上 SYSTYPE 的运行结果如图 4-1 所示。SYSTYPE 子程序返回了系统类型，并显示在“系统类型”一行中。另外，下一行显示了 PCI 的检测结果。PCI 信息由子程序 PCI-DETECT 提供，而 BIOSINFO 则给出了系统所使用 BIOS ROM 的详细信息。

SYSTEM ANALYSIS	v2.00（c）1994,1996 FVG
System type:	AT/ISA（Industry Satndard Architecture）
	PCI v 2.00（Peripheral Component Interconnect）
System BIOS:	American Megatrends Inc
	Date: 25-Jul-94
	BIOS supports LBA disks（> 504 MB）
	Motherboard is made in USA
Diskette a:	2.88MB 3.5"（80 tracks, 36 sectors per track）
Diskette b:	1.2MB 5.25"（80 tracks, 15 sectors per track）
Hard Drive 0:	IDE controller, LBA active
	Total size = 1545 MB（with diagnostic cylinder）
	785 Cylinders ,64 Heads,63 Sectors per track
Hard Drive 1:	AT or IDE type controller
	Total size = 503 MB with diagnostic cylinder
	1022 Cylinders,16 Heads,63 Sectors per track
Video Adapter:	SVGA,programs should use color attributes.
Video Vendor:	ATI MACH64

图 4-1 在一个 Pentium 系统上运行 SYSTYPE 的结果

下面的子程序 SYSVALUE 位于 SYSTYPE 中，用来确定系统的类型。

```
;--------------------------------------------------------------------------
;       系统类型检测子程序
;       确定软件
;       在何种类型的系统上运行
;
;       调用：无
;
;       返回:       al = 系统类型
;                           0 如果是PC（基于8088）
;                           1 如果是XT（基于8088）
;                           2 如果是PC可逆（基于8088）
;                           3 如果是PCjr（基于8088）
;                           4 其他早于80286的机器
;                           8 如果是XT（基于80286）
;                           10h 如果是AT或ISA
;                           20h 如果是EISA
;                           40h 如果是MCA
;
;       用到的寄存器:       ax, bx
;                           eax, ebx（386或更新式）
;
;       子程序调用:         cpu386

sysvalue proc   near
         push   cx
         push   dx
         push   es

         call   far ptr cpu386      ; 获取cpu类型，并保存在al中
         mov    cl, al              ; 保存CPU类

; 不要直接读BIOS ROM，因为有一些内存管理器，如386 MAX，会改变BIOS末尾的字节。

         push   cx                  ; 在堆栈中保存cpu号
```

```
        mov     ah, 0C0h
        int     15h                     ; 获取BIOS配置数据es:bx
        pop     cx
        jc      sys_skp1                ; 如果不支持配置则跳转
                                        ;   (老式BIOS)
        mov     dl, es:[bx+2]           ; 获取模式字节
        mov     dh, es:[bx+3]           ; 获取子模式字节

        mov     al, 40h                 ; 是否为 MCA
        test    byte ptr es:[bx+5], 2
        jnz     sys_Exit                ; 如果是MCA就退出
        jmp     sys_skp2

; 如果一个老式的PC不支持内存管理器，程序到这里执行

sys_skp1:                               ; 好，直接获取BIOS模式
        mov     ax, 0F000h
        mov     es, ax                  ; 指向系统BIOS
        mov     dx, es:[0FFFEh]         ; 获取模式和子模式字节

;现在可以利用DX中的模式和子模式字节来确定机器类型

sys_skp2:
        xor     al, al                  ; 是不是PC (al=0)
        cmp     dl, 0FFh
        je      sys_Exit                ; 如果是PC则跳转
        inc     al                      ; 是不是XT (al=1)
        cmp     dl, 0FEh
        je      sys_Exit                ; 如果是XT则跳转
        cmp     dl, 0FBh
        je      sys_Exit                ; 如果是XT则跳转
        inc     al                      ; 是不是PC可逆 (al=2)
        cmp     dl, 0F9h
        je      sys_Exit                ; 如果是PC可逆，则跳转
        inc     al                      ; 是不是PCjr (al=3)
        cmp     dl, 0FDh
```

```
        je      sys_Exit                ; 如果是PCjr则跳转
        inc     al                      ; 是不是其他286以前的类型 (al=4)
        cmp     cl, 2                   ; cl=CPU 类型—286以前的?
        jb      sys_Exit                ; 如果是，则跳转
        ja      sys_skp3                ; 如果是386或更新的类型则跳转

; 可能是一个286XT，使用模式和子模式字节来确定

        mov     al, 8                   ; 是不是286XT
        cmp     dx, 02FCh               ; 是286XT?
        je      sys_exit                ; 如果是，则跳转

; 检查位于地址F000:FFD9处是否有 " EISA " 串来判断是不是EISA系统

sys_skp3:
        mov     ax, 0F000h
        mov     es, ax
        mov     al, 10h                 ; 是不是标准 AT/ISA
        cmp     word ptr es:[0FFD9h], 'IE'
        jne     sys_exit                ; 如果不是EISA，则跳转
        cmp     word ptr es:[0FFDBh], 'AS'
        jne     sys_exit                ; 如果不是EISA，则跳转
        mov     al, 20h                 ; EISA机器
sys_Exit:
        pop     es
        pop     dx
        pop     cx
        ret
sysvalue endp
```

下面位于 SYSTYPE 中的子程序 PCI-DETECT，用来确定 PCI 的存在性。该子程序使用中断 1Ah 功能来检测 PCI 及其版本信息。

```
; ---------------------------------------------------------------------------
;    PCI检测子程序
;        确定BIOS是否支持
```

```
;       PCI，如果支持，则返回PCI BIOS版本
;
;       调用:           无
;
;       返回:           进位 = 0如果存在PCI BIOS，并:
;                         bx = BCD形式的版本
;                       进位 = 1  如果不支持PCI BIOS
;
;       使用到的寄存器:     ax, bx
;                           edx (386或更新类型)
;
;       子程序调用:     cpu386

.386

pci_detect proc     near
        push        cx

        call        far ptr cpu386      ; 获取CPU类型并保存在al中
        cmp         al, 3               ; 386或更新类型?
        jae         pci_check           ; 如果是，则跳转
        stc                             ; 设置进位标志 (无 PCI)
        jmp         pci_exit

pci_check:
        mov         ax, 0B101h          ; PCI检测功能
        mov         edx, " PCI"         ; 标识符
        int         1Ah                 ; 检查是否存在

pci_exit:
        pop         cx
        ret
pci_detect endp

.8086
```

BIOSVEN 子程序返回系统 BIOS 的关键信息。这些信息包括供应商身份和 BIOS 日期（按照统一的格式）。取决于供应商，其他的信息包括是否支持 LBA、是否使用芯片组、主板生产地点以及供应商的 ASCII 字符串。BIOSVEN 程序调用好几个专用子程序，这里没有将它们打印出来，但是完整的源代码参看软件包。

```
; ---------------------------------------------------------------------------
;    BIOS信息子程序
;       确定BIOS供应商及其他关键信息
;
;       调用:           al = 系统类型（可以调用syaualue获得）
;
;       返回:           al = BIOS
;                            0 是否支持LBA未知
;                            1 不支持LBA
;                            2 支持LBA
;                       ah = 主板制造商位置
;                            0 = 未知
;                            1 = 美国以外（来自AMI代码）
;                            2 = 美国Megatrends
;                            3 = 美国（来自AMI代码）
;                       dl = 供应商值
;                            0 = 未知
;                            1 = 美国Megatrends
;                            2 = Award 国际软件
;                            3 = Phoenix 技术
;                            4 = 芯片与技术
;                            5 = 康柏
;                            6 = DTK
;                            7 = Eurosoft
;                            8 = Faraday（西部数字）
;                            9 = 惠普
;                           10 = Landmark 国际研究
;                           11 = Microid室研究
;                           12 = Olivetti
;                           13 = Quadtel
;                           14 = 东芝
```

```
;                                   15 = 西部数字
;                                   16 = IBM
;                                   17 = IBM (老式)
;                               es:bx = 芯片组串 (ASCIIZ, 最大10个字符)
;                               es:si = 供应商串 (ASCIIZ, 最大40个字符)
;                               es:di = BIOS日期 (ASCIIZ, 格式年-月-日)
;
;       用到的寄存器:       ax,bx
;
;       子程序调用:         find_string, compare_date


; 串表，串的长度出现在最前面（最大31个字符）
;   biosven串必须是大写的，小于30h
;   的符号只能是空格或 " & "
;   建议将AMI、Award和Phoenix放在
;   前面，因为其他的供应商也可能使用这三种类型
;   IBM应该放在最后，因为许多BIOS会在
;   BIOS中包含IBM串，比如 " IBM兼容 " 。

biosven         db      19, 'AMERICAN MEGATRENDS'   ; 放在前面
                db      5,  'AWARD'                 ; 放在前面
                db      7,  'PHOENIX'               ; 放在前面
                db      20, 'CHIPS & TECHNOLOGIES'
                db      6,  'COMPAQ'
                db      3,  'DTK'
                db      4,  'ERSO'
                db      8,  'EUROSOFT'
                db      7,  'FARADAY'
                db      7,  'HEWLETT'
                db      8,  'LANDMARK'
                db      7,  'MICROID'
                db      5,  'MYLEX'
                db      8,  'OLIVETTI'
                db      7,  'QUADTEL'
                db      7,  'TOSHIBA'
```

```
                db      15, 'WESTERN DIGITAL'
                db      8,  'IBM CORP'                  ; 放在尾部
                db      7,  'IBM 198'                   ; 放在尾部
                db      0

biosc_string    db      10 dup (0)
biosv_string    db      40 dup (0)
biosd_string    db      '00/00/00', 0       ; 日期，格式年/月/日
motherboard     db      0                   ; 主板信息临时存放点
LBA_support     db      0                   ; LBA信息临时存放点

ami_LBA_date    db      '94/07/25'          ; AMI BIOS上的LBA日期

; 下面的串用来找寻供应商专用信息。
;       * = 任意字符, # = 数字0-9

ami1_string     db      31, '##-####-******-########-######'  ; 新式
ami2_string     db      15, '-####-######-K#'                 ; 老式

award_string    db      22, '##/##/##-***-####-####'

find_date1      db      8, '##/##*##'
find_date2      db      8, '##-##-##'

bios_size       dw      0                   ; BIOS中的要查看的字节

biosinfo proc   near
        push    ds
        push    cs
        pop     ds

        mov     bx, 0F000h                  ; 假定BIOS开始于F000
        mov     cx, 0FFE0h                  ; 要检查的字节数
        cmp     al, 8                       ; 老式PC/XT ?
        ja      biosi_skp1                  ; 如果不是，则跳转
        mov     bx, 0FE00h                  ; 使用较小的BIOS限
```

```
        mov     cx, 01FE0h              ; 要检查字节数
biosi_skp1:
        mov     [bios_size], cx
        mov     es, bx
        mov     si, offset biosven
        mov     dx, 1                   ; dl是BIOS供应商号

biosi_loop1:
        cmp     byte ptr [si], 0        ; 没有其他字节需要核对
        je      biosi_skp5

        call    find_string             ; 看看在BIOS的es:0处是否有
                                        ; 串[SI]
        jc      biosi_skp3              ; 如果有，则跳转
        inc     dx
        mov     al, [si]
        xor     ah, ah
        add     si, ax                  ; 移到下一个串
        inc     si
        jmp     biosi_loop1             ; 试一试下一个供应商串

        ; 找到了供应商串，串开始于es:bx

biosi_skp3:
        mov     si, offset biosv_string
        mov     cx, 39                  ; 串限制
biosi_loop2:
        mov     al, es:[bx]
        cmp     al, 20h                 ; 允许空格
        jb      biosi_skp5
        je      biosi_skp4
        cmp     al, '&'                 ; 允许'&'符号
        je      biosi_skp4
        cmp     al, 30h                 ; 跳过某些符号
        jb      biosi_skp5
        cmp     al, 7Eh                 ; 跳过小于7Eh的字符
```

```
        ja      biosi_skp5
biosi_skp4:
        mov     [si], al                ; 保存字符
        inc     si
        inc     bx
        loop    biosi_loop2

biosi_skp5:
        cmp     [biosv_string], 0       ; 没有找到串
        jne     biosi_skp6
        mov     dl, 0                   ; 设置未知标志

        ; 查找供应商专用信息

biosi_skp6:
        cmp     dl, 1                   ; AMI?
        jne     biosi_skp7              ; 如果不是则跳转

        mov     cx, [bios_size]
        mov     si, offset ami1_string
        call    find_string             ; 看看开始于es:0的BIOS中
                                        ; 是否有串[SI]
        jnc     biosi_ami1              ; 如果没有找到，则跳转
        mov     al, es:[bx]             ; 获取主板字节
        add     bx, 24                  ; 指向日期串
        jmp     biosi_ami2

biosi_ami1:
        mov     si, offset ami2_string
        call    find_string             ; 看看开始于es:0的BIOS中
                                        ; 是否有[SI]
        jnc     biosi_skp7              ; 如果没有找到，则跳转
        mov     ax, [bx-4]              ; 获取4个芯片组字符
        mov     word ptr [biosc_string], ax
        mov     ax, [bx-2]
        mov     word ptr [biosc_string+2], ax
```

```
        mov     al, es:[bx+1]           ; 获取主板字节
        add     bx, 6                   ; 指向日期串
biosi_ami2:
        sub     al, 2Fh                 ; 将AMI代码转换成
        and     al, 0FEh                ; 主板标识
        shr     al, 1
        cmp     al, 3
        ja      biosi_ami3              ; 如果值无效，则不用保存
        mov     [motherboard], al       ; 保存

        ; 将位于es:bx处的AMI日期转换成年-月-日格式
biosi_ami3:
        mov     di, offset biosd_string
        mov     ax, es:[bx]
        mov     [di+3], ax              ; 插入月
        mov     ax, es:[bx+2]
        mov     [di+6], ax              ; 插入日
        mov     ax, es:[bx+4]
        mov     [di], ax                ; 插入年

        ; AMI在94/7/25或以后的BIOS中开始支持LBA

        mov     [LBA_support], 1        ; 没有LBA
        mov     di, offset ami_LBA_date
        mov     si, offset biosd_string
        call    compare_date            ; 比较
        jnc     biosi_done2             ; 如果日期更早，则跳转
        mov     [LBA_support], 2        ; 应该支持 LBA
biosi_done2:
        jmp     biosi_done

biosi_skp7:
        cmp     dl, 2                   ; Award
        jne     biosi_skp8              ; 若不是，则跳转

        mov     cx, [bios_size]
```

```
        mov     si, offset award_string
        call    find_string             ; 看看有没有串[SI]
        jnc     biosi_skp8              ; 如果没有，则跳转

        mov     ax, es:[bx+9]           ; 获取芯片组字符
        mov     word ptr [biosc_string], ax
        mov     al, es:[bx+11]
        mov     [biosc_string+2], al

        mov     di, offset biosd_string
        mov     ax, es:[bx]
        mov     [di+3], ax              ; 插入月
        mov     ax, es:[bx+3]
        mov     [di+6], ax              ; 插入日
        mov     ax, es:[bx+6]
        mov     [di], ax                ; 插入年
        jmp     biosi_done

        ; 普通的BIOS数据搜索

biosi_skp8:
        push    es
        mov     ax, 0FFFFh
        mov     es, ax
        mov     cx, 15                  ; 看看串是否在最后15个字符中
        mov     si, offset find_date2
        call    find_string
        jc      biosi_date_found        ; 如果找到日期，则跳转
        mov     si, offset find_date1   ; 试一试其他格式
        call    find_string
        jc      biosi_date_found
        pop     es
        push    es
        mov     cx, [bios_size]         ; 仔细查看所有的BIOS
        call    find_string
        jc      biosi_date_found        ; 如果找到日期，则跳转
```

```
        mov     si, offset find_date2
        call    find_string
        jnc     biosi_skp10             ; 如果找不到日期，则跳转

biosi_date_found:
        mov     di, offset biosd_string
        mov     ax, es:[bx+6]
        mov     [di], ax                ; 插入年

        ; 检查是美国顺序还是国际通行的顺序

        mov     ax, es:[bx]             ; 如果是美国顺序，则这是月
        mov     cx, es:[bx+3]           ; 如果是美国顺序，则这是日
        xchg    al, ah
        cmp     ax, '12'                ; 比12大，不是美国顺序
        xchg    al, ah
        jbe     biosi_skp9              ; 如果像是美国顺序，则跳转
        xchg    ax, cx                  ; 改变成美国顺序
biosi_skp9:
        mov     [di+3], ax              ; 插入月
        mov     [di+6], cx              ; 插入日
biosi_skp10:
        pop     es

biosi_done:
        cmp     byte ptr [biosd_string], '8'    ; 20世纪80年代?
        je      biosi_skp11                     ; 如果不是，则无LBA
        cmp     byte ptr [biosd_string], '9'    ; 20世纪90年代?
        jne     biosi_skp12                     ; 如果不是，则未知
        cmp     byte ptr [biosd_string+1], '4'  ; 1994年或以后?
        jae     biosi_skp12                     ; 如果是，则未知
biosi_skp11:
        mov     [LBA_support], 1                ; 在1980~1993年的BIOS上无LBA
biosi_skp12:
        mov     ah, [motherboard]
        mov     al, [LBA_support]
```

```
        mov     bx, offset biosc_string  ; BIOS 芯片组ID
        mov     si, offset biosv_string  ; BIOS 供应商
        mov     di, offset biosd_string  ; BIOS 日期串
        push    ds
        pop     es
        pop     ds
        ret
biosinfo endp
```

CPU 信息

现在，一般在 PC 结构中主要使用六类处理器家族系列和三种标准的外部浮点处理器（FPU）。许多 CPU 有一系列变型，这取决于其制造商。当前 486 的一些变型涉及有没有浮点处理器、芯片所带高速缓存的不同大小以及外部总线宽度。例如，Cyrix 和 TI 提供两种 486 分别带有 1K 和 2K 的内部高速缓存，同时 Intel 80486 是 8K，IBM80486 是 16K。我创建了大量的程序来帮助标出所带的 CPU 类型。运行 CPUTYPE 程序来进行 CPU 的完全分析。CPU TYPE 带有下列命令行参数：

CPUTYPE	**普通操作**
CPUTYPE-	**不检测内部/外部 FPU**
CPUTYPE+	**包括测试高速缓存计时的时间**

减号选项表示禁止那些视为冒险的测试。例如，为了确定 FPU 是芯片内部的，比如 80486DX；还是芯片外部的，比如 80486SX，测试必须执行某个保护模式下的指令。某些早期的内存管理程序会产生一般保护性错误并终止这个程序。大多数的内存管理程序创建于 1993 年或者更晚，明显地它们会仿效这些措施。

针对那些带有活动的内部高速缓存的处理器，加号选项将加上实际的高速缓存定时测试结果。高速缓存的大小测试将确定 CPU 花多长时间去执行从一块内存中多次读取。测试的数据块大小从 0.5K 到 64K。定时测试的结果将规范化，以显示 0.5K、1K、2K、4K、8K、16K、32K、64K 大小高速缓存测试之间的相对差距。两次定时测试之间的显著差别表示到了高速缓存结尾。在 Pentium 和 Pentium Pro 上，只测量数据高速缓存的大小。

程序 CPUTYPE 用来说明每一个子程序是如何用来标识 CPU 的某个特定信息的。完整的 CPUTYPE 源代码包括在软盘中，但是这儿并没有显示出来。图 4-2 给出了运行 CPUTYPE 的结果。

CPU 分析信息 V2.00（c）1994, FVG	
CPU 类	奔腾/586（FPU 内置于 CPU）
CPU 指令集	奔腾带 MMX（矩阵-数学扩展）
CPU 供应商	英特尔
内部 CPU 速度	167MHz
预取队列大小	64
CPUID-供应商串	GenuineIntel
其他信息	代:1　模式:4　系列:5　最大 reg:1
特性（008001BF）	是　CPU 芯片中有浮点处理器
	是　支持增强型虚 80x86 模式
	是　支持 I/O 断点
	是　支持页面大小扩展
	是　支持时间标志计数器
	是　支持模式专用寄存器
	否　支持 2MB 分页/36 位寻址
	是　支持机器异常检查
	是　支持 CMPXCHG8B 指令
	否　CPU 内部有 APIC
当前 CPU 模式	实模式
内部 CPU 缓存	开放，回写
数据 L1 缓存大小	16KB
代码 L1 缓存大小	16KB（假定值，未测量）

图 4-2　在一台带 MMX 的 CPU 上运行 CPUTYPE 的结果

这里归纳了我所编写的检测 CPU 的专门子程序。这些程序包括：

CPUVALUE	用来检测 CPU，从 8088 到 P8
FPUTYPE	用来检测是否存一个浮点处理器（FPU）并确定其类型
FPULOC	确定 FPU 是 CPU 内置的还是外部的
CPUQ	以字节的形式给出预取队列的大小
CPUSTEP	确定 CPU 芯片的版本
CPUMODE	确定 CPU 工作在实模式还是保护模式
CPUVENDOR	确定 CPU 供应商（AMD，Cyrix，IBM、Intel，NEC，NexGen 或者 TI）
CPUSPEED	测量 CPU 的内部速度、单位 MHz
CPU_CACHE	显示是否开通了 CPU 内部高速缓存
CPU_D_SIZE	确定内部数据高速缓存的大小

代码例 4-2 标识处理器

下面的子程序确定正在运行的 CPU 类型。它使用一些内幕技巧来实现正确的检测过程。该方法可以在相当多的商业性产品上可靠地使用。至于这个程序是如何工作的，将和代码一起作详细说明。CPUTYPE 可以检测下列 CPU 类型：8088/8086、V20/V30、80188/80186、80286、80386、80486、Pentium/586、Pentium Pro/686，也可以处理将来的 P7 和 P8 处理器。

在考虑 CPU 类型时，最重要的是关心其指令集。例如，老式的 IBM386 实际上支持全部的 486 指令集。又比如，Cyrix 的 6x86 只支持 486 指令集(还支持一些但不是全部的 Pentium 指令)。CPUTYPE 程序既返回 CPU 的类型，也返回其工作指令集，这些指令集以 Intel 指令集为参照对象。

如果你只想为单个的处理器写出你所有的代码，那么就不必要使用 CPU 标识。大多数的类型支持 8088 指令，这不需要做任何检测。但是对其他的处理器来说，有必要明晰不同系统的 CPU 的指令和结构之间差别，并且依据不同的系统 CPU，作为相应的处理。

有必要在此提醒，为了检测到 80386、80486 和 Pentium 处理器之间的差别，必须改变一些标志位。这是 Intel 推荐的唯一方法，可以在大多数情况下可靠地发挥作用。改变标志位可能会产生一个保护性错误，这取决于 CPU 的模式。因为今天大多数的系统运行在保护（虚 86）模式下，所以将由内存管理程序或操作系统来负责处理保护性错误，并模拟该标志改变的动作，CPUTYPE 程序可以监测到这一点。这意味着将按照未发生错误一样执行指令 POPFD。这一切都不会出问题，除非保护模式中含有一些错误代码，而这些错误会不正确地处理模拟过程。某些情况下，增强模式的 Windows3.1 就会不正确地模拟 POPED。我所设计的 CPUTYPE 也会做一些额外的测试来处理这种罕见的问题。

在某些情况下，只需要检测处理器是否为 32 位的。参看代码例 4-3，它给出了程序 CPUVALUE 检测 8080 到 386 的处理器更为简单的版本。

```
; ----------------------------------------------------------------------------
; CPU标识子程序
;       标识从8080到P8的CPU。即使是对工
;       作在V86模式下的386或更新的CPU，
;       程序也可以起作用。注意，即使在进入
;       程序时关闭中断，在程序
;       退出时仍会开放中断。如果在运
;       行这个程序时需要禁止中断，只
;       须删除所有的CLI和STI指令，这
```

```
;       样在运行之前中断总是被禁止的
;
;       "CPU 类"是由供应商指定的CPU的
;       类别。它与性能直接相关
;
;       "CPU 标准指令集" 标识了CPU可用的最
;       高级别的Intel兼容指令集。例如
;       NexGen 5x86的性能等级为586，但是只
;       支持Intel 80386所定义的指令
;
;       调用:           无
;
;       返回:           al = CPU 系列
;                               0 如果是8088/8086或V20/V30
;                               1 如果是80186/80188
;                               2 如果是80286
;                               3 如果是80386
;                               4 如果是80486
;                               5 如果是Pentium/586
;                               6 如果是Pentium Pro/686
;                               7 如果是786 (未来)
;                               8 如果是886 (未来)
;                       ah =    位 0 = 0 如果CPUID不可用
;                                        1 如果CPUID可用
;                               位 1 = 0 如果不是V20/V30
;                                        1 如果是NEC V20/V30
;                               位 2 = 0 如果没有486+对齐检查
;                                        1 支持对齐检查
;                       bl = CPU 标准指令集
;                               0 如果是8088/8086
;                               1 如果是80186/80188
;                               2 如果是80286
;                               3 如果是80386
;                               4 如果是80486
;                               5 如果是Pentium
;                               6 如果是Pentium或更高级
;
```

```
;       用到的寄存器:    ax, bx
;                        eax, ebx (386或更新式)
;
;       子程序调用:      hook_int6, restore_int6, bad_op_handler

.8086   ; 包括所有8088/8086指令，被后来覆盖的除外

cpuvalue proc    far
         push    cx
         push    dx
         push    ds
         push    es

; 8088/8086 测试—使用旋转方法—当移
;   位一个字节cl位时，所有后来的CPU都会用0Fh
;   来屏蔽CL。本测试在CL中装入一个大值（20h）
;   然后右移AX。对于8088来说，AX中的所有位
;   都会被移出，得到结果0。在所有更高级的
;   处理器上，移动前CL的值20h会和0Fh相与。
;  这意味着有效的移位次数是0，所以AX不变。

         mov     cl, 20h          ; 在CL中装入一个较大的值
         mov     ax, 1            ; 在AX中装入一个非零值
         shr     ax, cl           ; 移位
         cmp     ax, 0            ; 如果是零，则是8088/86
         jne     up186            ; 如果不是8088/86，则跳转

; V20/V30 测试——现在是 V20/V30 或8088。
;   我将运用另外一种未公开的技巧来辨别
;   它们。在8088上，0Fh将执行POP CS。而在
;   V20/V30，它却是许多字节指令的前缀而已。
;   对于字节串0Fh、14h、C3h，CPU将执行如下操作：
;              8088/8086              V20/V30
;          pos     cs              set1    bl, cl
;          adc     al, 0C3h
```

```
        xor     al, al                  ; 清零al和进位标志
        push    cs
        db      0Fh, 14h, 0C3h          ; 指令（见上文）
        cmp     al, 0C3h                ; 如果al是C3h,则是8088/8086
        jne     upV20
        mov     ax, 0                   ; 设置 8088/8086 标志
        jmp     uP_Exit

upV20:
        pop     ax                      ; 补偿非pop cs操作
        mov     ax, 200h                ; 设置V20/V30标志
        jmp     uP_Exit

; 80186/80188 测试——检查在执行PUSH SP
;   指令时向堆栈中压入了什么内容。在SP的
;   值入栈之前，80186会更新堆栈指针，而
;   所有更高级的处理器会先向堆栈
;   中压入SP的当前值，然后更新堆栈指针。

up186:
        mov     bx, sp                  ; 保存当前的堆栈指针
        push    sp                      ; 测试
        pop     ax                      ; 获取入栈值
        cmp     ax, bx                  ; SP改变了么？
        je      up286                   ; 若没有，则是286+
        mov     ax, 1                   ; 设置80186标志
        jmp     uP_Exit

; 80286 测试 A—我们将查看EFLAGS
;   寄存器的高4位。在286上，这些位总是零。
;   更新的 CPU 则允许改变这些位。在本
;   测试过程中，我们将禁止所有
;   的中断以确保中断不会影响标志位

up286:
        cli                             ; 禁止中断
```

```
        pushf                           ; 保存当前标志位

        pushf                           ; 标志位入栈
        pop     ax                      ; 标志位出栈
        or      ax, 0F000h              ; 试一试将第12~15位置1
        push    ax
        popf                            ; 设置新标志
        pushf
        pop     ax                      ; 看看高位是否为0

        popf                            ; 标志位恢复为初始值
        sti                             ; 开放中断
        test    ax, 0F000h              ; 所有高位是1?
        jnz     up386plus               ; 如果是，则不是286

; 80286 测试 B—如果系统位于V86模式
;    （386或更高），POPF指令会生成一个保
;    护性错误，并且保护模式软件必须模拟POPF操作。
;    如果保护模式软件无效，就像Windows 3.1增强模式中
;    出现bug时遇到的那样，前一种测试方法可
;    能会得到286的结果，但实际上它是一个
;    更高级的处理器。我们将检查保护模式位是否为1。如果不是，
;    将确定的确是286。

.286P                                   ; 允许一个286指令
        smsw    ax                      ; 获取机器状态字
        test    ax, 1                   ; 位于保护模式
        jz      is286                   ; 若不是则跳转（一定是286）

; 80286 测试C—很有可能是386+，但也
;    不一定。系统位于286保护模式下仍有可
;    能，所以我们将进行最后一项测试。我
;    们将试着执行一条386独有的指令。以前，我们要先
;    将坏码中断向量（int 6）指向我们自己的服务程序。

        call    hook_int6               ; 调用
```

```
        mov     [badoff], offset upbad_op3  ; 如果是坏码，将跳到执行的地方
.386
        xchg    eax, eax                    ; 32位空指令（286上是坏码）

        call    restore_int6                ; 恢复中断向量
        jmp     up386plus                   ; 如果是386或以上，程序只运行到这里

; 如果系统为286（假定286保护模式中
;   断6处理程序会执行坏码中断），中断
;   6（坏码）完成后会执行到这里。

upbad_op3:
        call    restore_int6
is286:
        mov     ax, 2                       ; 设置80286标志
        jmp     uP_Exit

; CPUID测试—如果第21位不可变，则表
;   明CPU支持CPUID指令。在测试过程中，我
;   们将禁止中断以确保中断不会影响
;   标志位。

.586                                        ; 支持486指令

up386plus:
        cli                                 ; 禁止中断
        mov     cx, sp                      ; 保存当前堆栈指针
        and     sp, NOT 3                   ; 对齐中断，以免出现AC错误
        pushfd                              ; 标志位入栈
        pop     eax                         ; 获取eflags
        mov     ebx, eax                    ; 保存，供后面使用
        xor     eax, 200000h                ; 对第21位求反
        push    eax
        popfd                               ; 向CPU中装入改变后的eflags
        pushfd                              ; eflage入栈
        pop     eax                         ; 获取当前的eflags
```

```
        push    ebx                     ; 将原始的标志位推入堆栈
        popfd                           ; 恢复原始标志位
        mov     sp, cx                  ; 恢复堆栈指针
        sti                             ; 开放中断
        xor     dl, dl                  ; DL = 临时标志, 0=无CPUID
        xor     eax, ebx                ; 检查是否改变了位
        jz      up386                   ; 如果没有改变，表示无CPUID，跳转
        inc     dl                      ; 设置标志，有CPUID

; 80386 测试—在386上，EFLAGS的第18
;   位不可以设置，但是在486+的CPU上可
;   设置该位。第18位用来指示对齐错误。
;   在测试期间，我们将禁止中断来确保中断不会影响标志位。


up386:
        cli                             ; 禁止中断
        mov     cx, sp                  ; 保存当前的堆栈指针
        and     sp, NOT 3               ; 对齐堆栈，以免出现AC错误
        pushfd                          ; 标志位入栈
        pop     eax                     ; 获取eflags
        mov     ebx, eax                ; 保存，供以后使用
        xor     eax, 40000h             ; 求反第18位，AC标志
        push    eax
        popfd                           ; 向CPU中装入更改后的eflags
        pushfd                          ; eflags入栈
        pop     eax                     ; 获取当前eflags
        push    ebx                     ; 将原始值压入栈顶
        popfd                           ; 恢复原始标志位
        mov     sp, cx                  ; 恢复堆栈指针
        sti                             ; 开放中断
        xor     eax, ebx                ; 检查位是否改变
        jz      upNoAC                  ; 无对齐检查
        or      dl, 4                   ; DL = 临时标志，位2=1 表示有AC
        jmp     up486                   ; 改变了，则是486+
```

```
        ; 似乎是386—检查是否为NexGen Nx586

upNoAC:
        push    dx                      ; 保存临时标志（CPUID和AC）
        mov     ax, 5555h               ; 将AX初始化为非零值
        xor     dx, dx                  ; 将零标志位设置为1
        mov     cx, 2
        div     cx                      ; 在执行DIV时Nx586处理器不会
        pop     dx                      ; 改变零标志位
        mov     ax, 3                   ; 假定是386系列
        jnz     upCPUID                 ; 如果零标志位是0，则不是Nx586
        mov     ax, 5                   ; 设置586系列标志
        jmp     upCPUID

up486:
        mov     ax, 4                   ; 设置486系列标志
        test    dl, 1
        jnz     upCPUID                 ; 如果CPUID有效,就执行它

        ; 检查是否为Cyrix，到目前
        ; 为止，它可能仍很像486。某
        ; 些版本的Cyrix提供了一个选项来关闭CPUID指令，
        ; BIOS初始化时会将CPUID关闭。
        ; 在鲁棒性较差的CPU标识程序上，
        ; Cyrix 5x86和6x86常常被标识成486。

        ; 非Cyrix的CPU执行除法指令时只会
        ; 改变某些标志位（未确定），利用这一点
        ; 可以区分Cyrix。Cyrix会清除所有
        ; 的标志位（当然第1位为1不变除外）

        xor     ah, ah
        sahf                            ; 标志清零
        mov     ax, 10                  ; 除法使用的具体值不重要
        mov     cl, 4                   ;
        div     cl                      ; 执行除法操作
```

```
        lahf                            ; 获取标志位
        and     ah, 0FDh                ; 忽略第1位
        cmp     ah, 0                   ; 标志位为零?
        je      is_Cyrix                ; 如果是，则是Cyrix
        mov     ax, 4
        jmp     upInSet                 ; 一定是486，非Cyrix

        ; 是Cyrix 486+,没有CPUID可执行。接着
        ; 我们利用Cyrix独特的系统寄存器
        ; 端口22h和端口23h来确定
        ; Cyrix的具体类型。

is_Cyrix:
        mov     al, 0FEh
        call    read_cyrix_reg          ; 获取设备ID寄存器0(FEh)
        mov     bl, al                  ; 保存Cyrix CPU类型值
        mov     ax, 4                   ; 先假定是486类型
        and     bl, 0F0h                ; 忽略低四位
        cmp     bl, 060h                ; 未定义?
        jae     upInSet                 ; 一定是老式的Cyrix 486
        cmp     bl, 10h                 ; 定义成486?
        jbe     upInSet                 ; 如果是，则跳转
        mov     al, bl                  ; 改变成5或6
        shr     al, 4
        add     al, 3
        jmp     upInSet                 ; AX=5表示5x86, AX=6表示6x86

; 如果允许，则可以使用CPUID指令来获
;   得CPU类。CPUID会返回一个系列号0或更高
;   的值来表征处理器类。从1996年开始，
;   新式的Intel 486以后许多新式的处理器
;   (Pentium, Pentium Pro)都支持CPUID指令。其他的供应商
;   新发布的CPU也支持CPUID指令。

upCPUID:
        test    dl, 1                   ; CPUID 指令缺失?
```

```
        jz      upInSet                 ; 如果无CPUID, 则跳转

        push    ecx                     ; CPUID 将eax改变成edx
        push    edx                     ;
        mov     eax, 1                  ; 获取系列信息功能
        CPUID                           ; CPUID指令的宏
        and     eax, 0F00h              ; 找到系列信息
        shr     eax, 8                  ; 移位得到al
        mov     ah, 1                   ; 设置标志, 支持CPUID
        pop     edx
        pop     ecx


; AL中保存了CPU系列号（386或更高级），
; 我们只须测试指令集的有效性。我们
; 将尝试一个486、Pentium和Pentium Pro
; 特有的指令，以前，记住将坏码中
; 断向量（中断6）指向自己的处理程序。

upInSet:
        mov     ah, dl                  ; 退出时CPUID和AC标志
        cmp     al, 3                   ; 如果低于386，完成!
        jb      up_Exit

        push    ax
        call    hook_int6               ; 继续
        mov     [badoff], offset upbad_op4  ; 如果坏码，跳到这个地方执行
.486
        xadd    al, ah                  ; 交换加（486+）
        bswap   eax                     ; 字节交换（486+）

        call    restore_int6            ; 恢复向量
        pop     ax                      ; 支持486指令

        ; 现在试用 Pentium 指令
```

```
        cmp     al, 4                       ; 如果是486，完成!
        jbe     up_Exit

        push    ax
        call    hook_int6                   ; 继续
        mov     [badoff], offset upbad_op5  ; 坏码处理地点
.586
        mov     eax, cs:[0]                 ; 使用任意地址
        not     eax                         ; 确保比较
                                            ;   一定改变cs:[0]
        cmpxchg8b qword ptr cs:[0]          ; 比较并交换 (Pentium+)

        call    restore_int6                ; 恢复向量
        pop     ax                          ; 支持 Pentium 指令

        ; 现在试用 Pentium Pro 指令

        cmp     al, 5                       ; 如果是Pentium, 完成
        jbe     up_Exit

        push    ax
        call    hook_int6                   ; 继续
        mov     [badoff], offset upbad_op6  ; 坏码处理地点

;       cmovne  ax, bx                      ; 条件赋值，不相等
                                            ;   (Pentium Pro)
        db      0Fh, 45h, 0C3h              ; CMOVNE的字节编码，许多
                                            ;  汇编程序不能对其编码
        call    restore_int6                ; 恢复向量
        pop     ax                          ; 支持 Pentium Pro 指令
        jmp     up_Exit

; 如果CPU不支持486指令集，中断向量
;   6（坏码中断）会执行到这里

upbad_op4:
```

```
        call    restore_int6
        pop     ax
        mov     bl, 3                   ; 指令集 386
        jmp     uP_Exit2

; 如果CPU不支持Pentium指令集，中断
;   向量6（坏码中断）会执行到这里

upbad_op5:
        call    restore_int6
        pop     ax
        mov     bl, 4                   ; 指令集486
        jmp     uP_Exit2

; 如果CPU不支持Pentium Pro指令集
;   中断向量6（坏码中断）会执行到这里

upbad_op6:
        call    restore_int6
        pop     ax
        mov     bl, 5                   ; 指令集Pentium
        jmp     uP_Exit2

up_Exit:
        mov     bl, al                  ; 设置指令集
up_Exit2:
        pop     es
        pop     ds
        pop     dx
        pop     cx
        ret
cpuvalue endp
.8086                                   ; 返回到 8086 指令
```

代码例 4-3 简单的处理器识别

与代码例 4-2 不同，下面的程序更小更简单。它只检测 CPU 是 8088、8086、286 还是 386 或更晚的类型。

```
; --------------------------------------------------------------------------
;   CPU 386 标识子程序
        标识从8088到386+的CPU类型。
        这是适用范围更广的CPUVALUE程序的
        一个子集。当不必要标识386以上的
        CPU类型时，可以使用这个程序
;
;       调用:       无
;
;       返回:       al = CPU 类型
;                       0 如果是 8088/8086 或 V20/V30
;                       1 如果是 80186/80188
;                       2 如果是 80286
;                       3 如果是 80386+
;
;       使用的寄存器:    ax,bx
;                       eax（仅32-位CPU）
;
;       调用的子程序：hook_int6, restore_int6, bad_op_handler

 .8086  ; 所有的8088/8086指令，某些后来被覆盖的指令除外

cpu386  proc    far
        push    cx
        push    dx
        push    ds
        push    es

; 8088/8086 测试—使用旋转方法—
;   当移位一个字节CL位时，所有后来的CPU
;   都会用0Fh来屏蔽CL。本测试在CL中
```

```
;    装入一个大值（20h）然后右移AX。对于
;    8088来说，AX中的所有位都会被移出，
;    得到结果0。在所有更高级的处理器上，移
;    动前CL的值20h会和0Fh相“与”。这意味
;    着有效的移位次数是0，所以AX不变。

        mov     cl,20h          ; 在CL中装入一个较大的值
        mov     ax,1            ; 在AX中装入一个非零值
        shr     ax,cl           ; 移位
        cmp     ax,0            ; 如果是零，则是8088/86
        jne     uP186           ; 如果不是8088/86，则跳转
        jmp     uP_Exit
; 80186/80188 测试A—检查在执行PUSH SP
;    指令时向堆栈中压入了什么内容。在SP
;    的值入栈之前，80186会更新堆栈指针，
;    而所有更高级的处理器会先向堆栈中压入
;    SP的当前值，然后更新堆栈指针。

; 80286 测试A—我们将查看EFLAGS寄
;    存器的高4位。在286上，这些位总是零。
;    更新的CPU则允许改变这些位。在本测
;    试过程中，我们将禁止所有的中断以确保
;    中断不会影响标志位。
up286:
        cli                             ; 禁止中断
        pushf                           ; 保存当前标志位

        pushf                           ; 标志位入栈
        pop     ax                      ; 标志位出栈
        or      ax,0F000h               ; 试一试将第12~15位置1
        push    ax
        popf                            ; 设置新标志
        pushf
        pop     ax                      ; 看看高位是否为0

        popf                            ; 标志位恢复为初始值
```

```
        sti                             ; 开放中断
        test    ax,0F000h               ; 所有高位是1?
        jnz     up386plus               ; 如果是，则不是 286

; 80286 测试B—如果系统位于V86模
;       式（386或更高），POPF指令会生成一个
;       保护性错误，并且保护模式软件必须模拟
;       POPF操作。如果保护模式软件无效，就
;       像Windows 3.1增强模式中出现bug时
;       遇到的那样，前一种测试方法可能会得到286的结果，
;       但实际上它是一个更高级的处理器。我们将检查
;       保护模式位是否为1。如果不是，将确定的确是286。

.286P                                   ; 允许一个286指令
        smsw    ax                      ; 获取机器状态字
        test    ax, 1                   ; 位于保护模式
        jz      is286                   ; 若不是则跳转

; 80286 测试C—很有可能是386+，但也不一定。
;       系统位于286保护模式下仍有可能，所
;       以我们将进行最后一项测试。我们将试着
;       执行一条386独有的指令。以前，我们要先将
;       坏码中断向量（int6）指向我们自己的服务程序。

        call    hook_int6               ; 继续
        mov     [badoff], offset upbad_op3 ; 如果是坏码则到这个地方执行
        .386
        xchg    eax,eax                 ; 32 位空指令（286上是坏码）

        call    restore_int6            ; 恢复中断向量
        jmp     up386plus               ; 如果是386或以上，程序只运行
                                        ; 到这里

; 如果系统为286（假定286保护模式中断6
;       处理程序会执行坏码中断），中断6（坏
;       码）完成后会执行到这里。
```

```
upbad_op3:
        call    restore_int6
is286:
        mov     ax, 2                   ; 设置 80286 标志
        jmp     uP_Exit
up386plus:
        mov     ax, 3                   ; 32位CPU（386或以上）
up_Exit:
        pop     es
        pop     ds
        pop     dx
        pop     cx
        ret
cpu386  endp
.8086                                   ; 返回至8086指令
```

代码例 4-4 浮点处理器的检测器

这个程序确定是否带有浮点处理器及其类型：8087、80287 或 80387。该 80287FPU 是否附属于 80386CPU。记住，对于那些带有集成 FPU 的 CPU 类型，这个程序将返回值 4 或更高。这个程序检验是否带有 FPU，但并不指出 80486CPU 是 DX 类型（带有内置 FPU）还是带有外置 80387FPU 的 80486SX 类型。使用代码例 4-5 指示的程序 FPULOC 可以检测 FPU 的位置。

FPUTYPE 程序要求 CPUVALUE 程序在之前已经运行，并且将 CPU 类型保存在了变量 CPU-VAL 中。

```
; ---------------------------------------------------------------------------
;       浮点检测子程序
;           通过检查FPU的状态和控制字确定是否
;           存在数学协处理器。还检测80386系统
;           的FPU是80287还是80387
;
;           调用:       ds = cs (处理局部变量 fpu_temp)
;                           ds:[cpu_val] 设置为 CPU 类
;                           ds:[cpu_inst] 设置为 CPU 指令集
;
;           返回:       al = FPU 类型
```

```
;                                   0 如果无FPU
;                                   1 如果是 8087 .
;                                   2 如果是 80287
;                                   3 如果是 80387 或等价物
;                                   4 如果是 80486 带 FPU
;                                   5 如果是 Pentium (586) 带 FPU
;                                   6 如果是 Pentium Pro (686) 带 FPU
;                                   7 如果是 P7 (786) 带 FPU
;
;
;       用到的寄存器:       ax

fpu_temp dw     0                   ; 检测器临时存储字

.386
.387                                ; 支持数学指令

fputype proc    near
        fninit                      ; 初始化FPU（重启）
        mov     [fpu_temp], 1234h   ; 设定为非零值
        fnstsw  [fpu_temp]          ; 获取FPU状态字
        and     [fpu_temp], 0FFh    ; 只看低8位
        jnz     fput_not_found      ; 如果非零，则无FPU
        fnstcw  [fpu_temp]          ; 获取FPU控制字
        and     [fpu_temp], 103Fh   ; 舍去不需要的位
        cmp     [fpu_temp], 3Fh     ; 相应的位是否已经设置?
        je      fput_present        ; 如果是，则跳转，表示有FPU

fput_not_found:
        mov     ax, 0               ; 没有FPU
        jmp     fput_Exit

; 有FPU，接着确定其类型是8087、80287还是80387

fput_present:
        mov     ax, 1               ; 考虑 8087
```

```
        cmp     [cpu_val], 2            ; CPU 类型低于286?
        jb      fput_Exit               ; 如果是，则一定是8087
        mov     ax, 2                   ; 考虑 80287
        je      fput_Exit               ; 如果是80286，则FPU是80287
        xor     ah, ah
        mov     al, [cpu_inst]          ; 获取CPU指出，是80386还是更高级?
        cmp     al, 3                   ; 386 ?
        je      fput_386
        mov     al, [cpu_val]
        jmp     fput_Exit               ; 如果是486+则跳转

; 80386 的FPU可能是80287，也可能是80387。
;   要进一步区分，可以检查-infinity是否等于+infinity。如果不等，则是80387。

fput_386:
        fld1                            ; +1 入栈
        fldz                            ; +0 入栈
        fdiv                            ; 1/0 = infinity（无限大）
        fld     st                      ; +infinity 入栈
        fchs                            ; -infinity 入栈
        fcompp                          ; 比较正负infinity
        fstsw   [fpu_temp]              ; 临时变量中的比较状态
        test    [fpu_temp], 4000h       ; 相等?（测试零位）
        jnz     fput_Exit               ; 如果不等，则跳转（是387，al=3）
        mov     al, 2                   ; +/- infinity 相等，是80287

fput_Exit:
        ret
fputype endp
.8086
```

代码例 4-5　定位浮点处理器

这个程序找到 80486 与 Pentium CPU 上的 FPU 的位置。80486SX 在 CPU 上不带 FPU，但是可以在主板上带 80387 来提供 FPU 操作。这个程序指示 FPU 的驻留位置。

为了检测FPU的位置，有必要看看CR0的第4位是否可以改变。这需要执行MOV CR0，EAX 保护模式指令。某些保护模式下的产物，就像某些古老的内存管理程序，不允许执行这条指令。内存管理程序会产生一个一般性保护性错误，然后退出这个程序，而当前的内存管理程序可以模拟 MOV CR0 指令，于是就可以安全地进行测试。由于这一点，我建议在任意商业用途中不要包括这个程序。在使用这个程序时，包括进一些选项以供用户绕开这个程序不失为明智之举。例如，CPUTYPE 程序就带有一个命令行选项来跳过该项测试。

FPULOC 程序要求 FPUTYPE 在之前已经运行，并将 FPUTYPE 保存在变量 FPU_VAL 中。

```
; ------------------------------------------------------------------------------
;    浮点处理器定位子程序
;        检查486/Pentium上的FPU是位于CPU芯片上还
;        是独立存在的。可以检查CR0的扩展
;        类型第四位，看可否改变它。如果不
;        可以改变，则FPU内置于CPU中。
;        否则，FPU位于CPU外。
;
;                    ********** 重要提示 **********
;          在V86模式下改变CR0寄存器会生成
;          一个CPU错误，必须由保护模式软
;          件做透明处理。参见本书以了解细节。
;
;        调用:        ds:[fpu_val] 设置为 FPU 类型
;
;        返回:         al = 0 如果 FPU 不在 CPU 上或无FPU
;                           1 如果 FPU 内置于 CPU中
;
;        用到的寄存器:        ax

.386P

fpuloc   proc     near
         xor      al, al                ; 假定FPU不在CPU中
         cmp      [fpu_val], 4          ; 486 或更高的 CPU 带 FPU?
         jb       fpul_exit             ; 如果不是，则退出
         push     bx
         push     eax
         mov      eax, cr0
```

```
        mov     bx, ax                  ; 保存低位部分
        and     eax, 0FFFFFFEFh         ; 设置16位模式
        mov     cr0, eax
        mov     eax, cr0
        xchg    ax, bx
        mov     cr0, eax                ; 恢复CR0为初始值
        pop     eax
        mov     al, bl
        shr     al, 4                   ; 在第0位中返回ET位
        pop     bx
fpul_exit:
        ret
fpuloc  endp
.8086
```

代码例 4-6　预取队列大小检测器

这个程序确定 CPU 指令预取队列的大小。预取队列是一个先进先出的缓冲区。该缓冲区包含有待执行的指令。不同的处理器系列和不同的供应商有不同的缓冲区大小。

它应用起来还有点古怪。为了找出队列的长度，我利用了不能写队列这一事实。可利用自我修改代码来修改下一条待执行命令。单独一个重复保存串字节（REP STOSB）指令会使用大量的空操作（NOP）指令。在使用增加 BX（INC BX）指令时，也使用了大量的 NOP 指令。在内存中，总是大量地使用这条指令，但是在预取队列中不使用这条空指令。

我创建了一个小程序，来帮助说明自我修改代码的操作过程。表 4-1 显示了这个例子。我用的是一个 4 字节的预取队列。REP STOSB 指令总会向内存写入正确的信息，但是 CPU 不会更新装入到预取队列中的字节。BX 寄存器不会因预取队列的长度而增加。在该例子，我们处理了 6 条指令而 BX 是 2，所以队列长度为 4。

表 4-1　四字节预取队列的操作过程

偏移量	在 REP STOSB 前执行		在 REP STOSB 后执行	
	预取之前的队列	内　存	预取之后的队列	内　存
X	REP	REP	REP	REP
X+1	STOSB	STOSB	STOSB	STOSB
X+2	NOP	NOP	NOP	INC BX
X+3	NOP	NOP	NOP	INC BX
X+4*	NOP	NOP	INC BX	INC BX
X+5*	NOP	NOP	INC BX	INC BX

*这些字节尚未预取。

经过好几个小时的不同设计，我发现下面的程序对 Pentium 以前的系统来说最可靠（称为方法 1）。测试期间禁止了所有的中断，并且清空了预取队列，然后执行 16 个慢除法指令，来得到处理器取预取队列的时间。

不幸的是，某些系统的预取队列比预计的短得多。Intel 没有公布有关怎样装入预取队列的信息，甚至也没有指出某些处理器的预取队列的长度。由于有很多类型 CPU，预取队列子程序的结果并不总是那么可靠。

在 Pentium CPU 上，该测试会显示出没有预取队列的测试结果。在该家族系列中，Pentium 首次检查写操作是否会影响预取队列。如果会影响，那么就会更新预取队列。为了解决这个问题，我们建议采用一种可替代的办法（方法 2）。Pentium 的这种独特之处就在于它可用来得到预取队列精确的大小。这种方法和前面的一种方法有一点不同。

今天仍在使用的基本 AT 设计就带有一个选项，可用来关闭 CPU 的外部 A20 地址线。这种状态是系统上电后的缺省状态，并且它会在实模式下一直保持下去。这种方法一般用来模拟 8088，这时 1MB 以上地址就会被指回到低端内存中。

我们将方法 2 中的预取队列代码放在 64KB 以下的低端内存中，然后将数据段设定为参考 FFFFh。当偏移量为 16 时，实际上我们在访问的是低端内存空间。如果这时地址线是高电平，外部逻辑也会忽略它，所以地址会一直指回到低端内存。这样就不会更新在预取队列中的任意写内存操作，因此我们就可以检测其大小了。

方法 2 中存在的问题是，我们必须关闭 A20 地址线。这个程序在运行之前只会检查 A20 地址线是否关闭，而不会试图去关闭打开着的 A20 地址线。在开通了 A20 地址线时，这个程序会返回一个错误值。

令人吃惊的是，上面这两种方法竟然都不适用于 Pentium Pro。仔细看看 Pentium Pro 硬件手册，你就会发现，地址线 A20 的禁止逻辑是由 CPU 的内部控制所提供的。Pentium Pro 在发现了错误地址时会更新预取队列。尽管这种做法不失为明智之举，然而它也可能是一个 bug（我猜测 Intel 反对！）。除了可用来检测预取队列的大小之外，这两种方法没有多大的作用。

```
;--------------------------------------------------------------------
;    CPU预取队列深度子程序
;        本子程序返回预取队列的深度。每个处理器
;        使用不同大小的队列，
;        其中自我修正代码不起作用。这个子程序
;        巧妙地利用自我修正代码来确定预取
;        队列的大小。
;
;        Pentium+ 的处理器必须禁止A20
;        （即关闭内存管理程序，）来获得预取队列的大小。
```

```
;       这里使用了可供选择的方法2。Pentium Pro
;       再次改变了预取逻辑，因此在Pentium Pro上
;       两种方法的结果都不正确。
;
;       调用:       [cpu_val]
;                   al = 采用的方法
;                           0 = 自动
;                           1 = 采用方法1
;                           2 = 采用方法2
;
;       返回:       ax = CPU 预取队列大小
;                           0 如果Pentium类型的处理器上的A20
;                             开放而不能够测量
;                           2 不能测量（CPU更新
;                             预取队列写）
;                           4 如果是 8088/8086
;                           8 如果是 80286
;                           16 如果是 80386
;                           32 如果是 80486
;                           64 如果是 Pentium
;
;       用到的寄存器:       ax

QSIZE   equ     80h                     ; 最大队列大小
int_backup      db      256 dup(0)      ; 中断表临时存储空间

cpuQ    proc    far
        push    bx
        push    cx
        push    dx
        push    di
        push    es

; 对于所有不动态更新预取队列的CPU来说，方法1能很好地发挥作用。
; Pentium 级的CPU不能采用这个方法，它们必须
; 用到另外一个方法。
```

```
        cmp     al, 0                   ; 核对首先使用的方法
        je      cpuq_auto
        cmp     al, 1
        je      cpuq_method1
        jmp     cpuq_method2

cpuq_auto:
        cmp     [cpu_val], 5            ; Pentium 级?
        jb      cpuq_method1            ; 如果不是，则跳转
        jmp     cpuq_method2

; 通过触发大量的非常慢的DIV
;   指令来使CPU在执行慢速指令时装入预取队列，
;   从而实现队列大小的测试。
;   接着REP SIOB指令将用 " INC BX "
;   覆盖某些NOP指令。如果没有预取队列，
;   所有的NOP指令都会转化成INC BX。
;   如果有16字节的预取队列，
;  将只改变内存中的NOP指令，
;   而预取队列中的NOP指令不会
;   改变。

cpuq_method1:
        mov     ax, cs                  ; 初始化代码段
        mov     es, ax                  ;   中的队列
        mov     cx, QSIZE               ; 队列大小
        mov     di, offset qdata        ; 队列位置
        mov     al, 90h                 ; 在开始前将所有的队列设置为NOP（空）指令
        cld
        rep     stosb                   ; 所有队列字节为NOP

        xor     bx, bx                  ; 计数起始值
        mov     cx, QSIZE               ; 修改的字节数
        mov     di, offset qdata+QSIZE-1 ; 指向修改地址的指针
        std                             ; 以后往前写
```

```
        mov     al, 80h
        out     70h, al                 ; 禁止 NMI
        cli                             ; 禁止中断
        mov     ax, 43h                 ; 43h = "inc bx"
        mov     dl, 1                   ; 除数 1 (对AL无影响)
        jmp     short upqflush          ; 清空预取队列
ALIGN 16                                ; 对齐代码边界
upqflush:
        div     dl                      ; 在预取队列中装入大量的
        div     dl                      ;   慢指令
        div     dl
        div     dl
        div     dl
        div     dl
        div     dl
        div     dl
        rep     stosb                   ; al 变成 es:[di+39h] thru [di]
qdata   db      QSIZE dup (90h)         ; NOP 指令
        xor     al, al
        out     70h, al                 ; 开放 NMI
        sti
        cld
        mov     ax, QSIZE+2             ; 大小（加上2，因为REP STOSB）
        sub     ax, bx                  ; ax = 队列大小
cpuq_exit2:
        jmp     cpuq_exit

;   下面的方法检查是否关闭了A20。如果是，
;   则采用一种方法使CPU不要更新预取
;   队列，这样我们可以获得精确的大小。通过
;   执行内存前64KB内的代码，并且
;   利用段值FFFF来改变代码可以实
;   现这一点。如果设计合适，写操作会
;   翻转回到低端内存（这就是为什么要关闭
;   A20的原因），而处理器并不知道我们正在写预
;   取队列区域！由于Pentium Pro会更新预取队列，
```

```
;   所以这种方法无效。

cpuq_method2:
        call    check_A20               ; A20开放了吗?
        xor     ax, ax                  ; 假定它开放了
        cmp     cx, 1
        je      cpuq_exit2              ; 如果开放了，返回，AX=0

        ; 保存我们将使用的低端内存的内容
        ; （从中断向量C0h到FFh）

        push    ds
        xor     ax, ax
        mov     ds, ax
        push    cs
        pop     es
        mov     si, 300h                ; 中断C0向量地址
        mov     di, offset int_backup
        mov     cx, 128
        cld
        rep     movsw                   ; 保存C0到FF的内容，在
                                        ;  后面可能会用到这些区域

        ; 将队列测试码移到低端内存

        cli                             ; 禁止中断
        push    cs
        pop     ds
        xor     ax, ax
        mov     es, ax
        mov     si, offset cpuq_code
        mov     di, 300h
        mov     cx, 128
        rep     movsw                   ; 将代码装入低端内存
        db      0EAh                    ; 远跳转至cpu_code, 位于低端
        dw      300h, 0                 ;  内存 0:300 处
```

```
        ; 在执行之前将下面的代码移动到低端内存

ALIGN 16                                ; 对齐代码边界
cpuq_code:
        mov     ax, cs                  ; 初始化代码段中的
        mov     es, ax                  ;   队列
        mov     cx, QSIZE               ; 队列大小
        mov     di, offset qdata2 - offset cpuq_code + 300h
                                        ; 队列位置
        mov     al, 90h                 ; 在开始前将所有的队列设置为NOP（空）指令
        cld
        rep     stosb                   ; 所有的队列字节设置为NOP

        xor     bx, bx                  ; 计数起始值
        mov     cx, QSIZE               ; 要修改的字节数
        mov     di, offset qdata2+QSIZE-1 - offset cpuq_code + 300h + 10h
                                        ; di 指向队列尾
        mov     ax, 0FFFFh
        mov     es, ax                  ; 段值改为 FFFFh
        std                             ; 从尾往回写
        mov     al, 80h
        out     70h, al                 ; 禁止 NMI
        mov     ax, 43h                 ; 43h = "inc bx"
        mov     dl, 1                   ; 除数1 (不影响AL)
        db      0EAh                    ; 硬编码跳转到upqflush
        dw      offset upqflush2 - offset cpuq_code + 300h, 0
                                        ; 清空预取队列
ALIGN 16                                ; 对齐代码边界
upqflush2:
        div     dl                      ; 在预取队列中装入
        div     dl                      ;   大量的慢指令
        div     dl
        div     dl
        div     dl
        div     dl
```

```
        div     dl
        div     dl
        rep     stosb                   ; al 变成 es:[di+39h] thru [di]
qdata2  db      QSIZE dup (90h)         ; NOP 指令
        xor     al, al
        out     70h, al                 ; 开放 NMI
        cld
        mov     ax, QSIZE+2             ; 大小 (因为有 REP STOSB，所以加上2)
        sub     ax, bx                  ; ax = 队列大小
        db      0EAh                    ; 硬码远跳转回到
        dw      offset cpuq_ret         ;  cpuq子程序
        dw      seg cpuq_ret

        ; AX = 队列大小  恢复中断向量内存

cpuq_ret:
        xor     si, si
        mov     es, si
        push    cs
        pop     ds
        mov     di, 300h                ; 中断C0向量地址
        mov     si, offset int_backup
        mov     cx, 128
        cld
        rep     movsw                   ; 恢复C0-FF中断
        sti                             ; 开放中断
        pop     ds

cpuq_exit:
        pop     es
        pop     di
        pop     dx
        pop     cx
        pop     bx
        ret
cpuQ    endp
```

代码 4-7 处理器版本检测器

CPU 的版本信息代表了这种型号改进的级别。和许多软件程序一样，每个 CPU 类型也有许多改进版（称为“代”，step），表示进行了一些小错误和硬件缺陷的修改。在某些情况下，做这些修改的目的是为了缩小 CPU 芯片的体积、加快运行的速度以及降低生产的成本。

一般程序很少需要知道所使用的 CPU 的版本，当然也有一些例外的情况。例如某些指令被 80386 所支持，但是在其他的版本中就被删除掉了，这就是一个例外的情况。如果要修正早期的 386 版本的缺陷，而此时已经用完了所有的 CPU 微指令空间，为了获得额外的空间，他们只好删除一些指令。当然如果你想使用这些指令，最好是测试一下，这个 386CPU 是否为第一代（A Step）。

这个程序使用了大量的技术来帮助标识 CPU 的版本信息。尽管在许多系统中我还没有发现使用这个程序会出现什么问题，但是我仍然建议，不要将这个程序用作商业用途。这个程序使用了许多内幕技巧，有可能不适用于将来的 CPU 和系统。

在运行这个程序之前，必须已经由 CPUVALUE 设置了 CPU_VAL 和 CPU_INFO。这个程序使用 DOS 字符串的输出函数来输出它的测试结果。

```
; ---------------------------------------------------------------------------
;    CPU STEP
;        如果支持CPUID指令，
;        则通过它获取信息。
;        否则检查BIOS是否提供了CPU的代信息。
;        显示的信息随结果改变。
;
;        调用:              ds:[cpu_val] 设置为 CPU类型
;                           ds:[cpu_info] 设置为 CPUID 标志
;                           ds:[cpu_mfg] = 7 或 8，如果是Cyrix
;
;        返回:              显示代信息
;
;        用到的寄存器:      eax, ebx, ecx, edx, di
;
;        子程序调用:        hex2ascii, xferbytes, hook_int6,
;                           restore_int6, bad_op_handler

bver        ad      ' CPU version from BIOS:. '
bvernum     ad                  .               ' h '
```

```
bvercom    ad      '                    '
           ad      '  CR, LE, ' $ '

novermg    ad      '  CPU version detection:                '
           ad      'Not supported in this BIOS or CPU       '
           ad      CR, lf, .&
idtxtmmg   ad      '  CPUID -- Vendor string:  '
idtext     ad      '                ', CR, LF, ' $ '
stepmsg    ad      '    Other info:                Stepping:  '
Stepval    ad      '    Model:  '
Modval     ad      '    Family:  '
Famval     ad      '    Max reg:  '
caseval    ad      '  ', CR, LF, ' $ '
featmsg    ad      '    Features  (.'
featval    ad      'XXXXXXXX) : '
featbo     ad      ' No      Floating-point processor in CPU chip  '
           ad      CR, LF
           ad      '
featbl     ad      'No   Support for enhanced virtual 80x86 mode '
           ad      CR, LF
           ad      '
           ad      'No   I/O breakpoints supported          '
           ad      CR, LF
           ad      '
           ad      'No     page size extensions supprted          '

           ad      CR, LF
           ad      '
           ad      'No   Time stamp counter support          '
           ad      CR,LF
           ad      '
           ad      'No   Model specific registers supported      '
           ad      CR, LF
           ad      '
           ad      'No      2 MB paging/36-bit addressing supported'
           ad      CR, LF
```

```
        ad      '
        ad      'No     Support for machine check exception     '
        ad      CR, LF
        ad      '
        ad      'No     Support for CMPXCHG8B instruction       '
        ad      CR,LF
        ad      '
        ad      'No     Internal APIC in CPU                    '
        ad      CR, LF
        ad      ' $ '

stepa   ad      '   CPU versino detection:                      '
        ad      'Steap A (earliest released versino)            '
        ad      CR, LF, `&`
```

代值及其含义表:

```
bvt     dw      303h
        db      ' (80386 step B1) '
bvte    dw      305h
        db      ' (80386 step DO) '
        dw      308h
        db      ' (80386 step D1) '
        dw      400h
        db      ' (80486 step AO) '
        dw      401h
        db      ' (80486 step B2) '
        dw      403h
        db      ' (80486 step B3) '
        dw      404h
        db      ' (80486 step B4) '
        dw      405h
        db      ' (80486 step b5) '
        dw      406h
        db      ' (80486 step B6) '
        dw      407h
```

```
        db      '(80486 sfep C1)'
        dw      2300h
        db      '(80386 sx)     '
        dw      OA301h
        db      '(386SLC step 1)'       : IBM
        dw      OA412h
        db      '(486SLC)       '       : IBM
        dw      OA422h
        db      '(486SLC2)      '       :IBM
        dw      OFFFFh
.586
cpustep proc    near
        cmp     [cpu_val], 3            ; 386 还是更新版本?
        jae     cpust_skp1
        jmp     cpust_no_BIOS_rev

cpust_skp1:
        test    [cpu_info], 1           ; 支持CPUID?
        jnz     cpust_idok              ; 如果是，则跳转

; Cyrix 6x86 会禁止 CPUID, 所以要开放它

        cmp     [cpu_mfg], 7            ; Cyrix ?
        je      cpust_isCyrix           ; 如果是，则跳转
        cmp     [cpu_mfg], 8            ; Cyrix ?
        jne     cpust_noid              ; 如果不是，则跳转
cpust_isCyrix:
        mov     ah, 0E8h
        call    read_cyrix_reg          ; 获取配置寄存器4（E8h）
        or      al, 80h                 ; 试着开放 CPUID
        call    write_cyrix_reg
        call    read_cyrix_reg          ; 看看第7位是否开着
        test    al, 80h
        jz      cpust_noid              ; 如果不支持CPOID则跳转

cpust_idok:
```

```
mov     eax, 0                  ; 获取供应商串功能
CPUID                           ;   执行 CPUID
mov     di, offset idtext
mov     eax, ebx
call    xfer_bytes              ; 从 eax 发送4个文本字节
mov     eax, edx
call    xfer_bytes              ; 从 eax 发送4个文本字节
mov     eax, ecx
call    xfer_bytes              ; 从 eax 发送4个文本字节
OUTMSG  idtxtmg                 ; 显示供应商串

mov     eax, 1                  ; 获取代信息
CPUID                           ; 保存在al中

; 保存特性标志和插入代标识

push    edx
push    eax
and     al, 0Fh                 ; 只需要低4位
call    hex2ascii               ; 将al转化成ASCII，并保存在bx中
mov     stepval, bh             ; 发送ASCII字节
pop     eax

; 插入CPUID指令获得的模式信息

push    eax
shr     al, 4                   ; 获取模式号
and     al, 0Fh                 ; 只需要低4位
call    hex2ascii               ; 将 al 转化成 ASCII, 并保存在bx中
mov     modval, bh              ; 发送 ASCII 字节
pop     eax

; 插入 CPUID 指令获得的系列信息

shr     ax, 8                   ; 获取系列号
and     al, 0Fh                 ; 只需要代4位
```

```
        call    hex2ascii           ; 将 al 转化成ASCII并保存在bx中
        mov     famval, bh          ; 发送 ASCII 字节

        ; 获取 CPUID 的 case 号

        mov     eax, 0              ; 获取CPUID指令支持情况的信号
        CPUID                       ;   并保存在al中
        call    hex2ascii
        mov     di, offset caseval
        cmp     bl, '0'             ; 如果高4位是0，则跳过
        je      cpust_skp1a
        mov     [di], bl            ; 插入高4位
        inc     di
cpust_skp1a:
        mov     [di], bh            ; 插入低4位
        OUTMSG stepmsg


        ; 输出由 CPUID 指令获得的特性（edx）

        pop     edx
        push    edx
        mov     cx, 4               ; 处理4字节的edx
        mov     di, offset featval

cpust_loop1:
        rol     edx, 8
        mov     al, dl
        call    hex2ascii           ; 将 al 转化成 ASCII, 并保存在bx中
        mov     [di], bx            ; 将 ASCII 字节插入到串中
        add     di, 2
        loop    cpust_loop1

        pop     edx
        mov     cx, 9               ; 显示9个特性位
        mov     di, offset featb0
```

```
cpust_loop2:                            ; 将特性设置为YES或NO
        test    dl, 1
        jz      cpust_skp2
        mov     word ptr [di], 'eY'     ; 插入 " Yes "
        mov     byte ptr [di+2], 's'
cpust_skp2:
        add     di, offset featb1 - offset featb0
        ror     edx, 1
        loop    cpust_loop2

        OUTMSG  featmsg

; 检查是否为 Cyrix, 使用I/O端口方法（这种方法为 Cyrix 所特有）

cpust_noid:
        cmp     [cpu_mfg], 7            ; Cyrix ?
        je      cpust_isCyrix2
        cmp     [cpu_mfg], 8            ; Cyrix ?
        jne     cpust_anyid             ; 如果不是，试一试BIOS方法
cpust_isCyrix2:
        mov     al, 0FEh
        call    read_cyrix_reg          ; 获取设备ID寄存器0 (FEh)
        push    ax
        call    hex2ascii
        mov     di, offset cidval
        cmp     bl, '0'                 ; 如果高4位是零，跳过
        je      cpust_skp2a
        mov     [di], bl                ; 插入高4位
        inc     di
cpust_skp2a:
        mov     [di], bh                ; 插入低4位
        OUTMSG  cid0msg                 ; 显示ID值
        pop     ax

        ; 接下来获取代和修正版号
```

```
        and     al, 0F0h
        cmp     al, 0F0h
        je      cpust_anyid             ; 无ID寄存器
        mov     al, 0FFh
        call    read_cyrix_reg          ; 获取设备ID寄存器1（FFh）
        cmp     al, 0
        je      cpust_anyid             ; 如果是零，则认为无效
        cmp     al, 0FFh
        je      cpust_anyid             ; 如果是0FFh，也认为无效
        push    ax
        and     al, 0F0h                ; 仅高4位
        call    hex2ascii               ; 将al转化成ASCII并保存在bx中
        mov     cstpval, bh             ; 发送ASCII字节
        pop     ax
        and     al, 0Fh                 ; 仅低于4位
        call    hex2ascii               ; 将al转化成ASCII并保存在bx中
        mov     crevval, bh             ; 发送ASCII字节
        OUTMSG cid1msg                  ; 显示代和修订版本号
        jmp     cpust_exit

; 如果前面用过CPUID指令，则完成

cpust_anyid:
        test    [cpu_info], 1           ; 支持 CPUID 吗?
        jz      cpust_bios              ; 如果不支持，则跳转
        jmp     cpust_exit              ; 完成

; 试一试利用BIOS功能获取ID
;   不幸的是，大多数制造商的BIOS
;   不支持这种BIOS功能

cpust_bios:
        mov     ax, 0C910h              ; BIOS 获取芯片修订版本号
        int     15h                     ;   并返回到cx中
        jc      cpust_chk_A_step        ; 如果成功则进位
```

```
        jcxz    cpust_chk_A_step        ; 0 = 不支持
        mov     al, ch
        mov     di, offset bvernum
        call    hex2ascii
        mov     [di], bx
        add     di, 2
        mov     al, cl
        call    hex2ascii
        mov     [di], bx
        add     di, 4
        mov     si, offset bvt          ; 获取标识代的文本
        mov     dx, offset bver         ; 输入信息

cpust_loop:
        cmp     word ptr [si], 0FFFFh   ; 末尾?
        je      cpust_out_msg           ; 如果是，则跳转
        cmp     [si], ax                ; 版本匹配?
        je      cpust_bios_match        ; 如果是，则跳转
        add     si, offset bvte - offset bvt
        jmp     cpust_loop

cpust_bios_match:
        mov     cx, offset bvte - offset bvt -1
        cld
        rep     movsb                   ; 转移CPU和代信息文本
        jmp     cpust_out_msg

; CPUID和BIOS都不支持获取CPU代信息。
;   这时可以用其他一些方法确定是386或486是
;   版本A还是更新的版本

cpust_chk_A_step:
        call    hook_int6               ; 挂起坏操作码中断
        cmp     [cpu_val], 3            ; 仅386?
        jne     cpust_rev_486           ; 如果是，则跳转
        mov     [badoff], offset cpust_no_rev ; 坏操作码
```

```
        mov     ax, 1
        mov     bx, 1
        mov     cl, 1
        mov     dx, 1
        db      0Fh, 0A6h, 0DAh      ; xbts  bx, dx, ax, al
        nop                          ;  (仅386A级有效)
        nop
        OUTMSG  stepa                ; 使用A级信息
        jmp     cpust_restore

cpust_rev_486:
        cmp     [cpu_val], 4         ; 仅486?
        jne     cpust_no_rev         ; 如果不支持则跳转
        mov     [badoff], offset cpust_no_rev ; 坏操作码
        db      0Fh, 0A6h, 0DAh      ; cmpxchg   bx, dx
        nop                          ;  (仅486 A级有效)
        nop
        OUTMSG  stepa                ; 使用A级信息
        jmp     cpust_restore

cpust_no_rev:
        OUTMSG  novermg              ; BIOS信息不支持
cpust_restore:                       ; 恢复中断向量6
        call    restore_int6         ; 恢复中断6处理程序
        jmp     cpust_exit

cpust_no_BIOS_rev:
        mov     dx, offset novermg   ; BIOS信息不支持
cpust_out_msg:
        mov     ah, 9
        int     21h
cpust_exit:
        ret
cpustep endp
```

代码例 4-8 处理器模式探测器

这个程序用来识别 CPU 是否处于保护模式下。若 CPU 支持保护模式，那么就会检测该 CPU 的模式类型（实模式还是保护模式）以及当前的优先级。如果 CPU 是 80386 或更晚的类型，并且处于保护模式下，就会认为 CPU 处于 V86 模式（需要修改这个程序以包括一个保护模式程序）。CPU 有一个虚 86 模式位，但是对于运行在 V86 模式下的程序来说，这一位是不可读的。

CPUMODE 程序要求在此之前已经运行了 CPUVALUE，并且将 CPU 类型保存在了变量 CPU_VAL 中。

```
; ------------------------------------------------------------------------------
;    CPU 模式
;        检查286+CPU是位于实、保护还是V86
;        模式，并且假定，如果80386+处理器
;        位于保护模式，则一定就位于V86模式
;
;        调用:       ds:[cpu_val] 设置为CPU类型
;
;        返回:        al = 0 不支持保护模式
;                          1 实模式
;                          2 保护模式
;                          3 V86模式
;                     ah = 优先级0到3
;
;        用到的寄存器:       ax

.386P                                   ; 支持286/386指令

cpumode proc    near
        push    cx
        xor     cx, cx                  ; 假定不支持保护模式
        cmp     [cpu_val], 2            ; 286+CPU?
        jb      cpum_Exit               ; jump if not
        mov     cx, 1                   ; 假定是实模式
        smsw    ax                      ; 获取机器状态字
        test    ax, 1                   ; 位于保护模式?
```

```
        jz      cpum_Exit               ; 如果不是则跳转（实模式）

cpu_not_real:
        mov     cl, 2                   ; 保护模式
        pushf
        pop     ax                      ; 获取标志
        and     ax, 3000h               ; 获取I/O优先级
        shr     ax, 12
        mov     ch, al                  ; 保存优先级
        cmp     [cpu_val], 2            ; 如要是286，那么位于保护模式
        je      cpum_Exit               ; 如果是，则跳转

; 在386+上我们必须考虑是否为V86模式。
;  下面的四行代码看起来似乎可以正确地
;  检测V86模式。其实它起不了作用。因为
;  PUSHFD指令在将VM位推入堆栈之前会
;  清除该位。在386和486上没有公开这一点，
;  但在Pentium/Pentium pro上公开了这一点。

;       pushfd                          ; 将标志保存在堆栈
;       pop     eax                     ; 获取扩展标志
;       test    eax, 20000h             ; V86 模式?
;       jz      cpum_out_mode           ; 如不是则跳转

        mov     cl, 3                   ; 返回 V86 模式

cpum_Exit:
        mov     ax, cx                  ; 返回状态
        pop     cx
        ret
cpumode endp
.8086
```

代码例 4-9 供应商标识

当芯片不是 Intel 类型时，这个程序将识别出 CPU 的供应商。它可以检测出 AMD、Cyrix、

IBM、Intel、NEC 以及 NexGen 类型。

AMD 生产了一个同 Intel 一样的 386，这样就很难将它们区分开来。这个程序也不可能将具有 33MHz 和 25MHz 不同速度的型号区分开来。然而，只有 AMD 生产的 40MHz386，也只有 Intel 生产低于 25MHz 的类型除外。这个程序并不提供最佳的确定信息，但是在这个程序之后接着使用 CPUSPEED 程序，可以获得进一步的正确信息。我没有在 CPUVENDOR 程序中将 CPUSPEED 程序的测试包括进来。因为后者要用到前者的运行结果。程序 CPUTYPE，将显示这些程序的运行结果。它也使用 CUPSPEED 的测试结果来检测 Intel 和 AMD CPU 的区别。

```
; ---------------------------------------------------------------------------
;    CPU 供应商标识子程序
;       通过检查一系列供应商芯片的独特之处
;       来确定CPU制造商
;
;       调用:           ds:[cpu_val] 设置为CPU类型
;                       ds:[cpu_info] 设置为CPU附加信息
;                       ds:[cpu_inst] 设置为指令集
;
;       Returns:        al = 供应商号
;                            0 = 未知，8088 或 80286
;                            1 = CPUID串上供应商未知
;                            2 = NEC V20/V30
;                            3 = Intel 或 AMD，非IBM和Cyrix
;                            4 = Intel
;                            5 = AMD，仅限于40MHz的386
;                                    or from CPUID
;                            6 = IBM，仅限于带RDMSR的386/486
;                            7 = Cyrix 或 TI，不支持 UMOV
;                            8 = Cyrix (或586+)
;                            9 = NexGen
;                           10 = UMC
;                       将CPUID串转移到ds:[idstring]
;       用到的寄存器      ax、bx、ecx、dx
;       子程序调用:       hook_int6,restore_int6,bad_op_handler
;                       hook_intD,restore_intD
```

```
.586P                                   ; 支持CPUID指令

vendnam db      4,  5, 'INTEL'          ; 用来找寻CPUID中的供应商信息
        db      8,  5, 'CYRIX'          ; 第一个字节是供应商号
        db      9,  6, 'NEXGEN'         ; 第二个字节是串长度
        db      10, 3, 'UMC'            ; 其他字节是大写形成的串
        db      5,  3, 'AMD'
        db      0

cpuvendor proc  near
        push    ds
        push    es

        mov     al, 2                   ; 假定为NEC
        test    [cpu_info], 2           ; V20/V30? (仅NEC生产)
        jnz     cpuv_exit               ; 若是，则跳转
        mov     al, 0                   ; 假定供应商未知
        cmp     [cpu_val], 2            ; 获取CPU号
        jbe     cpuv_exit               ; 如果是286或更低级则跳转
        test    [cpu_info], 1           ; CPUID有效?
        jz      cpuv_NexGen             ; 若非，则跳转

        mov     eax, 0                  ; 获取供应商串功能
        CPUID                           ; 将CPUID信息保存到ebx、ecx、edx中
        mov     di, offset idstring
        mov     eax, ebx
        call    xfer_bytes              ; 从eax发送4个文本字节
        mov     eax, edx
        call    xfer_bytes              ; 从eax发送4个文本字节
        mov     eax, ecx
        call    xfer_bytes              ; 从eax发送4个文本字节

; 在已知的制造商中搜索CPUID串
```

```
        mov     ax, ds
        mov     es, ax
        push    cs
        pop     ds
        mov     si, offset vendnam
        mov     di, offset idstring
        mov     cx, 12                  ; 要查看12个字节

cpuv_loop1:
        cmp     byte ptr [si], 0        ; 没有其他的串要检查
        je      cpuv_skp3               ; 若没有发现则跳转

        mov     dl, [si]                ; 保存供应商号
        inc     si                      ; 指向长度字节
        call    find_string2            ; 看看串[si]是否出现在
                                        ; 位于es:[di]的CPUstring中
        jc      cpuv_skp2               ; 如果发现，则跳转
        mov     al, [si]
        xor     ah, ah
        add     si, ax                  ; 移到下一个串
        inc     si
        jmp     cpuv_loop1              ; 试一试下一个供应商串

cpuv_skp2:
        mov     al, dl                  ; 设置供应商号
        jmp     cpuv_exit

cpuv_skp3:
        mov     al, 1                   ; 未知的供应号
        jmp     cpuv_exit

; 检查是否为NecGen。
; 其指令类型属于586，但是指令集是486类型的，
; 并且不支持对齐检查
```

```
cpuv_NexGen:
        cmp     [cpu_val], 5          ; 586 系列?
        jne     cpuv_Cyrix            ; 若非，则跳转
        cmp     [cpu_inst], 4         ; 支持486指令?
        jne     cpuv_Cyrix            ; 若非，则跳转
        test    [cpu_info], 4         ; 支持AC吗?
        jnz     cpuv_Cyrix            ; 若是，则不是NexGen
        mov     al, 9                 ; 设置为NexGen
        jmp     cpuv_Exit

; 检查是否为Cyrix CPU—其除法
;   指令不影响标志位。只有在不支持
;   CPUID时才需要进行这项测试

cpuv_Cyrix:
        xor     ah, ah
        sahf                          ; 清除标志
        mov     ax, 10                ; 除法用到的具体值不重要
        mov     cl, 4                 ;
        div     cl                    ; 执行一次除法
        lahf                          ; 获取标志
        and     ah, 0FDh              ; 忽略第1位
        cmp     ah, 0                 ; 标志为零吗?
        jne     cpuv_skp4             ; 若非，则不清楚它是什么

        mov     al, 7                 ; 设置为Cyrix或TI
        cmp     [cpu_val], 4          ; 如果是486以上的，则必是Cyrix
        jbe     cpuv_Cyrix2
        mov     al, 8                 ; 显然是Cyrix
cpuv_Cyrix2:
        jmp     cpuv_Exit

; 检查是否为老式的Cyrix/TI CPU。
;  Cyrix不需要也不支持UMOV指令（Pentiun+
;  也不支持UMOV，但是
;  已经通过CPUID检查出了Pentium供应商）
```

```
cpuv_skp4:
        call    hook_int6               ; 挂起坏操作码中断
        call    hook_intD               ; 挂起一般保护性错误
        mov     [badoff], offset cpuv_badop  ; 坏码处理地点
        mov     al, 05Ah
        mov     bh, 0A5h
        clc                             ; 清除进位标志
        db      0Fh, 10h, 0F8h          ; umov al, bh
        db      90h, 90h
        jc      cpuv_badop              ; 不应设置进位
        cmp     al, bh
        jne     cpuv_badop              ; al应等于= bh
        jmp     cpuv_try_IBM            ; 支持UMOV，可能是Intel/AMD/IBM

; 386+CPU不支持UMOV，所以一定是Cyrix

cpuv_badop:
        mov     al, 7
        cmp     [cpu_val], 4            ; 如果486以上，则必是Cyrix
        jbe     cpuv_Cyrix3
        mov     al, 8                   ; 明显是Cyrix
cpuv_Cyrix3:
        jmp     cpuv_restore

; 检查是否为IBM CPU。
;   仅IBM芯片支持读模式专用寄存器（RDMSR）时ecx=1000h。
;   警告: 在Windows 增强模式下这样做时会关闭Windows。
;   运行Windows时CPUTYPE
;   不会调用这个程序

cpuv_try_IBM:
        mov     [badoff], offset cpuv_badop2 ; 坏码处理地点
        mov     ecx, 1000h
        RDMSR                           ; 读模式专用寄存器
        db      90h, 90h                ; safety NOPs
```

```
        mov     al, 6                   ; RDMSR起作用，一定是IBM
        jmp     cpuv_restore

cpuv_badop2:
        mov     al, 3                   ; 不是IBM，可能是Intel或AMD

cpuv_restore:
        call    restore_intD            ; 恢复中断D处理程序
        call    restore_int6            ; 恢复中断6向量
cpuv_exit:
        pop     es
        pop     ds
        ret
cpuvendor endp
.8086                                   ; 返回到8086指令
```

代码例 4-10 测量 CPU 速度

这个程序用来测量 CPU 的内部速度，单位 MHz。由于大多数供应商类型的微指令彼此不同，所使用的指令定时时间也千差万别，所以必须先由 CPUVENDOR 识别出供应商来。然后这个程序选择供应商专用的比例因子，以补偿这些差别。

这个程序在测量 CPU 内部速度方面相当出色，并且这项测量与外部高速缓存的设计无关。对于那些内部运算快于外部总线速度的 CPU 来说，这个程序返回其内部 CPU 速度。例如 100MHz 的 486DX2 芯片可以在其内部以 100MHz 运行，但是在外部只能以 33MHz 运行。CPUSPEED 程序将报告该 CPU 是 100MHz 的。

对于将来的 CPU 供应商和 CPU，可能会返回一个比实际情况要快的 CPU 速度，除非标识出了供应商并作了适当的补偿。

这个程序用到了 CLI 指令来关闭所有的中断，以免影响计时。然而一个 V86 环境，例如 Windows 增加模式，仍然会模拟 CLI 和 STI 指令，并维持虚中断标志。这时，通常仍允许中断，即使是在执行 CLI 之后。

```
; ----------------------------------------------------------------------
;    CPU 速度确定子程序
;        通过精确度量一小段循环来确定
;        CPU速度
;
```

```
;       调用:           ds:[cpu_val] 设置为CPU类型
;                       ds:[cpu_mfg] 设置为CPU供应商
;
;       返回:           ax = 速度, 单位MHz
;                       bx = 原始计时值
;
;       用到的寄存器:    ax, bx, cx
; 速度数据表—第1个字是循环次数。较大的数字
; 需要更长时间，也用于校验校快的CPU。
; 第2个字是以MHz为结尾的一个比例因子，
; 例如，对主频为33MHz的80486来说
; 第二个字是16550=（33*2006拍）/4。
; 每个供应商都有一个表。
; 如果第1个字是零，则是未知的（或在大多数场合，
; 供应商/处理器不存在）。带*号的项表示对应于Intel的标准。

TIMING_TYPE   equ     8             ; 供应商的CPU号
; 未知类型的定时方法 (0)
type0   dw    1,10848               ; *8088—循环次数，比例因子
        dw    1,10848               ; *80186 (5MHz=~8345拍)
        dw     2,3234               ; *80286 (12MHz=~1035拍)
        dw   10,16200               ; *80386 (33MHz=~1917拍)
        dw   10,16550               ; *80486 (33MHz=~2006拍)
        dw   20,34318               ; *Pentium (60HMz=~2269拍)
        dw   20,30935               ; *Pentium Pro
        dw       0,0                ; *P7
; 未知类型的定时方法 (1)
type1   dw    1,10848               ; *8088—循环次数，比例因子
        dw    1,10848               ; *80186 (5MHz=~8345拍)
        dw     2,3234               ; *80286 (12MHz=~1035拍)
        dw   10,16200               ; *80386 (33MHz=~1917拍)
        dw   10,16550               ; *80486 (33MHz=~2006拍)
        dw   20,34318               ; *Pentium (60HMz=~2269拍)
        dw   20,30935               ; *Pentium Pro
        dw       0,0                ; *P7
; NEC V20/V30类型的定时方法 (2)
```

```
type1       dw      1,10848             ; *8088—循环次数，比例因子
            dw      1,10848             ; *80186 (5MHz=~8345拍)
            dw       2,3234             ; *80286 (12MHz=~1035拍)
            dw     10,16200             ; *80386 (33MHz=~1917拍)
            dw     10,16550             ; *80486 (33MHz=~2006拍)
            dw     20,34318             ; *Pentium (60HMz=~2269拍)
            dw     20,30935             ; *Pentium Pro
            dw          0,0             ; *P7
; Intel 类型的定时方法 (3)
type1       dw      1,10848             ; *8088—循环次数，比例因子
            dw      1,10848             ; *80186 (5MHz=~8345拍)
            dw       2,3234             ; *80286 (12MHz=~1035拍)
            dw     10,16200             ; *80386 (33MHz=~1917拍)
            dw     10,16550             ; *80486 (33MHz=~2006拍)
            dw     20,34318             ; *Pentium (60HMz=~2269拍)
            dw     20,30935             ; *Pentium Pro
            dw          0,0             ; *P7
; 其他供应商产品的定时方法插入此处（见CPUTYPE源代码）

cpuspeed proc    near
         push    dx
         push    si
         mov     ah, TIMING_TYPES
         shl     ah, 1
         shl     ah, 1                  ; 每个入口4个字节
         mov     al, [cpu_mfg]          ; 获取CPU制造商
         mul     ah                     ; ax = 表索引
         mov     si, ax
         add     si, offset type0       ; 值表

         mov     al, [cpu_val]          ; 获取CPU号
         xor     ah, ah
         shl     ax, 1
         shl     ax, 1                  ; 每个入口4个字节
         add     si, ax                 ; 将SI指向CPU值
```

```
; 设置时钟2对指令执行计时

        mov     al, 0B0h            ; 时钟2命令, 模式0
        out     43h, al             ; 发送命令
        IODELAY
        mov     al, 0FFh            ; 计数器值FFFF
        out     42h, al             ; 向计数器发送最低有效位
        IODELAY
        out     42h, al             ; 向计数器发送最高有效位
        IODELAY

; 关闭所有中断, 包括NMI,
; 以免中断影响计时

        cli                         ; 关闭中断
        mov     al, 80h
        out     70h, al             ; 关闭NMI
        IODELAY
        in      al, 61h             ; 读取当前内容
        IODELAY
        or      al, 1               ; 将门控位设置为开
        out     61h, al             ; 开启时钟（开始计时）
        xor     dx, dx
        mov     bx, 1
        mov     ax, [si]

; 该循环执行一串除法指令

cpus_loop1:
        mov     cx, 10h             ; 循环值
cpus_loop2:
        div     bx                  ; ax = dx:ax/1   dx=rem
        div     bx                  ; (lots of cycles per inst)
        div     bx
        div     bx
        div     bx
```

```
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        loop    cpus_loop2
        dec     ax
        jnz     cpus_loop1          ; CPU循环x次

; 循环结束时停止计数器，重新开放
; 中断

        in      al, 61h             ; 读取当前内容
        IODELAY
        and     al, 0FEh            ; 关闭门控位
        out     61h, al             ; 关闭计数器
        xor     al, al
        out     70h, al             ; 开放NMI
        sti                         ; 开放中断

; 读取时钟内容后，就可以确定指令
; 执行周期

        mov     al, 80h             ; 锁存输出命令
        out     43h, al             ; 发送命令
        IODELAY
        in      al, 42h             ; 获取计数器的LSB
        IODELAY
        mov     dl, al
        in      al, 42h             ; 获取计数器的MSB
        mov     dh, al              ; dx = 计数器值
        mov     ax, 0FFFFh          ; 起始值
```

```
        sub     ax, dx                  ; ax = 计数周期
        mov     cx, ax
        mov     bx, ax                  ; 保存寄存器，以便退出时恢复
        mov     ax, cx
        cmp     word ptr [si+2], 0      ; 不需要调整因子?
        je      cpus_skp2               ; 带值返回

; 由于不同类执行指令时会采用不同的定时方法
; （例如，由于值不同，8088执行除法操作时可能会占
; 用144到162个时钟，而Pentium只须25个时钟），
; 所以需要对每个CPU类型和供应商进行补偿

        mov     ax, [si+2]              ; 获取因子
        xor     dx, dx
        shl     ax, 1
        rcl     dx, 1
        shl     ax, 1
        rcl     dx, 1                   ; 因子* 4
        div     cx                      ; 调整因子 (ax=dx:ax/cx)

; 在AX中返回CPU速度，单位为MHz

cpus_skp2:
        pop     si
        pop     dx
        ret
cpuspeed endp
```

代码例 4-11　处理器高速缓存检测器

这个程序返回 386 或更晚 CPU 的内部 CPU 高速缓存状态。奇怪的是，某些 386，例如 IBM，也有一个高速缓存。常常我第一次努力的结果只是徒劳。不同的供应商采用不同的手段控制高速缓存。不清楚他们为什么都模仿 Intel。通常由系统 BIOS 来控制高速缓存。所以，这些差别对在多数程序来说并不重要，除非你要编写 BIOS。

CPUMODE 程序要求在此之前已经执行过 CPUVALUE。并且 CPU 类型保存在了变量 CPU_VAL 中。另外，还要已经运行过了 CPUVEDOR，该供应商类型保存到了 CPU_MFG

中。

```
;--------------------------------------------------------------------
;    CPU 缓存
;        检查是否支持CPU缓存。
;        Cyrix, IBM, 和Intel 处理缓存信息的方法
;        有些差别
;
;        调用:          ds:[cpu_val] 设置为CPU类型
;                       ds:[cpu_info] 设置为CPUID标志
;                       ds:[cpu_mfg] 设置为制造商标识
;
;        返回:          al = 0 如果无缓存
;                            1 如果禁止缓存
;                            2 如果开放缓存, 无写通
;                            3 如果开放缓存, 写通
;                            4 如果开放缓存, 回写
;
;        用到的寄存器:  eax, bx

.586P                                   ; 允许486指令
cpu_cache proc   near
        xor     bl, bl                  ; 无缓存
        cmp     [cpu_val], 3            ; 仅在486+和某些386中有缓存
        jb      cpuc_type               ; 如果无则跳转
        mov     bh, [cpu_mfg]
        cmp     bh, 6                   ; IBM ?
        je      cpuc_ibm                ; 如果是则跳转
        cmp     bh, 7                   ; Cyrix ?
        je      cpuc_cyrix              ; 如果是则跳转
        cmp     bh, 8                   ; Cyrix ?
        je      cpuc_cyrix              ; 如果是则跳转

; 使用Intel 的检测方法（仅486+）

        cmp     [cpu_val], 4            ; 仅在486+中有缓存
```

```
        jb      cpuc_type               ; 如果无则跳转
        mov     eax, cr0                ; 获取控制寄存器
        inc     bl                      ; 有内部缓存
        test    eax, 40000000h          ; 关闭缓存?
        jnz     cpuc_type               ; 如果是则跳转
        inc     bl                      ; 返回缓存开放
        test    eax, 20000000h          ; 写通?
        jnz     cpuc_type               ; 如果不是则跳转
        inc     bl                      ; 支持写通
        jmp     cpuc_type

; 使用IBM方法检测缓存有没有开放

cpuc_ibm:
        mov     ecx, 1000h              ; 获取寄存器1000h
        RDMSR                           ; 读模式专用寄存器
        mov     bl, 1                   ; 有一个缓存，确认关闭
        test    eax, 80h                ; 开放缓存?
        jz      cpuc_type               ; 如果不是则跳转
        mov     bl, 3                   ; 有写通的缓存
        jmp     cpuc_type

; Cyrix类似于Intel，但是如果开放缓存，则一定是写通的

cpuc_cyrix:
        mov     eax, cr0                ; 获取控制寄存器
        mov     bl, 1                   ; 有一个内部缓存
        test    eax, 40000000h          ; 关闭缓存
        jnz     cpuc_type               ; 如果是则跳转
        mov     bl, 3                   ; 开放写通

; 如果开放缓存，那么老式设计采用写通，
; 而新式（如大多数586）支持回写方式。

cpuc_type:
        cmp     bl, 3                   ; 写通开放?
```

```
        jne     cpuc_Exit               ; 如果不是则跳转
        cmp     [cpu_val], 4            ; 检查CPU型号
        jb      cpuc_exit               ; 如果是386（无回号）则跳转
        ja      cpuc_586                ; 如果是586+则跳转
        cmp     [cpu_mfg], 5            ; AMD 486增强型
        jne     cpuc_Exit               ; 如果不是则跳转
        test    [cpu_info], 1           ; 仅只有CPUID的AMD 486
        jz      cpuc_Exit               ; 如果不是则跳转
        jmp     cpuc_writeback          ; 是回写缓存
cpuc_586:
        cmp     [cpu_mfg], 9            ; NexGen?
        je      cpuc_Exit               ; 如果是，则不回写
cpuc_writeback:
        inc     bl                      ; 设置为写回选通
cpuc_Exit:
        mov     al, bl                  ; 返回状态，并保存在al中
        ret
cpu_cache endp
.8086
```

代码例 4-12 处理器数据高速缓存的大小分析

一旦确定了 CPU 带有一个内部高速缓存，并且开通了该高速缓存，本程序就可以确定高速缓存的大小。对于带有一个高速缓存的 386 和 486 来说，该高速缓存既用做数据高速缓存，又用作代码高速缓存。然而，在 Pentium 和 Pentium Pro 的 CPU 上，两者已经分开。

该测试程序是通过一次比一次读取更大的数据块实现对高速缓存的测试的。首先，这个程序读一个 512 字节的块，读 128 次。接下来，所读块的大小加倍，但次数减半。读完 64K 的块后，比较每个块所花的时间，看看有没有显著的差别，当 CPU 必须开始使用外部数据而不是内部高速缓存，读取数据块的时间将至少要多花 5%。对于一个慢速系统或者如果系统没有外部高速缓存，那么第一次使用外部内存的时钟要比使用内部 CPU 高速缓存慢两倍以上。

如果 CPU 没有高速缓存或者外部高速缓存的速度和内部缓存的速度一样快，那么高速缓存的大小将返回零。这个程序检查时至少要保证计时的差别在 5%以上。80386 没有内部高速缓存，或者 80486 的高速缓存被禁止时，就会出现这种情况。

```
; ---------------------------------------------------------------------------
;    确定数据缓存的大小
;        确定内部CPU数据缓存的大小。如果
;        CPU没有内部缓存，则结果无效
;
;        调用:           无
;
;        返回:           ax = 缓存大小*1KB
;
;        用到的寄存器:   ax, bx, cx, si, di
;
;        子程序调用:     read_n_time
timings     dw  0                   ; .5K 定时
            dw  0                   ; 1K
            dw  0                   ; 2K
            dw  0                   ; 4K
            dw  0                   ; 8K
            dw  0                   ; 16K
            dw  0                   ; 32K
            dw  0                   ; 64K

cache_d_size proc  near
        mov     di, offset timings  ; 计时数据
        mov     bh, 8               ; 获取8组值
        mov     bl, 0               ; 重新读取的次数
                                    ;  (bl=0 表示读 256 次)
        mov     cx, 256             ; 初始值为0.5K定时 (256字)

chd_loop:
        call    read_n_time         ; 从 ds:0,处开始读取cx个字
                                    ;    在ax中返回持续时间

        mov     [di], ax            ; 保存值
        add     di, 2
        shr     bl, 1               ; 读取的字数除以2
        cmp     bl, 0               ; 如果是第一次，则bl=0 (256次)
```

```
        jne     chd_skp1
        mov     bl, 128                 ; 第二次使用128
chd_skp1:
        shl     cx, 1                   ; 字节数*2
        dec     bh
        jnz     chd_loop

        mov     bp, 1                   ; 缓存大小 1=1KB
        mov     dx, bp
        mov     di, offset timings
        mov     cx, 7                   ; 八次计时
        xor     bx, bx                  ; 第一次计时—最小值

chd_loop3:
        mov     ax, [di]
        sub     ax, [di+2]              ; 计时差
        jns     chd_not_neg
        not     ax                      ; 获取正的计时差
chd_not_neg:
        cmp     ax, bx                  ; 哪个大?
        jb      chd_skp2
        mov     bx, ax                  ; 保存新的大的值
        mov     dx, bp                  ; 保存缓冲大小
chd_skp2:
        shl     bp, 1
        add     di, 2
        loop    chd_loop3
        shr     dx, 1                   ; 调整恢复原样

; 检查时间差是否足够大
;   (例如，最大差值必须超过首次计时
;   值的5%)

        mov     cx, dx                  ; 保存大小
        mov     ax, bx                  ; bx
        mov     bx, [timings]           ; 获取首次计时值
```

```
        xor     dx, dx
        div     bx                      ; ax = ax/bx, dx = 余数
        or      ax, ax
        jnz     chd_ok                  ; 如果 ax > 0, 则有效
        cmp     dx, 5                   ; 如果 > 5% 行
        ja      chd_ok                  ; 如果是，则跳转
        xor     cx, cx                  ; 返回零
chd_ok:
        mov     ax, cx
        ret
cache_d_size endp


; ---------------------------------------------------------------------------
;   读取与计时
;       从内存 ds:0.处开始读取CX个字。
;       重复读取数据BL次。
;       使用硬件时钟2来对读取块所需的时间。
;       测试期间关闭中断以提高测试结果的准确性。
;
;       调用:           cx = 要读取的字数
;                       bl = 重复读取的次数
;                       bl = 0 表示 256 次
;
;       返回:           ax = duration of reads (ax*838nS = time)
;
;       用到的寄存器:   ax

read_n_time  proc       near
        push            dx
        push            si
        mov             al, 0B0h        ; 时钟2命令，模式0
        out     43h, al                 ; 发送命令
        IODELAY
        mov     al, 0FFh                ; 计数值FFFF
        out     42h, al                 ; 向计数器发送LSB
```

```
        IODELAY
        out     42h, al             ; 向计数器发送MSB
        IODELAY
        cli                         ; 禁止中断
        in      al, 61h             ; 读取当前内容
        IODELAY
        or      al, 1               ; 设置门位为开
        out     61h, al             ; 激活计数器
        cld
        push    bx

read_again:
        push    cx
        xor     si, si              ; 从0开始
        rep     lodsw               ; 读取块
        pop     cx
        dec     bl                  ; 循环次数
        jnz     read_again

        pop     bx
        in      al, 61h             ; 读取当前内容
        IODELAY
        and     al, 0FEh            ; 关闭门控位
        out     61h, al             ; 关闭计数器
        sti                         ; 开放中断

        mov     al, 80h             ; 锁存输出命令
        out     43h, al             ; 发送命令
        IODELAY
        in      al, 42h             ; 获取计数器的LSB
        IODELAY
        mov     dl, al
        in      al, 42h             ; 获取计数器的MSB
        mov     dh, al              ; dx = 计数器值
        mov     ax, 0FFFFh          ; 起始值
        sub     ax, dx              ; ax = 计数周期
```

```
        pop     si
        pop     dx
        ret
read_n_time endp
```

代码 4-13　内幕指令的测试

下面的程序用来测试指定 CPU 已知的内幕指令。这句话似乎有点自相矛盾，但是我所指的是那些我在第 3 章中讨论过的内幕指令。对每个内幕指令都进行测试，以验证它是否像第 3 章所指出的那样工作。

为了避免运行该测试时挂起 CPU，这个程序先显示将要测试的指令。测试完成后，它也显示测试的结果。为了避免挂起系统，这个程序挂起了坏码中断和双重错误中断。

尽管某个未知指令可能会挂起系统，但是在测试许多系统时，我还没有发现这个程序会造成冲突。

运行 CPUUNDOC，就可以知道在你的 CPU 上的运行结果。屏幕上会显示如图 4-3 所示的信息。

```
内幕指令测试 v2.00（c）1994, 1996 FVG
本 CPU 内幕指令总结
可能支持:    UMOV              （0Fh, 10h - 13h）    测试失败
应该支持:    RDTSC             （0Fh, 31h）          测试成功
应该支持:    SHL     AL,imm    （C0h, reg=110）      测试成功
应该支持:    SHL     AX,imm    （C1h, reg=110）      测试成功
应该支持:    SHL     AL,1      （D0h, reg=110）      测试成功
应该支持:    SHL     AX,1      （D1h, reg=110）      测试成功
应该支持:    AAM     imm       （D4h, 8）            测试成功
应该支持:    AAD     imm       （D5h, 10h）          测试成功
应该支持:    SETALC            （D6h）               测试成功
应该支持:    ICEBP             （F1h）               测试成功
应该支持:    TEST    AL,1      （F6h, reg=001）      测试成功
应该支持:    TEST    AX,1      （F7h, reg=001）      测试成功
```

图 4-3　在 Intel Pentium 系统上运行 CPUUNDOC 的测试结果

被测试的指令与 CPU 有关。当系统不在实模式下时，就不会测试某些指令，比如 ICEBP，因为这样做会出现问题。参看 CPUUNDOC 以获得完整的列表。在下面的子程序 CHKUNDOC 中显示了所有上述内容。

```
;--------------------------------------------------------------------------
;    检查内幕指令
;        检查并显示内幕指令。
;        测试了大多数的内幕指令来确保指令
;        按所描述的那样工作。
;        显示了测试结果。
;
;        如果已知一个系列的某些或全部
;        CPU支持某些特殊的内幕指令，
;        那么只在这些CPU上测试它们。
;        例如，只在8088/8086的CPU上测试POPCS，
;        在其他所有的CPU上这条指令都与其他指令相冲突。
;
;        调用:          ds:[cpu_val] 设置为CPU类型
;                       ds:[cpu_info] 设置为CPU信息
;                       ds:[cpu_prot] 设置为CPU保护
;                           状态
;
;        返回:          显示内幕信息
;
;        用到的寄存器:  ax, bx, cx, dx
cpu_val     db     0                    ; CPUVALUE的CPU值
cpu_info    db     0                    ; CPUVALUE标志
cpu_prot    db     0                    ; 保护状态

ud_flags        db      0               ; 如果中断1发生则位1=1

old_int6_seg    dw      0               ; 向量（坏操作码）
old_int6_off    dw      0               ; 如果坏偏移量中断6被
                                        ; 调用则临时返回偏移量
; 未公开指令的文本

ud_header   db  CR, LF
            db  '  Undocumented Instruction Summary for this CPU'
            db  CR, LF, '$'
ud_popcs    db  'Should support:    POP     CS      (OFh)           $'
ud_ldall2   db  'Should support:    LOADALL         (OFh, 05h)      $'
```

```
ud_ldall3   db  'Should  support:   LOADALL          (OFh, 07h)          $'
ud_umov     db  'Might   support :  UMOV             (OFh, 10h-13h)      $'
ud_rdtsc    db  'Should  support:   RDTSC            (OFh, 31h)          $'
ud_xbts     db  'Only 386 rev A:    XBTS             (OFh, A6h)          $'
ud_ibts     db  'Only 386 rev A:    IBTS             (OFh, A7h)          $'
ud_shlal3   db  'Should  support:   SHL     AL,imm   (C0h, reg=110)      $'
ud_shlax3   db  'Should  support:   SHL     AX,imm   (C1h, reg=110)      $'
ud_shlal    db  'Should  support:   SHL     AL,1     (D0h, reg=110)      $'
ud_shlax    db  'Should  support:   SHL     AX,1     (D1h, reg=110)      $'
ud_aam      db  'Should  support:   AAM     imm      (D4h, 8)            $'
ud_aad      db  'Should  support:   AAD     imm      (D5h, 10h)          $'
ud_setalc   db  'Should  support:   SETALC           (D6h)               $'
ud_icebp    db  'Should  support:   ICEBP            (F1h)               $'
ud_testal   db  'Should  support:   TEST    AL,1     (F6h, reg=001)      $'
ud_testax   db  'Should  support:   TEST    AX,1     (F7h, reg=001)      $'

ud_tested   db  ' Tested OK.' , CR, LF, '$'
ud_untest   db  ' Not tested.' , CR, LF, '$'
ud_untstv   db  ' Untested (V86 mode).' , CR, LF, '$'
ud_failed   db  ' Failed test.' , CR, LF, '$'
ud_notA     db  ' Failed, not rev A.' , CR, LF, '$'

chk_undoc proc  near
        OUTMSG  ud_header           ; 未公开的概要表头

; 首先，如果是286+，则陷入坏操作码中断6

        cmp     [cpu_val], 2        ; 是286+ ?
        jb      ud_skp1
        call    hook_int6           ; 挂起坏操作码中断

; 开始测试内幕指令

ud_skp1:
        cmp     [cpu_val], 0        ; 是8088 或 V20 吗?
        jne     ud_skp2             ; 若不是，则跳转
        test    [cpu_info], 2       ; V20/V30 ?
        jnz     ud_skp2             ; 如果是，则跳转（无POP CS）
        OUTMSG  ud_popcs            ; 内幕指令 POP CS
        push    cs
```

```
        POPCS
        nop
        OUTMSG  ud_tested               ; 测试成功

ud_skp2:
        cmp     [cpu_val], 2            ; 286 ?
        jne     ud_skp3                 ; 若不是，则跳转
        OUTMSG  ud_ldall2               ; LOADALL
        OUTMSG  ud_untest               ; 未测试

ud_skp3:
        cmp     [cpu_val], 3            ; 386 ?
        jne     ud_skp4                 ; 如果不是，则跳转
        OUTMSG  ud_ldall3               ; LOADALL
        OUTMSG  ud_untest               ; 未测试
ud_skp4:
        cmp     [cpu_val], 3            ; 386+
        jb      ud_skp6
        OUTMSG  ud_umov                 ; 可能支持UMOV
        mov     [badoff], offset ud_skp5 ; 坏操作码
        mov     bx, 5A5Ah
        xor     ax, ax
        db      0Fh, 10h, 0D8h          ; mov al, bl
        db      4 dup (90h)             ; NOP
        cmp     al, 5Ah
        jne     ud_skp5                 ; 测试失败
        xor     ax, ax
        db      0Fh, 11h, 0D8h          ; mov ax, bx
        db      4 dup (90h)             ; nops
        cmp     ax, 5A5Ah
        jne     ud_skp5                 ; 测试失败
        xor     ax, ax
        db      0Fh, 12h, 0C3h          ; mov al, bl
        db      4 dup (90h)             ; nops
        cmp     al, 5Ah
        jne     ud_skp5                 ; 测试失败
        xor     ax, ax
        db      0Fh, 13h, 0C3h          ; mov ax, bx
        db      4 dup (90h)             ; nops
        cmp     ax, 5A5Ah
```

```
        jne         ud_skp5                 ; 测试失败
        OUTMSG      ud_tested               ; 测试成功
        jmp         ud_skp6
ud_skp5:
        OUTMSG      ud_failed               ; UMOV 测试失败

ud_skp6:
        cmp         [cpu_val], 5            ; Pentium ?
        jb          ud_skp9                 ; 如果不是，则跳转
        OUTMSG      ud_rdtsc                ; RDTSC
        mov         [badoff], offset ud_skp8 ; 坏操作码
.586
        mov         eax, 0
        mov         edx, 0
        RDTSC                               ; 读时间标志寄存器
        nop
        nop
        cmp         eax, 0                  ; 在edx:eax中的返回值，
        jne         ud_skp7                 ;  将是重启后的滴答数，
        cmp         edx, 0                  ;  所以零值表示
        je          ud_skp8                 ;  RDTSC 无效
.8086
ud_skp7:
        OUTMSG ud_tested                    ; 成功!
        jmp         ud_skp9

ud_skp8:
        OUTMSG      ud_failed               ; 失败

ud_skp9:
        cmp         [cpu_val], 3            ; 仅386?
        jne         ud_skp13
        OUTMSG ud_xbts                      ; 内幕指令XBTS
        mov         [badoff], offset ud_skp10 ; 坏操作码
        mov         ax, 1
        mov         bx, 1
        mov         cl, 1
        mov         dx, 1
```

```
        db      0Fh, 0A6h, 0DAh          ; xbts  bx, dx, ax, al
        nop
        nop
        OUTMSG  ud_tested
        jmp     ud_skp11
ud_skp10:
        OUTMSG  ud_notA                  ; 失败，不是386的step A

ud_skp11:
        OUTMSG  ud_ibts                  ; 内幕指令IBTS
        mov     [badoff], offset ud_skp12 ; 坏操作码
        mov     ax, 1
        mov     bx, 1
        mov     cl, 1
        mov     dx, 1
        db      0Fh, 0A7h, 0DAh          ; ibts  bx, dx, ax, al
        nop
        nop
        OUTMSG ud_tested
        jmp     ud_skp13
ud_skp12:
        OUTMSG ud_notA                   ; 失败，不是386的stepA

ud_skp13:
        cmp     [cpu_val], 2             ; 80286+
        jae     ud_skp14                 ; 如果是，则跳转
        jmp     ud_skp22                 ; 如果不是，则跳转

ud_skp14:
        OUTMSG ud_shlal3                 ; 内幕指令SHL/SAL
        mov     [badoff], offset ud_skp15 ; 坏操作码
        mov     al, 1
        db      0C0h, 0F0h, 5            ; shl  al, 1
        db      4 dup (90h)              ; 空指令NOP
        cmp     al, 20h                  ; 发生移位了吗?
        jne     ud_skp15                 ; 如果没有发生，则跳转
        OUTMSG ud_tested
        jmp     ud_skp16
```

```
ud_skp15:
        OUTMSG ud_failed

ud_skp16:
        OUTMSG ud_shlax3                    ; 内幕指令SHL/SAL
        mov    [badoff], offset ud_skp17    ; 坏操作码
        mov    ax, 10h
        db     0C1h, 0F0h, 5                ; shl  ax, 5
        db     4 dup (90h)                  ; 空指令NOP
        cmp    ax, 200h                     ; 发生了移位吗?
        jne    ud_skp17                     ; 如果没有发生，则跳转
        OUTMSG ud_tested
        jmp    ud_skp18
ud_skp17:
        OUTMSG ud_failed
ud_skp18:
        OUTMSG ud_shlal                     ; 内幕指令SHL/SAL
        mov    [badoff], offset ud_skp19    ; 坏操作码
        mov    al, 2
        db     0D0h, 0F0h                   ; shl  al, 1
        db     4 dup (90h)                  ; 空指令NOP
        cmp    al, 4                        ; 发生位移了吗?
        jne    ud_skp19                     ; 如果没有发生，则跳转
        OUTMSG ud_tested
        jmp    ud_skp20
ud_skp19:
        OUTMSG ud_failed

ud_skp20:
        OUTMSG ud_shlax                     ; 内幕指令
        mov    [badoff], offset ud_skp21    ; 坏操作码
        mov    ax, 200h
        db     0D1h, 0F0h                   ; shl  ax, 1
        db     4 dup (90h)                  ; 空指令NOP
        cmp    ax, 400h                     ; 发生位移了吗?
        jne    ud_skp21                     ; 如果没有发生，则跳转
        OUTMSG ud_tested
        jmp    ud_skp22
```

```
ud_skp21:
        OUTMSG   ud_failed

ud_skp22:
        OUTMSG   ud_aam                          ; 内幕指令AAM，非0Ah立即数
        mov      [badoff], offset ud_skp22a      ; 坏操作码
        mov      ah, 0FFh                        ; 垃圾值（应忽略）
        mov      al, 12h
        db       0D4h, 8                         ; AAM, 立即数=8
        db       4 dup (90h)                     ; 空指令
        cmp      al, 2
        jne      ud_skp22a                       ; 如果指令无效则跳转
        cmp      ah, 2
        jne      ud_skp22a                       ; 如果指令无效则跳转
        OUTMSG   ud_tested                       ; 测试成功
        jmp      ud_skp22b
ud_skp22a:
        OUTMSG   ud_failed

ud_skp22b:
        OUTMSG   ud_aad                          ; 内幕指令AAD非0Ah立即数
        mov      [badoff], offset ud_skp22c      ; 坏操作码
        mov      ax, 0705h
        db       0D5h, 10h                       ; AAD, 立即数 =10h
        db       4 dup (90h)                     ; 空指令
        cmp      al, 75h
        jne      ud_skp22c                       ; 如果指令无效则跳转
        cmp      ah, 0
        jne      ud_skp22c                       ; 如果指令无效则跳转
        OUTMSG   ud_tested                       ; 测试成功
        jmp      ud_skp22d
ud_skp22c:
        OUTMSG   ud_failed

ud_skp22d:
        OUTMSG   ud_setalc                       ; 内幕指令SETALC
        mov      [badoff], offset ud_skp23       ; 坏操作码
        stc                                      ; 设置进位
        SETALC                                   ; 依进位设置al
```

```
        db      4 dup (90h)             ; 空指令
        cmp     al, 0FFh
        jne     ud_skp23                ; 如果指令无效则跳转
        clc
        SETALC                          ; 依进位设置al
        db      4 dup (90h)             ; 空指令
        cmp     al, 0
        jne     ud_skp23                ; 如果指令无效则跳转
        OUTMSG  ud_tested               ; 测试成功
        jmp     ud_skp24
ud_skp23:
        OUTMSG  ud_failed

ud_skp24:
        cmp     [cpu_val], 3            ; 386+
        jb      ud_skp27
        OUTMSG  ud_icebp                ; 可能支持ICEBP
        cmp     [cpu_prot], 2           ; 位于保护/v86模式?
        jb      ud_skp25                ; 如果非，则跳转（测试成功）
        OUTMSG  ud_untstv               ; 未测试信息（v86）
        jmp     ud_skp27

ud_skp25:
        mov     [badoff], offset ud_skp26   ; 坏操作码
        push    es
        xor     ax, ax
        mov     es, ax
        cli                             ; 禁止中断
        mov     dx, es:[4]              ; 获取中断1的偏移量
        mov     cx, es:[4+2]            ; 获取段值
        mov     ax, offset int1test
        mov     es:[4], ax              ; 设置新向量
        mov     ax, cs
        mov     es:[4+2], ax
        sti                             ; 开放中断
.386P
        mov     eax, dr7                ; 获取调试寄存器
        and     eax, 0FFFFEFFFh         ; 清零第12位来支持ICEBP
```

```
        mov         dr7, eax                    ; 设置
        ICEBP                                   ; 触发中断1
        db          4 dup (90h)                 ; 空指令
.8086
        cli                                     ; 禁止中断
        mov         es:[4], dx                  ; 恢复初始中断1
        mov         es:[4+2], cx
        sti                                     ; 开放中断
        pop         es
        test        [ud_flags], 2               ; 发生了中断1吗?
        jz          ud_skp26
        OUTMSG      ud_tested                   ; 测试成功
        jmp         ud_skp27

ud_skp26:
        mov         [ud_flags], 0               ; 清除坏操作码标志
        OUTMSG      ud_failed                   ; ICEBP测试失败

ud_skp27:
        OUTMSG      ud_testal                   ; 内幕指令测试al
        mov         [badoff], offset ud_skp28   ; 坏操作码
        mov         al, 1
        db          0F6h, 0C8h, 01h             ; test al, 1
        db          4 dup (90h)                 ; 空指令
        jz          ud_skp28                    ; 如果失败，则跳转
        mov         al, 0
        db          0F6h, 0C8h, 01h             ; test al, 1
        db          4 dup (90h)                 ; 空指令
        jnz         ud_skp28                    ; 如果失败，则跳转
        OUTMSG      ud_tested
        jmp         ud_skp29
ud_skp28:
        OUTMSG      ud_failed

ud_skp29:
        OUTMSG      ud_testax                   ; 内幕指令测试ax
        mov         [badoff], offset ud_skp30   ; 坏操作码
        mov         ax, 100h
```

```
        db      0F7h, 0C8h, 0, 1        ; test ax, 100h
        db      4 dup (90h)             ; 空指令
        jz      ud_skp30                ; 如果失败，则跳转
        mov     ax, 0
        db      0F7h, 0C8h, 0, 1        ; test ax, 100h
        db      4 dup (90h)             ; 空指令
        jnz     ud_skp30                ; 如果失败，则跳转
        OUTMSG  ud_tested
        jmp     ud_skp31
ud_skp30:
        OUTMSG  ud_failed

ud_skp31:
        cmp     [cpu_val], 2            ; 286+?
        jb      ud_skp32                ; 如果不是，则跳转

        call    restore_int6
ud_skp32:
        ret

;-------------------------------------------------------------------------------
; 如果支持ICEBP指令，则中断向量
;   指向此处

int1test:
        mov     cs:[ud_flags], 2        ; 设置标志，程序执行到此处。
        iret

chk_undoc endp
```

代码例 4-14　找寻新的内幕指令

前面的程序只测试所有已知的内幕指令，但是对于将来 CPU 上出现的新的秘密指令又该怎么办呢？下面的程序将搜寻每个可能的 CPU 指令，以定位隐藏指令。

CPUTEST 是这样工作的：先执行当前 CPU 未定义的某个指令，然后观察是否会触发坏码或双重错误中断。如果两种中断都不发生，那么程序会批示这条指令有效，但并不指出这条指令具体做了什么。尽管并非完全坚不可摧，但是这条指令还是证实，在当前的 CPU 系列中没有我还没有说明的其他隐藏的指令。

如果一个新的供应商提供了一种兼容芯片，那么这个程序可以帮助确定是否存在新的隐藏起来的指令。因为这个程序不知道指令会做些什么，所以应禁止中断，同时应将堆栈切换到另外一个替换的堆栈上，直到这条指令执行结束。尽管对每个指令都进行了一次测试，但是为了执行这条指令，可能必须设置某些寄存器，RDMSR 指令就是一个很好的例子，在这里，ECX 就有一个有限的值集。

这个程序还检测 CPU 是否支持 MMX。如果支持，CPUTEST 就不测试 MMX 指令。

在测试之前，这个程序会显示出将要测试的操作码。如果这条指令也会做一些异常的事情，比如挂起系统，那么将只显示造成错误的操作码组合。要看看这个程序在你的机器上的运行结果，请运行 CPUTEST。图 4-4 给出了一个例子。

```
CPU 内幕指令分析视图
测试指令: OFh, O4h                指令无效.
测试指令: OFh, OAh                指令无效.
测试指令: OFh, OBh                指令无效.
测试指令: OFh, OCh                指令无效.
测试指令: OFh, ODh                指令无效.
测试指令: OFh, OEh                指令无效.
测试指令: OFh, OFh                指令无效.
测试指令: OFh, 14h                指令无效.
测试指令: OFh, 15h                指令无效.
.
.
测试指令: OFh,C7h,32h             指令无效.
测试指令: OFh,C7h,3Ah             指令无效.
本 CPU 上未发现隐藏的内幕指令
```

图 4-4 Intel 486DX 系统上 CPUTEST 的部分输出

因为 CPUTEST 的代码并不是很有趣，也不能说明太多的问题，因此本书中并未列出它。这个程序的完整源代码保存在软盘上。

代码 4-15 寻找并显示型号专用寄存器

新指令读和写型号专用寄存器（RDMSR 和 WRMSR）最早出现在 IBM 的 386 和 486 上。Intel 在其 Pentium 和 Pentium Pro 中也包含有型号专用寄存器，在 AMD 的 5k86 中也包含了它们。这条指令允许访问内部 64 位的 CPU 寄存器以提供特殊的属性和功能。这条指令用 ECX 来指明要访问哪个型号专用寄存器。这意味着提供了超过 40 亿种可能的寄存器地址。

这个程序测试每个可能的型号专用寄存器，然后显示出所有已知和隐藏的寄存器。在 Intel 486 33MHz 上，这个程序要花 6 小时来测试所有的 40 亿组合，然后显示出标准的 Intel

486 还没有使用的任意一个型号专用寄存器。在 586+类型上，该测试要快些。我在不同的 CPU 上找到的所有 80000000h 以下的合法寄存器在 ECX 中的值是从零到 1004h。所有的这一切只要花几秒完了前 2000h 个可能的寄存器后停止测试。在所有情况下，你都可以用 Control-Break 在测试完成之前退出程序。

在 Pentium 中，从 ECX 值等于 80000000h 处开始，似乎有 32 个寄存器是重复的，它们类似于或完全等同于开始于 0 的那些寄存器。为了快速扫描这些寄存器，请在运行程序 CPURDMSR 时使用命令行选项“—”。

如果寄存器对 CPU 无效，那么型号专用寄存器指令会产生坏码错误或双重错误。这时，程序 CPURDMSR 会挂起这两个中断，来确定哪个寄存器有效。图 4-5 显示了在 Pentium 上运行 CPURDMSR 的结果。该测试只可以在实模式下运行。如果位于其他模式下这个程序会自动退出。

```
检测与分析模式专用寄存器 V2.00（C）1994，1996FVG
命令行选项: + 显示每个测试的寄存器
              显示位于 80000000h 处的 32 个内幕寄存器。
Ctrl - Break 退出测试

寄存器      返回的 64 位值                    描述
00000000h   edx : eax=00000000 : 00017AC0h    机器检查地址
00000001h   edx : eax=00000000 : 00000008h    机器检查异常类型
00000002h   edx : eax=00000000 : 00000004h    奇偶反转测试
00000004h   edx : eax=00000000 : 00000000h    指令数据测试
00000005h   edx : eax=00000000 : 00000000h    缓存数据测试
00000006h   edx : eax=00000000 : 00000000h    缓存标签测试
00000007h   edx : eax=00000000 : 00000000Eh   缓存控制测试
00000008h   edx : eax=00000000 : 00000000h    TLB 命令测试
00000009h   edx : eax=00000000 : 00000000h    TLB 数据测试
0000000Bh   edx : eax=00000000 : 00000000h    分支目标缓冲区标签测试
0000000Ch   edx : eax=00000000 : 00000000h    分支目标缓冲区目标测试
0000000Dh   edx : eax=00000000 : 0000001Ah    分支目标缓冲区控制测试
0000000Eh   edx : eax=00000000 : 0000001Ch    新特性控制
00000010h   edx : eax=00000000 : E9BF4ABFh    时间标志计数器
00000011h   edx : eax=00000000 : 00000000h    计数器事件选择和控制
00000012h   edx : eax=00000000 : 00000000h    计数器 0
00000013h   edx : eax=00000000 : 00000000h    计数 1
00000014h   edx : eax=00000000 : 00000000h    未公开
FFFFFFFFh
```

图 4-5　在 Intel Pentium 上运行 CPURDMSR 的输出

这个程序不检查是否允许写型号专用寄存器（WMSR），因为如果这个寄存器改变后，处理器会做些什么还不太清楚。许多寄存器在用于测试时，如果试图写操作会产生意料不到的后果。

由于 CPURDMSR 不太有趣，也不能说明太多的东西，所以在本书中我没有列出它。完整的源代码保存在软盘上。

代码例 4-16 显示时间标志计数器

Pentium 加入了一个全新的内幕指令，RDTSC，它读取隐藏的时间标志计数器，Pentium 所具有的新的性能。该计数器在第 3 章，内幕指令中有详细讨论。

这个程序利用这个新指令读取时间标志计数器，然后在屏幕上显示 64 位的返回值。这个程序可连续运行直到按下 Ctrl-Break。

图 4-6 显示了运行 CPURDTSC 的结果。CPURDTSC 是一个相当简单的程序，在软盘上包含了这个程序的源代码，在这里我没有列出。

```
读时间标志计数器                    V1.00
本程序显示时间标志计数器的当前内容
（操作码 0Fh, 31h），Ctrl – Break 停止测试

EDX:EAX 返回值
0000002D : 36E39086h

由于 Ctrl – Break 而中止测试
```

图 4-6 显示时间标志计数器

第5章

适配卡的开发

创建一个带有 BIOS ROM 并可以使用 I/O 端口的适配卡比较简单，但是关键部分的设计会使用户的安装和使用要么毫不费力，要么简直就是一场折磨。当然，这要么使你的客户支持群非常高兴，要么使他们不得不处理无尽的问题。本部分所讨论的问题大多数针对 AT 总线的机器，但是许多方面也适用于 MCA 与 EISA。通过灵活的软件可定义的 ROM 和 RAM，MCA 和 EISA 消除了某些上述的问题。EISA 还从硬件上支持灵活多变的端口地址选择。当然，AT 结构占据着市场的优势。Intel 即插即用的方法使那些带有即插即用系统的用户实现它更加简单。

ROM 表头和初始化

在系统 BIOS 上电自检（POST）阶段，POST 扫描一个从 C000 开始的内存段，一直到系统 BIOS 的起点（通常是 EF80）。POST 在每 2K 的边界上寻找是否存在 ROM。扫描 ROM 的过程可以分成几个阶段。在 POST 的最初阶段，扫描从 C000 到 C780 的段寻找视频 ROM，并初始化视频 BIOS。这将允许 POST 显示出现的错误信息。然后，POST 从 C800 到 DF80 扫描其他信息，以初始化可能有的适配器 ROM。

使用 128K 系统 ROM 的系统，例如 IBM 的 PS/2，会在 DF80 处停止扫描。其他带有 64KB 或更少系统 ROM 的系统将会继续扫描 EF80。

早期的 IBM AT 标准保留了段 E000，并把它作为一个可任选的系统 BIOS。但据我所知从未用过它。POST 只寻找段 E000，以获得 AA55h 的表头字。如果找到，在检测到 BIOS 时会跳到偏移量为 3 处的初始化代码。没有定义和检查偏移量为 2 处的表头大小。也没有另外扫描 E000 以后的段。

检测表头字 AA55h，识别可能有效的 ROM。所有的表头字如下：

第 0 字节	55h
第 1 字节	AAh
第 2 字节	ROM 的大小 单位：512 字节大小的页面（最大为 7Fh）
第 3 字节	初始化代码的第一个字节

例如，VGA BIOS 开始的字节如下所示：

```
C000:0000   55  AA 40  EB 04  37  34  30-30   E9  42  14  00  00  49  42
C000:0010   4D 20  56  47  41  20  43  6F-6D   70  61  74  69  62  6C  65
```

在此例中，偏移量 2 处的字节 40h 表示该 BIOS ROM 的大小为 32KB。初始化代码开始于偏移量 3 处。

当 POST 检测 ROM 的表头时，它首先将该 ROM 的所有字节加起来（被加起来的字节数由 ROM 的大小字节指出）。该结果的低字节应该等于 0。如果不等于 0，则会忽略该 ROM 的初始化。如果检查确实得到 0，POST 将远调用偏移量为 3 处的初始化代码。

ROM 可以完成任何必要的初始化，比如设置适配卡硬件、挂起中断向量等等。初始化完成后，ROM 必须通过 RETF 指令将控制权归还给 POST 程序。一些老式的机器在 POST 代码中存在 bug，它会导致 POST 在 POST 期间两次调用该适配器的初始化程序。写初始化程序时必须考虑到这一点。

MCA ROM 扫描

在 MCA 类型的机器上，POST 从段 C000 上开始扫描。该扫描一直到 DE80 以寻找显卡 ROM。除了寻找前面所谈到的 ROM 表头外，它不寻找标识串以确定该卡是否是一个显卡。一个 MCA 的显卡 ROM 包含下述信息：

byte	0Ch	Vidio identification string（77h,0CCh,'VIEDIO'）
byte	30h	Programmable Option Select (POS) for port 102
byte	31h	Programmable Option Select (POS) for port 103
byte	32h	Programmable Option Select (POS) for port 104
byte	33h	Programmable Option Select (POS) for port 105

在 POST 的后面，将会再次扫描从段 C000 到段 DF80 的相同区域，以初始化其他非视频的 ROM。

设置 ROM 大小和开始地址

表头中偏移量为 2 处的字节包含有 ROM 的大小，单位是 512 字节大小的页面。某些 ROM 常用的值如下：

10h=8k
20h=16k
30h=24k
40h=32k
80h=64k

使用的内存的大小对今天拥挤的计算机显得非常重要。不用考虑适配卡内存要求的年代一去不返了。由于适配卡占用越来越多的内存空间，留给其他适配器的空间就很少了。同时，适配器 ROM 也会降低用户装载 TSR 和驱动文件的能力。对于要求 32K 以上的 ROM 来说，应该仔细考虑其他的替代方法。这些替代方法包括使用 EMS 和 XMS 分页转移到 I/O 控制下的较小的内存寻址空间。

ROM 只使用专门的大小（也就是说，一个 24K 的 ROM 没有实用性）。聪明的设计者只使用必要的内存寻址空间。如果固件只要求 24K，适配卡硬件将可能使用一个 32K 的 ROM。另外，当访问那些 ROM 的未使用部分时，良好的设计风格将不会连上总线。在本例中，32K ROM 至少有 8K 不会在其他适配卡试图使用该寻址区域时发生冲突。

某些早期的 VGA 适配卡在 ROM 表头中特别指定了使用 24K，但是忽略了对剩下 8K 的访问的关闭操作。这会造成各种冲突，这些冲突牵涉到其他的适配卡以及依赖表头精确信息的其他软件。

大多数的 386 以及更新的计算机，都使用内存管理软件，比如 DOS 的 EMM386、内存命令器、QEMM 或者 386MAX。以上这些都可以使用未被使用的高端内存区。由于受到处理器的限制，高端内存的页面管理限于每页 4K，最小大小是 4K。由于这个原因，最好将适配器的起始地址取为 4K 的倍数。一个 4K+2K 的起始地址在高载时将会使 2K 的高端内存不可用。

当然，大多数灵活的设计可由软件来设置 ROM 的起始地址。这会减少与其他适配器的冲突，并且允许用户控制适配卡放置的确切位置。如果没有被内置到系统的 BIOS 中，通常就将显卡 ROM 放在 C000 处。非视频卡就避免放在 C000 处，以免发生重叠或初始化失序。

ROM 代码

适配器的 BIOS ROM 负责对卡进行初始化和操作，许多卡也将控制传递给其适配器设备驱动程序或与之共享控制。这些设备驱动程序控制操作系统的接口，某些情况下，驱动程序可以替代有缺陷或者过时的 BIOS ROM 程序。

必须遵守一些规则来最大化系统的兼容性，以尽可能地与其他软件保持兼容。

规则 1：不要向 ROM 写操作

由于 ROM 是不可写的，所以在程序中写 ROM 通常代表着一个 bug。当然这可能是故意的，但是我确实看到过某些商业的适配卡 BIOS 犯着这样的错误。如果使用内存管理程序掩盖了适配器的 ROM，那么写 ROM 会产生一个一般保护性错误。掩盖技巧实际上是将 8 位 ROM BIOS 内容拷贝到快速的 32 或 64 位 RAM 映射到老 ROM 地址上，这能显著加快 BIOS 服务速度。现在的内存管理程序处理一般保护性时是透明的，并且会忽略无效的写操作，这一点很重要，因为如果进行写操作，那么会明显地降低运行速度。一般保护性错误是 CPU 和内存管理软件中最慢的进程，即使它仅仅只是返回而不做任何事。

规则 2：不要在 ROM 中固定段

固定段意味着指定的 ROM 地址是硬编码的，并且 ROM 只能存在于一个地址中。例如，一个位于 D000 处的 BIOS ROM 以硬编码的 D000 为段址远跳转或远调用。如果一个 ROM 代码需要的内存空间少于 64K，那么对其段进行硬编码是没有必要的。跳转和调用指令应该总是相对于其指令指针的。相对地址有利于生成更快更紧密的段并避免上述问题。

即使卡没有提供硬件能力来使用可替换的 ROM 地址，先进的内存管理程序，比如 Memory Commander 和 386MAX，也可以将 ROM 移动到不同的地址以提高系统的灵活性。当在，只有在代码段没有嵌入到该代码时它才发挥作用。

获得必要 RAM 的决窍

某些类型的适配卡已经有很好的方法来包括 RAM 内存。例如，一个 VGA 卡在 A000h 处保留了 128K。该卡可以有很多方法来使用该区域，包括从较大的卡总内存中切换内存页面。

其他大多数卡并没有为它们需要的 RAM 预先指定方法。有大量的方法可以提供该 RAM，每种有长处也有不足。

方法 1：在卡上提供 RAM

这可能是提供 RAM 的最可靠的方法，但是会增加卡的成本和复杂性，也很难避免与内存管理软件发生冲突。

大多数问题发生在下面这个时候：内存管理程序不知道卡的 RAM，并且将卡的 RAM 当成了未使用的空间。内存管理程序会改变 CPU 的页面映射，将 RAM 映射到高端内存的未用区域。然后，内存管理程序会向该区域装入 TSR 和设备驱动程序。如果卡 RAM 没有

被排除在内存管理程序之外，那么卡的 RAM 将是不可访问的。当卡的 ROM 或设备驱动程序读或写卡 RAM 的区域时，卡会读到错误的数据，或者写到了 TSR 或设备程序的顶部！通常会产生系统冲突。

现有的内存管理程序会努力将 RAM 内存可以读写，这要求内存管理程序对适配卡的 RAM 内存可以读写数据。这会有些冒险，但是内存管理程序会将内存恢复到其原始状态。管理程序还假定读和写操作不会引发其他事件。

有两种方法可以减少这些冲突。首先，可以将适配卡的 RAM 地址当作适配器 ROM 的一部分。例如，适配器的 ROM 可以指示该适配器保留 32K 供其使用（参看前面部分），页实际情况是，适配器将 24K 空间用作 ROM，另外 8K 用作 RAM 区域。这种方法在初始化时不需要 RAM 的内容是一个已知的值，所以 POST 检查有效。

另一种方法使用假的表头，使得 RAM 就像 ROM。例如，RAM 的第一字总是 AA55h，接着就是 RAM 的大小（单位是 512 字节的页）。第四个字节是一个 RETF 指令（CBh）。如果 BIOS 或其他程序试图初始化该 ROM，RETF 就将控制返回而不做任意事。如果适配器在前 4 个字节这样写表头，那么任意扫描 ROM 的软件都会忽略该 RAM 区域。如果 RAM 的地址下面接着相关的 ROM，那么 ROM BIOS 可以将 RAM 初始化到需要的值。

这两种方法都可以解决许多问题，除非系统掩盖了该适配器的 ROM 区域。正如前面所指出的，这意味着 ROM 将被拷贝到 RAM 中，并且 RAM 将被重定位到与 ROM 相同的地址。这种掩盖起的 RAM 总是写保护的，所以上面列出的两种方法会失败。通常情况下不会自动掩盖适配器 ROM，因为经常出现与速度相关的问题。新的系统几乎总是掩盖视频 BIOS 和系统 BIOS。

方法 2：从 DOS 主存区的顶部借用 RAM

如果仅仅只需要少量的内存，特别是 10KB 以下的，该方法将非常容易。使用 DOS 内存可以节省卡的成本，但也会产生问题。用户努力获取内存可能的每一字节，减少用户主存的想法从来被广泛接受。大量的病毒也会从主存顶部开始占据内存。很难检测系统是感染上了病毒还是适配卡占据了这部分内存，这一点会进一步使用户和技术支持人员感受到困惑。为了在适配器 ROM 初始化期间从主存借用内存，每需要 1024 个字节时，应将在地址 40:13h 处的字减 1。该字保存了主存的全部大小信息，单位 1K。为了确定使用区域的开始段址，将 40:13h 中新的值乘以 40h。你必须保存该段址值。因为其他的适配卡和程序也可能从顶部占用额外的内存，而改变了 40:13h 处的值。

方法 3：使 BIOS 或中断数据区使用一些字节

在低 BIOS 数据区的尾部，段 40h（偏移量范围从 CFh 到 FFh）处，通常有许多字节未被使用。不推荐使用这些内存。制造商在将来可能会使用这些字节，没有简单可行的方法

保证不发生冲突。

在未用到的中断地址保存有限的数据也是可能的。是比使用 BIOS 数据区更好的一种方法，但是它也会造成冲突。某些系统 BIOS 也已经在这样做。某些不掩盖主要 BIOS ROM 的 AMI BIOS，就在 0:300h（中断 Coh 到 C7h）处保存用户定义的硬盘类型信息。

如果你必须使用这些方法中的某种，请记住一定要能检查保存信息的有效性，例如提供检查的手段，以防其他人也重写了它！如果有人挂起了数据保存地点的中断向量，系统也有可能会冲突。某些时候，控制会被传到保存了数据字节的向量地址。这时系统可能会跳到一个垃圾地址，通常会挂起系统。

方法 4：使用 I/O 寻址内存

当只要求有很少数量的内存并且速度不重要时，可以采用几个 I/O 端口地址来访问适配器上的内存，特别是使用一个端口来设置地址，然后另一端口用页传送数据。这种方法可避免目前为止我们所谈到的许多冲突，但要需要较为复杂的硬件接口。

系统端口 70h 和 71h 处的 CMOS 内存就是采用这种方法。既然我已经指出该处是 CMOS 内存，所以我强烈建议不要试图使用该端口。CMOS 很有限，并且也难以识别出那些未使用的字节。不同的 BIOS 制造商将这些字节用作不同的目的。如果需要，请参看第 15 章 CMOS RAM，以获取更多信息。

方法 5：使用设备驱动文件来保留必要的 RAM

使用设备驱动文件获得额外内存容量的一种有效的方法，且没有比它更有效的了。方法 2 减少了本来就有限的内存，可能会不利于程序的运行。有两种方法可以克服这种不利因素。首先，如果设备驱动由内存管理程序载入高端内存。那么就可以减少系统中发生这种问题的可能。其次，如果检测了 EMS 和/或 XMS 内存，可以从 EMS 或 XMS 分配足够的所需内存。

使用设备驱动文件主要要考虑两个方面。首先，如果适配器需要设备驱动文件，那么该适配器不会提供其服务却运行了 CONFIG.SYS 文件。这会使磁盘控制器相当于复杂，但是对网络适配器来说关系不大。

仔细考虑什么时候才真正需要设备驱动文件。由于有大量的驱动文件以及它们提供了专门的服务，所以很难做到驱动文件在 CONFIG.SYS 中放在“第 1 位”。显然，只能有一个设备驱动文件可以放在“第 1 位”。最佳设计应避免要求将设备驱动文件放在“第 1 位”，这样可以为用户和技术支持人员减少不少麻烦。使用设备驱动程序也会遇到没有装入的风险。用户可能没有正确地设置系统或者突然删除了设备驱动程序。如果某行 DEVICE=是可任选的，那么用户可能会不正确地选择该选项。任何设计都必须考虑到这些潜在的问题。

一个值得考虑的问题是，在其他操作系统上适配器会如何发挥作用。一个简单的自含

式磁盘控制器，例如 IDE 类型，可以运行在所有主要的操作系统上（DOS、OS/2、NT、UNIX 等等）。如果该适配器会被用于其他的操作系统，那么可能需要为每个系统准备一个独立的设备驱动程序。

方法 6：使用 DMA

不需要任意可寻址的适配器内存就可以处理许多情况。对于这些适配器来说，可以通过直接内存访问（DMA）来传送信息。适配器可以向系统或应用程序传送大量的内存内容而不用使用可寻址的内存。几乎所有的系统都通过 DMA 传送软盘数据。许多网卡和扫描仪卡也使用 DMA 传送数据。请参看有关 DMA 的章节获取详细信息。

不幸的是，使用 DMA 也存在问题，系统上可用的 DMA 通道总是有限的。在安装的过程中选择哪条 DMA 通道是一个大问题。也没有适合的方法来确定为其他的适配器保留了哪个 DMA 通道。通常用户被告知试一试不同的 DMA 通道，直到该适配器可以工作。这一点简直令人难以置信！许多用户对此一无所知，这样使用 DMA 通道不知会产生什么其他恶果。

即插即用系统和 EISA 系统解决了 DMA 通道的分配问题。这些系统提供了一个配置程序，通常可以解决 DMA 通道的使用和冲突问题。

方法 7：使用非常高的 RAM 地址

对于 386DX 和后来的 CPU，32 位的寻址范围允许访问最大到 4GB 空间。即使是 286 也可以访问 16MB 内存。适配器可以窃取一些未用的地址空间以供其使用，前提是适配器的 RAM 不会与系统的实际 RAM 发生冲突。可以通过 BIOS 服务中断 15h 的 87h 号功能来访问 RAM。如果适配器的代码运行在保护模式下，那么可以直接访问内存。

对于高端 RAM 地址，会发生两个主要的问题。首先，适配器 RAM 会和系统内存重叠。将内存的限制告诉给用户以处理系统内存冲突。记住，在许多 386 以及后来的 CPU 上，许多内存管理程序不允许访问不受其控制的内存空间。管理程序可能会阻止对系统内存地址范围以外的访问。

如果系统向适配卡发出所需的地址信号，这时会出现另外一个问题。一个标准的 16 位 ISA 卡插槽只能访问前 16MB 内存。8 位卡插槽是没有用的，因为 8 位插柄只允许访问前 1MB 内存。一个 VESA 局部总线，PCI、MCA 以及 EISA 专用卡可以访问全部的 4GB 地址空间。

方法 8：是否真正需要 RAM

在某些情况下，适配器可能只需要少量的临时内存。这时可以考虑使用堆栈内存。这可以提供少量内存，同时，如果是临时使用，该方法也非常有效。为了系统能可靠地工作，

要求所需的堆栈内存大小应少于 10 个字。如果使用得过多，堆栈就会在使用其他适配器和 TSR 时溢出。堆栈也不应当作长期内存。在适配器程序使用中断时堆栈会很有用，然而在其他情况不太有用。

选择 I/O 端口号

没有特别好的方法来保证所选的 I/O 端口号不会与其他适配器发生冲突——我敢肯定你一定听说过这一点！然而，你可以较为明智地挑选端口地址，以尽可能减少问题和冲突。

最佳选择方法应是找那些类似的适配卡的 I/O 端口。一种聪明的做法是，给新的显卡分配相同的端口，以代替系统的 VGA 适配器。如果新的适配器需和老的 VGA 适配器同时使用，那么必须选择另外不同的端口地址。这种方法也适和于其他类型的适配卡，例如声卡、扫描仪等等。先找到相似的适配器所使用的地址，然后使用相同的端口地址。

其次是使用没被使用的端口号。本书结尾处的索引指出许多通用的端口分配。

最好是尽可能少地使用 I/O 端口，并且允许使用多组端口。这并不会使硬件或安装指令更复杂，但是的确会对安装成功与否起重要的作用。如果某产品有一系列的安装程序，那么这些程序应保证能正确地安装适配器和使用指定的端口。这有利于解决许多问题，不会让用户和你的客户支持人员感到头疼。

很多端口

CPU 为系统总共指定了 65536 个 I/O 端口。但实际上在 PC 系统只有很少的端口。在非 MAC 类型的系统上，主板和许多适配卡并不为 I/O 地址线 A10 及其以上的地址线译码。地址线 A10 到 A15 成为系统和适配器忽略的部分。这样端口号就限定到只有 1024 个有用的端口。这似乎也已经很多了，但是生产商生产的产品要求有更多的端口供它们使用。

端口 0 到 FFh 为系统主板而保留。如果适配卡试图使用端口 0 到 FFh，就会产生严重的冲突。大多数主板并不为所有的端口号译码。例如，中断控制器使用端口 20h 和 21h，并且大多数系统没有定义 22h 到 3h。似乎从 22h 到 3Fh 这里又有 30 个未定义端口。事实并不是这样的。

大多数情况对 I/O 地址 20h 到 3Fh 忽略地址线 A1 到 A4。这意味着中断控制器不仅只访问 20h 和 21h，而且还会访问每个端口号一直到 3Fh！读端口 3Eh 等同于读端口 20h。由于这种糟糕的译码方式，并且还因制造商的不同而不同，试图使用任意从 0 到 FFh 端口号来用作普通用途的适配卡的开发人员都是极其愚蠢的。

通过适当的 I/O 地址译码，主板设计得可以额外分配 0~FFh 内的端口。带有先进芯片

组的系统使用端口 22h 和 23h 来访问该芯片组的先进特性。当然，这意味着他们至少可以为 I/O 地址线 0 和 1 译码，以避免和端口 20h 处的中断控制器发生重叠。

记住，许多较新的主板包含有内置于主板上的适配卡，最常见的如串行口和并行口。许多 PCI 和 MCA 系统也包含有 VGA 适配器和软硬盘控制器。所有这些内置的适配器使用 100h 到 3FFh 处的端口。另外，MCA 系统还为其要编程选项选择（POS）功能保留了端口 100h 到 107h。

表 5-1 总结了非 EISA 系统的端口分配情况。该表也显示了端口号 400h 后的端口地址是如何复制的。

表 5-1 系统端口的分配——PC/XT/AT/MAC

I/O 端的范围	由 谁 使 用
000~00FF	系统主板（即：DMA、时钟、键盘控制器等等）。
0100~03FF	可任选的适配卡（即：VGA、磁盘、网络等等）。
0400~04FF	重复 0~FF。
0500~07FF	重复 100~3FF。
.	.
.	.
.	.
FC00~FCFF	重复 0~FF。
FD00~FFFF	重复 100~3FF。

EISA 系统改进了 AT 类型的系统。它提供了更聪明的方法，以便在扩展可配的端口号之时仍保持与老系列的兼容。EISA 对 I/O 地址线全译码。这样，新区域 400h 到 4FFh 就可以用作另外的系统板功能。另外，每个 EISA 卡插槽可提供 1024 个不同的端口。在物理插槽的基础上定义端口，EISA 系统避免了任意冲突。

为了做到这一点，每个 EISA 适配卡必须让其端口号可以反映其对应的物理插槽。EISA 配置程序负责实现这一点。表 5-2 归纳了 EISA 端口分配情况。

表 5-2 系统端口分配——EISA

I/O 端口范围	由 谁 使 用
0000~00FF	ISA 兼容的系统主板（例如 DMA、时钟、键盘控制器等等）。
0100~03FF	ISA 可任选适配卡（例如 VGA、磁盘、网络等等）。
0400~04FF	ESA 系统主板（DMA 扩展、EISA 独有的寄存器等等）。

续表

I/O 端口范围	由 谁 使 用
0500~07FF	重复 100~3FF。
0800~08FF	为其他的 EISA 系统主板保留。
0900~0BFF	重复 100~3FF。
0C00~0CFF	为其他的 EISA 系统主板保留。
0D00~0FFF	重复 100~3FF。
x000~x0FF	供 EISA 的插槽 x 使用。
x100~0FFF	重复 100~3FF。
x400~x0FF	供 EISA 的插槽 x 使用。
x500~0FFF	重复 100~3FF。
x800~x0FF	供 EISA 的插槽 x 使用。
x900~0FFF	重复 100~3FF。
xC00~xCFF	供 EISA 的插槽 x 使用。
xD00~xFFF	重复 100~3FF。

注：在这里 x=1 到 F，代表 EISA 适配器插槽号从 1 到 15。

隐去 ROM 和 RAM

一些生产商认为，让适配器 ROM 和 RAM 只出现在运行适配器驱动文件的时候，是一种好方法。这种想法大错特错。IBM 的令牌适配卡就是众所周知的例子。问一问那些同这个东西打交道的网络管理员，你就会知道他们不愿使用这个卡。

有两个主要原因说明为什么不要隐去 ROM 和 RAM。首先，当隐去该卡时，很容易在同样的地址安装其他的适配卡。大多数高级的安装软件会行扫描内存中的 ROM，以确定新适配卡可以安装在哪些地方。如果隐去了某个适配器，用户就可以在相同的地址安装另外一个适配器。这会产生极令人头疼的事情。

另外一个原因源于现在的内存管理程序，大多数流行的内存管理程序，如 Memory Commander、QIMM 以及 386MAX，都会为适配卡而扫描定位安全区域来高载 TSR 和驱动程序。任何被找到是 ROM 或 RAM 的区域自动被排除。如果隐去了某适配器，那么为该适配器所保留的区域就会被切换到高端内存中去。这也是令人头疼的事情。

开关与跳线

通常，大多数工程人员总是试图设计带有最少开关和跳线的适配卡。它们会增加成本并且很容易被错误地设置。由于 EISA 系统提供了插槽唯一的 I/O 端口和不易变的插槽专用配置信息，所以它有能力不使用所有的开关及跳线。使用不易变的插槽专用配置信息，MCA 也成功地删去了所有的开关和跳线。Intel 的即插即用的方法提供了最好的途径来实现自动而快速地对卡进行配置。所有其他的系统代表了市场的大部分，它们几乎都需要至少一组的可替换 I/O 端口。其他功能可能也需要开关和跳线。例如，大多数的 EGA 和 VGA 适配器就带有开关来指定适配器的配置，以免与其他的视频适配卡相冲突。

开发一个带有开关和跳线的卡时，应尽可能地将设置信息明确地标在卡上。这听起似乎很自然，但实际中很少这样做。迟早会有人需要知道如何设置。尽管在电路板的部件印刷表面标明有关的信息并不增加成本，但是还是很少有公司这样做，这一点真令人感到奇怪。然而，许多公司总将资金投入到那些不必要的技术支持上，有的还会使客户更加感到困惑不解，这样的情况简直太多了。

即插即用

1994 年，Intel 为自动配置适配卡引入一种全新的概念。1995 年，许多 BIOS 开始支持即插即用。当然，大多数现存的系统仍不是即插即用的。这意味着当你为你的新系统提供即插即用的支持服务时，你可能需要处理许多与你的老系统有关的问题。参看附录 C，以获取完整的即插即用的特性的详细说明，以及其他相关的资料和书籍。

第 6 章

BIOS 数据和其他固定数据区

CPU、BIOS 以及普通的适配卡都占用了大量的固定数据区来在首兆字节地址空间执行各种功能。无论是何种操作系统或者硬件设计。这些地址在所有的 80x86 IBM 兼容机上非常常见。任何不同之处都会在专门的地址描述中注明。

固定数据区包括 CPU 中断向量表、保存在中断向量表内的数据以及 BIOS 和适配卡数据。我也列举了大量有关扩展 BIOS 数据区的未公开信息。第 7 章中详细介绍了中断向量以及存在中断向量中的数据。

BIOS 数据区

该区域含有系统 BIOS ROM 用到的关键数据信息。程序可以简单地访问 BIOS 数据。在所有系统上这个区域都开始于内存的 40h 段址处。通常可用两种方法来访问这段内存。其一是利用段址 40h 和偏离量为 0。第二种方法是使用段址 0，但偏离量开始于 400h。通常使用这两种方法可以访问相同的信息。

BIOS 数据区还包含一些信息，涉及某些常用的适配卡，例如串行口和并行口以及显卡。如果不需要安装这些可任选的设备，那么数据变量通常设为 0，表示未使用。

BIOS 数据项涉及相关的 BIOS 服务或功能。在专用服务的章节中详细解释了这些关系。

BIOS 数据区归纳

地　址	功　能	大　小	平　台
40:00h	串行 I/O 地址，端口 1	字	所有
40:02h	串行 I/O 地址，端口 2	字	所有
40:04h	串行 I/O 地址，端口 3	字	所有
40:06h	串行 I/O 地址，端口 4	字	所有

续表

地　　址	功　　能	大　　小	平　　台
40:08h	并行 I/O 地址，端口 1	字	所有
40:0Ah	并行 I/O 地址，端口 2	字	所有
40:0Ch	并行 I/O 地址，端口 3	字	所有
40:0Eh	扩展 BIOS 数据区段	字	AT+
	并行 I/O 地址，端口 4	字	PC、XT
40:10h	设备字	字	所有
40:12h	生产测试	字	所有
40:13h	主存大小，单位千字节	字	所有
40:15h	错误代码	字	AT+
	适配器内存大小	字	PC、XT
40:17h	键盘，换档标志，集合 1	字节	所有
40:18h	键盘，换档标志，集合 2	字节	所有
40:19h	键盘，ALT-数字键工作区	字节	所有
40:1Ah	键盘，缓冲区头指针	字	所有
40:1Ch	键盘，缓冲区尾指针	字	所有
40:1Eh	键盘，缓冲区	16 字	所有
40:3Eh	软盘，重校状态	字节	所有
40:3Fh	软盘，马达状态	字节	所有
40:40h	软盘，马达超时计数器	字节	所有
40:41h	软盘，控制器状态返回码	字节	所有
40:42h	软盘和硬盘控制器状态字节	7 字节	所有
40:49h	视频模式	字节	所有
40:4Ah	视频，列数	字	所有
40:4Ch	视频，每页的字节总数	字	所有
40:4Eh	视频，当前页面的偏移量	字	所有
40:50h	视频，光标位置，页面 0~7	8 个字	所有
40:60h	视频，光标形状	字	所有
40:62h	视频，活动显示页	字节	所有
40:63h	视频，I/O 端口号的基地址	字	所有
40:65h	视频，内部模式寄存器	字节	所有
40:66h	视频，色彩模式	字节	所有
40:67h	常用偏移量	字	XT+
	磁带，时间计数到数据边缘	字	PC

续表

地　址	功　能	大　小	平　台
40:69h	常用段址	字	XT+
	磁带，CRC寄存器	字	PC
40:6Bh	上次出现的中断	字节	XT+
	磁带，上次读取的值	字节	PC
40:6Ch	时钟滴答计数	双字	所有
40:70h	时钟滴答24小时翻转标志	字节	所有
40:71h	键盘，Ctrl-Break标志	字节	所有
40:72h	热启动标志	字	所有
40:74h	硬盘，上次操作状态	字节	XT+
40:75h	硬盘，附带的数目	字节	XT+
40:76h	硬盘，控制字节	字节	XT+
40:77h	硬盘，端口偏移量	字节	XT+
40:78h	并行打印机1，超时	字节	XT+
40:79h	并行打印机2，超时	字节	XT+
40:7Ah	并行打印机3，超时	字节	XT+
40:7Bh	并行打印机4，超时	字节	XT+
40:7Ch	串行口1，超时	字节	XT+
40:7Dh	串行口2，超时	字节	XT+
40:7Eh	串行口3，超时	字节	XT+
40:7Fh	串行口4，超时	字节	XT+
40:80h	键盘，指向缓冲区头的指针	字	XT+
40:82h	键盘，指向缓冲区头的指针	字	XT+
40:84h	视频，行数	字节	EGA+
40:85h	视频，每字符的像素数目	字	EGA+
40:87h	视频，选项	字节	EGA+
40:88h	视频，开关	字节	EGA+
40:89h	视频，保存区1	字节	EGA+
40:8Ah	视频，保存区2	字节	EGA+
40:8Bh	软盘，配置数据	字节	AT+
40:8Ch	硬盘，状态寄存器	字节	AT+
40:8Dh	硬盘，错误寄存器	字节	AT+
40:8Eh	硬盘，任务完成标志	字节	AT+
40:8Fh	软盘，控制信息	字节	AT+

续表

地　址	功　能	大　小	平　台
40:90h	软盘 0，媒质状态	字节	AT+
40:91h	软盘 1，媒质状态	字节	AT+
40:92h	软盘 0，可操作的起始状态	字节	AT+
40:93h	软盘 1，可操作的起始状态	字节	AT+
40:94h	软盘 0，当前磁柱	字节	AT+
40:95h	软盘 1，当前磁柱	字节	AT+
40:96h	键盘，状态标志 3	字节	AT+
40:97h	键盘，状态标志 4	字节	AT+
40:98h	用户的等待标志指针	双字	AT+
40:9Ch	用户的等待计数	双字	AT+
40:A0h	等待标志	字节	AT+
40:A1h	局域网	7 个字节	AT+
40:A8h	视频，参数控制块指针	双字	EGA+
40:CEh	时钟，1980 至今的天数（某些BIOS）	双字	AT+
50:00h	打印屏幕状态	字节	所有

BIOS 数据区细节

地址	描述	大小	平台
40:00h	串行口 1 I/O 地址，端口 1	字	所有

该字用来保存串行口 1 的第一个端口号。通常串行口 1 的 I/O 地址是 3F8h，但是任意的串行 I/O 端口都可以保存在该字中。

地址	描述	大小	平台
40:02h	串行口 2 I/O 地址，端口 2	字	所有

该字用来保存串行口 2 的第一个端口号。通常串行口 2 的 I/O 地址是 2F8h，但是任意串行 I/O 端口都可以保存在该字中。

地址	描述	大小	平台
40:03h	串行口 3 I/O 地址，端口 3	字	所有

该字用来保存串行口 3 的第一个 I/O 端口号。任意串行 I/O 端口都可以保存在该字中。

地址	描述	大小	平台
40:06h	串行口 4 I/O 地址，端口 4	字	所有

该字用来保存串行口4的第一个I/O端口号。任意串行I/O端口都可以保存在该字中。

地址	描述	大小	平台
40:08h	并行I/O地址，端口1	字	所有

并行I/O端口号保存在该字中。一般该地址值是3BCh、378h或278h，但是任意并行I/O端口都可以保存在该字中。大多数操作系统将它作为PRN或者LPT1访问。

地址	描述	大小	平台
40:0Ah	并行I/O地址，端口2	字	所有

并行I/O端口号保存在该字中。一般该地址值是378h，或278h，但是任意并行I/O端口都可以保存在该字中。大多数操作系统将它作为LPT2访问。

地址	描述	大小	平台
40:0Ch	并行I/O地址，端口3	字	所有

并行I/O端口号保存在该字中。一般并行口3设置为I/O地址278h，但是任意并行I/O端口都可以保存在该字中。大多数操作系统将它作为LPT3访问。

地址	描述	大小	平台
40:0Eh	EBDA段	字	AT+
	并行I/O地址，端口4	字	PC、XT

在AT+系统上，如果使用了扩展BIOS数据区，那么该字中保存了扩展BIOS数据区（EBDA）的段址。扩展数据区通常使用639K处常规内存顶部的1K RAM。参看本章后面部分，扩展数据区，获取更多信息。

在PC和XT系统上，某些情况下并没有使用扩展BIOS数据，这时，该值可用于保存并行I/O地址的第四个端口地址。该I/O端口地址必须由一个应用程序或驱动程序装入。与并行口1、2、3不同，POST并不设置该字。大多数操作系统将它作为LPT4访问。

地址	描述	大小
40:10h	设备字	字

该地址保存了由POST确定的配置信息。它的值可以在RAM中直接访问得到，或者调用中断11h，由设备确定该值。

AT+系统

位 15=x 由 POST 检测到的并行打印口数（0~3）
14=x
13=1 安装了内部调制解调器（某些系统）
12=0 未使用
11=x 附带的串行口数（0~4）
10=x
9=x
8=0 未使用
7=x 软驱数（如果第 0 位是 1）
6=x 软驱数（如果第 0 位是 1）

第 7 位	第 6 位
0	0=1 个软驱
0	1=2 个软驱

5=x 初始视频模式类型
4=x

第 5 位	第 4 位
0	0=EGA 或后来的适配器类型
0	1=彩色，40 列 25 行
1	0=彩色，80 列 25 行
1	1=单色，80 列 25 行

3=0 未用
2=1 在系统板上安装了鼠标口
1=1 安装了数学协处理器
0=1 安装了软盘驱动器

CP/XT 系统

位 15=x 由 POST 检测到的并行打印口数（0 到 3）
14=x
13=0 未用
12=1 附带的游戏端口
11=x 附带的串行口数（0~4）
10=x
9=x
8=0 未用
7=x 软驱数（如果第 0 位是 1）
6=x

第 7 位	第 6 位
0	0=1 个软驱
0	1=2 个软驱
1	0=3 个软驱
1	1=4 个软驱

5=x 初始视频模式类型

4=x	第 5 位	第 4 位
	0	0=未用
	0	1=彩色，40 列 25 行
	1	0=彩色，80 列 25 行
	1	1=单色，80 列 25 行
3=x	系统内存板的大小	
2=x	第 3 位	第 2 位
	0	0=16K
	0	1=32K
	1	0=48K
	1	1=64K
1=1	安装了数学协处理器（仅 XT）	
0=1	安装了软盘驱动器	

地址	描述	大小	平台
40:12h	生产测试	字节	所有

这个字节用来表示生产测试模式。通常情况下连接主板上的一个跳线来启动生产测试模式。在 POST 期间读取该跳线并将跳线状态保存到这个字节中。生产测试模式通常为系统的 burn-in 初始化测试周期，并协助进行某些维修工作。因每个 BIOS 供应商的不同，该制造测试的操作也有所不同。

非 MAC 系统

位	7~1=x	未使用，可能处于任何状态
	0=0	正常运行
	1	生产测试模式

MAC 系统

位	7=x	POST 标志，功能未知
	6=x	未使用
	5=x	未使用
	4=x	POST 标志，插槽 4 的适配器 ID=EDAFh
	3=1	POST 标志，视频类型 80×25 彩色
	2=x	POST 标志，功能未知
	0=0	正常运行
	1	生产测试模式

地址	描述	大小	平台
40:13h	内存大小	字	所有

该字包含主存的大小；单位是 1024 字节块。如果使用了扩展 BIOS 数据区，那么主存的大小将减去该区域的大小。对于常见的 640K 系统，该字中的值是 280h。

许多病毒程序会从主存的顶部窃取一些内存。其结果是保持病毒处于激活状态，并将该隐存区域的关键中断钩住。当在启动过程中病毒处于激活状态，它就将自己保存在主存的顶部，并减去其驻留所需的主存大小。

地址	描述	大小	平台
40:15h	错误代码	字	AT+
	适配器的内存大小	字	PC、XT

在当前的系统中，BIOS 供应商可任意分配这两个字节。在 IBM AT 系统上，它用来保存生产测试的错误代码。

在 PC/XT 系统，该字用来标示系统板外的主存大小。该值的单位是千字节。在某些技术参考书中，它用来指有歧意的词“I/O 通道大小”。这个过时的术语“I/O 通道大小”简单地讲就是适配卡总线。

地址	描述	大小	平台
40:17h	键盘换档集合 1	字节	所有

这个字节保存键盘 shift 键和组合键的当前状态。可以由低级键盘中断 9 获得。

CapsLock、NumLock 以及 Scroll Lock 和 AT 键盘的 LED 状态。中断 9 处理程序负责从这些标志中更新键盘的 LED 状态。没有为 Insert 按下提供虚拟的指示器。

位		
	7=1	Insert 开
	6=1	Caps Lock 开
	5=1	Num Lock 开
	4=1	Scroll Lock 开
	3=1	Alt 键按下（左或右）
	2=1	Control 键按下（左或右）
	1=1	左边 Shift 键按下
	0=1	右边 Shift 键按下

地址	描述	大小	平台
40:18	键盘换档集合 2	字节	所有

这个字节保存键盘的 Shift 键和组合键的当前状态，由低级键盘中断 9 获取。

位		
	7=1	Insert 键按下
	6=1	Caps Lock 键按下

5=1	Num Lock 键按下
4=1	Scroll Lock 键按下
3=1	Pause 激活
2=1	SysReq 键按下（83 键的键盘除外
1=1	左边 Shift 键按下（101/102 键键盘）
0=1	右边 Shift 键按下（101/102 键键盘）

地址	描述	大小	平台
40:19h	键盘 Alt-数字键盘工作区	字节	所有

如果在按下 Alt 键的同时按下数字键的某个键来输入一个十进制数值，那么键盘处理程序中断 9，不会生成一个从 0~FFh 的值。这个字节用作键盘处理器的临时工作区，来生成按下一次数字键的最终值。

地址	描述	大小	平台
40:1Ah	键盘，缓冲区头指针	字	所有

该字保存了对段 40 的偏移量，从这里获得键盘缓冲区里的下一个按键。如果该值指向地址 40:1Ch 缓冲区尾指针所指相同那么当前缓冲区空。

当从缓冲区移走一个键时，该指针加 2，直到缓冲区的最后一个字区域，这时重新设置它以回到了键盘缓冲区的起点。这样，缓冲区就是 FIFO 类型，保证最先存入缓冲区的键总是第一个取出。中断 16h（中间键盘处理程序）负责依据请求从缓冲区移走键并更新该指针。

地址	描述	大小	平台
40:1Ch	键盘，缓冲区指针	字	所有

该字保存了对段 40 的偏移量，用来保存键盘缓冲区的尾部指针。如果该字指向的地址和 40:1Ah 缓冲区头指针所指相同，那么当前缓冲区为空。

当向缓冲区加入一个键时，该指针加 2，直到指向缓冲区的最后一个字区域，这时重新设置它以回到键盘缓冲区的起点。这样，缓冲区就是 FIFO 类型，保证最后存入缓冲区的键总是最后一个取出。中断 9h（低级键盘处理程序）负责向缓冲区中加入键。另外，中断 16h 实现了强制键进入缓冲区。

如果向队列中加入一个新键会使尾指针等于头指针，那么就认为缓冲区是满的。这意味着在 RAM 地址 40:1Eh 处的 16 字缓冲区可以保存 15 个键。如果缓冲满了，键盘处理程序中断 9 就会忽略另外的按键。每次忽略都会发出一次错误的嘟嘟响。

地址	描述	大小	平台
40:1Eh	键盘缓冲区	16 字	所有

该 16 字的 FIFO 缓冲区可容纳 15 个键。每个字的高字节保存键的扫描码，低字节保存该扫描码的 ASCII 转换形式。在 RAM 地址 40:1Ah 和 40:1Ch 处的两个指针控制着缓冲区的开始入口和终端口。

地址	描述	大小	平台
40:1Fh	软盘，马达状态	字节	所有

这个字节保存每个软驱当前状态的状态信息。如果在地址 40:1Oh 处的马达计时器计时完毕，那么所有的四个马达开动的位都会被清零。

位		
	7=0	当前操作——读或检查
	1	当前操作——写或格式化，需要延迟
	6=x	未用
	5=x 4=x	驱动器选择

第 5 位	第 4 位
0	0=驱动器 0
0	1=驱动器 1
1	0=驱动器 2（仅 PC/XT）
1	1=驱动器 3（仅 PC/XT）

3=1	驱动器 3 的马达开（仅 PC/XT）
2=1	驱动器 2 的马达开（仅 PC/XT）
1=1	驱动器 1 的马达开
0=1	驱动器 0 的马达开

地址	描述	大小	平台
40:40h	软盘，马达计时器	字节	所有

这个字节用来保存上一次访问软盘后，保存软盘马达运动的时间。当任意一个软盘被访问时，这个字节中就被装入一个渐渐减少的值。中断 8，时钟 tick（一次滴答），在每次 tick 中减少该计数器。如果计数到零，中断 8 处理器程序就会关闭所有的软盘驱动器马达。

关闭马达所需的延迟长度由软盘参数表（参看第 10 章，表 10-2）的第三个字节设置。在多数系统设置该值为 25h，这样就会在关闭马达之前运行 2 秒钟。如果在计时器到期之前又有软盘操作发生，那么就会重新设置时钟，开始新的递减循环。

地址	描述	大小	平台
40:41h	软盘控制器返回码	字节	所有

如果 BIOS 中断 13h 检测到一个无效的 Ah 子功能值，那么就会在这个字节中保存这个

错误的信息。对于软盘控制器功能而言，该返回码依赖于控制器所返回的两个状态字节。参看第10章获得扩展返回状态码描述。

值	状态
0	运行成功
1	无效的值传送或不支持的功能
2	丢失地址标签
3	软盘写保护
4	找不到要法语的扇区
6	软盘改变线激活
8	DMA过载运行
9	数据溢出错误
0Ch	未找到媒体类型
10h	读期间发生CRC错误
20h	软盘控制器或驱动器有问题
40h	查找失败
80h	超时——软驱没有响应

地址	描述	大小	平台
40:42h	磁盘和软盘控制器状态寄存器0	字节	所有

所有系统上的软盘控制器都用到这个字节以及下一个字节。对于一个带PC/XT类型硬盘控制器的系统，前四个字节也用来保存出错时的控制器数据。参看第11章获取有关四个字节的详细信息。

软盘操作结束后，BIOS中断13h处理程序从控制器读取状态寄存器0。这个字节定义如下：

位		
	7=x 6=x	中断代码
		第7位 第6位
		0 0=命令正常结束
		0 1=执行中命令非正常结束
		1 0=无效命令
		1 1=非正常结束，正在读时更换软盘
	5=1	命令查找结束
	4=1	驱动器错误——要么从软驱收到一个错误信息，要么就是因磁道0信号错误而不能完成重新标准化
	3=1	驱动器未准备好——下面的情况会发生该错误：当驱动器没有准备好时试图读/写；或者，试图读/写单面软盘的side1（这种软盘是160K，非常古老了）

2=x	发生中断时的头状态	
1=x	中断发生时的驱动器	
0=x	第 1 位	第 0 位
	0	0=驱动器 0
	0	1=驱动器 1
	1	0=驱动器 2（仅 PC/XT）
	1	1=驱动器 3（仅 PC/XT）

地址	描述	大小	平台
40:43h	软盘控制器状态寄存器 1	字节	所有

在软盘操作结束时，BIOS 中断 13h、处理程序从控制器读取状态寄存器 1，然后保存在这里。

位	7=1	控制器试图访问软盘最后一磁柱之外的扇区
	6=0	未使用
	5=1	读取时检测到 CRC 错误
	4=1	DMA 过载运行——DMA 同控制器之间传送数据不够快，以致控制器发生超时
	3=0	未使用
	2=1	数据问题——找不到扇区，或者不能读取磁盘 ID 区
	1=1	在写保护时试图定或格式化
	0=1	丢失地址标签——两次读取后试图读取软盘 ID 的地址标签失败

地址	描述	大小	平台
40:44h	软盘控制器状态寄存器 2	字节	所有

在软盘操作结束后、BIOS 中断 13h 处理程序从控制器读取状态寄存器 2，然后保存在这里。

位	7=0	未用
	6=1	在读取软盘数据器核对的数据时，检测到数据地址标签被删除
	5=1	在数据检测到 CRC 错误
	4=1	错误的柱柱——被读取的磁柱不同于控制器指定的磁柱
	3=1	在检验过程中、满足相等条件
	2=1	在检验过程中，控制器在磁柱上找不到符合检验条件的扇区

1=1	坏磁柱——被读取的磁柱不同于由控制器指定的磁柱，并且值为FFh
0=1	读软盘时找不到地址标签（或者地址标签已被删除）

地址	描述	大小	平台
40:46h	软盘控制器的磁头数字	字节	所有

在软盘操作结束后，BIOS中断13h处理程序读取头数字。值0表示头0用于软盘的第0面，值1表示头1用于软盘的第1面。这就是命令结束时的头数字。某些BIOS不保存这个字节。

地址	描述	大小	支持平台
40:47h	软盘控制器的扇区号	字节	所有

在软盘操作结束后，BIOS中断13h处理程序读取扇区号。这就是命令结束时的扇区号。某些BIOS不保存这个字节。

地址	描述	大小	平台
40:48h	软盘控制器字节写	字节	所有

在软盘操作结束后，BIOS中断13h处理程序读取写入到一个扇区的字节数。某些BIOS不保存这个字节。

地址	描述	大小	平台
40:49h	视频模式	字节	所有

这里保存了活动的视频模式号。在第9章列出了不同适配器的可能模式。

地址	描述	大小	平台
40:4Ah	视频列数	字节	所有

设置了视频模式后，视频模式就将列数保存在该字中。在图形模式下，该值代表了可用视频BIOS功能来写文本的可能列数。对于80列模式，该值存为50h。

地址	描述	大小	平台
40:4Ch	视频 每页字节数	字	所有

视频BIOS保存上次视频设置下的每页字节数。

记住，不同的系统和视频BIOS制造商在该字中保存的字可能有点不同。例如，在模

式 3 下显示器可以在屏幕上显示 2000 个字符（包括属性共 4000 个字节）。在每页之间有 96 个未用字节。某些系统会显示每页 4000 个字节，而另外一些系统则显示有 4096 个字节。

地址	描述	大小	平台
40:4Eh	视频——当前页偏移量	字	所有

视频 BIOS 在这个字节中保存了当前页的偏移量。对于所有模式的第零页，保存起来的偏移量是 0。在视频模式 3 下，第 1 页的偏移量可能是 1000h（4096）。

地址	描述	大小	平台
40:50h	视频——第 0 页光标位置	字	所有

该字为第 0 页保存了光标位置。低字节保存了光标所在的行而高字节则保存了列，它们都是从零开始的。左上角所代表的字是 0000。对于一个 80 列 25 行的视频模式，右下角光标位置的值是 4F18h。

地址	描述	大小	平台
40:52h	视频——第 1 页光标位置	字	所有

该字保存了第 1 页的光标位置。详细细节参看地址 40:50h。

地址	描述	大小	平台
40:54h	视频——第 2 页光标位置	字	所有

该字保存了第 2 页的光标位置。详细细节参看地址 40:50h。

地址	描述	大小	平台
40:56h	视频——第 3 页光标位置	字	所有

该字保存了第 3 页光标位置。详细细节参看地址 40:50h。

地址	描述	大小	平台
40:58h	视频——第 4 页光标位置	字	所有

该字保存了第 4 页光标位置。详细细节参看地址 40:50h。

地址	描述	大小	平台
40:5Ah	视频——第 5 页光标位置	字	所有

该字保存了第 5 页光标位置。详细细节参看地址 40:50h。

地址	描述	大小	平台
40:5Ch	视频——第 6 页光标位置	字	所有

该字保存了第 6 页光标位置。详细细节参看地址 40:50h。

地址	描述	大小	平台
40:5Eh	视频——第 7 页光标位置	字	所有

该字保存了第 7 页光标位置。详细细节参看地址 40:50h。

地址	描述	大小	平台
40:60h	视频——光标形状	字	所有

该字保存了光标的形状。低字节保存了结束扫描线数，而在 40:61h 处的高字节保存了开始扫描线数。在 VGA 的模式 3 下字符块通常是 16 条扫描线高。关掉光标，将结束扫描线设在起始扫描线之上。表 6-1 显示了该模式下几种常用光标的形状。

表 6-1 光标形状举例

形 状	值
底部的两线光标	0607h
下半个光标	0307h
上半个或四分之一光标	0003h
全光标	0007h
空白光标	0100h

地址	描述	大小	平台
40:62h	视频——活动的显示页	字	所有

视频 BIOS 在这个字节中保存当前的页号。设置任意一种模式后的缺省页是 0。记住，允许的页数在不同的模式有所不同，这取决于视频硬件。

地址	描述	大小	平台
40:63h	视频——I/O 端口的基号	字	所有

该字保存活动显卡的基本端口号。表 6-2 详细说明了所用到的标准端口号，这取决于所使用的适配器和监视器。

表 6-2　基本视频端口地址

适配器	单色监视器用到的端口	彩色监视器用到的端口
MGA/HGA	3B4h	n/a
CGA	3D4h（合成单色）	3D4h
EGA	3B4h	3D4h
VGA/SVG	3B4h	3D4h
AXGA	3B4h	3D4h

地址	描述	大小	平台
40:65h	视频　内部模式寄存器	字节	所有

在 CGA 类型的适配卡端口 3D8h 和 MDA 适配卡的端口 3B8h 有一个视频适配器的内部模式寄存器。这个字节中就存有发送给这个寄存器的最后的值。EGA/VGA 适配器没有等同的端口，但是 EGA/VGA 视频 BIOS 却使这个值对视频模式 0 到 7 向下兼容。表 6-3 列出了不同视频模式下的标准值。

位	7=0	未使用
	6=0	未使用
	5=0	屏幕属性的第 7 位控制背景密度
	1	屏幕属性的第 7 位控制闪烁
	4=1	
	3=1	模式 6，640×200 图形操作，2 色（单色）
	2=0	开放视频信号
	1	彩色运行
	1=1	单色运行
	0=1	模式 4 和 5，320×200 图形操作 文本模式 2 和 3，80 列×25 行

表 6-3　第种视频模式的内部模式位

视频模式	描　述	位							
		7	6	5	4	3	2	1	0
0	40×25，文本，单色	0	0	?	0	?	1	0	0
1	40×25，文本，彩色	0	0	?	0	?	0	0	0
2	80×25，文本，单色	0	0	?	0	?	1	0	1
3	80×25，文本，彩色	0	0	?	0	?	0	0	1
4	320×200 图形，单色	0	0	x	0	?	1	1	0
5	320×200 图形，彩色	0	0	x	0	?	1	1	0
6	640×200 图形，单色	0	0	x	0	?	1	1	0
7	80×25，文本，单色	0	0	?	x	?	x	x	1

注：x =无效位

? =取决于用户选项，缺省 1

地址	描述	大小	平台
40:66h	视频—色彩调色板	字节	所有

这个字节中保存了送往视频适配器内部色彩寄存器的最后一个值，这个寄存器位于CGA类型适配器的端口3D9h。EGA/VGA适配器没有等同的端口，但是EGA/VGA视频BIOS却使这个值对视频模式0到6向下兼容。

位	7=0	未用
	6=0	未用
	5=0	模式5的前景色——绿/红/黄
	1	模式5的前景色——青/洋红/白
	4=0	普通背景色
	1	加强背景色——仅文本模式
	3=x	40×25文本模式下的加强边界色，以及图形模式5下的背景色
	2=x	红
	1=x	绿
	0=x	蓝

地址	描述	大小	支持平台
40:67h	通用偏移量	字	XT+
	磁带、时间计数器	字	PC

在XT+系统上，该字用作一系列目的的临时变量，包括286+系统保护模式下的堆栈指针，ROM初始化指针的偏移部分，以及其他临时功能。

在AT+系统上，它和地址69h一起也作为某些重启操作的返回指针。重启动后，CMOS关闭字节0Fh指导POST如何工作。提供了一系列选项来跳转到这两个字所指向的远地址。这一点允许系统以保护模式切换到实模式，这只能通过重启处理器实现。

在PC系统上，该字含有一个计数值，用来为早已过时的磁带驱动器计时。

地址	描述	大小	支持平台
40:69h	通用段	字	XT+
	磁带、CRC寄存器	字	PC

在XT+系统上，该字用于一系列目的临时变量，包括286+系统保护模式下的堆栈指针、ROM初始化指针的段部分以及其他临时功能。

在AT+系统上，它和地址40:67h作为某些重启操作的返回指针。参见地址40:67h获得其他细节。

在PC系统上该字用于磁带的CRC操作。

地址	描述	大小	支持平台
40:6Bh	最后一个中断	字节	XT+
	磁带、最后一个读取值	字节	PC

在 XT+系统上，当出现无效中断时，这一位保存了硬件 IRQ 位。部分 BIOS 初始化进程会将所有未使用的中断向量指向某个 BIOS 中断处理程序。该处理程序检查中断是否由硬件事件引发。如果不是，该值中就装入 FFh。如果确实发生了一个硬件中断，那么该硬件中断保存在这里。

位	7=1	发生了 IRQ7 硬件中断
	6=1	发生了 IRQ6 硬件中断
	5=1	发生了 IRQ5 硬件中断
	4=1	发生了 IRQ4 硬件中断
	3=1	发生了 IRQ3 硬件中断
	2=1	发生了 IRQ2 硬件中断（IRQ8 到 15，AT 上）
	1=1	发生了 IRQ1 硬件中断
	0=1	发生了 IRQ0 硬件中断

例如，硬件中断发生在 IRQ3（中断 0Bh），且该向量指向未使用的 BIOS 中断处理程序。未使用中断的处理程序检测到是 IRQ3 发出请求，并且在这个字节中装入二进值 00001000。发生 IRQ8 到 15 中断（仅 AT+）中的任意一个时，第 2 位将被设置。

在 PC 平台上，这个字节保存了读自磁带的最后一个数据字节。

地址	描述	大小	支持平台
40:6Ch	时钟滴答计数	双字	所有

该双字中保存了当前的时钟滴答计数，对中断 8 BIOS 时钟滴答服务程序的每次调用都会递增这个计数值。每隔 54.9ms 时钟 0 翻转一次时，都会触发中断 8。

如果计数值达到了 1800B2h，也就是计时满 24 小时，计数器会置零。另外，可以在地址为 40:70h 的 RAM 中设置时钟滴答翻转标志。上电自检期间 BIOS 会对计数器清零。对于 AT+系统来说，操作系统一般会读取 CMOS 的时钟值，并依据这个时钟值设置时钟滴答计数，这样，计数翻转总是发生在午夜。

地址	描述	大小	支持平台
40:70h	时钟滴答翻转标志	双字	所有

每次位于 40:6Ch 处的时钟滴答计数翻转时，该标志都会被设置为 1。一般每 24 小时置 1 一次。触发 BIOS 日期服务程序的功能 0 将会把这个标志重新设置为 0。

关于这个标志有许多值得澄清的地方。这个翻转只有两种状态，分别是 0 和 1，它永远也不会超过 1。一般情况下，操作系统会周期性地调用 BIOS 日期中断 1Ah 的功能 0，来读取当前的滴答计数和这个标志。然后，BIOS 服务处理程序会清除这个标志。如果某个应用程序在操作系统之前触发中断 1Ah 的功能 0，就会出问题。这时，系统会得不到正确的标志位信息，午夜时操作系统也不会改变日期。因此程序员应避免使用中断 1Ah 的功能 0。可以从位于 40:6Ch 和 40:70h 的内存直接读取这些值。

地址	描述	大小	支持平台
40:71h	键盘 Control-Break 标志	字节	所有

当组合键 Ctrl-Break 按下后，第 7 位被设置。另外，在这个时候，键盘 BIOS 触发中断 1Bh 处理 Ctrl-Break。

位	7=1	按下了 Control-Break 键
	6~0=x	未使用，便可能处于任意状态

地址	描述	大小	平台
40:72h	热启动标志	字	所有

该字用于标志是一次热启动。当同时按下三个键 Alt、Ctrl 和 Delete 键时，键盘将该热启动标志设置为 1234h。在 PC 和 XT 系统上键盘 BIOS 然后跳到上电自检（POST）程序的起点。在 AT+系统上，键盘 BIOS 会触发硬件 CPU 重启，这也会运行 POST 程序。POST 检查热启动标志。如果该值是 1234h，那么就会跳过内存测试，并且如果安装了一个可任选的 VGA+适配器，那么也会跳过内存测试。

PS/2 系统和某些 AT+系统的 BIOS 用到另外两个值。值 4321h 表示 POST 跳过内存测试，并且不改变内存。值 64h 用于生产测试。当 POST 下发现热启动标志等于 64h，那么 POST 测试会持续循环下去。这对于在内置过程中检测问题非常有用，也有助于标识间歇性问题。

地址	描述	大小	支持平台
40:74h	硬盘上最后一次操作的状态	字节	XT+

硬盘操作完成之后，该操作的状态就被保存在该字中，表 6-4 列出的返回码。并不是所有的硬盘服务程序都会生成所显示的第一个码。参看第 11 章获得有关每个返回码的详细信息。

表 6-4　硬盘返回状态码

十　六	描　述
0	操作成功

续表

十　六	描　述
1	坏命令或参数
2	丢失了地址标签
3	可移走媒质的写保护
4	所请求的扇区没有找到
5	重启失败
6	可移走媒质磁盘发生了改变
7	驱动器参数活动性失败
8	DMA 过载
9	数据边界错误
Ah	检测到了坏扇区标志
Bh	检测到了坏磁道
Dh	格式化时扇区数不合法
Eh	检测到了控制数据标签
Fh	DMA 仲裁水平越界
10h	读操作时出现了 ECC 错误
11h	ECC 纠正数据错误
20h	硬盘控制器或驱动器出现问题
31h	在可移走媒质驱动器内没有媒质
40h	查找操作失败
80h	超时
AAh	驱动器没有准备好或者没有选择驱动器
B0h	没有锁住驱动器内的磁盘组
B1h	锁住驱动器内的磁盘组
B2h	不可移动磁盘组
B3h	正在使用磁盘组
B4h	锁计数越界
B5h	有效的弹出请求失败
BBh	出现了未定义的错误
CCh	在所选中的驱动器上的写错误
E0h	设置了态错误标志，控制错误码设置为零（无错误！）
FFh	传感操作失败

地址	描述	大小	平台
40:75h	硬盘——数目	字节	XT+

这个字节指示POST检测到的硬盘数目。PC和XT BIOS以及1996年以前的在多数其他BIOS最大支持两个硬盘驱动器。大多数较新的BIOS（1996年以后）支持四个硬盘。自带BIOS的磁盘控制器只受控制器容量的限制。我发现一些系统在这个硬盘驱动器的计数器中包括了IDE CD-ROM驱动器，这是不正确的。

地址	描述	大小	平台
40:76h	硬盘——控制字节	字节	XT+

这个字节为硬盘BIOS保存了临时的控制标志。在许多硬盘操作中，BIOS从偏移为8处的磁盘参数表控制字节中取出值装入到该字节中。如果第6位或第7位高电平，表示不再重试发生的错误。在某些系统上，第3位用于表示驱动器有八个以上的磁头。直接支持ESDI驱动器的PS/2 BIOS不使用这个字节。

位		
	7=1	磁盘出错时禁止重试
	6=1	磁盘出错时禁止重试
	5=0	未使用
	4=0	未使用
	3=1	驱动器有八个以上的磁头
	2=0	未使用
	1=0	未使用
	0=0	未使用

地址	描述	大小	平台
40:77h	硬盘——端口偏移量	字节	XT

这个字节含有硬盘端口的临时偏移地址。它仅用于XT硬盘BIOS和一些早期的硬盘适配卡BIOS中。

地址	描述	大小	平台
40:78h	并行打印机1超时	字节	XT+

这个字节保存了打印机中断17h处理程序用到的一个固定值。当打印机BIOS抽打印机输出一个字符时，BIOS用该值作为打印忙时的最大延迟。如果该延迟后打印机仍然忙，BIOS会返回一个失败错误码。

大多数系统的POST会将该值初始化为14h。大多数BIOS的时间用完周期基于指令延迟。该持续时间因系统时钟速度不同而不同，通常每计数一次延迟2~8μs。对于一个定时

为 14h 的系统来说，这意味着等待 40~160μs。所有的 AT+系统和一些 XT BIOS 都使用这个字。

地址	描述	大小	平台
40:79h	并行打印机 2 超时	字节	XT+

该字为并行打印机 2 保存了一个定时 XT+值。参看 40:78h 获得细节。

地址	描述	大小	平台
40:7Ah	并行打印机 3 超时	字节	XT+

该字为并行打印机 2 保存了一个定时 XT+值。参看 40:78h 获得细节。

地址	描述	大小	平台
40:7Bh	并行打印机 4 超时	字节	XT
	VDS 支持	字节	AT+

该字为并行打印机 4 保存了一个定时值。参看 40:78h 获得细节。某些系统并不支持第四个打印机。这种情况下，未使用这个字节或者用于其他不相关的目的。

在 AT+系统上，虚 DMA 服务（VDS）使用第 5 位来表明 VDS 是否被激活。为了帮助 VDS 提供器恰当地串起中断 4Bh，MCA 系统把第 3 位用作标志位。

仅仅 MCA 系统，如果第 3 位是 1，那么 VDS 提供者一定是串起了中断 4Bh。这意味着 VDS 提供者将未用的 VDS 功能号传递给老的中断 4Bh 程序处理程序。如果第 3 位是 0，那么必须在 0:12Ch 处检查中断 4Bh 的段向量。如果在 0:12Ch 处的段值不是 0、E000 或 F000，那么 VDS 提供者一定串起了中断 4Bh。如果段值是 0、E000 或 F000，那么 VDS 一定没有串起中断 4Bh。这种情况下，仅仅只是返回无效 VDS 函数，而忽略了先前中断 4Bh 向量。

当未使用打印机超时时

位		
	7=0	未用
	6=0	未用
	5=1	VDS 服务支持
	4=0	未用
	3=1	中断 4Bh 要求串起（参看文字部分）
	2=0	未用
	1=0	未用
	0=0	未用

地址	描述	大小	平台
40:7Ch	串行口 1 超时	字节	XT+

串行口中断 14h 处理程序使用的固定定时值被保存在这个字节中。当串行口 BIOS 发送一个字符时，BIOS 使用该值作为等待激活数据终端准备好（DTR）线和要求发送（RTS）线的最大延迟。然后，BIOS 使用这个程序时值作为等待控制器准备好的时间。任意一种定时到时，中断处理程序都会返回一个错误条件。

当接收字符时发生类似的动作。首先，该值作为定时等待数据设置准备好（DSR）线被激活。然后，BIOS 又利用该定时值等待串行口控制器状态表明接收缓冲区有一个字符。如果在定时期间两种情况都没有发生，那么中断处理程序返回一个错误条件。

大多数系统的 POST 会将该值初始化为 1，每计数一次约等于 2ms 的。由于定时依赖于指令延迟，所以延迟长度将随系统时钟速度和 BIOS 的执行情况的不同而不同。

地址	描述	大小	平台
40:7Dh	串行口2超时	字节	XT+

这个字节为串行口 2 保存了定时值。参看地址 40:7Cb 获得详细描述。

地址	描述	大小	平台
40:7Eh	串行口3超时	字节	XT+

这个字节为串行口 3 保存了定时值。参看地址 40:7Ch 获得详细描述。

地址	描述	大小	平台
40:7Fh	串行口4超时	字节	XT+

这个字节为串行口 4 保存了定时值。参看地址 40:7Ch 获得详细描述。

地址	描述	大小	平台
40:80h	键盘——缓冲区起点	字	XT+

该字保存了段 40h 处的键盘缓冲区的起点指针，缺省地址是 1Eh。当键加入和移走缓冲区时，该值不会改变，这一点不同于键盘的头指针和尾指针。

地址	描述	大小	平台
40:82h	键盘——缓冲区终点	字	XT+

该字保存了段 40h 处的键盘缓冲区的起点指针，缺省地址是 3Eh。当键加入和移走缓冲区时，该值不会改变，这一点不同于键盘的头指针和尾指针。

地址	描述	大小	平台
40:84h	视频——行数	字节	EGA+

这个字节保存了屏幕上的行数。一个 25 行的显示器的保持值是 18h。仅适用于带有自己的 BIOS ROM 的显卡，例如 EGA/VGA 等。

地址	描述	大小	平台
40:85h	视频 --每字符的像素	字节	EGA+

该字保存了每个字符的扫描行数。对一个普通的 VGA 文本字符，16 像素高，9 像素宽，保存的值是 10h。它仅适用于自带 BIOS ROM 的视频适配器，例如 EGA/VGA 等。

地址	描述	大小	平台
40:87h	视频 --选项	字节	EGA+

视频适配器的选项保存在这个字节中，它仅适用于自带 BIOS ROM 的适配器，例如 EGA/VGA 等。

位		
	7=x	拷贝自上次视频模式设置的第 7 位
	0	当设置视频模式时清除显示缓冲区 RAM
	1	当设置视频模式时不清除显示缓冲区 RAM
	6=x 5=x	显卡上的内存 第 6 位　第 5 位 0　0=64K 0　1=128K 1　0=192K 1　1=256K 或更多
	4=x	未用
	3=1	视频适配器激活
	2=1	等待显示开放
	1=0	附带彩色显示器（使用端口 3Dxh）
	1	附带单色显示器（使用端口 3Bxh）
	0=0	光标的 CGA 模拟（参看 int 10h，功能 12h，光标大小控制）

地址	描述	大小	平台
40:88h	视频 开关	字节	EGA+

高级的视频适配器的开关和特性连结器的设置保存在这个字节中。它只适用于自带 BIOS ROM 的视频适配器，例如，EGA/VGA。

适配器开关位 0 到 3 直接与 EGA/VGA 开关 SW1~SW4 相关。当开关处于关闭位置时，保存在开关的值是零。位 0 到 3 控制启动时激活那个适配器，并将之作为基本显示。另外，开关选择支持可任选的 CGA 或 MDA 类型的附加显示器。

许多新的适配器清除了这些开关而改用视频器上的不变内存。这种情况下，适配器的安装软件控制这些位。

记住，在视频启动操作中开关只能读一次。然后，某些视频选取项可能会改变低四位的状态。例如，中断 10h 功能 12h 的子功能 30h 就会改变扫描行数，并且在这个字节的低四位中装入新的值。

现在很少用到特性连接器，它有两条线，在视频 BIOS 的 POST 运行时会读到它们。特性连接器也有一个输出状态线。当输出状态线被设置为 0 时，先读取两条特性线 0 和 1。然后，该状态线被设置为 1，再次读取两条特性线。特性线的值保存在第 4 位到第 7 位。

位						
7=x	输出状态为 0 时，特性连接器上特性 0 线的值					
6=x	输出状态为 0 时，特性连接器上特性 1 线的值					
5=x	输出状态为 1 时，特性连接器上特性 0 线的值					
4=x	输出状态为 1 时，特性连接器上特性 1 线的值					
3=x	适配器类型开关设置					
2=x	位	3	2	1	0	主	次
1=x		0	0	0	0	=MDA	彩色 40×25
0=x		0	0	0	1	=MDA	彩色 80×25
		0	0	1	0	=MDA	高分辨率 80×25
		0	0	1	1	=MDA	高分辨率增强
		0	1	0	0	=CGA40×25	单色
		0	1	0	1	=CGA80×25	单色
		0	1	1	0	=彩色 40×25	MDA
		0	1	1	1	=彩色 80×25	MDA
		1	0	0	0	=高分辨率 80×25	MDA
		1	0	0	1	=高分辨率增强	MDA
		1	0	1	0	=单色	CGA40×25
		1	0	1	1	=单色	CGA80×25
		1	1	0	0	=未使用	
		1	1	0	1	=未使用	
		1	1	1	0	=未使用	
		1	1	1	1	=未使用	

地址	描述	大小	平台
40:89h	视频 保存区 1	字节	EGA+

这个字节为高级的视频 BIOS、VGA 或后来的类型，保存了有关信息。某些视频 BIOS 供应商可能将这个字节用作他用。两条扫描线第 7 位和第 4 位中在模式设置之后有效。设置了新的扫描线后和 7 位被清零。

位	7=0	扫描线 350 或 400（参看第 4 位）
	1	扫描线 200（参看第 4 位=0）
	6=x	IBM 以及大多数其他系统没有使用这一位，只有某些供应商才提供专项功能
	5=0	未用
	4=0	扫描线 200 或 350（参看第 7 位）
	1	扫描线 400（参看第 7 位=0）
	3=0	在任意一次设置模式时都将彩色寄存器设置为缺省值
	1	在任意一次设置模式时不改变彩色寄存器
	2=0	附带彩色显示器
	1	附带单色显示器
	1=0	正常色彩
	1	将色彩寄存器的值转化为灰度级
	0=1	附带显示器支持所有的模式

地址	描述	大小	平台
40:8Ah	视频——保存区 2	字节	VGA+

这个字节保存了某些高级视频 BIOS、VGA 或后来的类型的有关信息。

地址	描述	大小	平台
40:8Bh	软盘——配置数据	字节	AT+

这个字节为软驱保存了数据传送率。如果控制器已经指定了合适的数据传送率，那么它用来保存时间。

如果媒体类型没有指定，那么在操作开始时就使用该数据传送率。使用不同的传送率部分可以正确地确定软盘类型。对于一个带有 1.44M 的驱动器和 720K 的驱动器，BIOS 先测试 500Kbps 传送率。如果失败，BIOS 就测试 250Kbps 的传送率。对于 720KB 的软盘，250Kbps 的传送率是正确的。

并不是所有的 BIOS 都保存最后一个驱动器的步进速率。在这各情况下，这些系统不使用第 4 位和第 5 位。

位	7=x 6=x	送到软盘控制器的最后数据率	
		第 7 位	第 6 位
		0	0=500Kbps
		0	1=300Kbps
		1	0=250Kbps
		1	1=尚未设置速率，或者在某些系统上表示 1Mbps

5=x 4=x	送到软盘控制器的最后驱动器步进速率	
	第 5 位	第 4 位
	0	0=8ms
	0	1=7ms（常用）
	1	0=6ms（常用）
	1	1=5ms
3=x 2=x	开始操作时设置的数据速率	
	第 3 位	第 2 位
	0	0=500Kbps
	0	1=300Kbps
	1	0=250Kbps
	1	1=1Mbps（如果支持）
1=x	大多数未用，某些复制品上功能也未知	
0=x	大多数未用，某些复制品上功能也未知	

地址	描述	大小	平台
40:8Ch	硬盘 状态	字节	AT+

来自硬盘控制器端口 1F7h 的实际状态保存在这个字节中。某些 BIOS 不支持这项功能。参看第 11 章中的端口 1F7h。

地址	描述	大小	平台
40:8Dh	硬盘 错误	字节	AT+

来自硬盘控制器端口 1F1h 的实际状态保存在这个字节中。某些 BIOS 不支持这项功能。参看第 11 章中的端口 1F1h。

地址	描述	大小	平台
40:8Eh	硬盘 任务完成标志	字节	AT+

来自硬盘控制器任务开始时，硬盘 BIOS 将该值设置为 0。当任务完成时，硬盘控制器硬件通过 IRQ14、中断 76h 发出信号。中断 76h 处理程序然后将这个字节设置为 0FFh。早期的 BIOS 不支持该项功能。

地址	描述	大小	平台
40:8Fh	软盘 控制器信息	字节	AT+

这个字节用作软盘控制器信息。大多数 AT+ BIOS 支持这项功能。

位		
	7=0	未使用
	6=1	确定了驱动器 1 的类型
	5=1	驱动器 1 是多速率的
	4=1	驱动器 1 带有一条软盘改变检测线
	3=0	未使用
	2=1	确定了驱动器 0 的类型
	1=1	驱动器 0 是多速率的
	0=1	驱动器 0 带有一条软盘改变检测线

某些早期的 AT BIOS 文档的陈述是不正确的，它们说这个字节的功能是指示系统何时带有硬盘/软盘适配卡。早期的 AT 系统未使用这个字节。

地址	描述	大小	平台
40:90h	软盘一介质状态	字节	AT+

这个字节为 AT+系统保存了软盘 0 的介质状态信息。值 0 表示没有驱动器。

位				
	7=x 6=x	数据传送率		
		第 7 位	第 6 位	
		0	0	=500Kbps
		0	1	=300Kbps
		1	0	=250Kbps
		1	1	=1Mbps
	5=1	要求双倍步进（1.2M 驱动器，带 360K 软盘）		
	4=1	已知软驱中的介质		
	3=0	未使用		
	2=x	确定上次访问		
	1=x	位 2	1	0
	0=x	0	0	0=在 360K 的驱动器中是 360K 的介质
		0	0	1=在 1.2M 的驱动器中是 360K 的介质
		0	1	0=在 1.2M 的驱动器中是 360K 的介质
		0	1	1=在 360K 的驱动器中已知 360K 的介质
		1	0	0=在 1.2M 的驱动器中已知 360K 的介质
		1	0	1=在 1.2M 的驱动器中已知 1.2M 的介质
		1	1	0=未使用状态
		1	1	1=在 720K 的驱动器中是 720K 的介质或者 1.44M 的驱动器中是 1.44M 的介质

地址	描述	大小	平台
40:91h	软盘 1 一介质状态	字节	AT+

这个字节为 AT+系统保存了软盘 1 的介质状态信息。值 0 表示没有驱动器

位		
	7=x 6=x	数据传送率 第 7 位 第 6 位 0 0=500Kbps 0 1=300Kbps 1 0=250Kbps 1 1=1Mbps
	5=1	要求双倍步进（1.2M 驱动器，带 360K 软盘）
	4=1	已知软驱中的介质
	3=0	未使用
	2=x	确定上次访问
	1=x	
	0=x	位 2 1 0 0 0 0=在 360K 的驱动器中是 360K 的介质 0 0 1=在 1.2M 的驱动器中是 360K 的介质 0 1 0=在 1.2M 的驱动器中是 360K 的介质 0 1 1=在 360K 的驱动器中已知 360K 的介质 1 0 0=在 1.2M 的驱动器中已知 360K 的介质 1 0 1=在 1.2M 的驱动器中已知 1.2M 的介质 1 1 0=未使用状态 1 1 1=在 720K 的驱动器中是 720K 的介质或者 1.44M 的驱动器中是 1.44M 的介质

地址	描述	大小	平台
40:92h	软盘 0 操作起始状态	字节	AT+

这个字节为软盘 0 保存了起始状态信息，某些 BIOS 没有使用这个字节。

位		
	7=x 6=x	数据传送率 第 7 位 第 6 位 0 0=500Kbps 0 1=300Kbps 1 0=250Kbps 1 1=1Mbps
	5=x	未知功能
	4=x	未知功能
	3=0	未使用
	2=1	确定了驱动器类型

1=1	驱动器是多速率	
0=1	驱动器带有软盘改变检测线	

地址	描述	大小	平台
40:93h	软盘 1——操作起始状态	字节	AT+

这个字节为软盘 1 保存了起始信息，某些 BIOS 没有使用这个字节。

位			
7=x 6=x	数据传送率		
	第 7 位	第 6 位	
	0	0=500Kbps	
	0	1=300Kbps	
	1	0=250Kbps	
	1	1=1Mbps	
5=x	未知功能		
4=x	未知功能		
3=0	未使用		
2=1	确定了驱动器类型		
1=1	驱动器是多速率		
0=1	驱动器带有软盘改变检测线		

地址	描述	大小	平台
40:94	软盘 0——当前磁柱	字节	AT+

这个字节为软盘 0 保存了当前所在的磁柱。

地址	描述	大小	平台
40:95	软盘 1——当前磁柱	字节	AT+

这个字节为软盘 1 保存了当前所在的磁柱。

地址	描述	大小	平台
40:96	键盘——状态标志 3	字节	AT+

这是高级键盘的状态字节。第 6 位和第 7 位用于读取键盘 ID。该 ID 通常是值 ABh 紧接着 41h。重启后会检查是否附带键盘和是否有键盘响应。

位		
7=1	正在读取键盘 ID 的两字节	
6=1	上次输入的一个键盘字节是 ID 的第一个字符，现在正在读取第二个	

	5=1	取得键盘 ID（重启）后强制 Num-Lock
	4=1	101 或 104 键键盘
	3=1	按下了右 Alt 键
	2=1	按下了右 Control
	1=1	当扫描码是 E0h 时，设置该标志，并从键盘读取下一个字节
	0=0	当扫描码是 E1h 时，设置该标志，并从键盘读取下一个字节
	=1	安装了键盘（早期的 AT）

地址	描述	大小	平台
40:97h	键盘 - 状态标志 4	字节	AT+

这个字节保存了键盘的附加信息以及 LED 状态信息。

位	7=1	键盘传送错
	6=1	程序中更新了 LED
	5=1	键盘发送了一个“重新发送”字节，FEh
	4=1	键盘发送了一个“接受到”字节，FAh
	3=0	未用
	2=1	CapLock LED 开
	1=1	NumLock LED 开
	0=1	ScrollLock LED 开

地址	描述	大小	平台
40:98h	用户等待标志指针	双字	AT+

通用的 BIOS 服务，中断 15h 的功能 83h，提供了一个用户时钟服务。在用户的调用函数中带有等待的微秒数以及指向用户标志的指针。用户指针以段：偏移量的形式保存在这个双字中。

用户将标志字节的第 7 位初始化为 0。如果用户指定的时间周期过期了，那么这个用户标志位，第 7 位会被设置为 1。参看 40:9Ch 获到相关计数器的更多信息。

地址	描述	大小	平台
40:9Ch	用户等待计数	双字	AT+

这个双字为用户时钟服务，中断 15h 的功能 83h 提供一个减数计数时钟。该功能将用户等待的时间以微秒的形式保存在这个双字中。激活后，这个计数器的值会每 976 μs 减少 976。如果计数值小于 0，用户等待标志字节第 7 位就会被设置为 1。参看 RAM 地址 40:98h 获取有关用户等待标志指针的信息。

CMOS 实时时钟的周期性中断输出连接到 IRQ8。正常安装时，实时时钟以 1,024Hz 的速率生成周期性信号。这意味着第 976 微秒发生一次中断。IRQ8 连接中断 70h，这个中断将这个双字计数器减少 976。

最大时间延迟是 FFFF FFFFh。转化为时间为 71 分钟。该机制存在两个精确性误差。该计数每 976.562 微秒才改变 976。这意味着减数计时要快百分之 0.06。当使用较小的值时，该效应更严重。作为一个极端的例子。为了计时一毫秒，该计数器装入 1,000。时钟那么会在两次中断后陷入，也就是在 1.96 毫秒后陷入，这个结果产生了百分之 95 的误差！

地址	描述	大小	平台
40:A0h	等待标志	字节	AT+

这个字节用于中断 15h，功能 86h 的延迟服务。调用该功能时带有一个微秒形式的双字延迟。该标志与等待计数器和等待标志指针有关，后者在地址 40:9Ch 和 40:98h 第 0 位用于控制等待的状态。中断 15h，功能 86h 将用户等待标志指针设置为这个字节。这意味着当时间周期到期时会设置第 7 位。

位	7=1	等待的时间到期（如果该地址保存在 40:98h 处的用户等待标志指针中）
	6=1	未使用
	5=1	未使用
	4=1	未使用
	3=1	未使用
	2=1	未使用
	1=1	未使用
	0=0	现在没有等待
	1	正在等待

地址	描述	大小	平台
40:A1h	局域网字节	7 个字节	AT+

IBM 详细说明了这七个字节为局域网而保留。系统 BIOS 使用没有它们。没有说明书指出哪个专用网是如何使用这些字节的。

地址	描述	大小	平台
40:A8h	视频参数控制块指针	双字	EGA+

该双字指向高级视频系统，例如 VGA 适配器的附加指针表。表 6-5 列出了这个指针表。以段：偏离量的形式保存该指针。

表6-5 视频保存表

偏移量	类型	指针指向
0	双字	视频参数
4	双字	参数保存区
8	双字	字母数字字符集
Ch	双字	图形字符集
10h	双字	第二个保存表指针（参看表6-6）
14h	双字	未使用（0:0）
18h	双字	未使用（0:0）

表6-5中在偏移量为10h处的指针指向第二个保存表指针，有关信息如下：

表6-6 第二个视频保存表

偏移量	类型	指针指向
0	字	本表中的全部字节
2	双字	组合代表码
6	双字	第二个字母字符集
Ah	双字	用户调色板表
Eh	双字	未使用（0:0）
12h	双字	未使用（0:0）
16h	双字	未使用（0:0）

参看第9章获取详细信息。

地址	描述	大小	持平台
40:CEh	时钟自1980以来的天数	字	AT+

在某些BIOS中，该字保存了从1980所到现在的天数。

地址	描述	大小	平台
50:00h	打印屏幕状态	字节	所有

当按下打印屏幕（Print-Screen）键时，低级键盘BIOS初始化打印屏幕功能。键盘BIOS触发中断5，打印屏幕处理程序。该处理程序通过该状态字节控制打印屏幕操作的状态。这个字节可通过地址40:100h或50:0访问。

这个字节保存下列值及相关功能：

00h= 打印屏幕准备好

01h= 正在打印屏幕

FFh= 打印时出错——缺纸。来自打印机的错误信号、或者等待超时

扩展 BIOS 数据区

某些系统的 BIOS 在 DOS 的顶部保存了另外的数据。该扩展 BIOS 数据区最常见的用处是为主板的鼠标端口、硬盘参数以及磁道缓冲区保存数据。

扩展 BIOS 数据区的段址通常保存在地址为 40:0Eh 的字中。由于该字最初用来保存打印口 4 的端口号，所以并不是所有的 BIOS 都支持这种半公开的标准。在 640K 的系统上，打印机通常被设置为 9EC0h，代表有 1K 的扩展 BIOS 数据区。虽然这很少，但是有些系统却为扩展 BIOS 数据区保留了 2K 甚至 4K 的空间。

快速判断扩展数据区是否被激活的方法是用 DOS 的 MEM 命令。对于一个不带内存管理器的 640K 系统来说，如果没有扩展 BIOS 数据区，系统通常会返回 640K 或 655360 字节的常规内存。但是如果使用了扩展 BIOS 数据区，该数字要小一些。

40:13h 处的主存大小字会减去扩展 BIOS 数据区的大小。该字通常设置为 280h，代表 640K 的主存。如果该值有点小，比如 27Fh，那么在显示区域下的 1K 空间可能用作了扩展 BIOS 数据区。某些系统，比如一些基于 SCSI 的老式 HP 系统，使用 4K 来保存硬盘信息。

显示区域下的空间唯一合法的用处就是用作扩展 BIOS 数据区。记住，某些病毒程序也使用这种技术来隐藏同一区域的病毒代码，大多数病毒是 2K 到 4K 之间大小。

因为这些信息常常未加以说明，所以不清楚大多数供应商是否遵循 IBM 惯例。Phoenix 似乎遵循 IBM 惯例，它的系统包含有鼠标支持以及一个看门狗（Watchdog）时钟。不同的型号不一定支持这里所讨论的功能。将来的 BIOS 版本可能会改变上述信息，但是某些项，比如鼠标，保留了至少六年而未变。

扩展 BIOS 数据用法归纳

下表指出了扩展 BIOS 数据区的通用用法。这些数据涉及到鼠标、看门狗时钟、键盘、软盘以及硬盘。段址可变，但是在 640K 主存的系统上段址通常是 9FC0h。来自 40:0Eh 的段址在这里表示为“EBDA”。

地址	功能	大小
EBDA:0	扩展 BIOS 数据区的大小	字节
EBDA:17h	POST 错误入口的数目	字节

EBDA:18h	错误日志	5个字
EBDA:22h	鼠标设备驱动程序远调用	双字
EBDA:26h	鼠标标志1	字节
EBDA:27h	鼠标标志2	字节
EBDA:28~2Fh	鼠标数据	8个字节
EBDA:39h	看门狗时钟	双字
EBDA:3D~4Ch	硬盘0的参数表	16个字节
EBDA:4D~5Ch	硬盘1的参数表	16个字节
EBDA:68h	高速缓存控制	字节
EBDA:6Eh	键盘的重复速率	字节
EBDA:6Fh	键盘重复的延迟	字节
EBDA:70h	附带的硬盘数目	字节
EBDA:71h	硬盘驱动器的DMA通道	字节
EBDA:72h	硬盘驱动器中断状态	字节
EBDA:73h	硬盘操作标志	字节
EBDA:74h	老式的中断76h向量指针	双字
EBDA:78h	硬盘DMA类型	字节
EBDA:79h	硬盘，上次操作的状态	节
EBDA:7Ah	硬盘，超时值	字节
EBDA:7E~8Dh	硬盘，控制器返回状态字	8个字
EBDA:E7h	软盘驱动器类型	字节
EBDA:ECh	装入的硬盘参数表	字节
EBDA:EEh	CPU家族系列ID	字节
EBDA:EFh	CPU级别号	字节
EBDA:117h	键盘ID	字
EBDA:11Ah	非BIOS中断18h标志	字节
EBDA:11Dh	用户中断18h远指针	双字

扩展BIOS数据区的详细说明

地址	描述	大小
EBDA:0	扩展BIOS数据区的大小	字节

这个字节指明了扩展BIOS数据区的1024字节块的数目。通常EBDA的大小是1024字节，所以该值是1。

地址	描述	大小
EBDA:17h	POST错误入口的数目	字节

大多数的 PS/2 系统在发生错误时，会将 POST 代码保存到错误日志中。这个字节的值就表示了保存在错误日志中错误的数目。出现了第 5 个错误后，不再保存其他的错误。

地址	描述	大小
EBDA:18h	错误日志	5 个字

当 POST 检测到错误时，POST 代码就被保存到该错误日志中，一个字一个错误。EBDA:17h 用作该日志的索引。当检测到新错误时，就把它放入一个未使用的日志字中。该日志最多保存 5 个错误。

地址	描述	大小
EBDA:22h	鼠标设备驱动程序远调用	双字

该双字作为一个指针指向鼠标设备驱动程序代码。如果发生了鼠标事件，数据就从堆栈上传送到设备驱动程序，参看第 13 章，中断 15h，功能 C2h，子功能 7 获得更多细节。

地址	描述	大小
EBDA:26h	鼠标标志 1	字节

这个字节保存了许多标志，这些标志涉及到主板鼠标端口控制器。

位		
	7=1	命令正在执行中
	6=1	鼠标发送了一个“重新发送”字节，FAh
	5=1	鼠标发送了一个“确认”字节，FEh
	4=1	鼠标发送了一个错误字节， FCh
	3=1	收到了未期望的值
	2=x	索引 0~7——用于从控制器中回收 1 到 8 个字节，每次触发中断 74h（鼠标中断）时减 1。每个接下来的字节保存在开始于 28h 的 EBDA 中
	1=x	
	0=x	

地址	描述	大小
EBDA:27h	鼠标标志 2	字节

这个字节保存了许多标志，这些标志涉及到主板鼠标端口的操作。

位		
	7=x	设备驱动程序的远调用标志
	6=x	未使用或未知功能
	5=x	未使用或未知功能
	4=x	未使用或未知功能
	3=x	未使用或未知功能

2=x 1=x 0=x	数据包计数值（1~8 个字节）。这是鼠标收到的字节的数目	

地址	描述	大小
EBDA:28~2Fh	鼠标数据	8 个字节

这 8 个字节用于保存来自鼠标/键盘控制器的数据。字节数由数据包计数，BEDA:27h 指示。如果所有的这些字节已经装入了本区域，来自本区域的数据就会推入栈顶，并对鼠标设备驱动程序进行远调用。

地址	描述	大小
EBDA:39h	看门狗时钟	字

这个字节保存了看门狗时钟初始计数值。值 0 表示看门狗时钟未被激活。

地址	描述	大小
EBDA:3D~4Ch	硬盘 0 的参数表	16 个字节

使用 IDE 驱动器时，硬盘 0 的参数表就保存在这 16 个字节中，参看第 11 章获取硬盘参数表的结构信息。

地址	描述	大小
EBDA:4D~5Ch	硬盘 1 的参数表	16 个字节

使用 IDB 驱动器时，硬盘 1 的参数表就保存在这 16 个字节中。参看第 11 章以获取硬盘参数表的结构信息。

地址	描述	大小
EBDA:68h	高速缓存控制	字

这个字节用于为那些带有高速缓存的 CPU 控制 CPU 高速缓存。这些 CPU 包括所有的 486 CPU 和 IBM 的 386SLC CPU。

位	7=0	未使用
	6=0	未使用
	5=0	未使用
	4=0	未使用
	3=0	未使用

2=0	未使用
1=0	高速缓存测试成功（这一位可能还有其他功能）
1	高速缓存测试失败，没有使用高速缓存
0=0	CPU 高速缓存开放
1	CPU 高速缓存被禁止

地址	描述	大小
EBDA:6Eh	键盘重复的速度	字节

送到键盘的上次重复速率保存在这里。保存在这里的值和中断 16h，功能 3 使用的值相同。参看第 8 章获得其他细节。

地址	描述	大小
EBDA:6Fh	延迟直到键盘重复	字节

上次送到键盘的重复速率保存在这里。这是从按下键到自动重复开始时等待的时间。保存在这里的值和中断 16h，功能 3 使用的值相同。参看第 8 章获得其他细节。

地址	描述	大小
EBDA:70h	附带的硬盘数目	字节

POST 检测到的附带的硬盘数目，0 到 2。

地址	描述	大小
EBDA:71h	硬盘驱动器的 DMA 通道	字节

为硬盘系统定义了 16 位的 DMA 通道。缺省时 POST 使用 DMA 通道 5。

地址	描述	大小
EBDA:72h	硬盘中断状态	字节

硬盘操作完成时，来自硬盘控制器的中断状态保存在这个字节中。如果在等待控制器响应或等待中断结束时出现等待超时，就会将这个字节设置为 1Fh。

地址	描述	大小
EBDA:73h	硬盘操作标志	字节

如果硬盘控制器触发了中断，那么就会设置第 7 位。这一位由中断 76h 操作完成程序设置。如果重启了控制器那么就会设置第 6 位，而一旦开始了控制器的 DMA 操作就会清除这一位。其他各位没有使用。

位	7=1	硬盘控制器触发了一个操作完成中断
	6=1	重启了控制器
	5=0	未使用
	4=0	未使用
	3=0	未使用
	2=0	未使用
	1=0	未使用
	1=0	未使用

地址	描述	大小
EBDA:74	老式中断76h的向量指针	双字

在用指向操作完成程序的指针替换向量之前，硬盘的首次初始化代码在该地址保存老式的中断76h向量。如果是由某人，而不是硬盘控制器触发了操作完成子程序，中断76h，那么BIOS就会将控制传给使用这个字指针的最初的中断76h程序。

地址	描述	大小
EBDA:78h	硬盘DMA类型	双字

这个字节为DMA扩展模式寄存器提供临时的存储空间。DMA扩展模式寄存器在第18章中由DMA服务，端口18h，命令7xh定义。

中断13h的BIOS处理程序装入一个值用来表示即将执行16位的DMA传送、读/写选择以及其他信息。一旦硬盘控制器指出DMA安装好了，该值就会被装入到DMA扩展模式寄存器中，通常值44h用作读，而4Ch用作写。

地址	描述	大小
EBDA:79h	硬盘，上次操作状态	字节

在这个字节中保存了上次硬盘操作的状态。它可由中断13h的功能（获得上次硬盘操作的状态）返回。

地址	描述	大小
EBDA:7Ah	硬盘，定时值	字节

POST设置该值用来表示硬盘服务BIOS应等待多长时间，让控制器发出操作完成的信息。如果控制器在定时周期内没有中断系统。BIOS服务会跳出当前的操作并返回一个超时错误。

地址	描述	大小
EBDA:7E~8D	硬盘控制器的状态字	8个字

读自硬盘控制器的一组字，例如状态块以及其他的数据，都被保存在这个地址中。读自控制器并保存在此处的字的数目取决于控制器的操作，但总是 8 个字或少于 8 个。参看第 11 章，硬盘系统中 PS/2 的 ESID 端口 3510h，以获取返回状态字的有关细节。

地址	描述	大小
EBDA:E7h	软驱类型	字节

这个字节表示是否带有软驱以及类型是 3.5"还是 5.25"的。软盘 BIOS 用它来确定软驱的合适参数。

位	7=0	有软驱
	6=0	未使用
	5=0	未使用
	4=0	未使用
	3=0	未使用
	2=0	未使用
	1=0	软驱 1 是 3.5"的
	1	软驱 1 是 5.25"的
	0=0	软驱 0 是 3.5"的
	1	软驱 0 是 5.25"的

地址	描述	大小
EBDA:ECh	装入了硬盘参数	字节

这个字节表示硬盘 BIOS 是否向扩展 BIOS 数据区（开始于 EBDA:3D）中装入了 IDE 驱动器参数

位	7=1	装入了硬盘参数
	6~0=0	未使用

地址	描述	大小
EBDA:EEh	CPU 家族系列 ID	字节

重启 CPU 时，家族系列的 ID 被放入了 DH 寄存器中。例如一个 486 的 ID 是 4。BIOS 将这个值保存在本地址中，它由中断 15h，AX=C9h 返回。

地址	描述	大小
EBDA:EFh	CPU 级别	字节

重启 CPU 时，会在 DL 寄存器中被放入 CPU 的级别信息。BIOS 将这个值保存在本地

址中，它可由中断 15h，AX=C9h 返回。

地址	描述	大小
EBDA:117h	键盘 ID	字节

该字保存了键盘 ID。今天在美国常见的 IBM 键盘的 ID 值是 41ABh。

地址	描述	大小
EBDA:11Ah	用 BIOS 中断 18h 志标	字

如果一个程序或 TSR 使用中断 18h（BASIC 中断），那么 BIOS 可能会再次挂起中断，并保存位于 BEDA:11Dh 处的用户指针。在调用用户向量之前，这个字节的第 0 位会被设置为 1，该指针的目的还不清楚，但是它可能与某些系统的屏幕设置有关。

位		
	7~1=0	未使用
	0=1	在调用用户中断 18h 之前

地址	描述	大小
EBDA:11D	用户中断 18h 的远指针	双字

如果一个程序或 TSR 使用中断 18h（BASIC 中断），那么 BIOS 可能会再次挂起中断，并保存位于该双字中的指针。该指针的目的还不清楚，它可能与某些系统的屏幕设置相关。

显示器内存

系统为显示器缓冲区保留了 128K 空间。在某些情况下，并未使用这个缓冲区的某些空间，具体情况取决于所使用的适配卡，许多高级的内存管理程序可以重新使用显示器区域中的这些空余空间，并将之作为上端内存块（UMB）——参看表 6-7。UMB 可以容纳 TSR 和驱动程序，以增加应用程序可用的低端 DOS 空间。

表 6-7 视频显示缓冲区

地　址	大　小	功　能
A000:00h	64K	EGA/VGA/XGA/XVGA 图形显示区和文本字节保存区。在 80x600 以上的解决方案中，某些视频模式和视频芯片操作会用完 128K 的地址空间。
B000:00h	4K	MDA 显示区，仅适用于文本，EGA+模拟支持多达八页文本，总共 32K 的地址空间，当未使用视频模式 7 时，该区域通常用作额外的上端内存区域。
B000:00h	64K	Hrecules 图形模式，两页。
B8000:00h	32K	CGA/EGA+彩色文本模式，八页。

适配器 ROM 和 UMB 内存

显示缓冲区以上的区域最初定义为适配卡 ROM 的固件区域。现在，硬件和软件供应商将它定义为容易访问而未说明的内存区域。尽管许多标准使每一件事情都能够和平共存，但是许多供应商仍然违背了这些标准，将高端内存搞得一团糟。参看第 5 章以获有关这个问题更多的信息。

第7章

中断向量表

本章的大部分将讨论直接与硬件有关的中断，过去通常没有对它们做出说明或者说明得很不够。本章详细介绍了与特定的 BIOS 子系统相关的 BIOS 中断功能。其他已经做过很好的说明的中断在这里只是简要的加以归纳。这里没有深入说明的中断向量描述请参考其他出色的文档信息。

中断向量通常保存在内存的前 1024 个字节中，并且含有 256 个双字中断指针。它们可分为 CPU 专用中断、BIOS 中断、通用适配卡中断、操作系统中断以及用户程序中断。从 286 CPU 开始，中断向量表可以在保护模式下切换到任何位置。这一点对于那些高级的操作系统和保护模式软件非常有用。

在 BIOS 所支持的功能和许多 CPU 异常之间常常存在着冲突。Intel 特意保留了中断 0 到 1Fh 以供内部 CPU 使用。在 8088 上实际只用到了前四个中断，所以 IBM 决定将其他中断用作 BIOS 功能。这是一个非常武断的决定，因为 IBM 可以选择剩下的 224 个中断中的任何一个。IBM 在系统设计方面实际上又犯了一个大错。一旦 PC 成为主要的标准之后，要想改变 BIOS 的中断分配就已经太晚了。幸运的是，Intel 后来加入到 CPU 中的 BIOS 功能并不和 BIOS/DOS 下的实模式 PC 环境发生冲突，一个可能发生 BIOS 和 CPU 中断重叠的地址是中断 0Ch。Intel 386 及后来的 CPU 上将 CPU 堆栈错误异常定义为中断 0Ch。同样的中断也经常用于串行口。如果系统工作在虚 86 或保护模式下，保护模式处理程序就会确定这个中断是由堆栈错误引起还是由串行 UART 引起。然后处理程序会指向堆栈错误处理程序代码或者跳转到串行口代码。

在实模式下，往往忽略了 BIOS 和 CPU 之间的这种冲突。在前面所举的例子中，都假定了从来都不会发生堆栈错误。当然，我们知道软件可以完美到不产生一个错误！如果一个不可能的事件还是发生了，并且发生了堆栈错误，那么系统可能会发生冲突。既然你不能为这做点什么，那么 BIOS 就只是简单地忽略它，让系统继续执行下去。

中断 20h 到 3Fh 是为 DOS 而保留的。大多数从 20h 到 2Fh 的中断都是供 DOS 使用的，但是 DOS 不使用从 30h 到 3F 的中断。相反，还发展了许多标准来使用从 30h 到 3Fh 的中断。表 7-1 归纳了大多数系统和软件是如何使用中断的。

某些中断向量还用于数据存储。如果是这样，这些特殊的向量就不能挪作它用。这些数据存储的入口显示为“数据”。

表 7-1　中断向量表归纳

中　断	地　址	类　型	功　能
0	0:00h	CPU	除法错误
1	0:04h	CPU	单步
2	0:08h	CPU	不可屏蔽中断
3	0:0Ch	CPU	断点指令
4	0:10h	CPU	溢出指令
5	0:14h	BIOS	打印屏幕
		CPU，286+	越界
6	0:18h	CPU，286+	无效操作码
7	0:1Ch	CPU，286+	没有协处理器
8	0:20H	硬件	IRQ0——系统时钟
		CPU，286	中断越界导致异常
		CPU，386+	双重异常
9	0:24h	硬件	IRQ1——键盘
		CPU，286+	协处理器段过载
A	0:28h	硬件	IRQ2——通用适配器的使用/级联
		CPU，386+	无效的 TSS
B	0:2Ch	硬件	IRQ3——串行口
		CPU，386+	段不存在
C	0:30h	硬件	IRQ4——串行口
		CPU，386+	堆栈异常
D	0:34h	硬件	IRQ5——通用适配器的使用
		CPU，286	段重叠异常
		CPU，386+	一般保护性错误
E	0:38h	硬件	IRQ6——软盘控制器
		CPU，386+	页面错误
F	0:3Ch	硬件	IRQ7——打印机 1
10	0:40h	BIOS	视频
		CPU，286+	协处理器错误（未使用）
11	0:44h	BIOS	设备配置
		CPU，486+	对齐校验
12	0:48h	BIOS	内存大小
		CPU，Pentium+	机器检查

续表

中 断	地 址	类 型	功 能
13	0:4Ch	BIOS	硬盘/软盘服务
14	0:50h	BIOS	串行通信
15	0:54h	BIOS	系统服务
16	0:58h	BIOS	键盘服务
17	0:5Ch	BIOS	打印机服务
18	0:60h	BIOS	ROM BASIC/引导错误
19	0:64h	BIOS	引导自举
1A	0:68h	BIOS	日期
1B	0:6Ch	BIOS	键盘 Control-Break
1C	0:70h	BIOS	用户时钟滴答
1D	0:74h	BIOS 数据	视频初始化数据指针
1E	0:78h	BIOS 数据	软盘配置指针
1F	0:7Ch	BIOS 数据	图形字符集体指针
20	0:80h	DOS	终止程序
21	0:84h	DOS	一般服务
22	0:88h	DOS 数据	终止地址
23	0:8Ch	DOS 数据	Control-C 处理程序地址
24	0:90h	DOS 数据	严重错误处理程序的地址
25	0:94h	DOS	绝对磁盘读
26	0:98h	DOS	绝对磁盘写
27	0:9Ch	DOS	终端驻留（过时）
28	0:A0h	DOS	DOS 空闲
29	0:A4h	DOS	显示字符
2A	0:A8h	DOS	NETBIOS
2B	0:ACh	DOS	未使用
2C	0:B0h	DOS	未使用
2D	0:B4h	DOS	未使用
2E	0:B8h	DOS	DOS 命令执行
2F	0:BCh	DOS	多路复用
30	0:C0h	DOS 数据	跳转到 DOS 的 CP/M 服务（5 字节）
31	0:C4h	用户	DPMI（在保护模式下）
32	0:C8h	用户	未使用

续表

中　断	地　址	类　型	功　能
33	0:CCh	用户	鼠标
34	0:D0h	用户	浮点，模拟指令 D8h
35	0:D4h	用户	浮点，模拟指令 D9h
36	0:D8h	用户	浮点，模拟指令 DAh
37	0:DCh	用户	浮点，模拟指令 DBh
38	0:E0h	用户	浮点，模拟指令 DCh
39	0:E4h	用户	浮点，模拟指令 DDh
3A	0:E8h	用户	浮点，模拟指令 DEh
3B	0:ECh	用户	浮点，模拟指令 DFh
3C	0:F0h	用户	浮点模拟
3D	0:F4h	用户	浮点模拟
3F	0:FCh	用户	覆盖管理/其他
40	0:100h	BIOS，AT+	重新指向软盘服务
41	0:104h	BIOS 数据	硬盘 0 的配置指针
42	0:108h	EGA+	老式的视频中断向量
43	0:10Ch	EGA+数据	全图形字符指针
46	0:118h	BIOS 数据	硬盘 1 的配置指针
4A	0:128h	BIOS，AT+	出现警报
4B	0:12Ch	用户	虚拟 DMA 服务
67	0:19Ch	用户	EMS 内存
70	0:1C0h	硬件，AT+	IRQ8——CMOS 实时时钟
71	0:1C4h	硬件，AT+	IRQ9—通用适配器使用
72	0:1C8h	硬件，AT+	IRQ10—通用适配器使用
73	0:1CCh	硬件，AT+	IRQ11——通用适配器使用
74	0:1D0h	硬件，AT+	IRQ12—鼠标端口
75	0:1D4h	硬件，AT+	IRQ13—数学协处理器
76	0:1D8h	硬件，AT+	IRQ14—硬盘主盘控制器
77	0:1DCh	硬件，AT+	IRQ15—通用适配器使用
C0~C3	0:300h	BIOS 数据	用户定义的硬盘 0 数据
C4~C7	0:310h	BIOS 数据	用户定义的硬盘 1 数据

中断向量表与数据描述

中断	描述	类型
0	除法错误	8088+

在试图执行除法指令时，发生了不合适的操作。如果指令的除数是0，就会触发中断0。由于被0除会产生一个无穷大的值，所以从来都不允许这样。程序中常常会出现以0为除数的bug。发生溢出时也会触发中断0。例如，AX=FFFFh，BL=1，指令DIV BL（意味着AL=AX/BX）会得到商FFFFh，它就不能放入结果寄存器AL中。

如果应用程序没有提供除法错误的处理程序，大多数操作系统将终止应用程序，并返回操作系统。当然并不总是这样，也有可能挂起系统，具体情况如何取决于造成除法溢出的程序的属性。

返回的地址指示了问题发生的地点，这个地址在各种CPU中会有所不同。在8088、8086、80188和80186 CPU上，返回的是造成除法错误的指令之后的远地址指令错误的指令。如果该指令带有前缀，比如EX，那么指向的是造成除法错误的指令的第一个前缀。

中断	描述	类型
2	单步	所有CPU

当CPU单步调试时，在每次执行一条指令后就调用这个中断。然而，通用的中断指令却是一个例外。当正在执行INT指令（比如INT Bh）时，所有这个中断的代码将被执行完。在这个中断返回后，CPU才触发中断1，就好像执行单独的一条指令一样。

80386以及后来的CPU也将此中断用作其他的调试功能。这些特殊的调试功能包括在下述情况下中断：发生了指令或地址断点、发生了任务切换断点、或者发生了内电路模拟器（ICE）冲突错误，可以使用调试寄存器DR6和DR7中的位来确定发生了上面哪种情况，如表7-2所示。表7-3解释了一系列相关的标志位。

表7-2 确定调试中断

测试条件	为什么发生中断
BS=1	单步
B0=1且（GE0=1或LE0=1）	DR0发生断点
B1=1且（GE1=1或LE1=1）	DR1发生断点
B2=1且（GE2=1或LE0=1）	DR2发生断点
B3=1且（GE3=1或LE1=1）	DR3发生断点
BD=1	由于正在使用ICE，所以调试寄存器不可用
BT=1	任务切换断点

表 7-3 调试标志

标　志	描　述	寄　存　器	位
B0	调试寄存器 0 断点	DR6	0
B1	调试寄存器 1 断点	DR6	1
B2	调试寄存器 2 断点	DR6	2
B3	调试寄存器 3 断点	DR6	3
BD	与 ICE（内电路模拟器）冲突	DR6	13
BS	单步断点	DR6	14
BT	任务断点	DR6	15
GE0	调试寄存器 0 全局开放	DR7	1
GE1	调试寄存器 1 全局开放	DR7	3
GR2	调试寄存器 2 全局开放	DR7	5
GR3	调试寄存器 3 全局开放	DR7	7
LE0	调试寄存器 0 局部开放	DR7	0
LE1	调试寄存器 1 局部开放	DR7	2
LE2	调试寄存器 2 局部开放	DR7	4
LE3	调试寄存器 3 局部开放	DR7	6

中断	描述	类型
2	不可屏蔽的中断	所有 CPU

CPU 有一条独立的硬件输入线不能被 CPU 关闭。它用于那些重要的错误，比如 DRAM 奇偶错误和协处理器错误。所有的 PC 系统都提供了“屏蔽”CPU 外部硬件 NMI 功能的能力。这主要是通过端口 70h 实现控制的。参看第 17 章获得其他细节。

PC 环境下这种常见的用法包括：检测 RAM 奇偶错误、数学协处理器错误、看门狗超时、掉电（来自 CPU 硬件）以及在某些硬件调试器上的中止操作。

中断	描述	类型
3	断点指令	所有 CPU

CPU 提供了一个特殊的单字节来触发中断 3。较老的调试器（如 DEBUG）使用它来在代码中插入一个断点。在断点位置，调试器保存当前指令字节，并且使用中断 3 指令 CCh 进行替换。执行 CCh 指令时，会发生中断 3，并返回调试器。

在某些情况下，应用程序使用中断 3 提供重要的中断功能，其目的是为了使程序难以被分析。这种技术主要用于拷贝保护。通过使用这个中断，当使用断点的调试器需要中断 3 时，程序就不能正确地发挥作用。为了使调试器的断点失败，程序将中断 3 的向量指向其自己的代码。与普通的中断挂起不同的是，程序的中断 3 处理程序并不将控制传递给原来的中断 3 处理程序，调试器。如果调试器将一个断点插入了代码中，并执行到了断点，中断并不返回到调试器，而是进入应用程序的处理程序。这样应用程序就可以终止或执行其他别的动作。当然，所有的 386+调试器都利用硬件断点的调试寄存器来避免这个问题。

中断	描述	类型
4	溢出指令	所有 CPU

如果设置了溢出标志位，那么一个特殊的单字节中断溢出指令会产生中断 4。我猜测最初是想通过一个中断处理程序来处理数学错误。在所有的商业程序中，我还未见过有人使用这条指令。

中断	描述	类型
5	打印屏幕	BIOS
	越界	80188+

这是下面一系列中断中的第一个，IBM 在这些中断中定义 BIOS 功能时和 Intel 在 80186 及以后的 CPU 上定义的异常发生了冲突。当按下屏幕打印（Print-Screen）键时，低级键盘处理程序中断 9 会调用中断 5。系统 BIOS 为中断 5 提供的处理程序会扫描当前的文本屏幕，将之输出到打印机。大多数高级视频 BIOS，比如 VGA，都提供了软件控制的选项，来用自己的 BIOS 程序替换系统的 BIOS 程序。这允许 Print-Screen 功能可以从一个比 80 行还宽，比 25 列还大的屏幕上打印文本。参看第 14 章以获得更多细节。

如果 CPU 检测到数组下标越界，CPU 会在 BOUND 指令的异常基础上触发中断 5。8088/8086 处理器不支持 BOUND 指令，但是所有后来的 CPU 都支持它。由于它可能和 BIOS 的屏幕打印功能相冲突，所以很少使用 BOUND 指令。如果它用于 BOUND 异常处理，一个远返回会重新执行失败的 BOUND 指令。

中断	描述	类型
6	无效操作码	80188+

执行无效的操作码时会触发这个中断。例如，在 80186 及以后的 CPU 上从未分配过指令字节 0Fh、17Fh。如果执行 0Fh、17Fh，CPU 就会触发中断 6。某些指令只允许以寄存器或内存地址作为操作数，但是两者不能同时使用。如果使用了错误的类型，这会产生一个无效指令异常。会产生这种异常的指令包括 BOUND、LDS、LES、LFS、LGS、LSS、LGDT

以及 LIDT 指令。

8088/8086 处理器不支持无效操作码异常，但是 80186/80188 及以后的 CPU 都支持它。先进的调试器利用它来帮助识别程序中的 bug，但是它不会出现在普通的操作中。软盘上提供的程序 CPUUNDOS 就是使用该无效操作码中断一寻找可能的内幕指令。

中断	描述	类型
7	未提供协处理器	80286+

对于 80286 或后来的 CPU，如果执行了 ESCAPE 指令操作码 D8h 到 DFh，或者执行了 WAIT 指令，但是没有连接数学协处理器，CPU 就会触发中断 7。如果系统不带数学协处理器，那么某些编译器会用它来模拟数学协处理指令。今天大多数的编译器提供了快速的数学协处理器模拟而不使用中断 7。参看开始于 34h 的一组中断来了解数学协处理器模拟的可任选的一种技术。

机器状态字中的 80286 状态位决定是否使用这个中断在软件中模拟数学指令，在 80386 以及后来的处理器上如果设置了 CPU 的模拟位，那么一条 ESACPE 指令会触发中断 7。只要设置了 CR0 的任务切换或者显示器处理器扩展位，ESCAPE 和 WAIT 指令就都会触发中断 7。

中断	描述	类型
8	IRQ0 一系统时钟	硬件
	双重错误	80286+

中断 8 用作系统计时，它与来自系统时钟 0 的 IRQ0 实现硬件上的连接。每 54.9ms 时钟 0 触发一次中断 8。参看第 16 章以获得更多细节。

在 80286 及以后的 CPU 上，在发生双重错误时会触发中断 8。双重错误指的是，在处理一个 CPU 错误时，处理程序又产生了第二个错误。例如，试图执行无效指令。这会造成触发坏操作码中断 6。如果与此同时堆栈指针为 FFFFh，CPU 将试图将坏操作码地址推入栈顶，这时还会产生堆栈错误。这就是造成调用双重错误中断 8。在实模式下，堆栈方面仍存在问题，所以进入中断 8 时产生另一个堆栈错误（第三个错误）。由于 CPU 不能处理这么严重的错误，所以它会停止运行。

既然双重错误很少见也很难恢复原样，因此 BIOS 和 DOS 都忽略了双重错误的处理。

并不是所有的错误都会产生双重错误。如果在发生下表 7-4 中所列的任何一个错误的同时又发生了这个表中的一个错误，那么就会触发双重错误。这条规则也有一个例外。如果正在处理该列表中的某个错误（页面错误除外），而同时发生了一个页面错误，就不会产生双重错误。

表 7-4 双重错误类型

中 断	错 误 描 述
0	除法错误
A	无效的任务状态段（TTS）
B	段不存在
C	堆栈错误
D	一般保护性错误（GPE）
E	页面错误

中断	描述	类型
9	键盘协处理器——段超载	硬件 286/386

中断 9 用作低级键盘处理程序。它在硬件上和来自键盘控制器的 IRQ1 相连接。参看第 8 章以获取更多细节。

如果 80286 或 80386 工作在保护模式下，并且向协处理器传送协处理器指令时发生了页面或段违规，那么就会触发中断 9。由于键盘处理程序从未工作在保护模式下，所以不会发生冲突。在 80486 以及后来的 CPU 上，协处理器段过载中断已被移到了中断 0Dh。

中断	描述	类型
A	IRQ2——无效的 TSS	硬件 80286+

硬件 IRQ2 触发中断 A。某些适配卡将它作为保留中断。在带有两个中断控制器的 AT+系统上，IRQ2 用作控制器的串级功能。IRQ9 和老式的 IRQ2 接相同的硬件线。所以若发生了 IRQ9，BIOS 必须采取一些措施。这保持了同老式卡的兼容。参看第 17 章中的中断 71h，以获取更多细节。IRQ2 通常用于网卡、某些 MIDI 接口中，每次发生垂直跟踪时也可用于某些 EGA/VGA 卡中。

对于 80286 及以后的 CPU，如果在任务切换过程中新的任务状态选择器（TSS）无效，就会发生中断 A。这只在保护模式下的软件中发生，表示软件中有严重的 bug。

中断	描述	类型
B	串行口	硬件
	段不存在	80286+

由硬件 IRQ3 触发中断 B。某些适配卡将它作为保留中断，并且通常由串行口 2 和 4 以及某些网卡使用它。参看第 12 章以获取有关串行口如何使用这个中断的更多信息。

对于 80286 和后来的 CPU，当处理器发现一个装入的段不存在时，在保护模式下会发生中断 B。保护模式操作系统用它来执行一个虚拟内存系统上的段。例如，OS/2 版本 1.x

就使用这种方法实现虚拟内存。使用它不会和 IRQ3 发生冲突，因为保护模式处理程序能够检测出这个中断是来自硬件还是 CPU。参看第 7 章，以了解如何确定一个中断是由硬件引起还是由 CPU 引起。参看 Intel 有关 80286 及以后的 CPU 的程序员参考手册，以获取“段不存在”异常的更多信息。

中断	描述	类型
C	IRQ4	硬件
	堆栈异常	80286+

硬件 IRQ4 触发中断 C。适配卡使用这种非专用 80286+中断，通常情况下是串行口 1 和 3 使用它。想知道更多的有关串行口如何使用这个中断，参看第 12 章相关内容。

对于 80286 和后来的 CPU，在改变 SS 寄存器或堆栈下溢/上溢时如果越栈，就会触发中断 C。它不会与 IRQ4 冲突，因为保护模式的处理程序可以检测出是硬件还是 CPU 引起这个中断。参看第 17 章，以了解如何确定一个中断是硬件还是由 CPU 引起，参看 Intel 有关 80286 及以后的 CPU 的程序员手册，以获取堆栈异常的更多信息。

中断	描述	类型
D	IRQ5	硬件
	段保护性错误	80286+

硬件 IRQ5 触发中断 D。这是一个供适配卡使用的非专用中断。在 PC 和 XT 上，中断 D 供硬盘适配器使用。参看第 11 章以获取更多有关硬盘使用的信息。在 AT+系统上，中断 D 通常供串行口 2 或网卡使用。

对于 80286 及以后的 CPU，一旦 CPU 检测到严重的错误（不能归入其他类错误中），就会触发中断 D。保护模式处理程序将确定是发生了错误还是 IRQ5 需要提供服务。参看第 17 章，以了解如何确定中断是由硬件还是由 CPU 引起。尽管一个一般性保护错误通常表示一个严重的 bug 或问题，但这里只有两个一般性保护错误更常见。它们是特权指令造成的错误或因执行/数据访问越过段界所造的错误。

如果在当前的优先级下不允许执行某特权指令，就会发生这种错误，不幸的是，Intel 为某些必须的指令提供了太多的保护性措施。例如，读 CR0 被认为是一种特权，在虚 86 模式下会产生错误（MOV EAX，CR0）。如果使用了内存管理程序，而程序试图读取 CR0，CPU 就会通过这个中断传递控制。较好的管理程序会模拟这个有用的指令，并将 CR0 的内容放入 EAX 中。

段越界错误可分为两类，代码和数据。如果指令指针返转就会发生代码错误，例如在 16 位模式下，在偏移量为 FFFFh 处执行一条非跳转指令。如果程序试图越地段的边界读或写，就会发生数据错误。例如，在 16 位模式从偏移量为 FFFDh 处读取双字。

尽管 CPU 异常与 IRQ5 的使用不发生冲突，但是在虚 86 模式下它也会减慢处理 IRQ5 的响应速度。参看第 17 章以更详细地了解有关一般保护性错误和 IRQ5 的执行情况。参看

第17章，以了解如何确定是硬件还是CPU引起了中断。

中断	描述	类型
E	IRQ6——软盘控制器更正错误	硬件 80386+

在当前操作结束后，软盘控制器会触发中断E。中断E处理程序将40:3Eh处的软件重校状态字节的第7位设置为1。在AT系统上，同时还触发中断15h功能AX=9101h表明控制器操作完成，参看第10章以获取其他细节。

在80386及以后的CPU上，如果允许分页但系统又不能访问指定的页面时，会触发中断E。386及以后的CPU用分页来将物理内存映射到不同的逻辑地址。所有的内存管理程序都使用这种重新映射技术来在显卡以上的区域中创建高端内存。在保护模式的代码中，中断E可能用来表示出错，或者在许多情况下，它用于保护、动态映射、虚拟内存以及其他功能。

实模式下永远也不会发生CPU触发的中断。在虚86模式下，保护模式处理程序必须确定是页面错误还是硬件引起了中断，并采取适当的行动。参看第17章以了解如何确定是硬件还是CPU引起了中断。

中断	描述	类型
F	IRQ7 打印机1	硬件

当来自打印机的确认线被确定之后（ACK=0）。每个打印口都可以触发一个中断。打印机控制器可以禁止这种能力。如果开放了打印机1的适配卡，打印机确认线就会调用中断F。BIOS中断17h处理程序不使用这个中断，缺省情况下打印机中断是被禁止的。参看第14章以获得其他细节。

当发生未知硬件中断的情况下，中断控制器也会使用这个中断。如果硬件中断请求过于简短，或者中断请求线上出现了噪声，就可能会出现上述情况。系统BIOS不支持这种情况，通常予以忽略。

中断	描述	类型
10h	硬件协处理器错误（表明）	BIOS 80386+

软件触发这个中断来访问视频功能。参看第9章以获取其他细节。

如果你看看任何一个80386或后来的Intel CPU手册，它们都为协处理器错误保留了中断10h的CPU功能。如果如Intel所建议的那样的连接了数学协处理器，那么是这样的。在AT+系统上，协处理器错误连接到IRQ13，中断75h。参看中断75获得其他细节。

80386和后来的CPU有一条ERROR输入线。Intel特意用它来连接ERROR输出线和数学协处理器。甚至在486DX、Pentium以及Pentium Pro上，也有一条ERROR输入线，尽管它们有内置的数学协处理器。如果激活了CPU的ERROR线，就会触发中断10h在所

有的正确设计的 AT+系统上，并没有使用 CPU 的 ERROR 输入，并且永远也不会产生中断 10h。当然，某些复制板可能会有所不同，它们同 Intel 建议的方法一样，而不是实用的方法，只要不安装数学协处理器，即使是这些系统也可以正常的工作。

中断	描述	类型
11h	设备配置对齐检测	BIOS 80486+

这个中断返回 POST 所确定的设备配置，它仅仅只是当地址 40:10h 处的设备字的内容返回到 AX。参看地址 40:10h 处描述的设备字的位的内容（第 6 章）。

在 80486 和后来的 CPU 上，提供了一个选项来陷入没有对齐的内存地址。EFLAGS 中的对齐检查位可以允许该选项。如果允许了该选项，CPU 位于优先级模式 3 下，并且内存参考指向了没有对齐的地址，那么就会触到这个中断 11h。AT+系统禁止了检查错误以避免和使用中断 11h 的设备配置相冲突。

一个没有对齐的地址指的是不可以被其大小所平分的数据参考地址。例如，一个四字节的 DWORD 数据项可以在偏移量 0 处访问，但是偏移量 1、2 和 3 会触发对齐检查中断。开发人员有时利用这个功能来识别那些没有对齐的数据参考地址，以避免减慢关键部分代码的执行速度。

对齐检查错误的另一个用处是向指针中加入其他信息。例如，双字指针的低两位可用于标识不同的指针类型。例如，在使用指针之间，低两可能象这样标识数据类型：

位 1	位 0	数据类型
0	0	=字节
0	1	=字
1	0	=双字
1	1	=四字

在使用指针之前，总会清除低两位。如果这些位没有进行适当的清除操作，对齐检查就会标志这个错误。这可能是一种很好的编码实践，它可以避免令人费解的设计，并且不用完全依赖于对齐检查。

中断	描述	类型
12h	内存大小	BIOS
	机器检查	Pentium

这个中断返回全部主存的大小，单位千字节。它仅仅只是向 AX 中返回地址 40:13h 处主存字的内容。对于 640K 的主存，该值是 280h。

在 Pentium 和 Pentium Pro CPU 上，新的机器检查异常会引发中断 12h。机器检查是 Pentium 类处理器的可任选功能，用来标志读数据时的奇偶错误和一个总线周期的不成功执

行。如果发生了机器检查错误，运行中的程序就不能重新开始执行，当然，还不太清楚出现奇偶错误时操作是否还会正确，但是在关机之前可能会显示出问题的某些提示。

为了开放机器检查特性，必须开放 CR4 中的机器开放位第 6 位。如果使用了该特性，就可以和标准 BIOS 功能共享。为了能一起工作，中断 12h 处理程序必须查看机器检查类型寄存器的第 0 位：

```
mov     ecx, 1                          ; 机器检查类型
rdmsr                                   ; 寄存器读模式
test    ecx., 1                         ; edx:eax = 寄存器类型
jz      get_memory_size                 ; 如果机器不检查则跳转
<<<<<< machine cheak code here (not restartable) >>>>>>
hlt
```

出现奇偶错误或总线周期问题时的地址会自动保存在 64 位的机器检查地址中，如果没有出现机器检查，该值是没意义的。可以按下述方法读取它。

```
mov     ecx, 0                          ; 检查地址
rdmsr                                   ; 读模式寄存器
                                        ; edx:eax = 地址
```

对于奇偶错误，Pentium 芯片上的奇偶开放引脚必须设置高电平，并且必须检测所读取的数据是奇匹配，并且开放检查引脚必须设置为高电平，并且必须开放机器检查特性。参看第 3 章，详细了解读模式专用寄存器指令和相关寄存器。

中断	描述	类型
13h	硬盘/软盘服务	BIOS

软件触发这个中断来访问硬盘和软盘。参看第 10 章和第 11 章，获得全部细节。也可以参考中断 40h，它用来保存软盘 BIOS 服务中断向量。

中断	描述	类型
14h	串行通信	BIOS

软件触发这个中断来初始化以及向串行口收发数据。第 12 章描述了该串行 BIOS 中断。

中断	描述	类型
15h	系统服务	BIOS

在早期的 PC 上，中断 15h 用来控制磁带驱动器。很高兴我从未使用过它。在后来的系统上，它发展成为所有难以归入其他类的功能总集。参看第 13 章以获取其他细节。

中断	描述	类型
16h	键盘服务	BIOS

软件触发这个中断来提供键盘缓冲区的操作系统及应用程序之间的主要接口。第 8 章讨论了中断 16h 操作。

中断	描述	类型
17h	打印机服务	BIOS

软件触发这个中断来初始化以及向打印机端口收发数据。第 14 章详细讨论了中断 17h。

中断	描述	类型
18h	ROM BASIC/引导失败	BIOS

如果没有找到引导设备就会调用这个中断。非 IBM 的系统通常只是返回一个信息，提示需要一张启动软盘。在 IBM 机器上，这会指向 ROM BASIC。几乎没有其他生产商象 IBM 那样将 BASIC 放入 BIOS 中。我认为，这是由于 Microsoft 和 IBM 最初达成的协议在起作用：如过 IBM 在其机器中包括了 Microsoft 的 BASIC，那么通过这项协议可以减少 IBM 的版税。正因为如此，每个 IBM 系统仍然含有那些过时的 ROM BASIC，并把它作为系统 BIOS 的一部分。

一种新型的 BIOS 引导说明文档（4Q1995）重新使用这个中断来支持从其他设备，例如 CD-ROM，来引导系统。如果引导设备（比如硬盘）不能引导，就会触发中断 18h，这就可以尝试从另外的设备引导。参看附录 C 文献，以获得有关 BIOS 引导说明文档的信息。

中断	描述	类型
19h	引导装入程序	BIOS

一旦 BIOS 完成了 POST 操作，就会触发中断 19h，从引导设备的 0 磁道 1 号扇区装入引导扇区。该数据然后传送到开始于固定地址 0:7C00h 处的内存中。装入 512 字节后，控制会传送给 0:7C00h 处的装入代码起点。DL 保存有扇区读自的驱动器号。如果发生任何问题，就会触发中断 18h。

许多人错误地认为，简单地触发中断 19h 就会重启系统。中断 19h 应仅由 BIOS 调用，并且应在正确的重启硬件以及 BIOS 数和中断向量设置为重启状态之后触发。如果应用程序试图调用中断 19h，系统可能会被挂起，这取决于 BIOS 数据区和中断向量表的状态。参看第 3 章，CPU 和内幕指令的结尾部分，了解正确重启系统的方法。

中断	描述	类型
1Ah	日期	BIOS

软件触发这个中断来访问日期功能。早期的 PC 并不支持它。第 15 章详细讨论了日期功能。

中断	描述	类型
1Bh	键盘 Control-Break	BIOS

低级键盘处理程序中断 9，检测 Control-Break 组合键。如果检测到该组合键，不会清空键盘缓冲区，设置在 40:17h 处的 BIOS Control-Break 标志，并由键盘处理程序触发中断 1Bh。通常操作系统在发生 Control-Break 时利用作为陷入软中断。如果操作系统是 DOS，那么这个中断会指向 DOS 内的一小段程序。这个程序记录了 Control-Break 的发生。然后，如果允许任何一个 DOS 中断 21h 功能，DOS 就会调用用户的 Control-Break 处理程序。用户的 Control-Break 处理程序地址保存在 0:8Ch 处（中断 23h 向量）。参看中断 23h 获得其他细节。

中断	描述	类型
1Ch	用户时钟滴答	BIOS

在 BIOS 系统时钟完成之前，它触发中断 1Ch。1Ch 的 BIOS 服务处理程序是最小的服务处理程序：BIOS 仅仅只是发 IRET 命令返回而不做任何事情。

这个中断唯一的目的是每 54.9ms 向应用程序发一个信号。要实现这一点，应用程序应先挂起中断 1Ch。参看中断 8 和第 16 章以获取更多信息。

中断	描述	类型
1Dh	视频初始化指针	数据

这是一个远指针，它指向 CGA 和单色视频卡的初始化值。通常它指向 ROM，但也可以指向一个用户表，当改变模式时，适当组的 16 字节数据被装入到 6845 CRT 控制器中，CGA 使用端口 3D4h 来指定使用哪个内部寄存器，端口 3D5h 用于向控制器发送数据。16 字节的数据被装入到了控制器上的 0~15 号寄存器中。对于 MDA 的做法完全一样，只是后来使用端口 3B4h 和 3B5h 罢了。表 7-5 描述了这个表的内容。

表 7-5 视频初始化参数表

偏移量	字节数	描　　述
0	16	CRT 控制器的安装数据，40 列 25 行（模式 0、1）。
10h	16	CRT 控制器的安装数据，80 列 25 行（模式 2、3）。
20h	16	CRT 控制器的安装数据，安装数据、图形（模式 2、5、6）。

续表

偏移量	字节数	描　　述
30h	16	CRT 控制器的安装数据，80 列 25 行（模式 7）。
40h	2	视频缓冲区大小，CGA 40x25 模式（800h）。
42h	2	视频缓冲区大小，CGA 80x25 模式（1000h）。
44h	2	视频缓冲区大小，CGA 图形模式（4000h）。
46h	2	未使用。
48h	8	模式 0 到 7 的列数。
4Ah	8	视频模式 0 到 7 发送到端口 3B8h/3D8h 的内部 CRT 控制器。

中断	描述	类型
1Eh	软盘配置指针	数据

该远指针指向软盘配置信息表。通常它指向 ROM，但是也可以指向一个用户表。表 7-6 归纳了软盘参数表。第 10 章提供了其他细节。

表 7-6　软盘参数表归纳

偏　移　量	字　节　数	描　　述
0	1	步进速率和磁头下载时间。
1	1	磁头装入时间和 DMA 模式标志。
2	1	上次访问后马达关闭所需的时间。
3	1	每扇区的字节数。
4	1	每磁道的扇区数。
5	1	扇区间的间隙长。
6	1	每扇区字节数为 0 时数据长度。
7	1	格式化期间扇区的间隙长。
8	1	格式化字节值。
9	1	磁头定位时间。
A	1	马达达到正常速度的时间。

中断	描述	类型
1Fh	图形字符集指针	数据

CGA 适配卡的图形模式 4、5、6 支持使用视频中断处理程序来向屏幕写字符。在系统 BIOS 中它支持 ASCII 字符 0 到 127。BIOS 不包含字符 128 到 225 的字体，并将其指针设定为 0:0.

可以将该指针指向用户提供的字符 128 到 225 的字体表。该字体模式表由每字符 8 字节的数据组成。例如，字母“I”的 8 字节数如下所示：

```
letter_I:       db      01111000b       ;“I”的顶部
                db      00110000b       ;“I”的中部为两位宽
                db      00110000b
                db      00110000b
                db      00110000b
                db      00110000b
                db      00111100b       ;“I”的底部
                db      00000000b       ; 空白线
```

注意：

剩下的有些中断并未包括在下面，它们用于操作系统、适配器或一般用途。文献中列出的许多技术参考书籍详细介绍了这些中断。

中断	描述	类型
21h	通用服务	DOS

这个中断提供广泛的操作系统服务。参看任何一种 DOS 技术书籍以获取完整细节。参看《DOS 内幕》了解许多隐藏功能和 DOS 的内部细节。

中断	描述	类型
22h	终止地址	数据

当完成一个 DOS 应用程序时，系统必须终止当前进程。该向量地址保存了指向可执行代码的指针。终止进程时，控制被传递给该代码。如果出现了一个关键性错误并且执行码是 2（终止），该地址就用来永久地终止当前的应用程序。它不是一个可调用的中断向量，永远也不可以作为一个中断进行触发。

中断	描述	类型
23h	Control-C 地址	数据

在所有的 DOS I/O 操作和大多数 DOS 功能期间如果按下了 Control-C 或者 Control-Break，该地址包含了 DOS 跳往的地址。DOS 中止标志的状态不影响该操作。甚至是 BREAK

OFF，DOS 仍会在按下 Control-C 或 Control-Break 时跳往保存在这个向量中的地址。它不是一个可调用的中断向量，永远也不可以作为一个中断进行触发。

Control-C 处理程序总可以在执行任何可行的 DOS 服务时进行调用。如果应用程序将该地址指向了用户代码，那么用户的 Control-C 处理程序将执行下面三操作之一：

1. 采取所有认为必要的措施，包括保存按下 Control-C 时的标志位。该 Control-C 处理程序在完成时发一个 IRET 命令。当然，该处理程序必须保存好所有的寄存器。
2. 采取所有认为必要的措施，并在结束时带进位标志指示器调用 IRET。如果进位标志被清零，DOS 继续正常执行。如果进位标志置 1，DOS 会中止当前应用程序。
3. 采取所有认为必要的措施，并直接跳回到应用而不返回 DOS。这种情况设计 Control-C 功能可以不用 IRET 功 RETF。

中断	描述	类型
24h	严重错误处理程序地址	数据

如果在一次操作中 DOS 检测到了某个错误，DOS 就使用保存在该位置的地址来将控制传递给一个错误处理程序。错误处理程序的主要任务是告诉操作系统在发生错误该如何继续。它不是一个可调用的中断向量，永远也不可以作为一个中断进行触发。

DOS 提供一个缺省的错误处理程序，该 DOS 错误处理程序会显示信息“Abort Retry or Fail?”然后依据用户的反应作出不同的处理。

应用程序可以替代该错误处理程序以改进处理或者避免从屏幕的中间显示信息“Abort Retry …”。如果应用程序将自己的错误处理程序的地址放在向量地址中，出现错误就会调用它。在程序退出时不必恢复该地址，DOS 在程序终止时会自动重设该地址。获得控制的应用程序将从 DOS 那里收到下述信息：

ah = 错误条件及支持选项标志

bit 7 = 0　磁盘错误（通常经过多次测试）

1　错误类型取决于 bp:[si+4]处的设备属性。

如果第 15 位=0，那么是文件分配表的

内存映像出了问题。如果第 15 位=1，那么

是字符设备出错。

device.

6 = 0　未使用

5 = 0　不允许“忽略”错误选项

1　允许“忽略”错误选项（仅 DOS3.0+）

4 = 0　不允许“重试”选项

1　允许“重试”选项（仅 DOS3.0+）

3 = 0　不允许“失败”选项

1 允许“失败”选项（仅 DOS3.0+）

2 = x | 磁盘出错时的错误位置

1 = x |

位 2	位 1
0	0 = DOS 区域
0	1 = 文件分配表
1	0 = 根目录
1	1 = 其他区域

0 = 0 读操作时出错

1 写操作时出错

al = 设备号，如果 al 的第 7 位 = 0（0 = a:, 1 = b:, 2 = c:, 等。）

bp:si = 指向设备表头控制块的指针

di = 错误代码

位 15-8 = x 未定义

位 7-0 = 错误代码：

0	出现写保护错误
1	未知单元
2	驱动器未准备好
3	触发了未知命令
4	数据错误或 CRC 坏
5	驱动器请求结构长度不正确
6	寻找错误
7	未知媒质类型
8	未找到扇区
9	打印机缺纸
Ah	写错误
Bh	读错误
Ch	一般性错误
Dh	共享错误*
Eh	锁存错误*
Fh	无效的磁盘变更*
10h	文件控制块（FCB）不可用*
11h	共享缓冲区溢出*
12h	代码页不匹配*
13h	输入无效*
14h	磁盘空间不够*

* DOS 3.0+

错误处理程序应保存好所有的寄存器，包括堆栈指针。在执行错误处理程序时，只能访问 DOS 中断 21h 的有限功能，包括功能 1~Ch、50h、51h、59h、62h，以及功能 33h 的某些子功能。在错误处理程序完成时，它发出 IRET 命令，并在 AL 寄存器中返回如下动作码：

0	忽略错误
1	入口
2	使用保存在 0:88h 的地址中止程序（中断 23h）
3	正在进行的系统调用失败

中断	描述	类型
30h	跳至 DOS 的 CP/M 服务	数据

DOS 在开始于 0:Ch 的地址处插入 5 字节的远跳转指令。该指令跳到 DOS 区域以便处理老式的 CP/M 功能。这个功能不仅过时而且未加说明，并且保护模式下的 DPMI 新服务处理程序会使用中断 31h 来破坏该 5 字节远跳转的最后一个字节。

中断	描述	类型
34h	浮点模拟	数据

许多编译器，包括那些 Microsoft 与 Borland 所提供的编译器，都使用中断 34h 到 30h 进行浮点模拟。通过使用一个中断，并紧接着浮点指令的源指令字节，编译器执行中断 34h 到中断 3Bh 来模拟数学协处理器指令。例如，一个普通的数学协处理器指令是：

```
D8  C1      FADD      ST，ST（1）
```

所有的 D8 数学指令都由编译器建立以使用中断 34h 。模拟指令会是：

```
CD      34          INT       34h
C1                  db        0C1h
```

程序首次执行这个特殊中断 34h 时，中断 34h 处理程序（是程序的一部分）检查系统是否有数学协处理器。如果系统有一个数学协处理器，它就会将代码修改为正确的协处理器指令。在上面这个例子中，三个字节 CD、34 和 C1 变成了 9B（等待）、D8 和 C1。中断程序改变了执行起点的返回地址，并返回实指令。这种情况下，只在第一次执行该指令时有一点执行困难。尽管我们都被教导，不要使用自我修改的代码，但是这种聪明的方法的确节省了代码空间，而同时在所有类型的系统上获得最高的运行速度。

如果中断 34h 处理程序没有发现数学协处理器，那么这个程序必须模拟该指令。由于调用了中断 34h，处理程序知道它是一条 D8 系列指令。处理程序然后找中断指令后的字节来完整的确定哪条指令被请求。接着处理器模拟该特定的指令。完成模拟之后，处理程序必须在该例的 C1 字节后返回。记住，不同的指令可能会用到中断后的不止一个字节。

中断	描述	类型
35h	浮点模拟器	用户

模拟以操作码 D1h 为开始的数学协处理器指令。参看中断 34h 以了解细节。

中断	描述	类型
36h	浮点模拟器	用户

模拟以操作码 D2h 为开始的数学协处理器指令。参看中断 34h 以了解细节。

中断	描述	类型
37h	浮点模拟器	用户

模拟以操作码 D3h 为开始的数学协处理器指令。参看中断 34h 以了解细节。

中断	描述	类型
38h	浮点模拟器	用户

模拟以操作码 D4h 为开始的数学协处理器指令。参看中断 34h 以了解细节。

中断	描述	类型
39h	浮点模拟器	用户

模拟以操作码 D5h 为开始的数学协处理器指令。参看中断 34h 以了解细节。

中断	描述	类型
3Ah	浮点模拟器	用户

模拟以操作码 D6h 为开始的数学协处理器指令。参看中断 34h 以了解细节。

中断	描述	类型
3Bh	浮点模拟器	用户

模拟以操作码 D7h 为开始的数学协处理器指令。参看中断 34h 以了解细节。

中断	描述	类型
3Ch	浮点模拟	用户

这表明在模拟下一条数学指令时需要 ES 越界。如果激活了一个数学协处理器，处理程序会用 9B（等待）和 26（ES 前缀越界）覆盖中断 3Ch 的指令字节 CD 和 3C。参看中断 34h 以进一步了解模拟浮点中断。

中断	描述	类型
3Dh	浮点模拟	用户

这表明应模拟 WAIT 指令。WAIT 指令常常用在浮点指令之后。这条指令使 CPU 处于等待状态，直到处理了前面指令产生的任何可能的错误。如果激活了数学协处理器，处理程序会用 90（空指令 NOP）和 9B（等待）覆盖中断 3Ch 的指令字节 CD 和 3C。参看中断 34h 以进一步了解模拟浮点中断。

中断	描述	类型
40h	重新映射软盘服务	BIOS

检测硬盘时，BIOS 将软盘 BIOS 服务从中断 13h 切换到中断 40h，它将中断 13h 向量指向了硬盘 BIOS 服务处理程序。调用该硬盘服务程序时，它检查 DL 的第 7 位是否为 0。如果是 0，就会触发中断 40h 来处理软盘操作。如果 DL 的第 7 位是 1，就执行指定的硬盘操作。

中断	描述	类型
41h	硬盘 0 的配置指针	数据

该入口是指向硬盘 0 配置信息的一个远指针。BIOS ROM 带有许多预设的驱动器类型，该指针经常指向 BIOS 驱动器类型表内的一个入口。表 7-7 列出了硬盘参数表的内容。大多数较新的 BIOS 经常提供了用户自定义的驱动器类型，可以将该指针设定指向保存在 RAM 中类型信息。例如，较老式的 AMI BIOS 就带有一个选项来在 0:C00h 处的中断表中保存用户的硬盘配置。大多数当今的系统都将用户自定义数据放在映像主存 BIOS ROM 的某个区域，映像将 BIOS ROM 内容移到 RAM 中。在将用户的值写到了 BIOS RAM 区后，该区就成为写保护的了。第 11 章提供了完整的细节。

表 7-7　硬盘参数表归纳

偏　移　量	大　小	描　述	系　统
0	字	磁柱的最大数目	XT/AT+
2	字节	磁头的最大数目	XT/AT+
3	字	减少写电流的起点磁柱	仅 XT
5	字	写预补偿的起点磁柱	XT/AT+
7	字节	最大 ECC 脉冲长度	仅 XT

续表

偏 移 量	大 小	描 述	系 统
8	字节	控制字节	XT/AT+
9	字节	标准的超时值	仅 XT
A	字节	格式化磁盘命令的超时值	仅 XT
B	字节	检查驱动器命令超时	仅 XT
C	字	装入区段	AT+
E	字节	每磁道的扇区数	AT+
F	字节	未使用	所有

警 告

记住，有许多技术参考资料指出硬盘 1 可使用保存在中断 41h 内的指针，这是不正确的。它一直只供硬盘 0 使用。

中断	描述	类型
42h	老式的视频中断向量	EGA+

系统 BIOS 初始化视频 BIOS ROM 时，视频 BIOS 将老式的视频中断 10h 保存在中断 42h 位置中。然后，视频 BIOS 将自己的向量放在中断 10h 中。大多数情况下中断 42h 会指向系统 BIOS 视频处理程序，一般后者不再使用。

中断	描述	类型
43h	个图形字符设置指针	数据

EGA/VGA 以及后来的视频适配器使用该指针来获得图形模式下文本的字体模式。对于视频 4、5 和 6，该字体模式供前 128 个字符（以看中断 1Fh 以了解指向前 128 个字符字体的指针）使用。在其他的图形模式下，它指向所有 256 个字符的字体模式表。

通常该指针由视频服务程序，中断 10h 功能 11h 设置。它允许该指针设置为一系列保存在视频 BIOS ROM 中的字体模式。这些服务也支持设置用户字体。

警 告

由于早期的 IBM EGA 说明文档有点错误，许多技术参考资料都指出该指针保存在 0:110h 处的向量 44h 中，这是不正确的。

中断	描述	类型
46h	硬盘 1 的配置指针	数据

该入口是指向硬盘 1 配置信息的一个远指针。BIOS ROM 带有许多预设的驱动器类型，该指针经常指向 BIOS 驱动器类型表内的一个入口。

大多数较新的 BIOS 经常提供用户自定义的驱动器类型。可以将该指针指向保存在 RAM 中的类型信息。例如，某些 BIOS 就带一个选项来在 0:C10h 处的中断表中保存用户的硬盘配置，参看第 11 章以获得其他细节。该硬盘表和表 7-7 所示具有相同的形式。

警　告

记住，有许多技术参考资料指出该指针是供硬盘 0 使用的，这是不正确的。

中断	描述	类型
4Ah	实现警报	BIOS

中断 4Ah 是一个由用户提供的中断处理程序，当触发了 CMOS 警报时就会调用该处理程序。用户使用 BIOS 服务 1Ah 来设定警报时间。如果警报时间和当前时间相同，就会触发中断 70h。而中断 70h 的处理程序反过来又会触发中断 4Ah。在第 15 章中断 1Ah 下有更多的细节谈到这一点。

中断	描述		类型
70h	IRQ8	CMOS 实时时钟	硬件

AT+系统的 CMOS 实时时钟中断线连着 IRQ8。这个中断线在 RTC 芯片的周期性模式和警报模式下可以被激活。这两种模式可以同时执行而发生冲突。

在 RTC 的周期性模式下，中断线按一定的周期性频率进行切换。BIOS 使用这种模式来实现定时，定时期间由中断 15h 的功能 83h 和 86h 设定。可以对 RTC 进行编程使其按 1024Hz 的频率中断，中断 70h 处理程序递减 40:9Ch 处的双字计数器。当时间到期时，就会关闭周期性中断模式，并将用户标志字节设定为 80h。用户标志双字指针保存在地址 40:98h 中。

在警报模式下，如果预设的警报时间和当前 CMOS 时间匹配，就会触发这个中断。使用 BIOS 中断 1Ah 的功能 6h 来设定警报时间。如果发生了警报，中断 70h 处理程序会触发中断 4Ah 来执行用户的警报处理程序。

参看第 15 章获得其他信息。

中断	描述	类型
71h	IRQ9 一般用于适配器用途	硬件

AT+系统没有分配这个中断可以用它来连接 IRQ9。针对 IRQ9 以及它和 IRQ2 的特殊

关系，存在许多误解。

首先，需要讲点背景。在设计 AT 时，IBM 又加入了一个中断控制器，以提供另外七个中断。这种中断控制器的设计方式允许两个中断控制器按特殊的“链接”模式同时工作。这要求前八个中断中的某一个贡献出来进行链接处理。IBM 选择了 IRQ2。这意味着系统不能再使用 IRQ2。

为了保证与 PC/XT 的单中断控制器保持兼容，对 IRQ9 作了特别处理。AT+系统连上 IRQ9 线的适配器总线与过去 XT/PC 的 IRQ2 的适配器完全相同。系统和供应商常常说 IRQ2，实际上指的是 IRQ9。如果激活了 IRQ9，中断控制器就会触发中断 71h。

在许多聪明的 BIOS 基础工作的帮助下，AT 系统可以和需要 IRQ2 的软硬件 100%兼容。实现这一点时，BIOS 中断 71h 处理程序只为第二个中断控制器触发结束中断（EOI）。然后，BIOS 触发中断 0Ah。这一点是在模拟早期的 PC/XT 服务，在那里 IRQ2 指向了中断 0Ah。这样，AT+系统就好像在软件和硬件上和早期的 PC/XT 完全相同。

由于处理 IRQ9 时有点不同寻常。所以在许多技术参考资料中常常会有一些错误。某些将 IRQ9 的物理线错误地表示为 IRQ2 的，尽管它的确执行 IRQ2 的功能，从软件的视角看，这一点并不重要。作为一个程序员，你可以认为所有的系统带有实 IRQ2 或虚 IRQ2。因此编程并无区别。

表 7-8 的前两个入口显示了同一个适配卡是如何在 PC/XT 和 AT+系统上工作的。作为一种替代方法，第三个入口显示了只适用 AT+卡的适配器。第三个入口绕过了中断 71h 的处理程序而直接处理中断 71h，通常不建议这样做，因为其他的适配器会不能使用虚 IRQ2。

表 7-8 IRQ2 和 IRQ9 操作

系　　统	实 IRQ	虚 IRQ	中断处理程序
PC/XT	2	无	0Ah
AT+	9	2	0Ah
AT+	9	无	71h

中断	描述	类型
72h	IRQ10	硬件

这个中断未被分配，可以用于连接 IRQ10 的 16 位适配器。

中断	描述	类型
73h	IRQ11	硬件

这个中断未被分配，可以用于连接 IRQ11 的 16 位适配器。

中断	描述	类型
74h	IRQ12	硬件

在带有内置鼠标端口的系统上，IRQ12 用来通知 BIOS 可能使用来自鼠标的数据。参看第 8 章以了解更多的有关鼠标端口的细节。对于没有内置鼠标端口的系统。这个中断未被分配，可以用于连接 IRQ12 的 16 位适配器。

中断	描述	类型
75h	IRQ13——数学协处理器错误	硬件

在 IBM 和完全兼容的 AT+系统上，来自数学协处理器的中断/错误线与 IRQ13h 相连。激活时会触发中断 75h。

在 PC/XT 上，数学协处理器中断线和 NMI 中断 2 相连。通过中断 75h BIOS 处理程序 AT+系统保持与 PC/XT 兼容。该处理程序向两个中断控制器发生结束中断信号，然后触发中断 2 这个不可屏蔽中段。

一些 AT+系统的协处理器中断/错误线错误地绕在主板上。参看中断 10h 以获取对这个中断的其他阐释。

在 EISA 系统上，这个中断也用于 DMA 适配器。如果 DMA 位于链接模式。当 DMA 在操作中需要下一组内存地址时，它就会向系统发出信号。参看第 18 章以获得 EISA DMA 中断 75h 的用法。

中断	描述	类型
76h	IRQ14 — 硬盘控制器	硬件

对于 AT+系统，如果主硬盘控制器完成了一个动作，它不会向 IRQ14 发出信号。主硬盘控制器可以处理多达两个设备。AT 和 EISA 系统的中断 76h 处理程序会将 40:8Eh 处的字节设定为 FFh 来标志操作完成。中断 76h 处理程序也触发中断 15h 功能 AX=9100h 来完成了硬盘控制器操作。参看地第 11 章获得更多细节。

中断	描述	类型
77h	IRQ15 — 通用适配器用途	硬件

这个中断未被分配，可以用于与 IRQ15 相连的 16 位适配器。对于带有两个 IDE 控制器（最多 4 个 IDE 设置）的系统，这个中断主要用于第二个控制器。第二个控制器用于第三个和第四个硬盘或相关的 IDE 设备。

中断	描述	类型
C0~C3h	硬盘 0 的用户自定义数据	数据

在某些 AT+系统上，BIOS 可以提供一个用户自定义的硬盘参数集，前面讨论的中断 41h 中阐释了指向 16 字节组的指针。创建一个用户自定义的入口时，会将它保存在 CMOS 内存中。系统的 POST 会将这 16 个字节从 CMOS 传给开始于地址 300h 的 RAM。保存在 RAM

在址 0:104h（中断定 41h）中的指针指向了这个数据。并不是所有的 BIOS 支持用户自定义的硬盘类型，并且对于那些支持用户自定义的 BIOS，有许多 BIOS 并不将信息保存在此处。

中断	描述	类型
C4~C7h	硬盘 1 的用户自定义数据	数据

在某些 AT+系统上，BIOS 可以提供一个用户自定义的硬盘参数集，在前面讨论的中断 46h 中阐释了指向 16 字节组的指针。在创建由一个用户自定义的入口时，会将它保存在 CMOS 内存中。系统的 POST 会将这 16 个字节从 CMOS 传给开始于地址 310h 的 RAM。保存在 RAM 在址 0:118h（中断定 41h）中的指针指向了这个数据。并不是所有的 BIOS 都支持用户自定义的硬盘类型，并且对于那些支持用户自定义的 BIOS，有许多 BIOS 并不将信息保存在此处。

第8章

键盘系统

表面上看起来键盘系统似乎比较简单，但是如果在最低层次上看键盘系统，其实它也相当复杂。在早期的 PC、AT 和 MAC 彼此之间，键盘控制已有了显著的改变。现在实际上有两个独立于主 CPU 的微处理器在运行键盘系统。

本章详细介绍了键盘操作的每一步发生的动作，涉及从按下一个键到应用程序接收到该键的每一步，包括两个微控制器、几个中断服务程序等等。

正如你早已知道的那样，键盘代码的翻译分好几个层次。我将介绍它是如何对一张表进行工作的，这个表涉及键盘到控制器的代码、控制器到系统的代码以及最终得到的 ASCII 码。与我所见过的任何一个表都不同，这个表是按便于查询的字母顺序排列的。

本章的后面部分介绍了如何访问键盘控制器，并提供了一个完整的源代码来收发信息和改变键盘的 LED 状态。一个程序实例还将所有的这些子程序串起来以显示那些少见的键盘到控制器代码的扫描码。

最后，我汇集了广泛的 I/O 端口功能和命令列表，它们是访问键盘控制器和硬件鼠标端口所必须的。该列表包括了键盘控制器尚未说明的功能和特征。例如，键盘控制器也处理 A20 地址线，该地址线用来访问一兆字节以上的内存空间。

简介键盘系统将按键符号转化为应用程序可用的代码。该系统还为应用程序、操作程序以及组合键直接操作提供许多服务。键盘系统的组成包括键盘本身、键盘同主板的接口、一个低级的 BIOS 中断服务处理程序（中断 9）、一个中间 BIOS 中断服务处理程序（中断服务处理程序（中断 16）以及操作系统接口。

这些部分组合起来，就为运行于操作系统之上的应用程序提供了键盘输入功能。不幸的是，在试图为应用程序简化键盘的信息访问时，产生了一系列的问题。许多应用程序必须在操作系统接口下访问键盘以提供所需功能。这些功能包括：

- 访问尚未支持的按键组合
- 捕获按键
- 按键替代
- 键盘“满”
- shift 键状态检测
- 增强运行效果

- 重复率和延时控制
- 终端驻留（TSR）热键

由于要保持在所有平台上的操作相同，必须尽可能地使用操作系统和中间 BIOS 服务。另外，这些接口通常有较好的文档说明。我们将只简要地谈谈这些服务。请参看你的操作系统的任何一本关于这些服务的好书，参考文献中也列出了某些参考资料。为了实现更高级的任务。你必须深入理解并学会访问键盘系统的低级功能部分。

基本操作

键盘带有一个 8031 或 8048 单芯片微控制器来处理所有的键盘动作。该控制器扫描键的按下和释放，在 AT 键盘上，它还控制三个 LED。它将键盘的按上和释放信息译码成键盘扫描码，该信息然后被传送到与主板相连的控制器还可以模拟某些组合键以和老式设计兼容。

早期的 PC/XT 主板带有硬件来将串行键盘数据转化为字节数据。该字节读自 8255 可编程接口芯片的一个端口。系统的键盘 BIOS 软件就和该 8255 端口通信来获得键盘信息，例如扫描码。

在 AT 系统上，一种更加复杂的结构替代了这种老式的 PC/XT 设计。一个 8042 单芯片微控制器用来将该系统和主板连接起来。该控制器负责所有的键盘串行连接器的通信、数据有效性检查、缓冲 BIOS 和键盘之间的数据，以及将键盘扫描码翻译成扫描码。

与早期的 PC 和 XT 不同的是，AT 的键盘的串行连接器是双向的，这样，主板控制器也可以向键盘发送命令。主板控制器还处理几个与键盘毫不相干的系统功能，例如开放 A20 地址线以及触发硬件系统重启。图 8-1 列出了系统的组织结构。

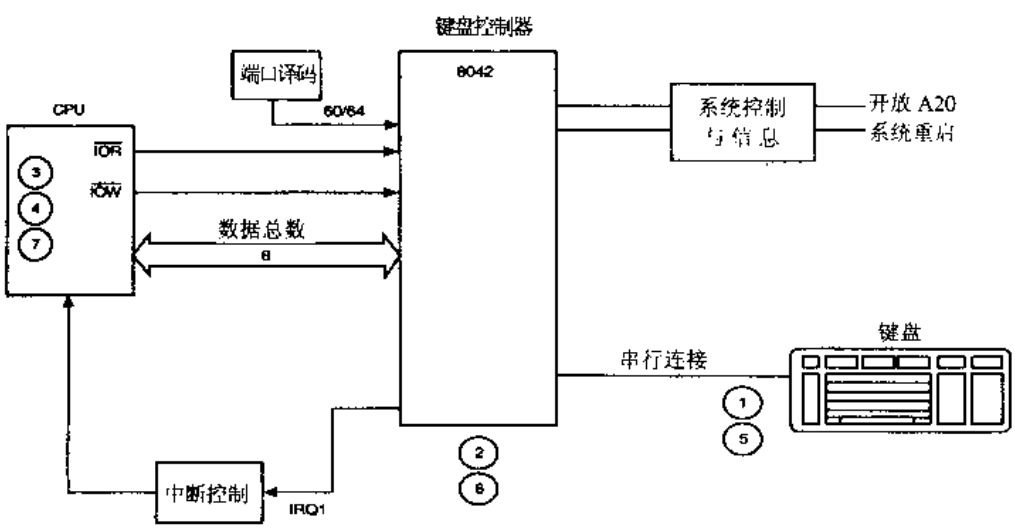


图 8-1 键盘控制器

AT 上的一个典型的按键操作

按下“P”键时，会引发一系列动作。首先，键盘内的 8031 控制器检测到按下了“P”键，并将值为 4Dh 的键盘扫描码发送到键盘串行连接器。主板上的 8042 键盘控制器收到该键盘扫描字节。8042 将键盘扫描码 4Dh 翻译成系统扫描码 19h，并将其放入输出缓冲区中。由于软件上的兼容，该扫描码结果和 PC/XT 键盘所产生的扫描码相匹配。然后，主板控制器触发一个中断请求来表明数据可用。这个中断请求调用中断 9 处理程序，键盘 BIOS。键盘 BIOS 从主板控制器读取该扫描码，并将它翻译成 ASCII 字节 70h。该例子中我们假定没有同时激活 shift 键。换一句话说，“P”键生成了一个小写“p”。键盘处理程序将扫描码和 ASCII 字节都放入 16 字节的 FIFO 键盘缓冲区的下一个可行的地点。最后，清除键盘中断，并退出键盘 BIOS，将控制返回全发生键盘时正在运行的任务。

操作系统或应用程序使用中断 16h，中间键盘 BIOS 服务，来访问键盘缓冲区。中断 16h 功能用来查看某个键是否可用以及确定某个键的值。中断 16h 的 Get Key（获取键）功能返回缓冲区中最早输入键盘的扫描码和 ASCII 码值。对于按下“P”键而言，这个功能返回值 1970h。中断键盘 BIOS 然后会从键盘缓冲区中移走该键。

当松开“P”键时，又会引发另外一系列的动作。主板上的微处理器检测到松开了“P”键，发送释放码值 F0H，并紧接着将键盘扫描码也发送到键盘串连接器。主板上的 8042 接收这两个字节，并将键盘扫描码 4Dh 翻译成系统扫描码 19h，同时将第 7 位置高来表示这是一个释放键。合成值 99h 然后被放在主板控制器的缓冲区。接着主板控制器触发一个中断请求，表明数据可用了。

中断请求会调用中断 9 处理程序，键盘 BIOS。键盘 BIOS 从主板控制器中读取该扫描码。由于释放“P”键没有任何作用，键盘 BIOS 会忽略该键。最后，清除键盘中断并再次退出键盘 BIOS。图 8-2 显示了键盘系统。

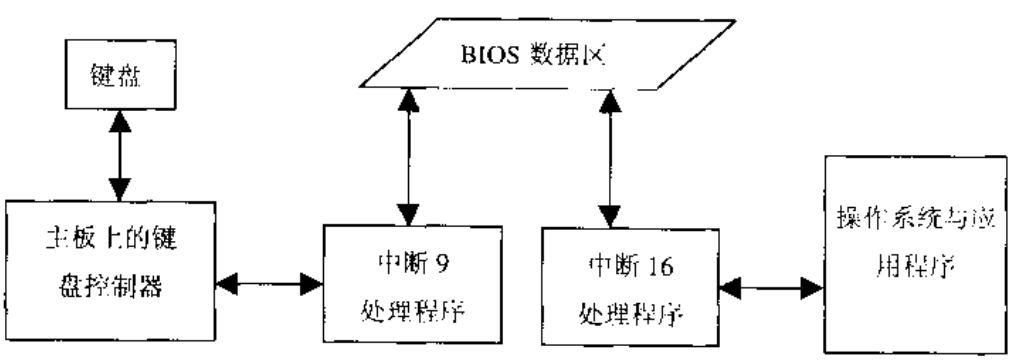


图 8-2　键盘系统

PC/XT上一个典型的按键操作

当按下“X”键时，会引发一系列动作，但是这些动作不同于当前 AT 系统上的动作。首先，主板内的 8048 微控制器检测到按下了“X”键，并将值为 2Dh 的扫描码发送到键盘串行连接器。主板逻辑会将该串行数据转化成一个字节并将其保存在锁存器中。该锁存器的输出连接到 8255 可编程接口芯片的端口 C 上，不需要象 AT+系统那样翻译扫描码。主板配件逻辑也会触发一个中断请求来表明主板有来自键盘的数据。这个中断请求调用中断 9 处理程序，键盘 BIOS。键盘 BIOS 从 8255 的端口 C 读取扫描码并将其翻译成 ASCII 字节 87h。这个例子中我们假定没有同时按下一 shift 键。键盘 BIOS 将扫描码和 ASCII 字节都放入 16 个字的 FIFO 键盘缓冲区的下一个可行的地点。最后，清除键盘中断并退出键盘 BIOS。

同前面讨论的 AT+系统一样，操作系统或应用程序使用中断 16h，中间键盘 BIOS 服务。来访问键盘缓冲区。中断 16h 功能用来查看某个键是否可用以及确定某个键盘的值。中断定 16h 的 Get Key（获取值）功能返回缓冲区中最早输入键的扫描码和 ASCII 码值。对于按下“X”键而言，这个功能返回 2D78h。中断键盘 BIOS 然后会从键盘缓冲区移走该键。

当释放“X”键时，又会引发一系列的动作。主板上的微处理器检测到释放了“X”键，并发送释放码值 ADh。这就是将第 7 位置 1 后的“X”的扫描码 2Dh，以表示该键刚刚被释放。按下该键时，该值被传给串行连接器并发送到 8255 的端口，产生一个新的中断来指示收到了新的数据。

中断请求调用中断 9 处理程序，键盘 BIOS。然后键盘 BIOS 从 8255 端口读取扫描码。由于释放“X”键没有任何作用，处理程序会忽略该键。最后清除键盘中断并再次退出处理程序。

控制器通信

在 AT+上，端口 60h 和 64h 同主板上的键盘控制器通信。端口 60h 用来读/写来自/发往键盘的信息。端口 64h 用来读取主板键盘控制器的状态，并向其发送命令。本章后面部分详细介绍了这些细节。

在 PC 和 XT 上，端口 60h 用于直接从主板读取数据。端口 61h 的几位可以禁止和开放键盘。第 13 章中介绍端口 61h。

键盘到主板的数据

主板控制器通过单独的一条串行线和键盘通信。当从键盘读取数据时，主板向键盘提供了一条同步时钟线。每 11 位串行帧包含一个起始位、8 个数据位和一个奇校验位、一个结束位。

=0	第 0 位	第 1 位	第 2 位	第 3 位	第 4 位	第 5 位	第 6 位	第 7 位	奇	=1

起始化位 —— 数据字节 —— 校验位 结束

在闲置状态下，所有的数据和时钟线都是高电平。开始向主板发送数据时，键盘在数据线上发送一个起始位（0）。主板的响应是开启时钟线，第一个时钟脉冲是下降沿。时钟继续送脉冲，键盘就轮流发送每一位。在第 11 个时钟脉冲时，键盘发送一个结束位（1），时钟线就恢复到闲置的高电平状态。

被发送的字节通常带有按键信息。另外，表 8-1 列出了键盘向主板发送的命令。在 PC 和 XT 键盘上只支持通信值 FFh。

表 8-1 键盘向主板的通信值

值	描述
0	超限定运行——内部 16 字符的先进先出缓冲区已经满了，并且可能丢失了键动作。该缓冲区与 BIOS 数据区中的 16 字键盘缓冲区毫不相干。（在具有多种扫描码集的键盘上，值 0 出现在集 2 和集 3 中，Overrun 使用集 1 来生成值 FFh。）
AAh	自检成功——通过了自检诊断。
EEh	回声（Echo）——在回声命令后的返回值。
F0h	Break 前缀——在释放一个键后，键盘发送一个 F0h，然后发送释放键的键盘扫描码号。
FAh	确认——键盘收到一个命令后，会返回这个确认字节。回声和重新发送命令不触发确认字节。
FCh	自检失败——没有通过自检诊断。
FDh	内部错误——键盘内部传感放大器的周期性测试失败。
FEh	重新发送——键盘收到了一条无效命令或发生奇偶错误时都会发送一个重新发送命令。
FFh	超限运行——内部 16 字符的先进先出缓冲区已经满了。并且可能丢失了键动作。（在具有多种扫描码集的键盘上，例如 PS/2，FFh 只出现在集 1 中，Overrun 使用集 2 和集 3 来生成 0 值。）

如果来自 AT+类型键盘的数据出现了奇偶错误，主板控制器会自动向键盘发送重试命令。如果在多次重试后仍无法消除奇偶错误，主板就会放弃该操作，并认为该数据不可读。设置奇偶错误状态位，并将主板控制器的输出缓冲区，端口 60h，设定为该奇偶错误码。错误码值取决于使用了命令 60h 送到端口 64h 的内部控制器命令字节。如果该命令字节的第 5、6、7 位都是 0，错误值就是 00h，否则错误值是 FFh。

按下键会发送一个扫描码或键盘扫描码给键盘控制器。如果一直按着该键，那么就会重复发送该码。当释放键时，除非是 Pause 键不生成释放码。

在键盘内部是一个先进先出缓冲区。在 PC/XT 上该缓冲区含 20 个字节，而在 AT+键盘上它有 16 个字节。如果没有太多的键盘动作，并且缓冲区充满，那么下一个动作产生一个过载码 0，并将之插入到丢失键信息的地方。如果一个键符号要求多个字节，那么全部的序列必须可以放入剩下的缓冲区中。如果不可以，那么出现过载，并将该键丢失。

在 AT+系统上，如果主板控制器接收到一个合适的按键键盘扫描码，那么通常会将其转化为系统扫描。然后从端口 60h 读取该扫描码。当按下 Escape 键时，键盘向键盘控制器发送键盘扫描码 76h。在普通的操作中，它会译成系统扫描码 01h。清除端口 64h 处的命令字节 20h 的第 6 位可以关闭普通扫描转化。

AT+键盘会对所有的传送作出响应，但命令 Echo 和 Respond 除外。键盘发送字节值 FAh 来表示成功地接收到上次传送。如果主板控制器在 25ms 内没有收到合适的反馈，主板控制器会将接收和传送的超时位都设定为 1。并将错误 FEh 装入主板控制器的输出缓冲区（读端口 60h）。

表 8-2 来自键盘的软件可读的通信值

值	描 述
0	键盘奇偶错误（如果命令字节的第 5、6、7 位都等于 0）。
F1h	没有安装密码（在命令 A4 之后），并非所有系统都使用。
FAh	对前面发送到键盘的数据进行确认。
FCh	鼠标传送错误，并非所有系统都使用。
FEh	键盘没有提供正确的确认信息。
FFh	键盘奇偶错误（如果命令字节的第 5、6、7 位不都为 0）。

AT 上主板到键盘的数据

前面谈到的串行格式也用于从主板向键盘发送数据。在随机状态下，数据和时钟都是高电平。开始向键盘发送数据时，主板先在数据线上发送一个起始位（0）。键盘的响应为启动时钟线，第一个时钟脉冲是下降沿。如果在 15ms 内还没有启动该时钟，或者第一个时钟周期超过了 2ms，就会标志出了问题，这时会设置主板控制器状态字节的第 5 位——传送超进位。另外，还会将错误值 FEh 装入到控制器的输出缓冲中，可以从端口 60h 读取它。

正常操作下，时钟会继续发出脉冲，主板控制器轮流发送第一位。在第 11 个时钟脉冲时，主板发送一个结束位（1），并将时钟线恢复为闲置状态高电平。

低级键盘 BIOS

当主板收到来自键盘的一个字节后，主板会触发硬件中断 IRQ1 来通知系统。这会触发由键盘系统 BIOS 处理的中断 9 服务。

AT+系统上的中断 9 服务程序执行下列动作：

1. 暂时禁止键盘和中断，因为程序还没有重入；
2. 从键盘控制器中读取字节；
3. 触发键捕获中断 15h，ah=4Fh，al=键，如果在返回时进位标志没有清零，则忽略该键并退出（详细细节请往下看）；
4. 检查有没有键盘命令重新发送、确认或者过载，必要时处理这些命令；
5. 如果 Cap、Num 或者 Scroll lock 发生了改变，更新 LED；
6. 处理该键（往下看）；
7. 开放中断，对中断控制器触发 EOI，重新开放键盘。

当键盘控制器发送一个扩展扫描码时，第一个读取的字节在 PC/XT 是零，而在 AT+键盘上是 E0h。这意味着必须再读一次才能获得扩展扫描码的字节。

处理键的操作取决于该键的组合情况。当检测到特殊的组合键时，键盘 BIOS 会采取许多特殊的行动。在这些情况下键不再插入到键盘缓冲区。表 8-3 列出了这些特殊的组合键。

表 8-3 特殊的组合键

组合键	键盘 BIOS 动作
Ctrl-Alt-Del	引起系统重启，但是跳过内存检查（热启动）。
Ctrl-Break	清空键盘缓冲区，并触发中断 1Bh。
Ctrl-Print Screen	将键盘回声切换到打印机。
Ctrl-Num Lock	暂停，直到按下下一个键，并吸收该键。
Pause	暂停，直到按下下一个键，并吸收该键。
Print Screen	触发中断 5。
System Request	按下时触发中断 15h，ax=8500h；松开时触发中断 15h，ax=8501h。
Alt-小键盘键	将不超过 3 位数字的数字键（0~255）转化为一个 16 进制值，并将该数字插入到 BIOS 键盘缓冲区中，就像按下了一个键一样。
Shift 键	保存新的换档（shift）状态（Ctrl、Alt、Shift、Inesrt、CapsLock、NumLock、ScrollLock）。
其他	将带有换档（shift）状态的扫描码转化为 ASCII，如果 BIOS 键盘缓冲区满，则发声，否则将一个字保存到键盘缓冲区，高字节是其扫描码而低字节是 ASCII 字符。

某些 BIOS 制造商还提供了其他用途的专用组合键，这些组合键因制造商的不同而有所差别，甚至同一制造商提供的不同版本之间也不相同。

表 8-4 非标准组合键

组合键	键盘 BIOS 的动作
Ctrl-Alt-减号	将处理器时钟设置为低速模式。在快速系统上运行时速度敏感的老式程序，这一点很有用。尽管目前很少使用，但有时它确实使某些问题处理得更快（减号位于数字键区内）。
Ctrl-Alt-加号	将处理器时钟设置为高速模式（加号位于数字键区）。
Ctrl-Alt-左 shift-减号	关闭系统高速缓存。提供这个功能是为了支持某些老式的高速缓存系统，当程序执行自我修改代码和其他非推荐使用的代码时，这些系统会出问题（较新的主速缓存提供了回写功能不避免这个问题）为了重新激活缓存，可以按下 Ctrl-Alt-左 shift-加号。
Ctrl-Alt-Esc	唤醒 BIOS 安装程序。
Ctrl-Alt-S	唤醒 BIOS 安装程序。
Ctrl-Alt-Ins	唤醒 BIOS 安装程序。

许多膝上型电脑还带有一些特殊的功能键来监视电池的剩余能量、设置单色显示器的颜色阴影效果，以及其他一些功能。没有一种统一的标准，并且每个膝上型电脑都做得各不相同（当然，某些制造商在自己的生产线上能生产相同样式的，但是这种情况非常少见）。

键盘 BIOS——中级

为了访问键盘缓冲区和状态，应用程序和操作系统必须通过中断 16h 使用键盘 BIOS 服务。这些服务从键盘数据获取键盘的状态，以及读取 shift 键的状态。如果可能，访问键盘就应该在这个层次上，或通过操作系统。中断 16h 由可达十种的服务组成，它取决于 BIOS 的具体执行情况。

功能	描述	平台
ah=0	读取键盘的输入	所有
ah=1	检查键盘状态	所有
ah=2	获取 shift 标志状态	所有
ah=3	设置键入速度和延时	AT+（某些）

ah=4	键盘敲击声调整	PC jr
ah=5	将键数据保存到缓冲区	AT+（大多数）
ah=9	重复键容量	AT+（某些）
ah=Ah	获取键盘 ID	某些
ah=10h	读取扩展键盘输入	AT+（大多数）
ah=11h	检查扩展键盘状态	AT+（大多数）
ah=12h	获取扩展 shift 控制	AT+（大多数）
ah=13h	DBCS shift 控制	DOS/V
ah=14h	shift 状态显示控制	DOS/V

中断	功能	描述	平台
16h	0	读取键盘输入	所有

这个功能读取下一个可能的键。如果在键盘缓冲区内有一个键，那么就从缓冲区中移走该两字节的值并将之保存在 AX 中。如果没有键，那么这个功能将等待直到有一个键，然后返回。

在 AT 系统上，如果没有键可读取，就会触发中断 15h，功能 AX=9002h，这会警告操作系统这个功能正在等待输入键，并且可能会去执行其他任务直到按下一个键。如果等待后收到了一个键，键盘 BIOS 会触发中断 15h，功能 AX=9102 来表示完成了这个中断，最后从缓冲区移走该键，并返回最初调用程序。

调用：	Ah=0
返回：	ah=扫描码 al=ASCII 字符 值 0 表示按下了一个非 ASCII 的功能键，比如 F1，AH 用于确定按下了哪一个功能键。
注意：	如果使用扩展键盘，一些独特的组合键会被转化成 84 键键盘所支持的键功能。使用功能 AH=10 来获取独特扩展信息。那些扩展键盘的键功能如果没有等同的 84 键键盘功能，那么早就会被扔掉了。

中断	功能	描述	平台
16h	1	检查键盘状态	所有

这个功能检查是否有键在缓冲区中等待被移走。如果有键，那么就会将扫描码和 ASCII 等值返回到 AX 中，但是并不从缓冲区中移走。

调用：　ah=1

返回：　如果没有键，则零标志位=1
如果有键，则零标志位=0
ah=扫描码
al=SACII 字符
值 0 表示按下一个非 ASCII 的功能键，比如 F1。AH 用于确定按下了哪能一个功能键。

注意：　如果使用扩展键盘，一些独特的组合键会被转化成 84 键键盘所支持的键功能。使用功能 11h 来获取独特扩展信息。如果扩展键盘的键功能如果没有等同的 84 键键盘功能，那么就会被忽略。

中断	功能	描述	平台
16h	2	获取 shift 标志状态	所有

这个功能返回 shift 标志的状态，该标志保存在 40:17h 处的 BIOS 数据区里。

调用：　ah=2

返回：　al=换档和切换状态

位		
	7=1	Insert 开
	6=1	CapsLock 开
	5=1	NumLock 开
	4=1	ScrollLock 开
	3=1	Alt 键按下
	2=1	Control 键按下
	1=1	左边 shift 键按下
	1=1	右边 shift 键按下

ah=值可能未被保存

注意：

当使用扩展键盘时，使用 AH=12h 功能可以获得另外的标志状态。

中断	功能	描述	平台
16h	3	重复键入速率和延时	AT+（大约）

几乎所有的 AT 及其后来的系统，以及某些 XT 系统，都能够设置重复键入速度，该重复键入速率是一个按下键每秒重复的次数。此外，这个功能还可以设置延时，延时指的是某键首次被按下到首次被重复的起始点之间的时间。

这个功能有六个子功能，只有某些 BIOS 才提供这些子功能，中断 16h，功能 AH=9 确定支持哪个子功能。参看功能 AH=9 获得其他细节。所有的 AT+系统都支持子功能 5，该子功能设置键入速度和延时。

子功能 AL=0（如果功能 AH=9 允许该子功能）

调用：　　ax=300h

返回：　　将键盘返回到缺省状态：重复键入设置为开，并将重复键入速率和延迟设定为缺省值

功能 AL=1（仅 PCjr）

调用：　　ax=301h

返回：　　重复键入延迟增加

子功能 AL=2（仅 PCjr）

调用：　　ax=302h

返回：　　放慢重复键入速率

子功能 AL=3（仅 PCjr）

调用：　　ax=303h

返回：　　重复键入延时增加，且放慢重复键入速率

子功能 AL=4（如果功能 AH=9 允许该子功能）

调用：　　ax=304h

返回：　　关闭重复键入特征

子功能 AL=5（AT+和某些 XT 系统）

调用：　　ax=305h

bl=待设置的重复键入速率

十六进制数	每秒字符数	十六进制数	每秒字符数
0	30.0	10	7.5
1	26.7	11	6.7
2	24.0	12	6.0
3	21.8	13	5.5
4	20.0	14	5.0
5	18.5	15	4.6
6	17.1	16	4.3
7	16.0	17	4.0
8	15.0	18	3.7
9	13.3	19	3.3

A	12.0	1A	3.0
B	10.9*	1B	2.7
C	10.0	1C	2.5
D	9.2	1D	2.3
E	8.6	1E	2.1
F	8.0	1F	2.0

*通常它就是缺省的值

bh=重复键入延迟

0=250ms

1=500ms

2=750ms

3=1000ms

返回：　键入速率和延时设定为指定的值

子功能 AL=6（如果功能 AH=9 允许该子功能）

调用：　ax=306h

bl=重复键入速率（参看前一个子功能）

bh=重复键入延时值

0=250ms

1=500ms

2=750ms

3=1000ms

中断	子功能	描述	平台
16h	4	键盘敲击声	PCjr 转换系统

这个功能控制 PC 可逆型和 PCjr 上的电子键盘敲击声响，某些早些的膝上型电脑也提供这种功能。

调用：　ah=4

al=0 关闭键盘敲击声

1 开通键盘敲击声

中断	功能	描述	平台
16h	5	将键数据保存到缓冲区	AT+（大多数）

几乎所有的 AT 及其后来的系统，以及某些 XT 系统，支持将键数据装入键盘缓冲区，就像按下了一个健一样。这个“塞入”的键放在键盘 FIFO 缓冲区的末尾，且不影响缓冲区中的其他键。这对于提供一种简单的宏能力或控制其他应用程序非常有用。本章后面的代码 8-7 检测了是否支持这个功能。

调用：　　ah=5
　　cl=ASCII 字符
　　ch=键盘扫描码
返回：　　al=0，如果操作成功
　　1，如果键盘缓冲区已满

中断	功能	描述	平台
16h	9	键入能力	AT+（某些）

它确定支持中断 16h 功能 3 的哪些子功能，它返回的位表明提供了哪个子功能。功能 3，子功能为所有的 AT+类机器所支持，所以没有必要检查是否支持它。

该操作有点令人容易产生误解！在触发功能 9 之前，必须触发中断 15h 功能 AH=C0h 来确定的确支持中断 16h 功能 9。下面的代码指出了其原理。第十三章，系统功能中的中断 15h 更详细地解释了这个功能。

```
mov     ah  ,0C0h               ;获取指向 ROM 配置的指针 es:bx
int     15h                     ;
jc      not_supported           ;如果不支持，则退出
test    byte  ptr  es:[bx+6],40h ;支持中断 16h,ah=9 吗?
jz      not_supported           ;如果不支持，则退出
mov     ah,  9                  ;执行这个中断功能
int     16h
```

一旦确定支持中断 16h，那么就可以触发功能 9：

调用：　　ah=9
返回：　　al=键入能力

位		
	7=x	未使用
	6=x	未使用
	5=x	未使用
	4=x	未使用
	3=1	支持中断 16h，功能 ax=306h
	2=1	支持中断 16h，功能 ax=305h
	1=1	支持中断 16h，功能 ax=304h
	0=1	支持中断 16h，功能 ax=300h

中断	功能	描述	平台
16h	Ah	获取键盘 ID	更新

从键盘获取键盘的 ID 字。这个功能利用端口 60h 命令从 F2h 来从键盘获取 ID 字。大多数 AT 和 PS/2 系统不支持这个功能，但是日本市场上的 IBM PS/2 model55 支持这个功能。键盘 ID 代码仅对日本市场上的 IBM 键盘有效。并且对其他供应商可能有所不同。

日本“G”和“P”型键盘同美国所使用的极为相似，但是前者提供了几个其他的功能，还可以输入日文字。“A”型键盘是一种更老式的设计，它提供了另外 24 个专用目的键。当使用中断 16h，功能 0、1、10h 以及 11h 时，所有这些键盘的动作相同。

调用：　　ah=Ah
返回　　bx=键盘 ID
　　0000=没有键盘
　　41AB=翻译模式下的新“G”型键盘
　　54AB=翻译模式下的新“P”型键盘
　　83AB=穿透模式下的新“G”型键盘
　　84AB=穿透模式下的新“P”型键盘
　　90AB=老式“G”型键盘
　　91AB=老式“P”型键盘
　　92AB=老式“A”型键盘

中断	功能	描述	平台
16h	10h	[illegible]	AT+ [illegible]

这个功能读取下一个可读的键。如果该键在键盘缓冲区中，那么就会从缓冲区移走该两字节值，并将之保存在 AT 中。如果没有键可以读取，这个功能处于等待状态直到有一个键可以读取，然后返回。几乎所有的 AT+系统和一些 AT 系统都支持这个功能。代码 8-7 用于检测是否支持这个功能。

在 AT 系统上，如果没有键可以读取，那么就会触发中断 15h 功能 AX=9002h。它会警告操作系统，这个功能正在等待输入键，可能会执行另外的任务直到收到一个键。如果等待之后收到了一个键，键盘 BIOS 会触发中断 15h 功能 AX=9002h 来表示完成了这个中断。然后从缓冲区移走该键，并返回到起初的调用程序。

调用：　　ah=10h
返回：　　ah=扫描码或者特殊字符 ID
　　al=ASCII 字符
　　值 0 或 E 表示按下了一个非 SACII 的功能键，比如 F1。然后使用 AH 确定按下了哪一个功能键

注意：

本功能不同于功能 0，对于功能 0，按任何一个键都会返回 ax=0D2Bh，但是对于功能 10h，shift 加号键返回 ax=0D2Bh，而灰色加号键返回 ax=4E2B。参看表 8-5 以了解键盘码。

某些扩展组合键只在使用功能 10h 和 11h 时才返回一个值，而在使用功能 0 和 1 时会忽略这些组合键。例如，使用功能 0 和 1 时会忽略 Alt-Esc。它们在表 8-5 中用灰色阴影显示。

中断	功能	描述	平台
16h	11h	检查扩展键盘状态	AT+

这个功能检查是否有键在缓冲区等待读取。如果有键可读取，那么就会将其返回到 AX 中，但是并不从缓冲区中移走。参看代码 8-7 检测是否支持这个功能。

调用：　ah=11h

返回：　零标志=1，如果没有键可以读取

零标志=0，如果缓冲区中有一个键

ah=扫描码或者特殊字符 ID

al=ASCII 字符

值 0 或 E0 表示按下了一个非 ASCII 的功能键，比如 F1。然后使用 AH 确定按下了哪一个功能键。

注意：

本功能不同于功能 1。扩展键盘键返回时未作改变。这一点支持区分某些键，比如是按下了 Gray 加号键还是 shift 加号键。对于功能 1，按下任何一个键都会返回 ax=0D2Bh，但是对于功能 11，shift 加号键返回 ax=0D2Bh，而 Gray 加号键返回 ax=4E2B。参看表 8-5 以了解键盘码。

中断	功能	描述	平台
16h	12h	获取扩展 shift 标志状态	AT+

这个功能将位于 BIOS 数据区 40:17h 处的 shift 标志状态返回并保存到 AL 中。另外，来自 BIOS 数据区 40:18h 和 40:96h 处的扩展 shift 标志被组合起来插入到了 AH 中。所有的 AT+系统和一些 XT 系统都支持这个功能。

调用：　ah=12h

返回：　al=换档和切换状态

位	7=1	Insert 开
	6=1	Caps Lock 开

	5=1	Num Lock 开
	4=1	Scroll Lock 开
	3=1	Alt 键按下
	2=1	Control 键按下
	1=1	左边 shift 键按下
	0=1	右边 shift 键按下
ah=扩展 shift 状态		
位	7=1	Sys Req 键按下
	6=1	Caps Lock 键按下
	5=1	Num Lock 键按下
	4=1	Scroll Lock 键按下
	3=1	右边 Alt 键按下
	2=1	右边 Ctrl 键按下
	1=1	右边 Alt 键按下
	0=1	左边 Ctrl 键按下

中断	功能	描述	平台
16h	13h	DBCS shift 控制	DOS/V

如果安装了日文的字体结束处理器（FEP），那么这个功能获取并设置双字节 字符集（DBCS）的 shift 状态信息。FEP 负责输入日文字并将之转化为双字节字符。

子功能	描述	中断	功能
AL=0	设置 DBCS shift 状态	16h	AH=13h

该子功能装入 DBCS shift 状态。

在改变 shift 状态之前，必须先使用功能 1301h 获取当前的状态。这里所讨论的位可以被改变，但是内部状态位不得改变。这意味着第 3、4、5 以及 8 到 15 位在写入时必须和读自功能 1301 的值一样。

调用：	ax=1300h	
	dx=shift 状态	
	位 15~8=x	内部状态
	7=0	没有从 KataKana 到 Kanji 的转化
	1	从 KataKana 到 Kanji 转化模式
	6=0	关闭 Romafi
	1	开放 Romaji
	5=x	内部状态
	4=x	内部状态

3=x	内部状态	
2=x 1=x	字符输入模式	
	第 2 位	第 1 位
	0	0=字母数字
	0	1=KataKana
	1	0=Hiragana
	1	1=未使用
0=0	一半大小	
1	全部大小	

返回：　　无

子功能	描述	中断	功能
AL=1	获得 DBCS shift 状态	16h	AH=13h

该子功能获取当前 DBCS 的 shift 状态。如果没有安装 FEP，DX 寄存器中返回零值。

3=x	ax=1301h		
	dx=shift 状态		
	位 15~8=x	内部状态	
	7=0	没有从 KataKana 到 Kanji 的转化	
	1	从 KataKana 到 Kanji 转化模式	
	6=0	关闭 Romafi	
	1	开放 Romaji	
	5=x	内部状态	
	4=x	内部状态	
	3=x	内部状态	
	2=x 1=x	字符输入模式	
		第 2 位	第 1 位
		0	0=字母数字
		0	1=KataKana
		1	0=Hiragana
		1	1=未使用
	0=0	一半大小	
	0=1	全部大小	

中断	功能	描述	平台
16h	14h	shift 状态显示控制	DOS/V

这个功能控制 shift 状态行，该状态显示在显示屏的最下一行。Shift 状态行显示了当前

的字符输入模式以及其他的相关信息。字体结束处理器（FEP）负责处理这个功能，并使用中断 10h，功能 19h 来控制显示日文 DBCS 换档信息。

AL=	描述
0	开放 shift 状态显示
1	禁止 shift 状态显示
2	获取 shift 状态显示样的状态

子功能	描述	中断	功能
AL=0	开放 shift 状态显示	16h	AH=14h

如果安装了 FEP，就指导它在显示屏的底部显示 shift 状态。

调用：　　ax=1400h
返回：　　无

子功能	描述	中断	功能
AL=1	禁止 Shift 状态显示	16h	AH=14h

如果安装了 FEP，就指导它在显示屏的底部关闭 shif 的状态显示。
调用：　　ax=1401h
返回：　　无

子功能	描述	中断	功能
AL=0	禁止 Shift 状态显示	16h	AH=14h

如果安装了 FEP，就指导它在显示屏的底部关闭 Shift 的状态显示。
调用：ax=1401h
返回：无

子功能	描述	中断	功能
AL=2	获取 shift 状态显示栏的状态	16h	AH=14h

从 FEP 中获取 shift 状态显示栏的当前状态。

调用：　　ax=1402h
返回：　　al=0　　开放 shift 状态显示
　　1　　禁止 shift 状态显示

键盘 BIOS 数据区

低级键盘 BIOS（中断 9）和中间键盘 BIOS（中断 16h）使用大量的 BIOS 内的区域来保存标志和按键信息。所有的 BIOS 数据区已经在第 6 章中讨论过了，但是这里也讨论了专供键盘系统使用的数据。

地址	描述	大小
40:17h	键盘标志，换档与切换状态	字节

中断 9 可以获得换档和切换键的标志位。

位		
	7=1	Insert 开
	6=1	Caps Lock 开
	5=1	Num Lock 开
	4=1	Scroll Lock 开
	3=1	Alt 键按下
	2=1	Control 键按下
	1=1	左边 shift 键按下
	0=1	右边 shift 键按下

地址	描述	大小
40:18h	键盘标志，当前的换档状态	字节

中断 9 在本字节中保存了切换与换档键位置的当前状态。

位		
	7=1	Insert 键按下
	6=1	Caps Lock 键按下
	5=1	Num Lock 键按下
	4=1	Scroll Lock 键按下
	3=1	Pause 激活
	2=1	Sys Req 键按下
	1=1	左 Alt 键按下
	0=1	左 Ctrl 键按下

地址	描述	大小
40:19h	Alt 数字入口暂存区	字节

BIOS 使用这个工作区域。如果按下 Alt 键和数字键来输入一个十进制数，该字节中就保存输入的数字。

地址	描述	大小
40:1Ah	键盘头指针	字

键盘头指针指的是键盘缓冲区中第一个键对段址 40h 的偏移量。如果键盘头指针等于其尾指针，则表明缓冲区没有键。

地址	描述	大小
40:1Ch	键盘尾指针	字

键盘尾指针指的是键盘缓冲区中下一个输入键值将会放入的地址对段址 40h 的偏移量。如果键盘指针等于其尾指针，则表明缓冲区中没有键。

地址	描述	大小
40:1Eh	键盘缓冲区	16 字

键盘缓冲区是一个 16 字的 FIFO 区域，它可以容纳多达 15 个来自中断 9 的扫描码/SACII 值。中间键盘 BIOS，中断 16，从这个缓冲区读取数据。头和尾指针各指向第一个和最后一个入口。

地址	描述	大小
40:71h	Control-Break 标志	字节

如果按下了 Ctrl-Break 键，第 7 位就等于 1。该位由低级中断 9 的键盘程序设置。

地址	描述	大小
40:80h	键盘缓冲区起点	字

该键盘缓冲区起点，保存了键盘缓冲区的开始点在段 40h 内的偏移量（仅 AT+）。

地址	描述	大小
40:82h	键盘缓冲区终点	字

该键盘缓中区终点保存了键盘缓冲区末端在段 40h 内的偏移量（仅 AT+）。

地址	描述	大小
40:96h	键盘状态	字节

该字节保存了供中断 9h 使用的键盘状态信息（仅 AT+）。

位		
	7=1	读键盘 ID
	6=1	上次代码是第一个 ID 字符
	5=1	强制 Num Lock
	4=1	101+键键盘
	3=1	右 Alt 键按下
	2=1	右 Ctrl 键按下
	1=1	上次扫描码是 E0h
	0=1	上次扫描码是 E1h

地址	描述	大小
40:97h	键盘内部标志	字节

该字节为中断 9 对键盘控制器的操作保存了键盘的标志以及当前键盘的 LED 状态（仅 AT+）。

位		
	7=1	键盘传送错误
	6=1	LED 正在更新
	5=1	键盘发送一条“重新发送”命令
	4=1	键盘发送一条“确认”命令
	3=1	未使用
	2=1	Caps Lock 的 LED 开
	1=1	Num Lock 的 LED 开
	0=1	Scroll Lock 的 LED 开

热键及访问未定义键

有必要让 TSR 在普通 BIOS 键盘服务程序之前访问某些键，比如热键组合。应用程序也需要检测出 BIOS 键盘服务程序所不支持的组合键。例如，某个应用程序可能需要截获 Piont-Screen 键来服务其他操作，例如在屏幕数据之前发送一条打印机设置信息。

为了在 AT 以前的机器上实现这一点，有必要替换标准的键盘 BIOS 中断 9 服务程序。不幸的是，仅仅只有一个程序可以替换键盘 BIOS 处理程序。还没有办法输入键时将该信息传递给另外一个键盘处理程序。所以当两个或两个以上的程序试图直接访问键盘输入流时，会产生许多冲突。

对于 AT 系统，IBM 的工程师增加了一种功能可以挂起键盘数据流。在 BIOS 程序读取了一个扫描码后，BIOS 立即调用中断 15h 功能 AH=4Fh。AL 寄存器中保存读取的扫描码，并设置进位标志位表示该键有效。这样，多个应用程序可以截取中断 15h 并找寻功能 4Fh。如果按下了组合键，应用程序将执行下面三个动作中的一个：

1. 忽略该键，并将控制传给被挂住的应用程序，然后回到 BIOS 键盘处理程序继续执

行。这种情况下，进位标志为设置状态。

2. 移走该键，就好像按下了某键一样。这对于 TSR 热键很有用。TSR 记录按下的热键，这样可以稍后采取相应措施。TSR 清除进位标志，以通知其他截取中断 15h 的应用程序和 BIOS 键盘处理程序，不再有键需要处理。BIOS 仍负责清除中断请求。
3. 用一个不同的键盘扫描码替代。这种情况下，AL 中的值发生了改变，但是进位标志仍保持设置状态。任何一个后来截取中断 15h 应用程序和 BIOS 键盘处理程序只会看到新的键。记住，改变后的键必须是其扫描码而不是 ASCII。

扫描码

许多技术手册看起来好像带有扫描码表，但是我发现很难使用它们，并且也缺乏细节。下面的表包括了某些其他人都会有的信息。但是我加入了有关从键盘到主板控制器的键盘扫描码的详细列表，这些键盘扫描码由 AT+键盘提供。这些键盘扫描码不同于那些系统扫描码，后者由主板传递给低级 BIOS 键盘处理程序中断 9。我认为你会发觉这张表组织得较好，便于快速查找。

可以关闭主板控制器所执行的翻译操作。主板控制器的命令字节，第 6 位，控制是否将键盘扫描码翻译成系统扫描码（参看端口 64h）。这一点对于你检测一些不常见的组合键，例如 shift-F1-Q，很有用。因为这些组合键通常会被主板控制器过滤掉。MCA 有几种可任选的键盘扫描码集。参看端口 60h。命令 F0，获取更多细节。MCA 系统缺省的是扫描码集 2。它同表 8-5 所示的键盘扫描码相同。

只显示了那些键盘“产生”的代码，这些代码出现在按键时，如果释放了一个键，通常会在键盘扫描码前插入字节 F0h。例如，按下字母“B”时，就会发送键盘扫描码 F0 和 32h。如果该键之前还有一个扩展功能字节 E0，那么释放代码就是扩展功能字节 E0h，然后是 F0h，最后是键盘扫描码。例如按下右 Ctrl 键时，两字节码是 E0、14，而在释放时，键盘扫描字节是 E0、F0、14。

主板控制器将释放码翻译成系统扫描码，并将第 7 位置 1。低级 BIOS 程序中断 9 会过滤掉释放扫描码，也不将它放入到 BIOS 键盘缓冲区。

表 8-5 键盘码

U.S.Key	Keybd	Kscan	Scan	——— ah/al from int 16h (scan/ASCII) ———					
Legend	Size	Code	Code	No-Shift		Shifted		Control	Alt
A		1C	1E	1E/61	a	1E/41	A	1E/01	1E/00
B		32	30	30/62	b	30/42	B	30/02	30/00
C		21	2E	2E/63	c	2E/43	C	2E/03	3E/00
D		23	20	20/64	d	20/44	D	20/04	20/00

续表

U.S.Key	Keybd	Kscan	Scan	ah/al from int 16h (scan/ASCII)					
Legend	Size	Code	Code	No-Shift		Shifted		Control	Alt
E		24	12	12/65	e	12/45	E	12/05	12/00
F		2B	21	21/66	f	21/46	F	21/06	21/00
G		34	22	22/67	g	22/47	G	22/07	22/00
H		33	23	23/68	h	23/48	H	23/08	23/00
I		43	17	17/69	i	17/49	I	17/09	17/00
J		3B	24	24/6A	j	24/4A	J	24/0A	24/00
K		42	25	25/6B	k	25/4B	K	25/0B	25/00
L		4B	26	26/6C	l	26/4C	L	26/0C	26/00
M		3A	32	32/6D	m	32/4D	M	32/0D	32/00
N		31	31	31/6E	n	31/4E	N	31/0E	31/00
O		44	18	18/6F	o	18/4F	O	18/0F	18/00
P		4D	19	19/70	p	19/50	P	19/10	19/00
Q		15	10	10/71	q	10/51	Q	10/11	10/00
R		2D	13	13/72	r	13/52	R	13/12	13/00
S		1B	1F	1F/73	s	1F/53	S	1F/13	1F/00
T		2C	14	14/74	t	14/54	T	14/14	14/00
U		3C	16	16/75	u	16/55	U	16/15	16/00
V		2A	2F	2F/76	v	2F/56	V	2F/16	2F/00
W		1D	11	11/77	w	11/57	W	11/17	11/00
X		22	2D	2D/78	x	2D/58	X	2D/18	2D/00
Y		35	15	15/79	y	15/59	Y	15/19	15/00
Z		1A	2C	2C/7A	z	2C/5A	Z	2C/1A	2C/00
0)		45	0B	0B/30	0	0B/29	)		81/00
1!		16	02	02/31	1	02/21	!		78/00
2@		1E	03	03/32	2	03/40	@	03/00	79/00
3#		26	04	04/33	3	04/23	#		7A/00
4$		25	05	05/34	4	05/24	$		7B/00
5%		2E	06	06/35	5	06/25	%		7C/00

续表

U.S.Key	Keybd	Kscan	Scan	ah/al from int 16h (scan/ASCII)					
Legend	Size	Code	Code	No-Shift		Shifted		Control	Alt
6^		36	07	07/36	6	07/5E	^	07/1E	7D/00
7&		3D	08	08/37	7	08/26	&		7E/00
8*		3E	09	09/38	8	09/2A	*		7F/00
9(		46	0A	0A/39	9	0A/28	(		80/00
1 End		69	4F	4F/00		4F/31	1	75/00	###
2 Down		72	50	50/00		50/32	2	**91/00**	###
3 PgDn		7A	51	51/00		51/33	3	76/00	###
4 Left		6B	4B	4B/00		4B/34	4	73/00	###
5 (center)		73	4C	4C/00		4C/35	5	**8F/00**	###
6 Right		74	4D	4D/00		4D/36	6	74/00	###
7 Home		6C	47	47/00		47/37	7	77/00	###
8 Up		75	48	48/00		48/38	8	**8D/00**	###
9 PgUp		7D	49	49/00		49/39	9	84/00	###
0 Insert		70	52	52/00		52/30	0	**92/00**	###
'"		52	28	28/17	'	28/22	"		**28/00**
,<		41	33	33/2C	,	33/3C	<		**33/00**
-_		4E	0C	0C/2D	-	0C/5F	_	0C/1F	82/00
.>		49	34	34/2E	.	34/2E	>		**34/00**
/?		4A	35	35/2F	/	35/3F	?		**35/00**
;:		4C	27	27/3B	;	27/3A	:		**27/00**
=+		55	0D	0D/3D	=	0D/2B	+		83/00
[{		54	1A	1A/5B	[	1A/7B	{	1A/1B	**1A/00**
\\|	102-	5D	2B	2B/5C	\	2B/7C	|	2B/1C	**2B/00**
]}		5B	1B	1B/5D	[	1B/7D	}	1B/1D	**1B/00**
`~		OE	29	29/60	`	29/7E	~		29/00
BackSpace		66	0E	0E/08		0E/08		0E/7F	**0E/00**
Tab		0D	0F	0F/09		0F/00		**94/00**	**A5/00**

续表

U.S.Key	Keybd	Kscan	Scan	ab/al from int 16h (scan/ASCII)					
Legend	Size	Code	Code	No-Shift		Shifted		Control	Alt
Enter-main		5A	1C	1C/0D		1C/0D		1C/0A	1C/00
Space		29	39	39/20		39/20		39/20	39/20
Del.		71	53	53/00		53/2E		93/00	
Gray-		7B	4A	4A/2D	-	4A/2D	-	8E/00	4A/00
Gray+		79	4E	4E/2B	+	4E/2B	+	90/00	4E/00
Enter		5A	1C	1C/0D		1C/0D		1C/0A	1C/00
Escape		76	01	01/1B		01/1B		01/1B	01/00
* PrintScrn	83/84	7C	37	37/2A	*			96/00	37/00
F1		05	3B	3B/00		54/00		5E/00	68/00
F2		06	3C	3C/00		55/00		5F/00	69/00
F3		04	3D	3D/00		56/00		60/00	6A/00
F4		0C	3E	3E/00		57/00		61/00	6B/00
F5		03	3F	3F/00		58/00		62/00	6C/00
F6		0B	40	40/00		59/00		63/00	6D/00
F7		83	41	41/00		5A/00		64/00	6E/00
F8		0A	42	42/00		5B/00		65/00	6F/00
F9		01	43	43/00		5C/00		66/00	70/00
F10		09	44	44/00		5D/00		67/00	71/00
F11	101	78	57	85/00		87/00		89/00	8B/00
F12	101	07	58	86/00		88/00		8A/00	8C/00
Enter-num	101	E0,5A	E0/1C	E0/0D		E0/0D		E0/0A	A6/00
				1C/0D		1C/0D		1C/0A	
Gray/	101	E0,4A[2]	E0,35	E0/2F	/	E0/2F	/	95/00	A4/00
				35/2F	/	36/2F	/		
Gray*	101	7C	37	37/2A	*	37/2A		96/00	37/00
Gray End	101	E0,69[1]	E0,4F	4F/E0		4F/E0		75/E0	9F/00
Gray Down	101	E0,72[1]	E0,50	50/E0		50/E0		91/E0	A0/00
Gray PgDn	101	E0,7A[1]	E0,51	51/E0		51/E0		76/E0	A1/00

续表

U.S.Key	Keybd	Kscan	Scan	ah/al from int 16h (scan/ASCII)					
Legend	Size	Code	Code	No-Shift		Shifted		Control	Alt
Gray Left	101	E0,6B[1]	E0,4B	4B/E0		4B/E0		73/E0	9B/00
Gray Right	101	E0,74[1]	E0,4D	4D/E0		4D/E0		74/E0	9D/00
Gray Home	101	E0,6C[1]	E0,47	47/E0		47/E0		77/E0	97/00
Gray Up	101	E0,75[1]	E0,48	48/E0		48/E0		8D/E0	98/00
Gray PgUp	101	E0,7D[1]	E0,49	49/00		49/E0		84/E0	99/00
Gray Ins	101	E0,70[1]	E0,52	52/E0		52/E0		92/E0	A2/00
Gray Del	101	E0,71[1]	E0,53	53/E0		53/E0		93/E0	A3/00
L-Ctrl		14	1D						
L-Shift		12	2A						
R-Shift		59	36						
CapsLock		58	3A						
NumLock		77	45						
ScrollLock		7E	46						
L-Alt		11	38						
R-Alt	101	E0,11	E0,38						
R-Ctrl	101	E0,14	E0,1D						
PrintScrn[3]	101	E0,12,	E0,2A,					72/00	
		E0,7C	E0,37						
Pause	101	E0,14	E1,1D,						
		77,E1[4]	E1,45						
SysReq	84+	84	54						
keya	102	5D							
keyb	102	61							
left window	104	E0,1F	E0,5B						
right window	104	E0,,27	E0,5C						
list bit	104	30,2F	E0,5D						

注：所有的值都是十六进制的

“xx,xx”表示多个字节组成了该键的扫描码。第一个字节E0h表示必须读取第二个字节才能确定该键。

“xx/xx”代表了保存在BIOS键盘缓冲区中的ah/al值对。中断16h的功能0,1,10h和11h将扫描码返回到ah中，而将ASCII返回到al中（如果该键有对应的ASCII）。仅限于功能0和1，如果表中al的值是

E0，那么实际的返回值是 00。

灰色阴影扫描码表示中断 16h 的功能 0 和 1 中不返回任何东西。只在功能 10h 和 11h 中返回灰色所示的值。

###如果输入了 Alt 和 1 到 3 个数字的组合，那么该十进制数，0 到 255，就会转化成相应的 ASCII 字符。

键盘大小：

空白 =适用于所有键盘

83/84 =只适用于 83/84 键键盘

84+ =适用于 84，101，102 和 104 键键盘（不适用于 83 键键盘）

101+ =只适用于 101，102 和 104 键键盘

102 - =不适用于 102 键键盘

102 =只适用于 102 键键盘（非美国键盘）

104 =Windows95 类型键盘，在 101 键键盘在另加入了三个键

[1]在 101+键盘上，这些键应采取指示的行动而不论 shift 键和 Num Lock 键的状态如何。为了做到这一点，并保持与 84 键键盘的键盘扫描码兼容，另外会发送一些与 shift 和 Num Lock 状态相关的键盘扫描码。如果 shift 或 Num Lock 开着（但不同时），那么发送字节就好像激活该键时 shift/ Num Lock 并未打开一样。这样做是为了愚弄主板控制器以完成指定的功能。（记住，老式的 84 键键盘带有这些数字键区功能，激活它们与 Num Lock 和 shift 有关）。下面的例子显示了灰色 HOME 键的键盘，E0，6C。最终的结果是主板控制器给出了 HOME 扫描码功能，而并未考虑 shift 和 Num Lock 的状态。请注意 shift 键的键盘扫描码是 12。

Shift	Num-Lock	生成的键盘扫描码	释放键盘扫描码
关闭	关闭	E0,6C	E0,F0,6C
关闭	开启	E0,12,E0,6C	E0,F0,6C,E0,F0,12
按下	开关	E0,F0,12,E0,6C	E0,F0,6C,E0,12
按下	开启	E0,6C	E0,F0,6C

[2]在 101+键键盘上，将依据键盘 shift 的状态修改键盘扫描码。如果按下了 shift，那么发送字节就好像激活该键时 shift 并未按下一样。这样做是为愚弄主板控制器来总是生成“/”键功能，否则它会像 84 键键盘那样生成一个“\”键功能。请注意 shift 键的键盘扫描码是 12。

Shift	生成的键盘扫描	释放键盘扫描码
关闭	E0,4A	E0,F0,4A
按下	E0,F0,12,E0,4A	E0,F0,4A,E0,12

[3]在 101+键键盘上，无论 shift 状态如何，Print Scrn 键一定会激活屏幕打印功能。如果同时还按下了 Alt 键，那么就会触发 Sys-Req 功能。由于必须保持屏幕打印和 84 键键盘兼容，所以键盘会发送额外的键

盘扫描码来显得就像按下了屏幕打印或 Sys-Req 键一样。Shift 的键盘扫描码是 12，Print Scrn 键的键盘扫描码是 7C，而 Sys-Req 的键盘扫描码是 84。

Shift	Ctrl	Alt	生成的键盘扫描码	释放键盘扫描码	功能
关闭	关闭	关闭	E0,12,E0,7C	E0,F0,7C,E0,F0,12	Prt Sert
关闭	关闭	按下	84	F0,84	Sys Req
关闭	按下	关闭	E0,7C	E0,F0,7C	Prt Sert
按下	关闭	关闭	E0,7C	E0,F0,7C	Prt Sert

[4] 在 101+键键盘上，Ctrl-Panse 键应该模拟 84 键键盘的 Ctrl-Break 功能。如激活 Ctrl 的同时按下了 Pause，键盘会生成一条“Ctrl-Break”代码。注意，该键没有释放码。

Ctrl	生成的键盘扫描码	功能
关闭	E1,14,77,E1,F0,14,F0,77	Pause
按下	E0, 7E,E0,F0, 7E	Pause

国外的键盘

国外的键盘同表 8-5 中显示的美国键盘只有一些键有点不同。表 8-6 和表 8-7 只显示了那些不同的键。键盘扫描码将同美国键盘扫描码相同，但是系统扫描码会有所不同，这取决于键盘的功能以及所使用的 BIOS 处理程序。

表 8-6 84 键的国外键盘

U.S.key	French	German	Ltallan	Spanish	U.K.
Legend	Legend	Legend	Legend	Legend	Legend
A	0				
M	,?				
Q	A	Z			
W	Z	Y			
Y					
Z	W				
0)	à0	0=	0=	1i	
1!	&1			2¿	2"
2@	é2	2"	2"		3£

续表

U.S.key	French	German	Ltalian	Spanish	U.K.
Legend	Legend	Legend	Legend	Legend	Legend
3#	"3	3 §	3£		
4$	'4				
5%	(5				
6^	§ 6	6&	6&	6/	
7&	è7	7/	7/		
8*	!8	8(	8(		
9(	ç 9	9)	9)		
' "	`%	Ä	`a#	;:	'@
,<	;.	.:	.;	,?	
-_	)°	B?	' ?		
.>	:/	.:	.:	.!	
/ ?	=+	-_	-_	' "	
;:	M	Ö	ò@	Ñ	
=+	-_	^	ì^		
[{	^ ¨	ü	èé	' "	
|	µ£	#^	ù §	ç	
]}	$*	+*	+*	`^	#~
`~	<>	<>	<>	<>	\ |

表 8-7　102 键外国键盘

美国联想	比利时	加拿大（法国）	丹麦	荷兰	法国	德国	意大利	拉丁美洲	挪威	葡萄牙	西班牙	瑞典	瑞士	英国
A	Q				Q									
C				C										
M	?	Mµ		Mµ	,?	Mµ								
O		§ O												
P		P												
Q	A				A	Q@								

续表

美国联想	比利时	加拿大（法国）	丹麦	荷兰	法国	德国	意大利	拉丁美洲	挪威	葡萄牙	西班牙	瑞典	瑞士	英国
S				S										
W	Z					Z								
X				X										
Y													Z	
Z	W			Z	W	Y							Y	
0)	à}0	0)	0}=	0	à@0	0}=	0=	0=	0}=	0}=	0=	0}=	0=	
1!	&|1	1+!		1}!	&1					1}!			1!+	
2@	é@2	2@"	2@"	2"	é~2	2"	2"	2"	2@"	2@"	2@"	2@"	2@"	2"
3#	"#3	3/	3#	3#	"#3	3§	3		3#	3#	3·	3#	3#*	3
4	'4	4$	4$	4$	{4				4$¤	4$§		§4$¤	4ç	
5%	(5	5¤%	5¤%		5%	([15								
6^	§¯6	6?	6&	6&	16	6&	6&	6&	6&	6&	6&	(2&)	6&	
7&	È7	è7	7|&	7{/	7_	è´7	7{/	7/	7/	7{/	7(/	7{/	7/	
8*	!8	!8	8^	8[(	8[(	-/8	8[(	8(	8[(	8[(	8(	8[(	8[(	
9(	{9	{9	93)	9])	^9	9])								
'	ù%	´¦´	0	´	ù%		à#°	{^[	Æ	ºª	´{"	Ä&	äà	@
,<	;	_	, .	, .	, .	, .	, .	, .	, .	, .	, .	, .	, .	
-_	)	-_	+?	/\?	)]"	\?	?	\?	+?	?	?	+\?	`?	
.>	:/	._	. :	.•:	:/	. :	. :	. :	. :	. :	. :	. :	. :	
/?	=~+	´	_	=	!§	_	_	_	_	-	_	_		
;:	M	;~:		+±	M		ò@ç	ò	O	ç	ò	ö	öéö	
=+	-_	=+	|´	¨~	}+	´	•^		|`	«»		`	¨~`	
[{	^["	^|^		"^	¨		è.é	º	Å	+"*	`[^	Å	üéü	
]}	$]*	"	"~^	*|	$¤	+~*	+]*	~*	"~^	´´	+]*	"~^	"!	
`~	23	#\\|	§	@§		^°	\\|	|¬°	|§	\\|	<>	§	§°	
键 a	µ´	<|>	'*	<>	*µ	#'	ù§	}`]	,*	~^	}ç	,*	$£	#~
键 b	<|>		<|>	||	<>	<|>	<>	<>	<>	<>	<>	<|>	<|>	\\|

注：有 3 个图注的键表示在按下 AltCar 键时会出现第 2 个图注（在 BIOS 控制下）

有 4 个图注的键（仅瑞士键盘）表示在按下 AltCar 键时会出现第 2 个图注，而在按下 shift-AltCar

时出现第 4 个图注。AltCar 和美国键盘上 Alt-R 相同。

美国键盘上不存在键 a 和键 b。其他国家的键盘上也没有美国键盘上的“\\”键。

扩展内存的 A20 访问

在所有的 AT 以及后来的系统上，键盘控制器提供了锁住 A20 地址线的功能，以保持和 8088 兼容。有必要提供一些背景来帮助理解 A20 封锁。

1MB 段的环绕返转

与 8088 的 1MB 最大内存地址范围相比，286 以及所有后来的 CPU 提供了 16MB 或更多的内存地址范围。8088 古怪的寻址方案允许程序使用任何一个越过 1MB 限制的段：偏移量对来访问最低的 64KB 区域。例如，段偏移量 FFFF:50 将访问 0:40 处的字节。尽管没有理由让软件去使用这种古怪的方案，但是那些使用段：偏移量越过 1MB 界限的老式程序仍会有不少 bug。因为这些程序似乎能正确地工作，因此在过去也没有采取措施来修正这些缺陷。

286 提供了 16MB 的寻址能力，在访问内存时又产生了新的古怪之处，它使用段：偏移量对来访问内存也会在 8088 上造成环绕返转。而在 286 上，CPU 将访问 1MB 以上的 64KB。

表 8-8　1MB 边界上的 CPU 内存访问

段：偏移量	8088 物理地址	80286 物理地址	注　　解
FFFF:0	0FFFF0h	0FFFF0h	物理地址相同
FFFF:10	0	100000h	相差 1M
FFFF:FFFF	0FFEFh	10FFEFh	相差 1M,64KB 块尾的 16 字节

A20 门

为了使 80286 系统的寻址看起来和 8088 一样，IBM 在 80286 加入了一个内部硬件将 A20 地址强制为 0。该硬件被称作 A20 门。当 A20 门处于缺省状态时，访问 1MB 处的内存实际上将访问第一个 64K 的内存，就像 8088 一样。如果使用表 8-8 的例子时将 A20 强制为零，结果如表 8-9 所示。

表 8-9　1MB 边界上的系统内存访问（A20 为零）

段：偏移量	8088 物理地址	80286 物理地址	注　　解
FFFF:0	0FFFF0	0FFFF0	物理地址相同
FFFF:10	0	0	物理地址相同
FFFF:FFFF	0FFEFh	0FFEFh	物理地址相同

为了支持访问 1MB 以上的内存，必须开放 A20 门。这样就可以传递 CPU 的地址线 20 的状态而不改变它。这一点允许在所有的 CPU 模式，实模式和保护模式下访问 1MB 以上的 64KB（少 16 字节）的内存空间。为了访问剩下的扩展内存，有必要进入保护模式，或者使用我所讨论过的决窍——使用未公开的 LOADALL 指令。

1MB 区域以上的 64K 区域被称作高端内存区（HMA）。最新版本的 DOS 可以将其部分装入 HMA 以为低端 640K 节省空间。为了使 HMA 发挥作用，必须开放 A20 线。

A20 状态

尽管使用键盘控制器命令 D0h 可以检查 A20 门的状态，但是一些系统还是不能遵循 IBM 标准。另外，从键盘控制器读取状态需要让大量的代码服从适当的协议。本章详细介绍了如何使用键盘控制器命令和命令 D0h。

我创建了一个代码段，如代码 8-1 所示，作为获取当前 A20 状态的一种可选的手段。它可用于任何系统。为了确定 A20 的状态，这个程序首先比较地址 0:0 和 FFFF:10h 处的字。如果值不相同，那么 A20 是开放的。如果值相同，这个程序会暂时求反地址 0:0 处的字，然后把它同地址 FFFF:10h 处的值进行比较。如果两个字相同，那么 A20 被认为是禁止的，同时这个程序从 AX 中返回值 0。如果 A20 是开放的，那么这两个字的值会不相同，并在 AX 中返回 1。

代码 8-1 获取当前的 A20 状态

```
;       获得 A20 的状态
;               调用参数                空
;
;               返回:                   ax=0        A20 禁止
;                                         =1        A20 启动
;                                       中断启动
;
;               用到的寄存器:           ax
getA20          proc        near
                push        cx
                push        di
                push        si
                push        ds
                push        es
```

```
        mov     di,10h
        mov     ax,0FFFh
        mov     es,ax                   ; es:di=FFFF:10h
        xov     ax,ax                   ; 默认      ax=0
        mov     si,ax
        mov     ds,ax                   ; ds:si=0:0
        mov     cs,ax

        cli                             ; 中断禁止
        mov     ax,es:[di]              ; 在地址 FFFF:10h 处取值
        cmp     ax,[si]                 ; 与 0:0 处的值一致?
        je      getA20_skip1            ; 如果状态未知就跳转
        jnc     cx                      ; A20 启动（比较失败）
        jmp     getA20_exit

getA20_skip1:
        not     word ptr ds:[si]        ; 转化为字
        mov     ax,es:[di]              ; 在地址 FFFF:10h 处取值
                                        ; 与 0:0 处的值一致?
        cmp     ax,[si]                 ; 如果状态未知就跳转
        je      getA20_skip2            ; A20 启动（比较失败）
        inc     cx
getA20_skip2:
        not     word ptr ds:[si]        ; 恢复字为初始值
getA20_exit:
        sti                             ; 启动字为初始值
        mov     ax,cx                   ; 用 ax 返回状态
        pop     es
        pop     ds
        pop     si
        pop     di
        pop     cx
getA20  endp
```

A20 门控制

通常只有很少的几个程序会去考虑 A20 门控制问题。可以处理 A20 门的四组程序

包括 BIOS、某些高级的操作系统（比如 NT 和 OS/2）、内存管理程序以及 DOS 扩充器。

如果你需要进入保护模式，我建议用第 13 章中描述的 BIOS 服务中断 15h 的功能 89h。作为这个服务的一部分，它对 A20 进行了处理来访问整个扩展内存。

99.9%的 PC 系统都使用键盘控制器来处理 A20 门。这似乎有点不可思议，但是键盘控制器的确有一些额外输出线，键盘操作不需要使用这些输出线。余下的一小部分系统不使用键盘控制器来处理 A20 门，由于它们和 IBM 并不完全兼容，目前已经被认为是过时之物了。在本章的其他部分，我不会考虑这些系统。

通常使用一个命令来控制 A20 门，这个命令是键盘控制器命令 D1h。命令 D1h 控制着 A20、重启、键盘数据以及其他一些与供应商无关的选项。为了避免和其他系统功能发生冲突，你必须首先使用命令 D0h 来读取当前的状态。有了当前的状态，你才可以在写回一个新的值之前改变 A20 门位。代码 8-2 显示了这个过程。它使用了在本章后面描述的键盘控制器子程序。

代码 8-2 A20 控制

```
;       Set A20 Status
;
;           Called with:    ah = 0 to disable A20
;                             1 to enable A20
;
;           Returns:        A20 state changed
;
;           Regs Used:      cx

setA20      proc    near
            push    ax
            cli                             ; 禁止中断

; get the keyboard controller output status

            mov     bl,0D0h                 ; 读端口命令
            call    keyboard_cmd            ; 激活
            call    error_cmd               ; 处理错误
            call    keyboard_read           ; 向 AL 读
            mov     bl,al
```

```
        pop     ax                      ; 反回设置
        mov     al,bl                   ; ah = 键盘控制值
        push    ax

; now write the old output status with the new A20 state

        mov     bl,0D1h                 ; 改变控制端口
        call    keyboard_cmd            ; 激活
        call    error_cmd               ; 处理错误
        pop     ax                      ; ah = 新状态（0 或 1）
                                        ; al = 控制器值
        cmp     ah, 0                   ; A20 可用吗?
        je      setA20_disable          ; 跳转至 so
        or      al,2                    ; 启用 A20
        jmp     setA20_skip

setA20_disable:
        and     al,0FDh                 ; 禁用 A20
setA20_skip:
        call    keyboard_write          ; 向键盘传输命令
        call    error_write             ; 处理错误
        sti
        ret
setA20  endp
```

些系统上提供了两个可交替的键盘控制器命令。命令 DDh 禁止 A20，而命令 DF 开放 A20。使用这些命令比较简单也比较快，但是大多数的供应商并不支持这些特征。

由于 A20 控制在所有的系统上不尽相同，改变其状态后核对一下是否选择了真正的 A20 状态不失为明智之举。（参看代码 8-1）

警　告

在从端口 60h 读取任何信息之前，必须检查控制器输出缓冲区状态，以确定可以读取一个字节。读取端口 64h 获取状态并检查第 0 位是 1。如果第 0 位是 0，那么没有有效的信息可以读取。带有 1 型控制器的 MCA 系统（下面将讨论）在第 2 位从 0 变成 1 后必须等待至少 7 毫秒才能从端口 60h 读取数据。Keyboard-read 子程序显示了实现这一点的代码。

在向端口 60h 写任何信息之前，控制器输入缓冲区必须为空。读取端口 64h 以获取状态并检查第 1 位是 0 。如果第 1 位是 1，那么控制器的输入缓冲区仍是满的，不能被写入。

当向键盘发送命令并紧接着向键盘发送数据字节时，必须在该命令之前禁止控制器。下面的节选代码显示了如何设置重复键入速度。记住，你不能单步执行该代码，因为这时禁止了键盘。

```
mov     bl,  0ADh
call    keyboard_cmd            ;禁止键盘（AD）
mov     al,  0F3h
call    keyboard_write          ;键入速率命令（F3）
mov     al,0
call    keyboard_write          ;设置最快的键入速率
mov     bl,  0AEh
call    keyboard_cmd            ;开放键盘（AE）
```

MCA 系统有两种不同类型的主板控制器，1 型和 2 型。在操作和功能上它们有一点小差别。下面的代码段用于检测控制器类型，采用的方法是检查命令字节的翻译位是否能设置为 1。只有 1 型控制器支持这一点。

```
mov     bl,         20          ;获得命令字节功能
call    keyboard_cmd
call    keyboard_read
mov     ah,         al          ;保存命令字节
mov     bl,         60          ;设置命令字节功能
call    keyboard_cmd
mov     al,         ah          ;恢复命令字节
mov     al,         40          ;设置键盘翻译位
call    keyboard_write          ;写新命令
                                ;再次读取命令
mov     bl,         20          ;获得命令字节功能
call    keyboard_cmd
call    keyboard_read
mov     bh,         al          ;保存结果
mov     bl,         60h         ;设置命令字节功能
call    keyboard_cmd
mov     al,         ah          ;恢复命令字节
```

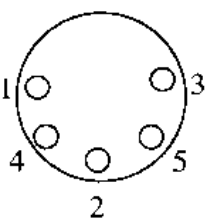
```
        call    keyboard_write          ;写初始命令字节
        and     bh,     0BFh            ;bh=0,如果是类型 2;
                                        ;bh=40h，如果是类型 1
```

键盘的连接和信号

在大多数 ISA/EISA 系统上，键盘通过一个 5 针的 DIN 连接器连接，尽管某些也使用 PS/2 的新型连接器。

针号	线
0	键盘时钟线
1	键盘串行数据
2	未用
3	地
4	电源（+5.0V DC）

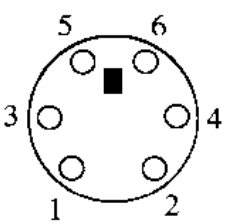
插件终端（从键盘向外看）



一种替代型键盘连接器，6 针的小型 DIN 连接器，主要用于大多数膝上型电脑以及较新的 PCI 系统和所有的 PC/1 及 PS/2 系统上。许多系统的鼠标也使用这种针式的连接器。

针号	线
0	串行数据
1	未用
2	地
3	电源（+5.0V DC）
4	时钟
5	未用

件终端（从键盘向外看）



信号与时钟线由一个开式连接驱动器驱动，同时还带有一个 10KΩ的上拉电阻器。这一点允许键盘和主板在需要的时候独自驱动时钟和数据线。该连接器对于监视带有一个逻辑分析器的串行连接器很有用。

代码例 8-3　控制器访问

下面四个程序处理基本键盘功能来读写字节以及向控制器发送命令。另外，SetLEDS 程序会装载键盘的三个 LED 状态。它们都是为 AT+系统而设计。

```
; ---------------------------------------------------------------------------
;    KEYBOARD_READ
;        从键盘向al读取一个字节（端口60h）
;
;        调用:              无
;
;        返回:              如果ah=0, 则al=读自键盘的字节
;                           如果ah=1, 超时后仍没有读入字节
;
;        用到的寄存器:      al

keyboard_read    proc    near
        push    cx
        push    dx

        xor     cx, cx                  ; 超时计数器 (64K)
key_read_loop:
        in      al, 64h                 ; 键盘控制器状态
        IODELAY
        test    al, 1                   ; 数据字节准备好?
        jnz     key_read_ready          ; 如果准备好读，则跳转
        loop    key_read_loop

        mov     ah, 1                   ; 返回状态—坏
        jmp     key_read_exit

key_read_ready:
        push    cx                      ; MCA类型1控制器所需要的延时
        mov     cx, 16                  ; 程序
key_read_delay:                         ; 确保不少于7μs
        IODELAY
        loop    key_read_delay          ; 假定CPU是80486
        pop     cx                      ; 66MHz 或更慢

        in      al, 60h                 ; 读字节
        IODELAY
```

```
        xor     ah, ah                  ; 返回状态—成功
key_read_exit:
        pop     dx
        pop     cx
        ret
keyboard_read   endp


; ---------------------------------------------------------------------------
;   KEYBOARD_WRITE
;       将字节AL发送到键盘控制器（端口60h）
;       假定没有激活BIOS中断9处理程序
;
;       如果由于缓冲区维持满状态超过定时，则
;       ah为非零值
;
;       调用:           al = 要发送的字节
;                       ds = cs
;
;       返回:           如果 ah = 0, 成功
;                       如果 ah = 1, 失败
;
;       用的寄存器:     ax

keyboard_write  proc    near
        push    cx
        push    dx
        mov     dl, al                  ; 保存键盘数据

        ; 等待，直到清除键盘接收超时（通常如此）

        xor     cx, cx                  ; 超时计数器
kbd_wrt_loop1:
        in      al, 64h                 ; 获取键盘状态
        IODELAY
```

```
        test    al, 20h                 ; 出现接收超时?
        jz      kbd_wrt_ok1             ; 如果没有，则跳转
        loop    kbd_wrt_loop1           ; 再试一次
                                        ; 失败
        mov     ah, 1                   ; 返回状态 - 失败
        jmp     kbd_wrt_exit

kbd_wrt_ok1:
        in      al, 60h                 ; 清除缓冲区

        ; 等待输入缓冲区清空（通常如此）

        xor     cx, cx                  ; 超时计数器（64K）
kbd_wrt_loop:
        in      al, 64h                 ; 获取键盘状态
        IODELAY
        test    al, 2                   ; 检查是否在作用缓冲区
        jz      kbd_wrt_ok              ; 如果没有作用，缓冲区
        loop    kbd_wrt_loop            ; 再试一次
                                        ; 失败，仍忙
        mov     ah, 1                   ; 返回状态—失败
        jmp     kbd_wrt_exit

        ; 写数据（临时保存在DL中）

kbd_wrt_ok:
        mov     al, dl
        out     60h, al                 ; 到控制器/键盘的数据
        IODELAY

        ; 等待，直到输入缓冲区清空（通常如此）

        xor     cx, cx                  ; 超时计数器 (64K)
kbd_wrt_loop3:
        in      al, 64h                 ; 获取键盘状态
        IODELAY
```

```
        test    al, 2                   ; 检查是否在使用缓冲区
        jz      kbd_wrt_ok3             ; 如果不在使用则跳转
        loop    kbd_wrt_loop3           ; 再试一次
                                        ; 失败，仍忙
        mov     ah, 1                   ; 返回状态—失败
        jmp     kbd_wrt_exit

        ; 等待，直到输出缓冲区清空

kbd_wrt_ok3:
        mov     ah, 8                   ; 更大的延时循环（8*64K）
kbd_wrt_loop4:
        xor     cx, cx                  ; 超时计时数器 (64K)
kbd_wrt_loop5:
        in      al, 64h                 ; 获取键盘状态
        IODELAY
        test    al, 1                   ; 检查是否正在使用缓冲区
        jnz     kbd_wrt_ok4             ; 如果没有作用则跳转
        loop    kbd_wrt_loop5           ; 再试一次
                                        ; 失败，仍忙
        dec     ah
        jnz     kbd_wrt_loop4
kbd_wrt_ok4:
        xor     ah, ah                  ; 返回状态，成功

kbd_wrt_exit:
        pop     dx
        pop     cx
        ret
keyboard_write  endp


; --------------------------------------------------------------------------
;   KEYBOARD_CMD
;       将寄存器BL中的命令发送到键盘控制器
;       (端口 64h).
```

```
;
;       如果由于缓冲区满超时，则
;       ah为非零
;
;       调用:           bl = 命令字节
;                       ds = cs
;
;       返回:           如果 ah = 0, 则成功
;                       如果 ah = 1, 则失败
;
;       Regs Used:      ax, cx


keyboard_cmd    proc    near
        xor     cx, cx                  ; 超时计数器（64K）
cmd_wait:
        in      al, 64h                 ; 获取控制器状态
        IODELAY
        test    al, 2                   ; 输入缓冲区满?
        jz      cmd_send                ; 准备接收命令?
        loop    cmd_wait                ; 否则跳转
                                        ; 失败，仍忙
        jmp     cmd_error

cmd_send:                               ; 发送命令字节
        mov     al, bl
        out     64h, al                 ; 发送命令
        IODELAY

        xor     cx, cx                  ; 超时计数器 (64K)
cmd_accept:
        in      al, 64h                 ; 获取控制器状态
        IODELAY
        test    al, 2                   ; 输入缓冲区满?
        jz      cmd_ok                  ; 如果接受命令则跳转
        loop    cmd_accept              ; 再试一次
```

```
                                        ; 失败，仍忙
cmd_error:
        mov     ah, 1                   ; 返回状态—失败
        jmp     cmd_exit
cmd_ok:
        xor     ah, ah                  ; 返回状态—成功
cmd_exit:
        ret
keyboard_cmd    endp

;-------------------------------------------------------------------------------
;   设置LED
;       从BL向键盘LED发送3位（调试
;       TSR很容易做到这一点）。不要更新
;       键盘标志位。下一次更新LED会恢复
;       这个LED状态。在小概率事件“正
;       在更新LED”发生时，会跳过更新命令。
;
;           40:97h = 键盘标志2
;                   位 7 = 1 如果出现键盘传输错误
;                   位 6 = 1 如果LED正在更新
;                   位 5 = 1 如果收到重新发送命令
;                   位 4 = 1 如果收到确认命令
;       本程序还假定标准的中断9 BIOS
;       键盘处理程序不会改变40:97h处
;       的键盘标志2的状态
;
;       调用:           bl: 位 0=1 开启 Scroll Lock LED
;                           1=1 开启 Num Lock LED
;                           2=1 开启 Caps Lock LED
;                           其他 3-7 位忽略
;
;       用到的寄存器:   ax

setLEDs proc    near
        push    es
```

```
        mov     ax, 40h
        mov     es, ax                  ; es 指向BIOS数据
        cli                             ; 更新时关闭中断
        test    byte ptr es:[97h], 40h  ; 已经更新?
        jnz     setLED_return3          ; 如果是，则返回

        or      byte ptr es:[97h], 40h  ; 设置正在更新标志
        mov     al, 0EDh                ; 更新LED命令
        call    keyboard_write          ; 发送键盘命令
        test    byte ptr es:[97h], 80h  ; xmit错误?
        jnz     setLED_return1          ; 如果是，则退出
        mov     al, bl
        and     al, 7                   ; 只发送3个LED位
        call    keyboard_write          ; 到键盘的LED数据
        test    byte ptr es:[97h        ; xmit错误?
        jz      setLED_return2          ; 如果是，则跳转
setLED_return1:
        mov     al, 0F4h                ; 开放键盘
        call    keyboard_write          ;  因为出错
setLED_return2:
        and     byte ptr es:[97h], 3Fh  ; 关闭和更新出错
setLED_return3:
        sti                             ; 开放中断
        pop     es
        ret
setLEDs endp
```

代码 8-4 系统重启

下面的代码段用来在所有的 AT+系统上强制重启系统，所采用的方法是向键盘发送一条命令来强制重启。在第 3 章中更深入地讨论了处理器的重启问题。

```
mov     bl,  0Feh           ;   重启系统命令
call    Keyboard-cmd        ;   不应该返回!
hlt
```

代码例 8-5 查看扫描码和键盘扫描码

KEYSCAN 显示了如何将本章的各种信息连接起来创建一个有用的程序。在 KEYSCAN 中，可以设置主板上控制器，来跳过对来自 AT 键盘的键盘扫描码所进行的翻译。当按下一个键时，会出现未改变的键盘扫描码。这些代码通常由 8042 主板控制器翻译成 XT 系统兼容的系统扫描码，并带有一些新建的扩展。为了查看键盘扫描码，可以使用命令行选项“K”。否则，只显示普通的扫描码。

该代码例可以完全控制键盘进程，它的操作不需要使用 BIOS 服务。KEYSCAN 也显示了来自键盘的键盘释放码。在键盘扫描模式下，键盘对于一个释放键会发送一个 FOW 字节，然后发送该键盘扫描码。某些工作在非翻译模式下的控制器会将两字节的释放码转化为单字节的键盘扫描字节，并将第 7 位置 1。按“Escape”键可以退出这个程序。

没有列出全部的程序，而只列出了其中主要的部分。你可以到中国电力出版社网站 www.infopower.com.cn 上下载完整的源代码。

```
;--------------------------------------------------------------------------
;                                KEYSCAN
;--------------------------------------------------------------------------
;
;    本程序演示了键盘和键盘控制器
;    的底层关系。它还显示了键盘和
;    控制器间的键盘扫描码
;    以及经控制翻译过并传送
;    给BIOS中断9处理程序的扫描码。
;
;    KEYSCAN执行下述操作：
;        放慢键盘重复速率到允许的最低值
;        关闭所有的键盘LED（仅仅显示
;          如何实现
;        显示实现翻译模式所写入的命令字节
;        显示按键操作时两种扫描码中的一种
;          缺省（无命令行选项）显示按键
;          时的常见扫描码（这正是通常
;          发送到中断9的那种）。使用命令
;          行参数-k来显示来自键盘
;          但控制器尚未翻译的键盘
;          扫描码。
```

```
;       按下Esc键或50键退出
;       恢复普通键盘操作
;       恢复LED状态
;       键盘重复速度设置为24cps，在重开始
;         之前延时500ms
;
;   所有的计算机都配有8042类型的
;   键盘控制器（所以PC/XT除外）
;
;   (c) 版权所有1994, 1996 Frank van Gilluwe
;   保留所有权力
;
;   V2.00 – 加入了键盘扫描码或扫描码选项
;           操作期间可以放慢键盘重复速率
;           退出时恢复键盘LED状态
;           按Escape键时显示键盘扫描码
;           如果按下50键并松开或Escape，则退出
;           支持Keyboard工作在最新式的PC上

include undocpc.inc


cseg    segment para public
        assume  cs:cseg, ds:cseg, ss:stacka

keyscan         proc near

message1 db     CR, LF
         db     'KEYBOARD INFORMATION AND SCAN CODES'
         db     '               v2.00 (c) 1994, 1996 FVG'
         db     CR, LF
         db     '- - - - - - - - - - - - -'
         db     '- - - - - - - - - - - - - - - - - - - - -'
         db     CR, LF
         db     '  To exit, press Escape', CR, LF, CR, LF, '$'
```

```
message2 db     '  Displays the normal scan code for each'
         db     ' key pressed.', CR, LF
         db     '  Use command line option -k to see the'
         db     ' untranslated scan codes.'
         db     CR, LF, CR, LF, '$'

message3 db     '  Displays the untranslated Kscan code for'
         db     ' each key pressed.', CR, LF, CR, LF, '$'

message4 db     'New keyboard command byte = '
commandb db     '  ', CR, LF, CR, LF, '$'

message5 db     'Returned scan code = '
scanbytN db     '  ', CR, LF, '$'

message6 db     'Returned Kscan code = '
scanbytK db     '  ', CR, LF, '$'

message7 db     'Keyboard read timeout error', CR, LF, '$'

message8 db     CR, LF
         db     'Complete - 50 keys pressed and released'
crlf     db     CR, LF, '$'

message9 db     CR, LF
         db     'Complete - Escape key pressed', CR, LF, '$'

cmd_line db     0                       ; 'K' 如果未翻译

start:
         xor    bl, bl
         cmp    byte ptr ds:[80h], 1    ; 命令行上有字符吗?
         jbe    no_options              ; 如果没有则跳转
         mov    bl, ds:[83h]            ; 获取命令行选项
no_options:
         mov    ax, cs
```

```
        mov     ds, ax
        mov     es, ax
        and     bl, 0DFh                ; 转化成大小
        mov     [cmd_line], bl          ; 保存选项
        OUTMSG  message1                ; 显示初始信息
        cmp     [cmd_line], 'K'         ; 是否翻译?
        je      no_translate
        OUTMSG  message2                ; 普通翻译
        jmp     slow_rate

no_translate:

        OUTMSG  message3                ; 显示键盘扫描码

; 尽管不必要，但我们仍会改变重复速率为
; 最小值，这样容易看到键的按下和
; 释放码。

slow_rate:
        mov     ax, 305h                ; 设置重复速率
        mov     bx, 031Fh               ; 使用 1 秒延时, 2 cps
        int     16h                     ; 执行

set_LEDs_on:
        mov     bl, 7                   ; 开启所有的LED
        call    setLEDs

; 发送命令来关闭键盘中断IRQ1
;   （位0=0）这样BIOS不会响应键盘。
;   如果选择了键盘扫描码模式，同时关闭翻译（位6=0)

        cli                             ; 禁止中断
        mov     bl, 60h                 ; 设置命令字节功能
        call    keyboard_cmd            ; 激活
        call    error_cmd               ; 显示是否出错 (ah=1)
        mov     al, 24h                 ; 关闭IRQ 1
```

```
                                        ;   (中断9)，无翻译
        cmp     [cmd_line], 'K'         ; 键盘扫描码模式?
        je      load_cmd                ; 如果是，则跳转
        mov     al, 64h                 ; 翻译
load_cmd:
        call    keyboard_write          ; 向键盘发送命令
        call    error_write             ; 如果出错，则显示

; 清空缓冲区中剩余的键

flush_all:
        in      al, 64h                 ; 获取状态
        test    al, 1                   ; 要读取键吗?
        jz      read_command
        call    keyboard_read           ; 读取，直到缓冲区空
        jmp     flush_all


; 读取编程命令字节并显示

read_command:
        mov     bl, 20h                 ; 获取命令字节功能
        call    keyboard_cmd            ; 激活
        call    error_cmd               ; 显示是否出错 (ah=1)
        call    keyboard_read           ; 向al中读入命令字节
        or      ah, ah                  ; 有效?
        jnz     error                   ; 若非，则跳转
        mov     bx, offset commandb
        call    hex                     ; 转化成ASCII
        OUTMSG  message4                ; 显示新的命令字节

        mov     dx, 100                 ; 退出前最大按键和
                                        ;   释放次数

; 在屏幕上显示按键扫描码，退出程序
;   时显示escape 键的释放码
```

```
next_key:
        call    keyboard_read           ; 向al中读入命令字节
        or      ah, ah                  ; 有效吗?
        jnz     next_key                ; 若无按键则循环

display_key:
        push    ax                      ; 保存键供后面使用
        push    dx
        cmp     [cmd_line], 'K'         ; 键盘扫描码类型?
        je      display_kscan

        mov     bx, offset scanbytN
        call    hex                     ; 转化成ASCII
        OUTMSG  message5                ; 显示扫描信息
        jmp     display_next

display_kscan:
        mov     bx, offset scanbytK
        call    hex                     ; 转化成ASCII
        OUTMSG  message6                ; 显示键盘扫描信息

display_next:
        pop     dx
        pop     ax

        mov     bl, 76h                 ; Escape 键的键盘扫描码
        cmp     [cmd_line], 'K'
        je      check_if_esc
        mov     bl, 1                   ; Escape 键的扫描码
check_if_esc:
        cmp     al, bl                  ; 是Escape 键吗?
        je      escape                  ; 若是，则退出
        dec     dx
        jnz     next_key                ; 如果小于100，则获取下一个键
        OUTMSG  message8                ; 显示完成
```

```
        jmp     done

; 按下Escape—这样将控制器恢复到正常操作状态。

escape:
        OUTMSG  message9                ; 显示完成
done:
        mov     bl, 60h                 ; 设置命令字节功能
        call    keyboard_cmd            ; 激活
        call    error_cmd               ; 显示是否出错 (ah=1)
        mov     al, 45h                 ; 重启回到正常状态
        call    keyboard_write          ; 向键盘发送命令
        call    error_write             ; 如果出错，则显示
        jmp     exit


; 显示键盘读超时错误信息

error:
        OUTMSG  message7                ; 显示错误信息

exit:
        sti
        mov     bl, 0AEh                ; 开放键盘
        call    keyboard_cmd            ; 执行
        call    error_cmd               ; 显进是否出错 (ah=1)

; 将键盘LED返回到正常状态

        mov     ax, 40h
        mov     es, ax
        mov     bl, es:[97h]            ; 获取LED实标志
        call    setLEDs

; 将重复速率设置到接近正常值
```

```
        mov     ax, 305h                ; 设速重复速率
        mov     bx, 102h                ; 使用500毫秒延时，24cps
        int     16h                     ; 执行

        mov     ah, 4Ch
        int     21h                     ; 退出
keyscan endp
```

代码例 8-6 交换键 TSR

这个小型的 TSR 用来交换 CapsLock 键和 Ctrl 键的功能。该 TSR 展示了改变键以适应用户的习惯是多么的简单。运行时，该 TSR 截取中断 15h 的功能 4Fh 来查看输入的系统扫描码。如果按下了“CapsLock”或“Ctrl”，那么就改变其扫描码。再次运行 TSR 将执行一个双重交换，其结果是回到键的正常操作状态。

```
; ---------------------------------------------------------------------------
;                                 CTRL2CAP
; ---------------------------------------------------------------------------
;
;   交换 Ctrl 和 CapsLock 键功能的TSR
;   系统扫描码 Ctrl = 1Dh, and for Caps Lock = 3Ah
;
;   所有的 AT+ (PC/XT除外).
;
;   (c) 版权所有 1994  Frank van Gilluwe 保留所有权力

include undocpc.inc


cseg    segment para public
        assume  cs:cseg, ds:cseg, ss:tsrstk

;
; 这是一个驻留处理程序来检测是否按下了CAPS和CTRL。
;   如果按下了键，则交换系统扫描码。
```

```
installmsg db      CR, LF
           db      'CTRL2CAP TSR Installed'
           db      ' - Ctrl key swapped with Caps-Lock'
           db      CR, LF, '$'

cseg       ends


;=========================================== 堆栈 ==========

tsrstk     segment para stack

           db      150 dup (0)

tsrstk     ends

           end     start
```

代码例 8-7　功能检测

有许多中断 16h 功能并不为所有的系统 BIOS 所支持。下两个子程序检测系统是否支持功能 5、9、10h、11h、12h。软盘中提供的程序 KEYBIOS 显示了这两个子程序的结果。

```
; ---------------------------------------------------------------------------
;    KEYTYPE
;        检查键盘BIOS是否支持扩展功能中断16h,
;        功能5、10h、11h和12h。
;        要实现这一点，必须清空键盘
;        缓冲区忽略前面的键。
;        程序然后将键值FFFF填入到缓冲区
;        并使用键盘BIOS扩展功能读取它。
;
;        返回:          如果ah=0, 则支持扩展功能
;                       如果ah=1, 则不支持扩展功能
```

```
; ================================================================
; 非驻留安装部分的起点

ctrl2cap        proc    far

start:
        push    cs
        pop     ds

; get the Current inderrupt 15h Veetor and save

        mov     al, 15h
        mov     ah, 35h
        int     21h                         ; 获取当前的中断15h指针
        mov     word ptr old_int_15h, bx
        mov     word ptr old_int_15h+2, es

; install our new interrupt 15h routine

        mov     dx, offset int_15h_hook
        mov     al, 15h
        mov     ah, 25h
        int     21h                         ; 安装我们的中断

        OUTMSG  installmsg                  ; 显示安装信息

; 确定剩余驻留（段落）的大小，成为TSR
;   and become a TSR

        mov     dx, (offset start - offset int_15h_hook) SHR 4
        add     dx, 11h                     ; PSP大小加上一段
        mov     ax, 3100h                   ; 退回到DOS，成为TSR
        int     21h
ctrl2cap        endp
```

```
int_15h_hook    proc    far
        pushf
        cmp     ah, 4Fh             ; 中断9键功能?
        jne     skip_change         ; 如果不是，则跳转

        cmp     al, 3Ah             ; 按下了CAPS?
        je      caps_make           ; 如果是，则跳转
        cmp     al, 0BAh            ; 释放了CAPS?
        je      caps_release        ; 如果是，则跳转

        cmp     al, 1Dh             ; 按下了Ctrl ?
        je      ctrl_make           ; 如果是，则跳转
        cmp     al, 9Dh             ; 释放了 ctrl ?
        je      ctrl_release        ; 如果是，则跳转
        jmp     skip_change

caps_make:
        mov     al, 1Dh             ; 换成Ctrl按键
        jmp     finish_int15
caps_release:
        mov     al, 9Dh             ; 换成Ctrl释放键
        jmp     finish_int15
ctrl_make:
        mov     al, 3Ah             ; 换成CAPS按键
        jmp     finish_int15
ctrl_release:
        mov     al, 0BAh            ; 换成CAPS释放

finish_int15:
skip_change:
        popf
        jmp     cs:old_int_15h      ; 处理老式中断15h

int_15h_hook    endp

old_int_15h     dd      0           ; 老式指针保存在这里
```

```
;
;       用到的寄存器:    ax

keytype proc      near
        push      cx
key_loop1:
        mov       ah, 1                ; 功能1，获取状态
        int       16h
        jz        key_try5             ; 如果缓冲区中无键，则跳转
        mov       ah, 0                ; 功能0，获取键
        int       16h                  ; 忽略键
        jmp       key_loop1            ; 循环直到缓冲区空

key_try5:
        mov       ah, 5                ; 尝试功能5
        mov       cx, 0FFFFh           ; 放入假的键值
        int       16h
        cmp       al, 0                ; al = 0 成功, 1 失败
        ja        key_no_ext

        mov       cx, 16               ; 试16次
key_loop2:
        mov       ah, 11h              ; 功能11h，获取状态
        int       16h
        jz        key_no_ext           ; 如果缓冲区中无键，则跳转
        mov       ah, 10h              ; 功能10h，获取键
        int       16h
        cmp       ax, 0FFFFh           ; 出现了键吗?
        je        key_ok               ; 如果出现，则跳转，成功
        loop      key_loop2            ; 试下一个键

; 不支持扩展BIOS功能

key_no_ext:
        mov       ah, 1
        jmp       key_ret
```

```
; 键盘BIOS不支持扩展功能

key_ok:
        xor     ah, ah
key_ret:
        pop     cx
        ret
keytype endp


; ---------------------------------------------------------------------------
;   KEY9
;       检查键盘BIOS是否支持扩展功能中断16h功能9。
;       确定支持其他扩展功能之后才进行这项确认。
;       别外，用到中断15h的功能C0h来进行确认。
;
;       返回:           如果 ah=0, 支持扩展功能9
;                       如果 ah=1, 不支持功能9
;
;       用到的寄存器:    ax

key9    proc    near
        push    bx
        push    es
        mov     ah, 0C0h                ; 获取指向ROM配置的指针es:bx
        int     15h                     ; ROM configuration
        jc      key9_not                ; 如果不支持，则跳转
        test    byte ptr es:[bx+6], 40h ; 检查是否支持
        jz      key9_not                ; 如果不支持，则跳转

; 支持中断16h功能9

        mov     ah, 0
        jmp     key9_exit
```

```
key9_not:
        mov     ah, 1
key9_exit:
        pop     es
        pop     bx
        ret
key9 endp
```

端口归纳

这个端口列表用于同键盘和主板控制器之间的通信。

端口	类型	功能	平台
60h	I/O	键盘数据	AT+
60h	输入	键盘数据	PC/XT
64h	I/O	键盘控制器状态与命令	AT+

命令误解

不同的命令被送到键盘和主板键盘控制器。另外，控制器还有一个“命令字节”。每个都提供了重要而又不同的功能，它们容易被误解，因为有很多相似的术语。下面的关键信息有助于区分每个命令类型。

- 键盘命令——送到键盘控制器的命令总是写到端口 60h。
- 主板控制器命令——送到主板键盘控制器的命令总是写到端口 64h。
- 命令字节——通过一条送到键盘控制器的命令访问键盘控制器的命令字节。

某些命令将数据传给主板键盘控制器或传给键盘本身。数据总是通过端口 60h 传送。我想在你阅读特定的端口功能时对这些内容会更加清楚。

端口细节

端口	类型	描述	平台
60h	I/O	键盘数据	AT+

这个端口与主板键盘控制器，主要是一个 8042 微处理器相连，它可用于输入和输出。从这个端口读取各种信息，例如按下键的扫描码。向端口 60h 发送一个字节将向主板控制器写数据。大多数情况下，数据作为一条命令或数据传递给键盘。向端口 64h 发送一个字

节被当作向主板控制器发送一条命令，在某些情况下，在这个发送命令之后会紧跟向端口 60h 发送数据。

注意，端口 64h 的状态位第 0 位和第 1 位表示了控制器的缓冲区状态。如果端口 64h 的第 1 位是 0，那么允许向端口 60h 读。

输入（第 0~7 位）——从主板键盘控制器读取一个字节

输出（第 0~7 位）——向主板键盘控制器写入一个字节

下表归纳了有效的命令功能

命令归纳

序号	端口 60h 命令
E6h	将鼠标比例设置为 1:1
E7h	将鼠标比例设置为 2:1
E8h	设置鼠标分辨率
E9h	获取鼠标信息
Deh	LED 写
EEh	诊断回声
F0h	设置/获取交替扫描码
F2h	读键盘 ID
F2h*	读鼠标 ID
F3h	设置重复键入信息
F3h*	设置鼠标采样速率
F4h	键盘开放
F4h*	鼠标开放
F5h	设置缺省和禁止键盘
F6h	设置缺省和禁止鼠标
F7h	将所有的键设置为重复键入
F8h	将所有的键设置为按键/释放
F9h	将所有的键置为按键
FAh	将所有的键设置为重复按键/按键/释放
FBh	将某个键设置为重复键入
FCh	将某个键设置为按键/翻译
FDh	将某个键仅设置为按键
FEh	重新发送
FFh	键盘重启
FFh*	鼠标重启

注：*这些命令带有到端口 64h 的前缀 D4h 命令

命令细节

命令	描述	端口
E6h	将鼠标比例设置为 1:1	60h

为配备有内置鼠标端口的系统设置鼠标比例因子为 1:1。在该命令之前必须先向端口 64h 发送 D4h 命令。

命令	描述	端口
E7h	将鼠标比例设置为 2:1	60h

为配备有内置鼠标端口的系统设置鼠标比例因子为 2:1。在该命令之前必须先向端口 64h 发送 D4h 命令。

命令	描述	端口
E8h	设置鼠标分辨率	60h

为配备有内置鼠标端口的系统设置鼠标分辨率。在该命令之前必须先向端口 64h 发送 D4h 命令。在将 E8h 送到端口 60h 后，紧接着向端口 60h 发送分辨率值字节。该分辨率字节保存了一个代码提供了下面四种可能的分辨率：

0=25dpi，　每毫米一个计数单位
1=50dpi，　每毫米两个计数单位
2=100dpi，　每毫米四个计数单位
3=200dpi，　每毫米八个计数单位

命令	描述	端口
E9h	获取鼠标信息	60h

获取鼠标的信息的设置。在该命令之前必须先向端口 64h 发送 D4h 命令。在将 E9h 送到 60h 后，从端口 60h 读取三个字节。这三个字节包含下面的信息：

第一个字节——状态

位		
	7=0	未使用
	6=0	流模式
	1	远程模式
	5=0	禁止
	1	开放
	4=0	比例设定为 1:1

1	比例设定为 2:1
3=0	未使用
2=1	按下左按钮
1=1	未使用
0=1	按下右按钮

第二个字节——分辨率

0=25dpi，每毫米一个计数单位
1=50dpi，每毫米两个计数单位
2=100dpi，每毫米四个计数单位
3=200dpi，每毫米八个计数单位

第三个字节——采样率值，单位为每秒通报次数（例如，64h=100 次通报每秒）

命令	描述	端口
EDh	LED 写	60h

在向端口 60h 发送了一个 Edh 的 LED 写命令字节后，再向端口写入一个字节来设置键盘的 LED 状态。

位	7=0	未使用
	6=0	未使用
	5=0	未使用
	4=0	未使用
	3=0	未使用
	2=1	CapsLock 的 LED 开
	1=1	NumLock 的 LED 开
	0=1	ScrollLock 的 LED 开

命令	描述	端口
EEh	诊断回声	60h

命令键盘回响一个字节，该返回的字节将也是值 EEh，这对于基本诊断很有用。

命令	描述	端口
F0h	设置/获取交替扫描码	60h

在 MCA 及其他一些系统上，键盘可以从三个键盘扫描码集中选择一个。向端口 60h 发送如下一个值：

0	读取当前的扫描码集
1	激活扫描码集 1（2 型控制器不支持）
2	激活扫描码集 2（缺省）
3	激活扫描码集 3

为了读取 1 型控制器上的当前扫描码集，主板控制器不得翻译输入的键盘扫描码。为了关闭键盘的翻译属性，必须将控制器命令的第 6 位置 0。然后将 F0h 命令发送到端口 60h，紧接着发送了功能字节值 0。这时读取端口 60h 就可以获取当前扫描码的值。最后就发一个控制器命令来恢复正常的翻译属性（第 6 位置 1）。

扫描码集 1 键盘看起来像一个 PC/XT 键盘，后者会产生系统扫描码。主板控制器并不要求翻译。扫描码集 2 使键盘生成键盘扫描码，它需要主板控制器翻译。在 101+键键盘上，某些键生成不同的输出，这取决于 shift 键和 Num Lock 键的状态。在这种情况下，按下单独的一个键可能会导致键盘向控制器发送多达 5 个字节。扫描码集 2 对 AT+系统来说是常见的。

PS/2 还提供了第三种集合。按下任何一个键时扫描码集 3 生成一个单独的字节。任何 shift 状态不影响键。因为其他 AT+系统不支持扫描码集 3，我可以深入控究的唯一有用的目的是在生产时测试键盘。

命令	描述	端口
F2h	读取键盘 ID	60h

在该命令之后读取端口 60h 两次可以获取键盘 ID 的两个字节。记住至少要等待 10ms 来等待键盘响应。并非所有键盘都支持。

命令	描述	端口
F2h	读取鼠标 ID	60h

在该命令之后读取端口 60h 两次可以获取鼠标 ID 的两个字节。在执行这个命令之前必须先向端口 64h 发一个 D4h 命令。记住至少要等待 10ms 来等待键盘响应。只有一些键盘支持该命令。

命令	描述	端口
F3h	设置重复延迟信息	60h

该命令设定键盘的重复速率以及延迟直到一个按下的键开始重复。F3h 命令之后的第 2 个字节发送到端口用来装入新的速率和延时值。参看本章的警告部分以获取一个设置键入信息的例子。

位	7=0	未使用
	6=x 5=x	设置键盘重复发生前的延迟 第 6 位　第 5 位 0　0=250ms 延时 0　1=500 ms 延时（缺省值） 1　0=750ms 延时 1　1=1000 ms 延时
	4=x 3=x 2=x 1=x 0=x	重复速率（参看表 8-10）

表 8-10　重复速率表

位 4 3 2 10	十六进制数	每秒字符数	位	十六进制数	每秒字符数
00000	0	30.0	10000	10	7.5
00001	1	26.7	10001	11	6.7
00010	2	24.0	10010	12	6.0
00011	3	21.8	10011	13	5.5
00100	4	20.0	10100	14	5.0
00101	5	18.5	10101	15	4.6
00110	6	17.1	10110	16	4.3
00111	7	16.0	10111	17	4.0
01000	8	15.0	11000	18	3.7
01001	9	13.3	11001	19	3.3
01010	A	12.0	11010	1A	3.0
01011	B	10.9*	11011	1B	2.7
01100	C	10.0	11100	1C	2.5
01101	D	9.2	11101	1D	2.3
01110	E	8.6	11110	1E	2.1
01111	F	8.0	11111	1F	2.0

注：*复位后的缺省值

命令	描述	端口
F3h	设定鼠标采样速率	60h

装入鼠标采样速率。发送到端口 64h 的命令必须在该命令之前发送。然后将 F3 命令发

送到端口 60h，紧跟着到端口 60h 的采样速率。该采样速率的单位是每秒的通报数。例如值 50 表明鼠标会每秒通报 50 次。只有某些系统才支持该命令。

命令	描述	端口
F4h	键盘开放	60h

如果出现了传输错误，会自动禁止键盘。该命令重新开放键盘，并清空键盘的 16 字符内部缓冲区。

命令	描述	端口
F4h	鼠标开放	60h

开放鼠标。在该命令之前，必须向端口 64h 发送命令 D4h。然后向端口 60h 发送命令。只有某些系统才支持该命令。

命令	描述	端口
F5h	设置缺省和禁止键盘	60h

将键盘重设为其缺省状态。其输出缓冲区被清空，三个 LED 都被关闭，并且重复键入速率和延时设定为它们的缺省值。键盘扫描被禁止。

命令	描述	端口
F5h	设置缺省和禁止鼠标	60h

将鼠标重设为其缺省状态。内部键盘输出缓冲区被清空，三个 LED 都被关闭，并且重复键入速率和延时设定为它们的缺省值。如果键盘是开放的，它将继续扫描键状态的改变。

命令	描述	端口
F6h	设置缺省	60h

重启键盘进入其缺省状态。清空键的，它将继续扫描键状态的改变。

命令	描述	端口
F7h	将所有的键设定为重复键入	60h

在 MCA 系统及某些其他的系统上，该命令清除内部键盘缓冲区，并在按键时间超过重复键入延时周期时自动重复该键。参看子命令 F3h 以了解更多的有关重复键入操作的信息。它只在设置了扫描码集 3 时才能发挥作用。参看键盘命令 F0h 以详细了解扫描码集 3。

命令	描述	端口
F8h	将所有的键设定为按键/释放码模式	60h

在 MC 及其他一些系统上，该命令清除内部键盘缓冲区，并在第一次按下某键时，设置所有的键来发出一个代码，而在释放该键时发出另外一个代码。这一点只在设置了扫描码集 3 时才会影响操作。参看键盘命令 F0h 以进一步了解扫描码集 3。

命令	描述	端口
F9h	将所有的键设定为按键码模式	60h

在 MCA 及其他一些系统上，该命令清除内部键盘缓冲区，并在某键第一次按下时，设置所有的键来发出一个代码。当释放键不生成代码。这一点只在设置了扫描码集 3 时才影响操作。参看键盘子命令 F0h 以进一步了解扫描码集 3。

命令	描述	端口
FAh	将所有的键设定为重复键入/按键/释放	60h

在 MCA 及其他系统上，该命令清除内部键盘缓冲区，并且在第一次按下某键时，设置所有的键发出一个代码，而释放某键时又发出另一个代码。参看子命令 F3h 以进一步了解有关键入速率选项。这一点只在设置了扫描码集 3 时才影响操作。参看键盘命令 F0h 以进一步了解扫描码集 3。

命令	描述	端口
FBh	将某个键设定为重复键入	60h

在 MCA 及其他一些系统上，该命令清除内部键盘缓冲区，并在按键时间超过重复键入延时周期时自动将该指定的键设置为重复模式。参看子命令 F3h 以更多地了解重复键入选项。发出了该子命令后，会向端口 60h 再次写入该键的键盘扫描码。

命令	描述	端口
FCh	将某个键设定为按键/释放	60h

在 MCA 及其他一些系统上，该命令清除内部键盘缓冲区，并在第一次按下某个键时将该键设置为发出代码模式，而释放该键会发出另外一个代码。在发出了该命令后，会向端口 60h 再次写入该键的键盘扫描码。

命令	描述	端口
FDh	将某个键设定为按键/释放	60h

在 MCA 及其他一些系统上，该命令清除内部键盘缓冲区，并在第一次按下某个键时将该指定的键设置为发出代码模式，但是在释放该键时不会生成代码。发出了该命令后。会向端口 60h 再次写入该键的键盘扫描码。

命令	描述	端口
FEh	重新发送	60h

在键盘发送出了问题之后，重新发送命令会指导键盘重新发送上一个字节。通常控制器使用该命令，而不是由键盘 BIOS 发送该命令。

命令	描述	端口
FFh	键盘重启	60h

强制键盘的完全重启。键盘会执行自检，清除其输出缓冲区，关闭三个 LED，并将重复键入速率和延时设置为缺省值。

命令	描述	端口
FFh	鼠标重启	60h

重启鼠标并将之设置为禁止状态。在该命令审定前必须先向端口 64h 发送命令 D4h。然后才向端口 60h 发送 FFh 命令。只有某些系统支持该命令。

端口	类型	描述	平台
60h	输入	键盘数据	PC/XT

在 PC/XT 上，同键盘的通信仅限于从键盘读取信息。不支持向键盘发送命令。

主板硬件将串行键盘数据转化成 8 位扫描码数。它通过端口 60h 读自 8255 芯片的端口 C。记住，扫描码是直接的键盘扫描码，不是 AT+类型机器上常见的再译码。

当键盘发送一个扩展扫描码时，读取的第一个字节是零。这意味着必须读取第二个字节才能获得该扩展扫描码字节。

输入（第 0 位~第 7 位）——按键信息

端口	类型	描述	平台
64h	I/O	键盘控制器状态和命令	AT+

这个端口直接与主板键盘控制器相连，后者通常是一个 Intel 8042 微处理器。任何时候都可以从这个端口读取主板控制器的状态。8042 处理器的控制数据可以通过端口 60h 和 64h 写入该处理器。当从端口 60h 向主板控制器发送数据时，就是一个写数据操作。向端口 64h 输出代表了一个写命令操作。参看端口 60h 获取相关信息。

输入（0~7 位）——控制器状态（AT/EISA）

位	7 r=1	来自键盘的串行连接器发生了奇偶错误（上次发送的字节是偶奇偶性，便是只允许奇奇偶性）
	6 r=1	接收超时，表示键盘开始发送信息，但是在适当的超时延时内没有完成传送
	5 r=1	传送超时，表示键盘传送超过了预先设定的时限。如果传送一个字节占用了太长的时间，或者字节传送完毕但是响应超时，都会发生这种情况。如果对超时的响应含有奇偶错误，也会发生这种错误（这时，奇偶错误位和传送超时位都会被设置）
	4 r=0	禁止键盘（通过键盘的锁开关）。无论何时向键盘控制器发送数据都会更新该标志位。在密码控制系统上，0表示键盘被禁止直到密码正确
	3 r=0	上次是数据送到控制器（使用端口 60h）
	1	上次是命令送到控制器（使用端口 64h）
	2 r=0	上电造成重启
	1	成功地完成了主板控制器自检。该位的状态也可以在命令 64h 中依据系统标志位设置。（参看输出，命令）
	1 r=0	主板控制器的输入缓冲区空，可以向端口 60h 或 64h 写
	1	主板控制器的输入缓冲区满。除非主板控制器清空了这些缓冲区，否则就不能向端口 60h 或 64h 写，这样做会丢失数据/命令
	0 r=0	主板控制器的输出缓冲区空，读端口 60h 无效
	1	主板控制器的输出缓冲区有一个字节可读，使用端口 60h 可以读取该字节

输入（第 0~7 位）——控制器状态（MCA PS/2）

位	7 r=1	来自键盘的串行连接器发生了奇偶错误（上次发送的字节是偶奇偶性的，但是只允许奇奇偶性）。如果发生了奇偶错误，就将 FFh 装入输出缓冲区（可以从端口 60h 读取）
	6 r=1	发生了一般性超时，表示发生了几种可能的错误条件。这时，FFh 被装入输出缓冲区（可以从端口 60h 读取）。可能的错误条件是： a) 键盘已开始传送信息，但是在适当的超时延时内没有完成传送

b) 键盘传送超过了预设的时间限。如果传送一个字节占用了太长的时间，或者字节传送完毕但是响应超时，都会发生这处情况。如果对超时的响应含有奇偶错误，那么也会出现这种错误（这时，奇偶错误位和传送超时位都会被设置为 1）

5 r=1 鼠标输出缓冲区满，也与第 0 位有关：

第 5 位	第 0 位
0	0=两个缓冲区都空
0	1=主板控制器输出缓冲区空
1	0=未使用
1	1=鼠标输出缓冲区空

4 r=0 禁止键盘（通过键盘的锁开关）。无论何时向键盘控制器发送数据都会更新该标志位。在密码控制的系统上，0 表示键盘被禁止直到密码正确

3 r=0 上次是数据送到控制器（使用端口 60h）

1 上次是命令发送到控制器（使用端口 64h）

2 r=0 上电造成重启

1 成功地完成了主板控制器自检。该位的状态也可以在命令 64h 中依据系统标志位设置。（参看输出，命令）

1 r=0 主板控制器的输入缓冲区空，可以向端口 60h 或 64h 写

1 主板控制器的输入缓冲区满。除非主板控制器清空了这些缓冲区，否则就不能向端口 60h 或 64h 写，这样做会丢失数据/命令

0 r=0 主板控制器的输出缓冲区空，读端口 60h 无效

1 主板控制器的输出缓冲区有一个字节可读，使用端口 60h 可以读取该字节。参看第 5 位了解哪个缓冲区（键盘或鼠标）为空

输入（第 0~7 位）——控制器状态发送一个命令字节

下面列出了命令功能，字节送往端口 64h。

命令归纳

号	端口 64h 命令
20h	获取命令字节
21h~3Fh	读控制器 RAM
60h	写命令字节
61h~7Eh	写控制器 RAM

A4h	检查是否安装了密码
A5h	装载密码
A6h	检查密码
A7h	禁止鼠标端口
A8h	开放鼠标端口
A9h	测试鼠标端口
AAh	自检
ABh	接口测试
ACh	诊断转储
ADh	禁止键盘
AEh	开放键盘
C0h	读输入端口
C1h	连续输入端口查询，低
C2h	连续输入端口查询，高
D0h	读输入端口
D1h	写输入端口
D2h	写键盘输入缓冲区
D3h	写鼠标输入缓冲区
D4h	向鼠标写
DDh	禁止 A20 地址线
DFh	开放 A20 地址线
E0h	读测试输入
F0h~FDh	脉冲输出位
FEh	系统重启

命令细节

命令	描述	端口
20h	获取命令字节	64h

读取当前键盘命令字节。首先将命令字节 20h 发送到端口 64h，然后从端口 60h 读取该值。键盘命令字节由下面的命令 60h 设置。在命令 60h 下也列出了命令字节的位置。

命令	描述	端口
20h~3Fh	读控制器 RAM	64h

读取内部主板控制器的 RAM。该地址是命令值减去 20h。要读取字节 9，可使用命令 29h。并非所有的控制器都支持这些功能。为了获得指定的字节，在该命令之后从端口 60h 读取字节。

在MCA系统上，1型控制器可以访问所有的31个地址。2型控制器只能访问RAM字节0、13h到17h、1Dh和1Fh。

偏移量	功能
0	命令字节——参看命令60h获得细节
13h	安全性开——允许密码时为非零（MCA）
14h	安全性关——密码匹配时为非零（MCA）
16h	密码忽略 1——如果在密码检测时生成码等于该字节，就会忽略(MCA)
17h	密码忽略 2——如果在密码检测时生成码等于该字节，就会忽略它(MCA)

命令	描述	端口
60h	写命令字节	64h

向控制器写一个命令字节。向端口60h写的下一个字节就作为新的主板控制器命令字节。大多数正常操作的命令字节值是45h。该命令字节中的位定义如下：

命令字节——ISA/EISA

位		
	7=0	未使用，设定为0
	6=0	不必转化键盘的扫描码
	1	标准扫描转化——来自键盘的扫描码转化为PC上使用的普通扫描码（1为正常操作）
	5=0	检查键盘的奇偶性，带扫描转化（0是正常操作）
	1	忽略键盘的奇偶性，不进行扫描码转化
	4=0	开放键盘
	1	将键盘时钟置低平从而强制关闭键盘。键盘不能进行数据收发
	3=1	覆盖键盘禁止功能。端口64h的第4位置1，忽略键盘锁开头（这用于上电期间的键盘测试）
	2=0	系统标志位表示上电重启
	1	控制器成功自检之后的系统标志位
	1=0	未用，设定为0
	0=0	键盘输出缓冲区满时不发送中断
	1	键盘输出缓冲区满引发中断（IRQ1）

命令字节——MCA与PS/2

位		
	7=0	未使用，设定为0
	6=0	不必转化键盘的扫描码

1	标准扫描转换——来自键盘的扫描码转化为 PC 上使用的普通扫描码（1 为正常操作）。MCA 2 型控制器不能将该位置 1。这种情况下设定扫描码转化需要对端口 64h 使用键盘命令 F0h
5=0	开放鼠标
1	将鼠标串行时钟线强制置低来禁止鼠标。鼠标不能进行数据收发
4=0	开放键盘
1	将键盘时钟强制置低来禁止键盘。键盘不能进行数据收发
3=0	未用，设定为 0
2=0	系统标志位表示上电重启
1	控制器成功自检之后的系统标志位
1=0	鼠标输出缓冲区满时不触发中断
1	鼠标输出缓冲区满时触发中断（IRQ12）
0=0	键盘输出缓冲区满时不触发中断
1	键盘输出缓冲区满时触发中断（IRQ1）

命令	描述	端口
60h~7Eh	写控制器 RAM	64h

写内部主板控制器的 RAM。地址是命令值减去 60h。要写第 5 个字节，使用命令 65h。并非所有的控制器都支持这些功能。要写指定的字节，该命令后紧接着写端口 60h。

在 MCA 系统上，1 型控制器可以访问所有的 31 个地址，但 2 型控制器只能访问 RAM 字节 0,13h 到 17h,10h 和 1Fh。

偏移量	功能
0	命令字节，参看命令 60h 获得细节
13h	安全性开——允许密码时为非零（MCA）
14h	安全性关——密码匹配时为非零（MCA）
16h	密码忽略 1——如果在检测密码时生成码等于该字节，就会忽略它(MCA)
17h	密码忽略 2——如果在检测密码时生成码等于该字节，就会忽略它(MCA)

命令	描述	端口
A4h	检查是否安装了密码	64h

该命令检查是否以前在主板控制器上保存了密码。该命令返回两个状态值中的一个，可以从端口 60h 读取。没有安装密码时返回 F1，安装了密码时返回 FA。许多系统不支持

该密码方法（但是所有的 MCA 系统支持）。

命令	描述	端口
A5h	装载密码	64h

要装入一个新密码，可以在发送了装载密码命令之后，紧接着在端口 60h 向主板控制器写入密码字节。如果送了一个 0，那么就没有密码。密码必须保存为扫描格式（不是 ASCII）。许多系统不支持该密码方法（但是所有的 MCA 系统支持）。

命令	描述	端口
A6h	检查密码	64h

如果主板控制器带有一个有效的密码，该命令可以指导控制器匹配键盘输入的符号和密码。如果成功匹配，就开放键盘。许多系统不支持该密码方法（但是所有的 MCA 系统支持）。

命令	描述	端口
A7h	禁止鼠标端口	64h

将命令字节的第 5 位置高可以禁止鼠标。然后鼠标的时钟线置低，防止鼠标发收任何数据（仅 MCA）。

命令	描述	端口
A8h	开放鼠标端口	64h

将命令字节的第 5 位置低可以开放鼠标。如果以前置低了到鼠标的时钟线，则现在被激活（仅 MAC）。

命令	描述	端口
A9h	测试鼠标端口	64h

对控制器和鼠标之间的串行连接器进行测试。它测试鼠标的数据和时钟线。从端口 60h 读取测试结果。下面的值列出了测试结果：

00h——未检测到错误
01h——鼠标时钟线粘低
02h——鼠标时钟线粘高
03h——鼠标数据线粘低
04h——鼠标数据线粘高

命令	描述	端口
AAh	自检	64h

启动主板控制器的内部自检。如果没有检测到错误，那么可以从端口 60h 读取值 55h。

命令	描述	端口
ABh	接口测试	64h

启动测试控制器和键盘之间的串行连接器。它测试键盘的数据和时钟线。从端口 60h 读取测试结果。下面的值列出了测试结果：

00h——未检测到错误
01h——键盘时钟线粘低
02h——键盘时钟线粘高
03h——键盘数据线粘低
04h——键盘数据线粘高

命令	描述	端口
ACh	诊断转储	64h

获取控制器 RAM 的 16 字节、当前控制器输入和输出端口状态以及控制的程序状态字。可以从端口 60h 读取该信息。

命令	描述	端口
ADh	禁止键盘	64h

将命令字节的第 4 位置高来禁止键盘。然后将键盘的时钟线置低，阻止键盘收发任何数据。

命令	描述	端口
AEh	开放键盘	64h

将命令字节的第 4 位置高来开放键盘。如果以前置低了键盘时钟线，就激活它。

命令	描述	端口
C0h	读取输入端口	64h

读 8042 控制器的输入端口，P1。然后就可从端口 60h 读取该字节。只有当控制器输出端口为空时才可以发送该命令。位的分配情况随系统有很大不同。

在发送了命令 C0h 后如果读取端口 60h，那么在早期的 IBM AT 上会返回下列数据。

位	7=1	键盘禁止开关开
	6=0	上电时缺省视频适配器是彩色文本/图形
	1	上电时缺省视频适配器是单色适配器
	5=0	安装了生产跳线
	1	正常
	4=0	RAM 大小选择开关=512K
	1	RAM 大小选择开关=256K
	3=x	未使用
	2=x	未使用
	1=x	未使用
	0=x	未使用

在发送了命令 C0h 后如果读取端口 60h，那么在 PS/2 MCA 系统上会返回下列数据。

位	7=0	未使用
	6=0	未使用
	5=0	未使用
	4=0	未使用
	3=0	未使用
	2=0	键盘电源正常
	1	没有键盘电源
	1=x	鼠标串行数据入
	0=x	键盘串行数据入

命令	描述	端口
C1h	连续输入端口P1低	64h

8042 主板控制器端口 P1 的低 4 位连续记录在一起，并把状态寄存器的第 4 位放到第 7 位中。可以从端口 64h 读取该状态寄存器的内容。这将继续下去直到收到下一个控制器命令。并非所有的系统都支持这个功能，但是带有 1 型控制器的所有 MCA 系统都支持这个功能。2 型控制器不支持这个功能。

命令	描述	端口
C2h	连续输入端口P1高	64h

8042 主板控制器端口 P1 的高 4 位连续记录在一起，并把状态寄存器的第 4 位放到第 7

位中。可以从端口 64h 读取该状态寄存器的内容。这将继续下去直到收到下一个控制器命令。并非所有的系统都支持这个功能，但是带有 1 型控制器的所有 MAC 系统都支持这个功能。2 型控制器不支持这个功能。

命令	描述	端口
D0h	读输出端口	64h

读 8042 控制器的输出端口 P2。可以从端口 60h 读取该字节。只有在控制器的输出端口为空时才可发送该命令。

IBM AT 如下分配各位：

位	7=x	到键盘线的数据
	6=x	键盘数据时钟
	5=0	输入缓冲区空
	4=1	键盘输出缓冲区满（与 IRQ1 相连）。当从端口 60h 读了输出缓冲区后会清除该位
	3=x	未使用
	2=x	未使用
	1=x	A20 状态。当前 A20 线的状态，0 表示 A20 被禁止。某些系统无论实际 A20 状态如何，都设置该位
	0=0	主处理器重启（该位不可以读取，因为从本质上讲，系统就位于重启模式下）
	1	正常

MCA 系统如下分配各位

位	7=x	到键盘的数据
	6=x	键盘数据时钟
	5=1	控制器输出缓冲区满，有一个鼠标字节（与 IRQ12 相连）。如果从端口 60h 读了输出缓冲区，就会清除该位
	4=1	控制器输出缓冲区满，有一个键盘字节（与 IRQ1 相连）。如果从端口 60h 读了输出缓冲区，就会清除该位
	3=x	到鼠标的数据
	2=x	鼠标数据时钟
	1=0	禁止 A20 地址线，将写 1MB 以外地址返转到低端内存。这一点模拟 8088 机器的操作，这是上电缺省状态
	1	开放 A20 地址线，允许访问 1MB 以上内存

	0=0	主处理器重启（该位不可以读到，因为从本质上讲，系统就位于重启模式下）
	1	正常

命令	描述	端口
D1h	[illegible]	64h

写往端口 60h 的下一字节被传送到控制器的输出端口 P2。

IBM AT 如下分配各位

位	7=x	到键盘的数据
	6=x	键盘数据时钟
	5=0	输入缓冲区空
	4=1	激活 IRQ1（中断 9）。如果从端口 60h 读了输出缓冲区，该位就被清零
	3=x	未用
	2=x	未用
	1=0	禁止 A20 地址线，将写 1MB 以外地址返转到低端内存。这一点模拟 8088 机器的操作，这是上电缺省状态。
	1	开放 A20 地址线，允许访问 1MB 以上内存
	0=0	重启主处理器（硬件重启）
	1	正常

带有 1 型控制器的 MCA 系统允许改变输出端口的所有位。2 型控制器忽略除第 1 位外的所有位，第 1 位用来控制 A20 线。下面的位分配情况仅适用于 MCA 系统：

位	7=x	到键盘的数据
	6=x	键盘数据时钟
	5=1	激活 IRQ12（中断 74h）。如果从端口 60h 读了输出缓冲区，该位就被清零
	4=1	激活 IRQ1（中断 9）。如果从端口 60h 读了输出缓冲区，该位就被清零
	3=x	到鼠标的数据
	2=x	鼠标数据时钟
	1=0	禁止 A20 地址线，将写 1MB 以外地址返转到低端内存。这一点模拟 8088 机器的操作，是上电缺省状态
	1	开放 A20 地址线，允许访问 1MB 以上内存
	0=0	重启主处理器（硬件重启）
	1	正常

命令	描述	端口
D2h	写键盘输出缓冲区	64h

在控制器输出缓冲区中装入下一个写往端口 60h 的字节。一旦该字节写到了端口 60h 并且控制器命令允许 IRQ1（第 0 位=1），那么 IRQ1 就会被激活，好像键盘激活了它一样。并非所有的 AT+系统都支持这个功能，但是所有的 MCA 系统都支持它。

命令	描述	端口
D3h	写鼠标输出缓冲区	64h

在控制器输出缓冲区装入下一个写往端口 60h 的字节。一旦该字节写到了端口 60h 并且控制器命令字节允许 IRQ12（第 1 位=1），那么 IRQ12 就会被激活，好像鼠标激活了它一样。并非所有的 AT+系统都支持这个功能，但是所有的 MCA 系统都支持它。

命令	描述	端口
D4h	向鼠标写	64h

写往端口 60h 的下一个字节被传送到鼠标。并非所有的 AT+系统都支持这个功能，但是所有的 MCA 系统都支持它。

命令	描述	端口
DDh	禁止 A20F 地址线	64h

禁止 A20 地址线。这将内存访问限制在 1MB 以内，有点像 8088 的寻址范围。大多数系统不支持该命令。

命令	描述	端口
DFh	开放 A20F 地址线	64h

开放 A20 地址线。这允许访问 1MB 以上的内存。大多数系统不支持该命令。

命令	描述	端口
E0h	读测试输入	64h

获取 8042 测试输入线 T1 和 T2 的状态。读端口 60h 可以得到该字节。只有在控制器的输出缓冲区为空时才可以发送该命令。

位	7=x	未使用
	6=x	未使用
	5=x	未使用

4=x	未使用
3=x	未使用
2=x	未使用
1=x	键盘数据
0=x	键盘时钟

命令	描述	端口
F0h~FDh	脉冲输出位	64h

该命令允许有选择性地 Pulse 控制器输出位。命令 F0h 到 FDh 的低四位直接控制控制器上端口 P2 的四个输出位。命令 F0 到 FDh 以及 FFh 无任何用处。参看下面的命令 FEh 系统重启。在带有 2 型控制器的 MCA 系统上，不支持所有的这些命令。

命令	描述	端口
FEh	系统重启	64h

通过将系统的重启线置低大约 16 微秒来发一个硬件重启命令。参看第 3 章以更多地了解 CPU 重启。

第 9 章

视频系统

如果你购买本书是为了全面地理解视频系统，那么你就错了！与 PC 中的其他子系统不同的是，视频系统在 IBM 技术参考资料中有详细的论述，并且在许多书籍中有深入的论述。有这么多的资料讨论视频适配器，讨论它们是如何工作的及其他一些相关的信息，所以很容易找到另外一本书谈论视频。

对于视频系统没有多少留下尚未说明的内幕，但是这些信息都很零散。许多 BIOS 功能说明得很不够，以至于几乎很难使用它们。本章详细论述了一些关键信息并极其详细地阐释了 BIOS 中断调用。我还为 VGA 像素任务提示了几个有趣的尚未说明的 BIOS 功能。我归纳了几种适配器的 I/O 端口，但是只有那些尚未说明的端口才作了深入讨论。

我完整而详细地讨论了重定位屏幕接口说明（PSIS）。该接口可以使许多程序工作在更广的环境下，例如 DOS/V，一种特殊的日文版 DOS。一个使用 RSIS 的程序，在高级内存管理程序，比如 Memory Commander 下，可以获得额外的 100 到 300K DOS 主存而不会带来任何运行的不良效果。许多程序员发现将一个存在的程序转化成 RSIS 所花的时间不超过 30 分钟。

本章所讨论的视频适配器包括 MGA、HGA、CGA、MCGA、EGA、VGA、SVGA、SGA 以及 VESA XGA。另外，还包括为 RSIS 和 DOS/V 提供的 BIOS 功能。对于那些过时的 PGA、PCjr 和 PC 可逆型视频系统，则只作了很有限的讨论。

本章末尾包括大量的 RSIS 的代码例，以及一个完整的程序来检测机器带有哪种视频适配器、使用时的正确属性以及其他有用的信息。

简　介

视频系统是系统最复杂的部分之一。几年来发展了许多标准。由于视频区通常是运行的最大瓶颈，大多数程序绕过非常慢的 BIOS 屏幕程序，而直接向屏幕缓冲区写。为了实现更深入的应用，自从 1988 年所有的视配器都支持多种类型的显示器。

许多视频适配器硬件设计在发布时是一次激动人心的技术发展，但却是程序的一场劫难。虽然有文档详细介绍了适配器内的许多寄存器，这使得为一个特定的视频适配卡创建程序不再很难，但是它们很少提供实例甚至是很基本的描述来说明各寄存器之间的相关性。

另外，早期的硬件设计似乎只喜欢提供只可写寄存器。这样就很难检测视频系统所处的状态。很难编写那些需要保存和恢复视频系统的 TSR 程序。更糟的是，许多功能零散在各寄存器中，很杂乱。上述这些问题有一些已得到解决，专业程序员最大的困难之一是处理更多的不同标准。

所有程序员必须面对的一个问题是，新的标准在不断发展中，到本书写稿时为止，最新的 IBM 适配器是 XGA。在过去的几年中，关注的中心限制了所提供给程序员的那些重要的说明文档。我猜测 IBM 有些人会对此执有异议，但是从 VGA 开始，越来越多的部分未加以说明。随着许多高度集成的 IC 的出现，已经很难指出如何最大限度地充分利用这些高级的视频适配器或说明它们是如何工作的。程序员缺乏技术支持是新型适配器需要花很多年时间才能让人们支持它的主要原因，也是许多新标准不利用它的原因。解决这个问题的一种方法是开发软件，就好像存在的最新适配卡是 VGA 一样。

我不打算深入详细讨论不同视频适配器是如何工作的或者如何才能最好地对它们编程。详细资料请参考附录 C。

视频适配器标准

大多数今天的视频适配器标准都是 IBM 依据它生成的各种适配器而制定的。只有两种非 IBM 制定的标准获得广泛的接受，它是是 Hercules 单色图形适配器（HGA）和高级 VGA（SVGA）。

CGA——彩色图形适配器

CGA 提供基本的文本模式，使用一种损伤视力的 8x8 字符 cell 字体。它提供 16 色，并且在 PC 启动后提供唯一的图形能力。它支持 CGA 彩色适配器合成（像电视那样）彩色和合成单色显示器。在写屏幕时许多 CGA 适配器会产生“雪花”，并且要求专门的编程来在访问视频缓冲区期间避免该现象。今天 CGA 早已过时，大多数程序员在当前的设计中根本不考虑对 CGA 的支持。

MDA——单色显示适配器

MDA 支持 9x14 文本格，多年来它的分辨率一直最高。该适配器不支持图形，它只支持单色显示器。

HGA——Hercules 图形适配器

HGA 是 IBM 之外第一个设计成功的视频适配器，它和 MDA 相同，但是它提供两页单色图形，它几乎是 CGA 分辨率的两倍。

PGA——专业图形适配器

PGA 提供高分辨率的彩色图像。由于昂贵的价格，以及某些软件兼容问题，PGA 从未流行过。我从未劝过某人去编写一个 PGA 专用的软件，今天它已彻底过时。

EGA——增强型图形适配器

虽然 EGA 花了很长一段时间才比较成功，但是它却提供了一种高分辨率彩色图形标准，并且提供了全彩色高分辨率 8×14 格字符。提供了 16 种颜色图形，分辨率高达 640×350。EGA 远比 MDA 和 CGA 复杂，但是提供了丰富的特性，大大优于其前面的适配器。其中包括可以装入用户自定义字体、从 64 色调色板中选择 16 种颜色、大量新的图形模式等等。这些新特性需要一个 EGA 显示器。

尽管一些系统仍使用 EGA 适配器，但是最近几年没有新式的系统去使用这种标准。不再提供 EGA 适配卡，它已完全被新的 VGA 标准所替代。目前大多数程序员都认为 EGA 适配器已过时。

MCGA

这种标准用于 80386 以前的老式 PS/2 AT 总线的机器上。这些系统的主板上带有 MCGA 视频硬件。据我所知，它从来作为一个独立的适配卡使用。MCGA 是 VGA 特性的子集，只提供一个最大的分辨率 640×480。文本格是 8×16。

VGA——视频图形阵列

VGA 是 IBM 为计算机的 PS/2 宏通道线而制定的一个新标准，还提供了一个独立的 AT 总线适配器。VGA 适配器也允许替代内置的第一台 PA/2 AT 总线系统。它提供了比 EGA 更高的分辨率。标准的文本格增加为 9×16，图形的分辨率增加为 640×480。VGA 对 EGA 向下兼容，并且它还删除了那些讨厌的只写寄存器。

VGA 也在逐渐过时，但是今天仍有许多系统在使用 VGA。因为它的编程操作比一些更新的标准简单，许多程序员仍将 VGA 作为主要的视频设计对象。

SVGA——超级 VGA

SVGA 不是由 IBM 制定的，部分是由于 IBM 对更高分辨率适配器的需求响应太慢了的缘故。由于许多供应商引进了许多新的用户功能，但是它们彼此又不兼容，所以市场呈现一种无序状态。后来，他们总算达成一致，创建了视频电子标准协会（VESA）。VESA 创建了 SVGA 标准来访问这些更高级的操作模式。所有的 SVGA 适配器支持 800×600 图形分辨率，大多数同时还支持 1024×768 模式，具体情况取决于内存容量。在这些高分辨率图形模式下提供了 16256 种，甚至更多的色彩。

SVGA 标准的最大优点是它支持加入新的更高分辨率，程序员可以检测并使用它。这给予 SVGA 标准扩展性的性能，而不再局限于一个不变的标准。

几乎所有今天购买的系统都包括了一个某种形式的 SVGA。

XGA——扩展图形适配器

XGA 是 IBM 最近提供的适配器。它大大扩展了 VGA 的功能。客户文本字符可以达到令人难以置信的 255×255 格大小。分辨率超过了普通 VGA 的两倍，提供高达 1024×78 的图形分辨率。同时还提供了另外一些性能来提高运行速度和支持图形用户界面下的快速操作，这些用户图形界面包括 OS/2 和 Windows。视频电子标准协会也进一步扩展了 XGA 适配器，并将它称为 VESA XGA。

虽然投放市场多年，但是 XGA 及其兼容产品的销售并不好。XGA 只占据了一小部分视频适配器市场。很少有软件要求使用 XGA。

JEGA、XT-VGA—日文适配器

这两种适配器的功能类似标准的 EGA 和 VGA，但是还包括了一些支持日文双字节字符的硬件。这些适配器有几种新的模式，并且改变了许多标准模式处理中日文环境的方法。如果对这些细节感兴趣，请参看《AX 技术参考指南》。

适配器的名称

本章使用了适配器的通用简称。另外，许多称谓上还加上一个加号(+)，表示该称谓指的是某个特定的适配器以及所有更优的适配器。表 9-1 显示了每个简称所包括的适配器。

表 9-1 适配器名称归类

适配器	包括
EGA+	EGA、VGA、SVGA、SGA、VESA、XGA
VGA+	VGA、SVGA、XGA、VESA、XGA
XGA+	XGA、VESA、XGA

有时，我还指出 DOS/V 是否支持某个特定的功能。DOS/V 是一种 MS-DOS，它工作在 VGA+类型的适配器上，提供了双字节字符集（DBCS），主要供亚洲字符集使用。尽管 DOS/V 不是适配器，它也可能限制或改变某些视频 BIOS 功能的操作。DOS 允许任何标准的 VGA 适配器显示 DBCS 日文字符的最大集。

BIOS 服务

中断 10h 提供下列功能：

功能	描述	适配器
ah=0	设置视频模式	所有
ah=1	设置光标类型	所有
ah=2	设置光标位置	所有
ah=3	读取光标位置和类型	所有
ah=4	读取光笔位置	所有
ah=5	选择活动的显示页	所有
ah=6	向上翻滚活动页	所有
ah=7	向下翻滚活动页	所有
ah=8	读取字符和属性	所有
ah=9	写字符和属性	所有
ah=A	只写字符	所有
ah=B	设置调色板	所有
ah=C	写点	所有
ah=D	读点	所有
ah=E	在 Teletype 模式下写	所有
ah=F	读取当前视频模式	所有
ah=10	设置调色板寄存器功能	MCGA/EGA/VGA+
ah=11	字符生成功能	所有
ah=12	多种功能	所有
ah=13	写串	所有

ah=14	LCD	可逆型 PC
ah=15	获取显示器类型	可逆型 PC
ah=18	字体板的请求	DOS/V
ah=1A	读/写显示器组合码	VGA+
ah=1B	返回视频系统的状态信息	VGA+
ah=1C	保存/恢复视频状态	VGA+
ah=1D	换档状态线功能	DOS/V
ah=1F	显示器模式信息	XGA+
ah=4E	VESA XGA 子功能	VESA XGA
ah=4F	超级 VGA 子功能	SVGA
ah=FE	获取重定位屏幕地址	所有
ah=FF	更新重定位屏幕	所有

中断	功能	描述	平台
10h	0	设置视频模式	所有

设置当前的视频模式，所选择的模式指定了视频的类型、文本或图形、屏幕密度以及物理屏幕缓冲区地址。表 9-2 列出了常见的模式，在选择一个模式之前，必须先确定适配器的类型和当前的显示器类型。如果选择了一个不正确的模式，功能不会返回一个错误码，但是会忽略该无效的模式号，适配器会保持在上次有效设置的模式下。

对于 MCGA、EGA 以及所有后来的适配器，模式的第 7 位表明了在改变模式时清空显示缓冲区。如果第 7 位置 0，就会清空缓冲区。在 CGA、MDA 和 HGA 上第 7 位必须是零，所以总会清空显示器。清屏时，填充文本模式，带有 ASCII 空间，属性 7。图形模式将屏幕缓区全清为零。

调用：　ah=0
　　al,第 7 位=0 清空显示缓冲区
　　1 保持显示缓冲区内容不变
　　第 6~0 位=视频模式（0~7Fh）
返回：　如果模式有效就设置为该视频模式 BIOS 值 40:19h 设置为视频模式

在系统上电时，有彩色显示器的系统会设置为缺省的模式 3。一个单色显示器的系统会设置为缺省的模式 7。这些缺省值在 PC/XT 主板上由开关控制，或者在 AT+系统上由 BIOS 设置程序记录在 CMOS 内存中。

表 9-2 中的显示器段址是显示器缓冲区的实际物理段址。参看中断 10h 功能 FEh 来确定实际的物理显示段址。

表 9-2　视频适配器家族的视频模式

Mode	MDA	CGA	MCGA	EGA	VGA	SVGA	XGA	DOS/V	AX&VGA	水平像素	垂直像素	字符大小	最大页	显示段	描　述
0		x	x	x	x	x	x			320	200	8x8	8	B800	文本,40 列
			x	x	x	x	x			320	350	8x14	8	B800	25 行
			x							320	400	8x16	8	B800	单色
					x	x	x			360	400	9x16	8	B800	彩色关闭
															以合成输出
1		x	x	x	x	x	x			320	200	8x8	8	B800	文本 40 列
			x	x	x	x	x			320	350	8x14	8	B800	25 行 16 色
			x							320	400	8x16	8	B800	
					x	x	x			360	400	9x16	8	B800	
2		x								640	200	8x8	4	B800	文本 80 列
			x	x	x	x	x			640	200	8x8	8	B800	25 行单色
			x	x	x	x	x			640	350	8x14	8	B800	彩色关闭以
			x							640	400	8x16	8	B800	合成输出
					x	x	x			720	400	9x16	8	B800	
							x			640	475	8x19	1	none	
3		x								640	200	8x8	4	B800	文本,80 列
			x	x	x	x	x			640	200	8x8	8	B800	25 行 16 色
			x	x	x	x	x			640	350	8x14	8	B800	
			x							640	400	8x16	8	B800	
					x	x	x			720	400	9x16	8	B800	
								x		640	475	8x19	1	none	
4		x	x	x	x	x	x			320	200	8x8	1	B800	图形,4 色
5		x	x	x	x	x	x			320	200	8x8	1	B800	图形，单色
															（彩色关闭）
6		x	x	x	x	x	x			640	200	8x8	1	B800	图形，2 色
7	x									720	350	9x14	1	B000	文本，80 列
				x	x	x	x			720	350	9x14	8	B000	25 行，单色
					x	x	x			720	400	9x16	8	B000	
8-C															Pcur 和无效模式
D				x	x	x	x			320	200	8x8	8	A000	图形，16 色
E				x	x	x	x			640	200	8x8	4	A000	图形，16 色
F				x	x	x	x			640	350	8x14	2	A000	图形，2 色
10h				x	x	x	x			640	350	8x14	2	A000	图形，16 色
11h			x		x	x	x	x		640	480	8x16	1	A000	图形，2 色

续表

Mode	MDA	CGA	MCGA	EGA	VGA	SVGA	XGA	DOS/V	AX& VGA	水平像素	垂直像素	字符大小	最大页	显示段	描 述
12h					x	x	x	x		640	480	8x16	1	A000	图形，16 色
13h			x		x	x	x			320	200	8x8	1	A000	图形，256 色
14h							x			640	400	8x16	4	B800	文本,132 列
															25 行,16 色
52h									x	640	480	8x19	1	A000	图形,16 色
															Kanji 显示
															和附加功能
53h									x	640	480	8x19	1	A000	图形,16 色
															Kanji 显示
															和附加功能
6Ah						x				800	600		1	A000	图形,16 色
72h								x		640	480	8x19	1	A000	图形,16 色
73h								x		640	475	8x19	1	none	文本,80 列
															25 行(在图形
															模式下模拟）
100h*						x				640	400		1	A000	图形,256 色
101h*						x				640	480		1	A000	图形,256 色
102h*						x				800	600		1	A000	图形,16 色
103h*						x				800	600		1	A000	图形 256 色
104h*						x				1024	768		1	A000	图形,16 色
105h*						x				1024	768		1	A000	图形,256 色
106h*						x				1024	1024		1	A000	图形,16 色
107h*						x				1024	1024		1	A000	图形,256 色

注：*要求为 SVGA 卡设置专门的模式，使用功能 4Fh 子功能 2

段缓冲区列中的“None”表示 DOS/V DBCS（双字节字符集）驱动程序要求使用中断 10h 功能 FEH 来获得虚拟缓冲区的段址和偏移量。

在所有的 EGA 以及后来的适配器上模式 0 和 2 与 1 和 3 的功能相同，并且支持全彩色。说明得不够之处是供应商如何使用模式 2。对于基于文本的应用程序，几乎每个人都使用模式 3。应用程序使用寄存器 10h 的功能 0Fh 来检查当前的模式。如果设置了模式 2，应用程序应切换到使用黑、白、亮白属性。这时，用户可以简单地设置模式 2 来使应用程序工作在单色模式下。为了在 DOS 系统上设置文本模式，可以在 DOS 提示符下运行 MODE 程序：

设置模式 0，黑白，40 列	MODE BW40
设置模式 1，彩色，40 列	MODE CO40
设置模式 2，黑白，80 列	MODE BW40
设置模式 3，彩色，80 列	MODE CO40

中断	功能	描述	平台
10h	1	设置光标类型	所有

文本模式下，这个功能定义了光标的格内位置以及闪烁的线数。在视频系统中只提供一种光标类型，与活动的页号无关。光标的类型保存在 BIOS 数据区 40:60h 处。

参看功能 12h 子功能 BL=34h，了解 VGA 系统是如何利用设置光标类型改变光标的大小，该大小是相对于字符格大小的。

调用：ah=1
ch=光标在格内的上端线（0~31）
cl=光标在格内的底端线（0~31）
返回：更新光标
ax=大多数情况下未做改变，但是某些 BIOS 会改变其值。

除非关闭了光标类型模拟，否则所有的适配器都假定字符格大小为 8 条扫描线。这包括 EGA/VGA 以后后来的适配器，实际上它们每个字符显示 14 或 16 条扫描线。光标类型的常用值包括：

cx=0607h	两线光标，接近或位于格的底部
cx=0307h	格底部的半框光标
cx=0003h	格顶部的半框或四分之一框光标
cx=0007h	全框光标
cx=0100h	没有光标

中断	功能	描述	平台
10h	2	设置光标位置	所有

设置指定视频页内的光标位置。BIOS 为每页保留了一个独立的光标位置。这些光标位置值保存在 BIOS 数据区开始于 40:50h 的 8 个字中。

调用：ah=2
bh=视频页（设置模式后的缺省页是 0）
dh=行（顶行为 0）
dl=列（最左列为 0）
返回：更新光标位置

中断	功能	描述	平台
10h	3	读光标位置和类型	所有

获得指定视频页内的光标位置和光标类型。BIOS 为每页保留了一个独立的光标位置。这些光标位置值保存在 BIOS 数据区开始于 40:50h 的 8 个字中，参看功能以进一步了解光标的类型。

调用：ah=3
bh=视频页（设置模式后的缺省页是 0）
返回：ch=类型，光标的格内顶线
cl=类型，光标的格内底线
dh=行（顶行为 0）
dl=列（最左列为 0）

中断	功能	描述	平台
10h	4	读取光标位置	CGA、EGA

获取当前笔的位置。从 VGA 开始不再支持光笔。

调用：ah=4
返回：如果没有光笔，没有激活，或者不支持：
ah=0
bx、cx、dx 无意义
如果合法，激活光笔：
ah=1
bx=光笔列（模式 4、5 和 DF 从 0~319）
（模式 6,E~10h 下从 0~319）
ch=光栅线（模式 4~6 下从 0~199）
cx=光栅线（模式 D~F 下从 0~199，仅 EGA）
（模式 F~10h 下从 0~199，仅 EGA）
dh=光笔定位的字符的行
dl=光笔定位的字符的列

中断	功能	描述	平台
10h	5	选择活动的显示页	所有

许多模式下支持多个页面，这个功能选择哪个页面是活动的。这是最快的方法在两个全视频页中切换。可以在任何时候写非活动页而不影响当前活动的显示页。活动页号保存在 BIOS 数据区的 40:62h 处。

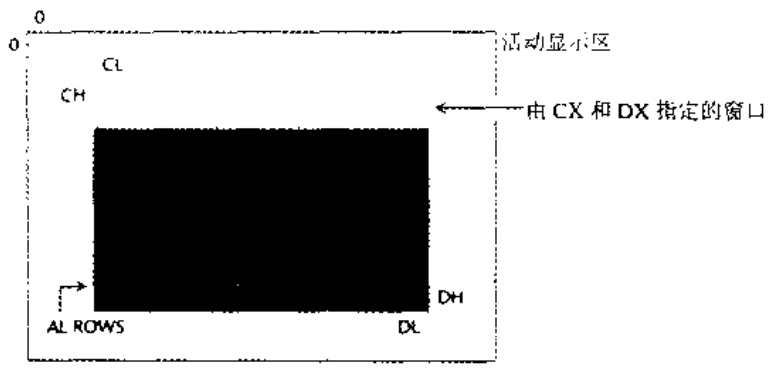
许多 BIOS 并不检查页的有效性，在使用一个越界的值时会发生意想不到的情况。使

用 DOS/V 时，只支持第 0 页。

调用：　　ah=5
　　al=活动页的页号，从零开始计数
返回：　　选择指定的页

中断	功能	描述	平台
10h	6	向下滚动活动页	所有

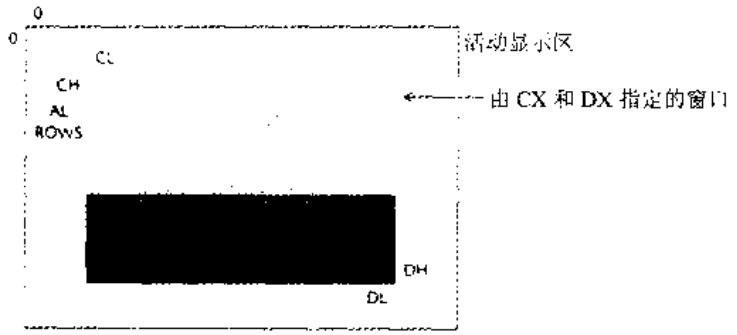
按 AL 指定的行数向上滚动活动页上的窗口区。滚动掉的顶部内容会被丢失。BH 指定了底部新空白行的颜色属性。所有模式，包括图形模式，都支持这个功能。在图形模式下，以每字符扫描线为单位的缺省字符大小指示了每行向上滚动多少扫描线。



调用：　　ah=6
　　al=向上滚动的行数
　　bh=空白底行所使用的属性
　　ch=滚动的顶行（0=最上一行）
　　cl=滚动的左边列（0=最左边一列）
　　dh=滚动的底行
　　dl=滚动的右边列
返回：　　向上滚动指定的屏幕区域

中断	功能	描述	平台
10h	7	向下滚动活动页	所有

按 AL 中指定的行数向下滚动活动页上的窗口区。滚动掉的底部内容会被丢失。BH 指定了顶部新空白行的颜色属性。所有模式，包括图形模式，都支持这个功能。在图形模式下，以每字符扫描线为单位的缺省字符大小指示了每行向下滚动多少扫描线。



调用：　　ah=7
　　al=向下滚动的行数
　　bh=空白顶行所使用的属性
　　ch=滚动的顶行（0=最上一行）
　　cl=滚动的左边列（0=最左边一列）
　　dh=滚动的底行
　　dl=滚动的右边列
返回：　　向下滚动指定的屏幕区域

中断	功能	描述	平台
10h	8	读字符和属性	所有

获取指定页中当前光标位置处的字符和属性。在图形模式下，这个功能的执行速度相当慢。图形模式下如果不同缺省的 ROM 字符表相匹配，那么 AL 被设定为 0。

调用：　　ah=8
　　bh=页号
返回：　　al=读取的字符
　　ah=文本模式下时返回属性，图形模式下无意义

中断	功能	描述	平台
10h	9	写字符和属性	所有

向指定页中当前光标位置处写入字符和属性。这个功能允许对相同的字符和属性重复写入多个字。在图形模式下，重复的次数不允许使被写的字符越过当前行的末尾。

在图形模式 4、5 或 6 下，如果使用的字符号大于等于 80h，那么图形字体将由 0:7Ch 的双字指针指定。应用程序设置该指针。图形字体指针不是由系统 BIOS 的适配卡设置。

当使用 DOS/V 的 DBCS（双字节字符集）驱动文件时，CX 应总设为 1。先写双字节字符的首字节，然后调用程序将光标右移一列，并使用这个功能写入双字符的末字节。为了使双字节字符的两半颜色相同，首字节和末字节属性应相同。

调用：　　ah=9
al=要写的字符（0~255）
bh=页号（模式 13 除外），背景色（0~255 仅模式 B）
bl=如果是文本模式，则为属性
如果图形背景设为 0（黑色）那么是前景色（值可达 15，除了模式 13h）
如果在图形模式下（仅模式 13h），颜色值将和当前位置前景色的当前颜色位相异或，那位第 7 位=1（值为 0~255）
cx=字符重复的次数
返回：　　更新后的屏幕

中断	功能	描述	平台
10h	A	仅写字符	所有

向指定页的当前光标位置写字符。这个功能允许多次重复写同一字符。在图形模式下，推荐使用功能 9。在图形模式下使用功能 A 将和功能 9 的作用相同，也是使用 BL 寄存器中的值来指定颜色，尽管这一点以前尚未说明。

当使用 DOS/V 的 DBCS（双字节字符集）驱动文件时，CX 总应设置为 1。先写双字节字符的首字节，然后调用程序将光标右移一列，并使用这个功能写入双字节字符的末字节。为了使双字节字符的两半颜色相同，首字节和末字节属性应相同。

调用：　　ah=Ah
al=要写的字符（0~255）
bh=页号
cx=字符重复的次数
返回：　　更新屏幕

中断	功能	描述	平台
10h	B	设置调色板	所有

这个功能用于设置模式 0 到 6 上的颜色选项。EGA 以及后来的适配器使用功能 10h 来提供许多先进的颜色选项。

如果当前位于文本模式 0~3 下，该选项还支持设置边界颜色。另外，它支持改变闪烁位第 7 位以提供高度背景色。当设置了闪烁位以后，字符要么是闪烁的，要么是将普通 8 种背景色扩展为 16 色。如果使用图形模式 4、5 或 6，这个功能可选择要使用的颜色。

EGA 适配卡在很少使用的模式 0、1、4 或 5 下不会显示边界。如果使用了 DOS/V，也不会支持这个功能。

调用：　　ah=Bh
bh=0（文本模式 0~3）
bl,第 0~3 位=边界色（0~15）
第 4 位=0 闪烁选项
=1 高度背景选项
0（图形模式 4 或 5）
bl=背景色（0~15）
0（图形模式 6）
bl=背景色（0~15）
1（图形模式 4 或 5）
bh=0 前景色绿/红/紫
1 前景色青/洋红/白
返回：　　更新像素

中断	功能	描述	平台
10h	C	写点	所有

这个功能在一个指定的地址写一个像素点。这是写图形相当于慢的一种方法。你应该考虑直接向显示器内存写。它支持所有的图形模式。这个功能不能用于文本模式，因为那样的话会改变字符或属性信息。

调用：　　ah=Ch
al=颜色值（除 13h 外的所有图形模式）
bl,第 0~3 位=颜色（0~15，取决于模式所支持的颜色范围）。
第 7 位=0 替换像素
1 异或当前像素颜色值（0~255，仅模式 13h）
bh=页号（如果只允许 1 页，那么会忽略该值）
cx=列号（0~639 或更多，取决于模式）
dx=行号（0~479 或更多，取决于模式）
返回：　　更新更素

中断	功能	描述	平台
10h	D	读点	所有

这个功能在指定的地址读取一个像素点。这是读图形相当慢的一种方法。你应该考虑直接从显示器内存读。它支持所有的图形模式。这个功能不应该用在文本模式下，因为返回值无任何意义。

调用：　　ah=Dh
bh=页号（如果只允许 1 页，那么会忽略该值）

cx=列号（0~639 或更多，取决于模式）
dx=行号（0~479 或更多，取决于模式）
返回：al=指定地点的像素颜色

中断	功能	描述	平台
10h	E	以电传模式写	所有

这个功能在活动显示页的当前光标位置向显示器发送一个字符。如果必要，它自动换行、翻转以及为特定动作解释某些控制字符。它支持文本和图形模式。

仅 7、8、A 和 D 被解释为命令的字节，它们执行下列动作：

值	名称	描述
7	铃	使用时钟 2 生成 1/2 秒的延时
8	回退	光标后退一列，除非光标位于第 0 列，这时会忽略该命令
A	前进一行	光标前进一行，除非光标位于最后一行，这时屏幕会向上翻滚一行
D	返回	光标回到第 0 列

在文本模式下向上翻滚时，新行的属性与光标以前所在的位置相同。图形模式下，新行以黑色作为背景色。

调用：ah=Eh
al=要写的字符
bh=活动页号（只有老式的 PC BIOS 才要求，所有的 XT+BIOS 都忽略它）
bl=前景色（仅图形模式）
返回：显示字符或执行命令。

中断	功能	描述	平台
10h	F	读当前视频状态	所有

获取当前的视频模式、列数以及活动页号。从 BIOS 数据区获得这些值。列数保存在 40:49 处，而活动页保存在 40:62h 处。DOS/V 在任何模式下只支持一页。

调用：ah=Fh
返回：ah=屏幕上的文本列数
al=视频模式
位 7=0 如果上次模式设置清除了视频缓冲区
1 如果上次模式设置没有改变视频缓冲区
位 6~0=视频模式号（参看表 9-2）
bh=活动页

中断	功能	描述	平台
10h	10	设置颜色寄存器功能	MCGA/EGA/VGA+

这个功能提供了许多子功能来控制高级适配器上的各种调色板。MCGA 标准对这个子功能有许多限制。

参看功能 12h、子功能 31h，了解如何锁住颜色寄存器，这样模式设置就不会改变颜色寄存器。还可以参看子功能 12h、子功能 33h，以了解如何强制设置寄存器，以便自动转换为 VGA 或更好适配器上的灰度级。

表 9-3 显示了每种属性的标准颜色。在一个 EGA 及后来的适配器上，每种属性颜色都可以改变成表 9-4 中的 64 种颜色中的一种。对于 VGA 或更好的适配器，每种属性颜色可以描述为 18 位颜色信息。表 9-4 所描述的信息可能会受显示器以及显示器设置的影响。它们仅供参考，并且假定已经调节好了显示器。

表 9-3 标准颜色值

色彩寄存器	16 进制值	描　　述
0	0	黑
1	1	蓝
2	2	绿
3	3	青
4	4	红
5	5	洋红
6	14	褐
7	7	白
8	38	灰
9	39	亮蓝
10	3A	亮绿
11	3B	亮青
12	3C	亮红
13	3D	亮洋红
14	3E	亮黄
15	3F	亮白

表 9-4 颜色描述，64 色

值	描　　述	值	描　　述
0	黑	20	深红
1	蓝	21	深蓝紫
2	绿	22	绿
3	青	23	青
4	红	24	明亮的橘红色
5	洋红	25	深粉红色
6	暗黄	26	桔黄
7	白	27	粉红
8	按蓝	28	深紫红色
9	中等蓝	29	中等蓝
A	军绿	2A	灰绿
B	淡蓝	2B	中等蓝
C	红宝石色	2C	桃红
D	淡紫色	2D	深洋红
E	明亮的金色	2E	亮桔黄
F	明亮的淡紫色	2F	亮淡紫
10	深绿	30	深军绿
11	中深绿	31	蓝紫
12	荧光绿	32	亮绿蓝
13	青绿	33	亮青绿
14	褐色	34	桔红
15	中等紫红色	35	热粉红
16	明亮的黄绿色	36	柠檬黄
17	暗绿	37	暖白
18	深青	38	灰
19	深蓝	39	亮蓝
1A	亮绿	3A	亮绿
1B	亮青	3B	亮青
1C	暗红	3C	亮红
1D	紫红	3D	亮洋红
1E	明亮的黄绿色	3E	亮蓝
1F	冰蓝	3F	亮白

尽管在下面的子功能列表中忽略了某些子功能，但是在EGA/VGA适配器上一般未使用它们，并且它们仅仅只是返回而不做任何事。将来的适配器可能会使用这些现在未使用的功能。

AL=	设置颜色寄存器的子功能	适配器/驱动文件
0	设置单个颜色寄存器	VGA+、EGA、MCGA、DOSV
1	设置边界色	VGA+、EGA
2	设置所有的颜色寄存器和边界	VGA+、EGA、DOSV
3	背景闪烁或增强	VGA+、EGA、MCGA
7	读单个颜色寄存器	VGA+、DOSV
8	读边界颜色	VGA+
9	读所有的颜色寄存器和边界	VGA+、DOSV
10h	设置单个18位颜色寄存器	VGA+、MCGA
12h	设置多个18位颜色寄存器	VGA+、MCGA
13h	选择颜色页面	VGA+
15h	读单个18位颜色寄存器	VGA+、MCGA
17h	读多个18位颜色寄存器	VGA+、MCGA
18h	写VGA像素屏蔽寄存器	VGA+
19h	读VGA像素屏蔽寄存器	VGA+
1Ah	读颜色页面状态	VGA+
1Bh	将颜色转化为灰度	VGA+、MCGA

功能10h的子功能详述

子功能	描述	中断	功能
AL=0	设置单个颜色寄存器	10h	AH=10h

在EGA以及VGA+适配器上，该子功能用于将16色寄存器设定为64种颜色中的一种。表9-3列出了16个寄存器的最常见的缺省值。其他的值用于某些图形模式，某些视频供应商在他们的硬件上将紫色调整得更加令人赏心悦目。表9-4列了这64色选择。

IBM建议在MCGA适配卡只使用BX值712h。这将16色换为8种不变色。其他的适配器使用这种专用值来将高8种颜色锁成为和低8种相同的颜色。锁住高8种颜色主要用在使用512字符字体时，这样在整个字体上颜色都保持不变。设置视频模式可以让系统返回到普通的16色模式。

调用：　ax=1000h
　　bl=设置的寄存器号（0~15,18）

bh=新值（0~63）
返回：　更新后的颜色寄存器

子功能	描述	中断	功能
AL=1	设置边界色	10h	AH=10h

将边界色设置为表 9-4 所列的 64 色之一。MCGA 不支持这个功能。VGA 适配器在不常使用的的模式 0、1、3、4 或 D 下不能显示边界。

调用：　ax=1001h
bh=新值（0~63）
返回：　更新后的边界颜色

子功能	描述	中断	功能
AL=2	设置所有的寄存器和边界	10h	AH=10h

为所有的 16 色寄存器和边界颜色设置颜色。每个寄存器选自表 9-4 所列出的 64 色之一。MCGA 不支持这个子功能。VGA 适配器在不常使用的模式 0、1、4 或 D 下不能显示边界。

调用：　ax=1002h
es:bx=指向 17 字节颜色表的指针
字节 0 到 F 指定了颜色寄存器
字节 10h 指定了边界色
返回：　更新后的颜色寄存器和边界

子功能	描述	中断	功能
AL=3	背景闪烁或加强	10h	AH=10h

在文本模式下，属性的第 7 位通常用来控制背景是闪烁的还是非闪烁的。它也可用来将普通的 8 种背景色扩展为 16 色。这个功能选择如何让属性的第 7 位来影响背景。

调用：　ax=1003h
bl=0 将属性位 7 用作增强色
1 将属性位 7 用作闪烁
返回：　保存后的属性控制

子功能	描述	中断	功能
AL=7	读单个颜色寄存器	10h	AH=10h

读取一个 16 色寄存器。表 9-4 列出了颜色值。只有 VGA 以及更先进的系统才支持这个子功能。

调用：　　ax=1007h
　　bl=待读取的寄存器号（0~15）
返回：　　bl=寄存器值（0~63）

子功能	描述	中断	功能
AL=8	读边界色	10h	AH=10h

获取当前的边界色。表9-4 列出了颜色值。只有VGA 以及更高级的系统才支持这个子功能。

调用：　　ax=1008h
返回：　　bh=边界颜色值（0~63）

子功能	描述	中断	功能
AL=9	读取所有的颜色寄存器和边界	10h	AH=10h

读取所有的 16 色寄存器，以及边界颜色寄存器。表 9-4 列出了颜色值。只有 VGA 以及更高级的系统才支持该这个子功能。

调用：　　ax=1009h
　　es:bx=指针，指向 17 字节颜色表的存储区域
　　字节 0 到 F 保存颜色寄存器
　　字节 10h 保存边界颜色
返回：　　被装入到表中的 17 个颜色字节

子功能	描述	中断	功能
AL=10h	设置单个 18 位颜色寄存器	10h	AH=10h

在 VGA+与 MCGA 适配器上，这个功能用于设置 64 个 18 位颜色寄存器中的一个。对于 VGA+/MCGA，每一种表中所列的64 色都由三个 6 位寄存器控制，这三个寄存器分别控制绿、红和蓝的浓度。这一点允许将任何一种64 EGA 类型色改为用户自定义色。EGA 不支持这个子功能。

调用：　　ax=1010h
　　bx=要设置的寄存器号（0~63）
　　ch=绿色值（0~63）
　　al=蓝色值（0~64）
　　dh=红色值（0~63）
返回：　　更新后的颜色寄存器

子功能	描述	中断	功能
AL=12h	设置多个 18 位颜色寄存器	10h	AH=10

在 VGA+及 MCGA 适配器上，这个功能用于设置多个 18 位颜色寄存器。对于 VGA+/MCGA，每一种表中所列出的 64 色都由三个 6 位寄存器控制，这三个寄存器分别控制绿、红和蓝的浓度。这一点允许将任何一组 64 EGA 类型色改为用户自定义色。EGA 不支持这个子功能。

调用：ax=1012h
bx=要设定的第一个颜色寄存器（0~63）
cx=要设置的颜色寄存器的数目（1~64）
es:bx=指向一色表的指针——这个表由三种颜色组成，每个寄存器的字节按红、绿、蓝排列（每个颜色值的范围为 0~63）
返回：更新后的颜色寄存器

子功能	描述	中断	功能
AL=13h	选择颜色页	10h	AH=10h

VGA+ 提供多页 18 位寄存器，可以在它们之间切换，在页间切换比重新装入所有的寄存器要快得多。游戏常常使用这一点来产生特殊的效果，如果组建得巧妙，还可以产生有限的动画效果。模式 13h 只有一张颜色页，所以在激活模式 13h 时这个功能无效。

无论何时中断 10h 的功能 0 设置了某个模式，系统缺省的情况是页模式有四张颜色页面，并激活第一页。只初始化第一页 64 色寄存器。

调用：ax=1013h
bl=0 选择页面模式
bh=0 使用四页，每一页有 64 个颜色寄存器（上电后的缺省值）
bh=1 使用 16 页，每页有 16 个颜色寄存器
bl=1 选择颜色页号
bh= 颜色号
如果在 4 色页面模式下，页面范围是 0~3
如果在 16 色页面模式下，页面范围是 0~15
返回：选择的颜色页或模式

子功能	描述	中断	功能
AL=15h	读取单个 18 位颜色寄存器	10h	AH=10h

在 VGA+和 MCGA 适配器上，这个功能用于读取 64 个 18 位颜色寄存器中的一个。EGA 不支持该子功能。

调用：　　ax=1015h
　　bx=要读取的寄存器号（0~63）
返回：　　ch=绿色值（0~63）
　　cl=蓝色值（0~63）
　　dh=红色值（0~63）

子功能	描述	中断	功能
AL=17h	读取多个18位颜色寄存器	10h	AH=10h

在VGA+和MCGA适配器上，这个功能用于读取多个18位颜色寄存器。EGA不支持该子功能。

调用：　　ax=1017h
　　bx=要读取的第一个颜色寄存器（0~63）
　　cx=要读取的颜色寄存器的数目（1~64）
　　es:bx=指向三色表的指针，这个表为每个寄存器保存了三种颜色，
　　按顺序为红、绿、蓝。
返回：　　装入了指定的颜色寄存器值的es:bx表。

子功能	描述	中断	功能
AL=18h	写VGA像素屏蔽寄存器	10h	AH=10h

这个未正式说明的功能仅仅只是在端口3C6h处装入VGA像素屏蔽寄存器。在BIOS模式设置期间该像素屏蔽寄存器中装入值FFh。好象大多数的BIOS供应商都支持子功能18h，但是视频BIOS却从未调用过它。这个像素屏蔽寄存器在VGA上也是一个未正式说明的I/O端口。

像素屏蔽寄存器能够用来提供最快的颜色变化。VGA有一个内部的256色表。尽管大多数模式只使用前面的16种颜色。如果要显示其中一种颜色，可由硬件选择该颜色的地址。硬件将表地址和像素屏蔽寄存器的内存相“与”。对于缺省的像素屏蔽值FFh而言，可显示所有的256色。像素屏蔽寄存器可以限制对颜色寄存器的访问，而不影响视频内存和颜色寄存器。

例如，如果装入了缺省的颜色，表入口0和1可用于访问黑色和蓝色。在像素屏蔽寄存器中装入1，意味着只有支持属性0（黑）、1（蓝）。任何其他属性在屏幕上要么显示为黑色要么就显示为蓝色。如果属性为青，表中地址为3，该地址与像素屏蔽寄存器相“与”的结果是在屏幕上显示蓝色，而不是青色。如果像素屏蔽值为5，那么屏幕上只显示黑、蓝、红和洋红。

调用：　　ax=1018h
　　bl=寄存器值

返回：　　装入后的像素屏蔽寄存器

子功能	描述	中断	功能
AL=19h	读 VGA 像素屏蔽寄存器	10h	AH=10h

这个未正式说明的功能读取未正式说明的在端口 3C6h 处的 VGA 像素屏蔽寄存器。似乎大多数的 BIOS 供应商都支持子功能 19h，但是视频 BIOS 却从未调用过它。参看子功能 18h 获取其他细节。

调用：　　ax=1019h
返回：　　bh=0
　　bl=像素屏蔽寄存器的内容

子功能	描述	中断	功能
AL=1Ah	读取颜色页面状态	10h	AH=10h

获取颜色页面模式及活动的颜色页。参看子功能 13h 获取更多细节。只有 VGA+适配器才支持该子功能。

调用：　　ax=101Ah
返回：　　bh=活动的颜色页（0~15）
　　bl=颜色页模式（0~1）

子功能	描述	中断	功能
AL=1Bh	将颜色转化为灰度	10h	AH=10h

该子功能从 18 位颜色寄存器中取出一个值并将之转化为灰度值。它取得每种颜色的百分比并将它们加在一起。然后将结果装入到 18 位颜色值的红、绿、蓝部分中。下面的例子有助于说明 VGA+适配器是如何在一个寄存器上执行这种转化的。下面所示的适配器的百分比权值是固定的。

起始颜色	值	颜色权值	结果	最终颜色值
蓝	58	11%	6	→ 33
绿	26	59%	15	→ 33
红	39	30%	12	→ 33
			——	
			和=33	

调用：　　ax=101Bh
　　bx=需要求和的第一个颜色寄存器（0~63）
　　cx=需要求和的颜色寄存器的数目（1~64）
返回：　　将指定的颜色寄存器转化为灰度级

子功能	描述	中断	功能
10h	11	字符生成功能	MCGA/EGA/VGA+

这个功能提供了许多子功能来设置与获取 MCGA 和 VGA+适配器上的字符生成属性。不同的适配器有点差别，这在每个子功能中均已指出。

尽管下面的子功能列表忽略了某些子功能，但是在 EGA/VGA 适配器上一般并没有使用它们，仅仅只是让它们返回而不做任何事。将来的适配器可能会使用这些现在未用的子功能。

AL=	字符生成器子功能	适配器 / 驱动程序
0	装入用户字体	VGA+、EGA、MCGA、DOSV
1	装入 8×14 字体	VGA+、EGA、MCGA
2	装入 8×8 字体	VGA+、EGA、MCGA
3	选择字体模式	VGA+、EGA、MCGA
4	装入 8×16 字体	VGA+、MCGA
10h	在设置模式后装入用户字体	VGA+、EGA、MCGA
11h	在设置模式后装入 8×14 字体	VGA+、EGA、MCGA
12h	在设置模式后装入 8×8 字体	VGA+、EGA、MCGA
14h	在设置模式后装入 8×16 字体	VGA+、MCGA
20h	设置中断 1Fh 图形字体指针	VGA+、EGA、MCGA
21h	设置中断 43h 图形字体指针	VGA+、EGA、MCGA
22h	向中断 43h 中装入 8×14 字体	VGA+、EGA、MCGA
23h	向中断 43h 中装入 8×8 字体	VGA+、EGA、MCGA
24h	向中断 43h 中装入 8×16 字体	VGA+、EMCGA
30h	获取字体信息	VGA+、EGA、MCGA

功能 11h 的子功能详述

子功能	描述	中断	功能
AL=0	装入用户字体	10h	AH=11h

在适配器字体内存中装入用户指定的字体字符。在适配器内部有许多 32×256 字体页。一个字体页中的每一个 32 字节字体块描述了每个文本字符所使用的位，大多数情况下，只需要 32 字节的一部分就可以组成完整的字符。例如，EGA 的标准 8×14 字体由 14 个字节

组成。字符“T”如下所示：

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

如果需要装入一个用户字体，只需要提供组成字符格的每个字符的字节数。对于 8×14 字体，用户字体表有 14×256 个字节。九位宽字体每行只使用 8 位，而由视频适配器硬件生成第九列。对于字符 0~BFh 和 E0~F0h，第九列总是空白。对于字符 C0~DFh，第 9 列复制第 8 列。这样，当线图形字体字符相邻地放在屏幕上时看起来就是连续的。

所有文本模式下的字体属性都保存在红色面板内，在高分辨率图形内存区的 2 号面板内。切换到图形模式 Dh 或更高模式时会清除字体信息，切换到文本模式会重新设定缺省字体。在图形模式下装入字体会在空白图形屏幕的上半部分生成红色的点。

如果我们已经提供了 256 个字符的 8×14 字体表，并且希望只重装一个字符“T”，寄存器可以如下设置：BH=14、BL=0、CX=1、DX=‘T’、ES:BP=表头位置。本例中字体表长度是 14×256 或 3584 字节，尽管只用到组成“T”的 14 个字节。

这个功能不设定屏幕上的扫描线的数目。参看子功能 30h 来实现这一点。屏幕上的行数等于扫描线的数目除以每个字符的行数。DOS/V 的功能 11h 只支持这个子功能。DOS/V 只支持一页。

调用：

ax=1100h
bh=每个字符的字节数（1~32）
bl=字节页
cx=要装入的字体字符数
dx=表中的字符号（0~255）

es:bp=用户字符表的起点
返回: 更新后的字体

子功能	描述	中断	功能
AL=1	装入8×14字体	10h	AH=10h

向指定的字体页中装入适配器的内部的8×14字体。将适配器重设为使用8×14字体。MCGA适配器不会显示8×14字体，如果试图执行了子功能1，就会触发子功能来显示8×16字体。

调用: ax=1101h
bl=字体页
返回: 装入8×14字体

子功能	描述	中断	功能
AL=2	装入8×8字体	10h	AH=11h

向指定的字体页中装入适配器的内部的8×8字体。重设适配器为8×8字体。

调用: ax=1102h
bl=字体页
返回: 8×8字体页

子功能	描述	中断	功能
AL=3	选择字体页模式	10h	AH=11h

该子功能指定活动的字体页。最多可同时激活两页字体，同时在屏幕上生成最大可能的512个字符。EGA和MCGA总共提供了四页字体，VGA及后来的适配器提供了八页字体。

访问512字符时，属性字节的第3位通常用来控制前景的色彩浓度。在512字符模式下，它还控制应该显示哪两页字体。为了获取八种不变色，有必要改变颜色寄存器，这样前八个寄存器和第二组八个寄存器相同。参看中断10h，功能AL=10h，子功能0，以了解实现这一点的特殊模式。

表9-5显示了基于属性位第3位以及EGA和MCGA的字体页面模式值（BL）是如何选择页面的。如果所选页的属性浓度的状态相同，那么只有256个字符可能被访问到。例如，如果字体页面模式设置为8，那么允许512个字符，当属性第3位是0时使用第0页字体。如果属性第3位是1，那么使用第2页字体。表9-6显示了VGA+适配器上在所有可能的字体模式下的页面选择。

表 9-5　字体页面选择，MCGA 和 EGA

页面模式	选择页面的属性		页面模式	选择页面的属性	
	位 3=0	位 3=1		位 3=0	位 3=1
0	0	0	8	0	2
1	1	0	9	1	2
2	2	0	A	2	2
3	3	0	B	3	2
4	0	1	C	0	3
5	1	1	D	1	3
6	2	1	E	2	3
7	3	1	F	3	3

表 9-6　字体页面的选择，VGA+

页面模式	选择页面的属性		页面模式	选择页面的属性	
	位 3=0	位 3=1		位 3=0	位 3=1
0	0	0	20	0	4
1	1	0	21	1	4
2	2	0	22	2	4
3	3	0	23	3	4
4	0	1	24	0	5
5	1	1	25	1	5
6	2	1	26	2	5
7	3	1	27	3	5
8	0	2	28	0	6
9	1	2	29	1	6
A	2	2	2A	2	6
B	3	2	2B	3	6
C	0	3	2C	0	7
D	1	3	2D	1	7
E	2	3	2E	2	7
F	3	3	2F	3	7
10	4	0	30	4	4
11	5	0	31	5	4
12	6	0	32	6	4

续表

页面模式	选择页面的属性		页面模式	选择页面的属性	
	位 3=0	位 3=1		位 3=0	位 3=1
13	7	0	33	7	4
14	4	1	34	4	5
15	5	1	35	5	5
16	6	1	36	6	5
17	7	1	37	7	5
18	4	2	38	4	6
19	5	2	39	5	6
1A	6	2	3A	6	6
1B	7	2	3B	7	6
1C	4	3	3C	4	7
1D	5	3	3D	5	7
1E	6	3	3E	6	7
1F	7	3	3F	7	7

调用：　　ax=1103h
　　bl=字体页面模式（参看文中所述）
返回：　　设置字体页面的模式

子功能	描述	中断	功能
AL=4	装入8x16字体	10h	AH=11h

向指定的字体中装入适配器的内部的8×16字体。重设适配器为8×16字体。只有MCGA和VGA+适配器才支持这个子功能，而EGA不支持。

调用：　　ax=1104h
　　bl=字体页
返回：　　装入8×16字体

子功能	描述	中断	功能
AL=10h	设置模式后装入用户字体	10h	AH=11h

这个子功能像子功能0那样装入用户自定义字体。与子功能0不同的是，只有在设置模式后才可以立即触发这个子功能，它可以执行许多操作。必须激活页面0（模式设置总是激活页面0）。保存在BIOS数据区地址40:84h处的行数由下列整数值确定：

最大行（从零开始算起）= 扫描线数 / 字符高度 - 1

例如，如果装入了一个 8×20 字体，如果扫描线设置为 400，那么结果会是（400/20）-1=19。因为是从零开始的，所以 19 实际上意味着有 20 行。

每页中的字节数经计算后保存在 BIOS 数据区地址 40:4Ah 处。每页中的字节数由下式计算：

每页字节数=（ 行数 +1 ）× 列数 × 2

某些通过端口 3D5h 访问的 CRT 控制寄存器，会被更新为新的值。这包括：

最大扫描本寄存器	(下标 9)=字符高度-1
光标起点寄存器	(下标 A)=字符高度-2
光标终点寄存器	(下标 9)=字符高度-1
垂直显示终点寄存器	(下标 12)=
	对 200 条扫描线寄存器： ((行+1)×字符高度×2)-2 对 250 条扫描线寄存器： ((行+1)×字符高度)-1

如果系统现在位于单色模式 7 下，那么还会更新 CRT 控制器的下划线寄存器。

下划线位置寄存器（下标 14h）=字符高度-1

在 MCGA 中，并不会执行这个子功能。如果试图执行这个子功能，就会触发子功能 0。

调用：	ax=1110h
	bh=每字符探险节数（2~232）
	bl=字体页面
	cx=要装入的字体字符数
	dx=表中的字符号（0~255）
	es:bp=用户字体表的起点
返回：	更新后的字体

子功能	描述	中断	功能
AL=11h	设置模式后装入 8×14 字体	10h	AH=11h

向指定的字体页中装入适配器的内部的 8×14 字体。这一点类似于功能 1，只是在触发功能 11h 之前必须先设置模式。参看子功能 10h 了解相关操作。

在设置了文本视频模式之后，该子功能用于设置一个 8×14 字体。MCGA 适配器不能显示 8×14 字体，如果试图使用子功能 11h，会触发子功能 4 来显示 8×16 字体。

调用：　　ax=1111h
　　bl=字体页面
返回：　　装入 8×14 字体。

子功能	描述	中断	功能
AL=12h	设置模式之后装入 8×8 字体	10h	AH=11h

向指定的字体页中装入适配器的内部 8×8 字体，这一点类似于功能 2，只是在触发功能之前必须先设置模式。参看子功能 10h 了解相关动作。

该子功能用于设置视频模式之后的 8×8 ROM 字体。MCGA 适配器不能实现该子功能，如果试图使用子功能 12h，会触发子功能 2。

调用：　　ax=1112h
　　bl=字体页面
返回：　　装入 8×8 字体

子功能	描述	中断	功能
AL=14h	模式设置后装入 8×16 字体	10h	AH=11h

向指定的字体页中装入适配器的内部的 8×8 字体，这一点类似于功能 4，只是在触发功能 14 之前必须先设置模式。参看功能 10h 了解相关动作。

该子功能用在设置了视频模式之后的 8×16 ROM 字体。只在 VGA+适配器才支持该子功能。EGA 不支持。在 MCGA，不能使用该子功能，如果试图使用，会触发子功能 4。

子功能	描述	中断	功能
AL=20h	设置中断 1Fh 图形字体指针	10h	AH=11h

将指针保存到用户提供的图形字符表中。这个表只用于图形模式 4、5 和 6 的字符 128 到 255。对于一个 8x8 格字符，这个表的配置为 128 个 8 字节字符。

该指针保存在中断向量 1Fh 地址 0:7Ch 中。这个功能仅仅只是将用户指针装入到中断 1Fh 的向量地址中。

调用：　　ax=1120h
　　es:bp=指针，指向用户提供的字符 128~255 的字体表
返回：　　装入指针

子功能	描述	中断	功能
AL=21h	设置中断 43h 图形字体指针	10h	AH=11h

保存指向图形字符表的所提供的指针。在模式 4、5 或 6 下，该指针指向前 128 个字符

的字体表。参看子功能 20h 了解如何处理前 128 个字符。在其他所有的模式下，该指针标识了要使用的全部 256 字体表。

该指针保存在中断向量 43h 的地址 0:10Ch 中，每字符的扫描线数目保存在 BIOS 数据区 40:85h 处，显示行数据保存在 40:84h 处。这个功能仅仅只是将指定的值装入内存中，并且只能在设置图形视频模式后立即使用。

调用：ax=1121h
bl=字符行选择号
0=由 DL 指定行数
1=14 行
2=25 行
3=43 行
cx=每字符的字节数（1~32）
dl=行数（如果 bl=0）
es:bp=指向 ROM 或用户提供的字体表的指针

返回：装入指针并更新 BIOS 值。

子功能	描述	中断	功能
AL=22h	在中断 43h 中装入 8×14 字体	10h	AH=11h

这个功能简单地将一个指向 8×14 ROM 字体的指针装入到中断 43h 向量地址 0:10Ch 中。它等同于子功能 21h 并且将 CX 设置为 14，ES:BP 指向 8×14 ROM 字体。BL 寄存器中设定了行数。

这个功能仅仅只是向内存中装入指定的值。在设置了图形视频模式后立即执行该子功能。

MCGA 适配器不能显示 8×14 字体，如果试图执行子功能 22h，那么会触发子功能 24h 来装入 8×16 字体指针。

调用：ax=1122h
bl=字符行数选择
0=由 DL 指定行数
1=14 行
2=25 行
3=43 行
dl=行数（如果 bl=0）

返回：装入 8×14 指针，更新 BIOS 值

子功能	描述	中断	功能
AL=23h	在中断 43h 中装入 8×8 字体	10h	AH=11h

这个功能简单地将一个指向 8×8 ROM 字体的指针装入到中断 43h 向量地址 0:10Ch 中。它等同于子功能 21h 并且 CX 设置为 8，ES:BP 指向 8×8 ROM 字体。在 BL 寄存器中设定了行数。

这个功能仅仅只是向内存中装入指定的值。在设置了图形视频模式后立即执行该子功能。

调用：ax=1123h
bl=字符行数选择
0=由 DL 指定行数
1=14 行
2=25 行
3=43 行
dl=行数（如果 bl=0）
返回：装入 8×8 指针，更新 BIOS 值

子功能	描述	中断	功能
AL=24h	在中断 43h 中装入 8×16 字体	10h	AH=11h

这个功能简单地将一个指向 8×16 ROM 字体的指针装入到中断 43h 向量地址 0:10Ch 中，它等同于子功能 21h 并且 CX 设置为 8，ES:BP 指向 8×8 ROM 字体。BL 寄存器中设定了行数。

这个功能仅仅只是向内存中装入指定的值。在设置了图形视频模式后立即执行该子功能。仅 MCGA 和 EGA+适配器才支持该子功能，EGA 不支持。

调用：ax=1124h
bl=字符行数选择
0=由 DL 指定行数
1=14 行
2=25 行
3=43 行
dl=行数（如果 bl=0）
返回：装入 8x16 指针，更新 BIOS 值

子功能	描述	中断	功能
AL=30h	获取字符信息	10h	AH=11h

获取指定字体的信息，并获取每字符的当前字节数以及行数。从 BIOS 数据区 40:85h 中取出每字符的当前数值。从 40:84h 中直接读取当前字符行数值，它代表的是屏幕上的文本行数，最少为 1 行。这两个值与所选的字体表无关。

字体表的格式是每一个8位行的字符格占一个字节。一个8×16字体长度为16×256，或者4096字节。

使用两个特殊的9位替换表来指定一系列的字符以替换一个8位宽字体。例如，小写“m”字符在9×14字体比它在8×14字体中要宽。并不用保存完全独立的9×14字体而浪费ROM空间，只须保存少数几个不同的字体。大约一般只须提供20个字符来替换标准的8位宽字符。

这些替换表是这样组织的：先用一个字节代表字符值，然后跟着组成该字符的字节。对于一个9×14字体，每个字符使用15个字节。字符值0用于表示表的结尾。BIOS不能通过一个BIOS调用来接受这些表。必须一次性使用子功能0或10h装入字符。

EGA不支持BH的值6和7，MCGA不支持BH值5和7。MCGA的功能BL=2会返回一个指向8×16字体的指针。

调用：　ax=1130h
　　bh=要返回的字体表指针
　　0=获取保存在中断1Fh中的指针
　　1=获取保存在中断43h中的指针
　　2=8×14 ROM字体（字符0~255）
　　3=8×8 ROM字体（字符0~255）
　　4=8×8 ROM字体（字符128~255）
　　5=以9×14替换8×14字符的特殊字符表
　　6=8×16 ROM字体（字符0~255）
　　7=以9×16替换8×16字符的特殊字符表

返回：　cx=每字符的当前字节数
　　dx=当前字符行数-1
　　es:bp=指向字体表的指针

子功能	描述	中断	功能
10h	12	多种功能	MCGA/EGA/VGA+

这个功能支持许多不能很好归类的子功能。这个功能有时被称作替换选择（Alternate Select），因为一个子功能选择一个可替换的屏幕打印功能。

安装了DOS/V之后，就不支持下面任何一个子功能。

AL=	多种子功能	适配器
10	获取信息	VGA+、EGA、MCGA
20	安装可替换的打印屏幕	VGA+、EGA、MCGA
30	为文本模式选择扫描线数	VGA+、MCGA
31	颜色寄存器装入选项	VGA+、MCGA

32	视频开放/禁止	VGA+、MCGA
33	将颜色转化为灰度级	VGA+、MCGA
34	光标大小控制	VGA+
35	显示开关	VGA+、MCGA
36	视频屏幕开/关	VGA+
37	主框交互支持	XGA+

子功能	描述	中断	功能
BL=10h	获取信息	10h	AH=12h

该子功能获取以前保存在BIOS数据区的信息。颜色类型从40:87的第1位获得。内存的大小从40:87的第6位和第5位获得。开关和特性连接器位保存在40:88中，其高四位是属性连接器位，在返回到CH之前先下移4位。在表9-7中所显示的开关设置控制着在启动之时如何配置适配器。某些适配器利用不变内存来保持这些开关设置，并且需要一个字装程序去改变这些值。对于所有的EGA+兼容适配器，这个功能返回的这些开关值将会与早期的EGA/VGA开关设置保持一致，即使是没有提供任何开关或者这些开关用作其他用途。

表9-7 早期的EGA+适配器开关

位3210	主配置（缺省）	次配置
0000	MDA/HGA 80×25 单色	EGA/VGA+ 40×25 彩色
0001	MDA/HGA 80×25 单色	EGA/VGA+ 80×25 彩色
0010	MDA/HGA 80×25 单色	EGA/VGA+ 80×25 彩色
0011	MDA/HGA 80×25 单色	EGA/VGA+ 80×25 彩色
0100	CGA 40×25 彩色	EGA/VGA+ 80×25 单色
0101	CGA80×25 彩色	EGA/VGA+ 80×25 单色
0110	EGA/VGA+ 40×25 彩色	MDA/HGA 80×25 单色*
0111	EGA/VGA+ 80×25 彩色	MDA/HGA 80×25 单色*
1000	EGA/VGA+ 80×25 彩色	MDA/HGA 80×25 单色
1001	EGA/VGA+ 80×25 彩色	MDA/HGA 80×25 单色
1010	EGA/VGA+ 80×25 单色	CGA 40×25 彩色*
1011	EGA/VGA+ 80×25 单色	CGA 80×25 彩色*

注：*表示可任选的适配器，但非必须

调用：　　ah=12h
　　bl=10h

返回：　　bh=彩色/单色类型
　　0=彩色模式，使用 3Dxh 端口
　　1=单色模式，使用 3Bxh 端口
　　bl=适配器的内存大小
　　0=64K
　　1=128K
　　2=192K
　　3=256K 或更多
　　ch=特性连接器位
　　位 7~4=0
　　3=特性连接器线 0，位于状态 1
　　2=特性连接器线 1，位于状态 1
　　1=特性连接器线 0，位于状态 2
　　0=特性连接器线 1，位于状态 2
　　cl=控制适配器类型的开关
　　位 7~4=0
　　3~0=开关（参看表 9-7）

子功能	描述	中断	功能
BL=20h	安装替换的打印屏幕	10h	AH=12h

这个子程序以来自视频 BIOS 的处理程序替换当前的中断 5 打印屏幕处理程序。通常对这个功能的理解很不够。早期的 PC 和 XT 打印屏幕功能只考虑 25 行，而不管实际上显示了多少行。为了与 EGA 和 VGA 多于 25 行显示的适配器相兼容，必须替换老式的屏幕打印程序。

只有在需要显示多于 25 行的文本以及系统是 PC 或 XT 时，才有必要安装该替换的屏幕打印。对于 AT 及以后的系统，已经更新了标准的 BIOS 屏幕打印来处理多于 25 行的情况。

由于这仅仅只是替换当前的中断 5 向量而不挂起旧的向量，所以它会产生系统稳定性问题。如果其他设备驱动文件或 TSR 已挂起了中断 5，那么在触发了中断 5 后，它们不会再获得控制。在触发该子功能之前，我极力推荐检查一下中断 5 向量。如果中断 5 向量指向系统 BIOS 段址 F000，那么使用这个功能替换屏幕打印程序是可行的。否则，会显示一条警告或错误信息，并且不会触发该子功能。

如果你正在编写一个要求知道打印屏幕状态的 TSR，就不要挂起中断 5。你总可以通过读取 50:0 处的字节知道打印屏幕的状态。参看第 6 章以更多地了解地址 50:0 的屏幕打印状态。

调用：　　ah=12h
　　bl=20h
返回：　　安装新的屏幕打印处理程序

子功能	描述	中断	功能
BL=30h	选择文本模式的扫描线数	10h	AH=12h

设置屏幕所显示的扫描线的数目。直到下次模式设置之后，新设置的扫描线才会发挥作用。设置相同的模式不会改变扫描线的数目。不幸的是，该子功能不支持扫描线值为480或更多的扫描线设置。只有VGA+适配器才支持这个功能。

如果40:87的第3位指出没有激活适配器，那么就会忽略这个功能。如果适配器位于单色模式下，那么将会忽略为设置200条扫描线所做的努力，一旦获得通过，该子功能简单地更新位于40:88h处的高级视频开关字节，就好像设置了一个替换的模式一样。另外，位于40:89处的高级视频保存区1字节中会保存更新后的200/350和400扫描线位。

调用：　　ah=12h
　　al=模式设置之后的扫描线值
　　0=200扫描线（CGA分辨率）
　　1=350扫描线（EGA分辨率）
　　2=400扫描线（VGA分辨率）
　　bl=30h
返回：　　al=12h（表示支持子功能）

子功能	描述	中断	功能
BL=31h	颜色寄存器装入选项	10h	AH=12h

控制在设置模式之后是否更新颜色寄存器。通常地，作为设置模式的一部分，64个8位颜色寄存器、边界颜色寄存器、以及256个18位颜色寄存器会设置为缺省值。这个功能在设置模式期间阻止对颜色寄存器所做的任何修改。

装入选项的状态保存在40:89的第3位中，只有MCGA和VGA+适配器才支持该子功能。

调用：　　ah=12h
　　al=颜色装入状态
　　0=允许正常的颜色寄存器装入
　　1=禁止装入颜色寄存器
　　bl=31h
返回：　　al=12h（表示支持子功能）

子功能	描述	中断	功能
BL=32h	视频开放/禁止	10h	AH=12h

它控制 VGA+适配器是否响应 I/O 端口以及是否使用视频内存区。如果被禁止，系统将不可访问 VGA 卡，除非使用本命令。这意味着将忽略 I/O 端口，全部由 VGA 卡从 A000 到 BFFF 映射的视频内存地址将不复存在。

该子功能并不清空屏幕或关闭向显示器的视频输出。实际的视频适配器仍然处于激活状态，并且会冻结屏幕的显示内容，直到开放视频和更新屏幕。在视频禁止时，内存中视频 BIOS ROM 保持激活状态。

该子功能执行的低级操作是，向端口 46E8h 写入值 0 来禁止视频，以及向这个端口写入值 E 来开放视频。某些适配器可能会使用其他的 I/O 端口来开放和禁止视频。只有 MCGA 和 VGA+适配器才支持该子功能。

调用：
```
ah=12h
al=视频适配器状态
    0=开放视频
    1=禁止视频
bl=32h
```
返回：
```
al=12h（表示支持子功能）
```

子功能	描述	中断	功能
BL=33h	将颜色转化为灰度	10h	AH=12h

在 VGA 灰度显示器上，希望总是以某种灰度级保存颜色。很少有应用程序支持灰度显示器。设置了该选项之后，使用色彩的应用程序通常很容易读取灰度显示器。这个功能设置一个标志位，在设置视频模式期间或在使用功能 10h 更新任何一个颜色寄存器期间，在颜色寄存器中强制装入灰度级。

转化的状态保存在 40:89 的求反位 1 中。只有 MCGA 和 VGA+适配器支持这个功能。

调用：
```
ah=12h
al=灰度转化模式
    0=开放
    1=禁止
bl=33h
```
返回：
```
al=12h(表示支持子功能)
```

子功能	描述	中断	功能
BL=34h	光标大小控制	10h	AH=12h

使用功能 1 指定光标的起点和终点可以设置光标大小。如果光标大小控制支持分级，

那么功能 1 传递的值就会改变成当前字符格大小的相对光标大小。这是缺省情况。如果没有分级，功能 1 的光标大小不会改变。

例如，如果希望一个半块光标，程序就会假定是一个 8×8 字符格，然后程序将光标的起点置为 3 而将终点置为 7。如果系统实际位于 8×16 字符格模式下，那么光标大小就会出错。分级模式将重新调整请求并获得接近的半块光标，即使是使用了错误的值。大小控制可靠的处理 8 到 32 条扫描线的任何一种字体，并支持无光标、超界光标、下划线光标、半格和全格光标。

光标大小控制的状态保存在 40:87 的位 0 中。只有 VGA+适配器才支持该子功能。

调用：　ah=12h
　　bl=光标大小控制
　　0=字符高度的分级大小
　　1=光标无分级
　　bl=34h
返回：　al=12h（表示支持该子功能）

子功能	描述	中断	功能
BL=35h	显示开关	10h	AH=12h

如果系统在主板上配有 VGA 就可以使用这个功能。显示开关支持在主板 VGA 和一个替换的 VGA 适配器之间切换。大多数 PA/2 系统在主板上安装有 VGA 系统。

该子功能所解决的两个问题是段址 40h 处关键 BIOS 数据区的字节共享使用和通信可能彼此覆盖的 I/O 地址。

第一次使用这个功能时，必须先同时调用 AL=0 以便在一个用户提供的 128 字节长的缓冲区中保存视频系统的状态。它还禁止了嵌入式的 VGA 适配器。接着使用 AL=2 来执行所有的切换属性，以禁止活动的 VGA，然后触发 AL=3 来开放以前已激活的 VGA。每次触发 AL=2，AL=3 命令时，显示将在主板 VGA 和适配器 VGA 之间切换。

只有 MCGA 和 VGA+系统及适配器才支持该子功能。

调用：　ah=12h
　　al=在 VGA 间切换的步骤
　　0=首次保存视频状态，并禁止 VGA+适配卡
　　1=首次开放主板 VGA
　　2=关闭活动的 VGA+
　　3=开启以前未被激活的 VGA+
　　bl=35h
　　es:dx=指针，指向一个 128 字节开关状态保存区（AL=1 不需要）
返回：　al=12h(表示支持子功能)

子功能	描述	中断	功能
BL=36h	视频屏幕开/关	10h	AH=12h

该子功能清空屏幕，该命令不影响视频内存和视频模式。当开放屏幕时，会恢复以前的屏幕内容。

只有 VGA+系统才支持该子功能。

调用：　ah=12h
　　al=视频屏幕
　　0=清空屏幕
　　1=开放屏幕
　　bl=36h

返回：　al=12h（表示支持子功能）

子功能	描述	中断	功能
BL=37h	主框交互支持	10h	AH=12h

仅对于 XGA 适配器而言，该新功能将普通的文本属性定义切换到一种与主框兼容的属性。表 9-8 列出了属性字节的普通定义和主框定义。

表 9-8　属性字节描述

位	描述（VGA 普通）	位	描述（主框类型）
7=0	普通背景	7=0	背景色 0。
1	普通速率闪烁，50%周期比或者增强比或者使用 8 色背景（参看 AH=10h，AL=3）。	1	双倍速率闪烁，75%周期或者使用 8 色。 背景（参看 AH=10h，AL=3）。
6=x	背景色，第 4、5、6 位从 8 色中选择一种。	6=0	普通。
		1	求反视频。
5=x	背景色。	5=0	普通。
		1	下划线开。
4=x	背景色。	4=0	普通。
		1	如果第 5 位=0，则将下划线的最左和最右点设置为前景色。如果第 5 位=1 则将这些点设置为背景色。

续表

位	描述（VGA 普通）	位	描述（主框类型）
3=x	增强前景，选择字符字体。	3=x	增强前景，选择字符字体。
2=x	前景色，第 0、1、2 位从 8 色中选择一种。	2=x	前景色，第 0、1、2 位从 8 色中选择一种。
1=x	前景色。	1=x	前景色。
0=x	前景色。	0=00x	前景色。

只有 XGA 适配器才支持该子功能。

调用：　　ah=12h

al=文本属性字节

0=普通的 VGA 类型属性

1=主框类型属性

bl=37h

返回　　ah=12h（表示支持子功能）

中断	功能	描述	平台
10h	13h	写串	所有

这个功能在显示屏的指定位置写入一串字符。如果必要，这个功能会自动换行、翻滚以及解释某些专用操作的控制字符。有两个子功能支持文本和图形模式，还有两个子功能只支持文本模式下的属性。

当使用 DOS/V 的双字节字符集（DBCS）时，下面所有的子功能都把一个 DBCS 字符算成两个“字符”。

只被解释成命令的字符有 7、8、 A 和 D，这些值执行下列操作。

值	名称	描述
7	铃响	使用时钟 2 生成 1/2 秒的响声
8	回退	光标后退一列，除非已经位于第 0 列，这各情况下会忽略该命令
A	前进一行	光标下移一行，除非已经位于最后一行，这时屏幕会向上翻滚一行
D	返回	光标移动第 0 列

文本模式下向上翻滚时，新行的属性同光标以前所在的位置相同，图形模式下，新行的背景色为黑。

下列子功能用入写串：

值	名称	描述
7	铃响	使用时钟 2 生成 1/2 秒的响声
8	回退	光标后退一列，除非已经位于第 0 列，这各情况下会忽略该命令
A	前进一行	光标下移一行，除非已经位于最后一行，这时屏幕会向上翻滚一行
D	返回	光标移动第 0 列

AL	写串子功能	平台
0	写字符，不影响光标	所有
1	写字符并移动光标	所有
2	写字符和属性，不影响光标	所有
3	写字符和属性并移动光标	所有
10h	读取 DBCS 的字符和属性，不影响光标	DOS/V
11h	读取 DBCS 的字符和属性组，不影响光标	DOS/V
20h	写入 DBCS 的字符和属性，不影响光标	DOS/V
21h	写入 DBCS 的字符和属性组，不影响光标	DOS/V

子功能	描述	中断	功能
AL=0	写字符，光标不变	10h	AH=13h

将所提供的串写入列显示屏，该子功能不影响光标的位置，在图形和文本模式都有效。

调用：
ax=1300h
bh=页号
bl=属性字节
cx=串中的字符数
dh=开始行（最上一行为 0）
dl=开始列（最左一列为 0）
es:bp=指针，指向字符串

返回：
显示字符

子功能	描述	中断	功能
AL=1	写字符并移动光标	10h	AH=13h

将所提供的串写入到显示屏，同时随着写入到屏幕的字符而移动光标。在图形模式和文本模式下这个功能都有效。

调用：
ax=1301h
bh=页号
bl=属性字节
cx=串中的字符数
dh=开始行（最上一行为0）
dl=开始列（最左一列为0）
es:bp=指针，指向字符串
返回：
显示字符

子功能	描述	中断	功能
AL=2	写字符和属性，不影响光标	10h	AH=13h

将所提供的串写入到显示屏，串中的每个字符后紧跟着相关的属性字节。这意味着串的格式如下：字符、属性、字符、属性等。该子功能不影响光标位置，但只在文本模式下才有效。

调用：
ax=1302h
bh=页号
bl=属性字节
cx=串中的字符数
dh=开始行（最上一行为0）
dl=开始列（最左一列为0）
es:bp=指针，指向字符串
返回：
显示字符

子功能	描述	中断	功能
AL=3	写字符和属性，移动光标	10h	AH=13h

将所提供的串写入到显示屏，串中的每个字符后紧跟着相关的属性字节。这意味着串的格式如下：字符、属性、字符、属性等。光标位置随着写到屏幕的字符而移动。这个功能只在文本模式下有效。

调用：
ax=1303h
bh=页号
bl=属性字节
cx=串中的字符数
dh=开始行（最上一行为0）
dl=开始列（最左一列为0）
es:bp=指针，指向字符/属性串
返回：
显示字符

子功能	描述	中断	功能
AL=10h	为 DBCS 读取字符和属性，不影响光标	10h	AH=13h

从屏幕向用户提供的缓冲区中读取字符和匹配的属性。串中每个字符后紧跟着相关的属性字节。这意味着串具有下列格式：字符、属性、字符、属性等。该子功能不影响光标的位置，但只在 DOS/V 上有效。

调用：
ax=1310h
bh=0
cx=串中的字符数
dh=开始行（最上行为 0）
dl=开始列（最左列为 0）
es:bp=指针，指向字符/属性缓冲区

返回：
es:bp=指针，指向读自屏幕的字符和属性

子功能	描述	中断	功能
AL=11h	为 DBCS 读取字符和属性串，不影响光标	10h	AH=13h

从屏幕向用户提供的缓冲区中读取字符和匹配的属性。串中每个字符后紧跟着相关的三个属性字节。这意味着串具有下列格式：字符、属性 0、属性 1、属性 2、字符、属性 0、属性 1、属性 2 等。每个字符有三个属性，属性 0 用于普通的文本模式，属性 2 总为 0，而属性 1 定义如下：

属性 1	位	7=1	下划线格，使用前景色
		6=1	将属性 0 字节中的前景和背景属性颠倒过来（具体操作时可能会不起作用）
		5=0	未使用
		4=0	未使用
		3=1	格中使用垂直方向的白色网线
		2=1	格中使用水平方向的白色网线
		1=0	未使用
		0=0	未使用

该子功能不影响光标的位置，但只在 DOS/V 上有效。

调用：
ax=1311h
bh=0

cx=串行口的字符数
dh=开始行（最上行为0）
dl=开始列（最左列为0）
es:bp=指针，指向字符/属性组缓冲区

返回：　es:bp=指针，指向读自屏幕的字符和属性

子功能	描述	中断	功能
AL=20h	为DBCS写入字符和属性，不影响光标	10h	AH=13h

从用户提供的缓冲区向显示屏写入字符和匹配的属性组。串中的每个字节后紧跟着相关的属性字节。这意味着串具有下列格式：字符、属性、字符、属性等。该子功能不影响光标的位置。这个子功能只在DOS/V下才有效。

调用：　ax=1320h
bh=0
cx=串中的字符数
dh=开始行（最上行为0）
dl=开始列（最左列为0）
es:bp=指针，指向字符和属性

返回：　显示字符和属性

子功能	描述	中断	功能
AL=21h	为DBCS写入字符和属性，不影响光标	10h	AH=13h

从用户提供的缓冲区向显示屏写入字符和匹配的属性组。串中的每个字节后紧跟着相关的一个属性字节。这意味着串具有下列格式：字符、属性0、属性1、属性2、字符、属性0、属性1、属性2等。参看子功能11h以了解这些属性的定义。

该子功能不影响光标的位置，但只在DOS/V使用视频模式73h时才有效。

调用：　ax=1321h
bh=0
cx=串中的字符数
dh=开始行（最上行为0）
dl=开始列（最左列为0）
es:bp=指针，指向字符和属性组

返回：　显示字符和属性

子功能	描述	中断	功能
10h	14h	LCD打印	PC可选择

这种过时的 PC 可逆型膝上电脑可以控制如何在屏幕上显示高密度字符。PC 可逆型上所使用的 LCD 屏幕只能显示像素的开或关，而没有灰度级。其他不支持 VGA 灰度级的膝上型电脑可能也会执行这个功能。

三个子功能用来控制如何显示字体

AL=	LCD 字体控制子功能
0	装入用户指定的字体
1	装入 ROM 字体
2	高密度属性效果

子功能	描述	中断	功能
AL=0	装入用户指定的字体	10h	AH=14h

调用：
ax=1400h
bh=每字符的字节数（1~32）
bl=字体页（0 或 1）
cx=待装入的字符数（1~256）
dx=起始字符号（1~255）（用作装入起点偏移量）
es:bp=指针，指向用户指定的字体字节

返回：
装入后的字体字符

子功能	描述	中断	功能
AL=1	装入 ROM 字体	10h	AH=14h

调用：
ax=1401h
bl=字体页（0 或 1）

返回：
从 ROM 装入字体字符

子功能	描述	中断	功能
AL=2	高密度属性效果	10h	AH=14h

调用：
ax=1402h
bl=如何解释高密度属性
0=忽略
1=反色显示字符
2=带下划线显示字符
3=显示来自替换的字体页面中的字符

返回：
保存选项

中断	功能	描述	平台
10h	15h	获取显示器类型	PC 可逆型

可以使用本命令来读取可逆型 PC 的显示器类型。其他早期的膝上型电脑可能支持这个功能。

调用：ah=15h

返回：ax=可替换的显示适配器类型

0=没有可替换的适配器

5140h=LCD

5153h=CGA 显示器

5151h=单色显示器

es:di=指向一个 7 字类型表的指针

字 1=显示模式号

字 2=每米中的垂直像素数

字 3=每米中的水平像素数

字 4=垂直像素的总数

字 5=水平像素的总数

字 6=相邻两像素中心到中心的水平距，单位微米

字 7=相邻两像素中心到中心的垂直距，单位微米

中断	功能	描述	平台
10h	18h	字体模式的请求	DOS/V

这个功能实现在用户缓冲区和系统字体缓冲区之间的传送。只有带 DOS/V 的系统才支持它。

子功能	描述	中断	功能
AL=0	装入字体模式	10h	AH=18h

调用：ax=1800h

bx=0

cx=单字节或双字节字符（单字节位于 cl 中而 ch=0）

dx=像素字符宽及高

0810h=8×16 单字节字体

0813h=8×19 单字节字体

1010h=16×16 双字节字体

1818h=24×24 用户提供的双字节字体

es:di=指针，指向保存了字体图像的缓冲区

返回：al=0,如果成功，其他任何值都表示出错

es:di=指针，指向装入了字体字节的用户缓冲区。

子功能	描述	中断	功能
AL=1	设置字体模式	10h	AH=18h

调用：ax=1801h
bx=0
cx=单字节或双字节字符（单字节位于 cl 中而 ch=0）
dx=像素字符宽及高
0810h=8×16 单字节字体
0813h=8×19 单字节字体
1010h=16×16 双字节字体
1018h=24×24 用户提供的双字节字体
es:di=指向保存了字体图像的缓冲区

返回：al=0,如果字体图像装入到了系统字体缓冲区中，其他任何值都表示出错

中断	功能	描述	平台
10h	1Ah	读/写显示器组合码	VGA+

这个功能返回较早的适配器的模拟信息，以及附带的显示器的基本类型。VGA BIOS 在初始化时确定这些信息。表 9-9 列出了这些适配器代码，只须 VGA+系统才支持这个功能。

表 9-9 适配器代码

代码	适配器和显示器
0	没有显示器。
1	单色或 HGA。
2	CGA。
4	EGA，带彩色 EGA 显示器。
5	EGA，带单色显示器。
6	PGA，带彩色 PGA 显示器。
7	VGA，带有模拟单色 VGA 显示器。
8	VGA，带有模拟彩色 VGA 显示器。
B	MCGA，带有模拟单色 VGA 显示器。
C	MCGA，带有模拟彩色 VGA 显示器。
FF	未知。

子功能	描述	中断	功能
AL=0	读显示器组合码	10h	AH=1Ah

调用：ax=1A00h
返回：al=1Ah（表示支持子功能）
bh=活动显示器的代码（参看表 9-9）
bl=可替换的显示器代码（参看表 9-9）

子功能	描述	中断	功能
AL=1	写显示器组合码	10h	AH=1Ah

调用：ax=1A01h
bh=活动显示器的代码（参看表 9-9）
bl=可替换的显示器代码（参看表 9-9）
返回：al=1Ah（表示支持子功能）

中断	功能	描述	平台
10h	1Bh	返回非凡现象系统的状态信息	VGA+

在用户提供的内存中装入有关当前视频状态的 64 字节信息表。这个表由大量的 BIOS 数据区字节、寄存器信息、ROM 数据以及保存 VGA 适配器上的值组成。表 9-10 列出了这个表的内容。第 6 章详细解释了这些来自 BIOS 数据区的数据。

只有 VGA+系统才支持这个功能，并且在使用 DOS/V 时可能不支持它。

表 9-10 视频状态表

十六进制			
偏移量	大 小	描 述	读 自
0	双字	ROM 中功能状态的远指针（参看表 9-11）	ROM
4	字节	视频模式	40:49h
5	字	字符的列数	40:4Ah
7	字	每页的总字节数	40:4Ch
9	字	当前页的偏移量	40:4Eh
B	8 个字	第 0~7 页的光标位置（高字节=行，低字节=列）	40:50h
1B	字	光标形状（起始和终止格行）	40:60h
1D	字节	活动的显示页	40:62h
1E	字	视频 I/O 端口基号	40:63h

续表

十六进制					
偏移量	大　小	描　述			读　自
20	字节	CGA/MDA 的当前模式的寄存器值			40:65h
21	字节	当前 CGA 调色板寄存器值			40:66h
22	字节	屏幕字符的行数（值 40:84h+1）			40:84h
23	字	每字符的扫描线数			40:85h
25	字节	活动的显示模式表（9-9）			
26	字节	可替换的显示模式（9-9）			
27	字	该模式支持的不同颜色数（2~FFFF）			ROM
29	字节	该模式的显示页数（1~8）			ROM
2A	字节	当前显示器的扫描线数			
		0=200 条扫描线			
		1=350 条扫描线			
		2=400 条扫描线			
		3=480 条扫描线			
2B	字节	当属性密度位=0 时的字体页			
2C	字节	当属性密度位=1 时的字体页			
2D	字节	多种状态位			
		位	7=0	未使用	
			6=0	未使用	
			5=0	属性位 7 控制密度	
			1	属性位 7 控制闪烁	
			4=0	不控制光标大小（参看 AH=12h，bl=34 以了解光标大小控制）	40:87
			1	普通的大小控制	40:89
			3=0	在任何模式设置时将颜色寄存器设置为缺省值	40:89
			2=0	彩色显示器	40:89
			1	单色显示器	40:89
			1=1	激活灰度转换	
			0=0	只支持彩色模式	
			1	显示器支持所有的可用的模式	
2E	字节	未使用（某些供应商可能用到）			
31	字节	视频内存值（来自 40:87 的第 6 和 5 位）			40:87
		0=64K			
		1=128K			

续表

十六进制					
偏移量	大 小	描 述			读 自
		2=192K			
		3=256K			
32	字节	保存指针状态信息			
		位	7=0	未使用	
			6=0	未使用	
			5=1	激活显示组合码	
			4=1	调色板超界激活	
			3=1	图形字体超界激活	
			2=1	Alpha 字体超界激活	
			1=1	动态保存区激活	
			0=1	512 字体集激活	
33	字节	未使用（可能某些供应商使用了）			

表 9-11 功能状态表（来自表 9=10）

十六进制					
偏移量	大 小	描 述			读 自
0	字节	允许视频模式			
1		位 x=1 允许模式 x+8			ROM
	字节	允许视频模式			
		位 x=1 允许模式 x+8			ROM
2	字节	允许视频模式			
		位	7=0	未使用	
			6=0	未使用	
			5=0	未使用	
			4=0	未使用	
			3=1	允许模式 13h	
			2=1	允许模式 12h	
			1=1	允许模式 11h	
			0=1	允许模式 10h	
3	4 字节	未用（为将来保留）			ROM
7	字节	文本模式下支持的扫描线数			ROM
		位	7=0	未使用	
			6=0	未使用	

续表

十六进制					
偏移量	大　小	描		述	读　自
			5=0	未使用	
			4=0	未使用	
			3=0	未使用	
			2=1	支持 400 条扫描线模式	
			1=1	支持 350 条扫描线模式	
			0=1	支持 200 条扫描线模式	
8	字节	同时可用的字体页数			ROM
9	字节	字体页的最大数目			ROM
A	字节	多种功能 1			ROM
		位	7=1	支持多页 256 色寄存器	
			6=1	支持 256 个 18 位颜色寄存器	
			5=1	支持 64 个 8 位颜色寄存器	
			4=1	支持光标形状控制（参看功能 AH=12h,BL=23h）	
			3=1	支持选项—禁止模式设置时的颜色寄存器更新（参看功能 AH=12h,BL=31h）	
			2=1	支持装入字符字体（参看 AH=11h）	
			1=1	支持灰度转化（参看 AH=12h，BL=33h）	
			0=0	只支持彩色模式	
			1	显示器接受所有的模式	
B	字节	多种功能 2			ROM
		位	7=0	未使用	
			6=0	未使用	
			5=0	未使用	
			4=0	未使用	
			3=1	支持显示组合码（参看功能 AH=1Ah）	
			2=0	支持背景选择、闪烁/增强选择（参看功能 AX=1003h）	
			1=1	支持保存和恢复状态（参看功能 AH=1Ch）	

续表

十六进制					
偏移量	大 小	描		述	读 自
			0=1	支持光笔（参看功能AH=4）	
C	2字节	未用（为将来保留）			
E	字节	保存功能所支持的指针			ROM
		位	7=0	未使用	
			6=0	未使用	
			5=1	显示组合码	
			4=1	颜色寄存器超界	
			3=1	图形字体超界	
			2=1	文本字体超界	
			1=1	动态保存区	
			0=1	512字符字体	
F	字节	未用（为将来保留）			ROM

调用：　　ah=1Bh
　　es:di=指针，指向一个保存了状态信息的64字节表
返回：　　al=1Bh（表示支持子功能）

中断	功能	描述	平台
10h	1Ch	保存/恢复视频状态	VGA+

这个功能用于保存和恢复VGA适配器的全部状态。对于那些需要开发能运行在所有模式下的TSR的人来说，这是一种极受欢迎的特性。

一旦保存了状态，就可以改变模式。对于一个TSR来说，应先使用本功能保存视频状态。会被覆盖的显示内存也应该加以保存。如果是从一个图形模式进入文本模式，这还应该包括一部分红色面板2，在这里保存了文本字符字体。将模式值的第7位置高来改变模式。这可以阻止适配器擦除视频缓冲区。然后可以在显示器上“弹出”TSR。

退出时，TSR将模式返回到起初的模式，更新显示缓冲区，然后恢复状态信息。如果幸运的话，它会全部发挥作用。

一个程序可能不必保存和恢复所有的信息，这取决于它将采取的操作。某些受空间限制的TSR可能会有选择地保存和恢复关键的信息，比如硬件和BIOS数据区的信息，而忽略颜色寄存器。表9-12列出了所有的选项。

表 9-12 保存和恢复状态选项

CX 位	值	描 述
0	1	保存/恢复视频硬件状态。
1	1	保存/恢复视频 BIOS 数据区。
2	1	保存/恢复数据转化（DAC）和颜色寄存器。
3~15	0	未使用。

保存/恢复视视频状态功能有三个子功能。只有 VGA+适配器才支持它，但是在运行 DOS/V 时不支持。

AL=	保存/恢复状态子功能
0	为保存状态获取缓冲区大小
1	保存适配器状态
2	恢复适配器状态

子功能	描述	中断	功能
AL=0	为保存状态获取缓冲区大小	10h	AH=1Ch

在保存视频状态之前先看看将需要多少内存不失为一种好的想法。这个功能返回所需要的 64 字节块的数目，值取决于需要保存的信息量。不同的供应商返回不同的值，不幸的是，这里也没有一个指定的上限。据我分析，SVGA 的状态选项所需要的 64 字节块的数目一般为：

保存/恢复视频硬件状态：	2（128 字节）
保存/恢复视频 BIOS 数据区：	2（128 字节）
保存/恢复 DAC 和颜色寄存器：	D（832 字节）
三项总共：	F（960 字节）

请准备尽可能大的空间。

调用：	a x=1C00h
	cx=要求保存/恢复的状态（参看表 9-12）
返回：	al=1Ch（表示支持子功能）
	bx=要求的 64 字节块的数目

子功能	描述	中断	功能
AL=1	保存适配器状态	10h	AH=1Ch

调用：　　ax=1C01h
　　cx=要求保存的状态（参看表 9-12）
　　es:bx=指针，指向保存状态的地址
返回：　　al=1Ch（表示支持子功能）

子功能	描述	中断	功能
AL=2	恢复适配器状态	10h	AH=1Ch

调用：　　ax=1C02h
　　cx=要求恢复的状态（参看表 9-12）
　　es:bx=指针，指向获取待恢复状态的地址
返回：　　al=1Ch（表示支持子功能）

中断	功能	描述	平台
10h	1Dh	换档状态线功能	DOS/V

这个功能控制底部状态线。底部状态线用来显示换档状态，在安装了输入辅助子系统驱动文件时还可以帮助输入双字节字符状态。在激活状态线时，会将位于 40:84 处的 BIOS 值减 1，该值表示了显示的行数，而在关闭状态线时又会加 1。这个功能仅用于 FEP（Font End Processor，字体结束处理器）驱动文件。

子功能	描述	中断	功能
AL=0	激活换档状态线	10h	AH=1Dh

调用：　　ax=1D00h
　　bx=在底部为换档状态所保留的行数（通常为 1）
返回：　　无

子功能	描述	中断	功能
AL=1	激活换档状态线	10h	AH=1Dh

调用：　　ax=1D01h
　　bx=在底部为换档状态所保留的行数（通常为 1）
返回：　　无

子功能	描述	中断	功能
AL=2	获取换档状态线的数目	10h	AH=1Dh

调用：　　ax=1D02h
返回：　　bx=在底部为换档状态所保留的行数（通常为 1）

中断	功能	描述	平台
10h	1Fh	显示模式信息	XGA+

这个功能在一个缓冲区中装入系统中所有 XGA 适配器相关的信息。XGA 体系结构在一个系统中支持多达 8 个的 XGA。每个适配器使用指定的内存和 I/O 地址以避免冲突。

子功能	描述	中断	功能
AL=0	获取 XGA 信息缓冲区的长度	10h	AH=1Fh

该子功能返回下一个子功能 AL=1 所需的缓冲区大小。对于下面的子功能，通常每个 XGA 适配器需要 32 个字节。这意味着，如果某个系统有三个适配器，那么下一个子功能将需要至少 96 字节的缓冲区。

只有 XGA 适配器才支持该子功能。该子功能是标识存在 XGA 适配器的理想方法。一个非 XGA 的适配器将不会改变任何寄存器，而只是在 AX 中返回 1F00h。

调用：　　ax=1F00h
返回：　　al=1Fh（表明支持子功能）
　　bx=缓冲区所需要的字节数

子功能	描述	中断	功能
AL=1	获取 XGA 信息	10h	AH=1Fh

将 XGA 适配器信息装入到用户提供的缓冲区空间中。参看子功能 0 以了解所需要的缓冲区大小。表 9-13 显示了每个适配器保存在缓冲区中的信息。如果使用多个适配器，那么返回的缓冲区将保存有多个数据块，一块对应一个 XGA。

表 9-13　XGA 信息缓冲区

偏移量	大小	描述
0	字	到下一个 XGA 适配器信息的偏移量，单位是字节。
2	字节	XGA 驻留的插槽号。
3	字节	XGA 执行的功能水平。
		3=基本 XGA 执行。
		5=XGA-N1 执行（包括额外的 XGA 功能）。
4	字节	XGA 执行的分辨力水平。
		3=基本 XGA 执行，45MHz 最大像素率。

续表

偏 移 量	大 小	描 述
		5= XGA-N1 执行，90MHz 最大像素率。
5	字	供应商 ID。
7	字	指定的供应商。
9	字	适配器 I/O 的基地址。
B	字	内存映射 XGA 协处理器寄存器出现在系统地址空间中的段址。
D	字	1MB 孔隙（aperture）的物理内存地址（值 3 表示孔隙开始于 3MB，值 0 表示没有分配孔隙）。
F	字	4MB 孔隙的物理内存地址（值 8 表示孔隙开始于 8MB，值 0 表示没有分配孔隙）。
11	字	视频内存缓冲区在系统内存中的地址（值 4 表示缓冲区开始于 4MB）。
13	字	由 POST 确定的显示器合成 ID。
15	字节	适配器上的总内存（单位是 256K，所以值 8 代表该卡有 2MB 的内存）。

调用：　　ax=1D01h
　　es:di=放置信息的缓冲区
返回：　　al=1Fh9（表示支持子功能）
　　es:di=装入了 XGA 信息的缓冲区

中断	功能	描述	平台
10h	4Fh	VESA XGA 子功能	VESA XGA

这个功能用于提供 VESA XGA 适配器所独有的大量服务，但是 IBM 的 XGA 适配器可能不支持这些子功能。所有 VESA XGA 制造商都限于使用这个共同的接口，但是不一定会执行每个功能。在完成了任何一个功能时，必须检查状态，以确认支持这个功能并成功的执行了该功能。

VESA XGA 功能有许多子功能，它们仅被 VESA XGA 适配器所支持。

AL=	VESA XGA 子功能
0	获取 VESA XGA 的环境信息
1	获取 VESA XGA 的子系统信息
2	获取 VESA XGA 的模式信息
3	获取 VESA XGA 的视频信息
4	获取当前的 VESA XGA 的视频信息

5　　设置 VESA XGA 的特性连接器状态
6　　获取 VESA XGA 的特性连接器状态

子功能	描述	中断	功能
AL=0	获取 VESA XGA 的环境信息	10h	AH=4Fh

向用户提供的缓冲区装入 256 字节的信息，这些信息涉及到 VESA XGA 适配器的功能。表 9-14 列出了该缓冲区的内容。

表 9-14　VESA XGA 环境信息表

偏移量	大　小	描　　述			
0	4 字节	VESA XGA 署名——应该是“VESA”，表明 XGA 适配器服从“视频电子标准协会”标准。			
4	字	VESA 版号（例如 0100h=版本 1.00）。			
6	双字	供应商字符串——这是一个指向由供应商提供字符串的远指针，该串以零结尾。该串通常保存有制造商的名称和插卡型号，但是它留给供应商去指定这个文本的具体内容。			
A	双字	环境标志。			
		位	31~3	未使用，通常置 0	
		位	2=0	系统不支持总线管理	
			1	系统支持总线管理，由 VESA XGA POST 确定	
			1=x	系统总线结构	
			0=x	第 1 位	第 0 位
				0	0=微通道（MCA）
				0	1=AT/ISA
				1	0=未定义
				2	1=EISA
E	字	XGA 的数目——保存了当前系统安装的 XGA 的总数（不超过 8）。			
14	240 个字节	未使用——缓冲区剩下的部分为将业的说明而保留 VESA XGA BIOS 在该区域全部返回零。			

调用：　　ax=4E00h
es:di=装入了信息的缓冲区
所有的其他的 AX 值都表示没有安装 VESA XGA

子功能	描述	中断	功能
AL=1	获取 VESA XGA 扩系统信息	10h	AH=4Fh

在用户提供的缓冲区中装入有关 XGA 适配器的 256 字节的信息和指针。该缓冲区的内容描述见表 9-15

表 9-15 VESA XGA 子系统信息表

偏移量	大小	描述			
0	双字	供应商字符串——这是一个指向由供应商提供字符串的远指针，该串以零结尾。该串通常保有制造商的名称的卡的型号，但是它留给了供应商去指定该文本。			
4	双字	能力标志。			
		位	31~z8	未使用，通常设置为 0	
		位	7=0	DMA 通道被禁止，或者不是 AT 系统	
			1	DMA 通道被允许，并且是 AT 系统	
			6=x	DMA 通道，0~7，分配用于 AT 系统上的总线管	
			5=x	理（所有其他的系统上都置 0）	
			4=x		
			3=0	未使用	
			2=0	未使用	
			1=x	系统总线结构	
			0=x	第 1 位	第 0 位
				0	0=微通道（MCA）
				0	1=AT/ISA
				1	0=未定义
				1	1=EISA
8	双字	ROM 指针——指向 XGA 的 BIOS ROM。如果是 0，表示没有安装 ROM。指针的格式是段：偏移量。			
C	双字	内存映射寄存器指针——指针的格式是段：偏移量。			
10	字	该 XGA 适配器的基地址。			
12	双字	视频显示内存指针——物理显示内存起点，指针是 32 位形式。			
16	双字	指向 4MB 孔隙（aperture）的 32 位指针——零表示没有安装 4MB。			
1A	双字	指向 1MB 孔隙的 32 位指针——零表示没有安装 1MB 孔隙。			
1E	双字	指向 64KB 孔隙的 32 位指针——零表示没有安装 64KB 孔隙。			
22	双字	指向供应商定义的大于 4MB 的孔隙的指针，零表示没有提供用户定义的孔隙。			
26	字	供应商定义的孔隙的大小，单位 64K 块。			
28	双字	指针，指向所支持视频模式的列表——格式是段：偏移量，指向适配器所支持 XGA 视频模式的列表。列表的每个入口都是一个字，以 FFFFh 标志结尾。该列表包括 VISA 指定的模式号以及供应商所特有的模式。该列表可能会包括因显示器或适配器内存限制页不支持的模式。参看子功能 2 以了解当前支持哪些模式。			
2C	字	总内存——64K 块的数目。值 20h 表明 XGA 适配器有 2MB 的内存。			
2E	双字	供应商的 ID 号。			
32	206 个字节	未使用——缓冲区余下的部分留到将来说明。VESA XGA BIOS 在该区域全部返回零。			

调用：　　ax=4E01h
　　es:di=装入了信息后的缓冲区

返回：　　如果成功：
　　ax=004Eh
　　es:di=装入了信息后的缓冲区
　　所有其他的 AX 值表明没有安装 VESA XGA。

子功能	描述	中断	功能
AL=2	获取 VESA XGA 模式信息	10h	AH=4Eh

获取指定视频模式的详细信息。前一个子功能返回的缓冲区包括一个指针指向了适配器所支持的视频模式列表，这个子功能就提供了每个模式的详细细节。不支持列表中未包括的视频模式。用户提供的缓冲区必须长为 256 字节。参看表 9-16 了解这个表的内容。

表 9-16　视频模式信息表

偏移量	大小	描述		
0	字	模式属性。		
		位	15~5	未使用，通常设置为 0。
		位	4=0	文本模式，激活 VGA 寄存器，而关闭了 XGA 寄存器。
			1	图形模式，关闭了 VGA 寄存器，而激活了 XGA 寄存器。
			3=0	未使用，通常置为 0。
			2=0	该模式不支持 BIOS 输出功能（翻滚、写字符、写电传模式、写像素等等）。
			1	支持 BIOS 输出功能。
			1=0	未使用，通置为 0。
			0=0	显示器或适配卡内存大小的强加限制，使不支持模式。
			1	支持视频模式。
2	字	每个逻辑扫描线的字节数——组成一逻辑扫描线的字节总数。一个逻辑扫描线可能比物理“显示”的扫描线长。		
4	字	水平分辨率——在图形模式下，是每条水平扫描线的像素数目，在文本模式下，是每行的字符数目。		
6	字	垂直分辨率——在图形模式下，是垂直方向的像素数目，在文本模式下，是字符行的数目。		
8	字节	字符格宽度。		
9	字节	字符格高度。		

续表

偏 移 量	大 小	描 述
A	字节	内存面板的数目——组成视频内存所需要的颜色位面板的数目。对于一个标准的 VGA 16 色模式，通常有四个面板。
B	字节	每像素的位数——每个屏幕点的颜色/阴影位数。对于 256 色模式，每个像素必须是 8 位。
C	字节	内存组织：
		0=文本模式（字符和属性）
		1=CGA 图形
		2=Hercules 图形
		3=4 面板
		4=包像素
		5=256 色非链 4
		6=直接着色
		7=YUV-24
		8~FF=为将来的 VESA 模式为保留
D	字节	图像页的数目——适配器为该模式同时所能处理的总页数。
E	字节	红色屏幕大小——使用直接着色内存组织时，为单像素定义红色浓度所使用的位数。在 YUV-24 内存组织下，该字节保有组成 V 组件的位数。所有其他的内存组织都未使用该值。
F	字节	红色区域位置——当使用直接着色或 YUV-24 内存组织时，该值指定了最不明显的红色的位置或者单像素数据区域内的 V 组件位。所有其他的内存组织都未使用该值。
10	字节	绿色屏幕大小——使用直接着色内存组织时，为单像素定义绿色浓度所使用的位数。在 YUV-24 内存组织下，该字节保有组成 V 组件的位数。所有其他的内存组织都未使用该值。
11	字节	绿色区域位置——当使用直接着色或 YUV-24 内存组织时，该值指定了最不明显的绿色的位置或者单像素数据区域内的 V 组件位。所有其他的内存组织都未使用该值。
12	字节	蓝色屏幕大小——使用直接着色内存组织时，为单像素定义蓝色浓度所使用的位数。在 YUV-24 内存组织下，该字节保有组成 V 组件的位数。所有其他的内存组织都未使用该值。
13	字节	蓝色区域位置——当使用直接着色或 YUV-24 内存组织时，该值指定了最不明显的蓝色的位置或者单像素数据区域内的 V 组件位。所有其他的内存组织都未使用该值。
14	字节	保留的屏蔽大小——该位数用来定义非颜色组件，例如使用直接着或 YUV-24 内存组织时的单像素浓度。所有其他的内存组织都未使用该值。
15	字节	保留区域位置——该值指定了最低有效位的非颜色组件，例如浓度的位置，这些组件位于单像素的数据区域内，这时使用的是直接着色或 YUK-24 内存组织。其他所有的内存组织都未使用该值。
16	234 个字节	未使用——缓冲区余下的部分为将来的说明而保留。VESA XGA BIOS 在该区域全部返回零值。

调用：　ax=4E02h
　　cx=来自子功能 1 列表中的视频模式号
　　es:di=放置信息的缓冲区
返回：　如果成功
　　ax=004Eh
　　ex:di=装入了信息后的缓冲区
　　如果支持功能，但是模式无效
　　ax=014Eh
　　如果不支持功能，al 不是 4Eh

子功能	描述	中断	功能
AL=3	设置 VESA XGA 视频模式	10h	AH=4Eh

改变视频模式。这个功能支持 VGA 类型的 8 位模式（将第 14~8 位置和 VESA XGA 的 15 位模式。参看子功能 1 和 2 以了解 VESA XGA 模式与支持。参看表 9-2 以了解通用的视频模式。

调用：　ax=4E03h
　　bx,第 15 位=0　　清空视频内存
　　1　　保持视频内存不变
　　14~0=视频模式号
　　cx,第 15~1 位=0　　未使用
　　第 0 位=0　　将特性连接器设定为缺省状态
　　1　　不改变特性连接器的状态
　　dx=XGA 处理程序
返回：　如果成功
　　ax=004Eh
　　如果支持该子功能，但模式无效
　　ax=014Eh
　　如果不支持该子功能，则 al 不是 4Eh

子功能	描述	中断	功能
AL=4	获取当前的 VESA XGA 视频模式	10h	AH=4Eh

返回当前的视频模式，包括标准的 VGA 模式 0~13h 以及新式的 15 位 VESA XGA 模式。该子功能不返回视频清空位，参看中断 10h 功能 F 以获取视频清空位。

调用：　ax=4E04h
　　dx=XGA 处理程序
返回：　如果成功

ax=004Eh
bx,第 15 位=0(固定为 0)
14~0=视频模式号
如果支持功能，但操作失败
ax=014Eh
如果不支持功能，则 al 不是 4Eh

子功能	描述	中断	功能
AL=5	设置 VESA XGA 特性连接器的状态	10h	AH=4Eh

该子功能控制 XGA 特性连接器上的数据方向，它还可以开放或禁止通过特性连接器的传输

调用：
ax=4E05h
bx,第 15~2 位=0 未使用
第 1 位=0 特性连接器的输入模式
1 特性连接器的输出模式
第 0 位=0 禁止特性连接器上的数据
1 开放特性连接器上的数据
dx=XGA 处理程序

返回：
如果成功
ax=004Eh
如果支持功能，但操作失败
ax=014Eh
如果不支持功能，则 al 不是 4Eh

子功能	描述	中断	功能
AL=6	获取 VESA XGA 特性连接器的状态	10h	AH=4Eh

该子功能获取 VESA XGA 特性连接器的当前状态，参看前一个子功能 4E05h 以了解如何设置这些选项。

调用：
ax=4E06h
dx=XGA 处理程序

返回：
如果成功：
ax=004Eh
bx,第 15~2 位=0 未使用
第 1 位=0 特性连接器的输入模式
1 特性连接器的输出模式
第 0 位=0 禁止特性连接器上的数据

1 开放特性连接器上的数据

如果支持这个功能，但操作失败：

ax=014Eh

如果不支持功能，则 al 不是 4Eh

中断	功能	描述	平台
10h	4Fh	超级 VGA 子功能	SVGA

该子功能用于超级 VGA 适配器所独有的一系列服务。所有 SVGA 卡的制造商都受限于这个共同的接口，但是不一定都选择执行每个功能。在完成任何一个功能时，必须检查状态以确认支持这个功能和成功地完成了这个功能。

该超级 VGA 功能有许多子功能，只有 SVGA 适配器才支持功能 4Fh。

AL=	SVGA 子功能
0	获取 SVGA 的通用信息
1	获取SVGA 的模式信息
2	设置 SVGA 的视频模式
3	获取当前的 SVGA 视频模式
4	获取保存缓冲区的大小（DL=0）
4	保存 SVGA 状态（DL=1）
4	恢复 SVGA 状态（DL=2）
5	设置 SVGA 内存窗口位置（BH=0）
5	获取 SVGA 内存窗口位置（BH=1）
6	设置逻辑扫描线长度（BL=0）
6	获取逻辑扫描线长度（BL=1）
7	设置逻辑显示起点（BL=0）
7	获取逻辑显示起点（BL=1）
4D	视频光标接口请求

子功能	描述	中断	功能
AL=0	获取 SVGA 的通用信息	10h	AH=4Fh

向用户提供的缓冲区中装入关于 SVGA 功能的 256 字节的信息。表 9-17 描述了这个表的内容。

调用：

ax=4F00h

es:di=放置信息的缓冲区

返回：　　如果成功

ax=004Fh

es:di=装入了信息的缓冲区

如果支持功能，但操作失败

ax=014Fh

如果不支持这个功能，则 al 不是 4Fh

表 9-17　SVGA 通用信息表

偏 移 量	大　小	描　　述
0	4 字节	SVGA 署名——必须是“VESA”，以表明 SVGA 服从视频电子标准协会（VESA）标准。
4	字	VESA 版本号（例如 0102h=版本 1.02）。
6	双字	供应商字符串——远指针，指向由供应商提供的字符串，以零结尾。该串通常存有制造商的名称和卡的型号，但是它留给了供应商来指定该文本。
A	双字	功能——该双字目前尚未定义，留作将来可能有的说明之用。
E	双字	模式列表——远指针，指向适配器所支持的 SVGA 视频模式列表。列表中的每个入口点都是一个字，列表的结尾由 FFFFh 标识。该列表可以包括因显示器或适配器内存限制而不支持的模式。参看子功能 1，以了解支持模式的更多信息。
12	字	总内存——该字显示了适配器所带的视频内存的总量，单位是 64K。例如：值 10h 表示卡有 1MB 的内存。
14	236 字节	未使用——缓冲区的余下部分为将来的说明而保留，大多数供应商在该区域返回全零。

子功能	描述	中断	功能
AL=1	获取 SVGA 模式信息	10h	AH=4Fh

获取某个指定的视频模式的详细信息。子功能 0 返回了一个指针指向了适配器所支持的视频模式列表。该子功能为每个模式提供了详细的细节。不支持表中未包括的视频模式，用户提供的缓冲区必须可以供返回 256 字节信息之用。参看表 9-18 获取这个表的内容。为了保证所有未使用的字节为零，在触发这个功能之前必须先将缓冲区全部置 0。

SVGA 提供了一种窗口方案，在这里视屏定义为两窗口之一。这里也描述了每个模式的 SVGA 窗口系统的信息。

表 9-18　视频模式信息表

偏　移　量	大　小	描　　述		
0	字	模式属性		
		位	15~5	未使用，通常置为 0
		位	4=0	文本模式
			1	图形模式
			3=0	单色模式，I/O 基地址是 3B4h（视频模式 7 或 0Fh 是单色模式）
			1	彩色模式，I/O 基地址是 3D4h
			2=0	该视频模式不支持 BIOS 输出功能（翻滚、写字符、写电传模式、写像素等等）
			1	支持 BIOS 输出功能
			1=0	没有扩展的模式信息
			1	扩展模式信息的起点偏移量为 12h
			0=0	由于显示器和适配卡上内存的大小的限制而不支持模式
			1	支持视频模式
2	字节	窗口 A 属性		
		位	7~3	未使用，通常为 0
		位	2=0	窗口 A 不可写
			1	窗口 A 可写
			1=0	窗口 A 不可读
			1	窗口 A 可读
			0=0	不支持窗口 A
			1	支持窗口 A
3	字节	窗口 B 属性		
		位	7~3	未使用，通常为 0
		位	2=0	窗口 B 不可写
			1	窗口 B 可写
			1=0	窗口 B 不可读
			1	窗口 B 可读
			0=0	不支持窗口 B
			1	支持窗口 B
4	字	窗口颗粒状——可放入视频内存的最小窗口，单位 KB。		
6	字	窗口大小——窗口所需的内存，单位：千字节。		
8	字	窗口 A 的段址——窗口 A 起点在视频内存中的段址。		
A	字	窗口 B 的段址——窗口 B 起点在视频内存中的段址。		
C	双字	窗口体系功能指针——远指针，指向窗口体系功能的远指针。该指针可能改变指向不同的模式。参看功能 4F05h 了解细节。调用窗口体系功能处理程序时不会保留 AX 和 DX 值，它也不提供返回状态。		
10	字	每条逻辑扫描线的字节数——逻辑扫描线的长度可能大于显示的扫描线只有当模式属性，字 0 的第 1 位置 1 时，下面的模式模式扩展信息才有效。		

续表

偏 移 量	大 小	描 述
12	字	水平分辨率——在图形模式下为垂直方向的像素数，在文本模式下，为文本行的数目。
14	字	垂直分辨率——在图形模式下为垂直方向的像素数，在文本模式下，为文本行的数目。
16	字节	字符格的宽度。
17	字节	字符格的高度。
18	字节	内存面板的数目——组成视频内存所需的色彩面板数。标准的16色VGA模式有四个面板。
19	字节	每像素的位数——每屏幕点的彩色/阴影位的数目对于256色模式，每像素必须有8位。
1A	字节	簇（bank）的数目——图形模式下，该字节存有每个扫描线组的扫描线簇的数目。模式4~6带有两个簇，而Hercules模式带有四个簇。对于带扫描线簇的模式，其返回值是1。模式D~13h不带扫描线簇。
1B	字节	内存组织。
		0=文本模式（字符和属性）
		1=CGA图形
		2=Hercules图形
		3=4-面板
		4=打包（Packed）像素
		4=打包（Packed）像素
		5=256色非链4
		6~F=为将来的VESA模式而保留
		10~FF=供应商定义
1C	字节	簇的大小，单位千字节——对于那些带有扫描线组的图形模式来说，每个簇内存大小要增加1KB，对于那些不支持bank的模式，值置为0。
1D	字节	图像页的数目——视频器为该模式同时处理的总页数。
1E	字节	页面功能（总设置为1）。
1F	224字节	未使用——缓冲区余下的部分为将来的说明而保留，大多数SVGA BIOS将在该区域全部返回零。

调用：

ax=4F01h

cx=来自功能0的视频模式号

es:di=放置信息的缓冲区

返回：

如果成功：

ax=004Fh

es:di=装入信息捕捞缓冲区

如果支持这个功能，但模式无效

ax=014Fh

如果不支持功能，则al不是4Fh

子功能	描述	中断	功能
AL=2	设置 VSGA 视频模式	10h	AH=4Fh

改变视频模式，这个功能支持 VGA 类型的 8 位模式（将第 14~8 位置 0）和 15 位 SVGA 模式。参看子功能 0 和 1 以了解 SVGA 模式及支持。参看表 9-2 以了解常见的视频模式。

调用：
ax=4F02h
bx,第 15 位=0 清除视频内存
1 保持视频内存不变
第 14~0 位 = 视频模式号

返回：
如果成功：
ax=004Fh
如果支持功能，但模式无效
ax=014Fh
如果不支持功能，则 al 不是 4Fh

子功能	描述	中断	功能
AL=3	获取当前的 SVGA 视频模式	10h	AH=4Fh

返回当前的视频模式，包括标准的 VGA 模式 0~13h，以及新式的 15 位 SVGA 模式。该子功能不返回视频清除位。参看中断 10，功能 F 以获取视频清除位。

调用：
ax=4F03h

返回：
如果成功：
ax=004Fh
bx,第 15 位=0（固定为 0）
14~0 位=视频模式号
如果支持功能，但操作失败
ax=014Fh
如果不支持功能，则 al 不是 4Fh

子功能	描述	中断	功能
AL=4,DL=0	获取保存状态缓冲区的大小	10h	AH=4Fh

获取保存 SVGA 状态所需的缓冲区大小。参看下面的两个子功能以了解保存和恢复 SVGA 状态的细节。

调用：
ax=4F03h
cx=保存类型
位 15~4=0 未使用

3=1	保存 SVGA 状态
2=1	视频 DAC（颜色寄存器）
1=1	视频 BIOS 数据状态
0=1	视频硬件状态

dx=0

返回：

如果成功：

ax=004Fh

bx=缓冲区 64 字节块的数目

如果支持功能，但操作失败

ax=014Fh

如果不支持功能，则 al 不是 4Fh

子功能	描述	中断	功能
AL=4,DL=1	保存 SVGA 状态	10h	AH=4Fh

在用户提供的缓冲区中保存 SVGA 的当前状态。参看前一个子功能，获取保存状态缓冲区的大小，以得到需要的缓冲区大小，若要保存 SVGA 适配器的全部状态，将 CX 的值取为 000F。该子功能不保存视频显示内存的内容。

调用：

ax=4F04h

cx=保存类型

位		
15~4=0	未使用
3=1	保存 SVGA 状态
2=1	保存视频 DAC（颜色寄存器）
1=1	保存视频 BIOS 数据状态
0=1	保存视频硬件状态

dx=1

es:bx=指针，指向用户提供的缓冲区

返回：

如果成功：

ax=004Fh

es:bx=装入了保存状态数据缓冲区

如果支持功能，但操作失败

ax=014Fh

如果不支持功能，则 al 不是 4Fh

子功能	描述	中断	功能
AL=4,DL=2	恢复 SVGA 状态	10h	AH=4Fh

恢复以前保存的状态。为了恢复 SVGA 适配器的全部状态，将 CX 的值取为 000F。该子功能假定状态以前保存在用户提供的缓冲区中，并且使用相同的保存类型 CX 来保存和

恢复状态。该子功能不恢复视频显示内存的内容。

调用：
ax=4F04h
cx=保存类型

位		
	15~4=0	未使用
	3=1	恢复 SVGA 状态
	2=1	恢复视频 DAC（颜色寄存器）
	1=1	恢复视频 BIOS 数据状态
	0=1	恢复视频硬件状态

dx=2
es:bx=指针，指向保存状态缓冲区

返回：
如果成功：
ax=004Fh
如果支持功能，但操作失败
ax=014Fh
如果不支持功能，则 al 不是 4Fh

子功能	描述	中断	功能
AL=5,BH=0	设置 SVGA 内存窗口位置	10h	AH=4Fh

该子功能设置指定窗口 A 或 B 在内存中的位置。SVGA 窗口模式的能力由中断 10 功能 4F01h 获得。功能 4F01h 显示了窗口的颗粒大小，以及段址对功能 4F01h 所返回的窗口体系功能指针进行远调用，也可以实现相同的功能。参看功能 4F01h 以了解有关直接调用本功能的额外信息。

调用：
ax=4F05h
bh=0
bl=0 给窗口 A，1 给窗口 B
dx=内存中的窗口位置（单位：窗口颗粒单元）

返回：
如果成功
ax=004Fh
如果支持功能，但是操作失败：
ax=014Fh
如果不支持功能，则 al 不是 4Fh

子功能	描述	中断	功能
AL=5,BH=1	获取 SVGA 内存窗口位置	10h	AH=4Fh

该子功能获取指定窗口，A 或 B，在内存中的位置。

对功能 4F01h 所返回的窗口体系功能指针进行远调用，可以实现相同的功能。参看功

能 4F01h 以了解有关直接调用本功能的额外信息。

调用：　　ax=4F05h
　　bh=1
　　bl=0 给窗口 A，1 给窗口 B
返回：　　如果成功
　　ax=004Fh
　　dx=窗口在内存中的位置（单位：窗口颗粒单元）
　　如果支持功能，但是操作失败：
　　ax=014Fh
　　如果不支持功能，则 al 不是 4Fh

子功能	描述	中断	功能
AL=6,BL=0	设置逻辑扫描线的长度	10h	AH =4Fh

SVGA 支持逻辑显示大于实际显示，该子功能设置逻辑显示的大小，在图形和文本模式下都有效。在所有模式下，有必要提供以像素为单位逻辑宽度。在文本模式下，这意味着要将列数和字符格的宽度乘起来，可以从子功能 4F01h 中获得字符格的宽度。

如果所要求的宽度超过了视频适配器或模式所能提供的宽度，就会选择最大允许的宽度，并返回实际选择的信息。参看功能 4F07h 来指定显示窗口的起点位置。

调用：　　ax=4F06h
　　bl=0
　　cx=所期望的宽度，单位是像素
返回：　　如果成功
　　ax=004Fh
　　bx=每条逻辑扫描线的字节数
　　cx=每条扫描线的实际像素数
　　dx=扫描线的最大数目
　　如果支持功能，但是操作失败：
　　ax=014Fh
　　如果不支持功能，则 al 不是 4Fh

子功能	描述	中断	功能
AL=6,BL=1	获取逻辑扫描线的长度	10h	AH =4Fh

获取逻辑和实际扫描线的宽度信息。参看前一个子功能以获得另外的细节

调用：　　ax=4F06h

bl=1

返回：　如果成功

ax=004Fh

bx=每条逻辑扫描线的字节数

cx=每条扫描线的实际像素数

dx=扫描线的最大数目

如果支持功能，但是操作失败：

ax=014Fh

如果不支持功能，则 al 不是 4Fh

子功能	描述	中断	功能
AL=7,BL=0	设置逻辑显示起点	10h	AH =4Fh

设置较大逻辑显示的位置，该显示出现在可见的屏上区域内。逻辑页中指定位置的左上角显示可见屏的左上角。参看子功能 4F06h 来定一个大于可见显示的逻辑显示。

设置逻辑显示起点对文本和图形模式都有效。在所有模式下，有必要提供以像素为单位的位置。对于文本模式，这意味着要将列位置号和字符格宽度乘起来，将行位置号和字符格高度乘起来。可以从子功能 4F01h 中获得字符格的宽度和高度。

这个功能支持光滑的硬件摇摄和翻滚功能。也可以用来从一个较大的逻辑屏中显示完整的屏幕以产生动画和特殊效果。

调用：　ax=4F07h

bx=0（bh 保留，必须置为 0）

cx=dx 中扫描线显示的第一个像素在逻辑显示中的位置

dx=显示的第一条扫描线在逻辑显示中的位置

bl=1

返回：　如果成功

ax=004Fh

如果支持功能，但是操作失败：

ax=014Fh

如果不支持功能，则 al 不是 4Fh

子功能	描述	中断	功能
AL=7,BL=1	获取逻辑显示起点	10h	AH =4Fh

获取当前较大的逻辑显示的位置，该逻辑显示正出现在显示的可见区域。参看前一个子功能 4F07h，BL=0，以了解其他细节

调用：　ax=4F07h

bl=1

返回：　如果成功

ax=004Fh

bx=0

cx=dx 中扫描线显示的第一个像素在逻辑显示中的位置

dx=显示的第一条扫描线在逻辑显示中的位置

如果支持功能，但是操作失败：

ax=014Fh

如果不支持功能，则 al 不是 4Fh

子功能	描述	中断	功能
AL=4D	视频光标接口请求	10h	AH=4Fh

这个功能是 VESA 标准的一部分，用来在所有的 VGA 和 SVGA 视频模式显示鼠标光标。VCI 标准提供了一个通用的接口，以支持鼠标驱动程序对 SVGA 硬件光标的支持。该标准的目的是为了让所有目前和将来的视频模式都可由单独的一个鼠标驱动程序所支持。鼠标驱动程序和 SVGA 卡处理操作，而应用程序不可用任何功能。

VESA 标准#VS911021 详细地介绍了完整的视频光标接口标准（参考附录 C）。

调用：　ax=4F4Dh

bx=VCI 可用的 RAM 缓冲区的字节数

ds:0=VCI 用到的 RAM 缓冲区的起点地址

es:di=VCI 驱动程序回调功能的地址

返回：　如果成功

ax=004Fh

cx=VCI 实际使用的字节数

es:di=调用 VCI 处理程序的指针

如果支持功能，但是操作失败：

ax=014Fh

如果不支持功能，则 al 不是 4Fh

中断	功能	描述	平台
10h	FEh	获取再分配屏幕地址	所有

这个功能从 RSIS 环境中获取再分配屏幕地址。程序必须先为当前模式确定缺省的显示段址，A000、B000 或 B800。本功能传递这个段址值，来确定是否使用一个可替换的再分配地址。它为所有的后续操作返回地址，即使系统不支持 RSIS。

某些系统，比如日本使用的 IBM DOS/V，必须使用这个功能，否则屏幕上什么也不会出现。DOS/V 只为视频模式 3 返回一个再分配地址。某些环境，例如 Desq View 和 Memort

Commander，会向服从 RSIS 的应用程序提供 300K 的额外的 DOS 内存，所以使用它时可能会很有利。本章后面有一整节讨论了 RSIS。

即使这个功能要求有一个偏移量，但是在所有情况下必须传递值为零的偏移量。虽然将来可能会设计出非零的偏移量，但目前似乎不可能。

调用：　　ah=FEh
　　es:di=指针，指向非再分配的显示基地址
返回：　　es:di=指针，指向直接写屏幕时用到的实际基段址：偏移量

中断	功能	描述	平台
10h	FFh	更新再分配屏幕	所有

这个功能命令环境更新屏幕。这个功能将字符信息取自再分配屏幕缓冲区，然后将它们写到屏幕上合适的窗口区。

目前，我知道只有两种环境要求这个功能。IBM 和 Microsoft 发布了 DOS/V，这是为日本专门设计的 DOS 版本，它允许任何标准的 VGA 适配器显示 DBCS 日本字符集。在任何时候，显示适配器维持在图形模式。可以转化 DOS 下基本文本的程序以工作在 DOS/V 环境下。程序所要做的一切只是触发中断 10h，功能 FEh 来获得再分配屏幕，并且在任何时候如果屏幕发生了改变就触发该调用。

最简单的支持可以带 CX=780h 来触发这个功能，以便一次更新 24 条线。仅仅只是更新这些改变了的区域而不是整个屏幕，将极大地提高运行速度，这个功能（如果用于 DOS/V），仅用于视频模式 3 下，而忽略所有的其他模式。

IBM 的非常过时的 Top View 程序也使用这个功能在屏幕上为不同的应用程序显示多个窗口。

调用：　　ah=FFh
　　cx =已经修改了的字符数目（每个 DBCS 字符计作二）
　　es:di=指针，指向改变了的再分配视频缓冲区的首字符
返回：　　更新显示

其他与视频系统相关的中断

中断	功能	描述	平台
42h	xx	老式中断 10h	EGA+

如果一个视频适配器有自带的 BIOS 代码，那么系统 BIOS 会在 POST 期间运行适配器的初始化代码。视频 BIOS 将老式中断 10h 视频服务指针传递给中断 42h 视频 BIOS 然后用

指向自己的中断 10h 服务程序的指针替换中断 10h 向量。

如果程序调用的中断功能不被适配器的 BIOS 所支持，那么会将它传递给中断 42h，系统视频 BIOS。如果系统 BIOS 仍不支持，就会忽略它。如果某系统有两个视频适配器，其中一个是 CGA 或 MDA/Hercules 适配器，与这些适配器相关的功能通常使用系统 BIOS。

通常，只有中断 10h 才能被所有的视频操作所触发，应该避免使用中断 42h。

中断	功能	描述	平台
6Dh	xx	替换中断 10h	VGA+

这是直接调用 VGA 适配器的一种替代方法。如果中断 10h 被许多 TSR 挂起，那么本中断将不会通过那些 TSR 传递控制。

我没有发现使用中断 6Dh 的任何理由，由于它未加说明，所以很容易想到某些 VGA 复制品不支持中断 6Dh。

在许多 VGA 适配器中，视频 BIOS 中断 10h 仅仅只是触发中断 6Dh。

重定位屏幕接口规范（RSIS）

将应用程序的屏幕缓冲区移动到内存的不同部分有许多好处。一系列的环境可以提供多种同时使用屏幕区域的方法，或者为应用程序恢复额外的内存，移动后的屏幕缓冲区被称为是重定位的。

使用 BIOS 和系统服务来读写屏幕的程序无需修改，并且缺省情况下都是服从 RSIS 的。然而大多数的商业程序直接向屏幕区域写以获得最大运行速度，所采用的方法是早期的 IBM PC 所有的网格定位法，IBM 从段址 0A000h 开始指定了 128K 的视频内存。

该说明为所有的应用程序提供了一种很好的途径来利用再分配屏幕，取决于环境所提供的服务类型，应用程序的执行速度和特性会有显著的提高。执行时，服从 RSIS 的应用程序将保持和 DOS 环境的兼容。服从 RSIS 的程序对应用程序用户完全透明，不需要其他的输入来利用这些特性。

创建一个服从 RSIS 的应用程序非常容易，通常只须作少量的修改，这取决于程序是如何使用视频系统的。本章的后面部分包括了一些用 C 和汇编语言编写的例子。

硬件与软件方面的考虑

创建一个服从 RSIS 的应用程序时不会出现兼容性问题。转化之后，应用程序保持和所有工业标准兼容的 PC 兼容。其中包括 PC、XT、AT、EISA 以及 MCA。RSIS 还支持某些程序在某些不兼容于 IBM 的日本系统上正常的工作。

创建一个服从 RSIS 程序

创建你自己的服从 RSIS 程序有二个基本步骤。本章的后面部分提供了用 C 语言和汇编代码编写的详细代码例。它们可以改写成任何语言。

- 第 1 步：程序开始时，程序的初始化代码必须确定，使用 RSIS 功能调用的当前的屏幕地址在何处；
- 第 2 步：所有的直接屏幕写必须使用第 1 步中所确定的屏幕基地址；
- 第 3 步：如果改变了视频模式，那么必须获得新的屏幕地址。如果让频模式需要改变成图形模式，那么必须先检查当前的环境条件下是否支持该改变；
- 第 4 步：如果使用多个文本页，必须检查所需要的页是否可用；
- 第 5 步：仅对 DOS/V 兼容系统而言，如果改变了显示，必须触发中断 10h，功能 FFh 来更新实际的显示。

RSIS 功能调用

中断 10h 的功能 Feh 如下这般提供了关键的信息：

调用：	ah=FEh es:di=指针，指向非再分配的基地址（A000:0,B000:0,或 B800:0）
返回：	es:di=指针，指向直接屏幕写所用到的实际段址：偏移量

非再分配显示地址就是你为直接写屏幕所确定的显示地址。EGA/VGA 图形的段址是 A000，单色文本和 Hercules 图形的段址是 B000，而彩色文本和 CGA 图形的段址是 B800，表 9-2 列出了不同视频模式的缺省显示段址，偏移量必须是零。

中断 10h 功能 FEh 必须在程序起点和改变了视频模式之后触发，因在文本与图文之间切换时，RSIS 环境可能需要改变再分配地址。

如果存在一种环境来重定位视屏，那么环境会截获这个中断功能并在 ES:DI 中返回实际的重定位视屏。RSIS 提供商负责管理重定位进程，而不是应用程序。

BIOS 不提供这个功能。如果不存在 RSIS 环境，BIOS 将不会改变寄存器，而将 ES:DI 指向正确的屏幕地址。

特殊考虑

非标准的视频模式 由于在非标准的视频适配器上有许多冲突模式，所以没有人尝试使用这些模式。如果使用 IBM 定义的模式 13h 以上的模式，并且以 A000 作为段址，那么

提供的程序会正常工作并保持 RSIS 兼容。

VESA 标准 视频电子标准协会（VESA）为视频硬件制造商制定了一个标准，来支持 VGA 模式 13h 以上的视频模式和分辨率。如果你打算使用这些模式，你就必须彻底地理解 VESA 的超级 VGA 说明书。VESA 模式的设置方式与标准模式不同，VESA 超级 VGA 说明书详细地说明如何检测卡是否支持这些模式通信和如何设置这些模式。如果你命令 VESA 兼容卡使用 A000 作为视频段的起点，那么在设置了 VESA 模式之后，可以使用下列程序段：

```
mov     dx,  0A000h              ;屏幕缓冲区的起点
mov     es,  dx                  ;将 es:di 设置到没有改变的屏幕段地址
xor     di,  di                  ;获得可替换的缓冲区地址
int     10h                      ;获得 es:di 的新的寄存器值，或者保持不变
mov     cs:sceen_segment,es      ;保存屏幕的段
mov     cs:screen_offset,di      ;保存屏幕的偏移量
```

XGA 和 8514/A 使用 XGA 和 8514/A 适配器的 VGA 特性时，所有列出的 RSIS 代码都会正确地工作。如果你正在编写的代码使用 8514/A 或 VGA 的非 VGA 特性，没有办法来重新引导这些非 VGA 模式的屏幕缓冲区。

Hercules 适配器 Hercules 单色和 Hercules Incolor 卡的适配器使用显示器 I/O 状态寄存器来表示系统中是否存在该卡。这些卡提供了两页使用段址 B000h 的图形。在使用 Hercules 专用图形期间，视频模式 7 保持有效。所有的程序将会同这些卡一起正确的发挥作用。

代码例 9-1 获取重定位屏幕地址，C 代码

大多数程序在初始化时确定屏幕缓冲区的驻留地址。常用的方法是检查是否使用模式 7（单色），如果在使用，就将 B000:0000h 作为缓冲区的起点。为了使程序服从 RSIS，在确定了段址之后，调用中断 10h 的功能 FEh 来确定是否应该使用下个替代的地址。下面的代码例就是一个运行在初始化期间的小程序。该代码设置了实际的段址和偏移量以用在程序的剩余部分中。

```
regs.x.ax = 0×0F00;
int86x（0×10, &regs, &sregs）;      //  中断 10h, 获取视频模式
regs.x.ax = regs.x.ax & 0×007F;     //  屏蔽掉一些位
if（regs.x.ax == 0×0007）
    regs.x.es = 0×B00h              //  缺省的单色模式
 else regs.x.es  == 0×B800h;        //  缺省的彩色文本段
 regs.x.ax = 0×FE00;
```

```
regs.x.di = 0;                          //  缺省的缓冲区偏移量
int86×(O×10,&regs,&regs,&sregs);        //  中断 10h，获取备用缓冲区
screen_segment = sregs.es;              //  屏幕缓冲区段
screnn_offset = regs.x.di;              //  屏幕缓冲区偏移量
```

代码例 9-2　获取重定位屏幕地址，汇编代码

不改变视频模式（基于文本）

大多数程序在初始化时确定屏幕缓冲区的驻留地址。常用的方法是检查是否使用模式 7（单色），如果在使用，就将 B000:0000h 作为缓冲区的起点。为了使程序服从 RSIS，在确定了段址之后，调用中断 10h 功能 FEh，来确定是否应该使用下个替代的地址。下面的代码例就是一个运行在初始化期间的小程序。该代码设置了实际的段址和偏移量以用在程序的剩余部分中。

```
;--------------------------------------------------------------------
;   屏幕初始化
;       获取直接显示写的当前屏幕缓冲区地址
;       writes.
;
;       调用:           无
;
;       返回:           设置 cs:screen_segment
;                       设置 cs:screen_offset
;
;       用到的寄存器: 无

screen_segment  dw   0                  ; 屏幕缓冲区段址
screen_offset   dw   0

init_screen proc        near
        push    ax                      ; 保存所有的用到的寄存器
        push    bx
        push    dx
        push    di
        push    es
```

```
        mov     ah, 0Fh                 ; 获取视频模式功能
        int     10h                     ; 将当前视频模式存入 al, 这个功能
                                        ;   还会影响 bh。
        and     al, 7Fh                 ; 屏蔽掉第 7 位（不是模式的一部分）
        mov     dx, 0B000h              ; 单色缓冲区地址
video_skip:
        mov     es, dx                  ; 将 es:di 设置成未改变的屏幕段地址
        xor     di, di                  ;   segment address
        mov     ah, 0FEh                ; 获取备用缓冲区地址
        int     10h                     ; 获取 es:di 新段或者保护不变

        mov     cs:screen_segment, es   ; 保存屏幕段
        mov     cs:screen_offset, di    ; 保存屏幕偏移量

        pop     es                      ; 恢复所有使用的寄存器
        pop     di
        pop     dx
        pop     bx
        pop     ax
        ret
init_screen     endp
```

改变视频模式的程序（文本和图形）

下面的子程序可用于改变视频模式。它可以简单地替换所有使用中断 10h 而功能 AH=0 的程序。

与标准模式设置不同，如果尝试了图形模式但是没有设置，该子程序就会设置进位标志。如果期望的适配不存在，或者当前的屏幕内存分配不足以设置期望的视频模式，就会出现这种情况。

某些环境可以减少可用的显示内存，以阻止使用某个特殊的视频模式。如果返回时设置了进位标志，那么该模式不可用，而应该使用需要较少内存的视频模式，或者显示一个错误信息。

该子程序首先检查是否有独特的 EGA+模式、0Dh 或以上的。如果模式号低于 0Dh，就设置该模式。如果尝试一个独特的 EGA+图形模式 0Dh 或以上，子程序会先检查是否支持。如果支持，就改变模式，并设置屏幕缓冲区地址。如果不支持，就不设置该模式，而

设置进位标志。

```
;--------------------------------------------------------------------------
;  保护视频模式
;        用程序替换标准的 BIOS
;        中断 10h 功能 0，模式设置。
;        所有的直接屏幕写更新屏幕地址。
;        查模式的支持和标识情况。
;
;        调用:          al = 视频模式
;
;        返回:          如果不支持所请求的命令
;                         则进位 = 1
;                       如果支持所请求的命令
;                         则进位 = 0
;                         设置 cs:screen_segment
;                         设置 cs:screen_offset
;
;         用到的寄存器:   无

screen_segment  dw      0               ; 屏幕缓冲区段
screen_offset   dw      0               ; 屏幕缓冲区偏移量


set_mode proc   near
        push    ax                      ; 保存所有用到的寄存器
        push    bx
        push    cx
        push    di
        push    es


        mov     bl, al                  ; 获取新模式
        and     bl , 7Fh                ; 屏蔽最高位
        cmp     bl, 0Dh                 ; 仅 EGA/VGA 模式?
        jb      set_the_mode            ; 若不是，则跳转
```

```
        mov     ah, 12h                 ; 备用选择功能
        mov     bl, 10h                 ;EGA/VGA 获取信息子功能
        mov     bh, 0FFh                ; 检查值
        int     10h                     ; 检查是否为 EGA/VGA（改变 cx）
        cmp     bh,1                    ; 若值大于 1，则不是 EGA/VGA
        ja      no_mode_set             ; 如果不是 EGA/VGA 图形模式则跳转
set_the_mode:
        mov     ah, 0                   ; 设置新模式功能
        int     10h                     ; 要设置的模式保存在 al 中

        mov     bl, al                  ; 保存模式
        and     bl, 7Fh                 ; 屏幕最高位
        mov     ax, 0A000h              ; 普通的 EGA/VGA 图形段
        cmp     bl,0Dh                  ; 模式 D 以上?
        jae     check_alternate         ; 如果是，则跳转
        mov     ax, 0B800h              ; 彩色文本或 CGA 图形
        cmp     bl, 7                   ; 单色文本?
        jne     check_alternate         ; 如果不是单色则跳转
        mov     ax, 0B000h              ; 普通的单色屏幕段

check_alternate:
        mov     es, ax                  ; 将 es:di 设置成未改变的屏幕
        xor     di, di                  ;    段地址
        mov     ah, 0FEh                ; 获取备用缓冲区地址
        int     10h                     ; 获取 es:di 新段或保持不变

        mov     cs:screen_segment, es   ; 保存屏幕段
        mov     cs:screen_offset, di    ; 保存屏幕偏移量
        clc                             ; 清除进位标志，模式设置成功
        jmp     short set_done
no_mode_set:
        stc                             ; 现在不支持 EGA/VGA 模式
set_done:
        pop     es                      ; 恢复所有用到的寄存器
```

```
        pop     di
        pop     cx
        pop     bx
        pop     ax
        ret
set_mode  endp
```

代码例 9-3　写显示器内存

下面摘选展示了从一个缓冲区写的文本屏幕。这里有许多处理直接写屏幕的方法，这个例子是其中的一种。该代码假定以前获得了屏幕的段址和偏移量。这个小程序不支持早期的 CGA 适配器的雪花问题。在 486 或更好的 CPU 上快速执行时，循环指令可以用两条指令替换，它们是 dec cx 和 jnz next_char。

```
        mov     es, cs:screen_segment           ; 屏幕缓冲区的驻留地址
        mov     di, cs:screen_offset
        mov     ds, segment program_screen      ; 程序屏幕的地址
        mov     si, offset program_screen       ;  80×25 文本字符
        mov     cx, 80*25                       ; 屏幕的大小
        mov     al, 7
        cld
next_char:
        movsb                                   ; 将字符传送到屏幕
        stosb                                   ; 将属性传送到屏幕
        loop    next_char
```

DOS/V 的屏幕更新功能

新式的日本操作系统 DOS/V，总是运行在图形模式下，尽管大多数程序在运行时就好象是文本模式一样。DOS/V 要求使用重定位屏幕地址，并且在需要更新屏幕时，必须调用屏幕更新功能。下面的代码片断告诉 DOS/V 需要更新整个屏幕。如果只改变部分屏幕，那么将极大地提高运行速度。参看中断 10h，功能 FFh 以了解更多信息。

```
        mov     ah,  0FFh
        mov     cx,  1920h              ; 更新 24×80 字符
        mov     bx,  screen_offset      ; 缓冲区起点
```

```
        mov     es, screen_segment      ;
        int     10h                     ; 更新屏幕
```

如果没有 DOS/V，就会忽略中断 10h 的功能 FFh，而什么也不做。

多页面

写多个页面时，程序可以改变屏幕的段值或屏幕的偏移值来向正确的页面写，只要简单地将（页大小）×（页号）加入到屏幕偏移量中，就可以实现这一点。例如，如果需要在模式 3 下向另外一页（第 1 页），并且页面大小是 1000h 字节，则只要将 1000h 加入到屏幕偏移量中就可以得到写屏幕的偏移量。

在某些环境下，会减少屏幕缓冲区的大小直到小于标准大小。例如，如果在文本模式 3 下分配的屏幕缓冲区只有 8K，那么只可以分配两页 80×25。为了知道是否允许所期望的页，可以试着激活该页，然后检查该页是否被激活。如果页面没有改变，那么表示不允许所请求的屏幕页。下面的例子尝试从上次模式设置后的缺省页（第 0 页）进入第 1 页。

```
        mov     ah, 5                   ; 选择活动显示页
        mov     al, 1                   ; 设置成第 1 页
        int     10h                     ; 执行!

        push    ax
        mov     ah, 0Fh                 ; 读取当前的显示状态
        int     10h                     ; 获取 bh 中的活动页
        pop     ax                      ; 恢复 al 中所期望的页
        cmp     al, bh                  ; 当前页面正确吗?
        je      page_set_ok             ; 如果是，则跳转

page_not_set:
        .                               ; 处理页面不可用的代码
        .
page_set_ok:
        .                               ; 设置新页面的代码
```

一旦发现某页不用，所有编号更低的页就都可用，并且在整个程序运行期间都可用。一旦知道某页有效后，就没有必要再次检查其可用性，除非更变了视频模式。

环境是如何提供 RSIS 支持的

这一部分供环境开发人员参考，他们希望在他们开发的环境中提供重定位屏幕服务。应用程序开发人员可以略过这一部分。

责　任

提供重定位屏幕地址的环境，负责提供屏幕数据区写入的正确的段址和偏移量。该替代地址必须有足够的内存，来执行程序所指定的所有功能，并且能处理所有应用程序适宜的模式。

环境必须解释中断 10h，功能 0FEh 来提供正确的屏幕地址而不改变未使用的寄存器。中断 10h 功能 FEh 描述了这个功能的操作。

RSIS 环境

有三种已知类型的环境需要重定位视频缓冲区地址：多应用程序、内存管理和支持外国语言。

多应用环境将应用程序的视频缓冲区重定位到一个 RAM 地址，该 RAM 地址为每个运行的应用程序所独占。这支持环境控制在显示器上显示应用程序的哪一部分。这样，应用程序看到的是一个完整的视频屏幕，但是多应用环境控制了实际上在屏幕上显示多少应用程序屏幕信息。

第一次装入多应用环境时，环境的代码也必须使用中断 10h 功能 0FEh 来确立重定位缓冲区地址。无论何时改变模式，多应用环境必须重新确立重定位缓冲区地址。

内存管理环境将视频内存的物理地址改变到一个更高的地址。这样做是为了获得额外的主存。也可以控制可用视频内存的大小以回收特定程序不必要的未用视频内存。

内存管理截获中断 10h 功能 FEh，只能发生在中断链的最低层次上。如果系统既有多应用环境又在运行内存管理，那么内存管理将无法直觉直接来自于应有程序的变更屏幕地址请求。

需要使用 RSIS 的第三种情况是，有些操作系统需要一次在屏幕上显示多于 512 个的字符。DOS/V 就使用了这种方案。DOS/V 将显示保持在图形模式下，这样就可以显示成千的日本字符。使用两个字节来描述 ASCII 和日本字符。应用程序将字符值写入到重定位屏幕缓冲区，并由操作系统转换后显示在图形屏幕上。

使用 RSIS 的环境

下面所列的部分环境列表支持某些或所有的 RSIS 特性。一个服从 RSIS 的应用程序在这些环境下使用时，可以改进功能和/或提高运行速度。

环境	出版商	描述
DOS/V*	IBM 公司	DOS，在 VGA 上支持日本字符
Top View*	IBM 公司	多应用*
Desq View	Quatterdeck 办公系统	多应用
Memory Commander	V 通信公司	386/486/pentium 内存管理和 DOS 扩展

注：* Top View 和 DOS/V 要求一个次级中断功能来更新屏幕，参看中断 10h，功能 FFh。

代码例 9-4 检测视频适配器

下面的子程序找出正确的适配器类型，类型范围从 MDA 到 VESA XGA。基于显示器和用户对系统所作的设置，这个程序还可以确定程序应该使用的属性范围。对于某些适配器，这个子程序还返回一个指针，指向带有供应商名称的串。

支持的三种属性类型分别是彩色、单色和灰度。不要在视频缓冲区出现的地方使用这些信息。它只用于传递屏幕上使用的属性字节。有少数情况下，使用的是单色或灰度属性，然而视频缓冲区的段址却是 B800h，这些地方通常是作为彩色文本视频内存区使用的！

灰度显示器是和 VGA 一起引入的，但是却没有向 VGA 那样流行。大多数的但是膝上型电脑支持灰度模式。许多应用程序忽视了对灰度的支持，并且不正确地假定显示器为彩色，因此往往难于读取显示器。如果你不想专门地支持灰度操作，在返回灰度属性值后，请只用黑、白和亮白三种属性。

VIDEO_TEXT 源代码放在 SYSTYPE 程序中。在某个系统中运行 SYSTYPE 的结果如图 9-1 所示。

```
系统分析                    V1.00（C）1994,1996 FVG
视频适配器: VGA，程序应使用彩色属性
视频供应商: Orichid Technology Fahrenheit VA
```

图 9-1 SYSTYPE 的视频结果

```
; ----------------------------------------------------------------------
;    视频类型检测
;
;        找出视频类型、类型属性以及可能有
;        的供应商串。本程序假定显示器位于
;        文本模式来确定属性字节CL。
;
;        属性选项表示程序应该使用彩色、
;        单色还是灰色模式。
```

```
;
;       调用:           无
;
;       返回:           al = 视频类型
;                           0 = MDA
;                           1 = HGA
;                           2 = CGA
;                           3 = MCGA
;                           4 = EGA
;                           5 = VGA
;                           6 = SVGA
;                           7 = XGA
;                           8 = VESA XGA
;                       ch = 属性类型
;                           0 = 彩色
;                           1 = 单色
;                           2 = 灰度（某些MCGA或VGA+）
;                       cl = es:bx中的供应商串
;                           0 = 无供应商串
;                           1 = 有供应商串
;                       es:si = 供应商串，零结尾（如果cl=1）
;                           (if cl = 1)
;
;       用到的寄存器:   ax, cx, si, es

infobuf db      256 dup (0)         ; 视频信息缓冲区
hercstr db      'Hercules', 0       ; Hercules的供应商串

video_type proc     near
        push    bx
        push    dx
        push    di
        push    bp
        mov     bp, 4               ; bp = 临时视频类型, 4=EGA

; --- 使用获取视频信息功能检查是否为EGA+
```

```
        mov     ah, 12h                 ; 获取视频信息功能
        mov     bh, 5Ah                 ; 测试值
        mov     bl, 10h                 ; 子功能EGA+信息
        int     10h
        cmp     bh, 1                   ; 必须为0，彩色。或者1，单色
        ja      below_EGA               ; 如果不是EGA+则跳转

; --- 是EGA+，然后测试是否为EGA

        push    bx                      ; 保存色彩信息供后面使用
        mov     ax, 1A00h               ; 获取显示模式
        int     10h
        cmp     al, 1Ah                 ; 支持功能吗?
        je      vid_VGA                 ; 如果支持，则至少是VGA
        jmp     type_found              ; 若不支持，则跳转（一定是EGA）

; --- 至少是VGA，现在测试是否为SVGA

vid_VGA:
        inc     bp                      ; 假定是VGA（5）
        push    cs
        pop     es
        mov     di, offset infobuf      ; 视频信息缓冲区
        mov     ax, 4F00h               ; 返回SVGA信息
        int     10h
        cmp     al, 4Fh                 ; 支持功能吗?
        jne     XGA_test                ; 如果不是SVGA则跳转
        inc     bp                      ; 假定是SVGA（6）

; --- 至少是一个VGA/SVGA，现在测试是否为XGA/VESA XGA

XGA_test:
        mov     ax, 1F00h               ; 获取XGA信息大小
        int     10h
        cmp     al, 1Fh                 ; 支持功能吗?
```

```
        jne     type_found              ; 若不支持，则跳转，是VGA或SVGA
        mov     bp, 7                   ; 设置为XGA

; --- 至少是XGA，现在测试是否为VESA XGA

        mov     di, offset infobuf      ; 视频信息缓冲区
        mov     ax, 4E00h               ; 返回VESA XGA信息
        int     10h
        cmp     ax, 004Eh               ; 支持功能吗?
        jne     type_found              ; 若不支持，则跳转
        inc     bp                      ; VESA XGA (8)
        jmp     type_found

; --- 如果适配器类型低于EGA，则运行到这里

below_EGA:
        mov     bp, 3                   ; 假定是MCGA（3）
        mov     ax, 1A00h               ; 获取显示模式
        int     10h
        cmp     al, 1Ah                 ; 支持功能吗?
        jne     vid_not_MCGA            ; 如果不是MCGA则跳转
        mov     ah, 0Fh
        int     10h                     ; 获取视频模式
        mov     cx, 100h                ; 假定是单色，无供应商
        cmp     al, 7
        je      vid_chk_gray            ; 如果是单色，则检查灰度
        mov     ch, 0                   ; 像彩色
        je      vid_chk_gray            ; 检查灰度级

; --- 不是MCGA，测试是CGA还是MDA/HGA

vid_not_MCGA:
        dec     bp                      ; 假定是CGA（2）
        mov     ah, 0Fh                 ; 获取视频模式
        int     10h
        cmp     al, 7                   ; 模式7单色
```

```
        je      vid_mono_type           ; 如果是，则跳转
        xor     cx, cx                  ; 返回彩色，无供应商
        jne     vid_mono_type
        jmp     vid_mono_chk            ; 一定是CGA

; --- 一定是MDA或HGA，现在找出是哪一个。
;       HGA会翻转MDA上的一个未定义位。
;       检查看看这一位是可以改变状态10次
;       甚至更多次数。

vid_mono_type:
        dec     bp                      ; 假设为HGA（1）
        xor     bl, bl                  ; 起始计数为0
        mov     dx, 3BAh                ; HGA/MDA上的状态端口
        xor     ah, ah                  ; ah用来保存以前的状态
        mov     cx, 0FFFFh              ; 测试很长一段时间
vid_loop:
        in      al, dx                  ; 读状态端口
        IODELAY
        and     al, 80h                 ; 分离HGA的翻转位
        cmp     al, ah                  ; 改变了吗?
        je      vid_no_toggle           ; 没有，则跳转
        inc     bl                      ; 位改变了，增加
        cmp     bl, 10                  ; 多余10次翻转
        jae     vid_herc                ; 若是，则是Hercnles卡
vid_no_toggle:
        loop    vid_loop                ; 再次读直到cx为0

; --- 如果是MDA，则失败（位不会翻转）

        dec     bp                      ; 设置为MDA（0）
        mov     cx, 100h                ; 单色属性，无供应商
        jmp     vid_exit

; --- 适配器是Hercules类型
```

```
vid_herc:
        mov     cx, 101h            ; 单色属性
        push    cs
        pop     es
        mov     si, offset hercstr  ; 设置串为Hercules
        jmp     vid_exit

; --- 对于MCGA/EGA/VGA/XGA, 将属性类型放在CL中

type_found:
        pop     bx                  ; 获取属性类型（0或1）
        mov     ch, bh
vid_chk_gray:
        mov     ax, 1A00h           ; 读取显示代码
        int     10h
        cmp     al, 1Ah             ; 检查是否支持
        jne     vid_string          ; 如果不是灰度级则跳转
        cmp     bl, 7               ; 灰度级监视器
        je      vid_is_gray         ; 如果是，则跳转
        cmp     bl, 0Bh             ; 灰度级监视器
        je      vid_is_gray         ; 如果是，则跳转
        cmp     bp, 5               ; VGA +?

        mov     ax, 40h             ; BIOS数据区
        mov     es, ax
        test    byte ptr es:[89h], 2  ; 灰度级?
        jz      vid_string          ; 无灰度级
vid_is_gray:
        mov     ch, 2               ; 将ch设置为灰度级

vid_string:
        xor     cl, cl              ; 假定为供应商串
        cmp     bp, 6               ; SVGA ?
        je      vid_skp1            ; 如果是，则跳转
        cmp     bp, 8               ; VESA XGA ?
        jne     vid_mono_chk        ; 如果不是，则跳转
```

```
vid_skp1:
        mov     di, offset infobuf      ; 获取SVGA缓冲区信息
        mov     si, cs:[di+6]           ; 获取供应商串的段
        mov     ax, cs:[di+8]           ;    和偏移量
        mov     es, ax                  ; es:bx 指向串
        inc     cl                      ; 供应商串有效

; 最后检查是否设置了视频模式2,
; 模式2表示必须使用单色属性
; （如果我们还没有检测单色操作）

vid_mono_chk:
        cmp     ch, 0                   ; 设置为彩色?
        jne     vid_exit                ; 若不是，则跳转
        mov     ah, 0Fh
        int     10h                     ; 获取视频模式
        and     al, 7Dh
        cmp     al, 0                   ; 视频模式是0还是2?
        jne     vid_exit                ; 如果不是0，则跳转
        mov     ch, 1                   ; 使用单色属性


vid_exit:
        mov     ax, bp                  ; 在al中返回适配器类型
        pop     bp
        pop     di
        pop     dx
        pop     bx
        ret
video_type endp
```

端口归纳

下面的端口列表用来控制所有主要的视频适配器类型

端口	类型	功能	适配器
3B4h	输出*	视频控制器寄存器选择	MDA、HGA、EGA+

3B5h	I/O	视频控制器数据	MDA、HGA、EGA+
3B8h	输出	视频适配器控制	MDA、HGA
3BAh	输入	视频适配器状态	MDA、HGA
3BAh	输入	视频状态寄存器 1（单色）	EGA+
3BAh	输出	视频特性控制（单色）	EGA+
3BFh	输出	Hercules 控制寄存器（单色）	HGA
3C0h	输出	视频属性控制器寄存器	EGA+
3C1h	输入	视频属性控制器寄存器	VGA+
3C2h	输出	视频多种输出寄存器	EGA+
3C2h	输入	视频状态寄存器 0	EGA+
3C3h	I/O	开放视频适配器	VGA
3C4h	输出*	选择视频序列发生器视频器	EGA+
3C5h	输出*	视频序列发生器数据	EGA+
3C6h	I/O	视频像素屏蔽寄存器	VGA+
3C7h	I/O	视频数模转换器	VGA+
3C8h	I/O	视频数模转换器	VGA+
3C9h	I/O	视频数模转换器	VGA+
3CAh	输入	视频特性控制寄存器 1	VGA+
3CCh	输出*	视频图形 1 位置	EGA+
3CDh	输出*	视频图形 2 位置	EGA+
3CEh	输出*	视频图形控制器寄存器选择	EGA+
3CFh	输出*	视频图形控制器数	EGA+
3D4h	输出*	视频控制器寄存器选择	CGA、EGA+
3D5h	输出*	视频控制器数据	CGA、EGA+
3D8h	输出	视频适配器模式控制	CGA
3D9h	输出	视频适配器颜色选择	CGA
3DAh	输入	视频适配器状态	CGA
3DAh	输入	视频状态寄存器 1（彩色）	EGA+
3DAh	输出	视频特性控制（彩色）	EGA+
3DBh	输出	视频适配器清除光笔锁存器	CGA
3DCh	输出	视频适配器预设光笔锁存器	CGA
xC80h	I/O	EISA 视频 ID	VESA XGA
xC81h	I/O	EISA 视频 ID	VESA XGA
xC82h	I/O	EISA 视频 ID	VESA XGA
xC83h	I/O	EISA 视频 ID	VESA XGA
xC84h	I/O	EISA 视频扩展板控制	VESA XGA
xC85h	I/O	EISA 设置控制	VESA XGA
xC88h	I/O	EISA 视频可编程选项选择 0	VESA XGA
xC89h	I/O	EISA 视频可编程选项选择 1	VESA XGA
xC8Ah	I/O	EISA 视频可编程选项选择 2	VESA XGA

xC8Bh	I/O	EISA 视频可编程选项选择 3	VESA XGA
xC8Ch	I/O	EISA 视频可编程选项选择 4	VESA XGA
xC8Dh	I/O	EISA 视频可编程选项选择 5	VESA XGA
xC8Eh	I/O	EISA 视频可编程选项选择 6	VESA XGA
xC8Fh	I/O	EISA 视频可编程选项选择 7	VESA XGA
21z0h	I/O	视频操作模式寄存器	XGA+
21z1h	I/O	视频操作开孔控制	XGA+
21z2h	I/O	视频未使用或未知功能	XGA+
21z3h	I/ O	视频未使用或未知功能	XGA+
21z4h	I/O	视频中断开放	XGA+
21z5h	I/O	视频中断状态	XGA+
21z6h	I/O	视频虚拟内存控制	XGA+
21z7h	I/O	视频虚拟内存中断状态	XGA+
21z8h	I/O	视频开孔索引	XGA+
21z9h	I/O	视频内存访问模式	XGA+
21zAh	I/O	视频数据索引	XGA+
21zBh	I/O	视频数据，字节宽	XGA+
21zCh	I/O	视频数据，字和双字宽	XGA+
21zDh	I/O	视频数据	XGA+
21zEh	I/O	视频数据	XGA+
21zFh	I/O	视频操数据	XGA+
46E8h	I/O	视频适配器开放	VGA

注：*在 EGA 上这些端口是只写的，在其他上可读了可写
x 代表 XGA 适配器安装时的 EISA 插槽号
z 代表中哪个 XGA，如果系统有多个 XGA 适配器

未公开的视频 I/O 端口的细节

端口	类型	描述	适配器
3C3h	I/O	开放视频适配器	VGA、SVGA

这个端口控制是否可访问某个视频卡。如果被禁止，那么除本端口外所有的 I/O 端口（与视频有关的）都将被忽略，从 A000 到 BFFF 的视频内存从总线上断开。视频卡保持当前的状态，显示保持不变，这个开放/禁止位主要用于在主板 VGA 和适配卡 VGA 之间进行切换。使用这个端口的另外一位也可能控制 MCA 的 POS 寄存器。

这个端口只可以在 VGA 类型的主板上执行。可以使用 VGA 适配卡上的端口 46E8h 完成相同的功能。

I/O（位 0~7）

位	7 r/w=x	未用，或由供应商定义
	6 r/w=x	未用，或由供应商定义
	5 r/w=x	未用，或由供应商定义
	4 r/w=1	为 POS 寄存器设置（仅 MCA 系统）
	3 r/w=0	禁止视频 I/O 和视频缓冲区
	1	开放视频 I/O 和视频缓冲区
	2 r/w=x	未用，或由供应商定义
	1 r/w=x	未用，或由供应商定义
	0 r/w=x	未用，或由供应商定义

端口	类型	描述	适配器
3C6h	I/O	视频 DAC 像素屏蔽器	VGA+

这个端口限制了屏幕上一次可显示的颜色或阴影的数目。BIOS 用它完成各种指定的模式功能。例如，图形模式 5 就限于四种颜色。

视频适配器使用一个数模转抵换器（DAC）来获得三个 6 位颜色值并将它们转化成模拟电压，作为送到显示器的红色、绿色和蓝色信号。DAC 有 256 个这样的 6 位颜色三元组。当由 VGA 硬件访问时，每个这样的颜色三元组将显示一个合成的颜色。

像素地址屏幕器会“隔断”到 DAC 的指定的地址线，这样，如果硬件指定了一个较高的颜色地址，DAC 将指向较低的地址。屏幕器中的每一位分别控制 256 色寄存器中的 8 个地址位之一。例如，在本寄存器中装入值 0Fh 会将颜色地址限制到前 16 个颜色三元组。硬件对三元组 2 的访问等同于访问 12h、22h、32h 等等。

目前还不太清楚是否所有的供应商都支持读取这个寄存器。视频 BIOS 从来没有读取过这个寄存器。

I/O（位 0~7）

位	7 w=0	忽略地址线 7（当作 0）
	6 w=0	忽略地址线 6（当作 0）
	5 w=0	忽略地址线 5（当作 0）
	4 w=0	忽略地址线 4（当作 0）
	3 w=0	忽略地址线 3（当作 0）
	2 w=0	忽略地址线 2（当作 0）
	1 w=0	忽略地址线 1（当作 0）
	0 w=0	忽略地址线 0（当作 0）

端口	类型	描述	适配器
3C7h	I/O	视频 DAC 颜色寄存器地址	VGA+

这个端口访问 DAC 内部的 18 位颜色寄存器。在向/从端口 3C9h 写/读之前先向这个寄

存器中写入地址0到255。读这个端口会返回当前的地址。

I/O（位0~7）颜色寄存器地址

端口	类型	描述	适配器
3C8h	I/O	视频DAC颜色寄存器初始化	VGA+

这个端口初始化DAC内部的18位颜色寄存器。向这个寄存器写入地址0到255，来指定初始化哪个颜色寄存器。简单读可以获取当前是哪个颜色寄存器。

I/O（位0~7）要初始化的颜色寄存器的地址

端口	类型	描述	适配器
3C9h	I/O	视频DAC颜色寄存器数据	VGA+

这个端口读或写一个颜色寄存器。必须已经用端口3C7h设置了地址。由于每个颜色寄存器由三个6位组组成，因此有必要读或写一次来访问所有的18位。访问的第一个字节是红，然后是绿，最后是蓝。每个字节含有颜色值的低6位，高2位被忽略。将来带有不同硬件的VGA适配器可能会使用每个寄存器的全部8位，来提供24位的颜色功能。

在访问了一个颜色寄存器的三个字节之后，地址值会自动递增。可以访问下一个颜色寄存器而不用向端口3C7h发送一个新地址。

I/O（位0~7）颜色寄存器数据（3字节，红/绿/蓝）

端口	类型	描述	适配器
46E8h	I/O	开放视频适配卡	VGA、SVGA

这个端口控制某个视频卡是否可被访问。如果被禁止，那么除本端口外所有的I/O端口（与视频有关的）都被忽略，从A000到BFFF的视频内存从总线上断开。视频卡保持当前的状态，显示保持不变，该开放/禁止位主要用于在主板VGA和适配卡VGA之间进行切换。使用这个端口的另外一位也可能控制MCA的POS寄存器。

这个端口只可以在VGA类型的适配卡上执行，端口3C3h用于主板VGA上完成相同的功能。

I/O（位0~7）

位		
	7 r/w=x	未使用或由供应商定义
	6 r/w=x	未作用或由供应商定义
	5 r/w=x	未作用或由供应商定义
	4 r/w=1	供POS寄存器设置使用
	3 r/w=0	禁止视频I/O端口和视频缓冲区
	1	启用视频I/O端口和视频缓冲区
	2 r/w=x	未使用或由供应商定义
	1 r/w=x	未使用或由供应商定义
	0 r/w=x	未使用或由供应商定义

第 10 章

软盘系统

相对于其他大多数的子系统，过去对软盘 BIOS 所做的说明要好得多。从第一台 PC 开始，软盘系统的基本控制器接口就基本保持未变。当然，AT 还加入了一些新的 BIOS 服务，最新的系统还支持 2.88M 的软盘。

我对软盘 BIOS 服务和错误代码进行了扩展，以提供额外的信息和帮助阐明那些对于可靠的应用 BIOS 和硬件服务很有必要的问题。另外，我还编写了一个软盘驱动器检测程序来标识所有的软盘驱动器，包括从早期的 320K 软驱到最近的 2.88M 软驱。本章后面附录了完整的源代码。

在直接访问软盘控制器时，我讨论了所有通用的端口，详细而深入地阐释了每个端口的相关细节。这些端口也指出了一些特殊的值，在对目前系统所支持的每种软盘控制器进行配置和操作时，会使用到这些值。目前很少有文档对软盘控制器做出了说明，并且对于程序员来说，它们的组织结构也比较差劲。我提供了所有的控制命令和值，提供的形式也比较易于理解和使用。

警告部分指出了使用软盘 BIOS 服务和控制器时许多没有被理解甚至被错误理解的一些问题。

简　介

软盘的访问要通过适配卡上的硬件和内置于系统主 BIOS 内的 BIOS 软件。该组合提供了简单方便的格式化、读和写软盘操作，从 AT 开始，BIOS 所支持的软盘驱动器的数目从四个减少到两个以内。访问两个以上的软驱需要专门的适配器硬件和驱动程序，比如 DOS 中的 DRIVER.SYS。包括 DOS 在内的大多数操作系统并未对软盘驱动器的数目和类型作出限制。

图 10-1 显示了软盘系统。其中软盘控制器是系统的核心，它连接着系统的每个软驱。所有的通信都通过软盘控制在系统和软驱之间流通。系统通过 I/O 命令访问控制器，并返回状态。

某些命令在完成请求的操作之后会中断处理器，在适配器和系统之间之间读与写扇区的传送通过 DMA 传送。系统使用 DMA 通道 2 完成这些传送。如果使用 BIOS 向软盘读和

写数据。那么 DMA 操作对用户是透明的。第 18 章详细阐释了 DMA 过程。

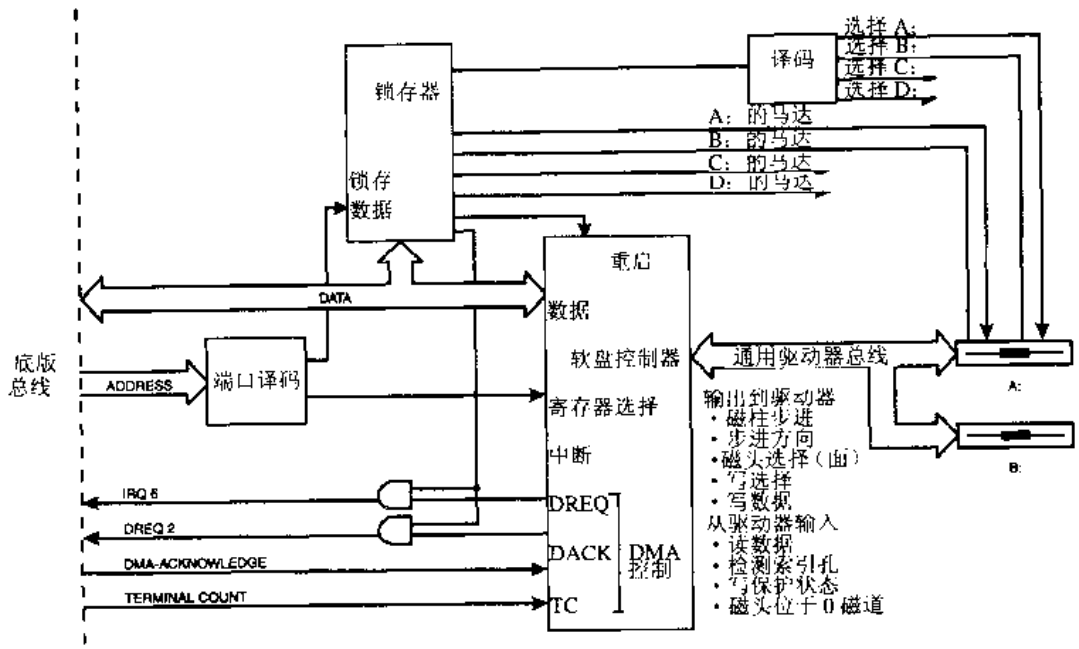


图 10-1 软盘系统

本章从头到尾都解释了与软盘结构相关的通用术语。每个软盘分为磁柱、磁道和扇区，如图 10-2 所示。最小的组成部分是扇区，通常有 512 个字节的数据。多个扇区组成一个磁道。例如，一个 1.44MB 的软盘每磁道有 18 个扇区。其他媒质类型每个磁道有 9 到 36 个扇区，这取决于在每英寸媒质上可以可靠保存的位数。两个磁道，软盘的一面各一个，定义为一个磁柱。软盘有 40 或 80 个磁柱，这取决于媒质类型。

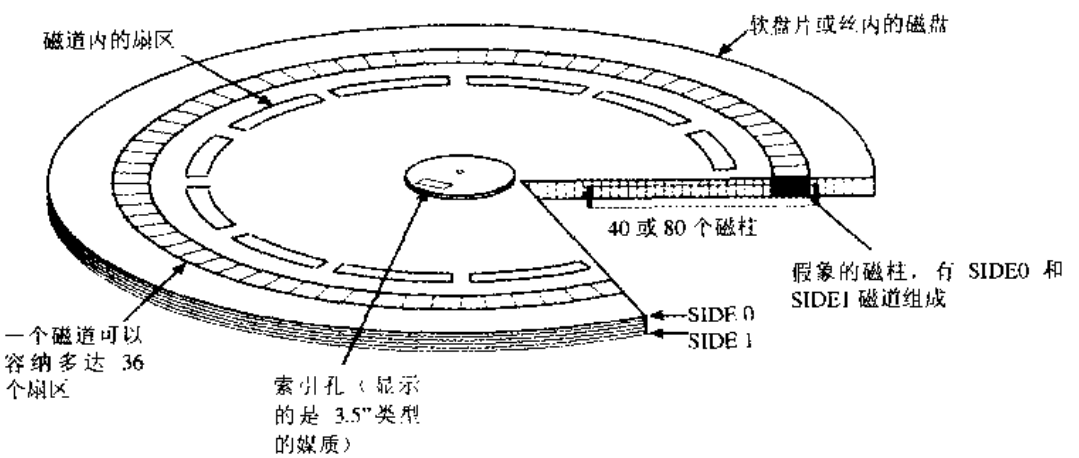


图 10-2 软盘媒质结构

软盘驱动器媒质表

表 10-1 显示了如何由 512 个扇区组成软盘的总容量。这个表也列出了所使用的传输率，单位是位每秒。它是数据在驱动器和控制器之间传输的速率。传输速率取决于软盘的容量、驱动器类型以及软盘转动的速率，单位是 RPM（转每分）。软盘转得越快，每线性英寸的媒质上被选中的数据就越多，传输速率总是固定的，软件不能对它进行定义。

表 10-1 媒质表

驱动器	软　盘	每面的磁道数	每磁道的扇区数	RPM	传输速率
360K，5.25"	360K	40	9	300	250Kbps
1.2M，5.25"	360K	40	9	360	300Kbps
1.2M，5.25"	1.2M	80	15	360	500Kbps
720K，3.5"	720K	80	9	360	250Kbps
1.44M，3.5"	720K	80	9	360	250Kbps
1.44M，3.5"	1.44M	80	18	360	500Kbps
2.88M，3.5"	720K	80	9	360	250Kbps
2.88M，3.5"	1.44M	80	18	360	500Kbps
2.88M，3.5'	2.88M	80	36	360	1Mbps

软盘数据格式化

软盘的每个磁道由间隙（gap）、同步脉冲以及数据组成，比如地址标签、扇区号、循环冗余校验（CRC），以及真正的扇区数据。明白各种错误条件，比如“丢失地址标签”，有助于理解有关磁道的各种信息。丢失地址标签错误表明地址字节值不是控制器所期望的。在数据传送到系统之前，控制器会剥去象地址标签这样的信息。

扇区号由四个字节组成，包括磁柱、磁头、扇区、以及扇区大小代码。这些由用户的格式化命令控制，通常等于物理磁柱、头扇区以及实际扇区大小的代码。这样做是非必须的。早期的防拷贝方案会在扇区号中写入一个古怪的值来帮助识别源软盘。一个操作系统，比如 DOS，总在格式化扇区号时让磁柱、磁头、扇区、以及扇区代码大小等于物理位置。

图 10-3 列出了目前 PC 上所使用的所有软盘类型的数据层次。

磁道字节

- 索引脉冲间隙区，80 个字节的 4Eh
- 磁道同步脉冲，12 个字节的 0h
- 磁道索引地址标签，3 个字节的 C2h，紧接着 1 字节的 FCh
- 磁道间隙区，50 个字节的 4Eh
- 扇区字节（每个扇区重复）
  - 扇区同步脉冲，12 个字的 0h
  - 索引地址标签，3 个字节的 A1h，紧接着 1 字节的 FEh
  - 磁柱字节（通常是 0~79，与媒质有关）
  - 磁头字节（通常是 0 或 1）
  - 扇区号字节（通常是 1~36，与媒质有关）
  - 扇区大小代码（通常是 2，表示是 512 字节的扇区）
  - CRC 字节，涵盖了前四个字节
  - 扇区间隙，在 360K 到 1.44M 媒质上是 22 个字节的 4Eh
    在 2.88M 的媒质上是 41 个字节的 4Eh
  - 扇区同步脉冲，12 字节的 0
  - 数据地址标签，3 个字节 A1h，接着 1 个字节的 FBh 或 F8h，
    （值表明是正常的数据或被删除的数据）
    - 数据，由扇区大小码指定的字节数
    - 扇区数据的 CRC 字节
  - 扇区间隙，格式化命令设置的大小
  - 磁道的额外扇区（总区可达 36，与媒质有关）
- 磁道间隙尾

图 10-3 一个软盘磁道的内容

软盘参数表

软驱参数表包含了有关驱动器的信息。BIOS 使用这个表来为软盘控制器编程和指定软驱的控制定时，指向这个参数表的指针保存在中断 1Eh 的地址 0:78h 中，留心某些技术参考书并不理解参数表是如何工作的，给出了令人误解或不正确的信息。

软盘参数表是一个重要的概念，如果系统上只有一个驱动器类型。正像早期的 PC 和 XT 系统一样。参数表在早期的 PS/2 上工作得也很好，在这里要从两种类型的驱动器中选择一个，比如 3.5" 1.44M。软盘参数表用于完整地定义驱动器的参数。

当 BIOS 支持两个或两个以上的不同类型的驱动器时，正如许多系统那样，会出现一个严重的问题。一个系统可能有两个驱动器，选自下面现有的 5 个软驱：360K、

1.2M、720K、1.44M 或 2.88M。没有方法使软盘参数表支持多个驱动器。结果是，大多数系统 BIOS 制造商编写 BIOS 软盘服务处理程序来忽略软盘参数表中的大多数字节。

BIOS 的多型软驱支持的处理方式因 BIOS 制造商不同而有所不同。BIOS POST 在初始化后指向的软盘参数表通常只对一个驱动器有效，即使忽略了参数表！

当 BIOS 软盘服务小程序使用来自软盘参数的值时，BIOS 通常只是将值直接传给软盘控制器而不做任何检查。BIOS 假定表中的值有效。表 10-2 列出了软盘参数表的内容。

表 10-2 软盘参数表的内容

偏移量	描述		
0	软盘控制器（参看端口 3F5h，专用命令，字节 2 了解完整细节）		
	位	7=x	步速率（计时随数据率变化）
		6=x	常见的值包括：
		5=x	Ah=3ms，在一个 2.88M 驱动器上
		4=x	Ah=6ms，在一个 1.44M 驱动器上
			Ah=12ms，在一个 720K 驱动器上
			Dh=3ms，在一个 1.2M 驱动器上
			Dh=6ms，在一个 720K 驱动器上
			Dh=6ms，在一个 360K 驱动器上
		3=x	磁头下载时间（计时随数据率变化）
		2=x	常见的值包括：
		1=x	1=16ms，在一个 1.44M 驱动器上（仅 PC/1 和 PS/2）
		0=x	1=27ms，在一个 720K 驱动器上（仅 PC/1 和 PS/2）
			Fh=120ms，在一个 2.88K 驱动器上
			Fh=240ms，在一个 1.44M 和 1.2M 驱动器上
			Fh=6ms，在一个 720K 和 360K 驱动器上
1	软盘控制器（参看端 3F5h，专用命令，字节 3 了解完整细节）		
	位	7=x	磁头载入时间（计时随数据率变化）
		6=x	常见的值是 0000001，代表载入时间是：
		5=x	2ms，在一个 2.88M 的驱动器上
		4=x	4ms，在一个 1.44M 和 1.2M 驱动器上
		3=x	8ms，在 720K 和 360K 驱动器上

续表

偏移量	描述		
		2=x	
		1=x	
		0=0	软盘操作使用 DMA（普通情况）
		1	没有使用 DMA
2	驱动器休眠马达关闭延时		
	初始化递减的延时值，来控制马达的休眠时间。中断 8 的时钟处理程序每 54ms 减		
	少它一次，常见的值是 25h，用来设置两秒的延时。		
3	每扇区的字节		
	0=128 字节（或者支持在偏移量 6 处指定数据长度字节。）		
	1=256 字节		
	2=512 字节（所有 PC 软盘的标准字节数）		
	3=1024 字节		
	4=2048 字节（并非所有的媒质都支持）		
	5=4096 字节（并非所有的媒质都支持）		
	6=8192 字节（并非所有的媒质都支持）		
	7=16384（并非所有的媒质都支持也并非所有的控制器都支持）		
	8~FFh		
4	每磁道的扇区数（常见值如下所示）		
	9=9 扇区，360K 和 720K 的软盘		
	0Fh=15 扇区，1.2M 软盘		
	12h=18 扇区，1.44M 软盘		
	24h=36 扇区，2.88M 软盘		
5	扇区之间的间隙长度（常见值如下所示）		
	1Bh=供 1.2M、1.44M 和 2.88M 软盘		
	2Ah=供 360K 和 720K 软盘		
6	数据长度		
	如果在偏移量 3 处的每扇区字节数的值为 0，则这个字节指定了扇区的大小，长度		
	可以从 1 到 255 字节。所有的 PC 驱动器媒质使用 512 字节的扇区，所以这个字节		
	并重新，常置为 FFh。		
7	格式化间隙长度		

续表

偏移量	描　　述
	格式化时扇区之间间隙的大小（常见值如下所示）
	50h=供 360K、720K 和 2.88M 软盘
	54h=供 1.2M 软盘
	6Ch=供 1.44M 软盘
8	格式化填充字节
	设置扇区数据来在格式化操作时填充字节，标准的填充字节值是 F6h。
9	磁头定位时间
	BIOS 软件延时，单位毫秒，在找磁头后的等待时间以定位到正确的磁道上，所有
	器使用缺省的延时 15ms（0Fh）。
A	等待马达开启时间
	在写操作期间，该值指定了首次开启了软盘马达后所等待的延时，单位是八分之一秒。
	大多数系统使用缺省值 8 来设置一秒的延时。某些 PS/2 系统假定 3.5"驱动器转动更快，
	因此只等待半秒的缺省值。读期间没有预设的延时，因为控制器仅仅只是开始读，直
	到数据稳定。

BIOS 初始化

PC 和 XT 系统 BIOS 只理解 360K 类型的驱动器，主板上的开关告诉 BIOS 配置了多少个驱动器。一旦读取了开关，控制就会被重设。

AT+系统的 POST 从 CMOS RAM 字节 10h 中读取信息，来理解有哪些软盘驱动器以及它们的驱动器类型。POST 然后对控制器触发软件重启。在 CMOS 信息帮助下，POST 使用启动盘合适的数据率对控制器进行编程。参看第 15 章以更多地了解 CMOS 字节。

对于所有的系统，在控制器重设并准备好之后，马达就打开了。大约延时 1 秒后，POST 发出查找命令来将磁头移到软盘的一个内部磁柱上。如果成功的话，该驱动器就可作为潜在的引导驱动器。然后关闭马达只到 POST 准备从软驱引导。

作为初始化最后步骤的一部分，指向系统 BIOS ROM 中的软盘参数表的远指针被装入到 0:78h 处的中断向量表中。POST 检查是否存在硬盘，如果存在，BIOS 将软盘中断向量 13h 拷贝到中断 40h。然后，BIOS 用一个指向硬盘服务程序的新指针替代旧有的中断 13h。为软驱传递给硬盘中断 13h 的功能，现在被重新指向了起初软驱服务程序，在中断 40h。

程序应总是使用中断 13h 来提供软盘服务，如果直接使用中断 40h，那么任何挂起中断 13h 的 TSR 和设备驱动程序都会被跳过，可能会造成系统冲突。

软盘 BIOS

通过中断 13h 访问软盘服务。大多数功能要求用 DL 中的一个数来指定选择哪个驱动器，在 AT 和所有后来的系统上，BIOS 只支持软驱 0（在 DOS 中为 A:）和软驱 1（在 DOS 为 B:）。早期的 PC 和 XT 都支持多达四个的软驱。注意，所有的软盘适配器使用的磁盘控制器可以支持多达四个的软驱，但是大多数和较新的适配器没有为第三个和第四个驱动器提供连接。少数支持四个软驱的适配卡必须提供一个专门的 BIOS ROM 来支持第三个和第四个软驱。

服务结束时，软盘 BIOS 会在 AH 寄存器中返回一个状态字节。该状态字节表示操作成功还是错误代码。该返回状态字节保存在地址为 40:41h 处的 BIOS 数据区中。表 10-3 详细描述了这些状态返回值。

表 10-3 AH 中的状态返回码

十六进制代码	描　　述
0	操作成功，无错误发生。
1	传递无效值或不支持功能。
2	丢失地址标签——没有扇区地址标签匹配该请求。
3	软盘写保护。
4	没有找到所请求的扇区，或者请求的扇区超过了软盘的最后一个磁柱。
6	软盘改变线激活——除了 360K 的驱动器，所有的驱动器提供了一条硬件线，在任何移动，改变和丢失软盘时来改变控制器。
8	DMA 超时—DMA 从/向控制器传递数据不够快，以至于控制器发生超时现象。
9	数据越界错误——DMA 操作必须完全位于一个 64K 的页面内。该代码意味着 DMA 操作试图越过一个 64K 的物理边界，比如地址是 FFFFh、1FFFFh、2FFFFh 之类。
C	没有找到媒质类型。
10	读时发生了 CRC 错误——记录在软盘扇区上的 CRC 与从扇区数据计算得到的 CRC 不匹配。数据极有可能出错。
20	软盘控制器或驱动器有问题，有两种情形： 1）控制器内命令的异常结束； 2）返回的驱动器状态字节含有无效值，可能是控制器出了问题。
40	查找操作失败——移动到指定软盘磁柱的努力失败。
80h	超时——软驱无响应。

软盘 BIOS 服务

某些技术参考书指出，一些中断 13h 的功能，像 AH=6、7 和 9，被“保留”。它们要么被共享这个中断的硬盘所使用，要么现在未被使用，并且据我所知，从来没有被使用过。

软盘中断 13h 提供了下列十一个服务，参看第 11 章以了解与硬盘相关的中断 13h 服务。

功能	描述	平台
ah=0	软盘控制器重启	所有
ah=1	读软盘状态	所有
ah=2	读软盘扇区	所有
ah=3	写软盘扇区	所有
ah=4	检查软盘扇区	所有
ah=5	格式化软盘磁道	所有
ah=8	读软驱磁道	AT+
ah=15h	读软驱参数	AT+
ah=16h	获取软盘改变线状态	AT+
ah=17h	为格式化设置软盘类型（老式）	AT+
ah=18h	为格式化设置软盘类型（新式）	AT+

中断	功能	描述	平台
13h	0	软盘控制器重启	所有

命令软盘控制器重启，命令控制器重新设定的驱动器。

调用：　ah=0
　　dl=驱动器，0~3
返回：　ah=返回状态（参看表 10-3）
　　进位=0，如果成功 1，如果出错

中断	功能	描述	平台
13h	1	读软盘状态	所有

在从每次软盘服务返回时，该返回状态值保存在 BIOS 数据区的 40:41h 处。这个功能从 40:41h 返回目前保存的值，该值代表了上次操作的状态。与其他功能不同，不需要使用软驱号。

调用：　ah=1
返回：　ah=返回状态（参看表 10-3）
　　进位=0，如果成功 1，如果出错

中断	功能	描述	平台
13h	2	读软盘扇区	所有

由于在读请求时软盘马达可能是关闭的，某些老式的 BIOS 会在每次试图读时返回一个错误，直到马达达到正确的速度。在确认一个错误之前必须至少试图读三次。大多数新式的 BIOS 会等待直到马达达到合适的速度，然后执行操作。

调用：
ah=2
al=要读的扇区数目，1~36*
ch=磁道号，0~79*
cl=扇区号，1~36*
注：*参看媒质表（表 10-1）
dh=磁头号，0~1
dl=驱动器号，0~3
es:bx=指针，指向放置从软盘读取的信息的位置

返回：
ah=返回状态（参看表 10-3）
al=实际读的扇区数目（参看警告部分）
进位=0，如果成功 1，如果出错

中断	功能	描述	平台
13h	3	写软盘扇区	所有

从所指向的位置开始写一系列的软盘扇区。由 ES:BX 指定读取数据的 RAM 地址。缓冲区不得越过 64K 线性地址边界。参看警告部分以了解更多细节。

由于在写请求时软盘马达可能是关闭的，某些老式的 BIOS 会在每次试图写时返回一个错误，直到马达达到正确的速度。在确认一个错误之前必须至少试图写三次。大多数新式的 BIOS 会等待直到马达达到合适的速度，然后执行操作。

调用：
ah=3
al=要写的扇区数目，1~36*
ch=磁道号，0~79*
cl=扇区号，1~36*
注：*参看媒质表（表 10-1）
dh=磁头号，0~1
dl=驱动器号，0~3
es:bx=指针，指向放要写到软盘的信息的来源地址

返回：
ah=返回状态（参看表 10-3）
al=实际写的扇区数目（参看警告部分）
进位=0，如果成功 1，如果出错

中断	功能	描述	平台
13h	4	检查软盘扇区	所有

从指定的位置开始检查一系列的软盘扇区。该命令实际上并不逐字节地检查数据是否和缓冲区的内容相匹配。相反地，控制器只是试着读取指定的内容，然后计算 CRC。如果扇区定位了、读取了并且匹配了 CRC，那么数据被认为是正确的。

大多数技术参考书指出检查操作需要 ES:BX 指向的缓冲区。在基于 AT、MCA 或者 EISA 的 BIOS 上并不需要用到它。某些老式的 BIOS 可能会用到该缓冲区，但是并不核对实际的内容。如果提供了一个缓冲区，那么该缓冲区不得越过 64K 线性地址边界。

这个功能通常用来确定一个软盘是否在软驱内。要做到这一点，必须先使用功能 0 重启驱动器，然后进行检查。为了加快运行结果，只须检查一个扇区，比如磁道 0、扇区 1、磁头 0。

由于在检查请求时软盘马达可能是关闭的，某些老式的 BIOS 会在每次试图检查时返回一个错误，直到马达达到正确的速度。在确认一个错误之前必须至少试图检查三次。大多数新式的 BIOS 会等待直到马达达到合适的速度，然后执行操作。

调用：
ah=4
al=要检查的扇区数目，1~36*
ch=磁道号，0~79*
cl=扇区号，1~36*
注：*参看媒质表（表 10-1）
dh=磁头号，0~1
dl=驱动器号，0~3
es:bx=未使用缓冲区（参看上文所述）

返回：
ah=返回状态（参看表 10-3）
al=实际检查的扇区数目（参看警告部分）
进位=0，如果成功 1，如果出错

中断	功能	描述	平台
13h	5	格式软盘磁道	所有

格式化软驱上的一个磁道。每个扇区的数据包括控制器的额外信息，同格式化命令一起被写入。这些信息有些可由用户指定，比如磁道号、磁头号、扇区号和扇区大小。另外，控制器还写入一些间隙、同步脉冲、CRC 字节以及用一个“填充”字节填充扇区的数据区。所有的 PC BIOS 使用 F6h 作为填充字节，但是可以通过修改软盘参数来改变为其他的值。

在 ES:BX 指向的表中提供了用户的信息。该地址表为每一个要格式化的扇区保留了一个入口。要格式化一个 18 扇区的磁道需要 18 个入口，每个有四字节长。地址表的内容如下：

字节	描述
0	磁道号 0~FFh
1	磁头号 0~FFh
2	扇区号 1~FFh
3	扇区大小 0=128 字节 1=256 字节 2=512 字节（所有 PC 软盘上所使用的标准大小） 3=1024 字节 4=2048 字节(并非所有的系统都支持) 5=4096 字节(并非所有的系统都支持) 6=8192 字节(并非所有的系统和媒质都支持) 6=16384 字节(并非所有的系统和媒质都支持) 8~FFh 无效

一个 DOS 兼容的格式化过程会使用地址表磁道号、磁头号和扇区号来匹配实际的磁道号、磁头号和扇区号，并且认为扇区大小是 512 字节。某些防拷贝方案改变这种一对一的映射或者使用非标准的扇区大小。例如，完全可以在相同的磁道上格式不同的扇区大小、丢失扇区、甚至改变扇区编号的次序。

一种防拷贝方案是在磁盘的一个磁道上以相反的次序写扇区。通过对普通磁道和“反”磁道的读操作进行计时，并比较结果，程序可以确定它是否是一个原始的防拷贝软盘。操作系统的磁盘拷贝不明白这种古怪的排序方式。也不会传送“反”扇区编号。所以很容易检测出这是一张复制品。

许多软驱支持多种格式。3.5"、2.88M 的驱动器支持 2.88M、1.44M 和 720K 软盘。在格式化之前，必须先确定媒质类型。功能 18h，设置格式化时的媒质类型，就可以实现这一点。如果格式存在问题，功能 18h 会返回一个错误信息。老式的系统不支持 3.5"软驱，也不支持功能 18h，必须使用功能 17h 来设置媒质类型。

对于非常老的系统，像早期的 PC 和大多数 XT 类型，BIOS 不支持多媒质类型驱动器。在 360K 5.25"驱动器上格式化软盘时不需要采取额外的步骤。

调用：

ah=5
al=要格式化的扇区数目，1~36*（通常为媒质设置每条磁道上的所有扇区）
ch=磁道号，0~79*
*参看媒质表 10-1
dh=磁头号，0 或 1
dl=驱动器号，0~3
es:bx=指针，指向地址表（参看上文所述）

返回：　　ah=返回状态（参看表 10-3）
　　进位=0，如果成功 1，如果出错

中断	功能	描述	平台
13h	8	读软驱参数	AT+

获取指定驱动器的驱动器参数指针和数据。驱动器类型读自 DMOS 地址 10h，并被用来返回其他信息。记住，驱动器的媒质并不限于这些数据，它们只是指定驱动器的最大功能而已。参看代码例 11-2，第 11 章中的程序 DISKTEST，了解检测驱动器中当前媒质类型的一种方法。

调用：　　ah=8
　　dl=驱动器号，0 或 1
返回：　　ah=返回状态（参看表 10-3）
　　进位=0，如果成功　　1，如果出错
　　如果成功，且已知驱动器类型
　　al=0
　　bh=0
　　bl=读自 CMOS 内存的驱动器类型值
　　0=没有 CMOS、CMOS 检查无效，或电池用完
　　1=360K 5.25"
　　2=1.2M 5.25"
　　3=720K 3.5"
　　4=1.44M 3.5"（如果支持）
　　5=2.88M 3.5"（如果支持）
　　6~FFh=未定义（留作将来的驱动器用）
　　ch=最大磁道号（值见下）
　　cl=最大扇区号（值见下）
　　dh=最大磁头号（总为 1）
　　dl=安装的软驱数目（0~2，不包括 BIOS 不支持的驱动器）
　　es:di=指针，指向 11 个字节的软驱参数表（参看表 10-2）
　　如果成功，但请求的驱动器尚未安装
　　ax=bx=cx=dh=es=0
　　如果成功，但是驱动器号大于 1，或者因 CMOS 信息不可用而造成驱动器类型未知：
　　ax=bx=cx=dh=es=0
　　al=安装的软驱数目（0~2，不包括 BIOS 不支持的驱动器）

标准驱动器的最大磁道号（从零开始）和扇区号（从零开始）分别是：

驱动器类型	最大磁道号	最大扇区号	CX返回值
360K，5.25"	39	9	2709h
1.2M，5.25"	79	15	4F0Fh
720K，3.5"	79	15	4F0Fh
1.44M，3.5"	79	18	4F12h
2.88M，3.5"	79	36	4F24h

中断	功能	描述	平台
13h	15h	读软驱类型	AT+

检查驱动器目前是否可用，是否支持改变线。参看中断 13h 功能 16h 以了解更多的有关改变线的信息。360K 的 40 磁道的驱动器没有改变线。有 80 磁道的驱动器，例如 1.2M、1.44M 和 2.88M，都有改变线。

调用：ah=15h
dl=驱动器号，0 或 1

返回：如果成功
ah=0 没有驱动器
1 有驱动器，不支持改变线
2 有驱动器，支持改变线
进位=0

如果失败
ah=1，不支持这个功能（例如，PC/XT）
进位=1

中断	功能	描述	平台
13h	16h	获取软盘改变线的状态	AT+

1.2M、1.44M 和 2.88M 驱动器检测软盘时可被移走或替换。如果没有软盘或者插入的新盘尚未检查媒质类型，那么就会激活改变线。一旦确定了媒质类型，改变线就不再处于激活状态。

如果未知软驱类型，必须先调用功能 15h 来确定驱动器是否支持改变线。

调用：ah=16h
dl=驱动器号，0~3

返回：如果成功
ah=0 改变线未被激活
1 无效的驱动器号（只支持 0 或 1）
6 改变线被激活或者不被支持

80h　　不存在驱动器
进位=0
如果失败
ah=1，不支持这个功能（例如，PC/XT）
进位=1

中断	功能	描述	平台
13h	17h	为格式化设置软盘类型（老式）	AT+

这个功能指定了某些类型的软驱上的媒质类型，应在触发格式化功能之前设置软盘类型，但是如果已经设置了正确的媒质类型，或者接受缺省的类型，那么这一点就没有必要了。它仅适用于 360K、1.2M 和 720K 的驱动器，不用于 1.44M 和 2.88M 的驱动器。

这个功能已被更灵活的功能 18h 所替代，后者支持较新的驱动器类型。只要可行，就应该用功能 18h 替代本功能。

调用：　ah=17h
al=1　在 360K 的驱动器中是 360K 的软盘
2　在 1.2M 的驱动器中是 360K 的软盘
3　在 1.2M 的驱动器中是 1.2M 的软盘
3　在 720K 的驱动器中是 720K 的软盘
所有其他的值都无效
返回：　ah=返回状态（参看表 10-3）
进位=0，如果功能 1，如果出错

中断	功能	描述	平台
13h	18h	为格式化设置软盘类型	AT+

这个功能指定了所有类型驱动器上的媒质类型，在触发格式化功能 5 之前应设置软盘类型，但是如果已经设置了正确的媒质类型，或者接受缺省的类型，那么这一点就没有必要了。这有助于在高容量的驱动器上格式化低密度的软盘。例如，在 2.88M 的驱动器上格式化一个 720K 的软盘，CX 设置为 4F0Fh，为 720K 软盘所提供的值。

在很早期的 AT BIOS 上不可以执行这个功能，因为不支持 1.44M 的驱动器。这些老式的 BIOS 要求使用较灵活的功能 17h。

调用：　ah=18h
ch=格式化时的最大磁道号（值见下）
cl=格式化的最大扇区号（值见下）
dl=驱动器号，0 或 1
返回：　ah=返回状态（参看表 10-3）

es:di=指针，指向 11 个字节的软盘参数表（参看表 10-2）
进位=0，如果成功 1，如果出错

下面列出了用于格式化指定媒质类型的 CX 值。如果系统不支持指定的媒质，就会返回一个错误。

要格式化的软盘	最大磁道号	最大扇区号	CX 值
360K，5.25"	39	9	2709h
1.2M，5.25"	79	15	4F0Fh
720K，3.5"	79	15	4F0Fh
1.44M，3.5"	79	18	4F12h
2.88M，3.5"	79	36	4F24h

软盘 BIOS 数据

软盘 BIOS（中断 13h）在 BIOS 数据区为其操作使用了大量的字节。表 10-4 列出了这些数据的地址和功能，在第 6 章中讨论了其他细节。

表 10-4 BIOS 数据

地 址	大 小	描 述		
40:3Eh	字节	重校准状态——如第 0 到 3 位中的任何一位置 0，则指定的驱动器尚未校准，		
		并且在下次查找之前必须重新校准，系统重启之后，或者任何时候插入了一		
		张新盘，驱动器都是尚未校准的。		
		位	7=1	发生了软盘中断（中断 0Eh）
			6=x	未使用
			5=x	未使用
			4=x	未使用
			3=0	驱动器 3 尚未校准（仅 PC/XT）
			2=0	驱动器 2 尚未校准（仅 PC/XT）
			1=0	驱动器 1 尚未校准
			0=0	驱动器 0 尚未校准
40:3Fh	字节	马达状态——软驱马达和操作的状态。		
		位	7=0	当前操作——读或检查，所以没有必要延时来开
				启马达

续表

地 址	大 小	描 述			
			1	当前操作——写或写格式化，如果开启马达则需	
				要短等等（参看软盘参看表的偏移量 AH）	
			6=x	未使用	
			5=x	驱动器选择	
			4=x	第 5 位	第 4 位
				0	0=驱动器 0
				0	1=驱动器 1
				1	0=驱动器 2
				1	1=驱动器 3
			3=1	驱动器 3 马达开	
			2=1	驱动器 2 马达开	
			1=1	驱动器 1 马达开	
			0=1	驱动器 0 马达开	
40:40h	字节	马达超时——驱动器马达不活动超时的计数器。初始化值来自软盘参数表、偏移量 2。			
40:41h	字节	上次操作后软盘控制器的返回码（参看表 10-3）。			
40:42h	7 个字节	软盘控制器的结果寄存器。			
40:8Bh	字节	配置数据——该字节存在软驱的数据传输率。并非所有的 BIOS 会保存上次			
		驱动器的步进速率，这时，这些系统可能没有使用第 4 位和第 5 位。			
		位	7=x	传送到软盘的上次数据速率	
			6=x	第 7 位	第 6 位
				0	0=500Kbps
				0	1=300Kbps
				1	0=250Kbps
				0	0=尚未设置速率，或者在某些
					系统上是 1Mbps
			5=x	传送到软盘的上次步进速率	
			4=x	第 5 位	第 4 位
				0	0=8ms
				0	1=7ms（常用）
				1	0=6ms（常用）
				0	0=5ms
			3=x	操作开始时设定的数据速率	
			2=x	第 3 位	第 2 位

续表

地 址	大 小	描 述				
				0	0=500Kbps	
				0	1=300Kbps	
				1	0=250Kbps	
				0	0=1Mbps（如果支持）	
			1=x	大多数情况下未使用，在某些复制品上功能未知		
			0=x			
40:8Fh	字节	软盘0——媒质状态。				
		位	7=x	数据传送率		
			6=x	第7位	第6位	
				0	0=500Kbps	
				0	1=300Kbps	
				1	0=250Kbps	
				0	0=1Mbps（如果支持）	
			5=1	要求双步进（1.2M驱动器带360K软盘）		
			4=1	软驱上的媒质已知		
			3=0	未使用		
			2=x	确定上次访问		
			1=x	第2位	第1位	第0位
			0=x	0	0	0=在360K软驱中
						试用360K的媒质
				0	0	1=在1.2M软驱中
						试用360K的媒质
				0	1	0=在1.2M软驱中
						试用1.2M的媒质
				0	1	1=已知在360K软
						驱中360K的媒质
				1	0	0=已知在1.2M软
						驱中360KB的媒质
				1	0	1=已知在1.2M软
						驱中1.2M的媒质
				1	1	0=未用状态
				1	1	1=720K媒质在
						720K驱动器中或
						1.44M的媒质在
						1.44M驱动器中。
40:91h	字节	软盘1——媒质状态（参看40:90h）。				

续表

地 址	大 小	描 述			
40:92h	字节	软盘 0——操作超始状态。			
		位	7=x	数据传送率	
			6=x	第 7 位	第 6 位
				0	0=500Kbps
				0	1=300Kbps
				1	0=250Kbps
				0	0=1Mbps
			5=x	未知功能	
			4=x	未知功能	
			3=0	未使用	
			2=1	确定了驱动器类型	
			1=1	驱动器是多速率的	
			0=1	驱动器改变了软盘检测线	
40:93h	字节	软盘 1——操作起始状态（参看 40:92h）。			
40:94h	字节	软盘 0——当前磁柱——该字节保存了驱动器 0 当前所在的磁柱。			
40:95h	字节	软盘 1——当前磁柱——该字节保存了驱动器 1 当前所在的磁柱。			

软盘控制器的常见类型

在过去 10 年中，有一些常见类型的软盘控制器用于系统中，所有的这些控制器都是基于 765 控制器，后者用于第一台 IBM PC 上，从软件的视角来看，所有的这些类型的功能源于原始的 765，但是可能提供了额外的增强功能或额外的命令。

765（NEC uPD765A） 该 IC（或 100%兼容类型）用在许多软盘适配卡上，一些制造商也生产了与之功能等同的 IC。765 有 15 条命令用于读、写和格式化软盘。

72065A（NEC uPD765A） 这个类型类似于早期的 765，只是作了一些硬件增强，同时加入了另外三条用于控制备用电源和重启的命令。

72065B（NEC uPD765B） B 版本加入一条新命令来确定现在使用哪个芯片版本，其他功能与 72065A 相同。

8272（Intel 8272A） 功能上等同于 765。

82077（Intel 82077AA） 组合了所有逻辑的最新 IC 之一，这些逻辑对于一个完全 PC 兼容的软盘控制器很有必要。82077 加入了许多增强功能，包括输入输出数据的 FIFO 模式，并且支持 2.88M 驱动器。82077 比 765/8277 多了 8 条新命令。

82078（Intel 82078） 提供了 Intel 82077 的所有特性，另外，82078 提供了选项支持磁带驱动器的 2Mbps 数据率、快速操作的新命令、以及标识硬件媒质和驱动器类型。

向软盘控制器发送命令

软盘控制器有一系列广泛的命令来执行各种操作，例如格式化一个磁道或读一个扇区。这些命令后通常接着信息字节，例如起始磁道和扇区。一旦接受了该命令，控制器就执行该操作。大多数情况下，在完成了该命令后，控制器返回许多状态字节。取决于命令类型，控制器可能还会触发 IRQ6（中断 Eh）来指示任务完成。整个过程由主状态寄存器中的两位同步。

该过程先从端口 3F4h 主状态寄存器读取数据传输模式。如果第 7 位和第 6 位各是 1 和 0，那么要写入控制器的命令已准备好。于是该命令字节发送到数据寄存器，端口 3F5h。监视主状态寄存器来检测软盘控制器何时准备好下一个参数字节，正如命令可能会指定的那样。当接收了所有的参数之后，控制器继续执行该命令。

在执行该命令期间，如果读取或写入了数据，不需要检查主状态寄存器。通过数据寄存器端口 3F5h 读或写数据。至于软盘数据读写中通常用到的 DMA 操作，数据如以前设置的那样通过 DMA 控制器传送。

完成了执行过程后，无论成功与否，大多数命令都会提供结果信息。在读取每个结果字节之前，必须先再次检查主状态寄存器。这时如果第 6 位和第 7 位都是“1”，那么一个结果字节准备好可以读取，从控制器数据端口 3F5h 读取该字节。如果读取了所有的状态字节，并且完成了该命令的操作，那么主状态寄存器会指示它已准备好下一条命令。

参看本章后面的端口 3F5h 以了解命令、参数和结果信息。

警 告

在程序直接访问硬件的操作中，对软盘控制的操作是最复杂的一种，如果可能，应使用中断 13h BIOS 服务作为接口，因为 BIOS 可处理不同 PC 平台间大量的微妙差别。

BIOS 扇区限

通常，BIOS 只支持同时传输两个磁道上的最大数据。为了高度可靠性，我建议不要同时读取一个以上的磁道。

在所有情况下，读取一个磁道的数据时，假定第一个扇区位于第 0 面的起点。在一次操作中读取两个磁道时，控制器会从第 0 面读取数据，并且如果必要，会将磁头切换到第 1 面以读取另外的扇区。这意味着你必须确切地知道数据相对于物理磁道和磁头的位置。

另外，软盘媒质可能不同于驱动器类型。如果一个 2.88M 驱动器读一个磁道，它会读总共 36 个扇区，如果同样的 2.88M 驱动器从一个 720K 软盘读取磁道，同时只会读取 9 个扇区。知道这一点很重要，因为没有 BIOS 服务会指出当前媒质类型，而只指出驱动器类型。

第 11 章描述了程序 DISKTEST，可用于确定软盘媒质类型，也可以测试一次可以可靠读取的扇区数。在我所进行的测试中，大多数系统可以一次正确地传输多于一个磁道。

DISKTEST 也显示出读操作之后的 AL 返回值。所有的系统显示了传输的扇区数目，但是某些 BIOS 会返回一个值，限于每个磁道的最大扇区数。这意味着，如果读取两个磁道，AL 返回值可能会错误地指示只读取一个磁道。

传输数据越过 64KB 边界的问题

当从或向软盘传送数据时（功能 2 和 3），缓冲区没有越过 64K 物理边界这一点很重要。在 BX 寄存器中使用一个低偏移量值并不会保证你不会越界。你还必须关注 ES 的值。例如，一个缓冲区的 ES:BX 等于 1FFF:0 会使大多数系统工作失败，本例中，处理了前 16 个字节后，数据将会出现在内存的下一个物理 64K 部分中。大多数程序设计成可重定位，这要求你加入专用代码来找到一个合适的位置作为你的软盘缓冲区内存，新程序员使用软盘服务时，这是他们最易犯的错误。因为它看起来在许多系统又可以工作。

如果你使用的是一个硬盘高速缓存程序，硬盘的高速缓存（如果写正确）会使你发现不了该问题。

软件计时问题

老式的软盘子系统可能是 PC 中最不稳定的部分。这部分由于 BIOS 在软件中执行了大量的计时循环、软盘 BIOS 代码应该和 CPU 及 CPU 速度相匹配，但是许多复制品和系统更改使 BIOS 软件计时循环太短。例如，如果 BIOS 代码是为 80486，33MHz 而编写，并且用在一个 100MHz 的类型上，所有的软件计时循环大约只会运行正常时间的三分之一。这经常造成读或写数据失败和不恰当的超时，另一个常见的问题是 BIOS 的阴影效果，系统或内存管理程序使用这种性能，来将慢速 8 位宽系统 BIOS ROM 复制到 32 位宽的快速 RAM 中。这意味着计时循环可能执行得更快，这取决于它们是如何编码的。该问题对下列系统还不太严重：新式的带有较大预取队列的系统，正如 80486 和后来的 CPU 提供的那样，和带有板上内存高速缓存的系统。

通过三种方法控制计时，最稳定可靠的方法是利用系统时钟，中断 8（参看第 16 章）。递减的软件循环常常用作短时间周期。循环预设的次数。在这种情况下，计时完全取决于在给定的 CPU 和 CPU 速度一循环中每条指令的执行速度。简单地跳到下一条指令会生成一个更短的延时。表 10-5 列出了软盘 BIOS 处理程序内主要的功能，它们依赖于定时完成

各种功能。

表 10-5 大多数软盘 BIOS 处理程序内的定时功能

功 能	定时时	常见的延
马达从关到正常速需的时间（仅写操作）	软件循环	500ms
马达从开到关而不进行另外的访问操作的时间	中断 8	2s
等待软盘控制器响应时间	软件循环	500ms
找到磁道后的磁头定位时间	软件循环	25ms
等待软盘控制器操作完成中断的时间	软件循环	2ms
对控制器连续 I/O 操作的时间间隔	软件跳转	0.5μs

某些较新的系统使用一种巧妙的方法来生成可靠的短计时循环。端口 61h 的刷新位第 4 位在每个其他的 DRAM 刷新循环中切换。可以对刷新位计数来生成短计时循环，可以不受 CPU 速度的影响。不幸的是，没有为刷新计时循环设置标准，所以它仅对专用主板的 BIOS 设计者有用。

代码例 10-1 检测软驱

该个程序检查是否带有指定的软驱，并确定其最大容量。程序所工作的系统可以带有多达四个软驱，而只使用一个软盘控制器卡。当然，AT/EISA/MCA 的 BIOS 只支持两个软驱，但是如果一个四驱控制器带有它自己的 BIOS，那么它就可以在这些系统上支持多达四个的驱动器。

这个程序作为一个子程序被包括在 SYSTYPE.ASM 中，运行 SYSTYPE 来看 DISKTYPE 关于一个系统上四个可能软驱的结果。

```
; ---------------------------------------------------------------------------
;    软驱类型检测子程序
;       确定软驱类型
;
;       调用:          dl = 要检查的驱动器 (0=a:, 1=b: 等)
;
;       返回:          al = 驱动器类型
;                           0 = 无驱动器
;                           1 = 360K
;                           2 = 1.2M
```

```
;                       3 = 720K
;                       4 = 1.44M
;                       5 = 2.88M
;                       6 = 320K（过时）
;                       7 = 未知类型
;
;       用到的寄存器:    ax

dsktype proc    near
        push    bx
        push    cx
        push    dx
        push    es

        int     11h             ; 设备检查
        test    ax, 1           ; 任意驱动器?
        jz      dskt_none       ; 如果不是，则跳转
        and     al, 0C0h        ; 获得驱动器位号
        mov     cl, 2
        rol     al, cl          ; 转换成驱动器号
        cmp     al, dl          ; 驱动器可用吗?
        jb      dskt_none       ; 如果不可用，则跳转

; 我们首先试一试从BIOS获得驱动器参数

        mov     ah, 8
        int     13h             ; 获得驱动器参数
        jc      dskt_skp1       ; 如果失败，则跳转
        mov     al, bl          ; 获取驱动器类型
        cmp     al, 5           ; 如果大于5，则类型未知
        jbe     dskt_done
        mov     al, 7           ; 设置为未知类型
        jmp     dskt_done

; 旧式系统不提供驱动器信息，所以要
;   使用软盘表。首先重启驱动器。这将强
```

```
;   制系统将软盘参数指针指向另外其他的
;   驱动器类型以确保类型指向正确。
;   中断向量1Eh会指向正确的软盘参数表。
;   仅360KB和1.2MB的驱动器会执行到此处。

dskt_skp1:
        xor     ah, ah
        int     13h                     ; 重启驱动器
        xor     ax, ax
        mov     es, ax
        mov     bx, es:[1Eh*4]          ; 获取表的偏移量
        mov     ax, es:[1Eh*4+2]        ; 获取表的段值
        mov     es, ax                  ; es:bx指向表
        mov     ah, es:[bx+4]           ; 获取每磁道的扇区数
        mov     al, 1                   ; 假定为360K
        cmp     ah, 9                   ; 每磁道9扇区?
        je      dskt_done               ; 如果是，则跳转
        inc     al
        cmp     ah, 15                  ; 每磁道15扇区?
        je      dskt_done               ; 如果是，则跳转
        mov     al, 6                   ; 未知类型
        cmp     ah, 8                   ; 每磁道8扇区?
        je      dskt_done               ; 如果是，则跳转
        inc     al                      ; al=7、未知类型
        jmp     dskt_done

dskt_none:
        xor     al, al
dskt_done:
        pop     es
        pop     dx
        pop     cx
        pop     bx
        ret
dsktype endp
```

端口归纳

下面的端口列表用于控制软盘控制器。注意，所支持的寄存器因平台而有所不同。

端口	类型	功能	平台
3F0h	输入	软盘控制器状态寄存器 A	PS/2
3F1h	输入	软盘控制器诊断寄存器	有限
3F1h	输入	软盘控制器状态寄存器 B	PS/2
3F2h	I/O	软盘控制器数字输出寄存器	所有
3F3h	I/O	软盘控制器磁带驱动器寄存器	有限
3F3h	输入	软盘控制器主状态寄存器	PS/2
3F4h	输入	软盘控制器主状态寄存器	所有
3F4h	输出	软盘控制器数据选择寄存器	无
3F5h	I/O	软盘控制器数据	所有
3F6h	I/O	软盘控制器保留	所有
3F7h	输入	软盘控制器数字输入寄存器	AT+
3F7h	输出	软盘控制器配置控制寄存器	AT+

端口细节

端口	类型	描述	平台
3F0h	输入	软盘控制器状态寄存器 A	PS/2

这个寄存器从控制器返回中断引脚的状态以及各种驱动器导线的当前状态。只有 PS/2 产品才支持它，并且在 PS/2 机器的 MCA 和 AT 类型之间表达有所不同。AT 或 ISA 兼容的系统不支持这个寄存器。

仅 PS/2 MCA 总线

来自软驱连线的输入将反映软驱的状态。硬件重启后，软驱的输出、中断悬挂、步进、磁头选择，以及方向都会是 0。

输入（第 0~7 位）

位	7 r=1	中断悬挂——跟踪来自控制器的中断请求线的当前状态
	6 r=0	带有第二软驱（输入来自软盘导线）
	5 r=1	步进——监视到软盘导线的 STEP 输出的当前状态
	4 r=0	软驱指示磁头目前位于磁道 0 上（输入来自软盘导线）

3 r=0	导线选中了磁头 0（软盘第 0 面）
1	导线选中了磁头 1（软盘第 1 面）
2 r=0	软驱指示它正在磁道起点（输入来自软盘导线）
1 r=0	软驱指示软盘写保护（输入来自软盘导线）
0 r=0	出方向——磁头从软盘中央向外沿移动
1	入方向——磁头从外沿向中央移动

仅 PS/2 AT 总线（即型号 30 及其他）

来自软驱连线的输入将反映软驱的状态。硬件重启后，软驱的输出、中断悬挂、DMA 请求，以及步进触发都会是零。硬件重启后，磁头选择和方向位都会置 1。

输入（第 0~7 位）

位	
7 r=1	中断悬挂——跟踪来自控制器的中断请求线的当前状态
6 r=1	DMA 请求——跟踪来自控制器的 DMA 请求线的当前状态
5 r=1	当 STEP 输出线激活时，设置步进触发当读数字输入寄存器（端口 3F7h）或发生软/硬件重启时，都会重新设置它。
4 r=0	软驱指示磁头目前位于磁道 0 上（输入来自软盘导线）
3 r=0	导线选中了磁头 0（软盘第 0 面）
1	导线选中了磁头 1（软盘第 1 面）
2 r=0	软驱指示它正在磁道起点（输入来自软盘导线）
1 r=0	软驱指示软盘写保护（输入来自软盘导线）
0 r=0	出方向——磁头从软盘中央向外沿移动
1	入方向——磁头从外沿向中央移动

端口	类型	描述	平台
3F1h	输入	软盘控制器诊断寄存器	有限

这个功能用在一些适配卡上来表示控制器是否接受多于一个的适配器类型。这是一种最早的方法来在同一系统上同时支持 360K 和 1.2M 驱动器。早期的 AT 系统和某些复制品都支持这个寄存器。

输入（第 0~7 位）

位	
7 r=x	适配器类型
6 r=x	01010=支持多种驱动器类型（360K 和 1.2M）
5 r=x	
4 r=x	
3 r=x	
2 r=x	未知或未使用
1 r=x	未知或未使用
0 r=x	未知或未使用

端口	类型	描述	平台
3F1h	输入	软盘控制器状态寄存器 B	PS/2

这个寄存器返回控制器在不同的驱动器导线上的输入和输出当前状态。只有 PS/2 产品才支持这个功能，并且其功能和状态在 MCA 和 AT 类型的 PS/2 机器之间有所不同。ISA 或 EISA 兼容的系统不支持这个寄存器。

仅 PS/2 MCA 总线

一个软件或硬件重启会清除读写触发位，硬件重启还会清除驱动器选择和马达选择位。

输入（第 0~7 位）

位		
	7 r=1	固定为 1
	6 r=1	固定为 1
	5 r=1	选择驱动器 0——监视控制器的输出驱动器 A 选择线
	4 r=x	写数据触发——将每个写的数据切换到软盘
	3 r=x	读数据触发——将每个写的数据切换到软盘
	2 r=1	可写——可写输出线的状态
	1 r=1	开放驱动器 1 马达——驱动器 1 马达开放输出线的状态
	0 r=1	开放驱动器 0 马达——驱动器 0 马达开放输出线的状态

仅 PS/2 AT 总线（例如，模式 30 及其他）

一个软件或硬件重启会清除读、写以及读和写开放触发位。从数字输入寄存器（端口 3F7h）读也可以重设这些触发位。

输入（第 0~7 位）

位		
	7 r=0	带有第二个软驱（来自软盘导线的输入）
	6 r=0	选择驱动器 1（监视控制器的驱动器 1 输出选择线）
	5 r=0	选择驱动器 0（监视控制器的驱动器 0 输出选择线）
	4 r=x	写数据触发——开启到软驱的数据写线的活动边界
	3 r=x	读数据触发——开启到通线的活动边界
	2 r=1	写通触发——开启写通线的活动边界
	1 r=1	选择驱动器 3（监视控制器的驱动器 3 的输出选择线，在不同的系统上可能有不同的驱动器字母）
	0 r=1	选择驱动器 2——（监视控制器的驱动器 2 的输出选择线，在不同的系统上可能有不同的驱动器字母）

端口	类型	描述	平台
3F2h	I/O	软盘控制器数字输出寄存器	所有

这个端口控制驱动器选择、驱动器马达以及 DMA 操作。大多数适配器只支持驱动器 0 和 1（A: 和 B:）。在这种情况下，来自软盘控制器芯片的驱动器 3 和 4 没有连上或不执行任何操作。通常同时选择驱动器和开启其他马达。

如果选择了一个驱动器，软件必须在读或写之前延时磁头装入时间，这段时间是磁头首次与软盘相接触到首次读写之间的时间。这样可以避免在磁头与软盘相接触之后的间跳发生读写问题。在早期带有老式 320K 软驱的 PC 上，磁头装入了的时间是 35ms。在现在的驱动器上，8ms 或更少就足够了。软盘参数表的字节 1 指定了实际的磁头装入时间。

I/O（第 0~7 位）

位	7 r/w=1	马达开，驱动器 3*
	6 r/w=1	马达开，驱动器 2*
	5 r/w=1	马达开，驱动器 1
	4 r/w=1	马达开，驱动器 0
	3 r/w=0	禁止 DMA 和中断功能——禁止对系统的硬件 DMA 请求，并忽略来自系统的 DMA 确认和终端计数。也禁止对系统的中断请求。在 PS/2 MCA 系统上总是开放 DMA 和中断功能，并且忽略该位
	1	DMA&中断功能开放（适合所有的 PS/2 MCA 系统设置）
	2 r/w=0	对软盘控制器和内部 FIFO 进行软件重设，控制器保持重设模式直到该设为 1。软件重设会清除该位并保持在重设模式直到该位由软件设置为 1
	1	普通操作
	1 r/w=x	驱动器选择
	0 r/w=x	

位 1	位 0
0	0=选择驱动器 0
0	1=选择驱动器 1
1	0=选择驱动器 2*
1	1=选择驱动器 3*

注：*可能不支持

端口	类型	描述	平台
3F3h	I/O	软盘控制器磁带驱动器寄存器	有限

某些软盘控制器支持附带在软盘驱动器导线上的磁带驱动器。这个寄存器用于选择哪个驱动器（1~3）与磁带驱动器相连。大多数软盘适配器控制器不支持它。

驱动器 0 总是作为一个软盘引导驱动器而保留，不能把它分配给某个磁带驱动器。

I/O（位 0~1）

位		
	7 r=x	未使用，可能呈现任何值
	6 r=x	未使用，可能呈现任何值
	5 r=x	未使用，可能呈现任何值
	4 r=x	未使用，可能呈现任何值
	3 r=x	未使用，可能呈现任何值
	2 r=x	未使用，可能呈现任何值
	1 r/w=x 0 r/w=x	选择驱动器

位 1	位 0
0	0=没有磁带驱动器
0	1=驱动器 1 是磁带驱动器
1	0=驱动器 2 是磁带驱动器
1	1=驱动器 3 是磁带驱动器

端口	类型	描述	平台
3F3h	输入	软盘控制器驱动器状态寄存器	PS/2

较新的 PS/2 系统支持该状态信息寄存器。驱动器状态寄存器指示了软盘媒质类型和驱动器容量。该额外的状态信息使带有 2.88M 软驱的 PS/2 系统运行比我所见的大多数其他方案都要可靠和快速。关键在于驱动器指示了媒质类型。其他大多数机器，2.88M 和 1.44M 线是断开的，或者不提供该线，它们需要软件来确定当前软驱中的软盘类型。

输入（位 0~1）

位		
	7 r=x	驱动器中的媒质类型（只有选择了一个驱动器后才有效）
	6 r=x	

位 7	位 6
0	0=未定义（不应出现这种情况）
0	1=2.88M 软盘媒质
1	0=1.44M 软盘媒质
1	1=720K 软盘媒质

5 r=x	驱动器类型（只有当选择了一个驱动器后才有效）
4 r=x	

位 5	位 4
0	0=1.44M, 3.5"
0	1=2.88M, 3.5"
1	0=1.2M, 5.25"
1	1=未定义（不应该出现这种情况）

3 r=x 2 r=x	用作引导启动的盘

位 3	位 2
0	0=软驱 0

	0	1=软驱1
	1	0=软驱2（如果支持）
	1	1=未定义（不应该出现这种情况）
1 r/w=x	未定义或功能未知	
0 r/w=x	未定义或功能未知	

端口	类型	描述	平台
3F4h	输入	软盘控制器主状态寄存器	所有

该只读寄存器指示了控制器及送往控制器的命令的当前状态。必须检查数据传输模式位以装入模式字节和从控制器获取状态字节。如果不允许访问，那么 BIOS 必须跳入循环检查这个寄存器，直到控制器准备好，支持从端口 3F5h 的额外写或读。

输入（位 0~1）

位		
7 r=x	数据传送模式（使用端口 3F5h）	
6 r=x	位 7	位 6
	0	0=不允许访问
	0	1=不允许访问
	1	0=数据写准备好
	1	1=数据读准备好
5 r=1	命令正处于执行状态（仅非 MCA 模式）	
4 r=1	正在进行读写命令	
3 r=1	驱动器 3 忙* **	
2 r=1	驱动器 2 忙* **	
1 r=1	驱动器 1 忙*	
0 r=1	驱动器 0 忙*	

注：*软驱正在查找或处于命令的重校位置
**可能不支持

端口	类型	描述	平台
3F4h	输出	软盘控制器数据率选择寄存器	有限

该只写寄存器主要用于设置数据率。大多数软盘控制器不支持它。这个寄存器也可以控制一个可替换的重启、节电模式以及写预补偿。为了为 2.88M 媒质设置数据率，必须向这个寄存器写入值 3，假定控制器支持新的 1Mbps 的数据率。

PC 系列通常使用缺省的写补偿值，写补偿用来处理一种现象，这种现象带有的磁媒质可能会“漂移”某些位模式。如果检测到这些数据模式。控制器就会或快或慢地漂移计时位来补偿该现象。一旦写入了预补偿数据，就可以正常地读取该数据。

软件重启并不改变当前的数据率或写预补偿值，硬件重启会将数据率设置为 250Kbps，

并将写预补偿设定为 25ns.

这个寄存器仅对 PS/2 系统有效，尽管 IBM 并没有定义第 5~7 位。在老式的 PS/2 系统上，不支持选择 300Kbps 的数据传输率。在一个 1.2M 软驱中读一个 360K 寄存器时要求 300Kbps 的数据率。

输出（位 0~1）

位				
	7 w=0	正常		
	1	软件重启——一个重启周期后位自动置为 0，这一点不同于平均数字输出寄存器，它会保存在重启模式		
	6 w=0	正常		
	1	省电模式，驱动器的输出被三态化，并且关闭了内部振荡器。软件或硬件重启会将系统返回到正常状态。		
	5 w=0	使用内部状态锁循环（正常）		
	1	外部状态锁循环（要求有外部硬件，并且从未用在 PC 的软盘控制器设计中）		
	4 w=x 3 w=x 2 w=x	写预补偿		
		位 4	位 3	位 2
		0	0	0=缺省使用（参照第 0 和 1 位）
		0	0	1=41.67ns
		0	1	0=83.34ns
		0	1	1=125.00ns
		1	0	0=166.67ns
		1	0	1=208.33ns
		1	1	0=250.00ns
		1	1	1=0（没有写预补偿）
	1 w =x 0 w =x	数据率选择		
		位 1	位 0	速率 缺省的写预补偿
		0	0	500kps 125ns
		0	1	300kps 125ns
		1	0	250kps 125ns
		1	1	1Nbps* 41.67ns

* 并非所有的控制器都支持 1Mbps

端口	类型	描述	平台
3F5h	I/O	软盘控制器数据	所有

所有的磁盘数据传送和命令参数都通过这个端口传送。支持 FIFO 并且激活了 FIFO 模式的控制器会通过软盘控制器的 16 字节 FIFO 发送数据。主状态寄存器，端口 3F4h 的第 6

和7位由软件监视来调节读写。

标准的参数和状态

大多数命令要求在命令之后装入相同的参数，并且在完成执行状态时返回相同的状态字节集。表10-6列出了标准参数，而表10-7列出了标准的状态结果。

表10-6 标准参数（输出到控制器）

字节0：	命令（参看命令归纳）。		
字节1：	驱动器选择和磁头选择字节。		
位	7=0	未使用或计数关	
	1	计数开	
	6=0	未使用	
	5=0	未使用	
	4=0	未使用	
	3=0	未使用	
	2=x	磁头号（0=第0面，1=第1面）	
	1=x	驱动器选择	
	0=x	位1	位0
		0	0=选择驱动器0
		0	1=选择驱动器1
		1	0=选择驱动器2（可能不支持）
		1	1=选择驱动器3（可能不支持）
字节2：	磁柱号——磁道号和磁柱号相同（0~79，与媒质有关）。		
字节3：	磁头号字节（0=第0面，1=第1面）。		
字节4：	动作扇区，或者多扇区操作时的第一个扇区字节。		
字节5：	扇区大小代码字节（通常为2）。		
	0=128字节		
	1=256字节		
	2=512字节（所有PC系列软盘的标准用法）		
	3=1024字节		
	4=2048字节（并非所有媒质都支持）		
	5=4096字节（并非所有媒质都支持）		
	6=8192字节（并非所有媒质都支持）		
	7=16384字节（并非所有媒质都支持，某些控制器也不支持）		
字节6：	磁道尾字节——一个磁道上可能的最后一个扇区号（参看媒质表10-1）。		
字节7：	两个扇区字节之间的间隙长度（来自软盘参数块）。5.25" 360K和1.2M的媒质使用间隙长度值2Ah。3.5" 720K到2.88M媒质都使用间隙长度值1Bh。标准512字节大小之外的扇区大		

续表

	小需要不同的间隙长度值。
字节 8：	用户扇区大小字节——所有的 PC 系列系统都未使用，应该设定为 FFh。如果将扇区大小值设定为 0，该字节将控制字节数目。

表 10-7 标准的状态结果（来自控制器的输入）

字节 0：状态寄存器 0			
位	7=x	中断码	
	6=x	位 7	位 6
		0	0=正常终止，没有错误
		0	1=非正常终止，执行开始了但未结束
		1	0=尝试了无效的命令
		1	1=因驱动器的准备线非正常终止（由于驱动器的准备线总维持高电平，所以不会发生该错误）
	5=1	查找尾——完成查找和重校命令，或者读写隐含查找完成	
	4=1	设备检查——在一个重校命令期间，驱动器没有指示检测，或者相对个查找命令试图移动到一个无效的磁道号（例如，试图移动到磁道 0 外）	
	3=0	准备状态（总为 0）	
	2=x	当前磁头号（0=第 0 面，1=第 1 面）	
	1=x	当前驱动器选择	
	0=x	位 1	位 0
		0	0=选择驱动器 0
		0	1=选择驱动器 1
		1	0=选择驱动器 2（可能不支持）
		1	1=选择驱动器 3（可能不支持）
字节 1：状态寄存器 1			
位	7=1	扇区越过磁道尾错误——试图访问一个越过了磁道最后一个扇区的扇区	
	6=0	未使用	
	5=1	数据出错——在扇区的 ID 区或数据区检测到了 CRC 错误	
	4=1	超时或轻载错误——CPU 或 DMA 传输数据太快或太慢，所以丢失了至少一个字节	
	3=0	未使用	
	2=1	无数据错误——在试图读数据或读已被删除的数据时，控制器找不到指定的扇区。如果读 ID 命令不能读取无错的 ID 时，也会发生数据错误。读磁道命令如果找不到正确的扇区系列，那么也会生成该错误	
	1=1	写保护错误——在试图执行数据写、写已删除的数据或格式化磁道时，激活了写保护线	

续表

	0=1	丢失地址标签错误——驱动器产生了两个索引脉冲（即，至少有
		了一个完整的软盘循环），但在指定的磁道上找不到 ID 地址标签。
		如果控制器的寻到指定磁道上的数也会出现这种错误
字节 2：状态寄存器 2		
位	7=0	未使用
	6=1	控制标签错误——在执行指定的数据字节时，（读数据或读已被删
		除的数据），检测到了数据地址指针的相反类型。如果发一条普通
		的读数据命令，将会找到一个已被删除了的数据地址标签。如果触
		发了读已删除数据的命令，将会找到一个非删除的数据地址标签
	5=1	数据 CRC 错误——控制器在扇区的数据区检测到了 CRC 错误
	4=1	错误的磁柱——控制器中的磁道地址找不到扇区 ID 区中相匹配
		的磁道号，该错误条件也设置状态寄存器 1 的第 2 位为值 1
	3=0	未使用（扫描命令除外——参看扫描命令）
	2=0	未使用（扫描命令除外——参看扫描命令）
	1=1	坏磁柱——条件和上面的错误的磁柱机同，但是在扇区 ID 中
		的磁道号是 FFh，表明该磁柱以前被标为有缺陷和有硬错误。该
		错误条件也设置状态寄存器 1 的第 2 位为值 1
	0=1	丢失地址标签——就在扇区的实际保存数据之前，保存了一个
		地址标签字节。该地址标签字节只能是两个值中的一个，0F8h 或
		0FEh，表示要么是跟着普通的数据，要么是跟着已被删除的数据，
		一旦检测到不是这两个值就会发生该错误
字节 3：完成时的磁柱号——磁道号和磁柱号相同（0~79，取决于媒质）		
字节 4：完成时的磁头号——0=第 0 面，1=第 1 面		
字节 5：完成时的扇区号——1 到 37，取决于媒质		
字节 5：完成时的扇区号——1 到 37，取决于媒质		
字节 6：完成时的扇区大小码——参看标准参数字节 5 以了解扇区大小码，2=512 字节		

端口 3F5h 命令归纳

软盘控制器有许多命令，但是只列出了那些对控制器编程必要的端口。某些被忽略的命令用于低密度过时的软盘格式中，任何 PC 家族系统从未使用过它们。其他被忽略的命令仅是命令列表中的复制品，它们降低了运行速度但没有任何好处。描述会提示那些命令，它们从未被 BIOS 软盘服务小程序使用过，但可能仍很有用。

只用于增强型控制器的命令用“增强”标出，它们不能用在标准的控制器上，使用版本命令 10h 来检测增强型的控制器。即使版本命令本身是一个增强命令，但是对于普通控制器的返回状态仍然有效。对于所有的无效命令，控制器在结果字节中返回 80h。

十六进制	描述	控制器
3	指定	所有
4	传感驱动器状态	所有
7	重校	所有
8	传感中断状态	所有
E	倾倒寄存器	增强
F	查找	所有
10	版本	所有
12	垂直 2.88MB	增强
13	配置	增强
14	开放 FIFO 功能	所有
34	退出备用待机模式	仅 72065
35	进入备用待机模式	仅 72065
36	硬重启	仅 72065
42	读磁道	所有
4A	读 ID	所有
4D	格式化磁道	所有
8F	向外相对查找	增强
94	锁住 FIFO 功能	增强
C5	写数据	所有
C6	读普通或已删除的数据	所有
C9	写删除的数据	所有
CC	读删除的数据	所有
CF	向内相对查找	增强
D1	全扫描等值比较	所有
D6	核对	增强
E6	读普通数据	所有
F1	扫描等值比较	所有
F6	核对	增强

命令	描述	端口
3h	指定	3F5h

该命令装入控制器的三个内部时钟，并选择普通 DMA 模式或非 DMA 模式。我所见到的每个 PC 系统都对软盘系统使用 DMA 模式，计时值的范围随着更快的数据率而自动缩小。这意味着一个值会适用于多种媒质类型。

步进速率时钟

步进速率时钟控制磁道步进马达各步之间的时间长度。步进率选择的时间与数据率有关，选择值 Fh 表示最短时间，选择值 0 表示最长的时间。大多数系统对 1.44M 和 2.88M 驱动器使用值 Ah，对其他驱动器使用 Dh。

选择的数据率：	1Mbps	500bps	300bps	250bps
媒质：	2.88M	1.44M/1.2M	360K*	720K/360K
		步进速率（ms）		
值 0	8.0	16	26.7	32（最长延时）
值 1	7.5	15	25.0	30
：				
值 A	3.0	6	10.0	12
值 B	2.5	5	8.33	10
值 C	2.0	4	6.67	8
值 D	1.5	3	5.0	6
值 E	1.0	2	3.33	4
值 F	0.5	1	1.67	2（最短延时）

注：*在 1.2M 驱动器中的 360K 媒质

磁头下载时间

磁头下载时间是完成读/写命令后等待的时间，供控制器将软盘磁头从媒质上移走。可以设置该时间以便在连续读写扇区划时让磁头停留在媒质上，但是一直没有发生立即访问就下载磁头以保护媒质和磁头磨损，值 1 表示最短时间，然后逐渐增加到 F，值 0 表示最大延时。

大多数系统用值 F，除了 PS/2 使用值 1。我看不出有什么理由大多数的 BIOS 制造商使用那么长的下载时间，因为标准的装入时间很短。速度效果显得很小。我只能假定他们遵循 IBM 在早期的 PC/XT/AT 中设定的值。

选择的数据率：	1Mbps	500bps	300bps	250bps
媒质：	2.88M	1.44M/1.2M	360K*	720K/360K
		磁头下载时间（ms）		
值 0	128	256	426	512（最长延时）
值 1	8	16	26.7	32
：				
值 E	112	224	373	448
值 FB	120	240	400	480

注：*位于 1.2M 驱动器中的 360K 媒质

磁头装入时间

磁头装入时间是从磁头正在装入（即：与软盘相接触）到开始读或写之间的时间。之所以要求该延时，是因为在首次装入的一段时间内磁头会回跳。在这个期间读或写的数据会有错。大多数系统将该延时设置为 1 或 2。

选择的数据率：	1Mbps	500bps	300bps	250bps
媒质：	2.88M	1.44M/1.2M	360K*	720K/360K
		磁头装入时间（ms）		
值 0	128	256	426	512（最长延时）
值 1	1	2	3.3	4
值 2	2	4	6.7	8
值 3	3	6	10.0	12
：				
值 C	127	254	423	308

*位于 1.2M 驱动器的 360K 媒质

命令　字节 0=3

字节 1	位	7=x	步进速率时间（0~Fh）
		6=x	参看上面的计时值
		5=x	
		4=x	
		3=x	磁头下载时间（0~Fh）
		2=x	
		1=x	
		0=x	
字节 2	位	7=x	磁头装入时间（0~7Fh）
		6=x	
		5=x	
		4=x	
		3=x	
		2=x	
		1=x	
		0=0	DMA 模式（正常选择）
		1	非 DMA 模式

执行——装入指定的参数
结果——无结果字节，无中断

命令	描述	端口
4h	传感驱动器状态	3F5h

获取指定驱动器的当前状态，包括来自软驱的几条线的当前状态，以及到驱动器的磁头和驱动器选择线的状态。

命令 字节0=4

字节1=驱动器选择和磁头选择字节（参看表10-6中的标准参数）

执行——获取状态

结果——没有中断

字节0=状态寄存器3

位		
	7=0	无驱动器缺省（所有系统上恒为0）
	6=1	写保护软盘（来自驱动器）
	5=1	驱动器准备好（所有系统上恒为1）
	4=1	磁头位于磁道0（来自驱动器）
	3=1	双面（所有系统上恒为1）
	2=0	磁头选择第0面（到驱动器）
	1	磁头选择第1面（到驱动器）
	1=x 0=x	驱动器选择

位1	位0
0	0=选择驱动器0
0	1=选择驱动器1
1	0=选择驱动器2
1	1=选择驱动器3

命令	描述	端口
7h	重校	3F5h

重校命令所选择的软驱将其磁头移动到磁道0，使用这个命令是为了让所有的动作能正确地定位到软盘起点，磁道0。一旦插入了一个新软盘或初始化了控制器和驱动器，就要求使用该命令。

在执行状态下，重校监视来自软驱的硬件线来指示驱动器是否位于磁道0。该命令会向外步进磁头直到激活了驱动器的零磁道线。在执行的过程，控制器设置为不忙状态，这就允许另外一个软驱触发重校命令。这一点可用于加速多驱动器的安装初始化。

如果步进多于79步而仍没有激活零磁道信号，就会终止该命令。注明错误条件并设置寄存器0的设备检查位。无论命令是否成功完成，都会在寄存器0中设置查找结束位。

在命令完成时，不返回状态字节，但是控制器会触发中断Eh，来指示操作完成，下面叙述的传感中断状态命令，应被触发以核对命令是否成功完成。传感中断命令返回状态寄存器0和当前的磁柱号，它必须是0。

命令 字节 0=7

字节 1=0	选择驱动器 0
1	选择驱动器 1
2	选择驱动器 2
3	选择驱动器 3

执行——将磁头拉回到磁道 0

结果——没有结果字节，但是会触发一个中断

命令	描述	端口
8h	传感中断状态	3F5h

该命令清除控制器触发的中断并返回中断发生的原因。该传感中断状态命令只能在中断发生之后触发，如果没有激活中断，那么会返回错误值 80h。

在一系列的命令完成其执行状态时，控制器会触发一个中断，这些命令包括写命令、读 ID、格式化磁道、核对、所有的查找命令以及重校命令。在查找命令和重校命令之后必须触发传感中断命令来终止它们的操作并检查磁头位置是否正确。

命令 字节 0=8

执行 获取状态

结果——无中断

字节 0=状态寄存器 0（参看表 10-7 了解标准状态结果细节）

字节 1=当前磁柱号（0~79）

命令	描述	端口
Eh	清空寄存器一缓冲	3F5h

该命令从高级的控制器中返回许多内部寄存器。设计它仅仅只是为了调试控制器软件。PC 系列的 BIOS ROM 不使用该命令。

命令 字节 0=Eh

执行——读取内部寄存器结果

结果——无中断

字节 0——驱动器 0 的当前磁柱号

字节 1——驱动器 1 的当前磁柱号

字节 2——驱动器 2 的当前磁柱号

字节 3——驱动器 3 的当前磁柱号

字节 4——步进速率和磁头下载时间（参看指定命令 3#的字节 1）

字节 5——磁头装入时间和 DAM 模式选项（参看指定命令 3#的字节 2）

字节 6——当前磁柱的最后一个扇区号

字节7——垂直信息（参看垂直2.88MB模式命令#12字节1，非增强型的控制上无定义）
字节8——配置信息（参看配置命令#13的字节2）
字节9——写预补偿开始（参看配置命令#13的字节3）

命令	描述	端口
Fh	查找	3F5h

控制器将指定驱动器的磁头移动到指定的磁柱上，通常在读或写操作之前执行该动作。

在命令完成时，不返回状态字节，但是控制器会触发一个中断，Eh，来指示操作完成，在命令完成后，必须触发传感中断状态命令8和读ID命令4Ah来检查命令是否成功的完成。必须检查来自读ID命令的结果状态来确定磁头是否现在位于期望的磁柱上。

命令　字节0=Fh
字节1=驱动器选择和磁头选择字节（参看表10-6中的标准参数）
字节2=定位磁头时的磁柱号（0~79，与媒质有关）
执行——在指定的磁柱上定位磁头
结果——没有结果字节，但会触发一个中断。

命令	描述	端口
10h	版本（增强）	3F5h

该版本命令检查这是一个标准的软盘控制器还是一个增强型的控制器。NEC 7656芯片是一个标准的控制器（许多复制品都由该芯片组成）。增强型的控制器包括Intel 82077软盘控制器，它是标准控制器的一个超集。增强型的控制器包括FIFO操作，支持2.88M软盘以及其他特性。NEC 72065B也支持该版本功能，并且提供了一些增强特性（但是不支持2.88M软盘）。

尽管只在一个增强型的控制器上才有该命令的说明，但是在所有的系统上使用该版本命令是安全的。不支持版本命令的控制器总是返回无效命令码80h。PC系列的BIOS ROM不支持该命令。

命令　字节0=10h
执行——获取状态
结果——无中断
字节=80h，标准控制器
=81h，Intel 82077
=90h，NEC 72065B

命令	描述	端口
12h	垂直 2.88M 模式（增强）	3F5h

2.88M 驱动器在 2.88M 媒质上使用特殊的记录方法来双倍化 1.44M 媒质上每英寸的位数。该命令允许控制器自动地改变扇区间隙的大小和访问 2.88M 驱动器时的写计时。作为初始化的一部分，必须记录所有 2.88M 驱动器来保证该命令，将第 0 位和第 1 位都设为 1。支持 2.88M 软盘的 BIOS ROM 全使用该命令。注意必须将第 7 位置为 1 来允许改变驱动器类型的第 2 到 5 位。

设计 2.88M 驱动器要求一个单独的擦写磁头来处理其独特的垂直记录方法。所有以前的驱动器类型只能简单地用写磁头覆盖数据。2.88M 驱动器中新的擦写磁头实际上以大约 200 毫米的速度推进读/写头。在开始写时，同时打开擦写磁头。对于高密度的 2.88M 媒质，200 微米的磁头分离意味着，在写磁头开始写时，有 38 个字节没有擦除。对于格式化来说这不成问题，因为数据被写在整个磁道，但是在一个磁道的扇区内写数据时，这就是一个问题。在所有以前的媒质类型上，扇区数据之前的间隙长仅 22 字节。为了补偿这一点，在 2.88M 的媒质上扇区的长度人增加到 44 字节。第 0 位和第 1 位控制这个功能。净效益是一旦触发了该命令，对 2.88M 软驱的操作对软件就是透明的了。

命令　字节 0=12h

字节 1	位	7=0	禁止改变第 2~5 位
		1	允许改变第 2~5 位
		6=0	固定为 0
		5=1	驱动器 3 是 2.88M
		4=1	驱动器 2 是 2.88M
		3=1	驱动器 1 是 2.88M
		2=1	驱动器 0 是 2.88M
		1=0	对于 2.88M 不改变扇区间隙
		1	如果必要，对 2.88M 扩展间隙
		0=0	不改变写允许计时
		1	改变写允许计时，以允许在 2.88M 的驱动器上装入预擦除

执行——更新内部寄存器

结果——没有结果字节和中断

命令	描述	端口
13h	配置（增强）	3F5h

该配置命令在一个增强的软盘控制器上设置一系列的控制器选项。非增强型的控制器不支持它，PC 系列的 BIOS ROM 也不使用它。

在读或写操作之前，会使用隐含的查找特性来自动执行对指定扇区的查找工作。标准

的控制器不支持该增强特性。

为了同早期的软盘控制器保持兼容，禁止了FIFO，用户软件可以利用FIFO模式。

查询特性扫描每个驱动器上的驱动改变线上的变化，并在变化时触发一个中断。在装载了软驱磁头（即，正处于操作状态）时，就会进行查询。

命令 字节0=13h

字节1=0

字节2	位	7=0	未用
		6=0	无隐含查找（缺省）
		1	在读或写时隐含查找
		5=0	开放FIFO
		1	禁止FIFO（缺省）
		4=0	开放查询（缺省）
		1	禁止查询
		3=x	FIFO阈值
		2=x	0=1字节阈值
		1=x	1=2字节阈值
		0=x	F=16字节阈值
字节3	写预补偿开始磁道——指示写预补偿应开始于哪个磁道号（0~79，与媒质有关，缺省值是0）		

执行——内部记录的配置信息

结果——无结果字节和中断

命令	描述	端口
14h	开放FIFO功能（缺省）	3F5h

触发该命令来支持软件重启以清除由配置命令13h设置的FIFO当前选项。当锁被激活时，软件重启会禁止FIFO，将FIFO阈值设置到1个字节，并将写预补偿设置为开始于磁道0。PC家族的BIOS ROM不使用该命令。

命令 字节0=14h

执行——允许重启改变FIFO功能

结果——无中断

字节0=0表示接受开放命令

命令	描述	端口
34h	退出备用待机模式（仅72065）	3F5h

从备用待机模式返回软盘控制器，参看命令35h以更多地了解备用待机模式。

命令　字节 0=34h
执行——从备用待机模式返回到普通模式
结果——无中断

命令	描述	端口
34h	进入备用待机模式（仅 72065）	3F5h

软盘控制器置于低功耗的备用待机模式。软盘控制器内部时钟被停止，控制器一直等待直到返回普通模式。如果接收到了退出备用待机模式命令 34h，或者软件或硬件重启了控制器，就会返回到普通模式。

命令　字节 0=35h
执行——软盘控制器进入备用待机模式
结果——无结果字节和中断

命令	描述	端口
36h	硬件重启（仅 72065）	3F5h

重启软盘控制器，就好像发生了硬件重启一样。

命令　字节 0=36h
执行——发生了控制器重启
结果——无结果字节和中断

命令	描述	端口
42h	读磁道	3F5h

读指定磁道上的所有数据。控制器从索引孔后的第一个扇区开始读取数据，并继续下去直到读磁道的最后一个扇区。与普通的读命令 E6h 不同，读磁道不会检查保存在每一个扇区 ID 区的实际扇区号。该命令也用来从扇中读数据，这些扇区可以是正常的，也可以是标志着 ID 已删除的坏扇区。标准参数的字节 6 指定了磁道中要读取的最后一个字节。该值是要读的扇区的数目，通常是磁道上扇区的总数目。PC 系列的 BIOS ROM 不使用该命令。

命令　字节 0=42h
　　字节 1~8=标准参数（表 10-6）
执行——读磁道
结果——触发一个中断
　　字节 0~6=标准状态结果（表 10-7）

命令	描述	端口
4Ah	读ID	3F5h

该命令定位磁头所处的当前磁道。控制器从当前的磁道开始读，而不必等待索引脉冲。然后它查找扇区地址标签和定位扇区ID信息。面对第一个有效地址标签，会保存数据，并通过用读取的信息更新状态字节来完成命令。

如果出现了两个索引脉冲（即，至少两次完整的循环）不用检测有效的扇区地址标签，会在状态寄存器0中返回一个错误，如同非正常终止一样。另外，设置状态寄存器的第0位，丢失地址标签错误标志。如果未格式化磁道，则也可能出现该错误。

命令 字节0=4Ah
字节1=驱动器选择和磁头选择字节（参数表10-6中的标准参数）
执行 ——读取当前磁柱上第一个正确的ID信息，并把它放在标准状态中
结果——触发一个中断
字节0~6=标准状态结果（表10-7）

命令	描述	端口
4Dh	格式化磁道	3F5h

格式化磁道命令向软盘上写一个完整的磁道的扇区。一旦装入了该命令字节，控制器就会等待驱动器的索引脉冲，表明磁道的起点。控制器然后向磁盘写信息，包括间隙、同步脉冲、ID和数据区、地址标签，以及每个扇区的CRC区。

对于每个扇区，必须从CPU提供四个扇区ID字节，因为控制器不能生成它们，我将它们定义为执行状态下的w、x、y和z。这意味着在执行该命令之前必须设置DMA进程，并生成一个数据缓冲区来保存扇区的ID信息。对于720K驱动器上的9扇区磁道，扇区必须是36字节。

允许用户控制每个扇区的扇区ID信息，将支持不规则排列，甚至是不可能的信息，比如扇区99，当然，如果一个扇区号太古怪，可能会不能访问，或者可能只对防拷贝方案有用。使用这种非序列的扇区的一个有效理由是，为不能处理过快软盘控制器数据率的系统交叉留下一些扇区：一个9扇区软盘上的2:1的交叉扇区放置扇区的顺序是1、6、2、7、3、8、4、9、5。读取数据时，控制器每格一个跳跃扇区。PC上从未使用它，因为每个80×86的处理器都能简单地处理最高数据率。一个2.88M 1Mbps驱动器，可以传送数据速率达到每秒128K字节！所有其他的驱动器要慢二到四倍。

参看本章前面的图10-3，以便更详细地了解磁道内容，这些信息由格式化命令设置。

命令 字节0=4Dh
字节1=驱动器选择和磁头选择*
字节2=扇区大小码（2=512字节）*

字节 3=要格式的扇区数目
字节 4=扇区间的间隙长度*
字节 5=扇区内初始化数据的填充字节（所有 PC 系列软盘格式的值为 0F6h）
注：*参看表 10-6 中标准参数以获取完整的解释

执行——格式化整个磁柱，有四个字节为磁道上每个扇区指定了扇区 ID 信息。（参看文中所述）

字节 w=磁柱号
字节 x=磁头号（0=第 0 面，1=第 1 面）
字节 y=扇区号（通常 1~36）
字节 z=扇区大小码（2=512 字节）

结果——触发一个中断

字节 0=状态寄存器 0（参看表 10-7 中标准状态结果）
字节 1=状态寄存器 1（参看表 10-7 中标准状态结果）
字节 2=状态寄存器 1（参看表 10-7 中标准状态结果）
字节 3=未定义
字节 4=未定义
字节 5=未定义
字节 6=未定义

命令	描述	端口
8Fh	相对向外查找（增强）	3F5h

该命令与普通的查找命令非常相似，它指定了向外边界移动的磁柱的数目，相对于当前的位置。参看查找命令 Fh 以获取深入细节。它为将来的软盘而设计，它们可能有多于 256 个磁道。任何 PC 系列的 BIOS ROM 都不使用该命令。也可以参考相对向内查找命令 CFh。

命令 字节 0=8Fh
字节 1=驱动器选择和磁头选择字节（参看表 10-6 中的标准参数）
字节 2=从当前磁柱向外移动到较低编号的磁柱上的磁柱数

执行——向外移动指定的磁柱数

结果——没有结果字节，但是会触发一个中断

命令	描述	端口
94h	锁住 FIFO 功能（增强）	3F5h

该命令阻止软件重启改变当前 FIFO 选项，该选项由配置命令 13h 设置。如果激活了锁，软件重启将不会改变当前 FIFO 状态（开放或禁止），FIFO 阈值，或写预偿值，PC 系列 BIOS ROM 不使用该命令。

命令 字节 0=94h
执行 ——阻止重启改变 FIFO 功能
结果——无中断

命令	描述	端口
C5h	写数据	3F5h

BIOS 软盘中断服务使用该命令来向磁盘写数据。写动作首先必须使用 DMA 通道 2 为传送设置 DMA 控制器。而后，发一个查找命令来将磁头定位到正确的磁道上。

在该命令的执行期间，装载磁头，在装载定时期满后，从磁头读取数据。如果来自软盘扇区的扇区地址与装载参数中指定的扇区相匹配，就开始 DMA 传送。在写了一个完整的扇区之后，如果 DMA 的终端计数线尚未过期，那么就会增加控制器的扇区号，并通过 DMA 进程传送下一个扇区。数据被写到指定的扇区，地址标签 ID 表示它是一个“正常数据”。写进程也确定扇区的 CRC 并将之写入到 CRC 区。

该命令提供多磁道选项，带有多扇区的多磁道传送会自动地从磁头 0 切换到磁头 1，并继续向第一面写扇区。这意味着一次触发该命令可写的最大扇区数将是每磁道扇区数的两倍。对于一个每磁道有 18 扇区的 1.44M 的软盘来说，同时可以读取 36 个扇区，也就是 18432 个字节（36 扇区乘以 512 字节每扇区）。

命令 字节 0=C5h
字节 1 到 8=标准参数（表 10-6）
执行——写数据，并标出“普通数据”的标识
结果——触发一个中断
字节 0 到 6=标准状态结果（表 10-7）

命令	描述	端口
C6h	读正常和已被删除的数据	3F5h

该命令与读正常数据命令 E6h 相同，只是控制器会读所有的扇区，包括那些标注着“已被删除的数据”和“正常数据”ID 的扇区。参看读正常数据命令以获取其他细节。PC 系列的BIOS ROM 不使用该命令。

命令 字节 0=C6h
字节 1 到 8=标准参数（表达 10-6）
执行——从扇区读数据
结果——触发一个中断
字节 0 到 6=标准状态结果（表 10-7）

命令	描述	端口
C9h	写已被删除的数据	3F5h

该命令与写数据命令 C5h 相同，只是控制器将每个扇区的 ID 标注为“已被删除的数据”。参看写数据命令获取另外的细节。PC 系列的 BIOS ROM 不使用该命令。

该命令故意将扇区标注成有错误的扇区，这样普通的读就会忽略它们，有时防拷贝方案用它来隐藏软盘上的数据，因为 BIOS 只会使用那些控制命令，它们会跳过标注着“已被删除的数据”ID 的扇区。

命令 字节 0=C9h
字节 1 到 8=标准参数（表 10-6）
执行——写数据，并带有“已被删除的数据”ID
结果——触发一个中断
字节 0 到 6=标准状态结果（表 10-7）

命令	描述	端口
CCh	读已被删除的数据	3F5h

该命令和读正常数据命令 E6h，只是控制器只读标注着“已被删除的数据”ID 的扇区。该命令跳过任何标注着“正常数据”的扇区。参看读取普通数据命令以获取其他细节。PC 系列的 BIOS ROM 不使用该命令。

命令 字节 0=CCh
字节 1 到 8=标准参数（表 10-6）
执行——从带有已被删除数据 ID 的扇区读取数据
结果——触发一个中断
字节 0 到 6=标准状态结果（表 10-7）

命令	描述	端口
CFh	相对向内查找一磁盘	3F5h

该命令与普通的查找命令很相似，它用来指定和磁柱中心移动的磁柱数目。相对于当前位置。参看查找命令 Fh 以深入了解细节。它特意为将来的多于 256 磁道的软盘而设计。PC 系列的 BIOS ROM 不使用它。也参看命令 8Fh 以了解相对向外查找。

命令 字节 0=CFh
字节 1=驱动器选择和磁头选择（参看表 10-6 中的标准参数）
字节 2=从当前磁柱向内到更高的磁柱所移动的磁柱数目。

执行——向内移动指定的磁柱
结果——无结果字节，但触发一个中断

命令	描述	端口
D1h	等待扫描等值比较	3F5h

控制器逐字节核对来自系统的数据和记录在软盘上的数据。如果软盘上的数据与缓冲区内容相匹配，则核对为真。如果发现第一个软盘字节的值比缓冲区的相关字节小，则会停止其他几个扫描命令。这个功能并不十分有用，如果你对此可任选命令感兴趣，请参看软盘控制器的硬件参考书籍。

会读取所有的扇区而不管其 ID 类型（“已被删除的数据”或“普通数据”）。PC 系列的 BIOS ROM 不使用扫描等值比较以及其他的扫描命令。参看相类似的命令 F1 来跳过标注着“已被删除的数据”ID 的扇区。

命令 字节 0=D1h
字节 1 到8=标准参数（表 10-6）
执行——扫描，直到匹配失败或完成
结果——无中断
字节 0 到 6=标准状态结果（表 10-7）
寄存器 2 的第 3 位和第 2 位含有扫描的状态。第 3 位和第 2 位的值各自为 01 表示所有的数据相等，而值 10 表示不匹配。

命令	描述	端口
D6h	检查一扇区	3F5h

检查指定扇区在的信息，看有没有错误。从每个扇区读取数据并确定一个新的 CRC，然后将它和读自软盘的 CRC 值相比较。如果不匹配，则检查失败。该命令包括所有标注着“已删除数据”ID 的扇区。PC 系列的 BIOS ROM 不使用该命令。

命令 字节 0=D6h
字节 1 到8=表 10-6 的标准参数，只是除了下述：
如果驱动器选择和磁头选择字节的第 7 位=1，那么用户扇区大小字节等于要核对的扇区数。
执行——不发生系统数据的传送
结果——触发一个中断
字节 0 到 6=标准状态结果（表 10-7）

命令	描述	端口
E6h	读普通数据	3F5h

BIOS 软盘中断服务使用该命令来从软盘上读取数据。读操作必须先设置读时的 DMA 控制器，使用 DMA 通道 2，然后触发一个查找命令来将磁头定位到正确的磁道上。

在读普通数据的命令执行期间，会装载磁头，并在磁头装载定时期满之后，从磁道读取数据。如果来自软盘扇区地址和装入参数中指定的扇区相匹配，就会开始 DMA 传送。传送了一扇区数据后，如果 DAM 的终止线还未过期，控制器就会增加其扇区号，并通过 DMA 进程传送下一个扇区。任何标注有“已删除数据”ID 的扇区将被忽略。

该命令使用多磁道选项，带有许多扇区传输的多磁道会自动地从磁头 0 切换到磁头 1，并继续从第 1 面读取数据。这意味着一次触发该命令所能读取的最大扇区数是每磁道扇区数的两倍。对于一个 2.88M 软盘，每磁道有 36 个扇区，可以一次读取总共 72 个扇区，合 36864 个字节（72 扇区乘以 512 字节每扇区）。

命令　字节 0=E6h
　　字节 1 到8=标准参数（表 10-6）
执行——从扇区读数据
结果——触发一个中断
　　字节 0 到 6=标准状态结果（表 10-7）

命令	描述	端口
F1h	扫描等值比较	3F5h

控制器逐字节核对来自系统的数据和记录在软盘上的数据。如果软盘上的数据与缓冲区内容相匹配，则核对为真。如果发现第一个软盘字节的值比缓冲区的相关字节小，则会停止其他几个扫描命令。这个功能并不十分有用，如果你对此可任选命令感兴趣，请参看软盘控制器的硬件参考书籍。

任何标注有“已被删除的数据”ID 的扇区将被忽略。PC 系列的 BIOS ROM 不使用扫描待值比较。参看相类似 D1h 来包括那些标注有“已被删除的数据”ID 的扇区。

命令　字节 0=F1h
　　字节 1 到 8=标准参数（表 10-6）
执行——扫描，直到匹配失败或完成
结果——无中断
　　字节 0 到 6=标准状态结果（表 10-7）
　　寄存器 2 的第 3 位和第 2 位含有扫描的状态。第 3 位和第 2 位的值各自为 01 表示所有的数据相等，而值 10 表示不匹配。

命令	描述	端口
F6h	标准[illegible]	3F5h

检查指定扇区的信息，看有没有错误。从每个扇区读取数据并确定一个新的 CRC，然后将它和读自软盘的 CRC 值相比较。如果不匹配，则检查失败。该核对命令忽略所有标注着“已删除数据”ID 的扇区。PC 系列的 BIOS ROM 不使用该命令。

命令 字节 0=F6h
字节 1 到8=表 10-6 的标准参数，只是除了下述：
如果驱动器选择和磁头选择字节的第 7 位=1，那么用户扇区大小字节等于要核对的扇区数。
执行——不发生系统数据的传送
结果——触发一个中断
字节 0 到 6=标准状态结果（表 10-7）

端口	类型	描述	平台
3F6h	I/O	软盘控制器保留	所有

没有使用这个寄存器地址，为将来可能有的软盘控制器特性而保留。

端口	类型	描述	平台
3F7h	输入	软盘控制器数字输入寄存器	AT+

该数字输入寄存器显示了当前选中的驱动器的软盘改变线的状态。所有的 3.5"软驱和 1.2M 5.25"驱动器的软盘改变线。该线指示了一张新软盘插入了驱动器中，或者驱动器中无软盘。

这个端口和硬盘控制器共享，PS/2 系统除外。如果存在硬盘控制器，硬盘控制器就会在低 6 位中返回信息。由于软盘控制器只返回第 7 位，所以我将它们显示为“未使用，三态”。这意味着，如果没有使用硬盘适配卡，这些未使用的位可能会返回任意值。

AT+系统（PS/2 AT 和 MCA 总线系统除外）
输入（第 7 位）

位	7 r=0	存在软盘，没有改变软盘
	1	不存在或改变了软盘
	6 r=x	
	5 r=x	未使用，三态（参看文中所述）
	4 r=x	
	3 r=x	
	2 r=x	
	1 r=x	
	0 r=x	

仅 PS/2 MCA 总线
输入（第 0~7 位）

位 7 r=0 存在软盘，没有改变软盘

1 不存在或改变了软盘

6 r=1 固定为 1

5 r=1 固定为 1

4 r=1 固定为 1

3 r=1 固定为 1

2 r=x | 1 r=x 数据率选择

第 2 位	第 1 位	速率
0	0=	500Kbps
0	1=	300Kbps（并非所有的样式都支持）
1	0=	250Kbps
1	1=	1Mbps（并非所有的样式都支持）

0r=0 选择高密度数据率（500Kbps 或 1Mbps）

1 选择低密度数据率（250Kbps 或 300Kbps）（硬件重启后的缺省值）

仅 PS/2 AT 总线（例如，样式 30 或其他）

输入（第 0~7 位）

位 7 r=0 存在软盘，没有改变软盘

1 不存在或改变了软盘

6 r=0 固定为 0

5 r=0 固定为 0

4 r=0 固定为 0

3 r=0 禁止 DMA 和中断功能（参看端口 3F2h）

1 开放 DMA 和中断功能

2 r=x 没有来自配置控制器，端口 3F7h 第 2 位的预补偿位状态（当前未用作任何用途!）

1 r=x | 0 r=x 数据率选择

第 2 位	第 1 位	速率
0	0=	500Kbps
0	1=	300Kbps（并非所有的样式都支持）
1	0=	250Kbps
1	1=	1Mbps（并非所有的样式都支持）

端口	类型	描述	平台
3F7h	输出	软盘控制器配置控制器	AT+

这个寄存器用于设置当前的数据率，在除 PS/2 以外的所有系统上这个寄存器是只写的。在 PS/2 上，寄存器是可读的，这与总线类型有关。参看数字输入寄存器，以了解读回数据率选择的确切细节。

AT+和 PS/2 MCA 总线系统（PS/2 AT 总线系统除外）

输入（第 0~1 位）

位

7 w=x 未使用
6 w=x 未使用
5 w=x 未使用
4 w=x 未使用
3 w=x 未使用
2 w=x 未使用
1 w=x
0 w=x 数据率选择

第 2 位	第 1 位	速率
0	0=	500Kbps
0	1=	300Kbps（并非所有的样式都支持）
1	0=	250Kbps
1	1=	1Mbps（并非所有的样式都支持）

仅 PS/2 AT 总线（即类型 30 以及其他）

输出（第 0~2 位）

位

7 w=x 未使用
6 w=x 未使用
5 w=x 未使用
4 w=x 未使用
3 w=x 未使用
2 w=x 无预补偿位——仅仅只是从数字输出寄存器，端口 3F7h，读取位。只有硬件重启才能清除该位。
1 w=x
0 w=x

第 2 位	第 1 位	速率
0	0=	500Kbps
0	1=	300Kbps（并非所有的样式都支持）
1	0=	250Kbps
1	1=	1Mbps（并非所有的样式都支持）

第 11 章

硬盘系统

这是本书中最长的章节之一，但是它仍未全部涵盖今天使用的所有硬盘系统。本章将讨论的重点放在系统 BIOS 所支持的驱动器和控制器上，而只是简单地讨论了一下那些非标准的控制器，后者用自己的磁盘 BIOS 替代了系统 BIOS 服务。

在仔细查看其他技术文档时，我惊讶地发现这些文档存在着一些错误，并且忽视了今天各种标准之间细微但是非常重要的差别。

我花费了许多时间来查看不同供应商所提供的 BIOS 代码和硬件控制器，以确保本章后面所提供的 BIOS 信息是可靠的。另外，一个称为 DISKTEST 的程序可以标识出某些 BIOS 错误，这些错误是我在测试不同的系统时偶然遇到的。

如果你想彻底地理解磁盘系统，那么本章有许多信息都对你很重要。当然，BIOS 中断部分对那些希望开发高级代码的人来说尤为重要。本部分包括一系列错误返回码，并对这些返回码做出了解释，包括解释错误的起因，紧接着描述 BIOS 中断功能，其中包括大量以前尚未说明的细节和功能。

那些需要设置系统 BIOS 的勇敢的人，会发现这里全面地描述了 BIOS 是如何对控制器进行操作的，这其中也包括对一系列 I/O 端口的详细描述。设置 BIOS 时要格外小心，因为不同的供应商有不同的未公开的信息。

在这一章中讨论 PS/2 时，我认为它们都使用了一个 ESDI 控制器和驱动器。这一点并不很严格。在 PS/2 生产的最后两年，各种不同的型号都开始使用 IDE 和 SCSI 驱动器。既然基于 PS/2 MCA 的计算机线没有继续生产下去，有关软件是否还支持这些机器的问题就并不是重要了。我已经为那些需要支持这些系统的用户详细地介绍了 PS/2 的 ESDI 接口。

简　介

跟软盘驱动器一样，硬盘也有许多旋转方式来记录和返回二进制信号。在开始写或读数据时，可以直接将驱动器定位到指定的点上。大多数硬盘驱动器都是基于磁性媒质的，当然也有些是支持光媒质的。

目前大多数系统包括了至少一个硬盘驱动器，甚至是那些无盘工作站也将网络服务器作为用户的一个硬盘驱动器。所有 AT-、EISA-和 MCA-类型的 BIOS 都支持多达四个硬盘

驱动器。PC 和 XT 使用适配卡上的 BIOS，这个 BIOS 与系统 BIOS 无关。大多数系统，即使看起来好像有两个以上的驱动器，其实也只是操作系统将多个逻辑盘映射到了一个或两个硬盘上，或者是某种特殊的磁盘适配卡替代了系统 BIOS 的磁盘功能来支持多个驱动器。

与第 10 章软盘系统类似，也有许多参考书涉及到与硬盘有关的通用术语。每个硬盘被分为磁头、磁柱、磁道和扇区，如图 11-1 所示。最小的组成部分是扇区，有 512 个字节。许多扇区组成一个磁道。一个磁柱是垂直的一组磁道，通常是盘面数的两倍。

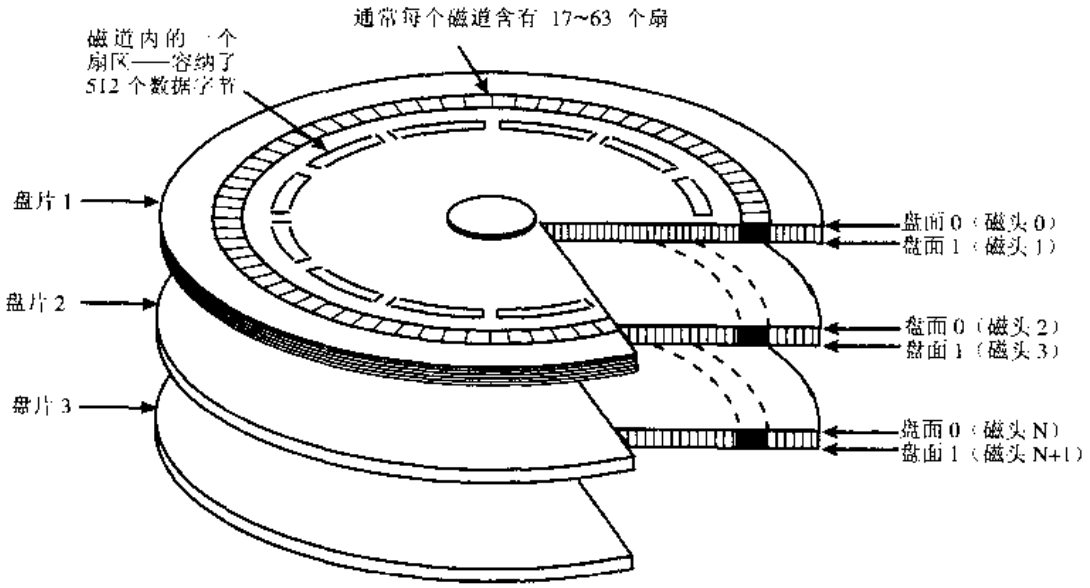


图 11-1　硬盘媒质的结构

硬盘可由下述组合来定义：每磁道的扇区数，每磁柱的磁道数以及磁柱数，通常由驱动器制造商来设置这些数据。例如，一个 XT 使用的 10MB 驱动器每磁道有 17 扇区、306 磁柱以及 4 个磁头：

（512 字节每扇区）×（17 扇区每磁道）×（306 磁柱）×（4 个磁头）=10,653,696 字节。

与老式的 MFM 驱动器不同，目前大多数驱动器会屏蔽驱动器的实际分层结构，指出这一点很重要。但是从软件的视角来看，这一点并不重要。软件总是认为驱动器有固定数目的每磁道扇区数、磁头数以及磁柱数。大多数驱动器有一个内置的转化器，用来将软件所提供的结构信息翻译成内部的驱动器映射。有时候某些翻译做得相当复杂，它们在驱动器上保留了一些扇区和磁柱来修正媒质缺陷，当然这一点对软件来说是不可见的。某些高容量的驱动器也采用一些方法来在外围磁柱上的每磁道获得更多的扇区数，而在内部磁柱上每磁道使用较少的扇区数。这样做可以提高驱动器的容量，因为它在每个磁道上放置了最大可能的位密度。所有的这些映射对访问驱动器的软件来说都是不可见的。

在访问硬盘驱动器时，程序与操作系统通信。由程序指定文件名，由操作系统确定该文件在硬盘上的物理位置。老式的操作系统，比如 DOS，就会使用 BIOS 服务的中断 13h 来执行低级工作，然后硬盘 BIOS 和硬盘控制器通信。最后控制器和驱动器进行通信来读

写所请求的数据。

图 11-2 显示了一个基本的硬盘子系统。该图指示了早期的 PC 和 XT 标准，通常称作 ST-506。尽管这个标准在今天已经过时，但大多数的系统 BIOS 仍采用这种方法来访问控制器。随着驱动器在伺服控制方面的进步，位置信息被嵌入到了驱动器中，并且内部的控制器到驱动器的功能，因供应商的差别而有很大的不同。幸运的是，一旦我们有了工作的概念模式之后，我们程序员很少需要考虑控制器以外的事情。

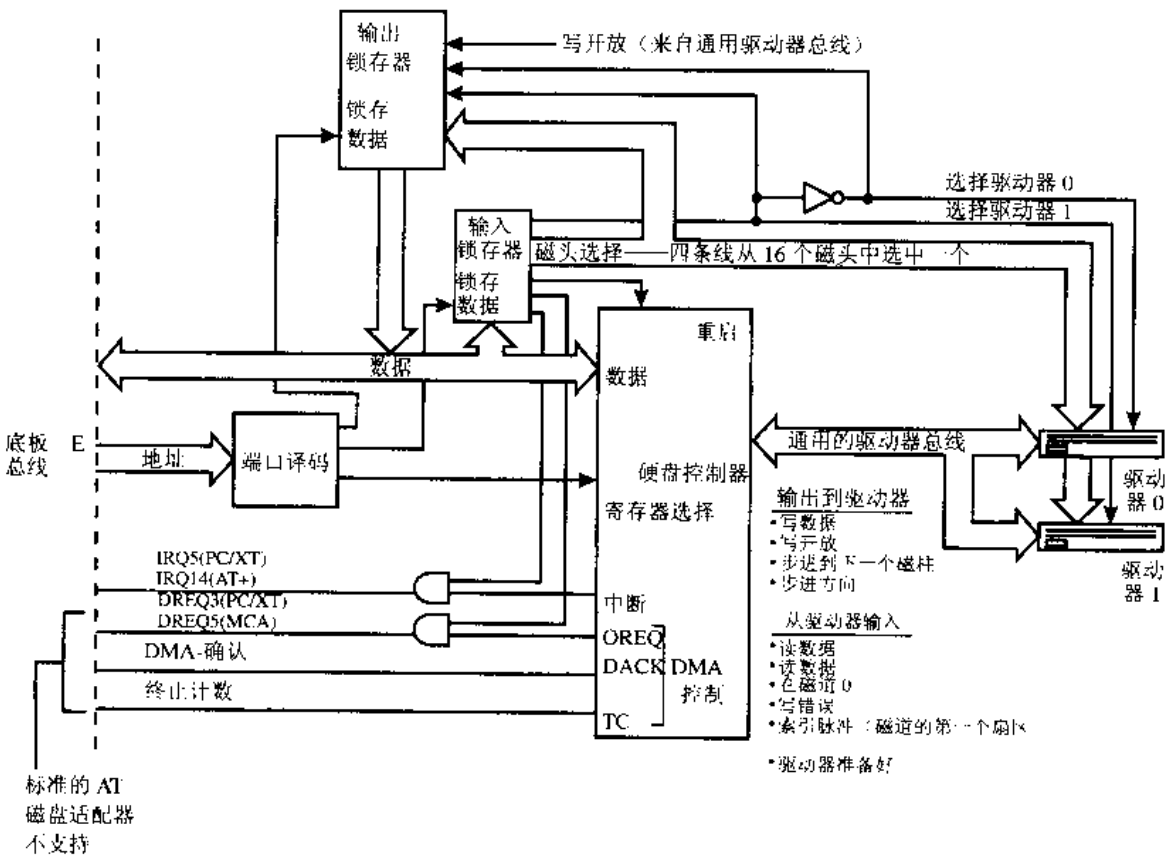


图 11-2　硬盘子系统

是否会给出实际的硬盘大小

硬盘驱动器制造商过去常常误导用户，使他们相信硬盘驱动器可以有比实际更大的数据容量。20 世纪 80 年代最常见的问题是，供应商只会提供驱动器的未格式化容量。当然，你只有在操作系统格式化了一个驱动器之后才能使用它。格式化进程定义了每个扇区和一些标识信息，这样就可以定位磁盘上的指定扇区。还需要一些位来标识间隙、扇区标识信息以及错误检查更正码（ECC）。这意味着格式化的容量比未格式化的容量要少 5%~10%。

那么在什么时候一个 200MB 格式化后的驱动器不等于 200 兆字节呢？大多数驱动器供应商现在指出了驱动器格式化后的大小，但是某些供应商会使用一些花招来使驱动器大小

显得比你在 PC 中实际得到的要多。大多数常见的花招是使用非标准的或更大的扇区大小。例如，200MB 格式化驱动器容量可能会指定为 2,048 字节的扇区，而每个 PC 平台要求 512 字节的扇区。每个扇区很容易获得 50 或更多的头字节，但是使用 2,048 字节扇区，硬盘容量可能会增加 5%~10%。图 11-3 有助于解释更大的扇区字节大小将减少所要求的硬盘空间。

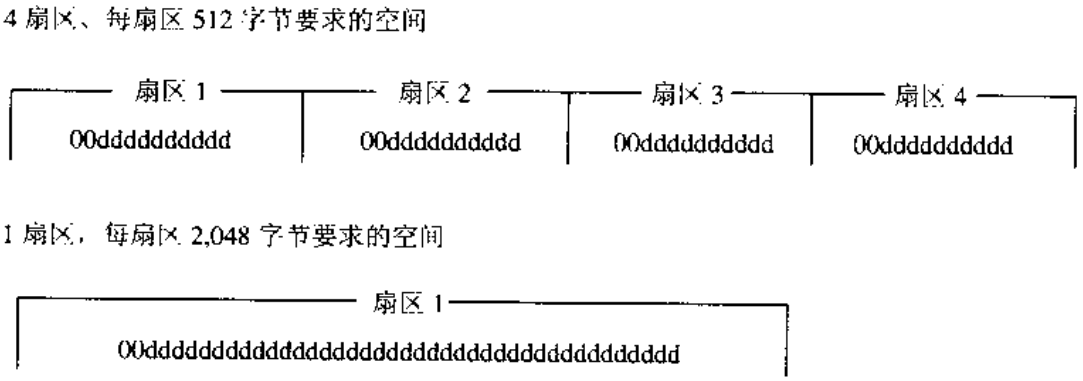


图 11-3 硬盘容量和扇区大小

另一个使空间看起来更大些的办法是，在磁盘容量数据中包括一些你不会访问的空余扇区。SCSI 驱动器以及某些其他的驱动器类型包括了空余的扇区和空余的磁柱。制造商用这些空间来替代磁盘上有缺陷的扇区。这些保留空间的字节数很少被公开过。

留心大于 512MB 的驱动器。所有 1995 年以前的老式 PC 和某些操作系统不能处理 1,024 个以上的磁柱。通常 512MB 以上的容量要求 1,024 个以上的磁柱。过多的磁柱在一个 PC 系统上是不可访问的，不能当成可用的容量。这意味着一个带有 1,080 磁柱 540MB 容量的驱动器只能使用 1,024 个磁柱——可能容量为 512MB。通常 BIOS 也保留了一个或两个磁柱用于诊断，这会进一步减少驱动器可用的容量，在这种情况下，驱动器制造商还算公道，但是系统仍会减少可用的空间。用可处理这些较大驱动器的 BIOS 和操作系统接口替代原来的 BIOS 和操作系统接口，新式的 PC 和控制器解决了这个问题。处理大型 IDE 驱动器的标准方法称作逻辑块访问（LBA）。后来的 LBA 能处理更大的容量。

驱动器行业中的这些方法非常常见，这意味着从制造商那里获得精确的可用空间容量几乎是不可能的，除非它明确指出该格式化后的大小用于标准的 DOS PC 平台。由于大多数的制造商都采用这些手段，所以在比较驱动器时，你应考虑到这个数据与可用容量有 10% 的差距。公平地讲，驱动器行业总是显示最佳的大小，尽管它们很少在 PC 平台上实现，但是在另外一些系统，如 UNIX 上却是可能的。竞争竟是如此地有意思！

接口标准和控制器

表 11-1 中列出了常见的接口标准。这些标准定义了控制器和驱动器之间的硬件连接和信号，每个标准要求有该驱动器类型专用的控制器。这就意味着每个硬盘控制器卡只能处理相同类型的磁盘。例如，你不能将一个 SCSI 驱动器和 IDE 驱动器连接到相同的硬盘控

制器上，即使控制器可能支持这两个驱动器。

今天，ST-506/ST-512 接口已经过时了，但是会存在于过去遗留下的机器上。ESDI 接口（ST-506 的增强型）今天也过时了。对于 PC 平台来说，IDE 和 EIDE 驱动器完全占据了所有用户系统市场。SCSI 仍是多驱服务器和多任务操作系统的首选。

表 11-1　常见的驱动器接口标准

ST-506/ST-512	希捷技术 506 接口
ESDI	增强型小型设备接口
IDE	集成驱动器电子（使用 AT 附件标准）
EIDE	增强型 IDE
SCSI I 和 II	小型计算机系统接口

对于 PC 和 XT 系统来说，大多数类型的控制器只支持 ST-506/ST-412 接口标准。适配卡支持硬盘 BIOS，因为很少有系统带有提供硬盘服务的系统 BIOS。这些控制器称为 PC/XT 控制器。所有的 PC/XT 控制器都使用 8 位数据通道。PC/XT 控制器建立了基本的中断 13h BIOS 接口，今天大多数控制器仍在使用该接口的一个子集。较低层次的到控制器的 I/O 是 PC/XT 系统所特有的，AT 在此基础上做了较大修改。

至于与驱动器通信的操作系统，大多数 AT 和 EISA 系统使用内置的系统 BIOS 硬盘服务。这些服务很好地定义了到控制器的通信通道。我将这些类型称作 AT 控制器，因为这是目前最流行的使用标准。AT 控制器有许多类型，支持表 11-1 中的 ESDI、IDE、EIDI 和 SCSI 标准。AT 总线控制器是基于 16 位数据通道设计的。使用 VLB 或 PCI 总线的 AT 控制器通常支持 32 位数据。目前，EIDE 是控制器到驱动器最流行的接口标准。所有的 AT 控制器都使用相同的 I/O 接口，本章后面有详细叙述。

老式的 PS/2 BIOS 只支持 ESDI 控制器上的 ESDI 驱动器。这些 PS/2 BIOS 不支持任何其他类型的控制器和驱动器。微通道和 AT 总线的 PS/2 系统更是如此。在本章中我把 PS/2 接口称为 PS/2 ESDI 控制器。记住，即使控制器和驱动器之间的实际接口是相同的，AT 系统上的 ESDI 驱动器也不使用 PS/2 ESDI 设计。如果带上另外一些硬件，并用硬盘 BIOS 替代系统 BIOS 的磁盘处理程序，PS/2 可能会支持其他的标准。

PS/1 是一个 AT 类型的系统，它使用 AT 类型的控制器。除了 PS/1 有一些特有的扩展之外，PS/1 的控制器可以被当作标准的 AT 控制器的扩充。

SCSI 接口支持附带七个设备的单个控制器。因为标准的系统 BIOS 只支持对两个通用设备的操作，所以大多数的 SCSI 适配器包括有一个驱动文件或 BIOS ROM，在与 SCSI 驱动器通信时替代系统 BIOS 服务。这样就可支持所有七个可能的设备。对于 PC、XT、EISA 和 PS/2 更是如此。一些 SCSI 适配器使用 AT 的 BIOS 软件，而限制使用它自己的 BIOS ROM。“无 ROM”的 SCSI 适配器最多可支持两个设备。

因为目前使用的接口标准范围很广，所以我建议不要直接写控制器，当然一些特殊情况除外。如果知道了驱动器接口类型，为了满足高速要求，最好直接同控制器通信。Windows

3.11 和 Windows 95 就是很好的例子。Windows 提供了快速磁盘选项来绕过 BIOS 硬盘处理程序而使用它自己更快的 32 位代码。在这种情况下，如果没有专门的驱动器，原来的控制器就不会支持其他的接口，例如 SCSI 或 PS/2 ESDI。当然，在 PC 和 XT 系统上替代硬盘 BIOS 没有多大意义，因为它们的 CPU 不支持 32 位代码。

AT 控制器接口的最大优点是它很简单，不需要专用设备驱动程序或 BIOS ROM 来浪费内存空间，每个目前或将来运行在 AT 平台上的操作系统都会正常工作。对于其他接口来说，控制器生产商或操作系统供应商必须为驱动器接口提供专门的支持。许多情况下，对替代的控制器的支持可能永远也不会实现，或者为了支持一个非 AT 控制器的系统往往要花费几个月甚至几年的时间。

既然每个人应该使用 AT 标准驱动器接口这一点是很显然的，那么为什么并非所有人都使用它呢？这与我们没完没了地追求更好的速度和性能有关。对于带单个驱动器的系统来说，AT 控制器会提供最好的系统运行速度。另外，IDE 接口安装起来比较简单，比 SCSI 也便宜很多。另一方面，如果在服务器或多任务操作系统上使用多个驱动器，那么设计良好的 SCSI 接口会极大地提高运行速度，因为 SCSI 适配器可以同时执行多个驱动器。对于一个两驱的 IDE 系统来说，一次只能访问一个驱动器。

如果要求多个驱动器，某些 SCSI 可用来提供多达 15 个设备的支持。目前，大多数系统只局限于四个 IDE 设备，当然有些适配器可以在一个系统上支持多达八个 IDE 设备。对于 SCSI 和 IDE 来说，设备可以是下列组合：磁盘驱动器，磁带复制系统、CD 驱动器以及其他硬件。

最后需要指出的是，IDE、ESDI 和 SCSI 驱动器会对用户屏蔽其内部的磁盘层次结构。在驱动器容量内的任何磁头、磁柱以及扇区的组合都是有效的。这一点使安装相当简单。控制器会将虚拟的映射转化成实际的内部驱动器映射。所有这些都由系统做透明处理。

驱动器操作

基本的磁盘驱动器是一个相当简单的设备。驱动器有许多旋转盘面，磁头围绕磁柱运动。使用驱动器内的电路来选择激活哪个磁头。一次只能激活一个磁头。当然，实际上也采用了一些策略来使驱动器可靠、迅速和容纳更多的数据——但这是其他人的工作。

数据以串流的形式保存在磁盘的每个磁道上。当控制器读磁盘时，控制器必须区分不同的间隙、扇区标识信息、错误检查更正（ECC）字节以及实际的数据。写数据时，控制器将磁头移动到合适的扇区位置，然后扇区字节转化为串流并发送到驱动器。控制器完全定义了每个磁道上信息的确切格式。

ST-506 类型驱动器

驱动器提供给控制器很有限的位置信息。在每次旋转时，驱动器指示控制器磁头位于磁道的起点。这是一个索引脉冲。另外，如果磁头位于磁柱 0 时，就会向控制器发送另一个信号。

为了将磁头定位在磁柱上，控制器向步进线发一个脉冲来将磁头移动一个磁柱。方向线指示磁头是向内移动还是向外移动。控制器必须跟踪在每个方向上移动的步数，这样控制器就可以知道磁头位于哪个磁柱上。如果位置未知，比如上电时，控制器会执行一次重校操作。重校只是简单地发送步进脉冲，选择的方向向外。如果驱动器指示磁头到达磁柱 0，就会停止脉冲，这样控制器就知道了磁头的位置。

为了在磁道内找到要读或要写的扇区，控制器读取每个扇区的扇区 ID 域，直到与所期望的扇区相匹配。在这一点继续读或开始写。

使用 IDE、EIDE、ESDI 和 SCSI 标准的驱动器

大多数较新的驱动器上带有控制器，用来在磁盘上定位磁头的方法由供应商指定。最常见的方法是使用伺服机制来保持精确的位置信息。可以认为驱动器是一个长系列的扇区。为了选择一个新位置，控制器简单地将磁柱、磁头号以及扇区转化为相对的扇区号。然后，驱动器就定位到相对的扇区号，并开始读或写操作。

大型的 IDE 驱动器

标准的中断 13h 磁盘服务和 IDE 驱动器接口有不同的磁柱、磁头和扇区限值，如表 11-2 所示。BIOS 和 IDE 标准之间的最小值作为最大结果值。表 11-2 也指出基于 BIOS 限、IDE 限和结果限的最大可能的驱动器大小。这就是那个令人讨厌的 540MB 驱动器上限的来源。

表 11-2　BIOS 和 IDE 驱动器限值

	最大 BIOS 值	最大 IDE 值	最大结果值
磁柱	1,024	65,536	1,024
磁头	256	16	16
扇区	63	255	63
最大驱动器	8GB	130GB	.5GB(504MB)

为了超过 540MB 限值，同时所有支持的软件和操作系统不必修改就能够工作，人们设

计了一种新方法，称作逻辑块寻址（Logic Block Adressing，LBA）。它使用标准 BIOS 限，并翻译磁柱/磁头/扇区地址以适应 IDE 驱动器接口的要求。例如，1GB IDE 驱动器有 2,048 个磁柱，16 个磁头和 63 个扇区。激活 LBA 时，这个驱动器对软件来说就好像有 1,024 个磁柱，32 个磁头和 63 个扇区。这种翻译技术标准的中断 13h 服务程序可以访问多达 8GB 的数据。使用中断 13h 服务的软件不必知道 BIOS 所执行的翻译操作以及何时将 LBA 地址直接传递给 IDE 驱动器。大多数 1995 年或以后的系统 BIOS 都支持 LBA。

为了超越 8GB 限值（SCSI 除外），人们又设计了一种新的标准，称作增强 BIOS（EBIOS）标准。它提供了一系列全新的中断 13h 服务（开始于功能 41h）。EBIOS 服务支持访问的 IDE 驱动器大小可达 138GB。EBIOS 服务也支持将来的驱动器标准，访问的驱动器大小可达 2,048GB。因为存在向后兼容问题，必须重写软件来使用这些新标准。只有一些操作系统，像 Windows 95 才支持系统带有 EBIOS 服务，不过将来会有更多的操作系统支持它。

磁盘参数表

有两个硬盘参数表保存了系统上每个硬盘驱动器的信息。BIOS 使用该表来对磁盘控制器编程和指定控制驱动器时的各种定时。指向驱动器 0 的参数表的指针保存在中断 41h 的地址，即 0:104h 处。驱动器 1 的参数表的指针保存在中断 46h 的地址，0:118h 中。习惯上将磁盘在数表中所有的未用字节都置 0。表 11-3 显示了磁盘参数表的内容。

表 11-3 磁盘参数表的内容

偏移量	大 小	描 述	
0	字	磁柱、驱动器上的最大磁柱号	
2	字节	磁头、驱动器上的最大磁头号，不越限	
		限值	控制器类型
		8	PC/XT 控制器
		16	AT 控制器和系统 BIOS（无 LBA）
		32	PS/2 ESDI 控制器
		255	IDE 和 EIED 控制器，带 LBA
3	字	为了减少写电流的起点磁柱（仅 XT，所有其他系统都不支持）	
5	字节	写预补偿的起点磁柱——用它是为了减少发送到驱动器的数据中的短延时，在需要写预偿的驱动器上，可以在从指定的磁柱到最里面的磁柱上激活它，通常是驱动器上的半数磁柱。所有 IDE 和 EIDE 类型的驱动器都不需要任何预补偿，并忽略该值。任何比最大磁数大的值都会阻止写预补偿	
7	字节	最大 ECC 数据长度（仅 XT，所有其他系统都不使用）	

续表

偏移量	大　小	描　　述					
8	字节	控制字节（AT 以及所有以后的系统）					
		位	7=x	操作失败后重试			
			6=x	位 7	位 6		
				0	0=出错时重试		
				0	1=不重试		
				1	0=不重试		
				1	1=不重试		
			5=1	磁盘制造商的缺陷映射驻留在最大磁柱号+1 的磁柱上（某些控制器和 BIOS 会忽略它）			
			4=0	未使用			
			3=1	驱动器的磁头数多于 8（某些控制器和 BIOS 会忽略它）			
			2=0	必须固定为 0（无重设）			
			1=0	必须固定为 0（禁止 IRQ）			
			0=0	未使用			
		控制字节（仅 XT）					
			7=0	如果访问磁盘有问题，就重试四次			
			1	访问磁盘有问题时禁止重试			
			6=1	读的过程如果出现 ECC，则禁止一般的重试操作			
			5=0	未使用			
			4=0	未使用			
			3=0	未使用			
			2=x	驱动器步进速度			
			1=x	位 2	位 1	位 0	步进率
			0=x	0	0	0	=3ms
				1	0	0	=200ms
				1	0	1	=70ms（正常情况）
				1	1	0	=3ms
				1	1	1	=3ms
9	字节	普通超时值（仅 XT，其他所有系统都未使用）					
A	字节	模式化磁盘驱动器的超时值（仅 XT，其他所有系统都未使用）					
B	字节	检查磁盘驱动器的超时值（仅 XT，其他所有系统都未使用）					
C	字节	装入带磁柱（仅 AT 以及后来的系统使用，XT 不使用）					
E	字节	每磁道的扇区数（仅 AT 以及后来的系统使用，XT 不使用）					
F	字节	未使用					

可以提供一个可替代的磁盘参数表作为增强型磁盘驱动器说明的一部分。它打破了EIDE驱动器的8GB限制，只有当驱动器有多于1,024个磁柱并且系统BIOS或软盘控制器的BIOS提供了EBIOS时才会用到它。表11-4显示了EBIOS磁盘参数表的内容。

表11-4 EBIOS磁盘参数表的内容

偏移量	大 小	描		述	
0	字	逻辑磁柱数，多达1,024			
2	字节	逻辑磁头数，多达255			
3	字节	翻译后的表的署名字节——总为A0h			
4	字节	每磁道的物理扇区数			
5	字	未使用（过时的预补偿字）			
7	字节	未使用			
8	字节	驱动器控制字节			
		位	7=x	失败时的重试操作	
			6=x	位7	位6
				0	0=出错时重试
				0	1=不重试
				1	0=不重试
				1	1=不重试
			5=1	磁盘制造商的缺陷映射驻留在位于磁柱号+1的磁柱上（某些控制器/BIOS会忽略它）	
			4=0	未使用	
			3=1	驱动器有多于8个的磁头（某些控制器/BIOS会忽略它）	
			2=0	必须固定为0（没有重启）	
			1=0	必须固定为0（禁止IRQ）	
			0=0	未使用	
9	字	物理磁柱数，多达65,536			
11	字节	物理磁头数，多达16			
12	字	未使用（过时的装入分区字）			
14	字节	每磁道的逻辑扇区数，多达63			
15	字节	字节0~14的检查字节			

驱动器类型表

系统BIOS ROM包含有一个通用的驱动器表，但通常限于20或30个驱动器类型。驱

动器类型表的每一个入口都是一个 16 字节的磁盘参数，正如表 11-3 所显示的那样。不同的平台和制造商为每个类型分配了不同的驱动器。由于这一点不一致，不能认为一个驱动器类型号总代表某个指定的驱动器。

在很老式的系统和某些适配卡上，总是由适配卡上的开关或跳线来选择硬盘类型。这一点并没有太大的实际意义，因为驱动器的数目和类型已越来越多。

在目前的系统中，系统设置选择驱动器类型。可用一个软盘启动程序来执行它，或者在引导过程中使用特殊的组合键来触发系统 BIOS 设置程序。在为每个硬盘驱动器选择驱动器类型时，类型号就被保存到了 CMOS 内存中。涉及用来保存驱动器类型的寄存器请参看 CMOS 一章以了解其他细节。参看表 11-5，以了解 IBM 为大多数 AT 和 PS/2 类型系统所分配的驱动器类型。其他供应商很少使用表 11-5 所示的同样入口。

注意，从未分配类型 15。由于早期考虑认为不会有多于 15 个的驱动器类型，只有一个字节的 CMOS 内存分配给两个驱动器。当然，该限制很快就被突破了，所以为每个驱动器分配了扩展 CMOS 字节。如果 CMOS 的四位类型是 15，那么扩展的 CMOS 字节就用作驱动器类型。安装程序透明地处理它，并解释了为什么不使用驱动器类型 15。

许多较新的内置型磁盘控制器提供了一种特性，支持在一个可能没有合适表入口的系统上使用任何驱动器参数。这些适配器通常让用户设置 CMOS 内存来使用磁盘类型 1，如果控制器的 BIOS 获得了控制，就好像是设置了类型 1。如果是这样，它会自动检查真实的驱动器容量并使用生成的驱动器参数。

表 11-5　每种驱动器类型的驱动器参数

类型	总大小	最大磁柱数	磁头数	扇区数	写预补偿磁柱	装入区	缺陷映射
0	无驱动器						
1	10MB	306	4	17	128	305	无
2	20MB	615	4	17	300	615	无
3	30MB	615	6	17	300	615	无
4	62MB	940	8	17	512	940	无
5	46MB	940	6	17	512	940	无
6	20MB	615	4	17	-1（无）	615	无
7	30MB	462	85	17	256	511	无
8	30MB	744		17	-1（无）	733	无
9	112MB	900	15	17	-1（无）	901	无
10	20MB	820	3	17	-1（无）	820	无
11	35MB	855	5	17	-1（无）	855	无
12	50MB	855	7	17	-1（无）	855	无

续表

类型	总大小	最大磁柱数	磁头数	扇区数	写预补偿磁柱	装入区	缺陷映射
13	20MB	306	8	17	128	319	无
14	43MB	733	7	17	-1（无）	733	无
15	保留（见正文）						
16	20MB	612	4	17	0（所有）	663	无
17	40MB	977	5	17	300	977	无
18	57MB	977	7	17	-1（无）	1023	无
19	60MB	1024	7	17	512	732	无
20	30MB	733	5	17	300	732	无
21	43MB	733	7	17	300	732	无
22	30MB	733	5	17	300	733	无
23	10MB	306	4	17	0（所有）	336	无
24*	10MB	612	4	17	305	663	无
25	10MB	306	4	17	-1（无）	340	无
26	20MB	612	4	17	-1（无）	670	无
27	41MB	698	7	17	300	732	有
27	41MB	698	7	17	300	732	有
28	41MB	976	5	17	488	977	有
29	10MB	306	4	17	0（所有）	340	无
30	20MB	611	4	17	306	663	有
31	43MB	732	7	17	300	732	有
32	43MB	1023	5	17	-1（无）	1023	有
33	29MB	614	4	25	-1（无）	663	有

* 某些最早的 AT BIOS 在类型 23 后的入口中是零

几乎所有 20 世纪 90 年代生产的 BIOS 都提供了至少两个用户可自定义的驱动器类型，通常称为类型 47。对于类型 47 来说，系统 BIOS 在 CMOS 中为每个驱动器保存了参数表的值。这样就为所有现在和将来的驱动器提供了极大的灵活性。某些较老式的 BIOS 供应商支持用户定义的一个磁盘类型，但是大多数提供了两个。大多数 1995 年或以后生成的 BIOS 支持四个驱动器，而放弃了驱动器表的概念。

如果系统 BIOS 没有提供用户可定义的驱动器参数表，那就需要将期望的参数值填到

BIOS 的驱动器类型表的空白入口中。这意味着要标识一个合适的地址，可使用一个 EPROM 编程器来改变 BIOS ROM 的内容。大多数 BIOS ROM 在 POST 期间检查 ROM，所以如果做了任何改动，ROM 中的字节相加必须为 0。你可以在另一个未使用的类型入口使用一个字节来使 ROM 的和加起来为 0。记住，一些制造商，例如 AT，有更复杂的检查进程，必须绕过它。为避免这些麻烦，你可以考虑更新你的 BIOS ROM，选用支持用户定义驱动器参数的 BIOS ROM。

非标准的驱动类型，像 SCSI 和 ESDI，不使用驱动器参数表。驱动器参数保存在驱动器中，这时，在 BIOS 中将驱动器类型设置为 0，这样，系统 BIOS 不会试图去访问这个非标准的驱动器。

IDE 驱动器能使用任何驱动器，只要总容量不越限即可。例如，一个 100MB 的驱动器可以使用表 11-5 中除类型 9 外的任何入口。不幸的是，最接近的匹配类型是 4，它只支持访问 62MB。如果 BIOS 带有一个用户可设置的驱动器参数，那么可使用一组更合适的值来使用驱动器的全部容量。大多数 IDE 驱动器制造商提供了一组推荐的驱动器参数来获取全部的驱动器容量。

BIOS 初始化

通过将软盘驱动器（以前由 BIOS 安装）的中断 13h 向量移动到中断 40h，BIOS 开始硬盘的初始化。接着，装入一个新的中断向量 13h 来指向硬盘处理程序。如果软件试图使用中断 13h 来访问软盘，硬盘处理程序确定是否有软盘功能。如果有，就调用中断 40h 来处理该软盘功能。

硬盘 BIOS 的初始化可获得磁盘驱动器的类型。在一些自带硬盘 BIOS ROM 的老式硬盘控制器卡上，BIOS 会读取开关或跳线来选择驱动器类型。对于 AT 及后来的系统来说，硬盘支持内置于系统 BIOS 中。AT 系统 BIOS 初始化程序从 CMOS 内存中获取驱动器类型。一旦获得了驱动器类型，它就用作 ROM 中驱动器类型的一个索引。该磁盘参数表通常开始于地址 F000:E401。由于每个参数长为 16 字节，BIOS 会将类型号乘以 16 位来定位期望的磁盘参数表入口。指向驱动器 0 入口的指针保存在中断向量 41h 中，指向驱动器 1 参数的指针保存在中断向量 46h 中。较新的 BIOS 在 ROM 中没有驱动器参数表，并且允许用户分配需要的值。它们都保存在 CMOS 内存中，通常在 BIOS 设置内提供了某种类型的自动检测来设置驱动器的参数值。

使用 BIOS 诊断、中断 13h 和功能 14h，可以检查每个硬盘驱动程序。诊断通过后，使用功能 11h 重校驱动器，并检查最后一个扇区。从驱动器的参数入口计算最后一个扇区。同软盘控制器一样，检查只是简单地读取该扇区和 ECC 字节，并且只检查 ECC 对读取的字节是否有效。如果检查失败，通常会显示一条错误信息来指示发生了驱动器错误。

如果在初始化期间发生了任何驱动器控制器错误，BIOS 会将硬盘参数和硬盘处

理程序放置好。这样，即使硬盘系统可能有某个问题，也可以继续由诊断软件来访问。

硬盘 BIOS

通过中断 13 访问硬盘服务。所有的功能都要求在 DL 中用一个号来指定要使用的驱动器。将 DL 中的第 7 位置高表示正在访问硬盘，否则就是在访问软盘。将 DL 设为 80h，可访问硬盘驱动器 0（在 DOS 是 C:）；同样地，当 DL 设为 81h 时可访问驱动器 1。老式的 BIOS 系统只支持两个硬盘驱动器。为了访问更多的驱动器，必须由控制卡提供替代程序来替代系统 BIOS 中的硬盘服务。大多数 1995 年或以后的系统 BIOS 支持多达四个 IDE 型硬盘驱动器。服务完成时，状态字节返回 AH 寄存器中。该状态字节通过返回零来指示操作成功，否则，它将含有一个错误码。该状态字节也保存在 BIOS 数据区地址 40:74h 处。表 11-6 描述了状态代码。某些代码是专用控制器所特有的。

表 11-6 AH 中的返回状态码

十六进制	描　　述
0	操作成功，没有错误发生
1	无效值传递或不支持功能
2	丢失地址标签——如果没有扇区地址标签匹配请求，或者如果磁盘地址（磁柱、磁头、扇区）越过了磁盘边界，就会发生该错误。在一个 PS/2 ESDI 系统上，如果没有检测到索引或选择脉冲，也会返回该错误
3	可移动的媒质写保护
4	找不到请求的扇区——磁柱和磁头有效，但是无法定位到所要求的扇区上，在 PS/2 ESDI 系统上，如果控制器不能读第一个或最后一个相对块地址标志，也会产生这个错误
5	重启失败——在命令控制器重启之后，控制器保持忙状态或重启后状态不正确
6	磁盘改变——改变了磁盘驱动器媒质（仅可移动的媒质）
7	驱动器参数激活失败——对一个不存在的驱动器做出驱动器参数请求
8	DMA 过载——DMA 向或从控制器传送数据不够快，以至于发生了控制器超时
9	数据边界错误——试图写或读数据时造成了段回跳现象。如果读取多于 80h 的扇区，或者缓冲区并没有和一个段落的边界对齐（80 扇区×512 字节每扇区=64K），就会发生这种错误。对于某些操作（功能 Ah 和 Bh），会加入 ECC 字节到扇区中，这样总共有 516 个字节。这意味着能传送的数据比 64K 字节略少，通常只相当于 7Fh 的扇区，否则就会发生该错误（某些驱动器会返回 7 字节的 ECC 信息，进一步将扇区的最大数目限制到 7Eh）
A	检测到坏扇区标志——每个扇区有一个标志来指示该扇区是否坏掉。由专门的格式化命令来设置标出一个坏扇区。通常在表面分析检测到扇区的数据不可靠或有缺陷之后，会标出该扇区坏掉
B	检测到坏磁道——在上次操作中检测到了坏磁道标志。不重试该操作。每个磁道有一个标志来指示整个磁道有缺陷、不能使用。试图从这些坏磁道读会生成该错误信息

续表

十六进制	描　　述
D	格式化时的无效扇区数目——仅 PS/2 系统会返回它，在使用低级格式化 ESDI 驱动器时会出现许多可能的问题，这些问题也可由它来指示。这些问题包括： 1. 由于控制器保持忙太久，而发生了超过，或者在命令请求后保持忙 2. 格式化命令在操作期间触发中断 15h，AH=0Fh 和功能 AX=9000h。如果一个程序的这些功能有异常分支并且设置了进位标志，格式化操作就会中止，并返回该错误码 3. 格式化顺序出错，并指示 BIOS 编码错误（应永不发生） 4. 控制器格式化失败，因为主要的二级缺陷是映射不可用或有缺陷，或者发生了检查错误，或者发生了诊断错误
E	检测到控制数据地址标签——PS/2 BIOS 技术参考书籍中标识了该信息。还不清楚 PS/2 系统用它标识了什么错误，并且也从未被触发过。很明显 PC/XT/AT 类型的系统从未触发过它
F	DMA 决断层次越界——PS/2 BIOS 技术参考书籍标识了该信息。还不清楚 PS/2 系统用它标识了什么错误，并且也从未被触发过。很明显 PC/XT/AT 类型的系统从未触发过它
10h	读期间发生了 ECC 错误——如果发生下列情况，就返回该错误： 1. 扇区 ID 出错。控制器可以检测出，但是不能改正扇区 ID 的错误。这时可能要对磁柱再格式化 2. 来自扇区的数据混杂在了一起，即使 ECC 也不能修复它 3. 如果正在运行一个控制器诊断程序，控制器的内部 ECC 生成器测试失败
11h	ECC 更正数据错误——扇区有一个数据错误，由 ECC 机制改正。记住，错误中的连续位的数目返回到 AL 中。ECC 有小概率的错误（大约 100,000 个更正的错误中有一个不正确）
20h	硬盘控制器或驱动器问题——如果发生下列情况，就会返回该错误： 1. 驱动器的索引信号丢失。每旋转一次应出现一次索引信号，并且通过单独的线传送给控制器（仅老式驱动器） 2. 驱动器的信号指示控制器有一个写错误 3. 重样失败——在向外移动了最大数目的磁柱后，驱动器仍未使用信号指示到达磁柱 0 4. 定时期满后驱动器仍未准备好 5. 在控制器的一次诊断执行期间，控制器发现了一个问题 6. 接口错误，包括奇偶错误、命令完成错误（仅 PS/2）
31h	在可移走媒质的驱动器没有媒质——对可移走媒质的驱动器试图操作时，驱动器中无媒质
40h	查找操作失败——如果发生了下述情况，就会返回该错误： 1. 移动到磁盘上指定的有效的磁柱的努力失败。在命令驱动器往指定的扇区移动后，驱动器没有在特意设计的“查找完成”线上发出信号指示查找完成 2. 要检查驱动器准备好功能后，驱动器仍在查找期望的磁柱（不是一个错误） 3. 在查找操作（包括读和写）之后，磁柱和/或磁头地址与指定的查找地址不匹配
80h	超时——磁盘驱动器没有响应或忙于执行一个命令
AAh	驱动器没有准备好或没有被选中。如果使用中断 13h 功能 19h 来定位 PS/2 驱动器，也会返回该代码
B0h	驱动器中的卷标未锁——试图打开一个并未锁上的可移走媒质的驱动器就会返回该错误。参看中断 13h 功能 45h
B1h	驱动器中的卷标已锁上——试图从锁住的可移走媒质的驱动器中弹出媒质，这时就会发生该错误。参看中断 13h 功能 46h

续表

十六进制	描　　述
B2h	卷标不可移走——试图从一个不带可移走媒质的驱动器中弹出媒质，这时就会发生该错误，参看中断 13h 功能 46h
B3h	卷标在使用——如果试图从一个使用的可移走媒质的驱动器中弹出媒质，操作系统可能触发该错误
B4h	锁计数越界——在一个驱动器上试图使用多于允许的锁。参看中断 13h 功能 45h
B5h	有效弹出请求失败——可移走媒质驱动器拒绝弹出驱动器。这可能归因于驱动器开关或选项，或者归因于驱动器的硬件错误
BBh	出现了未定义的错误——控制器触发了一个错误码，但 BIOS 并不知道它，从理论上来说，从未发生该错误
CCh	选中驱动器上的写错误——每个驱动器有一条硬件线，指示选中的驱动器上发生了错误
E0h	无错误时的状态错误——在 AT 控制器上，状态端口 1F7h，错误位 0 被设置。这表示应在错误寄存器中记录一个错误。如果查看错误寄存器端口 1F1h，就不会记录错误。PS/2 系统不会触发该错误码 理论上讲，应从来不会发生该错误，还不太清楚控制器何时生成该错误，除非控制器有缺陷，或者在获取状态和读错误寄存器之间触发了其他命令
FFh	传感操作失败——从控制器获取错误数据的努力失败，或归因于响应超时，保持忙超时，或归因于无效的控制器状态

有关 BIOS 功能的特别说明

一般地，一个 PC/XT 控制器会指示一个自带 BIOS 代码的适配卡，来用在 PC 和 XT 系统上，PC/XT 控制器类似于那些内置于 AT BIOS 的磁盘服务，但是也有一些不同。PS/2 系统 BIOS 也为 ESDI 驱动器提供了磁盘服务，但也和 AT 标准有点不同。在各种服务和数据表中解释了这些差别。

尽管如 80h 和 81h 一样指定了有效的驱动器号，但是其他非标准的方法可能支持更大的驱动器号。为 PC/XT、AT 和 PX/2 类型控制器提供的老式磁盘 BIOS 服务，局限于两个物理驱动器。其他的硬件，例如 SCSI 控制器，每个控制器可能支持七个设备。这些非标准的控制器必须替换中断 13h 程序来提供相似的功能和访问两个以上的设备。

某些技术参考书指出，BIOS 功能，像 AH=Ah，Bh 和 Eh 被保留。然而，这些却是实用的功能。某些功能只属于某些特殊的系统，例如 XT。非标准驱动器，例如 ESDI 和 SCSI，不支持某些功能。

硬盘 BIOS 服务提供了下述中断和服务：

中　　断	功　　能	描　　述	平　　台
0Dh		操作完成	PC/XT
13h	ah=0	磁盘控制器重启	所有
13h	ah=1	读磁盘状态	所有

续表

中　断	功　能	描　述	平　台
13h	ah=2	读磁盘扇区	所有
13h	ah=3	写磁盘扇区	所有
13h	ah=4	检查磁盘扇区	所有
13h	ah=5	格式化磁盘磁道	PC/XT/AT/EISA
13h	ah=6	格式化有坏扇区的磁盘磁道	PC/XT
13h	ah=7	格式化多个磁柱	PC/XT
13h	ah=8	读磁盘驱动器参数	所有
13h	ah=9	初始化驱动器参数	所有
13h	ah=Ah	读带有 ECC 的磁盘扇区	PC/XT/AT/EISA
13h	ah=Bh	写带有 ECC 的磁盘扇区	PC/XT/AT/EISA
13h	ah=Ch	查找	所有
13h	ah=Dh	交替磁盘重启	所有
13h	ah=Eh	读扇区缓冲区	PC/XT/PS/1
13h	ah=Eh	ESDI 内幕诊断	PS/2
13h	ah=Fh	写扇区缓冲区	PC/XT/PS/1
13h	ah=Fh	ESDI 内幕诊断	PS/2
13h	ah=10h	检查驱动器准备好	所有
13h	ah=1 1h	重校	所有
13h	ah=1 2h	控制器 RAM 诊断	PC/XT
13h	ah=1 3h	驱动器诊断	PC/XT
13h	ah=1 4h	控制器内部诊断	所有
13h	ah=1 5h	读硬盘驱动器大小	AT+
13h	ah=1 9h	停靠磁头	PS/2
13h	ah=1 Ah	格式化 ESDI 驱动器	PS/2
13h	ah=1 Bh	获取 ESDI 制造头标	PS/2
13h	ah=1 Ch	ESDI 专用功能	PS/2
13h	ah=21h	读磁盘扇区，多块	PS/1
13h	ah=22h	写磁盘扇区，多块	PS/1
13h	ah=23h	设置控制器特性寄存器	PS/1
13h	ah=24h	设置多块	PS/1

续表

中　断	功　能	描　述	平　台
13h	ah=25h	获取驱动器信息	PS/1
13h	ah=41h	检查是否存在扩展	EBIOS
13h	ah=42h	扩展读扇区	EBIOS
13h	ah=43h	扩展写扇区	EBIOS
13h	ah=44h	扩展检查扇区	EBIOS
13h	ah=45h	锁住和开锁驱动器	EBIOS
13h	ah=46h	弹出媒质请求	EBIOS
13h	ah=47h	扩展查找	EBIOS
13h	ah=48h	获取扩展驱动器参数	EBIOS
13h	ah=49h	获取扩展磁盘改变状态	EBIOS
13h	ah=4Ah	启动磁盘模拟	可引导的 CD-ROM
13h	ah=4Bh	终止磁盘模拟	可引导的 CD-ROM
13h	ah=4Ch	启动磁盘模拟和引导	可引导的 CD-ROM
13h	ah=4Dh	返回引导类别	可引导的 CD-ROM
13h	ah=4Eh	设置硬件配置	EBIOS
15h	ah=52h	钩住可移走媒质的弹射	EBIOS
4Fh	ax=8100h	SCSI 命令，通用访问方法	SCSI
4Fh	ax=8200h	SCSI 现场检查，通用 SCSI 访问方法	SCSI
76h		操作完成	AT/EISA
76h		操作完成	PS/2

中断	描述	平台
0Dh	硬盘操作完成	PC、XT

如果 PC/XT 硬盘控制器完成了当前的命令和所有要求的数据传输，控制器就会指示一个硬件中断请求 IRQ5。如果没有激活更高优先级的中断，并且开放了该中断，就会调用中断 9Dh 的服务程序。该程序重设硬件中断并禁止 DMA 通道。

BIOS 中的硬盘操作完成服务程序执行三步特定的操作。首先，普通的中断结束命令 20h 被送往中断控制端口 20h。这会清除刚调用的中断。接着，屏蔽掉 DMA 端口通道 3。最后，程序禁止中断控制器上的 IRQ5，然后该程序退出。

如果触发下一个硬盘命令，PC/XT 的磁盘 BIOS 会重新开放硬件中断 5 和 DMA 通道。

中断	功能	描述	平台
13h	0	磁盘控制器重启	所有

命令软盘和硬盘控制器重启。驱动器参数传递给控制器，并触发重校命令，将磁头定位到磁柱 0。要想只重启硬盘控制器而不重启软盘控制器，使用交替重启，功能 0Dh。

对于双驱动器系统而言，驱动器号 80h 或 81h 并不重要，因为两个驱动器都会被重启。如果系统只有一个驱动器，并且 81h 作为驱动器号传递，那么硬盘控制器不会采取任何动作。

这个功能通常用在读、写、检查或格式化失败之后，而尚未重复这步操作之前。

调用：　ah=0
　　dl=驱动器号，80h 或 81h（参看文中所述）
返回：　ah=返回状态（参看表 11-5）
　　进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	1	读磁盘状态	所有

在从每个磁盘服务程序返回时，返回的状态值保存在 BIOS 数据区地址 40:47h 中。该功能从 40:47h 中返回当前保存的值，它代表了上次操作的状态。获取状态之后，该功能将 40:74h 处的状态设置为零。

记住，某些 BIOS 带有一些 bug，它会改变 AL 的值，即使正式的 IBM 说明书指出不应该改变 AL。

调用：　ah=1
　　dl=驱动器号，80h 或 81h
返回：　ah=返回状态（参看表 11-5）
　　al 可变（在某些系统上）
　　40:74h 置为零
　　进位=0，如果 ah 是零（没有错误）
　　1，如果 ah 非零（状态有错误代码）

中断	功能	描述	平台
13h	2	读磁盘扇区	所有

查找指定的地址，并读取一系列 512 字节的磁盘扇区。读取的数据放在 RAM 中，RAM 的地址由 ES:BX 指定。

该缓冲区应足够大以容纳读取的数据，并且总共不得超过64K。如果传输的大小在64K的16字节之内，则应该保证BX是段落对齐的，否则会发生数据边界错误。

有趣的是，BIOS会规范ES:BX比，这样传递一块看起来可能会造成段回跳的扇区将是可以接受的。例如，读一个32K的块，将ES:BX指向一个xxxx:FFFF0h的缓冲区是可以接受的。BIOS会为数据传输将它转化为xxxx+FFF:0000h。

如果试图传递多于一个完整磁道的扇区，不同供应商的BIOS采取的措施不尽相同。操作似乎可能工作，并且返回成功返回码，但是BIOS可能不能正确地读取数据！参看代码例11-2，DISKTEST，以全面了解对这个问题的解释。

如果在一个扇区上发生一个不可修正的硬错误（除11h外的任何错误），会立即终止操作，并忽略将要读取的数据。

返回时，如果错误码是11h，传输到缓冲区的数据可能有效。错误11h表示检测到了一个错误，并由控制器的ECC机制改正过来了。IBM指出，ECC机制有一个改错的概率，每100,000个错误改正中有一个会改错。当出现ECC改正状态时，AL寄存器存在错误长度，作为错误中连续位的数目。

对于除1、4、9外的大多数错误条件来说，建议利用功能0重启磁盘，并再试一次读操作。

调用：
ah=2
al=要读的扇区数（参看文中所述）
ch=磁柱号（10位磁柱号的低8位，从零开始）
cl，第6和7位=磁柱号的第8和9位
位0~5=起点扇区号（1~63）
dh=起点磁头号（0~255）
dl=驱动器号（80h+从零开始计数的驱动器号）
es:bx=指针，指向从磁盘所读取信息的放置地点

返回：
ah=返回状态（参看表11-5）
al=错误长度，如果ah=11h
未定义，大多数BIOS的其他返回状态值
进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	3	写磁盘扇区	所有

找到指定的地点并写入一系列512字节的磁盘扇区。数据从ES:BX指定的RAM地址中写到磁盘。缓冲区必须足够大以容纳全部的数据，并且不得超过64K。如果传输的大小在64K的16字节内，必须保证BX是段落对齐的，否则会出现边界错误。

如果试图传递多于一个完整磁道的扇区，不同供应商的BIOS采取的措施不尽相同。操作似乎可能工作，并且返回成功返回码，但是BIOS可能不能正确地写入数据！参看代

码例 11-2，DISKTEST，以全面了解对这个问题的解释。

对于除 1、4、9 外的大多数错误条件来说，建议利用功能 0 重启磁盘，并再试一次写操作。

调用：
ah=3
al=要写的扇区数（参看文中所述）
ch=磁柱号（10 位磁柱号的低 8 位，从零开始）
cl，第 6 和 7 位=磁柱号的第 8 和 9 位
位 0~5=起点扇区号（1~63）
dh=起点磁头号（0~255）
dl=驱动器号（80h+从零开始计数的驱动器号）
es:bx=指针，要写入磁盘的信息放置地点
返回：
ah=返回状态（参看表 11-5）
进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	4	检查磁盘扇区	所有

检查开始于指定地址的一系列扇区。该命令实际上并不检查每个字节是否和缓冲区中的内容相匹配。相反地，控制器从磁盘中读取指定的内容，并检查计算得到的 ECC 是否和以前保存在磁盘上的 ECC 值相匹配。如果定位了一个扇区，读取，并且 ECC 有效，那么可认为该数据有效。

某些技术参考书籍指出，必须为老式的 PC/XT/AT BIOS 指定 ES:BX 指针。我认为这是错的，因为我所查看的所有的 IBM 文档和控制器 IC 文档都指出，不必为检查而使用缓冲区。这和控制器从读自磁盘的扇区检查 ECC 信息所用的方法是一致的，从来都不需要比较缓冲区。

如果试图传递多于一个完整磁道的扇区，不同供应商的 BIOS 采取的措施不尽相同。操作似乎可能工作，并且返回成功返回码，但是 BIOS 可能不能正确地检查数据！参看代码例 11-2，DISKTEST，以全面了解对这个问题的解释。

对于除 1、4 或 9 外的大多数错误条件来说，建议利用功能 0 重启磁盘，并再试一次检查操作。

调用：
ah=4
al=要检查的扇区数（参看文中所述）
ch=磁柱号（10 位磁柱号的低 8 位，从零开始）
cl，第 6 和 7 位=磁柱号的第 8 和 9 位
位 0~5=起点扇区号（1~63）
dh=起点磁头号（0~255）

dl=驱动器号（80h+从零开始计数的驱动器号）
es:bx=未使用的缓冲区（参看上文）
返回： ah=返回状态（参看表 11-5）
进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	5	格式化磁道	PC/XT/AT/EISA

格式化磁盘驱动器上指定的磁道。磁道内的每个扇区都分配了一个扇区号、标注了扇区类型、初始扇区数据以及 ECC 字节。

对于 AT 以及后来的系统来说，ES:BX 指向的缓冲区提供了一张扇区信息表。该 512 字节的表为要格式化磁道中的每个扇区各保存了两个字节。表 11-7 显示了该表的内容。

表 11-7 格式化扇区数据表

字 节	描 述
0	扇区类型
	0=好扇区
	20h=未从替代的地址分配该扇区
	40h=将该扇区分配到一个替代的地址
	80h=坏扇区
1	扇区号，1~128

第一个字节指示了扇区是好还是坏。如果确定扇区有硬错误或软错误，则扇区可能会被标志为坏扇区。它也可用于软件防拷贝方案中，使那些好的扇区就像坏扇区一样。某些控制器能用替代的扇区来替换坏扇区。扇区类型 20h 和 40h 用来控制这种机制。第二个字节是要写入到扇区的扇区号。对于交错扇区来说，扇区的编号是连续的。

PC/XT 支持交错设置每个磁道，与慢速 PC/XT 不同。目前系统可以很快、连续地从驱动器处理扇区。在没有交错的 PC/XT 上，读第一个扇区时，磁头已经移过了下一个扇区的起点。这意味着驱动器必须等待再转一圈来到达下一个扇区！交错可以极大地提高磁盘访问的速度，它参考系统处理扇区的能力来在磁盘上排列扇区。

作为一个例子，图 11-4 显示了格式化一个磁道后的缓冲区的内容。该磁道每隔一个交错一次，共 17 扇区，物理扇区坏掉。物理扇区的编号在表中是上面的字节，它们不是 512 字节数据表的一部分。

es:bx->

扇区 1	扇区 2	扇区 3	扇区 4	扇区 5	扇区 6	扇区 7	扇区 8
0,1	0,10	0,2	0,11	0,3	0,12	0,4	80h,13

扇区 9	扇区 10	扇区 11	扇区 12	扇区 13	扇区 14	扇区 15	扇区 16
0,5	0,14	0,6	0,15	0,7	0,16	0,8	0,17

扇区 17	未使用	未使用					
0,9	0,0	0,0	所有剩下的字节为零（直到 512 字节为止）				

图 11-4　512 字节格式化数据缓冲区

交错数为 1 表示扇区的编号是连续的。较高的交错数用来放慢从磁盘取数据的速度，但仍在系统的能力范围之内。大多数 PC/XT 使用交错数 2 或更高的值，来获得最佳运行速度。对系统来说，如果交错数太小，会极大地降低运行速度，因为磁盘可能必须旋转一周来到达下一个扇区（最坏的情况）。对于 PC/XT 系统，因使用中断 13h 功能 0Fh（写扇区缓冲区）来为格式化设置数据内容。通常，PC 系统使用 F6h 作为格式化数据字节。

带 ESDI 驱动器的 PS/2 不支持该功能。使用功能 1Ah 来低级格式化 ESDI 驱动器。

对于除 1、4 或 0Dh 外的大多数错误条件，建议使用功能 0 重启磁盘，并重试该操作。

调用：　ah=5
al=交错数，1~10h（该值仅用于 PC/XT 类型的控制器，所有其他类型的控制器不使用它）
ch=磁柱号（10 位磁柱号的低 8 号，从零开始）
cl，位 6 和 7=磁柱号的第 8 位和 9 位
位 0~5=起始磁头号（1~63）
dh=起始磁头号（0~255）
dl=驱动器号（80h+从零开始的硬盘驱动器号）
es:bx=指针，指向地址表（AT 及后来的控制器，用于 PC/XT 上）

返回：　ah=返回状态（参见表 11-5）
进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	6	格式化有坏扇区的磁道	PC/XT

格式化磁盘驱动器上的每一个磁道，并将每个扇区标注为坏扇区。该功能与功能 5 相同，只是磁道上的每个扇区被标注为坏扇区。检测到软或硬错误时这一点很有用。必须将该磁道排除在可使用范围之外。只有 PC/XT 才支持该功能。

调用：　ah=6（仅 PC/XT）
al=交错数，1~10h（该值仅用于 PC/XT 类型的控制器，所有其他类型的控制器不使用它）

ch=磁柱号（10位磁柱号的低8号，从零开始）
cl，位6和7=磁柱号的第8位和9位
位0~5=起始磁头号（1~63）
dh=起始磁头号（0~255）
dl=驱动器号（80h+从零开始的硬盘驱动器号）

返回：ah=返回状态（参见表11-5）
进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	7	格式化多个磁柱	PC/XT

格式化从CX中指定磁柱开始的所有磁柱。该功能与功能5相似，只是需要格式化多个磁柱。要格式化整个驱动器，请将CX置零。仅PC/XT才支持该功能。

对于除1、4或0Dh外的大多数错误条件，建议使用功能0重启磁盘，并重试该格式化命令。

调用：ah=7（仅PC/XT）
al=交错数，1~10h
ch=起始磁柱号（10位磁柱号的低8位，从0开始）
cl，第6和7位=磁柱号的位8和9
dl=驱动器号（80h+从零开始的硬盘驱动器号）

返回：ah=返回状态（参看表11-5）
进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	8	读磁盘驱动器参数	所有

获取指定驱动器的驱动器参数。对于AT类型的磁盘控制器来说，可使用指针间接地读取数据。指针保存在向量41h或40h中，41h或40h取决于选择的驱动器号是0还是1。

在PC/XT类型的控制器上，如果指定的驱动器开关都是ON，那么可利用保存在中断向量41h中的指针间接地读取数据。否则，PC/XT控制器卡上的开关用于索引四参数表入口中的一个，而中断向量41h中的指针作为该表的基地址。参看表11-8以了解这些开关的设置（在功能9中）。

在SCSI和PS/2 ESDI系统的控制上，返回保存在磁盘上的参数信息。目前的IDE控制器提供了LBA（逻辑块寻址）模式来访问大于504MB的驱动器。如果激活了LBA，将会修改返回的参数来支持访问整个驱动器。

必须把最大磁柱号加1来获得磁柱的总数目，因为其中不包括诊断磁柱，BIOS总保留一个磁柱用于诊断。可以访问该磁柱，但是在测试控制器和驱动器期间可能会损失坏数据。如果要求真实的磁柱数，请在总数目上再加1。据说某些Zenith PC系统还保留了一个磁柱，

但不知道有什么用。这种情况下，返回的磁柱数比硬盘参数表中指定的磁柱数少三。

如果试图从越界的驱动器号获取驱动器参数，或者该驱动器不存在，或者没有激活硬盘控制器，该功能会失败。该功能并不返回一个指向驱动器类型的表的指针，和许多其他的技术参考书籍所指出的一样。

调用：　ah=8
　　dl=驱动器号（80h+从零开始的硬盘驱动器号）
返回：　如果成功
　　ah=0
　　al=未定义
　　进位=0
　　ch=最大磁柱号（10 位磁柱号的低 8 位，从零开始）
　　cl，第 6 和 7 位=磁柱号的第 8 位和第 9 位
　　第 0~5 位=最大扇区号（1~63）
　　dh=最大磁头号（0~255）
　　dl=驱动器数目
　　如果失败（但至少激活了一个驱动器）
　　ah=7
　　进位=1
　　al=cx=dx=0
　　如果失败（因为没有硬盘存在）
　　ah=1
　　进位=1
　　al=bx=cx=dx=es=0

中断	功能	描述	平台
13h	9	初始化驱动器参数	PC

用指定的驱动器类型参数初始化指定的驱动器控制器。对于使用标准驱动器（即，不是 ESDI）的 AT 或后来的系统来说，如果选择了驱动器 0，该参数用来初始化驱动器，地址 0:104h 处中断向量 41h 中的指针指向该参数。如果选择了驱动器 1，那么地址 0:118h 处的中断向量指向初始化参数表。一个 PS/2 ESDI 驱动器不使用参数表，因为参数表就保存在驱动器上，对于 PS/2 ESDI 驱动器来说，这个命令不执行任何操作，而只是立即成功地返回。其他非 AT 控制器，例如 SCSI，也不执行任何操作，也只是成功返回。

对于 PC/XT 来说，中断 41h 指向驱动器类型表的起点，该表只有四个入口长。从控制器卡读取指定驱动器的开关来确定使用四种类型驱动器中的哪一个。开关设置如表 11-8 所示。

表 11-8 硬盘类型选择——PC/XT 控制器

使用表入口	驱动器 0		驱动器 1	
	开关 1	开关 2	开关 1	开关 2
0	开	开	开	开
1	开	关	开	关
2	关	开	关	开
3	关	关	关	关

在一个 PC/XT 上，如果驱动器号在 80h 到 87h 之间，那么驱动器 0 和 1 都会初始化（这不是文档说明有错，PC/XT BIOS 代码实际就是这么做的）。

调用： ah=9
dl=驱动器号（80h+从零开始的硬盘驱动器号）
返回： ah=返回状态（参看表 11-6）
进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	0Ah	读带 ECC 的磁盘扇区	PC/XT/AT/EISA

读多个扇区并保存到 ES:BX 指定的缓冲区中。传送到缓冲区的每个 512 字节扇区包括一个 4 字节的 ECC 代码。这意味着每个扇区需要 516 个字节的缓冲区空间。如果在一个扇区上检测到了错误数据，操作会马上停止，并且不会修正失败的扇区。

某些 ESDI 驱动器提供选项来使用 7 字节的 ECC 码，这意味着每个扇区将有 519 字节长。7 字节的 ECC 提高了纠错能力，但是大多数控制器缺省是 4 字节模式，以便和老式的软件完全兼容。

如果试图读取多于一个磁道的扇区，不同供应商的 BIOS 会有不同的措施来处理这个问题。似乎操作仍可以工作，返回成功的代码，但是 BIOS 可能读取了不正确的数据！请参看代码例 11-2，DISKTEST，以全面理解这个问题。

PC、XT、AT 以及 EISA 系统支持该功能，但是使用 ESDI 驱动器的 PS/2 却不支持它。通常，应用程序和操作系统使用中断 13h 功能 2 来读取多个扇区。

对于除 1、4 或 9 以外的大多数错误条件来说，建议用功能 0 重启磁盘，然后重试该功能。

调用： ah=0Ah
al=要读的扇区数目（参见上文）
ch=磁柱号（10 位磁柱号的低 8 位，从零开始）

cl，第 6 和 7 位=磁柱号的第 8 和 9 位
　　位 0~5=起点扇区号（1~63）
dh=起点磁头号（0~255）
dl=驱动器号（80h+从零开始的硬盘驱动器号）
es:bx=指针，指向读取扇区的保存地址

返回：
ah=返回状态（参看表 11-5）
进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	0Bh	写磁盘扇区，带 ECC	PC/XT/AT/EISA

从 ES/BX 指向的缓冲区向多个扇区写入。每 512 字节的扇区必须带有与其相关的 4 字节 ECC 码。这就意味着每个扇区需要 516 字节的缓冲区空间。

某些 ESDI 驱动器提供选项来使用 7 字节的 ECC 码，这意味着每个扇区将有 519 字节长。7 字节的 ECC 提高了纠错能力，但是大多数控制器缺省是 4 字节模式，以便和老式的软件完全兼容。

如果试图写入多于一个磁道的扇区，不同供应商的 BIOS 会有不同的措施来处理这个问题。似乎操作仍可以工作，返回成功的代码，但是 BIOS 可能写入了不正确的数据！请参看代码例 11-2，DISKTEST，以全面理解这个问题。

PC、XT、AT 以及 EISA 系统支持该功能，但是使用 ESDI 驱动器的 PS/2 却不支持它。通常，应用程序和操作系统使用中断 13h 功能 3 来写多个扇区。

对于除 1、4 或 9 以外的大多数错误条件来说，建议用功能 0 重启磁盘，然后重试该功能。

调用：
ah=0Bh
al=要写的扇区数目（参见上文）
ch=磁柱号（10 位磁柱号的低 8 位，从零开始）
cl，第 6 和 7 位=磁柱号的第 8 和 9 位
　　位 0~5=起点扇区号（1~63）
dh=起点磁头号（0~255）
dl=驱动器号（80h+从零开始的硬盘驱动器号）
es:bx=指针，指向写入扇区的数据获取地址

返回：
ah=返回状态（参看表 11-5）
进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	0Ch	查找	PC/XT/AT/EISA

将驱动器磁头移动到指定的磁柱上，对于 BIOS 读、写和核对扇区的功能来说，查找

操作自动进行（功能2，3，4，Ah和Bh）。该功能不传递数据。

对于除1或4外的错误条件，建议使用功能0重启硬盘，并重试该操作。

调用：　ah=0Ch
　　ch=磁柱号（10位磁柱号的低8位，从零开始）
　　cl，第6和7位=磁柱号的第8和9位
　　dh=起点磁头号（0~255）
　　dl=驱动器号（80h+从零开始的硬盘驱动器号）
返回：　ah=返回状态（参看表11-5）
　　进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	0Dh	替代的磁盘重启	所有

命令硬盘控制器重启。磁盘参数被送到控制器，并触发重校命令来将驱动器磁头定位到磁柱0。它执行与中断13h功能0（磁盘控制器重启）相同的操作，只是不重启软盘控制器。

对于双驱动器系统来说，驱动器号（80h或81h）并不重要，因为两个驱动器都会重启。如果系统只有一个驱动器并将值81h作为驱动器号，那么硬盘控制器不会发生任何动作。

调用：　ah=0Dh
　　dl=驱动器号80h或81h
返回：　ah=返回状态（参看表11-5）
　　进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	0Eh	读扇区缓冲区	PC/XT/PS/1

读取磁盘控制器的内部缓冲区并保存在ES:BX指向的地址中。此功能仅用于诊断，PC/XT系统和PS/1支持它。某些其他的系统可能也支持该功能，但是大多数不支持。没有数据读自磁盘驱动器。

调用：　ah=0Eh（仅PC/XT和PS/1）
　　dl=驱动器号80h或81h
　　es:bx=指针，指向保存扇区缓冲区的地址
返回：　ah=返回状态（参看表11-5）
　　进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	0Eh	ESDI内窥诊断	PS/2

在设置输入参数指向磁柱 0、扇区 1、磁头 0 后，对 ESDI 控制器触发一个保留命令 11h。看起来该操作使用 DMA，还不知道该功能做何测试用。

调用：　ah=0Eh（仅 PS/2 的 ESDI）
　　dl=驱动器号 80h 或 81h
返回：　ah=返回状态（参看表 11-5）
　　进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	0Fh	写扇区缓冲区	PC/XT/PS/1

从 ES:BX 向磁盘控制器的内部扇区写数据。它用于诊断，并为功能 5 的格式化设置数据内容。仅用该功能不会向磁盘驱动器写数据。PC/XT 和 PS/1 系统支持该功能。其他系统可能支持该功能，但大多数并不支持。

该功能为格式化装入数据，由格式化命令来初始化每个扇区。DOS 标准要求 512 字节缓冲区全是 F6 字节。其他的操作系统可能选择不同的字节。

调用：　ah=0Fh（仅 PC/XT 和 PS/1）
　　dl=驱动器号 80h 或 81h
　　es:bx=指针，指向要写到磁盘控制器的数据（512 字节）
返回：　ah=返回状态（参看表 11-5）
　　进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	0Fh	ESDI 内幕诊断	PS/2

在设置输入参数指向磁柱 0、扇区 1、磁头 0 后，对 ESDI 控制器触发命令 10h。看起来该操作使用 DMA，还不清楚该功能做何测试用。

调用：　ah=0Eh（仅 ESDI 和 PS/2）
　　dl=驱动器号 80h 或 81h
返回：　ah=返回状态（参看表 11-5）
　　进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	10h	检查驱动器准备好	所有

检查指定的驱动器是否准备好执行一个命令。返回状态为 0 表明控制器不忙，指定的控制器已准备好。

调用：　　ah=10h
　　dl=驱动器号（80h+从零开始的硬盘驱动器号）
返回：　　ah=返回状态（参看表 11-5）
　　进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	11h	重校	所有

将指定驱动器的磁头 0 移动到磁柱 0。除无效值传递错误号 1 外，对大多数错误条件来说，建议使用功能 0 重启磁盘，并重试该操作。

调用：　　ah=11h
　　dl=驱动器号（80h+从零开始的硬盘驱动器号）
返回：　　ah=返回状态（参看表 11-5）
　　进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	12h	控制器 RAM 诊断	PC/XT

启动对磁盘控制器的扇区缓冲区的内部测试，只有 PC/XT 系统才支持它。如果测试成功，ah 返回零。

调用：　　ah=12h（仅 PC/XT）
　　dl=驱动器号 80h 或 81h
返回：　　ah=返回状态（参看表 11-5）
　　进位=0，如果成功；=1，如果出错
　　al=0

中断	功能	描述	平台
13h	13h	驱动器诊断	PC/XT

对驱动器所带的磁盘控制器启动一次内部诊断，只有 PC/XT 系统才支持它。如果测试成功，ah 返回零。

调用：　　ah=13h（仅 PC/XT）
　　dl=驱动器号 80h 或 81h
返回：　　ah=返回状态（参看表 11-5）
　　进位=0，如果成功；=1，如果出错
　　al=0

中断	功能	描述	平台
13h	14h	控制器内部诊断	所有

启动一次内部磁盘控制器自检。PC/XT 及所有后来的系统都支持该功能，但是 AT+ 系统对此未作说明。如果测试成功，ah 被置零。

调用：　ah=14h
　　dl=驱动器号（80h+从零开始的硬盘驱动器号）
返回：　ah=返回状态（参看表 11-5）
　　进位=0，如果成功；=1，如果出错
　　al=0

中断	功能	描述	平台
13h	15h	读磁盘驱动器大小	AT+

检查驱动器是否有效，并获取硬盘驱动器上的 512 字节扇区数目。该命令不访问磁盘控制器和驱动器。从指定驱动器的当前驱动器参数表中确定该大小，计算方法如下：

总扇区数=（磁头数）×（每磁道的扇区数）×（磁柱数-1）

注意到该功能从总的大小中移走了一个磁柱用于诊断。奇怪的是，BIOS 代码从未使用过该磁柱。但是由于某些磁盘控制器的设计可能会将该磁柱用于诊断，所以最好不要使用最后一个磁柱。其他任何磁盘服务都可以完全访问该诊断磁柱。

调用：　ah=15h
　　dl=驱动器号（80h+从零开始的硬盘驱动器号）
返回：　如果成功
　　ax=3，硬盘可访问
　　进位=0，
　　cx:dx=扇区总数（32 位）
　　如果失败
　　ax=0，没有驱动器
　　进位=1
　　cx=dx=0

中断	功能	描述	平台
13h	19h	停靠磁头	PS/2

将指定驱动器的磁头停靠在一个安全的地方，该地方由驱动器制造商定义。触发一个停止马达命令，并锁住驱动器避免进一步使用，直到系统重启。触发该命令后，任何其他试图访问收起的驱动器的努力都会返回一个“驱动器没有准备好”错误码 Aah。

仅使用 ESDI 驱动器的 PS/2 系统（包括 MCA 和 ISA 总线类型）才支持该功能。

调用：　　ah=19h
　　dl=驱动器号 80h 或 81h
返回：　　ah=返回状态（参看表 11-6）
　　进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	1Ah	低级格式化 ESDI 驱动器	PS/2

对于带有 ESDI 的 PS/2 系统来说，该命令可以低级格式化整个驱动器，也可以执行几种相关的服务，例如表面分析检查缺陷。完成该命令时，所选择的驱动器上的所有数据都被损坏（unformat 工具不能恢复该信息），在使用该驱动器之前必须装入分区信息和进行高级格式化。在 DOS 中可以用 FDISK 和 FORMAT 实现。

提供了一个可任选的缺陷表来通告格式化进程跳过指定的缺陷。缺陷表的每个入口长四个字节，表 11-9 描述了缺陷表的入口结构。

表 11-9　缺陷表入口结构

偏移量	大　小	描　　述		
0	字节	相对扇区地址，LSB（位 0~7）		
1	字节	相对扇区区地址，（位 18~15）		
2	字节	相对扇区地址，MSB（位 16~24）		
3	字节	标志与缺陷计数字节		
		位	7=1	扇区是磁道上的最后一个逻辑扇区
			6=1	扇区是磁道上的第一个逻辑扇区
			5=1	扇区在磁柱的最后一个逻辑扇区上
			4=1	逻辑扇区被推到了下一个磁道
			3=x	一行中的缺陷数（参看下文）
			2=x	
			1=x	
			0=x	

一行中的缺陷数值是指从前一个磁柱推到本磁柱的扇区数。如果磁柱的缺陷太多而不能由该磁柱上的保留缺陷区所容纳，就会发生这种情况。一个磁柱上的所有扇区都使用相同的“一行中的缺陷数”。该缺陷管理方式假定后面的磁柱比普通的缺陷数要少，并且从一个磁柱推过来的扇区可以完全放入下一个磁柱中，控制器跟踪这些推过来的扇区以便透明地访问它们。

低级格式化传递的 CL 寄存器带有控制位来代表不同的子功能和操作。在每个磁柱操作完成时，第 4 位用来触发中断 15h 功能 AH=0Fh。中断 15h 功能同时调用 AL 指定的操作。AL 值为 1 表示表面分析，值 2 表示格式化。该特性支持格式化程序钩住中断，并通过统计

发生的次数，来显示正在格式化哪个磁柱。通过钩住中断 15h、AH=0Fh，可以提供一种可靠的方法来终止格式化进程。为了能继续格式化，从中断 15h、功能 AH=0Fh 返回时必须清除进位标志。如果操作需要中止退出，请设置进位标志。

要执行表面分析，必须首先将控制位 3 置零来格式化驱动器。一旦格式化完成，就会设置控制位 3 来启动表面分析。为了保存表面分析测试的结果，请设置第 2 位。它将来自表面分析的缺陷信息保存到次级缺陷映射中。如果要忽略表面分析的结果，请清除第 2 位。

控制位 0 和 1 指示是否使用缺陷映射。在正常情况下，应使用这两个映射。保持两位清零，就可以使用这两个缺陷映射。

调用：
ah=1Ah
al=0，不使用缺陷表
>0，缺陷表中入口的数目
cl=控制位
位 7 ~ 5=0
4=1 每磁~柱操作完成时发送中断（参看上文）
3=1 执行表面分析（参看文本）
2=1 将表面分析的结果保存到次级缺陷映射中
1=1 忽略次级缺陷映射
0=1 忽略主级缺陷映射（由驱动器制造商设置映射）
dl=驱动器号 80h 或 81h
es:bx=指向缺陷表的指针

返回：
ah=返回状态（参看表 11-5）
进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	1Bh	获取 ESDI 制造表头	PS/2

对于带有 ESDI 驱动器的 PS/2 系统来说，该命令用来获得制造表头，还可用来从驱动器中取走主级缺陷映射。其操作类似于标准的读扇区功能。从该命令读取的第一个扇区包括有制造表头，内有缺陷入口的数目，然后是缺陷映射的第一个入口。读其他的扇区会获得完整的缺陷映射。该功能不检查是否到了缺陷的终点，而只是读取所要求的扇区数，即使缺陷映射使用的扇区数比要求读取的少，该功能尚未说明。

调用：
ah=1Bh
al=待读扇区的数目
dl=驱动器号，80h 或 81h
es:bx=指针，指向读自磁盘的信息的放置地点

返回：
ah=返回状态（参看表 11-5）
进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	1Ch	ESDI 专用功能	PS/2

对于带有 ESDI 驱动器的 PS/2 系统来说，该命令从一系列内置于系统 BIOS 的子功能中选择一个子功能。由于这个功能尚未说明，特殊的子功能可能需要其他的寄存器。一般地，只有用替代服务程序替换了 PS/2 系统上的其他中断 13h 硬盘系统服务之后，才能使用该功能。

调用：ah=1Ch
dl=驱动器号 80h 或 81h
al=子功能
返回：ah=返回状态（参看表 11-6）
进位=0，如果成功；=1，如果出错

AL=	ESDI 专用子功能命令
0	返回“无操作”错误（无操作）
1	尚未说明的控制器命令 0612h，无 DMA
2	尚未说明的控制器命令 0612h，无 DMA
3	尚未说明的控制器命令 0612h，无 DMA
4	尚未说明的控制器命令 0612h，无 DMA
5	尚未说明的控制器命令 0612h，无 DMA
6	尚未说明的控制器命令 0612h，无 DMA
7	返回“无操作”错误（无操作）
8	获取命令完成状态
9	获取设备状态
A	获取设备配置
B	获取控制器配置
C	获取可编程选项选择（POS）信息
D	尚未说明的控制器命令 0614h，无 DMA
E	翻译相对扇区寻址
F	尚未说明的控制器命令 0614h，无 DMA
10h	未知的非控制器子功能
11h	未知的非控制器子功能
12h	尚未说明的控制命令 0614h，无 DMA

子功能细节

子功能	描述	中断	功能
AL=8	获取命令完成状态	13h	AH=1Ch

为指定的驱动器获取上次完成的命令的命令完成状态块。完成一个命令时，控制器触发 IRQ14，中断 76h，表示状态准备好可读。使用该子功能时，ES:BX 指向保存命令完成状态块的缓冲区。参看端口 3510h，输出，获取命令完成状态（命令 607h），以及表 11-22，以了解命令完成状态块的详细细节。

子功能	描述	中断	功能
AL=9	获取设备状态	13h	AH=1Ch

获取上次命令完成时所指定的驱动器的设备状态块。在完成一条命令时，控制器会触发中断 IRQ 14，中断 76h，来指示状态已经准备好，现在可以读取状态。在使用这个子功能时，ES: BX 指向设备状态块的缓冲区。参看端口 3510h，输出，获取设备状态（命令 608h），和表 11-23，来详细了解设备状态块的相关信息。

子功能	描述	中断	功能
AL=Ah	获取设备的配置	13h	AH=1Ch

命令控制器返回驱动器的配置状态。它包含了类似于驱动器参数表的信息，其中包括最大磁柱、磁道数以及其他相关信息。使用这个子功能时，ES:BX 指向设备状态块的缓冲区。参看端口 3510h，输出，获取配置（命令 609h），和表 11-24，来了解配置状态块的相关细节。

子功能	描述	中断	功能
AL=Bh	获取控制器的配置	13h	AH=1Ch

命令控制器返回控制器的配置状态块。它可以获取控制器中微代码的版本。使用这个子功能时，ES:BX 指向保存了配置状态块的缓冲区。参看端口 3510h，输出，获取配置（命令 6E9h），和表 11-26 来了解配置状态的细节。

子功能	描述	中断	功能
AL=Ch	获取 POS 信息	13h	AH=1Ch

获取可编程选项选择（POS）寄存器的内容。使用这个子功能时，ES:BX 指向保存了配置状态块的缓冲区。参看端口 3510h，输出，获取可编程选项选择信息（命令 6EAh），和表 11-27，来了解 POS 信息块的相关细节。

子功能	描述	中断	功能
AL=Eh	翻译相对扇区地址	13h	AH=1Ch

ESDI 驱动器有两种类型的寻址方式：相对扇区寻址（RSA）和绝对扇区寻址（ASA）。RSA 用于读、写、校验及其他普通的操作。相对扇区寻址隐藏了缺陷，而使整个磁盘看起

来像一个长的线性阵列。如果必须分配掉有缺陷的扇区，那么就必须使用绝对扇区寻址。这个功能可以将RSA转化为ASA。

参看端口3510h，输出，翻译相对扇区地址（命令6Ebh），以获取其他相关细节。

中断	功能	描述	平台
13h	21h	读磁盘扇区，多块	PS/1

这个命令类似于读磁盘扇区功能2，只是在每个扇区传输结束时不会生成中断，而是在传输了一组扇区之后才会生成中断。必须在这个功能之前先触发多块设置功能。参看多块设置功能以了解其他相关信息。

以前尚未公开过这个功能，大多数控制器并不支持它。

调用：
ah=21h
al=要读取的扇区块的数目
ch=磁柱号（10位磁柱号的低8位，从零开始）
cl, 位7，6=磁柱号的位9和8
位0~5 =起始扇区号（1~63）
dh=起始磁头号（0~255）
dl=驱动器号（80h+从零开始计数的硬盘号）
es:bx=指针，指向一个放置信息的地址，这些信息读自磁盘

返回：
ah=返回状态（参看表11-5）
al=阵发错误长度，如果ah=11h
进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	22h	写磁盘扇区，多块	PS/1

这个命令类似于写磁盘扇区功能3，只是在每个扇区传输结束时不会生成中断，而是在传输了一组扇区之后才会生成中断。必须在这个功能之前先触发多块设置功能。参看多块设置功能以了解其他相关信息。

以前尚未公开过这个功能，大多数控制器并不支持它。

调用：
ah=22h
al=要写入的扇区块的数目
ch=磁柱号（10位磁柱号的低8位，从零开始）
cl 位7，6=磁柱号的位9和8
位0~5 =起始扇区号（1~63）
dh=起始磁头号（0~255）
dl=驱动器号（80h+从零开始计数的硬盘号）
es:bx=指针，指向一个放置信息的地址，这些信息写向磁盘

返回：　　ah=返回状态（参看表 11-6）
进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	23h	设置控制器特性寄存器	PS/1

它控制控制器的各种内部特性。其他寄存器可能也含有某个特性所必须的值，这与供应商有关。还不清楚 PS/1 控制器是否支持所有下面描述的特性。开放和禁止特性设置控制器内部的标志位。一旦接受，就可以使用下一个设置特性功能而不用改变前面的特性设置。表 11-10 列出了特性选项。

表 11-10　控制器特性选项

十六进制值	特　　性
1	使用 8 位数据传输，而不使用普通的 16 位传输
2	开放写高速缓存
22	与写相同，用户定义区域
33	禁止重试
44	在带 ECC 读/写时返回的 ECC 字节长度被指定
55	禁止预测特性
66	禁止恢复到上电缺省状态
77	禁止 ECC
81	禁止 8 位数据传输，使用 16 位（普通情况）
82	禁止写高速缓存
88	开放 ECC（普通情况）
99	开放重试（普通情况）
AA	开放预测特性
BB	在带 ECC 命令读/写时返回 4 字节
CC	开放恢复到上电缺省状态
DD	与写相同，整个磁盘

该功能尚未说明，大多数 AT 控制器不支持它。

调用：　　ah=23h
al=功能号
dl=驱动器号（80h+从零开始的硬盘驱动器号）
可能用到其他寄存器

返回：　　ah=返回状态（参看表 11-5）
　　进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	24h	设置多块	PS/1

控制器提供了一种特性来一次传输多组扇区，每个块的扇区数目由本功能装入。有效的块大小为 2、4、8 和 16。如果控制器的内部缓冲区大于 8K，那么可能会支持更大的块。

在本功能之后可能跟着读或写磁盘扇区，多块功能 21h 或 22h。读或写磁盘扇区功能开始传输指定块的数据。控制器只在每块扇区传输完成时才中断系统。

例如，如果传输 5 扇区，块的大小是 2，那么先传前 2 个扇区，然后紧接着传下 2 个扇区，而最后一块会传送剩下的 1 扇区。本例中只产生了三次中断，而不是普通情况下为每个扇区都产生一次中断。

多块特性是为了提高操作速度而设，该功能尚未说明，并且有其他一些操作要求未知。

调用：　　ah=24h
　　al=每块的扇区数目，2、4、8 或 16
　　dl=驱动器号（80h+从零开始的硬盘驱动器号）
返回：　　ah=返回状态（参看表 11-5）
　　进位=0，如果成功；=1，如果出错

中断	功能	描述	平台
13h	25h	获取驱动信息	PS/1

该功能获取保存在驱动器上的 256 字的驱动器信息。只有在像 IDE 那样，在磁盘上保存了这些数据的驱动器上，该命令才能起作用。大多数 AT 控制器不支持该命令。参看端口 1F7h，命令 ECh 下的表 11-16，以了解驱动器信息。

中断	功能	描述	平台
13h	41h	检查是否存在扩展	EBIOS

该功能用来确定指定的硬盘驱动器支持哪种 EBIOS 功能。如果有的话，在检查任何其他 EBIOS 返回值的有效性之前，必须检查返回的进位位和 BX 寄存器。

调用：　　ah=41h
　　bx=55AAh
　　dl=驱动器号（80h+从零开始的硬盘驱动器号）
返回：　　如果成功

进位=0
ah=EBIOS 扩展的主版本
21h=版本 1.1（不要问为什么）
al=可变
bx=AA55h
cx=支持位
位 15 ~ 3=0　　未使用（留待将来增强用）
2=1　　支持增加型磁盘驱动器（EDD），支持 BIOS 子功能 41h、48h 和 4Eh
2=1　　支持锁住或弹出驱动器，支持 BIOS 子功能 41h、45h、46h、48h 和 49h，以及中断 15h 功能 52h。
0=0　　支持访问增强型驱动器，支持 BOIS 子功能 41h、42h、43h、44h、47h 和 48h
如果失败
进位=1 或 bx 不是 AA55

中断	功能	描述	平台
13h	42h	扩展读扇区	EBIOS

如果 EBIOS 服务支持，就可以从磁盘向内存读扇区。该扩展功能要求一个磁盘地址包来指定传输的扇区数，从磁盘何处开始读，以及将数据装入内存的何处。表 11-11 列出了磁盘地址包的确切形式。

表 11-11　扩展磁盘服务的磁盘地址包

偏移量	大　小	描　　述
0	字节	包的大小，单位：字节。目前包长为 16 字节，所以应使用值 10h。可接受更大的值，但是更小的值会使该功能被拒绝
1	字节	目前未使用，必须设为 0
2	字节	要处理的扇区数，从 1 到 127。拒绝更大的值
3	字节	目前未使用，必须置为 0
4	字	传送的内存偏移量（仅读/写）
6	字	传送的内存段址（仅读/写）
8	四字	传送的起点逻辑扇区，64 位。对于不支持逻辑扇区的驱动器来说，EBIOS 将地址转化为磁柱——磁头——扇区地址。下面的公式定义了逻辑扇区与所中的磁柱——磁头——扇区之间的关系：逻辑扇区=(((磁柱号×最大磁头数)+磁头号)×最大扇区数)+扇区号-1

调用：　　ah=42h
　　dl=驱动器号（80h+从零开始的硬盘驱动器号）
　　ds:si=指针，指向磁盘地址包（参看表 11-11）
返回：　　如果成功
　　ah=0
　　进位=0
　　如果失败
　　进位=1
　　ah=返回状态（参看表 11-6）
　　磁盘地址包的字节 2 指示成功读取的扇区数目

中断	功能	描述	平台
13h	43h	扩展写扇区	EBIOS

如果 EBIOS 支持写扇区，就会从内存磁盘写扇区。本扩展要求一个磁盘地址包来指定待传输的扇区数目，磁盘上开始写的地址，以及从内存何处获取数据。参看表 11-11 可了解磁盘地址包的确切格式。

提供了一个选项来检查某些 BIOS 操作的写。可用功能 48h 查看是否支持该特性。不带检查的写类似于标准的 BIOS 功能 3，写扇区。

调用：　　ah=43h
　　al=0 或 1，如果要关闭检查
　　=2，写时开放检查
　　dl=驱动器号（80h+从零开始的硬盘驱动器号）
　　ds:si=指针，指向磁盘地址包（参看表 11-10）
返回：　　如果成功
　　ah=0
　　进位=1
　　如果失败
　　进位=1
　　ah=返回状态（参看表 11-6）
　　磁盘地址包的字节 2 指示成功写入的扇区数目

中断	功能	描述	平台
13h	44h	扩展检查扇区	EBIOS

如果 EBIOS 支持，就会检查扇区是否确实没有错误。该扩展功能要求一个磁盘地址包指定要检查的扇区数目，磁盘上开始检查的地址。该功能不使用内存，参看表 11-11 可了解磁盘地址包的确切格式。

调用：　ah=44h
dl=驱动器号（80h+从零开始的硬盘驱动器号）
ds:si=指针，指向磁盘地址包（参看表 11-10）

返回：　如果成功
ah=0
进位=1
如果失败
进位=1
ah=返回状态（参看表 11-6）
磁盘地址包的字节 2 指示成功检查的扇区数目

中断	功能	描述	平台
13h	45h	锁住和开锁驱动器	EBIOS

对于带有可移走媒质的硬盘驱动器来说，该功能允许锁住、开锁或检查一个驱动器的当前锁状态。锁住的驱动器将禁止操作弹射功能（功能 AH=46h）。每个可移走的驱动器都带有一个锁计数器，支持驱动器的该计数值多达 255。BIOS 保存该锁计数器。如果向一个驱动器发了三条锁命令，那么在驱动器真正开锁之前必须发三条开锁命令。如果对一个已处于开锁状态的驱动器发开锁命令，那么就会返回“驱动器未锁上”错误码（AH=B0h）。如果超过了驱动器所允许的锁计数，那么就会返回一个“锁计数越界”错误码（AH=B4h）。

调用：　ah=45h
al=操作（值 3~FFh 无效）
0=锁上驱动器的卷标（增加锁计数）
1=开锁驱动器的卷标（减少锁计数）
2=返回锁住/开锁状态
dl=驱动器号（80h+从零开始的硬盘驱动器号）

返回：　如果成功
进位=0
ah=0
al=0　驱动器目前未锁住
1　驱动器目前锁住
如果失败
进位=1
ah=返回状态（参看表 11-6）

中断	功能	描述	平台
13h	46h	弹出媒质请求	EBIOS

对于一个带可移走媒质的驱动器来说，如果未锁住当前的驱动器该功能会弹出媒质。参看功能 45h 以更多地了解驱动器的锁住和开锁操作。

如果选中的可移走媒质驱动器带有一个未锁住的媒质，该 EBIOS 会触发中断 15h 功能 52h。通常，操作系统会钩住中断 15h 功能以便在弹出之前清空缓冲区，或者防止发生弹出。如果中断 15h 钩没有禁止弹出功能，那么它会命令驱动器弹出媒质。

在一系列错误条件下，弹出请求可能会失败。这些错误包括：试图弹出不可移走的媒质驱动器（AH=B1h）、可移走的驱动器没有媒质（AH=31h）、锁住了可移走的媒质驱动器（AH=B1h）、硬件失败（AH=B5h），或者从中断 15h 功能 52 钩提供了一个错误的返回码。

调用：　ah=46h
　　al=0
　　dl=驱动器号（80h+从零开始的硬盘驱动器号）
返回：　如果成功
　　进位=0
　　ah=0
　　如果失败
　　进位=1
　　ah=返回状态（参看表 11-5）

中断	功能	描述	平台
13h	47h	扩展的作找	EBIOS

类似于标准的BIOS 功能 7，将磁盘的磁头移动到磁盘地址包中（参看表 11-10）指定的位置。该功能不使用下列三个值域：要传输的扇区数目、内存偏移量和内存段址。

调用：　ah=47h
　　dl=驱动器号（80h+从零开始的硬盘驱动器号）
　　ds:si=指针，指向磁盘地址包（参看表 11-10）
返回：　如果成功
　　进位=0
　　ah=0
　　如果失败
　　进位=1
　　ah=返回状态（参看表 11-6）

中断	功能	描述	平台
13h	48h	获取扩展驱动器参数	EBIOS

获取扩展驱动器参数表信息（参看表 11-12）。必须在内存中为该功能提供一个缓冲区。

表的第一个字必须初始化为用户提供缓冲区的最大大小。小于 26 的缓冲区不会成功。缓冲区大小为 26 到 29 时会向表中装入 26 字节的信息。如果指定缓冲区的大小为 30 字节或更多，当 EBIOS 支持增强磁盘驱动器（参看中断 13h，功能 41h）时会返回 30 字节的信息，否则会返回前 26 字节的信息。

表 11-12　扩展驱动器参数

偏移量	大　小	描　　述		
0	字	缓冲区大小，至少 26 字节长（参见上文）		
2	字	信息标志		
		位	15~7=0	未使用
			6=0	媒质现在位于驱动器中
			1	没有要移走的媒质，磁柱、磁头以及扇区值都设为最大（仅可移走媒质）
			5=1	驱动器可锁住（仅可移走媒质）
			4=1	驱动器支持改变线（仅可移走媒质）
			3=1	驱动器支持带检查写
			2=1	设备可移走
			1=1	在偏移量 8 和 12 处的磁头和扇区值有效
			0=1	EBIOS 自动避免 DMA 边界错误
4	双字	可寻址磁柱的块数。该值比从零开始计数的最大磁柱号大 1		
8	双字	可寻址磁头的总数。该值比从零开始计数的最大磁头号大 1		
12	双字	每磁道的扇区数。该值与从 1 开始计数的最大扇区号相同		
16	四字	可寻址的扇区的总数，这是一个 64 位值		
24	字	每扇区的字节数，通常是 512		
26	双字	可任选的段：偏移量指针，指向增强的磁盘驱动器配置参数。只有当该表的偏移量 0 设置为 30 或更大，并且支持增强磁盘驱动器（由中断 13h 功能 41h 指示）时，才会返回该值。返回值 FFFF:FFFFh 指示该指针无效		

调用：　ah=48h
　　dl=驱动器号（80h+从零开始的硬盘驱动器号）
　　ds:si=指针，指向增强驱动器信息表（参看表 11-12）
返回：　如果成功，就填充该表，并且
　　进位=0
　　ds:si=指针，指向增强驱动器信息表（参看表 11-12）
　　如果失败
　　进位=1

ah=6（这个普通的错误码表示改变了可移走媒质）
如果失败
进位=1
ah=返回状态（参看表 11-5）

中断	功能	描述	平台
13h	49h	获取扩展磁盘改变状态	EBIOS

查看是否改变了一个可移走媒质驱动器的媒质。媒质改变在任何时候都表示移走了媒质，或者对驱动器开锁后又重新锁住驱动器，即使这时没有移走媒质。

调用：ah=49h
dl=驱动器号（80h+从零开始的硬盘驱动器号）
返回：如果成功，并且没有激活改变线（没有发生媒质改变）
进位=0
ah=0
如果成功，并且激活了改变线
进位=1
ah=6（这个普通的错误码表示改变了可移走媒质）
如果失败
进位=1
ah=返回状态（参看表 11-5）

中断	功能	描述	平台
13h	4Ah	启动磁盘模拟	可引导的 CD-ROM

向 BIOS 传递一张值表，该 BIOS 支持可引导的 CD-ROM 扩展，版本 1.0 或更晚。表 11-13 用来指定所用的驱动器字母。参看可引导的 CD-ROM 说明书以了解从 CD-ROM 驱动器引导的更多细节。

表 11-13 可引导 CD-ROM 说明表

偏移量	大 小	描 述		
0	字节	本包的大小，单位字节。目前包有 13 字节		
1	字节	将 CD-ROM 模拟成哪种引导媒质：		
		位	7=1	图标含有 SCSI 驱动文件
			6=1	图标含有 IDE 驱动文件
			5=0	未使用
			4=0	未使用

续表

偏移量	大　小	描　　述		
			3=x	模拟类型
			2=x	0=无模拟
			1=x	1=1.2MB 软盘
			0=x	2=1.44MB 软盘
				3=2.88MB 软盘
				4=硬盘（C:）
2	字节	CD-ROM 驱动器的控制器号（即：0=第一个控制器）		
4	双字	磁盘图标开始的设备扇区号（逻辑扇区号，第一个扇区号是 0）		
8	字	设备说明		
		如果是 IDE 设备（本表的字节 1 的第 6 位=1）		
		位	15~1=0	未使用
		位	0=0	如果 CD-ROM 设置为主盘
			1	如果 CD-ROM 设置为从盘
		如果是 SCSI 设备（本表的字节 1 的第 7 位=1）		
		位	15~8=0	总使用
		位	7~0=x	CD 设备的 SCSI 逻辑单元号（LUN）
Ah	字	可任选的高速缓存段址（如果不是零，那么在指定的段址，偏移量为 0 处使用用户提供的 3KB 缓冲区，以供缓冲区 CD-ROM 读操作）		
Ch	字	引导装入段址——从设备装入了引导图标的段址。只有在触发中断 13h 功能 4Ch 时才使用该值。零值会在 7C0:0h 处装入图标		
Eh	字	扇区计数——在装入段处装入内存的扇区总数。只有在触发中断 13h 功能 4Ch 时才使用该值		
10h	字节	低磁柱值——来自中断 13h 功能 8 的 CH 寄存器值		
11h	字节	扇区和高磁柱值——来自中断 13h 功能 8 的 CL 寄存器值		
12h	字节	磁头计数——中断 13h 功能 8 返回的 DH 寄存器值		

调用：　　ax=4A00h

ds:si=指针，指向供 CD-ROM 引导的值表（参看表 11-13）

返回：　　进位=0，正在模拟指定的驱动器

1，系统不处于模拟模式

ah=返回状态（参看表 11-6）

中断	功能	描述	平台
13h	4Bh	终止磁盘模拟	可引导的 CD-ROM

结束一个引导设备的 CD-ROM 模拟。软盘和硬盘驱动器恢复到正常编号状态，就好像

没有用 CD-ROM 引导一样。

调用：　　ah=4Bh
　　al=子功能
　　dl=要终止的驱动器号，使用 7Fh 可以终止所有驱动器
　　ds:si=指针，指向一个空表的值（参看表 11-13）
返回：　　进位=0，如果释放了系统
　　1，系统不处于模拟模式
　　ah=返回状态（参看表 11-6）
　　ds:si=指针，指向值的完成表（参看表 11-13）

AL=	终止磁盘模拟的子功能命令
0	终止模拟并返回状态
1	只返回状态

子功能	描述	中断	功能
AL=0	终止磁盘模拟并返回状态	13h	AH=4Bh

结束 CD-ROM 引导模拟，并将驱动器返回到普通的编号状态。还返回状态以及在 DS:SI 处返回一个值的完成表。

子功能	描述	中断	功能
AL=1	只返回状态	13h	AH=4Bh

返回状态以及在 DS:SI 处返回一个值的完成表。不改变模拟状态。

子功能	描述	中断	功能
13h	4Ch	启动磁盘模拟和引导	可引导的 CD-ROM

使用传递值表，通过将 CD-ROM 模拟成软盘和硬盘驱动器，来从 CD-ROM 设备重新引导系统。只有当没有一个指针指向有效引导图标时该功能才返回。

调用：　　ax=4C00h
　　ds:si=指针，指向供 CD-ROM 引导的值表（参看表 11-13）
返回：　　进位=1　系统引导失败
　　ah=返回状态（参看表 11-6）

子功能	描述	中断	功能
13h	4Dh	返回引导分类	可引导的 CD-ROM

从 CD-ROM 引导分类中获得要求的扇区数目。参看表 11-14 可了解必须由该功能传递的值。该分类与用户缓冲区的大小有关，且该分类可以一次传递一个扇区。

表 11-14 引导分类请求表

偏移量	大 小	描 述
0	字节	本包的大小，单位是字节。目前包的大小是 8 字节
1	字节	要传输的扇区数目
2	双字	用户指针（段址：偏移量），指向引导分类传送到的地址
6	字	要传输的第一个扇区（第一次调用时置 0）

调用：　　ax=4D00h
　　ds:si=指针传送引导分类值的表（参看表 11-14）
返回：　　进位=0　如果传送成功
　　1　传送失败，因为设备不是一个模拟 CD，或者表 11-14 的值越界。
　　ah=返回状态（参看表 11-6）

中断	功能	描述	平台
13h	4Eh	设置扩展硬件配置	EBIOS

为设备设置各种硬件特性。如果同一个 IDE 控制器附带了两个驱动器（主和从），为主驱或从驱所做的某些设置也会影响同一控制器上的其他驱动器。如果命令影响了指定驱动器之外的其他驱动器，所有的子功能会返回 AH=1。

目前，提供七个子功能：

AL=	描述
0	开放预取
1	禁止预取
2	设置最大 PIO 传输模式
3	设置 PIOS 模式 0
4	设置缺省的 PIO 传输模式
5	开放中断 13h DMA 最大模式
6	禁止中断 13h DMA

子功能	描述	中断	功能
AL=0	开放预取	13h	AH=4Eh

该子功能开放 IDE 预取特性。在出现在任何读时，预取特性从磁盘读取额外的扇区。例如，如果要求扇区 X，控制器会读取扇区 X，并且可能向控制器的缓冲区读取 X 后面的

扇区。有可能下一个要求读的扇区是 X+1，现在控制器的内存中含有该扇区就可以立即响应了。一般地，控制器的缓冲区足够大，可以装下一个完整磁道的扇区。

另一个有关使用预取技巧的目的是为了提高速度。当驱动器移动到指定的磁道时，平均起来控制器必须等待磁盘的半个旋转周期来达到指定的扇区并使之可读。由于读一块连续的扇区很常见，一旦磁头位于指定的磁道上时控制器就会开始读。当磁盘旋转到指定的扇区时，扇区就被读进了磁道缓冲区。在传送所期望的扇区时，也传送了后面的其他扇区。这样控制器不需要等待就可以进一步读取扇区，并可以迅速地传送保存在磁道缓冲区中的扇区。

调用：
ax=4E00h
dl=驱动器号（80h+从零开始的硬盘驱动器号）

返回：
如果成功
进位=0
ah=0
al=0，如果命令只影响指定的驱动器
1，如果命令会影响其他的驱动器
如果失败
进位=1
ah=返回状态（参看表 11-6）

子功能	描述	中断	功能
AL=1	禁止预取	13h	AH=4Eh

该子功能禁止了 IDE 的预取特性。参看开放预取特性以更多地了解预取特性。

调用：
ax=4E01h
dl=驱动器号（80h+从零开始的硬盘驱动器号）

返回：
如果成功
进位=0
ah=0
al=0，如果命令只影响指定的驱动器
1，如果命令会影响其他的驱动器
如果失败
进位=1
ah=返回状态（参看表 11-5）

子功能	描述	中断	功能
AL=2	设置最大 PIO 传送模式	13h	AH=4EH

允许控制器提供的最高性能 PIO 模式，数据在控制器和系统之间通过输入输出指令传送。此功能禁止驱动器使用 DMA。

调用：　ax=4E02h
　　dl=驱动器号（80h+从零开始的硬盘驱动器号）
返回：　如果成功
　　进位=0
　　ah=0
　　al=0，如果命令只影响指定的驱动器
　　1，如果命令会影响到其他的驱动器
　　如果失败
　　进位=1
　　ah=返回状态（参看表 11-6）

子功能	描述	中断	功能
AL=3	设置 PIO 模式 0	13h	AH=4Eh

开放控制器所提供的最低速编程 I/O。数据在控制器和系统之间通过输入输出指令传送。该功能也会禁止驱动器使用 DMA。

调用：　ax=4E03h
　　dl=驱动器号（80h+从零开始的硬盘驱动器号）
返回：　如果成功
　　进位=0
　　ah=0
　　al=0，如果命令只影响指定的驱动器
　　1，如果命令会影响其他的驱动器
　　如果失败
　　进位=1
　　ah=返回状态（参看表 11-5）

子功能	描述	中断	功能
AL=4	设置缺省 PIO 模式	13h	AH=4EH

在控制器中设置缺省 PIO 模式。数据在控制和系统之间通过输入、输出指令传输。此功能会禁用驱动器的 DMA。

调用：　ax=4E04h
　　dl=驱动器号（80h+从零开始的硬盘驱动器号）
返回：　如果成功

进位=0
ah=0
al=0，如果命令只影响指定的驱动器
al=1，如果命令会影响其他的驱动器
如果失败
进位=1
ah=返回状态（参看表 11-5）

子功能	描述	中断	功能
AL=5	开放中断 13h DMA 最大模式	13h	AH=4Eh

开放控制器中最高速 DMA 模式。数据在控制器和系统之间通过 DMA 控制器的一个通道传送。对于多任务操作系统来说，这是最好的选择。一旦设置了 DMA 操作，就可以向或从内存传送扇区而不用 CPU 参与。该功能也禁止了任何 PIO 模式设置。

调用：ax=4E05h
dl=驱动器号（80h+从零开始的硬盘驱动器号）
返回：如果成功
进位=0
ah=0
al=0，如果命令只影响指定的驱动器
1，如果命令会影响其他的驱动器
如果失败
进位=1
ah=返回状态（参看表 11-6）

子功能	描述	中断	功能
AL=6	禁止中断 13h DMA	13h	AH=4Eh

关闭驱动器的磁盘 DMA 传输模式。在访问驱动器之前必须设置三个 PIO 选项中的一个（子功能 2、3 或 4）。

调用：ax=4E06h
dl=驱动器号（80h+从零开始的硬盘驱动器号）
返回：如果成功
进位=0
ah=0
al=0，如果命令只影响指定的驱动器
1，如果命令会影响其他的驱动器
如果失败

进位=1
ah=返回状态（参看表 11-6）

中断	功能	描述	平台
15h	52h	可移走媒质弹出	EBIOS

在中断触发 13h 功能 46h 弹出可移走媒质时，EBIOS 会触发该功能。不能由程序调用它。程序或操作系统可以挂起该中断，这样在试图弹出可移走媒质时可以改变它。

程序可以执行某些操作，例如清空磁盘高速缓存，促使弹出发生。程序也可以阻止弹出。如果正在执行重要的读或写，这一点就很有用。为了阻止弹出，必须设置两个错误返回码，B1h 表示错误条件“驱动器中卷标锁住”，B3h 表示“卷标正在使用”。

调用：　ah=52h
　　dl=驱动器号（80h+从零开始的硬盘驱动器号）
返回：　如果成功
　　进位=0
　　ah=0
　　如果失败
　　进位=1
　　ah=B1h，驱动器中的卷标锁住
　　B3h，卷标正在使用

中断	功能	描述	平台
4Fh	8100h	SCSI 命令，通用访问方法	SCSI

许多较新的 SCSI 卡使用通用访问方法（CAM）来提供对 SCSI 卡与供应商无关的访问。本功能用于向 SCSI 卡发送命令。仅 SCSI 硬盘适配卡 BIOS 或 SCSI 设备驱动文件才支持该功能。

该功能指向一个 CAM 控制数据块，后者为该命令定义了一些命令和参数。要了解 CAM 控制块的当前细节，请参考附录 C 中的文档《SCSI-2 通用访问方法传输》（SCSI-2 Common Access Method Transport）和《SCSI 接口模块》（SCSI Interface Module）。

调用：　ax=8100h
　　es:bx=指针，指向 CAM 控制块
返回：　ah=0，如果成功。任何其他的值都表示出错。

中断	功能	描述	平台
4Fh	8200h	SCSI 存在性检查，通用访问方法	SCSI

许多较新的 SCSI 卡使用通用访问方法（CAM）来提供对 SCSI 卡与供应商无关的访问。

很少有系统提供 SCSI 支持 BIOS 并把它作为系统的一部分，但是通过 SCSI 卡上的 BIOS，系统可支持 SCSI。该功能检查是否存在 SCSI 卡，以及是否支持 CAM 声明。如果支持 CAM，就可以通过中断 4Fh，功能 8100h 传送命令。

调用：　ax=8200h
　　cx=9765h
　　dx=CBA9h
返回：　如果安装了 SCSI 卡：
　　ah=0
　　cx=9ABCh
　　dx=5678h
　　es:di 指针指向 SCSI-CAM 串

中断	描述	平台
76h	硬盘操作完成	AT/EISA

对于 AT 和 EISA 系统来说，在完成硬盘控制器命令和相关的数据传送时，会发生硬盘操作完成中断。控制器向 IRQ14 发信号。如果没有激活更高优先级的中断，并且开放了该中断，就会调用中断 76h 的服务程序。

该程序首先设置磁盘操作完成标志，指示活动的硬盘操作完成。要做到这一点，可通过 I/O 端口 20h 和 A0h 将 BIOS 40:8Eh 处的数据项设置为 FFh。触发普通的结束中断命令 20h 来清除当前的中断。于是重新开放了中断。该程序也调用中断 15h，AX=9100h 来向系统发送一条操作完成信息。其他的软件可以钩住中断 15h 以便每次磁盘操作完成时改变它。

在中断发生之前，磁盘 BIOS 循环查看磁盘操作完成标志。由控制器触发中断 76h。中断 76h 处理程序设置操作完成标志并退出。系统 BIOS 服务监测到操作完成标志的变化，接着会获取控制器状态，并返回中断 13h 的调用程序中，以完成磁盘操作。

多任务操作系统不会容忍 BIOS 的密循环等待。幸运的是，BIOS 提供了一种方法在等待期间将控制交给操作系统。要了解这种方法的细节，请参看第 13 章，系统功能中的中断 15h，功能 90h，设备忙。

中断	描述	平台
76h	操作完成	PS/2

PS/2 的操作完成类似于 AT 系统上的操作完成中断，磁盘控制器向 IRQ4 发信号。如果没有激活更高优先级的中断，并且开放了中断，就会调用中断 76h 的服务程序。

与 AT 不同的是，该程序可用于多种用途，硬盘控制器只是目前唯一的使用者。要实现该功能时，首先检查硬盘控制器，看控制器是否产生了中断。如果控制器正触发一个中断，端口 3512h 处的基本状态寄存器的第 0 位就会被设置。如果没有触发中断，系统 BIOS

程序会返回。允许其他任何设备挂起中断 76h 来执行预期动作。

如果控制器触发了该中断，就会从端口 3513h 中获得控制器中断状态，并将其保存在扩展 BIOS 数据区（EBDA）的偏移量 72h 处。设置 EBDA 偏移量 73h 处字节的第 7 位可以设置操作完成标志。

两个中断控制器都会触发普通的结束中断命令 20h 来清除当前的中断。于是重新开放了中断。该程序也调用中断 15h，AX=9100h 来向系统发一条操作完成信息。其他软件可以挂起中断 15h 以便每次磁盘操作完成时改变它。

在发生该中断之前。磁盘 BIOS 会密循环查看磁盘操作完成标志。由磁盘控制器触发中断 76h，它设置操作完成标志并退出。磁盘 BIOS 服务程序重新获得控制权，监测标志位的设置，然后进入下一步。该中断在 PS/2 上的用法和在 AT 设计上的用法有点不同。参看本章后面的“典型的读扇区操作”部分以及“PS/2 EISA 类型控制器和 BIOS”小节，以了解其他细节。

磁盘 BIOS 数据

磁盘 BIOS（中断 13h）在 BIOS 数据区为其操作使用大量的字节。表 11-15 列出了这些数据的地址和功能。PS/2 在扩展 BIOS 数据区中也使用了许多数值。第 6 章讨论了其他细节。

表 11-15　硬盘 BIOS 数据

地　址	大　小	描　　述
40:42h	4 字节	磁盘控制器返回状态字节——对于带有 PC/XT 类型控制器的系统来说，按下来的另外四个字节也用于在检测错误时保存控制器的传感数据。参看请求传感状态下的端口 320h 以了解这些字节的更多细节。软盘控制器也在这些字节中保存临时信息
40:74h	字节	上次操作的状态——在完成一次硬盘操作时，来自操作的状态保存在此处。前面的表 11-5 描述了状态代码。该字节也用在检测到 ECC 错误时保存临时的信息，但是是改正后的信息。ECC 错误保存在此处，但是操作会继续进行下去。操作结束时，如果任何一个扇区发生了 ECC 错误，由中断 13h 功能返回的状态会被置为 11h，订正 ECC 错误
40:75h	字节	附带硬盘的数目——该字节指示了 POST 检测到的硬盘数目。老式的 BIOS 最多支持两个驱动器，而新式的 BIOS 通常支持多达四个驱动器。自带 BIOS 的磁盘控制器只受限于控制器的容量，有些 BIOS ROM 带有 bug 会不正确地把 CD-ROM 驱动器计作一个硬盘驱动器
40:76h	字节	硬盘控制字节。该字节为硬盘 BIOS 保存了临时的控制标志。它只适用于某些硬盘 BIOS。对那些该字节的使用者来说，似乎没有标准的位分配

续表

地 址	大 小	描 述
40:77h	字节	硬盘端口偏移量——该字节临时保存了指向硬盘端口号的偏移量。它只用于 PC/XT 硬盘 BIOS 以及其他一些早期的硬盘适配器上
40:8Ch	字节	硬盘状态——来自硬盘控制器的实际状态保存在这里。某些 BIOS 不支持该功能
40:8Dh	字节	硬盘错误——来自硬盘控制器的错误寄存器保存在这里。某些 BIOS 不支持该功能
40:8Eh	字节	硬盘任务完成标志——当磁盘控制器任务开始时，硬盘 BIOS 将该值设置为 0。任务完成时，控制器会向 IRQ14 发信号，触发中断 76h。中断 76h 的处理程序就将这个字节设置为 0FFh。PC/XT BIOS 不使用这种完成的方式，也不使用这个字节

向磁盘控制器发送命令

磁盘控制器有一系列命令来执行不同的操作，例如格式化某个磁道，或者写一个扇区。命令通常作为命令数据块的一部分发送，该命令数据块用来描述执行相应的动作。在执行完该命令时，控制器会触发一个中断。在 PC 和 XT 系统上，磁盘完成信息使用中断 Dh。在 AT 以及所来后来的系统上，使用中断 76h 来指示命令完成。然后，硬盘 BIOS 处理程序从控制器中获取数据块，来检查命令是否成功完成，还是出现了错误。

一个典型的读扇区操作

这一部分很长，但是有助于更好地理解磁盘操作，它将磁盘系统的许多方面联系在一起，包括 BIOS 和控制器的最低层次。本部分涉及三种最常见的系统，即 PC/XT、AT/EISA 以及 PS/2 ESDI 硬盘系统。所有的三种系统除了初始部分的操作相同之外，剩下的操作大不相同，所有情况都假定只有一个主控制器。如果 BIOS 支持另外的控制器，则通过可替代的端口访问该控制器。

通用的读扇区操作

首先，我们假定重启了控制器，为正在使用的驱动器装入了表参数，并且重校了驱动器，重校仅仅只是将磁头移动到磁柱 0。

操作系统或应用程序必须将其文件系统转化为硬盘地址，硬盘地址由磁柱号、磁头号以及一个磁道上的扇区号组成。一旦确定了磁盘上的位置，就会调用硬盘服务程序，中断

13h，功能 2 来读扇区。

中断 13h 功能 2 传递从设备上读取数据的磁盘地址，该地址由磁柱、磁头号和扇区组成。要读取的扇区数和从磁盘读取数据后的保存地址的指针都传递给功能 2。当然，也要指定驱动器号，0 或更大，并将第 7 位置 1 表示是对硬盘操作，而不是对软盘操作。

完成读操作所须的步骤与系统使用的控制器有关，我们将逐个分析三种常见的控制器，PC/XT、AT 和 PS/2 EISA。

PC/XT 类型的控制器的 BIOS

检查驱动器号有效之后，BIOS 为一条读命令创建一个命令数据块。命令数据块有下述内容：

字节 0——命令（值 8 表示读）
字节 1——驱动器和磁头号
字节 2——扇区号和磁柱号（高 2 位）
字节 3——磁柱号（低 8 位）
字节 4——待读扇区的数目
字节 5——控制字节（值 5 表示普通的重试和普通的步进计时）

接着检查要读取的扇区，以确保它没有超过 DMA 操作的范围。一次最多可传输 64K。如果在操作范围内，就设置通道 3 来开放传送所要求的字节数，从磁盘适配卡上向内存指定地址传送。

通过向端口 322h 发送任何一个值，硬盘控制器选择脉冲，都可以将控制器设置为忙状态。然后向端口 323h 发送值 3 来清除 DMA 和中断屏蔽。这样就允许在合适的时候触发 DMA 和中断请求。从端口 321h 检查控制器的硬件状态，来保证控制器准备好。必须设置忙锁存器，并且控制器必须等待命令输入和请求准备好。这就意味着准备好时状态字节的低 4 位应该是 0Dh。

磁盘控制器准备好时，以前创建的命令数据块的六个字节会通过端口 320h 一次一个地传送到控制器。如果收到了这六个字节的确认信息，会再次检查硬件状态。必须清除位 0，它表示控制器没有接收到更多字节。如果没有清除该位，控制器会出现问题。

通过确认 DREQ3 线，控制器为通道 3 触发一个 DMA 请求。通过向 DMA 屏蔽寄存器，端口 Ah，发送值 3，BIOS 会解除对 DMA 通道 3 的屏蔽。同时，BIOS 清除中断控制器的屏蔽位，这样将允许 IRQ5，然后 BIOS 将进入一个循环等待数据传输完成。

这个时候，各种硬件信号会造成控制器从磁盘读数据和 DMA 通过数据总线向内存传送数据。传送完所要求的扇区数据后，控制器会向系统发一个 PC/XT 中断请求 5（IRQ5），硬盘操作完成中断 0Dh。所有这些工作由硬件完成而无需软件的介入。

BIOS 检查传送完成并执行最后的清理工作，向端口 323h 写入零来设置控制器和 DMA 的中断屏蔽。从端口 320h 读取控制器状态来查看该命令是否出错。如果没有出错，就完成操作并返回成功代码 AL=0。如果发生了错误，控制器就接收到错误信息，并在 AL 中将错误状态传递给用户。现在再看是不是很简单！

AT/IDE 类型控制器和 BIOS

硬盘 BIOS 检查所期望的驱动器是否有效，并检查操作指定的数据是否小于 64K。在本例中，如果请求的扇区多于 128 个，就会触发 DMA 越界错误，即使 AT 控制器不使用 DMA！等一会儿我会解释 BIOS 如何略过 DMA。

许多系统只允许一次操作最大传送 64K 数据。所有的控制器都能传输 128K 数据，而只有某些 BIOS 才支持传输 128K 数据。对于一般的编程使用时，64K 或更少是唯一可以相信的安全大小，因为在不同的系统 BIOS 使用多于 64K 时，会产生许多 bug，参看本章后面的 DISKTYPE 程序以更多地了解这个问题。

一个很小但是很重要的问题是对数据写放缓冲区起始偏移量的限制。如果偏移量的最低位不是零，那么只能传送 127 个扇区。请求传送 128 个扇区会发生相同的 DMA 边界错误。这一点很重要，因为传送 64K 是通过增加偏移量来实现的。奇怪的是，偏移量的高三位并不重要。64K 完全接受值为 FFF0h 的偏移量。在这种情况下，以磁盘读取的数据将从段址：FFF0h 的字节开始写入，大小为 64K。在传送之前，系统 BIOS 会将该地址规范化为段址+FFF:0000h，不会发生段址回跳现象。

通过读状态端口 1F7h 和检查第 7 位，来检查控制器以保证其不忙。控制器准备好时会清除该位。现在就可以磁头和驱动器选择寄存器端口 1F6h，并向其中装入驱动器的值。然后重新检查端口 1F7h 的值，不过这次检查位 6 的值以确认选择的驱动器已准备好。驱动器准备好时第 6 位是 1。必须清除写错误位 5，因为如果驱动器指出前面的操作有写错误，控制器是不会接受命令的。

然后系统 BIOS 获取该驱动器的驱动器参数。驱动器 0 的表指针保存在中断向量 41h 中，而驱动器 1 的保存在向量 46h 中，来自磁盘参数表偏移量 8 处的控制字节被装入到适配器控制器寄存器，端口 3F7h。这将告诉硬盘适配器，驱动器是否使用 8 个以上的磁头。

如果操作是一条写命令，有必要装入写预补偿寄存器端口 1F1h。该值从磁盘参数表偏移量 3 处获得。控制器的读命令不使用写预补偿寄存器，如果装入它也会给予忽略。所有 IDE 类型的驱动器会忽略该值。

其他的控制器寄存器要装入待读取的扇区数，以及由起点扇区、磁柱号、磁头号和驱动器号组成的磁盘地址。这些寄存器分布在端口 1F2h 或 1F6h。磁头和驱动器选择寄存器通常会将第 7 位置 1 来执行错误检查和更正（ECC）。磁头和驱动器选择寄存器的第 6 位和第 5 位分别设置为 0 和 1，来指定 512 字节的扇区。尽管控制器可使用其他扇区的大小，但是在所有 PC 系统上只使用 512 字节的扇区。大多数 AT 系统只支持 1,024 个磁柱、64 个

扇区和 16 个磁头。只有在清楚地知道控制器会支持它时，才可以使用更大的值。IDE 驱动器可以支持 1,024 个以上的磁柱（参看前面部分，大型 IDE 驱动器，以了解 BIOS 限制）。

要装入的寄存器归纳如下：

寄存器	名称	装入
1F1h	写预补偿	写预补偿的磁柱
1F2h	扇区计数	要读取的扇区数
1F3h	扇区号	起点扇区号
1F4h	磁柱低	起点磁柱号，低 8 位
1F5h	磁柱高	起点磁柱号，高 8 位
1F6h	驱动器与磁头	驱动器号 0 或 1，磁头号 0~15，位 7~5=101

在命令控制器开始之前，还需要最后一步，即必须清除 IRQ14 的中断控制器屏蔽位。它允许系统在控制器为每个扇区传送准备好时加以确认，我们很快就会阐明这一点。

为了开始读操作，如果正确地设置了所有的控制器寄存器，就可以往端口 1F7h 处的命令寄存器中写入读命令 20h。一旦控制器收到该命令，就会从驱动器控制器的扇区缓冲区中读取数据。这时，BIOS 进入循环以等待设置操作完成标志。

读完了一个完整的扇区后，控制器会触发数据准备好中断 IRQ14h。这时，中断 76h 处理 IRQ14，它只是简单地设置操作完成标志。系统 BIOS 检测到设置了该标志，并会直接从控制器的扇区缓冲区读取 256 个字。读 16 位端口 1F0h 来获取该数据。这一点和 PC/XT 及 PS/2 的 BIOS 操作有很大的不同。它们使用纯硬件的方法通过 DMA 传送数据，而 AT 只是从端口 1F0 中用一个软件循环来读取数据字。目前大多数的系统支持 32 位数据传送，这时可配置硬件来使传送速度加倍。

读完数据后，检查是否出错。读状态端口 1F7h 要检查不同的错误条件。任何致命的错误都要求可以退出程序。如果检测到了 ECC-更正错误，可以安全地忽略它。可以设置一个标志以向用户返回状态，指示检测到了 ECC 错误，但是业已更正。

如果没有出现致命的错误，就会减少起初的扇区计数值。如果扇区计数值仍不是零，就会读取下一个扇区。这种情况下软件不改变控制器中的扇区计数值，但是会减少中断 13h 调用传递过来的原始值副本。

从磁盘读取了所有期望的扇区并写入到内存之后，系统会返回操作的状态。

PS/2 EISA 类型的控制器和 BIOS

同前面两种方法类似，选中一个驱动器号后，BIOS 处理程序为读命令创建一个命令数据块。这个命令数据块包括下述内容：

字 0——命令，从驱动器 0（4601h）或驱动器 1（4621h）读取数据

字1——要传输的扇区数

字2——相对扇区寻址，最低有效位字

字3——相对扇区寻址，最高有效位字

与上面两种方法不同的是，PS/2 BIOS 必须将磁柱、磁头和扇区地址转化为相对扇区地址，后者是 PS/2 ESDI 控制器所要求的。我已经仔细查看过的 PS/2 系统只支持最大驱动器组合为 1,024 个磁柱、64 个磁头和 32 个扇区，或者总计 1,073MB。下面的等式可以计算相对扇区地址：

相对扇区地址=（磁柱号×2048）+（磁头号×32）+（扇区号）

另外，必须开启 PS/2 的硬盘忙指示灯。设置端口 92h 的最高位，BIOS 可以激活磁盘忙指示灯。

这时，控制器会装入该命令，处理它，然后清除它。如果你对这些细节不感兴趣，你可以略过下面几段。

定义了命令数据块后，通过向基本控制寄存器端口 3512h 发送值 1，可以清除控制器的中断屏蔽。接着读取基本控制寄存器端口 3512h，并检查控制器是否不忙，如果不忙，位 4 必须为零。

对注意寄存器，端口 3510h 发一个命令请求。发送值 2 将指示控制器使用驱动器 0 执行该命令。值 22h 用来指示使用驱动器 1。

如果控制器工作正确，它就应该处于忙状态。读取端口 3512h 的基本状态来检查现在是否已设置第 4 位，忙标志位。也是这个寄存器，若清除了第 2 位表明控制器已准备好命令数据块。在读命令时，通常端口 3510h 向控制器发送四个字。在发送了每个字之后和发送下个字之前，BIOS 必须检查清除了基本状态寄存器的第 2 位。

BIOS 会指示系统正准备通过触发中断 15h 功能 AX=9000h 来等待接收该命令。然后，BIOS 进入一个计时循环来等待磁盘完成中断。

控制器会触发一个硬件中断请求 IRQ14 来标志接受了一条命令，IRQ14 会调用中断 76h。中断 76h BIOS 程序会设置 RAM 中的磁盘完成标志来指示操作完成。PS/2 系统会设置扩展 BIOS 数据区的字节 73h 的第 7 位。中断 76h 也会触发中断 15h 功能 AX=9100h 来提示其他相关的程序或操作系统，现在磁盘操作已经完成。

磁盘 BIOS 目前正在循环等待完成中断，这时会监视到设置了磁盘完成标志，然后读取中断状态寄存器，端口 3513h。如果控制器准备好传输，控制器会返回“数据传送准备好”中断状态。中断状态的低五位是 0Bh。

磁盘 BIOS 然后为数据传送设置 DMA 控制器。从磁盘传送数据到达的 RAM 地址被装入到 DMA 通道 5 中。向端口 3512h，基本控制器发送值 3，于是开始了具体的传送操作。它同时支持 DMA 请求和中断请求。

BIOS 再次标示系统正在等待完成数据的传送，磁盘 BIOS 触发中断 15h 功能 AX=9000h。BIOS 进入计时循环，等待磁盘完成中断，同时开始通过 DMA 在控制器和内存之间传送数据。具体的数据传送不牵涉到软件操作。

当传送完成或者出现问题时，控制器触发 IRQ14，中断 76h。中断 76h 的处理程序设置磁盘完成标志，并触发一个操作完成中断 15h，AX=9100h。

磁盘 BIOS 目前正在循环等待中断完成，这时会监视到设置了磁盘完成标志，然后读取中断状态寄存器，端口 3513h。如果操作成功，中断状态的第 5 位将是值 1。其他任何值都表示传输出了某个问题。

对于错误条件，读取基本状态寄存器端口 3512h，如果有另外的状态，就会设置第 3 位。可以从端口 3510h 读取状态字，每读完一个字后，就会检查基本状态寄存器，看是否还有状态字要读取。这些状态字代表了错误的真实特性以及错误的相对扇区地址。

为了完成操作，向注意寄存器端口 3513h 发送结束中断代码，值 2 用于驱动器 0，值 22h 用于驱动器 1。现在整个操作已经完成，返回到最初的调用状态。

警　告

在程序直接访问硬件所能执行的所有操作中，操作硬盘控制器是最复杂的操作之一。如果可能，应使用操作系统来访问硬盘（DOS 的通用 IOCTL 提供了许多有用的低级磁盘功能的话）。如果做不到这一点，那么应使用中断 13h 作为接口，因为中断 13h 会透明地处理不同控制器和磁盘类型之间的许多差别。

直接访问控制器要求你确切地知道系统使用的是哪种控制器。如果系统使用磁盘压缩，例如 DoubleSpace、SuperStor、Stacker 或者其他的此类产品时，应格外小心。尽管可以全部访问磁盘的内容，但是具体的数目可以按许多种特有的格式编码。直接磁盘访问会绕过这些压缩程序。

为了 BIOS 操作可靠，永远不要一次读取多于一个磁道的数据。当试图越过磁道的边界读取数据时，许多 BIOS 的操作不可靠。参看代码 11-2 为说明这个问题而进行的一次简单的测试。

代码例 11-1　磁盘驱动器检测

该程序检测是否带有指定的磁盘驱动器，并确定其最大容量。该程序应对普通和非标准的驱动器都能起作用。该程序还检测某些通用类型的控制器，例如老式的 PC/XT 控制器、新式的 AT 控制器、IDE、PS/2、ESDI 以及常见的 SCSI 控制器等。记住，某些 ESDI 和 SCSI 控制器没有自带的磁盘 BIOS，它们依靠系统 BIOS 来提供所有的服务。在这些情况下，磁盘驱动器检测程序会指出是否正在使用一个 AT 类型的控制器。

为了使用该程序获取总磁盘大小，只要将磁柱数、磁头数和每磁道的扇区数乘起来就可以了。

该程序包括在 SYSTYPE.ASMK 中，运行 SYSTYPE 来看 HDRVTYPE 子程序的运行结果，在某个系统上，与磁盘驱动器检测相关的 SYSTYPE 结果部分如图 11-5 所示。

系统分析	v2.00
硬盘驱动器 0:	IDE 控制器，激活了 LBA
	总大小=1.545MB（包括诊断磁柱）
	785 个磁柱，64 个磁头，每磁道 63 扇区
硬盘驱动器 1:	AT 或 IDE 类型控制器
	总大小=503MB（包括诊断磁柱）
	1,022 个磁柱，16 个磁头，每磁道 63 扇区

图 11-5　SYSTYPE 的部分结果

```
; ----------------------------------------------------------------------------
;    硬盘驱动器类型检测子程序
;        确定是否带有硬驱，可能类型以及
;        驱动器容量。
;
;        调用:    ds = cs
;                 dl = 要检查的驱动器，80h = 驱动器0，等
;
;        返回:          al = 驱动器类型
;                       0 = 没有驱动器或控制器
;                       1 = XT类型控制器
;                       2 = AT或IDE类型控制器
;                       3 = SCSI类型控制器
;                       4 = PS/2 SCSI类型控制器
;                       5 = 未知类型控制器
;                       6 = IDE控制器，LBA被激活
;                       cx = 磁柱总数（包括诊断磁柱）
;                       dx = 磁头总数（1~256）
;                       ah = 512字节扇区数（1~63）
;        使用的寄存器:  ax

; 程序的局部数据

hd_parm_seg     dw      0               ; 磁盘指针
hd_parm_off     dw      0               ;  参数表
hd_cylinder     dw      0               ; 表的临时号
ESDIbuf         db      1024 dup (0)    ; ESDI信息缓冲区
```

```
hdsktype proc    near
         push    bx
         push    es
         xor     bp, bp                  ; 驱动器临时类型
         cmp     dl, 80h                 ; 至少80h?
         jb      hdsk_skp1               ; 如果不是，则退出

; 首先获得驱动器数目

         push    dx
         mov     dl, 80h                 ; 驱动器0
         mov     ah, 8
         int     13h                     ; 读磁盘驱动器参数
         pop     ax
         add     dl, 7Fh
         cmp     al, dl                  ; 是这个号的驱动器吗?
         jbe     hdsk_skp2
hdsk_skp1:
         jmp     hdsk_none               ; 如果越界，则跳转

; 现在确定控制器/磁盘类型

hdsk_skp2:
         inc     bp                      ; bp=1 假定XT类型的驱动器
         mov     dl, al                  ; dl = 驱动器号
         push    dx
         mov     ah, 15h
         int     13h
         pop     dx
         cmp     ah, 1                   ; 无效的请求状态
         je      hdsk_skp4               ; 如果是，则是XT类型，并退出
         inc     bp                      ; 设置为AT类型
         cmp     ah, 3                   ; 确认磁盘有效
         je      hdsk_skp3
         jmp     hdsk_none               ; 如果不是，则退出
```

```
; 看这个驱动器是否不使用驱动器参数表，
;    如SCSI和某些ESDI驱动器

hdsk_skp3:
        inc     bp                      ; 假定是SCSI
        cmp     dl, 87h                 ; 如果大于87h，则假定是SCSI
        jbe     hdsk_skp5
hdsk_skp4:
        jmp     hdsk_info               ; 如果是，则跳转

hdsk_skp5:
        mov     bx, 4*41h               ; 假定指向向量41h
        cmp     dl, 80h                 ; 驱动器0?
        je      hdsk_skp6               ; 如果是，则跳转
        mov     bx, 4*46h               ; 指针指向向量46h
hdsk_skp6:
        xor     ax, ax
        mov     es, ax
        mov     si, es:[bx]             ; 参数表偏移量
        mov     ax, es:[bx+2]           ; 参数表的段值
        mov     [hd_parm_seg], ax       ; 保存指针
        mov     [hd_parm_off], si
        mov     es, ax                  ; ex:si指向表

;   现在，指针指向了磁盘驱动器表，但是有两种类型。
;   检查偏移量2处的串是否不是‘IBM’，
;   如果不是，就使用普通表（偏移量0处的磁柱）。
;   否则，使用老式的PS/2类型，磁柱位于偏移量19h处

        cmp     word ptr es:[si+2], 'BI'    ; 串‘IBM’?
        jne     hdsk_skp7               ; 如果不是，则跳转
        cmp     word ptr es:[si+4], ' M'
        jne     hdsk_skp7
        add     si, 19h                 ; 调整为PS/2类型
hdsk_skp7:
        mov     ax, es:[si]             ; 从表中获取磁柱数
```

```
        mov     [hd_cylinder], ax          ; 保存
        cmp     ax, 0                      ; 无效的磁柱号?
        jne     hdsk_skp8                  ; 如果有效（非零）则跳转
        jmp     hdsk_info                  ; 考虑SCSI

hdsk_skp8:
        dec     bp                         ; 假定为AT类型（2）
        push    dx
        mov     ah, 8
        int     13h                        ; 获取磁盘参数
        call    hdconvert                  ; 转化成有用的值
        cmp     ax, 1021                   ; 接近IDE限的尾部?
        jb      hdsk_skp8a                 ; 如果不是，则跳转
        cmp     [hd_cylinder], 1020
        jae     hdsk_skp8b                 ; 如果大于等于，则跳转
        jmp     hdsk_skp9                  ; 不可能是AT或IDE类型

hdsk_skp8a:
        cmp     ax, [hd_cylinder]          ; 它们位于3个磁柱内?
        ja      hdsk_skp9                  ; 若非，则跳转
        add     ax, 3
        cmp     ax, [hd_cylinder]          ; 它们位于3个磁柱内?
        jb      hdsk_skp9                  ; 若非，则跳转

        ; 是AT类型，检查是否激活了LBA（磁头数>16）

hdsk_skp8b:
        cmp     dx, 16                     ; 多于16个磁头
        jbe     hdsk_info2                 ; 如果非，则跳转
        mov     bp, 6                      ; 设置为AT类型，LBA开
hdsk_info2:
        pop     dx
        jmp     hdsk_info                  ; 如果是，可能是AT类型
```

由于磁柱数不匹配，所以不可能是AT类型

首先，我们进行老式的未来域SCSI测试

```
; 未来域SCSI测试——中断13h功能18h
;  将返回无效命令，DL>=80h，如果
;  没有未来域SCSI卡

hdsk_skp9:
        pop     dx
        inc     bp                      ; 假定为SCSI
        push    dx
        mov     ah, 18h
        int     13h                     ; 未来域SCSI?
        pop     dx
        jc      hdsk_skp10              ; 如果不是则跳转
        cmp     ax, 4321h               ; 确认号
        jne     hdsk_skp10
        jmp     hdsk_info

; 检查是否会是PS/2 ESDI驱动器

hdsk_skp10:
        push    cs
        pop     es
        push    dx
        mov     ax, 1B0Ah
        mov     bx, offset ESDIbuf
        int     13h                     ; 获取ESDI配置
        pop     dx
        jc      hdsk_skp11              ; 如果不是PS/2 ESDI则跳转
        cmp     bx, offset ESDIbuf
        jne     hdsk_skp11
        mov     ax, cs
        mov     bx, es
        cmp     ax, bx                  ; CS = ES吗?
        jne     hdsk_skp11              ; 如果不是则跳转
        inc     bp
        jmp     hdsk_info
```

```
; 一般SCSI测试（未指定磁盘）

hdsk_skp11:
        xor     ax, ax
        mov     es, ax
        cmp     word ptr es:[4*4Fh+2], 0    ; 向量有效?
        je      hdsk_skp12                  ; 若非，则跳转
        push    dx
        mov     ax, 8200h
        mov     cx, 8765h
        mov     dx, 0CBA9h
        int     4Fh                         ; 检查 SCSI CAM
        cmp     dx, 5678h
        pop     dx
        jne     hdsk_skp12                  ; 若非，则跳转
        cmp     ah, 0
        jne     hdsk_skp12                  ; 若非，则跳转
        cmp     cx, 9ABCh
        jne     hdsk_skp12                  ; 若非，则跳转
        jmp     hdsk_info

hdsk_skp12:
        mov     bp, 5                       ; 指示未知
                                            ; 控制器类型

; 获取指定驱动器的磁盘参数

hdsk_info:
        mov     ah, 8
        int     13h                         ; 读取磁盘驱动器参数
        call    hdconvert                   ; 转化成有用的值
        xchg    cx, ax                      ; 放入合适的寄存器中
        mov     bl, al                      ; 扇区数
        mov     ax, bp                      ; 获取驱动器类型
        mov     ah, bl                      ; 插入扇区数
```

```
        jmp     hdsk_exit

hdsk_none:
        xor     al, al                  ; 没有这个号的驱动器
        xor     cx, cx
        xor     dx, dx
hdsk_exit:
        pop     es
        pop     bx
        ret
hdsktype endp


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

;   硬盘转化
;
;       磁柱号保存在cx中。本程序正确地将它们
;       组合成一个10位磁柱号并保存在AX中。由
;       于有一个诊断磁柱，所以这个数据自然加1。
;       由于计数从零开始，所以还要加1。CL的最
;       高两位清零后获得扇区号。磁头号也加1，
;       因为它也是从零开始计数的。
;
;       调用:           ch = 磁柱号的低8位
;                       cl 的高两位是磁柱号的第9位和第8位
;                       dh = 磁头号
;
;       返回:           ax = 磁柱总数
;               cl = 扇区号
;               dx = 磁头总数（1~256）
;
;       用到的寄存器:   cx, dh

hdconvert proc  near
        mov     ax, cx                  ; 获取零碎磁柱号
        rol     al, 1
```

```
        rol     al, 1
        and     al, 3
        xchg    al, ah                  ; ax = 磁柱号
        and     cx, 3Fh                 ; 屏蔽，获得扇区号
        mov     dl, dh
        xor     dh,dh
        inc     dx                      ; 调整磁头计数
        ret
hdconvert endp
```

代码例 11-2　中断 13h 磁盘读测试

DISKTYPE 程序用来检查指定的 BIOS 是如何处理读取多达 128K 块的操作的。它并不仅仅作为磁盘系统的测试程序，更重要的是它指出了许多系统上常见的某些严重的 BIOS bug，在 http://www.infopower.com.cn 处提供了可执行代码和源代码。由于源代码不是很重要，所有我没有在本书中列出它。

该测试首先从磁盘读取 128K 的数据，一次一个扇区。这个操作总是成功的，除非驱动器真正存在很严重的问题，然后，DISKTEST 执行一系列测试，让 BIOS 在一次单独的调用中读取紧接着的更多扇区。读取的数据与参考数据相比较，来保证 BIOS 和控制器正确地读取数据。每次测试的结果都可显示出来。

使用 DISKTYPE 时，为要测试的驱动器加入一个命令行值：软盘选 0 或 1，硬盘选 80 或 81。例如，要测试 C：硬盘驱动器，使用如下命令行：

```
DISKTEST80
```

在某个系统上 DISKTEST 的测试结果如图 11-6 所示。在这种情况下，至少可以可靠地读取 81h 个扇区。读失败时 BIOS 会准确地返回一个错误。

```
DISKTEST——中断 13h 读扇区测试                    V2.00©1994,1996FVG
装入 128KB，一次一个扇区，参考数据测试
.....

正在读取的扇区数目：01h  通过了比较，返回 AH=0，AL=00h
正在读取的扇区数目：33h  通过了比较，返回 AH=0，AL=00h
正在读取的扇区数目：66h  通过了比较，返回 AH=0，AL=00h
正在读取的扇区数目：7Fh  通过了比较，返回 AH=0，AL=00h
正在读取的扇区数目：80h  通过了比较，返回 AH=0，AL=00h
正在读取的扇区数目：81h  通过了比较，返回 AH=0，AL=00h
正在读取的扇区数目：C0h  失败，错误号=09h
```

图 11-6　DISKTEST 的运行结果

我惊奇地发现，许多 BIOS 不会正确地处理标准的数据传送。DISKTEST 在一系列系统上的测试结果可以分为如下五类：

1. 工作正常，支持传送多达 FFh 个扇区
2. 支持传送 80h 个扇区，但是如果试图传送 81h 个或者更多的扇区时，系统就会挂起
3. 工作正常，支持传送多达 80h 个扇区，但是在试图传送 81h 个以上的扇区时，即使数据是不正确的，BIOS 仍会指示传送成功
4. 在试图读取 80h 个以上的扇区时，会触发一个错误码来表示读取数据失败
5. 在读取多于一个磁道的数据时，BIOS 会指示读取成功，但实际上数据是错误的

不论返回了任何一个错误条件，测试都会多次重试。这一点对膝上型电脑的硬盘驱动器和软盘驱动器特别重要，在这些驱动器上马达达到正常速度所需要的时间很短。

所有的这些意味着什么呢？对于硬盘驱动器而言，传送一个磁道的最大扇区数总是可靠的。如果必须一次读取一个磁道以上的扇区，在这样做之前记住要测试一下系统。在读取一个完整的磁道时，首先要求读取磁头上的第一个扇区。在某些系统上越过磁道的边界来读取扇区是不可靠的。

例如，如果驱动器每磁道有 17 个扇区，并且你需要从扇区 15 处开始读取 8 个扇区，那么你就需要操作两次。从第一个磁道读取前三个扇区（磁头 0，扇区 15、16 和 17）。接着，从磁头 1 的扇区 0 处开始读取 5 个扇区。这样就获得了逻辑扇区 18~22。

端口归纳

下面的端口列表用来控制硬盘控制器，不同的平台可能会支持不同的寄存器。早期的 AT 系统只支持一个控制器，这个控制器可以支持两个驱动器。大多数 1995 年及以后的系统可以支持两个控制器，四个驱动器。某些系统和特殊的控制器可以支持四个控制器，多达八个驱动器。下面的端口分配情况比较典型，当然某些供应商也可能会使用其他替代端口。

端口	类型	功能	平台
70h	I/O	硬盘数据（第四个控制器）	AT/EISA
71h	输入	硬盘错误寄存器（第四个控制器）	AT/EISA
71h	输出	硬盘写预补偿寄存器（第四个控制器）	AT/EISA
72h	I/O	硬盘扇区计数（第四个控制器）	AT/EISA
73h	I/O	硬盘扇区号（第四个控制器）	AT/EISA
74h	I/O	硬盘磁柱号低（第四个控制器）	AT/EISA
75h	I/O	硬盘磁柱号高（第四个控制器）	AT/EISA
76h	I/O	硬盘驱动器和磁头（第四个控制器）	AT/EISA
77h	输入	硬盘状态（第四个控制器）	AT/EISA

77h	输出	硬盘命令（第四个控制器）	AT/EISA
F0h	I/O	硬盘数据（第三个控制器）	AT/EISA
F1h	输入	硬盘错误寄存器（第三个控制器）	AT/EISA
F1h	输出	硬盘写预补偿寄存器（第三个控制器）	AT/EISA
F2h	I/O	硬盘扇区计数（第三个控制器）	AT/EISA
F3h	I/O	硬盘扇区号（第三个控制器）	AT/EISA
F4h	I/O	硬盘磁柱号低（第三个控制器）	AT/EISA
F5h	I/O	硬盘磁柱号高（第三个控制器）	AT/EISA
F6h	I/O	硬盘驱动器和磁头（第三个控制器）	AT/EISA
F7h	输入	硬盘状态（第三个控制器）	AT/EISA
F7h	输出	硬盘命令（第三个控制器）	AT/EISA
170h	I/O	硬盘数据（第二个控制器）	AT/EISA
171h	输入	硬盘错误寄存器（第二个控制器）	AT/EISA
171h	输出	硬盘写预补偿寄存器（第二个控制器）	AT/EISA
172h	I/O	硬盘扇区计数（第二个控制器）	AT/EISA
173h	I/O	硬盘扇区号（第二个控制器）	AT/EISA
174h	I/O	硬盘磁柱号低（第二个控制器）	AT/EISA
175h	I/O	硬盘磁柱号高（第二个控制器）	AT/EISA
176h	I/O	硬盘驱动器和磁头（第二个控制器）	AT/EISA
177h	输入	硬盘状态（第二个控制器）	AT/EISA
177h	输出	硬盘命令（第二个控制器）	AT/EISA
1F0h	I/O	硬盘数据	AT/EISA
1F1h	输入	硬盘错误寄存器	AT/EISA
1F1h	输出	硬盘写预补偿寄存器	AT/EISA
1F2h	I/O	硬盘扇区计数	AT/EISA
1F3h	I/O	硬盘扇区号	AT/EISA
1F4h	I/O	硬盘磁柱号低	AT/EISA
1F5h	I/O	硬盘磁柱号高	AT/EISA
1F6h	I/O	硬盘驱动器和磁头	AT/EISA
1F7h	输入	硬盘状态	AT/EISA
1F7h	输出	硬盘命令	AT/EISA
276h	输入	硬盘替代状态（第四个控制器）	AT/EISA
276h	输出	硬盘适配器寄存器（第四个控制器）	AT/EISA
277h	输入	硬盘驱动器地址寄存器（第四个控制器）	AT/EISA
2F6h	输入	硬盘替代状态（第三个控制器）	AT/EISA
2F6h	输出	硬盘适配器寄存器（第三个控制器）	AT/EISA
2F7h	输入	硬盘替代状态（第三个控制器）	AT/EISA
320h	输入	硬盘驱动器控制器状态	PC/XT
320h	输出	硬盘命令	PC/XT
321h	输入	硬盘控制器状态	PC/XT

321h	输出	硬盘控制器重启	PC/XT
322h	输入	硬盘获取类型开关	PC/XT
322h	输出	硬盘控制器选择脉冲	PC/XT
323h	输出	硬盘 DMA 和中断屏蔽	PC/XT
376h	输入	硬盘替代状态（第二个控制器）	AT/EISA
376h	输出	硬盘适配器寄存器（第二个控制器）	AT/EISA
377h	输入	硬盘驱动器地址寄存器（第二个控制器）	AT/EISA
3F6h	输入	硬盘替代状态	AT/EISA
3F6h	输出	硬盘适配器寄存器	AT/EISA
3F7h	输入	硬盘驱动器地址寄存器	AT/EISA
3510h	输入	ESDI 硬盘状态（16 位）	PS/2
3510h	输出	ESDI 硬盘状态（16 位）	PS/2
3512h	输入	ESDI 硬盘基本状态（16 位）	PS/2
3512h	输出	ESDI 硬盘基本控制（16 位）	PS/2
3513h	输入	ESDI 硬盘中断状态	PS/2
3513h	输出	ESDI 硬盘注意	PS/2
3518h	输入	ESDI 硬盘状态（16 位），次级	PS/2
3518h	输出	ESDI 硬盘命令（16 位），次缘	PS/2
351Ah	输入	ESDI 硬盘基本状态，次级	PS/2
351Ah	输出	ESDI 硬盘基本控制，次级	PS/2
351Bh	输入	ESDI 硬盘中断状态，次级	PS/2
351Bh	输出	ESDI 硬盘注意，次级	PS/2

端口细节

端口	类型	描述	平台
70h	I/O	硬盘数据（第四个控制器）	AT/EISA

第四个硬盘适配器的替代地址。参看端口 1F0h，I/O。

端口	类型	描述	平台
71h	输入	硬盘错误寄存器（第四个控制器）	AT/EISA

第四个硬盘适配器的替代地址。参看端口 1F1h，输入。

端口	类型	描述	平台
71h	输出	硬盘写预补偿（第四个控制器）	AT/EISA

第四个硬盘适配器的替代地址。参看端口 1F1h，输出。

端口	类型	描述	平台
72h	I/O	硬盘扇区计数（第四个控制器）	AT/EISA

第四个硬盘适配器的替代地址。参看端口 1F2h，I/O。

端口	类型	描述	平台
73h	I/O	硬盘扇区号（第四个控制器）	AT/EISA

第四个硬盘适配器的替代地址。参看端口 1F3h，I/O。

端口	类型	描述	平台
74h	I/O	硬盘磁柱号低（第四个控制器）	AT/EISA

第四个硬盘适配器的替代地址。参看端口 1F4h，I/O。

端口	类型	描述	平台
75h	I/O	硬盘磁柱号高（第四个控制器）	AT/EISA

第四个硬盘适配器的替代地址。参看端口 1F5h，I/O。

端口	类型	描述	平台
76h	I/O	硬盘驱动器和磁头（第四个控制器）	AT/EISA

第四个硬盘适配器的替代地址。参看端口 1F6h，I/O。

端口	类型	描述	平台
77h	输入	硬盘状态（第四个控制器）	AT/EISA

第四个硬盘适配器的替代地址。参看端口 1F7h，输入。

端口	类型	描述	平台
77h	输出	硬盘命令（第四个控制器）	AT/EISA

第四个硬盘适配器的替代地址。参看端口 1F7h，输出。

端口	类型	描述	平台
F0h	I/O	硬盘数据（第三个控制器）	AT/EISA

第三个硬盘适配器的替代地址。参看端口 1F0h，I/O。

端口	类型	描述	平台
F1h	输入	硬盘错误寄存器（第三个控制器）	AT/EISA

第三个硬盘适配器的替代地址。参看端口 1F1h，输入。

端口	类型	描述	平台
F1h	输出	硬盘写预补偿（第三个控制器）	AT/EISA

第三个硬盘适配器的替代地址。参看端口 1F1h，输出。

端口	类型	描述	平台
F2h	I/O	硬盘扇区计数（第三个控制器）	AT/EISA

第三个硬盘适配器的替代地址。参看端口 1F2h，I/O。

端口	类型	描述	平台
F3h	I/O	硬盘扇区号（第三个控制器）	AT/EISA

第三个硬盘适配器的替代地址。参看端口 1F3h，I/O。

端口	类型	描述	平台
F4h	I/O	硬盘磁柱号低（第三个控制器）	AT/EISA

第三个硬盘适配器的替代地址。参看端口 1F4h，I/O。

端口	类型	描述	平台
F5h	I/O	硬盘磁柱号高（第三个控制器）	AT/EISA

第三个硬盘适配器的替代地址。参看端口 1F5h，I/O。

端口	类型	描述	平台
F6h	I/O	硬盘驱动器和磁头（第三个控制器）	AT/EISA

第三个硬盘适配器的替代地址。参看端口 1F6h，I/O。

端口	类型	描述	平台
F7h	输入	硬盘状态（第三个控制器）	AT/EISA

第三个硬盘适配器的替代地址。参看端口 1F7h，输入。

端口	类型	描述	平台
F7h	输出	硬盘命令（第三个控制器）	AT/EISA

第三个硬盘适配器的替代地址。参看端口 1F7h，输出。

端口	类型	描述	平台
170h	I/O	硬盘数据（第二个控制器）	AT/EISA

第二个硬盘适配器的替代地址。参看端口 1F0h，I/O。

端口	类型	描述	平台
171h	输入	硬盘错误寄存器（第二个控制器）	AT/EISA

第二个硬盘适配器的替代地址。参看端口 1F1h，输入。

端口	类型	描述	平台
171h	输出	硬盘写预补偿（第二个控制器）	AT/EISA

第二个硬盘适配器的替代地址。参看端口 1F1h，输出。

端口	类型	描述	平台
172h	I/O	硬盘扇区计数（第二个控制器）	AT/EISA

第二个硬盘适配器的替代地址。参看端口 1F2h，I/O。

端口	类型	描述	平台
173h	I/O	硬盘扇区号（第二个控制器）	AT/EISA

第二个硬盘适配器的替代地址。参看端口 1F3h，I/O。

端口	类型	描述	平台
174h	I/O	硬盘磁柱号低（第二个控制器）	AT/EISA

第二个硬盘适配器的替代地址。参看端口 1F4h，I/O。

端口	类型	描述	平台
175h	I/O	硬盘磁柱号高（第二个控制器）	AT/EISA

第二个硬盘适配器的替代地址。参看端口 1F5h，I/O。

端口	类型	描述	平台
176h	I/O	硬盘驱动器和磁头（第二个控制器）	AT/EISA

第二个硬盘适配器的替代地址。参看端口 1F6h，I/O。

端口	类型	描述	平台
177h	输入	硬盘状态（第二个控制器）	AT/EISA

第二个硬盘适配器的替代地址。参看端口 1F7h，输入。

端口	类型	描述	平台
177h	输出	硬盘命令（第二个控制器）	AT/EISA

第二个硬盘适配器的替代地址。参看端口 1F7h，输出。

端口	类型	描述	平台
1F0h	I/O	硬盘数据（16/32 位）	AT/EISA

该 I/O 端向或从硬盘控制器传送数据。通常一次传送 256 字或者一个扇区大小的数据。与使用 DMA 传送数据的 PC/XT 和 PS/2 ESDI 控制器不同，AT 控制器的 BIOS 直接通过该端口传送数据。

在传送第一个扇区的数据之前，BIOS 必须等待控制器发出信号指示数据可用。当控制器准备好可以传送数据时，端口 1F7h 的硬盘状态，数据请求位 3 会被置高。

数据按 256 字或 128 双字的块传送，通常和一个输入或输出串指令 INSW/INSD 或 OUTSW/OUTSD 一起使用。只有当系统总线和控制器硬件都支持时，才可以使用 32 位传送。例如，大多数基于 VLB 和 PCI 的控制器支持 32 位数据传送。传送每个扇区期间必须禁止中断。参看“通用读扇区操作”部分，以获取使用该寄存器的一个例子。

端口	类型	描述	平台
1F1h	输入	硬盘错误寄存器	AT/EISA

该错误寄存器保存了上次执行的命令的错误状态，或者诊断模式的错误状态。在普通操作模式下，如果在一个命令期间出现了一个错误，就会在状态寄存器中设置错误位，即端口 1F7 的第 0 位。只有当设置了该位之后，错误寄存器才对普通操作有效。

输入（0~7）普通操作，如果设置了错误位

位		
	7r=1	检测到坏扇区——设置了扇区 ID 域的坏块位。只有格式化命令才会设置坏块位，来通知扇区有一个硬数据错误。不应使用该扇区，或者指定的磁盘地址不正确。
	6r=1	不可更正的数据错误——扇区数据的错误超出了控制器错误更正机制所能修复的能力范围。
	5r=1	改变了媒质——移走了可移走媒质，并插入了另一个媒质。在使用之前必须确定媒质类型和容量。大多数控制器不支持该错误条件，在不支持的控制器上会返回零。如果支持它，则只有确认媒质改变命令 DBh 才可以清除该位。参看端口 1F7h 获取细节。
	4r=1	找不到 ID 或目标扇区——控制器不能定位到指定的磁柱、扇区或磁头地址。该错误可能也表示读自扇区的 ID 域存在一个错误。控制器通常会试 8 次或更多次来定位和/或读扇区。
	3r=1	媒质改变请求——用户激活了可移走媒质驱动器的释放媒质按钮或锁住按钮。可能要求软件使用供应商特有的命令开锁驱动器和/或弹出媒质。大多数控制器不支持该错误文件，在不支持的控制器上会返回零。
	2r=1	命令中止——如果指定的命令无意义，或者在驱动器忙或写错误时发了一条命令，就会设置该标志。检查端口 1F7h 或 3F6h 的状态位来获取驱动器忙和写错误的条件。
	1r=1	找不到磁道 0——在重校命令期间，控制器向外移动磁头达控制器所支持的最大磁柱数，但是仍未激活零磁道线。

0r=1	找不到地址标签——在读扇区命令期间找不到合适的扇区 ID 之后，没有检测到地址标签。扇区可能有一个硬错误，应不再使用。

在任何一个诊断模式下，或者重启之后控制器的自检期间，错误寄存器的内容指示了诊断测试的结果。

输入（0~7）诊断错误状态（缺省情况下选择驱动器 0）

十六制值	描述
1	驱动器 0 和 1 没有检测到错误
2	设备错误
3	扇区缓冲区（在控制器内）错误
4	ECC 电路错误
5	控制处理器错误
81	驱动器 1 失败，驱动器 0 无错误
82	驱动器 1 失败，驱动器 0 设备错误
83	驱动器 1 失败，驱动器 0 扇区缓冲区错误
84	驱动器 1 失败，驱动器 0 ECC 电路错误
85	驱动器 1 失败，驱动器 0 处理器错误

为了在诊断测试完成后获得驱动器 1 的错误状态，可以读取本寄存器。如果存在驱动器 1，那么第 7 位置 1 表示驱动器 1 有问题。向驱动器和磁头寄存器端口 1F6h 写入 10h，来设置驱动器 1 的驱动位。再次读取该错误寄存器会返回驱动器 1 的状态。并非所有的控制器都能从驱动器 1 获取诊断失败信息。

输入（0~7）诊断错误状态（诊断后选择驱动器 1）

十六进制	描述
1	没有检测到错误
2	设置错误
3	扇区缓冲区（在控制器内部）错误
4	ECC 电路错误
5	控制处理器错误

端口	类型	描述	平台
1F1h	输出	硬盘写预补偿	AT/EISA

某些老式的磁盘驱动器在写操作期间，要求在驱动器的内部磁柱上进行写预补偿。该寄存器指定从哪个磁柱开始写预补偿。将写补偿开始的磁柱号除以 4，在寄存器中装入所得的结果。要从磁柱 512 处开始写预补偿，可使用值 512/4 或 128。值 FFh 表示驱动器不需

要写预补偿。

由于目前只有少数的驱动器需要写预补偿，新式的控制器（IDE）已经删除了该特性，而将该字节重新作为控制器独特的用户特性。为了保持向后兼容性，会忽略该值的保存，除非在写后触发一个专用属性命令（参看端口 1F7h，命令 E9h 和 Efh）。

端口	类型	描述	平台
1F2h	I/O	硬盘扇区计数	AT/EISA

这个寄存器为读和写操作保存了要传输扇区的数目。值 0 表示要传输 256 个扇区。尽管在一次操作中控制器能传送多达 128K 的数据，但是在一次操作中处理 64K 以上数据时许多 BIOS 会有 bug。

命令完成时，这个寄存器保存着剩下的扇区数。值 0 表示已传送了所有的扇区，非零值表示需要继续传送的扇区数，以完成最初的请求。

其他命令，例如格式化和初始化驱动器，将这个寄存器定义为其他用途。参看特殊命令以获取有关细节。

端口	类型	描述	平台
1F3h	I/O	硬盘扇区号	AT/EISA

此寄存器记录了任何磁盘访问命令的开始扇区号。当命令控制器操作完成后，寄存器保持驱动器的当前扇区号。

在 IDE 驱动器中，当 LBA 模式被激活时（参见端口 1F6h），该端口保存 28 位 LBA 的第 0~7 位。

端口	类型	描述	平台
1F4h	I/O	硬盘磁柱低	AT/EISA

在任何一次磁盘访问，例如读、写、检查或查找时，这个寄存器保存了起点磁柱号的低 8 字节。完成命令控制器操作时，该寄存器保存了驱动器当前磁柱号的低 8 字节。

在 IDE 驱动器上，如果激活了 LBA 模式（参看端口 1F6h），那么该端口保存了 28 位 LBA 的第 8~15 位。

端口	类型	描述	平台
1F5h	I/O	硬盘磁柱高	AT/EISA

在任何一次磁盘访问，例如读、写、检查或查找时，这个寄存器保存了起点磁柱号的高字节。完成命令控制器操作时，该寄存器保存了驱动器当前磁柱号的高位。

老式的控制器只支持磁柱号的 10 位组合，并忽略这个寄存器的第 2~7 位。如果试图写磁柱 1,024，老式的控制器会向磁柱 0 写入！

大多数当前的 EIED 控制器在该寄存器中使用附加位才支持 10 位以上的磁柱号。

在 IDE 驱动器中，当 LBA 模式被激活时（参见端口 1F6h），此端口保存 28 位 LBA 的第 16~23 位。

端口	类型	描述	平台
1F6h	I/O	硬盘驱动器和磁头寄存器	AT/EISA

该寄存器指定了驱动器号和磁头号。当命令控制器完成其操作时，磁头号更新为当前的磁头号。IDE 驱动器固定为 512 字节的扇区大小，并再次使用第 6 位作为 LBA（逻辑块寻址）的指示器。如果激活了 LBA，低四位就会与扇区号和磁柱端口一起定义一个 28 位的 LBA 地址。

I/O（位 0~7）非 IDE 驱动器

位				
	7r/w=1	ECC 扩展数据区（用于所有情况下）。这使得磁盘上的扇区大小为 512 字节+7 个 ECC 字节。所有的系统都使用这种格式		
	6r/w=0 5r/w=1	扇区大小		
		第 6 位	第 5 位	每扇区的字节数
		0	0	=256
		0	1	=512（用于所有情况下）
		1	0	=1024
		1	1	=128
	4r/w=0	选择驱动器 0（主盘）		
	1	选择驱动器 1（从盘）		
	3r/w=x	磁头号（0~15）		
	2r/w=x			
	1r/w=x			
	0r/w=x			

I/O（0~7） IDE 驱动器——普通的磁柱-磁头-扇区模式

位		
	7r/w=1	必须设置为 1
	6r/w=0	驱动器寻址使用磁柱-磁头-扇区模式
	5r/w=1	必须设置为 1
	4r/w=0	选择驱动器 0（主盘）
	1	选择驱动器 1（从盘）
	3r/w=x 2r/w=x 1r/w=x 0r/w=x	磁头号（0~15）

I/O（0~7） IDE 驱动器——LBA（逻辑块寻址）模式

位	7r/w=1	必须设置为 1
	6r/w=1	LBA 模式标志
	5r/w=1	必须设置为 1
	4r/w=0	选择驱动器 0（主盘）
	1	选择驱动器 1（从盘）
	3r/w=x	LBA 地址的第 27 位
	2r/w=x	LBA 地址的第 26 位
	1r/w=x	LBA 地址的第 25 位
	0r/w=x	LBA 地址的第 24 位

端口	类型	描述	平台
1F7h	输入	硬盘状态	AT/EISA

该端口返回控制器的状态。读该端口会清除任何挂起的中断，并确认隐含的中断。使用端口 3F6h 输入，可以获取这个状态信息，但是不清除挂起的中断，也不确认中断。

如果设置了控制器位 7 忙，那么所有其他位都将无意义。另外，在忙时读取任何一个控制器端口 1F0~1F7，都会返回无意义的结果。在忙位从忙变成不忙之后 400ns，端口 1F0~1F7 中的值以及该寄存器中的其他位才有效。

如果中断是由异常中止命令（通常由于超时错误）引起的，那么第 6、5、4 位就会锁存起来而不会改变，直到读该寄存器。

输入（位 0~7）

位	7r=0	控制器不忙
	1	控制器忙，位 0~6 无意义（参看上文）
	6r=0	选中的驱动器未准备好
	1	选中的驱动器准备好，可能响应来自控制器的命令。如果第 4 位置 1，那么可以读写或查找
	5r=1	选中的驱动器发生写错误
	4r=1	选中的驱动器完成了查找工作，现在磁头位于所要求的位置
	3r=1	激活数据请求——控制器指示它已准备好。从扇区缓冲区发送和接收字。参看数据端口 1F0h 以了解细节
	2r=1	更正数据——读自磁盘的数据带有一个可更正的数据错误，该错误已被供应商的 ECC 算法改正了。该条件不阻止多扇区数据传送
	1r=1	索引脉冲——在老式的驱动器上，此位反映了索引脉冲的状态，并且磁盘每旋转一周就设置一次该位。目前，供应商可能将该位用做其他用途或者不使用该位
	0r=0	无错误

1	前面的命令中出现了错误——错误寄存器指示了错误的属性。向控制器送下一条命令时会清除该位

端口	类型	描述	平台
1F7h	输出	硬盘命令	AT/EISA

通过这个端口向控制器发命令。在发命令之前，控制器应准备好（不忙）。从状态寄存器端口 1F7 的第 7 位读取忙状态。装入了一条新命令之后会自动清除中断请求。传送命令之前，必须装入命令所要求的所有寄存器。

除非特别指出，否则每个命令必须带有磁盘地址，磁盘地址由驱动器号、磁柱、磁头和扇区组成。在使用老式的非 IDE 型驱动器时，写和格式化命令还必须设置写预补偿磁柱的值。

参看端口 1F0h，了解从 / 向控制器传送数据字的普通传送方法。某些自带硬盘 BIOS 的控制器可能会使用 DMA，而不像普通的非 DMA I/O 传送那样由系统 BIOS 完成。

一个命令标有“（有限）”表示，控制器可能不支持它，并且一个早于 1995 年的系统 BIOS 永远不会触发它。这些加入的命令供专用的供应商硬件使用，例如，可移走媒质的驱动器和专用的断电操作。请参考最近的 ATA 规范（ATA-1、ATA-2 和 ATA-3 规范）。

十六进制	命令
00	无操作
1x	重校
20	读多块扇区，带重试
21	读多块扇区，无重试
22	读扇区和 ECC 数据，带重试
23	读扇区和 ECC 数据，无重试
30	写多块扇区，带重试
31	写多块扇区，无重试
32	写扇区和 ECC 数据，带重试
33	写扇区和 ECC 数据，无重试
3C	写多个扇区并检查（有限）
40	检查多个扇区，带重试
41	检查多个扇区，无重试
50	格式化磁道
7x	查找
90	驱动器诊断
91	初始化驱动器参数
92	下载微码
94	备用立即（有限）
95	空闲立即（有限）
96	备用（有限）
97	空闲（有限）

98	检查电源模式（有限）
99	设置睡眠模式（有限）
B0	SMART 操作（有限）
C4	读多块（有限）
C5	写多块（有限）
C6	设置多块模式（有限）
C8	读 DMA，带重试（有限）
C9	读 DMA，无重试（有限）
CA	写 DMA，带重试（有限）
CB	写 DMA，无重试（有限）
DB	确认已改变了可移走媒质（有限）
DC	引导后的可移走媒质（有限）
DD	引导前的可移走媒质（有限）
DE	可移走媒质的门锁住（有限）
DF	开锁门时的可移走媒质的门开锁（有限）
E0	备用立即（有限）
E1	空闲立即（有限）
E2	备用（有限）
E3	空闲（有限）
E4	读缓冲区（有限）
E5	检查电源模式（有限）
E6	设置睡眠模式（有限）
E8	写缓冲区（有限）
E9	写相同（有限）
EC	获取设备信息（有限）
ED	媒质弹出（有限）
EE	获取设备信息 DMA（有限）
EF	设置特性（有限）
F1	安全设置密码（有限）
F2	安全性开锁（有限）
F3	安全擦除准备（有限）
F4	安全擦除单元（有限）
F5	安全冻结锁（有限）
F6	安全禁止密码（有限）

命令细节

命令	描述	端口
0	无操作	1F7h

在驱动器上执行无操作，它对异常中止命令错误寄存器，位 2=1 做出响应。

命令	描述	端口
1xh	重校	1F7h

只需要指定磁头的值，任何从 10h 到 17h 的命令都会将磁头定位到磁柱 0。该操作忽略所有其他的寄存器。

命令	描述	端口
20h	读多个扇区，带重试	1F7h

读指定的扇区数。如果检测到数据错误，则自动重试该操作来得到有效的数据。重试的次数与控制器供应商有关。

命令	描述	端口
21h	读多个扇区，不带重试	1F7h

同命令 20h 一样，但如果有数据错误不再重试。

命令	描述	端口
22h	读扇区和ECC 数据，带重试	1F7h

读一个扇区，包括 ECC 数据字节。读取的总字节数是 512，加上 4 或 7 字节的 ECC。命令 20h 执行普通的读操作，这时控制器不向用户传送 ECC 字节。

如果检测到数据错误，则自动重试该操作以得到有效的数据。重试的次数与控制器供应商有关。

命令	描述	端口
23h	读扇区和 ECC 数据，无重试	1F7h

与命令 21h 相同，但是读取的扇区包括 4 或 7 个 ECC 字节。扇区长 512 个字节加上 ECC 字节。

命令	描述	端口
30h	写多块扇区，带重试	1F7h

写入指定扇区的数目，如果在定位扇区 ID 时检测到了数据 ID，则自动重试该操作以获得有效的 ID 数据。重试的次数与控制器供应商有关。

命令	描述	端口
31h	写多块扇区，无重试	1F7h

与命令 30h 相同，只是在扇区 ID 数据出错时不重试。

命令	描述	端口
32h	写扇区和ECC数据，带重试	1F7h

从用户那里写一个扇区，包括 ECC 数据字节。控制器并不像命令 30h 那样生成 ECC 字节。这意味着用户提供的扇区大小为 512 字节加上 4 个或 7 个 ECC 字节。

如果在定位扇区 ID 时检测到了一个数据错误，就会自动重试该操作以获得有效的 ID 数据。重试的次数与控制器供应商有关。

命令	描述	端口
33h	写扇区和ECC数据，带重试	1F7h

该命令和命令 32h 相同，只是在扇区 ID 数据出错时不重试。

命令	描述	端口
3Ch	写多扇区并检查（有限）	1F7h

该命令写入指定的数据，然后在写操作完成之后检查数据。写操作部分类似于命令 30h，写多块扇区，大多数 AT 控制器不支持该命令。

命令	描述	端口
40h	检查多扇区，带重试	1F7h

读并检查指定的扇区。如果检测到一个数据错误，则自动重试该操作以获得有效的数据。重试的次数与控制器供应商有关。该命令不传递数据，在执行该命令期间不使用 DMA。

如果找到了指定的扇区，并且检查扇区的 ECC 字节时没有发现任何错误，则认为该检查是成功的。如果发现了 ECC 错误，则重读该扇区几次以获取有效的数据。无法获得有效的 ECC 会使检查失败。

命令	描述	端口
41h	检查多扇区	1F7h

与命令 40h 相同，只是在数据出错时不重试。

命令	描述	端口
50h	格式化磁道	1F7h

格式化指定磁道的扇区，并写入扇区 ID 信息和初始化数据。扇区计数寄存器，端口 1F2h 中装入每磁道的扇区数。该格式化操作不使用扇区号寄存器，端口 1F3h，并忽略它。

必须为格式化命令提供扇区数据表，并为磁道上的每个扇区包括两个字节的信息。该命令使用与写扇区相同的进程将扇区数据表传送到控制器。中断 13h 功能 5 中描述的表 11-6 列出了扇区数据表的层次结构，并解释了扇区数据表是如何设置交错的。

命令	描述	端口
7xh	查找	1F7h

任何从 70h 到 7Fh 的命令都会将磁头定位到指定的磁柱上。查找命令不使用扇区计数寄存器，端口 1F2h，并忽略它。

所有的读、写、检查以及格式化命令都会自动执行隐含的查找命令，所以没有必要在这些命令之前使用这种命令。

要想查找命令正常地发挥功能，不必格式化驱动器。

命令	描述	端口
90h	驱动器诊断	1F7h

该命令为所有控制器附带的驱动器激活驱动器诊断。必须选择驱动器 0 来运行诊断程序，但如果存在两个驱动器，则必须同时运行在两个驱动器上。忽略所有其他的寄存器，诊断的结果返回到错误寄存器端口 1F1h 中，参看端口 1F1h，输入，以获取其他信息。

命令	描述	端口
91h	初始化驱动器参数	1F7h

为指定的驱动器向控制器中装入驱动器扇区和磁头上限。在装入该命令之前，扇区计数寄存器，端口 1F2h，装入每磁道的扇区。驱动器和磁头寄存器，端口 61h，装入最大有效的磁头号，它比磁头的总数少 1。该命令忽略所有其他的命令。

命令	描述	端口
92h	下载微码（有限）	1F7h

改变控制器的微代码。该命令是特定的供应商所特有的。磁头寄存器的磁头位都置为零，两个磁柱寄存器也都置为 0。要装入的 512 字节的进位数以 16 位形式指定，在这里，高位部分保存在扇区号寄存器 0 中，而低位部分保存在扇区计数寄存器中。

特性寄存器设置为“1”，以供临时下载之用，而值“7”表示现在和将来都要使用下载。在完成之时两种下载类型立即有效。

命令	描述	端口
94h	备用立即（有限）	1F7h

驱动器设置为备用模式。这是一个 2 字节命令，要求在 94h 之后立即发送命令 E0h。如果已经关闭了驱动器马达，则不会有任何动作，否则就关闭驱动器马达。该命令忽略所有其他的命令。

AT 控制器不支持该命令，它特意为膝上型电脑和其他低功率类型电脑而设计。

命令	描述	端口
95h	空闲立即（有限）	1F7h

如果驱动器处于备用模式，关闭了马达，则重新打开马达。这是一个 2 字节命令，要求在 95h 之后发送命令 E1h。该命令忽略所有其他的寄存器。

大多数 AT 控制器不支持该命令，它特意为膝上型和其他低功率类型电脑而设计。

命令	描述	端口
96h	备用（有限）	1F7h

驱动器设置为备用模式。这是一个 2 字节命令，要求在 96h 之后立即发送命令 E2h。如果已经关闭了驱动器马达，则不会有任何动作。如果扇区计数寄存器非零，则驱动器开始进入空闲状态并同时开放减数计数器。在减数递减的最后，关闭驱动器马达。该命令忽略所有其他的寄存器。

大多数 AT 控制器不支持该命令，它特意为膝上型和其他低功率类型电脑而设计。

命令	描述	端口
97h	空闲（有限）	1F7h

如果驱动器处于备用模式，关闭了马达，则重新打开马达。这是一个 2 字节命令，要求在 97h 之后发送命令 E3h。该命令忽略所有其他的寄存器。

大多数 AT 控制器不支持该命令，它特意为膝上型和其他低功率类型电脑而设计。

命令	描述	端口
98h	检查电源模式（有限）	1F7h

找出指定驱动器的状态，空闲或备用。这是一个 2 字节命令，要求在 98h 之后立即发送命令 E5h。该命令忽略所有寄存器，但驱动器选择位除外。

在扇区计数寄存器中返回该状态。值零表示驱动器位于备用模式，关闭了马达，或者正在从/向备用模式过渡。值 FFh 表示驱动器位于空闲模式，马达正在运行。大多数 AT 控制器不支持该命令，它特意为膝上型电脑而设计。

命令	描述	端口
99h	设置睡眠模式（有限）	1F7h

关掉控制器及所有驱动器电源。这是一个 2 字节命令，要求在 99h 之后立即发送命令 E6h。忽略所有其他的寄存器。

不接受进一步的命令，直到软件或硬件重启为止。该命令忽略所有其他的寄存器。大多数 AT 控制器不支持该命令，它特意为膝上型电脑而设计。

命令	描述	端口
B0h	SMART 操作（有限）	1F7h

控制各种属性的 SMART（自监控、分析、报告技术），表 11-16 列出了可用的功能。在激活之前，在特性寄存器中放入十六进制功能值，在磁柱低寄存器中保存值 4Fh，在磁柱高寄存器中保存值 C2h。

表 11-16 SMART 子功能

功 能（十六进制）	描 述
D0	读属性值*
D1	读属性阈*
D2	禁止属性自动保存（扇区计数也设为 0）*
D3	保存属性值*
D2	开放属性自动保存（扇区计数也设为 F1h）*
D8	开放操作
D9	禁止操作
DA	返回状态

* SMART 系统可能不支持该功能

命令	描述	端口
C4h	读多块（有限）	1F7h

该命令类似于读多扇区命令，只是在传送每个扇区的结尾不生成中断。它特意为那些使用 DMA 进行硬盘传送的系统而设计。在该命令之前必须触发设置多块模式命令 C6h。参看设置多块模式命令 C6h 以获取其他信息。大多数 AT 控制器不支持该命令。

命令	描述	端口
C5h	写多块（有限）	1F7h

该命令类似于写多扇区命令，只是在传送每个扇区的结尾不生成中断。它特意为那些使用 DMA 进行硬盘传送的系统而设计。在该命令之前必须触发设置多块模式命令 C6h。参看设置多块模式命令 C6h 以获取其他信息。大多数 AT 控制器不支持该命令。

命令	描述	端口
C6h	设置多块模式（有限）	1F7h

如果支持，则控制器提供一种特性，一次发送多组扇区。在执行该命令之前，在扇区计数寄存器中装入每块的扇区数。有效的块大小是 2、4、8 或 16。如果控制器的内部缓冲区大于 8K，则可能支持更大的块。用到了驱动器和磁头寄存器中的驱动器位。忽略所有其他的寄存器。

在该命令之后总会是读或写多块的命令 C4h 或 C5h。它开始传送指定块的扇区，并且只在完成每个块的传送之后控制器才会中断系统。在执行读或写多块命令时，由扇区计数寄存器指定传输的扇区总数。

例如，如果要传送五个扇区，每个块的大小为 2，则先传送前两个扇区，紧接着随后两个，最后一块传送剩下的一个扇区。本例只生成三次中断，而不是普通情况下为每个扇区生成一个中断。

多块特性为使用 DMA 时提高运行速度而设计。大多数 AT 控制器不支持该命令。

命令	描述	端口
C8h	读 DMA，带重试，有限）	1F7h

该命令类似于读多块扇区，只是在读操作之前硬盘 BIOS 会设置 DMA。只有传送完所有的扇区之后才会生成一个中断。大多数 AT 控制器不支持该命令。

命令	描述	端口
C9h	读 DMA，无重试，有限）	1F7h

与命令 C8h 相同，但在数据出错时不重试，大多数 AT 控制器不支持该命令。

命令	描述	端口
CAh	写 DMA，带重试，有限）	1F7h

该命令类似于写多块扇区（命令 30h），只是在读操作之前硬盘 BIOS 会设置 DMA。只在传送完所有的扇区之后才会生成一个中断，大多数 AT 控制器不支持该命令。

命令	描述	端口
CBh	写 DMA，无重试，有限）	1F7h

与命令 CAh 相同，只是在扇区 ID 出错时不重试。大多数 AT 控制器不支持该命令。

命令	描述	端口
DCh	引导后的可移走媒质，有限）	1F7h

在系统引导之后触发该命令来将供应商的专用数据发送到可移走媒质驱动器中。大多数 AT 控制器不支持该命令。

命令	描述	端口
DBh	确认改变了可移走媒质（有限）	1F7h

如果一个可移走媒质驱动器出现了媒质改变错误，则该命令确认这个改变，并清除错误寄存器端口 1F1h 的位 5，媒质改变错误位。只用到驱动器和磁头寄存器的驱动器位。忽略所有其他的寄存器。大多数 AT 控制器不支持该命令。

命令	描述	端口
DDh	引导前的可移走媒质（有限）	1F7h

在系统引导之前触发该命令来将供应商的专用数据发送到可移走媒质驱动器中，大多数 AT 控制器不支持该命令。

命令	描述	端口
DEh	锁住可移走媒质的门（有限）	1F7h

如果目前驱动器已准备好，并且没有锁住门，则命令选中的驱动器锁住门。只用到驱动器和磁头寄存器的驱动器位。忽略所有其他的寄存器。

如果已经锁住了门，或者驱动器没有准备好，则控制器在状态寄存器（端口 1F7h）中设置驱动器为尚未准备好状态。大多数 AT 控制器不支持该命令。

命令	描述	端口
DFh	开锁可移走媒质的门（有限）	1F7h

如果目前驱动器已准备好，并且已锁住门，则命令选中的驱动器开锁。只用到驱动器和磁头寄存器的驱动器位。忽略所有其他的寄存器。

如果门已经开锁，或者驱动器没有准备好，则控制器在状态寄存器（端口 1F7h）中设置驱动器为尚未准备好状态。大多数 AT 控制器不支持该命令。

命令	描述	端口
E0h	备用立即（有限）	1F7h

参看等同的命令 94h。

命令	描述	端口
E1h	空闲立即（有限）	1F7h

参看等同的命令 95h。

命令	描述	端口
E2h	备用（有限）	1F7h

参看等同的命令 96h。

命令	描述	端口
E3h	空闲（有限）	1F7h

参看等同的命令 97h。

命令	描述	端口
E4h	读缓冲区（有限）	1F7h

读控制器的内部扇区缓冲区的内容。如果前一个命令是写缓冲区，则读缓冲区命令会读取 512 个字节，与前面写到缓冲区的字节相同。大多数 AT 控制器不支持该命令。

命令	描述	端口
E5h	检查电源模式（有限）	1F7h

参看等同的命令 98h，以获取相关细节。

命令	描述	端口
E6h	设置睡眠模式（有限）	1F7h

参看等同的命令 99h，以获取相关细节。

命令	描述	端口
E8h	写缓冲区（有限）	1F7h

写控制器的内部扇区缓冲区。写操作要求传送 512 字节。大多数 AT 控制器不支持该命令。

命令	描述	端口
E9h	写相同（有限）	1F7h

该命令向多个扇区中写入相同的数据。只有一个扇区被读入控制器，然后使用该数据来写所有指定的扇区。在使用该命令之前必须在特性寄存器（1F1h）中装入一个值，22h 或 DDh，否则命令会中止而不执行任何动作。使用命令 Efh 来装入特性寄存器。

特性寄存器中装入值 22h 时，该写操作类似于普通的写操作。目标磁盘地址作为起点，扇区计数寄存器指定了要写入的相同的 512 字节信息的数目。

如果特性寄存器的设定值为 DDh，则控制器会向磁盘上的每个扇区写入相同的 512 字节。只在操作完成时才触发唯一一个中断。大多数 AT 控制器不支持该命令。

命令	描述	端口
ECh	获取设置信息（有限）	1F7h

该命令获取保存在设备上的 256 字设备信息。只有对象 IDE 那样在磁盘上带有这些数据的驱动器，该命令才起作用。大多数 AT 控制器不支持该命令。

表 11-17 描述了通过端口 1F0h 返回的信息。目前的 ATA 说明书不支持星号标出的项，它们被认为已过时。

表 11-17 驱动器信息

偏移量（十六进制）	大 小	描 述					
0	字	配置位					
		位	15=0	ATA 设备			
			1	ATAPI 设备			
			14=1	要求格式化速度容忍空隙*			
			13=1	有磁道偏移量选项*			
			12=1	有数据选通偏移量*			
			11=1	磁盘速度容忍比大于 0.5%*			
			10=x	磁盘传送速率*			
			9=x	位 10	位 9	位 8	兆 bps*
			8=x	0	0	1	=5 或更少
				0	1	0	=10 或 10 和 5 之间
				1	0	0	=10 以上
			7=x	驱动器类型			
			6=x	位 7	位 6		
				0	1=硬扇区		
				1	0=软扇区		
			5=1	开放同步驱动器马达选项*			
			4=1	磁头之间的切换时间大于 15μs*			
			3=0	MFM 编码*			
			2=x	扇区类型*			
			1=x	位 2	位 1		
				0	1=硬扇区		
				1	0=软扇区		
			0=0	未使用			
2	字	逻辑磁柱的总数——如果偏移量 78h 处的双字超过了 16,515,072，则该字会设为 16,383					
4	字	未使用					
6	字	逻辑磁柱的总数——如果偏移量 78h 处的双字超过了 16,515,072，则该字会设为 16					
8	字	每磁道的总字节数，未格式化*					

续表

偏移量（十六进制）	大 小	描 述		
A	字	每扇区的总字节数，未格式化*		
C	字	每个磁道的逻辑扇区道数——如果偏移量78h处的双字超过了16,512,072，则该字会设为63		
E	3个字	供应商指定，非标准		
14	10个字	串行数，20个ASCII字符，排列好，如果第一个字是0，则表示供应商尚未分配串行数		
28	字	缓冲区类型*		
		0=未指定		
		1=单个扇区缓冲区，不能同时通过系统和磁盘传送数据		
		2=多扇区缓冲区，双端口，支持驱动器和系统同时访问		
		3=与2相同，带高速缓存读		
2A	字	驱动器上的缓冲区大小，单位是512字节/页。如果设定为零，则未指定缓冲区大小*		
2C	字	使用带ECC读/写命令时传递的ECC字节数，如果设定为零，则未指定缓冲区大小		
2E	4个字	驱动器的固件改写信息，8个SACII字符。如果第一个字设定为零，则未指定改写信息		
36	20个字	驱动器的固件改写信息。8个ASCII字符。如果第一个字设定为零，则未指定型式号		
5E	字节	使用读或写多块命令可以传送的扇区的最大数目		
5F	字节	供应商指定，非标准		
60	字	32位I/O，1=支持，0=只支持16位*		
62	字	能力		
		位	15=0	未使用
			14=0	未使用
			13=0	备用时钟值由供应商指定
			1	备用时钟值与ATA-3或后来的标准一致
			12=0	高级传送位的特性位
			11=0	可能支持IORDY
			1	支持IORDY
			10=1	可以使用设置特性命令禁止IORDY
			9到1=x	未使用，或过时

续表

偏移量（十六进制）	大　小	描　述		
		0=1		支持 DMA*
64	字	未指定和/或未分配		
66	字节	供应商指定，非标准		
67	字节	由软件控制的 I/O 数据传送周期计时模式		
		0=慢，1=中等，2=快		
68	字节	供应商指定，非标准		
69	字节	由 DMA 控制的数据传送周期计时模式		
6A	字	标志		
		位	15~2=x	未指定和/或未分配
		位	1=0	偏移量 80h 到 8Ch 处报告的域有效
			1	偏移量 80h 到 8Ch 处报告的域可能有效
		位	0=0	偏移量 6Ch 到 75h 处报告的域有效
			1	偏移量 6Ch 到 75h 处报告的域可能有效
6C	字	当前逻辑磁柱的数目		
6E	字	当前逻辑磁头的数目		
70	字	当前每磁道逻辑扇区的数目		
72	双字	当前扇区的总数（必须等于前三个字相乘的结果）		
76	字节	在一次中断期间 R/W 多块命令目前可以传递的扇区数目。如果 R/W 多块支持（偏移量 5Eh）是零，则它必须是零。只有设置了偏移量 77h 处多扇区字节的第 0 位后该值才有效		
77	字节	多扇区		
		位	7~1=x	未使用
			0=1	设置多扇区有效
78	双字	激活 LBA 模式时，用户可访问的扇区总数		
7C	字	未使用		
7E	字节	支持多字 DMA 传递模式		
		位	7=1	支持模式 7
			6=1	支持模式 6
			5=1	支持模式 5
			4=1	支持模式 4
			3=1	支持模式 3
			2=1	支持模式 2

续表

偏移量（十六进制）	大 小	描 述		
			1=1	支持模式 1
			0=1	支持模式 0
7F	字节	激活多字传送模式（只能设置一位）		
		位	7=1	激活模式 7
			6=1	激活模式 6
			5=1	激活模式 5
			4=1	激活模式 4
			3=1	激活模式 3
			2=1	激活模式 2
			1=1	激活模式 1
			0=1	激活模式 0
81	字节	为将来的 PIO 模式支持而保留		
82	字	每字的最小多字 DMA 传送周期时间，单位纳秒		
84	字	制造商推荐的每字多字 DMA 传送周期时间，单位纳秒		
86	字	无流量控制时的最小 PIO 传送周期，单位纳秒		
88	字	带 IORDY 流量控制的最小 PIO 传送周期，单位纳秒		
8A	22 双字	为将来命令重叠和排列而保留		
A0	字	说明书的主版本号（例如 ATA-1=1，ATA-2=2，等等），值 0 或 FFFFh 表示设备没有报告版本号		
A2	字	说明书的次版本号。值 0 或 FFFFh 表示设备没有报告次版本号		
A4	字	支持特性设置（只有在本偏移量和 A6 偏移量的值不是零和 FFFFh 时才有效）		
		位	15~4=0	为将来设置而保留
			3=1	支持管理电源特性设置
			2=1	支持可移走媒质特性设置
			1=1	支持安全特性设置
			0=1	支持 START 特性设置（自监视、分析、以及为设备降级和/或错误的检测和预测作出技术报告）
A6	字	支持特性设置（只有在偏移量和 A4 偏移量的值不是零和 FFFFh 时才有效）		
		位	15=0	
			14=1	
			13~0=x	保留位（未使用过，未知）

续表

偏移量（十六进制）	大　小	描　　述		
A8	90 字节	未使用		
100	字	安全状态		
		位	15~9=x	未使用或保留
			8=0	安全等级设置为高
			1	安全等级设置为最大
			7=x	未使用或保留
			6=x	未使用或保留
			5=x	未使用或保留
			4=1	安全计数到期
			3=1	冻结安全
			2=1	锁住安全
			1=1	开放安全
			0=0	支持安全

* 过时的项，最近的 ATA 规范不支持

命令	描述	端口
EDh	媒质弹出（有限）	1F7h

在完成了任何挂起操作之后，开锁驱动器的门（如果可行），并弹出媒质。

命令	描述	端口
EEh	获取设备信息 DMA（有限）	1F7h

从 DMA 模式的设备中获取 256 字信息。参看命令 ECh 获取返回信息。

命令	描述	端口
EFh	设置特性（有限）	1F7h

该命令控制各种控制器的内部特性。其他的寄存器可能包括了某个特性所必须的值，这一点与供应商有关。并非所有的特性都为所有的供应商所支持。开放或禁止的特性会设置控制器的内部标志。一旦接受，可以使用其他的设置特性命令而不用改变前面特性的设置。大多数老式的 AT 控制器不支持该命令，但是大多数 DIDE 控制器支持它。表 11-18 显示了特性代码。

表 11-18 特性代码

十六进制	特 性
1	开放 8 位数据传送，而不是普通的 16 位传送*
2	开放写高速缓存*
3	基于扇区计数寄存器中的值设置传送模式
4	开放所有的自动缺陷再分配*
22	写相同，用户指定的区域（参看写相同命令，E9h）*
33	禁止重试*
44	指定在带 ECC 读/写时返回的 ECC 字节长
54	在扇区计数寄存器中设置高速缓存的段值*
55	禁止预测特性
66	禁止回到上电缺省状态
77	禁止 ECC*
81	禁止 8 位数据传送，使用 16 位（普通）*
82	禁止写高速缓存*
84	禁止所有的自动缺陷再分配*
88	开放 ECC（普通）*
99	开放重试（普通）*
9A	设置设备最大平均电流（以控制膝上型电脑的功耗）*
AA	开放预测特性
AB	设置从扇区计数寄存器的最大预取*
BB	在带 ECC 写/读时返回 4 字节
CC	开放回到上电缺省状态
DD	写相同，整个磁盘（参看写相同命令，E9h）*

* 可能并非所有的 EIDE 控制器都支持

命令	描述	端口
F1h	安全设置密码（有限）	1F7h

如果没有锁住或冻结安全性，则向磁盘传送密码扇区（512 字节）。表 11-19 显示了密

码扇区所包含的信息。

表 11-19 设置密码扇区数据

偏移量（十六进制）	大 小	描		述
0	字节	控制位		
		位	15~7=x	未使用或保留
			8=0	安全等级高
			1	安全等级最大
			7~1=x	未使用或保留
			0=0	设置用户密码
			1	设置管理员密码
2	32 字节	密码字节		
22h	478 字节	未使用		

命令	描述	端口
F2h	安全开锁（有限）	1F7h

如果当前锁住了一个驱动器则开锁它。表 11-20 显示了传送到控制器的 512 字节扇区密码。如果接受了密码，则开锁驱动器。五次失败重试之后，会中止该命令和安全擦除单元（F4h）命令，直到出现硬件重启。

表 11-20 密码扇区数据

偏移量	大小	描		述
0	字	控制位		
		位	15~1=0	留作将来使用
			0=0	比较用户密码
			1	比较管理员密码
2	32 字节	密码字节		
22h	478 字节	未使用（设定为零）		

命令	描述	端口
F3h	安全擦除准备（有限）	1F7h

在触发安全擦除单元（F4h）命令之前立即触发该命令。它有助于阻止无意地擦除数据。参看安全擦除单元获取其他细节。

命令	描述	端口
F4h	安全擦除单元（有限）	1F7h

如果密码匹配，则擦除驱动器上所有的用户数据。以 512 字节扇区形式传送给控制器的密码如表 11-19 所示。必须在该命令之前使用安全擦除准备命令（F3h），否则中止该命令。

命令	描述	端口
F5h	安全冻结锁住（有限）	1F7h

触发该命令时，激活了一个冻结的模式，在该模式下忽略其他的安全性命令，直到驱动器关掉电源。一旦电源恢复，则自动释放冻结模式。

命令	描述	端口
F6h	安全禁止密码（有限）	1F7h

禁止锁住驱动器。表 11-19 中列出了以 512 字节扇区形式传送给控制器的密码。如果接受了密码，则不再允许锁住控制器。

端口	类型	描述	平台
320h	输入	硬盘控制器状态	PC/XT

PC/XT 类型控制器使用该端口来从控制器获取状态。在执行一些命令时，寄存器会返回该命令对应的信息。在所有其他情况下，寄存器返回一个状态字节。如果状态指示出错，则应触发传感状态命令 3 来获取有关该错误的其他信息。

在读该输入状态字节时，将改变控制器的状态，表明完成了上条命令，然后清除忙标志位。

输入（位 0~7）

字节 0——状态字节

位		
	7r=0	未使用
	6r=0	未使用
	5r=0	驱动器 0 状态
	1	驱动器 1 状态
	4r=0	未使用
	3r=0	未使用
	2r=0	未使用
	1r=0	未发生错误
	1	在执行该命令时发生了错误（触发一条请求传感状态命令来获取错误信息及地址）
	0r=0	未使用

端口	类型	描述	平台
320h	输出	硬盘命令	PC/XT

PC/XT 控制使用该端口来指定和必要数据一起使用的命令。

输入（位 0~7），数据控制块（6 字节，或 14，供初始化）

字节 0——命令（如表 11-21 所示）

字节 1——驱动器和磁头号

位		
	7=0	总设置为 0（未使用）
	6=0	总设置为 0（未使用）
	5=0	选择驱动器 0
	1	选择驱动器 1
	4=x	磁头号（0~1Fh）
	3=x	
	2=x	
	1=x	
	0=x	

字节 2——磁柱和扇区号　　位 6 和 7=磁柱号的高两位

0~5=扇区号（1~63）

字节 3——磁柱号（低 8 位）

字节 4——要传送的扇区数据（读、写和检查），1~16 交错（格式化命令）

字节 5——控制字节

位		
	7=0	对任何磁盘访问命令重试四次
	1	禁止重试
	6=0	在读和检查命令时，如果出现了 ECC 错误但仍不能确定读了错误数据，则重读一次。对所有其他的命令将该位设置 0。
	1	如果出现了 ECC 错误，不重读
	5=0	未使用
	4=0	未使用
	3=0	未使用
	2=x	设置步进速率
	1=x	
	0=x	

位 2	位 1	位 0	步进速率
0	0	0	=3ms
1	0	0	=200ms
1	0	1	=70ms（普通）
1	1	0	=3ms
1	1	1	=3ms

下面的字节只提供给初始化命令 0Ch。

字节 6——驱动器中磁柱的最大数目（低 8 位）

字节7——驱动器中磁柱的最大数目（高8位）
字节8——磁头的最大数目
字节9——减少的写电流开始的磁柱（低8位）
字节A——减少的写电流开始的磁柱（高8位）
字节B——最大ECC阵发数据长度（0Bh是一般情况下的值）

表11-21 PC/XT命令

十六进制	命令
0	测试驱动器是否准备好
1	重校
3	请求传感状态
4	格式化驱动器
5	检查多块扇区
6	格式化磁柱
7	格式化带坏扇区的磁柱
8	读多块扇区
A	写多块扇区
B	查找
C	初始化驱动器参数
D	获取ECC阵发错误长度
E	从扇区缓冲区中读数据
F	向扇区缓冲区中写数据
E0	控制器RAM诊断
E3	驱动器诊断
E4	控制器内部诊断
E5	带ECC数据读多块扇区
E6	带ECC数据写多块扇区

端口320h命令细节

命令	描述	端口
0h	测试驱动器是否准备好	320h

控制器更新读自端口321h的硬盘控制状态。该命令只需要定义字节1的驱动器选择位，第5位。不必考虑所有其他的位。

命令	描述	端口
1h	重校	320h

将磁头定位到磁柱 0。该命令只须定义字节 1 的驱动器选择位，第 5 位和字节 5。不必考虑所有其他的位。

命令	描述	端口
3h	请求传感状态	320h

命令控制器获取四个传感字节。该命令字节只须定义字节 1 的驱动器选择位，第 5 位。不必考虑其他所有的位。

如果检测到一个错误，硬盘 BIOS 会获得传感数据，并将错误转化为一个标准的错误返回码。在控制器的错误码和 BIOS 返回状态码之间没有一一对应关系。

在触发了这个命令块之后，可以从端口 320h 按顺序读取四个传感字节，它们的内容如下：

字节 0——错误字节

位		
	7=0	前面的命令没有使用磁盘地址
	1	前面的命令使用了磁盘地址
	6=0	未使用
	5~0=	错误码，十六进制（括号内的数表示由磁盘 BIOS 返回的十六进制状态码）
	0	没有检测到错误（0）
	1	索引信号丢失——在某次旋转出现第一个扇区之前通常会插入一个索引信号（20h）
	2	查找未完成——驱动器命令查找完成，但没有出现信号（ 40h）
	3	写错误——上次操作期间驱动器指示控制器有一个写错误（20h）
	4	驱动器没有准备好——选中了驱动器之后，驱动器没有发出准备好信号（80h）
	6	重校失败——在向外推进了最大数目的磁柱后，驱动器没有指示它到达磁柱 0（20h）
	8	驱动器仍在查找而没有准备好。该状态只出现在驱动器准备好命令 0 的测试之后以及驱动器仍在查找指定的扇区之时。（40h）
	10h	ID 读取错误——扇区 ID 的 ECC 检查失败。检测到 ECC 但是没有修复 ID 错误（10h）
	11h	数据错误——在一次读操作中扇区有一个不可恢复的 ECC 错误（10h）
	12h	丢失地址标签——没有发现匹配请求的地址标签（2）
	14h	找不到扇区——磁柱和磁头有效，但是控制器找不到指定的扇区（4）
	15h	查找错误——在一次查找操作之后，磁柱和/或

磁头地址不能匹配查找地址（40h）

18h 可更正的数据错误——读取了期望的数据，并被 ECC 更正（11h）

19h 坏磁道——上次操作中检测到了坏磁道标志。无重试(0Bh)

20h 无效命令（1）

21h 坏磁盘地址——磁柱、磁头、扇区地址超过了磁盘边界（2）

30h RAM 错误——控制器 RAM 诊断测试 E0h 失败(20h)

31h 检查错误——控制器内部诊断测试 E4h 失败，原因是其内部编程存储器出现了检查错误。

32h 内部 ECC 错误——控制器内部诊断测试 E4h 无法测试 ECC 生成器(10h)

字节 1——当前的驱动器磁头（如果出错，则指明错误的位置）

位 7=0 未使用

6=0 未使用

5=0 驱动器 0

1 驱动器 1

4=x | 3=x | 2=x | 1=x | 0=x 磁头号，0~31

字节 2——磁柱和扇区（如果出错，则指明错误的位置）

位 6 和 7=磁柱号的第 8 和 9 位

0~5=扇区号

字节 3——磁柱，10 位磁柱号的低 8 位（如果出错，则指明错误的位置）

命令	描述	端口
4h	格式化驱动器	320h

格式化驱动器。必须使用数据控制块的全部六个字节，但是字节 2 中的扇区域必须设置为 0。

命令	描述	端口
5h	检查多个扇区	320h

检查指定扇区的 ECC 是否有效。必须使用数据控制块的全部六个字节。

命令	描述	端口
6h	格式化磁柱	320h

格式化指定的磁柱。必须使用数据控制块的全部六个字节，但是字节 2 中的扇区域必须设置为 0。

命令	描述	端口
7h	格式化带坏扇区的磁柱	320h

格式化指定的磁柱，并将每个扇区标注上坏扇区标签。必须使用数据控制块的全部六个字节，但是字节 2 中的扇区域必须置 0。

命令	描述	端口
8h	读多个扇区	320h

读取一定数目的指定扇区。必须使用数据控制块的全部六个字节。

命令	描述	端口
Ah	写多个扇区	320h

写入一定数目的指定扇区。必须使用数据控制块的全部六个字节。

命令	描述	端口
Bh	查找	320h

将磁头移动到指定的磁柱。必须使用数据控制块的全部六个字节，但是字节 2 中的扇区域必须置为 0。不必考虑字节 4 中的扇区计数值。

命令	描述	端口
Ch	初始化驱动器参数	320h

向控制器中装入驱动器参数，该特殊情况必须传送总共 14 字节。不必考虑数据控制块的字节 1 到 5。但是必须定义字节 1 的驱动器选择位，第 5 位。

命令	描述	端口
Dh	获取 ECC 阵发错误长度	320h

返回控制器的 ECC 阵发错误长度的当前设置。不必考虑数据控制块的字节 1 到 5。在接受该命令之后，它读取端口 320h 来获取阵发错误长度值。该值是更正后的连续位的数目。

命令	描述	端口
Eh	从扇区缓冲区读数据	320h

读取控制器的内部扇区缓冲区 RAM 的内容，仅供诊断之用。不必考虑数据控制块的字节 1 到 5。

命令	描述	端口
Fh	向扇区缓冲区写数据	320h

向控制器的内部扇区缓冲区 RAM 写入数据，仅供诊断之用。不必考虑数据控制块的字节 1 到 5。

命令	描述	端口
E0h	控制器 RAM 诊断	320h

启动控制内部诊断其 RAM，状态寄存器（读自端口 320h）将提供测试的结果。不必考虑数据控制块的字节 1 到 5。

命令	描述	端口
E3h	驱动器诊断	320h

启动指定驱动器的诊断。状态寄存器（读自端口 320h）将提供测试的结果。不必考虑数据控制块的字节 1 到 5，但是必须定义字节 1 中的驱动器选择位，第 5 位。必须设置字节 5，控制字节。

命令	描述	端口
E4h	控制器内部诊断	320h

启动控制器内部诊断，状态寄存器（读自端口 320h）将提供测试的结果。不必考虑数据控制块的字节 1 到 5。

命令	描述	端口
E5h	读带 ECC 数据的多个扇区	320h

读取一定数目的指定扇区，每个读取的扇区将包括该扇区的 4 字节 ECC 数据。这意味着每个扇区长 516 字节。必须使用数据控制块的全部六个字节。

命令	描述	端口
E6h	写带 ECC 数据的多个扇区	320h

写入一定数目的指定扇区。每个写入的扇区将包括该扇区的 4 字节 ECC 数据。这意味着每个扇区长 516 字节。必须使用数据控制块的全部六个字节。

端口	类型	描述	平台
321h	输入	硬盘控制器硬件状态	PC/XT

从适配卡读取状态。如果低四位值为 0Dh（锁存忙，等待输入命令、请求准备好），则控制器准备好接收 6 字节的控制块。参看端口 320h 以获取控制块的细节。

如果准备好发送一个控制块，但控制器尚未准备好（如上所述），则应重读状态，直到控制器准备好或者某些预设定的超时值用完。

输入（位 0~7）

位		
	7r=x	无连接，任意值
	6r=x	无连接，任意值
	5r=1	激活了中断请求 5
	4r=1	激活了 DMA 请求 3
	3r=0	不忙
	1	设置了锁存忙
	2r=0	控制器等待数据（端口 320h）
	1	控制器等待命令（端口 320h）
	1r=0	控制器等待输入（端口 320h）
	1	控制器等待输出（端口 320h）
	0r=0	无请求
	1	请求准备好

端口	类型	描述	平台
321h	输出	硬盘控制器重启	PC/XT

重启硬盘控制器。忽略任何发送的值。

端口	类型	描述	平台
322h	输入	硬盘获取类型开关	PC/XT

读取适配卡上的开关设置。这些开关指定了驱动器 0 和驱动器 1 所使用的参数表。每个驱动器可以从四个固定的参数表中选择一个。BIOS 使用该端口来获取开关设置以及确定使用哪个参数表。参看表 11-71 以了解开关设置是如何选择参数表的。

输入（位 0~7）

位		
	7r=x	没有连接，任意值
	6r=x	没有连接，任意值
	5r=x	没有连接，任意值
	4r=x	没有连接，任意值
	3r=0	开关 1 开（驱动器 0）
	2r=0	开关 2 开（驱动器 0）
	1r=0	开关 3 开（驱动器 1）
	0r=0	开关 4 开（驱动器 1）

端口	类型	描述	平台
322h	输出	使硬盘控制器忙	PC/XT

任何输出到该端口的值都会锁存适配卡来表示它正忙。装入该命令，并且执行该命令之后，控制器会发送一个中断请求（IRQ5，如果允许）来标示操作完成。必须从控制器读取状态来清除忙锁存。

端口	类型	描述	平台
323h	输出	硬盘 DMA 和内部中断屏蔽	PC/XT

该寄存器控制是否允许 DMA 和/或中断请求。

输入（位 0~7）

位		
7r=x		未使用
6r=x		未使用
5r=x		未使用
4r=x		未使用
3r=x		未使用
2r=x		未使用
1r=0		禁止中断请求
1		允许 IRQ5 中断请求
0r=0		禁止 DMA 请求
1		允许 DMA 通道 3 请求

端口	类型	描述	平台
276h	输入	硬盘替代状态（第四个控制器）	AT/EISA

第四个硬盘适配器的替代地址，参看端口 3F6h，输入。

端口	类型	描述	平台
276h	输出	硬盘适配器寄存器（第四个控制器）	AT/EISA

第四个硬盘适配器的替代地址，参看端口 3F6h，输出。

端口	类型	描述	平台
277h	输入	硬盘驱动器地址寄存器（第四个控制器）	AT/EISA

第四个硬盘适配器的替代地址，参看端口 3F7h，输入。

端口	类型	描述	平台
2F6h	输入	硬盘替代状态（第三个控制器）	AT/EISA

第三个硬盘适配器的替代地址，参看端口 3F6h，输入。

端口	类型	描述	平台
2F6h	输出	硬盘适配器寄存器（第三个控制器）	AT/EISA

第三个硬盘适配器的替代地址，参看端口 3F6，输出。

端口	类型	描述	平台
2F7h	输入	硬盘驱动器地址寄存器（第三个控制器）	AT/EISA

第三个硬盘适配器的替代地址，参看端口 3F7h，输入。

端口	类型	描述	平台
376h	输入	硬盘替代状态（第二个控制器）	AT/EISA

第二个硬盘适配器的替代地址，参看端口 3F6h，输入。

端口	类型	描述	平台
376h	输出	硬盘适配器寄存器（第二个控制器）	AT/EISA

第二个硬盘适配器的替代地址，参看端口 3F6，输出。

端口	类型	描述	平台
377h	输入	硬盘驱动器地址寄存器（第二个控制器）	AT/EISA

第二个硬盘适配器的替代地址，参看端口 3F7h，输入。

端口	类型	描述	平台
3F6h	输入	硬盘替代状态	AT/EISA

该端口返回与硬盘状态（端口 1F7h）提供的信息相同的信息。与 1F7h 不同的是，读端口 3F6h 不会清除挂起的中断和隐含中断确认。

如果设置了控制器位 7 忙，则所有其他位都没有意义。另外，忙时读取任何控制器端口 1F0h 都会返回无意义的结果。当忙位从忙过渡到不忙时，在过渡后的 400ns，寄存器中的其他位以及端口 1F0h 到 1F7h 中的值才会有效。

输入（位 0~7）

位		
	7r=0	控制器不忙
	1	控制器忙，位 0~6 无意义（参看上文）
	6r=0	选中的驱动器未准备好
	1	选中的驱动器准备好，可能响应来自控制器的命令。如果第 4 位置 1，那么可以读写或查找

5r=1	选中的驱动器发生写错误
4r=1	选中的驱动器完成了查找工作，现在磁头位于所要求的位置
3r=1	激活数据请求——控制器指示它已准备好。从扇区缓冲区发送和接收字。参看数据端口 1F0h 以了解细节
2r=1	更正数据——读自磁盘的数据带有一个可更正的数据错误，该错误已被供应商的 ECC 算法改正了。该条件不阻止多扇区数据传送
1r=1	索引脉冲——在老式的驱动器上，这一痊反映了索引脉冲的状态，并且磁盘每旋转一周就设置一次该位。目前，供应商可能将该位用作其他用途或者不使用该位
0r=0	无错误
1	前面的命令中出现了错误——错误寄存器指示了错误的属性。向控制器送下一条命令时会清除该位

端口	类型	描述	平台
3F6h	输出	硬盘适配器控制寄存器	AT/EISA

该只写寄存器保存的位控制重启、中断请求、磁头信息以及其他条件。来自磁盘参数表的控制字节在执行该命令之前由中断 13h 处理程序写入到这个端口。

如果允许中断请求（位 1），则主控制器连接到 IRQ14，后者可以处理主驱和从驱。如果使用第二个控制器（通常使用端口 3F6h 作为硬盘适配器控制寄存器），大多数适配器将 IRQ15 作为第二个控制器的缺省连接。第三个和第四个控制器还没有标准，每个控制器通常会要求自己的 IRQ。

要重启控制器，只要将第 2 位置 1 并保持 4.8 微秒或更长，然后清除第 2 位即可。

输入（位 0~7）

位	
7w=0	未使用
6w=0	未使用
5w=0	未使用
4w=0	未使用
3w=0	如果驱动器有 1 到 8 个磁头
1	驱动器有 8 个以上磁头
2w=0	普通操作
1	重启控制器
1w=0	允许中断请求（通常主控制器使用 IRQ14）
1	禁止中断请求

0w=0　　未使用

端口	类型	描述	平台
3F7h	输入	硬盘驱动器地址寄存器	AT/EISA

该端口返回当前选中驱动器的驱动器号、磁头号以及写状态。它们读自驱动器接口，可能并不反映期望的结果，其原因涉及到计时问题、控制器的高速缓存及其他。所有的六位以反转状态返回。并非所有的控制器都支持该寄存器。

该端口与软盘控制器共享。如果存在软盘控制器，则会在第 7 位返回软盘状态改变位。我列出第 7 位为未使用，一态。这就意味着如果没有使用软盘适配器，则第 7 位返回任意值。

输入（位 0~7）

位		
	7r=x	未连接——可能返回任意值（参看上文）
	6r=0	驱动器正在写
	5r=x 4r=x 3r=x 2r=x	当前选中的磁头（求反） 0000=磁头 15 0001=磁头 14 …… 1111=磁头 0
	1r=0	选中了驱动器 1，并激活了它（从驱）
	0r=0	选中了驱动器 0，并激活了它（主驱）

端口	类型	描述	平台
3510h	输入	ESDI 硬盘驱动器状态（16 位）	PS/2

在完成一条命令后从控制器读取状态信息。控制器用中断请求 IRQ14，中断 76h 标志系统。与触发的命令有关，可以读回某些字。参看端口 3510h，输出，获取每条命令的专门说明以及读这个端口时它们返回的数据。

端口	类型	描述	平台
3510h	输出	ESDI 硬盘命令（16 位）	PS/2

该 16 位端口用来向控制器传送命令块。该命令进程开始于来自注意寄存器（端口 3513h）的命令请求。然后发送块，一次一个字。每次传送每个字节都执行一次握手，所以在发送命令字时不必要 I/O 延时。

在命令块的每个字传送之前，读取基本状态寄存器，并检查第 2 位。第 2 位是零，表示它已为下一个命令字做好准备。如果第 2 位不是零，则表示控制器还没有为下个字做好准备。应该连续读取状态直到第 2 位被清除，然后控制器接受下一个命令块字。

输出（位 0~7），命令控制块

该命令块是 2 个或 4 个字，这与具体的命令有关。下面显示了最常见的命令块。任何不同于该结构的命令在命令的专门说明内做了解释。

字 0—— 命令（如下所述）
字 1——要传送的扇区数
字 2——相对扇区地址，低位字
字 3——相对扇区地址，高位字

十六进制	命令
0606	将磁头停靠在驱动器 0 上
0607	获取驱动器 0 上的命令完成状态
0608	获取驱动器 0 上的设备状态
0609	获取驱动器 0 上的配置
0615	获取驱动器 0 上的制造头标
0616	低级格式化驱动器 0
0617	格式化准备驱动器 0
0626	将磁头停靠在驱动器 1 上
0627	获取驱动器 1 上的命令完成状态
0628	获取驱动器 1 上的设备状态
0629	获取驱动器 1 上的配置
0635	获取驱动器 1 上的制造头标
0636	低级格式化驱动器 1
0637	格式化准备驱动器 1
06E7	获取控制器命令完成状态
06E9	获取控制器配置
06EA	获取可编程选项选择信息
06EB	翻译相对扇区地址
4601	从驱动器 0 读数据
4602	向驱动器 0 写数据
4603	检查驱动器 0 上的数据
4604	带检查写驱动器 0
4605	在驱动器 0 上查找
4621	从驱动器 1 读数据
4622	向驱动器 1 写数据
4623	检查驱动器 1 上的数据
4624	带检查写驱动器 1
4625	在驱动器上查找

端口 3510h 命令细节

命令	描述	端口
0606h	将磁头停靠在驱动器 0 上	3510h

向驱动器 0 发送一条停止马达命令。驱动器将磁头移动到其安全的装入分区，并且关闭马达。驱动器将显示“未准备好”，直到移走电源。这意味着在重新访问驱动器之前必须有一个断电周期。

停靠磁头命令使用仅两个字的命令块。命令块的第二个字被设置为零。

命令	描述	端口
0607h	获取驱动器 0 上的命令完成状态	3510h

命令控制器返回驱动器 0 的命令完成状态块。该命令只使用两个字，其中第二个字设置为零。

在命令完成时，从端口 3510h 读取一组 7 字的命令完成状态块。表 11-22 显示了该块。

表 11-22 命令完成状态块

偏移量	大 小	描 述	
0	字节	07h（返回命令值）	
1	字节	该状态块中的字数（7）	
2	字节	十六进制的命令错误码（括号中的值是 PS/2 BIOS 返回的十六进制状态码。表 11-5 解释了返回状态码）	
		0	成功，无错误（BB）
		1	无效参数（1）
		2	未使用或未知功能（BB）
		3	不支持命令（1）
		4	忽略命令（80）
		5	未使用或未知功能（BB）
		6	控制器诊断失败，命令被拒绝（20）
		7	格式化失败——在格式化之前要求格式化准备命令（D）
		8	格式化错误——主映射存在问题（D）
		9	格式化错误——次映射存在问题（D）
		A	格式化错误——诊断失败（D）
		B	格式化警告——次映射太大（0）
		C	格式化警告——非零缺陷（10）

续表

偏移量	大 小	描 述	
		D	格式化错误——系统检查错误（D）
		E	格式化警告——设备不兼容（0）
		F	格式化警告——推入表溢出
		10	格式化警告——磁柱上的推入多于15个
		11	内部硬件错误（20）
		12	格式化警告——发现了检查错误（0）
		13	设备对命令无效（1）
		FF	设备错误
3	字节	十六进制的命令状态码	
		1	成功地完成了命令
		3	ECC后成功地完成了命令
		5	重试并成功地完成了命令
		6	部分格式化完成
		7	ECC及重试后成功地完成了命令
		8	命令完成，带有警告（参见字节2）
		9	中止完成
		A	重启完成
		B	数据传送准备好，无状态块
		C	命令完成，带有错误（参看字节4或5）
		D	DMA错误——待修理，重新运行整个命令
		E	命令块错误（参看字节2）
		F	注意码坏掉
4	字节	设备误码，第1组（括号中的值是PS/2 BIOS返回的十六进制状态码。表11-5解释了返回状态码）	
		0	无错误（BB）
		1	查找失败，但是检测到了设备
		2	由于奇偶性，注意或命令完成错误造成接口错误（20）
		3	不能定位扇区ID（4）
		4	没有格式化（2）
		5	不可恢复的ECC错误（10）
		6	扇区ID ECC错误——能检测到ECC，但是不能更正扇区ID错误（10）
		7	相对扇区地址越界（1）
		8	超时错误（BB）
		9	缺陷扇区（A）
		A	可移走媒质发生了改变（6）

续表

偏移量	大　小	描　　述		
		B	选择错误（1）	
		C	可移走媒质写保护（3）	
		D	写错误（CC）	
		E	读错误（BB）	
		F	无索引或扇区脉冲，可能归因于有缺陷的驱动器或缆线（2）	
		10	设备未准备好（10）	
		11	查找错误，但是检测到了适配器（40）	
		12	坏格式（A）	
		13	容量溢出（BB）	
		14	没有发现数据地址标签（2）	
		15	没有发现扇区 ID，没有 ID 地址标签（2）	
		16	丢失了设备配置数据（BB）	
		17	丢失了第一个和最后一个相对扇区地址标志（4）	
		18	磁道上没有发现扇区 ID（4）	
		81	超时——等待停止	
		82	超时——等待数据传送结束	
		84	格式化期间等待数据传送时停止	
		85	超时——等待磁头开关	
		86	超时——等待 DMA 完成	
5	字节	设备错误码，第 2 组		
		位	7=0	未使用
			6=0	未使用
			5=0	未使用
			4=1	准备好
			3=1	选中
			2=1	写错误
			1=1	磁道 0 标志
			0=1	查找或命令完成
6	字节	因错误或忽略操作而尚未处理的扇区数（停靠磁头及查找命令返回零）		
8	双字	上次处理的相对扇区地址		
C	字	需要错误恢复的扇区数——ECC 更正的扇区数（停靠磁头及查找命令返回零）		

命令	描述	端口
0608h	获取驱动器 0 的设备状态	3510h

命令控制器返回驱动器 0 的状态块。该命令只使用两个字，其中第二个字设为零。在命令完成时，从端口 3510h 读取一组九个字的状态块。表 11-23 显示了该块。

表 11-23　设备状态块

偏移量	大　小	描　　述		
0	字节	08h（返回命令值）		
1	字节	在这个块中的字数（9）		
2	字节	错误位		
		位	7~4=0	未使用
		位	3=1	等待激活来自驱动器的命令完成线，超时
			2=1	数据传送时出了奇偶错误
			1=1	等待来自驱动器的传送确认线使进入非激活状态，超时
			0=1	等待激活来自驱动器的传送确认线，超时
3	字节	未使用		
4	字节	命令错误码（参看获取命令完成状态（命令 607h）以了解错误码）		
5	字节	命令状态字节（参看获取命令完成状态（命令 607h）以了解错误码）		
6	字	ESDI 标准状态		
8	字	ESDI 驱动器供应商定义的状态#1		
A	字	ESDI 驱动器供应商定义的状态#2		
C	字	ESDI 驱动器供应商定义的状态#3		
E	字	ESDI 驱动器供应商定义的状态#4		
10	字	ESDI 驱动器供应商定义的状态#5		

命令	描述	端口
0609h	获取驱动器 0 的配置	3510h

命令控制器返回驱动器 0 的配置状态块。该命令只使用两个字节。其中第二个字设置为 0。

在执行完成时，从端口 3510h 读取一组六个字的配置状态块。表 11-24 显示了该块。

表 11-24　配置状态块

偏移量	大　小	描　　述
0	字节	09h（返回命令值）
1	字节	在这个状态块中的字数（6）
2	字节	标志

续表

偏移量	大　小	描　　述			
		位	7~5=0	未使用	
			4=1	无效的次级缺陷映射——次级缺陷映射的首扇区不可读。push 表及缺陷的相关信息未知	
			3=1	标志了零缺陷	
			2=1	倾斜扇区——将驱动器格式化为带倾斜扇区的驱动器，该驱动器的磁头切换时间较长。例如，倾斜度为 2 会如下放置扇区：	
				磁 头	扇 区 号
				0	1 2 3 4 5 6 7 8 9 10
				1	9 10 1 2 3 4 5 6 7 8
				2	7 8 9 10 1 2 3 4 5 6
倾斜与交错无关					
			1=1	驱动器带可移走媒质	
			0=0	读取和定位出错时执行普通的重试	
			1	禁止所有的重试	
3	字节	每磁柱的空余扇区数			
4	双字	相对扇区地址的总数（可用扇区）			
8	字	磁柱总数			
A	字节	每磁柱的磁道数			
B	字节	每磁道的扇区数			

命令	描述	端口
0615h	获取驱动器 0 的制造头标	3510h

读取驱动器 0 的制造头标及缺陷映射 1。制造头标位于缺陷映射 1 的起点，读一个扇区会获得该头标。头标数据中有缺陷映射的大小，紧接着头标的就是缺陷映射 1。这个命令的功能类似于读命令，并且通过 DMA 处理数据。该命令只要两个字，其中第二个字表示要读的扇区数。

命令	描述	端口
0616h	低级格式化驱动器 0	3510h

对整个驱动器 0 执行低级格式化。它提供了包括缺陷映射信息的选项以跳过有缺陷的扇区。参看格式化准备（命令 0617h）以了解其他细节。

低级格式化命令的命令块只要求两个字。命令块的第一个字指定了几种选项和 4 字节缺陷的数目。第二个字有下述功能：

位		
	15=0	未使用
	14=0	未使用
	13=0	未使用
	12=1	每次磁柱操作完成后发送中断
	11=1	执行表面分析
	10=1	保存表面分析的结果
	9=1	格式化时忽略次级缺陷映射
	8=1	格式化时忽略主级缺陷映射
	7=x 6=x 5=x 4=x 3=x 2=x 1=x 0=x	4 字节缺陷入口的数目 (参看本章中断 13h 功能 AH=1Ah 下的表 11-8，以了解其结构)

如果缺陷入口域非零，格式化将期望通过 DMA 读取缺陷映射表的方式同一条写命令完全相同。参看中断 13h，AH=1Ah（低级格式 ESDI 驱动器），以了解更多的对低级格式化的深入解释。并请参看 CL 寄存器的位描述，它们和这里第二个命令字的描述相同。

命令	描述	端口
0617h	格式化准备驱动器 0	3510h

在格式化驱动器命令之前必须马上触发这个命令。格式化准备提供了一种安全的联锁机制，这样才不会意外发生低级格式化。如果格式化之前没有触发格式准备命令，那么将不会格式化驱动器，并返回一个序列错误。格式化准备的命令控制块只使用两个字，其中第二字必须是 55AAh。

另外，格式准备传递给控制器一个或两个扇区的数据，包括那些格式化将标为缺陷的扇区地址。通过 DMA 类似于数据写那样处理该数据。该数据使用一种两字绝对扇区地址（ASA）的重复式结构。在最后一个缺陷地址之后，所有剩下的字节都存入 FFh，每扇区的最后一个字节除外。如果加起所有 512 字节的扇区，并忽略溢出，结果应该是零。未使用的入口填入 FFFFFFFFh。表 11-25 显示了一个例子的内容，该例子缺陷表只使用一个扇

区，尽管允许使用两个。

表 11-25　绝对扇区地址的缺陷表举例

偏移量	大　　小	描　　述
0	双字	第一个 ASA
4	双字	第二个 ASA
1E4	双字	最后一个 ASA
1E8	双字	未使用（设定为 0FFFFFFFFh）
……所有剩下的未使用双字都设置为 0FFFFFFFFh		
1FC	字	0FFFFh
1FE	字节	0FFh
1FF	字节	检查前 511 字节

命令	描述	端口
0626h	停靠驱动器 1 的磁头	3510h

向驱动器 1 发一条命令来停止马达，并将磁头移动到其安全装入区域。参看命令 606h 可了解相关的细节。

命令	描述	端口
0627h	获取驱动器 1 的命令完成状态	3510h

命令控制器返回驱动器 1 的命令完成状态。参看命令 607h 了解相关细节。

命令	描述	端口
0628h	获取驱动器 1 的设备状态	3510h

命令控制器返回驱动器 1 的状态块。参看命令 608h 了解相关细节。

命令	描述	端口
0629h	获取驱动器 1 的配置	3510h

命令控制器返回驱动器 1 的配置状态块。参看命令 609h 了解相关细节。

命令	描述	端口
0635h	获取驱动器 1 的制造头标	3510h

命令控制器返回驱动器 1 的制造头标和缺陷映射 1。参看命令 615h 了解相关细节。

命令	描述	端口
0636h	低级格式化驱动器 1	3510h

对整个驱动器 1 执行低级格式化。参看命令 0637h 以及相类似的命令 616h 了解相关细节。

命令	描述	端口
0637h	驱动器 1 的格式化准备	3510h

必须在格式化驱动器 1 之前立即触发这条命令。参看相类似的命令 0617h 了解相关细节。

命令	描述	端口
06E7h	获取控制器的命令完成状态	3510h

命令控制器返回控制器的配置状态块。该命令只使用两个字，其中第二个字设置为 0。

在命令完成时，该配置状态块是一组读自端口 3510h 的六个字。表 11-24 显示了这个状态块，这个表位于命令获取驱动器 0 的配置（0609h）下。

命令	描述	端口
06E9h	获取控制器配置	3510h

命令控制器返回控制器的配置状态块。该命令只使用两个字，其中第二个字设置为 0。

在命令完成时，返回配置状态块是一组读自端口 3510h 的六个字。如表 11-26 所示。

表 11-26　配置状态块——控制器

偏移量	大　　小	描　　述
0	字节	E9h（返回命令值）
1	字节	这个状态块中的字数（6）
2	字	0（未使用）
4	双字	控制器微代码版本等级
8	字	0（未使用）
A	字	0（未使用）

命令	描述	端口
06EAh	获取可编程选项选择信息	3510h

获取可编程选项（POS）寄存器的内容。该命令只使用两个字，其中第二个字设置为 0。

表 11-27 POS 信息状态块

偏移量	大　　小	描　　述
0	字节	EAh（返回命令值）
1	字节	此状态块中的字数（5）
2	字	0FFDDh（固定为这个值）
4	字节	POS 寄存器 3
5	字节	POS 寄存器 2
6	字节	POS 寄存器 5（未使用，设置为 FFh）
7	字节	POS 寄存器 4（未使用，设置为 FFh）
8	字节	POS 寄存器 7（未使用，设置为 FFh）
9	字节	POS 寄存器 6（未使用，设置为 FFh）

命令	描述	端口
06EBh	翻译相对扇区地址	3510h

ESDI 驱动器有两种类型的寻址方式，分别是相对扇区寻址（RSA）和绝对扇区寻址（ASA）。RSA 用于读、写、校验及其他普通的操作。相对扇区寻址掩藏了缺陷，并使整个磁盘看起来像一个长长的线性阵列。如果必须分配这些有缺陷的块，则必须使用绝对扇区地址。这个功能实现将 RSA 翻译成 ASA。

该命令的第一个字没有使用，应将其设置为零。第三个和第四个字包含了要翻译的 RSA。完成翻译命令时，可以从命令完成状态中获得 ASA 翻译结果。使用功能 06E9h 来读取这个状态信息。字 4 是最低有效位字，而字 5 是最高有效位字。

命令	描述	端口
4601h	从驱动器 0 读数据	3510h

从驱动器 0 读取指定数目的扇区。使用 DMA 进行数据传送。

命令	描述	端口
4602h	向驱动器 0 写数据	3510h

向驱动器 0 写入指定数目的扇区。使用 DMA 进行数据传送。

命令	描述	端口
4603h	校验驱动器 0 上的数据	3510h

检验是否可以从驱动器 0 读取指定的数据以及所读取的每个扇区是否有效。如果检测到了 ECC 错误，则不会执行重试操作。不会比较磁盘数据和原始数据。该命令不使用 DMA 进行数据传送。

命令	描述	端口
4604h	带校验写驱动器0	3510h

向驱动器 0 写入指定数目的扇区。写入所有的扇区后自动执行校验操作。校验操作从磁盘读取数据，并检查每个扇区的 ECC 的有效性。不会比较这个数据和原始数据。如果出现了 ECC 错误，则必须重新一次或多次写操作。使用 DMA 进行数据传送。

命令	描述	端口
4605h	在驱动器0上查找	3510h

将驱动器 0 的磁头定位到相对扇区地址（RSA）所指示的地址。查找 RSA=0 的功能和重校磁头相同。

命令	描述	端口
4621h	从驱动器1读数据	3510h

从驱动器 1 读取指定数目的扇区。使用 DMA 进行数据传送。

命令	描述	端口
4622h	向驱动器1写数据	3510h

向驱动器 1 写入指定数目的扇区。使用 DMA 进行数据传送。

命令	描述	端口
4623h	校验驱动器1上的数据	3510h

检验是否可以从驱动器 1 读取指定的数据以及所读取的每个扇区是否有效。如果检测到了 ECC 错误，则不会执行重试操作。不会比较磁盘数据和原始数据。该命令不使用 DMA 进行数据传送。

命令	描述	端口
4624h	带校验写驱动器1	3510h

向驱动器 1 写入指定数目的扇区。写入所有的扇区后自动执行校验操作。校验操作从磁盘读取数据，并检查每个扇区的 ECC 的有效性。不会比较这个数据和原始数据。如果出现了 ECC 错误，则必须重新一次或多次写操作。使用 DMA 进行数据传送。

命令	描述	端口
4625h	在驱动器1上查找	3510h

将驱动器 1 的磁头定位到相对扇区地址（RSA）所指示的地址。查找 RSA=0 的功能和重校磁头相同。

端口	类型	描述	平台
3512h	输入	ESDI 硬盘基本状态	PS/2

该端口用于控制器的握手和状态。在读该端口之前，必须禁止中断。

输入（位 0~7）

位		
	7 r = 0	禁止DMA（从端口3512h（输出）的位1设置DMA控制）
	1	开放 DMA
	6 r = 0	没有中断挂起
	1	中断挂起，不会向控制器发送命令，直到清除了挂起的中断
	5 r = 1	正在执行命令——控制器目前正在执行一条命令。在发送第一个命令字节时开启该位，在向注意信号端口3513h发送中断请求结束命令时清除该位
	4 r = 0	不忙——向注意信号端口发送命令是可以接受的
	1	正在使用注意信号端口，不能向它发送命令。在向注意信号端口发送了命令请求码后设置该位。完成扇区传送、或者完成了重新启动、或者退出了命令请求，都会清除该位
	3 r = 0	状态不可用
	1	端口 3510h 的状态可用——读端口 3510 时会自动清除该位
	2 r = 0	端口 3510h 准备好接受命令
	1	命令接口端口正忙，不能接受命令。控制器读取命令后会自动清除该位，并准备好接受下一条命令
	1 r = 1	传送请求——控制器已经准备好，可以接受命令
	0 r = 1	激活了中断请求——读取中断状态端口后会自动清除该位

端口	类型	描述	平台
3512h	输出	ESDI 硬盘基本控制	PS/2

该端口用来控制中断和 DMA，并向控制器发送重启命令。

输入（位 0~7）

位		
	7 w = 0	普通操作
	1	适配器的硬件重启，而不用清除可编程选项选择（POS）信息。重启操作后这一位会自动回到 0。
	6 w = 0	未使用
	5 w = 0	未使用
	4 w = 0	未使用

3 w = 0	未使用
2 w = 0	未使用
1 w = 0	禁止 DMA 请求
1	开放 DMA 请求——这时，控制器可以发送 DMA 请求。如果 POS 对适配器配置适当，那么在磁盘控制器完成数据传送后，这一位会自动清零
0 w = 1	允许中断请求

端口	类型	描述	平台
3513h	输入	ESDI 硬盘中断状态	PS/2

获取中断状态，并清除当前的中断请求。执行完这个命令后，必须读取该端口来完成当前的命令循环。如果命令出现问题，则在这个寄存器中包含了基本错误信息。在读取了这个端口之后，读取状态接口寄存器端口 3510h 来获取状态块。状态块可能会指明出现问题的磁盘区域，这一点取决于缺省条件。

输入（位 0-7）

位		
7 r = x	位完成这个命令的设备	
6 r = x	位 7	位 6	位 5
5 r = x	0	0	0=驱动器 0
	0	0	1=驱动器 1
	1	1	1=控制器命令
4 r = 0	未使用	
3 r = x	中断请求的情况	
2 r = x	1 = 成功地完成	
1 r = x	3 = 成功地完成，带 ECC	
0 r = x	5 = 成功地完成，带重试	
	6= 完成了部分格式化，读状态	
	7= 成功地完成，带有 ECC 和重试	
	8= 完成命令，带有警告	
	9= 完成异常中止命令	
	A= 完成重启	
	B= 数据传送准备好	
	C= 命令完成，带有错误	
	D= DMA 错误——读状态接口寄存器。如果要更正错误，需重试	
	E= 命令块错误	
	F= 注意信号错误	

端口	类型	描述	平台
3513h	输出	ESDI 硬盘注意信号	PS/2

这个端口用于初始化一条命令，完成命令进程，并重启适配器。这个被装入的功能将一直保持激活状态，直到一个新的请求改变它时为止。

在改变注意信号寄存器时，必须禁止中断，并从基本状态寄存器（端口 3512h）读取状态。该状态必须显示为控制器不忙（位 4=0），无中断挂起（位 6=0）。如果满足条件，就可以发送注意信号请求。

命令请求

命令请求用于请求控制器从数据端口 3510h 接受一个命令块。一旦装入了这个命令块，则可以对其他不忙的设备做出其他的命令请求。这就意味着控制器可以同时执行不超过三条命令。驱动器 0 和驱动器 1 各执行一条，另一条命令给适配器。

完成请求后，会执行三项操作。首先，适配器在 IRQ 14（如果允许）上生成中断请求，以通知系统操作完成。第二，更新端口 3513h 处的中断状态请求，指示请求完成状态。第三，端口 3510h 处的硬盘状态字将是命令完成状态块的第一个字。

中断结束请求

执行完一条命令以及生成中断指示命令完成之后，各个寄存器会保存一系列信息，例如来自控制器的中断状态和传感数据。这些数据将一直保持下去，直到触发了中断结束请求。控制器将不支持出现另外的中断，直到对端口 3510h 触发了这个请求为止。如果在读取完所有的传感数据之前触发了中断结束请求，则控制器认为系统不需要剩下的传感数据，并忽略它们。

异常中止命令请求

在一次适当的超时限定期间内，如果控制器没有触发命令完成中断，那么应触发异常中止命令来结束当前的命令操作。接收到中止命令后，从端口 3510h 读取传感数据，可能会有助于查明问题的起因。如果接受并完成了异常中止命令请求，则会生成一个中断。

重启请求

重启请求会中止当前的命令，并重启适配器。这个重启命令的操作类似于上电重启。重启命令将执行内部控制器自检，并重校所有的驱动器。重启完成时，触发中断 IRQ 14。控制器自检和重启的结果放置在端口 3513h 的中断状态寄存器中。

输出（位 0~7）

位	7 w = x	选择设备

6 w = x	位 7	位 6	位 5
5 w = x	0	0	0=驱动器 0
	0	0	1=驱动器 1
	1	1	1=适配器命令
4 w = 0	未使用		
3 w = 0	未使用		
2 w = x	请求码（参看上文）		
1 w = x	位 2	位 1	位 0
0 w = x	0	0	1=命令请求
	0	1	0=结束中断
	0	1	1=放弃指令
	1	0	0=重启适配器

端口	类型	描述	平台
3518h	输入	ESDI 硬盘状态（16 位）	PS/2

第二个硬盘适配器的替代地址。参看端口 3510h，输入。

端口	类型	描述	平台
3518h	输出	ESDI 硬盘命令（16 位）	PS/2

第二个硬盘适配器的替代地址。参看端口 3510h，输出。

端口	类型	描述	平台
351Ah	输入	ESDI 硬盘基本状态	PS/2

第二个硬盘适配器的替代地址。参看端口 3512h，输入。

端口	类型	描述	平台
351Ah	输出	ESDI 硬盘基本控制	PS/2

第二个硬盘适配器的替代地址。参看端口 3512h，输出。

端口	类型	描述	平台
351Bh	输入	ESDI 硬盘中断状态	PS/2

第二个硬盘适配器的替代地址。参看端口 3513h，输入。

端口	类型	描述	平台
351Bh	输出	ESDI 硬盘注意信号	PS/2

第二个硬盘适配器的替代地址。参看端口 3513h，输出。

第 12 章

串行口

本章讨论常用的串行口。串行口看起来似乎已经有了许多较好的文档说明，然而，我还是揭示了许多有趣但是尚未公开的方面，包括一些高级 FIFO 模式，现在许多新型的调制解调器和串行口都支持它。我还列出了一些常见的 bug，老式的串行口需要对它们做出特别处理。

为了讲清楚串行口，我还提供了完整的源代码，它们的作用有两点，第一个用处是描述系统中所有的串行口，并检查它们是否有扩展特性。第二个用处是找出两个 PC 之间的最大串行数据传送率，这个传送率可高达 115,200 波特。

我还展示了我所编写的一个小型调试工具，SSPY。它可在屏幕上为任何一个串行口显示调制解调器状态线。这个 TSR 有完整的源代码，所以很容易修改它，以适应各种特殊的情况。

简　介

若使用标准的串行 I/O 适配器，所有的 PC 都可以支持不超过四个串行口。串行口通常按 RS-232C 标准设计以满足电气要求和信号约定。也存在其他的标准，例如 RS-422、位同步、同步数据链路控制（SDLC）、IEEE-488（GPIB）、电流环等等。就大多数情况而言，这些替代的方法已经不复存在，也很少有人使用它们。BIOS 只支持 RS-232C 串行口，所以使用其他的标准时需要附加驱动程序。下面的端口归纳了这些很少使用的标准用到的寄存器，但是没有给出详细的功能信息。

串行口用于调制解调器以及一些定点设备，像鼠标、滚动球、打印机、网络、膝上型电脑的数据传送器。串行口的最大速度取决于串行口适配器内的波特率发生器、BIOS 软件以及系统执行 BIOS 代码的速度。另外，如果系统同时在执行一般情况下的一个相同优先级或更高优先级的代码，那么将会大大降低可靠的传输速度。在本章的后面，我将展示一个工具，它能够确定串行口最大可靠的波特率。

图 12-1 显示了串行口的设计结构。串行口操作主要由 UART（通用同步收发器）芯片处理。UART 负责完成从串行数据到字节以及字节到串行数据的转化。UART 也控制许多信号线，这些信号线用于和调制解调器及其他设备通信。UART 支持同时收发数据。

早期的设计使用国家半导体公司（National Semicondctor）生产的 8250 芯片。后来又

改用 CMOS 版，16450，这两种芯片具有相同的功能。某些较新式的设计使用 16550 或其他系列。这些都提供了和 16450 相同的特性，但是它们也加入了一些缓冲来减轻 CPU 的负荷。我将以上这些都称作 UART，只是在讲述不同芯片之间的特殊差别时才不这么称呼。

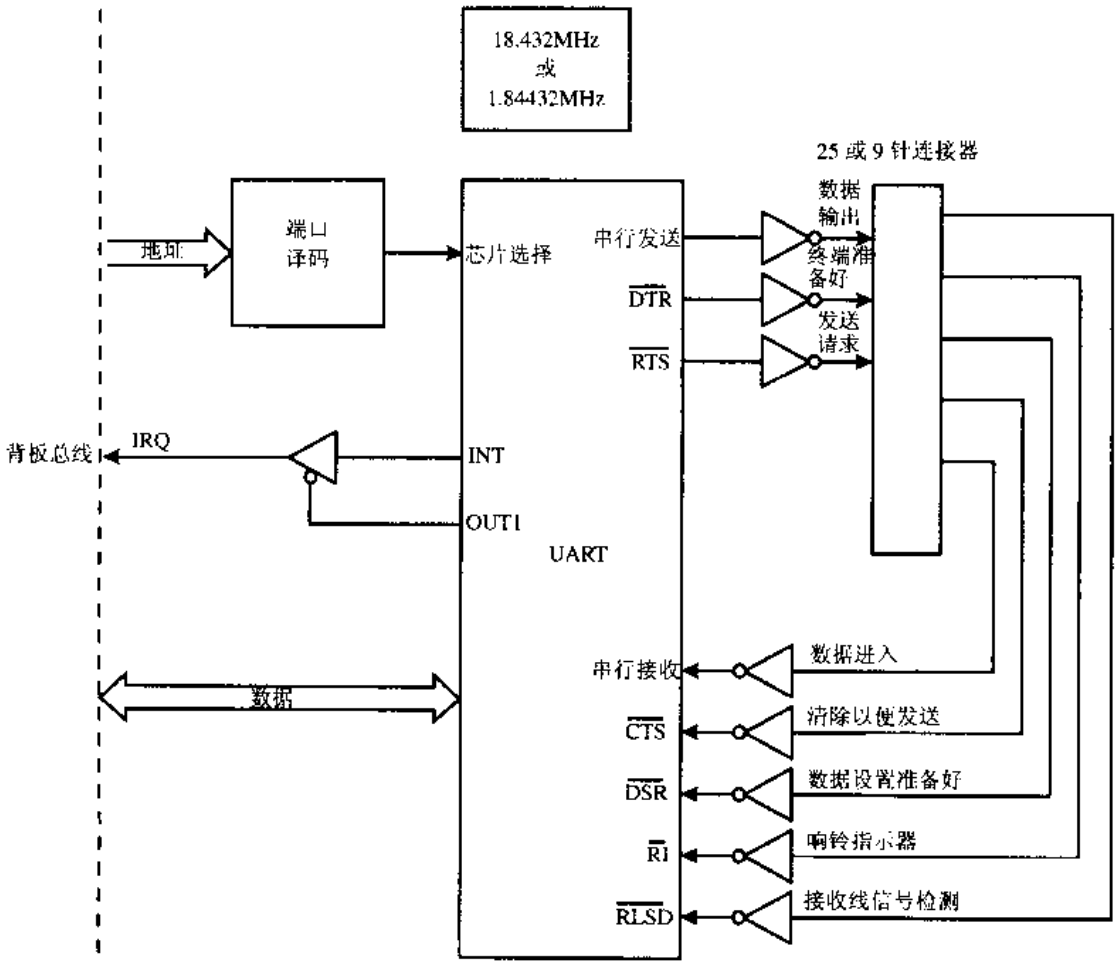


图 12-1 串行口

部分系统 BIOS（中断 41h）提供了一些服务来处理串行口与串行口适配器的通信。与并行口相似，POST 会检查是否带有串行口，并在 BIOS 数据区中记录活动的串行口的 I/O 地址。所有系统支持的都不超过四个串行口。BIOS 不支持另外的串行口。

串行口的编号及访问实在令人费解。共有四个可能的串行口，分别编号为 1、2、3 和 4。在 BIOS 数据区共有四个字保存了四个可能的串行口的 I/O 端口基地址。这就意味着串行口没有固定的 I/O 端口地址。

更令编程人员费解的是，所有的 BIOS 程序都使用从零开始算起的编号方式来指定端口。例如要访问端口 1，某个寄存器应设置为值 0。

BIOS 初始化

重启后，POST 检查是否安装了串行口。POST 先检查 I/O 端口组 3F8~3FFh，然后检查端口组 2F8~2FFh。为了检测活动的串行口，可从端口 3FAh 或 2FAh 读取中断标识寄存器。如果这个寄存器的第 3~7 位都是零，则 POST 认为这是一个活动的端口。一旦确定存在活动的串行 I/O 组，这个 I/O 端口基地址就被保存到未使用的最低串行口 BIOS RAM 地址中。起始于 40:0 的 RAM 被分配为活动串行口 I/O 地址。由于 IBM 没有为端口 3 和 4 定义标准的 I/O 端口地址，所以许多供应商的 POST 不会检查这些端口。

一般说来，大多数系统只检查两个串行口，当然，较新式的系统通常会检查四个可能的端口地址。MCA 系统会检查八个不同的可能串行口地址，以试图找到四个活动的串行口。表 12-1 列出了 BIOS 寻找活动端口的顺序。表中只列出了每组的基地址，MCA 系统一旦找到四个端口就不会继续检查其他的端口。

表 12-1　POST 检查串行 I/O 口的顺序

检查的顺序	大多数系统	新式的 AT 和 EISA 系统	MCA 系统
第一	3F8h	3F8h	3F8h
第二	2F8h	2F8h	2F8h
第三	无	3E8h	3220h
第四	无	2E8h	3228h
第五	无	无	4220h
第六	无	无	4228h
第七	无	无	5220h
第八	无	无	5228h

在完成串行口检查时，会保存串行口地址。表 12-2 列出了典型的四种端口地址的保存情况。情况 1 展示 POST 检测到两个串行口，情况 2 和 3 展示了只检测到一个串行口，情况 4 展示了没有检测到串行口的情况。

表 12-2　串行口检测到的可能结果

RAM 地址	串行口	情况 1 I/O 地址	情况 2 I/O 地址	情况 3 I/O 地址	情况 4 I/O 地址
40:0h	1	3F8h	3F8h	2F8h	0
40:2h	2	2F8h	0	0	0
40:4h	3	0	0	0	0
40:6h	4	0	0	0	0

这些测试结果并不指示 I/O 端口是否真正地带有一个串行口设备，测试只检查串行口是否存在于指定的 I/O 地址中。检测到的活动端口的总数保存在 BIOS RAM 地址 40:10h 处的设备字节的位 9~11 中。

串行口 BIOS

串行口 BIOS 提供了简单的功能来完成传送、接收和控制活动的串行口。BIOS 支持四个串行口，它们的 I/O 端口地址保存在 BIOS 数据区中。BIOS 数据区中的串行超时限定值用来防止在收发数据时无休止的等待，例如串行口所带的设备没有响应。如果发生了硬件错误或者连线没有接上，就可能发生这种串行口无响应错误。

BIOS 功能不使用串行口的中断特性，它使用的是软件计数循环。这就要求通过 BIOS 程序不断地查询 UART 状态。清除传送器的缓冲区后，就可以传送下一个字节。类似地，如果 UART 接收器的缓冲区满，则可以读取下一个字节。

大多数终端模拟程序会直接访问 UART，而完全跳过串行 BIOS 程序，这样支持软件可获得最大运行速率。此外，绕过 BIOS 可以执行中断控制以及获得 9600 以上的波特率。

串行 BIOS 的所有功能都会在 AH 中返回串行口的状态。表 12-3 描述了这些线的状态。

表 12-3 AH 中的 BIOS 线状态

位	7=1	无响应超时错误（位 0~6 无意义）
	6=1	传送保持寄存器和传送移位寄存器皆空
	5=1	传送缓冲区寄存器空，准备好接收下一个待传送字节
	4=1	收到停顿信号。当输入信号维持低电平超过一个完整的串行帧周期时就会发出停顿信号。这意味着起始位、数据位和奇偶校验位都是低电平，没有收到停止位（高电平）
	3=1	出现帧错误。接收数据时检测到的停止位是低电平，而实际上停止位应该是高电平。UART 将试图重新同步下一个接收到的帧。帧错误通常表示一个噪声或弱信号，或者表示波特率不匹配
	2=1	出现奇偶错误。接收数据的奇偶位和指定的奇偶模式不匹配
	1=1	出现了超限运行。在还没有从缓冲区中读走前一个收到的字节之前，接收到了第二个字节。新的字节覆盖了前面的字节，因此丢失了前面的字节。这通常指示系统对当前的波特率来说显得太慢，系统中的其他中断导致放慢了串行口响应处理，或者端口处理程序的软件有 bug
	0=1	数据准备好指示器。现在可以读取接收到的字节

除了发送字节和接收字节这两个功能之外，所有其他的功能都会在 AL 中返回调制解调器状态，如表 12-4 所示。位 0~3 列出了上次读取调制解调器状态寄存器后的状态改变情况。BIOS 读取 UART 寄存器时，会清除所有的位。

表 12-4 AL 中的 BIOS 调制解调器状态

位	7=x	连接器的数据载波检测线的状态
	6=x	连接器的响铃指示器线的状态
	5=x	连接器的数据准备好线的状态
	4=x	连接器的清除以便发送线的状态
	3=1	数据载波线发生了改变
	2=1	响铃检测线上出现了从高到低的下降沿信号
	1=1	数据设置准备好线发生了改变
	0=1	清除以便发送线上发生了改变

中断 14h 提供下列服务：

功能	描述	平台
ah=0	初始化串行口	所有
ah=1	发送字节	所有
ah=2	接收字节	所有
ah=3	读取状态	所有
ah=4	扩展初始化	MCA
ah=500h	扩展控制读	MCA
ah=501h	扩展控制写	MCA

中断	功能	描述	平台
14h	0	初始化串行口	所有

初始化指定串行口的速度、奇偶类型、停止位以及串行帧的字长度。

调用：ah=0

al=串行口寄存器的初始化值

位				
7=x	波特率选择			
6=x	位	7	6	5
5=x		0	0	0=110 波特
		0	0	1=150 波特
		0	1	0=300 波特
		0	1	1=600 波特
		1	0	0=1200 波特
		1	0	1=2400 波特
		1	1	0=4800 波特
		1	1	1=9600 波特
4=x	奇偶选择			
3=x	位	4	3	

		0	0=无奇偶校验
		0	1=奇奇偶校验
		1	0=无奇偶校验
		1	1=偶奇偶校验
2=0	使用一个停止位		
1	使用两个停止位（或者 1.5 位，如果选择了 5 位数据长度）		
1=x	数据长度		
0=x	位	1	0
		0	0=5 位
		0	1=6 位
		1	0=7 位
		1	1=8 位

dx = 串行口号 0~3（从零开始计数，即端口 1 使用 0，依此类推）

返回：ah = 线状态（参看表 12-2）

al = 调制解调器状态（参看 12-3）

中断	功能	描述	平台
14h	1	发送字节	所有

向指定的端口发送一个字节。在串行设备调制解调器线上，检查数据终端准备好和请求发送信号，以确保两条线都是高电平。在返回超时错误之前，BIOS 会等待 2ms 以便这些线变成高电平。另外，在传送一个字节之前，UART 的传送缓冲区必须为空。在返回一个超时错误之前，BIOS 也会等待 2ms 以便清空传送缓冲区。

2ms 的延迟通常是由原始的软件计时循环来完成，后者因 BIOS 的不同而有很大的变化。记住，由于某些串行设备不正确地断开了 DTR 和 RTS 线，所以使得 BIOS 传送功能不可靠。如果这些线“漂移”到了低电平，则可能不会传送。对此没有简单的修复方法，除非编写一个自定义程序，这个程序要带有一个选项来忽略 DTR 和 RTS。最好是向供应商提出这个问题，让他们来修理这些硬件故障。

调用：ah=1

al=要传送的字节

dx=串行口号 0~3（从零开始计数，即端口 1 使用 0，依此类推）

返回：ah=线状态（参看表 12-2）

al=没有改变（即仍是传送的字节）

中断	功能	描述	平台
14h	2	接收字节	所有

从指定的端口接收一个字节。检查数据终端准备好状态和数据设置准备好。如果都是

高电平，就从 UART 读取一个数据字节。BIOS 会等待 2ms 以便让每个信号都变成高电平。如果在超时期限内两个状态都没有变成高电平，则返回超时错误。

调用：　ah =2
　　dx = 串行口 0~3（从零开始计数，即端口 1 使用 0，依次类推）
返回：　ah=线状态（参看 12-2）
　　al=接收到的字节

中断	功能	描述	平台
14h	3	读状态	所有

获取指定端口的当前线和调制解调器的状态。

调用：　ah=3
　　dx=串行口 0~3（(从零开始计数，即端口 1 使用 0，依次类推）
返回：　ah=线状态（参看表 12-2）
　　al=调制解调器状态（参看表 12-3）

中断	功能	描述	平台
14h	4	扩展初始化	MCA

这个功能为 UART 提供了更多的初始化选项，这些选项是功能 0 所没有提供的。尽管每个 MCA 系统都支持扩展串行初始化，但是大多数系统并不支持这个功能。

调用：　ah=4

al=0	停顿控制关闭（普通情况）
1	将串行输出线强制为“0”，以传送一个停顿条件。这个条件在接收到 UART 线状态时会设置停顿标志。
bl=0	一个停止位
1	两个停止位（如果选择了五位数据长度，则停止位长度为 1.5 位）
bh=0	无奇偶校验位
1	奇奇偶校验（将所有的数据和奇偶校验位加起来必须是一个奇数）
2	偶奇偶校验（将所有的数据和奇偶校验位加起来必须是一个偶数）
3	粘性奇奇偶校验（发送的奇偶校验位为 0，接收时检查到的奇偶校验位也必须是 0）
4	粘性偶奇偶校验（发送的奇偶校验位为 1，接收时检查到的奇偶校验位也必须是 1）
cl=0	波特率为 110

	1	波特率为 150
	2	波特率为 300
	3	波特率为 600
	4	波特率为 1200
	5	波特率为 2400
	6	波特率为 4800
	7	波特率为 9600
	8	波特率为 19200
	9	波特率为 115200（PS/2 的 70 型号上没有公开这个选项，其他型号的机器可能也未公开这个选项）
ch=0		数据长度为 5 位
	1	数据长度为 6 位
	2	数据长度为 7 位
	3	数据长度为 8 位

dx=串行口 0~3（（从零开始计数，即端口 1 使用 0，依次类推）

返回：ah=线状态（参看表 12-2）

al=调制解调器状态（参看表 12-3）

中断	功能	描述	平台
14h	500h	扩展控制读	MCA

这个功能读取指定串行口的扩展调制解调器控制字节。所有的 MCA 系统及某些 AT+系统支持这个功能。

调用：ax=500h

dx=串行口 0~3（（从零开始计数，即端口 1 使用 0，依次类推）

返回：ah=线状态（参看表 12-2）

al=调制解调器状态（参看表 12-3）

bl=调制解调器控制寄存器

位	7=x	未使用
	6=x	未使用
	5=x	未使用
	4=1	回环（Loopback）模式处于活动状态（在后面的章节中将对这种模式做出解释）
	3=1	OUT2（IRQ 允许）
	2=x	OUT1（没有使用，也没有连接）
	1=1	请求发送（RTS）
	0=1	数据终端准备好（DTR）

中断	功能	描述	平台
14h	501h	扩展控制字	MCA

这个功能读取指定串行口的扩展调制解调器控制字节。所有的 MCA 系统及某些 AT+系统支持这个功能。

调用：　ax=501h

bl=调制解调器控制寄存器

位		
	7=x	未使用
	6=x	未使用
	5=x	未使用
	4=1	回环（Loopback）模式处于活动状态（在后面的章节中将对这种模式做出解释）
	3=1	OUT2（IRQ 允许）
	2=x	OUT1（没有使用，也没有连接）
	1=1	请求发送（RTS）
	0=1	数据终端准备好（DTR）

dx=串行口 0~3（从零开始计数，即端口 1 使用 0，依次类推）

返回：　ah=线状态（参看表 12-2）

al=调制解调器状态（参看表 12-3）

串行帧

串行帧由起始位、数据、奇偶位以及停止位组成。在串行连线上一次可以发送或接收一个帧。传送时，UART 在要传送的字节中加入起始位、可任选的奇偶位以及停止位。接收数据时，UART 检查帧中是否存在各种错误，并从帧中取出有用的数据。

在 UART 内部，由软件选择的各种条件决定着帧的大小。数据部分可能长 5~8 位，可以包括奇偶位，也可以不包括奇偶位，停止位可以选择 1~2 位。这意味着帧的总长度范围是 7~12 位。图 12-2 展示了一个典型的帧。在没有发送帧时，UART 的空闲状态是 1。

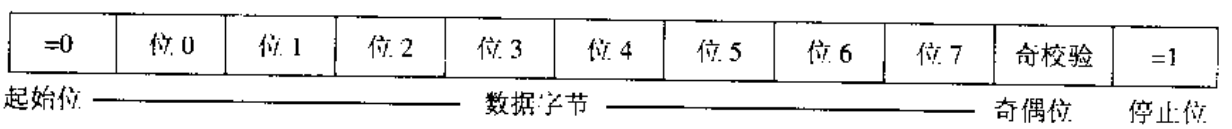


图 12-2　一个典型的串行帧

控制/调制解调器信号

在串行口和附带的设备之间使用了六条控制线和两条数据线。虽然没有必要使用所有

这些线，但是它们提供了一种与调制解调器及其他设备之间的清晰的通信方式。如果没有使用控制线，那么在连接器上的输入一般为低电平。表 12-5 和 12-6 描述了每一条线。

表 12-5 从设备到计算机的信号

信　号	简　写	描　述
清除以便发送	CTS	所附带的设备准备好，计算机可以发送数据
数据载波检测	DCD	所附带的设备（调制解调器）与另外一个调制解调器建立了连接
数据设置准备好	DSR	附带设备已经上电，准备好通信
响铃指示器	RI	连接到设备（调制解调器）的电话线正在响铃。它一般用于通信软件自动回复一个来电

表 12-6 从计算机到设备的信号

信　号	简　写	描　述
数据终端准备好	DTR	计算机发出一个信号指示所附带的设备，它已经上电，可以进行通信
请求发送	RTS	计算机发出一个信号指示它可以接收数据

RS-232C 调制解调器和控制线的低电平信号，或者说指示器的“0”状态，它的电平水准位于-3 到-15 伏之间，+3 到+15 伏之间的电平水准代表高电平信号，或指示器的“1”状态。

连接器会反转接收线和发送线，这一点不太容易理解。这意味着如果你传送了一个字节，比如将每一位设置为“1”，连接器上的电平将在-3~-15 伏之间，刚好和调制解调器和控制线上的电平逻辑相反。几乎每个人提到发送和接收逻辑时，指的都是没有反转的 UART 上的电平，而不是连接器上的电平。大多数串行口会生成-12V 和+12V 来作为逻辑电平。

事件顺序——串行传送

要通过串行线传送一个字节，必须用合适的波特率初始化串行端口，并确定串行帧的相关选项。下面的说明假定了字节在串行口 1 上传送。

1. 首先，从 BIOS 数据区 40:0h 处读取一个字，以此来确定串行 I/O 端口 1 的基地址。如果查到的值是零，则没有附带活动的串行口，因此不能发送数据。
2. 将两条调制解调器控制线，即数据终端准备好和请求发送线设置为高电平。数据终端准备好线通知设备，计算机已被激活，可以进行通信了。请求发送告诉附带设备，计算机想要发送数据。向调制解调器控制端口写入值 3 来激活这两条线。
3. 接着检查两条状态线：清除以便发送和数据设置准备好。它们是调制解调器状态寄存器的第 4 位和第 5 位。数据设置准备好指示所附带的设备已经上电并且准备好，清除以便发送指示所附带的设备可以接收数据。必须在 2ms 以内结束对这些状态线

的检查，或者在它们都变成高电平时结束检查。如果它们都变成了高电平，那么表示所附带的设备已经准备好接收一个字节。但是如果任意一条线维持低电平超过2ms，软件都将会发出一个超时错误信号。

4. 现在所附带的设备可以接收一个字节了。这时，必须检查 UART 以确认传送保持寄存器是否准备好。如果这个寄存器为空，并且已经准备好传送一个字节，则线状态寄存器的第 5 位会被设置为高电平。与第 3 步相似，如果在 2ms 内传送保持寄存器仍不可用，那么软件会声明一个超时性错误，并中止发送。
5. 如果没有发生超时性错误，那么可以向 UART 的传送保持寄存器中发送数据。
6. 然后 UART 将数据从传送保持寄存器发送到传送移位寄存器中，并通过串行连线发送串行帧。

事件顺序——串行接收

要从串行连线接收一个字节，必须以合适的波特率和串行帧选项来初始化串行端口。在下面的阐述中我假定从串行口 3 接收字节。

1. 首先，从 BIOS 数据区 40:4h 处读取一个字，以此来确定串行 I/O 端口 3 的基地址。如果查到的值是零，则没有附带活动的串行口，因此不能发送数据。
2. 将一条调制解调器控制线，即数据终端准备好线设置为高电平。它通知设备，计算机已被激活，可以进行通信了。向调制解调器控制端口写入值 1 来激活这两条线。
3. 接着检查数据设置准备好线。它是调制解调器状态寄存器的第 5 位。数据设置准备好指示所附带的设备已经上电并且准备好。必须在 2ms 超时期限以内结束对这些状态线的检查，或者在它们都变成高电平时结束检查。
4. 然后接收缓冲区查看是否接收到了数据。线状态寄存器的第 0 位就是一个数据准备好标志，在接收缓冲区内有数据时它被设置为 1。如果在 2ms 内没有设置这个数据准备好标志，则发送一个超时性错误，并中止操作。
5. 如果到目前为止还没有发生超时性错误，那么可以从 UART 的接收缓冲区读取字节。

回环（Loopback）操作

UART 提供了一种特殊的回环模式来测试 UART 的内部传输、接收和发送电路。BIOS 功能 5（控制控制）可以用来激活回环模式，或者设置调制解调器控制寄存器的第 4 位直接激活这个模式。

在回环操作期间，许多连接被强制为指定的状态，和/或断开与 UART 外部芯片的连接。它支持在诊断测试 UART 的同时不用影响串行口连接器所附带的设备。在回环模式下，UART 将每个输出设置为高电平，包括输出、-DTR、-RTS、-OUT1 和-OUT2。串行口适配卡的逻

辑处理电路将反转这些信号，所以与 UART 上的各自高电平输出相对应的连接器引脚都变成了低电平。传送低电平表示串行连线处于空闲状态。将-OUT1 设置为高电平，即断开了系统和中断请求之间的联系。

在回环操作期间，输入线 CTS、DSR、DCD 和 RI 也从连接器上断开。作为测试的一部分，UART 将这些线从内部和四个调制解调器控制信号连接起来，这四个信号是 DTR、RTS、OUT1 和 OUT2。这时可以检查一部分内部 UART 控制和状态线的粘连和交叉情况。这意味着向调制解调器控制器的低四位写入的数据可以直接从这个寄存器的高四位读取。

在回环操作期间，还将接收输入线从连接器上断开，而在 UART 内部将它直接和内部传输输出连接起来，于是可以立即收到传送的串行数据，这时接收和传送完全是 UART 的内部操作。虽然中断开放寄存器仍然控制着中断操作，但是正如我在前面所指出的那样，在回环期间，串行口硬件已经将中断请求线从系统上断开，所以测试不会产生真正的系统中断。

波特率

波特率是指串行连线上每秒所传输的位数。只要将波特率除以每个串行帧的位数，就可以得到每秒传输的字节数。大多数常见的帧包括 8 位数据，没有奇偶位，但是带有一个停止位和一个起始位，因此加起来总共 10 位。这意味着 9600 的波特率每秒可以传送 960 个字节。用下面这个例子说明这个数据会更形象：以 9600 波特率的速度填充一个 2,000 字符的标准文本屏幕大约需要两秒钟。

UART 有一个内置的波特率发生器。波特率应该设置成与所附带设备相匹配的大小。波特率除法因子的大小与晶振频率有关。选择晶体时应保证编程因子值在任意系统上都产生相同的波特率。用任意期望的波特率除以 115,200，来确定这个期望波特率的除法因子。

对于系统到系统的传送来说，波特率可以设置为很高的值，通常这个值受限于较慢系统的 CPU 速度。在最大的速率（115,200 波特）下，CPU 可以在每 87 微秒内读取并保存一个字节。

表 12-7 列出了常见的波特率、UART 除法因子以及标准 10 位帧时每秒传送的字节数。

表 12-7 波特率

波特率	除法因子	每秒字节数	典型应用
50	2340（900h）	5	古老的电传机
75	1536（600h）	7.5	古老的电传机
110	1047（417h）	11	很老式的电传机
300	384（180h）	30	老式的电传机，过时的终端
1200	96（ 60h）	120	老式的终端、过时的调制解调器
2400	48（ 30h）	240	过时的终端，调制解调器、打印机
4800	24（ 18h）	480	过时的终端，调制解调器、打印机

续表

波特率	除法因子	每秒字节数	典型应用
9600	12（0Ch）	960	终端，调制解调器、打印机、传真机
14400	8	1440	高速调制解调器
19200	6	1920	高速调制解调器、数据传送器
28800	4	2880	高速调制解调器、数据传送器
38400	3	3840	高速调制解调器、数据传送器
57600	2*	5760	数据传送器、ISDN 调制解调器
115200	1**	11520	数据传送器、ISDN 调制解调器

* 没有一个除法因子能够提供严格的 56000 波特率。这时，由除法因子设置的波特率实际上为 57600，比 56000 快 2.86%。这个问题并不严重，因为系统实际上能够容忍 5%的波特率偏差。

** 早期的 8250 UART 不支持这个波特率

中断控制

可以将串行口设置为中断操作模式（而不是查询模式）。无论什么时候出现错误、接收到一个字节、UART 准备好传送一个字节或者调制解调器控制线的状态发生了改变，都可以激发一个串行口中断服务程序来获得控制。可以通过写中断开放寄存器，有选择性地开放和禁止上述四种类型中断。

这四种类型的中断有任意一个发生时，适配器都会触发一个中断请求，通常是 IRQ3 或者 4。使用串行口中断控制的应用程序必须提供中断处理程序。BIOS 没有为串行口提供任何中断处理服务。

一旦出现了一个中断，处理程序必须读取串行口的中断标识寄存器来查看中断的起因。表 12-8 列出了读取串行口中断标识寄存器的可能结果。如果在读取中断标识寄存器之前发生了多个中断，那么首先提供最高优先级的功能。

表 12-8　中断类型与操作

中断标识寄存器位 3 2 1 0	优先级	类　型	中断源	清除中断的操作
0 0 0 0	最低	调制解调器状态	CTS、DSR、RI 或 DCD 被置为高电平	读取调制解调器状态寄存器
0 0 0 1	无	无活动的中断		
0 0 1 0	第三低	传送保持寄存器空	寄存器空	读中断标识寄存器和写传送保持寄存器
0 1 0 0	第二低	收到的数据可用	收到数据或触发 FIFO	读接收到的数据或者 FIFO 入口的总体水平降到触发水平以下

续表

中断标识寄存器位 3210	优先级	类 型	中断源	清除中断的操作
0110	最高	接收器的线状态	停顿中断、帧错误、过载超时错误、或奇偶错误	读取线状态寄存器
1*100	第二低	字符超时指示器	在接收 FIFO 中至少有一个字符，并且在四个字符时间内没有从或向 FIFO 中移动一个字节	读取接收缓冲区寄存器

* 仅在带有 FIFO 的 UART 上才定义了第 3 位，例如 16650，在老式类型的 UART 上第 3 位总是为零

系统为四个串行口仅保留了两个 IRQ。IRQ4 通常和串行口 1 和 3 相连，IRQ3 通常和串行口 2 和 4 相连，但是 IRQ3 更多的情况是用于另外的一些卡上，例如网络适配器。中断请求线 3 和 4 符合工业标准。如果串行口允许，还可以使用其他的串行口，但是在大多数情况下并不支持这样做。

在 MCA 系统上，两个或两个以上的串行口可以共享一个中断。要实现这一点，必须更改中断处理程序，以便使多个程序可以共享同一个中断。使用串行口 1 的应用程序的串行口处理程序为串行口 1 查看标识寄存器，中断标识寄存器指示是否激活了串行口 1 的中断。如果没有激活这个中断，那么由与这个中断相连的其他串行口来处理这个中断，而串行口 1 的处理程序不会采取进一步的行动。

由于更多的端口共享和使用了相同的中断，挂起中断的老式程序的响应时间会大大变长，这时可能要求降低波特率，以避免出现过载超时的数据丢失。为了保证较高的波特率，某些应用程序会改变中断控制器的优先级排列，以使串行口中断的优先级最高。这意味着串行口处理程序可以在任何其他的中断，包括系统时钟之前服务串行口。在第 17 章“中断控制和 NMI”中将更详细地解释中断优先级。

FIFO 模式

16550 系列的 UART 的操作和普通的 8250/16540 UART 一样，但是前者还提供了一种可任选的“FIFO 模式”。大多数老式的系统没有使用 16550，所以也就无法得益于 FIFO 模式操作。在某些情况下，8250/16450 的 UART 芯片插在一个插槽中，可以用 16550 替换掉。

一些系统使用另外一种 UART，82510。这种 Intel 芯片兼容 8250，也提供了 FIFO 选项。它并不像 16550 那样常见，但是的确存在于某些系统中。85510 的编程远比其他的 UART 复杂，它提供了许多其他的寄存器和模式。本章主要关心的是常见的 16550 FIFO UART。BIOS 中断服务程序不会激活 FIFO 模式。如果要求快速执行，则不要使用较慢的 BIOS。

激活 FIFO 模式时，UART 将缓冲 16 字节的接收数据和 UART 内 16 字节的发送数据。

这种额外的缓冲处理允许 CPU 有足够的时间去处理更高优先级的中断和其他功能。如果没有缓冲，必须在下一个数据到达之前读走前面接收到的数据。如果 CPU 在一个扩展周期内因为要处理更高优先级的任务而保持忙状态，同时又收到了两个或两个以上的字节，就会出现过载超时。过载超时错误表示丢失了字节，这种丢失的起因是 CPU 在下面的字节到来之前没有从 UART 中读走数据。

本章后面的串行口检测器样本代码展示了如何检测是否可以使用 FIFO 模式。一旦确定了支持 FIFO 模式，则用一组读写程序就可以替代非 FIFO 模式的发送和接收程序。这些替代程序可以对 UART 一次传送多个字节。

与普通模式的 UART 操作一样，可以将 FIFO 模式设置为中断或者查询操作形式。在 FIFO 模式查询操作下，收发程序检查线状态寄存器所使用的方法和非 FIFO 模式的查询一样。如果在接收 FIFO 中有一个或多个字节，那么会设置线状态寄存器的第 0 位。设置第 5 位则表示在传送 FIFO 中有一个或多个字节。软件可以透明地使用这个两个字节。这一点允许仍在使用 BIOS 串行口处理程序时激活 FIFO 模式。

至于 FIFO 中断操作，可以在 FIFO 控制寄存器中设置触发水平。触发水平的定义是，为了生成一个接收数据可用中断，在接收 FIFO 中所需要有的字节数。如果接收 FIFO 设置为 1、4、8 或 14 个字节，则可以对 UART 编程，以中断 CPU。为了将中断率维持在一个较低的水平上，应使用一个较高的触发水平。如果 CPU 对中断的响应不够快，则可以使用较低的触发水平。如果允许更高优先级的中断，或者 CPU 时钟速度或 CPU 类型而使系统速度很慢，那么可能会出现比较慢的 CPU 响应。

在 FIFO 中断模式下，如果发生了所有下述的三种情况，UART 将会触发一个接收超时中断：

- 接收 FIFO 有一个或一个以上的字节
- 在超时期限内连线上一个字节也没有接收到
- 在超时期限内 CPU 没有从 FIFO 读取任何字节

超时期限等于没有任何其他动作发生时接收 4 帧的时间。这意味着超时期限直接与波特率成比例。例如 10 位帧、9600 波特，其超时期限应该是 4.2ms。至于其他的帧和波特率，超时期限可以用下式计算：

$$\text{超时期限（单位 ms）} = \frac{\text{每帧中的位数}}{\text{波特率}} \times 4\text{ 帧} \times 1000 \quad \text{ms}$$

在收到一个新的字节，或者 CPU 从 FIFO 读取了一个字节时，都会重新设置超时期限计数器。如果发生了一个超时中断，当 CPU 从接收 FIFO 读取字节时会重新设置这个计数器。

在中断模式下，当传送 FIFO 变为空时，也会生成一个中断。接着，中断服务程序可以向传送 FIFO 传送 16 个字节。

BIOS 数据区

地址	描述	大小
40:00h	串行口1	字

这个字用来保存串行口1的第一个I/O端口号。一般串行口1位于I/O地址3F8h或2F8h，但是可以在这个字中保存任意串行口的I/O端口地址。

地址	描述	大小
40:02h	串行口2	字

这个字用来保存串行口2的第一个I/O端口号。一般串行口2位于I/O地址2F8h，但是可以在这个字中保存任意串行口的I/O端口地址。

地址	描述	大小
40:04h	串行口3	字

这个字用来保存串行口3的第一个I/O端口号。可以在这个字中保存任意串行口的I/O端口地址。

地址	描述	大小
40:06h	串行口4	字

这个字用来保存串行口4的第一个I/O端口号。可以在这个字中保存任意串行口的I/O端口地址。

地址	描述	大小
40:7Ch	串行口1超时	字节

这个字节保存了串行口中断14h处理程序用到的一个固定值。在串行口处理程序发送一个字节时，这个超时值用来等待各种条件的改变以便能够发送或接收一个字节。

在传送一个字节时，检查数据终端准备好（DTR）和请求发送（RTS）线。如果在超时期限内它们都没有变成高电平，那么将会出现超时错误，BIOS将返回一个超时错误条件。

接收一个字符的操作和上述情形类似。首先，使用一个超时期限值，来等待激活数据设置准备好线。然后再次使用这个超时值来等待直到控制器的状态指示接收缓冲区有一个字符。如果在超时期限内没有发生上述任何一种情况，那么BIOS将会返回一个超时错误条件。

大多数系统的POST会将这个值设置为1。每个计数单位是不小于2ms的延迟。由于超时是基于指令超时的，所以这个时间会随着系统的时钟速度和BIOS执行速度而改变。

早期的 PC 不支持超时字节，但是 XT 以及后来的平台都支持它。

地址	描述	大小
40:7Dh	串行口 2 超时	字节

参看 RAM 地址 40:7Ch 处的串行口 1 超时，以了解详情。

地址	描述	大小
40:7Eh	串行口 3 超时	字节

参看 RAM 地址 40:7Ch 处的串行口 1 超时，以了解详情。

地址	描述	大小
40:7Fh	串行口 4 超时	字节

参看 RAM 地址 40:7Ch 处的串行口 1 超时，以了解详情。

调　试

调试串行口程序有点复杂，难就难在不得不同时操作两个独立的系统。如果某个系统的程序有代码错误，那么很难确定是哪个系统生成了这个问题，也难以生成所有可能的测试条件来确保程序编写正确。

串行口分析器有助于迅速定位问题的起源，但是它的价格比较昂贵。某些分析器能为测试操作生成错误条件。

现在的串行口分析软件也可以可靠地运行在一个独立的 PC 上。这些软件可以用来替代专用的硬件分析产品。尽管要求运行于一台独立的 PC，可是这种软件的价格往往不足 $300。生产这种软件产品的一个公司是美国德克萨斯州霍斯顿的 WCSC，他们的联系电话是 713-498-4832。一个更廉价（低于$50）的可选方案是使用串行口测试器。一个带有 LED 的串行口测试器，可以用来指示 8 条信号线的状态：高电平、低电平或者断开。使用调试程序单步运行代码，同时监测信号的状态，可能有助于识别问题的起因。

我们为调试串行口提供了第三种免费的可选方案：使用程序 SSPY。SSPY 会连续监视四条调制解调器输入线，它们分别是 DCD、RI、DSR 和 CTS，同时它还监视两条输出线 DTR 和 RTS。监视的结果都显示在屏幕上。SSPY 是一个 TSR，它每秒 18 次读取指定串行口的 UART 状态，而不改变这些状态。

SSPY 不能检测有缺陷的硬件、导线短路或开路，但是它有助于隔离串行程序不正确的串行口操作。对要检查的端口 1~4 运行 SSPY。一旦装入 SSPY 后，可以使用 Scroll Lock 来激活它。打开 Scroll Lock 后，所有的状态信息都显示在屏幕的顶部。关闭 Scroll Lock 会禁止 SSPY。在某些系统运行高速数据传送时，SSPY 的轻微系统开销都可能造

成问题。如果你想对 SSPY 做一些修改，可以在 http://www.infopower.com.cn 处找到完整的源代码。

在许多情况下，低波特率传送有助于阐明两个系统是如何通信的。使用 BIOS 的初始化功能，你可以选择设置 75 波特的低波特率。这个波特率可能仍然嫌快，特别是当你使用一个简单的端口测试程序时更是如此。可以调用代码 12-1 中的子程序来将波特率放慢到每秒只传送 2 位，使用这个速度足以看到数据帧中的每一位。

UART 类型归纳

这些年对早期的 8259 UART 做了许多改进，下面的列表包括了大部分用在 PC 平台上的 8250 系列。记住，不同的供应商提供这些类型时往往带有前缀或后缀，这些前缀或后缀是供应商所特有的。

基本编号	描述
8250	基本 UART，执行时有一些限制
16450	8250 的改进
16550	在 16450 中加入了接收和发送 FIFO
16552	一个部件中有两个 16550 UART
16C454	一个部件中有四个 16450 UART
16C554	一个部件中有四个 16550 UART
16C1450	改进的 16450，带有程序可控的断电和重启
16C1550	改进的 16550，带有程序可控的断电和重启
82510	在 16450 中加入了接收和发送 FIFO 以及断电模式（FIFO 的操作与 1655 不同）

串行连接器

尽管许多程序员从未直接处理过硬件问题，但是他们也常常需要监视连接器处的信号线和其他信息。串行口使用 25 针或 9 针的 D 型连接器。表 12-9 列出了 25 针连接器的情况，表 12-10 列出了 9 针连接器的情况。输入输出所针对的方向是相对于系统而言的。这就意味着输出指的是向所带的设备发送一个信号。在“串行 BIOS”一节中对这个功能做出了更加详细的阐述。带减号前缀的功能在低电平时被激活，带加号前缀的功能在高电平时被激活。

某些串行适配卡支持老式电传机的电流环模式。9 针连接器没有引脚用来支持这种模式。目前大多数的串行口卡不再支持电流环模式。

表 12-9 串行口 25 针连接器

引 脚	方 向	功 能
2	输出	串行发送线
3	输入	串行接收线
4	输出	请求发送（RTS）线
5	输入	清除以便发送（CTS）线
6	输入	数据设置准备好（DSR）线
7	地	
8	输入	接收线信号检测器
9	输出	+电传机电流环数据
11	输出	-电传机电流环数据
18	输入	+接收电流环数据
20	输出	数据终端准备好（DTR）
22	输入	响铃指示器
25	输入	-接收电流环数据

没有列出的引脚表示没有使用或者没有连接

表 12-10 串行口 9 针连接器

引 脚	方 向	功 能
1	输入	接收线信号检测器
2	输出	串行发送线
3	输入	串行接收线
4	输出	数据终端准备好（DTR）
5	地	
6	输入	数据设置准备好（DSR）线
7	输出	请求发送（RTS）线
8	输入	清除以便发送（CTS）线
9	输入	响铃指示器

警 告

确保在 I/O 访问相同的 UART 之间有足够的延迟，特别是当系统的总线速度很快或者 UART 处理 I/O 的速度非常慢时更应该如此。在使用串行口之前检查一下系统正在使用哪一种 UART 不失为明智之举。一旦知道了芯片类型，就很容易避免一些芯片上的许多陷阱。代码例 12-2 展示了如何可靠地检测 UART 类型。

8250 bug 早期的 8250 UART 有许多 bug 和限制，这些 bug 和限制常常使程序员感到沮丧。在新式的 UART 芯片中对这些问题都加以了改正，这些新式的 UART 包括 16450 和 16550 等。你可以使用系统的 BIOS 串行口功能或者清楚明白地理解 8250 的这些局限性来避免这些问题。

这些问题包括：

- 无效的传送保持寄存器中断
- 开放中断位丢失
- 有限的波特率
- 允许 8 位数据长度和奇偶性问题

首先，在中断允许寄存器中设置开放传送保持寄存器中断功能时，会生成一个中断，即使这个时候传送保持寄存器是空的。如果打算使用这个中断功能，那么应该只在传送寄存器为空时才可以开放中断。作为一种可以替代的方法，中断处理程序可能会检查传送缓冲区是否真的为空，但是这样做可能会降低最大运行速度。

当芯片上电之后，UART 可能会错过第一次设置开放传送保持寄存器中断位的机会。为了解决这个 bug，在一行中简单地两次写中断开放寄存器。如果每次写这个寄存器都没有任何负效应，那么这个写操作才是可以接收的。例如代码可以如下所示：

```
mov     dx,  3F9h       ; 中断开放寄存器
mov     al,  2          ; 开放传送空位
out     dx,  al         ; 执行这个操作
jmp     short $+2       ; 延迟
out     dx,  al         ; 再次执行这个操作
```

尽管所有的 UART 都可以使用达 115,200 的波特率，但是实际上制造商已经将 8250、8250B 和 8250C 的最大波特率限制为 56,000。8250A 不受这个限制约束。在我的测试过程中，只要不使用中断，并提供足够的系统实时时钟吞吐能力，那么 56,000 的波特率算不上一个问题。

大多数情况下，有必要使用中断以便在当前的环境下能够正常工作。这样做可能会降低有效的波特率，降低到大约 9600 波特率。在清除了每一个中断时，如果 UART 接收器采用一些简陋的定时方式，那么就可能出现这种情况。与 8250 以后的其他芯片相比，16450 UART 接收器的定时要快 25 倍！8250 传送器的计时要比接收器计时快一些，带中断控制正常工作时的波特率可达 38,400 波特率。

据我交谈所知，几位受人尊敬的工程师也发现早期的 8250 UART 在使用 8 位数据和奇偶校验时会出问题。但是使用 7 位数据加奇偶校验位，或者 8 位数据位而没有奇偶校验位总是可以正常工作。

对老式的 8250 芯片进行测试时，我的测试程序没有发现任何问题。即使对于那些具有

最大帧大小的传送情况（8 位数据、一个奇偶位以及两个停止位），传送和发送功能都可以正常地工作。我所进行的测试包括一项测试，即在 2400 到 56,000 波特率的范围内，测试没有中断时 8250 可以使用的最大波特率。由于我不可能测试所有供应商提供的芯片的每种功能组合以及错误条件，因此，这可能才真的算得上一个问题。另一方面，由于我还没有发现某个人实际上遇到过这种问题，这种缺陷可能只是一个神话而没有任何实际的意义，或者它会与我已经讨论过的问题混淆起来。

16550A 坏的 FIFO　16550A UART 能够和 16450 百分之百地兼容，但是由于设计存在缺陷，这种特殊版本的 16550 的 FIFO 并不能正确工作。所有以后的系列，包括那些带有后缀 AF、C 以及 CF 的 16550，都对这个问题加以了更正。幸运的是，很容易检测出芯片是否带有坏的 FIFO，如代码 12-2 中所示。

16550 FIFO 不能传送字节　在所有类型的 16550 上使用 FIFO 特性时，都会发生一种异常情况，这种异常情况可能会阻碍字节的传送！如果字节已经传送到了移位寄存器之外并且 FIFO 当前为空，就会出现这种情况。如果向传送 FIFO 中装入了一个字节，那么 UART 必须在前一个字节移走之后马上将这个字节装入到移位寄存器中。在这种情况下，新字节可能会没有传送到移位寄存器，而只是保留在 FIFO 中。

如果向传送 FIFO 中装入了一个字节，UART 会继续执行普通的操作，而前面的字节会正确地装入到移位寄存器并发送出去。

没有什么有效的方法能更正传送的错误字节，除非一开始就防止这种情况的发生。一种方法是在要发送前面一个字节时检查 FIFO 是否为空。如果为空，则软件必须等待一个传送帧来保证向 FIFO 中装入下一个字节时移位寄存器为空。

代码例 12-1　使用较慢的波特率进行调试

可以调用下面的程序将波特率强制为一个很低的值。这一点对于调试串行代码非常有用。在这个程序中，我将波特率设置为 2 波特（每秒 2 位），使用的除法因子是 E100h。必须用 DX 将 I/O 端口的基地址传送给程序。

```
;--------------------------------------------------------------------
;       SLOW_BAUD
;           设置波特率为 2，而不改变任何其他设置
;
;           调用:           dx = 串行口的起始 I/O 地址
;
;           返回:           设置后的新的波特率
;
;           用到的寄存器:   无
```

```
divisor equ         OE100h              ; 波特率为 2 的除法因子

slow_baud proc far
        push    ax
        push    dx
        add     dx,3                    ; 线控制寄存器
        in      al,dx
        or      al,80h                  ; 将 DLAB 设置 1 来装入波特率
        IODELAY
        out     dx,al
        sub     dx,3                    ; 除法因子锁存 LSB
        mov     al,divisor
        out     dx,al                   ; 设置除法因子的 LSB 为 0
        inc     dx                      ; 除法因子锁存 MSB
        mov     al,ah
        out     dx,al                   ; 设置除法因子 MSB 为 E10h
        add     dx,2                    ; 线控制寄存器
        in      al,dx
        and     al,7Fh                  ; 设置 DLAB=0，设置了波特率
        IODELAY
        out     dx,al
        pop     dx
        pop     ax
        ret
slow_baud endp
```

代码例 12-2 串行口检测器

这个程序检查所有活动的串行口，并显示其端口号，然后确定它是哪种类型的 UART。如果在这个端口用作它用（例如鼠标）时运行这个工具，那么可能就会中止端口操作。SERTYPE 总是返回精确的信息。

运行 SERTYPE 时，会在屏幕上显示串行口分析的结果。某个系统的 SERTYPE 显示结果可能如下所示：

SERTYPE——串行口分析

串行	口	活动?	基本 UART 类型
#1	03F8	是	8250A 或 16450
#2	02F8	是	16550AF/C/CF 带 FIFO
#3	02E8	是	82510 带 FIFO
#4		非	

SERTYPE 所显示的是端口号（1~4）及相关的 I/O 端口地址。检查 UART 可以查看系统是否包含有它。如果 UART 能够在指定的 I/O 端口对波特率寄存器的读和写做出正确的响应，那么活动状态就设置为“是”。如果检测到了活动的 UART，程序接着就会检测 UART 的类型。

在这本书中，我只列出了 SERTYPE 中比较有趣的部分，即 TESTPORT 子程序。TESTPORT 用来检测指定的 I/O 端口是否存在一个活动的 UART，并确定其类型。在 http://www.infopower.com.cn 处提供了完整的 SERTYPE 源代码。

```
;--------------------------------------------------------------------
;    测试端口
;       检查UART是否响应，如果响应，
;       则找出其等价芯片。
;
;       调用:          dx = 串行I/O端口
;
;       返回:          bl = 芯片类型
;                           0 = 无响应，不是UART
;                           1 = 8250（无SCRATCH寄存器）
;                           2 = 8250A或16450
;                           3 = 16C1450
;                           4 = 16550 带缺陷FIFO
;                           5 = 16550AF/C/CF带FIFO
;                           6 = 16C1550带FIFO
;                           7 = 16552双UART带FIFO
;                           8 = 82510带FIFO

testport proc      near
         mov       bl, 0                ; 如果无响应返回的值
         mov       di, dx               ; 保存I/O端口
```

```
        cli                             ; 禁止中断
        call    DLAB1                   ; 除法因子锁存访问位1
        mov     ah, 5Ah                 ; 测试值1
        call    IOtest                  ; 写值，读取并比较
        jne     test_exit2              ; 如果无效，则退出
        mov     ah, 0A5h                ; 测试值2
        call    IOtest                  ; 写值，读取并测试
        jne     test_exit2              ; 如果无效，则退出
        sti                             ; 开放中断

; 检查SCRATCH PAD寄存器是否工作（8250上无）

        inc     bl                      ; 如果是8250，返回1
        add     dx, 7                   ; 指向SCRATCH PAD寄存器
        mov     ah, 5Ah                 ; 测试值1
        call    IOtest                  ; 写值读取并比较
        jne     test_exit2              ; 如果无SCRATCH PAD寄存器则退出
        mov     ah, 0A5h                ; 测试值2
        call    IOtest                  ; 写值读取并比较
        je      scratch_ok
test_exit2:
        jmp     test_exit               ; 如果无SCRATCH寄存器则退出

scratch_ok:
        mov     dx, di                  ; 恢复起始I/O端口

; 检查是否为1655x和82510FIFO

        cli                             ; 禁止中断
        add     dx, 2                   ; 中断ID寄存器
        in      al, dx
        IODELAY
        and     al, 0C0h
        cmp     al, 0C0h                ; 16550FIFO已经开放?
        je      is_1655x                ; 如果是则跳转
```

```
        mov     al, 1                   ; 试着开放1655x上的FIFO
        out     dx, al                  ;   （如果是82510，也设置第0排）
        IODELAY
        in      al, dx                  ; 读中断标识
        IODELAY
        mov     bh, al                  ; 保存读回值
        mov     al, 0                   ; 禁止16550上的FIFO或者选择
        out     dx, al                  ;   82510的第0排
        IODELAY
        and     bh, 0C0h
        cmp     bh, 0C0h                ; 16550开放了FIFO吗?
        je      is_1655x                ; 如果是，则跳转
        cmp     bh, 40h                 ; 是某些PS/2上的FIFO UART?
        je      is_16550C
        cmp     bh, 80h                 ; 16550带缺陷FIFO?
        je      is_16550

; 不是16550。通过切换到第3排来检查是否为82510
;   （只有82510上才有第3排）

        mov     al, 60h                 ; 选择第3排
        out     dx, al
        IODELAY
        in      al, dx                  ; 从端口读回数据
        IODELAY
        mov     bh, al                  ; 保存，供后面使用

        mov     al, 0                   ; 如果是83510，则返回到
        out     dx, al                  ;   第0排
        IODELAY
        and     bh, 60h                 ; 到第3排吗?
        cmp     bh, 60h
        je      is_82510                ; 如果是，则跳转（是82510）

; 无FIFO，所有UART是8250A或16450系列——检查UART上是否
;   有电源控制功能，该功能位于调试解调器控制寄存器中
```

```
no_FIFO:
        mov     dx, di
        add     dx, 4                   ; 选中调制解调控制寄存器
        mov     al, 80h                 ; 检查是否支持低功率模式
        out     dx, al
        IODELAY
        in      al, dx                  ; 获取调制解调寄存器
        IODELAY
        mov     ah, al                  ; 保存结果
        mov     al, 0
        out     dx, al                  ; 重新开启电源
        test    ah, 80h                 ; 低功率位设置了吗?
        jnz     is_16C1450              ; 若是，则跳转

is_8250A:
        mov     bl, 2                   ; 设置为8250A/16450 UART
        jmp     test_exit

is_16C1450:
        mov     bl, 3                   ; 设置为16C1450 UART
        jmp     test_exit

is_16550:
        mov     bl, 4                   ; 设置为16550带缺陷FIFO
        jmp     test_exit

; 检测到了FIFO，一定是16550系列

is_1655x:
        mov     dx, di
        call    DLAB1
        add     dx, 2
        mov     ah, 7
        call    IOtest                  ; 写值并读取
        mov     bh, al                  ; 保存结果供后面使用
```

```
        mov     al, 0                   ; 重启寄存器选择
        out     dx, al
        cmp     bh, 7                   ; 起作用吗?
        je      is_16552                ; 如果相等，则是16552

; 检查UART上的低功率位功能是否位于调制解调
;   控制寄存器中（仅针对16C1550)

        mov     dx, di
        add     dx, 4                   ; 选择调制解调控制寄存器
        mov     al, 80h                 ; 看是否支持低功率模式
        out     dx, al
        IODELAY
        in      al, dx                  ; 获取调制解调寄存器
        IODELAY
        mov     ah, al                  ; 保存结果
        mov     al, 0
        out     dx, al                  ; 重新开启电源
        test    ah, 80h                 ; 设置低功率位了吗?
        jnz     is_16C1550              ; 如果是，则跳转

is_16550C:
        mov     bl, 5                   ; 设置为16550AF/C/CF UART
        jmp     test_exit

is_16C1550:
        mov     bl, 6                   ; 设置为16C1550
        jmp     test_exit

is_16552:                               ; 设置为16552 UART
        mov     bl, 7
        jmp     test_exit

is_82510:
        mov     bl, 8                   ; 设置为82510 UART
```

```
test_exit:
        mov     dx, di                  ; 恢复起始端口值
        call    DLAB0                   ; 重设 DLAB为0
        sti                             ; 如果中断关闭，则开放之
        ret
testport endp


; ---------------------------------------------------------------------------
;   DLAB1
;       将除法因子锁存访问位设置为1而不
;       改变线控制的其他位
;
;       调用:     dx = 串行I/O端口
;
;       用到的寄存器:   al

DLAB1   proc    near
        push    dx
        add     dx, 3                   ; 指向线控制寄存器
        in      al, dx                  ; 获取当前状态
        IODELAY
        or      al, 80h                 ; 设置DLAB为1
        out     dx, al
        pop     dx
        ret
DLAB1   endp


; ---------------------------------------------------------------------------
;   DLAB0
;       设置除法因子访问位为0而不改变其
;       他的线控制位
;
;       调用:           dx = 串行I/O口
;
```

```
;       所使用的寄存器:     al

DLAB0   proc    near
        push    dx
        add     dx, 3                   ; 指向线控制寄存器
        in      al, dx                  ; 获取当前状态
        IODELAY
        and     al, 7Fh                 ; 设置DLAB为0
        out     dx, al
        pop     dx
        ret
DLAB0   endp


; ---------------------------------------------------------------------------
;   IOtest
;       将al的值写到端口，然后读回
;       该值并和初始值比较
;
;       调用:           ah = 用来测试的值
;                       dx = 测试的I/O端口
;
;       返回:            al = 读自端口的值
;                  零标志位 = 1 如果寄存器正常

IOtest  proc    near
        mov     al, ah                  ; 要测试的值
        out     dx, al                  ; 写值
        IODELAY
        in      al, dx                  ; 读回值
        IODELAY
        cmp     al, ah                  ; 比较是否相等
        ret
IOtest  endp
```

代码例 12-3 最大波特率分析器

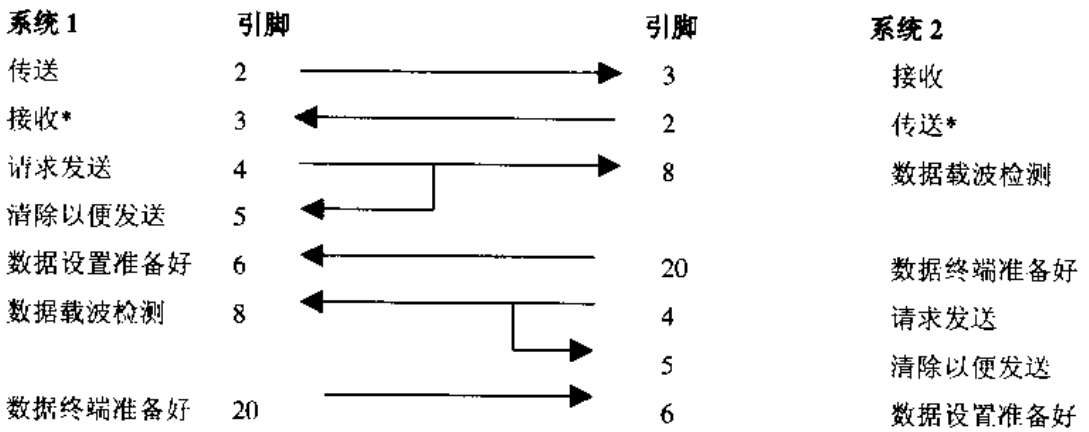
下面这两个程序要同时使用，以检测两台计算机之间最大的传送速率。使用这两个程序会发送 512 个字节，所采用的方法是逐步提高两个系统之间的波特率，直到检测到传送错误为止。

http://www.infopower.com..cn 处提供了这两个程序 MAX_XMIT 和 MAX_RECV。某些代码看起来比较古怪，编写它们是为了最大化运行速度。在这些编程方式比较古怪的地方，源代码做了详细的注释说明。

为了提高运行的速度，我们只使用了 RAM 进行数据传送。要传送的数据来自于 RAM，系统接收到数据后也把它们放在 RAM 中。实际的通信包可能会从/向软盘发送数据，这样会极大地降低可靠的传送速率。由于即使是最慢的系统也能够处理 2400 的波特率，因此这两个程序就选择使用这个初始波特率。

使用这两个程序时，一个系统运行 MAX_XMIT，另一个系统运行 MAX_RECV。在使用调制解调器控制信号运行这两个程序时，应该交错开启操作。完成测试后，会退出两个程序。你也可以使用 Ctrl-Break 来退出测试。

两个系统应该用一条 null 调制解调器导线连接起来。通常在任意一个货源充足的计算机商场都可以找到这种导线。图 12-3 展示了 null 调制解调器导线是如何连接的。



* 如果系统 1 运行 MAX_XMIT，而系统 2 运行 MAX_RECV，那么这条信号线没有使用

图 12-3 null 调制解调器导线的连接

```
; ---------------------------------------------------------------
;                          MAX_XMIT
; ---------------------------------------------------------------
;
;    本程序和对应的程序MAX_RECV一起来
```

```
;   确定两个系统之间最大的持久传输串行速率，
;   这两个系统通过一个null调制解调线相连。串
;   口1为缺省值，除非命令行指明用其他串口。
;   MAX_XMIT x表示使用端口x（1-4）
;
;   版权所有1994，1996 Frank van Gilluwe
;   保留所有权利

include undocpc.inc

cseg    segment para public
        assume  cs:cseg, ds:cseg, ss:tsrstk

max_xmit        proc    far

msg1    db      CR, LF
        db      'MAX_XMIT - Finds Maximum speed on serial link'
        db      CR, LF, '$'

msg2    db      CR, LF, '  Transmitter Loaded and ready.'
        db      '  Waiting for receiver MAX_RECV connection.'
        db      CR, LF
        db      '  (Press Ctrl-Break to exit)', CR, LF, '$'

msg3    db      'Testing at baud rate: $'
msgok   db      '  Test OK', CR, LF, '$'
msgfail db      '  Test failed', CR, LF, '$'
msgerr  db      '  Serial port not present', CR, LF, '$'


divisor db      30h, '2,400    $'  ; 除法因子和波特率表
div2nd  db      18h, '4,800    $'  ;  测试速率
        db      0Ch, '9,600    $'
        db      8,   '14,400   $'
        db      6,   '19,200   $'
        db      3,   '38,400   $'
```

```
        db      2,   '57,600 $'
        db      1,   '115,200$'
divisor_end     label   byte

buffer  db      512 dup (31h)           ; 发送字节缓冲区

start:
        mov     bx, 0                   ; 默认端口1
        cmp     byte ptr ds:[80h], 0    ; 命令行选项?
        je      max_skp1                ; 如果无，则跳转
        mov     al, ds:[82h]            ; 获取命令行的值
        sub     al, 31h                 ; 转化成数字0~3
        cmp     al, 3
        ja      max_skp1                ; 如果越界则忽略
        mov     bl, al                  ; 保存端口号
max_skp1:
        push    cs
        pop     ds
        mov     ax, 40h                 ; BIOS 数据区段值
        mov     es, ax
        mov     byte ptr es:[71h], 0    ; 强制标志关闭
        OUTMSG  msg1                    ; 输出初始信息

; 获取串行起始I/O口地址并设置指针

        shl     bx, 1                   ; 要获取的I/O端口
        mov     dx, es:[bx]             ; 起始I/O端口地址
        cmp     dx, 0
        jne     valid_port
        jmp     no_serial_port          ; 如果是0，则退出

valid_port:
        mov     bp, dx                  ; 保存起始端口地址
        mov     di, offset divisor      ; 指向除法因子表

; 通知远程系统我们准备好了
```

```
        add     dx, 4                   ; 调制解调控制寄存器
        mov     al, 3                   ;  设置请求发送和数据终
        out     dx, al                  ;  端准备好

        OUTMSG  msg2                    ; 输入“等待信息”

; 检查远程系统是否准备好。
;   数据设置准备好和清除以便发送线必须都为高电平

next_block:
        mov     dx, bp

; 设置调制解调器为8位数据，1个停止位，奇校验，
;   并设置当前的波特率除法因子

        add     dx, 3                   ; 线控制寄存器
        mov     al, 10001011b           ;  8位，1个停止位，奇校验
        out     dx, al                  ;  DLAB=1来装入波特率
        IODELAY
        mov     dx, bp                  ;  除法因子锁存LSB
        mov     al, [di]                ;  获取除法因子的LSB
        out     dx, al                  ;  设置除法因子的LSB
        IODELAY
        inc     dx
        mov     al, 0                   ;  除法因子的MSB为0
        out     dx, al                  ;  设置除法因子的MSB
        IODELAY
        add     dx, 2                   ; 线控制寄存器
        mov     al, 00001011b           ;  关闭DLAB
        out     dx, al
        IODELAY

        add     dx, 3                   ; 调制解调状态寄存器

max_loop1:
```

```
        in      al, dx
        IODELAY
        not     al
        test    al, 0A0h                ; 远程系统准备好?
        jz      ready_to_send           ; 如果是则跳转（DSR和DCD=1）
        test    byte ptr es:[71h], 80h  ; 出现中止?
        jz      max_loop1               ; 如果没有，则跳转
        jmp     max_xmit_end            ; 退出，Ctrl-Break

ready_to_send:
        OUTMSG  msg3                    ; 输出测试...
        mov     dx, di
        inc     dx                      ; 指向波特率文本
        mov     ah, 9
        int     21h                     ; 输出波特率数

        mov     si, offset buffer       ; 发送字节的缓冲区
        mov     cx, 512                 ; 发送512字节
        call    send_block              ; 传送512字节

; 等待接收方提示它完成块接收

max_xmit_wait:
        test    byte ptr es:[71h], 80h  ; 出现中止?
        jnz     max_xmit_end            ; 若是，则退出
        mov     dx, bp
        add     dx, 6                   ; 调制解调状态寄存器
        in      al, dx
        test    al, 80h                 ; 远程系统完成，DCD=0?
        jnz     max_xmit_wait           ; 循环直到完成
        test    al, 20h                 ; 远程错误? DSR=0
        jz      max_xmit_fail           ; 如果是，则跳转

; 接收顺利，输出OK进行下一项测试

        OUTMSG  msgok
```

```
        add     di, offset div2nd - offset divisor
        cmp     di, offset divisor_end
        jb      next_blocka
        jmp     max_xmit_end
next_blocka:
        jmp     next_block

max_xmit_fail:
        OUTMSG  msgfail                 ; 波特率失败

max_xmit_end:
        mov     dx, bp
        add     dx, 4                   ; 调制解调控制寄存器
        mov     al, 0                   ;  设置DTR关
        out     dx, al                  ;  完成!
        jmp     max_xmit_done

no_serial_port:
        OUTMSG  msgerr                  ; 输出错误信息

max_xmit_done:
        mov     ah, 4Ch
        int     21h                     ; 退回到DOS
max_xmit endp

; ---------------------------------------------------------------------------
;   SEND_BLOCK
;       通过串行线尽可能快地发送一块数据。如果
;       检测到Ctrl-Break键或接收清除以便发送不再
;       处于激活状态，则退出
;
;       调用:       cx = 要传送的字节数
;                   bp = 起始端口I/O地址
;                   ds:si = 数据源
;
;       返回:       字节发送到串行线上，除非
```

```
;                              按下了Ctrl-Break键
;
;       用到的寄存器:    al, cx, dx, si

send_block proc near
        mov     dx, bp
        cld                             ; 清除方向标志位

send_next:
        add     dx, 5
wait_loop1:
        test    byte ptr es:[71h], 80h  ; 出现中止?
        jnz     send_exit               ; 若是，则退出
        inc     dx                      ; 调制解调状态寄存器
        in      al, dx
        IODELAY
        test    al, 80h                 ; 远程问题? DCD=0
        jz      send_exit               ; 若是，则跳转
        dec     dx                      ; 线状态寄存器
        in      al, dx
        IODELAY
        test    al, 20h                 ; 保持寄存器准备好?
        jz      wait_loop1              ; 若非，则跳转

        mov     dx, bp                  ; 发送保持寄存器
        lodsb                           ; 从[si]获取字节并保存在al中
        out     dx, al                  ; 发送字节
        IODELAY
        loop    send_next

send_exit:
        ret
send_block endp

; ---------------------------------------------------------------------------
```

```
;                                   MAX_RECV
; ------------------------------------------------------------------------------
;
;   本程序和对应程序MAX_XMIT一起测出两台用null调制
;   解调线相连的两个系统之间最大的持久串行传输速率。串口
;   1为默认值，除非命令行参数指定了其他值。MAX_RECV
;   x表示命令执行指定了端口号x（1-4）
;
;   （c）版权所有1994，1996 Frank van Gilluwe
;   保留所有的权利

include undocpc.inc

cseg    segment para public
        assume  cs:cseg, ds:cseg, ss:tsrstk

max_recv        proc    far

msg1    db      CR, LF
        db      'MAX_RECV - Finds Maximum speed on serial link'
        db      CR, LF, '$'

msg2    db      CR, LF, '  Ready to receive data.'
        db      '  Waiting for transmitter MAX_XMIT connection.'
        db      CR, LF
        db      '  (Press Ctrl-Break to exit)', CR, LF, '$'

msg3    db      'Testing at baud rate: $'
msgok   db      '  Test OK', CR, LF, '$'
msgfail db      '  Test failed - $'

msgovr  db      'Overrun error', CR, LF, '$'
msgpar  db      'Parity error', CR, LF, '$'
msgfra  db      'Framing error', CR, LF, '$'
msgdata db      'Data is invalid', CR, LF, '$'
```

```
msgerr  db      ' Serial port not present', CR, LF, '$'

divisor db      30h, '2,400   $'  ; 除法因子和波特率表
div2nd  db      18h, '4,800   $'  ;  测试速率
        db      0Ch, '9,600   $'
        db      8,   '14,400 $'
        db      6,   '19,200 $'
        db      3,   '38,400 $'
        db      2,   '57,600 $'
        db      1,   '115,200$'
divisor_end     label   byte

buffer  db      512 dup (0h)            ; 要发送字节的缓冲区

ctrlb   db      0                       ; 1=Ctrl-Break按下


start:
        mov     bx, 0                   ; 默认端口1
        cmp     byte ptr ds:[80h], 0    ; 命令行选项?
        je      max_skp1                ; 若没有，则跳转
        mov     al, ds:[82h]            ; 获取命令行值
        sub     al, 31h                 ; 转化成数字0~3
        cmp     al, 3
        ja      max_skp1                ; 如果越界则跳过
        mov     bl, al                  ; 保存端口号
max_skp1:
        push    cs
        pop     ds
        mov     ax, 40h                 ; BIOS数据区段址
        mov     es, ax
        mov     byte ptr es:[71h], 0    ; 强制中止
        OUTMSG  msg1                    ; 输出初始信息

; 获取串行起始I/O端口地址并设置指针
```

```
        shl     bx, 1                   ; 获取I/O端口
        mov     dx, es:[bx]             ; 起始I/O端口地址
        cmp     dx, 0
        jne     valid_port
        jmp     no_serial_port          ; 若是零，则退出

valid_port:
        mov     bp, dx                  ; 保存起始端口地址
        mov     si, offset divisor      ; 指向除法因子表的指针

; 通知远程系统我们准备好

        add     dx, 4                   ; 调制解调控制寄存器
        mov     al, 1                   ;  设置数据终端准备好
        out     dx, al                  ;  但是还没有准备好发送
        IODELAY

        OUTMSG  msg2                    ; 输出“等待信息”

; 检查远程系统是否准备好。数据设置准备
;   好处于高电平

        mov     dx, bp
        add     dx, 6                   ; 调制解调状态寄存器

max_loop1:
        in      al, dx
        test    al, 20h                 ; 远程系统准备好?
        jnz     ready_to_recv           ; 如果是则跳转 (DSR = 1)
        test    byte ptr es:[71h], 80h  ; 出现中止?
        jz      max_loop1               ; 若非，则跳转
        jmp     max_recv_end            ; 退出，Ctrl-Break

; 调制解调器设置为8位数据，1个停止位，奇校验，
;   并设置当前测试波特率除法因子
```

```
ready_to_recv:
        mov     dx, bp
        add     dx, 3                   ; 线控制寄存器
        mov     al, 10001011b           ;  8位，1位停止，奇校验
        out     dx, al                  ;  DLAB=1 来装入波特率
        IODELAY
        mov     dx, bp                  ; 除法因子锁存LSB
        mov     al, [si]                ;  获取除法因子的LSB
        out     dx, al                  ;  设置除法因子的LSB
        IODELAY
        inc     dx
        mov     al, 0                   ;  除法因子的MSB是0
        out     dx, al                  ;  设置除法因子的MSB
        IODELAY
        add     dx, 2                   ; 线控制寄存器
        mov     al, 00001011b           ;  关闭DLAB
        out     dx, al
        IODELAY

; 清空输入缓冲区中的所有内容

        mov     dx, bp                  ; 装入起始地址
        add     dx, 5                   ; 线状态寄存器
        in      al, dx
        test    al, 1                   ; 接收到字节?
        jz      skip_input              ; 若非，则忽略
        mov     dx, bp                  ; 接收寄存器
        in      al, dx                  ; 清空接收缓冲区
skip_input:
        OUTMSG  msg3                    ; 输出测试...
        mov     dx, si
        inc     dx                      ; 指向波特率文本
        mov     ah, 9
        int     21h                     ; 输出波特率数

        mov     dx, bp
```

```
        add     dx, 5                   ; 线状态寄存器
        in      al, dx                  ; 一次读会清除错误标志
        mov     di, offset buffer       ; 接收字节缓冲区
        mov     cx, 512                 ; 预计512字节
        call    get_block               ; 预计512字节并放入ds:[di]中

; 检查是否出错。若是，则退出（al=线状态）

        test    byte ptr es:[71h], 80h  ; 出现中止?
        jnz     max_recv_end            ; 若是，则退出
        test    al, 0Eh                 ; 接收时出错?
        jnz     max_recv_fail           ; 若是，则跳转

        mov     cx, 512                 ; 检查数据是否有效
        mov     di, offset buffer
        mov     al, 0

max_loop2:
        cmp     byte ptr [di], 31h      ; 缓冲区中的值正确?
        jne     max_recv_fail           ; 若非，则跳转
        mov     byte ptr [di], al       ; 地址回零，供下次测试
        inc     di
        loop    max_loop2

; 测试成功，设置RTS=0输入成功并设置下一次的波特率

        mov     dx, bp
        add     dx, 4                   ; 调制解调控制寄存器
        mov     al, 1                   ;  设置数据终端准备好
        out     dx, al                  ;  但是请求发送目前处于关闭状态

        OUTMSG  msgok
        add     si, offset div2nd - offset divisor
        cmp     si, offset divisor_end
        jae     delay_a_bit             ; 如果上次除法因子操作完成则跳转
        jmp     ready_to_recv
```

```
delay_a_bit:
        mov     cx, 0FFFFh          ; 短延时确认
max_delay:                          ;  在设置之前发送器
        loop    max_delay           ;  接收到RTS=0信号
        jmp     max_recv_end        ;  DTR=0

max_recv_fail:
        push    ax
        OUTMSG  msgfail             ; 输出测试失败信息
        pop     ax
        mov     dx, offset msgovr   ; 考虑过载错误
        test    al, 2               ; 过载错误?
        jnz     max_recv_fail2      ; 若是，则跳转
        mov     dx, offset msgfra   ; 考虑帧错误
        test    al, 8               ; 帧错误?
        jnz     max_recv_fail2      ; 若是，则跳转
        mov     dx, offset msgpar   ; 考虑奇偶错误
        test    al, 4               ; 奇偶错误?
        jnz     max_recv_fail2      ; 若是，则跳转
        mov     dx, offset msgdata  ; 一定是数据出错（al=0）
max_recv_fail2:
        mov     ah, 9
        int     21h                 ; 输出错误类型

max_recv_end:
        mov     dx, bp
        add     dx, 4               ; 调制解调控制寄存器
        mov     al, 0               ;  关闭DTR和RTS告诉发
        out     dx, al              ;  送器停止发送
        jmp     max_recv_done

no_serial_port:
        OUTMSG  msgerr              ; 输出错误信息

max_recv_done:
```

```
        mov     ah, 4Ch
        int     21h                     ; 退回到DOS
max_recv endp

; ---------------------------------------------------------------------------
;   GET_BLOCK
;       通过串行连线尽可能快地接收数据块。
;       如果按下Ctrl-Break或出错，则退出。
;       没有定时问题
;
;       调用:     cx = 期望接收的字节数
;                 bp = 起始端口I/O地址
;              es:di = 数据目的地址
;
;       返回:         通过串行连线接收字节，除
;                         非按下了Ctrl-Break键
;
;       用到的寄存器:   ax, bx, cx, dx, di

get_block proc near
        push    si
        push    ds
        push    es

        mov     ax, ds
        mov     es, ax              ; es = 缓冲区段址
        mov     ax, 40h
        mov     ds, ax              ; ds = BIOS数据段址
        mov     si, 71h             ; BIOS中止标志的偏移量
        mov     ah, 5               ; 供调整DX用
        mov     bh, 8Eh             ; 测试错误和中止屏蔽
        mov     bl, 1               ; 如果字节可用则测试屏蔽

        mov     dx, bp
        add     dx, 4               ; 调制解调控制寄存器
        mov     al, 3               ; 设置数据终端准备好和
```

```
        out     dx, al                  ; 请求发送

        cld                             ; 清除方向标志
        mov     dx, bp                  ; 起始I/O地址

; 下面的循环已是最优，并且在寄存器中预先装入了固定的值。
; 此外，错误条件（线状态寄存器的第3~1位）和位于40:71h处
; 的BIOS中止位7也组合起来作为一次单独的退出测试。假定无
; FIFO模式，即线状态寄存器的第7位始终打开。STOSB指令将
; AL装入到ex:[di]中并增加DI

get_next:
        add     dl, ah                  ; 线状态寄存器
wait_loop1:
        in      al, dx                  ; 获取错误和数据准备好位
        or      al, [si]                ; 插入中止位7
        test    al, bh                  ; 错误或中止(bh=8E) ?
        jnz     get_exit                ; 若是，则跳转
        test    al, bl                  ; 接收到了字节 (bl=1) ?
        jz      wait_loop1              ; 若无，则循环
        mov     dx, bp                  ; 接收寄存器
        in      al, dx                  ; 获取字节
        stosb                           ; 将字节保存到es:[di]
        loop    get_next

        mov     al, 0                   ; 无错误
get_exit:
        pop     es
        pop     ds
        pop     si
        ret
get_block endp
```

端口归纳

下面的端口列表用于串行口操作。“平台”一列中的星号表示目前很少使用这个端口，

这里所列出的信息仅供参考，我没有为这些端口提供进一步的细节。

RS-232 串行口通过排列在一起的 8 个 I/O 端口访问 UART。基 I/O 地址参照第一个 I/O 端口组。最常见的基地址是 3F8h 和 2F8h，当然其他的地址值也比较常见。下面这个归纳列出了常见的基地址。在归纳后面的“UART 寄存器细节”一节中详细阐述了每个寄存器组。

端口	类型	功能	平台
2E8~2EFh	I/O	RS-232 串行口组	适配器
2F8~2FFh	I/O	RS-232 串行口组	适配器
380h	I/O	SDLC 串行口，8255 内部/外部传感	*
380h	I/O	位同步串行口，8255 内部/外部传感	*
381h	I/O	SDLC 串行口，8255 外部调制解调器接口	*
381h	I/O	位同步串行口，8255 内部/外部调制解调	*
382h	I/O	SDLC 串行口，8255 内部控制	*
382h	I/O	位同步串行口，8255 内部控制	*
383h	I/O	SDLC 串行口，8255 模式初始化	*
383h	I/O	位同步串行口，8255 模式初始化	*
384h	I/O	SDLC 串行口，8253 方波发生器计数器	*
384h	I/O	位同步串行口，8253 未使用的计数器	*
385h	I/O	SDLC 串行口，8253 休眠超时期限 1	*
385h	I/O	位同步串行口，8253 休眠超时期限 1	*
386h	I/O	SDLC 串行口，8253 休眠超时期限 2	*
386h	I/O	位同步串行口，8253 休眠超时期限 2	*
387h	I/O	SDLC 串行口，8253 时钟模式集	*
387h	I/O	位同步串行口，8253 时钟模式集	*
388h	输入	SDLC 串行口，8273 状态	*
388h	输出	SDLC 串行口，8273 命令	*
388h	I/O	位同步串行口，8251 数据	*
389h	输入	SDLC 串行口，8273 状态	*
389h	输出	SDLC 串行口，8273 参数	*
389h	输入	位同步串行口，8251 状态	*
389h	输出	位同步串行口，8251 命令	*
38Ah	I/O	SDLC 串行口，8273 传送中断状态	*

38Bh	I/O	SDLC 串行口，8273 接收中断状态	*
38Ch	I/O	SDLC 串行口，8273 数据	*
3A0h	I/O	位同步串行口，8255 内部/外部传感	*
3A1h	I/O	位同步串行口，8255 外部调制解调器接	*
3A2h	I/O	位同步串行口，8255 内部控制	*
3A3h	I/O	位同步串行口，8255 模式初始化	*
3A4h	I/O	位同步串行口，8253 未使用的计数器	*
3A5h	I/O	位同步串行口，8253 休眠超时期限 1	*
3A6h	I/O	位同步串行口，8253 休眠超时期限 2	*
3A7h	I/O	位同步串行口，8253 时钟模式集	*
3A8h	I/O	位同步串行口，8251 数据	*
3A9h	输入	位同步串行口，8251 状态	*
3A9h	输出	位同步串行口，8273 命令	*
3E8~3EFh	I/O	RS-232 串行口组	适配器
3F8~3FFh	I/O	RS-232 串行口组	适配器
3220~3227h	I/O	RS-232 串行口组	仅 MCA
3228~322Fh	I/O	RS-232 串行口组	仅 MCA
4220~4227h	I/O	RS-232 串行口组	仅 MCA
4228~422Fh	I/O	RS-232 串行口组	仅 MCA
5220~5227h	I/O	RS-232 串行口组	仅 MCA
5228~522Fh	I/O	RS-232 串行口组	仅 MCA

* 过时的适配器——仅供参考

UART 寄存器细节

本部分详细阐述了 UART 寄存器，这些寄存器用来访问 RS-232 串行口。在基 I/O 端口加上寄存器号就可以访问指定的寄存器。四个串行口的基 I/O 端口地址保存在 BIOS 数据区中。例如，要访问调制解调器寄存器 6，应该在基 I/O 端口上加上 6。如果基地址是 2E8h，那么用来访问这个调制解调器寄存器状态的地址是 2E8h+6，即 2EEh。

表 12-11 和 12-12 展示了 UART 的大致面貌。请注意，寄存器 3 中的除法因子锁存访问位（DLAB）可以控制访问不同的功能，这些功能主要涉及寄存器 0 和 1 的功能，有时也包括寄存器 2 的功能。

表 12-11 UART 寄存器概貌，DLAB 为 0

	寄存器	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0R	接收缓冲区	接收到的数据（5~8 位，位 0 总是最低有效位，并且第一个接收）							
0W	传送缓冲区	要发送的数据（5~8 位，位 0 总是最低有效位，并且第一个发送）							
1	中断允许	0	0	0 时钟 (82510)	0 传送机器 (82510)	模式状态	接收器线状态	传送保持寄存器空	接收数据可用
2R	中断标识	0 开放 FIFO	0 （1655x） 选择组，82510 00=组 0 缺省） 01=组 1 10=组 2 11=组 3 (没有展示组 1、2、3)	0	0	中断类型 0000=调制解调器的状态发生了改变 0001=无中断挂起 0010=传送保持寄存器空 0100=接收数据可用 0110=接收器线状态 1000=传送机器（仅 82510） 1010=时钟中断（仅 82510） 1100=字符超时（仅 1655x）			
2W	FIFO 控制（仅 1655x）	接收器触发水平 00=FIFO 中有 1 个字节 01=FIFO 中有 4 个字节 10=FIFO 中有 8 个字节 11=FIFO 中有 14 个字节		0	0	DMA 模式选择	传送 FIFO 重启	接收 FIFO 重启	FIFO 开放
3	线控制	DLAB （控制寄存器 0 和 1		暂停发送	奇偶位模式 00=奇 01=偶 10=总是 0 11=总是 1	开放奇偶	停止位的数目	字符字长度 00=5 位 01=6 位 10=7 位 11=8 位	
4	调制解调器控制	0 重启和断电，161450	0	0 OUT0，82510	回环	-OUT2 开放 IRQ	-OUT1 软件重启，161450	请求发送	数据终端准备好
5	线状态	0 接收 FIFO 出错，（1655x）	传送移位寄存器空	传送保持寄存器空	出现暂停中断	出现帧错误	出现奇偶错误	出现过载错误	接收到的数据准备好
6	调制解调器状态	数据载波检测	响铃指示器	数据设置准备好	清除以便发送	数据载波检测线发生了改变	响铃指示器发生了改变	数据设置准备好线发生改变	清除以便发送线发生了改变
7	缓冲暂存区 特殊控制	数据字节（UART 没有使用，早期的 8250 不支持） 数据字节（未使用、地址或者控制字符，由 82510 的选项决定）							

阴影代表某些高级的特性或者属性，只有某些 UART 才支持它们

表 12-12 UART 寄存器概貌，DLAB 为 1

#	寄存器	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
0	除法因子锁存器的 LSB	除法因子字的最低有效数据字节							
1	除法因子锁存器的 MSB	除法因子字的最高有效数据字节							
2	可替代功能（16552）	0	0	0	0	0	定义 OUT2 功能 00=开放 IRQ（缺省） 01=波特率时钟超时 10=接收准备好超时 11=未定义		同时向两个 UART 写

阴影代表某些高级的特性或者属性，只有某些 UART 才支持它们

寄存器	类型	描述	平台
0	I/O	传送/接收缓冲区（DLAB=0）	适配器

这个寄存器是一排 8 个 I/O 地址的第一个寄存器。向这个寄存器输出会在 UART 的传送保持缓冲区中保存一个字节。输入会从 UART 的接收缓冲区中读取一个字节。在 UART 内它们是两个独立的寄存器。

要访问这个寄存器，必须将除法因子锁存器设置为 0。线控制寄存器 3 的 DLAB 位是第七位。

输入（位 0~7） 获取接收缓冲区数据字节

输出（位 0~7） 保存传送缓冲区数据字节

寄存器	类型	描述	平台
0	I/O	除法因子锁存器的 LSB（DLAB=1）	适配器

这个寄存器保存了波特率除法因子锁存器的最低有效位字节。用期望的波特率除以 115,200，可以得到给定波特率下的除法因子。参看有关波特率的阐述部分以了解相关细节。可以参看表 12-7 了解常见波特率对应的除法因子。

要访问这个寄存器，必须将除法因子锁存器访问位 DLAB 设置为 1。线控制寄存器 3 的第七位是 DLAB 位。

I/O（位 0~7） 波特率除法因子锁存器的 LSB

寄存器	类型	描述	平台
1	I/O	中断开放寄存器（DLAB=0）	适配器

可以通过这个寄存器控制串行口的中断操作。如果在 UART 中出现了指定的操作，那

么在开放中断时会触发中断。大多数串行口适配卡可以使用跳线选择 IRQ3 或 IRQ4。参看寄存器 2 和表 12-8 了解其他相关细节。硬件重启会将所有的位清为 0，以禁止所有的中断。

要访问这个寄存器，必须先将除法因子锁存器访问位（DLAB）设置为 0。线控制寄存器 3 的第七位就是 DLAB。

I/O（位 0~7）

位		
	7 r = 0	未使用
	6 r = 0	未使用
	5 r/w = 0	未使用，或者 82510 禁止时钟中断
	1	开放时钟中断（仅 82510）
	4 r/w = 0	未使用，或者禁止传送机中断（仅 82510）
	1	开放传送机中断（仅 82510）
	3 r/w = 1	开放“Modem 状态”中断
	2 r/w = 1	开放“接收器线状态”中断
	1 r/w = 1	开放“传送保持寄存器空”中断
	0 r/w = 1	开放“接收到的数据可用”中断。在 16550 芯片上，如果工作在 FIFO 模式下，那么也开放超时中断。

寄存器	类型	描述	平台
1	I/O	除法因子锁存器的 MSB（DLAB=1）	适配器

这个寄存器保存了波特率除法因子锁存器的最高有效位字节。用期望的波特率除以 115,200，可以得到给定波特率下的除法因子。参看有关波特率的阐述部分以了解相关细节。可以参看表 12-7 了解常见波特率对应的除法因子。

要访问这个寄存器，必须将除法因子锁存器访问位 DLAB 设置为 1。线控制寄存器 3 的第七位是 DLAB 位。

I/O（位 0~7） 波特率除法因子锁存器的 MSB

寄存器	类型	描述	平台
2	输入	中断标识寄存器（DLAB=0）	适配器

中断标识寄存器指示了当前是否正在执行一个中断以及中断的起因。硬件重启后，中断标识寄存器设置为 1，表示无中断正在执行。

输入（位 0~7）

位		
	7r = x 6r = x	型号标志 / FIFO 模式 位 7　位 6

	0	0=关闭 FIFO，或者不是 16550	
	0	1=开启 FIFO，兼容 16550	
	1	0=开启 FIFO，老式 16550（坏 FIFO）	
	1	1=开启 FIFO，新式 16550 或更晚的芯片	
5r=0	未使用		
4r=0	未使用		
3r=x 2r=x 1r=x	中断标识（在表 12-8 中有详细的说明）		
	位 3	位 2	位 1
	0	0	0=调制解调器状态
	0	0	1=传送保持寄存器空
	0	1	0=接收数据可用
	0	1	1=接收器线状态
	1	1	0=字符超时
0r=0	有中断挂起		
1	无中断挂起		

寄存器	类型	描述	平台
2	输出	FIFO 控制寄存器；DLAB=0	通信器

向 FIFO 控制寄存器写入一个字节。仅 16650 才支持这个寄存器。在设置 FIFO 选项时，位 0（FIFO 开放位）必须设置为 1，以便在写周期内可以改变其他任意位。

在硬件重启后，FIFO 控制寄存器会被设置为 0，表示没有活动的 FIFO 模式。

输出（位 0~7）FIFO 控制（仅限于 16550 系列的芯片）

位	7w=x 6w=x	接收器 FIFO 中断触发水平	
		位 7	位 6
		0	0=FIFO 中有 1 个字节就触发中断
		0	1=FIFO 中有 4 个字节就触发中断
		1	0=FIFO 中有 8 个字节就触发中断
		1	1=FIFO 中有 14 个字节就触发中断
	5w=0	未使用（忽略写操作）	
	4w=0	未使用（忽略写操作）	
	3w=x	将 UART 的接收和发送准备好线设置为操作模式 0 或模式 1。这两种模式控制这些线如何反映 FIFO 和保持寄存器状态。通常这些来自 UART 的线是被断开的，所以这些功能并不重要	
	2w=1	清空传送 FIFO 中的所有内容，并将内容计数器重新设置为零。这个功能不清除传送移位寄存器，所以不会影响传入的字节。这一位会自动重新设置为 0	

	1w=1	清空接收 FIFO 中的所有内容，并将内容计数器重新设置为零。这个功能不清除接收移位寄存器，所以不会影响传入的字节。这一位会自动重新设置为 0
	0w=0	禁止 FIFO，并清空 FIFO 的内容
	1	开放接收和传送 FIFO

寄存器	类型	描述	平台
2	I/O	替代功能寄存器（DLAB=0）	适配器

16552 双 UART 提供了这个新的寄存器，但是在其他的类型上没有这个寄存器。这个替代功能寄存器控制使用 OUT2 的方法。这个寄存器也允许 I/O 同时对两个 UART 写操作。只有当 DLAB 的位 7 是 1 时，才可以访问这个替代功能寄存器（参看寄存器 3）。

位 0 控制双 UART 的写选项，这一点很有用，因为它可以删掉编程所需要的一半数目的寄存器，以加速初始化过程。双 UART 通常作为两个独立的端口集来访问，从功能来说，两者一般是独立的，只有这个寄存器功能比较特殊。从任意一个寄存器集都可以访问这个寄存器，设置或清除这个位只需要一次操作。设置时，对任意一个寄存器集所做的写操作都会复制到另外一个寄存器。但是读操作则彼此没有影响。记住，必须对 DLAB 执行写操作以确保两个 UART 使用相同的 DLAB 状态。

在其他的 UART 上没有这个寄存器。这时，设置 DLAB 的操作不起任何作用。只有在非 16552 的 UART 上才可以访问中断标识寄存器/FIFO 控制。

I/O（位 0-7）替代功能寄存器（仅 16552）

位	7r=0	未使用
	6r=0	未使用
	5r=0	未使用
	4r= 0	未使用
	3r=0	未使用
	2r/w=x 1r/w=x	OUT2 类型 位 2　位 1
		0　0=中断开放（缺省）
		0　1=输出波特率时钟×16
		1　0=FIFO 已经接收到了数据（可用作 DMA 请求）
		1　1=未使用
	0r/w = 0	普通情况
	1	向单个 UART 寄存器的任意写操作都会同时向两个 UART 写（参看上文所述）

寄存器	类型	描述	平台
3	I/O	线控制寄存器	适配器

这个寄存器控制串行帧的结构。在硬件重启后这个寄存器被设置为 0。由于重启后建立的是一个很少使用的 5 位数据奇校验帧，所以必须在线控制寄存器中装入合适的值，以获取期望类型的帧。目前常用的帧有 8 位数据，没有奇偶位，有 1 个停止位。

这个寄存器包含有除法因子锁存器地址位（DLAB）。DLAB 位控制对波特率发生器除法因子的访问（这时 DLAB 必须设置为 1），或者控制对接收缓冲区、传送保持寄存器或中断开放寄存器的访问（这时 DLAB 必须设置为 0）。

I/O（位 0~7）

位		
	7r/w=x	除法因子锁存器地址位——DLAB（参看上文）
	6r/w=0	关闭了暂停控制（普通情况）
	1	强制串行输出低电平来传送一个暂停条件。在接收器触发一个短期暂停时，会在接收 UART 的线状态寄存器中设置暂停标志。这个暂停周期的最小值应是当前波特率下的两个完整的字符帧传送所用的时间。
	5r/w=0	禁止粘性奇偶位操作
	1	粘性奇偶位操作强制奇偶位等于位 4 的求反结果（如果允许奇偶位）。这意味着在串行帧中有一个固定的奇偶位。总是传送相同的位值或者接收校验相同的奇偶位值。
	4r/w=0	奇奇偶校验（如果允许）
	1	偶奇偶校验（如果允许）
	3r/w=0	串行帧中无奇偶校验位
	1	允许奇偶校验位——传送时，在数据位后发送一个奇偶位。接收时，校验奇偶位以确定传送的有效性。
	1r/w=x	数据长度
	0r/w=x	位 1　位 0
		0　0=5 位
		0　1=6 位
		1　0=7 位
		1　1=8 位

寄存器	类型	描述	平台
4	I/O	调制解调器控制寄存器	适配器

这个寄存器提供了许多控制功能。它用于设置连接器输出线 RTS 和 DTR 的状态。其中一位允许禁止中断请求线而不禁止内部 UART 中断状态。回环位用于测试 UART 的内部传送和接收操作。参看回环操作一节以了解有关这个特性的完整信息。硬件重启后这个寄存器设置为 0。

I/O（位 0~7）调制解调器控制寄存器（16C1450/1550 系列除外）

位		
	7r=0	未使用
	6r=0	未使用
	5r/w=x	未使用，或者在 82510 上用作-OUT0 线
	4r/w=1	开放接收和传送之间的回环
	3r/w=1	-OUT2 线，用于开放中断请求（IRQ）
	2r/w=x	-OUT1 线，未连接或者未使用
	1r/w=x	设置连接器上请求发送线（RTS）的状态
	0r/w=x	设置连接器上数据终端准备好线（DTR）的状态

16C1450 和 16C1550 对这个寄存器提供了另外两个功能。一种掉电模式允许系统节省功耗。在掉电模式下会保留所有的寄存器，但是会关闭振荡器。这两个型号的芯片支持软件像硬件那样控制重启操作。其中第二位用于重启操作。但是对于其他型号的 UART 来说，第二位控制未使用的 OUT1 线。

I/O（位 0~7）调制解调器控制寄存器（16C1450/1550 系列）

位		
	7r=0	上电，普通操作
	1	省电模式
	6r=0	未使用
	5r/w=0	未使用
	4r/w=1	开放接收和传送之间的回环
	3r/w=1	允许开放中断请求（IRQ）
	2r/w=0	普通操作
	1	重启 UART（类似于一次硬件重启）
	1r/w=x	设置连接器上请求发送线（RTS）的状态
	0r/w=x	设置连接器上数据终端准备好线（DTR）的状态

寄存器	类型	描述	平台
5	输入	线状态	通用

此个寄存器保存了与收发数据有关的状态和错误信息。如果开放了接收器线状态中断，那么设置这个寄存器的位 1~4 中的任意一位都会引发中断。硬件重启后，这个寄存器被设置为 60h，表示没有错误，传送缓冲区为空。

工作在 FIFO 模式下的 16550 UART 会改变对收到的错误和暂停标志的操作。如果最早进入 FIFO 的字节（CPU 会读取这个字节）发生了溢出，那么在这个寄存器的位 1~4 中会设置相应的标志。

此寄存器仅供读操作使用。芯片不支持写操作，这些写操作一般在工厂测试时用到。芯片制造商不推荐对这个端口执行写操作，但是也没有指出写操作会出现什么问题。我自己对许多供应商的 16450 芯片的测试结果表明，位 0~4 是可写的。将这些位设置为高电平

不会发生中断。制造商们指出了某些位在FIFO模式下不可写。

输入（位0~7）

位		
	7r=0	未使用，或者16550位于非FIFO模式下
	1	位于FIFO模式下，且出现了至少一个错误条件，这些条件被保存在了FIFO中。这些错误条件包括奇偶错误、帧错误或者一个暂停指示。
	6r=0	传送保持寄存器或移位寄存器中有一个字节
	1	传送保持寄存器和移位寄存器都空
	5r=0	传送寄存器中有一个字节。任何时候CPU装入一个待发送字节时都会设置这个状态。
	1	传送寄存器空，准备好接收一个新的待传送字节。如果开放了传送保持中断，那么当这一位设置为1时会请求中断。在FIFO模式下，只有FIFO为空时才会设置这一位。
	4r=0	普通情况（没有检测到暂停条件）
	1	收到暂停信号。如果收到的信号保持高电平的时间超过了一个完整的串行帧周期，就会发出一个暂停信号。这意味着起始位、数据位以及奇偶校验位都是高电平，但是没有收到停止位（低电平）。读取这个寄存器会清除这个暂停位。
	3r=1	出现了帧错误。在接收一个数据时，检测到的停止位是高电平，但是实际的停止位应该是低电平信号。UART将努力对下一个收到的帧重新进行同步。一个帧错误通常表示有噪声或者信号过弱。读取这个寄存器会清除这个帧错误位。
	2r=1	出现了奇偶错误。所收到的数据的奇偶位与出线控制寄存器（端口3FBh）指定的奇偶性不匹配。读取这个寄存器会清除这个奇偶错误位。
	1r=1	出现过载错误。在CPU从缓冲区读走前一个字节之前收到了另外一个字节。新字节覆盖了前面的字节，所以前面的字节被丢失了。在FIFO模式下，只有在接收FIFO缓冲区满时才会出现过载。读取这个寄存器会清除这个过载错误位。
	0r=0	没有可用的接收数据。在FIFO模式下，接收FIFO为空。
	1	数据准备好指示器。CPU现在可以读取接收到的字节。

寄存器	类型	描述	平台
6	I/O	调制解调器状态	通用型

这个寄存器保存了调制解调器的状态标志。这些标志指示了当前控制线的状态，以及从上次读取这个寄存器后没有改变的线的状态。

一旦设置了状态改变位 0~3，并且开放了调制解调器状态中断，就会生成一个中断。在读这个寄存器或者硬件重启时，会将位 0~3 设置为 0。

在回环模式下，位 4~7 指示了调制解调器控制寄存器的状态。对于测试 UART 内部位的粘附和交叉情况，这一点非常有用。表 12-13 显示了这些位在回环模式下是如何连接的。

表 12-13　回环模式下的位

调制解调器	调制解调器
状态端口 3FEh	控制端口 3FCh
位 7	位 3，OUT2
位 6	位 2，OUT1
位 5	位 0，DTR
位 4	位 1，RTS

I/O（位 0~7）

位	7r/w=x	连接器的数据载波检测线。在回环模式下，显示 UART 内部的 OUT2 状态。
	6r/w=x	连接器的响铃指示器线。在回环模式下，显示 UART 内部的 OUT1 状态。
	5r/w=x	连接器的数据设置准备好线。在回环模式下，显示 UART 内部的 RTS 状态。
	4r/w=x	连接器的清除以便发送线。在回环模式下，显示 UART 内部的 DTR 状态。
	3r/w=1	数据载波检测线发生了改变
	2r/w=1	响铃检测线发生了改变
	1r/w=1	数据设置准备好线发生了改变
	0r/w=1	清除以便发送线发生了改变

寄存器	类型	描述	平台
7	I/O	缓冲暂存寄存器	适配器

这个寄存器常常被忽略。它是用来访问 UART 内部的一个 8 位读/写寄存器。UART 并不将它用作特殊的用途，而仅仅只是用来保存一些临时的数据。如果从这个端口读取了一个非零的值，你就应该考虑一下是否可能有其他的应用程序这时在使用这个端口。还没有什么方法可以阻止两个应用程序同时使用这个寄存器。

这个寄存器颇令人感兴趣的一个方面是：硬件重启时这个寄存器的内容仍然保持有效。

硬件重启时 UART 和 BIOS 并不清空它。依据我的测试来看，系统首次上电时这个寄存器总是为零，UART 制造商却没有指出是否结果总是如此。

早期的 8250 UART 不支持这个寄存器，而当前它却获得了许多 UART 的广泛支持，这些 UART 包括 8250A、16450 和 16550。82510 UART 也提供了这个寄存器，如果不是将它作为 16450 兼容的 UART 使用，那就是提供了另外两个功能。参看 Intel 8250 手册可了解相关细节。

第 13 章

系统功能

本章包括了通用的 BIOS 系统功能（中断 15h）以及各种不能归纳到其他类型的端口。有许多功能和端口以前从未公开过。

本章的内容包括各种系统控制和状态标志、数学协处理器功能、诊断代码以及 POST 操作。还包括了一些代码，这些代码将有助于解释清楚许多复杂功能的确切用法。

BIOS 服务

中断 15h 系统服务的变化范围很大，有许多功能仅仅局限于特定的平台。某些很早期的 80286 AT BIOS 就不支持某些服务，但是所有后来的 AT+平台却支持这些功能。如果希望程序能够运行在比较大范围的系统上，那么编写程序时应该小心一些，并且应该假定系统可能不支持这个功能，如果这个功能没有获得系统的支持，那么系统将会返回一个错误代码，对所有的功能都是如此。

中断 15h 提供下述服务：

功能	描述	平台
ah=0	开启磁带马达	PC
ah=1	关闭磁带马达	PC
ah=2	从磁带读	PC
ah=3	向磁带写	PC
ah=4	建立系统参数表	MCA
ah=5	建立初始化表	MCA
ah=Fh*	正在格式化	ESDI 驱动器
ah=21h	读/写错误日志	MCA
ah=22h	获取未知段址	PS/1
ah=23h	各种未知功能	PS/1
ah=24h	A20 控制	PS/1
ah=40h	膝上型电脑数据组的读/写	可逆
ah=41h	膝上型电脑等待外部事件	可逆

ah=42h	膝上型电脑断电请求	可逆
ah=43h	膝上型电脑读系统状态	可逆
ah=44h	膝上型电脑电源控制	可逆
ah=49h	获取 BIOS 类型	DBCS
ah=4Fh*	键盘截获	AT+
ah=50h	字体子系统访问	仅 DOS/V
ah=52h	可移走媒介质弹开	EBIOS
ah=80h	设备开	AT+
ah=81h	设备关	AT+
ah=82h	程序终止	AT+
ah=83h	事件等待	AT+
ah=84h	支持游戏杆	AT+
ah=85h*	按下系统请求键	AT+
ah=86h	等待	AT+
ah=87h	访问扩展内存	AT+
ah=88h	扩展内存大小	AT+
ah=89h	切换到保护模式	AT+
ah=90h*	设备忙	AT+
ah=91h*	中断完成	AT+
ah=C0h	返回系统 BIOS 配置	AT+
ah=C1h	返回扩展 BIOS 数据区	AT+
ah=C2h	鼠标 BIOS 接口	PS/2
ah=C3h	监视计时器控制	MCA、EISA
ah=C4h	可编程选项选择	MCA
ah=C5h*	开启系统中断	PS/2
ah=C8h	高速缓存控制	PS/2
ah=C9h	获取 CPU 的级别	某些 386+
ah=CAh	读/写 CMOS 内存	PS/1
ah=D8h	访问 EISA 系统信息	EISA

* 这些功能是 BIOS 才使用的回调功能，不应该由应用程序触发。BIOS 使用这些功能号调用中断 15h。应用程序可以钩起中断 15h，以便在 BIOS 触发这些回调功能时执行某些必要的操作

中断	功能	描述	平台
15h	0	开启磁带马达	PC

开启磁带驱动器马达。这个功能早已过时，并且只有 PC 和 PCjr 才支持这个功能。

调用： ah = 0

返回： 如果支持这个功能

ah = 0，进位 = 0
如果不支持这个功能
ah = 86h，进位 = 1

中断	功能	描述	平台
15h	1	关闭磁带马达	PC

关闭磁带驱动器马达。这个功能早已过时，并且只有 PC 和 PCjr 才支持这个功能。

调用：　ah = 1
返回：　如果支持这个功能
ah = 0，进位 = 0
如果不支持这个功能
ah = 86h，进位 = 1

中断	功能	描述	平台
15h	2	从磁带读	PC

从磁带驱动器读取许多的 256 字节块。读取的字节数必须是 256 的倍数。这个功能早已过时，并且只有 PC 和 PCjr 才支持这个功能。

调用：　ah = 2
cx = 读取的字节数
es:bx=从磁带读取数据后待存入的缓冲区地址
返回：　es:bx=指针，指向上次读取的最后一个字节后的一个字节
dx = 实际读取的字节数
如果没有出错
进位 = 0
如果出错
进位 = 1
ah = 错误代码
1 = CRC 错误
2 = 丢失传送的数据
4 = 未发现数据（超时错误）
86h=没有执行这个功能

中断	功能	描述	平台
15h	3	向磁带写	PC

向磁带驱动器写入许多的 256 字节块。写入的字节数必须是 256 的倍数。这个功能早已过时，并且只有 PC 和 PCjr 才支持这个功能。

调用：
ah = 3
cx = 要写入的字节数
es:bx=要写入数据的缓冲区地址
cx = 0

返回：
es:bx=指针，指向上次写入的最后一个字节后的一个字节
如果没有出错
进位 = 0
如果出错
进位 = 1
ah = 错误代码
1 = CRC 错误
2 = 丢失数据跃迁
4 = 未发现数据（超时错误）
86h=没有执行这个功能

中断	功能	描述	平台
15h	4	建立系统参数表	PS/2

IBM 的 PS/2 带有另外一个 BIOS，内置于标准的 BIOS 中，通常称为 ABIOS。只有 OS/2 系统才使用它，当然在没有 ABIOS 的系统上，PS/2 也能够正常工作。这个功能在给用户提供的缓冲区里创建了一个系统参数表和一个指针。表 13-1 简要地列出了系统的参数表。参看 IBM 个人系统以及个人电脑 BIOS 接口技术参考，可以了解高级 BIOS（即 ABIOS）的其他细节。

表 13-1 ABIOS 系统参数表

16 进制偏移量	大 小	描 述
0	双字	通用起点程序指针
4	双字	通用中断程序指针
8	双字	通用超时程序指针
C	字	堆栈所要求的字节数
E	16 字节	IBM 保留
1E	字	初始化表所要求的入口数目（在中断 15h，功能 5 中有相关叙述）

调用：　ah=4

ds:0=指针，指向 RAM 扩展区（如果没有扩展 RAM 区，则 ds=0）

es:di=指针，指向给用户提供的 32 字节的缓冲区

返回：　如果没有出错

进位=0

ah=0

al=未定义

es:di=用户缓冲区，装入了 ABIOS 参数表（参看表 13-1）

如果出错

进位=1

ax=未定义

如果不支持这个功能

进位=1

ah=80h 或 86h

中断	功能	描述	平台
15h	5	建立初始化表	PS/2

在一个 PS/2 系统上，这个功能在给用户提供的表中装入了初始化数据和指针。仅 OS/2 才使用它。参看功能 4 了解缓冲区返回的入口数目，每个入口占 24 个字节。表 13-2 展示了初始化表的一个入口。

表 13-2　ABIOS 初始化表

偏移量（16 进制）	大　小	描　述
0	字	设备 ID
		0=ABIOS 内部调用
		1=软盘驱动器
		2=硬盘驱动器
		3=视频
		4=键盘
		5=并行口
		6=串行口
		7=系统时钟
		8=实时时钟
		9=系统服务

续表

偏移量（16进制）	大　小	描　述
		A=不可屏蔽中断
		B=鼠标
		E=CMOS RAM
		F=DMA
		10h=可编程选项选择（POS）
		16h=键盘密码
2	字	逻辑 ID 的数口
4	字	设备块的长度，单位是字节（0 代表 ABIOS 扩展，不需要设备驱动文件块）
6	双字	指向程序的指针，用来初始化设备块和功能传送表
A	字	请求块的长度，单位是字节
C	字	功能传送表的长度，单位是字节
E	字	数据指针的长度，单位是字节
10	字节	第二个设备 ID
11	字节	修正版号
12	6 个字节	未使用或未公开

调用：　ah=5

ds:0=指针，指向 RAM 扩展区（如果没有扩展 RAM 区，则 ds=0）

es:di=指针，指向给用户提供的缓冲区

返回：　如果没有出错

进位=0

ah=0

al=未定义

es:di=用户缓冲区，装入了 ABIOS 初始化表（参看表 13-2）

如果出错

进位=1

ax=未定义

如果不支持这个功能

进位=1

ah=80h 或 86h

中断	功能	描述	平台
15h	F	一般格式化	ESDI 控制器

在完成每个磁柱的格式化时，PS/2 ESDI 硬盘同调用中断 15h，AH=F。这一点支持程序通过挂起中断 15h 并监视这个功能来显示格式化的状态。这个功能还显示是正在进行表面分析还是在进行格式化，并且允许暂停磁盘格式化操作。

格式化程序先钩起中断 15h，功能 F，然后触发中断 13h 低级格式化 ESDI 驱动器命令。在调用中断 15h 功能 F 时，格式化程序会增加计数来跟踪刚格式化的磁柱号。依据格式化磁柱的总数和格式化程序所截获的功能 F 调用的次数，这个格式化程序可能会显示完成格式化的进度百分比。每次截获后，都会清除进位标志，以便继续格式化，或者设置标志为安全退出格式化模式。

其他的驱动器类型，包括大多数的非 ESDI 驱动器，没有提供实现一次格式化整个驱动器的功能，它们也不需要这种类型的接口。与大多数 BIOS 功能调用不同的是，这个功能不是由应用程序调用的。

调用：　ah=Fh (仅由硬盘 BIOS 调用)
　　al=1，如果要进行的是表面分析
　　2，如果要进行的是低级格式化
返回：　进位=0，继续格式化或表面分析
　　1，结束格式化或表面分析
　　如果不支持这个功能
　　进位=1
　　ah=80h 或 86h

中断	功能	描述	平台
15h	21h	错误日志读写	MCA

MCA 系统能够记录上电自检期间扩展 CMOS 数据区出现的错误。这个功能能够读或写错误日志。POST 错误代码通常和输出到诊断端口的值相同。参看本章结尾处的端口 680h 以了解典型的 POST 代码。记住，不同的型号和供应商可能分配不同的错误代码，这里没有统一的标准。目前只有 MCA 系统才支持这个功能。

子功能	描述	中断	功能
AL=0	读错误日志	15h	AH=21h

获取一个指针，指向错误日志列表的起点，并找出所记录的错误总数。每个错误字由高字节和低字节组成，其中高字节代表设备代码，低字节代表 POST 设备错误码。

调用：　ax=2100h
返回：　如果支持这个功能

进位=0

ah=0

bx=记录错误日志中的错误代码的入口数

es:di=指针，指向第一个错误字

如果不支持这个功能

进位=1

ah=80h 或 86h

子功能	描述	中断	功能
AL=1	写错误日志	15h	AH=21h

向错误日志中写入一个错误代码。

调用：ax=2101h

bl=设备码

bh=设备错误

返回：如果支持这个功能

进位=0

ah=0

如果支持这个功能，但是操作失败（日志已经满了）

进位=1

ah=1

如果不支持这个功能

进位=1

ah=80h 或 86h

中断	功能	描述	平台
15h	22h	获取未公开的段地址	PS/1

这个未公开的功能返回 BIOS 内的一个段地址。在 486 PS/1 上，这个值是 F774h，由段 F000 加上偏移量 7740h 得到。F7740：0 中包含了一个近跳转指令，跳到一个零值区域。可能不会完整地执行这个功能的代码，或者还会有其他的未知用途。BIOS 从来没有执行过这个功能。

调用：ah=22h

返回：如果支持这个功能

进位=0

es=段值（参看上文）

ax=0

如果不支持这个功能

进位=1

ah=80h 或 86h

中断	功能	描述	平台
15h	23h	多个未公开的功能	PS/1

这组功能出现在 IBM 486 PS/1 的 BIOS 中，可能还出现在最近的 IBM BIOS 中，一共有四个子功能。

AL=	子功能
0	读取 CMOS 寄存器 2D 和 2E
1	设置 CMOS 寄存器 2D 和 2E
4	系统设置
5	读取处理器速度

子功能	描述	中断	功能
AL=0	读取 CMOS 寄存器 2D 和 2E	15h	AH=23h

调用：ax=2300h

返回：如果支持这个功能

进位=0

ax=2300h

cl=CMOS 寄存器 2Dh（未曾公开过）

ch=CMOS 寄存器 2Eh（未曾公开过）

如果不支持这个功能

进位=1

ah=80h 或 86h

子功能	描述	中断	功能
AL=1	设置 CMOS 寄存器 2D 和 2E	15h	AH=23h

调用：ax=2301h

cl=CMOS 寄存器 2Dh（未曾公开过）

ch=CMOS 寄存器 2Eh（未曾公开过）

返回：　如果支持这个功能

ax=2301h

进位=0

如果不支持这个功能

进位=1

ah=80h 或 86h

子功能	描述	中断	功能
AL=4	系统设置	15h	AH=23h

启动系统设置操作。这个功能显示有关当前系统设置的信息，并且允许用户更改这些信息。如果用户保存了这些更改，会更新 CMOS 内存来反映这些新的设置。这个功能将强制视频模式为彩色文本，使用模式 3。退出时，再次触发视频模式 3 来清除屏幕。

调用：　ax=2304h

dx=供操作用的 32KB RAM 的段址

返回：　如果支持这个功能

进位=0

ax=3

bx、cx、dx、bp、ds 和 es 都做了修改

如果不支持这个功能

进位=1

ah=80h 或 86h

子功能	描述	中断	功能
AL=5	读取处理器速度	15h	AH=23h

返回经 POST 计算得到的处理器速度，并将它规范为标准的速度值。可能出现的速度包括：20、25、33、40、50、66 和 80MHz。如果没有测量速度或者速度太快，这个功能在返回时会设置进位标志。

调用：　ax=2305h

返回：　如果支持这个功能

进位=0

ah=23h

al=处理器速度，单位是兆赫兹

如果速度未知，或者速度超过 80MHz

进位=1

ah=23h

al=FFh

如果不支持这个功能

进位=1

ah=80h 或 86h

中断	功能	描述	平台
15h	24h	A20 控制	PS/1

这些功能控制 A20 门。开放 A20 门时，可以访问所有的扩展内存。禁止时，A20 地址线被强制为低电平来模拟 8080 地址映射。这组功能出现在 IBM 486 PS/1 的 BIOS 中，可能还出现在最近的 IBM BIOS 中。A20 控制一共有四个子功能。

AL=	子功能
0	禁止 A20
1	开放 A20
2	读取 A20 状态
3	未知功能

子功能	描述	中断	功能
AL=0	禁止 A20	15h	AH=24h

调用：ax=2400h

返回：如果支持这个功能

进位=0

ah=0

如果不支持这个功能

进位=1

ah=80h 或 86h

子功能	描述	中断	功能
AL=1	开放 A20	15h	AH=24h

调用：ax=2401h

返回：如果支持这个功能

进位=0

ah=0
如果不支持这个功能
进位=1
ah=80h 或 86h

子功能	描述	中断	功能
AL=2	读取 A20 门状态	15h	AH=24h

调用：ax=2402h
返回：如果支持这个功能
进位=0
ah=0
al=0，禁止 A20
1，开放 A20
如果不支持这个功能
进位=1
ah=80h 或 86h

子功能	描述	中断	功能
AL=3	未知功能	15h	AH=24h

在 *IBM 486 PS/1* 上，这个功能仅仅只是在 BX 中返回值 2。其他型号的机器可能会返回另外的硬代码值或执行其他操作。

调用：ax=2403h
返回：如果支持这个功能
进位=0
ah=0
bx=2
如果不支持这个功能
进位=1
ah=80h 或 86h

中断	功能	描述	平台
15h	40h	键盘控制器数据缓冲的读写	可选

这个功能读取或者设置 PC 可逆系统和调制解调器的数据组。其他的膝上型电脑供应商可能也会使用这个功能。其他的平台还没有公开过这个功能。大多数有关功能 40h 的信息来自于 IBM 的技术参考资料，这种参考做法目前相当时髦。由于只有过时的 PC 可逆类型才支持这个功能，所以我也将某些服务给搪塞了过去，而没有仔细严格地探究。功能 40h 有四个子功能。

AL=	膝上型电脑的子功能
0	读系统数据组
1	写系统数据组
2	读内部调制解调器数据组
3	写内部调制解调器数据组

子功能	描述	中断	功能
AL=0	读系统数据组	15h	AH=40h

调用：ax=4000h

返回：如果支持这个功能

进位=0

ah=0

bx=数据组信息 1

cx=数据组信息 2

如果不支持这个功能，或者操作失败

进位=1

ah=80h 或 86h

子功能	描述	中断	功能
AL=1	写系统数据组	15h	AH=40h

调用：ax=4001h

bx=数据组信息 1

cx=数据组信息 2

返回：如果支持这个功能

进位=0

ah=0

如果不支持这个功能，或者操作失败

进位=1

ah=80h 或 86h

子功能	描述	中断	功能
AL=2	读内部调制解调器数据组	15h	AH=40h

调用：ax=4002h

返回：如果支持这个功能

进位=0

ah=0

bx=调制解调器数据组信息

如果不支持这个功能，或者操作失败

进位=1

ah=80h 或 86h

子功能	描述	中断	功能
AL=3	写内部调制解调器数据组	15h	AH=40h

调用：ax=4003h

bx=调制解调器数据组信息

返回：如果支持这个功能

进位=0

ah=0

如果不支持这个功能，或者操作失败

进位=1

ah=80h 或 86h

中断	功能	描述	平台
15h	41h	膝上型电脑等待外部事件	可逆

这个功能等待一个事件的发生，并在发生指定的事件时或者在超过定时后返回。只有 PC 可逆系统才支持这个功能，而其他的膝上型电脑供应商可能也会使用这个功能。其他的平台还没有公开过这个功能。与功能 40h 一样，大多数有关功能 41h 的信息来自于 IBM 的技术参考资料。由于只有过时的 PC 可逆类型才支持这个功能，所以我也将某些服务给搪塞了过去，而没有仔细严格地探究。功能 41h 有十个子功能。

AL=	膝上型电脑的子功能
0	发生任意事件后返回
1	比较内存，相等时返回
2	比较内存，不相等时返回

3	测试内存位，非零时返回
4	测试内存位，为零时返回
10h	发生任意事件后返回
11h	比较端口，相等时返回
12h	比较端口，不相等时返回
13h	测试端口位，非零时返回
14h	测试端口位，为零时返回

子功能	描述	中断	功能
AL=0	发生了任意事件后返回	15h	AH=41h

调用：ax=4100h

bl=超时值，单位是 55ms 步进（0=无超时）

返回：如果支持这个功能，事件发生后返回

进位=0

如果支持这个功能，但是发生了超时

进位=1

如果不支持这个功能，或者操作失败

进位=1

ah=80h 或 86h

子功能	描述	中断	功能
AL=1	比较内存，相等时返回	15h	AH=41h

调用：ax=4101h

bh=比较值

bl=超时值，单位是 55ms 步进（0=无超时）

es:di=指向字节的指针——等待直到等于 BH

返回：如果支持这个功能，字节匹配 BH

进位=0

如果支持这个功能，但是发生了超时

进位=1

如果不支持这个功能，或者操作失败

进位=1

ah=80h 或 86h

子功能	描述	中断	功能
AL=2	比较内存，不相等时返回	15h	AH=41h

调用：
ax=4102h
bh=比较值
bl=超时值，单位是 55ms 步进（0=无超时）
es:di=指向字节的指针——等待直到不等于 BH

返回：
如果支持这个功能，字节不等于 BH
　进位=0
如果支持这个功能，但是发生了超时
　进位=1
如果不支持这个功能，或者操作失败
　进位=1
　ah=80h 或 86h

子功能	描述	中断	功能
AL=3	测试内存位，非零时返回	15h	AH=41h

调用：
ax=4103h
bh=屏蔽值
bl=超时值，单位是 55ms 步进（0=无超时）
es:di=指向字节的指针——等待直到值和 BH 相“与”操作后不等于零

返回：
如果支持这个功能，位不是零
　进位=0
如果支持这个功能，但是发生了超时
　进位=1
如果不支持这个功能，或者操作失败
　进位=1
　ah=80h 或 86h

子功能	描述	中断	功能
AL=4	测试内存位，为零时返回	15h	AH=41h

调用：
ax=4104h
bh=屏蔽值
bl=超时值，单位是 55ms 步进（0=无超时）

es:di=指向字节的指针——等待直到值和 BH 相“与”操作后等于零

返回：
如果支持这个功能，位是零
进位=0
如果支持这个功能，但是发生了超时
进位=1
如果不支持这个功能，或者操作失败
进位=1
ah=80h 或 86h

子功能	描述	中断	功能
AL=10h	发生了任意事件后返回	15h	AH=41h

这个功能和子功能 AL=0 相同。

子功能	描述	中断	功能
AL=11h	比较端口，相等时返回	15h	AH=41h

调用：
ax=4111h
bh=比较值
bl=超时值，单位是 55ms 步进（0=无超时）
dx=要读取的 I/O 端口号，读取这个端口直到等于 BH

返回：
如果支持这个功能，字节匹配 BH
进位=0
如果支持这个功能，但是发生了超时
进位=1
如果不支持这个功能，或者操作失败
进位=1
ah=80h 或 86h

子功能	描述	中断	功能
AL=12h	比较端口，不相等时返回	15h	AH=41h

调用：
ax=4112h
bh=比较值
bl=超时值，单位是 55ms 步进（0=无超时）
dx=要读取的 I/O 端口号，读取这个端口直到等于 BH

返回：
如果支持这个功能，字节不等于 BH
进位=0

如果支持这个功能，但是发生了超时
进位=1
如果不支持这个功能，或者操作失败
进位=1
ah=80h 或 86h

子功能	描述	中断	功能
AL=13h	测试端口位，非零时返回	15h	AH=41h

调用：
ax=4113h
bh=屏蔽值
bl=超时值，单位是 55ms 步进（0=无超时）
dx=要读取的 I/O 端口号，读取这个端口直到值和 BH 相“与”操作后不等于零

返回：
如果支持这个功能，位不是零
进位=0
如果支持这个功能，但是发生了超时
进位=1
如果不支持这个功能，或者操作失败
进位=1
ah=80h 或 86h

子功能	描述	中断	功能
AL=14h	测试端口位，为零时返回	15h	AH=41h

调用：
ax=4114h
bh=屏蔽值
bl=超时值，单位是 55ms 步进（0=无超时）
dx=要读取的 I/O 端口号，读取这个端口直到值和 BH 相“与”操作后等于零

返回：
如果支持这个功能，位是零
进位=0
如果支持这个功能，但是发生了超时
进位=1
如果不支持这个功能，或者操作失败
进位=1
ah=80h 或 86h

中断	功能	描述	平台
15h	42h	膝上型电脑的源电请求	可逆

要求膝上型电脑关掉电源。只有 PC 可逆系统才支持这个功能，其他的膝上型电脑供应商可能也会使用这个功能。其他的平台还没有公开过这个功能。

调用：ah=42h
al=0，使用系统数据组关闭电源
1，强制性关闭电源，忽略数据组

返回：如果支持这个功能
ax=未定义
如果不支持这个功能
进位=1
ah=80h 或 86h

中断	功能	描述	平台
15h	43h	读膝上型电脑的系统状态	可逆

读系统状态。只有 PC 可逆系统才支持这个功能，其他的膝上型电脑供应商可能也会使用这个功能。其他的平台还没有公开过这个功能。

调用：ah=43h

返回：如果支持这个功能
ah=未定义
al=状态

位		
	7 =1	电池电量不足
	6 =1	操作时使用外部电源
	5 =1	丢失备用电源，时钟错误
	4 =1	闹钟激活了电源
	3 =1	内部调制解调器上电
	2 =1	串行口和并行口上电
	1 =x	未使用
	0 =1	没有带 LCD

如果不支持这个功能
进位=1
ah=80h 或 86h

中断	功能	描述	平台
15h	44h	膝上型电脑调制解调器控制	可逆

控制内部调制解调器的电源。只有 PC 可逆系统才支持这个功能，其他的膝上型电脑供应商可能也会使用这个功能。其他的平台还没有公开过这个功能。

调用：ah=44h
al=调制解调器的电源状态
0=关闭了电源
1=开启了电源，并且基于调制解调器数据组进行设置

返回：如果支持这个功能
进位=0
ah=0
如果不支持这个功能，或者操作失败
进位=1
ah=80h 或 86h

中断	功能	描述	平台
15h	49h	获取 BIOS 类型	DBCS

只有在 BIOS 或操作系统支持双字字节字符集（DBCS）时，才可以使用这个功能。例如，DOS/V 在标准的 AT 系统或 MCA 系统上才支持 DBCS。即使 DOS/V 设置为英文环境，这个功能也可以发挥作用。

调用：ax=4900h

返回：如果支持这个功能
进位=0
ah=0
bl=BIOS 模式
0=DOS/V
1=普通的 DBCS DOS BIOS（支持硬件 DBCS）
如果不支持这个功能
进位=1
ah=80h 或 86h

中断	功能	描述	平台
15h	4Fh	键盘截获	AT+

从 AT 开始，IBM 的工程师们在他们设计的系统中加入了一种功能，这种功能可以钩起低级键盘流。在中断 9 键盘 BIOS 程序读取一个键盘扫描码之后，BIOS 立即调用中断 15h，AH=4Fh。AL 寄存器中保存读入的键盘扫描码，同时设置进位标志来表示键有效。多个应用程序可以钩起中断 15h，并搜索功能 4Fh。如果发生了一个重要的组合键，那么应用程序会采取下列操作之一：

1. 忽略这个键，将键传给其他钩起这个中断的应用程序，然后回到 BIOS 键盘处理程序。进位标志保持不变。
2. 丢掉这个键，就好像从未出现过这个键操作一样。这一点对于 TSR 热键非常有用。TSR 在其内部记录热键发生情况，这样 TSR 就可以执行一个后续操作。TSR 清除标志位指示不再需要处理键。其他钩起中断 15h 的应用程序以及键盘 BIOS 服务程序会忽略这个键。仍由键盘 BIOS 负责清除中断请求。
3. 用一个完全不同的键盘扫描码替代出现的键。这时将改变 AL 中的值，但是进位标志保持不变。任意在这个应用程序之后钩起中断 15h 的应用程序以及 BIOS 键盘处理程序会只看到新的键。记住，改变后的键值必须是扫描码值，而不是 ASCII 形式的值。

为了知道低级键盘程序是否支持这个功能，可参看中断 15h，功能 C0h，读取特性信息字节 5。如果设置了位 4，那么键盘 BIOS 支持 4Fh 键盘截获这个功能。处理这个功能的系统 BIOS 仅仅只是发一个 IRET 命令，而不对寄存器或进位标志产生任何影响。

调用：　ah=4Fh（仅由中断 9 调用）
　　al=键盘扫描码值
　　进位=1，键有效
　　0，键无效
返回：　al=相同的或改变后的扫描码
　　进位=1，让键盘 BIOS 处理一个键
　　0，让这个键消失
　　如果不支持这个功能
　　进位=1
　　ah=80h 或 86h

中断	功能	描述	平台
15h	50h	访问字体子功能	仅 DOS/V

这个功能控制 DOS/V 字体系统。只有安装了 DOS/V 后才可以使用这个功能。这个功能有两个子功能。

子功能	描述	中断	功能
AL=0	获取汉字体功能地址	15h	AH=50h

调用：
ax=5000h
bx=字体类型
0=单字节字符集
1=双字节字符集
bl=0
dh=字体格宽度
dl=字体格高度
bp=代码页
0 = 缺省代码页
437 = 美国英语
932 = 日本
934 = 韩国
936 = 中国
938 = 台湾

返回：
如果支持这个功能，并且成功
进位=0
ah=0
es:bx=指针，指向读字体功能
如果功能失败
进位=1
ah=错误代码
1=BH 字体类型不是 0 或 1
2=BL 不是 0
3=DX 字体大小无效
4=BP 代码页无效
80h=不支持这个功能（PC）
86h=不支持这个功能（XT）

子功能	描述	中断	功能
AL=1	获取字体功能地址	15h	AH=50h

调用：
ax=5001h
bx=字体类型
0=单字节字符集
1=双字节字符集
bl=0
dh=字体格宽度
dl=字体格高度
bp=代码页

0 = 缺省代码页
437 = 美国英语
932 = 日本
934 = 韩国
936 = 中国
938 = 台湾

返回：　如果支持这个功能，并且成功
进位=0
ah=0
es:bx=指针，指向写字体功能
如果功能失败
进位=1
ah=错误代码
1=BH 字体类型不是 0 或 1
2=BL 不是 0
3=DX 字体大小无效
4=BP 代码页无效
80h=不支持这个功能（PC）
86h=不支持这个功能（XT）

中断	功能	描述	平台
15h	52h	可移走媒介质弹片	EBIOS

在触发中断 13h 功能 46h（弹出可移走媒介质）时，硬盘 EBIOS 触发这个功能。不能由应用程序调用这个功能。参看第 11 章“硬盘系统”以了解完整细节。

中断	功能	描述	平台
15h	80h	设备开	AT+

设备开功能是应用程序和多任务操作系统（如 OS/2）之间协议的一部分。由应用程序触发这个功能。设备开向 OS 发出信号，指示这个调用程序要使用这个设备。

在安装了某个操作系统之后，操作系统会钩起这个功能，这样应用程序就不能请求这个命令。如果操作系统没有钩起这个功能，系统会返回成功或者失败信息，这与具体的 BIOS 有关。BIOS 不执行任何操作，仅仅只是返回而已。IBM 定义了下述多任务环境的使用规范。

调用：　ah=80h
bx=设备 ID

cx=进程 ID

返回：如果成功或者不支持（AT+）

进位=0

ah=0

如果不支持这个功能（PC/XT）

进位=1

ah=80h 或 86h

中断	功能	描述	平台
15h	81h	设备关	AT+

设备关功能是应用程序和多任务操作系统（如 OS/2）之间协议的一部分。由应用程序触发这个功能。设备关向 OS 发出信号，指示不再需要这个设备。

在安装了某个操作系统之后，操作系统会钩起这个功能，这样应用程序就不能请求这个命令。如果操作系统没有钩起这个功能，系统会返回成功或者失败信息，这与具体的 BIOS 有关。BIOS 不执行任何操作，仅仅只是返回而已。IBM 定义了下述多任务环境的使用规范。

调用：ah=81h

bx=设备 ID

cx=进程 ID

返回：如果成功或者不支持（AT+）

进位=0

ah=0

如果不支持这个功能（PC/XT）

进位=1

ah=80h 或 86h

中断	功能	描述	平台
15h	82h	程序中止	AT+

这个程序中止功能是应用程序和多任务操作系统（如 OS/2）之间协议的一部分。由应用程序触发这个功能。程序中止向 OS 发出信号，指示应用程序完成了所有的设备操作，现在可以关闭所有的设备。参看功能 80h 和 81h 获取其他相关细节。

在安装了某个操作系统之后，操作系统会钩起这个功能，这样应用程序就不能请求程序中止命令。如果操作系统没有钩起这个功能，系统会返回成功或者失败信息，这与具体的 BIOS 有关。BIOS 不执行任何操作，仅仅只是返回而已。IBM 定义了下述多任务环境的使用规范。

调用：　ah=82h
　　cx=进程 ID
返回：　如果成功或者不支持（AT+）
　　进位=0
　　ah=0
　　如果不支持这个功能（PC/XT）
　　进位=1
　　ah=80h 或 86h

中断	功能	描述	平台
15h	83h	事件等待	AT+

在超过指定时限时，使用这个功能在用户内存中设置一个标志。在触发这个功能之后立即返回到应用程序。有两个子功能来开启或取消事件等待。

子功能	描述	中断	功能
AL=0	启动事件等待	15h	AH=83h

这个子功能检查是否可以使用事件等待。如果可以，则设置 CMOS RTS，以每 976ms 触发一次中断 70h。在 BIOS 数据区 40:9Ch 处保存了一个 32 位的递减计数器时钟。用户标志字节的指针保存在 40:98h 中。事件的在用等待标志位是 40:40h 的位 0。这些 BIOS 数据项共享功能 86h（等待），因此一次只允许一个功能。在用标志对程序起保护作用，以免一个应用例程覆盖了另一个应用例程。

触发了这个子功能后，中断 70h 每 976ms 递减 32 位计数器一次。如果计数到零，则在一个字节地址处设置事件过期标志位（第 7 位），由所提供的指针指向这个字节地址。在触发这个功能之前应用程序负责清除事件过期标志位。

某些老式的 PS/2 AT 系统不支持这个功能，就像这个功能总在被其他程序使用一样。

调用：　ax=8300h
　　cx:dx=32 位微秒计数器（每 976ms 递减 976）
　　es:bx=远指针，指向用户内存中的字节
返回：　如果成功
　　进位=0
　　ah=0
　　al=CMOS 寄存器 B 的当前内容（参看第十五章）
　　如果在用这个功能，因此其他应用程序不能使用这个功能
　　进位=1
　　ax=0

如果不支持这个功能（PC/XT）
进位=1
ah=80h 或 86h

子功能	描述	中断	功能
AL=1	取消事件等待	15h	AH=83h

取消当前的事件等待。在普通操作情况下完成事件等待时，它自动为下一个事件准备好。这个功能通常用来清除还未到期的事件等待。这个功能会关闭由 CMOS RTS 生成的周期性中断，并清除 BIOS 数据区 40:A0h 处正在执行的事件。只有由调用程序触发的事件等待才可以被取消。如果从另外一个应用程序取消事件等待，那么可能会挂起这个程序。

最早期的 AT BIOS 不支持这个功能，某些 PS/2 AT 总线也不支持这个子功能。

调用：ax=8301h
返回：如果成功
进位=0
如果不支持这个功能（PC/XT）
进位=1
ah=80h 或 86h

中断	功能	描述	平台
15h	84h	游戏杆支持	AT+

这个功能读取一些开关以及不超过四个的游戏杆的模拟位置。对于两个游戏杆来说，每个游戏杆支持 X 和 Y 位置。对于四个游戏杆来说，每个游戏杆只支持一个单独的 X 位置。BIOS 不检查是否带有游戏杆适配器或是否真正的带有游戏杆。

子功能	描述	中断	功能
AL=0	读游戏杆开关	15h	AH=84h

这个子功能简单地从游戏杆端口 201h 输入一个字节，并去掉低四位。剩下的就是不超过四位的开关状态。开关的数目取决于所使用的游戏杆的具体情况。

调用：ah=84
dx=0
返回：如果成功

进位=0

al=开关状态

位		
	7=x	游戏杆 B，开关 2
	6=x	游戏杆 B，开关 1
	5=x	游戏杆 A，开关 2
	4=x	游戏杆 A，开关 1
	3=0	
	2=0	
	0=0	
	1=0	

如果不支持这个功能（PC/XT）

进位=1

ah=80h 或 86h

子功能	描述	中断	功能
AL=1	读游戏杆位置	15h	AH=84h

检查游戏杆的四种可能的模拟输入的位置。每个模拟控制有一个电阻连接着游戏杆适配器。为了便于度量这个位置，电阻还控制着一个时钟，在不同的位置返回不同的时间延迟。通过测量时间延迟可以确定位置。所执行的操作还包括在对每个游戏杆模拟位置计时的同时暂时禁止中断。

调用：

ah=84

dx=1

返回：

如果成功

进位=0

ax=游戏杆 A，X 坐标值

bx=游戏杆 A，Y 坐标值

cx=游戏杆 B，X 坐标值

dx=游戏杆 B，Y 坐标值

如果不支持这个功能（PC/XT）

进位=1

ah=80h 或 86h

中断	功能	描述	平台
15h	85h	改变系统请求键	AT+

在按下系统请求键时，低级中断 9 键盘处理程序会触发这个功能。应用程序可以钩起这个功能，以便在任何时候系统请求键状态发生改变时通知应用程序。它不应由应用程序调用。调用这个功能时，缺省的系统 BIOS 中断 15h 处理程序不执行任何操作。

调用：　ah=85h
　　al=0，现在正按下系统请求键
　　1，现在释放了系统请求键
返回：　如果被执行
　　进位=0
　　ah=0
　　如果不支持这个功能（PC/XT）
　　进位=1
　　ah=80h 或 86h

中断	功能	描述	平台
15h	86h	等待	AT+

等待，直到指定的时间周期期满。调用程序不会重新获得控制直到等待周期期满。参看功能 83h（事件等待），了解另外一种可替代的方法。等待时间周期的微秒值保存在一个 32 位的计数器中。

触发这个命令时，系统检查是否正在执行来自这个功能或功能 83h 的等待操作。如果有，这个功能仅仅只是返回而不进行等待。如果允许等待，则它的操作和功能 83h 相同，只是系统 BIOS 的事件等待期限保存在地址 40:A0h 处。这个功能在一个紧密的循环中等待，直到设置了位 7（即时间期限到期），然后返回到用户程序。

这个等待功能的分辨率是 976 微秒，这意味着计数器每 976 微秒会递减 976，直到计数值小于零。某些 BIOS 服务可能会调用这个功能来执行操作定时，比如软盘磁头定位。如果已经在使用等待功能，则这些内部 BIOS 程序可以切换到软件定时循环下。

调用：　ah=86h
　　cx:dx=32 位计数器，单位微秒
返回：　如果成功
　　进位=0
　　ah=0
　　如果这个功能正在使用，因此不能被其他应用程序使用
　　进位=1
　　ah=0
　　al=未定义

如果不支持这个功能（PC/XT）
进位=1
ah=80h 或 86h

中断	功能	描述	平台
15h	87h	访问扩展内存	AT+

这个功能可以在前 1MB 的地址范围和扩展内存之间传送不超过 64K 的数据。在 386 以及后来的系统上，这个功能允许传送数据的最大字数为 128K 减 1。在 286 系统上，这个功能可能非常慢，因为系统必须首先进入保护模式，为了回到实模式，CPU 必须执行一次非常缓慢的重启操作。记住，在块移动时会禁止中断，并且可能会维持中断的禁止状态很长一段时间。

今天大多数程序应该使用老式的 EMS 标准或者新式的 XMS 内存标准，以便和其他应用程序能够可靠地共存。功能 87h 对扩展内存的访问是不可控制和管理的，需要访问扩展内存的两个应用程序可能会很容易地破坏彼此保存在扩展内存中的数据！如果正在使用内存管理程序，并且已经分配了扩展内存，则这个功能可能还会造成系统冲突。大多数内存管理程序会阻止操作这个功能。

由于你现在只是处于阅读阶段，所以你可能非常大胆，而对我上面提出的忠告不予理会，或者你仅仅只是对这个功能到底是如何工作的感兴趣。这个功能比其他的 BIOS 服务都要复杂。使用一组 CPU 可理解的描述符来描述内存的源和目标，如表 13-3 所示。你还必须知道 CPU 是 286 还是 386，或者是更晚的 CPU，因为 CPU 家族系列之间的描述符会有所不同。在功能的结尾处提供了一个实代码例。

表 13-3 扩展内存访问表的结构

es:di 偏移量	描　　述
0	未使用的描述符（必须全部设置为零）
8	GDT 描述符（必须全部设置为零，由 BIOS 填入）
10h	源描述符，数据的来源地址
18h	目标描述符，放置数据的地址
20h	代码描述符（必须全部设置为零，由 BIOS 填入）
28h	堆栈描述符（必须全部设置为零，由 BIOS 填入）

每个描述符都包含了访问的限制条件、要使用的地址，并用一个字节指示了如何访问这部分内存。段址限制描述了描述符可以访问的内存大小。在 286 上，值 FFFFh 将设置为上限，64K。在 386 以及后来的 CPU 上，上限设置为值 1FFFFh，表示 128K。有两个描述

符用来指示内存源地址和目标地址。这些都展示在表 13-4 和表 13-5 中，具体情况与系统的 CPU 有关。

表 13-4 286 描述符，扩展内存访问

偏移量	大 小	描 述
0	字	段址限制——字节数减 1。使用值 FFFFh 来访问 64K
2	字	24 位物理地址的低 16 位部分
4	字节	24 位物理地址的高 8 位部分
5	字节	访问权限字节，93h=可读/可写
6	字	未使用（必须为零）

表 13-5 386 和后来的系统的描述符，扩展内存访问

偏移量	大 小	描 述
0	字	段址限制——20 位限制的低 16 位——字节数减 1
2	字	32 位物理地址的低 16 位部分
4	字节	32 位物理地址的中间 8 位部分
5	字节	访问权限字节，93h=可读/可写
6	字节	低四位是 20 位段址限制的高四位。高四位是标志位，对于这个功能，应当设置为零
7	字节	32 位物理地址的高 8 位部分

调用：

ah=87h

cx =要传送的字节数

es:si=指针，指向 48 字节的描述符表（参看表 13-3）

返回：

如果成功

进位=0

ah=0

如果功能操作失败

进位=1

ah=错误代码

1=出现内存奇偶错误

2=出现了某些其他的异常中断

3=开启地址线 A20（通常称作 A20 门）的操作失败
如果不支持这个功能（PC/XT）
进位=1
ah=80h 或 86h

比如，某个程序可能希望将 32K 的显示区保存到起始于 4MB 地址处的扩展内存中。我假定程序已经使用功能 88h 检查了是否存在所期望的扩展内存，并且检查出 CPU 的类型——是 386 还是后来的 CPU。

```
        mov     ah,87h                  ; 访问扩展内存
        mov     cx,4000h                ; 传送 32x(16K 字)
        push    cs
        pop     es
        mov     si,offset GDTdata       ;es:si=指向表的指针
        int     15h                     ; 执行 BIOS
        jc      failure
        .                               ; 如果成功则运行到这里
        .
failure:
        .                               ; 如果有问题则运行到这里

; 供用户使用的描述符表
GDTdata dw      4 dup (0)               ; 未使用的描述符
        dw      4 dup (0)               ;GDT 描述符

; 源描述符的起始处 (C8000h)
        dw      OFFFFh                  ; 段限制, 20 位值= 1FFFFh
        dw      08000h                  ; 源, 32 位值= C8000h
        db      OCh                     ; 源, 中间 8 位
        db      93h                     ; 访问权限, 读/写
        db      1                       ; 标志和段顶限制
        db      0                       ; 源, 首 8 位

; 目的描述符的起始处 (4000000h)
        dw      OFFFFh                  ; 段限制, 20 位值= 1FFFFh
        dw      0                       ; 源, 32 位值= 400000h
```

```
        db      40h                     ; 源，中间8位
        db      93h                     ; 访问权限，读/写
        db      1                       ; 标志和段顶限制
        db      0                       ; 源，首8位

; 表的其余部分
        dw      4 dup (0)               ; 代码描述符
        dw      4 dup (0)               ; 堆栈描述符
```

中断	功能	描述	平台
15h	88h	扩展内存大小	AT+

这个功能获取扩展内存的总大小，单位是 1K 块。如果没有应用程序钩起这个功能，大多数 BIOS 会返回保存在 CMOS 寄存器 30 和 31h 中的值。IBM 的 PS/2 和 PS/1 使用 CMOS 寄存器 35 或 36h 来保存系统的扩展内存大小。

使用扩展内存的程序，例如内存管理程序，会将它们所使用的内存大小减去这个值。大多数内存管理程序会消耗掉所有的扩展内存，并且（除非带有某些其他的开关指令）会强制这个功能返回一个零值。

调用：　ah=88h

返回：　如果成功

　　进位=0

　　ax=扩展内存块（1K 块）的数目

　如果不支持这个功能（286 以前的系统）

　　进位=1

　　ah=80h 或 86h

中断	功能	描述	平台
15h	89h	切换到保护模式	AT+

这个功能将 CPU 切换到保护模式，一个 64 字节的描述符表提供了相关的切换信息。在使用这个功能之前，你必须熟悉保护模式、描述符以及其他与 CPU 相关的特性。在执行这个功能之前用户必须定义好描述符表。描述符表如表 13-6 所示。记住，286 和 386 以及后来的系统具有不同的描述符格式。在 386~Pentium Pro 的 CPU 上，FS 和 GS 段保持无效，直到用户在进入保护模式后装入了这个描述符表为止。

在保护模式下不提供任何一种系统 BIOS 服务，这意味着用户的代码必须处理所有的硬件中断。为了实现这一点，BIOS 在控制器中重新装入由用户提供的新的中断号。这些中

断号作为用户提供的中断描述符表（IDT）的索引。两个控制器都要进行设置，以屏蔽掉所有的硬件中断。另外，用户的保护模式中断描述符表不能覆盖实模式中断表。

这个 BIOS 功能会激活地址线 A20，即 A20 门。POST 完成后，A20 线会锁定为零，这样系统会像 8080 那样操作。激活 A20 门这一点意味着 CPU 可以访问系统的全部内存地址。

一旦激活了 A20 并设置了中断控制器，BIOS 将会为它自己的代码设置 BIOS CS 描述符。BIOS 装入全局描述符表寄存器（GDTR）和中断描述符表寄存器（IDTR）。然后 BIOS 会激活保护模式。在保护模式下，BIOS 装入用户的数据段址，最后返回到保护模式下的调用程序中。

表 13-6　保护模式 GDT

es:si 偏移量	描　　述
0	未使用的描述符（必须全部设置为零）
8	全局描述符表（GDT）描述符（由用户提供）
10h	中断描述符表（IDT）描述符（由用户提供）
18h	DS 描述符（由用户提供）
20h	ES 描述符（由用户提供）
28h	CS 描述符（由用户提供）
30h	SS 描述符（由用户提供）
38h	临时 BIOS CS 描述符（由 BIOS 提供，忽略用户提供的值）

调用：　ah=89h

bh=硬件中断请求 IRQ0~7 块的第一个中断号

bl=硬件中断请求 IRQ8~15 块的第一个中断号

es:si=指针，指向 64 字节的描述符表（参看表 13-6）

返回：　如果成功，并且位于保护模式下

进位=0

ah=0

al=未定义

bp=未定义

ds=18h

es=20h

ss=28h

cs=30h

如果这个功能开锁 A20 门失败，并且仍停留在实模式下

进位=1

ah=FFh

al=未定义

如果不支持这个功能（286以前的系统）

进位=1

ah=80h 或 86h

中断	功能	描述	平台
15h	90h	设备忙	AT+

许多 BIOS 程序在开始一个操作后不会返回，直到这个操作完成才返回。这个功能提供了一个挂钩，来让其他程序或多任务操作系统在等待设备操作完成期间获得控制。只有在超时周期期满或者设备操作完成时，这个功能才返回。如果没有钩起这个设备忙功能，系统 BIOS 只会简单返回，而让调用程序等待超时或者设备操作完成。这个功能不应由应用程序调用，而应由系统 BIOS 代码或者硬件设备驱动程序来调用。

例如，软盘 BIOS 可以启动一个来自软盘的数据传送，然后等待控制器在传送完成时中断系统。这可能要花 2 秒钟，这样就可以触发中断 15h 功能 90h。如果没有钩起这个中断，系统 BIOS 会简单地返回并清除进位标志，代表软盘 BIOS 必须等待直到超时期限期满或者操作完成。

假定一个多任务操作系统钩起了中断 15h 的功能 90h 和 91h。在出现中断 15h 的功能 90h 时，操作系统会获得控制并查看 AL 中传递的值。在这个例子中，刚刚调用了软盘中断。软盘设备类型是 1，操作系统仍可以花几秒钟来运行其他任务。在一般情况下，出现软盘服务中断 2 秒钟后，软盘硬件中断处理程序会触发中断 15h 的功能 91h，设备码为 1。系统监测到这个事件，并记录下现在完成的软盘服务功能。系统从中断 15h 的功能 91h 中返回，于是中断处理程序就可以完成其操作。如果方便，操作系统会返回到中断 15h 功能 90h 的调用程序中，并且设置进位标志，以指示软盘 BIOS 不必继续等待。

表 13-7 定义了下述三类设备类型：

- 第 1 类：不可重入设备，一次只有一个用户可以访问这种设备。这些设备使用的设备类型值为 0~7Fh。所有的这些设备都带有一个硬件中断处理程序，在设备操作完成时会触发中断 15h 的功能 91h。
- 第 2 类：可重入设备，这些设备可以被多次调用。这些设备使用的设备类型值为 80~BFh。所有的这些设备都带有一个硬件中断处理程序，在设备操作完成时会触发中断 15h 的功能 91h。另外，必须在 ES:BX 中提供一个设备控制块，来唯一标识正在处理设备的那个例程。由操作系统指定设备控制块的格式。系统 BIOS 不带任何第 2 类的设备。
- 第 3 类：这些设备没有为等待功能提供硬件中断，它们必须简单地等待一个时钟周期。例如在开启软盘马达时，软盘 BIOS 必须等待 2 秒钟，以等待马达在写操作时达到全速。

表 13-7 设备类型码

设备号	BIOS 中断	设备及中断源	是否有超时期限
0	13h	硬盘 BIOS，等待数据	是
1	13h	软盘 BIOS，等待数据	是
2	16h	键盘 BIOS，等待一个键	否
3		鼠标	是
80h		网络	否
FCh	13h	硬盘 BIOS，重启	是
FDh	13h	硬盘 BIOS，启动马达	是
FEh	17h	打印机 BIOS，打印忙	是

调用：　ah=90h（仅由 BIOS 或设备驱动文件调用）
al=设备号（参看表 13-7）
es:bx=指向设备控制块的指针，如果 al=80h~BFh

返回：　如果还没有满足等待条件
进位=0
如果设备完成操作，或者定时到期
进位=1
如果不支持这个功能（286 以前的系统）
进位=1
ah=80h 或 86h

中断	功能	描述	平台
15h	91h	中断完成	AT+

硬件设备触发一个中断指示设备操作完成。参看功能 90h 了解有关如何使用这个功能的细节。这个功能不应由应用程序调用，而应由系统 BIOS 代码或者某些硬件设备驱动程序来调用。表 13-8 列出了当前有定义的设备码。其他的设备码必须服从功能 90h 中所描述的三种类型。

表 13-8 设备类型码

设备号	BIOS 中断	设　　备
0	76h	硬盘操作完成中断
1	Eh	软盘操作完成中断
2	9	键盘 BIOS，有键按下
3		鼠标中断
80h		网络中断

调用：　ah=91h（仅由 BIOS 或设备驱动文件调用）
al=设备号（参看表 13-8）
es:bx=指向设备控制块的指针，如果 al=80h~BFh

返回：　如果不支持这个功能
ah=0
进位=0
如果不支持这个功能（286 以前的系统）
进位=1
ah=80h 或 86h

中断	功能	描述	平台
15h	C0h	返回系统的 BIOS 设置	部分 XT、AT+

获取指向系统 BIOS ROM 内配置信息的指针，这个信息常常保存在 F000：E6F5h 中，但是从 PS/2 系统开始，IBM 不再为这个表保留一个固定的地址。这个表通常长为 10 个字节。表 13-9 展示了配置表的标准内容。尽管在老式的系统上某些域可能有其他的值，但是这个表仍只列出了 AT 和后来系统上可能有的值，因为在老式的系统不支持这个功能。

表 13-9　BIOS ROM 配置信息

偏移量	大　小	描　述		
0	字	位于这个字后面的表的大小，单位是字节		
2	字节	型号字节		
		F8=PS/1、PS/2 及它们的复制类型		
		F9=PC 可逆类型		
		FA=PS/2 的 25 和 30 型		
		FC=AT、286XT 以及 286~Pentium 复制类型的大部分		
3	字节	子型号字节——因型号和供应商的不同有很大的差别		
4	字节	BIOS 改进版号（0 表示首次发布，但是并非所有的供应商都支持）		
5	字节	特性信息 1		
		位	7=0	硬盘没有使用 DMA 通道 3
			1	硬盘使用了 DMA 通道 3
			6=1	有两个中断控制器
			5=1	有实时时钟
			4=1	在每次收到键时，键盘 BIOS 调用中断 15h 的功能 4Fh
			3=0	不支持外部事件等待

续表

偏移量	大　　小	描　　述		
			1	支持外部事件等待
			2=0	没有使用扩展 BIOS 数据区
			1	可能使用了扩展 BIOS 数据区（通常就位于主存顶部下的 1-4K
			1=0	AT 或 ESDI 总线
			1	微通道总线
			0=0	未使用
6	字节	特性信息 2		
		位	7=0	未使用
			6=0	不支持中断 16h 的功能 9
			1	支持中断 16h 的功能 9
			5=0	未使用或未知功能
			4=0	未使用或未知功能
			3=0	未使用或未知功能（486 PS/1 将它设置 1）
			2=0	未使用或未知功能
			1=0	未使用或未知功能
			0=0	未使用或未知功能
7	字节	特性信息 3		
		位	7=0	未使用或未知功能
			6=0	未使用或未知功能
			5=0	未使用或未知功能
			4=0	未使用或未知功能（486 PS/1 将它设置 1）
			3=0	未使用或未知功能
			2=0	未使用或未知功能
			1=0	未使用或未知功能
			0=0	未使用或未知功能
8	字节	特性信息 4（未使用，设置为零）		
9	字节	特性信息 5（未使用，设置为零）		

调用：　　ah=C0h

返回：　　如果支持这个功能

　　ah=0

进位=0

es:bx=指针，指向配置表（参看表 13-9）

如果不支持这个功能或者不能确定

进位=1

ah=80h 或 86h

中断	功能	描述	平台
15h	C1h	返回扩展 BIOS 数据区	AT+

获取扩展 BIOS 数据区的段址。扩展 BIOS 数据区用于鼠标信息、额外的硬盘数据以及其他的用途。需要使用扩展 BIOS 数据区的 BIOS 从主存区的顶部占用内存，以 1K 为单位。大多数需要扩展 BIOS 数据区的 BIOS 只使用 1K 的主存，当然我也见过一些使用 4K 的例子。在执行这个功能之前，必须检查偏移量 5 处的特性字节的第 2 位，来查看是否使用扩展 BIOS 数据区。

大多数比较成功的内存管理程序会除去或者将扩展 BIOS 移动到高端内存区或者主存的底部。扩展 BIOS 数据区的段址保存在 BIOS 数据区地址 40:Eh 处。参看第 6 章以了解扩展 BIOS 数据区的内容。

调用：ah=C1h

返回：如果支持这个功能

ah=0

es=扩展 BIOS 数据区的段址

如果不支持这个功能或者没有扩展 BIOS 数据区

进位=1

ah=80h 或 86h

中断	功能	描述	平台
15h	C2h	鼠标 BIOS 接口	PS/2

PS/2 以及后来的 IBM 产品提供了一个内置鼠标端口，通过这里指出的八个服务来访问它。大多数带有内置鼠标端口的复制类型也使用这些功能。参看第 8 章以详细了解连接到键盘控制器的鼠标接口。这些功能应由鼠标设备驱动程序使用，而不应由应用程序使用。应用程序应该使用中断 35h 鼠标设备驱动程序来处理鼠标事件。

某些技术文档使用更常见的术语，即所谓的点设备。它包括鼠标、滚动球以及其他这种类型的设备。

依我看来，这些接口方法的功能相当薄弱。如果安装了鼠标，即使没有发生鼠标动作，也会发送连续的中断。在缺省的采样速率下，每秒大约生成 500 个中断。在大多数时间内

鼠标处于空闲状态。这意味着高中断率不必要地增加了系统的负担，从而减慢了总的运行速度，特别是在保护模式下更是如此。

在调用一个鼠标子功能时，会在 AH 中返回相应的状态，如表 13-10 所示。

表 13-10 鼠标状态返回码

值	描 述
0	操作成功，没有出错
1	无效的子功能调用（即，AL 大于 7）
2	无效的输入值（即，越出了允许的范围）
3	接口错误
4	从鼠标控制器收到了重新发送命令，设备驱动程序应该重试这个命令。
5	不能支持鼠标，因为没有安装远程调用
80h	没有执行鼠标服务（PC）
86h	没有执行鼠标服务

鼠标 BIOS 接口提供了下面十一个服务：

AL=	鼠标子功能
0，BH=0	禁止鼠标
0，BH=1	开放鼠标
1	重启鼠标
2	设置采样速率
3	设置分辨率
4	读取设备 ID
5	初始化鼠标
6，BH=0	返回状态
6，BH=1	比例因子设置为 1:1
6，BH=2	比例因子设置为 2:1
7	设置鼠标处理程序的地址

子功能	描述	中断	功能
AL=0，BH=0	禁止鼠标	15h	AH=C2h

向键盘/鼠标控制器发送一条命令来禁止鼠标。

调用：　ax=C200h
　　bl=0

返回：　如果成功

进位=0

ah=0

如果没有执行这个功能

进位=1

ah=80h 或 86h

子功能	描述	中断	功能
AL=0、BH=1	开放鼠标	15h	AH=C2h

向键盘/鼠标控制器发送一条命令来开放鼠标。如果没有使用功能 C207h 来安装远程调用指针，则不会开放鼠标。这时错误状态会设定为 5。

调用：ax=C200h

bl=1

返回：如果成功

进位=0

ah=0

如果没有执行这个功能

进位=1

ah=返回状态（参看表 13-10）

子功能	描述	中断	功能
AL=1	重启鼠标	15h	AH=C2h

重启鼠标并初始化值。这个功能还在 BX 中返回鼠标的设备 ID，但是 IBM 在其文档中指出 BL 并不重要。重启后会禁止初始化鼠标的状态，而采样率设置为每秒报告 100 次，100dpi 的分辨率以及 1:1 的比例。这个功能与功能 C205 相同，只是在使用这个功能时不改变数据包。参看功能 C205h 以了解有关数据包大小的更多信息。

调用：ax=C201h

返回：如果成功

进位=0

ah=0

bh=0（设备 ID）

bl=x（设备 ID 的低字节，无效）

如果功能失败

进位=1

ah=返回状态（参看表 13-10）

子功能	描述	中断	功能
AL=2	设置采样率	15h	AH=C2h

设置鼠标的报告位置和按钮状态信息的速率。鼠标生成的中断数目大约等于采样率乘以数据包的大小。比较大的采样率和/或数据包会生成比较频繁的中断率，因此会影响系统的总运行速度，特别是在保护模式或 V86 模式下更是如此。

尽管系统 BIOS 限制了鼠标接口可以处理的采样率的范围，但是这个接口仍然可以处理任意一个 1~255 次报告/秒的采样率。不幸的是，需要对键盘/鼠标控制器直接进行编程。参看第 8 章端口 60h 的命令 F3h，以了解低层次的细节。

调用：
ax=C202h
bh=采样率的取值
0= 10 次报告/秒
1= 20 次报告/秒
2= 40 次报告/秒
3= 60 次报告/秒
4= 80 次报告/秒
5=100 次报告/秒
6=200 次报告/秒

返回：
如果成功
进位=0
ah=0
如果功能失败
进位=1
ah=返回状态（参看表 13-10）

子功能	描述	中断	功能
AL=3	设置分辨率	15h	AH=C2h

设置鼠标的分辨率。对于一个给定的鼠标运动来说，比较低的数值会使鼠标在屏幕上移动得更快，比较高的数值可以提供更精确的控制。

要对这个值直接进行编程，可以参看第 8 章端口 60h 的命令 E8h。

调用：
ax=C203h
bh=分辨率
0= 25dpi，每毫米计数 1 次
1= 50dpi，每毫米计数 2 次

2=100dpi，每毫米计数4次
3=200dpi，每毫米计数8次

返回：　如果成功
进位=0
ah=0
如果功能失败
进位=1
ah=返回状态（参看表13-10）

子功能	描述	中断	功能
AL=4	读取设备ID	15h	AH=C2h

这个功能返回鼠标的设备ID。这个低级命令实际上获取一个双字节的ID号，但是在使用这个功能时，不会返回其中的低字节。

调用：　ax=C204h
返回：　如果成功
进位=0
ah=0
bh=设备ID（通常为0）
如果功能失败
进位=1
ah=返回状态（参看表13-10）

子功能	描述	中断	功能
AL=5	初始化鼠标	15h	AH=C2h

装入鼠标数据包的大小并重启鼠标。鼠标在每个数据包中可以发送1~8个字节，具体的取值与实际的设计有关。这个功能设置每个数据包中的字节数。这个值通常只需要发送一次，具体情况与鼠标和驱动程序有关。

在PS/1和PS/2系统上，这个数据包大小保存在扩展BIOS数据区的偏移量27h处。其他的供应商可能使用不同的区域来保存这个信息。

一旦保存好这个数据包，这个命令就会触发一条鼠标重启命令（AX=C201h）来重启鼠标。重启后会禁止初始化鼠标的状态，而采样率设置为每秒报告100次，100dpi的分辨率以及1∶1的比例。参看功能AX=C201h来重启鼠标而不改变数据包的大小。

调用：　ax=C205h

bh=数据包的大小，单位是字节（1~8）

返回：　如果成功

进位=0

ah=0

如果功能失败

进位=1

ah=返回状态（参看表 13-10）

子功能	描述	中断	功能
AL=6，BH=0	返回状态	15h	AH=C2h

获取鼠标状态及当前设置。

调用：　ax=C206h

bh=0

返回：　如果成功

进位=0

ah=0

bl=状态字节

位		
	7=0	未使用
	6=0	流模式
	1	远程模式
	5=0	禁止
	1	开放
	4=0	比例设置为 1:1
	1	比例设置为 2:1
	3=0	未使用
	2=1	按下左边的按钮
	1=1	未使用
	0=1	按下右边的按钮

cl=分辨率

0= 25dpi，每毫米计数 1 次

1= 50dpi，每毫米计数 2 次

2=100dpi，每毫米计数 4 次

3=200dpi，每毫米计数 8 次

dl=采样率值，单位是报告次数/秒（例如 64h=100 次报告/秒）

如果功能失败
进位=1
ah=返回状态（参看表 13-10）

子功能	描述	中断	功能
AL=6，BH=1	将比例因子设置为 1:1	15h	AH=C2h

为了直接对比例因子 1:1 进行编程，可以参看第 8 章端口 60h 的命令 E6h。

调用：
ax=C206h
bh=1
返回：
如果成功
进位=0
ah=0
如果功能失败
进位=1
ah=返回状态（参看表 13-10）

子功能	描述	中断	功能
AL=6，BH=2	将比例因子设置为 2:1	15h	AH=C2h

为了直接对比例因子 2:1 进行编程，可以参看第 8 章端口 60h 的命令 E6h。

调用：
ax=C206h
bh=2
返回：
如果成功
进位=0
ah=0
如果功能失败
进位=1
ah=返回状态（参看表 13-10）

子功能	描述	中断	功能
AL=7	设置鼠标处理程序的地址	15h	AH=C2h

鼠标设备驱动文件使用这个功能装入一个指向代码的指针。无论在什么时候报告一个新的鼠标状态时，系统 BIOS 都会对这个指针值进行远调用。有关鼠标状态和位置的数据会被保存在堆栈中。

鼠标的报告动作会触发中断 74h。中断 74h 的系统 BIOS 处理程序负责触发这个远调用。在进行远调用时，中断 74h 的处理程序将四字节数据推入到堆栈中，如表 13-11 所示。在堆栈上保存其他任何信息之前，会在远调用子程序的入口处显示每个字在堆栈内的位置。远调用子程序应该保存和恢复任何要使用和使用过的寄存器。

在远调用时应开放中断。这个调用子程序应该在返回时保持堆栈的状态与调用时的状态相同。触发 RETF 退出这个远调用，中断 74h 负责从堆栈中移走四个多余的字节。

表 13-11　远调用时堆栈内的鼠标信息

堆　栈	描　述		
字 0	[SP+Ah]处的状态字节		
	位	15~8=0	未使用
	位	7=1	Y 位置数据溢出
		6=1	X 位置数据溢出
		5=1	Y 位置数据为负
		4=1	X 位置数据为负
		3=1	未使用，但是设置为 1
		2=0	未使用
		1=1	按下了左边的按钮
		0=1	按下了右边的按钮
字 1	[SP+8]处的 X 位置		
	位	15~8=0	未使用
	位	7~0=x	X 坐标
字 2	[SP+8]处的 Y 位置		
	位	15~8=0	未使用
	位	7~0=x	Y 坐标
字 3	[SP+4]处的 Z 位置（将来使用）		
	位	15~0=0	未使用

调用：　ax=C207h

es:bx=指向子程序的远指针

返回：　如果成功

进位=0

ah=0

如果功能失败

进位=1

ah=返回状态（参看表 13-10）

中断	功能	描述	平台
15h	C3h	监视计时器控制	MCA/EISA

这个功能控制硬件监视计时器。MCA 和 EISA 系统提供了另外一个时钟，它可以在没有执行周期性更新时触发 NMI。参看第 16 章以了解时钟 3 的低层次细节。

如果在禁止中断时程序进入了一个无止境的循环，那么可以使用监视计时器来触发一个不可屏蔽中断，让 NMI 处理程序来执行任何必要的操作。在缺省情况下，BIOS POST 会禁止监视计时器，所以在出现监视计时器超时的时候，系统 BIOS 不会处理 NMI，而是让操作系统或环境来使用这个特性。一般操作系统或环境通过钩起中断 2，NMI 来使用这个特性。这里由两个子功能来禁止和开放监视计时器。

子功能	描述	中断	功能
AL=0	禁止监视计时器	15h	AH=C3h

调用：ax=C300h

返回：如果成功

进位=0

ah=0

如果不支持这个功能

进位=1

ah=80h 或 86h

子功能	描述	中断	功能
AL=1	开放监视计时器	15h	AH=C3h

调用：ax=C301h

bx=初始计数值（在 MCA 上为 1~255，在 EISA 上为 1~65535）

返回：如果成功

进位=0

ah=0

如果不支持这个功能

进位=1

ah=80h 或 86h

中断	功能	描述	平台
15h	C4h	可编程选项选择	MCA

可编程选项选择（POS）系统将系统中的每一个适配器的信息都记录到扩展 CMOS 内存中。POS 系统提供了一种机制来删去适配器开关和调解潜在的 I/O 端口冲突。

POS 接口由下面三个服务组成：

AL=	POS 描述
0	获取 POS 基 I/O 端口
1	允许安装槽
2	开放适配器

子功能	描述	中断	功能
AL=0	获取 POS 基 I/O 端口	15h	AH=C4h

大多数情况下，POS 的 I/O 端口地址固定为 100h，但是这个功能支持未来可能会改变的基 I/O 端口号。

调用：　ax=C400h

返回：　如果成功

　　进位=0

　　al=0

　　dx=POS 基 I/O 端口（通常为 100h）

　　如果失败

　　进位=1

　　al=0

　　如果不支持这个功能

　　进位=1

　　ah=80h 或 86h

子功能	描述	中断	功能
AL=1	开放安装槽	15h	AH=C4h

开放指定的安装槽。它会触发子功能 2 来完成这个操作。这个功能使用适配器开放和设置端口（端口 96h）来向指定的适配器插槽发出安装信息。

调用：　ax=C401h

　　bl=插槽号 1~8

返回：　如果成功

　　进位=0

al=1
bl=插槽号
如果插槽号超出了范围
进位=1
al=1
bl=未改变
如果不支持这个功能
进位=1
ah=80h 或 86h

子功能	描述	中断	功能
AL=2	开放适配器	15h	AH=C4h

这个子功能完成由子功能 1 启动的安装操作，它向端口 96h 发送一个零值来中止安装进程。如果在此之前没有触发子功能 1，那么系统不会执行任何操作。

调用：ax=C402h
返回：如果成功
进位=0
al=2
如果失败
进位=1
al=2
如果不支持这个功能
进位=1
ah=80h 或 86h

中断	功能	描述	平台
15h	C5h	启动系统中断	PS/2

系统 BIOS 内部的许多中断程序会调用这个功能来指示它们已经被触发。一个 TSR 或者操作系统可能会钩起这个功能，并阻止或改变各种 BIOS 服务。

在调用这个功能之前，BIOS 首先将初始的 AX 值传递给系统服务程序，然后触发中断 15h 的功能 C5h，并将 AL 设置为相应的中断代码。由于其他的 TSR 可能会钩起中断 15h，因此 AX 堆栈值并非总是可以访问的，这些 TSR 会向堆栈中加入其他的信息，甚至可能会在执行时改变到一个新的堆栈中。

如果必须由应用程序来触发这个功能，那么系统 BIOS 仅仅只会检查 AL 是否在 1~9 的

范围内，然后清除进位标志并返回。对于所有其他的 AL 取值，系统 BIOS 会认为不支持这个功能。

调用：　ah=C5h（来自系统 BIOS）
al=来自 BIOS 服务的中断代码
1=中断 19h，引导装入程序
2=中断 14h，串行口服务
3=中断 16h，键盘服务
4=中断 40h，软盘服务
5=中断 17h，打印机服务
6=中断 10h，视频服务
7=中断 12h，设备信息
8=中断 11h，主存大小
9=中断 1Ah，实时时钟服务

返回：　如果 BIOS 支持这个功能
进位=0
如果不支持这个功能
进位=1
ah=80h 或 86h

中断	功能	描述	平台
15h	C8h	高速缓存控制	PS/1

这个功能的子功能控制高速缓存的操作。这些子功能以前尚未公开过，在不同的系统上它们可能会有所不同。如果 CPU 处于保护模式下，那么功能 0~5 无效，并且会返回一个错误。

高速缓存控制功能提供了下面七个子功能：

AL=	高速缓存控制子功能
0	开放高速缓存
1	禁止高速缓存
2	未知的高速缓存子功能
3	未知的高速缓存子功能
4	开放高速缓存，带有准备操作
5	禁止高速缓存，带有准备操作
6	获取高速缓存状态

子功能	描述	中断	功能
AL=0	开放高速缓存	15h	AH=C8h

调用：ax=C800h

返回：如果成功

进位=0

ax=0

如果在此之前高速缓存没有通过 POST 测试

进位=1

ax=C804h

如果 CPU 处于保护模式下

进位=1

ax=C809h

如果不支持这个功能

进位=1

ah=80h 或 86h

子功能	描述	中断	功能
AL=1	禁止高速缓存	15h	AH=C8h

调用：ax=C801h

返回：如果成功

进位=0

ax=1

如果在此之前高速缓存没有通过 POST 测试

进位=1

ax=C803h

如果 CPU 处于保护模式下

进位=1

ax=C809h

如果不支持这个功能

进位=1

ah=80h 或 86h

子功能	描述	中断	功能
AL=2	未知的高速缓存功能	15h	AH=C8h

调用：ax=C802h

返回：如果成功

进位=0

ax=2

如果失败

进位=1

ax=C805h

如果 CPU 处于保护模式下

进位=1

ax=C809h

如果不支持这个功能

进位=1

ah=80h 或 86h

子功能	描述	中断	功能
AL=3	未知的高速缓存功能	15h	AH=C8h

调用：ax=C803h

返回：如果成功

进位=0

ax=3

如果失败

进位=1

ah=C8h

al=3、4 或 5，与错误类型有关

如果 CPU 处于保护模式下

进位=1

ax=C809h

如果不支持这个功能

进位=1

ah=80h 或 86h

子功能	描述	中断	功能
AL=4	开放高速缓存，准备操作	15h	AH=C8h

在某些情况执行一些与高速缓存有关的操作。在任何情况下，都开放高速缓存。

调用：　　ax=C804h

返回：　　如果成功

　　进位=0

　　ax=4

　　如果 CPU 处于保护模式下

　　进位=1

　　ax=C809h

　　如果不支持这个功能

　　进位=1

　　ah=80h 或 86h

子功能	描述	中断	功能
AL=5	禁止高速缓存，带准备操作	15h	AH=C8h

在某些情况执行一些与高速缓存有关的操作。在任何情况下，都禁止高速缓存。

调用：　　ax=C805h

返回：　　如果成功

　　进位=0

　　ax=5

　　如果失败

　　进位=1

　　ax=C803h

　　如果 CPU 处于保护模式下

　　进位=1

　　ax=C809h

　　如果不支持这个功能

　　进位=1

　　ah=80h 或 86h

子功能	描述	中断	功能
AL=6	获取高速缓存的状态	15h	AH=C8h

调用：　　ax=C806h

返回：　　如果禁止了高速缓存

　　进位=0

ax=6
bh=未知的高速缓存位（0 或 1）
bl=0
如果开放了高速缓存
进位=1
ax=6
bh=未知的高速缓存位（0 或 1）
bl=1
如果不支持这个功能
进位=1
ah=80h 或 86h

中断	功能	描述	平台
15h	C9h	返回 CPU 的级别代号	某些 386+

如果支持这个功能，则返回 CPU 的家族系列代号和级别代号。级别代号也就是 CPU 的版本号。在重启 386 及后来的 CPU 时，CPU 会初始化 DX 寄存器，在 DL 中装入级别代号，而在 DH 装入家族系列代号。如果 BIOS 在 POST 初期没有保存这个这个值，那么这个值就会被丢弃。新式的 486 以及后来的 CPU，例如 Pentium 和 Pentium Pro，提供了一条新的指令 CPUID，这条指令能够在任何时候获得这个级别代号的信息。

有几个供应商在他们最近推出的 BIOS 中支持了这个功能，IBM 就是其中之一。本书所提供的 CPUTYPE 程序就是利用这个功能来帮助标识 CPU 的级别代号的。如果在 CX 中的返回值是零，表示 BIOS 没有保存 CPU 信息。

调用：ah=C9h
返回：如果成功
进位=0
ah=0
ch=CPU 家族系列代号（4=486，5=586/Pentium，等等）
cl=CPU 级别代号
如果功能失败
进位=1
ah=80h 或 86h

中断	功能	描述	平台
15h	CAh	读/写 CMOS 内存	PS/1

提供了两个子功能来读和写 CMOS 内存寄存器 E~3Fh。

子功能	描述	中断	功能
AL=0	读 CMOS 内存寄存器	15h	AH=CAh

调用：
ax=CA00h
bl=要读取的 CMOS 寄存器（E~3Fh）

返回：
如果成功
进位=0
ah=0
cl=CMOS 寄存器的值
如果功能失败
进位=1
ah=错误代码
1=CMOS 掉电
3=bl 中的寄存器太高
4=bl 中的寄存器太低
80h=不支持这个功能（PC）
86h=不支持这个功能（其他）

子功能	描述	中断	功能
AL=1	写 CMOS 内存寄存器	15h	AH=CAh

在向一个 CMOS 寄存器写入了一个值后，这个子功能并不更新对 CMOS 内存的校验。如果 POST 检测到了 CMOS 的检查错误，则这个功能就不可用。

在写操作之前不必进行新的检查，因此支持对不同的寄存器进行写操作，只要不发生系统重启即可。没有专门的功能来创建和保存一个新的校验值。

调用：
ax=CA01h
bl=CMOS 寄存器（E~3Fh）
cl=要写入的值

返回：
如果成功
进位=0
ah=0
如果功能失败
进位=1
ah=错误代码

1=CMOS 掉电
3=bl 中的寄存器太高
4=bl 中的寄存器太低
80h=不支持这个功能（PC）
86h=不支持这个功能（其他）

中断	功能	描述	平台
15h	D8h	访问 EISA 系统信息	EISA

在 EISA 系统上，这个功能访问扩展 CMOS 内存，在这个扩展 CMOS 内存中含有系统和插槽的信息。参看第 15 章中有关 EISA 系统不同之处的章节以了解功能 D8h 服务的完整细节。

端口归纳

端口	类型	功能	平台
22h	输出	芯片组位标选择	AT+
23h	I/O	芯片组数据	AT+
26h	输出	电源管理位标选择	AT+
27h	I/O	电源管理数据	AT+
60h	输入	系统开关	PC
60h	输出	POST 诊断	XT
61h	I/O	多项功能和扬声器控制	AT+
61h	I/O	多项功能和扬声器控制	PC/XT
62h	输入	多项系统端口功能	XT
62h	输入	多项系统端口功能	PC
63h	输入	多项系统端口功能	XT
64h	输出	系统 8255 模式寄存器	PC/XT
80h	输出	POST 诊断	所有
84h	输出	POST 诊断	Compaq
84h	I/O	同步总线周期寄存器	EISA
90h	输出	POST 诊断	所有
91h	输入	卡选择反馈	MCA
92h	I/O	系统控制	MCA/EISA
94h	I/O	允许系统控制	MCA
E0h	I/O	拆分地址寄存器	MCA
E1h	I/O	内存寄存器	MCA
E3h	I/O	错误跟踪寄存器	MCA

E4h	I/O	错误跟踪寄存器	MCA
E5h	I/O	错误跟踪寄存器	MCA
E7h	I/O	错误跟踪寄存器	MCA
F0h	输出	数学协处理器——清除忙	AT+
F1h	输出	数学协处理器——重启	AT+
F8h	I/O	数学协处理器——操作码传送器	AT+
FAh	I/O	数学协处理器——操作码传送器	AT+
FCh	I/O	数学协处理器——操作码传送器	AT+
100h	输入	可编程选项选择 0	MCA
101h	输入	可编程选项选择 1	MCA
102h	I/O	可编程选项选择 2	MCA
103h	I/O	可编程选项选择 3	MCA
104h	I/O	可编程选项选择 4	MCA
105h	I/O	可编程选项选择 5	MCA
106h	输入	可编程选项选择 6	MCA
107h	输入	可编程选项选择 7	MCA
178h	输出	电源管理位标选择	AT+
179h	I/O	电源管理数据	AT+
300h	输出	POST 诊断	Award
461h	I/O	扩展 NMI 和控制	EISA
464h	输入	前一个确认总线管理器	EISA
465h	输入	前一个确认总线管理器	EISA
680h	输出	POST 诊断	MCA
C80h	输入	系统板 ID1	EISA
C81h	输入	系统板 ID2	EISA
C82h	输入	系统板 ID3	EISA
C83h	输入	系统板 ID4	EISA

端口细节

端口	类型	描述	平台
22h	输出	芯片组索引选择	AT+

许多新式的设计都使用集成芯片组来执行大量的操作，而这些操作以前都是由 IC 来处理的。这些芯片组通常需要访问另外一些功能和特性。通常使用两个 8 位 I/O 端口访问这些寄存器。向这个端口写入寄存器的索引，紧接着通过端口 23h 来访问寄存器。大多数寄存器由系统 BIOS POST 操作设置。

使用端口 22h 和 23h 的芯片组的详细规范说明，因各个供应商而有所不同。一般地，

只有系统 BIOS 才访问芯片组，因为 BIOS 要依据所使用的特定芯片来配置。大多数情况下，BIOS POST 操作仅仅只一次设置芯片组的值，然后就不再访问芯片组。

某些供应商，例如 Intel，支持通过 I/O 端口 22h 和 23h 来访问多个芯片。芯片标识寄存器控制是哪个芯片响应这些端口。一旦在标识寄存器中装入了芯片 ID，那么只有那个芯片才会响应端口 22h 和 23h。

Intel 芯片组使用下列芯片选择标识：

芯片 ID	访问的寄存器
1	82359 EISA DRAM 控制器，通用寄存器
2	82351 EISA 局部 I/O 支持
A1	82359 EISA DRAM 控制器，EMS 寄存器
FF	没有芯片和访问（缺省情况）

下面的例子展示了如何选择一个指定的芯片以及如何使用索引和数据寄存器。EGA/VGA 从段址 A000 处开始保存了 128K，但是一个单色系统只是从段址 B000 处开始使用视频内存。使用单色适配器时，可以获得另外的 64K DOS 内存，这样，总共可以有 704K 的 DOS 内存。下面的代码将访问 Intel 82539 DRAM 控制器，并激活地址 A000 处的另外的 64K。至于其他的芯片组，代码会有所不同。这个程序假定系统目前只有 640K 主存，没有使用扩展 BIOS 数据区和上端内存块（UMB）。

```
cli                                    ; 总是禁止中断
mov     al,     21h                    ; 访问芯片选择寄存器
out     22h,    al                     ; 将索引设置为 21h
mov     al,     1                      ; 选择 DRAM 控制器芯片
out     23h,    al                     ; 输出到索引 21h

mov     al,     42h                    ; 访问视频块寄存器
out     22h,    al                     ; 将索引设置为 42h
mov     al,     0FFh                   ; 将 A000~AFFF 设置为 RAM
out     23h,    al                     ; 输出到索引 42h

mov     ax,     40h                    ; 访问 BIOS 数据区
mov     es,     ax                     ;
out     word ptr es:[13h],704h         ; 将 RAM 设置为 704K

sti                                    ; 开放中断
```

Intel 芯片组支持将缺省的端口 22h 和 23h 设置为任意其他的端口 0~FFh。如果存在硬件冲突，则 POST 可能会改变端口号，但是实际上很少使用这一点。由虚索引和数据寄存器 22h 和 23h 来控制这一点。

下面的寄存器归纳总结了芯片组的控制对象。其他的供应商可能会提供不同的功能和控制，或者使用不同的寄存器分配方案。Cyrix 也使用这个端口来访问内部 CPU 寄存器。参看第 3 章可了解更多的详细细节。

表 13-12 寄存器归纳（Intel 82539 EISA DRAM 控制器，通用寄存器）

寄存器	功能
0	DRAM 内存阵列——第 1 行类型和大小
1	DRAM 内存阵列——第 2 行类型和大小
2	DRAM 内存阵列——第 3 行类型和大小
3	DRAM 内存阵列——第 4 行类型和大小
4	DRAM 速度检测和选择
5	线大小——控制 DRAM 交错
6	RSA 线模式
7	块高速缓存开放选择——视频、ROM 以及系统 BIOS
8	模式寄存器 A——DRAM 和高速缓存控制
9	模式寄存器 B——高速缓存、脉冲模式以及 BIOS 大小
A	模式寄存器 C——并行控制、脉冲和周期速度
10	主机定时
11	主机到系统的延迟计时
12	系统定时
13	DRAM 行预负荷定时
14	DRAM 行定时
15	DRAM 列定时
16	CAS 脉冲宽度
17	CAS 到 MDS 的延迟
21	芯片选择寄存器（设置为 1 表示访问这个寄存器）
22	虚索引寄存器
23	虚数据寄存器
28~2C	奇偶错误的陷入地址
30	读页面选中周期长度
31	读页面丢失周期长度
32	读行丢失周期长度
33	写页面选中周期长度

续表

寄　存　器	功　能
34	写页面丢失周期长度
35	写行丢失周期长度
40	开放低端内存块（0~7FFFF）
41	开放低端内存块（80000~9FFFF）
42	开放视频内存块（A0000~AFFFF）
43	开放视频内存块（B0000~BFFFF）
44	开放 ROM 内存块（C0000~CFFFF）
45	开放 ROM 内存块（D0000~DFFFF）
46	开放 BIOS 内存块（E0000~EFFFF）
47	开放 BIOS 内存块（F0000~FFFFF）
4E	将内存（80000~FFFFF）重新映射到扩展内存
50~53	可编程属性映射 1（高速缓存、写保护等等）
54~57	可编程属性映射 2（高速缓存、写保护等等）
58~5B	可编程属性映射 3（高速缓存、写保护等等）
5C~5F	可编程属性映射 4（高速缓存、写保护等等）
83~84	拆分地址寄存器，地址位 A20~32
85	高速缓存控制
8B	系统节流阀
8C	主机节流阀
8D	主机内存节流阀监视器
8E	主机系统节流阀
8F	主机系统节流阀监视器
90	开放 RAM
91	禁止 RAM
92~93	超时期满寄存器
94~95	主机内存对所有权的请求
96~97	系统内存对所有权的请求
98~99	主机内存所有权
9A~9B	系统总线所有权
9C~9D	主机的系统总线请求
9E~9F	内存所有权传送

表 13-13 寄存器归纳（Intel 82539 EISA DRAM 控制器，EMS 寄存器）

寄存器	功能
0	EMS 控制（EMS=扩展内存规范）
21	芯片选择寄存器（将它设置为“1”来访问这个寄存器）
22	虚索引寄存器
23	虚数据寄存器
80~8F	EMS 页面寄存器，页面 0~7

表 13-14 寄存器归纳（Intel 82351 EISA 局域 I/O 支持）

寄存器	功能
21	芯片选择寄存器（将它设置为“2”来访问这个寄存器）
C0	外部开放寄存器 A
C1	外部开放寄存器 B
C2	并行配置寄存器
C3	串行配置寄存器 A
C4	软盘控制器配置寄存器
C5	串行配置寄存器 B
C6	COM 3 芯片选择低端口地址
C7	COM 3 芯片选择高端口地址
C8	COM 4 芯片选择低端口地址
C9	COM 4 芯片选择高端口地址
D0~D3	通用芯片选择线，屏蔽寄存器 0~3
D4~D7	通用芯片选择线，低端口地址 0~3
D8~DB	通用芯片选择线，高端口地址 0~3
DC	扩展 CMOS RAM 页面高端口地址
DD	扩展 CMOS RAM 页面低端口地址
DF	扩展 CMOS RAM 访问选择高端口地址（端口地址的低部分选择 RAM 地址）
E8~EB	EISA ID 配置寄存器（供端口 C80~C83 使用）

端口	类型	描述	平台
23h	I/O	芯片组数据	AT+

这个端口和端口 22h 一起用来访问系统芯片组内的数据。首先在端口 22h 中指定寄存

器，然后就可以使用这个端口来写和读取数据。参看端口 22h 了解细节。

端口	类型	描述	平台
26h	输出	电源管理索引选择	AT+

某些电源可管理的膝上型电脑以及其他一些系统可以访问一组电源管理功能。首先向这个端口写入寄存器索引，然后通过端口 27h 即可访问那个寄存器。其他的外部电源管理常常使用其他的端口和寄存器。

表 13-15 寄存器归纳（Intel 82347 外部电源管理）

寄存器	功能
C0	悬挂和唤醒状态，系统状态
C1	供电状态，活动状态，一般用途的输出和控制
C2	控制位
C3	活动屏蔽
C4	NMI 屏蔽
C5	活动监视器的 I/O 范围
C6	开状态——电源输出控制位
C7	DOZE 状态——电源输出控制位
C8	睡眠状态——电源输出控制位
C9	悬挂状态——电源输出控制位
CA	电源控制位的极点控制
CB	当前输出位状态
CC	DOZE 时钟寄存器（缺省——4 秒钟）
CD	睡眠时钟寄存器（缺省——2 秒钟）
CE	悬挂时钟寄存器（缺省——1 秒钟）
CF	LCD 显示电源时钟寄存器（缺省——2 分钟）
D0	EL 显示电源时钟寄存器（缺省——2 分钟）

端口	类型	描述	平台
27h	I/O	电源管理数据	AT+

对带有电源管理的系统来说，这个端口访问一个寄存器，这个寄存器由前面一个端口 26h 的写操作来指定。参看端口 26h 了解细节。

端口	类型	描述	平台
60h	输入	系统复位	PC

早期的 PC 保留了一组四个寄存器来完成许多硬件功能。通过一个 8255 外部接口芯片或者与之等价的芯片，可以控制端口 60h~63h。端口 60h 用作主板上的开关，这些开关直接和 8255 上的端口 A 相连接。

使用中断 11h（获取设备信息）和中断 12h（获得主存大小）可以得到大多数的相关信息。

输入（0~7 位）

位		
7r=x 6r=x	软盘驱动器的数目（如果位 0=1）	
	位 7	位 6
	0	0=1 个软盘驱动器
	0	1=2 个软盘驱动器
	1	0=3 个软盘驱动器
	1	1=4 个软盘驱动器
5r=x 4r=x	视频显示类型	
	位 5	位 4
	0	0=未使用
	0	1=CGA，40 列 25 行
	1	0=CGA，80 列 25 行
	1	1=MDA，80 列 25 行
3r=x 2r=x	系统主板内存大小	
	位 3	位 2
	0	0=16KB
	0	1=32KB
	1	0=48KB
	1	1=64KB
1r=x	未使用	
0r=0	从 BASIC BIOS 引导（系统上无驱动器）	
1	从软盘驱动器引导	

端口	类型	描述	平台
60h	输出	POST 诊断	XT

在上电自检（POST）期间，每次测试时都向这个端口输出一个值。如果测试失败（或者挂起了系统），那么前面一个值会指示测试失败。端口号及其输出值与制造商有关，并且随着型号的差别而有所不同。

IBM XT BIOS 命令 POST 代码

POST 代码发送到端口 60h。XT 只使用有限的 POST 代码。另外，系统可能会发送一

个蜂鸣（Beep）代码，并显示其他相关的错误信息。

00h	处理器寄存器测试
01h	BIOS ROM 校验（如果字节加起来不等于零，测试失败）
02h	时钟 1 测试
03h	DMA 初始化和寄存器测试
04h	内存测试失败
FFh	处理器寄存器测试

端口	类型	描述	平台
61h	I/O	多项功能和扬声器控制	AT+

本端口控制扬声器和一组彼此无关的系统功能。在 AT 上这个端口和独立的硬件连接。大多数当前的副本将这个硬件接口作为所使用的芯片组的一部分。程序员主要用这个端口来生成扬声器音响。

在向这个寄存器写入时，记住位 4~7 总是零。最好是首先读取这个端口的值，然后只改变期望功能所需要的位。

输出（位 0~3，7），输入（位 0~7）

位		
	7r=1	RAM 奇偶错误（仅当位 2 开放时）
	6w=1	清除 IRQ 0 时钟锁存（仅 MCA）
	6r=1	I/O 奇偶错误（仅当位 3 开放时）
	5r=x	时钟 2 的输出（8254 或者与之等价的芯片）
	4r=x	刷新请求，时钟除以 2
	3r/w=0	开放 I/O 奇偶校验——如果出错则生成 NMI
	2r/w=0	开放 RAM 奇偶校验——如果出错则生成 NMI
	1r/w=1	允许扬声器数据
	0r/w=1	时钟 2 门开

代码例

将扬声器的开放位打开或关闭，可以向扬声器发送一个音调。下面这个程序使用的延迟是基于特殊指令的执行速度的，因此处理器和 CPU 的速度可能会改变音调。

```
spkr_beep    proc    near
             mov     bx,   256          ; 开/关循环时间周期
             mov     cx,   bx           ; 临时寄存器
             in      al,   61h          ; 读端口内容
```

```
                and     al,     0Feh            ; 关闭从时钟到扬声器的输出
                mov     dx,     80              ; 循环计数
spkr_cycle:
                or      al,     2
                out     61h,    al              ; 打开扬声器位
spkr_on_delay:
                loop    spkr_on_delay           ; 延迟
                and     al,     0FDh
                out     61h,    al              ; 关闭扬声器位
                mov     cx,     bx              ; 重新设置一个周期

spkr_off_delay:
                loop    spkr_off_delay          ; 延迟
                dec     dx
                jnz     spkr_cycle              ; 循环 80 次
                ret
skpr_beep       endp
```

端口	类型	描述	平台
61h	I/O	多项功能和扬声器控制	PC/XT

本端口控制扬声器和一组彼此不相关的系统功能。在 PC 和 XT 上，这个端口是 8255 外部接口芯片的输出端口 B。程序主要用它来生成扬声器音响。

改变这个端口的值时，首先读取当前的值，然后在回写新的值时只改变需要改变的位。

I/O（位 0~7）

位		
	7r/w=1	禁止端口 60h 开关，开放键盘数据，允许键盘 IRQ
	0	开放端口 60h 开关，禁止键盘数据，清除键盘 IRQ
	6r/w=0	保持键盘时钟低
	5r/w=0	开放 I/O 奇偶校验——如果出错则生成 NMI
	4r/w=0	开放 RAM 奇偶校验——如果出错则生成 NMI
	3r/w=0	磁带马达开（PC）
	2r/w=0	开放 RAM 大小的读开关（端口 62h 的位 0~3）
	1	开放读备用开关（端口 62h 的位 0）
	1r/w=1	允许扬声器数据
	0r/w=1	门时钟 2 输出到扬声器

代码例

参看端口 61h，AT+代码例。

端口	类型	描述	平台
62h	输入	多项系统端口功能	XT

本端口控制一组 XT 上彼此不相关的功能，当前的系统不再使用这个端口。

输入（位 0~7）

位	7r=1	RAM 奇偶错误（仅当端口 61h 的位 4 开放时）
	6r=1	I/O 奇偶错误（仅当端口 61h 的位 5 开放时）
	5r=x	时钟 2 的输出（8254 或者与之等价的芯片）
	4r=x	未使用
	3r=1	主板 RAM 大小类型 1
	2r=1	主板 RAM 大小类型 2
	1r=1	安装了数学协处理器，8087
	0r=1	在 POST 内循环（上电自检，以便诊断分析）

代码例

```
enable_RAM_parity:
        mov     al,     10h
        out     61h,    al              ; 开放 RAM 奇偶检查
        .
        .
check_RAM_parity:
        in      al,     62h
        test    al,     80h             ; 测试 RAM 奇偶错误
        jnz     parity_error            ; 如果出错，则跳转
        .                               ; 如果没有出错，则继续执行下去
        .
```

端口	类型	描述	平台
62h	输入	多项系统端口功能	PC

这个端口控制一组 PC 上彼此不相关的功能，当前的系统不再使用它。

输入（位 0~7）

位	7r=1	RAM 奇偶错误（仅当端口 61h 的位 4 开放时）			
	6r=1	I/O 奇偶错误（仅当端口 61h 的位 5 开放时）			
	5r=x	时钟 2 的输出（8254 或者与之等价的芯片）			
	4r=x	磁带数据输入			
	3r=x	附加内存×32MB			
	2r=x	位	3~0	0000 =	没有附加内存
	1r=x			1111 =	在主存之外还有 512K
	0r=x				

端口	类型	描述	平台
63h	输入	多项系统端口功能	XT

这个端口控制一组 XT 上彼此不相关的功能，当前的系统不再使用这个端口。

输入（位 0~7）

位	7r=x	软盘驱动器的数目			
	6r=x	位 7	位 6		
		0	0=1 个软盘驱动器		
		0	1=2 个软盘驱动器		
		1	0=3 个软盘驱动器		
		1	1=4 个软盘驱动器		
	5r=x	引导时的视频显示类型			
	4r=x	位 5	位 4		
		0	0=未使用		
		0	1=CGA，40 列 25 行		
		1	0=CGA，80 列 25 行		
		1	1=MDA，80 列 25 行		
	3r=x	系统主板内存大小			
	2r=x	位 3	位 2	（64K 芯片）	（256K 芯片）
		0	0	=16KB	256K
		0	1	=128KB	512K
		1	0	=192KB	576K
		1	1	=256KB	640K
	1r=x	未使用			
	0r= x	未使用			

端口	类型	描述	平台
64h	输出	系统 8255 模式寄存器	PC/XT

这个端口控制 8255 外围接口芯片和与之等价的芯片的操作。由于这些端口都操作在基本输入/输出模式下，所以 PC/XT 总在这个寄存器中装入值 99h。

这个端口不支持选通模式和双向模式，所以永远也不要对两种模式进行编程。在硬件重启期间，所有的端口都设置为输入模式，所以可以读取这些端口而不用先将模式寄存器设置为 99h。

输出（位 0~7）

位		
	7w=1	激活模式设置
	6w=x 5w=x	模式选择，端口 A 的全部位和端口 C 的高 4 位 位 6　　位 5 0　　0=基本操作（普通情况） 0　　1=选通模式（从未使用） 1　　x=双向模式（从未使用）
	4w=0	端口 A 设置为输出
	1	端口 A 设置为输入（普通情况）
	3w=0	端口 C 的高四位设置为输出（普通情况）
	1	端口 C 的高四位设置为输入
	2w=0	基本模式，端口 B 和端口 C 的低四位（普通情况）
	1	选通模式，端口 B 和端口 C 的低四位
	1w=0	端口 B 设置为输出（普通情况）
	1	端口 B 设置为输入
	0w=0	端口 C 的低四位设置为输出
	1	端口 C 的低四位设置为输入（普通情况）

代码例

```
init_8255:
    mov    al,   99h          ; 命令值
    out    64h,  al           ; 送往 8255
```

端口	类型	描述	平台
80h	输出	POST 诊断	所有

在上电自检（POST）期间，每次测试时都向这个端口输出一个值。如果测试失败（或者挂起了系统），那么前面一个值会指示测试失败。端口号及其输出值与制造商有关，并且随着型号的差别而有所不同。

许多诊断主板可以显示 POST 代码。常常有许多 POST 代码从未被使用过，具体情况与版本和制造商有关。

AMI BIOS 通用 POST 代码

POST 代码发送到端口 80h。如果出错，那么前面一个代码将会指示出了问题。带有（F）的代码表示这个错误是致命性的。下面的这些代码有一部分来自几本主板用户手册，这些手册对 AMI BIOS POST 代码做出了详细的说明。依据我的经验来看，大多数 1990 年或以后的 AMI BIOS 都使用了下面的 POST 代码。

01h	处理器标志测试失败（F）
02h	处理器寄存器测试失败（F）
03h	主 BIOS ROM 校验失败（F）
04h	时钟通道 2 失败
05h	时钟通道 2 的计数不正确
06h	时钟通道 1 的计数不正确（F）
07h	时钟通道 0（用于 DMA 刷新）的计数不正确（F）
08h	奇偶状态位粘着不变（F）
09h	RAM 刷新错误（F）
0Ah	地址低字节错误（F）
0Bh	前 64K RAM 的测试失败，序列数据（F）
0Ch	前 64K RAM 的测试失败，随机数据（F）
0Dh	RAM 奇偶错误（F）
0Eh	键盘控制器没有响应（F）
0Fh	键盘控制器自检失败（F）
10h	键盘控制器的模式不能正确地设置（F）
11h	键盘控制器的状态不正确（F）。
12h	CMOS 内存读/写失败（F）
13h	检测到了视频 ROM，但是校验失败
14h	视频 ROM 没有从初始化中返回（F）
15h	显示器内存测试失败
16h	显示器光标寄存器读/写失败（CGA/MDA）
17h	进入虚模式（F）
18h	开放 1MB+内存（A20 线）失败（F）
19h	全部 RAM 测试、地址测试失败
1Ah	全部 RAM 测试、序列数据模式测试失败
1Bh	全部 RAM 测试、随机数据模式测试失败
1Ch	在 RAM 测试时出现了奇偶错误
1Dh	切换回到实模式
1Eh	禁止 1MB+内存（A20 线）失败
1Fh	DMA 控制器页面寄存器读/写测试失败（F）

20h	DMA 控制器寄存器锁存测试失败（F）
21h	DMA 控制器 1 测试失败（F）
22h	DMA 控制器 2 测试失败（F）
23h	中断控制器 1 的屏蔽寄存器读/写测试失败（F）
24h	中断控制器 2 的屏蔽寄存器读/写测试失败
25h	中断控制器 1 上出现了伪中断（F）
26h	键盘错误
27h	键盘接口测试失败
28h	键盘数据线粘着不变
29h	键盘时钟线粘着不变
2Ah	键盘键按下状态保持粘着不变
2Bh	并行打印机端口设置
2Ch	RS-232 串行口设置
2Dh	时间和日期设置
2Eh	软盘控制器和驱动器设置
2Fh	硬盘控制器和驱动器设置
30h	在 C800 处检测到了 ROM，但是校验失败
31h	在 C800 处的 ROM 没有从初始化中返回（F）
32h	在 E800 处检测到了 ROM，但是校验失败
33h	在 E800 处的 ROM 没有从初始化中返回（F）
34h	通过 INT 19h 引导（引导自举）

AWARD BIOS 通用 POST 代码

POST 代码同时发送到端口 80h 和 300h，表示正在执行测试。

01h	处理器标志测试失败
02h	处理器寄存器测试失败
03h	初始化芯片，例如 RTS、数学协处理器、时钟、DMA、中断控制器等等
04h	测试内存刷新
05h	键盘初始化
06h	BIOS ROM 校验（如果字节加起来不等于零，则测试失败。也单独进行注册信号校验）
07h	测试 CMOS 接口和电池使用情况
08h	设置并测试前 256K 的 RAM，OEM 芯片组设置
09h	早期的高速缓存的内存初始化
0Ah	设置中断向量
0Bh	测试 CMOS RAM 校验

0Ch	初始化键盘和设置 Num-Lock 状态
0Dh	依据 CMOS 类型初始化视频接口
0Eh	测试视频内存并显示注册信息
0Fh	测试 DMA 控制器 1
10h	测试 DMA 控制器 2
11h	测试 DMA 页面寄存器
12h~13h	未使用
14h	测试 8254 时钟计数器 2
15h	测试 8259-1 的中断屏蔽位
16h	测试 8259-2 的中断屏蔽位
17h	测试 8259 粘着不变的中断屏蔽位
18h	测试 8259 中断控制器
19h	测试粘着不变的 NMI 位（来自奇偶或者 I/O 校验）
1Ah~1Eh	未使用
1Fh	如果支持 EISA 模式，则设置为 EISA 模式，并检查 EISA 配置内存接口和校验。如果不支持 EISA 模式，则执行 ISA 测试并清除 EISA 模式标志。
20h	初始化插槽 0（仅 EISA）
21h	初始化插槽 1（仅 EISA）
22h	初始化插槽 2（仅 EISA）
23h	初始化插槽 3（仅 EISA）
24h	初始化插槽 4（仅 EISA）
25h	初始化插槽 5（仅 EISA）
26h	初始化插槽 6（仅 EISA）
27h	初始化插槽 7（仅 EISA）
28h	初始化插槽 8（仅 EISA）
29h	初始化插槽 9（仅 EISA）
2Ah	初始化插槽 10（仅 EISA）
2Bh	初始化插槽 11（仅 EISA）
2Ch	初始化插槽 12（仅 EISA）
2Dh	初始化插槽 13（仅 EISA）
2Eh	初始化插槽 14（仅 EISA）
2Fh	初始化插槽 15（仅 EISA）
30h	确定基本内存和扩展内存的大小
31h	测试基本内存和扩展内存（如果按下了 ESC 键，则跳过测试）
32h	测试 EISA 扩展内存（如果在 ISA 模式下，或者按下了 ESC 键，则跳过测试）
33h~3Bh	未使用

3Ch	允许设置
3Dh	初始化和安装鼠标
3Eh	初始化高速缓存控制器
3Fh	依据设置条件设置影像 RAM
40h	未使用
41h	初始化软盘控制器和所附带的设备
42h	初始化硬盘控制器和所附带的设备
43h	检测并初始化任意的串行口和并行口
44h	未使用
45h	检测并初始化数学协处理器
46h	未使用
47h	设置系统速度
48h~4Dh	未使用
4Eh	如果安装了生产跳线，则重新启动。否则，在显示屏上显示任意的非致命性信息，并进入设置
4Fh	进入密码（如果带有这个配置）
50h	向 RAM 中写入 CMOS 值
51h	开放奇偶校验、NMI 以及高速缓存
52h	从 C800h~EF80h 检测并初始化可选 ROM
53h	从 CMOS 向 BIOS RAM 拷贝时间和日期
54h~62h	未使用
63h	设置堆栈，通过 INT 19h 引导
64h~AFh	未使用
B0h	保护模式下出现了伪中断
B1h	未声明的 NMI
B2h~BEh	未使用
BFh	程序芯片组
C0h	高速缓存控制（OEM 专用）
C1h	内存存在性测试（OEM 专用）
C2h	早期的内存初始化（OEM 专用）
C3h	扩展内存初始化（OEM 专用）
C4h	特殊视频显示开关处理（OEM 专用）
C5h	早期的影像效果（OEM 专用，以加快引导速度）
C6h	高速缓存的编程（OEM 专用）
C8h	特殊的速度切换（OEM 专用）
C9h	特殊的影像 RAM 处理（OEM 专用）
CAh	很早期的硬件初始化（OEM 专用）
CBh~E0h	未使用
E1h	设置页面 1

E2h	设置页面 2
E3h	设置页面 3
E4h	设置页面 4
E5h	设置页面 5
E6h	设置页面 6
E7h	设置页面 7
E8h	设置页面 8
E9h	设置页面 9
EAh	设置页面 A
EBh	设置页面 B
ECh	设置页面 C
EDh	设置页面 D
EEh	设置页面 E
EFh	设置页面 F
F0h~FEh	未使用
FFh	引导

IBM AT BIOS 通用 POST 代码

POST 代码被发送到端口 80h，指示正在进行测试。我从 IBM AT BIOS（发布日期为 6/10/85）中开发了下面这个列表。后来的 IBM AT POST 似乎基本上也使用下面这些代码。参看端口 680h 了解 IBM MCA 系统上使用的代码。

01h	处理器寄存器测试，重启视频
02h	BIOS ROM 校验（如果字节加起来不等于零则测试失败）
03h	CMOS 读/写
04h	时钟 1 测试，检查所有的计数寄存器位是否为开
05h	时钟 1 测试，检查所有的计数寄存器位是否为关
06h	DMA 0 初始化和寄存器测试
07h	DMA 1 初始化和寄存器测试，开始 DMA 刷新
08h	DMA 页面寄存器测试
09h	内存刷新测试
0Ah	键盘控制器接口测试，缓冲区清空
0Bh	键盘控制器，自检
0Ch	键盘控制器，命令测试
0Dh	键盘控制器，命令测试
0Eh	内存测试，前 64K
0Fh	内存测试，前 64K，冷启动

11h	核对速度和刷新时钟率
12h	保护模式寄存器测试，并初始化中断控制器 1
13h	初始化中断控制器 2
14h	将所有的中断向量设置到临时的处理程序
15h	设置中断向量 10h~17h
16h	检查 CMOS 电池状态
17h	检查 CMOS 校验
18h	禁止奇偶校验
19h	切换到保护模式
1Ah	保护模式测试
1Bh	RAM 大小 0~640K，检查奇偶错误
1Ch	检查主存是 512K 还是 640K
1Dh	确定扩展 RAM 的大小
1Eh	在 CMOS 地址 30h+31h 处设置扩展内存大小
1Fh	测试扩展内存的地址线（A19~A23）
20h	切换回实模式
21h	读 CMOS 设置信息和初始化视频
22h	视频测试
23h	检查是否存在 BIOS
24h	中断控制器 0 和 1，读/写屏蔽寄存器
25h	中断控制器 0 和 1，读/写屏蔽寄存器
26h	伪中断检测
27h	检查是否有 NMI 粘着不变
28h	NMI 粘着不变或者出现了伪 NMI
29h	测试时钟 2
2Ah	检查时钟 0（RAM 刷新速度）太慢
2Bh	检查时钟 0（RAM 刷新速度）太快
2Ch	设置时钟 0
2Dh	检查键盘控制器状态 0
2Eh	未使用
2Fh	内存测试的起点
30h	内存测试的起点，冷启动
31h	开放保护模式
31h	主存测试 0~640K（应该是 32h，但是在 BIOS 中是一个编码错误）
32h	未使用
33h	扩展内存测试
34h	切换回实模式
34h	通过了保护内存测试（复制 POST 代码）

35h	键盘测试的起点
35h	完成了保护内存测试（复制 POST 代码）
36h	重启键盘和检查缓冲区满和粘时钟或数据线
37h	键盘没有粘着
38h	检查粘着键
39h	检查键盘控制器接口
3Ah	设置 BIOS 中断向量
3Bh	检查 C800~DF80D 处的可选 ROM，初始化其他硬件
3Ch	软盘驱动器测试
3Dh	初始化软盘驱动器
3Eh	初始化硬盘
3Fh	初始化打印机、时间、日期，并清屏
40h	未使用
41h	检测和校验在 E000 处的 ROM
42h	初始化在 E000 处的 ROM
43h~DCh	未使用
DDh	向这个 POST 端口显示 RAM 错误地址（DD 是输出，接着是 RAM 地址字节，如此重复）
DEh~EFh	未使用
F0h	切换到保护模式
F1h	测试中断
F2h	测试一般保护性错误的处理程序
F3h	测试保护模式指令和控制标志
F4h	检查内存边界操作
F5h	测试 PUSHA 和 POPA 指令
F6h	测试描述功能正确的访问权限
F7h	测试 ARPL 指令功能
F8h	测试 LAR 指令
F9h	测试 LSL 指令
FAh	保护模式内存地址有效性检测

Phoenix ISA/EISA BIOS 通用 POST 代码

POST 代码被发送到这个诊断端口，指示正在测试或者显示测试错误。下面的信息部分来源于 Phoenix 的 BIOS 技术参考手册。

01h	测试 16 位的 CPU 寄存器
02h	对 CMOS 寄存器 0Eh 进行读/写测试

03h	主 BIOS ROM 校验
04h	测试时钟 0
05h	DMA 0，通道 0 地址和计数寄存器测试
06h	DMA 页面寄存器测试
07h	未使用
08h	内存 DRAM 刷新测试
09h	内存测试，前 64K RAM
0Ah	内存测试失败，前 64K RAM
0Bh~0Ch	未使用
0Dh	内存奇偶错误，前 64K RAM
0Eh	监视器时钟错误（仅 EISA 系统）
0Fh	强制性软件 NMI 端口错
10h~1Fh	64K RAM 测试失败，低四位指示了 RAM 芯片或数据线的 0~F 位中的哪一位测试失败
20h	DMA 1 初始化和寄存器测试（通道 1、2 和 3）
21h	DMA 0 初始化和寄存器测试（通道 1、2 和 3）
22h	初始化和测试中断控制器 1，屏蔽寄存器
23h	初始化和测试中断控制器 2，屏蔽寄存器
24h	未使用
25h	设置 BIOS 中断向量
26h	未使用
27h	键盘控制器，自检
28h	检查 CMOS 电池状态和 CMOS 校验
29h	有效的 CMOS 视频配置
2Ah	未使用
2Bh	测试视频文本内存区
2Ch	视频正在初始化
2Dh	测试视频回扫位切换
2Eh	检查是否存在视频 BIOS
2Fh	未使用
30h	检测到视频 BIOS，并且已经初始化
31h	单色视频操作设置和操作
32h	彩色 40 列模式视频操作设置和操作
33h	彩色 80 列模式视频操作设置和操作
34h	时钟 0 的 IRQ 0 测试
35h	处理器重启和恢复测试
36h	A20 门线失败
37h	在保护模式下发生了意外中断
38h	内存测试，64K 以上

39h	未使用
3Ah	测试时钟2
3Bh	测试CMOS实时时钟操作
3Ch	测试所有已经安装的串行口
3Dh	测试所有已经安装的并行口
3Eh	数学协处理器整数装入和保存测试

端口	类型	描述	平台
84h	输出	POST诊断	Compaq

在上电自检（POST）期间，每次测试时都向这个端口输出一个值。如果测试失败或者挂起了系统，那么上一个值会指示测试失败。端口号和值可能由于型号的差别而不同。

端口	类型	描述	平台
84h	I/O	同步总线主用寄存器	EISA

这个8位的寄存器值没有任何功能，它可以在写入一个值后读回这个值。读寄存器操作会生成一个外部I/O读周期。在这个I/O读周期完成之前，外部周期会更新EISA总线管理器或DMA和主存之间的任意缓冲区。这个操作的用处目前还不太清楚。可能并非所有的EISA都支持这个端口。

端口	类型	描述	平台
90h	输出	POST诊断	某些PS/2

在上电自检（POST）期间，每次测试时都向这个端口输出一个值。如果测试失败或者挂起了系统，那么上一个值会指示测试失败。端口号和值可能由于制造商和型号的差别而不同。目前大多数带有ISA总线的PS/2机器都使用端口90h。

端口	类型	描述	平台
91h	输入	选择卡ID字节	MCA

在访问一个适配卡的I/O时，这个卡会发送回一个信号来检测其是否存在。卡存在信号记录在这个端口中。读取这个端口之后，会清除存在性位。在访问系统I/O功能（例如串行口、并行口以及软盘控制器）之后，也会设置这一位。

输入（位0）

位　　0=1　　选择卡存在

端口	类型	描述	平台
92h	I/O	系统控制	MCA/EISA

这些位控制许多独特的系统功能。

输入（位 0~7）MCA

位		
	7r=x 6r=x	硬盘的 LED 活动状态 位 7　位 6 0　0=关 1　1=开 1　0=开 1　1=开
	5r=x	未使用（可能为 0，也可能为 1，取决于系统）
	4r=1	监视器超时
	3r=0	可以访问上电密码
	1	不可访问上电密码（从输出可获得更多细节）
	2r=x	未使用（可能为 0，也可能为 1，取决于系统）
	1r=0	禁止 A20 地址线，并将之强制为 0（实模式）
	1	开放 A20 地址线
	0r=0	冷引导
	1	强制高速重启（由 POST 读取）

输出（位 0~7）MCA

位		
	7w=x 6w=x	磁盘的 LED 活动控制 位 7　位 6 0　0=关 0　1=开 1　0=开 1　1=开
	5w=0	未使用
	4w=0	未使用
	3w=1	不可以访问上电密码字节。密码字节保存在 CMOS 寄存器 38h~3Fh 中。某些系统还包括字节 36 和 37h。一旦这一位设置为 1，则不可以清除这一位，除非上电重启。
	2w=0	未使用
	1w=0	禁止 A20 地址线，并将之强制为 0（实模式）
	1	开放 A20 地址线，允许访问 1MB 以上的内存
	0w=0	普通情况
	1	高速重启——系统在 13.4ms 内重启（用于从保护模式切换到实模式，而不用经过键盘控制器）

I/O（位 0~7）EISA（以前尚未说明，因供应商的不同而有所差别）

位		
	7r/w=x 6r/w=x	磁盘的 LED 活动控制 位 7　位 6
		0　0=关
		0　1=开
		1　0=开
		1　1=开
	5r=1	未使用或未知
	4r=0	未使用
	3r/w=0	可以访问上电密码
	3r/w=1	不可以访问上电密码字节。密码字节保存在 CMOS 寄存器 38h~3Fh 中。某些系统还包括字节 36 和 37h。一旦这一位设置为 1，则不可以清除这一位，除非上电重启。
	2r=1	未使用或未知
	1r/w=0	禁止 A20 地址线，并将之强制为 0（实模式）
	1	开放 A20 地址线
	0r/w=0	普通情况

端口	类型	描述	平台
94h	I/O	系统控制，开放	MCA

这个寄存器控制设置和开放某些主板功能。

在进入主板设置模式（位 7 或 5 设置为 0）之前，必须关闭适配器设置。读端口 96h 的第 3 位，看看是否为零，以检查是否关闭了适配器设置。如果没有关闭，向端口 94h 写入值 0 来关闭适配器设置模式。

在完成主板设置后，应向这个寄存器写入值 FFh。

I/O（位 0~7）

位		
	7r/w=x	软盘驱动器控制器、串行口、并行口和内存
	0	设置模式（通过端口 103h 控制内存，其他由端口 102h 控制）
	1	开放（缺省）
	6r/w=1	未使用
	5r=x	视频子系统
	0	设置模式（通过端口 102h 控制）
	1	开放，如果端口 102h 的位 0=1（缺省）
	4r/w=1	未使用
	3r/w=1	未使用

2r/w=1	未使用
1r/w=1	未使用
0r/w=1	未使用

端口	类型	描述	平台
96h	I/O	适配器有效和设置	MCA

这个寄存器允许控制和设置每个独立的适配卡插槽，也可作为一组用来重启所有的适配卡。

在进入任意一个适配器的设置模式之前，必须关闭系统设置。读端口 94h，看看是否为 FFh，以检查是否关闭了系统设置。如果返回了其他任何值，那么向端口 94h 写入一个值来退出系统设置和视频设置模式。

在完成适配器设置后，应向这个寄存器写入值 00h。

I/O（位 0~7）

位				
7w=0	普通情况		
1	激活了所有适配器插槽上的通道重启		
6r/w=1	未使用		
5r/w=1	未使用		
4r/w=1	未使用		
3r/w=1	向插槽发送卡设置信号，这个插槽由位 0~2 指定		
2r/w=x	选择卡插槽		
1r/w=x	位 2	位 1	位 0
0r/w=x	0	0	0=槽 1
	0	0	1=槽 2
	0	1	0=槽 3
	0	1	1=槽 4
	1	0	0=槽 5（并非所有系统）
	1	0	1=槽 6（并非所有系统）
	1	1	0=槽 7（并非所有系统）
	1	1	1=槽 8（并非所有系统）

端口	类型	描述	平台
E0h	I/O	拆分地址寄存器	MCA

这个寄存器指定了在 PS/2 的型号 80 系统上的拆分内存块的起点地址。

拆分内存指的是一种能力，可以将前 1 兆字节内存拆分成不同的用处。大多数系统都将主存设置成 640K，并使用剩下的内存来影映 BIOS ROM 和扩展内存。使用端口 E1h 处的内存寄存器来选择拆分类型。

拆分功能通常仅仅只留下 256K 或 384K 来加入到扩展内存中。在 1MB 的边界上提供 1MB 内存作为其他扩展内存部分。拆分后剩下的内存块必须放置到其他扩展内存的尾部。这个功能可以分配这些多余的内存块。

这些功能因主板类型的差别而有所不同。参看端口 E1h 开放拆分内存功能。

I/O（位 0~7）

位		
	7r/w=x	未使用
	6r/w=x	未使用
	5r/w=x 4r/w=x	连接器 2 的内存（2MB）（2 型主板） 位 5　位 4 0　0=允许 2MB 0　1=允许 2MB 中的后一个 1MB 1　0=允许 2MB 中的前一个 1MB 1　1=禁止或者没有安装
	3r/w=x 2r/w=x 1r/w=x 0r/w=x	拆分内存选择 1~15 处的哪一个 1MB 作为扩展内存区。如果激活了拆分内存，那么零值无效。

端口	类型	描述	平台
E1h	I/O	内存拆分寄存器	MCA

这个寄存器支持在 PS/2 的型号 80 系统上的主板内存的控制和状态。这些功能因主板类型的不同而有所差别。如果开放了拆分内存，端口 E1h 用来指定 1MB 以上的内存应该驻留在何处。如果系统有 16MB 以上的内存，就不要使用拆分内存。

I/O（位 0~7）1 型主板

位							
	7r/w=x 6r/w=x	连接器 2 的内存（1MB） 位 7　位 6 1　0=安装了 1　1=未安装					
	5r/w=x 4r/w=x	连接器 1 的内存（1MB） 位 5　位 4 1　0=安装了 1　1=未安装					
	3r/w=x 2r/w=x	拆分内存选择和 BIOS 影映控制 位 3	位 2	位 1	ROM	低端内存	1MB 以上的高端内存
	1r/w=x	0	0	1	=开	640K	384K

0	1	1	=开	512K	512K
1	0	0	=影映	640K	0
1	0	1	=开	640K	0
1	1	0	=影映	512K	0
1	1	1	=开	512K	0

0r/w=0 开放内存奇偶校验

1 清除内存奇偶错误（并重新写入一个 0 来重新开放）

I/O（位 0~7）2 型主板

位 7r/w=x 未使用

6r/w=x 未使用

5r/w=x 内存连接器 1

4r/w=x

位 5	位 4
0	0=允许 2MB
0	1=允许 2MB 中的后一个 1MB
1	0=允许 2MB 中的前一个 1MB

3r/w=x
2r/w=x
1r/w=x

拆分内存选择和 BIOS 影映控制

位 3	位 2	位 1	ROM	低端内存	1MB 以上的高端内存
0	0	0	=影映	640K	256K
0	0	1	=开	640K	256K
0	1	0	=影映	512K	384K
0	1	1	=开	512K	384K
1	0	0	=影映	640K	0
1	0	1	=开	640K	0
1	1	0	=影映	512K	0
1	1	1	=开	512K	0

0r/w=0 开放内存奇偶校验

1 清除内存奇偶错误（并重新写入一个 0 来重新开放）

端口	类型	描述	平台
E3h	I/O	错误跟踪寄存器	MCA

这是四个错误跟踪寄存器中的一个，某些 MCA 类型使用它们，比较典型的是用在 IBM 的 80 型上，但是 50/60 型不使用它。在发生每个 ERS（错误）信号的上升沿时，这个寄存器保存最高内存地址位。硬件重启会将这个寄存器强制设置为零。

I/O（位 0~7）

位 7r/w=x 地址位 23

6r/w=x	地址位 22
5r/w=x	地址位 21
4r/w=x	地址位 20
3r/w=x	地址位 19
2r/w=x	地址位 18
1r/w=x	地址位 17
0r/w=x	地址位 16

端口	类型	描述	平台
E4h	I/O	错误跟踪寄存器	MCA

这是四个错误跟踪寄存器中的一个，某些 MCA 类型使用它们。参看端口 E3h 了解细节。

I/O（位 0~7）

位 7r/w=x	地址位 15
6r/w=x	地址位 14
5r/w=x	地址位 13
4r/w=x	地址位 12
3r/w=x	地址位 11
2r/w=x	地址位 10
1r/w=x	地址位 9
0r/w=x	地址位 8

端口	类型	描述	平台
E5h	I/O	错误跟踪寄存器	MCA

这是四个错误跟踪寄存器中的一个，某些 MCA 类型使用它们。参看端口 E3h 了解细节。

I/O（位 0~7）

位 7r/w=x	地址位 7
6r/w=x	地址位 6
5r/w=x	地址位 5
4r/w=x	地址位 4
3r/w=x	地址位 3
2r/w=x	地址位 2
1r/w=0	地址对应于一个 I/O 端口
1	地址对应于内存
0r/w=0	允许
1	总线管理器控制的仲裁周期

端口	类型	描述	平台
E7h	I/O	错误跟踪寄存器	MCA

这是四个错误跟踪寄存器中的一个，某些 MCA 类型使用它们。参看端口 E3h 了解细节。这个寄存器保存了单独的一位，即当 ERS（错误）信号发生时其边沿的 D/C 信号线的状态。硬件重启会将这个寄存器强制设置为零。

I/O（位 0~7）

位		
	7r/w=x	未使用
	6r/w=x	未使用
	5r/w=x	未使用
	4r/w=x	未使用
	3r/w=x	未使用
	2r/w =x	未使用
	1r/w= x	未使用
	0r/w=0	为预取、暂停、中断确认指令来控制总线周期
	1	总线周期是内存和 I/O 访问的数据

端口	类型	描述	平台
F0h	输出	数学协处理器—清除忙	AT+

向这个端口写入一个值零会清除数学协处理器的忙标志。

如果数学协处理器正在执行一个给定的指令，或者出现了一个错误条件，那就会设置从协处理器到主处理器的硬件忙线。在一般操作情况下，会使用 WAIT 指令进行等待，直到在完成当前的数学协处理器指令时清除了忙线为止。如果发生了错误，那么在设置忙线的同时还会触发 IRQ 13（中断 75）。忙线会保持激活状态，直到向这个端口写入了一个数据字节 0 来清除这条线。

BIOS 通常会处理数学协处理器的错误条件。在出现中断 75 时，BIOS 会向端口 F0h 发送一个写命令来清除忙标志，同时还触发一个 NMI（中断 2）。这样做可以保证和老式的 PC/XT 计算机保持软件上的兼容。

输出（位 0~7）零值清除忙标志

端口	类型	描述	平台
F1h	输出	数学协处理器—重启	AT+

重启数学协处理器。如果向端口 71h 写入一个零值，就会重启数学协处理器。如果在此之前，数学协处理器处于保护模式下，重启操作会将数学协处理器切换回实模式。

这个重启操作的动作和上电重启或系统重启相同。

输出（位 0~7）零值重启数学协处理器

端口	类型	描述	平台
F8h	I/O	数学协处理器—操作码传送	AT+

处理器通过端口 F8h、FAh 以及 FCh 和数学协处理器通信。使用这些端口将操作码和操作数传递给数学协处理器和传回结果。

端口	类型	描述	平台
FAh	I/O	数学协处理器—操作码传送	AT+

参看端口 F8h。

端口	类型	描述	平台
FCh	I/O	数学协处理器—操作码传送	AT+

参看端口 F8h。

端口	类型	描述	平台
100h	输入	可编程选项选择 0	MCA

这个端口支持读取适配器标识字的低字节。必须在此之前使用端口 96h 将这个指定的适配器放置到设置模式中。

输入（位 0~7）适配器 ID 的低字节

端口	类型	描述	平台
101h	输入	可编程选项选择 1	MCA

这个端口支持读取适配器标识字的高字节。必须在此之前使用端口 96h 将这个指定的适配器放置到设置模式中。

输入（位 0~7）适配器 ID 的高字节

端口	类型	描述	平台
102h	I/O	可编程选项选择 2	MCA

这个端口用于设置系统主板和适配器。对于系统设置而言，必须使用端口 94h 激活系统设置状态。要选择一个指定的适配器，必须在此之前使用端口 96h 将这个指定的适配器放置到设置模式中。

I/O（位 0~7）系统设置

位		
	7r/w=0	并行口双向模式
	1	并行口仅输出模式（POST 后的缺省情况）
	6r/w=x 5r/w=x	IRQ 7 的并行口分配情况 位 6　位 5 0　0=用作 LPT1，地址为 3BCh~3BFh 0　1=用作 LPT2，地址为 378h~37Bh 1　0=用作 LPT3，地址为 378h~37Bh 1　1=无效
	4r/w=1	开放并行口（参看位 0）
	3r/w=0	串行口作为 COM2，地址为 2F8h~2FFh，使用 IRQ3
	1	串行口作为 COM1，地址为 3F8h~3FFh，使用 IRQ4
	2r/w=1	开放串行口（参看位 0）
	1r/w=1	开放软盘控制器（参看位 0）
	0r/w=0	禁止系统过载（并行口、串行口和软盘控制器都被禁止，而不管位 1、2 和 4 的设置如何）
	1	开放普通的系统操作

I/O（位 0~7）适配器设置——选项选择字节 1

位		
	7r/w=x 6r/w=x 5r/w=x 4r/w=x 3r/w=x 2r/w=x 1r/w=x	由适配卡定义
	0r/w=0	禁止选中的适配卡
	1	运行选中的适配卡

端口	类型	描述	平台
103h	I/O	可编程选项选择 3	MCA

这个端口用于设置系统主板和适配器。对于系统设置而言，必须使用端口 94h 激活系统设置状态。要选择一个指定的适配器，必须在此之前使用端口 96h 将这个指定的适配器放置到设置模式中。

I/O（位 0）系统设置（PS/2 的 50/55/60 型）

位		
	0r/w=0	禁止主板内存
	1	开放主板内存

输入（位0~7）系统设置（PS/2的70型，类型1和2）

位		
	7r=x	未使用
	6r=0	如果安装了内存，连接器3，是1MB
	1	如果安装了内存，连接器3，是2MB
	5r=0	内存安装在连接器3中
	4r=x	未使用
	3r=0	如果安装了内存，连接器2，是1MB
	1	如果安装了内存，连接器2，是2MB
	2r=0	内存安装在连接器2中
	1r=0	如果安装了内存，连接器1，是1MB
	1	如果安装了内存，连接器1，是2MB
	0r=0	内存安装在连接器1中

输入（位0~7）系统设置（PS/2的80型，类型1）；参看端口E1和E2h

位		
	7r=x	未使用
	6r=x	未使用
	5r=x	未使用
	4r=x	未使用
	3r=x 2r=x	连接器2的内存（1MB） 位3 位2 0 0=安装了 1 1=未安装
	1r=x 0r=x	连接器1的内存（1MB） 位1 位0 0 0=安装了 1 1=未安装

输入（位0~7）系统设置（PS/2的80型，类型2）；参看端口E1和E2h

位		
	7r=x	未使用
	6r=x	未使用
	5r=x	未使用
	4r=x	未使用
	3r=x 2r=x	连接器2的内存（2MB） 位3 位2 1 0=安装了 1 1=未安装
	1r=1 0r=0	连接器1的内存（必须有2MB）

I/O（位 0~7）适配器设置——选项选择字节 2，由指定的适配卡定义

端口	类型	描述	平台
104h	I/O	可编程选项选择 4	MCA

这个端口用于设置所有型号的适配器和某些型号的系统。对于系统设置而言，必须使用端口 94h 激活系统设置状态，还必须使用端口 105h 来选择指定的内存连接器。要选择一个指定的适配器，必须在此之前使用端口 96h 将这个指定的适配器放置到设置模式中。

I/O（位 0~7）系统设置（PS/2 的 55 型）

位						
7r=x	内存卡的 ID					
6r=x	位	7	6	5	4	大小	速度
5r=x		0	0	0	1 = 2MB	100ns
4r=x		0	0	1	0 = 1MB	100ns
		0	1	0	1 = 2MB	85ns
		0	1	1	0 = 1MB	85ns
3r=x	未使用					
2r=x	未使用					
1r/w=1	在 2MB 的卡上，开放第二个 1MB					
0r/w=1	开放第一个 1MB					

I/O（位 0~7）适配器设置——选项选择字节 3

位	
7r/w=x	通道检查活动指示器
6r/w=x	通道检查状态可用指示器
5r/w=x	由适配卡定义
4r/w=x	
3r/w=x	
2r/w=x	
1r/w=x	
0r/w=x	

端口	类型	描述	平台
105h	I/O	可编程选项选择 5	MCA

这个端口用于设置所有型号的适配器和某些型号的系统。对于系统设置而言，必须使用端口 94h 激活系统设置状态。要选择一个指定的适配器，必须在此之前使用端口 96h 将这个指定的适配器放置到设置模式中。

I/O（位 0~7）系统设置（PS/2 的 55 型）

位	7r=x	未使用
	6r=x	未使用
	5r/w=0	映射 BIOS 地址中的 128KB 内存和 1MB 以上的 256KB
	1	禁止 384KB 的内存（仍为 640KB 的主存）
	4r/w=1	BIOS 影映设置模式——读自 ROM 的地址 E0000~FFFFF，如果位 5=0，则写操作进入 RAM 中。
	0	开放 BIOS 影映——地址 E0000~FFFFF 从影映 RAM 中读取，禁止写
	3r/w=x	未使用
	2r/w=x 1r/w=x 0r/w=x	选择内存连接器 0~7（参看端口 104h）

I/O（位 0~7）适配器设置——选项选择字节 4

端口	类型	描述	平台
106h	输入	可编程选项选择 6	MCA

这个端口指定了适配器子地址扩展字的低字节。必须在此之前使用端口 96h 将这个指定的适配器放置到设置模式中。

I/O（位 0~7）适配器子地址扩展字的低字节

端口	类型	描述	平台
107h	输入	可编程选项选择 7	MCA

这个端口指定了适配器子地址扩展字的高字节。必须在此之前使用端口 96h 将这个指定的适配器放置到设置模式中。

I/O（位 0~7）适配器子地址扩展字的高字节

端口	类型	描述	平台
178h	输出	电源管理索引选择	AT+

某些膝上型电脑和其他一些系统用到电源管理，它们可以访问一组电源管理功能。这个端口是一个可选用的端口，具体情况与硬件的执行情况有关。它的功能和端口 26h 和 27h 中描述的功能相同。

端口	类型	描述	平台
179h	I/O	电源管理数据	AT+

对于带有电源管理功能的系统而言，这个端口访问电源管理寄存器，这个寄存器在前面已经由端口 178h 的写操作指定。参看端口 178h 了解相关细节。

端口	类型	描述	平台
300h	输出	POST 诊断	Award

POST 期间的诊断信息被发送到了端口 80h 和 300h。参看端口 80h 了解相关细节。

端口	类型	描述	平台
40Dh	输入	级别寄存器	EISA

读取 Intel 的 82357 EISA 集成系统外部芯片的级别（改进版本号）。其他的 EISA 不太可能会执行这项功能。

端口	类型	描述	平台
40Eh	I/O	测试寄存器 1	EISA

Intel 的 82357 EISA 集成系统外部芯片的测试寄存器，这个寄存器目前尚未公开。其他的 EISA 不太可能会执行这项功能。

端口	类型	描述	平台
40Fh	I/O	测试寄存器 2	EISA

Intel 的 82357 EISA 集成系统外部芯片的测试寄存器，这个寄存器目前尚未公开。其他的 EISA 不太可能会执行这项功能。

端口	类型	描述	平台
461h	I/O	扩展 NMI 和控制	EISA

这个寄存器控制各种错误条件和状态。

输出（位 0~3），输入（位 0~7）

位		
	7r=0	没有监视器超时错误，或者没有开放 NMI
	1	出现了监视器超时现象——NMI 挂起
	6r=0	没有总线超时错误，或者没有开放 NMI
	1	出现了总线超时现象——NMI 挂起
	5r=0	无 I/O 端口状态，或者没有开放 NMI
	1	I/O 端口状态——NMI 挂起
	4r=1	出现总线管理器抢占超时
	x	如果位 6=0，未定义

3r/w=0	禁止总线超时并被清除
1	允许总线超时——如果允许 NMI，并且出现了超时，那么会生成一个 NMI，并设置状态位 6
x	未定义，如果位 6=0
2r/w=0	禁止监视器超时并被清除
1	允许监视器超时——如果允许 NMI，并且出现了超时，那么会生成一个 NMI，并设置状态位 7
1r/w=0	禁止 NMI I/O 端口控制（缺省）
1	允许 NMI I/O 端口控制——如果允许 NMI，并且出现了超时，那么会生成一个 NMI，并设置状态位 5
0r/w=0	普通的总线重启操作（缺省）
1	总线重启激活

端口	类型	描述	平台
464h	输入	上次确认的总线管理器	EISA

获得总线控制的上一个总线管理器，被记录在这个寄存器中，记录的方式是将与它对应的位设置为 0。所有其他的位将会是 1。这个寄存器是只读的。

在出现总线超时的时候，找出是哪一个总线管理器适配器造成了这个问题，将是非常有用的。在检测出一个总线超时错误后，会由 NMI 程序来处理它。

输入（位 0~7）

位	
7r=0	上一个是总线管理器 1
6r=0	上一个是总线管理器 2
5r=0	上一个是总线管理器 3
4r=0	上一个是总线管理器 4
3r=0	上一个是总线管理器 5
2r=0	上一个是总线管理器 6
1r=0	上一个是总线管理器 7
0r=0	上一个是总线管理器 8

端口	类型	描述	平台
465h	输入	上次确认的总线管理器	EISA

获得总线控制的上一个总线管理器，被记录在这个寄存器中，记录的方式是将与它对应的位设置为 0。所有其他的位将会是 1。这个寄存器是只读的。

在出现总线超时的时候，找出是哪一个总线管理器适配器造成了这个问题，将是非常有用的。在检测出一个总线超时错误后，会由 NMI 程序来处理它。

输入（位 0~7）

位		
	7r=0	上一个是总线管理器 9
	6r=0	上一个是总线管理器 10
	5r=0	上一个是总线管理器 11
	4r=0	上一个是总线管理器 12
	3r=0	上一个是总线管理器 13
	2r=0	上一个是总线管理器 14
	1r=0	上一个是总线管理器 15
	0r=1	未使用

端口	类型	描述	平台
680h	输出	POST 诊断	MCA

在上电自检（POST）期间，每次测试时都向这个端口输出一个值。如果测试失败（或者挂起了系统），那么前面一个值会指示测试失败。端口号及其输出值与制造商有关，并且随着型号的差别而有所不同。端口 680h 目前仅用于带有 MCA 总线的机器上。

POST 代码发送到端口 680h，也发送到并行打印机 1，后者位于 I/O 端口 3BCh 处。

IBM MCA BIOS 通用 POST 代码

POST 代码发送到这个端口，并指示正在执行测试。下面这个列表是从 IBM PS/2 BIOS 的 70 型 BIOS 中发展而来，这个 BIOS 的发布日期为 2/7/89。其他的 IBM PS/2 MCA BIOS ROM 似乎也使用下面大部分的代码。

01h	处理器寄存器测试
02h	BIOS ROM 校验（如果字节加起来不等于零，那么校验失败）
03h	系统开放端口测试
04h	可编程选项选择 2 测试
05h	适配器设置端口测试
06h	CMOS 读/写
07h	扩展 CMOS 读/写
08h	DMA 和页面寄存器测试
09h	DMA 初始化
0Ah	内存刷新测试
0Bh	键盘控制器接口测试
0Ch	键盘控制器，自检
0Dh	键盘控制器，自检
0Eh	键盘控制器，自检错误
0Fh	系统内存设置

10h	测试系统内存（错误的地址发送到端口 681h），清空视频内存，初始化 VGA 控制器
11h	内存测试失败
12h	设置保护模式下的 LDGT/LIDT/SGDT/SIDT
13h	初始化中断控制器 1
14h	初始化中断控制器 2
15h	中断向量和指针被装入到未使用的中断处理程序中
16h	设置中断向量 10h~1Fh
17h	检查 CMOS 电池状态
18h	检查 CMOS 校验
19h	CMOS 电池电量不足或者没有连接
1Ah	冷启动——清除内存奇偶标志位和 IRQ 1 锁存器
1Bh	进入保护模式
1Ch	在保护模式下进行实际检查
1Dh	确定低端内存大小
1Eh	内存测试，允许奇偶校验
1Fh	BIOS 影映设置
20h	扩展内存检查
21h	DRAM 地址线测试
22h	开放内存和 I/O 端口奇偶校验
23h	热启动，在中断向量中装入未使用的处理程序
24h	CMOS 电池状态和校验检查
25h	冷启动键盘控制器测试
26~33h	未使用
34h	描述符测试正常
35h	描述符测试完成
36~3Fh	未使用
40h	视频初始化
41h	主板 ID 有效性检查
42h	中断控制器 0 和 1，读/写屏蔽寄存器 0
43h	中断控制器 0 和 1，读/写屏蔽寄存器 FF
44h	刷新时钟测试
45h	NMI 测试
46h	NMI 测试失败
47h	时钟 2 测试——接口和计数寄存器
48h	时钟 2 测试——输出，没有被短路到时钟 0 的输出上
49h	时钟 0 测试——接口和计数寄存器
4Ah	时钟 0 测试——没有被短路到时钟 2 的输出上
4Bh	时钟 0 测试——IRQ 0 操作

4Ch	时钟 3 测试——监视器时钟
4Dh	时钟 0 中断出现检查
4Eh	键盘控制器准备好
4Fh	软件重启测试，冷启动
50h	保护模式设置，冷启动
51h	进入保护模式，冷启动
52h	64K 内存测试，冷启动
53h	内存测试完成，冷启动
54h	系统重启并进入实模式，冷启动
55h	生产测试或普通测试
56h	键盘电源测试，并测试键盘数据线是否粘着
57h	键盘自检
58h	键盘测试成功
59h	键盘时钟和数据线测试
5Ah	鼠标测试和初始化
5Bh	鼠标禁止，键盘开放
5Ch	设置中断向量 8~0Fh
5Dh	设置中断向量 70~7Fh
5Eh	设置中断向量 2、5、18h，并将向量 60~6Fh 设置为零
5Fh	硬盘控制器重启
60h	硬盘 0 的配置设置
61h	软盘驱动器重启
62h	软盘驱动器测试
63h	软盘驱动器马达关
64h	串行口测试和键盘缓冲区的初始化
65h	时钟 0 中断开放（时钟滴答），如果前面的内存大小出错或者系统配置数据无效，标志将出错
66h	硬盘驱动器和软盘驱动器已经配置好
67h	设置中断向量 13h，并初始化软盘驱动器
68h	开放 CPU 仲裁
69h	检查 C000~DF80 处的可选 ROM
6Ah	串行口和并行口测试
6Bh	确定设备字节和检查 CMOS 校验
6Ch	数学协处理器初始化
6Dh	键盘 BIOS 的值的初始化，开放 IRQ 1（键盘），打印机端口初始化
6Eh	继续执行引导自举程序
6Fh	准备装入引导自举
70h	清空了区域 0:7C00 处的引导自举程序

71h	完成软盘重启
72h	将引导扇区读入到内存中
73h	引导失败，进入 ROM BASIC
74~BDh	未使用
BEh	设置 IDT 和 GDT，并进入保护模式
BFh	成功地进入了保护模式
C0~E0h	未使用
F0h	内存测试正常
F1h	在保护模式下检查
F2h	中断 20h 发挥作用（非 DOS 终止）
F3h	描述符表设置
F4h	LDT 和 TS 寄存器测试完成
F5h	边界检查测试
F6h	CPU 寄存器测试正常
F7h	检查访问测试完成
F8h	优先级测试完成
F9h	访问权限测试完成
FAh	段址限制测试完成

端口	类型	描述	平台
681h	输出	POST 辅助诊断	MCA

这个端口通常和端口 680h 一起使用，以提供某些测试所需要的额外信息。

端口	类型	描述	平台
C80h	输入	系统主板 ID 1	EISA

这个字节和端口 C81h 的字节组成一个 3 字符的制造商 ID。这些字节被压缩成 5 位的字符。对每个 5 位字符加上 40h，就可以将它们转化为 ASCII 字母 A~Z。

为了保证信息的有效性，可以向端口 C80h 写入 FFh，接着执行读操作。如果位 7 是 1，则没有有效的 ID 信息。

输入（位 0~6）

位	7r=0	固定
	6r=x	ID 字符 1
	5r=x	
	4r=x	
	3r=x	
	2r=x	
	1r=x	ID 字符 2 的高 2 位（参看端口 C81h）
	0r=x	

端口	类型	描述	平台
C81h	输入	系统主板 ID 2	EISA

这个字节和端口 C81h 的字节组成一个 3 字符的制造商 ID。这些字节被压缩成 5 位的字符。对每个 5 位字符加上 40h，就可以将它们转化为 ASCII 字母 A~Z。

输入（位 0~7）

位		
	7r=x	ID 字符 2 的低 3 位（参看端口 C80h）
	5r=x	
	6r=x	
	4r=x	ID 字符 3
	3r=x	
	2r=x	
	1r=x	
	0r=x	

端口	类型	描述	平台
C82h	输入	系统主板 ID 3	EISA

个别 EISA 主板制造商分配了这个端口，以实现某些特殊的或者独特的功能。

端口	类型	描述	平台
C83h	输入	系统主板 ID 4	EISA

这个字节定义了 EISA 的总线版本。第一个版本用 1 表示。

输入（位 0~7）

位		
	7r=x	由制造商定义
	6r=x	
	5r=x	
	4r=x	
	3r=x	
	2r=x	EISA 总线版本号
	1r=x	
	0r=x	

第 14 章

并行口和屏幕打印

从第一台 PC 开始，打印机操作改变很少。打印机 BIOS 功能当然很简单，但是我还是更为详细地解释了错误信息。

过去的几年中对并行口也做了一些改进，包括双向数据控制和设计快速端口来提供 10 倍于普通并行口的执行速度。详细解释了这些功能。

我创建了三个小工具来显示并行口不同的、鲜为人知的方面。包括一个过滤器 TSR，它能将 IBM 字符集的前 128 个字符转化为那些不支持完整 IBM 字符集的打印机可打印的字符。其他两个简单的工具将交换打印机，并检测系统上的打印机和标识出双向端口。

简　介

PC 系统支持不超过四个的并行口。这些并行口为打印机附件所设计。此外，并行口常常用于防拷贝锁、外部磁带备份设备、外部 CD-ROM 驱动器以及其他专用数据传输中。

某些并行口带有可任选的扩展模式，它支持完全的双向传输。有时用于外部磁盘和磁带驱动器、网络连接、以及机器到机器的数据传输。所有的 MCA 系统提供了带有外部模式功能的并行口。许多新式的系统也支持“快速并行口标准”。这极大的提高了并行口的最大流通能力。

通常，并行口驻留在独立的适配卡上。并行口由一组控制和数据锁存器组成，后者向并行口连接器发送信号。可以读取这个发送到连接器的信号来确认相应的操作。也可以从连接器读取其他的状态和控制信号。如果支持扩展模式，并且位于读模式下，就会禁止锁存数据输出，这样可以从连接器读取数据。

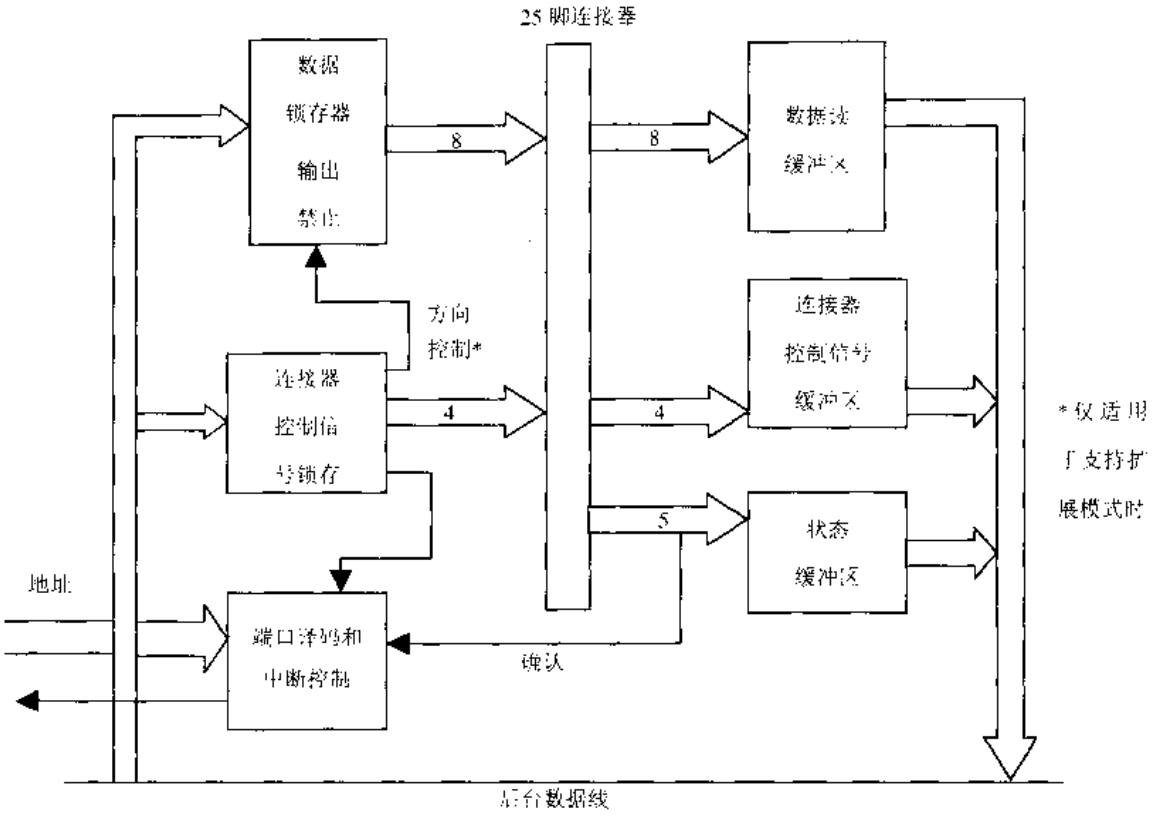


图 14-1　并行口

系统 BIOS 提供了专用的打印机服务中断 17h，来初始化打印机、获取打印机状态、以及向打印机发送数据。对于所有其他的并行口用户来说，有必要直接通过与并行口相关的 I/O 端口地址来实现通信。

有四个可能的并行口，编号为 1~4。通过一组连续的 I/O 端口地址访问它们。在 I/O 端口地址和并行口编号之间没有直接的映射关系。分配给每个并行口编号的 I/O 端口由系统的上电自检测试（POST）确定。

许多低级程序都是从零开始计数的。访问端口 1 时，寄存器中装入的值为 0。

BIOS 初始化

重启后，POST 检查是否安装了某个并行口。POST 按顺序仔细检查 I/O 端口地址 3BCh、378h、以及 278h。向某个 I/O 端口写一个字节，然后从这个端口读取该字节，来检测活动的端口。如果字节匹配，则 POST 接受该端口，并认为在测试的 I/O 地址有一个活动的并行口。然后将活动并行口的 I/O 地址保存在最低的未使用并行口 BIOS RAM 地址中。从 BIOS RAM 地址 40:8 开始分配了四个字。这些字节用来保存活动的并行口 I/O 地址。

所有这些意味着，并行口 1 地址保存在 BIOS RAM 地址 40:8 中，并行口 2 保存在 40:Ah 中，并行口 3 保存在 40:Ch 中。每个字的内容中有一个基本 I/O 端口地址，用来访问并行

口。值零表示没有检测到并行口硬件。

这些检测并不保证实际上是否有一个打印机或其他设备附于该 I/O 端口上。测试只检查在指定的 I/O 地址是否存在并行口硬件。POST 将检测到的活动并行口的数目保存在设备字节中，该字节位于 BIOS RAM 地址 40:10h 的位 14 和 15 中。

例如，如果一个系统有两个并行口，使用 I/O 地址 378h 和 278h，则并行口 RAM 地址将具有下述内容：

RAM 地址	并行口	I/O 端口地址
40:8h	1	378h
40:Ah	1	278h
40:Ch	1	0
40:Eh	1	0

一个系统可以有第四个并行口吗?

早期的 PC 和 XT 支持第四个并行口。从 AT 开始，以前在 BIOS RAM 地址 40:Eh 中用来保存第四个并行口端口地址的字被标注为“保留”。AT BIOS 并没有使用这个字节，所以在 AT 机器上很容易将它作为第四个并行口。

PS/2 是第一个真正将 BIOS RAM 字 40:E 用作扩展 BIOS 数据区（EBDA）的段址的系统。第六章详细描述了 EBDA。其他许多制造商也这样处理，所以在 AT+系统上使用第四个端口是不安全的，除非明确知道 BIOS 制造商没有将这个定义用于其他地方。仅仅检查到地址 40:E 的值为零是不够的，因为这也可能表示没有扩展 BIOS 数据区。在 BIOS 地址 40:E 处插入一个 I/O 端口地址，可能会使 BIOS，或者其他软件，认为这就是扩展 BIOS 数据区的起点。这可能造成所有荒唐的结果以及造成系统挂起。

POST 并不检查是否存在第四个端口。带有第四个并行口的系统必须作出检查，并使用设备驱动程序、固件、或其他软件手段来装入该端口的地址。尽管尚未说明，但是打印机服务处理程序，中断 17h，通常会支持并行口 4 的打印机。众所周知的例外是 MCA 系统，它只支持三台打印机。

打印机 BIOS

使用打印机 BIOS 来访问任何并行口的打印机。打印机 BIOS 依赖于一些端口地址，这些端口地址已经确定并保存在 BIOS RAM 区地址 40:8 处，正如 POST 所作的那样。如果试图访问不存在的并行口（I/O 端口值为 0），那么服务程序仅仅返回而不改变任何寄存器的值。

三个打印机 BIOS 服务在 AH 中返回执行了指定操作后的打印机的状态。读取打印机状态 I/O 端口，并对 I/O 错误位 3 和确认位求反，来获得这个状态值。结果放在 AH 中。

例如，如果基本 I/O 端口是 278h，则从端口 279h 读取状态。如果 BIOS 超时，则返回状态 AH 的位 0 设置为 1。这个值不是从打印机状态 I/O 端口读取，但是 BIOS 会设置它。如果设置了位 0，则表示出现超时。

返回的状态定义如下：

位 7 = 0　打印机忙　这是并行口连接器引脚 11 线的求反。如果打印机正忙而不能接受数据，则该位是零。如果打印机正在执行打印操作、离线、或者出错时，打印机处于忙状态。

位 6 = 1　打印机确认　这是并行口连接器引脚 10 线的求反，由打印机控制。来自打印机的 0.5 μS 的短脉冲表示打印机准备好接收数据。如果开放了并行口中断，确认操作也会生成一个硬件中断。

位 5 = 1　缺纸　它直接读自并行口连接器引脚 12。打印机缺纸时该位被置高。参看警告部分了解可能的问题。

位 4 = 1　打印机选择　它直接读自并行口连接器引脚 13。选择了一个活动的打印机时该位被置高。

位 3 = 1　I/O 错误　这是并行口连接器引脚 15 线的求反。如果打印机离线、缺纸、或者检测到打印机错误，该位被设置为 1。

位 2 = x　未使用

位 1 = x　未使用

位 0 = 1　超时　打印一个字符时，打印机保持忙的时间超过了超时设置周期。与其他状态位不同，这一位由 BIOS 服务处理程序设置。如果激活了这一位，状态字节中的其他所有的位都无意义。这意味着在评价其他位之前必须检查位 0。超时周期随系统而有所不同，但是平均周期是 40 μS。在开始于 40:78h 处的 BIOS 数据区中指定了超时延迟周期。

中断 17h 提供以下三个服务：

功能	描述
ah=0	打印字符
ah=1	初始化打印机
ah=2	获取打印机状态

中断	功能	描述	平台
17h	0	打印字符	所有

打印字符功能将 AL 中的字符发送到指定的并行口中。如果打印机正忙，则 BIOS 等待超时周期，直到打印机准备好。如果准备好，则发送该字符。如果出现超时，则 BIOS 设置超时状态位。打印机状态返回在 AL 中。应该检查状态，确保试图打印时没有出错。

调用： ah = 0
返回： al = 要打印的字符
dx = 0~3，并行口号 1~4（并非总是支持端口 4）
ah = 打印机状态位（前面一节中作出了解释）

中断	功能	描述	平台
17h	1	初始化打印机	所有

这个功能初始化 DX 中指定并行口处打印机。打印机 BIOS 将并行口连接器引脚 16 设置为低并保持 50uS 以上，就会执行初始化。

调用： ah = 1
返回： dx = 0~3，并行口号 1~4（并非总是支持端口 4）
ah = 打印机状态位（前面一节中作出了解释）

中断	功能	描述	平台
17h	2	获取打印机状态	所有

获取打印机状态并保存在 AH 中。由于该功能只是读取连接器线上的状态，所以不会出现超时。

调用： ah = 2
dx = 0~3，并行口号 1~4（并非总是支持端口 4）
返回： ah = 打印机状态位（前面一节中作出了解释）

屏幕打印

系统 BIOS 提供了屏幕打印服务，中断 5，来向并行口附带的打印机打印屏幕的内容。屏幕打印通常由低级键盘 BIOS，中断 9 在按下 Print Screen（打印屏幕）键时调用。在打印屏幕操作期间，打印屏幕 BIOS 将光标移动到正在读取的字节处，并将该字符发送到打印机。屏幕打印完成时，恢复到初始光标位置。

在屏幕打印操作期间如果出现了错误条件，地址 50:0 处的屏幕打印状态字节设置为 FFh。然后退出屏幕打印 BIOS。错误条件包括打印机错误、打印机忙、或者打印机缺纸。打印时，50:0 处的状态字节设置为 1。如果正在执行前一个屏幕打印时出现了又一个屏幕打印请求，则忽略新请求。这意味着你可以简单的锁住所有的屏幕打印请求，方法是在地址 50:0 中装入值 1。

早期的 PC 和 XT 屏幕打印服务总是假定屏幕是 25 行的。AT+系统所使用的视频行数保存在 40:84 处的 BIOS 数据区字节中。为了支持屏幕打印在带有 EGA/VGA/XGA 适配器

的 PC 或 XT 上打印 25 行以上，可以使用一个新的屏幕打印处理程序，将它作为视频 BIOS 的一部分。该替代屏幕打印程序基本上和 AT+ BIOS 屏幕打印相同。视频屏幕打印版也检查 BIOS 数据区来获得视频的行数。要激活这种替代屏幕打印，可使用下面的代码：

```
mov     ah，12h                 ；替代选择
mov     bl，20h                 ；安装屏幕打印
int     10h                     ；视频服务
```

该替代屏幕打印有一个怪癖之处，它不检查是否有其他 TSR 或驱动程序挂起了中断 5。安装屏幕打印功能仅仅只是在中断向量中装入一个指针，指向视频 BIOS 内部新的屏幕打印程序。如果一个 TSR 或驱动程序以前挂起了中断 5，这样做就会引起问题。在屏幕打印之前或之后，TSR 将不再获得控制。AT+系统不需要替代屏幕打印，以避免这个可能有的问题。

屏幕打印程序并不直接从屏幕读取数据。屏幕打印使用视频 BIOS 中断 10h 功能 8 来读取屏幕，并检查从屏幕读取的字节是否为零。如果是，则将它转化为一个空格。本章后面的代码例 14-1 就是一个小型 TSR，如果打印机不能处理完整的 256 字符集，则它进一步将打印机限制到仅处理 ASCII 字符。

中断	描述	平台
5h	屏幕打印	所有

向并行口 1 附带的打印机打印当前视频屏幕的内容。在处理屏幕打印时，50:0 处的状态字节设置为 1。屏幕打印使用视频服务，中断 10h，来获取当前行和列信息，并显示内容。由屏幕打印服务程序，中断 17h 执行实际的屏幕打印。

调用：　无

返回：　更新 50:0 处的屏幕打印状态
0 = 完成
1 = 已经在处理或者禁止
FFh = 出现错误

BIOS 数据区

地址	描述	大小
40:08h	并行口 1	字

起始（base）并行 I/O 口号保存在这个字中。常用的 I/O 地址值是 3BCh、378h、或 278h，

但是任何并行 I/O 口都可以保存在这个字中。DOS 把它作为 PRN 或 LPT1 访问。

地址	描述	大小
40:0Ah	并行口 2	字

起始（base）并行 I/O 口号保存在这个字中。常用的 I/O 地址值是 378h、或 278h，但是任何并行 I/O 口都可以保存在这个字中。DOS 把它作为 LPT2 访问。

地址	描述	大小
40:0Ch	并行口 3	字

起始（base）并行 I/O 口号保存在这个字中。通常并行口 3 设置的 I/O 地址值是 278h，但是任何并行 I/O 口都可以保存在这个字中。DOS 把它作为 LPT3 访问。

地址	描述	大小
40:0Eh	并行口 4/扩展 BIOS 指针	字

在 AT+系统上使用扩展 BIOS 数据区时，该字保存了扩展 BIOS 数据区的段址。扩展 BIOS 数据区通常使用 639K 处的主存顶部 1KB 的 RAM。参看第 6 章了解这方面的更多信息。

当不支持扩展 BIOS 数据时（例如，在一个 AT 系统上），该字可用来保存第四个并行口的 I/O 地址。应用程序或驱动程序必须装入这个 I/O 端口地址。与并行口 1、2、以及 3 不同，POST 不设置这个值。DOS 把它作为 LPT4 访问。

地址	描述	大小
40:78h	并行口 1 超时	字节

这个字节保存了打印机 BIOS 中断 17h 所使用的一个固定值。在打印机 BIOS 向打印机输出一个字符时，该值代表打印机忙的最大延迟。如果在这段延迟周期之后打印机仍处于忙状态，则 BIOS 会返回一个失败的错误代码。

大多数系统的 POST 会将初始值设置为 14h。由于延迟是基于指令延迟的，该周期将随着系统的时钟速度而改变。所有的 AT+系统以及一些 XT BIOS 使用这个字节。对于值 14h，在不同的系统上超时时间在 20~50uS 范围内变化。

地址	描述	大小
40:79h	并行口 2 超时	字节

参看 RAM 地址 40:78h 下的描述。

地址	描述	大小
40:7Ah	并行口 3 超时	字节

参看 RAM 地址 40:78h 下的描述。

地址	描述	大小
40:7Bh	并行口 4 超时	字节

参看 RAM 地址 40:78h 下的描述。某些系统不支持第四个打印机。这时，没有使用该字节，或者可能用于其他不相关的用途。

地址	描述	大小
50:00h	屏幕打印状态	字节

在按下 Print Screen（打印屏幕）键时由低级键盘 BIOS 启动该屏幕打印功能。键盘 BIOS 触发中断 5，屏幕打印处理程序。该处理程序通过这个状态字节控制屏幕打印的状态。可以在 40:100h 和 50:0 处访问该字节。

该字节带有下列值和功能：

0 = 屏幕打印准备好
1 = 屏幕打印正在进行或禁止屏幕打印
FFh = 打印开始之前出现了错误：打印机忙或缺纸
在打印时出现了错误：打印机纸用完、或者打印机发出了一个错误信号、或者在等待打印机不忙时出现超时。

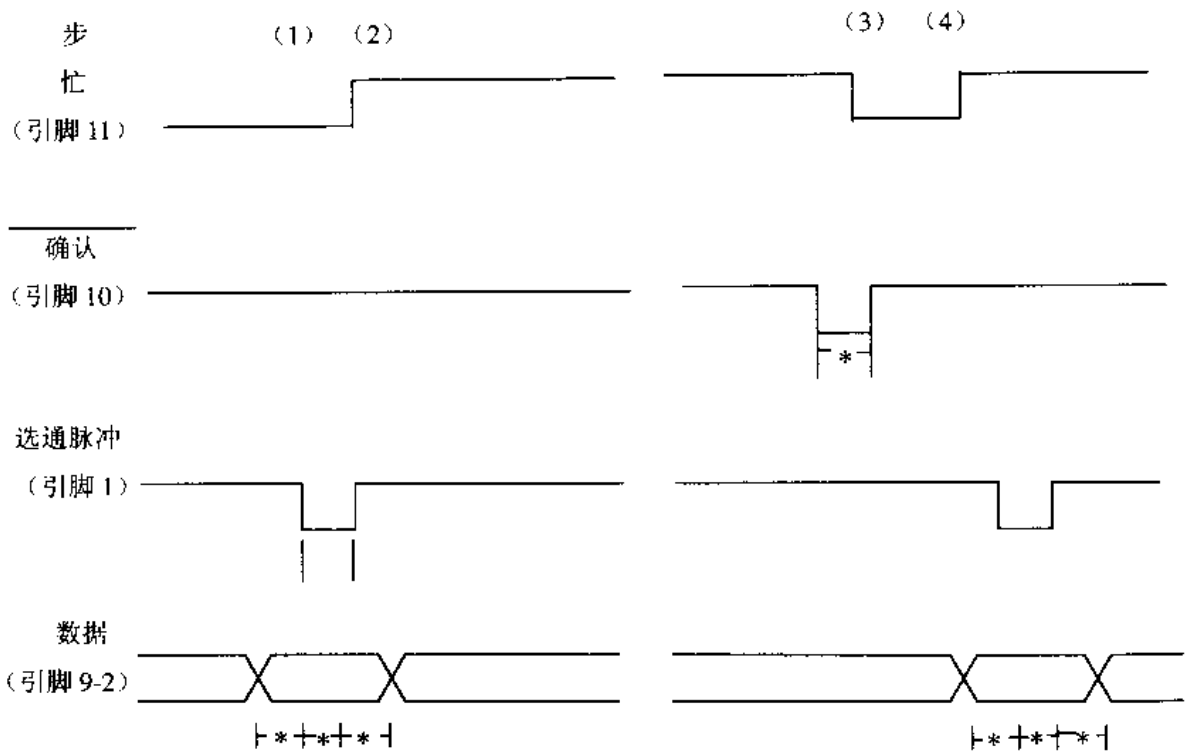
并行口定时

向一个并行口设备，通常是打印机，发送字节时，必须服从一个指定的定时序列。这些定时序列如图 14-2 所示。选通脉冲和数据线定时延迟在软件中执行。

如果忙线是低电平，则第一个字节的数据输出到端口（1）。在最大 0.5μS 的时间后，选通脉冲线设置为低电平，并维持至少 0.5μS（2）。使选通脉冲设置为低电平的操作会使相连的设备将忙线设置为高电平。一旦设备处理完该字节并准备好下一个字节时，设备将确认线设置为低电平并维持最少 0.5μS（3）。在确认线回到高电平状态时，设备将忙线降低为低电平（4）。这时，如果需要向设备发送其他字节，则开始下一个周期。

定时要求支持最好情况下每 2μS 传输一个字节。这要求设备立即响应确认线，在操作之间没有任何延迟。结果是，理论上的最大的传输速率是每秒 500K 字节。

通常传输速率要慢的多，原因是设备响应和 BIOS 定时存在延迟。BIOS 的定时部分取决于指令速度。为了保持定时最小，设计 BIOS 软件延迟，应考虑使可能最快的处理器也能正确地操作。为 20MHz~66MHz 范围的 CPU 设计 BIOS 时，应该在 20MHz 下加入延迟来确保 66MHz 的系统也可以起作用。一个 80486DX 当然要比 80386SX CPU 快得多，但是可以在两个系统中使用相同的 BIOS 代码。大多数用户都希望获得最佳传输速率，大约每秒 150K 字节。



*定时最小为 0.5μS。仅仅只有显示的定时才是关键的

图 14-2　软件定时

并行口连接器

在开发并行口的接口软件时，理解通过连接器传输哪条控制线是很有用的。并行口使用一个 25 针 'D' 壳式连接器。连接器的插脚引线如表 14-1 所示。方向是相对于系统的。这意味着一个输出向附带设备发送一个信号。在有关打印机 BIOS 的部分详细描述了功能。带减号前缀的功能在低电平时被激活，带加号前缀的功能在高电平时被激活。

表 14-1　并行口连接器功能

引　脚	描　述	功　能
1	输出*	-选通脉冲——低电平脉冲表示数据线上的数据有效
2	输出*	数据位 0
3	输出*	数据位 1
4	输出*	数据位 2
5	输出*	数据位 3
6	输出*	数据位 4

续表

引　脚	描　述	功　能
7	输出*	数据位 5
8	输出*	数据位 6
9	输出*	数据位 7
10	输入	-确认——打印机发送一个 0.5μS 的低电平脉冲来表示它不在忙
11	输入	+忙状态——高电平信号表示打印机正在执行打印操作、离线、或出现了错误。
12	输入	+纸完——高电平表示打印机的纸用完
13	输入	+选择状态——高电平表示目前选择这个打印机
14	输出	-自动走纸——低电平信号表示每次换行后打印机会自动回位。某些打印机不支持。
15	输入	-错误——低电平信号表示打印机离线、纸用完、或者出现了打印机错误
16	输出	-初始化——50μS 或更长的低电平信号会初始化打印机
17	输出	-选择——低电平信号选择这个打印机。并非所有的打印机都支持
18~25	地	

*这些引脚在扩展模式下用作输入。许多系统不支持扩展模式。

快速并行口

标准并行口设计的运行速度有限，因为某些定时是在软件中执行的。快速并行口设计极大地提高了传输速率，通常从 150K 字节/秒到 2M 字节/秒。通过在软件中自动执行数据选通操作来提高速度。

标准的并行口设计只能附带一个设备。快速并行口设计支持双向数据传输，在一个并行口连接器上可以附带多达 256 个设备。

在并行口与高速设备，例如外部磁盘和磁带驱动器，相连，并且在两个系统间传输数据时，快速端口特性很有用。快速端口特性对具有有限扩展功能的膝上型电脑最有益。

开放快速端口模式时，25 针连接器上有五个控制引脚的功能发生了改变。如表 14-2 所示。当关闭快速端口模式时，并行口的操作和标准的并行口相同。它是硬件重启后的状态。激活快速模式时，下面五个连接器引脚改变了功能。

表 14-2 快速模式连接器功能

引 脚	标 准 功 能	快 速 功 能
1	-选通脉冲	-写——输出线，指示引脚 2~9 上的数据是有效的输出数据
10	-确认	-中断请求——这条输入线上的低电平信号指示中断请求，如果启用此功能（enable）
11	+忙	-等待——输入线指示远程设备没有准备好
14	-自动进纸	-数据选通脉冲——这个输出线指示引脚 2~9 上的数据是有效的数据
17	-选择	-地址选通——低电平时，引脚 2~9 上的数据是设备的地址（支持不超过 256 个设备）

Intel 在 386SL 和 486SL 芯片组及其他新式的芯片组上支持快速并行口，它称之为增强型并行口（EPP）。IBMHW 论坛上的 Compuserve 提供了其他信息，请下载文件 PARA14.ZIP。

警 告

从并行口读取忙状态时，应连续读取该状态两次。这包括读取端口 279h、379h、以及 3BDh。在忙线没有连接，处于漂浮状态时，这一点增进了某些并行口的可靠性。忽略第一次读取的结果，使用第二次状态读取的值。

在某些老式的 Okidata 打印机上，在忙线从高电平变成低电平时，打印机会在缺纸信号线上发送一个 50μS 的假脉冲（glitch）。BIOS 缺纸或直接从端口 279h、379h、或 3BDh 读取状态时会出现这种情况。在声明缺纸情况之前，请再次检查状态，以确定是否真的缺纸。

代码例 14-1 只打印 ASCII

下面这个 TSR 可以阻止非 ASCII 字符送到打印机。此外，这个 TSR 将许多 IBM 的图形字符转化为可打印的字符。对于那些不支持完整的 IBM 字符集的打印机来说，这一点非常有用。

```
;--------------------------------------------------------------------
;                          PRNASCII
;--------------------------------------------------------------------
;
```

```
;    本 TSR 为那些不能打印全部 IBM 字符的
;    打印机将非 ASCII 字符转化成可接受的字符。
;    适用于大多数程序和屏幕打印
;
;    (c) 1994 Frank van Gilluwe 版权所有

include undocpc.inc


cseg    segment para public
        assume  cs:cseg, ds:cseg, ss:tsrstk


; This is the resident handler than converts characters to be
;   printed

int_17h_hook    proc    far
        pushf
        cmp     ah, 0                   ; 打印一个字符?
        jne     skip_change             ; 若否，则跳转

        cmp     al, 20h                 ; 控制字符?
        jb      control                 ; 若是，则跳转
        cmp     al, 7Fh                 ; 范围之内?
        jb      skip_change             ; 若是，则跳转
        je      set_space               ; 如果是 7Fh，则设置为空格

        sub     al, 80h                 ; 将大写字符转化成新字符
        push    bx                      ;
        mov     bx, offset new_chars
        xlat    cs:[bx]
        pop     bx
        jmp     finish_int17

control:
        cmp     al, 1Bh                 ; 溢出?
```

```
        je      skip_change         ; 若是，则跳转
        cmp     al, 08h             ; BS, HT, LF, VT, FF, CR ?
        jb      set_space           ; 若非，则跳转
        cmp     al, CR
        jbe     skip_change         ; 若是，不用管它
                                    ; 所有其他的字符都转化成空格
set_space:
        mov     al, ' '             ; 转化成空格

finish_int17:
skip_change:
        popf
        jmp     cs:old_int_17h      ; 处理老式中断 17h

int_17h_hook    endp

old_int_17h     dd      0           ; 此处贮存老式指针

new_chars       db 'CueaaaaceeeiiiAA'       ; 80-8F
                db 'E AooouuyOU      '      ;  90-9F
                db 'aiounN   ?==   !<>'     ;   A0-AF
                db '###|||   ||      '      ;   B0-BF
                db '   _|__|||   _|_|_'     ;   C0-CF
                db '___      ||   ##**#'    ;   D0-DF
                db '                 '      ;   E0-EF
                db '            ...  .'     ;   F0-FF

;=============================================================================
; 非驻留安装部分的起点

prnascii        proc    far

start:
        push    cs
        pop     ds
```

```
; 获取当前中断 17h 向量并保存

        mov     al, 17h
        mov     ah, 35h
        int     21h                     ; 获取当前中断 17h 指针
        mov     word ptr old_int_17h, bx
        mov     word ptr old_int_17h+2, es

; 安装我们新的中断例程

        mov     dx, offset int_17h_hook
        mov     al, 17h
        mov     ah, 25h
        int     21h                     ; 安装中断

        OUTMSG  msg1                    ; 输出安装信息

; 确定剩余驻留部分的大小（段落）
;   并成为一个 TSR

        mov     dx, (offset start - offset int_17h_hook) SHR 4
        add     dx, 11h                 ; PSP 大小加上 1 个段落
        mov     ax, 3100h               ; 退回到 DOS，成为一个 TSR
        int     21h
prnascii        endp

msg1    db      CR, LF
        db      'PRNASCII TSR installed'
        db      ' - Print non-ASCII as ASCII.'
        db      CR, LF, '$'
cseg    ends

;============================================ 堆栈 ========

srstk   segment para stack
        db      150 dup (0)
```

```
tsrstk    ends
          end     start
```

代码例 14-2 交换打印机

在带有两台打印机的系统上，这个程序交换打印机 1 和打印机 2。它不是一个 TSR，只是仅仅改变打印机 BIOS 的端口地址字和超时字节。运行这个程序两次可以恢复到原始状态。PRNSWAP 工作在硬件最低层，其他可以截获一个并行口的程序，例如网络重定向，可能不能正确的使用这个工具。

```
; ---------------------------------------------------------------------------
;                                  PRNSWAP
; ---------------------------------------------------------------------------
;
;    交换打印机1和2的并行口（非TSR）
;
;    (c) 1994，1996 Frank van Gilluwe版权所有
;
;    v2.0-修改了一些小错误，定时1的地址错

include undocpc.inc

cseg      segment para public
          assume  cs:cseg, ds:cseg

          org     100h


prnswap proc      near

start:
          push    cs
          pop     ds

          mov     ax, 40h
          mov     es, ax                    ; ES用作BIOS段址
```

```
        mov     dx, offset msg2         ; 仅1个端口的信息提示

        mov     ax, es:[0Ah]            ; 获取并口2地址
        cmp     ax, 0                   ; 并口2激活了么?
        je      exit                    ; 若非，则跳转

        xchg    ax, es:[8]              ; 和端口1交换
        mov     es:[0Ah], ax            ; 将老式端口1放入到端口2中

        mov     al, es:[79h]            ; 获取打印机定时2
        xchg    al, es:[78]             ; 和定时1交换
        mov     es:[78h], al            ; 在定时2中放入老式的定时1
        mov     dx, offset msg1         ; 端口交换完毕信息
exit:
        mov     ah, 9
        int     21h                     ; 显示信息
        mov     ah, 4Ch
        int     21h                     ; 退出

msg1    db      CR, LF, 'Printers 1 & 2 swapped.', CR, LF, '$'
msg2    db      CR, LF, 'Less than 2 active ports.', CR, LF, '$'

prnswap endp
cseg    ends
        end     start
```

代码例 14-3 检测是否支持扩展模式

该程序扫描四个可能的并行口，来查看某个并行口是否在物理上和系统相连接。对于所带的活动端口，显示 I/O 地址，并测试该端口来看这个端口是否支持扩展的双向模式。然后读写几种数据模式。在扩展读模式下，输出锁存没有连接到输入锁存上。如果不支持扩展模式，读操作将一直匹配写操作。运行 PRNTYPE 时，在屏幕上显示测试的结果。某个系统的测试结果如下：

PRNTYPE—并行口分析

并行口		活动?	扩展模式?
#1	03BC	是	否
#2	0378	是	是
#3	0278	是	是
#4		否	否

```
;-------------------------------------------------------------------------
;                                PRNTYPE
;-------------------------------------------------------------------------
;
;     检测是否存在活动的并口，并识别哪
;     些端口支持扩展模式。
;
;     (c) 1994 Frank van Gilluwe版权所有

include undocpc.inc

cseg    segment para public
        assume  cs:cseg, ds:cseg

        org     100h                    ; 汇编成COM文件


start:
        jmp     begin

msg1    db      CR, LF, CR, LF, TAB, TAB
        db      '   PRNTYPE - Parallel Port Analysis'
        db      CR, LF, CR, LF, TAB, TAB
        db      'Parallel Port    Active?    Extended Mode?'
        db      CR, LF, TAB, TAB
        db      '                '
        db      CR, LF, '$'

msg2    db      TAB, TAB, '#'
```

```
msgnum   db       '1        '
msghigh  dw       '  '
msglow   dw       '  '
         db                    '       '
msgon    db                          'No          '
msgExt   db                                      'No '
         db       CR, LF, '$'

msg3     db       TAB, TAB
msg3num  db       '#4  Likely Extended BIOS data area segment'
         db       CR, LF, '$'


prntype  proc     near

begin:
         push     cs
         pop      ds
         OUTMSG   msg1                    ; 显示初始信息

         mov      ax, 40h
         mov      es, ax                  ; ES用作BIOS段址

         mov      cl, '1'                 ; 要检查的打印机号
         mov      si, 0                   ; 打印地址索引

prnex_loop:
         mov      msgnum, cl              ; 插入打印机号
         mov      msghigh, '  '
         mov      msglow, '  '            ; 空白端口号
         MSGNO    msgon                   ; 假定未激活
         MSGNO    msgExt                  ; 假定不支持扩展

         mov      dx, es:[si+8]           ; 获取并口地址
         cmp      dx, 0                   ; 并口激活了吗?
         je       prnex_output            ; 若非，则跳转
```

```
        cmp     cl, '4'                 ; 最后一个端口
        jne     prnex_ok                ; 若非，则跳转
        cmp     dx, 1000h               ; 端口号过高?
        ja      prnex_bios              ; 若是，则扩展BIOS数据
prnex_ok:
        MSGYES  msgon                   ; 并口激活

        mov     bx, offset msghigh      ; 插入端口号
        mov     al, dh
        call    hex                     ; 转化成ASCII
        mov     bx, offset msglow
        mov     al, dl
        call    hex                     ; 转化成ASCII

                                        ; 检查是否支持扩展模式
        add     dx, 2                   ; 设置控制端口
        in      al, dx
        IODELAY
        mov     ah, al                  ; 保存值供以后恢复
        or      al, 20h
        out     dx, al                  ; 将扩展读位5设置为开

        sub     dx, 2                   ; dx = 读/写端口号
        mov     al, 55h
        out     dx, al                  ; 发送值55h
        IODELAY
        in      al, dx
        cmp     al, 55h                 ; 读取值和前面写的值匹配吗?
        jne     prnex_extended          ; 若非，则跳转

        mov     al, 0AAh
        out     dx, al                  ; 发送值AAh
        IODELAY
        in      al, dx
        cmp     al, 0AAh                ; 读取值和前面写入的值匹配吗?
        je      prnex_restore           ; 若是，则无扩展模式
```

```
prnex_extended:
        MSGYES   msgExt              ; 端口支持扩展模式

prnex_restore:
        add      dx, 2
        mov      al, ah
        out      dx, al              ; 恢复到初始状态
        jmp      prnex_output

prnex_bios:
        OUTMSG   msg3                ; 扩展BIOS数据区
        jmp      exit                ; 完成

prnex_output:
        OUTMSG   msg2                ; 显示端口线
        add      si, 2               ; 下一个索引
        inc      cl
        cmp      cl, '4'             ; 完成了吗?
        ja       exit                ; 若是，则跳转
        jmp      prnex_loop          ; 继续下一个端口

exit:
        mov      ah, 4Ch
        int      21h                 ; 退出
prntype endp
```

端口总结

下面的端口列表用于并行口。

端口	类型	功能	平台
278h	I/O	并行口数据	适配器
279h	输入	并行口状态	适配器
27Ah	I/O	并行口控制	适配器

27Ch	I/O	并行口数据（备用的）	适配器
27Dh	输入	并行口控制（备用的）	适配器
27Eh	I/O	并行口控制（备用的）	适配器
378h	I/O	并行口数据	适配器
379h	输入	并行口状态	适配器
37Ah	I/O	并行口控制	适配器
37Ch	I/O	并行口数据（备用的）	适配器
37Dh	输入	并行口控制（备用的）	适配器
37Eh	I/O	并行口控制（备用的）	适配器
3BCh	I/O	并行口数据	适配器
3BDh	输入	并行口状态	适配器
3BEh	I/O	并行口控制	适配器

端口细节

端口	类型	描述	平台
278h	I/O	并行口数据	适配器

该 I/O 地址端口是三个连续 I/O 地址中的一个，这三个地址控制并行口。取决于其他并行口的数目，POST 会将这个 I/O 地址分配为并行口 1、2 或 3。

该 I/O 端口用来从或向与并行口连接器相连的设备传输数据。某些新式的系统提供了另外的硬件来支持双向操作扩展模式。参看前面的并行口定时章节，来了解如何通过并行口发送数据。参看并行口连接器一节了解连接器引脚功能的定义。

在非扩展模式下，向该端口写会将数据送到并行口连接器。从该端口读会从并行口连接器获得数据，这个数据通常是上次写往该端口的数据。这一点支持确认锁存器是否正常工作。通过一系列的读和写操作，有可能测试出输出锁存器及其所带的电缆的数据线之间或线和地之间是否存在短路。

使用扩展模式时，数据输出锁存，但是如果控制端口的方向位设置为 0，则只发送到相关的并行口连接器。如果方向位是 0，从这个端口读取等价于一个非扩展模式下的读，从以前写入该端口的数据中读取数据。如果方向位设置为 1，则读这个端口的操作将从与该并行口相连的设备中读取字节。

I/O（位 0~7）数据字节

端口	类型	描述	平台
279h	输入	并行口状态	适配器

读取这个 I/O 地址，来获取相关并行口的状态。一次 I/O 读将从并行口连接器的许多线上输入当前的状态。参考并行口连接器一节，进一步了解连接器的状态位功能。参看有关

警告的章节获取其他信息。

输入（位 0~7）

位		
	7 r = 1	设备忙，+忙线求反，引脚 11
	6 r = 1	确认完成，来自-确认线，引脚 10
	5 r = 1	缺纸，来自+纸用完线，引脚 12（参看有关警告的章节了解缺纸问题。
	4 r = 1	设备选中，来自+选中线，引脚 13
	3 r = 0	错误，-错误线求反，引脚 15
	2 r = 0	接收到确认线后挂起中断（仅适用于支持扩展模式的并行口）
	1 r = x	未使用
	0 r = x	未使用

端口	类型	描述	平台
27Ah	I/O	并行口控制	适配器

使用这个端口写并行口连接器控制引脚。该端口也控制中断操作和扩展模式的方向。不支持控制模式的系统没有使用方向控制位 5。参看有关并行口连接器的章节了解连接器引脚的功能定义。

开放一个并行口中断时，确认脉冲起点会触发一个硬件中断。通常有跳线来选择使用哪个硬件中断请求，IRQ 7、5 还是 2。可能也提供了其他选中，这与具体的卡有关。包括并行口的新式芯片组，常常允许用户使用系统 BIOS 设置工具，来指定由哪个 IRQ 与每个端口相关联。

在组合式的 DMA 并行口适配器和主板上带有并行口的老式系统上，这个中断请求固定为 IRQ 7。系统 BIOS 不使用任何中断来执行并行口操作。

读取这个端口时，位 0~3 可以直接从连接器引脚读取，它们应该反映上次写入位 0~3 的值。这一点支持确认锁存器是否正常工作。通过一系列的读和写操作，有可能测试出输出锁存器及其所带的电缆的数据线之间或线和地之间是否存在短路。

输入（位 0~7）

位		
	7 r = x	未使用
	6 r = x	未使用
	5 r = x	未使用
	4 r = 1	开放并行口中断
	3 r = 1	选择打印机（引脚 17 求反）
	2 r = 0	初始化打印机（引脚 16）
	1 r = 1	自动进纸（引脚 14 求反）
	0 r = 1	选通脉冲（引脚 1 求反）

输入（位0~7）

位		
	7 w = x	未使用
	6 w = x	未使用
	5 w = x	扩展模式方向控制
	0	数据写到连接器
	1	数据读自连接器，写锁存器与连接器断开
	4 w = 1	开放并行口中断
	3 w = 1	选择打印机（引脚17求反）
	2 w = 0	初始化打印机（引脚16）
	1 w = 1	自动进纸（引脚14求反）
	0 w = 1	选通脉冲（引脚1求反）

端口	类型	描述	平台
27Ch	I/O	并行口数据	适配器

很少使用这个端口，它可以替代I/O地址278h。

端口	类型	描述	平台
27Dh	输入	并行口状态	适配器

很少使用这个端口，它可以替代I/O地址279h。

端口	类型	描述	平台
27Eh	I/O	并行口控制	适配器

很少使用这个端口，它可以替代I/O地址27Ah。

端口	类型	描述	平台
378h	I/O	并行口数据	适配器

该I/O地址端口是三个连续I/O地址中的一个，这三个地址控制并行口。取决于其他并行口的数目，POST会将这个I/O地址分配为并行口1或2。

参看并行口I/O地址378h了解功能细节。

端口	类型	描述	平台
379h	输入	并行口状态	适配器

参看并行口I/O地址279h了解功能细节。

端口	类型	描述	平台
37Ah	I/O	并行口控制	适配器

参看并行口 I/O 地址 27Ah 了解功能细节。

端口	类型	描述	平台
37Ch	I/O	并行口数据	适配器

很少使用这个端口，它可以替代 I/O 地址 378h。

端口	类型	描述	平台
37Dh	输入	并行口状态	适配器

用这个端口，它可以替代 I/O 地址 379h。

端口	类型	描述	平台
37Eh	I/O	并行口控制	适配器

很少使用这个端口，它可以替代 I/O 地址 37Ah。

端口	类型	描述	平台
3BCh	I/O	并行口数据	适配器

该 I/O 地址端口是三个连续 I/O 地址中的一个，这三个地址控制并行口。如果存在，POST 总是将这个 I/O 地址分配为并行口 1。

参看并行口 I/O 地址 278h 了解功能细节。

端口	类型	描述	平台
3BDh	输入	并行口状态	适配器

参看并行口 I/O 地址 279h 了解功能细节。

端口	类型	描述	平台
3BEh	I/O	并行口控制	适配器

参看并行口 I/O 地址 27Ah 了解功能细节

第 15 章

CMOS 内存和实时时钟

本章包含了广泛的未公开信息，也包括大量的有关标准 CMOS 寄存器的信息。许多 CMOS 寄存器从未公开过，然而却包含了有用的信息。此外，你还可以详细了解一些扩展 CMOS 功能，在新式的 AT、MCA、以及 EISA 系统上它们鲜为人知。

提供了程序例来可靠的读和写 CMOS 内存。此外，CMOSVIEW 工具程序标记并显示普通和扩展 CMOS 内存寄存器的内容。

实时时钟是 CMOS 内存芯片的一部分，对它也作出了详细的解释。还详细分析了与时钟有关的一些问题，这些问题可能会使胆大的程序员编写不稳定的代码。

简　介

系统带有一个小型的 CMOS 内存，其中保存着系统断电时的数据。该内存用来记录软盘的类型和数目、硬盘大小信息、内存大小以及其他重要的系统数据。CMOS 芯片还含有一个实时时钟（RTC），用来保持当前时间。早期的 AT 使用一个 Motorola MCI 146818A 实时时钟 IC。

关掉电源时，RTC 由计算机内部的电池供电。电池保持时钟处于活动状态，以及保留 CMOS 内存内容。如果在断电时电池无法正常供电，将丢失 CMOS 内容。在 BIOS 系统上电自检（POST）期间，大多数 BIOS 会识别出这一点。

可以直接或者通过实时时钟 BIOS 访问时钟寄存器。对于访问 CMOS 内存或 RTC 上的四个状态和控制寄存器，没有公开的 BIOS 服务。本章后面描述了如何通过 I/O 端口来直接访问 CMOS 内存。IBM 的 PS/2 提供了对 CMOS 的内存访问，但是这些内容没有公开过。请参看第 13 章，了解有关细节。

实时时钟（RTC）的常用信息

有十个寄存器用来访问 RTC 时间和数据。所有的时钟寄存器采用 BCD（二进制编码的十进制）格式。另外四个寄存器用来保存时钟、闹钟和 CMOS 内存的状态和控制信息。状

态和控制寄存器称作寄存器 A、B、C 和 D。在本章后面的端口 70h 的描述中详细描述了所有的 RTC 寄存器。

RTC 有许多古怪之处，要求专门的编程来保证访问可靠。使用下一部分描述的 RTC BIOS 来避免这些令人头疼的问题，我极力推荐这样做。

通过 I/O 端口 70h 装入任何数据或时间寄存器时，必须停止时钟。这一点可以防止在设置新的日期或时间期间更新寄存器。要停止时钟，可以将寄存器 B 的第 7 位设置为 1。更新完成时，将寄存器 B 的第 7 位设置为 0 来激活时钟。

保存一个新的日期和时间时，不要在该月最后一天的日期将要翻转的 2 秒内设置 RTC。在这个时刻，如果装入了新值，月尾改变日期时的硬件逻辑处理可能会不正确的改变这个新值。如果准备激活夏令制时间，就不要在夏令制改变的前两秒内设置时钟，否则可能会不正确的设置内部计数器。RTC 使用四月的最后一个星期天来切换到夏令制时间。当前美国的夏令制时间开始于四月的第一个星期天。这一点使夏令制时间选项一点用处也没有。

不幸的是，读取时钟值比改变它们要棘手些。是什么使我们的编程生活如此舒适？RTC 芯片有一个每秒一次的内部更新周期，更新占用大约 2ms。在内部更新期间，从时间、日期、和闹钟寄存器读将会获得意想不到的数据。如果你只想追求最好，那么每 500 次尝试将会有一次是坏数据。

在读取时钟时你当然希望获得有效的值。为了实现这一点，请关闭中断，进入一个密循环读取寄存器 A 在位 7 上的更新。当从高电平向低电平过渡时，你有 244 微秒的时间来读取你想要的时间寄存器。为了避免这些问题，可以使用实时时钟 BIOS 功能来读和写时钟值。

实时时钟 BIOS

使用 BIOS 中断 1Ah、日期时间服务、可以读或写 CMOS 时钟和闹钟寄存器。建议通过中断 1Ah 来访问时钟服务，而不要试图直接对 CMOS 时间寄存器编程。

功能 0 和 1 用来访问 BIOS 时钟滴答计数。BIOS 时钟滴答计数器与 CMOS 时钟无关，因为它随着系统时钟而增加。参看第 16 章了解更多系统时钟的信息。

中断 1Ah 提供下列服务：

功能	描述
ah=0	从 BIOS RAM 中获取当前的时钟计数
ah=1	在 BIOS RAM 中设置当前的时钟计数
ah=2	读 CMOS 时间
ah=3	设置 CMOS 时间
ah=4	读 CMOS 日期
ah=5	设置 CMOS 日期

ah=6	设置 CMOS 中的闹钟时间
ah=7	关闭闹钟
ah=8	设置激活上电的 RTC
ah=9	读取 RTC 闹钟时间和状态
ah=A	读取从 1980 年开始算的天数
ah=B	设置从 1980 年开始算的天数

中断	功能	描述	平台
1Ah	0	获取当前时钟计数	AT+

这个功能从 BIOS RAM 位于 40:6Ch 处的双字中读取时钟滴答计数。该功能也返回时钟滴答的 24 小时翻转标志，该标志保存在 BIOS RAM 地址 40:70h 处的字节中。在读取翻转标志会清除它。

只有操作系统才能使用这个调用。如果一个应用程序在午夜后不久立即使用这个调用，则读计数的操作会清除 24 小时翻转标志。接着，在操作系统读取该信息时，翻转标志已经被错误的设置为零了。结果是，操作系统时钟会丢失一天。许多书籍建议使用这个子功能，却没有意识到这个严重的问题。

我们强烈建议，直接从 BIOS RAM 区读取该值。这样做是可靠的，而且可以避免和操作系统发生冲突。在读取计数和翻转标志时记住要禁止中断。这样做可以防止时钟滴答中断在读取一部分信息期间更新计数和翻转标志。

调用：　ah = 0

返回：　ax = 修改
　　cx = 时钟滴答计数的高位字
　　dx = 时钟滴答计数的低位字
　　al = 0　如果从上次读算起，计数没有超过 24 小时
　　1　如果计数超过 24 小时（读取后会清除这个 24 小时溢出标志）

中断	功能	描述	平台
1Ah	1	设置当前时钟计数	AT+

这个功能向 BIOS RAM 位于 40:6Ch 处的双字中装入指定的时钟滴答计数。该功能也清除时钟滴答的 24 小时溢出标志，该标志保存在 BIOS RAM 地址 40:70h 处的字节中。

调用：　ah = 1
　　cx = 时钟滴答计数的高位字
　　dx = 时钟滴答计数的低位字

返回：　ax = 修改

中断	功能	描述	平台
1Ah	2	读取 CMOS 时间	AT+

直接从 CMOS 实时时钟中读取当前的时间。读取成功后会清除进位标志。

实时时钟 BIOS 检查是否正在执行内部 CMOS 时钟更新。取决于 BIOS 所采用的技术，如果正在执行更新，则这个功能可能仅仅只是退出而不获取时间。大约每 500 次可能就有一次出现这种情况。如果在返回时设置了进位标志，再读一次通常可以获得成功的结果。

调用：　ah = 2

返回：　如果进位=0

ax=修改

ch=BCD 形式的小时数，0~23（假设缺省设置是 24 小时模式）

cl=BCD 形式的分钟数，0~59

dh=BCD 形式的秒数，0~59

dl=0 如果禁止夏令制时间

1 如果开放夏令制时间（参看有关 RTC 的一般信息这一部分，了解有关夏令制时间的问题）

如果进位=1

ax=修改

正在更新时钟，或者没有运行时钟

中断	功能	描述	平台
1Ah	3	设置 CMOS 时间	AT+

直接向 CMOS 实时时钟写入指定的时间。与设置时钟滴答（功能 1）不同，这个功能永久性的更新了时间。

调用：　ah = 3

ch = BCD 形式的小时数，0~23（假设缺省设置是 24 小时模式）

cl = BCD 形式的分钟数，0~59

dh = BCD 形式的秒数，0~59

dl = 0　禁止夏令制时间

1　如果开放夏令制时间（参看有关 RTC 的一般信息这一部分，了解有关夏令制时间的问题）

返回：　进位 = 0

ax = 修改

中断	功能	描述	平台
1Ah	4	读取 CMOS 日期	AT+

直接从CMOS实时时钟中读取当前的时间，读取成功后会清除进位标志。

实时时钟BIOS检查是否正在执行内部CMOS时钟更新。取决于BIOS所采用的技术，如果正在执行更新，则这个功能可能仅仅只是退出而不获取日期。大约每500次可能就有一次出现这种情况。如果在返回时设置了进位标志，再读一次通常可以获得成功的结果。

调用：ah = 4

返回：如果进位= 0

ax=修改

ch=BCD形式的世纪，19或20

cl=BCD形式的年，0~99

dh=BCD形式的月，0~12

dl=BCD形式的天，0~31

如果进位=1

ax = 修改

正在更新时钟，或者没有运行时钟

中断	功能	描述	平台
1Ah	5	设置CMOS日期	AT+

直接向CMOS实时时钟写入指定的日期。

调用：ah = 5

ch = BCD形式的世纪，19或20

cl = BCD形式的年，0~99

dh = BCD形式的月，0~12

dl = BCD形式的天，0~31

返回：进位 = 0

ax = 修改

中断	功能	描述	平台
1Ah	6	设置CMOS中的闹钟时间	AT+

CMOS RTC有一个内置的闹钟。在每天当前时间和闹钟时间相等时，就调用中断70h。使用这个闹钟特性时，首先使用功能6来设置闹钟时间。如果进位标志被清除，则表示设置成功，然后中断4Ah被引导到用户提供的闹钟处理程序。如果设置失败，设置了进位标志，则错误条件指示不可以使用闹钟。由于这个功能一次只支持一个使用者，所以永远也不要重设其他程序在使用的闹钟。

在闹钟使用完成时，使用中断4Ah，功能7来禁止闹钟。将中断4Ah向量恢复为其初始值不失为一种好习惯。

每个闹钟寄存器都有一个特殊的不管模式，向一个寄存器装入任何从 C0h 到 FFh 的值，在确定一个闹钟时间时会忽略该寄存器。例如，要使在本小时和每小时后 15 分钟时都出现闹钟，可以在小时寄存器 CH 中装入 FFh，在分钟寄存器 CL 中装入 15h，而在秒钟寄存器 DH 中装入 0。

调用：　ah = 6
　　ch = BCD 形式的小时，0~23，或 FFh
　　cl = BCD 形式的分钟，0~59，或 FFh
　　dh = BCD 形式的秒，0~59，或 FFh
返回：　如果进位 = 0
　　ax = 0
　　如果进位 = 1
　　ax = 修改
　　已经开放了闹钟。这一点表示另外一个应用程序正在使用闹钟、时钟正在更新、或者没有运行时钟

中断	功能	描述	平台
1Ah	7	关闭闹钟	AT+

禁止闹钟功能。参看中断 1Ah，功能 6，来设置闹钟。

调用：　无
返回：　进位 = 0
　　ax = 修改

中断	功能	描述	平台
1Ah	8	设置激活上电的 RTC	可选型 PC

设置断电后激活膝上型电脑上电的时间。

调用：　ah = 8
　　ch = BCD 形式的小时，0~23
　　cl = BCD 形式的分钟，0~59
　　dh = BCD 形式的秒，0~59
返回：　进位 = 0　如果成功

中断	功能	描述	平台
1Ah	9	读取 RTC 闹钟时间和状态	有限

读取 RTC 时间。仅可逆型 PC 和 PS/2 型号为 30 的才支持该功能。可逆型 PC 也指示一个活动的闹钟是否会激活系统的上电操作。

调用：　ah = 9
返回：　ch = BCD 形式的小时，0~23
　　cl = BCD 形式的分钟，0~59
　　dh = BCD 形式的秒，0~59
　　dl = 0 如果没有开放闹钟
　　1 如果开放了闹钟
　　2 如果开放了闹钟，将对系统上电。

中断	功能	描述	平台
1Ah	Ah	读取从 1980 年开始算起的天数	XT/MCA

获取从 1980 年一月一日开始算起的天数。仅带有 1/10/80 及以后 BIOS 的 IBM XT 才支持这个功能。AT 不支持这个功能，但是所有的 PS/2 和 MCA 总线机器都支持这个功能。

调用：　ah = 0Ah
返回：　cx = 从 1980 年开始算起的天数

中断	功能	描述	平台
1Ah	Bh	设置从 1980 年开始算起的天数	XT/MCA

设置从 1980 年一月一日开始算起的天数。仅带有 6/10/85 及以后 BIOS 的 XT 才支持这个功能。AT 不支持这个功能，但是所有的 PS/2 和 MCA 总线机器都支持这个功能。

调用：　ah = 0Bh
　　cx = 从 1980 年开始算起的天数
返回：　无

EISA 系统的不同之处

EISA 系统含有另外的 8K CMOS 内存来保存特殊的 BIOS 信息以及插槽配置数据。访问 CMOS 内存的确切方法留给制造商去解决（因为有太多的标准）。大多数制造商采用 32 组 256 个寄存器。端口部分的端口 8xxh 和 C00h 阐释了这一点。

保存在扩展 CMOS 中的信息通常由专门的配置程序来处理，这些配置程序对 CMOS 进行读和写操作。配置程序使用三个 EISA 中断服务来读和写扩展 CMOS 内存中的信息。

中断	功能	描述	平台
15h	D800h	读取插槽配置信息	EISA

该功能读取某个指定的主板插槽的配置信息。该功能也指示了子功能的数目，这些子功能利用功能 AX=D801h 获取附加的配置信息。

调用：
ax = D800h
cl = 主板插槽号（1=插槽 1）

返回：
进位 = 1 无法获取信息
进位 = 0 成功
ah = 0 获取信息成功
al = 0 主板 ID 可读，不可复写
bx = 配置工具的改进等级
cx = 配置文件的校验求和
dh = 这个主板上的功能数（参看下面的读功能）
dl = 功能信息字节
di,si = 主板 ID，4 字节

中断	功能	描述	平台
15h	D801h	读取功能配置信息	EISA

每个板都有重要的附加信息，可以使用这个功能来读取，并在 CH 中指定子功能。

调用：
ax = D801h
cl = 板插槽号（1 = 插槽 1）
ch = 子功能号
0 = I/O 端口要求
1 = ROM、IRQ 和 DMA 信息
3 = IRQ 和 I/O 端口信息
ds:si = 指针，指向数据写入的地址（缓冲区至少有 320 字节）

返回：
进位 = 1 不能获取信息
进位 = 0 成功
ah = 0 成功的获取了信息
ds:si 缓冲区现在保存了所请求的信息

这里归纳了所返回的缓冲区的内容。每个子功能返回相同类型的初始信息。所有的子功能来源于 EISA 规范 v3.12。该文档有一些不一致的地方，有可能是一个错误。使用这些信息时请小心。进一步的细节请参考最近的 EISA 规范。

所有的子功能如下：

偏移量	字节数	描述
0h	2	主板 ID，第一和第二个字节
2h	1	产品号，第一和第二个十六进制数字
3h	1	产品号，第三个数字和改进信息
4h	1	ID 及插槽信息
5h	1	ID 信息，多种
6h	1	配置工具改进等级，主
7h	1	配置工具改进等级，次
8h	1	第一选择
9h	1	第二选则
Ah	1	第三选择（某些子功能没有使用）
Bh	1	第四选择（某些子功能没有使用）
Ch	1	第五选择（某些子功能没有使用）
Dh	21	未使用
21h	1	功能信息
23h	80	ASCII 类型和子类型串（参看下表）

ASCII 类型信息

设备类型	描述
COM，ASY	串口，ISA，兼容 8250
COM，ASY，FIFO	串口，ISA，兼容 16550（FIFO）
COM，SYN	SDLC 端口，ISA 兼容
KEY,xxx,KBD=yy	键盘，xxx=键的数目（083/084/101/103） yy=两个字母的国家代码

AE=阿拉伯，英语	IS=以色列
AF=阿拉伯，法语	A=日本汉字
AU=澳大利亚	LA=拉美
BE=比利时	MA=中东
BF=比利时，佛兰德语	NE=荷兰
CE=加拿大，英语	NO=挪威
CF=加拿大，法语	PO=波兰
CH=中国	SP=西班牙
DN=丹麦	SW=瑞典
DU=荷兰	ST=瑞士
EE=欧洲英语	SF=瑞士，法语
FN=芬兰	SG=瑞士，德语
FR=法国	TA=台湾
GR=德国	UK=英国

	HA=匈牙利	US=美国
	IT=意大利	
CPU,80386 SX	386 微处理器，SX 类型	
CPU,80386	386 微处理器	
CPU,80486SX	486 微处理器，SX 类型	
CPU,80486	486 微处理器	
CPU,PENTIUM	Pentium 微处理器	
JOY	游戏杆	
MEM	系统内存	
MSD,DSKCTL	硬盘控制器，ISA 兼容	
MSD,FPYCTL	软盘控制器，ISA 兼容	
MSD,TAPCTL	磁带控制器，ISA 兼容	
NET,ETH	网络，以太网	
NET,TKR	网络，令牌网	
NET,ARC	网络，环型网	
NPX,387	数学协处理器，387	
NPX,387SX	数学协处理器，387，SX 类型	
NPX,W1167	数学协处理器，Weitek 1167	
NPX,W3167	数学协处理器，Weitek 3167	
OSE	操作系统或环境	
PAR	并行口，ISA 兼容	
PAR,BID	并行口，双向	
PTR,8042	鼠标控制器	
SYS	主板	
VID,MDA	视频，单色	
VID,MDA,MGA	视频，单色，Hercules 图形	
VID,CGA	视频，CGA，回扫时不需要写同步	
VID,CGA,RTR	视频，CGA，回扫时需要写同步	
VID,EGA	视频，EGA	
VID,VGA	视频，VEA	

子功能 0，缓冲区的次级部分

偏移量	字节	描述
73h	91	未使用
104h	1	IOPORT 1 初始化字节
105h	2	IOPORT 1 地址，16 位
107h	1	IOPORT 1 值，8 位
108h	1	IOPORT 2 初始化字节

109h	2	IOPORT 2 地址，16 位
10Bh	2	IOPORT 2 值，16 位
10Dh	2	IOPORT 2 屏蔽值，16 位
10Fh	1	IOPORT 3 初始化字节
110h	2	IOPORT 3 地址，16 位
112h	1	IOPORT 3 值，8 位
113h	1	IOPORT 3 屏蔽值，8 位
114h	1	IOPORT 4 初始化字节
115h	2	IOPORT 4 地址，16 位
117h	1	IOPORT 4 值，8 位
118h	1	IOPORT 4 屏蔽值，8 位
119h	1	IOPORT 5 初始化字节
11Ah	2	IOPORT 5 地址，16 位
11Ch	1	IOPORT 5 屏蔽值，8 位
11Dh	1	IOPORT 6 初始化字节
11Eh	2	IOPORT 6 地址，16 位
120h	1	IOPORT 6 值，8 位
121h	1	IOPORT 6 屏蔽值，8 位
122h	1	IOPORT 7 初始化字节
123h	2	IOPORT 7 地址，16 位
125h	1	IOPORT 7 值，8 位
126h	1	IOPORT 7 屏蔽值，8 位

子功能 1，缓冲区的次级部分

偏移量	字节	描述
73h	1	内存配置（ROM=18h）
74h	1	ROM 内存大小
75h	3	ROM 起始地址（在段中的）
78h	2	ROM 大小/1024
7Ah	56	未使用
B2h	1	IRQ 号（值与 20h 相或）
B3h	1	保留
B4h	12	未使用
C0h	1	DMA 通道号
C1h	1	DMA 模式类型

子功能 2，缓冲区的次级部分

偏移量	字节	描述
73h	1	内存配置（ROM=19h）
74h	1	RAM 内存大小
75h	3	RAM 起始地址（在段中的）
78h	2	RAM 大小/1024

子功能 3，缓冲区的次级部分

偏移量	字节	描述
B2h	1	IRQ 号（值与 20h 相或）
B3h	1	保留
B4h	20	未使用
C0h	1	端口 IO 范围入口
C1h	2	端口地址，16 位

中断	功能	描述	平台
15h	D803h	写功能配置信息	EISA

向主板上的扩展 CMOS 内存区写配置信息。这个功能传递的缓冲区必须包含要写入到 CMOS 的数据。插槽在缓冲区中指定为某个值。缓冲区的长度是可变的，这取决于缓冲区中可变长度域的数目。

调用：
ax = D803h
cx = 41h
ds:si=指针，指向读取数据的地点

返回：
如果成功（即：保存了信息）
进位 = 0
ah = 0
如果失败
进位 = 1

许多指定的字节指示了相连部分的长度，因此没有显示偏移量。

字节	子功能	描述
2		底板 ID，第一个和第二个字节
1		产品号，第一和第二个十六进制数字
1		产品号，第三个数字和改进信息
1		ID 及插槽信息
1		保留

1		配置工具改进等级，主
1		配置工具改进等级，次
2		功能 0 入口的长度
1	0	下一个选择项的长度，字节
1	0	第一选择
1	0	第二选择
?	0	附加选择
1	0	功能 0 的信息字节（21h）
1	0	下面 ASCII 类型串的长度
?	0	ASCII 类型和子类型串（参看前一个表）
1	0	IOPORT 1 初始化字节
2	0	IOPORT 1 地址，16 位
1	0	IOPORT 1 值，8 位
1	0	IOPORT 2 初始化字节
2	0	IOPORT 2 地址，16 位
2	0	IOPORT 2 值，16 位
2	0	IOPORT 2 屏蔽值，16 位
1	0	IOPORT 3 初始化字节
2	0	IOPORT 3 地址，16 位
1	0	IOPORT 3 值，8 位
1	0	IOPORT 3 屏蔽值，8 位
1	0	IOPORT 4 初始化字节
2	0	IOPORT 4 地址，16 位
1	0	IOPORT 4 值，8 位
1	0	IOPORT 4 屏蔽值，8 位
1	0	IOPORT 5 初始化字节
2	0	IOPORT 5 地址，16 位
1	0	IOPORT 5 屏蔽值，8 位
1	0	IOPORT 6 初始化字节
2	0	IOPORT 6 地址，16 位
1	0	IOPORT 6 值，8 位
1	0	IOPORT 6 屏蔽值，8 位
1	0	IOPORT 7 初始化字节
2	0	IOPORT 7 地址，16 位
1	0	IOPORT 7 值，8 位
1	0	IOPORT 7 屏蔽值，8 位
2		功能 1 的入口长度（从下一个字节开始，直到功能 2 的入口）
1	1	下一个选择域的长度，字节

1	1	第一选择
1	1	第二选择
?	1	附加选择
1	1	功能 1 的信息字节（0Fh）
1	1	下面 ASCII 类型串的长度
?	1	ASCII 类型和子类型串（参看前一个表）
1	1	ROM 内存配置（18h）
1	1	ROM 内存大小
3	1	ROM 起始地址（在段中的）
2	1	ROM 大小/1024
1	1	IRQ 号（值与 20h 相或）
1	1	保留
1	1	DMA 通道号
1	1	DMA 模式类型
2		功能 2 的入口长度（从下一个字节开始，直到功能 2 的入口）
1	2	下一个选择域的长度，字节
1	2	第一选择
1	2	第二选择
?	2	附加选择
1	2	功能 2 的信息字节（03h）
1	2	下面 ASCII 类型串的长度
?	2	ASCII 类型和子类型串（参看前一个表）
1	2	RAM 内存配置（19h）
1	2	RAM 内存大小
3	2	RAM 起始地址（在段中的）
2	2	RAM 大小/1024
2		功能 3 的入口长度（从下一个字节开始，直到结束）
1	3	下一个选择域的长度，字节
1	3	第一选择
1	3	第二选择
?	3	附加选择
1	3	功能 3 的信息字节（0Fh）
1	3	下面 ASCII 类型串的长度
?	3	ASCII 类型和子类型串（参看前一个表）
1	3	IRQ 号（值与 20h 相或）
1	3	保留
2	3	端口 IO 范围入口

2	3	端口地址，16 位
2		上一个功能的长度（固定为 0）
2		配置文件求和校验，16 位

系统数据区

在 POST 期间要读取许多 CMOS 寄存器。某些信息从 CMOS 寄存器拷贝到 BIOS 数据区，该数据区的段址是 40h。参看第 6 章获取这些 BIOS 数据区的完整列表。

扩展 CMOS 寄存器

目前许多系统包括的寄存器比 AT 兼容机所提供的原始 64 个寄存器要多。当前使用它们的常见方法可以分为三种，新式 AT 兼容机、MCA、及 EISA 系统。CMOSVIEW 程序显示了所有标准的 64 个 CMOS 寄存器和其他检测到的扩展 CMOS 寄存器。在本章后面的代码例中显示了 CMOSVIEW 的关键部分。

AT 扩展 CMOS 内存 许多 AT 兼容机现在提供了另外 64 个寄存器。访问它们的方法和前 64 个寄存器相同。端口 70h 中装入一个 40h~7Fh 之间的地址，来访问这些附加的寄存器。检查许多支持 128 个寄存器的系统，发现它们通常用来保存一些信息，这些信息有关第三个和第四个驱动器，以及主板芯片组的专用数据项。

MCA 扩展 CMOS 内存 大多数 MCA 系统包括了另外 2K 的 CMOS 内存。它用来保存插槽配置信息。其中有些信息直接复制于常规 CMOS 内存，例如卡插槽 0~3 的可编程选项信息。参看本章后面所描述的端口 74h、75h 以及 76h 来访问这些内存，及其他进一步的细节。

EISA 扩展 CMOS 内存 EISA 标准为系统和插槽配置信息定义了另外一个 8K 的 CMOS。参看本章后面描述的端口 8xxh 和 C00h 来访问这个内存。

警 告

中断 在访问 CMOS 寄存器时必须禁止所有的中断。访问一个 CMOS 寄存器时，一个到端口 70h 的输出选择使用哪个寄存器。然后通过端口 71h 读或写这个寄存器。如果在两个 I/O 端口访问之间出现了中断，并且中断处理程序会访问 CMOS，你的代码将会读或写错误的寄存器。

除了禁止中断之外，还要禁止不可屏蔽中断（NMI）。NMI 主要用于陷入 RAM 奇偶错误和数学协处理器错误。在访问 CMOS 时很可能出现这些情况。不幸的是，NMI 禁止是一

个只写标志，所以前面状态的无法知道。每个人常用的方法是假定 NMI 是开放的，这就意味着在访问 CMOS 后总会开放 NMI。

在观察访问一个 CMOS 寄存器的步骤时，你必须首先使用 CLI 指令禁止中断。选择 CMOS 寄存器时应禁止了 NMI。恰巧由端口 70h 的位 7 控制 NMI，这个端口和指定 CMOS 寄存器的端口相同！CMOS 访问完成时，再次写端口 70h，并清除位 7，来重新开放 NMI。然后使用 STI 指令来开放一般的中断。在下面的程序 READ_CMOS 和 WRITE_CMOS 中我考虑了以上这些问题。

CMOS 求和校验 大多数的 BIOS 供应商对 10h 以上的 CMOS 寄存器采用求和校验。这用来检测 CMOS 内存问题和对 CMOS 寄存器可能的无效写。BIOS 制造商相信只有他们提供的 CMOS 系统设置程序才可以改变 CMOS 寄存器。最后，求和校验所有不同的寄存器并将其保存于供应商提供的 CMOS 寄存器中。一个 BIOS 上电测试保证 CMOS 求和校验有效。如果校验无效，BIOS 通常会停止系统的继续操作，直到 CMOS 内存重新装入正确的值。

许多 BIOS 供应商，例如 AMI 和 Phoenix，将从 10h 到 2Dh 的 CMOS 寄存器加起来，并将这个和值字保存在寄存器 2Eh 和 2Fh 中。寄存器 2Eh 保存着高位校验字节，而寄存器 2Fh 保存着低位校验字节。IBM 的 PS/1 486 BIOS 对寄存器 10h~31h 和 40~5Fh 使用两组求和校验。10h~31h 的求和校验字的高位保存在 CMOS 寄存器 32h 中，低字节保存在寄存器 33h 中。50h~4Fh 的求和校验字的高位保存在 CMOS 寄存器 60h 中，低字节保存在寄存器 61h 中。除非你绝对清楚求和校验是如何以及在哪里构造的，除非你会在下次系统引导之前将它们恢复为原始值，否则应该避免单独的 CMOS 内存写。

时钟寄存器 在设置和读取时钟寄存器时有几个指令比较特殊。参看本章开始有关实时时钟的一般信息一节。还要注意到读状态寄存器 C 和 D 可能会改变几位。参看状态端口 70h 中的寄存器 C 和 D 了解细节。

代码例 15-1 读 CMOS

```
;--------------------------------------------------------------------------
;   READ_CMOS
;       读取指定CMOS寄存器的内容
;
;       调用:           al = CMOS要读取的CMOS地址
;
;       返回:           ah = 寄存器的内容
;                           开放NMI，如果禁止

read_CMOS       proc    near
```

```
        or      al, 80h             ; 禁止NMI
        cli                         ; 禁止中断
        out     70h, al             ; al= 要读取的寄存器
        IODELAY
        in      al, 71h             ; 返回寄存器的内容
        mov     ah, al
        mov     al, 0
        IODELAY
        out     70h, al             ; 开放NMI
        sti                         ; 开放中断
        ret
read_CMOS       endp
```

代码例 15-2 写 CMOS

```
;--------------------------------------------------------------------------
;   WRITE_CMOS
;       向一个CMOS寄存器写
;
;       调用:       al = 要写入的CMOS寄存器
;                   ah = 要写入的值
;
;       返回:       NMI开放，如果禁止

write_CMOS      proc    near
        cli                         ; 禁止中断
        or      al, 80h             ; 禁止NMI
        out     70h, al             ; al= 要写的寄存器
        mov     al, ah
        IODELAY
        out     71h, al             ; 写寄存器
        mov     al, 0
        IODELAY
        out     70h, al             ; 开放NMI
        sti                         ; 开放中断
        ret
write_CMOS      endp
```

代码例 15-3 显示 AT CMOS

该程序显示标准的 CMOS 寄存器，从 Eh 到 3Fh。如果检测到扩展 CMOS 寄存器 40h~7Fh，那么也显示它们。在网站 www.infopower.com.cn下载区所提供的程序包中文件 CMOSVIEW.ASM 中提供了完整的程序。程序包中包括了数据定义和其他要求的子程序。

```
; ---------------------------------------------------------------------------
;     DISPLAY_CMOS
;         显示CMOS RAM内存的内容，
;         可以将其存入一个文件中。
;
;         调用:              无
;
;         返回:              发送到标准输出的CMOS寄存器值
;
;         用到的寄存器:      所有
;
;         子程序调用:        hex,read_CMOS, write_CMOS

display_CMOS    proc    near
        push    cs
        pop     ds
        push    cs
        pop     es

        OUTMSG  disp_CMOS_head      ; 输出表头串

; 首先确定CMOS大小是3Fh还是7Fh。如果
;   寄存器组E~3Fh和4Fh~7Fh相同，那么只支持64
;   个寄存器。否则可能支持128个寄存器

        mov     cl, 0Eh             ; 从CMOS地址0E开始
        mov     ch, 7Fh             ; 假定有128个寄存器
        mov     al, cl

disp_CMOS_size:
```

```
        call    read_CMOS               ; 获取ah中的地址值
        mov     bh, ah                  ; 临时存入bh
        add     cl, 40h                 ; 切换到可能的大写集
        call    read_CMOS               ; 获取ah中的地址值
        sub     cl, 40h
        cmp     ah, bh                  ; 寄存器相同吗?
        jne     disp_CMOS_128           ; 若非，则支持128个寄存器
        inc     cl
        cmp     cl, 3Fh                 ; 完成了吗?
        je      disp_CMOS_size          ; 若非，则循环

        mov     ch, 3Fh                 ; 设置大小为64个寄存器
        jmp     disp_CMOS_regs

; 寄存器不相同，所以可能有128个寄存器，但是为了
;   证实这一点，可以写入值并进行确认

disp_CMOS_128:
        mov     al, 7Fh                 ; 选择最后一个地址
        call    read_CMOS               ; 获取ah中的地址
        mov     bh, al                  ; 保存以供后面使用
        not     ah                      ; 翻转当前值
        mov     bl, ah                  ; 保存以供比较
        call    write_CMOS
        call    read_CMOS
        cmp     bl, ah                  ; 是写入的值吗?
        je      disp_CMOS_skp1          ; 若是，则跳转，128个寄存器
        mov     cl, 3Fh                 ; 最多64个寄存器
disp_CMOS_skp1:
        mov     ah, bh
        call    write_CMOS              ; 恢复初始值

; 从E开始向寄存器输出

disp_CMOS_regs:
        mov     cl, 0Eh                 ; 开始于CMOS地址E
```

```
disp_CMOS_loop:
        mov     al, cl
        mov     bx, offset disp_CMOS_loc
        call    hex                     ; 将地址转化成ASCII16进制
        mov     al, cl
        call    read_CMOS               ; 获取ah中的地址值
        mov     al, ah
        mov     bx, offset disp_CMOS_value
        call    hex                     ; 将值转化成ASCII16进制

        cmp     cl, 40h                 ; 在3F后插入馈送线
        jne     disp_CMOS_skp2
        OUTMSG crlf
disp_CMOS_skp2:
        cmp     cl, 40h                 ; 40h+ 是供应商专用
        ja      disp_CMOS_skp3          ;   重新使用文本
        mov     si, offset msgE
        mov     di, offset disp_CMOS_type
        mov     dx, offset msgF - offset msgE
        mov     al, cl
        sub     al, 0Eh
        mul     dl                      ; ax = dl*al
        add     si, ax
        push    cx
        mov     cx, dx
        cld
        rep     movsb                   ; 传送类型串
        pop     cx
disp_CMOS_skp3:
        OUTMSG  disp_CMOS_text          ; 输出信息行
        inc     cl
        cmp     cl, ch                  ; 所有都完成?
        jbe     disp_CMOS_loop

        ret
```

```
display_CMOS    endp
```

代码例 15-4 显示扩展 MCA CMOS

该程序显示 MCA 系统的 2048 个 CMOS 寄存器。从网上下载的（地址：www.infopower.com.cn）的文件 CMOSVIEW.ASM 中提供了完整的程序。在程序中包括了数据定义和其他要求的子程序。

```
;--------------------------------------------------------------------------
;    DISP_MCA_CMOS
;        显示MCA系统上的扩展
;        CMOS RAM内存内容。
;
;        调有:          无
;
;        返回:          发送到标准输出的CMOS寄存器值
;
;        用到的寄存器:  所有
;
;        子程序调用:    hex

disp_MCA_CMOS    proc    near
        push    cs
        pop     ds
        push    cs
        pop     es

; output the registers, starting at 0

        mov     cx, 0                   ; 扩展CMOS地址0

d_MCA_loop1:
        mov     bx, offset disp_eCMOS
        mov     al, ch
        call    hex                     ; 将寄存器转化成ASCII16进制
        add     bx, 2
```

```
        mov     al, cl
        call    hex                     ; 将寄存器转化成ASCII 16进制
        add     bx, 4

d_MCA_loop2:
        mov     al, cl
        cli                             ; 禁止中断
        out     74h, al                 ; 设置寄存器的低部分
        mov     al, ch
        out     75h, al                 ; 设置寄存器的高部分
        IODELAY
        in      al, 76h                 ; 获取端口值
        sti                             ; 开放中断
        call    hex
        add     bx, 3
        inc     cx
        test    cl, 7                   ; 在线第8个值?
        jnz     d_MCA_skp1              ; 若非，则跳转
        inc     bx
d_MCA_skp1:
        test    cl, 0Fh                 ; 16个值输出?
        jnz     d_MCA_loop2
        OUTMSG  disp_eCMOS_all          ; 16个寄存器的输出线
        cmp     cx, 800h                ; 所有寄存器输出完毕
        jb      d_MCA_loop1             ; 若非，则循环

d_MCA_done:
        ret
disp_MCA_CMOS   endp
```

代码例 15-5 显示扩展 EISA CMOS

该程序显示 EISA 系统的 8K CMOS 内存。网上程序包中的文件 CMOSVIEW.ASM 中提供了完整的程序。网上程序包中包括了数据定义和其他要求的子程序。

```
;--------------------------------------------------------------------------
```

```
;     DISP_EISA_CMOS
;          显示EISA系统上的扩展CMOS RAM内存内容
;          system.
;
;          调用:          无
;
;          返回:          CMOS寄存器发送到标准输出
;
;          使用的寄存器: 所有
;
;          子程序调用:    hex

disp_EISA_CMOS     proc     near
          push     cs
          pop      ds
          push     cs
          pop      es

; 从0开始输出寄存器

          mov      cl, 0                ; 扩展CMOS第0排
d_EISA_next_bank:                       ;   (1 of 32 banks)
          mov      dx, 800h             ; 端口800到8FF
d_EISA_loop1:
          mov      bx, offset disp_eCMOS
          mov      al, cl               ; 获取排号
          call     hex                  ; 转化成ASCII
          add      bx, 2
          mov      byte ptr [bx], '-'   ; 分隔符
          inc      bx
          mov      byte ptr [bx], '8'   ; 总是 800-8FFh
          inc      bx
          mov      al, dl
          call     hex                  ; 将寄存器转化成ASCII16进制
          add      bx, 4
```

```
d_EISA_loop2:
        cli                             ; 禁止中断
        push    dx
        mov     dx, 0C00h
        mov     al, cl                  ; 排选择
        out     dx, al
        pop     dx
        IODELAY
        in      al, dx                  ; 获取寄存器
        sti                             ; 开放中断
        call    hex
        add     bx, 3
        inc     dx
        test    dl, 7                   ; 在线第8个值?
        jnz     d_EISA_skp1             ; 若非，则跳转
        inc     bx
d_EISA_skp1:
        test    dl, 0Fh                 ; 16个值输出?
        jnz     d_EISA_loop2
        push    dx
        OUTMSG  disp_eCMOS_all          ; 16个寄存器的输出线
        pop     dx
        cmp     dx, 900h                ; 256个寄存器输出
        jb      d_EISA_loop1
        OUTMSG  crlf                    ; 插入一个排线
        inc     cl
        cmp     cl, 32                  ; 所有的32排都已经输出?
        jb      d_EISA_next_bank
        ret
disp_EISA_CMOS    endp
```

端口归纳

端口	类型	功能	平台
70h	输出	CMOS 内存地址和开放 NMI	AT+
71h	I/O	CMOS 内存数据	AT+

74h	输出	扩展 CMOS 地址 LSB	MCA
75h	输出	扩展 CMOS 地址 MSB	MCA
76h	I/O	扩展 CMOS 数据寄存器	MCA
8xxh	I/O	扩展 CMOS 寄存器	EISA
C00h	输出	扩展 CMOS 块选择	EISA

端口详述

中断	功能	描述	平台
70h	输出	CMOS 内存地址和开放 NMI	AT+

访问 CMOS 内存时，RAM 地址首先写到端口 70h，然后使用端口 71h 来读或写前面指定的 RAM 地址。

输出（位 0~7） CMOS 地址和 NMI

位		
7 w = 0		支持不可屏蔽中断，NMI 中断 2
1		禁止 NMI（对 CMOS RAM 的普通访问中使用）
6 w = x		未使用（某些系统扩展 CMOS 的上端 CMOS 地址位）
5 w = x		下次读或写的 CMOS RAM 地址
4 w = x		
3 w = x		
2 w = x		
1 w = x		
0 w = x		

下面的列表归纳了 CMOS 地址。早期的 AT 含有 64 字节的 RAM 以及时钟信息。目前某些系统带有另外的内存，供其他选项使用。

寄存器归纳——时间和日期

地址	寄存器描述
00h	秒
01h	秒闹钟
02h	分钟
03h	分钟闹钟
04h	小时
05h	小时闹钟
06h	星期几
07h	几号
08h	月

09h	年
0Ah	状态寄存器 A
0Bh	状态寄存器 B
0Ch	状态寄存器 C
0Dh	状态寄存器 D

寄存器归纳——CMOS 内存

在一些显示了 CMOS 寄存器的技术参考资料中，许多地方标注着“保留”字。保留的入口通常是一些未使用的入口，尽管它们也用来保存某些 BIOS 提供的特有性能和选项。我将“保留”改称为“供应商专用”，并提供了一些主要制造商对它们的使用情况。

记住，目前大多数 BIOS 使用专用的主板和 CPU。这意味着特定发布日期内的 BIOS 可能有不同的配置情况，有可能与 CMOS 寄存器的使用发生冲突！通常在地址 F000:FFF5 处可以找到 ASCII 形式的 BIOS 发布日期。永远也不要假定，代码日期和供应商相同的 BIOS 使用 CMOS 寄存器的方法相同。

你可以使用软件包中的 CMOSVIEW 工具，来在你的特定系统上找到 CMOS 寄存器。输出可以被拷贝到一个文件中。然后重新引导系统并进入 BIOS 设置程序。改变一个标志或值，并比较以前保存的设置结果和当前的设置结果，来看看信息确切的保存位置。

用一个快速的批处理文件和一个差分程序，我发现要找出一个 BIOS 上的大部分标志和值需要大约三到四个小时。记住某些值，例如芯片组的设置和 RAM 的刷新设置，可能会使系统不稳定。某些测试可能也需要你从软盘驱动器引导。例如，改变硬盘驱动器的设置可能会影响（也就是破坏）硬盘驱动器上的信息！

地址	描述
0Eh	诊断状态字节
0Fh	关机状态字节
10h	软盘驱动器类型（驱动器 A 和 B）
11h	供应商专用
12h	硬盘类型
13h	供应商专用
14h	设备字节
15h	基本内存，低
16h	基本内存，高
17h	扩展内存，低
18h	扩展内存，高
19h	硬盘 0 的扩展字节
1Ah	硬盘 1 的扩展字节

1Bh	供应商专用
1Ch	供应商专用
1Dh	供应商专用
1Eh	供应商专用
1Fh	供应商专用
20h	供应商专用
21h	供应商专用
22h	供应商专用
23h	供应商专用
24h	供应商专用
25h	供应商专用
26h	供应商专用
27h	供应商专用
28h	供应商专用
29h	供应商专用
2Ah	供应商专用
2Bh	供应商专用
2Ch	供应商专用
2Dh	供应商专用
2Eh	CMOS求和校验，高（MCA除外）
2Fh	CMOS求和校验，低（MCA除外）
30h	扩展内存，低
31h	扩展内存，高
32h	日期的世纪字节（ISA）；CRC高（MCA）
33h	上电信息标志（ISA）；CRC低（MCA）
34h	供应商专用
35h	供应商专用
36h	供应商专用
37h	日期的世纪字节（MCA）
38h	供应商专用
39h	供应商专用
3Ah	供应商专用
3Bh	供应商专用
3Ch	供应商专用
3Dh	供应商专用
3Eh	供应商专用
3Fh	供应商专用
40h~7Fh	供应商专用

寄存器详述

寄存器	描述	类型
00h	秒	CMOS

这个寄存器保存了 BCD 格式的 RTC 当前的秒值。有效范围是 0~59。参看一般信息部分了解对这个地址的读写限制。

寄存器	描述	类型
01h	秒闹钟	CMOS

这个地址保存了 BCD 格式的 RTC 闹钟当前的秒值。有效范围是 0~59。有许多方法对闹钟编程。参看寄存器 A~D 了解更多细节。如果最高两位设置为 1（值为 C0h~FFh），则忽略闹钟的这个秒钟寄存器。参看一般信息部分对实时时钟的描述，来了解对这个地址的读写限制。

寄存器	描述	类型
02h	分钟	CMOS

这个寄存器保存了 BCD 格式的 RTC 当前的分钟值。有效范围是 0~59。参看一般信息部分了解对这个地址的读写限制。

寄存器	描述	类型
03h	分钟闹钟	CMOS

这个地址保存了 BCD 格式的 RTC 闹钟当前的分钟值。有效范围是 0~59。有许多方法对闹钟编程。参看寄存器 A~D 了解更多细节。如果最高两位设置为 1（值为 C0h~FFh），则忽略闹钟的这个分钟寄存器。参看一般信息部分对实时时钟的描述，来了解对这个地址的读写限制。

寄存器	描述	类型
04h	小时	CMOS

这个寄存器保存了 BCD 格式的 RTC 当前的小时值。在 12 小时模式下有效范围是 1~12。如果是上午，则该字节的位 7 为 0，如果是下午，则该字节的位 7 为 1。在 24 小时模式下有效范围是 0~23。寄存器 B 的位 1 控制使用 12 还是 24 小时模式。参看一般信息部分了解对这个地址的读写限制。

寄存器	描述	类型
05h	小时闹钟	CMOS

这个地址保存了 BCD 格式的 RTC 闹钟当前的小时值。在 12 小时模式下有效范围是 1~12。如果是上午，则该字节的位 7 为 0，如果是下午，则该字节的位 7 为 1。在 24 小时模式下有效范围是 0~23。有许多方法对闹钟编程。参看寄存器 A~D 了解更多细节。如果最高两位设置为 1（值为 C0h~FFh），则忽略闹钟的这个小时寄存器。参看一般信息部分对实时时钟的描述，来了解对这个地址的读写限制。

寄存器	描述	类型
06h	星期几	CMOS

该地址指示是星期几。尽管可以从日期中确定星期几，但是这个仍可能被设置为错误的数据。一般地，操作系统会忽略该字节，而执行它自己的操作，来从日期中确定星期几。参看一般信息部分来了解对这个地址的读写限制。

值	星期几
1	星期天
2	星期一
3	星期二
4	星期三
5	星期四
6	星期五
7	星期六

寄存器	描述	类型
07h	日号	CMOS

该地址指示 RTC 目前是几号，格式是 BCD。有效范围是 1~31。参看一般信息部分来了解对这个地址的读写限制。

寄存器	描述	类型
08h	月份	CMOS

该地址指示 RTC 目前是几月，格式是 BCD。有效范围是 1~12。参看一般信息部分来了解对这个地址的读写限制。

寄存器	描述	类型
09h	年份	CMOS

该地址指示 RTC 目前年份，格式是 BCD。有效范围是 0~99。参看一般信息部分来了解对这个地址的读写限制。

寄存器	描述	类型
0Ah	状态寄存器 A	CMOS

该地址控制 RTC 的操作，硬件重启不会影响任何位。注意位 7 是只可读的，向这个寄存器写不会改变该位。

位 7 r = 1　正在更新。在 PC 上，大约每秒有 2 μs 该位保持高电平。时钟只会在该位从高到低过渡后的 244 μs 内更新。

6 r/w = 0
5 r/w = 1
4 r/w = 0　除法因子控制——为选中的指定晶体设置除法因子。PC 使用 32.768kHz 的晶体，所以位 6，5，4 应该是 010。其他的值下的时钟要慢得多。

位 6	位 5	位 4
0	0	0=普通的时钟频率除以 128
0	1	1=普通的时钟频率除以 32
0	1	0=普通

其他的组合用于测试模式

3 r/w = x
2 r/w = x
1 r/w = x
0 r/w = x　速率选择除法因子控制——用于设置周期性中断出现的频率。通常设置为 6（0110b），每 976 μs 生成一个中断

低四位字节

0=不生成周期性中断
1 = 3.90625 ms
2 = 7.8125 ms
3 = .122070 ms
4 = .244141 ms
5 = .488281 ms

6 = .976572 ms（缺省）
7 = 1.953125 ms
8 = 3.90625 ms
9 = 7.8125 ms
A = 15.625 ms
B = 31.25 ms
C = 62.5 ms

D = 125.0 ms
E = 250.0 ms
F = 500.0 ms

寄存器	描述	类型
0Bh	状态寄存器 B	CMOS

该寄存器控制 RTC 的操作。硬件重启不改变位 7、2、1 和 0，在硬件重启之后位 6、5、4 和 3 都设置为 0。

位	7 r/w = 0	每秒更新一次时钟
	1	停止时钟（用来将时钟更新为一个新的时间或日期）
	6 r/w = 0	在更新特性后，它也强制位 3~0 关闭中断，如果允许阻止周期性中断的发生。硬件重启会将该位设置为零（缺省）
	1	开放周期性中断。触发中断的频率由寄存器 A，频率除法因子，指定，因此是每 976.562μS 一次。
	5 r/w = 0	禁止闹钟中断（缺省）
	1	开放闹钟中断。如果这三个闹钟寄存器匹配当前的时间，就触发中断。当秒寄存器位于不管状态时，只要剩下的寄存器匹配，就每秒触发一次。
	4 r/w = 0	禁止更新完成中断（缺省）
	1	允许每次时钟更新完成时触发一个中断（每秒一次）。如果将位 7 设置为 1，则清除该标志。
	3 r/w = 0	禁止序列波（缺省）
	1	开放序列波。由于芯片的序列波引脚没有连接在主板上，所以这个序列波功能不可用
	2 r/w = 0	时间和日期保存为 BCD 格式（缺省）
	1	时间和日期保存为二进制格式（从未使用）
	1 r/w = 0	小时保存为 12 小时模式
	1	小时保存为 24 小时模式（缺省）
	0 r/w = 0	禁止夏令制时间（缺省）
	1	允许夏令制时间。如前所述，内部时间的改变是基于日期的，美国不再使用这种方式。激活时，在四月的最后一个星期天，上午 1:59:59，时钟会切换到上午 3:00:00。在十月的最后一个星期天，上午 1:59:59，时钟会切换到上午 1:00:00。

寄存器	描述	类型
0Ch	状态寄存器 C	CMOS

该寄存器含有中断操作的当前标志状态。寄存器 C 是只可读的。读取或者硬件重启后会清除所有的位。

位	7 r = 0	没有激活中断请求。周期性中断（位 6）、闹钟（位 5）和更新开放标志（位 4）都为 0。
	1	激活了中断请求，位 6、5 或 4 中的任何一位会被设置为 1，并激活 IRQ8。
	6 r = 1	在每个周期性循环内设置一次周期性中断标志，频率由寄存器 A 的频率选择位设置。寄存器 B 的位 6 也应设置为 1。此时位 7 设置为 1，激活 IRQ8。
	5 r = 1	这个闹钟标志指示闹钟时间和当前时间相匹配。必须将寄存器 B 的位 5 设置为 1。此时位 7 设置为 1，激活 IRQ8。
	4 r = 0	每次时钟更新周期结束后设置这个更新结束标志。寄存器 B 的第四位应设置为 1。此时位 7 设置为 1，激活 IRQ8。
	3 r = 0	未使用
	2 r = 0	未使用
	1 r = 0	未使用
	0 r = 0	未使用

寄存器	描述	类型
0Dh	状态寄存器 D	CMOS

有效 RAM 和时间 RTC 寄存器。寄存器 D 是只读的，硬件重启时不会改变这个寄存器。读取后，位 7 设置为 1。

位	7 r = 0	由于 RTC 芯片丢失电源，RAM 的内容和时间无效。如果系统断电时没有连接电池，或者电池没有电，或者电池的电压不足以运行 RTC，就会出现这种情况。
	1	从上次读取这个寄存器后开始，RTC 电源一直是稳定的。
	6 r = 0	未使用
	5 r = 0	未使用
	4 r = 0	未使用
	3 r = 0	未使用
	2 r = 0	未使用
	1 r = 0	未使用
	0 r = 0	未使用

CMOS 内存

下面的CMOS RAM地址含有系统主BIOS的不可变信息。

寄存器	描述	类型
0Eh	诊断状态	CMOS

该寄存器保存了系统POST（系统上电自检）的错误信息。

位	7 r/w = 1	RTC丢失电源，其内容不再有效
	6 r/w = 1	CMOS求和校验失败
	5 r/w = 1	检测到不正确的配置信息
	4 r/w = 1	内存大小和配置记录不匹配
	3 r/w = 1	硬盘驱动器0（C:）的适配器或驱动器没有初始化，普通的引导失败。
	2 r/w = 1	时间或日期无效
	1 r/w = 1	适配器与配置不相匹配（仅MCA/EISA）
	0 r/w = 1	读取适配器ID时超时（仅MCA/EISA）

寄存器	描述	类型
0Fh	关机状态	CMOS

这个字节包含了一个代码，指示关闭系统后系统应返回到何处。BIOS使用它来仔细检查一个重启操作，可能会执行其他的操作，而不是普通的重启操作。对于触发一次重启来从保护模式返回到实模式，这一点很有用。

代码0、4、5、9和Ah一般和不同的BIOS制造商保持一致。不同BIOS制造商和微处理器的其他的代码的功能可能会有所不同。

重启后，某些功能可能会跳转或调用以前保存在40:67h双字中的地址。在跳转或调用时，POST会禁止中断和NMI。对端口触发命令OUT，来输入值0，可以开放NMI。然后触发一条STI命令来开放中断。用户必须为代码5恢复中断屏蔽寄存器。某些功能会向中断控制器触发一条中断结束（EOI）命令，而其他的功能不会触发该命令。参看处理EOI的专用功能。在通过跳转或调用传递控制之前，POST会将堆栈SS:SP设置为0:400h。这意味着中断向量表的顶部被用作一个临时的堆栈。用户应该确保将堆栈指针改变成一个更合适的地址。

代码	0 = 正在处理普通的POST（软件重启）
	1 = 为实模式操作初始化芯片组
	2 = 内存测试后关机

3 = 内存出错时关机
4 = 跳转到磁盘引导程序
5 = 跳转到 40:67h 处的双字指针（触发 EOI，清空键盘）
6 = 跳转到 40:67h 处的双字指针（不触发 EOI）
7 = 返回到 BIOS 扩展内存块传送（正在执行中断 15h,功能 87h）
8 = 返回到 POST 内存测试
9 = 返回到 BIOS 扩展内存块传送（正在执行中断 15h,功能 87h）
A = 跳转到 40:67h 处的双字指针（不触发 EOI）
B = 返回，就好像通过 40:67 触发了 IRET 一样
C = 返回，就好像通过 40:67 触发了 IRET 一样

寄存器	描述	类型
10h	软盘驱动器类型	CMOS

定义所安装的软盘驱动器的类型。只允许定义对指定 BIOS 有效的值。如果一个 BIOS 不支持 2.88MB 的驱动器，你就不能在 CMOS 中插入与 2.88MB 软盘驱动器相对应的值。

位	7 r/w = x	软盘类型，驱动器 0	
	6 r/w = x	高四位字节	
	5 r/w = x	0	无驱动器
	4 r/w = x	1	360KB 5.25 英寸驱动器
		2	1.2MB 5.25 英寸驱动器
		3	720KB 3.5 英寸驱动器
		4	1.44MB 3.5 英寸驱动器
		5	2.88MB 3.5 英寸驱动器
	3 r/w = x	软盘类型，驱动器 0	
	2 r/w = x	高四位字节	
	1 r/w = x	0	无驱动器
	0 r/w = x	1	360KB 5.25 英寸驱动器
		2	1.2MB 5.25 英寸驱动器
		3	720KB 3. 5 英寸驱动器
		4	1.44MB 3.5 英寸驱动器
		5	2.88MB 3.5 英寸驱动器

寄存器	描述	类型
11h	供应商专用	CMOS

这是供应商专用的 CMOS 地址，某些系统可能没有使用它。许多 BIOS 都服从共同的标准，但是也有一些 BIOS 供应商对特殊的主板使用不同的标志，即使它们代码的编写日期相同！

AMI BIOS，代码日期 11-11-92

位		
	7 r/w =1	开放鼠标（如果 BIOS 支持鼠标）
	6 r/w =1	测试 1MB 以上内存
	5 r/w =1	测试内存时发出滴答声
	4 r/w =1	开放内存奇偶校验
	3 r/w =1	在引导时，显示设置键
	2 r/w =x	仅在 RAM 中没有影射（shadow）系统 BIOS 时，指定用户定义的硬盘数据的地址。开放影映时，用户磁盘数据保存在影映的内存区，并忽略该位。
	0	在 0:300h 处保存用户定义的磁盘类型
	1	保存在主存的顶部（640K 变成 639K）
	1 r/w =1	如果引导时出错，则要求按 F1 继续
	0 r/w =0	引导时关闭了 Num lock
	1	引导时开放了 Num lock

Pnoenix BIOS，代码日期 1-11-96

位		
	7 r/w =x	PCI 插槽 1——潜伏时间时钟
	6 r/w =x	对于时钟值 0~F8h，将值乘以 8
	5 r/w =x	
	4 r/w =x	
	3 r/w =x	
	2 r/w =1	PCI 插槽 1——主开放
	1 r/w =0	未使用或未知功能
	0 r/w =1	PCI 插槽 1——激活缺省的潜伏时间时钟

寄存器	描述	类型
12h	硬盘类型	CMOS

定义硬盘 0 和 1 的类型号。这个类型指定了磁盘类型信息 ROM 表的一个入口。

位			
	7 r/w =x	硬盘类型，驱动器 0	
	6 r/w =x	高四位字	
	5 r/w =x	0	无驱动器
	4 r/w =x	1~0Eh	类型 1~14
		0Fh	类型 16~255，由驱动器 0 的扩展类型 CMOS 寄存器 19h 指定
	3 r/w =x	硬盘类型，驱动器 1	
	2 r/w =x	高四位字	

1 r/w =x	0	无驱动器
0 r/w =x	1~0Eh	类型 1~14
	0Fh	类型 16~255，由驱动器 0 的扩展类型 CMOS 寄存器 1Ah 指定

大多数 BIOS 制造商在他们的驱动器表中所使用的内容并不相同。同一个制造商的不同 BIOS 版本之间表的数据也往往有所不同。这个表的组织结构并不总是保持一致。

每个类型的入口由下面 14 个字节的信息组成（参看表 11-3 进一步了解细节）：

偏移量	大小	描述
0	字	磁柱数
2	字节	磁头数
3	字	低写电流，磁柱开始（仅 XT，其他都未使用）
5	字	预补偿磁柱
7	字节	错误纠正脉冲长度（仅 XT，其他都未使用）
8	字节	磁盘信息位 位 7 = 1 不重试 位 6 = 1 不重试 位 5 = 1 坏的映射位于最后一个扇区+1 位 4 = 0 未使用 位 3 = 1 如果磁头数大于 8 位 2 = 0 必须为零（不重设） 位 1 = 0 必须为零（禁止 IRQ） 位 0 = 0 未使用
9	字节	一般性超时（仅 XT，其他都未使用）
A	字节	格式化超时（仅 XT，其他都未使用）
B	字节	校验超时（仅 XT，其他都未使用）
C	字节	每磁道的扇区数
D	字节	未使用

注意，预补偿和装入区域磁柱值早已过时。当前的驱动器不考虑使用这些值。如果这个信息未知，则将每个值设置为相同的值，即磁柱总数。

大多数新式的 BIOS 保留了多达四个的类型数，它们用作用户自定义的入口。这一点支持设置一个用户自定义的驱动器类型。在 IDE 驱动器上它能很好的工作，在这种驱动器上，确切的磁柱数、磁头数以及每磁道的扇区数没有太大的意义。驱动器的总容量是它们乘在一起所得的值，这才是最重要的。并且，用户自定义设置支持最大化驱动器的有用容量。参看开始于 1Bh 的供应商专用地址。

寄存器	描述	类型
13h	供应商专用	CMOS

这是一个供应商专用地址，可能某些系统并未使用这个寄存器。许多 BIOS 服从相同的标准，但是某些 BIOS 供应商对特殊的主板使用不同的标志，尽管它们的代码编写日期相同！

AMI BIOS，代码日期 11-11-92

位 7 r/w = 1：将键盘的重复速率和延迟强制为初始值。

位 6 r/w = x，位 5 r/w = x：选择出现键盘重复前的延迟

位 6	位 5
0	0 = 250 ms
0	1 = 500 ms
1	0 = 750 ms
1	1 = 1000 ms

位 4 r/w = x，位 3 r/w = x，位 2 r/w = x：选择键盘重复速率

位 4	位 3	位 2
0	0	0 = 6 个字符/秒
0	0	1 = 8 个字符/秒
0	1	0 = 10 个字符/秒
0	1	1 = 12 个字符/秒
1	0	0 = 15 个字符/秒
1	0	1 = 20 个字符/秒
1	1	0 = 25 个字符/秒
1	1	1 = 30 个字符/秒

位 1 r/w = 0：重复速率的低两位

位 0 r/w = 0：控制分辨率，但是这些位不是由 BIOS 配置程序设置。参看端口 F3h 了解所有的重复率值。（第 8 章，键盘系统）

PS/2 MCA BIOS

这是一个内部 POST 操作使用的保留字节。

位		
	7 r/w = 1	POST 设置 VGA 像素信息
	6 r/w = 0	RTC 丢失电池电源
	5 r/w = 1	从 POST 进入 ROM BASIC
	4 r/w = 0	POST 设置键入速率：10.9cps，延迟 500ms
	1	POST 设置键入速率：30cps，延迟 250ms
	3 r/w = x	未使用或未知功能
	2 r/w = x	未使用或未知功能
	1 r/w = 1	安装了网络密码
	0 r/w = 0	安装了上电密码

Compaq BIOS

某些 COMPAQ 系统使用这个地址来指示存在一个次级硬盘控制器。这时，驱动器 0 使用一个控制器，驱动器 1 使用另一个控制器。要检测一个 Compaq 系统，在 F000:FFE8 处查找串 03COMPAQ。

字节=	零	如果没有第二个控制器
	非零	如果使用了第二个控制器

Phoenix BIOS，代码日期 1-11-96

位	7 r/w =x	PCI 插槽 2——潜伏时间时钟
	6 r/w =x	对于时钟值 0~F8h，将值乘以 8
	5 r/w =x	
	4 r/w =x	
	3 r/w =x	
	2 r/w =1	PCI 插槽 2——主开放
	1 r/w =0	未使用或未知功能
	0 r/w =1	PCI 插槽 2——激活缺省的潜伏时间时钟

寄存器	描述	类型
14h	设备字节	CMOS

这个寄存器包含了附属的软盘驱动器、引导时的主显示器及其他信息。POST 使用这个入口在 40:10h 处创建设备字。中断 11h 可用来从 40:10h 处读取设备信息。

位	7 r/w = x	安装的软盘驱动器数	
	6 r/w = x	位 7	位 6
		0	0 = 安装了 1 个软盘驱动器
		0	1 = 安装了 2 个软盘驱动器
		1	x = 未使用
	5 r/w = x	主视频显示	
	4 r/w = x	位 5	位 4
		0	0 = 带有 BIOS ROM 的视频卡（EGA/VGA+）
		0	1 = 40 列，25 行，彩色，CGA
		1	0 = 80 列，25 行，彩色，CGA
		1	1 = 80 列，25 行，单色
	3 r/w = 0	未使用（大多数系统）或者忽略 POST 键盘测试（AMI BIOS）	
	2 r/w = 0	未使用	
	1 r/w = 1	安装了标准的数学协处理器	
	0 r/w = 0	不从软盘驱动器引导	
	1	安装了软盘驱动器引导盘	

寄存器	描述	类型
15h	基本内存——低	CMOS

和寄存器 16h 一起组成一个字来指示基本内存的大小，单位是 1,024 字节。一个典型的值是 0280h 表示有 640KB 的基本内存。POST 将这个字保存在 BIOS 内存取地址 40:13h 中。可以使用中断 12h，获取内存大小，来读取该值。

寄存器	描述	类型
16h	基本内存——高	CMOS

和寄存器 15h 一起组成一个字来指示基本内存的大小，单位是 1KB。参看 CMOS 寄存器 15h 了解更多信息。

寄存器	描述	类型
17h	扩展内存——低	CMOS

和寄存器 17h 一起组成一个字来指示扩展内存的大小，单位是 1,024 字节。例如，3C00h 表示有 15MB 的扩展内存。许多老式的 BIOS 最大只支持 15MB。许多新式的 BIOS 可以接受多达 63MB 的扩展内存，这是这个字的上限。在 63MB 以外没有标准。

寄存器	描述	类型
18h	扩展内存——高	CMOS

和寄存器 17h 一起组成一个字来指示扩展内存的大小，单位是 1KB。参看 CMOS 寄存器 17h 了解更多信息。

寄存器	描述	类型
19h	硬盘 0 的扩展字节	CMOS

非 MCA 系统

当硬盘类型的高四位字节是 16~255 时，驱动器类型字节 12h 就是 Fh。

MCA 系统

这个寄存器保存了卡插槽 0 的适配卡 ID。这个字节直接复制于扩展 MCA CMOS 寄存器 0。参看端口 76h，寄存器 0 了解细节。

寄存器	描述	类型
1Ah	硬盘 1 的扩展字节	CMOS

当硬盘类型的低四位字节是 16~255 时，驱动器类型字节 12h 就是 Fh。

在 MCA 系统上，这个寄存器保存了卡插槽 0 的适配卡 ID。这个字节直接复制于扩展 MCA CMOS 寄存器 1。参看端口 76h，寄存器 1 了解细节。

寄存器	描述	类型
1Bh	供应商专用	CMOS

这是一个供应商专用的地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 11-11-92

这个寄存器保存了硬盘 0 的磁柱字的低部分。只有当硬盘 0 指定为类型 47（用户定义）时才可以使用这个寄存器。

Phoenix BIOS，日期 9-28-95

这个寄存器保存了硬盘 3 的磁柱字的低部分。

Phoenix BIOS，日期 1-11-96

这个寄存器保存了硬盘 0 的磁柱字的低部分。只有当硬盘 0 指定为类型 47（用户定义）时才可以使用这个寄存器。

MCA 系统

这个寄存器保存了卡插槽 1 的适配卡 ID。这个字节直接复制于扩展 MCA CMOS 寄存器 23h。参看端口 76h、寄存器 23h 了解细节。

寄存器	描述	类型
1Ch	供应商专用	CMOS

这是一个供应商专用的地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 11-11-92

这个寄存器保存了硬盘 0 的磁柱字的高部分。只有当硬盘 0 指定为类型 47（用户定义）时才可以使用这个寄存器。

Phoenix BIOS，日期 9-28-95

这个寄存器保存了硬盘 3 的磁柱字的高部分。

Phoenix BIOS，日期 1-11-96

这个寄存器保存了硬盘 0 的磁柱字的高部分。只有当硬盘 0 指定为类型 47（用户定义）时才可以使用这个寄存器。

MCA 系统

这个寄存器保存了卡插槽 1 的适配卡 ID。这个字节直接复制于扩展 MCA CMOS 寄存

器 24h。参看端口 76h，寄存器 24h 了解细节。

寄存器	描述	类型
1Dh	供应商专用	CMOS

这是一个供应商专用的地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 11-11-92

硬盘 0 的磁头数。只有当硬盘 0 指定为类型 47（用户定义）时才可以使用这个寄存器。

Phoenix BIOS，日期 9-28-95

这个寄存器保存了硬盘 3 用来放写预补偿字的磁柱号的低部分。

Phoenix BIOS，日期 1-11-96

这个寄存器保存了硬盘 1 的磁柱字的低部分。只有当硬盘 1 指定为类型 47（用户定义）时才可以使用这个寄存器。

MCA 系统

这个寄存器保存了卡插槽 2 的适配卡 ID。这个字节直接复制于扩展 MCA CMOS 寄存器 46h。参看端口 76h，寄存器 46h 了解细节。

寄存器	描述	类型
1Eh	供应商专用	CMOS

这是一个供应商专用的地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 11-11-92

这个寄存器保存了硬盘 0 用来放写预补偿字的磁柱号的低部分。只有当硬盘 0 指定为类型 47（用户定义）时才可以使用这个寄存器。

Phoenix BIOS，日期 9-28-95

这个寄存器保存了硬盘 3 用来放写预补偿字的磁柱号的高部分。

Phoenix BIOS，日期 1-11-96

这个寄存器保存了硬盘 1 的磁柱字的高部分。只有当硬盘 1 指定为类型 47（用户定义）时才可以使用这个寄存器。

MCA 系统

这个寄存器保存了卡插槽 2 的适配卡 ID。这个字节直接复制于扩展 MCA CMOS 寄存器 47h。参看端口 76h，寄存器 47h 了解细节。

寄存器	描述	类型
1Fh	供应商专用	CMOS

这是一个供应商专用的地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 11-11-92

这个寄存器保存了硬盘 0 用来放写预补偿字的磁柱号的高部分。只有当硬盘 0 指定为类型 47（用户定义）时才可以使用这个寄存器。

Phoenix BIOS，日期 9-28-95

这个寄存器保存了硬盘 0 用来放写预补偿字的磁柱号的高部分。只有当硬盘 0 指定为类型 47（用户定义）时才可以使用这个寄存器。和寄存器 20h 一起，值 0FFFh 表示没有写预补偿。

Phoenix BIOS，日期 1-11-96

这个寄存器保存了硬盘 1 的磁柱字的低部分。只有当硬盘 1 指定为类型 47（用户定义）时才可以使用这个寄存器。

MCA 系统

这个寄存器保存了卡插槽 3 的适配卡 ID。这个字节直接复制于扩展 MCA CMOS 寄存器 69h。参看端口 76h，寄存器 69h 了解细节。

寄存器	描述	类型
20h	供应商专用	CMOS

这是一个供应商专用的地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 11-11-92

这个寄存器保存了硬盘 0 的磁盘信息标志。只有当硬盘 0 指定为类型 47（用户定义）时才可以使用这个寄存器。

位		
	7 r/w = 1	无重试
	6 r/w = 1	无重试
	5 r/w = 1	在最后磁柱加 1 上的坏映射
	4 r/w = 0	未使用
	3 r/w = 1	如果磁头数大于 8
	2 r/w = 0	未使用
	1 r/w = 0	未使用
	0 r/w = 0	未使用

Phoenix BIOS，日期 9-28-95

这个寄存器保存了硬盘 0 用来放写预补偿字的磁柱号的高部分。只有当硬盘 0 指定为类型 47（用户定义）时才可以使用这个寄存器。和寄存器 IFh 一起，值 0FFFh 表示没有写预补偿。

Phoenix BIOS，日期 1-11-96

这个寄存器保存了硬盘 1 的磁柱字的高部分。只有当硬盘 1 指定为类型 47（用户定义）时才可以使用这个寄存器。

MCA 系统

这个寄存器保存了卡插槽 3 的适配卡 ID。这个字节直接复制于扩展 MCA CMOS 寄存器 6Ah。参看端口 76h，寄存器 6Ah 了解细节。

寄存器	描述	类型
21h	供应商专用	CMOS

这是一个供应商专用的地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 11-11-92

这个寄存器保存了硬盘 0 用来用来装入区域字的磁柱号的低部分。只有当硬盘 0 指定为类型 47（用户定义）时才可以使用这个寄存器。

MCA 系统

这个寄存器保存了可编程选项选择配置字节 2，POST 将它发送到适配卡插槽 0。这个字节直接复制于扩展 MCA CMOS 寄存器 3。

寄存器	描述	类型
22h	供应商专用	CMOS

这是一个供应商专用的地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 11-11-92

这个寄存器保存了硬盘 0 用来装入区域字的磁柱号的高部分。只有当硬盘 0 指定为类型 47（用户定义）时才可以使用这个寄存器。

Phoenix BIOS，日期 9-28-95

位		
	7 r/w = x	硬盘 1 每磁道的扇区数，低部分
	6 r/w = x	（高四位在寄存器 23h）
	5 r/w = x	硬盘 0 每磁道的扇区数
	4 r/w = x	
	3 r/w = x	
	2 r/w = x	
	1 r/w = x	
	0 r/w = x	

MCA 系统

这个寄存器保存了可编程选项选择配置字节 3，POST 将它发送到适配卡插槽 0。这个字节直接复制于扩展 MCA CMOS 寄存器 4。

寄存器	描述	类型
23h	供应商专用	CMOS

这是一个供应商专用的地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 11-11-92

这个寄存器保存了硬盘 0 每磁道的扇区数。只有当硬盘 0 指定为类型 47（用户定义）时才可以使用这个寄存器。

Phoenix BIOS，日期 9-28-95

位		
	7 r/w = x	硬盘 2 每磁道的扇区数，低部分（高两位在寄存器 24h）
	6 r/w = x	
	5 r/w = x	
	4 r/w = x	
	3 r/w = x	硬盘 1 每磁道的扇区数，高部分（低两位在寄存器 24h）
	2 r/w = x	
	1 r/w = x	
	0 r/w = x	

MCA 系统

这个寄存器保存了可编程选项选择配置字节 4，POST 将它发送到适配卡插槽 0。这个字节直接复制于扩展 MCA CMOS 寄存器 5。

寄存器	描述	类型
24h	供应商专用	CMOS

这是一个供应商专用的地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 11-11-92

这个寄存器保存了硬盘 1 磁柱字的低部分。只有当硬盘 1 指定为类型 47（用户定义）时才可以使用这个寄存器。

Phoenix BIOS，日期 9-28-95

位		
	7 r/w = x	硬盘 3 每磁道的扇区数
	6 r/w = x	
	5 r/w = x	
	4 r/w = x	
	3 r/w = x	
	2 r/w = x	
	1 r/w = x	硬盘 2 每磁道的扇区数，高部分（低四位在寄存器 23h）
	0 r/w = x	

MCA 系统

这个寄存器保存了可编程选项选择配置字节 5，POST 将它发送到适配卡插槽 0。这个字节直接复制于扩展 MCA CMOS 寄存器 6。

寄存器	描述	类型
25h	供应商专用	CMOS

这是一个供应商专用的地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 11-11-92

这个寄存器保存了硬盘 1 磁柱字的高部分。只有当硬盘 1 指定为类型 47（用户定义）时才可以使用这个寄存器。

Phoenix BIOS，日期 9-28-95

位		
	7 r/w = x	硬盘 1 的磁头数，指定类型为 47
	6 r/w = x	（值 0=1 个磁头；值 F=16 个磁头）
	5 r/w = x	
	4 r/w = x	
	3 r/w = x	硬盘 0 的磁头数，指定类型为 47
	2 r/w = x	（值 0=1 个磁头；值 F=16 个磁头）
	1 r/w = x	
	0 r/w = x	

MCA 系统

这个寄存器保存了可编程选项选择配置字节 2，POST 将它发送到适配卡插槽 1。这个字节直接复制于扩展 MCA CMOS 寄存器 26h。

寄存器	描述	类型
26h	供应商专用	CMOS

这是一个供应商专用的地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 11-11-92

这个寄存器保存了硬盘 1 的磁头数。只有当硬盘 1 指定为类型 47（用户定义）时才可以使用这个寄存器。

MCA 系统

这个寄存器保存了可编程选项选择配置字节 3，POST 将它发送到适配卡插槽 1。这个字节直接复制于扩展 MCA CMOS 寄存器 27h。

寄存器	描述	类型
27h	供应商专用	CMOS

这是一个供应商专用的地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 11-11-92

这个寄存器保存了硬盘 1 用来保存写预补偿字的磁柱号的低部分。只有当硬盘 1 指定为类型 47（用户定义）时才可以使用这个寄存器。

Phoenix BIOS，日期 9-28-95

位		
	7 r/w = x	硬盘 3 的磁头数，指定类型为 47
	6 r/w = x	（值 0=1 个磁头；值 F=16 个磁头）
	5 r/w = x	
	4 r/w = x	
	3 r/w = x	硬盘 2 的磁头数，指定类型为 47
	2 r/w = x	（值 0=1 个磁头；值 F=16 个磁头）
	1 r/w = x	
	0 r/w = x	

MCA 系统

这个寄存器保存了可编程选项选择配置字节 4，POST 将它发送到适配卡插槽 1。这个字节直接复制于扩展 MCA CMOS 寄存器 28h。

寄存器	描述	类型
28h	供应商专用	CMOS

这是一个供应商专用的地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 11-11-92

这个寄存器保存了硬盘 1 用来保存写预补偿字的磁柱号的高部分。只有当硬盘 1 指定为类型 47（用户定义）时才可以使用这个寄存器。

Phoenix BIOS，日期 9-28-95

位		
	7 r/w = 0	未使用或未知功能
	6 r/w = x	多扇区传输，硬盘 1
	5 r/w = x	0 = 禁止
	4 r/w = x	1 = 2 个扇区
		2 = 4 个扇区
		3 = 8 个扇区
		4 = 16 个扇区
	3 r/w = 0	未使用或未知功能

2 r/w = x		多扇区传输，硬盘 0
1 r/w = x		0 = 禁止
0 r/w = x		1 = 2 个扇区
		2 = 4 个扇区
		3 = 8 个扇区
		4 = 16 个扇区

MCA 系统

这个寄存器保存了可编程选项选择配置字节 5，POST 将它发送到适配卡插槽 1。这个字节直接复制于扩展 MCA CMOS 寄存器 29h。

寄存器	描述	类型
29h	供应商专用	CMOS

这是一个供应商专用的地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 11-11-92

这个寄存器保存了硬盘 1 的磁盘信息标志。只有当硬盘 1 指定为类型 47（用户定义）时才可以使用这个寄存器。

位	7 r/w = 1	无重试
	6 r/w = 1	无重试
	5 r/w = 1	在最后磁柱加 1 上的坏映射
	4 r/w = 0	未使用
	3 r/w = 1	如果磁头数大于 8
	2 r/w = 0	未使用
	1 r/w = 0	未使用
	0 r/w = 0	未使用

5Phoenix BIOS，日期 9-28-9

位	7 r/w = 0	未使用或未知功能
	6 r/w = x	多扇区传输，硬盘 3
	5 r/w = x	0 = 禁止
	4 r/w = x	1 = 2 个扇区
		2 = 4 个扇区
		3 = 8 个扇区
		4 = 16 个扇区
	3 r/w = 0	未使用或未知功能

2 r/w = x 1 r/w = x 0 r/w = x	多扇区传输，硬盘 2 0 = 禁止 1 = 2 个扇区 2 = 4 个扇区 3 = 8 个扇区 4 = 16 个扇区

MCA 系统

这个寄存器保存了可编程选项选择配置字节 2，POST 将它发送到适配卡插槽 2。这个字节直接复制于扩展 MCA CMOS 寄存器 49h。

寄存器	描述	类型
2Ah	供应商专用	CMOS

这是一个供应商专用的地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 11-11-92

这个寄存器保存了硬盘 1 用来用来装入区域字的磁柱号的低部分。只有当硬盘 1 指定为类型 47（用户定义）时才可以使用这个寄存器。

MCA 系统

这个寄存器保存了可编程选项选择配置字节 3，POST 将它发送到适配卡插槽 2。这个字节直接复制于扩展 MCA CMOS 寄存器 4Ah。

寄存器	描述	类型
2Bh	供应商专用	CMOS

这是一个供应商专用的地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 11-11-92

这个寄存器保存了硬盘 1 用来装入区域字的磁柱号的高部分。只有当硬盘 1 指定为类型 47（用户定义）时才可以使用这个寄存器。

Phoenix BIOS，日期 1-11-96

这个寄存器保存了硬盘 0 用来保存写预补偿字的磁柱号的低部分。只有当硬盘 0 指定为类型 47（用户定义）时才可以使用这个寄存器。

MCA 系统

这个寄存器保存了可编程选项选择配置字节 4，POST 将它发送到适配卡插槽 2。这个字节直接复制于扩展 MCA CMOS 寄存器 4Bh。

寄存器	描述	类型
2Ch	供应商专用	CMOS

这是一个供应商专用的地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 11-11-92

这个寄存器保存了硬盘 1 每磁道的扇区数。只有当硬盘 1 指定为类型 47（用户定义）时才可以使用这个寄存器。

Phoenix BIOS，日期 1-11-96

这个寄存器保存了硬盘 0 用来保存写预补偿字的磁柱号的高部分。只有当硬盘 0 指定为类型 47（用户定义）时才可以使用这个寄存器。

MCA 系统

这个寄存器保存了可编程选项选择配置字节 5，POST 将它发送到适配卡插槽 2。这个字节直接复制于扩展 MCA CMOS 寄存器 4Ch。

寄存器	描述	类型
2Dh	供应商专用	CMOS

这是一个供应商专用的地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 11-11-92

该字节保存了各种系统标志，如下所示：

位	7 r/w = 1	存在 Weitek 数学协处理器
	6 r/w = 0	引导时普通的软盘搜索
	1	快速引导时跳过软盘搜索
	5 r/w = 0	系统引导顺序，先驱动器 C:，然后驱动器 A:
	1	系统引导顺序，先驱动器 A:，然后驱动器 C:
	4 r/w = 0	引导时慢速系统时钟
	1	引导时快速系统时钟（普通）
	3 r/w = 1	开放外部高速缓存内存
	2 r/w = 1	开放内部高速缓存内存
	1 r/w = 0	通过键盘控制器控制普通的 A20 门
	1	快速 A20 门开关支持绕过慢速的键盘 A20 控制（只有在主板支持快速 A20 控制时）
	0 r/w = 0	禁止系统 turbo 切换
	1	支持系统 turbo 切换功能

MCA 系统

这个寄存器保存了可编程选项选择配置字节 2，POST 将它发送到适配卡插槽 3。这个字节直接复制于扩展 MCA CMOS 寄存器 6Ch。

寄存器	描述	类型
2Eh	求和校验——高	CMOS

非 MCA 系统

这个寄存器和寄存器 2Fh 一起组成一个字来指示 10h~2Dh 的 CMOS 寄存器总数。

MCA 系统

这个寄存器保存了可编程选项选择配置字节 3，POST 将它发送到适配卡插槽 3。这个字节直接复制于扩展 MCA CMOS 寄存器 6Dh。

寄存器	描述	类型
2Fh	求和校验——低	CMOS

非 MCA 系统

这个寄存器和地址 2Eh 一起组成一个字来指示 10h~2Dh 的 CMOS 寄存器总数。

MCA 系统

这个寄存器保存了可编程选项选择配置字节 4，POST 将它发送到适配卡插槽 3。这个字节直接复制于扩展 MCA CMOS 寄存器 6Eh。

寄存器	描述	类型
30h	检测到的扩展内存——低	CMOS

非 MCA 系统

和寄存器 31h 组成一个字来指示扩展内存的大小，单位 1,024 字节。每次运行 POST 时计算该值。例如，3C00h 表示有 15MB 的扩展内存。许多老式的 BIOS 最大只能处理 15MB 的扩展内存。大多数新式的 BIOS 可以接受多达 63MB 的扩展内存。中断 15h，功能 88h 用来从 CMOS 获取这个字。

MCA 系统

这个寄存器保存了可编程选项选择配置字节 5，POST 将它发送到适配卡插槽 3。这个字节直接复制于扩展 MCA CMOS 寄存器 6Fh。

寄存器	描述	类型
31h	检测到的扩展内存——高	CMOS

非 MCA 系统

和寄存器 30h 组成一个字来指示扩展内存的大小，单位 1,024 字节。参看 30h 了解更多信息。

MCA 系统

这个寄存器为主板控制可编程选项选择字节 2。

寄存器	描述	类型
32h	世纪/求和校验	CMOS

非 MCA 系统

这个寄存器包含了世纪的 BCD 码值。例如 1995 年会对应值 19h。

MCA 系统

这个寄存器保存了 CRC 字的高字节。CRC 覆盖了 CMOS RAM 字节 10h~31h。

寄存器	描述	类型
33h	供应商专用	CMOS

这是一个供应商专用的 CMOS 地址，某些系统可能没有使用这个寄存器。

IBM AT（早期的）

早期的 IBM AT 如下定义了这些字节：

位	7 r/w = 0	512K 基本内存系统
	1	640 K 基本内存系统
	6 r/w = 1	在 SETUP 程序中显示一条用户信息
	5 r/w = 0	未使用
	4 r/w = 0	未使用
	3 r/w = 0	未使用
	2 r/w = 0	未使用
	1 r/w = 0	未使用
	0 r/w = 0	未使用

AMI BIOS，日期 12-12-91

位	7 r/w = 1	640K 系统（与一个 512K 系统相对）	
	6 r/w = 0	未使用或未知功能	
	5 r/w = x	慢速 RAM 刷新选项	
	4 r/w = x	位 5	位 4
		0	0 = 15 S
		0	1 = 30 S
		1	0 = 60 S
		1	1 = 120 S
	3 r/w = 1	开放系统高速缓存	
	2 r/w = 1	开放视频高速缓存	

	1 r/w = 0	未使用或未知功能
	0 r/w = 0	未使用或未知功能

AMI BIOS，日期 6-6-92

位	7 r/w = 1	640K 系统（与一个 512K 系统相对）
	6 r/w = 0	未使用或未知功能
	5 r/w = 0	DRAM 写 CAS 脉冲 = 1T（20/25MHz）
	1	DRAM 写 CAS 脉冲 = 2T（33~55MHz）
	4 r/w = 0	未使用或未知功能
	3 r/w = 1	开放自动配置功能——安装菜单支持一个选项。将所有的 BIOS 缺省值或者维护值装入到 CMOS 中。维护值是最稳定的值，可以设置它，但是通常提供最差的执行速度。
	2 r/w = 0	未使用或未知功能
	1 r/w = 0	未使用或未知功能
	0 r/w = 0	未使用或未知功能

AMI BIOS，日期 11-11-92

这个寄存器设置为值 0FBh，不随着设置选项而改变。

MCA 系统

这个寄存器保存了 CRC 字的低字节。CRC 覆盖了 CMOS RAM 字节 10h~31h。

寄存器	描述	类型
34h	供应商专用	CMOS

这是一个供应商专用的 CMOS 地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 6-6-92

位	7 r/w = 1	开放引导扇区病毒保护（某些系统也使用这一位作为
	6 r/w = 1	引导时要求密码（如果支持）
	5 r/w = 1	在 C800 处影映（shadow）32K 字节的适配器 ROM
	4 r/w = 1	慢速 CPU，25MHz 以下
	3 r/w = 1	在 D000 处影映 32K 字节的适配器 ROM
	2 r/w = 0	未使用或未知功能
	1 r/w = 0	在 D800 处影映 32K 字节的适配器 ROM
	0 r/w = 0	开放 BIOS 高速缓存选项（某些版本未使用）

AMI BIOS，日期 11-11-92

这个寄存器设置为值 0，不随着设置选项而改变。

MCA 和 PS/1 系统

位		
	7 r/w = x	POST 检测到的串口数（0~8）
	6 r/w = x	
	5 r/w = x	
	4 r/w = x	
	3 r/w = x	在重启系统返回实模式之前，BIOS 中断 15h，功能 87h，访问扩展内存的状态
	2 r/w = x	
	1 r/w = x	
	0 r/w = x	

寄存器	描述	类型
35h	供应商专用	CMOS

这是一个供应商专用的 CMOS 地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 12-12-91

位		
	7 r/w = 1	在 E000 处影映 32K 字节的适配器 ROM
	6 r/w = 0	未使用或未知功能
	5 r/w = 1	在 E800 处影映 32K 字节的适配器 ROM
	4 r/w = 0	未使用或未知功能
	3 r/w = 1	开放自动配置功能
	2 r/w = 1	在 C000 处影映 32K 字节的视频 ROM
	1 r/w = 0	未使用或未知功能
	0 r/w = 0	未使用或未知功能

AMI BIOS，日期 6-6-92

位		
	7 r/w = 1	在 E000 处影映 32K 字节的适配器 ROM
	6 r/w = 0	DRAM 写周期，0 等待状态（20/25MHz）
	1	DRAM 写周期，1 等待状态（33~55MHz）
	5 r/w = 1	在 E800 处影映 32K 字节的适配器 ROM
	4 r/w = 0	未使用或未知功能
	3 r/w = 1	开放视频高速缓存选项
	2 r/w = 1	在 C000 处影映 32K 字节的视频 ROM
	1 r/w = 0	未使用或未知功能
	0 r/w = 0	未使用或未知功能

AMI BIOS，日期 11-11-92

这个寄存器设置为值 0Fh，不随着设置选项而改变。

寄存器	描述	类型
36h	供应商专用	CMOS

这是一个供应商专用的 CMOS 地址，某些系统可能没有使用这个寄存器。

AMI BIOS，日期 12-12-91

位 7 r/w = 0　未使用或未知功能

位 6 r/w = 0　未使用或未知功能

位 5 r/w = x, 位 4 r/w = x, 位 3 r/w = x　ISA 总线速度

位 5	位 4	位 3
0	0	0 = 7.15 MHz
0	0	1 = CLK/2（2 MHz）
0	1	0 = CLK/3（3 MHz）
0	1	1 = CLK/4（4 MHz）
1	0	0 = CLK/5（5 MHz）
1	0	1 = CLK/6（6 MHz）
1	1	0 = CLK/8（8 MHz）
1	1	1 = CLK/10（10 MHz）

位 2 r/w = x, 位 1 r/w = x　16 位 ISA 等待状态

位 2	位 1
0	0 = 1 个等待状态
0	1 = 2 个等待状态
1	0 = 3 个等待状态
1	1 = 4 个等待状态

位 0 r/w = 0　未使用或未知功能

AMI BIOS，日期 6-6-92

位 7 r/w = x, 位 6 r/w = x　DRAM 速度选项

位 7	位 6
0	0 = 最慢（50MHz）
0	1 = 较慢（40MHz）
1	0 = 较快（33MHz）
1	1 = 最快（25MHz）

位 5 r/w = 0　未使用或未知功能

位 4 r/w = 0　高速缓存读周期，1T（20~25MHz 或 33MHz 带 64/256KB 高速缓存

1　高速缓存读周期，2T（40~50MHz 或 33MHz 带 128KB 高速缓存

3 r/w = 0	高速缓存写周期，3T（33~55MHz）		
1	高速缓存读周期，2T（20~25MHz）		
2 r/w = x	ISA 总线速度		
1 r/w = x	位 2	位 1	位 0
0 r/w = x	0	0	0 = 7.15 MHz
	0	1	0 = CLK/4
	1	1	1 = CLK/6
	1	0	0 = CLK/5

AMI BIOS，日期 11-11-92

这个寄存器设置为值 0Fh，不随着设置选项而改变。

EISA 系统

在某些 EISA 系统上，地址 36h~3Fh 用于密码控制，并且一旦 POST 完成后 BIOS 不可访问这些地址。参看第 13 章，系统功能，端口 92h。

MCA 系统

这个字节保存了 POST 检测到的扩展内存大小的高字节。高字节和低字节组合再乘以 1,024 就得到扩展内存的总字节数。

寄存器	描述	类型
37h	世纪日期	CMOS

这是一个供应商专用的 CMOS 地址，某些系统可能没有使用这个寄存器。

EISA 系统

在某些 EISA 系统上，这个字节用于密码操作中。参看寄存器 36h。

MCA 系统

这个字节包含有世纪日期的 BCD 码值。例如，1997 年对应于值 19h。

寄存器	描述	类型
38h	供应商专用	CMOS

这是一个供应商专用的 CMOS 地址，某些系统可能没有使用这个寄存器。

EISA 系统

在某些 EISA 系统上，这个字节用于密码操作中。参看寄存器 36h。

MCA 系统

在上电操作期间，每个 POST 代码都保存在这里。参看端口 680h 下的为 PS/2 MCA 系统所编写的 POST 代码。

寄存器	描述	类型
39h	供应商专用	CMOS

这是一个供应商专用的 CMOS 地址，某些系统可能没有使用这个寄存器。

EISA 系统

在某些 EISA 系统上，这个字节用于密码操作中。参看寄存器 36h。

MCA 系统

在某些 MCA 系统上，密码保存在这个寄存器中。

寄存器	描述	类型
3Ah	供应商专用	CMOS

这是一个供应商专用的 CMOS 地址，某些系统可能没有使用这个寄存器。

EISA 系统

在某些 EISA 系统上，这个字节用于密码操作中。参看寄存器 36h。

MCA 系统

在某些 MCA 系统上，密码保存在这个寄存器中。

寄存器	描述	类型
3Bh	供应商专用	CMOS

这是一个供应商专用的 CMOS 地址，某些系统可能没有使用这个寄存器。

EISA 系统

在某些 EISA 系统上，这个字节用于密码操作中。参看寄存器 36h。

MCA 系统

在某些 MCA 系统上，密码保存在这个寄存器中。

寄存器	描述	类型
3Ch	供应商专用	CMOS

这是一个供应商专用的 CMOS 地址，某些系统可能没有使用这个寄存器。

Award BIOS，日期 9-14-87

位				
	7 r/w = 0	未使用或未知功能		
	6 r/w = 0	未使用或未知功能		
	5 r/w = x	系统速度选择		
	4 r/w = x	位 5	位 4	位 3
	3 r/w = x	0	0	0 = 无变化

0	0	1 = 低速
1	1	1 = 高速

2 r/w = x
1 r/w = x
0 r/w = x

POST 期间出错时采取的动作

位 2	位 1	位 0
0	0	1 = 任何错误都停机
0	1	0 = 任何错误都不停机
0	1	1 = 除了键盘错误外，所有的错误都停机
1	1	1 = 除了磁盘或键盘错误外，所有的错误都停机

EISA 系统

在某些 EISA 系统上，这个字节用于密码操作中。参看寄存器 36h。

MCA 系统

在某些 MCA 系统上，密码保存在这个寄存器中。

寄存器	描述	类型
3Dh	供应商专用	CMOS

这是一个供应商专用的 CMOS 地址，某些系统可能没有使用这个寄存器。

EISA 系统

在某些 EISA 系统上，这个字节用于密码操作中。参看寄存器 36h。

MCA 系统

在某些 MCA 系统上，密码保存在这个寄存器中。

寄存器	描述	类型
3Eh	供应商专用	CMOS

这是一个供应商专用的 CMOS 地址，某些系统可能没有使用这个寄存器。

Award BIOS，日期 9-14-87

这个地址用作关键保留字节的单字节求和校验。

AMI BIOS（大多数）

这个寄存器保存了对寄存器 34h~3Dh 求和校验字的高字节。把寄存器 34h~3Dh 加起来，得到这个字的求和校验值。在使用扩展寄存器 40h~7Fh 的 AMI BIOS 上，也对某些扩展寄存器进行求和校验，然后把校验字的高字节保存在这里。

EISA 系统

在某些 EISA 系统上，这个字节用于密码操作中。参看寄存器 36h。

MCA 系统

在某些 MCA 系统上，密码保存在这个寄存器中。

寄存器	描述	类型
3Fh	供应商专用	CMOS

这是一个供应商专用的 CMOS 地址，某些系统可能没有使用这个寄存器。

AMI BIOS（大多数）

这个寄存器保存了对寄存器 34h~3Dh 求和校验字的低字节。把寄存器 34h~3Dh 加起来，得到这个字的求和校验值。在使用扩展寄存器 40h~7Fh 的 AMI BIOS 上，也对某些扩展寄存器进行求和校验，然后把校验字的高字节保存在这里。

EISA 系统

在某些 EISA 系统上，这个字节用于密码操作中。参看寄存器 36h。

MCA 系统

这个字节保存了 CMOS 字节 38h~3Eh 的求和校验值。如果这七个 CMOS 字节加起来为零，则忽略求和校验。

寄存器	描述	类型
40h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phenix BIOS，日期 9-28-95

这个字节保存了硬盘 0 的磁柱字的低部分。只有在硬盘 0 指定为类型 47h（用户定义）时才可以使用这个寄存器。

Phenix BIOS，日期 1-11-96

位		
7 r/w = x 6 r/w = x	硬盘 1 的传输模式	
	位 7	位 6
	0	0 = 标准
	0	1 = PIO 1
	1	0 = PIO 2
	1	1 = PIO 3
5 r/w = 0	未使用或未知功能	
4 r/w = x 3 r/w = x	硬盘 0 的传输模式	
	位 4	位 3

	0	0 = 标准
	0	1 = PIO 1
	1	0 = PIO 2
	1	1 = PIO 3
2 r/w = 0	未使用或未知功能	
1 r/w = 0	未使用或未知功能	
0 r/w = 0	未使用或未知功能	

寄存器	描述	类型
41h	供应商专用	扩展CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phenix BIOS，日期 9-28-95

这个字节保存了硬盘0的磁柱字的高部分。只有在硬盘0指定为类型47h（用户定义）时才可以使用这个寄存器。

Phenix BIOS，日期 1-11-96

位			
	7 r/w = 0	未使用或未知功能	
	6 r/w = 0	未使用或未知功能	
	5 r/w = 0	未使用或未知功能	
	4 r/w = 0	未使用或未知功能	
	3 r/w = 0	未使用或未知功能	
	2 r/w = 0	未使用或未知功能	
	1 r/w = x 0 r/w = x	键盘重复速率（参看寄存器42h）	
		位1 位0	位7（寄存器42 h）
		0 0	0 = 30 cps
		0 0	1 = 26.7 cps
		0 1	0 = 21.8 cps
		0 1	1 = 18.5 cps
		1 0	0 = 13.3 cps
		1 0	1 = 10 cps
		1 1	0 = 6cps
		1 1	1 = 2 cps

寄存器	描述	类型
42h	供应商专用	扩展CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phenix BIOS，日期 9-28-95

这个字节保存了硬盘 1 的磁柱字的低部分。只有在硬盘 1 指定为类型 47h（用户定义）时才可以使用这个寄存器。

Phenix BIOS，日期 1-11-96

位		
	7 r/w = x	键盘重复速率（参看寄存器 41h）
	6 r/w = 0	未使用或未知功能
	5 r/w = 0	未使用或未知功能
	4 r/w = 0	未使用或未知功能
	3 r/w = 0	未使用或未知功能
	2 r/w = 0	未使用或未知功能
	1 r/w = 0	未使用或未知功能
	0 r/w = 0	未使用或未知功能

寄存器	描述	类型
43h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phenix BIOS，日期 9-28-95

这个字节保存了硬盘 1 的磁柱字的高部分。只有在硬盘 1 指定为类型 47h（用户定义）时才可以使用这个寄存器。

寄存器	描述	类型
44h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phenix BIOS，日期 9-28-95

这个字节保存了硬盘 1 用来保存写预补偿字的磁柱号的低部分。只有在硬盘 1 指定为类型 47h（用户定义）时才可以使用这个寄存器。这个寄存器和寄存器 45h 一起组成一个字。值 0FFFh 表示没有写预补偿。

寄存器	描述	类型
45h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phenix BIOS，日期 9-28-95

这个字节保存了硬盘 1 用来保存写预补偿字的磁柱号的高部分。只有在硬盘 1 指定为类型 47h（用户定义）时才可以使用这个寄存器。这个寄存器和寄存器 44h 一起组成一个字。值 0FFFh 表示没有写预补偿。

寄存器	描述	类型
46h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phenix BIOS，日期 9-28-95

这个字节保存了硬盘 2 磁柱字的低部分。

寄存器	描述	类型
47h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phenix BIOS，日期 9-28-95

这个字节保存了硬盘 2 磁柱字的低部分。

Phonix BIOS，日期 1-11-96

位		
	7 r/w = 0	未使用或未知功能
	6 r/w = 0	UART 2，红外模式
	1	UART 2，标准模式
	5 r/w = 0	未使用或未知功能
	4 r/w = 0	未使用或未知功能
	3 r/w = 0	未使用或未知功能
	2 r/w = 0	未使用或未知功能
	1 r/w = 0	未使用或未知功能
	0 r/w = 0	未使用或未知功能

寄存器	描述	类型
48h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phonix BIOS，日期 1-11-96

位			
	7 r/w = x 6 r/w = x	键盘重复延迟 位 7	位 6
		0	0 = 250 ms 延迟
		0	1 = 500 ms 延迟
		1	0 = 750 ms 延迟
		1	1 = 1 秒 延迟
	5 r/w = x 4 r/w = x	引导序列 位 5	位 4

	0	0 = A:引导，如果存在；否则 C:引导
	0	1 = 只从 C:引导
	1	0 = C:引导，如果存在；否则 A:引导
3 r/w = x 2 r/w = x	LPT 端口模式	
	位 3	位 2
	0	0 = 仅输出
	0	1 = 双向
	1	1 = ECP
1 r/w = x 0 r/w = x	主板 IDE 控制器	
	位 3	位 2
	0	0 = 禁止
	0	1 = 设置为主控制器
		= 设置为从控制器

寄存器	描述	类型
51h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phonix BIOS，日期 1-11-96

位	7 r/w = 0	其他大型的磁盘模式
	1	DOS 的大型磁盘模式
	6 r/w = 0	未使用或未知功能
	5 r/w = 0	未使用或未知功能
	4 r/w = 0	未使用或未知功能
	3 r/w = 0	未使用或未知功能
	2 r/w = 0	未使用或未知功能
	1 r/w = 0	未使用或未知功能
	0 r/w = 0	未使用或未知功能

寄存器	描述	类型
52h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phonix BIOS，日期 1-11-96

位	7 r/w = 1	开放 BIOS 归纳显示
	6 r/w = 1	引导检查时开放软盘搜索
	5 r/w = 1	如果出现 POST 错误，则停止
	4 r/w = 1	引导时激活显示，以便进入设置

	3 r/w = 0	未使用或未知功能
	2 r/w = 0	未使用或未知功能
	1 r/w = 1	支持即插即用特性
	0 r/w = 1	开放软盘控制器

寄存器	描述	类型
53h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phonix BIOS，日期 1-11-96

位	7 r/w = 0	未使用或未知功能
	6 r/w = 0	未使用或未知功能
	5 r/w = 1	开放硬盘 1 的 LBA 模式
	4 r/w = 1	开放硬盘 0 的 LBA 模式
	3 r/w = 0	未使用或未知功能
	2 r/w = 0	未使用或未知功能
	1 r/w = 1	硬盘 1 支持 32 位访问
	0 r/w = 1	硬盘 0 支持 32 位访问

寄存器	描述	类型
54h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phonix BIOS，日期 1-11-96

位	7 r/w = 1	硬盘 0 上的写保护 MBR
	6 r/w = 0	未使用或未知功能
	5 r/w = 0	由用户设置的硬盘 1 参数
	1	自动设置的硬盘 1 参数
	4 r/w = 0	Num Lock 关
	1	Num Lock 开
	3 r/w = 0	未使用或未知功能
	2 r/w = 0	未使用或未知功能
	1 r/w = 0	未使用或未知功能
	0 r/w = 0	由用户设置的硬盘 0 参数
	1	自动设置的硬盘 0 参数

寄存器	描述	类型
58h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phonix BIOS，日期 9-28-95

位		
7 r/w = x 6 r/w = x	硬盘 0 的传输模式 位 7　位 6 0　0 = 标准 0　1 = PIO1 1　0 = PIO2 1　1 = PIO3	
5 r/w = 0	未使用或未知功能	
4 r/w = 0	未使用或未知功能	
3 r/w = 0	未使用或未知功能	
2 r/w = 0	未使用或未知功能	
1 r/w = 0	未使用或未知功能	
0 r/w = 0	未使用或未知功能	

寄存器	描述	类型
59h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phonix BIOS，日期 9-28-95

位		
7 r/w = x	硬盘 3 的传输模式（参看寄存器 62h 了解状态）	
6 r/w = 0	未使用或未知功能	
5 r/w = x 4 r/w = x	硬盘 2 的传输模式 位 5　位 4 0　0 = 标准 0　1 = PIO1 1　0 = PIO2 1　1 = PIO3	
3 r/w = 0	未使用或未知功能	
2 r/w = 0	未使用或未知功能	
1 r/w = 0	未使用或未知功能	
0 r/w = 0	未使用或未知功能	

寄存器	描述	类型
5Ah	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phonix BIOS，日期 9-28-95

位		
	7 r/w = 0	未使用或未知功能
	6 r/w = 0	未使用或未知功能
	5 r/w = 0	未使用或未知功能
	4 r/w = 0	未使用或未知功能
	3 r/w = 0	未使用或未知功能
	2 r/w = 0	未使用或未知功能
	1 r/w = 0	未使用或未知功能
	0 r/w = 0	开放对硬盘 3 的 32 位访问

寄存器	描述	类型
5Dh	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phonix BIOS，日期 9-28-95

位		
	7 r/w = x 6 r/w = x	Mum Lock 状态 位 7　位 6 0　0 = 自动 0　1 = 开 1　0 = 关
	5 r/w = 0	未使用或未知功能
	4 r/w = 0	引导顺序——从 A:引导，如果存在；否则从 C:引导 引导顺序——从 C:引导，如果存在；否则从 A:引导
	3 r/w = 0	未使用或未知功能
	2 r/w = 0	未使用或未知功能
	1 r/w = 0	未使用或未知功能
	0 r/w = 0	未使用或未知功能

寄存器	描述	类型
5Eh	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phonix BIOS，日期 9-28-95

位		
	7 r/w = 1	在 D800h 处影映 16KB 的内存
	6 r/w = 1	在 D400h 处影映 16KB 的内存

	5 r/w = 1	在 D000h 处影映 16KB 的内存
	4 r/w = 1	在 CC00h 处影映 16KB 的内存
	3 r/w = 1	在 C800h 处影映 16KB 的内存
	2 r/w = 0	未使用或未知功能
	1 r/w = 0	未使用或未知功能
	0 r/w = 0	未使用或未知功能

寄存器	描述	类型
5Fh	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phonix BIOS，日期 9-28-95

位	7 r/w = 0	未使用或未知功能
	6 r/w = 0	未使用或未知功能
	5 r/w = 0	未使用或未知功能
	4 r/w = 0	未使用或未知功能
	3 r/w = 0	未使用或未知功能
	2 r/w = 0	未使用或未知功能
	1 r/w = 1	在 DC00h 处影映 16KB 的内存

寄存器	描述	类型
60h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phonix BIOS，日期 9-28-95

位	7 r/w = 0	未使用或未知功能
	6 r/w = 0	未使用或未知功能
	5 r/w = 0	未使用或未知功能
	4 r/w = 1	在 C000h 处影映 32KB 的内存
	3 r/w = 0	未使用或未知功能
	2 r/w = 1	开放高速缓存
	1 r/w = 0	未使用或未知功能

寄存器	描述	类型
61h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phonix BIOS，日期 9-28-95

位	7 r/w = 1	开放对硬盘 2 的 32 位访问
	6 r/w = 1	开放对硬盘 1 的 32 位访问
	5 r/w = 1	开放对硬盘 0 的 32 位访问
	4 r/w = 0	未使用或未知功能
	3 r/w = 0	未使用或未知功能
	2 r/w = 0	未使用或未知功能
	1 r/w = 0	未使用或未知功能
	0 r/w = 0	未使用或未知功能

寄存器	描述	类型
62h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phonix BIOS，日期 9-28-95

位	7 r/w = 0	由用户设置硬盘 2 的参数
	1	自动设置硬盘 2 的参数
	6 r/w = 0	未使用或未知功能
	5 r/w = 0	未使用或未知功能
	4 r/w = 1	硬盘 3 开放 LBA 模式
	3 r/w = 1	硬盘 2 开放 LBA 模式
	2 r/w = 1	硬盘 1 开放 LBA 模式
	1 r/w = 1	硬盘 0 开放 LBA 模式
	0 r/w = x	位 0　位 7（寄存器 59h）
		0　0 = 标准
		0　1 = PIO1
		1　0 = PIO2
		1　1 = PIO3

寄存器	描述	类型
63h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

AMI BIOS，日期 11-11-92

位	7 r/w = 0	未使用或未知功能
	6 r/w = 0	未使用或未知功能
	5 r/w = 1	开放快速系统定时
	4 r/w = 1	对 VL 返回准备好允许触发准备好脉冲

	3 r/w = 0	未使用或未知功能
	2 r/w = 1	开放 15MB 处的上存切断功能
	1 r/w = 0	高速缓存设置为写回模式
	1	高速缓存设置为写通模式
	0 r/w = 1	允许高速 ISA 总线（快速）

AMI BIOS，日期 1-11-96

位	7 r/w = 0	未使用或未知功能
	6 r/w = 1	PCI 插槽 1——激活了缺省的潜伏时间时钟
	5 r/w = 0	未使用或未知功能
	4 r/w = 0	未使用或未知功能
	3 r/w = 0	未使用或未知功能
	2 r/w = 0	未使用或未知功能
	1 r/w = 0	未使用或未知功能
	0 r/w = 0	未使用或未知功能

寄存器	描述	类型
64h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

AMI BIOS，日期 11 -11 -92

位	7 r/w = 0	未使用或未知功能
	6 r/w = 0	未使用或未知功能
	5 r/w = 0	未使用或未知功能
	4 r/w = x 3 r/w = x 2 r/w = x	ISA 总线速度 vs. CPU 速度 位 4　位 3　位 2 0　0　0 = 自动 0　1　0 = 1/6 0　1　1 = 1/5 1　0　0 = 1/4 0　0　1 = 1/3 1　1　0 = 1/2
	1 r/w = 0	未使用或未知功能
	0 r/w = 0	未使用或未知功能

AMI BIOS，日期 1-11-96

位	7 r/w = 0	未使用或未知功能
	6 r/w = x	PCI 插槽 1——潜伏时间时钟

5 r/w = x	要获得时钟值 0~F8h，将这个值乘以 8
4 r/w = x	
3 r/w = x	
2 r/w = x	
1 r/w = x	
0 r/w = 1	PCI 插槽 1——主开放

寄存器	描述	类型
67h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

AMI BIOS，日期 1-11-96

位		
	7 r/w = x	硬盘 0 每磁道的扇区数，低部分
	6 r/w = x	（高两位在寄存器 68h 中）
	5 r/w = x	
	4 r/w = x	
	3 r/w = 0	未使用或未知功能
	2 r/w = 0	未使用或未知功能
	1 r/w = 0	未使用或未知功能
	0 r/w = 0	未使用或未知功能

寄存器	描述	类型
68h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

AMI BIOS，日期 1-11-96

位		
	7 r/w = x	硬盘 1 每磁道的扇区数
	6 r/w = x	
	5 r/w = x	
	4 r/w = x	
	3 r/w = x	
	2 r/w = x	
	1 r/w = x	硬盘 0 每磁道的扇区数，高部分
	0 r/w = x	（低四位在寄存器 67h 中）

寄存器	描述	类型
6Ah	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

AMI BIOS，日期 1-11-96

位		
	7 r/w = x	硬盘 0 的磁头数，指定类型为 47
	6 r/w = x	（值 0=1 个磁头；值 F=16 个磁头）
	5 r/w = x	
	4 r/w = x	
	3 r/w = 0	未使用或未知功能
	2 r/w = 0	未使用或未知功能
	1 r/w = 0	未使用或未知功能
	0 r/w = 0	未使用或未知功能

寄存器	描述	类型
6Bh	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

AMI BIOS，日期 1-11-96

位		
	7 r/w = 0	未使用或未知功能
	6 r/w = 0	未使用或未知功能
	5 r/w = 0	未使用或未知功能
	4 r/w = 0	未使用或未知功能
	3 r/w = x	硬盘 0 的磁头数，指定类型为 47
	2 r/w = x	（值 0=1 个磁头；值 F=16 个磁头）
	1 r/w = x	
	0 r/w = x	

寄存器	描述	类型
6Ch	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phoenix BIOS，日期 1-11-96

位		
	7 r/w = 0	未使用或未知功能
	6 r/w = x	多扇区传输，硬盘 0
	5 r/w = x	0 = 禁止
	4 r/w = x	1 = 2 个扇区
		2 = 4 个扇区
		3 = 8 个扇区
		4 = 16 个扇区

	3 r/w = 0	未使用或未知功能
	2 r/w = 0	未使用或未知功能
	1 r/w = 0	未使用或未知功能
	0 r/w = 0	未使用或未知功能

寄存器	描述	类型
6Dh	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phoenix BIOS，日期 1-11-96

位	7 r/w = 0	未使用或未知功能
	6 r/w = 0	未使用或未知功能
	5 r/w = 0	未使用或未知功能
	4 r/w = 0	未使用或未知功能
	3 r/w = 0	未使用或未知功能
	2 r/w = x	多扇区传输，硬盘 1
	1 r/w = x	0 = 禁止
	0 r/w = x	1 = 2 个扇区
		2 = 4 个扇区
		3 = 8 个扇区
		4 = 16 个扇区

寄存器	描述	类型
70h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

AMI BIOS，日期 11-11-92

位	7 r/w = x	内存影映区域，段址 C000h，32KB	
	6 r/w = x	位 7	位 6
		0	0 = 禁止影映
		0	1 = 只读影映
		1	0 = 可缓存
		1	1 = 内存影映
	5 r/w = x	内存影映区域，段址 C800h，32KB	
	4 r/w = x	位 5	位 4
		0	0 = 禁止影映
		0	1 = 只读影映

	1	0 = 可缓存
	1	1 = 内存影

3 r/w = x
2 r/w = x

内存影映区域，段址 D000h，32KB

位 3	位 2
0	0 = 禁止影映
0	1 = 只读影映
1	0 = 可缓存
1	1 = 内存影映

1 r/w = x
0 r/w = x

内存影映区域，段址 D800h，32KB

位 1	位 0
0	0 = 禁止影映
0	1 = 只读影映
1	0 = 可缓存
1	1 = 内存影映

寄存器	描述	类型
71h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

AMI BIOS，日期 11-11-92

位 7 r/w = x
位 6 r/w = x

内存影映区域，段址 E000h，64KB

位 7	位 6
0	0 = 禁止影映
0	1 = 只读影映
1	0 = 可缓存
1	1 = 内存影映

5 r/w = x
4 r/w = x

内存影映区域，段址 F000h，64KB

位 5	位 4
0	0 = 禁止影映
0	1 = 只读影映
1	0 = 可缓存
1	1 = 内存影映

3 r/w = x　未使用或未知功能

2 r/w = x
1 r/w = x
0 r/w = x

CPU 频率设置

位 2	位 1	位 0
0	0	0 = 25 MHz

0	0	1 = 33 MHz
0	1	0 = 40 MHz
0	1	1 = 50 MHz
1	0	1 = 66 MHz
1	1	1 = 自动检测

寄存器	描述	类型
72h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phenix BIOS，日 1-11-96

位				
7 r/w = x	COM 2 操作		
6 r/w = x	位 7	位 6	位 5
5 r/w = x	0	0	0 = 禁止
	0	1	0 = 使用端口 2F8h 和 IRQ 3
	1	0	0 = 使用端口 2E8h 和 IRQ 3
	1	0	1 = 自动
4 r/w = x	COM 1 操作		
3 r/w = x	位 4	位 3	位 2
2 r/w = x	0	0	0 = 禁止
	0	1	0 = 使用端口 3F8h 和 IRQ 4
	1	0	0 = 使用端口 3E8h 和 IRQ 4
	1	0	1 = 自动
1 r/w = 0	未使用或未知功能		
0 r/w = 0	未使用或未知功能		

寄存器	描述	类型
73h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phenix BIOS，日期 1-11-96

位	
7 r/w = 0	未使用或未知功能
6 r/w = 0	未使用或未知功能
5 r/w = 0	未使用或未知功能
4 r/w = 0	未使用或未知功能
3 r/w = 0	未使用或未知功能

	LPT 端口模式		
2 r/w = x	位 2	位 1	位 0
1 r/w = x	0	0	0 = 禁止
0 r/w = x	0	0	1 = 端口 378h，IRQ 7
	0	1	0 = 端口 278h，IRQ 7
	0	1	1 = 端口 3BCh，IRQ 7
	1	1	1 = 自动

寄存器	描述	类型
75h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phenix BIOS，日期 1-11-96

这个寄存器和寄存器 76h 及 77h 一起，保存了大多数（或全部）寄存器 40h~74h 的更正求和校验。

寄存器	描述	类型
76h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phenix BIOS，日期 1-11-96

这个寄存器和寄存器 75h 及 77h 一起，保存了大多数（或全部）寄存器 40h~74h 的更正求和校验。

寄存器	描述	类型
77h	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phenix BIOS，日期 1-11-96

这个寄存器和寄存器 75h 及 76h 一起，保存了大多数（或全部）寄存器 40h~74h 的更正求和校验。

寄存器	描述	类型
7Ah	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phenix BIOS，日期 9-28-95

位	7 r/w = 0	未使用或未知功能
	6 r/w = 0	未使用或未知功能
	5 r/w = 0	未使用或未知功能
	4 r/w = 0	未使用或未知功能
	3 r/w = 0	未使用或未知功能
	2 r/w = 0	未使用或未知功能
	1 r/w = 0	由用户设置的硬盘 1 参数
	1	自动设置的硬盘 1 参数
	0 r/w = 0	由用户设置的硬盘 0 参数
	1	自动设置的硬盘 0 参数

寄存器	描述	类型
7Ch	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phenix BIOS，日期 9-28-95

这个寄存器和寄存器 7Dh 一起，保存了大多数（或全部）寄存器 40h~7Bh 的更正求和校验。

寄存器	描述	类型
7Dh	供应商专用	扩展 CMOS

如果存在这个寄存器，那么它是一个供应商专用地址，也可能没有使用。

Phenix BIOS，日期 9-28-95

这个寄存器和寄存器 7Ch 一起，保存了大多数（或全部）寄存器 40h~7Bh 的更正求和校验。

端口	类型	描述	平台
71h	I/O	CMOS 内存数据	AT+

端口 70h 选择使用哪个 CMOS 内存地址。然后使用端口 71h 从以前选择的地址读或写数据。参看本章前面的端口 70h，READ_CMOS 和 WRITE_CMOS 例了解更多细节。

端口	类型	描述	平台
74h	输出	扩展 CMOS 地址 LSB	MCA

大多数 MCA 系统包含有另外 2K 的 CMOS RAM 内存区。访问这个内存时，最低有效地址字节输出到端口 74h 中。参看端口 76h 了解细节。

输出（位 0~7） 扩展 CMOS 地址 LSB

端口	类型	描述	平台
75h	输出	扩展 CMOS 地址 MSB	MCA

大多数 MCA 系统包含有另外 2K 的 CMOS RAM 内存区。访问这个内存时，最高有效地址字节输出到端口 74h 中。参看端口 76h 了解细节。

输出（位 0~7） 扩展 CMOS 地址 MSB

位		
	7 w = 0	未使用
	6 w = 0	未使用
	5 w = 0	未使用
	4 w = 0	未使用
	3 w = 0	未使用
	2 w = x	CMOS 扩展内存地址的高 3 位。
	1 w = x	
	0 w = x	

端口	类型	描述	平台
76h	I/O	扩展 CMOS 数据寄存器	MCA

这个寄存器用来从另外 2K 个扩展寄存器内存中写和读数据字节。必须向端口 74h 和 75h 写入一个 11 位地址，来选择一个地址。

这个扩展 CMOS 内存用于大多数 MCA 系统上的标准 CMOS 内存以外的附加配置和系统信息。这个附加 CMOS 内存的主要功能是保存适配器描述文件的信息。据我所知，在所有的 MCA 系统中，只有 IBM 50 型的计算机不支持扩展 MCA CMOS 内存。

寄存器归纳 下面这个列表中没有包括的扩展 MCA CMOS 内存寄存器一般很少使用，但是将来的系统可能会用到它。星号表示没有详细描述该寄存器，因为它们和该列表中前面描述的寄存器的功能相同。例如磁盘 1 寄存器的功能和磁盘 0 寄存器的功能相同，而不是和其他磁盘号的寄存器相同。

寄存器	描述
00h	卡插槽 0，LSB 适配器 ID
01h	卡插槽 0，MSB 适配器 ID
02h	卡插槽 0，使用的 POS 字节数
03h	卡插槽 0，POS 字节 2
04h	卡插槽 0，POS 字节 3
05h	卡插槽 0，POS 字节 4

06h	卡插槽0，POS字节5
23h*	卡插槽1，LSB适配器ID
24h*	卡插槽1，MSB适配器ID
25h*	卡插槽1，使用的POS字节数
26h*	卡插槽1，POS字节2
27h*	卡插槽1，POS字节3
28h*	卡插槽1，POS字节4
29h*	卡插槽1，POS字节5
46h*	卡插槽2，LSB适配器ID
47h*	卡插槽2，MSB适配器ID
48h*	卡插槽2，使用的POS字节数
49h*	卡插槽2，POS字节2
4Ah*	卡插槽2，POS字节3
4Bh*	卡插槽2，POS字节4
4Ch*	卡插槽2，POS字节5
69h*	卡插槽3，LSB适配器ID
6Ah*	卡插槽3，MSB适配器ID
6Bh*	卡插槽3，使用的POS字节数
6Ch*	卡插槽3，POS字节2
6Dh*	卡插槽3，POS字节3
6Eh*	卡插槽3，POS字节4
6Fh*	卡插槽3，POS字节5
8Ch*	卡插槽4，LSB适配器ID
8Dh*	卡插槽4，MSB适配器ID
8Eh*	卡插槽4，使用的POS字节数
8Fh*	卡插槽4，POS字节2
90h*	卡插槽4，POS字节3
91h*	卡插槽4，POS字节4
92h*	卡插槽4，POS字节5
AFh*	卡插槽5，LSB适配器ID
B0h*	卡插槽5，MSB适配器ID
B1h*	卡插槽5，使用的POS字节数
B2h*	卡插槽5，POS字节2
B3h*	卡插槽5，POS字节3
B4h*	卡插槽5，POS字节4
B5h*	卡插槽5，POS字节5
D2h*	卡插槽6，LSB适配器ID
D3h*	卡插槽6，MSB适配器ID
D4h*	卡插槽6，使用的POS字节数

D5h*	卡插槽 6，POS 字节 2
D6h*	卡插槽 6，POS 字节 3
D7h*	卡插槽 6，POS 字节 4
D8h*	卡插槽 6，POS 字节 5
F5h*	卡插槽 7，LSB 适配器 ID
F6h*	卡插槽 7，MSB 适配器 ID
F7h*	卡插槽 7，使用的 POS 字节数
F8h*	卡插槽 7，POS 字节 2
F9h*	卡插槽 7，POS 字节 3
FAh*	卡插槽 7，POS 字节 4
FBh*	卡插槽 7，POS 字节 5
161h	扩展 CMOS CRC 高字节
162h	扩展 CMOS CRC 低字节
163h	扩展内存的大小，LSB
164h	扩展内存的大小，ISB
165h	扩展内存的大小，MSB
166h	磁盘 0，最大磁柱数，LSB
167h	磁盘 0，最大磁柱数，MSB
168h	磁盘 0，最大磁头数
16Bh	磁盘 0，写预补偿磁柱，LSB
16Ch	磁盘 0，写预补偿磁柱，MSB
16Eh	磁盘 0，控制字节
172h*	磁盘 1，装入区域磁柱，LSB
173h*	磁盘 1，装入区域磁柱，MSB
174h*	磁盘 1，每磁道的扇区数
176h*	磁盘 1，最大磁柱数，LSB
177h*	磁盘 1，最大磁柱数，MSB
178h*	磁盘 1，最大磁头数
17Bh*	磁盘 1，写预补偿磁柱，LSB
17Ch*	磁盘 1，写预补偿磁柱，MSB
17Eh*	磁盘 1，控制字节
182h*	磁盘 1，装入区域磁柱，LSB
183h*	磁盘 1，装入区域磁柱，MSB
184h*	磁盘 1，每磁道的扇区数
186h	POST CMOS 存在性测试
18Eh	MCA 插槽总数
389h	错误记录号（0~5）
38Ah	错误记录组 0
39Eh*	错误记录组 1

3B2h*	错误记录组 2
3C6h*	错误记录组 3
3DAh*	错误记录组 4
3EEh*	错误记录组 5

寄存器细节

寄存器	描述	类型
00h	卡插槽 0，LSB 适配器 ID	扩展 MCA CMOS

这个寄存器保存了卡插槽 0 的适配卡 ID。这里保存了这个 ID 值的最低有效位字节。而在扩展寄存器 1 中保存了最高有效位字节。由适配器提供的参考软盘保存这个值。

寄存器	描述	类型
01h	卡插槽 0，MSB 适配器 ID	扩展 MCA CMOS

参看扩展 MCA 寄存器 0。

寄存器	描述	类型
02h	卡插槽 0，使用的 POS 字节	扩展 MCA CMOS

这个寄存器指定了可编程选项选择（POS）的字节数，这些字节由 POST 从扩展 CMOS 装入到卡插槽 0 的适配器中。在下面的扩展 MCA CMOS 寄存器中指定 POS 字节。

POS 字节最初由适配器提供的安装程序装入到 CMOS 中。它们所起的作用和 AT 类型卡上的开关相同，而且可能还包括了端口选择、使用 IRQ、DMA 通道以及其他系统和适配器专用的信息。

寄存器	描述	类型
03h	卡插槽 0，POS 字节 2	扩展 MCA CMOS

卡插槽 0 的可编程选项选择（POS）的字节 2。

寄存器	描述	类型
04h	卡插槽 0，POS 字节 3	扩展 MCA CMOS

卡插槽 0 的可编程选项选择（POS）的字节 3。

寄存器	描述	类型
05h	卡插槽 0，POS 字节 4	扩展 MCA CMOS

卡插槽 0 的可编程选项选择（POS）的字节 4。

*未作阐释，参看前文

寄存器	描述	类型
06h	卡插槽 0，POS 字节 5	扩展 MCA CMOS

卡插槽 0 的可编程选项选择（POS）的字节 6。

寄存器	描述	类型
161h	扩展 CMOS CRC 高字节	扩展 MCA CMOS

在这个寄存器和下一个寄存器中保存了一个字，作为 CRC 机制的一部分。选择好这个字的值，这样扩展 CMOS 寄存器 0~162h 的 CRC 成为 0。经计算后由参考软盘写入。

寄存器	描述	类型
162h	扩展 CMOS CRC 低字节	扩展 MCA CMOS

参看寄存器 161h。

寄存	描述	类型
163h	扩展内存的大小，LSB	扩展 MCA CMOS

一个系统的扩展内存大于 63MB 时，这个寄存器和下两个寄存器组成一个 24 位的值，来表示实际扩展内存的大小，单位是 1,024 字节。

早期的 AT 设计仅在常规的 CMOS 内存寄存器 17h 和 18h 中提供了一个字来表示扩展内存的大小。因此，这就将扩展内存局限于最大 64MB。使用 24 位的值，系统提供了高达 16GB 的扩展内存容量，在近一两年可能也不会超过这个上限。按每兆字节$3 计算，16GB 内存的价格是大约是$50,000。

寄存器	描述	类型
164h	扩展内存的大小，ISB	扩展 MCA CMOS

24 位扩展内存大小的中间字节，参看 163h。

寄存器	描述	类型
165h	扩展内存的大小，MSB	扩展 MCA CMOS

24 位扩展内存大小的最高有效位字节，参看 163h。

寄存器	描述	类型
166h	磁盘 0 的最大磁柱数，LSB	扩展 MCA CMOS

这个寄存器保存了磁盘 0 的磁柱总数的低部分。

寄存器	描述	类型
167h	磁盘 0 的最大磁柱数，MSB	扩展 MCA CMOS

这个寄存器保存了磁盘 0 的磁柱总数的高部分。

寄存器	描述	类型
168h	磁盘 0 的最大磁头数	扩展 MCA CMOS

这个寄存器保存了磁盘 0 的磁头总数。

寄存器	描述	类型
16Bh	磁盘 0 的写预补偿磁柱号，LSB	扩展 MCA CMOS

这个寄存器保存了磁盘 0 的写预补偿磁柱号的低部分。

寄存器	描述	类型
16Ch	磁盘 0 的写预补偿磁柱号，MSB	扩展 MCA CMOS

这个寄存器保存了磁盘 0 的写预补偿磁柱号的高部分。

寄存器	描述	类型
16Eh	磁盘 0 的控制字节	扩展 MCA CMOS

这个寄存器保存了磁盘 0 的磁盘信息标志。

位		
	7 r/w = 1	无重试
	6 r/w = 1	无重试
	5 r/w = 1	在最后一个磁柱加 1 上的坏映射
	4 r/w = 0	未使用
	3 r/w = 1	如果磁头数大于 8
	2 r/w = 0	未使用
	1 r/w = 0	未使用
	0 r/w = 0	未使用

寄存器	描述	类型
172h	磁盘 0 的装入区域磁柱，LSB	扩展 MCA CMOS

这个寄存器保存了磁盘 0 的装入区域磁柱的低部分。

寄存器	描述	类型
173h	磁盘 0 的装入区域磁柱，MSB	扩展 MCA CMOS

这个寄存器保存了磁盘 0 的装入区域磁柱的高部分。

寄存器	描述	类型
174h	磁盘 0 每磁道的扇区数	扩展 MCA CMOS

这个寄存器保存了磁盘 0 每磁道的扇区数。

寄存器	描述	类型
186h	POST CMOS 存在性测试	扩展 MCA CMOS

POST 在这个地址写入和读取一个值，来确认存在扩展 MCA CMOS。

寄存器	描述	类型
18Eh	MCA 插槽总数	扩展 MCA CMOS

系统上 MCA 适配器插槽的总数。

寄存器	描述	类型
389h	错误记录号（0~5）	扩展 MCA CMOS

POST 检测到错误保存在六个错误记录组中的一个当中。这个寄存器指定上次使用的是哪个记录组。

寄存器	描述	类型
38Ah	错误记录组 0	扩展 MCA CMOS

这个寄存器是 20 个寄存器的起点，用来保存来自系统 BIOS 的错误信息。

端口	类型	描述	平台
8xxh	I/O	扩展 CMOS 寄存器	EISA

在许多 EISA 系统上，端口 800h~8FFh 用来访问当前区的 256 个扩展 CMOS 内存地址。参看端口 C00h 了解其他细节。下面列出了 CMOS 扩展内存地址第 0 区的部分列表。

第 0 区端口	描述
800h	扩展配置位
801h	扩展配置位
802h	扩展配置位
803h	扩展配置位
821h	前 32 个地址的求和校验
87Ah	求和校验的高字节，除了这个求和校验字外的所有扩展 CMOS 字节（包括第 0 区，端口 800h~87Fh 处的字节）
87Bh	求和校验的低字节，除了这个求和校验字外的所有扩展 CMOS 字节（包括第 0 区，端口 800h~87Fh 处的字节）

端口	类型	描述	平台
C00h	I/O	扩展 CMOS 区选择	EISA

EISA 系统为扩展配置数据和插槽配置数据提供了另外的 8KB 的 CMOS RAM。EISA 的规范说明书没有指出访问这些寄存器的确切方法。个别制造商将使用的方法和端口留给读者自己去决定！

大多数的制造商使用端口 C00h 进行扩展内存区选择。这个 8K 的 CMOS RAM 分成 32 区，每个区包括了 256 个寄存器。第 0 区用来保存基本 CMOS 内存中没有的其他配置信息。

插槽配置信息使用其他的区（1~1Fh）。区选择的结果将保持不变，直到系统重启，或者出现另外一次区选择。可以读取当前选择的区。

访问扩展 CMOS 内存时，先选中一个区，然后对端口 800h~8FFh 中指定的一个寄存器输入或输出。在此期间，必须禁止中断。否则在选择一个区后而在访问指定的内存地址之前，可能会出现一个中断。如果中断服务程序改变了选择区，那么在当前的操作继续进行时，会访问一个错误的 CMOS 内存区。EISA BIOS 也可以访问扩展 CMOS 内存。中断 15h 服务就是用来实现这个功能的。本章开始有关“EISA 系统的不同之处”一节中详细阐释了这些内容。

第 16 章

系统时钟

系统时钟有多种用途，从系统定时到用作看门狗时钟。我将显示如何使用它们以及不同的操作模式。与其他技术参考手册不同，我还会显示实际上可能使用的属性，但是主板设计却为它们强加了许多限制。本章也指出了 AT、MCA 以及 EISA 系统之间微妙或不那么微妙的差别。

本章提供了一个工具来演示如何设置和使用时钟。它提供了非常精确的方法，来对所有 CPU 上的一个事件，例如一个子程序，精确定时。如果你使用的是 Pentium 或 Pentium Pro，那么你需要仔细阅读未公开的读时间标志计数器指令。第三章中描述的这个指令提供了一个更为精确的时钟。

简　介

每个 PC 系统至少包含有一个 8254 可编程时钟或等价的芯片。这个时钟包含有三个独立的 16 位时钟。时钟 0 用作基本系统时钟，时钟 1 用于 EISA 系统上的 DRAM 刷新，MCA 系统不支持这个时钟。时钟 2 用于一般的应用程序，例如扬声器音调控制。MCA 和 EISA 系统还提供了第四个计数器，时钟 3，用作看门狗时钟。EISA 系统还提供了第五个计数器，时钟 5（不是 4），用于可任选的 CPU 控制。前三个时钟的硬件连接有所不同，如图 16-1 所示。

图 16-2 显示了一个时钟通道的内部逻辑。每个时钟包含有一个模式和状态寄存器，并带有一个 16 位计数器。从一个独立的锁存器装入该计数器。在某些操作模式下，在每次计数完时，计数器会自动从这个锁存器重新载入。CPU 可以将计数器的内容锁存到计数器的输出锁存器中，以便读取。不同的操作模式控制不同寄存器、锁存器以及计数器的操作。

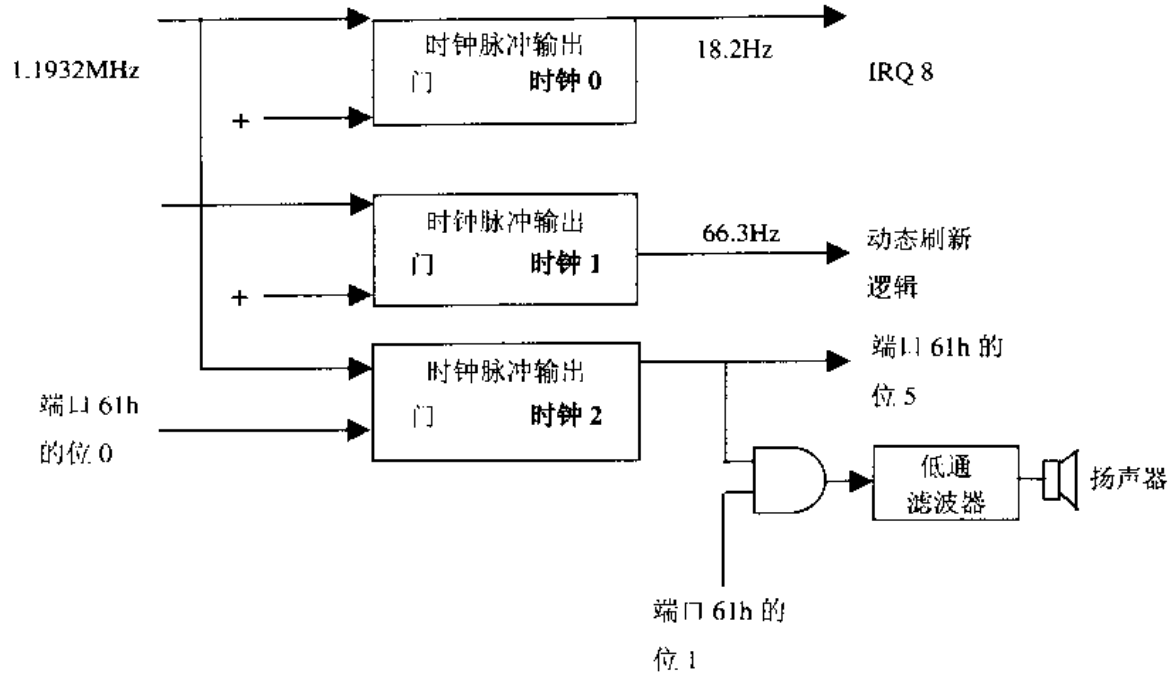


图 16-1 定时系统

操作模式

8254 时钟为每个通道提供了六种操作模式。由于芯片硬件连接的问题，某些模式对于特定的通道并不实用。下面所有的时序图都只显示了低 8 位计数。假定采用的是二进制计数模式，尽管作为一个可编程选项可以使用 BCD 计数。但是 BCD 计数很少使用，所以没有显示。

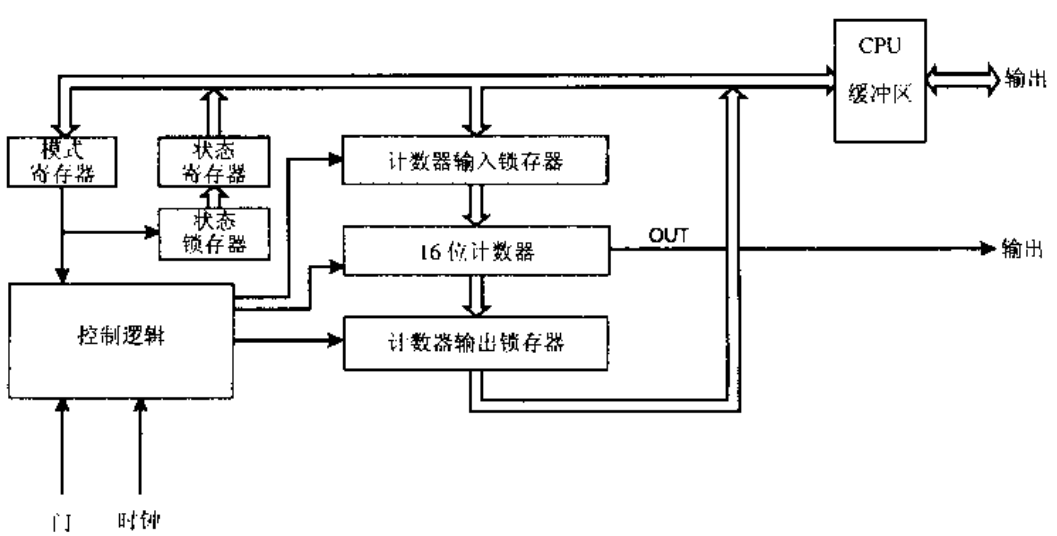


图 16-2 内部时钟逻辑

模式 0：单事件超时

该模式用于在一个预设的时间间隔后触发一个事件，例如中断。这个时间间隔可以长达 54.9ms，这时使用的起点计数值为 0。时钟 0、1、2、3 以及 5 都可以使用这个单事件超时模式。

在这个时钟中装入一个递减计数值。图 16-3 中所示的定时时序图使用的计数值为 4。输出线为低电平。当门线（gnte line）为高电平时，时钟计数器以 1.1932MHz 的频率递减。当时钟到达计数值 0 时，输出线变为高电平。计数时，如果门线变为了低电平，计数将停止，直到门线再次回到高电平。

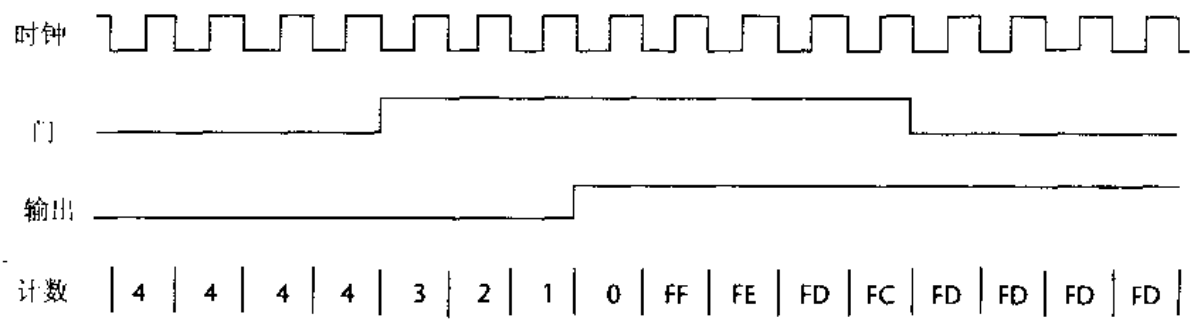


图 16-3　定时时序图，模式 0

模式 1：可重新触发的一次脉冲（ONE-SHOT）

该模式使用门线触发一次固定周期的输出。一旦输出周期完成，时钟就准备好在下次触发门线时生成另一个 ONE-SHORT。输出脉冲的最大持续时间是 54.9ms，这时使用起点计数值 0。计数器总是在检验计数器值是否为 0 之前递减。如果检验到值为零，则停止计数器。

通常只有时钟 2 可用于这种模式，因为只有时钟 2 有可设置的门线。对端口 61h，位 0 的输出控制时钟 2 的门线。EISA 系统的时钟 3 和 5 通常也可以使用这种模式。

在时钟中装入一个计数值。对于图 16-4 中的定时时序图，装入的值是 3。输出线的初始化状态是高电平。门线用来触发一个 ONE-SHOT。当门线变成高电平时，从下一个时钟周期开始，计数器开始递减，并将计数器的输出线设置为低电平。通常门线很快设置为低电平，以便在递减计数结束时准备下一个 ONE-SHOT。在计数值为 0 时，输出线回到高电平。时钟自动恢复为装入值，并准备好下一次触发来再次开始这个过程。

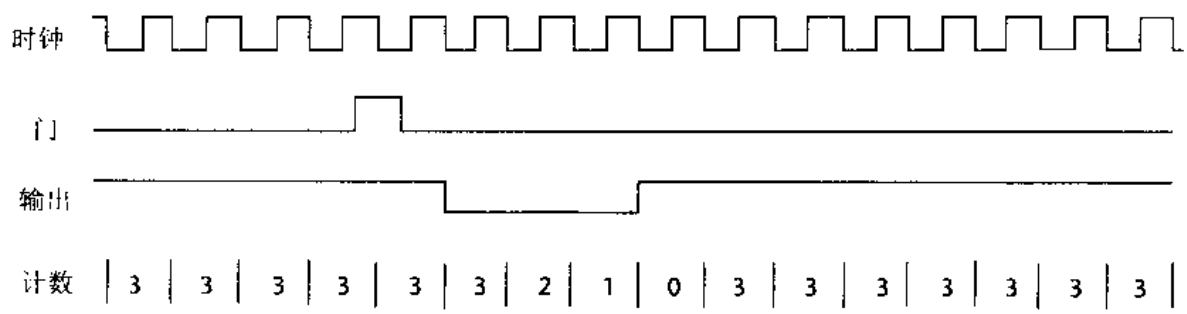


图 16-4 定时时序图，模式 1

模式 2：比率发生器

比率发生器模式用来生成一个周期性短输出脉冲。计数值为 0 时，输出周期最大，每 54.9ms 出现一次输出。时钟 0、1 以及 2 可用于比率发生器模式。EISA 系统的时钟 3 和 5 也可以使用这个模式。

在时钟中装入一个计数值。对于图 16-5 中的定时时序图，装入的值是 3。输出线的初始化状态是高电平。当递减计数为 1 时，输出线变成低电平，并维持一个计数值时间。然后输出线返回到高电平。自动重新装入计数值，并自动继续递减计数。如果门线变成了低电平（仅时钟 2），会暂时停止计数，直到门线返回高电平。如果在门线变成低电平时输出是低电平，那么输出立即变成高电平（没有显示）。不要装入计数值 1，因为可能出现意想不到的结果。

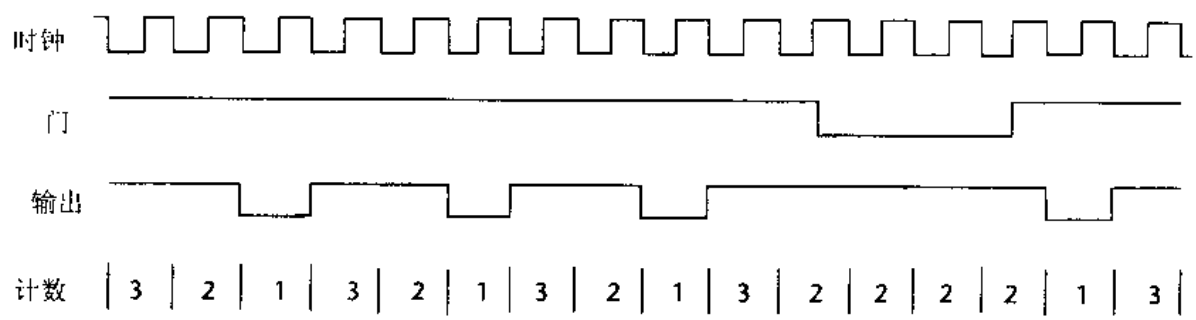


图 16-5 定时时序图，模式 2

模式 3：方波模式

方波模式用来生成一个周期性方波输出。通常时钟 0 使用这个模式来生成系统时钟中断。计数值设置为 0 时，获得最大的触发周期，每 54.9ms 触发一次 IRQ 0。这意味着中断 8 以每秒 18.2 次的速率获得触发。时钟 0、1 和 2 可以使用方波模式。EISA 系统的时钟 3 和 5 也可以使用这个模式。

操作因装入的值是基数还是偶数而有所不同。我们先看看偶数情况，装入偶数，图 16-6 中的值是 6，然后将这个值传输到计数器中，输出线的初始状态为高电平，每个时钟计数器递减两次。当计数到期时，输出线从前一个状态翻转。自动装入计数器，并继续该过程。

在另外一种情况下，装入奇数。输出线的初始状态是高电平。在下一个时钟，这个奇数值减一后传输到计数器。跟偶数情况相似，每个时钟递减计数两次。计数到期后一个时钟脉冲时输出变成低电平。自动装入这个奇数值减一后的值。每个时钟递减这个计数两次。计数到期时（无时钟脉冲延迟），输出返回到高电平。然后重复这个过程。对于奇数计数，时钟的输出线将在（N+1）/2 个计数内是高电平。输出线将在（N-1）/2 个计数内回到低电平。N 是装入的计数值。

如果门线变成了低电平（仅时钟 2），则暂时停止计数，直到门线回到高电平。如果在门线变成低电平时输出线位于高电平状态，则输出线被强制到低电平状态（没有显示）。不要装入值 1，因为那样会出现意想不到的结果。

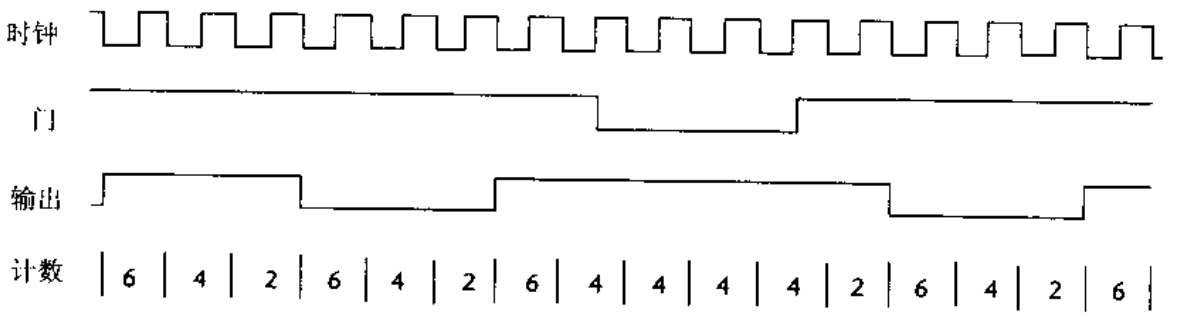


图 16-6　定时时序图，模式 3

模式 4：软件触发选通脉冲时钟门

这个模式用于在一个由软件启动的延迟后生成一个脉冲输出。这个延迟可以高达 54.9ms。时钟 0、1 和 2 可以使用软件触发选通脉冲模式。EISA 系统的时钟 3 和 5 也可以使用这种模式。

装入一个延迟值。对于图 16-7 中的例子，装入的计数值是 6。输出的初始化状态设置为高电平。这个装入的值被传输到计数器，计数器立即开始递减计数。当计数从 1 变到 0 时，输出变成低电平并维持一个周期。不会采取进一步的行动，直到软件装入了一个新的计数值。如果门线变成了低电平（仅时钟 2），会暂时停止计数，直到门线返回高电平。

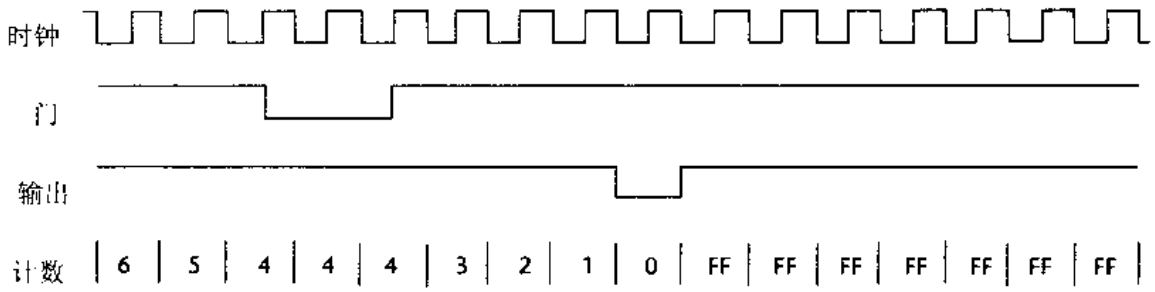


图 16-7 定时时序图，模式 4

模式 5：硬件可触发选通脉冲

这个模式用来在门线触发后输出一个脉冲。这个时间延迟可以高达 54.9ms。只有时钟 2 可以用于这种模式，因为只有时钟 2 有可设置的门线。对端口 61h，位 0 的输出可以控制这个门线。

装入延迟值。对于图 16-8 中的例子，装入的计数值是 5。输出的初始状态设置为高电平。门线的上升沿会触发将装入的值传输到计数器，然后计数器立即开始递减。在计数从 1 变成 0 时，输出变成低电平并维持一个周期。不会采取进一步的行动直到门的另一个上升沿触发了该计数器。

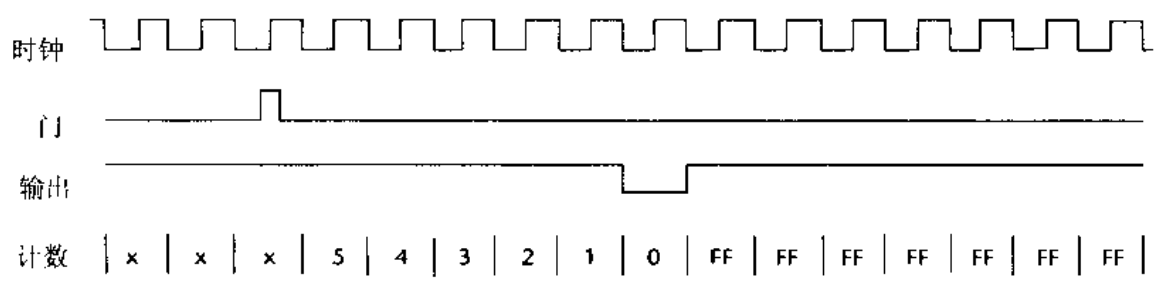


图 16-8 定时时序图，模式 5

通用操作——所有模式

表 16-1 归纳了每个模式的门线操作。时钟 0 和 1 的门线永远处于高电平状态。时钟 2 的门线由对端口 61h 位 0 的写控制。时钟 3 的门线与时钟 0 的输出相连（仅 MCA）。在 EISA 系统上，时钟 3 的门线永远处于高电平状态，而时钟 5 的门线与时钟 1 的刷新请求输出相连。

表 16-1 门操作

动作				
模式	触发器	低电平	上升沿	高电平
0	电平检测	停止计数		允许计数
1	上升沿		1）开始计数	
			2）下一个时钟	
			输出低电平	
2	电平检测	1）停止计数	开始计数	允许计数
	和上升沿	2）立即输出		
		高电平		
3	电平检测	1）停止计数	开始计数	允许计数
	和上升沿	2）立即输出		
4	电平检测	高电平		
5	上升沿	停止计数		允许计数
			开始计数	

对于不同的模式操作，所装入的计数值有不同的限制。表 16-2 显示了这些限制。下面显示的最大初始计数值是 0。在模式 2 和 3 下，在计数到期后才装入计数值。在所有其他的模式下，计数值会继续递减下去，即使出现了一个特殊的操作。这些非周期性的模式会造成计数环绕返回，在二进制下从 0 到 FFFF，在 BCD 操作下从 0 到 9999。

表 16-2 初始计数限制

计数的最大数					
模式	最小	最大*	16 位二进制	4 位 BCD	计数到期后的动作
0	1	0	65536	1000	环绕返回
1	1	0	65536	1000	环绕返回
2	2	0	65536	1000	重新装入
3	2	0	65536	1000	重新装入
4	1	0	65536	1000	环绕返回
5	1	0	65536	1000	环绕返回

*计数的最大数出现在零处，因为在检查新的计数器值是否为零之前递减计数器。

时钟0——系统定时

时钟0的输入频率固定为1.1932MHz，而不论CPU系统速度有多快。POST将时钟0设置为模式3操作，来输出一个方波信号，每秒18.207次。这个方波信号通过IRQ 0和中断控制器相连。每个周期内，这个时钟触发中断8一次。这一点用来保持系统时间，以及执行专门的定时服务，例如软盘马达开多长时间。此外，中断8调用中断1Ch，用户时钟滴答中断。

希望执行它们自己的低分辨率定时的程序可以挂起中断1Ch，以便每54.926ms触发一次（每秒18.207次）。时钟2通常用于高分辨率的定时。

参看警告部分，以了解某些系统如何对时钟0使用模式2，而不是模式3的IBM标准。

时钟1——DRAM刷新

与时钟0和2一样，时钟1的输入频率固定为1.1932MHz，而不论CPU系统速度有多快。在AT系统上，POST通常设置时钟1为模式2操作，来每15.09uS输出一次脉冲信号。装入一个值12h来实现这一点。其他的系统可能会使用其他模式和定时值，这取决于刷新设计。

时钟1的输出将会输入到主板上的DRAM刷新电路中。对这个时钟编程时使用不同的周期率或模式，可能会造成DRAM刷新操作停止，这样会丢失所有的DRAM内存内容。因此，我极力建议不要使用时钟1。

时钟2——一般用途和扬声器

一般的程序可以使用时钟2。它的输入频率固定为1.1932MHz，而不论CPU系统速度有多快。时钟2的门线可由端口61h的位0控制。从端口61h读取一个字节并检查位5，可以读取时钟2的输出。

此外，它的输出可以作为一个门输入到扬声器来生成音调及其他简单的效果。当端口61h的位1设置为1时，时钟2的输出就作为扬声器的门控逻辑。使用模式3，周期性方波时，装入一个16位计数值(=1,193,200Hz / 输出频率（Hz）)来设置频率。

例如，要生成一个2KHz的音调，可以在这个时钟中装入一个16位的计数值，这个值是1,193,200除以2,000，或597。

时钟3——看门狗（仅MCA）

看门狗时钟用来检测系统问题，例如软件的无止境的循环，并禁止中断。提高检查是

否定期设置系统时钟，看门狗时钟来检测是否出现系统问题。如果看门狗检测到 IRQ 0 的服务不够快（即：丢失了时钟滴答），时钟溢出会触发一个不可屏蔽中断（NMI）。

这个时钟仅操作在模式 0 下，并且只有一个 8 位的计数器。它的连接情况如图 16-9 所示。

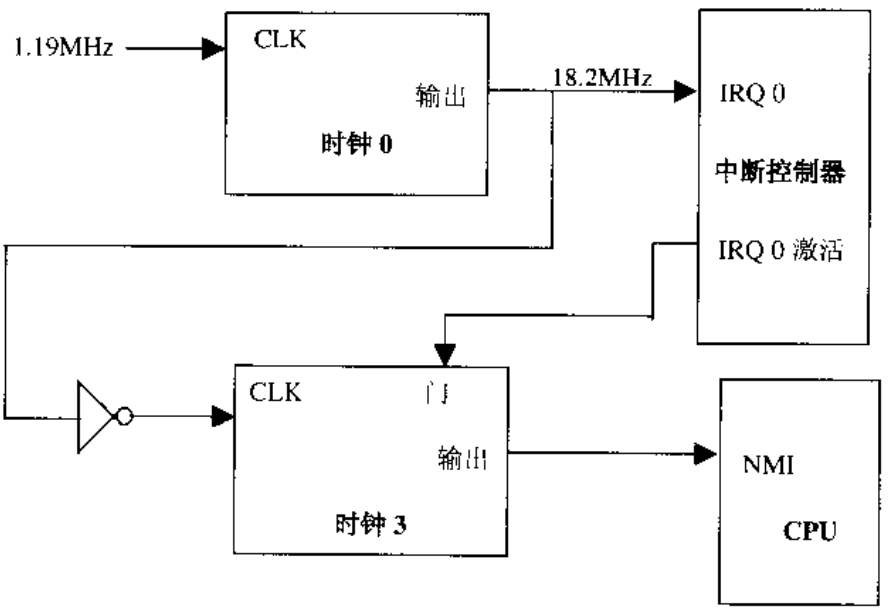


图 16-9　看门狗时钟的连接

在普通的操作中，每次触发 IRQ 0 会启动中断 8 时钟服务程序。在时钟服务程序完成时，会清除 IRQ 0 线。这条线也输入到时钟 3 的门上。由于在激活时钟 3 的门时时钟 0 没有出现时钟，所以时钟 3 的状态保持不变。

如果挂起了系统，则不会为 IRQ 0 提供服务，IRQ 0 激活线仍停留在高电平上。将允许来自时钟 0 输出的下一个时钟信号关闭时钟 3。一旦预设的计数到期，时钟 3 将输出高电平，并产生一个 NMI。然后 BIOS 的 NMI 服务处理程序可以采取适当的操作，例如重启系统。

读端口 92h，检查位 4，来检查时钟 3 的输出。

时钟 3——看门狗（仅 EISA）

看门狗时钟用来检测系统问题，例如软件的无止境的循环，并禁止中断。它的门逻辑是由硬件开启的，输入的时钟频率是 298.3MHz。它的输出通过由端口 70h 控制 NMI 门逻辑连接到 CPU 的 NMI 上。激活时，可以周期性更新这个时钟，以避免出现看门狗时钟超时。

先将这个时钟设置为模式 1 操作来开放看门狗时钟特性。装入一个初始值，并将端口 461h 处的扩展 NMI 控制寄存器的位 2 设置为开。必须周期性的装入一个新的值，以免出现超时。看门狗时钟每 3.352 μS 递减一次。

出现超时时，如果扩展 NMI 控制支持 NMI，则向 CPU 发送一个 NMI。如果是因为看门狗超时才出现 NMI，那么端口 461h 的位 7 会是 1。第 13 章（系统功能）中显示了端口

461h 的细节。

尽管这个仅用作看门狗时钟，它却是一个完整的 16 位时钟。这个时钟支持所有的模式和标准 8254 时钟操作。

时钟 4——未使用（仅 EISA）

在 EISA 系统上没有时钟 4，尽管有时钟 3 和 5。

时钟 5——CPU 速度控制（仅 EISA）

BIOS 将这个计数器保留用于 CPU 速度控制，主板设计人员可以将时钟 5 设计成其他用途，它不用作一般性用途。

时钟 5 是一个完全的 16 位时钟/计数器，可以操作在所有模式下。这个计数器的门由刷新时钟 1 的输出控制，它的输入时钟信号是 BCLK 线，由底板提供，通常是 8MHz。

用于 CPU 速度控制时，时钟 5 应设置为模式 1，可重触发 ONE-SHOT 操作。每次系统触发一个刷新请求时，ONE-SHOT 陷入并停止 CPU 一个 SHOT 的时间。这样放慢了 CPU 的速度，大小与初始计数值成正比。

典型的时钟设置和操作

1. 首先关闭时钟 2 的门。从端口 61h 的输入可以获取 61h 所有位的当前设置。位 0，时钟 2 的门值，应该设置为 0，并将新的字节写回到端口 61h（这样可以避免改变其他不相关的位）。
2. 接着，向端口 43h 输出一个命令字节来设置时钟 2 的命令模式。命令 0B6h 表示该计数器设置为模式 3：周期性方波，二进制计数模式，读/写端口 42h 时，先传输最低有效位字节，接着传输最高有效位字节。
3. 向时钟 2 中写入开始计数值。例如，有一个 16 位的计数器值是 1000h。首先写入最低有效位，向端口 42h 输出 00h。然后向端口 42h 输出最高有效位字节 10h 。
4. 开放时钟 2 的门。从端口 61h 的输入操作可以获取当前的设置。将位 0，时钟 2 的门值设置为 1，并将新的字节写回到端口 61h 中。
5. 时钟生成一个方波输出，频率为 291Hz。
6. 可以读取端口 61h 并检查位 5，来访问时钟 2 的输出。

此外，时钟 2 的输出可以作为主板扬声器的门控逻辑。读取 61h 的当前内容，并将位 1 设置 1。然后将这个字节写回到端口 61h，来开放扬声器。

典型用途

时钟 0、1、3、和 5 由系统使用，一般的程序不使用它们。时钟 2 供程序使用。这些用处包括：

- 扬声器音调的生成——时钟设置为模式 3，周期性方波输出，并开放扬声器的门控逻辑。
- 查询定时——在计数器中装入时间，并设置模式 4 或 5。开启时钟，则可以在软件中查询输出状态的改变。
- 事件的精确定时——选择模式 0，单事件超时。时钟中装入最大值，0。在事件开始时，时钟 2 的门切换到高电平。在事件结束时，这条门线回到低电平。将事件起点和终点之间的计数差乘以 .838uS，可以得到事件的持续时间。事件持续最大的时间是 54.9ms。
- 高中断率控制——这种特殊的情况使用时钟 0。例如，如果你想获得复杂的声音效果，比如语音，可以使用系统时钟来生成更高中断率。大约 10KHz 就可能从内置的扬声器中生成比较理想的语音和复杂的音乐效果。大多数游戏采用这种方法不用特殊的声卡就获取特殊的声音效果。参看代码例 16-3，了解如何实现这一点。

访　问

BIOS 直接控制时钟 0、1 和 3 的操作，供内部系统使用。没有 BIOS 中断服务程序或功能对某个时钟提供直接的用户控制。参看第 15 章，CMOS 内存和实时时钟，了解使用实时时钟的 BIOS 定时服务。

警　告

如果在计数器改变值时读取，那么读取的时钟计数值可能不精确。使用锁存的方法来确保读取精确的值。命令 0xh、4xh、8xh 分别控制时钟 0、1 和 2 的输出锁存（参看端口 43h）。

从 1994 年开始，许多，如果不是全部，使用时钟 0 的 AMI BIOS 引入了一个 bug。没有使用 IBM 定义的标准模式 3，AMI BIOS 对模式 2 进行编程。它生成的时钟滴答时间间隔不变，每 54.926ms 一次，可是内部计数器计数值的递减速率却只有模式 3 的一半！如果应用程序在访问时钟 0 时，没有考虑到这个问题，那么它将以这些系统一半的速度定时运行。

不幸的是，你无法读取 BIOS 设置的模式，所以必须执行一个“定时检查”来检测所使用的模式。对于高分辨率的定时和声音效果，读取时钟 0 当前计数的程序在使用模式 2 的系统上必须加以补偿。代码 16-2 显示了如何实现定时检查。

代码例 16-1 精确的事件定时

这个小型程序设置时钟来度量一个软件事件的时间。可以使用 TIMEVNT。如果事件持续的时间超过 54ms，就会检测到溢出。TIMEVNT 首先读取上次事件值，并显示事件的持续时间。然后 TIMEVNT 重启时钟，准备好下一个事件。误差精度为+/-838ns，即一个时钟的时钟脉冲周期。

使用下面的代码激活计数器，来开始事件定时：

```
in      al, 61h                 ; 读取当前内容
or      al, 1                   ; 将门位设置为开
out     61h, al                 ; 激活计数器
```

使用下面的代码禁止计数器，来停止事件定时：

```
in      al, 61h                 ; 读取当前内容
and     al, 0FEh                ; 将门位设置为关
out     61h, al                 ; 禁止计数器
```

程序 TESTCASE 中含有这个起始/停止事件定时代码，来度量 DOS 将一个字符输出到显示屏所用的时间。下面的 TIMEVEN 程序初始化时钟并度量一个事件的持续时间。这里没有显示 TESTCASE 代码以及 TIMEVENT 内调用的子程序的代码，但是附带的软盘中包括了它们的源代码。

如果你想在你的机器上试一试 TIMEVNT 程序，运行 TIMEVNT，然后运行 TESTCASE，接着再次运行 TIMEVNET。操作的结果显示和图 16-10 相似。

```
C:\> timevent
上个事件持续 54,142us    为下个事件重新设置计数器。

C:\>testcase
TESTCASE – 在此之前和之后使用 TIMEVENT 来度量 DOS 向显示屏
输出 "X" 所用的时间。

X

C:\> timevent
上个事件持续时间 = 54,142us    为下个事件重新设置计数器。
```

图 16-10 TIMEVENT 的结果

```
;--------------------------------------------------------------------------
;                                TIMEVENT
;--------------------------------------------------------------------------
;
;    本程序显示前面事件的周期，并重启
;    时钟来为下一个事件计时。
;
;    (c) 1994，1996 Frank van Gilluwe版权所有

include undocpc.inc

cseg    segment para public
        assume  cs:cseg, ds:cseg

        org     100h                    ; 汇编成COM文件


timevent proc   far

; first, check if timer has overflowed

start:
        in      al, 61h
        test    al, 20h                 ; 检查时钟2的输出状态
        jz      timeskp1                ; 若未发生溢出，则跳转
        mov     dx, offset ovrflow      ; 输出，溢出信息
        jmp     timeskp2

; read Timer 2 counter contents

timeskp1:
        mov     al, 80h                 ; 锁存输出命令
        out     43h, al                 ; 发送命令
        IODELAY
        in      al, 42h                 ; 获取计数器的LSB
        IODELAY
```

```
        mov     dl, al
        in      al, 42h             ; 获取计数器的MSB
        mov     dh, al              ; dx = 计数值
        mov     ax, 0FFFFh          ; 开始值
        sub     ax, dx              ; ax = 周期计数值

; 将周期计数值ax乘以838得到微秒数

        mov     bx, 838             ; 假定时钟处于模式3
        mul     bx                  ; dx:ax = ax * bx
        call    make_decimal        ; 将dx:ax转化成十进制
        mov     dx, offset event    ; 周期信息
timeskp2:
        mov     ah, 9
        int     21h                 ; 显示信息

; 重新设定时钟2准备为下一个周期计时

        mov     al, 0B0h            ; 时钟2命令，模式0
        out     43h, al             ; 发送命令
        IODELAY
        mov     al, 0FFh            ; 计数器值FFFF
        out     42h, al             ; 发送计数器的LSB
        IODELAY
        out     42h, al             ; 发送计数器的MSB

        mov     ah, 4Ch
        int     21h                 ; DOS 中止
timevent endp


event           db      CR, LF
                db      'Last event duration = '
out_value       db      '          us    '
                db      '   Counter reset for next event.'
                db      CR, LF, '$'
```

```
ovrflow         db      CR, LF
                db      'Last event duration exceeded 54,142 us'
                db      '    Counter reset for next event.'
                db      CR, LF, '$'
```

代码例 16-2　时钟 0 模式

大多数系统的 BIOS 将时钟 0 设置为模式 3。某些系统会错误的设置为模式 2，这将影响基于当前时钟 0 计数的定时。模式不影响系统时钟的持续时间，后者每 54ms 触发一次中断 8。参看警告，以更多的了解有关模式的问题。这个程序找出时钟 0 使用模式 3 还是模式 2。要运行这个测试，在 DOS 提示符下键入 TIMERTST。图 16-11 显示了在一个系统上运行 TIMERTST 的结果。

```
C:\ > timertst
时钟 0 的模式测试                                    v1.00  (c)  1996  FVG
---------------------------------------------------------------------------
当前的模式:    2
```

图 16-11　TIMERTST 的运行结果

```
;                       TIMERTST
; 查出时钟 0 是运行在普通的模式 3 下，还是运行在半速模式 2 下。
; 模式并不影响时钟触发中断 8 的时钟节拍的频度。
;       (c) 1996 Frank van Gilluwe 版权所有
cseg    segment para pub lic
        assume cs:cseg, ds:cseg

        org     100h                    ; 汇编成 COM 文件

timertst proc   far

start:
        call    calibrate               ; 确定时钟位于模式 2 还是 3
        jnc     output_msg              ; 如果是普通模式 3，则跳转
        dec     mode_value              ; 检测到是时钟模式 2
output_msg:
```

```
        mov     dx, offset timer_mode
        mov     ah, 9
        int     21h                     ; 显示信息
        mov     ah, 4Ch
        int     21h                     ; DOS 中止
timertst endp


timer_mode      db      CR, LF, CR, LF
                db      ' TIMER 0 MODE TEST                '
                db      '                  V1.00 (C) 1996 FVG'
                db      CR, LF
                db      ' ------------------------------------------'
                db      ' ------------------------------------------'
                db      CR, LF
                db      ' Current mode:                    '
mode_value      db      ' 3  '
                db      CR, LF, '$'

; ---------------------------------------------------------------------------
; 校正
; 检查时钟 0 是位于普通的模式 3 下还是位于模式 2 下。模式 2 下的操作
; 速度只有模式 3 下的一半。两种模式下系统中断触发的速率相同。
;       调用:           无
;       返回:         进位=0      普通模式 3
                        1       模式 2 操作
;       用到的寄存器:  ax,bx,cx,dx
;       子程序调用:    无

calibrate proc  near
        push    es
        mov     cx, 20h                 ; cx=缓存屏蔽时钟
        xor     dx, dx                  ; 节拍 (tick) 的最大数目 (200000h)
        mov     ax, 40h                 ; 使用段址 40h 的 BIOS 数据
        mov     es, ax
        mov     ax, es:[6Ch]            ; 获取时钟节拍的低部分
```

```
        ; 跳入循环等待时钟节拍（54ms）出现

cal_nochange:
        cmp     ax, es:[6Ch]
        jne     cal_change                  ; 等待改变
        sub     dx, 1
        sub     dx, 0
        jcxz    cal_done                    ; 跳过处理，缓存介入
        jmp     cal_nochange

        ; 刚出现时钟节拍，保存当前值

cal_change:
        mov     bx, es:[6Ch]                ; 保存新值
        mov     cx, -1                      ; 寄存器的值减 1

cal_loop:
        mov     al, 0
        out     43h, al                     ; 锁存时钟信息
        IODELAY
        in      al, 40h                     ; 获取低字节，时钟 0
        xchg    ah, al
        cmp     cx, -1                      ; 首次读时钟？
        jne     cal_notfirst                ; 若非，则跳转
        cmp     ax, 2
        jne     cal_loop                    ; 等待，直到重启时钟

cal_notfirst:
        cmp     ax, cx                      ; 看新值是否比最小值还要小
        ja      cal_rolled                  ; 如果时钟翻转，则跳转
        mov     cx, ax                      ; 保存新的最小值
        jmp     cl_loop

        ; 时钟只翻转了一次——如果是模式 3，那么还
        ; 不会出现中断 8（需要时钟翻转两次）。但如果
```

```
        ; 是模式 2，第一次翻转时就会出现中断 8。

cal_rolled:
        IODELAY                         ; 如果位于模式 3，需要少量的时
        IODELAY                         ; 间来处理中断 8
        mov     ax, es:[6Ch]            ; 获取时钟节拍的低部分
        cmp     ax, bx                  ; 时钟节拍发生了改变?
        je      cal_done                ; 若非，则跳转，必是模式 3
        stc                             ; 时钟处于模式 2
        jmp     cal_done2
cal_done:
        clc
cal_done2:
        pop     es
        ret
calibrate endp

cseg    ends

        end     start
```

代码例 16-3 快速中断处理程序

某些应用程序要求相当快中断率，远快于标准的系统时钟每秒 18.2 次。只能使用 CMOS 时钟来生成一个每 976us 的周期性中断，或者是其他固定的周期（参看 CMOS 寄存器 0Ah）。此外，PC 和 XT 不包括 CMOS 时钟—插入式卡可能并服从 AT 标准。一种解决方案加速了系统时钟中断。

为了以更高的速率获取控制，必须采取几步操作。首先，假定需要 50 倍于普通情况的速率，或者每 1,099us 触发一次。程序必须挂起中断 8 的服务程序。与一般的挂起操作不同，这个加速 TSR 只能每 50 次才有一次将控制传给原始的服务程序。挂起的中断 8 每 1,099uS 将被调用一次，某些短操作，比如向扬声器发送一位，就可以被执行。

记住，随着中断率越高，问题将会越大。取决于 CPU、CPU 速度、系统要求的时间、以及你的中断 8 处理程序，系统很容易在运行其他事件时用完时间，甚至可能丢失时钟滴答。游戏和其他加速时钟 0 的程序通常会检查系统，来看看它可以多大程度的提高速率。如果系统不支持期望的最大速率，就会选择一个较低的速率。对于那些需要加速时钟 0 来产生音响效果的产品来说，如果系统不能处理期望的最高速率，那么会降低音响的品质。

这里有一个重要的警告值得注意。一次只有一个程序可以安全的加速系统时钟。没有确定的方法来共享加速系统时钟。为了检测其他的应用程序是否使用了加速系统时钟，有必要同时挂起中断 8 和 1Ch。在每次使用中断 8 时代码必须检查是否还出现了一个中断 ICh。如果中断 8 出现的频率高于中断 1Ch，你将会受到其他程序的惩罚。

在程序结束时，必须恢复原始的中断 8h 服务程序。你还应该防止非正常情况下的退出，例如按下 Control-Break，没有恢复中断就退出。

下面这个程序例在屏幕上部写入两个不同颜色的字符。快速切换时，这两个字符会重叠在一起，在在一个绿色文本字符的上部会显示出一个白色的加粗标记。尽管这个演示程序并不实用，但是该代码显示了可以改变中断率，来远远超过了普通的速度。

为了将这个程序用于它处，只要简单的将 DEMO_TWO_CHAR 替换成你自己的程序就可以了。NEWRATE 指定了对系统标准时钟 0 速率的倍乘因子。一个 NEWRATE 值 10 意味着你的中断 8 处理程序每秒将被调用 182 次。在文件 TMRFAST.ASM 中可以找到。完整的源代码。虽然 TMRFAST 是一个 TSR，但是没有必要将这个加速技术作为一个 TSR 来使用。作为非 TSR 应用时，必须在退回到 DOS 之前，恢复原始的中断 8 向量。

```
;--------------------------------------------------------------------------
;                              TIMER FAST
;--------------------------------------------------------------------------
;
;     本程序显示了如何通过加速系统时钟0
;     来获得比18次每秒更快的中断控制。
;     在这个例子中，中断发生的频度是普通情况下的50位，
;     即910次每秒
;
;     (c) 1994，1996 Frank van Gilluwe版权所有

include undocpc.inc

NEWRATE equ     50                      ; 提高中断速率的倍数
                                        ;

cseg    segment para public
        assume  cs:cseg, ds:cseg, ss:tsrstk

countdown       dw      NEWRATE         ; 递减时钟
old_int_8       dd      0               ; 旧时钟保存在此处
```

```
; ------------------------------------------------------------------------------
; 下面是处理中断8的驻留处理程序

int_8_hook      proc      far
          pushf

; 此处插入快速率代码
;

          call      demo_two_char              ; 演示程序


; 快速代码调用结束后，原来的中断
;    8处理程序仍恢复到原来的18.2次每秒
;

          dec       cs:[countdown]
          jnz       int_8_exit
          mov       cs:[countdown], NEWRATE    ; 重新装入计数器
          popf
          jmp       cs:[old_int_8]             ; 处理原来的中断8

int_8_exit:
          cli                                  ; 禁止中断
          push      ax
          mov       al, 20h                    ; 中断结束命令
          out       20h, al                    ; 向控制器发命令
          pop       ax
          popf
          iret
int_8_hook      endp


; ------------------------------------------------------------------------------
; 本演示程序快速地在 "/" 字符
```

```
;       和字母之前切换来产生重叠的效果。
;       任何两个字符都可以利用这个程序
;       进行重叠。注意有三种不同的颜色
;       出现在单个文本格中（黑、白和绿）

democount       dw      0
demochar        db      'A'             ; 要显示的字符
SLASH_ATTRB     equ     0Fh             ; 属性，亮白
CHAR_ATTRB      equ     0Ah             ; 属性，绿色

demo_two_char   proc    near
        push    ax
        push    di
        push    es

        mov     ax, 40h
        mov     es, ax                  ; 指向BIOS内存区
        mov     ax, 0B800h              ; 假定彩色文本视频区域
        cmp     byte ptr es:[49h], 7    ; 单色视频模式?
        jne     demo1                   ; 若非，则跳转
        mov     ax, 0B000h              ; 使用单色段
demo1:
        mov     es, ax                  ; 装入屏幕段
        mov     di, 79 * 2              ; 指向首行末尾
        mov     al, '/'                 ; 重叠字符
        mov     ah, SLASH_ATTRB         ; 属性
        test    cs:[democount], 1       ; 字符间切换
        jz      demo2                   ;  取决于计数
        mov     al, cs:[demochar]
        mov     ah, CHAR_ATTRB          ; 字符属性
demo2:
        cld
        stosw                           ; 将AX保存到屏幕的es:[di]处
        inc     cs:[democount]
        mov     ax, cs:[democount]
        cmp     ax, 182*(NEWRATE/10) ; 1秒期满?
```

```
        jb      demo3                   ; 若非，则跳转
        mov     cs:[democount], 0       ; 重新设置演示计数
        inc     cs:[demochar]           ; 使用另外一个字符
demo3:
        pop     es
        pop     di
        pop     ax
        ret
demo_two_char   endp


;=============================================================
; 非驻留安装部分的起点

tmrfast proc    far

start:
        push    cs
        pop     ds

; 获取当前的中断8向量并保存

        mov     al, 8
        mov     ah, 35h
        int     21h                     ; 获取当前的中断8指针
        mov     word ptr old_int_8, bx
        mov     word ptr old_int_8+2, es

; 安装新的中断8程序

        mov     dx, offset int_8_hook
        mov     al, 8
        mov     ah, 25h
        int     21h                     ; 安装我们的中断
```

```
        OUTMSG   installmsg          ; 显示安装信息

; 重设时钟0到交换频率

        cli                          ; 禁止中断
        mov      al, 36h             ; 16位信息
        out      43h, al             ; 模式3，二进制操作
        IODELAY
        mov      ax, 65535/NEWRATE; 计数器值
        out      40h, al             ; 在计数器0中装入计数LSB
        IODELAY
        mov      al, ah              ; 获取最大有效位字节
        out      40h, al             ; 在计数器0中装入计数MSB
        sti                          ; 开放中断

; 确定剩下驻留部分的大小（段落）
;   并成为TSR

        mov      dx, (offset start - offset countdown) SHR 4
        add      dx, 11h             ; PSP大小加上1个段落
        mov      ax, 3100h           ; 退回到DOS成为TSR
        int      21h
tmrfast endp


installmsg db    CR, LF
           db    'TIMER FAST Installed - Demonstrates fast '
           db    'interrupt 8 handler.', CR, LF
           db    '   Makes the top right corner of screen '
           db    'overlap two colors and characters.', CR, LF, '$'

cseg       ends
```

端口归纳

下面这个端口列表用于时钟控制。

端口	类型	功能	平台
40h	I/O	时钟 0——系统滴答	所有
41h	I/O	时钟 1——DRAM 刷新	所有，除了 MCA
42h	I/O	时钟 2——通用用途	所有
43h	输出	时钟 0~2——模式控制	所有
44h	I/O	时钟 3——看门狗	MCA
47h	输出	时钟 3——模式控制	MCA
48h	I/O	时钟 3——看门狗	EISA
4Ah	I/O	时钟 5——CPU 速度控制	EISA
4Bh	I/O	时钟 3 & 5——模式控制	EISA

端口细节

端口	类型	描述	平台
40h	I/O	时钟 0——系统滴答	所有

时钟 0 用作系统时钟。通常将它设置为模式 3，周期性方波操作。在计数值中装入 0，来生成每秒 18.2 次的脉冲。参看有关系统定时的小节，了解时钟 0 的更多细节，以及系统如何使用很少被兼容的模式 2。

输入（位 0~7）——读计数寄存器 0

这个写入到端口 43h 的命令字节控制如何从这个端口读取信息。

1xh = 只读最低有效位字节
2xh = 只读最高有效位字节
3xh = 先读最低有效位字节，然后读最高有效位字节

输处（位 0~7）——写计数寄存器 0

这个写入到端口 43h 的命令字节控制如何向这个端口写入信息。

1xh = 写入最低有效位字节，并将最高有效位字节设置为 0
2xh = 写入最高有效位字节，并将最低有效位字节设置为 0
3xh = 先写入最低有效位字节，然后写入最高有效位字节

为了帮助说明这一点，下面的代码段显示了如何将时钟 0 改变为每秒 1820.7 次。必须在时钟 0 中装入计数值 655（1193200/1820.7 = 655）来实现这一点。将使用模式 3，因为它提供了一个连续的方波输出来触发中断。参看代码例 16-3 来了解实现这种方法的完整持续。

```
mov     al,   36h          ; 16 位命令
out     43h,  al           ; 模式 3 二进制操作
IODELAY
```

```
mov     ax,     655             ; 计数值
out     40h,    al              ; 装入时钟 0 计数，LSB
IODELAY
Mov     al,     ah
mov     40h,    al              ; 装入时钟 0 计数，LSB
```

端口	类型	描述	平台
41h	I/O	时钟 1——DRAM 刷新	所有

时钟 1 用作系统刷新。通常将它设置为模式 2，比率发生器操作。在计数值中装入 12h，来每 15uS 生成一次的脉冲。参看有关时钟 1，“DRAM 刷新”的小节，了解更多细节。

输入（位 0~7）——读计数寄存器 1

这个写入到端口 43h 的命令字节控制如何从这个端口读取信息。

5xh = 只读最低有效位字节

6xh = 只读最高有效位字节

7xh = 先读最低有效位字节，然后读最高有效位字节

输处（位 0~7）——写计数寄存器 1

这个写入到端口 43h 的命令字节控制如何向这个端口写入信息。

5xh = 写入最低有效位字节，并将最高有效位字节设置为 0

6xh = 写入最高有效位字节，并将最低有效位字节设置为 0

7xh = 先写入最低有效位字节，然后写入最高有效位字节

端口	类型	描述	平台
42h	I/O	时钟 2 通用用途	所有

时钟 1 用于扬声器操作和通用用途。参看有关时钟 2，“一般用途和扬声器”的小节，了解更多细节。

输入（位 0~7）——读计数寄存器 2

这个写入到端口 43h 的命令字节控制如何从这个端口读取信息。

9xh = 只读最低有效位字节

Axh = 只读最高有效位字节

Bxh = 先读最低有效位字节，然后读最高有效位字节

输出（位 0~7）——写计数寄存器 2

这个写入到端口 43h 的命令字节控制如何向这个端口写入信息。

9xh = 写入最低有效位字节，并将最高有效位字节设置为 0

Axh = 写入最高有效位字节，并将最低有效位字节设置为 0
Bxh = 先写入最低有效位字节，然后写入最高有效位字节

端口	类型	描述	平台
43h	输出	时钟 0~2 一模式控制	所有

使用这个端口来设置模式，以控制时钟 0、1 和 2。参看有关模式操作小节

输出（位 0~7）

位			
7 w = x	命令		
6 w = x			
5 w = x			
4 w = x			
3 w = x	模式选择		
2 w = x	位 3	位 2	位 1
1 w = x	0	0	0 = 模式 0
	0	0	1 = 模式 1
	0	1	0 = 模式 2
	0	1	1 = 模式 3
	1	0	0 = 模式 4
	1	0	1 = 模式 5
	1	1	0 = 模式 2
	1	1	1 = 模式 3
0 w = 0	二进制计数模式（16 位）		
1	BCD 计数模式（四位二——十进制编码数字）		

命令归纳

下面命令中的 *x* 指定时钟模式和二进制/BCD 位。在命令细节中解释了低四位数据。

命令号	端口 43h 命令
0xh	计数器锁存时钟 0
1xh	计数器 0 的 LSB 模式
2xh	计数器 0 的 MSB 模式
3xh	时钟 0 的 16 位模式
4xh	计数器锁存时钟 1
5xh	计数器 1 的 LSB 模式
6xh	计数器 1 的 MSB 模式
7xh	时钟 1 的 16 位模式

8xh	计数器锁存时钟 2
9xh	计数器 2 的 LSB 模式
Axh	计数器 2 的 MSB 模式
Bxh	时钟 1 的 16 位模式
Dxh	通用计数器锁存
Exh	时钟的锁存状态

端口细节

命令	描述	端口
0xh	计数器锁存时钟 0	43h

将时钟 0 计数器的值传输到这个输出锁存寄存器。保持锁存的值直到端口 40h 读取了这个值，或者时钟 0 中装入了一个新的计数值。忽略后续的计数器锁存值，直到读取了这个值。忽略这个命令的低四位。

使用这个命令和从该计数器的一般读取有所不同。如果没有锁存，从端口 40h 读取的值就是时钟 0 的即时值。

命令	描述	端口
1xh	时钟 0 的 LSB 模式	43h

设置时钟 0 为只读和写最低有效字节。写时钟 0 计数时，最高有效字节自动设置为零。其余的各位控制时钟 0 的操作模式。

位			
7 w = 0	命令		
6 w = 0			
5 w = 0			
4 w = 1			
3 w = x	模式选择		
2 w = x	位 3	位 2	位 1
1 w = x	0	0	0 = 模式 0
	0	0	1 = 模式 1（不可用）
	0	1	0 = 模式 2
	0	1	1 = 模式 3
	1	0	0 = 模式 4
	1	0	1 = 模式 5（不可用）
0 w = 0	二进制计数模式（16 位）		
1	BCD 计数模式（四位二——十进制编码数字）		

命令	描述	端口
2xh	时钟0的MSB模式	43h

设置时钟0为只读和写最高有效字节。写时钟0计数时，最低有效字节自动设置为零。其余的各位控制时钟0的操作模式。

位				
7 w = 0	命令			
6 w = 0				
5 w = 1				
4 w = 0				
3 w = x	模式选择			
2 w = x	位3	位2	位1	
1 w = x	0	0	0 = 模式 0	
	0	0	1 = 模式 1（不可用）	
	0	1	0 = 模式 2	
	0	1	1 = 模式 3	
	1	0	0 = 模式 4	
	1	0	1 = 模式 5（不可用）	
0 w = 0	二进制计数模式（16位）			
1	BCD计数模式（四位二——十进制编码数字）			

命令	描述	端口
3xh	时钟0的16位模式	43h

设置时钟0为读和写完整的16位计数器。读和写时，先发送最低有效字节，然后发送最高有效字节。其余的各位控制时钟0的操作模式。

位				
7 w = 0	命令			
6 w = 0				
5 w = 1				
4 w = 1				
3 w = x	模式选择			
2 w = x	位3	位2	位1	
1 w = x	0	0	0 = 模式 0	
	0	0	1 = 模式 1（不可用）	
	0	1	0 = 模式 2	
	0	1	1 = 模式 3	
	1	0	0 = 模式 4	
	1	0	1 = 模式 5（不可用）	
0 w = 0	二进制计数模式（16位）			
1	BCD计数模式（四位二——十进制编码数字）			

命令	描述	端口
4xh	计数器锁存时钟 1	43h

将时钟 1 计数器的值传输到这个输出锁存寄存器。保持锁存的值直到端口 41h 读取了这个值，或者时钟 1 中装入了一个新的计数值。忽略后续的计数器锁存值，直到读取了这个值。忽略这个命令的低四位。

使用这个命令和从该计数器的一般读取有所不同。如果没有锁存，从端口 41h 读取的值就是时钟 1 的即时值。

命令	描述	端口
5xh	时钟 1 为 LSB 模式	43h

设置时钟 1 为只读和写最低有效字节。写时钟 1 计数时，最高有效字节自动设置为零。其余的各位控制时钟 1 的操作模式。

位		命令			
7 w = 0					
6 w = 1					
5 w = 0					
4 w = 1					
3 w = x	模式选择				
2 w = x		位 3	位 2	位 1	
1 w = x		0	0	0 = 模式 0	
		0	0	1 = 模式 1（不可用）	
		0	1	0 = 模式 2	
		0	1	1 = 模式 3	
		1	0	0 = 模式 4	
		1	0	1 = 模式 5（不可用）	
0 w = 0	二进制计数模式（16 位）				
1	BCD 计数模式（四位二——十进制编码数				

命令	描述	端口
6xh	时钟 1 为 MSB 模式	43h

设置时钟 1 为只读和写最高有效字节。写时钟 1 计数时，最低有效字节自动设置为零。其余的各位控制时钟 1 的操作模式。

位	命令
7 w = 0	
6 w = 1	
5 w = 1	
4 w = 0	

3 w = x	模式选择		
2 w = x	位 3	位 2	位 1
1 w = x	0	0	0 = 模式 0
	0	0	1 = 模式 1（不可用）
	0	1	0 = 模式 2
	0	1	1 = 模式 3
	1	0	0 = 模式 4
	1	0	1 = 模式 5（不可用）
0 w = 0	二进制计数模式（16 位）		
1	BCD 计数模式（四位二——十进制编码数字）		

命令	描述	端口
7Ah	时钟 1 的 16 位模式	43h

设置时钟 1 为读和写完整的 16 位计数器。读和写时，先发送最低有效字节，然后发送最高有效字节。其余的各位控制时钟 1 的操作模式。

位	7 w = 0	命令		
	6 w = 1			
	5 w = 1			
	4 w = 1			
	3 w = x	模式选择		
	2 w = x	位 3	位 2	位 1
	1 w = x	0	0	0 = 模式 0
		0	0	1 = 模式 1（不可用）
		0	1	0 = 模式 2
		0	1	1 = 模式 3
		1	0	0 = 模式 4
		1	0	1 = 模式 5（不可用）
	0 w = 0	二进制计数模式（16 位）		
	1	BCD 计数模式（四位二——十进制编码数字）		

命令	描述	端口
8Ah	计数器锁存时钟 2	43h

将时钟 2 计数器的值传输到这个输出锁存寄存器。保持锁存的值直到端口 42h 读取了这个值，或者时钟 2 中装入了一个新的计数值。忽略后续的计数器锁存值，直到读取了这个值。忽略这个命令的低四位。

使用这个命令和从该计数器的一般读取有所不同。如果没有锁存，从端口 42h 读取的

值就是时钟 2 的即时值。

命令	描述	端口
9xh	时钟 0 的 LSB 模式	43h

设置时钟 2 为只读和写最低有效字节。写时钟 2 计数时，最高有效字节自动设置为零。其余的各位控制时钟 2 的操作模式。

位				
	7 w = 1	命令		
	6 w = 0			
	5 w = 0			
	4 w = 1			
	3 w = x	模式选择		
	2 w = x	位 3	位 2	位 1
	1 w = x	0	0	0 = 模式 0
		0	0	1 = 模式 1（不可用）
		0	1	0 = 模式 2
		0	1	1 = 模式 3
		1	0	0 = 模式 4
		1	0	1 = 模式 5（不可用）
	0 w = 0	二进制计数模式（16 位）		
	1	BCD 计数模式（四位二——十进制编码数字）		

命令	描述	端口
Axh	时钟 2 的 MSB 模式	43h

设置时钟 2 为只读和写最高有效字节。写时钟 2 计数时，最低有效字节自动设置为零。其余的各位控制时钟 2 的操作模式。

位				
	7 w = 1	命令		
	6 w = 0			
	5 w = 1			
	4 w = 0			
	3 w = x	模式选择		
	2 w = x	位 3	位 2	位 1
	1 w = x	0	0	0 = 模式 0
		0	0	1 = 模式 1（不可用）
		0	1	0 = 模式 2
		0	1	1 = 模式 3
		1	0	0 = 模式 4

	1	0	1 = 模式 5（不可用）
0 w = 0	二进制计数模式（16 位）		
1	BCD 计数模式（四位二——十进制编码数字）		

命令	描述	端口
Bxh	时钟 2 的 16 位模式	43h

设置时钟 2 为读和写完整的 16 位计数器。读和写时，先发送最低有效字节，然后发送最高有效字节。其余的各位控制时钟 2 的操作模式。

位				
7 w = 1	命令			
6 w = 0				
5 w = 1				
4 w = 1				
3 w = x	模式选择			
2 w = x	位 3	位 2	位 1	
1 w = x	0	0	0 = 模式 0	
	0	0	1 = 模式 1（不可用）	
	0	1	0 = 模式 2	
	0	1	1 = 模式 3	
	1	0	0 = 模式 4	
	1	0	1 = 模式 5（不可用）	
0 w = 0	二进制计数模式（16 位）			
1	BCD 计数模式（四位二——十进制编码数字）			

命令	描述	端口
Dxh	通道计数器锁存	43h

该命令用于向输出锁存器传输计数器内容。对时钟 0、1 和 2 的传输可以独立进行，也可以组合进行。与命令 0xh、4xh 和 8xh 相似，锁存器的值会保持到各自的时钟端口 40h~42h 读取了该值，或者新的值装入到了计数器中。对于任何已经锁存但是还未读取的时钟，将忽略所有后续的计数器锁存命令。

位		
7 w = 1	命令	
6 w = 1		
5 w = 0		
4 w = 1		
3 w = 1	选择计数器 2	
2 w = 1	选择计数器 1	
1 w = 1	选择计数器 0	
0 w = 0	未使用	

命令	描述	端口
Eah	锁存时钟的状态	43h

该命令用于向输出锁存器锁存选中时钟的状态。一旦锁存了一个时钟的状态，可以分别从端口 40h~42h 读取该状态。对于任何已经锁存但是还未读取的时钟，将忽略所有后续的状态锁存命令。

位		
	7 w = 1 6 w = 1 5 w = 1 4 w = 0	命令
	3 w = 1	选择计数器 2
	2 w = 1	选择计数器 1
	1 w = 1	选择计数器 0
	0 w = 0	未使用

由这个命令锁存状态时，可以从各自的端口 40h~42h 读取该状态。该状态字节在其 0~5 位中保存了模式信息及其他信息，如下所示：

位				
	7 r = x	时钟输出引脚的状态		
	6 r = 0	无效计数，计数器中装入了一个新的计数		
	1	计数还未装入计数器中		
	5 r = x	读/写模式		
	4 r = x	位 5	位 4	
		0	1 = 仅读/写最高有效字节	
		1	0 = 仅读/写最低有效字节	
		1	1 = 先读/写最低有效字节，然后读/写最高有效字节	
	3 w = x 2 w = x 1 w = x	模式选择		
		位 3	位 2	位 1
		0	0	0 = 模式 0
		0	0	1 = 模式 1
		0	1	0 = 模式 2
		0	1	1 = 模式 3
		1	0	0 = 模式 4
		1	0	1 = 模式 5
		1	1	0 = 模式 2
		1	1	1 = 模式 3
	0 w = 0	二进制计数模式（16 位）		
	1	BCD 计数模式（四位二——十进制编码数字）		

端口	类型	描述		平台
44h	I/O	时钟 3	看门狗	MCA

时钟 3 用作看门狗时钟。对时钟 3 的操作和模式有一些限制。参看有关时钟 3“看门狗（MCA）”的小节，了解更多细节。

输入（位 0~7）——读计数寄存器 3（仅 8 位计数器）
输出（位 0~7）——写计数寄存器 3（仅 8 位计数器）

端口	类型	描述		平台
47h	输出	时钟 3	模式控制	MCA

使用这个端口来设置时钟 3 的模式。

输出（位 0~7）

位		命令
	7 w = x	
	6 w = x	
	5 w = x	
	4 w = x	
	3 w = 0	未使用
	2 w = 0	未使用
	1 w = 0	未使用
	0 w = 0	未使用

命令归纳

命令号	端口 47h 命令
00h	计数器锁存时钟 3
10h	时钟 3 的模式

命令	描述	端口
00h	计数器锁存时钟 3	47h

将时钟 3 计数器的值传输到这个输出锁存寄存器。保持锁存的值直到端口 44h 读取了这个值，或者时钟 3 中装入了一个新的计数值。忽略后续的计数器锁存值，直到读取了这个值。忽略这个命令的低四位，通常将它们设置为零。

使用这个命令和从该计数器的一般性读取有所不同。如果没有锁存，从端口 44h 读取的值就是时钟 4 的即时值。

命令	描述	端口
10h	计数器锁存时钟 3	47h

设置时钟 3 来读和写计数器字节。没有该命令的其他位。时钟 3 的硬件特性决定了它固定为一个 8 位的计数器，采用二进制计数、模式 0 操作。

位		命令
	7 w = 0	
	6 w = 0	
	5 w = 0	
	4 w = 1	
	3 w = 0	未使用
	2 w = 0	未使用
	1 w = 0	未使用
	0 w = 0	未使用

端口	类型	描述	平台
48h	I/O	时钟 3——看门狗	EISA

时钟 3 用作看门狗时钟。对时钟 3 的操作和用的模式有一些限制。时钟 3 的输入时钟为 298.3KHz，门线总是处于开状态。参看有关时钟 3，看门狗（EISA），的小节，了解更多细节。

输入（位 0~7）——读计数寄存器 3

这个写入到端口 4Bh 的命令字节控制如何从这个端口读取信息。

1xh = 只读最低有效位字节

2xh = 只读最高有效位字节

3xh = 先读最低有效位字节，然后读最高有效位字节

输出（位 0~7）——写计数寄存器 3

这个写入到端口 4Bh 的命令字节控制如何向这个端口写入信息。

1xh = 写入最低有效位字节，并将最高有效位字节设置为 0

2xh = 写入最高有效位字节，并将最低有效位字节设置为 0

3xh = 先写入最低有效位字节，然后写入最高有效位字节

端口	类型	描述	平台
4Ah	I/O	时钟 5——CPU 速度控制	EISA

时钟 2 用做可任选的 CPU 速度控制或由主板设计者确定的其他功能。参看有关时钟 5

的小节。

输入（位 0~7）——读计数寄存器 5

这个写入到端口 4Bh 的命令字节控制如何从这个端口读取信息。

9xh = 只读最低有效位字节

Axh = 只读最高有效位字节

Bxh = 先读最低有效位字节，然后读最高有效位字节

输出（位 0~7）——写计数寄存器 5

这个写入到端口 4Bh 的命令字节控制如何向这个端口写入信息。

9xh = 写入最低有效位字节，并将最高有效位字节设置为 0

Axh = 写入最高有效位字节，并将最低有效位字节设置为 0

Bxh = 先写入最低有效位字节，然后写入最高有效位字节

端口	类型	描述	平台
4Bh	输出	时钟 3 & 5 的模式控制	EISA

使用这个端口来设置控制时钟 3 和 5 的模式。参看有关操作模式的小节。

输出（位 0~7）

位				
7 w = x	命令			
6 w = x				
5 w = x				
4 w = x				
3 w = x	模式选择			
2 w = x		位 3	位 2	位 1
1 w = x		0	0	0 = 模式 0
		0	0	1 = 模式 1
		0	1	0 = 模式 2
		0	1	1 = 模式 3
		1	0	0 = 模式 4
		1	0	1 = 模式 5
		1	1	0 = 模式 2
		1	1	1 = 模式 3
0 w = 0	二进制计数模式（16 位）			
1	BCD 计数模式（四位 BCD 十进制编码数字）			

命令归纳

命令号	端口 4Bh 命令
0xh	计数器锁存时钟 3
1xh	计数器 3 的 LSB 模式
2xh	计数器 3 的 MSB 模式
3xh	时钟 3 的 16 位模式
8xh	计数器锁存时钟 5
9xh	计数器 5 的 LSB 模式
Axh	计数器 5 的 MSB 模式
Bxh	时钟 5 的 16 位模式
Dxh	通用计数器锁存
Exh	时钟的锁存状态

端口细节

命令	描述	端口
0xh	计数器锁存时钟 3	4Bh

将时钟 3 计数器的值传输到这个输出锁存寄存器。保持锁存的值直到端口 48h 读取了这个值，或者时钟 3 中装入了一个新的计数值。忽略后续的计数器锁存值，直到读取了这个值。忽略这个命令的低四位。

使用这个命令和从该计数器的一般读取有所不同。如果没有锁存，从端口 48h 读取的值就是时钟 3 的即时值。

命令	描述	端口
1xh	时钟 3 的 LSB 模式	4Bh

设置时钟 3 为只读和写最低有效字节。写时钟 3 计数时，最高有效字节自动设置为零。其余的各位控制时钟 3 的操作模式。

位				
	7 w = 0	命令		
	6 w = 0			
	5 w = 0			
	4 w = 1			
	3 w = x	模式选择		
	2 w = x	位 3	位 2	位 1
	1 w = x	0	0	0 = 模式 0

	0	0	1 = 模式 1
	0	1	0 = 模式 2
	0	1	1 = 模式 3
	1	0	0 = 模式 4
	1	0	1 = 模式 5（不可用）

0 w = 0　二进制计数模式（16 位）

1　BCD 计数模式（四位二——十进制编码数字）

命令	描述	端口
2xh	时钟 3 的 MSB 模式	4Bh

设置时钟 3 为只读和写最高有效字节。写时钟 3 计数时，最低有效字节自动设置为零。其余的各位控制时钟 3 的操作模式。

位　7 w = 0　命令

6 w = 0

5 w = 1

4 w = 0

3 w = x　模式选择

2 w = x

1 w = x

位 3	位 2	位 1
0	0	0 = 模式 0
0	0	1 = 模式 1
0	1	0 = 模式 2
0	1	1 = 模式 3
1	0	0 = 模式 4
1	0	1 = 模式 5（不可用）

0 w = 0　二进制计数模式（16 位）

1　BCD 计数模式（四位二——十进制编码数字）

命令	描述	端口
3xh	时钟 3 的 16 位模式	4Bh

设置时钟 3 为读和写完整的 16 位计数器。读和写时，先发送最低有效字节，然后发送最高有效字节。其余的各位控制时钟 3 的操作模式。

位　7 w = 0　命令

6 w = 0

5 w = 1

4 w = 0

3 w = x	模式选择		
2 w = x	位 3	位 2	位 1
1 w = x	0	0	0 = 模式 0
	0	0	1 = 模式 1
	0	1	0 = 模式 2
	0	1	1 = 模式 3
	1	0	0 = 模式 4
	1	0	1 = 模式 5（不可用）
0 w = 0	二进制计数模式（16 位）		
1	BCD 计数模式（四位二——十进制编码数字）		

命令	描述	端口
8xh	计数器锁存时钟 5	4Bh

将时钟 5 计数器的值传输到这个输出锁存寄存器。保持锁存的值直到端口 4Ah 读取了这个值，或者时钟 5 中装入了一个新的计数值。忽略后续的计数器锁存值，直到读取了这个值。忽略这个命令的低四位。

使用这个命令和从该计数器的一般读取有所不同。如果没有锁存，从端口 4Ah 读取的值就是时钟 5 的即时值。

命令	描述	端口
9xh	时钟 5 的 LSB 模式	4Bh

设置时钟 5 为只读和写最低有效字节。写时钟 5 计数时，最高有效字节自动设置为零。其余的各位控制时钟 5 的操作模式。

位	7 w = 0	命令		
	6 w = 0			
	5 w = 1			
	4 w = 0			
	3 w = x	模式选择		
	2 w = x	位 3	位 2	位 1
	1 w = x	0	0	0 = 模式 0
		0	0	1 = 模式 1
		0	1	0 = 模式 2
		0	1	1 = 模式 3
		1	0	0 = 模式 4

	1	0	1 = 模式 5（不可用）
0 w = 0	二进制计数模式（16 位）		
1	BCD 计数模式（四位二——十进制编码数字）		

命令	描述	端口
Ach	时钟 5 的 MSB 模式	4Bh

设置时钟 5 为只读和写最高有效字节。写时钟 5 计数时，最低有效字节自动设置为零。其余的各位控制时钟 5 的操作模式。

位			
7 w = 1 6 w = 0 5 w = 1 4 w = 1	命令		
3 w = x 2 w = x 1 w = x	模式选择 位 3	位 2	位 1
	0	0	0 = 模式 0
	0	0	1 = 模式 1
	0	1	0 = 模式 2
	0	1	1 = 模式 3
	1	0	0 = 模式 4
	1	0	1 = 模式 5（不可用）
0 w = 0	二进制计数模式（16 位）		
1	BCD 计数模式（四位二——十进制编码数字）		

命令	描述	端口
Bxh	时钟 5 的 16 位模式	4Bh

设置时钟 5 为读和写完整的 16 位计数器。读和写时，先发送最低有效字节，然后发送最高有效字节。其余的各位控制时钟 5 的操作模式。

位			
7 w = 1 6 w = 0 5 w = 1 4 w = 1	命令		
3 w = x 2 w = x 1 w = x	模式选择 位 3	位 2	位 1
	0	0	0 = 模式 0
	0	0	1 = 模式 1

0	1	0 = 模式 2
0	1	1 = 模式 3
1	0	0 = 模式 4
1	0	1 = 模式 5（不可用）

0 w = 0　二进制计数模式（16 位）

1　BCD 计数模式（四位二——十进制编码数字）

命令	描述	端口
Dxh	通用计数器锁存	4Bh

该命令用于向输出锁存器传输计数器内容。对时钟 3 和 5 的传输可以独立进行，也可以组合进行。与命令 0xh 和 8xh 相似，锁存器的值会保持到各自的时钟端口 48h 或 4Ah 读取了该值，或者新的值装入到了计数器中。对于任何已经锁存但是还未读取的时钟，将忽略所有后续的计数器锁存命令。

位		
	7 w = 1	命令
	6 w = 1	
	5 w = 0	
	4 w = 1	
	3 w = 1	选择计数器 5
	2 w = 0	未使用
	1 w = 1	选择计数器 3
	0 w = 0	未使用

命令	描述	端口
Exh	锁存时钟的状态	4Bh

该命令用于向输出锁存器锁存选中时钟的状态。一旦锁存了一个时钟的状态，可以分别从端口 40h~42h 读取该状态。对于任何已经锁存但是还未读取的时钟，将忽略所有后续的状态锁存命令。

位		
	7 w = 1	命令
	6 w = 1	
	5 w = 1	
	4 w = 0	
	3 w = 1	选择计数器 5
	2 w = 0	未使用
	1 w = 1	选择计数器 3
	0 w = 0	未使用

由这个命令锁存状态时，可以从各自的端口 48h~4Ah 读取该状态。该状态字节在其 0~5 位中保存了模式信息，及其它信息，如下所示：

位		
	7 r = x	时钟输出引脚的状态
	6 r = 0	无效计数，计数器中装入了一个新的计数
	1	计数还未装入计数器中
	5 r = x	读/写模式
	4 r = x	位 5　位 4
		0　1 = 仅读/写最高有效字节
		1　0 = 仅读/写最低有效字节
		1　1 = 先读/写最低有效字节，然后读/写最高有效字节
	3 w = x	模式选择
	2 w = x	位 3　位 2　位 1
	1 w = x	0　0　0 = 模式 0
		0　0　1 = 模式 1
		0　1　0 = 模式 2
		0　1　1 = 模式 3
		1　0　0 = 模式 4
		1　0　1 = 模式 5
		1　1　0 = 模式 2
		1　1　1 = 模式 3
	0 w = 0	二进制计数模式（16 位）
	1	BCD 计数模式（四位二—— 十进制编码数字）

第 17 章

中断控制和 NMI

中断控制管理计算机的硬件中断过程。我将显示中断控制器的基本操作、不可屏蔽中断以及对中断控制器编程的基础知识，以适应特殊的高速需要。同时还解释了 PC、AT、MCA 和 EISA 系统之间中断控制的差别。

程序人员会遇到的一个问题时，有时需要重定向硬件中断，让它们指向另一个向量地址。这样做可以提高系统运行的速度或者在特殊情况下避免冲突。我也阐释了如何来实现这一点，并举出了几个实际的例子。

简　介

设计计算机系统时，要求硬件设备，例如键盘或串口，可以即时的请求服务。使用中断控制可以实现这一点。它避免让 CPU 将一些时间浪费在检查或查询硬件上，这些硬件可能在无法预见的时候需要服务。重要的是，一个硬件设备会中断 CPU 的当前进程，并运行一个服务处理程序。一旦硬件设备服务结束，CPU 将继续它以前的操作。

在 PC/XT 系统上，中断控制器可以处理八个中断，在 AT+的机器上可以处理十五个中断。许多硬件中断线实现特定的功能，例如时钟、键盘以及磁盘驱动器操作。主板的底板上提供其中一些硬件线，所以适配卡也可以使用硬件中断系统。PCI 总线对每个 PCI 插槽提供了中断端口的可编程连接。它支持“即插即用”BIOS 自动配置 PCI 卡。

通过一个或两个 8259 中断控制器芯片处理中断控制。这些中断控制器芯片管理多个中断请求和 CPU 的中断表达。

8259 有许多内部寄存器来控制中断操作。这些寄存器控制支持哪个硬件中断、优先级、硬件中断的类型、以及 CPU 应处理哪个特殊的中断软件程序（从 265 个选一个）。

典型的中断过程

图 17-1 显示了启动一个中断时事件发生的顺序过程。假定适配卡将其跳线跳到了 IRQ5，并且不会激活任何其他请求。此外，中断控制器的屏蔽寄存器允许中断 5。（该屏蔽寄存器用来允许或锁住中断。）

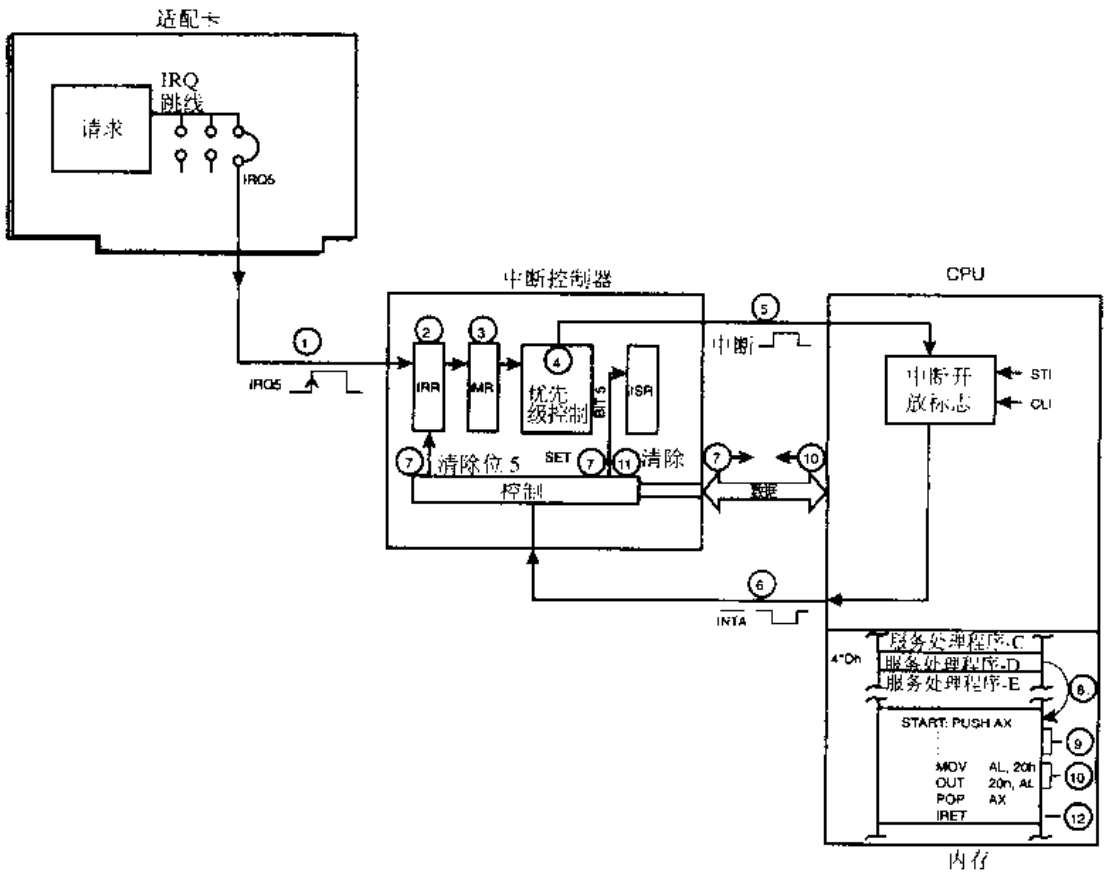


图 17-1　中断的执行顺序

一个单独的中断的完整处理过程如下所述：

1. 适配卡将硬件中断 IRQ5 的底板线从低电平拉到高电平，来请求中断服务。
2. IRQ5 线连接到中断控制器。边沿触发请求被记录在控制器的中断请求寄存器中。这时，中断请求寄存器的位 5 被设置为一。
3. 中断控制器首先检查中断屏蔽寄存器，看是否允许中断 5。假定已经清除了 IRQ5 的屏蔽位，支持这个请求。这时，控制器继续进行下一步检查。
4. 控制器检查是否有更高优先级的中断已被激活或者正在执行。如果有，不会有进一步的动作，直到所有的高优先级的中断服务结束。
5. 如果没有更高优先级的中断处于激活状态，控制器会通过与 CPU 相连的 INT 硬件线通知 CPU。
6. 如果开放了这个硬件中断，那么 CPU 会向中断控制器返回一个确认信号。

7. CPU 的确认信号造成控制器向 CPU 发送中断字节 Dh。在这种情况下，IRQ 和中断 D 相关联。此外，激活控制器的在服务寄存器（ISR）的 IRQ5 位（开）。最后，清除控制器中断请求锁存器（IRR）的 IRQ5 位，表示正在执行中断服务。
8. CPU 中断当前的进程，在堆栈中保存当前指令的地址和标志，然后调用位于向量 Dh 处中断服务处理程序。中断 Dh 的远地址保存在段址为 0，偏移量为 4*Dh（即：0000：0034）处的内存底部。
9. 被调用的中断服务处理程序执行所有必要的操作。处理程序必须执行的一个重要的操作是，删除由该适配器做出的中断请求。通常在中断服务处理程序的操作，例如向适配器读或写一个字节，完成时，执行这项删除操作。这时适配器的 IRQ5 线变成低电平。
10. 在服务处理程序结束时，处理程序向端口 20h 发送值 20h，来向中断控制器发送一个中断结束（EOI）信号。
11. 来自 CPU 的 EIO 命令会清除 IRQ5 在服务位。这为中断控制器接受另外一个来自 IRQ5 的中断或更低优先级的中断扫清了道路。
12. 服务处理程序触发一条 IRET 指令，于是处理器恢复标志位并返回前面被中断的进程。

边沿/电平控制

一旦激活了一个 IRQ 线就生成一个硬件中断。每个中断控制器能够以两种方式触发这个事件。使用端口 20h 或 A0h 处的初始化命令字 1 来进行选择。该选择通常基于系统的硬件设计，不应改变。

边沿触发 这是检测中断请求的第一种方法，它使用 IRQ 线的上升沿来检测中断请求，可称为边沿触发。所有 PC/XT/AT 类型的机器使用这种方法，在 ESIA 机器上这时缺省的方法。

在 PC、XT 和 AT 系统上，一个边沿触发只可以供一个设备使用。如果两个设备试图使用相同的 IRQ 发送不同的信号，那么高电平和低电平信号会发生冲突。这时 IRQ 线处于一个未确定状态，可能不会触发该中断。

电平触发 这是检测中断请求的第二种方法，它使用 IRQ 线的状态来检测中断请求。在 IRQ 设置为高电平时，就会发出一个中断信号。如果在中断服务结束和清除了中断之后，IRQ 仍处于高电平，这时就会触发另一个中断。这种方法的优点是，它允许多个硬件设备共享同一个 IRQ 线。MCA 就是这样设计的。EISA 系统可以选择为电平触发模式。PCI 插槽的中断也是电平触发的。

设计系统时，使多个适配器可以将同一个中断请求线拉到高电平而不发生冲突，就可以实现对一个硬件中断的共享。使用活动的拉电平设计，可以实现这一点。这种设计也支

持多个适配器同时触发相同的中断请求，这样做也是安全的。此外，每个适配器必须提供一个中断挂起位，通常软件使用特定的 I/O 端口来读取该位。

适配器的中断服务程序必须检查中断是否由与它相关的适配器引起。可以从适配器读取挂起位来做出判断。如果不是该适配器触发这个中断，那么服务处理程序必须忽略该中断，并将控制传递给中断钩（hooked）链上的下一个中断。这样，就可以让使用相同中断的其他服务处理程序处理该请求。

中断共享将增加响应的时间，这个时间称为中断休眠期。对于那些不需要立即处理的设备，例如使用低波特率的串口设备来说，这一点是可以接受的。如果一个设备需要使中断休眠期最小时，应避免使用中断共享。

NMI——不可屏蔽中断

CPU 有一条独立的硬件不可屏蔽中断（NMI）线，它不受硬件控制器的控制。激活 NMI 时，处理器向量指向中断 2 的处理程序。它主要由 BIOS 用来陷入内存奇偶错误和 I/O 通道奇偶错误。在端口 70h 中使用一个标志位可以屏蔽掉 NMI。

这种会触发一个 NMI 的类型的错误，比如一个内存奇偶错误，可以独立的开放或禁止。参看第 13 章，系统功能，中的端口 60h 和 61h。

由于 NMI 通常被共享，所以出现 NMI 时，应检查所有可能的起因。一旦标识出并处理了该起因，应清除 NMI。执行处理程序的 IRET 时，允许新的 NMI。如果挂起了其他的 NMI，那么会立即出现下一个 NMI。由于硬件设计的问题，在一个有效的 NMI 之后，可能会得到一个虚幻的 NMI。如果没有找到 NMI 起因，那么 NMI 处理程序应正常的退出。

开放 NMI 时，应检查系统的标志来确定中断起因。表 17-1 列出了 AT+系统上最常见的功能。

表 17-1　NMI 状态位

端　　口	位	功　　能
61h	7	1 = 出现了 RAM 奇偶错误
61h	6	1 = 出现了 I/O 通道奇偶错误
461h	7	1 = 出现看门狗超时（仅 EISA）
461h	6	1 = 出现总线超时（仅 EISA）
461h	5	1 = I/O 端口状态（仅 EISA）

浮点协处理器和 NMI

数学协处理器在出现错误时会生成一个中断。在所有的系统上，使用数学协处理器的

程序必须挂起中断 2（NMI）来捕获可能的数学协处理器错误条件。由于数学协处理器和其他的功能，例如奇偶错误，共享，所以被挂起的程序必须检查是什么引起了这个中断。如果数学协处理器没有触发这个中断，则必须调用以前的中断 2 处理程序。

在 PC 和 XT 系统上，错误中断线（通过软件可控的门）和 CPU 的 NMI 输入相连。在 AT 以及所有后来的系统上，数学协处理器的错误线和 IRQ13（中断 75h）相连。为了保持和为 PC 及 XT 系统设计的软件兼容，BIOS 为中断 75h 安装了一个服务处理程序。出现一个数学错误时，BIOS 处理程序中断 75h 会删除该中断请求，并对中断控制器发一条中断结束（EOI）命令。这时，BIOS 程序触发一个中断 2 来模拟 PC 和 XT 系统上的 NMI 操作。处理中断 75h 的 BIOS 代码段与下面的代码段类似：

```
int_75h_entry:
        push    ax
        move    al,     0
        out     0F0h,   al              ; 清除数学 IRQ
        mov     al,     20h
        out     0A0h,   al              ; 对控制器 2 发一条 EOI 命令
        out     20h,    al              ; 对控制器 1 发一条 EOI 命令
        pop     ax
        int     2                       ; 触发 NMI，兼容性
        iret                            ; 结束！
```

如果你仔细看一下一本 80286 或后来 CPU 的用户手册，它在 16（10h）处描述了数学协处理器中断。这一点和我们刚刚讨论的内容相矛盾，并且和视频服务中断 10h 相重叠！80286 及所有后来的 CPU 在处理器内部有一个单独的数学错误中断。如果使用了这个特殊的错误中断输入，CPU 会触发中断 10h。由于它会造成冲突，所以 AT+系统不使用这种方法。

MCA 系统的不同之处

与 AT 系统的 IRQ 边沿触发不同，所有的 MCA IRQ 线都是电平触发的。这意味着在中断服务程序完成之前，硬件必须删除它的中断请求，否则会再次出现这个中断。相比于边沿触发，电平触发 IRQ 有两个优点。它对虚假噪声的敏感性更低，同时，它支持多个设备共享一个 IRQ。

在 MCA 系统上，不可屏蔽中断还有另外一个触发起因。看门狗时钟的输出（时钟 3）也连接到了 NMI 上。如果出现看门狗超时，就会触发 NMI。参看第 16 章，系统时钟“中的系统时钟 3”了解更多细节。

EISA 系统的不同之处

EISA 系统能够控制每个 IRQ 线的触发方式。端口 4D0h 和 4D1h 支持 IRQ 的触发方式可以是边沿触发，也可以是电平触发。IRQ0、1、2、8 和 13 的内部触器发已经由主板设计者做出了固定选择，永远也不要改变。

在 EISA 系统上，不可屏蔽中断还有另外三个触发起因。包括总线超时、看门狗时钟和 I/O 端口状态。如果开放了 NMI，软件向端口 462h 写入任何值也会触发 NMI。

PCI 系统的不同之处

来自每个 PCI 卡插槽的所有中断都使用电平触发。由于大多数的 PCI 系统仍带有独立的 AT 总线插槽，AT 总线中断必须保持为边沿触发。每个 PCI 插槽可以访问四个中断。通过一个可编程的中断发送程序，这些四个中断组可以被发送到任何一个可用的 IRQ。这个发送程序是 PCI 芯片组的一部分，因芯片组供应商和具体实现的不同，该发送程序的功能和操作有所不同。即插即用 BIOS 负责解决冲突和对 PCI 卡分配 IRQ。因为是电平触发，多个 PCI 卡在必要时可以共享一个中断。

典型使用

硬件中断请求（IRQ）可以分成两类。第一类 IRQ 包括一些专用功能，例如处理时钟滴答、键盘服务等等。第二类 IRQ 用于可任选的适配卡。大多数卡支持从可用的 IRQ 线中选择一条，以避免和其他适配卡冲突。大多数情况下，使用适配卡上的专用开关或跳线对 IRQ 进行手工选择。MCA、EISA、PCI 以及其他一些系统可以通过安装软件选择 IRQ。

系统 BIOS 对中断控制器实现初始化和编程。大多数情况下，唯一一条对中断控制器的非 BIOS 访问是发送命令来清除当前的中断和检查正在执行的中断位。少数情况下，软件也可以对中断控制器进行编程。

表 17-2 和表 17-3 归纳了中断请求的使用者和每个相关的中断。系统上中断请求线直接和主板上的硬件相连。适配卡通过底插槽使用其他的中断请求。仅显示了常见的适配器，但是其他许多适配卡也使用 IRQ 线。

表 17-2 早期 PC 的中断请求使用者

中断请求	线连到	中断控制器	中断向量	IRQ 源
IRQ0	系统	1	8	系统时钟 0 的输出
IRQ1	系统	1	9	键盘
IRQ2	底板	1	A	并行口 2
IRQ3	底板	1	B	串口 2
IRQ4	底板	1	C	串口 1
IRQ5	底板	1	D	硬盘
IRQ6	底板	1	E	软盘控制器
IRQ7	底板	1	F	并行口 1

任何适配器插槽都可以使用 IRQ2~IRQ7

表 17-3 AT+的中断请求使用者

中断请求	线连到	中断控制器	中断向量	IRQ 源
IRQ0	系统	1	8	系统时钟 0 的输出
IRQ1	系统	1	9	键盘
IRQ2	系统	1	无	用于控制器 2 级联
IRQ3	底板	1	B	串口 2 & 4，网卡
IRQ4	底板	1	C	串口 1 & 3
IRQ5	底板	1	D	并行口 2，网卡
IRQ6	底板	1	E	软盘控制器
IRQ7	底板	1	F	并行口 1
IRQ8	系统	2	70	CMOS 实时时钟
IRQ9	底板	2	A/71	EGA/VGA 垂直扫描选项
IRQ10	底板-16	2	72	
IRQ11	底板-16	2	73	
IRQ12	底板-16	2	74	主板鼠标端口
IRQ13	系统	2	75	数学协处理器错误，DMA
IRQ14	底板-16	2	76	硬盘控制器（主盘）
IRQ15	底板-16	2	77	硬盘控制器（从盘）

任何适配器插槽都可以使用 IRQ2~IRQ7。只有 AT+系统上在一个 16 位适配器插槽里的 16 位卡才可以使用 TRQ 10、11、12、14 和 15。PCI 卡提供了对 IRQ 的可配置访问

由于 IRQ2 用来串联控制器 2 和控制器 1，所以不可以使用它。由于设计的巧妙，IRQ2 看起来和可用的 IRQ 一样。目前 IRQ9 连接的底板线和以前 PC/XT 系统上 IRQ2 相同。为了保持和早期的 PC 完全兼容，如果一个适配器使用硬件线 IRQ2/9（一条线），那么就要触发中断 71h。BIOS 中断 71h 处理程序调用中断 Ah 处理程序，后者正是早期 PC 上的 IRQ2 的处理程序。这样，在 PC/XT 和 AT 类型的机器上，一个适配卡使用 IRQ2 看起来就没有什么区别，没有必要修改软件。

中断数据区

中断向量保存在内存底部 1K 处，在这里保存了 256 个指针。此外，大多数系统的 BIOS 使用一个字节来记录未使用的软件和硬件中断事件。表 17-4 简单的归纳了这些地址。在第 7 章“中断向量表”中显示了中断向量的完整列表。

表 17-4　BIOS 数据区

地　址	大　小	名	功　能
0:0	双字	中断向量 0	除法错误
0:4	双字	中断向量 1	调试
0:8	双字	中断向量 2	不可屏蔽中断
	.		
	.		
0:3FCh	双字	中断向量 FF	未使用的向量
40：6Bh			如果出现了硬件中断，而又没有相关的中断服务程序，则在这里保存中断号。如果出现软件中断，而又没有相关的中断处理程序，则在这个字节中保存值 FFh。

警　告

噪声信号　尽管不应出现一个噪声信号，不良的总线设计或者适配卡设计都可能引起噪声信号出现在某条 IRQ 线上。噪声信号会触发一个不合法的中断。如果噪声信号出现 IRQ0~IRQ7 上，会调用中断 F。如果噪声信号出现 IRQ8~IRQ15 上，会调用中断 77h。中断 F 和 77h 的服务处理程序应读取在线服务寄存器来确认是否出现了一个真实的中断。不合法中断的在线服务位 7 是清除的，有效中断的位 7 是设置的。参看端口 20h 的命令 3 来为中断 F 读取在服务寄存器。对于中断 FFh，使用端口 A0h 的命令 3 来读取第二个中断控制器上的在线服务寄存器。

中断优先级 通常最好不要改变中断优先级。很难知道某些其他的硬件设备是否使用了缺省的中断优先级。如果不同的程序需要改变优先级，记住没有提供优先级管理。最后一个改变优先级的程序不会通知其他可能已经改变了优先级的程序。

使用高波特率串口的程序经常改变中断优先级，因为如果串口服务不够快，则将会丢失字节。这时键盘和系统时钟通常位于一个比串口更低的优先级上。

COM2，IRQ3 据报道，在使用一个 DOS 扩展器，例如 PharLap，或者推理系统时，COM2 的 IRQ3（中断 Bh）会引发软件问题。这一点极有可能，因为中断 0Bh 也是保护模式下的段不存在异常。对于 IRQ3 的使用者来说，建议在认定发生了真实的中断之前，检查一下在服务位。下面的代码段确保在这种情况下普通的非共享的 IRQ3 能够正确操作。

```
int_Bh_entry:
        push    ax
        cli                             ; 禁止中断
        mov     al,     0Bh             ; 命令读取 ISR
        out     20h,    al
        IODELAY
        in      al,     20h             ; 获取在服务寄存器
        sti                             ; 开放中断
        test    al,     8               ; IRQ3 正在服务?
        jz      int_Bh_exit             ; 如果不是，跳走

        用户的代码放在这里

int_Bh_exit:
        pop     ax
        iret
```

CPU 重叠和 IRQ5 对于中断 5~10h，中断使用存在重叠。Intel 特意指出，0~1Fh 的所有中断为未来的 CPU 内部使用而保留。然而 IBM 在 PC 设计中并未理睬这一点，并继续将这些中断用作其他用途。例如，PC 将中断 5 定义为屏幕打印功能，它与 80286 的越界中断相重叠。80386 以及后来的处理器定义了更多的中断，和 PC 的 BIOS 中断相重叠。预设 IRQ0~7 使用中断向量 8~Fh，这些向量以被保留用于许多处理器的错误条件。

奇怪的是，这些中断重叠并没有造成许多问题。大多数的处理器错误中断都被局限于保护模式或 V86 模式下。这意味着只有那些编写保护模式软件的开发人员才需要考虑重叠。有一个例是中断 D。80386 及后来的处理器把它用于一般保护性（GP）错误。一般保护性错误可以出现在保护模式和实模式下。由于保护性错误是很严重的错误，通常在实模

式下它会被忽略。错误的类型包括在偏移量 FFFFh 处访问一个数据字，或者一条指令的长度超过 15 个字节。如果任何保护模式下的指令用在 V86 模式下，也会出现 GP 错误。

如果需要高中断率或非常快的响应，则不要在虚（V86）模式系统下使用 IRQ5（中断 D）。这一点可能很难做到，因为可用的 IRQ 已经很少。在带有 386 或后来的 CPU 的机器上运行 V386 模式时，中断 D 用于 GP 处理。由处理器生成这个中断来处理各种处理器错误，并将它们传递给保护模式管理程序进行处理。带有任何 386 内存管理程序时，系统可操作在 V86 模式下。

因为 V86 模式下的处理器就是这样设计的，保护模式的管理程序在用户的中断服务程序之前获得控制。对于中断 D，保护模式的管理程序必须确定是否出现了错误，并在找不到错误时将控制传递给原始的中断服务程序。不幸的是，确定 GP 错误的起因需要大量的代码来确定是否出现了错误，以及采取任何必要的操作。在慢速系统上，当将控制传递回原始的中断服务程序时，它会造成明显的响应延迟。

大多数的浮点数学库在没有 FPU 时会模拟一个浮点处理器。对于任何浮点指令，都会发出一个中断来获取适当的模拟程序。在 V86 模式下这也会增加延迟，给处理每个中断加入了一些额外的开销。

所有其他的中断在 V86 模式下时，都会面临一定的延迟问题。处理器首先将控制传递给保护模式的管理程序，然后后者必须将控制传回中断服务程序。由于中断 D 以外的其他中断需要保护模式的管理程序执行很少的操作，如果有的话，所以该效果很小。Pentium 的处理器似乎提供了一种新的方法来避免让每个中断进入保护模式。尽管它无助于解决与中断 D 有关的这个问题，但是它有助于确保其他许多中断的响应速度最大。

代码例 17-1 IRQ 重定向

通过将 IRQ5 和处理器的一般保护性错误中断 D 分离开来，IRQ_REDIRECT 提高了与 IRQ5 相连的硬件设备的执行速度。在运行了 IQR_REDIRECT 之后，如果硬件设备触发了一个 IRQ，那么中断控制器会使用中断 50h~57h，而不是中断 8~F。这样，保护模式的管理程序在每次出现 IRQ5 时就不用执行额外的处理，因为 IRQ5 不再使用中断 D。这些程序的代码段在软件包中的 IRQRECIR.ASM 中。

```
;--------------------------------------------------------------------------------
;     IRQ_REDIRECT
;          将IRQ0到IRQ7重新定向到另外一组中断向量上。
;          当系统处于V86模式而IRQ5用于高速中断速率时，
;          重定向可能会提高系统的运行速度。
;          参看警告部分以更多的了解有关IRQ5
;          和中断0Dh的冲突问题。
```

```
;
;       某些软件如OS/2、Deskview、Double DOS和
;       IBM3270模拟器通常会执行一些类似的任务，
;       在应用程序之前获得这些功能。
;       当多任务软件必须和应用软件
;       共享硬件时，这一点非常有用。
;       不要让这些程序也利用这一点。
;
;       注意:   1) 在改变向量之前，没有
;                  检查是否使用了中断50h到57h
;               2) 在IRQ-redirect运行之后挂起
;                  中断8到0Fh的任何程序或TSR将永远
;                  不会获得对该中断的访问权限！
;                  参看本代码后的注释来获得另外一种实现方法。
;
;       Regs used:      ax, cx, di, si, ds, es

IRQ_redirect    proc    near
        cli                             ; 禁止中断

                                        ; 对中断控制器1进行编程
                                        ; ---------------------
        in      al, 21h                 ; 获取当前的中断屏蔽信息
        mov     ah, al                  ; 保存以供后面使用
        IODELAY
        mov     al, 11h                 ; 初始命令1，cascade&
        out     20h, al                 ;    需要第4个初始化字节
        IODELAY
        mov     al, 50h                 ; 初始化命令2，
        out     21h, al                 ;    切换IRQ到中断50h
        IODELAY
        mov     al, 4                   ; 初始化命令3
        out     21h, al                 ;    控制器通IRQ接通从控制器
        IODELAY
        mov     al, 1                   ; 初始化命令4，普通EOI
        out     21h, al                 ;    非缓冲，80x86
```

```
        IODELAY
        mov     al, ah                  ; 操作命令1
        out     21h, al                 ;  恢复中断屏蔽

                                        ; 复制中断向量
                                        ; ------------------------
        xor     ax, ax
        mov     es, ax                  ; 将段寄存器设置为
        mov     ds, ax                  ;  指向中断表
        mov     cx, 8 * 2               ; 要传送的字数
        mov     si, 8 * 4               ; 从中断8开始
        mov     di, 50h * 4             ; 传送到中断50-57h
        cld
        rep     movsw                   ; 传送中断向量
        sti                             ; 开放中断
        ret
IRQ_redirect    endp
```

每个中断 50h~57h 可以从八个中断程序中选择调用一个。这些程序轮流调用安装在 8 到 F 处的向量。这种方法可用于多任务操作，或者有必要在其他可能挂起该向量的使用者之前或之后获得中断控制时，使用这种方法。与上面采用的直接复制向量的方法不同，中断 50h~57h 指向八个程序，这些程序和下面的中断 51h 类似。

```
;-------------------------------------------------------------------------------
;   INTERRUPT_51h
;       可替换代码——参见上文叙述
;
;       用到的寄存器:  所有

Interrupt_51h   proc    far

; <<<<可能会在此处放入其他需要在前面处理的代码>>>>

        push    ax
        push    es
        mov     ax, 0
```

```
        mov     es, ax                          ; 中断表的段址
        mov     ax, es:[9 * 4]                  ; 当前中断9的偏移量
        mov     cs:[old_int_9_off], ax          ; 保存
        mov     ax, es:[9 * 4 + 2]              ; 当前中断9的段
        mov     cs:[old_int_9_seg], ax          ; 保存
        pop     es
        pop     ax
        pushf
        call    dword ptr cs:[old_int_9_off]    ; 调用当前的中断9

; <<<<可能会在此处放入其他需要在后面处理的代码>>>>

        ret     2                               ; 除去多余的标志位
Interrupt_51h   endp

old_int_9_off   dw      0                       ; 临时的中断9偏移量
old_int_9_seg   dw      0                       ; 临时的中断9段址
```

通常，当出现指定的中断时，TSR 或驱动程序会挂起（booked）中断来获得控制。挂起中断是获得控制的一种定义明确的方法。不推荐使用这些代码例来获得控制，除非在极特殊的情况下。

端口归纳

下面这个端口列表用来控制 8259 中断控制器。

端口	类型	功能	平台
20h	输入	8259-1 读中断请求/服务寄存器	所有
20h	输出	8259-1 中断命令	所有
21h	输入	8259-1 中断屏蔽寄存器	所有
21h	输出	8259-1 中断命令	所有
70h	输出	开放 NMI	AT+
A0h	输入	8259-2 读中断请求/服务寄存器	AT+
A0h	输出	8259-2 中断命令	AT+
A0h	输出	不可屏蔽中断控制	PC/XT
A1h	输入	8259-2 中断屏蔽寄存器	AT+
A1h	输出	8259-2 中断命令	AT+

462h	输出	触发 NMI	EISA
4D0h	I/O	8259-1 IRQ 触发字节	EISA
4D1h	I/O	8259-2 IRQ 触发字节	EISA

端口细节

端口	类型	描述	平台
20h	输入	8259-1 读中断请求/服务寄存器	所有

这个功能读取中断请求寄存器（IRR）或中断服务寄存器（ISR）的内容。通过向端口 20h 发送一条命令，你可以指定要读取哪个寄存器。命令值 0Ah 选择 IRR，而命令值 0Bh 选择 ISR。一旦发送了一条命令，可以多次执行读操作来获得同一个寄存器的内容。没有必要重新发送寄存器选择命令。

参看端口 20h，输出命令 0Ah 和 0Bh，了解这些寄存器的更多细节。

端口	类型	描述	平台
20h	输出	8259-1 中断命令	所有

这个寄存器控制中断控制器对中断请求线 0~7 的初始化和操作。最常见的使用是触发中断结束，命令 20h。如下可以实现这一点：

```
mov    AL,  20h
out    20h,  AL          ; 触发 EOI
```

初始化

通常只有 BIOS 在重启时初始化中断控制器。初始化进程需要向端口 20h 发送一个 8 位的命令，然后连续向端口 21h 发送两到三个 8 位命令。下面只显示了可任选的选项，其他的选择和 PC 平台不兼容。参看端口 21h 了解其他初始化字节。

输出（位 0~7），初始化命令 1

位	7 w = 0	未使用
	6 w = 0	未使用
	5 w = 0	未使用
	4 w = 1	启动初始化
	3 w = 0	边沿触发模式（PC、XT、AT）
	1	电平触发模式（MCA）
	2 w = 0	未使用

1 w = 0	级联模式（使用两个控制器时，AT+）
1	单模式（使用一个控制器时，PC/XT）
0 w = 0	要求第四个初始化字节

普通操作

一旦初始化好了中断控制器后，这个端口可用来接受许多命令。在定义了每条命令的位后，详细说明了这些命令。

输出（位 0~7），操作命令 2

位 7 w = x, 6 w = x, 5 w = x 子命令（*指示很少使用的命令）

位 7	位 6	位 5
0	0	0 = 自动 EOI 模式下的翻转(清除)*
0	0	1 = 非专用的 EOI
0	1	0 = 无操作*
0	1	1 = 专用的 EOI
1	0	0 = 自动 EOI 模式下的翻转(设置)*
1	0	1 = 非专用的 EOI 上的翻转*
1	1	0 = 设置优先级命令
1	1	1 = 专用的 EOI 上的翻转*

4 w = x, 3 w = x, 2 w = x 命令 2

1 w = x, 0 w = x 优先级水平（命令：设置优先级，专用 EOI 上的翻转）

位 2	位 1	位 0	最低	最高
0	0	0 =	IRQ0	IRQ1
0	0	1 =	IRQ1	IRQ2
0	1	0 =	IRQ2	IRQ3
0	1	1 =	IRQ3	IRQ4
1	0	0 =	IRQ4	IRQ5
1	0	1 =	IRQ5	IRQ6
1	1	0 =	IRQ6	IRQ7
1	1	1 =	IRQ7	IRQ0（缺省）

要重设的在服务位（命令：专用的 EOI）

位 2	位 1	位 0
0	0	0 = 重设在服务位 0
0	0	1 = 重设在服务位 1
0	1	0 = 重设在服务位 2
0	1	1 = 重设在服务位 3

1	0	0 = 重设在服务位 4
1	0	1 = 重设在服务位 5
1	1	0 = 重设在服务位 6
1	1	1 = 重设在服务位 7

输出（位 0~7），操作命令 3

位

7 w = 0	固定
6 w = x	专用屏蔽模式
5 w = x	位 6 位 5
	0 x = 无操作
	0 0 = 清除专用屏蔽模式
	1 1 = 设置专用屏蔽模式
4 w = 0	命令 3
3 w = 1	
2 w = 0	无查询命令
1 w = x	读指定的寄存器
0 w = x	位 1 位 0
	0 x = 无操作
	1 0 = 读中断请求寄存器
	1 1 = 读在服务寄存器

命令归纳

命令号	端口 20h 命令
00h	自动 EOI 模式下翻转（清除）
0Ah	读中断请求寄存器
0Bh	读在服务寄存器
20h	中断结束命令
40h	无操作
48h	清除专用屏蔽模式
60h	专用 EOI——IRQ 0
61h	专用 EOI——IRQ 1
62h	专用 EOI——IRQ 2
63h	专用 EOI——IRQ 3
64h	专用 EOI——IRQ 4
65h	专用 EOI——IRQ 5
66h	专用 EOI——IRQ 6
67h	专用 EOI——IRQ7

68h	设置专用屏蔽模式
80h	自动 EOI 模式下翻转（设置）
A0h	非专用 EOI 上的翻转
C0h	IRQ 0 最低优先级
C1h	IRQ 1 最低优先级
C2h	IRQ 2 最低优先级
C3h	IRQ 3 最低优先级
C4h	IRQ 4 最低优先级
C5h	IRQ 5 最低优先级
C6h	IRQ 6 最低优先级
C7h	IRQ 7 最低优先级
E0h	EOI 和 IRQ 0 最低优先级
E1h	EOI 和 IRQ 1 最低优先级
E2h	EOI 和 IRQ 2 最低优先级
E3h	EOI 和 IRQ 3 最低优先级
E4h	EOI 和 IRQ 4 最低优先级
E5h	EOI 和 IRQ 5 最低优先级
E6h	EOI 和 IRQ 6 最低优先级
E7h	EOI 和 IRQ 7 最低优先级

命令细节

命令	描述	端口
00h	自动 EOI 模式下翻转（清除）	20h

从技术上讲，8259 提供了这条命令，但是在任何一个 PC 系统上，自动 EOI 不是一个实际可用的选项，这条命令实际上是不可用的。如果中断控制器处于自动 EOI 模式下，在 CPU 返回中断确认后，立即触发中断结束命令。版权早于 1985 年的早期的 8259 芯片上有一个 bug，在使用两个中断控制器时会阻止第二个中断控制器使用 EOI。庆幸的是，现在这些老式的芯片早已成为了历史。

命令	描述	端口
0Ah	读中断请求寄存器	20h

在触发这条命令之后，从端口 20h 的下一次读取将获得中断请求寄存器的内容。中断请求时，相关的位被设置。在触发这个命令之前应禁止中断，在从端口 20h 读取这个寄存器之后应重新开放中断。

中断请求寄存器用来查看在禁止中断期间是否挂起了某个硬件中断。这个寄存器也可用来在一个中断服务处理程序内标识是否挂起了低优先级的中断。

输入（位 0~7 ）

位		
	7 w = 1	IRQ 7 请求服务
	6 w = 1	IRQ 6 请求服务
	5 w = 1	IRQ 5 请求服务
	4 w = 1	IRQ 4 请求服务
	3 w = 1	IRQ 3 请求服务
	2 w = 1	IRQ 2 请求服务
	1 w = 1	IRQ 1 请求服务
	0 w = 1	IRQ 0 请求服务

命令	描述	端口
0Bh	读中断在服务寄存器	20h

在触发这条命令之后，从端口 20h 的下一次读取将获得中断在服务寄存器的内容。这个寄存器包含有当前正在服务的中断。

大多数情况下，只有一个中断正在被服务，但是如果在处理低优先级的中断过程中出现了高优先级的中断，那么这个寄存器将指示多个中断。80286 及后来的系统也使用中断在服务寄存器来确定一个中断是由硬件引发还是由 CPU 异常造成。如果出现中断时没有设置相关的在服务位，那么表示是 CPU 引发了这个中断而不是硬件。

一个中断结束（EOI）命令将清除当前最高优先级的在服务位。

输入（位 0~7 ）

位		
	7 w = 1	IRQ 7 在服务
	6 w = 1	IRQ 6 在服务
	5 w = 1	IRQ 5 在服务
	4 w = 1	IRQ 4 在服务
	3 w = 1	IRQ 3 在服务
	2 w = 1	IRQ 2 在服务
	1 w = 1	IRQ 1 在服务
	0 w = 1	IRQ 0 在服务

命令	描述	端口
20h	中断结束命令	20h

在硬件中断服务程序结束时，程序对 IRQ 0~7 触发一条中断结束命令。它指示中断控制器接受其他的中断，同时接受在执行当前中断期间被锁住的低优先级中断。

使用 IRQ 8~15 时，有必要先向第二个中断控制器（端口 A0h）触发一条 EOI 命令，

接着对主中断控制器（端口 20h）发送另一条 EOI 命令。

命令	描述	端口
40h	无操作	20h

这个子命令执行一个空操作。很少使用这个命令。

命令	描述	端口
48h	清除专用屏蔽模式	20h

将控制器恢复为普通操作。参看设置专用屏蔽模式了解该操作的细节。

命令	描述	端口
60~67h	专用 EOI	20h

重设一个 IRQ 的专用在服务位，支持新的中断服务。命令 60h~67h 分别与 IRQ 0~7 相关。对于 PC 系列，通常使用命令 20h，一般的 EOI，来替代这些命令清除相应的在服务位。

命令	描述	端口
68h	设置专用屏蔽模式	20h

激活专用屏蔽模式操作。在某些情况下，一个中断服务程序可能希望在当前的服务程序执行期间允许出现一个更低优先级的中断。普通的操作阻止处理任何其他更低优先级的中断请求，直到执行完当前高优先级的中断程序。带有专用的屏蔽模式设置时，可以更新屏蔽寄存器（端口 21h）来禁止当前的请求，而支持其他任何请求，包括更低优先级的中断请求。

在启动专用屏蔽寄存器的服务程序结束之前，程序必须清除专用屏蔽模式，将屏蔽寄存器恢复到以前的状态。

命令	描述	端口
80h	自动 EOI 模式下的循环，设置	20h

从技术上将，8259 提供了这条命令，但是在任何一个 PC 系统上，自动 EOI 不是一个实际可用的选项，这条命令实际上是不可用的。

命令	描述	端口
A0h	非专用 EOI 上的循环	20h

这个命令执行两项操作。在中断服务程序的结尾处触发它时，中断控制器将已完成的中断的优先级设置位到最低，然后接受其他的中断。

在 PC 环境下很少使用这个命令。使用这个方法要求所有的中断在完成时触发这条命令。这个功能并不实用，除非你正在创建一个自定义的系统 BIOS。相反地，所有的 BIOS 程序都使用了非专用 EOI 命令，20h。

命令	描述	端口
C0~C7h	IRQ 最低优先级	20h

将一个 IRQ 的优先级设置为最低。命令 C0h~C7h 对应于将 IRQ 0~7 设置为最低优先级。表 17-5 归纳了这个操作。

表 17-5　命令优先级

优先级	
命　　令	最低　　最高
C0h	0 7 6 5 4 3 2 1
C1h	1 0 7 6 5 4 3 2
C2h	2 1 0 7 6 5 4 3
C3h	3 2 1 0 7 6 5 4
C4h	4 3 2 1 0 7 6 5
C5h	5 4 3 2 1 0 7 6
C6h	6 5 4 3 2 1 0 7
C7h	7 6 5 4 3 2 1 0

命令	描述	端口
E0~E7h	EOI 和设置最低优先级	20h

这个命令执行两项操作。在中断服务程序的结尾处触发它来执行和专用 EOI 相同的操作，并允许下一个中断。参看命令 60h~67h，以更多的了解专用 EOI。第二项操作是将一个 IRQ 的优先级设置为最低。命令 E0h~E7h 对应于将 IRQ 0~7 设置为最低优先级。表 17-6 归纳了这个操作。

在 PC 环境下很少使用这个命令。它们特意为那些所有中断优先级相同的系统而设计。如果总是激活八个中断，每个可能会每八次中才被服务一次。

表 17-6 EOI 和命令优先级

优先级		
命　　令	最低　最高	专用 EOI
E0h	0 7 6 5 4 3 2 1	IRQ 0
E1h	1 0 7 6 5 4 3 2	IRQ 1
E2h	2 1 0 7 6 5 4 3	IRQ 2
E3h	3 2 1 0 7 6 5 4	IRQ 3
E4h	4 3 2 1 0 7 6 5	IRQ 4
E5h	5 4 3 2 1 0 7 6	IRQ 5
E6h	6 5 4 3 2 1 0 7	IRQ 6
E7h	7 6 5 4 3 2 1 0	IRQ 7

命令	描述		端口
21h	输入	8259-1 中断屏蔽寄存器	所有

这个中断屏蔽寄存器指示了允许哪个中断请求（值 0），以及禁止哪个中断请求（值 1）

输入（位 0~7）中断屏蔽寄存器

位	7 w = 0	开放 IRQ 7
	6 w = 0	开放 IRQ 6
	5 w = 0	开放 IRQ 5
	4 w = 0	开放 IRQ 4
	3 w = 0	开放 IRQ 3
	2 w = 0	开放 IRQ 2
	1 w = 0	开放 IRQ 1
	0 w = 0	开放 IRQ 0

命令	描述		端口
21h	输出	8259-1 中断命令	所有

这个寄存器控制中断控制器对中断请求线 0~7 的初始化和操作。

初始化

在从端口 20h 发送了初始化命令后，另外两到三个完成该进程。使用单模式时会跳过初始化命令字节 3，单模式在初始化命令字节 1 中由位 1=1 指定（PC/XT）。

输出（位 0~7），初始化命令 2

位　7 w = x　指定这个中断控制器的 8 中断组
　　6 w = x　位 7　6　5　4　3　2　1
　　5 w = x　0　0　0　0　0　0　0 = 使用向量 0~7
　　4 w = x　0　0　0　0　0　0　1 = 使用向量 8~F(普通情况)
　　3 w = x　....
　　1　1　1　1　1　1　1 = 使用向量 8F~FF
　　2 w = 0　未使用
　　1 w = 0　未使用
　　0 w = 0　未使用

输出（位 0~7）初始化命令 3

位　7 w = 0　没有从设备连接到 IRQ 7
　　6 w = 0　没有从设备连接到 IRQ 6
　　5 w = 0　没有从设备连接到 IRQ 5
　　4 w = 0　没有从设备连接到 IRQ 4
　　3 w = 0　没有从设备连接到 IRQ 3
　　2 w = 0　没有从设备连接到 IRQ 2（PC/XT 缺省）
　　1　从控制器连接到 IRQ2（AT+缺省）
　　1 w = 0　没有从设备连接到 IRQ 1
　　0 w = 0　没有从设备连接到 IRQ 0

输出（位 0~7）初始化命令 4

位　7 w = 0　未使用
　　6 w = 0　未使用
　　5 w = 0　未使用
　　4 w = 0　非特定完全嵌套模式
　　3 w = x　缓冲模式
　　2 w = x　位 3　位 2
　　0　x = 无缓冲（AT+）
　　1　0 = 缓冲）PC/XT）
　　1 w = 0　普通的中断结束（EOI）
　　1　自动中断结束（任何系统都不支持）
　　0 w = 1　80x86 模式

普通操作

一旦初始化好了中断控制器后，就可以开放或禁止单独的中断请求线。将一位设置为 1 可以禁止该请求。在 BIOS POST 操作的结尾，开放了所有的中断。

输出（位 0~7），操作命令 1

位		
	7 w = 0	开放 IRQ 7
	6 w = 0	开放 IRQ 6
	5 w = 0	开放 IRQ 5
	4 w = 0	开放 IRQ 4
	3 w = 0	开放 IRQ 3
	2 w = 0	开放 IRQ 2
	1 w = 0	开放 IRQ 1
	0 w = 0	开放 IRQ 0

端口	类型	描述	平台
70h	输出	开放 NMI	AT+

在 POST 结尾自动开放 NMI。向这个端口写将设置 NMI 的状态。因为对端口 71h 的读写可以访问 CMOS 内存，所以低 7 位值并不重要。

输出（位 0~7） CMOS 地址和 AMI

位		
	7 w = 0	允许不可屏蔽中断，NMI 中断 2
	6 w = x	未使用
	5 w = x	CMOS RAM 地址（参看端口 CMOS 一章中的端口 70h）
	4 w = x	
	3 w = x	
	2 w = x	
	1 w = x	
	0 w = x	

端口	类型	描述	平台
A0h	输入	8259-2 读中断请求/服务寄存器	AT+

这个功能读取中断请求寄存器（IRR）或中断服务寄存器（ISR）的内容。通过向端口 A0h 发送一条命令，你可以指定要读取哪个寄存器。命令值 0Ah 选择 IRR，而命令值 0Bh 选择 ISR。一旦发送了一条命令，可以多次执行读操作来获得同一个寄存器的内容。没有必要重新发送寄存器选择命令。

参看端口 A0h，输出命令 0Ah 和 0Bh，了解这些寄存器的更多细节。

端口	类型	描述	平台
A0h	输出	8259-2 中断命令	AT+

这个寄存器控制中断控制器对中断请求线 8~15 的初始化和操作。

初始化

通常只有 BIOS 在重启时初始化中断控制器。初始化进程需要向端口 A0h 发送一个 8 位的命令，然后连续向端口 A1h 发送两到三个 8 位命令。下面只显示了可任选的选项，其他的选择和 PC 平台不兼容。参看端口 A1h 了解其他初始化字节。

输出（位 0~7），初始化命令 1

位		
	7 w = 0	未使用
	6 w = 0	未使用
	5 w = 0	未使用
	4 w = 1	启动初始化
	3 w = 0	边沿触发模式（AT）
	1	电平触发模式（MCA）
	x	未使用（EISA——从端口 4D1h 控制）
	2 w = 0	未使用
	1 w = 0	级联模式（使用两个控制器时，AT+）
	0 w = 0	要求第四个初始化字节

普通操作

一旦初始化好了中断控制器后，这个端口可用来接受许多命令。在定义了每条命令的位后，详细说明了这些命令。

输出（位 0~7），操作命令 2

位					
	7 w = x	子命令（*指示很少使用的命令）			
	6 w = x	位 7	位 6	位 5	
	5 w = x	0	0	0 = 自动 EOI 模式下的翻转(清除)*	
		0	0	1 = 非专用的 EOI	
		0	1	0 = 无操作*	
		0	1	1 = 专用的 EOI	
		1	0	0 = 自动 EOI 模式下的翻转（设置）*	
		1	0	1 = 非专用的 EOI 上的翻转*	
		1	1	0 = 设置优先级命令	
		1	1	1 = 专用的 EOI 上的翻转*	
	4 w = x	命令 2			
	3 w = x				
	2 w = x	优先级水平（命令：设置优先级，			
	1 w = x	专用 EOI 上的翻转）			
	0 w = x	位 2	位 1	位 0	最低 最高

0	0	0 =	IRQ8	IRQ9
0	0	1 =	IRQ9	IRQ10
0	1	0 =	IRQ10	IRQ11
0	1	1 =	IRQ11	IRQ12
1	0	0 =	IRQ12	IRQ13
1	0	1 =	IRQ13	IRQ14
1	1	0 =	IRQ14	IRQ15
1	1	1 =	IRQ15	IRQ8（缺省）

要重设的在服务位（命令：专用的 EOI）

位 2	位 1	位 0
0	0	0 = 重设在服务位 0
0	0	1 = 重设在服务位 1
0	1	0 = 重设在服务位 2
0	1	1 = 重设在服务位 3
1	0	0 = 重设在服务位 4
1	0	1 = 重设在服务位 5
1	1	0 = 重设在服务位 6
1	1	1 = 重设在服务位 7

输出（位 0~7），操作命令 3

位

7 w = 0	固定	
6 w = x 5 w = x	专用屏蔽模式	
	位 6	位 5
	0	x = 无操作
	0	0 = 清除专用屏蔽模式
	1	1 = 设置专用屏蔽模式
4 w = 0 3 w = 1	命令 3	
2 w = 0	无查询命令	
1 w = x 0 w = x	读指定的寄存器	
	位 1	位 0
	0	x = 无操作
	1	0 = 读中断请求寄存器
	1	1 = 读在服务寄存器

命令统计

命令号	端口 20h 命令
00h	自动 EOI 模式下翻转（清除）

0Ah	读中断请求寄存器
0Bh	读在服务寄存器
20h	中断结束命令
40h	无操作
48h	清除专用屏蔽模式
60h	专用 EOI——IRQ 8
61h	专用 EOI——IRQ 9
62h	专用 EOI——IRQ 10
63h	专用 EOI——IRQ 11
64h	专用 EOI——IRQ 12
65h	专用 EOI——IRQ 13
66h	专用 EOI——IRQ 14
67h	专用 EOI——IRQ15
68h	设置专用屏蔽模式
80h	自动 EOI 模式下翻转（设置）
A0h	非专用 EOI 上的翻转
C0h	IRQ 8 最低优先级
C1h	IRQ 9 最低优先级
C2h	IRQ 10 最低优先级
C3h	IRQ 11 最低优先级
C4h	IRQ 12 最低优先级
C5h	IRQ 13 最低优先级
C6h	IRQ 14 最低优先级
C7h	IRQ 15 最低优先级
E0h	EOI 和 IRQ 8 最低优先级
E1h	EOI 和 IRQ 9 最低优先级
E2h	EOI 和 IRQ 10 最低优先级
E3h	EOI 和 IRQ 11 最低优先级
E4h	EOI 和 IRQ 12 最低优先级
E5h	EOI 和 IRQ 13 最低优先级
E6h	EOI 和 IRQ 14 最低优先级
E7h	EOI 和 IRQ 15 最低优先级

命令细节

命令	描述	端口
00h	自动 EOI 模式下翻转（清除）	A0h

从技术上将，8259 提供了这条命令，但是在任何一个 PC 系统上，自动 EOI 不是一个

实际可用的选项，这条命令实际上是不可用的。参看端口 20h 下的命令 00h 了解细节

命令	描述	端口
0Ah	读中断请求寄存器	A0h

在触发这条命令之后，从端口 A0h 的下一次读取将获得中断请求寄存器的内容。中断请求时，相关的位被设置。在触发这个命令之前应禁止中断，在从端口 A0h 读取这个寄存器之后应重新开放中断。

输入（位 0~7 ）

位		
	7 w = 1	IRQ 15 请求服务
	6 w = 1	IRQ 14 请求服务
	5 w = 1	IRQ 13 请求服务
	4 w = 1	IRQ 12 请求服务
	3 w = 1	IRQ 11 请求服务
	2 w = 1	IRQ 10 请求服务
	1 w = 1	IRQ 9 请求服务
	0 w = 1	IRQ 8 请求服务

命令	描述	端口
0Bh	读中断在服务寄存器	A0h

在触发这条命令之后，从端口 A0h 的下一次读取将获得中断在服务寄存器的内容。这个寄存器包含有当前正在服务的中断。在触发这个命令之前应禁止中断，在从端口 A0h 读取这个寄存器之后应重新开放中断。

大多数情况下，只有一个中断正在被服务，但是如果在处理低优先级的中断过程中出现了高优先级的中断，那么这个寄存器将指示多个中断。

一个中断结束（EOI）命令将清除当前最高优先级的在服务位。

输入（位 0~7 ）

位		
	7 w = 1	IRQ 15 在服务
	6 w = 1	IRQ 14 在服务
	5 w = 1	IRQ 13 在服务
	4 w = 1	IRQ 12 在服务
	3 w = 1	IRQ 11 在服务
	2 w = 1	IRQ 10 在服务
	1 w = 1	IRQ 9 在服务
	0 w = 1	IRQ 8 在服务

命令	描述	端口
20h	中断结束命令（EOI）	A0h

在硬件中断服务程序结束时，程序对 IRQ 8~15 触发一条中断结束命令。它通知中断控制器接受其他的中断，同时接受在执行当前中断期间被锁住的低优先级中断。

使用 IRQ 8~15 时，有必要先向第二个中断控制器（端口 A0h）触发一条 EOI 命令，接着对主中断控制器（端口 20h）发送另一条 EOI 命令。

命令	描述	端口
40h	无操作	A0h

这个子命令执行一个空操作。很少使用这个命令。

命令	描述	端口
48h	清除专用屏蔽模式	A0h

将控制器恢复为普通操作。

命令	描述	端口
60~67h	专用 EOI	A0h

重设一个 IRQ 的专用在服务位，支持新的中断服务。命令 60h~67h 分别与 IRQ 8~15 相关。对于 PC 系列，通常使用命令 20h，一般的 EOI，来替代这些命令清除相应的在服务位。

命令	描述	端口
68h	设置专用屏蔽模式	A0h

激活专用屏蔽模式操作。在某些情况下，一个中断服务程序可能希望在当前的服务程序执行期间允许出现一个更低优先级的中断。普通的操作阻止处理任何其他更低优先级的中断请求，直到执行完当前高优先级的中断程序。带有专用的屏蔽模式设置时，可以更新屏蔽寄存器（端口 A1h）来禁止当前的请求，而支持其他任何请求，包括更低优先级的中断请求。

在启动专用屏蔽寄存器的服务程序结束之前，程序必须清除专用屏蔽模式，将屏蔽寄存器恢复到以前的状态。

命令	描述	端口
80h	自动EOI模式下的旋转（设置）	A0h

从技术上将，8259 提供了这条命令，但是在任何一个 PC 系统上，自动 EOI 不是一个实际可用的选项，这条命令实际上是不可用的。

命令	描述	端口
A0h	非专用 EOI 上的翻转	A0h

这个命令执行两项操作。在中断服务程序的结尾处触发它时，中断控制器将已完成的中断的优先级设置位到最低，然后接受其他的中断。

在 PC 环境下很少使用这个命令。使用这个方法要求所有的中断在完成时触发这条命令。相反地，所有的 BIOS 程序都使用了非专用 EOI 命令，20h。这个功能并不实用，除非你正在创建一个自定义的系统 BIOS。

命令	描述	端口
C0~C7h	IRQ 最低优先级	A0h

将一个 IRQ 的优先级设置为最低。命令 C0h~C7h 对应于将中断控制器 2 上的 IRQ 8~15 设置为最低优先级。IRQ8~15 组成一组，看起来就象系统带有一个 IRQ2 的优先级一样。缺省的优先级如表 17-7 所示。大多数系统都以这种缺省形式配置大多数时间。

表 17-6　中断缺省的优先级

控制器 1	控制器 2	
请　求	请　求	缺省的优先级
IRQ0		1（最高）
IRQ1		2
IRQ2		链接到第二个控制器
	IRQ8	3
	IRQ9	4
	IRQ10	5
	IRQ11	6
	IRQ12	7
	IRQ13	8
	IRQ14	9
	IRQ15	10
IRQ3		11
IRQ4		12
IRQ5		13
IRQ6		14
IRQ7		15（最低）

命令	描述	端口
E0~E7h	EOI 和设置最低优先级	A0h

这个命令为 IRQ 8~15 设置专用 EOI，同时还设置中断控制器 2 的最低优先级。参看端口 20h 的命令 E0~E7h 了解细节。在 PC 环境下通常不使用这个命令。

命令	描述	端口
A1h	输入 8259-2 中断屏蔽寄存器	AT+

这个中断屏蔽寄存器指示了允许哪个中断请求（值 0），以及禁止哪个中断请求（值 1）

输入（位 0~7）中断屏蔽寄存器

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

端口	类型	描述	平台
A1h	输出	8259-2 中断命令	AT+

这个寄存器控制中断控制器对中断请求线 8~15 的初始化和操作。

初始化

在从端口 A0h 发送了初始化命令后，另外两到三个完成 8259 初始化。

输出（位 0~7），初始化命令 2

位									
7 w = x	指定这个中断控制器的 8 中断组								
6 w = x	位	7	6	5	4	3	2	1	
5 w = x	0	0	0	0	0	0	0 =	使用向量 0~7	
4 w = x	0	0	0	0	0	0	1 =	使用向量 8~F(普通情况)	
3 w = x			....						
	1	1	1	1	1	1	1 =	使用向量 8F~FF	
2 w = 0	未使用								
1 w = 0	未使用								

0 w = 0　　未使用

输出（位 0~7）初始化命令 3

位		
	7 w = 0	未使用
	6 w = 0	未使用
	5 w = 0	未使用
	4 w = 0	未使用
	3 w = 0	未使用
	2 w = 0	从控制器 ID
	1 w = 1	位 2　位 1　位 0
	0 w = 0	0　1　0 = IRQ2 上的从控制器

输出（位 0~7）初始化命令 4

位		
	7 w = 0	未使用
	6 w = 0	未使用
	5 w = 0	未使用
	4 w = 0	非特定完全嵌套模式
	3 w = x	缓冲模式
	w = x	位 3　位 2
		0　x = 无缓冲（AT+）
		1　0 = 缓冲）PC/XT）
	1 w = 0	普通的中断结束（EOI）
	1	自动中断结束（任何系统都不支持）
	0 w = 1	80x86 模式

普通操作

一旦初始化好了中断控制器后，就可以开放或禁止单独的中断请求线。将一位设置为 1 可以禁止该请求。

输出（位 0~7），操作命令 1

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

端口	类型	描述	平台
462h	输出	触发 NMI	EISA

开放 NMI 时任何一次向这个端口的写操作都会引发一个 NMI。忽略写入的实际值。

端口	类型	描述	平台
4D0h	I/O	8259-1 IRQ 触发类型	EISA

这个寄存器控制每条 IRQ 线的触发类型。清除该位代表边沿触发（AT 总线兼容机器的缺省模式）。设置该位则选择电平触发模式（PCI 总线插槽使用这种模式）。

在写这个寄存器之前，先读取它的内容。不要改变 IRQ 0、1、2 位，因为它们由主板制造商的 BIOS 设置，用来反映特定的主板设计。

I/O（位 0~7）　IRQ 触发类型

位		
	7 r/w = 0	IRQ 7 边沿触发
	6 r/w = 0	IRQ 6 边沿触发
	5 r/w = 0	IRQ 5 边沿触发
	4 r/w = 0	IRQ 4 边沿触发
	3 r/w = 0	IRQ 3 边沿触发
	2 r/w = x	IRQ 2 触发方式（不要改变）
	1 r/w = x	IRQ 1 触发方式（不要改变）
	0 r/w = x	IRQ 0 触发方式（不要改变）

端口	类型	描述	平台
4D1h	I/O	8259-2 IRQ 触发类型	EISA

这个寄存器控制每条 IRQ 线的触发类型。清除该位代表边沿触发（ISA 总线兼容机器的缺省模式）。设置该位则选择电平触发模式（PCI 总线插槽使用这种模式）。

在写这个寄存器之前，先读取它的内容。不要改变 IRQ 8 或 IRQ 13 的状态，因为它们由主板制造商的 BIOS 设置，用来反映特定的主板设计。

I/O（位 0~7）　IRQ 触发类型

位		
	7 r/w = 0	IRQ 15 边沿触发
	6 r/w = 0	IRQ 14 边沿触发
	5 r/w = 0	IRQ 13 触发方式（不要改变）
	4 r/w = 0	IRQ 12 边沿触发
	3 r/w = 0	IRQ 11 边沿触发
	2 r/w = x	IRQ 10 边沿触发
	1 r/w = x	IRQ 9 边沿触发
	0 r/w = x	IRQ 8 触发方式（不要改变）

第 18 章

DMA 服务和 DRAM 刷新

在和别的软件工程师交谈时，我惊奇的发现他们对直接内存访问（DMA）存在许多误解。本章将彻底解释 DMA 是如何工作的。少数程序员会发现，有必要直接对 DMA 系统编程，但是理解这一点对许多 PC 子系统来说是非常重要的。

过去的几年中，PC、AT、EISA 以及 MCA 系统有了很大的变化，我也已详细的说明了这一点。另外，还简要的涉及了虚拟设备规范。虚拟设备规范解决了许多保护模式下软件和 DMA 的杂乱的问题。提供了一个小程序例子，来看看如何控制 DMA 系统。接着是一个 75 个以上端口的参考说明，这些端口用来在不同的 PC 平台上对 DMA 系统编程。

简　介

直接内存访问提供了一种方法来实现内存和 I/O 总线之间的高速数据传送。一个 DMA 控制器在物理内存上执行这些传送而不用处理器的介入。

一旦设置好了后，一个专用的 IC，8237 DMA 控制器，会自动执行数据传送。老式的 AT 以前的类型的机器只使用一个 DMA 芯片。它提供了四个独立的 DMA 通道，每个可以实现 8 位内存传送（DMA-1）。从 AT 开始，所有目前的 PC，都又添加了一个 DMA 控制器（DMA-2），这个控制器提供了另外四个通道，每个可以实现 16 位内存传送。保留了一个 16 位通道，用来链接上第一个控制器，DMA-1。所以，总共可用七个通道。

每个 DMA 控制器有许多寄存器来控制 DMA 操作、指定要传送的字节/字数以及为传送分配内存地址。这些寄存器都通过 I/O 端口来访问。通道 0~3 的操作不同于通道 4~7，如表 18-1 所示。

表 18-1 DMA 用法和限制

通道号	功能	系统	DMA 芯片数	数据大小	最大块
0	DRAM 刷新	PC	1	8	64K
	未分配，所有其他的	AT+	1	8	64K
1	未分配，	所有	1	8	64K
2	软盘	所有	1	8	64K
3	硬盘	所有	1	8	64K
	未分配，所有其他的	AT+			
4	级联 DMA-1	AT+	2	16	128K
5	硬盘	PS/2	2	16	128K
	未分配，所有其他的	AT/EISA			
6	未分配	AT+	2	16	128K
7	未分配	AT+	2	16	128K

8237DMA 控制器使用其内部 16 位地址寄存器只能访问 64K 不同的内存地址。为了访问全部的 16MB 地址空间（8088 机器上 1MB 空间），必须为每个 DMA 通道提供一个独立的页面寄存器。每个页面寄存器为每个通道加入另外的 8 位。与 DMA 通道相关的页面寄存器被装入到指定使用的内存区。这意味着，在通道 0~3 上传送数据必须位于一个 64K 边界的 64K 块内。16 位通道 5~7 必须位于一个 128K 边界的 128K 块内。表 18-2 和表 18-3 显示了 DMA 传送期间由页面寄存器控制的端口和地址线。

表 18-2 早期的 PC 页面寄存器

		页面寄存器的地址位							
页面寄存器	端口	0	1	2	3	4	5	6	7
DMA 通道 0	80h	a16	a17	a18	a19	x	x	x	x
DMA 通道 1	81h	a16	a17	a18	a19	x	x	x	x
DMA 通道 2	82h	a16	a17	a18	a19	x	x	x	x
DMA 通道 3	83h	a16	a17	a18	a19	x	x	x	x

表 18-3 AT/EISA/MCA 页面寄存器

页面寄存器的地址位									
页面寄存器	端口	0	1	2	3	4	5	6	7
DMA 通道 0	87h	a16	a17	a18	a19	a20	a21	a22	a23
DMA 通道 1	83h	a16	a17	a18	a19	a20	a21	a22	a23
DMA 通道 2	81h	a16	a17	a18	a19	a20	a21	a22	a23
DMA 通道 3	82h	a16	a17	a18	a19	a20	a21	a22	a23
DMA 刷新*	8Fh	a16*	a17	a18	a19	a20	a21	a22	a23
DMA 通道 5	8Bh	a16*	a17	a18	a19	a20	a21	a22	a23
DMA 通道 6	89h	a16**	a17	a18	a19	a20	a21	a22	a23
DMA 通道 7	8Ah	a16**	a17	a18	a19	a20	a21	a22	a23

*DMA 通道 4 被保留用于级联 DMA-1DMA-2，其页面寄存器用于 DMA 刷新。

**向通道 4~7 页面寄存器的第 0 位写入非零值时，输出总是零，这样保证 DMA 传送总在 128K 边界上。

所有的 DMA 内存传送必须在第一个 16MB 地址空间内。不支持 DMA 访问 16MB 以外的空间。另外，必须将通道 4~7 的访问地址排列在字边界上。这些通道仅用于传送 16 位数据。8 位 I/O 适配器不能使用它们。

DMA 系统可以执行三种类型的传送：读、写以及校验，如表 18-4 所示。DMA 读操作从内存向 I/O 传送数据。DMA 写操作从 I/O 向内存传送数据。校验操作类似于 DMA 读操作，但是不向 I/O 写。适配器能读取数据并在适配器内执行校验操作。尽管许多软盘和硬盘适配器使用 DMA，但是很少使用 DMA 校验功能。

表 18-4 内存和 I/O 传送类型

传送类型	源	目 的
读	内存	I/O
写	I/O	内存
校验	内存	无

澄清模糊认识

有关 DMA 的一个模糊观念是，数据在 DMA 和 I/O 之间传送时间使用相同的总线。通常称它们位内存总线，个别也称为 I/O 总线或通道。在系统中只有一条物理的数据和地址总线。

独立的控制线指示了地址是用于内存访问还是 I/O 访问。有四条线控制地址/数据线用于活动的内存读或写，还是用于 I/O 端口读或写。对指定的端口地址使用 CPU 的 IN 或 OUT 指令时，通常出现 I/O 端口的读或写。

与 CPU 不同，DMA 控制器能同时激活内存和 I/O 访问线。同时激活内存读线和 I/O 写线，可以从内存向适配卡传送一个字节。传送时，适配卡将在指定的 DMA 通道上获得一个确认信号。这时情况下，传送发生在内存和适配卡之间。

一个典型的I/O到内存的传送

图 18-1 显示了在初始化 DMA 和从适配卡的 I/O 直接向内存传送数据时事件发生的顺序。这通常称作一次 DMA 写，下面这个例子中使用 DMA 通道 3，没有同时执行其他 DMA 传送。

完整处理数据传送的顺序如下：

1. 设备驱动程序的初始化子程序对 DMA 编程，设置下列信息：
   写操作开始的内存基地址：低 16 位写入到 DMA 控制器，通道 3 中。高地址位写入到页面寄存器 3 中。传送的字节数减一后放入到 DMA 控制器，通道 3 中。传送模式选择为“命令模式”（其他选项下为单模式和块模式）。
   计数方向设置为递增（其他选项下选择递减）
   传送的类型是“写”（其他选项下是读和校验）
   更新 DMA 屏蔽位，允许 DMA 请求 3（DREQ3）
2. 通过声明 DMA 请求线 3（DREQ3）为高电平，适配卡来请求 DMA 服务。
3. DMA 控制器检验是否允许 DREQ3，然后声明保持请求线（HRQ）来请求 CPU 进入保持模式。
4. CPU 声明保持确认（HLDA），并进入总线保持状态。
5. 由 DMA 系统生成的地址传送给总线。激活内存写和 I/O 读控制线。激活 DMA 确认（DACK3）信号，于是适配卡知道正在传送数据。
6. 数据直接从适配器向内存传送，而不通过 DMA 控制器。在传送时，CPU 在 DMA 控制下插入一些普通的周期和总线保持周期。
7. 当 DMA 控制器的计数完成，或者，由于传送对适配器来说太快适配器降低 DREQ 线电平而暂时停止数据传送时，传送结束。
8. 结束时，DMA 控制器声明终端计数（TC）线而向适配卡发出信号，来通知传送结束。减活保持请求（HRQ）和 DMA 确认（DAKC3）线。
9. 适配器关闭请求，DREQ3。
10. CPU 继续普通的总线控制。

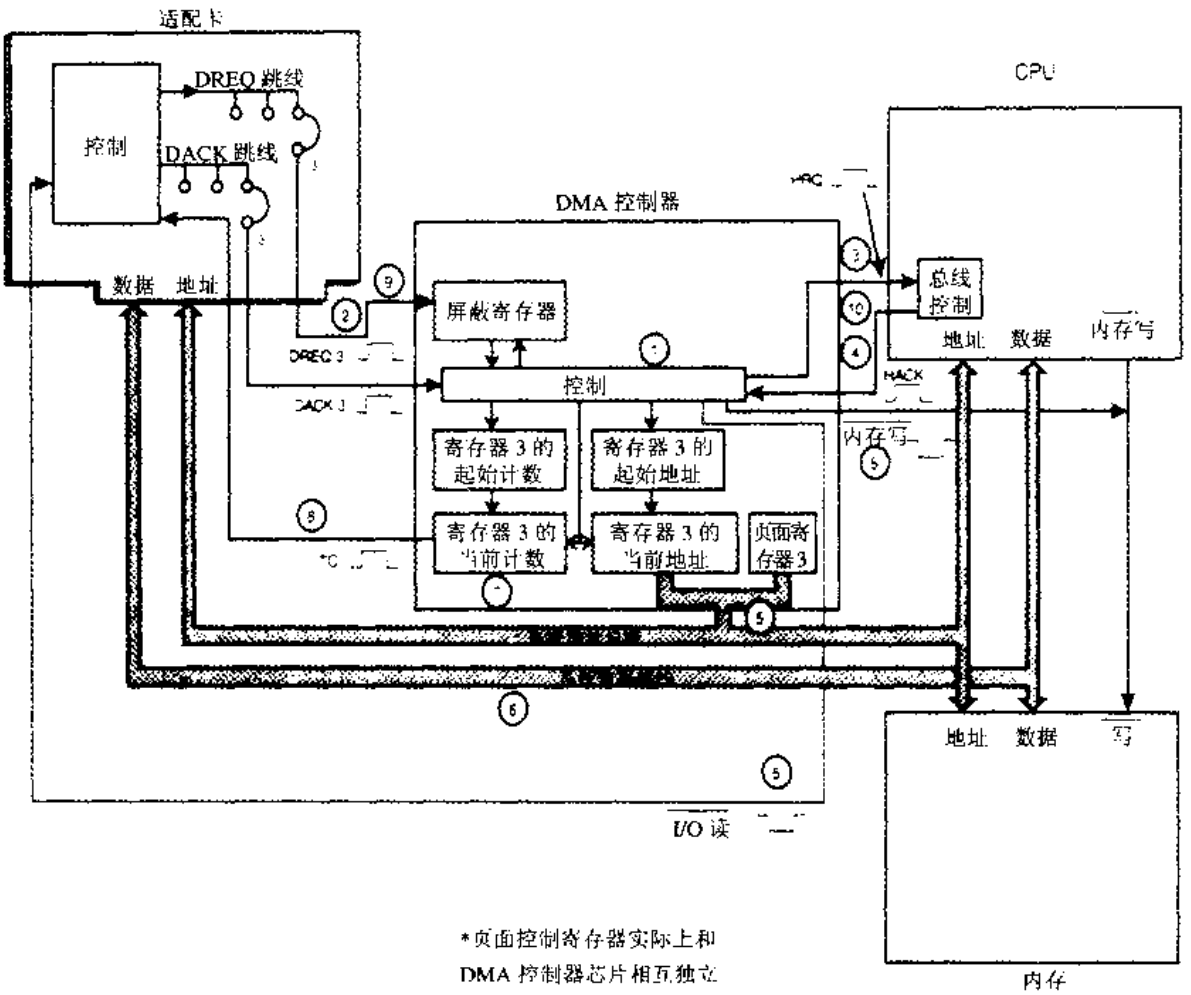


图 18-1 DMA Transfer Sequence

内存到内存 DMA

DMA 控制器能在软件的控制下使用两个 DMA 通道，从一个内存地址向另一个内存地址传送内存字节。因为这种特性比 80386 的串移动指令慢，所以很少使用该特性。该特性在每个传送周期内只能传送一个字节，而 80386 的 MOVSD 能够一次传送 4 个字节。如果正在执行内存到内存的传送，不能出现内存到 I/O 的传送。其他进一步的限制用处不大。

使用内存到内存的传送可能会产生奇怪的问题。几乎在所有的情况下，使用内存到内存传送的应用程序从来没有实现过。延迟可能会生成一个超时错误，或者其他难以预料的操作以及数据丢失。

只有通道 0 和 1 才能执行内存到内存的传送。在最初的 PC 上，DMA 通道用于 DRAM

刷新。这一点使内存到内存的传送非常棘手，因为在执行 DMA 传送时必须禁止刷新。长时间传送而禁止刷新，DRAM 可能会丢失其内容而造成系统冲突。一般情况下，最好不要使用内存到内存 DMA。

设置命令寄存器的第 0 位，来开放内存到内存的传送。在通道 0 基地址寄存器中装入内存源地址，而在通 1 基地址寄存器中装入内存目标地址。通道 1 的字计数寄存器中保存了要传送的字节数减 一。使用请求寄存器将 DREQ 的第 0 位置高来启动传送。

操作模式

DMA 提供了四种传送模式：单模式、块模式、命令模式、以及级联模式。另外，可以在前三种模式中使用自动初始化选项。激活通道的 DREQ 后就开始传送。使用端口 Bh 和 D6h 处的模式寄存器来为每个通道选择操作模式。

自动初始化

自动初始化提供了一种途径，通过向当前地址和当前计数寄存器装入基地址和计数寄存器，来重新开始一次传送。如果通道开放了自动初始化，并出现了终端计数（TC），则传送寄存器。字计数器从 0 到 FFFF 滚动时，会出现一次 TC。

单传送模式

正如其名所示，它将 DMA 设置为每次 DREQ 只传送单个字节或字。将当前的地址调整为下一个内存地址。当前的计数寄存器递减一次。如果它造成 TC 从 0 到 FFFF 滚动，则在开放自动初始化时会出现自动初始化。

如果 DREQ 线保持激活状态，DMA 会在单传送结束时释放总线。允许 CPU 一个总线周期。DMA 然后再次声明控制并执行下一个单传送。

块传送模式

DREQ 激活时开始传送。传送会一直继续下去，直到字计数器从 0 到 FFFF 滚动达到 TC 值。这种情况下，在通道 0~3 上一次可以传送 64K 以内的字节，或者在通道 5~7 一次传送 64K 以内的字。如果已经对自动初始化编程，那么在出现 TC 时的传送结尾，会自动初始化通道。如果 DREQ 仍处于活动状态，或者接着会激活 DREQ，那么将开始传送下一个块。

命令传送模

DREQ 激活时开始传送。传送会一直继续下去，达到 TC 值或 DREQ 进入不活动状态。这一点允许 I/O 设备调整传送。如果一个慢速 I/O 设备使用完了数据，适配器会停用其 DREQ 线。如果适配器准备好发送更多的数据，会重新声明 DREQ 并继续传送。如果已经对自动初始化编程，那么在出现 TC 时的传送结尾，会自动初始化通道。如果 DREQ 仍处于活动状态，或者接着会激活 DREQ，那么将开始传送下一个块。

级联模式

这种模式用于带有两个 DMA 控制器的系统上，例如 AT 系统和 EISA 系统。通道 4 用于硬件以级联这两个 DMA 控制器。初始化 DMA 控制器时，BIOS 将通道 4 设置为级联模式。永远也不要改变通道 4 的级联模式，否则将无法使用 DMA 通道 5、6 和 7。

MCA 系统的不同之处

微通道系统用自定义逻辑替代双重 8237 DMA 控制器来模拟相同的功能操作。另外，MCA DMA 允许通过端口 18h 和 1Ah 来访问特殊的扩展功能。

这些是 8237 DMA 控制器与 MCA 系统的主要差别。MCA 系统能够对每个通道的传送大小（字节或字）进行编程。大小仍依赖于相连接的硬件，但是却提供了另外的灵活性。

MCA 系统也支持向指定的端口传递数据。这种情况下，通过一个临时的寄存器给出 DMA 的这个通道的路径。由于每次传送需要两次总线访问，所以对指定的 I/O 端口的 DMA 传送比标准的 DMA 传送要慢。这种方法的优点是多个设备可以共用同一个 DMA 端口。目前只有很少的设备，如果存在，使用这种替代方法。

MCA 系统有一个仲裁系统来控制对多个 DMA 用户的总线访问。给竞争的设备分配不同的优先级来解决冲突。另外，主板预先分配给刷新和 NMI 两个最高优先级的总线仲裁。表 18-5 显示了缺省情况下的仲裁等级，FEh 代表最高优先级而 0Fh 代表最低优先级。DMA 通道 0~4 可以分配给任意从 0 到 Eh 的优先级。使用端口 18h 功能 80h 为通道 0、功能 84h 为通道 4 设置优先级。

表 18-5　MCA DMA 仲裁优先级

等　级	功　能
FEh	RAM 刷新
FFh	NMI
0	DMA 通道 0（缺省等级，但是可以改变）

续表

等　　级	功　　能
1	DMA 通道 1
2	DMA 通道 2
3	DMA 通道 3
4	DMA 通道 4（缺省等级，但是可以改变）
5	DMA 通道 5
6	MA 通道 6
7	DMA 通道 7
8~Eh	未使用
Fh	留作 CPU 用

EISA 系统的不同之处

EISA 系统完全支持两个 8237 等效 DMA 控制器以及所有相关的操作。为其他 AT 类型的机器所编写的 DMA 传送不必作任何软件修改就可在 EISA 机器上运行。EISA 加入了许多美妙的改进，包括到任何 32 位地址的 32 位宽 DMA 传送。相对于 AT 类型系统的标准 16 位宽 24 位地址限制，这是一个主要的扩展。另外，EISA 系统提供了一种独特的 DMA 脉冲串式，可以提供比标准 DMA 快 33 倍的传送速度。

除了几个关键的考虑之外，这种改进也是显著的。依赖这种独特 EISA 改进的代码当然只会在一个 EISA 机器上发挥作用。对于一个 EISA 适配卡，这可能是可接受的，但是在其他任何 AT 类型的卡上，EISA 扩展明显的增加了复杂性。另一个很重要的问题有关 CPU 的虚模式。这种情况下，DMA 所使用的物理内存地址通常与设备驱动程序所使用的虚地址有所不同。还没有清晰的规范来处理一台 EISA 机器上出现的这些情况。有关虚 DMA 服务的部分详细讨论了这个问题。不幸的是，还不清楚 VDS 规范是否会一直处理这些新的 EISA 特性。

周期定时模式

EISA 系统一个扩展特性提供了更快的传送速率。EISA 提供了许多 DMA 定时模式，被称为类型 A、B 以及 C。这些新式的定时模式一般和 AT 适配器并不兼容，但是 EISA 适配器可以利用这些新模式。这些新模式一个主要的优点是 EISA DMA 可以将较小的 I/O 数据大小翻译成内存系统的满 32 位宽数据。表 18-6 显示了一个 EISA 系统可以获得的典型的数据传送速率。

表 18-6 EISA DMA 传送速率

模　　式	I/O 数据宽	传送速率
AT 兼容型	8 位	1.0MB/sec
	16 位	2.0MB/sec
类型 A	8 位	1.3MB/sec
	16 位	2.6MB/sec
	32 位	5.3MB/sec
类型 B	8 位	2.0MB/sec
	16 位	4.0MB/sec
	32 位	8.0MB/sec
类型 C,脉冲 DMA	8 位	8.2MB/sec
	16 位	16.5MB/sec
	32 位	33.0MB/sec

这些定时模式、DMA 传送模式、数据大小以及传送类型组合在一起时有一些限制。AT 兼容型的定时模式只允许带有普通的地址的 8 位数据，或者带有移位地址的 16 位数据，后者在该表中没有显示。移位地址能够访问一个更大的地址范围。例如，如果所有的数据是字对齐模式的，则不需要字地址的最低位。通过将地址向上“移动”一位，则可以访问两倍大小的内存。允许的定时模式组合如表 18-7 所示。

表 18-7 允许的高级 EISA 组合

传送类型	DMA 模式	定时模式	地址选项	数据大小（位）
读/写/校验	单	A	普通	8
读/写/校验	单	A	移位	16
读/写	单	A	普通	16
读/写	单	A	普通	32
读/写/校验	单	B	普通	8
读/写/校验	单	B	移位	16
读/写	单	B	普通	16
读/写	单	B	普通	32
读/写/校验	命令	A	普通	8

续表

传送类型	DMA 模式	定时模式	地址选项	数据大小（位）
读/写/校验	命令	A	普通	16
读/写	命令	A	移位	16
读/写	命令	A	普通	32
读/写/校验	命令	B	普通	8
读/写/校验	命令	B	移位	16
读/写	命令	B	普通	16
读/写	命令	B	普通	32
读/写	命令	C	普通	8
写	命令	C	移位	16
读	命令	C	普通	16
读/写	块	C	普通	32
读/写/校验	块	A	普通	8
读/写/校验	块	A	移位	16
读/写	块	A	普通	16
读/写	块	A	普通	32
读/写/校验	块	B	普通	8
读/写/校验	块	B	移位	16
读/写	块	B	普通	16
读/写	块	B	普通	32
读/写	块	C	普通	8
写	块	C	移位	16
读	块	C	普通	16
读/写	块	C	普通	32

EISA FIFO 传送

EISA 系统提供一种机制来创建一个先进先出（FIFO）缓冲区，或者一个环状缓冲区。DMA 系统向缓冲区传送信息而软件从缓冲区读取最早的信息。由基地址寄存器定义缓冲区起点，由基础计数寄存器定义缓冲区的大小。一个 DMA 块传送完成时，即当前计数到达其限定值时，DMA 会自动初始化并继续在缓冲区的起点装入其他的数据。

同时，软件从缓冲区读取数据。软件可设置的停止寄存器指示下一个要读取的数据地址。软件应读取 DMA 的当前地址寄存器来阻止读取一个空的缓冲区。软件读取缓冲区后，它会将停止寄存器中的值更新为下一个接着尚未读取的地址。DMA 控制器使用这个地址来

阻止覆盖 CPU 尚未读取的数据。

如果 CPU 读取数据的速度不够快，DMA 可能会对一个与停止寄存器相同的地址传送，从而使缓冲区达到其满容量。DMA 控制器检测到这个问题并停止传送。接着自动禁止该通道。可能会出现一个 DMA 超载，并且软件会查看屏蔽状态寄存器来检查该超载条件。

图 18-2 显示了传送开始前和 DMA 开始后的一个典型 FIFO 配置。软件已经开始读取某些 DMA 传送数据。

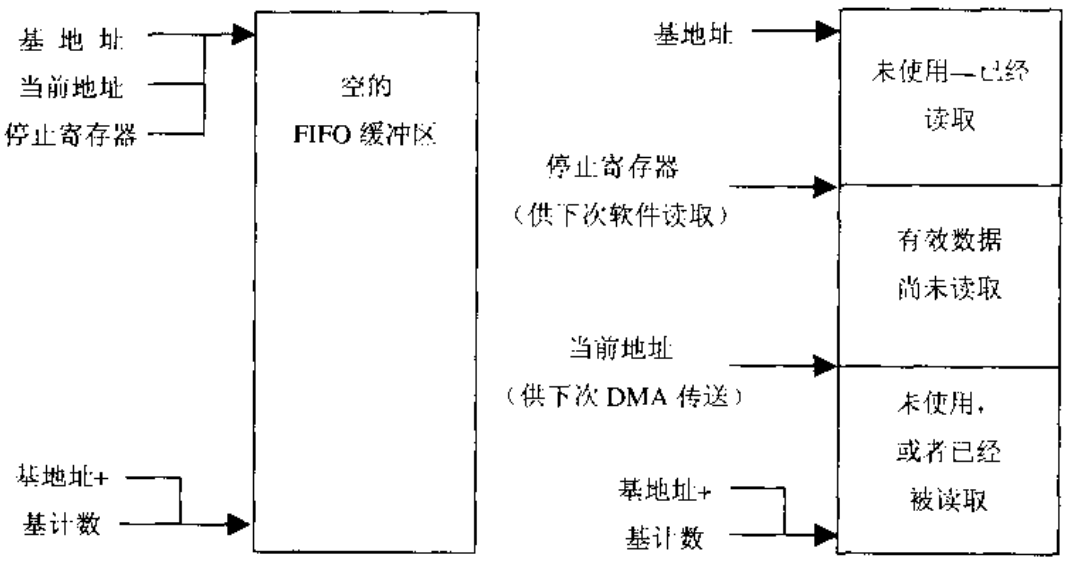


图 18-2 典型的 FIFO 缓冲区配置

EISA 链式传送

链式传送时，一个 EISA 系统可以在一个连续的传送中将数据传送到内存中的许多不同区域。当正在执行当前的传送时，通过让 DMA 控制器中断软件以获取最新的信息，来实现这项功能。软件装入一组新的信息，在传送完成时，会自动装入新信息和继续进行传送。

实现这一点所必须的操作有：

1. 为要传送的第一块装入基（base）地址和计数。
2. 设置通道的链式模式位来开放链接（端口 40Ah 或 4D4h，位 3&2=01）。它将基（base）和计数寄存器拷贝到当前的基和当前的计数寄存器中。
3. 为下一个块传送装入基地址和计数寄存器。
4. 设置通道的链式模式位来指示更新完成（端口 40Ah 或 4D4h 的位 3&2=11），并开始该进程。它启动第一个传送。
5. 第一个传送完成时，以前装入的基和计数会自动装入到当前地址和当前的计数寄存器中，接着继续新的传送。

这些操作也会在 IRQ13，75h 上生成一个 DMA 中断。中断处理程序必须检查链接中断寄存器（端口 40Ah）来确认中断由链接引起，并且检查来自哪个通道。知道这些后，处理程序然后可以装入下一组基地址和计数值。

6. 再次设置通道的链式模式位来指示更新完成（端口 40Ah 或 4D4h 的位 3&2=11）。它会清除中断，然后继续第 5 步的处理。

虚拟 DMA 服务（VDS）

现在生活复杂多了。DMA 内存传送发生在物理内存和一个 I/O 通道之间。内存管理软件、386 DOS 扩充器以及 Windows 增强模式都会创建一个虚拟内存映射，通常与物理内存有很大的不同。由于大多数带 386 或更好 CPU 的系统通常操作在虚拟模式（V86）下，这可能会是一个很大的问题。DMA 系统不知道虚拟地址，而软件设备驱动程序不知道物理系统！

解决这种困境用到了两个过程。第一个过程由虚拟内存管理程序处理，而第二个过程提供了一种称为虚拟 DMA 服务（VDS）的特殊接口。

Max8(386Max)、EMM386、Memory Commander、QEMM 以及其他的一些内存管理程序，通过陷入所有对 DMA 系统地 I/O 操作巧妙地解决了这个问题。在一个设备驱动程序写 DMA 芯片时，内存管理程序会截获该信息，如果必要，并将虚地址翻译成物理地址。然后内存管理程序将该新信息写入 DMA 控制器地寄存器中。

如果该物理地址在 DMA 控制器地范围之外，则虚拟内存管理程序将使用 DMA 控制器范围内的一个临时的缓冲区。每个虚拟内存管理程序为该用途分配一个 16K 的缓冲区，所以驱动程序在任何一次传送中最大只能传送 16K 字节，这一点很重要。如果试图传送 16K 以上的字节，并且要求缓冲区，则一般会出现一个一般保护性错误而暂停系统。用户将不得不重启系统，这一点常常令用户非常恼火！大多数上述内存管理程序提供了一种方法来扩展缓冲区，以便能处理最大 128K 的传送大小，但是这会浪费扩展内存，并且要求用户明确地更改内存管理程序地设置。

如果适配卡自带了 DMA 控制器，通常称作总线管理 DMA 控制器，则上面讨论的 DMA 翻译方法存在一个严重的问题。虚拟内存管理程序只能识别主板上的两个 DMA 控制器。幸运的是，在 IBM 对 PS/2 线加入 SCSI 支持之前就已经检测到了这个问题。IBM 开发了虚拟设备标准（VDS）解决方案。

VDS 是一组 INT 4Bh 功能，VDS 执行程序（内存管理程序）提供它们来与总线管理 DMA 的驱动程序进行物理和虚拟地址关系的通信。在物理地址不能匹配虚拟地址时，如果总线管理 DMA 的驱动程序不能使用 VDS，则它可能造成严重的系统稳定性问题，并挂起系统。反过来亦然：如果虚拟内存管理程序没有提供 VDS 服务，一次总线管理 DMA 传送可能会造成系统冲突。所有主要的内存管理程序——像 Max8(386Max)、EMM386、Memory Commander、QEMM、DOS 5.0 及以后 DOS 提供的 EMM386——支持 VDS。Windows 增

强模式虚拟 DMA 设备（VDMAD）VxD 也支持 VDS。

通过软件中断 4Bh 功能 ah=81h 提供 VDS 服务。如果激活了 VDS，则地址 40:7Bh 的位 5 将是 1。这时，总线管理 DMA 驱动程序必须使用 VDS 服务来执行相应的 DMA 传送。

参考完整的 VDS 规范说明，以了解用法及需要的操作。Microsoft 和 IBM 都提供了免费的 VDS 规范说明，表 18-8 显示了功能的归纳。

表 18-8 VDS 服务（中断 4Bh）归纳

AX=	描　　述
8102h	获取 VDS 版本
8103h	锁定 DMA 区域
8104h	开锁 DMA 区域
8105h	分散/收集锁定区域
8106h	分散/收集开锁区域
8107h	请求 DMA 缓冲区
8108h	释放 DMA 缓冲区
8109h	复制到 DMA 缓冲区
8109h	从 DMA 缓冲区复制
810Ah	禁止 DMA 翻译
810Bh	开放 DMA 翻译

典型用途

在 AT 以前的机器上，使用 DMA 通道来刷新动态内存。所有机器上的软盘适配器也使用一个 DMA 通道来从/向软盘控制器传送数据。其他可以使用 DMA 的包括老式的 SDLC（同步数据联结控制器）、许多网络适配卡、数字音响卡、扫描仪、以及 PC/XT 和许多 PS/2 系统上的硬盘适配器。

DMA BIOS 数据区

DMA 操作一般不使用任何 BIOS 数据，在使用虚拟 DMA 服务时是一个例外。会重新使用一个保留的字节用来保存 VDS 信息，通常这个字节为打印机号 4 的超时而保留。通常可以接受这一点，因为大多数的 AT+系统只使用了三个打印口。CPU 低于 80386 的系统没有提供 VDS。

安装一个内存管理程序时，必须挂起中断 4Bh。可能其他服务处理程序已经在使用该中断，所以必须检查向量来查看是否在使用该中断。如果在使用，则必须挂起该中断。如

果没有使用该中断，内存管理程序可以将中断向量直接指向 VDS 服务处理程序。如果位 3 是 0，那么没有截获中断。检查向量 4Bh（地址为 0:12Ch）的内容。如果内容是 0:0，或者段址是 E000 或 F000，则不允许挂起中断。如果 VDS_标志位 3 是 1，那么应串起中断 4Bh。

地址	描述	大小
40:7Bh	虚拟 DMA 标志	字节

VDS 服务提供者使用该标志。

位	7=x	未使用
	6=x	未使用
	5=0	不支持 VDS
	1	激活了 VDS
	4=x	未使用
	3=0	没有截获中断 4Bh
	1	要求链接
	2=x	未使用
	1=x	未使用
	0=x	未使用

警 告

在可能出现 DMA 请求时不要读和写地址以及计数寄存器。要做到这一点，可以禁止控制器（参看端口 8h/D0h 处的命令寄存器，位 2），或者你可以屏蔽该通道（参看端口 Ah/D4h）。对 DMA 控制器编程时应禁止中断，以避免中断处理程序中途改变操作。

传送的数目总是比装入到计数寄存器中的数目多一。为了传送 200 个字节或字，可以使用计数 199。计数 0 会执行一个字节或字的传送。

尽管 DMA 系统每次可以传送 128K 以内的字节，将每次传送限制在 16K 字节内不失为明智之举。参看虚拟 DMA 服务获取其他细节。

不要使用大块传送。如果 I/O 设备较快，则命令模式下的较大计数可能也会造成问题。较大块/命令模式传送时，其他的关键设备，像 RAM 刷新，就不能获得总线控制。在扩展模式类型 A、B 或 C 下，EISA 系统可以执行大块传送，而不用考虑这个问题。

自带 DMA 控制器的总线管理卡应检查是否激活了 VDS。如果激活 VDS，应该使用 VDS 服务来确保操作可靠。

在一个 AT 类型的系统上，虽然系统保留了 DMA 通道 2 和 4，但是却没有分配剩下的通道。不幸的是，没有方法来指出系统上的其他适配器正在使用哪一个 DMA 通道。像 I/O 端口一样，任何硬件/软件包应能够选择 DMA 通道。在显示开始信号和版权信息时，如果

BIOS、设备驱动程序或者 TSR 能显示正在使用哪一个 DMA 通道，那么将是非常有用的。虽然这不能自动解决冲突，但是用户知道了这些信息后，在选择其他 DMA 通道时更容易避免重叠。

例如，Novell 的 IPX 网络 TSR 就显示了软件使用哪一个 DMA 通道（如果使用）。IPX 软件能在 DOS 提示符下使用 I 命令行选项来重新显示当前重要的选择。而在另一个极端，那些短命的制造商热衷于使用户忙于查看手册、破坏系统、移走适配卡以及设置适当的跳线，仅仅只是为了知道使用了哪一个 DMA 通道。要求用户插入一个诊断软盘并运行一个特殊地程序来获取相关信息，同样是一种不好的方法。如果你隐藏了这些关键的信息，你的用户不会喜爱你。对于新式设备，请重点考虑支持可插既可用，来自动分配 DMA 通道。

不要完全依赖于诊断包来查找未使用的 DMA 通道。如果他们确实检测出了一个在使用的通道，那么信息是精确的。但是没有可保证的方法来检测主要的 DMA 用户。可能看起来没有使用通道，但是在将来又可能激活它。

代码例 18-1 DMA 传送设置

这个子程序例子显示了一个适配卡是如何将 DMA 控制器设置为 16 位 I/O 到内存的传送模式的。

```
;--------------------------------------------------------------------------------
;   DMA_READ
;           设置 DMA 控制器为 16 位适配器到内存
;           的传送模式，通道 5 被硬件 DMA 请求
;           5 激活
;
;           Notes:  1) 仅适用于 AT+系统（不用于 PC/XT）
;                   2) ds:si 必须被字线性化，即 si 位 0 = 0
;                   3) ds:si + dx 不可穿过一个 128KB 边界
;
;          Called with:     ds:si = 读入内存缓冲区
;                           dx    = 统计，计字数（不是字节）
;
DMA_read        proc    near
        cli
        mov     al, 00000101b           ; 设置通道 5 的遮蔽位
        out     0D4h, al                ;  禁用 DMA
        IODELAY
```

```
out         0D8h                ; 清除位 flip/flop
                                ;  (al 值不重要)
IODELAY
mov         al, 01001001h       ; 模式字节：单一模式，
                                ;  增加，或自动初始化，
                                ;  读通道 5
out         OD6h, al
IODELAY
                                ; 转换 es:si 到 32 位
                                ;  线性化，输出到 PMA-2
mov         ax, ds
mov         cl, 4
rol         ax, cl
push        ax
and         al, Ofh             ; al = 地址位为 a16 到 a23
push        ax                  ; 调用通道 5 页寄存器
IODELAY
pop         ax
and         ax, OFFFOh          ; 段* 16
add         ax, si              ; ax = 地址位 a0 到 a15
out         OC4h, al            ; 输出低地址字节
IODELAY
mov         al, ah
out         OC4h, al            ; 输出高地址字节
IODELAY
                                ; 送字数到 DMA-2
mov         ax, dx              ; 获取数据
dec         dx                  ; 调节，从 DMA 数
                                ;  大于 1 开始
out         OC6h, al            ; 输出低数位字节
IODELAY
mov         al, ah
out         OC6h, al            ; 输出高位字节
IODELAY
mov         al, 00000001b       ; 清除频道 5 遮蔽位
```

```
        out         OD4h, al            ;  启用 DMA
        IODELAY
                                        ; 现在，直到硬件
        sti
        ret
DMA_read        endp
```

端口归纳

下面是标准 DMA 端口的分布情况。AT+类型的机器和早期的 PC 机器所使用的端口有所不同，对不同的平台有多个入口。

端口	类型	功能	平台
00h	I/O	DMA-1 基和当前地址，通道 0	所有
01h	I/O	DMA-1 基和当前计数，通道 0	所有
02h	I/O	DMA-1 基和当前地址，通道 1	所有
03h	I/O	DMA-1 基和当前计数，通道 1	所有
04h	I/O	DMA-1 基和当前地址，通道 2	所有
05h	I/O	DMA-1 基和当前计数，通道 2	所有
06h	I/O	DMA-1 基和当前地址，通道 3	所有
07h	I/O	DMA-1 基和当前计数，通道 3	所有
08h	输入	DMA-1 状态寄存器	PC/XT/AT/EISA
08h	输出	DMA-1 命令寄存器	PC/XT/AT/EISA
09h	输出	DMA-1 请求寄存器	所有
0Ah	输出	DMA-1 屏蔽寄存器位	所有
0Bh	输出	DMA-1 模式寄存器	所有
0Ch	输出	DMA-1 清除字节触发	所有
0Dh	输入	DMA-1 临时寄存器	所有
0Dh	输出	DMA-1 主寄存器禁止	所有
0Eh	输出	DMA-1 清除屏蔽寄存器	所有
0Fh	输出	DMA-1 写所有寄存器位	所有
18h	I/O	DMA-1 扩展功能寄存器	MCA
1Ah	I/O	DMA-1 扩展功能执行	MCA
80h	I/O	DMA 页面寄存器通道 0	PC
81h	I/O	DMA 页面寄存器通道 2	AT+
81h	I/O	DMA 页面寄存器通道 1	PC
82h	I/O	DMA 页面寄存器通道 3	AT+
82h	I/O	DMA 页面寄存器通道 2	PC

83h	I/O	DMA 页面寄存器通道 1	AT+
83h	I/O	DMA 页面寄存器通道 3	PC
87h	I/O	DMA 页面寄存器通道 0	AT+
89h	I/O	DMA 页面寄存器通道 6	AT+
8Ah	I/O	DMA 页面寄存器通道 7	AT+
8Bh	I/O	DMA 页面寄存器通道 5	AT+
8Fh	I/O	DMA 页面寄存器通道 4（刷新）	AT+
90h	输入	DMA 仲裁状态	MCA
90h	输出	DMA 仲裁寄存器	MCA
C0h	I/O	DMA-2 基和当前地址，通道 4	AT+
C2h	I/O	DMA-2 基和当前计数，通道 4	AT+
C4h	I/O	DMA-2 基和当前地址，通道 5	AT+
C6h	I/O	DMA-2 基和当前计数，通道 5	AT+
C8h	I/O	DMA-2 基和当前地址，通道 6	AT+
CAh	I/O	DMA-2 基和当前计数，通道 6	AT+
CCh	I/O	DMA-2 基和当前地址，通道 7	AT+
CEh	I/O	DMA-2 基和当前计数，通道 7	AT+
D0h	输入	DMA-2 状态寄存器	AT+
D0h	输出	DMA-2 命令寄存器	AT/EISA
D2h	输出	DMA-2 请求寄存器	AT/EISA
D4h	输出	DMA-2 屏蔽寄存器位	AT
D6h	输出	DMA-2 模式寄存器	AT
D8h	输出	DMA-2 清除字节触发	AT
DAh	输入	DMA-2 临时寄存器	AT
DAh	输出	DMA-2 主寄存器禁止	AT
DCh	输出	DMA-2 清除屏蔽寄存器	AT
DEh	输出	DMA-2 写所有寄存器位	AT
401h	I/O	DMA-1 基和当前计数，通道 0，高	EISA
403h	I/O	DMA-1 基和当前计数，通道 1，高	EISA
405h	I/O	DMA-1 基和当前计数，通道 2，高	EISA
407h	I/O	DMA-1 基和当前计数，通道 3，高	EISA
40Ah	输入	DMA-1 链式中断状态	EISA
40Ah	输出	DMA-1 链式模式寄存器	EISA
40Bh	输出	DMA-1 扩展模式寄存器	EISA
40Ch	输入	DMA 链式终端计数动作状态	EISA
481h	I/O	DMA 页面寄存器通道 2，高	EISA
482h	I/O	DMA 页面寄存器通道 3，高	EISA
483h	I/O	DMA 页面寄存器通道 1，高	EISA

487h	I/O	DMA 页面寄存器通道 0，高	EISA
489h	I/O	DMA 页面寄存器通道 6，高	EISA
48Ah	I/O	DMA 页面寄存器通道 7，高	EISA
48Bh	I/O	DMA 页面寄存器通道 5，高	EISA
4C6h	I/O	DMA 基和当前计数，通道 5，高	EISA
4CAh	I/O	DMA 基和当前计数，通道 6，高	EISA
4CEh	I/O	DMA 基和当前计数，通道 7，高	EISA
4D4h	输入	DMA 链式状态	EISA
4D4h	输出	DMA-2 链式模式寄存器	EISA
4D6h	输出	DMA 扩展模式寄存器	EISA
4E0~4E2h	I/O	DMA 停止寄存器，通道 0	EISA
4E4~4E6h	I/O	DMA 停止寄存器，通道 1	EISA
4E8~4EAh	I/O	DMA 停止寄存器，通道 2	EISA
4EC~4EEh	I/O	DMA 停止寄存器，通道 3	EISA
4F4~4F6h	I/O	DMA 停止寄存器，通道 5	EISA
4F8~4FAh	I/O	DMA 停止寄存器，通道 6	EISA
4FC~4FEh	I/O	DMA 停止寄存器，通道 7	EISA

端口细节

端口	类型	描述	平台
00h	I/O	DMA-1 基和当前地址，通道 0	所有

在 AT/EISA/MCA 总线机器上没有分配通道 0 DMA，它被用于内存和 I/O 总线之间的高速传送。

在老式的 PC/XT 系统上，系统使用通道 0 DMA 来刷新 RAM。对系统时钟进行编程，来周期性的请求一次默默的传送。它会生成内存读周期。内存读用来保持动态内存的内容有效，正如 RAM 要求周期性读/写一样，否则会丢失内容。

输入（位 0-7）—读当前地址寄存器

读 DMA 通道 0 的 16 位当前地址寄存器。第一次 8 位读将返回该字的低部分，第二次读将返回该字的高部分。

当前地址寄存器保存着一次 DMA 传送中所用到的当前内存地址。每次 DMA 传送之后自递增或递减会它。

输出（位 0-7）—写基和当前地址寄存器

两个连续的 8 位 I/O 写操作向基地址寄存器和当前地址写。先写入该字的低 8 位，然后装入该字的高 8 位。基地址是只可写的。

基地址寄存器用于保存当前地址寄存器的原始值，在 DMA 传送期间不递增也不递减。

端口	类型	描述	平台
01h	I/O	DMA-1 基和当前计数，通道 0	所有

该端口控制 DMA 通道 0 的基和当前计数寄存器。参看端口 0h 以了解 DMA 通道 0 的用法。

输入（位 0-7）—读当前计数寄存器

读取 DMA 通道 0 的 10 位当前计数寄存器。前 8 位读将返回该字的低部分，后 8 位读将返回该字的高部分。

当前计数寄存器保存了加一后传送的剩余数目。一个 100h 的值表示还剩下 101h 个传送，每次传送后会递减该值。当该寄存器从 0 翻转到 FFFFh 时，传送结束。

输出（位 0-7）—写基（base）和当前计数寄存器

两个连续的 8 位 I/O 写操作向基计数寄存器和当前计数寄存器写。先写入该字的低 8 位，然后装入该字的高 8 位。基计数寄存器是只可写的。

基计数寄存器用于保存当前计数寄存器的原始值。

端口	类型	描述	平台
02h	I/O	DMA-1 基和当前地址，通道 1	所有

通道 1 DMA 用于 SDLC（同步数据连结控制），以实现内存和 I/O 总线之间的高速传送。如果没有使用 SDLC，则其他的适配卡可以使用通道 1。

输入（位 0-7）—读当前地址寄存器（参看端口 0）

输出（位 0-7）—写基（base）和当前地址寄存器（参看端口 0）

端口	类型	描述	平台
03h	I/O	DMA-1 基和当前计数，通道 1	所有

该端口控制 DMA 通道 1 的基和当前计数寄存器。参看端口 2h 了解 DMA 通道 1 的用法。

输入（位 0-7）—读当前计数寄存器（参看端口 1）

输出（位 0-7）—写基和当前计数寄存器（参看端口 1）

端口	类型	描述	平台
04h	I/O	DMA-1 基和当前地址，通道 2	所有

软盘服务使用通道 2 DMA 来实现内存和 I/O 总线之间的高速传送。

输入（位 0-7）—读当前地址寄存器（参看端口 0）

输出（位 0-7）—写基和当前地址寄存器（参看端口 0）

端口	类型	描述	平台
05h	I/O	DMA-1 基和当前计数，通道 2	所有

该端口控制 DMA 通道 2 的基和当前计数寄存器。参看端口 4h 了解 DMA 通道 2 的用法。

输入（位 0-7）—读当前计数寄存器（参看端口 1）

输出（位 0-7）—写基和当前计数寄存器（参看端口 1）

端口	类型	描述	平台
06h	I/O	DMA-1 基和当前地址，通道 3	所有

在 AT+系统上，DMA 通道是一个未分配的端口，用于内存和 I/O 总线之间的高速传送。PC/XT 的硬盘适配卡使用通道 3 在硬盘控制器和内存之间传送数据。

输入（位 0-7）—读当前地址寄存器（参看端口 0）

输出（位 0-7）—写基和当前地址寄存器（参看端口 0）

端口	类型	描述	平台
07h	I/O	DMA-1 基和当前计数，通道 3	所有

该端口控制 DMA 通道 3 的基和当前计数寄存器。参看端口 6h 了解 DMA 通道 2 的用法。

输入（位 0-7）—读当前计数寄存器（参看端口 1）

输出（位 0-7）—写基和当前计数寄存器（参看端口 1）

端口	类型	描述	平台
08h	输入	DMA-1 状态寄存器	所有

DMA 状态寄存器保存标志（位 0~3）指示了每个通道何时达到计时终点（传送完成）。读该计数器时，清除低四位。

该状态寄存器也包含了每个通道的挂起 DMA 请求标志。声明所期望 DMA 通道请求线时会出现 DMA 请求。

输入（位 0~7）

位 7r=1 DMA 通道 3 请求

6r=1	DMA 通道 2 请求
5r=1	DMA 通道 1 请求
4r=1	DMA 通道 0 请求
3r=1	DMA 通道 3 达到计数终点
2r=1	DMA 通道 2 达到计数终点
1r=1	DMA 通道 1 达到计数终点
0r=1	DMA 通道 0 达到计数终点

端口	类型	描述	平台
08h	输出	DMA-1 命令寄存器	PC/XT/AT/EISA

这个 8 位寄存器控制 DMA 的操作。硬件重启或主禁止指令（端口 0Dh）可以清除该寄存器。PC 使用的命令值是 0。通常，所有的位是 0，除了位 2 常常设置为 1 以便在写 DMA 寄存器时禁止控制器。该寄存器是只可写的，MCA 系统不支持该寄存器。

输出（位 0~7）

位		
	7w = 0	DACK 传感活动低（缺省值）
	1	DACK 传感活动高
	6w = 0	DREQ 传感活动低（缺省值）
	1	DREQ 传感活动高
	5w = 0	迟写选择
	1	扩展写选择
	4w = 0	固定优先级（缺省值）
	1	循环优先级
	3w = 0	普通定时
	1	压缩定时
	x	如果位 1=0（缺省）
	2w = 0	开放控制器
	1	禁止控制器
	1w = 0	禁止保持通道 0 地址
	1	开放保持通道 0 地址
	x	如果位 0=0（缺省）
	0w = 0	禁止内存到内存的传送（缺省）
	1	开放内存到内存的传送

端口	类型	描述	平台
09h	输出	DMA-1 请求寄存器	PC/XT/AT/EISA

除了声明一条硬件请求线可以启动 DMA 服务请求外，软件也可以启动 DMA 请求。请求寄存器可用于设置和清除任何通道的请求位。DMA 控制器使用这些功能时，不需要位于

块模式下，而在模式寄存器（端口 Bh）中需要这样设置。该寄存器是只可写的，MCA 系统不支持该寄存器。

输出（位 0~7）

位		
	7w = x	未使用
	6w = x	未使用
	5w = x	未使用
	4w = x	未使用
	3w = x	未使用
	2w = 0	清除请求位
	1	设置请求位
	1w = x 0w = x	选择通道号

位 1	位 0
0	0 = 通道 0
0	1 = 通道 1
1	0 = 通道 2
1	1 = 通道 3

端口	类型	描述	平台
0Ah	输出	DMA-1 屏蔽寄存器位	所有

该屏蔽寄存器用来禁止或开放单个引入的请求。将屏蔽位设置为开会禁止选中的通道，硬件重启会设置所有的屏蔽位来禁止所有的通道。该寄存器是只可写的。

输出（位 0~7）

位		
	7w = x	未使用
	6w = x	未使用
	5w = x	未使用
	4w = x	未使用
	3w = x	未使用
	2w = 0	清除屏蔽位
	1	设置屏蔽位
	1w = x 0w = x	选择通道号

位 1	位 0
0	0 = 通道 0
0	1 = 通道 1
0	0 = 通道 2
0	1 = 通道 3

端口	类型	描述	平台
0Bh	输出	DMA-1 模式寄存器	所有

该模式寄存器指示每个 DMA 通道 0~3 的操作的模式。每个通道都有一个独立的 6 位模式寄存器，通过模式寄存器端口装入每个寄存器。

输出（位 0~7）

位

7w = x, 6w = x — 模式类型选择

位 7	位 6	
1	0	= 选择命令模式
0	1	= 选择单模式
1	0	= 选择块模式
0	0	= 选择级联模式（无效）

5w = 1 — 选择地址递增

1 — 选择地址递减（在 EISA 32 位和 EISA 普通地址 16 位传送模式下无效）

4w = 0 — 禁止自动初始化

1 — 开放自动初始化

3w = x, 2w = x — 传送类型

位 3	位 2	
0	0	= 校验
0	1	= 写
0	0	= 读
0	1	= 无效
x	x	= 如果位于级联模式（位 6 和 7）

1w = x, 0w = x — 选择通道号

位 1	位 0	
0	0	= 通道 0
0	1	= 通道 1
1	0	= 通道 2
0	1	= 通道 3

端口	类型	描述	平台
0Ch	输出	DMA-1 清字节触发器	所有

任何向该端口的输出值都会清除 DMA 控制器 1 内的内部首/尾触发器。在向需要连续两次 8 位访问的 16 位端口执行 8 位读或写之前，可以执行该操作来完成该字的传送。在清除了触发器后，读或写低字节，然后读或写高字节，来访问 16 位 DMA 寄存器。只能清除该触发器，而不可读取该触发器。

端口	类型	描述	平台
0Dh	输入	DMA-1 临时寄存器	所有

该临时寄存器保存了在内存到内存数据传送期间的数据。传送完成后，临时寄存器保存了最后一次的传送数据。在 DMA 控制器没有执行一次 DMA 传送时，可以读取该临时寄存器。重启时会清除该寄存器，它是只可读的。

端口	类型	描述	平台
0Dh	输出	DMA-1 主禁止	所有

向这个端口写入任何值都会重启控制器 1。该命令执行一次与硬件重启相同的操作。设置屏蔽位（禁止通道 0~3）。清除所有的命令、临时以及字节触发器。该寄存器是只可写的。

端口	类型	描述	平台
0Eh	输出	DMA-1 清除屏蔽寄存器	所有

向该端口写入任何值都会清除屏蔽寄存器。清除屏蔽寄存器会开放全部四个通道来接受 DMA 请求。该寄存器是只可写。

端口	类型	描述	平台
0Fh	输出	DMA-1 写所有屏蔽位	所有

该屏蔽寄存器用来禁止或开放单个引入的请求。将一个屏蔽位设置为开会禁止选中的通道。一次硬件重启会设置所有的屏蔽位而禁止所有的通道。该寄存器是只可写的。

输出（位 0~7）

位		
	7w = x	未使用
	6w = x	未使用
	5w = x	未使用
	4w = x	未使用
	3w = 0	通道 3 清除屏蔽位（开放）
	1	通道 3 开放屏蔽位
	2w = 0	通道 2 清除屏蔽位（开放）
	1	通道 2 开放屏蔽位
	1w = 0	通道 1 清除屏蔽位（开放）
	1	通道 1 开放屏蔽位
	0w = 0	通道 0 清除屏蔽位（开放）
	1	通道 0 开放屏蔽位

端口	类型	描述	平台
18h	输出	DMA 扩展功能寄存器	MCA

MCA 机器用自定义的逻辑替代一对 8237 DMA 控制器来模拟 8237 的基本功能和添加许多扩展功能。该寄存器支持 DMA 寄存器的替代访问和扩展。

向该端口输出会启动一个指定 DMA 通道的扩展命令。取决于具体的命令，该命令字节可能接着向端口 1Ah 执行一到三个读或写操作来完成该操作。

输出（位 0~7）

位			
7w = x 6w = x 5w = x 4w = x	扩展命令 0~F		
3w = 0	未使用		
2w = x 1w = x 0w = x	选择通道号		
	位 2	位 1	位 0
	0	0	0 = 通道 0
	0	0	1 = 通道 1
	0	1	0 = 通道 2
	0	1	1 = 通道 3
	1	0	0 = 通道 4
	1	0	1 = 通道 5
	1	1	0 = 通道 6
	1	1	1 = 通道 7

扩展命令归纳

下面命令号中 x 代表命令的通道号，从 0~7。

号	端口 18h 命令
0xh	I/O 地址寄存器
2xh	基和当前地址写
3xh	基地址读
4xh	基和当前计数写
5xh	基计数读
6xh	状态寄存器读
7xh	扩展模式寄存器
8xh	仲裁等级—通道 0
8xh	仲裁等级—通道 4

9xh	屏蔽寄存器禁止通道
Axh	屏蔽寄存器开放通道
Dxh	主禁止

扩展命令细节

命令	描述	端口
0xh	I/O 地址寄存器	18h

该 I/O 地址寄存器支持为一次 DMA 传送设置 I/O 端口地址。每个通道都有一个 16 位 I/O 地址寄存器。如果设置为 0000，则该通道的 DMA 操作与 AT/EISA 上发现的标准的 8237 DMA 相同。非零值表示指定的端口出现了 DMA 操作。

使用一个指定的端口地址时，传送的每个字节/字必须保存在 DMA 通道的临时寄存器中。使用这种操作模式比标准 DMA 要慢，因为每次传送需要两次总线访问。最大的优点是可避免与其他 DMA 用户的冲突。

执行下面的步骤来读或写一个与 DMA 通道 x 相关的指定 I/O 地址寄存器。所有的读和写必须是单字节的。

步	读地址寄存器
1	输出到端口 18h，值 0xh
2	从端口 1Ah 输入，a0~a7
3	从端口 1Ah 输入，a8~a15

步	写地址寄存器
1	输出到端口 18h，值 0xh
2	输出到端口 1Ah，a0~a7
3	输出到端口 1Ah，a8~a15

命令	描述	端口
2xh	基和当前地址写	18h

该命令向基和当前地址寄存器中装入一个 24 位值。它的功能类似于标准的 16 位基和当前地址写，由端口 0、2、4、6、C0h、C8h 或 CCh 指定，接着向开始于端口 81h 的页面寄存器执行一次写操作。

写一个与 DMA 通道 x 相关的指定基和当前地址寄存器时，会出现下面的步骤。所有的写必须是单字节的。8 位和 16 位传送的地址位有所不同，如下所示。

步	8 位 DMA 传送
1	输出到端口 18h，值 2xh（x = 通道 0~7）

2	输出到端口 1Ah，a0~a7
3	输出到端口 1Ah，a8~a15
4	输出到端口 1Ah，a16~a23
步	**16 位 DMA 传送**
1	输出到端口 18h，值 2xh（x = 通道 0~7）
2	输出到端口 1Ah，a1~a8
3	输出到端口 1Ah，a9~a16
4	输出到端口 1Ah，a17~a23

命令	描述	端口
3xh	基地址读	18h

该命令读取基地址寄存器。在 8237DMA 控制器上没有等价的命令。

执行下面的步骤来读取一个与 DMA 通道 x 相关的指定 24 位基地址寄存器。所有的读和写必须是单字节的。8 位和 16 位传送的地址位有所不同，如下所示。

步	**8 位 DMA 传送**
1	输出到端口 18h，值 3xh（x = 通道 0~7）
2	从端口 1Ah 输入，a0~a7
3	从端口 1Ah 输入，a8~a15
4	从端口 1Ah 输入，a16~a23

步	**16 位 DMA 传送**
1	输出到端口 18h，值 2xh（x = 通道 0~7）
2	从端口 1Ah 输入，a1~a8
3	从端口 1Ah 输入，a9~a16
4	从端口 1Ah 输入，a17~a23

命令	描述	端口
4xh	基和当前计数写	18h

该命令向基和当前计数寄存器中写入一个 16 位的值。它的功能类似于由端口 1、3、5、7、C2、C6、CA 或 Ceh 指定的标准 16 位基础和当前计数写操作。

执行下面的步骤来写一个与 DMA 通道 x 相关的指定的基和当前计数寄存器。所有的写必须是单字节的。

步	**写基地址和当前计数**
1	输出到端口 18h，值 4xh（x = 通道 0~7）

2	输出到端口 1Ah，计数位 0~7
3	输出到端口 1Ah，计数位 8~15

命令	描述	端口
5xh	基计数读	18h

该命令读取基计数寄存器。在 8237DMA 控制器上没有等价的命令。

执行下面的步骤来读取一个与 DMA 通道 x 相关的指定 16 位基计数寄存器。所有的读或写必须是单字节的。8 位和 16 位传送的地址位有所不同，如下所示。

步	读基计数
1	输出到端口 18h，值 5xh（x = 通道 0~7）
2	从端口 1Ah 输入，位 0~7
3	从端口 1Ah 输入，位 8~15

命令	描述	端口
6xh	状态寄存器读	18h

该 DMA 状态寄存器保存了那些到达终端计数（传送完成）的通道的标志。读取该寄存器时，清除低四位。

该状态寄存器也包含了每个通道上的 DMA 请求挂起标志。声明期望的 DMA 通道请求线会出现 DMA 请求。

执行下面的步骤来读取两个状态寄存器中的一个。当指定一个 DMA 通道“0~3”时，会返回通道 0 到 3 所有的信息，这一点和使用通道 8 来读取状态相似。当指定一个 DMA 通道，4~7，时，会返回通道 4 到 7 所有的信息，这一点和使用通道 D0h 来读取状态相似。所有的读必须是单字节的。

步	通道 0~3 的状态	
1	输出到端口 18h，值 50h~53h	
2	从端口 1Ah 输入，状态	
	位 7 r = 1	DMA 通道 3 请求
	6 r = 1	DMA 通道 2 请求
	5 r = 1	DMA 通道 1 请求
	4 r = 1	DMA 通道 0 请求
	3 r = 1	DMA 通道 3 达到计数终点
	2 r = 1	DMA 通道 2 达到计数终点
	1 r = 1	DMA 通道 1 达到计数终点
	0 r = 1	DMA 通道 0 达到计数终点

步	通道4~7的状态	
1	输出到端口18h，值54h~57h	
2	从端口1Ah输入，状态	
	位 7 r = 1	DMA通道7请求
	6 r = 1	DMA通道6请求
	5 r = 1	DMA通道5请求
	4 r = 1	DMA通道4请求
	3 r = 1	DMA通道7达到计数终点
	2 r = 1	DMA通道6达到计数终点
	1 r = 1	DMA通道5达到计数终点
	0 r = 1	DMA通道4达到计数终点

命令	描述	端口
7xh	扩展模式寄存器	18h

MCA系统为每个通道提供了一个扩展模式寄存器。该扩展模式寄存器同8237DMA控制器上的模式寄存器有些相似。对标准的寄存器编程时，MCA会重组信息并将它放入到相对应的扩展模式寄存器中。

执行下面的步骤来读或写与DM通道x相关的指定扩展寄存器。所有的读和写必须是单字节的。

步	读模式寄存器
1	输出到端口18h，值7xh（x=通道0~7）
2	从端口1Ah输入，模式字节

步	写模式寄存器
1	输出到端口18h，值7xh（x=通道0~7）
2	输出到端口1Ah，模式字节

扩展模式字节：

位		
	7 r/w = 0	未使用
	6 r/w = 0	8位传送
	1	16位传送
	5 r/w = 0	未使用
	4 r/w = 0	未使用
	3 r/w = 0	读内存传送
	1	写内存传送
	2 r/w = 0	读校验操作
	1	数据传送操作

1 r/w = 0		未使用
0 r/w = 0		无 I/O 地址（地址 = 0）
1		使用指定的 I/O 地址

命令	描述	端口
80h	仲裁等级—通道 0	18h

该命令为虚拟 DMA 操作设置通道 0 的仲裁等级。在 8237 上没有等价的功能。通道 1、2、3、5、6、或 7 没有仲裁等级寄存器。

步	读仲裁等级
1	输出到端口 18h，值 80h
2	从端口 1Ah 输入，等级

步	写仲裁等级
1	输出到端口 18h，值 80h
2	输出到端口 1Ah，等级

仲裁等级字节：

位		
	7 r/w = 0	未使用
	6 r/w = 0	未使用
	5 r/w = 0	未使用
	4 r/w = 0	未使用
	3 r/w = x	仲裁等级 0 ~ 14
	2 r/w = x	
	1 r/w = x	
	0 r/w = x	

命令	描述	端口
84h	仲裁等级—通道 4	18h

该命令为虚拟 DMA 操作设置通道 4 的仲裁等级。在 8237 上没有等价的功能。通道 1、2、3、5、6、或 7 没有仲裁等级寄存器。

步	读仲裁等级
1	输出到端口 18h，值 84h
2	从端口 1Ah 输入，等级

步	写仲裁等级
1	输出到端口 18h，值 84h

2 输出到端口 1Ah，等级

仲裁等级字节：

位	7 r/w = 0	未使用
	6 r/w = 0	未使用
	5 r/w = 0	未使用
	4 r/w = 0	未使用
	3 r/w = x	仲裁等级 0 ~ 14
	2 r/w = x	
	1 r/w = x	
	0 r/w = x	

命令	描述	端口
9xh	屏蔽寄存器禁止通道	18h

该屏蔽寄存器用来开放或禁止单个的引入请求。该命令重设一个指定的屏蔽位来禁止选中的通道。8237 DMA 控制器上的端口 Ah 和 D4h 提供了相似的功能。

命令	描述	端口
Axh	屏蔽寄存器开放通道	18h

该屏蔽寄存器用来开放或禁止单个的引入请求。该命令重设一个指定的屏蔽位来开放选中的通道。8237 DMA 控制器上的端口 Ah 和 D4h 提供了相似的功能。

命令	描述	端口
Dxh	主禁止	18h

该命令的效果和硬件重启相同。设置所有的屏蔽寄存器并禁止所有的通道，清除所有的命令、状态、请求、临时、以及字节触发器。8237 DMA 控制器上的端口 Dh 和 DAh 提供了相似的功能。

使用任何命令 D0h~D3h 都会禁止通道 0~3，使用任何命令 D4h~D7h 都会禁止通道 4~7。

端口	类型	描述	平台
1Ah	I/O	DMA 扩展功能执行	MCA

基于前面从端口 18h 设置的命令，使用这个端口来读和写字节。参考端口 18h 以获取细节。

端口	类型	描述	平台
80h	I/O	DMA 页面寄存器通道 0	PC/XT

该寄存器为 DMA 向内存传送的通道 0 的地址位 A16~A19。低 16 位地址由 DMA 控制器生成。

端口	类型	描述	平台
81h	I/O	DMA 页面寄存器通道 2	AT+

该寄存器为 DMA 向内存传送的通道 2 的地址位 A16~A23，低 16 位地址由 DMA 控制器生成。它支持前 16MB 内存的 DMA 传送。

端口	类型	描述	平台
81h	I/O	DMA 页面寄存器通道 1	PC/XT

该寄存器为 DMA 向内存传送的通道 1 的地址位 A16~A19。低 16 位地址由 DMA 控制器生成。

端口	类型	描述	平台
82h	I/O	DMA 页面寄存器通道 3	AT+

该寄存器为 DMA 向内存传送的通道 3 的地址位 A16~A23。低 16 位地址由 DMA 控制器生成。它支持前 16MB 内存的 DMA 传送。

端口	类型	描述	平台
82h	I/O	DMA 页面寄存器通道 2	PC/XT

该寄存器为 DMA 向内存传送的通道 2 的地址位 A16~A19。低 16 位地址由 DMA 控制器生成。

端口	类型	描述	平台
83h	I/O	DMA 页面寄存器通道 1	AT+

该寄存器为 DMA 向内存传送的通道 1 的地址位 A16~A23。低 16 位地址由 DMA 控制器生成。它支持前 16MB 内存的 DMA 传送。

端口	类型	描述	平台
83h	I/O	DMA 页面寄存器通道 3	PC/XT

该寄存器为 DMA 向内存传送的通道 3 的地址位 A16~A19。低 16 位地址由 DMA 控制器生成。

端口	类型	描述	平台
87h	I/O	DMA 页面寄存器通道 0	AT+

该寄存器为 DMA 向内存传送的通道 0 的地址位 A16~A23。低 16 位地址由 DMA 控制器生成。它支持前 16MB 内存的 DMA 传送。

端口	类型	描述	平台
89h	I/O	DMA 页面寄存器通道 6	AT+

该寄存器为 DMA 向内存传送的通道 6 的地址位 A17~A23。低 16 位地址由 DMA 控制器生成。它支持前 16MB 内存的 DMA 传送。

端口	类型	描述	平台
8Ah	I/O	DMA 页面寄存器通道 7	AT+

该寄存器为 DMA 向内存传送的通道 7 的地址位 A17~A23。低 16 位地址由 DMA 控制器生成。它支持前 16MB 内存的 DMA 传送。

端口	类型	描述	平台
8Bh	I/O	DMA 页面寄存器通道 5	AT+

该寄存器为 DMA 向内存传送的通道 5 的地址位 A17~A23。低 16 位地址由 DMA 控制器生成。它支持前 16MB 内存的 DMA 传送。

端口	类型	描述	平台
8Fh	I/O	DMA 页面寄存器通道 4（刷新）	AT+

由于通道 4 用于从 DMA 控制器 1 到 DMA 控制器 2 的级联，所以该寄存器在某些系统上作为刷新寄存器的一部分用于动态 RAM 的刷新。

端口	类型	描述	平台
90h	输入	DMA 仲裁状态	AT+

该寄存器用于读取仲裁状态。

输入（位 0~7）

位 7 r = 1 在仲裁期间开放 CPU 周期

6 r = 1	出现 NMI，因此屏蔽了仲裁
5 r = 1	出现总线超时。清除位 6
4 r = 1	未使用
3 r = x	返回上次许可的仲裁等级（0~15）
2 r = x	
1 r = x	
0 r = x	

端口	类型	描述	平台
90h	输出	DMA 仲裁寄存器	AT+

该寄存器用于改变仲裁信息。

输出（位 0~7）

位		
7 w = 0		在仲裁期间禁止 CPU 周期
1		在仲裁期间开放 CPU 周期
6 w = 0		普通操作（缺省）
1		开放仲裁状态，CPU 控制通道（仅用于诊断）
5 w = 0		普通仲裁周期（最小 300 uS，缺省）
1		扩展仲裁周期（最小 600 uS）
4 w = 0		未使用
3 w = 0		未使用
2 w = 0		未使用
1 w = 0		未使用
0 w = 0		未使用

端口	类型	描述	平台
C0h	I/O	DMA-2 基和当前地址，通道 4	AT+

通道 4 DMA 用于 DMA 控制器 1 的级联功能。通道 4 不能用于其他方面。

端口	类型	描述	平台
C2h	I/O	DMA-2 基和当前计数，通道 4	AT+

通道 4 DMA 用于 DMA 控制器 1 的级联功能。通道 4 不能用于其他方面。

端口	类型	描述	平台
C4h	I/O	DMA-2 基和当前地址，通道 5	AT+

在AT和EISA系统上，DMA通道5尚未分配，用于内存和I/O总线之间的高速传送。在PS/2系统上，通道5用于硬盘DMA操作。

输入（位0~7）—读当前地址寄存器（参看端口0）
输出（位0~7）—写基和当前地址寄存器（参看端口0）

端口	类型	描述	平台
C6h	I/O	DMA-2 基和当前计数，通道5	AT+

在AT和EISA系统上，DMA通道5尚未分配，用于内存和I/O总线之间的高速传送。在PS/2系统上，通道5用于硬盘DMA操作。

输入（位0~7）—读当前计数寄存器（参看端口1）
输出（位0~7）—写基和当前地址寄存器（参看端口1）

端口	类型	描述	平台
C8h	I/O	DMA-2 基和当前地址，通道6	AT+

DMA通道6尚未分配，用于内存和I/O总线之间的高速传送。

输入（位0~7）—读当前地址寄存器（参看端口0）
输出（位0~7）—写基和当前地址寄存器（参看端口0）

端口	类型	描述	平台
CAh	I/O	DMA-2 基和当前计数，通道6	AT+

DMA通道6尚未分配，用于内存和I/O总线之间的高速传送。

输入（位0~7）—读当前计数寄存器（参看端口1）
输出（位0~7）—写基和当前地址寄存器（参看端口1）

端口	类型	描述	平台
CCh	I/O	DMA-2 基和当前地址，通道7	AT+

DMA通道7尚未分配，用于内存和I/O总线之间的高速传送。

输入（位0~7）—读当前地址寄存器（参看端口0）
输出（位0~7）—写基和当前地址寄存器（参看端口0）

端口	类型	描述	平台
CEh	I/O	DMA-2 基和当前计数，通道7	AT+

DMA 通道 7 尚未分配，用于内存和 I/O 总线之间的高速传送。

输入（位 0~7）—读当前计数寄存器（参看端口 1）
输出（位 0~7）—写基和当前地址寄存器（参看端口 1）

端口	类型	描述	平台
D0h	输入	DMA-2 状态寄存器	AT+

DMA 状态寄存器保存标志（位 0~3）指示了每个通道何时达到计时终点（传送完成）。读该计数器时，清除低四位。

该状态寄存器也包含了每个通道的挂起 DMA 请求标志。声明所期望 DMA 通道请求线时会出现 DMA 请求。

输入（位 0~7）

位		
	7r=1	DMA 通道 7 请求
	6r=1	DMA 通道 6 请求
	5r=1	DMA 通道 5 请求
	4r=1	DMA 通道 4 用于级联
	3r=1	DMA 通道 7 达到计数终点
	2r=1	DMA 通道 6 达到计数终点
	1r=1	DMA 通道 5 达到计数终点
	0r=1	DMA 通道 4 用于级联

端口	类型	描述	平台
D0h	输出	DMA-2 命令寄存器	AT/EISA

这个 8 位寄存器控制 DMA 的操作。硬件重启或主禁止指令（端口 0Dh）可以清除该寄存器。PC 使用的命令值是 0。通常，所有的位是 0，除了位 2 常常设置为 1 以便在写 DMA 寄存器时禁止控制器。该寄存器是只可写的，MCA 系统不支持该寄存器。

输出（位 0~7）

位		
	7w = 0	DACK 传感活动低（缺省值）
	1	DACK 传感活动高
	6w = 0	DREQ 传感活动低（缺省值）
	1	DREQ 传感活动高
	5w = 0	迟写选择
	1	扩展写选择
	4w = 0	固定优先级（缺省值）
	1	循环优先级

3w = 0	普通定时
1	压缩定时
x	如果位 1=0（缺省）
2w = 0	开放控制器
1	禁止控制器
1w = 0	禁止保持通道 0 地址
1	开放保持通道 0 地址
x	如果位 0=0（缺省）
0w = 0	禁止内存到内存的传送（缺省）
1	开放内存到内存的传送

端口	类型	描述	平台
D2h	输出	DMA-2 请求寄存器	AT/EISA

除了声明一条硬件请求线可以启动 DMA 服务请求外，软件也可以启动 DMA 请求。请求寄存器可用于设置和清除任何通道的请求位。DMA 控制器使用这些功能时，不需要位于块模式下，而在模式寄存器（端口 D6h）中需要这样设置。该寄存器是只可写的，MCA 系统不支持该寄存器。

输出（位 0~7）

位		
	7w = x	未使用
	6w = x	未使用
	5w = x	未使用
	4w = x	未使用
	3w = x	
	2w = 0	未使用
	1	清除请求位
	1w = x	设置请求位
	0w = x	选择通道号

位 1	位 0
0	0 = 通道 4
0	1 = 通道 5
1	0 = 通道 6
1	1 = 通道 7

端口	类型	描述	平台
D4h	输出	DMA-2 屏蔽寄存器位	AT+

该屏蔽寄存器用来禁止或开放单个引入的请求。将屏蔽位设置为开会禁止选中的通道。硬件重启会设置所有的屏蔽位来禁止所有的通道。该寄存器是只可写的。

输出（位 0~7）

位		
7w = x	未使用	
6w = x	未使用	
5w = x	未使用	
4w = x	未使用	
3w = x	未使用	
2w = 0	清除屏蔽位	
1	设置屏蔽位	
1w = x	选择通道号	
0w = x	位 1	位 z0
	0	0 = 通道 4
	0	1 = 通道 5
	1	0 = 通道 6
	1	1 = 通道 7

端口	类型	描述	平台
D6h	输出	DMA-2 模式寄存器位	AT+

该模式寄存器指示每个 DMA 通道 4~7 的操作的模式。每个通道都有一个独立的 6 位模式寄存器，通过模式寄存器端口装入每个寄存器。

输出（位 0~7）

位			
7w = x 6w = x	模式类型选择		
	位 7	位 6	
	0	0 =	选择命令模式
	0	1 =	选择单模式
	1	0 =	选择块模式
	0	0 =	选择级联模式（无效）
5w = 1	选择地址递增		
0	选择地址递减（在 EISA 32 位和 EISA 普通地址 16 位传送模式下无效）		
4w = 0	禁止自动初始化		
1	开放自动初始化		
3w = x 2w = x	传送类型		
	位 3	位 2	

0	0	=	校验	
0	1	=	写	
1	0	=	读	
1	1	=	无效	
x	x	=	如果位于级联模式（位 6 和 7）	

1w = x
0w = x

选择通道号

位 1	位 0		
0	0	=	通道 4
0	1	=	通道 5
1	0	=	通道 6
1	1	=	通道 7

端口	类型	描述	平台
D8h	输出	DMA-2 清除字节触发器	AT+

任何向该端口的输出值都会清除 DMA 控制器 2 内的内部首/尾触发器。在向需要连续两次 8 位访问的 16 位端口执行 8 位读或写之前，可以执行该操作来完成该字的传送。在清除了触发器后，读或写低字节，然后读或写高字节，来访问 16 位 DMA 寄存器。只能清除该触发器，而不可读取该触发器。

端口	类型	描述	平台
DAh	输入	DMA-2 临时寄存器	AT+

该临时寄存器保存了在内存到内存数据传送期间的数据。传送完成后，临时寄存器保存了最后一次的传送数据。在 DMA 控制器没有执行一次 DMA 传送时，可以读取该临时寄存器。重启时会清除该寄存器，它是只可读的。

端口	类型	描述	平台
DAh	输出	DMA-2 主禁止	AT+

向这个端口写入任何值都会重启控制器 2。该命令执行一次与硬件重启相同的操作。设置屏蔽位（禁止通道 4~7），清除所有的命令、临时、以及字节触发器。该寄存器是只可写的。

端口	类型	描述	平台
DCh	输出	DMA-2 清除屏蔽寄存器	AT+

向该端口写入任何值都会清除屏蔽寄存器。清除屏蔽寄存器会开放通道 5、6 和 7 来接受 DMA 请求。该寄存器是只可写。

端口	类型	描述	平台
DEh	输出	DMA-2 写所有屏蔽位	AT+

该屏蔽寄存器用来禁止或开放单个引入的请求。将一个屏蔽位设置为禁止选中的通道。一次硬件重启会设置所有的屏蔽位而禁止所有的通道，该寄存器是只可写的。

输出（位 0~7）

位		
	7w = x	未使用
	6w = x	未使用
	5w = x	未使用
	4w = x	未使用
	3w = 0	通道 7 清除屏蔽位（开放）
	1	通道 7 开放屏蔽位
	2w = 0	通道 6 清除屏蔽位（开放）
	1	通道 6 开放屏蔽位
	1w = 0	通道 5 清除屏蔽位（开放）
	1	通道 5 开放屏蔽位
	0w = 0	通道 4 清除屏蔽位（开放）
	1	通道 4 开放屏蔽位

端口	类型	描述	平台
401h	I/O	DMA-1 基和当前计数，通道 0，高	EISA

EISA 系统在端口 1 的 16 位计数寄存器中又加入了 8 位，来提供 24 位计数。与传送的大小有关，一次可以传送 64MB（32 位传送*16M 的最大计数）

输入（位 0~7）—读当前计数寄存器

读取 DMA 通道 0 的 24 位当前计数寄存器的高 8 位。

输入（位 0~7）—写基和当前计数寄存器

写入两个 24 位计数寄存器的高 8 位。如果激活了链接模式，则只写入基计数。基计数寄存器是只可写的。

端口	类型	描述	平台
403h	I/O	DMA-1 基和当前计数，通道 1，高	EISA

该端口控制 DMA 通道 1 的 24 位基和当前计数寄存器，端口 401h 有进一步的祥述。

输入（位 0~7）—读当前计数寄存器

输入（位 0~7）—写基和当前计数寄存器

端口	类型	描述	平台
405h	I/O	DMA-1 基和当前计数，通道 2，高	EISA

该端口控制 DMA 通道 2 的 24 位基和当前计数寄存器。端口 401h 有进一步的样述。

输入（位 0~7）—读当前计数寄存器

输入（位 0~7）—写基和当前计数寄存器

端口	类型	描述	平台
407h	I/O	DMA-1 基和当前计数，通道 3，高	EISA

该端口控制 DMA 通道 3 的 24 位基和当前计数寄存器。端口 401h 有进一步的样述。

输入（位 0~7）—读当前计数寄存器

输入（位 0~7）—写基和当前计数寄存器

端口	类型	描述	平台
40Ah	输入	DMA-1 链式中断状态	EISA

使用链式模式和出现计数终端（传送完成）时，会激活 IRQ13。另外还会为指定的通道设置这个中断状态寄存器的某位。在该通道的链式模式设置为编程完成“11”，或禁止链式，或终止“00”时，会自动清除该位。该寄存器是只可读的。

输入（位 0~7）

位		
	7 r = 1	出现 DMA 通道 7 链式中断
	6 r = 1	出现 DMA 通道 6 链式中断
	5 r = 1	出现 DMA 通道 5 链式中断
	4 r = 1	未使用
	3 r = 1	出现 DMA 通道 3 链式中断
	2 r = 1	出现 DMA 通道 2 链式中断
	1 r = 1	出现 DMA 通道 1 链式中断
	0 r = 1	出现 DMA 通道 0 链式中断

端口	类型	描述	平台
40Ah	输出	DMA-1 链式模式寄存器	EISA

该寄存器控制链式 DMA 传送。参考有关 EISA 链式传送的章节以进一步了解有关细节，该寄存器是只可写的。

输出（位 0~7）

位		
	7 w = 0	未使用

6 w = 0　未使用
5 w = 0　未使用
4 w = 0　在 IRQ13 上发信号指示上次传送结束（缺省）
1　在计数终端线上发信号指示上次传送结束
3 w = x | 2 w = x | 链式模式

位 3	位 2	
0	0	= 禁止或终止链接
0	1	= 开放链式编程
1	0	= 无效代码
1	1	= 编程完成，开始

1 w = x | 0 w = x | 通道选则

位 1	位 0	
0	0	= 通道 0
0	1	= 通道 1
1	0	= 通道 2
1	1	= 通道 3

端口	类型	描述	平台
40Bh	输出	DMA-1 扩展模式寄存器	EISA

该寄存器包含了 EISA DMA 控制器上通道 0~3 高级特性。AT 兼容系统使用二进制值“000000xx”，并且是硬件重启后每个通道的缺省值，有些组合无效。参看表 18-7 了解这些限制。DMA 上清除不会影响该信息，这个寄存器是只可写的。

在 AT 系统上，通道 0~3 只支持 8 位传送。EISA 提供了许多选项来支持使用移 1 位地址实现 16 位传送，这样可以访问 64K 字。这一点和标准的 AT 支持通道 5、6 和 7 相似。

输出（位 0~7）

位　7 w = 0　禁止停止寄存器
1　开放停止寄存器
6 w = 0　达到计数终端（TC）时输出 TC
1　计数终端线作为输入来结束一次传送
5 w = x | 4 w = x | 周期定时模式

位 5	位 4	
0	0	= 兼容 AT 总线
0	1	= 类型 A
1	0	= 类型 B
1	1	= 类型 C，脉冲 DMA

3 w = x | 2 w = x | 地址翻译和 I/O 传送大小

位 3	位 2	
0		0 = 普通地址，8 位（AT）

	0	1 = 移位地址，16 位（AT）
	1	0 = 普通地址，32 位
1 w = x	1	1 = 普通地址，16 位
0 w = x	通道选择	
	位 1	位 0
	0	0 = 通道 0
	0	1 = 通道 1
	1	0 = 通道 2
	1	1 = 通道 3

端口	类型	描述	平台
40Ch	输入	DMA 链式计数终端状态	EISA

链式模式寄存器的位 4 指定了出现计数终端（TC）时的操作。所有通道的选择状态都从这个寄存器读取。一个“1”表示在出现 TC 时通道会标志 TC 线。一个“0”表示出现 TC 时通道会标志 IRQ13。该寄存器是只可读的。

输入（位 0~7）

位		
	7 r = 0	链接 TC 的 DMA 通道 7 标志 IRQ13
	6 r = 0	链接 TC 的 DMA 通道 6 标志 IRQ13
	5 r = 0	链接 TC 的 DMA 通道 5 标志 IRQ13
	4 r = 0	未使用
	3 r = 0	链接 TC 的 DMA 通道 3 标志 IRQ13
	2 r = 0	链接 TC 的 DMA 通道 2 标志 IRQ13
	1 r = 0	链接 TC 的 DMA 通道 1 标志 IRQ13
	0 r = 0	链接 TC 的 DMA 通道 0 标志 IRQ13

端口	类型	描述	平台
481h	I/O	DMA 页面寄存器，通道 2，高	EISA

该寄存器包含了通道 2 DMA 传送的地址位 A24~A31。总在写了基和低位页面寄存器后写这个寄存器。向该通道的基地址寄存器或向端口 81h 处的低位页面寄存器写，都会自动清除这个寄存器。这确保与 AT 系统上的 DMA 兼容。

端口	类型	描述	平台
482h	I/O	DMA 页面寄存器，通道 3，高	EISA

该寄存器包含了通道 3 的 32 位地址寄存器的高 8 位。参看端口 481h 了解细节。

端口	类型	描述	平台
483h	I/O	DMA 页面寄存器，通道 1，高	EISA

该寄存器包含了通道 1 的 32 位地址寄存器的高 8 位。参看端口 481h 了解细节。

端口	类型	描述	平台
487h	I/O	DMA 页面寄存器，通道 0，高	EISA

该寄存器包含了通道 0 的 32 位地址寄存器的高 8 位。参看端口 481h 了解细节。

端口	类型	描述	平台
489h	I/O	DMA 页面寄存器，通道 6，高	EISA

该寄存器包含了通道 6 的 32 位地址寄存器的高 8 位。参看端口 481h 了解细节。

端口	类型	描述	平台
48Ah	I/O	DMA 页面寄存器，通道 7，高	EISA

该寄存器包含了通道 7 的 32 位地址寄存器的高 8 位。参看端口 481h 了解细节。

端口	类型	描述	平台
48Bh	I/O	DMA 页面寄存器，通道 5，高	EISA

该寄存器包含了通道 5 的 32 位地址寄存器的高 8 位。参看端口 481h 了解细节。

端口	类型	描述	平台
4C6h	I/O	DMA-2 基和当前计数，通道 5，高	EISA

该寄存器控制 DMA 通道 5 的 24 位基和当前计数寄存器。参看端口 401h 了解细节。

输入（位 0~7）—读当前计数寄存器
输入（位 0~7）—写基和当前计数寄存器

端口	类型	描述	平台
4CAh	I/O	DMA-2 基和当前计数，通道 6，高	EISA

该寄存器控制 DMA 通道 6 的 24 位基和当前计数寄存器。参看端口 401h 了解细节。

输入（位 0~7）—读当前计数寄存器
输入（位 0~7）—写基和当前计数寄存器

端口	类型	描述	平台
4CEh	I/O	DMA-2 基和当前计数，通道 7，高	EISA

该寄存器控制 DMA 通道 7 的 24 位基和当前计数寄存器。参看端口 401h 了解细节。

输入（位 0~7）—读当前计数寄存器

输入（位 0~7）—写基和当前计数寄存器

端口	类型	描述	平台
4D4h	输入	DMA 链式状态	EISA

链式状态是一组标志位，用来指示是否激活了每个通道的链式模式。该寄存器是只可读的。

输入（位 0~7）链式状态

位		
	7 r = 1	激活 DMA 通道 7 链式
	6 r = 1	激活 DMA 通道 6 链式
	5 r = 1	激活 DMA 通道 5 链式
	4 r = 0	未使用
	3 r = 1	激活 DMA 通道 3 链式
	2 r = 1	激活 DMA 通道 2 链式
	1 r = 1	激活 DMA 通道 1 链式
	0 r = 1	激活 DMA 通道 0 链式

端口	类型	描述	平台
4D4h	输出	DMA-2 链式模式寄存器	EISA

该寄存器控制链式 DMA 传送。参考有关 EISA 链式传送的章节以进一步了解有关细节，该寄存器是只可写的。

输出（位 0~7）

位		
	7 w = 0	未使用
	6 w = 0	未使用
	5 w = 0	未使用
	4 w = 0	在 IRQ13 上发信号指示上次传送结束（缺省）
	1	在计数终端线上发信号指示上次传送结束
	3 w = x 2 w = x	链式模式

位 3	位 2
0	0 = 禁止或终止链接
0	1 = 开放链式编程
1	0 = 无效代码

	1	1 = 编程完成，开始
1 w = x 0 w = x	通道选则	
	位 1	位 0
	0	0 = 通道 4
	0	1 = 通道 5
	1	0 = 通道 6
	1	1 = 通道 7

端口	类型	描述	平台
4D6h	输出	DMA-2 扩展模式寄存器	EISA

该寄存器包含了 EISA DMA 控制器上通道 4~7 高级特性。AT 兼容系统使用二进制值“000000xx”，并且是硬件重启后每个通道的缺省值，有些组合无效。参看表 18-7 了解这些限制。DMA 主清除不会影响该信息。这个寄存器是只可写的。

在 AT 系统上，使用移 1 位地址来实现 16 位传送，这样可以访问 64K 字。EISA 提供了相同的兼容模式，但是也提供了普通的非移位 16 位模式。这一点可用于非字对齐的传送。

输出（位 0~7）

位		
7 w = 0		禁止停止寄存器
1		开放停止寄存器（参看 EISA FIFO 传送）
6 w = 0		达到计数终端（TC）时输出 TC
1		计数终端线作为输入来结束一次传送
5 w = x 4 w = x	周期定时模式	
	位 5	位 4
	0	0 = 兼容 AT 总线
	0	1 = 类型 A
	1	0 = 类型 B
	1	1 = 类型 C，脉冲 DMA
3 w = x 2 w = x	地址翻译和 I/O 传送大小	
	位 3	位 2
	0	0 = 普通地址，8 位（ISA）
	0	1 = 移位地址，16 位（ISA）
	1	0 = 普通地址，32 位
	1	1 = 普通地址，16 位
1 w = x 0 w = x	通道选择	
	位 1	位 0
	0	0 = 通道 4
	0	1 = 通道 5
	1	0 = 通道 6
	1	1 = 通道 7

端口	类型	描述	平台
4E0~4E2h	I/O	DMA-停止寄存器，通道 0	EISA

DMA 停止寄存器用来在低 16MB 内存中建立 FIFO 缓冲区。如果 DMA 传送达到保存在停止寄存器中的地址（缓冲区满），则停止传送并禁止该通道。从 8 位端口创建停止寄存器，该寄存器必须对齐双字边界。

端口 4E0h—DMA 通道 0 的停止地址 a0~a7（忽略 a0 和 a1）
端口 4E1h—DMA 通道 0 的停止地址 a8~a15
端口 4E2h—DMA 通道 0 的停止地址 a16~a23

端口	类型	描述	平台
4E4~4E6h	I/O	DMA-停止寄存器，通道 1	EISA

该寄存器是通道 1 的停止寄存器。参看端口 4E0~4E2h 了解细节。

端口 4E4h—DMA 通道 1 的停止地址 a0~a7（忽略 a0 和 a1）
端口 4E5h—DMA 通道 1 的停止地址 a8~a15
端口 4E6h—DMA 通道 1 的停止地址 a16~a23

端口	类型	描述	平台
4E8~4EAh	I/O	DMA-停止寄存器，通道 2	EISA

该寄存器是通道 2 的停止寄存器。参看端口 4E0~4E2h 了解细节。

端口 4E8h—DMA 通道 2 的停止地址 a0~a7（忽略 a0 和 a1）
端口 4E9h—DMA 通道 2 的停止地址 a8~a15
端口 4EAh—DMA 通道 2 的停止地址 a16~a23

端口	类型	描述	平台
4EC~4EEh	I/O	DMA-停止寄存器，通道 3	EISA

该寄存器是通道 3 的停止寄存器，参看端口 4E0~4E2h 了解细节。

端口 4ECh—DMA 通道 3 的停止地址 a0~a7（忽略 a0 和 a1）
端口 4EDh—DMA 通道 3 的停止地址 a8~a15
端口 4EEh—DMA 通道 3 的停止地址 a16~a23

端口	类型	描述	平台
4F4~4F6h	I/O	DMA-停止寄存器，通道 5	EISA

该寄存器是通道 5 的停止寄存器，参看端口 4E0~4E2h 了解细节。

端口 4F4h—DMA 通道 5 的停止地址 a0~a7（忽略 a0 和 a1）
端口 4F5h—DMA 通道 5 的停止地址 a8~a15
端口 4F6h—DMA 通道 5 的停止地址 a16~a23

端口	类型	描述	平台
4F8~4FAh	I/O	DMA-停止寄存器，通道 6	EISA

该寄存器是通道 1 的停止寄存器。参看端口 4E0~4E2h 了解细节。

端口 4F8h—DMA 通道 6 的停止地址 a0~a7（忽略 a0 和 a1）
端口 4F9h—DMA 通道 6 的停止地址 a8~a15
端口 4FAh—DMA 通道 6 的停止地址 a16~a23

端口	类型	描述	平台
4FC~4FEh	I/O	DMA-停止寄存器，通道 7	EISA

该寄存器是通道 7 的停止寄存器。参看端口 4E0~4E2h 了解细节。

端口 4FCh—DMA 通道 7 的停止地址 a0~a7（忽略 a0 和 a1）
端口 4FDh—DMA 通道 7 的停止地址 a8~a15
端口 4FEh—DMA 通道 7 的停止地址 a16~a23

附录 A

软件包中的程序

本附录归纳了软件包中的所附带的文件。

可执行程序归纳

程序名	描述	所在章
a20test	测试键盘控制器改变 A20 的能力*	8
beepx	显示汇编和 C 程序	3
biosview	显示键盘 BIOS 值和完整描述*	6
cmosview	显示主和扩展 CMOS 寄存器*	15
cpurdmsr	寻找并描述模式专用寄存器*	4
cpurdtsc	测试未公开的 Pentium 时间标志计数器	4
cputest	寻找未公开的操作码*	4
cputype	标识 CPU 信息*	4
cpuundoc	测试所有未公开的指令*	4
ctrl2cap	交换 Ctrl 和 Caps Lock 的 TSR	8
disktest	测试 BIOS 问题，扇区读*	10、11
iodelay	任何 CPU 上 I/O 的短延迟	3
iospy	监视 I/O 端口状态的 TSR	2
irqredir	改变 IRQ 重定向	17
keybios	检查扩展键盘的支持性	8
keyscan	显示按下任何键时的键盘码或扫描码*	8
max_recv	用于找出串口传送的最大速率	12
max_xmit	用于找出串口传送的最大速率	12
prnascii	支持非 ASCII 打印的 TSR	14
prnswap	交换并口*	14
prntype	并口的信息	14
sertype	串口的信息*	12

sspy	显示串口状态的 TSR	12
systype	显示系统类型、BIOS、磁盘和视频信息*	4、9、10、11
testcase	事件时钟程序 TIMEVENT 的测试情况	16
timevent	高精度事件持续时钟	16
tmrfast	使系统时钟快 50 倍	16
unpc	观察 I/O 端口，运行其他测试	2

*对 PC 内幕第一版来说是新加入的、修改的和/或增强的

有趣的子程序

子程序名	描述	母程序
biosinfo	标识 BIOS 制造商和日期，有时该子程序也标识芯片集、LBA 支持以及主板制造地址	systype.asm
cache_d_size	测量内部 CPU 高速缓存的大小	cputype.asm
check_A20	找出现在是否禁止或开放 A20	a20test.asm
cpu386	标识 CPU 8088 到 80386+。不测试 80386 以上的 CPU	systype.asm
cpu_cache	确定是否开放了 CPU 高速缓存	cputype..asm
cpuq	测量了 CPU 预取队列的大小	cputype.asm
cpumode	返回 CPU 处于实模式、保护模式还是 V386 模式	cputype.asm
cpuspeed	确定 CPU 速度，单位 MHz	cputype.asm
cpustep	标识 CPU 修订版	cputype.asm
cpuvalue	获取当前 CPU 类型，8088~P7 或其他系列、CPU 支持的指令集以及 CPU 是否支持指令 CPUID。	cputype.asm
cpuvendor	标识 CPU 制造商	cputype.asm
disktype	获取软盘的类型，302K~2.88M	systype.asm
dma_read	显示如何执行一次 DMA 传送	dma_read.sub
fpuloc	确定 CPU 内是否有浮点处理器	cputype.asm
fputype	标识浮点处理器	cputype.asm
hdsktype	获取硬盘类型和大小	systype.asm
iodelay	1uS 短延迟的子程序	iodelay.asm
key9	检查是否支持中断 16h 功能 9	keybios.asm
keyboard_cmd	向键盘控制器发送一条命令	keyview.asm
keyboard_read	从键盘读取一个字节	keyview.asm
keyboar_write	向键盘写入一个字节	keyview.asm
keytype	检查是否支持扩展键盘	keybios.asm
PCI_detect	检查主板是否有 PCI 总线	systype.asm
read_CMOS	读取指定 CMOS 寄存器的内容	cmosview.asm
serslow	将 URAT 放慢到每秒 2 位	serslow.asm

setLEDs	调试一个端口时控制键盘上的 LED	keyview.asm
sysvalue	获取系统的类型：PC、XT、AT、EISA、MCA 或 PCI	systype.asm
testport	标识 URAT 芯片类型	systype.asm
video_type	获取视频类型，MDA~XGA 等等	systype.asm
write_CMOS	向指定的 CMOS 寄存器写内容	cmosview.asm

可执行程序的详细解释

A20TEST

该程序显示了当前 A20 状态，只要 CPU 不位于 V86 模式下，它就试图通过主板控制器改变状态。完成时，A20 的状态会返回到以前的值。

BEEPx

有三个程序显示了中断和 I/O 端口操作，汇编和 C 代码。程序是 BEEPA.ASM、BEEPB.C、以及 MEEP.C。后两个程序是 Borland C 和 Microsoft C 的。Beepx 使用时钟 2 来生成一个 2 秒的 5KHz 嘟嘟声。

Biosview

显示低端内存中当前的 BIOS 值，并加以详细描述。观察结果输出的最好方法是将输出重定向到一个文件中。例如，“BIOSVIEW > BIOSDATA.TXT”。

CMOSVIEW

显示 CMOS 的内容，并公开功能名。参看第 15 章了解供应商专用功能，它因 BIOS 制造商而有所不同。CMOSVIEW 也检查三个扩展 CMOS 数据区和显示找到的任何扩展数据。

CPURDMSR

某些 386+ CPU 支持新式指令，读和写模式专用寄存器（RDMSR 和 WRMSR）。它们访问 CPU 的专门功能。调用该功能时，ECX 设置为要查看的寄存器。该测试程序仔细查看每个可能的组合来看 CPU 使用哪些寄存器。对任何找到的寄存器都会提供给一个当前的 64 位寄存器值和描述（如果知道）。

使用“CPURDMSR+”来显示每次执行的测试。它的运行速度比缺省时要慢 30 倍，后者仅仅每 65,536 次测试才显示一次——除非找到了一个寄存器。使用“CPURDMSR-”来测试和显示开始于 80000000h 的 32 个隐藏寄存器。

CPURDTSC

Pentium 处理器使用一条新的指令，RDTSC，来读取时间标志计数器。这个程序连续显示时间标志计数器的值，直到按下了 Ctrl-Break。

CPUTEST

定位 80286 及后来 CPU 上的潜在隐藏指令。该程序查看未使用的操作码是否陷入了坏码中断，来定位这些潜在的隐藏指令。忽略 Intel 已经公开的已知指令以及已知的未公开指令。CPU 指示支持 MMX 指令集时，该程序也忽略 MMX 指令。使用 CPUUNDOC 来显示和测试已知未公开指令的有效性。

CPUTYPE

显示 CPU 信息。显示许多独立的检测子程序的结果。显示的信息包括 CPU 类型（由制造商提供）、所支持的真正指令集（相对于 Intel）、CPU 供应商、内部 CPU 速度、预取队列的大小、CPU 级别和修订版信息、L1 高速缓存状态和测量到的 L1 高速缓存大小。此外，该程序还标识是否支持 CPUID、CyrixID 寄存器、或者 MMX 指令。如果支持 COUID 或 CyrixID，则还会提供其他的相关信息。

使用命令行选项“-”来阻止要求使用保护模式指令的测试，V86 模式可能不会正确的处理该指令。使用选项“+”来显示内部 CPU 速度和 L1 高速缓存大小测试的原始计时值。

CPUUNDOC

测试当前 CPU 的所有未公开指令。大多数测试校验每个未公开指令的功能。

CTRL2CAP

这个 TSR 交换 Ctrl 和 Caps Lock 键的功能。演示中断 15h 功能 4Ah 钩起扫描码的用法。

DISKTEST

从指定的磁盘读取扇区来测试 BIOS 是如何响应的。如果使用 BIOS 服务中断 13h 一次读取了一个以上的磁道，该测试程序会查找问题的所在。许多系统有不同的限制。对硬盘和软盘都可以进行测试，但是对软盘测试需要几分钟。在命令行上指明磁盘号，软盘驱动器是 0 或 1，硬盘驱动器是 81、82 等等。

IODELAY

显示 IODELAY 子程序的用法。这个例子并无实际用处，而仅仅只是显示快速接连访问同一个 I/O 端口时如何使用 IPDELAY。

I/O SPY

这个 TSR 显示了指定端口的当前状态。每秒读取指定端口 18 次。一次可以监视不超过 8 个端口。有许多命令行选项。

IRQREDIR

将 IRQ 0~7 重定向为一个新的中断向量。如果系统位于 V86 模式，并且 IRQ 5 用于高中断率，则这一点可能会提高运行速度。参看第 17 章的警告部分以了解 IRQ 5 与中断 0Dh 之间的冲突。

KEYBIOS

检查 BIOS 是否支持中断 16h 扩展 BIOS 服务。检查的特殊服务包括功能 5、9、10h、11h 以及 12h。

KEYSCAN

显示当前键盘控制器的命令字节，然后关闭普通的键盘控制器翻译，并显示按下某个键时的未翻译码。Escap 键用来退出程序。所有带有键盘控制器的计算机都可以使用该程序，但是 PC/XT 除外。程序也显示了如何打开和关闭键盘 LED。

使用命令行选项“-k”来保持激活翻译，并显示按下键的扫描码。

MAX_RECV

该程序和程序 MAX_XMIT 一起使用来找出通过空的调制解调电缆相连的两个系统之间可承受的最大传输速率。缺省时使用串口 1，除非在命令行上指定了串口 1~4。

MAX_XMIT

该程序和程序 MAX_RECV 一起使用来找出通过空的调制解调电缆相连的两个系统之间可承受的最大传输速率。缺省时使用串口 1，除非在命令行上指定了串口 1~4。

PRNASCII

这个 TSR 将非 ASCII 字符转化为那些不能打印全部 IBM 字符集的打印机可接受的字符。大多数程序和屏幕打印都可以使用它。

PRNSWAP

交换打印机 1 和 2 的并口。PRNSWAP 不是一个 TSR。

PRNTYPE

检测活动的并口，并标识这些端口支持哪些扩展模式。

SERIAL TYPE

报告活动的串口 I/O 地址，并标识 UART 类型。

SERIAL SPY

这个驻留程序显示一个指定端口当前的调制解调输入和输出状态。在命令行上指定要监视的端口，1~4。

SYSTYPE

确定系统总线类型、BIOS 供应商和日期、软盘类型、硬盘类型、视频显示类型，有时还确定视频显示器的供应商。

TESTCASE

这个程序用来显示 TIMEVENT 是如何监视一段代码的操作时间的。运行该程序之前先运行 TIMEVENT，然后运行 TESTCASE。然后再次运行 TIMEVENT，来看运行 TESTCASE 中的一段代码所花的时间。

TIMEVENT

这个程序用来显示任何前面事件的执行周期，并重设时钟来为下一个事件计时。参看 TESTCCASE。

TMRFAST

这个演示程序显示提高系统的中断率，以超过普通的每秒 18.2 次的中断率。当需要较高中断率但又不想影响其他程序时，这个程序比较有用。演示程序在一个黑色背景的变化的绿色字符上放上一个白色的强调字符。在文本模式下，每秒快速交换字符 910 次，而在普通的系统时钟速率下是每秒 50 次。

UNPC

观察所有的 I/O 端口、运行工具、以及显示信息。这也是运行程序和查看本书中所包括的许多工具的源代码的最快的方法。

附录 B

术语表

常用的缩写形式

ABIOS IBM PS/2 上的一部分系统 BIOS，仅 OS/2 使用。

AT 先进技术，IBM 使用这个称谓来指它的早期基于 286 的 PC，它提供了前 16 位适配器总线。通常称许多新式的机器是 AT 兼容的，即使它们使用 386 或更晚的 CPU。ISA（工业标准结构）是 AT 设计的产物，这两个缩写可以互换使用。

AT+ 286 以及后来的 PC，包括 AT、ISA、EISA 和 MCA 总线设计

ATA (AT 附件) ATA 定义了内部块设备的命令及物理方面。最常见的块设备是 IDE 硬驱。

BIOS 基本输入/输出系统，它提供了对计算机硬件的最底层软件控制。

CMOS 在 PC 领域，它指的是一个不可改变内存设备，用于保存少量的系统配置信息（它真正的含义是互补金属氧化物半导体）。

CRC 循环冗余校验——数据，用来校验数据块中是否出现讹误。通常软盘驱动器使用它。

CTS 清除以便发送——来自串口连接器的一条输入线

DAC 数模转化器，VGA 生成色彩时使用到。

DBCS 双字节字符集

DMA 直接内存访问

dpi 每英寸的点数，鼠标、显示器以及其他设备分辨率的度量单位

DPMI DOS 保护模式接口。运行于 DOS 保护模式下的程序的服务标准

DSR 数据终端准备好——来自串口连接器的一条输入线

DTR 数据设置准备好——到串口连接器的一条输出线

EBDA 扩展 BIOS 数据区——主存顶部的保存区域，供系统 BTOS 使用。

ECC 硬盘驱动器的错误检查和纠正——检测并纠正脉冲错误的方法。常用的方法能够检测并纠正 11 位以内的毁坏序列。

EISA 扩展工业标准结构——这个独立的标准定义了 32 位 AT 设计的硬件和总线设计，它是标准 AT/ISA 的扩展。

ESDI 增强型小设备接口——一个早已过时了的硬盘驱动器接口

FEP 字体终点处理器（用于日本 DBCS 字符的输入）

FIFO 先进先出队列

FPU 浮点处理器

GDT 全局描述符表

HLDA 保持来自 CPU 的确认

HMA 高端内存区，由 64K 减去 16 字节组成，开始于 1MB 地址边界

ICE 内部电路模拟器，调试器中的极限点

IDT 中断描述符表

int 中断

IOPL I/O 优先级（0~3，0 的优先级最高）

IRQ 中断请求（一条硬件线）

ISA 工业标准结构——一个独立标准，为 IBM AT 复制品定义了 AT 硬件和软件 16 位总线设计。

kbps 千位每秒

LBA 逻辑块访问——用于将一个块设备，例如硬盘驱动器，参考为线形扇区映射。

LIDT 逻辑中断描述符表

LSB 字节/字的最低位（与上下文有关）

mbps 每秒百万位

MCA 微通道结构——不连续 PS/2 系统上所使用的总线标准。不兼容 AT 类型的适配卡，但是其他软件和 AT 系统兼容。

MSB 字节/字的最高位（与上下文有关）

NMI 不可屏蔽中断

PC 个人计算机——IBM 的计算机，使这本书成为可能。

PCI 外部设备互连——ISA 的替代卡总线标准。当今大多数 Pentium 类型的系统上通常可以找到这种标准。

pel 视频显示器上的一个象素或点

PIO 编程输入/输出——一种数据传输格式，该传输通过 I/O 端口，而不是提供内存。

POS 可编程选项选则，PS/2 MCA 系统的新特性。

POST BIOS 在上电或重启之后立即执行的上电自检。

RGB 红-绿-蓝，老式 EGA 访问显示器的格式，由三种颜色组成：红、绿、蓝。有时使用格式 IRGB，其中“I”代表强度。

RI 令牌指示器——来自串口连接器的一条重要的输入线

RSIS 重定位屏幕接口规范

RTC 实时时钟（CMOS 时钟）

RTS 请求发送——到串口连接器的一条重要的输出线

SCSI 小型计算机系统接口——主要用来连接磁盘驱动器、磁带备份系统、以及 CD ROM 驱动器。

TSR 终端驻留程序（内存驻留）

UART 通用异步接收器传输器——串口操作硬件

V86 虚拟 86 模式。386 以及后来 CPU 上的操作模式，用来模拟在监督程序下的 8088 环境。

VCPI 虚拟控制程序接口——实现保护模式下 DOS 程序和内存管理程序共存的一种方法。

VDS 虚拟 DMA 规范。在 386 及后来的系统带有活动分页时 VDS 用于管理 DMA 操作和内存。

VESA 视频电子标准协会。一个视频卡制造商的团体，他们提供一种通用的高分辨率标准。

VxD 虚拟设备驱动程序——32 位 Windows 设备驱动程序

XT IBM 早期 PC 的后继产品，它支持一个硬盘。大多数 XT 以 8088 为基础，但是有些 80286 也使用它。所有的适配卡都设计成 8 位的。

YUV 标准 NTSC 电视上使用的色彩标准。Y 控制亮度，U 和 V 控制颜色。

常见芯片的编号和功能

下面这个列表包括了各种子系统所使用的常见的 LSI 部件，通过 I/O 端口总线访问它们。不包括 CPU、RAM 或 EPROM。新式的芯片组用更少的芯片代替了许多上述芯片，但却提供了相同的功能。仅列出了早期的芯片供应商或最常见的制造商。大多数情况下，其他的供应商提供了相同的部件，与所列出的部件号只有一点点差别。

765 软盘控制器（NEC）

6845 CGA 和 MDA 的视频控制器（Motorola）

8031 微控制器，键盘和主板键盘控制器使用（Intel ROM 版）

8042 微控制器，主板键盘控制器使用（Intel）

8048 微控制器，老式键盘使用（Intel）

8051 微控制器，键盘和主板键盘控制器使用（Intel，EPROM 版）

8037 DMA 控制器（各种）

8250 串口 UART（国家半导体）

8254 可编程间隔时钟，带 3 个时钟/计数器（Intel）

8255 这个接口芯片为硬件提供了三个 8 位端口。仅用在 PC/XT 上。在后来的系统上，某些 I/O 端口仍描述为 8255，但是芯片组和其他方法用等价的功能替代了实际的 8255。

8259 可编程中断控制器（Intel）

8272 软盘控制器（Intel）

16450 串口 UART（国家半导体）

16550 串口 UART，带有可任选的 FIFO 模式（国家半导体）

16552 双串口 UART，每个都带有可任选的 FIFO 模式（国家半导体）

72065B 软盘控制器（NEC）

72077 软盘控制器，支持 2.88MB(Intel)

82510 串口 UART，带有可任选的 FIFO 模式(Intel)

146818 CMOS 实时时钟和 CMOS 内存（Motorala）

161450 串口 UART，类似于 16450，但是包括了软件断电控制（Startech 半导体）

161550 串口 UART，类似于 16550，但是包括了软件断电控制（Startech 半导体）

读者信息反馈表

亲爱的读者：

首先感谢您对本社电脑图书的关爱，为了更好地完善本社的服务体系，希望您能在百忙之中抽出一点时间，填妥以下表格。我们将审慎考虑您的宝贵意见并加以改进。同时，您也将有机会免费获得我社提供的精品图书。再次向您表示深深的谢意。

请寄往以下地址：100044　北京西城三里河路 6 号　中国电力出版社计算机编辑室

网　址：www.infopower.com.cn　E-mail：cepp01@mx.cei.gov.cn

姓名：_____　性别：□1.男□0.女　生日：_____年_____月_____日

文化程度：□1.小学　□2.中学　□3.高中/中专　□4.大学　□5.硕士及以上

计算机应用水平：□1.初学者　□2.一般爱好者　□3.电脑发烧友

□4.计算机专业人员　□5.计算机高级技术人员

职　业：□1.系统/网络管理员　□2.采购　□3.销售　□4.行政

□5.财务/文秘　□6.设计/研究　□7.维修/质量控制/测试

□8.生产/制造　□9.教育培训　□10 其他_____

工作单位：_____

地　址：_____

电　话：_____ 传真：_____

邮　编：_____ E-mail: _____

书名：_____　购买书店：_____

本书品质：5. 非常好　4. 很好　3. 普通　2. 不好　1. 很差

- 封面设计：5□4□3□2□1□
- 印刷：5□4□3□2□1□
- 作者功力：5□4□3□2□1□
- 编排：5□4□3□2□1□

本书的内容，对您而言：

□太难了　□有点难　□恰到好处　□有点简单　□太简单

如果您要买电脑图书，您最常到：

□一般书店　□计算机图书专卖店　□电脑门市　□新华书店　□电脑图书展

□邮购　□网上订购　□其他_____

您选择购买本书的原因为：（可复选）

□品质优良　□朋友建议　□价格合理　□书店推荐　□报刊杂志推荐　□作者　□出版社

□广告　□促销优惠　□附加价值（光盘或磁盘）　□其他_____

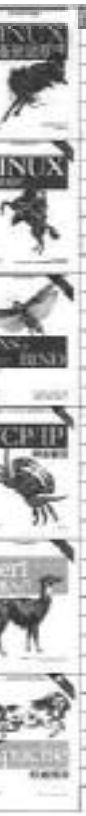
您希望中国电力出版社将来为您出版哪一类型电脑图书？（可复选）

□操作系统　□数据库　□网络/通信　□程序设计　□图像处理/多媒体设计

□电脑硬件/接口开发　□开源软件　□办公自动化软件　□儿童电脑　□其他_____

您是否愿意收到中国电力出版社的计算机图书目录？　□是　□否

其他建议：_____



近 机图书重点推荐

1	开源软件丛书	指南》	46.00
2		入门》（第二版）	49.00
3		络管理》（第二版）	59.00
4		的 C 编程》	45.00
5		威指南》（第三版）（附光盘）	69.00
6		备驱动程序》	59.00
7		威指南》（影印版）（附光盘）	59.00
8		—Win32 版》	49.00
9		》上、下卷（第二版）（影印版）	118.00
10		ND》	59.00
11		Mozilla 源代码指南》（附光盘）	45.00
12		编程》（即将出版）	79.00
13	系统网络管理丛书	ows NT 版》	39.00
14		NT TCP/IP 网络管理》	49.00
15		2000 活动目录》	49.00
16		由器管理》	40.00
17		性能》	49.00
18		《Unix 系统管理》（即将出版）	69.00
19	职业网络管理员培训教程丛书	《联网基础》	45.00
20		《联网设备与概念》	45.00
21		《局域网络》	45.00
22		《进程与协议》	45.00
23		《广域网络》	45.00
24		《网络分析与设计》	45.00
25		《因特网基础》	45.00
26		《TCP/IP 与网络体系结构》	45.00
27	高级界面特效制作百例（附光盘）	《Visual C++高级界面特效制作百例》	79.00
28		《Delphi 高级界面特效制作百例》	65.00
29		《Visual Basic 高级界面特效制作百例》	69.00
30		《C++ Builder 高级界面特效制作百例》	65.00
31	数学工具软件丛书	《Matlab 5.3 实例教程》	24.00
32		《Mathematica 4.0 实例教程》	24.00
33		《Mathcad 2000 实例教程》	24.00
34		《Maple V R5 实例教程》	24.00
35		《Origin 6.0 实例教程》	24.00
36	硬件接口开发系列	《并行端口编程》	39.00
37		《Windows VxD 与设备驱动程序权威指南》	即将出版
38		《SCSI 程序员指南》	
39		《AGP 系统体系》	
40		《USB 系统体系》	

网 址：www.infopower.com.cn

E-mail：cepp01@mx.cei.gov.cn

北航 C0535883