



Microsoft® Programming Books

Microsoft Press

Programming the **Windows Driver Model**

Covers
Windows
2000



The official
guide to the
Microsoft Windows
Driver Model

Walter Oney

Copyright© 1999 by Walter Oney

PUBLISHED BY

微软出版社

微软公司的子公司

One Microsoft Way Redmond, Washington 98052-6399

版权 沃尔特 Oney 1999 所有版权保留。

没有出版者的书面许可这本书的内容的任何部份不可以复制或传播不论何种形式的或无论如何。

Library of Congress Cataloging-in-Publication Data Oney, Walter.

Programming the Microsoft Windows Driver Model p. cm. Includes index.

ISBN 0-7356-0588-2

1. Microsoft Windows NT device drivers (Computer programs)
2. Computer programming. I. Title QA76.76.D49O54 1999 005.7'126--dc21 99-33878 CIP

印刷和装订于美国。

1 2 3 4 5 6 7 8 9 QMOM 4 3 2 1 0 9

发行在加拿大 by 企鹅书业加拿大股份有限公司

此书的 CIP 目录分类档案生效在英国图书馆。

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at mspress.microsoft.com. Intel is a registered trademark of Intel Corporation. Microsoft, Microsoft Press, MSDN, Visual C++, Visual Studio, Win32, Windows, and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person, or event is intended or should be inferred.

采集编辑: Ben Ryan

项目编辑: Devon Musgrave

技术编辑: Robert Lyon

献给

我的父母亲，他们给了我生命并教会我做一切。

前言

Windows Driver Model(WDM)的根源可追溯到几年前一种叫做 Windows for Workgroups 3.10 的操作系统。那时候我们努力地支持无数不同的 SCSI 控制器，我长期地注意 WindowsNT 开发组创建的小端口驱动程序类型。不久就认识到重新构造必要的映象加载器(image loader)和小端口驱动程序需要的执行环境比把这些小端口驱动程序重写成某些 VXD 形式的驱动程序并调试完毕所花费的努力要少得多。

不幸的是，Windows from Workgroups 3.10 已经停止发行带有 SCSI 小端口支持的版本，主要是由于象 ASPI(高级 SCSI 编程接口)这样的外围问题。然而，跨 Windows 和 windows NT 操作系统共享同样的驱动程序执行映象的基础是适当的并且可以在 win95 中见到，它(win95)可以与 NT 共享 SCSI 和 NDIS 小端口驱动程序二进制代码。

共享驱动程序模式的潜在意义是重大的。驱动程序开发人员感兴趣的是支持双平台，共享驱动模式能降低开发和调试的一半开销。对微软来说，共享模式意味着更容易地从 win9x 迁移到 Windows2000 或这个平台的未来版本。对最终用户来说，大量不同种类的稳定驱动程序可以在这个家族中的所有成员之间通用。

下一个(逻辑)步骤是

The next logical step, then, was to create a driver model with the ability to share general purpose drivers across both platforms. But what form should it take? Three requirements were immediately obvious: it must be multiprocessor-capable, it must be processor-independent, and it must support Plug and Play (PnP). Fortunately, the Windows NT 4.0 driver model met the first two requirements, and it seemed clear that the next major release of Windows NT would support PnP as well. As a result, WDM can be considered a proper subset of what is now the Windows NT driver model.

The potential benefits of a shared driver model can be realized today for many classes of devices, and choosing the WDM driver model will continue to pay dividends in the future. For example, a correctly written WDM driver requires only a recompile before functioning in an NT 64-bit environment prototype.

WDM will continue to evolve as new platforms and device classes are supported. Future versions of Windows 9x and Windows 2000 will contain upwardly revised WDM execution environments. Fortunately, WDM is designed to be "backward compatible," meaning that WDM drivers written according to the Windows 2000 DDK and designed to work for the intended environment will continue to work in a subsequent WDM environments.

There is a lot to WDM, and in this book Walter does an excellent job in offering an in-depth tour of every aspect as well as the philosophy of the Windows Driver Model.

Forrest Foltz
微软公司 Windows 开发体系结构设计者

致谢

我感谢所有帮助我完成此书的人们. Devon Musgrave, Robert Lyon, 和其余为把此书从原始的 winword 手稿转换成在你手中的精美作品而服务的微软出版社工作人员。我清楚为了这个项目采编 Ben Ryan 花费无数小时并飞行数千英里寻找好的作者和有用的新书, 祝他下次好运。微软的 Sandy Spinrad, 在百忙之中还有力地协助查找技术资料, 作硬件测试, 更新版本, 和我所依赖的其它许多资料。许多 windows98 和 windows2000 基本开发组成员审阅了这些材料, 值得个别提起, 他们要求匿名, 但至少我知道他们是谁。我们研究会的学者和网上团体在大大小小的各个方面给予帮助, 提出深刻的问题和共享难得的洞察力。最后, 我要感谢我的妻子 Marty, 她总是在我工作最困难时在我身边。

Walter Oney

<http://www.oneysoft.com>

驱动开发网的全体翻译人员也要感谢网上为我们提供工具和协助翻译的人们。znsoft 要感谢他的女友每天打电话问候他。

目录

献给	3
前言	4
致谢	5
目录	6
第一章：导言	15
操作系统概述	16
Windows 2000 概述	16
Windows 98 概述	17
Windows 2000 驱动程序	19
内核模式驱动程序的属性	19
• 可移植性	19
• 可配置性	20
• 可抢先性和可中断性	20
• 多处理器安全	20
• 基于对象	20
• 包驱动	21
• 异步	21
WDM 驱动程序模型	21
例子代码	23
随书光盘	23
关于创建例子驱动程序	24
GENERIC.SYS	24
本书的结构	25
关于书中的错误	25
其它资源	26
驱动程序开发书籍	26
其它参考书籍	26
杂志	26
新闻组	26
讲座	26
注意事项	27
第二章：WDM 驱动程序的基本结构	28
设备和驱动程序的层次结构	29
系统怎样装入驱动程序	30
递归枚举	30
注册表的角色	31
驱动程序装入顺序	35
设备对象之间如何关联	36
检查设备堆	38
驱动程序对象	39
设备对象	41
DriverEntry 例程	45
DriverEntry 概述	45
DriverUnload 例程	46
驱动程序再初始化例程	47
AddDevice 例程	48
创建设备对象	48
为设备命名	49
符号连接	50
应该命名设备对象吗？	52
设备名称	53
设备接口	53
其它全局性的设备初始化操作	57
初始化设备扩展	57

初始化默认的DPC对象	58
设置缓冲区对齐掩码	59
其它对象	59
初始化设备标志	59
设置初始电源状态	60
建立设备堆	60
清除DO_DEVICE_INITIALIZING标志	60
Windows 98 兼容问题.....	61
DriverEntry调用上的不同	61
注册表组织的不同	61
\??目录	61
未实现的设备类型	61
第三章：基本编程技术	62
内核模式编程环境	63
使用标准运行时间库函数	63
注意侧效	64
错误处理	65
状态代码	65
结构化异常处理	66
Try-Finally块	68
Try-Except块	69
异常过滤表达式	70
生成异常	72
一些真实环境中的例子	72
Bug Checks	74
内存管理	76
用户模式地址空间与内核模式地址空间	76
一页有多大？	77
分页和非分页内存	77
编译时控制分页能力	78
运行时控制分页能力	79
堆分配符	81
释放内存块	81
ExAllocatePoolWithTag	82
ExAllocatePool的其它形式	82
链表	82
双链表	83
单链表	85
Lookaside(后援式)链表	86
字符串操作	89
分配和释放串缓冲区	90
Blob数据(大块数据)	90
其它编程技术	92
访问注册表	92
打开注册表键	92
其它打开注册表键的方法	93
获取和设置注册表值	94
删除子键或键值	95
枚举子键或键值	95
访问文件	97
打开已存在文件然后读	97
创建或重写文件	97
浮点运算	99
调试技巧	99
Windows 98 兼容问题	101
第四章：同步	102

一个原始的同步问题	103
中断请求级	105
IRQL的变化	106
基本同步规则	106
IRQL与线程优先级	106
IRQL和分页	107
IRQL的隐含控制	107
IRQL的明确控制	107
自旋锁	109
使用自旋锁	109
内核同步对象	111
何时阻塞和怎样阻塞一个线程	111
在单同步对象上等待	112
在多同步对象上等待	113
内核事件	113
内核信号灯	115
内核互斥对象	116
内核定时器	117
通知定时器用起来象事件	118
通知定时器与DPC例程	118
同步定时器	119
周期性定时器	119
取消一个周期性定时器	120
一个例子	120
定时函数	121
内核线程同步	121
线程警惕和APC	122
APC与I/O请求	122
如何指定Alertable和WaitMode参数	123
其它内核模式同步要素	125
快速互斥对象	125
互锁运算	126
InterlockedXxx函数	127
ExInterlockedXxx函数	128
链表的互锁访问	129
初始化	129
插入元素	129
删除元素	130
IRQL的限制	130
第五章: I/O请求包	132
数据结构	133
IRP结构	133
I/O堆栈	134
IRP处理的“标准模型”	137
创建IRP	137
发往派遣例程	138
派遣例程的职责	138
StartIo例程	139
中断服务例程	140
DPC例程	140
定制队列	141
完成I/O请求	144
完成机制	144
使用完成例程	145
完成例程如何获得调用	146
完成例程为什么要调用IoMarkIrpPending	147

向下级传递请求	150
取消I/O请求	153
要是没有多任务就.....	153
同步化取消操作	153
情况 1: CPU A先获得自旋锁	157
情况 2: 就在CPU A刚要获取自旋锁前CPU B获得了自旋锁	157
情况 3: CPU B获得自旋锁两次	158
清除相关的IRP.....	158
管理自己的IRP.....	162
使用IoBuildSynchronousFsdRequest	162
清除	163
取消同步IRP.....	163
使用IoAllocateIrp	164
松散的结尾	166
使用IoBuildDeviceIoControlRequest.....	166
使用IoBuildAsynchronousFsdRequest	166
设备对象指针从哪来?	167
第六章: 即插即用	169
IRP_MJ_PNP派遣函数.....	171
启动和停止设备	173
前进和等待IRP.....	173
提取资源分配信息	175
IRP_MN_STOP_DEVICE	176
IRP_MN_REMOVE_DEVICE.....	177
IRP_MN_SURPRISE_REMOVAL	178
管理PnP状态转换.....	180
使用DEVQUEUE来排队和取消IRP	181
用DEVQUEUE排队PnP请求	183
启动设备	183
可以停止设备吗?	184
当设备停止时	185
可以删除设备吗?	185
同步删除	186
DEVQUEUE如何工作	190
初始化DEVQUEUE	190
停止队列	191
排队IRP	191
出队IRP	192
取消IRP.....	193
等待当前的IRP.....	195
放弃请求	195
其它配置功能	197
过滤资源需求	197
设备用途通知	198
DeviceUsageTypePaging.....	199
DeviceUsageTypeDumpFile	199
DeviceUsageTypeHibernation	200
控制器和多功能设备	200
整体结构	200
创建子设备对象	200
设备向PnP管理器告知自己含有子设备	202
以PDO角色处理PnP请求	203
处理设备删除	206
处理IRP_MN_QUERY_ID请求	206
处理IRP_MN_QUERY_DEVICE_RELATIONS请求.....	207
处理子设备资源	207

PnP通知.....	208
WM_DEVICECHANGE扩充.....	208
何时关闭设备句柄	209
Windows 2000 的服务通知	210
内核模式通知	210
定制通知	213
Windows 98 兼容问题.....	215
第七章：读写数据	216
配置设备	217
寻址数据缓冲区	219
指定缓冲方式	219
Buffered方式	220
Direct方式.....	220
Neither方式	222
端口与寄存器	223
端口资源	224
内存资源	225
响应中断	227
配置中断	227
处理中断	228
ISR中的编程限制.....	228
选择一个合适的上下文参数	229
ISR的同步操作.....	229
DPC.....	230
DPC调度	232
定制DPC对象	232
一个中断驱动设备的例子	233
初始化PCI42	233
启动一个读操作	234
处理中断	236
测试PCI42	237
直接内存存取(DMA).....	239
传输策略	240
执行DMA传输	241
使用分散/聚集表的传输	246
使用GetScatterGatherList.....	248
使用系统控制器的传输	249
使用公用缓冲区	251
分配公用缓冲区	251
使用公用缓冲区的Slave模式DMA传输.....	252
使用公用缓冲区的总线主控模式DMA传输.....	252
使用公用缓冲区的注意事项	253
释放公用缓冲区	253
总线主控设备的一个例子	253
在PKTDMA中处理中断.....	254
测试PKTDMA.....	255
第八章：电源管理	256
WDM电源管理模型	257
WDM驱动程序的角色	257
设备电源状态与系统电源状态	257
电源状态转换	258
处理IRP_MJ_POWER请求	258
管理电源状态转换	262
有限状态机概述	262
新IRP的初始化处理.....	264
提升电源级别的系统电源IRP	266

处理失败	268
映射系统状态为设备状态	269
请求设备电源IRP	272
完成系统IRP	273
降低电源级别的系统电源IRP	273
设备电源IRP	276
设置更高级的设备电源状态	278
查询更高级的设备电源状态	280
设置更低级的设备电源状态	281
查询更低级的设备电源状态	283
其它电源管理细节	286
在AddDevice中设置的标志	286
设备的唤醒特征	286
何时发出WAIT_WAKE	288
空闲检测	288
指出自己没有处于空闲状态	289
空闲超时的选择	289
从空闲状态中唤醒	290
用序列号优化状态改变	291
Windows 98 兼容问题	292
DO_POWER_PAGABLE的重要性	292
请求设备电源IRP	292
PoCallDriver	292
其它不同之处	292
第九章：专门问题	294
过滤器驱动程序	295
DriverEntry例程	296
AddDevice例程	297
派遣例程	298
登记错误	301
创建错误登记包	302
创建消息文件	303
I/O控制操作	306
DeviceIoControl API	306
DeviceIoControl的同步和异步调用方式	307
定义I/O控制代码	308
处理IRP_MJ_DEVICE_CONTROL	309
BUFFERED模式	310
DIRECT模式	312
NEITHER模式	312
内部I/O控制操作	313
应用程序关注事件的通知	315
使用异步IOCTL	316
辅助例程的工作原理	317
系统线程	320
系统线程的创建与终止	320
用系统线程循检设备	321
工作项	325
IoXxxWorkItem	326
看门狗定时器	327
Windows 98 兼容问题	330
错误登记	330
IOCTL与Windows 98 虚拟设备驱动程序	330
挂起IOCTL操作时的注意事项	330
等待系统线程结束	330
第十章：Windows管理诊断	331

WMI概念	332
一个规划例子	332
WDM驱动程序与WMI	334
委托WMILIB处理IRP	335
QueryRegInfo回调函数.....	336
QueryDataBlock回调函数	338
SetDataBlock回调函数	339
SetDataItem回调函数	340
高级特征	341
处理多实例	341
实例命名	342
处理多类	343
Expensive统计	343
WMI事件	344
WMI方法例程	345
标准数据块	346
标准控制	347
用户模式程序与WMI	348
COM是什么	348
接口是什么	348
对象的创建与销毁	349
访问WMI信息	349
连接一个命名空间	350
枚举类实例	351
项目值的获取与设置	352
接收事件通知	352
调用方法例程	353
Windows 98 兼容问题	355
第十一章：USB总线	356
编程架构	357
设备层次	357
高速和低速设备	357
电源	357
设备中有什么？	358
信息流动	359
信息打包	360
端点的状态	361
控制传输	361
批量传输	364
中断传输	365
等时传输	365
描述符	365
设备描述符	366
配置描述符	367
接口描述符	368
端点描述符	368
串描述符	369
其它描述符	370
使用总线驱动程序	370
初始化请求	370
发送URB	371
URB返回的状态	372
配置	373
读取配置描述符	373
选择配置	375
寻找句柄	379

关闭设备	379
管理批量传输管道	379
错误恢复	383
管理中断管道	384
控制请求	385
控制特征	385
测定状态	386
管理等时管道	387
保留带宽	387
初始化离散的等时传输	390
获得可接受的性能	391
主IRP的取消处理	393
流等时传输	397
同步等时传输	397
第十二章：安装驱动程序	399
INF文件	400
Install段	402
定义Driver Service	405
设备标识符	405
PCI设备	405
PCMCIA设备	406
SCSI设备	407
IDE设备	408
ISAPNP设备	408
USB设备	409
1394 设备	409
通用设备标识符	409
硬件键	410
标准属性	410
非标准属性	411
INF文件工具	412
定义设备类	412
属性页程序	413
其它相关信息	416
运行应用程序	416
AutoLaunch服务	416
触发AutoLaunch	417
鸡和蛋的问题	419
运行服务	420
Windows 98 兼容问题	421
属性页提供程序	421
注册表用法	421
获得设备属性	422
应用程序执行	422
附录A: Windows 98 不兼容处理	423
为内核模式例程定义桩	424
版本兼容	425
桩函数	425
确定操作系统版本	428
附录B: 使用GENERIC.SYS	429
附录C: 使用WDMWIZ.AWX	430
基本驱动程序信息	431
DeviceIoControl代码	433
I/O资源	434
电源管理能力	435
USB端点	436

WMI支持	438
INF文件参数	440
注意事项	441
关于作者	441
Walter Oney	441
译者	441
关于电子版	441
你的信息源	442
关于本PDF电子档	442

第一章：导言

以某种观点来看，Windows 2000 或 Windows 98 都是由一个操作系统核心和多个驱动程序组成，这些驱动程序与系统中的硬件相对应。本书的内容全部都是关于驱动程序及其相关的技术。

- 操作系统概述
- Windows 2000 驱动程序
- 例子代码
- 本书的结构
- 其它资源
- 注意事项

操作系统概述

WDM 模型为存在于 Windows 98 和 Windows 2000 操作系统中的设备驱动程序提供了一个参考框架。尽管对于最终用户来说这两个操作系统非常相似，但它们的内部工作却有很大不同。在这一节，我将对这两个操作系统做一个简要描述。

Windows 2000 概述

图 1-1 是以我的视点所看到的 Windows 2000 操作系统，该图着重了驱动程序开发者所关心的特征。软件要么执行在用户模式中，要么执行在内核模式中。当用户模式程序需要读取设备数据时，它就调用 Win32 API 函数，如 **ReadFile**。Win32 子系统模块(如 KERNEL32.DLL)通过调用平台相关的系统服务接口实现该 API，而平台相关的系统服务将调用内核模式支持例程。在 **ReadFile** 调用中，调用首先到达系统 DLL(NTDLL.DLL)中的一个入口点，**NtReadFile** 函数。然后这个用户模式的 **NtReadFile** 函数接着调用系统服务接口，最后由系统服务接口调用内核模式中的服务例程，该例程同样名为 **NtReadFile**。

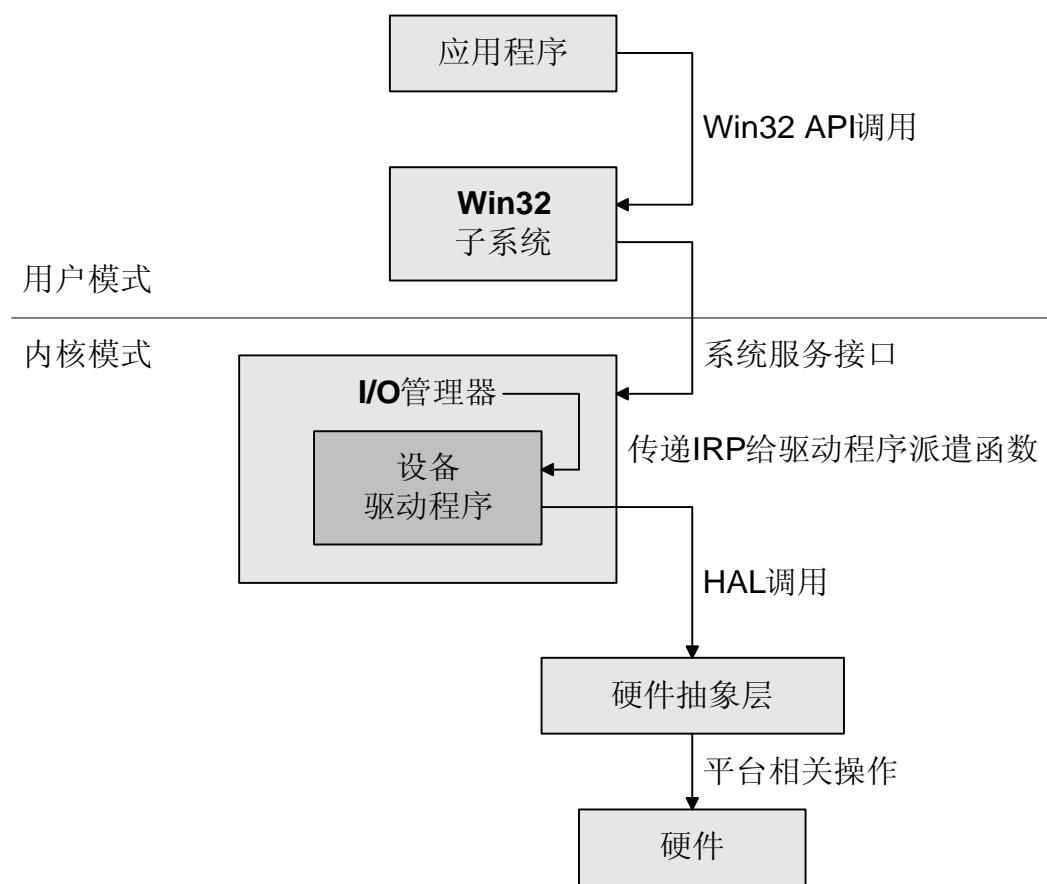


图 1-1. Windows 2000 系统结构

我们经常说 **NtReadFile** 是 I/O 管理器的一部分。“I/O 管理器(I/O Manager)”这个术语多少有些误导，系统中并不存在名为“I/O 管理器”的单独执行模块。但当我们讨论围绕在驱动程序周围的操作系统服务“云”时，我们需要使用一个名字来代表，而“I/O 管理器”就是我们通常使用的名字。

系统中还有许多与 **NtReadFile** 相似的服务例程，它们同样运行在内核模式中，为应用程序请求提供服务，并以某种方式与设备交互。它们首先检查传递给它们的参数以保护系统安全或防止用户模式程序非法存取数据，然后创建一个称为“**I/O 请求包(IRP)**”的数据结构，并把这个数据结构送到某个驱动程序的入口点。在刚才的 **ReadFile** 调用中，**NtReadFile** 将创建一个主功能代码为 **IRP_MJ_READ**(DDK 头文件中的一个常量)的 **IRP**。实际的处理细节可能会有不同，但对于 **NtReadFile** 例程，可能的结果是，用户模式调用者得到一个返回值，表明该 **IRP** 代表的操作还没有完成。用户模式程序也许会继续其它工作然后等待操作完成，或者立即进入等待状态。不论哪种方式，设备驱动程序对该 **IRP** 的处理都与应用程序无关。

执行 IRP 的设备驱动程序最后可能会访问硬件。对于 PIO 方式的设备，一个 IRP_MJ_READ 操作将导致直接读取设备的端口(或者是设备实现的内存寄存器)。尽管运行在内核模式中的驱动程序可以直接与其硬件会话，但它们通常都使用硬件抽象层(HAL)访问硬件。读操作最后会调用 READ_PORT_UCHAR 从某个 I/O 口读取单字节数据。HAL 例程执行的操作是平台相关的。在 Intel x86 计算机上，HAL 使用 IN 指令访问设备端口，在 Alpha 计算机上，HAL 使用内存提取指令访问设备实现的内存寄存器。

驱动程序完成一个 I/O 操作后，通过调用一个特殊的内核模式服务例程来完成该 IRP。完成操作是处理 IRP 的最后动作，它使等待的应用程序恢复运行。

Windows 98 概述

图 1-2 显示了 Windows 98 的基本结构。其操作系统内核称为虚拟机管理器(VMM)，因为它的主要工作就是创建“虚拟”机器，这些虚拟机器共享同一个物理机器。Windows 3.0 引入虚拟设备驱动程序(VxD)的原始目的就是为了虚化设备，以帮助 VMM 实现每个虚拟机器都拥有全部硬件的假象。VMM 架构也被引入 Windows 98，并能处理新硬件和 32 位应用程序。

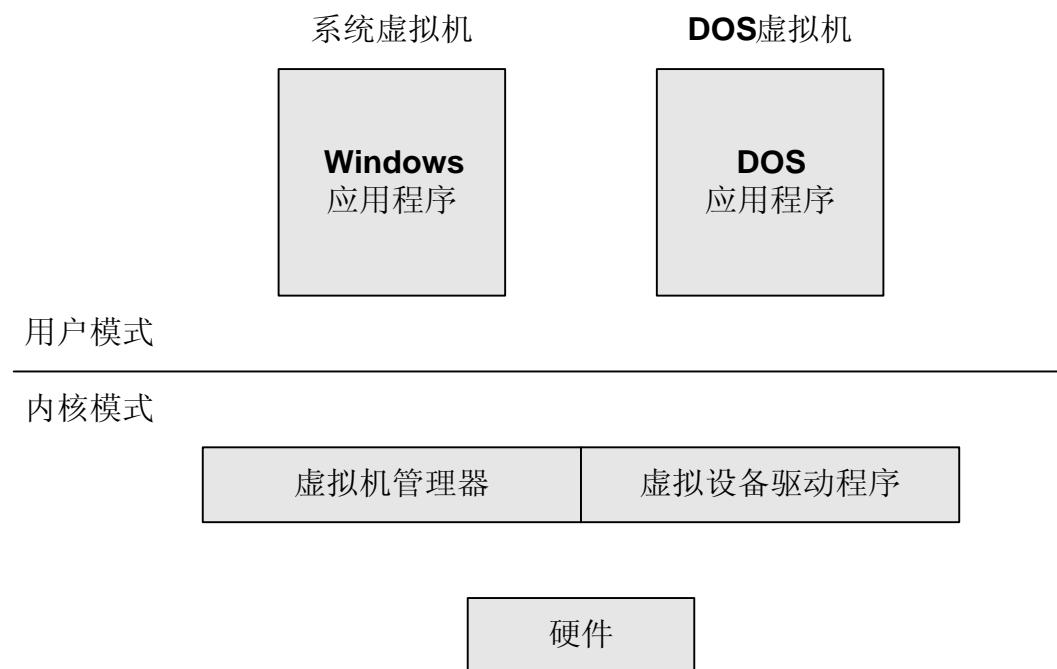


图 1-2. Windows 98 系统结构

Windows 98 不能象 Windows 2000 那样整洁地处理 I/O 操作。在处理磁盘操作、通讯口操作、键盘操作，等等方面与 Windows 2000 有很大不同。Windows 98 以两种完全不同的方式为 32 位应用程序和 16 位应用程序提供服务。见图 1-3。

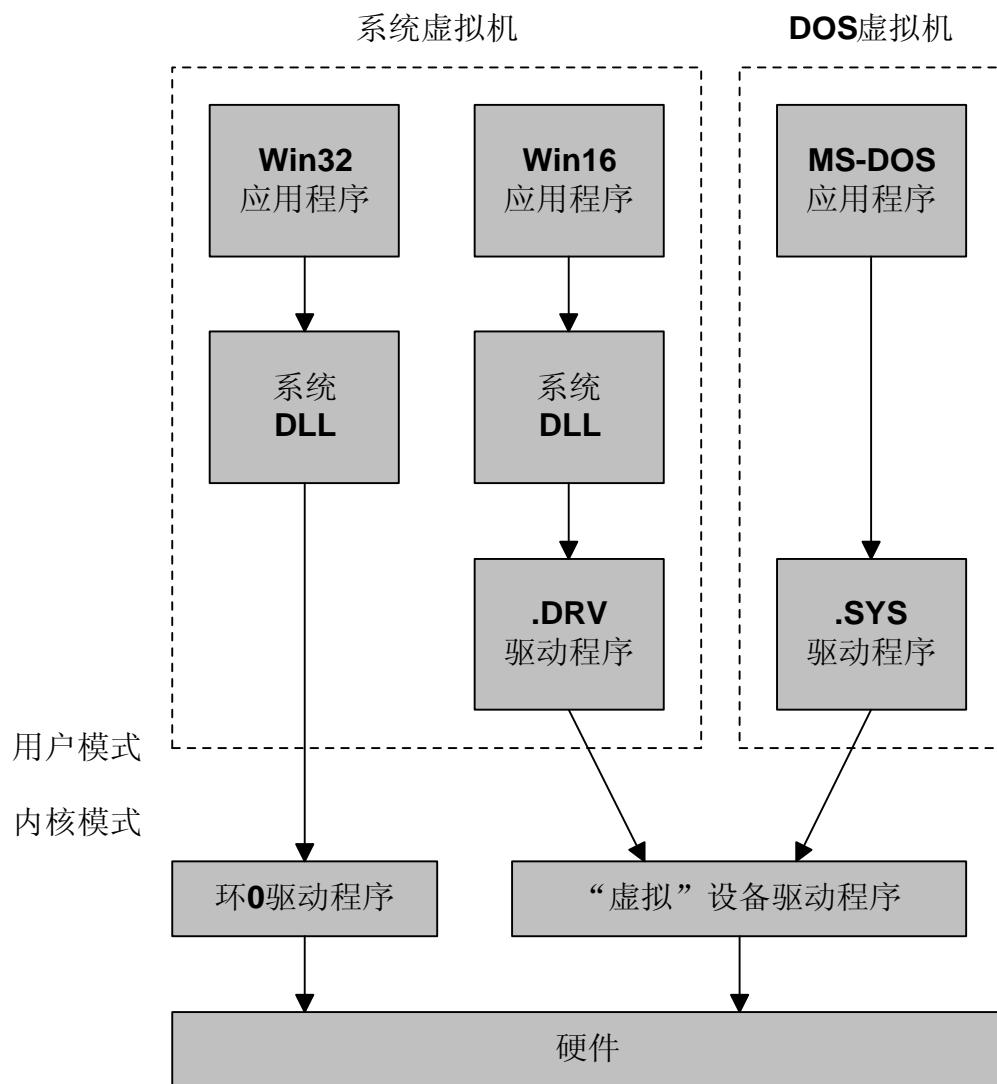


图 1-3. Windows 98 中的 I/O 请求

图 1-3 的左侧显示了 32 位应用程序的 I/O 请求处理过程。应用程序调用的 Win32API(例如 ReadFile)是系统 DLL(如 KERNEL32.DLL)中的服务例程，但应用程序仅能用 ReadFile 读磁盘文件、通讯口，和有 WDM 驱动程序的设备。对于其它种设备，应用程序必须使用基于 **DeviceIoControl** 的特殊方式。并且 Windows 98 的系统 DLL 含有与 Windows 2000 不同的代码。ReadFile 的用户模式部分(如参数检验，Windows 2000 在内核中实现)使用某个专用机制到达内核模式驱动程序。磁盘文件操作使用一种机制，串行口操作使用另一种机制，而 WDM 设备也有自己专用的机制进入内核。所有这些机制都利用软件中断 30h 来实现用户模式到内核模式的转换，但它们之间又完全不同。

图 1-3 的中间显示了 16 位 Windows 应用程序的 I/O 请求处理过程，右侧是 MS-DOS 应用程序的 I/O 请求处理过程。在这两种形式中，用户模式应用程序直接或间接地调用了用户模式的驱动程序，原理上，这些用户模式驱动程序可以直接操作机器硬件而不用其它系统部件支持。例如，Win16 程序通过调用名为 COMM.DRV 的 16 位 DLL 间接地执行串行口 I/O。(到 Windows 95 为止，COMM.DRV 仍是一个单独的驱动程序，它挂在 IRQ3 和 IRQ4 上，直接向串行口芯片发出 IN 和 OUT 指令) 虚拟通信(指虚拟机器之间的沟通)设备(VCD)驱动程序通过截获 I/O 端口操作来保证两个虚拟机不同时访问相同的端口。如果以一种神秘的方式思考这个过程，你可以这样认为，用户模式驱动程序使用了一个基于 I/O 截获操作的“API”，象 VCD 这样的“虚拟化”驱动程序就是通过冒充硬件操作来实现假 API 服务的。

Windows 2000 的所有内核模式 I/O 操作都使用一个公用的数据结构(IRP)。而 Windows 98 没有达到这样高度统一，其串行口驱动程序要遵从由 VCOMM.VXD 规定的 port 驱动程序函数调用规范，而磁盘驱动程序则遵从 IOS.VXD 实现的包驱动层次架构。其它设备类驱动程序也有其它的实现方式。

如果要把 WDM 引入 Windows 98，就必须使 Windows 98 内部架构与 Windows 2000 非常类似。Windows 98 包含了 NTKERN.VXD(VMM32.VXD)系统模块，该模块含有大量 Windows NT 内核支持函数的 Windows 实现。NTKERN.VXD 使用与 Windows 2000 相同的方式创建 IRP 并发送 IRP 到 WDM 驱动程序。实际上，WDM 驱动程序几乎区别不出这两个环境的不同。

Windows 2000 驱动程序

Windows 2000 系统可以使用多种驱动程序，图 1-4 显示了其中几种。

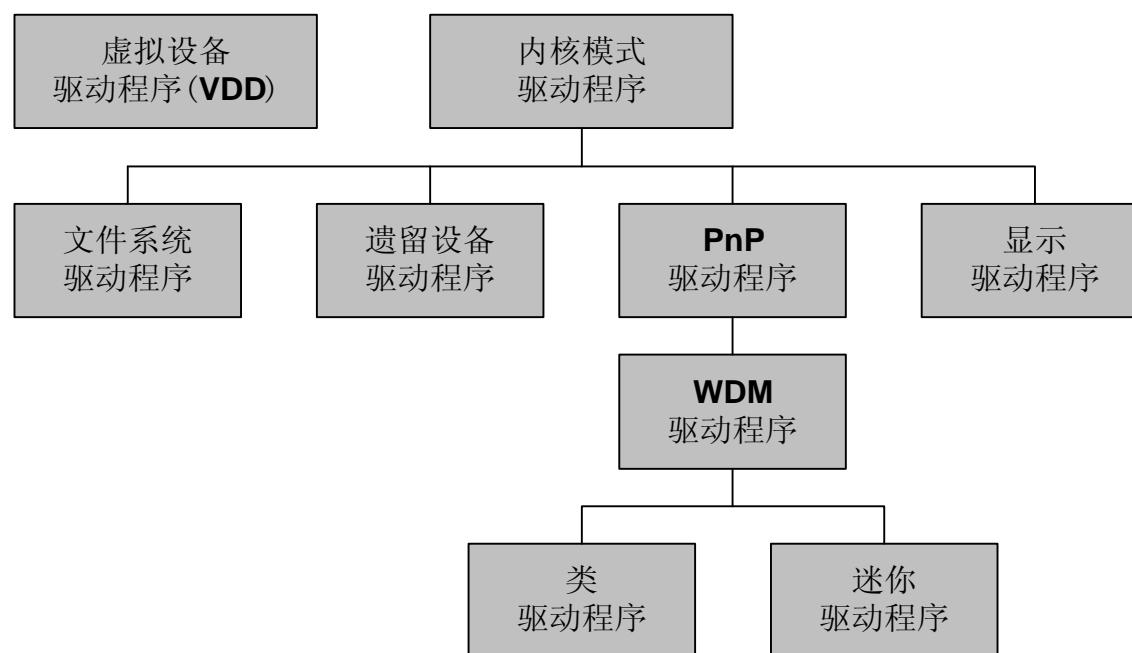


图 1-4. Windows 2000 中的设备驱动程序种类

- **虚拟设备驱动程序(VDD)**是一个用户模式部件，它可以使 DOS 应用程序访问 x86 平台上的硬件。VDD 通过屏蔽 I/O 权限掩码来捕获端口存取操作，它基本上是模拟硬件操作，这对于那些直接对裸机硬件编程的应用程序特别有用。尽管这种驱动程序在 Windows 98 和 Windows 2000 中共享一个名称并且有相同的功能，但实际上它们完全不同。我们用 VDD 缩写代表这种驱动程序，用 VxD 缩写代表 Windows 98 中的虚拟设备驱动程序以示区别。
- **内核模式驱动程序**的分类包含许多子类。**PnP 驱动程序**就是一种遵循 Windows 2000 即插即用协议的内核模式驱动程序。准确地说，本书涉及的所有内容都是面向 PnP 驱动程序的。
- **WDM 驱动程序**是一种 PnP 驱动程序，它同时还遵循电源管理协议，并能在 Windows 98 和 Windows 2000 间实现源代码级兼容。WDM 驱动程序还细分为**类驱动程序(class driver)**和**迷你驱动程序(minidriver)**，类驱动程序管理属于已定义类的设备，迷你驱动程序向类驱动程序提供厂商专有的支持。
- **显示驱动程序**是用于显示和打印设备的内核模式驱动程序。
- **文件系统驱动程序**在本地硬盘或网络上实现标准 PC 文件系统模型(包括多层次目录结构和命名文件概念)。
- **遗留设备驱动程序**也是一种内核模式驱动程序，它直接控制一个硬件设备而不用其它驱动程序帮助。这种驱动程序主要包括 Windows NT 早期版本的驱动程序，它们可以不做修改地运行在 Windows 2000 中。

内核模式驱动程序的属性

内核模式驱动程序有许多共有的属性，下面我将分别讨论这些属性。(注意，在这本书中，缩写“DDK”仅指 Windows 2000 DDK，如果要讨论其它 DDK，我将给出具体的名字)

- **可移植性**

内核模式驱动程序的源代码应该可以移植于所有 Window NT 平台。WDM 驱动程序在其定义中就规定了其源代码可以在 Windows 98 和 Windows 2000 之间相互移植。为了实现这种可移植性，驱动程序应该全部用 C 写，并且只使用 ANSI C 标准规定的语言元素。应避免使用编译器厂商专有的语言特征，并避免使用没有被操作系统内核输出的运行时间库函数(参见第三章)。如果不能避免驱动程序中的平台依赖，至少应该用条件编译指令隔离这些代码。如果严格遵循这些设计方针，那么仅需要重新编译连接源代码，生成的驱动程序就可以运行在任何新的 Windows NT 平台上。

在大多数情况下，WDM 驱动程序的二进制映像可以兼容 Windows 98 和 Windows 2000(32 位版本)。如果仅使用 WDM.H 中声明的内核模式支持函数，那么可以很容易地实现驱动程序的源代码级兼容。但操作系统之间的差异会在某些地方影响驱动程序的移植性，我将在本书的多个地方讨论这个问题。

- 可配置性

内核模式驱动程序应避免对设备特征或某些系统设置作绝对假设，这些系统设置会随着平台的改变而改变。例如，在 x86 平台上，标准串行口使用一个专用的 IRQ 和 8 个 I/O 端口，这些数值持续 20 年从未改变。把这些值直接写到驱动程序中将使驱动程序失去可配置性。在第八章，我将讨论两个用户可控制的电源管理特征——空闲检测和系统唤醒；一个总使用特定空闲超时常量的驱动程序或者只利用针对自己设备唤醒特征的驱动程序将不能实现用户可控制。这样的驱动程序将是不可配置的。

为了实现可配置性，首先应该在代码中避免直接引用硬件，即使是在平台相关的条件编译块中也是这样。应该使用 HAL 工具或调用低级总线驱动程序，或者实现一个标准的或定制的控制接口，并通过控制面板程序与用户交互。另外，还应该支持 Windows 管理仪器(WMI)控件，这种控件允许用户和管理员在分布式企业环境中配置硬件特征(见第十章)。最后，应该使用注册表作为配置信息的数据库，这可以使配置信息在系统重新启动后仍然存在。

- 可抢先性和可中断性

Windows 2000 和 Windows 98 都是多任务操作系统，可以为任意多个线程分配 CPU 时间。在大部分时间中，驱动程序例程执行在可以被其它线程(在同一个 CPU 上)抢先的环境中。线程抢先取决于线程的优先级，系统使用系统时钟为线程分配 CPU 时间片。

Windows 2000 还使用了一个中断优先级的概念，即 IRQL。我将在第四章中详细讨论 IRQL，在这里我先简单介绍一下。你可以认为 CPU 中有一个 IRQL 寄存器，它记录着 CPU 当前的执行级别。有三个 IRQL 值对设备驱动程序有重要意义：PASSIVE_LEVEL(值为 0)、DISPATCH_LEVEL(值为 2)，以及所谓的设备中断请求级 DIRQL(大于 2 的值，设备的中断服务例程在该级上执行)。在大部分时间中，CPU 执行在 PASSIVE_LEVEL 级上，所有的用户模式代码也运行在 PASSIVE_LEVEL 级上，并且驱动程序的许多活动也都发生在 PASSIVE_LEVEL 级上。当 CPU 运行在 PASSIVE_LEVEL 级时，当前运行的线程可以被任何优先级大于它的线程抢先。然而，一旦 CPU 的 IRQL 大于 PASSIVE_LEVEL 级，线程抢先将不再发生，此时 CPU 执行在使 CPU 越过 PASSIVE_LEVEL 级的任意线程上下文中。

你可以把高于 PASSIVE_LEVEL 级的 IRQL 看成是针对中断的优先级。这是一个与支配线程抢先机制不同的优先级方案，正如我前面说的，在 PASSIVE_LEVEL 级上没有线程抢先发生。但是，运行在任何 IRQL 级上的活动都可以被更高 IRQL 级上的活动中断。所以驱动程序必须假定在任何时刻都可能失去控制权，而此时系统可能需要执行更基本的任务。

- 多处理器安全

Windows 2000 可以运行在多处理器计算机上。Windows 2000 使用对称多处理器模型，即所有的处理器都是相同的，系统任务和用户模式程序可以执行在任何一个处理器上，并且所有处理器都平等地访问内存。多处理器的存在给设备驱动程序带来了一个困难的同步问题，因为执行在多个 CPU 上的代码可能同时访问共享数据或共享硬件资源。Windows 2000 提供了一个同步对象，自旋锁(spin lock)，驱动程序可以使用它来解决多处理器的同步问题。(见第四章)

- 基于对象

Windows 2000 内核是基于对象的，即驱动程序和内核例程使用的许多数据结构都有公共的特征，这些特征集中由对象管理器管理。这些特征包括名称、参考计数、安全属性，等等。在内部，内核中包含了许多执行公共对象管理的方法例程，例如打开和关闭对象或析取对象名。

驱动程序使用内核部件输出的服务例程来维护对象或对象中的域。某些内核对象，例如内核中断对象，是不透明的，DDK 头文件中没有其数据成员的声明。其它内核对象，如设备对象或驱动程序对象则是部分不透明的，

DDK 头文件中声明了其结构的全部成员，但 DDK 文档中仅描述了可访问的成员并警告驱动程序开发者不要直接访问或修改其它成员。对于驱动程序必须间接访问的不透明域，可以用支持例程访问。部分不透明的对象类似于 C++ 的类，有任何人都能访问的公共成员，还有必须通过方法函数才能访问的私有成员和保护成员。

- 包驱动

I/O 管理器和设备驱动程序使用 I/O 请求包来管理 I/O 操作的具体细节。首先，某个内核模式部件创建一个 IRP，该 IRP 可以是让设备执行一个操作、向驱动程序发送一个命令，或者向驱动程序询问某些信息的请求。然后 I/O 管理器把这个 IRP 发送到驱动程序输出的例程上。一般，每个驱动程序例程仅执行 IRP 指定的一部分工作然后返回 I/O 管理器。最后，某个驱动程序例程完成该 IRP，之后 I/O 管理器删除该 IRP 并向原始请求者报告结束状态。

- 异步

Windows 2000 允许应用程序和驱动程序在起动一个 I/O 操作后继续执行，而此时 I/O 操作仍在进行。所以，需要长时间运行的 I/O 操作应该以异步方式执行。即当驱动程序接收到一个 IRP 后，它首先初始化用于管理该 I/O 操作的任何状态信息，然后安排 IRP 的执行，而该 IRP 将在以后的某个时刻完成，最后返回调用者。由调用者决定是否等待该 IRP 的完成。

作为一个多任务操作系统，Windows 2000 根据线程的适合性和优先级来调度它们在有效处理器上的执行。通常，驱动程序在某些不可预测线程的上下文中应该使用异步方式处理 I/O 请求。我们使用术语任意线程上下文 (arbitrary thread context) 来描述驱动程序并不知道(或并不关心)处理器当前执行在哪一个线程上的上下文。驱动程序应避免阻塞任意线程，这使得驱动程序有了这样的架构：通过执行离散的操作并返回来响应硬件事件。

WDM 驱动程序模型

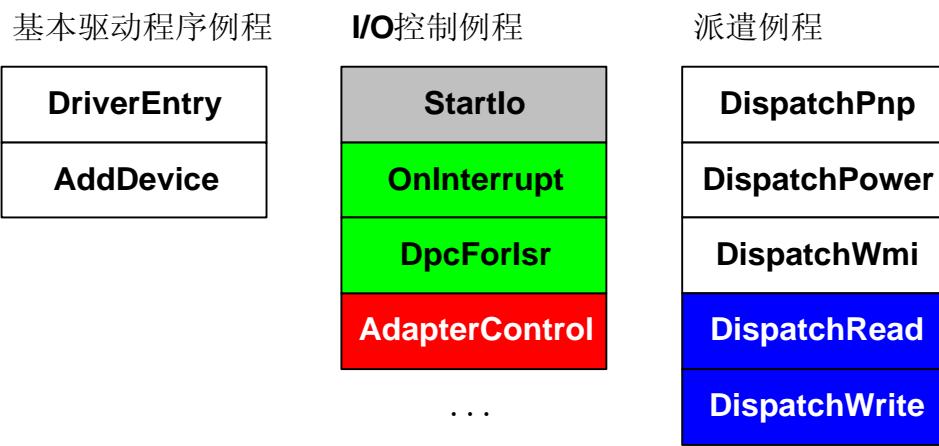
在 WDM 驱动程序模型中，每个硬件设备至少有两个驱动程序。其中一个驱动程序我们称为功能(function)驱动程序，通常它就是你认为的那个硬件设备驱动程序。它了解使硬件工作的所有细节，负责初始化 I/O 操作，有责任处理 I/O 操作完成时所带来的中断事件，有责任为用户提供一种设备适合的控制方式。

另一个驱动程序我们称为总线(bus)驱动程序。它负责管理硬件与计算机的连接。例如，PCI 总线驱动程序检测插入到 PCI 槽上的设备并确定设备的资源使用情况，它还能控制设备所在 PCI 槽的电流开关。

有些设备有两个以上的驱动程序。我们使用术语过滤器驱动程序(filter driver)来描述它们。某些过滤器驱动程序仅仅是在功能驱动程序执行 I/O 操作时进行监视。多数情况是：硬件或软件厂商利用过滤器驱动程序修改现有功能驱动程序的行为。上层过滤器驱动程序在功能驱动程序之前看到 IRP，它们有机会为用户提供额外的特征，而功能驱动程序根本不知道。有时，一个上层驱动程序可以修正功能驱动程序或硬件存在的毛病或缺陷。低层过滤器驱动程序在功能驱动程序要向总线驱动程序发送 IRP 时看到 IRP。在某些情况下，例如当 USB 设备插入 USB 总线时，低层过滤器驱动程序可以修改功能驱动程序要执行的总线操作流。

WDM 功能驱动程序通常由两个分离的执行文件组成。一个文件是类驱动程序，它了解如何处理操作系统使用的 WDM 协议(有些协议相当复杂)，以及如何管理整个设备类的基本特征。USB 照相机类驱动程序就是一个例子。另一个文件称为迷你驱动程序(minidriver)，它包含类驱动程序用于管理设备实例的厂商专有特征例程。类驱动程序和迷你驱动程序合在一起才成为一个完整的功能驱动程序。

可以把一个完整的驱动程序看作是一个容器，它包含许多例程，当操作系统遇到一个 IRP 时，它就调用这个容器中的例程来执行该 IRP 的各种操作。图 5-1 表现了这个概念。有些例程，例如 **DriverEntry** 和 **AddDevice**，还有与几种 IRP 对应的派遣函数将出现在每一个这样的容器中。需要对 IRP 排队的驱动程序一般都有一个 **StartIo** 例程。执行 DMA 传输的驱动程序应有一个 **AdapterControl** 例程。大部分能生成硬件中断的设备，其驱动程序都有一个中断服务例程(ISR)和一个推迟过程调用(DPC)例程。驱动程序一般都有几个支持不同类型 IRP 的派遣函数，其中三个派遣函数是必须的。所以，WDM 驱动程序开发者的一个任务就是为这个容器选择所需要的例程。



- 必须的驱动程序例程
- 处理请求队列需要包含**StartIo**
- 设备产生中断需要包含中断服务和**DPC**例程
- DMA**设备需要包含**AdapterControl**例程
- 可选的**IRP**派遣例程

图 1-5. WDM 驱动程序“容器”中的内容

例子代码

本书的随书光盘中包含许多驱动程序例子和测试程序。编写例子程序的目的是为了配合文中讨论的问题或技术。但是这些例子仅仅是一些“玩具”而已，你不能仅改变几行代码后就发行它们。第七章和第十一章有一些可以驱动真正硬件的驱动程序例子，这些硬件就是 PCI 芯片厂商和 USB 芯片厂商提供的开发板，除此之外，所有驱动程序都是针对假想的硬件。

几乎每个驱动程序例子都带有一个简单的用户模式测试程序，你可以用它来检查例子驱动程序的操作。这些测试程序十分小，仅包含几行涉及被测试驱动程序特殊功能的代码。

随书光盘

光盘含有每个例子的完整源代码和执行文件。此外还有几个你可能用得着的工具程序。可以在 WDMBOOK.HTM 文件中看到例子程序的索引和工具程序的使用说明。

光盘中的安装程序可以把全部例子安装到硬盘上，当然也可以直接在光盘上使用这些例子。安装程序不会向你的系统目录安装任何内核模式部件。安装程序将询问你是否可以向 AUTOEXEC.BAT 文件中加入某些环境变量。Build 程序可能需要这些环境变量。安装程序还向注册表中加入了必要的表项以定义一个 SAMPLE 设备类，所有例子驱动程序都属于这个 SAMPLE 设备类。

如果你的计算机上既有 Windows 2000 又有 Windows 98，我建议你在其中一个操作系统上做一次完全安装，在另一个操作系统上做一个简便安装。(注意，Windows 2000 不再使用 AUTOEXEC.BAT 文件初始化环境变量)

每个例子都有一个阐述如何创建和测试该例子的 HTML 文件说明，我建议你在安装一个例子驱动程序前先阅读这个说明，因为某些例子有特别的安装要求。一旦安装了某个例子驱动程序，你将在设备管理器中发现一个额外的属性页(如图 1-6)。

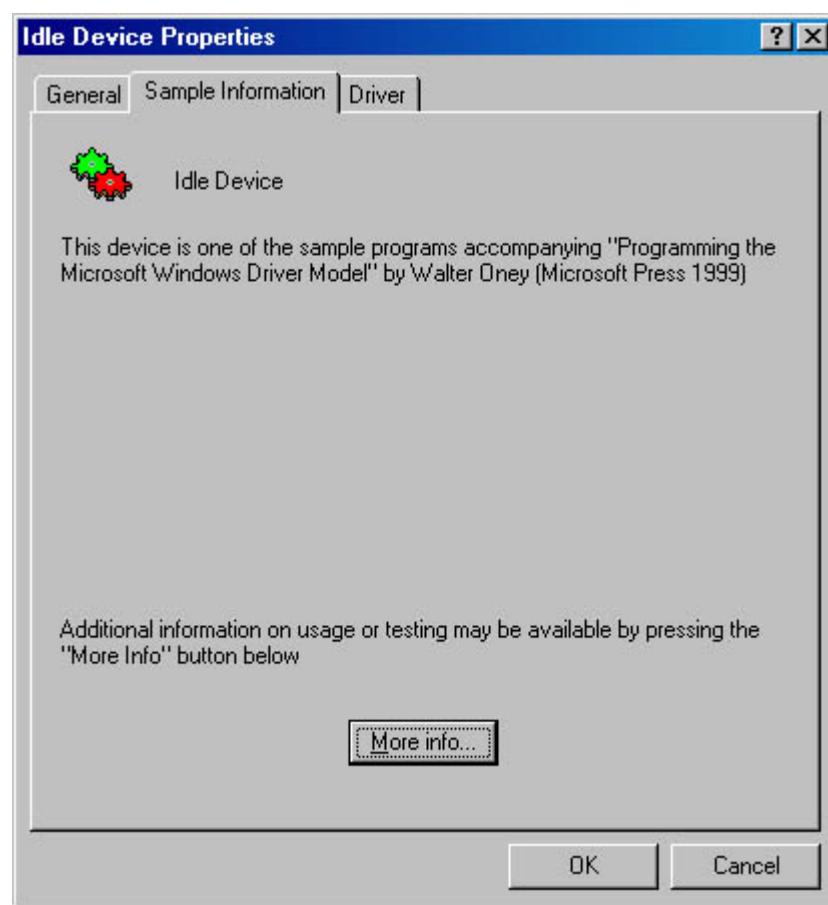


图 1-6. 某个例子驱动程序的定制属性页

关于创建例子驱动程序

有一个原因可以解释为什么我的例子程序看起来都像从一个模子里印出来的：实际上，它就是用特殊的模子印出来的，由于要写许多例子驱动程序，所以我决定写一个应用程序向导。Microsoft Visual C++6.0 的向导功能完全可以用于创建 WDM 驱动程序工程，我决定使用它。这个向导就是 WDMWIZ.AWX，你可以在随书光盘中找到它。我在附录 C 中给出了它的用法。你可以用它构造驱动程序框架。但这个向导不是一个产品级软件，其用途仅是帮助你学习驱动程序编程。另外，由于这个向导功能有限，有时你需要手动更改驱动程序工程的某些设置。更详细的信息请参考 WDMBOOK.HTM 文件。

安装了 Windows 2000 DDK 后，开始菜单上将出现 checked build environment 和 free build environment 两个菜单项。每个环境都有与驱动程序创建方法相适应的环境变量。BUILD.EXE 工具将根据不同的环境和工程描述文件 SOURCES 创建不同的驱动程序。我为每个例子驱动程序工程都提供了一个 SOURCES 文件。

我个人更喜欢用 Microsoft Visual Studio 环境管理驱动程序工程。过去我曾提倡使用 BUILD.EXE 工具，这是因为 I Microsoft 有一天会改变某些重要的编译连接选项，那样的话，任何基于 IDE 的方法都将被破坏。事实上，这种事情已经在 Windows 2000 的 beta 版中发生了。(有人决定修改沿用了十年的库文件结构，为此我就必须修改某些工程设置) 我认为使用 IDE 带来的高生产效率要比冒险在将来做某些修改更值得。

GENERIC.SYS

WDM 驱动程序中包含大量处理即插即用和电源管理的代码。这些代码很长，另人厌烦，并且容易出错。所以我在所有依赖这些代码的驱动程序中都调用了 GENERIC.SYS(内核模式 DLL)来代替这些代码。WDMWIZ.AWX 可以创建使用 GENERIC.SYS 的工程。附录 B 详细列出了 GENERIC.SYS 的输出函数。

本书的结构

在教授了几年驱动程序编程讲座之后，我渐渐发现人们通常以两种不同的方法学习新事物。一些人喜欢先学会理论然后再学习实际操作。而另一些人喜欢先学习实际操作然后再学习理论。我把前者称为演绎方式，把后者称为归纳方式。我本人更喜欢归纳方式，所以我把这本书也组织成与归纳学习法相适合的结构。

我的目标是阐述如何写设备驱动程序。具体地说，我想为你提供一个写实际驱动程序所必须的最小的背景知识，然后再过渡到更专门的主题上。这个“最小背景知识”包含的内容很广泛，它占了本书的前六章。但是一旦你跨过了第七章，其后的章节虽然重要但对于写一个能工作的驱动程序来说并不是必须的。

- 第二章，“WDM 驱动程序的基本结构”解释了 Windows 2000 用于管理 I/O 设备的基本数据结构，以及驱动程序与这些数据结构相联系的基本方式。我将讨论驱动程序对象和设备对象。还将讨论两个基本例程：`DriverEntry` 和 `AddDevice`，每个 WDM 驱动程序“容器”都包含这两个例程。
- 第三章，“基本编程技术”描述了一些最重要的服务函数，你可以调用这些函数来完成普通的编程工作。我还将讨论错误处理、内存管理，和其它一些编程技术。
- 第四章，“同步”讨论驱动程序怎样在多任务和多处理器的环境中同步访问共享数据，你将详细地学习 `IRQL`，学习用操作系统提供的各种同步原语解决同步问题。
- 第五章，“I/O 请求包”是本书的真正主题。我将解释 I/O 请求包从哪来，讨论 IRP 处理的“标准模型”。另外，我还将讨论一个更复杂的题目：`IRP 取消`，它涉及到同步问题。
- 第六章，“即插即用”仅涉及到一种类型的 I/O 请求包，即 `IRP_MJ_PNP`。PnP 管理器通过发送这种 IRP 来报告设备的配置，并在设备存在期间向驱动程序通知重要事件。多数设计良好的 PnP 驱动程序不能使用“标准模型”来处理 IRP。所以，我描述了一个名为 `DEVQUEUE` 的对象，你可以在 PnP 事件发生时使用它来正确地入队(queue)和出队(dequeue)IRP。
- 第七章，“读写数据”我们将在这一章里讨论执行 I/O 操作的驱动程序代码。我将讲述怎样从 PnP 管理器那里获得配置信息，怎样用这些信息初始化驱动程序，以使它能处理数据读写 IRP。我还给出了两个简单的驱动程序例子：一个用于 PIO 设备，一个用于总线主控的 DMA 设备。
- 第八章，“电源管理”描述了驱动程序如何参与电源管理。电源管理颇为复杂。不幸的是，你必须参与系统的电源管理协议，否则整个系统将不能正常工作。
- 第九章，“专门问题”包含了对过滤器驱动程序、错误登记、I/O 控制操作，和系统线程的讨论。
- 第十章，“Windows 管理仪器”涉及一个企业范围的计算机管理方案，你的驱动程序能够并且也应该参与这个方案。我将解释如何为监视程序提供统计数据和性能数据，怎样响应标准的 WMI 控制，怎样向控制程序报告重要事件的发生。
- 第十一章，“USB 总线”描述怎样写 USB 设备驱动程序。
- 第十二章，“安装设备驱动程序”告诉你怎样把驱动程序安装到用户系统中，你将学习 INF 文件的基本写法，还可以学到关于系统注册表的一些有趣并且有用的东西。
- 附录 A，“Windows 98 的不兼容处理”介绍一个基于 VxD 的方案，该方案允许你在 Windows 98 和 Windows 2000 平台上使用相同的驱动程序映像。由于 Windows 2000 出现在 Windows 98 之后，它输出了 Windows 98 没有输出或没有实现的服务例程，我将介绍一个短小的 VxD 程序，它可以解决这个问题。
- 附录 B，“使用 GENERIC.SYS”描述了 GENERIC.SYS 的公共接口。大部分驱动程序例子使用了这个库。
- 附录 C，“使用 WDMWIZ.AWX”描述了如何使用这个应用程序向导来创建一个驱动程序。我再提一下，`WDMWIZ.AWX` 并不是一个商品化的工具包。

关于书中的错误

我尽可能使本书不出现错误。但是想一想，当写一本含有复杂技术和许多新概念的书时，你不可能保证百分之百的正确。另外，几个月后当 Windows 2000 从 beta 版变为正式版时，WDM 不可避免地会有些改变。为此，我将在 <http://www.oneysoft.com/> 上发布一个勘误表。

其它资源

本书并不是关于驱动程序编程的唯一资源。它仅着重描述了我认为重要的特征；但你有可能需要一些我没有涉及到的信息，或者你有与我不同的学习方法。我没有讲述超出驱动程序编制需求之外的操作系统知识。如果你是一个演绎式学者，或者你仅仅想知道一些理论背景，可以参考下面列出的资源。

驱动程序开发书籍

- Art Baker, 《*The Windows NT Device Driver Book: A Guide for Programmers*》, (Prentice Hall, 1997)。
- Chris Cant, 《*Writing Windows WDM Device Drivers*》, (R&D Press, 1999)。
- Edward N. Dekker 和 Joseph M. Newcomer, 《*Developing Windows NT Device Drivers: A Programmer's Handbook*》, (Addison-Wesley, 1999)。
- Rajeev Nagar, 《*Windows NT File System Internals: A Developer's Guide*》, (O'Reilly & Associates, 1997)。
- Peter G. Viscarola 和 W. Anthony Mason, 《*Windows NT Device Driver Development*》, (Macmillan, 1998)。

其它参考书籍

- David A. Solomon, 《*Inside Windows NT, Second Edition*》, (Microsoft Press, 1998)。

杂志

- *Microsoft Systems Journal* 偶尔会有一些关于驱动程序开发的文章。

新闻组

- `comp.os.ms-windows.programmer.nt.kernel-mode` 新闻组提供了一个讨论内核模式编程问题的论坛。

讲座

我办了一个WDM编程讲座，关于讲座的信息和时间表请访问<http://www.oneysoft.com/>。这个技术专题的讲座也有其他人在办，我们以此为生。我相信你能谅解我没有为你明确指出我的竞争者的地址链接。

注意事项

出于解释说明的目的，书中出现的代码没有做任何错误检测。我遵循这样的原则：尽可能一步一步地解释复杂的题目，而不是把你淹没在太多的代码中。

随书光盘中的例子驱动程序全部做了错误检测并且有产品级驱动程序所需要的一切内容。当你要向自己的代码中加入某些东西时，应该先查阅光盘。

第二章：WDM驱动程序的基本结构

在第一章中，我介绍了 Windows 2000 和 Windows 98 操作系统的基本架构。我引入这样一个概念：设备驱动程序是一个包含了许多操作系统可调用例程的容器，这些例程可以使硬件设备执行相应的动作。本章将描述驱动程序容器应包含的基本内容、驱动程序的层次结构，还将讲述 `DriverEntry` 和 `AddDevice` 函数，这两个函数存在于每个 WDM 驱动程序中。

- 设备和驱动程序的层次结构
- `DriverEntry` 例程
- `AddDevice` 例程
- Windows 98 兼容问题

设备和驱动程序的层次结构

WDM 模型使用了如图 2-1 的层次结构。图中左边是一个设备对象堆栈。设备对象是系统为帮助软件管理硬件而创建的数据结构。一个物理硬件可以有多个这样的数据结构。处于堆栈最底层的设备对象称为物理设备对象(physical device object), 或简称为 PDO。在设备对象堆栈的中间某处有一个对象称为功能设备对象(functional device object), 或简称 FDO。在 FDO 的上面和下面还会有一些过滤器设备对象(filter device object)。位于 FDO 上面的过滤器设备对象称为上层过滤器, 位于 FDO 下面(但仍在 PDO 之上)的过滤器设备对象称为下层过滤器。

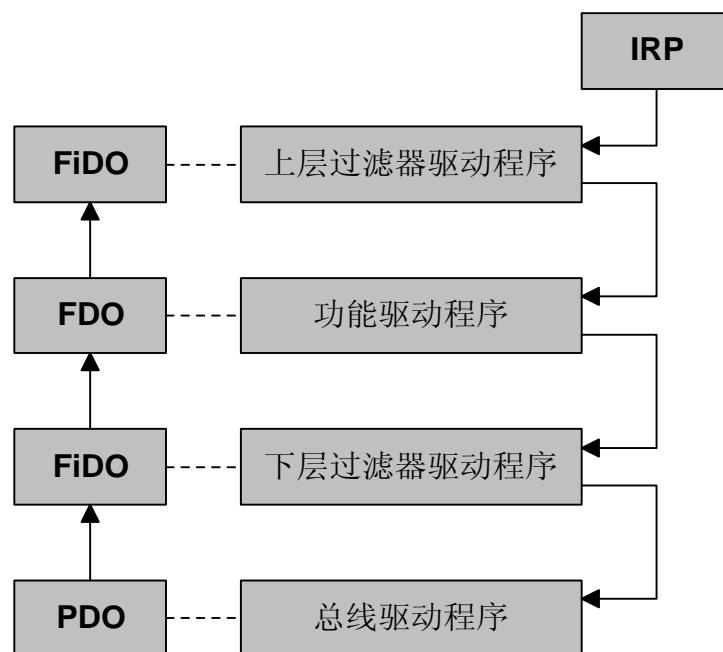


图 2-1. WDM 中设备对象和驱动程序的层次结构

关于过滤器设备对象(Filter Device Object)的缩写

我们这个行业大量使用缩写词, 我觉得这有些怪, 为什么术语过滤器设备对象(filter device object)没有官方缩写词, FDO 已经用于功能设备对象。以前, Microsoft 曾用 FiDO 表示过滤器设备对象。这个缩写的缺点是不能表明是上层过滤器还是下层过滤器。另外, 这个缩写有些不适合严肃的技术论述。例如, 在我的讲座上, 有的学生很快指出, 在堆栈上面的 FiDO 是一个 top-dog。

我将在本书中使用 FiDO 来代表过滤器设备对象。我想驱动程序编程(至少是这本书)将要 go to the dogs。

操作系统的 PnP 管理器按照设备驱动程序的要求构造了设备对象堆栈, 在本书中, 我们用通用术语“总线(bus)”来描述与设备进行电气连接的硬件。这是一个广义的定义, 它不仅包括 PCI 总线, 还包括 SCSI 卡、并行口、串行口、USB 集线器(hub), 等等。实际上, 它可以是任何能插入多个设备的硬件设备。总线驱动程序的一个任务就是枚举总线上的设备, 并为每个设备创建一个 PDO。一旦总线驱动程序检查到新硬件存在, PnP 管理器就创建一个 PDO, 之后便开始描绘如图 2-1 所示的结构。

创建完 PDO 后, PnP 管理器参照注册表中的信息查找与这个 PDO 相关的过滤器和功能驱动程序, 它们出现在图的中部。系统安装程序负责添加这些注册表项, 而驱动程序包中控制硬件安装的 INF 文件负责添加其它表项。这些表项定义了过滤器和功能驱动程序在堆栈中的次序。PnP 管理器先装入最底层的过滤器驱动程序并调用其 AddDevice 函数。该函数创建一个 FiDO, 这样就在过滤器驱动程序和 FiDO 和之间建立了水平连接。然后, AddDevice 把 PDO 连接到 FiDO 上, 这就是设备对象之间连线的由来。PnP 管理器继续向上执行, 装入并调用每个低层过滤器、功能驱动程序、每个高层过滤器, 直到完成整个堆栈。

层次结构可以使 I/O 请求过程更加明了, 见图 2-1 的右侧。每个影响到设备的操作都使用 I/O 请求包。通常 IRP 先被送到设备堆栈的最上层驱动程序, 然后逐渐过滤到下面的驱动程序。每一层驱动程序都可以决定如何处理 IRP。有时, 驱动程序不做任何事, 仅仅是向下层传递该 IRP。有时, 驱动程序直接处理完该 IRP, 不再向下传递。还有时, 驱动程序既处理了 IRP, 又把 IRP 传递下去。这取决于设备以及 IRP 所携带的内容。

在单个硬件的驱动程序堆栈中，不同位置的驱动程序扮演了不同的角色。功能驱动程序管理 FDO 所代表的设备。总线驱动程序管理计算机与 PDO 所代表设备的连接。过滤器驱动程序用于监视和修改 IRP 流。由于设备对象与驱动程序软件之间关系紧密，有时使用 FDO 驱动程序来代表功能驱动程序，用 PDO 驱动程序来代表总线驱动程序。

听我讲座的一个学生曾看到过与图 2-1 类似的图，他把图中描述的层次结构误认为是 C++ 中的类继承。设计设备驱动程序架构的一个好方法是定义基类，程序员可以从它派生出更多的专用类。以这种方式，你可以用一组抽象类来管理不同种类的 PDO，还可以用它们派生出 FDO 驱动程序。系统把 IRP 发送到虚拟函数，其中一些 IRP 被 PDO 驱动程序中的基类处理，另一些被 FDO 驱动程序中的派生类处理。但是，WDM 并不是以这种方式工作，PDO 驱动程序执行的工作与 FDO 驱动程序完全不同。FDO 通过下传 IRP 把某些工作委托给 PDO 驱动程序去做，这种关系更象链条上的环节，而不象类间的继承关系。

系统怎样装入驱动程序

介绍完 WDM 驱动程序层次结构后，我们可以更深入一步，但这里有一个明显的“鸡生蛋”问题需要解决。我曾说过总线驱动程序创建了 PDO，然后 PnP 管理器根据该 PDO 的注册表项装入它的驱动程序。那么总线驱动程序从哪来？下面我将解释这个问题。在装入和配置驱动程序过程中，注册表扮演了一个关键角色。所以我将解释与这个过程相关的注册表键，以及它的内容是什么。

递归枚举

首先，PnP 管理器有一个内建的驱动程序，它与一个实际不存在的根总线相对应。根总线概念性地把计算机与所有那些不能用电子方式声明自己存在的设备连接起来，这包括主硬件总线(如 PCI)。根总线驱动程序从注册表中获取有关计算机的信息。而这些关于计算机本身的注册表信息是由 Windows 2000 系统安装程序初始化的。安装程序通过运行一个精心制作的硬件检测程序以及向用户提出一些适当的问题来获取这些信息。所以，根总线驱动程序有足够的信息为主总线创建 PDO。

然后，主总线的功能驱动程序用电子方式枚举自己的硬件。例如，PCI 总线提供了一种访问每个插入设备的硬件配置空间的方法，配置空间保存了设备的资源需求和描述信息。当总线驱动程序枚举硬件时，它的动作就象一个普通的功能驱动程序。检测完一个硬件后，它改变角色，变成总线驱动程序并为查到的硬件创建 PDO。然后 PnP 管理器装入该 PDO 的驱动程序。有时，设备的功能驱动程序还要枚举其它硬件，如果是这样，整个过程将递归重复。结果将出现如图 2-2 的树形结构，总线设备堆栈将分叉成其它设备堆栈，每个设备堆栈代表着插入到该总线上的一个硬件设备：

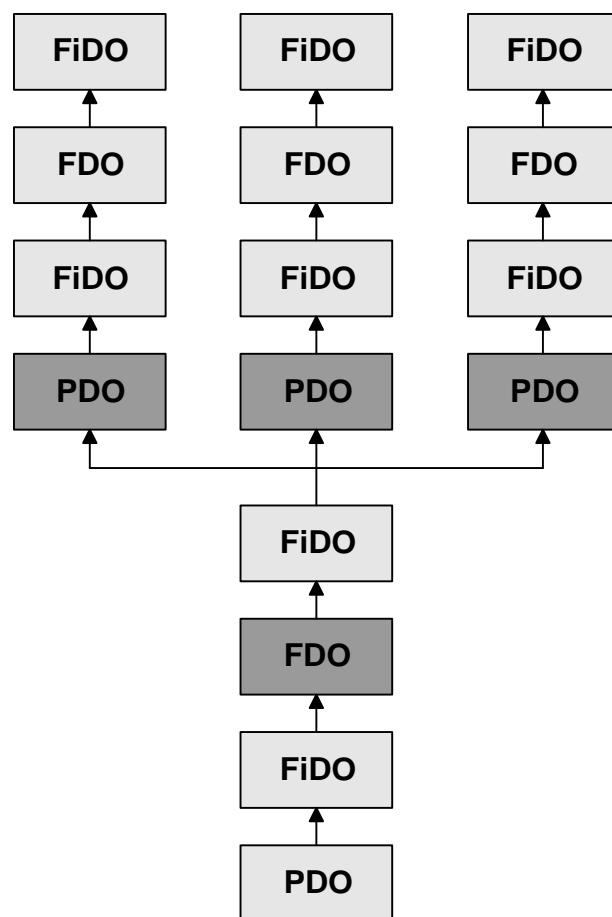


图 2-2. 设备递归枚举过程

注册表的角色

有三种注册表键负责配置。它们是硬件(hardware)键、类(class)键、服务(service)键。必须明确一点，这些名字(指 hardware、class、service)并不是某个专用子键的名称：它们是这三种键的一般称谓，其具体的路径名要取决于它们所属的设备。概括地讲，硬件键包含单个设备的信息，类键涉及所有相同类型设备的共同信息，服务键包含驱动程序信息。有时人们用“实例(instance)键”和“软件(software)键”来代表硬件键和服务键。这个命名差异是由于 Windows 95/98 与 Windows 2000 是由两个不同的项目组开发所造成的。

硬件(或实例)键 设备的硬件键出现在注册表 local machine 分支的\System\CurrentControlSet\Enum 子键上。通常你不能查看到该键的内部信息，系统只允许拥有系统帐号的用户访问该键。即只有内核模式程序和运行在系统帐号下的用户模式服务可以读写 Enum 键和其子键，但是即使是管理员也不应该直接修改这些键的内容。要想查看 Enum 键的内容，可以在 Administrator 特权级帐户下使用 REGEDIT32.EXE 工具查看。图 2-3 显示了 Enum 键的内部情况：

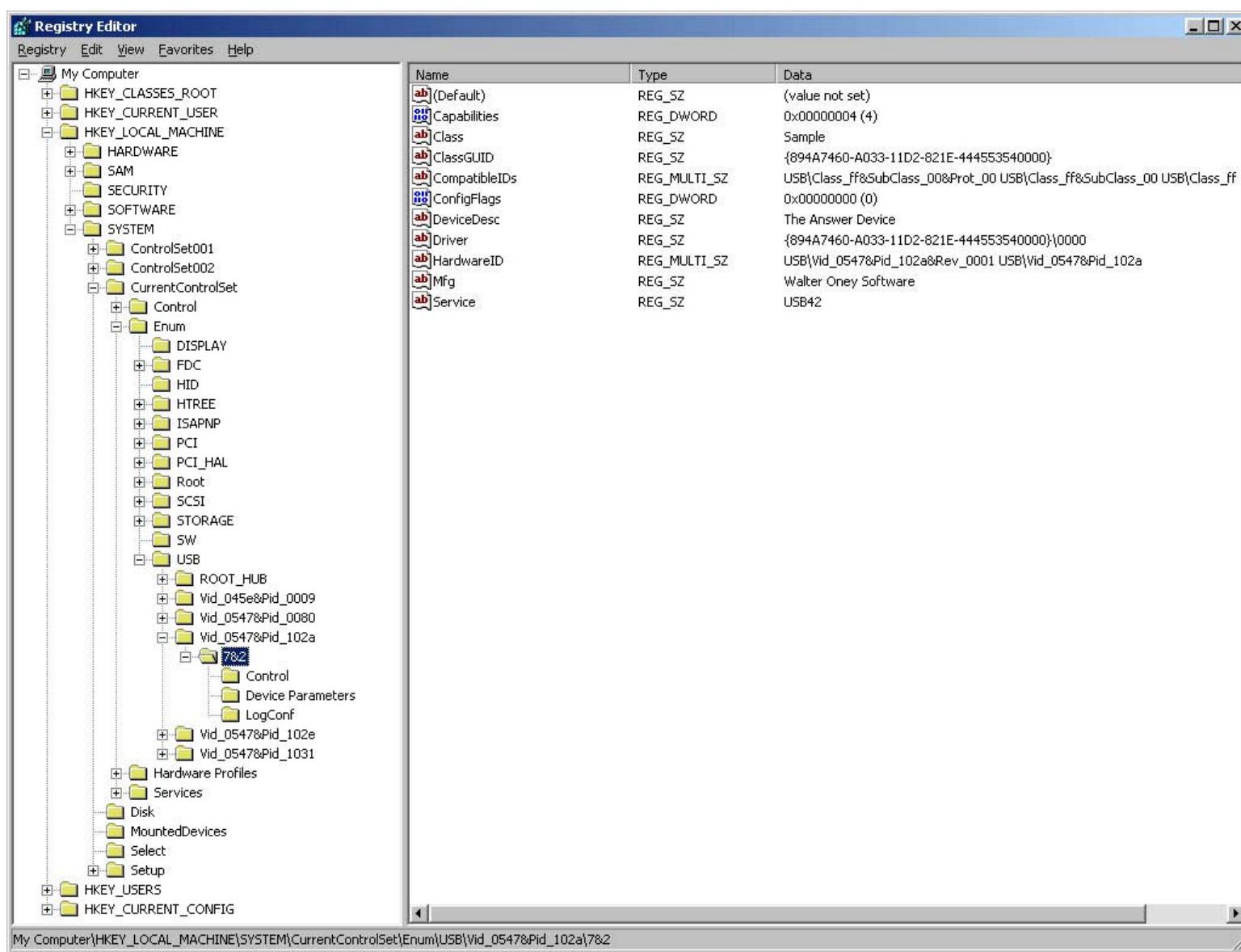


图 2-3. 注册表中的硬件键

如何命名注册表键

注册表顶级键的命名可能使第一次看到它的人感到迷惑。当你用用户模式的 Win32API 函数访问注册表时，可以用 HKEY_CLASSES_ROOT、HKEY_CURRENT_USER、HKEY_LOCAL_MACHINE，等等的预定义句柄常量代表顶级注册表键。注册表编辑器 REGEDIT.EXE 也使用这些预定义常量，如图 2-3。你还可以用缩写 HKCR、HKCU、HKLM 代替它们。即 HKCR 是 HKLM\Software\Classes 的别名，HKCU 是 HKEY_USER 的某个子键的别名，这两个别名所代表的具体键值要取决于被处理的具体情况。

在内核模式中，你应使用另一种基于内核命名空间的命名方案。在这种命名方案中，顶级键命名为 \Registry\User 和 \Registry\Machine。Machine 分支就是用户模式中的 HKLM 分支，在这里你可以找到与设备驱动程序相关的所有信息。除非另有所指，否则你应该假定本文中所引用的注册表键都在 \Registry\Machine 中。

Enum 键下的第一级子键与系统中的各种总线枚举器相对应。\\Enum\\USB 子键中包含了所有以前用过和现在存在的 USB 设备的描述。在 USB42 例子中，我将阐述怎样把设备的硬件 ID(vendor 0574, product 102A)转换成键名(Vid_0574&Pid_102A)，以及如何使有该 ID 的设备实例被表示为下一层的子键 7&2。7&2 就是该设备的硬件(或实例)键名。

硬件键中的某些键值提供的描述性信息可以被象设备管理器这样的用户模式部件所使用。图 2-4 显示了设备管理器给出的 USB42 设备属性。参见文字框“从用户模式中访问设备键”解释设备管理器如何收集这些信息，虽然它自身并不能跨过 Enum 键的正常安全保护。

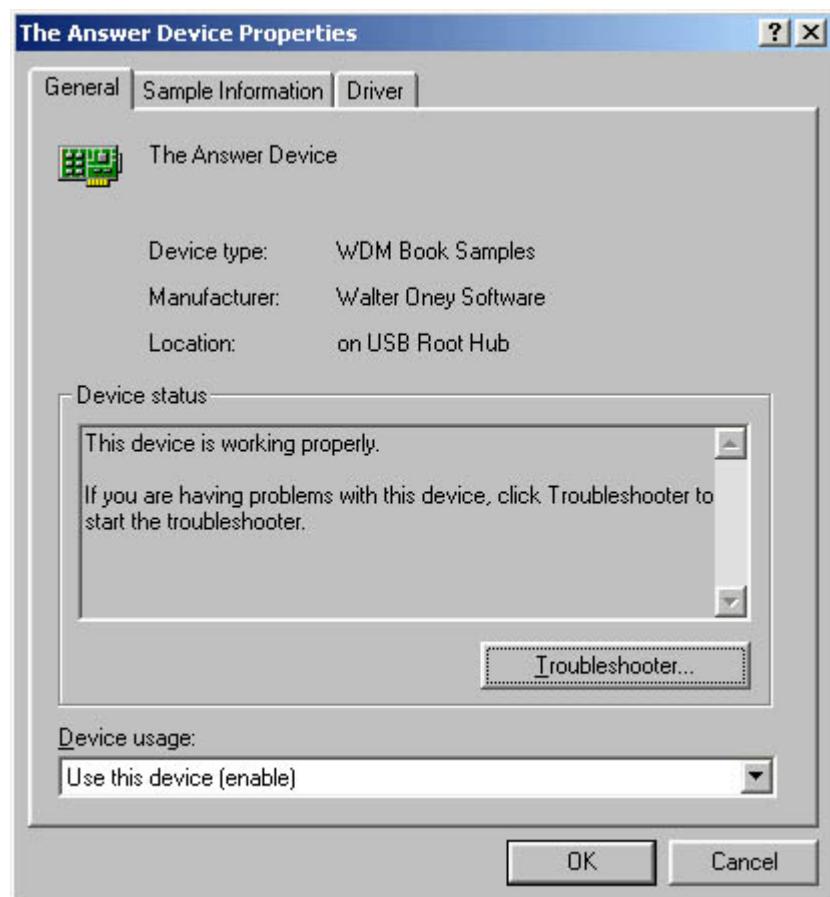


图 2-4. 设备管理器显示的设备属性页

从用户模式中访问设备键

应用程序经常需要访问注册表中关于硬件设备的信息。为了使这成为可能而同时又不暴露重要的 **Enum** 键，Microsoft 提供了一组 **SetupDiXxx** 函数。

假设你的驱动程序使用 **IoRegisterDeviceInterface** 函数寄存了一个设备接口，并且你有一个该接口的符号连接名(通过枚举该接口 GUID 的所有实例或者从 **WM_DEVICECHANGE** 消息的参数中获得这个名字)。为了从硬件键中获得 **Manufacturer** 名字，你可以使用下面代码：

```
#include <setupapi.h>
...
LPTSTR lpszDeviceName;
HDEVINFO info = SetupDiCreateDeviceInfoList(NULL, NULL);
SP_DEVICE_INTERFACE_DATA ifdata = { sizeof(SP_DEVICE_INTERFACE_DATA) };
SetupDiOpenDeviceInterface(info, lpszDeviceName, 0, &ifdata);
SP_DEVINFO_DATA did = { sizeof(SP_DEVINFO_DATA) };
SetupDiGetDeviceInterfaceDetail(info, &ifdata, NULL, 0, NULL, &did);
TCHAR buffer[256];
SetupDiGetDeviceRegistryProperty(info,
    &did,
    SPDRP_MFG,
    NULL,
    (PBYTE) mfgname,
    sizeof(mfgname),
    NULL);
SetupDiDestroyDeviceInfoList(info);
```

lpszDeviceName 是一个象“USB\Vid_0547&Pid_102A\7&2”一样的串。

硬件键还包含几个标识设备所属类和驱动程序的值。**ClassGUID** 是设备类 GUID(全局唯一标识符)的 ASCII 形式；在效果上，它是一个指向该设备类键的指针。**Service** 指向服务键。可选值(**USB42** 中没有)**LowerFilters** 和 **UpperFilters** 分别标识低层过滤器和高层过滤器的服务名。

最后，硬件键还可以有名为 **Security**、**Exclusive**、**DeviceType**、和 **DeviceCharacteristics** 的超越值，这四种值强制驱动程序创建有某种属性的设备对象。我将在后面讲述如何创建设备对象时再讨论这些超越值的重要性。

硬件键中的大部分值是在安装过程中自动填入的，或者在安装开始后的某个时刻，系统识别出了新硬件(或者经由硬件安装向导)并由系统自动填入的。有些值是由于 **INF**(硬件安装文件)中曾指明要放入这里的。我将在第十二章讨论 **INF** 文件。

类键 所有设备类的类键都出现在 **HKLM\System\CurrentControlSet\Control\Class** 键中。它们的键名是由 Microsoft 赋予的 GUID 值。图 2-5 显示了 SAMPLE 设备的类键，**USB42** 例子和本书中所有其它例子设备都属于该类。

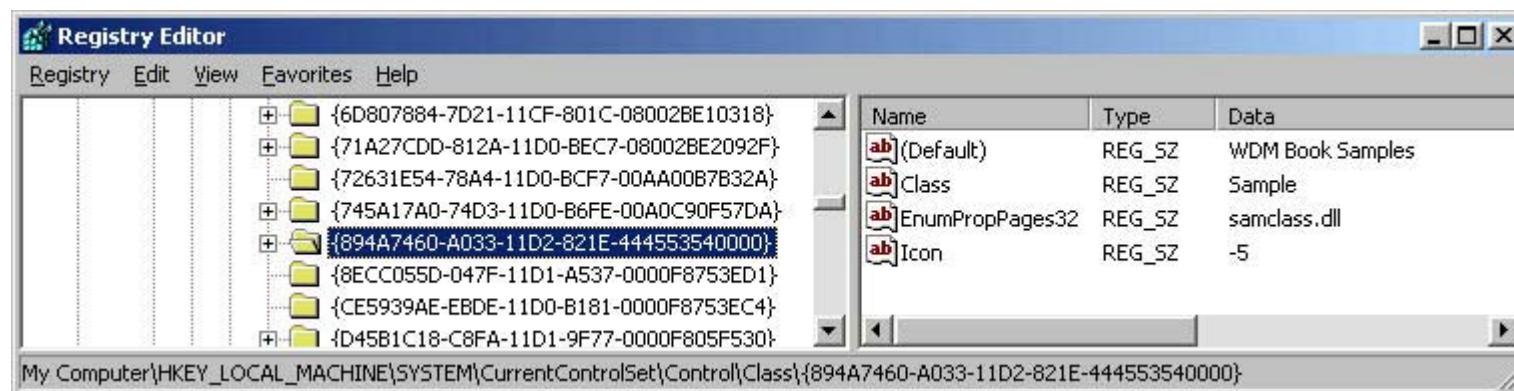


图 2-5. 注册表中的类键

Sample 类并不是特别有代表性，因为它缺少某些可选值，如：

- **LowerFilters** 和 **UpperFilters**，为该类所有设备指定过滤器驱动程序。
- **Security**、**Exclusive**、**DeviceType**，和 **DeviceCharacteristics**，如果类键的一个 **Properties** 子键中出现这些值，则指定值将超越该类所有设备对象的某些默认参数设置。但这些超越值要比硬件键中的超越值优先级低。系统管理员能通过管理控制台设置这些超越值。

每个设备还可以在所属类键下拥有自己的子键。该键的键名就是设备硬件键中的 **Driver** 值。图 2-6 显示了这种子键，这个子键的作用是把所有这些注册表项与安装设备使用的 **INF** 文件相关联。

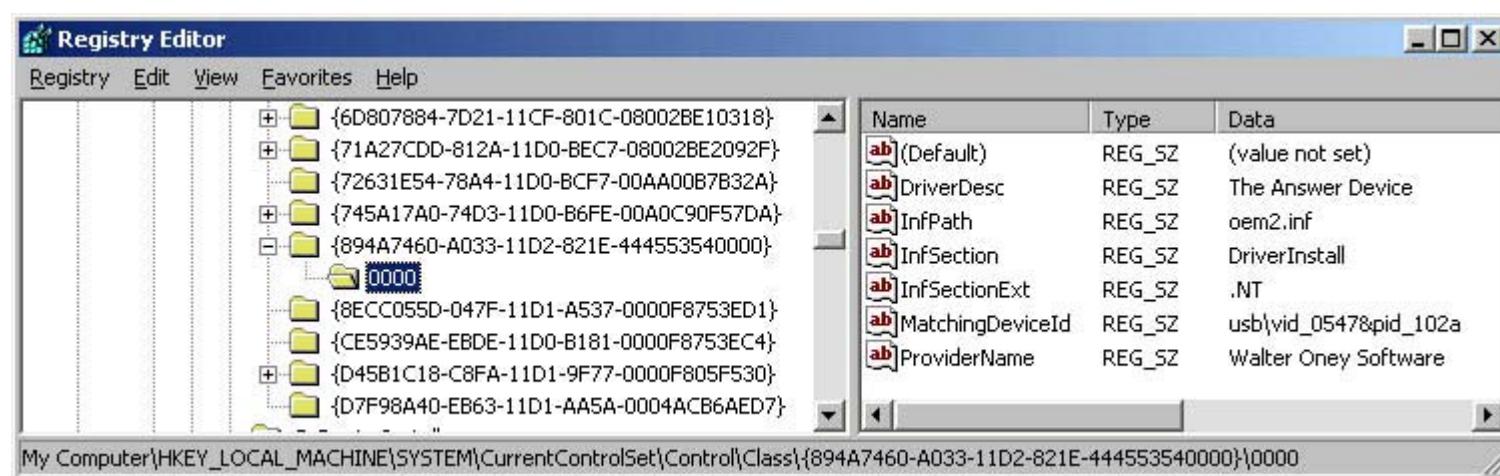


图 2-6. 存在于类键中的设备专用子键

服务(或软件)键 对设备驱动程序重要的最后一个键是服务键。它指出驱动程序执行文件的位置，以及控制驱动程序装入的一些参数。服务键位于 **HKLM\System\CurrentControlSet\Services** 键中。图 2-7 是 **USB42** 的服务键。

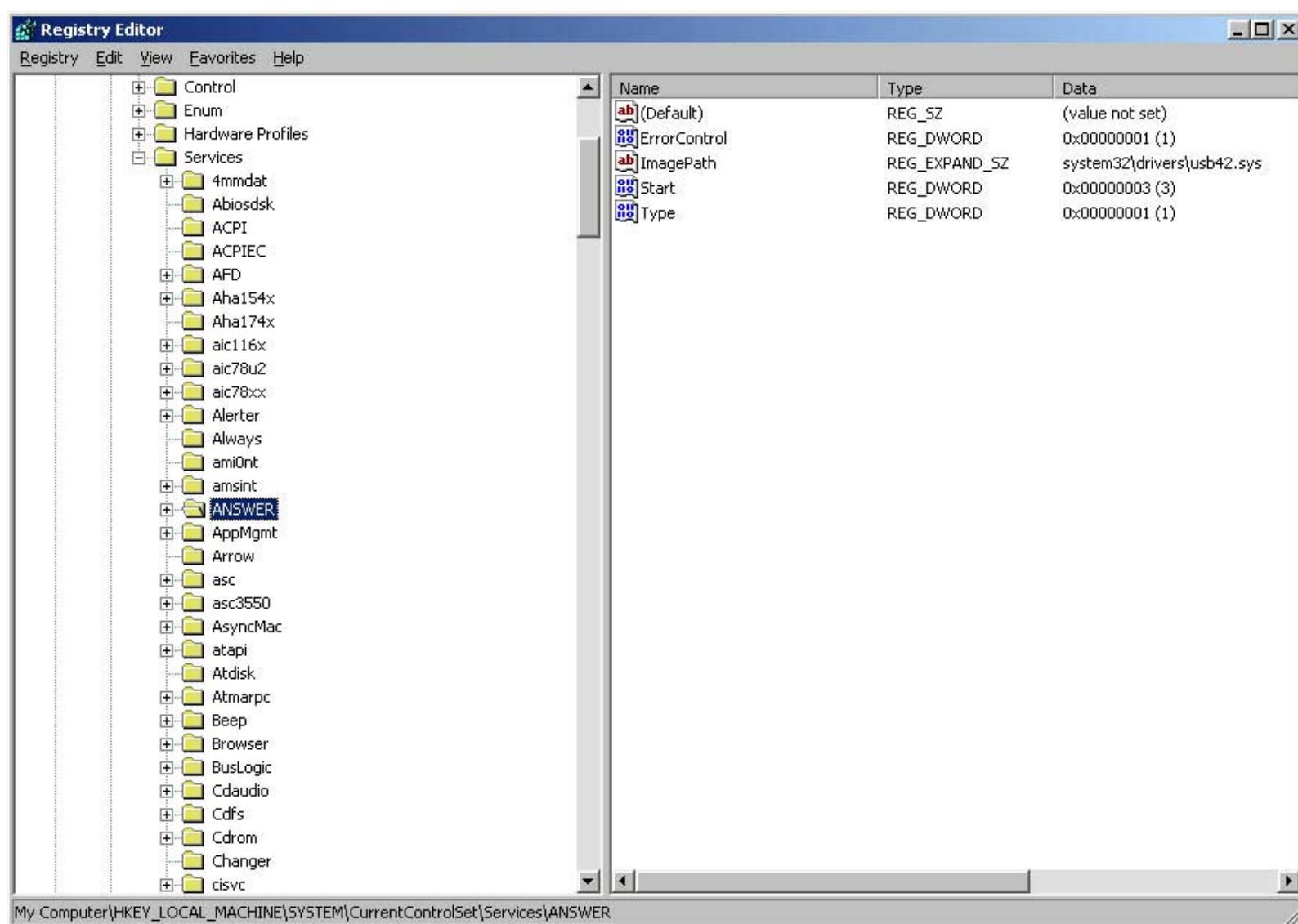


图 2-7. 注册表中的一个服务键

服务键的内容在平台 SDK 的“Service Install”中有详尽描述，因此我不想再重述。在 USB42 的例子中，这些值有如下意义：

- **ImagePath** 指出驱动程序执行文件为 USB42.SYS，路径为%SystemRoot%\system32\drivers，注意这个路径名起始于系统根目录。
- **Type(1)** 指出该表项描述一个内核模式驱动程序。
- **Start(3)** 指出系统应动态装入这个驱动程序。(该值与 CreateService 中的 SERVICE_DEMAND_START 常量对应，用于内核模式驱动程序时它代表不必明确调用 StartService 函数或发出 NET START 命令来启动驱动程序)
- **ErrorControl(1)** 指出如果装入该驱动程序失败，系统应登记该错误并显示一个消息框。

驱动程序装入顺序

当 PnP 管理器遇到一个新设备时，它打开设备的硬件键和类键，然后按如下顺序装入驱动程序：

1. 硬件键中指定的任何低层过滤器驱动程序。由于 LowerFilter 值的类型是 REG_MULTI_SZ，所以它可以指定多个驱动程序，其装入顺序由它们在串中的位置决定。
2. 类键中指定的任何低层过滤器驱动程序。同样，它也可以指定多个驱动程序，装入顺序同 1。
3. 硬件键中 Service 值指定的驱动程序。
4. 硬件键中指定的任何高层过滤器驱动程序，同样，它也可以指定多个驱动程序，装入顺序同 1。
5. 类键中指定的任何高层过滤器驱动程序，同样，它也可以指定多个驱动程序，装入顺序同 1。

当我说系统“装入”一个驱动程序时，是指系统把驱动程序的映像映射到虚拟内存中，并重定位内存参考，最后调用驱动程序的主入口点。主入口点通常命名为 DriverEntry。我将在本章后面讲述 DriverEntry 函数。如果驱动程序已经在内存中，则装入过程仅仅是增加驱动程序映像的参考计数。

你可能注意到了，属于类和设备实例的上下层过滤器驱动程序在装入过程中并没有出现象你想象的嵌套方式。在我知道这个事实前，我猜想设备级的过滤器驱动程序应比类级过滤器驱动程序更靠近功能驱动程序。但是，正如我后面讲到的，驱动程序的装入顺序并不是很重要。系统以 PnP 管理器装入驱动程序的顺序调用驱动程序的 AddDevice 函数，而这个顺序与设备对象在设备堆栈中出现的顺序完全一致。

设备对象之间如何关联

图 2-2 显示了一个由多个设备堆栈组成的树形结构，但这并不表示 IRP 必须从上一层的 PDO 流向下一层的 FiDO。实际上，一个堆栈的 PDO 驱动程序就是其下一层堆栈的 FDO 驱动程序，见图 2-2 中的阴影块。当驱动程序以 PDO 角色接收到一个 IRP 时，它对该 IRP 执行一些操作但不发出这个或其它 IRP 到设备(以 FDO 角色)。相反，当总线驱动程序以 FDO 角色接收到一个 IRP 时，它可能需要向设备发送某些 IRP(以 PDO 角色)。

举一些例子可以澄清 FiDO、FDO、PDO 之间的关系。第一个例子是 PCI 总线(PCI 总线本身是通过 PCI-to-PCI 桥芯片附着到主总线上)上的一个直接到设备的读操作。为了使问题更简单，我们假设该设备仅有一个 FiDO，如图 2-8，该操作被系统转换成一个主功能码为 IRP_MJ_READ 的请求(如何转换我们以后再讨论)。这个请求首先流到上层 FiDO，然后再传递给该设备的功能驱动程序(即图中标为 FDO_{dev} 的设备对象)。最后功能驱动程序直接调用硬件抽象层执行读操作，所以图中其它驱动程序看不到该 IRP。

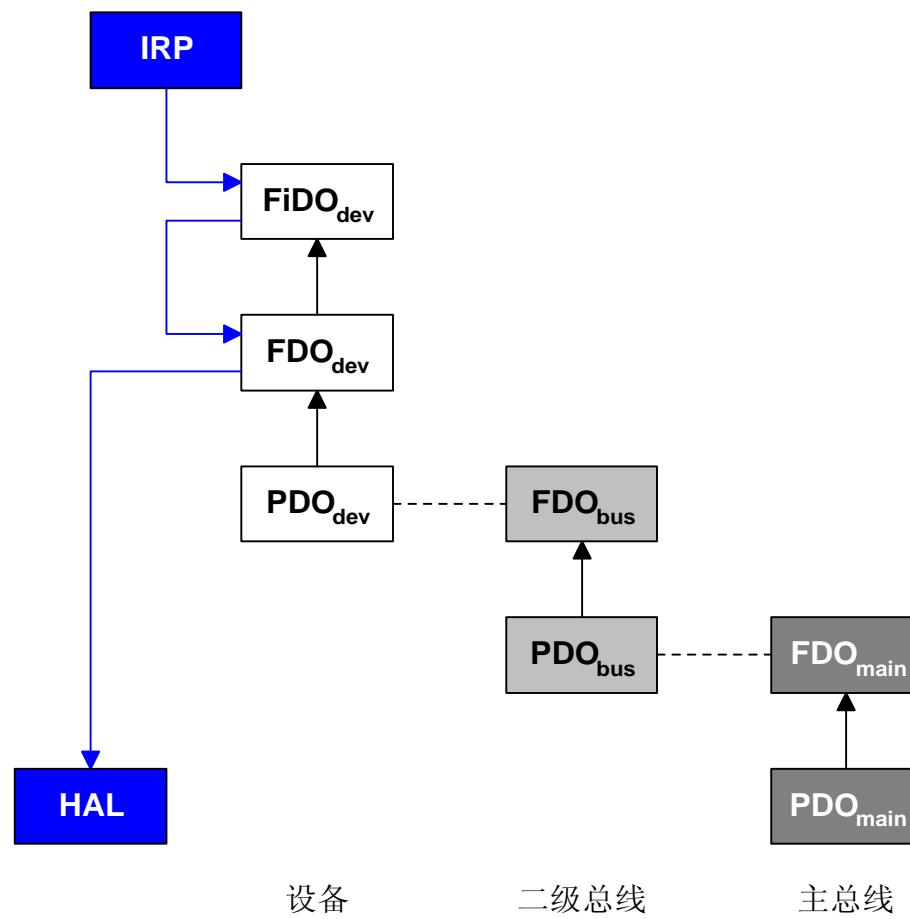


图 2-8. 设备(位于二级总线)读请求处理流程

我们对第一个例子稍做修改，如图 2-9。这次我们的读请求将发往一个 USB 设备(USB 设备插入 USB hub, USB hub 再插入主机控制器)，因此，USB 设备堆栈、hub 设备堆栈、主机控制器设备堆栈将加入设备树。

IRP_MJ_READ 请求先穿过 FiDO 到达功能驱动程序，然后功能驱动程序生成一个或多个不同种类的 IRP 并发送到自己的 PDO。USB 设备的 PDO 驱动程序就是 USBHUB.SYS，该驱动程序把这些 IRP 送到主机控制器堆栈的最上层驱动程序，跳过中间的 USB 集线器堆栈(含有两个驱动程序)。

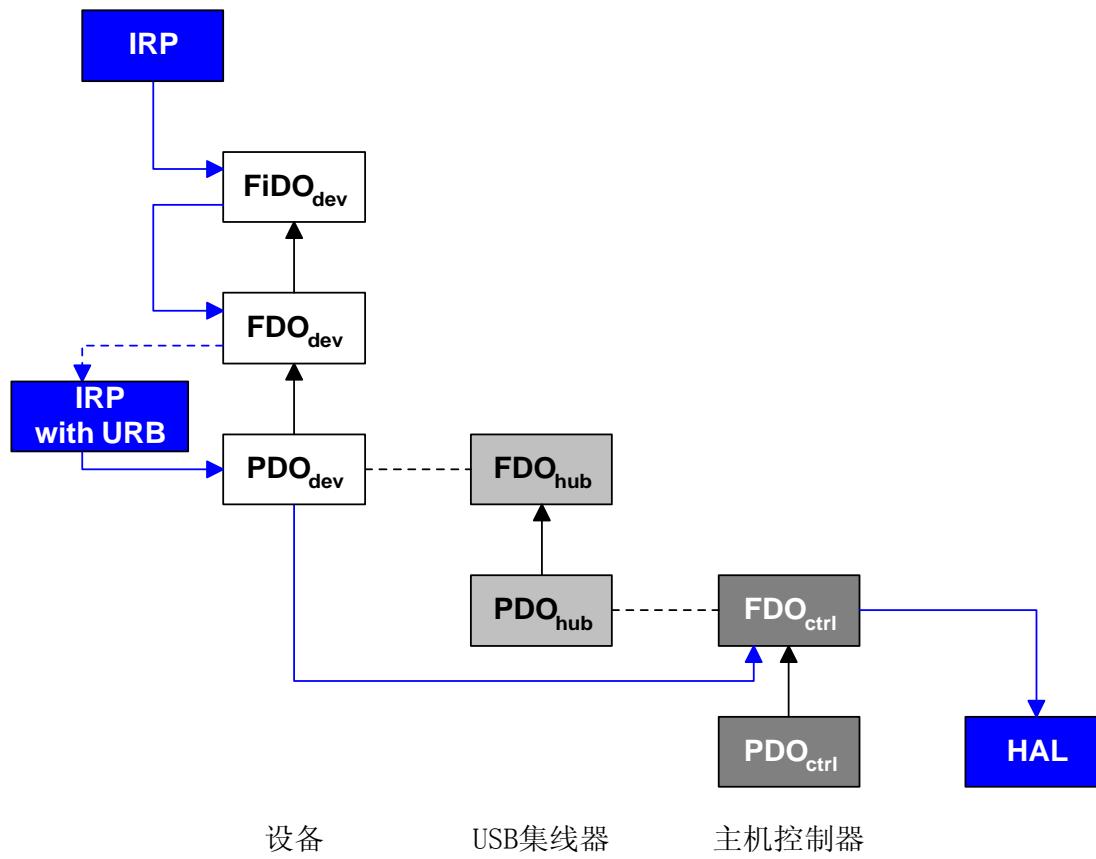


图 2-9. USB 设备的读请求处理流程

第三个例子与第一个类似，不同的是这里的 IRP 是一个通知 IRP，涉及到一个 PCI 总线上的磁盘驱动器是否被用做系统分页文件库。该通知以主功能码为 IRP_MJ_PNP 副功能码为 IRP_MN_DEVICE_USAGE_NOTIFICATION 的 IRP 形式出现。在这里，FiDO 驱动程序把请求传递到 FDO_{dev} 驱动程序，FDO_{dev} 驱动程序得到通知再把请求传递到 PDO_{dev} 驱动程序。这个通知暗示驱动程序该如何处理 PnP 或电源管理请求，所以 PDO_{dev} 驱动程序向二级总线堆栈(它的 FDO_{bus} 就是现在设备堆栈中的 PDO_{dev})发出相同的通知，如图 2-10。(注意并不是所有的总线驱动程序都这样做，但 PCI 总线是这样的)

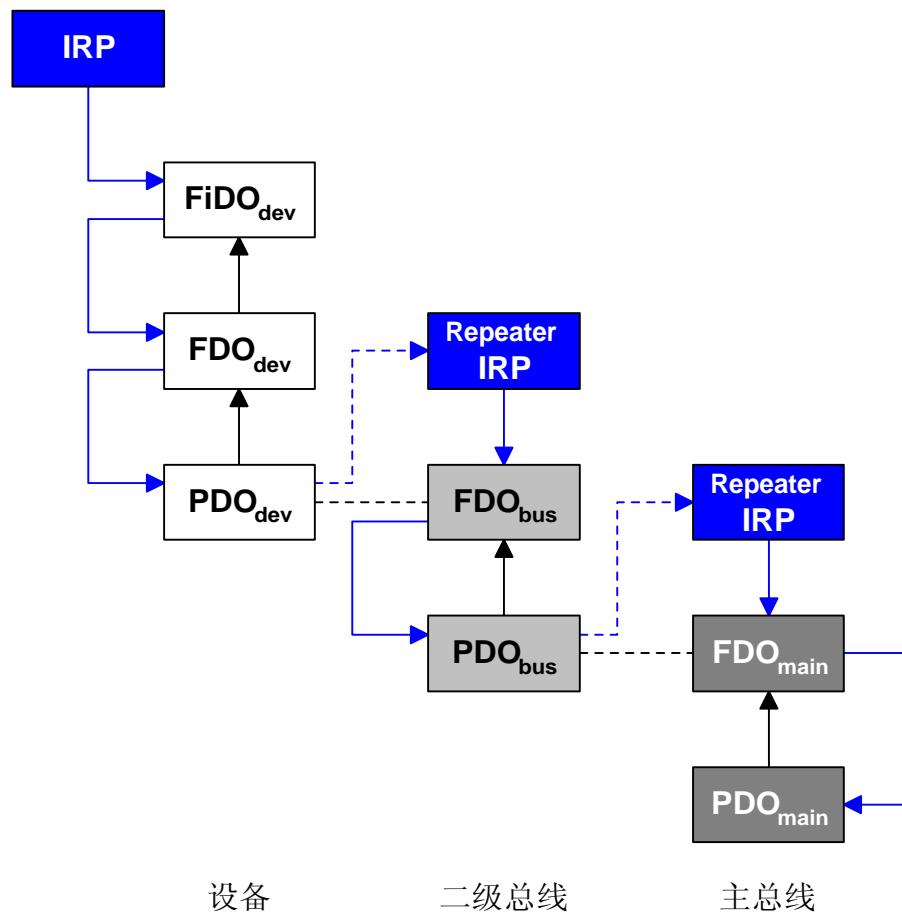


图 2-10. 一个设备用途通知的处理流程

检查设备堆

最好能形象地表现一下设备对象和驱动程序的层次结构，所以我写了一个辅助工具 DevView。把名为 USB42 的设备插入 USB hub，然后运行 DevView，屏幕上将出现图 2-11，图 2-12 所示的对话框。

这个设备仅使用两个设备对象。 PDO 由 USBHUB.SYS 管理， FDO 由 USB42.SYS(USB42 设备的驱动程序)管理。在图 2-11 中你可以看到关于 PDO 的一些信息。对照注册表中与 USB42 相关的键，现在你应该知道这些信息是从哪来的吧。

用 DevView 查看你自己的系统，可以更深刻地了解你的计算机。

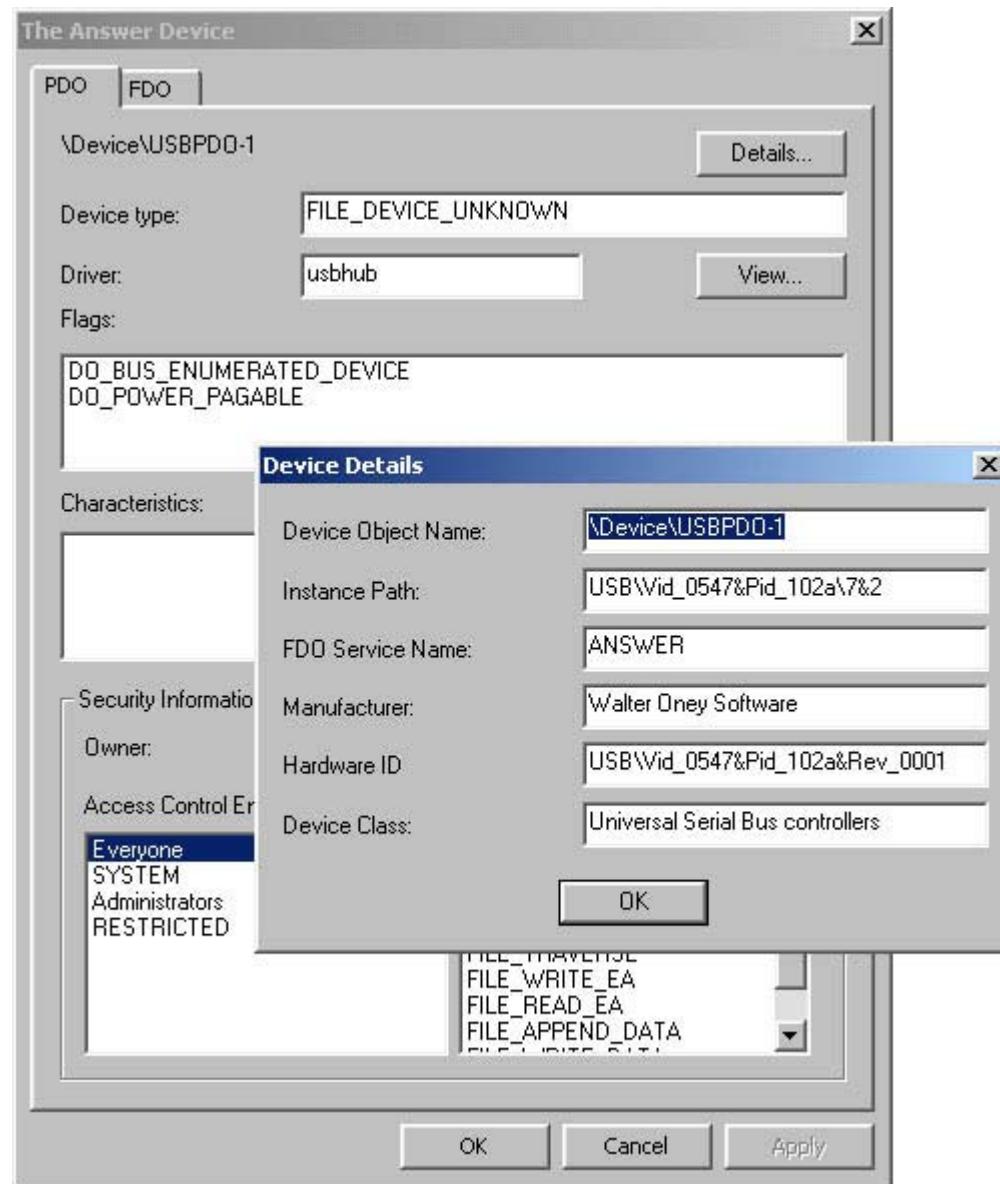


图 2-11. DevView 显示 USB42 的 PDO 信息

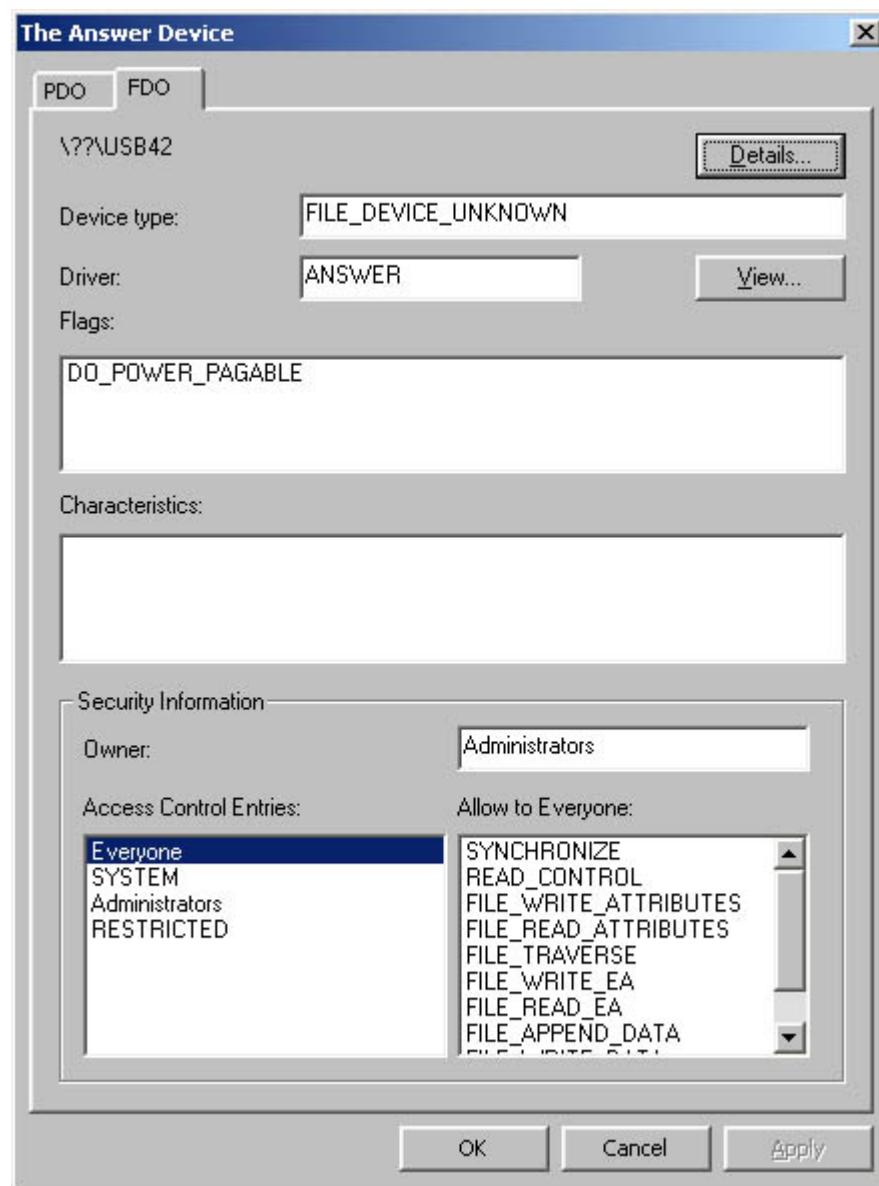


图 2-12. DevView 显示的 USB42 的 FDO 信息

驱动程序对象

I/O 管理器使用驱动程序对象来代表每个设备驱动程序，见图 2-13。就象我们将要讨论的许多数据结构一样，驱动程序对象是部分不透明的。这意味着虽然 DDK 头中公开了整个结构，但我们仅能直接访问或修改结构中的某些域。在图中，我把驱动程序对象的不透明域用灰背景表示。这些不透明域类似于 C++ 类中的私有成员或保护成员，而透明域类似于公共成员。

Type	Size
DeviceObject	
Flags	
DriverStart	
DriverSize	
DriverSection	
DriverExtension	
DriverName	
HardwareDatabase	
FastIoDispatch	
DriverInit	
DriverStartIo	
DriverUnload	
MajorFunction	

图 2-13. DRIVER_OBJECT 数据结构

DDK 头文件中声明了包括驱动程序对象在内的所有内核模式数据结构，其声明遵循如下形式(摘自 WDM.H):

```
typedef struct _DRIVER_OBJECT {  
    CSHORT Type;  
    CSHORT Size;  
    ...  
} DRIVER_OBJECT, *PDRIVER_OBJECT;
```

这就是类型名为 **DRIVER_OBJECT** 的结构的声明形式，它同时还声明了一个 **PDRIVER_OBJECT** 指针类型和结构标签 **_DRIVER_OBJECT**。这种结构声明方式在 DDK 中大量使用。DDK 头文件还声明了一组类型名(如 **CSHORT**)，这些类型名用于描述内核模式中的原子数据类型。表 2-1 列出了一些这样的类型。例如，**CSHORT** 代表用作基数的有符号短整型数。基数(**cardinal number**)用于表示数值，序数(**ordinal number**)用于表示计数器。

表 2-1. 内核模式驱动程序可以使用的公用类型名

类型名	描述
PVOID, PVOID64	通用指针(默认长度和 64 位长度)
NTAPI	用于声明服务函数，强制使用 <code>__stdcall</code> 调用约定，用于 x86 平台上
VOID	等价于“void”
CHAR, PCHAR	8 位字符，指向 8 位字符的指针(有无符号取决于编译器默认设置)
UCHAR, PUCHAR	无符号 8 位字符，无符号 8 位字符指针
SCHAR, PSCHAR	有符号 8 位字符，有符号 8 位字符指针
SHORT, PSHORT	有符号 16 位整数，有符号 16 位整数指针
USHORT, PUSHORT	无符号 16 位整数，无符号 16 位整数指针
LONG, PLONG	有符号 32 位整数，有符号 32 位整数指针
ULONG, PULONG	无符号 32 位整数，无符号 32 位整数指针
WCHAR, PWSTR	Unicode 字符，Unicode 字符串
PCWSTR	常量 Unicode 串指针
NTSTATUS	状态代码(类型为有符号长整型)
LARGE_INTEGER	有符号 64 位整数
ULARGE_INTEGER	无符号 64 位整数
PSZ, PCSZ	ASCIIZ(单字节)串指针，常量 ASCIIZ(单字节)串指针
BOOLEAN, PBOOLEAN	TRUE 或 FALSE (等价于 UCHAR)

关于 64 位类型

DDK 头中包含的类型名可以使相同的驱动程序源代码相对容易地在 Intel 32 位或 64 位平台上编译。例如，不假设长整型与指针有相同大小，我们可以声明 **LONG_PTR** 或 **ULONG_PTR** 变量。这样的变量既可以是一个长整型也可以是一个指针。这些 32/64 类型定义可以在 DDK 头文件 **BASETSD.H** 中找到。

现在我们简要地讨论一下驱动程序对象中的透明域。

DeviceObject(PDEVICE_OBJECT) 指向一个设备对象链表，每个设备对象代表一个设备。I/O 管理器把多个设备对象连接起来并维护这个域。非 WDM 驱动程序的 **DriverUnload** 函数利用这个域来遍历设备对象列表，以便删除其中的设备对象。WDM 驱动程序没有必要使用这个域。

DriverExtension(PDRIVER_EXTENSION) 指向一个不大的子结构，其中只有 **AddDevice(PDRIVER_ADD_DEVICE)** 成员可以直接访问，见图 2-14。AddDevice 是一个指针，它指向驱动程序中创建设备对象的函数；该函数内容较多，我将在本章的后面讨论它。

DriverObject
AddDevice
Count
ServiceKeyName

图 2-14. DRIVER_EXTENSION 数据结构

HardwareDatabase(PUNICODE_STRING)指向一个串，该串为设备的硬件数据库键名。内容与“\Registry\Machine\Hardware\Description\System”类似，这个注册表键保存着该设备的资源分配信息。WDM 驱动程序没有必要访问该键下的信息，因为 **PnP 管理器自动执行资源分配**。该名字以 **Unicode** 方式存储。(实际上，所有内核模式串都使用 **Unicode 编码**)，我将在下一章讨论 **UNICODE_STRING** 结构的格式和用法。

FastIoDispatch(PFAST_IO_DISPATCH)指向一个函数指针表，这些函数是由文件系统和网络驱动程序输出的。它们的使用已经超出了本书的范围。如果你想学习更多的关于文件系统驱动程序的知识，请参考 **Rajeev Nagar** 的《*Windows NT File System Internals: A Developer's Guide* (O'Reilly & Associates, 1997)》。

DriverStartIo(PDRIVER_STARTIO)指向驱动程序中处理串行 I/O 请求的函数，I/O 管理器自动为驱动程序串行化多个 I/O 请求。我将在第五章详细讨论该函数和请求排队。

DriverUnload (PDRIVER_UNLOAD)指向驱动程序中的清除函数。我将在 **DriverEntry** 函数后面讨论该函数，实际上，WDM 驱动程序根本就没有什么重要的清除工作要做。

MajorFunction (array of PDRIVER_DISPATCH)是一个函数指针表，指向存在于驱动程序中的二十多种 IRP 处理函数。这个表同样是一个大话题，因为它定义了 I/O 请求如何进入驱动程序。

设备对象

图 2-15 给出了设备对象的格式，阴影处代表不透明域。WDM 驱动程序可以调用 **IoCreateDevice** 函数创建设备对象，但设备对象的管理则由 I/O 管理器负责。

DriverObject(PDRIVER_OBJECT)指向与该设备对象相关的驱动程序对象，通常就是调用 **IoCreateDevice** 函数创建该设备对象的驱动程序对象。过滤器驱动程序有时需要用这个指针来寻找被过滤设备的驱动程序对象，然后查看其 **MajorFunction** 表项。

NextDevice(PDEVICE_OBJECT)指向属于同一个驱动程序的下一个设备对象。是这个域把多个设备对象连接起来，起始点就是驱动程序对象中的 **DeviceObject** 成员。WDM 驱动程序没有必要使用这个域。

Type	Size
ReferenceCount	
DriverObject	
NextDevice	
AttachedDevice	
CurrentIrp	
Timer	
Flags	
Characteristics	
DeviceExtension	
DeviceType	
StackSize	...
AlignmentRequirement	
	...

图 2-15. DEVICE_OBJECT 结构

CurrentIrp(PIRP)指向最近发往驱动程序 StartIo 函数的 I/O 请求包。在第五章中，当讨论取消例程时，我还要更详细地讨论这个域。

Flags(ULONG)包含一组标志位。表 2-2 列出了其中可访问的标志位。

表 2-2. DEVICE_OBJECT 结构中的 Flags 标志

标志	描述
DO_BUFFERED_IO	读写操作使用缓冲方式(系统复制缓冲区)访问用户模式数据
DO_EXCLUSIVE	一次只允许一个线程打开设备句柄
DO_DIRECT_IO	读写操作使用直接方式(内存描述符表)访问用户模式数据
DO_DEVICE_INITIALIZING	设备对象正在初始化
DO_POWER_PAGABLE	必须在 PASSIVE_LEVEL 级上处理 IRP_MJ_PNP 请求
DO_POWER_INRUSH	设备上电期间需要大电流

Characteristics(ULONG)是另一组标志位，描述设备的可选特征，见表 2-3。I/O 管理器基于 IoCreateDevice 的一个参数初始化这些标志。过滤器驱动程序沿设备堆栈向上方向传播这些标志。

表 2-3. DEVICE_OBJECT 结构中的 Characteristics 标志

标志	描述
FILE_REMOVABLE_MEDIA	可更换媒介设备
FILE_READ_ONLY_DEVICE	只读设备
FILE_FLOPPY_DISKETTE	软盘驱动器设备
FILE_WRITE_ONCE_MEDIA	只写一次设备
FILE_REMOTE_DEVICE	通过网络连接访问的设备
FILE_DEVICE_IS_MOUNTED	物理媒介已在设备中
FILE_DEVICE_SECURE_OPEN	在打开操作中检查设备对象的安全属性

DeviceExtension(PVOID)指向一个由用户定义的数据结构，该结构可用于保存每个设备实例的信息。I/O 管理器为该结构分配空间，但该结构的名字和内容完全由用户决定。一个常见的做法是把该结构命名为 DEVICE_EXTENSION。使用给定的设备对象指针 fdo 可访问这个用户结构，代码如下：

```
PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
```

也许是巧合，在内存中，设备扩展(DeviceExtension)的位置紧接在设备对象后面。但依赖这种巧合，放弃使用指针访问设备扩展却是个完全错误的想法。

DeviceType(DEVICE_TYPE)是一个枚举常量，描述设备类型。I/O 管理器基于 IoCreateDevice 的一个参数初始化这个成员。过滤器驱动程序有可能需要探测该值。在本书写作时，该成员大约有 50 多种可能值，见表 2-4。

表 2-4. 设备类型代码和默认的安全属性

设备类型	默认安全属性
FILE_DEVICE_BEEP	Public Open Unrestricted
FILE_DEVICE_CD_ROM	Modified Public Default Unrestricted
FILE_DEVICE_CD_ROM_FILE_SYSTEM	Public Default Unrestricted
FILE_DEVICE_CONTROLLER	Public Open Unrestricted
FILE_DEVICE_DATALINK	Public Open Unrestricted
FILE_DEVICE_DFS	Public Open Unrestricted
FILE_DEVICE_DISK	Modified Public Default Unrestricted

FILE_DEVICE_DISK_FILE_SYSTEM	Public Default Unrestricted
FILE_DEVICE_FILE_SYSTEM	Public Default Unrestricted
FILE_DEVICE_IMPORT_PORT	Public Open Unrestricted
FILE_DEVICE_KEYBOARD	Public Open Unrestricted
FILE_DEVICE_MAILSLOT	Public Open Unrestricted
FILE_DEVICE_MIDI_IN	Public Open Unrestricted
FILE_DEVICE_MIDI_OUT	Public Open Unrestricted
FILE_DEVICE_MOUSE	Public Open Unrestricted
FILE_DEVICE_MULTI_UNC_PROVIDER	Public Open Unrestricted
FILE_DEVICE_NAMED_PIPE	Public Open Unrestricted
FILE_DEVICE_NETWORK	Modified Public Default Unrestricted
FILE_DEVICE_NETWORK_BROWSER	Public Open Unrestricted
FILE_DEVICE_NETWORK_FILE_SYSTEM	Modified Public Default Unrestricted
FILE_DEVICE_NULL	Public Open Unrestricted
FILE_DEVICE_PARALLEL_PORT	Public Open Unrestricted
FILE_DEVICE_PHYSICAL_NETCARD	Public Open Unrestricted
FILE_DEVICE_PRINTER	Public Open Unrestricted
FILE_DEVICE_SCANNER	Public Open Unrestricted
FILE_DEVICE_SERIAL_MOUSE_PORT	Public Open Unrestricted
FILE_DEVICE_SERIAL_PORT	Public Open Unrestricted
FILE_DEVICE_SCREEN	Public Open Unrestricted
FILE_DEVICE_SOUND	Public Open Unrestricted
FILE_DEVICE_STREAMS	Public Open Unrestricted
FILE_DEVICE_TAPE	Public Open Unrestricted
FILE_DEVICE_TAPE_FILE_SYSTEM	Public Default Unrestricted
FILE_DEVICE_TRANSPORT	Public Open Unrestricted
FILE_DEVICE_UNKNOWN	Public Open Unrestricted
FILE_DEVICE_VIDEO	Public Open Unrestricted
FILE_DEVICE_VIRTUAL_DISK	Modified Public Default Unrestricted
FILE_DEVICE_WAVE_IN	Public Open Unrestricted
FILE_DEVICE_WAVE_OUT	Public Open Unrestricted
FILE_DEVICE_8042_PORT	Public Open Unrestricted
FILE_DEVICE_NETWORK_REDIRECTOR	Public Open Unrestricted
FILE_DEVICE_BATTERY	Public Open Unrestricted
FILE_DEVICE_BUS_EXTENDER	Public Open Unrestricted
FILE_DEVICE_MODEM	Public Open Unrestricted
FILE_DEVICE_VDM	Public Open Unrestricted
FILE_DEVICE_MASS_STORAGE	Modified Public Default Unrestricted
FILE_DEVICE_SMB	Public Open Unrestricted
FILE_DEVICE_KS	Public Open Unrestricted
FILE_DEVICE_CHANGER	Public Open Unrestricted
FILE_DEVICE_SMARTCARD	Public Open Unrestricted
FILE_DEVICE_ACPI	Public Open Unrestricted
FILE_DEVICE_DVD	Public Open Unrestricted
FILE_DEVICE_FULLSCREEN_VIDEO	Public Open Unrestricted

FILE_DEVICE_DFS_FILE_SYSTEM	Public Open Unrestricted
FILE_DEVICE_DFS_VOLUME	Public Open Unrestricted
FILE_DEVICE_SERENUM	Public Open Unrestricted
FILE_DEVICE_TERMSRV	Public Open Unrestricted
FILE_DEVICE_KSEC	Public Open Unrestricted

StackSize(CCHAR) 统计从该设备对象开始向下直到 PDO 之间的设备对象个数。该域的目的是告诉其它代码，如果把该设备对象的驱动程序作为其 IRP 的第一发送对象，那么应在这个 IRP 中创建多少个堆栈单元(stack location)。WDM 驱动程序通常不需要修改该值，因为创建设备堆栈的支持函数会自动完成这个任务。

如何建立设备堆栈

在 DEVICE_OBJECT 的讨论中，我指出有一个 NextDevice 域把所有属于特定驱动程序的设备对象水平地连接在一起，但我没有描述把多个设备对象垂直地连接成一个设备堆栈的方法，即从上层 FiDO 到 FDO，再到下层 FiDO，最后到 PDO。不透明域 **AttachedDevice** 就是用于这个目的。从 PDO 开始，每个设备对象都有指向更上一层设备对象的指针。由于没有已公开的向下方向的指针，所以驱动程序必须自己记住下层设备对象是谁。(实际上，**IoAttachDeviceToDeviceStack** 确实在一个结构中建立了向下方向的指针，但 DDK 中没有完全声明该结构。不要试图去查出这个结构，它随时都有可能被修改)

AttachedDevice 域故意没有公开，因为要正确地使用该域需要与删除设备对象的代码同步。但我们可以调用 **IoGetAttachedDeviceReference** 函数，该函数在给定的堆栈中寻找最上层设备对象并增加参考计数，这样可以防止该设备对象被过早地从内存中删除。如果你要自己下到 PDO，可以向设备发送一个 IRP_MJ_PNP 请求，其副功能码为 IRP_MN_QUERY_DEVICE_RELATIONS，Type 参数为 **TargetDeviceRelation**。PDO 驱动程序将返回 PDO 的地址。但是，我猜想这个 IRP 是保留给系统使用的，所以你最好不要使用这个 IRP。我们完全可以在第一次建立设备对象时记住 PDO 的地址。

同样，为了知道你下层是什么设备对象，需要在第一次把设备对象加入堆栈中时保存对象的指针。另外为了派遣 IRP，堆栈中的每个驱动程序都有可能用自己的方式实现向下指针。注意，一旦设备堆栈建立好就不要再改变它。

DriverEntry 例程

在上一节，我讲到 PnP 管理器先装入硬件需要的驱动程序，然后再调用驱动程序中的 AddDevice 函数。一个驱动程序可以被多个类似的硬件使用，但驱动程序的某些全局初始化操作只能在第一次被装入时执行一次。而 DriverEntry 例程就是用于这个目的。

DriverEntry 是内核模式驱动程序主入口点常用的名字。I/O 管理器按下面方式调用该例程：

```
extern "C" NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING  
RegistryPath)  
{  
    ...  
}
```

注意

标准 Build 脚本将驱动程序入口点定为 DriverEntry，你最好遵守这个假设，否则必须修改 Build 脚本。

在我讲述 DriverEntry 代码之前，我想谈一下函数原型本身。也许我们都不知道(除非你仔细看过 build 脚本中的编译器选项)，内核模式函数和驱动程序中的函数在为 x86 平台编译时都使用 `__stdcall` 调用约定。这虽然对编程没有任何影响，但应该记住这一点，它有利于你调试程序。我有时使用 `extern "C"` 编译指令，这是因为有时候我们需要在 C 代码中使用了 C++ 语法，C++ 语法允许在程序的任何地方声明变量而不仅仅是在左大括号后面。这个预编译指令将禁止编译器生成 C++ 形式的外部函数名修饰，这样连接器就能找到该函数。使用这个指令编译后，驱动程序入口函数的外部名将为 `_DriverEntry@8`。

关于函数原型还有一点要提的是“IN”关键字。**IN**、**OUT**、**INOUT** 在 DDK 中都被定义成空串，它们的功能就象程序注释，当你看到一个 IN 参数时，应该认定该参数是纯粹用于输入目的。OUT 参数的内容无意义，它仅用于函数的输出信息，INOUT 用于既可以输入又可以输出的参数。DDK 头文件并不真正使用这些关键字。例如 DriverEntry 例程，它的 DriverObject 指针是 IN 参数，即你不能改变这个指针本身，但你完全可以改变它指向的对象。

关于函数原型最后一点要注意的是返回值类型，DriverEntry 函数返回一个 NTSTATUS 值。NTSTATUS 实际就是一个长整型，但你应该使用 NTSTATUS 定义该函数的返回值而不是 LONG，这样代码的可读性会更好。大部分内核模式支持例程都返回 NTSTATUS 状态代码，你可以在 DDK 头文件 NTSTATUS.H 中找到 NTSTATUS 的代码列表。

DriverEntry 概述

DriverEntry 的第一个参数是一个指针，指向一个刚被初始化的驱动程序对象，该对象就代表你的驱动程序。WDM 驱动程序的 DriverEntry 例程应完成对这个对象的初始化并返回。非 WDM 驱动程序需要做大量额外的工作，它们必须探测自己的硬件，为硬件创建设备对象(用于代表硬件)，配置并初始化硬件使其正常工作。而对于 WDM 驱动程序，颇麻烦的硬件探测和配置工作由 PnP 管理器自动完成，我将在第六章讨论 PnP。如果你想知道非 WDM 驱动程序是如何初始化自身的，参见 Art Baker 的《The Windows NT Device Driver Book (Prentice Hall, 1997)》、Viscarola 和 Mason 的《Windows NT Device Driver Development (Macmillan, 1998)》。

DriverEntry 的第二个参数是设备服务键的键名。这个串不是长期存在的(函数返回后可能消失)，如果以后想使用该串就必须先把它复制到安全的地方。

对于 WDM 驱动程序的 DriverEntry 例程，其主要工作是把各种函数指针填入驱动程序对象。这些指针为操作系统指明了驱动程序容器中各种子例程的位置。它们包括下面这些指针成员(驱动程序对象中)：

- **DriverUnload** 指向驱动程序的清除例程。I/O 管理器会在卸载驱动程序前调用该例程。通常，WDM 驱动程序的 DriverEntry 例程一般不分配任何资源，所以 DriverUnload 例程也没有什么清除工作要做。
- **DriverExtension->AddDevice** 指向驱动程序的 AddDevice 函数。PnP 管理器将为每个硬件实例调用一次 AddDevice 例程。由于 AddDevice 例程对 WDM 驱动程序特别重要，所以我将在本章下一节单独讲述它。
- **DriverStartIo** 如果驱动程序使用标准的 IRP 排队方式，应该设置该成员，使其指向驱动程序的 StartIo 例程。如果你不了解什么是“标准”排队方式，不要着急，到第五章你就会完全明白，许多驱动程序都使用这种方法。
- **MajorFunction** 是一个指针数组，I/O 管理器把每个数组元素都初始化成指向一个哑派遣函数，这个哑派遣函数仅返回失败。驱动程序可能仅需要处理几种类型的 IRP，所以至少应该设置与那几种 IRP 类型相对应的指针元素，使它们指向相应的派遣函数。我将在第五章详细讨论 IRP 和派遣函数。现在，你仅需要知道至少有三种 IRP 必须处理。

下面是一个近乎完整的 DriverEntry 例程：

```
extern "C"
NTSTATUS
DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING RegistryPath)
{
    DriverObject->DriverUnload = DriverUnload;
    DriverObject->DriverExtension->AddDevice = AddDevice;
    DriverObject->DriverStartIo = StartIo;
    DriverObject->MajorFunction[IRP_MJ_PNP] = DispatchPnp;           <--2
    DriverObject->MajorFunction[IRP_MJ_POWER] = DispatchPower;
    DriverObject->MajorFunction[IRP_MJ_SYSTEM_CONTROL] = DispatchWmi;
    ...
    servkey.Buffer = (PWSTR) ExAllocatePool(PagedPool, RegistryPath->Length + sizeof(WCHAR)); <--4
    if (!servkey.Buffer)
        return STATUS_INSUFFICIENT_RESOURCES;
    servkey.MaximumLength = RegistryPath->Length + sizeof(WCHAR);
    RtlCopyUnicodeString(&servkey, RegistryPath);
    return STATUS_SUCCESS;
}
```

1. 前三条语句为驱动程序的其它入口点设置了函数指针。在这里，我用了能表达其功能的名字命名了这些函数：DriverUnload、AddDevice、StartIo。
2. 每个 WDM 驱动程序必须能处理 PNP、POWER、SYSTEM_CONTROL 这三种请求；应该在这里为这些请求指定派遣函数。在早期的 Windows 2000 DDK 中，IRP_MJ_SYSTEM_CONTROL 曾被称作 IRP_MJ_WMI，所以我把系统控制派遣函数命名为 DispatchWmi。
3. 在省略号处，你可以插入设置其它 MajorFunction 指针的代码。
4. 如果驱动程序需要访问设备的服务键，可以在这里备份 RegistryPath 串。例如，如果驱动程序要作为 WMI 生产者(见第十章)，则需要备份这个串。这里我假设已经在某处声明了一个类型为 UNICODE_STRING 的全局变量 servkey。
5. 返回 STATUS_SUCCESS 指出函数成功。如果函数失败，应该返回 NTSTATUS.H 中的一个错误代码，或者返回用户定义的错误代码。STATUS_SUCCESS 的值为 0。

DriverUnload 例程

在 WDM 驱动程序中，DriverUnload 例程的作用就是释放 DriverEntry 例程在全局初始化过程中申请的任何资源，但它几乎没什么可做。如果你在 DriverEntry 中备份了 RegistryPath 串，应该在这里释放备份所占用的内存：

```
VOID DriverUnload(PDRIVER_OBJECT DriverObject)
{
    RtlFreeUnicodeString(&servkey);
}
```

如果 DriverEntry 例程返回一个失败状态代码，系统将不再调用 DriverUnload 例程。所以，不能让 DriverEntry 例程出错后产生任何副作用，必须在它返回错误代码前消除副作用。

驱动程序再初始化例程

I/O 管理器提供了一个服务函数：**IoRegisterDriverReinitialization**。它可以为非 WDM 驱动程序解决一个奇特的问题。这里我想解释一下这个问题，从中你可以知道为什么 WDM 驱动程序不用担心问题。非 WDM 驱动程序需要在 **DriverEntry** 例程中枚举它的硬件，并在其硬件的所有可能实例被识别前装入内存并初始化。例如，鼠标和键盘设备就是这样。假定 **DriverEntry** 枚举了所有鼠标或键盘硬件并为它们创建了设备对象，但如果 **DriverEntry** 例程运行的太快，那么这些驱动程序将不能正常工作。因此，它们必须使用 **IoRegisterDriverReinitialization** 函数寄存一个例程，之后 I/O 管理器在某个驱动程序检测到新硬件存在时再回调这个寄存例程。最后“再初始化例程”运行，同时也把自身寄存为下一次回调的函数。

WDM 驱动程序不需要寄存再初始化例程，因为它们不需要用自己的代码去检测硬件。PnP 管理器自动把新硬件匹配到正确的 WDM 驱动程序上，并调用该驱动程序的 AddDevice 例程，再由 AddDevice 例程做所有必要的初始化工作。

AddDevice 例程

在前一节中，我讲述了当 WDM 驱动程序被第一次装入时如何初始化。通常，一个驱动程序可以被多个设备利用。WDM 驱动程序有一个特殊的 **AddDevice** 函数，PnP 管理器为每个设备实例调用该函数。该函数的原型如下：

```
NTSTATUS AddDevice(PDRIVER_OBJECT DriverObject, PDEVICE_OBJECT pdo)
{ }
```

DriverObject 参数指向一个驱动程序对象，就是你在 **DriverEntry** 例程中初始化的那个驱动程序对象。**pdo** 参数指向设备堆栈底部的物理设备对象。

对于功能驱动程序，其 **AddDevice** 函数的基本职责是创建一个设备对象并把它连接到以 **pdo** 为底的设备堆栈中。相关步骤如下：

1. 调用 **IoCreateDevice** 创建设备对象，并建立一个私有的设备扩展对象。
2. 寄存一个或多个设备接口，以便应用程序能知道设备的存在。另外，还可以给出设备名并创建符号连接。
3. 初始化设备扩展和设备对象的 **Flag** 成员。
4. 调用 **IoAttachDeviceToDeviceStack** 函数把新设备对象放到堆栈上。

下面我将详细解释这些步骤。

创建设备对象

调用 **IoCreateDevice** 函数创建设备对象，例如：

```
PDEVICE_OBJECT fdo;
NTSTATUS status = IoCreateDevice(DriverObject,
                                sizeof(DEVICE_EXTENSION),
                                NULL,
                                FILE_DEVICE_UNKNOWN,
                                FILE_DEVICE_SECURE_OPEN,
                                FALSE,
                                &fdo);
```

第一个参数(**DriverObject**) 就是 **AddDevice** 的第一个参数。该参数用于在驱动程序和新设备对象之间建立连接，这样 I/O 管理器就可以向设备发送指定的 IRP。

第二个参数是设备扩展结构的大小。正如我在本章前面讲到的，I/O 管理器自动分配这个内存，并把设备对象中的 **DeviceExtension** 指针指向这块内存。

第三个参数在本例中为 **NULL**。它可以是命名该设备对象的 **UNICODE_STRING** 串的地址。决定是否命名设备对象以及以什么名字命名还需要仔细考虑，我将在本节后面深入讨论这个问题。

第四个参数(**FILE_DEVICE_UNKNOWN**) 是表 2-4 中列出的设备类型。这个值可以被设备硬件键或类键中的超越值所替代，如果这两个键都含有该参数的超越值，那么硬件键中的超越值具有更高的优先权。对于属于某个已存在类的设备，必须在这些地方指定正确的值，因为驱动程序与外围系统的交互需要依靠这个值。另外，设备对象的默认安全设置也依靠这个设备类型值。

第五个参数(**FILE_DEVICE_SECURE_OPEN**) 为设备对象提供 **Characteristics** 标志(见表 2-3)。这些标志主要关系到块存储设备(如软盘、CDROM、Jaz 等等)。未公开标志位 **FILE_AUTOGENERATED_DEVICE_NAME**

仅用于内部使用，并不是 DDK 文档忘记提到该标志。这个参数同样也能被硬件键或类键中的对应值超越，如果两个值都存在，那么硬件键中的超越值具有更高的优先权。

第六个参数(**FALSE**) 指出设备是否是排斥的。通常，对于排斥设备，I/O 管理器仅允许打开该设备的一个句柄。这个值同样也能被注册表中硬件键和类键中的值超越，如果两个超越值都存在，硬件键中的超越值具有更高的优先权。

注意

排斥属性仅关系到打开请求的目标是命名设备对象。如果你遵守 Microsoft 推荐的 WDM 驱动程序设计方针，没有为设备对象命名，那么打开请求将直接指向 PDO。PDO 通常不能被标记为排斥，因为总线驱动程序没有办法知道设备是否需要排斥特征。把 PDO 标为排斥的唯一的机会在注册表中，即设备硬件键或类键的 Properties 子键含有 **Exclusive** 超越值。为了完全避免依赖排斥属性，你应该利用 IRP_MJ_CREAT 例程弹出任何有违规行为的打开请求。

第七个参数(**&fdo**) 是存放设备对象指针的地址，**IoCreateDevice** 函数使用该变量保存刚创建设备对象的地址。

如果 **IoCreateDevice** 由于某种原因失败，则它返回一个错误代码，不改变 **fdo** 中的值。如果 **IoCreateDevice** 函数返回成功代码，那么它同时也设置了 **fdo** 指针。然后我们进行到下一步，初始化设备扩展，做与创建新设备对象相关的其它工作，如果在这之后又发现了错误，那么在返回前应先释放刚创建的设备对象并返回状态码。见下面例子代码：

```
NTSTATUS status = IoCreateDevice(...);
if (!NT_SUCCESS(status))
    return status;
...
if (<some other error discovered>)
{
    IoDeleteDevice(fdo);
    return status;
}
```

NTSTATUS 状态代码和 **NT_SUCCESS** 宏的解释见下一章。

为设备命名

Windows NT 使用对象管理器集中管理大量的内部数据结构，包括我们讨论过的驱动程序对象和设备对象。David Solomon 在《*Inside Windows NT, Second Edition* (Microsoft Press, 1998)》的第三章“System Mechanisms”中给出了关于 Windows NT 对象管理器和命名空间的一个比较完整的阐述。对象都有名称，对象管理器用一个层次化的命名空间来管理这些名称。图 2-16 是 DevView 显示的顶层对象名。图中以文件夹形式显示的对象是目录对象，它可以包含子目录或常规对象，其它图标则代表正常对象。(从这一点上看，DevView 与平台 SDK 中的 WINOBJ 工具相类似，但 WINOBJ 不能给出设备对象和驱动程序的相关信息)

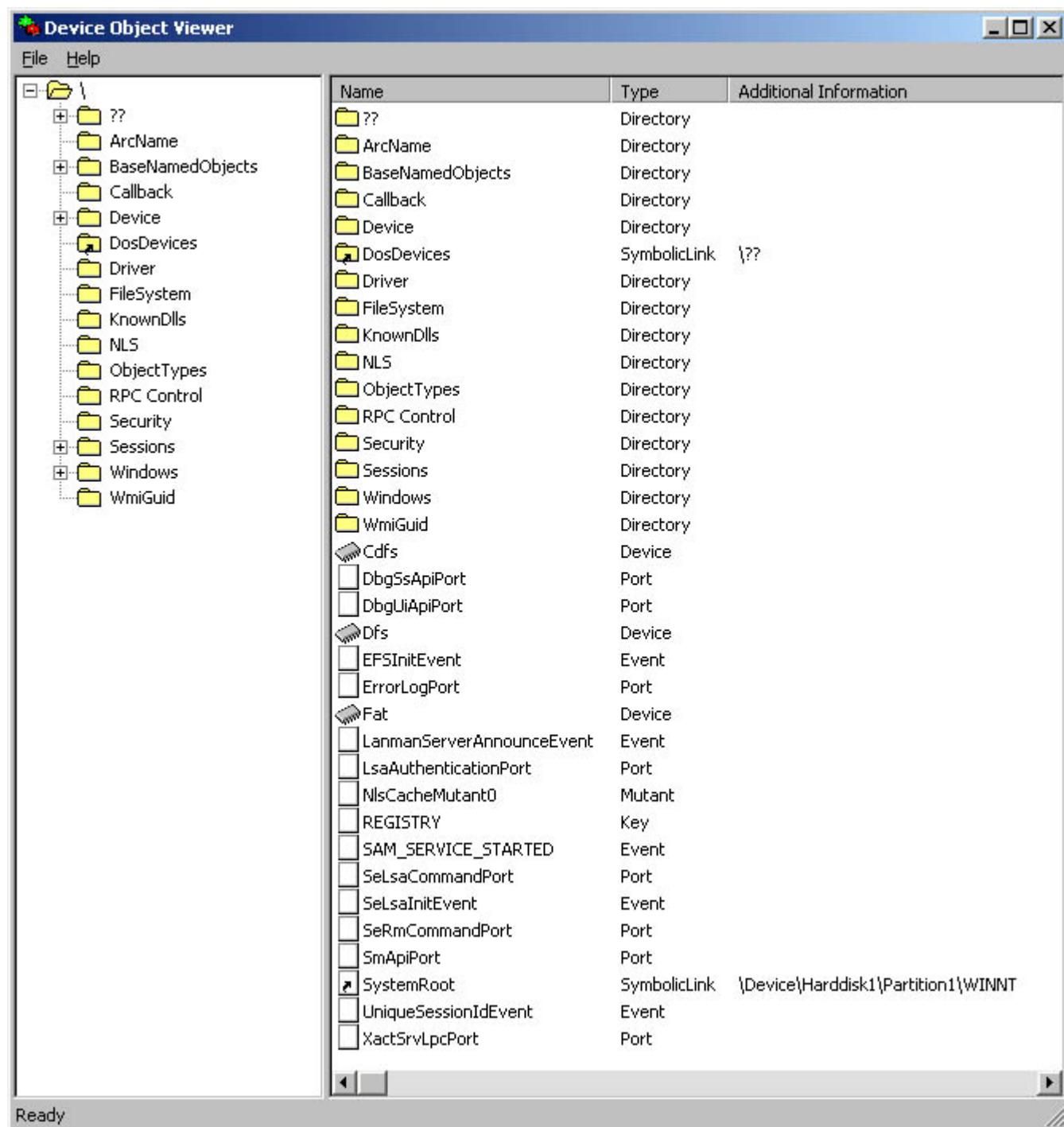


图 2-16. 用 DevView 观察命名空间

通常设备对象都把自己的名字放到\Device 目录中。在 Windows 2000 中，设备的名称有两个用途。第一个用途，设备命名后，其它内核模式部件可以通过调用 **IoGetDeviceObjectPointer** 函数找到该设备，找到设备对象后，就可以向该设备的驱动程序发送 IRP。

另一个用途，允许应用程序打开命名设备的句柄，这样它们就可以向驱动程序发送 IRP。应用程序可以使用标准的 **CreateFile API** 打开命名设备句柄，然后用 **ReadFile**、**WriteFile**，和 **DeviceIoControl** 向驱动程序发出请求。应用程序打开设备句柄时使用\\路径前缀而不是标准的 UNC(统一命名约定)名称，如 C:\MYFILE.CPP 或\\FRED\C-Drive\HISFILE.CPP。在内部，I/O 管理器在执行名称搜索前自动把\\转换成\\??\\。为了把\\??目录中的名字与名字在其它目录(例如，在\Device 目录)中的对象相连接，对象管理器实现了一种称为符号连接(symbolic link)的对象。

符号连接

符号连接有点象桌面上的快捷方式，符号连接在 Windows NT 中的主要用途是把处于列表前面的 DOS 形式的名称连接到设备上。图 2-17 显示了\\??目录的部分内容，这里就有一些符号名，例如，“C:”和其它一些用 DOS 命名方案命名的驱动器名称，它们被连接到\Device 目录中，而这些设备对象的真正名称就放在\Device 目录中。符号连接可以使对象管理器在分析一个名称时能跳到命名空间的某个地方。例如，如果我用 **CreateFile** 打开名为“C:\MYFILE.CPP”的对象，对象管理器将以下面过程打开该文件：

1. 内核模式代码最开始看到的名称是\??\C:\MYFILE.CPP。对象管理器在根目录中查找“??”。
2. 找到\??目录后，对象管理器在其中查找“C:”。它发现找到的对象是一个符号连接，所以它就用这个符号连接组成一个新的内核模式路径名：\Device\HarddiskVolume1\MYFILE.CPP，然后析取它。
3. 使用新路径名后，对象管理器重新在根目录中查找“Device”。
4. 找到\Device 目录后，对象管理器在其中查找“HarddiskVolume1”，最后它找到一个以该名字命名的设备。

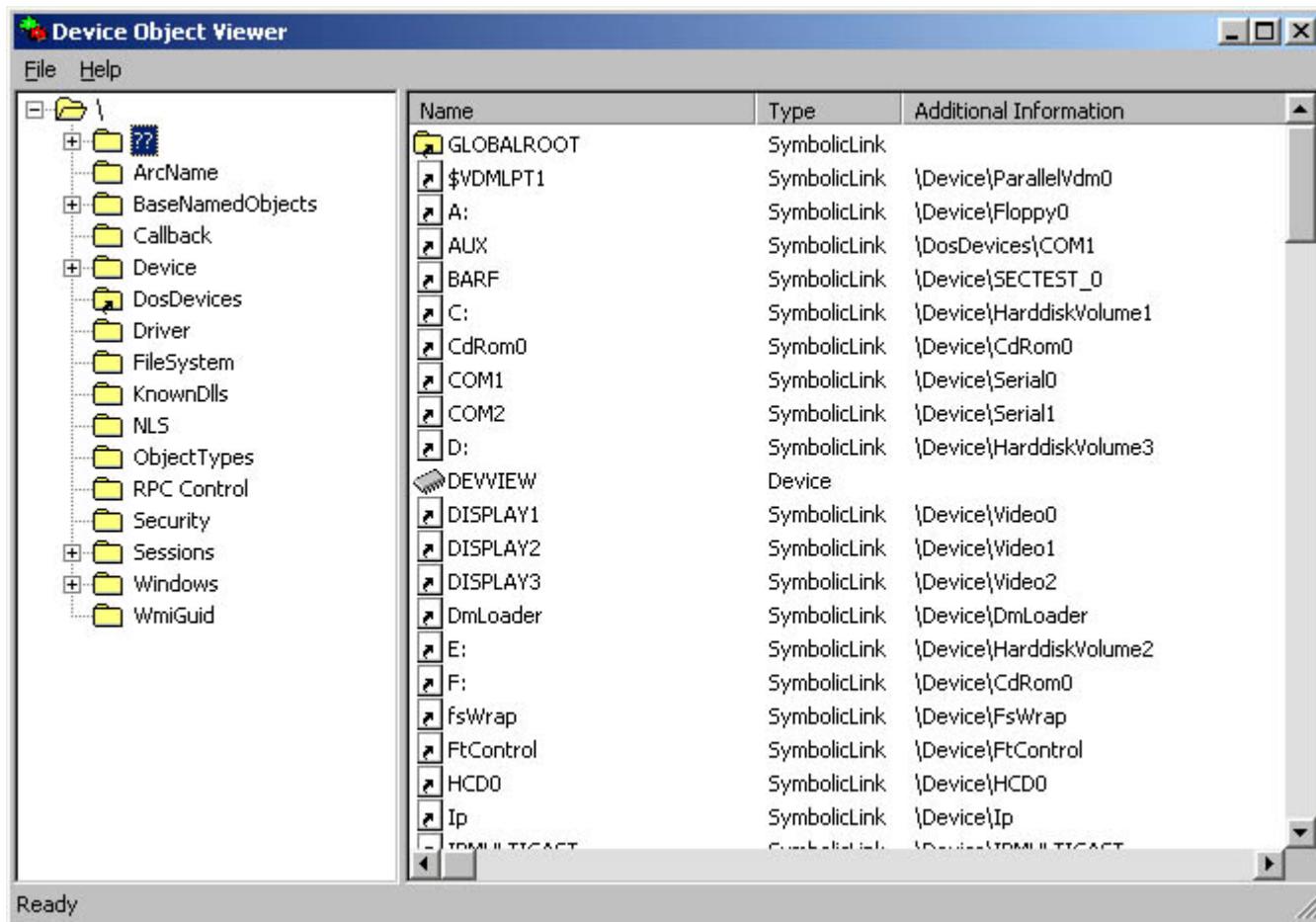


图 2-17. \??目录和部分符号连接

现在，对象管理器要创建一个 IRP，然后把它发到 HarddiskVolume1 设备的驱动程序。该 IRP 最终将使某个文件系统驱动程序或其它驱动程序定位并打开一个磁盘文件。描述文件系统驱动程序的工作过程已经超出了本书的范围。如果我们使用设备名 COM1，那么最终收到该 IRP 的将是\Device\Serial0 的驱动程序。

用户模式程序可以调用 **DefineDosDevice** 创建一个符号连接，如下例：

```
BOOL okay = DefineDosDevice(DDD_RAW_TARGET_PATH, "barf", "\Device\SECTEST_0");
```

图 2-17 中显示了上面调用的结果。

如果你需要在 WDM 驱动程序中创建一个符号连接，可以调用 **IoCreateSymbolicLink** 函数：

```
IoCreateSymbolicLink(linkname, targname);
```

linkname 是要创建的符号连接名，**targname** 是要连接的名字。顺便说一下，对象管理器并不关心 **targname** 是否是已存在对象的名字，如果连接到一个未定义的符号名，那么访问该符号连接将简单地收到一个错误。如果你想允许用户模式程序能超越这个连接而转到其它地方，应使用 **IoCreateUnprotectedSymbolicLink** 函数替代上面函数。

ARC 名字

在 ARC(Advanced RISC Computing)架构的计算机中，Windows 2000 需要依赖一个称为 ARC 命名的概念。你可以在 BOOT.INI 文件中看到使用中的 ARC 名字，BOOT.INI 文件位于引导驱动器的根目录。下面是该文件的一个例子：

```
[boot loader]
timeout=30
default=c:\

[operating systems]
C:="Microsoft Windows 98"
scsi(0)disk(1)rdisk(0)partition(1)\BETA2F="Win2k Beta-2 (Free Build)" /fastdetect /noguiboot
scsi(0)disk(1)rdisk(0)partition(1)\WINNT = "Win2K Beta-3 (Free Build)" /fastdetect /noguiboot
```

在 Intel 平台上，象 scsi(0)disk(1)rdisk(0)partition(1) 的 ARC 名其实是符号连接，它们存在于内核的 \ArcName 目录中。

除了硬盘之外的块存储设备，它们的驱动程序在初始化时需要调用 **IoAssignArcName** 建立一个这样的符号连接。由于系统引导的需要，I/O 管理器自动为硬盘设备建立 ARC 名。

应该命名设备对象吗？

决定为设备对象命名之前，你应该多想一想。如果命名了设备对象，那么任何内核模式程序都可以打开该设备的句柄。另外，任何内核模式或用户模式程序都能创建连接到该设备的符号连接，并可以使用这个符号连接打开设备的句柄。你可能允许也可能不允许这种事情发生。

是否命名设备对象的主要考虑是安全问题。当有人打开一个命名对象的句柄时，对象管理器将检查他是否有权这样做。当 **IoCreateDevice** 为你创建设备对象时，它也为设备对象设置了一个默认安全描述符(基于第四个参数中的设备类型)。下面是三个基本分类，I/O 管理器基于这些分类来选择安全描述符。(参考表 2-4 中的第二列)

- 大部分文件系统设备对象(磁盘、CD-ROM、文件、磁带)将得到“public default unrestricted”ACL(访问控制表)。该表对系统(SYSTEM)和管理员(administrator)之外的所有账户给予了 SYNCHRONIZE、READ_CONTROL、FILE_READ_ATTRIBUTES、FILE_TRAVERSE 访问权限。顺便说一下，文件系统设备对象就是作为 CreateFile 函数的目标而存在，CreateFile 函数将打开一个由文件系统管理的文件。
- 磁盘设备和网络文件系统对象将得到与文件系统对象相同的 ACL，但做了一些修改。例如，任何人对命名软磁盘设备对象都有全部访问权，管理员有足够的权限运行 ScanDisk。(用户模式的网络支持 DLL 需要更大的权限来访问其对应文件系统驱动程序的设备对象，这就是网络文件系统需要与其它文件系统区别对待的原因)
- 所有其它的设备对象将得到“public open unrestricted”ACL，它允许任何有设备句柄的人不受限制地使用该设备。

可以看出，如果非磁盘设备的驱动程序在调用 **IoCreateDevice** 时给出设备对象名，那么任何人都可以读写这个设备，因为默认安全设置几乎允许用户有全部的访问权限，而且在创建符号连接时根本不进行安全检查。安全检查仅发生在对设备的打开操作上，基于命名对象的安全描述符。这对于在同一堆栈中的有更严格安全限制的其它设备对象也是这样。

DevView 可以显示设备对象的安全属性。你可以通过测试一个文件系统、一个磁盘设备、或者任何其它随机存取设备了解到我刚描述过的默认操作规则。

PDO 也得到一个默认安全描述符，但这个安全描述符可能被存储在硬件键或类键的 **Properties** 子键中的安全描述符超越(当两者都存在时，硬件键中的超越值有更高的优先权)。即使没有指定安全描述符超越，如果硬件键或类键的 **Properties** 子键中有设备类型或特征的超越值，那么 I/O 管理器也会基于新类型为对象构造一个新的默认安全描述符。但 I/O 管理器不会超越 PDO 上面的任何其它设备对象的安全设置。因此，由于超越的影响，你不应该命名你的设备对象。但不要失望，应用程序仍可以使用注册的接口(interface)访问你的设备。

关于安全问题的最后一点：当对象管理器析取对象名时，对于名字的中间部分仅需要具有 FILE_TRAVERSE 访问权，它仅在最终对象名上执行全部的安全检查。所以，假设某个设备对象可以通过 \Device\SECTEST_0 名或符号连接 \??\SecurityTest_0 名到达，那么，如果设备对象的安全描述符设置为拒绝写，则试图以写方式打开 \\.\SecurityTest_0 的用户模式应用程序将被阻塞。但如果应用程序试图打开名为 \\.\SecurityTest_0\ExtraStuff 的对象，那么打开请求(IRP_MJ_CREATE 形式)将被允许，而此时用户对 \\.\SecurityTest_0\ 仅有 FILE_TRAVERSE 权限。I/O 管理器希望设备驱动程序自己去处理额外名称部件的安全检查。

为了避免涉及到我刚描述过的安全问题，你可以在调用 **IoCreateDevice** 时指定设备特征参数为 **FILE_DEVICE_SECURE_OPEN**。该标志将使 Windows 2000 在额外名称部件存在的情况下仍检查调用者是否有权限打开设备句柄。

设备名称

如果你决定命名设备对象，通常应该把对象名放在名称空间的\Device 分支中。为了命名设备对象，首先应该创建一个 **UNICODE_STRING** 结构来存放对象名，然后把该串作为调用 **IoCreateDevice** 的参数：

```
UNICODE_STRING devname;
RtlInitUnicodeString(&devname, L"\Device\Simple0");
IoCreateDevice(DriverObject, sizeof(DEVICE_EXTENSION), &devname, ...);
```

我将在下一章中讨论 **RtlInitUnicodeString** 的用法。

通常，驱动程序用设备类型串后加上一个以 0 开始的实例号作为设备对象名(如上面的 Simple0)。一般，你不希望象我上面做的那样使用带有硬编码性质的名称。你希望用串操作函数动态地合成一个名字：

```
UNICODE_STRING devname;
static LONG lastindex = -1;
LONG devindex = InterlockedIncrement(&lastindex);
WCHAR name[32];
_snwprintf(name, arraysize(name), L"\Device\SIMPLE%2.2d", devindex);
RtlInitUnicodeString(&devname, name);
IoCreateDevice(...);
```

我将在后两章中解释上面代码中出现的服务函数。如上面代码所示，从私有设备类型得出的实例号应该是一个静态变量。

设备命名的注意事项

如果仅仅想在开发过程中为应用程序打开设备句柄提供一个快速方法，你应该在\??目录中为设备赋予一个名字。然而，对于一个产品级的驱动程序来说，最好把设备对象名放到\Device 目录中。

\??目录以前叫做\DosDevices。实际上，\DosDevice 仍可以使用，但它本身是\??目录的符号连接。这种改变将使经常查找的用户模式目录名能位于字母排序的目录列表前面。如果你要在命名中使用\??，应该先参考本章“Windows 98 兼容问题”节中的注意事项。

注意，上面提到的把设备对象名放到\??目录中可能不适用于 Windows 2000 的 Terminal Server 版本。由于设备对象不能复制到控制台事务的外边，而符号连接可以，因此你应该在\Device 目录中保存设备命名，而在\DosDevices 目录中放一个符号连接。

在以前版本的 Windows NT 中，某些种类设备(特别是磁盘、磁带、串行口，和并行口)的驱动程序通过调用 **IoGetConfigurationInformation** 来获得一个全局表的指针，该表包含这些类中的设备计数。驱动程序应使用当前计数值来合成设备名称，例如 Harddisk0、Tape1，等等，并同时增加该计数器的值。然而，WDM 驱动程序并不需要使用这个服务函数以及它返回的计数器表，为这些类中的设备构造名称现在是 Microsoft 专有的类驱动程序的责任，如 DISK.SYS。

设备接口

用旧的命名方法命名设备对象，并创建一个应用程序能够使用的符号连接，存在着两个主要问题。命名设备对象所带来的潜在安全问题我们已经讨论过。此外，访问设备的应用程序需要先知道设备采用的命名方案。如果你的硬件仅由你的应用程序访问，那么不会有什么问题。但是，如果有其它公司想为你的硬件写应用程序，并

且有许多硬件公司想制作相似的设备，那么设计一个合适的命名方案是困难的。最后，许多命名方案将依赖于程序员所说的自然语言，这不是一个好的选择。

为了解决这些问题，WDM 引入了一个新的设备命名方案，该方案是语言中立的、易于扩展的、可用于许多硬件和软件厂商，并且易于文档化。该方案依靠一个设备接口(device interface)的概念，它基本上是软件如何访问硬件的一个说明。一个设备接口被一个 128 位的 GUID 唯一标识。你可以用平台 SDK 中的 UUIDGEN 工具或者 GUIDGEN 工具生成 GUID，这两个工具输出同一种数，但格式不同。这个想法就象某些工业组织联合起来共同制定某种硬件的标准访问方法一样。在标准制作过程中，产生了一些 GUID，这些 GUID 将永远关联到某些接口上。

关于 GUID

GUID 用于标识软件接口，它与 COM(部件对象模型)中用于标识 COM 接口的标识符相同，它还用于 OSF(开放软件基金)的 DCE(分布式计算环境)中，标识 RPC(远程过程调用)目标。如果你想了解 GUID 如何生成以及为什么能在统计意义上唯一，请参考 Kraig Brockschmidt 的《Inside OLE, Second Edition (Microsoft Press, 1995)》第 66 页，原始算法规范由 OSF 制定，相关部分见

<http://www.opengroup.org/onlinepubs/9629399/apdxh.htm>。

为了在设备驱动程序中使用 GUID，首先需要使用 UUIDGEN 或者 GUIDGEN 生成 GUID，然后把结果放到头文件中。GUIDGEN 更易于使用，它允许选择 GUID 的输出格式，并把结果送到剪贴板。图 2-18 显示了 GUIDGEN 的运行窗口。你可以把它的输出粘贴到头文件中：

```
// {CAF53C68-A94C-11D2-BB4A-00C04FA330A6}  
DEFINE_GUID(<<name>>,  
0xCAF53C68, 0xA94C, 0x11D2, 0xBB, 0x4A, 0x00, 0xC0, 0x4F, 0xA3, 0x30, 0xA6);
```

然后，用有意义的名字换掉 <<name>>，如 GUID_SIMPLE，并把这个定义包含到驱动程序或应用程序中。

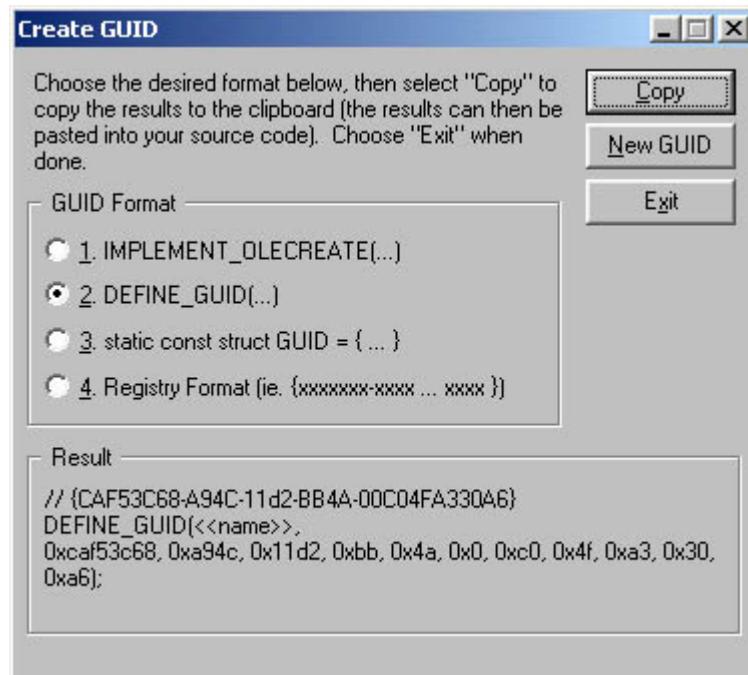


图 2-18. 使用 GUIDGEN 生成 GUID

我想接口类似于蛋白质合成器，它能制作活细胞的细胞膜。访问特定种类设备的应用程序有自己的蛋白质合成器，它就象一把钥匙，可以插入到所有有匹配合成器的设备驱动程序中。如图 2-19。

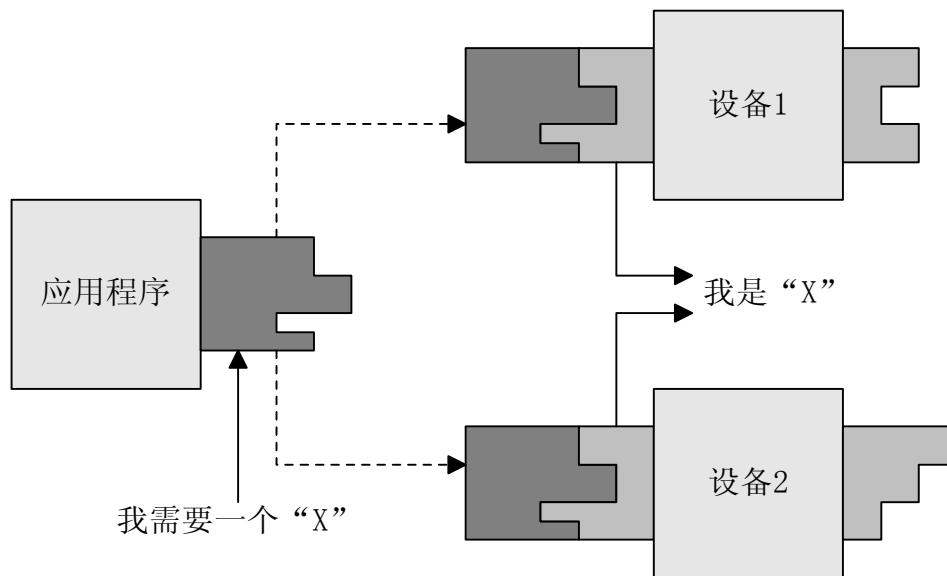


图 2-19. 用设备接口匹配应用程序和设备

注册设备接口 调用 **IoRegisterDeviceInterface** 函数, 功能驱动程序的 AddDevice 函数可以注册一个或多个设备接口:

```
#include <initguid.h>                                     <-1
#include "guids.h"                                         <-2
...
NTSTATUS AddDevice(...)

{
    ...
    IoRegisterDeviceInterface(pdo, &GUID_SIMPLE, NULL, &pdx->ifname);      <-3
    ...
}
```

1. 我们包含了 GUIDS.H 头文件, 那里定义了 **DEFINE_GUID** 宏。**DEFINE_GUID** 通常声明一个外部变量。在驱动程序的某些地方, 我们不得不为将要引用的每个 GUID 保留初始化的存储空间。系统头文件 INITGUID.H 利用某些预编译指令使 **DEFINE_GUID** 宏在已经定义的情况下仍能保留该存储空间。
2. 我使用单独的头文件来保存我要引用的 GUID 定义。这是一个好的想法, 因为用户模式的代码也需要包含这些定义, 但它们不需要那些仅与内核模式驱动程序有关的声明。
3. **IoRegisterDeviceInterface** 的第一个参数必须是设备 PDO 的地址。第二个参数指出与接口关联的 GUID, 第三个参数指出额外的接口细分类名。只有 Microsoft 的代码才使用名称细分类方案。第四个参数是一个 **UNICODE_STRING** 串的地址, 该串用于接收设备对象的符号连接名。

IoRegisterDeviceInterface 的返回值是一个 **Unicode** 串, 这样在不知道驱动程序编码的情况下, 应用程序能用该串确定并打开设备句柄。顺便说一下, 这个名字比较丑陋; 后面例子是我在 Windows 98 中为 Sample 设备生成的名字: \DosDevices\0000000000000007#\{CAF53C68-A94C-11d2-BB4A-00C04FA330A6}。

注册过程实际就是先创建一个符号连接名, 然后再把它存入注册表。之后, 当响应 PnP 请求 **IRP_MN_START_DEVICE** 时, 驱动程序将调用 **IoSetDeviceInterfaceState** 函数“使能”该接口:

```
IoSetDeviceInterfaceState(&pdx->ifname, TRUE);
```

在响应这个调用过程中, I/O 管理器将创建一个指向设备 PDO 的符号连接对象。以后, 驱动程序会执行一个功能相反的调用禁止该接口(用 **FALSE** 做参数调用 **IoSetDeviceInterfaceState**)。最后, I/O 管理器删除符号连接对象, 但它保留了注册表项, 即这个名字将总与设备的这个实例关联; 但符号连接对象与硬件一同到来或消失。

因为接口名最终指向 PDO, 所以 PDO 的安全描述符将最终控制设备的访问权限。这样比较好, 因为只有管理员才可以控制 PDO 的安全属性。

枚举设备接口 内核模式代码和用户模式代码都能定位含有支持它们感兴趣接口的设备。下面我将解释如何在用户模式中枚举所有含有特定接口的设备。枚举代码写起来十分冗长, 最后我不得不写一个 C++类来实现。你可

以在 DEVICELIST.CPP 和 DEVICELIST.H 文件中找到这些代码，这些文件是第八章“电源管理”中 WDMIDLE 例子的一部分。它们声明并实现了一个 **CDeviceList** 类，该类包含一个 **CDeviceListEntry** 对象数组。这两个类声明如下：

```
class CDeviceListEntry
{
public:
    CDeviceListEntry(LPCTSTR linkname, LPCTSTR friendlyname);
    CDeviceListEntry(){}
    CString m_linkname;
    CString m_friendlyname;
};

class CDeviceList
{
public:
    CDeviceList(const GUID& guid);
    ~CDeviceList();
    GUID m_guid;
    CArray<CDeviceListEntry, CDeviceListEntry&> m_list;
    int Initialize();
};
```

该类使用了 **CString** 类和 **CArray** 模板，它们都是 MFC 的一部分。这两个类的构造函数仅简单地把它们的参数复制到数据成员中：

```
CDeviceList::CDeviceList(const GUID& guid)
{
    m_guid = guid;
}

CDeviceListEntry::CDeviceListEntry(LPCTSTR linkname, LPCTSTR friendlyname)
{
    m_linkname = linkname;
    m_friendlyname = friendlyname;
}
```

所有实际的工作都发生在 **CDeviceList::Initialize** 函数中。其执行过程大致是这样：先枚举所有接口 **GUID** 与构造函数得到的 **GUID** 相同的设备，然后确定一个“友好”名，我们希望向最终用户显示这个名字。最后返回找到的设备号。下面是这个函数的代码：

```
int CDeviceList::Initialize()
{
    HDEVINFO info = SetupDiGetClassDevs(&m_guid, NULL, NULL, DIGCF_PRESENT | DIGCF_INTERFACEDEVICE);    <-1
    if (info == INVALID_HANDLE_VALUE)
        return 0;
    SP_INTERFACE_DEVICE_DATA ifdata;
    ifdata.cbSize = sizeof(ifdata);
    DWORD devindex;
    for (devindex = 0; SetupDiEnumDeviceInterfaces(info, NULL, &m_guid, devindex, &ifdata); ++devindex)
    {
        DWORD needed;
        SetupDiGetDeviceInterfaceDetail(info, &ifdata, NULL, 0, &needed, NULL);
        PSP_INTERFACE_DEVICE_DETAIL_DATA detail = (PSP_INTERFACE_DEVICE_DETAIL_DATA)
```

```

malloc(needed);

    detail->cbSize = sizeof(SP_INTERFACE_DEVICE_DETAIL_DATA);
    SP_DEVINFO_DATA did = { sizeof(SP_DEVINFO_DATA) };
    SetupDiGetDeviceInterfaceDetail(info, &ifdata, detail, needed, NULL, &did));

    TCHAR fname[256];
    if (!SetupDiGetDeviceRegistryProperty(info,
                                          &did,
                                          SPDRP_FRIENDLYNAME,
                                          NULL,
                                          (PBYTE) fname,
                                          sizeof(fname),
                                          NULL)
        && !SetupDiGetDeviceRegistryProperty(info,
                                              &did,
                                              SPDRP_DEVICEDESC,
                                              NULL,
                                              (PBYTE) fname,
                                              sizeof(fname),
                                              NULL))
    )
    _tcscpy(fname, detail->DevicePath, 256);

    CDeviceListEntry e(detail->DevicePath, fname);
    free((PVOID) detail);

    m_list.Add(e);
}

SetupDiDestroyDeviceInfoList(info);
return m_list.GetSize();
}

```

1. 该语句打开一个枚举句柄，我们用它寻找寄存了指定 **GUID** 接口的所有设备。
2. 循环调用 **SetupDiEnumDeviceInterfaces** 以寻找每个匹配的设备。
3. 有两项信息是我们需要的，接口的“细节”信息和设备实例信息。这个“细节”信息就是设备的符号名。因为它的长度可变，所以我们两次调用了 **SetupDiGetDeviceInterfaceDetail**。第一次调用确定了长度，第二次调用获得了名字。
4. 通过询问注册表中的 **FriendlyName** 键或 **DeviceDesc** 键，我们获得了设备的“友好”名称。
5. 我们用设备符号名同时作为连接名和友好名创建了类 **CDeviceListEntry** 的一个临时实例 **e**。

友好名

你可能会疑惑，注册表怎么会有设备的 **FriendlyName** 名。安装设备驱动程序的 **INF** 文件中有一个指定设备参数的 **HW** 段，这些参数将被添加到注册表中。通常我们可以在这里为设备提供一个 **FriendlyName** 名。

其它全局性的设备初始化操作

在 **AddDevice** 中还需要加入其它一些步骤来初始化设备对象，下面我将按顺序描述这些步骤。

初始化设备扩展

设备扩展的内容和管理全部由用户决定。该结构中的数据成员应直接反映硬件的专有细节以及对设备的编程方式。大多数驱动程序都会在这里放入一些数据项，下面代码声明了一个设备扩展结构：

```

typedef struct _DEVICE_EXTENSION {
    PDEVICE_OBJECT DeviceObject;           <--1
    PDEVICE_OBJECT LowerDeviceObject;       <--2
    PDEVICE_OBJECT Pdo;                   <--3
    UNICODE_STRING ifname;                <--4
    IO_REMOVE_LOCK RemoveLock;            <--5
    DEVSTATE devstate;                  <--6
    DEVSTATE prevstate;
    DEVICE_POWER_STATE devpower;          <--7
    SYSTEM_POWER_STATE syspower;
    DEVICE_CAPABILITIES devcaps;          <--8
    ...
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

```

1. 我模仿 DDK 中官方的结构声明模式声明了这个结构。
2. 我们可以用设备对象中的 **DeviceExtension** 指针定位自己的设备扩展。同样，我们有时也需要在给定设备扩展时能定位设备对象。因为某些函数的逻辑参数就是设备扩展本身(这里有设备每个实例的全部信息)。所以，我认为这里应该有一个 **DeviceObject** 指针。
3. 我在一些地方曾提到过，在调用 **IoAttachDeviceToDeviceStack** 函数时，应该把紧接着你下面的设备对象的地址保存起来。**LowerDeviceObject** 成员用于保存这个地址。
4. 有一些服务例程需要 PDO 的地址，而不是堆栈中某个高层设备对象的地址。由于定位 PDO 非常困难，所以最好的办法是在 **AddDevice** 执行时在设备扩展中保存一个 PDO 地址。
5. 无论你用什么方法(符号连接或设备接口)命名你的设备，都希望能容易地获得这个名字。所以，这里我用一个 Unicode 串成员 **ifname** 来保存设备接口名。如果你使用一个符号连接名而不是设备接口，应该使用一个有相关含义的成员名，例如“linkname”。
6. 当你调用 **IoDeleteDevice** 删除这个设备对象时，需要使用一个自旋锁来解决同步安全问题，我将在第六章中讨论同步问题。因此，需要在设备扩展中分配一个 **IO_REMOVE_LOCK** 对象。**AddDevice** 有责任初始化这个对象。
7. 你可能需要一个成员来记录设备当前的 PnP 状态和电源状态。**DEVSTATE** 和 **POWERSTATE** 是枚举类型变量，我假设事先已经在头文件中声明了这些变量类型。我将在后面章节中讨论这些状态变量的用途。
8. 电源管理的另一个部分涉及电源能力设置的恢复，设备扩展中的 **devcaps** 结构用于保存这些设置。

下面是 **AddDevice** 中的初始化语句(着重设备扩展部分的初始化):

```

NTSTATUS AddDevice(...)
{
    PDEVICE_OBJECT fdo;
    IoCreateDevice(..., sizeof(DEVICE_EXTENSION), ..., &fdo);
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    pdx->DeviceObject = fdo;
    pdx->Pdo = pdo;
    IoInitializeRemoveLock(&pdx->RemoveLock, ...);
    pdx->devstate = STOPPED;
    pdx->devpower = PowerDeviceD0;
    pdx->syspower = PowerSystemWorking;
    IoRegisterDeviceInterface(..., &pdx->ifname);
    pdx->LowerDeviceObject = IoAttachDeviceToDeviceStack(...);
}

```

初始化默认的DPC对象

许多设备使用中断来报告操作完成。我将在第七章“读写数据”中讨论中断处理，其中对中断服务例程(**ISR**)能做什么做了严格的限定。特别是 **ISR** 不能调用用于报告 IRP 完成的例程(**IoCompleteRequest**)。利用 DPC(推迟过程调用)可以绕过这个限制。你的设备对象中应包含一个辅助 DPC 对象，它可以调度你的 DPC 例程，该对象应该在设备对象创建后不久被初始化。

```
NTSTATUS AddDevice(...)  
{  
    IoCreateDevice(...);  
    IoInitializeDpcRequest(fdo, DpcForIsr);  
}
```

设置缓冲区对齐掩码

执行 DMA 传输的设备直接使用内存中的数据缓冲区工作。HAL 要求 DMA 传输中使用的缓冲区必须按某个特定界限对齐，而且设备也可能有更严格的对齐需求。设备对象中的 **AlignmentRequirement** 域表达了这个约束，它是一个位掩码，等于要求的地址边界减一。下面语句可以把任何地址圈入这个界限：

```
PVOID address = ...;  
SIZE_T ar = fdo->AlignmentRequirement;  
address = (PVOID) ((SIZE_T) address & ~ar);
```

还可以把任意地址圈入下一个对齐边界：

```
PVOID address = ...;  
SIZE_T ar = fdo->AlignmentRequirement;  
address = (PVOID) (((SIZE_T) address + ar) & ~ar);
```

在这两段代码中，我使用了 **SIZE_T** 把指针类型(它可以是 32 位也可以是 64 位，这取决于编译的目标平台)转化成一个整型，该整型与原指针有同样的跨度范围。

`IoCreateDevice` 把新设备对象中的 **AlignmentRequirement** 域设置成 HAL 要求的值。例如，Intel 的 x86 芯片没有对齐需求，所以 **AlignmentRequirement** 的默认值为 0。如果设备需要更严格的缓冲区对齐(例如设备有总线主控的 DMA 能力，要求对齐数据缓冲区)，应该修改这个默认值，如下：

```
if (MYDEVICE_ALIGNMENT - 1 > fdo->AlignmentRequirement)  
    fdo->AlignmentRequirement = MYDEVICE_ALIGNMENT - 1;
```

我假设你在驱动程序某处已定义了一个名为 **MYDEVICE_ALIGNMENT** 的常量，它是 2 的幂，代表设备的数据缓冲区对齐需求。

其它对象

设备可能还有其它一些需要在 `AddDevice` 中初始化的对象。这些对象可能包括各种同步对象，各种队列头(queue anchors)，聚集/分散列表缓冲区，等等。事实上，在本书的其它地方讨论这些对象的初始化更合适。

初始化设备标志

设备对象中有两个标志位需要在 `AddDevice` 中初始化，并且它们在以后也不会改变，它们是 **DO_BUFFERED_IO** 和 **DO_DIRECT_IO** 标志。你只能设置并使用其中一个标志，它将决定你以何种方式处理来自用户模式的内存缓冲区。(我将在第七章中讨论这两种缓冲模式的不同，以及你如何选择) 由于任何在后面装入的上层过滤器驱动程序将复制你的标志设置，所以在 `AddDevice` 中做这个选择十分重要。如果你在过滤器驱动程序装入后改变了设置，它们可能会不知道。

设备对象中有两个标志位属于电源管理范畴。与前两个缓冲区标志不同，这两个标志在任何时间都可以被改变。我将在第八章中详细讨论它们，但这里我先介绍一下。**DO_POWER_PAGABLE** 意味着电源管理器将在 **PASSIVE_LEVEL** 级上向你发送 **IRP_MJ_POWER** 请求。**DO_POWER_INRUSH** 意味着你的设备在上电时将汲取大量电流，因此，电源管理器将确保没有其它 **INRUSH** 设备同时上电。

设置初始电源状态

大部分设备一开始就进入全供电状态。如果你知道你的设备的初始电源状态，应该告诉电源管理器：

```
POWER_STATE state;  
state.DeviceState = PowerDeviceD0;  
PoSetPowerState(fdo, DevicePowerState, state);
```

电源管理的细节请见第八章。

建立设备堆

每个过滤器驱动程序和功能驱动程序都有责任把设备对象放到设备堆栈上，从 PDO 开始一直向上。你可以调用 **IoAttachDeviceToDeviceStack** 完成你那部分工作：

```
NTSTATUS AddDevice(..., PDEVICE_OBJECT pdo)  
{  
    PDEVICE_OBJECT fdo;  
    IoCreateDevice(..., &fdo);  
    pdx->LowerDeviceObject = IoAttachDeviceToDeviceStack(fdo, pdo);  
}
```

IoAttachDeviceToDeviceStack 的第一个参数是新创建的设备对象的地址。第二个参数是 PDO 地址。**AddDevice** 的第二个参数也是这个地址。返回值是紧接着你下面的任何设备对象的地址，它可以是 PDO，也可以是其它低级过滤器设备对象。如果该函数失败则返回一个 NULL 指针，因此你的 **AddDevice** 函数也是失败的，应返回 **STATUS_DEVICE_REMOVED**。

清除DO_DEVICE_INITIALIZING标志

在 **AddDevice** 中最后一件需要做的事是清除设备对象中的 **DO_DEVICE_INITIALIZING** 标志：

```
fdo->Flags &= ~DO_DEVICE_INITIALIZING;
```

当这个标志设置时，I/O 管理器将拒绝任何打开该设备句柄的请求或向该设备对象上附着其它设备对象的请求。在驱动程序完成初始化后，必须清除这个标志。在以前版本的 Windows NT 中，大部分驱动程序在 **DriverEntry** 中创建所有需要的设备对象。当 **DriverEntry** 返回时，I/O 管理器自动遍历设备对象列表并清除该标志。但在 WDM 驱动程序中，设备对象在 **DriverEntry** 返回后才创建，所以 I/O 管理器不会自动清除这个标志，驱动程序必须自己清除它。

Windows 98 兼容问题

Windows 98 在创建某些外围设备对象和装载驱动程序上与 Windows 2000 有一些细节上的不同。本节将解释能影响到驱动程序开发的某些差异。

DriverEntry 调用上的不同

如我以前提到的, DriverEntry 例程有一个 UNICODE_STRING 参数, 该参数指定驱动程序的服务键。在 Windows 2000 中, 该串以全注册表路径形式出现“\Registry\Machine\System\CurrentControlSet\Services\xxx”(xxx 是服务表项的名称)。在 Windows 98 中, 该串出现的形式是“System\CurrentControlSet\Services\<classname>\<instance#>”(<classname>是设备所属类的名称, <instance-#>是实例号, 例如 0000)。用 **ZwOpenKey** 函数打开该键可以避免这个环境差异。

注册表组织的不同

Windows 98 与 Windows 2000 在设备注册表项的组织上有些差异。下面这些简短解释说明了这个问题, 如果你阅读了第十二章资料后再回头看这些解释会更加明了。

- Windows 98 的硬件键位于 HKLM\Enum 下并且没有任何保护(因为 Windows 98 没有安全系统)。它没有 **Service** 值; 但有一个 **Driver** 值, 提供服务键名的另外两个部分。由于 Windows 98 注册表没有 MULTI_SZ 类型, 所以 **LowerFilters** 和 **UpperFilters** 值将以二进制形式出现, 并且该值使用 8 位字符来命名驱动程序映像文件(.SYS 扩展名)。
- Windows 98 的类键在 HKLM\System\CurrentControlSet\Services\Class 下。
- Windows 98 的服务键是类键的一个子键。服务键的表项中包含一个 **DevLoader** 值和一个 **NTMPDriver** 值, 前者指向 NTKERN.VXD, 后者命名驱动程序映像(.SYS 扩展名), 这个驱动程序必须存在于%SystemRoot%\System32\Drivers 目录中。

\??目录

Windows 98 不能识别\??目录名。所以, 你需要把符号连接名放到\DosDevices 目录中。该目录在 Windows NT 中也可以使用, 因为它就是\??目录的符号连接。

未实现的设备类型

原始版本的 Windows 98 不支持创建块存储设备对象, 这些设备类型包括 FILE_DEVICE_DISK、FILE_DEVICE_TAPE、FILE_DEVICE_CD_ROM, 和 FILE_DEVICE_VIRTUAL_DISK。即使你调用了 IoCreateDevice 函数, 并且它也返回了 STATUS_SUCCESS 状态码, 但并没有实际的设备对象出现, 而且在最后一个参数给出的 PDEVICE_OBJECT 地址也没有被填写。

问题的根源是 Windows 98 的磁盘驱动程序必须使用为 Windows 95 制定的 I/O 管理架构。但是, 为什么 IoCreateDevice 以沉默方式返回失败则有些另人迷惑。

第三章：基本编程技术

写一个 WDM 驱动程序就象在做一次软件工程方面的练习。无论硬件有什么需求，你都必须把各种编程元素组合起来以形成一个完整的程序。在上一章中，我描述了 WDM 驱动程序的基本结构，并详细阐述了两个基本例程：`DriverEntry` 和 `AddDevice`。在这一章中，我将集中到更基本的主题上，如错误处理、内存和数据结构管理、注册表和文件存取，以及其它一些主题。另外，我还用一些辅助驱动程序调试的简短讨论来充实这章内容。

- 内核模式编程环境
- 错误处理
- 内存管理
- 字符串操作
- 其它编程技术
- Windows 98 兼容问题

内核模式编程环境

图 3-1 显示了 Windows NT 操作系统的某些组成部分。每个部分都输出一些服务函数，这些函数以两个特别的字母组合开头：

- I/O 管理器(**Io** 前缀) 包含许多驱动程序可以使用的服务函数，对这些函数的描述遍及本书。
- 进程结构模块(**Ps** 前缀) 创建并管理内核模式线程。普通的 WDM 驱动程序应使用一个独立的线程来循环生成中断的设备。
- 内存管理器(**Mm** 前缀) 控制页表，页表定义了虚拟内存到物理内存之间的映射。
- executive (**Ex** 前缀) 提供堆管理和同步服务。本章将讨论堆管理函数。下一章讨论同步服务。
- 对象管理器(**Ob** 前缀) 集中控制 Windows NT 中的各种数据对象。WDM 驱动程序仅需要对象管理器维护对象的参考计数，以防止对象被意外删除。
- 安全参考监视器(**Se** 前缀) 使文件系统驱动程序执行安全检测。I/O 请求到达 WDM 驱动程序前已经做完了安全检测，所以本书不讨论这些函数。
- 运行时间库部件(**Rtl** 前缀) 包含工具例程，例如列表和串管理例程，内核模式驱动程序可以用这些例程来替代常规的 ANSI 标准例程。大部分例程可以从其名字上直接看出它的功能。
- Win32 子系统存在于用户模式中，所以用户模式中的应用程序可以容易地调用其例程。为了方便，Windows NT 在内核模式中实现了一些有 **Zw** 前缀名的函数，这些函数可以使驱动程序调用 Win32 子系统例程。Windows 2000 DDK 中仅暴露一小部分这样的函数给驱动程序使用，包括访问文件和注册表的函数。我将在本章讨论这些函数。
- Windows NT 内核(**Ke** 前缀) 所有多线程和多处理器的低级同步活动都发生在内核中，我将在下一章中讨论 **KeXxx** 函数。
- 在操作系统的最底层是硬件抽象层(**HAL**, **Hal** 前缀)。操作系统把所有关于计算机硬件如何连接的信息都存放在 **HAL** 中。**HAL** 了解如何在特定平台上实现中断操作，如何实现自旋锁，如何寻址 I/O 或内存映射设备，等等。WDM 驱动程序不直接与硬件对话，它通过调用 **HAL** 中的函数来达到目的。所以 WDM 驱动程序能够实现平台无关和总线无关。

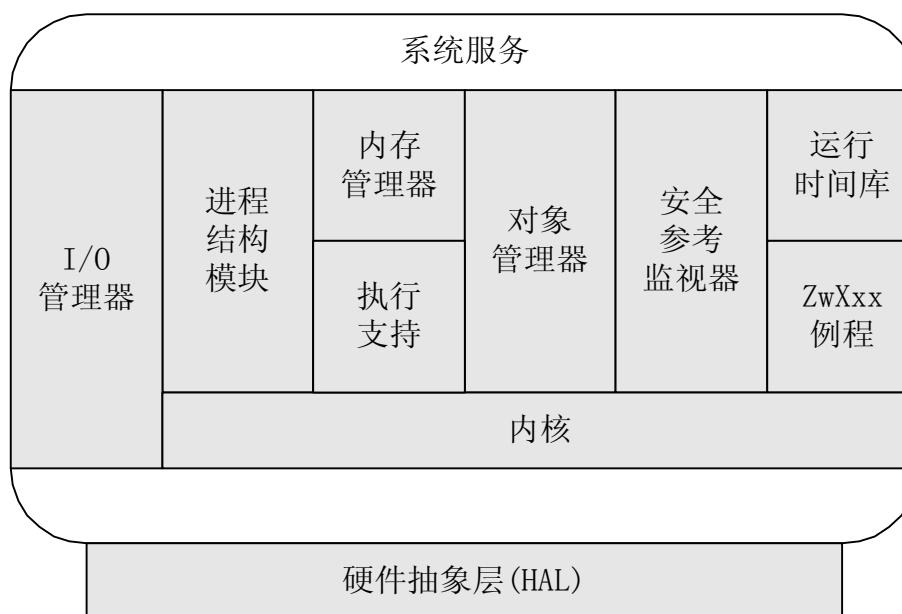


图 3-1. 内核模式支持例程概观

使用标准运行时间库函数

在历史上，Windows NT 的设计者认为，不应该在驱动程序中使用 C 编译器厂商提供的运行时间库。部分原因是由于 Windows NT 是在 ANSI 标准时设计的，每个 C 编译器厂商都有自己的实现方法和品质标准。另一个原因是因为标准运行时间库中的例程有时需要依赖用户模式中的应用程序来初始化，并且有些例程并不是以多线程或多处理器安全的方式实现的。

直到现在，官方认为内核模式驱动程序仅应调用 DDK 中公开的函数。例如，你不能在驱动程序中调用 **wcscmp** 函数，而应该调用 **RtlCompareUnicodeString**。然而，这里有一个公开的秘密，用于创建驱动程序的标准输入库(ntoskrnl.lib)定义了许多函数，而这些函数却是在诸如 **string.h**、**stdio.h**、**stdlib.h**，和 **ctypes.h** 的头文件中

声明的，这些头文件都是应用程序经常使用的头文件。所以，为什么我们不能使用它们？实际上，倘若你了解所有的内部细节，你完全可以调用它们。但你不能总这样做，例如，你不能总用 **memcpy** 替代 **RtlCopyBytes**，因为这两者稍有不同。**(RtlCopyByte)** 可以保证一个字节一个字节地复制数据而不是以较大的块，大块复制数据在某些 **RISC** 平台上会出现麻烦）

注意侧效

驱动程序中使用的许多支持“函数”其实是 DDK 头文件中定义的宏。我们都应该避免在宏的参数中使用带有边效的表达式，原因很明显，宏可以多次使用其参数，见下面代码：

```
int a = 2, b = 42, c;  
c = min(a++, b);
```

a 的值是什么？(**c** 的值又是什么？) 让我们看看这个似是而非的 **min** 宏：

```
#define min(x,y) (((x)<(y)) ? (x) : (y))
```

如果你用 **a++** 代替 **x**，你将看到 **a** 最后等于 4，因为表达式 **a++** 执行了两次。而“函数”**min** 将返回 3 而不是 2，因为函数的返回值是在第二次计算 **a++** 之前提取的 **a** 值。

通常，你不能知道 DDK 什么时候使用宏，什么时候使用真正的外部函数。有时候，一个特殊的服务函数在某些平台上是宏而在其它平台上却是外部函数。此外，Microsoft 也可能在将来改变想法。所以，当你写 WDM 驱动程序时应坚守下面原则：

决不在内核模式服务函数的参数中使用带有侧效的表达式。

错误处理

人总会犯错误，错误恢复是软件工程的一部分。程序中总会发生异常情况，其中一些源自程序中的 Bug，或者在我们的代码中或者在调用我们代码的用户模式应用程序中。另一些涉及到系统装载或硬件的瞬间状态。无论什么原因，代码必须能对不寻常的情况作出恰当的反应。在这一节中，我将描述三种错误处理形式：状态代码、结构化异常处理，和 **bug check**。一般，内核模式支持例程通过返回状态代码来报告意外错误。对于正常情况，它们将返回布尔值或者数值而不是正式的状态代码。结构化异常处理为异常事件发生后的清除工作提供了一个标准化方法，它可以避免因为异常事件而导致系统崩溃，异常事件是指诸如被零除或参考无效指针等的意外错误。**Bug check** 实际上就是致命错误的内部名称，对于这种错误，唯一的解决办法就是重启系统。

状态代码

内核模式支持例程(以及你的代码)通过向其调用者返回一个状态代码来表明调用是否成功。**NTSTATUS** 是一个由多个子域组成的 32 位整数，如图 3-2。高两位(**Severity**)指出状态的严重性——成功、信息、警告、错误。客户位(**Customer**)是一个标志，完成的 IRP 将携带一个表明完成状态的状态代码，如果这个状态代码中的 **Customer** 标志被设置，那么这个状态代码将被不修改地传回应用程序(应用程序通过调用 **GetLastError** 函数获得)。通常，状态代码在返给应用程序前要翻译成 Win32 错误代码(Win32 错误代码可以在 **KBase Q113996** 文章中查到)。**facility** 代码指出该状态是由哪个系统部件导致的，一般用于减少开发组之间的代码关联。剩下的 16 位代码指出实际的状态。

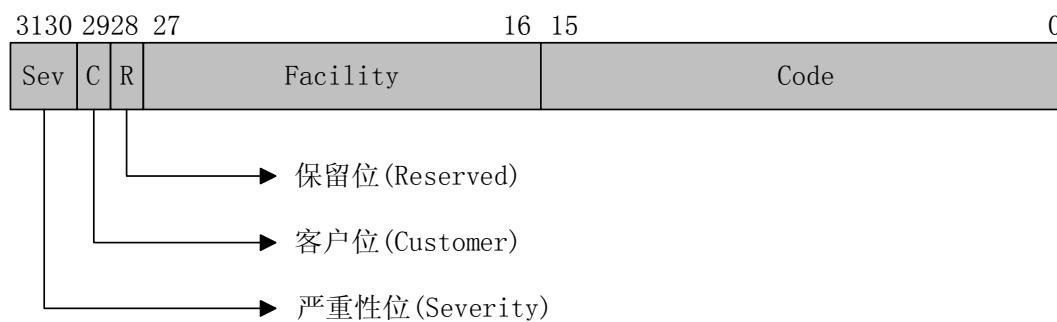


图 3-2. **NTSTATUS** 代码的格式

我们应该总是检测例程的返回状态。为了不让大量的错误处理代码干扰例子代码所表达的实际意图，我经常省略代码片段中错误检测部分，但你在实际练习中不要效仿我。

如果状态码高位为 0，那么不管其它位是否设置，该状态代码仍旧代表成功。所以，绝对不要用状态代码与 0 比较来判断操作是否成功，应该使用 **NT_SUCCESS** 宏：

```
NTSTATUS status = SomeFunction(...);
if(!NT_SUCCESS(status))
{
    <handle error>
}
```

不仅要检测调用例程的返回状态，还要向调用你的例程返回状态代码。在上一章中，我讲述了两个驱动程序例程，**DriverEntry** 和 **AddDevice**，它们都定义了 **NTSTATUS** 返回代码。所以，如果这些例程成功，则应返回 **STATUS_SUCCESS**。如果在某个地方出错，应返回一个适当的错误状态代码，有时函数返回的状态代码就是出错函数返给你的状态代码。

例如，下面是 **AddDevice** 函数的一些初始化代码，包括完整的错误检测代码：

```
NTSTATUS AddDevice(PDRIVER_OBJECT DriverObject, PDEVICE_OBJECT pdo)
{
```

```

NTSTATUS status;
PDEVICE_OBJECT fdo;
status = IoCreateDevice(DriverObject,
                       sizeof(DEVICE_EXTENSION),
                       NULL,
                       FILE_DEVICE_UNKNOWN,
                       0,
                       FALSE,
                       &fdo);
if (!NT_SUCCESS(status))
{
    KdPrint(("IoCreateDevice failed - %X\n", status));
    return status;
}
PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
pdx->DeviceObject = fdo;
pdx->Pdo = pdo;
pdx->state = STOPPED;
IoInitializeRemoveLock(&pdx->RemoveLock, 0, 0, 255);
status = IoRegisterDeviceInterface(pdo, &GUID_SIMPLE, NULL, &pdx->ifname);
if (!NT_SUCCESS(status))
{
    KdPrint(("IoRegisterDeviceInterface failed - %X\n", status));
    IoDeleteDevice(fdo);
    return status;
}
...
}

```

1. 如果 **IoCreateDevice** 失败，我们就把这个状态代码返回给上层调用者。注意代码中 **NT_SUCCESS** 宏的使用。
2. 打印出任何错误状态信息是一个好的习惯，尤其是在调试驱动程序时。我将在本章后面讨论如何使用 **KdPrint**。
3. **IoInitializeRemoveLock** 是一个 VOID 函数，这意味着它不会失败。所以没有必要检测它的状态代码，该函数在第六章讨论。
4. 如果 **IoRegisterDeviceInterface** 失败，我们就需要在返回前做些清除工作；即我们必须删除刚创建的设备对象。

并不是所有被调用例程导致的错误都要处理，有些错误是可以忽略的。例如，在第八章电源管理中，我将提到带有 **IRP_MN_POWER_SEQUENCE** 子类型的电源管理请求，使用它可以避免上电过程中不必要的状态恢复过程。这个请求不仅对你是可选的，而且总线驱动程序在实现该请求上也是可选的。所以如果该请求执行失败，你不用做任何处理，继续其它工作。同样，你也可以忽略 **IoAllocateErrorLogEntry** 产生的错误，因为不能向错误登记表添加一条记录根本不是什么严重错误。

结构化异常处理

Windows NT 提供了一种处理异常情况的方法，它可以帮助我们避免潜在的系统崩溃。结构化异常处理与编译器的代码生成器紧密集成，它允许你在自己的代码段周围加上保护语句，如果被保护代码段中的任何语句出现异常，系统将自动调用异常处理程序。结构化异常处理还便于你提供清除语句，不管控制以何种方式离开被保护代码段，清除代码都会被执行。

许多读者并不熟悉结构化异常方法，所以我在这里先解释一些基本概念。使用这个方法可以写出更好更稳固的代码。在许多情况下，**WDM** 驱动程序例程接收到的参数都是经过其它代码严格检验的，一般不会成为导致异常的原因。但我们仍要遵循这样基本原则：对用户模式虚拟内存直接引用的代码段应该用结构化异常帧保护起来。这样的引用通常发生在调用 **MmProbeAndLockPages**、**ProbeForRead**，和 **ProbeForWrite** 函数时。

注意

结构化异常机制可以使内核模式代码在访问一个非法的用户模式地址后避免系统崩溃。但它不能捕捉其它处理器异常，例如被零除或试图访问非法的内核模式地址。从这一点上看，这种机制在内核模式中不象在用户模式中那样具有通用性。

内核模式程序通过在内存堆栈上建立异常帧来实现结构化异常，这个堆栈就是程序用于参数传递、子程序调用，和分配自动变量所使用的内存堆栈，我不将详细描述这个机制的内部过程，因为该机制在不同的 Windows NT 平台上会不同。然而，这个机制与在用户模式程序中使用的结构化异常机制相同，你可以在这两个地方找到对这个机制的详细描述，Matt Pietrek 的文章“*A Crash Course on the Depths of Win32 Structured Exception Handling*”*Microsoft Systems Journal* (January 1997)。Jeff Richter 在《*Programming Applications for Microsoft Windows, Fourth Edition* (Microsoft Press, 1999)》中的专题讨论。

当异常发生时，操作系统通过扫描堆栈异常帧来寻找相应的异常处理程序。图 3-3 描绘了这个逻辑流程。每个异常帧都指定一个过滤函数，系统调用这个过滤函数来回答这样的问题：“你能处理这个异常吗？”，当系统找到对应的异常处理程序时，它就回卷堆栈异常帧以恢复处理程序所需要的上下文。回卷过程包括调用同一组过滤函数，并指定一个有这样含义的参数：“我们正在回卷；如果你回答：是，那么立即接管控制”，如果没有代码处理这个异常，那么这里将是一个默认处理例程，该例程使系统崩溃。

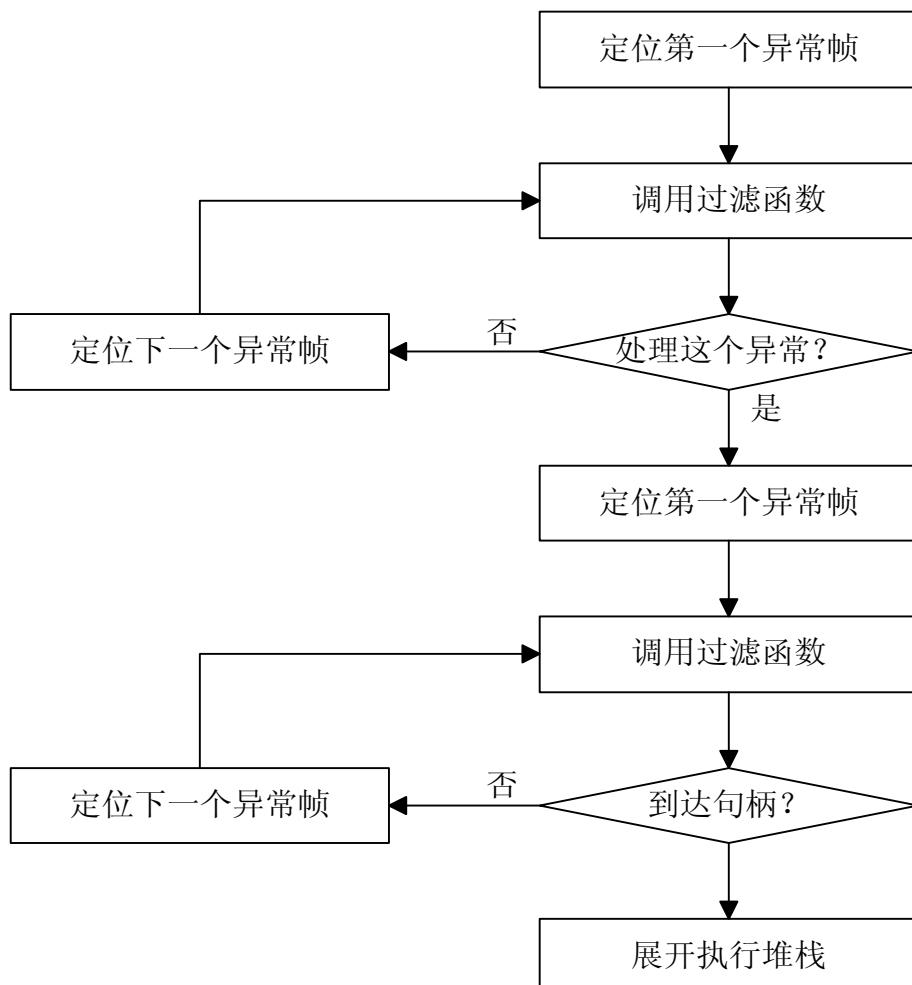


图 3-3. 结构化异常处理逻辑

当使用 Microsoft 编译器时，你可以使用 C/C++ 的 Microsoft 扩展，它隐藏了使用某些操作系统原语的复杂性。例如，用 `_try` 语句指定被保护代码段，用 `_finally` 语句指定终止处理程序，用 `_except` 语句指定异常处理程序。

注意

最好总使用带有双下划线的关键字，如 `_try`、`_finally`、和 `_except`。在 C 编译单元中，DDK 头文件 `WARNING.H` 也把 `try`、`finally`、和 `except` 宏定义成这些双下划线的关键字。DDK 例子程序使用这些宏而不是直接使用带双下划线的关键字。有一点需要注意：在 C++ 编译单元中，`try` 语句必须与 `catch` 语句成对出现，这是一个完全不同的异常机制，是 C++ 语言的一部分。C++ 异常机制不能用于驱动程序中，除非你自己从运行时间库中复制出某些基础结构。Microsoft 不推荐那样做，因为这将增加驱动程序的内存消耗并增大执行文件的大小。

Try-Finally块

从 *try-finally* 块开始解释结构化异常处理最为容易，用它你可以写出象下面这样的清除代码：

```
__try
{
    <guarded body>
}
__finally
{
    <termination handler>
}
```

在这段伪代码中，被保护体*<guarded body>*是一系列语句和子例程。通常，这些语句会有副作用，如果没有副作用，就没有必要使用一个 *try-finally* 块，因为没有东西需要清除。终止处理程序*<termination handler>*包含一些恢复语句，用于部分或全部恢复被保护体产生的副作用。

语法上，*try-finally* 按下面方式工作。首先，计算机执行被保护体*<guarded body>*。由于某种原因控制离开被保护体，计算机执行终止处理程序。如图 3-4。

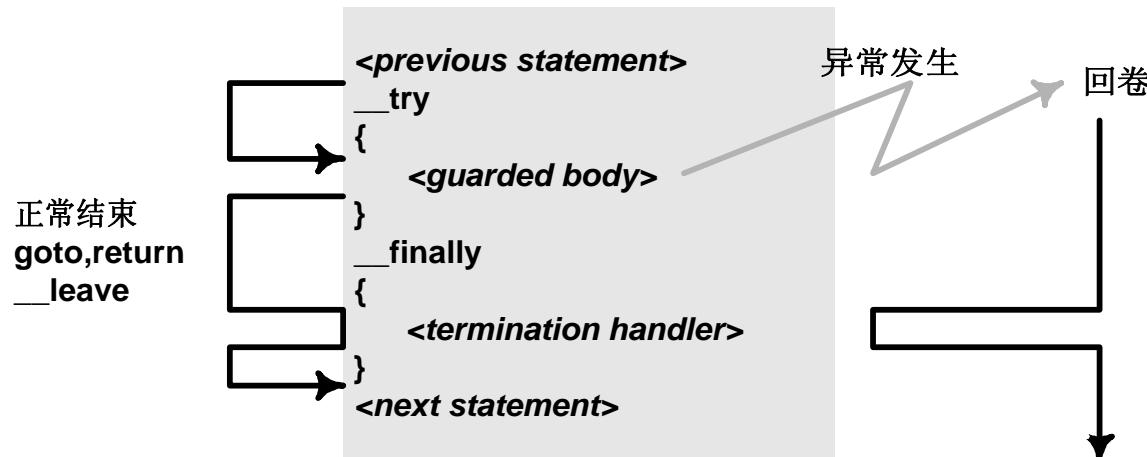


图 3-4. *try-finally* 中的控制流程

这里有一个简单的例子：

```
LONG counter = 0;
__try
{
    ++counter;
}
__finally
{
    --counter;
}
KdPrint((""%d\n", counter));
```

首先，被保护体执行并把 **counter** 变量的值从 0 增加到 1。当控制穿过被保护体右括号后，终止处理程序执行，又把 **counter** 减到 0。打印出的值将为 0。

下面是一个稍复杂的修改：

```
VOID RandomFunction(PLONG pcounter)
{
    __try
```

```

{
    ++*pcounter;
    return;
}
__finally
{
    --*pcounter;
}
}

```

该函数的结果是： **pcounter** 指向的整型值不变，不管控制以何种原因离开被保护体，包括通过 **return** 语句或 **goto** 语句，终止处理程序都将执行。开始，被保护体增加计数器值并执行一个 **return** 语句，接着清除代码执行并减计数器值，之后该子程序才真正返回。

下面例子可以加深你对 **try-finally** 语句的理解：

```

static LONG counter = 0;
__try
{
    ++counter;
    BadActor();
}
__finally
{
    --counter;
}

```

这里我们调用了 **BadActor** 函数，我假定该函数将导致某种异常，这将触发堆栈回卷。作为回卷“执行和异常堆栈”过程的一部分，操作系统将调用我们的恢复代码并把 **counter** 恢复到以前的值。然后操作系统继续回卷堆栈，所以不论我们在 **__finally** 块后有什么代码都得不到执行。

Try-Except块

结构化异常处理的另一种使用方式是 **try-except** 块：

```

__try
{
    <guarded body>
}
__except(<filter expression>)
{
    <exception handler>
}

```

try-except 块中的被保护代码可能会导致异常。你可能调用了象 **MmProbeAndLockPages** 这类的内核模式服务函数，这些函数使用来自用户模式的指针，而这些指针并没有做过明确的有效性检测。也许是因为其它原因。但不管什么原因，如果程序在通过被保护代码段时没有发生任何错误，那么控制将转到异常处理代码后面继续执行，你可以认为这是正常情况。如果在你的代码中或任何你调用的子例程中发生了异常，操作系统将回卷堆栈，并对 **__except** 语句中的过滤表达式求值。结果将是下面三个值中的一个：

- **EXCEPTION_EXECUTE_HANDLER** 数值上等于 1，告诉操作系统把控制转移到你的异常处理代码。如果控制走到处理程序的右大括号之外(如执行了 **return** 语句或 **goto** 语句)，那么控制将转到紧接着异常处理代码的后面继续执行。(我看过了平台 **SDK** 中关于异常控制返回点的文档，但那不正确)
- **EXCEPTION_CONTINUE_SEARCH** 数值上等于 0，告诉操作系统你不能处理该异常。系统将继续扫描堆栈以寻找其它处理程序。如果没有找到为该异常提供的处理程序，系统立即崩溃。

- `EXCEPTION_CONTINUE_EXECUTION` 数值上等于-1，告诉操作系统返回到异常发生的地方。关于这个值我稍后再谈。

图 3-5 显示了 `try-except` 块中可能出现的控制路径。

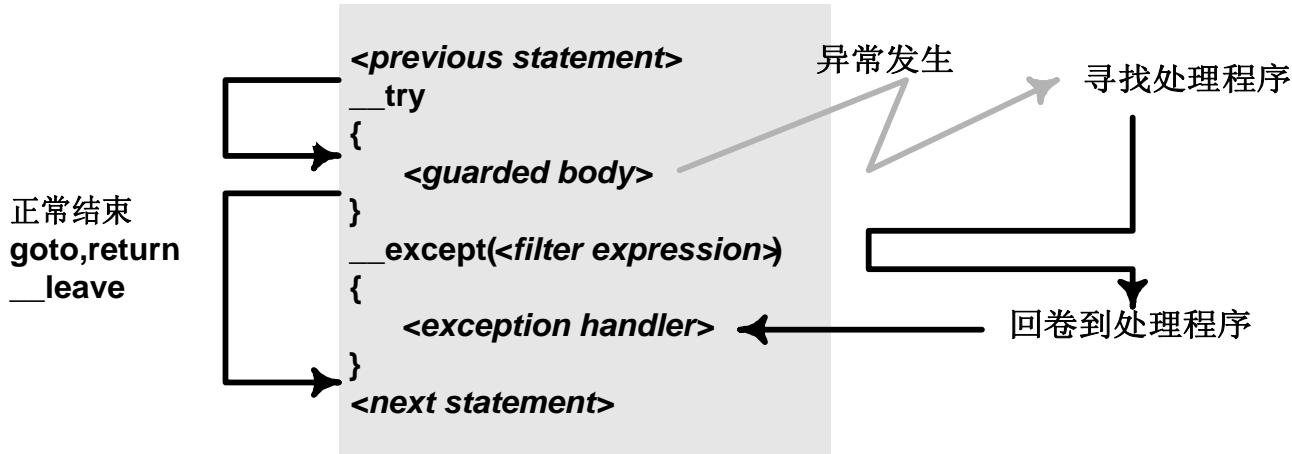


图 3-5. `try-except` 块中的控制流程

例如，下面代码演示了如何防止接收非法指针。(见光盘中的 SEHTEST 例子)

```

PVOID p = (PVOID) 1;
__try
{
    KdPrint("About to generate exception\n");
    ProbeForWrite(p, 4, 4);
    KdPrint("You shouldn't see this message\n");
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
    KdPrint("Exception was caught\n");
}
KdPrint("Program kept control after exception\n");
  
```

ProbeForWrite 测试一个数据区域是否有效。在这个例子中，它将导致一个异常，因为我们提供的指针参数没有以 4 字节边界对齐。然后，异常处理程序得到控制。最后，异常处理完成后控制将转到异常处理程序后面的代码。

在上面的例子中，如果你返回 `EXCEPTION_CONTINUE_SEARCH`，操作系统将继续回卷堆栈以寻找适合的异常处理程序。这时，异常处理程序和跟在它后面的代码都得不到控制，此时或者系统崩溃或者由更高级的处理程序接管控制。

不能在内核模式中返回 `EXCEPTION_CONTINUE_EXECUTION`，因为你没有办法改变导致异常的情况，所以就不能实现重试。

注意，你不能用结构化异常捕获算术异常、页故障，和非法指针引用等等。你必须保证你的代码不产生这样的异常。

异常过滤表达式

你也许会惊奇，仅对一个能产生三种值的表达式求值，如何能执行麻烦的错误检测和修正。你可以用 C/C++ 的逗号操作符把多个表达式串联起来：

```

__except(expr-1, ... EXCEPTION_CONTINUE_SEARCH){}
  
```

原始异常处理与 Microsoft 语法

语句 `__try`、`__except`、`__finally` 是 C 语言的 Microsoft 扩展，它简化了原始异常处理机制(系统底层提供的)的使用。在图 3-3 的流程中，显示了对过滤函数的两次调用。一次为了定位异常处理程序，另一次为了回卷堆栈。运行时间库包含了操作系统调用的实际过滤函数。当使用 `__try`、`__except`、`__finally` 语句时，实际上你正与运行时间库中的某些函数对话。`__except` 子句中的过滤表达式在每次异常发生时只求值一次。

逗号操作符总是放弃它左边的表达式而对右边表达式求值。所以，最后一个表达式的值就是整个表达式的求值结果。

你可以使用 C/C++ 条件操作符来执行更复杂的表达式计算：

```
__except(<some-expr> ? EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
```

如果 `some_expr` 表达式的值为 `TRUE`，则执行你自己的处理程序。否则，通知操作系统继续寻找堆栈中的其它异常处理程序。

当然，你也可以写一个返回 `EXCEPTION_Xxx` 值的子程序：

```
LONG EvaluateException()
{
    if (<some-expr>)
        return EXCEPTION_EXECUTE_HANDLER;
    else
        return EXCEPTION_CONTINUE_SEARCH;
}

...
__except(EvaluateException())
...
```

如果你需要获得更多的关于异常的信息，有两个函数可以在 `__except` 的求值表达式中调用，它们可以提供本次异常的相关信息。实际上，这两个函数是在 Microsoft 编译器的内部实现的，所以仅能用于特定时刻：

- **GetExceptionCode()** 返回当前异常的数值代码。该值是一个 `NTSTATUS` 值。该函数仅在 `__except` 表达式和其后的异常处理代码中有效。
- **GetExceptionInformation()** 返回 `EXCEPTION_POINTERS` 结构的地址，该结构包含异常的所有详细信息，在哪发生、发生时寄存器的内容，等等。该函数仅在 `__except` 表达式中有效。

注意

C/C++ 语言的名称范围规则同样适用于 `try-except` 和 `try-finally` 块中的名称。你可以在 `__try` 后面的复合语句内部声明变量，但这些变量名在过滤表达式、异常句柄，或终止句柄中都是不可见的。而平台 SDK 或 MSDN 中关于这点的叙述是不正确的。有价值的是：所有在被保护代码段中声明的局部变量，其堆栈帧在过滤表达式求值时期仍然存在，所以如果你有一个指向被保护代码段内部声明变量的指针(假定该指针已在外部声明)，那么你仍能在过滤表达式中安全地引用这些变量。

由于这两个函数在使用上的限制，你可以以调用某过滤函数的形式使用它们，象下面这样：

```
LONG EvaluateException(NTSTATUS status, PEXCEPTION_POINTERS xp)
{
    ...
}
```

```
__except(EvaluateException(GetExceptionCode(), GetExceptionInformation()))  
{...}
```

生成异常

程序中的 bug 可以导致异常并使系统调用异常处理机制。应用程序开发者应该熟悉 Win32 API 中的 **RaiseException** 函数，它可以生成任意异常。在 WDM 驱动程序中，你可以调用表 3-1 列出的例程。由于下面规则，我不能给你举一个使用这些函数的例子：

仅当你知道存在一个异常处理代码并知道你真正正在做什么时，才可以在非任意线程上下文下生成一个异常。

表 3-1. 用于生成异常的服务函数

服务函数	描述
ExRaiseStatus	用指定状态代码触发异常
ExRaiseAccessViolation	触发 STATUS_ACCESS_VIOLATION 异常
ExRaiseDatatypeMisalignment	触发 STATUS_DATATYPE_MISALIGNMENT 异常

特别地，不要通过触发异常来告诉你的调用者你在普通执行状态中的信息，你完全可以返回状态代码。应该尽量避免使用异常，因为堆栈回卷机制非常消耗资源。

一些真实环境中的例子

尽管建立异常帧然后撕去异常帧会非常消耗资源，但在特殊情况下，驱动程序必须使用结构化异常语法。并且，在一些时间不是特别重要的场合中，如果想要得到更好的程序也可以使用结构化异常机制。

有一个地方你必须使用结构化异常处理机制，那就是当调用 **MmProbeAndLockPages** 函数锁定被 MDL(内存描述符表)使用的内存页时，必须建立一个异常处理例程。对于 WDM 驱动程序，这个问题不经常出现，因为你使用的 MDL 都已经被其它程序探测并锁定(probe-and-lock)过。但是，由于你可以定义使用 **METHOD_NEITHER** 缓冲方法的 I/O 控制(IOCTL)操作，所以你必须按下面方式写代码：

```
PMDL mdl = MmCreateMdl(...);  
__try  
{  
    MmProbeAndLockPages(mdl, ...);  
}  
__except(EXCEPTION_EXECUTE_HANDLER)  
{  
    NTSTATUS status = GetExceptionCode();  
    ExFreePool((PVOID) mdl);  
    return CompleteRequest(Irp, status, 0);  
}
```

CompleteRequest 是用于实现 I/O 请求完成机制的辅助函数。在第五章我将详细解释 I/O 请求和完成 I/O 请求。**ExFreePool** 是一个内核模式服务例程，它释放由 **MmCreateMdl** 等函数创建的内存块。

另一个真实环境中的例子，考虑一下我在本章前面提到的关于 **AddDevice** 函数的错误处理代码。当你进入这个函数后，所有累积的副作用在发现一个错误后都必须消除。结构化异常处理可以使这个函数更具维护性。下面代码略去了一些不相关代码而着重错误处理：

```
NTSTATUS AddDevice(...)  
{  
    NTSTATUS status = STATUS_UNSUCCESSFUL;  
    PDEVICE_OBJECT fdo;
```

```

PDEVICE_EXTENSION pdx;
status = IoCreateDevice(..., &fdo);
if (!NT_SUCCESS(status))
    return status;
__try
{
    pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    ...
    IoInitializeRemoveLock(&pdx->RemoveLock, ...);
    status = IoRegisterDeviceInterface(..., &pdx->ifname);
    if (!NT_SUCCESS(status))
        return status;
    ...
}
__finally
{
    if (!NT_SUCCESS(status))
    {
        ...
        if (pdx->ifname.Buffer)
            RtlFreeUnicodeString(&pdx->ifname);
        IoDeleteDevice(fdo);
    }
}
return status;
}

```

这里的关键思想是：一旦我们发现某个服务函数执行失败，我们仅仅执行一个 **return status** 语句。**return status** 语句触发了终止处理程序的执行，这个处理程序消除了所有累积的副作用。为了利用这种技术，必须先做两件事：第一，由于终止处理程序总是被执行，即便被保护代码无错误发生时也是这样，所以必须知道何时消除副作用何时不用。在这里我们通过测试 **status** 变量。如果状态为成功，我们不必做任何清除工作，否则，我们必须彻底清除由于异常给程序带来的副作用。要做的第二件事是了解需要清除哪一个副作用。我们把所有可能产生副作用的变量都初始化为 **NULL**。如果在寄存一个设备接口时失败，那么 **pdx->ifname** 中将不会存在一个串，从而也不需要释放操作，如此等等。

在上面情况中使用 **try-finally** 块的一个最大好处是易于代码修改。你可以在 **IoCreateDevice** 调用和 **IoRegisterDeviceInterface** 调用间加入任何语句，这些语句在成功执行后也可能对函数造成副作用。正确的清除操作就是在终止处理程序中加入与产生副作用相反的补偿语句。相反，如果不用结构化异常块语句，那么你必须在每次状态代码测试后都加入明确的清除代码，这更容易导致错误，你可能从这样的地方退出函数，因此不得不记住每个加入了清除语句的代码位置。

__Leave 语句

Microsoft 在其 C/C++ 语言中加入了 **__leave** 语句，它解决了文中 **AddDevice** 例程出现的效率问题。如果在 **__try** 块中发出一个普通的 **return** 语句，将触发昂贵的回卷机制。然而，**__leave** 语句可以直接把控制传递到终止处理程序，最后到达终止处理程序后面的语句。由于它不造成任何回卷动作，所以要比 **return** 语句快得多。在这个例子中，我们总是执行终止处理程序并返回一个状态代码。因为我们在代码执行成功或失败后都要执行相同的语句：**return status**，所以可以用 **__leave** 代替 **return**。

如果要分配一块内存，我们只需向 **AddDevice** 直接插入一些语句即可(粗体部分)：

```

NTSTATUS AddDevice(...)
{
    NTSTATUS status = STATUS_UNSUCCESSFUL;
    PDEVICE_OBJECT fdo;

```

```

PDEVICE_EXTENSION pdx;
status = IoCreateDevice(..., &fdo);
if (!NT_SUCCESS(status))
    return status;
__try
{
    pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    ...
pdx->DeviceDescriptor = (PUSB_DEVICE_DESCRIPTOR)
ExAllocatePool(NonPagedPool, sizeof(USB_DEVICE_DESCRIPTOR));
if (!pdx->DeviceDescriptor)
    return STATUS_INSUFFICIENT_RESOURCES;
IoInitializeRemoveLock(&pdx->RemoveLock, ...);
status = IoRegisterDeviceInterface(..., &pdx->ifname);
if (!NT_SUCCESS(status))
    return status;
...
}
__finally
{
    if (!NT_SUCCESS(status))
    {
        ...
        if (pdx->ifname.Buffer)
            RtlFreeUnicodeString(&pdx->ifname);
        if (pdx->DeviceDescriptor)
            ExFreePool((PVOID) pdx->DeviceDescriptor);
        IoDeleteDevice(fdo);
    }
}
return status;
}

```

如果不使用结构化异常，那么在剩下的程序中，你必须在每个返回错误的代码后面都加入一个 **ExFreePool** 调用。

Bug Checks

Bug check 是系统检测到的错误，一旦发现这种错误，系统立即以一种可控制的方式关闭。许多内核模式部件运行时都进行一致性检测，如果某个系统部件发现一个不可恢复的错误，将生成一个 **bug check**。如果可能，所有内核模式部件都先登记遇到的错误，然后继续运行，而不是调用 **KeBugCheckEx**，除非这种错误将使系统本身变得不可靠。程序可以在任何 **IRQL** 上调用 **KeBugCheckEx**。如果程序发现一个不可恢复的错误，并且该程序继续运行将会破坏系统，那么该程序就调用 **KeBugCheckEx** 函数，这个函数将使系统以一种可控制的方式关闭。

当内核模式中出现不可恢复错误时，会出现一个称为死亡蓝屏(**BSOD blue screen of death**)的画面，驱动程序开发者应该十分熟悉它。图 3-6 就是一个例子(出自手写，因为在这种情况下根本不可能运行屏幕截取软件)。在内部，这种错误被称为 **bug check**，它的主要特征是，系统尽可能以正常的方式关闭并弹出一个死亡蓝屏。一旦死亡蓝屏出现，则表明系统已经死掉必须重启动。

```
*** STOP: 0x000000BE (0xFBB6D898,0x03E34121,0x00000060,0x0000000B)
An attempt was made to write to read-only memory.

*** Address 7BB6D898 base at FBB62000, DateStamp 361e1ad8 - junkola.sys

If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any Windows NT updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Refer to your Getting Started manual for more information on
troubleshooting Stop errors.
```

图 3-6. “蓝屏死亡”

可以按下面方式调用 KeBugCheckEx:

```
KeBugCheckEx(bugcode, info1, info2, info3, info4);
```

bugcode 是一个数值，指出出错的原因，**info1**、**info2** 等是整型参数，将出现在死亡蓝屏中以帮助程序员了解错误细节。该函数从不返回(!)。

我不将解释死亡蓝屏中的信息。你可以在 Art Baker 的《The Windows NT Device Driver Book (Prentice Hall, 1997)》17.3 段中找到更多的信息。Microsoft 自己的 bugcheck 代码在 DDK 头文件 bugcodes.h 中列出；对该代码的更完整解释以及各种参数的含义可以在 KBase 文章 Q103059 “Descriptions of Bug Codes for Windows NT”中找到。

如果需要，你也可以创建自己的 bugcheck 代码。Microsoft 定义的值是从 1(APC_INDEX_MISMATCH)到 0xDE(POOL_CORRUPTION_IN_FILE_AREA)之间的整数。为了创建你自己的 bugcheck 代码，你需要定义一个整型常量(类似 STATUS_SEVERITY_SUCCESS 的状态代码)，并指出 customer 标志或非 0 的 facility 代码。例如：

```
#define MY_BUGCHECK_CODE 0x002A0001
...
KeBugCheckEx(MY_BUGCHECK_CODE, 0, 0, 0, 0);
```

使用非 0 的 facility 代码(例子中为 42)或 customer 标志(例子中为 0)是为了与 Microsoft 使用的代码区分开。

现在，我已经告诉你如何生成自己的 BSOD，那么我们什么时候使用它呢？回答是决不，或者仅在驱动程序的内部调试中使用。我们不可能写出这样的驱动程序，它发现了一个错误并且只有通过关闭系统才能解决。更好的做法是记录这个错误(使用错误登记工具，将在第九章中描述)并返回一个状态码。

内存管理

这一节我们讨论内存管理。Windows 2000 采用多种方式分割虚拟地址空间。一种方式是基于安全性和完整性，有两种地址：用户模式地址和内核模式地址。另一种方式基于处理器的分页能力，有两种内存：分页内存和非分页内存。全部用户模式地址和某些内核模式地址使用分页内存，内存管理器可以在分页内存页帧和磁盘扇区间交换内容，而另一些内核模式地址总是引用物理内存中固定的页帧。由于 Windows 2000 允许驱动程序的某些部分存在于分页内存中，所以我将阐述如何在编译时和运行时控制驱动程序的分页属性。

Windows 2000 提供了多种内存管理方法。我将描述其中的两个基本服务函数，ExAllocatePool 和 ExFreePool，你可以用它们从堆上分配和释放任意大小的内存块。我还将在描述把内存块组织成结构链表的原语。最后，我将描述 lookaside(后援式)链表的概念，它可以使你高效率地分配和释放相同大小的内存块。

用户模式地址空间与内核模式地址空间

Windows NT 和 Windows 98 都是运行在支持虚拟地址空间的计算机上，虚拟地址空间或者映射到一段真实的物理内存，或者映射到交换文件中的页帧。为了简化问题，可以认为虚拟地址空间由两部分组成：内核模式部分和用户模式部分。见图 3-7。

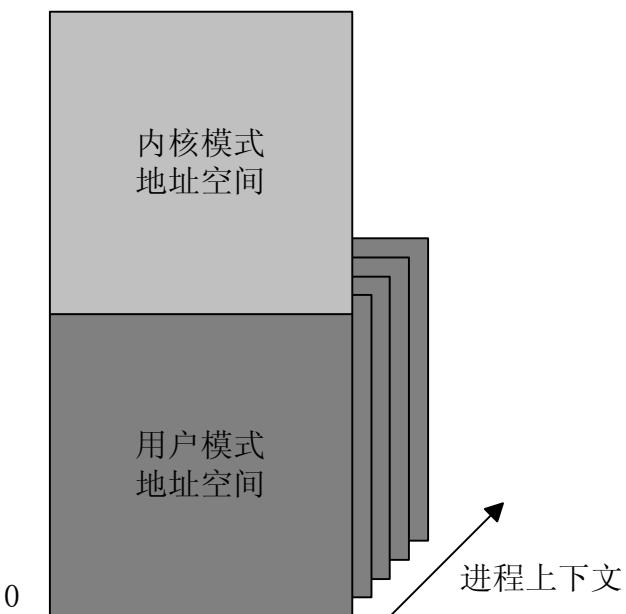


图 3-7. 地址空间中用户模式部分和内核模式部分

每个用户模式进程都有自己的地址上下文，它把用户模式的虚拟地址映射成一组唯一的物理页帧。这意味着，当 Windows NT 调度器把控制从一个进程的当前线程切换到另一个进程的某个线程时，与进程相对应的虚拟地址空间也被更换。线程切换的一个步骤就是改变处理器当前使用的页表，以便它能引用新线程的进程上下文。

注意

如果你熟悉 Alpha，那么你可能不认同我的叙述。Alpha 没有页表概念，但 Alpha 中有一个称为转换缓冲区(translation buffers)的部件，该部件把虚拟页地址映射到物理页地址。对我来说，这与 PC 没有什么不同，这就象说：《奥德修》是由另一个叫荷马的人写的，而不是由历史学家以前认为的那个荷马写的。

一般情况下，WDM 驱动程序不太可能执行在 I/O 请求发起者的线程上下文中。我们之所以说“运行在任意线程上下文”是因为我们不知道当前用户模式地址上下文到底属于哪个进程。我们不能简单地使用属于用户模式中的虚拟地址，因为我们没有办法知道它指向的是哪块物理内存。由于这种不确定性，在编写驱动程序时我们要遵守下面原则：

决不(或几乎从不)直接引用用户模式的内存地址

即，不使用用户模式应用程序提供的任何地址。我将在以后章节中讨论用户模式数据缓冲区的访问技术。现在，我们所要知道的就是，无论何时我们需要访问计算机内存，都要使用内核模式的虚拟地址。

一页有多大？

在虚拟内存系统中，操作系统以固定大小的页帧组织物理内存和交换文件。在 WDM 驱动程序中，常量 PAGE_SIZE 指出页的大小。在某些 Windows NT 计算机中，一页有 4096 字节；在另一些计算机中，一页有 8192 字节。有一个相关常量 PAGE_SHIFT，你可以从下面语句中看出它的值：

```
PAGE_SIZE == 1 << PAGE_SHIFT
```

下面预处理宏可以简化页大小的使用：

- ROUND_TO_PAGES 把指定值舍入为下一个页边界。例如，在 4KB 页的计算机上，ROUND_TO_PAGES(1)的结果为 4096，ROUND_TO_PAGES(4097)的结果为 8192。
- BYTES_TO_PAGES 得出给定的字节数需要多少页来保存。例如，BYTES_TO_PAGES(42)在所有平台上都等于 1，而 BYTES_TO_PAGES(5000)在 4KB 页的平台上为 2，在 8KB 页的平台上为 1。
- BYTE_OFFSET 返回虚拟地址的字节偏移部分。例如，在 4KB 页的计算机上，BYTE_OFFSET(0x12345678)的结果为 0x678。
- PAGE_ALIGN 把虚拟地址舍向上一个页边界。例如，在 4KB 页的计算机上，PAGE_ALIGN(0x12345678)的结果为 0x12345000。
- ADDRESS_AND_SIZE_TO_SPAN_PAGES 返回从指定虚拟地址开始的指定字节数所跨过的页数。例如，在 4KB 的计算机上，ADDRESS_AND_SIZE_TO_SPAN_PAGES(0x12345FFF, 2)的结果为 2，因为这两个字节跨过了页边界。

分页和非分页内存

虚拟内存系统的特征就是能使软件有一个比物理内存大得多的虚拟内存空间。为了做到这一点，内存管理器需要在物理内存和磁盘文件间交换页帧。但操作系统的某些部分是不能被分页的，这些内存用来支持内存管理器本身。最明显的例子就是，用于处理页故障的代码和数据结构必须常驻内存。

Windows NT 把内核模式地址空间分成分页内存池和非分页内存池。(用户模式地址空间总是分页的) 必须驻留的代码和数据放在非分页池；不必常驻的代码和数据放在分页池中。Windows NT 为决定代码和数据是否需要驻留非分页池提供了一个简单规则。我将在下一章详细说明这个规则，但在这里先稍提一下：

执行在高于或等于 DISPATCH_LEVEL 级的代码不可以引发页故障。

在驱动程序的 checked 版中，你可以使用 PAGED_CODE 预处理宏(在 wdm.h 中声明)来帮助发现有违背这个规则的代码。例如：

```
NTSTATUS DispatchPower(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PAGED_CODE()
    ...
}
```

PAGED_CODE 包含条件编译语句。在 checked-build 方式中，如果当前 IRQL 太高，它就打印出一行信息并生成一个断言失败。在 free-build 方式中，它不做任何事。如果测试驱动程序时包含 DispatchPower 代码的页正好在内存中，那么你不会发现已经在提升的 IRQL 上调用了 DispatchPower 函数。即使这样，PAGED_CODE 仍能查出问题。如果该页碰巧不在内存中，系统将产生一个 bug check。

Windows 2000 的驱动程序检验特征可以帮助你调试驱动程序，包括程序分段，内存堆的使用，等等。这个特征在本书出版时仍在修改，所以我只能做一些简单讨论。**PAGED_CODE** 宏只能在驱动程序的 checked 版中检测问题，并且只能发现调用它的那点上出现的问题。而驱动程序检验器可以诊断函数任何地方出现的问题，不论是 free 版本还是 checked 版本的驱动程序。

编译时控制分页能力

有时，驱动程序的某些部分必须驻留内存而另一些可以被分页，这就需要一种能控制代码和数据是否分页的方法。通过指导编译器的段分配可以实现这个目的。在运行时，装入器通过检查驱动程序中的段名把段放到你指定的内存池中。此外在运行时调用内存管理器的例程也能实现这个目的。

注意

Win32 执行文件，包括内核模式驱动程序，在内部都是由一个或多个段组合而成。段可以包含代码或数据，通常还会有诸如可读性、可写性、共享性、执行性，等等附加属性。段是指定分页能力的最小单元。当装载一个驱动程序映像时，操作系统把以“page”或“.eda(.edata)”为段名开头的段放到分页池中，除非 HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management 中的 **DisablePagingExecutive** 值被设置(在这种情况下，驱动程序占用的内存不被分页)。在 Windows 2000 中运行 Soft-ICE 需要用这种方式禁止内核分页。但这使得把驱动程序代码或数据误放到分页池中所造成的错误特别难以查找。如果你使用这种调试器，我推荐你最好使用 **PAGED_CODE** 宏和驱动程序检查器。

使编译器把代码放到特定段的传统方法是使用 **alloc_text** 编译指示。但不是每种编译器都支持这个编译指示，判断 DDK 中是否定义了 **ALLOC_PRAGMA** 可以帮助决定能否使用 **alloc_text** 编译指示。这个编译指示可以把驱动程序的单独例程放到特定段中：

```
#ifdef ALLOC_PRAGMA
    #pragma alloc_text(PAGE, AddDevice)
    #pragma alloc_text(PAGE, DispatchPnp)
    ...
#endif
```

上面语句把 **AddDevice** 和 **DispatchPnp** 函数的代码放到分页池中。

Microsoft 的 C/C++ 编译器在 **alloc_text** 的使用上加了两个讨厌的限制：

- 该编译指示必须跟在函数声明后面而不能在前面。你可以把驱动程序中的所有函数集中到一个头文件中，并在包含该头文件的源文件中，在`#include` 语句的后面使用 **alloc_text**。
- 该编译指示仅能用于有 C 连接形式的函数。即，它不能用于类成员函数或 C++ 源文件中未用 **extern "C"** 声明的函数。

控制数据变量的布置需要使用另外一个编译指示：

```
#ifdef ALLOC_DATA_PRAGMA
    #pragma data_seg("PAGE")
#endif
```

data_seg 编译指示使所有在其后声明的静态数据变量进入分页池。这个编译指示与 **alloc_text** 完全不同。一个分页段可以从`#pragma data_seg("PAGE")` 出现的地方开始到`#pragma data_seg()` 出现的地方结束。而 **Alloc_text** 仅应用于单个函数。

在把数据放到分页段前应该仔细考虑，因为这有可能把事情搞得更坏。最小的页单位大小为 **PAGE_SIZE** 长。仅把一点字节放到分页段可能很蠢，因为整个页的内存都会被占用。另外，一个脏页(从磁盘读取后被改变过)在其物理页帧能被重用之前需要写回硬盘。

关于段布置

我发现用 **code_seg** 编译指示安排代码段更方便，它与 **data_seg** 的用法一样，但仅用于代码。你可以象下面这样通知 Microsoft 编译器把函数放到分页池：

```
#pragma code_seg("PAGE")
NTSTATUS AddDevice(...){...}
NTSTATUS DispatchPnp(...){...}
```

AddDevice 和 **DispatchPnp** 函数将进入分页池。为了检测编译器是否是 Microsoft 的编译器，可以测试预定义宏 **_MSC_VER** 是否存在。

加入下面语句，可以恢复到默认的代码段设置安排：

```
#pragma code_seg()
```

类似的，加入下面语句，可以使数据放入通常使用的非分页数据段：

```
#pragma data_seg()
```

顺便提一下，如果某些代码在驱动程序完成初始化后不再需要，可以直接把它插入到 **INIT** 段。例如：

```
#pragma alloc_text(INIT, DriverEntry)
```

该语句强制 **DriverEntry** 函数进入 **INIT** 段。当函数返回后，系统将释放掉它占用的内存。这个小的节省并不十分重要，因为 WDM 的 **DriverEntry** 函数没有多少工作要做。以前的 Windows NT 驱动程序有一个大的 **DriverEntry** 函数，用于创建设备对象，定位资源，配置设备，等等。所以对于老式驱动程序，这个特征可以节省一些内存。

使用 Microsoft Visual C++ 中的 DUMPBIN 工具可以查看驱动程序中有多少分页部分。你的销售人员甚至会吹嘘你们的驱动程序要比竞争对手少使用多少非分页内存。

运行时控制分页能力

表 3-2 列出了一些服务函数，你可以在运行时使用它们调整驱动程序的分页布局。这些函数的功能是释放被不再需要的代码和数据所占用的物理内存。在第八章中，我将讲述如何向电源管理器寄存你的设备，这样，在一段不活动时期后设备可以自动掉电。掉电期间是释放锁定内存页的最佳时期。

表 3-2. 动态锁定和解锁驱动程序占用内存页的例程

服务函数	描述
MmLockPagableCodeSection	锁定含有给定地址的代码段
MmLockPagableDataSection	锁定含有给定地址的数据段
MmLockPagableSectionByHandle	用 MmLockPagableCodeSection 返回的句柄锁定代码段(仅用于 Windows 2000)
MmPageEntireDriver	解锁所有属于某驱动程序的页
MmResetDriverPaging	恢复整个驱动程序的编译时分页属性
MmUnlockPagableImageSection	为一个锁定代码段或数据段解锁

下面我要描述一种用这些函数实现代码分页的方法，其它方法可参考 DDK 文档。首先，把驱动程序的某些例程放到单独命名的代码段，例如：

```
#pragma alloc_text(PAGEIDLE, DispatchRead)
```

```
#pragma alloc_text(PAGEIDLE, DispatchWrite)
...
```

即，定义一个以“PAGE”开头加任意四个字母做后缀为段名的段。然后用 `alloc_text` 编译指示把一些例程放到该段。你可以有任意多个专用分页段，但那会带来许多麻烦的维护问题。

在初始化期间(即，在 `DriverEntry`)，按下面方式锁定分页段：

```
PVOID hPageIdleSection;
NTSTATUS DriverEntry(...)
{
    hPageIdleSection = MmLockPagableCodeSection((PVOID) DispatchRead);
}
```

调用 `MmLockPagableCodeSection` 时，你可以指定任何一个要锁定段中的地址。在 `DriverEntry` 中调用该函数的真正目的是为了获得其返回的句柄，该句柄被保存到全局变量 `hPageIdleSection` 中。在后面，当这个段不再需要存在于内存中时会用到该句柄：

```
MmUnlockPagableImageSection(hPageIdleSection);
```

该调用将解锁包含有 `PAGEIDLE` 段的页并允许它们按需要进出内存。之后，如果你需要它出现在内存中时，可以调用：

```
MmLockPagableSectionByHandle(hPageIdleSection);
```

该调用完成后，`PAGEIDLE` 段将再次进入非分页内存(可以是与上一次不同的物理内存)。注意该函数仅在 windows 2000 中有效，并且必须使用 `ntddk.h` 文件代替 `wdm.h` 文件。对于其它系统，只能使用 `MmLockPagableCodeSection` 函数。

把数据对象放入分页段与上面类似：

```
PVOID hPageDataSection;

#pragma data_seg("PAGE")
ULONG ulSomething;
#pragma data_seg()

hPageDataSection = MmLockPagableDataSection((PVOID) &ulSomething);

MmUnlockPagableImageSection(hPageDataSection);

MmLockPagableSectionByHandle(hPageDataSection);
```

这些内存管理器服务函数后面的主要思想是：你一开始先锁定包含一个或多个页的段，并获得其句柄，而后面的调用将用到该句柄。调用 `MmUnlockPagableImageSection` 并传递同样的句柄你还可以解锁该段占用的页，再次调用 `MmLockPagableSectionByHandle` 又可以锁定该段。

如果你确信全部驱动程序都不必驻留，可以调用 `MmPageEntireDriver` 把驱动程序的所有段都变为分页式。相反，调用 `MmResetDriverPaging` 将把整个驱动程序恢复到编译时的分页属性布局。调用这些函数仅需要一个存在于驱动程序中的某个地址。例如：

```
MmPageEntireDriver((PVOID) DriverEntry);
...
MmResetDriverPaging((PVOID) DriverEntry);
```

如果设备使用中断，那么你在练习使用这些内存管理器例程时需要格外小心。有时会发生假中断，这将导致系统调用不存在的 ISR，这种随机系统崩溃的原因特别难于发现。DDK 推荐的规则是：不要把 ISR 和与其相关的 DPC 代码所占用的内存置成分页式。

堆分配符

内核模式中的基本堆分配函数是 **ExAllocatePool**。调用方式如下：

```
PVOID p = ExAllocatePool(type, nbytes);
```

type 参数是表 3-3 中列出的 POOL_TYPE 枚举常量，**nbytes** 是要分配的字节数。返回值是一个内核模式虚拟地址指针，指向已分配的内存块。如果内存不足，则返回一个 NULL 指针。如果指定的内存池类型为“must succeed”类型，即 **NonPagedPoolMustSucceed** 或 **NonPagedPoolCacheAlignedMustS**，那么内存不足将导致一个代码为 MUST_SUCCEED_POOL_EMPTY 的 bug check。

注意

驱动程序不应该分配“must succeed”类型内存。驱动程序不应使系统在低内存状态下崩溃。另外，整个系统中仅存在有限的“must succeed”内存。实际上，Microsoft 希望他们从来就没有公布过“must succeed”内存类型。

表 3-3. *ExAllocatePool* 的内存池类型参数

内存池类型	描述
NonPagedPool	从非分页内存池中分配内存
PagedPool	从分页内存池中分配内存
NonPagedPoolMustSucceed	从非分页内存池中分配内存，如果不能分配则产生 bugcheck
NonPagedPoolCacheAligned	从非分页内存池中分配内存，并确保内存与 CPU cache 对齐
NonPagedPoolCacheAlignedMustS	与 NonPagedPoolCacheAligned 类似，但如果不能分配则产生 bugcheck
PagedPoolCacheAligned	从分页内存池中分配内存，并确保内存与 CPU cache 对齐

调用 **ExAllocatePool** 时的最基本原则是被分配内存块是否可以交换出内存。这取决于驱动程序的那一部分需要访问这块内存。如果在大于或等于 DISPATCH_LEVEL 级上使用该内存块，那么必须从非分页池中分配内存。如果你总是在低于 DISPATCH_LEVEL 级上使用内存块，那么既可以从非分页池中分配内存也可以从分页池中分配内存。

你获得的内存块至少是按 8 字节边界对齐的。如果把某结构的实例放到分配的内存中，那么编译器赋予结构成员的 4 或 8 字节偏移在新内存中也将是 4 或 8 字节偏移。但在某些 RISC 平台上，结构成员可能以双字和四字对齐。出于性能上的考虑，希望内存块能适合处理器 cache 行的最少可能数，使用 **XxxCacheAligned** 类型代码可以达到这个要求。如果请求的内存多于一页，那么内存块将从页的边界开始。

释放内存块

调用 **ExFreePool** 可以释放由 **ExAllocatePool** 分配的内存块：

```
ExFreePool((PVOID) p);
```

你确实需要记录分配的内存以便在该内存不再需要时释放它，因为没有人为你做这些事。例如，在 **AddDevice** 函数中，有一个 **IoRegisterDeviceInterface** 调用，该函数存在副作用：它分配了一块内存以保存接口名。你有责任在以后释放该内存。

不用说，访问从内核模式内存池中分配来的内存必须格外小心。因为驱动程序代码可能执行在处理器的最高特权模式下，在这里，系统对内存数据没有任何保护。

ExAllocatePoolWithTag

调用 **ExAllocatePool** 是从内核模式堆中分配内存的标准方式。另一个函数 **ExAllocatePoolWithTag**，与 **ExAllocatePool** 稍有不同，它提供了一个有用的额外特征。当使用 **ExAllocatePoolWithTag** 时，系统在你要求的内存外又额外地多分配了 4 个字节的标签。这个标签占用了开始的 4 个字节，位于返回指针所指向地址的前面。调试时，如果你查看分配的内存块会看到这个标签，它帮助你识别有问题的内存块。例如：

```
PVOID p = ExAllocatePoolWithTag(PagedPool, 42, 'KNUJ');
```

在这里，我使用了一个 32 位整数常量作为标签值。在小结尾的计算机如 x86 上，组成这个标签的 4 个字节的顺序与正常拼写相反。

WDM.H 中声明的内存分配函数受一个预处理宏 **POOL_TAGGING** 控制。WDM.H(NTDDK.H 中也是)中无条件地定义了 **POOL_TAGGING**，结果，无标签的函数实际上是宏，它真正执行的是有标签函数并加入标签‘**mdW**’(指明为 WDM 的内存块)。如果在未来版本的 DDK 中没有定义 **POOL_TAGGING**，那么带标签函数将成为无标签函数的宏。Microsoft 现在还没打算改变 **POOL_TAGGING** 的设置。

由于 **POOL_TAGGING** 宏的存在，当你在程序中调用 **ExAllocatePool** 时，最终被调用的将是 **ExAllocatePoolWithTag**。如果你关闭了该宏，自己去调用 **ExAllocatePool**，但 **ExAllocatePool** 内部仍旧调用 **ExAllocatePoolWithTag** 并带一个‘**enoN**’(即 **None**)的标签。因此你无法避免产生内存标签。所以你应该明确地调用 **ExAllocatePoolWithTag** 并加上一个你认为有意义的标签。实际上，Microsoft 强烈鼓励你这样做。

ExAllocatePool 的其它形式

尽管 **ExAllocatePoolWithTag** 函数是分配堆内存时应该使用的函数，但在某些特殊场合你也可以使用该函数的另外两种形式：

- **ExAllocatePoolWithQuota** 分配一块内存并充入当前线程的调度配额中，该函数仅用于顶层驱动程序，如文件系统驱动程序或其它运行在非任意线程上下文中的驱动程序。
- **ExAllocatePoolWithQuotaTag** 同上，但加入一个标签。

链表

Windows NT 广泛使用链表，用链表把一些相似的数据结构组织起来。在这章中，我将讨论管理双链表和单链表的一些基本服务函数。另一些服务函数可以使多线程和多处理器共享链表；我将在下一章解释这些函数。

通常，不管你是用单链表还是用双链表组织数据，都需要把一个子结构(用于连接链表的连接元素)——或者是 **LIST_ENTRY**，或者是 **SINGLE_LIST_ENTRY**——嵌入到你自己的数据结构中。另外你还需要在某处保存一个链表头，它与连接元素有相同的结构。下面是一个例子：

```
typedef struct _TWOWAY
{
    ...
    LIST_ENTRY linkfield;
    ...
} TWOWAY, *PTWOWAY;

LIST_ENTRY DoubleHead;

typedef struct _ONEWAY
{
    ...
}
```

```

SINGLE_LIST_ENTRY linkfield;
...
} ONEWAY, *PONEWAY;

SINGLE_LIST_ENTRY SingleHead;

```

当调用任何一个链表管理函数时，应该总是使用连接域或链表头——决不直接使用包含它的数据结构本身。所以，假设你得到了一个 TWO WAY 结构的指针(**pdElement**)，为了把这个结构加入到链表中，应该象下面这样引用嵌入的连接域：

```
InsertTailList(&DoubleHead, &pdElement->linkfield);
```

类似地，当你要从链表中提取一个元素时，真正使用的地址是嵌入连接域的地址。为了得到外层数据结构的地址，可以使用 **CONTAINING_RECORD** 宏。(见图 3-8)

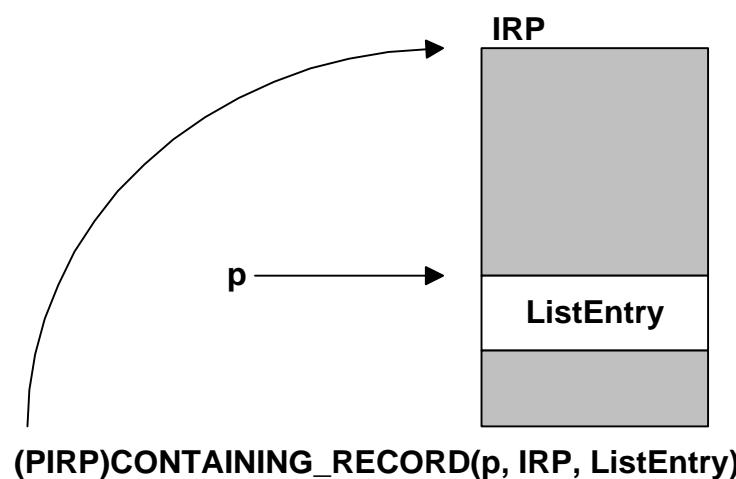


图 3-8. **CONTAINING_RECORD** 宏

所以，如果要丢弃一个单链表中的所有元素，应该象下面这样：

```

PSINGLE_LIST_ENTRY psLink = PopEntryList(&SingleHead);
while (psLink)
{
    PONEWAY psElement = (PONEWAY) CONTAINING_RECORD(psLink, ONEWAY, linkfield);
    ...
    ExFreePool(psElement);
    psLink = PopEntryList(&SingleHead);
}

```

在开始循环前和每次循环中间，我们都调用 **PopEntryList** 来获得当前链表的第一个元素。**PopEntryList** 返回 **ONEWAY** 结构中内嵌连接域的地址，如果链表为空，则返回 **NULL**。注意，不要不加选择地使用 **CONTAINING_RECORD** 宏，应该先检查 **PopEntryList** 返回的连接域地址是否为 **NULL**。

双链表

双链表以两个方向把元素连接起来并形成一个环形拓扑。见图 3-9。即从任何一个元素开始，以两个相反的方向前进，最终都能回到原来的地方。双链表的关键特征是可以在链表的任何地方添加或删除元素。

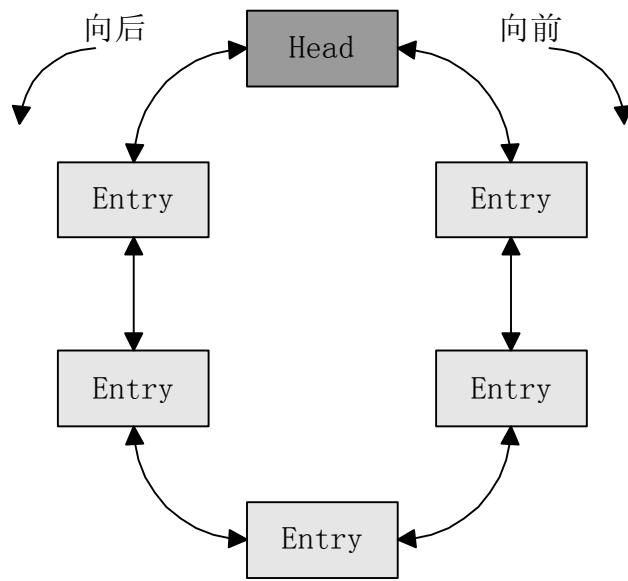


图 3-9. 双链表的拓扑结构

表 3-4 列出了管理双链表的服务函数。

表 3-4. 双链表服务函数

服务函数或宏	描述
InitializeListHead	初始化链表头中的 LIST_ENTRY 结构
InsertHeadList	在起始处插入一个元素
InsertTailList	在结尾处插入一个元素
IsListEmpty	判断链表是否为空
RemoveEntryList	删除元素
RemoveHeadList	删除第一个元素
RemoveTailList	删除最后一个元素

下面例子程序演示了这些函数的用法：

```

typedef struct _TWOWAY {
    ...
    LIST_ENTRY linkfield;
    ...
} TWOWAY, *PTWOWAY;

LIST_ENTRY DoubleHead;
InitializeListHead(&DoubleHead);                                     <--1
ASSERT(IsListEmpty(&DoubleHead));

PTWOWAY pdElement = (PTWOWAY) ExAllocatePool(PagedPool, sizeof(TWOWAY));
InsertTailList(&DoubleHead, &pdElement->linkfield);                  <--2

...
if (!IsListEmpty(&DoubleHead))                                         <--3
{
    PLIST_ENTRY pdLink = RemoveHeadList(&DoubleHead);
    pdElement = CONTAINING_RECORD(pdLink, TWOWAY, linkfield);          <--4
    ...
    ExFreePool(pdElement);
}
    
```

1. **InitializeListHead** 初始化一个 LIST_ENTRY 结构并使其指向自身。这种情况代表链表为空。
2. **InsertTailList** 把一个元素添加到链表尾部。注意，要用嵌入连接域的地址而不是 TWO WAY 结构的地址。调用 **InsertHeadList** 可以把新元素插入到链表头部。实际上，如果给出链表中某个 TWO WAY 结构的内嵌连接域地址，你可以把新元素插入该结构的前面或后面。
3. 空的双链表仅有一个指向自身的链表头。而 **IsListEmpty** 就是做这样的检测。注意，**RemoveXxxList** 的返回值永远都不会为 NULL！
4. **RemoveHeadList** 断开该元素与链表的连接，并返回该元素内嵌连接域的地址。**RemoveTailList** 与其相反。

我们应该了解 RemoveHeadList 和 RemoveTailList 的实现细节，这样可以避免一些错误。例如，观察下面语句：

```
if (<some-expr>)
    pdLink = RemoveHeadList(&DoubleHead);
```

很明显，上面语句的意图是有条件地从链表中删除第一个元素。但是，当调试这段代码时，你会发现第一个链表元素总是神秘地消失，**pdLink** 会在 **if** 表达式为 TRUE 时被更改，但是 RemoveHeadList 看起来好象在 **if** 表达式为 FALSE 时也被调用。

天哪！怎么回事？实际上 RemoveHeadList 是一个宏，在预编译后被扩展成多个语句。下面是编译器真正看到的程序：

```
if (<some-expr>
    pdLink = (&DoubleHead)->Flink;
{
    PLIST_ENTRY _EX_Blink;
    PLIST_ENTRY _EX_Flink;
    _EX_Flink = ((&DoubleHead)->Flink)->Flink;
    _EX_Blink = ((&DoubleHead)->Flink)->Blink;
    _EX_Blink->Flink = _EX_Flink;
    _EX_Flink->Blink = _EX_Blink;
}
```

啊！现在链表元素神秘消失的原因终于变得明了了。**if** 语句的 TRUE 分支仅有 **pdLink = (&DoubleHead)->Flink** 语句，它把一个指针放到第一个元素中。而删除链表元素的语句块却跑到了 **if** 语句外面，所以这些语句总是被执行。RemoveHeadList 和 RemoveTailList 都被翻译成一个表达式加上一个复合语句的形式，因此你不能在单表达式或单语句的地方使用它们。:-)

其它链表操作函数(宏)没有这个问题。

单链表

单链表以一个方向连接元素，如图 3-10。Windows NT 用单链表实现下推堆栈，表 3-5 列出了下推堆栈的服务函数。与双链表相同，这些“函数”在 wdm.h 中以宏实现。**PushEntryList** 和 **PopEntryList** 也生成多条语句，所以你只能把它们用在等号右边。

表 3-5. 单链表服务函数

服务函数或宏	描述
PushEntryList	向链表顶加入元素
PopEntryList	删除最上面的元素

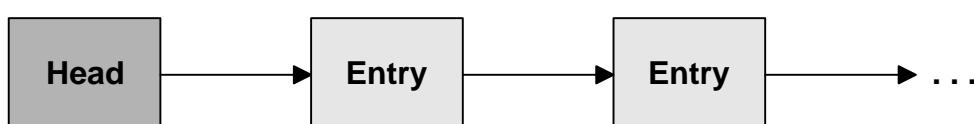


图 3-10. 单链表的拓扑结构

下面代码演示了单链表的使用方法：

```
typedef struct _ONEWAY {
    ...
    SINGLE_LIST_ENTRY linkfield;
} ONEWAY, *PONEWAY;

SINGLE_LIST_ENTRY SingleHead;
SingleHead.Next = NULL;                                <--1

PONEWAY psElement = (PONEWAY) ExAllocatePool(PagedPool, sizeof(ONEWAY));
PushEntryList(&SingleHead, &psElement->linkfield);      <--2

SINGLE_LIST_ENTRY psLink = PopEntryList(&SingleHead);     <--3
if (psLink)
{
    psElement = CONTAINING_RECORD(psLink, ONEWAY, linkfield);
    ...
    ExFreePool(psElement);
}
```

1. 与双链表中的表头初始化不同，单链表仅把表头的 **Next** 域置为 **NULL**。同样，测试单链表是否为空时也仅是测试 **Next** 域本身。
2. **PushEntryList** 把一个元素放到链表头部，链表头部是唯一可以直接访问的链表部分。同样应该仅使用内嵌连接域的地址而不是使用 **ONEWAY** 结构的地址。
3. **PopEntryList** 断开第一个元素与链表的连接并返回该元素中的连接域地址。不同于双链表，单链表用 **NULL** 来代表空链表。单链表没有与双链表 **IsEmpty** 函数对应的函数。

Lookaside(后援式)链表

即使使用最好的算法，堆管理器仍不时地需要一些处理器时间来处理随机出现的内存空洞。图 3-11 显示了这种情况，当内存块 **B** 返回堆时，块 **A** 和 **C** 已经空闲，之后，堆管理器可以把块 **A**、**B**、**C** 组合成一个更大的单块内存。而这个内存块可以满足比单个 **A**、**B**、**C** 都大的内存申请。

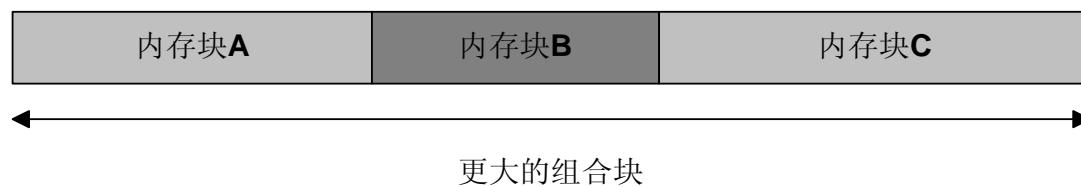


图 3-11. 接合邻近的空闲内存块

如果你总是使用固定大小的内存块，你可以制定一个更有效的方案来管理堆。例如，你可以预先分配一个大内存块，然后再把这块内存细分成多块固定大小的小块内存。然后再设计一个方案来确定哪一块内存是空闲的，哪一块是使用的，就象图 3-12 那样。向这样的堆返回内存块仅仅是把被返回块置成空闲，而不必接合临近空闲块，因为你从不提供随机大小的内存块。

仅仅分配一个大内存块然后再细分成多个小内存块并不是实现固定大小堆的最佳办法。通常，我们难于猜测应预先分配多少内存。如果太大就会浪费内存，如果太小，你的算法会因内存不足而失败，或者频繁地向随机堆管理器申请更多内存。为此，Microsoft 创建了 **lookaside** 链表对象，以及一组相应的算法来克服这些缺点。

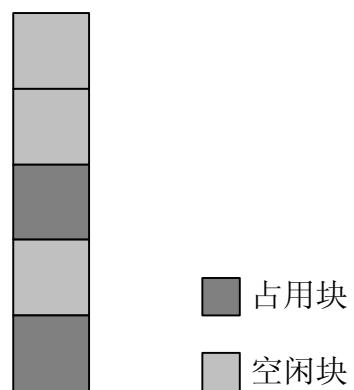


图 3-12. 仅接受固定大小内存请求的堆

图 3-13 显示了 **lookaside** 链表的概念。假设有一个可以在水池中直上直下平衡的玻璃杯子。这个杯子就代表 **lookaside** 链表对象。当初始化该对象时，你告诉系统需要多大的内存块(杯中的水滴)。在早期版本的 Windows NT 中，你还要指出杯子的容量，但现在的操作系统可以自动适应。为了满足一个内存分配请求，系统首先尝试着从链表中取出(删除)一块内存(从杯子中取出一滴水)。如果连一块内存也没有，系统就从外面内存池中取。相反，释放内存时，系统首先尝试着放到链表上(向杯子中加入一滴水)。但是，如果链表满了，那么这个内存块就返回到外界的内存池中(杯子中的水溢出到水池)。

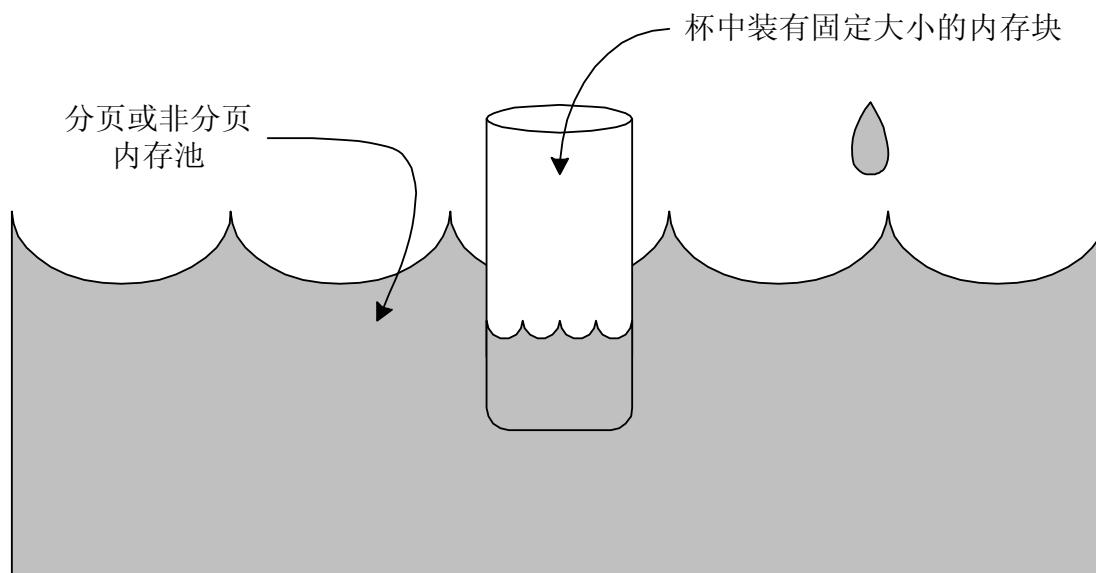


图 3-13. **Lookaside** 链表

系统根据实际的使用量周期性调整 **lookaside** 链表的深度。算法的细节并不重要，而且还会变化。基本上，在当前发行的操作系统中，对于那些最近不常使用的 **lookaside** 链表，系统将降低其深度。但这个深度从不低于 4，这也是新链表的初始深度。

表 3-6 列出了 8 个 **lookaside** 链表服务函数。实际上它们分成两组，每组 4 个函数，一组管理利用分页内存的 **lookaside** 链表(**ExXxxPagedLookasideList**)，另一组管理利用非分页内存的 **lookaside** 链表(**ExXxxNonPagedLookasideList**)。首先要做的是为一个 **PAGED_LOOKASIDE_LIST** 或 **NPAGED_LOOKASIDE_LIST** 对象保留非分页内存。这两个对象十分相似，但分页类型的链表使用 **FAST_MUTEX** 来同步，而非分页类型的链表要使用自旋锁(下一章将讨论这些同步对象)。**PAGED_LOOKASIDE_LIST** 对象所指向的内存块将出现在分页池中，但它自身应出现在非分页内存中，因为系统需要在提升的 IRQL 级上访问它。

表 3-6. **lookaside** 链表的服务函数

服务函数	描述
ExInitializeNPagedLookasideList	初始化 lookaside 链表
ExInitializePagedLookasideList	

ExAllocateFromN PagedLookasideList ExAllocateFromPagedLookasideList	分配一个固定大小的内存块
ExFreeToN PagedLookasideList ExFreeToPagedLookasideList	将一个内存块释放回 lookaside 链表
ExDeleteN PagedLookasideList ExDeletePagedLookasideList	删除 lookaside 链表

为 lookaside 链表对象保留完存储后，应调用相应的初始化函数：

```
PPAGED_LOOKASIDE_LIST pagedlist;
PNPAGED_LOOKASIDE_LIST nonpagedlist;

ExInitializePagedLookasideList(pagedlist, Allocate, Free, 0, blocksize, tag, 0);
ExInitializeNPagedLookasideList(nonpagedlist, Allocate, Free, 0, blocksize, tag, 0);
```

(这两个函数仅在函数名和第一个参数的拼写上不相同)

这两个函数的第一个参数都指向[N]PAGED_LOOKASIDE_LIST 对象，我们已经为该对象保留了存储。**Allocate** 和 **Free** 指向由你写的可以在随机堆中分配和释放内存的例程。如果把它们指定为 **NULL**，系统将使用 **ExAllocatePoolWithTag** 和 **ExFreePool** 来分配和释放内存。**blocksize** 参数是该链表能分配的内存块大小，**tag** 是一个 32 位的标签值，将放到每个这样块的前面。两个 0 值的地方用于与以前版本的 Windows NT 兼容，现在的系统可以自己决定这些值；它们分别是用于控制分配类型的标志和 lookaside 链表深度。

从链表上分配一个内存块，调用 **AllocateFrom** 函数：

```
PVOID p = ExAllocateFromPagedLookasideList(pagedlist);
PVOID q = ExAllocateFromNPagedLookasideList(nonpagedlist);
```

向链表返回一个内存块，调用 **FreeTo** 函数：

```
ExFreeToPagedLookasideList(pagedlist, p);
ExFreeToNPagedLookasideList(nonpagedlist, q);
```

删除链表，调用 **Delete** 函数：

```
ExDeletePagedLookasideList(pagedlist);
ExDeleteNPagedLookasideList(nonpagedlist);
```

经常犯的错误是忘记删除 lookaside 链表。应该在 lookaside 链表的有效范围内删除它。例如，如果在 **AddDevice** 函数中创建了 lookaside 链表，应该把链表对象放到设备对象中并在调用 **IoDeleteDevice** 前删除它。如果在 **DriverEntry** 函数中创建了 lookaside 链表，应该把链表对象放到全局变量中，并在 **DriverUnload** 例程返回前删除它(在 **DriverUnload** 例程的尾部)。

字符串操作

WDM 驱动程序可以使用下面 4 种格式的字符串：

- Unicode 串，由 UNICODE_STRING 结构描述，包含 16 位字符。Unicode 有足够的代码空间编码地球上所有语言的字型。参见<http://www.indigo.ie/egt/standards/csor/klingon.html>。
- ANSI 串，由 ANSI_STRING 结构描述，包含 8 位字符。另一种 OEM_STRING 串与其相似，也是用 8 位字符描述串。两者的不同是，OEM 串中的字符字型由当前代码页决定，而 ANSI 串的字符字型不依赖任何代码页。WDM 驱动程序不必处理 OEM 串，因为它们只能来自用户模式，在驱动程序看到这些串之前，它们已经被某些内核模式部件转换成 Unicode 串。
- 空结尾的字符串。你可以用普通的 C 语法表达串常量，例如，"Hello, world!"，该串使用了类型为 CHAR 的 8 位字符，这些字符被假定属于 ANSI 字符集。串常量中的字符来自你创建源程序所用的编辑器。如果你的编辑器需要依赖当前代码页才能显示编辑窗口中的字符字型，那么应该注意，这些字符在 Windows ANSI 字符集中可能会有不同的含义。
- 空结尾的宽字符(WCHAR 类型)串，用 C 语法也可以表达宽串常量，例如，L"Goodbye, cruel world!"，该串看起来象 Unicode 串常量，但是，最后从文本编辑器出来的串实际仅使用了 Windows ANSI 字符集中 ASCII 和 Latin1 区(0020-007F 和 00A0-00FF)中的字符。

UNICODE_STRING 和 ANSI_STRING 有相同的数据结构，如图 3-14。Buffer 域指向包含串数据的缓冲区。MaximumLength 给出了缓冲区的长度，Length 提供串的当前长度，它不考虑任何可能存在的空结束符。length 域的单位是字节，对于 UNICODE_STRING 结构也是这样。

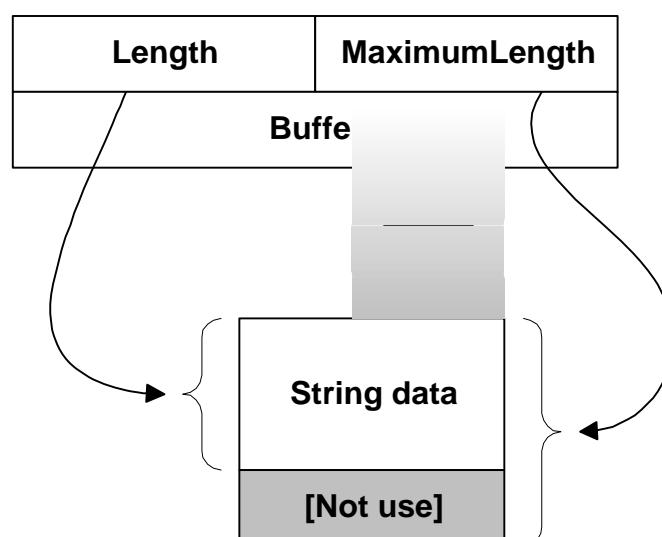


图 3-14. UNICODE_STRING 结构和 ANSI_STRING 结构

表 3-7 列出了用于处理 Unicode 和 ANSI 字符串的服务函数。我还对应列出了一些标准 C 运行时间库函数，这些函数在内核模式中也存在，它们可以处理常规的 C 样式串。虽然它们从来没有在 DDK 文档中公开过，但标准 DDK 头中有这些函数的声明，而且连接库中也包含了它们，所以没有理由不使用它们。

表 3-7. 串处理函数

操作	ANSI 串函数	Unicode 串函数
Length	strlen	wcslen
Concatenate	strcat, strncat	wcscat, wcsncat, RtlAppendUnicodeToString, RtlAppendUnicodeToString
Copy	strcpy, strncpy, RtlCopyString	wcscpy, wcsncpy, RtlCopyUnicodeString
Reverse	_strrev	_wcsrev
Compare	strcmp, strncmp, _strcmp, _strnicmp, RtlCompareString, RtlEqualString	wcsicmp, wcsncmp, _wcsicmp, _wcsncmp, RtlCompareUnicodeString, RtlEqualUnicodeString,

		RtlPrefixUnicodeString
Initialize	_strset, _strnset, RtlInitAnsiString, RtlInitString	_wcsnset, RtlInitUnicodeString
Search	strchr, strrchr, strspn, strstr	wcschr, wcsrchr, wcsspn, wcsstr
Upper/lowercase	_strupr, _strlwr, RtlUpperString	_wcsupr, _wclwr, RtlUpcaseUnicodeString
Character	isdigit, islower, isprint, isspace, isupper, isxdigit, tolower, toupper, RtlUpperChar	towlower, towupper, RtlUpcaseUnicodeChar
Format	sprintf, vsprintf, _snprintf, _vsnprintf	swprintf, _snwprintf
String conversion	atoi, atol, _itoa	_itow, RtlIntegerToUnicodeString, RtlUnicodeStringToInteger
Type conversion	RtlAnsiStringToUnicodeSize, RtlAnsiStringToUnicodeString	RtlUnicodeStringToAnsiString
Memory release	RtlFreeAnsiString	RtlFreeUnicodeString

系统 DLL 还输出了许多 **RtlXxx** 函数，但我仅列出了在 DDK 头文件中定义了原型的函数，在驱动程序中我们仅应使用这些函数。

分配和释放串缓冲区

我不将详细描述这些串处理函数，DDK 中的描述已经十分详细。你可以根据你的串使用经验使用它们。但这里有一个问题我要提一下。

你可能经常把 **UNICODE_STRING** 或 **ANSI_STRING** 类型的串定义成自动变量，或者定义成设备扩展的一部分。而串的数据缓冲区通常是动态分配的内存。但有时候你希望使用串常量，记录谁拥有某个串缓冲区可能是一个问题。考虑下面函数片段：

```
UNICODE_STRING foo;
if (bArriving)
    RtlInitUnicodeString(&foo, L"Hello, world!");
else
{
    ANSI_STRING bar;
    RtlInitAnsiString(&bar, "Goodbye, cruel world!");
    RtlAnsiStringToUnicodeString(&foo, &bar, TRUE);
}
...
RtlFreeUnicodeString(&foo); // <--don't do this!
```

在第一种情况下，我们为驱动程序中的宽字符串常量初始化 **foo.Length**、**foo.MaximumLength**，和 **foo.Buffer**。在另一种情况下，我们要求系统(使用第三个参数为 **TRUE** 的 **RtlAnsiStringToUnicodeString** 调用)为转化 **ANSI** 串分配内存。在第一种情况下，调用 **RtlFreeUnicodeString** 就是一个错误，因为它将无条件地释放常量串“Hello, world!”占用的内存，由于 **foo** 串的 **Buffer** 成员指向一个常量串，不是动态分配的内存，所以根本不能释放。在另一种情况下，由于我们指定了 **TRUE** 参数，系统为表达转换结果动态地分配了 **Unicode** 串缓冲区，所以我们必须调用 **RtlFreeUnicodeString** 释放该缓冲区，以避免内存泄漏。

Blob数据(大块数据)

我从数据库技术中借来术语“*blob*”来描述一块无结构定义的字节组合。表 3-8 列出了处理这种数据的函数(包括一些来自标准运行时间库中的函数)，你可以在内核模式中调用它们。我假设你能从它们的名字上看出它们的用法。然而，我需要指出一些不明显的事：

- 内存的“copy”和“move”操作之间的区别在于可否容忍源和目的相重叠。move 操作不管源和目的是否重叠。而 copy 操作在源和目的有任何重叠时不工作。
- “byte”操作和“memory”操作的区别是操作的间隔尺寸。byte 操作保证按字节为单位执行。而 memory 操作可以在内部使用更大的块，所有这些块的和等于指定的字节数。这个区别会根据平台的不同而改变，在 32 位 Intel 计算机上，byte 操作实际上是对应 memory 操作的宏。但在 Alpha 平台上，**RtlCopyBytes** 与 **RtlCopyMemory** 是完全不同的函数。

表 3-8. 处理 blob 数据的服务函数

服务函数或宏	描述
memchr	在 blob 中寻找一个字节
memcpy, RtlCopyBytes, RtlCopyMemory	复制字节，不允许重叠
memmove, RtlMoveMemory	复制字节，允许重叠
memset, RtlFillBytes, RtlFillMemory	用给定的值填充 blob
memcmp, RtlCompareMemory, RtlEqualMemory	比较两个 blob
memset, RtlZeroBytes, RtlZeroMemory	blob 清零

其它编程技术

在本章的剩余部分我将讨论一些杂项编程技术，你的驱动程序或许会用到这些技术。我将从如何访问注册表开始，注册表保存着许多涉及代码控制和硬件配置的信息。然后讲述如何访问磁盘文件和其它命名设备，如何在 WDM 驱动程序中执行浮点运算。最后，我还要讲述一些驱动程序调试技术。

访问注册表

Windows NT 和 Windows 98 把配置信息和其它重要信息都记录到注册表(registry)中。WDM 驱动程序可以使用表 3-9 列出的函数访问注册表。如果你在用户模式编程中曾涉及过注册表访问，你可能会猜出如何在驱动程序中使用这些函数。然而这些内核模式中的支持函数确有些不同，我想有必要描述一下它们的用法。

表 3-9. 注册表访问函数

服务函数	描述
IoOpenDeviceRegistryKey	打开 PDO 专用键
IoOpenDeviceInterfaceRegistryKey	打开与注册设备接口相连的键
RtlDeleteRegistryValue	删除一个注册表值
RtlQueryRegistryValues	从注册表中读取多个值
RtlWriteRegistryValue	向注册表写一个值
ZwClose	关闭注册表键句柄
ZwCreateKey	创建一个注册表键
ZwDeleteKey	删除一个注册表键
ZwEnumerateKey	枚举子键
ZwEnumerateValueKey	枚举某注册表键中的值
ZwFlushKey	把注册表更改提交到磁盘
ZwOpenKey	打开一个注册表键
ZwQueryKey	取关于某注册表键的信息
ZwQueryValueKey	取某个注册表键中的值
ZwSetValueKey	置某个注册表键中的值

在这一节，我将描述 **ZwXxx** 函数族和 **RtlDeleteRegistryValue** 函数，这些函数基本上可以满足大部分 WDM 驱动程序的需要。

打开注册表键

在读注册表的某个值之前，你需要先打开包含该值的键。用 **ZwOpenKey** 函数可以打开一个已存在的键。

ZwCreateKey 函数既可以打开已存在键又可以创建新键。这两个函数都需要事先用键名以及某些其它信息初始化一个 **OBJECT_ATTRIBUTES** 结构。**OBJECT_ATTRIBUTES** 结构的声明如下：

```
typedef struct _OBJECT_ATTRIBUTES {  
    ULONG Length;  
    HANDLE RootDirectory;  
    PUNICODE_STRING ObjectName;  
    ULONG Attributes;
```

```
PVOID SecurityDescriptor;  
PVOID SecurityQualityOfService;  
} OBJECT_ATTRIBUTES;
```

除了手工初始化这个结构外，你还可以调用 **InitializeObjectAttributes** 宏来初始化它。

例如，假设我们想要打开驱动程序的服务键。它的键名我们可以从 I/O 管理器传递给 **DriverEntry** 函数的一个参数中得到。所以，我们的代码如下：

```
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)  
{  
    ...  
    OBJECT_ATTRIBUTES oa;  
    InitializeObjectAttributes(&oa, RegistryPath, 0, NULL, NULL);           <--1  
    HANDLE hkey;  
    status = ZwOpenKey(&hkey, KEY_READ, &oa);                                <--2  
    if (NT_SUCCESS(status))  
    {  
        ...  
        ZwClose(hkey);                                                 <--3  
    }  
    ...  
}
```

1. 我们用 I/O 管理器提供的注册表路径名和为 **NULL** 的安全描述符来初始化 **OBJECT_ATTRIBUTES** 结构对象 **oa**。
ZwOpenKey 总是忽略安全描述符，安全描述符仅在创建键时才被真正使用。
2. **ZwOpenKey** 以读方式打开该键，并把返回句柄保存到 **hkey** 变量中。
3. **ZwClose** 是关闭内核模式对象句柄的通用例程。在这里，我们用它关闭注册表键句柄。

尽管我们经常把注册表说成数据库，但它并不具备真正数据库的所有特征。例如，它不允许提交或回卷数据更改。另外，在打开键操作中指定的访问权限是用于安全检测而不是为了防止共享冲突。即两个不同的进程能以写方式打开同一个键。然而，系统仍能自动保护与破坏性写操作同时发生的读操作，也能保证使用中的键不被意外删除。

其它打开注册表键的方法

除了 **ZwOpenKey** 函数，Windows 2000 又提供了两个打开注册表键的函数。

IoOpenDeviceRegistryKey 打开某设备对象专用的注册表键：

```
HANDLE hkey;  
status = IoOpenDeviceRegistryKey(pdo, flag, access, &hkey);
```

pdo 是物理设备对象的地址，**flag** 指出你要打开哪一个专用键(见表 3-10)，**access** 是访问掩码，如 **KEY_READ**。

表 3-10. *IoOpenDeviceRegistryKey* 函数使用的注册表键代码

Flag 值	被选择的注册表键
PLUGPLAY_REGKEY_DEVICE	Enum 键中的硬件(或实例)子键
PLUGPLAY_REGKEY_DRIVER	服务(或软件)键

IoOpenDeviceInterfaceRegistryKey 打开与已注册设备接口关联的键：

```
HANDLE hkey;
```

```
status = IoOpenDeviceInterfaceRegistryKey(linkname, access, &hkey);
```

linkname 是已注册接口的符号连接名，**access** 是访问掩码，如 **KEY_READ**。

接口键是 **HKLM\System\CurrentControlSet\Control\DeviceClasses** 下的一个子键，可以常驻注册表。我们可以在这里放入需要与用户模式程序共享的参数信息，用户模式程序可以调用 **SetupDiOpenDeviceInterfaceRegKey** 函数访问该键。

在第十二章中，我将描述如何用安装脚本向硬件和接口键插入值，以及应用程序如何访问这些值。

获取和设置注册表值

通常，打开一个注册表键是为了从注册表中提取某个值。使用 **ZwQueryValueKey** 函数可以达到这个目的。例如，为了从驱动程序的服务键中提取 **ImagePath** 值，你可以参考下面代码：

```
UNICODE_STRING valname;
RtlInitUnicodeString(&valname, L"ImagePath");
size = 0;
status = ZwQueryValueKey(hkey, &valname, KeyValuePartialInformation, NULL, 0, &size);
if (status == STATUS_OBJECT_NAME_NOT_FOUND || size == 0)
    <handle error>;
PKEY_VALUE_PARTIAL_INFORMATION vpip = (PKEY_VALUE_PARTIAL_INFORMATION)
ExAllocatePool(PagedPool, size);
if (!vpip)
    <handle error>;
status = ZwQueryValueKey(hkey, &valname, KeyValuePartialInformation, vpip, size, &size);
if (!NT_SUCCESS(status))
    <handle error>;
<do something with vpip->Data>
ExFreePool(vpip);
```

这里，我们两次调用了 **ZwQueryValueKey** 函数。第一次调用是为了获取 **KEY_VALUE_PARTIAL_INFORMATION** 结构的长度，然后为其分配空间。第二次调用接收其内容。最开始我认为第一次调用 **ZwQueryValueKey** 将返回 **STATUS_BUFFER_TOO_SMALL** 错误码，因为我传给它一个空缓冲区指针 **NULL**，实际上该函数并没有这样做。最重要的错误代码是 **STATUS_OBJECT_NAME_NOT_FOUND**，它指出注册表中不存在该值。如果这里还有其它错误使 **ZwQueryValueKey** 函数不工作，第二次调用时会被发现。

这个所谓的“partial”信息结构包含该注册表值的数据和数据类型描述：

```
typedef struct _KEY_VALUE_PARTIAL_INFORMATION {
    ULONG    TitleIndex;
    ULONG    Type;
    ULONG    DataLength;
    UCHAR    Data[1];
} KEY_VALUE_PARTIAL_INFORMATION, *PKEY_VALUE_PARTIAL_INFORMATION;
```

Type 是表 3-11 中列出的一个注册表数据类型。(可能还有其它数据类型，但设备驱动程序一般不会用到)

DataLength 是数据的长度，**Data** 是数据本身。**TitleIndex** 与驱动程序无关。下面事实可以帮助你了解各种数据类型：

- **REG_DWORD** 是一个 32 位无符号整型，其格式根据平台而变(大结尾或小结尾)。
- **REG_SZ** 是一个空结尾的 Unicode 字符串。这个 NULL 结束符被计入 **DataLength** 中。
- 扩展 **REG_EXPAND_SZ** 值需要使用环境变量替换%-escapes，你应该使用 **RtlQueryRegistryValues** 函数获取这种类型的数据。但访问环境变量的内部例程没有对驱动程序公开。

- `RtlQueryRegistryValues` 也可以用于获取 `REG_MULTI_SZ` 类型值，如果含有多个串，该函数将调用你指定的回调函数来处理。

注意

`RtlQueryRegistryValues` 是一个复杂的函数，我没有为它写例子，DDK 中有几个例子驱动程序用到了该函数。

表 3-11. WDM 驱动程序使用的注册表值类型

数据类型常量	描述
<code>REG_BINARY</code>	可变长二进制数据
<code>REG_DWORD</code>	无符号长整型，格式依平台而变
<code>REG_DWORD_BIG_ENDIAN</code>	无符号长整型，大结尾格式
<code>REG_EXPAND_SZ</code>	空结尾的 Unicode 串，包含%-escapes 环境变量
<code>REG_MULTI_SZ</code>	一个或多个空结尾的 Unicode 串，最后有一个额外的 null
<code>REG_SZ</code>	空结尾的 Unicode 串

为了设置注册表值，你必须对其父键拥有 `KEY_SET_VALUE` 访问权。我前面用过的 `KEY_READ` 并没有这样的权力。你可以使用 `KEY_WRITE` 或 `KEY_ALL_ACCESS`，尽管它们提供的权力有点超出你的需求。然后调用 `ZwSetValueKey`，例如：

```
RtlInitUnicodeString(&valname, L"TheAnswer");
ULONG value = 42;
ZwSetValueKey(hkey, &valname, 0, REG_DWORD, &value, sizeof(value));
```

删除子键或键值

为了删除已打开键中的键值，可以使用 `RtlDeleteRegistryValue` 函数：

```
RtlDeleteRegistryValue(RTL_REGISTRY_HANDLE, (PCWSTR) hkey, L"TheAnswer");
```

`RtlDeleteRegistryValue` 是一个通用的服务函数，它的第一个参数可以指向注册表中的几个特殊位置。当使用 `RTL_REGISTRY_HANDLE` 时，表示你要删除已打开键中的键值。第二参数指出键。第三个参数是一个空结尾的 Unicode 串，它是被删除键值的名称。在这里，你不必为描述该串创建一个 `UNICODE_STRING` 结构。

如果你对某注册表键有 `DELETE` 权限(用 `KEY_ALL_ACCESS` 打开)，可以调用 `ZwDeleteKey` 删除该键：

```
ZwDeleteKey(hkey);
```

被删除键要等到其所有句柄都关闭后才真正消失，所有后来访问或打开该键的请求都将失败，并返回 `STATUS_KEY_DELETED` 错误代码。键删除后，仍需要调用 `ZwClose` 函数关闭其句柄。(DDK 文档中关于 `ZwDeleteKey` 函数的内容指出那个句柄将变为无效，但实际不是这样，你仍要调用 `ZwClose` 关闭它)

枚举子键或键值

枚举键中的元素(子键和键值)是一个较复杂的操作。为此，需要先调用 `ZwQueryKey` 以获得该键子键和键值的一些信息，如个数、最长名的长度，等等。`ZwQueryKey` 有一个参数可以让你指出需要该键的哪种类型的信息。有三种类型：基本信息、节点信息，和全部信息。为了准备枚举操作，你最好指出需要全部信息：

```
typedef struct _KEY_FULL_INFORMATION {
```

```

LARGE_INTEGER LastWriteTime;
ULONG TitleIndex;
ULONG ClassOffset;
ULONG ClassLength;
ULONG SubKeys;
ULONG MaxNameLen;
ULONG MaxClassLen;
ULONG Values;
ULONG MaxValueNameLen;
ULONG MaxValueDataLen;
WCHAR Class[1];
} KEY_FULL_INFORMATION, *PKEY_FULL_INFORMATION;

```

该结构实际上是可变长的，因为 **Class[0]** 只是类名的第一个字符。通常，第一次调用获得要分配缓冲区的大小，第二次调用获得实际的数据，如下：

```

ULONG size;
ZwQueryKey(hkey, KeyFullInformation, NULL, 0, &size);
PKEY_FULL_INFORMATION fip = (PKEY_FULL_INFORMATION) ExAllocatePool(PagedPool, size);
ZwQueryKey(hkey, 0, KeyFullInformation, bip, size, &size);

```

用 subkeys 值做计数器，循环调用 **ZwEnumerateKey**：

```

for (ULONG i = 0; i < fip->SubKeys; ++i)
{
    ZwEnumerateKey(hkey, i, KeyBasicInformation, NULL, 0, &size);
    PKEY_BASIC_INFORMATION bip = (PKEY_BASIC_INFORMATION) ExAllocatePool(PagedPool, size);
    ZwEnumerateKey(hkey, i, KeyBasicInformation, bip, size, &size);
    <do something with bip->Name>
    ExFreePool(bip);
}

```

每个子键的关键信息就是它的名字，存在于 **KEY_BASIC_INFORMATION** 结构中：

```

typedef struct _KEY_BASIC_INFORMATION {
    LARGE_INTEGER LastWriteTime;
    ULONG Type;
    ULONG NameLength;
    WCHAR Name[1];
} KEY_BASIC_INFORMATION, *PKEY_BASIC_INFORMATION;

```

这个名字并不是空结尾的字符串，因此你必须使用 **NameLength** 成员确定其长度，不要忘了长度的单位是字节。这个名字并不是完整的注册表路径，它就是这个子键的名称。这样更好，因为只要有子键名和其父键的句柄我们就可以打开这个子键了。

为了枚举键中的键值，可使用下面方法：

```

ULONG maxlen = fip->MaxValueNameLen + sizeof(KEY_VALUE_BASIC_INFORMATION);
PKEY_VALUE_BASIC_INFORMATION vip = (PKEY_VALUE_BASIC_INFORMATION)
ExAllocatePool(PagedPool, maxlen);
for (ULONG i = 0; i < fip->Values; ++i)
{
    ZwEnumerateValueKey(hkey, i, KeyValueBasicInformation, vip, maxlen, &size);
    <do something with vip->Name>
}
ExFreePool(vip);

```

基于已获得的 **KEY_FULL_INFORMATION** 结构中的 **.MaxValueNameLen** 成员，你可以为最大可能的 **KEY_VALUE_BASIC_INFORMATION** 结构分配空间。在循环中，你可以处理该键值的名称，这个名称存在于 **KEY_VALUE_BASIC_INFORMATION** 结构中：

```
typedef struct _KEY_VALUE_BASIC_INFORMATION {
    ULONG    TitleIndex;
    ULONG    Type;
    ULONG    NameLength;
    WCHAR    Name[1];
} KEY_VALUE_BASIC_INFORMATION, *PKEY_VALUE_BASIC_INFORMATION;
```

再一次，我们有了键值名和其父键的句柄，因此可以直接获取该键值。

访问文件

有时，WDM 驱动程序需要读写常规的磁盘文件。你可能需要向硬件下载一大段微代码，或者创建自己的全面的硬件操作记录。有一组 **ZwXxx** 函数可以做这些事情。

访问磁盘文件的第一步是调用 **ZwCreateFile** 打开一个文件句柄。DDK 中详细地描述了该函数。我在这里仅描述该函数的两个方面，即读/写一个已知名的文件。

打开已存在文件然后读

下面例子打开一个已存在的文件然后读该文件：

```
NTSTATUS status;
OBJECT_ATTRIBUTES oa;
IO_STATUS_BLOCK iostatus;
HANDLE hfile;           // the output from this process
PUNICODE_STRING pathname; // you've been given this

InitializeObjectAttributes(&oa, pathname, OBJ_CASE_INSENSITIVE, NULL, NULL);
status = ZwCreateFile(&hfile,
                     GENERIC_READ,
                     &oa,
                     &iostatus,
                     NULL,
                     0,
                     FILE_SHARE_READ,
                     FILE_OPEN,
                     FILE_SYNCHRONOUS_IO_NONALERT,
                     NULL,
                     0);
```

创建或重写文件

为了创建一个新文件，或打开一个已存在文件并截为 0 长度，用下面方式调用 **ZwCreateFile** 函数：

```
status = ZwCreateFile(&hfile,
                     GENERIC_WRITE,
                     &oa,
                     &iostatus,
                     NULL,
```

```
FILE_ATTRIBUTE_NORMAL,  
0,  
FILE_OVERWRITE_IF,  
FILE_SYNCHRONOUS_IO_NONALERT,  
NULL,  
0);
```

在上面这些例子中，我们建立了一个 **OBJECT_ATTRIBUTES** 结构，其目的是用它指向被打开文件的完整路径。我们指定了 **OBJ_CASE_INSENSITIVE** 属性，因为 Win32 文件系统模型对路径名的大小写不区分。然后我们调用 **ZwCreateFile** 打开文件句柄。

- 第一个参数(**&hfile**)，是 **HANDLE** 类型变量的地址，**ZwCreateFile** 将向该地址返回它创建的句柄。
- 第二个参数(**GENERIC_READ** 或 **GENERIC_WRITE**)，指定文件访问方式，读或写。
- 第三个参数(**&oa**)，是 **OBJECT_ATTRIBUTES** 结构的地址，该结构包含要打开的文件名。
- 第四个参数(**&iostatus**)，指向一个 **IO_STATUS_BLOCK** 结构，该结构接收 **ZwCreateFile** 操作的结果状态。
- 第五个参数(**NULL**)，是一个指针，指向一个 64 位整数，该数指定文件的初始分配大小。该参数仅关系到创建或重写文件操作，如果忽略它(如我在这里所做的)，那么文件长度将从 0 开始，并随着写入而增长。
- 第六个参数(**0** 或 **FILE_ATTRIBUTE_NORMAL**)，指定新创建文件的属性。
- 第七个参数(**FILE_SHARE_READ** 或 **0**)，指定文件的共享方式。如果仅为读数据而打开文件，则可以与其它线程同时读取该文件。如果为写数据而打开文件，你可能不希望其它线程访问该文件。
- 第八个参数(**FILE_OPEN** 或 **FILE_OVERWRITE_IF**)，表明当指定文件存在或不存在时应如何处理。在只读情况中，我指定 **FILE_OPEN**，因为我希望打开一个已存在的文件，当文件不存在时返回错误。在只写情况中，我指定 **FILE_OVERWRITE_IF**，因为我希望改写任何已存在的文件或创建一个新文件。
- 第九个参数(**FILE_SYNCHRONOUS_IO_NONALERT**)，指定控制打开操作和句柄使用的附加标志位。在这里，我指明要做同步 I/O 操作(即我希望读写函数在 I/O 操作完成后才返回，此时调用者处于等待状态)。
- 第十个参数(**NULL**)，一个指针，指向可选的扩展属性区。
- 第十一个参数(**0**)，扩展属性区的长度。

我们希望 **ZwCreateFile** 返回 **STATUS_SUCCESS** 状态并设置句柄变量。然后我们用 **ZwReadFile** 和 **ZwWriteFile** 函数执行读写操作，最后调用 **ZwClose** 函数关闭文件句柄。

```
ZwClose(hfile);
```

你可以执行同步或异步读写操作，这取决于你在 **ZwCreateFile** 中指定的标志。我在上面例子中指定了同步操作，读写函数在完成后才返回。如下例：

```
PVOID buffer;  
ULONG bufsize;  
status = ZwReadFile(hfile, NULL, NULL, NULL, &iostatus, buffer, bufsize, NULL, NULL);
```

-or-

```
status = ZwWriteFile(hfile, NULL, NULL, NULL, &iostatus, buffer, bufsize, NULL, NULL);
```

这些调用与用户模式中的非重叠 **ReadFile** 或 **WriteFile** 调用类似。当这些函数返回时，你可能对 **iostatus.Information** 域感兴趣，该域携带了操作所传输的字节数量。

如果你想把整个文件读入内存缓冲区，应该先调用 **ZwQueryInformationFile** 获得文件的总长度：

```
FILE_STANDARD_INFORMATION si;  
ZwQueryInformationFile(hfile, &iostatus, &si, sizeof(si), FileStandardInformation);  
ULONG length = si.EndOfFile.LowPart;
```

文件操作的时间选择

当驱动程序为响应 IRP_MN_START_DEVICE 请求而初始化设备时，可能需要读磁盘文件。由于设备初始化可能出现在系统初始化各种设备的不同阶段，所以使用普通的路径名如\?\C:\dir\file.ext 有时不能访问到文件。为了安全起见，你应该把数据文件放到系统根目录下的某个目录中，如\SystemRoot\dir\file.ext，而名称空间中的 SystemRoot 分支总是可访问的，因为操作系统在启动时也需要读磁盘文件。

浮点运算

有时候整数运算并不能满足要求，我们需要浮点运算。在 Intel 处理器上，浮点协处理器还可以执行 MMX 指令。在历史上，驱动程序在执行浮点运算上有两个问题。对于没有浮点协处理器的计算机，操作系统将用软件仿真一个，但是仿真的浮点协处理器会消耗很大的 CPU 处理能力，并且需要一个处理器异常来捕捉浮点指令。在内核模式中处理异常，尤其是在提升的 IRQL 级上，是困难的。另外，在有浮点协处理器的计算机上，由于 CPU 结构上的原因，当线程上下文切换时，需要一个耗时的操作来保存和恢复浮点协处理器的状态。所以，通常的做法是禁止在内核模式驱动程序中使用浮点运算。

Windows 2000 和 Windows 98 提供了一种方法来绕过这个问题。首先，有一个运行在低于或等于 DISPATCH_LEVEL 级上系统线程，该线程可以自由地使用浮点协处理器。另外，运行在低于或等于 DISPATCH_LEVEL 级上的任意线程背景的驱动程序可以调用两个系统函数，用这两个函数调用可以把浮点运算程序括起来：

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
KFLOATING_SAVE FloatSave;
NTSTATUS status = KeSaveFloatingPointState(&FloatSave);
if (NT_SUCCESS(status))
{
    ...
    KeRestoreFloatingPointState(&FloatSave);
}
```

这两个调用必须成对出现，它们为当前 CPU 保存和恢复浮点协处理器状态，即浮点协处理器的状态可以维持到任意多个指令周期。这个状态信息包括寄存器、控制字，等等。而在某些其它种类的 CPU 上，这两个调用不做什么事，因为这种 CPU 在线程切换时可以自动保存和恢复浮点协处理器的状态。出于这种原因，Microsoft 建议，除非必要，应避免在内核模式驱动程序中使用浮点运算。

调用 **KeSaveFloatingPointState** 时发生了什么，不同的 CPU 结构会有不同的答案。例如，在 Intel 处理器上，该函数通过执行 FSAVE 指令把整个浮点处理器状态保存起来。它既可以保存到当前线程的上下文块中，也可以保存到一块动态分配的内存区中。它还用一个不透明的 **FloatSave** 区记录关于被保存状态的中间信息，这可以使 **KeRestoreFloatingPointState** 正确地恢复浮点协处理器的状态。

如果系统没有浮点协处理器，KeSaveFloatingPointState 将失败并返回 STATUS_ILLEGAL_FLOAT_CONTEXT 代码。(顺便提一下，多 CPU 计算机的每个 CPU 必须同时拥有或同时没有协处理器) 因此你的驱动程序必须以其它方式实现浮点计算，或者干脆拒绝把驱动程序装入无浮点处理器的计算机上(由于 DriverEntry 的失败)。

调试技巧

我的驱动程序总是有 bug，你也许也象我这样不幸。当创建驱动程序时，你或者选择“checked”创建环境，或者选择“free”创建环境。在 checked 创建环境中，预处理符号 DBG 等于 1，而在 free 创建环境中它等于 0。因此，我们可以在驱动程序中提供一些调试代码，而这些代码仅在 checked 创建中才会产生作用：

```
#if DBG
    <extra debugging code>
#endif
```

一个最有用的调试技术就是让程序不时地打印出一些信息。**DbgPrint**就是一个用于这种目的的内核模式服务例程，你可以用它在任何调试器的输出窗口中显示格式化的信息。另一种查看**DbgPrint**输出的方法是使用**DbgView**工具，它可以从<http://www.sysinternals.com/>下载。除了在代码中直接调用**DbgPrint**外，你还可以使用**KdPrint**宏，它在**DBG**为真时才调用**DbgPrint**，如果**DBG**为假，它不生成任何代码：

```
KdPrint(("KeReadProgrammersMind failed with code %X\n", status));
```

必须为**KdPrint**使用两组括号，因为它就是这样定义的。第一个参数是带有%-escapes的串，在输出时，%-escapes出现的地方将被后面的值代替。第二个，第三个，等等的参数用于替代后来出现的多个%-escapes值。这个宏调用**DbgPrint**，而**DbgPrint**内部使用标准运行时间库例程_**vsnprintf**来格式化串。所以，你可以象在应用程序中使用_**vsnprintf**函数那样使用%-escape代码。

另外一个有用的调试技术需要依赖**ASSERT**宏：

```
ASSERT(1 + 1 == 2);
```

在**checked**版本的驱动程序中，**ASSERT**生成对布尔表达式求值的代码。如果表达式为假，**ASSERT**将使调试器中的程序停止执行，这样就可以看到出了什么问题。如果表达式为真，程序将正常执行。

如果你使用**Soft-ICE**调试器，那么**DDK**提供的**ASSERT**将不会向通常那样有用：第一，它需要调用**RtlAssert**，而该函数在**free**版本的操作系统中不做任何事。（你可以在**checked**版本的操作系统中测试你的驱动程序，但最好在**free**版本的操作系统中调试你的驱动程序）第二，如果它确实生成一个调试异常，也是在**RtlAssert**内部生成的而不是在你的代码中，这使你难于查看局部变量。为了克服这些问题，你可以用下面代码来替代**DDK**中的**ASSERT**宏：

```
#if DBG && defined(_X86_)
#undef ASSERT
#define ASSERT(e) if(!(e)){DbgPrint("Assertion failure in "\ 
    __FILE__ ", line %d: " #e "\n", __LINE__);\
    _asm int 1;\
}
#endif
```

记住一定要在这个**ASSERT**宏中发出**Soft-ICE**命令*i1here on*，以使调试器能停止程序执行。象这样替换**ASSERT**也有一个不利之处，如果在无调试器运行的情况下发生**ASSERT**失败，那么即使运行在**free**版本的操作系统上也会出现**bug check**。

Windows 98 兼容问题

用于访问磁盘文件的 `ZwXxx` 例程不能工作在零售版本的 Windows 98 上，原因有两个，一个是 Windows 98 的结构问题，另一个看起来象通常的 bug。

第一个问题，由于 Windows 98 必须按顺序初始化各种虚拟设备驱动程序(VxD)。而配置管理器(CONFIGMG.VXD)是在可安装文件系统管理器(IFSMGR.VXD)之前初始化的。在系统启动期间，WDM 驱动程序是在配置管理器(CONFIGMG)的初始化阶段收到 `IRP_MN_START_DEVICE` 请求的，而此时可安装文件系统管理器(IFSMGR)还没有初始化。所以不可能用 `ZwCreateFile` 函数或本章讨论的其它函数执行文件 I/O 操作。另外，也没有办法把 WDM 驱动程序的 `IRP_MN_START_DEVICE` 处理推迟到文件系统初始化之后。如果你没有象 Soft-ICE 那样的调试器，那么这个问题的征兆就是在初始化 CONFIGMG 时出现一个 Windows 保护错误的蓝屏。

第二个问题更差劲，`ZwReadFile`、`ZwWriteFile`，和 `ZwQueryInformationFile` 函数需要对其参数进行有效性检测。如果你提供了一个内核模式内存中的 `IO_STATUS_BLOCK`(只能这样，没有别的办法)，这些函数将探测一个不存在的虚拟地址，结果就发生了页故障，而结构化异常机制将捕捉到这个异常并返给你 `STATUS_ACCESS_VIOLATION` 状态代码，尽管你的做法没有任何毛病。这个问题在 1998 年 7 月的零售版 Windows 98 上也没有得到解决。

随书光盘上有一个 FILEIO 例子，它演示了一种绕过这些 Windows 98 问题的方法。执行文件操作时，FILEIO 在运行时决定是调用 `ZwXxx` 函数还是调用 VxD 服务函数。

第四章：同步

Windows 2000 可以运行在对称多处理器平台上。我并不想在这里严格描述 Windows NT 的多任务处理能力；如果你想了解这些内容，可参考 David Solomon 的《*Inside Windows NT, Second Edition* (Microsoft Press, 1998)》。作为驱动程序开发者我们需要了解的是：执行在某线程上下文中的代码在任何时刻都可能被系统夺去控制权。另外，只有在多处理器的计算机上才能真正实现多线程的并发执行。一般，我们需要做两个最差的假定：

- 操作系统可以在任何时间抢先任何例程并停留任何长的时间，所以我们不能保证自己的任务不被干扰或延迟。
- 即使我们能防止被抢先，但其它 CPU 上执行的代码也会干扰我们代码的执行，甚至一个程序的代码可以在两个不同线程的上下文中并发执行。

Windows NT 为解决一般的同步问题提供了两种方法，一个是中断请求优先级(IRQL)方案，另一个是在关键代码段周围声明和释放自旋锁。IRQL 可以避免在单 CPU 上的破坏性抢先，而自旋锁可以防止多 CPU 间的干扰。

- 一个原始的同步问题
- 中断请求级
- 自旋锁
- 内核同步对象
- 其它内核模式同步原语

一个原始的同步问题

举一个常见的例子，假设你的驱动程序有一个静态整型变量，该变量用于统计当前未完成的 I/O 请求数：

```
static LONG lActiveRequests;
```

再假设你接收到一个请求后就把该变量增 1，完成一个请求后就把该变量减 1：

```
NTSTATUS DispatchPnp(PDEVICE_OBJECT fdo, PIRP Irp)
{
    ++lActiveRequests;
    ... // process PNP request
    --lActiveRequests;
}
```

我想你已经认识到了，象这样的计数器不应该是静态变量：它应该是设备扩展的一个成员，这样每个设备对象就可以有自己的计数器。我们暂且假定你的驱动程序仅管理一个设备。为了使这个例子更有意义，最后假定你的驱动程序中有这样一个函数，该函数在设备对象被删除时调用。你应该一直推迟删除操作，直到所有等待的请求都被完成，因此你应该加入计数器测试代码：

```
NTSTATUS HandleRemoveDevice(PDEVICE_OBJECT fdo, PIRP Irp)
{
    if (lActiveRequests)
        <wait for all requests to complete>
    IoDeleteDevice(fdo);
}
```

这个例子描述了一个真实的问题，我将在第六章中讨论 PnP 请求时再解决这个问题。I/O 管理器能在驱动程序处理请求时删除我们的设备对象，所以我们需要某种计数器来保护设备对象。我将在第六章用 **IoAcquireRemoveLock** 函数和其它一些相关函数来解决这个问题。

有一个可怕的同步问题潜伏在上面代码片段中，但只有当你看到增 1 和减 1 操作的内部代码后才能看出这个问题。在 x86 处理器上，编译器为这两个操作生成下面汇编指令：

```
; ++lActiveRequests;
    mov eax, lActiveRequests
    add eax, 1
    mov lActiveRequests, eax

...
; --lActiveRequests;
    mov eax, lActiveRequests
    sub eax, 1
    mov lActiveRequests, eax
```

为了查出这个同步问题，让我们先考虑单 CPU 的情况。假定有两个线程在大致相同的时间内都要进入 **DispatchPnp** 函数。我们知道它们并不是真正地并发执行，因为我们仅有一个 CPU。再假定有一个线程刚好执行到函数的尾部并且就在另一个线程要抢先它之前把 **IActiveRequests** 的当前内容读入了 EAX 寄存器。假设那一时刻 **IActiveRequests** 的值等于 2。作为线程切换的一部分，操作系统把 EAX 寄存器的内容(值为 2)保存到该线程的上下文中。

现在，假设另一个线程正执行到 DispatchPnp 函数的头部，它把 `IActiveRequests` 的值从 2 增为 3(因为上个线程并没有修改该变量)。如果此时这个线程又被第一个线程抢先，操作系统将恢复第一个线程的上下文，这包括 `EAX` 寄存器中的值 2。然后，第一个线程继续执行，它把 `EAX` 的值减 1，最后把结果存入 `IActiveRequests`。此时 `IActiveRequests` 的值为 1，但这是不正确的。这造成的结果是，我们可能过早地删除我们的设备对象，因为确实丢失了一个 I/O 请求记录。

在 x86 计算机上可以很容易地解决这个特殊问题，我们仅需要把 `load/add/store` 和 `load/subtract/store` 指令序列替换为原子指令：

```
; ++lActiveRequests;  
inc lActiveRequests  
...  
; --lActiveRequests;  
dec lActiveRequests
```

在 Intel x86 CPU 上，`INC` 和 `DEC` 指令不能被中断，所以绝不会出现线程在更新计数器时被抢先的情况。虽然如此，但这两个指令在多处理器环境中仍然是不安全的，因为这两个指令都是由几条微代码(CISC 的特点)实现的。我们可以使用 x86 处理器中的 `LOCK` 指令前缀避免这个多 CPU 问题：

```
; ++lActiveRequests;  
lock inc lActiveRequests  
...  
; --lActiveRequests;  
lock dec lActiveRequests
```

`LOCK` 指令前缀可以使当前执行多微码指令的 CPU 锁定总线，从而保证数据访问的完整性。

不幸的是，并不是所有的同步问题都可以这样容易地解决。我举的这个例子主要是为了阐述同步问题的来源：如果一个线程正在改变某状态变量时被另一个线程抢先；那么两个有矛盾的改变操作就可能被同时执行。在本章的剩下内容中，我将讲述如何使用 `IRQL` 优先级方案来避免抢先；如何使用自旋锁来防止并发执行。

中断请求级

Windows NT 为每个硬件中断和少数软件事件赋予了一个优先级，即中断请求级(interrupt request level - IRQL)。IRQL 为单 CPU 上的活动提供了同步方法，它基于下面规则：

一旦某 CPU 执行在高于 **PASSIVE_LEVEL** 的 IRQL 上时，该 CPU 上的活动仅能被拥有更高 IRQL 的活动抢先。

图 4-1 显示了 x86 平台上的 IRQL 值范围。(通常，这个 IRQL 数值要取决于你所面对的平台) 用户模式程序执行在 **PASSIVE_LEVEL** 上，可以被任何执行在高于该 IRQL 上的活动抢先。许多设备驱动程序例程也执行在 **PASSIVE_LEVEL** 上。第二章中讨论的 **DriverEntry** 和 **AddDevice** 例程就属于这类，大部分 IRP 派遣例程也属于这类。

某些公共驱动程序例程执行在 **DISPATCH_LEVEL** 上，而 **DISPATCH_LEVEL** 级要比 **PASSIVE_LEVEL** 级高。这些公共例程包括 **StartIo** 例程，**DPC**(推迟过程调用)例程，和其它一些例程。这些例程的共同特点是，它们都需要访问设备对象和设备扩展中的某些域，它们都不受派遣例程的干扰或互相干扰。当任何一个这样的例程运行时，上面陈述的规则可以保证它们不被任何驱动程序的派遣例程抢先，因为派遣例程本身执行在更低级的 IRQL 上。另外，它们也不会被同类例程抢先，因为那些例程运行的 IRQL 与它们自己的相同。只有拥有更高 IRQL 的活动才能抢先它们。

注意

派遣例程(Dispatch routine)和 **DISPATCH_LEVEL** 级名称类似。之所以称做派遣例程是因为 I/O 管理器向这些函数派遣 I/O 请求。而存在派遣级(**DISPATCH_LEVEL**)这个名称是因为内核线程派遣器运行在这个 IRQL 上，它决定下一次该执行哪个线程。(现在，线程调度程序通常运行在 **SYNCH_LEVEL** 级上)

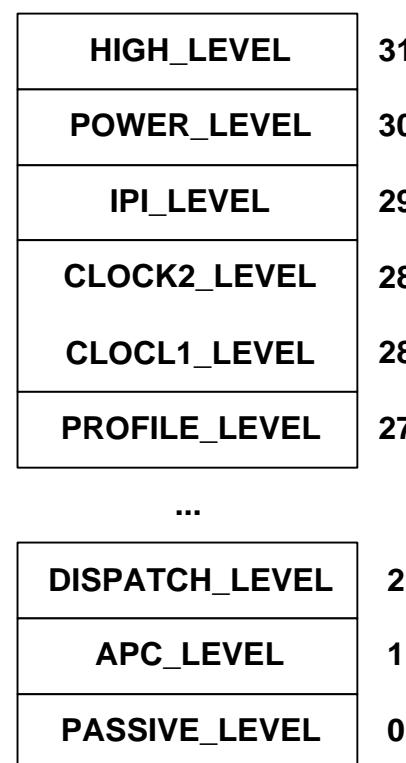


图 4-1. 中断请求级

在 **DISPATCH_LEVEL** 级和 **PROFILE_LEVEL** 级之间是各种硬件中断级。通常，每个有中断能力的设备都有一个 IRQL，它定义了该设备的中断优先级别。WDM 驱动程序只有在收到一个副功能码为 **IRP_MN_START_DEVICE** 的 **IRP_MJ_PNP** 请求后，才能确定其设备的 IRQL。设备的配置信息作为参数传递给该请求，而设备的 IRQL 就包含在这个配置信息中。我们通常把设备的中断级称为设备 IRQL，或 DIRQL。

其它 IRQL 级的含义有时需要依靠具体的 CPU 结构。这些 IRQL 通常仅被 Windows NT 内核内部使用，因此它们的含义与设备驱动程序的编写不是特别密切相关。例如，我将要在本章后面详细讨论的 APC_LEVEL，当系统在该级上为某线程调度 APC(异步过程调用)例程时不会被同一 CPU 上的其它线程所干扰。在 HIGH_LEVEL 级上系统可以执行一些特殊操作，如系统休眠前的内存快照、处理 bug check、处理假中断，等等。

IRQL 的变化

为了演示 IRQL 的重要性，参见图 4-2，该图显示了发生在单 CPU 上的一系列事件。在时间序列的开始处，CPU 执行在 PASSIVE_LEVEL 级上。在 t1 时刻，一个中断到达，它的服务例程执行在 DIRQL1 上，该级是在 DISPATCH_LEVEL 和 PROFILE_LEVEL 之间的某个 DIRQL。在 t2 时刻，另一个中断到达，它的服务例程执行在 DIRQL2 上，比 DIRQL1 低一级。我们讨论过抢先规则，所以 CPU 将继续服务于第一个中断。当第一个中断服务例程在 t3 时刻完成时，该中断服务程序可能会请求一个 DPC。而 DPC 例程是执行在 DISPATCH_LEVEL 上。所以当前存在的未执行的最高优先级的活动就是第二个中断的服务例程，所以系统接着执行第二个中断的服务例程。这个例程在 t4 时刻结束，假设这之后再没有其它中断发生，CPU 将降到 DISPATCH_LEVEL 级上执行第一个中断的 DPC 例程。当 DPC 例程在 t5 时刻完成后，IRQL 又落回到原来的 PASSIVE_LEVEL 级。

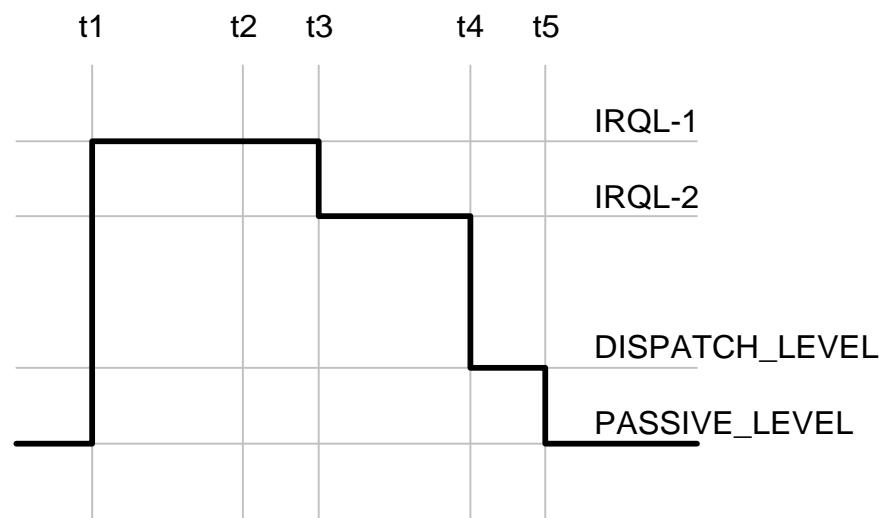


图 4-2. 变化中的中断优先级

基本同步规则

遵循下面规则，你可以利用 IRQL 的同步效果：

所有对共享数据的访问都应该在同一(提升的)IRQL 上进行。

换句话说，不论何时何地，如果你的代码访问的数据对象被其它代码共享，那么你应该使你的代码执行在高于 PASSIVE_LEVEL 的级上。一旦越过 PASSIVE_LEVEL 级，操作系统将不允许同 IRQL 的活动相互抢先，从而防止了潜在的冲突。然而这个规则不足以保护多处理器机器上的数据，在多处理器机器中你还需要另外的防护措施——自旋锁(spin lock)。如果你仅关心单 CPU 上的操作，那么使用 IRQL 就可以解决所有同步问题。但事实上，所有 WDM 驱动程序都必须设计成能够运行在多处理器的系统上。

IRQL 与线程优先级

线程优先级是与 IRQL 非常不同的概念。线程优先级控制着线程调度器的调度动作，决定何时抢先运行线程以及下一次运行什么线程。然而，当 IRQL 级高于或等于 DISPATCH_LEVEL 级时线程切换停止，无论当前活动的是什么线程都将保持活动状态直到 IRQL 降到 DISPATCH_LEVEL 级之下。而此时的“优先级”仅指 IRQL 本身，由它控制到底哪个活动该执行，而不是该切换到哪个线程的上下文。

IRQL和分页

执行在提升的 IRQL 级上的一个后果是，系统将不能处理页故障(系统在 APC 级处理页故障)。这意味着：

执行在高于或等于 **DISPATCH_LEVEL** 级上的代码绝对不能造成页故障。

这也意味着执行在高于或等于 **DISPATCH_LEVEL** 级上的代码必须存在于非分页内存中。此外，所有这些代码要访问的数据也必须存在于非分页内存中。最后，随着 IRQL 的提升，你能使用的内核模式支持例程将会越来越少。

DDK 文档中明确指出支持例程的 IRQL 限定。例如，**KeWaitForSingleObject** 例程有两个限定：

- 调用者必须运行在低于或等于 **DISPATCH_LEVEL** 级上。
- 如果调用中指定了非 0 的超时，那么调用者必须严格地运行在低于 **DISPATCH_LEVEL** 的 IRQL 上。

上面这两行想要说明的是：如果 **KeWaitForSingleObject** 真的被阻塞了指定长的时间(你指定的非 0 超时)，那么你必定运行在低于 **DISPATCH_LEVEL** 的 IRQL 上，因为只有在这样的 IRQL 上线程阻塞才是允许的。如果你所做的一切就是为了检测事件是否进入信号态，则可以执行在 **DISPATCH_LEVEL** 级上。但你不能在 ISR 或其它运行在高于 **DISPATCH_LEVEL** 级上的例程中调用 **KeWaitForSingleObject** 例程。

IRQL的隐含控制

在大部分时间里，系统都是在正确的 IRQL 上调用驱动程序中的例程。虽然我们还没有详细地讨论过这些例程，但我希望举一个例子来表达这句话的含义。你首先遇到的 I/O 请求就是 I/O 管理器调用你的某个派遣例程来处理一个 IRP。这个调用发生在 **PASSIVE_LEVEL** 级上，因为你需要阻塞调用者线程，还需要调用其它支持例程。当然，你不能在更高的 IRQL 级上阻塞一个线程，而 **PASSIVE_LEVEL** 也是唯一能让你无限制地调用任何支持例程的 IRQL 级。

如果你的派遣例程通过调用 **IoStartPacket** 来排队 IRP，那么你第一个遇到的请求将发生在 I/O 管理器调用你的 **StartIo** 例程时。这个调用发生在 **DISPATCH_LEVEL** 级，因为系统需要在没有其它例程(这些例程能在队列中插入或删除 IRP)干扰的情况下访问 I/O 队列。回想一下前面提到的规则：所有对共享数据的访问都应该在同一(提升的)IRQL 级上进行。因为每个能访问 IRP 队列的例程都执行在 **DISPATCH_LEVEL** 级上，所以任何例程在操作队列期间都不可能被打断(仅指在单 CPU 系统)。

之后，设备可能生成一个中断，而该中断的服务例程(ISR)将在 **DIRQL** 级上被调用。设备上的某些寄存器也许不能被安全地共享。但是，如果你仅在 **DIRQL** 上访问那些寄存器，可以保证在单 CPU 计算机上没人能妨碍你的 ISR 执行。如果驱动程序的其它代码需要访问这些关键的硬件寄存器，你应该让这些代码仅执行在 **DIRQL** 级上。**KeSynchronizeExecution** 服务函数可以帮助你强制执行这个规则，我将在第七章的“与中断处理连接”段中讨论这个函数。

再往后，你应该安排一个 DPC 调用。DPC 例程执行在 **DISPATCH_LEVEL** 级上，它们需要访问你的 IRP 队列，并取出队列中的下一个请求，然后把这个请求发送给 **StartIo** 例程。你可以调用 **IoStartNextPacket** 服务函数从队列中提取下一个请求，但必须在 **DISPATCH_LEVEL** 级上调用。该函数在返回前将调用你的 **StartIo** 例程。注意，这里的 IRQL 吻合得相当巧妙：队列访问，调用 **IoStartNextPacket**，和调用 **StartIo** 都需要发生在 **DISPATCH_LEVEL** 级上，并且系统也是在这个 IRQL 级上调用 DPC 例程的。

尽管明确地控制 IRQL 也是可能的，但几乎没有理由这样做，因为你需要的 IRQL 和系统调用你时使用的 IRQL 总是相应的。所以不必不时地提高 IRQL，例程希望的 IRQL 和系统使用的 IRQL 几乎总是正确对应的。

IRQL的明确控制

如果必要，你还可以在当前处理器上临时提升 IRQL，然后再降回到原来的 IRQL，使用 **KeRaiseIrql** 和 **KeLowerIrql** 函数。下面代码运行在 **PASSIVE_LEVEL** 级上：

```
KIRQL oldirql;                                <--1
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);    <--2
KeRaiseIrql(DISPATCH_LEVEL, &oldirql);             <--3
...
KeLowerIrql(oldirql);                          <--4
```

1. KIRQL 定义了用于保存 IRQL 值的数据类型。我们需要一个变量来保存当前 IRQL。
2. 这个 ASSERT 断定了调用 KeRaiseIrql 的必要条件：新 IRQL 必须大于或等于当前 IRQL。如果这个关系不成立，KeRaiseIrql 将导致 bug check。(即用死亡蓝屏报告一个致命错误)
3. KeRaiseIrql 把当前的 IRQL 提升到第一个参数指定的 IRQL 级上。它同时还把当前的 IRQL 值保存到第二个参数指定的变量中。在这个例子中，我们把 IRQL 提升到 DISPATCH_LEVEL 级，并把原来的 IRQL 级保存到 oldirql 变量中。
4. 执行完任何需要在提升的 IRQL 上执行的代码后，我们调用 KeLowerIrql 把 IRQL 降低到调用 KeRaiseIrql 时的级别。

DDK 文档中提到，你必须用与你最近的 KeRaiseIrql 调用所返回的值调用 KeLowerIrql。这在大的方面是对的，因为你提升了 IRQL 就必须再降低它。然而，由于你调用的代码或者调用你的代码所做的各种假设会使后面的决定变得不正确。所以，文档中的这句话从严格意义上讲是不正确的。应用到 KeLowerIrql 函数的唯一的规则就是新 IRQL 必须低于或等于当前 IRQL。

当系统调用你的驱动程序例程时，你降低了 IRQL(系统调用你的例程时使用的 IRQL，或你的例程希望执行的 IRQL)，这是一个错误，而且是严重错误，尽管你在例程返回前又提升了 IRQL。这种打破同步的结果是，某些活动可以抢先你的例程，并能访问你的调用者认为不能被共享的数据对象。

有一个函数专用于把 IRQL 提升到 DISPATCH_LEVEL 级：

```
KIRQL oldirql = KeRaiseIrqlToDpcLevel();
...
KeLowerIrql(oldirql)
```

注意：该函数仅在 NTDDK.H 中声明，WDM.H 中并没有声明该函数，因此 WDM 驱动程序不应该使用该函数。

自旋锁

IRQL 概念仅能解决单 **CPU** 上的同步问题，在多处理器平台上，它不能保证你的代码不被运行在其它处理器上的代码所干扰。一个称为自旋锁(**spin lock**)的原始对象可以解决这个问题。为了获得一个自旋锁，在某 **CPU** 上运行的代码需先执行一个原子操作，该操作测试并设置(**test-and-set**)某个内存变量，由于它是原子操作，所以在该操作完成之前其它 **CPU** 不可能访问这个内存变量。如果测试结果表明锁已经空闲，则程序获得这个自旋锁并继续执行。如果测试结果表明锁仍被占用，程序将在一个小的循环内重复这个“测试并设置(**test-and-set**)”操作，即开始“自旋”。最后，锁的所有者通过重置该变量释放这个自旋锁，于是，某个等待的 **test-and-set** 操作向其调用者报告锁已释放。

关于自旋锁有两个明显的事实。第一，如果一个已经拥有某个自旋锁的 **CPU** 想第二次获得这个自旋锁，则该 **CPU** 将死锁(**deadlock**)。自旋锁没有与其关联的“使用计数器”或“所有者标识”；锁或者被占用或者空闲。如果你在锁被占用时获取它，你将等待到该锁被释放。如果碰巧你的 **CPU** 已经拥有了该锁，那么用于释放锁的代码将得不到运行，因为你使 **CPU** 永远处于“测试并设置”某个内存变量的自旋状态。

关于自旋锁的另一个事实是，**CPU** 在等待自旋锁时不做任何有用的工作，仅仅是等待。所以，为了避免影响性能，你应该在拥有自旋锁时做尽量少的操作，因为此时某个 **CPU** 可能正在等待这个自旋锁。

关于自旋锁还存在着一个不太明显但很重要的事实：你仅能在低于或等于 **DISPATCH_LEVEL** 级上请求自旋锁，在你拥有自旋锁期间，内核将把你的代码提升到 **DISPATCH_LEVEL** 级上运行。在内部，内核能在高于 **DISPATCH_LEVEL** 的级上获取自旋锁，但你和我都做不到这一点。

使用自旋锁

为了明确地使用一个自旋锁，首先要在非分页内存中为一个 **KSPIN_LOCK** 对象分配存储。然后调用 **KeInitializeSpinLock** 初始化这个对象。接着，当代码运行在低于或等于 **DISPATCH_LEVEL** 级上时获取这个锁，并执行需要保护的代码，最后释放自旋锁。例如，假设你的设备扩展中有一个名为 **QLock** 的自旋锁，你用它来保护你专用 **IRP** 队列的访问。你应该在 **AddDevice** 函数中初始化这个锁：

```
typedef struct _DEVICE_EXTENSION {
    ...
    KSPIN_LOCK QLock;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

...
NTSTATUS AddDevice(...)
{
    ...
    PDEVICE_EXTENSION pdx = ...;
    KeInitializeSpinLock(&pdx->QLock);
    ...
}
```

在驱动程序的其它地方，假定就在某种 **IRP** 的派遣函数中，你在某些必须的队列操作代码周围获取了(并很快释放)该自旋锁。注意这个函数必须存在于非分页内存中，因为在某个时期它会执行在提升的 **IRQL** 上。

```
NTSTATUS DispatchSomething(...)
{
    KIRQL oldirql;
    PDEVICE_EXTENSION pdx = ...;
    KeAcquireSpinLock(&pdx->QLock, &oldirql);           <--1
    ...
}
```

```
    KeReleaseSpinLock(&pdx->QLock, oldirql);                                <--2
}
```

1. 当 **KeAcquireSpinLock** 获取自旋锁时，它也把 IRQL 提升到 DISPATCH_LEVEL 级上。
2. 当 **KeReleaseSpinLock** 释放自旋锁时，它也把 IRQL 降低到原来的 IRQL 级上。

如果你知道代码已经处在 DISPATCH_LEVEL 级上，你可以调用两个专用函数来获取自旋锁。这个技术适合于 DPC、StartIo，和其它执行在 DISPATCH_LEVEL 级上的驱动程序例程：

```
KeAcquireSpinLockAtDpcLevel(&pdx->QLock);
...
KeReleaseSpinLockFromDpcLevel(&pdx->QLock);
```

内核同步对象

Windows NT 提供了五种内核同步对象(Kernel Dispatcher Object)，你可以用它们控制非任意线程(普通线程)的流程。表 4-1 列出了这些内核同步对象的类型及它们的用途。在任何时刻，任何对象都处于两种状态中的一种：信号态或非信号态。有时，当代码运行在某个线程的上下文中时，它可以阻塞这个线程的执行，调用 **KeWaitForSingleObject** 或 **KeWaitForMultipleObjects** 函数可以使代码(以及背景线程)在一个或多个同步对象上等待，等待它们进入信号态。内核为初始化和控制这些对象的状态提供了例程。

表 4-1. 内核同步对象

对象	数据类型	描述
Event(事件)	KEVENT	阻塞一个线程直到其它线程检测到某事件发生
Semaphore(信号灯)	KSEMAPHORE	与事件对象相似，但可以满足任意数量的等待
Mutex(互斥)	KMUTEX	执行到关键代码段时，禁止其它线程执行该代码段
Timer(定时器)	KTIMER	推迟线程执行一段时期
Thread(线程)	KTHREAD	阻塞一个线程直到另一个线程结束

在下几段中，我将描述如何使用内核同步对象。我将从何时可以调用等待原语阻塞线程开始讲起，然后讨论用于每种对象的支持例程。最后讨论与线程警惕(thread alert)和提交 APC(异步过程调用)相关的概念。

何时阻塞和怎样阻塞一个线程

为了理解 WDM 驱动程序何时以及如何利用内核同步对象阻塞一个线程，你必须先对线程有一些基本了解。通常，如果在线程执行时发生了软件或硬件中断，那么在内核处理中断期间，该线程仍然是“当前”线程。而内核模式代码执行时所在的上下文环境就是指这个“当前”线程的上下文。为了响应各种中断，Windows NT 调度器可能会切换线程，这样，另一个线程将成为新的“当前”线程。

术语“任意线程上下文(arbitrary thread context)”和“非任意线程上下文(nonarbitrary thread context)”用于精确描述驱动程序例程执行时所处的上下文种类。如果我们知道程序正处于初始化 I/O 请求线程的上下文中，则该上下文不是任意上下文。然而，在大部分时间里，WDM 驱动程序无法知道这个事实，因为控制哪个线程应该激活的机会通常都是在中断发生时。当应用程序发出 I/O 请求时，将产生一个从用户模式到内核模式的转换，而创建并发送该 IRP 的 I/O 管理器例程将继续运行在非任意线程的上下文中。我们用术语“最高级驱动程序”来描述第一个收到该 IRP 的驱动程序。

通常，只有给定设备的最高级驱动程序才能确切地知道它执行在一个非任意线程的上下文中。这是因为驱动程序派遣例程通常把请求放入队列后立即返回调用者。之后通过回调函数，请求被提出队列并下传到低级驱动程序。一旦派遣例程挂起某个请求，所有对该请求的后期处理必须发生在任意线程上下文中。

解释完线程上下文后，我们可以陈述出关于线程阻塞的简单规则：

当我们处理某个请求时，仅能阻塞产生该请求的线程。

通常，仅有设备的最高级驱动程序才能应用这个规则。但有一个重要的例外，IRP_MN_START_DEVICE 请求，所有驱动程序都以同步方式处理这个请求。即驱动程序不排队或挂起该类请求。当你收到这种请求时，你可以直接从堆栈中找到请求的发起者。正如我在第六章中讲到的，处理这种请求时你必须阻塞那个线程。

下面规则表明在提升的 IRQL 级上不可能发生线程切换：

执行在高于或等于 DISPATCH_LEVEL 级上的代码不能阻塞线程。

这个规则表明你只能在 `DriverEntry` 函数、`AddDevice` 函数，或驱动程序的派遣函数中阻塞当前线程。因为这些函数都执行在 `PASSIVE_LEVEL` 级上。没有必要在 `DriverEntry` 或 `AddDevice` 函数中阻塞当前线程，因为这些函数的工作仅仅是初始化一些数据结构。

在单同步对象上等待

你可以按下面方法调用 `KeWaitForSingleObject` 函数：

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LARGE_INTEGER timeout;
NTSTATUS status = KeWaitForSingleObject(object, WaitReason, WaitMode, Alertable, &timeout);
```

`ASSERT` 语句指出必须在低于或等于 `DISPATCH_LEVEL` 级上调用该例程。

在这个调用中，`object` 指向你要等待的对象。注意该参数的类型是 `PVOID`，它应该指向一个表 4-1 中列出的同步对象。该对象必须在非分页内存中，例如，在设备扩展中或其它从非分页内存池中分配的数据区。在大部分情况下，执行堆栈可以被认为是非分页的。

`WaitReason` 是一个纯粹建议性的值，它是 `KWAIT_REASON` 枚举类型。实际上，除非你指定了 `WrQueue` 参数，否则任何内核代码都不关心此值。线程阻塞的原因被保存到一个不透明的数据结构中，如果你了解这个数据结构，那么在调试某种死锁时，你也许会从这个原因代码中获得一些线索。通常，驱动程序应把该参数指定为 `Executive`，代表无原因。

`WaitMode` 是 `MODE` 枚举类型，该枚举类型仅有两个值：`KernelMode` 和 `UserMode`。

`Alertable` 是一个布尔类型的值。它不同于 `WaitReason`，这个参数以另一种方式影响系统行为，它决定等待是否可以提前终止以提交一个 `APC`。如果等待发生在用户模式中，那么内存管理器就可以把线程的内核模式堆栈换出。如果驱动程序以自动变量(在堆栈中)形式创建事件对象，并且某个线程又在提升的 `IRQL` 级上调用了 `KeSetEvent`，而此时该事件对象刚好又被换出内存，结果将产生一个 `bug check`。所以我们应该总把 `alertable` 参数指定为 `FALSE`，即在内核模式中等待。

最后一个参数 `&timeout` 是一个 64 位超时值的地址，单位为 100 纳秒。正数的超时表示一个从 1601 年 1 月 1 日起的绝对时间。调用 `KeQuerySystemTime` 函数可以获得当前系统时间。负数代表相对于当前时间的时间间隔。如果你指定了绝对超时，那么系统时钟的改变也将影响到你的超时时间。如果系统时间越过你指定的绝对时间，那么永远都不会超时。相反，如果你指定相对超时，那么你经过的超时时间将不受系统时钟改变的影响。

为什么是 1601 年 1 月 1 日

许多年以前，当我第一次学习 Win32 API 时，我曾迷惑为什么选择 1601 年 1 月 1 日作为 Windows NT 的时间起点。在我写了一组时间转换函数后我明白了这个问题的原因。每个人都知道可被 4 整除的年份是闰年。许多人也知道世纪年(如 1900 年)应例外，虽然这些年份都能被 4 整除但它们不是闰年。少数人还知道能被 400 整除的年份(如 1600 和 2000)是例外中的例外，它们也是闰年。而 1601 年 1 月 1 日正好是一个 400 年周期的开始。如果把它作为时间信息的起点，那么把 NT 时间信息转换为常规日期表达(或相反)就不用做任何跳跃操作。

指定 0 超时将使 `KeWaitForSingleObject` 函数立即返回，返回的状态代码指出对象是否处于信号态。如果你的代码执行在 `DISPATCH_LEVEL` 级上，则必须指定 0 超时，因为在这个 `IRQL` 上不允许阻塞。每个内核同步对象都提供一组 `KeReadStateXxx` 服务函数，使用这些函数可以直接获得对象的状态。然而，取对象状态与 0 超时等待不完全等价：当 `KeWaitForSingleObject` 发现等待被满足后，它执行特殊对象要求的附加动作。相比之下，取对象状态不执行任何附加动作，即使对象已经处于信号态。

超时参数也可以指定为 `NULL` 指针，这代表无限期等待。

该函数的返回值指出几种可能的结果。`STATUS_SUCCESS` 结果是你所希望的，表示等待被满足。即在你调用 `KeWaitForSingleObject` 时，对象或者已经进入信号态，或者后来进入信号态。如果等待以第二种情况满足，则

有必要在同步对象上执行附加动作。当然，这个附加动作还要参考对象的类型，我将在后面讨论具体对象类型时再解释这一点。(例如，一个同步类型的事件在你的等待满足后需要重置该事件)

返回值 **STATUS_TIMEOUT** 指出在指定的超时期限内对象未进入信号态。如果指定 0 超时，则函数将立即返回。返回代码为 **STATUS_TIMEOUT**，代表对象处于非信号态，返回代码为 **STATUS_SUCCESS**，代表对象处于信号态。如果指定 **NULL** 超时，则不可能有返回值。

其它两个返回值 **STATUS_ALERTED** 和 **STATUS_USER_APIC** 表示等待提前终止，对象未进入信号态。原因是线程接收到一个警惕(alert)或一个用户模式的 APC。

在多同步对象上等待

KeWaitForMultipleObjects 函数用于同时等待一个或多个同步对象。该函数调用方式如下：

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LARGE_INTEGER timeout;
NTSTATUS status = KeWaitForMultipleObjects(count,
                                         objects,
                                         WaitType,
                                         WaitReason,
                                         WaitMode,
                                         Alertable,
                                         &timeout,
                                         waitblocks);
```

在这里，**objects** 指向一个指针数组，每个数组元素指向一个同步对象，**count** 是数组中指针的个数。**count** 必须小于或等于 **MAXIMUM_WAIT_OBJECTS** 值(当前为 64)。这个数组和它所指向的所有对象都必须在非分页内存中。**WaitType** 是枚举类型，其值可以为 **WaitAll** 或 **WaitAny**，它指出你是等到所有对象都进入信号态，还是只要有一个对象进入信号态就可以。

waitblocks 参数指向一个 **KWAIT_BLOCK** 结构数组，内核用这个结构数组管理等待操作。你不需要初始化这些结构，内核仅需要知道这个结构数组在哪里，内核用它来记录每个对象在等待中的状态。如果你仅需要等待小数量的对象(不超过 **THREAD_WAIT_OBJECTS**，该值当前为 3)，你可以把该参数指定为 **NULL**。如果该参数为 **NULL**，**KeWaitForMultipleObjects** 将使用线程对象中预分配的等待块数组。如果你等待的对象数超过 **THREAD_WAIT_OBJECTS**，你必须提供一块长度至少为 **count * sizeof(KWAIT_BLOCK)** 的非分页内存。

其余参数与 **KeWaitForSingleObject** 中的对应参数作用相同，而且大部分返回码也有相同的含义。

如果你指定了 **WaitAll**，则返回值 **STATUS_SUCCESS** 表示等待的所有对象都进入了信号态。如果你指定了 **WaitAny**，则返回值在数值上等于进入信号态的对象在 **objects** 数组中的索引。如果碰巧有多个对象进入了信号态，则该值仅代表其中的一个，可能是第一个也可能是其它。你可以认为该值等于 **STATUS_WAIT_0** 加上数组索引。你可以先用 **NT_SUCCESS** 测试返回码，然后再从其中提取数组索引：

```
NTSTATUS status = KeWaitForMultipleObjects(...);
if (NT_SUCCESS(status))
{
    ULONG iSignalled = (ULONG) status - (ULONG) STATUS_WAIT_0;
    ...
}
```

如果 **KeWaitForMultipleObjects** 返回成功代码，它也将执行等待被满足的那个对象的附加动作。如果多个对象同时进入信号态而你指定的 **WaitType** 参数为 **WaitAny**，那么该函数仅执行返回值指定对象的附加动作。

内核事件

表 4-2 列出了用于处理内核事件的服务函数。为了初始化一个事件对象，我们首先应该为其分配非分页存储，然后调用 **KeInitializeEvent**:

```
ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);
KeInitializeEvent(event, EventType, initialstate);
```

event 是事件对象的地址。**EventType** 是一个枚举值，可以为 **NotificationEvent** 或 **SynchronizationEvent**。通知事件(notification event)有这样的特性，当它进入信号态后，它将一直处于信号态直到你明确地把它重置为非信号态。此外，当通知事件进入信号态后，所有在该事件上等待的线程都被释放。这与用户模式中的手动重置事件相似。而对于同步事件(synchronization event)，只要有一个线程被释放，该事件就被重置为非信号态。这又与用户模式中的自动重置事件相同。而 **KeWaitXxx** 函数在同步事件对象上执行的附加动作就是把它重置为非信号态。最后的参数 **initialstate** 是布尔量，为 **TRUE** 表示事件的初始状态为信号态，为 **FALSE** 表示事件的初始状态为非信号态。

表 4-2. 用于内核事件对象的服务函数

服务函数	描述
KeClearEvent	把事件设置为非信号态，不报告以前的状态
KeInitializeEvent	初始化事件对象
KeReadStateEvent	取事件的当前状态
KeResetEvent	把事件设置为非信号态，返回以前的状态
KeSetEvent	把事件设置为信号态，返回以前的状态

注意

在这些关于同步原语的段中，我还要再谈论一下 DDK 文档中对 IRQL 的使用限定。在当前发行的 Windows 2000 中，DDK 有时比 OS 实际要求的有更多的限制。例如，**KeClearEvent** 可以在任何 IRQL 上调用，但 DDK 却要求调用者必须在低于或等于 **DISPATCH_LEVEL** 级上调用。**KeInitializeEvent** 也可以在任何 IRQL 上调用，但 DDK 要求仅在 **PASSIVE_LEVEL** 级上调用该函数。然而，你应该尊重 DDK 中的描述，也许某一天 Microsoft 会利用文档中的这些限制。

调用 **KeSetEvent** 函数可以把事件置为信号态:

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LONG wassignalled = KeSetEvent(event, boost, wait);
```

在上面代码中，**ASSERT** 语句强制你必须在低于或等于 **DISPATCH_LEVEL** 级上调用该函数。**event** 参数指向一个事件对象，**boost** 值用于提升等待线程的优先级。**wait** 参数的解释见文字框“**KeSetEvent** 的第三个参数”，WDM 驱动程序几乎从不把 **wait** 参数指定为 **TRUE**。如果该事件已经处于信号态，则该函数返回非 0 值。如果该事件处于非信号态，则该函数返回 0。

多任务调度器需要人为地提升等待 I/O 操作或同步对象的线程的优先级，以避免饿死长时间等待的线程。这是因为被阻塞的线程往往是放弃自己的时间片并且不再要求获得 CPU，但只要这些线程获得了比其它线程更高的优先级，或者其它同一优先级的线程用完了自己的时间片，它们就可以恢复执行。注意，正处于自己时间片中的线程不能被阻塞。

用于提升阻塞线程优先级的 **boost** 值不太好选择。一个较好的笨方法是指定 **IO_NO_INCREMENT** 值，当然，如果你有更好的值，可以不用这个值。如果事件唤醒的是一个处理时间敏感数据流的线程(如声卡驱动程序)，那么应该使用适合那种设备的 **boost** 值(如 **IO_SOUND_INCREMENT**)。重要的是，不要为一个愚蠢的理由去提高等待者的优先级。例如，如果你要同步处理一个 **IRP_MJ_PNP** 请求，那么在你要停下来等待低级驱动程序处理完该 **IRP** 时，你的完成例程应调用 **KeSetEvent**。由于 PnP 请求对于处理器没有特殊要求并且也不经常发生，所以即使是声卡驱动程序也也应该把 **boost** 参数指定为 **IO_NO_INCREMENT**。

KeSetEvent 的第三个参数

wait 参数的目的是允许在内部快速地把控制从一个线程传递到另一个线程。除了设备驱动程序之外，大部分系统部件都可以创建双事件对象。例如，客户线程和服务器线程使用双事件对象来界定它们的通信。当服务器线程需要唤醒对应的客户线程时，它首先调用 **KeSetEvent** 函数，并指定 **wait** 参数为 **TRUE**，然后立即调用 **KeWaitXxx** 函数使自己进入睡眠状态。由于这两个操作都以原子方式完成，所以在控制交接时没有其它线程被唤醒。

DDK 总是稍稍地描述一些内部细节，但我发现有些描述另人迷惑。我将以另一种方式解释这些内部细节，看过这些细节后你就会明白为什么我们总指定这个参数为 **FALSE**。在内部，内核使用一个“同步数据库锁(dispatcher database lock)”来保护线程的阻塞、唤醒，和调度操作。**KeSetEvent** 函数需要获取这个锁，**KeWaitXxx** 函数也是这样。如果你把这个参数指定为 **TRUE**，则 **KeSetEvent** 函数将设置一个标志以便 **KeWaitXxx** 函数知道你使用了 **TRUE** 参数，然后它返回，并且不释放这个锁。当你后来(应该立即调用，因为你此时正运行在一个比任何硬件设备都高的 **IRQL** 上，并且你占有着一个被极其频繁争夺的自旋锁)调用 **KeWaitXxx** 函数时，它不必再获取这个锁。产生的效果就是你唤醒了等待的线程并同时把自己置入睡眠状态，而不给其它线程任何运行的机会。

你应该明白，以 **wait** 参数为 **TRUE** 调用 **KeSetEvent** 的函数必须存在于非分页内存中，因为它在某段时间内执行在提升的 **IRQL** 上。很难想象一个普通设备驱动程序会需要使用这种机制，因为驱动程序决不会比内核更了解线程的调度。底线是：对该参数总使用 **FALSE**。实际上，Microsoft 暴露该参数给我们的原因仍不十分清楚。

调用 **KeReadStateEvent** 函数(在任何 **IRQL** 上)可以测试事件的当前状态：

```
LONG signalled = KeReadStateEvent(event);
```

返回值不为 0 代表事件处于信号态，为 0 代表事件处于非信号态。

注意

Windows 98 不支持 **KeReadStateEvent** 函数，但支持上面描述的其它 **KeReadStateXxx** 函数。为了获得事件的状态，我们必须使用 Windows 98 的其它同步原语。

调用 **KeResetEvent** 函数(在低于或等于 **DISPATCH_LEVEL** 级)可以立即获得事件对象的当前状态，但该函数会把事件对象重置为非信号状态。

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LONG signalled = KeResetEvent(event);
```

如果你对事件的上一个状态不感兴趣，可以调用 **KeClearEvent** 函数，象下面这样：

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
KeClearEvent(event);
```

KeClearEvent 函数执行得更快，因为它在读取事件的当前状态后不设置事件为非信号态。

内核信号灯

内核模式信号灯是一个有同步语义的整数计数器。信号灯计数器为正值时代表信号态，为 0 时代表非信号态。计数器不能为负值。释放信号灯将使信号灯计数器增 1，在一个信号灯上等待将使该信号灯计数器减 1。如果计数器值被减为 0，则信号灯进入非信号态，之后其它调用 **KeWaitXxx** 函数的线程将被阻塞。注意如果等待线程的个数超过了计数器的值，那么并不是所有等待的线程都可以恢复运行。

内核提供了三个服务函数来控制信号灯对象的状态。(见表 4-3) 信号灯对象应该在 PASSIVE_LEVEL 级上初始化:

```
ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);
KeInitializeSemaphore(semaphore, count, limit);
```

在这个调用中, **semaphore** 参数指向一个在非分页内存中的 KSEMAPHORE 对象。**count** 是信号灯计数器的初始值, **limit** 是计数器能达到的最大值, 它必须与信号灯计数器的初始值相同。

表 4-3. 内核信号灯对象服务函数

服务函数	描述
KeInitializeSemaphore	初始化信号灯对象
KeReadStateSemaphore	取信号灯当前状态
KeReleaseSemaphore	设置信号灯对象为信号态

如果你创建信号灯时指定 **limit** 参数为 1, 则该对象与仅有一个线程的互斥对象类似。但内核互斥对象有一些信号灯没有的特征, 这些特征用于防止死锁。所以, 没有必要创建 **limit** 为 1 的信号灯。

如果你以一个大于 1 的 **limit** 值创建信号灯, 则该信号灯允许多个线程同时访问某些资源。在队列理论中我们会发现同样的原理, 单队列可以被多个服务程序使用。多个服务程序使用一个队列要比每个服务程序都有各自的队列更合理。这两种形式的平均等待时间是相同的, 但前者的等待次数更少。使用信号灯, 你可以把一组软件或硬件服务程序按照队列原理组织起来。

信号灯的所有者可以调用 **KeReleaseSemaphore** 函数释放信号灯:

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LONG wassignalled = KeReleaseSemaphore(semaphore, boost, delta, wait);
```

这里出现了一个 **delta** 参数, 它必须为正数, 该函数把 **delta** 值加到 **semaphore** 指向的信号灯计数器上, 这将把信号灯带入信号态, 并使等待线程释放。通常, 该参数应该指定为 1, 代表有一个所有者释放了它的权利。**boost** 和 **wait** 参数与在 **KeSetEvent** 函数中的作用相同。返回值为 0 代表信号灯的前一个状态是非信号态, 非 0 代表信号灯的前一个状态为信号态。

KeReleaseSemaphore 不允许你把计数器的值增加到超过 **limit** 指定的值。如果你这样做, 该函数根本就不调整计数器的值, 它将产生一个代码为 STATUS_SEMAPHORE_LIMIT_EXCEEDED 的异常。除非系统中存在捕获该异常的处理程序, 否则将导致一个 bug check。

下面调用读取信号灯的当前状态:

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LONG signalled = KeReadStateSemaphore(semaphore);
```

非 0 返回值表示信号灯处于信号态, 0 返回值代表信号灯为非信号态。不要把该返回值假定为计数器的当前值。

内核互斥对象

互斥(mutex)就是互相排斥(mutual exclusion)的简写。内核互斥对象为多个竞争线程串行化访问共享资源提供了一种方法(不一定是最好的方法)。如果互斥对象不被某线程所拥有, 则它是信号态, 反之则是非信号态。当线程为了获得互斥对象的控制权而调用 **KeWaitXxx** 例程时, 内核同时也做了一些工作以帮助避免可能的死锁。同样, 互斥对象也需要与 **KeWaitForSingleObject** 类似的附加动作。内核可以确保线程不被换出, 并且阻止所有 APC 的提交, 内核专用 APC(如 **IoCompleteRequest** 用以完成 I/O 请求的 APC)除外。

通常我们应该使用 **executive** 部件输出的快速互斥对象而不是内核互斥对象。这两者的主要不同是, 内核互斥可以被递归获取, 而 **executive** 快速互斥则不能。即内核互斥的所有者可以调用 **KeWaitXxx** 并指定所拥有的互

斥对象从而使等待立即被满足。如果一个线程真的这样做，它必须也要以同样的次数释放该互斥对象，否则该互斥对象不被认为是空闲的。

如果你需要长时间串行化访问一个对象，你应该首先考虑使用互斥(而不是依赖提升的 IRQL 和自旋锁)。利用互斥对象控制资源的访问，可以使其它线程分布到多处理器平台上的其它 CPU 中运行，还允许导致页故障的代码仍能锁定资源而不被其它线程访问。表 4-4 列出了互斥对象的服务函数。

表 4-4. 互斥对象服务函数

服务函数	描述
KeInitializeMutex	初始化互斥对象
KeReadStateMutex	取互斥对象的当前状态
KeReleaseMutex	设置互斥对象为信号态

为了创建一个互斥对象，你需要为 KMUTEX 对象保留一块非分页内存，然后象下面这样初始化：

```
ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);
KeInitializeMutex(mutex, level);
```

mutex 是 KMUTEX 对象的地址，**level** 参数最初是用于辅助避免多互斥对象带来的死锁。但现在，内核忽略 **level** 参数。

互斥对象的初始状态为信号态，即未被任何线程拥有。KeWaitXxx 调用将使调用者接管互斥对象的控制并使其进入非信号态。

利用下面函数可以获取互斥对象的当前状态：

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LONG signalled = KeReadStateMutex(mutex);
```

返回值 0 表示互斥对象已被占用，非 0 表示未被占用。

下面函数可以使所有者放弃其占有的互斥对象并使其进入信号态：

```
ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);
LONG wassignalled = KeReleaseMutex(mutex, wait);
```

wait 参数与 KeSetEvent 函数中的含义相同。该函数返回值总是 0，表示该互斥对象曾被占用过，如果不是这种情况(所有者释放的不是它自己的对象)，KeReleaseMutex 将产生 bug check。

出于完整性的考虑，我想提一下 KeWaitForMutexObject 函数，它是 DDK 中的宏(见 WDM.H)。其定义如下：

```
#define KeWaitForMutexObject KeWaitForSingleObject
```

内核定时器

内核还提供了一种定时器对象，该对象可以在指定的绝对时间或间隔时间后自动从非信号态变为信号态。它还可以周期性地进入信号态。我们可以用它来安排一个定期执行的 DPC 回调函数。表 4-5 列出了用于定时器对象的服务函数。

表 4-5. 内核定时器对象的服务函数

服务函数	描述
KeCancelTimer	取消一个活动的定时器

KeInitializeTimer	初始化一次性的通知定时器
KeInitializeTimerEx	初始化一次性的或重复通知的或同步的定时器
KeReadStateTimer	获取定时器的当前状态
KeSetTimer	为通知定时器设定时间
KeSetTimerEx	为定时器设定时间和其它属性

通知定时器用起来象事件

在这一段中，我们将创建一个通知定时器对象并等到它达到预定时间。首先，我们在非分页内存中分配一个 **KTIMER** 对象。然后，我们在低于或等于 **DISPATCH_LEVEL** 级上初始化这个定时器对象：

```
PKTIMER timer;           // someone gives you this
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
KeInitializeTimer(timer);
```

在此，定时器处于非信号状态，它还没有开始倒计时，在这样的定时器上等待的线程永远得不到唤醒。为了启动定时器倒计时，我们调用 **KeSetTimer** 函数：

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LARGE_INTEGER duetime;
BOOLEAN wascounting = KeSetTimer(timer, duetime, NULL);
```

duetime 是一个 64 位的时间值，单位为 100 纳秒。如果该值为正，则表示一个从 1601 年 1 月 1 日算起的绝对时间。如果该值为负，则它是相对于当前时间的一段时间间隔。

返回值如果为 **TRUE**，则表明定时器已经启动。(在这种情况下，如果我们再调用 **KeSetTimer** 函数，则定时器放弃原来的时间重新开始倒计时)

下面语句读取定时器的当前状态：

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
BOOLEAN counting = KeReadStateTimer(timer);
```

KeInitializeTimer 和 **KeSetTimer** 实际上是旧的服务函数，它们已经被新函数取代。我们可以用下面调用初始化定时器：

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
KeInitializeTimerEx(timer, NotificationTimer);
```

定时器设置函数也有扩展版本，**KeSetTimerEx**：

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LARGE_INTEGER duetime;
BOOLEAN wascounting = KeSetTimerEx(timer, duetime, 0, NULL);
```

我将在本章后面解释该函数扩展版本的新参数。

即使定时器开始倒计时，它仍处于非信号态，直到到达指定的时间。在那个时刻，该定时器对象自动变为信号态，所有等待的线程都被释放。

通知定时器与DPC例程

在这一小节中，我们想让定时器去触发一个 DPC 例程。使用这种方法，不论你的线程有什么优先级都会响应超时事件。(因为线程只能在 PASSIVE_LEVEL 级上等待，而定时器到时间后，获取 CPU 控制权的线程是随机的。然而，DPC 例程执行在提升的 IRQL 级上，它可以有效地抢先所有线程)

我们用同样的方法初始化定时器对象。另外我们还再初始化一个 KDPC 对象，该对象应该在非分页内存中分配。如下面代码：

```
PKDPC dpc; // points to KDPC you've allocated  
ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);  
KeInitializeTimer(timer);  
KeInitializeDpc(dpc, DpcRoutine, context);
```

用 KeInitializeTimer 或 KeInitializeTimerEx 初始化定时器对象。**DpcRoutine** 是一个 DPC(推迟过程调用)例程的地址，这个例程必须存在于非分页内存中。**context** 参数是一个任意的 32 位值(类型为 PVOID)，它将作为参数传递给 DPC 例程。**dpc** 参数是一个指向 KDPC 对象的指针(该对象必须在非分页内存中。例如，在你的设备扩展中)。

当开始启动定时器的倒计时，我们把 DPC 对象指定为 KeSetTimer 或 KeSetTimerEx 函数的一个参数：

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);  
LARGE_INTEGER duetime;  
BOOLEAN wascounting = KeSetTimer(timer, duetime, dpc);
```

这个 KeSetTimer 调用与上一段中的调用的不同之处是，我们在最后一个参数中指定了一个 DPC 对象地址。当定时器时间到时，系统将把该 DPC 排入队列，并且只要条件允许就立即执行它。在最差的情况下，它也与在 PASSIVE_LEVEL 级上唤醒线程一样快。DPC 函数的定义如下：

```
VOID DpcRoutine(PKDPC dpc, PVOID context, PVOID junk1, PVOID junk2)  
{  
    ...  
}
```

即使你为 KeSetTimer 或 KeSetTimerEx 提供了 DPC 参数，你仍可以调用 KeWaitXxx 函数使自己在 PASSIVE_LEVEL 级上等待。在单 CPU 的系统上，DPC 将在等待完成前执行，因为它执行在更高的 IRQL 上。

同步定时器

与事件对象类似，定时器对象也有两种形式：通知方式和同步方式。通知定时器允许有任意数量的等待线程。同步定时器正相反，它只允许有一个等待线程。一旦有线程在这种定时器上等待，定时器就自动进入非信号态。为了创建同步定时器，你必须使用扩展形式的初始化函数：

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);  
KeInitializeTimerEx(timer, SynchronizationTimer);
```

SynchronizationTimer 是枚举类型 TIMER_TYPE 的一个枚举值。另一个枚举值是 **NotificationTimer**。

如果你在同步定时器上使用 DPC 例程，可以把排队 DPC 看成是定时器到期时发生的额外事情。即定时器到期时，系统把定时器置成信号态，并把 DPC 对象插入 DPC 队列。定时器进入信号态将使阻塞的线程得以释放。

周期性定时器

到现在为止，我们讨论过的定时器仅能定时一次。通过使用定时器的扩展设置函数，你可以请求一个周期性的超时：

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LARGE_INTEGER duetime;
BOOLEAN wascounting = KeSetTimerEx(timer, duetime, period, dpc);
```

这里，**period** 是周期超时值，单位为毫秒(ms)，**dpc** 是一个可选的指向 **KDPC** 对象的指针。这种定时器在第一次倒计时时使用 **duetime** 时间，到期后再使用 **period** 值重复倒计时。为了准确地完成周期定时，应该把 **duetime** 时间指定为与周期间隔参数一样的相对时间。指定为 0 的 **duetime** 参数将使定时器立即完成第一次倒计时，然后开始周期性倒计时。由于不用重复等待超时通知，所以周期性定时器常常与 **DPC** 对象联用。

取消一个周期性定时器

在定时器对象超出定义范围之外前，一定要调用 **KeCancelTimer** 取消任何已创建的周期性定时器。如果这个周期性定时器带有一个 **DPC**，则还需要在取消该定时器之后调用 **KeRemoveQueueDpc**。甚至即使你做了这两件事，还可能出现一个无法解决的问题。如果你在 **DriverUnload** 例程中取消这种定时器，可能会出现一种罕见的情形：你的驱动程序已被卸载，但那个 **DPC** 例程的实例却仍运行在另一个 **CPU** 上。这个问题只有等待未来版本的操作系统来解决。你可以尽早地取消这种定时器以便减少该问题出现的可能性，比如在 **IRP_MN_REMOVE_DEVICE** 的处理程序中。

一个例子

内核定时器的一个用处是为定期检测设备活动的系统线程提供循环定时。今天很少有设备需要循检服务，但你可能遇到例外情况。我将在第九章中讨论这个主题。随书光盘中有一个例子(**POLLING**)演示了这个概念。这个例子的部分代码以固定的间隔时间循检设备。这个循环可以被设置的 **kill** 事件所打破，所以程序使用了 **KeWaitForMultipleObjects** 函数。实际的代码要比下面的例子更复杂一些，下面代码片段主要侧重于定时器的使用：

```
VOID PollingThreadRoutine(PDEVICE_EXTENSION pdx)
{
    NTSTATUS status;
    KTIMER timer;
    KeInitializeTimerEx(&timer, SynchronizationTimer);           <--1
    PVOID pollevents[] = {                                         <--2
        (PVOID) &pdx->evKill,
        (PVOID) &timer,
    };
    ASSERT(arraysize(pollevents) <= THREAD_WAIT_OBJECTS);

    LARGE_INTEGER duetime = {0};
    #define POLLING_INTERVAL 500
    KeSetTimerEx(&timer, duetime, POLLING_INTERVAL, NULL);       <--3
    while (TRUE)
    {
        status = KeWaitForMultipleObjects(arraysize(pollevents),      <--4
            pollevents,
            WaitAny,
            Executive,
            KernelMode,
            FALSE,
            NULL,
            NULL);
        if (status == STATUS_WAIT_0)
            break;
        if (<device needs attention>)                                <--5
            <do something>;
    }
}
```

```
    }
    KeCancelTimer(&timer);
    PsTerminateSystemThread(STATUS_SUCCESS);
}
```

- 在此，我们把一个内核定时器初始化成同步方式。它只能用于一个线程，本线程。
- 我们需要为 **KeWaitForMultipleObjects** 函数提供一个同步对象指针数组。第一个数组元素是 **kill** 事件对象，驱动程序的其它部分可能在系统线程需要退出时设置这个对象，以终止循环。第二个数组元素就是定时器对象。
- KeSetTimerEx** 语句启动周期定时器。由于 **duetime** 参数是 0，所以定时器立即进入信号态。然后每隔 500 毫秒触发一次。
- 在循检循环内，我们等待定时器到期或 **kill** 事件发生。如果等待由于 **kill** 事件而结束，我们退出循环，并做一些清理工作，最后终止这个系统线程。如果等待是由定时器到期而结束，我们就前进到下一步处理。
- 在这里，设备驱动程序可以做一些与硬件有关的操作。

定时函数

除了使用内核定时器对象外，你还可以使用另外两个定时函数，它们也许更适合你。第一函数是 **KeDelayExecutionThread**，你可以在 **PASSIVE_LEVEL** 级上调用该函数并给出一个时间间隔。该函数省去了使用定时器时的麻烦操作，如创建，初始化，设置，等待操作。

```
ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);
LARGE_INTEGER duetime;
NTSTATUS status = KeDelayExecutionThread(WaitMode, Alertable, &duetime);
```

在这里，**WaitMode**、**Alertable**，和函数返回代码与 **KeWaitXxx** 中的对应部分有相同的含义。**duetime** 也是内核定时器中使用的同一种时间表达类型。

如果你需要延迟一段非常短的时间(少于 50 毫秒)，可以调用 **KeStallExecutionProcessor**，在任何 IRQL 级上：

```
KeStallExecutionProcessor(nMicroSeconds);
```

这个延迟的目的是允许硬件在程序继续执行前有时间为下一次操作做准备。实际的延迟时间可能大大超过你请求的时间，因为 **KeStallExecutionProcessor** 可以被其它运行在更高 IRQL 级上的活动抢先，但不能被同一 IRQL 级上的活动抢先。

内核线程同步

操作系统的进程结构部件(**Process Structure**)提供了一些例程，WDM 驱动程序可以使用这些例程创建和控制内核线程，这些例程可以帮助驱动程序周期性循检设备，我将在第九章中讨论这些例程。出于完整性考虑，我在这里先提一下。如果在 **KeWaitXxx** 调用中指定一个内核线程对象，那么你的线程将被阻塞直到那个内核线程结束运行。那个内核线程通过调用 **PsTerminateSystemThread** 函数终止自身。

为了等待某内核线程结束，你首先应获得一个 **KTHREAD** 对象(不透明对象)的指针，在内部，该对象用于代表内核线程，但这里还有一点问题，当你运行在某线程的上下文中时，你可以容易地获取当前线程的 **KTHREAD** 指针：

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
PKTHREAD thread = KeGetCurrentThread();
```

不幸的是，当你调用 **PsCreateSystemThread** 创建新内核线程时，你仅能获取该线程的不透明句柄。为了获得 **KTHREAD** 对象指针，你必须使用对象管理器服务函数：

```
HANDLE hthread;
PKTHREAD thread;
```

```
PsCreateSystemThread(&hread, ...);
ObReferenceObjectByHandle(hread,
    THREAD_ALL_ACCESS,
    NULL,
    KernelMode,
    (PVOID*) &thread,
    NULL);

ZwClose(hread);
```

ObReferenceObjectByHandle 函数把你提供的句柄转换成一个指向下层内核对象的指针。一旦有了这个指针，你就可以调用 **ZwClose** 关闭那个句柄。在某些地方，你还需要调用 **ObDereferenceObject** 函数释放对该线程对象的引用。

```
ObDereferenceObject(thread);
```

线程警惕和APC

在内部，Windows NT 内核有时使用线程警惕(thread alert)来唤醒线程。这种方法使用 APC(异步过程调用)来唤醒线程去执行某些特殊例程。用于生成警惕和 APC 的支持例程没有输出给 WDM 驱动程序开发者使用。但是，由于 DDK 文档和头文件中有大量地方引用了这个概念，所以我想在这里谈一下。

当某人通过调用 **KeWaitXxx** 例程阻塞一个线程时，需要指定一个布尔参数，该参数表明等待是否是警惕的(alertable)。一个警惕的等待可以提前完成，即不用满足任何等待条件或超时，仅由于线程警惕。线程警惕起源于用户模式的 native API 函数 **NtAlertThread**。如果因为警惕等待提前终止，则内核返回特殊的状态值 **STATUS_ALERTED**。

APC 机制使操作系统能在特定线程上下文中执行一个函数。APC 的异步含义是，系统可以有效地中断目标线程以执行一个外部例程。APC 的动作有点类似于硬件中断使处理器从任何当前代码突然跳到 ISR 的情形，它是不可预见的。

APC 来自三种地方：用户模式、内核模式，和特殊内核模式。用户模式代码通过调用 Win32 API 函数 **QueueUserAPC** 请求一个用户模式 APC。内核模式代码通过调用一个未公开的函数请求一个 APC，而且该函数在 DDK 头文件中没有原型。某些逆向工程师可能已经知道该例程的名称以及如何调用它，但该函数的确是仅用于内部，所以我不在这里讨论它。系统把 APC 排入一个特殊线程直到合适的执行条件出现。合适的执行条件要取决于 APC 的类型，如下：

- 特殊的内核 APC 被尽可能快地执行，既只要 APC_LEVEL 级上有可调度的活动。在很多情况下，特殊的内核 APC 甚至能唤醒阻塞的线程。
- 普通的内核 APC 仅在所有特殊 APC 都被执行完，并且目标线程仍在运行，同时该线程中也没有其它内核模式 APC 正执行时才执行。
- 用户模式 APC 在所有内核模式 APC 执行完后才执行，并且仅在目标线程有警惕属性时才执行。

如果系统唤醒线程去提交一个 APC，则使该线程阻塞的等待原语函数将返回特殊状态值 **STATUS_KERNEL_APIC** 或 **STATUS_USER_APIC**。

APC与I/O请求

内核使用 APC 概念有多种目的。由于本书仅讨论驱动程序的编写，所以我仅解释 APC 与执行 I/O 操作之间的关系。在某些场合，当用户模式程序在一个句柄上执行同步的 **ReadFile** 操作时，Win32 子系统就调用一个名为 **NtReadFile**(尽管未公开，但已经被广泛了解)的内核模式例程。该函数创建并提交一个 IRP 到适当的设备驱动程序，而驱动程序通常返回 **STATUS_PENDING** 以指出操作未完成。**NtReadFile** 然后向 **ReadFile** 也返回这个状态代码，于是 **ReadFile** 调用 **NtWaitForSingleObject** 函数，这将使应用程序在那个用户模式句柄指向的文件对象上等待。**NtWaitForSingleObject** 接着调用 **KeWaitForSingleObject** 以执行一个非警惕的用户模式的等待，在文件对象内部的一个事件对象上等待。

当设备驱动程序最后完成了读操作时，它调用 `IoCompleteRequest` 函数，该函数接下来排队一个特殊的内核模式 APC。该 APC 例程然后调用 `KeSetEvent` 函数使文件对象进入信号态，因此应用程序被释放并得以继续执行。有时，I/O 请求被完成后还需要执行一些其它任务，如缓冲区复制，而这些操作又必须发生在请求线程的地址上下文中，因此会需要其它种类的 APC。如果请求线程不处于警惕性的等待状态，则需要内核模式 APC。如果在提交 APC 时线程并不适合运行，则需要特殊的 APC。实际上，APC 例程就是用于唤醒线程的机制。

内核模式例程也能调用 `NtReadFile` 函数。但驱动程序应该调用 `ZwReadFile` 函数替代，它使用与用户模式程序一样的系统服务接口到达 `NtReadFile`(注意，`NtReadFile` 函数未公开给设备驱动程序使用)。如果你遵守 DDK 的限定调用 `ZwReadFile` 函数，那么你向 `NtReadFile` 的调用与用户模式中的调用几乎没有什不同，仅有两处不同。第一，`ZwReadFile` 函数更小，并且任何等待都将在内核中完成。另一个不同之处是，如果你调用了 `ZwCreateFile` 函数并指定了同步操作，则 I/O 管理器将自动等待你的读操作直到完成。这个等待可以是警惕的也可以不是，取决于你在 `ZwCreateFile` 调用中指定的实际选项。

如何指定 `Alertable` 和 `WaitMode` 参数

现在你已经有足够的背景资料了解等待原语中的 `Alertable` 和 `WaitMode` 参数。作为一个通用规则，你绝不要写同步响应用户模式请求的代码，仅能为确定的 I/O 控制请求这样做。一般说来，最好挂起长耗时的操作(从派遣例程中返回 `STATUS_PENDING` 代码)而以异步方式完成。再有，你不要一上来就调用等待原语。线程阻塞仅适合设备驱动程序中的某几个地方使用。下面几段介绍了这几个地方。

内核线程 有时，当你的设备需要周期性循环时，你需要创建自己的内核模式线程。

处理 PnP 请求 我将在第六章中讨论如何处理 PnP 管理器发送给你的 I/O 请求。有几个 PnP 请求需要你在驱动程序这边同步处理。换句话说，你把这些请求传递到低级驱动程序并等待它们完成。你将调用 `KeWaitForSingleObject` 函数并在内核模式中等待，这是由于 PnP 管理器是在内核模式线程的上下文中调用你的驱动程序。另外，如果你需要执行作为处理 PnP 请求一部分的辅助请求时，例如，与 USB 设备通信，你应该在内核模式中等待。

处理其它 I/O 请求 当你正在处理其它种类的 I/O 请求时，并且你知道正运行在一个非任意线程上下文中时，那么你在行动前必须仔细考虑，如果你确信那个线程可以被阻塞，你应该在调用者所处的处理器模式中等待。在多数情况下，你可以利用 IRP 中的 `RequestorMode` 域。此外，你还可以调用 `ExGetPreviousMode` 来确定前一个处理器模式。如果你在用户模式中等待，并允许用户程序调用 `QueueUserAPC` 提前终止等待，你应该执行一个警惕性等待。

我最后要提到的情况是，在用户模式中等待并要允许用户模式 APC 打断，你应使用警惕性等待。

底线是：使用非警惕性等待，除非你知道不这样做的原因。

下面是 MFC 中的同步对象及其用途，仅供参考

类 `CSemaphore` 的对象代表一个“信号灯”：一种同步对象，允许一个或多个进程中的有限数量的线程访问某个资源。一个 `CSemaphore` 对象维护着当前正访问某个资源的进程的个数。信号灯通常用于控制仅能支持有限数量用户的共享资源的访问。`CSemaphore` 对象计数器的当前值表示还可以允许多少个用户使用它保护的共享资源。当这个数到达 0 时，所有试图访问被保护资源的操作都被放入一个系统队列等待，直到计数数值上升到 0 以上或等待超时。

类 `CEvent` 的对象代表一个“事件”：一种同步对象，用于一个线程通知另一个线程某事件发生。事件通常用于线程想知道何时执行其任务。例如，复制数据到某文件的线程需要被通知数据何时准备好。用 `CEvent` 对象可以通知复制线程数据已经有效，这样线程就可以尽快地执行其任务。`CEvent` 对象有两种类型：手动和自动。手动 `CEvent` 对象将停留在 `SetEvent` 或 `ResetEvent` 设置的状态，除非你再次设置它。自动 `CEvent` 对象在一个线程释放后自动回到非信号态(无效状态)。

类 `CMutex` 的对象代表一个“互斥”：一种同步对象，可以使一个线程排斥性地访问某个资源。互斥可以用于一次仅允许一个线程访问的资源。例如，向链表添加一个接点的过程就是一次只允许一个线程执行。通过使用 `CMutex` 对象控制链表，一次只能有一个线程获得链表的访问权。

类 **CCriticalSection** 的对象代表一个“关键段”：一种同步对象，代表一次只允许一个线程访问的资源或代码片段。例如，向链表添加一个节点。使用 **CCriticalSection** 对象控制的链表一次只允许一个线程访问该链表。如果着重于执行速度并且被保护资源不跨进程使用，也可以用关键段代替互斥。

其它内核模式同步要素

Windows 2000 内核为同步线程执行和保护共享对象访问提供了一些额外的方法。在这一节中，我们将讨论快速互斥(**fast mutex**)对象，通过对无竞争情况的优化处理，它可以提供比普通内核互斥对象更快的执行性能。我还将描述一种名称中含有“互锁(Interlocked)”术语的支持函数。这些函数都执行某种公用操作，例如增加或减少一个整数的值，从链表中插入或删除一个表项，这些操作都是以原子方式执行，从而可以避免多任务或多处理器的干扰。

快速互斥对象

参照内核互斥，表 4-6 列出了快速互斥的优点和缺点。有利的一面，快速互斥在没有实际竞争的情况下可以快速获取和释放。不利的一面，你不能递归获取一个快速互斥对象。即如果你拥有快速互斥对象你就不能发出 APC，这意味着你将处于 APC_LEVEL 或更高的 IRQL，在这一级上，线程优先级将失效，但你的代码将不受干扰地执行，除非有硬件中断发生。

表 4-6. 内核互斥和快速互斥的比较

内核互斥	快速互斥
可以被单线程递归获取(系统为其维护一个请求计数器)	不能被递归获取
速度慢	速度快
所有者只能收到“特殊的”内核 APC	所有者不能收到任何 APC
所有者不能被换出内存	不自动提升被阻塞线程的优先级(如果运行在大于或等于 APC_LEVEL 级)，除非你使用 XxxUnsafe 函数并且执行在 PASSIVE_LEVEL 级上
可以是多对象等待的一部分	不能作为 KeWaitForMultipleObjects 的参数使用

表 4-7 列出了与快速互斥相关的服务函数

表 4-7. 快速互斥服务函数

服务函数	描述
ExAcquireFastMutex	获取快速互斥，如果必要则等待
ExAcquireFastMutexUnsafe	获取快速互斥，如果必要则等待，调用者必须先停止接收 APC
ExInitializeFastMutex	初始化快速互斥对象
ExReleaseFastMutex	释放快速互斥
ExReleaseFastMutexUnsafe	释放快速互斥，不解除 APC 提交禁止
ExTryToAcquireFastMutex	获取快速互斥，如果可能，立即获取不等待

为了创建一个快速互斥，你必须先在非分页内存中分配一个 **FAST_MUTEX** 数据结构。然后调用 **ExInitializeFastMutex** 函数初始化该快速互斥对象。实际上，在 WDM.H 中该函数是一个宏：

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
ExInitializeFastMutex(FastMutex);
```

FastMutex 是 **FAST_MUTEX** 对象的地址。快速互斥开始于无主状态。为了获取快速互斥对象，调用下面函数：

```
ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
ExAcquireFastMutex(FastMutex);
```

或

```
ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
ExAcquireFastMutexUnsafe(FastMutex);
```

第一种函数等待互斥变成有效状态，然后再把所有权赋给调用线程，最后把处理器当前的 IRQL 提升到 APC_LEVEL。IRQL 提升的结果是阻止所有 APC 的提交。第二种函数不改变 IRQL。在使用这个“不安全”的函数获取快速互斥前你需要考虑潜在的死锁可能。必须避免运行在同一线程上下文下的 APC 例程获取同一个互斥或任何其它不能被递归锁定的对象。否则你将冒随时死锁那个线程的风险。

如果你不想在互斥没立即有效的情况下等待，使用“尝试获取”函数：

```
ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
BOOLEAN acquired = ExTryToAcquireFastMutex(FastMutex);
```

如果返回值为 TRUE，则你已经拥有了该互斥。如果为 FALSE，表明该互斥已经被别人占有，你不能获取。

为了释放一个快速互斥并允许其它线程请求它，调用适当的释放函数：

```
ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
ExReleaseFastMutex(FastMutex);
```

或

```
ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
ExReleaseFastMutexUnsafe(FastMutex);
```

快速互斥之所以快速是因为互斥的获取和释放步骤都为没有竞争的情况做了优化。获取互斥的关键步骤是自动减和测试一个整数计数器，该计数器指出有多少线程占有或等待该互斥。如果测试表明没有其它线程占有该互斥，则没有额外的工作需要做。如果测试表明有其它线程拥有该互斥，则当前线程将阻塞在一个同步事件上，该同步事件是 FAST_MUTEX 对象的一部分。释放互斥时必须自动增并测试计数器。如果测试表明当前没有等待线程，则没有额外的工作要做。如果还有线程在等待，则互斥所有者需调用 KeSetEvent 函数释放一个等待线程。

互锁运算

在 WDM 驱动程序能调用的函数中，有一些函数可以以线程安全和多处理器安全的方式执行算术运算。见表 4-8。这些例程有两种形式，第一种形式以 **Interlocked** 为名字开头，它们可以执行原子操作，其它线程或 CPU 不能干扰它们的执行。另一种形式以 **ExInterlocked** 为名字开头，它们使用自旋锁。

表 4-8. 互锁运算服务函数

服务函数	描述
InterlockedCompareExchange	比较并有条件地交换两个值
InterlockedDecrement	整数减 1
InterlockedExchange	交换两个值
InterlockedExchangeAdd	加两个值并返回和
InterlockedIncrement	整数加 1
ExInterlockedAddLargeInteger	向 64 位整数加
ExInterlockedAddLargeStatistic	向 ULONG 加
ExInterlockedAddUlong	向 ULONG 加并返回原始值
ExInterlockedCompareExchange64	交换两个 64 位值

InterlockedXxx 函数可以在任意 IRQL 上调用；由于该函数不需要自旋锁，所以它们还可以在 PASSIVE_LEVEL 级上处理分页数据。尽管 **ExInterlockedXxx** 函数也可以在任意 IRQL 上调用，但它们需要在大于或等于 DISPATCH_LEVEL 级上操作目标数据，所以它们的参数需要在非分页内存中。使用 **ExInterlockedXxx** 的唯一原因是，如果你有一个数据变量，且需要增减该变量的值，并且有时还需要用其它指令序列直接访问该变量。你可以在对该变量的多条访问代码周围明确声明自旋锁，然后仅用 **ExInterlockedXxx** 函数执行简单的增减操作。

InterlockedXxx函数

InterlockedIncrement 向内存中的长整型变量加 1，并返回加 1 后的值：

```
LONG result = InterlockedIncrement(pLong);
```

pLong 是类型为 **LONG** 的变量的地址，概念上，该函数的操作等价于 C 语句：**return ++*pLong**，但它与简单的 C 语句的不同地方是提供了线程安全和多处理器安全。**InterlockedIncrement** 可以保证整数变量被成功地增 1，即使其它 CPU 上的线程或同一 CPU 上的其它线程同时尝试改变这个整数的值。就操作本身来说，它不能保证所返回的值仍是该变量当前的值，甚至即使仅仅过了一个机器指令周期，因为一旦这个增 1 原子操作完成，其它线程或 CPU 就可能立即修改这个变量。

InterlockedDecrement 除了执行减 1 操作外，其它方面同上。

```
LONG result = InterlockedDecrement(pLong);
```

InterlockedCompareExchange 函数可以这样调用：

```
LONG target;
LONG result = InterlockedCompareExchange(&target, newval, oldval);
```

target 是一个类型为 **LONG** 的整数，既可以用于函数的输入也可以用于函数的输出，**oldval** 是你对 **target** 变量的猜测值，如果这个猜测正确，则 **newval** 被装入 **target**。该函数的内部操作与下面 C 代码类似，但它是以原子方式执行整个操作，即它是线程安全和多处理器安全的：

```
LONG CompareExchange(PLONG pttarget, LONG newval, LONG oldval)
{
    LONG value = *pttarget;
    if (value == oldval)
        *pttarget = newval;
    return value;
}
```

换句话说，该函数总是返回 **target** 变量的历史值给你。此外，如果这个历史值等于 **oldval**，那么它把 **target** 的值设置为 **newval**。该函数用原子操作实现比较和交换，而交换仅在历史值猜测正确的情况下才发生。

你还可以调用 **InterlockedCompareExchangePointer** 函数来执行类似的比较和交换操作，但该函数使用指针参数。该函数或者定义为编译器内部的内联函数，或者是一个真实的函数，取决于你编译时平台的指针宽度，以及编译器生成内联代码的能力。下面例子中使用了这个指针版本的比较交换函数，它把一个结构加到一个单链表的头部，而不用使用自旋锁或提升 IRQL：

```
typedef struct _SOMESTRUCTURE {
    struct _SOMESTRUCTURE* next;
    ...
} SOMESTRUCTURE, *PSOMESTRUCTURE;
...
void InsertElement(PSOMESTRUCTURE p, PSOMESTRUCTURE* anchor)
{
    PSOMESTRUCTURE next, first;
```

```

do
{
    p->next = first = *anchor;
    next = InterlockedCompareExchangePointer(anchor, p, first);
}
while (next != first);
}

```

每一次循环中，我们都假设新元素将连接到链表的当前头部，即变量 **first** 中的地址。然后我们调用 **InterlockedCompareExchangePointer** 函数来查看 **anchor** 是否仍指向 **first**，即使在过了几纳秒之后。如果是这样，**InterlockedCompareExchangePointer** 将设置 **anchor**，使其指向新元素 **p**。并且如果 **InterlockedCompareExchangePointer** 的返回值也与我们的假设一致，则循环终止。如果由于某种原因，**anchor** 不再指向那个 **first** 元素(可能被其它并发线程或 CPU 修改过)，我们将发现这个事实并重复循环。

最后一个函数是 **InterlockedExchange**，它使用原子操作替换整数变量的值并返回该变量的历史值：

```

LONG value;
LONG oldval = InterlockedExchange(&value, newval);

```

正如你猜到的，还有一个 **InterlockedExchangePointer** 函数，它交换指针值(64 位或 32 位，取决于具体平台)。

ExInterlockedXxx函数

每一个 **ExInterlockedXxx** 函数都需要在调用前创建并初始化一个自旋锁。注意，这些函数的操作数必须存在于非分页内存中，因为这些函数在提升的 **IRQL** 上操作数据。

ExInterlockedAddLargeInteger 加两个 64 位整数并返回被加数的历史值：

```

LARGE_INTEGER value, increment;
KSPIN_LOCK spinlock;
LARGE_INTEGER prev = ExInterlockedAddLargeInteger(&value, increment, &spinlock);

```

value 是被加数。**increment** 是加数。**spinlock** 是一个已经初始化过的自旋锁。返回值是被加数的历史值。该函数的操作过程与下面代码类似，但除了自旋锁的保护：

```

__int64 AddLargeInteger(__int64* pvalue, __int64 increment)
{
    __int64 prev = *pvalue;
    *pvalue += increment;
    return prev;
}

```

注意，并不是所有编译器都支持 **__int64** 整型类型，并且不是所有计算机都能用原子指令方式执行 64 位加操作。

ExInterlockedAddUlong 与 **ExInterlockedAddLargeInteger** 类似，但它的操作数是 32 位无符号整数：

```

ULONG value, increment;
KSPIN_LOCK spinlock;
ULONG prev = ExInterlockedAddUlong(&value, increment, &spinlock);

```

该函数同样返回被加数的加前值。

ExInterlockedAddLargeStatistic 与 **ExInterlockedAddUlong** 类似，但它把 32 位值加到 64 位值上。该函数在本书出版时还没有在 DDK 中公开，所以我在这里仅给出它的原型：

```
VOID ExInterlockedAddLargeStatistic(PLARGE_INTEGER Addend, ULONG Increment);
```

该函数要比 **ExInterlockedAddUlong** 函数快，因为它不需要返回被加数的加前值。因此，它也不需要使用自旋锁来同步。该函数的操作也是原子性的，但仅限于调用同一函数的其它调用者。换句话说，如果你在一个 CPU 上调用 **ExInterlockedAddLargeStatistic** 函数，而同时另一个 CPU 上的代码正访问 **Addend** 变量，那么你将得到不一致的结果。我将用该函数在 Intel x86 上的执行代码(并不是实际的源代码)来解释这个原因：

```
mov eax, Addend  
mov ecx, Increment  
lock add [eax], ecx  
lock adc [eax+4], 0
```

这个代码在低 32 位没有进位的情况下可以正常工作，但如果存在着进位，那么在 ADD 和 ADC 指令之间其它 CPU 可能进入，如果那个 CPU 调用的 **ExInterlockedCompareExchange64** 函数复制了这个时刻的 64 位变量值，那么它得到值将是不正确的。即使每个加法指令前都有 lock 前缀保护其操作的原子性(多 CPU 之间)，但多个这样的指令组成的代码块将无法保持原子性。

链表的互锁访问

Windows NT 的 **executive** 部件提供了三组特殊的链表访问函数，它们可以提供线程安全的和多处理器安全的链表访问。这些函数支持双链表、单链表，和一种称为 **S** 链表(**S-List**)的特殊单链表。我在前面章中已经讨论过单链表和双链表的非互锁访问。在这里，我将解释这些链表的互锁访问。

如果你需要一个 **FIFO** 队列，你应该使用双链表。如果你需要一个线程安全的和多处理器安全的下推栈，你应该使用 **S** 链表。为了以线程安全和多处理器安全的方式使用这些链表，你必须为它们分配并初始化一个自旋锁。但 **S** 链表并没有真正使用自旋锁。**S** 链表中存在顺序号，内核利用它可以实现比较-交换操作的原子性。

用于互锁访问各种链表对象的函数都十分相似，所以我将以函数的功能来组织这些段。我将解释如何初始化这三种链表，如何向这三种链表中插入元素，如何从这三种链表中删除元素。

初始化

你可以象下面这样初始化这些链表：

```
LIST_ENTRY DoubleHead;  
SINGLE_LIST_ENTRY SingleHead;  
SLIST_HEADER SListHead;  
  
InitializeListHead(&DoubleHead);  
  
SingleHead.Next = NULL;  
  
ExInitializeSListHead(&SListHead);
```

不要忘记为每种链表分配并初始化一个自旋锁。另外，链表头和所有链表元素的存储都必须来自非分页内存，因为支持例程需要在提升的 IRQL 上访问这些链表。注意，在链表头的初始化过程中不需要使用自旋锁，因为此时不存在竞争。

插入元素

双链表可以在头部或尾部插入元素，但单链表和 **S** 链表仅能在头部插入元素：

```
PLIST_ENTRY pdElement, pdPrevHead, pdPrevTail;
```

```
PSINGLE_LIST_ENTRY psElement, psPrevHead;
PKSPIN_LOCK spinlock;

pdPrevHead = ExInterlockedInsertHeadList(&DoubleHead, pdElement, spinlock);
pdPrevTail = ExInterlockedInsertTailList(&DoubleHead, pdElement, spinlock);

psPrevHead = ExInterlockedPushEntryList(&SingleHead, psElement, spinlock);

psPrevHead = ExInterlockedPushEntrySList(&SListHead, psElement, spinlock);
```

返回值是插入前链表头(或尾)的地址。注意，被插入的链表元素地址是一个链表表项结构的地址，这个地址通常要嵌入到更大的应用结构中，调用 **CONTAINING_RECORD** 宏可以获得外围应用结构的地址。

删除元素

你可以从这些链表的头部删除元素：

```
pdElement = ExInterlockedRemoveHeadList(&DoubleHead, spinlock);

psElement = ExInterlockedPopEntryList(&SingleHead, spinlock);

psElement = ExInterlockedPopEntrySList(&SListHead, spinlock);
```

如果链表为空则函数的返回值为 **NULL**。你应该先测试返回值是否为 **NULL**，然后再用 **CONTAINING_RECORD** 宏取外围应用结构的指针。

IRQL的限制

你只能在低于或等于 **DISPATCH_LEVEL** 级上调用 **S** 链表函数。只要所有对链表的引用都使用 **ExInterlockedXxx** 函数，那么访问双链表和单链表的 **ExInterlockedXxx** 函数可以在任何 **IRQL** 上调用。这些函数没有 **IRQL** 限制的原因是因为它们在执行时都禁止了中断，这就等于把 **IRQL** 提升到最高可能的级别。一旦中断被禁止，这些函数就获取你指定的自旋锁。因为此时在同一 **CPU** 上没有其它代码能获得控制，并且其它 **CPU** 上的代码也不能获取那个自旋锁，所以你的链表是安全的。

注意

DDK 文档中关于这条规则的陈述过于严格，它认为所有调用者必须运行在低于或等于你的中断对象 **DIRQL** 之下的某个 **IRQL** 上。实际上，并不需要所有调用者都在同一 **IRQL** 上，同样也不必限制 **IRQL** 必须小于或等于 **DIRQL**。

最好在代码的一个部分使用 **ExInterlockedXxx** 互锁函数访问单链表或双链表(不包括 **S** 链表)，在另一部分使用非互锁函数(**InsertHeadList** 等等)。在使用一个非互锁原语前，应该提前获取调用使用的自旋锁。另外，应该限制低于或等于 **DISPATCH_LEVEL** 级的代码访问链表。例如：

```
// Access list using noninterlocked calls:

VOID Function1()
{
    ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
    KIRQL oldirql;
    KeAcquireSpinLock(spinlock, &oldirql);
    InsertHeadList(...);
    RemoveTailList(...);
    ...
}
```

```
    KeReleaseSpinLock(spinlock, oldirql);
}

// Access list using interlocked calls:

VOID Function2()
{
    ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
    ExInterlockedInsertTailList(..., spinlock);
}
```

第一个函数必须运行在低于或等于 `DISPATCH_LEVEL` 上，因为这里需要调用 `KeAcquireSpinLock` 函数。第二个函数的 IRQL 限定原因是这样的：假定 **Function1** 在准备访问链表阶段获取了自旋锁，而获取自旋锁时需要把 IRQL 暂时提升到 `DISPATCH_LEVEL` 级，现在再假定在同一 CPU 上有一个中断发生在更高级的 IRQL 上，然后 **Function2** 获得了控制，而它又调用了一个 `ExInterlockedXxx` 函数，而此时内核正要获取同一个自旋锁，因此 CPU 将死锁。导致这个问题的原因是允许用同一个自旋锁的代码运行在两个不同的 IRQL 上：**Function1** 在 `DISPATCH_LEVEL` 级上，而 **Function2** 在 `HIGH_LEVEL` 级上。

共享数据的非互锁访问

如果你要提取一个对齐的数据，那么调用任何一个 `InterlockedXxx` 函数就可以正确地做到。支持 NT 的 CPU 必然保证你能获得一个首尾一致的值，即使互锁操作发生在数据被提取前后的短暂时间内。然而，如果数据没有对齐，当前的互锁访问也会禁止其它的互锁访问，不至于造成并发访问而取到不一致的值。想象一下，如果有 一个整数，其大小跨过了物理内存中的缓冲边界，此时，CPU A 想提取这个整数，而 CPU B 在同一时间要在这个值上执行一个互锁加 1 操作。那么即将发生的一系列事情可能是：(a) CPU A 提取了含有该值高位部分的缓冲线，(b) CPU B 执行了一个互锁增 1 操作并向该值高位部分产生了一个进位，(c) CPU A 接着提取了包含该值低位部分的缓冲线。确保这个值不跨过一个缓冲界限可以避免这个问题，但最容易的解决办法是确保该值按其数据类型的自然边界对齐，如 `ULONG` 类型按 4 字节对齐。

第五章：I/O请求包

操作系统使用一种称为 I/O 请求包(IRP)的数据结构与内核模式驱动程序通信。在这一章中，我们将讨论这个重要的数据结构，包括它的创建、发送、处理，以及最后的销毁。在本章的最后，我将讨论相对复杂的 IRP 取消操作。本章的内容相对深奥一些，因为我事先没有谈到 IRP 的任何相关概念。因此，你完全可以先跳过这一章直接读后面的章节，然后再回过头来阅读本章。

- 数据结构
- IRP 处理的“标准模型”
- 完成 I/O 请求
- 向下级传递请求
- 取消 I/O 请求
- 管理自己的 IRP
- 松散的结尾

数据结构

有两个数据结构对 I/O 请求的处理至关重要：I/O 请求包(IRP)本身和 IO_STACK_LOCATION 结构。下面我将详细描述这两个结构。

IRP 结构

图 5-1 显示了 IRP 的数据结构，阴影部分代表不透明域。下面是该结构中重要域的简要描述。

MdlAddress(PMDL) 域指向一个内存描述符表(MDL)，该表描述了一个与该请求关联的用户模式缓冲区。如果顶级设备对象的 Flags 域为 DO_DIRECT_IO，则 I/O 管理器为 IRP_MJ_READ 或 IRP_MJ_WRITE 请求创建这个 MDL。如果一个 IRP_MJ_DEVICE_CONTROL 请求的控制代码指定 METHOD_IN_DIRECT 或 METHOD_OUT_DIRECT 操作方式，则 I/O 管理器为该请求使用的输出缓冲区创建一个 MDL。MDL 本身用于描述用户模式虚拟缓冲区，但它同时也含有该缓冲区锁定内存页的物理地址。为了访问用户模式缓冲区，驱动程序必须做一点额外工作。

Type	Size
MdlAddress	
Flags	
AssociateIrp	
ThreadListEntry	
IoStatus	
RequestorMode	PendingReturned
Cancel	CancellingIql
StackCount	
ApcEnvironment	
AllocationFlags	
UserIosb	
UserEvent	
Overlay	
CancelRoutine	
UserBuffer	
Tail	

图 5-1. I/O 请求包数据结构

Flags(ULONG) 域包含一些对驱动程序只读的标志。但这些标志与 WDM 驱动程序无关。

AssociatedIrp(union) 域是一个三指针联合。其中，与 WDM 驱动程序相关的指针是 **AssociatedIrp.SystemBuffer**。SystemBuffer 指针指向一个数据缓冲区，该缓冲区位于内核模式的非分页内存中。对于 IRP_MJ_READ 和 IRP_MJ_WRITE 操作，如果顶级设备指定 DO_BUFFERED_IO 标志，则 I/O 管理器就创建这个数据缓冲区。对于 IRP_MJ_DEVICE_CONTROL 操作，如果 I/O 控制功能代码指出需要缓冲区（见第九章），则 I/O 管理器就创建这个数据缓冲区。I/O 管理器把用户模式程序发送给驱动程序的数据复制到这个缓冲区，这也是创建 IRP 过程的一部分。这些数据可以是与 WriteFile 调用有关的数据，或者是 **DeviceIoControl** 调用中所谓的输入数据。对于读请求，设备驱动程序把读出的数据填到这个缓冲区，然后 I/O 管理器再把缓冲区的内容复制到用户模式缓冲区。对于指定了 METHOD_BUFFERED 的 I/O 控制操作，驱动程序把所谓的输出数据放到这个缓冲区，然后 I/O 管理器再把数据复制到用户模式的输出缓冲区。

IoStatus(IO_STATUS_BLOCK) 是一个仅包含两个域的结构，驱动程序在最终完成请求时设置这个结构。

IoStatus.Status 域将收到一个 NTSTATUS 代码，而 **IoStatus.Information** 的类型为 ULONG_PTR，它将收到一个信息值，该信息值的确切含义要取决于具体的 IRP 类型和请求完成的状态。Information 域的一个公认用法是用于保存数据传输操作，如 IRP_MJ_READ，的流量总计。某些 PnP 请求把这个域作为指向另外一个结构的指针，这个结构通常包含查询请求的结果。

RequestorMode 将等于一个枚举常量 **UserMode** 或 **KernelMode**，指定原始 I/O 请求的来源。驱动程序有时需要查看这个值来决定是否要信任某些参数。

PendingReturned(BOOLEAN)如果为 TRUE，则表明处理该 IRP 的最低级派遣例程返回了 STATUS_PENDING。完成例程通过参考该域来避免自己与派遣例程间的潜在竞争。

Cancel(BOOLEAN)如果为 TRUE，则表明 **IoCancelIrp** 已被调用，该函数用于取消这个请求。如果为 FALSE，则表明没有调用 **IoCancelIrp** 函数。取消 IRP 是一个相对复杂的主题，我将在本章的最后详细描述它。

CancellableIrql(KIRQL)是一个 IRQL 值，表明那个专用的取消自旋锁是在这个 IRQL 上获取的。当你在取消例程中释放自旋锁时应参考这个域。

CancelRoutine(PDRIVER_CANCEL)是驱动程序取消例程的地址。你应该使用 **IoSetCancelRoutine** 函数设置这个域而不是直接修改该域。

UserBuffer(PVOID) 对于 METHOD_NEITHER 方式的 IRP_MJ_DEVICE_CONTROL 请求，该域包含输出缓冲区的用户模式虚拟地址。该域还用于保存读写请求缓冲区的用户模式虚拟地址，但指定了 DO_BUFFERED_IO 或 DO_DIRECT_IO 标志的驱动程序，其读写例程通常不需要访问这个域。当处理一个 METHOD_NEITHER 控制操作时，驱动程序能用这个地址创建自己的 MDL。

Tail.Overlay 是 **Tail** 联合中的一种结构，它含有几个对 WDM 驱动程序有潜在用途的成员。图 5-2 是 **Tail** 联合的组成图。在这个图中，以水平方向从左到右是这个联合的三个可选成员，在垂直方向是每个结构的成员描述。**Tail.Overlay.DeviceQueueEntry**(KDEVICE_QUEUE_ENTRY) 和 **Tail.Overlay.DriverContext**(PVOID[4]) 是 **Tail.Overlay** 内一个未命名联合的两个可选成员(只能出现一个)。I/O 管理器把 DeviceQueueEntry 作为设备标准请求队列中的连接域。当 IRP 还没有进入某个队列时，如果你拥有这个 IRP 你可以使用这个域，你可以任意使用 DriverContext 中的四个指针。**Tail.Overlay.ListEntry**(LIST_ENTRY) 仅能作为你自己实现的私有队列的连接域。

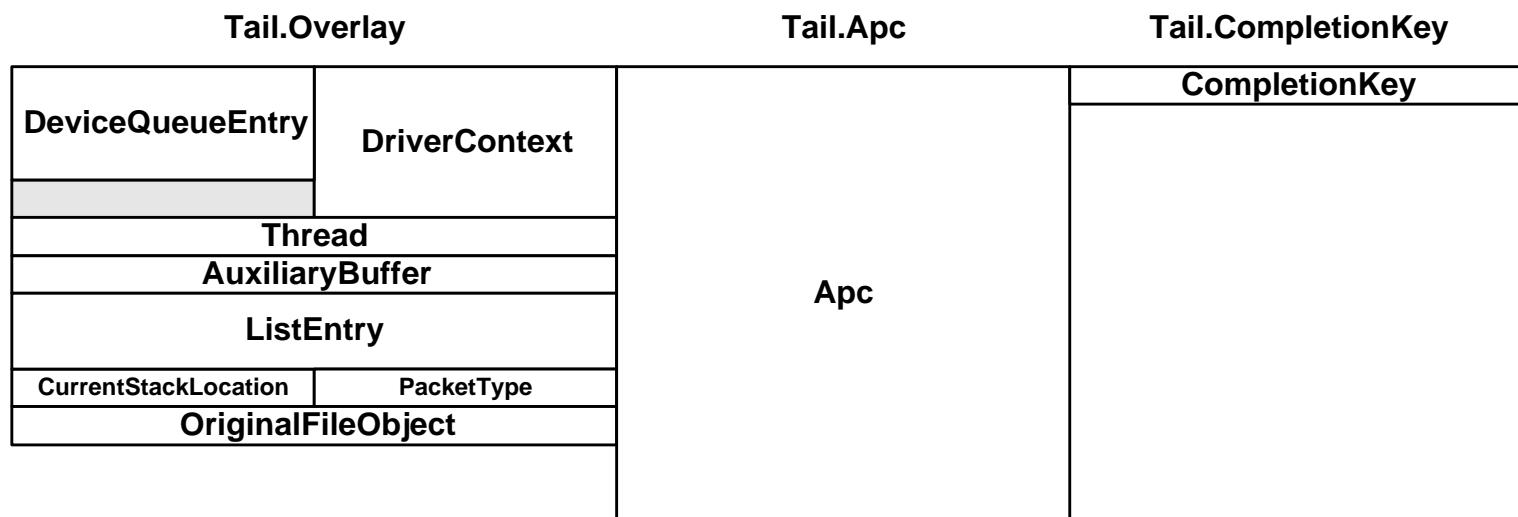


图 5-2. IRP 中 Tail 联合的组成图

CurrentLocation (CHAR) 和 **Tail.Overlay.CurrentStackLocation**(PIO_STACK_LOCATION) 没有公开为驱动程序使用，因为你完全可以使用象 **IoGetCurrentIrpStackLocation** 这样的函数获取这些信息。但意识到 CurrentLocation 就是当前 I/O 堆栈单元的索引以及 CurrentStackLocation 就是指向它的指针，会对驱动程序调试有一些帮助。

I/O堆栈

任何内核模式程序在创建一个 IRP 时，同时还创建了一个与之关联的 IO_STACK_LOCATION 结构数组：数组中的每个堆栈单元都对应一个将处理该 IRP 的驱动程序，另外还有一个堆栈单元供 IRP 的创建者使用(见图 5-3)。堆栈单元中包含该 IRP 的类型代码和参数信息以及完成函数的地址。图 5-4 显示了堆栈单元的结构。

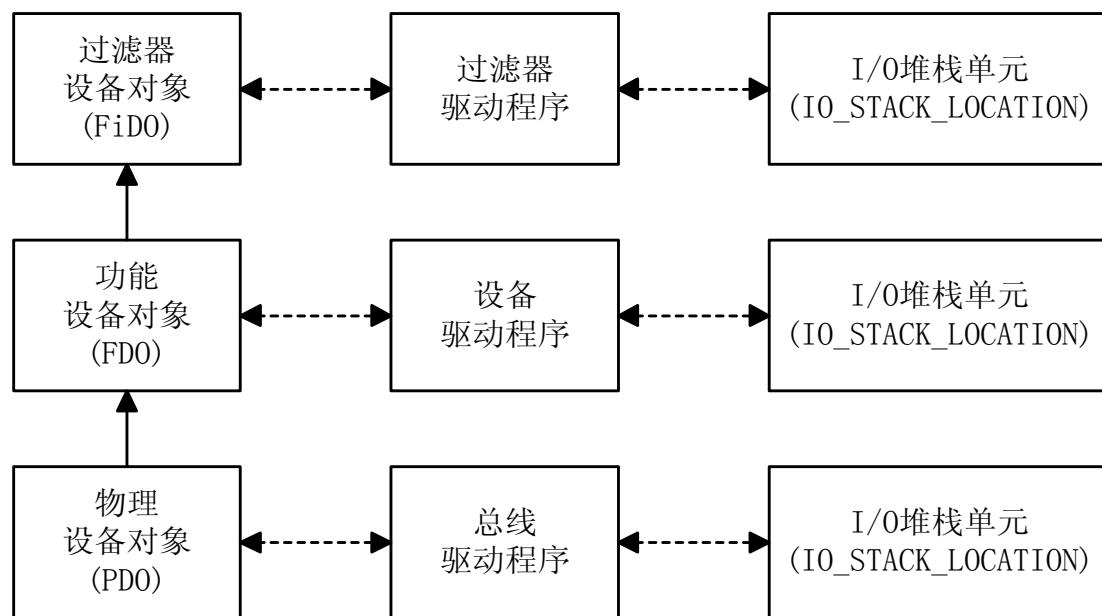


图 5-3. 驱动程序和 I/O 堆栈之间的平行关系

注意

我将在本章稍后处讨论 IRP 的创建机制。将帮助你了解 DEVICE_OBJECT 的 **StackSize** 域，该域指出 IRP 应为其目标设备驱动程序保留多少堆栈单元。

MajorFunction	MinorFunction	Flags	Control
Parameters			
	DeviceObject		
	FileObject		
	CompletionRoutine		
	Context		

图 5-4. I/O 堆栈单元数据结构

MajorFunction(UCHAR) 是该 IRP 的主功能码。这个代码应该为类似 IRP_MJ_READ 一样的值，并与驱动程序对象中 MajorFunction 表的某个派遣函数指针相对应。如果该代码存在于某个特殊驱动程序的 I/O 堆栈单元中，它有可能一开始是，例如 IRP_MJ_READ，而后被驱动程序转换成其它代码，并沿着驱动程序堆栈发送到低层驱动程序。我将在第十一章(USB 总线)中举一个这样的例子，USB 驱动程序把标准的读或写请求转换成内部控制操作，以便向 USB 总线驱动程序提交请求。

MinorFunction(UCHAR) 是该 IRP 的副功能码。它进一步指出该 IRP 属于哪个主功能类。例如，IRP_MJ_PNP 请求就有约一打的副功能码，如 IRP_MN_START_DEVICE、IRP_MN_REMOVE_DEVICE，等等。

Parameters(union) 是几个子结构的联合，每个请求类型都有自己专用的参数，而每个子结构就是一种参数。这些子结构包括 **Create**(IRP_MJ_CREATE 请求)、**Read**(IRP_MJ_READ 请求)、**StartDevice**(IRP_MJ_PNP 的 IRP_MN_START_DEVICE 子类型)，等等。

DeviceObject(PDEVICE_OBJECT) 是与该堆栈单元对应的设备对象的地址。该域由 **IoCallDriver** 函数负责填写。

FileObject(PFILE_OBJECT) 是内核文件对象的地址，IRP 的目标就是这个文件对象。驱动程序通常在处理清除请求(IRP_MJ_CLEANUP)时使用 FileObject 指针，以区分队列中与该文件对象无关的 IRP。

CompletionRoutine(PIO_COMPLETION_ROUTINE)是一个 I/O 完成例程的地址，该地址是由与这个堆栈单元对应的驱动程序的更上一层驱动程序设置的。你绝对不要直接设置这个域，应该调用 **IoSetCompletionRoutine** 函数，该函数知道如何参考下一层驱动程序的堆栈单元。设备堆栈的最低一级驱动程序并不需要完成例程，因为它们必须直接完成请求。然而，请求的发起者有时确实需要一个完成例程，但通常没有自己的堆栈单元。这就是为什么每一级驱动程序都使用下一级驱动程序的堆栈单元保存自己完成例程指针的原因。

Context(PVOID)是一个任意的与上下文相关的值，将作为参数传递给完成例程。你绝对不要直接设置该域；它由 **IoSetCompletionRoutine** 函数自动设置，其值来自该函数的某个参数。

IRP处理的“标准模型”

粒子物理学里有关于宇宙的“标准模型”，WDM 也是这样。图 5-5 显示了一个典型的 IRP 在各个处理阶段的所有权流程。并不是每种 IRP 都经过这些步骤，由于设备类型和 IRP 种类的不同某些步骤会改变或根本不存在。尽管这个过程可能有各种变化形式，但这个图为我们将要展开的讨论提供了一个很好的起点。

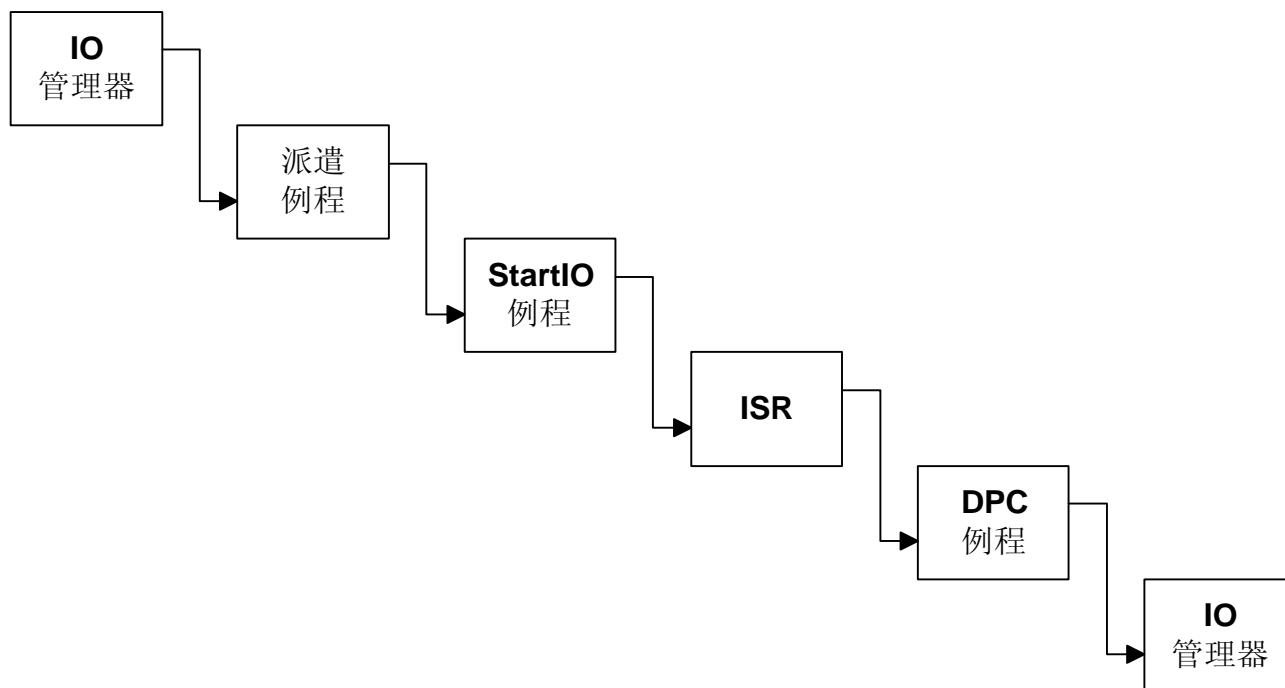


图 5-5. IRP 处理的“标准模型”

比你想象的更复杂...

当你第一次遇到 IRP 处理标准模型这个概念时，你也许认为这是个比较复杂的概念。但不幸的是，这个概念还不能满足所有问题，比如热拔插设备、动态资源再分配，和电源管理等等。在后面的章节中，我将描述处理这些额外问题的其它 IRP 排队和取消方法。标准模型仅作为你阅读时清晰的参考模型！

抛开这些特殊问题，许多设备仍能利用这个标准模型。如果你的设备在系统运行时不能被删除或重配置，并且在低电源状态下拒绝 I/O 请求，那么你可以使用这个标准模型。

创建IRP

IRP 开始于某个实体调用 I/O 管理器函数创建它。在上图中，我使用术语“I/O 管理器”来描述这个实体，尽管系统中确实有一个单独的系统部件用于创建 IRP。事实上，更精确地说，应该是某个实体创建了 IRP，并不是操作系统的某个例程创建了 IRP。例如，你的驱动程序有时会创建 IRP，而此时出现在图中第一个方框中的实体就应该是你的驱动程序。

可以使用下面任何一种函数创建 IRP：

- **IoBuildAsynchronousFsdRequest** 创建异步 IRP(不需要等待其完成)。该函数和下一个函数仅适用于创建某些类型的 IRP。
- **IoBuildSynchronousFsdRequest** 创建同步 IRP(需要等待其完成)。
- **IoBuildDeviceIoControlRequest** 创建一个同步 IRP_MJ_DEVICE_CONTROL 或 IRP_MJ_INTERNAL_DEVICE_CONTROL 请求。
- **IoAllocateIrp** 创建上面三个函数不支持的其它种类的 IRP。

前两个函数中的 **Fsd** 表明这些函数专用于文件系统驱动程序(FSD)。虽然 FSD 是这两个函数的主要使用者，但其它驱动程序也可以调用这些函数。DDK 还公开了一个 **IoMakeAssociatedIrp** 函数，该函数用于创建某些 IRP 的从属 IRP。WDM 驱动程序不应该使用这个函数。

决定该调用哪一个函数，和决定对 IRP 执行什么额外的初始化是更复杂的问题，我将在本章的结尾再回到这个问题上。

发往派遣例程

创建完 IRP 后，你可以调用 **IoGetNextIrpStackLocation** 函数获得该 IRP 第一个堆栈单元的指针。然后初始化这个堆栈单元。在初始化过程的最后，你需要填充 MajorFunction 代码。堆栈单元初始化完成后，就可以调用 **IoCallDriver** 函数把 IRP 发送到设备驱动程序：

```
PDEVICE_OBJECT DeviceObject;           //something gives you this
PIO_STACK_LOCATION stack = IoGetNextIrpStackLocation(Irp);
stack->MajorFunction = IRP_MJ_Xxx;
<other initialization of "stack">
NTSTATUS status = IoCallDriver(DeviceObject, Irp);
```

IoCallDriver 函数的第一个参数是你在某处获得的设备对象的地址。我将在本章的结尾处描述获得设备对象指针的两个常用方法。在这里，我们先假设你已经有了这个指针。

IRP 中的第一个堆栈单元指针被初始化成指向该堆栈单元之前的堆栈单元，因为 I/O 堆栈实际上是 **IO_STACK_LOCATION** 结构数组，你可以认为这个指针被初始化为指向一个不存在的“-1”元素，因此当我们要初始化第一个堆栈单元时我们实际需要的是“下一个”堆栈单元。**IoCallDriver** 将沿着这个堆栈指针找到第 0 个表项，并提取我们放在那里的主功能代码，在上例中为 **IRP_MJ_Xxx**。然后 **IoCallDriver** 函数将利用 **DriverObject** 指针找到设备对象中的 **MajorFunction** 表。**IoCallDriver** 将使用主功能代码索引这个表，最后调用找到的地址(派遣函数)。

你可以把 **IoCallDriver** 函数想象为下面代码：

```
NTSTATUS IoCallDriver(PDEVICE_OBJECT device, PIRP Irp)
{
    IoSetNextIrpStackLocation(Irp);
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    stack->DeviceObject = device;
    ULONG fcn = stack->MajorFunction;
    PDRIVER_OBJECT driver = device->DriverObject;
    return (*driver->MajorFunction[fcn])(device, Irp);
}
```

派遣例程的职责

IRP 派遣例程的原型看起来像下面这样：

```
NTSTATUS DispatchXxx(PDEVICE_OBJECT device, PIRP Irp)
{
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);           <--1
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) device->DeviceExtension;
    ...
    return STATUS_Xxx;
}
```

1. 你通常需要访问当前堆栈单元以确定参数或副功能码。

2. 你可能还需要访问你创建的设备扩展。
3. 你将向 `IoCallDriver` 函数返回某个 `NTSTATUS` 代码，而 `IoCallDriver` 函数将把这个状态码返回给它的调用者。

在本书中，我使用 `DispatchXxx`(如 `DispatchRead`、`DispatchPnp`，等等)来代表例子驱动程序中的派遣例程。其它人可能会使用另外的约定，但 Microsoft 推荐用这样的方法，例如，如果你的驱动程序名为 `RANDOM.SYS`，那么你应该命名 `IRP_MJ_READ` 派遣函数为 `RandomDispatchRead`。这个方法使驱动程序调试跟踪起来更容易，但它同时也需要你输入更多的文字。由于这些名称在驱动程序的名空间之外是不可见的，所以由你自己决定是使用 Microsoft 推荐的命名方案，还是使用你认为更有意义的命名方法。

在上面派遣函数原型中省略号的地方，是派遣函数必须做出决定的地方，有三种选择：

- 派遣函数立即完成该 `IRP`。
- 把该 `IRP` 传递到处于同一堆栈的下层驱动程序。
- 排队该 `IRP` 以便由这个驱动程序中的其它例程来处理。

我将在本章详细讨论这三种选择，但在这里我仅讨论排队的可能性，因为这个过程就是 `IRP` 处理标准模型所描述的。你知道，当有大量读写请求进入设备时，通常需要把这些请求放入一个队列中，以便使硬件访问串行化。

每个设备对象都自带一个请求队列对象，下面是使用这个队列的标准方法：

```
NTSTATUS DispatchXxx(...)
{
    ...
    IoMarkIrpPending(Irp);                                <--1
    IoStartPacket(device, Irp, NULL, NULL);               <--2
    return STATUS_PENDING;                                <--3
}
```

1. 无论何时，当你的派遣例程返回 `STATUS_PENDING` 状态代码时，你应该先调用这个 `IoMarkIrpPending` 函数，以帮助 I/O 管理器避免内部竞争。我们必须在放弃 `IRP` 所有权之前做这一点。
2. 如果设备正忙，`IoStartPacket` 就把请求放到队列中。如果设备空闲，`IoStartPacket` 将把设备置成忙并调用 `StartIo` 例程。`IoStartPacket` 的第三个参数是用于排序队列的键(`ULONG`)的地址，例如磁盘驱动程序将在这里指定一个柱面地址以提供顺序搜索的排队。如过你在这里指定一个 `NULL`，则该请求被加到队列的尾部。最后一个参数是取消例程的地址。我将在本章的后面讨论取消例程，这种例程比较复杂。
3. 返回 `STATUS_PENDING` 以通知调用者我们没有完成这个 `IRP`。

注意，一旦我们调用了 `IoStartPacket` 函数，就不要再碰 `IRP`。因为在该函数返回之前，`IRP` 可能已经被完成并且其占用的内存可能被释放，而我们拥有的该 `IRP` 的指针也许是无效的。

StartIo 例程

每处理一个 `IRP`，I/O 管理器就调用一次 `StartIo` 例程：

```
VOID StartIo(PDEVICE_OBJECT device, PIRP Irp)
{
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) device->DeviceExtension;
    ...
}
```

`StartIo` 例程在 `DISPATCH_LEVEL` 级上获得控制，这意味着该函数不能生成任何页故障。另外，设备对象的 `CurrentIrp` 域和 `Irp` 参数都指向 I/O 管理器送来的 `IRP`。

`StartIo` 的工作是就着手处理 `IRP`。如何做要完全取决于你的设备。通常你需要访问硬件寄存器，但可能有其它例程，如你的中断服务例程，或者是驱动程序中的其它例程也需要访问这些寄存器。实际上，有时着手一个新操作的最容易的方式是在设备扩展中保存某些状态信息，然后伪造一个中断。由于这些方法的执行都需要在一

个自旋锁的保护之下，而这个自旋锁与保护你的 ISR 所使用的是同一个自旋锁，所以正确的方法是调用 **KeSynchronizeExecution** 函数。例如：

```
VOID StartIo(...)  
{  
    ...  
    KeSynchronizeExecution(pdx->InterruptObject, TransferFirst, (PVOID) pdx);  
}  
  
BOOLEAN TransferFirst(PVOID context)  
{  
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) context;  
    ...  
    return TRUE;  
}
```

这里的 **TransferFirst** 例程是同步关键段(SynchCriticalSection)的一个例子，之所以这样做是因为 **StartIo** 需要与 ISR 同步。我将在第七章中详细讨论同步关键段(SynchCriticalSection)的概念。

一旦 **StartIo** 使设备忙于处理新请求，它就立即返回。当设备完成传输并发出中断时你将看到下一个请求。

中断服务例程

当设备完成数据传输后，它将以硬件中断形式发出通知。在第七章中，我将讲述如何用 **IoConnectInterrupt** 函数“钩住”一个中断，该函数的一个参数就是 ISR 的地址。因此当中断发生时，硬件抽象层(HAL)就调用你的 ISR。ISR 运行在 DIRQL 上，并由 ISR 专用的自旋锁保护。ISR 的函数原型如下：

```
BOOLEAN OnInterrupt(PKINTERRUPT InterruptObject, PVOID context)  
{  
    ...  
}
```

ISR 的第一个参数是中断对象的地址，中断对象由 **IoConnectInterrupt** 函数创建，但是你不太可能用到这个参数。第二个参数是在调用 **IoConnectInterrupt** 时你指定的任意上下文值；它可能是设备对象或设备扩展的地址，完全由你决定。

我将在第七章中详细讨论 ISR 的职责。为了继续标准模型的讨论，我要告诉你一点，一个 ISR 最可能做的事就是调度 DPC 例程(推迟过程调用)。而 DPC 的目的就是让你做某些事情，如调用 **IoCompleteRequest**，而该调用不可能运行在 ISR 运行的 DIRQL 级上。所以，你的 ISR 中将有下面一行语句(**device** 是指向设备对象的指针)：

```
IoRequestDpc(device, device->CurrentIrp, NULL);
```

那么下一次你将在 DPC 例程中看到这个 IRP，这个 DPC 例程是你在 **AddDevice** 函数中用 **IoInitializeDpcRequest** 寄存的。DPC 例程的传统名称为 **DpcForIsr**，因为它是由 ISR 请求的。

DPC 例程

DpcForIsr 例程在 DISPATCH_LEVEL 级上获得控制。通常，它的工作就是完成 IRP(导致最近的中断发生)。但一般情况下，它通过调用 **IoCompleteRequest** 函数把剩余的工作交给完成例程来做。

```
VOID DpcForIsr(PKDPC Dpc, PDEVICE_OBJECT device, PIRP Irp, PDEVICE_EXTENSION pdx)  
{  
    ...  
    IoStartNextPacket(device, FALSE);
```

<-1

```
IoCompleteRequest(Irp, boost); <--2  
}
```

1. **IoStartNextPacket** 取出设备队列中的下一个 IRP 并发送到 StartIo。FALSE 参数指出该 IRP 不能以通常方式取消。
2. **IoCompleteRequest** 完成第一个参数指定的 IRP。第二参数是等待线程的优先级提高值。注意在调用 **IoCompleteRequest** 之前你还要填充 IRP 中的 **IoStatus** 块。

调用 **IoCompleteRequest** 例程是处理 I/O 请求的标准结束方式。在这个调用之后, I/O 管理器(或者是任何在开始处创建该 IRP 的实体)将再次拥有该 IRP。最后该 IRP 被这个实体销毁并解除等待线程的阻塞状态。

定制队列

有些设备的操作需要多个请求队列。一个常见的例子就是串行口, 它可以同时地并且分开地处理输入输出请求流。**IoStartPacket** 和 **IoStartNextPacket** 函数(以及其它含有键排序功能的等价函数)都使用设备对象自带的队列。创建与标准队列有相同工作方式的附加队列要相对容易一些。

为了使我们更容易讨论问题, 让我们假设你需要一个单独的队列来管理 **IRP_MJ_SPECIAL**(并不存在这个主功能码, 使用它是为了使问题更具体一些)请求。你将写两个与 **StartIo** 和 **DpcForIsr** 例程功能类似的, 但专用于处理这些假想 IRP 的辅助例程:

- 与 **StartIo** 类似的函数 --- 我们称它为 **StartIoSpecial** --- 它启动下一个 **IRP_MJ_SPECIAL** 请求。
- 与 DPC 类似的函数 --- 我们称它为 **DpcSpecial** --- 它处理 **IRP_MJ_SPECIAL** 请求的完成。

你还需要在你的设备扩展中创建一个 **KDEVICE_QUEUE** 对象, 并在 **AddDevice** 例程中初始化这个队列对象:

```
NTSTATUS AddDevice(...)  
{  
    ...  
    KeInitializeDeviceQueue(&pdx->dqSpecial);  
    ...  
}
```

dqSpecial 就是 **KDEVICE_OBJECT** 对象的名字, 用于排队 **IRP_MJ_SPECIAL** 请求。设备队列对象是一种三态对象(见图 5-6)。这三种状态反映了设备队列例程是如何操作设备队列的:

- **idle** 状态是指设备不忙于处理任何请求并且队列为空。 **KeInsertDeviceQueue** 或 **KeInsertByKeyDeviceQueue** 函数把队列标记为 **busy-empty** 状态并返回 **FALSE**。你不应该在队列为 **idle** 状态时调用 **KeRemoveDeviceQueue** 或 **KeRemoveByKeyDeviceQueue** 函数。
- **busy-empty** 状态是指设备忙但队列中没有 IRP。**KeInsertDeviceQueue** 和 **KeInsertByKeyDeviceQueue** 函数向队列尾加入新 IRP, 使队列进入 **busy-not empty** 状态, 并返回 **TRUE**。**KeRemoveDeviceQueue** 或 **KeRemoveByKeyDeviceQueue** 函数返回 **NULL** 并使队列进入 **idle** 状态。
- **busy-not empty** 状态是指设备忙且队列中至少存有一个 IRP。**KeInsertDeviceQueue** 和 **KeInsertByKeyDeviceQueue** 函数向队列尾加入新 IRP 并返回 **TRUE**, 但队列状态不变。**KeRemoveDeviceQueue** 或 **KeRemoveByKeyDeviceQueue** 函数提取队列的第一个 IRP 并返回其地址, 此时, 如果队列为空, 这些函数将把队列置成 **busy-empty** 状态。

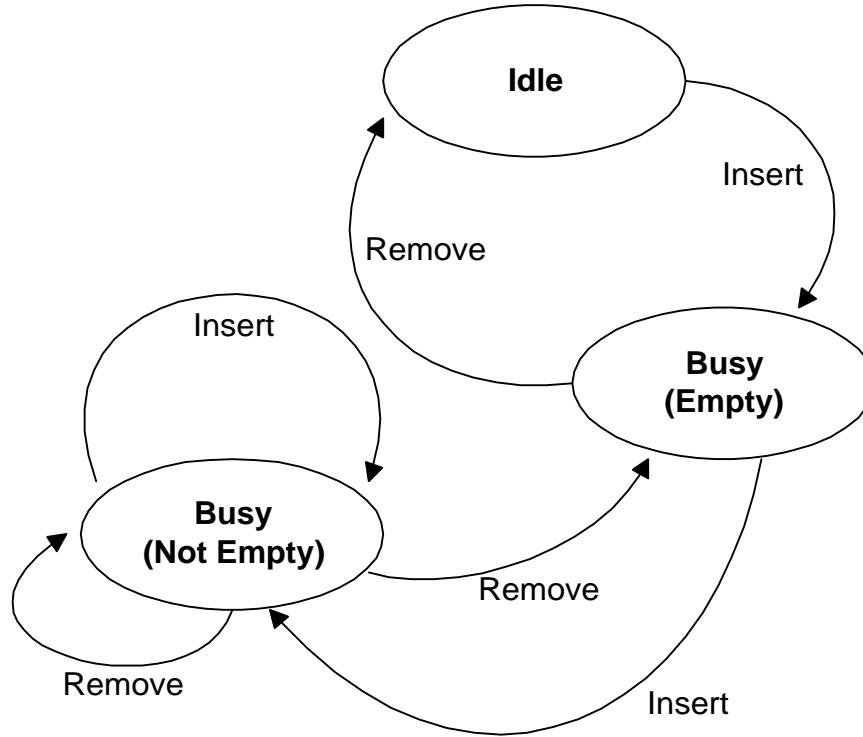


图 5-6. KDEVICE_QUEUE 队列的三种状态

在下面代码中，我们在派遣例程和 DPC 例程中使用了这些支持例程和我们专用的设备队列：

```

NTSTATUS DispatchSpecial(PDEVICE_OBJECT fdo, PIRP Irp)
{
    IoMarkIrpPending(Irp);                                     <--1
    KIRQL oldirql;
    KeRaiseIrql(DISPATCH_LEVEL, &oldirql);
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    if (!KeInsertDeviceQueue(&pdx->dqSpecial, &Irp->Tail.Overlay.DeviceQueueEntry)) <--2
        StartIoSpecial(fdo, Irp);
    KeLowerIrql(oldirql);
    return STATUS_PENDING;
}

VOID DpcSpecial(...)
{
    ...
    PKDEVICE_QUEUE_ENTRY qep = KeRemoveDeviceQueue(&pdx->dqSpecial);
    if (qep)
        StartIoSpecial(fdo, CONTAINING_RECORD(qep, IRP, Tail.Overlay.DeviceQueueEntry));
    ...
}

```

- 作为一个“规矩”的派遣例程，我们把该 IRP 标记为 pending，因为我们要让它进入队列，然后派遣例程返回 STATUS_PENDING 状态。
- KeInsertDeviceQueue 函数和我们自己的 StartIoSpecial 例程希望在 DISPATCH_LEVEL 级上被调用。所以我们明确地提升了 IRQL，之后很快我们又调用 KeLowerIrql 函数把 IRQL 降低到原来的级别(可能是 PASSIVE_LEVEL)。
- KeInsertDeviceQueue 函数把 IRP 加入到专用队列中，如果返回值为 TRUE，表明 IRP 已被加入队列中，所以我们不用再做任何关于 IRP 的事。如果设备正空闲，那么返回值应该为 FALSE 并且 IRP 也用不着排入队列，我们直接调用 StartIoSpecial 例程。
- DPC 中的这个 KeRemoveDeviceQueue 调用将产生两种结果。如果队列当前为空，则返回值为 NULL 并且我们也不用启动新请求。否则，返回值将为某 IRP 内嵌连接域的地址。我们可以使用 CONTAINING_RECORD 宏来获得外围的真正的 IRP 地址。然后我们把这个地址传递给 StartIoSpecial 例程。注意，这个 DPC 例程已经运行在 DISPATCH_LEVEL 级上，所以我们不需要在删除队列表项或调用 StartIo 之前调整 IRQL。

我以前描述的 `StartPacket` 和 `StartNextPacket` 函数使用一个名为 **DeviceQueue** 的 `KDEVICE_QUEUE` 对象。该对象是设备对象中的一个不透明域，其工作原理与管理私有设备队列相同。

完成I/O请求

每个 IRP 都渴望被完成。在标准模型中，你至少有两种完成 IRP 的环境。DpcForIsr 通常用于完成导致最近中断的 IRP。派遣函数也可以在下面这两种情况下完成 IRP：

- 如果请求是错误的(可以以容易的检测方式查明，例如要求打印机倒纸请求或卸载键盘请求)，则派遣例程应以失败方式完成该请求并返回适当的出错代码。
- 如果请求要求得到的仅是派遣函数可以容易确定的信息(例如一个询问驱动程序版本号的控制请求)，则派遣例程应立即给出回答并完成请求，返回成功状态码。

完成机制

完成一个 IRP 必须先填充 **IoStatus** 块的 **Status** 和 **Information** 成员，然后调用 **IoCompleteRequest** 例程。**Status** 值就是 **NTSTATUS.H** 中定义的状态代码。表 5-1 简要地列出了常用的状态代码。而 **Information** 值要取决于你完成的是何种类型的 IRP 以及是成功还是失败。通常情况下，如果 IRP 完成失败(即，完成的结果是某种错误状态)，你应把 **Information** 域置 0。如果你成功地完成了一个数据传输 IRP，通常应该把 **Information** 域设置成传输的字节量。

表 5-1. 一些常用的 **NTSTATUS** 代码

状态代码	描述
STATUS_SUCCESS	正常完成
STATUS_UNSUCCESSFUL	请求失败，没有描述失败原因的代码
STATUS_NOT_IMPLEMENTED	一个没有实现的功能
STATUS_INVALID_HANDLE	提供给该操作的句柄无效
STATUS_INVALID_PARAMETER	参数错误
STATUS_INVALID_DEVICE_REQUEST	该请求对这个设备无效
STATUS_END_OF_FILE	到达文件尾
STATUS_DELETE_PENDING	设备正处于被从系统中删除过程中
STATUS_INSUFFICIENT_RESOURCES	没有足够的系统资源(通常是内存)来执行该操作

注意

在设置 IRP 的 **IoStatus.Information** 域时，我们应该首先参考 DDK 文档中的正确设置。例如，在某些 **IRP_MJ_PNP** 中，这个域用于指向一个数据结构，而该数据结构由 PnP 管理器负责释放。如果你在处理这些 **IRP_MJ_PNP** 请求失败后把 **Information** 域置成 0，将造成资源丢失。

通常你常做的工作就是完成某个请求，所以我建议你编制一个辅助函数：

```
NTSTATUS CompleteRequest(PIRP Irp, NTSTATUS status, ULONG_PTR Information)
{
    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = Information;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}
```

该函数将返回其第二个参数给出的状态值。该函数适用于需要完成一个请求并立即返回状态码的场合。例如：

```

NTSTATUS DispatchControl(PDEVICE_OBJECT device, PIRP Irp)
{
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    ULONG code = stack->Parameters.DeviceIoControl.IoControlCode;
    if (code == IOCTL_TOASTER_BOGUS)
        return CompleteRequest(Irp, STATUS_INVALID_DEVICE_REQUEST, 0);
    ...
}

```

你也许注意到了，**CompleteRequest** 函数的 **Information** 参数类型为 **ULONG_PTR**。即该参数既可以是一个 **ULONG** 也可以是一个指针。

当你调用 **IoCompleteRequest** 时，应该为等待线程提供一个优先级推进值，该值将用于提高等待该请求完成的线程的优先级。一般说来，你需要根据设备类型来选择这个推进值，表 5-2 列出了一些设备的建议值。优先级的调整提高了那些需要频繁等待 I/O 操作完成的线程的吞吐量。对于那些直接响应用户的事件，如键盘或鼠标操作，应该有一个比较大的优先级推进，以提高交互任务的表现。因此，你要仔细地选择推进值。不要绝对地在声卡驱动程序完成的每个操作后都使用 **IO_SOUND_INCREMENT** 值。例如，没有必要在获取驱动程序版本控制请求后提高线程的优先级。

表 5-2. *IoCompleteRequest* 的优先级推进值

推进值常量	优先级推进值
IO_NO_INCREMENT	0
IO_CD_ROM_INCREMENT	1
IO_DISK_INCREMENT	1
IO_KEYBOARD_INCREMENT	6
IO_MAILSLOT_INCREMENT	2
IO_MOUSE_INCREMENT	6
IO_NAMED_PIPE_INCREMENT	2
IO_NETWORK_INCREMENT	2
IO_PARALLEL_INCREMENT	1
IO_SERIAL_INCREMENT	2
IO_SOUND_INCREMENT	8
IO_VIDEO_INCREMENT	1

顺便说一下，不要以专用状态代码 **STATUS_PENDING** 来完成一个 IRP。派遣例程经常要使用 **STATUS_PENDING** 代码作为返回值，但你决不能在 **IoStatus.Status** 中设置这个值。所以，在 checked 版本的 **IoCompleteRequest** 函数中有一个 **ASSERT** 语句用于检查该函数的最终返回值是否为 **STATUS_PENDING**。另一个常犯的错误是在返回值中使用“-1”，该值作为 NTSTATUS 代码没有任何意义，所以 **IoCompleteRequest** 函数中也有检查这种错误的 **ASSERT** 语句。

使用完成例程

通常，你需要知道发往低级驱动程序的 I/O 请求的结果。为了了解请求的结果，你需要安装一个完成例程，调用 **IoSetCompletionRoutine** 函数：

```

IoSetCompletionRoutine(Irp,
                      CompletionRoutine,
                      context,
                      InvokeOnSuccess,
                      InvokeOnError,
                      InvokeOnCancel);

```

Irp 就是你要了解其完成的请求。**CompletionRoutine** 是被调用的完成例程的地址，**context** 是任何一个指针长度的值，将作为完成例程的参数。**InvokeOnXxx** 参数是布尔值，它们指出在三种不同的环境中是否需要调用完成例程：

- **InvokeOnSuccess** 你希望完成例程在 IRP 以成功状态(返回的状态代码通过了 NT_SUCCESS 测试)完成时被调用。
- **InvokeOnError** 你希望完成例程在 IRP 以失败状态(返回的状态代码未通过了 NT_SUCCESS 测试)完成时被调用。
- **InvokeOnCancel** 如果驱动程序在完成 IRP 前调用了 **IoCancelIrp** 例程，你希望在此时调用完成例程。**IoCancelIrp** 将在 IRP 中设置取消标志，该标志也是调用完成例程的条件。一个被取消的 IRP 最终将以 STATUS_CANCELLED(该状态代码不能通过 NT_SUCCESS 测试)或任何其它状态完成。如果 IRP 以失败方式完成，并且你也指定了 **InvokeOnError** 参数，那么是 **InvokeOnError** 本身导致了完成例程的调用。相反，如果 IRP 以成功方式完成，并且你也指定了 **InvokeOnSuccess** 参数，那么是 **InvokeOnSuccess** 本身导致了完成例程的调用。在这两种情况中，**InvokeOnCancel** 参数将是多余的。如果你省去 **InvokeOnSuccess** 和 **InvokeOnError** 中的任何一个参数或两个都省去，并且 IRP 也被设置了取消标志，那么 **InvokeOnCancel** 参数将导致完成例程的调用。

这三个标志中至少有一个设置为 TRUE。注意，**IoSetCompletionRoutine** 是一个宏，所以你应避免使用有副作用的参数。这三个标志参数和一个函数指针参数在宏中被引用了两次。

IoSetCompletionRoutine 将把完成例程地址和上下文参数安装到下一个 **IO_STACK_LOCATION** 中，即下一层驱动程序将在那个堆栈单元中找到这些参数。因此，最底层的驱动程序不应该安装一个完成例程。

一个完成例程看起来应该像这样：

```
NTSTATUS CompletionRoutine(PDEVICE_OBJECT device, PIRP Irp, PVOID context)
{
    if (Irp->PendingReturned)
        IoMarkIrpPending(Irp);

    ...
    return <some status code>;
}
```

该函数将收到一个设备对象指针和一个 **IRP** 指针，还收到一个任意上下文值，该值在 **IoSetCompletionRoutine** 调用中指出。完成例程通常在 **DISPATCH_LEVEL** 级和任意线程上下文中被调用，但有时也在 **PASSIVE_LEVEL** 或 **APC_LEVEL** 级被调用。为了适应大多数情况(**DISPATCH_LEVEL**)，完成例程应存在于非分页内存中，并且仅使用可在 **DISPATCH_LEVEL** 级上调用的服务例程。然而，为了适应在低级 **IRQL** 上调用该例程的可能情况，完成例程不应调用像 **KeAcquireSpinLockAtDpcLevel** 这样的函数，因为这些函数假定开始执行于 **DISPATCH_LEVEL** 级上。

注意

完成例程使用的设备对象指针参数就是 I/O 堆栈单元中 **DeviceObject** 域中的指针。通常 **IoCallDriver** 设置该值。有时，在创建 **IRP** 时还同时创建一个额外的堆栈单元，以便能向完成例程传递参数而不用创建一个额外的上下文结构。如果此时创建者不去设置这个额外的 **DeviceObject** 域，那么完成例程得到的设备对象指针将为 **NULL**。

完成例程如何获得调用

IoCompleteRequest 函数负责调用每个驱动程序安装在各自堆栈单元中的完成例程。这个调用过程见流程图 5-7，开始，底层驱动程序的某段代码调用 **IoCompleteRequest** 例程以通知 IRP 处理结束。然后，**IoCompleteRequest** 参考当前的堆栈单元以查明其上层驱动程序是否安装了完成例程。如果没有，它就把堆栈指针前进到上一层堆栈单元并重复测试，直到找到某个完成例程或者到达堆栈顶部。最后 **IoCompleteRequest** 函数执行其它操作(如释放 IRP 占用的内存)。

当 **IoCompleteRequest** 函数发现含有完成例程指针的堆栈单元时，它就调用这个完成例程并检查其返回代码。如果返回代码是除了 **STATUS_MORE_PROCESSING_REQUIRED** 以外的其它值，它就把堆栈指针移动到上一层并重复前面的工作。如果返回代码是 **STATUS_MORE_PROCESSING_REQUIRED**，**IoCompleteRequest**

将停止前进并返回到调用者，而此时的 IRP 将处于一个中间状态。因此，如果完成例程在堆栈单元回卷过程中停止，那么其驱动程序有责任处理这个处于中间状态的 IRP。

在完成例程内部，一个 `IoGetCurrentIrpStackLocation` 调用将获得上一层堆栈单元的指针。上层堆栈单元的完成例程不应该依赖任何下层堆栈单元中的内容。为了加强这个规则，`IoCompleteRequest` 在调用完成例程前清除了下一个堆栈单元中的大部分内容。

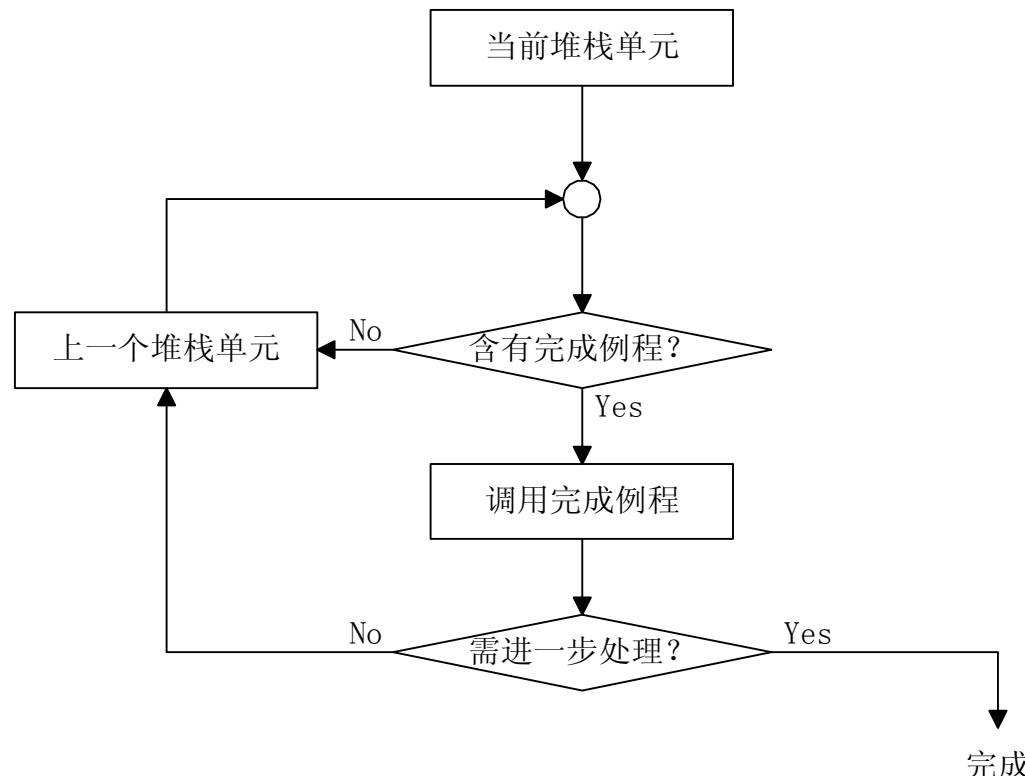


图 5-7. `IoCompleteRequest` 函数的执行过程

完成例程为什么要调用 `IoMarkIrpPending`

你可能已经注意到了上面完成例程框架代码的前两行：

```
if (Irp->PendingReturned)
    IoMarkIrpPending(Irp);
```

所有不返回 `STATUS_MORE_PROCESSING_REQUIRED` 状态的完成例程都需要这两行代码。如果你想知道为什么，读下面这些段。然而，你应该明白编写驱动程序不应该依靠有关 I/O 管理器是如何处理未决 IRP 的信息，这个处理过程在未来的 Windows 版本中可能会改变。

何时调用 `IoMarkIrpPending`

上文中陈述的规则“如果 `Irp->PendingReturned` 为 TRUE，那么任何不返回 `STATUS_MORE_PROCESSING_REQUIRED` 的完成例程都应该调用 `IoMarkIrpPending`”，这几乎完全是对的，但仍有例外。如果驱动程序分配了 IRP，安装了完成例程，然后在未改变堆栈指针的情况下调用 `IoCallDriver`，那么完成例程就不应该包含这两行代码，因为没有堆栈单元与你的驱动程序关联。(这种情况与完成例程的设备对象参数为 `NULL` 的情形类似。驱动程序通常做的是分配一个带有额外堆栈单元的 IRP，在第一个单元中设置 `DeviceObject` 指针，在调用 `IoSetCompletionRoutine` 和 `IoCallDriver` 前用 `IoSetNextIrpStackLocation` 函数跳过那个额外堆栈单元。如果你这样做，那么在完成例程中调用 `IoMarkIrpPending` 将不会出现问题，并且完成例程也能得到了一个有效的设备对象)

注意

下面解释是复杂的！

为了使系统吞吐量最大化，I/O 管理器希望驱动程序推迟其耗时 IRP 的完成。驱动程序通过调用 **IoMarkIrpPending** 函数并在派遣例程中返回 **STATUS_PENDING** 来表示完成操作被推迟。I/O 管理器的原始调用者通常希望在继续执行之前等待操作完成，所以 I/O 管理器在处理推迟完成时有下面类似的逻辑(不代表真正的 Microsoft 源代码):

```
Irp->UserEvent = pEvent; // don't do this yourself  
status = IoCallDriver(...);  
if (status == STATUS_PENDING)  
    KeWaitForSingleObject(pEvent, ...);
```

换句话说，如果 **IoCallDriver** 返回 **STATUS_PENDING**，则该段代码将在一个内核事件上等待。**IoCompleteRequest** 有责任在 IRP 最后完成时设置这个事件。该事件(**UserEvent**)的地址在 IRP 的一个不透明域中，所以 **IoCompleteRequest** 能够找到它。但实际的内容比这要多。

为了使问题更简单，假设请求仅涉及一个驱动程序。该驱动程序的派遣函数仅做两件事情：调用 **IoMarkIrpPending**，返回 **STATUS_PENDING**，而 **STATUS_PENDING** 实际上就是 **IoCallDriver** 返回的状态代码，此外，某段代码将要在一个事件上等待。**IoCompleteRequest** 调用发生在任意线程上下文中，因此该函数将调度一个特殊的内核 APC，这个 APC 执行在原始线程(现在正被阻塞)的上下文中。APC 例程将设置那个事件，并释放任何正等待操作完成的对象。有一些原因我们现在不需要深入，例如为什么用 APC 来做这个工作而不是用一个简单的 **KeSetEvent** 调用。

但是，排队一个 APC 是相对昂贵的。设想一下，不是直接返回 **STATUS_PENDING**，而是派遣例程自己调用 **IoCompleteRequest** 并返回某个其它状态。在这种情况下，**IoCompleteRequest** 的调用者将与 **IoCallDriver** 的调用者处于同一个线程上下文中。因此就没有必要排队一个 APC。另外，甚至没有必要调用 **KeSetEvent**，因为如果 I/O 管理器没有得到派遣例程返回的 **STATUS_PENDING**，它就不用等待某个事件。如果 **IoCompleteRequest** 恰好知道发生的这种情况，它将优化这个处理以避免调用 APC，能这样做吗？这就是 **IoMarkIrpPending** 的来处。

IoMarkIrpPending 是什么，它是 WDM.H 中的一个宏，这你可以自己去看，它在当前的堆栈单元中设置了一个名为 **SL_PENDING_RETURNED** 的标志。**IoCompleteRequest** 将把 IRP 的 **PendingReturned** 标志设置为它在顶级堆栈单元中找到的任何值。然后，它查看这个标志以确定派遣例程是否已返回或将返回 **STATUS_PENDING**。如果你做的正确，那么派遣例程在 **IoCompleteRequest** 做这个检查之前返回或在之后返回都无关紧要。在这种情况下的“正确做法”就是指你在做任何使 IRP 完成的操作之前都调用 **IoMarkIrpPending**。

所以，无论如何，**IoCompleteRequest** 都将查看 **PendingReturned** 标志。如果该标志设置，并且如果 IRP 是那种可以以异步方式完成的 IRP，那么 **IoCompleteRequest** 将简单地返回其调用者并不排队 APC。它假定自己运行在 IRP 发起者的线程上下文中，并且派遣例程很快会返回一个非未决状态的代码给请求发起者。请求发起者也不用等待那个事件，因为没有代码使那个事件进入信号态。到目前为止一切顺利。

现在，让我们把其它驱动程序加入到假想图中。顶级驱动程序不了解下面发生了什么，它只简单地把请求传递到下面，就象下面代码：

```
IoCopyCurrentIrpStackLocationToNext(Irp);  
IoSetCompletionRoutine(Irp, ...);  
return IoCallDriver(...);
```

换句话说，顶级驱动程序安装了一个完成例程并调用 **IoCallDriver**，然后返回从 **IoCallDriver** 得到的任何值。这个过程被重复几次，经过中间的驱动程序，当 IRP 到达能处理它的那个驱动程序级时，派遣例程就调用 **IoMarkIrpPending** 并返回 **STATUS_PENDING**。然后该 **STATUS_PENDING** 值按原路返回到顶级驱动程序，最后回到 IRP 的发起者。而发起者将立即在那个事件上等待，直到某个代码使那个事件变为信号态。

但要注意，调用 **IoMarkIrpPending** 的驱动程序仅在它自己的堆栈单元中设置了 **SL_PENDING_RETURNED** 标志。上面的驱动程序实际上仅返回 **STATUS_PENDING** 状态代码，它们没有调用 **IoMarkIrpPending**，因为它们不知道底层驱动程序到底发生了什么。这就是完成例程中那两行代码的来处。当 **IoCompleteRequest** 沿着 I/O 堆栈向上走时，它在每一层都停下来并把每层中的 **SL_PENDING_RETURNED** 标志设置为 **PendingReturned** 标志。如果某一层没有完成例程它就前进到上一层。这样，**SL_PENDING_RETURNED** 标志就被自下向上传播

到堆栈的顶层，并且如果任何驱动程序曾调用过 `IoMarkIrpPending`，则 IRP 的 `PendingReturned` 标志最终为 `TRUE`。

然而，`IoCompleteRequest` 不能自动传播 `SL_PENDING_RETURNED`。完成例程必须自己测试 IRP 的 `PendingReturned` 标志并调用 `IoMarkIrpPending` 来做到这个。如果每个完成例程都做了这个工作，那么 `SL_PENDING_RETURNED` 将顺利地从下而上传播到顶层驱动程序，就象 `IoCompleteRequest` 自己做了所有工作。

现在，我已经解释完这些复杂的细节，如果派遣例程要明确返回 `STATUS_PENDING`，那么返回前它必须调用 `IoMarkIrpPending`，并且在某些情况下，完成例程也应该这样做。如果完成例程打破了这个链条，那么线程将在事件上空等待，而且这个事件注定永远也不会被置成信号态。如果没有发现 `PendingReturned` 标志，那么 `IoCompleteRequest` 在处理完成过程时就象在同一个上下文中，因此它也不排队使事件改变状态的 APC。这与派遣例程忽略了 `IoMarkIrpPending` 调用而直接返回 `STATUS_PENDING` 的结果一样。

另一方面，调用 `IoMarkIrpPending` 然后同步完成 IRP 是正确的，尽管效率会低一点。这样做的结果是 `IoCompleteRequest` 将排队 APC，这个 APC 将改变事件的状态，但没有任何线程在这个事件上等待(其目的是使在调用 `KeSetEvent` 前保证这个事件存在)。这会降低一些效率，但不会有什害处。

另外，不要在完成例程中尝试避开 `IoMarkIrpPending` 调用，就象下面代码：

```
status = IoCallDriver(...);
if (status == STATUS_PENDING)
    IoMarkIrpPending(...); // DON'T DO THIS!
```

原因是如果你调用了 `IoCallDriver` 并给出了 IRP 指针，那么该函数返回后这个 IRP 指针可能是无效的。能完成 IRP 的接收者可能会调用 `IoFreeIrp`，而该函数将使 IRP 指针无效。

向下级传递请求

WDM 使用分层设备对象结构的目的就是使 IRP 能方便地从一层驱动程序传递到下一层驱动程序。回到第二章“WDM 驱动程序的基本结构”，我讲述了 AddDevice 例程如何创建设备对象堆栈，其中的一条语句如下：

```
pdx->LowerDeviceObject = IoAttachDeviceToDeviceStack(fdo, pdo);
```

fdo 是设备对象地址，**pdo** 是处于设备堆栈底部的物理设备对象地址。**IoAttachDeviceToDeviceStack** 函数返回给你下一层设备对象的地址。当你决定把从上层收到的 IRP 发送到自己的下层时，那么这个设备对象就是调用 **IoCallDriver** 时所指定的设备对象。

向下传递 IRP 时，你有责任初始化一个 **IO_STACK_LOCATION** 结构，下一层驱动程序将使用该结构获取它的参数。一种方法是执行物理拷贝，象这样：

```
...
IoCopyCurrentIrpStackLocationToNext(Irp);
status = IoCallDriver(pdx->LowerDeviceObject, Irp);
...
```

IoCopyCurrentIrpStackLocationToNext 是 WDM.H 中的一个宏，它把当前堆栈单元的所有域，除了一个属于 I/O 完成例程的域，都复制到下一个堆栈单元。在以前版本的 Windows NT 中，内核模式驱动程序开发者有时会复制整个堆栈单元，这会导致调用者的完成例程被调用两次。(回想一下，你的完成例程指针进入了下一层驱动程序的堆栈单元) 如果你想深入了解这个问题，参见《The NT Insider (May 1997, vol. 4, no. 3)》中的“Secrets of the Universe Revealed! How NT Handles I/O Completion”。**IoCopyCurrentIrpStackLocationToNext** 是 WDM 新引入的宏，利用它你可以避免这个问题。

如果你的驱动程序不用关心 IRP 传递到下层驱动程序之后的事情，你可以利用一个捷径来避免复制堆栈单元。在这种情形下，我们不需要安装完成例程——我刚说过驱动程序不关心该 IRP 以后的情况。图 5-8 显示了在这种情况下先后出现的事件。

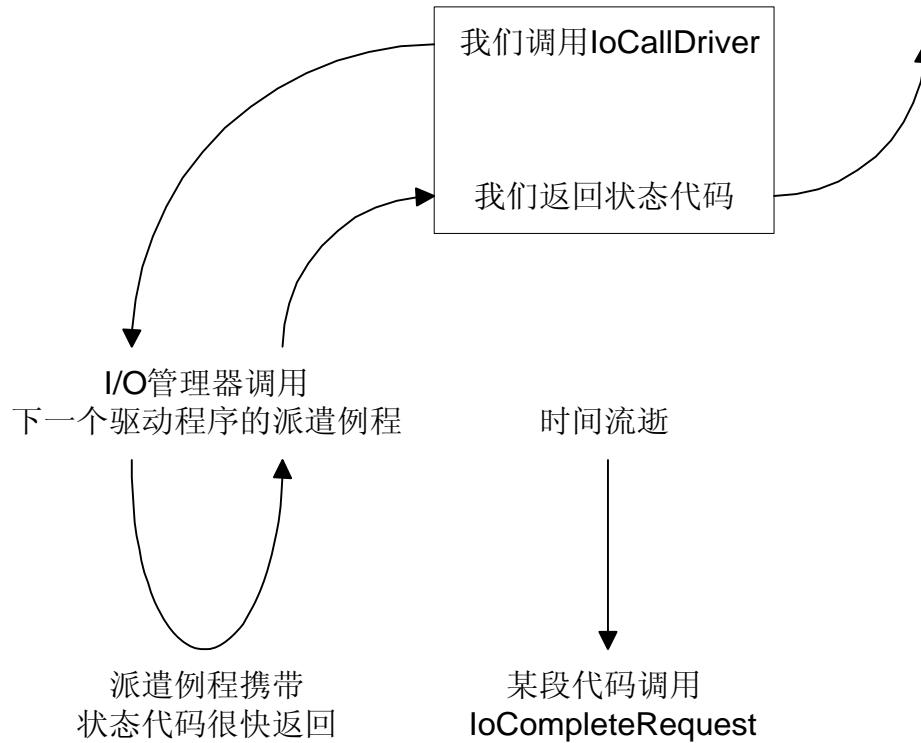


图 5-8. 向下传递 IRP 并忽略其结束状态

没有必要花费处理器时间去把你的堆栈单元内容复制到下一个堆栈单元，因为那个堆栈单元已经含有下一层驱动程序要得到的参数，以及自己上一层驱动程序可能给出的任何完成例程指针。因此，你可以使用下面捷径方法：

```
NTSTATUS ForwardAndForget(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    IoSkipCurrentIrpStackLocation(Irp);
    return IoCallDriver(pdx->LowerDeviceObject, Irp);
}
```

这个捷径存在于 **IoSkipCurrentIrpStackLocation** 函数(它实际上是一个宏，名字有些另人误解)中。这个宏的作用就是使堆栈指针少前进一步，而 **IoCallDriver** 函数会使堆栈指针向前一步，中和的结果就是堆栈指针不变。当下一个驱动程序的派遣例程调用 **IoGetCurrentIrpStackLocation** 时，它将收到与我们正使用的完全相同的 **IO_STACK_LOCATION** 指针，因此，它所处理的将是同一个请求(相同的主副功能代码)以及相同的参数。

你也许会注意到，**IO_STACK_LOCATION** 数组的最后一项在这里没有使用。实际上，如果我们下层的驱动程序也使用这个办法，也许会有更多的堆栈单元不被使用。虽然这不会产生任何问题，但确实有一些不必要的堆栈单元被分配了。也许完成处理中的堆栈单元回卷操作会更快一些。在回卷堆栈单元时，**IoCompleteRequest** 不使用任何绝对索引或指针。当它获得控制时，它仅从当前堆栈单元开始然后向上调用各个完成例程。所有被安装的完成例程都得到了调用。

流程图可以帮助你了解 **IoSkipCurrentIrpStackLocation** 函数的工作过程。图 5-9 显示了这样一种情形：某设备堆栈有三个驱动程序，你的驱动程序(功能设备对象[FDO])和其它两个驱动程序(一个上层过滤器设备对象 [FiDO]，一个 PDO)。在图(a)中，你将看到执行复制堆栈单元的 **IoCopyCurrentIrpStackLocationToNext** 函数，堆栈单元、各个参数，和完成例程之间的关系。在图(b)中，你将看到还是这样的关系，但使用的是 **IoSkipCurrentIrpStackLocation** 函数，第三个和最后一个堆栈单元被跳过，但没有人会对这个事实感到迷惑。

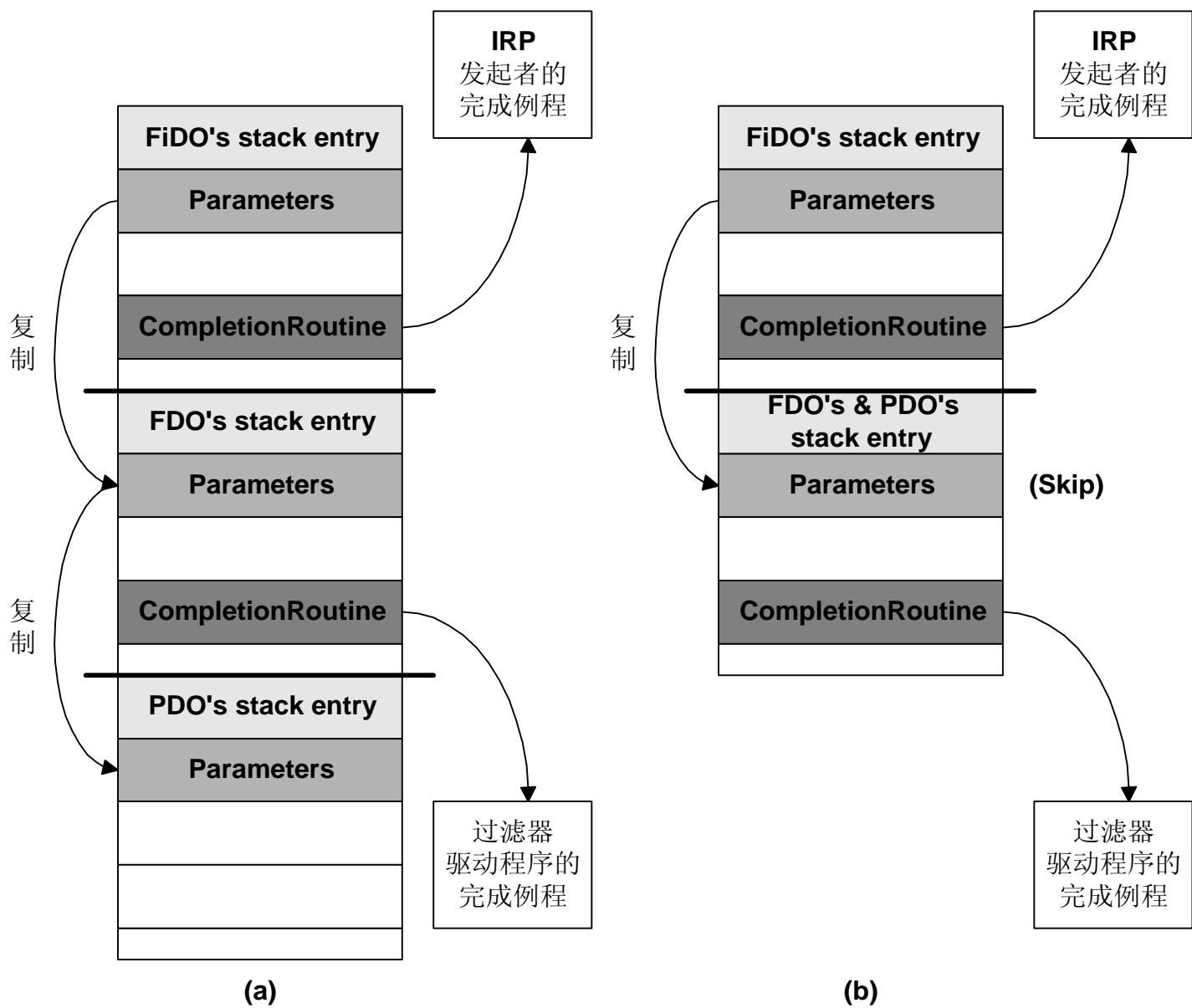


图 5-9. 复制 I/O 堆栈单元与跳过 I/O 堆栈单元

取消I/O请求

就象人们在现实生活中碰到的问题一样，程序有时会取消它们原来请求的 IRP。我并不是在讲某些简单的变化。应用程序可能发出某些需要长时间才能完成的请求，然后这个应用程序结束执行，而这个 IRP 仍然是未完成的。这种情况在 WDM 模型中尤为常见，例如当新硬件插入系统时，驱动程序必须停止执行以等待配置管理器重新分配硬件资源，设备电源关闭时也是这样。

为了在内核模式中取消一个请求，IRP 的创建者需调用 **IoCancelIrp** 函数。如果某线程终止时，它发出的请求仍然未完成，则操作系统自动为每个 IRP 调用 **IoCancelIrp**。用户模式应用程序调用 **CancelIo** 函数可以取消给定线程发出的所有未完成的异步操作。**IoCancelIrp** 仅仅是简单地设置 IRP 的 **Cancel** 标志位然后调用 IRP 的取消例程。即：它并不知道你在是否修改过 IRP 指针，也不知道你是否正在处理这个 IRP，所以它必须依靠一个你提供的取消例程来做大部分 IRP 取消工作。

IoCancelIrp 调用的作用更多的是一个建议而不是命令。如果每个确实要取消的 IRP 都能以 **STATUS_CANCELLED** 状态被完成，则很好。但是如果 IRP 可以很快被完成，那么驱动程序完全可以直接完成该 IRP。你应该提供一种方法来放弃那些长时间在派遣例程和 **StartIo** 例程的队列中等待的 IRP。多长时间算长由你自己判断；我的建议是在取消例程这边尝试，因为这比较容易去做并可以使你的驱动程序更匹配操作系统。

注意

关于如何在驱动程序中加入取消逻辑的解释非常复杂。你应该跳过细节论述而直接阅读代码例子，不用过多理会它们是如何工作的。

要是没有多任务就...

与取消 IRP 相关的是一个复杂的同步问题。在我描述和解决这个问题之前，我想描述一下在没有多任务和多处理器情况下的取消操作。在那个理想环境中，仅有部分 I/O 管理器，你的 **StartIo** 例程，和一个你提供的取消例程，如下：

- 当你调用 **IoStartPacket** 时，你指定了保存在 IRP 中的取消例程的地址。当你调用 **IoStartNextPacket**(在你的 DPC 例程中)时，你为一个布尔参数指定了 **TRUE** 值，指定这个值表示你要使用标准取消机制。在 **IoStartPacket** 或 **IoStartNextPacket** 调用你的 **StartIo** 例程之前，它把设备对象中的 **CurrentIrp** 域设置为指向你要发送的 IRP。如果队列中没有请求，则 **IoStartNextPacket** 把 **CurrentIrp** 域设置为 **NULL**。
- 你的 **StartIo** 例程要做的第一件事就是把 IRP 中的取消例程指针设置为 **NULL**。
- **IoCancelIrp** 无条件地设置 IRP 中的 **Cancel** 标志。然后它查看该 IRP 是否指定了一个取消例程。在你调用 **IoStartPacket** 函数和你的 **StartIo** 例程获得控制之间，IRP 中的取消例程指针将为非空值。在这种情况下，**IoCancelIrp** 调用了你的取消例程。最后，你从队列中取出该 IRP，并以 **STATUS_CANCELLED** 状态完成了该 IRP。然而，在 **StartIo** 开始处理该 IRP 之后，取消例程指针将为 **NULL**，并且 **IoCancelIrp** 将不做任何事。

同步化取消操作

不幸的是，我们写的代码要求运行在多处理器和多任务环境下，所以有时有些事情会提前出现。在我刚描述的逻辑中将会出现至少三种竞争情况。图 5-10 显示了这些竞争情况，下面我将阐述这些竞争情况：

- 假设 **IoCancelIrp** 刚设置了 **Cancel** 标志，而此时(在另一个 CPU 上运行的)**IoStartNextPacket** 把这个 IRP 从队列中提出并发送到 **StartIo** 例程。因为 **IoCancelIrp** 将很快把同一个 IRP 送到你的取消例程，所以 **StartIo** 例程不能用这个 IRP 做任何事。
- 这两个参与者(你的取消例程和 **IoStartNextPacket** 函数)或多或少可能会同时尝试从请求队列中删除同一个 IRP。这明显是不行的。

- StartIo 有可能先通过了 Cancel 标志测试阶段, 而该标志你现在正要设置, 由于第一种竞争, IoCancelIrp 还可能在 StartIo 能废除取消例程指针之前偷偷测试那个指针。现在, 你已经有一个完成这个请求的完成例程, 但同时, 另一个例程(可能是你的 DPC 例程)也要去完成该请求。天哪!

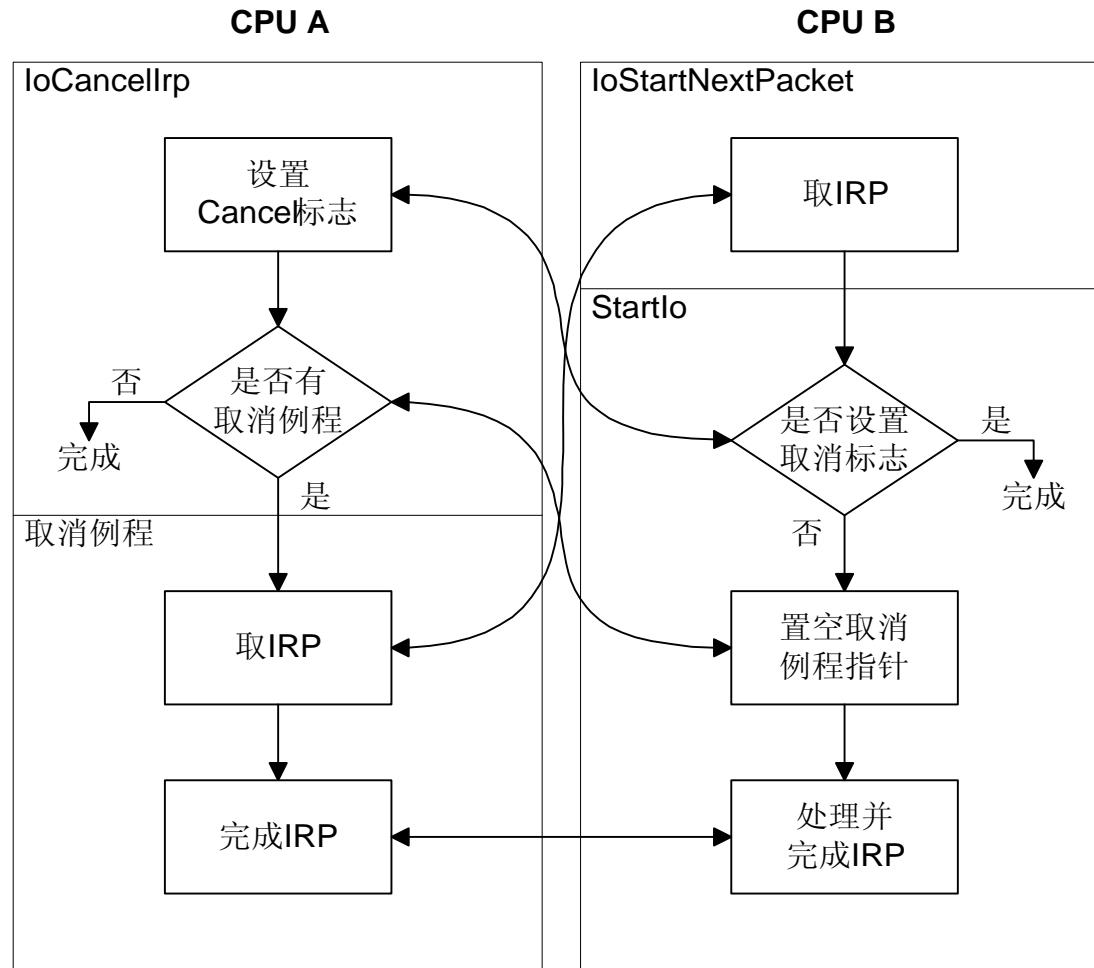


图 5-10. IRP 取消中的竞争

防止这些竞争的标准方法是使用一个在系统范围内有效的自旋锁, 称为取消自旋锁(cancel spin lock)。要取消某 IRP 的线程在 IoCancelIrp 函数中请求取消自旋锁, 在驱动程序的取消例程中释放取消自旋锁。启动某 IRP 的线程需要获取和释放取消自旋锁两次: 一次在调用 StartIo 前, 一次在 StartIo 函数中。实际的代码应该象下面这样:

```

VOID StartIo(PDEVICE_OBJECT fdo, PIRP Irp)
{
    KIRQL oldirql;
    IoAcquireCancelSpinLock(&oldirql);
    if (Irp != fdo->CurrentIrp || Irp->Cancel)
    {
        IoReleaseCancelSpinLock(oldirql);
        return;
    }
    else
    {
        IoSetCancelRoutine(Irp, NULL);
        IoReleaseCancelSpinLock(oldirql);
    }
    ...
}

VOID OnCancel(PDEVICE_OBJECT fdo, PIRP Irp)
{
    if (fdo->CurrentIrp == Irp)

```

```

{
    KIRQL oldirql = Irp->CancelIrql;
    IoReleaseCancelSpinLock(DISPATCH_LEVEL);
    IoStartNextPacket(fdo, TRUE);
    KeLowerIrql(oldirql);
}
else
{
    KeRemoveEntryDeviceQueue(&fdo->DeviceQueue, &Irp->Tail.Overlay.DeviceQueueEntry);
    IoReleaseCancelSpinLock(Irp->CancelIrql);
}
CompleteRequest(Irp, STATUS_CANCELLED, 0);
}

```

避免使用全局取消自旋锁

Microsoft 认为全局取消自旋锁是多处理器系统中的一个瓶颈，驱动程序处理每个 IRP 时都需要多次获取和释放这个自旋锁，而 CPU 在等待自旋锁时什么也不能干。现在，Windows 2000 中的 `IoSetCancelRoutine` 函数以原子交换操作实现，并且 `IoCancelIrp` 函数的操作也遵循一个精确的顺序，因此我们完全可以避免使用全局取消自旋锁。Ervin Peretz 的文章 "The Windows Driver Model Simplifies Management of Device Driver I/O Requests" (Microsoft Systems Journal, January 1999)，讲述了如何实现不用取消自旋锁的取消操作。

虽然依靠全局取消自旋锁是一个坏的方法，但有时你不能避免使用它。比如，用标准模型处理 IRP。

在内部，调用你的代码的系统例程象下面代码(注意：这不是真正的 Windows 2000 源代码)：

```

VOID IoStartPacket(PDEVICE_OBJECT device, PIRP Irp, PULONG key, PDIVER_CANCEL cancel)
{
    KIRQL oldirql;
    IoAcquireCancelSpinLock(&oldirql);
    IoSetCancelRoutine(Irp, cancel);
    device->CurrentIrp = Irp;
    IoReleaseCancelSpinLock(oldirql);
    device->DriverObject->DriverStartIo(device, Irp);
}

VOID IoStartNextPacket(PDEVICE_OBJECT device, BOOLEAN cancancel)
{
    KIRQL oldirql;
    if (cancancel)
        IoAcquireCancelSpinLock(&oldirql);
    PKDEVICE_QUEUE_ENTRY p = KeRemoveDeviceQueue(&device->DeviceQueue));
    PIRP Irp = CONTAINING_RECORD(p, IRP, Tail.Overlay.DeviceQueueEntry);
    device->CurrentIrp = Irp;
    if (cancancel)
        IoReleaseCancelSpinLock(oldirql);
    device->DriverObject->DriverStartIo(device, Irp);
}

BOOLEAN IoCancelIrp(PIRP Irp)
{
    IoAcquireCancelSpinLock(&Irp->CancelIrql);
    Irp->Cancel = TRUE;
    PDIVER_CANCEL cancel = IoSetCancelRoutine(Irp, NULL);
}

```

```
if (cancel)
{
    (*cancel)(device, Irp);
    return TRUE;
}
IoReleaseCancelSpinLock(&Irp->CancelIrql);
return FALSE;
}
```

很明显，上面只是实际例程的框架代码，真正的系统例程不能这么简单。例如，**IoStartNextPacket** 将测试 **KeRemoveDeviceQueue** 的返回值。另外，我省略了 **IoStartNextPacketByKey** 例程，它是 **IoStartNextPacket** 例程的姊妹函数，基于一个排序键来选择请求。

为了说明这个代码的工作过程，我们需要考虑三种情况。图 5-11 可以帮助你理解下面的讨论。我们假设上面代码运行在多处理器平台上，CPU A 中的代码要取消一个 IRP，而 CPU B 中的代码却要启动这个 IRP。因为同时仅有两个活动与这个 IRP 相关，所以我们不用考虑系统中有超过两个 CPU 的情况。

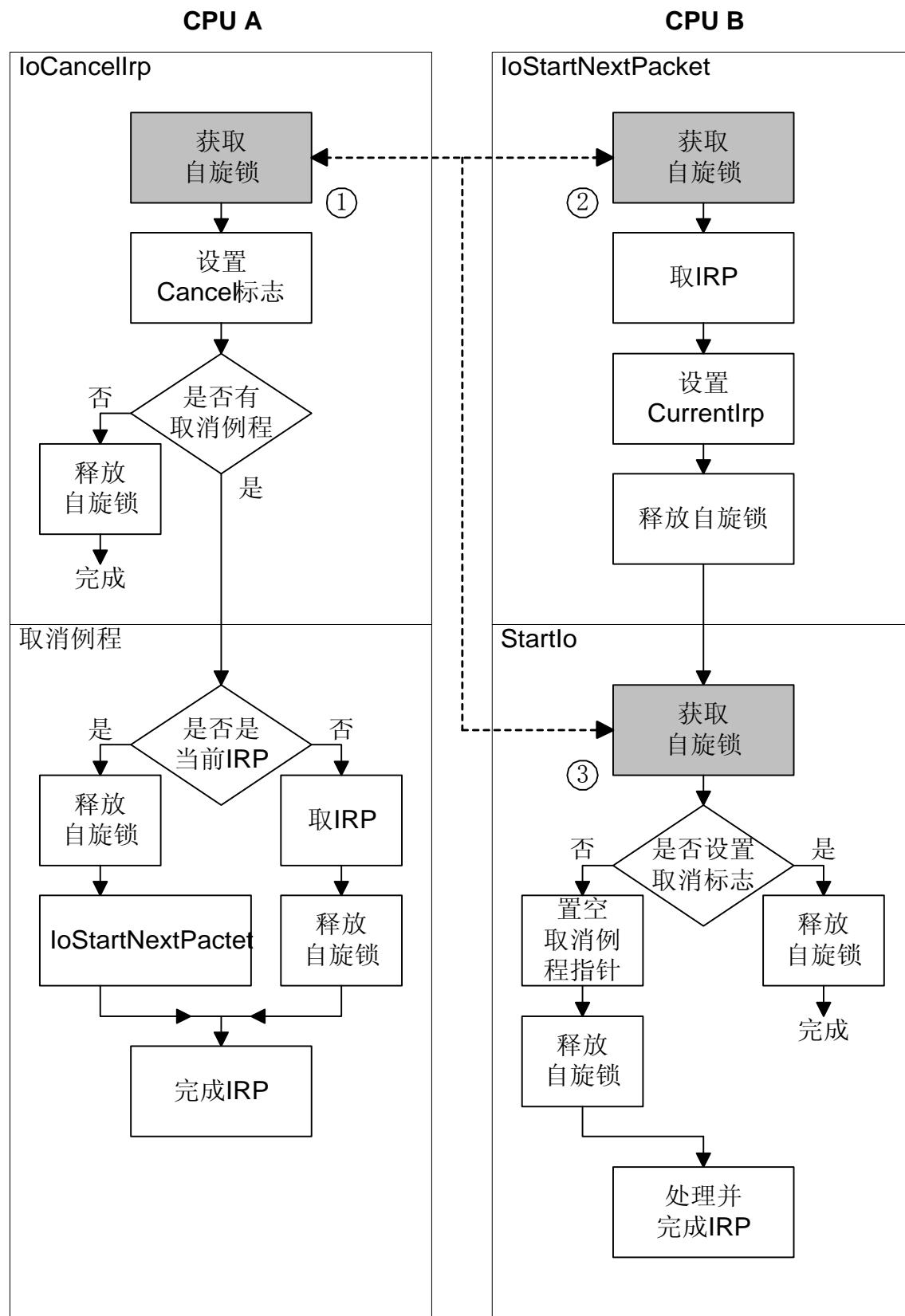


Figure 5-11. 利用取消自旋锁保护取消操作

情况 1：CPU A先获得自旋锁

假定 CPU A 先通过了标记 1 并获取了自旋锁。它首先设置该 IRP 的 Cancel 标志，然后测试 IRP 是否有一个取消例程(测试 IRP 的 CancelRoutine)。回答应该是 Yes，因为使 CancelRoutine 指针无效的代码还没有运行，正在标记 2 处等待着自旋锁。接着 CPU A 调用取消例程，出队该 IRP，最后释放自旋锁。此时，在标记 2 处等待的 CPU B 获取了自旋锁，它先从队列中取出一个 IRP，这个 IRP 与 CPU A 处理的 IRP 不是同一个，它只是队列中的下一个 IRP。因此，CPU A 将以 STATUS_CANCELLED 状态完成该 IRP，而 CPU B 将前进并初始化下一个排队的请求。

情况 2：就在CPU A刚要获取自旋锁前CPU B获得了自旋锁

现在假定在 CPU A 刚要获取自旋锁之前，CPU B 在标记 2 处获取了自旋锁。CPU B 首先从队列中提取一个 IRP 并设置设备对象的 `CurrentIrp` 指向该 IRP。然后它释放自旋锁并调用 `StartIo`。这时，CPU A 在标记 1 处抓住了自旋锁，因此，CPU B 将不能前进到标记 3 处。CPU A 设置 `Cancel` 标志并调用取消例程。取消例程发现该 IRP 是当前 IRP，所以它释放了自旋锁。这样，CPU B 就可以自由通过 `StartIo` 中的标记 3 处。它将发现该 IRP 的 `Cancel` 标志已被设置，所以它释放了自旋锁并返回。在这一时刻，设备是空闲的。CPU A 继续执行取消例程，它将调用 `IoStartNextPacket` 例程，然后完成取消请求。

拥有取消自旋锁时不调用 `IoStartNextPacket` 例程这一点非常重要，因为该例程也要获取这个锁。如果我们在拥有取消自旋锁时调用了这个函数，则我们所在的 CPU 将死锁，自旋锁不能被递归获取。

`StartIo` 中的代码还避免了另一个竞争情况。你可能想知道为什么 `StartIo` 在测试 `Cancel` 标志前测试了 `CurrentIrp` 域。假设 CPU A 在 CPU B 快要到达标记 3 前马上就要完成该 IRP。此时 CPU A 上的某些代码可能调用 `IoFreeIrp` 释放该 IRP 的存储。所以 CPU B 使用的 `Irp` 指针可能是无效的，引用这样的指针将是不安全的。

再看一眼前面的 `IoStartNextPacket` 代码，它在取消自旋锁的保护下改变了设备对象的 `CurrentIrp` 指针。取消例程在完成 IRP 之前调用了 `IoStartNextPacket` 例程。因此，必然会发生下面两种情况：一种情况是 CPU B 的 `StartIo` 在 CPU A 的 `IoStartNextPacket` 之前获得自旋锁，这时的 IRP 指针是安全的并且 `Cancel` 标志将是设置的，另一种情况是 CPU B 的 `StartIo` 在 CPU A 的 `IoStartNextPacket` 之后获得自旋锁，而这时的 `Irp` 变量将不等于 `CurrentIrp`，`IoStartNextPacket` 改变了它，所以 CPU B 不能引用这个指针。

从前面两段论述中我们可以得出这样的推论，如果你不想在取消例程中调用 `IoStartNextPacket`(或者 `IoStartNextPacketByKey`)例程，你必须确保在拥有取消自旋锁时设置 `CurrentIrp` 为 `NULL`。

情况 3：CPU B 获得自旋锁两次

第三个情况是，在 `StartIo` 中，CPU B 在标记 3 处获得了自旋锁，而 CPU A 停在标记 1 处。`StartIo` 例程在释放自旋锁前先把 `CancelRoutine` 指针置空，CPU A 获得了自旋锁后立即设置了 IRP 中的 `Cancel` 标志，但它将不能调用取消例程，因为那个指针现在为空。与此同时，CPU B 将处理该 IRP 直至完成，尽管这时 IRP 的 `Cancel` 标志已被设置，但如果 IRP 可以很快完成，这样做是可以的。

清除相关的IRP

与取消 IRP 紧密关联的主题是带有 `IRP_MJ_CLEANUP` 主功能码的 I/O 请求。为了解释这个请求的处理，你需要一些背景资料。

当应用程序或其它驱动程序要访问你的设备时，它们首先打开该设备的一个句柄。应用程序可以调用 `CreateFile` 函数做到这一点；驱动程序则调用 `ZwCreateFile` 函数。在内部，这些函数先创建一个内核文件对象，然后把这个文件对象附加到 `IRP_MJ_CREATE` 请求中，最后把 `IRP_MJ_CREATE` 请求发往你的驱动程序。当这个句柄完成使命后，应用程序或其它驱动程序就调用 `CloseHandle` 或 `ZwClose` 函数关闭该句柄。在内部，这些函数向你的驱动程序发送 `IRP_MJ_CLOSE` 请求。然而，就在发送 `IRP_MJ_CLOSE` 请求前，I/O 管理器先向你发送了 `IRP_MJ_CLEANUP` 请求，这样你就可以有机会取消所有未处理的 IRP，这些 IRP 属于这个文件对象但仍停留在你的队列中。从驱动程序的角度来看，这些请求都有一个共同点，即在任何时刻它们的堆栈单元都指向同一个文件对象。

图 5-12 显示了当你收到 `IRP_MJ_CLEANUP` 请求时，你应做的动作。

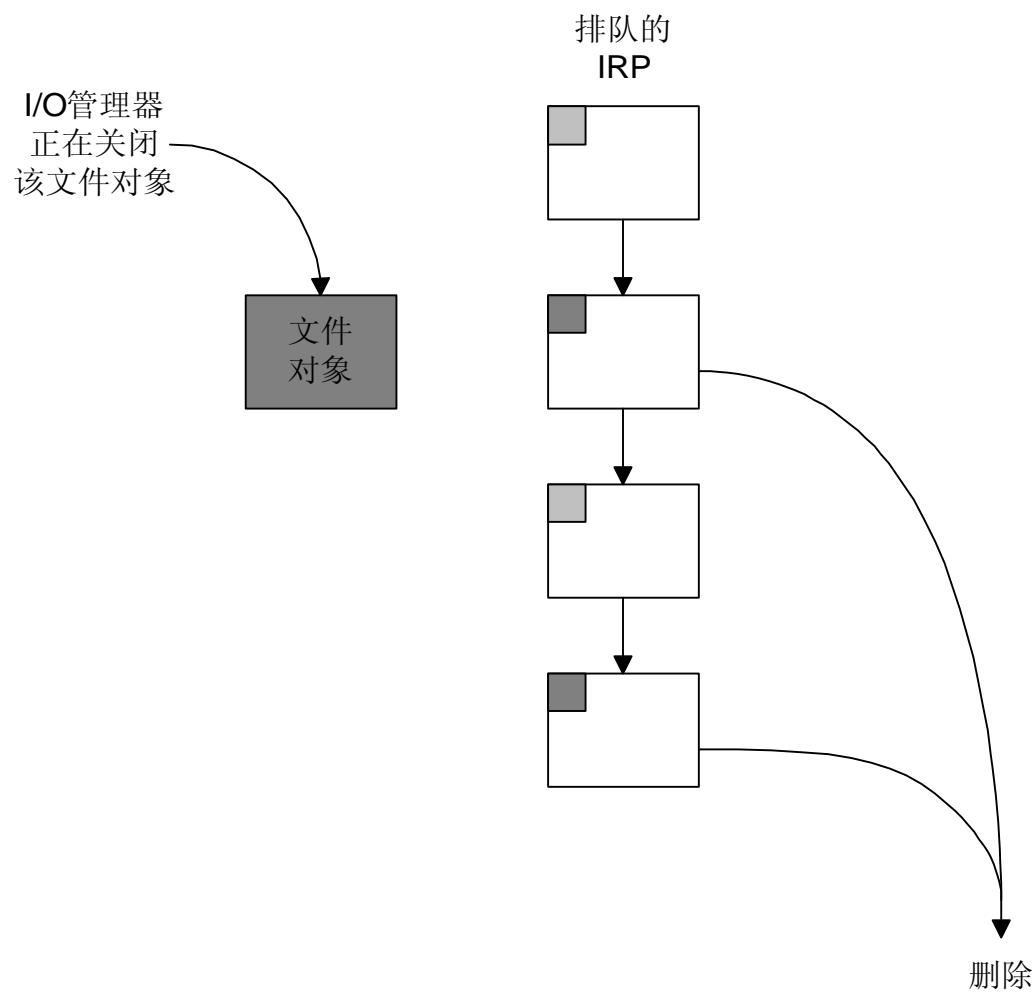


图 5-12. 驱动程序响应 *IRP_MJ_CLEANUP* 请求

如果你使用标准模型，你的派遣函数应该象下面这样：

```

NTSTATUS DispatchCleanup(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    PFILE_OBJECT fop = stack->FileObject;                                <--1
    LIST_ENTRY cancellist;                                                 <--2
    InitializeListHead(&cancellist);

    KIRQL oldirql;                                                       <--3
    IoAcquireCancelSpinLock(&oldirql);
    KeAcquireSpinLockAtDpcLevel(&fdo->DeviceQueue.Lock);

    PLIST_ENTRY first = &fdo->DeviceQueue.DeviceListHead;
    PLIST_ENTRY next;

    for (next = first->Flink; next != first; )                         <--4
    {
        PIRP QueuedIrp = CONTAINING_RECORD(next,
            IRP, Tail.Overlay.ListEntry);
        PIO_STACK_LOCATION QueuedIrpStack =
            IoGetCurrentIrpStackLocation(QueuedIrp);

        PLIST_ENTRY current = next;
        next = next->Flink;

        if (QueuedIrpStack->FileObject != fop)                            <--5
    }
}

```

```

        continue;

        IoSetCancelRoutine(QueuedIrp, NULL);
        RemoveEntryList(current);
        InsertTailList(&cancelist, current);
    }

    KeReleaseSpinLockFromDpcLevel(&fdo->DeviceQueue.Lock);           <--6
    IoReleaseCancelSpinLock(olddirql);

    while (!IsListEmpty(&cancelist))                                     <--7
    {
        next = RemoveHeadList(&cancelist);
        PIRP CancelIrp = CONTAINING_RECORD(next, IRP, Tail.Overlay.ListEntry);
        CompleteRequest(CancelIrp, STATUS_CANCELLED, 0);
    }

    return CompleteRequest(Irp, STATUS_SUCCESS, 0);                      <--8
}

```

1. 我们要在队列中寻找与 **IRP_MJ_CLEANUP** 同属于一个文件对象的 **IRP**。堆栈单元中含有该文件对象的指针。
2. 我们的策略是在两个自旋锁的保护下，把要取消的 **IRP** 从主设备队列中拉出来。由于可能有多个 **IRP** 要取消，所以创建一个临时的链表可以方便地把它们组织起来，因此我们在这里初始化一个链表头。
3. 为了从队列中提取 **IRP**，我们需要两个自旋锁。我们获取全局取消自旋锁以避免 **IoCancelIrp** 的干扰。我们还需要一个保护设备队列的自旋锁，它可以防止与 **ExInterlockedXxxList** 函数同操作一个队列时所带来的干扰。
4. 利用这个循环我们可以检查设备队列上的每个 **IRP**。我们知道，由于我们拥有保护队列的自旋锁，所以其它代码不能向队列添加或删除 **IRP**。因此我们使用常规链表原语(非互锁)访问队列链表。
5. 当发现某个 **IRP** 属于指定文件对象时，我们就把它从设备队列中删除并添加到我们临时创建的 **cancelist** 链表中。我们同时还把这个 **IRP** 中的取消例程指针置空。注意，我们应该测试排队 **IRP** 堆栈单元中的文件对象指针，但查看排队 **IRP** 中的不透明域 **Tail.Overlay.OriginalFileObject** 是错误的。在 **IRP** 完成时，**I/O** 管理器利用该域指出何时该废除文件对象指针。所以它有时可能是 **NULL**，即使那时，该 **IRP** 仍属于那个文件对象。而对于堆栈单元，如果 **IRP** 正确创建，那么它总是含有正确的文件对象指针。
6. 我们在循环结束处释放两个自旋锁。
7. 这个循环取消了我们在上一个循环中找到的 **IRP**。因此，我们不再需要自旋锁。
8. 最后这个 **IoCompleteRequest** 调用属于 **IRP_MJ_CLEANUP** 请求本身，它应该是成功的。

文件对象

通常，在设备堆栈中仅有一个驱动程序(实际上是功能驱动程序)实现了全部三种请求：**IRP_MJ_CREATE**、**IRP_MJ_CLOSE**、**IRP_MJ_CLEANUP**。**I/O** 管理器先创建一个文件对象(一个常规内核对象)，然后把该文件对象连接到 **I/O** 堆栈中，带有这个文件对象的 **I/O** 堆栈将被发往到这三种请求的派遣例程。任何发往设备的 **IRP** 都应该带有该文件对象的指针，并且这个指针也应该存在于 **I/O** 堆栈中。从某种意义上说，处理这三种 **IRP** 的驱动程序就是该文件对象的拥有者，因此，只有这个驱动程序才有资格使用该文件对象中的 **FsContext** 和 **FsContext2** 域。所以你的 **DispatchCreate** 例程可以把某些信息放到其中一个上下文域中，而其它派遣例程都可以使用这些信息，最后由 **DispatchClose** 例程清除这些信息。

上面代码的真正要点是第一个循环，在这个循环中我们把要取消的 **IRP** 从设备队列中删除。拥有设备队列的自旋锁可以保证队列操作本身的完整性。我们还需要获得全局取消自旋锁。如果没有这个自旋锁，其它代码将在我们删除某 **IRP** 时调用 **IoCancelIrp** 例程处理这个 **IRP**，而 **IoCancelIrp** 将调用取消例程，但取消例程在从队列中提取 **IRP** 时会被阻塞。(参考“同步化取消”段中的取消例程例子)一旦我们释放了队列锁，取消例程将提取队列中的 **IRP** 并完成它。这两步都是错误的，因为派遣例程正在做与这两步完全相同的事情。解决的办法是阻止 **IoCancelIrp** 在拥有全局取消自旋锁的情况下继续前进，即，使 **IRP** 变为不可取消的。

你可能已经注意到了，我们先获取了全局取消自旋锁，然后再获取设备队列锁。以相反顺序获取这两个锁可能会导致死锁：因为取消例程和 **I/O** 管理器中的例程(如 **IoStartPacket**)是先获取全局锁，然后再调用

KeXxxDeviceQueue 例程，而这些例程都需要获取队列锁。因此，可能出现的情况是，我们获取了队列锁然后在获取全局锁时阻塞，而其它例程获取了全局锁并等待队列锁的释放，这就形成了死锁。

在以前的文字框“避免使用全局取消自旋锁”中，我曾提到全局取消自旋锁可能造成严重的系统瓶颈，这里还有一个事实使这个瓶颈问题变得更严重，即 **IRP_MJ_CLEANUP** 例程需要占用这个自旋锁足够长的时间以检查整个 **IRP** 队列。假设没有人试图执行意外的活动(如取消一个 **IRP**)，那么每个驱动程序在每次调用 **IoStartPacket**、**IoStartNextPacket**、**StartI/o**、和 **DispatchCleanup** 例程时都会请求这个锁，此外，当系统变得更慢时，队列中的 **IRP** 会更多，而清除派遣例程将需要更多的时间检查队列，从而增加了全局取消自旋锁的竞争，这又降低了系统性能。

由于存在着性能瓶颈，所以，如果可能我们应该避免使用全局取消自旋锁。要做到这一点，我们需要管理自己的 **IRP** 队列。详细内容请见下一章。

管理自己的IRP

现在，我已经解释完 IRP 处理的所有底层结构，我们可以返回到创建 IRP 的主题上。我曾提到过有四种不同的服务函数可以用来创建 IRP。但我不得不推迟到现在才讨论如何选择它们。下面事实供你在选择时参考：

- `IoBuildAsynchronousFsdRequest` 和 `IoBuildSynchronousFsdRequest` 函数仅能用于创建主功能码在表 5-3 中列出的 IRP。
- `IoBuildDeviceIoControlRequest` 仅能用于创建主功能码为 `IRP_MJ_DEVICE_CONTROL` 或 `IRP_MJ_INTERNAL_DEVICE_CONTROL` 的 IRP。
- 当调用 `IoCompleteRequest` 时，有些代码可能会释放 IRP 占用的内存。
- 你应该事先做好安排以便该 IRP 能被 `IoCancelIrp` 调用取消。

表 5-3. 适用于 `IoBuildXxxFsdRequest` 的 IRP 类型。

主功能码
<code>IRP_MJ_READ</code>
<code>IRP_MJ_WRITE</code>
<code>IRP_MJ_FLUSH_BUFFERS</code>
<code>IRP_MJ_SHUTDOWN</code>
<code>IRP_MJ_PNP</code>
<code>IRP_MJ_POWER</code>

使用`IoBuildSynchronousFsdRequest`

`IoBuildSynchronousFsdRequest` 函数的调用格式如下：

```
PIRP Irp = IoBuildSynchronousFsdRequest(MajorFunction,
                                         DeviceObject,
                                         Buffer,
                                         Length,
                                         StartingOffset,
                                         Event,
                                         IoStatusBlock);
```

MajorFunction(ULONG) 是新 IRP 的主功能码(见表 5-3)。**DeviceObject(PDEVICE_OBJECT)** 是该 IRP 最初要发送到的设备对象的地址。对于读写请求，你必须提供 **Buffer(PVOID)**、**Length(ULONG)**、**StartingOffset(PLARGE_INTEGER)** 参数。Buffer 是一个内核模式数据缓冲区的地址，Length 是读写操作的字节长度，StartingOffset 是读写操作在目标文件中的定位。对于该函数创建的其它请求，这些参数将被忽略。(这就是为什么该函数在 WDM.H 中的原型将这些参数定为“可选的”，但对于读写请求它们则是必需的) I/O 管理器假定你给出的缓冲区地址在当前进程上下文中是有效的。

Event(PKEVENT) 是一个事件对象的地址，`IoCompleteRequest` 应该在操作完成时设置这个事件，**IoStatusBlock(PIO_STATUS_BLOCK)** 是一个状态块的地址，该状态块用于保存 IRP 结束状态和信息。在操作完成前，事件对象和状态块必须一直存在于内存中。

如果创建的是读写 IRP，那么在提交该 IRP 前你不需要做任何事。如果创建的是其它类型的 IRP，你需要用附加的参数信息完成第一个堆栈单元；**MajorFunction** 已经被设置，不能设置未公开的 **Tail.Overlay.OriginalFileObject** 域，这样做将使文件对象在 IRP 完成时被错误地废除。同样也不可能设置 **RequestorMode**，它已经被初始化成 **KernelMode**。(我提到这两点是因为我回想起曾读过该函数的一个公开讨论，该讨论认为应该做这两件事，但我认为没有必要) 现在，你可以提交该 IRP 并等待其完成：

```
PIRP Irp = IoBuildSynchronousFsdRequest(...);
NTSTATUS status = IoCallDriver(DeviceObject, Irp);
if (status == STATUS_PENDING)
    KeWaitForSingleObject(Event, Executive, KernelMode, FALSE, NULL);
```

IRP 完成后，你可以通过察看你的 I/O 状态块来了解该 IRP 的结束状态和相关信息。

很明显，在等待操作完成前你应该运行在 PASSIVE_LEVEL 级上和非任意线程上下文中。

清除

我曾说过，你必须事先计划好 IRP 占用内存的释放以及 IRP 的取消。第一个问题很容易解决，当使用 `IoBuildSynchronousFsdRequest` 创建 IRP 时，I/O 管理器自动释放 IRP 占用的内存。如果是需要系统缓冲区或内存描述符的读写请求，I/O 管理器也能自动清除。由于该函数的这些便利性，我们经常调用它。

尽管清除同步 IRP 比较容易，但取消它并不那么容易。继续往下读...

取消同步IRP

系统中仅有两个实体可以取消 IRP。一个是 I/O 管理器中称为 `thread rundown` 的代码，当线程结束时仍有未处理的 IRP，就执行这段代码。另一个实体就是产生该 IRP 的第一个驱动程序。我们应该尽量小心地避免模糊的，低概率的问题。为了说明问题，假设你要强制执行一个 5 秒钟长的 I/O 操作。当过了 5 秒钟后操作仍未完成，你希望取消该操作。下面是完成这个功能的假想代码：

```
SomeFunction()
{
    KEVENT event;
    IoInitializeEvent(&event, ...);
    PIRP Irp = IoBuildSynchronousFsdRequest(...);
    NTSTATUS status = IoCallDriver(DeviceObject, Irp);
    if (status == STATUS_PENDING)
    {
        LARGE_INTEGER timeout;
        timeout.QuadPart = -5 * 10000000;
        if (KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, &timeout) == STATUS_TIMEOUT)
        {
            IoCancelIrp(Irp); // don't do this!
            KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);
        }
    }
}
```

对 `KeWaitForSingleObject` 的第二次调用确保了事件对象在 I/O 管理器用完它之前不超出有效范围。任何拥有该 IRP 的人都必须迅速完成它，所以任何发生在此处的过度延迟都将是一个程序错误。(说起来容易，哼?)

上面代码的错误确实很小。设想一下，如果某人要调用 `IoCompleteRequest` 完成该 IRP，时间正好与我们要调用 `IoCancelIrp` 取消该 IRP 相同。例如，在 5 秒超时后的短暂操作。`IoCompleteRequest` 初始化一个 IRP 处理过程，在处理过程的结尾它调用了 `IoFreeIrp` 例程。如果 `IoFreeIrp` 调用碰巧发生在 `IoCancelIrp` 破坏该 IRP 之前，那么当 `IoCancelIrp` 修改 IRP 的 `Cancellable`、`Cancel`、`CancelRoutine` 域的同时也破坏了正常的内存数据。

我刚描述的情节确实不太可能发生。但正如 James Thurber 说过的“一次就够了”。这种错误几乎不可能被发现，所以你应该尽可能地预防它。在当前版本的 Windows 98 和 Windows 2000 中，一个公认的技术是依赖这样一个事实，`IoFreeIrp` 调用都发生在产生该 IRP 的线程的 APC 中。你应该先确保自己在那个线程中，然后提升 IRQL 到 APC_LEVEL，检测该 IRP 是否已被完成，如果没有则调用 `IoCancelIrp`。在当前的系统中，这样做可以有效

地阻塞 APC 及有问题的 `IoFreeIrp` 调用。DDK 中的 `USBCAMD` 例子就是这样做的。在 Compuware Numega 的 Web 站点上可以找到关于这项技术的讨论和注释。

不能认为未来版本的 Windows 也使用 APC 来执行 IRP 的清除。因此，你不能把提升 IRQL 到 APC_LEVEL 作为避免 `IoCancelIrp` 和 `IoFreeIrp` 竞争的方法。我的意思是，未来版本的操作系统很可能会改变，因此，这种技术将不足以避免竞争发生。所以，你需要另外一种方法来解决这个问题。

解决这个问题的关键是使 `IoFreeIrp` 调用发生在任何可能的 `IoCancelIrp` 调用之后。为了做到这一点，我们使用一个返回 `STATUS_MORE_PROCESSING_REQUIRED` 状态的完成例程，见下面代码：

```
SomeFunction()
{
    KEVENT event;
    IoInitializeEvent(&event, ...);
    PIRP Irp = IoBuildSynchronousFsdRequest(...);
    IoSetCompletionRoutine(Irp, OnComplete, (PVOID) &event, TRUE, TRUE, TRUE);
    NTSTATUS status = IoCallDriver(...);
    if (status == STATUS_PENDING)
    {
        LARGE_INTEGER timeout;
        timeout.QuadPart = -5 * 10000000;
        if (KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, &timeout) == STATUS_TIMEOUT)
        {
            IoCancelIrp(Irp); // okay in this context
            KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);
        }
    }
    KeClearEvent(&event);
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);
}

NTSTATUS OnComplete(PDEVICE_OBJECT junk, PIRP Irp, PVOID pev)
{
    KeSetEvent((PKEVENT) pev, IO_NO_INCREMENT, FALSE);
    return STATUS_MORE_PROCESSING_REQUIRED;
}
```

粗体部分的代码用于防止竞争。假设 `IoCallDriver` 返回 `STATUS_PENDING`。在通常情况下，操作将正常完成，某些低级驱动程序将调用 `IoCompleteRequest`。因此，我们的完成例程将获得控制，它把事件对象设置成信号态，而此时我们的主程序(`SomeFunction`)正在该事件上等待。因为该完成例程返回 `STATUS_MORE_PROCESSING_REQUIRED`，所以 `IoCompleteRequest` 将停在该 IRP 上。在 `SomeFunction` 程序中我们又获得了控制，并且我们的等待是以正常方式结束的。由于 IRP 还没有被清除，我们需要再一次调用 `IoCompleteRequest` 例程以触发正常的清除机制。我们仍旧不能让事件对象太快地超出函数范围，所以又执行了一次等待。

现在我们假设要取消这个 IRP，由于我们的完成例程返回 `STATUS_MORE_PROCESSING_REQUIRED`，这阻止了清除机制的执行，所以我们不用担心 IRP 会被 `IoFreeIrp` 释放。不管 `IoCancelIrp` 怎么处置我们的 IRP，该 IRP 只有在 `IoCancelIrp` 安全返回后，在第二次调用 `IoCompleteRequest` 时才被释放。

使用 `IoAllocateIrp`

如果你想更深入一些，你可以使用 `IoAllocateIrp` 函数创建任何类型的 IRP：

```
PIRP Irp = IoAllocateIrp(StackSize, ChargeQuota);
```

StackSize(CCHAR)是与 IRP 一同分配的 I/O 堆栈单元的数目，**ChargeQuota(BOOLEAN)**指出该内存分配是否应充入进程限额。通常，你可以从该 IRP 对应的设备对象中获得 StackSize 参数，ChargeQuota 参数指定为 FALSE，例如：

```
PDEVICE_OBJECT DeviceObject;
PIRP Irp = IoAllocateIrp(DeviceObject->StackSize, FALSE);
```

当你使用 `IoAllocateIrp` 时，必须为它创建的 IRP 安装一个完成例程，并且该完成例程必须返回 `STATUS_MORE_PROCESSING_REQUIRED`。另外，你还要负责释放该 IRP 以及任何相关的对象。如果你不打算取消该 IRP，你的完成例程应该象这样：

```
NTSTATUS OnComplete(PDEVICE_OBJECT DeviceObject, PIRP Irp, PVOID Context)
{
    IoFreeIrp(Irp);
    return STATUS_MORE_PROCESSING_REQUIRED;
}
```

如果 IRP 的发起线程提前结束，则 `IoAllocateIrp` 创建的 IRP 不能被自动取消。

松散的结尾

我将以描述以前未曾提到的，而你又需要知道的一些其它事情来结束本章。包括两种 IRP 创建方法，以及如何定位 `IoCallDriver` 函数使用的设备对象。

使用 `IoBuildDeviceIoControlRequest`

我将在第九章“如何执行 I/O 控制操作”中讨论 `IoBuildDeviceIoControlRequest` 函数。在清除和取消方面，它所创建的 IRP 与 `IoBuildSynchronousFsdRequest` 创建的 IRP 有相同的行为。

使用 `IoBuildAsynchronousFsdRequest`

`IoBuildAsynchronousFsdRequest` 是用于创建表 5-3 中列出的 IRP 的另一个例程。该函数的原型如下：

```
PIRP IoBuildAsynchronousFsdRequest(ULONG MajorFunction,
                                     PDEVICE_OBJECT DeviceObject,
                                     PVOID Buffer,
                                     ULONG Length,
                                     PLARGE_INTEGER StartingOffset,
                                     PIO_STATUS_BLOCK IoStatusBlock);
```

这个原型与 `IoBuildSynchronousFsdRequest` 不同，它没有 `Event` 参数，并且 `IoStatusBlock` 指针可以为 `NULL`。你仍需要为该函数创建的 IRP 安装一个完成例程，这个完成例程的工作就是调用 `IoFreeIrp` 并返回 `STATUS_MORE_PROCESSING_REQUIRED`。

我想知道这两个函数在创建 IRP 上有什么不同之处，因此我稍微深入了一些。这两个函数的代码基本上是相同的。实际上，`IoBuildAsynchronousFsdRequest` 是 `IoBuildSynchronousFsdRequest` 的子函数。

`IoBuildSynchronousFsdRequest` 仅有的额外操作就是保存事件指针到 IRP 中(I/O 管理器需要找到这个事件对象并把它置成信号态)，并把该 IRP 放到当前线程的 IRP 队列中。这可以使 IRP 在线程死亡时能被取消。

你可能在这样两种情况下需要调用 `IoBuildAsynchronousFsdRequest`: 第一种情况，当你发现自己执行在任意线程上下文中并需要创建一个 IRP 时，`IoBuildAsynchronousFsdRequest` 函数就是理想选择，因为当前线程(任意线程)的结束过程不能取消这个新创建的 IRP。第二种情况，当你运行在 APC_LEVEL 级的非任意线程上下文时，你需要同步执行一个 IRP。`IoBuildSynchronousFsdRequest` 不能满足这个要求，因为这种 IRQL 将阻塞设置事件的 APC。所以你应该调用 `IoBuildAsynchronousFsdRequest` 并在某个事件上等待，而这个事件将由你的完成例程去置成信号态。第二种情况不经常出现在设备驱动程序中。

通常情况下，与 `IoBuildAsynchronousFsdRequest` 配合使用的完成例程不仅仅是调用 `IoFreeIrp`。实际上，你需要实现 I/O 管理器用于清除已完成 IRP 的内部例程(`IopCompleteRequest`)。你不能依赖 I/O 管理器来清除 `IoBuildAsynchronousFsdRequest` 创建的 IRP。因为在当前版本的 Windows 98 和 Windows 2000 中清除操作需要一个 APC，并且在任意线程下执行 APC 是错误的，所以 I/O 管理器不能为你做清除工作，你必须自己做全部的清除工作。

如果 IRP 的目标设备对象设置了 DO_DIRECT_IO 标志，`IoBuildAsynchronousFsdRequest` 将创建一个 MDL，这个 MDL 必须由你自己释放，如下面代码：

```
NTSTATUS CompletionRoutine(...)
{
    PMDL mdl;
    while ((mdl = Irp->MdlAddress))
    {
        Irp->MdlAddress = mdl->Next;
```

```
    IoFreeMdl(mdl);
}
...
IoFreeIrp(Irp);
return STATUS_MORE_PROCESSING_REQUIRED;
}
```

如果 IRP 的目标设备对象设置了 DO_BUFFERED_IO 标志，`IoBuildAsynchronousFsdRequest` 将分配一个系统缓冲区，但这个缓冲区需要你来释放。如果你正在做一个输入操作，那么在释放这个缓冲区之前你还需要把输入数据从系统缓冲区复制到你自己真正的输入缓冲区。并且要保证这个真正的缓冲区在非分页内存中，因为完成例程需要在 DISPATCH_LEVEL 级上运行。你还需要确保你得到的是缓冲区的内核地址，因为完成例程运行在任意线程上下文中。如果这些限制还不足以使你在使用 `IoBuildAsynchronousFsdRequest`(用于 DO_BUFFERED_IO 设备)时感到泄气，那么想一想你还必须在完成例程中检测未公开标志位 IRP_BUFFERED_IO、IRP_INPUT_OPERATION、IRP_DEALLOCATE_BUFFER。我不将给出关于这个函数的代码，因为我曾保证过不在本书中使用未公开的内部技术。

我的建议是，仅在你知道 IRP 的目标设备不使用 DO_BUFFERED_IO 标志时，才使用 `IoBuildAsynchronousFsdRequest`。

设备对象指针从哪来？

`IoCallDriver` 函数需要一个 PDEVICE_OBJECT 作为它的第一个参数。你也许想知道我是从哪得到设备对象的指针的。

一个最明显的获得设备对象指针的方式是调用 `IoAttachDeviceToDeviceStack` 函数，这也是每个 WDM 驱动程序的 `AddDevice` 函数应做的一步。在本书的所有例子驱动程序中，你都将看到下面一行代码：

```
pdx->LowerDeviceObject = IoAttachDeviceToDeviceStack(fdo, pdo);
```

任何时候我们需要沿着设备堆栈向下传送一个 IRP 时，我们就使用这个设备对象指针。

另一个常用的定位设备对象的方法是使用对象名称：

```
PUNICODE_STRING DeviceName; // something gives you this
PDEVICE_OBJECT DeviceObject; // an output from this process
PFILE_OBJECT FileObject; // another output
NTSTATUS status = IoGetDeviceObjectPointer(DeviceName, <access mask>, &FileObject, &DeviceObject);
```

通过指定设备对象名称和与其相关的文件对象指针你可以得到这个设备对象的指针。文件对象就是文件句柄指向的对象。最后，你还需要解除文件对象参考，如下：

```
ObDereferenceObject(FileObject); // DeviceObject now poison!
```

解除文件对象参考后，你还要释放设备对象的隐含引用。如果你要继续使用该设备对象，应该先做设备对象引用：

```
ObReferenceObject(DeviceObject);
ObDereferenceObject(FileObject); // DeviceObject still okay
```

然而，你不应该机械地把前两行代码加入到你的驱动程序中。实际上，当你向地址由 `IoGetDeviceObjectPointer` 获得的设备对象发送 IRP 时，你应该带着文件对象：

```
PIRP Irp = IoBuildXxxRequest(...);
PIO_STACK_LOCATION stack = IoGetNextIrpStackLocation(Irp);
stack->FileObject = FileObject;
```

```
IoCallDriver(DeviceObject, Irp);
```

这里是这个额外语句的解释。在内部，`IoGetDeviceObjectPointer` 打开设备对象的一个常规句柄，驱动程序应创建与这个文件对象相关联的某些辅助数据结构，处理 `IRP_MJ_CREATE` 之后的 `IRP` 也许会用到这些结构。处理 `IRP_MJ_CLOSE` 时将销毁这些结构。因此，你需要在发往驱动程序的每个 `IRP` 的第一个堆栈单元中设置 **FileObject** 指针。

你不必总在新 `IRP` 中设置文件对象指针，如果你在 `IRP_MJ_CREATE` 的对应例程中获得了文件对象，那么你下面的驱动程序就没有必要查看该文件对象。而在前面描述的情况下，文件对象的所有者是你用 `IoGetDeviceObjectPointer` 函数获得设备对象的驱动程序，在这种情况下，你必须设置文件对象指针。

不要使用 `IoAttachDevice`

这里没有讨论 `IoAttachDevice` 函数，该函数用于以前版本的 NT 中，通过给出设备名来定位设备对象。该函数有一个可怕的 bug，在该函数返回前它向附着的设备发送一个 `IRP_MJ_CLOSE` 请求。不了解 `IoAttachDevice` 内部实现的附着驱动程序将不能向目标驱动程序传递请求，从而导致一个孤立句柄。此外，如果目标设备碰巧有 `DO_EXCLUSIVE` 属性(如 Windows 2000 中的串行口驱动程序)，则该目标设备的句柄将不能再次打开。

第六章：即插即用

即插即用(Plug and Play -- PnP)管理器使用主功能码为 IRP_MJ_PNP 的 IRP 与设备驱动程序交换信息和请求。这种类型的请求是新引入到 Windows 2000 和 WDM 中的，在以前版本的 Windows NT 中，大部分检测和配置设备的工作由设备驱动程序自己做。而 WDM 驱动程序可以让 PnP 管理器做这个工作。为了与 PnP 管理器协同工作，驱动程序开发者需要了解一些相关的 IRP。

在 WDM 中，PnP 请求扮演了两个角色。在第一个角色中，这些请求指示驱动程序何时以及如何配置或取消其硬件或自身的设置。表 6-1 列出了 PnP 请求可以指定的二十多个副功能码。其中有 9 个副功能码仅能由总线驱动程序处理，以星号标出；过滤器驱动程序或功能驱动程序仅下传这些 IRP。剩下的副功能码中，有三个对过滤器驱动程序或功能驱动程序特别重要。PnP 管理器使用 IRP_MN_START_DEVICE 来通知功能驱动程序其硬件被赋予了什么 I/O 资源，以及指导功能驱动程序做任何必要的硬件或软件设置以便设备能正常工作。
IRP_MN_STOP_DEVICE 告诉功能驱动程序关闭设备。IRP_MN_REMOVE_DEVICE 告诉功能驱动程序关闭设备并释放与之关联的设备对象。我将在本章和下一章详细讨论这三种副功能码；以及其它未用星号标出的，过滤器驱动程序或功能驱动程序会处理的副功能码。

- IRP_MJ_PNP 派遣函数
- 启动和停止设备
- 管理 PnP 状态转换
- 其它配置功能
- Windows 98 兼容问题

表 6-1. IRP_MJ_PNP 的副功能码 (*指出仅由总线驱动程序处理)

IRP 副功能码	描述
IRP_MN_START_DEVICE	配置并初始化设备
IRP_MN_QUERY_REMOVE_DEVICE	设备可以被安全地删除吗？
IRP_MN_REMOVE_DEVICE	关闭并删除设备
IRP_MN_CANCEL_REMOVE_DEVICE	忽略以前的 QUERY_REMOVE
IRP_MN_STOP_DEVICE	关闭设备
IRP_MN_QUERY_STOP_DEVICE	设备可以被安全地关闭吗？
IRP_MN_CANCEL_STOP_DEVICE	忽略以前的 QUERY_STOP
IRP_MN_QUERY_DEVICE_RELATIONS	给出与指定特征相关的设备列表
IRP_MN_QUERY_INTERFACE	获得直接调用函数地址
IRP_MN_QUERY_CAPABILITIES	取设备能力
IRP_MN_QUERY_RESOURCES*	取引导配置
IRP_MN_QUERY_RESOURCE_REQUIREMENTS*	取 I/O 资源需求
IRP_MN_QUERY_DEVICE_TEXT*	获得描述信息或位置串
IRP_MN_FILTER_RESOURCE_REQUIREMENTS	修改 I/O 资源需求列表
IRP_MN_READ_CONFIG*	读配置空间
IRP_MN_WRITE_CONFIG*	写配置空间
IRP_MN_EJECT*	弹出设备
IRP_MN_SET_LOCK*	设备弹出锁定/解除
IRP_MN_QUERY_ID*	取设备硬件 ID
IRP_MN_QUERY_PNP_DEVICE_STATE	取设备状态

IRP_MN_QUERY_BUS_INFORMATION*	取父总线类型
IRP_MN_DEVICE_USAGE_NOTIFICATION	通知分页、dump、睡眠文件被创建或删除
IRP_MN_SURPRISE_REMOVAL	通知设备已经被删除

PnP 请求的第二个角色是指导驱动程序完成一系列状态转换，如图 6-1 所示。WORKING 和 STOPPED 是设备的两个基本状态。当你创建设备对象后，设备就立即进入 STOPPED 状态。WORKING 状态指出设备是全部可操作的。此外，还有两个中间状态，PENDINGSTOP 和 PENDINGREMOVE，它们出现在 WORKING 状态前。SURPRISEREMOVED 发生在物理硬件突然被移去的情况下。

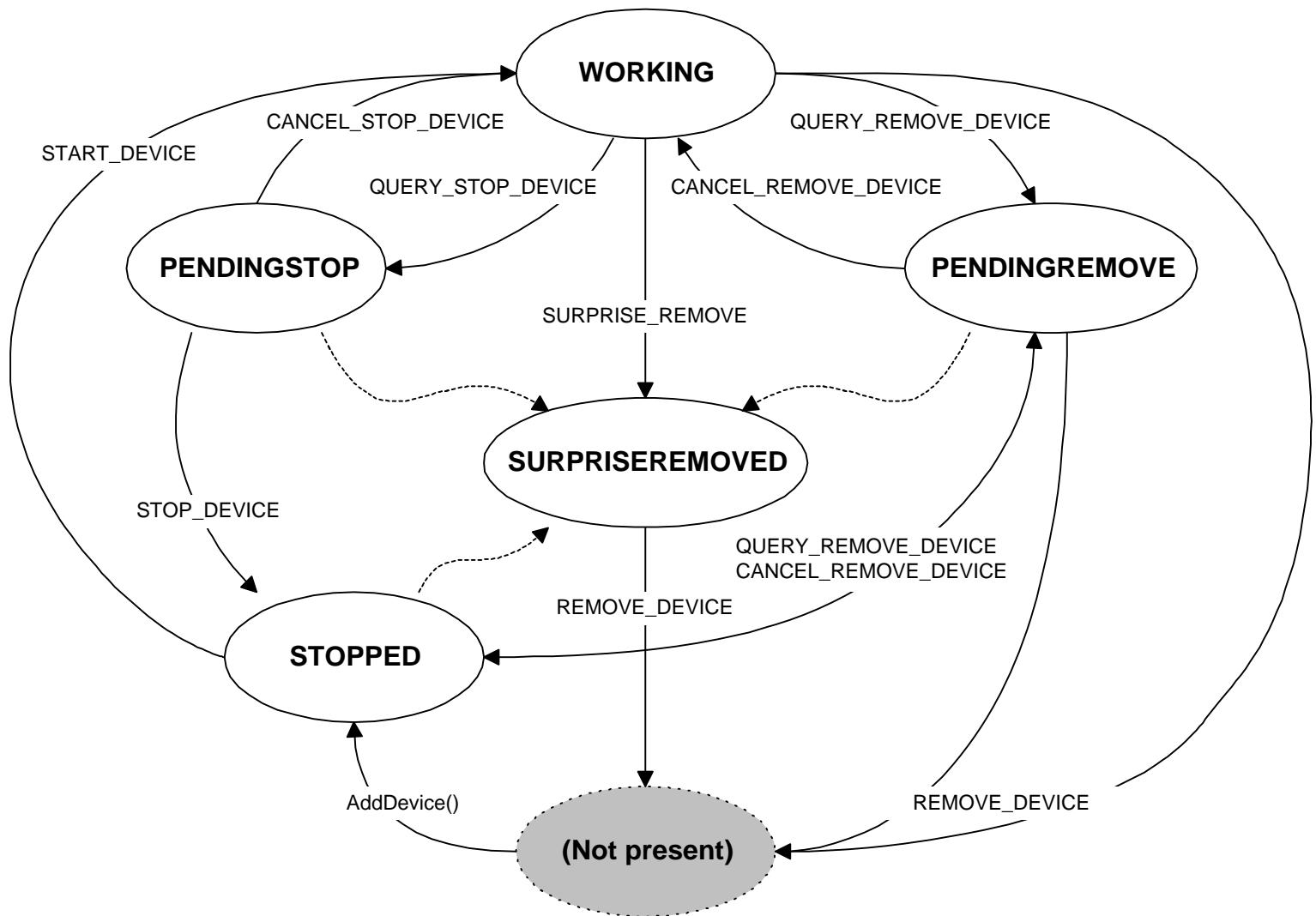


图 6-1. 设备状态图

在上一章描述的 IRP 处理标准模型中，我指出即插即用对 IRP 排队和取消有额外需求。在这章我将描述 DEVQUEUE 对象，它可以满足这些需求，并帮助你管理状态转换。

IRP_MJ_PNP 派遣函数

在第五章我解释了向驱动程序堆栈下层传递 IRP 的两种情况：一种情况你关心 IRP 的结果，因此你需要一个完成例程，另一种情况你不关心 IRP 的结果，因此也不需要安装一个完成例程。有许多 PnP 请求属于第二类，你收到这样的 IRP，然后把它们向下传递，不关心该 IRP 以后的情况。因此，我建议你写一个辅助函数，该函数以“发射后不管”的方式下传 IRP，代码见下面。

```
NTSTATUS DefaultPnpHandler(PDEVICE_OBJECT fdo, PIRP Irp)
{
    IoSkipCurrentIrpStackLocation(Irp);
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    return IoCallDriver(pdx->LowerDeviceObject, Irp);
}
```

一个简化版的 IRP_MJ_PNP 派遣函数看起来象这样：

```
NTSTATUS DispatchPnp(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);           <--1
    ULONG fcn = stack->MinorFunction;                                         <--2

    static NTSTATUS (*fcntab[])(PDEVICE_OBJECT, PIRP) =                         <--3
    {
        HandleStartDevice, // IRP_MN_START_DEVICE
        HandleQueryRemove, // IRP_MN_QUERY_REMOVE_DEVICE
        <etc.>,
    };

    if (fcn >= arraysize(fcntab))                                              <--4
        return DefaultPnpHandler(fdo, Irp);
    return (*fcntab[fcn])(fdo, Irp);                                              <--5
}
```

1. IRP 的所有参数，包括全部重要的副功能码，都在堆栈单元中。因此，我们通过调用 **IoGetCurrentIrpStackLocation** 函数获得一个指向它的指针。
2. 我们期望 IRP 的副功能码是表 6-1 中列出的一个。
3. 处理这二十多个副功能码的一种方法就是为每一个我们将要处理的功能码写一个子派遣函数，并定义一个指向这些子派遣函数的指针表。表中的许多表项将是 **DefaultPnpHandler**。而象 **HandleStartDevice** 的子派遣函数将使用设备指针和 IRP 作为参数，返回 NTSTATUS 状态码。
4. 如果收到一个不认识的副功能码，可能是 DDK 新定义的代码，正确的做法是调用默认处理程序把不认识的功能码沿设备堆栈向下传递。顺便说一下，**arraysize** 是一个宏，它返回数组中的元素个数。其定义为**#define arraysize(p) (sizeof(p)/sizeof((p)[0]))**。
5. 该语句索引指针表中的子派遣函数并调用它。

使用函数指针表

用派遣函数指针表来对应副功能码需要冒一些风险。未来版本的操作系统也许会改变某些代码的含义。尽管在发行版本的操作系统中可以不用担心这一点，但在系统的 **beta** 版测试期间，有些代码可能会不确定。我喜欢使用指针表来代表子派遣函数，因为用分离的函数来代表副功能码看起来更符合软件工程。举例来说，如果我要为此设计一个 C++ 类库，我愿意定义一个基类，把每个副功能码用虚拟函数来代表。

大部分程序员可能在 DispatchPnp 例程中放一个 **switch** 语句。通过简单的重编译你可以适应副功能码的任何重赋值。但重编译有时也不能解决问题，代码名字的改变往往表示代码功能的改变。实际上，在 Windows 98 和 Windows 2000 的 beta 测试中曾发生过一两次这样的事情。另外，一个有优化功能的编译器可能会使用一个转跳表来产生稍微快速的 **switch** 语句代码，而不是调用子派遣函数。

我认为选择 **switch** 语句还是一个函数指针表只是个人偏好，以我看来代码可读性和模块性要比执行效率更重要。为了避免 beta 测试中副功能代码的不确定性，你可以在代码中放置适当的 **ASSERT** 语句。例如，在 HandleStartDevice 函数中加入 **start->MinorFunction == IRP_MN_START_DEVICE** 的断言。如果你用新的 beta 版 DDK 重编译你的驱动程序，你将捕获到代码的任何数值和名称上的更改。

启动和停止设备

在 Windows 2000 和 Windows 98 中，通过使用总线驱动程序，PnP 管理器能够自动检测硬件和分配 I/O 资源。大部分现代设备都有即插即用特征，可以允许系统软件自动检测并提取它们的 I/O 资源需求。但早期的遗留设备不能用电子方式向操作系统标识自身和表达资源需求，不过注册表可以包含检测和资源赋予操作所需要的信息。

注意

我发现给出术语“I/O 资源”的抽象定义十分困难，所以我将以描述具体实例来解释这个术语。WDM 包含四种标准 I/O 资源类型：I/O 端口、内存寄存器、DMA 通道、中断请求。

当 PnP 管理器检测到硬件时，它首先参考注册表以了解有哪些过滤器驱动程序将管理该硬件。正如我在第二章中讨论的，如果必要(某些驱动程序可能因为其它硬件的需要已经被系统装入)它将装入这些驱动程序，并调用它们的 **AddDevice** 函数。最后 **AddDevice** 函数创建设备对象并连入设备堆栈。此后，PnP 管理器将为所有设备驱动程序分配 I/O 资源。

开始，PnP 管理器为每个设备创建一个资源需求列表并允许驱动程序过滤这个列表。在此我将忽略过滤步骤，因为并不是每个驱动程序都需要经过这个步骤。给出需求列表，则 PnP 管理器在分配资源时能协调当前系统中所有硬件的潜在资源冲突。例如，图 6-2 显示了 PnP 管理器如何裁决两个有重叠中断请求号设备的资源需求。

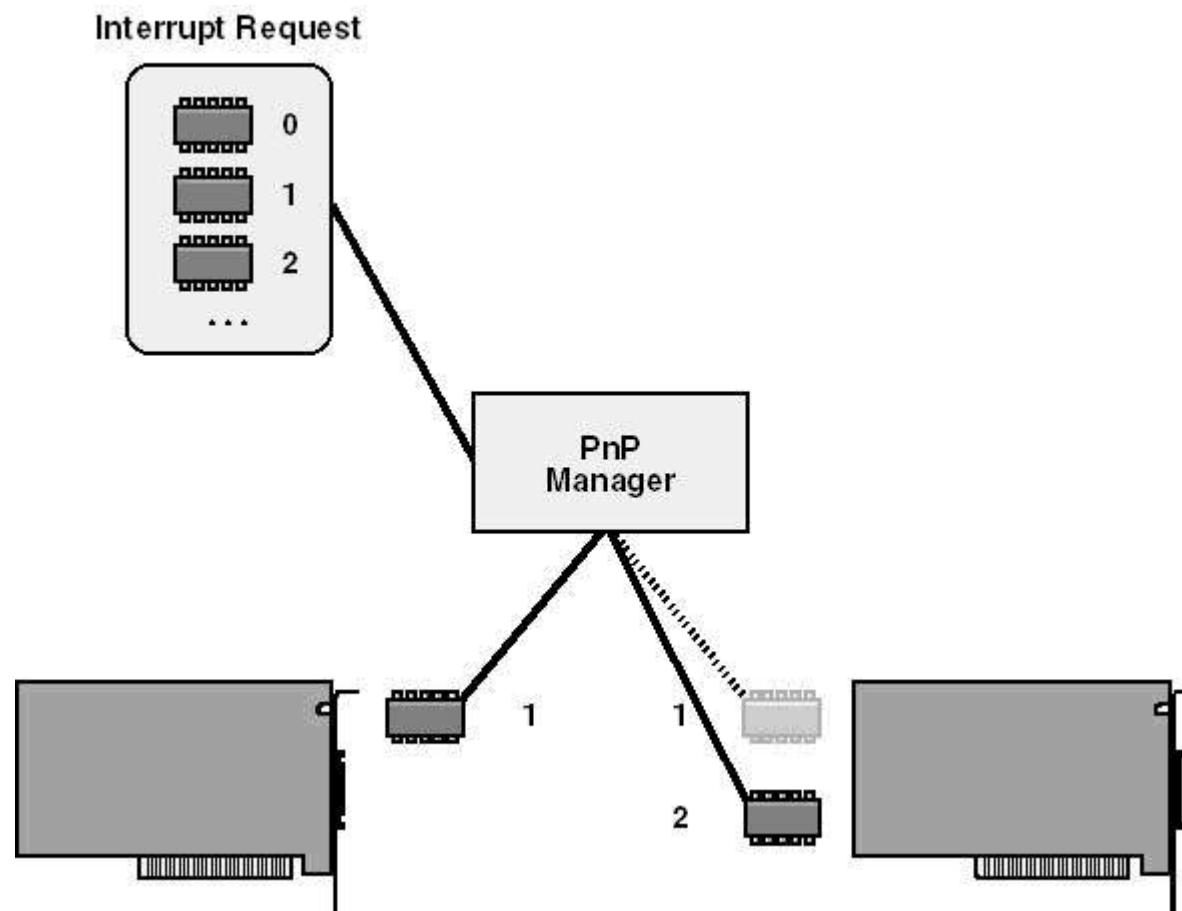


图 6-2. 裁决 I/O 资源冲突

一旦资源分配确定，PnP 管理器通过向每个设备发送一个带 IRP_MN_START_DEVICE 副功能码的 PnP 请求来通知设备。通常过滤器驱动程序对这个 IRP 不感兴趣，所以它们使用 DefaultPnpHandler 方式把请求向下传。而功能驱动程序正好相反，它需要在这个 IRP 上做大量工作，包括分配并配置额外的软件资源以及为设备操作做准备。这个工作需要在 PASSIVE_LEVEL 级上进行，并在低层驱动程序处理完该 IRP 后完成。

前进和等待IRP

为了在下传 `IRP_MN_START_DEVICE` 请求后再获得控制，派遣例程需要等待一个内核事件，该事件最终由低层驱动程序对 `IRP` 的完成操作来通知。在第四章中，我曾警告过你不要阻塞任意线程。但 `PnP` 请求是在系统线程(可以被阻塞)上下文中发送给你的，所以警告是不必要的。因为前进和等待 `IRP` 在其它上下文中也是一个有潜在用途的功能，所以我建议你写一个辅助函数来执行这个机制：

```
NTSTATUS ForwardAndWait(PDEVICE_OBJECT fdo, PIRP Irp)
{
    KEVENT event;                                <--1
    KeInitializeEvent(&event, NotificationEvent, FALSE);
    IoCopyCurrentIrpStackLocationToNext(Irp);      <--2
    IoSetCompletionRoutine(Irp,
        (PIO_COMPLETION_ROUTINE) OnRequestComplete,
        (PVOID) &event,
        TRUE,
        TRUE,
        TRUE);
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    IoCallDriver(pdx->LowerDeviceObject, Irp);      <--4
    KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL); <--5
    return Irp->IoStatus.Status;                  <--6
}
```

1. 我们创建一个内核事件对象。`KeInitializeEvent` 必须在 `PASSIVE_LEVEL` 级上被调用。幸运的是，`PnP` 请求总是在 `PASSIVE_LEVEL` 上发送，所以正好符合这种需求。事件对象本身必须占用非分页内存。另外，在大多数情况下，你也可以认为执行堆栈也是非分页的。
2. 由于我们要安装一个完成例程，所以必须向下一层驱动程序复制堆栈参数。
3. 指定一个完成例程以便我们能知道下层驱动程序何时完成该 `IRP`。我们应该等待完成操作发生，所以必须确保我们的完成例程被调用。这就是为什么我把三个标志参数都指定为 `TRUE`，它们指出我们希望在 `IRP` 正常完成、遇到错误、被取消这三种情况下都调用 `OnRequestComplete`。完成例程的上下文参数是 `event` 对象的地址。
4. `IoCallDriver` 调用下一层驱动程序，可以是一个低层过滤器驱动程序或是 `PDO` 驱动程序本身。`PDO` 驱动程序将执行某些处理，或者是立即完成该请求，或者返回 `STATUS_PENDING`。
5. 不管 `IoCallDriver` 返回什么，我们都将调用 `KeWaitForSingleObject` 在我们以前建立的内核事件上永远等待。当下层驱动程序完成该 `IRP` 并把事件置成信号态时，我们的完成例程将再次获得控制。
6. 这里，我们捕获 `IRP` 的最终状态并返回给我们的调用者。

一旦我们调用了 `IoCallDriver`，我们就放弃了 `IRP` 的控制权，直到某些运行在任意线程上下文中的代码调用 `IoCompleteRequest` 通知该 `IRP` 完成，`IoCompleteRequest` 将调用我们的完成例程。图 6-3 显示了这个过程中的时间顺序。完成例程特别简单：

```
NTSTATUS OnRequestComplete(PDEVICE_OBJECT fdo, PIRP Irp, PKEVENT pev)
{
    KeSetEvent(pev, 0, FALSE);                      <--1
    return STATUS_MORE_PROCESSING_REQUIRED;          <--2
}
```

1. 我们把阻塞 `ForwardAndWait` 的事件置成信号态。
2. 通过返回 `STATUS_MORE_PROCESSING_REQUIRED`，我们停止了 I/O 堆栈的回卷处理。此时，上层过滤器驱动程序安装的任何完成例程都得不到调用，并且 I/O 管理器将停止在该 `IRP` 上的工作。这种情形就象根本没有调用过 `IoCompleteRequest` 一样，当然，某些已经调用过的低级完成例程除外。在这一时刻，该 `IRP` 将处于一个中间状态，但我们的 `ForwardAndWait` 例程将再次获得该 `IRP` 的所有权。

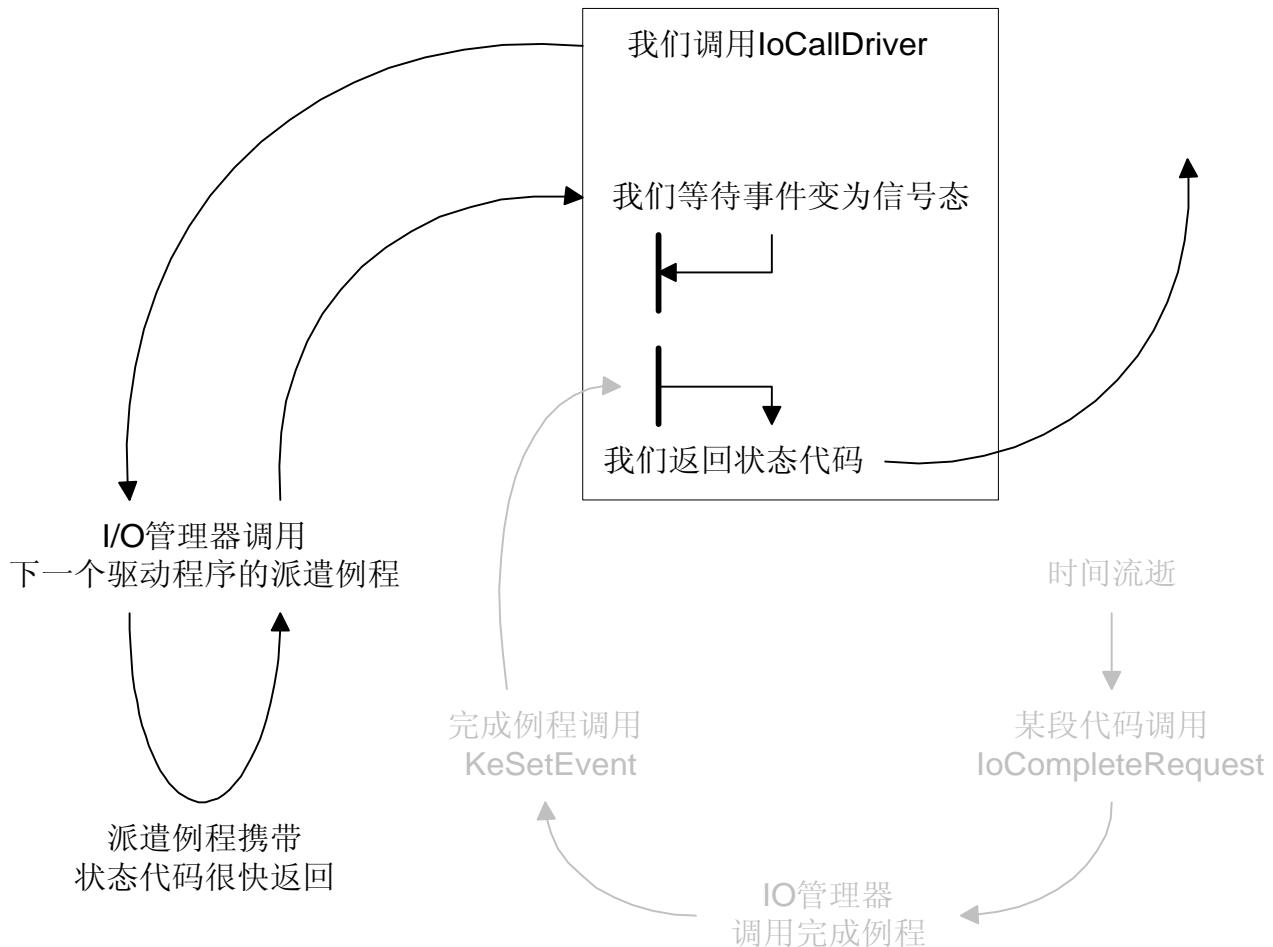


图 6-3. *ForwardAndWait* 的时间序列

关于 *ForwardAndWait*

当我在描述 *ForwardAndWait* 如何与 *OnRequestComplete* 协同工作时，我掩盖了两个细微之处。第一，线程的内核堆栈有时也可能被换出物理内存，这仅发生在线程被阻塞于用户模式中。David Solomon 的《*Inside Windows NT, Second Edition* (Microsoft Press, 1998)》中第 194 页给出了这种可能性的状态图。*ForwardAndWait* 中所有处理事件对象的调用都要求 **event** 对象驻留内存。因为我们指定一个内核模式等待，所以我们的堆栈不能被换出内存，因此 **KeSetEvent** 也会在内存中找到 **event** 对象。

第二，你可能已经注意到了，在完成例程开始处没有了样板代码 **if(Irp->PendingReturned)** **IoMarkIrpPending(Irp)**。如果完成例程返回 **STATUS_MORE_PROCESSING_REQUIRED**，你就不需要那个样板代码。但大部分标准完成例程需要这个样板代码。这也是为什么所有 DDK 例子都包含这个样板代码，尽管这段代码有时并不严格需要。

提取资源分配信息

在前一段中，我解释了如何使用 *ForwardAndWait* 辅助函数发送 **IRP_MN_START_DEVICE** 请求并等待其完成。另外，你还可以从子派遣例程中调用 *ForwardAndWait*，这个子派遣函数就是前面提到的 *DispatchPnp* 派遣函数的一个分支，其框架如下：

```
NTSTATUS HandleStartDevice(PDEVICE_OBJECT fdo, PIRP Irp)
{
    Irp->IoStatus.Status = STATUS_SUCCESS;                                <-1
    NTSTATUS status = ForwardAndWait(fdo, Irp);
    if (!NT_SUCCESS(status))                                              <-2
        return CompleteRequest(Irp, status, Irp->IoStatus.Information);
```

```

PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);           <--3
status = StartDevice(fdo, <additional args>);                         <--4
return CompleteRequest(Irp, status, Irp->IoStatus.Information);
}

```

1. 总线驱动程序利用 **IoStatus.Status** 中的设置来判断上层驱动程序是否已经处理了该 IRP。对于 **IRP_MJ_PNP** 的其它几种副功能码，总线驱动程序也做类似的判断。因此，我们需要把 IRP 的 Status 域初始化成 **STATUS_SUCCESS**，然后再传递下去。
2. **ForwardAndWait** 返回一个状态码。如果这个状态码指出低层驱动程序的某种失败，我们把这个失败代码返回给我们的调用者。因为我们的完成例程返回 **STATUS_MORE_PROCESSING_REQUIRED**，它在 **IoCompleteRequest** 内部停止了完成处理。因此，我们必须再一次完成该请求。
3. 我们需要的配置信息隐藏在堆栈参数中，我将在后面继续讨论。
4. **StartDevice** 是你写的用于提取和处理配置信息的辅助函数。在我的例子驱动程序中，我把它放到源文件 **READWRITE.CPP** 中。设备对象后面的参数我稍后再解释。

你也许会猜到，**IRP_MN_START_DEVICE** 处理程序的工作涉及到设备从 **STOPPED** 状态转换到 **WORKING** 状态。现在我还不能解释这些，我需要先解释 PnP 请求关于状态转换、IRP 排队、IRP 取消的其它分支。所以，我将在 PnP 请求的配置方面上停留一会。

I/O 堆栈单元的 **Parameters** 联合有一个名为 **StartDevice** 的子结构，该结构包含的配置信息，见表 6-2，将被传递到 **StartDevice** 辅助函数。

表 6-2. *IO_STACK_LOCATION.Parameters.StartDevice* 子结构中的域

域名	描述
AllocatedResources	包含原始的资源分配信息
AllocatedResourcesTranslated	包含转换后的资源分配信息

AllocatedResources 和 **AllocatedResourcesTranslated** 都是同一种类数据结构的实例，这种数据结构就是 **CM_RESOURCE_LIST**。如果你看到该结构在 **WDM.H** 中的声明，你可能认为它是一个十分复杂的数据结构。然而，当用于启动设备的 IRP 时，仅有一个表项是有用的，即 **CM_PARTIAL_RESOURCE_LIST**，它描述了设备被赋予的所有 I/O 资源。

可以使用下面语句访问这两个列表：

```

PCM_PARTIAL_RESOURCE_LIST raw, translated;
raw = &stack->Parameters.StartDevice.AllocatedResources->List[0].PartialResourceList;
translated = &stack->Parameters.StartDevice.AllocatedResourcesTranslated->List[0].PartialResourceList;

```

raw 和 **translated** 资源列表是 **StartDevice** 辅助函数的实际参数：

```

status = StartDevice(fdo, raw, translated);

```

因为 I/O 总线与 CPU 在寻址物理硬件的方式上不同，所以存在着两种资源列表。**raw** 资源包含总线相关的数值，而 **translated** 资源包含系统相关的数值。在 WDM 出现之前，内核模式驱动程序从注册表、PCI 配置空间、或其它地方获取 **raw** 资源值，并通过调用诸如 **HalTranslateBusAddress** 或 **HalGetInterruptVector** 函数转换这些数值。参见 Art Baker 的《*The Windows NT Device Driver Book: A Guide for Programmers* (Prentice Hall, 1997)》第 122-162 页。现在，接收和转换工作全部由 PnP 管理器来完成，WDM 驱动程序需要做的仅是访问设备启动 IRP 的 **Parameters** 结构。

StartDevice 函数中对资源信息的实际操作是下一章的主题。

IRP_MN_STOP_DEVICE

设备停止请求通知你关闭设备，然后 PnP 管理器重新分配 I/O 资源。在硬件级，关闭设备将包括暂停或停止当前活动并阻止后来的中断。在软件级，关闭设备将涉及释放设备启动时配置的 I/O 资源。基于派遣例程/子派遣例程架构，你应该有一个如下面这样的子派遣函数：

```
NTSTATUS HandleStopDevice(PDEVICE_OBJECT fdo, PIRP Irp)
{
    <complicated stuff>                                <-1
    StopDevice(fdo, oktouch);                            <-2
    Irp->IoStatus.Status = STATUS_SUCCESS;
    return DefaultPnpHandler(fdo, Irp);                  <-3
}
```

1. 在这里，你需要插入一些代码来处理 IRP 的排队和取消。在本章后面的“当设备停止时”段中我将给出这里的代码。
2. 对照设备启动阶段，在那时我们先向下传递请求，然后做设备相关的工作。但在这里，我们应该先做设备相关的工作，然后再向下传递请求。这里的思想是，当低层驱动程序看到这个请求时我们的硬件已经停止了活动。我写了一个名为 **StopDevice** 的辅助函数来做设备关闭工作，该函数的第二个参数指出它可在必要时接触硬件。参考文字框“停止设备时接触硬件”中的解释来设置这个参数。
3. 我们总是向下传递 PnP 请求。在这里，我们不需要关心低层驱动程序如何处理该 IRP，所以我们简单地使用 **DefaultPnpHandler** 向下传递该请求。

上面例子中对 **StopDevice** 辅助函数的调用是与 **StartDevice** 中配置步骤功能相反的重要代码。我将在下一章解释 **StartDevice** 函数，关于这个函数有一个重要的事实，即该函数可以被多次单独调用。对于 PnP 请求的处理例程，知道是否已经调用过 **StopDevice** 函数并不总是容易的，但使用 **StopDevice** 检测该函数的重复调用还是比较容易的。

停止设备时接触硬件

在 **HandleStopDevice** 框架代码中，我用了一个 **oktouch** 变量，但没有描述它如何初始化。在本书介绍的驱动程序编写方案中，**StopDevice** 函数将得到一个 **BOOLEAN** 参数，该参数指出能否对硬件执行安全的 I/O 操作。这个参数背后的想法是，你可能需要向设备发送自定义的指令，但在某些情况下例外。例如，你可以向 PCMCIA 调制解调器发出挂断命令，尽管用户这时已经从计算机上拔出了调制解调器卡。

没有确切的方法知道你的硬件是否物理地连接到计算机上，除非你真正访问它。然而 Microsoft 推荐，如果你成功地处理了 **START_DEVICE** 请求，那么在你处理 **STOP_DEVICE** 请求和其它某些 PnP 请求时你可以访问你的硬件。当我在后面讨论如何跟踪 PnP 状态改变时，如果我们确信设备当前处于工作状态，我们应设置 **oktouch** 参数为 **TRUE**，反之为 **FALSE**。

IRP_MN_REMOVE_DEVICE

回想一下，PnP 管理器通过调用驱动程序中的 **AddDevice** 函数来通知你已经找到你要管理的硬件实例，并给你一个创建设备对象的机会。当设备将要被系统删除时，PnP 管理器向你发送副功能码为 **IRP_MN_REMOVE_DEVICE** 的 PnP 请求(不是调用函数做补充操作)。为了响应这个请求，你应该做与 **IRP_MN_STOP_DEVICE** 相同事情，关闭你的设备，然后删除设备对象：

```
NTSTATUS HandleRemoveDevice(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION)fdo->DeviceExtension;
    <complicated stuff>
    DeregisterAllInterfaces(pdx);
    StopDevice(fdo, oktouch);
    Irp->IoStatus.Status = STATUS_SUCCESS;
    NTSTATUS status = DefaultPnpHandler(fdo, Irp);
    RemoveDevice(fdo);
```

```
    return status;  
}
```

这段代码看起来与 `HandleStopDevice` 非常相似，除了两条附加的语句。`DeregisterAllInterfaces` 将禁止所有(可能在 `AddDevice` 中寄存的，在 `StartDevice` 中允许的)设备接口，并释放它们的符号连接名所占用的内存。`RemoveDevice` 将撤消你在 `AddDevice` 中所做的所有工作。

```
VOID RemoveDevice(PDEVICE_OBJECT fdo)  
{  
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;  
    IoDetachDevice(pdx->LowerDeviceObject);  
    IoDeleteDevice(fdo);  
}
```

1. `IoDetachDevice` 调用与 `AddDevice` 中的 `IoAttachDeviceToDeviceStack` 调用正好相反。
2. `IoDeleteDevice` 调用与 `AddDevice` 中的 `IoCreateDevice` 调用正好相反。一旦该函数返回，设备对象将不复再存在。如果你的驱动程序没有其它设备需要管理，那么很快你的驱动程序也将被从内存中卸载。

当低层驱动程序仍在处理 `IRP_MN_REMOVE_DEVICE` 请求时，调用 `IoDeleteDevice` 函数会不会出现麻烦？这样做是无害的，因为对象管理器为你的设备对象维护一个引用计数器，这可以防止设备对象在有任何活动指针指向它时被意外删除。

注意，你不能在删除设备请求后再希望获得停止设备请求。删除设备请求潜在地包含了停止设备操作，所以在这个请求中你应该做两份工作。

IRP_MN_SURPRISE_REMOVAL

有时用户可能不经过任何用户接口交互操作突然地拆卸设备。如果系统检测到这种突然的删除，它就向驱动程序发送副功能码为 `IRP_MN_SURPRISE_REMOVAL` 的 PnP 请求，后面还会跟着一个 `IRP_MN_REMOVE_DEVICE` 请求。除非你以前在处理 `IRP_MN_QUERY_CAPABILITIES` 时曾设置 `SurpriseRemovalOK` 标志(见第八章)，否则系统将显示一个对话框，通知用户这样做是危险的。

为了响应突然删除请求，设备驱动程序应该禁止所有已寄存的接口。这将给应用程序一个机会关闭设备的句柄，但应用程序必须事先关注这种通知，我将在后面的“PnP 通知”中再讨论这些，然后驱动程序应释放 I/O 资源并向上传递请求：

```
NTSTATUS HandleSurpriseRemoval(PDEVICE_OBJECT fdo, PIRP Irp)  
{  
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;  
    <complicated stuff>  
    EnableAllInterfaces(pdx, FALSE);  
    StopDevice(fdo, oktouch);  
    Irp->IoStatus.Status = STATUS_SUCCESS;  
    return DefaultPnpHandler(fdo, Irp);  
}
```

IRP_MN_SURPRISE_REMOVAL 来自何处？

简单且直接地从计算机上拆卸设备并不能产生突然删除 PnP 通知。有些总线能够察觉设备消失。例如，拔掉一个 USB 设备将产生一个电信号，这个电信号能被总线驱动程序注意到。然而，对于大多数其它总线类型，没有任何信号能用于通知总线驱动程序。因此，PnP 管理器需要依靠其它方法来判断设备是否消失。

功能驱动程序可以通知其管理的设备的消失(如果它知道)，通过调用 `Io InvalidateDeviceState` 函数，然后从紧接着的 `IRP_MN_QUERY_PNP_DEVICE_STATE` 中返回 `PNP_DEVICE_FAILED`。

PNP_DEVICE_REMOVED、PNP_DEVICE_DISABLED 中的任何一个值。举例来说，如果你的 ISR 在读通常结果为 1 和 0 混合的状态端口时突然得到了全部为 1 的值，你的驱动程序就应该通知设备消失。更普通的情况是，总线驱动程序调用 **Io InvalidateDeviceRelations** 函数触发一个再枚举操作时报告某个设备枚举失败。另外，如果系统处于休眠或在其它低电源状态时用户拆卸了设备，那么驱动程序在接收到 IRP_MN_SURPRISE_REMOVAL 请求前先收到了一系列电源管理 IRP。

这些事实意味着什么，实际地说，就是你的驱动程序应该能应付由设备突然消失所造成错误。

管理PnP状态转换

正如本章开头提到的，WDM 需要跟踪设备的状态转换。状态跟踪还涉及到如何排队和取消 IRP。而取消操作需要用到一个全局取消自旋锁，但该锁在多处理器机器上会造成性能瓶颈。IRP 处理的标准模型不能解决所有这些问题。因此，在这一节中，我将介绍一种新的对象类型，DEVQUEUE，你可以在 PnP 请求处理中使用这种对象，它可以代替标准模型中的 **StartPacket** 和 **StartNextPacket** 例程。DEVQUEUE 是我自己的创造，使用它的例子驱动程序有 PNPPOWER 和 CANCEL。关于 IRP 取消的其它讨论见 Ervin Peretz 的文章“*The Windows Driver Model Simplifies Management of Device Driver I/O Requests*”(Microsoft Systems Journal, January 1999)。我描述的一部分 IRP 取消逻辑还来自 Peretz 和其它 Microsoft 雇员的工作。

我以前描述的 KDEVICE_QUEUE 队列对象有三种状态：idle、busy-empty、busy-not empty。用于维护 KDEVICE_QUEUE 的支持例程假定设备当前不忙，你所做的仅是启动设备可执行的新请求。而这种行为正是我们管理 PnP 状态时所要克服的。图 6-4 显示了 DEVQUEUE 的状态。

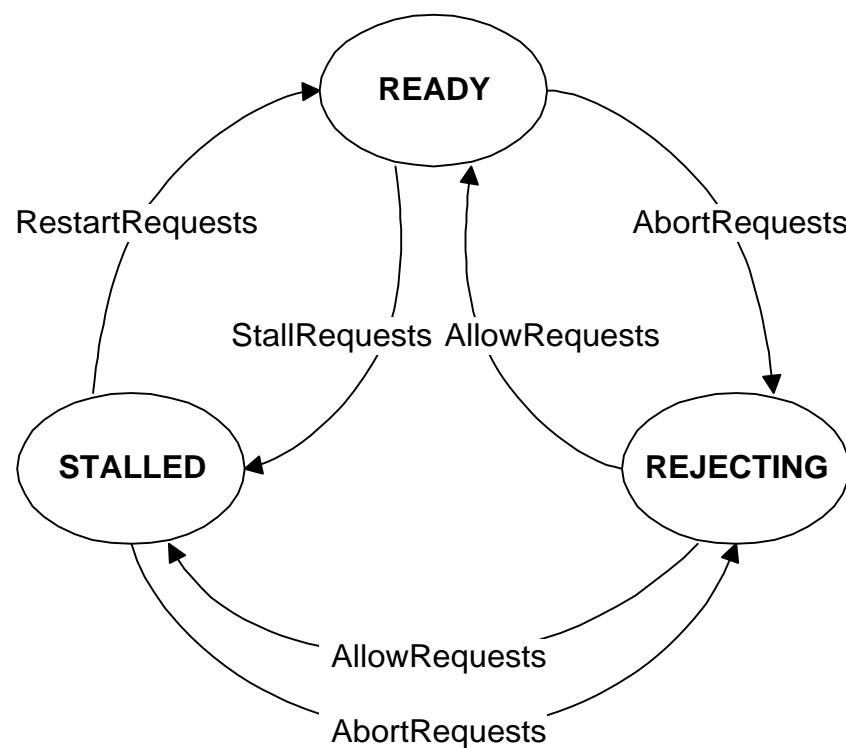


图 6-4. DEVQUEUE 对象的状态

在 READY 状态中，队列的操作更像一个 KDEVICE_QUEUE，它接收并发送请求到你的 StartIo 例程。在 STALLED 状态中，队列停止向 StartIo 运送 IRP，即使设备处于空闲状态。在 REJECTING 状态，队列不接受新的 IRP。图 6-5 显示了穿过队列的 IRP 流。

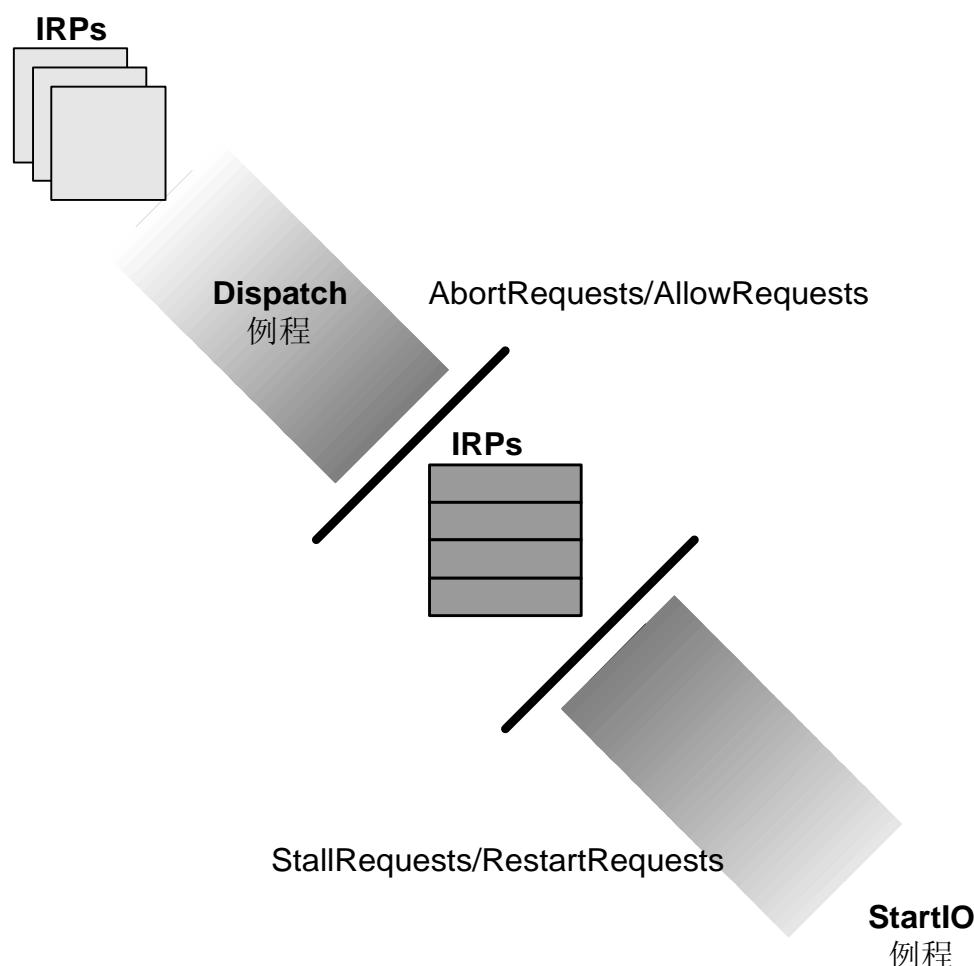


图 6-5. DEVQUEUE 中的 IRP 流

使用DEVQUEUE来排队和取消IRP

你可以为驱动程序要管理的每个请求队列定义一个 **DEVQUEUE** 对象。例如，如果你的设备用一个单独的队列来管理读写请求，你应该定义一个 **DEVQUEUE**：

```
typedef struct _DEVICE_EXTENSION {
    ...
    DEVQUEUE dqReadWrite;
    ...
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

表 6-3 列出了 **DEVQUEUE** 的支持函数

表 6-3. **DEVQUEUE** 的服务函数

支持函数	描述
AbortRequests	放弃当前及将来的请求
AllowRequests	允许请求，与 AbortRequests 相反
AreRequestsBeingAborted	检查新请求是否被取消
CancelRequest	普通的取消例程
CheckBusyAndStall	用一个原子操作检测空闲和停止
CleanupRequests	取消给定文件对象的所有请求，用于处理 IRP_MJ_CLEANUP
GetCurrentIrp	取当前正被处理的请求
InitializeQueue	初始化 DEVQUEUE 对象

RestartRequests	重启动一个停止的队列
StallRequests	停止队列
StartNextPacket	出队并启动下一个请求
StartPacket	启动或排队一个新请求
WaitForCurrentIrp	等待当前 IRP 完成

现在, 我将讨论用于替代标准 IRP 处理模型中 StartPacket 和 StartNextPacket 函数的支持例程。对于每个队列, 都应该提供一个独立的 StartIo 例程。DriverEntry 例程将不在驱动程序对象的 DriverStartIo 指针域存储任何东西。相反, 应该在 AddDevice 例程中初始化你的队列对象:

```
NTSTATUS AddDevice(...)
{
    ...
    PDEVICE_EXTENSION pdx = ...;
    InitializeQueue(&pdx->dqReadWrite, StartIo);
    ...
}
```

对于使用 DEVQUEUE 队列的 IRP, 其派遣函数应该使用下面模式:

```
NTSTATUS DispatchWrite(PDEVICE_OBJECT fdo, PIRP Irp)
{
    <some power management stuff you haven't heard about yet>
    IoMarkIrpPending(Irp);
    StartPacket(&pdx->dqReadWrite, fdo, Irp, OnCancel);
    return STATUS_PENDING;
}
```

即不调用 **IoStartPacket**, 而调用队列的 StartPacket 函数, 其参数为队列对象地址、设备对象、IRP、取消例程。在派遣例程的开始处, 你还要花一点代码来处理电源恢复问题, 这将在第八章中讨论。

下面是用于 DEVQUEUE 的新 **StartIo** 例程:

```
VOID StartIo(PDEVICE_OBJECT fdo, PIRP Irp)
{
    <some PnP stuff you haven't heard about yet>
    // start request on device
}
```

StartIo 不必担心 IRP 的取消。在这里使用的取消例程与标准取消例程不同, 它只是简单地把所有工作都委托给 DEVQUEUE 来处理:

```
VOID OnCancel(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    CancelRequest(&pdx->dqReadWrite, Irp);
}
```

CancelRequest 将释放全局取消自旋锁, 然后以线程安全和多处理器安全的方式取消 IRP。

在请求完成时使用的 DPC 例程也与标准模型中的 DPC 例程稍有不同, 见下面代码:

```
VOID DpcForIsr(PKDPC Dpc, PDEVICE_OBJECT device, PIRP junk, PVOID context)
```

```

{
    PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
    ...
    StartNextPacket(&pdx->dqReadWrite, device);
    <some PnP stuff you haven't heard about yet>
    CompleteRequest(Irp, ...);
}

```

就象 **IoStartNextPacket** 函数, **StartNextPacket** 函数也从队列中删除下一个 IRP 并把它发送到你的(这种队列专用)StartIo 例程。它也返回 IRP 地址, 如果返回地址为 NULL 则代表该 IRP 因为某种原因被取消或放弃, 所以试图完成这样的 IRP 是不正确的。你可以调用 **GetCurrentIrp** 来获得正在完成中的 IRP 的地址, 不要使用来自 DPC 例程第三个参数的 IRP 指针, 为此我命名该参数为 **junk** 以强化这一点。

DEVQUEUE 也简化了 **IRP_MJ_CLEANUP** 的处理。实际的代码很短小:

```

NTSTATUS DispatchCleanup(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    CleanupRequests(&pdx->dqReadWrite, stack->FileObject, STATUS_CANCELLED);
    return CompleteRequest(Irp, STATUS_SUCCESS, 0);
}

```

用**DEVQUEUE**排队**PnP**请求

使用 **DEVQUEUE** 代替 **KDEVICE_QUEUE** 的关键原因是 **DEVQUEUE** 能简化 PnP 状态转换的管理。在我所有的例子驱动程序中, 设备扩展都含有一个状态变量 **state**。我还定义了一个枚举变量 **DEVSTATE**, 其值对应 PnP 状态。当你在 **AddDevice** 中初始化设备对象时, 你应为每个设备队列调用 **InitializeQueue** 函数并指定设备为 **STOPPED** 状态:

```

NTSTATUS AddDevice(...)
{
    ...
    PDEVICE_EXTENSION pdx = ....;
    InitializeQueue(&pdx->dqRead, StartIoReadWrite);
    pdx->state = STOPPED;
    ...
}

```

AddDevice 返回后, 系统发出各种 **IRP_MJ_PNP** 请求, 这些请求将指导设备进入各种 PnP 状态。

启动设备

刚初始化的 **DEVQUEUE** 将处于 **STALLED** 状态, **StartPacket** 调用会把一个请求排入队列, 不管设备是否处于空闲状态。你应该使队列保持 **STALLED** 状态直到你成功地处理了 **IRP_MN_START_DEVICE**, 如下面代码:

```

NTSTATUS HandleStartDevice(...)
{
    status = StartDevice(...);
    if (NT_SUCCESS(status))
    {
        pdx->state = WORKING;
        RestartRequests(&pdx->dqReadWrite, fdo);
    }
}

```

```
    }  
}
```

你先把设备的当前状态记录为 WORKING，然后为每个队列调用 **RestartRequests** 函数，该函数将释放放在 **AddDevice** 运行后和你收到 **IRP_MN_START_DEVICE** 前所到达的所有 IRP。

可以停止设备吗？

PnP 管理器在停止你的设备前总是先询问，得到允许后才向你发送 **IRP_MN_STOP_DEVICE** 请求。询问以 **IRP_MN_QUERY_STOP_DEVICE** 请求的形式出现，你可以回答成功或失败。询问的基本含义是，“如果系统在几纳秒后向你发送 **IRP_MN_STOP_DEVICE**，你能立即停止设备吗？”你可以以两种稍微不同的方式处理这个询问请求。第一种方式适合于可以迅速完成或者能容易地中途结束的 IRP：

```
NTSTATUS HandleQueryStop(PDEVICE_OBJECT fdo, PIRP Irp)  
{  
    Irp->IoStatus.Status = STATUS_SUCCESS;  
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;  
    if (pdx->state != WORKING)  
        return DefaultPnpHandler(fdo, Irp);  
    if (!OkayToStop(pdx))  
        return CompleteRequest(Irp, STATUS_UNSUCCESSFUL, 0);  
    StallRequests(&pdx->dqReadWrite);  
    WaitForCurrentIrp(&pdx->dqReadWrite);  
    pdx->state = PENDINGSTOP;  
    return DefaultPnpHandler(fdo, Irp);  
}
```

1. 该语句用于处理引导设备这种特殊情况：如果你还没有完成设备初始化，PnP 管理器将向你发送 **QUERY_STOP**。你希望忽略这样的查询，这等于对 PnP 管理器说“是”。
2. 在此，你执行某种调查函数，看是否可以回复到 **STOPPED** 状态。我们马上就要讨论这个问题。
3. **StallRequests** 把 **DEVQUEUE** 设置成 **STALLED** 状态，以便任何新的 IRP 能进入队列。**WaitForCurrentIrp** 等待当前的请求完成。使设备停止有两个步骤，这两个步骤执行后我们就可以知道设备是否真的停止或仍在活动。
4. 在此，我们没有理由犹豫，所以我们把设备的状态记录为 **PENDINGSTOP**，然后把请求下传，这样其它驱动程序就有机会接受或拒绝这个查询。

另一种处理 **QUERY_STOP** 的方式适合于需要长时间才能完成并且不能被中途停止的 IRP，例如磁带机的备份操作就不能被中途打断。在这种情况下，你可以使用 **DEVQUEUE** 的 **CheckBusyAndStall** 函数。如果你的设备忙，该函数返回 **TRUE**，因此，你将以 **STATUS_UNSUCCESSFUL** 回答 **QUERY_STOP** 查询。如果设备空闲，该函数返回 **FALSE**，在这种情况下，你还需要停止队列。(检测设备状态和停止队列操作需要一个自旋锁的保护，这就是为什么我要先写这个函数)

失败一个设备停止查询可以有多种原因。例如，用于分页机制的磁盘设备不能被停止。保存睡眠文件或 **dump** 文件的设备也不能被停止。(这些特征可以从 **IRP_MN_DEVICE_USAGE_NOTIFICATION** 请求中得到，我将在后面的“其它配置功能”中讨论这个请求) 此外还有其它原因。

即使你成功地回答了查询，但下层驱动程序可能会失败这个查询。即使所有驱动程序都成功地回答了查询，PnP 管理器也可能决定不关闭你的设备。在这种情况下，你将收到另一个副功能码为 **IRP_MN_CANCEL_STOP_DEVICE** 的 PnP 请求，它通知设备不将被关闭。之后你应该清除在查询中设置的任何 **state** 值：

```
NTSTATUS HandleCancelStop(PDEVICE_OBJECT fdo, PIRP Irp)  
{  
    Irp->IoStatus.Status = STATUS_SUCCESS;  
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;  
    if (pdx->state != PENDINGSTOP)
```

```

    return DefaultPnpHandler(fdo, Irp);
NTSTATUS status = ForwardAndWait(fdo, Irp);
if (NT_SUCCESS(status))
{
    pdx->state = WORKING;
    RestartRequests(&pdx->dqReadWrite, fdo);
}
return CompleteRequest(Irp, status, Irp->IoStatus.Information);
}

```

我们首先查看是否有一个未处理的停止操作。某些高级驱动程序可以否决一个查询，所以我们将看不到这样的查询，因此我们仍处于 **WORKING** 状态。如果我们没有处于 **PENDINGSTOP** 状态，我们就简单地下传该 IRP。否则，我们发送 **CANCEL_STOP** 请求来同步低级驱动程序。即我们用自己的 **ForwardAndWait** 辅助函数来下传该 IRP 并等待其完成。我们等待低级驱动程序是因为我们要继续处理该 IRP，并且低级驱动程序在我们向它发送 IRP 前也许会有工作要做。如果低级驱动程序成功地处理了 **IRP_MN_CANCEL_STOP_DEVICE**，我们就改变 **state** 变量为 **WORKING** 状态，并且调用 **RestartRequests** 函数重新启动队列。

当设备停止时

如果所有设备驱动程序都成功地回答了查询并且 PnP 管理器也决定要关闭你的设备，你将收到一个 **IRP_MN_STOP_DEVICE** 请求。你的子派遣函数应该象这样：

```

NTSTATUS HandleStopDevice(PDEVICE_OBJECT fdo, PIRP Irp)
{
    Irp->IoStatus.Status = STATUS_SUCCESS;
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    if (pdx->state != PENDINGSTOP);                                     <--1
    {
        <complicated stuff>
    }
    StopDevice(fdo, pdx->state == WORKING);                                <--2
    pdx->state = STOPPED;                                                 <--3
    return DefaultPnpHandler(fdo, Irp);                                       <--4
}

```

1. 我们希望系统在发送 **STOP** 请求前发送一个 **QUERY_STOP**，因此我们的设备能够提前进入 **PENDINGSTOP** 状态并且让设备的所有队列都停止。然而，Windows 98 中有一个 bug，当我们需要 **REMOVE** 请求时，有时会得到 **STOP**，而这个 **STOP** 之前却没有 **QUERY_STOP** 请求。因此，你必须采取一些动作以避免驱动程序拒绝任何新 IRP，当收到一个 **REMOVE** 请求时，你不要真的删除你的设备对象或其它与真正 **REMOVE** 相关的动作。
2. **StopDevice** 是一个辅助函数，我们已经在取消设备配置时讨论过。
3. 现在我们进入 **STOPPED** 状态。这与 **AddDevice** 刚执行完的情形几乎相同。即所有的队列都停止，设备没有 I/O 资源。仅有的不同是我们注册的接口仍然是允许的，这意味着应用程序还没有接到设备删除通知并且它们手中的设备句柄还是打开的。在这种情况下，应用程序仍然能打开新的设备句柄。但并不要紧，因为停止状态不会持续多长时间。
4. 正如我们以前讨论过的，处理 **IRP_MN_STOP_DEVICE** 的最后一件事就是向下层驱动程序传递该请求。

可以删除设备吗？

与 PnP 管理器在停止设备前向你询问一样，它在删除设备前也向你询问，即 **IRP_MN_QUERY_REMOVE_DEVICE** 请求。你可以回答成功或失败。与停止查询相似，如果 PnP 管理器中途改变想法，它就发送 **IRP_MN_CANCEL_REMOVE_DEVICE** 请求。

```

NTSTATUS HandleQueryRemove(PDEVICE_OBJECT fdo, PIRP Irp)
{
    Irp->IoStatus.Status = STATUS_SUCCESS;
}

```

```

PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
if (OkayToRemove(fdo))                                     <--1
{
    StallRequests(&pdx->dqReadWrite);                      <--2
    WaitForCurrentIrp(&pdx->dqReadWrite);
    pdx->prevstate = pdx->state;                           <--3
    pdx->state = PENDINGREMOVE;
    return DefaultPnpHandler(fdo, Irp);
}
return CompleteRequest(Irp, STATUS_UNSUCCESSFUL, 0);
}

NTSTATUS HandleCancelRemove(PDEVICE_OBJECT fdo, PIRP Irp)
{
    Irp->IoStatus.Status = STATUS_SUCCESS;
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    if (pdx->state != PENDINGREMOVE)                         <--4
        return DefaultPnpHandler(fdo, Irp);
    NTSTATUS status = ForwardAndWait(fdo, Irp);
    if (NT_SUCCESS(status))
    {
        pdx->state = pdx->prevstate;                        <--5
        if (pdx->state == WORKING)
            RestartRequests(&pdx->dqReadWrite, fdo);
    }
    return CompleteRequest(Irp, status, Irp->IoStatus.Information);
}

```

1. **OkayToRemove** 辅助函数回答这样的问题，“这个设备可以删除吗？”通常，这个回答将包含某些专用设备的成分，如设备是否存储分页文件或休眠文件，等等。
2. 就象 **IRP_MN_QUERY_STOP_DEVICE** 一样，你还需要停止队列，并且如果需要，你还要等待一小段时间，直到当前请求完成。
3. 如果你仔细观察图 6-1，你会注意到当设备处于 **WORKING** 或 **STOPPED** 状态时驱动程序有可能收到一个 **QUERY_REMOVE**。正确的做法是返回到原来的状态。所以我在设备扩展中放了一个 **prestate** 变量，它记录查询前的设备状态。
4. 如果我们上面或下面的驱动程序否决了 **QUERY_REMOVE**，则我们将收到 **CANCEL_REMOVE** 请求。如果我们根本就没有看到该查询，我们将仍旧处于 **WORKING** 状态并且不需要对该 **IRP** 做任何事。否则，我们就必须在处理前把该 **IRP** 发送到低级驱动程序，因为我们希望低级驱动程序在我们处理删除请求前做好准备。
5. 在这里，如果我们成功地回答了 **QUERY_REMOVE**，我们就恢复做过的步骤，回复到以前的状态。如果以前的状态是 **WORKING**，则还要启动在处理查询时被停止的队列。

同步删除

有时候 I/O 管理器发出的 PnP 请求会与其它 I/O 请求(如包含读写的请求)同时出现。这完全有可能，例如当你处理其它 IRP 时收到了 **IRP_MN_REMOVE_DEVICE** 请求。你必须自己避免这种麻烦产生，标准的做法是使用一个 **IO_REMOVE_LOCK** 对象和几个相关的内核模式支持例程。

防止设备被过早地删除的基本想法是在每一次开始处理请求时都获取删除锁，处理完成后释放删除锁。在你删除你的设备对象前，应确保删除锁未被使用。否则，你将等到这个锁的所有引用都被释放。图 6-6 显示了这个过程。

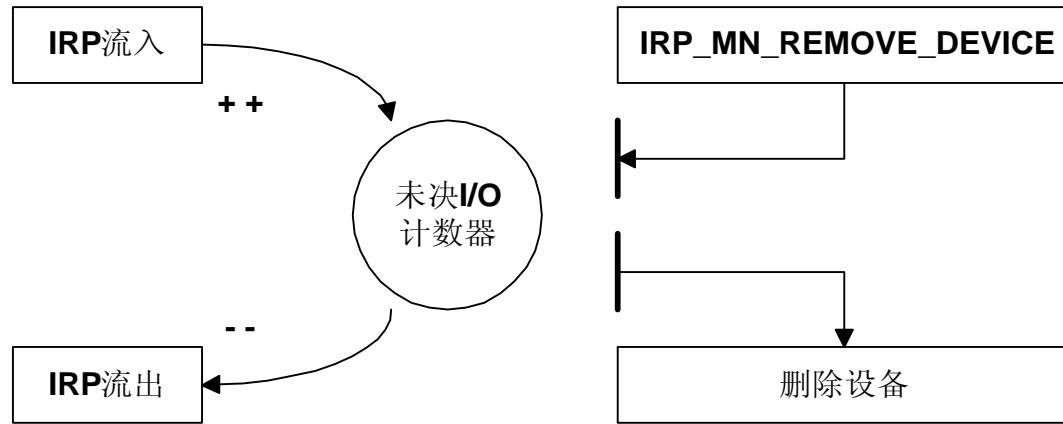


图 6-6. 操作 `IO_REMOVE_LOCK`

为了实现这个处理机制，你应该在设备扩展中定义一个锁变量：

```
struct DEVICE_EXTENSION {
    ...
    IO_REMOVE_LOCK RemoveLock;
    ...
};
```

还应该在 `AddDevice` 中初始化这个锁对象：

```
NTSTATUS AddDevice(PDRIVER_OBJECT DriverObject, PDEVICE_OBJECT pdo)
{
    ...
    IoInitializeRemoveLock(&pdo->RemoveLock, 0, 0, 0);
    ...
}
```

`IoInitializeRemoveLock` 的最后三个参数分别代表标签值、锁的最大生存期、锁计数器的最大值，它们在 free 版本的操作系统中不被使用。第三个和第四个参数为 0 代表你不希望执行对应的错误检测(生存期和嵌套级)，即使在 checked 版本的操作系统中。

无论何时，当你收到一个 I/O 请求时(除了 `IRP_MJ_CREATE`)，你就调用 `IoAcquireRemoveLock`。如果删除设备的操作正在进行，则 `IoAcquireRemoveLock` 返回 `STATUS_DELETE_PENDING`。否则，该函数将获得删除锁并返回 `STATUS_SUCCESS`。一旦你完成一个 I/O 操作，就调用 `IoReleaseRemoveLock`，该函数将释放删除锁以及目前未处理的删除操作。下面是一个纯粹假设的派遣函数，该函数正完成手头的 IRP：

```
NTSTATUS DispatchSomething(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
    if (!NT_SUCCESS(status))
        return CompleteRequest(Irp, status, 0);
    ...
    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    return CompleteRequest(Irp, <some code>, <info value>);
}
```

`IoAcquireRemoveLock` 和 `IoReleaseRemoveLock` 的第二个参数仅是一个标签值，checked 版本的操作系统用它来匹配请求或释放调用。

获取和释放删除锁的调用与 PnP 派遣函数和删除设备子派遣函数中的附加逻辑正好吻合。首先，**DispatchPnp** 必须遵守锁定设备和解锁设备的规则，所以它将包含下面代码，这些代码以前没有在“IRP_MJ_PNP”派遣函数中出现过：

```
NTSTATUS DispatchPnp(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
    if (!NT_SUCCESS(status))
        return CompleteRequest(Irp, status, 0);
    ...
    status = (*fcntab[fcn](fdo, Irp));
    if (fcn != IRP_MN_REMOVE_DEVICE)
        IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    return status;
}
```

换句话说，**DispatchPnp** 先锁定设备，然后调用子派遣函数，最后解锁设备。而关于 **IRP_MN_REMOVE_DEVICE** 的子派遣函数也含有你没见过的专用逻辑：

```
NTSTATUS HandleRemoveDevice(PDEVICE_OBJECT fdo, PIRP Irp)
{
    Irp->IoStatus.Status = STATUS_SUCCESS;
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    AbortRequests(&pdx->dqReadWrite, STATUS_DELETE_PENDING);           <--1
    DeregisterAllInterfaces(pdx);
    StopDevice(fdo, pdx->state == WORKING);
    pdx->state = REMOVED;
    NTSTATUS status = DefaultPnpHandler(pdx->LowerDeviceObject, Irp);      <--2
    IoReleaseRemoveLockAndWait(&pdx->RemoveLock, Irp);                      <--3
    RemoveDevice(fdo);
    return status;
}
```

1. Windows 98 不发送 **SURPRISE_REMOVAL** 请求，所以这个删除 IRP 就是第一个指出设备已经消失的 IRP。调用 **StopDevice** 允许你释放设备占用的所有 I/O 资源。调用 **AbortRequests** 将使你完成任何排队的 IRP，并且从现在开始将不再接受任何新的 IRP。
2. 我们已经做完了自己的工作，现在把这个请求传递到低层驱动程序。
3. PnP 派遣例程占有着删除锁。我们现在调用专用的 **IoReleaseRemoveLockAndWait** 函数来释放锁引用并等待所有对该锁的引用都被释放。一旦 **IoReleaseRemoveLockAndWait** 函数返回，任何后来的 **IoAcquireRemoveLock** 调用都将得到 **STATUS_DELETE_PENDING** 返回状态，指出设备正在删除。

注意

有时当某个 IRP 完成时，**IRP_MN_REMOVE_DEVICE** 处理程序会被阻塞。这在 Windows 98 和 Windows 2000 中确实允许，因为它们的设计允许这种可能性，**IRP_MN_REMOVE_DEVICE** 是在一个系统线程的上下文中发送的，所以允许阻塞。某些 WDM 功能甚至存在于 OEM 版本的 Windows 95 中，但你不能在那里阻塞设备删除请求。因此，如果你的设备需要运行在 Windows 95 中，你需要避免这种阻塞。

这就是防止设备在使用时被删除的锁定和解锁机制。为了运用这个机制，你还需要知道何时调用 **IoAcquireRemoveLock** 和 **IoReleaseRemoveLock** 函数。基本上，可以快速完成请求的 IRP 派遣函数应该获取和释放这种锁。

然而，需要排队 IRP 的派遣例程不应该获取删除锁。对于一个在队列中的 IRP，你应该在 StartIo 函数中获取删除锁，并在 DPC 例程中释放删除锁。因此，我们扩展了 **StartIo** 和 **DpcForIsr** 的框架例程：

```
VOID StartIo(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);           <--1
    if (!NT_SUCCESS(status))
    {
        CompleteRequest(Irp, status, 0);                                         <--2
        return;
    }

    // start request on device
}

VOID DpcForIsr(PKDPC Dpc, PDEVICE_OBJECT device, PIRP junk, PVOID context)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);

    ...
    StartNextPacket(&pdx->dqReadWrite, device);
    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);                            <--3
    CompleteRequest(Irp, ...);
}
```

1. 我们在这里获取了删除锁而不是在派遣例程中。我们不希望以在队列中存放一个 IRP 的方式来防止 PnP 管理器关闭我们的设备。另外，我们最好不让取消例程关心这个删除锁。
2. **IoAcquireRemoveLock** 只有在删除操作未决时才失败。该函数的返回值可以为 **STATUS_SUCCESS** 或 **STATUS_DELETE_PENDING**。在失败的情况下，不要调用 **StartNextPacket** 函数，因为在设备将要消失时我们不能启动一个新操作。如果我们调用 **StartNextPacket** 函数，将递归调用 **StartIo** 函数，**StartIo** 函数将再次获取删除锁并失败，而这又导致调用 **StartNextPacket**，它又会去调用 **StartIo**，最后由于堆栈溢出而导致 BSOD。
3. **IoReleaseRemoveLock** 调用与 **StartIo** 中的 **IoAcquireRemoveLock** 调用相对。

在 **IRP_MJ_CREATE** 派遣例程中没有必要获取删除锁。第六章最后给出的兼容性讨论解释了这个问题，在 Windows 98 中处理 **IRP_MJ_CREATE** 请求时不应该获取删除锁，否则将导致死锁。在 Windows 2000 中你也不需要获取删除锁，尽管这不会带来什么坏处，因为当设备句柄打开时，PnP 管理器不会向你发送 **IRP_MN_REMOVE_DEVICE**。Service Pack-2 包含了修订后的例子驱动程序和一个修订后的驱动程序向导。

如果用户使用设备管理器删除设备，而某应用程序正拥有该设备的打开句柄，那么操作系统将拒绝删除该设备并通知用户。如果设备被用户从计算机上物理地摘除并且没有使用设备管理器，那么一个良好的应用程序应该注意 **WM_DEVICECHANGE** 消息，该消息通知应用程序设备已经被卸载，应用程序接着应该关闭设备句柄，驱动程序应该延迟执行 **IRP_MN_REMOVE_DEVICE** 请求，直到句柄被真正关闭，其实这也是删除锁逻辑允许你做的。

IO_REMOVE_LOCK 的兼容问题

IO_REMOVE_LOCK 对象和与之相关的服务函数并不是 WDM 的一部分。但 **WDM.H** 中却包含它们的声明，并且 **WDM.LIB** 也包含删除锁函数的输入定义。但 Windows 98 并没有实际输出这些函数。因此，引用这些函数的驱动程序将不能在 Windows 98 中运行。这非常不幸，因为每个 WDM 驱动程序都需要互锁的设备删除操作。

DDK 中的例子程序使用两种方式应付这个不兼容问题。有些例子使用自创建的机制来代替 **IO_REMOVE_LOCK**。另一些例子提供了名称类似于 **XxxAcquireRemoveLock** 的服务函数。

我的例子驱动程序使用与第二种方式类似的方法。通过使用#define语句，我用自己声明的IO_REMOVE_LOCK对象和支持函数替换了官方的对象和函数。这样，我的例子代码将调用IoAcquireRemoveLock，等等。在使用GENERIC.SYS的例子中，预处理器中的欺骗代码实际上把这些调用导向GENERIC.SYS中的GenericAcquireRemoveLock等函数。在没有使用GENERIC.SYS的例子中，预处理器中的欺骗代码把这些调用导向REMOVELOCK.CPP文件中的AcquireRemoveLock等函数。

在Windows 2000中你可以调用标准的删除锁函数。为了使例子程序能够运行在Windows 98中，我写了一个实现删除锁函数的小驱动程序，它是一个VxD(虚拟设备驱动程序)，在运行例子驱动程序前你需要先安装这个VxD。(见附录A)但我并不认为这是解释WDM编程的好方法。

DEVQUEUE如何工作

不同于本书中的其它例子，我将在这里给出DEVQUEUE对象的全部实现代码。

初始化DEVQUEUE

在DEVQUEUE.H头文件中，DEVQUEUE对象有如下声明：

```
typedef struct _DEVQUEUE {
    LIST_ENTRY head;
    KSPIN_LOCK lock;
    PDRIVER_START StartIo;
    LONG stallcount;
    PIRP CurrentIrp;
    KEVENT evStop;
    NTSTATUS abortstatus;
} DEVQUEUE, *PDEVQUEUE;
```

InitializeQueue按下面方式初始化这种对象：

```
VOID NTAPI InitializeQueue(PDEVQUEUE pdq, PDRIVER_STARTIO StartIo)
{
    InitializeListHead(&pdq->head);                                <--1
    KeInitializeSpinLock(&pdq->lock);                               <--2
    pdq->StartIo = StartIo;                                         <--3
    pdq->stallcount = 1;                                            <--4
    pdq->CurrentIrp = NULL;                                         <--5
    KeInitializeEvent(&pdq->evStop, NotificationEvent, FALSE);      <--6
    pdq->abortstatus = (NTSTATUS) 0;                                 <--7
}
```

1. 我们使用一个普通(非互锁)的双链表来排队IRP。因为我们用自己的自旋锁保护链表访问，所以不必使用互锁链表。
2. 该自旋锁既保护队列的访问，也保护DEVQUEUE结构中的其它域。它还代替了全局取消自旋锁，既保护了全部的取消处理，也提高了系统性能。
3. 每个队列都有自己的StartIo函数。
4. 停止计数器指出提交到停止的StartIo的IRP的次数。初始化这个计数器为1代表IRP_MN_START_DEVICE处理程序必须调用RestartRequests释放一个IRP。
5. CurrentIrp域记录着最近发往StartIo例程的IRP。初始化这个域为NULL指出设备开始是空闲的。
6. 必要时，我们使用这个事件来阻塞WaitForCurrentIrp。我们将在StartNextPacket函数中设置这个事件，该函数在当前IRP完成时总被调用。

7. 有两种情况我们必须拒绝到来的 IRP。第一种情况是在我们真正提交设备删除请求后，当我们必须启动一个状态为 STATUS_DELETE_PENDING 的新 IRP 时。第二种情况是在低电源期间，参照我们管理的设备类型，我们应该把新 IRP 标上 STATUS_DEVICE_POWERED_OFF 失败代码。**abortstatus** 域记录着我们拒绝的 IRP 所使用的状态代码。

停止队列

停止 IRP 队列包含两个 DEVQUEUE 函数：

```
VOID NTAPI StallRequests(PDEVQUEUE pdq)
{
    InterlockedIncrement(&pdq->stallcount);                                <-1
}

BOOLEAN NTAPI CheckBusyAndStall(PDEVQUEUE pdq)
{
    KIRQL oldirql;
    KeAcquireSpinLock(&pdq->lock, &oldirql);                                <-2
    BOOLEAN busy = pdq->CurrentIrp != NULL;                                    <-3
    if (!busy)                                                               <-4
        InterlockedIncrement(&pdq->stallcount);
    KeReleaseSpinLock(&pdq->lock, oldirql);
    return busy;
}
```

1. 为了停止请求，我们仅需要设置停止计数器为一个非零的值。没有必要使用一个自旋锁来保护这个增 1 操作，因为任何与我们竞争的设备也必须使用互锁的增 1 或减 1 函数来更改这个值。
2. 因为 **CheckBusyAndStall** 函数需要以原子方式操作，所以我们应该先获取队列的自旋锁。
3. **CurrentIrp** 为非 NULL 则代表设备正忙于处理队列中的某个请求。
4. 如果设备当前空闲，则该语句启动停止的队列，这能防止设备后来变为忙。

排队IRP

当某派遣函数调用 **StartPacket** 时，一个 IRP 就被加入到队列中：

```
VOID NTAPI StartPacket(PDEVQUEUE pdq, PDEVICE_OBJECT fdo, PIRP Irp, PDRIVER_CANCEL cancel)           // StartPacket
{
    KIRQL oldirql;                                                        <-1
    KeAcquireSpinLock(&pdq->lock, &oldirql);
    NTSTATUS abortstatus = pdq->abortstatus;                                <-2
    if (abortstatus)
    {
        KeReleaseSpinLock(&pdq->lock, oldirql);                           // aborting all requests now
        Irp->IoStatus.Status = abortstatus;
        IoCompleteRequest(Irp, IO_NO_INCREMENT);
    }                                                               // aborting all requests now
    else if (pdq->CurrentIrp || pdq->stallcount)                         <-3
    {
        IoSetCancelRoutine(Irp, cancel);                                     // queue this irp
        if (Irp->Cancel && IoSetCancelRoutine(Irp, NULL))
        {
            KeReleaseSpinLock(&pdq->lock, oldirql);                         // IRP has already been cancelled
            Irp->IoStatus.Status = STATUS_CANCELLED;
            IoCompleteRequest(Irp, IO_NO_INCREMENT);
        }                                                               // IRP has already been cancelled
    }
}
```

1. 获取自旋锁可以使我们访问 **DEVQUEUE** 中的域不受其它支持例程的干扰，主要是 **StartNextPacket**，该函数试图访问同一个队列。
 2. 如我以前描述的，我们有时需要拒绝到来的 IRP。如果 **abortstatus** 为非零，我们就完成该 IRP。我们的调用者将得到 **STATUS_PENDING**，所以需要由我们自己来完成操作。
 3. 如果设备当前正忙，或者驱动程序的其它例程停止了设备队列，我们需要排队该 IRP 以便以后处理。
 4. 我们可能与 **IoCancelIrp** 的一个实例竞争，该函数试图取消这个 IRP。我们先用 **IoSetCancelRoutine** 例程在该 IRP 中安装我们自己的取消例程，该例程执行一个互锁的数据交换。然后我们测试 **Cancel** 标志。如果 **Cancel** 标志被设置，我们的取消例程可能被调用，也可能没被调用，这取决于我们的代码与 **IoCancelIrp** 执行的确切顺序。如果我们的取消例程被调用，则第二次调用 **IoSetCancelRoutine** 将返回 **NULL**；这样我们就能把该 IRP 加入队列，并利用取消例程立即提取该 IRP 并完成它。如果我们的取消例程没有被调用，我们将在这里完成该 IRP。
 5. 这里我们真正地排队该 IRP。IRP 的 **Tail.Overlay.ListEntry** 域被设计成这样使用。
 6. 最后一种情况是，当队列处于 **READY** 状态并且设备当前不忙。我们设置 **DEVQUEUE** 中的 **CurrentIrp** 指针，释放自旋锁，然后在 **DIRECT_DISPATCH_LEVEL** 级上调用 **StartIo** 例程。

拥有自旋锁的程序可以修改 `CurrentIrp`, 因此我们确信在测试 `CurrentIrp` 时不会得到含糊的结果。另一方面, 停止计数器在 `StallRequests` 中被增 1 却没有自旋锁的保护。很明显, 问题发生的唯一机会就是当计数器被从 0 增到 1 时, 由于我们并不关心计数器有什么非零值, 因此我们的例程可能与此同时执行。假设有一个 `StallRequests` 调用要把计数器的值从 0 增到 1, 如果我们阻止增 1 操作而发现计数器的值为 0, 我们将前进并启动下一个请求。因为 `StallRequests` 的调用者愿意使设备处于忙状态(如果调用者不愿意, 它应使用 `CheckBusyAndStall` 函数), 所以这样做可以。如果我们发现计数器已增到 1, 我们就排队该 IRP, 这也符合 `StallRequests` 调用者的意图。

出队IRP

出队大部分 IRP 的函数是 **StartNextPacket**, 该函数在一个 DPC 例程中被调用:

```
PIRP NTAPI StartNextPacket(PDEVQUEUE pdq, PDEVICE_OBJECT fdo)
{
    KIRQL oldirql;                                         <-1
    KeAcquireSpinLock(&pdq->lock, &oldirql);
    PIRP CurrentIrp = (PIRP) InterlockedExchangePointer (&pdq->CurrentIrp, NULL);
    if (CurrentIrp)                                         <-3
        KeSetEvent(&pdq->evStop, 0, FALSE);
    while (!pdq->stallcount && !pdq->abortstatus && !IsListEmpty(&pdq->head)) <-4
    {
        PLIST_ENTRY next = RemoveHeadList(&pdq->head);
        PIRP Irp = CONTAINING_RECORD(next, IRP, Tail.Overlay.ListEntry);
        if (!IoSetCancelRoutine(Irp, NULL))
        {
            InitializeListHead(&Irp->Tail.Overlay.ListEntry);
        }
    }
}
```

```

        continue;
    }
    pdq->CurrentIrp = Irp;
    KeReleaseSpinLockFromDpcLevel(&pdq->lock);
    (*pdq->StartIo)(fdo, Irp);
    KeLowerIrql(olddirql);
    return CurrentIrp;
}
KeReleaseSpinLock(&pdq->lock, oldirql);
return CurrentIrp;
}

```

1. 我们首先获取该队列的自旋锁，以便我们能不受干扰地访问队列对象的内部结构。
2. 我们将把当前 IRP 的地址作为返回值，并且还设置 **CurrentIrp** 指针为 NULL。由于使用了自旋锁，我们不必使用原子操作来提取并置空 CurrentIrp 的值。
3. 某些例程可能在 **WaitForCurrentIrp** 中等待当前请求的完成。调用 **KeSetEvent** 将满足那些等待。
4. 这一系列检测决定我们是否能出队一个请求。队列不是停止的，也不处于 REJECTING 状态，队列至少应包含一个请求，这样我们调用 **RemoveHeadList** 才有意义。
5. 这行代码删除队列中最旧的一个 IRP。
6. 置空 IRP 中的取消例程指针将阻止 **IoCancelIrp** 企图取消该 IRP。如果试图取消该 IRP 的 **IoCancelIrp** 函数此时正在另一个 CPU 上运行，我们应该从 **IoSetCancelRoutine** 函数得到 NULL 返回值。当 **CancelRequest** 又获得控制时，它将需要获取队列的自旋锁以便继续进一步处理。在这一点上，它将盲目地从当前队列中删除该 IRP。在该 IRP 自己的连接域上调用 **InitializeListHead** 将使 **CancelRequest** 在做进一步处理时更安全。
7. 在这里我们把刚出队的 IRP 送到 **StartIo** 进行处理。

RestartRequests 函数平衡 **StallRequests** 调用或 **CheckBusyAndStall** 调用。它把队列的第一个 IRP 送到 **StartIo** 例程，幸好，我们可以利用 **StartNextPacket**:

```

VOID NTAPI RestartRequests(PDEVQUEUE pdq, PDEVICE_OBJECT fdo)
{
    if (InterlockedDecrement(&pdq->stallcount) > 0)
        return;
    StartNextPacket(pdq, fdo);
}

```

取消IRP

StartPacket 寄存了由其调用者提供的取消例程，该例程然后又把工作委托给队列的 **CancelRequest** 函数:

```

VOID NTAPI CancelRequest(PDEVQUEUE pdq, PIRP Irp)
{
    KIRQL oldirql = Irp->CancelIrql;
    IoReleaseCancelSpinLock(DISPATCH_LEVEL);
    KeAcquireSpinLockAtDpcLevel(&pdq->lock);
    RemoveEntryList(&Irp->Tail.Overlay.ListEntry);
    KeReleaseSpinLock(&pdq->lock, oldirql);
    Irp->IoStatus.Status = STATUS_CANCELLED;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
}

```

我们在拥有全局取消自旋锁的情况下被调用，而我们立刻又释放了这个自旋锁。这之后，所有的保护工作由队列的自旋锁来代替。当 **IoCancelIrp** 调用 **IoAcquireCancelSpinLock** 时，它在 IRP 的 **CancelIrql** 域保存了以前的中断请求级，我们最后需要回复到那个 IRQL，因此，我们把这个值保存到 **oldirql** 变量中。

注意

IoCancelIrp 的调用者有责任确保该 IRP 没有被完成。

IRP_MJ_CLEANUP 也能取消 IRP，该请求在我们收到 IRP_MJ_CLOSE 请求前收到。DEVQUEUE 的 **CleanupRequests** 函数几乎与标准模型中的 **DispatchCleanup** 例程完全相同。两者仅有的不同之处是 CleanupRequests 函数需要获取队列的自旋锁：

```
VOID NTAPI CleanupRequests(PDEVQUEUE pdq, PFILE_OBJECT fop, NTSTATUS status)
{
    LIST_ENTRY cancellist;                                <--1
    InitializeListhead(&cancellist);
    KIRQL oldirql;
    KeAcquireSpinLock(&pdq->lock, &oldirql);
    PLIST_ENTRY first = &pdq->head;
    PLIST_ENTRY next;
    for (next = first->Flink; next != first; )           <--2
    {
        PIRP Irp = CONTAINING_RECORD(next, IRP, Tail.Overlay.ListEntry);
        PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);      <--3
        PLIST_ENTRY current = next;
        next = next->Flink;                                              <--4
        if (fop && stack->FileObject != fop)
            continue;
        if (!IoSetCancelRoutine(Irp, NULL))                                <--5
            continue;
        RemoveEntryList(current);                                         <--6
        InsertTailList(&cancellist, next);
    }
    KeReleaseSpinLock(&pdq->lock, oldirql);                      <--7
    while (!IsListEmpty(&cancellist))
    {
        next = RemoveHeadList(&cancellist);
        PIRP Irp = CONTAINING_RECORD(next, IRP, Tail.Overlay.ListEntry);
        Irp->IoStatus.Status = status;
        IoCompleteRequest(Irp, IO_NO_INCREMENT);
    }
}
```

1. 我们的策略是，在队列自旋锁的保护下把需要删除的 IRP 移到我们私有的队列中。因此，第一步就是初始化这个私有队列并获取自旋锁。
2. 该循环遍历整个队列，直到返回队列头。注意，**for** 语句的第三个子句没有循环自增步骤。
3. 如果我们为处理 IRP_MJ_CLEANUP 请求而被调用，则 **fop** 参数就是将要被关闭的文件对象的地址。我们要排除同属于这个文件对象的 IRP，因此我们需要先找到该 IRP 的堆栈单元。
4. 如果我们决定从队列中删除这个 IRP，我们就不能方便地找到主队列中的下一个 IRP，因此我们在这里使用这样的语句，这里也执行了循环的自增步骤。
5. 这个尤其明智的语句是由 Jamie Hanrahan 添加的。我们需要知道是否有人试图取消我们当前正在寻找的 IRP。它们仅能到达 **CancelRequest** 试图获取自旋锁的地方。然而在到达那里之前，它们必须执行 **IoCancelIrp** 中置空取消例程指针的语句。如果我们用 **IoSetCancelRoutine** 函数发现那个指针为空，我们就能确信有人真的要取消这个 IRP。因此我们就在这个循环中简单地跳过该 IRP，稍后我们再允许取消例程完成该 IRP。
6. 在这里我们把 IRP 从主队列中取出并放入我们自己的队列。
7. 一旦完成 IRP 的移动，我们就可以释放自旋锁。然后我们取消找到的所有 IRP。

CleanupRequests 可以在驱动程序的任何地方被调用。例如，从我以前提到的 **IRP_MN_REMOVE_DEVICE** 处理程序中调用，它给出一个 **NULL** 文件对象指针(指出清除所有 **IRP**)和一个为 **STATUS_DELETE_PENDING** 的状态代码。

等待当前的IRP

IRP_MN_STOP_DEVICE 的处理程序需要等待当前 **IRP** 的完成，调用 **WaitForCurrentIrp**:

```
VOID NTAPI WaitForCurrentIrp(PDEVQUEUE pdq)
{
    KeClearEvent(&pdq->evStop);                                     <-1
    ASSERT(pdq->stallcount != 0);                                     <-2
    KIRQL oldirql;                                                 <-3
    KeAcquireSpinLock(&pdq->lock, &oldirql);
    BOOLEAN mustwait = pdq->CurrentIrp != NULL;
    KeReleaseSpinLock(&pdq->lock, oldirql);
    if (mustwait)
        KeWaitForSingleObject(&pdq->evStop, Executive, KernelMode, FALSE, NULL);
}
```

1. 每次调用 **StartNextPacket** 函数，它都使 **evStop** 事件进入信号态。为了不使我们的等待被错误地满足，我们先清除该事件。
2. 不先停止队列就调用该函数是无意义的。此外，**StartNextPacket** 将启动下一个 **IRP**，因此，设备将再次进入忙状态。
3. 如果设备当前忙，我们将在 **evStop** 事件上等待，直到某段代码调用了 **StartNextPacket** 使该事件进入信号态。我们需要用自旋锁保护对 **CurrentIrp** 的探测操作，通常，测试一个指针是否为 **NULL** 并不是一个原子事件。如果指针现在为 **NULL**，以后它就不能被改变，因为我们假定队列是停止的。

放弃请求

设备的突然删除要求我们立即停止所有试图触及硬件的未决 **IRP**。另外，我们还要能拒绝所有后来的 **IRP**。**AbortRequests** 函数可以完成这个任务:

```
VOID NTAPI AbortRequests(PDEVQUEUE pdq, NTSTATUS status)
{
    pdq->abortstatus = status;
    CleanupRequests(pdq, NULL, status);
}
```

abortstatus 的设置将使队列进入 **REJECTING** 状态，这样所有后来的 **IRP** 都将被拒绝，其返回的状态值由我们的调用者提供。在这里，以 **NULL** 文件对象指针调用 **CleanupRequests** 函数将清空整个队列。

我们不应该碰任何当前在硬件上活动的 **IRP**。不使用 **HAL** 访问硬件的驱动程序，例如 **USB** 驱动程序，将依靠 **hub** 和主控制器驱动程序来废弃当前 **IRP**。而使用 **HAL** 的驱动程序可能会挂起系统，或至少使 **IRP** 进入不稳定状态，因为不存在的硬件不会生成中断使 **IRP** 完成。为了处理这种情况，你应该调用

AreRequestsBeingAborted:

```
NTSTATUS AreRequestsBeingAborted(PDEVQUEUE pdq)
{
    return pdq->abortstatus;
}
```

在这个例程中使用队列自旋锁是愚蠢的。假定我们要以线程安全和多处理器安全的方式捕获 **abortstatus** 的瞬间值，那么一旦我们释放该自旋锁，这个值就将变成无用值。

注意

如果你的设备可以以这种方式删除，即简单地挂起未决的请求，你应该有一个看门狗定时器(Watchdog Timer)在运行，它可以在一段时间后杀死这些 IRP。看门狗定时器见第九章。

有时我们需要恢复上一个 AbortRequest 调用的作用。**AllowRequests** 可以做到这一点：

```
VOID NTAPI AllowRequests(PDEVQUEUE pdq)
{
    pdq->abortstatus = (NTSTATUS) 0;
}
```

其它配置功能

到现在，我已经讲完了关于写一个硬件设备驱动程序所需的全部重要概念。为了结束本章，我将讨论两个不太重要的副功能码 **IRP_MN_FILTER_RESOURCE_REQUIREMENTS** 和 **IRP_MN_DEVICE_USAGE_NOTIFICATION**，在实际的驱动程序中你有时需要处理这两种请求。然后，我将讲述如何写一个小小的总线驱动程序以支持非标准控制器或多功能设备。最后，我还将在关于如何寄存 PnP 事件的接收通知，这些事件可以影响你自己以外的其它设备。

注意

其它 PnP 请求我没有讨论，因为 DDK 参考手册已经给出详细的解释，没有必要重述。例如，向其它驱动程序输出一个直接调用接口可能有潜在的用途，但在一般情况下没有必要这样做。因此，我不将为 **IRP_MN_QUERY_INTERFACE** 提供例子和解释。我将在第八章“电源管理”中再讨论 **IRP_MN_QUERY_CAPABILITIES**，因为那里比较适合。

过滤资源需求

有时 PnP 管理器会错误地报告驱动程序的资源需求。这可能由于硬件或固件上存在着 bug，也可能因为某个遗留设备的 INF 文件中出现错误，或者是其它原因。系统以 **IRP_MN_FILTER_RESOURCE_REQUIREMENTS** 请求的形式提供了一个切入点，它在 PnP 管理器将要分配资源前 给你提供一个机会来检测或修改资源列表。

当你收到一个过滤请求时，其堆栈单元的 **Parameters** 联合的 **FilterResourceRequirements** 子结构指向一个 **IO_RESOURCE_REQUIREMENTS_LIST** 数据结构，该结构列出了设备的资源需求。另外，如果你上层面上的任何驱动程序处理了该 IRP 并修改了其中的资源需求，则该 IRP 的 **IoStatus.Information** 域将指向第二个 **IO_RESOURCE_REQUIREMENTS_LIST**，你应该使用这个数据结构。策略是这样的：如果你想向当前的资源列表中加入资源，应该在派遣函数中做。你先把该 IRP 同步地传向堆栈下层，即，使用 **ForwardAndWait** 方法启动一个设备请求。当你最后有获得控制时，就可以修改列表中的任何资源描述。

下面是一个演示过滤处理机制的简单函数：

```
NTSTATUS HandleFilterResources(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    PIO_RESOURCE_REQUIREMENTS_LIST original =
        stack->Parameters.FilterResourceRequirements.IoResourceRequirementList;
    PIO_RESOURCE_REQUIREMENTS_LIST filtered =
        (PIO_RESOURCE_REQUIREMENTS_LIST) Irp->IoStatus.Information;
    PIO_RESOURCE_REQUIREMENTS_LIST source = filtered ? filtered : original;
    if (source->AlternativeLists != 1)
        return DefaultPnpHandler(fdo, Irp);
    ULONG sizelist = source->ListSize;
    PIO_RESOURCE_REQUIREMENTS_LIST newlist = (PIO_RESOURCE_REQUIREMENTS_LIST)
        ExAllocatePool(PagedPool, sizelist + sizeof(IO_RESOURCE_DESCRIPTOR));
    if (!newlist)
        return DefaultPnpHandler(fdo, Irp);
    RtlCopyMemory(newlist, source, sizelist);
    newlist->ListSize += sizeof(IO_RESOURCE_DESCRIPTOR);
    PIO_RESOURCE_DESCRIPTOR resource = &newlist->List[0].Descriptors[newlist->List[0].Count++];
    RtlZeroMemory(resource, sizeof(IO_RESOURCE_DESCRIPTOR));
    resource->Type = CmResourceTypeDevicePrivate;
    resource->ShareDisposition = CmResourceShareDeviceExclusive;
```

```

resource->u.DevicePrivate.Data[0] = 42;
Irp->IoStatus.Information = (ULONG_PTR) newlist;
if (filtered && filtered != original)
    ExFreePool(filtered);
NTSTATUS status = ForwardAndWait(fdo, Irp);
if (NT_SUCCESS(status))
{
    // stuff
}
Irp->IoStatus.Status = status;
IoCompleteRequest(Irp, IO_NO_INCREMENT);
return status;
}

```

1. 该请求的参数包含一个 I/O 资源需求列表。这些信息得自设备的配置空间、注册表、或总线驱动程序能发现的任何地方。
2. 高级驱动程序也许已经过滤了这些资源，并加入了另一个列表。如果是这样，它们就设置 **IoStatus.Information** 域，使其指向扩展后的请求列表。
3. 如果这里没有过滤后的列表，我们就扩展原始的列表。如果有过滤后的列表，我们就扩展过滤后的列表。
4. 理论上，将存在几个可选的需求列表，但处理这些列表超过了这个简单例子的能力。
5. 在向下传递请求前，我们需要加入资源。首先我们分配一个新的需求列表并把旧需求复制到新表。
6. 我们小心地保留已存在的资源描述顺序，然后加入自己的资源描述。在这个例子中，我们加入一个我们驱动程序私有的资源。
7. 我们在 IRP 的 **IoStatus.Information** 域中保存扩展后的请求列表的地址，该域将被低级驱动程序和 PnP 系统查看。如果我们扩展的是已过滤的列表，我们需要释放被旧表占用的内存。
8. 我们使用 **ForwardAndWait** 辅助函数下传该请求。在该 IRP 向上返回时我们不修改其任何资源描述，所以我们在此应调用 **DefaultPnpHandler**，并同时传递下层的返回状态。
9. 当完成该 IRP 时，不论我们指定成功或失败，都必须注意不要修改 I/O 状态块中的 **Information** 域：该域可能存放一个指向某驱动程序资源需求列表的指针，而这个列表也可能是我们添加的。PnP 管理器将释放额外表结构所占用的内存。

设备用途通知

磁盘驱动程序(以及磁盘控制器驱动程序)有时需要了解外来的关于它们如何被操作系统使用的信息，**IRP_MN_DEVICE_USAGE_NOTIFICATION** 请求提供了获取这些信息的手段。在 IRP 堆栈单元的 **Parameters.UsageNotification** 子结构中包含了这样两个参数，见表 6-4。**InPath** 值(Boolean)指出设备是否处于支持用途信息的设备路径上，**Type** 值指出几种可能的特殊用法。

表 6-4. I/O 堆栈单元中 *Parameters.UsageNotification* 子结构中的域

参数	描述
InPath	如果设备处于 Type 指定的路径中，则为 TRUE，否则为 FALSE
Type	用法类型

在通知请求的子派遣例程中，你应该用一个 **switch** 语句来区分各种通知。在大多数情况下，你将把该 IRP 下传。下面是这个子派遣例程的框架代码：

```

NTSTATUS HandleUsageNotification(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    DEVICE_USAGE_NOTIFICATION_TYPE type = stack->Parameters.UsageNotification.Type;
    BOOLEAN inpath = stack->Parameters.UsageNotification.InPath;

```

```

switch (type)
{
    case DeviceUsageTypeHibernation:
        ...
        Irp->IoStatus.Status = STATUS_SUCCESS;
        break;
    case DeviceUsageTypeDumpFile:
        ...
        Irp->IoStatus.Status = STATUS_SUCCESS;
        break;
    case DeviceUsageTypePaging:
        ...
        Irp->IoStatus.Status = STATUS_SUCCESS;
        break;
    default:
        break;
}
return DefaultPnpHandler(fdo, Irp);
}

```

仅当你明确地认为该通知是送往总线驱动程序的信号时才设置其 **Status** 域为 **STATUS_SUCCESS**。如果你没有设置 **STATUS_SUCCESS**，则总线驱动程序将假设你并不知道，因此也不处理该通知。

你应该知道你的设备不能支持某种用途。例如，假设你的磁盘设备不能用来存储休眠文件，但如果该 IRP 指定 **InPath** 值，你应该使该 IRP 失败：

```

...
case DeviceUsageTypeHibernation:
    if (inpath)
        return CompleteRequest(Irp, STATUS_UNSUCCESSFUL, 0);

```

下面我将简要地描述一下每种使用类型。

DeviceUsageTypePaging

如果该通知的 **InPath** 为 **TRUE** 则指出将有一个内存交换文件在这个设备上打开。如果为 **FALSE** 则指出有一个内存交换文件已被关闭。通常，你应该为被通知的内存交换文件维护一个计数器。当交换文件仍在活动时，你应该使 **STOP** 或 **REMOVE** 的查询失败。另外，当你收到第一个分页通知时，应该确保你的 **READ**、**WRITE**、**DEVICE_CONTROL**、**PNP**、**POWER** 派遣例程锁定在内存中。你还应该清除设备对象中的 **DO_POWER_PAGABLE** 标志以强制电源管理器在 **DISPATCH_LEVEL** 级上向你发送电源管理 IRP。为了安全，我还建议你置空任何已经寄存的空闲通知。(检测空闲状态的讨论见第八章)

注意

在第八章“电源管理”中我将讨论如何在设备对象中设置 **DO_POWER_PAGABLE** 标志。当你拥有某个设备对象时，如果该设备对象设置了这个标志，你应该确保决不清除这个标志。你仅应该在完成例程中，在低级驱动程序已清除其自己的标志后，再清除这个标志。总之，你需要一个完成例程，因为如果 IRP 在低级驱动程序中失败，你需要恢复在派遣例程中所做的任何操作。

DeviceUsageTypeDumpFile

如果该通知的 **InPath** 为 **TRUE** 则指出设备已被选定用于保存系统崩溃时的 dump 文件。如果为 **FALSE** 则取消这个选定。应该用一个计数器来保存 **TRUE** 通知个数减去 **FALSE** 通知个数的差值。当计数器不为 0 时：

- 应确保你的电源管理代码决不让设备超过 D0，或进入全部打开状态。

- 应避免寄存设备的空闲检测，并置空任何未处理的寄存操作。
- 应确保你的驱动程序在处理停止和删除查询请求时返回失败状态。

DeviceUsageTypeHibernation

如果该通知的 `InPath` 为 `TRUE` 则指出设备被选定用于保存休眠文件。如果为 `FALSE` 则取消这个选定。应该用一个计数器来保存 `TRUE` 通知个数减去 `FALSE` 通知个数的差值。你对指定了 `PowerSystemHibernate` 的系统电源管理 IRP 的响应应与通常不同，因为你的设备将用于瞬间记录休眠文件。关于磁盘驱动程序的这方面细节已经超出本书的范围。

控制器和多功能设备

有两类设备不能整洁地匹配到 PnP 框架中。它们是控制器设备：它管理一组子设备；或多功能设备：它在一个卡上提供多种功能。对这两种设备的管理需要加入对使用独立 I/O 资源的多设备对象的创建过程。

在 Windows 2000 中支持 PCI、PCMCIA、USB 设备十分容易，因为它们遵从各自的多功能设备标准。PCI 总线自动识别 PCI 多功能卡。对于 PCMCIA 多功能设备，你可以按照 DDK 中的详细指导把 `MF.SYS` 做为你的多功能卡的功能驱动程序；而 `MF.SYS` 然后将枚举你卡上的功能设备，并使 PnP 管理器装入单独的功能驱动程序。通常 `USB hub` 驱动程序也为单配置设备上的每个接口装入一个独立的功能驱动程序。

除了 USB 设备，原始版本的 Windows 98 缺乏象 Windows 2000 那样的对多功能设备的支持。在 Windows 98 中，为了处理控制器设备或多功能设备，或者处理非标准设备，你需要凭借更英勇的手段。你需要为你的主设备提供一个功能驱动程序，还要分别为每个连接到主设备上的子设备提供一个功能驱动程序。主设备的功能驱动程序就象一个小型的总线驱动程序，它枚举每个子设备并为 PnP 请求和电源管理请求提供默认处理。而写一个完整的总线驱动程序将是一个巨大的工作，我并不打算在这里描述其过程。但我要描述一下枚举子设备的基本机制。你可以利用这些信息写控制器设备或多功能设备的驱动程序，不过这样的驱动程序并不符合 Microsoft 提供的标准模型。

整体结构

在第二章中，图 2-2 显示了当一个父设备，如总线驱动程序，含有子设备时的拓扑结构。控制器和多功能设备使用与此类似的拓扑结构。父设备插入到某个标准总线，该标准总线的驱动程序检测到这个父设备，然后 PnP 管理器就象配置任何普通设备一样配置这个父设备，当它启动该父设备后，它又向这个设备发送了一个带有 `IRP_MN_QUERY_DEVICE_RELATIONS` 副功能码的 PnP 请求，以了解该父设备的总线关系。实际上，这个查询发生于所有设备，因为 PnP 管理器并不知道设备是否含有子设备。

为了响应总线关系请求，父设备的功能驱动程序定位或创建附加的设备对象，每个这样的对象都将成为子设备的 PDO。然后，PnP 管理器继续装入子设备对应的功能驱动程序和过滤器驱动程序，因此最后你就看到了象图 2-2 这样的拓扑结构。

父设备的驱动程序不得不扮演两个角色。它是控制器或多功能设备的 FDO，它还是子设备的 PDO。在 FDO 角色中，它以功能驱动程序的方式处理 PnP 和电源管理请求。在 PDO 角色中，它的行为就象最后处理 PnP 和电源管理请求的驱动程序。

创建子设备对象

在这条路的某个地方，我们可能要处理 `IRP_MN_START_DEVICE` 请求，而父设备驱动程序，以其 FDO 角色，需要为其子设备创建 PDO，并需要记录下这些 PDO 以备以后使用。在这个早期阶段只有一个复杂地方：FDO 和所有 PDO 都属于同一个驱动程序对象，这意味着任何指向这些设备对象的 IRP 都将进入唯一一组派遣例程。驱动程序需要区别处理 FDO 和 PDO 的 PnP 与电源管理请求。因此，你需要为派遣例程提供一个方便的方式来区别 FDO 和某个子设备的 PDO。我处理这个问题的方法是定义了两个有公共开头的设备扩展结构，如下：

```
// The FDO extension:
```

```

typedef struct _DEVICE_EXTENSION {
    ULONG flags;
    ...
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

// The PDO extension:

typedef struct _PDO_EXTENSION {
    ULONG flags;
    ...
} PDO_EXTENSION, *PPDO_EXTENSION;

// The common part:

typedef struct _COMMON_EXTENSION {
    ULONG flags;
} COMMON_EXTENSION, *PCOMMON_EXTENSION;

#define ISPDO 0x00000001

```

IRP_MJ_PNP 的派遣例程如下：

```

NTSTATUS DispatchPnp(PDEVICE_OBJECT DeviceObject, PIRP Irp)
{
    PCOMMON_EXTENSION pcx = (PCOMMON_EXTENSION)DeviceObject->DeviceExtension;
    if (pcx->flags & ISPDO)
        return DispatchPnpPdo(DeviceObject, Irp);
    else
        return DispatchPnpFdo(DeviceObject, Irp);
}

```

在随书光盘上有一个例子驱动程序，MULFUNC，它是一个十分简陋的多功能设备：它仅有两个子设备，我称它们为 A 和 B。MULFUNC 在响应 IRP_MN_START_DEVICE 请求时创建了 A 和 B 的 PDO，下面是它执行的代码(真正的代码含有更多的错误检测)：

```

NTSTATUS StartDevice(PDEVICE_OBJECT fdo, ...)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION)fdo->DeviceExtension;
    CreateChild(pdx, CHILDTYPEA, &pdx->ChildA); <
    CreateChild(pdx, CHILDTYPEB, &pdx->ChildB);
    return STATUS_SUCCESS;
}

NTSTATUS CreateChild(PDEVICE_EXTENSION pdx, ULONG flags, PDEVICE_OBJECT* ppdo)
{
    PDEVICE_OBJECT child;
    IoCreateDevice(pdx->DriverObject,
                  sizeof(PDO_EXTENSION),
                  NULL,
                  FILE_DEVICE_UNKNOWN,
                  FILE_AUTOGENERATED_DEVICE_NAME,
                  FALSE,
                  &child);
    PPDO_EXTENSION px = (PPDO_EXTENSION)child->DeviceExtension;
    px->flags = ISPDO | flags;
}

```

```

px->DeviceObject = child;
px->Fdo = pdx->DeviceObject;
child->Flags &= ~DO_DEVICE_INITIALIZING;
*ppdo = child;
return STATUS_SUCCESS;
}

```

1. CHILDTYPEA 和 CHILDTYPEB 是 **flags** 成员的附加标志位。如果你正写一个真正的总线驱动程序，你不应在这里创建子设备的 PDO，你应该为响应 IRP_MN_QUERY_DEVICE_RELATIONS 请求而枚举你的实际硬件，然后创建 PDO。
2. 我们在这里创建一个命名的设备对象，但我们要求系统自动生成名字，我们已在 **DeviceCharacteristics** 的参数中指定了 FILE_AUTOGENERATED_DEVICE_NAME 标志。

创建过程的最后结果是两个指向设备对象(ChildA 和 ChildB)的指针，这两个指针存在于父设备 FDO 的设备扩展中。

设备向PnP管理器告知自己含有子设备

PnP 管理器通过向每个设备发送类型码为 **BusRelations** 的 IRP_MN_QUERY_DEVICE_RELATIONS 请求来了解其是否含有子设备。以 FDO 角色，父设备使用如下代码响应这个请求：

```

NTSTATUS HandleQueryRelations(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PDEVICE_EXTENSION pdx = ...;
    PIO_STACK_LOCATION stack = ...;
    if (stack->Parameters.QueryDeviceRelations.Type != BusRelations)           <--1
        return DefaultPnpHandler(fdo, Irp);
    PDEVICE_RELATIONS newrel = (PDEVICE_RELATIONS)
        ExAllocatePool(PagedPool, sizeof(DEVICE_RELATIONS) + sizeof(PDEVICE_OBJECT));
    newrel->Count = 2;
    newrel->Objects[0] = pdx->ChildA;
    newrel->Objects[1] = pdx->ChildB;
    ObReferenceObject(pdx->ChildA);                                         <--3
    ObReferenceObject(pdx->ChildB);
    Irp->IoStatus.Information = (ULONG_PTR) newrel;                         <--4
    Irp->IoStatus.Status = STATUS_SUCCESS;
    return DefaultPnpHandler(fdo, Irp);
}

```

1. 除了我们关心的总线关系之外，该 IRP 还能涉及多种类型的关系。我们简单地把该查询请求送到总线驱动程序去处理。
2. 这里，我们分配了能包含两个设备对象指针的结构。DEVICE_RELATIONS 结构的最后是一个一维数组，所以我们仅需加上一个附加指针的大小。
3. 我们调用 **ObReferenceObject** 来增加与每个设备对象关联的参考计数，这些参考计数存在于 DEVICE_RELATIONS 数组中。PnP 管理器会在适当的时间解除对象的参考。
4. 我们需要把该请求下传到真正的总线驱动程序，该总线驱动程序或某个低级过滤器了解我们不知道的额外事实。该 IRP 使用一个不寻常的协议来执行下传操作和完成操作。如果你真正处理了该 IRP，你应该象这样设置 **IoStatus**；否则，不要碰它。注意这里的 **Information** 域用于保存 DEVICE_RELATIONS 结构的指针。在其它情况下，Information 域总是保存一个数值。

在前面的代码片段中，我省略了一个因素。上层过滤器也许已经在 IRP 的 **IoStatus.Information** 域中安装了一个设备对象列表。我们必须扩展这个表，加入我们自己的设备对象指针。

在设备启动时 PnP 管理器自动发送一个总线关系查询请求。你可以通过调用下面服务函数强制发出该请求：

```
IoInvalidateDeviceRelations(pdx->Pdo, BusRelations);
```

你应该仅在检测到你某个子设备到来或离开时才做这个调用。

以 PDO 角色处理 PnP 请求

戴上 PDO 驱动程序的帽子，则父设备驱动程序必须以与功能驱动程序十分不同的方式处理 PnP 请求。表 6-5 列出这些请求。

表 6-5. PDO 驱动程序要处理的 PnP 请求

PnP 请求	处理方式
IRP_MN_START_DEVICE	成功
IRP_MN_QUERY_REMOVE_DEVICE	成功
IRP_MN_REMOVE_DEVICE	成功
IRP_MN_CANCEL_REMOVE_DEVICE	成功
IRP_MN_STOP_DEVICE	成功
IRP_MN_QUERY_STOP_DEVICE	成功
IRP_MN_CANCEL_STOP_DEVICE	成功
IRP_MN_QUERY_DEVICE_RELATIONS	专门处理
IRP_MN_QUERY_INTERFACE	忽略
IRP_MN_QUERY_CAPABILITIES	委派
IRP_MN_QUERY_RESOURCES	成功
IRP_MN_QUERY_RESOURCE_REQUIREMENTS	成功
IRP_MN_QUERY_DEVICE_TEXT	成功
IRP_MN_FILTER_RESOURCE_REQUIREMENTS	成功
IRP_MN_READ_CONFIG	委派
IRP_MN_WRITE_CONFIG	委派
IRP_MN_EJECT	委派
IRP_MN_SET_LOCK	委派
IRP_MN_QUERY_ID	专门处理
IRP_MN_QUERY_PNP_DEVICE_STATE	委派
IRP_MN_QUERY_BUS_INFORMATION	委派
IRP_MN_DEVICE_USAGE_NOTIFICATION	委派
IRP_MN_SURPRISE_REMOVAL	成功
Any other	忽略

通常，父设备驱动程序仅简单地使大多数 PnP 请求成功，并不做特别的处理：

```
NTSTATUS SucceedRequest(PDEVICE_OBJECT pdo, PIRP Irp)
{
    Irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}
```

在上面这个短例程中有一点值得注意，即它不需要修改 IRP 的 IoStatus.Information 域。PnP 管理器总在发送 IRP 前以某种方式初始化该域，在某些情况下，该域可以被过滤器驱动程序或功能驱动程序修改为指向某个数据结构。PDO 驱动程序不应该修改这个域。

父设备驱动程序可以忽略某些 IRP。忽略一个 IRP 会使该 IRP 失败并返回错误代码类似，除了驱动程序不改变 IRP 的 status 域。

```
NTSTATUS IgnoreRequest(PDEVICE_OBJECT pdo, PIRP Irp)
{
    NTSTATUS status = Irp->IoStatus.Status;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}
```

一个缩小的总线驱动程序，就象我们正讨论的这个，它可以把某些 PnP 请求委派给父设备 FDO 下面的真正总线驱动程序去完成。在这种情况中的委派并不是仅仅调用 IoCallDriver 那样简单，因为此时我们是以 PDO 驱动程序的角色接收 IRP，但 I/O 堆栈通常都被用光了，因此我们必须创建一个转发(repeater)IRP，然后再把它发送到我们以 FDO 驱动程序角色出现的驱动程序堆栈中：

```
NTSTATUS RepeatRequest(PDEVICE_OBJECT pdo, PIRP Irp)
{
    PPDO_EXTENSION pdx = (PPDO_EXTENSION) pdo->DeviceExtension;
    PDEVICE_OBJECT fdo = pdx->Fdo;
    PDEVICE_EXTENSION pfx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);

    PDEVICE_OBJECT tdo = IoGetAttachedDeviceReference(fdo); <
    PIRP subirp = IoAllocateIrp(tdo->StackSize + 1, FALSE); <

    PIO_STACK_LOCATION substack = IoGetNextIrpStackLocation(subirp); <
    substack->DeviceObject = tdo;
    substack->Parameters.Others.Argument1 = (PVOID) Irp; <

    IoSetNextIrpStackLocation(subirp); <
    substack = IoGetNextIrpStackLocation(subirp); <
    RtlCopyMemory(substack, stack, FIELD_OFFSET(IO_STACK_LOCATION, CompletionRoutine)); <
    substack->Control = 0; <
    BOOLEAN needsvote = <I'll explain later>; <-4
    IoSetCompletionRoutine(subirp, OnRepeaterComplete, (PVOID) needsvote, TRUE, TRUE, TRUE); <
    subirp->IoStatus.Status = STATUS_NOT_SUPPORTED; <
    IoMarkIrpPending(Irp); <
    IoCallDriver(tdo, subirp); <
    return STATUS_PENDING
}

NTSTATUS OnRepeaterComplete(PDEVICE_OBJECT tdo, PIRP subirp, PVOID needsvote)
{
    ObDereferenceObject(tdo); <
    PIO_STACK_LOCATION substack = IoGetCurrentIrpStackLocation(subirp); <
    PIRP Irp = (PIRP) substack->Parameters.Others.Argument1; <-7 <
    if (subirp->IoStatus.Status == STATUS_NOT_SUPPORTED) <
    {
        if (needsvote)
            Irp->IoStatus.Status = STATUS_UNSUCCESSFUL;
    }
    else
```

```

    Irp->IoStatus = subirp->IoStatus;
    IoFreeIrp(subirp);
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_MORE_PROCESSING_REQUIRED;
}

```

1. 我们将把这个转发 IRP 发送到我们 FDO 所属堆栈的最顶层过滤器驱动程序。该服务例程返回顶级设备对象地址，同时它又增加了该对象参考计数。
2. 分配 IRP 时，我们创建了一个额外的堆栈单元，在这个堆栈单元中，你可以为将要安装的完成例程记录某些上下文信息。在这个额外堆栈单元中放置的 **DeviceObject** 指针将成为完成例程的第一个参数。
3. 这里，我们首先初始化一个实际的堆栈单元，它就是 FDO 堆栈最顶层驱动程序将要接收的堆栈单元。然后安装我们的完成例程。这里我们不能使用标准的 **IoCopyCurrentIrpStackLocationToNext** 宏来复制堆栈单元，因为我们正处理两个不同的 I/O 堆栈。
4. 应该事先计划好如何处理父设备堆栈不真正处理该转发 IRP 的情况。我们后面的处理需要取决于这个被复制 IRP 的副功能码。我们所做的就是计算一个布尔值 **needsvote**，并把该值作为上下文参数传递给我们的完成例程。
5. 你应该把新 PnP 请求的 **status** 域初始化为特殊值 **STATUS_NOT_SUPPORTED**，否则驱动程序校验器会产生 **bug check**。
6. 该语句释放我们对 FDO 堆栈中顶级设备对象的引用。
7. 我们在这里保存原始 IRP 的地址。
8. 这些语句为原始 IRP 设置了完成状态。
9. 我们分配了转发 IRP，因此有责任删除它。
10. 我们现在可以完成这个原始的 IRP，因为 FDO 驱动程序堆栈已经收到该 IRP 的复制品。
11. 我们必须返回 **STATUS_MORE_PROCESSING_REQUIRED**，因为该 IRP 已经被删除，其完成操作在转发 IRP 中处理。

上面代码处理了一个较复杂的问题。PnP 管理器初始化 PnP 请求，使其包含 **STATUS_NOT_SUPPORTED** 状态。因此通过测试这些 IRP 的结束状态就可以知道该 IRP 是否被驱动程序处理过。如果 IRP 以 **STATUS_NOT_SUPPORT** 完成，则 PnP 管理器就知道该 IRP 没有被任何驱动程序处理。如果 IRP 以其它状态完成，则 PnP 管理器可以了解到该 IRP 或者被某些驱动程序置成失败或者被成功地处理，并不是简单地被忽略。

象 MULFUNC 这样的驱动程序在创建 PnP 请求时也必须把 **IoStatus.Status** 设置为 **STATUS_NOT_SUPPORTED**。正如我所提到的，驱动程序校验器将检验这一点。但这个初始化会导致这样的问题：子堆栈(子设备 PDO 之上)中的某个设备在向我们以 PDO 角色出现的驱动程序下传某个 IRP 时可能会改变 **IoStatus.Status**。我们将创建一个转发 IRP，并预置 **IoStatus.Status** 为 **STATUS_NOT_SUPPORTED**，然后下传这个 IRP 到父堆栈(即我们以 FDO 驱动程序角色出现的堆栈)。如果转发 IRP 也以 **STATUS_NOT_SUPPORTED** 状态完成，那么原始 IRP 将以什么状态完成呢？它不应该是 **STATUS_NOT_SUPPORTED**，因为这将意味着任何子堆栈驱动程序都没有处理该 IRP(有一个驱动程序处理了，但却改变了主 IRP 的状态)。这就是使用 **needsvote** 标志的原因。

对于某些需要转发的 IRP，我们不用关心父驱动程序是否真正处理该 IRP。也就是说，父驱动程序不需要“表决”该 IRP。如果你仔细观察 **OnRepeaterComplete** 函数，你将看到我们并没有在这种情况下改变主 IRP 的结束状态。对于另一些要转发的 IRP，如果父堆栈驱动程序忽略该 IRP，我们并不能给出一个真正答案。对于这些 IRP，父驱动程序必须“表决”，而我们将使主 IRP 失败并置上 **STATUS_UNSUCCESSFUL** 状态。为了查明哪一个 IRP 属于“需要表决”类，哪一个 IRP 不属于，你可以查看 MULFUNC 例子中的 **RepeatRequest** 函数(在 **PlugPlayPdo.cpp** 中)。

如果有一个父驱动程序真正处理了该转发 IRP，我们就复制整个 **IoStatus** 域，包括 **Status** 和 **Information** 值，到主 IRP 中。**Information** 域可能包含某个查询的回答，并且这个复制步骤也是我们把回答上传的过程。

在 **RepeatRequest** 中我还做了另一件事，我把该 IRP 标记为未决并返回 **STATUS_PENDING**。大部分 PnP 请求都是以同步方式完成，所以 **IoCallDriver** 调用很可能立即开始该 IRP 的完成操作。这就是我为什么把该 IRP 标记为未决并使 I/O 管理器在完成主 IRP 时不必调度 APC。如果我们不从派遣函数中返回 **STATUS_PENDING** 状态，回想一下，**RepeatRequest** 是作为 **IRP_MJ_PNP** 派遣例程的子例程运行的，我们必须返回与完成该 IRP 使用的值完全相同的值。只有我们的完成例程知道是哪一个值，通过检测 **STATUS_NOT_SUPPORTED** 和 **needsvote** 标志。

处理设备删除

PnP 管理器了解了父 FDO 和其子 PDO 之间的父子关系。因此，当用户删除父设备时，PnP 管理器将自动删除其所有子设备。尽管有些奇怪，通常当父设备驱动程序收到 IRP_MN_REMOVE_DEVICE 请求时，它不应该删除子设备 PDO。PnP 管理器希望在底层硬件移去之前子设备 PDO 一直存在。因此多功能设备的驱动程序直到被通知删除父设备 FDO 时才删除子设备 PDO。当总线驱动程序在某个枚举过程中不能成功地报告某个设备时，它将收到 IRP_MN_REMOVE_DEVICE，这时它才可以删除这个子设备 PDO。

MULFUNC 在处理设备删除事件时删除了子设备的 PDO。

对于控制器类型的设备(相对于非标准多功能设备)，你的控制器驱动程序需要某些附加代码来枚举设备。我忽略了这些代码，因为我的例子设备的子设备在主设备出现时总是存在。另外在枚举操作前不要忘记恢复控制器的电源。

处理IRP_MN_QUERY_ID请求

父设备驱动程序要处理的最重要的 PnP 请求是 IRP_MN_QUERY_ID。PnP 管理器以多种形式发出这个请求，它们用于获得设备标识符，而这些标识符能定位子设备的 INF 文件。响应这些请求你只需要返回一个包含设备标识符的 MULTI_SZ 值(在 IoStatus.Information 中)。MULFUNC 设备有两个子设备(全是假的)，标识符为 *WCO0604 和 *WCO0605，即第六章的第四个和第五个设备。下面是它处理该查询请求的程序：

```
NTSTATUS HandleQueryId(PDEVICE_OBJECT pdo, PIRP Irp)
{
    PPDO_EXTENSION pdx = (PPDO_EXTENSION) pdo->DeviceExtension;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    PWCHAR idstring;
    switch (stack->Parameters.QueryId.IdType)
    {
        case BusQueryInstanceId:
            idstring = L"0000";                                     <--1
            break;
        case BusQueryDeviceID:
            if (pdx->flags & CHILDTYPEA)                         <--2
                idstring = LDIVERNAME L"\\"*WCO0604";
            else
                idstring = LDIVERNAME L"\\"*WCO0605";
            break;
        case BusQueryHardwareIDs:
            if (pdx->flags & CHILDTYPEA)                         <--3
                idstring = L"*WCO0604";
            else
                idstring = L"*WCO0605";
            break;
        default:
            return CompleteRequest(Irp, STATUS_NOT_SUPPORTED, 0);
    }
    ULONG nchars = wcslen(idstring);
    ULONG size = (nchars + 2) * sizeof(WCHAR);
    PWCHAR id = (PWCHAR) ExAllocatePool(PagedPool, size);
    wcscpy(id, idstring);
    id[nchars + 1] = 0;
    return CompleteRequest(Irp, STATUS_SUCCESS, (ULONG_PTR) id);
}
```

1. 实例标识符是一个字符串，它唯一标识总线上一个特定类型的设备。如果使用“0000”常量将限制计算机上只出现父类型的一个实例设备。
2. 设备标识符是一个形式为“enumerator\type”的字符串，它基本上提供了硬件注册表键的两个部分。例如，ChildA 设备的硬件键将是\Enum\Mulfunc*WCO0604\0000。
3. 硬件标识符是唯一标识设备类型的串。在这里，我们制作了两个假 EISA 标识符*WCO0604 和*WCO0605。

注意

如果你要以我在这介绍的方式构造一个设备标识符，你应该用自己的设备名代替 MULFUNC。应该强调一下，你不应该象我的例子程序那样把名字硬编码到常量中，这里的代码使用的 LDRIVERNAME 常量定义在 MULFUNC 工程的 DRIVER.H 文件中。

Windows 98 的 PnP 管理器可以容忍你提供一个与硬件标识符相同的设备标识符，但 Windows 2000 的 PnP 管理器不允许这样做。把枚举器名组合到设备标识符中非常困难。通过调用 **IoGetDeviceProperty** 来获得 PDO 的枚举器名将导致一个 bug check，因为 PnP 管理器最后将得到一个 NULL 串指针。使用父设备枚举器名将导致奇怪的结果，PnP 管理器在你把父设备删除后又把子设备还给你！

处理IRP_MN_QUERY_DEVICE_RELATIONS请求

最后要考虑的 PnP 请求是 IRP_MN_QUERY_DEVICE_RELATIONS。回想一下，FDO 驱动程序是以提供一组子 PDO 列表来回答这个请求的。然而，戴上 PDO 的帽子，父设备驱动程序仅需要提供 PDO 的地址来回答一个称为目标设备关系的请求。

```
NTSTATUS HandleQueryRelations(PDEVICE_OBJECT pdo, PIRP Irp)
{
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    NTSTATUS status = Irp->IoStatus.Status;
    if (stack->Parameters.QueryDeviceRelations.Type == TargetDeviceRelation)
    {
        PDEVICE_RELATIONS newrel = (PDEVICE_RELATIONS) ExAllocatePool(PagedPool,
        sizeof(DEVICE_RELATIONS));
        newrel->Count = 1;
        newrel->Objects[0] = pdo;
        ObReferenceObject(pdo);
        status = STATUS_SUCCESS;
        Irp->IoStatus.Information = (ULONG_PTR) newrel;
    }
    Irp->IoStatus.Status = status;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return
}
```

处理子设备资源

如果你的设备是控制器类型，那么插入该控制器的子设备可能会要求自己的 I/O 资源。如果你有一个自动的方式获得该设备的资源需求，你可以在响应 IRP_MN_QUERY_RESOURCE_REQUIREMENTS 请求时返回该设备的需求列表。如果没有自动方式来获得这个资源需求，则子设备的 INF 文件应有一个 LogConfig 段能确定这些信息。

如果你需要应付多功能设备，碰巧父设备又替所有子设备获取了所有必须的 I/O 资源。如果子设备有单独的 WDM 驱动程序，你必须设计一个方法按子设备分开这些资源，并让每个子设备知道哪个资源属于它。这并不简单。PnP 管理器通常在 IRP_MN_START_DEVICE 请求中告诉子设备其资源的分配情况(详细讨论见下一章)。这里

没有一个普遍的方法强制 PnP 管理器使用你的资源而不是它分配的资源。注意，响应需求查询或过滤请求并没有什么帮助，因为那些请求处理的资源需求可以被 PnP 管理器更新。

Microsoft 的 MF.SYS 驱动程序通过使用系统资源仲裁器的某些内部接口来处理资源细分，这些内部接口对作为第三方开发者的我们是不能访问的。有两个不同的方式可以细分资源：一个用于 Windows 2000，一个用于 Windows 98。因为我们不能象 MF.SYS 那样做，所以我们需要寻找另外一种方式来细分父设备的资源。我并没有尝试去实现这两种方法，但我希望听到有某个读者实现了这两个想法。

如果你能控制所有子设备的功能驱动程序，那么你的父设备驱动程序可以输出一个直接调用的接口。子设备驱动程序通过向父设备驱动程序发送 IRP_MN_QUERY_INTERFACE 请求来获得一个指向父设备接口描述符的指针。这些子设备应该在设备启动和停止时调用父设备驱动程序中的函数来获得和释放父设备拥有的资源。

如果你不能修改子设备的功能驱动程序，你可以在每个子设备的 FDO 上安装一个微小的上层过滤器来解决这个资源细分问题。这个过滤器的唯一目的就是向每个 IRP_MN_START_DEVICE 请求插入一组细分配后的资源。过滤器可以经由一个直接调用接口与父设备驱动程序通信。

PnP 通知

Windows 2000 和 Windows 98 提供了一种方法来通知用户模式和内核模式部件刚发生的 PnP 事件。Windows 95 有一个 WM_DEVICECHANGE 消息，用户模式可以通过处理该消息来监视，或控制系统中的硬件和电源配置的改变。新操作系统中的 WM_DEVICECHANGE 消息还允许用户模式程序容易地检测到某驱动程序允许或禁止寄存的设备接口。内核模式驱动程序也可以注册类似的通知。

注意

参见平台 SDK 中关于 WM_DEVICECHANGE、RegisterDeviceNotification、UnregisterDeviceNotification 的文档。我将给出关于使用这个消息和相关 API 的例子，但我不能解释它们的所有可能的用法。在后面的某些解释中我将假设你熟悉 Microsoft 的 MFC 编程。

WM_DEVICECHANGE 扩充

带有窗口的应用程序可以预定与特定接口 GUID 相关的 WM_DEVICECHANGE 消息。下面是一个例子：

```
int CAutoLaunch::OnCreate(LPCREATESTRUCT csp)
{
    DEV_BROADCAST_DEVICEINTERFACE filter = {0};
    filter.dbcc_size = sizeof(filter);
    filter.dbcc_devicetype = DBT_DEVTYP_DEVICEINTERFACE;
    filter.dbcc_classguid = GUID_AUTOLAUNCH_NOTIFY;
    HDEVNOTIFY hNotification = RegisterDeviceNotification(m_hWnd, (PVOID) &filter,
    DEVICE_NOTIFY_WINDOW_HANDLE);
    ...
}
```

这里的关键语句是调用 **RegisterDeviceNotification** 函数，它要求 PnP 管理器在任何人允许或禁止 GUID_AUTOLAUNCH_NOTIFY 接口时向我们的窗口发送一个 WM_DEVICECHANGE 消息。所以，如果一个设备驱动程序在其 AddDevice 函数中用这个接口 GUID 调用 **IoRegisterDeviceInterface**，那么当那个驱动程序调用 **IoSetDeviceInterfaceState** 允许或禁止该寄存的接口时，我们将得到通知。

注意

平台 SDK 告诉你调用 UnregisterDeviceNotification 来反寄存用 RegisterDeviceNotification 得到的通知句柄。这对于

Windows 2000 确实可以，但不能用于 Windows 98。虽然 Windows 98 支持调用 RegisterDeviceNotification 作为预定属于某设备接口 WM_DEVICECHANGE 消息的一种方式，但 UnregisterDeviceNotification 好象不太可靠。在我的测试中，调用这个函数将导致随机的系统崩溃。而我取消 UnregisterDeviceNotification 调用后系统好象就不再出现异常。

WM_DEVICECHANGE 消息的处理程序应该象这样：

```
BOOL CAutoLaunch::OnDeviceChange(UINT evtype, DWORD dwData)
{
    _DEV_BROADCAST_HEADER* dbhdr = (_DEV_BROADCAST_HEADER*) dwData;
    if (!dbhdr || dbhdr->dbcd_devicetype != DBT_DEVTYP_DEVICEINTERFACE)
        return TRUE;
    PDEV_BROADCAST_DEVICEINTERFACE p = (PDEV_BROADCAST_DEVICEINTERFACE) dbhdr;
    CString devname = p->dbcc_name;
    if (evtype == DBT_DEVICEARRIVAL)
        <handle arrival>
    else if (evtype == DBT_DEVICEREMOVECOMPLETE)
        <handle removal>
    return TRUE;
}
```

这个程序忽略了所有不属于设备接口的消息。**devname** 变量是将要到来或离去的设备的符号连接名。(这个名字与调用 **SetupDiGetDeviceInterfaceDetail** 获得的以及传递给 **CreateFile** 的是同一个名字) 第十二章详细介绍了如何用各种 **SetupDiXxx** API 来了解新设备的相关信息。

何时关闭设备句柄

PnP 管理器不允许你在某应用程序还持有设备句柄时删除设备对象。为了删除设备，驱动程序必须促使应用程序关闭设备句柄。对在上面段中谈到的设备接口通知改变消息做些修改就可以解决这个问题。

如果某应用程序拥有你的设备的句柄，它应该调用 **RegisterDeviceNotification** 来寄存句柄通知。

```
DEV_BROADCAST_HANDLE filter = {0};
filter.dbch_size = sizeof(filter);
filter.dbch_devicetype = DBT_DEVTYP_HANDLE;
filter.dbch_handle = m_hDevice; // the device handle
HDEVNOTIFY hNotify = RegisterDeviceNotification(m_hWnd, &filter,
DEVICE_NOTIFY_WINDOW_HANDLE);
```

现在，应用程序应注意 **wParam**(事件代码)等于 **DBT_DEVICEQUERYREMOVE**，**devicetype** 为 **DBT_DEVTYP_HANDLE** 的 WM_DEVICECHANGE 消息。该消息意味着接口将要被禁止，因此你应该关闭你拥有的设备句柄。你还应该在该消息的处理程序中无条件地返回 **TRUE**。

注意

根据平台 SDK 文档，你在响应 **DBT_DEVICEQUERYREMOVE** 消息时应返回 **BROADCAST_QUERY_DENY**，这个特殊的返回值表示你不希望设备被删除或禁止。我在各种版本的 Windows 98 和 Windows 2000 中测试了这个返回值但却得到了不一致的结果。所以我建议你的应用程序应总使这个查询成功。

PNPEVENT 例子

例子驱动程序 PNPEVENT(更恰当地说是 TEST 程序)显示了如何使用 WM_DEVICECHANGE 消息检测一个已寄存接口的到来和离开，以及你何时必须关闭设备句柄以允许设备的禁止和删除。你可以在安装 PNPEVENT“设备”之前或之后运行 TEST 程序看到这些。

在 TEST 程序对话框中，你将看到一个 Send Event 按钮。点击这个按钮将使驱动程序产生一个定制 PnP 事件。稍后我再讨论定制事件。取关于一个定制事件的用户模式通知不会成功，所以除非你运行 PNPMON 的测试程序，否则当你点击这个按钮时什么也不会发生。

Windows 2000 的服务通知

Windows 2000 的服务程序也能预定 PnP 通知。服务程序应该先调用 **RegisterServiceCtrlHandlerEx** 寄存一个扩展的控制处理函数，然后它才能寄存关于设备接口改变的服务控制通知。例如，观察下面代码(见 AUTOLANCH 例子):

```
DEV_BROADCAST_DEVICEINTERFACE filter = {0};  
filter.dbcc_size = sizeof(filter);  
filter.dbcc_devicetype = DBT_DEVTYPE_DEVICEINTERFACE;  
filter.dbcc_classguid = GUID_AUTOLAUNCH_NOTIFY;  
m_hNotification = RegisterDeviceNotification(m_hService, (PVOID) &filter,  
DEVICE_NOTIFY_SERVICE_HANDLE);
```

这里，**m_hService** 是由服务管理器提供的服务句柄，**DEVICE_NOTIFY_SERVICE_HANDLE** 指出你寄存的是服务控制通知而不是窗口消息。在收到 **SERVICE_CONTROL_STOP** 命令后，你应该反寄存这个通知句柄:

```
UnregisterDeviceNotification(m_hNotification);
```

当包含某接口 GUID 的 PnP 事件发生时，系统调用你的扩展服务控制处理函数:

```
DWORD __stdcall HandlerEx(DWORD ctlcode, DWORD evtype, PVOID evdata, PVOID context)  
{  
}
```

ctlcode 将等于 **SERVICE_CONTROL_DEVICEEVENT**，**evtype** 将等于 **DBT_DEVICEARRIVAL** 或其它 **DBT_Xxx** 代码，**evdata** 将为 **DEV_BROADCAST_DEVICEINTERFACE** 结构的地址，**context** 将为你在调用 **RegisterServiceCtrlHandlerEx** 时指定的任意上下文值。

内核模式通知

WDM 驱动程序可以使用 **IoRegisterPlugPlayNotification** 函数来申请接口和句柄通知。下面是从 PNPMON 例子驱动程序中摘录的语句，PNPMON 程序通过一个 I/O 控制(IOCTL)操作来为接口的到来和离开寄存通知。

```
status = IoRegisterPlugPlayNotification(EventCategoryDeviceInterfaceChange,  
                                         PNPNOTIFY_DEVICE_INTERFACE_INCLUDE_EXISTING_INTERFACES,  
                                         &p->guid,  
                                         pdx->DriverObject,  
                                         (PDRIVER_NOTIFICATION_CALLBACK_ROUTINE)  
                                         OnPnpNotify,  
                                         reg,  
                                         &reg->InterfaceNotificationEntry);
```

第一个参数指出我们希望在指定接口 GUID 被某人允许或禁止时能收到通知。第二个参数是一个标志，指出我们希望在接口 GUID 的所有实例都被允许时得到回调。第三个参数是接口 GUID，它来自应用程序的 IOCTL。第四个参数是驱动程序对象的地址。PnP 管理器向这个对象增加引用，所以当有未关闭的通知句柄时驱动程序对象不会被卸载。第五个参数是通知回调函数的地址。第六个参数是这个回调函数的上下文参数。在这里，我指定了一个结构(**reg**)的地址，该结构包含了与这个注册调用相关的信息。第七个参数给出了 PnP 管理器用于记录通知句柄的地址。最后我们将用这个句柄调用 **IoUnregisterPlugPlayNotification** 函数。

关闭注册句柄需要调用 **IoUnregisterPlugPlayNotification** 函数，因为 **IoRegisterPlugPlayNotification** 向你的驱动程序对象增加了一个引用，所以在 **DriverUnload** 函数中调用这个例程不会得到什么好处。而 **DriverUnload** 只有当引用落回 0 时才被调用，所以如果 **DriverUnload** 本身中有反寄存调用将得不到调用。这个问题比较容易解决，你只需要在合适的时间调用反寄存函数，例如你可以在处理一个接口时做反寄存操作。

为要允许的接口给定一个符号连接名，你也可以用设备的连接名来请求通知。例如：

```
PUNICODE_STRING SymbolicLinkName;           // input to this process
PDEVICE_OBJECT DeviceObject;                 // an output
PFILE_OBJECT FileObject;                    // another output
IoGetDeviceObjectPointer(&SymbolicLinkName, 0, &FileObject, &DeviceObject);
IoRegisterPlugPlayNotification(EventCategoryTargetDeviceChange,
    0,
    FileObject,
    pdx->DriverObject,
    (PDRIVER_NOTIFICATION_CALLBACK_ROUTINE) OnPnpNotify,
    reg,
    &reg->HandleNotificationEntry);
```

你不能把这段代码放到你的 PnP 事件处理程序中。在内部，**IoGetDeviceObjectPointer** 对命名设备对象执行一个打开操作。如果目标设备正要执行某种 PnP 操作，则发生死锁。你应该改为调用 **IoQueueWorkItem** 来调度工作项。第九章将详细介绍工作项(work item)。PNPMON 例子驱动程序显示了如何在这种特殊情况下使用工作项。

这些注册调用的通知将以调用你指定的回调函数的形式出现：

```
NTSTATUS OnPnpNotify(PPLUGPLAY_NOTIFICATION_HEADER hdr, PVOID Context)
{
    ...
    return STATUS_SUCCESS;
}
```

PLUGPLAY_NOTIFICATION_HEADER 结构是 PnP 管理器用于通知的几种不同结构的公共头结构：

```
typedef struct _PLUGPLAY_NOTIFICATION_HEADER {
    USHORT Version;
    USHORT Size;
    GUID Event;
} PLUGPLAY_NOTIFICATION_HEADER, *PPLUGPLAY_NOTIFICATION_HEADER;
```

Event(GUID) 指出报告给你的是何种事件。见表 6-6。DDK 头文件 WDMGUID.H 包含了这些 GUID 的定义。

表 6-6. PnP 通知 GUID

GUID 名	通知的目的
GUID_HWPROFILE_QUERY_CHANGE	可以改变到新硬件 profile 上吗？
GUID_HWPROFILE_CHANGE_CANCELLED	上一个改变要求已被取消

GUID_HWPROFILE_CHANGE_COMPLETE	上一个改变要求已被完成
GUID_DEVICE_INTERFACE_ARRIVAL	设备接口刚被允许
GUID_DEVICE_INTERFACE_REMOVAL	设备接口刚被禁止
GUID_TARGET_DEVICE_QUERY_REMOVE	设备对象可以删除吗？
GUID_TARGET_DEVICE_REMOVE_CANCELLED	上一个删除要求已被取消
GUID_TARGET_DEVICE_REMOVE_COMPLETE	上一个删除要求已被完成

如果你收到 DEVICE_INTERFACE 通知，你可以把回调函数的 **hdr** 参数强制转换成指向下面结构的指针：

```
typedef struct _DEVICE_INTERFACE_CHANGE_NOTIFICATION {
    USHORT Version;
    USHORT Size;
    GUID Event;
    GUID InterfaceClassGuid;
    PUNICODE_STRING SymbolicLinkName;
} DEVICE_INTERFACE_CHANGE_NOTIFICATION,
*PDEVICE_INTERFACE_CHANGE_NOTIFICATION;
```

在这个接口改变通知结构中，**InterfaceClassGuid** 是接口 GUID，**SymbolicLinkName** 是刚被允许或禁止的接口实例名。

如果你收到任何一种 TARGET_DEVICE 通知，你可以把 **hdr** 参数强制转换成指向下面结构的指针：

```
typedef struct _TARGET_DEVICE_REMOVAL_NOTIFICATION {
    USHORT Version;
    USHORT Size;
    GUID Event;
    PFILE_OBJECT FileObject;
} TARGET_DEVICE_REMOVAL_NOTIFICATION, *PTARGET_DEVICE_REMOVAL_NOTIFICATION;
```

FileObject 是你用于请求通知的文件对象。

最后，如果你收到任何一种 HWPROFILE_CHANGE 通知，**hdr** 将是下面结构的指针：

```
typedef struct _HWPROFILE_CHANGE_NOTIFICATION {
    USHORT Version;
    USHORT Size;
    GUID Event;
} HWPROFILE_CHANGE_NOTIFICATION, *PHWPROFILE_CHANGE_NOTIFICATION;
```

这里并没有包含比头结构本身更多的信息，它仅仅是头结构的另一个不同的定义名。

使用这些通知的一种方式是为整个设备接口类实现一个过滤器驱动程序。(实现过滤器驱动程序有一个标准方法，或者为一个单独驱动程序或者为一类设备，这基于注册表中的设置。我将在第九章中讨论这个主题。这里，我正谈到的是过滤所有寄存了特定接口的设备，所以没有其它机制可以选择) 在驱动程序的 **DriverEntry** 例程中，你可以为一个或多个接口 GUID 寄存 PnP 通知。当你收到到达通知时，你使用 **IoGetDeviceObjectPointer** 例程打开一个设备对象，然后寄存与设备关联的目标设备通知。你还从 **IoGetDeviceObjectPointer** 得到一个设备对象指针，并且你可以调用 **IoCallDriver** 向该设备发送 IRP。你应该时刻监视 **GUID_TARGET_DEVICE_QUERY_REMOVE** 通知，因为你必须在设备删除前解除文件对象的引用。

PNPMON 例子

PNPMON 例子显示了如何在内核模式中寄存和处理 PnP 通知。为了让你能看到真正在计算机上运行的例子，我设计了 **PNPMON**，它把通知简单地传递回用户模式应用程序(**TEST** 程序)。这比较蠢，因为用户模式应用程序可以通过调用 **RegisterDeviceNotification** 获得这些通知。

PNPMON 不同于本书中的其它驱动程序例子。它可以作为一个用户模式应用程序的辅助程序而被动态地加载。其它驱动程序则用于管理真正的或假想的硬件。用户模式应用程序使用服务管理器 API 来装入 **PNPMON**，而 **PNPMON** 将在 **DriverEntry** 例程中创建一个设备对象，以便应用程序能使用 **DeviceIoControl** 做只有在内核模式中才能做的事。当应用程序退出时，它关闭其句柄并调用服务管理器终止驱动程序。

PNPMON 还包含一个 Windows 98 的 VxD 驱动程序，该 VxD 可以被 **test** 应用程序动态装入。在 Windows 98 中使用未公开的函数也可以动态装入一个 WDM 驱动程序(**_NtKernLoadDriver**，如果你敢)，但用这种方法装入的驱动程序没有办法卸载。你不需要凭借未公开的函数，因为 VxD 可以直接调用大部分 WDM 的支持例程，使用 Windows 98DDK 中的 **WDMVXD** 输入库。所以你应该在你的 VxD 工程中把 **WDM.H** 包含语句放到 VxD 头文件前边，并在连接器的输入列表中加入 **WDMVXD.CLB**。因此 **PNPMON.VXD** 仅仅把自己当做一个 WDM 驱动程序来寄存 PnP 通知，并与 **PNPMON.SYS** 相同，都支持同一个 **IOCTL** 接口。

定制通知

最后，我将解释 WDM 驱动程序如何生成定制 PnP 通知。为了发出定制 PnP 事件，先创建一个定制通知结构的实例，然后调用 **IoReportTargetDeviceChange** 或 **IoReportTargetDeviceChangeAsynchronous** 函数。异步特性将使调用者立即返回，同步特性将使调用者等待一段时间，以我的经验，这段时间将持续到通知被发出后。通知结构有如下声明：

```
typedef struct _TARGET_DEVICE_CUSTOM_NOTIFICATION {
    USHORT Version;
    USHORT Size;
    GUID Event;
    PFILE_OBJECT FileObject;
    LONG NameBufferOffset;
    UCHAR CustomDataBuffer[1];
} TARGET_DEVICE_CUSTOM_NOTIFICATION, *PTARGET_DEVICE_CUSTOM_NOTIFICATION;
```

Event 是你为这个通知定义的定制 GUID。**FileObject** 将为 NULL，PnP 管理器将把通知送到打开某文件对象的驱动程序，而这个文件对象的 PDO 与 **IoReportXxx** 调用中指定的 PDO 相同。**CustomDataBuffer** 包含你选择的任意二进制数据。如果你没有任何串数据则 **NameBufferOffset** 为 -1，否则它将是那个二进制数据的长度。从 **Size** 中减去 **CustomDataBuffer** 的域偏移可以得到整个有效数据的大小。

下面解释了为什么你在 **test** 对话框中按下 **Send Event** 按钮会生成 PNPEVENT 定制通知：

```
struct _RANDOM_NOTIFICATION
: public _TARGET_DEVICE_CUSTOM_NOTIFICATION {
    WCHAR text[14];
};

...
_RANDOM_NOTIFICATION notify;
notify.Version = 1;
notify.Size = sizeof(notify);
notify.Event = GUID_PNPEVENT_EVENT;
notify.FileObject = NULL;
notify.NameBufferOffset = FIELD_OFFSET(RANDOM_NOTIFICATION, text)
    - FIELD_OFFSET(RANDOM_NOTIFICATION, CustomDataBuffer);
*(PULONG)(notify.CustomDataBuffer) = 42;
wcscpy(notify.text, L"Hello, world!");
```

```
IoReportTargetDeviceChangeAsynchronous(pdx->Pdo, &notify, NULL, NULL);
```

即，**PNPEVENT** 生成了一个定制通知，该通知的数据部分包含了数值 **42** 和一个串“**Hello, world!**”。

顺便提一下，我推荐你使用异步报告 API，因为它可以立即返回，为此你必须用 **NTDDK.H** 替换 **WDM.H**，但连接时必须包含 **WDM.LIB** 和 **NTOSKRNL.LIB** 库。

如果通知回调例程得到的通知结构中 **Event** 域含有非标准 **GUID**，可以认为它是某人的定制通知 **GUID**，因此在你处理其 **CustomDataBuffer** 之前需要了解这个 **GUID** 的含义。

用户模式应用程序也可以接收定制事件通知，但我现在还不能使其工作。

Windows 98 兼容问题

Windows 98 从不发出 `IRP_MN_SURPRISE_REMOVAL` 请求。因此，WDM 驱动程序需要把非正常的 `IRP_MN_REMOVE_DEVICE` 请求当做意外删除设备。本章介绍的代码例子考虑到了这一点，如果遇到这样的 IRP 它将调用 `AbortRequests` 和 `StopDevice`。

Windows 98 在调用 `IoReportTargetDeviceChange` 函数时会失败并返回 `STATUS_NOT_IMPLEMENTED`。并且它根本就没有输出 `IoReportTargetDeviceChangeAsynchronous` 函数；因此调用该函数的驱动程序不能装到 Windows 98 中。参考附录 A，查看如何解决这个问题以及其它不支持函数的问题。

Windows 98 的结构不允许以内核模式阻塞方式来等待用户模式程序完成任务。这个事实使我难于把我的 USB 例子驱动程序(USBINT)连接起来。该例子驱动程序的 `test` 程序打开一个句柄并发出一个异步 `DeviceIoControl` 调用。如果你现在拔掉设备，你想会发生什么：驱动程序将收到一个 `IRP_MN_SURPRISE_REMOVAL`，而它将取消未处理的 `DeviceIoControl`。`test` 程序然后关闭其句柄。这时，我们再回到驱动程序这边看看，`REMOVE_DEVICE` 处理程序将被阻塞在 `IoReleaseRemoveLockAndWait` 调用上。当 `IRP_MJ_CLOSE` 到来时，驱动程序将释放最后获得的自旋锁并允许设备删除处理。这个过程在 Windos 2000 上进行得很好，但在 Windows 98 中会挂起，因为 `test` 程序没有运行的机会，因此也就不会关闭其句柄。(Windows 98 没有 `SURPRISE_REMOVAL`，但我们得到的 `REMOVE_DEVICE` 可以起到同样的作用) 利用 `QUERY_REMOVE` 的代码将不会挂起系统。所以，如果你的设备可以被用户热拔插，那么在 Windows 98 中，如果你有一个打开的句柄，就不要再获取删除自旋锁。

第七章：读写数据

经过以前各章讲述的基础内容，必然引出了本章的内容，即如何在设备上读写数据。我将讨论几个重要的服务函数，这些函数能对插入传统总线(例如 **PCI**)上的设备进行重要操作。由于许多设备使用硬件中断向系统软件通知有关 I/O 完成或异常事件，因此我还将讨论如何处理中断。中断处理通常需要调度一个 **DPC**，所以我还将描述 **DPC** 机制。最后，我将讲述如何在设备和内存之间安排 **DMA** 传输。

- 配置设备
- 寻址数据缓冲区
- 端口与寄存器
- 响应中断
- 直接内存存取(DMA)

配置设备

在上一章中，我们讨论了 PnP 管理器发送给你的各种 IRP_MJ_PNP。PnP 管理器为你的设备分配 I/O 资源，而 IRP_MN_START_DEVICE 是把这个 I/O 资源分配信息传递给你的运载工具。我向你展示了如何获得两组原始的和经过转换的资源描述信息，以及如何调用 **StartDevice** 辅助函数，该函数有如下原型：

```
NTSTATUS  
StartDevice(PDEVICE_OBJECT fdo, PCM_PARTIAL_RESOURCE_LIST raw,  
PCM_PARTIAL_RESOURCE_LIST translated)  
{  
    ...  
}
```

现在我将解释如何使用这些资源列表。简单地说，你首先从转换后的资源表中提取描述信息，然后用这些描述信息创建一个内核对象，最后，使用这个内核对象访问你的硬件。

CM_PARTIAL_RESOURCE_LIST 结构包含一个计数器和一个 CM_PARTIAL_DESCRIPTOR 结构的数组，数组元素的描述见图 7-1。数组中每个资源描述符都有一个 **Type** 成员，它指出所描述的资源类型，另外还有一些附加成员，它们描述某些资源的细节信息。不要对该数组中的信息感到惊讶：如果你的设备使用一个 IRQ 和一组 I/O 端口，你的数组将包含两个资源描述符。一个描述符描述 IRQ，另一个描述符描述 I/O 端口范围。不幸的是，你不能提前知道这些描述符在数组中出现的位置。因此，你的 StartDevice 函数必须用循环先把资源值提取到一组局部变量中，然后再处理这些资源信息。

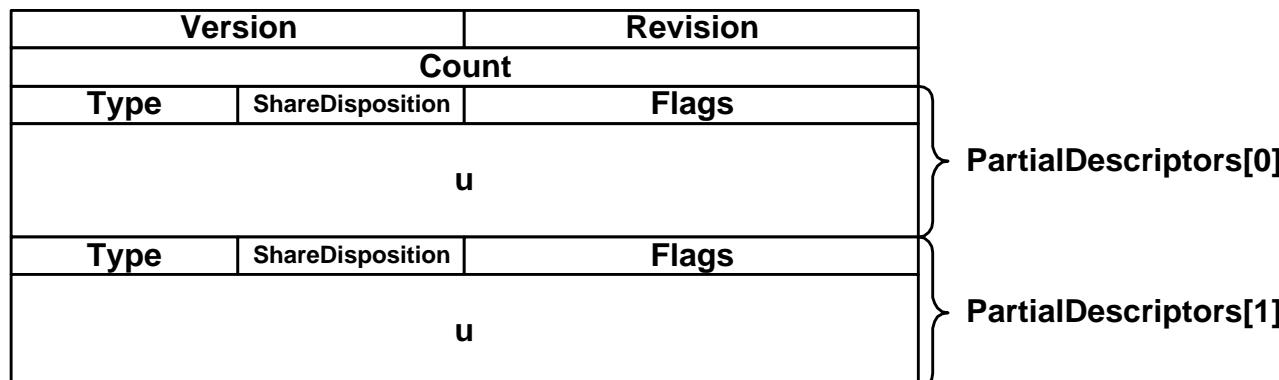


图 7-1. 局部资源表结构

你的 **StartDevice** 函数基本上应该象下面这样：

```
NTSTATUS  
StartDevice(PDEVICE_OBJECT fdo, PCM_PARTIAL_RESOURCE_LIST raw,  
PCM_PARTIAL_RESOURCE_LIST translated)  
{  
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;  
    PCM_PARTIAL_RESOURCE_DESCRIPTOR resource = translated->PartialDescriptors; <  
    ULONG nres = translated->Count; <-2  
    <local variable declarations> <  
    for (ULONG i = 0; i < nres; ++i, ++resource) <  
    { <  
        switch (resource->Type) <  
        { <  
            case CmResourceTypePort: <  
                <save port info in local variables> <  
                break; <  
        } <
```

```

case CmResourceTypeInterrupt:
    <save interrupt info in local variables>
    break;
case CmResourceTypeMemory:
    <save memory info in local variables>
    break;
case CmResourceTypeDma:
    <save DMA info in local variables>
    break;
}
}
<use local variables to configure driver & hardware>           <--5
IoSetDeviceInterfaceState(&pdx->ifname, TRUE);                  <--6
}

```

1. 我用这个 **resource** 指针来指向可变长数组中的当前资源描述符。在循环的结尾，该指针将指向最后有效描述符后面地址。
2. 资源表中的 **Count** 成员指出数组 **PartialDescriptors** 中有多少个描述符。
3. 你应该为每个希望接收的 I/O 资源声明一个适当的局部变量。我将后面的标准 I/O 资源处理中详细讨论这些细节。
4. 循环中使用 **switch** 语句把资源描述保存到适当的局部变量中。在本文中，我安装了一个仅需要一个 I/O 端口范围和一个中断的设备，这个设备希望得到 **CmResourceTypePort** 和 **CmResourceTypeInterrupt** 类型的资源。此外，我还详细讨论另外两个标准资源类型，**CmResourceTypeMemory** 和 **CmResourceTypeDma**。
5. 一旦跳出循环，在各个 **case** 语句中初始化的局部变量将包含你需要的资源信息。
6. 如果你在 **AddDevice** 期间注册了一个设备接口，这时就应该使能这个接口，以便应用程序能找到设备并打开设备句柄。

如果你在某个资源类型上有多个资源，那么你需要创造一种方法分开资源描述符。为了给出一个具体的例子(本例纯属虚构)，假设某设备使用一个 4KB 的内存范围来实现控制，用另外的 16KB 内存范围作为数据捕获缓冲区。那么你希望 PnP 管理器发给你两个 **CmResourceTypeMemory** 资源。控制内存是 4KB 长的内存块，而数据内存是 16KB 长的内存块。如果该设备的资源有可以区别的特征，如这个例子中两个资源的需求长度，你就可以知道哪个资源是哪个。

当遇到同类型多资源的情况时，不要假设资源描述符会以在设备配置空间表中出现的顺序出现，也不要假设同一个总线驱动程序在不同的平台或不同的操作系统版本间总是按同一个顺序构造资源描述符。

我会在本章的适当地方解释如何处理这四种标准 I/O 资源类型。表 7-1 列出了处理每种资源类型的关键步骤。

表 7-1. I/O 资源处理步骤概述

资源类型	处理概述
Port	可能映射端口范围；应在设备扩展中保存端口范围基址
Memory	映射内存范围；应在设备扩展中保存内存范围基址
Dma	调用 IoGetDmaAdapter 函数创建适配器对象
Interrupt	调用 IoConnectInterrupt 函数创建中断对象，中断对象指向 ISR(中断服务例程)

寻址数据缓冲区

当应用程序发起一个读或写操作时，通过给出一个用户模式虚拟地址和长度，应用程序向 I/O 管理器提供了一个数据缓冲区。正如我在第三章中提到的，内核模式驱动程序几乎从不使用用户模式虚拟地址访问内存，因为 你不能把线程上下文确定下来。Windows 2000 为驱动程序访问用户模式数据缓冲区提供了三种方法：

- 在 *buffered* 方式中，I/O 管理器先创建一个与用户模式数据缓冲区大小相等的系统缓冲区。而你的驱动程序将使用这个系统缓冲区工作。I/O 管理器负责在系统缓冲区和用户模式缓冲区之间复制数据。
- 在 *direct* 方式中，I/O 管理器锁定了包含用户模式缓冲区的物理内存页，并创建一个称为 MDL(内存描述符表)的辅助数据结构来描述锁定页。因此你的驱动程序将使用 MDL 工作。
- 在 *neither* 方式中，I/O 管理器仅简单地把用户模式的虚拟地址传递给你。而使用用户模式地址的驱动程序应十分小心。

图 7-2 显示了前两种方法。后一种方法实际上并不是方法，因为系统没有为你获取数据而给予任何帮助。

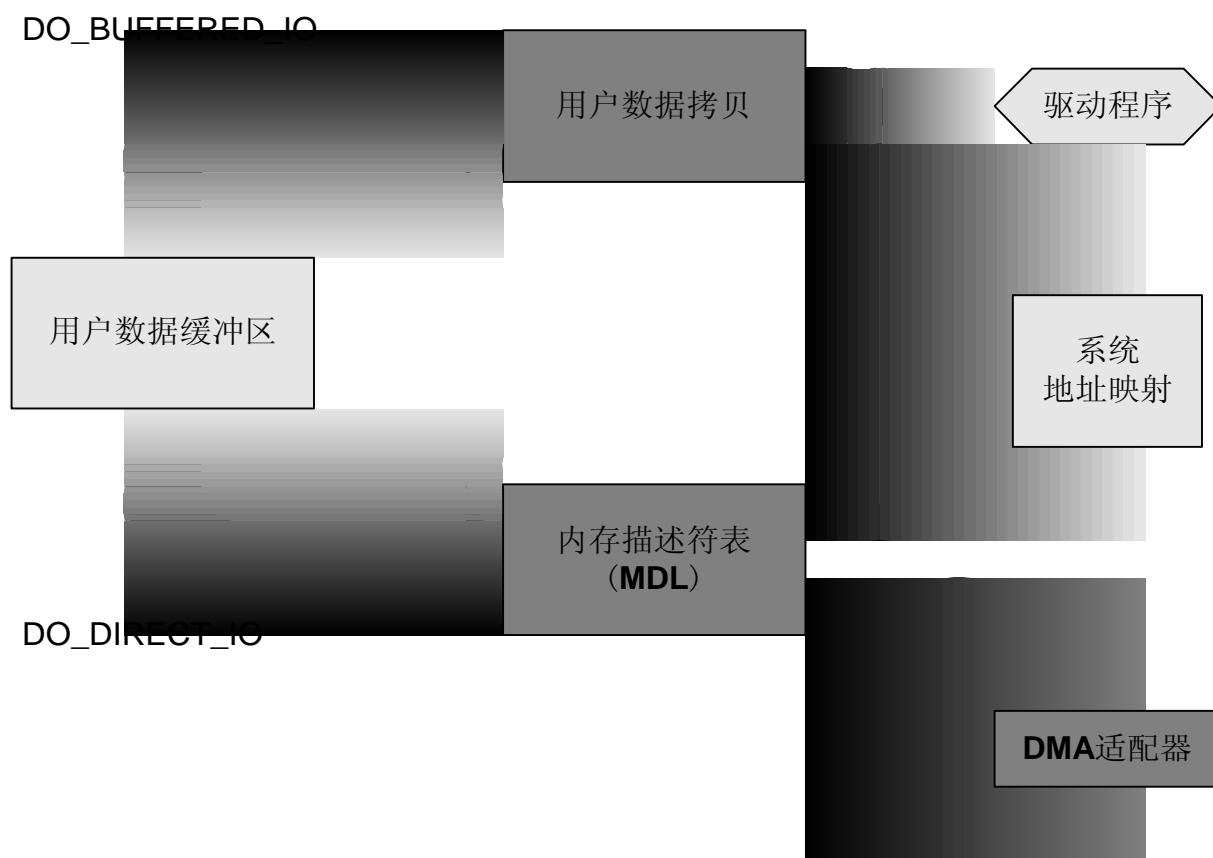


图 7-2. 访问用户模式数据缓冲区

指定缓冲方式

为了指定设备读写的缓冲方式，你应该在 **AddDevice** 函数中，在创建设备对象后，立即设置其中的标志位：

```
NTSTATUS AddDevice(...)  
{  
    PDEVICE_OBJECT fdo;  
    IoCreateDevice(..., &fdo);  
    fdo->Flags |= DO_BUFFERED_IO;  
    <or>  
    fdo->Flags |= DO_DIRECT_IO;  
    <or>  
    fdo->Flags |= 0;           // i.e., neither direct nor buffered  
}
```

这之后你不能改变缓冲方式的设置，因为过滤器驱动程序将复制这个标志设置，并且，如果你改变了设置，过滤器驱动程序没有办法知道这个改变。

Buffered 方式

当 I/O 管理器创建 IRP_MJ_READ 或 IRP_MJ_WRITE 请求时，它探测设备的缓冲标志以决定如何描述新 IRP 中的数据缓冲区。如果 DO_BUFFERED_IO 标志设置，I/O 管理器将分配与用户缓冲区大小相同的非分页内存。它把缓冲区的地址和长度保存到两个十分不同的地方，见下面代码片段中用粗体字表示的语句。你可以假定 I/O 管理器执行下面代码(注意这并不是 Windows NT 的源代码)：

```
PVOID uva;           // user-mode virtual buffer address
ULONG length;        // length of user-mode buffer

PVOID sva = ExAllocatePoolWithQuota(NonPagedPoolCacheAligned, length);
if (writing)
    RtlCopyMemory(sva, uva, length);

Irp->AssociatedIrp.SystemBuffer = sva;

PIO_STACK_LOCATION stack = IoGetNextIrpStackLocation(Irp);
if (reading)
    stack->Parameters.Read.Length = length;
else
    stack->Parameters.Write.Length = length;

<code to send and await IRP>

if (reading)
    RtlCopyMemory(uva, sva, length);

ExFreePool(sva);
```

可以看出，系统缓冲区地址被放在 IRP 的 **AssociatedIrp.SystemBuffer** 域中，而数据的长度被放到 **stack->Parameters** 联合中。在这个过程中还包含作为驱动程序开发者不必了解的额外细节。例如，读操作之后的数据复制工作实际发生一个 APC 期间，在原始线程的上下文中，由一个与构造该 IRP 完全不同的子例程执行。I/O 管理器把用户模式虚拟地址(uva 变量)保存到 IRP 的 **UserBuffer** 域中，这样一来复制操作就可以找到这个地址。但你不要使代码依赖这些事实，因为它们有可能会改变。IRP 最终完成后，I/O 管理器将释放系统缓冲区所占用的内存。

Direct 方式

如果你在设备对象中指定 DO_DIRECT_IO 方式，I/O 管理器将创建一个 MDL 用来描述包含该用户模式数据缓冲区的锁定内存页。MDL 结构的声明如下：

```
typedef struct _MDL {
    struct _MDL *Next;
    CSHORT Size;
    CSHORT MdlFlags;
    struct _EPROCESS *Process;
    PVOID MappedSystemVa;
    PVOID StartVa;
    ULONG ByteCount;
    ULONG ByteOffset;
} MDL, *PMDL;
```

图 7-3 显示了 MDL 扮演的角色。**StartVa** 成员给出了用户缓冲区的虚拟地址，这个地址仅在拥有数据缓冲区的用户模式进程中有效。**ByteOffset** 是缓冲区起始位置在一个页帧中的偏移值，**ByteCount** 是缓冲区的字节长度。**Pages** 数组没有被正式地声明为 MDL 结构的一部分，在内存中它跟在 MDL 的后面，包含用户模式虚拟地址映射为物理页帧的个数。

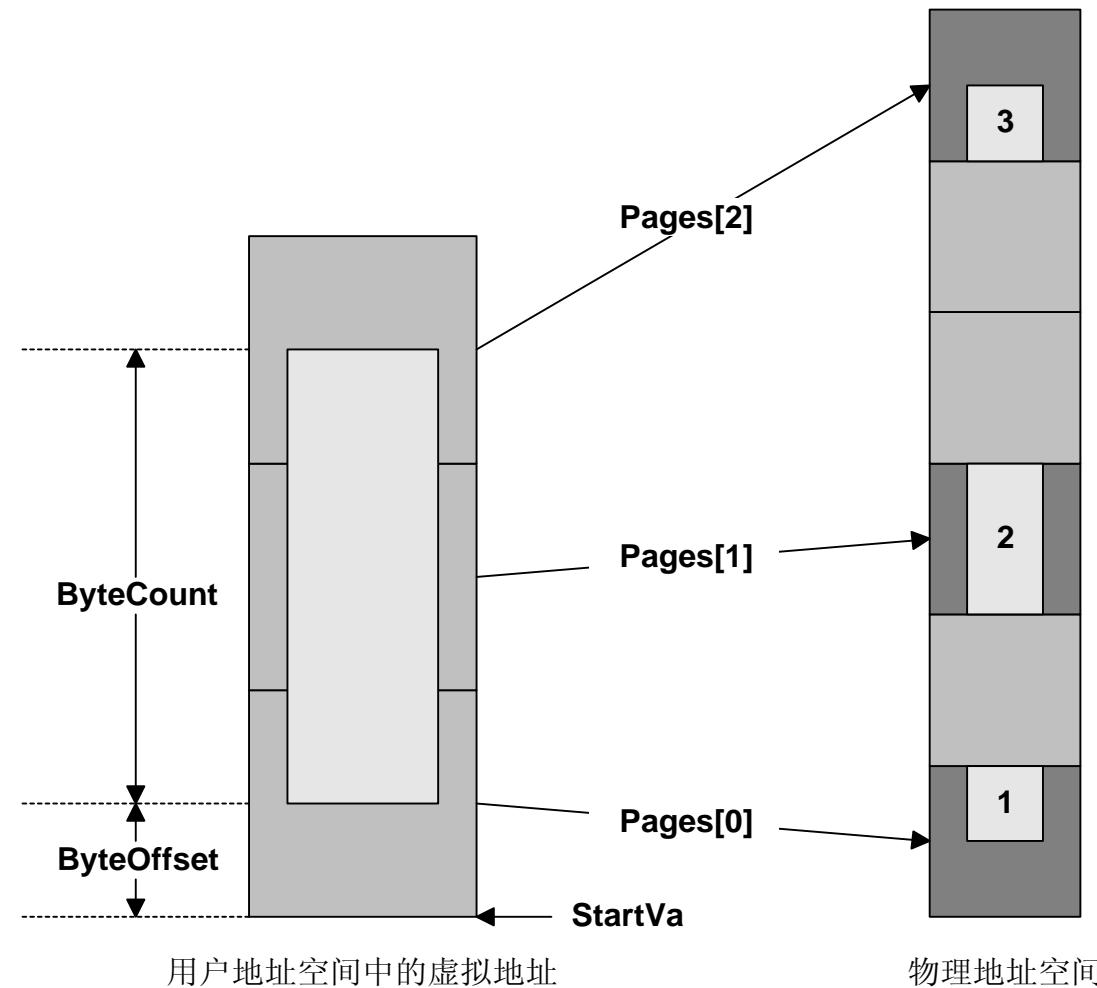


图 7-3. 内存描述符表(MDL)结构

顺便说一下，我们不可以直接访问 MDL 的任何成员。应该使用宏或访问函数，见表 7-2。

表 7-2. 用于访问 MDL 的宏和访问函数

宏或函数	描述
IoAllocateMdl	创建 MDL
IoBuildPartialMdl	创建一个已存在 MDL 的子 MDL
IoFreeMdl	销毁 MDL
MmBuildMdlForNonPagedPool	修改 MDL 以描述内核模式中一个非分页内存区域
MmGetMdlByteCount	取缓冲区字节大小
MmGetMdlByteOffset	取缓冲区在第一个内存页中的偏移
MmGetMdlVirtualAddress	取虚拟地址
MmGetSystemAddressForMdl	创建映射到同一内存位置的内核模式虚拟地址
MmGetSystemAddressForMdlSafe	与 MmGetSystemAddressForMdl 相同，但 Windows 2000 首选
MmInitializeMdl	(再)初始化 MDL 以描述一个给定的虚拟缓冲区
MmPrepareMdlForReuse	再初始化 MDL
MmProbeAndLockPages	地址有效性校验后锁定内存页
MmSizeOfMdl	取为描述一个给定的虚拟缓冲区的 MDL 所占用的内存大小

MmUnlockPages

为该 MDL 解锁内存页

对于 I/O 管理器执行的 *Direct* 方式的读写操作，其过程可以想象为下面代码：

```
KPROCESSOR_MODE mode; // either KernelMode or UserMode  
PMDL mdl = IoAllocateMdl(uva, length, FALSE, TRUE, Irp);  
MmProbeAndLockPages(mdl, mode, reading ? IoWriteAccess : IoReadAccess);  
  
<code to send and await IRP>  
  
MmUnlockPages(mdl);  
ExFreePool(mdl);
```

I/O 管理器首先创建一个描述用户缓冲区的 MDL。**IoAllocateMdl** 的第三个参数(FALSE)指出这是一个主数据缓冲区。第四个参数(TRUE)指出内存管理器应把该内存充入进程配额。最后一个参数(Irp)指定该 MDL 应附着的 IRP。在内部，**IoAllocateMdl** 把 **Irp->MdlAddress** 设置为新创建 MDL 的地址，以后你将用到这个成员，并且 I/O 管理器最后也使用该成员来清除 MDL。

这段代码的关键地方是调用 **MmProbeAndLockPages**(以粗体字显示)。该函数校验那个数据缓冲区是否有效，是否可以按适当模式访问。如果我们向设备写数据，我们必须能读缓冲区。如果我们从设备读数据，我们必须能写缓冲区。另外，该函数锁定了包含数据缓冲区的物理内存页，并在 MDL 的后面填写了页号数组。在效果上，一个锁定的内存页将成为非分页内存池的一部分，直到所有对该页内存加锁的调用者都对其解了锁。

在 *Direct* 方式的读写操作中，对 MDL 你最可能做的事是把它作为参数传递给其它函数。例如，DMA 传输的 **MapTransfer** 步骤需要一个 MDL。另外，在内部，USB 读写操作总使用 MDL。所以你应该把读写操作设置为 DO_DIRECT_IO 方式，并把结果 MDL 传递给 USB 总线驱动程序。

顺便提一下，I/O 管理器确实在 **stack->Parameters** 联合中保存了读写请求的长度，但驱动程序应该直接从 MDL 中获得请求数据的长度。

```
ULONG length = MmGetMdlByteCount(mdl);
```

Neither方式

如果你在设备对象中同时忽略了 DO_DIRECT_IO 和 DO_BUFFERED_IO 标志设置，你将得到默认的 *neither* 方式。对于这种方式，I/O 管理器将简单地把用户模式虚拟地址和字节计数(以粗体显示的代码)交给你，其余的工作由你去做。

```
Irp->UserBuffer = uva;  
PIO_STACK_LOCATION stack = IoGetNextIrpStackLocation(Irp);  
if (reading)  
    stack->Parameters.Read.Length = length;  
else  
    stack->Parameters.Write.Length = length;  
  
<code to send and await IRP>
```

端口与寄存器

图 7-4 显示了 Windows 2000 驱动程序访问硬件设备的几种模式。通常，CPU 的内存地址空间和 I/O 地址空间是分离的。为了访问“内存映射”设备，CPU 用 load 或 store 操作直接对一个虚拟地址进行内存引用，然后 CPU 利用一组页表把虚拟地址转换成物理地址。为了访问“I/O 映射”设备，CPU 必须使用特殊的机制，如使用 x86 处理器上的 IN 和 OUT 指令。

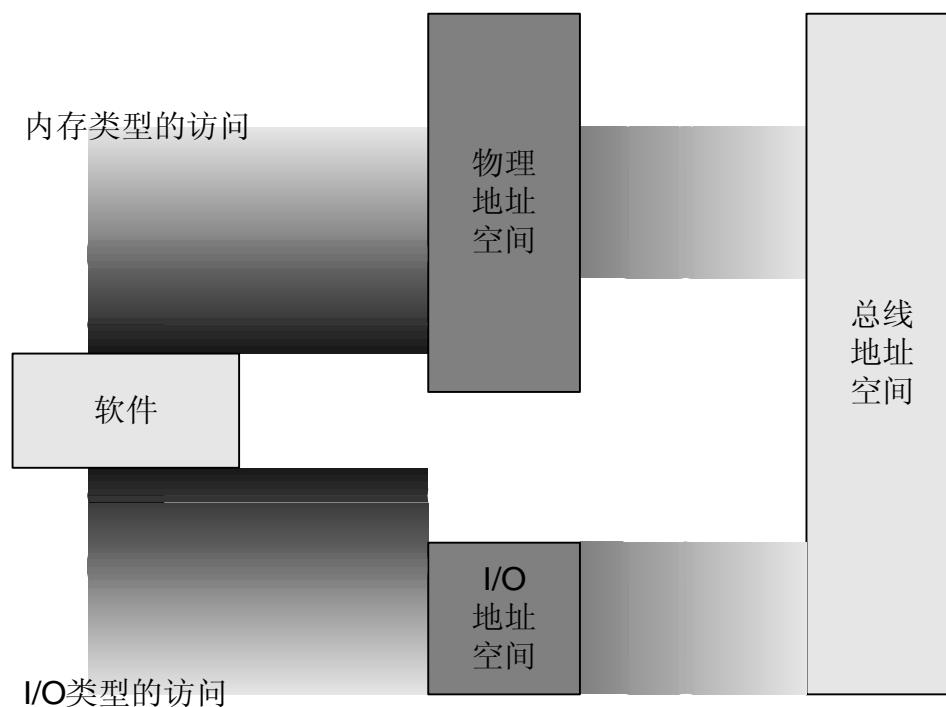


图 7-4. 访问端口与寄存器

设备使用总线专有的方式解码内存和 I/O 地址。在 PCI 总线中，主 PCI 桥把 CPU 物理地址和 I/O 地址映射到总线地址空间，而设备可以直接访问总线地址空间。对于两种地址空间都支持的 CPU，设备配置空间中的标志位决定了主 PCI 桥是把设备寄存器映射成内存地址还是映射成 I/O 地址。

正如我刚提到的，某些 CPU 有分离的内存地址空间和 I/O 地址空间。例如，Intel 架构的 CPU 同时支持这两种地址空间。其它 CPU，例如 Alpha，仅有一个内存地址空间。如果你的设备是 I/O 映射的，PnP 管理器将赋予你端口资源。如果你的设备是内存映射的，它将赋予你内存资源。

为了避免为兼容各种平台而使用大量条件编译代码，Windows NT 的设计者发明了硬件抽象层(HAL)，这个概念我曾在本书的多个地方提到过。HAL 提供了用于访问端口和内存资源的函数，见表 7-3。正如这个表所指出的，你可以从或向一个 PORT/REGISTER 资源 READ/WRITE 一个或一组 UCHAR/USHORT/ULONG 值。有 24 个函数用于设备访问，但 WDM 驱动程序不直接依靠 HAL 做任何事，你可以把这 24 个函数看成是 HAL 的全部公共接口。

表 7-3. 用于访问端口和内存寄存器的 HAL 函数

存取宽度	端口访问函数	内存访问函数
8 位	READ_PORT_UCHAR WRITE_PORT_UCHAR	READ_REGISTER_UCHAR WRITE_REGISTER_UCHAR
16 位	READ_PORT USHORT	READ_REGISTER USHORT

	WRITE_PORT USHORT	WRITE_REGISTER USHORT
32 位	READ_PORT ULONG WRITE_PORT ULONG	READ_REGISTER ULONG WRITE_REGISTER ULONG
8 位字节串	READ_PORT_BUFFER UCHAR WRITE_PORT_BUFFER UCHAR	READ_REGISTER_BUFFER UCHAR WRITE_REGISTER_BUFFER UCHAR
16 位字串	READ_PORT_BUFFER USHORT WRITE_PORT_BUFFER USHORT	READ_REGISTER_BUFFER USHORT WRITE_REGISTER_BUFFER USHORT
32 位双字串	READ_PORT_BUFFER ULONG WRITE_PORT_BUFFER ULONG	READ_REGISTER_BUFFER ULONG WRITE_REGISTER_BUFFER ULONG

很明显，这些访问函数的内部实现与平台高度相关。例如，Intel x86 版本的 **READ_PORT_CHAR** 函数用 **IN** 指令从指定的 I/O 端口读一个字节。而在 Windows 98 中，有些地方甚至把驱动程序对这个函数的 **CALL** 指令直接替换成 **IN** 指令。该函数的 Alpha 版本则执行一个内存提取。**READ_REGISTER_UCHAR** 函数的 Intel x86 版本将执行一个内存提取；而该函数的 Alpha 版本则是一个直接内存引用的宏。该函数的缓冲区版本，**READ_REGISTER_BUFFER_UCHAR**，在 x86 环境中需要做一些额外工作，以确保在操作完成时所有 CPU 的高速缓存都被正确刷新。

使用 **HAL** 的关键原因是你可以不必担心平台的不同，也不用担心 Windows 2000 的多任务和多处理器环境对设备访问的特殊要求。因此，你的工作将十分简单：使用 **PORT** 调用访问端口资源，使用 **REGISTER** 调用访问内存资源。

端口资源

I/O 映射设备把硬件寄存器暴露给某些种类的 CPU(包括 Intel x86)，软件可以使用 I/O 地址空间寻址这些硬件寄存器。而其它种类的 CPU 没有分离的 I/O 地址空间，因此它们使用常规的内存引用来寻址这些寄存器。幸运的是，你完全可以不用了解这些复杂的寻址过程。如果你的设备需要一个端口资源，那么你将在转换后的描述符中找到一个 **CmResourceTypePort** 类型的描述符，你需要保存这个描述符的三个方面的信息。

```

typedef struct _DEVICE_EXTENSION {
    ...
    PUCHAR portbase;
    ULONG nports;
    BOOLEAN mappedport;
    ...
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

PHYSICAL_ADDRESS portbase;      // base address of range
...
for (ULONG i = 0; i < nres; ++i, ++resource)
{
    switch (resource->Type)
    {
        case CmResourceTypePort:
            portbase = resource->u.Port.Start;                                <-1
            pdx->nports = resource->u.Port.Length;
            pdx->mappedport = (resource->Flags & CM_RESOURCE_PORT_IO) == 0;    <-2
    }
}

```

```

break;
...
}

...
if (pdx->mappedport)
{
    pdx->portbase = (PUCHAR) MmMapIoSpace(portbase, pdx->nports, MmNonCached);      <-3
    if (!pdx->portbase)
        return STATUS_NO_MEMORY;
}
else
    pdx->portbase = (PUCHAR) portbase.QuadPart;                                     <-4

```

1. 资源描述符含有一个名为 **u** 的联合，**u** 联合含有每个标准资源类型的子结构。**u.Port** 含有关于端口资源的信息。**u.Port.Start** 是 I/O 端口范围的起始地址，**u.Port.Length** 是在该范围内的端口数量。起始地址是一个 64 位的 PHYSICAL_ADDRESS 值。
2. 端口资源的描述符有一个 **Flags** 成员，如果 CPU 支持分离的 I/O 地址空间，而这个给定端口又属于这个空间，那么这个标志将有 CM_RESOURCE_PORT_IO 设置。
3. 如果没有设置 CM_RESOURCE_PORT_IO 标志，这可能是在一个 Alpha 平台或其它 RISC 平台上，你必须调用 **MmMapIoSpace** 函数来获得能访问该端口的内核模式虚拟地址。访问将真正使用内存引用，但在驱动程序中你仍要调用 HAL 中的 PORT 风格的例程(如 READ_PORT_UCHAR)。
4. 如果设置了 CM_RESOURCE_PORT_IO 标志，这将在一个 x86 平台上，因此你不必映射端口地址。所以，在你的驱动程序中你应该使用 PORT 风格的 HAL 例程访问设备端口。HAL 例程要求一个 PUCHAR 类型的端口地址参数，因此我们需要把基地址强制转换成这个类型。**QuadPart** 引用将使你得到一个与编译平台相适应的 32 位或 64 位指针。

不管端口地址是否需要经过 **MmMapIoSpace** 函数映射，你总应该调用处理 I/O 端口资源的 HAL 例程：**READ_PORT_UCHAR** 或 **WRITE_PORT_UCHAR**，等等。在一个需要映射端口地址的 CPU 上，HAL 将生成内存引用。在不需要映射的 CPU 上，HAL 将生成 I/O 引用；在 x86 处理器上，这意味着使用 IN 和 OUT 指令族。

如果你需要映射你的端口资源，那么你的 **StopDevice** 函数必须执行一个小的清除任务：

```

VOID StopDevice(...)
{
    ...
    if (pdx->portbase && pdx->mappedport)
        MmUnmapIoSpace(pdx->portbase, pdx->nports);
    pdx->portbase = NULL;
    ...
}

```

内存资源

内存映射设备暴露了软件可以使用 **load** 和 **store** 指令访问的寄存器。你从 PnP 管理器得到的转换后的资源值是一个物理地址，你需要保留该物理地址对应的虚拟地址。以后，你将使用处理内存寄存器的 HAL 例程，如 **READ_REGISTER_UCHAR** 和 **WRITE_REGISTER_UCHAR**，等等。你的提取和配置代码应该象下面这样：

```

typedef struct _DEVICE_EXTENSION {
    ...
    PUCHAR membase;
    ULONG nbytes;
    ...
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

```

```

PHYSICAL_ADDRESS membase;      // base address of range
...
for (ULONG i = 0; i < nres; ++i, ++resource)
{
    switch (resource->Type)
    {
        case CmResourceTypeMemory:
            membase = resource->u.Memory.Start;           <--1
            pdx->nbytes = resource->u.Memory.Length;
            break;
        ...
    }
    ...
    pdx->membase = (PUCHAR) MmMapIoSpace(membase, pdx->nbytes, MmNonCached);   <--2
    if (!pdx->membase)
        return STATUS_NO_MEMORY;
}

```

1. 在资源描述符中，**u.Memory** 含有内存资源信息。**u.Memory.Start** 是一个内存范围的起始地址，**u.Memory.Length** 是该范围的字节长度。起始地址是一个 64 位的 PHYSICAL_ADDRESS 值。**u.Port** 和 **u.Memory** 子结构完全相同，这是故意的，并非偶然，如果你需要你可以依靠这个事实。
2. 你必须调用 **MmMapIoSpace** 函数获得一个内核模式虚拟地址，这样内存范围才能被访问。

你的 **StopDevice** 函数可以无条件地解除内存资源映射：

```

VOID StopDevice(...)
{
    ...
    if (pdx->membase)
        MmUnmapIoSpace(pdx->membase, pdx->nbytes);
    pdx->membase = NULL;
    ...
}

```

响应中断

许多设备使用异步方式的中断来通知处理器其 I/O 操作的完成。在这节中，我们将讨论如何为中断处理配置驱动程序，以及如何响应中断。

配置中断

你应该在 `StartDevice` 函数中配置中断资源，使用从 `CmResourceTypeInterrupt` 描述符中提取的参数来调用 `IoConnectInterrupt` 函数。调用 `IoConnectInterrupt` 时，你的驱动程序和设备必须完全准备好，因为中断可能发生在函数返回前，所以通常应该在接近配置过程的结尾处做这个调用。某些设备的硬件特征可以允许驱动程序禁止其硬件中断。如果你的设备有这种特征，你应该在调用 `IoConnectInterrupt` 前禁止设备中断，之后再允许设备中断。一个中断的提取和配置代码应该象下面这样：

```
typedef struct _DEVICE_EXTENSION {
    ...
    PKINTERRUPT InterruptObject;
    ...
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

ULONG vector;           // interrupt vector
KIRQL irql;            // interrupt level
KINTERRUPT_MODE mode;  // latching mode
KAFFINITY affinity;   // processor affinity
BOOLEAN irqshare;      // shared interrupt?
...
for (ULONG i = 0; i < nres; ++i, ++resource)
{
    switch (resource->Type)
    {
        case CmResourceTypeInterrupt:
            irql = (KIRQL) resource->u.Interrupt.Level;
            vector = resource->u.Interrupt.Vector;
            affinity = resource->u.Interrupt.Affinity;
            mode = (resource->Flags == CM_RESOURCE_INTERRUPT_LATCHED) ? Latched : LevelSensitive;
            irqshare = resource->ShareDisposition == CmResourceShareShared;
            break;
        ...
    }
    ...
}

status = IoConnectInterrupt(&pdx->InterruptObject,
                           (PKSERVICE_ROUTINE) OnInterrupt,
                           (PVOID) pdx,
                           NULL,
                           vector,
                           irql,
                           irql,
                           mode,
                           irqshare,
                           affinity,
                           FALSE);
```

1. **Level** 参数指定这个中断的 IRQL。

2. **Vector** 参数指定这个中断的硬件中断矢量。因为我们仅充当 PnP 管理器和 IoConnectInterrupt 之间的连接桥梁，所以不用关心这个数值。这个数的含义只有 HAL 能了解。
3. **Affinity** 是一个位掩码，它指出该中断应由哪个 CPU 来处理。
4. 我们需要告诉 IoConnectInterrupt 我们的中断是边缘触发方式还是水平触发方式。如果资源标志 **Flags** 为 CM_RESOURCE_INTERRUPT_LATCHED，则我们的中断是边缘触发方式，否则是水平触发方式。
5. 这个语句查明中断是否是共享的。

在这段代码的最后是 IoConnectInterrupt 调用，在这里我们使用了从中断资源描述符中获得的各个值。第一个参数(**&pdx->InterruptObject**)指出在哪里存放连接操作的结果，即一个指向内核中断对象的指针，该对象描述你的中断。第二个参数(**OnInterrupt**)指向你的中断服务例程(ISR)；稍后我将讨论 ISR。第三个参数(**pdx**)是每次当你的设备中断时作为参数传递给 ISR 的一个上下文值。在后面的“选择适当的上下文参数”段中，我将详细讨论这个上下文参数。

第五和第六个参数(**vector** 和 **irql**)为被连接的中断指定中断矢量号和中断请求级。第八个参数(**mode**)应该为 **Latched** 或 **LevelSensitive**，它指出中断是边缘触发方式还是水平触发方式。第九个参数如果为 **TRUE**，则表明这个中断可以被其它设备共享，反之为 **FALSE**。第十个参数(**affinity**)是该中断的处理器亲合掩码。第十一个和最后一个参数指出当中断发生时是否需要操作系统保存浮点上下文。因为在 x86 平台上的 ISR 中不允许执行浮点运算，所以一个可移植的驱动程序应该总把该标志设为 **FALSE**。

有两个 IoConnectInterrupt 的参数我还没有描述。当设备使用多个中断时，这两个参数就变得特别重要。在这种情况下，你需要为这些中断创建自旋锁，并调用 **KeInitializeSpinLock** 函数初始化这个自旋锁。在连接中断前，你还需要计算出这些中断中的最大 IRQL。在每次调用 IoConnectInterrupt 时，你应该在第四个参数中(在例子中为 **NULL**)指定这个自旋锁的地址，以及在第七个参数(在例子中为 **irql**)中指定最大 IRQL。第七个参数指定的 IRQL 用于同步多个中断，所以你应该使用这些中断中最大的 IRQL，这样你一次就只需要响应一个中断。

然而，如果设备只使用一个中断，你就不必指定自旋锁(因为 I/O 管理器自动为你分配一个)，并且中断的同步级别将与中断的 IRQL 相同。

中断要连接到 CPU 时需要满足两个条件，一、**affinity** 中指定的 CPU，二、当前使用的。换句话说，决定由哪个 CPU 接受中断是在 IoConnectInterrupt 中作出的。

处理中断

当设备生成中断时，HAL 将基于你指定的 CPU 亲合掩码选择一个 CPU 来服务这个中断。然后它把选定 CPU 的 IRQL 提升到合适的同步级别并请求一个与中断对象关联的自旋锁。最后调用 ISR，ISR 的框架结构见下面代码：

```
BOOLEAN OnInterrupt(PKINTERRUPT InterruptObject, PVOID Context)
{
    if (<device not interrupting>)
        return FALSE;
    <handle interrupt>
    return TRUE;
}
```

Windows NT 的中断处理机制假定多个设备可以共享一个硬件中断。因此，ISR 的首要工作就是找出哪一个设备发生了中断。如果没有，你应该立刻返回 **FALSE**，以便 HAL 能把中断送往其它设备驱动程序。如果有，你应该先在设备级清除该中断然后返回 **TRUE**。HAL 是否调用其它驱动程序的 ISR，主要取决于设备中断是边缘触发方式还是水平触发方式，以及具体的平台。

ISR 的主要工作就是清除中断，执行了这个主要任务之后就可以向 HAL 返回 **TRUE**，表明你已经服务了该设备中断。

ISR中的编程限制

ISR 执行在高于 DISPATCH_LEVEL 的 IRQL 上。因此 **ISR** 中使用的所有代码和数据必须存在于非分页内存中，此外，**ISR** 能调用的内核模式函数十分有限。

因为 **ISR** 执行在提升的 IRQL 上，所以它冻结了其 CPU 上所有低于或等于该 IRQL 的其它活动。为了提高系统性能，你的 **ISR** 应该尽可能快地执行。基本上，只做服务硬件所需的最小量的工作，然后立即返回。如果有额外的工作需要做(例如完成一个 **IRP**)，应该交给 **DPC** 来完成。

尽管 **ISR** 的代码应该尽可能地短，但你不要把这个想法运用到极端。例如，如果某设备的中断是表明它已经为接收下一个字节而准备就绪，你就应该在 **ISR** 中直接发出这个字节。为了传输一个字节而调用 **DPC** 是愚蠢的。

不要发疯似地在 **ISR** 中计算 pi 到上千位。你应该在 **ISR** 和 **DPC** 间找到正确的负载平衡。

选择一个合适的上下文参数

在 **IoConnectInterrupt** 调用中，第三个参数是一个任意的上下文值，这个值最后将成为 **ISR** 的第二个参数。你应该选择好这个参数以使 **ISR** 能尽可能快地执行；设备对象的地址或设备扩展的地址将是一个好的选择。设备扩展中保存了许多有用的数据，如设备的端口基址，你可能使用这个端口地址来测试设备是否真的发出了中断。例如，假定你的设备硬件使用 I/O 映射模式，其基址是设备的状态端口，而状态值的低位指出设备当前是否发出中断。如果你采纳我的建议，你的 **ISR** 的前几行应该象这样：

```
BOOLEAN OnInterrupt(PKINTERRUPT InterruptObject, PDEVICE_EXTENSION pdx)
{
    UCHAR devstatus = READ_PORT_UCHAR(pdx->portbase);
    if (!(devstatus & 1))
        return FALSE;
    <etc.>
}
```

该函数的全优化代码仅需要几条指令来读状态端口以及测试低位。

如果你把设备扩展作为上下文参数，在调用 **IoConnectInterrupt** 函数时应做一个强制类型转换：

```
IoConnectInterrupt(..., (PKSERVICE_ROUTINE) OnInterrupt, ...);
```

如果你忽略了强制类型转换，编译器将生成一个特别含糊的错误消息，因为 **OnInterrupt** 例程(一个 PDEVICE_EXTENSION)的第二个参数不匹配 **IoConnectInterrupt** 函数的函数指针参数，它需要一个 PVOID 类型。

ISR的同步操作

作为一个通用规则，**ISR** 可以与驱动程序的其它部分共享数据和硬件资源。任何时候你听到“共享”这个词，就应该立即想到同步问题。例如，一个标准的 **UART** 设备有一个数据端口，驱动程序就用这个端口来读写数据。你可能希望串口驱动程序的 **ISR** 能随时访问这个端口。改变波特率必须设置一个控制标志，因此应该先在这个数据端口上执行两个单字节的写操作，然后清除控制标志。如果 **UART** 在改变波特率的途中被中断，你可能看到两种意外情况，或者是中断传输的数据字节被放到波特率分配寄存器中，或者是应该传递给分配寄存器的控制数据被当成普通数据传输。

系统在一个相对高的 IRQL(DIRQL)上用一个自旋锁来保护 **ISR** 的执行。为了简化获取该自旋锁以及提升 IRQL 到与中断相同的级，系统提供了下面服务函数：

```
BOOLEAN result = KeSynchronizeExecution(InterruptObject, SynchRoutine, Context);
```

InterruptObject(PKINTERRUPT)是一个指向中断对象的指针，这个中断对象描述了我们要同步的中断。

SynchRoutine(PKSYNCHRONIZE_ROUTINE)是存在于驱动程序中的回调函数的地址。**Context(PVOID)**是作

为参数传递到 **SynchRoutine** 的任意上下文参数。我们使用通用术语“同步关键段例程(synch critical section routine)”来描述 **KeSynchronizeExecution** 应用的子例程。同步关键段例程有如下原型：

```
BOOLEAN SynchRoutine(PVOID Context);
```

即该例程接受一个参数并返回一个布尔类型的结果。当该例程获得控制时，当前 CPU 将运行在 **IoConnectInterrupt** 调用中指定的同步 IRQL 上，并且拥有与中断关联的自旋锁。因此，设备中断将被临时阻塞，**SynchRoutine** 就可以自由地访问与 ISR 共享的数据和硬件资源。

KeSynchronizeExecution 将返回 **SynchRoutine** 返回的任何值。由于布尔类型声明为无符号字符类型，因此你只能有一个字节用来从 **SynchRoutine** 返回信息。

DPC

完整的中断服务通常需要执行这样一些操作，这些操作不适合在 ISR 中执行，或者考虑到执行在提升 IRQL 上的 ISR 中会对系统性能造成影响。为了解决这个问题，Windows NT 的设计者提供了推迟过程调用(DPC)机制。DPC 是一个通用机制，但通常都用在中断处理中。在最普通的情况下，ISR 决定当前请求的完成并请求一个 DPC。之后，内核在 DISPATCH_LEVEL 级上调用这个 DPC 例程。因此 DPC 中的代码要比 ISR 中的代码有更少的限制。特别是，DPC 例程可以调用象 **IoCompleteRequest** 或 **IoStartNextPacket** 这样的例程，在一个 I/O 操作的结尾处调用这些例程在逻辑上是必要的。

每个设备对象中都含有一个 DPC 对象。即，DEVICE_OBJECT 中有一个内置的 DPC 对象 **Dpc**。创建设备对象后你需要立刻初始化这个内置 DPC 对象。

```
NTSTATUS AddDevice(...)
{
    PDEVICE_OBJECT fdo;
    IoCreateDevice(..., &fdo);
    IoInitializeDpcRequest(fdo, DpcForIsr);
    ...
}
```

IoInitializeDpcRequest 是 WDM.H 中的一个宏，它初始化设备对象中的内置 DPC 对象。第二个参数是 DPC 例程的地址，很快，我们将讨论这个例程。

如果你在适当的地方初始化一个 DPC 对象，你就可以在 ISR 中用下面函数(宏)请求一个 DPC：

```
BOOLEAN OnInterrupt(...)
{
    ...
    IoRequestDpc(pdx->DeviceObject, NULL, (PVOID) pdx);
    ...
}
```

这个 **IoRequestDpc** 函数将把设备对象中的 DPC 对象放到一个系统范围的队列，见图 7-5。

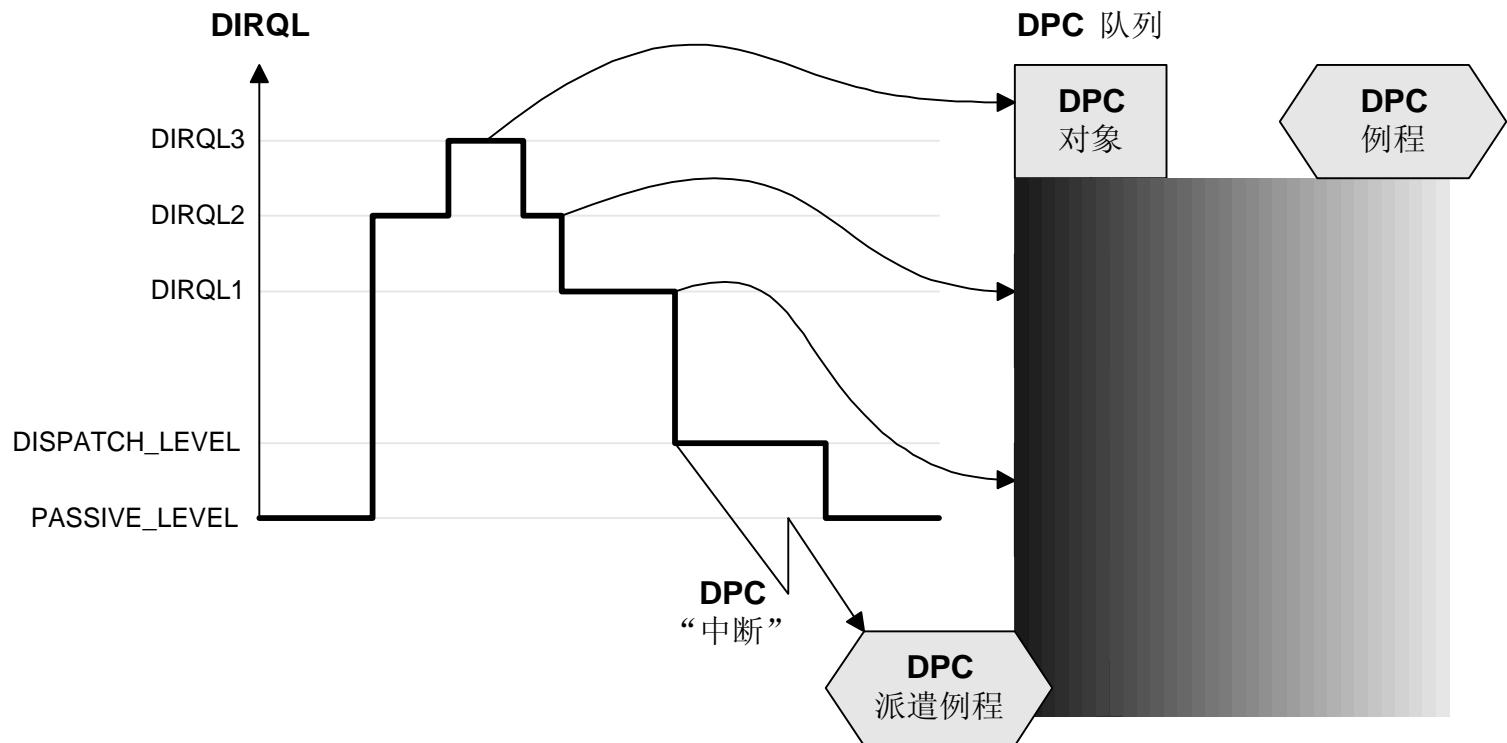


图 7-5. 处理 DPC 请求

两个为 `NULL` 的参数是上下文值，在这里没有什么实际用处。之后，当 `DISPATCH_LEVEL` 级上没有其它活动时，内核从队列上删除你的 DPC 对象，然后调用你的 DPC 例程，DPC 例程有如下原型：

```
VOID DpcForIsr(PKDPC Dpc, PDEVICE_OBJECT fdo, PIRP junk, PVOID Context)
{ }
```

DPC 例程所做的工作大部分要取决于设备的工作方式。一个可能的任务是完成当前的 IRP 以及发出队列中的下一个 IRP。如果你使用标准模型来排队 IRP，则代码如下：

```
VOID DpcForIsr(...)
{
    PIRP Irp = fdo->CurrentIrp;
    IoStartNextPacket(fdo, TRUE);
    IoCompleteRequest(Irp, <boost value>);
}
```

IoStartNextPacket 的 `TRUE` 参数指出下一个 IRP 可以被取消，即 **IoStartPacket** 的原始调用中指定了取消例程，这将使 **IoStartNextPacket** 函数在访问设备对象和 **CurrentIrp** 时需申请全局取消自旋锁的保护。

在这个代码片段中，我们依赖这样一个事实，I/O 管理器设置设备对象的 `CurrentIrp` 域为指向其发往 **StartIo** 例程的 IRP 的指针。我们要完成的 IRP 就是当我们启动 DPC 例程时 `CurrentIrp` 指向的 IRP。通常我们在调用 **IoCompleteRequest** 前先调用 **IoStartNextPacket** 函数，这样在我们开始完成有可能是长耗时的 IRP 处理前，可以使设备忙于处理新请求。

如果你使用上一章讲述的 `DEVQUEUE` 对象来排队 IRP，则代码将类似标准模型：

```
VOID DpcForIsr(...)
{
    PDEVICE_EXTENSION pdx = ...;
    PIRP Irp = GetCurrentIrp(&pdx->dqRead);
    StartNextPacket(&pdx->dqRead, fdo);
    IoCompleteRequest(Irp, <boost value>);
}
```

DPC调度

到现在我必须揭示关于 DPC 的两个重要细节和一个相对次要的细节。第一个重要的细节暗含在这样的事实内，你有一个 DPC 对象，你用 `IoRequestDpc` 函数把它放到队列中。如果在 DPC 例程真正运行前，你的设备又生成一个中断，并且如果你的 ISR 为此又请求了一个 DPC，那么内核将简单地忽略第二个 DPC 请求。换句话说，不管 ISR 连续请求了多少个 DPC，你的 DPC 对象仅在队列上一次，而且内核也仅调用你的回调例程一次。在这个调用中，你的 DPC 例程需要完成所有发生在上一个 DPC 之后的所有中断的相关工作。

一旦 DPC 派遣器提取(出队)了你的 DPC 对象，某些代码可能再次把它排入队列，甚至在这个 DPC 例程正执行时。如果该对象在同一 CPU 上被排队两次，那么不会产生任何问题。关于 DPC 处理的第二个重要细节是必须参考 CPU 的中断亲合性。通常，内核把 DPC 对象排队到与请求该 DPC 对象的同一个 CPU 上来处理，例如，刚服务了那个中断并调用了 `IoRequestDpc` 例程的 CPU。一旦 DPC 派遣器取出了该 DPC 对象并在某个 CPU 上调用了它的回调例程，那么在理论上你的设备可以中断另一个 CPU，而这个 CPU 最终也要请求那个 DPC，结果两个不同的 CPU 将同时执行一个 DPC。DPC 例程的并发执行是否会产生问题，很明显，主要取决于你代码的具体处理细节。

对于 DPC 并发地运行在多个 CPU 上的情况，有几种方法可以避免这个潜在的问题。第一种方法并不是最好的，即调用 `KeSetTargetProcessorDpc` 函数把 DPC 指定为特定的 CPU 才能运行。同样，理论上你可以在第一次把中断连接到 CPU 时限定其 CPU 亲合性；如果你从不在 ISR 之外的地方排队 DPC，那么该 DPC 也不会在任何其它的 CPU 上执行。然而，你指定 DPC 或中断的 CPU 亲合性的真正原因是提高性能，即，使 DPC 或 ISR 例程访问的数据和代码能存在于高速缓存中。

你还可以使用自旋锁或其它同步原语来防止多个 DPC 例程实例间的相互干扰。但在这里使用自旋锁要小心：通常你应该用 ISR 调整假定的 DPC 的多个实例，而使用普通的自旋锁，ISR 的运行 IRQL 又太高，互锁链表(用 `ExInterlockedInsertHeadList` 族函数来维护)可以帮助你，因为(只要你不获取用于保护该链表的自旋锁)你可以在任何 IRQL 上使用这种链表。互锁形式的位 OR 和 AND 操作也可以帮助你，通过控制要完成的 DPC 例程的屏蔽位(例如用屏蔽位指定最近的中断情况)；必要时可以用 `InterlockedCompareExchange` 函数来辅助这些函数。

最简单的办法是确保设备不在你请求 DPC 和 DPC 例程完成其工作之间发生中断。(“咳，硬件设计者，不要产生不必要的中断！”)

第三个 DPC 细节，就是我认为比前两个次要的细节，关系到 DPC 的“重要性”属性。调用 `KeSetImportanceDpc` 函数，你可以为 DPC 指定三种重要性级别：

- **MediumImportance** 是默认值，它指出该 DPC 应该排在所有当前排队的 DPC 之后。如果该 DPC 被排到其它处理器的队列中，则那个处理器不必立即中断去执行该 DPC。如果它被排入当前处理器的队列，则只要可能，内核将立即请求一个 DPC 中断并开始服务该 DPC。
- **HighImportance** 将使 DPC 被排到队列头部。如果有两个高度重要的 DPC 几乎同时请求，则后排入队列的 DPC 先得到服务。
- **LowImportance** 使 DPC 排到队列末尾。此外，内核不必立即中断相关处理器去执行这种 DPC。

DPC 重要级的效果是产生影响，没有必要去控制 DPC 发生的速度。即使一个有低重要性的 DPC 也会触发一个处理器的 DPC 中断，只要该处理器到达排队 DPC 的下限。如果设备能在 DPC 例程运行前再次产生中断，则可以降低该 DPC 的重要性，但这会增加你执行多工作项(work item)的可能性。如果你的 DPC 指定了与请求该 DPC 之外的 CPU 的亲合，则为 DPC 选择高重要性将增加 ISR 与 DPC 例程同时运行的可能性。这些可能性都不是确定的，改变或不改变 DPC 的重要性并不能消除这些可能性。

定制DPC对象

除了设备对象内置的 DPC 对象 `Dpc` 之外，你还可以创建其它 DPC 对象。在你的设备扩展或其它非分页的区域中为 `KDPC` 对象保留存储，然后初始化该对象：

```
typedef struct _DEVICE_EXTENSION {  
    ...  
}
```

```
KDPC CustomDpc;  
...  
};  
  
KeInitializeDpc(&pdx->CustomDpc, (PKDEFERRED_ROUTINE) DpcRoutine, fdo);
```

在 **KeInitializeDpc** 调用中，第二个参数是存在于非分页内存中的 DPC 例程的地址，第三个参数是一个任意上下文参数，将作为 DPC 例程的第二个参数发往 DPC 例程。

为了请求定制 DPC 例程的推迟调用，调用 **KeInsertQueueDpc** 函数：

```
BOOLEAN inserted = KeInsertQueueDpc(&pdx->CustomDpc, arg1, arg2);
```

这里，**arg1** 和 **arg2** 是任意上下文指针，将被传递到定制 DPC 例程。如果 DPC 对象已进入某处理器的队列则该函数返回 FALSE，否则返回 TRUE。

同样，调用 **KeRemoveQueueDpc** 函数可以从处理器队列中删除一个 DPC 对象。

一个中断驱动设备的例子

我写了一个例子驱动程序，PCI42，来演示如何写典型的中断驱动(非 DMA)设备上的各种驱动程序例程。处理这种设备的方法通常称为 PIO(programmed I/O)方法，因为每传输一个单位的数据都需要程序来干预。

PCI42 是为 S5933 PCI 芯片组写的一个哑驱动程序。S5933 就象 PCI 总线与其上设备之间的媒介。S5933 非常灵活。实际上，你可以编程其上的非易失 RAM 以便为你的设备初始化 PCI 配置空间。但 PCI42 使用 S5933 出厂时的默认状态。

为了简化问题，WDM 驱动程序与 S5933 上的设备之间的通信或者采用 DMA 方式，或者通过一组收发数据的信箱寄存器。PCI42 将使用信箱寄存器，以一次一个字节的方式传输数据。

S5933 的开发包(S5933DK1)包括两个开发板和一个 ISA 接口卡，该卡经过一个带状电缆连接到 S5933 开发板。ISA 卡允许你从另一侧直接访问 S5933，以便提供额外的软件仿真功能。PCI42 例子的一个组成部分就是那个 ISA 卡的驱动程序(S5933DK1.SYS)，它输出一个供测试程序使用的接口。

硬件工程师可能会嘲笑 PCI42 管理设备的简单方式。但使用这个简单例子你可以看到 I/O 处理操作的每一步。

初始化PCI42

PCI42 中的 **StartDevice** 函数将初始化一个端口资源和一个中断资源。端口资源描述了 I/O 空间中的一组 32 位寄存器(共 16 个)，中断资源描述了主机赋予设备的 INTA#中断。在 **StartDevice** 的最后，我们有下面设备专用代码：

```
NTSTATUS StartDevice(...)  
{  
    ...  
    ResetDevice(pdx);  
    status = IoConnectInterrupt(...);  
    KeSynchronizeExecution(pdx->InterruptObject, (PKSYNCHRONIZE_ROUTINE) SetupDevice, pdx);  
    return STATUS_SUCCESS;  
}
```

我们调用了一个辅助例程(**ResetDevice**)来重置硬件。**ResetDevice** 的一个任务是尽可能地阻止设备生成任何中断。然后我们调用 **IoConnectInterrupt** 函数把设备中断连接到我们的 ISR。在 **IoConnectInterrupt** 返回前，设备可能生成中断，所以驱动程序和硬件必须提前做好准备。中断连接后，我们调用另一个名为 **SetupDevice** 的

辅助函数来编程设备，使其正常工作。因为这一步可能使用与 ISR 相同的硬件寄存器，所以我们必须做好与 ISR 的同步，我们不希望存在任何向设备发送不一致指令的可能性。**SetupDevice** 是 **StartDevice** 函数中的最后一步，这与我在第二章“WDM 驱动程序的基本结构”中讲的正好相反，原因是 **PCI42** 没有寄存任何设备接口，因此也不必在这里有允许接口的代码。

ResetDevice 是高度设备相关的例程，如下面代码：

```
VOID ResetDevice(PDEVICE_EXTENSION pdx)
{
    PAGED_CODE();

    WRITE_PORT ULONG((PULONG) (pdx->portbase + MCSR), MCSR_RESET);

    LARGE_INTEGER timeout;
    timeout.QuadPart = -10 * 10000; // i.e., 10 milliseconds

    KeDelayExecutionThread(KernelMode, FALSE, &timeout);
    WRITE_PORT ULONG((PULONG) (pdx->portbase + MCSR), 0);

    WRITE_PORT ULONG((PULONG) (pdx->portbase + INTCSR), INTCSR_INTERRUPT_MASK);
}
```

1. **S5933** 有一个主控/状态寄存器(MCSR)，它控制着总线主控的 DMA 传输和其它动作。其中有 4 个位能重置设备的不同特征。我定义了 **MCSR_RESET** 常量来作为这四个标志位的掩码。所有这些常量都可以在 **PCI42** 工程的 **S5933.H** 文件中找到。
2. 前三个重置标志属于 **S5933** 的内部特征，如果重置会立即产生效果。第四个标志设置为 1，就是要求 **S5933** 上的功能设备进入重置状态。如果想取消重置状态，你必须明确地把这个标志重置为 0。通常，需要给硬件一点时间来认识重置信号。**KeDelayExecutionThread** 可以使线程进入睡眠状态 10 毫秒。根据硬件的实际需求你可以增加或减少这个常量值，但要记住这个值不要小于系统时钟的单位值。因为我们将要阻塞自己的线程，所以我们必须运行在 **PASSIVE_LEVEL** 级上的一个非任意线程上下文中。当前的情况是适合的，因为我们的最终调用者是 **PnP** 管理器，是它向我们发送了 **IRP_MN_START_DEVICE** 请求，并希望我们阻塞所在的系统线程。
3. 重置设备的最后一步是清除任何未决的中断。**S5933** 的中断控制/状态寄存器(INTCSR)中有六个中断标志。向这六个位置写入 1 将清除所有未决的中断。(如果我们写回的位掩码中的中断标志位置上有为 0 的位，则对应的中断状态不受影响。这种标志位被称为读/写清除或 R/WC 标志) INTCSR 中的其它位用于使能各种中断。在这些地方写入 0 位将禁止相应的中断。

我们的 **SetupDevice** 函数十分简单：

```
VOID SetupDevice(PDEVICE_EXTENSION pdx)
{
    WRITE_PORT ULONG((PULONG) (pdx->portbase + INTCSR),
                    INTCSR_IMBI_ENABLE
                    | (INTCSR_MB1 << INTCSR_IMBI_REG_SELECT_SHIFT)
                    | (INTCSR_BYTE0 << INTCSR_IMBI_BYTE_SELECT_SHIFT)
    );
}
```

该函数重编程了 INTCSR，使其内部的信箱寄存器 1 的字节 0 在发生改变时能产生中断。我们还指定了该芯片的其它中断条件，包括清空一个对外信箱寄存器的特定字节，一个读 DMA 传输的完成操作和一个写 DMA 传输的完成操作。

启动一个读操作

PCI42 的 **StartIo** 例程使用下面编程模式：

```

VOID StartIo(IN PDEVICE_OBJECT fdo, IN PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
    if (!NT_SUCCESS(status))
    {
        CompleteRequest(Irp, status, 0);
        return;
    }

    if (!stack->Parameters.Read.Length)
    {
        StartNextPacket(&pdx->dqReadWrite, fdo);
        CompleteRequest(Irp, STATUS_SUCCESS, 0);
        return;
    }

    pdx->buffer = (PUCHAR) Irp->AssociatedIrp.SystemBuffer;
    pdx->nbytes = stack->Parameters.Read.Length;
    pdx->numxfer = 0;

    KeSynchronizeExecution(pdx->InterruptObject, (PKSYNCHRONIZE_ROUTINE) TransferFirst, pdx);
}

```

- 在这里，我们把描述输入操作过程的参数保存到设备扩展中。PCI42 使用 DO_BUFFERED_IO 方式，该方式不算典型但可以简化例子驱动程序。
- 由于我们的中断已经连接完，因此设备可以随时发出中断。**ISR** 将在中断发生时传输数据字节，但我们希望 **ISR** 不必为具体使用哪个数据缓冲区或要读的字节数所烦恼。为了抑制 **ISR** 的热心，我们把一个标志放到设备扩展中，该标志名为 **activerequest**，平常为 FALSE。现在正是把该标志设为 TRUE 的时候。通常，当处理共享资源时，我们设置该标志的代码需要与 **ISR** 中测试该标志的代码同步，因此我们需要调用 **SynchCriticalSection** 例程。这种情况也发生在在一个数据字节已准备好时，这样第一个中断将不再发生。**TransferFirst** 是一个辅助例程，它检测这种可能性并读取第一个字节。插入设备有办法知道我们清空了信箱寄存器，因此它会在预期的时间发送下一个字节。下面就是 **TransferFirst** 例程：

```

VOID TransferFirst(PDEVICE_EXTENSION pdx)
{
    pdx->activerequest = TRUE;
    ULONG mbef = READ_PORT ULONG((PULONG) (pdx->portbase + MBEF));
    if (!(mbef & MBEF_IN1_0))
        return;

    *pdx->buffer = READ_PORT UCHAR(pdx->portbase + IMB1);
    ++pdx->buffer;
    ++pdx->numxfer;
    if (--pdx->nbytes == 0)
    {
        pdx->activerequest = FALSE;
        IoRequestDpc(pdx->DeviceObject, NULL, NULL);
    }
}

```

S5933 有一个信箱空/满寄存器(**MBEF**)，其中的位指出每个信箱寄存器的每个字节的当前状态。这里，我们检测用于输入(内部信箱寄存器 1，字节地址 0)的寄存器字节是否未被读过。如果是，我们就读出它。这也许会达到该次传输要求量。我们事先准备好的处理例程(**DpcForIsr**)知道该怎么做，所以如果第一个字节满足了该请求，

我们就请求一个 DPC。(想一想，因为我们作为一个 `SynchCriticalSection` 例程被调用，所以我们正执行在 DIRQL 级上，并在一个中断自旋锁的保护下，因此我们不能立刻完成该 IRP)

处理中断

在 PCI42 的正常操作中，S5933 在信箱 1 到达一个新数据字节时发生中断。然后下面的 ISR 获得控制：

```

{
    pdx->activerequest = FALSE;
    IoRequestDpc(pdx->DeviceObject, NULL, NULL);
}

return TRUE;
}

```

1. 第一个任务是检查设备现在是否要中断。读 S5933 的 INTCSR 并测试其中一个位(INTCSR_INTERRUPT_PENDING)。如果该位被清除，我们立即返回。在这里，我选择设备扩展指针作为该例程(当调用 IoConnectInterrupt 时)的上下文参数，原因很明显：我们需要立即访问这个结构以获得端口基地址。
2. 如果我们使用 DEVQUEUE 形式的队列，我们依靠队列对象来跟踪当前的 IRP。这个中断也许是不需要的中断，因为我们现在没有服务任何 IRP。在这种情况下，我们仍需要清除中断，但不做其它事情。
3. 这时也可能发生 PnP 或电源管理事件，这将使派遣例程拒绝任何新的 IRP。DEVQUEUE 的 **AreRequestsBeingAborted** 函数能告诉我们这个事实，所以我们可以立即放弃当前的请求。对于以一个字节一个字节处理的设备，放弃一个活动的请求是合理的。同样，检测一个需要长时间完成的 IRP 是否被取消也是一个好想法。如果设备仅在一个长的数据传输后才发生中断，你也可以不在 ISR 中做这个测试。
4. 现在我们进入一个循环，该循环在设备当前的所有中断都被清除后结束。在循环的结尾，我们重读了 INTCSR 以确定是否还有中断产生。如果有，我们将重复这个循环。不应该在这里过多地占用 CPU 时间，我们应该避免重叠中断，因为中断服务自身是相对昂贵的。
5. 如果 S5933 发生了中断，我们就从信箱寄存器中读入一个字节到当前 IRP 的 I/O 缓冲区中。如果你要在读操作后立即查看一下 MBEF 寄存器，你应该查看内部信箱寄存器 1 地址 0 对应的位，这个位将被对它的读操作清除。注意，我们不必通过测试 MBEF 来确定数据字节是否真正被改变，因为我们把设备编程为仅在该字节被改变时才发生中断。
6. 把 INTCSR 写入以前的内容将产生清除六个 R/WC 中断位的效果，同时不改变那些只读位，并保留了所有读写控制位的原始设置。
7. 这里，我们读出 INTCSR 以确定是否有另外的中断发生。如果有，我们将重复这个循环。
8. 当我们执行过前面的代码后，如果完成当前 IRP 需要一个 DPC，我们就设置 **dpc** 变量为 TRUE。

PCI42 的 DPC 例程如下：

```

VOID DpcForIsr(PKDPC Dpc, PDEVICE_OBJECT fdo, PIRP junk, PVOID Context)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    NTSTATUS status = STATUS_SUCCESS;
    PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
    ULONG info = pdx->numxfer;
    StartNextPacket(&pdx->dqReadWrite, fdo);
    CompleteRequest(Irp, status, info);
}

```

测试PCI42

如果你要测试 PCI42 的操作，需要做几件事情。第一，安装 S5933DK1 开发板卡，包括 ISA 卡。使用添加硬件向导把 S5933DK1.SYS 驱动程序和 PCI42.SYS 驱动程序装入系统。(我发现 Windows 98 开始把这个开发板识别为一个不工作的声卡，我不得不用设备管理器先删除这个声卡，然后再安装 PCI42 驱动程序。Windows 2000 正常地识别了这个开发板)

然后，同时运行 ADDONSIM 和 TEST 程序，它们在随书光盘的 PCI42 目录中。ADDONSIM 通过 ISA 接口向信箱寄存器写入一个数据字节。TEST 从 PCI42 中读取这个数据字节。

书中的例子代码与真正代码的不同之处

这里的 DpcForIsr、TransferFirst、OnInterrupt 例程与随书光盘中的代码在细节上有些不同：

1. 在实际的例子中，**activerequest** 变量的名字为 **busy**，但含义相同。

2. 实际例子中的 **DpcForIsr** 例程仅简单地完成 IRP，并不设置 IRP 中的 **IoStatus**。**TransferFirst** 和 **OnInterrupt** 例程将在请求 DPC 前填充这些值。这种安排的目的是使 ISR 能在遇到错误时能更容易地提前结束该 IRP。
3. 实际例子中的 **OnInterrupt** 例程有更完善的意外情况检测。在标记 2 和标记 3 之间的代码实际应该是这样：

```
PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
NTSTATUS status;
if (pdx->busy)
{
    if (Irp->Cancel)
        status = STATUS_CANCELLED;
    else
        status = AreRequestsBeingAborted(&pdx->dqReadWrite);
    if (!NT_SUCCESS(status))
        dpc = TRUE, pdx->nbytes = 0;
}
```

这个逻辑在 IRP 已被取消，或放弃所有 IRP 的 PnP/电源管理事件发生时，使 ISR 能停止该 IRP。

直接内存存取(DMA)

Windows 2000 的 DMA 传输基于如图 7-6 这样的抽象模型。在这个模型中，计算机被认为有这样一组“映射寄存器”，它们在 CPU 的物理地址和总线地址之间做相互转换。每个映射寄存器保存着一个物理页帧的地址。硬件使用一个“逻辑的”或总线专有的地址来读写内存。对软件来说映射寄存器扮演了与页表项相同的角色。

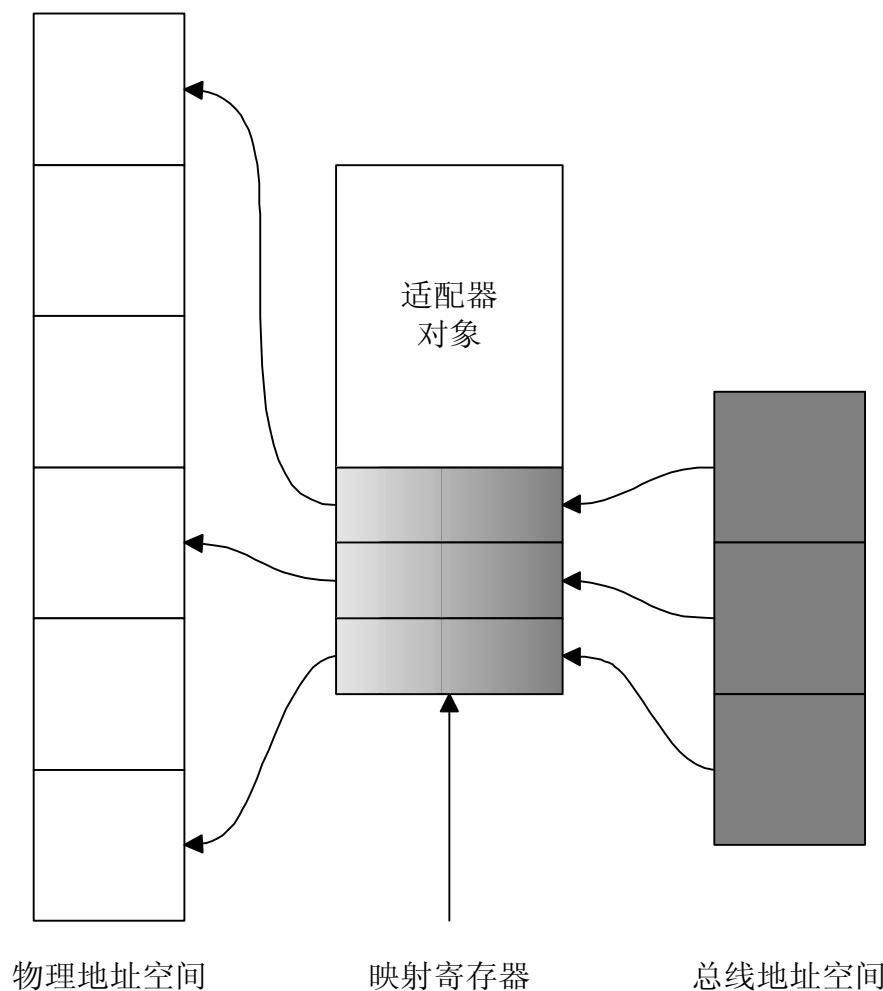


图 7-6. DMA 传输中的抽象计算机模型

某些 CPU，如 Alpha，含有真正的硬件映射寄存器，因此在初始化 DMA 传输时需要有一个步骤来保存某些映射寄存器。其它 CPU，如 Intel 的 x86，没有物理的映射寄存器，但写驱动程序时我们应该假定存在这些寄存器。在这样的计算机中，MapTransfer 函数应保留使用属于系统的物理内存缓冲区，这样，DMA 操作将使用保留的缓冲区。显然，在 DMA 传输前后必须有代码用来在 DMA 缓冲区和内存间复制数据。例如，在某些情况下，当应付一个有分散/聚集能力的总线主控设备时，MapTransfer 阶段应该做无映射寄存器机器不能完成的全部工作。

Windows 2000 内核使用一个称为适配器对象的数据结构来描述设备上的 DMA 特征，并用它来控制访问潜在的共享资源，如系统 DMA 通道和映射寄存器。你可以在 StartDevice 函数中调用 **IoGetDmaAdapter** 获得适配器对象的指针。适配器对象中有一个指针，指向一个名为 **DmaOperations** 的结构，该结构包含了所有你需要的其它函数，见表 7-4。这些函数代替了以前版本 Windows NT 中的全局函数(如 **IoAllocateAdapter**、**IoMapTransfer**，等等)。事实上，现在的全局函数都是调用 DmaOperations 函数的宏。

表 7-4. DMA 辅助例程中的 **DmaOperations** 函数指针

DmaOperations 函数指针	描述
PutDmaAdapter	销毁适配器对象
AllocateCommonBuffer	分配公用缓冲区
FreeCommonBuffer	释放公用缓冲区

AllocateAdapterChannel	保留适配器和映射寄存器
FlushAdapterBuffers	传输后刷新中间数据缓冲区
FreeAdapterChannel	释放适配器和映射寄存器
FreeMapRegisters	仅释放映射寄存器
MapTransfer	安排传输的一个阶段
GetDmaAlignment	取适配器的地址对齐要求
ReadDmaCounter	取余量计数
GetScatterGatherList	保留适配器并构造分散/聚集列表
PutScatterGatherList	释放分散/聚集列表

传输策略

执行一个 DMA 传输需要考虑下面几个因素：

- 如果设备有总线主控能力，那么它就有访问主存的必要电子部件，因此我们只需要告诉它几个基本事实，如从哪开始，需要传输多少单位的数据，是输入操作还是输出操作，等等。你可以向硬件设计者咨询以得到细节部分，否则你只能参考硬件级的说明文档。
- 一个有分散/聚集能力的设备可以在自身与不连续的物理内存区之间传输大块数据。设备的分散/聚集能力对软件十分有利，它可以避免对具有连续页帧的大数据块的需求。页可以被简单地锁定在所在的物理内存，然后把内存地址告诉设备。
- 如果设备不是总线主控设备，那么你需要使用计算机主板上的系统 DMA 控制器。这种形式的 DMA 传输有时被称为副 DMA。与 ISA 总线连接的系统 DMA 控制器对所能访问的物理内存和一次传输的数据量会有些限制。EISA 总线的 DMA 控制器去掉了这些限制。在 Windows 2000 中，你不必知道硬件具体插入到哪种类型的总线，因为系统自动参考这些不同的限定。
- 通常，DMA 操作将包括编程硬件映射寄存器或操作前后的数据复制。如果设备需要连续地读写数据，你不会希望在每次 I/O 请求中都做这两步，这将大大地降低处理速度，在某些情况下是不能接受的。因此，你应该分配一个公用缓冲区，设备和驱动程序可以在任何时间同时访问这个缓冲区。

尽管这四种因素的相互影响会产生许多种不同的结果，但你执行的步骤中有许多共同的特征。图 7-7 显示了一次传输过程。在 **StartIo** 例程中，你通过请求适配器对象的所有权而启动了 DMA 传输。所有权仅在你与其它设备共享一个系统 DMA 通道时才有意义，但 Windows 2000 的 DMA 模型要求你必须执行这一步。当 I/O 管理器同意给你适配器对象的所有权后，它就分配一些映射寄存器为你临时使用，并回调你提供的 **AdapterControl** 例程。在你的 **AdapterControl** 例程中，你执行一个“传输映射”步骤来安排传输的第一阶段(也许是仅有的)。如果映射寄存器不够，则需要多个传输阶段；你的设备必须能处理发生在两个阶段间的任何延迟。

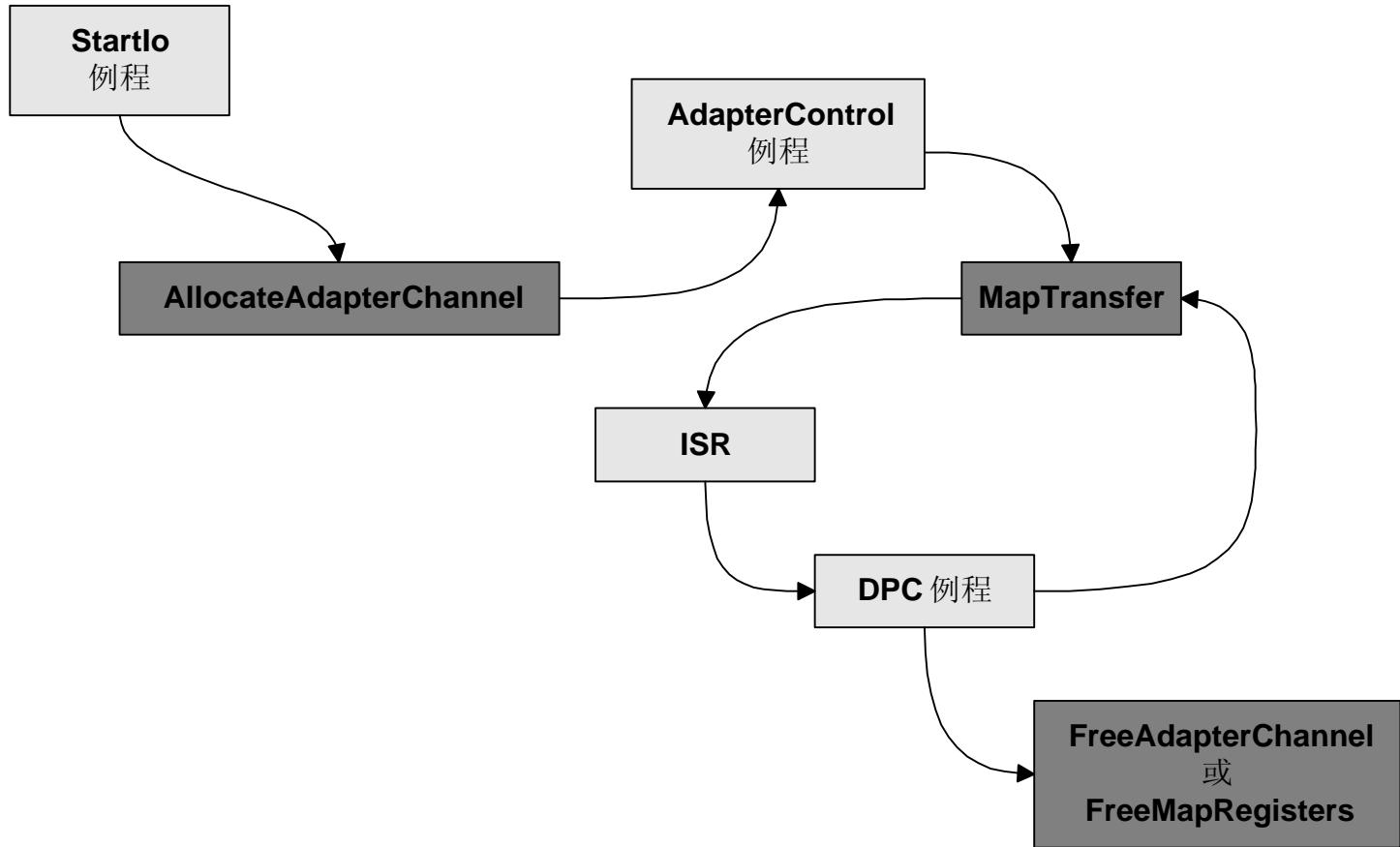


图 7-7. DMA 传输期间的所有权流程

一旦 **AdapterControl** 例程初始化完第一阶段的映射寄存器，就可以通知设备开始操作。传输完成后设备将产生一个中断，于是系统调度一个 **DPC**。如果必要，**DPC** 例程将初始化下一阶段的传输，否则它将完成这个请求。

在这个过程的某个时刻，你将释放映射寄存器和适配器对象。这两个事件发生的时间会因为上面提到的几种因素而改变。

执行DMA传输

现在让我们详细了解被称为“基于包(packet-based)”的 DMA 传输，在这种方式中，你将使用 IRP 携带的数据缓冲区来传输一个固定量的数据。为了简单，让我们假设面对当前最普通的情况：一个基于 PCI 的总线主控设备并且没有分散/聚集能力。

开始，当你创建设备对象时，你通常指定使用 **Direct** 缓冲方式，即设置 **DO_DIRECT_IO** 标志。这里你也应该使用这种方法，因为你调用的 **MapTransfer** 函数需要一个内存描述符列表作为参数，尽管这种选择会对缓冲区对齐造成一点小麻烦。除非应用程序在 **CreateFile** 调用中使用 **FILE_FLAG_NO_BUFFERING** 标志，I/O 管理器从不强制设备对象的 **Alignment-Requirement** 值对齐到用户模式的数据缓冲区上。(除非在 checked 版本的系统中，否则绝不要对内核模式的调用者做这种强制要求) 但是，如果你的设备或 HAL 确实需要 DMA 缓冲区开始于某个特殊的边界，你应该最后把一小部分用户数据复制到正确对齐的内部缓冲区中以迎合这个对齐需求，或者干脆拒绝任何缓冲区未对齐请求。

在 **StartDevice** 函数中，可以使用下面代码创建一个适配器对象：

```

INTERFACE_TYPE bustype;
ULONG junk;
IoGetDeviceProperty(pdx->Pdo,
                    DevicePropertyLegacyBusType,
                    sizeof(bustype),
                    &bustype,
                    &junk);

```

```

DEVICE_DESCRIPTION dd;
RtlZeroMemory(&dd, sizeof(dd));
dd.Version = DEVICE_DESCRIPTION_VERSION;
dd.Master = TRUE;
dd.InterfaceType = bustype;
dd.MaximumLength = MAXTRANSFER;
dd.Dma32BitAddresses = TRUE;

pdx->AdapterObject = IoGetDmaAdapter(pdx->Pdo, &dd, &pdx->nMapRegisters);

```

这段代码的最后一条语句特别重要。**IoGetDmaAdapter** 函数将与总线驱动程序或 HAL 沟通，以创建一个适配器对象，然后返回地址。第一个参数(**pdx->Pdo**)指出设备的 PDO。第二个参数指向一个 **DEVICE_DESCRIPTION** 结构，初始化该结构使其描述设备的 DMA 特征。最后一个参数指示系统应该在哪存放映射寄存器的最大数量。注意，我在设备扩展中保留了两个域(**AdapterObject** 和 **nMapRegisters**)，它们将用于接收该函数的这两个输出。

在 **StopDevice** 函数中，使用这个调用销毁适配器对象：

```

VOID StopDevice(...)
{
    ...
    if (pdx->AdapterObject)
        (*pdx->AdapterObject->DmaOperations->PutDmaAdapter)(pdx->AdapterObject);
    pdx->AdapterObject = NULL;
    ...
}

```

如果设备是总线主控方式，那么你不能获得 DMA 资源。即你的资源提取循环语句中不需要 **CmResourceTypeDma** 类型的 **case** 子句。PnP 管理器不赋予你任何 DMA 资源，因为硬件本身包含有执行 DMA 传输所必须的所有电路逻辑，所以系统没必要赋予你 DMA 资源。

以前版本的 Windows NT 依赖一个名为 **HalGetAdapter** 的函数来获取 DMA 适配器对象。为了兼容，该函数现在仍存在，但新的 WDM 驱动程序应该调用 **IoGetDmaAdapter** 函数替代。这两个函数的不同之处是，**IoGetDmaAdapter** 函数先发出一个即插即用请求 **IRP_MN_QUERY_INTERFACE** 来确定 PDO 是否支持直接调用接口 **GUID_BUS_INTERFACE_STANDARD**。如果支持，**IoGetDmaAdapter** 将使用这个接口来分配适配器对象。如果不支持，它就简单地调用 **HalGetAdapter** 函数。

表 7-5 列出了 **DEVICE_DESCRIPTION** 结构中的所有域，你需要把这个结构传递给 **IoGetDmaAdapter** 函数。与总线主控设备相关的域就是前面 **StartDevice** 代码中使用的域。HAL 也许需要或不需要了解设备是否认识 32 位或 64 位地址，(例如，Intel x86 的 HAL 仅在你分配公共缓冲区时才使用这个标志) 但为了保持可移植性，你应该指出这个能力。通过清零整个结构，我们设置了 **ScatterGather** 为 FALSE。由于我们不使用系统 DMA 通道，所以创建适配器对象的例程将不检查 **DmaChannel**、**DmaPort**、**DmaWidth**、**DemandMode**、**AutoInitialize**、**IgnoreCount**、**DmaSpeed** 域。

表 7-5. *IoGetDmaAdapter* 函数使用的 **DEVICE_DESCRIPTION** 结构

域名	描述	与设备的关系
Version	结构的版本号，初始化为 DEVICE_DESCRIPTION_VERSION	All
Master	总线主控设备，设置基于实际设备	All
ScatterGather	设备支持分散/聚集表，设置基于实际设备	All
DemandMode	使用系统 DMA 控制器的需求模式，设置基于实际设备	Slave
AutoInitialize	使用系统 DMA 控制器的自动初始化模式，设置基于实际设备	Slave
Dma32BitAddresses	能够使用 32 位物理地址	公共缓冲区

IgnoreCount	控制器不维护一个精确的传输计数，设置基于实际设备	Slave
Reserved1	保留，必须为 FALSE	
Dma64BitAddresses	能够使用 64 位物理地址	公共缓冲区
DoNotUse2	保留，必须为 0	
DmaChannel	DMA 通道号，用资源描述符中的通道属性信息初始化	Slave
InterfaceType	总线类型，DevicePropertyLegacyBusType 来自 IoGetDeviceProperty 调用的结果	All
DmaWidth	传输的宽度，Width8Bits、Width16Bits、Width32Bits	Slave
DmaSpeed	传输速度，Compatible、TypeA、TypeB、TypeC、TypeF	Slave
MaximumLength	单个传输的最大长度，设置基于实际设备(近似为 PAGE_SIZE 的整数倍)	All
DmaPort	Microchannel 类型总线端口号，用端口资源描述符属性信息初始化	Slave

为了初始化一个 I/O 操作，StartIo 例程必须首先调用适配器对象的 **AllocateAdapterChannel** 例程以保留适配器对象。该函数的一个参数是 AdapterControl 例程的地址，I/O 管理器将在保留操作完成后调用这个地址。下面代码例子演示了 AllocateAdapterChannel 函数的准备和调用过程：

```

typedef struct _DEVICE_EXTENSION {
    ...
    PADAPTER_OBJECT AdapterObject;           // device's adapter object          <--1
    ULONG nMapRegisters;                    // max # map registers
    ULONG nMapRegistersAllocated;           // # allocated for this xfer
    ULONG numxfer;                         // # bytes transferred so far
    ULONG xfer;                            // # bytes to transfer during this stage
    ULONG nbytes;                          // # bytes remaining to transfer
    PVOID vaddr;                           // virtual address for current stage
    PVOID regbase;                         // map register base for this stage
    ...
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

VOID StartIo(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);           <
    if (!NT_SUCCESS(status))
    {
        CompleteRequest(Irp, status, 0);
        return;
    }

    PMDL mdl = Irp->MdlAddress;           <
    pdx->numxfer = 0;
    pdx->xfer = pdx->nbytes = MmGetMdlByteCount(mdl);
    pdx->vaddr = MmGetMdlVirtualAddress(mdl);           <

    ULONG nregs = ADDRESS_AND_SIZE_TO_SPAN_PAGES(pdx->vaddr, pdx->nbytes);   <
    if (nregs > pdx->nMapRegisters)
    {
        nregs = pdx->nMapRegisters;
        pdx->xfer = nregs * PAGE_SIZE - MmGetMdlByteOffset(mdl);
    }
    pdx->nMapRegistersAllocated = nregs;
}

```

```

status = (*pdx->AdapterObject->DmaOperations->AllocateAdapterChannel)
        (
            pdx->AdapterObject,
            fdo,
            nregs,
            (PDRIVER_CONTROL) AdapterControl,
            pdx);
if (!NT_SUCCESS(status))
{
    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    CompleteRequest(Irp, status, 0);
    StartNextPacket(&pdx->dqReadWrite, fdo);
}
}

```

1. 你的设备扩展需要几个与 DMA 传输相关的域。注释信息指出了这些域的用途。
2. 此时正是请求“删除自旋锁”的时候，这个自旋锁将阻止在 I/O 操作未决期间出现 PnP 删除事件。平衡 **IoReleaseRemoveLock** 函数的调用发生在最终完成请求的 DPC 例程中。
3. 这几条语句初始化设备扩展中的几个域，为传输的第一阶段做准备。
4. 这里，我们计算在这次传输中要求系统为我们保留多少映射寄存器。我们从计算整个传输需求的寄存器个数开始。 ADDRESS_AND_SIZE_TO_SPAN_PAGES 宏会考虑到缓冲区也许会跨过页界限。然而，这个数值最后可能超过 **IoGetDmaAdapter** 调用允许的最大值。在这种情况下，我们需要执行多步传输。因此我们将回到第一阶段以便能使用允许数量的映射寄存器。我们还需要记住分配了多少个映射寄存器(在设备扩展的 **nMapRegistersAllocated** 域)以便以后我们能正确地释放它们。
5. 在这个 **AllocateAdapterChannel** 调用中，我们指定了适配器对象的地址、我们自己设备对象的地址，映射寄存器个数的计算结果，以及 **AdapterControl** 例程的地址。最后一个参数(**pdx**)是 **AdapterControl** 例程的上下文参数。

通常，几个设备可以共享一个适配器对象。但在实际环境中，共享仅发生在需要依赖系统 DMA 控制器时；总线主控设备拥有自己专用的适配器对象。但是，因为你不必了解系统在创建适配器对象时是如何做决定的，所以你不要做关于这一点的任何假设。当你调用 **AllocateAdapterChannel** 时，适配器对象也许会忙，因此你的请求将被放到一个队列中，直到这个适配器对象恢复到有效状态。同样，计算机上的所有 DMA 设备共享一组映射寄存器。进一步的延迟能跟着发生，直到请求数量的寄存器都变为有效。这两个延迟都发生在 **AllocateAdapterChannel** 中，该函数会在适配器对象和所有你请求的映射寄存器都变为有效时才调用你的 **AdapterControl** 例程。

尽管 PCI 总线主控设备拥有自己的适配器对象，但如果设备没有分散/聚集能力，它仍需要使用映射寄存器。而在 Alpha 这样的 CPU 上存在着映射寄存器，**AllocateAdapterChannel** 将把这些寄存器保留给你使用。但像 Intel 这样的 CPU 不存在映射寄存器，**AllocateAdapterChannel** 将使用软件模拟，例如物理内存中的一段连续区域。

AllocateAdapterChannel 把什么排入队列？

为了等待适配器对象或必要数量的映射寄存器，**AllocateAdapterChannel** 把你的设备对象放入了队列。有些设备结构可以允许你同时执行多个 DMA 传输。因为你一次只能把一个设备对象放入适配器对象的队列中(这样不会弄坏系统)，你需要创建多个哑设备对象来利用这种多 DMA 传输的能力。

正如我们讨论过的，**AllocateAdapterChannel** 最终将调用你的 **AdapterControl** 例程(在 **DISPATCH_LEVEL**，就象你的 **StartIo** 例程)。你有两个任务需要完成。第一，你应该调用适配器对象的 **MapTransfer** 例程以便为 I/O 操作的第一阶段准备映射寄存器和其它系统资源。如果是总线主控设备，**MapTransfer** 将返回一个代表传输第一阶段开始点的逻辑地址。这个逻辑地址可能会与 CPU 的物理内存地址相同，但也可能会不同。你所需要知道的全部就是该地址是编程设备硬件的正确地址。**MapTransfer** 会修正你请求的传输长度以适合其使用的映射寄存器，所以你应该把保存当前阶段传输长度的变量地址作为参数提供。

第二个任务是执行与“通知设备这个物理地址”操作中涉及的任何设备相关的操作步骤，然后开始操作设备硬件：

IO_ALLOCATION_ACTION

```

AdapterControl(PDEVICE_OBJECT fdo, PIRP junk, PVOID regbase, PDEVICE_EXTENSION pdx)
{
    PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
    PMDL mdl = Irp->MdlAddress;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    BOOLEAN isread = stack->MajorFunction == IRP_MJ_READ;
    pdx->regbase = regbase;
    KeFlushIoBuffers(mdl, isread, TRUE);
    PHYSICAL_ADDRESS address = (*pdx->AdapterObject->DmaOperations->MapTransfer)
        (pdx->AdapterObject,
         mdl,
         regbase,
         pdx->vaddr,
         &pdx->xfer,
         !isread);

    ...
    return DeallocateObjectKeepRegisters;
}

```

1. **AdapterControl** 的第二个参数名为 **junk**, 是设备对象的 **CurrentIrp** 域中存放的任何值。如果你使用 **DEVQUEUE** 来排队 IRP, 你需要询问 **DEVQUEUE** 对象当前是什么 IRP。如果你使用标准模型, 即, 使用 **IoStartPacket** 和 **IoStartNextPacket** 来管理队列, 那么 **junk** 将是正确的 IRP。在那种情况下, 我就把这个参数命名为 **Irp**。
2. 使用 DMA 的输入和输出操作之间很相似, 所以为了方便通常把处理这两个操作的代码放到一个单独的子程序中。这行代码检查 IRP 的主功能码以确定是读操作还是写操作。
3. 这个函数的 **regbase** 参数是一个不透明的句柄, 标识映射寄存器组, 这些寄存器为这个操作而保留。你以后会需要这个值, 因此应该把它保存到设备扩展中。
4. **KeFlushIoBuffers** 确保所有处理器高速缓冲中的内容为你正使用的内存缓冲区中的内容。第三个参数(**TRUE**)指出你正为准备一个 DMA 操作而刷新高速缓冲。有些 CPU 结构需要这个步骤, 但通常情况下, DMA 传输直接面向内存不必涉及高速缓冲。
5. **MapTransfer** 例程为传输的一个阶段编程 DMA 硬件, 然后返回传输起始的物理地址。注意你应该为该函数的第二个参数提供一个 **MDL** 的地址。因为在这里你需要一个 **MDL**, 在第一次创建设备对象时你还可以选择 **DO_DIRECT_IO** 缓冲区模式, 这样 I/O 管理器将为你自动创建 **MDL**。你还需要传递映射寄存器基地址(**regbase**)。你要指出 **MDL** 的哪部分卷入这个阶段的操作, 即提供一个虚拟地址(**pdx->vaddr**)和一个字节计数(**pdx->xfer**)。**MapTransfer** 将使用这个虚拟地址参数计算缓冲区中的一个偏移, 利用这个偏移地址它能确定包含你的数据的物理页号。
6. 在这里你可以以设备专用的方式编程你的硬件。例如, 你也许使用某个 **WRITE_Xxx HAL** 例程向卡上的寄存器发送物理地址和字节计数值, 然后你可能选通某个命令寄存器开始传输数据。
7. 我们返回常量 **DeallocateObjectKeepRegisters** 以指出我们使用完适配器对象但仍在使用映射寄存器。在这个特别的例子中(PCI 总线主控), 对适配器对象没有任何争夺, 所以释放适配器对象不会有事。在其它总线主控形式中, 我们可能与其它设备共享一个 DMA 控制器。释放适配器对象将允许其它设备使用我们用过的映射寄存器组进行传输。

中断通常都发生在传输启动后的很短一段时间后, 并且中断服务例程通常都请求一个 **DPC** 来处理传输第一阶段的完成。你的 **DPC** 例程可以是这样:

```

VOID DpcForIsr(PKDPC Dpc, PDEVICE_OBJECT fdo, PIRP junk, PVOID Context)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
    PMDL mdl = Irp->MdlAddress;
    BOOLEAN isread = IoGetCurrentIrpStackLocation(Irp)->MajorFunction == IRP_MJ_READ;
    (*pdx->AdapterObject->DmaOperations->FlushAdapterBuffers)(pdx->AdapterObject,
        mdl,
        pdx->regbase,
        pdx->vaddr,
        pdx->xfer,
        !isread);

    pdx->nbytes -= pdx->xfer;
}

```

```

pdx->numxfer += pdx->xfer;
NTSTATUS status = STATUS_SUCCESS;
...
if (pdx->nbytes && NT_SUCCESS(status))
{
    pdx->vaddr = (PVOID) ((PUCHAR) pdx->vaddr + pdx->xfer);
    pdx->xfer = pdx->nbytes;
    ULONG nregs = ADDRESS_AND_SIZE_TO_SPAN_PAGES(pdx->vaddr, pdx->nbytes);
    if (nregs > pdx->nMapRegistersAllocated)
    {
        nregs = pdx->nMapRegistersAllocated;
        pdx->xfer = nregs * PAGE_SIZE;
    }
    PHYSICAL_ADDRESS address =
(*pdx->AdapterObject->DmaOperations->MapTransfer)(pdx->AdapterObject,
                                                       mdl,
                                                       pdx->regbase,
                                                       pdx->vaddr,
                                                       pdx->xfer,
                                                       !isread);
    ...
}
else
{
    ULONG numxfer = pdx->numxfer;
    (*pdx->AdapterObject->DmaOperations->FreeMapRegisters)(pdx->AdapterObject,
                                                               pdx->regbase,
                                                               pdx->nMapRegistersAllocated);
    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    StartNextPacket(&pdx->dqReadWrite, fdo);
    CompleteRequest(Irp, status, numxfer);
}
}
}

```

1. 当使用 **DEVQUEUE** 排队 IRP 时，你可以依靠这种队列对象来跟踪当前 IRP。
2. **FlushAdapterBuffers** 用于应付使用系统拥有的中间缓冲区进行传输的情形。如果你做完的输入操作跨过了一个页边界，则输入数据现在停留在中间缓冲区中，需要复制到用户模式缓冲区。
3. 这里，我们在刚完成的传输阶段后更新了剩余的和累积的数据计数。
4. 在这点上，确定完成的当前传输阶段是成功还是返回错误。例如，读一个状态端口或检查由中断例程执行的类似操作所返回的结果。在这个例子中，我把 **status** 变量设置为 **STATUS_SUCCESS**，期望你如果发现错误就改变这个变量。
5. 如果传输还没完成，你需要编程下一个传输阶段。这个过程的第一步是计算用户模式缓冲区下一部分的虚拟地址。记住，这个计算只是用一个数，实际上我们并不用这个虚拟地址访问内存。访问内存是一个坏的想法，因为我们当前执行在一个任意线程上下文中。
6. 接下来的几个语句与在第一阶段中 **StartIo** 和 **AdapterControl** 执行的语句几乎完全相同。最后的结果将是一个逻辑地址，可以被编程到你的设备中。它对应的物理地址可能会也可能不会被 **CPU** 理解。一个小的不利是我们被限制在只能使用由 **AdapterControl** 例程分配的那些映射寄存器；**StartIo** 把这些寄存器的个数保存在设备扩展的 **nMapRegistersAllocated** 域中。
7. 如果整个传输现在完成，我们需要释放使用过的映射寄存器。
8. 该 DPC 例程的剩下几条语句用于处理 IRP 的完成机制。在这里我们释放了在 **StartIo** 中获取的删除自旋锁。

使用分散/聚集表的传输

如果硬件支持分散/聚集能力，那么系统可以更容易地处理设备的 DMA 传输。分散/聚集能力允许设备传输所涉及的页在物理内存中不连续。

这种设备的 **StartDevice** 例程在创建适配器对象上使用与前面讨论的同样的方式，除了 **ScatterGather** 标志应该设为 **TRUE**。

传统方法(即在以前版本 Windows NT 中使用的方法)在安排一个涉及分散/聚集功能的 DMA 传输时，几乎等同于前面段“执行 DMA 传输”中的基于包的例子。新方法的唯一不同之处是它需要为传输的每个阶段多次调用 **MapTransfer**。每次调用后你都会得到分散/聚集列表中的一个包含物理地址和长度的数组元素。当循环完成时，你可以使用设备专用的方法把分散/聚集列表发送到设备，然后开始传输。

我需要做一些假设以便你能恰当地构造分散/聚集列表。首先，我假设你已经定义了一个 **MAXSG** 常量，它指出设备能处理的最大的分散/聚集列表元素个数。为了使事情更简单，我还要假设你能使用 WDM.H 中的 **SCATTER_GATHER_LIST** 结构构造这个列表：

```
typedef struct _SCATTER_GATHER_ELEMENT {
    PHYSICAL_ADDRESS Address;
    ULONG Length;
    ULONG_PTR Reserved;
} SCATTER_GATHER_ELEMENT, *PSCATTER_GATHER_ELEMENT;

typedef struct _SCATTER_GATHER_LIST {
    ULONG NumberOfElements;
    ULONG_PTR Reserved;
    SCATTER_GATHER_ELEMENT Elements[];
} SCATTER_GATHER_LIST, *PSCATTER_GATHER_LIST;
```

最后，我简单地假设你在 **AddDevice** 函数中分配了最大容量的分散/聚集列表：

```
pdx->sglist = (PSCATTER_GATHER_LIST)
    ExAllocatePool(NonPagedPool,
    sizeof(SCATTER_GATHER_LIST)+MAXSG*sizeof(SCATTER_GATHER_ELEMENT));
```

有了这些基础，你的 **AdapterControl** 例程应该像这样：

```
IO_ALLOCATION_ACTION AdapterControl(PDEVICE_OBJECT fdo, PIRP junk, PVOID regbase,
PDEVICE_EXTENSION pdx)
{
    PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
    PMDL mdl = Irp->MdlAddress;
    BOOLEAN isread = IoGetCurrentIrpStackLocation(Irp)->MajorFunction == IRP_MJ_READ;
    pdx->regbase = regbase;
    KeFlushIoBuffers(mdl, isread, TRUE);
    PSCATTER_GATHER_LIST sglist = pdx->sglist;

    ULONG xfer = pdx->xfer;
    PVOID vaddr = pdx->vaddr;
    pdx->xfer = 0;
    ULONG isg = 0;
    while (xfer && isg < MAXSG)
    {
        ULONG elen = xfer;
        sglist->Elements[isg].Address =
            (*pdx->AdapterObject->DmaOperations->MapTransfer)(pdx->AdapterObject,
            mdl,
            regbase,
            pdx->vaddr,
            &elen,
            !isread);
```

```

sglist->Elements[isg].Length = elen;
xfer -= elen;
pdx->xfer += elen;
vaddr = (PVOID) ((PUCHAR) vaddr + elen);
++isg;
}
sglist->NumberOfElements = isg;
...
return DeallocateObjectKeepRegisters;
}

```

1. 见早期的讨论(在“执行 DMA 传输”中), 关于如何在 **AdapterControl** 例程中获得正确的 IRP 指针。
2. 我们以前(在 **StartIo**)曾基于可允许的映射寄存器数量计算过 **pdx->xfer**。我们将要开始传输那些数据, 但是可以允许的分散/聚集元素个数会进一步限制这个阶段的传输量。在接下来的循环中, **xfer** 将等于还没有被映射的字节个数, 因此我们需要重新计算 **pdx->xfer**。
3. 这就是我们调用 **MapTransfer** 函数来构造分散/聚集元素的循环。我们将继续这个循环直到已经映射了该传输的整个阶段, 或者到我们用完分散/聚集元素。
4. 当我们为一个分散/聚集设备调用 **MapTransfer** 函数时, 该函数将修改长度参数(**elen**)以指出给定虚拟地址(**vaddr**)中从哪开始是物理上连续的区域, 从而可以被映射为一个单独的分散/聚集元素。它还返回这个连续区域的起始物理地址。
5. 这里就是我们修改描述当前传输阶段变量的地方。当离开这个循环时, **xfer** 将回到 0(否则就是我们用完了分散/聚集元素), **pdx->xfer** 将等于我们能映射的全部元素的总数, **vaddr** 将等于最后一个映射之后的字节, 我们不用修改设备扩展中的 **pdx->vaddr** 域, 我们正在 DPC 例程中做这件事情。另一个讨厌的细节。
6. 这里是我们增加分散/聚集元素索引以表示我们已经使用过一个。
7. 在这点上, 我们把分散/聚集元素 **isg** 编程到设备中。然后为该请求启动设备工作。
8. 返回 **DeallocateObjectKeepRegisters** 对于总线主控设备是合适的。理论上你可以有一个有分散/聚集能力的非主控设备, 但应返回 **KeepObject**。

设备现在大概正执行 DMA 传输, 然后用中断通知传输完成。你的 ISR 请求了一个 DPC, 之后你的 DPC 例程又初始化该操作的下一个阶段。DPC 例程将执行一个 **MapTransfer** 循环, 就象我在初始化过程中给出的循环。我把这段代码细节交给你去练习。

使用**GetScatterGatherList**

Windows 2000 为普通情况下的循环调用 **MapTransfer** 函数提供了一个捷径, 它避免了相对麻烦的循环, 你可以不使用映射寄存器或者使用不超过映射寄存器最多个数来完成整个传输。这个捷径在光盘上的 **SCATGATH** 例子中给出, 它包含了对 **GetScatterGatherList** 例程的调用而不是 **AllocateAdapterChannel**。你的 **StartIo** 例程应该像这样:

```

VOID StartIo(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
    if (!NT_SUCCESS(status))
    {
        CompleteRequest(Irp, status, 0);
        return;
    }
    PMDL mdl = Irp->MdlAddress;
    ULONG nbytes = MmGetMdlByteCount(mdl);
    PVOID vaddr = MmGetMdlVirtualAddress(mdl);
    BOOLEAN isread = stack->MajorFunction == IRP_MJ_READ;
    pdx->numxfer = 0;
    pdx->nbytes = nbytes;
}

```

```

status = (*pdx->AdapterObject->DmaOperations->GetScatterGatherList)
(pdx->AdapterObject,
 fdo,
 mdl,
 vaddr,
 nbytes,
 (PDRIVER_LIST_CONTROL) DmaExecutionRoutine,
 pdx,
 !isread);
if (!NT_SUCCESS(status))
{
    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    CompleteRequest(Irp, status, 0);
    StartNextPacket(&pdx->dqReadWrite, fdo);
}
}

```

GetScatterGatherList 的调用在上面代码片段中用粗体显示，如果需要，该函数将等待，直到系统同意你使用适配器对象和所有你需要的映射寄存器。然后该函数创建一个 **SCATTER_GATHER_LIST** 结构并传递给 **DmaExecutionRoutine** 例程。之后你可以使用分散/聚集元素中的物理地址编程你的设备并开始传输：

```

VOID DmaExecutionRoutine(PDEVICE_OBJECT fdo, PIRP junk, PSCATTER_GATHER_LIST sglist,
PDEVICE_EXTENSION pdx)
{
    PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
    pdx->sglist = sglist;
    ...
}

```

1. 在 **DPC** 例程中你会需要分散/聚集列表的地址，这个地址将被 **PutScatterGatherList** 用于释放列表。
2. 在这个点上，使用分散/聚集列表中的地址长度对(**address and length pairs**)来编程设备的读写操作。如果列表中的元素个数超过设备一次能处理的能力，你需要分阶段执行整个传输。如果你能相当快速地编程一个阶段，我建议你在 **ISR** 中加入初始化下一阶段的逻辑。如果你也这么认为，你的 **DmaExecutionRoutine** 函数也许会与你的 **ISR** 同时执行，所以这个额外的逻辑不要大。我在 **SCATGATH** 例子中使用了这个想法。

当传输完成时，我们调用适配器对象的 **PutScatterGatherList** 函数释放列表和适配器：

```

VOID DpcForIsr(PKDPC Dpc, PDEVICE_OBJECT fdo, PIRP junk, PVOID Context)
{
    ...
    (*pdx->AdapterObject->DmaOperations->PutScatterGatherList)(pdx->AdapterObject, pdx->sglist, !isread);
    ...
}

```

为了判断你是否能使用 **GetScatterGatherList**，你需要能预测是否有适合使用该函数的前提。在 **Intel** 的 32 位平台上，**PCI** 或 **EISA** 总线上的分散/聚集设备可以确保不需要任何映射寄存器。即使在 **ISA** 总线上，你只被允许请求最多 16 个映射寄存器替代(如果是总线主控的设备，则只能有 8 个)，除非物理内存十分紧张以至于 I/O 系统不能分配那么多的中间 I/O 缓冲区。在那种情况下，你不能用传统模式做 DMA 操作，所以也没有需要担心的地方。

如果你在编写驱动程序时不能准确地预测你能否使用 **GetScatterGatherList**，我的建议是回到传统的 **MapTransfer** 调用循环。总之，你需要放入处理 **GetScatterGatherList** 不工作情况的代码，并使驱动程序中的这两段逻辑保持简洁。

使用系统控制器的传输

如果你的设备不是总线主控方式，则需要使用系统 DMA 控制器来执行 DMA 传输，即人们常使用的词“*slave DMA*”，指设备不能管理自己的 DMA 传输。系统 DMA 控制器有几个特征，这影响到如何进行 DMA 传输过程的内部细节：

- 所有 *slave* 设备必须共享有限的 DMA 通道数量。AllocateAdapterChannel 在这种共享情形中具有真正意义，因为一次只能有一个设备可以使用特定的通道。
- 你可以在 PnP 管理器发给你的 I/O 资源列表中找到一个 CmResourceTypeDma 资源。
- 你的硬件应该连接到它使用的特定通道，或者是物理上或者是逻辑上。如果你能配置 DMA 通道连接，应该在 StartDevice 中发出适当的命令。
- ISA 总线的系统 DMA 控制器仅能访问前 16 兆字节的物理内存，其中有四个通道用于传输 8 位数据，三个通道用于传输 16 位数据。8 位通道的控制器不能正确地处理越过 64KB 界限的缓冲区；16 位通道的控制器不能正确地处理越过 128KB 界限的缓冲区。

抛开这些因素，你的驱动程序代码将与总线主控设备的代码非常相似。但 StartDevice 例程中准备 IoGetDmaAdapter 调用的过程会麻烦一些，你的 AdapterControl 例程和 DPC 例程分别负责释放适配器对象和释放映射寄存器。

在 StartDevice 函数中，需要用一点额外的代码来确定 PnP 管理器到底赋给你哪一个 DMA 通道，并且你还需要为 IoGetDmaAdapter 函数初始化 DEVICE_DESCRIPTION 结构中更多的域：

```
NTSTATUS StartDevice(...)

{
    ULONG dmachannel;      // system DMA channel #
    ULONG dmaport;         // MCA bus port number
    ...
    for (ULONG i = 0; i < nres; ++i, ++resource)
    {
        switch (resource->Type)
        {
            case CmResourceTypeDma:
                dmachannel = resource->u.Dma.Channel;
                dmaport = resource->u.Dma.Port;
                break;
        }
    }
    ...
    INTERFACE_TYPE bustype;
    IoGetDeviceProperty(...);

    DEVICE_DESCRIPTION dd;
    RtlZeroMemory(&dd, sizeof(dd));
    dd.Version = DEVICE_DESCRIPTION_VERSION;
    dd.InterfaceType = bustype;
    dd.MaximumLength = MAXTRANSFER;

    dd.DmaChannel = dmachannel;
    dd.DmaPort = dmaport;
    dd.DemandMode = ??;
    dd.AutoInitialize = ??;
    dd.IgnoreCount = ??;
    dd.DmaWidth = ??;
    dd.DmaSpeed = ??;

    pdx->AdapterObject = IoGetDmaAdapter(...);
}
```

1. I/O 资源列表中将含有一个 DMA 资源，从这个资源中你可以提取通道号和端口数。通道号指出系统 DMA 控制器支持的一个 DMA 通道。而端口号仅关系到有微通道(MCA)总线的机器。
2. 如何确定总线类型见以前的讨论(在“执行 DMA 传输”段中)。
3. 从这里开始，你必须初始化 DEVICE_DESCRIPTION 结构中的几个域，基于你对自己设备的了解。见表 7-5。

关于适配器控制和 DPC 例程的每件事都与前面处理无分散/聚集能力的总线主控设备中相同，但有两个细节除外。第一，**AdapterControl** 返回另一个值：

```
IO_ALLOCATION_ACTION AdapterControl(...)  
{  
    ...  
    return KeepObject;  
}
```

返回值 **KeepObject** 指出我们希望保留映射寄存器和 DMA 通道的控制。第二，因为我们在 AdapterControl 返回时没有释放适配器对象，所以我们必须在 DPC 例程中做，调用 **FreeAdapterChannel** 替代 **FreeMapRegisters**：

```
VOID DpcForIsr(...)  
{  
    ...  
    (*pdx->AdapterObject->DmaOperations->FreeAdapterChannel)(pdx->AdapterObject);  
    ...  
}
```

另外，你不需要记住被赋予多少个映射寄存器，即我以前在设备扩展的 **nMapRegistersAllocated** 变量中保存的数值，因为你不再调用 **FreeMapRegisters** 函数。

使用公用缓冲区

正如我在“传输策略”中提到的，你可能希望在执行 DMA 传输中为设备分配一个公共缓冲区。公共缓冲区就是一块非分页的，物理上连续的内存。驱动程序使用固定的虚拟地址来访问这种缓冲区，设备则使用固定的逻辑地址来访问同一个缓冲区。

使用公共缓冲区有多种方式，你可以使用系统 DMA 控制器的自动启动模式来支持设备和内存间的连续数据传输。在这种模式中，传输的完成触发了控制器立即启动下一次传输。

公用缓冲区的另一个用途是避免额外数据拷贝。**MapTransfer** 例程通常把你提供的数据复制到 I/O 管理器拥有的辅助缓冲区中并在 DMA 传输中使用。如果你要在 ISA 总线上做 **Slave** 方式的 DMA，那么 **MapTransfer** 很可能按 ISA DMA 控制器的 16MB 地址和缓冲区对齐需求复制数据。但是如果你有一个公用缓冲区，你就可以避免这个复制步骤。

分配公用缓冲区

通常你应该在 **StartDevice** 中，在创建适配器对象后分配公用缓冲区：

```
typedef struct _DEVICE_EXTENSION {  
    ...  
    PVOID vaCommonBuffer;  
    PHYSICAL_ADDRESS paCommonBuffer;  
    ...  
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;  
  
dd.Dma32BitAddresses = ??;
```

```
dd.Dma64BitAddresses = ??;
pdx->AdapterObject = IoGetDmaAdapter(...);
pdx->vaCommonBuffer =
(*pdx->AdapterObject->DmaOperations->AllocateCommonBuffer)(pdx->AdapterObject,
                                                               <length>,
                                                               &pdx->paCommonBuffer,
                                                               FALSE);
```

在调用 **IoGetDmaAdapter** 之前，你应该在 **DEVICE_DESCRIPTION** 结构中设置 **Dma32BitAddresses** 或 **Dma64BitAddresses** 标志以表明设备的实际寻址能力，即如果设备能使用 32 位物理地址寻址一个缓冲区，则设置 **Dma32BitAddresses** 为 TRUE。如果设备能使用 64 位物理地址寻址一个缓冲区，则应设置 **Dma64BitAddresses** 为 TRUE。一定要正确设置这个标志，否则在带有物理内存扩展的 Intel 机器上，驱动程序将不能正常工作。

在调用 **AllocateCommonBuffer** 时，第二个参数是你要分配缓冲区的字节长度。第四个参数是布尔值，它表明你是否需要使分配的内存进入 CPU 高速缓存。

AllocateCommonBuffer 返回一个虚拟地址。它就是你在驱动程序中用于访问已分配缓冲区的地址。**AllocateCommonBuffer** 还通过其第三个参数设置了 **PHYSICAL_ADDRESS** 为设备使用的逻辑地址。

注意

DDK 谨慎地使用术语“逻辑地址”来代表由 **MapTransfer** 返回的或由 **AllocateCommonBuffer** 第三个参数返回的地址值。在许多 CPU 构架中，逻辑地址通常就是 CPU 可以理解的物理地址。但在某些 CPU 中，逻辑地址可以是仅由 I/O 总线才理解的地址。也许“总线地址”才是一个更好的术语。

使用公用缓冲区的**Slave**模式**DMA**传输

如果你准备执行 **Slave** 模式的 DMA 操作，那么你必须为接收数据的虚拟地址创建一个 **MDL**。**MDL** 的实际目的是在最后调用 **MapTransfer** 中占用一个参数槽，它需要用 **MDL** 来找到不必复制的数据。通常应该在 **StartDevice** 函数中，紧接着分配公用缓冲区之后创建该 **MDL**:

```
pdx->vaCommonBuffer = ...;
pdx->mdlCommonBuffer = IoAllocateMdl(pdx->vaCommonBuffer, <length>, FALSE, FALSE, NULL);
MmBuildMdlForNonPagedPool(pdx->mdlCommonBuffer);
```

为了执行一个输出操作，首先应通过某些手段(如一个明确的内存复制)使公用缓冲区中包含有要发往到设备的数据。而所需的其它 DMA 逻辑基本上与我以前在“执行 DMA 传输”中讲的一致。你将调用 **AllocateAdapterChannel**，该函数调用你的适配器控制例程，这个例程又调用 **KeFlushIoBuffers**(如果你分配了一个可被高速缓存的缓冲区)，最后调用 **MapTransfer**。你的 DPC 例程将调用 **FlushAdapterBuffers** 和 **FreeAdapterChannel** 函数。在所有这些调用中，你应该指定公用缓冲区的 **MDL** 来代替读写 IRP 携带的 **MDL**。在输入操作的结尾，你应该把数据从公用缓冲区复制到其它某个地方。

为了完成一个传输数据量大于公用缓冲区的读写请求，你应该定时地补充或清空缓冲区。适配器对象的 **ReadDmaCounter** 函数可以帮助你了解数据传输的进展。

使用公用缓冲区的总线主控模式**DMA**传输

如果设备是总线主控模式，那么在分配公用缓冲区时就没有必要调用 **AllocateAdapterChannel**、**MapTransfer**、**FreeMapRegisters** 函数。因为 **AllocateCommonBuffer** 也能保留必要的映射寄存器。每个总线主控设备都有一个适配器对象，该对象不与其它设备共享，因此你不必等待。由于你拥有可以在任何时间都能访问缓冲区的虚

拟地址，又由于设备的总线主控能力允许你使用物理地址(由 **AllocateCommonBuffer** 返回)访问该缓冲区，所以没有额外的工作需要做。

使用公用缓冲区的注意事项

这里有一些关于公用缓冲区分配和使用的注意事项。在运行系统中，物理上连续的内存是十分稀有的，有时你根本得不到这样的内存，除非在系统启动的早期请求这样的内存。内存管理器为了满足你的请求会对内存页进行重排列，但它的尝试十分有限，并且这个过程会推迟 **AllocateCommonBuffer** 的返回。有时尝试会失败，所以你必须能够处理这种失败情况。公用缓冲区不仅与稀有的物理内存页相关，而且还与其它设备争用映射寄存器。由于这两个原因，你应该明智地使用公用缓冲区。

关于公用缓冲区的另一个注意事项源自这个事实，内存管理器必定给你一个或多个整页内存。分配一个仅有几字节长的公用缓冲区是浪费内存，应该避免。另外，分配几页实际上并不需要物理上连续的内存同样是浪费。因此，正如 DDK 建议的那样，最好为不必连续的多个小块做多次请求。

释放公用缓冲区

通常应该在 **StopDevice** 例程中，在删除适配器对象之前释放被公用缓冲区占用的内存：

```
(*pdx->AdapterObject->DmaOperations->FreeCommonBuffer)(pdx->AdapterObject,
                                                               <length>,
                                                               pdx->paCommonBuffer,
                                                               pdx->vaCommonBuffer,
                                                               FALSE);
```

FreeCommonBuffer 的第二个参数是与分配缓冲区时使用的相同的长度值。最后一个参数指出该段内存是否被 CPU 高速缓存，它同样与 **AllocateCommonBuffer** 调用中的最后一个参数相同。

总线主控设备的一个例子

光盘上的 **PKTDMA** 例子演示了如何使用 AMCC S5933 芯片执行无分散/聚集的总线主控 DMA 操作。我已经详细讨论了该驱动程序如何在 **StartDevice** 中初始化设备，以及它如何在 **StartIo** 中初始化一个 DMA 传输。我还讨论了该驱动程序的 **AdapterControl** 和 **DpcForIsr** 例程中发生的几乎全部的事情。我以前曾指出，这些例程会含有一些为了启动设备上的操作而存在的与设备相关的代码；因此我写了一个名为 **StartTransfer** 的辅助函数：

```
VOID StartTransfer(PDEVICE_EXTENSION pdx, PHYSICAL_ADDRESS address, BOOLEAN isread)
{
    ULONG mcsr = READ_PORT ULONG((PULONG)(pdx->portbase + MCSR));
    ULONG intcsr = READ_PORT ULONG((PULONG)(pdx->portbase + INTCSR));
    if (isread)
    {
        mcsr |= MCSR_WRITE_NEED4 | MCSR_WRITE_ENABLE;
        intcsr |= INTCSR_WTCI_ENABLE;
        WRITE_PORT ULONG((PULONG)(pdx->portbase + MWTC), pdx->xfer);
        WRITE_PORT ULONG((PULONG)(pdx->portbase + MWAR), address.LowPart);
    }
    else
    {
        mcsr |= MCSR_READ_NEED4 | MCSR_READ_ENABLE;
        intcsr |= INTCSR_RTCI_ENABLE;
        WRITE_PORT ULONG((PULONG)(pdx->portbase + MRTC), pdx->xfer);
        WRITE_PORT ULONG((PULONG)(pdx->portbase + MRAR), address.LowPart);
    }
}
```

```

    WRITE_PORT ULONG((PULONG)(pdx->portbase + INTCSR), intcsr);
    WRITE_PORT ULONG((PULONG)(pdx->portbase + MCSR), mcsr);
}

```

这个例程为执行一个 DMA 传输设置了 S5933 操作寄存器，然后启动传输。过程如下：

1. 设置地址寄存器(**MxAR**)和传输计数寄存器(**MxTC**)与数据流方向相适应。AMCC 使用术语“读”来描述数据从内存流向设备的操作。因此，当我们实现 **IRP_MJ_WRITE** 时，我们在芯片级上安排了一个读操作。我们使用的地址是由 **MapTransfer** 返回的逻辑地址。
2. 当传输计数到达 0 时，通过写入 **INTCSR** 寄存器允许一个中断。
3. 设置 **MCSR** 寄存器中的传输允许位以启动传输。

实际上，S5933 有同时进行 DMA 读操作和 DMA 写操作的能力，但上面代码没有明显地表现这一点。我写的 **PKTDMA** 一次仅能执行一个操作(或者读或者写)。为了实现允许同时发生两种操作的驱动程序，你需要，一、实现分离的读写 **IRP** 队列，二、创建两个设备对象和两个适配器对象(一对用于读，一对用于写)，这样就可以顺利地在 **AllocateAdapterChannel** 中多次排队相同的对象。我认为把额外的复杂代码加入到例子中会使你感到迷惑。

在**PKTDMA**中处理中断

PCI42 包含了一个中断例程，它做了一些移动数据的工作。**PKTDMA** 的中断例程更简单：

```

BOOLEAN OnInterrupt(PKINTERRUPT InterruptObject, PDEVICE_EXTENSION pdx)
{
    ULONG intcsr = READ_PORT ULONG((PULONG) (pdx->portbase + INTCSR));
    if (!(intcsr & INTCSR_INTERRUPT_PENDING))
        return FALSE;

    ULONG mcsr = READ_PORT ULONG((PULONG) (pdx->portbase + MCSR));
    WRITE_PORT ULONG((PULONG) (pdx->portbase + MCSR), mcsr & ~(MCSR_WRITE_ENABLE |
MCSR_READ_ENABLE));<-1

    intcsr &= ~(INTCSR_WTCI_ENABLE | INTCSR_WTCI_ENABLE);

    BOOLEAN dpc = GetCurrentIrp(&pdx->dqReadWrite) != NULL;

    while (intcsr & INTCSR_INTERRUPT_PENDING)
    {
        InterlockedOr(&pdx->intcsr, intcsr);
        WRITE_PORT ULONG((PULONG) (pdx->portbase + INTCSR), intcsr);
        intcsr = READ_PORT ULONG((PULONG) (pdx->portbase + INTCSR));
    }

    if (dpc)
        IoRequestDpc(pdx->DeviceObject, NULL, NULL);

    return TRUE;
}

```

我仅讨论该 ISR 与 **PCI42** 中的 ISR 的不同之处：

1. 一旦设置了 **MCSR** 中的允许位，S5933 就开始传输数据。该语句清除了这两个读写允许位。如果你的驱动程序正处理并发的读写操作，你需要测试 **INTCSR** 中的中断标志来确定刚刚完成的是哪种操作。
2. 我们立即向 **INTCSR** 回写以清除中断。这个语句还禁止了“传输至 0”中断以防止再次发生中断。但能处理并发读写操作的驱动程序仅应该禁止刚发生实际中断。

3. **InterlockedOr** 是我写的一个辅助例程，这样我就不必担心在累加中断标志上与 **DpcForIsr** 竞争。

测试PKTDMA

如果你有 S5933DK1 开发板，你就可以测试 PKTDMA 程序。如果你运行过 PCI42 测试，你应该已经安装了 ISA 接口卡的驱动程序 S5933DK1.SYS。如果没有做过 PCI42 测试，你需要为这个测试安装 S5933DK1.SYS。然后安装 PKTDMA.SYS，它作为 S5933 开发板本身的驱动程序。最后运行 PKTDMA\TEST\DEBUG 目录中的 TEST.EXE 程序。TEST 程序向 PKTDMA 发出一个 8192 字节的写操作。它还向 S5933DK1 发出一个 **DeviceIoControl** 命令从 ISA 卡上读回这些数据，并校验读出的内容。

第八章：电源管理

电源是计算的必要条件，但是直到最近，个人计算机才实现高效地管理其电源。由于三个原因，更高效的电源管理是十分重要的。第一，作为一个生态学问题，减少电力消耗可以最小化计算对环境的影响。不仅计算机需要电力，计算机所在房屋的空调系统也需要电力。第二，许多旅行者需要良好的电源管理：电池技术的发展不能跟上各种移动计算的需求。并且更多的消费者认为 PC 应该是具有电源管理能力的家用器具。现在的机器工作时风扇和磁盘都会发出噪音，并且其接通电源后的启动时间过长。减少上电启动的延迟和消除噪音(这也意味着最小化电源消耗以减轻系统的冷却需求)对于 PC 能成为大众消费品是必须的。

在本章中，我将讨论在 Windows 2000 和 Windows 98 系统中，WDM 驱动程序在电源管理方面所扮演的角色。本章的第一节是“WDM 电源管理模型”，陈述了一些需要了解的总体概念。第二节“管理电源状态转换”是本章的主要部分：我将描述一个典型功能驱动程序应该执行的非常复杂的任务。本章的最后，我将讨论 WDM 功能驱动程序与电源管理有关的其它责任。

- WDM 电源管理模型
- 管理电源状态转换
- 其它电源管理细节
- Windows 98 兼容问题

WDM电源管理模型

在 Windows 2000 和 Windows 98 中，操作系统接管了大部分电源管理工作。当然，这是因为只有操作系统才能真正了解电源管理的内部过程。例如，系统 BIOS 负责的电源管理不能区分应用程序使用的屏幕和屏幕保护程序使用的屏幕之间的区别。但操作系统可以区分开这种不同，从而确定是否可以关闭显示器。

作为计算机全局电源策略，操作系统支持一些用户接口元素，用户可以通过这些接口元素控制最终的电源管理策略。这些用户接口元素包括控制面板、开始菜单上的命令、控制设备唤醒特征的 API。通过向设备发送 IRP，内核的电源管理部件实现了操作系统的电源策略。WDM 驱动程序主要是作为响应这些 IRP 的被动角色。

WDM驱动程序的角色

设备的某个驱动程序需要充当设备电源策略的管理者。通常都是由功能驱动程序充当这个角色。电源管理器可以改变整个系统的电源状态。功能驱动程序接收电源管理器发来的 IRP(系统 IRP)，作为设备电源策略的管理者，功能驱动程序用设备理解的术语翻译这些 IRP 并引发新的 IRP(设备 IRP)。当响应设备 IRP 时，功能驱动程序需要关心设备专有的细节。设备硬件会有自己的上下文信息，不应该在设备处于低电源期间丢失这些信息。例如，键盘驱动程序会保存锁定键(如 CAPS-LOCK、NUM-LOCK、SCROLL-LOCK)、LED 等信息。功能驱动程序有责任保存并恢复这些上下文信息。某些设备带有唤醒特征，当外部事件发生时，这些设备可以唤醒系统；功能驱动程序应与用户协同工作以确保唤醒特征在需要时有效。许多功能驱动程序还管理含有大量 IRP 的队列(设备读写 IRP)，因此当设备电源状态转变时需要停止或释放这些队列。

处于设备堆栈底端的总线驱动程序有责任控制设备的电流和执行任何与设备唤醒特征相关的电气步骤。过滤器驱动程序通常作为电源管理 IRP 通过的管道，它们用专用的协议向下层驱动程序传递电源管理请求。

设备电源状态与系统电源状态

WDM 模型使用与 ACPI(Advanced Configuration and Power Interface)规范(见 <http://www.teleport.com/~acpi/spec.htm>)相同的术语来描述电源状态。设备能呈现图 8-1 所描述的四种电源状态。在 D0 状态中，设备处于全供电状态。在 D3 状态中，设备处于无供电(或最小限度的电流)状态。中间的 D1 和 D2 状态指出设备的两个不同睡眠状态。随着设备从 D0 状态变化到 D3 状态，设备将消耗越来越少的电力，同时需要保留的当前状态上下文信息也越来越少。而设备再转变回 D0 状态的延迟期则相应增加。

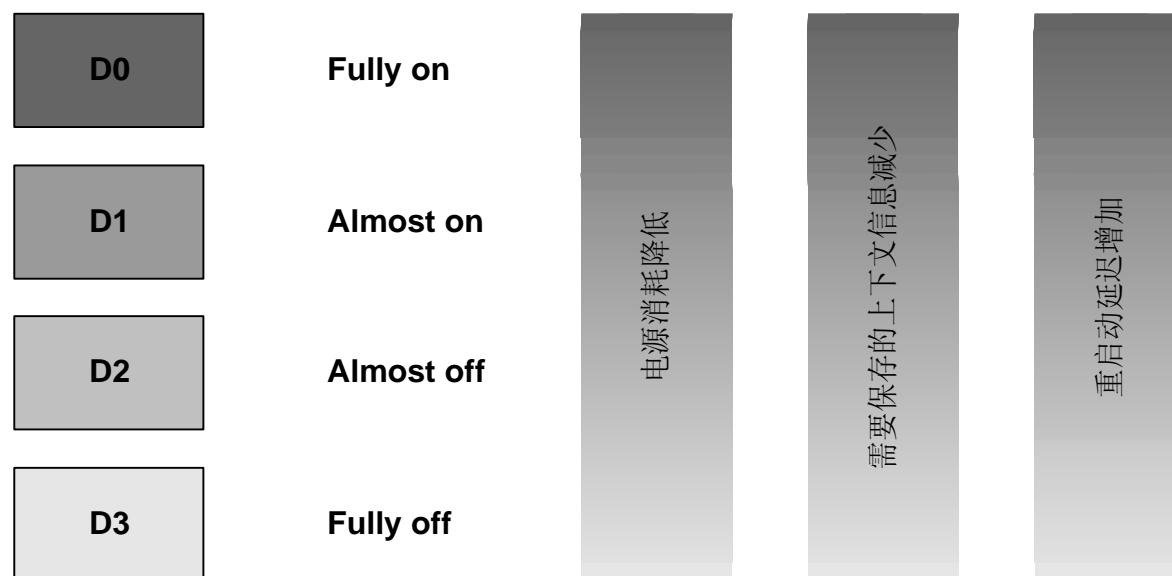


图 8-1. ACPI 电源状态

Microsoft 规定了不同类型设备的类专用电源需求。这个需求规范可以在 <http://www.microsoft.com/hwdev/specs/PMref/> 找到。例如，这个规范要求每个设备至少要支持 D0 和 D3 两个状态。

输入设备(键盘、鼠标等)还应该支持D1 状态。Modem设备需要另外支持D2 状态。设备类上的这些不同规定可能来源于设备的用途和工业上的实践。

操作系统不直接处理设备的电源状态，由设备驱动程序专门处理。系统使用一组与 ACPI 设备状态类似的系统电源状态来控制电源，见图 8-2。Working 状态是全供电状态，计算机可以实现全部功能。程序仅能在 Working 状态下执行。

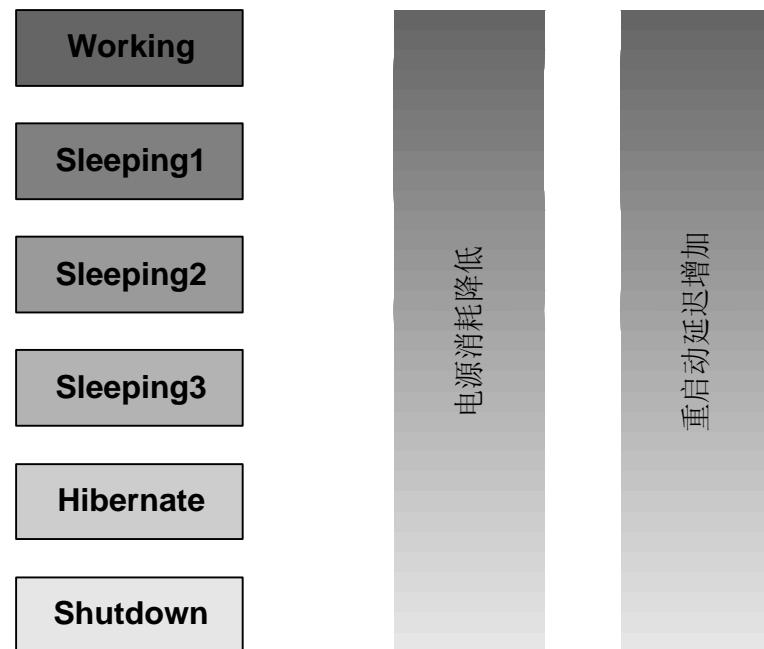


图 8-2. 系统电源状态

其它系统电源状态对应更小的电力需求配置，在这些系统电源状态中，计算机不能执行任何指令。Shutdown 状态就是电源关闭状态。Hibernate 状态是另一种 Shutdown 状态，它把计算机的整个状态都记录到硬盘上，因此在电源恢复供电时可以使计算机快速恢复到记录前的状态。在 Hibernate 和 Working 状态之间是三个有不同电力消耗级别的中间状态。

电源状态转换

系统初始化后即进入 Working 状态。大部分设备也以 D0 状态启动，但某些设备的驱动程序会在设备启动时使设备进入低电源消耗状态。在系统启动并正常运行后，这些设备的驱动程序才使设备进入一个稳定的状态，在这个状态中，系统电源处于 Working 状态，而设备处于的状态取决于具体活动和设备自身的能力。

用户的活动或外部事件会导致电源状态的改变。一个常见的电源状态转换情景是用户在开始菜单上选择“关闭系统”中的“standby”选项，使计算机进入等待状态。在响应这个命令过程中，电源管理器首先向每个驱动程序发送带有 IRP_MN_QUERY_POWER 副功能码的 IRP_MJ_POWER 请求以询问设备能否接受即将到来的电源关闭请求。如果所有驱动程序都同意，电源管理器将发送第二个带有 IRP_MN_SET_POWER 副功能码的电源管理 IRP，然后驱动程序把其设备置入低电源状态以响应这个 IRP。如果有任何一个驱动程序否决了这个查询，电源管理器仍旧发出这个 IRP_MN_SET_POWER 请求，但它用原来的电源级别换成了请求的电源级别。

系统并不总是发送 IRP_MN_QUERY_POWER 请求。某些事件(如电池电力将要耗尽)必须被无条件接受，并且操作系统也不再发出查询请求。如果查询发出后，并且驱动程序也接受了请求的电源状态，那么驱动程序将不再启动任何会妨碍未来电源状态设置请求的操作。例如，磁带机驱动程序在使一个进入低电源状态的查询请求成功返回前先确保当前没有执行备份操作。另外，该驱动程序还拒绝任何后来的备份命令，除非是另一个电源状态设置请求。

处理IRP_MJ_POWER请求

电源管理器与驱动程序沟通使用 IRP_MJ_POWER 类型的 I/O 请求包。表 8-1 列出了当前可以使用的四个副功能码。

表 8-1. *IRP_MJ_POWER* 的副功能码

副功能码	描述
IRP_MN_QUERY_POWER	确定预期的电源状态改变是否安全
IRP_MN_SET_POWER	命令驱动程序改变电源状态
IRP_MN_WAIT_WAKE	命令总线驱动程序使用唤醒特征；使功能驱动程序能了解唤醒信号何时发生
IRP_MN_POWER_SEQUENCE	为上下文保存和恢复提供优化

IO_STACK_LOCATION 的 **Parameters** 联合中的 **Power** 子结构有四个参数描述了电源管理请求，但大多数 WDM 驱动程序仅对其中两个参数感兴趣。见表 8-2。

表 8-2. *IO_STACK_LOCATION* 的 *Parameters.Power* 子结构中的各个域

域名	描述
SystemContext	电源管理器内部使用的上下文值
Type	DevicePowerState 或 SystemPowerState (POWER_STATE_TYPE 类型的枚举值)
State	电源状态，可为 DEVICE_POWER_STATE 或 SYSTEM_POWER_STATE
ShutdownType	指出转换到 PowerSystemShutdown 状态的原因代码

所有驱动程序，包括过滤器驱动程序和功能驱动程序通常都向其下面的驱动程序传递电源管理请求。唯一的例外是 **IRP_MN_QUERY_POWER** 请求。

控制如何把电源管理请求传递到低级驱动程序有特殊的规则。图 8-3 显示了三种可能的处理过程。第一，在释放一个电源管理请求的控制之前，你必须调用 **PoStartNextPowerIrp**，即使你以错误状态完成该 IRP，也要这样做。做这个调用的原因是，电源管理器自己需要维持一个电源管理请求队列，所以必须通知它确实可以出队一个请求，以及向设备发送下一个请求。除了调用 **PoStartNextPowerIrp**，你还必须调用专用例程 **PoCallDriver**(代替 **IoCallDriver**)来向下层驱动程序发送请求。

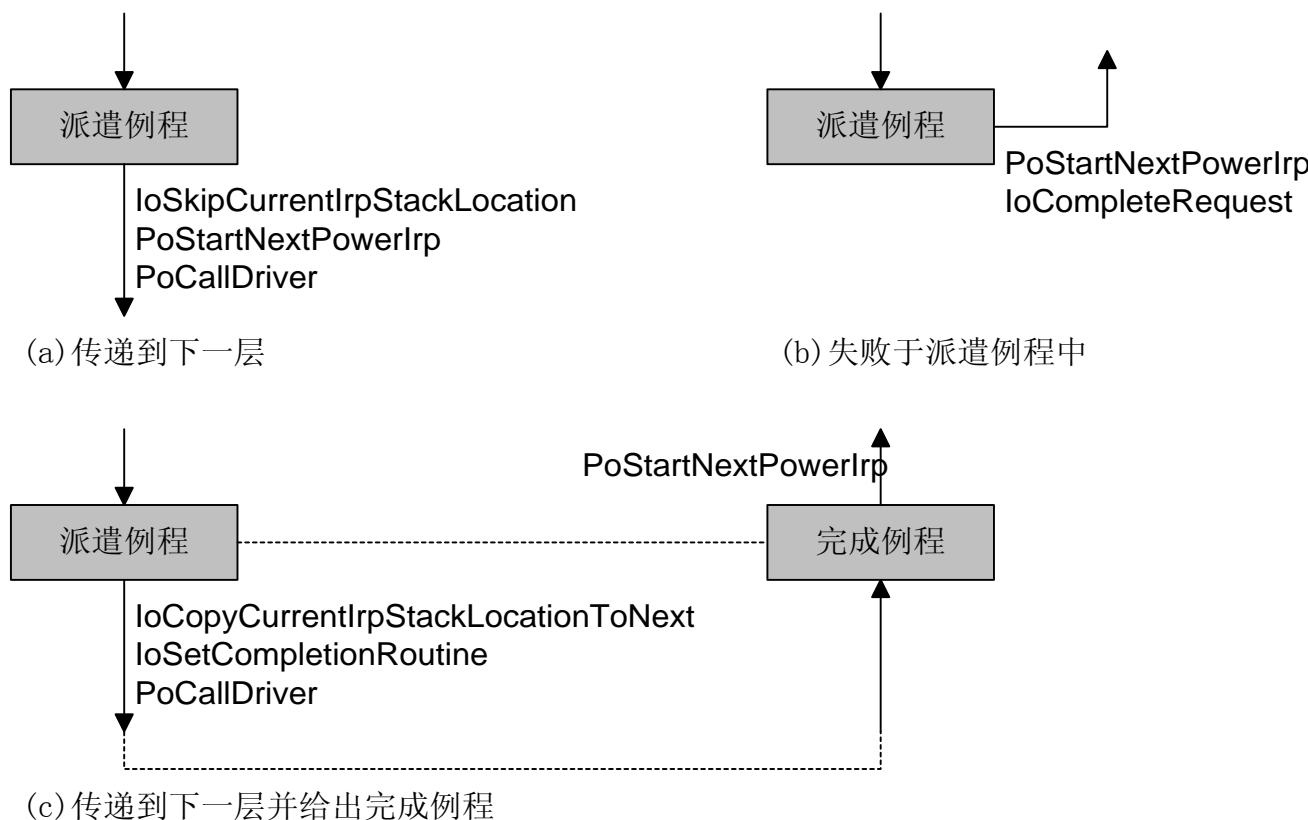


图 8-3. 处理 *IRP_MJ_POWER* 请求

注意

电源管理器需要为每个设备维护两个电源管理 IRP 队列。一个是系统电源管理 IRP 队列(如指定系统电源状态的 IRP_MN_SET_POWER 请求), 另一个是设备电源管理 IRP 队列(如指定设备电源状态的 IRP_MN_SET_POWER 请求)。这两种 IRP 可以同时处于活动状态。驱动程序还可以同时处理 PnP 请求和其它 IRP。

下面函数显示了一个电源管理请求沿设备堆栈被向下传递的过程:

```
NTSTATUS DefaultPowerHandler(IN PDEVICE_OBJECT fdo, IN PIRP Irp)
{
    PoStartNextPowerIrp(Irp); <
    IoSkipCurrentIrpStackLocation(Irp); <
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    return PoCallDriver(pdx->LowerDeviceObject, Irp); <-3
}
```

1. **PoStartNextPowerIrp** 通知电源管理器可以出队并发送下一个电源管理 IRP。你必须在每收到一个电源管理 IRP 后都做这个调用。即该调用或者发生在你的派遣例程中, 在你把请求发送给 **PoCallDriver** 之前, 或者发生在完成例程中。
2. 由于 **PoCallDriver** 会立即把 IRP 的堆栈单元指针前进一步, 所以我们使用 **IoSkipCurrentIrpStackLocation** 函数使 IRP 的堆栈单元指针倒退一步。这与我们在向下传递一个请求并忽略其以后的处理中使用的技术相同。
3. 我们使用 **PoCallDriver** 函数继续传递电源管理请求。Microsoft 在实现这个函数上使用了在 **IoCallDriver** 函数加入一些条件逻辑的办法来处理电源管理, 但这对性能带来了微小但可察觉的冲击。

功能驱动程序使用两个步骤来传递 IRP 并执行其设备相关的动作, 如图 8-4 所示: 当进入更低的电源状态时, 它先执行设备相关的步骤, 然后向下传递请求。当回到更高的电源状态时, 它先向下传递请求, 然后在完成例程中执行设备相关的步骤。这个巧妙的嵌套操作保证了当驱动程序操作硬件时, 硬件一直处于供电状态。

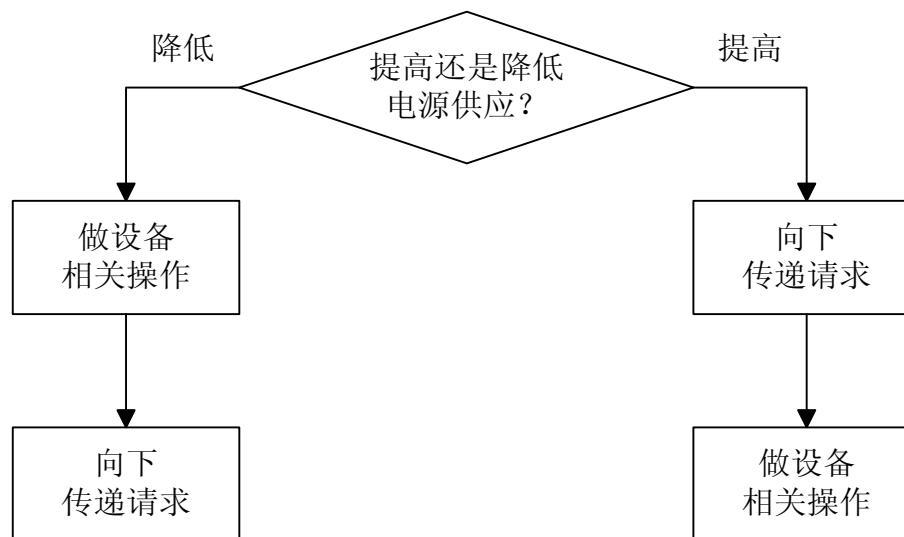


图 8-4. 处理系统电源管理请求

电源管理 IRP 到来时, 你处于一个系统线程的上下文中, 不要阻塞这个线程。如果你的设备有 INRUSH 特征, 或者你清除了设备对象中的 DO_POWER_PAGABLE 标志, 那么电源管理器将在 DISPATCH_LEVEL 级上向你发送 IRP。而当你执行在 DISPATCH_LEVEL 级上时, 不可能阻塞一个线程。如果你设置了 DO_POWER_PAGABLE 标志, 你会在 PASSIVE_LEVEL 级上收到电源管理 IRP, 此时, 如果你在服务一个系统 IRP 时请求了设备电源管理 IRP 然后阻塞, 将导致死锁: 电源管理器不会向你发送设备 IRP, 除非你的系统 IRP 派遣例程返回, 因此你将永远等待下去。

功能驱动程序在处理某些电源管理请求时会执行一些步骤以获得完成时间。DDK 指出, 你可以在用户不易察觉的范围内延迟电源管理 IRP 的完成, 但是能够延迟并不意味着能够阻塞。在这些操作完成时, 不能阻塞意味着需要使用完成例程来使这些步骤异步。

对 IRP_MN_QUERY_POWER 查询你可以回答“**Yes**”或“**No**”。即你可以用该副功能码失败该 IRP。失败该 IRP 就是说“**No**”。但对于 IRP_MN_SET_POWER 请求却没有这样的自由：你必须执行它携带的指令。

管理电源状态转换

正确地执行电源管理任务需要非常精确的编码，需要面对许多复杂的因素。例如，设备可以有唤醒系统的能力。决定成功还是失败一个查询，决定由哪个设备电源状态来对应新系统电源状态，这些都取决于设备当前是否使用了唤醒特征。你可以在没有任何活动时关闭设备的电源供应，当有 IRP 到来时再恢复设备的电源供应。也许你的设备是一个“INRUSH”设备，这种设备在上电时需要大的电流，因此电源管理器需要特殊对待这种设备。

当我思考用传统方法解决 query-power 和 set-power 操作时，即用通常的派遣例程和完成例程，我对需要许多全然不同的但最终都做类似工作的子例程感到沮丧。因此我决定围绕一个有限状态机来建立我自己的电源管理程序，有限状态机可以方便地处理异步活动。

我将在 GENERIC.SYS 中解释有限状态机，GENERIC.SYS 是随书光盘中大部分例子使用的支持驱动程序。附录 B 的“使用 GENERIC.SYS”中详细描述了 GENERIC.SYS 的客户接口。GENERIC.SYS 等价于一个包含有 WDM 驱动程序可使用的辅助例程的内核模式 DLL。你可以认为它是一个有广泛适用性的通用类驱动程序。客户驱动程序，包括大部分我自己的例子驱动程序，通过调用 **GenericDispatchPower** 把电源管理 IRP 委托给 GENERIC.SYS 处理。GENERIC.SYS 还实现了我在第六章中讨论过的 DEVQUEUE 对象。

有限状态机概述

我写了一个名为 HandlePowerEvent 的函数来实现管理电源 IRP 的有限状态机。该函数有两个参数：

```
NTSTATUS HandlePowerEvent(PPOWCONTEXT ctx, enum POWEREVENT event);
```

第一个参数是一个包含状态变量的上下文结构：

```
typedef struct _POWCONTEXT {
    LONG id;
    LONG eventcount;
    PGENERIC_EXTENSION pdx;
    PIRP irp;
    enum POWSTATE state;
    NTSTATUS status;
    PKEVENT pev;
    DEVICE_POWER_STATE devstate;
    UCHAR MinorFunction;
    BOOLEAN UnstallQueue;
} POWCONTEXT, *PPOWCONTEXT
```

id 和 **eventcount** 域用于调试。如果你在宏 VERBOSETRACE 定义为非零值时编译 GENERIC 工程中的 POWER.CPP，则 POWTRACE 宏将产生跟踪消息的量值。我使用这个特征来调试有限状态机。光盘上的 GENERIC.SYS 为预创建版本，没有使用 VERBOSETRACE 宏来消去跟踪消息。

pdx 成员为给定设备指出 GENERIC 的设备扩展部分。设备扩展中有两个成员与电源管理有关，我将在后面的“新 IRP 的初始化处理”中再讨论它们。**irp** 成员指向有限状态机当前工作的电源 IRP；**state** 是有限状态机的状态变量。**status** 成员代表 IRP 的结束状态。在某些情况下，我们希望在 HandlePowerEvent 发出和完成一个设备电源 IRP 时等待；所以我们用 **pev** 来指向等待完成的对象。**devstate** 成员保存设备的电源状态，我们将在设备 IRP 中使用这个状态值，**MinorFunction** 保存着我们希望在 IRP 中使用的副功能码(IRP_MN_QUERY_POWER 或 IRP_MN_SET_POWER)。最后，**UnstallQueue** 指出我们是否希望有限状态机在完成处理当前电源管理 IRP 时打开 IRP 队列。

HandlePowerEvent 函数的第二个参数是一个事件代码，指出我们为什么要调用该函数。有三种事件代码：

- **NewIrp** 指出我们正在向有限状态机提交一个新的电源 IRP。那个上下文结构中的 **irp** 成员指向这个 IRP。
- **MainIrpComplete** 指出 IRP 被完成。
- **AsyncNotify** 指出有其它异步活动发生。

HandlePowerEvent 使用了状态变量的值和事件代码来决定要采取的动作，见表 8-3。（表中空的单元指出一个不可能的情况，这种情况会使 **checked** 版本的 **GENERIC.SYS** 产生 **ASSERT** 错误）一个动作对应于电源 IRP 在其处理路径上的一系列程序步骤。

表 8-3. 下表给出了每个事件和状态的初始动作

状态		事件	
	NewIrp	MainIrpComplete	AsyncNotify
InitialState	TriageNewIrp		
SysPowerUpPending		SysPowerUpComplete	
SubPowerUpPending			SubPowerUpComplete
SubPowerDownPending			SubPowerDownComplete
SysPowerDownPending		SysPowerDownComplete	
DevPowerUpPending		DevPowerUpComplete	
DevPowerDownPending		CompleteMainIrp	
ContextSavePending			ContextSaveComplete
ContextRestorePending			ContextRestoreComplete
DevQueryUpPending		DevQueryUpComplete	
DevQueryDownPending		DevQueryDownComplete	
QueueStallPending			QueueStallComplete
FinalState			

因为许多事件在某些情况下需要多个动作，因此 **HandlePowerEvent** 代码看起来比较特别：

```
NTSTATUS HandlePowerEvent(...)  
{  
    NTSTATUS status;  
    POWACTION action = ...;  
    while (TRUE)  
    {  
        switch (action)  
        {  
            case <someaction>:  
                action = <someotheraction>;  
                continue;  
            case <anotheraction>:  
                break;  
        }  
        break;  
    }  
    return status;  
}
```

该函数有一个无限循环，循环中对 **action** 代码进行 **switch** 判断。带有 **continue** 语句的 **case** 块重复这个循环；这也是我在一个函数调用中把一系列 **action** 串起来的原因。带有 **break** 的 **case** 块使程序从 **switch** 块跳到另一个 **break** 语句，这个 **break** 语句将结束那个无限循环，从而该函数可以返回到调用者。

对有限状态机采用这种编码风格是因为我对以前熟悉的结构化编程规则仍念念不忘。我希望该函数仅有一个 `return` 语句，可以更易于证明该函数工作正常。为了辅助这个证明，我自己推导出三个原则，可以在函数的结尾处通过检查或使用 `ASSERT` 语句来测试。下面就是这些规则：

- 每个代码路径最终都会跳到 `break` 语句，从而到达 `return` 语句。在代码执行过程中，必须有人修改 `status` 变量和 `state` 变量。
- 任何 `continue` 语句之前都应该先修改 `action` 变量。
- 任何会导致递归调用 `HandlePowerEvent` 函数的 `case` 块，例如调用了 `PoCallDriver` 函数，必须立即从循环中 `break` 出去，并且不要碰上下文结构或 IRP。

新IRP的初始化处理

当我们收到一个新的 `query-power` 或 `set-power` 请求时，我们创建一个上下文结构来驱动有限状态机并调用 `HandlePowerEvent`：

```
NTSTATUS GenericDispatchPower(PGENERIC_EXTENSION pdx, PIRP Irp)
{
    NTSTATUS status = IoAcquireRemoveLock(pdx->RemoveLock, Irp);
    if (!NT_SUCCESS(status))
        return CompleteRequest(Irp, status);

    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    ULONG fcn = stack->MinorFunction;
    if (fcn == IRP_MN_SET_POWER || fcn == IRP_MN_QUERY_POWER)
    {
        PPOWCONTEXT ctx = (PPOWCONTEXT) ExAllocatePool(NonPagedPool, sizeof(POWCONTEXT));
        RtlZeroMemory(ctx, sizeof(POWCONTEXT));
        ctx->pdx = pdx;
        ctx->irp = Irp;
        status = HandlePowerEvent(ctx, NewIrp);
    }
    IoReleaseRemoveLock(pdx->RemoveLock, Irp);
    return status;
}
```

1. 客户驱动程序提供了一个删除锁，它和 `GENERIC` 共同使用这个锁来防止提前删除设备对象。`GENERIC` 中的实际代码要比我在这里给出的稍复杂一些，删除锁并不是必须的。所以实际代码在使用删除锁前先测试 `RemoveLock` 指针是否为 `NULL`。此外还有一些不太重要的考虑，包括错误检测。
2. 对于设置和查询操作，我们先为上下文结构分配一块非分页内存并初始化。`state` 变量初始化为 `InitialState`，其数值上等于 0。

有限状态机的初始状态是 `InitialState`。当我们为 `NewIrp` 事件调用 `HandlePowerEvent` 函数时，发生的第一动作是 `TriageNewIrp`：

```
case TriageNewIrp:
{
    status = STATUS_PENDING;
    IoMarkIrpPending(Irp);
    IoAcquireRemoveLock(pdx->RemoveLock, Irp);
    if (stack->Parameters.Power.Type == SystemPowerState)
    {
        // system IRP
        if (stack->Parameters.Power.State.SystemState < pdx->systime)
            <--3
        {
            action = ForwardMainIrp;
            ctx->state = SysPowerUpPending;
        }
    }
}
```

```

else
{
    action = SelectDState;
    ctx->state = SubPowerDownPending;
}
// system IRP
else
{
    // device IRP
    ctx->state = QueueStallPending;
    if (!pdx->StalledForPower)
    {
        ctx->UnstallQueue = TRUE;
        pdx->StalledForPower = TRUE;
        NTSTATUS qstatus = StallRequestsAndNotify(pdx->dqReadWrite,
                                                GenericSaveRestoreComplete,
                                                ctx);
        if (qstatus == STATUS_PENDING)
            break;
    }
    action = QueueStallComplete;
}
// device IRP
continue;
}

```

1. 我们总是挂起送到我们的电源管理 IRP。几乎在每种情况下，我们都需要延迟 IRP 的完成直到某个异步活动发生之后。
2. 我们多一次获取了删除锁，跨越了派遣例程中的对它的获取。我们将在最后完成该 IRP 时释放这个锁。
3. 如果 IRP 中的电源状态在数值上小于设备扩展中的 **syspower** 值，则该 IRP 代表一个更高级的系统电源状态。
4. 该语句演示了 **HandlePowerEvent** 如何在一个调用中执行多个动作。后面我们将执行一个 **continue** 语句来重复这个无限循环。但 **action** 值会不同，这导致了另一段代码的执行。
5. 该语句演示了 **action** 的 **case** 语句是如何改变了有限状态机的状态。为了简化用于调试打印语句的条件编译代码，**GENERIC** 中的实际代码使用了一个名为 **SETSTATE** 的宏来执行这个赋值。
6. 我们将要调用一个函数(**StallRequestsAndNotify**)，这个函数会造成对当前函数的递归调用。因为我们以后将不被允许访问上下文结构，所以我们现在设置这个标志。该标志意味着 **CompleteMainIrp** 函数应该调用 **RestartRequests** 来重启动队列。
7. 该语句显示了一个 **action case** 语句如何能使 **HandlePowerEvent** 返回。这个 **break** 语句使程序从 **switch** 块中退出。紧跟这个 **switch** 块之后的语句是另一个 **break** 语句，它使程序退出 **while** 循环。

基本上，**TriageNewIrp** 区别对待各种电源 IRP(IRP 的类型为 **SystemPowerState**)，包括提高电源级别的系统电源 IRP、降低电源级别的系统电源 IRP、设备电源 IRP(IRP 的类型为 **DevicePowerState**)，但不考虑设备电源级别是升高还是降低。在这个阶段中，有限状态机不能区别 **QUERY_POWER** 和 **SET_POWER** 请求，所以在此它们被视为非常类似的 IRP。

不管电源是升还是降，我们的设备扩展都需要两个变量来保存系统电源状态和设备电源的状态：

```

typedef struct _GENERIC_EXTENSION {
    ...
    DEVICE_POWER_STATE devpower; // current dev power state
    SYSTEM_POWER_STATE syspower; // current sys power state
} GENERIC_EXTENSION, *PGENERIC_EXTENSION;

```

当客户驱动程序第一次寄存 **GENERIC.SYS** 时，我们初始化这两个值为 **PowerDeviceD0** 和 **PowerSystemWorking**。

从上下文中你可以猜到，设备扩展还有一个名为 **StalledForPower** 的布尔成员。当这个标志被设置时，表明由于电源管理的需要，IRP 队列当前已经停止。另外，你会注意到我们没有明确地同步访问电源状态域或这个标志。由于电源管理器已经强制序列化访问这些数据，所以没有必要同步。

现在我将分别讨论三个 IRP 初始类。

提升电源级别的系统电源IRP

如果一个系统电源 IRP 表明要提升系统的电源级别，你应立即把它传递给下一层驱动程序。而在你为该系统电源 IRP 准备的完成例程中，你应该请求对应的设备电源 IRP 并返回 STATUS_MORE_PROCESSING_REQUIRED，以临时停止完成处理。在设备电源 IRP 的完成例程中，你应该完成系统电源 IRP 的完成处理。图 8-5 显示了穿过所有驱动程序的 IRP 流。图 8-6 是一个状态图，显示了我们的有限状态机如何处理这些 IRP。

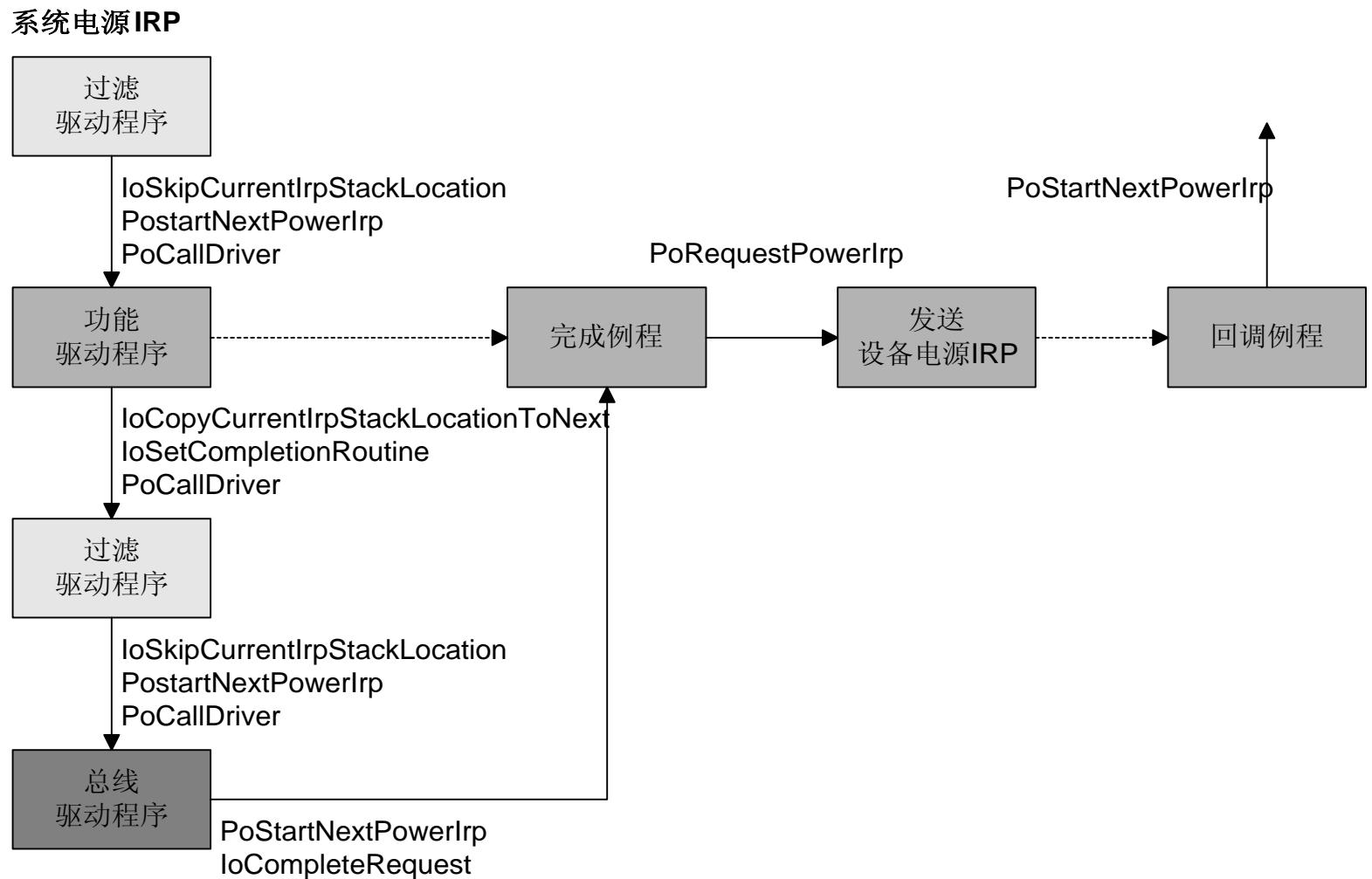


图 8-5. 增加系统电源级别的 IRP 流

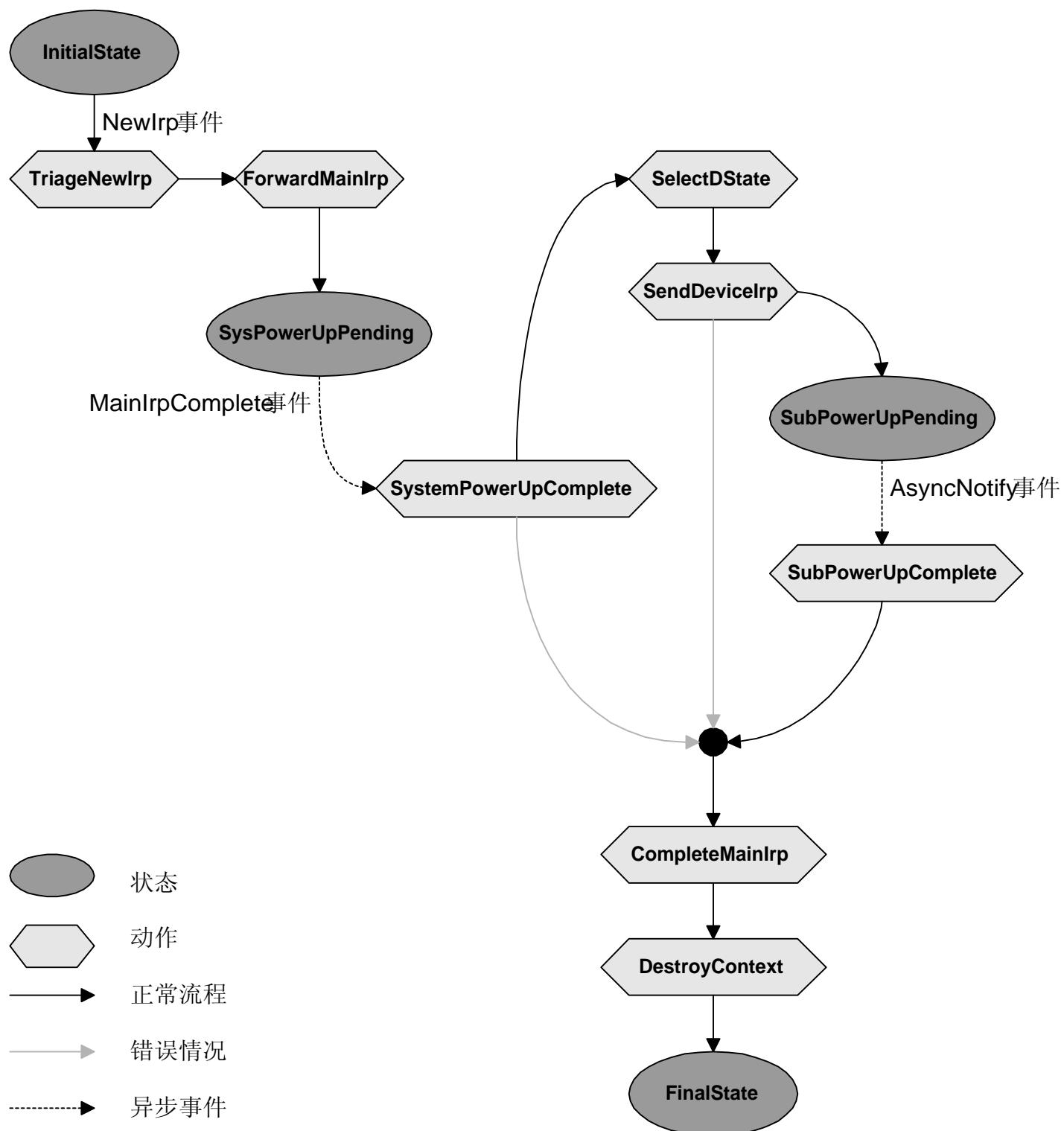


图 8-6. 增加系统电源时的状态转换

根据代码的操作过程，TriageNewIrp 函数把有限状态机置入 **SysPowerUpPending** 状态，并请求 **ForwardMainIrp** 动作，下面是这个动作的代码：

```

case ForwardMainIrp:
{
    IoCopyCurrentIrpStackLocationToNext(Irp);
    IoSetCompletionRoutine(Irp,
        (PIO_COMPLETION_ROUTINE) MainCompletionRoutine,
        (PVOID) ctx,
        TRUE,
        TRUE,
        TRUE);
    PoCallDriver(pdx->LowerDeviceObject, Irp);
    break;
}

```

`HandlePowerEvent` 将返回 `STATUS_PENDING`, 如 `TriageNewIrp` 中的代码所要求那样。这个返回值会被 `GenericDispatchPower` 函数过滤, 大概会是客户驱动程序中的 `IRP_MJ_POWER` 派遣函数。

我们下一次与该 `IRP` 接触是当总线驱动程序完成它时。我们自己的 **MainCompletionRoutine** 函数将作为完成处理的一部分而获得控制, 我们把 `IRP` 的结束状态保存到上下文结构的 `status` 域, 并调用有限状态机。

```
NTSTATUS MainCompletionRoutine(PDEVICE_OBJECT junk, PIRP Irp, PPOWERCONTEXT ctx)
{
    ctx->status = Irp->IoStatus.Status;
    return HandlePowerEvent(ctx, MainIrpComplete);
}
```

我们的初始动作将是 **SysPowerUpComplete**:

```
case SysPowerUpComplete:
{
    if (!NT_SUCCESS(ctx->status))
        action = CompleteMainIrp;                                <
    else
    {
        if (stack->MinorFunction == IRP_MN_SET_POWER)           <
            pdx->systime = stack->Parameters.Power.State.SystemState;
        action = SelectDState;
        ctx->state = SubPowerUpPending;
        status = STATUS_MORE_PROCESSING_REQUIRED;
    }
    continue;
}
```

1. 如果该 `IRP` 在低级驱动程序中失败, 我们将使其完成, 并且对这个电源事件不做任何更多的工作。我将在后面的“处理失败”中解释 **CompleteMainIrp** 做了什么。
2. 我们在这里记录下新的系统电源状态。当检查新系统 `IRP` 是升高还是降低电源级别时, 我们使用 **systime** 值。
3. **MainCompletionRoutine** 将调用我们, 并且当我们生成设备 `IRP` 时, 该函数将中断系统 `IRP` 的完成。因此, 我们使 `MainCompletionRoutine` 返回 `STATUS_MORE_PROCESSING_REQUIRED`。

处理失败

如果 `IRP` 失败, 我们将做 **CompleteMainIrp** 动作:

```
case CompleteMainIrp:
{
    PoStartNextPowerIrp(Irp);                                <
    if (event == MainIrpComplete)                            <
        status = ctx->status;
    else
    {
        Irp->IoStatus.Status = ctx->status;
        IoCompleteRequest(Irp, IO_NO_INCREMENT);
    }
    IoReleaseRemoveLock(pdx->RemoveLock, Irp);             <
    if (ctx->UnstallQueue)                                <-5
    {
        pdx->StalledForPower = FALSE;
        RestartRequests(pdx->dqReadWrite, pdx->DeviceObject);
    }
}
```

```
action = DestroyContext;
continue;
}
```

1. 我们必须为每个电源管理 IRP 调用 **PoStartNextPowerIrp**。
2. 如果我们进入 **MainIrpComplete** 事件的处理代码，我们的调用者必须是 **MainCompletionRoutine**，并且第一个动作例程将设置 **status** 等于 **STATUS_MORE_PROCESSING_REQUIRED** 以短接完成过程。因为我们决定要完成这个 IRP，正确的做法是返回另一个状态代码并允许完成例程按其正常方式执行。
3. 如果我们进入其它事件处理，我们需要明确地完成该 IRP。
4. 这个 **IoReleaseRemoveLock** 调用平衡我们在 **TriageNewIrp** 中的 **IoAcquireRemoveLock** 调用。
5. 我将在本章后面讨论设备 IRP 时再解释这块代码。

当处理提升电源级别的系统电源 IRP 时，有限状态机在 **MainIrpComplete** 事件后进入 **CompleteMainIrp** 状态。**CompleteMainIrp** 将安排返回错误状态，这个状态是我们最初在该 IRP 中提取的(在 **MainCompletionRoutine** 中)。这将使完成处理继续执行。这里还有其它我们仍没有讨论过的代码分支，在那些分支中，**CompleteMainIrp** 将调用 **IoCompleteRequest**。**CompleteMainIrp** 完成时会请求另一个动作：

```
case DestroyContext:
{
    if (ctx->pev)
        KeSetEvent(ctx->pev, IO_NO_INCREMENT, FALSE); <
    else
        ExFreePool(ctx); <
    break;
}
```

1. 这个分支将在 **SendDeviceSetPower** 函数调用状态机去创建并等待一个设备 IRP 时被执行。
2. 这个分支将在 **GenericDispatchPower** 调用状态机去处理一个 IRP 时被执行。

DestroyContext 是有限状态机要执行的最后一个动作。

映射系统状态为设备状态

另一种办法是让 **SysPowerUpComplete** 生成一个设备电源 IRP，该 IRP 的电源状态与系统电源状态对应。我们在 **SelectDState** 动作中把系统状态映射成设备状态：

```
case SelectDState:
{
    SYSTEM_POWER_STATE sysstate = stack->Parameters.Power.State.SystemState;
    if (sysstate == PowerSystemWorking)
        ctx->devstate = PowerDeviceD0;
    else
    {
        DEVICE_POWER_STATE maxstate = pdx->devcaps.DeviceState[sysstate];
        DEVICE_POWER_STATE minstate = pdx->WakeupEnabled ? pdx->devcaps.DeviceWake : PowerDeviceD3;
        ctx->devstate = minstate > maxstate ? minstate : maxstate;
    }
    ctx->MinorFunction = stack->MinorFunction;
    action = SendDeviceIrp;
    continue;
}
```

注意，电源管理器绝不从一个低级系统电源状态直接转换到其它系统电源状态：它总是经过 **PowerSystemWorking** 状态。因此，我编写了 **SelectDState** 来映射各种系统电源状态。

通常，我们总是把设备设置成与设备的当前活动、设备的唤醒特征、设备的能力、以及迫近的系统状态，相一致的最低的电源状态。这些因素会以相当复杂的方式相互影响。为了详细解释这些，我需要讨论以前我在第六章没有讨论的 PnP 请求：IRP_MN_QUERY_CAPABILITIES。

PnP 管理器在启动设备后(也可能是其它时间)的很短时间内向设备发出查询设备能力请求。该请求的参数是一个 DEVICE_CAPABILITIES 结构，该结构包含了与电源管理有关的几个域。因为本书仅在这里讨论这个结构，所以我将给出该结构的完整声明：

```
typedef struct _DEVICE_CAPABILITIES {
    USHORT Size;
    USHORT Version;
    ULONG DeviceD1:1;
    ULONG DeviceD2:1;
    ULONG LockSupported:1;
    ULONG EjectSupported:1;
    ULONG Removable:1;
    ULONG DockDevice:1;
    ULONG UniqueID:1;
    ULONG SilentInstall:1;
    ULONG RawDeviceOK:1;
    ULONG SurpriseRemovalOK:1;
    ULONG WakeFromD0:1;
    ULONG WakeFromD1:1;
    ULONG WakeFromD2:1;
    ULONG WakeFromD3:1;
    ULONG HardwareDisabled:1;
    ULONG NonDynamic:1;
    ULONG Reserved:16;

    ULONG Address;
    ULONG UINumber;

    DEVICE_POWER_STATE DeviceState[PowerSystemMaximum];
    SYSTEM_POWER_STATE SystemWake;
    DEVICE_POWER_STATE DeviceWake;
    ULONG D1Latency;
    ULONG D2Latency;
    ULONG D3Latency;
} DEVICE_CAPABILITIES, *PDEVICE_CAPABILITIES;
```

表 8-4 描述了该结构中与电源管理有关的各个域。

表 8-4. DEVICE_CAPABILITIES 结构中与电源管理相关的域

域	描述
DeviceState	与每个系统状态对应的可能存在的设备最高级状态数组
SystemWake	最低级的系统电源状态，在该状态中设备可以生成唤醒信号。PowerSystemUnspecified 指出设备不能唤醒系统
DeviceWake	最低级的设备电源状态，在该状态中设备可以生成唤醒信号。PowerDeviceUnspecified 指出设备不能生成唤醒信号
D1Latency	设备从 D1 切换到 D0 状态所需的最长时间(100 微秒单位)。
D2Latency	设备从 D2 切换到 D0 状态所需的最长时间(100 微秒单位)。
D3Latency	设备从 D3 切换到 D0 状态所需的最长时间(100 微秒单位)。

WakeFromD0	标志，指出当设备处于指定的状态中时，设备的系统唤醒特征是否是可操作的
WakeFromD1	同上
WakeFromD2	同上
WakeFromD3	同上

通常你需要同步处理这个查询设备能力的 IRP，即向下传递该 IRP 并等待低层完成。下传该 IRP 后，你可以通过总线驱动程序对设备的能力记录做任何需要的修改。你的子派遣例程应该象这样：

```
NTSTATUS HandleQueryCapabilities(IN PDEVICE_OBJECT fdo, IN PIRP Irp)
{
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PDEVICE_CAPABILITIES pdc = stack->Parameters.DeviceCapabilities.Capabilities;
    if (pdc->Version < 1)                                <
        return DefaultPnpHandler(fdo, Irp);
    NTSTATUS status = ForwardAndWait(fdo, Irp);
    if (NT_SUCCESS(status))
    {
        stack = IoGetCurrentIrpStackLocation(Irp);
        pdc = stack->Parameters.DeviceCapabilities.Capabilities;
        <stuff>                                         <
        pdx->devcaps = *pdc;                            <-3
    }
    return CompleteRequest(Irp, status);
}
```

1. 设备能力结构中有一个版本号成员，在当前系统中总是为 1。该结构被设计成总是向上兼容，因此你可以使用 DDK 中该结构的现在和以后版本。但如果你需要面对一个比你所用结构更旧的结构，你应该忽略这个 IRP，直接下传。
2. 在这里你可以不管总线驱动程序设置的任何能力属性。
3. 应该做一份能力结构的拷贝。我已经描述过如何在收到系统电源 IRP 时使用 **DeviceState**。有必要参考一下该结构中的其它域。

不要在下传 IRP 前修改 characteristics 结构：总线驱动程序将完全再初始化该结构。当你再次获得控制时，可以修改 **SystemWake** 和 **DeviceWake**，使它们指定一个比总线驱动程序认为的合适状态更高的电源状态。你不能把唤醒域指定为某个低级电源状态，并且如果你的设备不能唤醒系统，你就不能超越总线驱动程序的决定。如果设备是 ACPI 兼容的，则 ACPI 过滤器将基于设备的 ASL(ACPI Source Language)描述自动设置

LockSupported、**EjectSupported**、**Removable** 标志，因此你不用担心这些能力设置。

你也许会在能力处理程序的点“2”处设置 **SurpriseRemovalOK** 标志。设置这个标志将使 Windows 2000 在检测到设备被意外删除时所弹出的对话框不再出现。这对于可热拔插的 USB 或 1394 设备是正常的，因此其功能驱动程序应该设置这个标志以避免干扰用户。

回到我们对 SelectDState 的讨论上，假定我们正在处理一个 set-power 请求，该请求将把计算机从 Working 状态带入 Sleeping1 状态；因此我们将执行 SelectDState 中 if 语句的第二个分支。我们假定总线驱动程序知道设备能在系统进入 Sleeping1 状态时可进入 D0、D1、D2、D3 任何一种状态。当它回答 PnP 的能力查询时，它将把 PowerDeviceD0 填到设备能力结构中的 **DeviceState[PowerSystemSleeping1]** 中，因为 D0 是设备在该系统状态上所能达到的最高电源状态。我们先记下 PowerDeviceD0，然后作为 **maxstate** 值。

设备还可以有唤醒特征。我们以后再讨论设备的唤醒特征。总线驱动程序将设置能力结构的 **DeviceWake** 成员等于唤醒能够发生的最低电源状态。让我们假定该值为 **PowerDeviceD1**。如果我们的唤醒特征碰巧现在正允许，我们将设置 **minstate** 为 PowerDeviceD1。

如果我们没有唤醒特征，或者我们有唤醒特征但该特征当前被禁止，那么我们就可以自由选择任何低于 **maxstate** 值的设备电源状态，**maxstate** 值来自设备能力结构。我们可以选择 D3，但这不能适合各种设备类型，因为从 D3 到 D0 的恢复时间要比从 D2 到 D1 的恢复时间长。在这种情况下的选择需要取决其它一些因素。例

如,如果设备能进入 D2 状态,你可以用 D2 状态对应任何系统 sleeping 状态,而 D3 状态则对应系统的 hibernate 和 shutdown 状态。

当系统从一个 sleeping 状态恢复时,使设备停留在一个低电源状态是合理的。DDK 建议你这样做。但有两种情况在系统进入 Working 状态时,你的设备应恢复到 D0 状态。第一种情况是你的设备具有 INRUSH 特征。在这种情况下,电源管理器在设备上电前不会向其它具有 INRUSH 特征的设备发送电源 IRP。第二种情况是当你已经得到许多排队的 IRP,并且这些 IRP 等待设备上电后运行。尽管设备处于低电源状态是个好想法,但你会注意到,我刚给你演示的 SelectDState 代码片段无条件地回到了 D0 状态。在我的测试中,如果我不这样做,Windows 2000 会在从 standby 状态的返回中挂起。这可能是我的代码也可能是操作系统代码中的一个错误。请关注本书的勘误信息。

请求设备电源IRP

在第五章的“I/O 请求包”中我们讨论了象 **IoAllocateIrp** 这样的用来创建 IRP 的函数。但你不能使用这些函数创建电源管理 IRP(实际上,你可以用那些函数创建 IRP_MN_POWER_SEQUENCE 请求,但其它 IRP_MJ_POWER 请求不行),使用 **PoRequestPowerIrp**,下面代码是执行在 **SelectDState** 之后的 **SendDeviceIrp** 动作代码:

```
case SendDeviceIrp:  
{  
    if (win98 && ctx->devstate == pdx->devpower)  
    {  
        ctx->status = STATUS_SUCCESS;  
        action = actiontable[ctx->state][AsyncNotify];  
        continue;  
    }  
    POWER_STATE powstate;  
    powstate.DeviceState = ctx->devstate;  
    NTSTATUS poststatus = PoRequestPowerIrp(pdx->Pdo,  
                                            ctx->MinorFunction,  
                                            powstate,  
                                            (PREQUEST_POWER_COMPLETE) PoCompletionRoutine,  
                                            ctx,  
                                            NULL);  
    if (NT_SUCCESS(poststatus))  
        break;  
    action = CompleteMainIrp;  
    ctx->status = poststatus;  
    continue;  
}
```

1. 参见本章后面的“Windows 98 兼容问题”中对这段代码的解释。

2. **PoRequestPowerIrp** 的第一个参数是设备 PDO 的地址。注意,我们请求的 IRP 总要先发往最顶层的过滤器设备对象 (FiDO)。第二个参数是该 IRP 的副功能码,可以为 IRP_MN_QUERY_POWER 或 IRP_MN_SET_POWER。第三个参数是 POWER_STATE,当我们请求一个查询或设置操作时,它应该包含一个设备电源状态值。第四个和第五个参数分别是当 IRP 完成时的回调函数地址和该函数上下文参数。最后一个参数是一个可选的 PIRP 变量的地址,该变量用于接收 PoRequestPowerIrp 创建的 IRP 的地址。
3. **PoRequestPowerIrp** 在创建并发出一个电源 IRP 后通常都返回 STATUS_PENDING。这个结果与任何其它的成功返回代码,实际上都意味着我们的回调函数最终会被调用。而回调函数又会调用 **HandlePowerEvent**。
4. 如果 **PoRequestPowerIrp** 失败,它不会再创建该 IRP,并且我们的回调函数也不会被调用。因此我们使该系统 IRP 失败,并返回 PoRequestPowerIrp 返回的任何状态码。

在系统上电期间,有限状态机在我们到达 SendDeviceIrp 时将处于 **SubPowerUpPending** 状态。**status** 变量将为 STATUS_MORE_PROCESSING_REQUIRED,如果我们正在等待设备 IRP 的完成,那么

`MainCompletionRoutine` 返回这个值是正确的。然后，当我们从 `SendDeviceIrp` 中 break 出来时，我们就中断了系统电源 IRP 的完成处理。

下面我将讨论由 `PoRequestPowerIrp` 请求的设备 IRP 会发生什么事情。

完成系统IRP

最后，`SendDeviceIrp` 请求的设备 IRP 被完成，于是电源管理器将调用回调例程 **PoCompletionRoutine**。回调例程再调用 `HandlePowerEvent` 并给出事件代码 **AsyncNotify**。在 `SubPowerUpPending` 状态中，我们的第一个动作将是 **SubPowerUpComplete**：

```
case SubPowerUpComplete:  
{  
    if (status == -1)  
        status = STATUS_SUCCESS;  
    action = CompleteMainIrp;  
    continue;  
}
```

这个动作例程执行的唯一任务是修改 **status** 变量。做这个的原因是 `HandlePowerEvent` 的结尾有一个 `ASSERT` 语句，该语句将确保某人改变了 **status**。在我们的假定中，我们并不关心返回的 **status** 的具体值，因为 `PoCompletionRoutine` 函数是一个无返回值函数。我想你不会希望触发一个 `ASSERT` 或一个 `BSOD`，除非确实有什么错误存在。

`SubPowerUpComplete` 的下一个动作是 `CompleteMainIrp`，它将导致 `DestroyContext` 调用。

降低电源级别的系统电源IRP

如果系统电源 IRP 暗含着不改变或降低系统电源级别，那么你也要用相同的副功能码请求设备电源 IRP，并且请求与系统电源状态对应的设备电源状态。当设备电源 IRP 完成时，你应该把这个系统电源 IRP 下传到下一层驱动程序。你需要为该系统电源 IRP 写一个完成例程，以便能够调用 `PoStartNextPowerIrp`，并且还可以执行一些额外的清除操作。图 8-7 显示了在这种情况下该 IRP 如何穿过系统。

系统电源 IRP

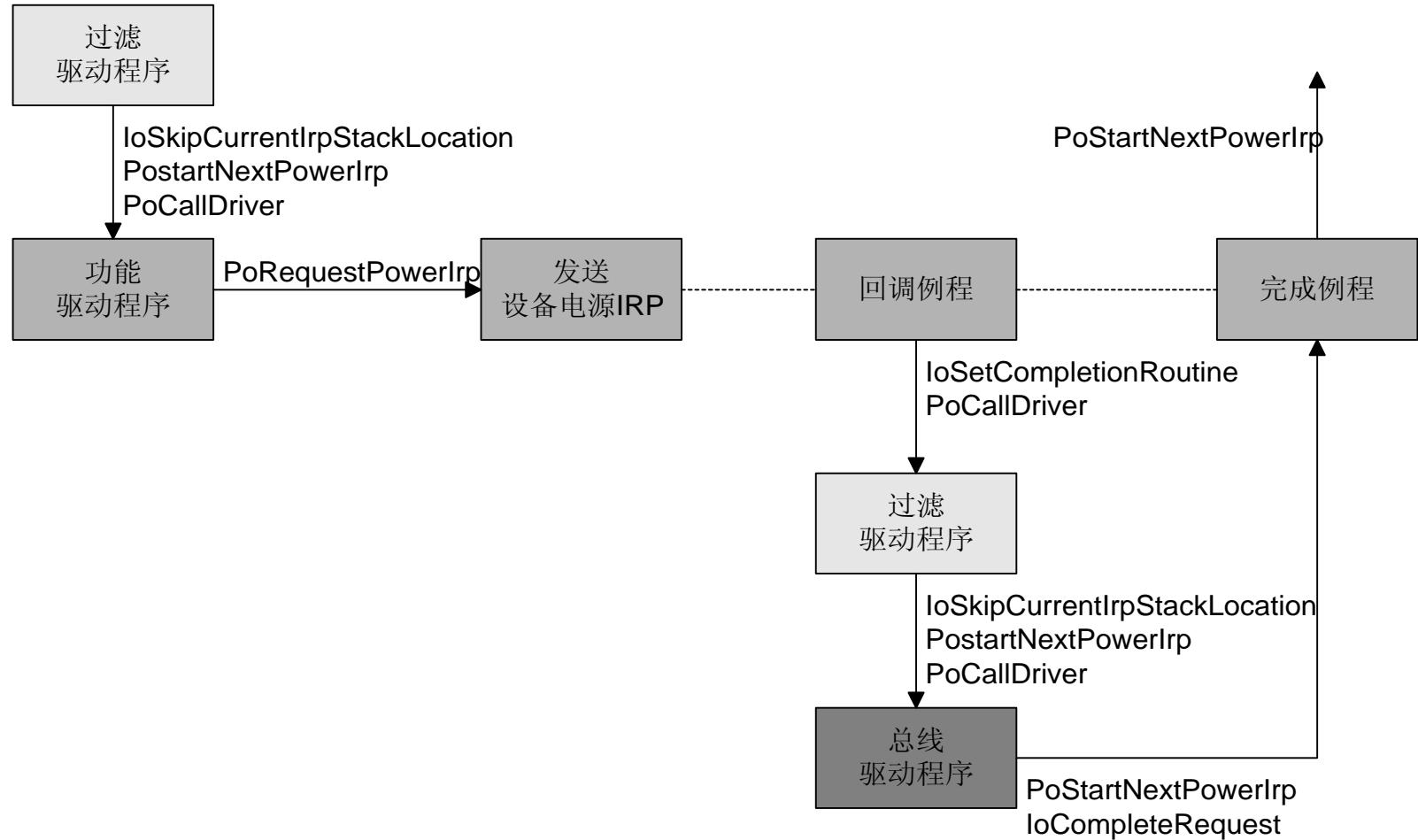


图 8-7. 降低系统电源消耗时的 IRP 流

图 8-8 显示了我们的有限状态机如何处理这种类型的 IRP。TriageNewIrp 把有限状态机置入 **SubPowerDownPending** 状态并跳到 SelectDState 动作。你已经看到了 SelectDState 选择了一个设备电源状态并引发了一个 SendDeviceIrp 动作，该动作请求了一个设备电源 IRP。在系统转入低电力消耗状态期间，我们应该在该设备 IRP 中指定一个更低的电源状态。

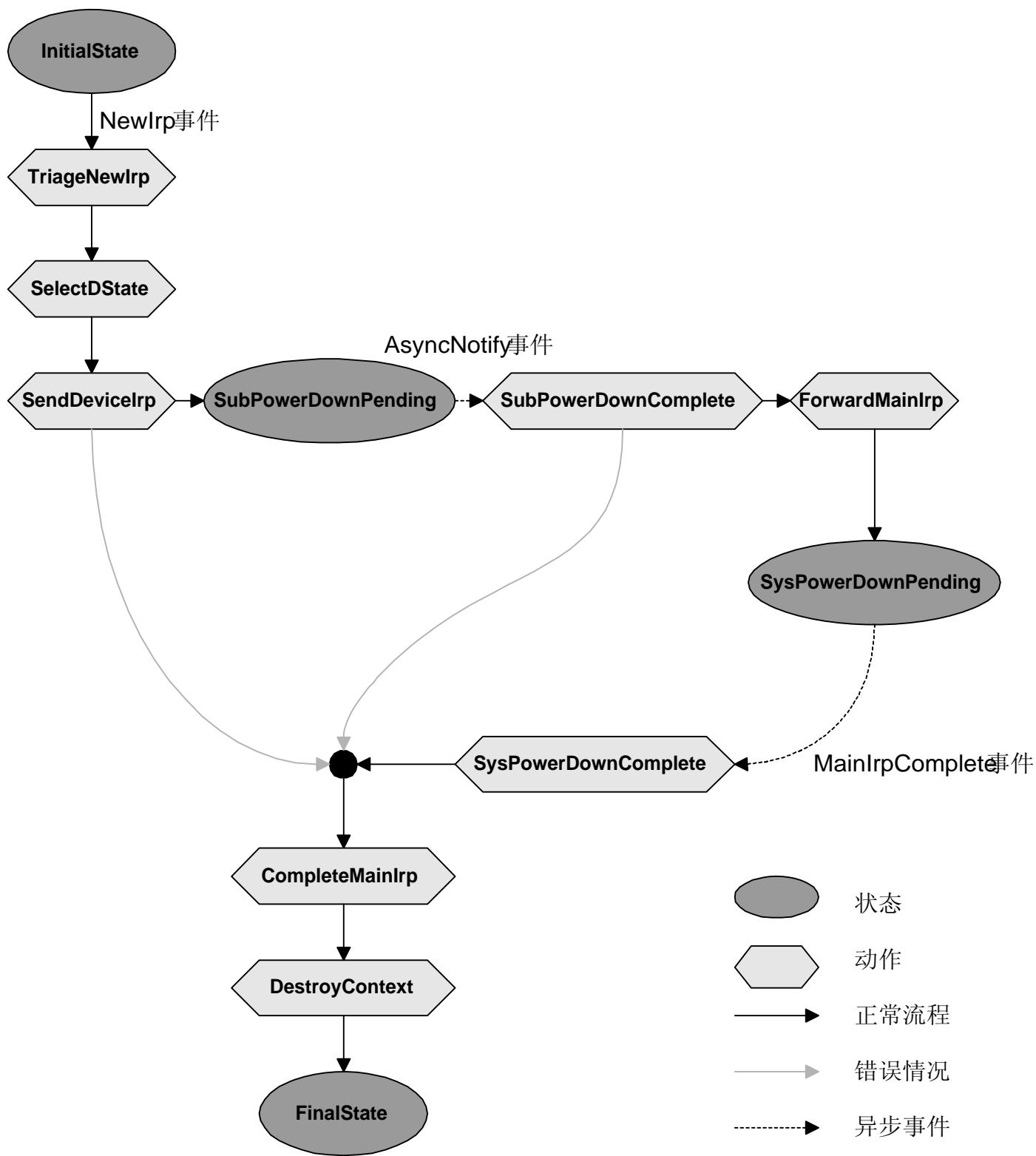


图 8-8. 降低系统电源级别时的状态转换

当设备 IRP 完成时，我们执行 **SubPowerDownComplete**:

```
case SubPowerDownComplete:
{
    if (status == -1)
        status = STATUS_SUCCESS;
    if (NT_SUCCESS(ctx->status))
    {
        ctx->state = SysPowerDownPending;
        action = ForwardMainIrp;
    }
    else
        action = CompleteMainIrp;
    continue;
}
```

正如你看到的，如果设备 IRP 失败，我们也使系统 IRP 失败。如果设备 IRP 成功，我们就进入 **SysPowerDownPending** 状态并经由 `ForwardMainIrp` 退出。当系统 IRP 完成，并且 `MainCompletionRoutine` 运行时，我们执行 **SysPowerDownComplete**：

```
case SysPowerDownComplete:
{
    if (stack->MinorFunction == IRP_MN_SET_POWER)
        pdx->syspower = stack->Parameters.Power.State.SystemState;
    action = CompleteMainIrp;
    continue;
}
```

这个动作的唯一目的是在我们的设备扩展中记录下新的系统电源状态，然后经由 `CompleteMainIrp` 和 `DestroyContext` 退出。

设备电源IRP

实际上，我们为系统电源 IRP 所做的一切就是为它们形成一个通道，并在系统 IRP 上下穿越驱动程序堆栈时请求设备 IRP。但对于设备电源 IRP，我们却有更多的事情要做。

首先，我们不希望在设备电源状态正改变时有任何实质性的 I/O 请求到来。因此我们应该尽早地进入一个使设备电源消耗逐渐降低的序列，我们等待任何未决的操作完成，并且我们停止处理新的操作。由于我们不能阻塞接收电源 IRP 的系统线程，因此需要一个异步机制。一旦当前 IRP 完成，我们将继续处理设备 IRP。

如果设备电源 IRP 暗含着要提高设备电源消耗级别，我们就把该 IRP 送到下一层驱动程序。图 8-9 显示了该 IRP 如何流过系统。总线驱动程序将使用某种专有方式处理设备的 set-power 请求，例如，使用总线专有的机制打开通向设备的电流，并完成该 IRP。你的完成例程将开始恢复设备上下文信息所需要的任何操作，并且将返回 `STATUS_MORE_PROCESSING_REQUIRED` 以中断该设备 IRP 的完成处理。当上下文恢复操作完成后，你可以回到处理实质性 IRP 的程序中并结束设备 IRP 的完成操作。

设备电源IRP

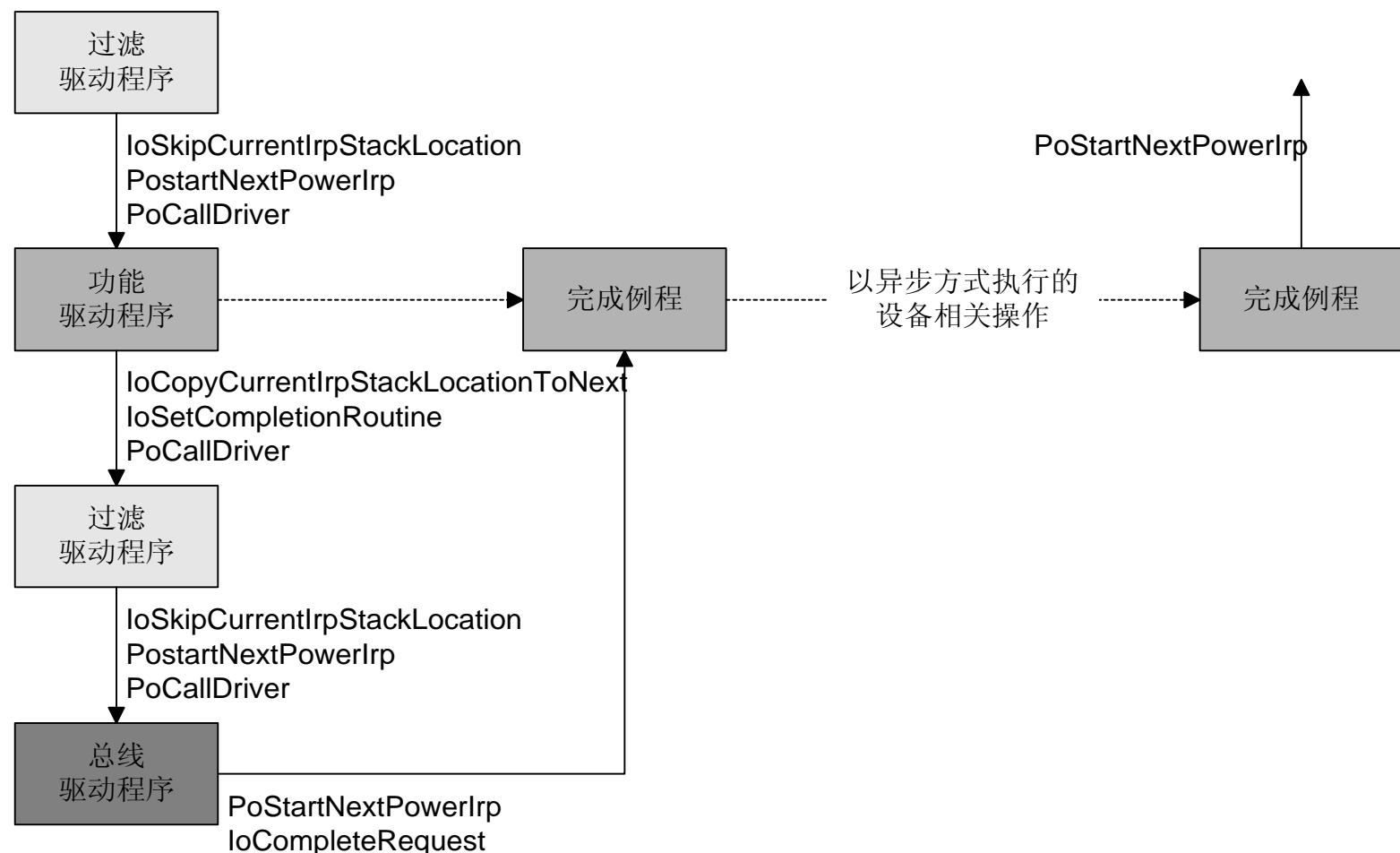


图 8-9. 提升设备电源级别时的 IRP 流

如果设备电源 IRP 暗含着不改变或降低设备的电源级别，那么你应该先执行任何设备相关的处理(以异步方式)，然后把该设备 IRP 送到下一层驱动程序，见图 8-10。一个设置操作的“设备专用处理”将包括保存设备上下文信息到内存中，以便以后恢复。对于一个查询操作，除了决定是成功还是失败该 IRP 之外，可能没有任何设备相关的处理。总线驱动程序将完成该请求。在查询操作中，你可能希望总线驱动程序以 STATUS_SUCCESS 结果完成该请求，以指出允许 IRP 中指定的电源状态改变。在设置操作中，你可能希望总线驱动程序做任何使设备进入指定设备电源状态的与总线相关的步骤。你的完成例程通过调用 PoStartNextPowerIrp 做一些清除工作。

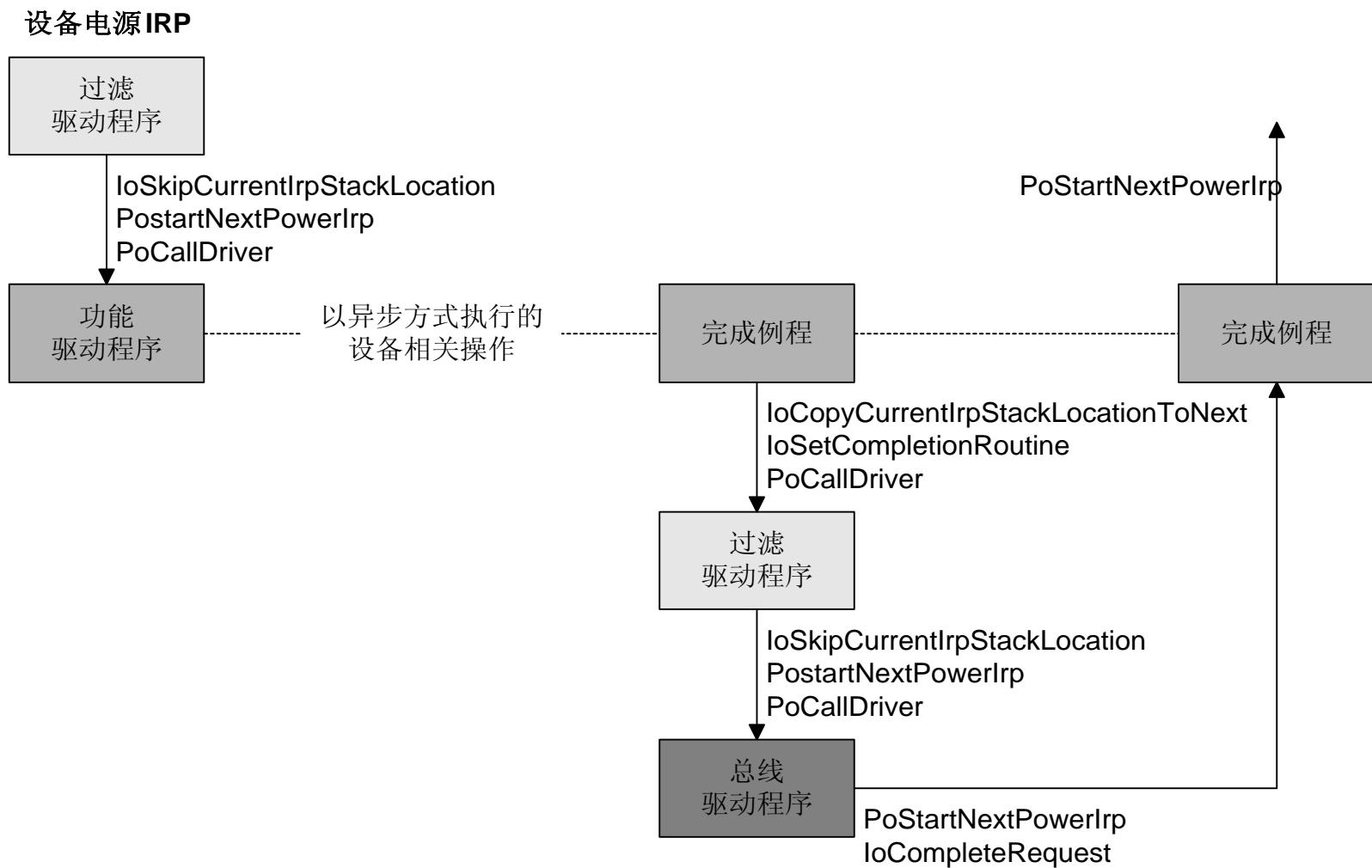


图 8-10. 降低设备电源级别时的 IRP 流

我写了 **StallRequestsAndNotify** 例程，该例程被 TriageNewIrp 使用。它执行的第一步是停止请求队列。如果设备当前正忙，它就记录下回调函数地址(使用 **GenericSaveRestoreComplete**，为了接收通知，我超载了该函数)，并返回 STATUS_PENDING。然后 TriageNewIrp 在 **QueueStallPending** 状态下退出。

如果设备不忙，**StallRequestsAndNotify** 将返回 STATUS_SUCCESS，并且不安排任何回调函数；因为队列被停止了，设备不会处于忙状态。然后 TriageNewIrp 直接进入 **QueueStallComplete** 动作。

我们或者直接从 TriageNewIrp 到达(当设备空闲时，队列可能因为某些与电源相关的其它原因而停止)**QueueStallComplete** 例程，或者当客户驱动程序调用 **StartNextPacket**(以指出其已经完成当前 IRP 的处理)时到达 **QueueStallComplete** 例程。**StartNextPacket** 将调用我们在 **StallRequestsAndNotify** 中给出的通知例程，并且通知例程会向有限状态机发出一个 **AsyncNotify** 事件。现在 **QueueStallComplete** 把设备 IRP 分成如下四类：

```

case QueueStallComplete:
{
    if (stack->MinorFunction == IRP_MN_SET_POWER)
    {
        if (stack->Parameters.Power.State.DeviceState < pdx->devpower)
        {
            action = ForwardMainIrp;
            SETSTATE(DevPowerUpPending);
        }
    }
}

```

```

    }
    else
        action = SaveContext;
    }
    else
    {
        if (stack->Parameters.Power.State.DeviceState < pdx->devpower)
        {
            action = ForwardMainIrp;
            SETSTATE(DevQueryUpPending);
        }
        else
            action = DevQueryDown;
    }
    continue;
}

```

`QueueStallComplete` 将使我们执行下一个动作，该动作由我们正处理的 **IRP** 类型指出，类型见表 8-5。

表 8-5. 设备 **IRP** 的下一个动作

副功能码	提升还是降低电源级别？	下一个动作
IRP_MN_QUERY_POWER	提升电源级别 降低或维持当前电源级别	ForwardMainIrp DevQueryDown
IRP_MN_SET_POWER	提升电源级别 降低或维持当前电源级别	ForwardMainIrp SaveContext

设置更高级的设备电源状态

图 8-11 显示了一个 **IRP_MN_SET_POWER** 请求带来的状态转换，该请求指定了一个比当前设备电源状态更高级的电源状态。

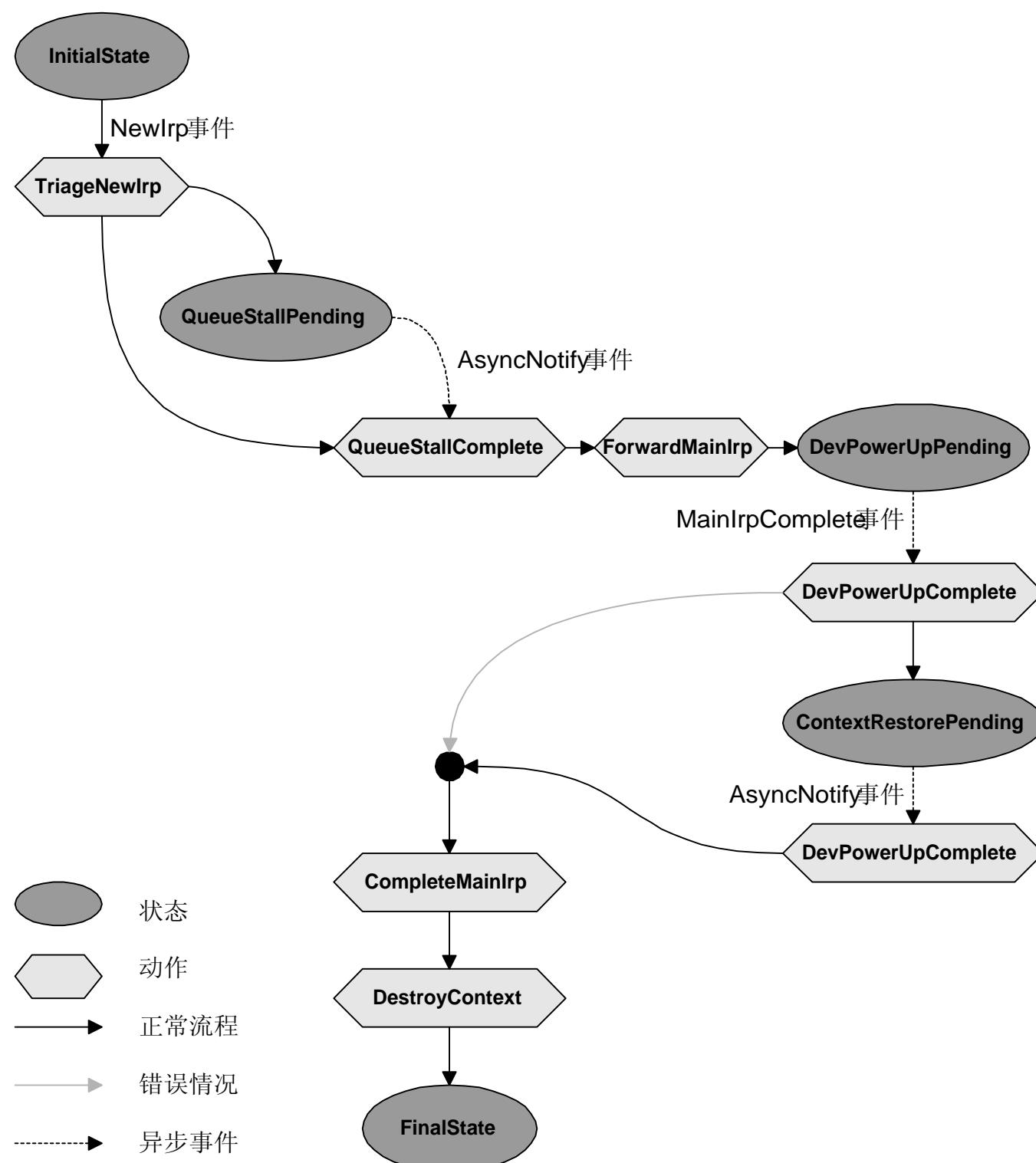


图 8-11. 设置更高级设备电源状态时发生的状态转换

ForwardMainIrp 将安装一个完成例程并向下发送该 IRP。当 MainCompletionRoutine 最后获得控制时，它发出一个 **MainIrpComplete** 事件。我们将处于 **DevPowerUpPending** 状态，因此我们将执行 **DevPowerUpComplete** 动作：

```

case DevPowerUpComplete:
{
    if (!NT_SUCCESS(ctx->status) || stack->MinorFunction != IRP_MN_SET_POWER)
    {
        action = CompleteMainIrp;
        continue;
    }
    status = STATUS_MORE_PROCESSING_REQUIRED;
    DEVICE_POWER_STATE oldpower = pdx->devpower;
    pdx->devpower = stack->Parameters.Power.State.DeviceState;
    if (pdx->RestoreContext)
    {

```

```

ctx->state = ContextRestorePending;
(*pdx->RestoreDeviceContext)(pdx->DeviceObject, oldpower, pdx->devpower, ctx);
break;
}
action = ContextRestoreComplete;
continue;
}

```

这里我们需要完成的主要任务是恢复在上一个 power-down 转换时丢失的设备上下文。由于我们不可以阻塞自己的线程，所以我们先执行任何必需的操作，然后返回 STATUS_MORE_PROCESSING_REQUIRED 中断设备 IRP 的完成。当恢复操作完成时，客户驱动程序调用 GenericSaveRestoreComplete，该函数发出 AsyncNotify 事件。在这一点上，我们将处于 **ContextRestorePending** 状态，所以我们执行 **ContextRestoreComplete** 动作：

```

case ContextRestoreComplete:
{
    if (event == AsyncNotify)
        status = STATUS_SUCCESS;
    action = CompleteMainIrp;
    if (!NT_SUCCESS(ctx->status) || pdx->devpower != PowerDeviceD0)
        continue;
    ctx->UninstallQueue = TRUE;
    continue;
}

```

该动作例程的主要结果是，我们在得到一个使设备进入 D0 状态的 IRP_MN_SET_POWER 请求后重新启动了实质性的 IRP 队列。然后经由 CompleteMainIrp 和 DestroyContext 退出。

查询更高级的设备电源状态

你不应该期望收到一个比设备当前电源状态更高的 IRP_MN_QUERY_POWER，但是如果碰巧收到一个，你也不要破坏系统。下面代码显示了当这样的查询在低级驱动程序完成时 GENERIC 是如何做的。(参见图 8-12 的状态图)

```

case DevQueryUpComplete:
{
    if (NT_SUCCESS(ctx->status) && pdx->QueryPower)
        if (!(*pdx->QueryPower)(pdx->DeviceObject,
                                  pdx->devpower,
                                  stack->Parameters.Power.State.DeviceState))
            ctx->status = STATUS_UNSUCCESSFUL;
    action = CompleteMainIrp;
    continue;
}

```

即，GENERIC 允许客户驱动程序通过调用 **QueryPower** 函数接受或拒绝这种查询，然后经由 CompleteMainIrp 和 DestroyContext 退出。

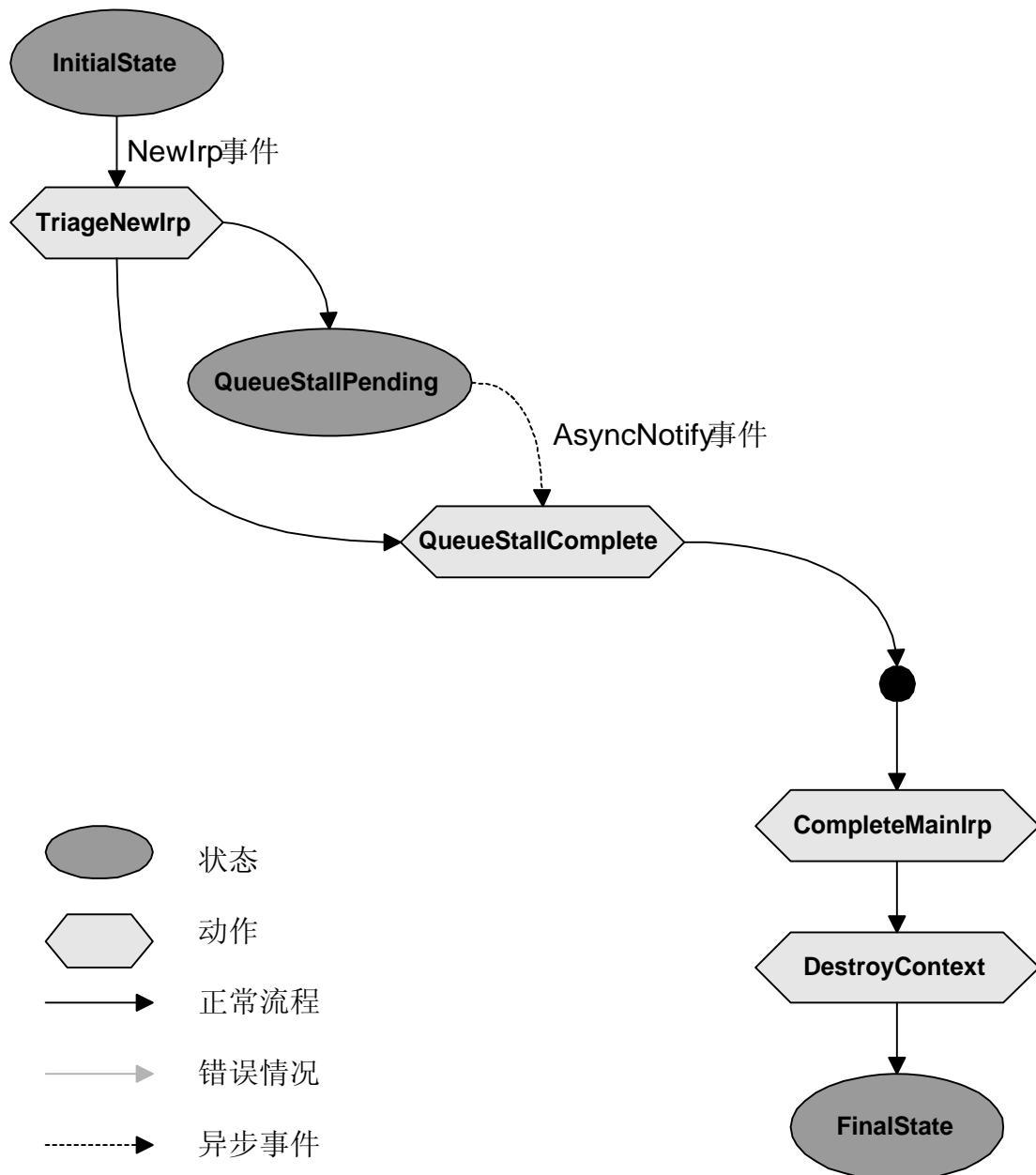


图 8-12. 关于更高级设备电源状态的查询的状态转换

设置更低级的设备电源状态

如果是设置一个低于或等于当前设备电源状态的 IRP_MN_SET_POWER 请求，则有限状态机将穿过图 8-13 所示的状态转换过程。

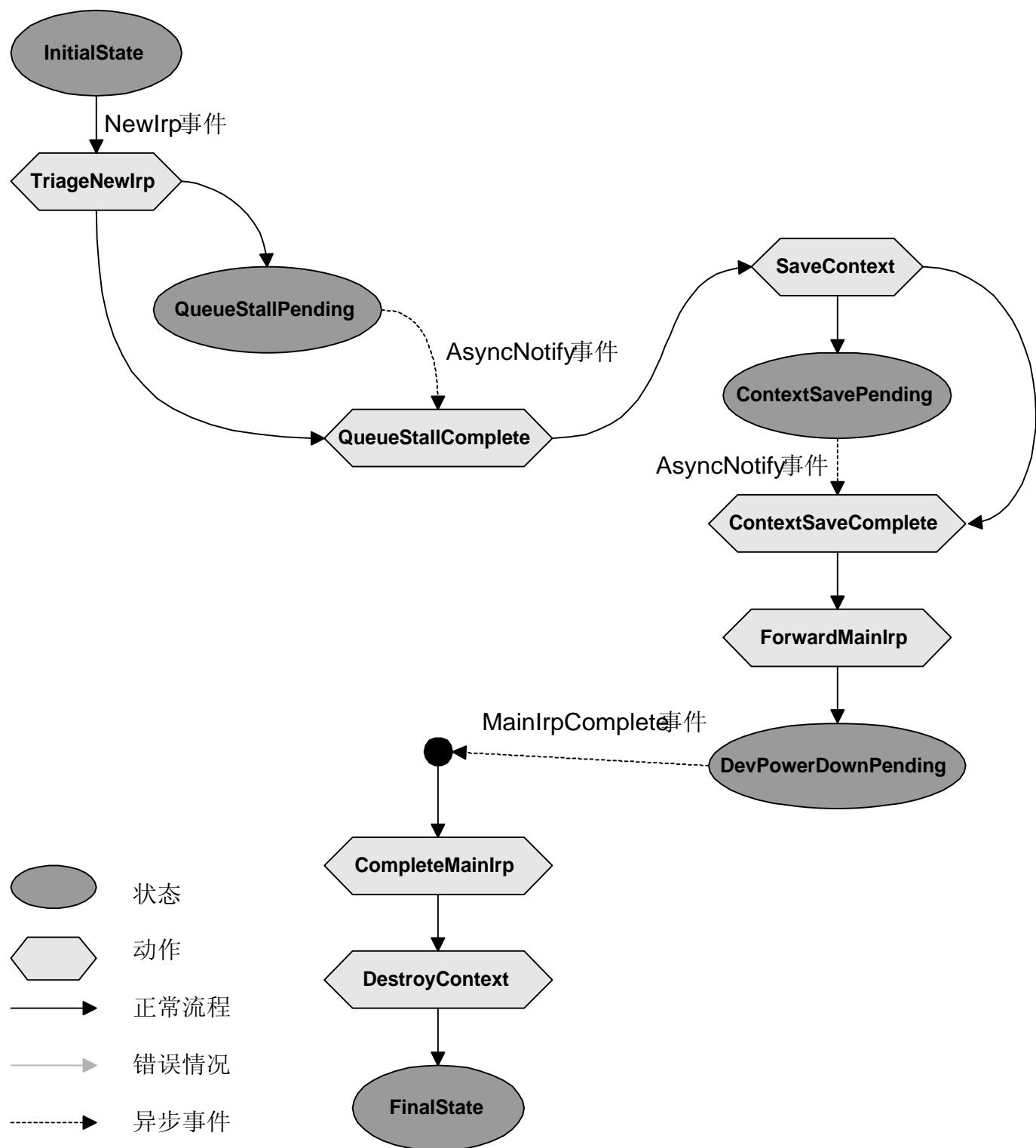


图 8-13. 设置更低级设备电源状态时的状态转换

SaveContext 将开始一个异步过程以保存设备上下文信息:

```

case SaveContext:
{
    DEVICE_POWER_STATE devpower = stack->Parameters.Power.State.DeviceState;
    if (pdx->SaveDeviceContext && devpower > pdx->devpower)
    {
        ctx->state = ContextSavePending;
        (*pdx->SaveDeviceContext)(pdx->DeviceObject, pdx->devpower, devpower, ctx);
        break;
    }
    action = ContextSaveComplete;
}
    
```

当保存操作完成后，客户驱动程序将调用 **GenericSaveRestoreComplete**，该函数将发出 **AsyncNotify** 事件。在此我们会处于 **ContextSavePending** 状态，所以我们执行 **ContextSaveComplete** 动作:

```

case ContextSaveComplete:
{
    if (event == AsyncNotify)
        status = STATUS_SUCCESS;
    ctx->state = DevPowerDownPending;
    action = ForwardMainIrp;
    DEVICE_POWER_STATE devpower = stack->Parameters.Power.State.DeviceState;
    if (devpower <= pdx->devpower)                                     <--2
        continue;
    pdx->devpower = devpower;
    if (devpower > PowerDeviceD0)                                     <
        ctx->UninstallQueue = FALSE;                                    <
    continue;
}

```

1. 我们直接从 **GenericSaveRestoreComplete** 到这里，我们需要改变 **status** 以防止 **ASSERT** 故障(没有其它原因)。
2. 如果我们没有真正改变电源状态，这里就没有其它工作要做。
3. 当我们降低电源状态时，我们在这里记录下新的设备电源状态。
4. 如果设备正处于 **low-power** 或 **no-power** 状态，我们将使实质性 IRP 队列停止。

下一个动作是 **ForwardMainIrp**，它向驱动程序堆栈发送设备 IRP。总线驱动程序将关闭物理电流并完成该 IRP。当 **MainCompletionRoutine** 通知一个 **MainIrpComplete** 事件时，我们将再次看到它，它带我们直接到达 **CompleteMainIrp**，最后到达 **DestroyContext**。

查询更低级的设备电源状态

一个指定了低于或等于当前设备电源状态的 **IRP_MN_QUERY_POWER** 请求是功能驱动程序表决电源级别改变时所使用的基本手段。尽管 DDK 没有专门说应该在处理系统查询时创建这样的请求，但这是个好想法。总之，你必须处理设备查询请求并且最好把所有查询逻辑都放到一个地方。图 8-14 显示了有限状态机如何处理这样的查询。

对于这种 IRP，**QueueStallComplete** 后面将跟着 **DevQueryDown**:

```

case DevQueryDown:
{
    DEVICE_POWER_STATE devpower = stack->Parameters.Power.State.DeviceState;
    if (devpower > pdx->devpower
        && pdx->QueryPower
        && !(*pdx->QueryPower)(pdx->DeviceObject, pdx->devpower, devpower))
    {
        ctx->status = STATUS_UNSUCCESSFUL;
        action = DevQueryDownComplete;
        continue;
    }
    ctx->state = DevQueryDownPending;
    action = ForwardMainIrp;
    continue;
}

```

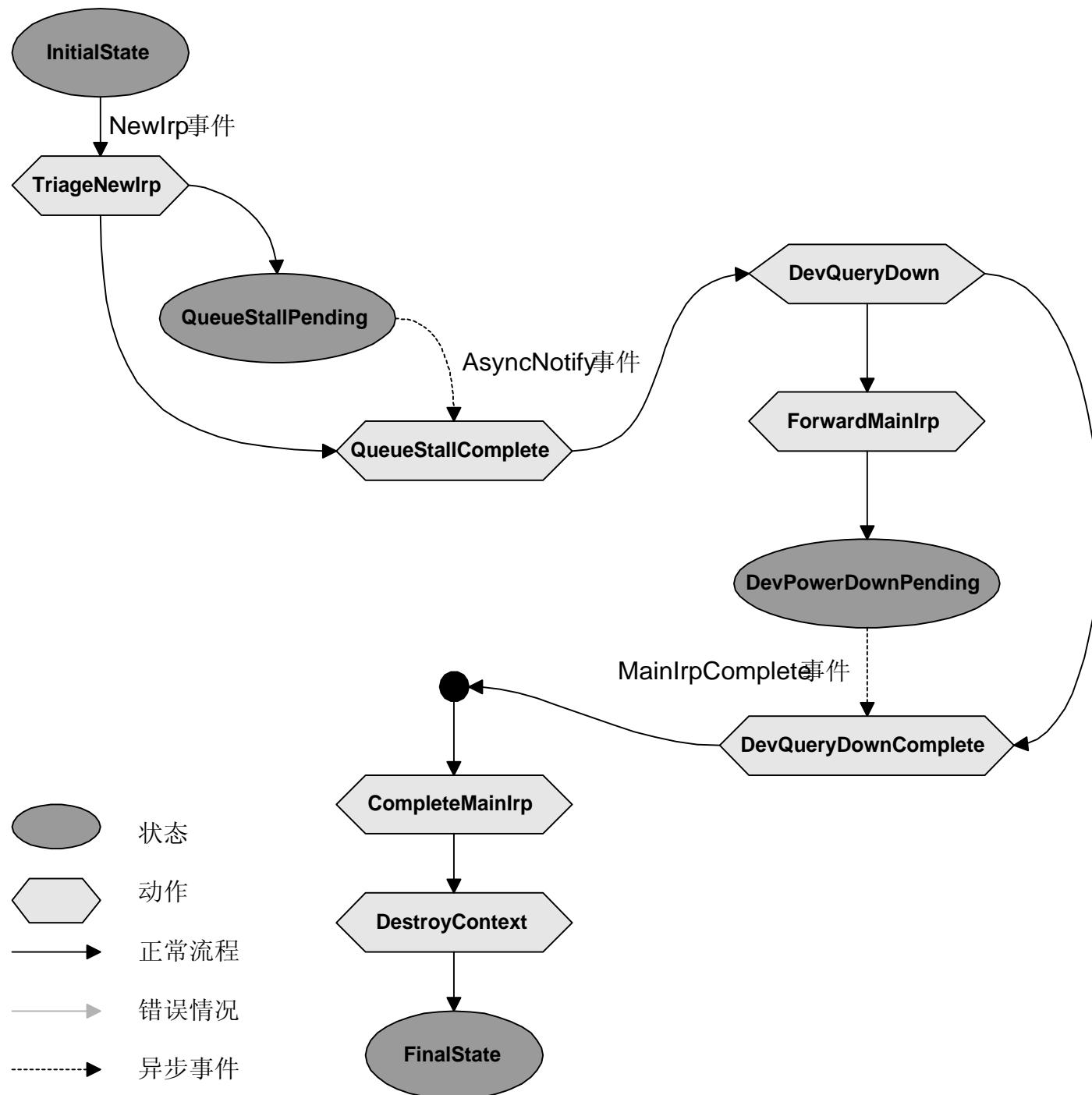


图 8-14. 更低级设备电源状态查询的状态转换

基本上，**GENERIC** 让客户驱动程序来决定该查询是否应该成功。如果客户驱动程序说“**Yes**”，我们就进入 **DevQueryDownPending** 状态，并经由 **ForwardMainIrp** 向下发出该查询请求，然后退出。该 IRP 的完成操作使我们进入了 **DevQueryDownComplete** 动作：

```
case DevQueryDownComplete:
{
    if (ctx->status == STATUS_INVALID_PARAMETER)
        ctx->status = STATUS_SUCCESS;
    if (NT_SUCCESS(ctx->status))
        ctx->UnstallQueue = FALSE;
    action = CompleteMainIrp;
    continue;
}
```

如果查询成功，我们所要做的基本动作是使实质性的 IRP 队列停止(如果 **CompleteMainIrp** 看到上下文结构中的 **UnstallQueue** 标志被设置，它将重新启动该队列)。回想一下我们在收到该查询时首先停止了这个队列。我们把这个队列停止，直到最后有人向我们发送一个 **set-power** 请求并使设备进入 D0 状态，那时再启动。

Windows 98 的USB hub驱动程序有一个bug, 会使任何指定了设备电源状态的**IRP_MN_QUERY_POWER**请求失败。为了使我的电源管理代码避免遇到这个永久的bug(也防止你的计算机进入standby状态), 你可以象上面粗体字那样修改**DevQueryDownComplete**程序。修改后的例子代码在本书的[SP-2](#)服务包中给出。

其它电源管理细节

在这一节中，我将描述其它与电源管理相关的细节，包括需要在设备对象中设置的标志，设备唤醒特征的控制，在设备空闲后的预定时间内安排设备进入更低的电源状态，以及优化上下文恢复操作。

在**AddDevice**中设置的标志

设备对象中的两个标志位，见表 8-6，控制电源管理的各个方面。当你在 **AddDevice** 函数中调用了 **IoCreateDevice** 例程之后，这两个位都将被置 0，你可以根据具体环境设置这两个位。

表 8-6. *DEVICE_OBJECT* 中的电源管理标志

标志	简要描述
DO_POWER_PAGABLE	驱动程序的 IRP_MJ_POWER 派遣例程必须运行在 PASSIVE_LEVEL 级
DO_POWER_INRUSH	该设备在上电时需要大电流

如果 IRP_MJ_POWER 请求的派遣函数必须运行在 PASSIVE_LEVEL 级别，你需要设置 DO_POWER_PAGABLE 标志。该标志的名称含有相关的含义，因为只有在 PASSIVE_LEVEL 级上才允许分页操作。如果你把该标志设置为 0，则电源管理器可以在 DISPATCH_LEVEL 级上向你发送电源管理请求。实际上，在当前发行的 Windows 2000 中总应该这样做。

如果你的设备在上电时需要大电流，设置 DO_POWER_INRUSH 标志，这可以使多个有这样要求的设备不同步上电。另外，电源管理器会在 DISPATCH_LEVEL 级上向 inrush 设备发送电源管理请求，这意味着你不能同时设置 DO_POWER_PAGABLE 标志。

如果设备的 ASL 描述中这样指定，那么系统的 ACPI 过滤器驱动程序将在 PDO 中自动设置 INRUSH 标志，这样系统会正确地顺序化有 INRUSH 标志设备的电源供应，因此你并不需要在自己的设备对象中设置这个标志。但是，如果系统不能自动确定你的设备是否需要 inrush 对待，你需要自己设置这个标志。

设备的所有设备对象中的 PAGABLE 和 INRUSH 标志设置都要一致。如果 PDO 设置了 PAGABLE 标志，则其它设备对象也应该设置 PAGABLE 标志。否则将发生代码为 DRIVER_POWER_STATE_FAILURE 的 bug check(一个 PAGABLE 设备在一个非 PAGABLE 设备之上是合法的，但相反则不行)。如果设备对象设置了 INRUSH 标志，那么它自己和任何它下面的低级设备对象都不应该设置 PAGABLE 标志，否则将发生代码为 INTERNAL_POWER_ERROR 的 bug check。如果你正写一个磁盘驱动程序，不要忘了为响应设备分页文件的 PnP 通知，你可以随时在分页和非分页状态间改变。

设备的唤醒特征

某些设备具有硬件唤醒特征，这允许它们在外部事件发生时可以唤醒沉睡中的系统，见图 8-15。当前 PC 上的电源开关就是一个这样的设备。许多 modem 和网卡也能监听进来的电话和数据包。

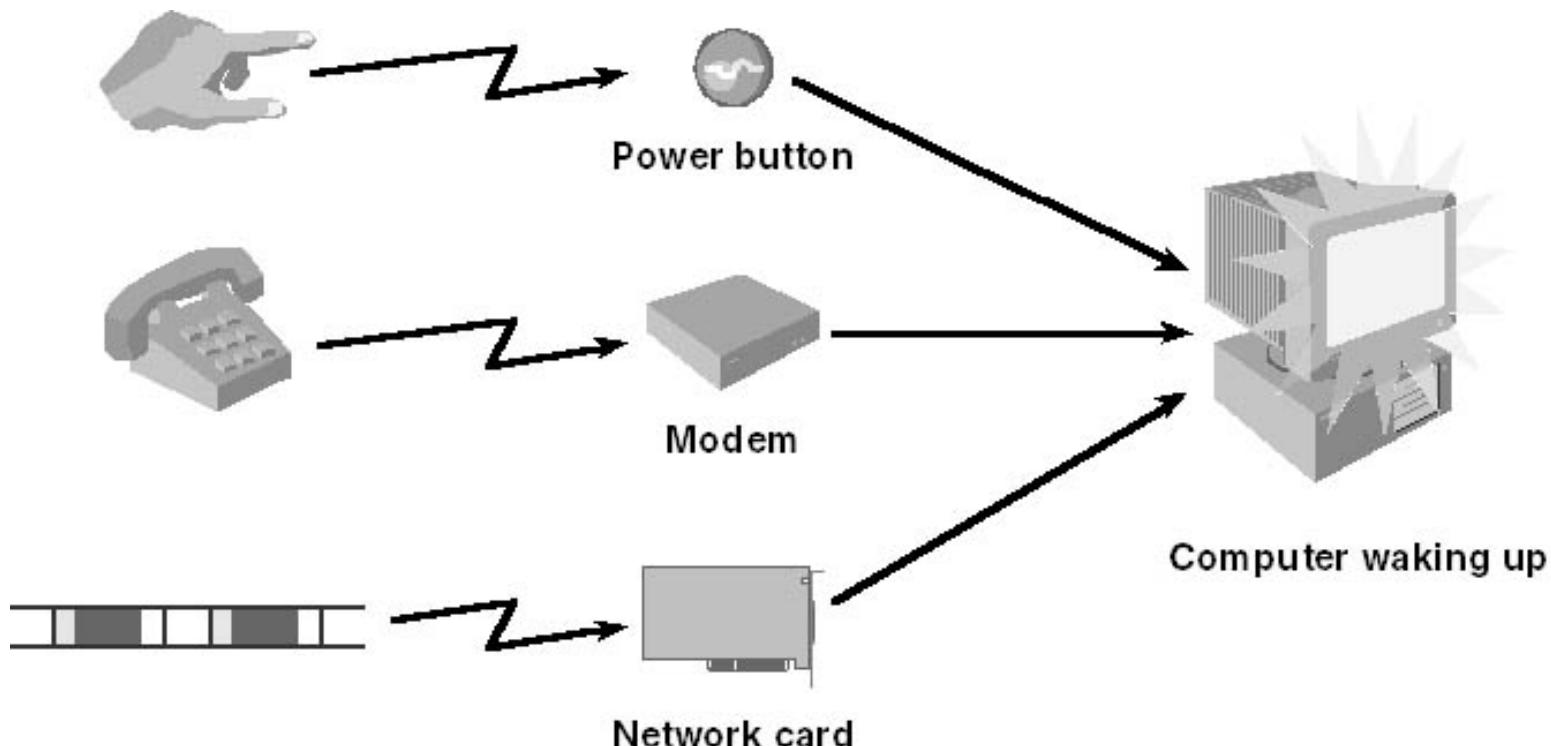


图 8-15. 设备唤醒系统的例子

如果你的设备有唤醒特征，那么你的功能驱动程序就必须承担额外的电源管理责任，这超出了我们已经讨论过的内容。第一个额外责任是处理 **IRP_MJ_POWER** 请求族中的 **IRP_MN_WAIT_WAKE** 请求。除了安装一个标准 I/O 完成例程和下传该 IRP 之外，大部分设备中与 **WAIT_WAKE** 请求对应的派遣函数不必做任何处理。例如，**USB** 和 **PCI** 总线的驱动程序实现了总线专有的装备、解除、和探测唤醒特征的能力。更明确地说，除了总线规范中描述的相关特征之外，如果你的设备没有与设备唤醒相关的额外特征，就不必做任何专门的处理。

如果一个 **IRP_MN_QUERY_POWER** 请求中指定了与你的唤醒特征不兼容的电源状态，你应该使该 IRP 失败。如果是一个查询系统电源状态的请求，用设备能力结构中的 **SystemWake** 域与被提议的新状态相对比，这将得出设备可以唤醒系统所需要的最低的系统电源状态。如果是设备电源状态的查询，用 **DeviceWake** 域与被提议的新状态进行比较，这将得出设备能发出唤醒信号所需的最低的设备电源状态。如果比较的结果表明提议的电源状态太低，以 **STATUS_INVALID_DEVICE_STATE** 失败该查询 IRP。否则完成该查询。

你需要在适当时间发出 **IRP_MN_WAIT_WAKE** 请求。因此，要象下面代码这样调用 **PoRequestPowerIrp**:

```
typedef struct _DEVICE_EXTENSION {
    PIRP WaitWakeIrp;
};

NTSTATUS SomeFunction(...)
{
    ...
    POWER_STATE junk;
    junk.SystemState = pdx->devcaps.SystemWake;
    status = PoRequestPowerIrp(pdx->Pdo,
        IRP_MN_WAIT_WAKE,
        junk,
        (PREQUEST_POWER_COMPLETE) WaitWakeCallback,
        pdx,
        &pdx->WaitWakeIrp);
    ...
}
```

关系到唤醒的最后的额外责任是当 **WAIT_WAKE** 请求不再需要时取消它，如下面代码:

```
PIRP Irp = (PIRP) InterlockedExchangePointer(&pdx->WaitWakeIrp, NULL);
```

```
if (Irp)
    IoCancelIrp(Irp);
```

对于大多数设备，当 WAIT_WAKE 完成时你需要执行三个任务。应该把设备扩展结构中指向活动 WAIT_WAKE 请求的成员置空。这可以防止驱动程序的其它部分认为该 WAIT_WAKE 仍旧活动。你应该发出一个设备 set-power 请求以恢复设备的电源。在这里，某些设备还会执行解除设备唤醒特征的某种设备相关操作。最后，你希望 WAIT_WAKE 请求会自动重发，以便将来再次装备设备唤醒特征。这些任务中的第一个(置空 WAIT_WAKE 请求的指针)应该在派遣函数安装的标准 I/O 完成例程中被完成。另外两个任务(为你的设备上电，请求一个新的 WAIT_WAKE IRP)应该在回调函数中完成，这个回调函数(在我给出的代码片段中是 **WaitWakeCallback**)是你调用 PoRequestPowerIrp 时指定的。

注意

在我认为百分之百保证你在为 WAIT_WAKE 调用 **IoCancelIrp** 函数时会得到一个有效的指针会十分困难。在你的 I/O 完成例程置空该 IRP 隐藏指针前的一纳秒，你可以决定取消该 IRP。完成操作将运行，并且一旦你的回调例程返回，电源管理器将调用 **IoFreeIrp** 结束这个完成操作。此后，**IoCancelIrp** 或总线驱动程序的取消例程会试图使用这个现在已经无效的 IRP。这与我们在第五章所遇到的是同一类问题。为此，我和 Microsoft 的一个开发员给出了一个优美的解决方案，具体代码见光盘中的 **GENERIC** 例子。

何时发出WAIT_WAKE

在上一段中，我解释了如何发出 WAIT_WAKE IRP、如何取消该 IRP，以及其完成时我们该做什么。现在我解释何时该发出该 IRP。

这个问题的第一部分回答是，你需要一种方法来了解用户是否希望你的设备使用唤醒特征。除非用户阻止，你的驱动程序应该使用唤醒特征。在系统电源降低时，用户通过使用某种用户接口元素(例如类似于 POWCPL.DLL 的控制面板小件)指出设备是否可以使用唤醒特征。然后用户接口元素与驱动程序沟通，使用私有的 IOCTL 接口或通过 WMI 控制来设置。在 Windows 2000 中的某些地方，用户模式程序可能会使用至今还没实现的 **RequestDeviceWakeup** 和 **CancelDeviceWakeupRequest** API 来触发对驱动程序的 WMI 调用。

这个问题的第二部分回答关系到你何时调用 PoRequestPowerIrp 来请求 WAIT_WAKE。DDK 指出，当设备处于 D0 状态并且现在没有设备电源转换正在进行时，你可以请求一个 WAIT_WAKE。较合适的时间是在用户告诉你可以允许设备的唤醒特征时，或者当你处理一个将要降低设备电源状态的系统电源查询时。

如果用户告诉你要禁止唤醒特征，或者当你处理一个请求时，你应该禁止设备的唤醒特征(并取消未决的 WAIT_WAKE)。

空闲检测

通常用户在设备不被使用时不希望其消耗任何电力。你可以向电源管理器寄存这种空闲检测要求，如果你的设备空闲了指定长度的时间，则电源管理器自动向设备发出降低电源消耗的设备 IRP。这个空闲检测机制涉及两个服务函数：**PoRegisterDeviceForIdleDetection** 和 **PoSetDeviceBusy**。

为了寄存空闲检测，调用下面服务函数：

```
pdx->idlecount = PoRegisterDeviceForIdleDetection(pdx->Pdo,
                                                    ulConservationTimeout,
                                                    ulPerformanceTimeout,
                                                    PowerDeviceD3);
```

PoRegisterDeviceForIdleDetection 的第一个参数是设备 PDO 的地址。第二个和第三个参数指定了单位为秒的超时时间。**conservation** 时期将用于用电池供电的系统中，尽量维持电力供应。**performance** 时期应用于使用

交流电源供电的系统中，使系统达到最大性能。第四个参数指定了设备在经过指定时间的空闲状态后被强制进入的设备电源状态。

指出自己没有处于空闲状态

PoRegisterDeviceForIdleDetection 的返回值是一个长整型变量的地址，系统把该变量用做计数器。每一秒钟，电源管理器都增加该整型的值。如果它到达适当的超时值，电源管理器就向你发送一个设备 **set-power IRP** 以指出你寄存的电源状态。在驱动程序的许多地方，你应该把该计数器重置为 0，以重启动空闲检测周期：

```
if (pdx->idlecount)
    PoSetDeviceBusy(pdx->idlecount);
```

PoSetDeviceBusy 是 WDM.H 文件中的一个宏，它无条件地向其指针参数中存如一个 0。这表明 **PoRegisterDeviceForIdleDetection** 将返回一个 **NULL** 指针，所以你在调用 **PoSetDeviceBusy** 前不用检测其是否为 **NULL**。

现在，我已经描述了 **PoSetDeviceBusy** 所做的工作，所以你可能看出了它的名字有些误导。它没有告诉电源管理器你的设备是“忙”的，在这种情况下，你希望以后会有另一个调用来指出设备不再“忙”。另外，它指出了在特定时刻下设备的状态，即在你使用这个宏时设备是非空闲的。这并不是咬文嚼字。如果你的设备正忙于某种活动请求，你应该希望有代码来阻止空闲检测。所以，你会希望在驱动程序的多个地方调用 **PoSetDeviceBusy**：从各种派遣例程、从 **StartIo** 例程，等等。基本上，你应该确保这个检测时期长于你在两个 **PoSetDeviceBusy** 调用之间的最长经过时间。调用 **PoSetDeviceBusy** 是你在一个请求的正常处理中做出的。

注意

PoRegisterSystemState 允许你阻止电源管理器改变系统电源状态，但你不能使用它来阻止空闲超时。此外，Windows 98 没有实现该函数，所以需要在 Windows 2000 和 Windows 98 之间移植的驱动程序必须注意。

空闲超时的选择

选择恰当的空闲超时值并不十分简单。某些设备可以在其设备类中的标准电源策略上指定-1 超时。在写这本书时，仅有 **FILE_DEVICE_DISK** 和 **FILE_DEVICE_MASS_STORAGE** 设备可以这样做。你可能需要超时常量有默认值，这个最后仍可以被用户控制。

除非你的设备适合系统设计者制定的一般空闲检测方案，否则你需要提供一个用户模式部件以允许用户指定超时值。为了更好地融入操作系统，这个部件应该是电源控制面板小件的属性页扩展。即，你需要提供一个实现了 **IShellPropSheetExt** 和 **IShellExtInit** COM 接口的用户模式 DLL。这个 DLL 将符合 shell extension DLL 的通用描述。如果你需要了解如何写这个特殊的用户接口软件，应该研究 shell extension DLL。

光盘中的 **WDMIDLE** 例子包括了一个 shell extension DLL(**POWCPL.DLL**)，你可以复制并修改它。如果你安装了这个例子，你将看到电源选项属性单中多了一个属性页，见图 8-16。**POWCPL.DLL** 使用用户模式函数来枚举所有已经寄存了 **GUID_WDMIDLE** 接口的设备，并在一个列表框中列出它们的“友好”名称。它使用一个私有的 I/O 控制(**IOCTL**，见第九章)方案来查询并修改 **WDMIDLE.SYS** 使用的空闲超时常量。使用 **IOCTL** 方法可以得到一个在 Windows 2000 和 Windows 98 中通用的方案。另一种可能的方法是使用 COM 接口，该接口是 **WMI** 的一部分。(见第十章)这种方法十分麻烦并且不能用于 Windows 98 的原始版本，所以我没有用这种方法编制 **POWCPL.DLL**。

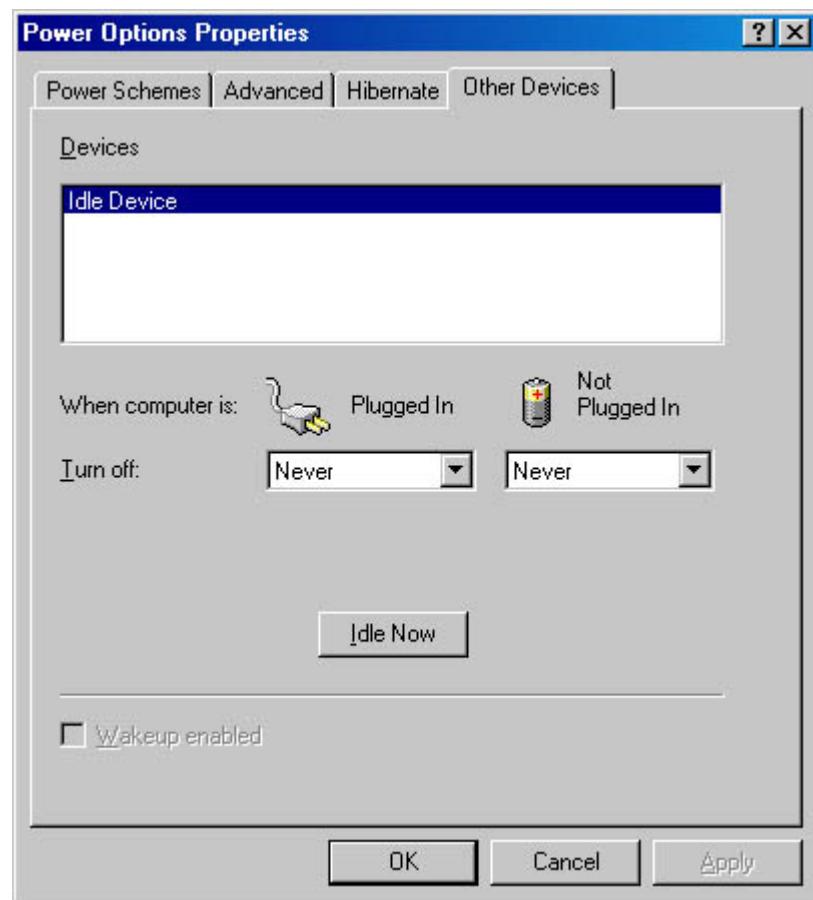


图 8-16. 空闲设备的属性页

在有用户接口的驱动程序这一边，**IRP_MJ_DEVICE_CONTROL** 处理程序将回答各种改变电源管理设置的请求和各种查询请求。用户希望这些设置一旦指定就可以在系统重启动后保持有效。因此，驱动程序需要把这些常量的当前值记录到注册表中。另外，在 **StartDevice** 函数执行期间，驱动程序需要从注册表中读取这些固定设置，并根据这些用户期望值初始化驱动程序。

所有这些细节，尽管对一个要交付产品的十分重要，但它们都略微触及了电源管理问题，所以我不将在这里讨论它们的代码。

从空闲状态中唤醒

如果你实现了空闲检测，你还需要提供一种方法，以便以后恢复设备的电源，注意没有人帮你做这些。我写了一个名为 **SendDeviceSetPower** 的函数来处理这个细节。你可以把类似下面的代码放到恢复电源 IRP 的派遣函数中。

```
NTSTATUS DispatchWrite(IN PDEVICE_OBJECT fdo, IN PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    if (pdx->idlecount)
        PoSetDeviceBusy(pdx->idlecount);

    if (pdx->powerstate > PowerDeviceD0)
    {
        NTSTATUS status = SendDeviceSetPower(fdo, PowerDeviceD0, FALSE);
        if (!NT_SUCCESS(status))
            return CompleteRequest(Irp, status, 0);
    }

    IoMarkIrpPending(Irp); //--4
    StartPacket(&pdx->dqReadWrite, fdo, Irp, OnCancel);
    return STATUS_PENDING;
}
```

}

1. 这就是某驱动程序中用于处理 IRP_MJ_WRITE 请求的派遣函数。在这个例程的开始处是你应该调用 **PoSetDeviceBusy** 函数的地方之一，这个函数将重置发生于每一秒的空闲倒计时。
2. 你应该在设备不活动一段时期后降低设备的电源，或者当系统从 **standby** 中恢复时简单地关闭设备电源。无论什么原因，系统中没有任何程序会认为设备立刻会需要电源，因此你必须启动设备上电过程。
3. 如果设备的 **set-power** 请求因为某种原因失败，你应该失败这个写请求。
4. 这个派遣函数的剩下部分与前面章节中讨论的相同。我们标记这个 IRP 为 **pending**，把它放入写请求队列，并返回 **STATUS_PENDING** 以通知我们的调用者我们不必在派遣例程中结束该 IRP。

通常，我们在任意线程上下文中获得读写请求，所以我们不应该阻塞该线程。因此当我们打开设备的电源时，我们不用等待 **power-up** 操作完成就可返回。当电源完全恢复时，**DEVQUEUE** 将负责启动该请求。

SendDeviceSetPower 辅助函数将直接调用 **PoRequestPowerIrp**。其结果 IRP 将按我们曾经讨论过的方式处理。

用序列号优化状态改变

你可能希望在降低和恢复设备电源时使用优化技术。有两个背景因素会帮助你理解这种优化技术。第一，总线驱动程序并不总是降低设备电源，即使在它收到 **set-power** IRP 时。出现这种不协调情况的原因是计算机内部供电的连接方式。可能有一个或多个电源通道，而在任何给定的通道内都会连接任意组合的多个设备。这些设备共享一个电源连接。单个设备不能独自降低电源，除非在同一电源通道的所有其它设备同时降低电源。因此，我在这里举一个比较恐怖的例子，假定你要降低电源供应的 **modem** 设备与计算机控制的人工心肺机共享一个电源通道，那么系统就会拒绝降低 **modem** 的电源供应。

第二个背景因素是某些设备需要大量时间来改变电源状态。我们返回到前面的例子，假设你的 **modem** 就是一个这样的设备。在某一时刻，你的驱动程序收到并传递一个设备 **set-power** 请求，该请求将使 **modem** 进入睡眠状态。然而你并不知道总线驱动程序实际上并没有降低 **modem** 的电源供应。当到了恢复 **modem** 的电源供应时，如果你知道 **modem** 并没有被降低电源供应，你就可以节省一些时间，这就是这个优化所要针对的地方。

在你关闭设备电源时，你可以创建并发送一个带有 **IRP_MN_POWER_SEQUENCE** 副功能码的电源请求给下层驱动程序。尽管这个 IRP 在技术上是一个 **IRP_MJ_POWER** 请求，但你可以用 **IoAllocateIrp** 创建它。在处理该 IRP 时你还要使用 **PoStartNextPowerIrp** 和 **PoCallDriver** 函数。总线驱动程序在你提供的数组中保存了三个序列号后该请求才完成。序列号指出设备有多少次被置入 **D1**、**D2**、**D3** 状态。当你被调用恢复设备的电源时，你创建并发送另一个 **IRP_MN_POWER_SEQUENCE** 请求以获得一组新的序列号。如果这组新序列号与在 **power-down** 时获得的序列号相同，那么你就会知道状态改变没有发生，因此你可以跳过代价昂贵的电源恢复过程。

因为 **IRP_MN_POWER_SEQUENCE** 优化的过程在不优化的情况下也可以正常工作，所以你不用必须使用它。另外，总线驱动程序也不用必须支持它，并且你也不要将 **power-sequence** 请求的失败当作错误。光盘上的 **GENERIC** 例子包含了使用该优化方法的代码，但这里我不想再对状态机进行更复杂的文字讨论。

Windows 98 兼容问题

Windows 98 实现的许多电源管理特征都不完整。因此，Windows 98 环境对错误的检查要比 Windows 2000 宽松。这虽然促进了驱动程序的初始开发，但因为 Windows 98 容忍了许多 Windows 2000 不能容忍的错误，所以你必须确保在 Windows 2000 下测试驱动程序的所有电源管理功能。

DO_POWER_PAGABLE 的重要性

在 Windows 98 中，DO_POWER_PAGABLE 标志会有意想不到的重要性。除非设备堆栈的每个设备对象，包括 PDO 和所有过滤器设备都设置了这个标志，否则 I/O 管理会告诉 Windows 98 的配置管理器该设备仅支持 D0 电源状态并且不能唤醒系统。这样，忘记设置 DO_POWER_PAGABLE 标志的一个结果是，你通过 PoRegisterDeviceForIdleDetection 做的任何空闲通知请求都被完全忽略，即你将不能收到表示空闲超时的电源 IRP。另一个结果是，设备的唤醒特征将不被使用。

请求设备电源IRP

Windows 98 中的一个 bug 会使 PoRequestPowerIrp 成功，即它返回了 STATUS_PENDING，但你却不会收到设备 set-power 请求。当你发出一个 set-power 请求时，如果指定了设备当前的电源状态就会出现这样问题，Windows 98 的配置管理器认为没有新的情况发生，所以它没有发出一个配置事件来向 NTKERN 中的配置函数报告，而 NTKERN 负责处理 WDM 驱动程序这方面的事情。因此，如果你正等待一个设备 IRP 的完成，你的设备将简单地在这一点上停止响应。

我用了一个明显的 workaround 来克服这个问题：如果我们正运行在 Windows 98 下并且我们将要请求一个设备电源 IRP，而这个 IRP 中指定的电源状态与设备当前处于的电源状态相同，我将简单地假装该设备 IRP 已成功。根据 HandlePowerEvent 穿过的状态转换，我从 SendDeviceIrp 直接跳到适合的动作程序 (SubPowerUpComplete 或 SubPowerDownComplete)。

PoCallDriver

在 Windows 98 中，PoCallDriver 仅仅是调用 IoCallDriver。因此，你更容易犯下用 IoCallDriver 去下传电源 IRP 的错误。然而在 Windows 98 中，还有一个更差的问题正等着你。

Windows 2000 版本的 PoCallDriver 函数确保了在 PASSIVE_LEVEL 级上向有 DO_POWER_PAGABLE 标志的驱动程序发送电源 IRP，在 DISPATCH_LEVEL 级上向有 INRUSH 标志的驱动程序或处于非分页内存中的驱动程序发送电源 IRP。我利用了这个事实，GENERIC 就是在某 I/O 完成例程在 DISPATCH_LEVEL 级上调用 HandlePowerEvent 时下传了电源 IRP。在 Windows 98，由于 PoCallDriver 就是 IoCallDriver，并没有切换 IRQL。所以在 Windows 98 中所有电源 IRP 都是在 PASSIVE_LEVEL 级上发出的。因此我写了一个辅助函数 SafePoCallDriver 来帮助 GENERIC 排队可执行工作项(executive work item，参见第九章)，以使电源 IRP 能够在 PASSIVE_LEVEL 级上发出。

其它不同之处

你应该了解 Windows 98 与 Windows 2000 在处理电源管理特征上的其它不同之处。我简要地描述一下这些不同之处并指出它们如何影响驱动程序开发。

当你调用 PoRegisterDeviceForIdleDetection 时，你必须提供 PDO 的地址而不是自己设备对象的地址。这是因为，在内部，系统需要找到 Windows 98 配置管理器使用的 DEVNODE 对象的地址，并且该对象仅能通过 PDO 访问。在 Windows 2000 中你还可以把 PDO 作为该参数，所以你首先应该以这种方式写你的代码。

PoSetPowerState 支持例程在 Windows 98 中是一个空操作。另外，尽管该函数声称返回前一个设备或系统的电源状态，但该函数的 Windows 98 版本仅返回你提供的任何状态参数，这可能是一个新状态，与旧状态不同，或者仅仅是一个随机数。

PoStartNextPowerIrp 在 Windows 98 中也是一个空操作，所以当你在 Windows 98 中开发驱动程序时，你可能会很容易忘记调用过该函数。

处理设备电源关系的服务例程(**PoRegisterDeviceNotify** 和 **PoCancelDeviceNotify**)在 Windows 98 中没有定义。Windows 98 不能发出 PowerRelations 查询请求来收集支持回调函数的信息。服务例程 **PoRegisterSystemState**、**PoSetSystemState**、**PoUnregisterSystemState** 在 Windows 98 中也没有实现。为了装入使用了这些未定义服务函数的驱动程序，你需要提供一个带桩的虚拟设备驱动程序，参见附录 A，“对付 Windows 98 的不兼容问题”。

第九章：专门问题

在前八章中，我描述了能适合任何硬件设备的成熟 WDM 驱动程序的大部分特征。但你还应该了解一些更通用的技术，我将在本章描述这些技术。在本章的第一节，我将解释如何创建过滤器驱动程序。然后描述如何登记错误。之后，我将讨论一个相当重要的主题：**IOCTL(I/O 控制操作)**，它允许应用程序控制硬件或驱动程序的特征。这个讨论解释了 WMD 驱动程序如何向应用程序报告其“感兴趣”的事件。后面我还给出了如何创建系统线程的指导，如何排队工作项(**work item**)，使其在已存在的系统线程上下文中执行，以及如何为不回应的设备建立看门狗定时器。

- 过滤器驱动程序
- 登记错误
- I/O 控制操作
- 系统线程
- 工作项
- 看门狗定时器
- Windows 98 兼容问题

过滤器驱动程序

WDM 模型假定硬件设备可以有多个驱动程序，每个驱动程序都有自己管理设备的方法。WDM 根据设备对象堆栈来完成驱动程序的分层。到现在为止，我仅讨论了管理设备主要功能的功能驱动程序。在这一节中，我将描述如何写一个过滤器驱动程序，该驱动程序可位于功能驱动程序的上面或下面，它通过过滤流经它的 IRP 来修改设备的行为。

处于功能驱动程序之上的过滤器驱动程序称为上层过滤器；处于功能驱动程序之下（仍处于总线驱动程序之上）称为下层过滤器。虽然这两种驱动程序本身用于不同的目的，但创建这两种驱动程序的机制完全相同。实际上，创建过滤器驱动程序就象创建任何其它 WDM 驱动程序一样，都有 **DriverEntry** 例程、**AddDevice** 例程、一组派遣函数，等等。

上层过滤器驱动程序的用途是帮助支持这样的设备，这种设备的大多数方面都象其所属类的普通设备，但有一些附加功能。你可以依靠一个通用的功能驱动程序来支持设备的普通行为。为了处理设备的附加功能，你可以写一个上层过滤器驱动程序来干预 IRP 流。举一个有趣的例子，假设存在一个烤面包机设备的标准类，并且已经有人为其写了一个标准驱动程序。再假设你的特殊烤面包机有一个高级的面包片弹出特征，它可以把烤好的面包片弹到两英尺高的空中。而控制这个 AWE(Advanced Waffle Eject) 特征的工作就是上层过滤器驱动程序的任务，见图 9-1。

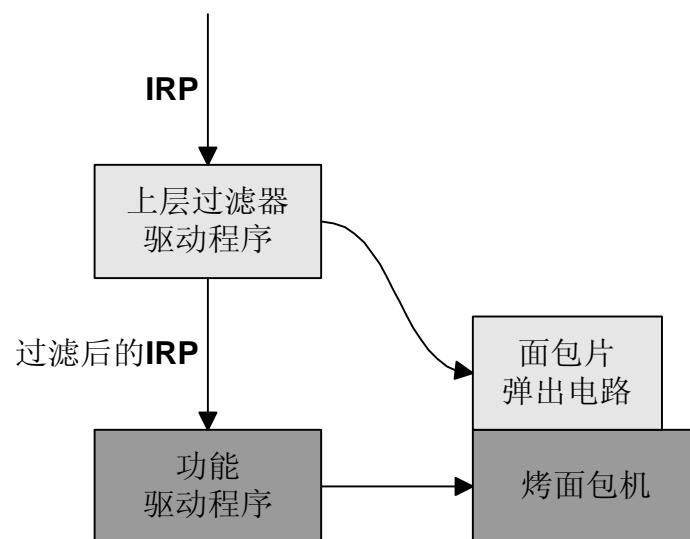


图 9-1. 上层过滤器驱动程序所扮演的角色

上层过滤器驱动程序的另一个用途是修正硬件或功能驱动程序中出现的 bug。如果过滤器驱动程序用于这个目的，Microsoft 恳求你在这个过滤器驱动程序上加上版本标签，并且如果在以后某一天这个 bug 被纠正，你应该在你的控制范围内修改任何相关部件的版本号。否则，Microsoft 将难于使系统自动更新。

下层过滤器驱动程序不能干涉功能驱动程序直接执行的正常操作。因为功能驱动程序可能通过 HAL 调用直接访问硬件来实现大部分实质的请求。而下层过滤器仅能看到经过它传递的 IRP，它看不到 HAL 调用。

下层过滤器驱动程序可以用于 USB 设备的驱动程序堆栈中。对于 USB 设备，其功能驱动程序把内部控制 IRP 作为 URB(USB 请求块)的容器。下层过滤器驱动程序可能会监视并修改这些 IRP，见图 9-2。

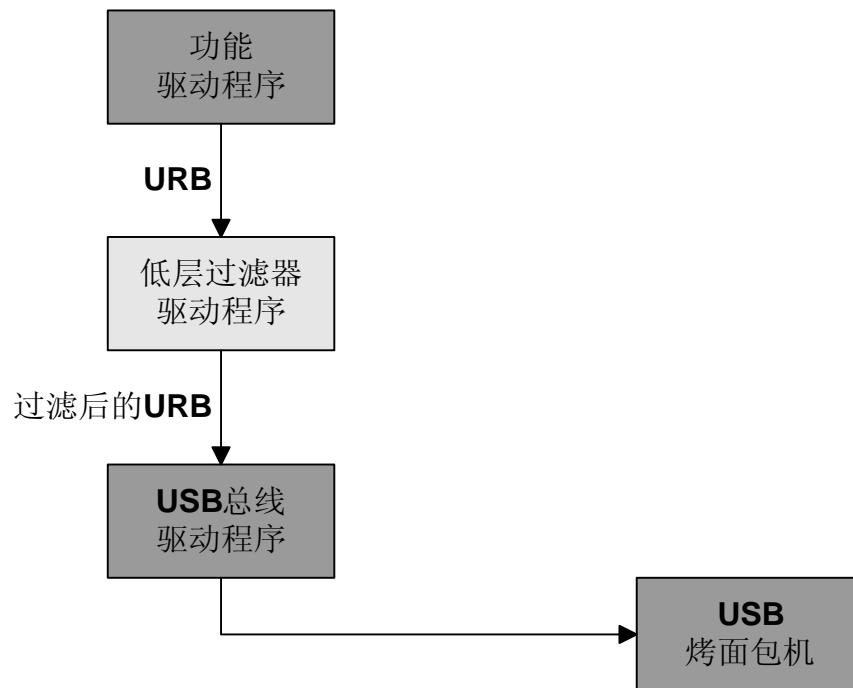


图 9-2. 下层过滤器驱动程序所扮演的角色

下层过滤器驱动程序的另一个用途是帮助你写一个总线无关的驱动程序。假设一种设备有三种不同总线形式的产品，PCI 总线产品、USB 总线产品、PCMCIA 总线产品。你可以写一个完全独立于总线结构的功能驱动程序，这样的驱动程序不直接与设备对话。另外你还要写三个下层过滤器驱动程序，每个下层过滤器对应一个总线类型，如图 9-3。当功能驱动程序需要与硬件对话时，它就向相应的下层过滤器驱动程序发送 IRP(可能是 IRP_MJ_INTERNAL_DEVICE_CONTROL)。

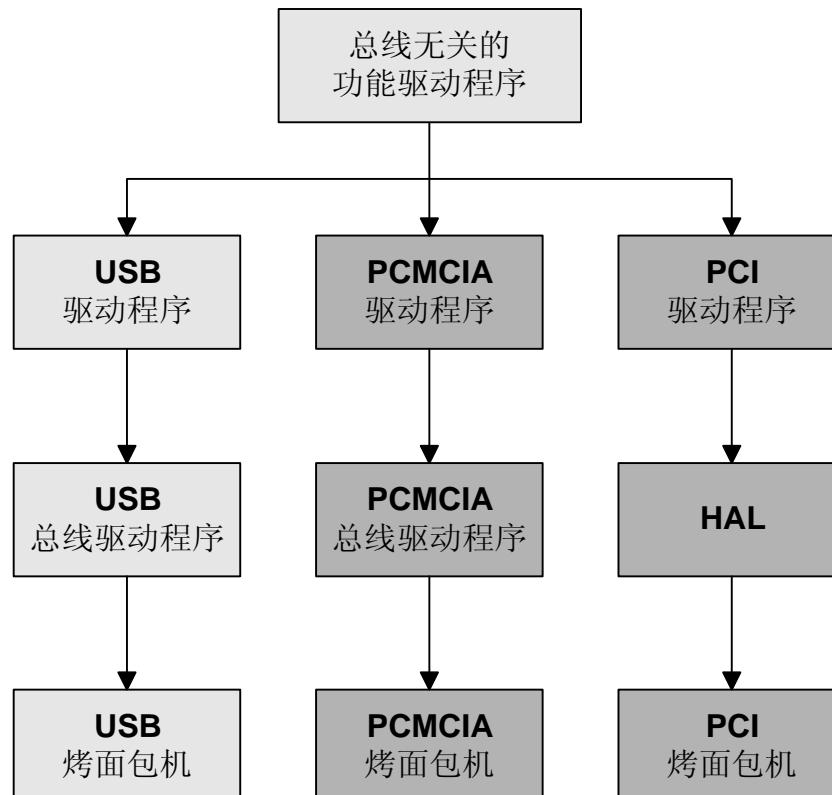


图 9-3. 用下层过滤器驱动程序完成总线无关性

DriverEntry 例程

过滤器驱动程序的 **DriverEntry** 例程与功能驱动程序的十分类似。主要的区别是，过滤器驱动程序必须为每种类型的 IRP 都安装派遣函数，不仅仅是安装其希望处理的 IRP 类型。

```
extern "C" NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
```

```

{
    DriverObject->DriverUnload = DriverUnload;
    DriverObject->DriverExtension->AddDevice = AddDevice;
    for (int i = 0; i < arraysize(DriverObject->MajorFunction); ++i)
        DriverObject->MajorFunction[i] = DispatchAny;
    DriverObject->MajorFunction[IRP_MJ_POWER] = DispatchPower;
    DriverObject->MajorFunction[IRP_MJ_PNP] = DispatchPnp;
    return STATUS_SUCCESS;
}

```

过滤器驱动程序与其它驱动程序一样，也有 **DriverUnload** 函数和 **AddDevice** 函数。我在主功能表中填入了一个名为 **DispatchAny** 函数的地址，该函数把所有随机请求继续传递下去。但我为电源管理请求和 PnP 请求指定了特殊的派遣函数。

过滤器驱动程序必须处理每种类型的 IRP 的原因是调用顺序，**AddDevice** 函数在 **DriverEntry** 中被调用。通常，过滤器驱动程序必须支持所有与紧在其下面的驱动程序支持的相同类型的 IRP。如果过滤器驱动程序使一个特别的 **MajorFunction** 表项处于默认状态，则该类型的 IRP 将会以 **STATUS_INVALID_DEVICE_REQUEST** 结果失败。(I/O 管理器含有一个默认的派遣函数，该函数就以这个结果简单地完成一个 IRP。系统最初送给驱动程序对象中的所有 **MajorFunction** 表项都指向这个默认例程) 但是直到 **AddDevice** 函数把设备对象放到你下面，你才知道这个原因。你可以在 **AddDevice** 函数中调查每个下层设备驱动程序，并在你自己的 **MajorFunction** 表中插入需要的派遣指针，但要记住你可能处于多设备堆栈中，因此 **AddDevice** 调用可能是重复的。在 **DriverEntry** 函数中声明支持所有 IRP 会比较容易。

AddDevice 例程

过滤器驱动程序有 **AddDevice** 函数，该函数被每个合适的硬件调用。你将调用 **IoCreateDevice** 函数创建一个未命名的设备对象，然后调用 **IoAttachDeviceToDeviceStack** 把该对象插入驱动程序堆栈。另外，你还需要从下层设备对象中复制出几个设置。

```

NTSTATUS AddDevice(PDRIVER_OBJECT DriverObject, PDEVICE_OBJECT pdo)
{
    PDEVICE_OBJECT fido;
    NTSTATUS status = IoCreateDevice(DriverObject,
                                    sizeof(DEVICE_EXTENSION),
                                    NULL,
                                    FILE_DEVICE_UNKNOWN,
                                    0,
                                    FALSE,
                                    &fido);
    if (!NT_SUCCESS(status))
        return status;
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
    __try
    {
        pdx->DeviceObject = fido;
        pdx->Pdo = pdo;
        PDEVICE_OBJECT fdo = IoAttachDeviceToDeviceStack(fido, pdo);
        pdx->LowerDeviceObject = fdo;
        fido->Flags |= fdo->Flags & (DO_DIRECT_IO|DO_BUFFERED_IO);
        fido->Flags |= DO_POWER_PAGABLE
        fido->DeviceType = fdo->DeviceType;
        fido->Characteristics = fdo->Characteristics;
        fido->Flags &= ~DO_DEVICE_INITIALIZING;
    }
    __finally

```

```

{
    if (!NT_SUCCESS(status))
        IoDeleteDevice(fido);
}
return status;
}

```

与功能驱动程序不同的地方使用了粗体显示。基本上，我们仅需要从下层设备对象中传递少数标志位、以及 **DeviceType** 值和 **Characteristics** 值。这样做的原因是因为 I/O 管理器在了解最上层设备对象时要基于这些信息。特别是判断一个读写 IRP 是使用 MDL 还是使用系统复制缓冲区，要取决于最上层设备对象是设置了 **DO_DIRECT_IO** 标志还是设置了 **DO_BUFFERED_IO** 标志。我们不必复制下层设备对象中的 **SectorSize** 或 **AlignmentRequirement** 成员，**IoAttachDeviceToDeviceStack** 将自动做这个工作。

换句话说，上层过滤器应该复制 **DO_DIRECT_IO** 和 **DO_BUFFERED_IO** 标志，并且总是设置 **DO_POWER_PAGABLE** 标志。即使下一层驱动程序有一个非分页的电源管理派遣例程，电源管理器也能处理有分页式派遣例程的 FiDO。但是，如果 FiDO 使用非分页派遣例程，那么低层驱动程序就不能有分页式派遣例程，否则将产生一个 bug check。

如果你正写一个下层过滤器驱动程序，可以不必关心怎样处理 **DO_DIRECT_IO** 和 **DO_BUFFERED_IO** 标志，因为它们可能是上面功能驱动程序的设置。如果上层驱动程序有一个非分页的电源管理派遣例程，那么下层驱动程序就不能设置 **DO_POWER_PAGABLE** 标志。如果下层驱动程序有分页式电源管理派遣例程，那么上层驱动程序不能清除 **DO_POWER_PAGABLE** 标志。遵循这个规则的唯一方法是了解你周围的驱动程序是怎样设置这些标志的，有两个地方需要注意：一、在初始化期间，二、在它们接收 **IRP_MN_DEVICE_USAGE_NOTIFICATION** 时。除了可读写磁盘设备的低级过滤器之外，你可以假定功能驱动程序使用分页式电源管理派遣例程，在这种情况下，**DO_POWER_PAGABLE** 标志对于上层过滤器才是正确的。但是，事先你必须和那个驱动程序的作者沟通或查看它的源代码以确定这一点。

不需要在过滤器中设置 **DO_POWER_INRUSH** 标志，因为堆栈中仅有一个设备对象需要这种标志。

[SP-3](#) 包含一个修订版的 FILTER，和实现了这些建议的 WDMWIZ 向导。

注意

由于过滤器驱动程序只能在功能驱动程序执行其 **AddDevice** 函数时复制这些设置，而且如果你的功能驱动程序后来改变了这些设置，过滤器驱动程序也没有办法知道。因此你需要在功能驱动程序的 **AddDevice** 函数中指明是 **buffered** 方式还是 **direct I/O** 方式，并且之后不要再改变。

通常，过滤器设备对象(FiDO)不必有自己的名称。如果功能驱动程序命名了它的设备对象并创建了符号连接，或者功能驱动程序为其设备读写寄存了设备接口，那么应用程序就能够打开该设备的句柄。每个发往该设备的 IRP 都先被发往最顶层的 FiDO 驱动程序，不管该 FiDO 是否有自己的名字。

创建 FiDO 对象时，不要使用 **FILE_DEVICE_SECURE_OPEN** 特征。PnP 管理器会在设备对象堆栈中散布这个标志和一些其它值。对一个文件打开操作进行强制安全检测并不是过滤器驱动程序的工作，而是功能驱动程序或总线驱动程序的工作。

派遣例程

写过滤器驱动程序的首要原因是你想以某种方式修改一个设备的行为。因此，需要用一些派遣例程来修改某些 IRP。但大部分 IRP 都被直接传递下去：

```

NTSTATUS DispatchAny(PDEVICE_OBJECT fido, PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
}

```

```

if (!NT_SUCCESS(status))
    return CompleteRequest(Irp, status, 0);
IoSkipCurrentIrpStackLocation(Irp);
status = IoCallDriver(pdx->LowerDeviceObject, Irp);
IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
return status;
}

NTSTATUS DispatchPnp(PDEVICE_OBJECT fido, PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
    if (!NT_SUCCESS(status))
        return CompleteRequest(Irp, status, 0);
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    ULONG fcn = stack->MinorFunction;
    IoSkipCurrentIrpStackLocation(Irp);
    status = IoCallDriver(pdx->LowerDeviceObject, Irp);
    if (fcn == IRP_MN_REMOVE_DEVICE)
    {
        IoReleaseRemoveLockAndWait(&pdx->RemoveLock, Irp);
        IoDetachDevice(pdx->LowerDeviceObject);
        IoDeleteDevice(fido);
    }
    else
        IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    return status;
}

NTSTATUS DispatchPower(PDEVICE_OBJECT fido, PIRP Irp)
{
    PoStartNextPowerIrp(Irp);
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
    if (!NT_SUCCESS(status))
        return CompleteRequest(Irp, status, 0);
    IoSkipCurrentIrpStackLocation(Irp);
    status = PoCallDriver(pdx->LowerDeviceObject, Irp);
    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    return status;
}

```

在这个例子中你可以看到，有必要为过滤器驱动程序的设备对象获取和释放删除锁。第一个 **IoAcquireRemoveLock** 调用用来检测当前 FiDO 是否有未处理的设备删除。如果是，派遣函数就立即以 STATUS_DELETE_PENDING 失败该 IRP，这是 **IoAcquireRemoveLock** 能返回的唯一的非成功值。当过滤器驱动程序在某个派遣函数中拥有自己的删除锁时，另一个正用 **DispatchPnp** 处理 IRP_MN_REMOVE_DEVICE 的线程将在其调用 **IoReleaseRemoveLockAndWait** 时被阻塞。因此应该防止调用 **IoDetachDevice**，它会使低层设备对象消失。我们自己的设备对象被一个引用计数保护起来，该引用计数是在这个 IRP 到来之前，我们的调用者通过 **IoGetAttachedDeviceReference** 获得的。

除了 IRP_MJ_PNP 请求，过滤器驱动程序中的所有派遣函数都必须在非分页内存中，并且不要假定会在 PASSIVE_LEVEL 级上被调用。这里有两个实际的例子，第一、一个 USB 设备的下层过滤器将接收并传递包含 URB 的 IRP_MJ_INTERNAL_DEVICE_CONTROL 请求。这些 IRP 中的某些会在 PASSIVE_LEVEL 级上到达，另一些会在 DISPATCH_LEVEL 级上到达，因为它们来自 I/O 完成例程。第二个例子是磁盘驱动程序，它会在 PASSIVE_LEVEL 级上开始处理电源管理请求，因为它设置了 DO_POWER_PAGABLE 标志。但磁盘驱动程序后来可能被通知其设备用于存放一个分页文件，或者是其它某些特殊文件，于是它锁定它的电源管理程

序并清除 `DO_POWER_PAGABLE` 标志。这样，在同一个堆栈上的所有过滤器驱动程序就突然都在 `DISPATCH_LEVEL` 级上获得电源管理请求。

注意

在编写一个过滤器驱动程序时，你应该遵守这样的方针：不要做有害的事情，即不要使在你之上的驱动程序和在你之下的驱动程序失败。

登记错误

到目前为止，我所讨论的错误处理仅涉及到检测(以及传播)状态代码，以及处理那些反映代码错误的辅助调试技术(**checked** 版本)。然而在驱动程序的 **free** 版本中，有些错误会严重到应该提请系统管理员注意。例如，一个磁盘驱动程序可能会发现磁盘的物理表面有大量的坏扇区。或者驱动程序遇到异常频繁的数据错误或某种难以配置或启动的设备。

为了处理这些情况，驱动程序应该向系统日志写入一条记录。而系统管理员可以使用 Windows 2000 系统中的管理工具，事件查看器(Event Viewer applet)查看所发生的问题。图 9-4 显示了事件查看器的工作窗口。指出意外错误的另一种方式是发出一个 WMI 事件。我将在这一节讨论事件登记。WMI 主题见第十章。

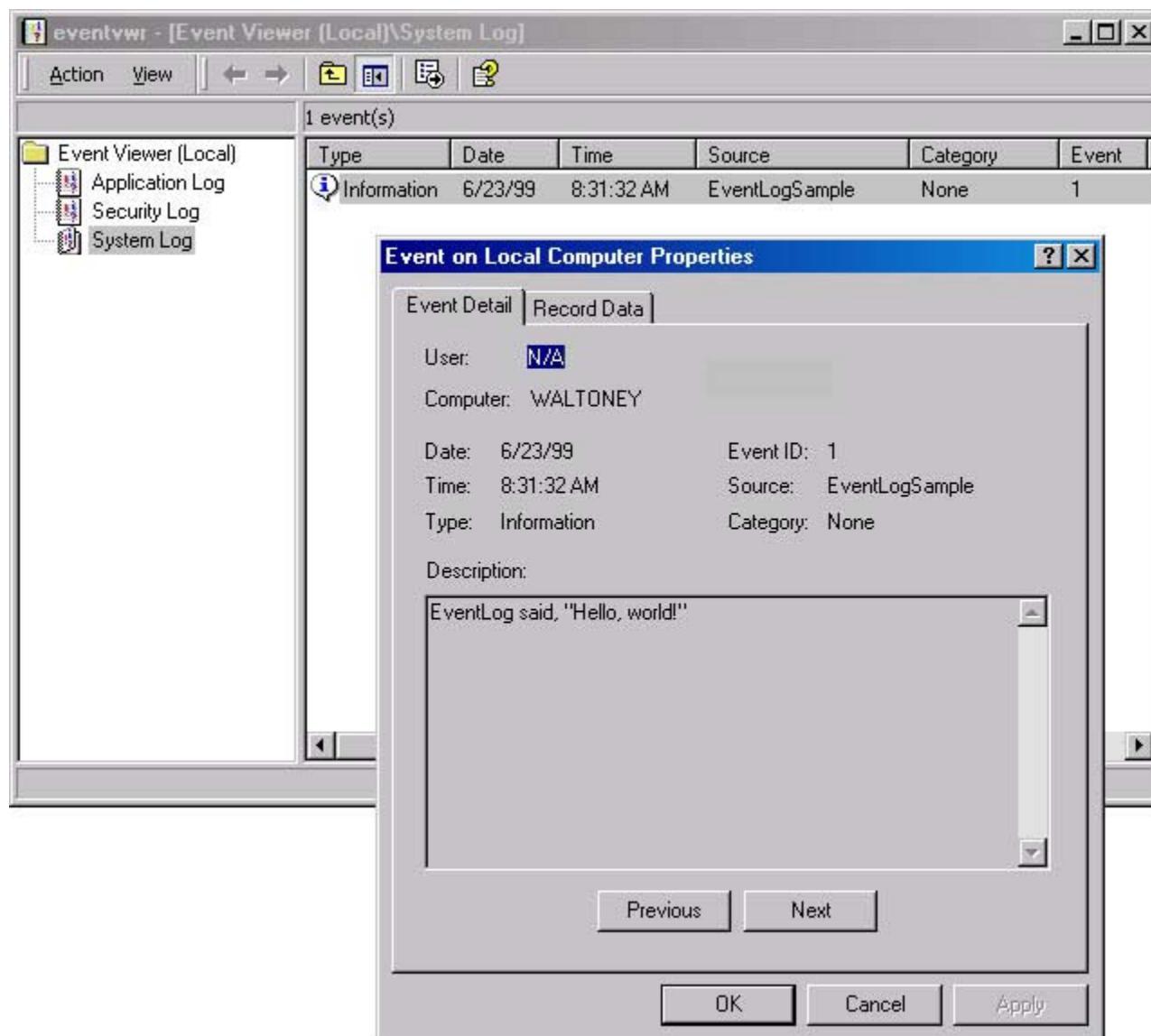


图 9-4. Windows 2000 的事件查看器

由错误记录产生管理报告的步骤见图 9-5。驱动程序使用内核模式服务函数 **IoWriteErrorLogEntry** 向事件登记服务器发送错误登记包(error log packet)。包中含有一个代替消息文本的数值代码。如果时间允许，事件登记程序就把这个包写入磁盘上的登记文件。然后，事件查看器把登记文件中的包和一组消息文件中的消息文本组合并生成管理报告。消息文件就是一个普通的 32 位 DLL，其中包含有用本地语言描述的对应所有可能登记事件的文本。

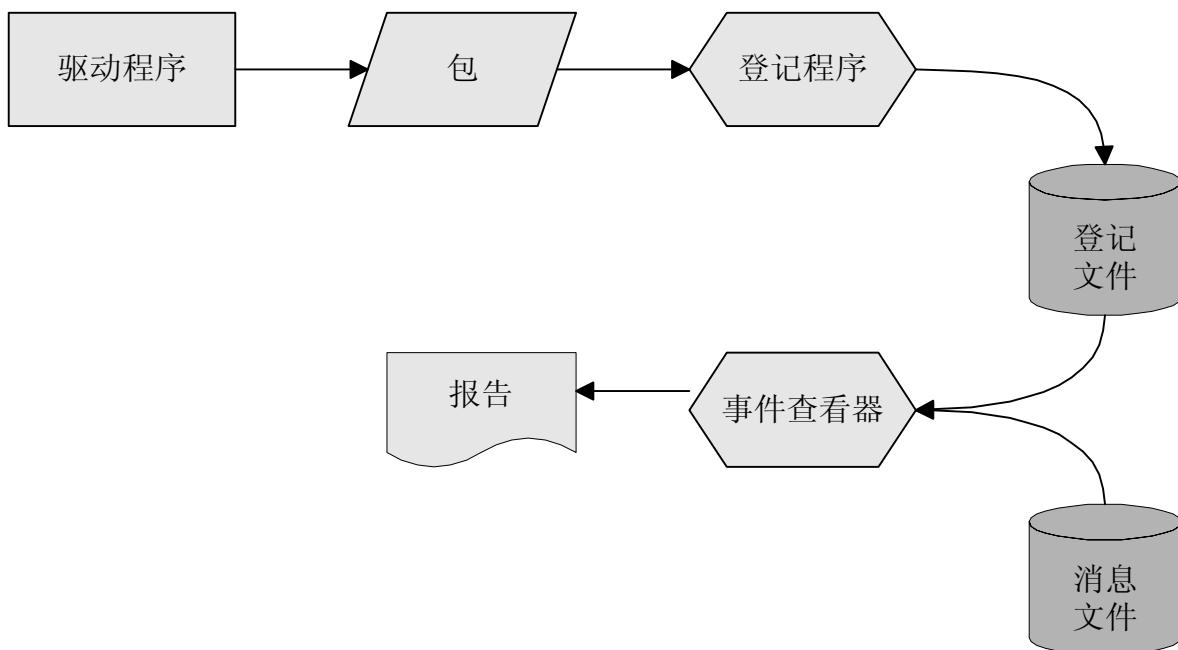


图 9-5. 事件的登记和报告过程

作为驱动程序的作者，当发生重要的事件时应创建适当的错误登记包。另外，你大概还需要以至少一种自然语言创建消息文件。

创建错误登记包

为登记一个错误，驱动程序需要创建一个 **IO_ERROR_LOG_PACKET** 结构，然后发往内核模式登记器。这个包是一个可变长结构，带有一个固定大小的头，头中包含有登记时的常规信息，见图 9-6。**ErrorCode** 指出登记的事件；它对应消息文件中的消息文本。在固定大小的头之后是一个名为 **DumpData** 的数组，数组元素为双字，长度为 **DumpAxisSize** 字节，当要求显示该事件的详细信息时，事件查看器将以十六进制形式显示这些信息。尽管该数组声明是一个 32 位整型数组，但数组大小的单位仍是字节。在 **DumpData** 之后是一些空结尾的 Unicode 串，这些串最终都会被事件查看器用格式化的消息文本所替代。串区域开始于包的 **StringOffset** 字节处，包含 **NumberOfStrings** 个字符串。

0	主功能代码	RetryCount	DumpAxisSize
4	NumberOfStrings		StringOffset
8	EventCategory		
C		ErrorCode	
10		UniqueErrorValue	
14		FinalStatus	
18		SequenceNumber	
1C		IoControlCode	
20		DeviceOffset	
28		DumpData[DumpAxisSize]	
		<string data>	

图 9-6. **IO_ERROR_LOG_PACKET** 结构

除了我刚提到的，你不必填写任何固定头成员，参考登记表项的诊断工具可以帮助你追查到问题所在。

由于登记包的长度是可变的，所以你的第一个任务就是确定被创建包需要多少内存。例如下面代码来自 **EVENTLOG** 例子中，它分配一个能容纳 4 字节 **dump** 数据再加上一个单串的错误登记包：

```
VOID LogEvent(NTSTATUS code, PDEVICE_OBJECT fdo)
{
```

```

PWSTR myname = L"EventLog";
ULONG packetlen = (wcslen(myname) + 1) * sizeof(WCHAR) + sizeof(IO_ERROR_LOG_PACKET) + 4;
if (packetlen > ERROR_LOG_MAXIMUM_SIZE)
    return;
PIO_ERROR_LOG_PACKET p = (PIO_ERROR_LOG_PACKET)
    IoAllocateErrorLogEntry(fdo, (UCHAR) packetlen);
if (!p)
    return;
...
}

```

这段代码中有一个容易出错的地方，错误登记包有 152 字节的最大长度，`ERROR_LOG_MAXIMUM_SIZE`。另外，`IoAllocateErrorLogEntry` 参数的大小是一个 `UCHAR`，它仅有 8 字节宽。我们可以请求一个长 400 字节的包，但实际上仅得到 144 字节。（400 等于 `0x190`，144 是 `0x90`，它是前一个数值被截掉 8 位后的结果）

注意 `IoAllocateErrorLogEntry` 的第一个参数是设备对象的地址。如果该设备对象有名称，则其名称会出现在登记表项中以替换%1 substitution 转义代码，关于转义代码将在下一节讨论。

这段代码还显示了你对分配一个登记表项出错时所做的反应：什么也不做。不能登记某些错误不被认为是一项错误，所以你不要失败任何 IRP，应该生成一个 `bug check` 或仅仅终止你的处理。实际上，你会注意到这个 `LogEvent` 辅助函数的返回值为 `VOID`，因为程序员不必关心该函数的成功或失败。

成功地分配登记包后，下一项工作是初始化该结构并把控制权交给登记器。例如：

```

...
memset(p, 0, sizeof(IO_ERROR_LOG_PACKET));
p->ErrorCode = code;

p->DumpDataSize = 4;
p->DumpData[0] = <whatever>;

p->StringOffset = sizeof(IO_ERROR_LOG_PACKET) + p->DumpDataSize;
p->NumberOfStrings = 1;
wcscpy((PWSTR)((PUCHAR)p + p->StringOffset), myname);

IoWriteErrorLogEntry(p);
}

```

登记设备错误时，应该在包头中填入更多的域，不仅仅是错误代码。对于这些域，参见 `IoAllocateErrorLogEntry` 函数的 DDK 文档。

创建消息文件

事件查看器使用包中的 `ErrorCode` 来定位消息文件中的对应消息文本，这个消息文件与你的驱动程序关联。消息文件就是一个 **DLL**，包含由一种或多种自然语言描述的文本资源。因为 **WDM** 驱动程序使用的可执行文件格式与 **DLL** 相同，所以你私有消息的消息文件就可以是驱动程序文件本身。这里我将介绍如何创建一个消息文件。在 MSDN 和 James D. Murray 的《Windows NT Event Logging (O'Reilly & Associates, 1998)》的第 125-57 页还可以找到更多的信息。

图 9-7 显示了添加消息文本到驱动程序文件中的过程。开始，你先创建一个扩展名为 **MC** 的消息源文件。消息编译器(**MC.EXE**)将把你写的脚本翻译成消息。消息编译器的一个输出是包含与消息对应的符号常量的头文件；的驱动程序应该包含这个头文件，而这些常量就是被登记事件的 `ErrorCode` 值。消息编译器的另一个输出是一组中间文件和一个资源文件(**.RC**)，中间文件包含以一种或多种自然语言描述的消息文本，而资源文件列出了这些中间文件。然后这些资源文件进入连接器，最后生成的驱动程序将包含支持事件查看器的消息资源。

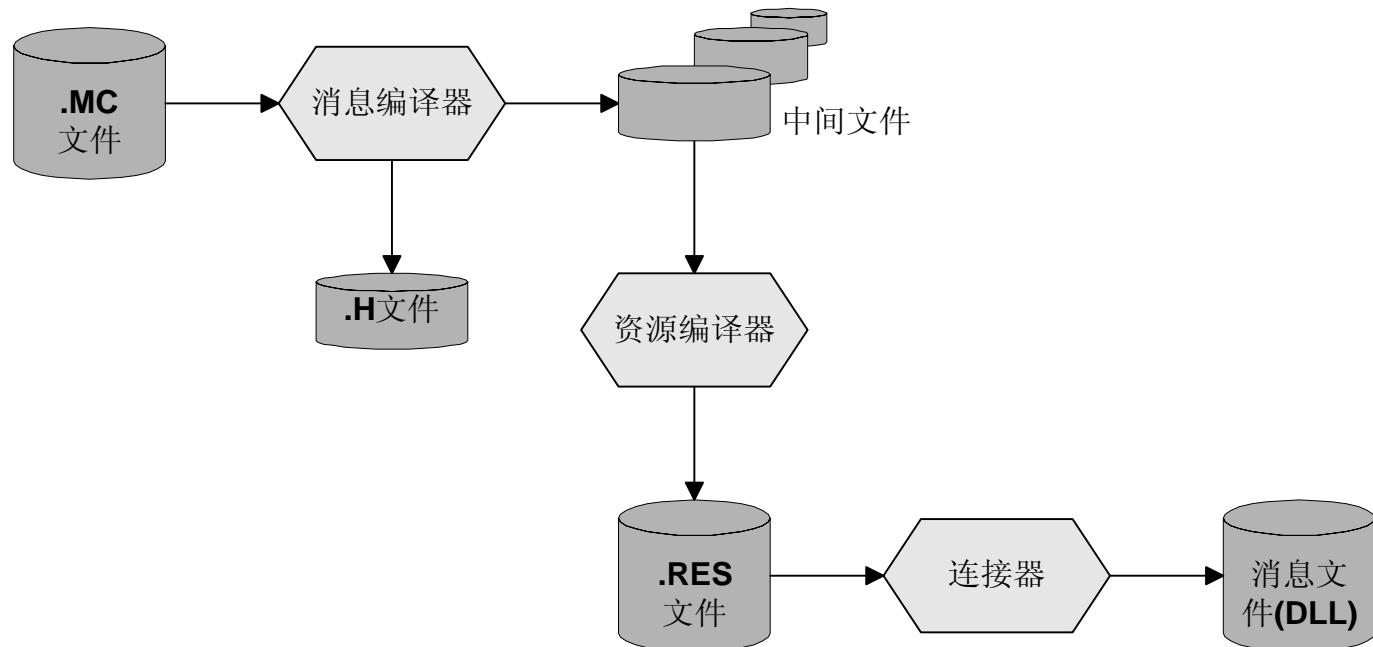


图 9-7. 消息文件的创建

下面是一个消息资源文件的例子。(是 EVENTLOG 例子的一部分)

```

MessageIdTypedef = NTSTATUS <

SeverityNames = ( <-2
    Success      = 0x0:STATUS_SEVERITY_SUCCESS
    Informational = 0x1:STATUS_SEVERITY_INFORMATIONAL
    Warning       = 0x2:STATUS_SEVERITY_WARNING
    Error         = 0x3:STATUS_SEVERITY_ERROR
)

FacilityNames = ( <-3
    System       = 0x0
    Eventlog     = 0x2A:FACILITY_EVENTLOG_ERROR_CODE
)

LanguageNames = ( <
    English      = 0x0409:msg00001
    German       = 0x0407:msg00002
    French       = 0x040C:msg00003
)

MessageId = 0x0001 <
Facility = Eventlog
Severity = Informational
SymbolicName = EVENTLOG_MSG_TEST
Language = English
%2 said, "Hello, world!" <

Language = German <
%2 hat gesagt, <<Wir sind nicht mehr im Kansas!>> <

Language = French <
%2 a dit, <<Mon chien a mangé mon devoir!>> <


```

1. **MessageIdTypedef** 语句指定一个符号，该符号可以作为消息标识符常量的强制类型转换操作符，消息标识符常量由这个消息文件生成。例如后面我们将用 EVENTLOG_MSG_TEST 符号名定义一个消息。**MessageIdTypedef** 语句的出现将使消息编译器在头文件中生成(**NTSTATUS**)0x602A0001L 符号定义。
2. **SeverityNames** 语句允许你定义自己的四种严重性代码名称。等号左边的名(Success、Informationaland, 等等)可以出现在这个文件的消息定义中。冒号后面的符号最后被定义成冒号前面的数字(在输出头文件中)。例如, **#define STATUS_SEVERITY_SUCCESS 0x0**。
3. **FacilityNames** 语句允许你定义自己的 Facility 代码, 这些代码将被包含在消息标识符定义中。我们将在后面的 Facility 语句中使用 Eventlog 名字。消息编译器把这三行的 FacilityNames 语句编译成**#define FACILITY_EVENTLOG_ERROR_CODE 0x2A**。
4. **LanguageNames** 语句允许你定义自己的自然语言名称。这里我们使用名称 English, 它对应 LANGID 0x0409, 即 Windows NT 语言方案中的标准英语。冒号之后的名字是中间二进制文件的名称, 该文件接收以这种自然语言编译的消息。
5. 每个单独的消息定义中都包含某些头文件语句。在这个例子中, **MessageId** 语句指定了一个绝对数字, 它还可以指定对上一消息的增量(如 **MessageId = +1**)。这里你可以指定 Facility 代码和严重性(使用前面定义的严重性名称)。你还可以用 **SymbolicName** 语句为该消息指定一个符号名。消息编译器将在它生成的头文件中定义这个符号。
6. 你可以为在 LanguageNames 语句中定义的每种语言定义一个像这样的消息文本。它开始于一个 **Language** 语句, 然后是该消息的文本。每个消息文本的定义都以一个仅包含一个点字符的单行结束。

在消息文本中, 我们可以用一个百分号再加一个整数来指出替代串。**%1** 引用生成该消息的设备对象名。这个名是创建错误登记表项时的潜在参数; 你不用直接指定这个名称。转义符号**%2**、**%3**, 等将对应添加到登记表项中的第一个、第二个, 等 Unicode 串。在上面的例子中, **%2** 将被 **EventLog** 替换。

这种指定替换串的方法可以使你自由地在文本中加入串。因此, 如果你的消息文本用英语时是“The %1 %2 fox jumped over the %3 dog”, 但用德语时就应该为“Der %3 Hund wurde vom %1 %2 Fuchs ubergesprungen”。(当然这仅是一个例子, 你的驱动程序必须提供“quick”、“brown”、“lazy”替换串, 这些替换串将以英文形式出现在该消息的各种显示版本中)

事件查看器需要参考某些注册表项来寻找消息文件。Windows NT 注册表的 services 分支(HKLM\System\CurrentControlSet\Services)中有一个名为 EventLog 的键, 每个驱动程序或登记了事件的其它服务都在该分支下有自己的子键。每个专用服务的子键都有名为 **EventMessageFile** 和 **TypesSupported** 的值。**EventMessageFile** 值为 REG_SZ 或 REG_EXPAND_SZ 类型, 它命名事件查看器需要访问的所有消息文件。这个值可能是象“%SystemRoot%\System32\iologmsg.dll; %SystemRoot%\System32\Drivers\EventLog.sys”这样的数据串。ILOGMSG.DLL 包含所有标准 NTSTATUS.H 代码的文本。参看下面文字框以了解如何在安装驱动程序时自动设置这些注册表项。**TypesSupported** 值只能是 REG_DWORD 类型, 应该等于 7, 以指出驱动程序可以生成所以可能的事件, 即错误、警告、普通消息。(实际上, 指定这个值可能是由于某个人为的历史原因)

关于消息文件

关于把消息资源放到驱动程序中, 有两个实际问题难于发现: 如何使 build 脚本能编译消息, 如何确信系统硬件安装程序把必要的表项放入了注册表。Art Baker 的《The Windows NT Device Driver Book: A Guide for Programmers (Prentice Hall, 1997)》在第 308 页间接地给出了第一个问题的解决办法。而 DDK 中关于 INF 文件的讨论解释了如何用 **AddService** 语句解决第二个问题。

就象本书中的其它例子程序, EVENTLOG 使用 Visual C++ 6.0 的工程文件。我修改了工程定义以包含 EVENTLOG.MC 的定制创建步骤。

在本书的后面章节中我将讨论如何用 INF 文件安装驱动程序。为了了解如何在一个 INF 文件中指定消息文件, 你可以先看看 EVENTLOG 工程目录中的 DEVICE.INF 文件, 特别是其中的 AddService 语句。你将看到 AddService 语句指向一个 **EventLogLogging** 段, 而该段又用 **AddReg** 语句指向一个 **EventLogAddReg** 段。而这个段将把 **EventMessageFile** 和 **TypesSupported** 值加入到事件登记器的服务专用子键上。

I/O控制操作

如果你观察发往设备的各种类型的请求，你会发现大部分请求都是读写数据。然而应用程序有时候会需要设备执行 IOCTL 操作，应用程序使用标准 Win32 API 函数 **DeviceIoControl** 来执行这样的操作。在驱动程序一方，这个 DeviceIoControl 调用被转化成一个带有 IRP_MJ_DEVICE_CONTROL 功能码的 IRP。

DeviceIoControl API

这个用户模式的 DeviceIoControl API 函数有下面原型：

```
result = DeviceIoControl(Handle,
                         Code,
                         InputData,
                         InputLength,
                         OutputData,
                         OutputLength,
                         &Feedback,
                         &Overlapped);
```

Handle(HANDLE) 是一个已打开的设备句柄。以下面这种方式调用 **CreateFile** 可以获得这个句柄：

```
Handle = CreateFile("\\\\.\\IOCTL",
                     GENERIC_READ | GENERIC_WRITE,
                     0,
                     NULL,
                     OPEN_EXISTING,
                     flags,
                     NULL);
if (Handle == INVALID_HANDLE_VALUE)
    <error>
...
CloseHandle(Handle);
```

CreateFile 的 **flags** 参数可以为 **FILE_FLAG_OVERLAPPED** 或 0，指出该文件句柄能否执行异步操作。句柄打开后，应用程序可以调用 **ReadFile**、**WriteFile**、**DeviceIoControl** 函数。完成设备的访问后，应用程序可以调用 **CloseHandle** 函数明确地关闭句柄。要记住，操作系统在进程终止后会自动关闭所有仍打开的句柄。

DeviceIoControl 的 **Code(DWORD)** 参数是一个控制代码，它指出应用程序要执行何种控制操作。稍后我再讨论如何定义这些代码。**InputData(PVOID)** 和 **InputLength(DWORD)** 参数描述了为驱动程序提供数据的缓冲区(以驱动程序的角度看，该区域为驱动程序提供输入数据)。**OutputData(PVOID)** 和 **OutputLength(DWORD)** 参数描述了接受驱动程序输出数据的缓冲区。驱动程序会更新变量 **Feedback(DWORD)** 以指出它返回了多少字节的输出数据。图 9-8 显示了应用程序、驱动程序和这些缓冲区之间的关系。**Overlapped(OVERLAPPED)** 结构用于帮助控制异步操作，详细描述见后面段。如果你在 **CreateFile** 调用中指定了 **FILE_FLAG_OVERLAPPED** 标志，那么你必须指定 **OVERLAPPED** 结构指针。如果没有指定 **FILE_FLAG_OVERLAPPED** 标志，你应该在这个最后的参数中给出 **NULL** 值。

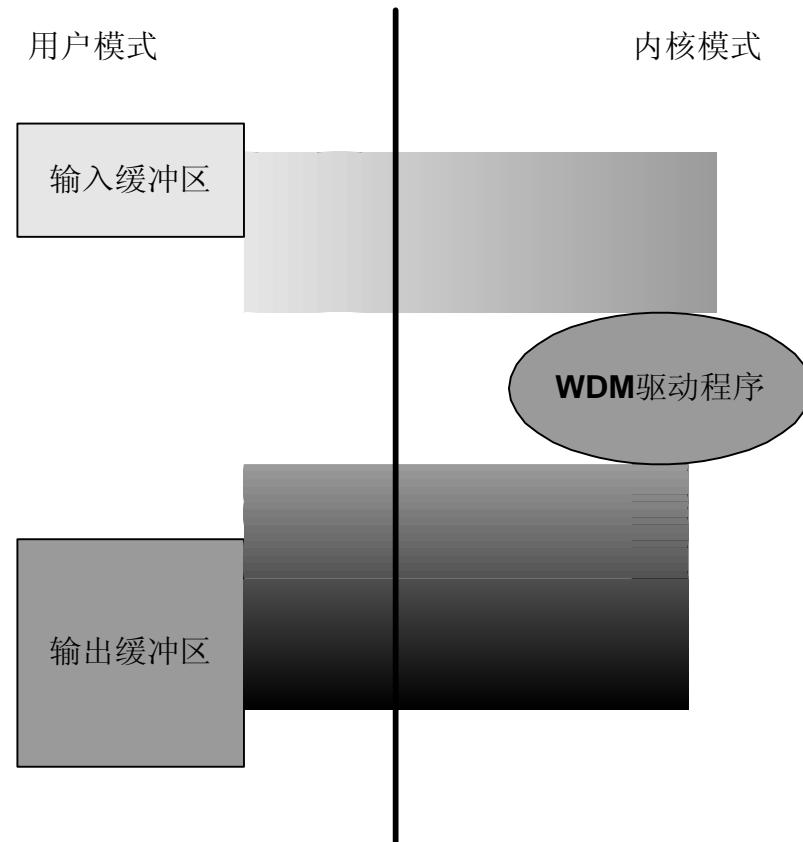


图 9-8. *DeviceIoControl* 函数的输入输出缓冲区

一个控制操作是否需要输入输出缓冲区要取决于其执行的功能。例如，一个接收驱动程序版本号的 IOCTL 可能仅需要输出缓冲区。一个仅用于通知驱动程序属于应用程序的某些事实的 IOCTL 可能仅需要输入缓冲区。有些操作可能同时需要这两种缓冲区，这完全取决于控制操作的具体内容。

DeviceIoControl 的返回值是 Boolean 值。如果返回 FALSE，则应用程序可以调用 **GetLastError** 函数查找调用失败的原因。

DeviceIoControl的同步和异步调用方式

当以同步方式调用 *DeviceIoControl* 时，调用线程将被阻塞直到控制操作完成。例如：

```
HANDLE Handle = CreateFile("\\\\.\\IOCTL", ..., 0, NULL);
DWORD version, junk;
if (DeviceIoControl(Handle,
                    IOCTL_GET_VERSION_BUFFERED,
                    NULL,
                    0,
                    &version,
                    sizeof(version),
                    &junk,
                    NULL))
    printf("IOCTL.SYS version %d.%2d\n", HIWORD(version), LOWORD(version));
else
    printf("Error %d in IOCTL_GET_VERSION_BUFFERED call\n", GetLastError());
```

这里，我们以不指定 **FILE_FLAG_OVERLAPPED** 标志的情况下打开设备句柄。因此，在这之后对 *DeviceIoControl* 的调用将不返回，直到驱动程序对我们的请求做出回答。

当以异步方式调用 *DeviceIoControl* 函数时，调用线程不立即阻塞。相反，它继续执行直到到达某一点，而该点的执行需要控制操作的结果。在这一点上，调用者调用某个阻塞线程的 API 函数，线程即被阻塞直到驱动程序完成该操作。例如：

```

HANDLE Handle = CreateFile("\\\\.\IOCTL", ..., FILE_FLAG_OVERLAPPED, NULL);
DWORD version, junk;
OVERLAPPED Overlapped;

Overlapped.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
DWORD code;

if (DeviceIoControl(Handle, ..., &Overlapped))
    code = 0;
else
    code = GetLastError();

<continue processing>

if (code == ERROR_IO_PENDING)
{
    if (GetOverlappedResult(Handle, &Overlapped, &junk, TRUE))
        code = 0;
    else
        code = GetLastError();
}
CloseHandle(Overlapped.hEvent);
if (code != 0)
    <error>

```

这个异步例子与前面的同步例子有两个主要的不同之处。第一，在调用 `CreateFile` 时指定了 `FILE_FLAG_OVERLAPPED` 标志。第二，对 `DeviceIoControl` 的调用中指定了一个 `OVERLAPPED` 结构的地址，在这个结构中有一个 `hEvent` 事件句柄，而我们把该事件初始化成一个手动重置事件。关于事件和线程同步的全面信息见 Jeffrey Richter 的《Programming Applications for Microsoft Windows, Fourth Edition [Microsoft Press, 1999]》

对 `DeviceIoControl` 的异步调用将产生三种结果。第一，它可能返回 `TRUE`，代表设备驱动程序的派遣例程能够立刻完成请求。第二，它可能返回 `FALSE`，并且用 `GetLastError` 会得到一个特殊的错误代码 `ERROR_IO_PENDING`。这个结果指出驱动程序的派遣例程返回了 `STATUS_PENDING` 并且推迟完成这个控制操作。注意那个 `ERROR_IO_PENDING` 并不是一个真正的错误，这是系统表明执行正常的两种方式中的一种。第三个可能结果是返回值为 `FALSE`，`GetLastError` 得到的值不是 `ERROR_IO_PENDING`。这样的结果才是一个真正的错误。

在应用程序需要这个控制操作的结果的那一点上，它调用一个 Win32 同步原语，如 `GetOverlappedResult`、`WaitForSingleObject`、或其它相类似的函数。我在这个例子中使用的是 `GetOverlappedResult` 同步原语，这个函数使用起来特别方便，因为它同时还提供传输字节量值并设置 `GetLastError` 结果。虽然你可以调用 `WaitForSingleObject` 或其它相关的 API 函数，并把 `Overlapped.hEvent` 事件句柄作为一个参数传递过去，但这样你不能了解到 `DeviceIoControl` 的操作结果；你仅仅知道该操作已经完成。

定义 I/O 控制代码

`DeviceIoControl` 的 `Code` 参数是一个 32 位数值常量，图 9-9 显示了操作系统分割这个 32 位代码的一种方式。

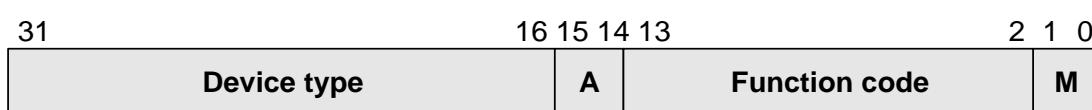


图 9-9. I/O 控制代码中的域

这些域的解释如下：

- Device type(16 位, CTL_CODE 宏的第一个参数) 指出能实现这个控制操作的设备类型。我并不知晓 Windows 98 或 Windows 2000 中的任何“IOCTL police”，但是我认为该域的内容实际上是任意的。习惯上我们使用与驱动程序中调用 IoCreateDevice 时使用的同一个值(例如, FILE_DEVICE_UNKNOWN)。
- 访问代码("A"占 2 位, CTL_CODE 的第四个参数) 指出使用设备句柄发出这个控制操作时, 应用程序必须具有的访问权限。
- Function code(12 位, CTL_CODE 的第二个参数) 指出该代码描述的是哪一个控制操作。Microsoft 保留了该域的前半部分, 从 0 到 2047 的值。因此我们只能使用从 2048 到 4095 之间的值。
- 缓冲方式("M"占两位, CTL_CODE 的第三个参数) 指出 I/O 管理器如何处理应用程序提供的输入输出缓冲区。在后面段中当讲述如何实现 IRP_MJ_DEVICE_CONTROL 时, 我会更详细地讲述这个域。

需要阐明一点, 编写驱动程序时, 你可以自由地设计一系列 IOCTL 操作, 应用程序也可以使用这些 IOCTL 操作与你的驱动程序对话。虽然有时你定义的 IOCTL 操作在数值上与其它人定义的完全相同, 但系统不会被这种重叠所迷惑, 因为你定义的 IOCTL 代码仅被你的驱动程序解释。

通常, 如果应用程序开发者需要调用你的驱动程序, 简单的办法是把你所有的 IOCTL 定义都放到一个专用的头文件中。光盘上的每个例子驱动程序工程都有一个名为 IOCTLS.H 的头文件, 如下例:

```
#ifndef CTL_CODE
    #pragma message ("CTL_CODE undefined. Include winioctl.h or wdm.h")
#endif

#define IOCTL_GET_VERSION_BUFFERED \
    CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_GET_VERSION_DIRECT \
    CTL_CODE(FILE_DEVICE_UNKNOWN, 0x801, METHOD_OUT_DIRECT, FILE_ANY_ACCESS)
#define IOCTL_GET_VERSION_NEITHER \
    CTL_CODE(FILE_DEVICE_UNKNOWN, 0x802, METHOD_NEITHER, FILE_ANY_ACCESS)
```

#pragma 语句指出未找到 CTL_CODE 定义时的警告信息, 应该包含 winioctl.h 或 wdm.h 头文件。

处理IRP_MJ_DEVICE_CONTROL

在用户模式中对 DeviceloControl 的调用将使 I/O 管理器创建一个带有 IRP_MJ_DEVICE_CONTROL 主功能码的 IRP, 并把该 IRP 发送到设备堆栈最上层驱动程序的派遣例程。最上层的堆栈单元包含表 9-1 中列出的参数。过滤器驱动程序会解释某些它们自己的代码, 然后把其它 IRP 下传。一个了解如何处理 IOCTL 的派遣函数将驻留在驱动程序堆栈的某个地方, 最可能是在功能驱动程序中。

表 9-1. IRP_MJ_DEVICE_CONTROL 的堆栈单元参数

Parameters.DeviceloControl 域	描述
OutputBufferLength	输出缓冲区的长, DeviceloControl 的第六个参数
InputBufferLength	输入缓冲区的长, DeviceloControl 的第四个参数
IoControlCode	控制代码, DeviceloControl 的第二个参数
Type3InputBuffer	输入缓冲区的用户模式虚拟地址, METHOD_NEITHER 模式

控制操作的派遣函数的基本结构如下:

```
#pragma PAGEDCODE

NTSTATUS DispatchControl(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PAGED_CODE();
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
```

```

NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp); <
if (!NT_SUCCESS(status))
    return CompleteRequest(Irp, status, 0);
ULONG info = 0;

PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp); <
ULONG cbin = stack->Parameters.DeviceIoControl.InputBufferLength;
ULONG cbout = stack->Parameters.DeviceIoControl.OutputBufferLength;
ULONG code = stack->Parameters.DeviceIoControl.IoControlCode;

switch (code)
{
    ...
}

default:
    status = STATUS_INVALID_DEVICE_REQUEST;
    break;

}

IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
return CompleteRequest(Irp, status, info);
}

```

1. 你将在 **PASSIVE_LEVEL** 级上被调用，因此这个派遣函数可以在分页内存中。
2. 象其它派遣函数一样，该派遣函数在工作时需要一个删除锁。它可以防止设备对象由于一个 **PnP** 事件而意外消失。
3. 下几行语句从 **I/O** 堆栈的 **parameters** 联合中提取功能码和缓冲区大小值。不管处理哪个 **IOCTL**，你都会需要这些值，所以我总是在函数的前面加入这些语句。
4. 在这里你可以对你支持的每个 **IOCTL** 加入一个 **case** 语句。
5. 如果你得到了一个你不认识的 **IOCTL** 操作，返回一个有意义的状态代码。

处理每个 **IOCTL** 需要依靠两个因素。第一，也是最重要的，是该 **IOCTL** 的真正目的。第二，是你选择的缓冲用户模式数据的方式。

在第七章中，我讨论了如何与用户模式程序交换数据。那时我指出，当读写请求到来时，你必须记住在 **AddDevice** 中指出的访问用户模式缓冲区所使用的模式，是 **buffered** 模式还是 **direct** 模式(或者两者都不是)。控制请求也利用这些寻址方式，但有一些差异。不是用设备对象中的标志来指定全局寻址方式，而是用 **IOCTL** 中的功能码的低两位来为每个 **IOCTL** 指定寻址方式。因此，有些 **IOCTL** 使用 **buffered** 方式，而有些 **IOCTL** 使用 **direct** 方式，还有些 **IOCTL** 使用 **neither** 方式。此外，**IOCTL** 中指定的缓冲方式并不影响普通读写 **IRP** 的寻址缓冲区。

选择哪个缓冲方式基于下面几个因素。大部分 **IOCTL** 操作传输的数据少于一个页，应该使用 **METHOD_BUFFERED** 方式。传输超过一个页数据的操作应该使用 **direct** 方式。如果你知道你将在应用程序的线程中获得控制，这对于 **IOCTL** 通常是对的，因为你上面没有过滤器驱动程序需要挂起，你可以使用 **METHOD_NEITHER** 方式任意访问用户模式数据。

BUFFERED模式

使用 **METHOD_BUFFERED** 方式时，**I/O** 管理器创建一个足够大的内核模式拷贝缓冲区(与用户模式输入和输出缓冲区中最大的容量相同)。当派遣例程获得控制时，用户模式的输入数据被复制到这个拷贝缓冲区。在 **IRP** 完成之前，你应该向拷贝缓冲区填入需要发往应用程序的输出数据。当 **IRP** 完成时，你应该设置 **IoStatus.Information** 域等于放入拷贝缓冲区中的输出字节数。然后 **I/O** 管理器把数据复制到用户模式缓冲区并设置反馈变量。图 9-10 显示了这些复制操作。

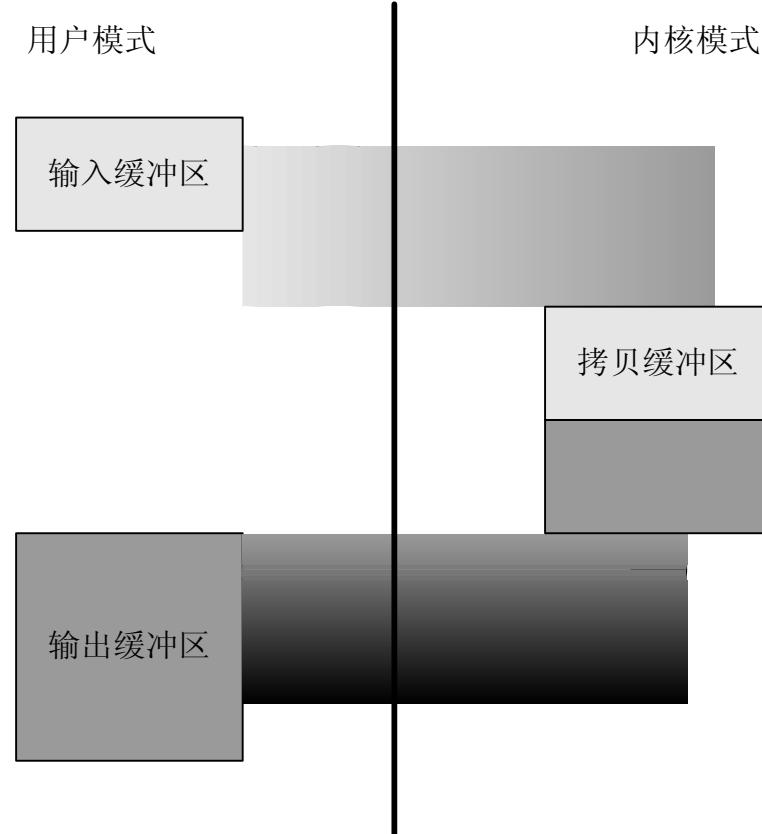


图 9-10. *METHOD_BUFFERED* 方式的缓冲区管理

在驱动程序内部，你以同一个地址访问这两个缓冲区，即 IRP 中的 **AssociatedIrp.SystemBuffer** 指针。这是一个内核模式虚拟地址，指向输入数据的一份拷贝。很明显，你应该在完成输入数据的处理之后再向这个缓冲区写入输出数据。

这里有一个简单的例子，是 IOCTL 例子中用于处理一个 *METHOD_BUFFERED* 操作的代码：

```
case IOCTL_GET_VERSION_BUFFERED:
{
    if (cbout < sizeof(ULONG))
    {
        status = STATUS_INVALID_BUFFER_SIZE;
        break;
    }
    PULONG pversion = (PULONG) Irp->AssociatedIrp.SystemBuffer;
    *pversion = 0x0004000A;
    info = sizeof(ULONG);
    break;
}
```

我们首先要确认得到的输出缓冲区的长度至少能容纳一个双字。然后我们使用 **SystemBuffer** 指针寻址系统拷贝缓冲区，在系统拷贝缓冲区中我们存入这个简单操作的结果。当外围的派遣例程完成这个 IRP 时，局部变量 **info** 中的值将写入 **IoStatus.Information** 域。I/O 管理器将把系统拷贝缓冲区中的数据复制回用户模式输出缓冲区。

一个安全漏洞？

我总是对 IOCTL 功能代码中 **buffering** 和 **access-control** 的重要性感到不安。假定某个有恶意的应用程序在提交一个 IOCTL 时使用了另外的标志值。这会造成驱动程序失败或其它不应该发生的事情吗？通常是不会的。

大部分 IOCTL 请求的派遣函数带有一个 **switch** 语句。**case** 语句所指定的数值常量必须与应用程序提供的 32 位代码精确匹配。所以，如果应用程序改变了某个 IOCTL 中的任何位，驱动程序中就没有与之匹配的 **case** 语句，那么某个默认动作将发生。

DIRECT模式

在驱动程序中，METHOD_IN_DIRECT 和 METHOD_OUT_DIRECT 模式都以相同方式处理。仅有的不同是它们访问用户模式缓冲区时所需的访问权限；METHOD_IN_DIRECT 需要读权限；METHOD_OUT_DIRECT 既需要读权限又需要写权限。使用这两种模式时，I/O 管理器会为输入数据提供一个内核模式拷贝缓冲区（AssociatedIrp.SystemBuffer），为输出数据缓冲区提供一个 MDL。MDL 细节见第七章，图 9-11 描述了这种缓冲区管理方法。

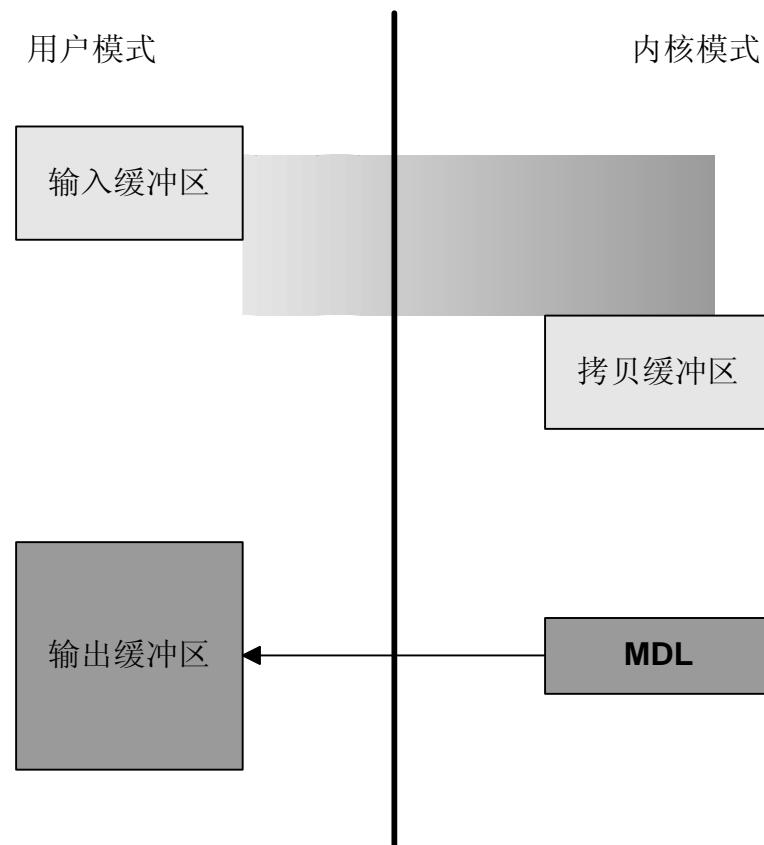


图 9-11. METHOD_XXX_DIRECT 方式的缓冲区管理

这里有一个处理 METHOD_XXX_DIRECT 请求的简单例子：

```
case IOCTL_GET_VERSION_DIRECT:  
{  
    if (cbout < sizeof(ULONG))  
    {  
        status = STATUS_INVALID_BUFFER_SIZE;  
        break;  
    }  
    PULONG pversion = (PULONG) MmGetSystemAddressForMdl(Irp->MdlAddress);  
    *pversion = 0x0004000B;  
    info = sizeof(ULONG);  
    break;  
}
```

这个例子与前一个例子的实质性区别是粗体字部分(我还改变了返回的版本号，以便能方便地知道我正从 test 程序调用正确的 IOCTL)。对于任何一种 DIRECT 模式的请求，我们使用 MDL(由 IRP 中的 **MdlAddress** 域指向)来访问用户模式输出缓冲区。你可以用这个地址来做 DMA 传输。在这个例子中，我仅调用了 **MmGetSystemAddressForMdl** 函数来获得一个内核模式别名地址，该地址指向由 MDL 描述的物理内存。

NEITHER模式

使用 **NEITHER** 模式时，I/O 管理器不翻译用户模式的虚拟地址。你得到输入缓冲区的用户模式虚拟地址(在堆栈单元的 **Type3InputBuffer** 参数中)，和输出缓冲区的用户模式虚拟地址(在 IRP 的 **UserBuffer** 域)。除非你知道你与用户模式调用者运行在同一个进程上下文中，否则这些地址是无用的。如果你恰好知道你正运行在这样的进程上下文中，你可以直接使用这些指针：

```
case IOCTL_GET_VERSION_NEITHER:  
{  
    if (cbout < sizeof(ULONG))  
    {  
        status = STATUS_INVALID_BUFFER_SIZE;  
        break;  
    }  
    PULONG pversion = (PULONG) Irp->UserBuffer;  
    if (Irp->RequestorMode != KernelMode)  
    {  
        __try  
        {  
            ProbeForWrite(pversion, sizeof(ULONG), 1);  
            *pversion = 0x0004000A;  
        }  
        __except(EXCEPTION_EXECUTE_HANDLER)  
        {  
            status = GetExceptionCode();  
            break;  
        }  
    }  
    else  
    {  
        *pversion = 0x0004000A;  
        info = sizeof(ULONG);  
        break;  
    }  
}
```

如上面粗体部分代码所示，这里唯一的毛病是你希望对这个缓冲区的写操作是正确的，而这个缓冲区来自一个未被确认的源。**ProbeForWrite** 是一个标准的内核模式服务例程，用于测试一个给定的用户模式虚拟地址是否可写。第二个参数指出你要探测的数据区域的长度，第三个参数指出数据区域需要的对齐值。在这个例子中，我们希望能正确地写入四个字节，我们愿意忍受数据区域本身的单字节对齐要求。**ProbeForWrite**(或**ProbeForRead**)实际上是测试给定地址范围是否有正确的对齐值，以及该地址空间中的用户模式部分，它并没有真的写或读被询问的内存。

你不要用我讲的这些方式直接访问用户模式内存，因为在同一个进程中的其它线程会在 **ProbeForXxx** 调用和你访问实际内存之间调用 **VirtualFree** 函数释放掉这段内存。所以你应该总是创建一个 **MDL** 并调用 **MmGetSystemAddressForMdl** 来获得一个安全的虚拟地址。实际上，更安全的办法是直接访问用户模式指针，但必须满足三个条件：第一，你必须运行在缓冲区所属于的进程上下文中。第二，你必须完成一次对 **ProbeForXxx** 的调用，第三，你必须在结构化异常帧中执行内存访问。如果在内存访问时缓冲区的某部分仍属于不存在的页上，则内存管理器将产生一个异常而不是 **bug-check**。你的异常句柄将捕获这个异常并阻止系统崩溃。

内部I/O控制操作

系统使用 **IRP_MJ_DEVICE_CONTROL** 来实现用户模式的 **DeviceIoControl** 调用。有时驱动程序需要相互交谈，它们可以使用 **IRP_MJ_INTERNAL_DEVICE_CONTROL**。下面是一个典型的例子代码：

```
ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);  
KEVENT event;  
KeInitializeEvent(&event, NotificationEvent, FALSE);  
IO_STATUS_BLOCK iostatus;
```

```

PIRP Irp = IoBuildDeviceIoControlRequest(IoControlCode,
                                         DeviceObject,
                                         pInBuffer,
                                         cbInBuffer,
                                         pOutBuffer,
                                         cbOutBuffer,
                                         TRUE,
                                         &event,
                                         &iostatus);

if (IoCallDriver(DeviceObject, Irp) == STATUS_PENDING)
    KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);

```

必须在 **PASSIVE_LEVEL** 级上调用 **KelInitializeEvent** 和 **IoBuildDeviceIoControlRequest** 函数，同样，在事件对象上阻塞也需要在 **PASSIVE_LEVEL** 级。

IoBuildDeviceIoControlRequest 的 **IoControlCode** 参数是一个控制代码，它代表你要求目标设备驱动程序执行的操作。这个代码与常规控制操作中使用的是同一种代码。**DeviceObject** 指向一个设备对象，该设备对象的驱动程序将执行指定的操作。输入和输出缓冲区参数与用户模式 **DeviceIoControl** 调用中对应的参数含义相同。第七个参数，我在这里把它指定为 **TRUE**，它指出你创建的是一个内部控制操作。(创建 **IRP_MJ_DEVICE_CONTROL** 时该参数应为 **FALSE**)

IoBuildDeviceIoControlRequest 将创建一个 **IRP** 并用操作代码和缓冲区初始化该 **IRP** 的第一个堆栈单元。它返回该 **IRP** 的指针，以便你能做额外的初始化工作。例如，在第十一章中我将讲述如何使用内部控制请求向 **USB** 总线驱动程序提交 **URB**。该过程的一部分就是设置堆栈参数域，使其指向 **URB**。然后调用 **IoCallDriver** 来发送该 **IRP** 到达目标设备。无论返回什么值，你都将等待在 **IoBuildDeviceIoControlRequest** 函数的第八个参数指定的 **event** 对象上。当该 **IRP** 完成时 **I/O** 管理器将设置这个事件，并用 **status** 和 **information** 的最后值填充你的 **iostatus** 结构。最后 **I/O** 管理器将调用 **IoFreeIrp** 释放该 **IRP**。因此，在调用 **IoCallDriver** 之后你不应该访问这个 **IRP** 指针。

因为内部控制操作需要在两个驱动程序之间协作，所以关于发送它们的规则较少。你不必使用 **IoBuildDeviceIoControlRequest** 创建这种请求，例如，你可以仅仅调用 **IoAllocateIrp** 并执行自己的初始化。如果目标驱动程序不希望在 **PASSIVE_LEVEL** 级上处理内部控制操作，你还可以在 **DISPATCH_LEVEL** 级上发送这样的 **IRP**，即从 **I/O** 完成例程或从 **DPC** 例程中发出。(当然，在这种情况下你不能使用 **IoBuildDeviceIoControlRequest**，并且你也不能等待该 **IRP** 的完成。但你可以发送它，因为 **IoAllocateIrp** 和 **IoCallDriver** 可以运行在 **DISPATCH_LEVEL** 级上) 你甚至不必象常规 **IOCTL** 那样准确地使用 **I/O** 堆栈的各 **parameter** 域。实际上，在调用 **USB** 总线驱动程序中用于保存 **URB** 指针的域就是输出缓冲区长度域。所以，如果你正为你自己的两个驱动程序设计一个内部控制协议，可以把 **IRP_MJ_INTERNAL_DEVICE_CONTROL** 看做是你要发的各种消息的信封。

内部和外部控制操作使用同一个派遣例程不是一个好的想法，这里有一个例子解释了其原因。假定你的驱动程序有一个外部控制接口，它允许应用程序查询驱动程序的版本号，还有一个内部控制接口，它允许被信任的内核模式调用者获得某些关键信息，这些信息不希望被用户模式应用程序共享。然后假设你使用一个例程同时处理这两个接口，下面是该例子的代码：

```

NTSTATUS DriverEntry(...)
{
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DispatchControl;
    DriverObject->MajorFunction[IRP_MJ_INTERNAL_DEVICE_CONTROL] = DispatchControl;
    ...
}

NTSTATUS DispatchControl(...)
{
    ...
    switch (code)
    {

```

```
case IOCTL_GET_VERSION:  
...  
case IOCTL_INTERNAL_GET_SECRET:  
    ... // exposed for user-mode calls  
}  
}
```

如果有个应用程序知道了 `IOCTL_INTERNAL_GET_SECRET` 的数值，那么它就能发出一个常规的 `DeviceIoControl` 调用并越过该函数安全检查。

应用程序关注事件的通知

`IOCTL` 操作的一个极其重要的用途就是为 WDM 驱动程序提供了一种方法来通知应用程序其关注事件的发生。为了引发这个讨论，假定你有一个需要与驱动程序紧密合作的应用程序，当某种硬件事件发生时，驱动程序就通知应用程序以便其作出某种用户可见的动作。例如，当按下医疗设备上的某个按钮时将触发应用程序收集和显示数据。对于这种情况，Windows 为驱动程序通知应用程序提供了两种方法(异步过程调用或发送 `window` 消息)，但这些方法不能在 Windows 2000 中使用(缺乏必要的底层支持)，一种可行的方法是使应用程序发出 `IOCTL` 操作，而驱动程序在被关注事件发生时完成该操作。实现这个方法需要格外小心，我将详细解释这个方法。

当应用程序想从驱动程序接收事件通知时，应该调用 `DeviceIoControl`:

```
HANDLE hDevice = CreateFile("\\\\.\\<driver-name>", ...);  
BOOL okay = DeviceIoControl(hDevice, IOCTL_WAIT_NOTIFY, ...);
```

(`IOCTL_WAIT_NOTIFY` 是我用在 `NOTIFY` 例子中的控制代码)

驱动程序将挂起这个 `IOCTL` 并在以后完成。如果没有“其它考虑”，实际代码就象下面这样简单:

```
NTSTATUS DispatchControl(...)  
{  
    ...  
    switch (code)  
    {  
        case IOCTL_WAIT_NOTIFY:  
            pdx->NotifyIrp = Irp;  
            IoMarkIrpPending(Irp);  
            return STATUS_PENDING;  
        ...  
    }  
}  
  
VOID OnInterestingEvent(...)  
{  
    ...  
    CompleteRequest(pdx->NotifyIrp, STATUS_SUCCESS, 0);  
}
```

使用事件通知应用程序

有时驱动程序所要做的全部就是通知应用程序事件已经发生，不必向应用程序传递任何解释性的数据。能做到这一点的一个标准技术是让驱动程序发出一个普通的 Win32 事件信号。为此，应用程序首先要调用 **CreateEvent** 或 **OpenEvent** 打开一个事件对象句柄，然后通过调用 `DeviceIoControl` 把该事件句柄传递给驱动程序。驱动程序通过下面函数可以把该用户模式句柄转换成一个对象指针:

```
PKEVENT pEvent;
status = ObReferenceObjectByHandle(hEvent,
                                    EVENT_MODIFY_STATE,
                                    *ExEventObjectType,
                                    Irp->RequestorMode,
                                    (PVOID*) &pEvent,
                                    NULL);
```

注意该 IOCTL 必须在 PASSIVE_LEVEL 级上处理，并且在拥有 **hEvent** 句柄的进程上下文中。

在此，驱动程序有了一个 KEVENT 对象的指针，它可以在适当时候作为调用 **KeSetEvent** 的参数。该驱动程序还拥有对该事件对象的参考计数，因此它必须在某一点上调用 **ObDereferenceObject**。取消对象参考的恰当时间取决于应用程序与驱动程序的合作方式。比较好的办法是作为 IRP_MJ_CLOSE 处理的一部分，光盘上的 EVWAIT 例子演示了这种使用方法。

内核服务例程 **IoCreateNotificationEvent** 和 **IoCreateSynchronizationEvent** 创建的事件对象可以被用户模式应用程序所共享。但它们在 Windows 98 中无效，因此对真正的 WDM 驱动程序也无效。

我曾提到的“其它考虑”对于打造一个正常工作的驱动程序十分重要。IRP 的创造者可以取消 IRP。应用程序调用 **CancelIo** 或应用程序线程的终止也可以导致某个内核模式部件调用 **IoCancelIrp**。在任何一种情况下，我们都必须提供一个取消例程以便该 IRP 能够完成。如果设备失去电源，或者设备突然被从计算机上摘除，我们需要终止任何等待的 IOCTL 请求。通常，需要终止的 IOCTL 数量是不可以预测的。因此我们需要一个链表，由于有多个线程会访问这个链表，所以我们还需要一个自旋锁以使访问安全。

使用异步IOCTL

为了方便，我写了一组管理异步 IOCTL 的辅助函数。其中两个最重要的是 **CacheControlRequest** 和 **UncacheControlRequest**。它们假定每个设备对象一次仅能接受一个带有特殊控制代码的异步 IOCTL。因此，你可以在设备扩展中保留一个指向当前等待 IRP 的指针域。在 NOTIFY 例子中，我称这个指针域为 **NotifyIrp**。你应该以下面方式接收异步 IRP：

```
IoAcquireRemoveLock(...);
switch (code)
{
case IOCTL_WAIT_NOTIFY:
    if (<parameters invalid in some way>)
        status = STATUS_INVALID_PARAMETER;
    else
        status = CacheControlRequest(pdx, Irp, &pdx->NotifyIrp);
    break;
}

IoReleaseRemoveLock(...);
return status == STATUS_PENDING ? status : CompleteRequest(Irp, status, info);
```

这里重要的语句是 **CacheControlRequest** 调用，如果必要，它寄存该 IRP 以便我们以后能够取消它。它还在设备扩展的 **NotifyIrp** 成员中记录下该 IRP 的地址。我们希望它返回 **STATUS_PENDING**，这样我们可以避免完成该 IRP 并简单地向我们的调用者返回 **STATUS_PENDING**。

注意

你可以容易地归纳出这个方案，允许应用程序为每个打开的句柄分配一个等待类型的 IRP。除了把当前 IRP 指针放到设备扩展中，你还可以把它放到与 FILE_OBJECT 关联的结构中，而 FILE_OBJECT 与句柄对应。你可以在 IRP_MJ_CREATE、IRP_MJ_CLOSE(实际上是所有为该文件句柄生成的 IRP 中)的 I/O 堆栈单元中得到该 FILE_OBJECT 的指针。你还可以任

意的使用该文件对象中的 **FsContext** 或 **FsContext2** 域。

之后，如果有应用程序期待的事件发生，则执行下面代码：

```
PIRP nfyirp = UncacheControlRequest(pdx, &pdx->NotifyIrp);
if (nfyirp)
{
    <do something>
    CompleteRequest(nfyirp, STATUS_SUCCESS, <info value>);
}
```

这些代码提取未决 IOCTL_WAIT_NOTIFY 请求的地址，向应用程序返回数据，然后完成这个未决的 IRP。

辅助例程的工作原理

我在 CacheControlRequest 和 UncacheControlRequest 函数中隐藏了大量的复杂代码。这两个函数实现了线程安全和多处理器安全机制，可以跟踪异步 IOCTL 请求。它们使用了与排队和出队 IRP 时为防止其它人取消该 IRP 所使用的类似的技术。下面是一些特别的代码(参见 NOTIFY 例子中的 CONTROL.CPP)：

```
typedef struct _DEVICE_EXTENSION {
    KSPIN_LOCK IoctlListLock;
    LIST_ENTRY PendingIoctlList;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

NTSTATUS CacheControlRequest(PDEVICE_EXTENSION pdx, PIRP Irp, PIRP* pIrp)
{
    KIRQL oldirql;
    KeAcquireSpinLock(&pdx->IoctlListLock, &oldirql);
    NTSTATUS status;
    if (*pIrp)
        status = STATUS_UNSUCCESSFUL;
    else if (pdx->IoctlAbortStatus)
        status = pdx->IoctlAbortStatus;
    else
    {
        IoSetCancelRoutine(Irp, OnCancelPendingIoctl);
        if (Irp->Cancel && IoSetCancelRoutine(Irp, NULL))
            status = STATUS_CANCELLED;
        else
        {
            IoMarkIrpPending(Irp);
            status = STATUS_PENDING;
            PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
            stack->Parameters.Others.Argument1 = (PVOID) *pIrp;
            IoSetCompletionRoutine(Irp,
                (PIO_COMPLETION_ROUTINE) OnCompletePendingIoctl,
                (PVOID) pdx,
                TRUE,
                TRUE,
                TRUE);
            PFILE_OBJECT fop = stack->FileObject;
            IoSetNextIrpStackLocation(Irp);
            stack = IoGetCurrentIrpStackLocation(Irp);
            stack->DeviceObject = pdx->DeviceObject;
        }
    }
}
```

```

stack->FileObject = fop;

    *pIrp = Irp;
    InsertTailList(&pdx->PendingIoctlList, &Irp->Tail.Overlay.ListEntry);
}
}

KeReleaseSpinLock(&pdx->IoctlListLock, oldirql);
return status;
}

VOID OnCancelPendingIoctl(PDEVICE_OBJECT fdo, PIRP Irp)
{
    KIRQL oldirql = Irp->CancelIrql;
    IoReleaseCancelSpinLock(DISPATCH_LEVEL);
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    KeAcquireSpinLockAtDpcLevel(&pdx->IoctlListLock);
    RemoveEntryList(&Irp->Tail.Overlay.ListEntry);
    KeReleaseSpinLock(&pdx->IoctlListLock, oldirql);
    Irp->IoStatus.Status = STATUS_CANCELLED;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
}

NTSTATUS OnCompletePendingIoctl(PDEVICE_OBJECT junk, PIRP Irp, PDEVICE_EXTENSION pdx)
{
    KIRQL oldirql;
    KeAcquireSpinLock(&pdx->IoctlListLock, &oldirql);
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    PIRP* pIrp = (PIRP*) stack->Parameters.Others.Argument1;
    if (*pIrp == Irp)
        *pIrp = NULL;
    KeReleaseSpinLock(&pdx->IoctlListLock, oldirql);
    return STATUS_SUCCESS;
}

PIRP UncacheControlRequest(PDEVICE_EXTENSION pdx, PIRP* pIrp)
{
    KIRQL oldirql;
    KeAcquireSpinLock(&pdx->IoctlListLock, &oldirql);
    PIRP Irp = (PIRP) InterlockedExchangePointer(pIrp, NULL);
    if (Irp)
    {
        if (IoSetCancelRoutine(Irp, NULL))
        {
            RemoveEntryList(&Irp->Tail.Overlay.ListEntry);
        }
        else
            Irp = NULL;
    }
    KeReleaseSpinLock(&pdx->IoctlListLock, oldirql);
    return Irp;
}

```

1. 我们使用一个自旋锁来保护未决 IOCTL 链表，同时也用于保护所有指向不同种类异步 IOCTL 请求的当前实例的保留指针单元。
2. 这里就是那个强制规则，其实这更是一个设计决策，即一次仅能有每种类型 IRP 的一个实例可以等待。
3. if 语句表明这个事实，由于电源或 PnP 事件，我们可能在某个点上需要启动失败的 IRP。

4. 由于我们可能长时间挂起这个 IRP，所以我们需要为它提供一个取消例程。
5. 这里，我们决定前进并隐藏这个 IRP 以便我们能在以后完成它。因为我们最后要从 **DispatchControl** 函数返回 **STATUS_PENDING**，所以需要调用 **IoMarkIrpPending**。
6. 当取消 IRP 时，我们需要用某种方法置空隐藏指针单元。向一个取消例程输送上下文参数十分困难，所以我决定建立一个 I/O 完成例程。我用堆栈单元中的 **Parameters.Others.Argument1** 来记录这个隐藏指针单元。
7. 为了我们刚刚安装并将要调用的完成例程，我们必须调用 **IoSetNextIrpStackLocation** 函数前进 I/O 堆栈指针。在这个特定驱动程序中，我们知道这里必定有一个或更多的堆栈单元供我们使用，因为我们的 **AddDevice** 函数在发现我们下面没有一个驱动程序对象时会失败。后面例程需要的设备和文件对象指针来自下一个当前堆栈单元，所以我们还需要初始化它们。
8. 这个语句安装完成例程。如果 IRP 被取消了，我们最后会置空隐藏指针。
9. 在事件处理的正常过程中，这个语句揭示一个 IRP。
10. 现在我们已经揭示我们的 IRP，我们不希望它被取消。如果 **IoSetCancelRoutine** 返回 NULL，我们知道该 IRP 当前正在被取消。在这种情况下我们返回一个空 IRP 指针。

NOTIFY 还有一个用于处理未决 IOCTL 的 **IRP_MJ_CLEANUP** 例程，该例程看起来与处理读写操作的清除例程相同。最后它包含一个 **AbortPendingIoctl**s 辅助函数用于处理设备掉电或设备突然消失的特殊情况，如下：

```
VOID AbortPendingIoctl(PDEVICE_EXTENSION pdx, NTSTATUS status)
{
    InterlockedExchange(&pdx->IoctlAbortStatus, status);
    CleanupControlRequests(pdx, status, NULL);
}
```

CleanupControlRequests 是 **IRP_MJ_CLEANUP** 的处理例程。如果该函数的第三个参数(通常是一个文件对象指针)为空则取消所有未决的 IRP。

对比真正的驱动程序，**NOTIFY** 有点太简单。下面是一些额外的考虑，你可以考虑用于你自己的设计中。

- 一个驱动程序可以有多种能触发通知的事件类型。你可以仅用一个 IOCTL 代码来对应这些事件类型，但这样需要你用输出数据来指出事件类型。另一种选择是使用多个 IOCTL 代码。
- 你可以允许多线程寄存事件。但这样你就不能在设备扩展中仅保存一个 IRP 指针，你需要一种方法来跟踪某种事件类型的所有 IRP。如果对于所有通知，你仅有一个 IOCTL 类型，那么跟踪 IRP 的一种方法是依靠 **PendingIoctlList**。然后，当一个事件发生时，执行一个循环，循环调用 **ExInterlockedRemoveHeadList** 和 **IoCompleteRequest** 来清空未决链表。(由于我强制一次只运行测试程序的一个实例，所以在 NOTIFY 中我避免了这个复杂性)
- 你的 IOCTL 派遣例程可能与生成事件的活动相竞争。例如，在第十一章的 **USBINT** 例子中，IOCTL 派遣例程和服务 USB 设备上某中断端点的假中断例程之间存在着一个潜在的竞争。为了避免丢失事件或产生不协调动作，你需要一个自旋锁。如何恰当使用这个自旋锁请参见光盘上的 **USBINT** 例子。(在 NOTIFY 中同步不是一个问题，因为当操作者执行释放事件信号的按键动作时，通知请求几乎总是挂起的。如果不是，通知请求失败)

关于 NOTIFY 例子的更多内容

NOTIFY 由一个 WDM 设备驱动程序(位于 **SYS** 子目录)和一个控制台模式测试程序(在 **TEST** 子目录)组成。你可以使用添加新硬件向导或 **FASTINST** 工具安装这个驱动程序。然后运行测试程序。该程序将产生一个单独的线程并发出 **IOCTL_WAIT_NOTIFICATION** 请求。然后该测试程序提示你按任意键或按 **Ctrl+Break** 退出测试程序。如果你按了一个键，则测试程序发出一个 **IOCTL_GENERATE_EVENT**，输入数据就是这次按键的扫描码。然后驱动程序存储这个扫描码(作为输出数据)，之后完成这个未决的通知 IRP。如果你按下了 **Ctrl+Break**，I/O 管理器将取消正等待的通知 IRP。

系统线程

到现在为止，本书的内容还未十分深入地涉及驱动程序例程所处于的线程上下文。我们的子例程大部分时间是运行在任意线程上下文中，这意味着我们不能阻塞该线程，也不能直接访问用户模式虚拟内存。这些限制使一些设备特别难于编程。

有些设备的最佳处理方式是循检。例如，不能异步中断 CPU 的设备就需要不断地检测其状态。另一些设备由于自身的特点，需要把一个操作分成多个步骤来执行，并且这些步骤间需要插入等待。例如，一个软盘驱动程序需要一系列步骤才能完成一个操作。通常，该驱动程序必须先命令软盘驱动器马达转到一定速度，等待马达转速稳定、然后开始传输、接着再等待一会、最后使驱动器马达停转。你可以设计这样一个驱动程序，其操作象一个有限状态机，使用回调函数来顺序化操作步骤。但在一个直线式程序的适当地方插入事件和计数器等待也可以使处理更容易一些。

用属于驱动程序的系统线程来处理需要定期查询的设备会更容易一些。系统线程是一个完全操作在整个操作系统进程的保护之下的线程。我仅讨论执行在内核模式中的系统线程。在下一段，我将描述创建和销毁系统线程。然后我将给出一个例子，该例子演示了如何用系统线程管理使用循检方式的输入设备。

系统线程的创建与终止

调用 **PsCreateSystemThread** 可以创建一个系统线程。该函数的一个参数是“线程过程(thread procedure)”的地址，该线程过程是新线程的主程序。当线程过程要终止其线程时，它调用 **PsTerminateSystemThread** 函数，该函数不返回。一般，你需要为 PnP 事件提供一种方式来通知线程终止并等待终止发生。综合所有这些因素，最后你将得到这三个子例程：

```
typedef struct _DEVICE_EXTENSION {
    ...
    KEVENT evKill;
    PKTHREAD thread;
};

NTSTATUS StartThread(PDEVICE_EXTENSION pdx)
{
    NTSTATUS status;
    HANDLE hthread;
    KeInitializeEvent(&pdx->evKill, NotificationEvent, FALSE); <
    status = PsCreateSystemThread(&hthread,
        THREAD_ALL_ACCESS,
        NULL,
        NULL,
        NULL,
        (PKSTART_ROUTINE) ThreadProc,
        pdx);
    if (!NT_SUCCESS(status))
        return status;
    ObReferenceObjectByHandle(hthread,
        THREAD_ALL_ACCESS,
        NULL,
        KernelMode,
        (PVOID*) &pdx->thread,
        NULL);
    ZwClose(hthread);
    return STATUS_SUCCESS;
```

```

}

VOID StopThread(PDEVICE_EXTENSION pdx)
{
    KeSetEvent(&pdx->evKill, 0, FALSE);
    KeWaitForSingleObject(pdx->thread, Executive, KernelMode, FALSE, NULL); <
    ObDereferenceObject(pdx->thread);
}

VOID ThreadProc(PDEVICE_EXTENSION pdx)
{
    ...
    KeWaitForXxx(<at least pdx->evKill>);
    ...
    PsTerminateSystemThread(STATUS_SUCCESS);
}

```

1. 在设备扩展中声明一个名为 **evKill** 的 KEVENT 对象，这样，PnP 事件就能通知线程终止。现在是初始化该事件对象的恰当时间。
2. 该语句产生新线程。调用成功的结果是返回一个线程句柄，该句柄返回到函数第一个参数指定的变量中。第二个参数指出你为该线程请求的访问权限；在这里 THREAD_ALL_ACCESS 是一个合适的值。下三个参数属于该线程的用户模式进程部分，在 WDM 驱动程序中调用该函数时这些参数应该为 NULL。接下来的参数(**ThreadProc**)为该线程指出主程序。最后一个参数(**pdx**)是一个上下文参数，它是线程过程仅有的一个参数。
3. 为了等待线程终止，你需要 KTHREAD 对象的地址来代替从 **PsCreateSystemThread** 获得的线程句柄。调用 **ObReferenceObjectByHandle** 获得这个地址。
4. 实际上，一旦我们得到 KTHREAD 的地址我们就不再需要线程句柄了，所以我们调用 **ZwClose** 关闭这个句柄。
5. **StopDevice** 例程在我们的驱动程序模块化设计方案中执行 **IRP_MN_STOP_DEVICE** 的设备相关部分，它调用 **StopThread** 停止该系统线程。第一步是设置 **evKill** 事件。
6. 该调用显示了如何等待线程结束。一个内核线程对象也是一个可以等待的同步对象。当线程最后结束时，它被假定为信号态。在 Windows 2000 中，你应该总执行这个等待以避免当你的某个系统线程执行在其关闭处理的最后几条指令时你的驱动程序映像被解除映射。即不要仅仅等待一个特殊的“kill acknowledgment”事件，而该事件就在线程退出前被设置。在驱动程序安全卸载前线程必须执行 **PsTerminateSystemThread**。参见本章后的 Windows 98 兼容问题(等待系统线程结束)。
7. 该 **ObDereferenceObject** 调用与创建该线程时的 **ObReferenceObjectByHandle** 调用相对应。有必要使对象管理器释放由 KTHREAD 对象占用的内存。
8. 线程过程中的代码逻辑取决于你要完成的实际目标。如果你为等待某个外部事件而阻塞，你应该调用 **KeWaitForMultipleObjects** 并指定 **evKill** 事件为其中一个对象。
9. 当你检测到 **evKill** 已经进入信号态时，调用 **PsTerminateSystemThread** 函数，该函数终止这个线程，但不返回。注意除非你在该线程本身的上下文中调用该函数，否则不能终止这个系统线程。

用系统线程循检设备

如果你不得不为一个不能中断 CPU 的设备写驱动程序，一个系统线程可以专门用于循检设备。我将给你阐述一个用于这种目的的系统线程使用方法。这个例子基于一个有两个输入端口的假想设备。一个端口作为控制端口；当无输入数据时该端口可读出字节值 0，当有输入数据时该端口可读出字节值 1。从另一个端口读出一个单字节数据并重置控制端口。

在这个例子中，我们在处理 **IRP_MN_START_DEVICE** 请求时创建系统线程。当收到一个 PnP 请求 **IRP_MN_STOP_DEVICE** 或 **IRP_MN_REMOVE_DEVICE** 时终止该线程。该线程大部分时间处于阻塞状态。当 **StartIo** 例程开始处理 **IRP_MJ_READ** 请求时，它设置一个事件，循检线程就在这个事件上等待。循检线程然后进入服务该请求的循环。在这个循环中，循检线程首先阻塞一个固定循检间隔时间。间隔时间一到，该线程就读控制端口。如果读出的值是 1，该线程就读一字节数据。然后线程重复这个循环直到请求被满足，最后它返回睡眠状态直到 **StartIo** 收到另一个请求。

POLLING 例子的线程例程如下：


```

        NULL,
        NULL);
if (!NT_SUCCESS(status))
{
    kill = TRUE;
    break;
{
if (status == STATUS_WAIT_0)
{
    status = STATUS_DELETE_PENDING;
    kill = TRUE;
    break;
}
if (pdx->nbytes)
{
    if (READ_PORT_UCHAR(pdx->portbase) == 1)
    {
        *pdx->buffer++ = READ_PORT_UCHAR(pdx->portbase + 1);
        --pdx->nbytes;
        ++numxfer;
    }
}
if (!pdx->nbytes)
    break;
} // read next byte
KeCancelTimer(&timer);
StartNextPacket(&pdx->dqReadWrite, pdx->DeviceObject);
if (Irp)
{
    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    CompleteRequest(Irp, STATUS_SUCCESS, numxfer);
}
} // until told to quit

PsTerminateSystemThread(STATUS_SUCCESS);
}

```

1. 我们以后将使用这个内核定时器来控制循检设备的频率。
2. 在该函数中我们调用 **KeWaitForMultipleObjects** 两次以阻塞循检线程直到某些事情发生。这两个数组提供我们要等待的同步对象的地址。**ASSERT** 语句检查是否有足够的等待事件。
3. 当有错误发生时或当 **evKill** 变成信号态时这个循环终止。然后我们终止整个循检线程。
4. 当 **evKill** 或 **evRequest** 变成信号态时这个等待终止。我们的 **StartIo** 例程将置 **evRequest** 为信号态，以指出有一个 IRP 需要服务。
5. **KeSetTimerEx** 将启动我们的计数器。这是一个反复计数器，基于启动期和周期。我们指定了 0 启动期，这将使我们立即循检设备。**POLLING_INTERVAL** 的单位是毫秒。
6. 当 **evKill** 事件变成信号态或当我们完成当前 IRP 时这个内圈循环终止。
7. 当我们在这个循环中开始操作时，当前的 IRP 可能被取消，或者我们收到一个要求我们终止该 IRP 的 PnP 或电源管理请求。
8. 在调用 **KeWaitForMultipleObjects** 中，我们利用这个事实：内核定时器就象一个事件对象。当 **evKill** 进入信号态(这意味着我们应该一起终止循检线程)或定时器到时间(这意味着我们应该执行下一次循检)时该调用结束。
9. 这是该驱动程序中的真正循检步骤。我们读控制端口，其地址是由 PnP 管理器给出的基本端口地址。如果该值指出数据已准备好，我们就读数据端口。

与这个循检例程协作的 **StartIo** 例程首先设置设备扩展中的 **buffer** 和 **nbytes** 域；循检例程使用这些域来顺序化输入请求。然后它设置 **evRequest** 事件唤醒循检线程。

除了我阐述的这种方式，你也可以用其它方式组织一个循检驱动程序。例如，一旦新到来的请求中指出设备空闲，就创建新循检线程。该线程服务于这个请求直到设备变为空闲，然后线程终止。这个策略更适合于活动间隔时间长的设备，因为在长时间的间隔期中，使用这种方式可以使循检线程不占用虚拟内存。然而，如果设备常常处于连续性的忙状态，则第一个策略更合适，它可以避免重复启动和停止循检线程的消耗。

练习 POLLING 例子

你仅能在 Windows 98 上测试 POLLING 例子驱动程序。按照光盘上的指导运行 DEVTEST 仿真器，仿真 POLLING 管理的假硬件。然后运行用户模式的 TEST 程序执行读操作。

工作项

有时你希望通过临时降低处理器的中断请求级(**IRQL**)来执行某些任务或其它必须在**PASSIVE_LEVEL**级下执行的任务。但是降低 **IRQL** 显然是不行的。不过如果你运行在低于或等于 **DISPATCH_LEVEL** 级上，你可以排队一个工作项(**work item**)，之后这个工作项会请求回调驱动程序中的例程。回调将发生在 **PASSIVE_LEVEL** 级，在由操作系统所拥有的一一个 **worker** 线程的上下文中运行。使用工作项(**work item**)可以避免自己创建仅偶尔醒来的线程的麻烦。

我将描述一种使用工作项的简单方法。首先声明一个结构，该结构开始于一个无名的 **WORK_QUEUE_ITEM** 结构实例。下面代码来自光盘上的 **WORKITEM** 例子：

```
struct _RANDOM_JUNK {
    struct _WORK_QUEUE_ITEM;
    <other stuff>
} RANDOM_JUNK, *PRANDOM_JUNK;
```

声明工作项(**Work-Item**)结构

Microsoft 的 C/C++ 扩展允许一个大结构包含一个未命名的联合或结构成员。在本文的例子中，你可以直接引用标准 **WORK_QUEUE_ITEM** 的成员而不必提供中间名称限定。

如果你能使用 **C++** 语法，则可以用更好的方式声明这样的结构：

```
struct _RANDOM_JUNK : public _WORK_QUEUE_ITEM {
    <other stuff>
};
typedef _RANDOM_JUNK RANDOM_JUNK, *PRANDOM_JUNK;
```

这个语法指出 **_RANDOM_JUNK** 派生自 **_WORK_QUEUE_ITEM**，即它继承了基类的所有成员。你可能熟悉 **C++** 的类派生概念，但结构也可以派生。使用这种声明方法，你仍旧能不用额外名称限定而直接引用 **WORK_QUEUE_ITEM** 域，应该避免依靠 Microsoft 的扩展语法。

准备好后，从堆上分配一个该结构的实例并初始化：

```
PRANDOM_JUNK junk = (PRANDOM_JUNK) ExAllocatePool(NonPagedPool, sizeof(RANDOM_JUNK));
PIO_WORKITEM item = IoAllocateWorkItem(fdo);
junk->item = item;
<additional initialization>
```

初始化完上下文结构(即 **junk**)后，排队这个工作项：

```
IoQueueWorkItem(item,
    (PIO_WORKITEM_ROUTINE) WorkItemCallback,
    DelayedWorkQueue,
    junk);
```

排队工作项之后，操作系统将在某个系统 **worker** 线程的上下文中回调你的例程。你将执行在 **PASSIVE_LEVEL** 级上。回调例程所做的工作大部分由你决定，但有一个要求：你必须释放由上下文结构和工作队列项占用的内存。下面是一个工作项回调例程的框架代码：

```
VOID WorkItemCallback(PDEVICE_OBJECT fdo, PRANDOM_JUNK junk)
```

```
{  
...  
    IoFreeWorkItem(junk->item);  
    ExFreePool(junk);  
}
```

这段代码调用了 **ExFreePool**, 与以前的内存分配相对应。

关于工作项有几个重要的地方需要指出。不能从系统队列中删除工作项。然而如果你为满足一个 PnP 请求而删除了你的设备，那么出现驱动程序被卸载而工作项仍然未决的情况也是有可能的。你可以使用删除锁机制来防止这种事情发生，如下：

- 在排队一个工作项之前，调用 **IoAcquireRemoveLock** 申请一个删除锁，防止驱动程序被提前卸载。
- 在工作项回调例程的结尾，调用 **IoReleaseRemoveLock** 释放删除锁。为此，你需要在回调函数中访问你的设备扩展。因此，你应该事先把设备扩展指针或设备对象指针放到 RANDOM_JUNK 结构中。

另外，你的回调函数还需要做些工作，以防止在设备意外删除或掉电的情况下其它例程访问硬件，等等。

IoXxxWorkItem

IoXxxWorkItem 例程包括 **IoAllocateWorkItem**、**IoQueueWorkItem**，和 **IoFreeWorkItem**，它们在 WDM.H 中声明，在 NTOSKRNL.LIB(不是 WDM.LIB)中定义。Microsoft 推荐你用它们替代以前的 **ExXxxWorkItem** 函数(executive 部件输出的函数)，新函数在原来的函数外围加上了对用户指定设备对象的引用代码。这个引用可以防止设备对象意外消失。Windows 98 没有实现这些新函数，所以使用工作项的驱动程序或者使用 WDMSTUB.VXD，或者提供两个版本：Windows 2000 版本使用 **IoXxxWorkItem** 例程，Windows 98 版本使用 **ExXxxWorkItem**。

关于 WORKITEM 例子

光盘上的 WORKITEM 例子驱动程序演示了工作项的使用。其工作过程是这样的：测试程序先发出一个带有 IOCTL_SUBMIT_ITEM 代码的 DeviceIoControl 请求。然后驱动程序把它当做一个异步 IOCTL 来执行。在从 DEVICE_CONTROL 派遣函数返回前驱动程序还排队一个工作项。当工作项回调发生时，驱动程序完成 IOCTL_SUBMIT_ITEM 请求。

看门狗定时器

有些设备在发生错误时并不通知你，它们仅简单地不响应。每个设备对象都有一个内置的 **IO_TIMER** 对象，你可以使用这个对象来避免不明确的操作等待。当这种定时器运行时，**I/O** 管理器将每秒调用一次定时器回调例程。在回调例程中，你可以分步终止任何未决的操作，这些操作早应该完成但未完成。

应该在 **AddDevice** 函数中初始化定时器对象：

```
NTSTATUS AddDevice(...)  
{  
    ...  
    IoInitializeTimer(fdo, (PIO_TIMER_ROUTINE) OnTimer, pdx);  
    ...  
}
```

fdo 是设备对象的地址，**OnTimer** 是定时器回调例程，**pdx** 是 **I/O** 管理器调用 **OnTimer** 时使用的上下文参数。

通过调用 **IoStartTimer** 函数启动定时器计时，调用 **IoStopTimer** 函数停止定时器计时。在这之间，你的 **OnTimer** 例程每隔一秒被调用一次。

光盘上的 **PIOFAKE** 例子演示了一个使用 **IO_TIMER** 做看门狗的方法。我把一个 **timer** 成员放到这个假设备的设备扩展中：

```
typedef struct _DEVICE_EXTENSION {  
    ...  
    LONG timer;  
    ...  
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

当处理设备的第一个 **IRP_MJ_CREATE** 时，我启动定时器计时。当设备的最后一个句柄被关闭时，我停止该定时器：

```
NTSTATUS DispatchCreate(...)  
{  
    ...  
    if (InterlockedIncrement(&pdx->handles) == 1)  
    {  
        pdx->timer = -1;  
        IoStartTimer(fdo);  
    }  
    ...  
}  
  
NTSTATUS DispatchClose(...)  
{  
    ...  
    if (InterlockedDecrement(&pdx->handles) == 0)  
        IoStopTimer(fdo);  
    ...  
}
```

timer 单元的初始值为 **-1**。我在 **StartIo** 例程和每次中断后都设置这个值为 **10**。即，我允许设备最长在 **10** 秒内消化掉输出的字节并为准备好接收下一字节而生成中断。(本节后面的文本框解释了这个假想设备的工作原理)

`OnTimer` 例程每隔一秒执行一次，其操作需要与 `ISR` 同步。因此我在中断自旋锁的保护下，在 `DIRQL` 级上使用 `KeSynchronizeExecution` 来调用辅助例程 `CheckTimer`。定时器定时的执行例程与 `ISR` 和 `DPC` 例程的协作过程见下面代码：

```
VOID OnTimer(PDEVICE_OBJECT fdo, PDEVICE_EXTENSION pdx)
{
    KeSynchronizeExecution(pdx->InterruptObject, (PKSYNCHRONIZE_ROUTINE) CheckTimer, pdx);
}

VOID CheckTimer(PDEVICE_EXTENSION pdx)
{
    if (pdx->timer <= 0 || --pdx->timer > 0)                                <
        return;
    PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
    if (!Irp)
        return;
    Irp->IoStatus.Status = STATUS_IO_TIMEOUT;
    Irp->IoStatus.Information = 0;
    IoRequestDpc(pdx->DeviceObject, Irp, NULL);
}

BOOLEAN OnInterrupt(...)
{
    ...
    if (pdx->timer <= 0)
        return TRUE;
    if (!pdx->nbytes)
    {
        Irp->IoStatus.Status = STATUS_SUCCESS;
        Irp->IoStatus.Information = pdx->numxfer;
        pdx->timer = -1;
        IoRequestDpc(pdx->DeviceObject, Irp, NULL);
    }
    ...
    pdx->timer = 10;
}

VOID DpcForIsr(...)
{
    ...
    PIRP Irp = StartNextPacket(&pdx->dqReadWrite, fdo);
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    ...
}
```

1. 一个值为-1 的定时器意味着当前没有请求等待。值为 0 意味着当前请求已经超时。在这两种情况下，我们不希望或不必在这个例程中做任何工作。`if` 表达式的第二部分递减定时器的值。如果还没有减少到 0，我们就返回，不做什么工作。
2. 该驱动程序使用一个 `DEVQUEUE`，所以我们调用 `DEVQUEUE` 例程 `GetCurrentIrp` 来获得当前被处理请求的地址。如果该值为 `NULL`，则设备当前空闲。
3. 在这里，我们希望终止当前请求，因为 10 秒内没有发生中断。我们填充完 `IRP` 的 `status` 域后请求了一个 `DPC`。这个状态代码 `STATUS_IO_TIMEOUT` 转换成 `Win32` 错误代码为 `ERROR_SEM_TIMEOUT`，但其标准错误文本("The semaphore timeout period has expired")并没有真正指出错误所在。如果请求这个操作的应用程序还在你的控制之下，你应该提供一个更有意义的代码解释。

4. 如果定时器等于 0，则当前请求已超时。因为 **CheckTimer** 例程已经请求了 DPC，所以在 ISR 中除了解除中断之外我们不必也不希望做更多的工作。通过设置 **timer** 为 -1，我们防止了再次调用 **CheckTimer**，否则它会为同一个请求而请求另一个 DPC。PIOFAKE 现在使用一个 **busy** 标志(在设备扩展中)来避免为同一个 IRP 请求多个 DPC。
5. 在两次中断之间我们最长允许 10 秒的等待时间。
6. 不管什么活动请求了该 DPC，我们还是要填充 IRP 的状态域。然后，我们仅需要调用 **IoCompleteRequest**。

关于 PIOFAKE

PIOFAKE 例子驱动程序为一个不存在的设备工作，该设备遵从 PIO 模式。设备有一个输出端口，你可以向该端口写 ASCII 字符。在它消化完数据字节后，它会在它的 DIRQ 上生成一个中断。

如果在 Windows 2000 上安装 PIOFAKE 并运行相关的 TEST 程序，10 秒内不会发生任何事情。之后 PIOFAKE 会报告超时，因为它没有看到任何中断，于是测试程序会报告一个超时错误。

在 Windows 98 中，你可以使用 DEVTEST 设备仿真器来练习该例子驱动程序的 PIO 部分。额外信息参见 [PIOFAKE.HTM](#) 中的指导。

Windows 98 兼容问题

在本章所讨论的材料中，有一些地方在 Windows 98 和 Windows 2000 中存在着不同之处。

错误登记

Windows 98 没有实现错误登记文件和事件查看器。当你在 Windows 98 中调用 `IoWriteErrorLogEntry` 时，所发生的全部就是在调试终端上出现几行数据。我认为这个信息的输出格式缺乏美感，所以，在 Windows 98 中我宁愿不使用错误登记工具。如何确定你是运行在 Windows 98 下还是运行在 Windows 2000 下参见附录 A。

IOCTL与Windows 98 虚拟设备驱动程序

Win32 应用程序可以使用 `DeviceIoControl` 与 Windows 98 虚拟设备驱动程序(VxD)通信，就象与一个 WDM 驱动程序一样。但针对 WDM 驱动程序的 IOCTL 与针对 VxD 驱动程序的 IOCTL 之间仍存在着三个小区别。最重要的区别是你从 `CreateFile` 函数获得的设备句柄的含义。使用 WDM 驱动程序时，这个句柄属于某个明确的设备，但与 VxD 驱动程序对话时，你得到的句柄是属于驱动程序的。实际上，VxD 驱动程序需要实现一个假句柄机制(嵌入到 IOCTL 数据流中)，以允许应用程序参考由 VxD 驱动程序管理的特定硬件实例。

另一个不同之处是为控制操作分配控制代码数值。正如我前面讨论的，WDM 驱动程序用 `CTL_CODE` 宏来定义控制代码，因此你定义的代码不能超过 2048 个。对于 VxD，可以是除了 0 和 -1 之外的所有 32 位值。如果你写的应用程序希望既能使用 VxD 驱动程序也能使用 WDM 驱动程序，应该使用 `CTL_CODE` 定义控制代码，VxD 也可以使用 `CTL_CODE` 生成的结果数值。

最后的区别更小，`DeviceIoControl` 的倒数第二个参数，一个指向反馈变量的 `PDWORD`，当调用 WDM 驱动程序时这个参数是必须的，但调用 VxD 驱动程序时该参数是不必要的。换句话说，如果你调用 WDM 驱动程序，你必须提供一个指向 `DWORD` 的非空值。但如果你调用 VxD 驱动程序，如果你对知道有多少字节进入输出缓冲区不感兴趣，可以指定该参数为 `NULL`。不过调用 VxD 驱动程序时提供反馈变量也是无害的。

挂起IOCTL操作时的注意事项

如果应用程序使用 IOCTL 挂起技术来等待驱动程序告诉它硬件事件，那么在运行时它必须有一个打开的句柄。如果你的设备可以从计算机上突然删除，你需要使挂起的 IOCTL 失败，以使应用程序关闭其句柄。在 Windows 2000 中，你可以推迟处理最后的 `IRP_MN_REMOVE_DEVICE` 请求直到所有句柄都被关闭。然而在 Windows 98 中不要这样做，因为可能会发生我在第六章结尾处描述的死锁。如果你看一看我的例子驱动程序，特别是 `NOTIFY`，你会看到这些例子在处理 `IRP_MJ_CREATE` 时没有请求删除锁。这意味着它们允许自身在设备句柄打开时仍可以被卸载。幸运的是，Windows 98 能够处理这个后果，不会带来额外影响。

等待系统线程结束

Windows 98 不支持线程对象指针(PKTHREAD)作为 `KeWaitForSingleObject` 或 `KeWaitForMultipleObjects` 的参数。这些支持函数简单地把它们的对象指针参数传递给 `VWIN32.VXD`，并不做任何有效性检测，`VWIN32` 会因为这些线程对象没有能支持同步使用的结构成员而崩溃。

在 Windows 98 中，如果要等待一个内核模式线程完成，你需要该线程就在其调用 `PsTerminateSystemThread` 函数之前发出一个事件。置这个事件会造成正终止的线程对等待同一个事件的其它线程失去控制。在技术上，这个正终止的线程仍旧是活的，这个结果对 Windows 98 并不算糟糕。在 Windows 2000 中，你可以容易地在终止线程时发现驱动程序被卸载；Windows 2000 中，应该确保在线程对象本身上等待。

第十章：Windows 管理诊断

Windows 2000 支持一种称为 Windows 管理诊断(WMI)的控件，用于管理计算机系统。WBEM(基于 Web 的企业管理)是一个广泛的工业标准，而 WMI 是这个工业标准的 Microsoft 实现。WMI 的目标是为系统管理和企业网络中管理数据的描述提供了一个模型，并尽可能独立于专用 API 或数据对象模型。这种独立性促进了能创建、传输，和显示控制数据的独立系统部件的发展。

WDM 驱动程序以三种方式适应 WMI，见图 10-1。第一，WMI 通常能响应提取性能数据的请求。第二，各种控制器应用程序可以使用 WMI 方式控制设备的通用特征。第三，WMI 提供了一个事件通知机制，允许驱动程序通知应用程序有重要的事件发生。在这一章中，我将描述驱动程序编程的所有这三方面。为了帮助你了解本章例子驱动程序的测试程序，我将描述 WMI 机制的用户模式一侧是如何工作的。

- WMI 概念
- WDM 驱动程序与 WMI
- 用户模式程序与 WMI
- Windows 98 兼容问题

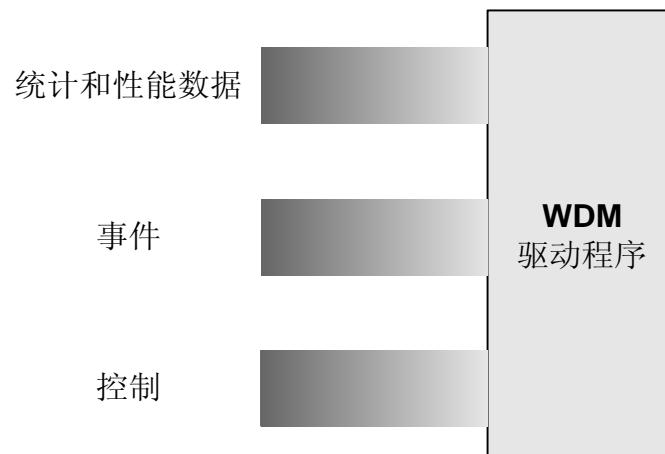


图 10-1. WDM 驱动程序在 WMI 环境中所扮演的角色

关于 WMI 和 WBEM 的名称

公用信息模型(CIM)是由 DMTF(Distributed Management Task Force)支持的基于 Web 的企业管理规范，以前被称为 DMTF。Microsoft 命名其 CIM 实现为“WBEM”，其本质上是“CIM for Windows”。CIM for Windows 的内核模式部分称为“WMI”。为了使 CIM 被更广泛地采纳，DMTF 启动了一个市场行动并使用 WBEM 作为 CIM 的名称。之后 Microsoft 重命名其 WBEM 实现为 WMI，并重命名 WMI(内核模式部分)为“WMI extensions for WDM”。这就是说，WMI 兼容 CIM 和 WBEM 规范。

我担心本章的各种术语会使你混淆。我建议你把这本书或 Microsoft 提供的任何文档中出现的“CIM”或“WBEM”都当做“WMI”。

WMI概念

图 10-2 展示了 WMI 的整体结构，在 WMI 模型中，数据和事件被分成了消费者和生产者两类。数据块就是抽象类的实例，其概念与 C++ 中的类概念一致。如同 C++ 中的类，WMI 类也有数据成员和实现对象行为的方法。数据块中的内容并不是由 WMI 指定，而是由数据生产者和数据的使用目的决定的。送往驱动程序的数据最有可能来自管理者本身的操作。而驱动程序发出的数据通常是某种性能的统计数据，这些数据的消费者可能是某个性能监视程序。

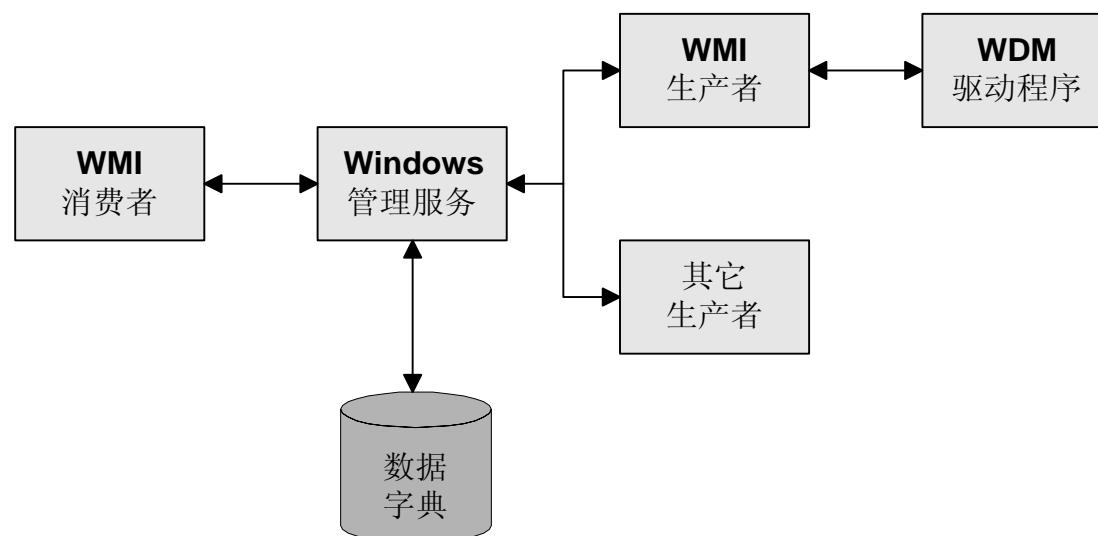


图 10-2. WMI 世界

WMI 允许存在多重命名空间，每个命名空间中包含的类属于一个或多个用户模式生产者。生产者使用平台 SDK 中公开的 COM 接口来寄存 Windows 管理服务(Windows Management Service)。一旦 Windows 2000 发行，操作系统(包括所有设备驱动程序)将支持一个名为 **root\cimv2** 的命名空间，里面包含了 CIM 版本 2。在本书写作时，CIMV2 命名空间的结构还未确定，因此 Microsoft 决定临时为设备驱动程序类使用另一个命名空间 **root\wmi**。

WDM 驱动程序可以作为 WMI 类实例的生产者。一个描述了驱动程序支持的各种类(驱动程序可以为这些类提供数据)的脚本称为驱动程序规划(schema)。我们可以使用 MOF(Managed Object Format)语言定义规划。系统则维护一个称为 **repository** 的数据字典，它包含了所有已知的规划定义。如果驱动程序做得正确，系统将在初始化驱动程序时自动把规划放到 **repository** 中。

一个规划例子

在本章的后面，我将演示一个名为的 WMI42.SYS 例子，该例子有下面 MOF 规划：

```
[Dynamic, Provider("WMIProv"),  
 WMI,  
 Description("Wmi42 Sample Schema"),  
 guid("A0F95FD4-A587-11d2-BB3A-00C04FA330A6"),  
 locale("MS\0x409")]  
  
class Wmi42  
{  
    [key, read]  
    string InstanceName;  
  
    [read]  
    boolean Active;
```

```
[WmiDataId(1), Description("The Answer to the Ultimate Question")]
    uint32 TheAnswer;
};
```

我不打算描述所有 MOF 语法细节；你可以在平台 SDK 或 WMI SDK 文档中找到相关信息。你也可以手工构造 MOF，就象我在上面例子中做的那样，或者使用 WBEM CIM Studio 工具，该工具可以在平台 SDK 或 WMI SDK 中找到。下面是对这个例子的简要解释：

1. 生产者名为 **WMIProv**，是一个系统部件，它知道该如何实例化这个类。例如，它知道如何调用内核模式例程以及发送一个 IRP 到适当的驱动程序。它还能根据 GUID 找到正确的驱动程序。
2. 该规划声明了一个 **WMI42** 类，有三个属性 **InstanceName**、**Active**、**TheAnswer**。

作为开发者，我们用 MOF 编译器编译这个规划并产生一个二进制文件，这个文件最后将成为驱动程序可执行文件的一个资源。在驱动程序初始化过程中，有一部分代码就是告诉 WMI 生产者这个资源在哪里，以便它能读取这个规划并添入到 repository。

编译完规划后，我们还应该运行 WMIMOFCK.EXE 工具，在 DDK 中可以找到该工具，该工具执行检测以便规划能与 WMI 兼容。

MOFF 文件与 Beta 发行版

在 Windows 2000 的 beta 测试期间，WMI 仍处于开发过程中，因此使用每个 Windows 2000beta 版时都要再次运行 MOF 编译器，并手动修改 WMI repository，以便各种 COM 接口都能够访问驱动程序的规划。下面命令行语法把规划放到 WMI 命名空间中：

```
mofcomp -N:root\wmi <name>
```

之后，你就可以使用 WBEMTEST.EXE 工具来测试驱动程序，并且附带的控制台模式测试程序也可以工作。(对于 Windows 2000，MOFTEST.EXE 和 WBEMTEST.EXE 包含在%windir%\system32\wbem 目录中。对于 Windows 98，它们包含在%windir%\ system\wbem 目录中。不过 Windows 98 系统可能需要先安装 WMI 支持。见本章结尾关于“Windows 98 兼容问题”的附加信息)

WDM 驱动程序与 WMI

内核模式驱动程序对 WMI 的支持主要是基于对主代码为 **IRP_MJ_SYSTEM_CONTROL** 的 IRP 的支持。为了能接收到这种 IRP，你必须先寄存这种需求：

```
IoWMIRegistrationControl(fdo, WMI_ACTION_REGISTER);
```

做这个寄存调用的恰当时间是在 **AddDevice** 例程中，寄存完成后，一旦系统认为可以安全地向驱动程序发送系统控制 IRP 时，它就向驱动程序发出一个 **IRP_MJ_SYSTEM_CONTROL** 请求，以获得设备的详细寄存信息。与寄存调用作用相反的调用应该在 **RemoveDevice** 函数中做出：

```
IoWMIRegistrationControl(fdo, WMI_ACTION_DEREGISTER);
```

如果在进行反寄存调用时存在着未决的 WMI 请求，**IoWMIRegistrationControl** 例程将等待直到它们完成。因此有必要使驱动程序在反寄存时仍可以响应 IRP。你可以以 **STATUS_DELETE_PENDING** 失败新的 IRP，但必须响应。

在解释如何服务寄存请求之前，我必须描述处理系统控制 IRP 的一般方法。一个 **IRP_MJ_SYSTEM_CONTROL** 请求可以有表 10-1 列出的副功能码。

表 10-1. **IRP_MJ_SYSTEM_CONTROL** 的副功能码

副功能码	描述
IRP_MN_QUERY_ALL_DATA	获得数据块中每一项的所有实例
IRP_MN_QUERY_SINGLE_INSTANCE	获得单一数据块中的每一项
IRP_MN_CHANGE_SINGLE_INSTANCE	替换单一数据块中的每一项
IRP_MN_CHANGE_SINGLE_ITEM	改变数据块中的一项
IRP_MN_ENABLE_EVENTS	允许事件生成
IRP_MN_DISABLE_EVENTS	禁止事件生成
IRP_MN_ENABLE_COLLECTION	开始收集"expensive"统计信息
IRP_MN_DISABLE_COLLECTION	停止收集"expensive"统计信息
IRP_MN_REGINFO	获得详细寄存信息
IRP_MN_EXECUTE_METHOD	执行一个方法函数

堆栈单元中的 **Parameters** 联合包含了一个 **WMI** 子结构，里面含有系统控制请求的参数：

```
struct {
    ULONG_PTR ProviderId;
    PVOID DataPath;
    ULONG BufferSize;
    PVOID Buffer;
} WMI;
```

ProviderId 指向一个设备对象，请求将被引向这个设备对象。**Buffer** 是一个输入/输出缓冲区的地址，**BufferSize** 是长度，该缓冲区的前几个字节被 **WNODE_HEADER** 结构所映射。派遣例程将从这个缓冲区提取某些信息，也把结果返回到这个缓冲区。在除了 **IRP_MN_REGINFO** 之外的所有副功能码中，**DataPath** 代表一个 128 位 GUID 的地址，该 GUID 标识一个数据块类。对于副功能码 **IRP_MN_REGINFO**，**DataPath** 域可以为

WMIREGISTER 或 **WMIUPDATE**(分别为 0 和 1)，取决于你是否被要求提供初始寄存信息还是仅更新你以前提供的信息。

处理系统控制请求有两种方法，一种方法利用 **WMILIB** 支持“驱动程序”，**WMILIB** 实际上一个内核模式 **DLL**，它输出的服务可以被其它驱动程序调用，这些服务可以直接处理这种 **IRP**。另一种方法就是用户直接处理系统控制 **IRP** 本身。使用 **WMILIB** 可以简化代码，但你不能使用 **WMI** 的全部特征，你被限制在仅由 **WMILIB** 支持的 **WMI** 功能子集上，另外，这样的驱动程序将不能运行在原始版本的 **Windows 98** 上，因为在 **Windows 98** 发行时 **WMILIB** 还没有实现，参见本章结尾的 **Windows 98** 兼容问题。

但 **WMILIB** 仍可以满足大部分驱动程序的需求，所以我仅讨论使用 **WMILIB** 的情况。如果你想自己处理系统控制 **IRP**，参见 **DDK** 文档。

委托**WMILIB**处理**IRP**

在系统控制 **IRP** 的派遣例程中，你可以委托 **WMILIB** 来完成大部分工作，代码如下：

```
WMIGUIDREGINFO guidlist[] = {  
    {&GUID_WMI42_SCHEMA, 1, WMIREG_FLAG_INSTANCE_PDO},  
};  
  
WMILIB_CONTEXT libinfo = {  
    arraysize(guidlist),  
    guidlist,  
    QueryRegInfo,  
    QueryDataBlock,  
    SetDataBlock,  
    SetDataItem,  
    ExecuteMethod,  
    FunctionControl,  
};  
  
NTSTATUS DispatchWmi(IN PDEVICE_OBJECT fdo, IN PIRP Irp)  
{  
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;  
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);  
    if (!NT_SUCCESS(status))  
        return CompleteRequest(Irp, status, 0);  
  
    SYSCONTROL_IRP_DISPOSITION disposition;  
    status = WmiSystemControl(&libinfo, fdo, Irp, &disposition);  
  
    switch (disposition)  
    {  
  
        case IrpProcessed:  
            break;  
  
        case IrpNotCompleted:  
            IoCompleteRequest(Irp, IO_NO_INCREMENT);  
            break;  
  
        default:  
            case IrpNotWmi:  
            case IrpForward:  
                IoSkipCurrentIrpStackLocation(Irp);  
    }  
}
```

```

status = IoCallDriver(pdx->LowerDeviceObject, Irp);
break;
}

IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
return status;
}

```

1. WMILIB_CONTEXT 声明的有效范围是整个文件，它描述了驱动程序支持的类 GUID，并列出几个 WMILIB 用于处理 WMI 请求的回调函数。
2. 与其它派遣例程相同，我们在处理这种 IRP 时也获取和释放删除锁。我们要防止 PnP 事件使下层设备对象消失。我们自己的设备对象不会消失，因为 IoWMIRegistrationControl 调用获得了对它的一次引用。
3. 这个语句调用 WMILIB 来处理该 IRP。我们传递了 WMILIB_CONTEXT 结构的地址。通常我们应使用一个静态上下文结构，因为从一个 IRP 到另一个 IRP，其中的信息不可能被改变。WmiSystemControl 返回两个信息：一个 NTSTATUS 代码和一个 SYSCTL_IRP_DISPOSITION 值。
4. 执行这个 IRP 时我们可能需要做一些额外工作，这取决于它的特征代码，如果这个代码为 IrpProcessed，则该 IRP 已经完成，我们不需要再做任何事情。对于除 IRP_MN_REGINFO 之外的其它副功能码，这种情况就是通常情况。
5. 如果代码是 IrpNotCompleted，则我们有责任完成该 IRP。这也是通常情况，除了 IRP_MN_REGINFO。WMILIB 已经填充完 IRP 中的 IoStatus 块，所以我们仅需要调用 IoCompleteRequest。
6. default 和 IrpNotWmi 情况不应该发生在 Windows 2000 中。如果不能处理所有可能的特征代码，我们将到达 default。如果我们向 WMILIB 发送一个 IRP，但其副功能码在 WMI 中未定义，则我们到达 IrpNotWmi 处。
7. IrpForward 情况发生于该系统控制 IRP 是发往其它驱动程序的。回想一下 ProviderId 参数，它指出处理该 IRP 的驱动程序。WmiSystemControl 用设备对象指针指向的值与第二个参数比较。如果不相同，它就返回到 IrpForward，然后我们把该 IRP 下传到下一个驱动程序。

WMI 消费者通过查看 WMILIB_CONTEXT 结构中的 GUID 来判断驱动程序是否是一个 WMI 生产者。当一个消费者想提取数据时，它通过(间接地)访问 WMI 数据字典(repository)，把一个符号对象名翻译成一个 GUID，这个 GUID 就是以前提到的 MOF 语句的一部分，它应该与 WMILIB_CONTEXT 结构中的 GUID 一致，WMILIB 会关心这个匹配。

WMILIB 将回调驱动程序中的例程来执行设备相关或驱动程序相关的处理。回调函数大部分时间以同步方式执行 IRP 的操作。除了 IRP_MN_REGINFO 之外，我们可以推迟 IRP 处理并返回 STATUS_PENDING。如果一个回调例程挂起了该 IRP，那么它应该额外再调用一次 IoAcquireRemoveLock。任何完成该 IRP 的例程都应调用相反的 IoReleaseRemoveLock 函数。

QueryRegInfo 回调函数

在寄存调用之后第一个到来的系统控制 IRP 带有 IRP_MN_REGINFO 副功能码。我们把该 IRP 传递给 WmiSystemControl，该函数又回头调用 QueryRegInfo 函数(地址在 WMILIB_CONTEXT 中给出)。下面代码摘自 WMI42.SYS：

```

NTSTATUS
QueryRegInfo(PDEVICE_OBJECT fdo,
              PULONG flags,
              PUNICODE_STRING instname,
              PUNICODE_STRING* regpath,
              PUNICODE_STRING resname,
              PDEVICE_OBJECT* pdo)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    *flags = WMIREG_FLAG_INSTANCE_PDO;
    *regpath = &servkey;
    RtInitUnicodeString(resname, L"MofResource");
    *pdo = pdx->Pdo;
    return STATUS_SUCCESS;
}

```

}

我们设置 **regpath** 为一个 **UNICODE_STRING** 结构的地址，该结构包含驱动程序的服务键名（在...\\System\\CurrentControlSet\\Services 下）。我们的 **DriverEntry** 例程会收到这个键名并保存到全局变量 **servkey** 中。我们设置 **resname** 为我们的规划资源名。下面是 WMI42.SYS 的资源文件，从中我们能看到这个名字的来处：

```
#include <windows.h>

LANGUAGE LANG_ENGLISH, SUBLANG_NEUTRAL
MofResource MOFDATA wmi42.bmf
```

wmi42.bmf 就是编译后的 MOF 文件。你可以随意命名这个资源，通常是 **MofResource**。但这个名字的缺点是与 **QueryRegInfo** 调用时指定名字相同。

其余值的设置取决于驱动程序如何处理实例命名。关于实例命名我们稍后再讨论。最简单的选择，也是 Microsoft 强烈推荐的做法(WMI42.SYS 采用)是：使用系统自动生成的名称，即基于总线驱动程序赋予 PDO 的名字。如果使用这种命名方法，我们需要在 **QueryRegInfo** 中做下面工作：

- 在 GUID 列表中设置 **WMIREG_FLAG_INSTANCE_PDO** 标志，该标志是 **WMILIB_CONTEXT** 结构的一部分。这意味着相关 WMI 类的数据块的实例名将使用 PDO 的名字。
- 在我们返回给 WMILIB 的 **flags** 值中设置 **WMIREG_FLAG_INSTANCE_PDO** 标志。它告诉 WMILIB，我们的对象中至少有一个使用了 PDO 命名。
- 在返回给 WMILIB 的值中设置 **pdo**。例子驱动程序的设备扩展中有一个名为 Pdo 的域，我在 **AddDevice** 中设置了这个域以使它在这时有效。

基于 **PDO** 的实例名可以允许查看器程序自动确定驱动程序的友好名和其它属性而不用你在驱动程序中多做任何事。

当你在 **QueryRegInfo** 中返回成功状态后，WMILIB 继续执行并创建一个复杂的结构 **WMIREGINFO**，该结构包含了你的 GUID 列表、你的注册表键、你的资源名，以及与实例名相关的信息。然后返回到你的派遣例程中，最后派遣函数完成该 IRP 并返回。图 10-3 显示了这个过程。

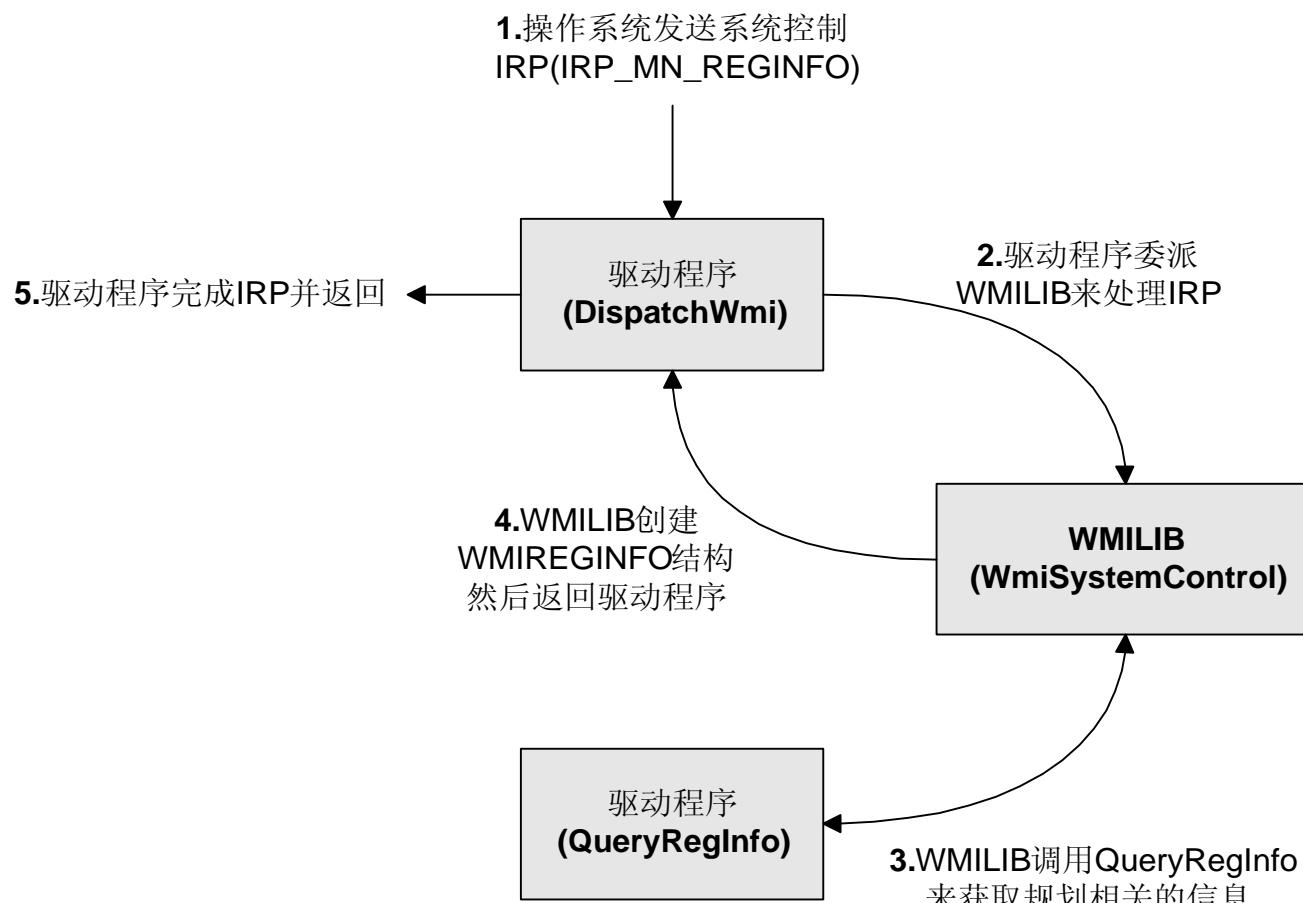


图 10-3. `IRP_MN_REGINFO` 的控制流程

QueryDataBlock 回调函数

你在开始的寄存查询中回答的信息使系统能够把相关数据操作转给你。用户模式代码可以使用各种 COM 接口，以多种详细程度提取或设置数据值。表 10-2 列出了四种详细程度。

表 10-2. 数据查询形式

IRP 副功能码	WMILIB 回调	描述
<code>IRP_MN_QUERY_ALL_DATA</code>	<code>QueryDataBlock</code>	获得所有实例的所有项
<code>IRP_MN_QUERY_SINGLE_INSTANCE</code>	<code>QueryDataBlock</code>	获得一个实例的所有项
<code>IRP_MN_CHANGE_SINGLE_INSTANCE</code>	<code>SetDataBlock</code>	设置一个实例的所有项
<code>IRP_MN_CHANGE_SINGLE_ITEM</code>	<code>SetDataItem</code>	设置一个实例的一个项

当某人想得到你手中的数据时，它们向你发送一个副功能码为 `IRP_MN_QUERY_ALL_DATA` 或 `IRP_MN_QUERY_SINGLE_INSTANCE` 的系统控制 IRP。如果使用 WMILIB，你可以把该 IRP 委托给 WmiSystemControl，该函数然后调用你的回调例程 `QueryDataBlock`。你将在 `QueryDataBlock` 中给出要求的数据，然后调用另一个 WMILIB 例程 `WmiCompleteRequest` 来完成该 IRP，最后返回到 WMILIB。此时，由于你已经完成了该 IRP，WmiSystemControl 将返回 `IrpProcessed` 代码。图 10-4 显示了这个控制流程。

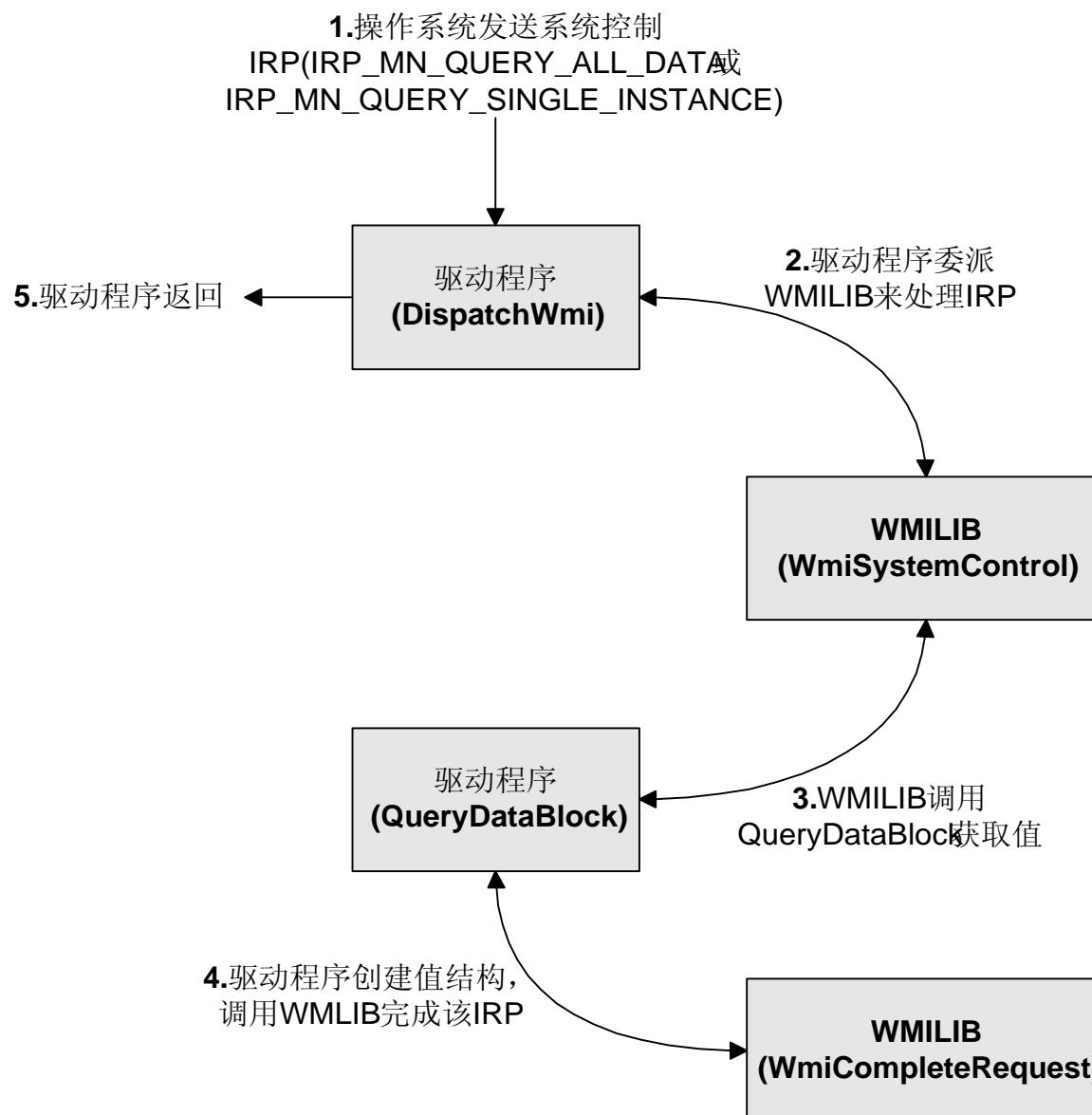


图 10-4. 数据查询的控制流

如果驱动程序要维护数据块的多个实例，并且数据块会随着实例的不同而改变大小，那么你的 `QueryDataBlock` 回调函数可能会成为一个复杂的函数。我以后在“处理多实例”中再讨论这种复杂情况。`WMI42` 例子仅演示了简单情况的处理，即驱动程序仅维护 WMI 类的一个实例：

```
NTSTATUS
QueryDataBlock(PDEVICE_OBJECT fdo,
                PIRP Irp,
                ULONG guidindex,
                ULONG instindex,
                ULONG instcount,
                PULONG instlength,
                ULONG bufsize,
                PUCHAR buffer)
{
    if (!instlength || bufsize < sizeof(ULONG))
        return WmiCompleteRequest(fdo, Irp, STATUS_BUFFER_TOO_SMALL, sizeof(ULONG),
IO_NO_INCREMENT);

    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;

    PULONG pvalue = (PULONG) buffer;
    *pvalue = pdx->TheAnswer;
    instlength[0] = sizeof(ULONG);

    return WmiCompleteRequest(fdo, Irp, STATUS_SUCCESS, sizeof(ULONG), IO_NO_INCREMENT);
}
```

1. 我们被迫做这个检测以确定该缓冲区的长度能否容纳下我们将要填入的数据和数据长度值。测试的第一部分判断是否存在 `instlength` 数组。测试的第二部分确定该缓冲区能否容纳下一个 `ULONG`。在这个简单的驱动程序中，我们仅提供一个 `ULONG` 值。
2. `buffer` 参数指向一块内存区域，在这里我们放入数据。`instlength` 参数指向一个数组，在那里我们放入返回的每个数据实例的长度。在此，我们装入一个 `ULONG` 数据值(`TheAnswer` 属性的值)和其长度。
3. WMILIB 规范要求我们调用 `WmiCompleteRequest` 例程完成该 IRP。第四个参数指出数据值需要多大的缓冲区。

这里我没有讨论 `QueryDataBlock` 的 `guidindex`、`instindex`、`instcount` 参数。稍后在讨论 WMI 的复杂特征时我再回头讨论这些内容。在 `WMI42.SYS` 中，这些值分别应该是 0、0、1。

SetDataBlock 回调函数

系统会通过发出 `IRP_MN_CHANGE_SINGLE_INSTANCE` 请求来要求你改变某个类的整个实例。`WmiSystemControl` 通过调用你的 `SetDataBlock` 回调例程来处理这个 IRP。下面是这个例程的简化版本：

```
NTSTATUS
SetDataBlock(PDEVICE_OBJECT fdo,
             PIRP Irp,
             ULONG guidindex,
             ULONG instindex,
             ULONG bufsize,
             PUCHAR buffer)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    if (bufsize == sizeof(ULONG))
    {
        pdx->TheAnswer = *(PULONG) buffer;
        status = STATUS_SUCCESS;
    }
}
```

```

    info = sizeof(ULONG);
}
else
{
    status = STATUS_INFO_LENGTH_MISMATCH, info = 0;
    return WmiCompleteRequest(fdo, Irp, status, info, IO_NO_INCREMENT);
}

```

1. 系统应该知道每个类的实例有多大(基于 MOF 的声明), 因此它应该给我们正确大小的缓冲区。如果不是这样, 我们应该失败该 IRP。另外, 我们还应该备份数据块的值。
2. 我们有责任完成该 IRP, 所以调用 **WmiCompleteRequest**。

SetDataItem 回调函数

有时, 消费者仅仅想改变 WMI 对象中的某个域。每个域都有一个标识数字, 该数字出现在域的 MOF 声明中的 **WmiDataId** 属性中。**(Active** 和 **InstanceName** 属性是不可修改的, 而且也没有标识。此外, 它们是由系统实现的, 甚至不出现在我们使用的数据块中) 为了改变某个域, 消费者参考该域 ID, 后来我们就收到一个 **IRP_MN_CHANGE_SINGLE_ITEM** 请求, 该请求由 **WmiSystemControl** 通过调用我们的 **SetDataItem** 回调函数来处理:

```

NTSTATUS
SetDataItem(PDEVICE_OBJECT fdo,
            PIRP Irp, ULONG guidindex,
            ULONG instindex,
            ULONG id,
            ULONG bufsize,
            PUCHAR buffer)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    NTSTATUS status;
    ULONG info;

    if (bufsize == sizeof(ULONG))
    {
        pdx->TheAnswer = *(PULONG) buffer;
        status = STATUS_SUCCESS;
        info = sizeof(ULONG);
    }
    else
        status = STATUS_INFO_LENGTH_MISMATCH, info = 0;

    return WmiCompleteRequest(fdo, Irp, status, info, IO_NO_INCREMENT);
}

```

在 **WMI42.SYS** 例子中, 你会注意到这个 **SetDataItem** 例程与 **SetDataBlock** 例程相同, 因为我的类只有一项。

注意

显然, 在 Windows 2000 的 beta 版中用于生成调用 **SetDataItem** 例程的 WMI 系统代码还不完整。调用这个例程的唯一方法是使用 Microsoft 的一个内部测试工具, 而且还需要用一个为 0 的项来代替 MOF 规划中声明为 1 的项。我不知道这是不是这个内部工具的一个 bug。我建议你以 **STATUS_WMI_NOT_SUPPORTED** 失败这个调用, 直到你真正知道这个项目 ID 的含义。

高级特征

在前面，我们讨论了为监测程序提供性能信息的技术。想象一下：不是仅提供一个统计信息(**TheAnswer**)，你还可以累积并返回你的专有信息，而这些信息只与你的设备相关。此外，你还可以支持有更特殊目的的某些额外 WMI 特征。现在我们将讨论这些特征。

处理多实例

WMI 允许你为单个设备对象的某个类数据块创建多个实例。如果你的设备是一个控制器设备或者是可以插入其它设备的设备，你可能希望能提供多个实例，每个实例代表一个子设备的数据。你可以在 WMIGUIDREGINFO 结构中指定类的实例个数。例如，如果 WMI42 有其标准数据块的三个不同实例，那么在它的 WMILIB_CONTEXT 结构中应该有如下 GUID 列表：

```
WMIGUIDREGINFO guidlist[] = {  
    {&GUID_WMI42_SCHEMA, 3, WMIREG_FLAG_INSTANCE_PDO},  
};
```

这与以前 GUID 列表的不同之处是实例个数，这里是 3。这个列表声明这里有三个 WMI42 数据块的实例，每个都有自己的三个属性值(即 **InstanceName**、**Active**、**TheAnswer**)。

如果实例的数目发生了改变，你应该用 WMIREG_ACTION_UPDATE_GUID 动作码调用 IoWmiRegistrationControl，这将使系统向你发送另一个寄存请求，在处理这个请求时你应该使用更新后的 WMILIB_CONTEXT 结构。顺便说一下，如果你想修改你的寄存信息，你应该在自由内存池中申请 WMILIB_CONTEXT 结构和 GUID 列表而不是使用静态变量。

如果用户模式代码要枚举 GUID_WMI42_SCHEMA 的所有实例，它可以找到这三个实例。然而这个结果却使用户代码迷惑，通过枚举不可能提前知道这三个实例属于同一个设备，这无法与三个 WMI42 设备每个都给出同一类的一个实例的情况相区分。为了使 WMI 客户识别出这两种情况的不同，你的规划应该包括一个额外的属性(如设备名)。

一旦你允许多实例存在，你的各种 WMILIB 回调函数就需要修改：

- **QueryDataBlock** 应该既能返回单实例的数据块，又能返回指定索引处的任意数量的实例数据。
- **SetDataBlock** 应该解释其实例号参数以决定哪个实例需要改变。
- **SetDataItem** 也应该解释其实例号参数以定位实例。

图 10-5 显示了在需要提供数据块的多个实例的情况下你的 **QueryDataBlock** 函数如何使用输出缓冲区。假定你被要求提供两个实例的数据，开始于实例号 2。你将复制数据值(大小会不同)到该数据缓冲区，并以 8 字节边界开始每个实例。在完成该查询时你应该给出复制的总共字节数，并在 **instlength** 数组中填入每个实例的数据长度。

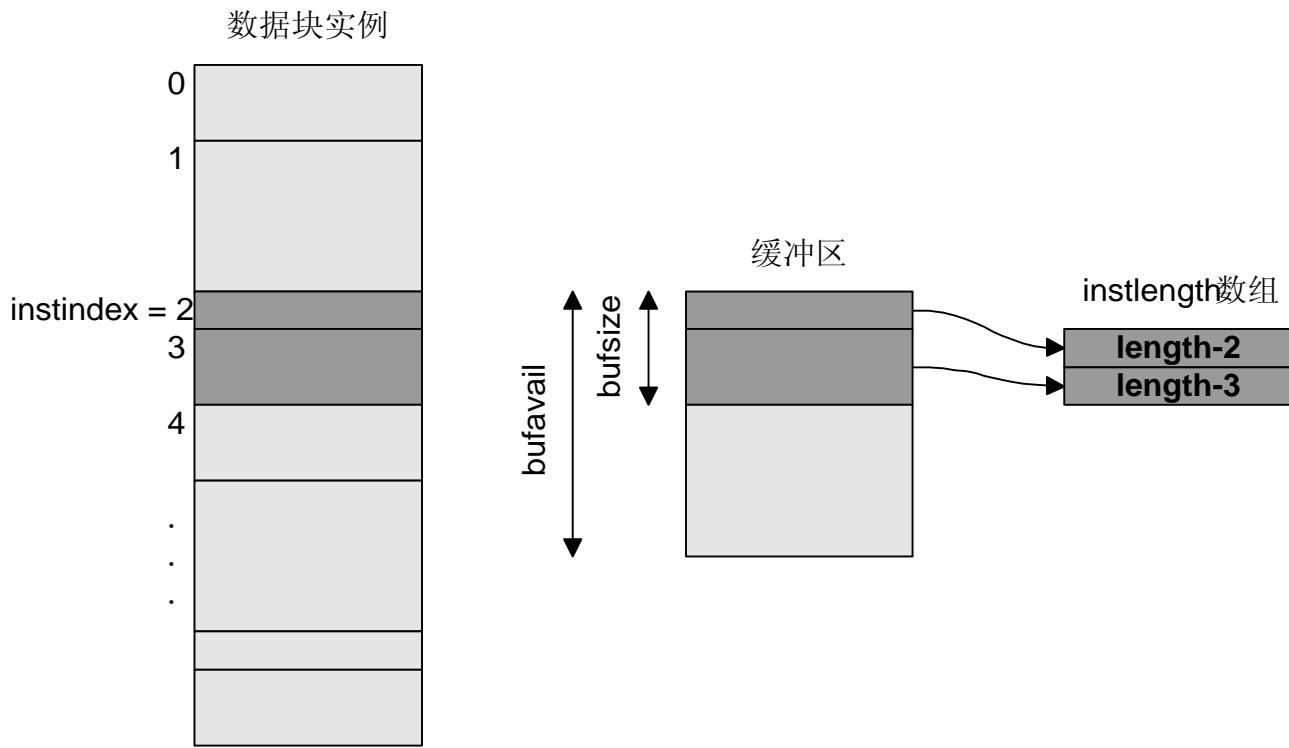


图 10-5. 获得多实例数据块

实例命名

WMI 类的每个实例都有一个唯一的名字。知道这个实例名字的消费者能够执行查询或调用方法例程。不知道实例名字的消费者也可以通过枚举该类来得到。不管什么情形你都有责任生成消费者可使用或可发现的名字。

这里有一个为客户数据块命名的最简单的方式(从驱动程序角度来看), 即请求 WMI 自动生成一个基于设备 PDO 名的静态的、唯一的名字。例如, 如果你的 PDO 名为 **Root*WCO0A01\0000**, 那么一个单实例的基于 PDO 的名字应该是 **Root*WCO0A01\0000_0**。结尾的_0 使这个名字唯一。这个名字是静态的, 可以一直维持到你反寄存或更新你的寄存信息。

基于 PDO 名的实例名很方便, 因为你所要做的就是在每个 WMIGUIDREGINFO 结构中设置 **WMIREG_FLAG_INSTANCE_PDO** 标志, 以及在 WMILIB 传递给回调函数 **QueryRegInfo** 的 **flags** 中设置该标志。但消费程序的作者并不知道这个名字, 因为这个名字会随着所安装的设备而变化。为了使实例名更可预测, 你可以用一个常量名来代替对象实例名。通过忽略 WMIGUIDREGINFO 结构中的 **WMIREG_FLAG_INSTANCE_PDO** 标志来指出这种选择, 下面是修改后的 **QueryRegInfo** 回调函数:

```

NTSTATUS
QueryRegInfo(PDEVICE_OBJECT fdo,
              PULONG flags,
              PUNICODE_STRING instname,
              PUNICODE_STRING* regpath,
              PUNICODE_STRING resname,
              PDEVICE_OBJECT* pdo)
{
    *flags = WMIREG_FLAG_INSTANCE_BASENAME;
    *regpath = &servkey;
    RtlInitUnicodeString(resname, L"MofResource");
    static WCHAR basename[] = L"WMIEXTRA";
    instname->Buffer = (PWCHAR) ExAllocatePool(PagedPool, sizeof(basename));
    if (!instname->Buffer)
        return STATUS_INSUFFICIENT_RESOURCES;
    instname->MaximumLength = sizeof(basename);
    instname->Length = sizeof(basename) - 2;
}

```

```
RtlCopyMemory(instname->Buffer, basename, sizeof(basename));  
}
```

该函数与前面例子中的 `QueryRegInfo` 函数的不同部分用粗体显示。在 `WMIEXTRA` 例子中，每个数据块仅有一个实例，名字都是 **WMIEXTRA**。

如果你使用命名前缀，应该避免使用象 `Toaster` 这样的通用名称，因为那可能会产生混乱。这个特征的目的是让你使用象 `AcmeWaffleToaster` 这样的特殊名称。

在某些场合，静态实例名不能适应需求。如果你需要维护一个会常常改变的数据块个数，使用静态名意味着每次数据块个数改变时你都要请求一个寄存更新。更新过程比较昂贵，应该避免常常做这种请求。可以动态分配一个实例名，而这个实例名就是查询的一部分。但不幸的是，`WMILIB` 并不支持使用动态实例名。因此，为了使用这个特征，你必须抛开 `WMILIB` 自己解释 `IRP_MJ_SYSTEM_CONTROL`。如何自己处理这些 `IRP` 已经超出本书的范围，详细信息可以参考 `DDK` 文档。

处理多类

`WMI42` 仅能处理一个数据块类。如果你想支持多个类，需要有一个更大的 `GUID` 信息结构数组，如 `WMIEXTRA` 中的：

```
WMIGUIDREGINFO guidlist[] = {  
    {&GUID_WMIEXTRA_EVENT, 1, WMIREG_FLAG_INSTANCE_PDO |  
     WMIREG_FLAG_EVENT_ONLY_GUID},  
    {&GUID_WMIEXTRA_EXPENSIVE, 1, WMIREG_FLAG_EXPENSIVE |  
     WMIREG_FLAG_INSTANCE_PDO},  
    {&GUID_WMIEXTRA_METHOD, 1, WMIREG_FLAG_INSTANCE_PDO},  
};
```

在调用某个回调例程之前，`WMILIB` 会在你的列表中查看与 `IRP` 相关的 `GUID`。如果该 `GUID` 不在列表中，`WMILIB` 将失败该 `IRP`。如果在，`WMILIB` 将调用你的回调函数，并把 `guidindex` 参数设置为等于该 `GUID` 在列表中的 `GUID` 索引。通过检查这个参数，你可以知道请求的是哪个数据块。

你可以在一个 `GUID` 信息结构中使用特殊标志 `WMIREG_FLAG_REMOVE_GUID`。该标志的目的是当寄存更新时从列表中删除一个特定 `GUID`。使用这个标志也可以防止 `WMILIB` 让你在一个要被删除的 `GUID` 上执行操作。

Expensive统计

有时候，收集所有对用户或管理员有潜在用途的统计信息是一个繁重的任务。例如，磁盘驱动程序(很可能是与磁盘驱动程序在同一设备堆栈上的过滤器驱动程序)收集用于显示磁盘上某扇区 `I/O` 请求频率的数据。这个数据对磁盘碎片整理程序会有用，它可以把最频繁访问的扇区数据放到磁盘中部以优化寻道时间。然而你不会希望每次都收集这样数据，因为收集数据需要内存，并且可能出现页交换这种特殊 `I/O` 请求，所以这个内存又必须是非分页的。

`WMI` 允许你声明一个特殊的 `expensive` 数据块，这样你就可以在需求时才收集其信息，下面代码摘自 `WMIEXTRA` 程序：

```
WMIGUIDREGINFO guidlist[] = {  
    ...  
    {&GUID_WMIEXTRA_EXPENSIVE, 1, WMIREG_FLAG_EXPENSIVE},  
    ...  
};
```

`WMIREG_FLAG_EXPENSIVE` 标志指出由 `GUID_WMIEXTRA_EXPENSIVE` 标识的数据块有 `expensive` 特征。

当一个应用程序对 **expensive** 数据块中的值“表达兴趣”时，WMI 就向你发送一个副功能码为 **IRP_MN_ENABLE_COLLECTION** 的系统控制 IRP。如果没有应用程序对 **expensive** 数据块中的内容感兴趣，WMI 将向你发送另一个副功能码为 **IRP_MN_DISABLE_COLLECTION** 的系统控制 IRP。如果你把这些 IRP 委托给 WMILIB，它会回头调用你的 **FunctionControl** 回调函数，或者允许或者禁止收集该数据块中的值：

```
NTSTATUS
FunctionControl(PDEVICE_OBJECT fdo, PIRP Irp, ULONG guidindex, WMIENABLEDISABLECONTROL
fcn, BOOLEAN enable)
{
    ...
    return WmiCompleteRequest(fdo, Irp, STATUS_SUCCESS, 0, IO_NO_INCREMENT);
}
```

在这些参数中，**guidindex** 是该 **expensive** 数据块在你的 GUID 列表中的 GUID 索引，**fcn** 将等于枚举值 **WmiDataBlockControl**，它指出是允许还是禁止收集 **expensive** 统计信息，**enable** 是布尔量，指出是否应该收集这个统计信息。在该函数返回前，你应该调用 **WmiCompleteRequest**。

顺便说一下，应用程序对一个数据块“感兴趣”是通过提取一个 **IWbemClassObject** 接口指针，该指针被捆绑到你的数据块的 WMI 类的特定实例上。虽然应用程序需要找到这个类的一个实例，但调用 **FunctionControl** 回调函数时并没有出现实例索引。因此收集或不收集 **expensive** 统计信息的指令将应用到这个类的所有实例上。

WMI事件

生产者可以使用 WMI 提供的方法来通知消费者其期待的事件发生。设备驱动程序也可以使用这种方法来警告用户某个设备操作需要用户干预。例如，磁盘驱动程序可以发出磁盘上有异常的大量坏块的通知。如何登记这样的事件见第九章，这也是把这个事实通知给人类世界的一种方式，管理员必须积极地查看事件登记的每一项。如果某人要写一个事件监视小程序(applet)，应该注意到这个异常并发出一个 WMI 事件，则这个事件应该能立即引起用户注意。

WMI 事件就是用特殊方法使用的常规 WMI 类。在 MOF 语句中，你必须从抽象类 **WMIEvent** 派生出这种数据块，下面是 WMIEVENT 的 MOF 文件：

```
[Dynamic, Provider("WMIProv"),
 WMI,
 Description("Event Info from WMIEExtra"),
 guid("c4b678f6-b6e9-11d2-bb87-00c04fa330a6"),
 locale("MS\0x409")]
```

```
class wmiextra_event : WMIEvent
{
    [key, read] string InstanceName;

    [read] boolean Active;

    [WmiDataId(1), read] uint32 EventInfo;

};
```

尽管事件可以是正常的数据块，但不希望应用程序分别读写它们，否则需要在 GUID 的声明中使用 **EVENT_ONLY** 标志：

```
WMIGUIDREGINFO guidlist[] = {
    ...
    {&GUID_WMIEXTRA_EVENT, 1, WMIREG_FLAG_INSTANCE_PDO |
    WMIREG_FLAG_EVENT_ONLY_GUID},
```

```
...  
};
```

当应用程序对某个事件“表达兴趣”时，WMI 就向驱动程序发送一个副功能码为 IRP_MN_ENABLE_EVENTS 的系统控制 IRP。如果不再有应用程序对这个事件感兴趣，WMI 会向驱动程序发送一个副功能码为 IRP_MN_DISABLE_EVENTS 的系统控制 IRP。如果你把这些 IRP 委托给 WMILIB 来处理，它会回头调用你的 FunctionControl 回调函数，并在你的 GUID 列表中指出 GUID 索引，以及一个 WmiEventControl 的 fcn 代码，一个 enable 布尔标志。

为了发出一个事件，首先应该在非分页内存中构造一个事件类的实例然后调用 **WmiFireEvent**。例如：

```
PULONG junk = (PULONG) ExAllocatePool(NonPagedPool, sizeof(ULONG));  
*junk = 42;  
WmiFireEvent(fdo, (LPGUID) &GUID_WMIEXTRA_EVENT, 0, sizeof(ULONG), junk);
```

在适当时间，WMI 子系统将释放由事件对象占用的内存。

WMI方法例程

除了定义传输数据和通知事件的机制之外，WMI 还为消费者调用生产者实现的方法例程规定了一种方法。下面是 WMIEXTRA 定义的只有一个方法例程的类：

```
[Dynamic, Provider("WMIProv"),  
 WMI,  
 Description("WMIExtra class with method"),  
 guid("cd7ec27d-b6e9-11d2-bb87-00c04fa330a6"),  
 locale("MS\0x409")]  
  
class wmiextra_method  
{  
    [key, read] string InstanceName;  
  
    [read] boolean Active;  
  
    [Implemented, WmiMethodId(1)] uint32 AnswerMethod([in,out] uint32 TheAnswer);  
};
```

这个声明表明 **AnswerMethod** 将接受一个名为 **TheAnswer**(一个 32 位无符号整数)的输入/输出参数并返回一个 32 位无符号整数。

如果你把系统控制 IRP 委托给 WMILIB 处理，某个方法例程会调用你的 **ExecuteMethod** 回调函数：

```
NTSTATUS  
ExecuteMethod(PDEVICE_OBJECT fdo,  
              PIRP Irp,  
              ULONG guidindex,  
              ULONG instindex,  
              ULONG id,  
              ULONG cbInbuf,  
              ULONG cbOutbuf,  
              PUCHAR buffer)  
{  
    NSTATUS status = STATUS_SUCCESS;
```

```

ULONG bufused = 0;
...
return WmiCompleteRequest(fdo, Irp, status, bufused, IO_NO_INCREMENT);
}

```

buffer 区域包含输入类的一个映像，长度为 **cbInbuf**。你的工作就是执行该方法并用输出类来覆盖这个 **buffer** 区域。你以输出类的字节大小(**bufused**)完成该请求。在 **WMIEXTRA** 中，我在省略号的地方放了如下代码(我忽略了错误检测)：

```

switch (guidindex)
{
case 2:
    bufused = sizeof(ULONG);
    (*(PULONG) buffer)++;
    break;
default:
    status = STATUS_WMI_GUID_NOT_FOUND;
    break;
}

```

这个简单的方法例程仅把输入参数值加 1。

在我写本章时，方法例程调用周围的细节部分仍不明确。下面是你要考虑的一些问题：

- 驱动程序没有办法使方法调用返回一个值。你仅能返回一个输出参数类实例。
- 输入输出参数在描述函数时一同给出。系统把这些参数描述翻译成两个 **WMI** 类：一个是输入参数，一个是输出参数。用户模式消费者能够很容易地了解这些类的内容，但在编写驱动程序时，你必须猜测对应结构的内存分配。我猜测一个 32 位无符号整数参数会占用输入/输出缓冲区中的一个 **ULONG** 位置。
- 仅枚举类实例的方法(如 **wmiextra_method**)会触发一个数据块请求。你必须使这个数据查询成功，即使这个类仅包含方法例程没有数据成员。在这种情况下，你可以以 0 数据长度来完成该查询。

标准数据块

Microsoft 已经为各种类型的设备定义了一些标准化数据块。如果你的设备属于某个已定义了标准化数据块的类，你应该在驱动程序中支持这些块。标准化类的定义可参考 DDK 中的 WMICORE.MOFF 文件，表 10-3 列出了标准数据块。

表 10-3. 标准数据块

设备类型	标准类	描述
Keyboard	MSKeyboard_PortInformation	配置和性能信息
Mouse	MSMouse_PortInformation	配置和性能信息
Disk	MSDiskDriver_Geometry MSDiskDriver_Performance	格式信息、性能信息
Storage	MSStorageDriver_FailurePredictStatus	确定驱动器是否预测了一个失败
	MSStorageDriver_FailurePredictData	失败预测数据
	MSStorageDriver_FailurePredictEvent	如果预测失败则发出事件
	MSStorageDriver_FailurePredictFunction	与失败预测相关的方法
Serial	MSSerial_PortName	端口名称
	MSSerial_CommInfo	通讯参数

	MSSerial_HardwareConfiguration	I/O 资源信息
	MSSerial_PerformanceInformation	性能信息
	MSSerial_CommProperties	通讯参数
Parallel	MSParallel_AllocFreeCounts	分配和释放操作计数
	MSParallel_DeviceBytesTransferred	传输计数

为了实现对某个标准数据块的支持，包括列表中对应的 GUID，应该实现获取和设置数据的支持代码，允许和禁止事件的支持代码，等等。不要在你自己的规划中包含标准数据块定义；那些类已经存在于 repository 中了，不要超越它们。

在许多情况下，Microsoft 的类驱动程序将提供对这些标准类的实际 WMI 支持，因此你不用做什么工作。

标准控制

Windows 2000 有一天会把 WMI 作为向驱动程序发送某些公用命令的方法。特别是管理程序将能使用 WMI 发送与电源管理有关的命令。但现在仅定义了两个这样的命令。见表 10-4。

表 10-4. 标准 WMI 命令

WMI 类(WMICORE.MOFF)	GUID 名称(WDMGUID.H)	目的
MSPower_DeviceEnable	GUID_POWER_DEVICE_ENABLE	当系统工作时设备电源能否动态开关？
MSPower_DeviceWakeEnable	GUID_POWER_DEVICE_WAKE_ENABLE	设备是否支持唤醒特征？

如果你参考 WMICORE.MOFF，你将看到 **DeviceEnable** 和 **DeviceWakeEnable** 类仅包含一个布尔成员 **Enable**，WMI 客户可以读写这个成员。在驱动程序中支持这两个类，包括 GUID 列表中的两个 GUID 的实现代码与 WMI42 中的代码十分相似，所以我不再给出这个代码。

如果你向前追溯一下 Windows 2000 的 beta 发行版，你会发现 Microsoft 最开始好象计划要实现另一个 WMI 类(名字可能为 **MSPower_DeviceTimeouts**)，用于查询并设置两个超时值，这两个值用于电源管理器寄存空闲检测。这个计划好象被搁浅了。但其 GUID 定义(GUID_POWER_DEVICE_TIMEOUTS)仍存在于 WDMGUID.H 中。

用户模式程序与WMI

WMI 使用 COM 方式支持用户模式应用程序。通过实现几个 COM 接口，Windows 管理服务就象消费者与生产者之间的信息流交易场所。生产者经由某个接口向 Windows 管理程序寄存它们的存在。消费者通过接口间接地与生产者通信。平台 SDK 中公开了所有这些接口，因此我仅描述消费者使用的重要方法例程。但开始我还要为那些对 COM 没有经验的读者解释一下使用 COM 的基本方法。

COM是什么

本段针对那些对使用 COM 接口没有经验的读者。我尽量避免直接讨论 COM，因为这是个唯一我认为难以理解的术语。

当你谈论 COM 时，你将遇到三个关键的术语。在 COM 中，一个对象就是一个软件实体，它实现了属于某个接口的方法。COM 客户需要面对的关键元素就是一个指向某接口的指针。获得接口指针有两种途径，或者是某人给你的，或者是调用了某个 API 函数返回的。从客户程序的角度看，有个神秘的实体掌握着对象的创建和销毁。

接口是什么

现在让我们慢慢地了解这三个概念，从最后一个开始。接口就是一个 C++类，但仅有一些虚拟成员函数没有数据成员，也没有非虚拟成员函数。你可以用多种语言实现或使用 COM 接口，不仅仅是 C++。但 C++是我们了解这种概念的常用方式。这里有一个简单接口的声明，是未被翻译成 COM 语言时的形式：

```
class IUnknown
{
public:
    virtual long __stdcall QueryInterface(const GUID& riid, void** ppvObject);
    virtual unsigned long __stdcall AddRef();
    virtual unsigned long __stdcall Release();
};
```

IUnknown(COM 中的对象)实现了三个公用的虚拟函数(方法): **QueryInterface**、**AddRef**、**Release**。**AddRef** 和 **Release** 用于维持对象对客户的有效性。**QueryInterface** 用于客户程序获得对象支持的接口指针。

使用 COM 的接口定义语言(IDL)，该接口描述应该象这样：

```
interface IUnknown
{
    HRESULT QueryInterface(REFIID riid, void** ppvObject);
    ULONG AddRef();
    ULONG Release();
};
```

抛开语法上的差异，这与 C++ 的类定义非常相似。IDL 编译器可以把这种接口描述形式翻译成 C 或 C++ 编译器可以理解的语法形式。有些编程语言甚至能直接理解 IDL 语法，不需要这种翻译。

就象 C++ 的类，一个接口可以派生自另一个接口。在 COM 中不可以声明超过一个基类的接口。另外，每个 COM 接口最终都派生自 **IUnknown**，既每个 COM 对象都支持 **QueryInterface**、**AddRef**、**Release** 方法。下面是一个来自 WMI 世界的例子，我们以后会用到这个例子：

```
interface IWbemLocator : IUnknown
{
```

```
HRESULT ConnectServer(BSTR strNetworkResource,
                      BSTR strUser,
                      BSTR strPassword,
                      BSTR strLocale,
                      long lSecurityFlags,
                      BSTR strAuthority,
                      IWbemContext* pCtx,
                      IWbemServices** ppNameSpace);
};
```

因此实现了 **IWbemLocator** 的对象会有四个方法例程：QueryInterface、AddRef、Release、**ConnectServer**。

对象的创建与销毁

有了接口指针你就可以与对象对话，获得接口指针有多种方法。通常的办法是调用 **CoCreateInstance**：

```
IWbemLocator* locator;
HRESULT hr = CoCreateInstance(CLSID_WbemLocator,
                           NULL,
                           CLSCTX_INPROC_SERVER,
                           IID_IWbemLocator,
                           (PVOID*) &locator);
```

CoCreateInstance 首先参考注册表来定位一个服务器，该服务器能够例化一个 **CLSID_WbemLocator** 类的对象。**CLSID_WbemLocator** 是一个 GUID，称为类标识符，它标识一种或一类 COM 对象。注册表的 HKEY_CLASSES_ROOT 分支包含一个名为 **CLSID** 的键，其子键是 COM 了解的所有类标识符的 ASCII 表达。在上面例子中，**CLSID_WbemLocator** 应该表达为{4590f811-1d3a-11d0-891f-00aa004b2e24}，它的 **InProcServer32** 子键指定了一个 DLL(名为 wbemprox.dll，是 WMI 内核的一部分)，是实现了这个类的服务器。

定位了注册表中的类键，**CoCreateInstance** 然后把指定的服务器装入到你的地址空间。**(IID_IWbemLocator** 是 WBEMCLI.H 中声明的另一个 GUID，应该在你的消费者工程文件中包含这个头文件)

调用 **CoCreateInstance** 成功之后，你就获得了一个接口指针，使用接口指针就象使用 C++ 的类指针，你可以调用接口中的任何方法函数。计算机的某个地方(甚至不是在同一个计算机中)必然存在一个实现了那些方法函数的具体对象。对象要占用存储空间，实现的执行代码也要占存储空间。一旦你利用完这个接口指针，你将销毁对象，或许还要卸载程序。但问题是，在什么时候？这就是 **AddRef** 和 **Release** 的用处。

每个 COM 对象都有一个参考计数。一旦有人获得了对象接口的指针，实现该对象的程序就增加参考计数。所以，**CoCreateInstance** 总是返回一个被引用的接口指针，你可以认为这个指针一直有效。调用 **AddRef** 可以明确地增加对象的参考计数。当你利用完接口指针后，调用 **Release** 方法，**Release** 将减少参考计数。如果参考计数为 0，服务器实现代码就删除该对象。如果这个服务器不拥有任何对象，它本身就可以被卸载了。

作为一个 COM 客户，你的工作就是在不再需要该对象时简单地释放对接口的参考。下面是典型代码：

```
IWbemLocator* locator;
HRESULT hr = CoCreateInstance(...);
if (SUCCEEDED(hr))
{
    ...
    locator->Release();
}
```

访问WMI信息

如果想在用户模式中访问 WMI 信息，你首先需要与一个特殊的命名空间建立连接。在该命名空间的上下文中，你可以找到 WMI 类的实例。你可以查询并设置与该类实例相关的的数据块，调用它们的方法例程，监视它们生成的事件。

连接一个命名空间

当连接一个 WMI 命名空间时，首先需要获得 WMI 实现的 **IWbemServices** 接口的指针。见下面代码(基于 WMI42 例子中的 TEST 程序)：

```
HRESULT hr = CoInitializeEx(NULL, 0);                                <
if (!SUCCEEDED(hr))
    return;
hr = CoInitializeSecurity(NULL,                                         <
    -1,
    NULL,
    NULL,
    RPC_C_AUTHN_LEVEL_NONE,
    RPC_C_IMP_LEVEL_IMPERSONATE,
    NULL,
    0,
    0);
if (!SUCCEEDED(hr))
{
    CoUninitialize();
    return;
}

IwbemLocator* locator;
hr = CoCreateInstance(CLSID_WbemLocator,                               <-3
    NULL,
    CLSCTX_INPROC_SERVER,
    IID_IWbemLocator,
    (PVOID*) &locator);
if (SUCCEEDED(hr))
{
    IWbemServices* services;
    BSTR pnamespace = SysAllocString(L"root\\CIMV2");
    hr = locator->ConnectServer(pnamespace,                           <-4
        NULL,
        NULL,
        0,
        0,
        &services);
    SysFreeString(pnamespace);
    if (SUCCEEDED(hr))
    {
        IClientSecurity* security;                                <
        hr = services->QueryInterface(IID_IClientSecurity, (PVOID*) &security);
        if (SUCCEEDED(hr))
        {
            security->SetBlanket(services,
                RPC_C_AUTHN_WINNT,
                RPC_C_AUTHZ_NONE,
                NULL,
                RPC_C_AUTHN_LEVEL_CONNECT,
```

```

        RPC_C_IMP_LEVEL_IMPERSONATE,
        NULL,
        EOAC_NONE);
    security->Release();
}
// use the services interface
services->Release();                                     <--6
}
locator->Release();
}
CoUninitialize();

```

1. 每个使用 COM 的程序都需要调用 **CoInitialize** 或 **CoInitializeEx** 来初始化 COM 库, 调用 **CoUninitialize** 来关闭 COM 库。
2. 先别管这个调用的原因。
3. 我们在这里例化一个 **WbemLocator** 对象并获取其 **IWbemLocator** 接口的指针。如果该调用成功, 我们最后还要释放对这个接口的参考。
4. 我们使用 **IWbemLocator** 接口来连接 CIMV2 命名空间。(在 beta 版本中, 这应该是 WMI 命名空间) 使用 **ConnectServer** 方法的一个不便之处是你必须调用 **SysAllocString** 来复制一份该命名空间的 Unicode 名称拷贝。
5. 不要管这个! 我曾用两天时间想找出在这调用 **IClientSecurity::SetBlanket** 的原因, 因为在我写本章时, 关于这个调用的 SDK 文档还没有跟上实现代码。(@#\$\$!)
6. 在这里你可以使用 **IWbemServices** 接口指针定位 WMI 类实例并访问其它的 WMI 服务。

枚举类实例

利用 **IWbemServices** 接口, 你可以枚举一个特定 WMI 类的所有实例。例如, **WMI42** 的测试程序用下面代码枚举了 **WMI42** 类的所有实例:

```

IEnumWbemClassObject* enumerator = NULL;
BSTR bs = SysAllocString(L"WMI42");
HRESULT hr = services->CreateInstanceEnum(bs,
    WBEM_FLAG_SHALLOW | WBEM_FLAG_RETURN_IMMEDIATELY |
    WBEM_FLAG_FORWARD_ONLY, NULL, &enumerator);
SysFreeString(bs);
if (SUCCEEDED(hr))
{
    while (TRUE)
    {
        ULONG junk;
        IWbemClassObject* cop = NULL;
        hr = Enumerator->Next(INFINITE, 1, &cop, &junk);
        if (hr == WBEM_S_FALSE)
            break;
        if (!SUCCEEDED(hr))
            break;
        // Use IWbemClassObject interface
        cop->Release();
    }
    enumerator->Release();
}

```

1. **IWbemServices::CreateInstanceEnum** 将为一个命名 WMI 类的所有实例创建枚举器。该接口方法有两个缺点, 第一, 类名必须以单独分配的 **BSTR** 传递。第二, 你必须初始化目标接口指针为 **NULL**, 尽管这里假定仅有一个输出参数, 如果该指针一开始就非法则会导致系统崩溃。

2. 实例枚举器的 **Next** 方法以 **IWbemClassObject** 接口指针的形式给出后续实例的指针。如果该类的实例都被枚举，则 **Next** 方法返回 **WBEM_S_FALSE**。为了避免系统崩溃，需要先初始化这个假定输出参数为 **NULL**。

项目值的获取与设置

IWbemClassObject 接口是打开驱动程序 WMI 功能的钥匙。利用该接口的指针你可以方便地获取或设置数据块中的项目值：

```
IWbemClassObject* cop;  
VARIANT answer;  
BSTR propname = SysAllocString(L"TheAnswer");  
cop->Get(propname, 0, &answer, NULL, NULL);  
VariantClear(&answer);  
  
answer.vt = VT_I4;  
answer.lVal = 6 * 9; // should be done in base 13!  
cop->Put(propname, 0, &answer, 0);  
VariantClear(&answer);  
  
SysFreeString(propname);
```

在这段代码中，我们使用一个系统串来命名我们要获取或设置的属性(即我们规划中的项目)，并使用一个 OLE VARIANT 结构(可以存储任何类型的数据)作为数据值。调用该接口的 **Get** 方法将使驱动程序得到一个 **QUERY_ALL_DATA** 或 **QUERY_SINGLE_INSTANCE** 请求。调用 **Put** 方法会使驱动程序得到 **CHANGE_SINGLE_INSTANCE** 或 **CHANGE_SINGLE_ITEM** 请求。不要试图精确预测哪种类型的 IRP 将支持 **Get** 或 **Put** 调用，因为 WMI 生产者可以自由地以任何方便的形式包装发往驱动程序的数据请求。

接收事件通知

为了接收发生的 WMI 事件通知，应用程序必须对感兴趣的事件进行寄存操作。为此你必须用一种称为 **WQL(WMI Query Language)** 的语言陈诉一个询问。WQL 就象关系数据库中的 **SQL** 语言。例如，为了表示要接收 **WMIEXTRA_EVENT** 通知，你应该提交下面查询：

```
IWbemServices* services;  
BSTR query = SysAllocString(L"select * from WMIEXTRA_EVENT");  
BSTR language = SysAllocString(L"WQL");  
IEnumWbemClassObject* enumerator = NULL;  
HRESULT hr = services->ExecNotificationQuery(language,  
                                              query,  
                                              WBEM_FLAG_FORWARD_ONLY |  
                                              WBEM_FLAG_RETURN_IMMEDIATELY,  
                                              NULL,  
                                              &enumerator);  
  
SysFreeString(language);  
SysFreeString(query);  
if (SUCCEEDED(hr))  
{  
    ...  
    enumerator->Release();  
}
```

ExecNotificationQuery 的 **flag** 参数必须明确指出。

一旦有了这个枚举接口，你就可以调用它的 **Next** 方法查询事件。例如：

```
IWbemClassObject* cop = NULL;  
DWORD junk;-  
hr = enumerator->Next(1000, 1, &cop, &junk);
```

在 **Next** 调用中我们指出, 为了获得一个事件我们将等待 1000 毫秒。如果事件已经发生或在等待期间发生, **Next** 将返给我们一个 **IWbemClassObject**(引用)指针(回忆一下前面关于驱动程序如何发出一个由 WMI 类实例代表的事件), 因此我们可以调用该对象的 **Get** 方法来查询事件的属性。

在真正的应用程序中, 你应该使用 **ExecNotificationQueryAsync** 代替 **ExecNotificationQuery** 调用。异步形式的查询允许你提供一个 **IWbemObjectSink** 接口, WMI 可以在事件发生时调用该接口。其它信息请参考平台 SDK。

调用方法例程

调用一个方法例程仅需要少数简单的语句, 见下面摘自 **WMIEXTRA** 测试程序的代码:

```
IWbemServices* services; // developed as shown earlier  
IWbemClassObject* result = NULL;  
BSTR pmethod = SysAllocString(L"AnswerMethod");  
BSTR objpath; // more about this later  
IWbemClassObject* inarg; // ditto  
  
HRESULT hr = services->ExecMethod(objpath, pmethod, 0, NULL, inarg, &result, NULL);  
...  
result->Release();  
SysFreeString(pmethod);  
<more cleanup>
```

ExecMethod 函数用于调用方法例程。**inarg** 对象用于为输入参数提供值。方法例程调用的结果将出现在 **result** 对象中。

如果不是为了两个复杂因素, 象上面这样调用一个方法例程是烦琐的。首先你必须用完整的路径名(即 **ExecMethod** 的 **objpath** 参数)寻址你要的对象。此外你还要构造并初始化一个 WMI 对象以便为该方法调用提供输入参数。见下面摘自 **WMIEXTRA** 测试程序的片段:

```
IWbemServices* services; // someone gives me this  
BSTR pclass = SysAllocString(L"wmiextra_method");  
BSTR objpath = NULL;  
HRESULT hr;  
  
IEnumWbemClassObject* enumerator = NULL;  
hr = services->CreateInstanceEnum(pclass, <etc.>);  
if (SUCCEEDED(hr))  
{  
    IWbemClassObject* instance = NULL;  
    ULONG junk;  
    hr = enumerator->Next(INFINITE, 1, &instance, &junk);  
    if (SUCCEEDED(hr))  
    {  
        VARIANT instname;  
        BSTR propname = SysAllocString(L"InstanceName");  
        hr = instance->Get(propname, 0, &instname, NULL, NULL);  
        SysFreeString(propname);  
        if (SUCCEEDED(hr))  
        {
```

```

WCHAR fullpath[256];
WCHAR escapedname[256];
<code to double backslashes in instname>
swprintf(fullpath, L"%ws.InstanceName=\"%s\"", pclass, escapedname);
objpath = SysAllocString(fullpath);
VariantClear(&instname);
}
instance->Release();
}
enumerator->Release();
}

```

尤其是用于穿过实例名的代码(文中已忽略)，把每个反斜线都改为双反斜线。我认为，这应该是 **IWbemClassObject** 接口上的一个方法，调用它可以获得对象的全路径名。这样的方法用于隐藏其它系统部件用于构造实例名的算法。但没人接受我的观点。

平台 SDK 文档中描述了如何创建输入参数(即 **ExecMethod** 的 **inarg** 参数)。下面是我在 WMIEXTRA 中的做法：

```

IWbemClassObject* cop = NULL; // the class, not an instance
hr = services->GetObject(pclass, 0, NULL, &cop, NULL);
if (SUCCEEDED(hr))
{
    IWbemClassObject* iop = NULL; // another class
    hr = cop->GetMethod(pmmethod, 0, &iop, NULL);
    if (SUCCEEDED(hr))
    {
        IWbemClassObject* inarg = NULL; // an instance of iop
        hr = iop->SpawnInstance(0, &inarg);
        if (SUCCEEDED(hr))
        {
            BSTR argname = SysAllocString(L"TheAnswer");
            VARIANT argval;
            argval.vt = VT_I4;
            argval.lVal = 41;
            hr = inarg->Put(argname, 0, &argval, 0);
            SysFreeString(argname);

            <the actual call to ExecMethod>

            inarg->Release();
        }
        iop->Release();
    }
    cop->Release();
}

```

这段代码使用数据字典来获得输入参数类(**iop** 变量)的描述。然后它创建并初始化输入参数类的一个实例(**inarg** 变量)作为调用方法例程的参数。

我没有检查，但我假定 MFC 会提供一个改进的方法来做这些事情。

Windows 98 兼容问题

一个良好的驱动程序应该支持 WMI，但原始版本的 Windows 98 没有提供对 WMILIB 的支持，因此你需要为 WMILIB 函数提供一个 VxD 桩。写一个 VxD 桩参见附录 A。(附录中对 WDMSTUB VxD 的讨论没有包括 WMILIB 函数，但它描述了构造方法)

大量的 bug 困扰着原始版本的 Windows 98 对 WMI 的支持。Windows 98 的改进版(第二版和 SP1)修复了这些 bug。(或者是其中某些。我有一个笔记本电脑，可以正常运行 Windows 2000 和 WMI。但在我的桌面系统上，WMI 不能在 Windows 98 下初始化)。标准安装方式的默认选择是不安装 WMI。为了支持 WMI，你可以使用控制面板中的添加/删除程序项，选择 Windows 安装程序标签，在 Internet 工具类中选择安装 Web 基础的企业管理项。

第十一章：USB总线

USB总线的成功关键是使用户感到了使用USB设备的方便。即插即用(PnP)概念的使用使某些硬件的安装过程得到了简化。然而，那些遗留设备如串行口、并行口、键盘，和麦克风的配置问题仍然折磨着用户。USB总线通过提供一种能连接大量自识别中低速设备的统一连接方法解决了这一问题。USB规范中指出，适合迁移到USB上的硬件限定于那些低速到中速的外设，包括调制解调器、应答机、扫描仪，和个人数字助理。即这些设备的数据传输速率低于12Mb/sec，并且能通过单一的PC接口被系统软件识别。

尽管这本书仅关注软件，但USB的某些电子和机械概念对于软件开发者仍然很重要。从普通用户的观点来看，USB的主要特征是每个设备都有同样的四芯电缆及标准化插头，都可以插入到PC后部的标准化插座或与PC相连的某个HUB上。另外，你还可以随意拔插USB设备而不管应用程序是否打开以及是否会造电气损害。本章覆盖了下面两个主题。

- 编程架构
- 使用总线驱动程序

在本章的第一部分，我将描述USB的编程架构。这个架构包含了多个概念，包括把设备附着到计算机上的层次方法，电源管理的通用方案，硬件的多层次描述符的自识别标准。USB标准把定时帧(frame)分解成数据包(packet)，从而实现设备和主机之间的数据传输。最后，在主机和设备上的端点之间，USB支持四种数据传输模式。一种模式称为等时传输(isochronous)，它可以每隔1毫秒(ms)传输固定量无错误校验的数据。其它模式为：控制传输、批量传输，和中断传输，它们仅能传输少量(64字节或更少)带有错误校验的数据。

在本章的第二部分，我将描述WDM驱动程序模型对于USB设备的附加特征。USB驱动程序高度依赖其总线驱动程序(USBD.SYS)，而不直接使用HAL函数与硬件通信。USB驱动程序为了向其硬件设备发送一个请求，首先创建一个USB请求块(URB)，然后把URB提交到总线驱动程序。例如，为了配置一个USB设备，驱动程序需要提交几个URB来读取各种描述符或发送命令，最后由USBD.SYS把请求送到总线上。

USB规范的官方修订版为1.1。该规范和许多其它文档均由USB委员会制定，他们的工作组位于<http://www.usb.org/developers/>。Don Anderson的《Universal Serial Bus System Architecture》(Addison-Wesley, 1997)，以另一种形式概括了大部分USB规范。

关于例子程序

Anchor Chips公司(<http://www.anchorchips.com/>)向我提供了一份他们的EZ-USB开发包。Anchor Chips公司的USB芯片集是以一个改进的8051微处理器以及一个核心逻辑为基础来实现USB规范中的一些低级协议。开发板上还含有附加的外部存储器、一个UART、一组按钮、一组LED读出器，用于开发和调试基于Anchor Chips软件框架的8051固件(firmware)。Anchor Chips芯片集的一个关键特征是可以通过USB下载固件。对于一个像我这样恐惧硬件，特别是EEPROM编程的程序员来说，这个特征简直太好了！

随书光盘中的USB例子驱动程序演示了最简单的USB设备以及它所能执行的一些任务。如果你碰巧有一个Anchor Chips的开发包，就可以用真正的固件实验这些例子程序。每个例子都包含一个WDM驱动程序(在SYS子目录)、一个Win32测试程序(在TEST子目录)、和一个固件程序(在EZUSB子目录)。你可以按照每个例子中的HTM文件的指导编译这些部件或者直接安装光盘中的已编译好的版本。

有一点需要指出，Anchor Chips提供的是简化版本的8051开发工具，版权由Keil Elektronik GmbH保留，如果你需要开发真正的固件，必须从Keil得到工具软件全部功能的使用许可。Keil现在提供改进的32位8051开发工具uVision2。

编程架构

USB 可以使程序员在不了解总线电气特性的情况下写出主机和设备的驱动软件。USB 规范的第五章“USB 数据流模型”和第九章“USB 设备框架”描述了许多对于驱动程序作者有用的特征。在这节中，我将概述这些章。

设备层次

图 11-1 演示了一个简单的 USB 配置拓扑。USB 主控制器与其它 I/O 设备一样直接连接到系统总线上。操作系统与主控制器通信使用 I/O 口或内存寄存器，通过普通的中断信号，系统可以接受主控制器的事件通知。主控制器连接一棵 USB 设备树。一种称为 hub 的设备作为其它设备的连接点。多个 hub 能以菊花链方式连接，可以连接到 USB 规范中定义的最大深度。其它设备，如照相机、麦克风、键盘等等，直接连到 hub 上。为了精确地表达概念，USB 使用术语 **function** 来描述非 hub 设备。

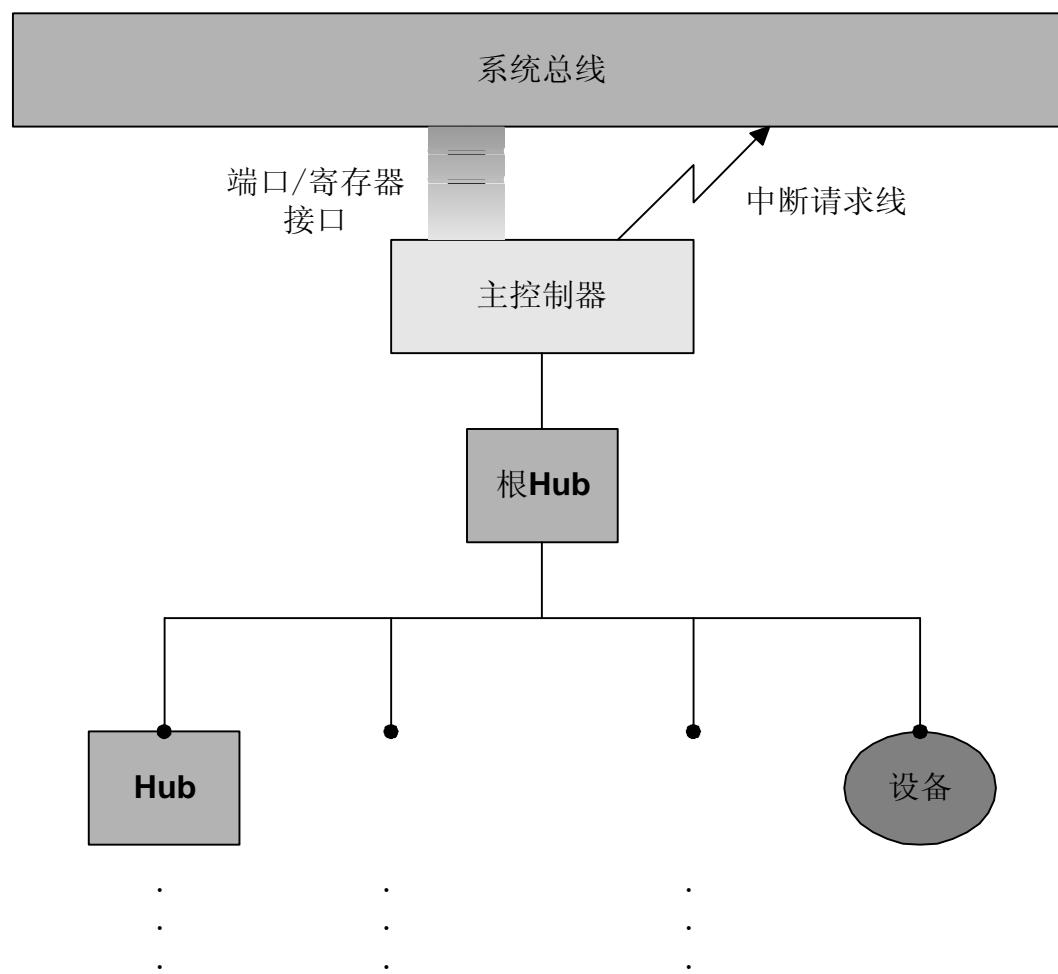


图 11-1. USB 设备层次结构

高速和低速设备

USB 规范中定义了两种设备，高速设备和低速设备。低速设备以 1.5Mb/sec 速率通信，高速设备以 12Mb/sec 速率通信。hub 能用电子方式区分这两种设备。发生在总线上的通讯通常都是高速的，hub 一般不向低速设备发送数据。操作系统把任何发往低速设备的消息前加上一个前导包，这将使 hub 临时降为低速，并完成低速设备的数据发送。

电源

USB 电缆中含有两条电源线。hub 可以为连接在其上的设备提供电力，USB 规范限定了总线供电设备所消耗的电流。这个限定会因为设备插入可供电 hub 或与最近可供电 hub 的远近而改变。另外，USB 允许设备有睡眠状

态，处于睡眠状态下的设备仅消耗非常低的电力并足够支持唤醒和配置信号。另外，设备还可以不用总线电力而直接使用外接电源。

USB 设备可以唤醒睡眠中的系统。当系统进入节能状态后，操作系统也把 **USB** 总线置入节能状态。一个有远程唤醒特征的设备能发出唤醒信号，信号首先被送到 hub，然后又被 hub 送到主控制器，最后由主控制器送往系统。

USB 设备设计者应该知道唤醒特征的一些限制。首先，远程唤醒功能仅能工作在有高级电源配置接口(ACPI)BIOS 的计算机上，而不能工作在早期的系统上，这些早期系统或者支持 APM 或者根本就没有电源管理特征。另一个限定在驱动程序的通知上。**WDM** 提供了一种方法——**IRP_MN_WAIT_WAKE** 电源管理 IRP，当设备唤醒系统时该 IRP 被发往驱动程序。然而，当 **USB** 设备从节能状态恢复到正常状态时，而此时系统又处于正常工作状态，该 IRP 将不出现。

设备中有什么？

一般，每个 **USB** 设备有一个或多个配置(configuration)来控制其行为，如图 11-2。使用多配置的一个原因是操作系统的支持，例如，系统 BIOS 可以使用一个简单的配置而操作系统中的驱动程序则使用另一个更复杂的配置。

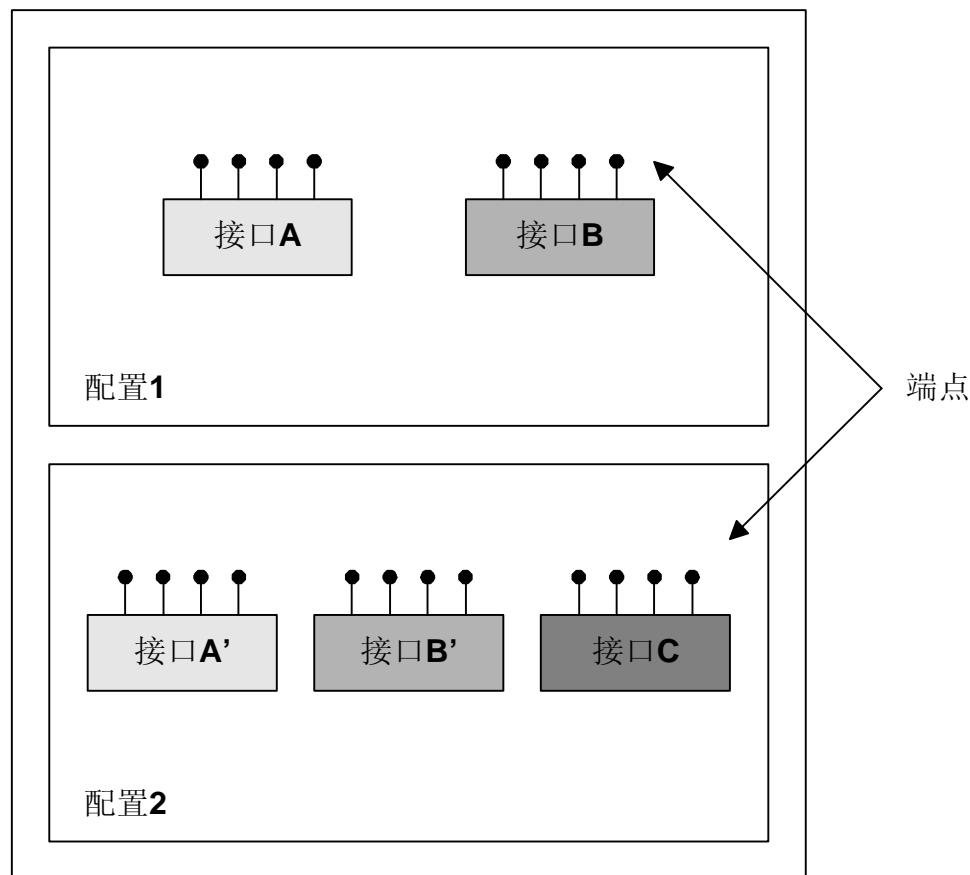


图 11-2. **USB** 设备的配置、接口、端点。

设备的每个配置中都含有一个或更多的接口(interface)，接口指出软件应该怎样访问硬件。接口的概念与第二章(“**WDM** 驱动程序的基本结构”)中讨论的连接命名设备中的概念类似，即支持相同接口的设备本质上可以互换，因为它们以相同的方式响应相同的命令。另外，接口一般都有替换设置(alternate setting)以适应不同的带宽需求。

设备的接口露出一个或多个端点(endpoint)，端点作为通信管道的一个终点。图 11-3 显示了一个多层次结构的通信模型，它表明了端点和管道所扮演的角色。在最低一级，**USB** 电缆把主控制器与设备的总线接口连接起来。在第二级，一个控制管道把系统软件与逻辑设备连接起来。在第三级，一捆数据管道把客户软件与一组接口连接起来，这些接口组成设备的 function。信息实际上是在图中两侧垂直流动，但把它理解为在这些分层的管道中水平流动更清晰。

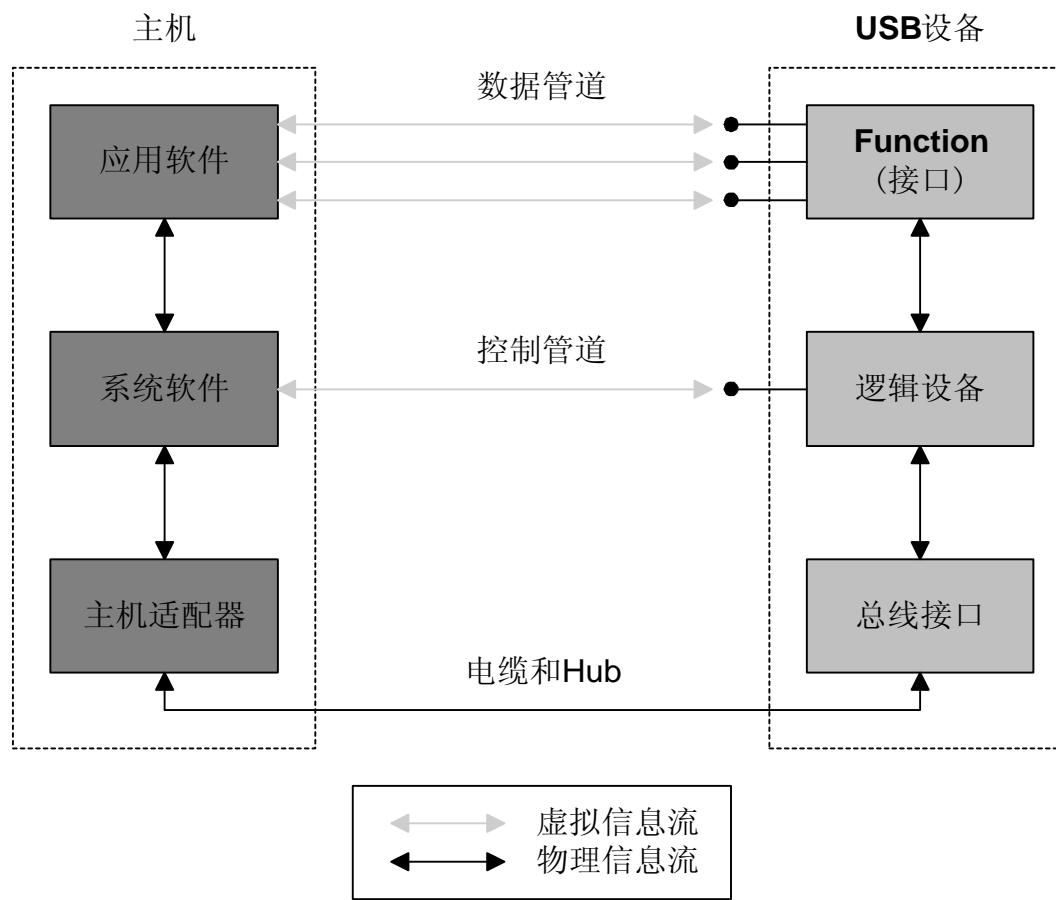


图 11-3. USB 的多层次通信模型

一组由 Microsoft 提供的驱动程序占据了图中系统软件方块中的底部。这些驱动程序包括主控制器驱动程序(OPENHCI.SYS 或者 UHCD.SYS), hub 驱动程序(USBHUB.SYS), 和一个类驱动程序(USBD.SYS), 由控制器驱动程序使用。为了方便, 我把 USBD 下面的所有驱动程序看成一个整体, 我们的驱动程序主要就是与这个整体进行交互, 它们管理着硬件连接和管道通信。WDM 驱动程序, 就是你和我将要写的, 占据系统软件方块中的顶部。广义地说, WDM 驱动程序的工作就是把客户软件的请求翻译成 USBD 能执行的事务(transaction)。客户软件处理实际的设备功能。例如, 一个图象生成程序占据的客户软件槽可能与一个静态图象 function(如数码相机)对应。

信息流动

USB 定义了四种数据传输方式, 如表 11-1 所示。它们的不同之处有: 单个事务能携带的数据量(下一段将解释术语“事务”transaction)、能否保证特定的周期或延迟, 能否自动校正错误。每种传输方式对应特定类型的端点。实际上, 给定类型的端点(控制、批量、中断、等时)总是使用对应类型的传输。

表 11-1. 数据传输类型

传输类型	描述	纠错	包容量(字节)	延迟保证?
控制	用于发送和接收 USB 定义的结构化信息	是	少于或等于 8,16,32,64	尽最大能力保证不延迟
批量	用于发送或接收小块无结构数据	是	少于或等于 8,16,32,64	无
中断	与批量管道相似, 但包括一个最大延迟	是	少于或等于 64	以保证的最小速率轮询
等时	用于发送或接收有周期保证的大块无结构数据	否	少于或等于 1023	每 1 毫秒帧中的固定部分

端点除了传输类型外还有其它几个属性。其中一个属性是单一事务中端点能够提供或消耗的最大数据量。控制和批量端点必须指定某个离散值, 而中断和等时端点能指定少于或等于最大值的任何值。端点的另一个属性是传输方向, 输入(数据从设备到主控制器)或输出(数据从主控制器到设备)。最后, 每个端点都有一个端点号, 其中包含输入输出方向, 作为端点的地址使用。

当主控制器要求设备执行某些多少有些规则的功能时，USB 使用一个轮检(polling)协议。当一个设备需要向主控制器发送数据时，主控制器必须注意到并且向要发送数据的设备发出一个请求使其发送数据。即 USB 设备不用传统方式中断主计算机，而是提供中断端点，主机周期轮检中断端点。

信息打包

当客户程序通过 USB 管道发送或接收数据时，它首先调用 Win32 API，调用最终将使 function 的驱动程序收到一个 IRP。而驱动程序的工作就是把客户的请求引导到有正确端点的管道上。它把请求提交到总线驱动程序，总线驱动程序再把请求分解成多个事务(transaction)，然后这些事务被送往总线。总线上的信息流以每毫秒一帧数据的形式流动。总线驱动程序必须安排好多个事务以便它们能被装入同一帧中，图 11-4 显示了这个过程。

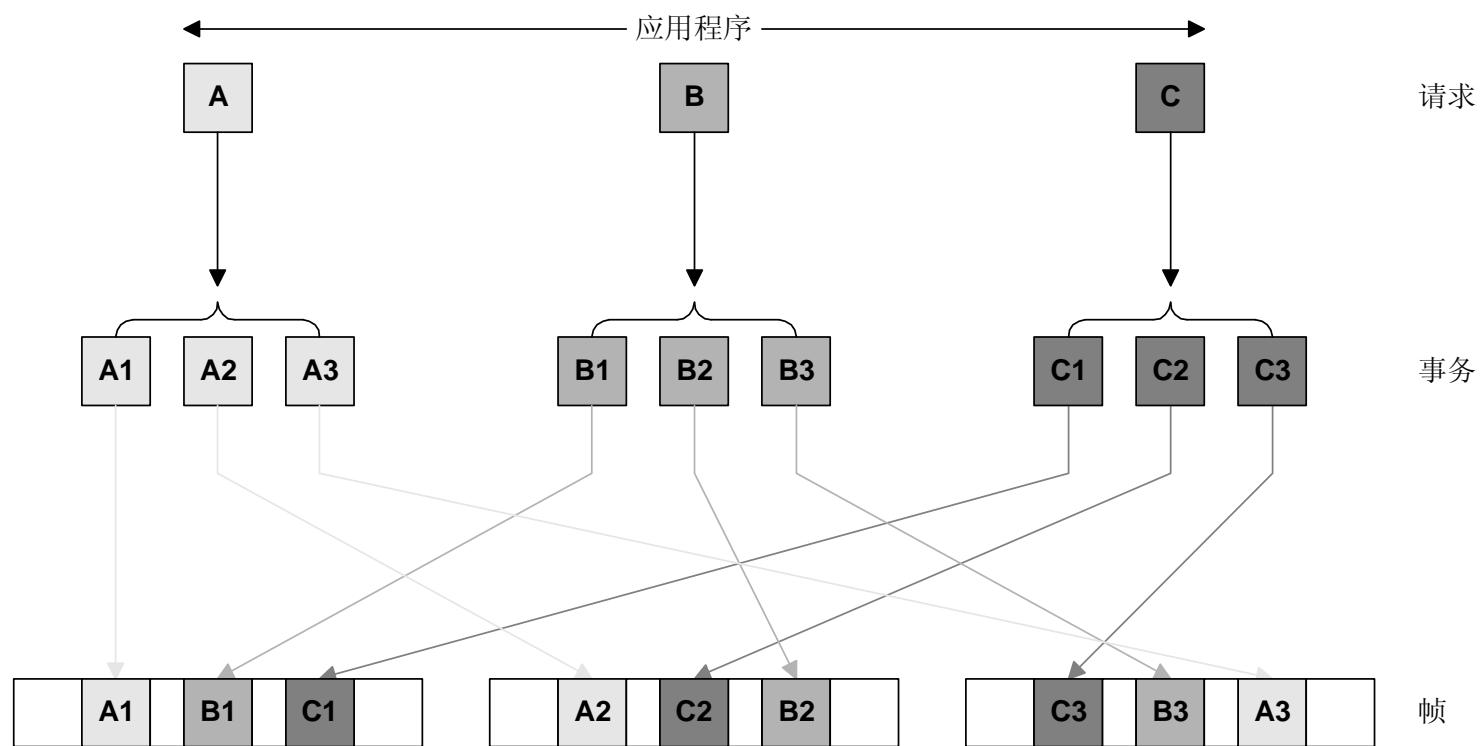


图 11-4. 信息流中的事务和帧模型

在 USB 中，事务由一个或多个阶段(phase)组成。阶段有令牌(token)、数据(data)、握手(ack)三种类型。根据不同的类型，事务有一个令牌阶段、一个可选的数据阶段、和一个可选的握手阶段组成，如图 11-5 所示。在令牌阶段，主控制器向所有已配置的设备广播该令牌包。令牌包中含有设备地址，通常还有端点号，仅有被寻址的设备才会处理事务；当事务寻址设备时，任何设备都不读写总线。在数据阶段，数据被放到总线上。对于输出事务，主机把数据放到总线上，而被寻址的设备消耗这些数据。对于输入事务，情况相反，设备把数据放到总线上由主机消耗。在握手阶段，由设备或主机把握手包放到总线上，包中含有状态信息。当设备发出握手包时，ACK 包指出成功地接收了信息，NAK 包指出忙并且不试图接收信息，STALL 包指出事务被正确接收但在逻辑上无效。当主机发送握手包时，它仅能发送 ACK 包。

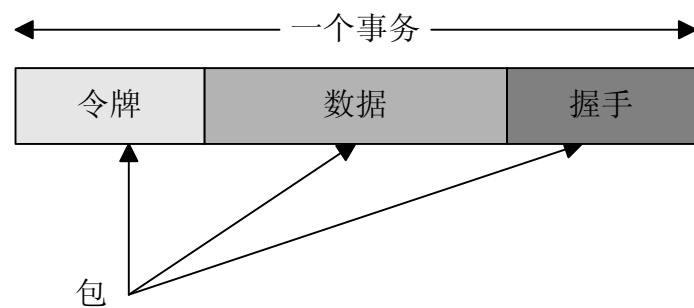


图 11-5. 总线事务的阶段

你也许会注意到，没有发出握手包的事务就代表“在这个事务中出现了一个传输错误”。正在等待握手包的一方应认为缺少握手包可能是发生了错误并重试刚才的事务。**USB** 的设计者确信这种错误很少发生，由于重试而造成的偶然延迟对于总线吞吐量不会有大的影响。

关于设备寻址的更多内容

上文提到所有被配置的设备都接收每个事务中的电子信号。这几乎是正确的，但一个真正的程序员应该了解更详细的内容。当一个 **USB** 设备第一次接入时，它使用默认地址(碰巧是 0，你不必知道)响应。然后，某个电子信号通知总线驱动程序有一个新设备插入总线，于是总线驱动程序找出一个未用的设备地址并发送一个控制事务告诉“0 号设备”什么才是它的真实地址。这之后，设备就放弃使用默认地址 0，而用真实地址来应答。

另一些细节涉及到低速设备。低速设备的电气特征使它在面对传输率是其八倍的总线时会发生信号混乱。另外，低速设备的电缆没有电磁屏蔽，遇到高速信号会产生电子干扰。结果，低速设备在大部分时间并不连接到总线上。当总线进行高速传输时，**hub** 就隔离低速设备。当主机需要与低速设备通信时，它就发送一个特殊的前导包把总线临时切换到低速操作，所以，低速设备只能看到低速事务，而高速设备能看到所有事务。

端点的状态

一般，端点可以进入图 11-6 中所示的任何一种状态。在空闲状态中，端点准备处理主机发起的新事务。在忙碌状态中，端点忙于处理手中的事务而不能接受新事务。如果主机试图向一个忙端点发起新事务(不是控制端点，下段描述)，设备将用 **NAK** 握手包响应，主机将在以后重发刚才的包。如果设备发现自己内部出现错误(不包括传输错误)，设备将在当前事务中发送一个 **STALL** 握手包然后进入停止状态。控制端点在接到新事务时会自动从停止状态恢复，但其它三种端点必须由主机明确发送一个清除特征(**feature**)控制请求后才能从停止状态中恢复。

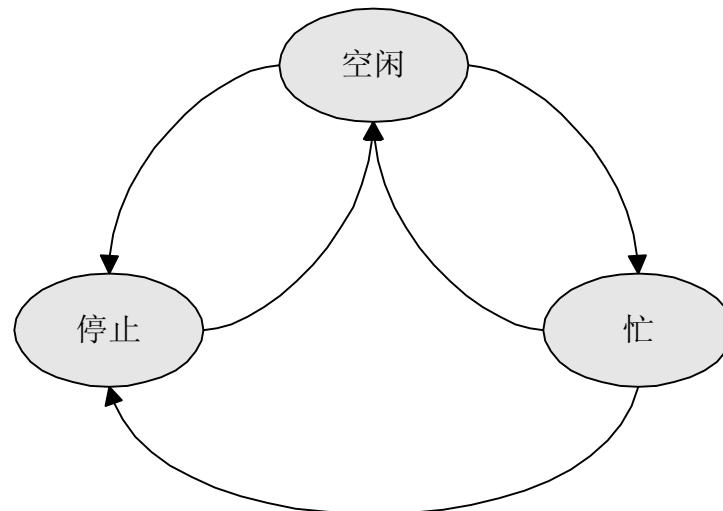


图 11-6. 端点的三种状态

控制传输

控制传输在主机和控制端点之间传送控制信息。例如，在操作系统配置 **USB** 设备过程中，有一步就是用控制传输从物理设备中读出各种描述符(**descriptor**)，配置过程的另一步是利用控制传输设置一种可能的配置并使能一个或多个接口。控制传输是一种纠错传输，在遇到传输错误时重试三遍，如果错误仍存在就放弃传输并向上层软件报告错误。如表 11-1 中所指出的，控制端点所能指定的最大数据传输长度为 8、16、32、64 字节。一个单独的事务可以包含少于最大数据传输长度的数据但不能多于。

控制事务在 **USB** 中具有最高优先级。设备必须处理控制事务。此外，总线驱动程序为控制事务保留了 10% 的带宽。对于轻量级的负载，主机能确保在 1 毫秒内完成一个控制事务。对于重量级的负载，未完成的控制传输将被强制到下一帧中传输，因此会产生一些延迟。

每个 USB 设备至少应有一个编号为 0 的控制端点以响应控制事务的输入输出。严格地讲，端点应属于配置，但端点 0 是一个例外，因为它是设备默认控制管道的终点。端点 0 甚至在设备被配置前就被激活而不管其它端点是否有效。除了端点 0，一个设备没有必要拥有另外的控制端点(尽管 USB 规范中允许有这种可能)，因为端点 0 对于大部分控制请求都可以很好地完成。如果你定义了一个厂商专用的请求并且该请求不能在一帧中完成，你应该创建额外的控制端点以防止设备接收器被新事务抢先。

每个控制事务包括一个 **SETUP** 令牌，之后可以带一个可选的数据阶段和一个握手阶段，在握手阶段设备会发出 **ACK** 包或者 **STALL** 包，或者根本就不响应，如图 11-7。设备必须在任何时间都能接受控制传输，并且不能用 **NAK** 响应控制端点忙。向控制端点发送一个无效请求将导致 **STALL** 响应，但当设备再次接收到一个 **SETUP** 包时会自动清除停止状态。这个特殊的 **STALL** 在 USB 规范(8.5.2.4 段)中被称为协议停止(protocol stall)。

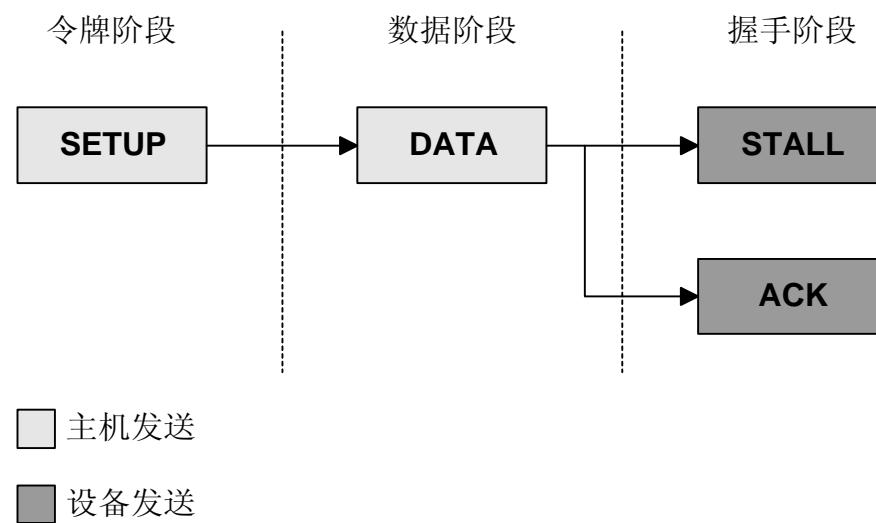


图 11-7. 控制传输中的三个阶段

开始控制传输的 **SETUP** 令牌由 8 个字节组成，如图 11-8 所示。在下面这个和后面的数据结构图中，我按照数据在 USB 导线上的传输顺序列出其中的数据字节，但是每个字节中的位是以高位开始。在传输线上的字节传输是从低位开始的，但主机软件和设备固件通常以相反的方向使用字节。Intel 计算机和 USB 总线协议使用小结尾的数据表达，即低位字节占用低地址。许多 USB 芯片集包括 Anchor Chips 芯片集使用 8051 微处理器，这个处理器使用大结尾的数据表达。固件程序设计者必须注意这一点。

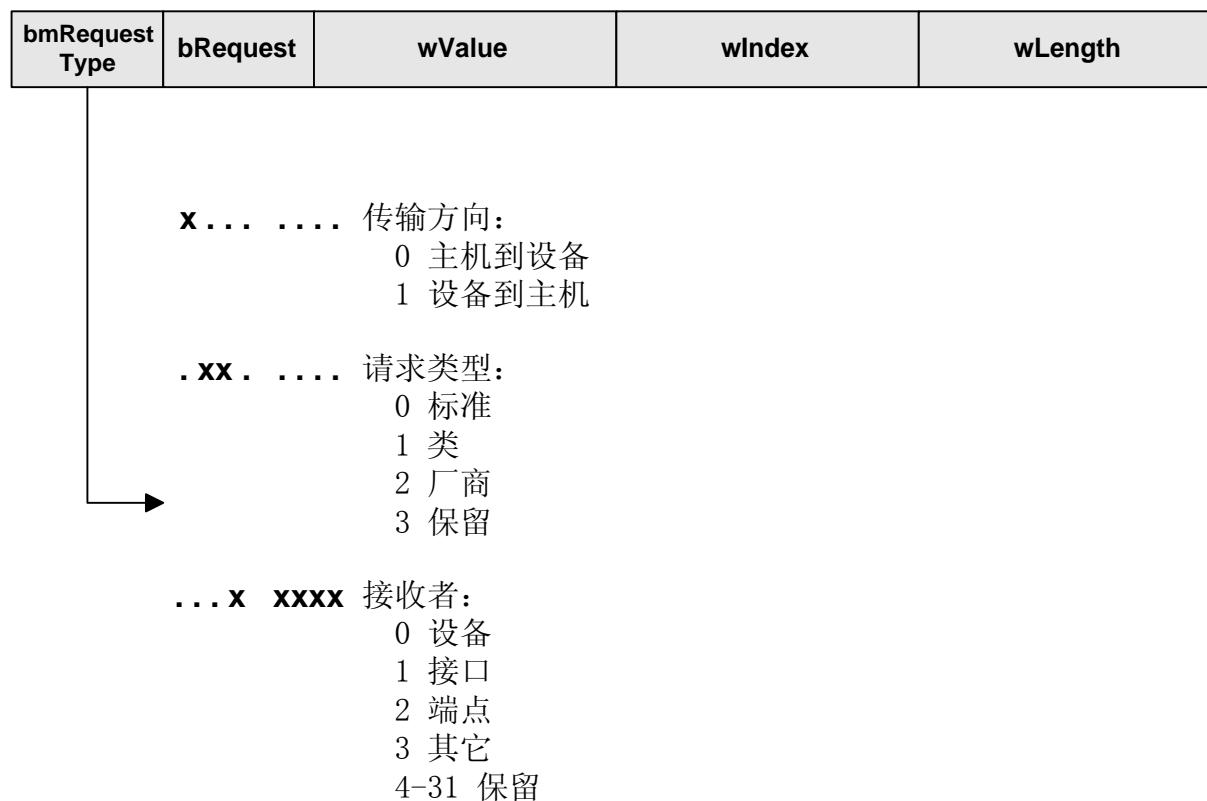


图 11-8. **SETUP** 令牌的内容

SETUP令牌的第一字节指出信息流的方向、请求的类型、和接收控制传输的实体类型。请求类型有标准类型(由USB规范定义)、类类型(由USB工作组定义的一类设备)，和厂商类型(由设备制造者定义)。控制请求可以发到整个设备、一个指定接口、一个指定的端点，或者到设备上的厂商专有实体。**SETUP**令牌中的第二字节指出具体的请求，其类型由第一字节指出。表 11-2 列出了当前定义的标准请求。对于类专用请求，应参考对应设备类的说明(<http://www.usb.org/developers/>)。设备制造者可以自由定义自己专用的请求代码。例如，**Anchor Chips** 使用请求代码 A0h 表示从主机下载固件程序。

注意

改变端点状态的控制请求将发往控制端点而不是发往被改变状态的端点。

表 11-2 标准设备请求

请求代码	符号名	描述	可能的接受者
0	GET_STATUS	获得状态信息	任何
1	CLEAR_FEATURE	清除一个双态特征	任何
2		(保留)	
3	SET_FEATURE	设置一个双态特征	任何
4		(保留)	
5	SET_ADDRESS	设置设备地址	设备
6	GET_DESCRIPTOR	取设备、配置，或串描述符	设备
7	SET_DESCRIPTOR	设置一个描述符(可选)	设备
8	GET_CONFIGURATION	取当前配置索引	设备
9	SET_CONFIGURATION	设置一个新的当前配置	设备
10	GET_INTERFACE	取当前的 alt 接口索引	接口
11	SET_INTERFACE	使能 alt 接口设置	接口
12	SYNCH_FRAME	报告同步帧号	(等时)端点

SETUP 包中其余的内容包括一个 **value** 代码(其含义与具体的请求相关)、一个 **index** 值(当控制请求寻址端点或接口时，**index** 值指出具体地址)、一个 **length** 域(指出控制事务的数据阶段要传输的数据量。为 0 表示该事务没有数据阶段)。

我并不想详细描述各种控制请求的细节；你可以参考 USB 规范的第 9.4 段。但我确实想简要地讨论一下设备特征(**feature**)这个概念，USB 认为，属于设备的任何可寻址实体都可以有 1 个特征位。已有两个特征实现了标准化。

DEVICE_REMOTE_WAKEUP 特征 —— 属于设备整体的特征，它指出当外部事件发生时设备能否用这个能力(如果有)唤醒计算机。主机软件(尤其是总线驱动程序)通过 **SET_FEATURE** 或 **CLEAR_FEATURE** 命令可以允许或禁止这个特征，用 **value** 代码为 1 来指出允许唤醒特征。DDK 中用符号名 **USB_FEATURE_REMOTE_WAKEUP** 代表这个特征码。

ENDPOINT_HALT 特征 —— 属于单个端点的特征，指出端点是否处于停止状态。主机软件可以向端点发送 **SET_FEATURE** 命令并且 **value** 为 0(指定 **ENDPOINT_HALT**)来强制端点进入停止状态。当然，管理端点的设备固件也能使端点进入停止状态。如果端点被固件设为停止状态，主机软件(总线驱动程序)也可以发送一个 **value** 为 0 的 **CLEAR_FEATURE** 命令清除该端点的停止状态。DDK 使用符号名 **USB_FEATURE_ENDPOINT_STALL** 代表这个特征码。

USB 规范中并没有为厂商使用的设备或端点的特征代码规定范围。为了避免后来的标准化问题，你应该避免定义设备级或端点级的特征，而仅应该定义自己的厂商类型控制事务。在本章后面我将演示一个例子驱动程序(**FEATURE**)，它能控制 **Anchor Chips** 开发板上的 7 段 LED 显示。我在这个例子中定义了一个编号为 42 的接口级特征。(USB 现在为电源管理定义了一些接口级特征，所以你不应效仿我的例子，除非你想知道特征是怎样工作的)

尽管 Anchor Chips(现在是 Cypress Semiconductor)的 EZ-USB 可以容易地从驱动程序下载新固件，但你不应在产品级设备上使用这个特征。你需要开发一个“**Loader**”驱动程序和一个 **function** 驱动程序，前者用于下载固件到 **USB** 设备，后者用于管理设备。不幸的是，如果计算机进入节能状态，操作系统也把 **USB** 总线置入节能状态，设备的固件内容将丢失。当回到正常电源状态时，系统卸载 **function** 驱动程序并重新装载“**Loader**”驱动程序，**Loader** 再次下载固件并重新装载 **function** 驱动程序。而节能前的 **function** 驱动程序的状态信息将全部丢失，所以当你使用这个芯片时应考虑把固件放到 **EEPROM** 上。

批量传输

批量(bulk)传输能在主机和批量端点间一次传输最多 64 字节的数据。就象控制传输一样，批量传输是纠错传输，不同的是批量传输没有任何延迟保证。如果主机发现帧中除了预定带宽外还有剩余，它就把等待的批量传输送往总线。

图 11-9 显示了组成批量传输的各个阶段。传输开始于表明传输方向的 **IN** 或 **OUT** 令牌，这些令牌同时还指定目标设备和端点。对于输出事务，后面是一个数据阶段，把数据从主机送到设备，最后在握手阶段设备向主机提供状态反馈。如果端点正忙不能接收新数据，它就在握手阶段发出一个 **NAK** 包，主机以后会重试该事务。如果端点处于停止(STALL)状态，它就在握手阶段发出一个 **STALL** 包，主机在重试前必须清除端点的停止状态。如果端点正确地接收并处理了数据，将在握手阶段返回一个 **ACK** 包。最后一种情况是端点因为某种原因没有正确地接收数据并且在握手阶段也没有发出应答包，主机将检查端点是否应答，并且自动重试三次原事务。

IN 令牌后面是一个批量输入传输。如果端点将数据准备好，设备将把数据发往主机，主机或者回应一个 **ACK** 包指出数据无误地接收，或者保持沉默以指出发生了某种错误。如果主机查出一个错误，**ACK** 的缺发将使设备数据保持有效，之后主机将重试输入操作；如果端点忙或停止，设备将会在数据阶段发出 **NAK** 或 **STALL** 握手包而不是数据。**NAK** 指出主机应在以后重试该输入操作，**STALL** 需要主机先清除端点的停止特征再重试输入操作。

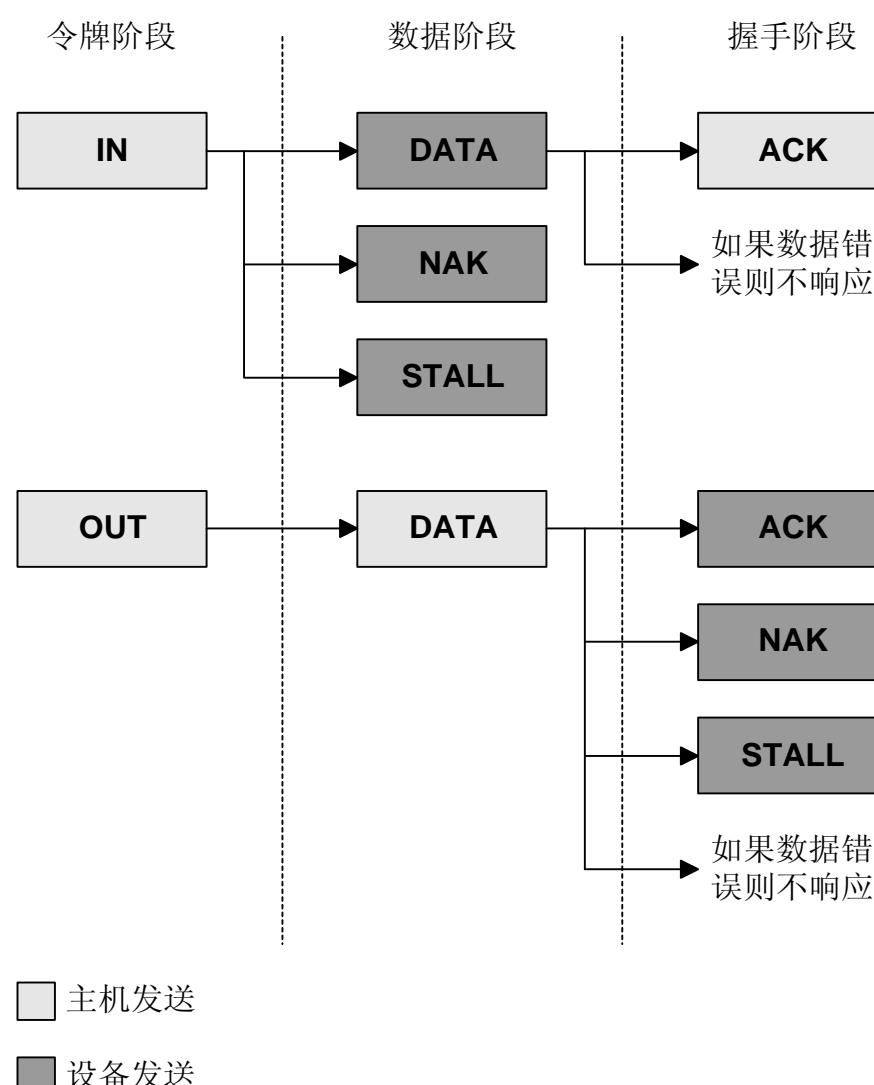


图 11-9. 批量传输和中断传输中的阶段

中断传输

中断(interrupt)传输在总线操作和涉及设备方面上几乎与批量传输完全相同。它能把最大 64 字节的数据无误地在主机和中断端点间传输。中断传输和批量传输仅有的不同是它必须考虑延迟。中断端点需指定一个范围在 1-255 毫秒内的查询周期。主机保留足够的带宽以确保在指定频率上直接向中断端点发出 IN 或 OUT 事务。

注意

注意 USB 设备不生成异步中断，它们总是响应循检，Microsoft 的主控制器驱动程序把中断端点描述符中指定的循检周期简化为不大于 32 的 2 的幂。例如，一个指定循检周期为 31 毫秒的端点实际上是以 16 毫秒进行循检。指定周期在 32-255 毫秒之间的循检实际上是以 32 毫秒为循检周期。

等时传输

等时(isochronous)传输可以在一个总线帧内最多传输 1023 字节数据。因为等时传输有周期保证，因此特别适用于时间敏感的数据传输，如音频信号，但这种周期保证是有代价的，等时传输在数据出错时不会自动重试。USB 设计者假定等时数据流的接收者允许偶尔的数据丢失。

等时传输由 IN 或 OUT 令牌阶段后跟一个数据阶段组成。因为不进行数据纠错，所以等时传输没有握手阶段。如图 11-10。

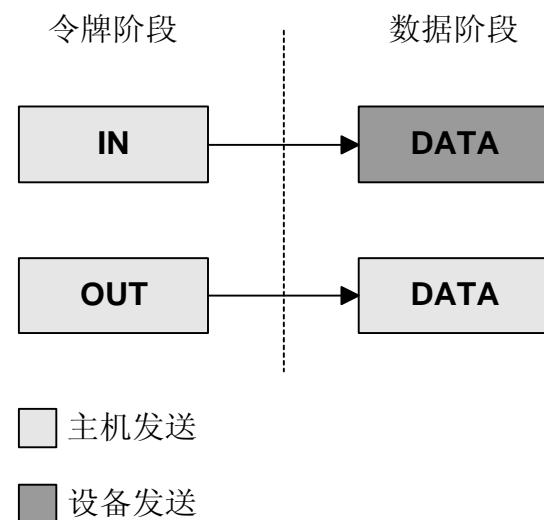


图 11-10. 等时传输中的阶段

主机为等时传输和中断传输保留最多 90% 的带宽。实际上，系统软件应提前保留带宽以确保总线能适应所有活动的设备。

描述符

USB 设备硬件中的数据结构称为描述符，可以被主机软件识别。表 11-3 列出了不同种类的描述符。每个描述符开始于一个两字节的头，头中指出该描述符的字节长度(包括头)和描述符类型。事实上，如果我们不讨论特殊的串描述符，描述符的长度对于相同的描述符类型是固定的，即所有给定类型的描述符长度相同。在描述符头中保存明确的长度便于描述符将来的扩展。

下面我将使用 DDK(位于 USB100.H 文件中)中定义的数据结构描述每一种描述符。官方的解释见 USB 规范第 9.6 段。

表 11-3. 描述符表

描述符类型	描述
设备	描述整个设备
配置	描述设备的一个配置
接口	描述配置中的一个接口
端点	描述接口中的一个端点
串	一个 Unicode 串，该串用自然语言描述设备、配置、接口，或端点
电源配置	描述电源管理能力
接口电源	描述 function 的电源管理能力

设备描述符

每个设备都有一个唯一的设备描述符，它向主机软件标识该设备。主机使用 **GET_DESCRIPTOR** 控制事务直接从设备的 0 号端点读取该描述符。该描述符在 DDK 中的定义如下：

```
typedef struct _USB_DEVICE_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    USHORT bcdUSB;
    UCHAR bDeviceClass;
    UCHAR bDeviceSubClass;
    UCHAR bDeviceProtocol;
    UCHAR bMaxPacketSize0;
    USHORT idVendor;
    USHORT idProduct;
    USHORT bcdDevice;
    UCHAR iManufacturer;
    UCHAR iProduct;
    UCHAR iSerialNumber;
    UCHAR bNumConfigurations;
} USB_DEVICE_DESCRIPTOR, *PUSB_DEVICE_DESCRIPTOR;
```

设备描述符的 **bLength** 域应等于 18, **bDescriptorType** 域应等于 1 以指出该结构是一个设备描述符。**bcdUSB** 域包含该描述符遵循的 USB 规范的版本号(以 BCD 编码)。现在，设备可以使用值 0x0100 或 0x0110 来指出它所遵循的是 1.0 版本还是 1.1 版本的 USB 规范。

bDeviceClass、**bDeviceSubClass**、**bDeviceProtocol** 指出设备类型。可能的设备类代码在 USB 规范中定义，本书所包括的类代码在表 11-4 中列出。USB 委员会的独立设备类工作组为每个设备类定义子类和协议代码。例如，音频类有控制、流，和 MIDI 流接口的子类代码。大容量存储类为使用各种端点的数据传输方法定义了协议代码。

你可以为整个设备或仅在接口级指定一个类，但事实上，设备类、子类、和协议代码通常出现在接口描述符中而不是出现在设备描述符中。USB 还为特殊类型的设备指定了一个特殊的设备类代码 255。厂商可以使用这个代码指出其设备是一个非标准设备，并且在子类和协议域中填入厂商设定的值。例如，使用 Anchor Chips 芯片集的设备，其设备描述符中的类、子类，和协议代码全为 255。

设备描述符的 **bMaxPacketSize0** 域给出了默认控制端点(端点 0)上的数据包容量的最大值。每个设备都必须提供 0 号控制端点，由于 USB 规范并没有为该端点规定一个单独的端点描述符，所以这个域是唯一描述这个端点的地方。因为这个域在设备描述符的偏移 7 处，所以即使该端点使用最小的传输容量(8 字节)主机也能读到这个域。一旦主机知道了端点 0 的最大传输容量，它就可以分块读出整个描述符。

idVendor 和 **idProduct** 域指定厂商代码和厂商专用的产品标识。**bcdDevice** 指出设备的发行版本号(0x0100 对应版本 1.0)。当主机软件检测设备时，这三个域决定了主机应该装入哪个驱动程序。USB 组织提供厂商代码，厂商提供产品代码。

表 11-4. USB 设备类代码

符号名	类代码	描述
USB_DEVICE_CLASS_RESERVED	0	指出类代码存在于接口描述符中
USB_DEVICE_CLASS_AUDIO	1	操作模拟或数字音频、语音、和其它与声音相关的数字设备
USB_DEVICE_CLASS_COMMUNICATIONS	2	电讯设备，如调制解调器、电话、应答机，等等
USB_DEVICE_CLASS_HUMAN_INTERFACE	3	人类接口设备，如键盘、鼠标、麦克风，等等
USB_DEVICE_CLASS_MONITOR	4	显示器
USB_DEVICE_CLASS_PHYSICAL_INTERFACE	5	含有实时物理反馈的人类接口设备，如力反馈游戏杆
USB_DEVICE_CLASS_POWER	6	执行电源管理的人类接口设备，如电池、充电器，等等
USB_DEVICE_CLASS_PRINTER	7	打印机
USB_DEVICE_CLASS_STORAGE	8	大容量存储设备，如磁盘和 CD-ROM
USB_DEVICE_CLASS_HUB	9	USB hubs
USB_DEVICE_CLASS_VENDOR_SPECIFIC	255	厂商定义的设备类

设备版本号

Microsoft 强烈建议厂商在硬件或固件的修订版中增加设备版本号以便于下层软件更新。一般，厂商发行新版本硬件的同时也带来驱动程序的修订版。同样，硬件升级应该使以前用于掩盖硬件错误的软件补丁或过滤器驱动程序无效。系统的自动升级机制在遇到一个版本不明确的硬件时会失败。

iManufacturer、**iProduct**、和 **iSerialNumber** 域指向一个串描述符，该串描述符用人类可读的语言描述设备生产厂商、产品、和序列号。这些串是可选的，0 值代表没有描述串。如果你在设备上放入了序列号串，Microsoft 建议应使每个物理设备的序列号唯一。

最后，**bNumConfigurations** 指出该设备能实现多少种配置。Microsoft 的驱动程序仅工作于设备的第一种配置（1 号配置）。我将在后面解释怎样使用设备的多个配置。

配置描述符

每个设备有一个或多个配置描述符，它们描述了设备能实行的各种配置方式。DDK 中定义的配置描述符结构如下：

```
typedef struct _USB_CONFIGURATION_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    USHORT wTotalLength;
    UCHAR bNumInterfaces;
    UCHAR bConfigurationValue;
    UCHAR iConfiguration;
    UCHAR bmAttributes;
    UCHAR MaxPower;
} USB_CONFIGURATION_DESCRIPTOR, *PUSB_CONFIGURATION_DESCRIPTOR;
```

bLength 和 **bDescriptorType** 域应为 9 和 2，即是一个 9 字节长的配置描述符。**wTotalLength** 域为该配置描述符长度加上该配置内所有接口和端点描述符长度的总和。通常，主机在发出一个 **GET_DESCRIPTOR** 请求并正确接收到 9 字节长的配置描述符后，就会再发出一个 **GET_DESCRIPTOR** 请求并指定这个总长度。第二个请求把这个大联合描述符传输回来。（注意这个传输不会把不属于这个配置的接口和端点描述符传输回来）

bNumInterfaces 指出该配置有多少个接口。这个值仅是接口的数量，不包括接口中的替换设置。这个域的目的是允许多功能设备存在，如一个有定位器（类似于鼠标）的键盘。

bConfigurationValue 域是该配置的索引值。你可以用这个值在 **SET_CONFIGURATION** 控制请求中选择这个配置。注意设备的第一个配置描述符的索引为 1。(选择配置 0 将把设备置入未配置状态，此时仅有端点 0 是活动的)

iConfiguration 域是一个可选的串描述符索引，指向描述该配置的 Unicode 字符串。此值为 0 表明该配置没有串描述符。

bmAttributes 字节包含描述该配置中设备电源和其它特性的位掩码，见表 11-5。未提到的位为未来标准保留。一个支持远程唤醒的配置应有远程唤醒属性位。该字节最高两位与 **MaxPower** 域一起描述配置中的电源特性。基本上，设置了最高位的配置都同时在 **MaxPower** 域中指出要从 USB 总线上获取的最大电流量(单位为 2mA)。使用外接电源的配置需要设置自供电属性位。

表 11-5. 配置的属性位

位掩码	符号名称	描述
80h	USB_CONFIG_BUS_POWERED	废弃 -- 应总为 1
40h	USB_CONFIG_SELF_POWERED	该配置为自供电
20h	USB_CONFIG_REMOTE_WAKEUP	该配置有远程唤醒特征

接口描述符

每个配置有一个或多个接口描述符，它们描述了设备提供功能的接口。

DDK 中的接口描述符结构定义如下：

```
typedef struct _USB_INTERFACE_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    UCHAR bInterfaceNumber;
    UCHAR bAlternateSetting;
    UCHAR bNumEndpoints;
    UCHAR bInterfaceClass;
    UCHAR bInterfaceSubClass;
    UCHAR bInterfaceProtocol;
    UCHAR iInterface;
} USB_INTERFACE_DESCRIPTOR, *PUSB_INTERFACE_DESCRIPTOR;
```

bLength 和 **bDescriptorType** 域应为 9 和 4。**bInterfaceNumber** 和 **bAlternateSetting** 是索引值，用在 **SET_INTERFACE** 控制事务中以指定要激活的接口。这些值可以是任意的，但习惯上，配置中的接口号从 0 开始，每个接口中的替换设置也是从 0 开始的。

bNumEndpoints 域指出该接口有多少个端点，不包括端点 0，端点 0 被认为是总存在的，并且是接口的一部分。

bInterfaceClass、**bInterfaceSubClass**、和 **bInterfaceProtocol** 域描述了接口提供的功能。一个非 0 的类代码应该是上面讨论的类代码中的一个，同时子类和协议代码也必须有与该类相类似的含义。这些域不允许有 0 值，0 值为未来标准保留。

最后，**iInterface** 是一个串描述符的索引，0 表示该接口无描述串。

端点描述符

接口可以没有或有多个端点描述符，它们描述了处理事务的端点。

DDK 中定义的端点描述符结构如下：

```

typedef struct _USB_ENDPOINT_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    UCHAR bEndpointAddress;
    UCHAR bmAttributes;
    USHORT wMaxPacketSize;
    UCHAR bInterval;
} USB_ENDPOINT_DESCRIPTOR, *PUSB_ENDPOINT_DESCRIPTOR;

```

bLength 和 **bDescriptorType** 域应为 7 和 5。**bEndpointAddress** 域编码端点的方向性和端点号，如图 11-11 所示。例如，地址值 0x82 指出该端点是一个端点号为 2 的 IN 端点，而 0x02 地址指出一个端点号为 2 的 OUT 端点。除了端点 0，两个端点可以有相同的端点号但方向相反。

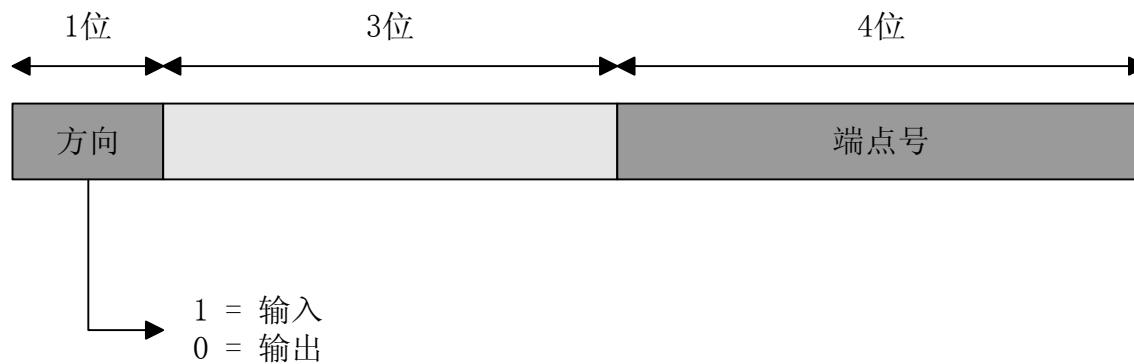


图 11-11. 端点描述符地址域的位排列

bmAttributes 的低两位指出端点的类型。见表 11-6。其余的位保留给将来使用，应设为 0。

表 11-6. 端点类型代码

符号名称	值	端点类型
USB_ENDPOINT_TYPE_CONTROL	0	控制端点
USB_ENDPOINT_TYPE_ISOCRONOUS	1	等时端点
USB_ENDPOINT_TYPE_BULK	2	批量端点
USB_ENDPOINT_TYPE_INTERRUPT	3	中断端点

wMaxPacketSize 值指出该端点在一个事务中能传输的最大数据量，表 11-1 列出了每种端点的可能值。

中断端点和等时端点描述符还有一个用于指定循检间隔时间的 **bInterval** 域，时间单位为毫秒。这个数指出主机以多长的周期循检这些端点，以查看是否有可能的数据传输。对于中断端点，该值的范围为 1 到 255 毫秒，代表两次循检间的最大时间间隔。对于等时端点，该值应该为 1，因为帧周期固定为 1 毫秒，每帧都应该循检。

串描述符

设备、配置、端点描述符都可以包含一个指向人类可读串的指针。串本身以 USB 串描述符的形式保存在设备中，串字符使用 **Unicode** 编码。

DDK 中的串描述符结构声明如下：

```

typedef struct _USB_STRING_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    WCHAR bString[1];
} USB_STRING_DESCRIPTOR, *PUSB_STRING_DESCRIPTOR;

```

bLength 值根据串数据长度可变。 **bDescriptorType** 域的值应为 3。 **bString** 域包含串数据本身。注意，串的空结尾符应包含在描述符长度内。

USB 设备可以以多种语言支持串描述符。0 号串描述符是设备所支持语言的标识符数组，它不是一个真正的串描述符。(用在其它描述符中代表无描述串指定的值 0，在这里被用于索引串语言标识数组)语言标识与 Win32 程序中使用的 **LANGID** 相同。例如，**0x409** 是美国英语的代码。如果向设备询问串描述符的某种未支持语言的表达，其结果在 USB 规范中没有规定，所以你应该先读取串 0 数组。关于语言标识请参考 USB 规范第 9.6.5 段。

其它描述符

USB 是一个发展的规范，我能表达的仅仅是它的概要。例如，一个 USB 工作组最近完成了一个接口级的电源管理规范。你可以在 USB 网站上读到这些内容，并且 DDK 头文件 **USB100.H** 中包含了这个规范的定义。时间不允许我探索这些新功能。幸好，对于 WDM 驱动程序的作者并不需要了解这些，解释接口特征描述符是 hub 驱动程序的任务而不是 WDM 功能驱动程序或过滤器驱动程序的任务。

使用总线驱动程序

与传统 PC 总线(如 PCI 总线)设备的驱动程序相比，USB 设备驱动程序从不直接与硬件对话。相反，它仅靠创建 URB(USB 请求块)并把 URB 提交到总线驱动程序就可完成硬件操作。

可以把 USBD.SYS 看作是接受 URB 的实体，向 USBD 的调用被转化为带有主功能代码为 **IRP_MJ_INTERNAL_DEVICE_CONTROL** 的 IRP。然后 USBD 再调度总线时间，发出 URB 中指定的操作。

在这一节，我将讲述功能驱动程序使用 USBD 的方法。首先讲述怎样建立并提交一个 URB，然后再讨论设备配置和重配置的方法。最后，我将概述如何管理四种通信管道。

初始化请求

为了创建一个 URB，你首先应该为 URB 分配内存，然后调用初始化例程把 URB 结构中的各个域填入请求要求的内容，例如，当你为响应 **IRP_START_DEVICE** 请求而配置设备时，首要的任务就是读取该设备的设备描述符。下面代码片段可以完成这个任务：

```
USB_DEVICE_DESCRIPTOR dd;
URB urb;
UsbBuildGetDescriptorRequest(&urb,
    sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
    USB_DEVICE_DESCRIPTOR_TYPE,
    0,
    0,
    &dd,
    NULL,
    sizeof(dd),
    NULL);
```

我们首先声明一个局部变量 **urb** 来保存 URB 数据结构。**URB** 在 **USBDI.H** 中声明，是一个多子结构的联合，我们将使用 **UrbControlDescriptorRequest** 子结构，它的类型是 **_URB_CONTROL_DESCRIPTOR_REQUEST**。

使用自动变量当然是可以的，但你事先必须了解系统堆栈上是否有足够的空间装下最大的 URB，并且在这个 URB 被完成前，你不能离开当前函数，否则自动变量将被释放。

当然，你还可以在系统堆上为 URB 动态地分配内存：

```
PURB urb = (PURB) ExAllocatePool(NonPagedPool, sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST));
if (!urb)
    return STATUS_INSUFFICIENT_RESOURCES;
UsbBuildGetDescriptorRequest(urb, ...);
...
ExFreePool(urb);
```

UsbBuildGetDescriptorRequest 看上去象一个正常的服务例程，实际上它是一个宏(在 **USBDLIB.H** 中声明)，用于生成“读描述符请求子结构”各个域的初始化语句。在 **DDK** 头文件中定义了这些创建各种 URB 的宏，见表 11-7。因为它们是预处理宏，所以在它们的参数中应避免使用有侧效的表达式。

表 11-7. 用于创建 URB 的辅助宏

辅助宏	事务类型
UsbBuildInterruptOrBulkTransferRequest	对中断或批量端点的输入和输出
UsbBuildGetDescriptorRequest	端点 0 的 GET_DESCRIPTOR 控制请求
UsbBuildGetStatusRequest	对设备、接口、端点的 GET_STATUS 请求
UsbBuildFeatureRequest	对设备、接口、端点的 SET_FEATURE 或 CLEAR_FEATURE 请求
UsbBuildSelectConfigurationRequest	SET_CONFIGURATION
UsbBuildSelectInterfaceRequest	SET_INTERFACE
UsbBuildVendorRequest	任何厂商定义的控制请求

在前面的代码片段中，我们指定把设备描述符信息接收到一个局部变量(**dd**)中，其地址和长度我们在参数中都提供了。涉及到数据传输的 URB 可以指定两种方式的非分页数据缓冲区。你可以象上面代码那样指定缓冲区的虚拟地址和长度。另一种方法是提供一个内存描述符列表(**MDL**)，但事先必须调用 **MmProbeAndLockPages** 函数对这个 **MDL** 进行探测并锁定(**probe-and-lock**)。

关于 URB 的更多内容

在内部，总线驱动程序总是用 **MDL** 描述数据缓冲区。如果你指定一个缓冲区地址，**USBD** 就自己创建一个 **MDL**。如果你也用 **MDL** 指定缓冲区，**USBD** 会调用 **MmGetSystemAddressForMdl** 函数取得缓冲区的原虚拟地址，然后用这个虚拟地址再创建一个描述相同缓冲区的 **MDL**！

URB 还有一个连接域 **Urblink**，它是 **USBD** 内部用于向主控制器驱动程序一次提交多个 URB 所使用的域。各种用于初始化 URB 的宏函数都有一个参数对应这个连接域，理论上你可以为其指定一个值，但我们总是用 **NULL** 来填充，原因是“接连 URB”这个概念还没有全部实现。实际上，在此处填值可能导致系统崩溃。

发送URB

创建完 URB 后，你需要创建并发送一个内部 I/O 控制(**IOCTL**)请求到 **USBD** 驱动程序，**USBD** 驱动程序位于驱动程序层次结构的低端。在大多数情况下，你需要等待设备回应，可以使用下面辅助函数：

```
NTSTATUS SendAwaitUrb(PDEVICE_OBJECT fdo, PURB urb)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    KEVENT event;
    KeInitializeEvent(&event, NotificationEvent, FALSE);
```

```

IO_STATUS_BLOCK iostatus;
PIRP Irp = IoBuildDeviceIoControlRequest(IOCTL_INTERNAL_USB_SUBMIT_URB,           <--2
                                         pdx->LowerDeviceObject,
                                         NULL,
                                         0,
                                         NULL,
                                         0,
                                         TRUE,
                                         &event,
                                         &iostatus);

PIO_STACK_LOCATION stack = IoGetNextIrpStackLocation(Irp);                         <--3
stack->Parameters.Others.Argument1 = (PVOID) urb;
NTSTATUS status = IoCallDriver(pdx->LowerDeviceObject, Irp);                      <--4
if (status == STATUS_PENDING)
{
    KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);
    status = iostatus.Status;
}
return status;
}

```

- 我们将等待 URB 完成，所以我们必须先创建一个内核事件对象。这与我在第六章"即插即用"中使用的辅助函数 **ForwardAndWait** 类似。
- 创建内部 IOCTL 请求最简单的方法是调用 **IoBuildDeviceIoControlRequest** 函数，第一个参数 (**IOCTL_INTERNAL_USB_SUBMIT_URB**) 指出 I/O 控制代码。第二个参数 (**pdx->LowerDeviceObject**) 指定接收请求的设备对象；**IoBuildDeviceIoControlRequest** 使用这个指针来决定需要接收多少个堆栈单元。接下来的四个参数描述输入输出缓冲区，提交这种 URB 并不需要这些信息，所以例子中将它们置成 NULL 或 0 值。第七个参数为 TRUE，它指出我们创建的是 IRP_MJ_INTERNAL_DEVICE_CONTROL 请求而不是 IRP_MJ_DEVICE_CONTROL 请求。最后两个参数指出等待 URB 完成的事件和一个接收该操作最终状态的结构 **IO_STATUS_BLOCK**。
- 被提交的 URB 的地址被填入 **Parameters.Others** 的 **Argument1** 域。对于普通的 IOCTL 请求，该偏移对应 **OutputBufferLength** 域。
- 我们用 **IoCallDriver** 把请求发送到下一层驱动程序。USBD 将处理该 URB 请求并完成，然后 I/O 管理器将那个 IRP 删除并置事件信号。由于我们没有提供自己的完成例程，所以不能确定 I/O 管理器在所有可能的完成情况下都置事件信号。我们仅当低级派遣例程返回 **STATUS_PENDING** 时才等待那个事件。

注意

需要强调的是驱动程序把 URB 包装成一个带有 IRP_MJ_INTERNAL_DEVICE_CONTROL 主功能码的普通 IRP。为了使上层过滤器驱动程序可以发送自己的 URB，每个 USB 设备驱动程序应有一个可以把 IRP 传递到下一层的派遣函数。

URB 返回的状态

当你提交一个 URB 到 USB 总线驱动程序时，你最终将收到一个描述该操作结果的 NTSTATUS 代码。其间，总线驱动程序使用另一组类型名为 **USBD_STATUS** 的状态代码。这些代码并不是 NTSTATUS 代码。

当 USBD 完成一个 URB 时，它就把 URB 的 **UrbHeader.Status** 域设置为某个 **USBD_STATUS** 值。DDK 中的 **URB_STATUS** 宏可以简化这个值的存取：

```

NTSTATUS status = SendAwaitUrb(fdo, &urb);
USBD_STATUS ustatus = URB_STATUS(&urb);
...

```

并没有特别的协议关于保留这个值以及把它传递回应用程序，你可以随意处理这个值。

配置

USB 总线驱动程序自动检测新插入的 USB 设备。然后它读取设备内的设备描述符以查明插入的是何种设备，描述符中的厂商和产品标识以及其它描述符一同决定具体安装哪一个驱动程序。

配置管理器调用驱动程序的 **AddDevice** 函数。**AddDevice** 做所有你已知的任务：创建设备对象，把设备对象连接到驱动程序堆栈上，等等。最后，配置管理器向驱动程序发送一个即插即用请求 **IRP_MN_START_DEVICE**。在第六章中，我已经讲述了怎样处理这个请求，通过调用一个名为 **StartDevice** 的辅助函数并传递一些参数，这些参数描述了赋予设备的经过转换的和未经转换的 I/O 资源。有一个好消息，你不必再为 USB 驱动程序的 I/O 资源担心了，因为它们不用任何 I/O 资源。所以你可以按下面框架写一个 **StartDevice** 辅助函数：

```
NTSTATUS StartDevice(PDEVICE_OBJECT fdo)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    <configure device>
    return STATUS_SUCCESS;
}
```

这里我可以直接说配置设备而不是配置硬件，因为你不用再涉及 I/O 端口、中断、DMA 适配器对象，或者任何其它面向资源的元素(我在第七章中描述过)。

驱动程序在哪里？

我将在第十二章中详细讨论 WDM 驱动程序的安装，为了配合本章内容我先在这里简单介绍一下。让我们假设设备的厂商标识和产品标识分别为 0x547 和 0x102A。USB 使用了许多方法来帮助操作系统定位驱动程序(或驱动程序集)，包括设备上的设备描述符、配置描述符，以及接口描述符。对于有厂商和产品标识的设备，配置管理器首先在注册表中查找名为 **USB\VID_0547&PID_102A** 的设备。如果注册表中没有这个表项，配置管理器将触发“新硬件向导”来寻找该设备的 INF 文件。新硬件向导向用户询问 INF 文件的位置，然后安装驱动程序并填写注册表。一旦配置管理器找到了注册表表项，它就可以动态地装载驱动程序。

StartDevice 的执行过程大致如下，首先为设备选择一个配置。如果你的设备象大多数设备一样，应该仅有一种配置。选定了某个配置后，接着应该选择配置中的一个或多个接口。顺便说一下，支持多接口的设备并不少见。选定了一个配置和一组接口后，你应该向总线驱动程序发送配置选择 URB。最后，总线驱动程序向设备发出命令使能选定的配置和接口。总线驱动程序负责创建管道和用于访问管道的句柄，管道提供功能驱动程序与选定接口端点之间的通信，它同时还创建配置句柄和接口句柄。你可以从完成的 URB 中提取这些句柄并保存为以后使用。至此，设备的配置过程全部结束。

多功能设备

如果你的设备有一个配置和多个接口，Microsoft 的总线驱动程序将自动把这种设备当作一个组合的，或多功设备。你必须为每个接口提供一个功能驱动程序，通过在 INF 中用接口类和子类替换厂商标识和产品标识。总线驱动程序为每个接口创建一个物理设备对象(PDO)，这样，PnP 管理器就可以装载独立的驱动程序。当任何一个功能驱动程序读取配置描述符时，总线驱动程序就提供一个修改后的仅描述一个接口的描述符。关于 INF 文件中的设备标识的各种可能形式请参考第十二章。

读取配置描述符

可以把固定大小的配置描述符看做是某个可变长结构的头，这个可变长结构描述了一个配置和该配置的所有接口、以及每个接口的所有端点。见图 11-12。

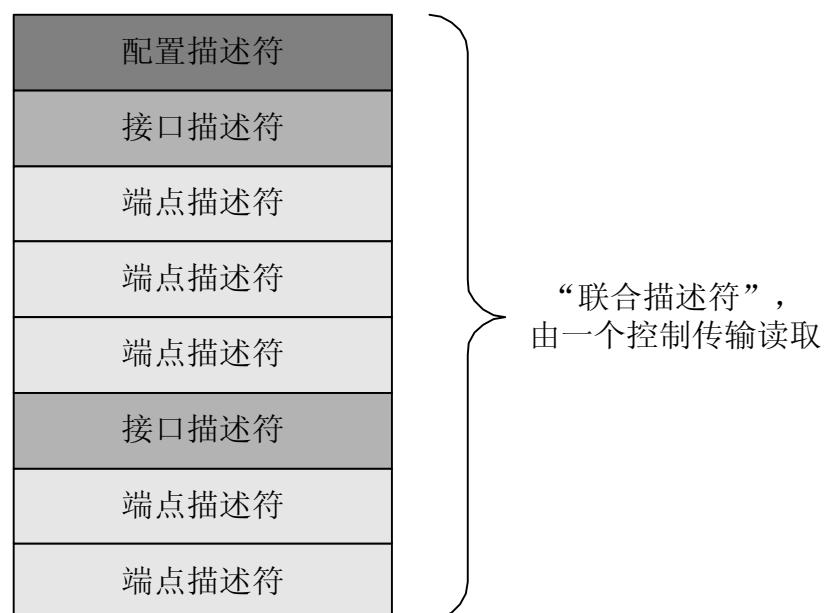


图 11-12. 配置描述符的结构

因为硬件不允许直接访问接口和端点描述符，所以必须把整个可变长结构读入一个连续的内存区。不幸的是，在读之前我们并不能确切地知道这个结构有多长。下面代码演示了如何用两个 URB 来读取配置描述符：

```

ULONG iconfig = 1;
URB urb;
USB_CONFIGURATION_DESCRIPTOR tcd;
UsbBuildDescriptorRequest(&urb,
    sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
    USB_CONFIGURATION_DESCRIPTOR_TYPE,
    iconfig,
    0,
    &tcd,
    NULL,
    sizeof(tcd),
    NULL);

SendAwaitUrb(fdo, &urb);
ULONG size = tcd.wTotalLength;
PUSB_CONFIGURATION_DESCRIPTOR pcd = (PUSB_CONFIGURATION_DESCRIPTOR)
ExAllocatePool(NonPagedPool, size);
UsbBuildDescriptorRequest(&urb,
    sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
    USB_CONFIGURATION_DESCRIPTOR_TYPE,
    iconfig,
    0,
    pcd,
    NULL,
    size,
    NULL);

SendAwaitUrb(fdo, &urb);
...
ExFreePool(pcd);

```

在这段代码中，我们先发出一个 URB，把配置描述符读取到临时描述符缓冲区 **tcd**，我指定配置号 1，它是第一个配置描述符。这个描述符包含了可变长结构的总长度(**wTotalLength**)。然后我们分配该长度的内存块，并发出了第二个 URB 读取整个描述符。最后，**pcd** 变量指向包含整个配置信息的内存区。(注意，真正的应用代码应该有错误检测)

如果设备仅有一个配置，我们可以直接前进到下一步，并使用刚读出的描述符集。否则，我们需要枚举所有配置(即，使变量 **iconfig** 从 1 循环到设备描述符中 **bNumConfigurations** 指定的值)并使用某种筛选算法来提取它们。

读配置描述符时使用的描述符索引来自于设备固件响应 **GET_DESCRIPTOR** 控制请求的结果，而不受 USB 规范的限定。尽管配置号从 1 开始，但配置描述符应从 0 开始数。

选择配置

通过向设备发送一系列控制命令来选择配置及使能需要的接口。我们将使用 **USBD_CreateConfigurationRequestEx** 函数创建这种 URB。其中的一个参数指向要使能的接口描述符数组。下一步就是准备这个数组。

读串描述符

设备内部可以保存一些字符串。在 **USB42** 的例子中，设备保存了厂商、产品等的英语描述符。我写了下面读串描述符的辅助函数。

```
NTSTATUS GetStringDescriptor(PDEVICE_OBJECT fdo, UCHAR istring, PUNICODE_STRING s)
{
    NTSTATUS status;
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    URB urb;

    UCHAR data[256];

    if (!pdx->langid)
    {
        UsbBuildGetDescriptorRequest(&urb,
                                     sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
                                     USB_STRING_DESCRIPTOR_TYPE,
                                     0,
                                     0,
                                     data,
                                     NULL,
                                     sizeof(data),
                                     NULL);

        status = SendAwaitUrb(fdo, &urb);
        if (!NT_SUCCESS(status))
            return status;
        pdx->langid = *(LANGID*)(data + 2);
    }

    UsbBuildGetDescriptorRequest(&urb,
                                sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
                                USB_STRING_DESCRIPTOR_TYPE,
                                istring,
                                pdx->langid,
                                data,
                                NULL,
                                sizeof(data),
                                NULL);

    status = SendAwaitUrb(fdo, &urb);
    if (!NT_SUCCESS(status))
        return status;
}
```

```

ULONG nchars = (data[0] - 2) / 2;
PWSTR p = (PWSTR) ExAllocatePool(PagedPool, data[0]);
if (!p)
    return STATUS_INSUFFICIENT_RESOURCES;
memcpy(p, data + 2, nchars*2);
p[nchars] = 0;
s->Length = (USHORT) (2 * nchars);
s->MaximumLength = (USHORT) ((2 * nchars) + 2);
s->Buffer = p;

return STATUS_SUCCESS;
}

```

这个函数的特殊之处在于串提取 URB 的初始化。我们除了要提供串索引之外，还要提供一个标准的 LANGID 语言标识符。这个语言标识符与在 Win32 应用程序中使用的语言标识符相类似。正如我以前提到的，设备中的串可以以多种自然语言来表达，而串描述符 0 则包含着该设备所支持的语言标识符列表。为了确保请求一个被设备支持的语言，我首先读取了串 0，并选择第一个被支持的语言。在我的例子驱动程序中，语言标识符总是 0x409，即美国英语。然后，USBD.SYS 把这个语言标识符和串索引作为取描述符请求操作的参数发往设备。最后，由设备本身决定究竟返回哪个串。

GetStringDescriptor 函数的输出是一个 UNICODE_STRING 结构。最终，你可以调用 **RtlFreeUnicodeString** 函数释放这个串。

我在 USB42 例子中使用 **GetStringDescriptor** 函数生成关于设备的额外调试信息。例如，**StartDevice** 中有这样的代码：

```

UNICODE_STRING sd;
if (pcd->iConfiguration && NT_SUCCESS(GetStringDescriptor(fdo, pcd->iConfiguration, &sd)))
{
    KdPrint(("USB42 - Selecting configuration named %ws\n", sd.Buffer));
    RtlFreeUnicodeString(&sd);
}

```

实际上我使用了一个宏(**KdPrint**)，这样我就不用重复输入相同的代码。

回想一下，当我们读配置描述符时，我们同时也把该配置的所有接口描述符读入了调整后的内存区中。这个内存区域包含了一系列描述符：一个配置描述符、一个接口描述符及其所有端点，下一个接口描述符和其所有端点，等等。有一种方法可以从这些描述符中提取你感兴趣的接口，总线驱动程序提供的

USBD_ParseConfigurationDescriptorEx 函数可以简化这个过程：

```

PUSB_INTERFACE_DESCRIPTOR pid;
pid = USBD_ParseConfigurationDescriptorEx(pcd,
                                            StartPosition,
                                            InterfaceNumber,
                                            AlternateSetting,
                                            InterfaceClass,
                                            InterfaceSubclass,
                                            InterfaceProtocol);

```

在这个函数中，**pcd** 是那个复合配置描述符的地址。**StartPosition** 可以是配置描述符的地址，或者是起始搜索的描述符地址。其余参数指定描述符搜索准则。值 -1 表示在搜索中不使用该准则。下面这些准则可以在搜索中指定：

- **InterfaceNumber**
- **AlternateSetting**

- **InterfaceClass**
- **InterfaceSubclass**
- **InterfaceProtocol**

当 **USBD_ParseConfigurationDescriptorEx** 返回一个接口描述符时，你应该把它保存在 **USBD_INTERFACE_LIST_ENTRY** 数组元素的 **InterfaceDescriptor** 成员中。然后，你跨过这个接口描述符前进到下一个接口。这个接口表项数组最后将成为调用 **USBD_CreateConfigurationRequestEx** 函数的一个参数，数组元素由下面结构定义：

```
typedef struct _USBD_INTERFACE_LIST_ENTRY {
    PUSB_INTERFACE_DESCRIPTOR InterfaceDescriptor;
    PUSBD_INTERFACE_INFORMATION Interface;
} USBD_INTERFACE_LIST_ENTRY, *PUSBD_INTERFACE_LIST_ENTRY;
```

当初始化数组中的表项时，你应该把要使能接口的接口描述符地址赋予 **InterfaceDescriptor** 成员，并把 **Interface** 成员置 **NULL**。你应该为每个接口定义一个表项，并在最后加入一个 **InterfaceDescriptor** 为 **NULL** 的结束表项。在我的 **USB42** 例子中，我事先知道仅存在一个接口，所以使用下面代码创建接口列表：

```
PUSB_INTERFACE_DESCRIPTOR pid = USBD_ParseConfigurationDescriptorEx(pcd,
    pcd,
    -1,
    -1,
    -1,
    -1,
    -1,
    -1);
USBD_INTERFACE_LIST_ENTRY interfaces[2] = {
    {pid, NULL},
    {NULL, NULL},
};
```

即，析取配置描述符并定位在第一个遇到的接口描述符上。然后我定义了仅有两个元素的数组来描述这个接口。

如果你要支持多功能设备，则需要使能多个接口，在循环中重复析取调用。如下例：

```
ULONG size = (pcd->bNumInterfaces + 1) * sizeof(USBD_INTERFACE_LIST_ENTRY);           <--1
PUSBD_INTERFACE_LIST_ENTRY interfaces = (PUSBD_INTERFACE_LIST_ENTRY)
ExAllocatePool(NonPagedPool, size);
RtlZeroMemory(interfaces, size);
ULONG i = 0;
PUSB_INTERFACE_DESCRIPTOR pid = (PUSB_INTERFACE_DESCRIPTOR) pcd;
while ((pid = USBD_ParseConfigurationDescriptorEx(pcd, pid, ...)))           <--2
    interfaces[i++].InterfaceDescriptor = pid++;                                <--3
```

1. 首先为接口列表申请内存并清空该内存区。
2. 析取接口。在第一次循环中，**pid** 指向配置描述符。在下一次循环中，它指向前一个调用返回的接口描述符。
3. 初始化指向接口描述符的指针。如果调用 **USBD_ParseConfigurationDescriptorEx** 时遇到一个符合指定准则的接口描述符，将返回指向这个描述符的指针。

配置过程的下一步是创建 URB 并发送到设备：

```
PURB selurb = USBD_CreateConfigurationRequestEx(pcd, interfaces);
```

除了创建 URB，**USBD_CreateConfigurationRequestEx** 还初始化 **USBD_INTERFACE_LIST** 表项中的 **Interface** 成员，使其指向一个 **USBD_INTERFACE_INFORMATION** 结构。这个结构与 URB 在同一物理内存区，在调用 **ExFreePool** 释放 URB 时该结构一同被释放。该结构描述如下：

```

typedef struct _USBD_INTERFACE_INFORMATION {
    USHORT Length;
    UCHAR InterfaceNumber;
    UCHAR AlternateSetting;
    UCHAR Class;
    UCHAR SubClass;
    UCHAR Protocol;
    UCHAR Reserved;
    USBD_INTERFACE_HANDLE InterfaceHandle;
    ULONG NumberOfPipes;
    USBD_PIPE_INFORMATION Pipes[1];
} USBD_INTERFACE_INFORMATION, *PUSBD_INTERFACE_INFORMATION;

```

其中管道信息结构是我们在此真正关心的，因为结构中的其它域是在提交 URB 后由 USBD 填充的。管道结构描述如下：

```

typedef struct _USBD_PIPE_INFORMATION {
    USHORT MaximumPacketSize;
    UCHAR EndpointAddress;
    UCHAR Interval;
    USBD_PIPE_TYPE PipeType;
    USBD_PIPE_HANDLE PipeHandle;
    ULONG MaximumTransferSize;
    ULONG PipeFlags;
} USBD_PIPE_INFORMATION, *PUSBD_PIPE_INFORMATION;

```

这样，我们已经有了一组 USBD_INTERFACE_LIST 表项，每一项中的 USBD_INTERFACE_INFORMATION 结构又包含一组 USBD_PIPE_INFORMATION 结构。接下来的任务是填充管道信息结构中的 **MaximumTransferSize** 成员，如果不填充这个成员，USBD 将使用默认值 USBD_DEFAULT_MAXIMUM_TRANSFER_SIZE，其值等于 DDK 中的 PAGE_SIZE。这个成员值与端点在单事务中的最大传输量(在一个总线事务中可以传输多少字节)或端点接收量(由设备上有多少有效内存决定)不直接相关。相反，它代表单一 URB 中能携带的最大数据量。这个值可以小于应用程序向设备或设备向应用程序发送的数据量的最大值，在这种情况下，驱动程序必须把应用程序的请求分成不大于这个值的小块。我们将在"管理批量传输管道"中再讨论这个问题。

提供最大传输量的原因是由于一个调度算法，主控制器驱动程序使用这个算法把 URB 请求分成总线帧中的事务。如果要发送大量数据，我们的数据可能占满整个帧而其它设备的数据会被挤出去。所以，通过为 URB 指定一个适当的单次传输最大值来均衡总线带宽的使用。

初始化管道信息结构可以参考下面代码：

```

for (ULONG ii = 0; ii < <number of interfaces>; ++ii)
{
    PUSBD_INTERFACE_INFORMATION pii = interfaces[ii].Interface;
    for (ULONG ip = 0; ip < pii->NumberOfPipes; ++ip)
        pii->Pipes[ip].MaximumTransferSize = <some constant>;
}

```

注意

USBD_CreateConfigurationRequestEx 函数用 USBD_DEFAULT_MAXIMUM_TRANSFER_SIZE 和 0 来初始化每个管道信息中的 MaximumTransferSize 成员和 PipeFlags 成员。

一旦你完成管道信息结构的初始化，就可以提交这个配置 URB：

```
SendAwaitUrb(fdo, selurb);
```

寻找句柄

配置选择 URB 成功完成后，我们应该把一些句柄保存下来供以后使用：

- URB 成员 **UrbSelectConfiguration.ConfigurationHandle** 返回该配置句柄。
- **USBD_INTERFACE_INFORMATION** 结构中的 **InterfaceHandle** 返回接口句柄。
- 每个 **USBD_PIPE_INFORMATION** 结构中都含有与每个端点对应的管道句柄 **PipeHandle**。

例如，**USB42** 例子在设备扩展中保存了两个句柄：

```
typedef struct _DEVICE_EXTENSION {
    ...
    USBD_CONFIGURATION_HANDLE hconfig;
    USBD_PIPE_HANDLE hpipe;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

pdx->hconfig = selurb->UrbSelectConfiguration.ConfigurationHandle;
pdx->hpipe = interfaces[0].Interface->Pipes[0].PipeHandle;
ExFreePool(selurb);
```

至此，那个配置选择 URB 就不再需要了，应该删除。

关闭设备

当驱动程序接到一个 **IRP_MN_STOP_DEVICE** 请求时，应该把设备置成未配置状态(配置选择号 0)，创建并提交一个含 **NULL** 配置指针的配置选择 URB 即可以达到这个目的：

```
URB urb;
UsbBuildSelectConfigurationRequest(&urb,
    sizeof(_URB_SELECT_CONFIGURATION),
    NULL);
SendAwaitUrb(fdo, &urb);
```

管理批量传输管道

随书光盘中的两个例子程序演示了批量传输。第一个例子名为 **USB42**。设备上有一个批量输入端点，该端点在每次读它时都返回一个常量值 42。读数据代码如下：

```
URB urb;
UsbBuildInterruptOrBulkTransferRequest(&urb,
    sizeof(_URB_BULK_OR_INTERRUPT_TRANSFER),
    pdx->hpipe,
    Irp->AssociatedIrp.SystemBuffer,
    NULL,
    cbout,
    USBD_TRANSFER_DIRECTION_IN |
    USBD_SHORT_TRANSFER_OK,
    NULL);
status = SendAwaitUrb(fdo, &urb);
```

这些代码运行在 **DeviceIoControl** 调用的处理程序中，所以 IRP 的 **SystemBuffer** 域指向应提交的数据。**cbout** 变量是我们要填充的缓冲区的大小。

置 **USBD_TRANSFER_DIRECTION_IN** 标志或不置该标志分别表示对端点的读或写。你还可以选择性地指出另一个标志位 **USBD_SHORT_TRANSFER_OK**，它表明提交的数据量少于端点的最大传输数据量。

LOOPBACK 例子要比 **USB42** 复杂些。该例子中有两个批量端点，一个是输入端点另一个是输出端点。我们首先向输出端点送出 16384 字节的数据，然后再从输入端点读回这些数据。驱动程序本身使用标准的 **IRP_MJ_READ** 和 **IRP_MJ_WRITE** 请求来实现数据传输。读写请求的处理比较简单，派遣函数仅把这些请求发到名为 **ReadWrite** 的函数。

```
NTSTATUS DispatchRead(PDEVICE_OBJECT fdo, PIRP Irp)
{
    return ReadWrite(fdo, Irp, TRUE);
}

NTSTATUS DispatchWrite(PDEVICE_OBJECT fdo, PIRP Irp)
{
    return ReadWrite(fdo, Irp, FALSE);
}

NTSTATUS ReadWrite(PDEVICE_OBJECT fdo, PIRP Irp, BOOLEAN read)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
    if (!NT_SUCCESS(status))
        return CompleteRequest(Irp, status, 0);
    ...
    IoMarkIrpPending(Irp);
    IoSetCompletionRoutine(Irp, (PIO_COMPLETION_ROUTINE) OnReadWriteComplete, ...);
    IoCallDriver(...);
    return STATUS_PENDING;
}
```

简单地说，**ReadWrite** 先申请了删除锁(**remove lock**)，创建一个 **URB** 来做批量传输，安装了一个完成例程，最后向总线驱动程序发出 **URB**。

LOOPBACK 向总线驱动程序提交请求的总体策略是把读写 **IRP** 改变为包含有 **URB** 的 **IRP_MJ_INTERNAL_DEVICE_CONTROL** 请求，并把这个修改后的 **IRP** 送到下层驱动程序。对于我们和我们上面的驱动程序来说，该 **IRP** 看起来就是一个 **IRP_MJ_READ** 或 **IRP_MJ_WRITE**，但对于下面的驱动程序，该 **IRP** 是一个内部控制请求。那个完成例程(completion routine)将再次提交这个 **IRP** 以执行大数据传输的其余部分。除去真正 **LOOPBACK** 例子中的错误检测，下面就是 **ReadWrite** 程序和与其关联的完成例程：

```
struct _RWCONTEXT : public _URB
{
    ULONG_PTR va;
    ULONG length;
    PMDL mdl;
    ULONG numxfer;
};

NTSTATUS ReadWrite(PDEVICE_OBJECT fdo, PIRP Irp, BOOLEAN read)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp); <
    if (!NT_SUCCESS(status))
```

```

        return CompleteRequest(Irp, status, 0);
USBD_PIPE_HANDLE hpipe = read ? pdx->hinpipe : pdx->houtpipe;

LONG haderr;
if (read)
    haderr = InterlockedExchange(&pdx->inerror, 0);
else
    haderr = InterlockedExchange(&pdx->outerror, 0);
if (haderr && !NT_SUCCESS(ResetPipe(fdo, hpipe)))
    ResetDevice(fdo);

PRWCONTEXT ctx = (PRWCONTEXT) ExAllocatePool(NonPagedPool, sizeof(RWCONTEXT));
RtlZeroMemory(ctx, sizeof(RWCONTEXT));

ULONG length = Irp->MdlAddress ? MmGetMdlByteCount(Irp->MdlAddress) : 0;
if (!length)
{
    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    return CompleteRequest(Irp, STATUS_SUCCESS, 0);
}
ULONG_PTR va = (ULONG_PTR) MmGetMdlVirtualAddress(Irp->MdlAddress);

ULONG urbflags = (read ? USBD_TRANSFER_DIRECTION_IN :
USBD_TRANSFER_DIRECTION_OUT);

ULONG seglen = length;                                     <--6
if (seglen > MAXTRANSFER)
    seglen = (ULONG_PTR) PAGE_ALIGN(va) + PAGE_SIZE - va;

PMDL mdl = IoAllocateMdl((PVOID) va, PAGE_SIZE, FALSE, FALSE, NULL);
IoBuildPartialMdl(Irp->MdlAddress, mdl, (PVOID) va, seglen);

UsbBuildInterruptOrBulkTransferRequest(ctx,
                                         sizeof(_URB_BULK_OR_INTERRUPT_TRANSFER),
                                         hpipe,
                                         NULL,
                                         mdl,
                                         seglen,
                                         urbflags,
                                         NULL);

ctx->va = va + seglen;
ctx->length = length - seglen;
ctx->mdl = mdl;
ctx->numxfer = 0;

PIO_STACK_LOCATION stack = IoGetNextIrpStackLocation(Irp);
stack->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;
stack->Parameters.Others.Argument1 = (PVOID) (PURB) ctx;
stack->Parameters.DeviceIoControl.IoControlCode = IOCTL_INTERNAL_USB_SUBMIT_URB;

IoSetCompletionRoutine(Irp, (PIO_COMPLETION_ROUTINE) OnReadWriteComplete, (PVOID) ctx,
TRUE, TRUE, TRUE);

IoMarkIrpPending(Irp);
status = IoCallDriver(pdx->LowerDeviceObject, Irp);

```

```

        return STATUS_PENDING;
    }

NTSTATUS OnReadWriteComplete(PDEVICE_OBJECT fdo, PIRP Irp, PRWCONTEXT ctx)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    BOOLEAN read = (ctx->UrbBulkOrInterruptTransfer.TransferFlags &
    USBD_TRANSFER_DIRECTION_IN) != 0;
    ctx->numxfer += ctx->UrbBulkOrInterruptTransfer.TransferBufferLength;

    NTSTATUS status = Irp->IoStatus.Status;
    if (NT_SUCCESS(status) && ctx->length)                                <--9
    {
        ULONG seglen = ctx->length;
        if (seglen > MAXTRANSFER)
            seglen = (ULONG_PTR) PAGE_ALIGN(ctx->va) + PAGE_SIZE - ctx->va;

        IoBuildPartialMdl(Irp->MdlAddress, ctx->mdl, (PVOID) ctx->va, seglen);

        ctx->UrbBulkOrInterruptTransfer.TransferBufferLength = seglen;           <--11

        PIO_STACK_LOCATION stack = IoGetNextIrpStackLocation(Irp);
        stack->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;
        stack->Parameters.Others.Argument1 = (PVOID) (PURB) ctx;
        stack->Parameters.DeviceIoControl.IoControlCode = IOCTL_INTERNAL_USB_SUBMIT_URB;
        IoSetCompletionRoutine(Irp, (PIO_COMPLETION_ROUTINE) OnReadWriteComplete, (PVOID) ctx,
        TRUE, TRUE, TRUE);

        ctx->va += seglen;
        ctx->length -= seglen;

        IoCallDriver(pdx->LowerDeviceObject, Irp);                                <
        return STATUS_MORE_PROCESSING_REQUIRED;
    }

    if (NT_SUCCESS(status))
        Irp->IoStatus.Information = ctx->numxfer;
    else
    {
        if (read)
            InterlockedIncrement(&pdx->inerror);
        else
            InterlockedIncrement(&pdx->outerror);
    }

    ExFreePool(ctx->mdl);
    ExFreePool(ctx);
    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);

    return status;
}

```

1. ReadWrite 需要创建一个能与 **OnReadWriteComplete** 共享的 URB，另外它还需要一个上下文信息来跟踪操作的进展。RWCONTEXT 结构可以实现这两个目的。除了 URB，该结构还包含 **va**，用户模式缓冲区当前部分的虚拟地址；**length**，该操作中的余量计数；**mdl**，描述传输当前段的内存描述符表；**numxfer**，传输量的累加计数。
2. 在这里我们请求了删除锁，释放删除锁的 **IoReleaseRemoveLock** 调用在完成例程中执行。

3. 这里是 **ReadWrite** 需要区分读写请求的很少几处。在这里，我们获得了传输管道的句柄。
4. 不管是输入管道还是输出管道都应该有一个最近错误记录，如果错误发生，设备扩展中的 **inerror** 或 **outerror** 将被置位。在进行一个新操作前，我们需要把有错误的管道重置。如果它仍不能工作，我们就重置整个设备。我将在下一段解释 **ResetPipe** 和 **ResetDevice** 辅助函数。
5. 该驱动程序在 **AddDevice** 例程中声明使用 **DO_DIRECT_IO** 缓冲方法，所以 **IRP** 中的指针指向一个内存描述符列表，该表描述用户模式缓冲区的锁定内存页。习惯上我们从 **MDL** 中获得传输长度，而不是从堆栈单元中。
6. 我们以块方式执行操作，块的大小不能大于一页。处理页对齐的缓冲区可以提高操作性能，我把第一个传输用于传输不足一页的零头，这样，其余的传输将全部是页对齐的。
7. 我们为传输的每个阶段使用一个内存描述符表。分配完 **MDL** 后，调用 **IoBuildPartialMdl** 映射起始段。
8. 现在我们可以为读或写的第一阶段创建并提交 **URB** 了。这里的关键任务并不是读或写，而是初始化下层驱动程序的堆栈单元。做这些的主要益处是当主读写 **IRP** 被取消时，我们不必处理其它辅助 **IRP** 的取消。
9. 当第一次传输成功后，总线驱动程序调用 **IoCompleteRequest** 例程，然后我们的完成例程获得控制。如果还有数据没有传输，我们就用新缓冲区地址和长度再发一个 **URB**。如果传输全部完成，完成例程将运行下去直至结束。注意，我们正在处理的 **IRP** 是主功能码为 **IRP_MJ_READ** 或 **IRP_MJ_WRITE** 的原始 **IRP**。
10. 这里，我们为下一阶段的传输设置部分 **MDL**。因为该完成例程运行在任意线程上下文中，所以用户模式的虚拟地址没有用。然后调用 **IoBuildPartialMdl** 映射一个 **MDL** 的子集，这仅仅是从主 **MDL** 中复制物理页号，所以不必依靠任何特定的内存上下文。
11. 这里，我们为下一阶段的传输设置 **URB** 和 I/O 堆栈单元。**URB** 中需要改变的域仅有字节计数，**URB** 的 **MDL** 指针、标志，等等都不变。**(MDL** 自身会变，但其在内存中的位置不变) 我们需要完成下一个堆栈单元的初始化，因为 **IoCompleteRequest** 把其中大部分域设为 0。
12. 我们再次发出 **IRP** 到总线驱动程序并返回状态码 **STATUS_MORE_PROCESSING_REQUIRED**，这将使 **IoCompleteRequest** 中的完成处理停止，当新的阶段完成时，完成例程再次获得控制。
13. 在这里，读写请求将要全部完成，我们把 **IoStatus.Information** 域设置成总共传输的数据量，释放 **ReadWrite** 中分配的内存。释放删除自旋锁。

你可能注意到了，在这个例子中，完成例程并没有包含标准代码——有条件地调用 **IoMarkIrpPending**，因为我们已经在 **ReadWrite** 中调用过，所以没有必要在完成例程中再次调用。

你可能也注意到了，当完成例程调用 **IoCallDriver** 发送 **URB** 后，却无条件地返回了 **STATUS_MORE_PROCESSING_REQUIRED** 状态。这里有一个重要的原因，如果总线驱动程序正常地接受了新的 **URB**，它将向我们返回 **STATUS_PENDING**。**(USBD** 就是这样工作的，但这并不代表所有总线驱动程序都有这个特征) 在这种情况下，我们确实需要返回 **STATUS_MORE_PROCESSING_REQUIRED**，因为此时我们希望 **IoCompleteRequest** 停止处理 **IRP**，总线驱动程序将再次完成它。但是，如果总线驱动程序在新传输中失败，或者由于某种原因在派遣函数中就完成了该 **IRP**，而派遣函数在返回(执行 **return** 语句)前要调用 **IoCompleteRequest**，这样我们就进入了递归调用！所以，我们不应该对这个 **IRP** 做任何事，也不应该允许对 **IoCompleteRequest** 调用做任何初始化工作。在这里立即返回 **STATUS_MORE_PROCESSING_REQUIRED** 总是正确的。

错误恢复

当你用批量端点传输数据时，总线或总线驱动程序自动重试失败的传输。结果，如果 **URB** 表明成功完成，那么你可以确信数据已被正确传输。如果发生错误，你的驱动程序需要做某种恢复操作。第一步就是使处于停止状态的端点脱离停止状态，这样才可以再通信。下面是一个名为 **ResetPipe** 的辅助函数：

```
NTSTATUS ResetPipe(PDEVICE_OBJECT fdo, USBD_PIPE_HANDLE hpipe)
{
    URB urb;
    urb.UrbHeader.Length = (USHORT) sizeof(_URB_PIPE_REQUEST);
    urb.UrbHeader.Function = URB_FUNCTION_RESET_PIPE;
    urb.UrbPipeRequest.PipeHandle = hpipe;

    NTSTATUS status = SendAwaitUrb(fdo, &urb);
    return status;
}
```

如你所见，上面就是提交管道重置 URB 的全部过程。由于这个辅助例程间接地等待 URB 被完成，所以你必须在 PASSIVE_LEVEL 级上调用它。这个 URB 所做的工作用 USB 术语来说，就是清除该管道端点的 ENDPOINT_HALT 特征。

如果重置管道失败，则需要重置整个设备，使用 **ResetDevice** 函数：

```
VOID ResetDevice(PDEVICE_OBJECT fdo)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;

    KEVENT event;
    KeInitializeEvent(&event, NotificationEvent, FALSE);
    IO_STATUS_BLOCK iostatus;

    PIRP Irp = IoBuildDeviceIoControlRequest(IOCTL_INTERNAL_USB_RESET_PORT,
                                              pdx->LowerDeviceObject,
                                              NULL,
                                              0,
                                              NULL,
                                              0,
                                              TRUE,
                                              &event,
                                              &iostatus);

    if (!Irp)
        return;

    NTSTATUS status = IoCallDriver(pdx->LowerDeviceObject, Irp);
    if (status == STATUS_PENDING)
        KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);
}
```

hub 口重置命令可以使 hub 驱动程序在重新初始化设备的同时保留设备当前配置。如果该命令失败，hub 驱动程序将返回 STATUS_UNSUCCESSFUL。

管理中断管道

从设备方面看，中断管道与批量管道几乎完全相同。唯一的不同是，主机将以某个有保证的频率循检中断端点，除非中断端点有中断产生，否则设备将以 NAK 响应主机循检。报告中断事件时，设备以 ACK 应答并提供少量数据。

从驱动程序这边看，管理中断管道仅比管理批量管道稍复杂一点。当驱动程序读写批量管道时，它仅仅是创建 URB 并发送到总线驱动程序。但中断管道是用于向主机通知硬件事件，驱动程序需要始终维持一个未完成的读请求。我不推荐使用系统循检线程，因为电源管理问题将使独立线程的管理大大复杂化。维持一个读请求最好是使用我在 LOOPBACK 中使用的方法，用一个完成例程使 URB 循环提交。

USBINT 例子演示了怎样用循环 URB 管理中断管道。我写了一些辅助例程来协助这个工作：

- **CreateInterruptUrb** 创建 URB 和一个关联的 IRP。设备扩展中的域 **PollingUrb** 和 **PollingIrp** 指向这两个结构。我们在处理 IRP_MN_START_DEVICE 时调用这个函数。
- **DeleteInterruptUrb** 删除 URB。无论何时，当我们关闭设备时就调用这个函数来释放 IRP 和 URB 占用的内存。
- **StartInterruptUrb** 发送一个 URB 来循检设备的中断端点。无论何时当我们激活设备时就调用这个函数。
- **OnInterrupt** 是一个标准的 I/O 完成函数，作为设备的中断服务例程。它看起来象这样：

```
NTSTATUS OnInterrupt(PDEVICE_OBJECT junk, PIRP Irp, PDEVICE_EXTENSION pdx)
{
```

```

if (NT_SUCCESS(Irp->IoStatus.Status))
{
    KdPrint(("USBINT - Interrupt!\n"));           <-1
    StartInterruptUrb(pdx->DeviceObject);        <-2
}

return STATUS_MORE_PROCESSING_REQUIRED;          <-3
}

```

1. 这里进行中断处理。在 **USBINT** 例子中，这里的代码将增加未决中断计数或者完成一个未决的 IOCTL，应用程序使用该 IOCTL 来了解中断何时发生。
2. 在此，我们用相同的 URB 初始化另一轮中断循检。
3. 返回 STATUS_MORE_PROCESSING_REQUIRED，因为我们不希望 IoCompleteRequest 做任何事。

关于 **USBINT** 例子

随书光盘中的 **USBINT** 例子演示了如何管理一个有中断管道的设备。设备固件(在 EZUSB 子目录)定义了一个中断端点。每次当你按下然后再释放 Anchor Chips 开发板上的 F1 按钮时，固件就把 LED 显示的整数值加 1 并向中断端点送出 4 字节数据，在下一个主机 IN 事务时中断端点将交付这 4 个字节。驱动程序不断地尝试读该端点。测试程序发出 DeviceIoControl 调用来累加并显示所发生的中断数。按 **Ctrl+Break** 结束测试程序。设备上显示的数应与测试程序显示数据的低位相同。

重新初始化中断循检 IRP 的正确方法是调用 **IoReuseIrp** 而不是仅仅清除 **Cancel** 标志，不幸的是，Win98 中没有实现 **IoReuseIrp** 函数，在 Win98 系统中装入 **WDMSTUB.VXD**(包含该函数的一个实现)可以解决这个问题。

控制请求

如果你回头查看表 11-2，会看到有 11 种标准控制请求类型。我们从不明确地发出 **SET_ADDRESS** 请求，这个请求是当总线驱动程序察觉到有新设备插入总线时发出的。而当我们的 **WDM** 驱动程序获得控制时，总线驱动程序已经为设备赋予了地址并且已经读出了设备描述符。我们已经讨论过读取描述符、设置配置或接口的控制请求。在这段中，我们将讨论其余的控制请求。

控制特征

如果我们想设置或清除设备、接口、端点的特征，可以提交一个特征 URB。例如，下面代码设置了一个厂商定义的接口特征：

```

URB urb;
UsbBuildFeatureRequest(&urb,
    URB_FUNCTION_SET_FEATURE_TO_INTERFACE,
    FEATURE_LED_DISPLAY,
    1,
    NULL);
status = SendAwaitUrb(fdo, &urb);

```

函数的第二个参数指明我们设置或清除的特征是属于设备、接口、端点，还是厂商专有的实体。这个参数可以有八种可能的值，不用我告诉你，你完全可以用下面的公式组合出你需要的形式：

URB_FUNCTION_ [SET | CLEAR] _FEATURE_TO_ [DEVICE | INTERFACE | ENDPOINT | OTHER]

`UsbBuildFeatureRequest` 中的第三个参数指明请求修改的特征。在 **FEATURE** 例子中，我发明了一个称为 **FEATURE_LED_DISPLAY** 的特征。第四个参数指出被寻址的任何类型实体。在这个例子中，我希望寻址接口 1。

USB 定义了两个标准特征：远程唤醒特征和端点停止特征。然而，你不必自己设置或清除这些特征，总线驱动程序会自动完成这些操作。当你发出一个 `IRP_MN_WAIT_WAKE` 请求时(参见第八章“电源管理”)，总线在设备当前配置允许远程唤醒的情况下自动使能设备的远程唤醒特征。当你发出一个 `RESET_PIPE URB` 时总线驱动程序自动向端点发出清除停止特征请求。

关于 **FEATURE** 例子

随书光盘中的 **FEATURE** 例子演示了怎样设置或清除特征。设备固件(EZUSB 子目录)定义了一个没有端点的设备。该设备支持一个接口级特征 42，42 是 **FEATURE_LED_DISPLAY** 在驱动程序中引用的值。当该特征设置时，**Anchor Chips** 开发板上的 7 段 LED 变量将显示一个时间值。当该特征被清除时，LED 仅显示一个小数点。**FEATURE** 设备驱动程序(位于 **SYS** 子目录)包含设置和清除特征的代码，此外还练习了一些响应 **IOCTL** 请求的其它控制命令。

测试程序(位于 **TEST** 子目录)是一个 **Win32** 控制台应用程序，它通过执行 **DeviceIoControl** 来设置特征；发出其它 **DeviceIoControl** 调用获得状态掩码，配置号，当前接口的替换设置；然后等待 5 秒；最后执行另一个 **DeviceIoControl** 清除特征。每次运行这个测试程序，你需要观察开发板上的 LED 显示 5 秒钟。

测定状态

如果你想获得设备、接口、端点的当前状态，你可以象下面这样给出 **URB**：

```
URB urb;
USHORT epstatus;
UsbBuildGetStatusRequest(&urb,
    URB_FUNCTION_GET_STATUS_FROM_ENDPOINT,
    <index>,
    &epstatus,
    NULL,
    NULL);
SendAwaitUrb(fdo, &urb);
```

你可以指定四种 **URB** 功能码，如表 11-8。

设备的状态掩码指出设备是否是自供电，是否可以远程唤醒，见图 11-13。端点的掩码指出端点当前是否处于停止状态，见图 11-14。**USB** 规范现在又定义了接口级的电源管理状态位。参见 <http://www.usb.org/developers/devclass.html> 的“USB Feature Specification: Interface Power Management”文章。**USB** 从不规定厂商专用状态位，它们由厂商自己定义。

表 11-8. 用于取状态的 **URB** 功能码

操作码	从何处接收状态
<code>URB_FUNCTION_GET_STATUS_FROM_DEVICE</code>	整个设备
<code>URB_FUNCTION_GET_STATUS_FROM_INTERFACE</code>	指定的接口
<code>URB_FUNCTION_GET_STATUS_FROM_ENDPOINT</code>	指定的端点
<code>URB_FUNCTION_GET_STATUS_FROM_OTHER</code>	厂商专有对象

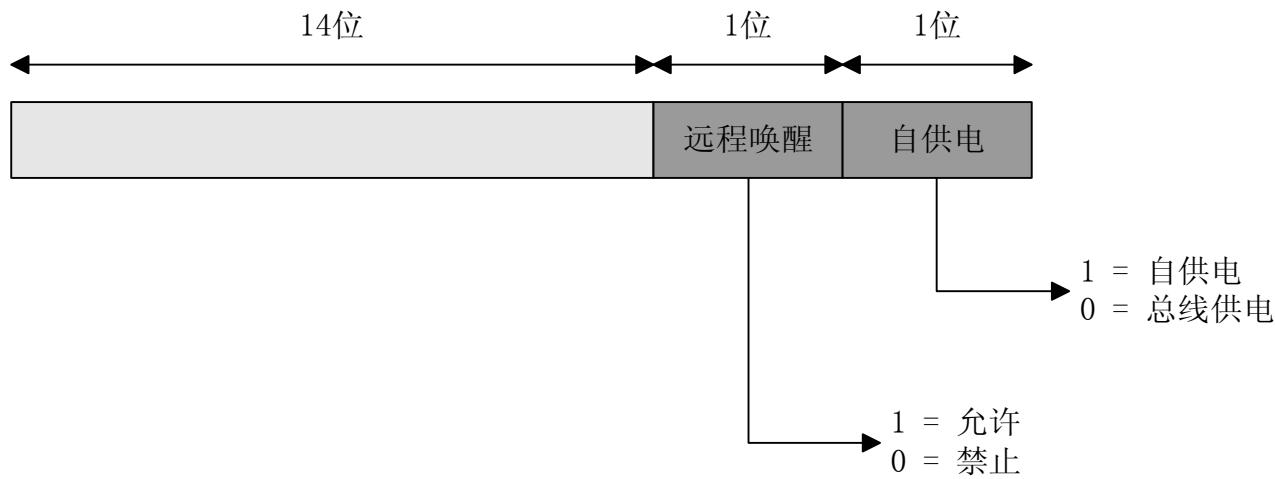


图 11-13. 设备状态位

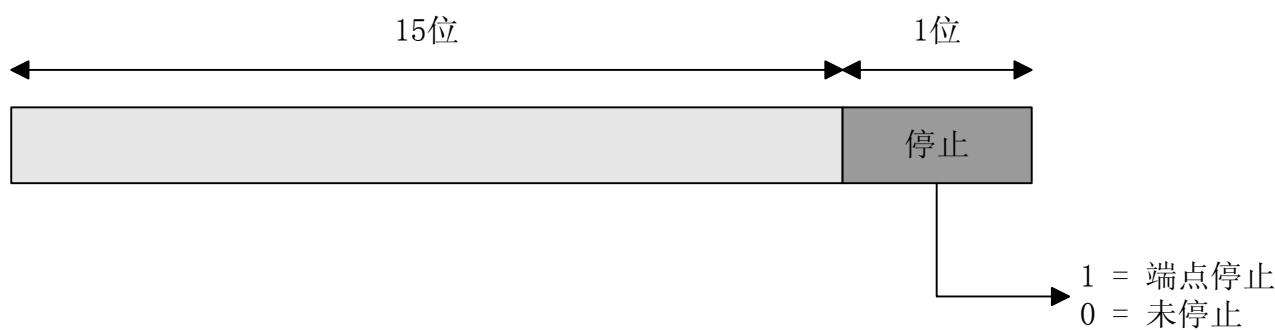


图 11-14. 端点状态位

管理等时管道

使用等时管道可以实现主机和设备之间有规律的时间敏感的数据交换。总线驱动程序将使用 90% 的总线带宽进行等时和中断传输。这意味着在每 1ms 的帧中将有足够的保留时间以容纳中断和等时端点(当前活动)的最大量传输。图 11-15 显示了这个概念。设备 A 和 B 都有一个等时端点，每帧中都有一个固定的时间段为等时端点保留。设备 C 有一个中断端点，其循检频率是每两帧一次；每两帧中它将拥有一小部分的保留时间。在没有中断端点循检的帧中，空出的带宽可以用于批量传输或其它目的。

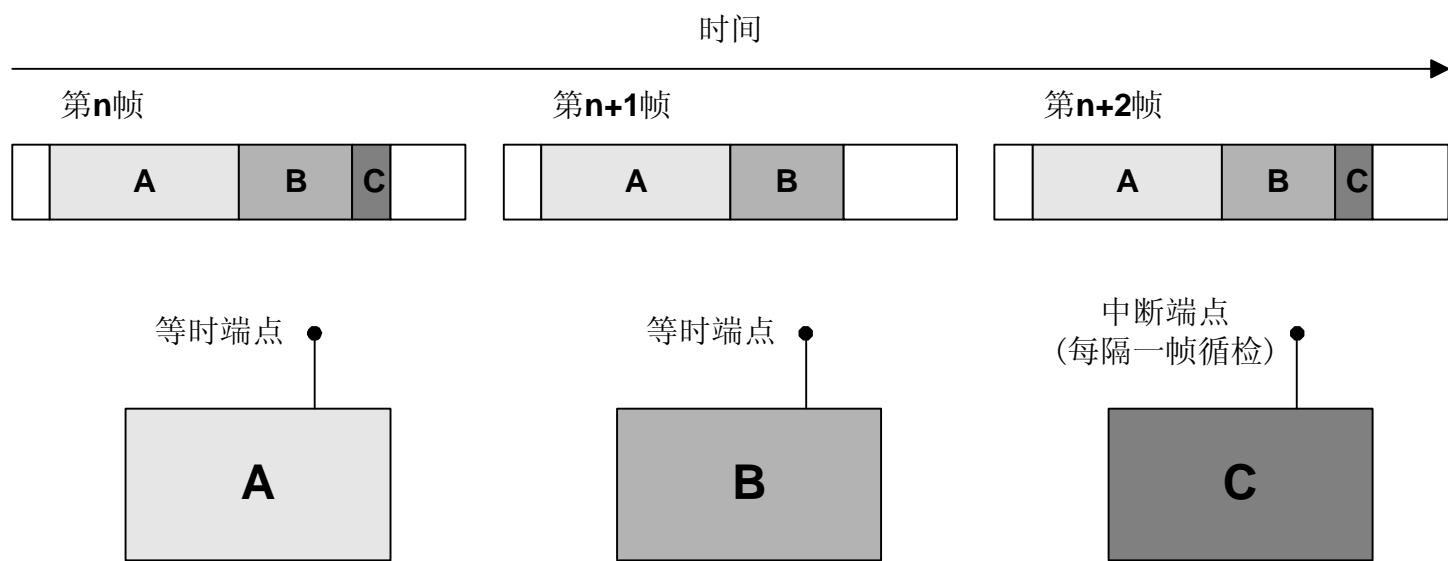


图 11-15. 等时端点和中断端点的带宽分配

保留带宽

当你使能一个接口时，总线驱动程序就为你保留了带宽。保留带宽就象买电影票，如果你不使用也不会退还。所以，对于含有等时端点的接口，应该在需要时才使能该端点，并且应使用最大传输量与你的要求相适应的替换设置。通常，有等时能力的设备其接口都有一个默认替换设置，该默认设置中没有任何等时或中断端点。当你需要使用等时能力时，应使能同一接口中含有等时或中断端点的替换设置。

下面例子明确了带宽保留的机制。USBISO 例子中的一个接口就有两个设置，默认设置(可能是替换设置 0)不包含任何端点，而替换设置 1 中有一个最大传输量为 256 字节的等时端点。见图 11-16。

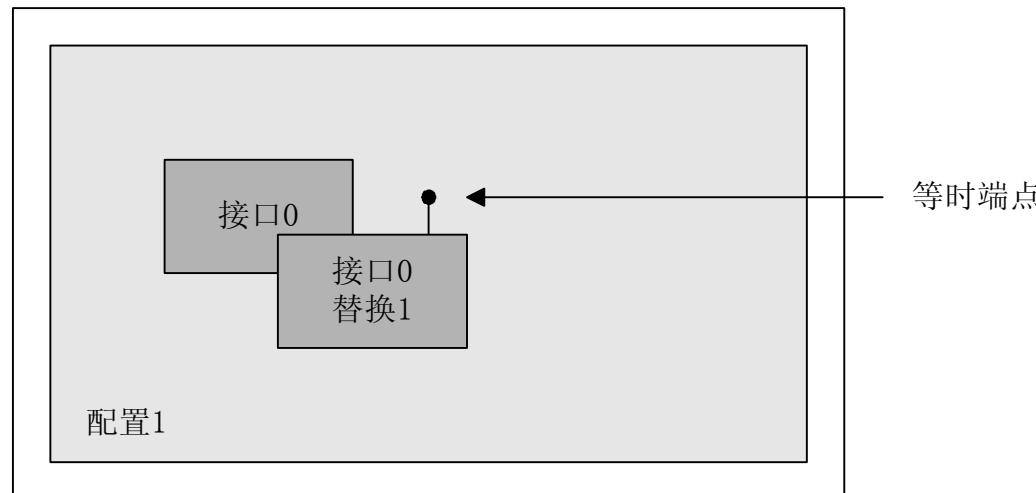


图 11-16. USBISO 设备的描述符结构

在 StartDevice 例程执行时，我们选择了默认接口设置。因为默认接口设置没有任何等时或中断管道，这样我们就不用保留任何带宽。当我们需要打开设备的句柄时，我们调用 **SelectAlternateInterface** 辅助函数切换到正常的接口设置(注意，这里我省略了错误检测)。

```
NTSTATUS SelectAlternateInterface(PDEVICE_OBJECT fdo)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PUSB_INTERFACE_DESCRIPTOR pid = USBD_ParseConfigurationDescriptorEx(pdx->pcd,
        pdx->pcd,
        0,
        1,
        -1,
        -1,
        -1,
        -1);

    ULONG npipes = pid->bNumEndpoints;
    ULONG size = GET_SELECT_INTERFACE_REQUEST_SIZE(npipes);
    PURB urb = (PURB) ExAllocatePool(NonPagedPool, size);
    RtlZeroMemory(urb, size);
    UsbBuildSelectInterfaceRequest(urb, size, pdx->hconfig, 0, 1);           <--3
    urb->UrbSelectInterface.Interface.Length = GET_USBD_INTERFACE_SIZE(npipes); <--4
    urb->UrbSelectInterface.Interface.Pipes[0].MaximumTransferSize = PAGE_SIZE;
    NTSTATUS status = SendAwaitUrb(fdo, &urb);                                <--5
    if (NT_SUCCESS(status))                                                 <--6
    {
        pdx->hinpipe = urb.UrbSelectInterface.Interface.Pipes[0].PipeHandle;     <--7
        status = STATUS_SUCCESS;
    }
    ExFreePool(urb);
    return status;
}
```

1. 在我们为 URB 分配空间之前，需要知道有多少个管道描述符。最普通的方法是在那个复合配置描述符中找出接口为 0，替换设置为 1 的描述符。该描述符含有端点号，与我们要打开的管道号相同。

2. GET_SELECT_INTERFACE_REQUEST_SIZE 宏计算指定管道号的接口选择请求的字节数。然后，为 URB 分配内存并初始化为 0。真正的例子代码还对 **ExAllocatePool** 调用是否成功做了检测。
3. 这里，我们创建了一个接口号为 0，替换设置为 1 的 URB。
4. 我们还必须做两个初始化步骤才能完成 URB 的设置。
5. 当我们提交这个 URB 时，USBD 自动关闭该接口的当前设置，包括它的所有端点。然后，USBD 通知设备使能指定的替换设置，并为其中的端点创建管道描述符。如果新接口设置打开失败，USBD 将重新打开以前的接口设置，所有以前的接口设置和管道句柄仍然有效。
6. 如果接口打开失败，**SendAwaitUrb** 辅助函数仅简单地返回一个错误值。关于这里的错误处理我将在后面讨论。
7. 除了在设备级选择新接口，USBD 还创建了一个管道描述符数组，我们可以从这个数组中提取管道句柄。

如果没有足够的带宽，接口选择调用也会失败。测试 URB 的状态可以找出失败的原因：

```
if (URB_STATUS(&urb) == USBD_STATUS_NO_BANDWIDTH)
...
...
```

处理带宽不足会有一些问题，当前的操作系统没有为竞争带宽的驱动程序提供一个便利方法来磋商带宽分配，也没有提供驱动程序请求带宽失败时的通知机制，也许我们能让出自己的一些带宽。但你还有另外两种选择，一个办法是提供多个替换接口设置，每一个设置中的等时端点都有不同的最大传输量。当申请带宽失败时，你可以选择最大传输量小一些的设置直到成功为止。

聪明的用户可能会查看 Windows 2000 设备管理小程序(applet)的 USB 主控制器属性页，见图 11-17，它显示了当前的带宽分配。双击任何一个列出的设备将显示该设备的属性页，一个设计良好的属性页应能与相连的设备驱动程序通信，从而让用户修改所占用的带宽。

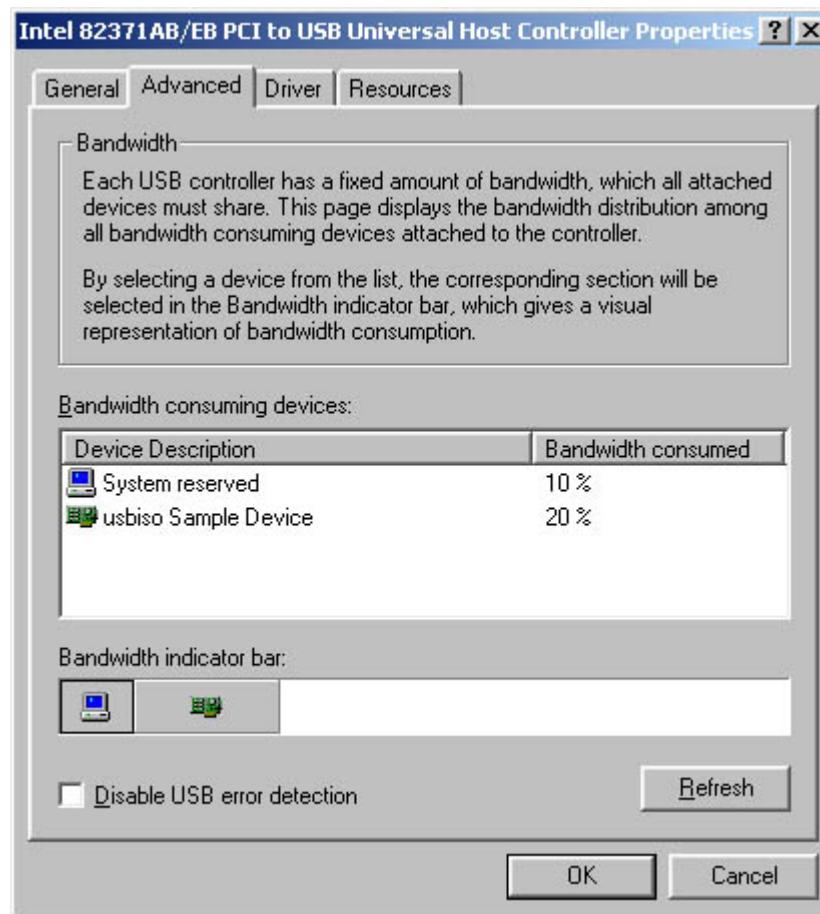


图 11-17. USB 主控制器的属性页

带宽缺乏的另一个解决办法是允许失败的 IRP 通知应用程序带宽不足，应用程序再向用户报告带宽不足。也许用户会从 USB 上拔出某些设备，这样你的设备就可以获得足够的带宽。这就是我在 **USBISO** 例子中使用的方法，我没有把响应带宽分配失败的代码放到测试程序中，TEST 的反应仅仅是失败。为了采用这个方法，你需要知道失败信息是怎样被报告回用户模式的。虽然 URB 以 **USBD_STATUS_NO_BANDWIDTH** 失败，但从内部控制 IRP 中取回的代码将是 **STATUS_DEVICE_DATA_ERROR**，这并不能说明失败的原因。应用程序调用 **GetLastError** 也只能收到 **ERROR_CRC** 错误代码。结果，到现在为止还没有简单的办法使应用程序能知道真正的失败原因。如果你对这个问题感兴趣，阅读下面文字框中的内容。

应用程序怎样发现带宽不足?

假设你已经做了 USBISO 所做的一切并在收到一个 IRP_MJ_CREATE 时试图选择一个高带宽的替换接口。再假设当带宽不足时, 你完成了带有状态码 STATUS_DEVICE_DATA_ERROR 的 IRP。但你的应用程序最终将收到 ERROR_CRC 错误码, 就象我在上面说的那样。应用程序不能向你发送一个 IOCTL 去查找真正的错误原因, 因为它没有你的设备句柄, 你曾经响应 IRP_MJ_CREATE 失败, 记得吗? 所以你也许需要一种方法去打开不保留带宽的设备句柄, 然后你需要其它办法使应用程序请求带宽, 可能用一个 IOCTL 操作。或者你的应用程序仅仅把 **CreateFile** 调用返回的 ERROR_CRC 解释为带宽不足。实际的数据错误毕竟很少, 所以这种解释大部分时间都是对的。

但是, 最好的解决办法是有一个代表“带宽不足”的 NTSTATUS 代码和一个匹配的 Win32 错误代码。你应该把眼睛盯在 NTSTATUS.H 和 WINERROR.H 的未来版本上。

USBISO 在接收到 IRP_MJ_CLOSE 时应再选择原来的无任何管道的默认接口设置, 以释放带宽。

初始化离散的等时传输

你既可以以读写离散数据块的方式使用等时管道, 也可以以生产或消费连续数据流的方式使用等时管道。数据流方式可能是使用等时管道最频繁的方式。如果你想操作流管道, 除了要了解 USB 总线驱动程序的工作机制之外, 你还必须了解并解决关于数据缓冲、速率匹配, 等等的其它问题。操作系统的内核流部件可以处理所有这些问题。不幸的是, 我没有时间在本书中包括关于内核流的章节。我仅讲述等时管道上的离散传输。

为了读写等时管道, 你当然应使用带有适当功能码的 URB。但这里还有一些关于创建和提交等时 URB 的问题。首先, 你必须明白设备怎样把传输分成包。通常, 设备可以接受或发出任何小于端点声明的最大包容量的数据(任何剩余的带宽都将空闲)。设备使用的包大小与以下这些数值没有任何必要的关系, 它们是端点最大包容量、单 URB 能携带的最大数据量、在一系列事务中设备与应用程序可以交换的数据量。例如, 对于 USBISO 设备的固件, 尽管描述符中表明其等时端点每帧可以处理 256 字节, 但实际上它仅以 16 字节的包工作。在你构造 URB 之前, 必须先知道这些包有多大。因为 URB 必须为每个要交换的包添加一个描述符数组, 每个描述符必须指出包有多大。

下面简化代码分配一个等时 URB:

```
ULONG length = MmGetMdlByteCount(Irp->MdlAddress);
ULONG packsize = 16; // a constant in USBISO
ULONG npackets = (length + packsize - 1) / packsize;
ASSERT(npackets <= 255);
ULONG size = GET_ISO_URB_SIZE(npackets);
PURB urb = (PURB) ExAllocatePool(NonPagedPool, size);
RtlZeroMemory(urb, size);
```

上面代码的关键步骤是使用 **GET_ISO_URB_SIZE** 宏来计算一个等时 URB 所需的全部内存大小, 该 URB 传输给定数量的数据包。一个单独的 URB 一次最多能容纳 255 个等时包。限制应用程序仅仅传输 255 个包是不实际的, USBISO 例子中的实际程序要比上面代码复杂些, 可以不受这个限制。

由于没有用于创建等时 URB 的宏(**UsbBuildXxxRequest**), 所以我们必须手动初始化新的 URB:

```
urb->UrbIsochronousTransfer.Hdr.Length = (USHORT) size;
urb->UrbIsochronousTransfer.Hdr.Function = URB_FUNCTION_ISOCH_TRANSFER;
urb->UrbIsochronousTransfer.PipeHandle = pdx->hInPipe;
urb->UrbIsochronousTransfer.TransferFlags = USBD_TRANSFER_DIRECTION_IN |
                                             USBD_SHORT_TRANSFER_OK |
                                             USBD_START_ISO_TRANSFER_ASAP;
urb->UrbIsochronousTransfer.TransferBufferLength = length;
urb->UrbIsochronousTransfer.TransferBufferMDL = Irp->MdlAddress;
```

```

urb->UrbIsochronousTransfer.NumberOfPackets = npackets;

for (ULONG i = 0; i < npackets; ++i, length -= packsize)
{
    urb->UrbIsochronousTransfer.IsoPacket[i].Offset = i * packsize;
}

```

包描述符数组整体地描述了我们读写用的全部数据缓冲区。该缓冲区在虚拟内存中必须连续，即你必须用一个 **MDL** 来描述它。每个包描述符仅包含“部分缓冲区”的偏移和长度，而不是一个实际的指针。主控制器驱动程序有责任设置该长度；你必须设置偏移。

启动一个等时传输的第二个问题是定时(**timing**)。USB 在每 1ms 的帧中标识一个自增数。这对于开始于特定帧的传输是重要的。USBD 允许你在 URB 的 **StartFrame** 域明确地指出帧号。USBISO 并不需要定时，它设置 **USBD_START_ISO_TRANSFER_ASAP** 标志表明传输应尽快开始。

等时处理的最后一个问题是传输怎样结束。尽管一个或多个包有数据错误，但 URB 本身是成功的。URB 的 **ErrorCount** 域指出了有多少包出现了错误。如果该值在传输结束时不为 0，你可以查找包描述符各自的状态域。

获得可接受的性能

为了获得多 URB 等时传输的可接受性能，你的驱动程序应该比我给出的例子更复杂。当一个 URB 完成时，你希望总线驱动程序立即开始下一个 URB 的处理。使用完成例程(在 LOOPBACK 中使用过)并不能达到足够快。使数据传输更快的最简单的策略是在 USBISO 例子中使用的方法：创建一组“IRP/URB 对”并一起提交它们。

注意

创建多个 IRP 的需求将带来更复杂的取消逻辑。如果我们能使用 **UrbLink** 域把多个 URB 串联到一个 IRP 上，就可以避免使用下面这些复杂的代码。

USBISO 读写逻辑中的基本思想是当上一个辅助 IRP 完成时，使用当前辅助 IRP 的完成例程完成主读写 IRP。为了实现这个想法，我声明了下面有特别用途的上下文结构：

```

typedef struct _RWCONTEXT {
    PDEVICE_EXTENSION pdx;
    PIRP mainirp;
    NTSTATUS status;
    ULONG numxfer;
    ULONG numirps;
    LONG numpending;
    LONG refcnt;
    struct {
        PIRP irp;
        PURB urb;
        PMDL mdl;
    } sub[1];
} RWCONTEXT, *PRWCONTEXT;

```

IRP_MJ_READ 的派遣例程(USBISO 不能处理 IRP_MJ_WRITE 请求)计算完成传输所需要的辅助 IRP 个数并分配一个上面描述的上下文结构：

```

ULONG packsize = 16;
ULONG segsize = USBD_DEFAULT_MAXIMUM_TRANSFER_SIZE;
if (segsize / packsize > 255)
    segsize = 255 * packsize;

```

```

ULONG numirps = (length + segsize - 1);
ULONG ctxsize = sizeof(RWCONTEXT) + (numirps - 1) * sizeof(((PRWCONTEXT) 0)->sub);
PRWCONTEXT ctx = (PRWCONTEXT) ExAllocatePool(NonPagedPool, ctxsize);
RtlZeroMemory(ctx, ctxsize);
ctx->numirps = ctx->numpending = numirps;
ctx->pdx = pdx;
ctx->mainirp = Irp;
ctx->refcnt = 2;
Irp->Tail.Overlay.DriverContext[0] = (PVOID) ctx;

```

上面程序最后两条语句的意图将在讨论 **USBISO** 的取消逻辑时再作解释。现在我们执行一个循环来构造 **numirps** 个 **IRP_MJ_INTERNAL_DEVICE_CONTROL** 请求，在每一次循环，我们调用 **IoAllocateIrp** 创建 **IRP**。我们还分配一个 **URB** 来控制一个阶段的传输，和一个“部分 **MDL**”来描述当前阶段的部分主 **I/O** 缓冲区。我们在 **RWCONTEXT** 中记录 **IRP**、**URB**，和部分 **MDL** 的地址。我们用与以前相同的方式初始化 **URB**。然后，我们初始化辅助 **IRP** 的前两个 **I/O** 堆栈单元，见下框：

```

IoSetNextIrpStackLocation(subirp);
PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(subirp);
stack->DeviceObject = fdo;
stack->Parameters.Others.Argument1 = (PVOID) urb;
stack->Parameters.Others.Argument2 = (PVOID) mdl;

stack = IoGetNextIrpStackLocation(subirp);
stack->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;
stack->Parameters.Others.Argument1 = (PVOID) urb;
stack->Parameters.DeviceIoControl.IoControlCode = IOCTL_INTERNAL_USB_SUBMIT_URB;

IoSetCompletionRoutine(subirp,
                      (PIO_COMPLETION_ROUTINE) OnStageComplete,
                      (PVOID) ctx,
                      TRUE,
                      TRUE,
                      TRUE);

```

第一个堆栈单元由我们安装的 **OnStageComplete** 完成例程所使用。第二个堆栈单元由低级驱动程序使用。

一旦我们建立完所有的 **IRP** 和 **URB**，就把它们提交到总线驱动程序。在提交之前，我们应该谨慎地检查主 **IRP** 是否被取消，并且有必要为主 **IRP** 安装一个完成例程。派遣例程的尾部代码如下。

```

IoSetCancelRoutine(Irp, OnCancelReadWrite);
if (Irp->Cancel)
{
    status = STATUS_CANCELLED;
    if (IoSetCancelRoutine(Irp, NULL))
        --ctx->refcnt;
}
else
    status = STATUS_SUCCESS;

IoSetCompletionRoutine(Irp,
                      (PIO_COMPLETION_ROUTINE) OnReadWriteComplete,
                      (PVOID) ctx,
                      TRUE,
                      TRUE,
                      TRUE);

IoMarkIrpPending(Irp);

```

```

IoSetNextIrpStackLocation(Irp);

if (!NT_SUCCESS(status))
{
    for (i = 0; i < numirps; ++i)
    {
        if (ctx->sub[i].urb)
            ExFreePool(ctx->sub[i].urb);
        if (ctx->sub[i].mdl)
            IoFreeMdl(ctx->sub[i].mdl);
    }
    CompleteRequest(Irp, status, 0);
    return STATUS_PENDING;
}

for (i = 0; i < numirps; ++i)
    IoCallDriver(pdx->LowerDeviceObject, ctx->sub[i].irp);

return STATUS_PENDING;

```

主IRP的取消处理

USBISO 例子使用了两个完成例程，它们是用于主 IRP 的 **OnReadWriteComplete**，用于每个辅助 IRP 的 **OnStageComplete**。我需要解释 USBISO 怎样处理主 IRP 的取消操作。取消操作是一种关照处理，因为我们提交大量辅助 IRP 时需要时间。直到所有辅助 IRP 都完成后主 IRP 才能完成。所以，我们应该提供一种方法取消主 IRP 和所有等待的辅助 IRP。

回想一下第五章“**I/O 请求包**”，IRP 取消操作会带来许多潜在的同步问题。由于我们不能控制辅助 IRP 完成例程的定时，所以一旦我们提交这些辅助 IRP，它们就由总线驱动程序所拥有，这使 USBISO 的取消逻辑变得更复杂。假设我们写出了下面取消例程：

```

VOID OnCancelReadWrite(PDEVICE_OBJECT fdo, PIRP Irp)
{
    IoReleaseCancelSpinLock(Irp->CancelIrql);
    PRWCONTEXT ctx = (PRWCONTEXT) Irp->Tail.Overlay.DriverContext[0];           <--1
    for (ULONG i = 0; i < ctx->numirps; ++i)
        IoCancelIrp(ctx->sub[i].irp);                                         <--2
        <additional steps>
}

```

1. 由于前面我们已经把 RWCONTEXT 的地址保存到 IRP 的 **DriverContext** 域中，所以这里我们可以读出该地址。只要我们拥有 IRP，就可以使用 **DriverContext**。因为我们在派遣例程中返回 STATUS_PENDING，所以我们仍拥有该 IRP。
2. 这里，我们取消所有的辅助 IRP。如果某个 IRP 已经完成或者正在活动，则对应的 **IoCancelIrp** 调用将不做任何事。如果某个辅助 IRP 仍旧在主控制器驱动程序的队列中，那么主控制器驱动程序的取消例程将运行并完成该辅助 IRP。在这三种情况中，我们可以确信所有的辅助 IRP 将被很快完成。

该版本的 **OnCancelReadWrite** 只差一步就完成了，我将在同步问题之后再讲述这一步骤。我在完成例程中设置两个直观的错误来演示这个问题。下面程序是阶段完成例程，它是全部传输的一个阶段：

```

NTSTATUS OnStageComplete(PDEVICE_OBJECT fdo, PIRP subirp, PRWCONTEXT ctx)
{
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);                  <--1
    PIRP mainirp = ctx->mainirp;
    PURB urb = (PURB) stack->Parameters.Others.Argument1;
    if (NT_SUCCESS(Irp->IoStatus.Status))

```

```

ctx->numxfer += urb->UrbIsochronousTransfer.TransferBufferLength;           <--2
else
    ctx->status = Irp->IoStatus.Status;                                     <--3
    ExFreePool(urb);                                                       <--4
    IoFreeMdl((PMDL) stack->Parameters.Others.Argument2);
    IoFreeIrp(subirp);                                                 //don't do this <--5
    if (InterlockedDecrement(&ctx->numpending) == 0)                      <--6
    {
        IoSetCancelRoutine(mainirp, NULL);                                //also needs some work
        mainirp->IoStatus.Status = ctx->status;
        IoCompleteRequest(mainirp, IO_NO_INCREMENT);
    }
return STATUS_MORE_PROCESSING_REQUIRED;
}

```

1. 该堆栈单元是由派遣例程分配的额外堆栈单元。我们需要该阶段的 URB 地址，而一个堆栈单元是保存该地址的最方便地方。
2. 当一个阶段正常完成时，我们更新主 IRP 的传输累加计数。**numxfer** 的最终值将存入主 IRP 的 **IoStatus.Information** 域。
3. 由于我们把整个上下文结构清 0，所以 **status** 的初始状态为 **STATUS_SUCCESS**。如果某阶段发生错误，该语句将记录下错误状态。最后，该值将存入主 IRP 的 **IoStatus.Status** 域。
4. 我们不再需要 URB 和该阶段的部分 MDL，所以我们在这里释放了它们。
5. 在这里调用 **IoFreeIrp** 是错误的。
6. 当最后一个阶段完成后，我们同时也完成了主 IRP。一旦我们提交辅助 IRP，这里便是完成主 IRP 的唯一的地方，所以必须确保主 IRP 指针有效。

这是一个简化版本的主 IRP 完成例程，加入了另一个错误例子：

```

NTSTATUS OnReadWriteComplete(PDEVICE_OBJECT fdo, PIRP Irp, PRWCONTEXT ctx)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) ctx->pdx;
    if (Irp->Cancel)                                              <--1
        Irp->IoStatus.Status = STATUS_CANCELLED;
    else if (NT_SUCCESS(Irp->IoStatus.Status))
        Irp->IoStatus.Information = ctx->numxfer;

    ExFreePool(ctx);                                              //don't do this <--2

    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);                  <--3
    return STATUS_SUCCESS;                                         <--4
}

```

1. 如果有人试图放弃主 IRP，该语句将设置对应的结束状态。
2. 在这里释放上下文结构将是一个问题。
3. **IoReleaseRemoveLock** 调用释放了在派遣函数中获取的删除锁。
4. 如果我们在此返回除 **STATUS_MORE_PROCESSING_REQUIRED** 之外的任何值，**IoCompleteRequest** 将继续其工作，而不改变 IRP 的完成状态。

我已经暴露出一个与 IRP 取消操作关联的同步问题，假设我们在 **OnStageComplete** 中调用了 **IoFreeIrp**，而我们的取消例程又被调用，那么传递给 **IoCancelIrp** 的指针将是非法指针。或者，如果取消例程与 **OnReadWriteComplete** 同时被调用，那么取消例程可能会遇到一个被删除的上下文结构。

你可能提出几种解决办法，能否在 **OnStageComplete** 中将上下文结构的辅助 IRP 指针置空？可否在调用 **IoCancelIrp** 前用 **OnCancelReadWrite** 先检查 IRP 的有效性？（是的，但这仍不能保证 **IoFreeIrp** 调用不挤进 **OnCancelReadWrite** 的检测期间，而此时 **IoCancelIrp** 正好完成 IRP 的取消操作相关域的修改）能否用一个自

旋锁来保护多个清除步骤？(这是一个可怕的想法，因为你在调用耗时函数时占有自旋锁) 能否利用 Windows 2000 的新特征，在 APC 例程中，Windows 2000 总是清除已完成的 IRP？(不能，原因我在第五章已讨论过)

我思考这个问题有一段时间，最后我得到了灵感。我们为什么不用一个参考计数器来保护上下文结构和辅助 IRP 指针，这样取消例程和主完成例程就都能清除 IRP，这就是我们最后要做的。我在上下文结构中加入一个参考计数器(**refcnt**)并初始化为 2。一个为取消例程参考，另一个主完成例程参考。我写了下面辅助函数，它用于释放产生问题的内存对象。

```
BOOLEAN DestroyContextStructure(PRWCONTEXT ctx)
{
    if (InterlockedDecrement(&ctx->refcnt) > 0)
        return FALSE;
    for (ULONG i = 0; i < ctx->numirps; ++i)
        if (ctx->sub[i].irp)
            IoFreeIrp(ctx->sub[i].irp);
    ExFreePool(ctx);
    return TRUE;
}
```

我在取消例程的最后调用了这个辅助函数：

```
VOID OnCancelReadWrite(PDEVICE_OBJECT fdo, PIRP Irp)
{
    IoReleaseCancelSpinLock(Irp->CancelIrql);
    PRWCONTEXT ctx = (PRWCONTEXT) Irp->Tail.Overlay.DriverContext[0];
    for (ULONG i = 0; i < ctx->numirps; ++i)
        IoCancelIrp(ctx->sub[i].irp);
    PDEVICE_EXTENSION pdx = ctx->pdx;
    if (DestroyContextStructure(ctx))
    {
        CompleteRequest(Irp, STATUS_CANCELLED, 0);
        IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    }
}
```

我省略了阶段完成例程中的 **IoFreeIrp** 调用，并加入几行代码以减少参考计数：

```
NTSTATUS OnStageComplete(PDEVICE_OBJECT fdo, PIRP subirp, PRWCONTEXT ctx)
{
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    PIRP mainirp = ctx->mainirp;
    PURB urb = (PURB) stack->Parameters.Others.Argument1;
    if (NT_SUCCESS(Irp->IoStatus.Status))
        ctx->numxfer += urb->UrbIsochronousTransfer.TransferBufferLength;
    else
        ctx->status = Irp->IoStatus.Status;
    ExFreePool(urb);
    IoFreeMdl((PMDL) stack->Parameters.Others.Argument2);
    if (InterlockedDecrement(&ctx->numpending) == 0)
    {
        if (IoSetCancelRoutine(mainirp, NULL))
            InterlockedDecrement(&ctx->refcnt);
        mainirp->IoStatus.Status = ctx->status;
        IoCompleteRequest(mainirp, IO_NO_INCREMENT);
    }
    return STATUS_MORE_PROCESSING_REQUIRED;
}
```

}

IoSetCancelRoutine 返回取消指针的以前值。如果为空，则取消例程已经被调用，可以调用 **DestroyContextStructure**。如果不为空，则取消例程不可能再被调用，所以我们必须用尽取消例程在上下文结构中的全部权利。

我还把主完成例程中的无条件 **ExFreePool** 调用换成 **DestroyContextStructure** 调用：

```
NTSTATUS OnReadWriteComplete(PDEVICE_OBJECT fdo, PIRP Irp, PRWCONTEXT ctx)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) ctx->pdx;
    if (Irp->Cancel)
        Irp->IoStatus.Status = STATUS_CANCELLED;
    else if (NT_SUCCESS(Irp->IoStatus.Status))
        Irp->IoStatus.Information = ctx->numxfer;

    if (DestroyContextStructure(ctx))
    {
        IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
        return STATUS_SUCCESS;
    }
    else
        return STATUS_MORE_PROCESSING_REQUIRED;
}
```

这就是这个额外逻辑的工作过程。如果取消例程被调用，它将通过上下文结构为每个辅助 IRP 调用 **IoCancelIrp** 函数。即使这些 IRP 都被完成，这些调用仍旧是安全的，因为我们根本就没有调用 **IoFreeIrp**。上下文的引用也是安全的，因为我们也没有调用 **ExFreePool**。取消例程以 **DestroyContextStructure** 调用结束，该调用减少参考计数的值。如果主完成例程根本就没有运行，**DestroyContextStructure** 将返回 FALSE，因此取消例程将返回。此时上下文结构仍然存在而且是完好的。完成例程的最后一次调用 **DestroyContextStructure** 将释放辅助 IRP 和上下文结构本身。然后，它放开删除锁并返回 **STATUS_SUCCESS**，这样主 IRP 就可以结束完成操作。

假设取消例程和主完成例程的调用以另一种顺序发生，那么 **OnReadWriteComplete** 中的 **DestroyContextStructure** 调用将简单地减少参考计数并返回 FALSE，而 **OnReadWriteComplete** 将返回 **STATUS_MORE_PROCESSING_REQUIRED**。上下文结构仍然存在，这还可以确定我们仍拥有该 IRP。取消例程中的 **DestroyContextStructure** 调用将把参考计数减到 0，释放内存，返回 TRUE。然后，取消例程释放删除锁并为主 IRP 调用 **IoCompleteRequest**。这会造成对同一 IRP 两次调用 **IoCompleteRequest**。虽然一个 IRP 不能被完成两次，但这个禁止本身并不是针对两次调用 **IoCompleteRequest**。如果第一次 **IoCompleteRequest** 调用导致调用一个返回 **STATUS_MORE_PROCESSING_REQUIRED** 的完成例程，那么后面的重复调用不会造成问题。

这个分析的最后一种情况是当取消例程根本就没有得到调用。当然一般情况下 IRP 不会被取消。但这个事实会出现在我们为准备完成主 IRP 而调用 **IoSetCancelRoutine** 时。如果 **IoSetCancelRoutine** 返回一个非 NULL 值，我们知道对于主 IRP，**IoCancelIrp** 还没有被调用。(取消指针应该为 NULL，**IoSetCancelRoutine** 应该返回 NULL) 另外我们知道，现在我们自己的取消例程绝不会被调用了，因此也没有机会去减少参考计数。所以我们需要手工减少参考计数，以便 **OnReadWriteComplete** 能调用 **DestroyContextStructure** 释放内存。

同步在哪里？

你会注意到我前面介绍的派遣例程中的取消代码没有用自旋锁来保护测试操作。这段代码和某些 **IoCancelIrp** 假定调用者之间有潜在的同步问题，因为 **IoSetCancelRoutine** 是一个互锁交换操作，并且 **IoCancelIrp** 是在调用 **IoSetCancelRoutine** 之前设置了 Cancel 标志。参见第五章中 **IoCancelIrp** 的讨论。

派遣例程中的第一个 **IoSetCancelRoutine** 调用会发生在 **IoCancelIrp** 设置 Cancel 标志之后，在 **IoCancelIrp** 自己调用 **IoSetCancelRoutine** 之前。派遣例程会看到 Cancel 标志被设置并做第二次 **IoSetCancelRoutine** 调用。如果这个第二次 **IoSetCancelRoutine** 调用碰巧发生在 **IoCancelIrp** 调用 **IoSetCancelRoutine** 之前，

则取消例程将不会被调用。我们还要减少对上下文结构的参考计数，以便在第一次调用 `DestroyContextStructure` 时该结构能被释放。

如果派遣例程的第二次 `IoSetCancelRoutine` 调用发生在 `IoCancelIrp` 之后，我们将不减少参考计数。其它取消例程或完成例程最终将释放上下文结构。

如果派遣例程在 `IoCancelIrp` 设置 `Cancel` 标志前测试该标志，或者如果根本就没有为该 IRP 调用 `IoCancelIrp`，那么我们将继续并启动辅助 IRP。如果 `IoCancelIrp` 在我们设置一个取消例程前被调用，那么它仅简单地设置 `Cancel` 标志并返回。这之后发生的事情与派遣例程在 `IoCancelIrp` 调用 `IoSetCancelRoutine` 之前置空取消指针所发生的情况一样。

所以你看，实现多处理器安全并不总是需要一个自旋锁：有时一个原子互锁操作也可以做到。

关联 IRP？

`IoMakeAssociatedIrp` 看起来象创建 USBISO 所需辅助 IRP 的另一种方式。`IoMakeAssociatedIrp` 后面的思想是，你可以创建许多关联 IRP 来实现一个主 IRP。当最后一个关联 IRP 完成时，I/O 管理器会自动创建主 IRP。

不幸的是，关联 IRP 并不能很好地解决 USBISO 面临的问题。最重要的是 WDM 驱动程序不能假定使用 `IoMakeAssociatedIrp`。实际上 Windows 98 中的关联 IRP 完成逻辑是不正确的，它不能在最后一个关联 IRP 结束时为主 IRP 调用完成例程。甚至在 Windows 2000 中，I/O 管理器也不能在主 IRP 被取消时取消关联 IRP。另外，针对关联 IRP 的 `IoFreeIrp` 调用发生在 `IoCompleteRequest` 内部，在任意线程中。这使安全地取消关联 IRP 更加困难。

流等时传输

在前面段中，我描述了通过等时管道的长传输技术。你可能还需要实现连续数据流传输。我在这里给出一个快速的实现框架。

在流驱动程序中，你需要为不丢任何帧的连续传输提供一个或多个数据缓冲区。你还需要至少分配两个“IRP/URB 对”用于传输。在这种情况下，即使 URB 连接域可以使用也不能帮助你：你需要知道每个 URB 完成的时间，找出这个值的唯一时间是在关联 IRP 的完成例程被调用时。

开始你象总线驱动程序提交所有 IRP。当一个 IRP 完成时，你立即再次使用它(在完成例程中)。办法是使主机控制器驱动程序的请求队列中总有 URB 准备运行，一旦当前 URB 完成，队列中的 URB 就开始执行。你还需要调整数据缓冲区的大小和数量，以及 IRP/URB 对的数量以避免由于生产者或消费者一时没有跟上设备而导致缓冲区溢出。

同步等时传输

同步性是多种等时数据流的一个重要属性。为了给出一个简单例子，假设你的计算机上有两个扬声器和一个麦克风。你希望送往扬声器的声音数据能与来自麦克风的数据同步，这样，在感觉上听见的声音可以跟上麦克风的输入。另外你还希望一个扬声器的声音输出与另一个同步。

要实现可接受的同时性有几个难点。USB 规范的第 5.10 段详细地描述了这些原因以及解决方案所需要的硬件基础。我仅概述这些难点以便我能指出 USBD 对驱动程序的用处。

数据的源和接收器会有不同的采样大小和速率。例如，一个麦克风每秒会生成 8000 字节的采样信息，而一个扬声器每秒会消耗 44100 个 32 位采样。某些硬件或软件代理必须使用缩放和插补来匹配源和接收器。

设备本身也会有内部延迟。数据源需要时间来捕获并编码数据，然后再发送给主机，数据接收器也需要时间来解码和使用数据。在这个例子中有一个数据源和两个相似的接收器，它们的延迟也许不太重要。但考虑一下多

输入设备存在的情形，每个设备都有自己的延迟特点，它们捕获同一系列外部事件的不同方面。(例如，一组麦克风和 **MIDI** 设备) 某些代理程序需要了解被各种源设备带入的延迟，以整理主机接收的数据流。另一些代理程序还需要了解接收器设备的延迟以使实际的输出信号能在恰当的时间到达外部环境。因为 **USB** 需要以帧为单位测量设备的延迟，所以驱动程序可以通过在其生成的等时传输 **URB** 中明确设置 **StartFrame** 成员来处理延迟。为了设置这个域，你需要计算帧号，从某个到达的输入数据帧号(可以从完成的 **URB** 的 **StartFrame** 成员中提取)开始或从当前帧号开始。

最后，设备必须提供某种方法来使自己的时钟与系统同步。同步是第一需要，因为时钟可能会随时间漂移(由于石英晶体的频率细微差异)或跳变(由于温度或其它波动)。**USB** 可以识别三种可选的端点时钟同步方法：异步、同步、自适应。

异步端点不能与任何外部源同步其操作。源端点通过提供的数据量来暗中通知主机其数据提交率。接收端点需要访问一个辅助同步端点，如一个中断端点，来报告其数据消耗的进展。

同步端点把其操作绑到总线的 **1kHz** 速率的帧上。它这样做或者通过把自己的时钟附属到每帧开始的 **SOF(start-of-frame)**包上，或者强制总线帧率匹配自己的时钟。**USB** 允许任何一个设备成为帧主控者，并允许其改变标准的 **1毫秒**帧周期。在驱动程序一方，发出一个功能码为 **URB_FUNCTION_TAKE_FRAME_LENGTH_CONTROL** 的 **URB** 可以使自己成为帧主控者，发出功能码为 **URB_FUNCTION_RELEASE_FRAME_LENGTH_CONTROL** 的 **URB** 放弃作为帧主控者。如果你是帧主控者，你可以发出功能码为 **URB_FUNCTION_GET_FRAME_LENGTH** 或 **URB_FUNCTION_SET_FRAME_LENGTH** 的 **URB** 来获取或设置帧长度。

一个自适应源端点有某种能够接收数据接收者反馈的方法(例如一个控制管道)，这可以使其生成与接收器匹配的采样。自适应接收端点简单地把自己的接收速率转换为已接收数据流中暗含的速率。

第十二章：安装驱动程序

在驱动程序开发过程的前期，对于用户如何安装你的驱动程序以及其服务的硬件做些考虑是重要的。Windows 2000 和 Windows 98 使用一个扩展名为 **INF** 的文本文件来控制与安装驱动程序相关的大部分活动。**INF** 文件应该由你提供。它可以存在于软盘上，或存在于随硬件包装的光盘上，或者由 Microsoft 把它放到 Windows 2000 的安装光盘上。通过 **INF** 文件你可以告诉操作系统哪一个文件需要复制到用户硬盘上，应该添加或修改哪一个注册表项，如此等等。

在本章我将讨论驱动程序安装的几个方面，引导你熟悉一个简单 **INF** 文件的各个重要部分，帮助你联系到 DDK 文档中关于 **INF** 文件的语法。我还将详细解释用于各种设备类型的设备标识符格式，如何初始化设备硬件键中的属性值，如何在驱动程序和应用程序中访问这些属性。我为本书使用的所有例子“设备”定义了一个设备类，设备类的定义过程也许会对你有用。我还将讨论一种方法，使用这种方法可以在 PnP 管理器启动你的设备时自动运行一个应用程序。

- INF 文件
- 定义设备类
- 运行应用程序
- Windows 98 兼容问题

INF文件

INF 文件包含一些名字由方括号括起来的段。大部分段都含有一系列“**keyword = value**”形式的指令。INF 文件开始于一个 **Version** 段，该段确定文件中描述的设备类型：

```
[Version]
Signature=$CHICAGO$
Class=Sample
ClassGuid={894A7460-A033-11d2-821E-444553540000}
```

Signature 可以是三种特定值之一：\$Chicago\$、\$Windows NT\$(含有一个空格)、\$Windows 95\$(也含有一个空格)。**Class** 确定设备类。表 12-1 列出了 Windows 2000 支持的预定义类。**ClassGuid** 唯一地标识该设备类。DDK 头文件 DEVGUID.H 定义了标准设备类的 GUID，DDK 文档中关于 Version 段的内容也描述了这些标准设备类。

在产品级的 INF 文件中，Version 段中还需要有 **DriverVer** 和 **CatalogFile** 语句。另外你还应该给出一个包含词“copyright”的注释(以分号开头的行)以满足 CHKINF 程序的检查。操作系统可以接受没有这个注释的 INF 文件，但 Microsoft 不会认证没有这项内容的驱动程序。所需的 INF 语法细节参见 DDK 文档。

把一个大的 INF 文件看成是一个树结构的线形描述可以更容易理解 INF 文件。一个段就是树上的一个节点，而每个指令就是指向另一个段的指针。图 12-1 显示了这个概念。

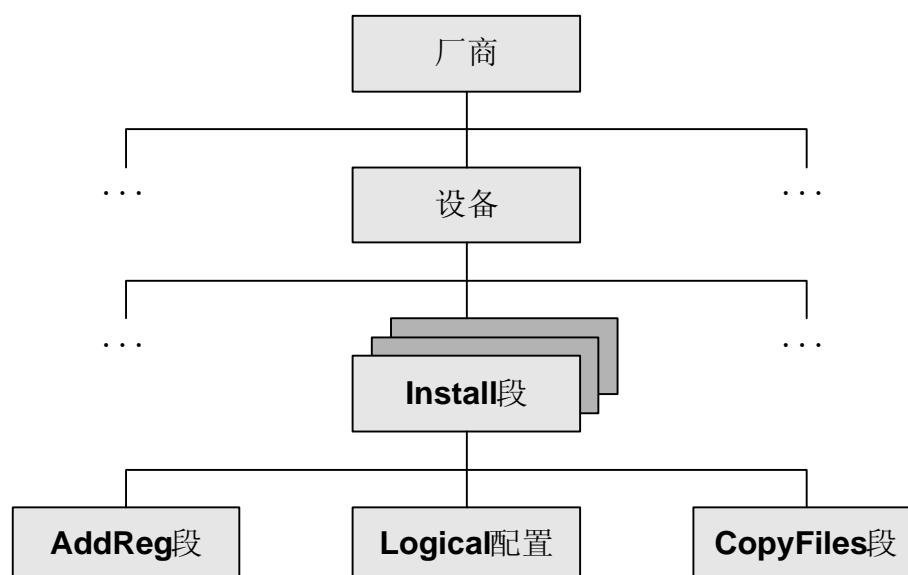


图 12-1. INF 文件的树形结构

表 12-1. INF 文件的设备类

INF 类名	描述
1394	IEEE 1394 总线控制器(不是外设)
Battery	电池设备
CDROM	CD-ROM 驱动器，包括 SCSI 和 IDE
DiskDrive	硬盘驱动器
Display	显示适配器
FDC	软盘控制器
FloppyDisk	软盘驱动器
HDC	硬盘控制器

HIDClass	人机接口设备
Image	静态图象捕捉设备，包括数码相机和扫描仪
Infrared	Serial-IR 和 Fast-IR 红外端口的 NDIS miniport 驱动程序
Keyboard	键盘
MediumChanger	SCSI 媒体交换器设备
Media	多媒体设备，包括音频、DVD、游戏杆、全动态视频捕捉设备
Modem	调制解调器
Monitor	监视器
Mouse	鼠标和其它指点设备
MTD	内存设备的内存技术驱动程序
Multifunction	多功能设备
MultiportSerial	智能多端口串行卡
Net	网络适配器
NetClient	网络文件系统和打印提供者(客户方)
NetService	网络文件系统的服务器方支持
NetTrans	网络协议驱动程序
PCMCIA	PCMCIA 和 CardBus 主机控制器(不是外设)
Ports	串行和并行口
Printer	打印机
SCSIAdapter	SCSI 和 RAID 控制器，主机总线适配器 miniports，和磁盘阵列控制器
SmartCardReader	智能卡读写器
System	系统设备
TapeDrive	磁带驱动器
USB	USB 主控制器和 hub(不是外设)
Volume	逻辑存储卷驱动程序

这棵树的根是 **Manufacturer** 段，列出了文件中有硬件描述的所有公司。例如：

```
[manufacturer]
"Walter Oney Software"=DeviceList
"Finest Organization On Earth Yet"=FOOEY

[DeviceList]
...
[FOOEY]
...
```

每个独立厂商的 **model(硬件型号)**段(例子中的 **DeviceList** 和 **FOOEY**)描述了一个或多个设备：

```
[DeviceList]
Description=InstallSectionName,DeviceId
...
```

Description 是一个人类可读的设备描述, **DeviceId** 标识一个硬件设备。**InstallSectionName** 标识(或指向)INF 文件中的另一个段, 该段含有为特定设备安装软件的指令。下面是一个例子(摘自第七章的 PKTDMA 例子):

[DeviceList]

"AMCC S5933 Development Board (DMA)"=DriverInstall,PCI\VEN_10E8&DEV_4750

系统为一个硬件安装驱动程序时需要参考 **Manufacturer** 段和 **model** 段中的信息。PnP 设备以电子方式表明其存在和身份。总线驱动程序会自动发觉 PnP 设备并使用其板上的数据构造设备标识符。然后系统寻找这个设备的预安装 INF 文件。这些 INF 文件保存在 Windows 目录的 INF 子目录中。如果系统没有找到合适的 INF 文件, 它会请求用户为其指定一个。

遗留设备不能表明自己的存在或身份。因此需要用户运行硬件安装向导来安装遗留设备并帮助其定位正确的 INF 文件。这个过程中的关键步骤是指定要安装的设备类型及厂商的名字, 见图 12-2。

硬件安装向导通过枚举特定类型设备的所有 INF 文件来构造一个类似图 12-2 的对话框, 包括所有 **Manufacturer** 段中的语句, 和每个 **manufacturer** 的所有 **model** 语句。

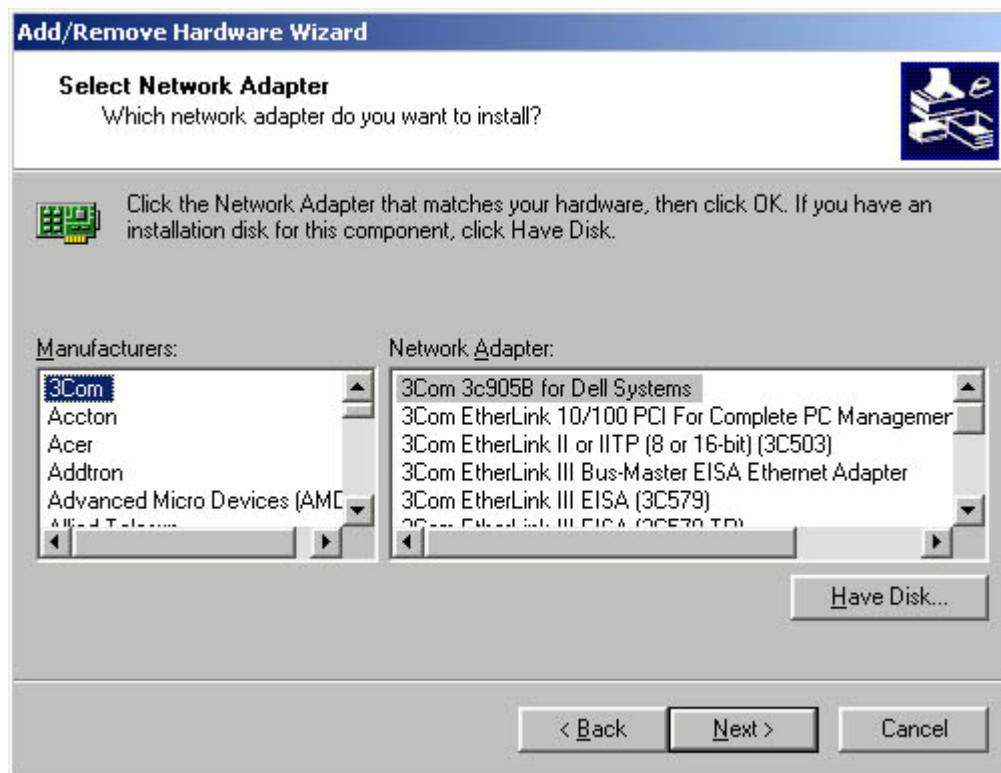


图 12-2. 安装期间选择设备

关于硬件安装向导对话框

一旦安装向导经过搜寻 PnP 设备的阶段, 它就创建一个设备类列表, 并使用 **SETUPAPI.DLL** 中的各种 **SetupDi_Xxx** 例程提取图标和描述。**SETUP API** 例程使用的信息最终来自注册表, **ClassInstall32** 段中的信息将送到注册表。并不是每个设备类都会出现在列表中, 安装向导将禁止使用有 **NoInstallClass** 属性的类的信息。

用户选择完设备类之后, 安装向导就调用 **SETUP API** 函数构造一个如文中描述的厂商和设备的列表, **ExcludeFromSelect** 语句中提到的设备将不出现在列表中。

Install 段

install 段包含指导安装程序安装所需软件的实际指示。我们还考察 PKTDMA 例子。对于这个设备, 型号段 **DeviceList** 指出 **DriverInstall** 名。我想可以把这个名看作是一个段数组的标识, 每个元素对应每个 Windows 平台。数组中的第 0 项元素含有该段(**DriverInstall**)的基本名称。你可以使与平台相关的数组元素的名称含有这个

基本名称，再加上表 12-2 中列出的一个后缀。设备安装程序搜寻有最合适后缀的 `install` 段。例如，假设你有三个 `install` 段，第一个无后缀，后两个分别带有 `.NT` 后缀和 `.NTx86` 后缀。如果你安装到 x86 平台的 Windows 2000 系统中，安装程序会使用 `.NTx86` 段。如果安装到在非 Intel 平台上运行的 Windows 2000 中，它将使用 `.NT` 段。如果你安装到 Windows 98 中，它使用无后缀段。

表 12-2. 每个平台的 `Install` 段后缀

平台	<code>Install</code> 段后缀
任何平台，包括 Windows 98	[none]
任何 Windows 2000 平台	<code>.NT</code>
运行在 Intel x86 上的 Windows 2000	<code>.NTx86</code>

由于这个搜寻规则，我的例子驱动程序的所有 INF 文件都有无后缀的 `install` 段和有“`.NT`”后缀的 `install` 段。这使得这些 INF 文件可以很好地应付 Intel 平台。(你可能已经知道，Microsoft 和 Compaq 已经不再支持 Alpha 平台上的 32 位版本 Windows 2000)

以后我将讨论以 `install` 名开头的其它 INF 段。如果你的“数组”中有多个 `install` 段，那么这些段必须包含平台相关的后缀。例如，我将讨论一个 `Services` 段，该段用于向注册表中装入驱动程序描述。你可以用 `install` 段的名字作为基本名(如 `DriverInstall`)再加上平台后缀(如 `NT`)，最后加上词 `Services`，即 `[DriverInstall.NT.Services]`。

一个典型的 Windows 2000 安装段仅包含一个 `CopyFile` 指令：

```
[DriverInstall.nt]
CopyFiles=DriverCopyFiles
```

这个 `CopyFiles` 指出我们想让安装程序用另一个 INF 段中的信息来指导文件复制。对于 PKTDMA 例子，这个段名为 `DriverCopyFiles`：

```
[DriverCopyFiles]
pktdma.sys,,,2
```

该段指示安装程序把 `PKTDMA.SYS` 复制到用户的硬盘上。

`CopyFiles` 段中的语句有下面通用形式：

```
Destination,Source,Temporary,Flags
```

Destination 是被复制文件出现在用户系统中的文件名(不包括任何路径名)。**Source** 是源文件名，如果目标文件与源文件名字相同则这里可以为空。在 Windows 98 中，如果你安装的文件在安装过程中要被使用，你可以在 **Temporary** 参数中指定一个临时名。Windows 98 将在下一次引导时把这个临时文件名该为目标文件名。该参数对于 Windows 2000 没有必要，系统会自动生成临时文件名。

Flags 参数包含一个位掩码，控制着是否要解压缩文件，以及如何处理文件同名的情况。该标志的解释需要取决于 Microsoft 是否认证了该驱动程序包。表 12-3 列出了这些标志位。表中的斜体标志在有数字签名的包中被忽略。我用双线分开互斥的标志。这样，在一个未签名的包中，你可以指定 `NOSKIP` 或 `WARN_IF_SKIP` 标志，但不能同时指定。

文件名本身不能为安装程序复制文件提供足够的信息。你还需要知道文件要复制到的目录名。另外，如果安装过程中需要多个磁盘，则需要知道哪个磁盘含有源文件。这些信息来自 INF 文件的其它段，如图 12-3 所示。在 PKTDMA 例子中，这些段如下：

```
[DestinationDirs]
DefaultDestDir=10\System32\Drivers
```

```

[SourceDisksFiles]
pktdma.sys=1

[SourceDisksNames]
1="WDM Book Companion Disc",disk1

```

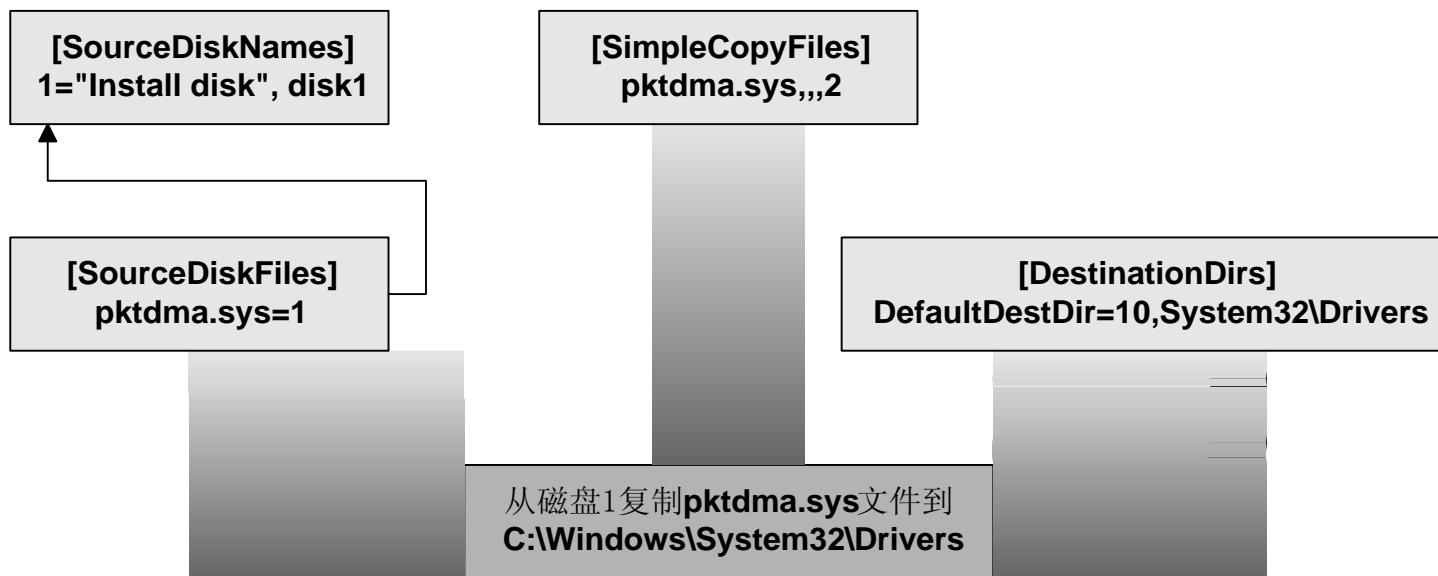


图 12-3. 文件复制的源和目标信息

表 12-3. *CopyFile* 指令中的 *Flags* 标志

符号名	数值	描述
COPYFLG_REPLACEONLY	0x00000400	仅在目标文件存在的情况下复制
COPYFLG_NODECOMP	0x00000800	不解压缩文件
COPYFLG_FORCE_FILE_IN_USE	0x00000008	总是用临时文件名复制，并在下一次引导时改名
COPYFLG_NO_OVERWRITE	0x00000010	不覆盖已存在的文件(该标志不与其它标志同用)
COPYFLG_REPLACE_BOOT_FILE	0x00001000	替换载入程序需要的引导文件，将提示用户重启动系统
COPYFLG_NOPRUNE	0x00002000	复制该文件，即使 Setup 认为其已存在
COPYFLG_NOVERSIONCHECK	0x00000004	覆盖一个文件，即使被覆盖文件比源文件版本更新
COPYFLG_NO_VERSION_DIALOG	0x00000020	不产生版本冲突对话框
COPYFLG_OVERWRITE_OLDER_ONLY	0x00000040	仅覆盖版本更旧的文件
COPYFLG_NOSKIP	0x00000002	不允许用户跳过该文件
COPYFLG_WARN_IF_SKIP	0x00000001	允许用户跳过该文件并给出警告

SourceDisksFiles 段指出安装程序可以在 1 号盘上找到 PKTDMA.SYS 文件。**SourceDiskNames** 段指出磁盘 1 有一个人类可读的标签“WDM Book Companion Disc”并包含一个名为 disk1 的文件，用于安装程序寻找并检验驱动器中是否有正确的磁盘。注意这两个段名中有一个容易忘记的字母“s”。

DestinationDirs 段指出文件复制操作的目标路径。**DefaultDestDir** 是未指定目标目录文件的默认目标路径。你可以用一个数值代码来指定目标路径，这个代码的完整列表见 DDK 文档，经常使用的仅有如下几个：

- 目录“10”为 Windows 目录(例如“\Windows”或“\Winnt”)。
- 目录“11”是 System 目录(例如，“\Windows\System”或“\Winnt\System32”)。
- 目录“12”是 Drivers 目录，指在 Windows 2000 系统中(例如，“\Winnt\System32\Drivers”)。但在 Windows 98 中这个数值有不同的含义(例如“\Windows\System\osubsys”)。

WDM 驱动程序存在于 **Driver** 目录中。如果你的 **CopyFiles** 段仅用于向 Windows 2000 安装，那么仅指定目录数值 12 就可以了。如果你还想使该段能向 Windows 98 中安装，我推荐你用“10,System32\Drivers”，它以两种方式标识 **Driver** 目录。

定义**Driver Service**

到现在为止我们所讨论的 **INF** 语法足够把你的驱动程序复制到用户硬盘上。另外我们还需要为 **PnP** 管理器做些安排，以使它知道应该载入哪个文件。段.**Services** 用于实现这个目标，见下例：

```
[DriverInstall.NT.Services]
AddService=PKTDMA,2,DriverService

[DriverService]
ServiceType=1
StartType=3
ErrorControl=1
ServiceBinary=%10%\system32\drivers\pktdma.sys
```

AddService 指令中的“2”指出 **PKTDMA service** 将成为该设备的功能驱动程序。该段名字由 **install** 段名再加上“**Services**”词组成。

这个指令的结果是使 **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services** 中出现一个 **PKTDMA** 键 (**AddService** 指令的第一个参数)。它将为内核模式(**ServiceType** 为 1)驱动程序定义 **service** 项，内核模式驱动程序由 **PnP** 管理器按需求自动装载(**StartType** 为 3)。在装入时发生的错误应该被登记，但不要阻止系统启动 (**ErrorControl** 为 1)。执行映像为**\Winnt\System32\Drivers\pktdma.sys**(**ServiceBinary** 的值)。如果你查看注册表，会看到执行文件的名存在于 **ImagePath** 名下而不是 **ServiceBinary**。

service 的名最好与驱动程序的二进制文件名相同。这可以使 **service** 名明显与驱动程序对应，还避免了两个不同 **service** 键指向同一个驱动程序所带来的问题：如果某设备使用的驱动程序与在其后启动的设备所使用的驱动程序相同，除非在不同的 **service** 名下，否则其自身不能被启动。

设备标识符

对于真正的即插即用设备，其设备标识符应该出现在厂商型号段中。即插即用设备可以以电子方式声明它们的存在和身份。总线枚举器能够自动找到这些设备，并能读出某些硬件信息，以便确定该设备为何种设备。例如，**USB** 设备的设备描述符中含有厂商和产品标识代码，**PCI** 设备的配置空间也包含厂商代码和产品代码。

当枚举器检测到一个设备时，它就构造一个设备标识串列表。列表中的一项就是设备的完整标识。而这个表项最后将用于命名注册表中的硬件键。列表中的额外表项是“兼容”标识符。**PnP** 管理器在匹配设备与 **INF** 文件时会使用列表中的所有标识符。枚举器把更专用一些的标识符放到通用的标识符前面，这样，厂商提供的专用驱动程序就可以先于通用驱动程序而被找到。构造这个串的算法取决于枚举器，见下面：

PCI设备

设备标识符的完整形式如下：

```
PCI\VEN_ffff&DEV_dddd&SUBSYS_ssaaaa&REV_rr
```

ffff 是厂商标识符，由 **PCI** 组织赋予每个厂商，**dddd** 是设备标识符，由厂商赋予每个设备，**ssaaaa** 是设备报告的子系统 id(一般为 0)，**rr** 是修订号。

例如，我的笔记本电脑上的显示适配器(基于 C&T 的 65550 芯片)有这样的标识符：

PCI\VEN_102C&DEV_00E0&SUBSYS_00000000&REV_04

设备也可以用下面标识符形式匹配 INF 中的 model 段：

```
PCI\VEN_vvvv&DEV_dddd&SUBSYS_ss:ss:ss:ss  
PCI\VEN_vvvv&DEV_dddd&REV_rr  
PCI\VEN_vvvv&DEV_dddd  
PCI\VEN_vvvv&DEV_dddd&REV_rr&CC_ccss  
PCI\VEN_vvvv&DEV_dddd&CC_ccsspp  
PCI\VEN_vvvv&DEV_dddd&CC_ccss  
PCI\VEN_vvvv&CC_ccsspp  
PCI\VEN_vvvv&CC_ccss  
PCI\VEN_vvvv  
PCI\CC_ccsspp  
PCI\CC_ccss
```

cc 来自配置空间的基础类代码，**ss** 是子类代码，**pp** 是编程接口。例如，下面这些标识符得自我的笔记本电脑的显示适配器，也可以匹配 INF 文件中的信息：

```
PCI\VEN_102C&DEV_00E0&SUBSYS_00000000  
PCI\VEN_102C&DEV_00E0&REV_04  
PCI\VEN_102C&DEV_00E0  
PCI\VEN_102C&DEV_00E0&REV_04&CC_0300  
PCI\VEN_102C&DEV_00E0&CC_030000  
PCI\VEN_102C&DEV_00E0&CC_0300  
PCI\VEN_102C&CC_030000  
PCI\VEN_102C&CC_0300  
PCI\VEN_102C  
PCI\CC_030000  
PCI\CC_0300
```

实际上，系统在安装该驱动程序时使用的是第三个标识符，它仅有一个厂商标识符和一个设备标识符。

PCMCIA设备

一个简单设备的标识符有如下形式：

PCMCIA\Manufacturer-Product-Crc

例如我的笔记本电脑上的 3Com 网卡有如下设备标识符：

PCMCIA\MEGAHERTZ-CC10BT/2-BF05

对于多功能设备上的一个独立功能，其标识符有如下形式：

PCMCIA\Manufacturer-Product-DEVddd-Crc

Manufacturer 是厂商的名字，**Product** 是产品的名字。PCMCIA 枚举器直接从卡上获取这些串。**Crc** 是该卡的 4 位 16 进制 CRC 校验和。子功能号(**ddd**)是一个无前导 0 的十进制数。

如果卡没有厂商名，则标识符将会是下面三种形式之一：

PCMCIA\UNKNOWN_MANUFACTURER-Crc

```
PCMCIA\UNKNOWN_MANUFACTURER-DEVddd-Crc  
PCMCIA\MTD-0002
```

(这最后三种选择用于没有厂商标识符的闪存卡)

除了刚描述的设备标识符，**INF** 文件的 **model** 段也可以包含这样的标识符，用 4 位十六进制的厂商代码串替换 4 位十六进制的 CRC，加上一个连字号，再加上 4 位十六进制的厂商信息代码(都来自设备硬件)。例如：

```
PCMCIA\MEGAHERTZ-CC10BT/2-0128-0103
```

SCSI设备

完整的设备标识符为：

```
SCSI\ttttvvvvvvvpppppppppppppprrrr
```

tttt 是设备类型代码，**vvvvvvvv** 是一个 8 字符的厂商标识符，**pppppppppppppppp** 是一个 16 字符的产品标识符，**rrrr** 是一个 4 字符的修订级值。设备类型代码是唯一有可变长度的标识符部件。总线驱动程序通过用设备类型代码索引一个内部串表来确定这部分设备标识符，见表 12-4。剩下部分正好是设备查询数据中出现的串，但含有特殊字符(包括空格、逗号、和任何非打印字符)，需要用下划线替换。

表 12-4. SCSI 设备的类型名

SCSI 类型代码	设备类型	通用类型
DIRECT_ACCESS_DEVICE (0)	Disk	GenDisk
SEQUENTIAL_ACCESS_DEVICE (1)	Sequential	
PRINTER_DEVICE (2)	Printer	GenPrinter
PROCESSOR_DEVICE (3)	Processor	
WRITE_ONCE_READ_MULTIPLE_DEVICE (4)	Worm	GenWorm
READ_ONLY_DIRECT_ACCESS_DEVICE (5)	CdRom	GenCdRom
SCANNER_DEVICE (6)	Scanner	GenScanner
OPTICAL_DEVICE (7)	Optical	GenOptical
MEDIUM_CHANGER (8)	Changer	ScsiChanger
COMMUNICATION_DEVICE (9)	Net	ScsiNet
	Other	ScsiOther

例如，我工作站上的一个磁盘驱动器有这样的标识符：

```
SCSI\DiskSEAGATE_ST39102LW_____0004
```

总线驱动程序也可能创建下面形式的标识符：

```
SCSI\ttttvvvvvvvpppppppppppppp  
SCSI\ttttvvvvvvv  
SCSI\vvvvvvvvpppppppppppppppp  
vvvvvvvvpppppppppppppppp  
gggg
```

在这些附加标识符的第三和第四中，**r** 仅代表修订标识符的第一个字符。在最后一个标识符中，**gggg** 是表 12-4 中的通用类型代码。

总线驱动程序也可能生成下面形式的设备标识符：

```
SCSI\DiskSEAGATE_ST39102LW_____
SCSI\DiskSEAGATE_
SCSI\DiskSEAGATE_ST39102LW_____0
SEAGATE_ST39102LW_____0
GenDisk
```

这些标识符中的最后一个(**GenDisk**)可以用于 PnP 管理器自动安装设备的驱动程序。实际上，INF 文件中仅含有这个通用标识符，因为 SCSI 驱动程序趋向于通用。

IDE设备

IDE 设备的设备标识符非常类似于 SCSI 的标识符：

```
IDE\tttvpvprrrrrrrr
IDE\vpvpvrccccccccc
IDE\tttvpvvp
vpvpvrccccccccc
gggg
```

ttt 是一个设备类型名(与 SCSI 相同)；**vvpv** 是一个串，含有厂商名、一个下划线、厂商的产品名、以及用于补足 40 个字符的足够下划线；**rrrrrrrr** 是一个 8 字符的修订号；**gggg** 是一个通用类型名(几乎与表 12-4 中的 SCSI 类型名相同)。除了 IDE 交换设备，通用类型名为 **GenChanger**；其它 IDE 通用名与 SCSI 相同。

例如，下面是我桌面系统中的 IDE 硬盘驱动器的设备标识符：

```
IDE\DiskMaxtor_91000D8_____SASX1B18
IDE\Maxtor_91000D8_____SASX1B18
IDE\DiskMaxtor_91000D8_____
Maxtor_91000D8_____SASX1B18
GenDisk
```

ISAPNP设备

ISAPNP 枚举器构造两个硬件标识符：

```
ISAPNP\id
*altid
```

id 和 **altid** 是设备的 EISA 样式标识符，三个字母标识厂商，4 位 16 进制数标识特定设备。如果设备是多功能卡上的一个功能，则列表中的第一个标识符有这样的形式：

```
ISAPNP\id_DEVnnnn
```

nnnn 是该功能的 10 进制索引(带有前导 0)

例如，Crystal Semiconductor 声卡的 **codec** 功能有这两个硬件标识符：

```
ISAPNP\CSC6835_DEV0000
```

*CSC0000

第二个标识符与实际的 INF 文件匹配。

USB设备

完整的设备标识符为：

USB\VID_<vvvv>&PID_<cccc>&REV_<rrrr>

<vvvv> 是 4 位 16 进制厂商代码，由 USB 委员会赋予厂商，**<cccc>** 是 4 位 16 进制产品代码，由厂商赋予设备，**<rrrr>** 是修订代码。这三个值都出现在设备的接口描述符或设备描述符中。

INF model 段也能指定这些选择：

USB\VID_<vvvv>&PID_<cccc>
USB\CLASS_<cc>&SUBCLASS_<ss>&PROT_<pp>
USB\CLASS_<cc>&SUBCLASS_<ss>
USB\CLASS_<cc>
USB\COMPOSITE

<cc> 是来自设备描述符或接口描述符的类代码，**<ss>** 是子类代码，**<pp>** 是协议代码。这些值都是 2 位 16 进制数格式。

1394 设备

1394 总线驱动程序为一个设备构造下面这些标识符：

1394\<VendorName>&<ModelName>
1394\<UnitSpecId>&<UnitSwVersion>

<VendorName> 是硬件厂商的名字，**<ModelName>** 标识设备，**<UnitSpecId>** 标识软件规范权限，**<UnitSwVersion>** 标识软件规范。构造这些标识符的信息来自设备的配置 ROM。

如果设备有厂商和型号名称串，则 1394 总线驱动程序使用第一个标识符作为硬件 ID，用第二个标识符作为唯一的兼容 ID。如果设备缺少一个厂商名称串或缺少型号名称串，则总线驱动程序使用第二个标识符作为硬件 ID。

我的计算机上没有 1394 总线，我从驱动程序开发者 Jeff Kellam 得到了两个例子。第一个例子是一个 Sony 照相机，其设备标识符为：

1394\SONY&CCM-DS250_1.08

第二个例子是处于诊断模式的 1394 总线本身，这个设备标识符是：

1394\031887&040892

通用设备标识符

PnP 管理器还使用通用设备标识符，这些通用设备存在于许多不同的总线上。这些标识符的形式如下：

*PNP<cccc>

dddd是一个4位16进制的类型标识符。在本书出版时，这些标识符的官方列表见www.microsoft.com/hwdev/download/respec/devids.txt。

硬件键

硬件键记录了驱动程序所管理的特定硬件实例的信息。每个设备枚举器都有自己的注册表键，在HKEY_LOCAL_MACHINE\System\CurrentControlSet\Enum下。当枚举器找到一个有特定标识符的设备时，它就为该标识符创建一个键，并为同一个设备的每个实例创建一个子键。例如，PKTDMA设备的标识符为PCI\VEN_10E8&DEV_4750。该设备第一个实例的硬件键应该象这样：

```
\Registry\Machine\System\CurrentControlSet\Enum\PCI\VEN_10E8&DEV_4750\BUS_00&DEV_04&FUNC_00
```

标准属性

PnP管理器在硬件键中保存设备的某些标准信息。在WDM驱动程序中调用IoGetDeviceProperty函数可以获得这个信息，属性代码见表12-5。

表12-5. 硬件键中的标准设备属性

属性名	值名	源	描述
DevicePropertyDeviceDescription	DeviceDesc	model语句的第一个参数	设备描述
DevicePropertyHardwareId	HardwareID	model语句的第三个参数	标识设备
DevicePropertyCompatibleIDs	CompatibleIDs	由总线驱动程序在检测期间创建	可匹配的设备类型
DevicePropertyClassName	Class	Version段的Class参数	设备类名
DevicePropertyClassGuid	ClassGUID	Version段的ClassGuid参数	设备类的唯一标识
DevicePropertyDriverKeyName	Driver	安装过程中自动创建	指定驱动程序的服务(software)键名
DevicePropertyManufacturer	Mfg	被找到model段中的Manufacturer	硬件厂商名
DevicePropertyFriendlyName	FriendlyName	INF文件中的AddReg,或类安装器	给用户看的“友好”名

例如，为了获得设备的描述，可以使用下面代码。(见DEVPROP例子中的AddDevice函数)

```
WCHAR name[256];
ULONG junk;
status = IoGetDeviceProperty(pdo, DevicePropertyDeviceDescription, sizeof(name), name, &junk);
KdPrint((DRIVERNAME "- AddDevice has succeeded for '%ws' device\n", name));
```

注意表12-5，除了友好名，PnP管理器和总线驱动程序共同负责自动创建所有标准设备属性。你可以在INF文件中给出一个友好名：

```
[DriverInstall.NT.hw]
AddReg=DriverHwAddReg

[DriverHwAddReg]
HKR,,FriendlyName,, "Packet DMA Demonstration Device"
```

注意，在同一计算机上如果你采用这个方法，那么这种类型的每个设备最后会有相同的友好名。多个设备有同一个友好名会使用户迷惑。如果你察觉出可能会存在重复的友好名，应该提供一个安装协助 DLL 来计算唯一的名字。

用户模式应用程序可以使用 **SetupDiGetDeviceRegistryProperty** 函数接收标准设备属性。在使用 setup API 枚举已寄存接口的过程中可以使用下面方法：

```
HDEVINFO info = SetupDiGetClassDevs(...);  
SP_DEVINFO_DATA did = { sizeof(SP_DEVINFO_DATA) };  
SetupDiGetDeviceInterfaceDetail(info, ..., &did);  
TCHAR fname[256];  
SetupDiGetDeviceRegistryProperty(info,  
    &did,  
    SPDRP_FRIENDLYNAME,  
    NULL,  
    (PBYTE) fname,  
    sizeof(fname),  
    NULL);
```

用于指定接收各种属性的 SPDRP_XXX 值请参见 **SetupDiGetDeviceRegistryProperty** 函数的 DDK 文档。

正如你所见，你必须为 **SetupDiGetDeviceRegistryProperty** 提供一个设备信息集句柄(一个 HDEVINFO)和一个 SP_DEVINFO_DATA 结构。如果你在枚举设备接口实例循环的中间，你可以很容易地做到这一点。但如果你仅有设备的符号名呢？你可以使用下面技巧：

```
LPCTSTR devname; // someone gives you this  
HDEVINFO info = SetupDiCreateDeviceInfoList(NULL, NULL);  
SP_DEVICE_INTERFACE_DATA ifdata = { sizeof(SP_DEVICE_INTERFACE_DATA) };  
SetupDiOpenDeviceInterface(info, devname, 0, &ifdata);  
SP_DEVINFO_DATA did = { sizeof(SP_DEVINFO_DATA) };  
SetupDiGetDeviceInterfaceDetail(info, &ifdata, NULL, 0, NULL, &did);
```

在此你可以继续以正常方式调用象 **SetupDiGetDeviceRegistryProperty** 这样的函数。

注意

在 Windows 98 和 Windows NT4 中，应用程序使用 CFGMGR32 API 来获得设备信息和与 PnP 管理器交互。为了兼容，Windows 98 和 Windows 2000 继续支持这些 API，但 Microsoft 不鼓励新代码中使用这些 API。因此我在此也不给出调用这些 API 的例子。

非标准属性

PnP 管理器在硬件键下创建了一个名为 **Device Parameters** 的子键。该子键包含设备的非标准属性。你可以在 INF 中的添加注册表段中初始化非标准属性：

```
[DriverInstall.nt.hw]  
AddReg=DriverHwAddReg  
  
[DriverHwAddReg]  
HKR,,SampleInfo,"% wdmbook%\chap7\pktdma\pktdma.htm"
```

WDM 驱动程序可以容易地用 **IoOpenDeviceRegistryKey** 函数打开设备参数键的句柄。应用程序使用 **SetupDiOpenDevRegKey** 函数可以访问该键。

INF文件工具

如果你查看 Windows 2000 DDK 中的 TOOLS 子目录，你会找到两个用于处理 INF 文件的有用工具。GENINF 能帮助你创建一个新 INF 文件，CHKINF 能帮助你检验 INF。

CHKINF实际上是一个运行PERL脚本的BAT文件，该脚本将检验INF文件的合法性。显然，使用这个工具你必须先有一个PERL解释程序。可从<http://www.perl.com>得到。

可以从命令行上运行 CHKINF。例如：

```
E:\Ntddk\tools\chkinf>chkinf C:\wdmbook\chap12\devprop\sys\device.inf
```

CHKINF 在一个 HTM 子目录中产生一个 HTML 输出文件。图 12-4 显示了我在检测 DEVPROP 例子的 DEVICE.INF 所给出的输出。

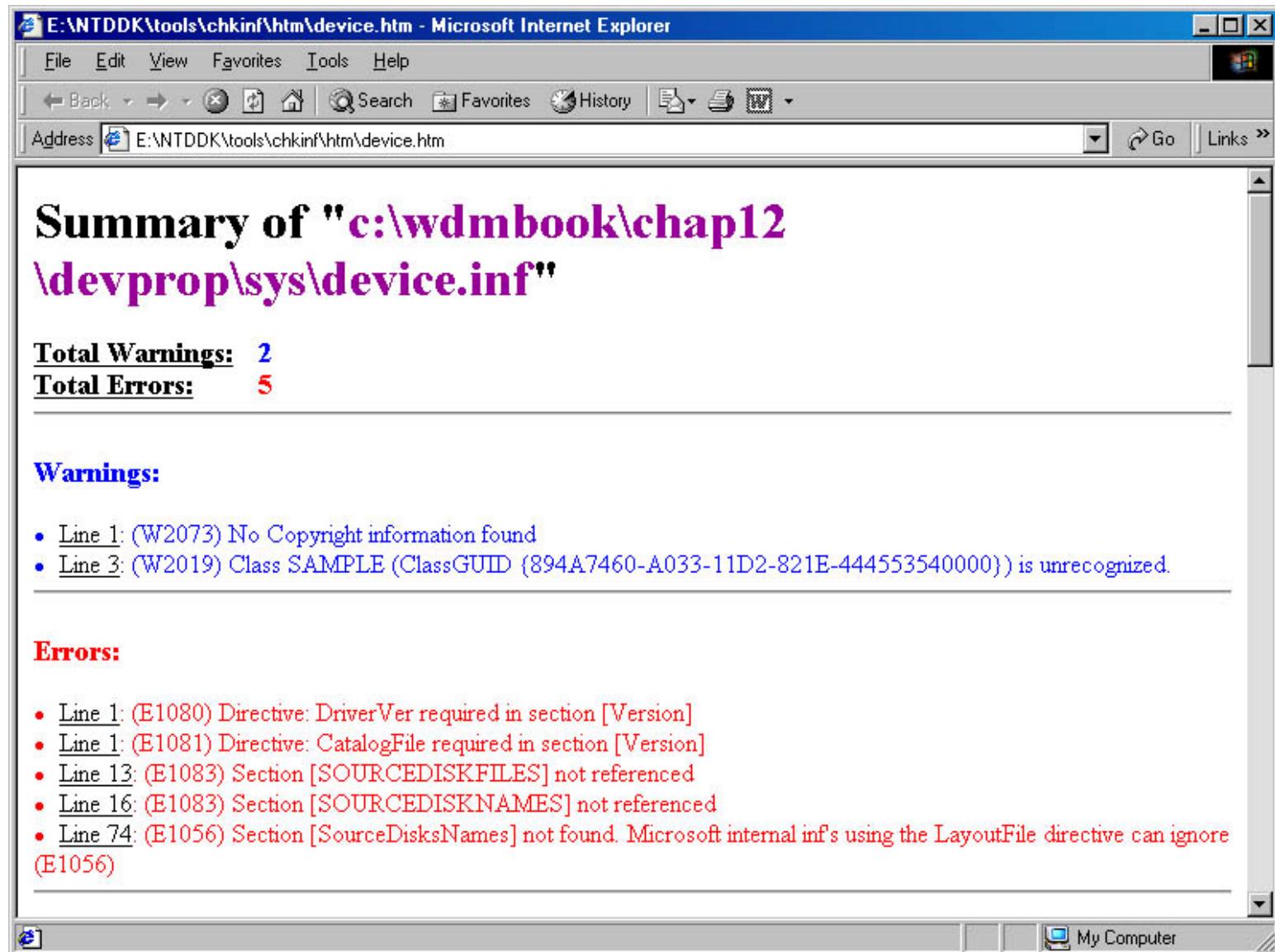


图 12-4. CHKINF 的输出例子

在 Windows 2000 中，设备安装程序在 Windows NT 目录的 SETUPAPI.LOG 文件中登记其操作的各种信息。你可以通过手工修改键 HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Setup 下的注册表项来控制登记信息的详细程度和登记文件名。这些设置的详细信息请参见 DDK 文档。

定义设备类

假设你有一个设备不属于 Microsoft 定义的任何设备类。当你开始测试设备和驱动程序时，虽然使用 **Unknown** 可能会成功，但产品级的设备不应该假定为 **Unknown** 类。你可以在 INF 文件中定义一个新设备类，在这一节我将解释如何创建一个定制设备类。

我们以前讨论过的 INF 例子就是依靠一个这样的定制设备类：

```
[Version]
Signature=$CHICAGO$
Class=Sample
ClassGuid={894A7460-A033-11d2-821E-444553540000}
```

实际上，本书上的所有例子都使用 **Sample** 类。

定义新设备类时，你仅需要完成一件任务：运行 **GUIDGEN** 为这个类生成一个唯一的 GUID。为了使这个设备类的用户接口更完美，你还可以做一些额外工作，如为设备管理器写一个属性页程序，放一些特殊的表项到该类可以使用的注册表键里。你还可以提供过滤器驱动程序和超越参数，它们将用于该类的每个设备。你可以用 INF 文件中的语句控制这些额外特征。例如：

```
[ClassInstall32]
AddReg=SamclassAddReg
CopyFiles=SamclassCopyFiles

[SamclassAddReg]
HKR,,,,"WDM Book Sample"
...

[SamclassCopyFiles]
...
```

上面给出的设备类注册表项目在设备管理器或在添加新硬件向导的设备类型列表中会变成“友好名”。我将解释一些需要添加到 **class** 键中的额外注册表项。

注意

我的每个 INF 文件都没有 **ClassInstall32** 段。因为例子光盘的 **setup** 程序已经把必要的类信息直接放到注册表中了。但如果你在一个产品级的驱动程序包中定义了自己的设备类，你将需要这个段。注意 Microsoft 鼓励用 INF 文件安装新类。

属性页程序

回到第一章，图 1-6 显示了 **Sample** 设备类的属性页。SAMCLASS 例子程序是该属性页程序的源代码。下面我解释它是如何工作的。

设备类的属性页程序是一个 32 位 DLL，包含下面内容：

- 属性页支持的每个类的输出入口点
- 每个属性页的对话框资源
- 每个属性页的对话框过程

一般，一个 DLL 可以为多个设备类提供属性页，例如 Microsoft 随操作系统提供的某些属性页 DLL。但 SAMCLASS 仅为一个设备类提供了一个页。它的唯一的输出入口点指向下面函数：

```

extern "C"
BOOL CALLBACK EnumPropPages (PSP_PROPSHEETPAGE_REQUEST p, LPFNADDPROPSHEETPAGE
AddPage, LPARAM lParam)
{
    PROPSHEETPAGE page;
    HPROPSHEETPAGE hpage;
    memset(&page, 0, sizeof(page));
    page.dwSize = sizeof(PROPSHEETPAGE);
    page.hInstance = hInst;
    page.pszTemplate = MAKEINTRESOURCE(IDD_SAMPAGE);
    page.pfnDlgProc = PageDlgProc;
    <some more stuff>
    hpage = CreatePropertySheetPage(&page);
    if (!hpage)
        return TRUE;
    if (!(*AddPage)(hpage, lParam))
        DestroyPropertySheetPage(hpage);
    return TRUE;
}

```

当设备管理器要为一个设备构造属性页面时，它参考其 **class** 键，看它是否有一个属性页提供者。你可以在 INF 文件中指定一个提供者，如下：

```

[SamclassAddReg]
HKR,,EnumPropPages32,"samclass.dll,EnumPropPages"

```

设备管理器装入你指定的 DLL(SAMCLASS.DLL)并调用入口点函数(**EnumPropPages**)。如果该函数返回 TRUE，则设备管理器就显示属性页，否则不显示。通过调用 **AddPage** 例程该函数还可以加入更多的页。

在 SP_PROPSHEETPAGE_REQUEST 结构中有两个非常有用的信息：一个是设备信息集的句柄，另一个是 SP_DEVINFO_DATA 结构的地址。只要属性页可见，这些数据项就是有效的，你可以在属性页的对话框过程中访问它们。Windows SDK 的 Programming 101 会教你如何去做。首先创建一个辅助结构，其地址作为 PROPSHEETPAGE 结构的 **lParam** 成员传递给 **CreatePropertySheetPage**：

```

struct SETUPSTUFF {
    HDEVINFO info;
    PSP_DEVINFO_DATA did;
};

BOOL EnumPropPages(...)
{
    PROPSHEETPAGE page;
    ...
    SETUPSTUFF* stuff = new SETUPSTUFF;
    stuff->info = p->DeviceInfoSet;
    stuff->did = p->DeviceInfoData;
    page.lParam = (LPARAM) stuff;

    page.pfnCallback = PageCallbackProc;
    page.dwFlags = PSP_USECALLBACK;
    ...
}

UINT CALLBACK PageCallbackProc(HWND junk, UINT msg, LPPROPSHEETPAGE p)
{

```

```

if (msg == PSPCB_RELEASE && p->lParam)
    delete (SETUPSTUFF*) p->lParam;
return TRUE;
}

```

WM_INITDIALOG 消息的 lParam 参数是指向同一个 PROPSHEETPAGE 结构的指针，所以你可以在那里收到 stuff 指针。然后你可以使用 SetWindowLong 和 GetWindowLong 在 DWL_USER 中保存任何需要的信息。在 SAMCLASS 中，我选择了描述例子驱动程序的 readme 文件的名字。下面我将给出代码。

你还需要提供一种方法来删除无用的 SETUPSTUFF 结构。最简单的方式(不管你是否得到 WM_INITDIALOG 消息)是使用一个属性页回调函数。

在定制属性页中你可以做任何事情。对于一个例子类，我想要一个按钮，按下这个按钮会显示出每个例子设备的解释。为此，我把命名含有解释信息文件的值 **SampleInfo** 放到设备的硬件键中。为了调出该解释文件的查看程序，我们可以调用 **ShellExecute**，它会解释文件扩展名并定位一个适当的查看程序。对于书中的例子，其解释文件是 HTML 格式，因此查看程序应该是 Web 浏览器。

SAMCLASS 的大多数工作发生在 WM_INITDIALOG 处理程序中。(这里忽略了错误检测)

```

case WM_INITDIALOG:
{
    SETUPSTUFF* stuff = (SETUPSTUFF*) ((LPPROPSHEETPAGE) lParam)->lParam;
    BOOL okay = FALSE;
    TCHAR name[256];
    SetupDiGetDeviceRegistryProperty(stuff->info,
                                    stuff->did,
                                    SPDRP_FRIENDLYNAME,
                                    NULL,
                                    (PBYTE) name,
                                    sizeof(name),
                                    NULL);
    SetDlgItemText(hdlg, IDC_SAMNAME, name);

    HKEY hkey = SetupDiOpenDevRegKey(stuff->info,
                                    stuff->did,
                                    DICS_FLAG_GLOBAL,
                                    0,
                                    DIREG_DEV,
                                    KEY_READ);
    DWORD length = sizeof(name);
    RegQueryValueEx(hkey, "SampleInfo", NULL, NULL, (LPBYTE) name, &length);
    LPSTR infofile;
    DoEnvironmentSubst(name, sizeof(name));                                     <--3
    infofile = (LPSTR) GlobalAlloc(GMEM_FIXED, strlen(name)+1);
    strcpy(inffile, name);
    SetWindowLong(hdlg, DWL_USER, (LONG) infofile);
    RegCloseKey(hkey);
    break;
}

```

1. 这里我们为设备确定 **FriendlyName** 并把它放到一个静态文本控制中。
2. 接下来的语句从硬件键的 parameter 子键中确定 **SampleInfo** 文件名。
3. 放到注册表中的串的形式为%wdmbook%\chap12\devprop\devprop.htm， %wdmbook% 为 WDMBOOK 环境变量。
DoEnvironmentSubst 是一个标准的 Win32 API，用于扩展环境变量。
4. 我需要在某个地方保存 SampleInfo 文件名， **SetWindowLong** 可以方便地做到这一点。

当用户按下属性页上的 More Information 按钮时，对话框过程就收到一个 WM_COMMAND 消息，它的处理如下：

```
case WM_COMMAND:  
    switch (LOWORD(wParam))  
    {  
        case IDB_MOREINFO:  
        {  
            LPSTR infofile = (LPSTR) GetWindowLong(hdlg, DWL_USER);  
            ShellExecute(hdlg, NULL, infofile, NULL, NULL, SW_SHOWNORMAL);  
            return TRUE;  
        }  
    }  
    break;
```

ShellExecute 将运行与 SampleInfo 文件相关联的应用程序，即你的 Web 浏览器，于是你就看到了这个文件的内容。

其它相关信息

在前面段中，我解释了如何用 **EnumPropPages32** 注册表项控制设备属性页的显示。这里还有一些控制类特征的其它注册表项：

- **Installer32** 指定一个 DLL，该 DLL 用于执行类所属设备的安装功能。写一个类安装程序是一项巨大的工程，并不是因为 DDK 文档没有跟上这方面的软件。
- **Class** 是类名，出现在 INF 文件的 Class=语句中。
- **Icon** 为该类指定用户接口显示图标。其值为一个含有十进制整数的串。正值代表 Installer32 DLL 中的图标；文档中说如果你没有类安装程序，系统会使用 EnumPropPages32 DLL 中的图标，但我没有遇到这种情况。负值代表 SETUPAPI.DLL 中的图标(用绝对值索引)。如果你不指定一个图标，系统会使用一个灰色菱形来代表。我决定用值-5 来指定 Sample 类的图标，这个图标很象 PCI 卡。实际上，系统使用它作为网卡图标，但我喜欢这个图标。
- **NoInstallClass**，如果存在且不等于 0，则指出某枚举器将自动检测属于该类的任何设备。如果类有这个属性，硬件向导将不在设备类列表中列出该类。
- **SilentInstall**，如果存在且不等于 0，将使 PnP 管理器在安装该类的设备时不呈现任何对话框给用户。
- **UpperFilters** 和 **LowerFilters** 为过滤器驱动程序指定 service 名。PnP 管理器为每个属于该类的驱动程序装入这些过滤器。(注意在设备硬件键中指定的过滤器驱动程序仅用于一个设备)
- **NoDisplayClass**，如果存在且不等于 0，则禁止设备管理器显示该类设备。

class 键还可以指定 **DeviceCharacteristics**、**DeviceType**、和/或 **Security** 属性，以及某种设备属性的超越值。PnP 管理器在创建设备 PDO 时会使用这些超越值。但我怀疑某一天系统管理员也能够检测并修改这些属性。

运行应用程序

为了增加用户对硬件的体验，你可以提供一个应用程序，该应用程序在设备存在时随时可以启动。Microsoft 为静态图象照相机提供了一个专用的机制，但没有提供其它设备可以使用的通用机制。在这里我将描述一个通用机制，称为 AutoLaunch。

AutoLaunch服务

Windows 98 和 Windows 2000 都可以在硬件事件发生时通知应用程序。Windows 95 引入了 WM_DEVICECHANGE 消息。作为 Windows 95 的一个原始设想，系统在用户模式中为几种可能的设备事件向所有顶级窗口广播这个消息。

基于 WM_DEVICECHANGE 消息，当设备驱动程序允许或禁止一个寄存的设备接口时，Windows 2000 就向关注的服务应用程序生成通知。我的 AutoLaunch 服务就是利用了这些通知。通过调用一个新的用户模式 API 服务 **RegisterDeviceNotification**，该服务可以预定某接口 GUID 的通知。

```
#include <dbt.h>

DEV_BROADCAST_DEVICEINTERFACE filter = {0};
filter.dbcc_size = sizeof(filter);
filter.dbcc_devicetype = DBT_DEVTYP_DEVICEINTERFACE;
filter.dbcc_classguid = GUID_AUTOLAUNCH_NOTIFY;

HDEVNOTIFY hNotification = RegisterDeviceNotification(hService,
    (PVOID) &filter,
    DEVICE_NOTIFY_SERVICE_HANDLE);
```

为了接收这个接口通知，该服务必须在其 **ServiceMain** 函数中调用 **RegisterServiceCtrlHandlerEx** 来初始化，而不是调用 **RegisterServiceCtrlHandler**。

```
hService = RegisterServiceCtrlHandlerEx(<svcname>, HandlerEx, <context>);
```

调用 **RegisterServiceCtrlHandlerEx** 时，你需要指定一个 **HandlerEx** 事件句柄函数，该函数有三个以上的参数，而不是调用标准的 **Handler** 服务函数：

```
DWORD __stdcall HandlerEx(DWORD ctlcode, DWORD evtype, PVOID evdata, PVOID context)
{}
```

在这种情况下，**ctlcode** 将等于 SERVICE_CONTROL_DEVICEEVENT，**evtype** 将等于 DBT_DEVICEARRIVAL，**evdata** 将是一个设备接口广播结构的地址。**context** 将作为 RegisterServiceCtrlHandlerEx 的第三个参数。

设备接口广播结构应该象这样：

```
struct _DEV_BROADCAST_DEVICEINTERFACE_W {
    DWORD dbcc_size;
    DWORD dbcc_devicetype;
    DWORD dbcc_reserved;
    GUID dbcc_classguid;
    WCHAR dbcc_name[1];
};
```

dbcc_devicetype 值为 DBT_DEVTYP_DEVICEINTERFACE。**dbcc_classguid** 是一个 128 位的接口 GUID，**dbcc_name** 是用于打开设备句柄的符号连接名。这个特别结构有两种版本：ANSI 和 Unicode。服务通知总是使用 Unicode 版本，而 AutoLaunch 使用 ANSI 版本。

触发AutoLaunch

为了向 AutoLaunch 触发一个设备接口到来通知，驱动程序必须用 AutoLaunch GUID 寄存并允许一个接口。

```
typedef struct _DEVICE_EXTENSION {
```

```

...
UNICODE_STRING AutoLaunchInterfaceName;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

NTSTATUS AddDevice(...)
{
    ...
IoRegisterDeviceInterface(pdo, &GUID_AUTOLAUNCH_NOTIFY, NULL,
&pdx->AutoLaunchInterfaceName);
    ...
}

NTSTATUS StartDevice(PDEVICE_OBJECT fdo, ...)
{
    ...
IoSetDeviceInterfaceState(&pdx->AutoLaunchInterfaceName, TRUE);
    ...
}

```

第二章讨论的设备接口是作为命名一个设备的方法，利用它应用程序可以找到设备并打开其句柄。一个设备也可以寄存多个接口。在这种情况下，你可以在所支持的接口之外再寄存一个 AutoLaunch 接口，AutoLaunch 接口的唯一目的就是为等待的服务生成通知。

当驱动程序允许其 GUID_AUTOLAUNCH_NOTIFY 接口时，系统向 AutoLaunch 服务发送一个设备到来通知，服务用下面函数处理该通知：

```

DWORD CAutoLaunch::HandleDeviceChange(DWORD evtype, _DEV_BROADCAST_HEADER* dbhdr)
{
    if (!dbhdr || evtype != DBT_DEVICEARRIVAL || dbhdr->dbcd_devicetype !=
DBT_DEVTYP_DEVICEINTERFACE)
        return 0;
    PDEV_BROADCAST_DEVICEINTERFACE_W p = (PDEV_BROADCAST_DEVICEINTERFACE_W)
dbhdr;
    CString devname = p->dbcc_name;
    HDEVINFO info = SetupDiCreateDeviceInfoList(NULL, NULL);
    SP_DEVICE_INTERFACE_DATA ifdata = { sizeof(SP_DEVICE_INTERFACE_DATA) };
    SP_DEVINFO_DATA devdata = { sizeof(SP_DEVINFO_DATA) };
    SetupDiOpenDeviceInterface(info, devname, 0, &ifdata);
    SetupDiGetDeviceInterfaceDetail(info, &ifdata, NULL, 0, NULL, &devdata);
    OnNewDevice(devname, info, &devdata);
    SetupDiDestroyDeviceInfoList(info);
    return 0;
}

```

1. 除了我们感兴趣的通知外，这里还有其它通知。其中一些是查询。对于某些不需要明确处理的查询我们返回 0。实际上，真正的 AUTOLAUNCH 例子还处理 DBT_DEVICEREMOVECOMPLETE 通知，以便记录哪一个是处理过的通知并在系统启动时避免重复处理。为了避免混乱我去掉了这个细节。
2. 我生成了非 UNICODE 的 AutoLaunch 例子。所以这个语句把通知结构中的 Unicode 连接名转换成 ANSI 形式。

我的 **OnNewDevice** 函数会生成一个新进程，以执行该函数在注册表中找到的任何命令行命令。使用设备的硬件键作为命令行的保存库特别方便，下面就是实现代码：

```

void CAutoLaunch::OnNewDevice(const CString& devname, HDEVINFO info, PSP_DEVINFO_DATA
devdata)
{

```

```

HKEY hkey = SetupDiOpenDevRegKey(info, devdata, DICS_FLAG_GLOBAL, 0, DIREG_DEV,
KEY_READ);    <--1

DWORD junk;
TCHAR buffer[_MAX_PATH];
DWORD size = sizeof(buffer);
CString Command;
RegQueryValueEx(hkey, "AutoLaunch", NULL, &junk, (LPBYTE) buffer, &size);      <
Command = buffer;

CString FriendlyName;           <
SetupDiGetDeviceRegistryProperty(info,
                                devdata,
                                SPDRP_FRIENDLYNAME,
                                NULL,
                                (PBYTE) buffer,
                                sizeof(buffer),
                                NULL);          <

FriendlyName.Format(_T("\'%s\'"), buffer);

RegCloseKey(hkey);

ExpandEnvironmentStrings(Command, buffer, arraysize(buffer));           <
CString name;
name.Format(_T("\'%s\'"), (LPCTSTR) devname);
Command.Format(buffer, (LPCTSTR) name, (LPCTSTR) FriendlyName);        <

STARTUPINFO si = { sizeof(STARTUPINFO) };
si.lpDesktop = "WinSta0\\Default";
si.wShowWindow = SW_SHOW;

PROCESS_INFORMATION pi;
CreateProcess(NULL, (LPTSTR) (LPCTSTR) Command, NULL, NULL, FALSE, 0, NULL, NULL, &si,
&pi);    <--7
CloseHandle(pi.hProcess);           <
CloseHandle(pi.hThread);          <
}

}

```

1. 该语句打开设备硬件键中的 **Device Parameters** 子键。
2. INF 文件把一个 **AutoLaunch** 值放到注册表中。我们在这里读取该值。
3. 在这里我们提取设备的 **FriendlyName**，并作为命令行参数使用。名中可能会有些空格，因此我们先用引号引上这个名，然后再提交该命令。
4. 我想让注册表的命令行模板中包含环境变量。该语句扩展环境串。
5. 我还想让命令行模板使用 **%s** 转义码，以指出设备名和友好名的出处。该语句产生一个含有代替符号的命令行。
6. 我们要调用 **CreateProcess** 来执行该命令。除非我们小心，否则该命令将使用相同的隐含桌面作为服务进程，这对用户不会有太大用处。因此我们创建一个 **STARTUPINFO** 结构，该结构指定一个交互桌面。
7. 我们在这里运行该应用程序。**CreateProcess** 将很快返回；应用程序一直运行直到有人关闭它。
8. **CreateProcess** 还返回该进程的句柄和其初始线程的句柄。我们需要关闭这些句柄。

鸡和蛋的问题

我刚描述的进程在一个不变的情况下会工作得很好，这种情况是指当设备出现并试图运行一个专用应用程序时，**AutoLaunch** 服务已经运行在计算机上。但另两个情况仍需要应付。

第一情况是指系统刚进入引导程序，已经安装的设备在服务管理器启动 AutoLaunch 服务之前要寄存它们的 GUID_AUTOLAUNCH_NOTIFY 接口。然而，你仍希望 AutoLaunch 应用程序启动。

AutoLaunch 这样处理这个启动问题，当它第一次启动时将枚举该接口的所有实例：

```
VOID CAutoLaunch::EnumerateExistingDevices(const GUID* guid)
{
    HDEVINFO info = SetupDiGetClassDevs(guid, NULL, NULL, DIGCF_PRESENT | DIGCF_INTERFACEDEVICE);
    SP_INTERFACE_DEVICE_DATA ifdata;
    ifdata.cbSize = sizeof(ifdata);
    DWORD devindex;
    for (devindex = 0; SetupDiEnumDeviceInterfaces(info, NULL, guid, devindex, &ifdata); ++devindex)
    {
        DWORD needed;
        SetupDiGetDeviceInterfaceDetail(info, &ifdata, NULL, 0, &needed, NULL);
        PSP_INTERFACE_DEVICE_DETAIL_DATA detail = (PSP_INTERFACE_DEVICE_DETAIL_DATA)malloc(needed);
        detail->cbSize = sizeof(SP_INTERFACE_DEVICE_DETAIL_DATA);
        SP_DEVINFO_DATA devdata = { sizeof(SP_DEVINFO_DATA) };
        SetupDiGetDeviceInterfaceDetail(info, &ifdata, detail, needed, NULL, &devdata);
        CString devname = detail->DevicePath;
        free((PVOID) detail);
        OnNewDevice(devname, guid);
    }
}
```

这个函数中的重要代码是粗体显示部分，这些代码用于获得必要的 SP_DEVINFO_DATA 结构和符号连接名。然后我们调用 **OnNewDevice** 来处理这个预先存在的设备。

运行服务

第二个启动情况是当设备第一次被安装到计算机上时，而该计算机从没有见过 AutoLaunch 服务。你的 INF 文件需要定义 AutoLaunch 服务并复制该服务的二进制文件到用户的计算机。还要向注册表加入一个名为 **RunOnce** 的键以触发该服务。例如：

```
[DestinationDirs]
AutoLaunchCopyFiles=10
[etc.]


[DriverInstall.NT]
CopyFiles=DriverCopyFiles,AutoLaunchCopyFiles
AddReg=DriverAddReg.NT


[DriverAddReg.NT]
HKLM,%RUNONCEKEYNAME%,AutoLaunchStart,,
    "rundll32 StartService,StartService AutoLaunch"


[DriverInstall.NT.Services]
AddService=AutoLaunch,,AutoLaunchService
[etc.]


[AutoLaunchCopyFiles]
AutoLaunch.exe,,,0x60
StartService.exe,,,0x60
```

```
[AutoLaunchService]
ServiceType=16
StartType=2
DisplayName="AutoLaunch Service"
ErrorControl=1
ServiceBinary=%10%\AutoLaunch.exe

[Strings]
RUNONCEKEYNAME="Software\Microsoft\Windows\CurrentVersion\RunOnce"
```

全部代码见 AUTOLAUNCH 例子的 SYS 目录中的 DEVICE.INF 文件。

设备安装完成后，系统将执行 RunOnce 键中的任何命令。我们放在那里的命令就是启动 AutoLaunch 服务。注意 STARTSERVICE.DLL 是一个很小的 DLL，用于启动一个服务，它不显示任何用户接口信息或弹出对话框。如果你需要远程安装驱动程序包，你应该用 RUNDLL32 来替换 RunOnce 值中的命令动词。

注意

Microsoft 的 KBase 文章第 Q173039 暗示 RunOnce 键中有立即处理行为的表项在调用 RUNDLL32 时会产生侧效。但一个负责设备安装程序的 Microsoft 开发员向我保证，RunOnce 值在安装一个新设备时总是被处理。别管该文章中的暗示。

Windows 98 兼容问题

在设备安装与维护方面，Windows 98 使用了与 Windows 2000 完全不同的技术。本节我将介绍某些有影响的因素。

属性页提供程序

新设备类的属性页提供程序必须是一个 16 位的 DLL。因此先不要丢弃你的 16 位编译器。

注册表用法

Windows 98 使用 software 注册表键来定位设备驱动程序。为了初始化这个键，你的 Windows 98 install 段应该有一个这样的 AddReg 指令：

```
[DriverInstall]
AddReg=DriverAddReg
<other install directives>

[DriverAddReg]
HKR,,DevLoader,*ntkern
HKR,,NTMPDriver,pktdma.sys
```

即指定 NTKERN.VXD 作为设备载入程序，指定你的 WDM 驱动程序作为 NTKERN 寻找的 NTMPDriver。另外，你应该忽略 Services 段，Windows 98 并不使用它。

相对于 Windows 2000, Windows 98 把标准和非标准的设备属性都放到了硬件键中, 而不是把非标准属性分开放到 **Device Parameters** 子键中。这也许会更方便, 因为你不必使用 **IoGetDeviceProperty** 来获取标准属性。

获得设备属性

Windows 98(包括 Windows 98 第二版)用于读取设备硬件键中标准属性的函数 **IoGetDeviceProperty** 存在错误。为了在 WDM 驱动程序中获取这些属性, 你应该使用 **IoOpenDeviceRegistryKey** 函数并用名字查询属性。**DEVPROP** 例子显示了如何用这种方法获得标准设备描述属性。

应用程序执行

Windows 98 没有服务管理器, 因此你不能把 AutoLaunch 作为服务运行。但可以使用注册表中的 **Run** 关键字。这个 AutoLaunch 包的一部分是 ALNCH98.EXE 小程序(applet), 它可以以这种方式执行。使用它的任务条图标可以停止该程序。

附录A: Windows 98 不兼容处理

本书的许多章都是以讨论 Windows 98 的兼容问题为结尾。Microsoft 最初计划让 Windows 98 和 Windows 2000 使用同一种驱动程序二进制文件，但事实证明这个目标太高。不要惊讶，Windows 2000 会继续发展，它可以支持一些 Windows 98 不支持的内核模式服务函数。如果一个 WDM 驱动程序调用了这种函数，Windows 98 将拒绝装入该驱动程序，因为它不能解决其中的符号引用。在这个附录中，我将描述一个静态虚拟设备驱动程序(VxD)，即光盘上的 WDMSTUB 例子，它解决了其中一些符号问题。一旦你可以装入含有 Windows 98 不支持函数调用的驱动程序，你会发现在运行时确定所在操作系统是 Windows 98 还是 Windows 2000 是十分必要的；我会描述做这个判断的一个方法。

- 为内核模式例程定义桩
- 确定操作系统版本

为内核模式例程定义桩

WDMSTUB.VXD 使用的桩技术与 Microsoft 移植数百个 Windows NT 内核模式支持函数到 Windows 98 环境中所使用的技术类似，即扩充运行时间装入器在解决输入引用时使用的符号表。为了扩充这个符号表，你需要先定义三个驻留内存的数据表：

- 给出你定义函数名称的名称表
- 给出这些函数地址的地址表
- 关联函数名称与函数地址的顺序表

这里是 WDMSTUB 中的一些表项：

```
static char* names[] = {
    "PoRegisterSystemState",
    ...
    "ExSystemTimeToLocalTime",
    ...
};

static WORD ordinals[] = {
    0,
    ...,
    6,
    ...
};

static PFN addresses[] = {
    (PFN) PoRegisterSystemState,
    ...
    (PFN) ExSystemTimeToLocalTime,
    ...
};
```

顺序表的用途是为一个给定 **names** 表项提供 **address** 中的索引。即名为 **names[i]** 的函数地址为 **address[ordinals[i]]**。

如果不是应付版本兼容问题，你可以调用 **_PELDR_AddExportTable** 函数：

```
HPEEXPORTTABLE hExportTable = 0;

extern "C" BOOL OnDeviceInit(DWORD dwRefData)
{
    _PELDR_AddExportTable(&hExportTable,
        "ntoskrnl.exe",
        arraysize(addresses), // don't do it this way!
        arraysize(names),
        0,
        (PVOID*) names,
        ordinals,
        addresses,
        NULL);

    return TRUE;
}
```

`_PELDR_AddExportTable` 调用扩展了载入程序使用的符号表。`NTKERN.VXD` 是 Windows 98 中支持 WDM 驱动程序的主要模块，它用其支持的几百个函数的地址来初始化这个表。`WDMSTUB.VXD` 是一个静态 VxD，它的初始化顺序在 `NTKERN` 之后，但在 Windows 98 配置管理器之前。因此，`WDMSTUB` 的输出定义在系统装入任何 WDM 驱动程序时已被确定。结果，`WDMSTUB` 扩展了 `NTKERN`。

版本兼容

版本兼容问题是这样的：Windows 98 所支持的能够被 WDM 驱动程序使用的函数是 Windows 2000 的一个子集。Windows 98 第二版支持一个更大的子集。下一版本的 Windows，代码名为 Millennium，将支持更大的子集(甚至是超集，因为它在 Windows 2000 之后发布)。你不会希望 VxD 桩复制任何 OS 支持的函数。`WDMSTUB` 的实际工作是在系统初始化过程中展开的，是动态地构造一个表，然后传递给 `_PELDR_AddExportTable`：

```
HPEEXPORTTABLE hExportTable = 0;

extern "C" BOOL OnDeviceInit(DWORD dwRefData)
{
    char** stubnames = (char**) _HeapAllocate(sizeof(names), HEAPZEROINIT);
    PFN* stubaddresses = (PFN*) _HeapAllocate(sizeof(addresses), HEAPZEROINIT);
    WORD* ordinals = (WORD*) _HeapAllocate(arraysize(names) * sizeof(WORD), HEAPZEROINIT);
    int i, istub;
    for (i = 0, istub = 0; i < arraysizes(names); ++i)
    {
        if (_PELDR_GetProcAddress((HPEMODULE) "ntoskrnl.exe", names[i], NULL) == 0)
        {
            stubnames[istub] = names[i];
            ordinals[istub] = istub;
            stubaddresses[istub] = addresses[i];
            ++istub;
        }
    }
    _PELDR_AddExportTable(&hExportTable,
                         "ntoskrnl.exe",
                         istub, istub,
                         0,
                         (PVOID*) stubnames,
                         ordinals,
                         stubaddresses,
                         NULL);
    return TRUE;
}
```

粗体部分是这里的关键步骤，它确保我们不会替换 `NTKERN` 中或其它系统 VxD 中已存在的函数。

在解决版本兼容问题上还有一个小问题。Windows 98 第二版仅输出了管理 `IO_REMOVE_LOCK` 对象的四个支持函数中的三个。另外一个函数是 `IoRemoveLockAndWaitEx`。`WDMSTUB.VXD` 驱动程序将补偿这个遗漏，或者支持所有四个函数，或者一个也不支持，取决于系统中是否有这个函数。

桩函数

`WDMSTUB.VXD` 的主要目的是解决驱动程序会引用但没有实际调用的符号。对于某些函数，如 `PoRegisterSystemState`，`WDMSTUB.VXD` 仅简单地包含一个桩，如果调用会返回一个错误指示：

```
PVOID PoRegisterSystemState(PVOID hstate, ULONG flags)
{
```

```

    ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
    return NULL;
}

```

创建 WDMSTUB

为了正确创建 WDMSTUB，我需要合并两个非标准特征。每个桩函数必须使用 `__stdcall` 调用约定，而 VxD 通常使用 `__cdecl`。

我希望从 VxD 桩中调用 `KeGetCurrentIrql` 函数和其它 WDM 服务函数。标准做法是在所有 VxD 头文件前面包含 `WDM.H` 或 `NTDDK.H`，并连接 `WDMVXD.LIB` 输入库。`WDMVXD.LIB` 假定你要输入的函数以 `__declspec(dllexport)` 指示声明，当你包含了 `WDM.H` 或 `NTDDK.H` 头文件时这通常是对的。因为它们都用一个预处理宏 `NTKERNELAPI` 来声明，而该宏通常定义为 `__declspec(dllexport)`。不幸的是，如果你定义一个标记为 `dllimport` 的函数，编译器会假定你要输出这个函数。一个 VxD 的第一个输出必须是定义该驱动程序的设备描述块(DDB)，不是某个随机输出的桩函数。我猜想在模块定义文件中指定 DDB 的顺序号为 1 会强制 DDB 成为第一个输出，但我错了。我最后得到一个不能载入的 VxD。

为了越过这些输入输出声明问题，我不得不强迫 `NTDDK.H` 不以正常方式定义 `NTKERNELAPI`。(见 `WDMSTUB` 工程中的 `STDVXD.H` 文件) 由于 `WDMVXD.LIB` 词表的限制，我不得不使用象 `_KeGetCurrentIrql@0` 这样的符号引用。在 `KeGetCurrentIrql` 的特殊情况下，驱动程序可以发出一个标准 `VxDCall`，调用名为 `ObsoleteKeGetCurrentIrql` 的服务并到达 Windows 98 内核中的正确函数。另外，你也可以定义一个(如 `MyGetCurrentIrql`)调用 `KeGetCurrentIrql` 的函数，然后把它放到源模块中，并用 `NTKERNELAPI` 的正常设置编译该模块。

虽然有时你并不需要写一个失败该函数的函数桩，实际上你可以实现该函数，如下例：

```

VOID ExLocalTimeToSystemTime(PLARGE_INTEGER localtime, PLARGE_INTEGER systime)
{
    systime->QuadPart = localtime->QuadPart + GetZoneBias();
}

```

`GetZoneBias` 是一个辅助例程，通过查询注册表 `TimeZoneInformation` 键中的 `ActiveTimeBias` 值来确定时区偏差。

表 A-1 列出了 `WDMSTUB.VXD` 输出的内核模式支持函数。

表 A-1. `WDMSTUB.VXD` 输出的函数

支持函数	备注
<code>ExLocalTimeToSystemTime</code>	实现
<code>ExSystemTimeToLocalTime</code>	实现
<code>IoAcquireRemoveLockEx</code>	实现
<code>IoAllocateWorkItem</code>	实现
<code>IoFreeWorkItem</code>	实现
<code>IoInitializeRemoveLockEx</code>	实现
<code>IoQueueWorkItem</code>	实现
<code>IoReleaseRemoveLockEx</code>	实现
<code>IoReleaseRemoveLockAndWaitEx</code>	实现
<code>IoCreateNotificationEvent</code>	失败桩
<code>IoCreateSynchronizationEvent</code>	失败桩
<code>IoReportTargetDeviceChangeAsynchronous</code>	失败桩

KdDebuggerEnabled	实现
KeEnterCriticalSection	实现
KeLeaveCriticalSection	实现
KeNumberProcessors	返回 1
KeSetTargetProcessorDpc	实现
PoRegisterSystemState	失败桩
PoSetSystemState	失败桩
PoUnregisterSystemState	失败桩
PsGetVersion	实现
RtlInt64ToString	失败桩
RtlUlongByteSwap	实现
RtlUlonglongByteSwap	实现
RtlUshortByteSwap	实现

确定操作系统版本

驱动程序一旦装入，就需要布置某些 Windows 98 支持例程桩，这需要在运行时确定当前操作系统的版本。例如，你要调用一个非 WDM(严格地讲)函数。**IoReportTargetDeviceChangeAsynchronous** 就是这样的函数，我在 PNPEVENT 例子中使用过该函数。

通过调用 **GetVersionEx** 函数，应用程序可以很容易了解操作系统。而最类似的内核模式函数是 **IoIsWdmVersionAvailable**：

```
BOOLEAN IoIsWdmVersionAvailable(MajorVersion, MinorVersion);
```

Windows 2000 支持 WDM 版本 1.10，即 WDM.H 中含有 **WDM_MAJORVERSION(0x01)** 和 **WDM_MINORVERSION(0x10)** 常量。Windows 98(包括第二版)仅支持 WDM1.0。你可以利用这个支持级别的不同来判断所在的操作系统。

探测操作系统版本

我过去依靠不同的探测方法来确定操作系统的版本，但随着操作系统的改进这些方法不能使用了。例如，在 Windows 98 的原始发行版中，当系统以正常方式调用你的 **DriverEntry** 时，驱动程序的 **DriverExtension** 中的 **ServiceKeyName** 长度为 0。但在 Windows 2000 中，该参数是一个非空串。因此后来版本的 Windows 98 也是这样，这可以用于探测原始版本的 Windows 98。

Vireo 软件公司建议使用 \Registry\Machine\SAM 键的存在作为判断 Windows 2000 的标准。但这种方法对于在 Windows 2000 启动时装入的驱动程序不可行，所以也不能依靠这种测试。该公司现在推荐测试 **HKLM\System\CurrentControlSet\Control\Class** 键，该键只存在于 Windows 2000 种，而 **HKLM\System\CurrentControlSet\Services\Class** 键仅存在于 Windows 98 种。

附录B：使用**GENERIC.SYS**

本附录解释了支持库 **GENERIC.SYS** 的公共接口，本书中的大部分例子驱动程序都使用这个库。首先我需要解释关于 **GENERIC** 的一些事情。

我创建 **GENERIC** 的原因是因为在写本书时我需要不断地修改例子驱动程序的 PnP 和电源管理支持。本书出版后我仍需要修改这个支持。为了避免在每次了解到新的 PnP 和电源管理特征时修改 20 多个例子驱动程序，我决定创建 **GENERIC** 并让它处理所有的 IRP_MJ_PNP 和 IRP_MJ_POWER 请求。

我使 WDMWIZ.AWX(下一附录中介绍)和 **GENERIC** 保持同步。即，如果你使用 WDMWIZ 创建一个驱动程序，那么不管你有没有使用 **GENERIC** 都会得到同样的结果。如果你决定使用 **GENERIC**，你的驱动程序可以调用 **GENERIC** 来处理 WDM 驱动程序需要做的一些复杂事情。如果你不使用它，你的驱动程序将包含所有那些代码。

最后，我使用 Microsoft 的 AUTODUCK 工具自动生成了 **GENERIC** 输出函数的文档。AUTODUCK 为源代码加上特殊格式的注释，并把它们转换为文档。如果你能记得更新注释，就可以容易地更新文档。该文档是位于 **GENERIC** 目录中的 **GENERIC.RTF** 文件。

本书的大部分例子程序需要依靠 **GENERIC.SYS** 来处理某些极其复杂的 PnP 和电源管理操作。当读到第六章和第八章时，你可能想了解例子程序是如何与 **GENERIC** 联系起来的，见下面：

- 例子程序中的 **AddDevice** 函数调用 **InitializeGenericExtension** 例程，并传递一个结构，该结构含有处理 PnP 和电源管理请求的函数指针。
- IRP_MJ_PNP 和 IRP_MJ_POWER 的派遣例程把这些 IRP 委托给 **GENERIC** 来处理。
- **GENERIC** 在适当的地方回调进入驱动程序。
- **GENERIC** 使用的各种 API 和回调函数参见 **GENERIC** 目录中的 **GENERIC.RTF** 文件。

表 AB-1. **GENERIC** 使用的驱动程序回调例程

IRP 类型	回调函数	目的
IRP_MJ_PNP	StartDevice	启动设备(映射内存寄存器、连接中断，等等)
	StopDevice	停止设备并释放 I/O 资源(解除内存寄存器映射、断开中断，等等)
	RemoveDevice	执行与 AddDevice 中执行的相反的步骤(断开低层设备对象、删除设备对象，等等)
	OkayToStop	(可选) 现在能否停止该设备？(用于处理 IRP_MN_QUERY_STOP_DEVICE 时)
	OkayToRemove	(可选) 现在能否删除该设备？(用于处理 IRP_MN_QUERY_REMOVE_DEVICE 时)
IRP_MJ_POWER	QueryPower	(可选) 设备能进入这个电源状态吗？(用于处理 IRP_MN_QUERY_POWER 时)
	SaveDeviceContext	(可选) 保存任何可能在低电源状态期间丢失的设备上下文
	RestoreDeviceContext	(可选) 低电源状态结束后恢复设备上下文

附录C：使用WDMWIZ.AWX

本附录描述了如何使用 WDMWIZ.AWX 向导创建能被 Visual C++ 6.0 使用的驱动程序工程。我写这个向导的目的是想更方便地为本书生成例子驱动程序。我想你或许能利用这个向导。

光盘上的 WDMBOOK.HTM 文件描述了如何把这个向导装入你的系统。一旦装入后，你会发现 Visual C++ 在创建新工程时，New 对话框中多了一个 WDM Driver Wizard 工程标签。

WDMWIZ.AWX 并不是一个产品。因此我没有更改其用户界面的计划。另外，我对使用 WDMWIZ.AWX 完成的驱动程序不做任何品质保证。

- 基本驱动程序信息
- DeviceIoControl 代码
- I/O 资源
- 电源管理能力
- USB 端点
- WMI 支持
- INF 文件参数
- 注意事项

基本驱动程序信息

开始对话框(见图 AC-1)要求你给出被创建驱动程序的基本信息。

对于驱动程序的类型，你有这些选择：

- **通用功能驱动程序** 为一个通用设备创建功能驱动程序。
- **通用过滤器驱动程序** 创建一个默认处理所有类型 IRP 的过滤器驱动程序。
- **USB 功能驱动程序** 创建一个 USB 设备的功能驱动程序。
- **空驱动程序工程** 创建一个无任何文件的工程，但设置选项已为 WDM 驱动程序设置好。

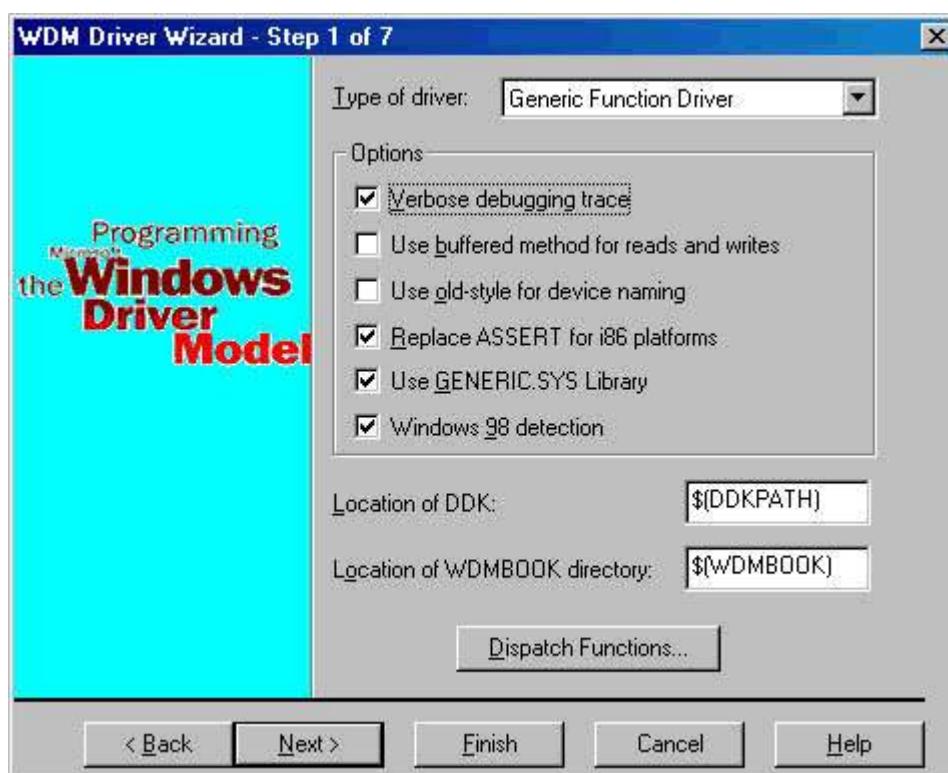


图 AC-1. 输入基本驱动程序信息的对话框

你可以选择下面选项：

- **Verbose Debugging Trace** 如果你选择该项，驱动程序工程文件将包含许多 **KdPrint** 宏调用，这可以跟踪驱动程序中的重要操作。
- **Use Buffered Method For Reads And Writes** 如果你用 DO_BUFFERED_IO 方式来执行读写操作则设置这项。清除这项表明你希望使用 DO_DIRECT_IO 方法。
- **Use Old-Style For Device Naming** 设置该项将生成命名设备对象。清除这项表明生成一个使用设备接口的驱动程序。这也是 Microsoft 推荐做的。
- **Replace ASSERT For i86 Platforms** DDK 的 **ASSERT** 宏将调用内核模式支持例程(**RtlAssert**)，该例程在 free build 的 Windows 2000 中是一个空操作。因此即使是 checked build 的驱动程序也不会在 free build 的操作系统中停下来。设置这个选项将重定义 ASSERT，以便 checked build 的驱动程序能够在 free build 的操作系统中停下来。
- **Use GENERIC.SYS Library** 设置这个选项将使驱动程序使用 GENERIC.SYS 中的标准化代码。清除这个选项将使驱动程序含有所有标准化代码。
- **Windows 98 Detection** 设置这个选项将使驱动程序包含运行时间系统版本检测代码。清除该选项将使驱动程序忽略这个检测。

你还可以指定 Windows 2000 DDK 和本书例子程序的基本路径名。而默认值\$(DDKPATH)和\$(WDMBOOK)将使用本书安装程序创建的环境变量。

最后点击 **Dispatch Functions** 按钮，指定驱动程序将要处理的 IRP 类型，如图 AC-2。该对话框含有某些不可超越的设计决定。驱动程序将支持 IRP_MJ_PNP 和 IRP_MJ_POWER 请求。如果你指定要处理

IRP_MJ_CREATE，那么你也需要处理 IRP_MJ_CLOSE。如果你指定要处理 IRP_MJ_READ、IRP_MJ_WRITE、IRP_MJ_DEVICE_CONTROL，你也要支持 IRP_MJ_CREATE 以及 IRP_MJ_CLOSE。WDMWIZ.AWX 不能生成仅被文件系统驱动程序使用的 IRP 的派遣函数框架代码。

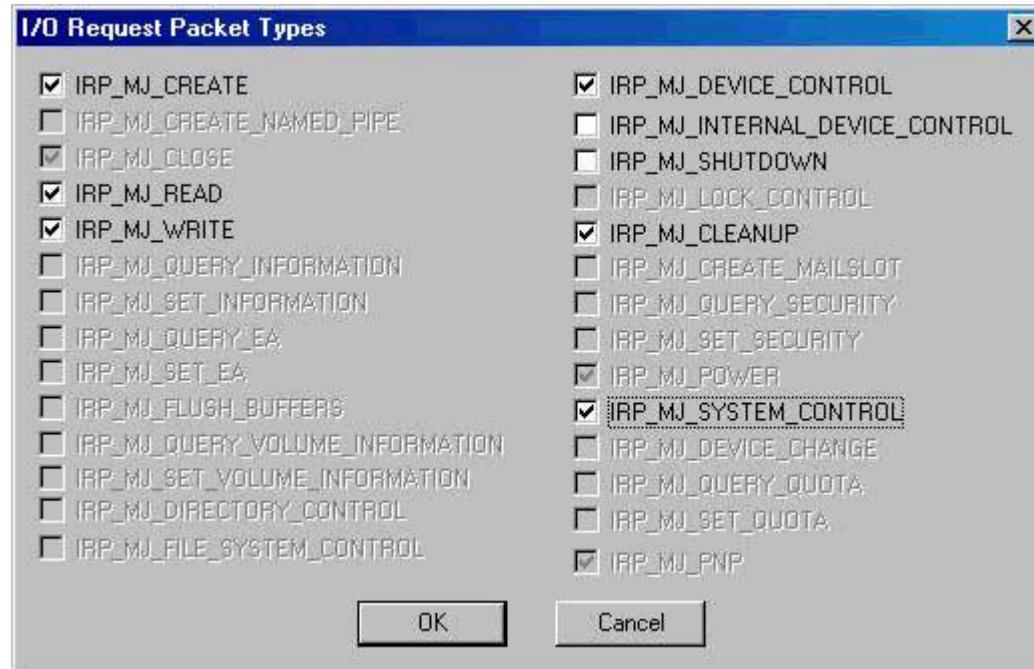


图 AC-2. 指定驱动程序能处理的 IRP 类型

DeviceIoControl 代码

如果你指定处理 IRP_MJ_DEVICE_CONTROL，向导将显示另一个对话框(见图 AC-3)，允许你指定控制操作的相关信息。

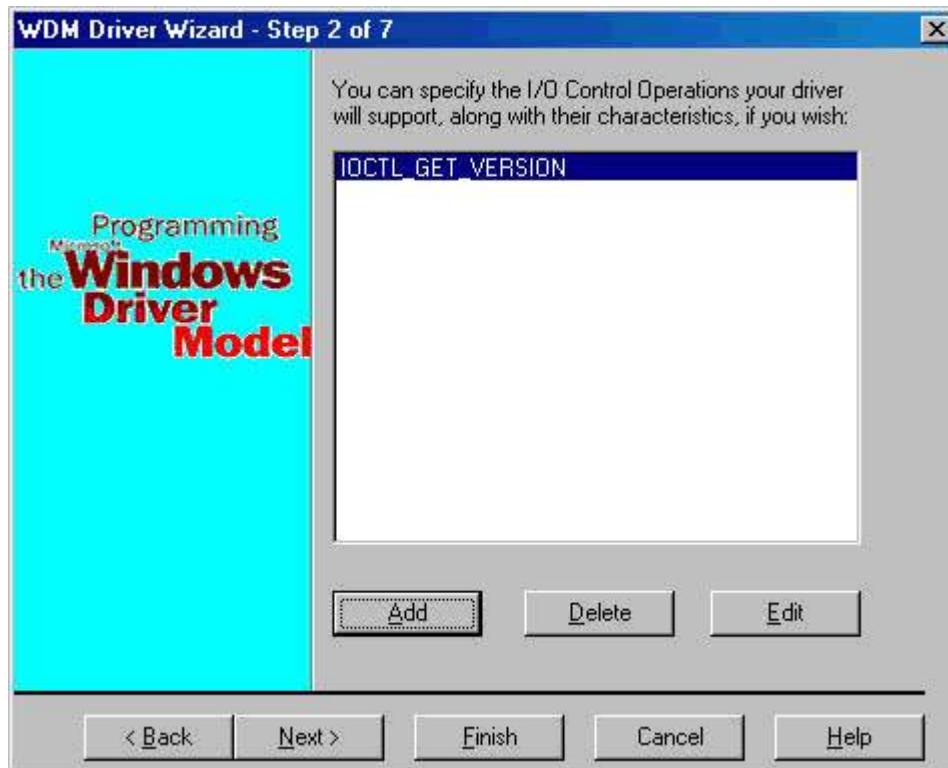


图 AC-3. 指定 I/O 控制操作的对话框

图 AC-4 是指定某 **DeviceIoControl** 操作相关信息的例子。大部分域与 CTL_CODE 宏的参数直接对应，无须解释。设置异步选项将生成对异步操作完成的支持，在派遣函数返回 STATUS_PENDING 之后。

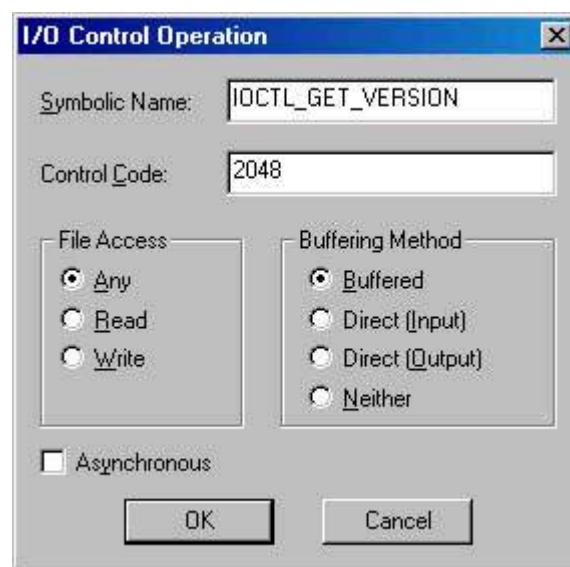


图 AC-4. 添加和编辑 I/O 控制操作的对话框

I/O资源

如果驱动程序使用 I/O 资源，你需要在第三个对话框中填入一些信息，见图 AC-5。

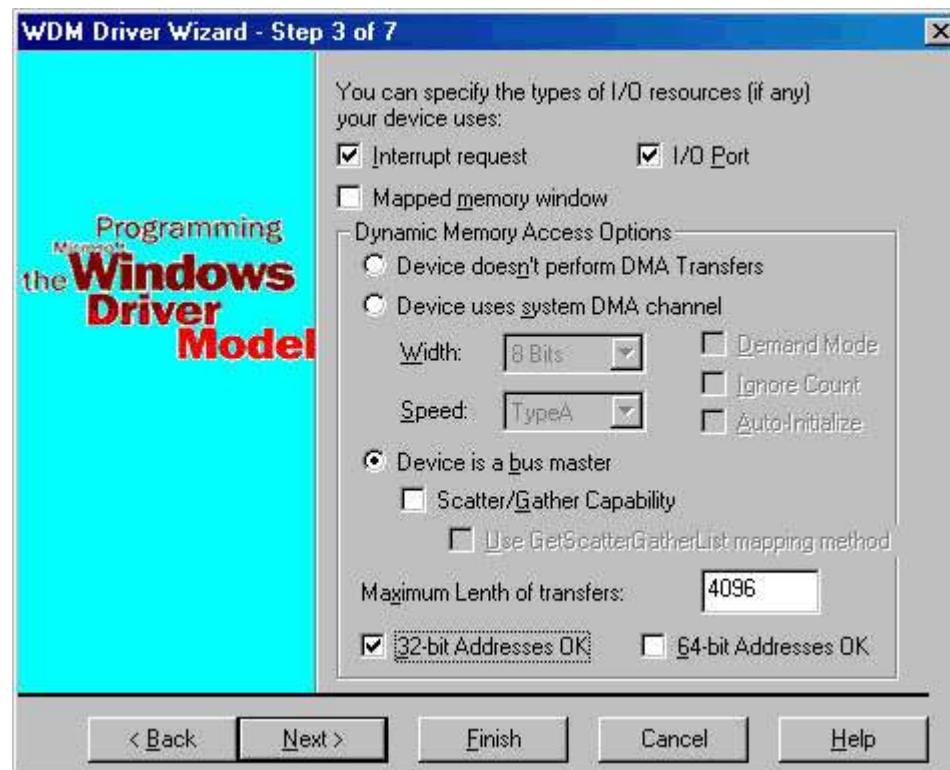


图 AC-5. 指定 I/O 资源对话框

电源管理能力

向导的第四步(如图 AC-6)指定各种电源管理能力。

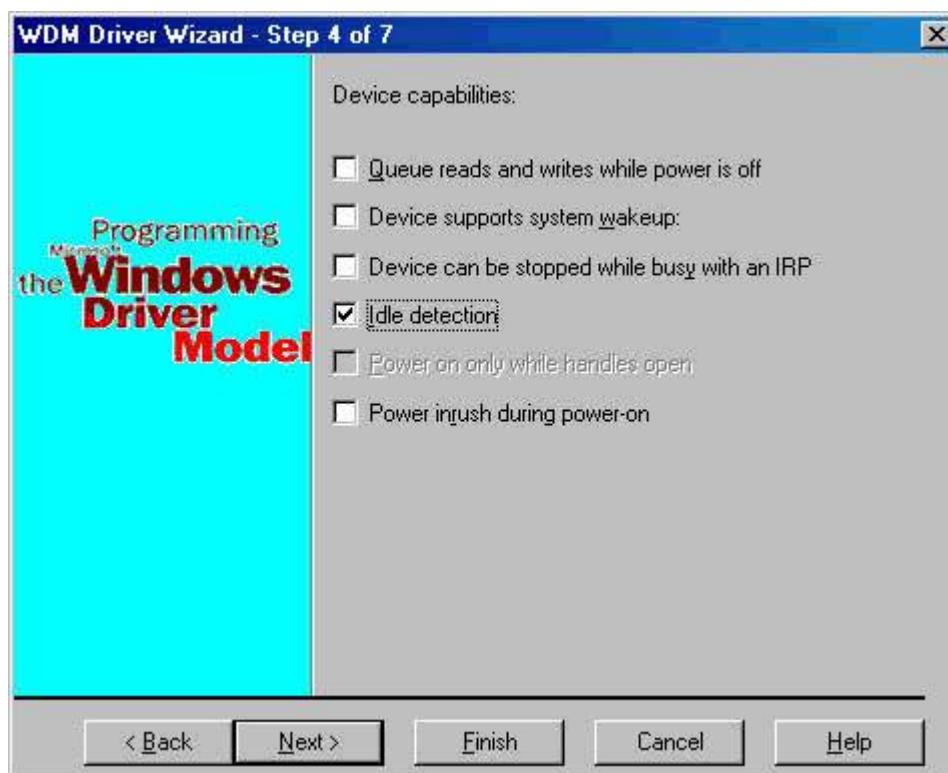


图 AC-6. 指定电源管理能力对话框

下面是该对话框中的选项：

- **Queue Reads And Writes While Power Is Off** 如果驱动程序在设备掉电后还要接收并排队读写 IRP 则设置这项。
- **Device Supports System Wakeup** 如果设备有系统唤醒能力则设置该项。否则清除该项选择。这个选项将在适当地方生成支持 IRP_MN_WAIT_WAKE 请求的框架代码。
- **Device Can Be Stopped While Busy With An IRP** 如果驱动程序能够在 IRP_MN_STOP_DEVICE 到来时停止活动的读写 IRP，则可以设置该项。否则不要选该项。
- **Idle Detection** 如果你希望设备在用户规定的不活动时间后自动掉电，则设置该项。如果驱动程序使用 GENERIC.SYS，会自动得到一组 IOCTL 支持。
- **Power On Only While Handles Open** 该选项仅对 USB 设备有效。如果你在应用程序拥有设备的句柄时希望设备一直上电则选择该项。
- **Power Inrush During Power-On** 如果设备在上电时需要大电流则选择该项。

USB端点

如果你在第一个对话框中选择 **USB** 功能驱动程序，则向导将呈现一个让你描述设备端点的对话框，见图 AC-7。该对话框列出了设备扩展中的一些变量名。名字的顺序与设备上端点描述符的顺序相对应。

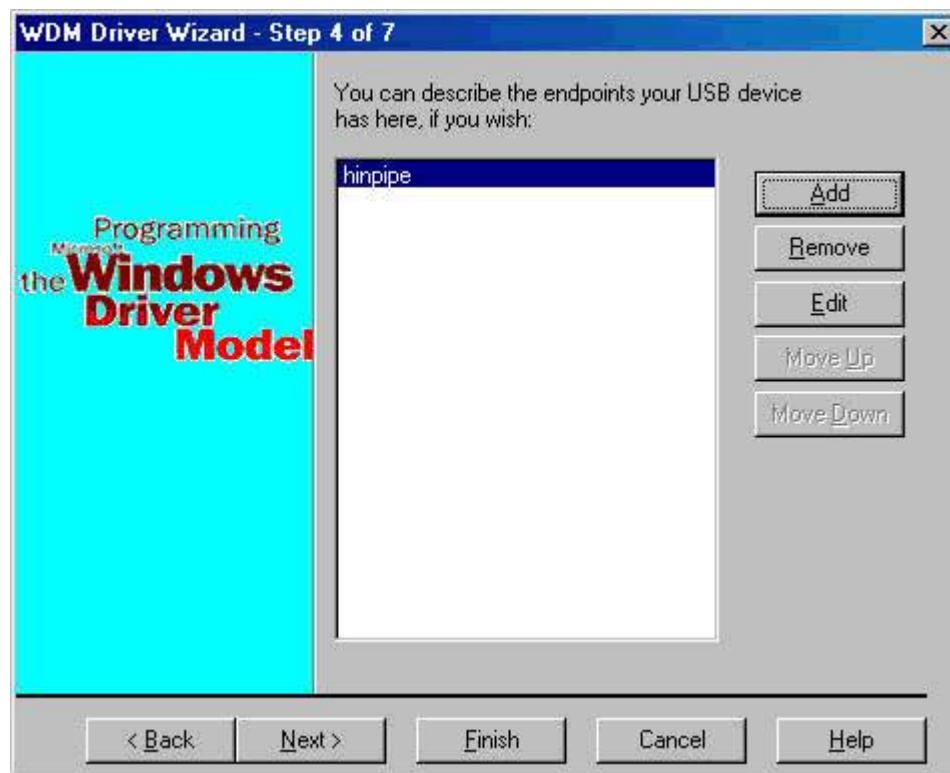


图 AC-7. 定义 **USB** 端点对话框

图 AC-8 显示了用于描述单一端点的对话框。Description Of Endpoint 组合框与设备固件中的端点描述相关，应该是自解释的。Resources In The Driver 组合框中的域如下：

- **Name Of Pipe Handle In Device Extension** 给出 DEVICE_EXTENSION 中保存管道句柄的成员名，这个管道与这个端点相关。
- **Maximum Transfer Per URB** 指出一个 URB 所能携带的最大字节数量。通常，这个值比端点最大传输值要大些。

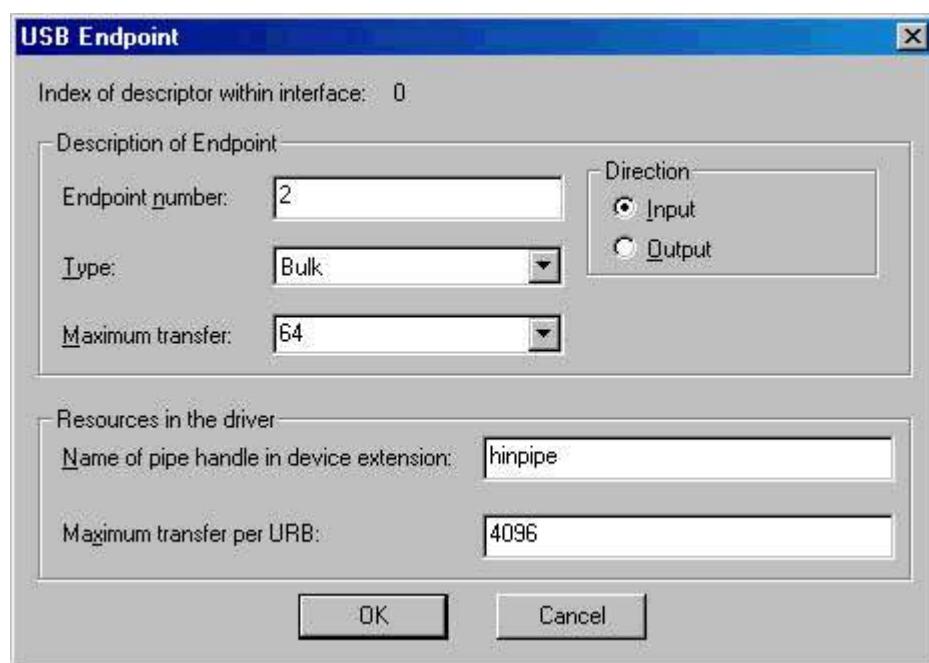


图 AC-8. 添加和编辑 USB 端点的对话框

WMI支持

如果你指出要处理 IRP_MJ_SYSTEM_CONTROL 请求，向导将呈现如图 AC-9 对话框，允许你指定定制 WMI 规划中的元素。

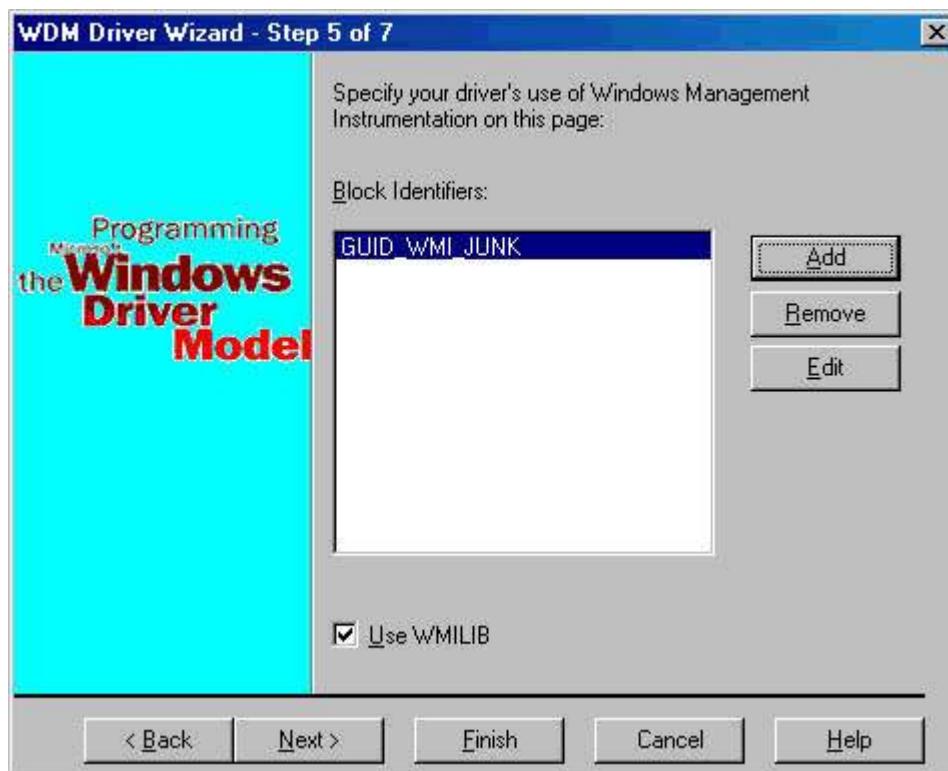


图 AC-9. 指定 WMI 选项对话框

在这个对话框中，你应该总选择 Use WMILIB 选项，因为自动生成的代码不会对处理 WMI 请求有多大帮助。Block Identifiers 列表框列出了定制规划中的类 GUID 名称。

图 AC-10 显示了如何在定制规划中描述一个类。顶级控制(无标签)是向导将要自动生成的 GUID 的符号名。你可以指定该类的下列属性：

- **Number Of Instances** 指出驱动程序将创建该类多少个实例。
- **Expensive** 指定一个 expensive 类。
- **Event Only** 指出该类仅用于发出事件。
- **Traced** 对应一个 WMI 选项。

你可以选择是使用基于 PDO 的实例命名方法还是使用带基本名的实例命名方法。Microsoft 推荐使用前者。

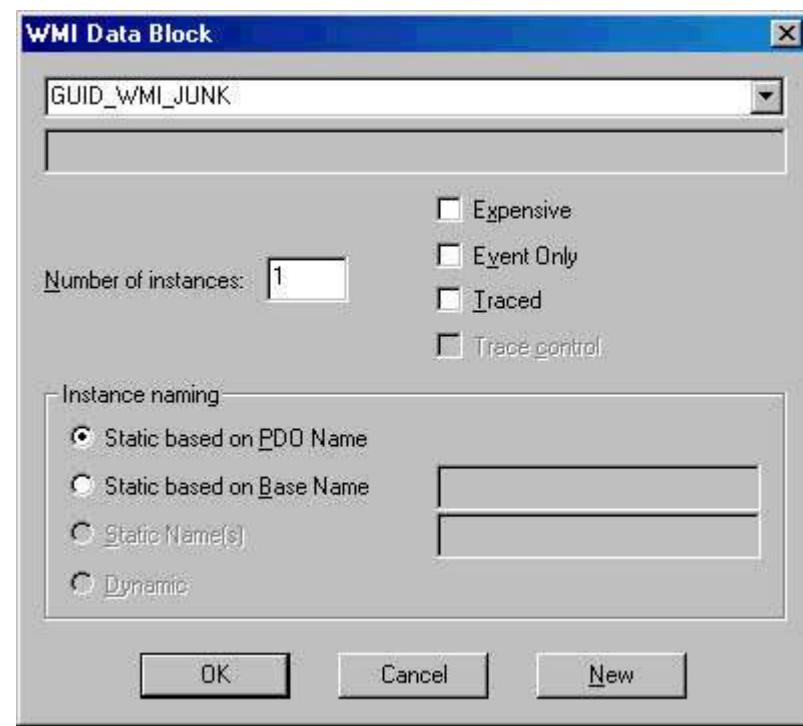


图 AC-10. 指定一个 WMI 类的对话框

INF文件参数

向导的最后一个对话框(如图 AC-11)指定 INF 文件信息。

下面是该对话框中出现的域:

- **Manufacturer Name** 硬件厂商名称。
- **Device Class** 设备所属的标准设备类。
- **Hardware ID** 设备的硬件标识符。你指定的标识符应该与总线驱动程序将创建的某个标识符相匹配。
- **Friendly Name For Device** 如果你想向设备的硬件键中插入一个 **FriendlyName** 值, 在这里指定其名称。
- **Auto-Launch Command** 如果你希望 AutoLaunch 服务在设备启动时自动运行一个应用程序, 在这里指定命令行。
- **Device Description** 在这里插入设备的描述。

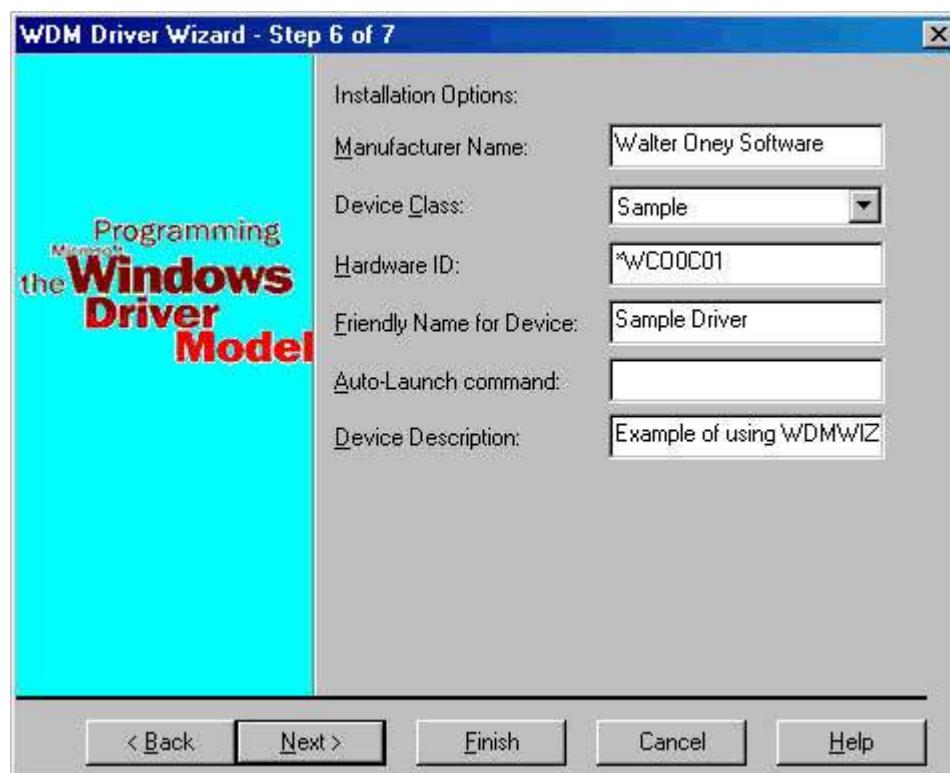


图 AC-11. 指定 INF 文件选项对话框

注意事项

运行完向导之后，你将得到一个用于创建驱动程序的工程。由于 Visual C++ 定制向导的限制，你需要手工加入某些工程设置。对这些设置的描述参见 [WDMBOOK.HTM](#)。

生成的代码中会含有许多 **TODO** 注释，在那里你可以写入代码。可以用 **Find In Files** 命令定位这些 **TODO**。

关于作者

Walter Oney

Walter Oney 是一个麻萨诸塞州波士顿的自由软件作家和顾问。1968 班一员，麻萨诸塞州技术学院电机工程学士和硕士。在不讲授程序设计时，他喜欢跑步，骑车，看芭蕾舞和演奏双簧管。

译者

mhs 马少华(mhsinet@263.net)

整理及制作

znsoft: [驱动开发网](#)站长，专业的驱动开发人员。

版权

本书版权由微软出版社所有。mhs 网友为方便大家使用而翻译。

关于电子版

这本电子版书是按原样创建的并可以作为印刷品购买。为了简便起见，本书已经和原样稍微有些不同。例如：可能参考一些例子文件但它们已经在印刷品书中出现。它们在印刷品书中是可用的，但在电子版中没有提供。(译注：你可以参看附带的文件包)



你的信息源

微软出版社是微软公司的一个分部,是最主要的全面同步学习,培训,评估和支持资料的来源,帮助从开发者到IT专家到最终用户从微软得到最多的技术。用可利用的及时正确的信息从微软制作的超过 250 种的印刷品、多媒体材料、便于学习的网络解答中挑选出来。更多的信息,请访问 <http://mspress.microsoft.com/>.

关于本**PDF**电子档

本书由Sean Wan (SeanWan@msn.com)基于原CHM文档并去掉了其中冗余的原图重新制作。