

原创经典，威盛一线工程师倾力打造

Broadview
www.broadview.com.cn

深入驱动核心，剖析操作系统底层运行机制

通过实例引导，快速学习编译、安装、调试的方法

Windows

驱动开发技术详解

Windows Driver Development Internals

张帆 史彩成 等编著

珍藏版

- 从Windows最基本的两类驱动程序的编译、安装、调试入手讲解，非常容易上手
- 用实例详细讲解PCI、USB、虚拟串口、虚拟摄像头、SDIO等驱动程序的开发
- 归纳了多种调试驱动程序的高级技巧，如用WinDbg和VMWare软件对驱动进行源码级调试
- 介绍了多种实用的工具软件，如BusHound、IRPTrace、DebugView等
- 深入Windows操作系统的底层和内核，透析Windows驱动开发的本质



CD-ROM

电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.eipress.com.cn

院士推荐

本书是作者结合教学和科研实践经验编写而成的，不仅详细介绍了Windows 内核原理，而且介绍了编程技巧和应用实例，兼顾了在校研究生和工程技术人员的实际需求，对教学、生产和科研有现实的指导意义，是一本值得推荐的专著。

——中国工程院院士

王仲

内容概述

- ◎ 驱动程序HelloDDK、HelloWDM代码分析
- ◎ 编译环境配置、安装及调试
- ◎ Windows内存管理
- ◎ 派遣函数
- ◎ IRP的同步
- ◎ 驱动程序调用驱动程序
- ◎ 让设备实现即插即用
- ◎ I/O端口操作
- ◎ USB设备驱动
- ◎ 虚拟串口设备驱动
- ◎ IRP
- ◎ 高级调试技巧
- ◎ Windows操作驱动基本概念
- ◎ 驱动程序基本结构
- ◎ Windows内核函数
- ◎ 驱动程序的同步处理
- ◎ 定时器
- ◎ 分层驱动程序
- ◎ 电源管理
- ◎ PCI设备驱动
- ◎ SDIO设备驱动
- ◎ 摄像头设备驱动程序
- ◎ 过滤驱动程序



光盘中含有书中所有实例源代码，方便读者学习。

网上订购: www.dearbook.com.cn
第二书店·第一服务



责任编辑: 高洪霞
责任美编: 谢丹丹



本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

上架建议: 操作系统 > Windows驱动

ISBN 978-7-121-06846-1



9 787121 068461 >

定价: 65.00元(含光盘1张)

Windows

驱动开发技术详解

Windows Driver Development Internals

张帆 史彩成 等编著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书由浅入深、循序渐进地介绍了 Windows 驱动程序的开发方法与调试技巧。本书共分 23 章，内容涵盖了 Windows 操作系统的基本原理、NT 驱动程序与 WDM 驱动程序的构造、驱动程序中的同步异步处理方法、驱动程序中即插即用功能、驱动程序的各种调试技巧等。同时，还针对流行的 PCI 驱动程序、USB 驱动程序、虚拟串口驱动程序、摄像头驱动程序、SDIO 驱动程序进行了详细的介绍，本书最大的特色在于每一节的例子都是经过精挑细选的，具有很强的针对性。力求让读者通过亲自动手实验，掌握各类 Windows 驱动程序的开发技巧，学习尽可能多的 Windows 底层知识。

本书适用于中、高级系统程序员，同时也可用做高校计算机专业操作系统实验课的补充教材。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目 (CIP) 数据

Windows 驱动开发技术详解 / 张帆等编著. —北京: 电子工业出版社, 2008.7

ISBN 978-7-121-06846-1

I. W… II. 张… III. 窗口软件, Windows—驱动程序—程序设计 IV. TP316.7

中国版本图书馆 CIP 数据核字 (2008) 第 080993 号

责任编辑: 高洪霞

印 刷: 北京智力达印刷有限公司

装 订: 北京中新伟业印刷有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 787×1092 1/16 印张: 34.75 字数: 851 千字

印 次: 2008 年 7 月第 1 次印刷

印 数: 5000 册 定价: 65.00 元 (含光盘 1 张)

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线: (010) 88258888。

院士推荐



目前,电子系统设计广泛采用通用操作系统,达到降低系统的设计难度和缩短研发周期。实现操作系统与硬件快速信息交换是电子系统设计的关键。

通用操作系统硬件驱动程序的开发,编写者不仅需要精通硬件设备、计算机总线,而且需要 Windows 操作系统知识以及调试技巧。学习和掌握 Windows 硬件驱动程序的开发是电子系统设计人员必备的能力。

本书是作者结合教学和科研实践经验编写而成的,不仅详细介绍了 Windows 内核原理,并且介绍了编程技巧和应用实例,兼顾了在校研究生和工程技术人员的实际需求,对教学、生产和科研有现实的指导意义,是一本值得推荐的专著。

中国工程院院士

2008 年 5 月



自序



写这本书，是为了圆自己一个梦！

1. 你有这样的困惑吗？

你在学习 Windows 驱动程序开发的时候，有没有这样的感觉：觉得入门太难了；总有一大堆莫名其妙的术语，如“中断请求级别”、“派遣函数”、“线程上下文”、“完成例程”等；总能碰到很多诸如 PCI 总线、USB 总线等内容；还有那些无休止的死机、蓝屏等错误……

这可能让你感到很困惑。但这很正常，因为 Windows 驱动开发涉及 Windows 操作系统底层的很多知识，而且很多概念非常抽象，不容易理解。这对于入门人员，甚至有一定基础的开发者都有一定的困难。我也曾经有过和你们一样的经历，当然除了困惑之外，还有解决困惑之后的喜悦。

2. 我的经历

当我第一次接触 Windows 驱动开发时，就觉得非常吃力。那是在研究生一年级的時候，实验室在开发一个 PCI 总线视频采集卡，需要编写驱动程序来读取 PCI 卡上的数据。由于不熟悉 Windows 驱动程序，编译、安装等很简单的问题都困惑我很长时间。通过长时间的摸索，当我第一次用自己编写的驱动程序采集到 PCI 卡上的数据时，我感到非常兴奋。虽然几秒钟后，Windows 无情地蓝屏重启了，但我依然感觉很有成就感。那时候最喜欢做的事情，就是叫周围的同学“欣赏”设备管理器中我编写的设备。又经过很长时间，我才最终将蓝屏死机的原因找出，让驱动程序稳定地运行。

后来，我又开发了很多驱动程序，有 PCI 驱动、USB 驱动、摄像头驱动、SDIO 驱动。渐渐地，我发现驱动程序开发并没有想象中的那么困难。只要对驱动程序开发“入门”后，它就不再是一个神秘的事情了。

我还编写过一些 Linux 驱动程序，结果发现 Linux 设备驱动程序和 Windows 设备驱动程序有很多类似的地方。相比而言，Linux 驱动模型比较简单，加之 Linux 内核的源码是开放的，所以很多地方可以对照 Linux 内核源码进行学习。而 Windows 驱动程序模型比较复杂，其内核也没有提供源码，所以 Windows 驱动程序的编写相对困难一些。

3. 圆自己一个梦

回想当初自己学习 Windows 驱动开发的情景，感想颇多。各种各样的困难，完成一个驱动开发后的喜悦，为了找一本好的学习资料几乎翻遍了图书馆……这些至今都还深深地印在我的脑海里。

随着开发经验的积累和技术水平的提升，越来越想写本 Windows 驱动开发的书，以便向更多的人介绍 Windows 驱动程序的开发经验，使那些初学者快速入门，少走弯路，也能让已经有一定基础的人有所借鉴。这也算是圆我自己的一个梦吧。

当我把这个计划告诉我的老师史彩成时，他非常支持我，而且也愿意和我一起来完成这本书的写作。经过一年多的努力，我们终于完成了这个“大工程”，心中也自然非常喜悦。这无论是对我们，还是对渴望学习 Windows 驱动开发的人，都算是有了一个“交代”。

但愿这本书能够成为想致力于 Windows 驱动开发人员的良师益友，让你有所获益。我也相信，当你读完本书后应该已经能够编写大部分的 Windows 设备驱动程序了。

张 帆

前言

PREFACE

你是否想知道 USB 移动硬盘插入 PC 后，Windows 是如何识别的？
你是否想知道 Windows 是如何得到显卡中的数据的数据的？
你是否想知道什么导致了系统蓝屏死机？
你是否被老板或者导师逼着写一个 PCI、USB 等驱动程序，正感到无从下手？
你是否对 Windows 内核怀着强烈的好奇心？
如果你的回答为“是”，那么阅读本书将是最佳的选择！



上图是 Windows 操作系统的一个示意图。一般的 Windows 程序员都是编写应用程序或者用户 DLL，而不会对 Windows 底层有更深入的了解。而驱动程序位于操作系统的底层，它和内核紧密联系。另外，驱动程序直接操作硬件设备，但究竟如何操作，大部分程序员都不能清楚地讲出来。这些都使得驱动程序开发变得很神秘，仿佛都应该是编程高手的事情。

对于驱动程序开发，书店里很少能见到这方面的书籍。笔者在学习的时候尝到了各种苦头。为了帮助大家快速掌握驱动程序开发，笔者萌生了写一本书的想法。

本书的特点

1. 快速上手：为了让读者快速上手，笔者先给出两个驱动程序例子。这两个例子分别代表 Windows 两类最基本的驱动程序，NT 式驱动程序和 WDM 式驱动程序。笔者非常详细地介绍了驱动程序编译、安装、调试的方法。编译驱动程序一般使用 build 工具，但是考虑到很多读者都是 VC 程序员，笔者特意介绍了如何用 VC 编译器编译驱动程序。
2. 内容翔实，实例丰富：本书详细地介绍了 PCI 驱动程序、USB 驱动程序、虚拟串口程序、虚拟摄像头程序、SDIO 驱动程序的开发，并辅以大量实例，使读者可以边学技术，边进行实践。
3. 介绍多种调试技巧：驱动程序由于运行在内核模式下，很难像普通应用程序那样可以方便地调试。尤其对于 VC 程序员来说，以前的那些调试技巧，很多都不能用了。另外，莫名其妙的“蓝屏死机”也会成为驱动程序开发人员的梦魇。笔者结合自己开发驱动

程序多年的经验，归纳了多种调试驱动程序的高级技巧。这些包括用 WinDbg 和 VMWare 软件对驱动进行源码级调试、用 WinDbg 调试蓝屏后的 Dump 文件等。

4. 灵活地使用一些工具：工欲善其事，必先利其器。很多工具软件会帮助我们更好地了解驱动程序内部的运行情况。本书将介绍很多实用的工具软件，如调试 USB 驱动程序的 BusHound 软件、查看 IRP 的 IRPTrace 软件、查看调试信息的 DebugView 软件、加载 NT 式驱动的 DriverMonitor 软件、加载 WDM 式驱动的 EzDriverInstaller 及查看设备对象的 DeviceTree 工具等。

5. 分析本质：本书对驱动程序的讨论不是仅停留在“表面”，更多地方是带领读者深入到操作系统的底层。本书对驱动程序涉及的操作系统中各个组件都有深入的介绍。另外，本书详细地介绍了驱动程序中的同步处理和异步处理。正确处理同步与异步，会使驱动程序更稳定，运行效率更高。

6. 探讨 Windows 内核：驱动程序和 Windows 的内核紧密相连。本书讲述了很多 Windows 内核的原理。由于 Windows 不是开源的操作系统，所以很少有书籍涉及 Windows 内核的原理。深入理解 Windows 内核的构造与原理，将更好地帮助程序员写出稳定的驱动程序。

本书的内容

本书由 23 章组成，内容分布如下：

入门篇	编译、安装方法 (1)	介绍 NT 式、WDM 式驱动程序的编译、安装方法
	驱动程序开发的基本方法 (2~7)	介绍驱动程序的基本概念、基本数据结构。介绍驱动程序中经常用到的内核函数。介绍驱动程序的入口函数、卸载函数、IRP 处理函数等
进阶篇	同步和异步处理 (8~9)	介绍驱动程序内部对同步操作请求和异步操作请求的处理。介绍如何编写同步和异步的 IRP 处理函数
	定时器 (10)	介绍两种内核模式下的定时器使用方法，另外还介绍了 4 种在内核模式下等待的方法
	驱动程序之间的调用 (11~12)	介绍驱动程序之间的调用方法
	即插即用和电源管理 (13~14)	介绍驱动程序中即插即用和电源管理功能。这些都是 WDM 驱动程序的重点内容
实用篇	各类硬件设备或者模拟设备的驱动程序 (15~20)	介绍几类硬件设备或者模拟设备的驱动程序。包括 USB 设备驱动、PCI 设备驱动、虚拟串口驱动、虚拟摄像头驱动、SDIO 设备驱动等。
提高篇	再论 IRP (21)	讨论一些高级的 IRP 处理方法
	过滤驱动程序 (22)	介绍如何编写过滤驱动程序
	高级调试技巧 (23)	介绍一些高级的驱动程序调试技巧

目 录

Contents

第 1 篇 入门篇

第 1 章 从两个最简单的驱动谈起

2

本章向读者呈现两个最简单的 Windows 驱动程序,一个是 NT 式的驱动程序,另一个是 WDM 式的驱动程序。这两个驱动程序没有操作具体的硬件设备,只是在系统里创建了虚拟设备。在随后的章节中,它们会作为基本驱动程序框架,被本书其他章节的驱动程序开发所复用。笔者将带领读者编写代码、编译、安装和调试程序。

1.1 DDK 的安装	2
1.2 第一个驱动程序 HelloDDK 的代码分析	3
1.2.1 HelloDDK 的头文件	4
1.2.2 HelloDDK 的入口函数	5
1.2.3 创建设备例程	6
1.2.4 卸载驱动例程	8
1.2.5 默认派遣例程	9
1.3 HelloDDK 的编译和安装	9
1.3.1 用 DDK 环境编译 HelloDDK	9
1.3.2 用 VC 集成开发环境编译 HelloDDK	11
1.3.3 HelloDDK 的安装	14
1.4 第二个驱动程序 HelloWDM 的代码分析	16
1.4.1 HelloWDM 的头文件	16
1.4.2 HelloWDM 的入口函数	17
1.4.3 HelloWDM 的 AddDevice 例程	18
1.4.4 HelloWDM 处理 PNP 的回调函数	20
1.4.5 HelloWDM 对 PNP 的默认处理	22
1.4.6 HelloWDM 对 IRP_MN_REMOVE_DEVICE 的处理	23
1.4.7 HelloWDM 对其他 IRP 的回调函数	23
1.4.8 HelloWDM 的卸载例程	24
1.5 HelloWDM 的编译和安装	24
1.5.1 用 DDK 编译环境编译 HelloWDM	24
1.5.2 HelloWDM 的编译过程	25
1.5.3 安装 HelloWDM	25
1.6 小结	29

第2章 Windows 操作驱动的基本概念

31

驱动程序被操作系统加载在内核模式下,它与 Windows 操作系统内核的其他组件进行密切交互。本章主要介绍 Windows 操作系统内核的基本概念,同时还介绍应用程序和驱动程序之间的通信方法。

2.1	Windows 操作系统概述	31
2.1.1	Windows 家族	31
2.1.2	Windows 特性	32
2.1.3	用户模式和内核模式	34
2.1.4	操作系统与应用程序	36
2.2	操作系统分层	37
2.2.1	Windows 操作系统总体架构	37
2.2.2	应用程序与 Win32 子系统	38
2.2.3	其他环境子系统	40
2.2.4	Native API	41
2.2.5	系统服务	41
2.2.6	执行程序组件	42
2.2.7	驱动程序	44
2.2.8	内核	44
2.2.9	硬件抽象层	45
2.2.10	Windows 与微内核	45
2.3	从应用程序到驱动程序	46
2.4	小结	48

第3章 Windows 驱动编译环境配置、安装及调试

49

本章将带领读者一步步对驱动程序进行编译、安装和简单的调试工作。这些步骤虽然简单,但往往困惑着初次接触驱动程序的开发者。

3.1	用 C 语言还是用 C++语言	49
3.1.1	调用约定	50
3.1.2	函数的导出名	52
3.1.3	运行时函数的调用	53
3.2	用 DDK 编译环境编译驱动程序	54
3.2.1	编译版本	55
3.2.2	nmake 工具	55
3.2.3	build 工具	56
3.2.4	makefile 文件	57
3.2.5	dirs 文件	58
3.2.6	sources 文件	58
3.2.7	makefile.inc 文件	59
3.2.8	build 工具的环境变量	60

3.2.9	build 工具的命令行参数	61
3.3	用 VC 编译驱动程序	62
3.3.1	建立驱动程序工程	62
3.3.2	修改编译选项	62
3.3.3	修改链接选项	63
3.3.4	其他修改	64
3.3.5	VC 编译小结	65
3.4	查看调试信息	66
3.4.1	打印调试语句	66
3.4.2	查看调试语句	67
3.5	手动加载 NT 式驱动	68
3.6	编写程序加载 NT 式驱动	68
3.6.1	SCM 组件和 Windows 服务	69
3.6.2	加载 NT 驱动的代码	71
3.6.3	卸载 NT 驱动的代码	74
3.6.4	实验	76
3.7	WDM 式驱动的加载	78
3.7.1	WDM 的手动安装	78
3.7.2	简单的 INF 文件剖析	79
3.8	WDM 设备安装在注册表中的变化	81
3.8.1	硬件子键	81
3.8.2	类子键	83
3.8.3	服务子键	85
3.9	小结	86

第 4 章 驱动程序的基本结构

87

本章首先对 Windows 驱动程序的两个重要数据结构进行介绍,分别是驱动对象和设备对象数据结构。另外还要介绍 NT 驱动程序和 WDM 驱动程序的入口函数、卸载例程、各种 IRP 派遣上函数等。

4.1	Windows 驱动程序中重要的数据结构	87
4.1.1	驱动对象 (DRIVER_OBJECT)	87
4.1.2	设备对象 (DEVICE_OBJECT)	89
4.1.3	设备扩展	91
4.2	NT 式驱动的基本结构	92
4.2.1	驱动加载过程与驱动入口函数 (DriverEntry)	92
4.2.2	创建设备对象	95
4.2.3	DriverUnload 例程	97
4.2.4	用 WinObj 观察驱动对象和设备对象	98
4.2.5	用 DeviceTree 观察驱动对象和设备对象	101
4.3	WDM 式驱动的基本结构	102

4.3.1	物理设备对象与功能设备对象	102
4.3.2	WDM 驱动的入口程序	104
4.3.3	WDM 驱动的 AddDevice 例程	105
4.3.4	DriverUnload 例程	107
4.3.5	对 IRP_MN_REMOVE_DEVICE IRP 的处理	108
4.3.6	用 Device Tree 查看 WDM 设备对象栈	109
4.4	设备的层次结构	110
4.4.1	驱动程序的垂直层次结构	111
4.4.2	驱动程序的水平层次结构	112
4.4.3	驱动程序的复杂层次结构	112
4.5	实验	114
4.5.1	改写 HelloDDK 查看驱动结构	114
4.5.2	改写 HelloWDM 查看驱动结构	116
4.6	小结	117

第 5 章 Windows 内存管理

118

本章围绕着驱动程序中的内存操作进行了介绍。在驱动程序开发中，首先要注意分页内存和非分页内存的使用。同时，还需要区分物理内存地址和虚拟内存地址这两个概念。

5.1	内存管理概念	118
5.1.1	物理内存概念 (Physical Memory Address)	118
5.1.2	虚拟内存地址概念 (Virtual Memory Address)	119
5.1.3	用户模式地址和内核模式地址	120
5.1.4	Windows 驱动程序和进程的关系	121
5.1.5	分页与非分页内存	122
5.1.6	分配内核内存	123
5.2	在驱动中使用链表	124
5.2.1	链表结构	124
5.2.2	链表初始化	125
5.2.3	从首部插入链表	126
5.2.4	从尾部插入链表	126
5.2.5	从链表删除	127
5.2.6	实验	129
5.3	Lookaside 结构	130
5.3.1	频繁申请内存的弊端	130
5.3.2	使用 Lookaside	130
5.3.3	实验	132
5.4	运行时函数	133
5.4.1	内存间复制 (非重叠)	133
5.4.2	内存间复制 (可重叠)	134
5.4.3	填充内存	134

5.4.4	内存比较	135
5.4.5	关于运行时函数使用的注意事项	135
5.4.6	实验	137
5.5	使用 C++特性分配内存	137
5.6	其他	139
5.6.1	数据类型	139
5.6.2	返回状态值	140
5.6.3	检查内存可用性	142
5.6.4	结构化异常处理 (try-except 块)	142
5.6.5	结构化异常处理 (try-finally 块)	144
5.6.6	使用宏需要注意的地方	146
5.6.7	断言	147
5.7	小结	147

第 6 章 Windows 内核函数 148

本章介绍了 Windows 内核模式下的一些常用内核函数,这些函数在驱动程序的开发中将会经常用到。

6.1	内核模式下的字符串操作	148
6.1.1	ASCII 字符串和宽字符串	148
6.1.2	ANSI_STRING 字符串与 UNICODE_STRING 字符串	149
6.1.3	字符初始化与销毁	151
6.1.4	字符串复制	152
6.1.5	字符串比较	153
6.1.6	字符串转成大写	154
6.1.7	字符串与整型数字相互转换	155
6.1.8	ANSI_STRING 字符串与 UNICODE_STRING 字符串相互转换	157
6.2	内核模式下的文件操作	158
6.2.1	文件的创建	158
6.2.2	文件的打开	161
6.2.3	获取或修改文件属性	163
6.2.4	文件的写操作	166
6.2.5	文件的读操作	167
6.3	内核模式下的注册表操作	169
6.3.1	创建关闭注册表	170
6.3.2	打开注册表	172
6.3.3	添加、修改注册表键值	173
6.3.4	查询注册表	175
6.3.5	枚举子项	178
6.3.6	枚举子键	180
6.3.7	删除子项	182
6.3.8	其他	183

6.4 小结	185
--------	-----

第 7 章 派遣函数 186

本章重点介绍了驱动程序中的处理 IRP 请求的派遣函数。所有对设备的操作最终将转化为 IRP 请求，这些 IRP 请求会被传送到派遣函数处理。

7.1 IRP 与派遣函数	186
7.1.1 IRP	186
7.1.2 IRP 类型	188
7.1.3 对派遣函数的简单处理	188
7.1.4 通过设备链接打开设备	190
7.1.5 编写一个更通用的派遣函数	191
7.1.6 跟踪 IRP 的利器 IRPTrace	193
7.2 缓冲区方式读写操作	196
7.2.1 缓冲区设备	196
7.2.2 缓冲区设备读写	197
7.2.3 缓冲区设备模拟文件读写	200
7.3 直接方式读写操作	203
7.3.1 直接读取设备	204
7.3.2 直接读取设备的读写	205
7.4 其他方式读写操作	207
7.4.1 其他方式设备	207
7.4.2 其他方式读写	208
7.5 IO 设备控制操作	209
7.5.1 DeviceIoControl 与驱动交互	209
7.5.2 缓冲内存模式 IOCTL	210
7.5.3 直接内存模式 IOCTL	212
7.5.4 其他内存模式 IOCTL	214
7.6 小结	216

第 2 篇 进阶篇

第 8 章 驱动程序的同步处理 218

本章介绍了驱动程序中常用的同步处理办法，并且将内核模式下的同步处理方法和用户模式下的同步处理方法做了比较。另外，本章还介绍了中断请求级、自旋锁等同步处理机制。

8.1 基本概念	218
8.1.1 问题的引出	218
8.1.2 同步与异步	219
8.2 中断请求级	219
8.2.1 中断请求 (IRQ) 与可编程中断控制器 (PIC)	220
8.2.2 高级可编程控制器 (APIC)	221

8.2.3	中断请求级 (IRQL)	221
8.2.4	线程调度与线程优先级	222
8.2.5	IRQL 的变化	223
8.2.6	IRQL 与内存分页	223
8.2.7	控制 IRQL 提升与降低	224
8.3	自旋锁	224
8.3.1	原理	224
8.3.2	使用方法	225
8.4	用户模式下的同步对象	225
8.4.1	用户模式的等待	226
8.4.2	用户模式开启多线程	226
8.4.3	用户模式的事件	227
8.4.4	用户模式的信号灯	229
8.4.5	用户模式的互斥体	230
8.4.6	等待线程完成	232
8.5	内核模式下的同步对象	232
8.5.1	内核模式下的等待	232
8.5.2	内核模式下开启多线程	234
8.5.3	内核模式下的事件对象	236
8.5.4	驱动程序与应用程序交互事件对象	237
8.5.5	驱动程序与驱动程序交互事件对象	239
8.5.6	内核模式下的信号灯	240
8.5.7	内核模式下的互斥体	241
8.5.8	快速互斥体	243
8.6	其他同步方法	244
8.6.1	使用自旋锁进行同步	245
8.6.2	使用互锁操作进行同步	247
8.7	小结	249

第 9 章 IRP 的同步 250

本章详细地介绍了 IRP 的同步处理方法和异步处理方法。另外，本章还介绍了 StartIO 例程、中断服务例程、DPC 服务例程。

9.1	应用程序对设备的同步异步操作	250
9.1.1	同步操作与异步操作原理	250
9.1.2	同步操作设备	252
9.1.3	异步操作设备 (方式一)	253
9.1.4	异步操作设备 (方式二)	254
9.2	IRP 的同步完成与异步完成	256
9.2.1	IRP 的同步完成	256
9.2.2	IRP 的异步完成	257

9.2.3 取消 IRP	262
9.3 StartIO 例程	264
9.3.1 并行执行与串行执行	264
9.3.2 StartIO 例程	265
9.3.3 示例	267
9.4 自定义的 StartIO	270
9.4.1 多个串行化队列	270
9.4.2 示例	271
9.5 中断服务例程	273
9.5.1 中断操作的必要性	273
9.5.2 中断优先级	274
9.5.3 中断服务例程 (ISR)	274
9.6 DPC 例程	275
9.6.1 延迟过程调用例程 (DPC)	275
9.6.2 DpcForISR	275
9.7 小结	276

第 10 章 定时器

277

本章总结了在内核模式下的四种等待方法，读者可以利用这些方法灵活地用在自己的驱动程序中。最后本章还介绍了如何对 IRP 的超时情况进行处理。

10.1 定时器实现方式一	277
10.1.1 I/O 定时器	277
10.1.2 示例代码	278
10.2 定时器实现方式二	280
10.2.1 DPC 定时器	280
10.2.2 示例代码	282
10.3 等待	284
10.3.1 第一种方法：使用 KeWaitForSingleObject	284
10.3.2 第二种方法：使用 KeDelayExecutionThread	285
10.3.3 第三种方法：使用 KeStallExecutionProcessor	285
10.3.4 第四种方法：使用定时器	286
10.4 时间相关的其他内核函数	286
10.4.1 时间相关函数	286
10.4.2 示例代码	288
10.5 IRP 的超时处理	289
10.5.1 原理	289
10.5.2 示例代码	289
10.6 小结	291

第 11 章 驱动程序调用驱动程序

292

本章主要介绍了如何在驱动程序中调用其他驱动程序。比较简单的方法是将被调用的驱动程序以文件的方式操作。比较高级的方法是构造各种 IRP，并将这些 IRP 传送到被调用的驱动程序中。

11.1 以文件句柄形式调用其他驱动程序	292
11.1.1 准备一个标准驱动	292
11.1.2 获得设备句柄	294
11.1.3 同步调用	295
11.1.4 异步调用方法一	297
11.1.5 异步调用方法二	299
11.1.6 通过符号链接打开设备	301
11.2 通过设备指针调用其他驱动程序	303
11.2.1 用 IoGetDeviceObjectPointer 获得设备指针	304
11.2.2 创建 IRP 传递给驱动的派遣函数	305
11.2.3 用 IoBuildSynchronousFsdRequest 创建 IRP	306
11.2.4 用 IoBuildAsynchronousFsdRequest 创建 IRP	308
11.2.5 用 IoAllocateIrp 创建 IRP	311
11.3 其他方法获得设备指针	314
11.3.1 用 ObReferenceObjectByName 获得设备指针	314
11.3.2 剖析 IoGetDeviceObjectPointer	317
11.4 小结	318

第 12 章 分层驱动程序

319

本章主要介绍了分层驱动的概念。分层驱动可以将功能复杂的驱动程序分解为多个功能简单的驱动程序。多个分层的驱动程序形成一个设备堆栈，IRP 请求首先发送到设备堆栈的顶层，然后依次穿越每层的设备堆栈，最终完成 IRP 请求。

12.1 分层驱动程序概念	319
12.1.1 分层驱动程序的概念	319
12.1.2 设备堆栈与挂载	321
12.1.3 I/O 堆栈	322
12.1.4 向下转发 IRP	323
12.1.5 挂载设备对象示例	324
12.1.6 转发 IRP 示例	325
12.1.7 分析	326
12.1.8 遍历设备栈	327
12.2 完成例程	330
12.2.1 完成例程概念	330
12.2.2 传播 Pending 位	332
12.2.3 完成例程返回 STATUS_SUCCESS	333

• XVII •

12.2.4	完成例程返回 STATUS_MORE_PROCESSING_REQUIRED	334
12.3	将 IRP 分解成多个 IRP	336
12.3.1	原理	336
12.3.2	准备底层驱动	337
12.3.3	读派遣函数	338
12.3.4	完成例程	341
12.3.5	分析	342
12.4	WDM 驱动程序架构	344
12.4.1	WDM 与分层驱动程序	344
12.4.2	WDM 的加载方式	345
12.4.3	功能设备对象	346
12.4.4	物理设备对象	346
12.4.5	物理设备对象与即插即用	348
12.5	小结	349

第 13 章 让设备实现即插即用

350

本章首先介绍即插即用的概念和驱动程序支持即插即用功能的必要性。另外，本章还介绍如何利用 WDM 驱动程序开发框架设计支持即插即用功能的驱动程序。

13.1	即插即用概念	350
13.1.1	历史原因	350
13.1.2	即插即用的目标	351
13.1.3	Windows 中即插即用相关组件	351
13.1.4	遗留驱动程序	352
13.2	即插即用 IRP	352
13.2.1	即插即用 IRP 的功能代码	353
13.2.2	处理即插即用 IRP 的派遣函数	353
13.3	通过设备接口寻找设备	356
13.3.1	设备接口	356
13.3.2	WDM 驱动中设置接口	357
13.3.3	应用程序寻找接口	359
13.3.4	查看接口设备	360
13.4	启动和停止设备	361
13.4.1	为一个实际硬件安装 HelloWDM	362
13.4.2	启动设备	364
13.4.3	转发并等待	366
13.4.4	获得设备相关资源	367
13.4.5	枚举设备资源	368
13.4.6	停止设备	372
13.5	即插即用的状态转换	373
13.5.1	状态转换图	373

13.5.2	IRP_MN_QUERY_STOP_DEVICE	374
13.5.3	IRP_MN_QUERY_REMOVE_DEVICE	374
13.6	其他即插即用 IRP	375
13.6.1	IRP_MN_FILTER_RESOURCE_REQUIREMENTS	375
13.6.2	IRP_MN_QUERY_CAPABILITIES	376
13.7	小结	377

第 14 章 电源管理 378

本章主要介绍了如何在 WDM 驱动程序中进行电源处理。电源处理主要是处理好电源状态和设备状态。

14.1	WDM 电源管理模型	378
14.1.1	概述	378
14.1.2	热插拔	378
14.1.3	电源状态	379
14.1.4	设备状态	379
14.1.5	状态转换	380
14.2	处理 IRP_MJ_POWER	381
14.3	处理 IRP_MN_QUERY_CAPABILITIES	381
14.3.1	DEVICE_CAPABILITIES	381
14.3.2	一个试验	382
14.4	小结	384

第 3 篇 实用篇

第 15 章 I/O 端口操作 386

本章总结了多种 I/O 端口操作的方法。这些方法本质上是一样的，都是将端口输入输出的汇编指令运行在内核模式中。

15.1	概述	386
15.1.1	从 DOS 说起	386
15.1.2	汇编实现	387
15.1.3	DDK 实现	389
15.2	工具软件 WinIO	390
15.2.1	WinIO 简介	390
15.2.2	使用方法	390
15.3	端口操作实现方法一	391
15.3.1	驱动端程序	391
15.3.2	应用程序端程序	393
15.4	端口操作实现方法二	394
15.4.1	驱动端程序	394

15.4.2	应用程序端程序	396
15.5	端口操作实现方法三	397
15.5.1	驱动端程序	397
15.5.2	应用程序端程序	398
15.6	端口操作实现方法四	399
15.6.1	原理	399
15.6.2	驱动端程序	400
15.6.3	应用程序端程序	401
15.7	驱动 PC 喇叭	402
15.7.1	可编程定时器	402
15.7.2	PC 喇叭	403
15.7.3	操作代码	404
15.8	操作并口设备	405
15.8.1	并口设备简介	405
15.8.2	并口寄存器	406
15.8.3	并口设备操作	408
15.9	小结	410

第 16 章 PCI 设备驱动

411

本章主要介绍 PCI 设备的驱动开发。首先介绍了 PCI 总线协议。作为驱动程序员，开发 PCI 驱动程序首先要了解 PCI 配置空间。根据读取 PCI 配置空间，可以得到 PCI 设备的所有资源。另外，本章还总结了四种获取 PCI 配置空间的方法。

16.1	PCI 总线协议	411
16.1.1	PCI 总线简介	411
16.1.2	PCI 配置空间简介	412
16.2	访问 PCI 配置空间方法一	414
16.2.1	两个重要寄存器	414
16.2.2	示例	415
16.3	访问 PCI 配置空间方法二	417
16.3.1	DDK 函数读取配置空间	417
16.3.2	示例	418
16.4	访问 PCI 配置空间方法三	419
16.4.1	通过即插即用 IRP 获得 PCI 配置空间	420
16.4.2	示例	420
16.5	访问 PCI 配置空间方法四	421
16.5.1	创建 IRP_MN_READ_CONFIG	422
16.5.2	示例	422
16.6	PCI 设备驱动开发示例	423
16.6.1	开发步骤	424
16.6.2	中断操作	424

16.6.3 操作设备物理内存	425
16.6.4 示例	426
16.7 小结	429

第 17 章 USB 设备驱动 430

本章首先介绍了 USB 总线协议的基本框架，其中包括 USB 总线的拓扑结构，USB 通信的流程，还有 USB 的四种传输模式。另外，本章介绍了如何编写 USB 总线设备的驱动程序。

17.1 USB 总线协议	430
17.1.1 USB 设备简介	430
17.1.2 USB 连接拓扑结构	431
17.1.3 USB 通信的流程	433
17.1.4 USB 四种传输模式	435
17.2 Windows 下的 USB 驱动	438
17.2.1 观察 USB 设备的工具	438
17.2.2 USB 设备请求	440
17.2.3 设备描述符	440
17.2.4 配置描述符	442
17.2.5 接口描述符	443
17.2.6 端点描述符	443
17.3 USB 驱动开发实例	444
17.3.1 功能驱动与物理总线驱动	444
17.3.2 构造 USB 请求包	445
17.3.3 发送 USB 请求包	446
17.3.4 USB 设备初始化	447
17.3.5 USB 设备的插拔	447
17.3.6 USB 设备的读写	448
17.4 小结	450

第 18 章 SDIO 设备驱动 451

本章首先介绍了 SDIO 协议，讲述了 SD 内存卡和 SDIO 卡的兼容问题。然后介绍了 SDIO 协议中的发送命令、回应命令、传送数据等相关协议。随后，本章又介绍了 Windows 中，DDK 提供的对 SDIO 卡设备的支持。然后介绍了如何利用总线驱动，使 SDIO 设备初始化，接收中断，发送和接收数据等操作。

18.1 SDIO 协议	451
18.1.1 SD 内存卡概念	451
18.1.2 SDIO 卡概念	452
18.1.3 SDIO 总线	452
18.1.4 SDIO 令牌	453
18.1.5 SDIO 令牌格式	455
18.1.6 SDIO 的寄存器	456

18.1.7	CMD52 命令	458
18.1.8	CMD53 命令	459
18.2	SDIO 卡驱动开发框架	459
18.2.1	SDIO Host Controller 驱动	459
18.2.2	SDIO 卡的初始化	460
18.2.3	中断回调函数	461
18.2.4	获得和设置属性	462
18.2.5	CMD52	464
18.2.6	CMD53	465
18.3	SDIO 开发实例	467
18.4	小结	467

第 19 章 虚拟串口设备驱动 469

本章介绍了串口开发的框架模型。在串口的 AddDevice 例程中需要暴露出一个串口的符号连接，另外在相应的注册表中需要进行设置。在串口与应用程序的通信中，主要是一组 DDK 定义的 IO 控制码，这些 IO 控制码负责由应用程序向驱动发出请求。

19.1	串口简介	469
19.2	DDK 串口开发框架	470
19.2.1	串口驱动的入口函数	470
19.2.2	应用程序与串口驱动的通信	473
19.2.3	写的实现	475
19.2.4	读的实现	477
19.3	小结	478

第 20 章 摄像头设备驱动程序 479

本章主要介绍了微软提供的摄像头驱动框架。在该框架中，微软提供了类驱动和小驱动的概念。对于驱动程序员的任务就是编写小驱动程序。

20.1	WDM 摄像头驱动框架	479
20.1.1	类驱动与小驱动	479
20.1.2	摄像头的类驱动与小驱动	480
20.1.3	编写小驱动程序	480
20.1.4	小驱动的流控制	481
20.2	虚拟摄像头开发实例	482
20.2.1	编译和安装	482
20.2.2	虚拟摄像头入口函数	484
20.2.3	对 STREAM_REQUEST_BLOCK 的处理函数	485
20.2.4	打开视频流	487
20.2.5	对视频流的读取	488
20.3	小结	489

第4篇 提高篇

第21章 再论 IRP

492

本章将相关 IRP 的操作做了进一步的总结。首先是转发 IRP，归纳了几种不同的方式。其次总结了创建 IRP 的几种不同方法。创建 IRP 总的来说分为创建同步 IRP 和创建异步 IRP。对于创建同步 IRP，操作比较简单，I/O 管理器会负责回收 IRP 的相关内存，但是使用不够灵活。对于创建异步 IRP，操作比较复杂，程序员需要自己负责对 IRP 及相关内存回收，但使用十分灵活。

21.1 转发 IRP	492
21.1.1 直接转发	492
21.1.2 转发并且等待	492
21.1.3 转发并且设置完成例程	494
21.1.4 暂时挂起当前 IRP	495
21.1.5 不转发 IRP	496
21.2 创建 IRP	496
21.2.1 IoBuildDeviceIoControlRequest	497
21.2.2 创建有超时的 IOCTL IRP	498
21.2.3 用 IoBuildSynchronousFsdRequest 创建 IRP	499
21.2.4 关于 IoBuildAsynchronousFsdRequest	501
21.2.5 关于 IoAllocateIrp	502
21.3 小结	505

第22章 过滤驱动程序

506

本章主要介绍 WDM 和 NT 式过滤驱动程序开发。过滤驱动程序开发十分灵活，可以修改已有驱动程序的功能，也可以对数据进行过滤加密。另外，利用过滤驱动程序还能编写出很多具有相当功能强大的程序来。

22.1 文件过滤驱动程序	506
22.1.1 过滤驱动程序概念	506
22.1.2 过滤驱动程序的入口函数	506
22.1.3 U 盘过滤驱动程序	509
22.1.4 过滤驱动程序加载方法一	510
22.1.5 过滤驱动程序加载方法二	511
22.1.6 过滤驱动程序的 AddDevice 例程	512
22.1.7 磁盘命令过滤	513
22.2 NT 式过滤驱动程序	516
22.2.1 NT 式过滤驱动程序	516
22.2.2 NT 过滤驱动的入口函数	517
22.2.3 挂载过滤驱动	517
22.2.4 过滤键盘读操作	518
22.3 小结	520

第 23 章 高级调试技巧

521

本章将介绍一些 Windows 开发驱动的高级调试技巧。有一些高级驱动程序调试技巧，可以帮助程序员找出驱动程序中的 Bug。另外，利用一些第三方工具软件，也可以帮助程序员找到驱动程序中的漏洞，从而提高开发效率。

23.1 一般性调试技巧	521
23.1.1 打印调试信息	521
23.1.2 存储 dump 信息	521
23.1.3 使用 WinDbg 调试工具	522
23.2 高级内核调试技巧	524
23.2.1 安装 VMWare	525
23.2.2 在虚拟机上加载驱动程序	526
23.2.3 VMWare 和 WinDbg 联合调试驱动程序	527
23.3 用 IRPTrace 调试驱动程序	528
23.4 小结	530

第 1 篇

入门篇

第 1 章 从两个最简单的驱动谈起

第 2 章 Windows 操作驱动的基本概念

第 3 章 Windows 驱动编译环境配置、安装及调试

第 4 章 驱动程序的基本结构

第 5 章 Windows 内存管理

第 6 章 Windows 内核函数

第 7 章 派遣函数

第 1 章 从两个最简单的驱动谈起

Windows 驱动程序的编写,往往需要开发人员对 Windows 内核有深入了解和大量的内核调试技巧,稍有不慎,就会造成系统的崩溃。因此,初次涉及 Windows 驱动程序开发的程序员,即使拥有大量 Win32 程序的开发技巧,往往也很难入门。

本章向读者呈现两个最简单的 Windows 驱动程序,一个是 NT 式的驱动程序,另一个是 WDM 式的驱动程序。这两个驱动程序没有操作具体的硬件设备,只是在系统里创建了虚拟设备。在随后的章节中,它们会作为基本驱动程序框架,被本书其他章节的驱动程序开发所复用。笔者将带领读者编写代码、编译、安装和调试程序。相信对第一次编写驱动程序的读者来说,这将是非常激动和有趣的。代码的具体讲解将分散在后面的章节论述。现在请和笔者一起,开始 Windows 驱动编程之旅吧!

1.1 DDK 的安装

在编写第一个驱动之前,需要先安装微软公司提供的 Windows 驱动程序开发包 DDK (Driver Development Kit)。笔者计算机里安装的是 Windows XP 2462 版本的 DDK,建议读者安装同样版本或者更高版本的 DDK,如图 1-1 所示。

在安装的时候请选择完全安装,即安装 DDK 的所有部件,如图 1-2 所示。因为除了 DDK 的基本编译环境外,DDK 还提供了大量的源代码和实用工具,这对于 Windows 驱动程序的初学者进行学习和编写驱动程序将是非常有用的。

安装完毕后,会在开始菜单中出现相应的项目。其中,主要用到的是 Build Environment,如图 1-3 所示。该版本的 DDK 会同时安装上 Windows 2000 和 Windows XP 的编译环境。



图 1-1 DDK 的安装

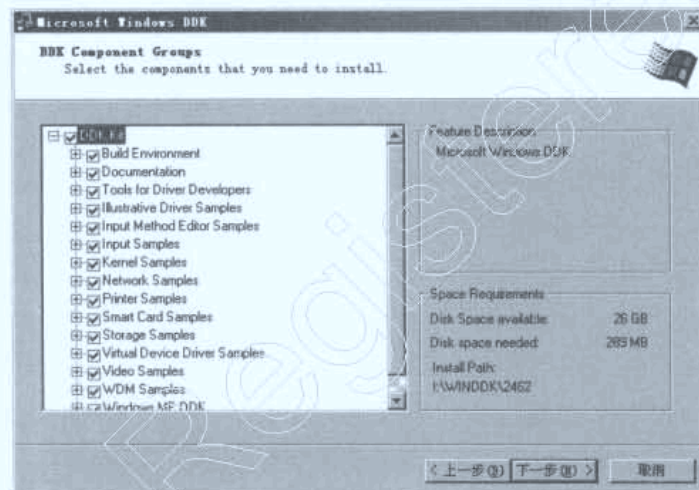


图 1-2 DDK 的安装

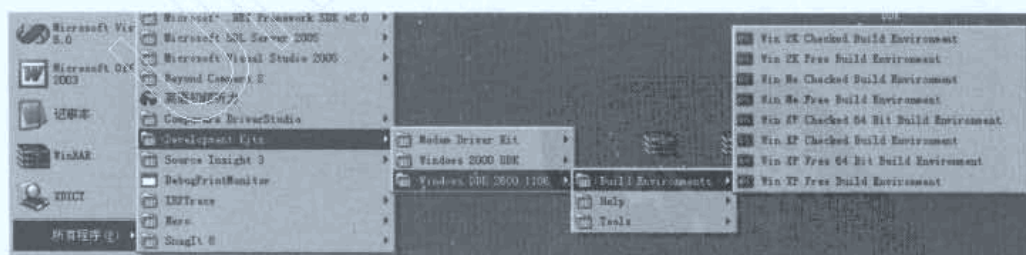


图 1-3 DDK 的编译环境

1.2 第一个驱动程序 HelloDDK 的代码分析

Windows 驱动程序分为两类，一类是不支持即插即用功能的 NT 式驱动程序，另一类

是支持即插即用功能的 WDM 驱动程序。本节介绍的 HelloDDK 是一个最简单的 NT 式驱动程序。在 1.4 节中将给出一个 WDM 式的驱动程序。

1.2.1 HelloDDK 的头文件

HelloDDK 的头文件主要是为了导入驱动程序开发所必需的 NTDDK.h 头文件，此头文件里包含了对 DDK 的所有导出函数的声明。NT 式的驱动程序要导入的头文件是 NTDDK.h，而 WDM 式的驱动程序要导入的头文件为 WDM.h。另外，此头文件中定义了几个标签，分别在程序中指明函数和变量分配在分页内存中或非分页内存中（分页和非分页内存的概念将在第 3 章中讲述）。最后，该头文件给出了此驱动的函数声明。

```
#001  /*****
#002  * 文件名称:Driver.h
#003  * 作 者:张帆
#004  * 完成日期:2007-11-1
#005  *****/
#006  #pragma once
#007
#008  #ifdef __cplusplus
#009  extern "C"
#010  {
#011  #endif
#012  #include <NTDDK.h>
#013  #ifdef __cplusplus
#014  }
#015  #endif
#016
#017  #define PAGEDCODE code_seg("PAGE")
#018  #define LOCKEDCODE code_seg()
#019  #define INITCODE code_seg("INIT")
#020
#021  #define PAGEDDATA data_seg("PAGE")
#022  #define LOCKEDDATA data_seg()
#023  #define INITDATA data_seg("INIT")
#024
#025  #define arraysize(p) (sizeof(p)/sizeof((p)[0]))
#026
#027  typedef struct _DEVICE_EXTENSION {
#028      PDEVICE_OBJECT pDevice;
#029      UNICODE_STRING ustrDeviceName;    //设备名称
#030      UNICODE_STRING ustrSymLinkName;   //符号链接名
#031  } DEVICE_EXTENSION, *PDEVICE_EXTENSION;
#032
#033  // 函数声明
#034
#035  NTSTATUS CreateDevice (IN PDRIVER_OBJECT pDriverObject);
#036  VOID HelloDDKUnload (IN PDRIVER_OBJECT pDriverObject);
#037  NTSTATUS HelloDDKDispatchRoutine(IN PDEVICE_OBJECT pDevObj,
#038                                  IN PIRP pIrps);
#039
```

此段代码可以在配套光盘中本章的 NT_Driver 目录下找到。

- 代码 6~15 行, 包含 `ddk.h` 头文件, 所有的 NT 式驱动程序都要包含此头文件。因为这里采用的是 C++ 语言编写, 如果直接包含 `ntddk.h`, 函数的符号表会导入错误, 所以需要加入 `extern "C"`, 这样可以保证符号表正确导入。关于 C++ 编写驱动需要注意的地方, 将在第 3 章进行论述。
- 代码 17~23 行, 定义分页标记、非分页标记和初始化内存块。在 Windows 驱动程序的开发中, 所有程序的函数和变量要被指明被加载到分页内存中还是在非分页内存中。程序代码中加入这里定义的宏, 就会被指明函数和变量是位于分页或非分页内存中。另外, 有一个特殊的函数 `DriverEntry` 需要放在 INIT 标志的内存中。INIT 标志指明该函数只是在加载的时候需要载入内存, 而当驱动程序成功加载后, 该函数可以从内存中卸载掉。
- 代码 27~31 行, 指定一个设备扩展结构体, 这种结构体广泛应用于驱动程序中。根据不同驱动程序的需要, 它负责补充定义设备的相关信息。
- 代码 33~38 行是函数的声明。

1.2.2 HelloDDK 的入口函数

和普通的应用程序不同, Windows 驱动程序的入口函数不是 `main` 函数, 而是一个叫做 `DriverEntry` 的函数, 代码将在下面列出。`DriverEntry` 函数由内核中的 I/O 管理器负责调用, 其函数有两个参数: `pDriverObject` 和 `pRegistryPath`。其中, `pDriverObject` 是 I/O 管理器传递进来的驱动对象, `pRegistryPath` 是一个 Unicode 字符串, 指向此驱动负责的注册表。

```
#001  /*****
#002  * 文件名称:Driver.cpp
#003  * 作    者:张凯
#004  * 完成日期:2007-11-1
#005  *****/
#006
#007  #include "Driver.h"
#008
#009  /*****
#010  * 函数名称:DriverEntry
#011  * 功能描述:初始化驱动程序, 定位和申请硬件资源, 创建内核对象
#012  * 参数列表:
#013      pDriverObject:从 I/O 管理器中传进来的驱动对象
#014      pRegistryPath:驱动程序在注册表的中的路径
#015  * 返回值:返回初始化驱动状态
#016  *****/
#017  #pragma INITCODE
#018  extern "C" NTSTATUS DriverEntry (
#019      IN PDRIVER_OBJECT pDriverObject,
#020      IN PUNICODE_STRING pRegistryPath  )
#021  {
#022      NTSTATUS status;
#023      KdPrint(("Enter DriverEntry\n"));
```

```
#024
#025 //注册其他驱动调用函数入口
#026 pDriverObject->DriverUnload = HelloDDKUnload;
#027 pDriverObject->MajorFunction[IRP_MJ_CREATE] = HelloDDKDispatchRoutine;
#028 pDriverObject->MajorFunction[IRP_MJ_CLOSE] = HelloDDKDispatchRoutine;
#029 pDriverObject->MajorFunction[IRP_MJ_WRITE] = HelloDDKDispatchRoutine;
#030 pDriverObject->MajorFunction[IRP_MJ_READ] = HelloDDKDispatchRoutine;
#031
#032 //创建驱动设备对象
#033 status = CreateDevice(pDriverObject);
#034
#035 KdPrint(("DriverEntry end\n"));
#036 return status;
#037 }
```

此段代码可以在配套光盘中本章的 NT_Driver 目录下找到。

- 代码 17 行，用 `#pragma` 指明此函数是加载到 INIT 内存区域中，即成功卸载后，可以退出内存。
- 代码 18 行，标志 `DriverEntry` 函数的开始。注意此处函数体的前面用 `extern "C"` 修饰，这样在编译的时候会编译成 `_DriverEntry@8` 的符号。如果不加入此修饰符号，编译器会自动按照 C++ 的符号名编译，导致错误链接。
- 代码 23 行，打印一行调试信息。`KdPrint` 其实是一个宏，在调试版本（Checked 版）中，会用 `DbgPrint` 代替。而在发行版（Free 版）中，则不执行任何操作，其功能类似于 MFC 中的 `TRACE` 宏。由于驱动程序是运行在 Windows 的核心态，没有用户界面，所以查看调试信息有别于 Win32 程序。关于查看调试信息的讲解将在第 3 章论述。
- 代码 26~30 行，驱动程序向 Windows 的 I/O 管理器注册一些回调函数。回调函数是由程序员定义的函数，这些函数不是由驱动程序本身负责调用，而是由操作系统负责调用。程序员将这些函数的入口地址告诉操作系统，操作系统会在适当的时候调用这些函数。在这个例子中，这几个回调函数基本是自解释型的，读者可以根据函数名分析出其作用。当驱动被卸载时，调用 `HelloDDKUnload`。当驱动程序处理创建、关闭和读写相关的 IRP 时，调用 `HelloDDKDispatchRoutine`（这里只是将处理函数简化为一个函数，实际情况要比这个复杂）。
- 代码第 33 行，调用 `CreateDevice` 函数，此函数的解释见下一节。
- 代码第 36 行，返回 `CreateDevice` 的执行结果。如果执行正确，驱动将被成功加载。

1.2.3 创建设备例程

`CreateDevice` 函数是一个帮助函数（Helper Function），辅助 `DriverEntry` 创建一个设备对象。其完全可以展开放在 `DriverEntry` 中，但为了代码的条理性，笔者将其构造成为一个辅助函数。


```

#001  /*****
#002  * 函数名称:CreateDevice
#003  * 功能描述:初始化设备对象
#004  * 参数列表:
#005      pDriverObject:从 I/O 管理器中传进来的驱动对象
#006  * 返回 值:返回初始化状态
#007  *****/
#008  #pragma INITCODE
#009  NTSTATUS CreateDevice (
#010      IN PDRIVER_OBJECT pDriverObject)
#011  {
#012      NTSTATUS status;
#013      PDEVICE_OBJECT pDevObj;
#014      PDEVICE_EXTENSION pDevExt;
#015
#016      //创建设备名称
#017      UNICODE_STRING devName;
#018      RtlInitUnicodeString (&devName,L"\\Device\\MyDDKDevice");
#019
#020      //创建设备
#021      status = IoCreateDevice( pDriverObject,
#022                              sizeof(DEVICE_EXTENSION),
#023                              &(UNICODE_STRING)devName,
#024                              FILE_DEVICE_UNKNOWN,
#025                              0, TRUE,
#026                              &pDevObj );
#027      if (!NT_SUCCESS(status))
#028          return status;
#029
#030      pDevObj->Flags |= DO_BUFFERED_IO;
#031      pDevExt = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
#032      pDevExt->pDevice = pDevObj;
#033      pDevExt->ustrDeviceName = devName;
#034      //创建符号链接
#035      UNICODE_STRING symLinkName;
#036      RtlInitUnicodeString(&symLinkName,L"\\??\\HelloDDK");
#037      pDevExt->ustrSymLinkName = symLinkName;
#038      status = IoCreateSymbolicLink( &symLinkName,&devName );
#039      if (!NT_SUCCESS(status))
#040      {
#041          IoDeleteDevice( pDevObj );
#042          return status;
#043      }
#044      return STATUS_SUCCESS;
#045  }

```

此段代码可以在配套光盘中本章的 NT_Driver 目录下找到。

- 代码 16~18 行，构造一个 Unicode 字符串，此字符串用来存储此设备对象的名称。Unicode 字符串大量运用在驱动程序开发中，有关 Unicode 的讲解请参考第 3 章。
- 代码 21~28 行，用 IoCreateDevice 函数创建一个设备对象。其对象名称来自于上一步构造的 Unicode 字符串，设备类型为 FILE_DEVICE_UNKNOWN，且此种设备为独占设备，即设备只能被一个应用程序所使用。
- 代码第 30 行，表明此种设备为 BUFFERED_IO 设备。设备对内存的操作分为两种，

Windows 驱动开发技术详解

BUFFERED_IO 和 DO_DIRECT_IO, 此部分讲解请参考第 3 章。

- 代码 31~33 行, 填写设备的扩展结构体, 在其他驱动程序的函数中, 可以很方便地得到这个结构体, 进而得到该设备的自定义信息。此结构体的定义在 Driver.h 中。
- 代码 34~38 行, 创建符号链接。驱动程序虽然有了设备名称, 但是这种设备名称只能在内核态可见, 而对于应用程序是不可见的。因此, 驱动需要暴露一个符号链接, 该链接指向真正的设备名称。
- 代码 39~44 行, 当设备创建成功后返回。如果不成功, 则删除该设备。

1.2.4 卸载驱动例程

卸载驱动例程用来设备被卸载的情况, 由 I/O 管理器负责调用此回调函数。此例程遍历系统中所有的此类设备对象。第一个设备对象的地址存在于驱动对象的 DeviceObject 域中, 每个设备对象的 NextDevice 域记录着下一个设备对象的地址, 这样就形成一个链表。卸载驱动例程的主要目的就是遍历系统中所有的此类设备对象, 然后删除设备对象以及符号链接。

```
#001  /*****
#002  * 函数名称:HelloDDKUnload
#003  * 功能描述:负责驱动程序的卸载操作
#004  * 参数列表:
#005      pDriverObject:驱动对象
#006  * 返回值:返回状态
#007  *****/
#008  #pragma PAGEDCODE
#009  VOID HelloDDKUnload (IN PDRIVER_OBJECT pDriverObject)
#010  {
#011      PDEVICE_OBJECT pNextObj;
#012      KdPrint(("Enter DriverUnload\n"));
#013      pNextObj = pDriverObject->DeviceObject;
#014      while (pNextObj != NULL)
#015      {
#016          PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
#017              pNextObj->DeviceExtension;
#018
#019          //删除符号链接
#020          UNICODE_STRING pLinkName = pDevExt->ustrSymLinkName;
#021          IoDeleteSymbolicLink(&pLinkName);
#022          pNextObj = pNextObj->NextDevice;
#023          IoDeleteDevice( pDevExt->pDevice );
#024      }
#025  }
```

此段代码可以在配套光盘中本章的 NT_Driver 目录下找到。

- 代码第 13 行, 由驱动对象得到设备对象。
- 代码 19~21 行, 删除设备对象的符号链接。

➤ 代码 22~23 行，遍历设备对象，并删除。

1.2.5 默认派遣例程

对设备对象的创建、关闭和读写操作，都被指定到这个默认的派遣例程中。由于这是一个最简单的演示程序，故只是简单地将其成功返回。后面的章节中，笔者将会扩充该例程。

```
#001  /*****
#002  * 函数名称:HelloDDKDispatchRoutine
#003  * 功能描述:对读 IRP 进行处理
#004  * 参数列表:
#005      pDevObj:功能设备对象
#006      pIrp:从 I/O 请求包
#007  * 返回值:返回状态
#008  *****/
#009  #pragma PAGEDCODE
#010  NTSTATUS HelloDDKDispatchRoutine(IN PDEVICE_OBJECT pDevObj,
#011      IN PIRP pIrp)
#012  {
#013      KdPrint(("Enter HelloDDKDispatchRoutine\n"));
#014      NTSTATUS status = STATUS_SUCCESS;
#015      // 完成 IRP
#016      pIrp->IoStatus.Status = status;
#017      pIrp->IoStatus.Information = 0;    // bytes xfered
#018      IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#019      KdPrint(("Leave HelloDDKDispatchRoutine\n"));
#020      return status;
#021  }
```

此段代码可以在配套光盘中本章的 NT_Driver 目录下找到。

- 代码 16 行，设置 IRP 的状态为成功。IRP 的讲解请参考第 4 章。
- 代码 17 行，设置操作的字节数为 0，这里无实际意义。
- 代码 18 行，指示完成此 IRP。
- 代码 20 行，成功返回。

1.3 HelloDDK 的编译和安装

本节会带领读者一步步地对 HelloDDK 进行编译和安装。编译和安装往往是初学者最先需要面对的问题，笔者将会从两个方面讲解编译过程。一是传统的用 DDK 编译环境编译，二是用 Visual C++（以下简称 VC）集成开发环境编译。

1.3.1 用 DDK 环境编译 HelloDDK

这种编译驱动的办法是 DDK 文档中所提倡的办法。此种方法需要编写一个编译脚本

Windows 驱动开发技术详解

文件，在这个脚本中描述了 DDK 驱动程序的源文件、用到的 lib 文件和 include 路径名、编译输出的目录和文件名等信息，具体介绍请参考第 3 章。编写此类脚本对于 Windows 程序员可能比较陌生，尤其是当源文件较多时，编写脚本文件可能显得更如麻烦。下一节将向读者介绍一种简单的用 Visual C++ 6.0 IDE（以下简称 VC IDE）环境编译驱动的方法。在源程序的相同目录下创建两个文件 `makefile` 和 `Sources`，这两个文件都是文本文件，内容如下。

Sources:

```
#001 TARGETNAME=HelloDDK
#002 TARGETTYPE=DRIVER
#003 TARGETPATH=OBJ
#004
#005 INCLUDES=$(BASEDIR)\inc;\
#006          $(BASEDIR)\inc\ddk;\
#007
#008 SOURCES=Driver.cpp\
```

此段代码可以在配套光盘中本章的 NT Driver 目录下找到。

- 第1行说明此驱动的名称。
- 第2行指明此驱动的类型为 NT 型驱动。
- 第3行设置编译输出目录。
- 第5~6行设置 include 目录。
- 第8行指定源文件。

编写完这两个脚本后，在 Windows 的开始菜单中选择“Windows XP Checked Build Environment”编译环境。这里选择的是 Checked 版本，而不是 Free 版本。两者的区别类似于 Win32 程序开发的 Debug 版本和 Release 版本，具体的差别详见第 3 章。

选择版本后，进入的是一个命令行方式的窗口。用 `cd` 命令进入需要编译的目录，然后输入“`build`”DDK 的编译环境会自动调用编译器进行编译，笔者计算机中的编译结果如图 1-4 所示。

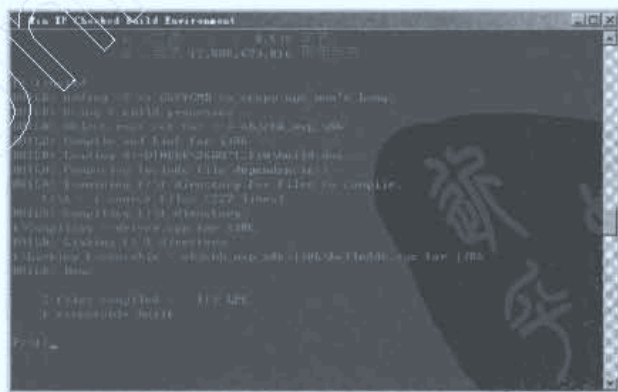


图 1-4 DDK 的编译环境

编译好的结果位于源码目录下的子目录 objchk_wxp_x86i386(如果读者是用 Windows 2000 DDK 编译的, 目录会稍有不同)中。编译出来的二进制文件为 HelloDDK.sys, 它不

像 exe 文件那样运行，而是必须通过特殊的加载方式加载，详见 1.3.3 节。

1.3.2 用 VC 集成开发环境编译 HelloDDK

初次学习编写 Windows 驱动程序的开发人员，大部分是熟悉 VC IDE 开发环境的 Windows 程序员。他们可能不喜欢用编辑脚本来描述一个工程，而是更希望在熟悉的 VC IDE 环境下编译，并且利用 VC IDE 可以方便快速地对代码进行交叉索引等操作。本节将向读者介绍此种方法。

(1) 用 VC 建立一个新工程。在 VC IDE 环境中选择“File”|“New”，弹出“New”对话框。在该对话框中，选择“Project”选项卡。在“Project”选项卡中，选择 Win32 Application（因为 VC 并没有提供驱动程序的工程，所以在 Win32 工程的基础上进行修改）。工程名为“DriverDev”，如图 1-5 所示。单击“OK”按钮，进入下一个对话框。在该对话框中，选择一个空的工程，如图 1-6 所示。

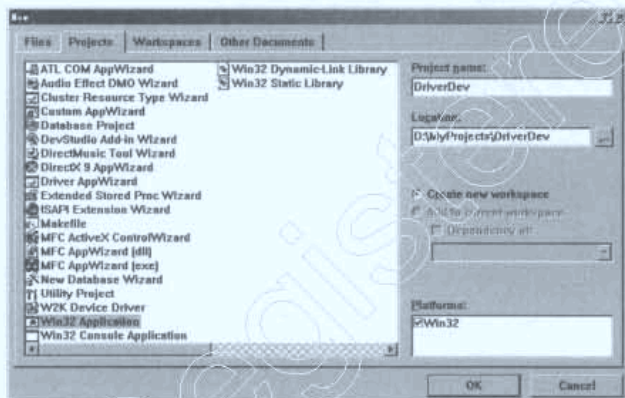


图 1-5 添加新工程

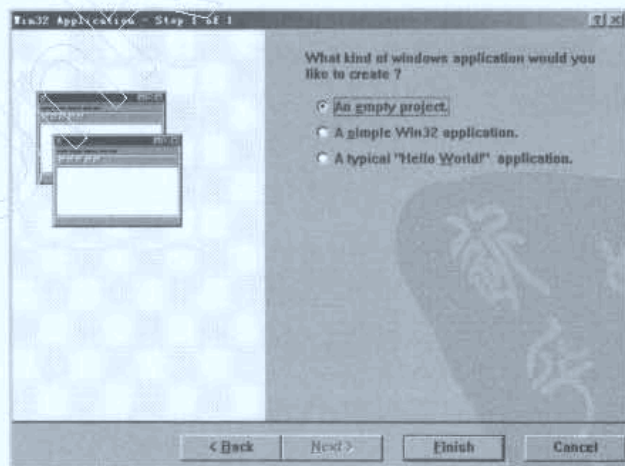


图 1-6 创建新工程

(2) 将两个源文件 Driver.h 和 Driver.cpp 拷贝到工程目录中，并添加到工程中，如图

Windows 驱动开发技术详解

1-7 所示。

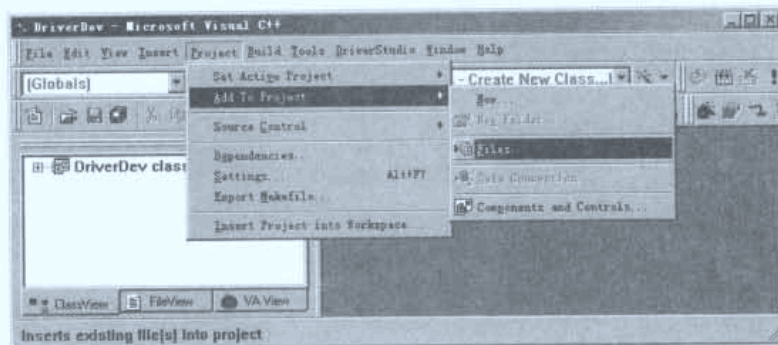


图 1-7 添加新文件到工程

(3) 增加新的编译版本, 去掉 Debug 和 Release 版本, 如图 1-8 和图 1-9 所示。

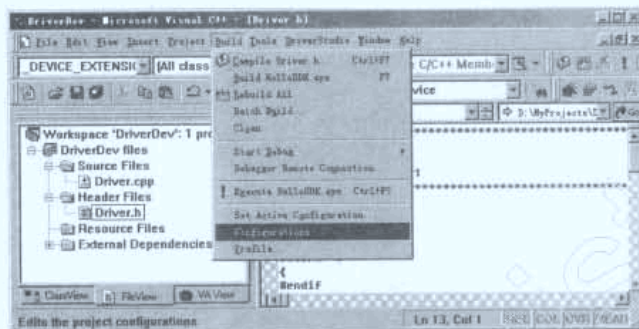


图 1-8 配置编译版本

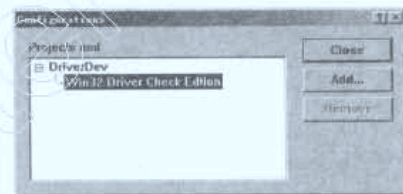


图 1-9 修改后的 check 版本

(4) 修改工程属性。选择“Project”|“Setting”, 或者直接按下 Alt+F7 键, 弹出“Project Settings”对话框。在对话框中, 选择“General”选项卡。将 Intermediate files 和 Output files 改为 MyDriver_Check, 如图 1-10 所示。

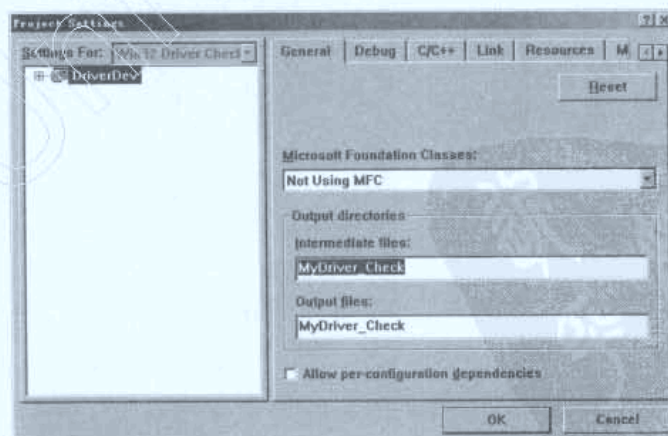


图 1-10 修改输出目录

选择 C/C++ 选项卡, 将原有的 Project Options 内容全部删除, 替换成如下内容, 如图

1-11 所示。

```
/nologo /Gz /MLd /W3 /WX /Z7 /Od /D WIN32=100 /D _X86_=1 /D WINVER=0x500 /D DBG=1
/Fo"MyDriver_Check/" /Fd"MyDriver_Check/" /FD /c
```

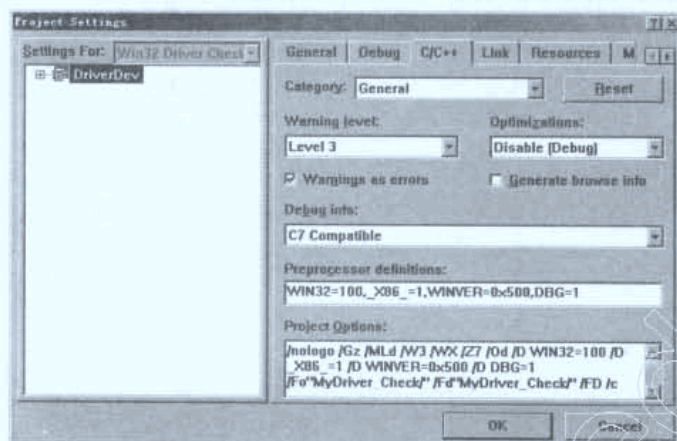


图 1-11 修改 C++选项卡

选择 Link 选项卡，将原有的 Project Options 内容全部删除，替换成如下内容，如图 1-12 所示。

```
ntoskrnl.lib /nologo /base:"0x10000" /stack:0x400000,0x1000 /entry:"DriverEntry"
/subsystem:console /incremental:no /pdb:"MyDriver_Check/HelloDDK.pdb" /debug
/machine:I386 /nodefaultlib /out:"MyDriver_Check/HelloDDK.sys" /pdbtype:sept
/subsystem:native /driver /SECTION:INIT_D /RELEASE /IGNORE:4078
```

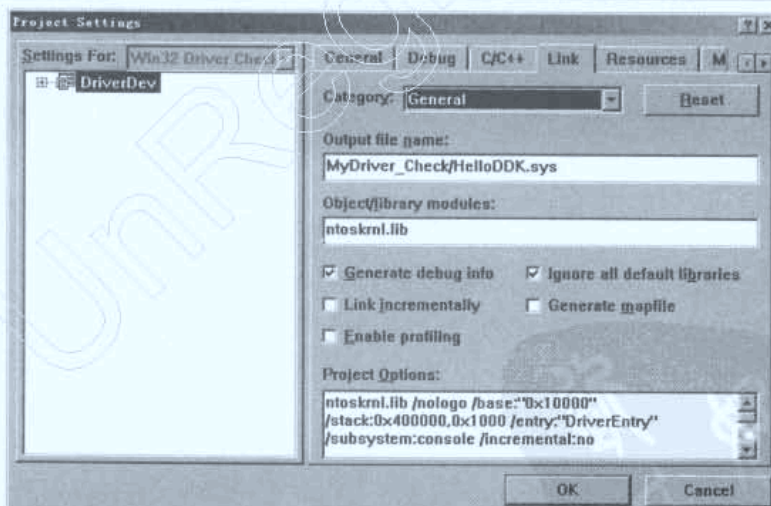
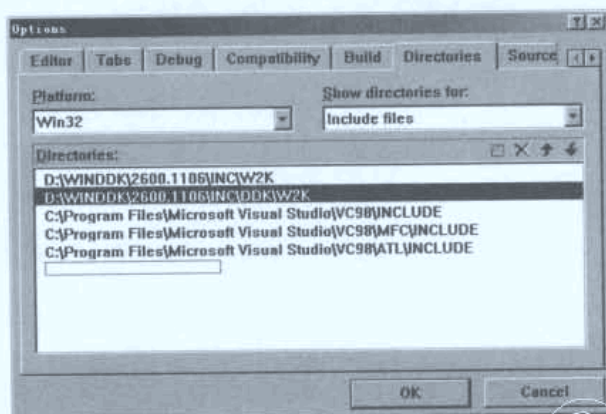


图 1-12 修改 link 选项卡

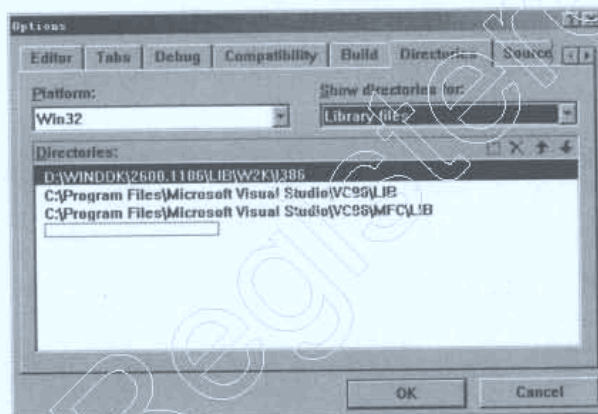
(5) 修改 VC 的 lib 目录和 include 目录。在 VC 中选择“Tools”|“Options”，在弹出的对话框中选择“Directories”选项卡。在“Show directories for”下拉菜单中选择“Include files”菜单。添加“D:\WINDDK\2600.1106\INC\W2K”和“D:\WINDDK\2600.1106\INC\DDK\W2K”，并将这两个目录置于最上，如图 1-13 (a) 所示。读者可将“D:\WINDDK\2600.1106”替换

Windows 驱动开发技术详解

成自己的 DDK 安装目录。这里应该选择 W2K 子目录，DDK 中还会有相应的 XP 子目录。因为 XP 驱动编译时需要高版本的 VC 编译器，所以这里用的是 W2K 子目录，它编译的代码完全可以应用于 Windows 2000 和 Windows XP 操作系统下。



(a)



(b)

图 1-13 设置 include 目录和设置 lib 目录

在“Show directories for”下拉菜单中选择“Library files”菜单，添加目录“D:\WINDDK\2600.1106\LIB\W2K\I386”，并置于最上端，如图 1-13 (b) 所示。

(6) 编译。按下 F7 键，和 1.3.2 节一样，同样会编译出一个 HelloDDK.sys 文件。

1.3.3 HelloDDK 的安装

NT 式驱动程序类似于 Windows 服务程序，以服务的方式加载在系统中。在第 3 章中，笔者将给出加载驱动的代码。为了简化步骤，这里利用一个叫做 DriverMonitor 的工具软件加载 HelloDDK。DriverMonitor 是 Compuware 公司开发的 DriverStudio 中的一个工具，笔者将在第 4 章对 DriverStudio 提供的一系列工具软件逐一介绍。笔者强烈推荐读者安装 DriverStudio，因为它提供的一系列工具对调试驱动非常有用。如果读者没

有安装 DriverMonitor, 请参阅第 3 章, 笔者将介绍如何编写一个 NT 式驱动程序的加载器。

运行 DriverMonitor, 选择 “File” | “Open Driver”, 将会弹出文件选择对话框, 选择编译好的 HelloDDK.sys 文件。再次选择 “File” | “Start Driver”。至此, NT 驱动加载成功, DriverMonitor 会报告加载情况, 如图 1-14 所示。

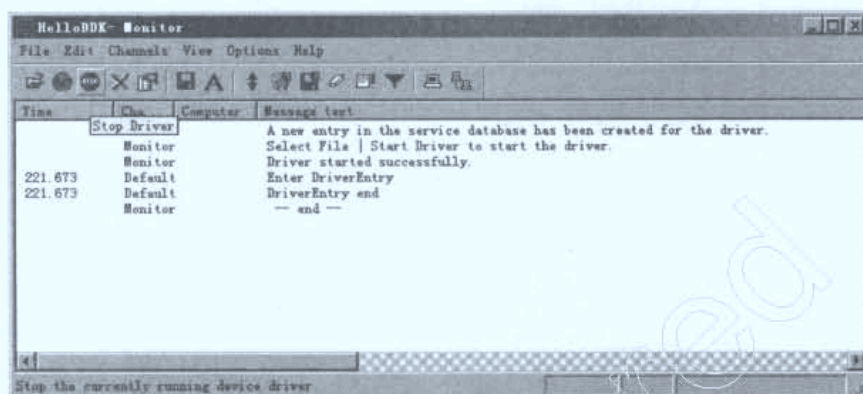


图 1-14 用 Driver Monitor 安装 HelloDDK

用 Driver Monitor 加载驱动时, 默认是加载一次。重新启动电脑后, 该驱动不会被加载。如果想在每次开机启动时自动加载, 需要修改设置。选择 “Edit” | “Properties”, 弹出如图 1-15 所示的对话框。在 “Start Type” 选择组中, 选择 “Automatic” 单选按钮。保存后, 就可以在每次开机启动时自动加载该驱动。

成功加载的驱动, 会出现在 Windows 的设备管理器中。默认情况下, NT 式的驱动程序是隐藏的, 在设备管理器中选择 “查看” | “显示隐藏设备”, 如图 1-16 所示。

在 “Not-Plug and Play Drivers” 的列表中, 会出现 HelloDDK 的驱动程序, 如图 1-17 所示。

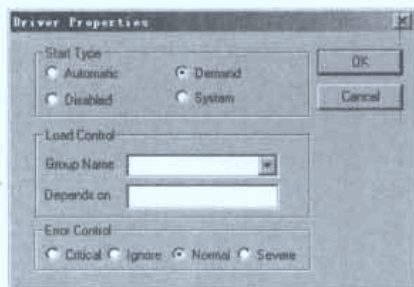


图 1-16 选择加载方式



图 1-17 在设备管理器中显示 HelloDDK 设备



图 1-17 在设备管理器中显示设备

1.4 第二个驱动程序 HelloWDM 的代码分析

本节编写的 HelloWDM 是基于 WDM 的驱动程序,和前面介绍的 HelloDDK 非常相似,且多了对即插即用功能的支持。NT 式驱动和 WDM 驱动的异同将在第 3 章介绍。

1.4.1 HelloWDM 的头文件

HelloWDM 的头文件主要是为了导入驱动程序开发所必需的 WDM.h 头文件,此头文件里包含了对 DDK 所有导出函数的声明。NT 式的驱动程序要导入的是 NTDDK.h,而 WDM 驱动要导入的头文件为 WDM.h。另外,此头文件中定义了几个标签,分别在程序中指明函数和变量分配在分页内存中或非分页内存中。最后,该头文件给出了此驱动程序的函数声明。

```
#001  /*****
#002  * 文件名称:HelloWDM.h
#003  * 作 者:张帆
#004  * 完成日期:2007-11-1
#005  *****/
#006
#007  #ifdef __cplusplus
#008
#009  extern "C"
#010  {
#011  #endif
```

```

#012 #include <wdm.h>
#013 #ifdef __cplusplus
#014 }
#015 #endif
#016
#017 typedef struct _DEVICE_EXTENSION
#018 {
#019     PDEVICE_OBJECT fdo;
#020     PDEVICE_OBJECT NextStackDevice;
#021     UNICODE_STRING ustrDeviceName; // 设备名
#022     UNICODE_STRING ustrSymLinkName; // 符号链接名
#023 } DEVICE_EXTENSION, *PDEVICE_EXTENSION;
#024
#025 #define PAGEDCODE code_seg("PAGE")
#026 #define LOCKEDCODE code_seg()
#027 #define INITCODE code_seg("INIT")
#028
#029 #define PAGEDDATA data_seg("PAGE")
#030 #define LOCKEDDATA data_seg()
#031 #define INITDATA data_seg("INIT")
#032
#033 #define arraysize(p) (sizeof(p)/sizeof((p)[0]))
#034
#035 NTSTATUS HelloWDMAddDevice(IN PDRIVER_OBJECT DriverObject,
#036                             IN PDEVICE_OBJECT PhysicalDeviceObject);
#037 NTSTATUS HelloWDMNp(IN PDEVICE_OBJECT fdo,
#038                     IN PIRP Irp);
#039 NTSTATUS HelloWMDispatchRoutine(IN PDEVICE_OBJECT fdo,
#040                                 IN PIRP Irp);
#041 void HelloWDMUnload(IN PDRIVER_OBJECT DriverObject);
#042
#043 extern "C"
#044 NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
#045                     IN PUNICODE_STRING RegistryPath);

```

此段代码可以在配套光盘中本章的 WDM_Driver 目录下找到。

- 代码 7~15 行, 包含头文件 wdm.h。和 HelloDDK 的头文件类似, 在引用时也使用 extern "C" 进行修饰, 为的是保持符号链接的正确性。注意, 此处包含的是 wdm.h 而非 ntddk.h, 这是 WDM 驱动程序和 NT 驱动程序最主要的区别。关于 WDM 驱动程序和 NT 驱动程序的不同, 请参考第 3 章。
- 代码 17~23 行, 定义了设备扩展结构体, 此结构体为本设备记录相关信息。
- 代码 25~31 行, 定义了分页内存、非分页内存和 INIT 段内存的标志, 以便在下面的程序中声明。
- 代码 35~45 行为代码声明。

1.4.2 HelloWDM 的入口函数

和 NT 式驱动程序一样, WDM 的入口函数地址同样是 DriverEntry, 且在 C++ 编译的时候需要用 extern "C" 修饰。

```

#001 /*****

```



```
#002 * 函数名称:DriverEntry
#003 * 功能描述:初始化驱动程序, 定位和申请硬件资源, 创建内核对象
#004 * 参数列表:
#005     pDriverObject:从 I/O 管理器中传进来的驱动对象
#006     pRegistryPath:驱动程序在注册表的中的路径
#007 * 返回值:返回初始化驱动状态
#008 *****/
#009 #pragma INITCODE
#010 extern "C" NTSTATUS DriverEntry(IN PDRIVER_OBJECT pDriverObject,
#011                                IN PUNICODE_STRING pRegistryPath)
#012 {
#013     KdPrint(("Enter DriverEntry\n"));
#014
#015     pDriverObject->DriverExtension->AddDevice = HelloWDMAddDevice;
#016     pDriverObject->MajorFunction[IRP_MJ_PNP] = HelloWDMNp;
#017     pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
#018     pDriverObject->MajorFunction[IRP_MJ_CREATE] =
#019     pDriverObject->MajorFunction[IRP_MJ_READ] =
#020     pDriverObject->MajorFunction[IRP_MJ_WRITE] = HelloWMDDispatchRoutine;
#021     pDriverObject->DriverUnload = HelloWDMUnload;
#022
#023     KdPrint(("Leave DriverEntry\n"));
#024     return STATUS_SUCCESS;
#025 }
```

- 代码第 9 行, 将此函数放在 INIT 段中, 当驱动加载结束后, 此函数就可以从内存中卸载掉。
- 代码 10~11 行, 定义驱动入口函数, 用 extern "C" 修饰。其中传进来两个参数, 第一个参数 pDriverObject 为驱动对象, 第二个参数 pRegistryPath 为此驱动的注册表路径。
- 代码第 15 行, 设置 AddDevice 回调函数, 此回调函数只出现在 WDM 驱动程序中, 而在 NT 式的驱动中没有此回调函数。此回调函数的作用是创建设备对象并由 PNP (即插即用) 管理器调用。
- 代码 16 行, 设置对 IRP_MJ_PNP 的 IRP 的回调函数。对 PNP 的 IRP 处理, 是 NT 式驱动和 WDM 驱动的重大区别之一。
- 代码 17~20 行, 设置常用 IRP 的回调函数。这里只是简单地指向了同一个默认函数 HelloWMDDispatchRoutine。
- 代码 21 行, 向系统注册卸载例程。在 WDM 程序中, 大部分卸载工作已不在此处理, 而是放在对 IRP_MN_REMOVE_DEVICE 的 IRP 的处理函数中处理。

1.4.3 HelloWDM 的 AddDevice 例程

在 WDM 的驱动程序中, 创建设备对象的任务不再由 DriverEntry 承担, 而需要驱动程序向系统注册一个称做 AddDevice 的例程。此例程由 PNP 管理器负责调用, 其函数主要职责是创建设备对象。HelloWDMAddDevice 例程有两个参数, DriverObject 和

PhysicalDeviceObject。DriverObject 是由 PNP 管理器传递进来的驱动对象，此对象是 DriverEntry 中的驱动对象。PhysicalDeviceObject 是 PNP 管理器传递进来的底层驱动设备对象，这个概念在 NT 式的驱动中是没有的。关于这个概念，请参考第 3 章。

```
#001  /*****
#002  * 函数名称:HelloWDMAddDevice
#003  * 功能描述:添加新设备
#004  * 参数列表:
#005      DriverObject:从 I/O 管理器中传进来的驱动对象
#006      PhysicalDeviceObject:从 I/O 管理器中传进来的物理设备对象
#007  * 返回值:返回添加新设备状态
#008  *****/
#009  #pragma PAGEDCODE
#010  NTSTATUS HelloWDMAddDevice(IN PDRIVER_OBJECT DriverObject,
#011                             IN PDEVICE_OBJECT PhysicalDeviceObject)
#012  {
#013      PAGED_CODE();
#014      KdPrint(("Enter HelloWDMAddDevice\n"));
#015
#016      NTSTATUS status;
#017      PDEVICE_OBJECT fdo;
#018      UNICODE_STRING devName;
#019      RtlInitUnicodeString(&devName,L"\\Device\\MyWDMDevice");
#020      status = IoCreateDevice(
#021          DriverObject,
#022          sizeof(DEVICE_EXTENSION),
#023          &(UNICODE_STRING)devName,
#024          FILE_DEVICE_UNKNOWN,
#025          0,
#026          FALSE,
#027          &fdo);
#028      if( !NT_SUCCESS(status))
#029          return status;
#030      PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION)fdo->DeviceExtension;
#031      pdx->fdo = fdo;
#032      pdx->NextStackDevice = IoAttachDeviceToDeviceStack(fdo, Physical
DeviceObject);
#033      UNICODE_STRING symLinkName;
#034      RtlInitUnicodeString(&symLinkName,L"\\DosDevices\\HelloWDM");
#035
#036      pdx->ustrDeviceName = devName;
#037      pdx->ustrSymLinkName = symLinkName;
#038      status = IoCreateSymbolicLink(&(UNICODE_STRING)symLinkName,&(UNICODE_
STRING)devName);
#039
#040      if( !NT_SUCCESS(status))
#041      {
#042          IoDeleteSymbolicLink(&pdx->ustrSymLinkName);
#043          status = IoCreateSymbolicLink(&symLinkName,&devName);
#044          if( !NT_SUCCESS(status))
#045          {
#046              return status;
#047          }
#048      }
#049
#050      fdo->Flags |= DO_BUFFERED_IO | DO_POWER_PAGABLE;
#051      fdo->Flags &= ~DO_DEVICE_INITIALIZING;
```



```
#052
#053     KdPrint(("Leave HelloWDMAddDevice\n"));
#054     return STATUS_SUCCESS;
#055 }
```

此段代码可以在配套光盘中本章的 WDM_Driver 目录中找到。

- 代码第 9 行, 用 `#pragma` 指明将此例程分派在分页内存中。
- 代码第 13 行, `PAGED_CODE` 是一个 DDK 提供的宏, 只在 `check` 版中有效。当此例程所在的中断请求级超过 `APC_LEVEL` 时, 会产生一个断言, 断言会使程序终止, 并报告出错地址。有关中断请求级的讲解, 请参考第 3 章。
- 代码 16~27 行, 创建设备对象, 此处和 `HelloDDK` 的创建方法一样。
- 代码 30 行, 得到设备对象扩展数据结构。
- 代码 31 行, 记录设备扩展中的功能设备对象为其刚才创建的设备。
- 代码 32 行, 用 `IoAttachDeviceToDeviceStack` 函数将此 `fdo` (功能设备对象) 挂载在设备堆栈上, 并将返回值 (下层堆栈的位置), 记录在设备扩展结构中。
- 代码 37~38 行, 创建设备的符号链接, 此处和 `HelloDDK` 方法一样。
- 代码 50~51 行, 设置设备为 `BUFFERED_IO` 设备, 并指明驱动初始化完成。
- 代码 54 行, 成功返回。

1.4.4 HelloWDM 处理 PNP 的回调函数

WDM 式驱动程序主要区别在于对 `IRP_MJ_PNP` 的 `IRP` 的处理。其中, `IRP_MJ_PNP` 会细分为若干个子类。例如, `IRP_MN_START_DEVICE`、`IRP_MN_REMOVE_DEVICE`、`IRP_MN_STOP_DEVICE` 等。本例中除了对 `IRP_MN_REMOVE_DEVICE` 做特殊处理, 其他 `IRP` 则做相同处理。

```
#001  /*****
#002  * 函数名称:HelloWDMpnp
#003  * 功能描述:对即插即用 IRP 进行处理
#004  * 参数列表:
#005      fdo:功能设备对象
#006      Irp:从 I/O 请求包
#007  * 返回值:返回状态
#008  *****/
#009  #pragma PAGEDCODE
#010  NTSTATUS HelloWDMpnp(IN PDEVICE_OBJECT fdo,
#011                      IN PIRP Irp)
#012  {
#013      PAGED_CODE();
#014
#015      KdPrint(("Enter HelloWDMpnp\n"));
#016      NTSTATUS status = STATUS_SUCCESS;
#017      PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
#018      PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
#019      static NTSTATUS (*fcntab[]) (PDEVICE_EXTENSION pdx, PIRP Irp) =
#020      {
```

```

#021     DefaultPnpHandler,           // IRP_MN_START_DEVICE
#022     DefaultPnpHandler,           // IRP_MN_QUERY_REMOVE_DEVICE
#023     HandleRemoveDevice,          // IRP_MN_REMOVE_DEVICE
#024     DefaultPnpHandler,           // IRP_MN_CANCEL_REMOVE_DEVICE
#025     DefaultPnpHandler,           // IRP_MN_STOP_DEVICE
#026     DefaultPnpHandler,           // IRP_MN_QUERY_STOP_DEVICE
#027     DefaultPnpHandler,           // IRP_MN_CANCEL_STOP_DEVICE
#028     DefaultPnpHandler,           // IRP_MN_QUERY_DEVICE_RELATIONS
#029     DefaultPnpHandler,           // IRP_MN_QUERY_INTERFACE
#030     DefaultPnpHandler,           // IRP_MN_QUERY_CAPABILITIES
#031     DefaultPnpHandler,           // IRP_MN_QUERY_RESOURCES
#032     DefaultPnpHandler,           // IRP_MN_QUERY_RESOURCE_REQUIREMENTS
#033     DefaultPnpHandler,           // IRP_MN_QUERY_DEVICE_TEXT
#034     DefaultPnpHandler,           // IRP_MN_FILTER_RESOURCE_REQUIREMENTS
#035     DefaultPnpHandler,           //
#036     DefaultPnpHandler,           // IRP_MN_READ_CONFIG
#037     DefaultPnpHandler,           // IRP_MN_WRITE_CONFIG
#038     DefaultPnpHandler,           // IRP_MN_EJECT
#039     DefaultPnpHandler,           // IRP_MN_SET_LOCK
#040     DefaultPnpHandler,           // IRP_MN_QUERY_ID
#041     DefaultPnpHandler,           // IRP_MN_QUERY_PNP_DEVICE_STATE
#042     DefaultPnpHandler,           // IRP_MN_QUERY_BUS_INFORMATION
#043     DefaultPnpHandler,           // IRP_MN_DEVICE_USAGE_NOTIFICATION
#044     DefaultPnpHandler,           // IRP_MN_SURPRISE_REMOVAL
#045 };
#046
#047     ULONG fcn = stack->MinorFunction;
#048     if (fcn >= arraysize(fcntab))
#049     {
#050         status = DefaultPnpHandler(pdx, Irp); // 对于未知的 IRP 类别, 我们让
                                                // DefaultPnpHandler 函数处理
#051         return status;
#052     } // 未知的 IRP 类别
#053
#054     #if DBG
#055         static char* fcname[] =
#056         {
#057             "IRP_MN_START_DEVICE",
#058             "IRP_MN_QUERY_REMOVE_DEVICE",
#059             "IRP_MN_REMOVE_DEVICE",
#060             "IRP_MN_CANCEL_REMOVE_DEVICE",
#061             "IRP_MN_STOP_DEVICE",
#062             "IRP_MN_QUERY_STOP_DEVICE",
#063             "IRP_MN_CANCEL_STOP_DEVICE",
#064             "IRP_MN_QUERY_DEVICE_RELATIONS",
#065             "IRP_MN_QUERY_INTERFACE",
#066             "IRP_MN_QUERY_CAPABILITIES",
#067             "IRP_MN_QUERY_RESOURCES",
#068             "IRP_MN_QUERY_RESOURCE_REQUIREMENTS",
#069             "IRP_MN_QUERY_DEVICE_TEXT",
#070             "IRP_MN_FILTER_RESOURCE_REQUIREMENTS",
#071             "",
#072             "IRP_MN_READ_CONFIG",
#073             "IRP_MN_WRITE_CONFIG",
#074             "IRP_MN_EJECT",
#075             "IRP_MN_SET_LOCK",
#076             "IRP_MN_QUERY_ID",
#077             "IRP_MN_QUERY_PNP_DEVICE_STATE",
#078             "IRP_MN_QUERY_BUS_INFORMATION",

```



```
#079         "IRP_MN_DEVICE_USAGE_NOTIFICATION",
#080         "IRP_MN_SURPRISE_REMOVAL",
#081     };
#082
#083     KdPrint(("PNP Request (%s)\n", fcnname[fcn]));
#084 #endif // DBG
#085
#086     status = (*fcntab[fcn])(pdx, Irp);
#087     KdPrint(("Leave HelloWDMpnp\n"));
#088     return status;
#089 }
```

- 代码 13 行，用 PAGED_CODE 宏确保该例程运行在低于 APC_LEVEL 的中断优先级的级别上。
- 代码 17 行，得到设备扩展结构。
- 代码 18 行，得到当前 IRP 的堆栈。设备堆栈是一个很复杂的概念，笔者将在第 4 章进行论述。
- 代码 19~52 行，将对应类别的即插即用 IRP 调用做不同的处理，并打印出调试信息。其中，IRP_MN_REMOVE_DEVICE 由 HandleRemoveDevice 处理，而其他 IRP 则由 DefaultPnpHandler 处理。

1.4.5 HelloWDM 对 PNP 的默认处理

除了 IRP_MN_STOP_DEVICE 以外，HelloWDM 对其他 PNP 的 IRP 做同样的处理，即直接传递到底层驱动，并将底层驱动的结果返回。

```
#001  /*****
#002  * 函数名称:DefaultPnpHandler
#003  ** 功能描述:对 PNP IRP 进行默认处理
#004  * 参数列表:
#005      pdx:设备对象的扩展
#006      Irp:从 I/O 请求包
#007  * 返回值:返回状态
#008  *****/
#009  #pragma PAGEDCODE
#010  NTSTATUS DefaultPnpHandler(PDEVICE_EXTENSION pdx, PIRP Irp)
#011  {
#012      PAGED_CODE();
#013      KdPrint(("Enter DefaultPnpHandler\n"));
#014      IoSkipCurrentIrpStackLocation(Irp);
#015      KdPrint(("Leave DefaultPnpHandler\n"));
#016      return IoCallDriver(pdx->NextStackDevice, Irp);
#017  }
```

这段代码可以在配套光盘中本章的 WDM_Driver 目录下找到。

- 代码 12 行，确保该例程处于 APC_LEVEL 之下。
- 代码 14 行，略过当前堆栈。
- 代码 16 行，用下层堆栈的驱动设备对象处理此 IRP。

1.4.6 HelloWDM 对 IRP_MN_REMOVE_DEVICE 的处理

对 IRP_MN_REMOVE_DEVICE 的处理类似于在 NT 式驱动中的卸载例程,而在 WDM 式的驱动中,卸载例程几乎不用做处理。

```
#001  /*****
#002  * 函数名称:HandleRemoveDevice
#003  * 功能描述:对 IRP_MN_REMOVE_DEVICE IRP 进行处理
#004  * 参数列表:
#005      fdo:功能设备对象
#006      Irp:从 I/O 请求包
#007  * 返回值:返回状态
#008  *****/
#009  #pragma PAGEDCODE
#010  NTSTATUS HandleRemoveDevice(PDEVICE_EXTENSION pdx, PIHP Irp)
#011  {
#012      PAGED_CODE();
#013      KdPrint(("Enter HandleRemoveDevice\n"));
#014
#015      Irp->IoStatus.Status = STATUS_SUCCESS;
#016      NTSTATUS status = DefaultPnpHandler(pdx, Irp);
#017      IoDeleteSymbolicLink(&(UNICODE_STRING)pdx->ustrSymLinkName);
#018
#019      //调用 IoDetachDevice()把 fdo 从设备栈中脱开
#020      if (pdx->NextStackDevice)
#021          IoDetachDevice(pdx->NextStackDevice);
#022      ,
#023      //删除 fdo
#024      IoDeleteDevice(pdx->fdo);
#025      KdPrint(("Leave HandleRemoveDevice\n"));
#026      return status;
#027  }
```

此段代码可以在配套光盘中本章的 WDM_Driver 目录中找到。

- 代码 15 行,设置此 IRP 的状态为顺利完成。
- 代码 16 行,调用默认的 PNP 的 IRP 的处理函数。
- 代码 17 行,删除此设备对象的符号链接。
- 代码 19~21 行,从设备堆栈中卸载此设备对象。
- 代码 24 行,删除设备对象。

1.4.7 HelloWDM 对其他 IRP 的回调函数

此处对创建、关闭、读写设备的默认处理同 HelloDDK 中,所以不重复说明。

```
#001  /*****
#002  * 函数名称:HelloWDMDispatchRoutine
#003  * 功能描述:对默认 IRP 进行处理
```



```
#004  * 参数列表:
#005      fdo:功能设备对象
#006      Irp:从 I/O 请求包
#007  * 返回值:返回状态
#008  *****/
#009  #pragma PAGEDCODE
#010  NTSTATUS HelloWDMDispatchRoutine(IN PDEVICE_OBJECT fdo,
#011                                  IN PIRP Irp)
#012  {
#013      PAGED_CODE();
#014      KdPrint(("Enter HelloWDMDispatchRoutine\n"));
#015      Irp->IoStatus.Status = STATUS_SUCCESS;
#016      Irp->IoStatus.Information = 0; // no bytes xfered
#017      IoCompleteRequest( Irp, IO_NO_INCREMENT );
#018      KdPrint(("Leave HelloWDMDispatchRoutine\n"));
#019      return STATUS_SUCCESS;
#020  }
```

1.4.8 HelloWDM 的卸载例程

由于 WDM 式的驱动程序将主要的卸载任务放在了对 IRP_MN_REMOVE_DEVICE 的处理函数中, 在标准的卸载例程几乎没有什么需要做的。在这里, 仅仅是打印几行调试信息。

```
#001  /*****
#002  * 函数名称:HelloWDMUnload
#003  * 功能描述:负责驱动程序的卸载操作
#004  * 参数列表:
#005      DriverObject:驱动对象
#006  * 返回值:返回状态
#007  *****/
#008  #pragma PAGEDCODE
#009  void HelloWDMUnload(IN PDRIVER_OBJECT DriverObject)
#010  {
#011      PAGED_CODE();
#012      KdPrint(("Enter HelloWDMUnload\n"));
#013      KdPrint(("Leave HelloWDMUnload\n"));
#014  }
```

此段代码可以在配套光盘中本章的 WDM_Driver 目录下找到。

1.5 HelloWDM 的编译和安装

HelloWDM 的编译和安装与 HelloDDK 的过程有一些不同, 尤其是安装过程。这主要是因为 HelloWDM 是即插即用式的驱动程序, 并且需要借助 INF 文件安装。

1.5.1 用 DDK 编译环境编译 HelloWDM

同 HelloDDK 一样, 首先介绍用 DDK 编译环境编译, 再介绍用 VC6 IDE 的编译方法。

编译 HelloWDM 需要编写两个脚本文件, makefile 和 Sources。makefile 同 HelloDDK 中的一样, 这里不再给出。Sources 稍有不同, 如下:

```
TARGETNAME=HelloWDM
TARGETTYPE=DRIVER
DRIVERTYPE=WDM
TARGETPATH=OBJ

INCLUDES=$(BASEDIR)\inc;\
          $(BASEDIR)\inc\ddk;\

SOURCES=HelloWDM.cpp\
```

此段代码可以在配套光盘中本章的 WDM_Driver 目录下找到。

将两个文件和源文件放在同一目录下, 进入 “Windows XP Checked Build Environments” 编译环境。用 cd 进入对应目录, 运行 build, 会编译出相应的 HelloWDM.sys 文件。

1.5.2 HelloWDM 的编译过程

HelloWDM 的编译过程和 HelloDDK 的方法类似

- ① 用 VC 建立一个新工程, 步骤同 HelloDDK。
- ② 将两个源文件 HelloWDM.h 和 HelloWDM.cpp 复制到工程目录中, 并添加到工程中, 步骤同 HelloDDK。
- ③ 增加新的编译版本, 去掉 Debug 和 Release 版本, 步骤同 HelloDDK。
- ④ 修改工程属性。在 VC 中选择 “Project” ! “Setting”。
- ⑤ 选择 General 选项卡。将 “Intermediate files” 和 “Output files” 改为 MyDriver_Check。
- ⑥ 选择 C/C++ 选项卡, 将原有的 Project Options 内容全部删除, 替换成如下内容:

```
/nologo /Gz /MLd /W3 /WX /Z7 /Od /D WIN32=100 /D _X86_=1 /D WINVER=0x500 /D DBG=1
/Fo"MyDriver_Check/" /Fd"MyDriver_Check/" /FD /c
```

- ⑦ 选择 Link 选项卡, 将原有的 Project Options 内容全部删除, 替换成如下内容:

```
wdm.lib /nologo /base:"0x10000" /stack:0x400000,0x1000 /entry:"DriverEntry"
/subsystem:console /incremental:no /pdb:"MyDriver_Check/HelloWDM.pdb" /debug
/machine:I386 /nodefaultlib /out:"MyDriver_Check/HelloWDM.sys" /pdbtype:sept
/subsystem:native /driver /SECTION:INIT,D /RELEASE /IGNORE:4078
```

- ⑧ 修改 VC 的 lib 目录和 include 目录, 同 HelloDDK。
- ⑨ 编译。按下 F7 键, 编译生成 HelloWDM.sys。

1.5.3 安装 HelloWDM

WDM 式驱动程序和 NT 式驱动程序的安装有很大的出入, 首先为了安装 HelloWDM 驱动程序, 需要为驱动程序编写一个 inf 文件。inf 文件描述了 WDM 驱动程序的操作硬件设备的信息和驱动程序的一些信息。下面是这个 inf 文件的代码, 请将这个文件和其他源文件放在同一个目录里。

Windows 驱动开发技术详解

HelloWDM.inf:

```
#001 ;: Win2K DDK 文档中有详细参考
#002
#003 ;----- 版本区域 -----
#004
#005 [Version]
#006 Signature="$CHICAGO$"
#007 Provider=Zhangfan_Device
#008 DriverVer=11/1/2007,3.0.0.3
#009
#010 ; 如果设备是一个标准类别, 使用标准类的名称和 GUID
#011 ; 否则创建一个自定义的类别名称, 并自定义它的 GUID
#012
#013 Class=ZhangfanDevice
#014 ClassGUID={EF2962F0-0D55-4bff-B8AA-2221EE8A79B0}
#015
#016
#017 ;----- 安装磁盘节 -----
#018
#019 ; 这些节确定安装盘和安装文件的路径
#020 ; 读者可以按照自己的需要修改
#021
#022 [SourceDisksNames]
#023 1 = "HelloWDM",Disk1,,
#024
#025 [SourceDisksFiles]
#026 HelloWDM.sys = 1,MyDriver_check,
#027
#028 ;----- ClassInstall/ClassInstall32 Section -----
#029
#030 ; 如果使用标准类别设备, 下面的是不需要的
#031
#032 ; 9X Style
#033 [ClassInstall]
#034 Addreg=Class_AddReg
#035
#036 ; NT Style
#037 [ClassInstall32]
#038 Addreg=Class_AddReg
#039
#040 [Class_AddReg]
#041 HKR,,, %DeviceClassName%
#042 HKR,,Icon,, "-5"
#043
#044 ;----- 目标文件节 -----
#045
#046 [DestinationDirs]
#047 YouMark_Files_Driver = 10,System32\Drivers
#048
#049 ;----- 制造商节 -----
#050
#051 [Manufacturer]
#052 %MfgName%=Mfg0
```

```

#053
#054 [Mfg0]
#055
#056 ; 在这里描述 PCI 的 VendorID 和 ProductID
#057 ; PCI\VEN_aaaa&DEV_bbbb&SUBSYS_cccccc&REV_dd
#058 ;改成自己的 ID
#059 %DeviceDesc%=YouMark_DDI, PCI\VEN_9999&DEV_9999
#060
#061 ;----- DDInstall Sections -----
#062 ; ----- Windows 9X -----
#063
#064 ; 如果在 DDInstall 中的字符串超过 19, 将会导致严重的问题
#065 ;
#066
#067 [YouMark_DDI]
#068 CopyFiles=YouMark_Files_Driver
#069 AddReg=YouMark_9X_AddReg
#070
#071 [YouMark_9X_AddReg]
#072 HKR,,DevLoader,,*ntkern
#073 HKR,,NTMPDriver,,HelloWDM.sys
#074 HKR, "Parameters", "BreakOnEntry", 0x00010001, 0
#075
#076 ; ----- Windows NT -----
#077
#078 [YouMark_DDI.NT]
#079 CopyFiles=YouMark_Files_Driver
#080 AddReg=YouMark_NT_AddReg
#081
#082 [YouMark_DDI.NT.Services]
#083 Addservice = HelloWDM, 0x00000002, YouMark_AddService
#084
#085 [YouMark_AddService]
#086 DisplayName = %SvcDesc%
#087 ServiceType = 1 ; SERVICE_KERNEL_DRIVER
#088 StartType = 3 ; SERVICE_DEMAND_START
#089 ErrorControl = 1 ; SERVICE_ERROR_NORMAL
#090 ServiceBinary = %10%\System32\Drivers\HelloWDM.sys
#091
#092 [YouMark_NT_AddReg]
#093 HKLM, "System\CurrentControlSet\Services\HelloWDM\Parameters",\
#094 "BreakOnEntry", 0x00010001, 0
#095
#096
#097 ; ----- 文件节 (common) -----
#098
#099 [YouMark_Files_Driver]
#100 HelloWDM.sys
#101
#102 ;----- 字符串节-----
#103
#104 [Strings]
#105 ProviderName="Zhangfan."
#106 MfgName="Zhangfan Soft"
#107 DeviceDesc="Hello World WDM!"

```


Windows 驱动开发技术详解

```
#108 DeviceClassName="Zhangfan_Device"  
#109 SvcDesc="Zhangfan"
```

此段代码可以在配套光盘中本章的 WDM_Driver 目录下找到。

HelloWDM 是个虚拟设备，安装需要如下方法。

① 首先，在第一次安装 HelloWDM 驱动程序时，进入控制面板，选择添加硬件。系统会自动检索是否有新设备插入，并弹出对话框询问是否将硬件连接到计算机，选择是，如图 1-18 所示。

② 在弹出的下一个对话框中选择“添加新的硬件设备”，并单击“下一步”按钮，如图 1-19 所示。

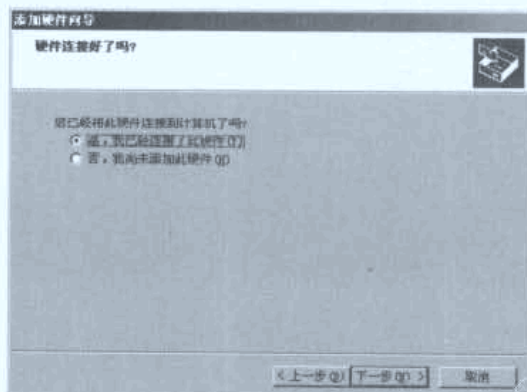


图 1-18 添加虚拟设备

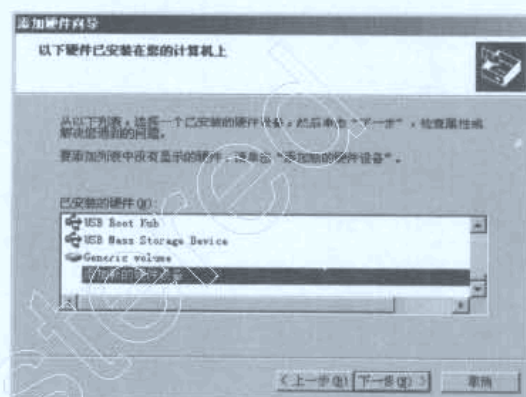


图 1-19 选择添加新硬件

③ 在弹出的下一个对话框中选择“安装我手动从列表选择的硬件（高级）”，并单击“下一步”按钮，如图 1-20 所示。

④ 在弹出的下一个对话框中选择显示所有设备，并单击“下一步”按钮，如图 1-21 所示。

⑤ 在弹出的下一对话框中选择“从磁盘安装”并选择 inf 文件，如图 1-22 所示。

⑥ 最后系统提示安装成功，如图 1-23 所示。

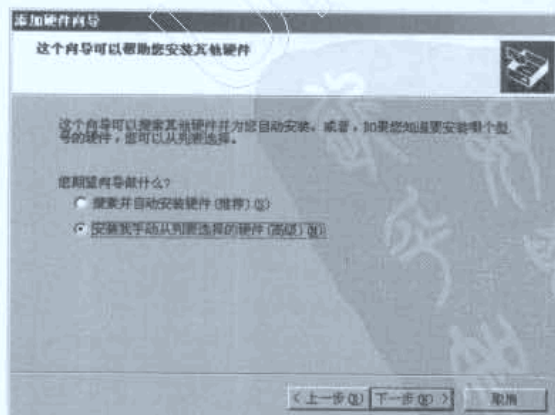


图 1-20 手动安装设备

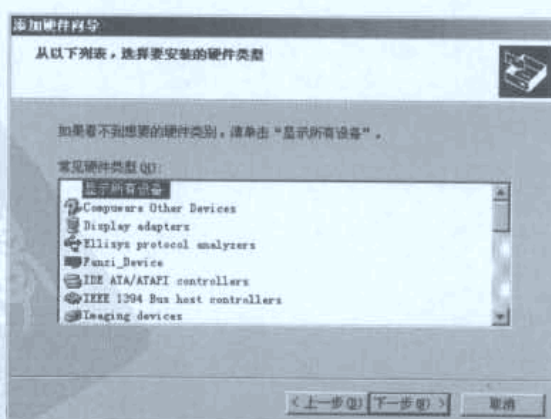


图 1-21 选择显示所有设备

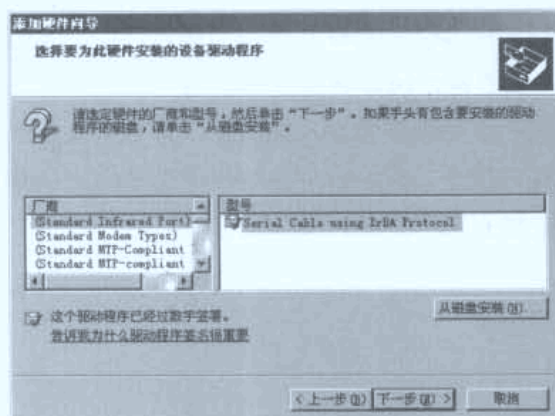


图 1-22 从磁盘安装设备驱动

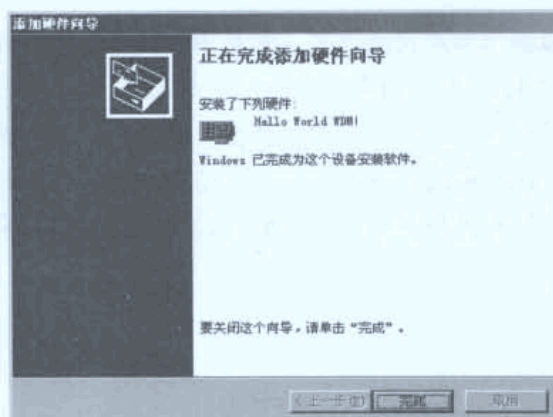


图 1-23 安装完毕

和 HelloDDK 一样，也可以在设备管理器中找到这个虚拟的设备，如图 1-24 所示。

这种安装 WDM 驱动程序的方法，过于烦琐，同时需要等待系统枚举设备，因此会等待很久的时间。在这里，笔者向读者介绍一个快速安装 WDM 程序的方法。使用一个叫做 EzDriverInstaller 的工具，这个工具也是 DriverStudio 自带的一个工具软件。打开 EzDriverInstaller，选择“File”|“Open”，在弹出的对话框中，选择需要安装的 inf 文件，单击“Add New Device”按钮，如图 1-25 所示。很快地，驱动程序就被加载了，EzDriverInstaller 还提供了删除驱动程序功能、开启/关闭驱动功能、屏蔽驱动功能、重启驱动功能等。在调试驱动的时候，EzDriverInstaller 不失为一个很好的加载 WDM 驱动程序的工具。



图 1-24 在设备管理器中显示 HelloWDM 设备



图 1-25 用 EzDriverInstaller 安装 HelloWDM 驱动

1.6 小结

本章笔者带领读者编写了两个非常简单的驱动程序——HelloDDK 和 HelloWDM，并详细说明了这两个驱动程序的编译过程、安装过程。笔者在初学 Windows 驱动程序开发的时候，对编译的方法摸索了很久，尤其是用 VC 编译驱动程序（其实用 VC 还是 DDK 环

Windows 驱动开发技术详解

境编译本质完全一样，都是用 `cl.exe` 进行编译)。这方面的资料非常少，就连微软官方的 DDK 文档中都很少有介绍。通过本章的学习，读者能很快地将这两个驱动程序编译并且顺利安装。在以后的章节里，笔者会陆续对这两个驱动程序做深入的剖析，并且重复利用这两个驱动程序的框架。驱动程序的代码量往往都很小，在这两个驱动程序的基础上，就能编写出很多有意思的驱动程序来。相信读者已经进入 Windows 驱动开发的大门了。

第 2 章 Windows 操作驱动的基本概念

在上一章节中，笔者介绍了两个简单驱动程序的编写及安装过程，从中可以看出编写 Windows 驱动程序并不十分复杂。然而深刻理解上述两个驱动程序的各个细节，却不是那么容易，因为这涉及 Windows 操作系统内核的各个方面。对于 Windows 驱动程序的介绍，不可避免地需要了解 Windows 操作系统的底层，这是因为驱动程序都加载在 Windows 内核模式下。

本章主要介绍 Windows 操作系统内核的基本概念，同时还介绍了应用程序和驱动程序之间的通信方法。

2.1 Windows 操作系统概述

驱动程序被操作系统加载在内核模式下，它与 Windows 操作系统内核的其他组件进行密切交互。对 Windows 操作系统内核的各个组件深入理解，有助于读者编写出性能优良的驱动程序。

2.1.1 Windows 家族

Windows 操作系统有着庞大的家族体系，其中主要分为两大分支。一个分支是基于 Windows 95 系列的 Windows，这包括 Windows 95、Windows 98、Windows ME（千禧年版）等，本书将这一系列统称为 Win 9X 系列。另一个分支是基于 NT 技术系列的 Windows，这包括 Windows NT 3.1、Windows NT 3.5、Windows NT 4.0、Windows 2000、Windows XP、Windows Server 2003 等，本书将这一系列统称为 Win NT 系列。

Win 9X 系列作为从 16 位操作系统到 32 位操作系统的过渡产品，基本已经被淘汰。

Windows 驱动开发技术详解

所以本书以讲解 Win NT 系列的驱动程序开发为主。同时,如不作特殊说明,本书所指的 Windows 即 Win NT 系列的 Windows。表 2-1 列出 Win NT 系列的发布日期和内部版本号。

表 2-1 Windows 发布的版本

产品名称	内部版本号	发布时间
Windows NT 3.1	3.1	1993 年 7 月
Windows NT 3.5	3.5	1994 年 9 月
Windows NT 3.51	3.51	1995 年 5 月
Windows NT 4.0	4.0	1996 年 7 月
Windows 2000	5.0	1999 年 12 月
Windows XP	5.1	2001 年 8 月
Windows Server 2003	5.2	2003 年 3 月

从表 2-1 可知,Windows 2000 的内部版本号为 5.0,Windows XP 的内部版本号是 5.1,Windows Server 2003 的内部版本号是 5.2。可以想象,从 Windows 2000 到 Windows XP,再到后来的 Windows Server 2003,这些操作系统的内核结构变化并不大。而这些 Windows 版本的差异主要集中在用户界面和易用性上。因此对 Windows 2000 内核的研究,对于整个 NT 系列家族的 Windows 就有了通用性。

本章所讲解的 Windows 的内核结构和特性,都是基于 Windows 2000 而言的。同时对其他更高版本的 Windows,也有着同样的适用性。同时,对于本书所介绍的驱动程序开发,也大部分适用于从 Windows 2000 到 Windows Server 2003 的各个平台上。其中有细微不同的地方,本书会予以说明。

2.1.2 Windows 特性

当 Windows 发展到 Windows 2000 时,已经比先前版本的 Windows 有了质的飞跃。Windows 2000 在设计上是十分先进的,微软公司随后推出的 Windows XP 和 Windows 2003,其基本架构上都没有太大的变化。在 Windows 2000 设计之初,微软公司的架构师制定了如下的设计目标。

1. 可移植性

可移植性是指只需要少量修改,操作系统即可在尽可能多的硬件平台上运行。Windows 必须能运行在多种硬件体系结构上,必须根据市场的需要,相对容易地移植到新的体系结构上。

为了实现移植性,Windows 被设计成为软件分层的体系结构。和硬件紧密关联的只有硬件抽象层(HAL)。而操作系统的其他重要组件几乎和硬件没有关联性,这就使操作系统的大部分不依赖于特定硬件。当需要将 Windows 移植到别的平台时,只需修改相关的硬件抽象层。

为了将 Windows 移植到更多的硬件上,Windows 在设计的时候引入了面向对象编程的

思想。操作系统抽象成各个组件，其中只有很少的组件会涉及具体硬件。这在某种程度上，保证了可移植性。当需要移植到新的硬件上时，只需重新改写与硬件相关联的那个部分。实际上，Windows 在从 32 位 CPU 移植到 64 位 CPU 的时候，只改动了相当少的代码。

2. 兼容性

兼容性是指让应用程序尽可能多的在各个版本上运行。Windows 家族体系庞大，应该让应用程序可以运行于各个版本的 Windows。这需要 Windows 在各个版本中有非常好的兼容性，这体现在 Windows 有着一致的 Win32 API 接口。尽管各个版本 Windows 的 API 实现方法不尽相同，但 API 保持着一贯的名称和调用接口。这保证了绝大多数的应用程序，不用重新编译即可在不同版本的 Windows 上运行，即能保证二进制级的兼容性。

另外，为了支持以前的部分 DOS 程序和 16 位 Windows 的程序，Windows 最大限度地保留了对原有 DOS 函数的支持和对旧的 16 位 API 的支持。同时，Windows 引入了环境子系统概念。不同的环境子系统，向各自的应用程序提供相应的 API 支持。

3. 健壮性和可靠性

Windows 的健壮性和可靠性主要源于用户模式和内核模式的划分。应用程序运行在特权级别低的用户模式。在用户模式下，所有的错误操作都会被操作系统侦测到，并给予提示。而操作系统的大部分核心代码，运行于特权级别高的内核模式下。操作系统是经过严格测试过的，可以保证正确性。对于任何涉及操作硬件的操作，应用程序都无法在用户模式下完成，必须通过对内核模式中的系统调用来完成。关于内核模式和用户模式的讲述，会在下一节有更深入的介绍。

Windows 的健壮性同样来自于自身的分层设计，且每层的特权不同。处在最上层的应用程序，对于操作系统的操作仅仅限于对 API 的操作。API 是操作系统提供给应用程序的唯一接口。当应用程序想访问硬件设备时，必须向操作系统提出申请。操作系统会检验应用程序通过 API 提出的请求，并校验请求的参数。当认为参数非法时，会返回一个错误，这大大提高了系统的健壮性和可靠性。

4. 可扩展性

可扩展性是指操作系统应该易于增加新的功能和支持新的硬件，并且对已有代码的影响达到最小。在 Windows 2000 中，一个很重要的设计就是内核从执行体组件中分离了出来。操作系统内核只负责关于线程的调度工作。线程运行在自己的线程上下文中。线程上下文指的是 CPU 寄存器的状态，比如堆栈寄存器、指令寄存器等，还包括线程 ID、线程的优先权、线程的本地存储等线程相关的信息。

内核的主要作用是调度线程活动，而其他操作系统的组件，如内存管理组件、进程管理组件等作为独立于内核的组件，统称为执行程序组件。执行程序组件按照模块化的方法设计，在需要改进的时候可以修正或者增加执行程序组件。

保持内核自身的短小和执行组件的模块化，保证了 Windows 2000 的可扩展性。

5. 性能

Windows 在总体设计上是基于分层的，各个层次之间的调用会从某种程度上，带来一些性能上的损失。然而这点性能上的损失，可以在其他地方弥补过来。例如，在硬件抽象层，功能调用是通过宏来调用的，而不是通过函数调用。

另外，Windows 的 I/O 操作是基于异步设计的。也就是线程在发起一个 I/O 操作的时候，可以不等待这个 I/O 操作完成，就发起另外的 I/O 操作请求。这样 CPU 不会将时间浪费在等待 I/O 操作完成上。

同时，Windows 是基于多进程和多线程的，并且尽可能使多个任务并行执行。在内核调度线程的时候，应该尽可能多的让各个线程保持看上去同时运行，而不是处于等待状态，这样会最小化处理器等待的时间。

最后，Windows 是一个完全支持异步操作的操作系统。也就是当用户提出一个 I/O 请求的时候，不用等到 I/O 结束，即可立即返回，进而去执行其他操作。这样的好处是，CPU 不用白白地等待 I/O 操作完成，而可以去做更有意义的事情，这大大地提高了操作系统针对 I/O 的吞吐能力。在读者编写驱动程序的时候，也要注意尽量让驱动程序支持异步操作，提高 I/O 吞吐能力。

2.1.3 用户模式和内核模式

Windows 从总体上分为内核模式（Kernel Mode）和用户模式（User Mode）。谈到操作系统的内核模式和用户模式，一般会与 CPU 的特权层联系起来。CPU 一般会有多个特权层，例如，Intel 的 386 CPU 就有 4 个特权层，分别是第 0 环（Ring 0），第 1 环（Ring 1），第 2 环（Ring 2），第 3 环（Ring 3）。其中，Ring 0 的特权最高，也就是可以执行任意代码，而 Ring 3 特权最低，只能执行有限的代码。其他 CPU 也有类似的特权级别。

Windows 将内核模式运行在 CPU 的 Ring0 层，而将用户模式运行在 CPU 的 Ring3 层，如图 2-1 所示。Ring0 层和 Ring3 层是 CPU 上的概念，而用户模式和内核模式是操作系统上的概念。

Windows 的核心代码运行在内核模式下，而非核心代码运行在用户模式下。运行在内核模式下的 Windows 的核心组件是安全的，且不会受到恶意攻击，所以这些核心组件可以进行所有权限的操作。而运行在用户模式下的应用程序，是不安全且容易受到攻击的，所以用户模式下的应用程序的权限是受到限制的。如果应用程序想进行一些敏感操作，如直接访问物理内存、物理端口，应用程序需要向内核模式下的组件提出请求。

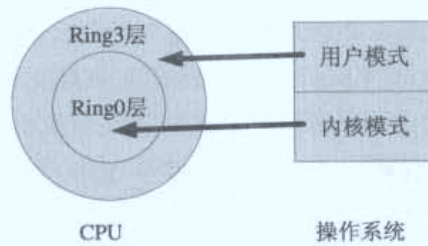


图 2-1 CPU 特权层和操作系统用户模式、内核模式的关系

本书所介绍的 Windows 驱动程序，都是运行在内核模式下的。编写驱动程序主要是为了操作硬件设备。这些对硬件设备的操作包括访问物理映射内存、设备端口等。

在早期的操作系统（如 DOS）中，没有用户模式和内核模式之分，所有程序都运行在 Ring 0 层。应用程序可以不用编写驱动而直接操作硬件设备。然而在 Windows 操作系统中，对硬件的操作必须通过驱动程序来完成。驱动程序相当于 Windows 内核的“补丁”，针对不同的硬件设备会有不同的“补丁”。

驱动程序运行在内核模式下，拥有操作系统的最高权限。因此，编写驱动程序时需要格外小心。在用户模式下的各种保护措施，在内核模式下都没有。例如，在应用程序中对空指针操作时，操作系统会弹出对话框，提示这是非法操作并终止进程。然而在驱动程序中对空指针进行操作时，操作系统不会检测这个操作是否非法，而是直接导致操作系统的崩溃。

很多计算机病毒、木马程序或者恶意软件，利用驱动程序运行在内核模式的特点，将自己以驱动程序的形式加载到内核中，从而获取操作系统的最高权限。应当指出，程序员在编写驱动程序的时候，应该避免漏洞，如缓冲区溢出漏洞。在 Windows 中，用户模式和内核模式的切换是通过软件中断实现的。

下面做一个实验，查看 Windows 运行内核模式和用户模式的运行情况。在控制面板中，选择“管理工具”，然后选择“性能”图标。此工具可以查看 Windows 运行性能。

如图 2-2 所示，性能工具默认会加入三个查看项目。将此三条删除，并添加查看内核模式和用户模式的项目，如图 2-3 所示。

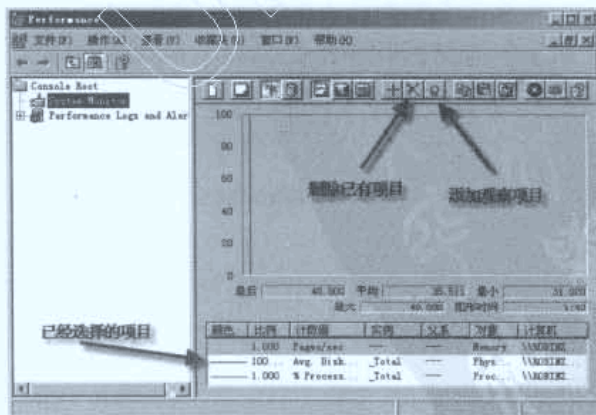


图 2-2 Windows 提供的性能工具

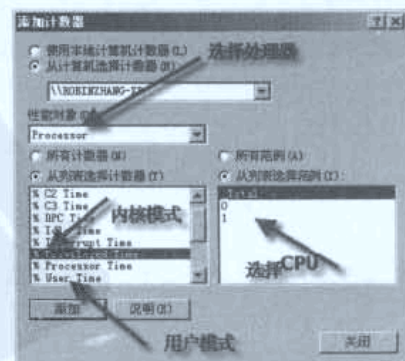


图 2-3 添加性能查看选项

Windows 驱动开发技术详解

在性能对象中，选择下拉菜单中的“处理器”选项。再从列表中选择“计数器”，并添加“Privilege Time”和“User Time”选项。这样会分别显示出内核模式和用户模式下所占用的 CPU 时间。笔者用的电脑配备双核 CPU，因此这里会显示出 2 个 CPU。

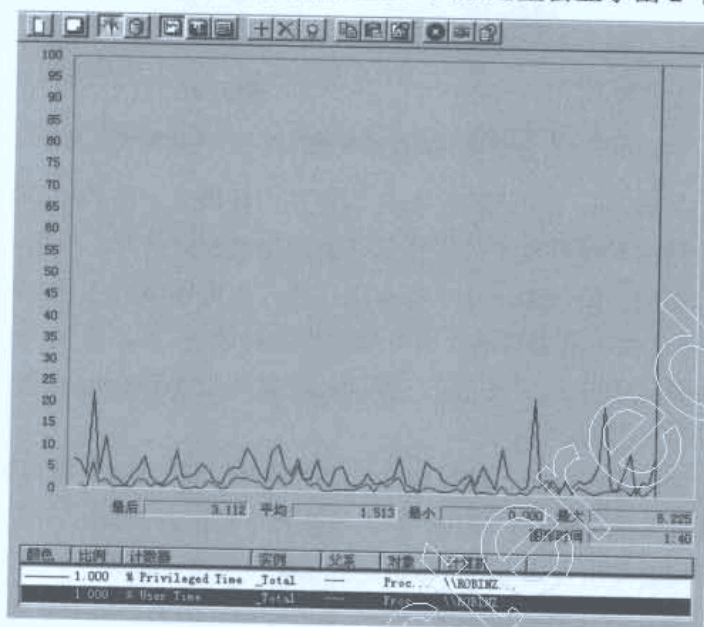


图 2-4 查看 Windows 在用户模式和内核模式的运行情况

如图 2-4 所示，两条曲线分别代表了 CPU 在用户模式和内核模式的运行情况。这个工具可以方便地统计 Windows 操作系统的运行性能。同时这个工具也可以查看驱动程序在操作系统中的运行负荷，例如，当发现某一进程对驱动程序操作后，其内核模式的运行时间陡然上升，则说明驱动程序消耗了大量的内核资源。

2.1.4 操作系统与应用程序

在多数的现代操作系统中，应用程序和操作系统是相互隔离的。操作系统的核心代码运行在特权模式下，即内核模式。而应用程序运行在非特权模式下，即用户模式。操作系统和应用程序的关系类似于服务器（Server）和客户端（Client）的关系，如图 2-5 所示。

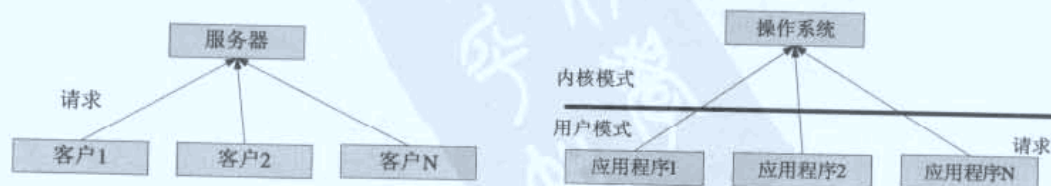


图 2-5 操作系统与应用程序的关系

在服务器和客户端的关系中，多个客户端对应一个服务器。各个客户端会并发的向服

务器发起请求，服务器会判断请求的合理性，从而完成请求或者拒绝请求。

操作系统和应用程序的关系类似这种关系。在操作系统上，会同时运行多个应用程序。每个应用程序向操作系统提出请求，例如，分配回收内存、读写文件、操作硬件等。优秀的操作系统会调度好每一个应用程序，并根据请求快速地做出反应，即拥有很大的吞吐能力。并且操作系统会根据请求，做出合法性的判断，拒绝一些危险的操作，如空指针读写操作等。

对计算机任何硬件设备的访问，例如，访问设备的映射内存、物理端口、中断等，必须通过操作系统的调用。早期的 DOS 或者 Win 9X 的很多程序，它们需要直接访问硬件，（例如，某些游戏需要直接对某段显卡内存操作，或者某些磁盘软件需要直接改写物理扇区等），因此这些软件是无法在 Windows 2000 及以后版本的 Windows 上正确运行的。

2.2 操作系统分层

现代的操作系统都是基于分层设计思路设计的。Windows 总体上分为用户模式和内核模式，内核模式的接口对用户模式的应用程序提供服务。在内核模式下，各个模块各司其职，并且有良好的机制保证各个模块间进行正确的通信。

2.2.1 Windows 操作系统总体架构

Windows 既可以运行在单 CPU 的 PC 上，也可以运行在对称的多 CPU 的 PC 上。“对称”多 CPU 不同于“主从”多 CPU，在“对称”多 CPU 的 PC 上，每个 CPU 将负载平分。而在“主从”多 CPU 的 PC 上，“主”CPU 负责调度“从”CPU。

现在操作系统的总体设计是基于分层设计思路的，每层由若干个组件组成。操作系统作为一个整体，它的运行依赖于上层组件向下一层组件的调用。每层的组件有固定的接口，靠近底层的组件有更高的操作权限，靠近上层的组件将任务转化成对底层组件的调用。

Windows 的设计思想是将内核设计的尽可能小，并且采用“客户端—服务器”的结构。操作系统各个组件或者模块是通过消息进行通信的。

如图 2-6 所示，这是 Windows 操作系统的简化结构。由于是简图，所以很多组件的细节没有表现出来。后面将陆续介绍有关模块的功能与作用。

在图 2-6 中，有一条粗线将 Windows 操作系统划分为两个部分，即用户模式和内核模式。在用户模式下，应用程序调用各自子系统的 API 接口。其中，子系统包括 Win32 子系统、OS/2 子系统、POSIX 子系统等。这些子系统是为了 Windows 可以更好地兼容旧的 16 位程序和移植其他系统的程序而设计的。遗憾的是，除了 Win32 子系统外，其他子系统很少被用到。Win32 子系统是 Windows 最主要的子系统，其他子系统都是通过 Win32 子系

统的接口来实现的。

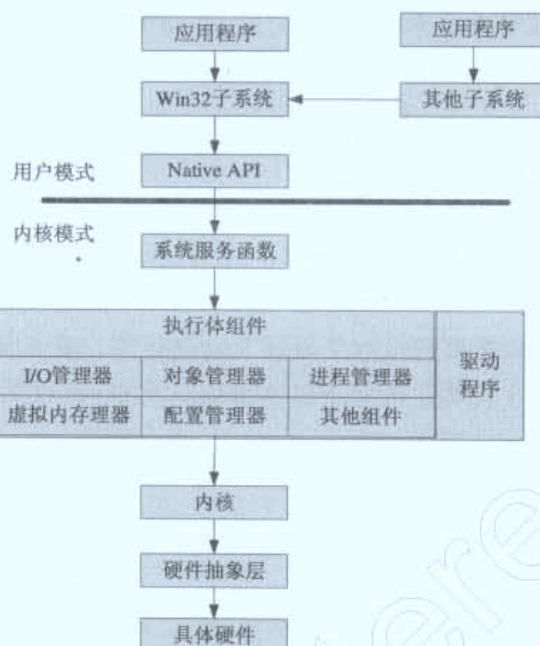


图 2-6 Windows 架构简图

Win32 子系统将 API 函数转化为 Native API 函数。在 Native API 接口中，已经没有了子系统的概念，它将这种调用转化为系统服务函数的调用。其中，Native API 穿过了用户模式和内核模式的界面，达到了内核模式。系统服务函数通过 I/O 管理器将消息传递给驱动程序。

在内核模式下，执行体组件提供了大量的内核函数供驱动程序调用。内核主要负责进程、线程的调度情况。驱动程序通过硬件抽象层与具体硬件进行操作。

2.2.2 应用程序与 Win32 子系统

在 Windows 的设计之初，Windows 的设计者们为了将其他操作系统的程序方便地移植到 Windows 上，因此设计了子系统。

每个应用程序在编译和链接的时候，都会指明该应用程序属于哪个子系统。例如，“/subsystem:windows”是指这个应用程序是 Win32 子系统的程序，该应用程序调用的是 Win32 子系统的 API。

Win32 子系统是最纯正的 Windows 子系统，提供了大量的 API 函数。程序员只要熟练使用这些 Win32 的 API，就可以编写出 Windows 应用程序。当然，程序员也可以考虑使用第三方的类库，如 VC 提供的 MFC，但这只不过是在应用程序和 Win32 子系统中间加了一层封装而已，没有本质的区别。Windows API 分为三类，分别是 USER 函数、GDI 函数

和 KERNEL 函数。

- USER 函数：这类函数管理窗口、菜单、对话框和控件。
- GDI 函数：这类函数在物理设备上执行绘图操作。
- KERNEL 函数：这类函数管理非 GUI 资源，例如，进程、线程、文件和同步服务等。

读者可以发现 Windows 系统目录中有对应的三个系统文件，分别是 USER32.dll、GDI32.dll 和 KERNEL32.dll。这三个文件提供了以上三类 API 的接口。当应用程序加载的时候，操作系统除了将应用程序加载到内存中，同时将以上三个 DLL 文件加载到内存中。

Windows SDK 中提供了一个工具叫 Dependency，它可以很方便地查看应用程序用到了哪些 DLL 文件，如图 2-7 所示。

Dependency 工具列出了这个应用程序所需要加载的 DLL 文件，并且可以递归的显示出加载某个 DLL 文件需要加载其他的 DLL 文件。一个 Windows 应用程序，不管需要加载什么 DLL 文件，都会依赖以上三个 DLL 文件。

笔者想更深入地介绍一下这三个 DLL 文件。在 Windows 设计的初期，这三个 DLL 文件担负了 Win32 子系统的全部“重担”，所有 Win32 API 的实现都是在这三个 DLL 中。然而在 Windows 2000 中，这三个 DLL 文件的实现全部被移入了内核模式，而这三个 DLL 只剩下了“躯壳”，如图 2-8 所示。

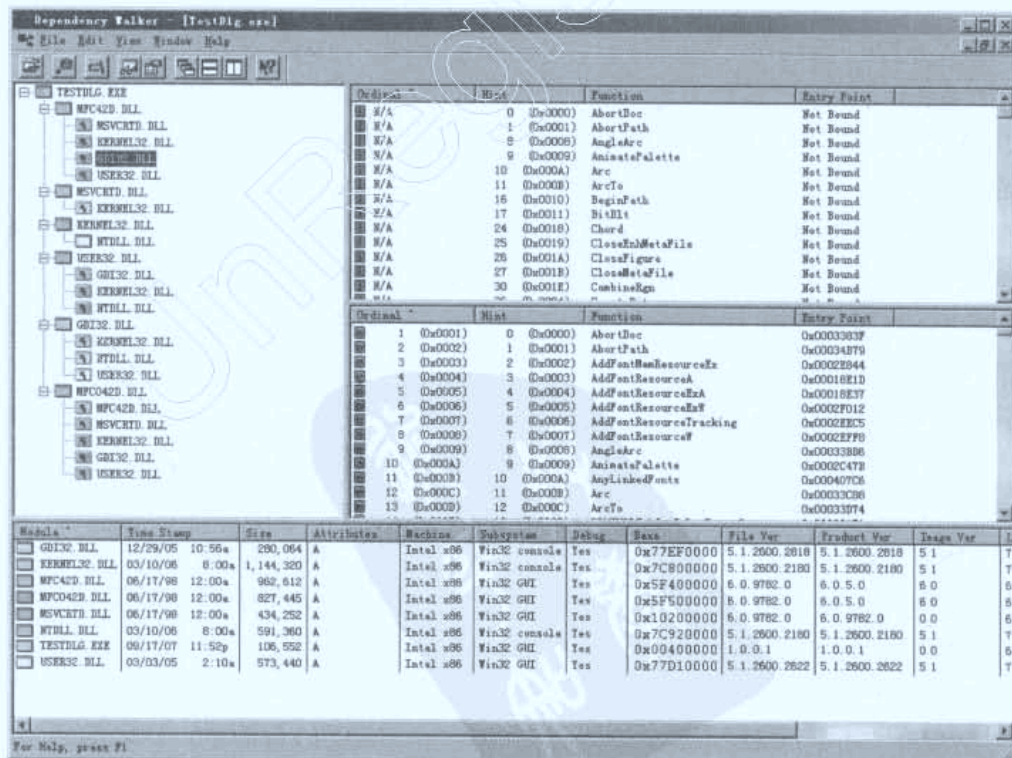


图 2-7 用 Dependency 工具显示应用程序加载的 DLL

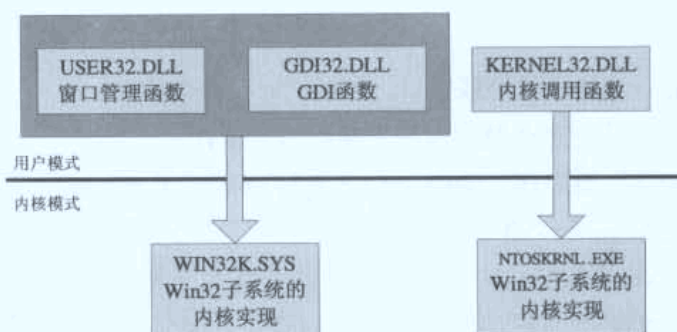


图 2-8 Win32 子系统在内核中的实现

USER32 模块和 GDI 模块的实现纷纷移进了内核模式，这从某种程度上背离了 Windows 开始设计的初衷。因为在内核模式下的模块越多，系统不稳定的几率就越大。然而将这两个 DLL 的实现放到内核模式，会大大提高图形界面的运行效率。Windows 的图形处理模块是在内核模式下运行的，这是 Windows 的窗口表现力非常出色的主要原因。

另外，KERNEL32 模块也是基于内核的执行体组件实现的，其中对线程的调度，是基于执行体下面的内核实现的。

2.2.3 其他环境子系统

除了 Win32 子系统外，Windows 还有 OS/2 子系统、POSIX 子系统、WOW 子系统、VDW 子系统等。这些子系统为各自的应用程序提供相应的 API。其中的 OS/2 子系统，是为了从 OS/2 移植程序到 Windows 平台而设计的。而 POSIX 子系统是为了移植 UNIX 下的程序而设计的。

除了 Win32 子系统外，其他子系统都不是 Windows 的纯正子系统，必须通过调用 Win32 子系统来实现，如图 2-6 所示。这类似虚拟机的概念，例如，在 Windows 上本来没有针对 UNIX 的 fork 等系统调用，POSIX 子系统模拟了 fork 的函数。因此，原有 UNIX 下的程序只需进行少量修改，就可以运行在 Windows 上了。

- **POSIX 子系统：**POSIX 子系统提供了 POSIX 1003.1 标准。遵循此种标准，并使用此种 API 编写的应用程序，可以在 Windows 上的各个版本上运行。POSIX 子系统是为了方便移植某些 UNIX 下的程序。
- **OS/2 子系统：**这个子系统是为了兼容 16 位的 OS/2 操作系统。OS/2 子系统提供了 OS/2 的标准 API，在 Windows 中模拟出 OS/2 操作系统的环境，支持此种标准的 OS/2 应用程序可以在 Windows 上的各个版本上运行。
- **WOW 子系统 (Windows On Windows)：**此版本是为了兼容原先 16 位的 Windows。WOW 子环境系统模拟出 Windows 3.X 的环境，原先 16 位的 Windows 的应用程序，大部分可以很好地运行在这个子系统中。然而对于有些旧的 16 位 Windows 应用程

序，需要直接控制硬件，WOW 子系统当遇到此种情况的时候，将返回一个异常。

- VDM 子系统 (Virtual DOS Machine)：为了保持对 DOS 程序的兼容，Windows 提供了 VDM 子系统，并模拟了对原有 DOS 程序的支持。同样，当原有 DOS 程序需要直接访问硬件的时候，该子系统抛出异常。

然而这些子系统几乎很少被用到。另外，由于种种原因，UNIX 下的程序很少可以很好地移植到 Windows 上。

2.2.4 Native API

大部分 Win32 子系统的 API，都通过 Native API 实现的。Native API 的函数一般都是在 Win32 API 上加上 Nt 两个字母。例如，CreateFile 函数对应着 NtCreateFile 函数。所有 Native API 都是在 Ntdll.dll 中实现的，如图 2-6 所示，三个 Win32 子系统的核心 dll 文件都是依赖于 Ntdll.dll 的。

在 Win32 的底下设置一层 Native API 的调用，是基于版本兼容的考虑。Win32 API 从 Windows NT 到 Windows 2000，再到 Windows XP，基本保持一致，变化的只是 Native API。作为应用程序的开发者，只需了解 Win32 的 API，而不用关心 Native API 的变化。这种机制，可以让 WindowsNT 上的程序直接在更高版本的 Windows 上运行，而不用重新编译。

Native API 没有相应的文档可以查询，但作为 Windows 整体框架的一部分，读者应该了解 Win32 API 调用了底层的 Native API。Native API 是从用户模式进入内核模式的大门，它通过软件中断方式进入到内核模式，并调用内核的系统服务。

微软公司不鼓励程序员直接使用 Native API，而是鼓励程序员使用固定接口的 Win32 API。这是因为 Native API 在各个版本的 Windows 中会有所不同。然而从效率考虑，程序员可以根据自己的需要，在必要时可以直接调用 Native API。

2.2.5 系统服务

Native API 从用户模式穿越进入内核模式，调用系统服务。从用户模式进入到内核模式，是通过软中断的方式进入的。在不同版本的 Windows 下其实现方式略有不同，在 Windows 2000 下是通过“int 2eh”进入内核模式，在 Windows XP 下是通过“sysenter”指令完成的。

软中断会将 Native API 中的参数和系统服务号的参数一同传进内核模式，不同的 Native API 会对应不同的系统服务号。在系统服务组件中，有一个系统服务描述符表 (System Service Descriptor Table)。根据这个系统服务号为索引，从表中可以查出对应系统服务函数的函数地址，如图 2-9 所示。

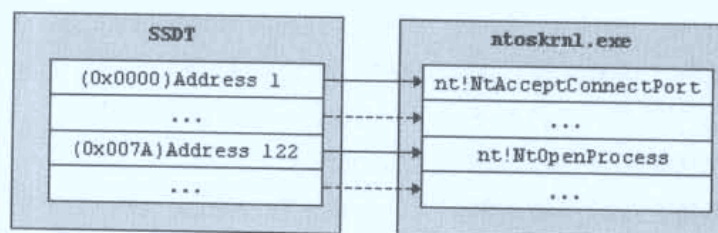


图 2-9 系统服务描述符表

某些黑客软件或计算机病毒，往往更改系统描述符表，将标准的系统服务函数的地址，替换成有恶意代码的地址。系统服务函数一般会与 Native API 有着相同的名字。例如，Native API 中的 `NtCreateFile` 函数，它会调用具有相同名字的系统调用，两个函数名称一样。

在系统调用中，会检验参数的合法性，这也是操作系统最后一道屏障。以后的任何操作，操作系统不会做任何检查，任何错误都会导致系统的崩溃。例如，在用户模式的异常，只会导致进程的退出，不会影响到其他进程或者操作系统内核。而在内核模式下的异常，往往会导致操作系统的崩溃。系统服务会将具体的 I/O 操作通过 I/O 管理器传递到驱动程序中。

2.2.6 执行程序组件

Windows 执行体组件位于 `Ntoskrnl.exe` 的上层，而内核位于其下层。在 Windows 2000 以后，执行体组件和内核从逻辑上分开，这是从代码模块化和可移植性的角度考虑的。

执行程序组件是内核模式下的一组服务函数，它们都位于 `ntoskrnl.exe` 中。执行服务组件又可以细分为若干个组件，下面对这些组件进行说明。

1. 对象管理程序

Windows 操作系统提供的所有服务几乎都是以对象的形式存在的。Windows 的对象类似于面向对象（OOP）语言中对象的概念。Windows 虽然不是用面向对象语言开发，但借鉴了很多面向对象的思想。在 Windows 中，定义了很多的数据结构，这些数据结构就是所谓的“对象”。这些数据结构中的某些变量可以直接被程序员读写，类似于对象概念中的公有成员变量。而数据结构中的有些变量不能被直接读写，类似于对象概念中的私有成员变量。访问这些私有的成员变量，必须通过 Windows 提供的一些例程，而这些例程类似于公有成员函数。

面向对象编程被认为是一种十分有效的软件设计技术，可以提供对代码的全面组织和封装。通过对象隐藏了实现细节，提高了软件的可靠性、健壮性和可移植性。

对象管理器程序就是创建、管理、回收这些对象的组件。在驱动程序开发中，涉及很多的对象，如驱动程序对象（Driver Object）、设备对象（Device Object）等。

2. 进程管理程序

一个进程拥有一个或者多个线程。每个进程维护一个专门的地址空间和安全身份。在 Windows 2000 中，进程不是运行的最小执行单位，而线程才是最小的执行单位。进程相当于线程的容器。

进程管理器负责创建和终止进程，线程的调度是由内核负责的。进程管理程序依赖其他执行程序组件，如对象管理程序、虚拟内存管理程序等。

3. 虚拟内存管理程序

在 Windows 中引入虚拟内存的概念，这有别于物理地址的概念。在 CPU 的内存管理单元 (MMU) 的协助下，通过某种映射将物理内存和虚拟内存关联起来。

每个进程拥有 4GB 的虚拟内存。每个进程观察到的虚拟内存是完全不同的，这是因为每个进程的物理内存到虚拟内存的映射是不同的。这样处理的好处是，物理上保证了每个进程之间不会相互干预，即 A 进程不会破坏 B 进程中虚拟内存的数据。进程间如果需要相互访问虚拟内存，必须通过固定的几种进程间通信机制。

Windows 规定，将 4GB 的虚拟内存分成两个部分。虚拟内存地址 0~0X7FFFFFFF 规定为用户模式的地址，在用户模式下的程序只能访问这段虚拟地址。虚拟内存地址 0X80000000~0xFFFFFFFF 规定为内核模式的地址，这段虚拟地址只能被内核模式的程序访问。

另外，Windows 规定所有进程内核模式下的虚拟内存的映射方式完全一样。这样在每个进程中，顶端 2GB 的内核模式地址的数据完全一致，如图 2-10 所示。

虚拟内存管理程序是负责对虚拟内存管理的模块。对虚拟内存的申请、回收等操作都是由该模块实现的。

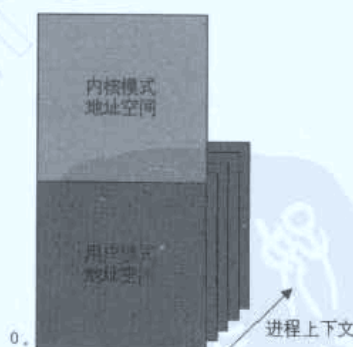


图 2-10 内核模式地址与用户模式地址

4. I/O 管理器

I/O 管理器负责发起 I/O 请求，并且管理这些请求。它由一系列内核模式下的例程所组成，这些例程为用户模式下的进程提供了统一接口。I/O 管理器的目标是使来自用户模

式的 I/O 请求独立于设备。

无论是对端口的读写、对键盘的访问，还是对磁盘文件的操作都统一为 IRP (I/O Request Packages) 的请求形式。其中 IRP 包含了对设备操作的重要数据，例如是读操作还是写操作、读多少字节、写多少字节，是直接读到用户进程中，还是先读到系统缓冲中，再读到用户进程中等。

IRP 被传递到具体设备的驱动程序中，驱动程序负责“完成”这些 IRP，并将完成的状态按原路返回到用户模式下的应用程序中。实际上，I/O 管理器担当着用户模式代码和设备驱动程序之间的接口。

5. 配置管理程序

在 Windows 上，配置管理程序记录所有计算机软件、硬件的配置信息。它使用一个被称为注册表 (Registry) 的数据库保存这些数据。设备驱动程序根据注册表中的信息进行加载。

另外，驱动程序还会从注册表中提取相应的参数，这样可以提高驱动程序的灵活性。例如，设备操作的延时时间，可以作为参数写进注册表。驱动程序加载的时候读取该值，而不是将延时时间在编写程序的时候写成定值。

2.2.7 驱动程序

驱动程序是本书重点介绍的内容。I/O 管理器接收应用程序的请求后，创建相应的 IRP，并传送到驱动程序进行处理，有如下几种处理的方法。

- (1) 根据 IRP 的请求，直接操作具体硬件，然后完成此 IRP，并返回。
- (2) 将此 IRP 的请求，转发到更底层的驱动中去，并等待底层驱动的返回。
- (3) 接受到 IRP 请求后，不是急于完成。而是分配新的 IRP 发到其他驱动程序中，并等待返回。

驱动程序处理 IRP 的过程往往不是单独的操作，而是将以上这几种操作结合在一起。关于驱动程序处理 IRP 的问题，会在后面陆续进行介绍。

2.2.8 内核

内核被认为是 Windows 操作系统的核心。Windows 的内核从执行体组件分割出来。和执行体组件相比，内核是非常小的。内核为执行体组件提供最基本的支持，它负责提供进程和线程的调度，通过自旋锁 (Spin Lock) 提供对多 CPU 同步支持，提供中断处理等。内核提供了以下功能：

- 对内核对象的支持。
- 对线程的调度。

- 对多处理器同步的支持。
- 中断处理函数的支持。
- 对错误陷阱的支持。
- 对其他硬件特殊功能的支持。

Windows 内核执行在最高的特权之上，它被设计成可以并行地运行在多处理器上。内核在调度线程的时候不能被其他线程所打断，即不能允许线程的切换。但是内核可以被更高的中断请求级别（IRQL）所打断。

2.2.9 硬件抽象层

Windows 操作系统易于移植到各个硬件平台的操作系统。事实上，Windows 已经可以运行在 Intel 的 32 位 X86 CPU 上和 64 位的 CPU 上，同时可以运行在 DEC Alpha 的平台，也可以运行在基于 MIPS 的平台上（但由于种种原因，微软公司放弃了对这个平台的支持计划）。

另外，Windows 能更好地支持更多的总线结构，例如，PCI 总线、CPCI 总线、USB 总线、老式的 ISA 总线、EISA 总线等。这些良好的移植性和兼容性得益于硬件抽象层（HAL）的设计。Windows 的设计者用硬件抽象层隔绝了操作系统和硬件的直接连接，而在中间插入了硬件抽象层。硬件抽象层使操作系统和具体硬件进行通信，其中硬件抽象层对各种硬件操作进行了抽象，这也是硬件抽象层名字的由来。

各个平台上的硬件操作不尽相同，不同平台提供不同的硬件抽象层，并对上层提供统一的操作硬件接口。对硬件抽象层操作的不光是操作系统的内核，还有本书介绍的驱动程序。在驱动程序中，应尽量避免使用针对平台的汇编指令，而应该使用与平台无关的 HAL 函数或宏。例如，对端口操作，驱动程序虽然可以嵌入汇编指令，如 IN、OUT 汇编指令，但是应该尽量使用硬件抽象层提供的函数，如 READ_PORT_BUFFER_UCHAR、WRITE_PORT_BUFFER_UCHAR。

2.2.10 Windows 与微内核

微内核的概念与单一内核的概念是相互对立的。单一内核，一般是将系统的主要核心组件全部在内核实现。例如，内存管理器、进程管理器和 I/O 管理等。可以想象，这样设计的内核各组件之间的关联很大，也就是常说的耦合性很大，不利于模块化设计。但优点也显而易见，就是速度快。各组件之间的通信全部在内核模式下完成，没有进程间的切换，也没有从用户模式到内核模式的切换。典型的单一内核的操作系统就是 Linux 操作系统。

和单一内核相反，微内核定义，操作系统的主要组件（例如，内存管理器、进程管理器和 I/O 管理器）运行在独立的进程中，它们有自己私有的地址空间，在这组件之上是微

内核提供的一组服务原语。原语是通过进程间通信进行的，频繁的进程间通讯需要以消耗昂贵的 CPU 时间为代价。而这种代价，换来的是操作系统核心模块的耦合性降低。例如，如果想改变进程模块中进程调度的算法，只需更换进程调度模块，保持原有的原语接口。而在单一内核的操作系统中，各个模块的数据结构紧密结合，牵一发而动全身，即有一个地方稍有改动，则需要进行较大改动。

Windows NT 在设计之初，曾考虑设计成为一个纯粹的微内核操作系统。Windows 的所有程序，全部依赖于 Win32 核心子系统的三大模块，即 Kernel32.dll、User32.dll、Gdi32.dll。其中，User32.dll 负责对窗口消息的分发处理等，GDI32.dll 负责对窗口的图形操作。以前这部分的实现完全是在用户模式下实现的，而在 Windows NT 4.0 后，这两个 dll 的核心代码被放进内核模式下的 Win32k.sys，同时也保留了用户模式下的原有 dll。这使得 User32.dll 和 Gdi32.dll 变得很小，它们只是负责调用内核模式下的 Win32k.sys。这样提高了 Windows 的绘制图形的效率。因此，Windows 能在众多操作系统中，图形能力表现得非常出色。

由此可以看出，Windows 不是纯粹的微内核系统，但它在内核的各个核心组件，达到了相对小的耦合性，同时在效率上克服了微内核效率低的特点。

2.3 从应用程序到驱动程序

打开 Windows 的设备管理器，可以发现这里罗列着计算机里安装的所有设备。这些设备有的是真实的物理设备，例如，网卡设备、显卡设备等。有些设备则是虚拟设备，例如，第 1 章编写的 HelloWDM，它没有对应着 PC 的任何设备，而完全是虚拟出来的“假”设备。虚拟光驱也是这样的虚拟设备。还有些设备介于真实物理设备和虚拟设备之间，比如磁盘的卷设备，磁盘对应磁盘设备，磁盘上的分区又会产生卷设备，这个完全是逻辑上的概念，对卷设备的所有操作，全部会转化成磁盘设备的操作。表 2-2 列出了 Windows 中常用的设备。

表 2-2 Windows 的常用设备

设备分类	功 能
文件设备	对存储文件的操作
目录设备	对目录的操作
逻辑磁盘设备	对逻辑磁盘的操作
物理磁盘设备	对物理磁盘的操作
串口设备	对诸如串口鼠标、串口 Modem 设备的操作
并口设备	对诸如并口打印机的操作

PC 上的设备千差万别，所实现的功能完全不同，如何用统一的接口操作不同的设备，是一个很麻烦的问题。Windows 的设计者们为了简化对不同设备的操作，实现对不同设备

第2章 Windows 操作驱动的基本概念

统一接口，将所有设备以普通文件看待。也就是说在 Windows 中，无论何种设备，都用操作文件的办法去操作设备。

对所有设备的操作统一成和文件操作一样的操作，这一方法非常巧妙。文件操作和设备操作有很多类似的地方。例如，二者都有打开、关闭、读、写、取消等操作。表 2-3 列举了文件操作和设备操作所使用的 Win32 API 函数。

表 2-3 Windows 发布的版本

Win32 API	对文件操作	对设备操作
CreateFile	打开或创建文件	打开或创建设备
CloseHandle	关闭文件	关闭设备
ReadFile	读文件	读设备
WriteFile	写文件	写设备
CanceledIo	取消读写文件操作	取消读写设备操作
DeviceIoControl	(无)	对设备进行特殊操作

下面更深入地介绍一下，Win32API 是如何一步步对设备驱动程序进行读写操作的。图 2-11 是对图 2-6 的简化，读者可以很清晰地看到应用程序到驱动程序是怎样操作设备的。

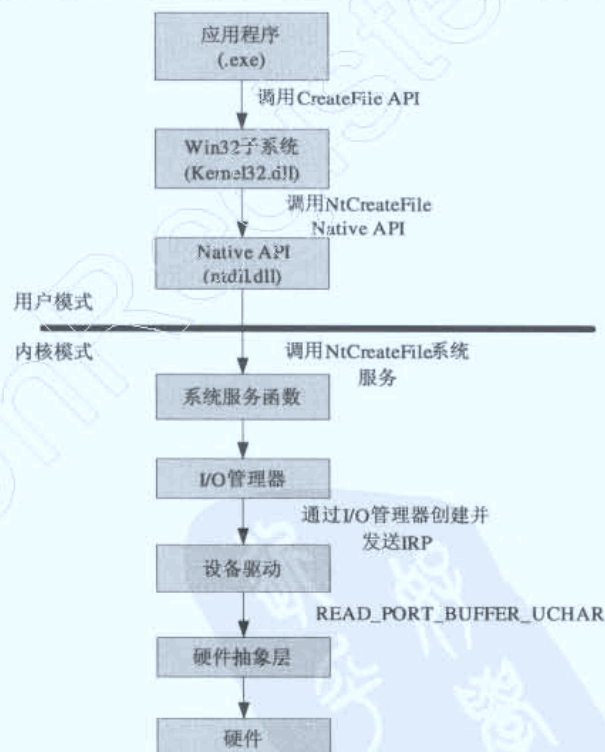


图 2-11 应用程序到驱动程序

这里以 CreateFile API 为例，其他操作设备的 API 类似。首先应用程序调用 CreateFile API，这个 API 是由 Win32 子系统的三大模块中的 Kernel32.dll 实现的。CreateFile 函数会

Windows 驱动开发技术详解

调用 `Ntdll.dll` 中的 `NtCreateFile` 函数，其中 `NtCreateFile` 是未文档化的函数，程序员最好不要直接使用这个未文档化的函数。

`NtCreateFile` 的作用是穿越用户模式的边界，进入到内核模式，这个步骤是通过软中断实现的。进入到内核模式后，会调用系统的服务函数，这里会调用同名的系统服务 `NtCreateFile`。（这里很容易搞混，虽然都叫 `NtCreateFile`，但一个是位于用户模式的 Native API，另一个是位于内核模式的系统服务调用）

`NtCreateFile` 系统函数调用通过 I/O 管理器，创建 IRP 并传输到设备的驱动程序中。IRP（I/O Request Package）即输入输出请求包，是驱动程序开发中重要的数据结构。驱动程序的运行，完全是靠 IRP 驱动的。下面会有对 IRP 的专门介绍，这里读者可以将 IRP 理解为一个消息。这个消息告诉驱动程序，是需要读操作还是写操作。

驱动程序根据 IRP，进行相应的操作。这些操作一般是对设备的直接操作，例如对端口的读操作。对端口的读操作根据不同的硬件平台，实现方法会有所不同，Windows 根据不同的硬件平台，会有不同的硬件抽象层（HAL）。硬件抽象层提供一组宏，如 `READ_PORT_BUFFER_UCHAR`。例如，对 32 位 X86 系列 CPU 中的 Windows，`READ_PORT_BUFFER_UCHAR` 被解释为汇编代码 `IN` 操作。

回想这个复杂的过程，经过了多个层次的交互，只是为了执行一个读写端口的操作。对于有过 DOS 编程经验的程序员来说，在 DOS 中操作硬件完全可以不使用驱动，直接使用 `IN` 汇编代码就可以实现。事实的确如此，但 Windows 这样做完全是为了安全的考虑。所有直接操作硬件的指令，如读写物理内存、读写端口都认为是危险的操作，必须经过驱动才能完成。

试想一下，如果应用程序能任意执行 `IN` 和 `OUT` 汇编指令，那么就可轻易地对磁盘进行控制，这会给操作系统带来安全隐患。又例如，如果能直接读写物理内存，黑客会很容易地对当前进程以外的其他进程进行内存读写，那么盗取账号密码将会变得非常简单。

因此，在应用程序中无法执行 `IN` 汇编指令，而必须通过驱动程序来执行。在一层层地调用中，每层又都会严格地检查，保证参数的合法性。

2.4 小结

本章对 Windows 操作系统的内部构造做了概要性地介绍。编写的驱动需要在内核模式下加载，并且需要和操作系统的各个核心组件相互配合，因此了解操作系统的内部结构是非常必要的。

应用程序无法直接与硬件设备通信，必须借助于驱动程序。读者可以将驱动程序理解成对操作系统的“补丁”。应用程序就是借助这个“补丁”来操作硬件设备的。下一章将为读者详细地介绍 Windows 驱动程序的编译、安装。

第 3 章 Windows 驱动编译 环境配置、安装及调试

本章将带领读者一步步对驱动程序进行编译、安装和简单的调试工作。这些步骤虽然简单，但往往困惑着初次接触驱动程序的开发者。另外，本章还介绍了 C 语言开发的注意细节。

3.1 用 C 语言还是用 C++语言

在编写 Windows 驱动程序之前，需要先确定使用的开发语言。Windows 驱动程序和普通 Win32 的应用程序一样，同样是 PE (Portable Execute) 格式的文件。因此从广义的角度讲，只要某种语言的编译器能编译出 PE 格式的二进制文件，并能正确指明驱动程序的入口地址，那么该种语言就可以开发 Windows 驱动程序。所以读者可以选择 C、C++、汇编，甚至可以用 Delphi 开发。但由于微软提供给用户的 DDK 开发环境所提供的包含文件和链接用的库只支持 C 和 C++语言，因此本书讲解的开发全是基于 C 语言或 C++语言。笔者曾经看到过某些研究内核的网站上，提供了汇编开发驱动程序的开发环境。如果读者是 Windows 汇编的爱好者，可以将本书中的 C 代码改写成汇编代码，同样能编译出 Windows 驱动程序。

由于 C++语言比 C 语言多了很多灵活的特性，例如，函数中允许对变量的声明，声明位置不一定非要在函数的首部，而允许在函数的任意部分。又例如，C++可以将一组函数封装在一个类中，使代码更加模块化。因此本书中的代码大部分采用 C++编写。然而需要注意的是，本书介绍的驱动程序仅仅使用一些简单的 C++特性。C++的许多高级特性是无法也不适宜在 Windows 驱动开发中用到的，同样用 C 语言编写驱动程序，也有若干限制，本节将一一进行介绍。

3.1.1 调用约定

调用约定指的是函数在被调用的时候，会按照不同的规则，翻译成不同的汇编代码。为了解释这个概念，首先了解一下调用堆栈的概念。当一个函数被调用时，首先会将返回地址压入堆栈，紧接着会将函数的参数依次压入堆栈。当函数退出时会以相反的顺序依次退出堆栈。因此，函数在被调用前和调用后的堆栈保持平衡，如图 3-1 所示。

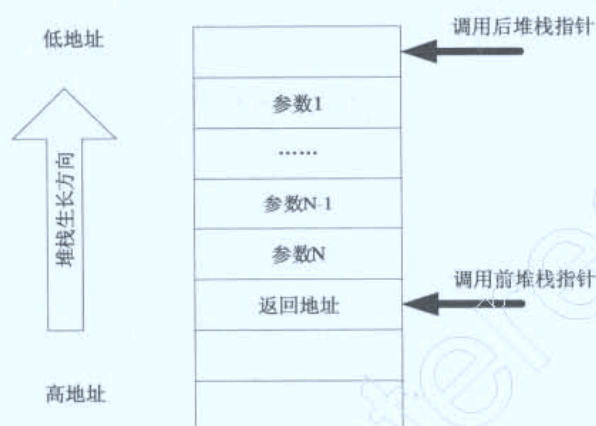


图 3-1 函数调用堆栈

不同的调用约定，会指明不同的参数入栈顺序，还会指明不同的清理堆栈的方法。用 C 语言或者 C++ 语言编译器编译程序的时候，会按照四种不同的调用约定去编译函数。其分别是 C 语言调用约定，函数由 `__cdecl` 修饰；标准调用约定，函数由 `__stdcall` 修饰；快速调用约定，函数由 `__fastcall` 修饰；C++ 类成员函数的调用约定，函数由 `thiscall` 修饰。不同的调用约定编译后，会产生不同的汇编代码。下面对 C 语言调用约定和标准调用约定进行简单介绍。

1. C 语言调用约定

C 语言调用约定，要求在声明函数时用 `__cdecl` 对函数进行修饰，例如：

```
void __cdecl Foo(int a,int b);
```

C 语言调用会在目标（Object）文件中产生一个符号来代表这个函数，此符号的形式为下划线+函数名，且函数体以 `ret` 形式返回。例如：

```
Foo(0x12345678,0x11223344);
```

展成汇编代码如下：

```
push    11223344h
push    12345678h
call    _Foo
add     esp,8
```

第3章 Windows 驱动编译环境配置、安装及调试

即从右至左，将参数推进堆栈，执行结束后，以 `ret` 返回。此时的堆栈和调用前的堆栈不一致，需要“调用者”恢复堆栈，用 `add` 指令将堆栈恢复平衡。

注意：C 语言或者 C++ 语言在编译成汇编代码时，符号名会对变量名和函数名进行替换，替换成一个内存上的地址。由于地址不容易分辨，所以用常量代替。例如，函数 `Foo` 编译成汇编代码就会变成 `_Foo` 或 `_Foo@8`。

2. 标准调用约定

标准调用约定，要求在声明函数时用 `__stdcall` 对函数进行修饰，例如：

```
void __cdecl Foo(int a, int b);
```

C 语言调用会在目标文件中产生一个符号来代表这个函数，此符号形式为下划线+函数名+X。其中，X 代表清理堆栈时需要的数字，函数以 `ret X` 形式返回。例如：

```
Foo(0x12345678, 0x11223344);
```

展成汇编代码如下：

```
push    11223344h
push    12345678h
call    _Foo@8
```

即从右至左，将参数推进堆栈，当函数调用完，函数以 `ret 8` 返回函数。Foo 函数负责恢复堆栈，而“调用者”不负责恢复堆栈，这个是 C 语言调用和标准调用最重要的区别之一。

一般程序中，很少见到用关键字指定函数的调用约定，编译器会选择默认的调用约定进行编译，在 VC 编译器中，默认使用 C 语言的调用约定。而在 Windows 驱动程序的编写中，需要使用标准调用约定，尤其是入口函数。系统会寻找 `_DriverEntry@8` 作为驱动程序的入口点。如果用 C 语言调用约定，会将 `DriverEntry` 编译成 `_DriverEntry`，而不是 `_DriverEntry@8`，那么会导致链接错误。因此在编译驱动时，需要改变默认的编译调用约定，如图 3-2 所示。

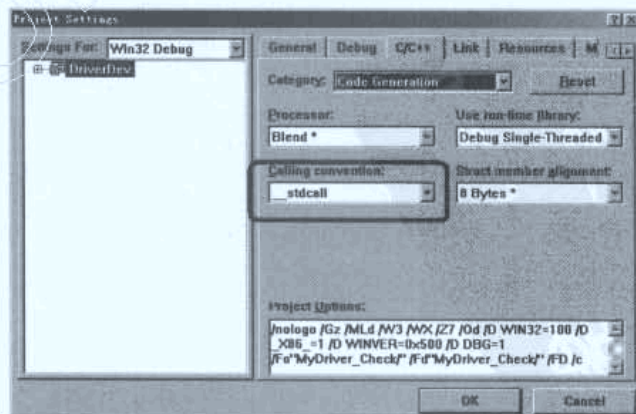


图 3-2 设置调用约定

Windows 驱动开发技术详解

如果读者用 DDK 编译环境编译驱动程序，默认采用的是标准调用约定，所以可以忽略此项。

3.1.2 函数的导出名

同样一个函数，用在 C 语言编译器和 C++ 语言编译器编译出的符号名是不同的，这点尤其需要注意。因为在链接的时候，链接器不知道源程序的函数名，而只会去目标 (Object) 文件中寻找相应的函数符号表。VC 或者 DDK 提供的编译器 cl.exe，既可以编译 C 语言，又可以编译 C++ 语言。默认情况下，编译器会根据源文件的扩展名，来判断使用哪种方式编译。当文件扩展名是 .cpp 时候，编译器会用 C++ 方式编译，当文件扩展名是 .c 时候，编译器会用 C 编译器方式编译。例如，同样使用标准约定编译的函数：

```
void __stdcall Foo(int a,int b);
```

在 C++ 编译器中会编译成符号 ?Foo@@YGXHH@Z，而在 C 编译器中会编译成符号 _Foo@8。C++ 复杂的函数符号名是为了支持函数的重载功能的，这里不做深究，有兴趣的读者可以参考编译器的相关文档。

Windows 驱动程序的入口函数规定为 _DriverEntry@8，因此用 C++ 编译的时候，会导致符号链接错误。解决办法是采用 extern "C" 修饰符，例如：

```
#pragma INITCODE
extern "C" NTSTATUS DriverEntry (
    IN PDRIVER_OBJECT pDriverObject,
    IN PUNICODE_STRING pRegistryPath );
```

另外，在 C++ 程序中，需要包含 ntddk.h 或者 wdm.h 的时候，会出现如下错误信息。

```
-----Configuration: DriverDev - Win32 Debug-----
Compiling...
HelloWDM.cpp
Linking...
HelloWDM.obj : error LNK2001: unresolved external symbol "unsigned long __cdecl
DbgPrint(char *,...)" (?DbgPrint@@YAKPADZZ)
HelloWDM.obj : error LNK2001: unresolved external symbol "__declspec(dllimport) long
__stdcall IoDeleteSymbolicLink(struct _UNICODE_STRING *)"
(__imp_?IoDeleteSymbolicLink@@YGJPAU_UNICODE_STRING@@@Z)
HelloWDM.obj : error LNK2001: unresolved external symbol "__declspec(dllimport) long
__stdcall IoCreateSymbolicLink(struct _UNICODE_STRING *,struct _UNICODE_STRING *)"
(__imp_?IoCreateSymbolicLink@@YGJPAU_UNICODE_STRING@@@Z)
HelloWDM.obj : error LNK2001: unresolved external symbol "__declspec(dllimport)
struct _DEVICE_OBJECT * __stdcall IoAttachDeviceToDeviceStack(struct _DEVICE_OBJECT
*,struct _DEVICE_OBJECT *)" (__imp_?IoAttachDeviceToDeviceStack@@YGJPAU_DEVICE_OBJECT@
@PAUL@@@Z)
e o o o o
```

出现错误的原因是，按照 C++ 的编译方式，源文件包含了 ntddk.h 或者 wdm.h。当驱动程序中用到内核函数时（如 IoCreateSymbolicLink）连接器会寻找符号 ?IoCreateSymbolicLink@@YGJPAU_UNICODE_STRING@@@Z，而不会寻找符号 _IoCreateSymbolicLink@8。

因此，报告的是一个链接错误，而不是编译错误。修改的办法很简单，包含头文件的时候，用 `extern "C"` 去修饰，代码如下：

```
#001  #ifdef __cplusplus
#002  extern "C"
#003  {
#004  #endif
#005  #include <NTDDK.h>
#006  #ifdef __cplusplus
#007  }
#008  #endif
```

`#ifdef __cplusplus` 判断是否使用 C++ 的编译方式编译。将包含语句用大括号括住，并用 `extern "C"` 修饰，这样声明的时候，就按照 C 的方式编译了。

注意：应注意区分编译（Compile）和链接（Link）这两个概念。简单地讲，编译是将源文件变成目标文件的过程，链接是将目标文件变成最终二进制映像的过程。

3.1.3 运行时函数的调用

Windows 驱动程序虽然和普通 Win32 应用程序一样，都是用 C 语言或者 C++ 编写，但是比起普通应用程序，增加了很多严格的限制。很多 C 语言和 C++ 语言的使用技巧，要慎重使用。例如，在 Windows 驱动程序中，不能使用编译器运行时函数（Run Time Function），甚至 C 语言中的 `malloc` 函数和 C++ 语言中默认的 `new` 操作符都不能使用（如果使用 `new`，必须重载 `new` 操作符），这些涉及很多 Windows 操作系统体层的知识和编译器的知识。

编译器厂商一般在发布编译器的同时，会同时将其运行时函数一起发布给用户。运行时函数是一个程序运行时所必不可少的函数。它由编译器提供，针对不同的操作系统实现也有所不同，但接口基本上是标准的。例如，`malloc` 函数就是典型的运行时函数，所有编译器厂商都必须提供这个函数，不同厂商在不同操作系统上，实现方法是不同的。

对于读过 VC 运行时函数源代码的读者，大概已经明白为什么在 Windows 驱动程序里不能使用编译器提供的运行时函数了。原因很简单，大部分运行时函数是通过 Win32 API 实现的，而 API 是针对 Ring3 层（用户模式）的程序的，Windows 驱动运行在 Ring0 层（内核模式）。内核模式下的程序是无法调用用户模式提供的 API 函数的。C 语言的 `malloc` 函数和 C++ 语言提供的 `new` 操作符，无一例外最终调用了 Windows 的 API 函数。有兴趣的读者可以在 VC 环境中单步跟踪一下 `new` 操作符的代码，会发现它最后会调用 `HeapAlloc`。

`HeapAlloc` 是 Windows 提供的 API 函数，操作系统为用户提供了“堆”的使用，用户不必关心内部复杂的算法。`HeapAlloc` 是若干个操作堆的 API 中的一个，它最终会调用操作虚拟内存的 API 函数，如 `VirtualAlloc`。不管是应用程序，还是运行时函数，在用户态最终调用的是操作虚拟内存管理 API。而编写驱动程序时，驱动位于内核模式下（Ring0 层）。Windows 操作系统规定，在内核模式的程序是无法调用用户模式的程序，而用户态可以调用内核态的程序（这需要严格的参数审查）。应用程序管理内存分配，如图 3-3 所示。

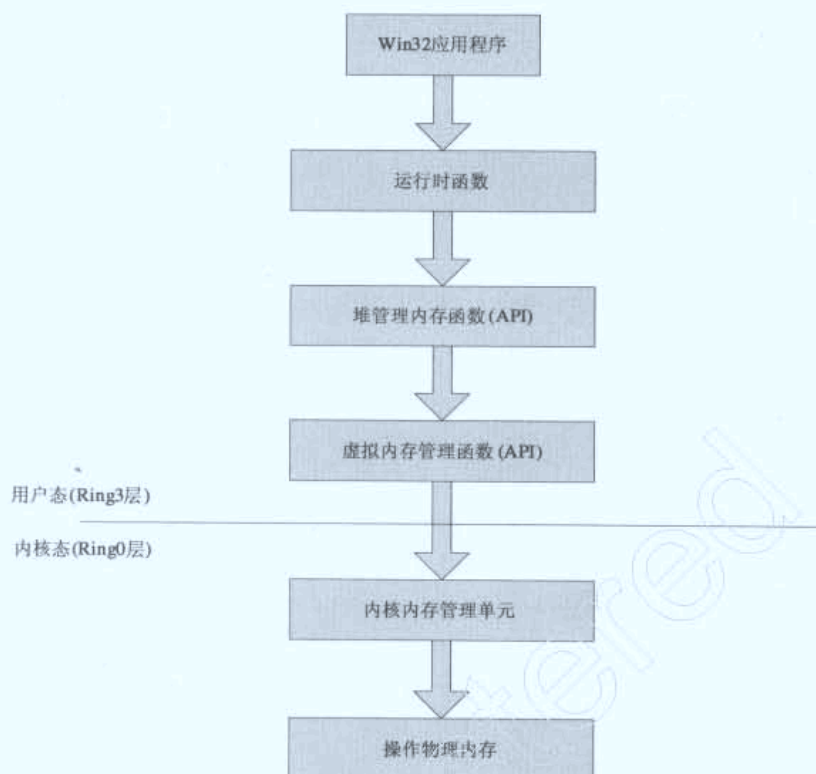


图 3-3 运行时函数调用情况

Windows 为用户提供了内核态的运行时函数，它可以替代应用程序的运行时函数。在内核态的运行时函数一般形如 `RtlXXXX`，这些函数会在第 4 章进行详细介绍。

有一些运行时函数，如 `strcpy` 等，他们的实现不依赖于 API，读者完全可以将其用在 Windows 驱动的编写中，但是如果读者并不清楚这个函数的实现方法，最好用 DDK 提供的运行时函数。

有些读者用过 DriverStudio 软件，DriverStudio 就提供了 `new` 操作符。DriverStudio 对 `new` 操作符进行了重载。默认情况下，编译 `new` 操作符的时候，编译器会根据自己的运行时函数编译出结果。如果必须在驱动中用到 `new` 操作符，请重载 `new` 操作符，当然一定还要重载 `delete` 操作符。

注意：笔者建议，在驱动程序中，尽量完全用 DDK 提供的运行时函数。

3.2 用 DDK 编译环境编译驱动程序

DDK 提供给开发者一系列编译和链接的工具，这些工具包括编译程序 `cl.exe` 和链接程序 `link.exe`。这些编译程序和链接程序与 VC 中提供的工具完全一样，只是 VC 6 中的编译器版本可能略低一些。所以理论上，驱动程序既可以用 DDK 编译环境编译，也可以用 VC

来编译。

在 DDK 编译环境下，用户需要编写脚本文件，描述工程里用到的源文件名、包含目录路径、库目录路径、编译优化参数等信息。然后用户在命令行方式运行 `build.exe` 命令，编译环境会自动解析脚本文件，编译链接驱动程序。

3.2.1 编译版本

DDK 编译环境为用户提供两种编译版本，Checked 版本和 Free 版本。Checked 版本和 Free 版本的关系类似 VC 中的 Debug 版本和 Release 版本。

Free 版本是最终的发行版本，要进行必要的优化并删除所有的调试符号，编译环境会将编译器的全部优化参数打开。因此，此版本编译出来的驱动体积是最小的，运行速度是最快的，但无法进行源代码级的调试。

Checked 版本和 Free 版本相反，是一个未优化的调试版本，里面包含了大量的调试符号，来对应源代码中具体的位置。关闭所有的优化参数，是为了便于调试，因为很多优化会打乱源代码和编译结果的关联。Checked 版本编译出来的驱动体积是最大的，运行速度是稍慢的，其最大优点是可以进行源代码级别的调试，可以使用 Softice 或者 WinDbg 进行调试。观察 Win XP Checked Build Environment 的快捷方式，内容为：

```
C:\WINDOWS\system32\cmd.exe /k d:\WINDDK\2600~1.110\bin\setenv.bat d:\WINDDK\2600~1.110 chk
```

很明显，快捷方式运行 `setenv.bat`。此批处理会接收两个参数，其中一个是 DDK 的根目录，另外一个是指定 checked 版本还是 free 版本。如果 XP DDK 完全安装的，会有八个不同的编译版本，如图 3-3 所示。

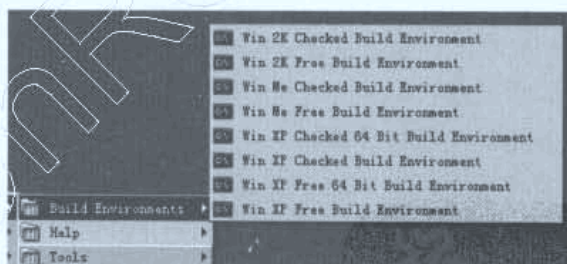


图 3-4 Checked 版本和 Free 版本

不同的版本会调用不同的批处理，且参数不同，目的是产生不同的环境变量。Build 会根据环境变量的不同，产生不同版本的二进制映像。

3.2.2 nmake 工具

在早期程序员编程时没有 IDE 开发环境，必须在一个 `makefile` 的文件中指定需要编译的文件，并通过 `nmake` 工具解析 `makefile` 文件。此文件按照编译的依赖顺序做出判断，需

要先编译哪些文件后编译哪些文件。

例如, exe 文件是依赖于 obj 文件的, 也就是 obj 文件的最后修改日期晚于 exe 的最后修改日期, 那么就认为 exe 是过时的, 需要重新编译等。一般的程序依赖关系如图 3-5 所示, 箭头的方向表示“依赖于”。

在 makefile 文件中, 依赖于的关系用下列语法声明。

```
A:B,C
    action
```

此语法的意思是 A 文件依赖于 B 文件和 C 文件, 如果 A 的最后修改日期晚于 B 和 C 中任意一个文件的最后修改日期, 则执行第二行的 action 动作。action 前面不是空格, 而是 tab 位 (这个很重要, 如果用空格代替了 tab 位, nmake 会分析错误)。action 可以是任何动作, 比如编译、链接、甚至是拷贝文件。

使用 Makefile 文件是十分古老的技术, 但效率会比集成开发环境高。所以直到现在, VC 中依然可以看到将工程文件转换成 makefile 的使用方法, 如图 3-6 所示。

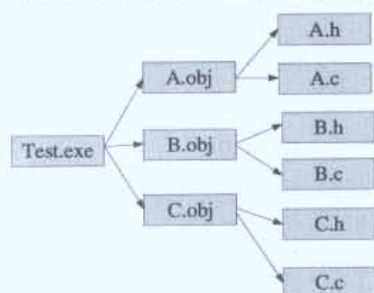


图 3-5 makefile 中的依赖关系

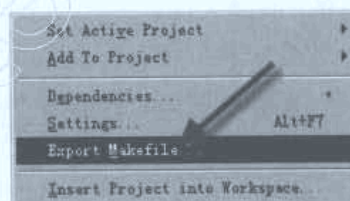


图 3-6 VC 中输出 makefile

编好 makefile 文件, 就可在命令行方式下运行 namke 了。在命令行方式下, 首先运行 VCVARS32.BAT, 这个批处理文件负责设置一些系统环境变量, 这些环境变量一般是关于 include 目录的位置、lib 的位置等, 此批处理的位置在 C:\Program Files\Microsoft Visual Studio\VC98\Bin。运行 nmake 的整体步骤如下:

```
VCVARS32
nmake Test.mak (Test.mak 是此工程的 makefile 的名称)
```

运行完 namke 后, 会编译生成所需要的最终二进制映像。

3.2.3 build 工具

DDK 提供的编译环境, 主要是调用 build 工具编译和链接代码。Build 的主要工作是调用 nmake 工具, 将默认的参数传进 nmake 工具中。同时 build 会根据不同的编译版本, 设置不同的系统环境变量。这样在 nmake 调用的时候, 会根据不同版本的环境变量, 编译生成出不同的结果。

build 工具会根据 checked 版本和 free 版本生成代码。同时, 也会根据指定的操作系统,

如 Windows XP、Windows 2003 或其他版本的 windows 生成代码（针对不同的操作编译出来的驱动程序略有不同）。而且，也会根据不同的平台，如 386 系列平台（i386）、Itanium 平台，编译出不同的代码。

build 可以生成不同版本的编译结果，而不需要编写多个版本的 makefile 文件，只需要一个 makefile 就够了，这就是 build 工具的优点，其调用关系如图 3-7 所示。

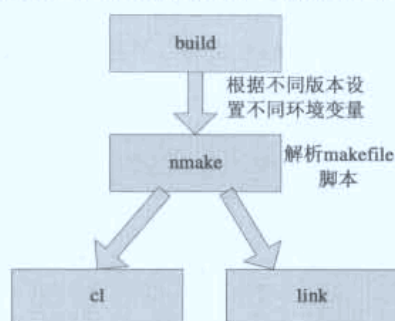


图 3-7 build 的示意图

因此，DDK 提供的 build 工具比 nmake 工具编译更加便利。build 工具将所有的要生成的编译版本隔绝，使之互不影响。例如，通过 build 工具，可以指定在编译 32 位驱动程序时，用一系列的包含文件、库文件和一些编译开关。而编译 64 位驱动时，指定使用另外一系列的包含文件、库文件和编译开关。从而不用修改源文件、makefile，就可以达到同时生成多个平台上的驱动。

初次接触 build 工具的程序员，恐怕会觉得很不适应。因为控制编译和链接的选项全部要用脚本来描述，而不像在 VC 集成开发环境那样，在相应的菜单里就可以轻易设置。因此在下一节中，将介绍用 VC 集成开发环境编译驱动的方法。其实不管是 build 工具、nmake 工具，还是 VC 集成开发环境，都是最终调用的 cl.exe（编译器）和 link.exe（链接器），所以其原理都是大体相同的。

读者可以根据自己的喜好选择编译方法。了解各种编译方法，会对编译器和链接器的各项参数有更深入的理解。

3.2.4 makefile 文件

makefile 文件指定文件之间的相互依赖关系，确定项目中哪些文件需要重新编译。在 DDK 程序进行编译时，build 工具会调用 nmake 工具去解析 makefile 文件进行编译，关于 makefile 更加详细的信息参见 MSDN 文档。

makefile 需要列出一系列依赖关系，如果从头写 makefile 会是很烦人的事情。大多数情况下，用户只需写如下一个 makefile 文件。

```
!INCLUDE $(NTMAKEENV)\makefile.def
```

这个 makefile 除了前面的注释外，主要是包含了 DDK 目录中的 makefile.def。

DDK 文档不建议用户修改该 makefile 文件，请遵从 DDK 文档的这个建议。

3.2.5 dirs 文件

build 工具可以递归地指定需要编译的文件和目录。dirs 文件中描述需要编译的子目录。如果当前目录中存在一个 dirs 文件，build 会分析此文件，依次进入这些子目录编译。这些子目录可以含有其他 dirs 文件或者 sources 文件。下面是 DDK 根目录下 src 中的 dirs 文件，如果执行 build 命令后，这些子目录将依次进行编译。

```
DIRS= \  
    general \  
    ime \  
    input \  
    kernel \  
    mmedia \  
    network \  
    print \  
    setup \  
    smartcrd \  
    storage \  
    vdd \  
    video \  
    wdm
```

注意：其中“\”符号是换行符号，当一行的结尾出现“\”符号时候，下一行的文本会被认为与上一行为同一行。这样的目的一般是为了将同一行的文本分散到多行，可以一目了然。

3.2.6 sources 文件

build 工具会寻找当前目录下的 source 文件或者 dirs 文件中指定子目录下的 source 文件。此文件记录了需要编译的源文件文件名、包含目录路径、库目录路径等信息。

一般情况下，这些所指定的信息全部是修改 build 工具提供的变量，build 会根据这些变量通知 nmake 工具进行编译。下面列出几个常用的 build 变量。

- TARGETNAME: 描述目标驱动驱动的名称。
- TARGETTYPE: 描述目标代码生成的类别。TARGETTYPE=DRIVER 意味着是生成驱动，如果 TARGETTYPE=PROGRAM，则编译成 Win32 程序。
- DDKROOT: 设置 DDK 的根目录。
- C_DEFINES: 指示 c 预编译定义参数，其作用相当于在 c 文件中用#define 声明的定义。
- TARGETPATH: 指示目标代码生成的路径。
- INCLUDES: 设定包含目录的路径。
- TARGETLIBS: 设置目标代码所需要的库。

- **MSC_WARNING_LEVEL**: 指明编译警告级别。一般情况下设为/W3, 即第三级别的警告。
- **SOURCES**: 指定此工程所有的源文件, 注意只指定 c 文件或者 c++ 文件, 而不需要指定 h 文件。

下面是一个标准的 sources 文件, 读者可以根据此 source 文件, 修改成自己所需要的 source 文件。

```
#001 TARGETNAME=bulkusb
#002 TARGETTYPE=DRIVER
#003 DDKROOT=$( _NTDRIVE )$( _NTROOT )
#004
#005 C_DEFINES= $(C_DEFINES) -DWMI_SUPPORT -DUSB2
#006
#007 TARGETPATH=obj
#008
#009 INCLUDES=$(DDKROOT)\private\ntos\inc; \
#010     ..\..\inc
#011
#012 NTTARGETFILE0=mofcomp
#013
#014 USE_MAPSYM=1
#015
#016 TARGETLIBS=$(DDK_LIB_PATH)\hidclass.lib \
#017     $(DDK_LIB_PATH)\usbd.lib \
#018     $(DDK_LIB_PATH)\ntoskrnl.lib
#019
#020 MSC_WARNING_LEVEL=/W3 /WX
#021
#022 SOURCES=bulkusb.c \
#023     bulkpnp.c \
#024     bulkpwr.c \
#025     bulkdev.c \
#026     bulkwmi.c \
#027     bulkwr.c \
#028     bulkusb.rc
```

3.2.7 makefile.inc 文件

makefile.inc 是可选的, 它也是一个 makefile 文件。如果当前目录存在这个文件, build 会分析此文件, 并按照此文件进行 c 语言编译之外的动作。

```
#001 mofcomp: bulkusb.bmf
#002
#003 bulkusb.bmf: bulkusb.mof
#004     mofcomp -B:bulkusb.bmf bulkusb.mof
#005     wnimofck bulkusb.bmf
```

这是一个标准的 makefile 文件, mofcomp 依赖于 bulkusb.bmf 文件, 如果 mofcomp 的比 bulkusb.bmf 的日期“过时”, 则执行一个行为, 但此处的行为没有指定, 即空行为。后面指定 bulkusb.bmf 依赖于 bulkusb.mof, 如果 bulkusb.bmf 比 bulkusb.mof “过时”, 则执行 mofcomp -B:bulkusb.bmf bulkusb.mof。

3.2.8 build 工具的环境变量

build 工具的环境变量指的是在上述这些文件中出现的环境变量。build 会初始这些变量，用户可以根据需要更改这些变量，这里介绍一些常用的环境变量。

- **BASEDIR**: 此变量指定驱动目录的基准，默认为 DDK 的根目录。
- **BUILD_DEFAULT**: build 命令会调用 nmake 命令，此环境变量设置默认调用的参数。例如：

```
c:\> build -eswM -nmake -i
```

等价于

```
c:\> set BUILD_DEFAULT=-eswM -nmake -i
c:\> build
```

- **BUILD_DEFAULT_TARGETS**: 此环境变量设置默认的编译目标平台。假设在 Itanium 平台下，可以编译出 x86 的代码。一般来说这个变量设置就是用户使用的那个平台。例如：

```
c:\> build -386
```

等价于：

```
c:\> set BUILD_DEFAULT_TARGETS=-386
c:\> build
```

- **BUILD_MAKE_PROGRAM**: 此环境变量设置 makefile 的文件名，默认情况下，此变量就是当前目录下的 makefile，一般情况下请不要修改此变量。
- **C_DEFINES**: 此环境变量可以设置预编译的定义，这如同在源文件中指定 #define 预定义。例如，想通知编译器定义 DEBUG_BUILD，可以像如下代码一样修改这个变量：

```
C_DEFINES = /DDEBUG_BUILD
```

这里有个小技巧，在 sources 文件中，为了增加新的预定义，而不会将已定义预定义覆盖掉，可以做如下定义：

```
C_DEFINES = $(C_DEFINES) /DUNICODE
```

- **CRT_INC_PATH**: 此变量指定 C 运行时的包含文件路径。默认值是 %ddkroot%\inc\crt。
- **CRT_LIB_PATH**: 此变量指定 C 运行时的库目录文件路径。默认值是 %ddkroot%\lib*。
- **DDK_INC_PATH**: 此变量指定 DDK 的包含文件路径。默认值是 %ddkroot%\inc\ddk\wxp。

- DDK_LIB_DEST: 此变量指定 DDK 编译库的路径。默认值是%ddkroot%\lib。
- DDK_LIB_PATH: 此变量指定 DDK 的导入库目录路径。默认值是%ddkroot%\lib*。
- WDM_INC_PATH: 此变量指定 WDM 头文件的目录路径。默认值是%ddkroot%\inc\ddk\wdm\wxp。

3.2.9 build 工具的命令行参数

build 工具功能十分强大, 用户可以根据需要进行不同参数的设置。

- -0: 编译器只扫描一下代码, 不进行编译和链接。
- -2: 扫描两遍代码。
- -3: 同参数-Z。
- -b: 显示冗余的错误信息。
- -c: 删除掉所有的生成库。
- -C: 删除掉多有的生成库。
- -clean: 等价于调用"-nmake clean"。
- -D: 在编译器前检测依赖关系。
- -e: 生成错误、警告和错误信息。
- -E: 保持所有的 log 文件。
- -f: 强制扫描所有的源文件和包含文件。
- -i: 忽略编译器给出的警告信息。
- -j: filename。其中, filename 为 log 信息文件名。此参数将生成下列文件: filename.log、filename.wrn 和 filename.err。
- -j: path pathname。其中, pathname 作为 log 文件输出目录路径, 而不是在当前目录产生 log 文件。
- -k: 保留过期的目标文件。
- -l: 只链接, 不编译。
- -L: 只编译, 不链接。
- -n: make arg。将参数传进 NMAKE 命令。
- -o: 显示过期文件。
- -O: 相对于在当前目录产生 obj\objects.mac 文件。
- -p: 在编译和链接的阶段暂停一下。
- -P: 输出每个文件编译的时间。
- -q: 仅查询, 不去运行 NMAKE。
- -r: dirpath 重新清除所指定目录。
- -s: 在底部显示目前编译的状态。

- -t: 显示依赖关系树。
- -T: 显示完整的依赖目录树。
- -\$: 显示完整的依赖目录树框架。
- -v: 允许包含目录版本检测。
- -w: 在屏幕上显示警告信息。
- -y: 显示扫描文件过程。
- -z: 不进行依赖检测，只是一遍编译和链接。
- -Z: 不进行依赖检测，三遍编译和链接。
- -386: 编译 386 平台的驱动。
- -x86: 同参数-386。
- -ia64: 编译 Itanium 平台驱动。
- -?: 显示帮助信息。

3.3 用 VC 编译驱动程序

VC 提供了强大的集成开发环境，比起简陋的命令行方式，用户可能更喜欢用 VC 开发驱动程序。VC 编译驱动程序同样是调用编译器 `cl.exe` 和连接程序 `link.exe`。

通过 VC IDE 可以很方便地浏览各个源文件和头文件，并且可以交叉索引。同时，在编译出错的情况下，单击出错信息，直接可以跳到出错代码的位置。另外，用户不用记烦琐的编译参数，所有的参数都是以图形界面的方式指定。这些大大提高了开发效率。

3.3.1 建立驱动程序工程

VC 创建工程一般都是通过 Wizard 向导创建，但是 VC 并没有提供创建一个驱动程序的 Wizard 向导。这里需要创建一个空的 VC 工程，并进行一系列的修改。本书曾在第 1 章给出创建空 VC 工程的实例，不熟悉的读者请复习第 1 章。

3.3.2 修改编译选项

生成代码最重要就是编译和链接两个步骤。先了解一下编译的选项，如图 3-8 所示。编译选项集中在“C/C++”选项卡中的“Project Options”里。内容为：

```
/nologo /Gz /MLd /W3 /WX /Z7 /Od /D WIN32=100 /D _X86_=1 /D WINVER=0x500 /D DBG=1  
/Fo"MyDriver_Check/" /Fd"MyDriver_Check/" /FD /c
```

- /nologo: 代表不显示编译的版权信息。
- /Gz: 默认函数调用采用标准调用 (`__stdcall`)。
- /W3: 采用第三级的警告模式。

- /WX: 将警告信息变成错误信息, 最大程度地保证了代码的可靠性。
- /Z7: 用 Z7 模式产生调试信息。VC 默认的 Program Database for "Edit & Continue", 这个和 link 的 /driver 选项冲突。
- /Od: 关闭调试模式。驱动程序不需要像 Win32 程序那样用 VC 调试器调试, 而需要用内核调试器调试。

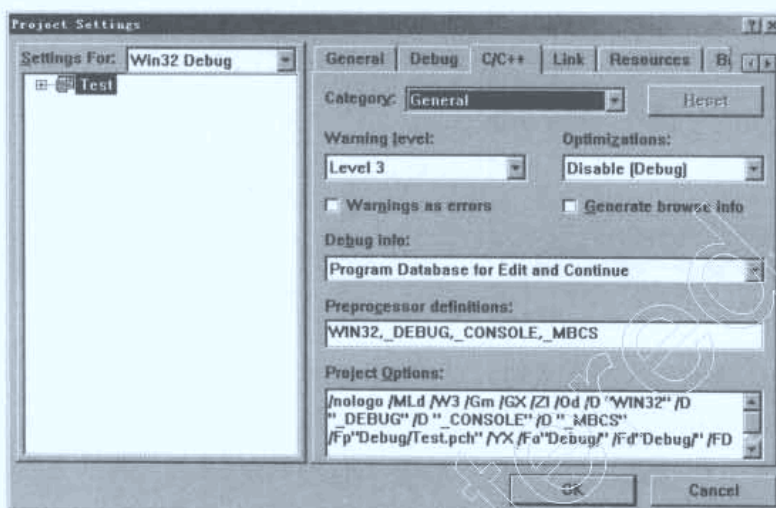


图 3-8 VC 中编译选项的修改

- /D WIN32=100 /D _X86_=1 /D WINVER=0x500 /D DBG=1: 定义一些宏, 这些是必需的。
- /Fo"MyDriver_Check/": 设置中间生成的目标代码的路径。
- /Fd"MyDriver_Check/": 设置 pdb 文件的目录位置, pdb 文件中包含了大量的符号, 这是调试驱动时候所必需的。
- /FD: 生成文件依赖。
- /c: 只进行编译, 而不链接。

3.3.3 修改链接选项

编译选项集中在“Link”选项卡中的“Project Options”里, 如图 3-9 所示。

在 Project Options 中填写:

```
ntoskrnl.lib /nologo /base:"0x10000" /stack:0x400000,0x1000 /entry:"DriverEntry"
/subsystem:console /incremental:no /pdb:"MyDriver_Check/HelloDDK.pdb" /debug
/machine:I386 /nodefaultlib /out:"MyDriver_Check/HelloDDK.sys" /pdbtype:sept
/subsystem:native /driver /SECTION:INIT,D /RELEASE /IGNORE:4078
```

- ntoskrnl.lib NT: 驱动程序需要链接此库。如果是 WDM 驱动程序, 则需要链接 wdm.lib。
- /nologo: 链接时不显示版权信息。

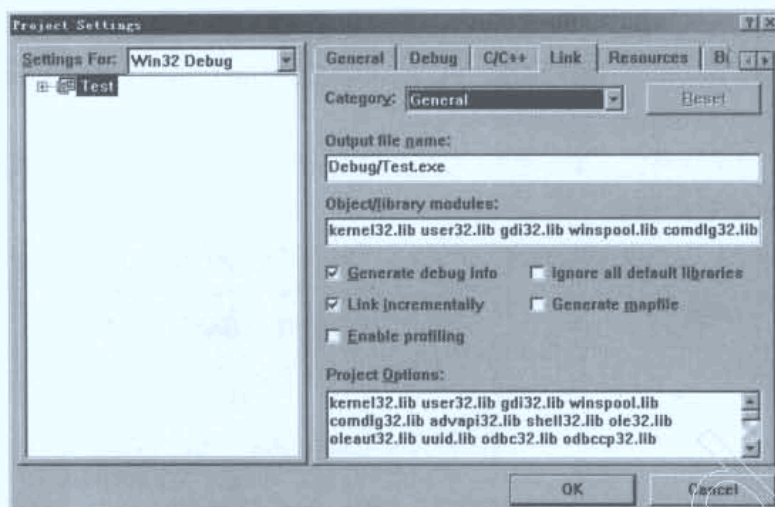


图 3-9 VC 中链接选项的修改

- /base:"0x10000": 加载驱动时, 设定加载在虚拟内存中的位置。
- /stack:0x400000, 0x1000: 设定函数使用堆栈的大小。
- /entry:"DriverEntry": 入口函数的地址, 此函数必须是符号标准函数调用的。
- /subsystem:console: 设置子系统。
- /incremental:no: 非递增式的链接。
- /pdb:"MyDriver_Check/HelloDDK.pdb": 设置 pdb 文件的文件名。
- /debug: 以 debug 方式链接。
- /machine:I386: 产生代码是 386 兼容平台的。
- /nodefaultlib: 不使用默认的库。
- /out:"MyDriver_Check/HelloDDK.sys": 输出二进制代码的名称。
- /pdbtype:sept: 设置 pdb 文件类型。
- /subsystem:native: 子系统是内核系统。
- /driver: 编译驱动。
- /SECTION:INIT, D: 将 INIT 的段设置为可抛弃的。
- /IGNORE:4078: 忽略 4078 号警告错误。

3.3.4 其他修改

在修改完编译和链接参数后, 还可以根据需要设定一些特殊的设置。例如, 可以将生成的 sys 驱动文件复制到系统目录下的\system32\drivers 目录中。在 Post-build step 中进行设定, 如图 3-10 所示。

```
copy $(TargetPath) $(WINDIR)\system32\drivers
```

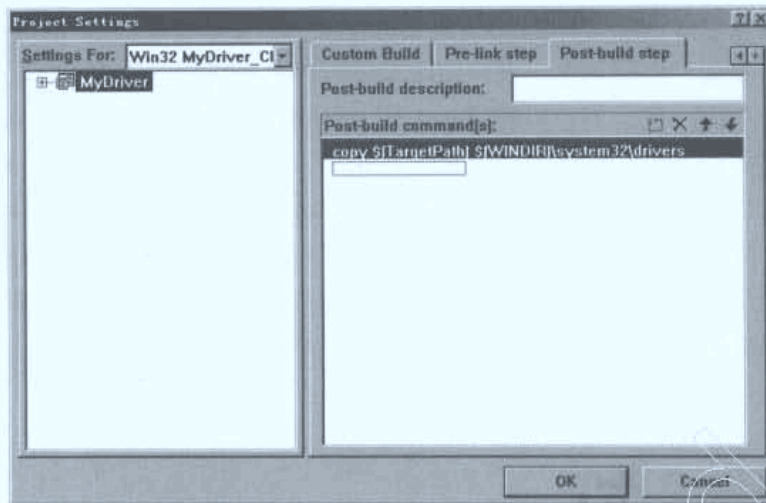


图 3-10 编译后附加动作

3.3.5 VC 编译小结

现在回忆一下用 VC 集成开发环境编译驱动程序步骤，再联想前面介绍的用 build 工具编译驱动程序，读者可以发现其本质是一样的。在这里做一个实验，在 VC 集成开发环境中，将编译选项和链接选项中的“/nologo”选项去掉，可以看出编译和链接的每一个步骤。

```
Deleting intermediate files and output files for project 'DriverDev - Win32 Debug'.
-----Configuration: DriverDev - Win32 Debug-----
Compiling...
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8168 for 80x86
Copyright (C) Microsoft Corp 1984-1998. All rights reserved.
cl /Gz /MLd /W3 /WX /Z7 /Od /D WIN32=100 /D _X86_=1 /D WINVER=0x500 /D DBG=1
/Fo"MyDriver_Check/" /Fd"MyDriver_Check/" /FD /c
"F:\chapter01\WDM_Driver\2\HelloWDM.cpp"
HelloWDM.cpp
Linking...
Microsoft (R) Incremental Linker Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.
wdm.lib "/base:0x10000" /stack:0x400000,0x1000 "/entry:DriverEntry"
/subsystem:console /incremental:no "/pdb:MyDriver_Check/HelloWDM.pdb" /debug
/machine:I386 /nodefaultlib "/out:MyDriver_Check/HelloWDM.sys" /pdbtype:sept
/subsystem:native /driver /
SECTION:INIT,D /RELEASE /IGNORE:4078
".\MyDriver_Check\HelloWDM.obj"

HelloWDM.sys - 0 error(s), 0 warning(s)
```

因此广义地说，有四种编译方法，而这四种方法其实质又是一样的。

- 方法一：手动一行行输入编译命令、行或者链接行。
- 方法二：建立 makefile，用 nmake 工具进行编译链接。
- 方法三：建立 makefile、sources、dirs 文件，用 build 工具编译链接。

➤ 方法四：用 VC 集成开发环境编译链接。

一般说来，方法一比较烦琐，需要记住大量的编译和链接参数。方法二比方法一先进了一点，用一个 `nmake` 命令就可以将整个工程全部编译，但是对于不同版本，或者不同平台，需要编写多套的 `makefile` 文件。方法三简化了 `makefile` 的编写，用 `sources` 和 `dirs` 描述需要编译的文件和编译参数。同时只需要一套 `makefile` 文件，就可以编译出不同的平台版本。方法四使用 VC 集成开发环境，不用记住命令行参数，只需单击鼠标即可设置完成。编译时只需单击菜单，并有详细的编译信息提供给用户。同时，在编译错误的时候，可以方便地定位到出错的地方。本书中大部分代码采用 VC 集成开发环境编译，部分代码采用 `build` 工具编译。

3.4 查看调试信息

编写驱动程序的程序员，经常遇到一个头疼的问题，这就是调试。因为驱动程序都是运行在内核模式下，很难像一般 Win32 程序一样进行调试。一般来说，驱动程序的调试主要有两个途径。其一是在关键的地方打印出调试信息，也就是俗称的打 `log`。其二是调用内核调试工具，诸如 `Softice` 或者 `WinDbg` 等，进行内核调试。本节主要介绍查看调试信息的方法。源代码的内核调试将在后面的章节陆续进行介绍。

3.4.1 打印调试语句

查看以前的 `HelloWDM` 和 `HelloDDK` 的代码，里面打印调试信息都是调用了语句 `KdPrint`。其实这不是一个函数，而是一个宏。在 `Checked` 版本中，它将参数传给了内核函数 `DbgPrint`，此函数会将调试信息记录下来。而在 `Free` 版本中，`KdPrint` 则什么也不做。

在驱动程序编写中，应该尽量用 `KdPrint`，而不用 `DbgPrint`。这样就能在 `Checked` 版本中，查看调试信息，而在真正发布 `Free` 版本的时候，将调试信息隐蔽起来。`KdPrint` 的功能十分类似于 MFC 提供给程序员的 `TRACE` 宏。`KdPrint` 的语法十分类似 `printf`，但由于是宏，所以使用的时候需要两重括号。

下面给出几种最常用的用法。

(1) 直接打印字符串。

```
KdPrint(("Enter HelloWDMAddDevice\n"));
```

(2) 打印字符串。

```
char * name = "Hello";  
KdPrint((" %s\n",name));
```

(3) 打印 UNICODE 字符或宽字符。

```
UNICODE_STRING devName;
.....初始化 unicode 操作等
KdPrint(("S\n", devName.Buffer)); //注意是大写的 S
KdPrint(("ws\n", devName.Buffer)); //或者是 ws
```

(4) 打印数字。

```
Int number=100;
KdPrint(("d\n",number));
```

(5) 打印十六进制数字。

```
Int number=100;
KdPrint(("X\n",number));
```

3.4.2 查看调试语句

很多种调试工具都可以查看调试信息。其中有 WinDbg, 还有 DriverMonitor。但在这里向读者介绍另外一个工具 DbgView, 这个软件是免费的, 读者可以在微软的网站找到它。

DbgView 可以同时监听内核和 Win32 上层应用程序发送的调试信息。如果接收的信息太多时, 用户还可以根据自己的需要过滤掉无用的调试信息。

首先开启 DbgView, 然后加载 HelloWDM 或者 HelloDDK 程序, 这时候驱动里调试信息就会一行行地打印到 DbgView 上, 如图 3-11 所示。

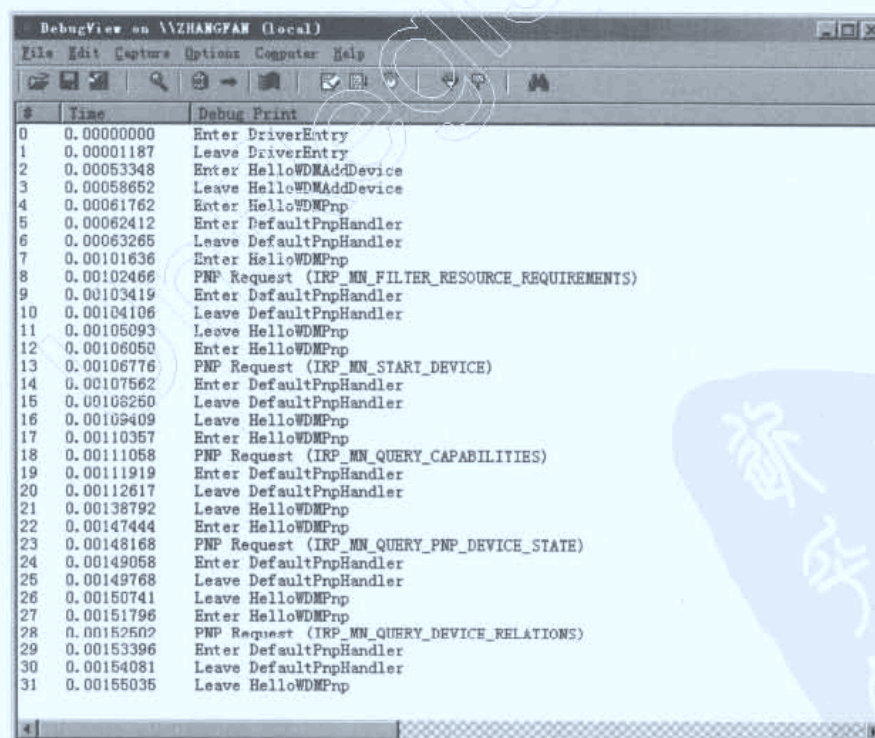


图 3-11 DbgView 查看调试信息

3.5 手动加载 NT 式驱动

在第 1 章曾经介绍过用 DriverMonitor 加载 NT 式驱动, 这里介绍另外一种手动的加载方式。其原理是一样的, 都是在注册表中填写相应的字段, 如图 3-12 所示。Windows 对 NT 式驱动程序的加载, 是基于服务的方式加载的, 类似于 Windows 服务程序的加载。

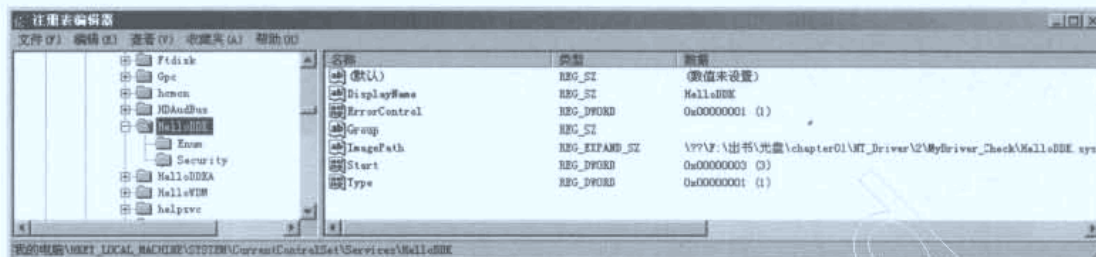


图 3-12 NT 驱动所需要修改的注册表

① 在 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services 中添加新项目, 项目名称为这个驱动名称, 例如, HelloDDK。

② 添加以下子键:

```

DisplayName (REG_SZ): HelloDDK
ErrorControl (REG_DWORD): 1
Group (REG_SZ):
ImagePath (REG_EXPAND_SZ):
\\?F:\出书\光盘\chapter01\NT_Driver\2\MyDriver_Check\HelloDDK.sys
Start (REG_DWORD): 3
Type (REG_DWORD): 1
  
```

DisplayName 的内容, 会在设备管理器中显示设备名字的时候显示出来。

ImagePath 要以 \\? 开头, 这代表符号链接。

Start 为 3, 表明规定按需要装入此驱动。

Type 为 1, 表明此驱动在内核模式下加载。

③ 在命令行方式运行 net start 驱动名称, 例如, net start HelloDDK。这时驱动会被 I/O 管理器加载, 并且调用入口函数 DriverEntry。停止驱动的方式很类似, 运行 net stop HelloDDK。如果不采用命令行的方式, 可以在设备管理器中运行或停止。如图 3-13 所示, 单击“启动”或“停止”按钮。

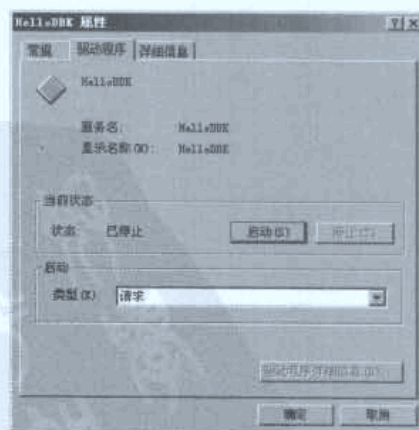


图 3-13 启动和停止 NT 驱动

3.6 编写程序加载 NT 式驱动

设备驱动程序的动态加载主要由服务控制管理程序 (Service Control Manager, SCM)

系统组件完成。SCM 组件为 Windows 2000 中运行的组件提供许多服务。例如，启动、停止和控制服务。服务是类似于 linux 的守护进程 (daemon)。编写加载驱动程序，主要是操作 SCM 组件。

3.6.1 SCM 组件和 Windows 服务

Windows 服务应用程序遵循服务控制管理器 (Service Control Manager)。Windows 服务可以在系统启动时加载，用户需在服务控制平台开启或者关闭服务。程序员可以通过 Windows 提供的相关服务函数进行加载或者卸载该服务。服务程序可以在用户没有登录系统的时候，就被载入系统并且被执行。Driver Service 是 Windows 服务的一个特例，它遵从 Windows 服务的协议。加载和卸载 NT 驱动分为四个步骤。

- ① 为 NT 驱动创建新的服务。
- ② 开启此项服务。
- ③ 关闭此项服务。
- ④ 删除 NT 驱动所创建的服务。

以上四个步骤，都是通过调用 SCM 组件的服务来实现的。下面列出常用的 SCM 组件 API 相关函数。

1. 打开 SCM 管理器函数

此函数的作用是打开 SCM 管理器，用于 SCM 的初始化。

```
SC_HANDLE OpenSCManager(
    LPCTSTR lpMachineName, // 计算机名称
    LPCTSTR lpDatabaseName, // SCM 数据库名称
    DWORD dwDesiredAccess // 使用权限
);
```

- lpMachineName: 指定计算机名称。如果为 NULL 代表是本机。
- lpDatabaseName: 指定 SCM 数据的名称。如果为 NULL 代表使用缺省数据库。
- dwDesiredAccess: 使用权限。一般设置为 SC_MANAGER_ALL_ACCESS。
- 返回值: 如果成功返回 SCM 管理器的句柄，如果失败返回 NULL。

2. 关闭服务句柄

此函数的作用是关闭 SCM 管理器的句柄，用于 SCM 的清除工作。

```
BOOL CloseServiceHandle(
    SC_HANDLE hSCObject // 要关闭的 SCM 句柄
);
```

- hSCObject: 要关闭的句柄。

3. 创建服务

此函数的作用是创建 SCM 管理器的句柄，后面介绍的操作都是基于这个句柄进行的。


```
SC_HANDLE CreateService(  
    SC_HANDLE hSCManager,        // SCM 管理器的句柄  
    LPCTSTR lpServiceName,       // 服务名称  
    LPCTSTR lpDisplayName,       // 服务显示出的名称  
    DWORD dwDesiredAccess,       // 打开权限  
    DWORD dwServiceType,         // 服务类型  
    DWORD dwStartType,           // 打开服务的时间  
    DWORD dwErrorControl,        // 关于错误处理的代码  
    LPCTSTR lpBinaryPathName,    // 二进制文件的代码  
    LPCTSTR lpLoadOrderGroup,    // 用何用户组开启服务  
    LPDWORD lpdwTagId,           // 输出验证标签  
    LPCTSTR lpDependencies,      // 所依赖的服务的名称  
    LPCTSTR lpServiceStartName,  // 用户账户名称  
    LPCTSTR lpPassword           // 用户口令  
);
```

- hSCManager: SCM 管理器的句柄, 也就是 OpenSCManager 打开的句柄。
- lpServiceName: 服务名称, 就是在设备管理器中看到的设备名称。
- dwDesiredAccess: 打开权限。如果没有特殊要求, 一般设置为 SERVICE_ALL_ACCESS。
- dwServiceType: 服务类型, 有以下几种选择。
 - SERVICE_FILE_SYSTEM_DRIVER: 文件系统的驱动。
 - SERVICE_KERNEL_DRIVER: 普通程序的驱动, 一般使用此项。
- dwStartType: 打开服务的时间, 有以下几种选择。
 - SERVICE_AUTO_START: 驱动自动加载。
 - SERVICE_BOOT_START: 被 system loader 加载, 即系统启动前就被启动。
 - SERVICE_DEMAND_START: 按照需要时启动, 一般选择此项。
- dwErrorControl: 关于错误处理的代码, 有以下几种选择。
 - SERVICE_ERROR_IGNORE: 遇到错误全部忽略掉。
 - SERVICE_ERROR_NORMAL: 遇到错误按照缺省办法处理。
 - SERVICE_ERROR_CRITICAL: 增加对错误处理的校验, 并提示出对话框, 并且记录错误信息到 log 文件中。
- lpBinaryPathName: 服务所用的二进制代码, 也就是编译后的驱动程序。

4. 打开服务

此函数的作用是针对已经创建过的服务, 再次打开此项服务。

```
SC_HANDLE OpenService(  
    SC_HANDLE hSCManager,        // SCM 数据库的句柄  
    LPCTSTR lpServiceName,       // 服务名称  
    DWORD dwDesiredAccess       // 访问权限  
);
```

- hSCManager: SCM 管理器的句柄, 也就是 OpenSCManager 打开的句柄。
- lpServiceName: 已经创建的服务名称。
- dwDesiredAccess: 打开权限。如果没有特殊要求, 一般设置为 SERVICE_ALL_ACCESS。

5. 控制服务

此函数的作用是对相应的服务, 发送控制码, 根据不同的控制码操作服务。

```
BOOL ControlService(  
    SC_HANDLE hService,           // 服务的句柄  
    DWORD dwControl,             // 控制码  
    LPSERVICE_STATUS lpServiceStatus // 返回状态码  
);
```

- hService: 服务的句柄, 也就是用 CreateService 创建的句柄, 或者 OpenService 打开的句柄。
- dwControl: 对服务的控制码, 此处列出常用的控制码。
 - SERVICE_CONTROL_CONTINUE: 针对暂停的服务发出继续运行的命令。
 - SERVICE_CONTROL_PAUSE: 针对正运行的服务发出暂停的命令。
 - SERVICE_CONTROL_STOP: 针对运行的服务发出停止的命令。
- dwDesiredAccess: 打开权限。如果没有特殊要求, 一般设置为 SERVICE_ALL_ACCESS。
- lpServiceStatus: 服务返回的状态码。

3.6.2 加载 NT 驱动的代码

笔者将加载驱动封装在 LoadNTDriver 函数里, 读者可在配套光盘中找到相应程序。函数输入参数为驱动程序的名称和驱动映像文件的路径名, 返回值代表加载是否成功。LoadNTDriver 的步骤如下:

- ① 调用 OpenSCManager, 打开 SCM 管理器。如果返回 NULL, 则返回失败, 否则继续。
- ② 调用 CreateService, 创建服务。创建成功则转步骤⑥。
- ③ 用 GetLastError 得到错误返回值。
- ④ 如果错误, 返回值为 ERROR_IO_PENDING, 说明服务已经创建过, 不需重新创建, 用 OpenService 打开此服务。
- ⑤ 如果错误, 返回值为其他值, 说明创建服务失败, 返回失败。
- ⑥ 调用 StartService 开启服务。
- ⑦ 成功返回。

其流程图如图 3-14 所示。

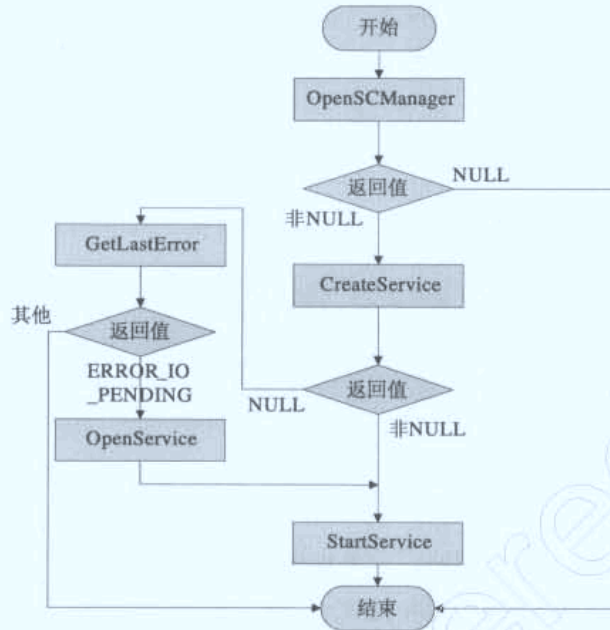


图 3-14 LoadNTDriver 流程图

```

#001 //装载 NT 驱动程序
#002 BOOL LoadNTDriver(char* lpszDriverName,char* lpszDriverPath)
#003 {
#004     char szDriverImagePath[256];
#005     //得到完整的驱动路径
#006     GetFullPathName(lpszDriverPath, 256, szDriverImagePath, NULL);
#007
#008     BOOL bRet = FALSE;
#009
#010     SC_HANDLE hServiceMgr=NULL;//SCM 管理器的句柄
#011     SC_HANDLE hServiceDDK=NULL;//NT 驱动程序的服务句柄
#012
#013     //打开服务控制管理器
#014     hServiceMgr = OpenSCManager( NULL, NULL, SC_MANAGER_ALL_ACCESS );
#015
#016     if( hServiceMgr == NULL )
#017     {
#018         //OpenSCManager 失败
#019         printf( "OpenSCManager() Failed %d ! \n", GetLastError() );
#020         bRet = FALSE;
#021         goto BeforeLeave;
#022     }
#023     else
#024     {
#025         //OpenSCManager 成功
#026         printf( "OpenSCManager() ok ! \n" );
#027     }
#028
#029     //创建驱动所对应的服务
#030     hServiceDDK = CreateService( hServiceMgr,
  
```

第3章 Windows 驱动编译环境配置、安装及调试

```
#031     lpszDriverName,           //驱动程序的在注册表中的名字
#032     lpszDriverName,           // 注册表驱动程序的 DisplayName 值
#033     SERVICE_ALL_ACCESS,        // 加载驱动程序的访问权限
#034     SERVICE_KERNEL_DRIVER,     // 表示加载的服务是驱动程序
#035     SERVICE_DEMAND_START,      // 注册表驱动程序的 Start 值
#036     SERVICE_ERROR_IGNORE,     // 注册表驱动程序的 ErrorControl 值
#037     szDriverImagePath,        // 注册表驱动程序的 ImagePath 值
#038     NULL,
#039     NULL,
#040     NULL,
#041     NULL,
#042     NULL);
#043
#044     DWORD dwRtn;
#045     //判断服务是否失败
#046     if( hServiceDDK == NULL )
#047     {
#048         dwRtn = GetLastError();
#049         if( dwRtn != ERROR_IO_PENDING && dwRtn != ERROR_SERVICE_EXISTS )
#050         {
#051             //由于其他原因创建服务失败
#052             printf( "CrateService() Failld %d ! \n", dwRtn );
#053             bRet = FALSE;
#054             goto BeforeLeave;
#055         }
#056         else
#057         {
#058             //服务创建失败,是由于服务已经创立过
#059             printf( "CrateService() Failld Service is ERROR_IO_PENDING or
ERROR_SERVICE_EXISTS! \n" );
#060         }
#061
#062         // 驱动程序已经加载, 只需要打开
#063         hServiceDDK = OpenService( hServiceMgr, lpszDriverName, SERVICE_ALL_
ACCESS );
#064         if( hServiceDDK == NULL )
#065         {
#066             //如果打开服务也失败, 则意味着错误
#067             dwRtn = GetLastError();
#068             printf( "OpenService() Failld %d ! \n", dwRtn );
#069             bRet = FALSE;
#070             goto BeforeLeave;
#071         }
#072         else
#073         {
#074             printf( "OpenService() ok ! \n" );
#075         }
#076     }
#077     else
#078     {
#079         printf( "CrateService() ok ! \n" );
#080     }
#081
#082     //开启此项服务
#083     bRet= StartService( hServiceDDK, NULL, NULL );
#084     if( !bRet )
#085     {
```



```
#086     DWORD dwRtn = GetLastError();
#087     if( dwRtn != ERROR_IO_PENDING && dwRtn != ERROR_SERVICE_ALREADY_RUNNING )
#088     {
#089         printf( "StartService() Failed %d ! \n", dwRtn );
#090         bRet = FALSE;
#091         goto BeforeLeave;
#092     }
#093     else
#094     {
#095         if( dwRtn == ERROR_IO_PENDING )
#096         {
#097             //设备被挂住
#098             printf( "StartService() Failed ERROR_IO_PENDING ! \n");
#099             bRet = FALSE;
#100             goto BeforeLeave;
#101         }
#102         else
#103         {
#104             //服务已经开启
#105             printf( "StartService() Failed ERROR_SERVICE_ALREADY_RUNNING !
\n");
#106             bRet = TRUE;
#107             goto BeforeLeave;
#108         }
#109     }
#110 }
#111 bRet = TRUE;
#112 //离开前关闭句柄
#113 BeforeLeave:
#114     if(hServiceDDK)
#115     {
#116         CloseServiceHandle(hServiceDDK);
#117     }
#118     if(hServiceMgr)
#119     {
#120         CloseServiceHandle(hServiceMgr);
#121     }
#122     return bRet;
#123 }
```

此段代码可以在配套光盘中本章的 LoadNTDriver 目录下找到。

3.6.3 卸载 NT 驱动的代码

UnloadNTDriver 函数作用是停止驱动并卸载驱动。函数的输入参数为驱动程序的名
称，返回值代表卸载驱动是否成功。UnloadNTDriver 的步骤如下：

① 调用 OpenSCManager，打开 SCM 管理器。如果返回 NULL，则返回失败，否则
继续。

② 调用 OpenService。如果返回 NULL，则返回失败，否则继续。

③ 调用 DeleteService 卸载此项服务。

④ 成功返回。

其流程图如图 3-15 所示。

第3章 Windows 驱动编译环境配置、安装及调试

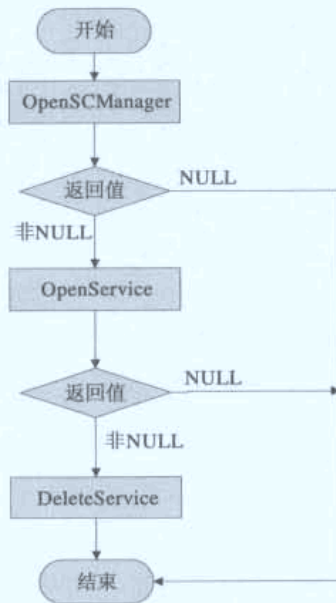


图 3-15 UnloadNTDriver 流程图

```
#001 //卸载驱动程序
#002 BOOL UnloadNTDriver( char * szSvrName )
#003 {
#004     BOOL bRet = FALSE;
#005     SC_HANDLE hServiceMgr=NULL; //SCM 管理器的句柄
#006     SC_HANDLE hServiceDDK=NULL; //NT 驱动程序的服务句柄
#007     SERVICE_STATUS SvrSta;
#008     //打开 SCM 管理器
#009     hServiceMgr = OpenSCManager( NULL, NULL, SC_MANAGER_ALL_ACCESS );
#010     if( hServiceMgr == NULL )
#011     {
#012         //带开 SCM 管理器失败
#013         printf( "OpenSCManager() Failed %d ! \n", GetLastError() );
#014         bRet = FALSE;
#015         goto BeforeLeave;
#016     }
#017     else
#018     {
#019         //带开 SCM 管理器失败成功
#020         printf( "OpenSCManager() ok ! \n" );
#021     }
#022     //打开驱动所对应的服务
#023     hServiceDDK = OpenService( hServiceMgr, szSvrName, SERVICE_ALL_ACCESS );
#024
#025     if( hServiceDDK == NULL )
#026     {
#027         //打开驱动所对应的服务失败
#028         printf( "OpenService() Failed %d ! \n", GetLastError() );
#029         bRet = FALSE;
#030         goto BeforeLeave;
#031     }
#032     else
#033     {
```



```

#034     printf( "OpenService() ok ! \n" );
#035     }
#036     //停止驱动程序, 如果停止失败, 只有重新启动才能, 再动态加载
#037     if( !ControlService( hServiceDDK, SERVICE_CONTROL_STOP , &SvrSta ) )
#038     {
#039         printf( "ControlService() Failed %d !\n", GetLastError() );
#040     }
#041     else
#042     {
#043         //打开驱动所对应的失败
#044         printf( "ControlService() ok !\n" );
#045     }
#046     //动态卸载驱动程序。
#047     if( !DeleteService( hServiceDDK ) )
#048     {
#049         //卸载失败
#050         printf( "DeleteService() Failed %d !\n", GetLastError() );
#051     }
#052     else
#053     {
#054         //卸载成功
#055         printf( "DeleteService() ok !\n" );
#056     }
#057     bRet = TRUE;
#058     BeforeLeave:
#059     //离开前关闭打开的句柄
#060     if(hServiceDDK)
#061     {
#062         CloseServiceHandle(hServiceDDK);
#063     }
#064     if(hServiceMgr)
#065     {
#066         CloseServiceHandle(hServiceMgr);
#067     }
#068     return bRet;
#069 }

```

这段代码可以在配套光盘中本章的 LoadNTDriver 目录下找到。

3.6.4 实验

为了实验前面加载和卸载函数的使用, 这里给出一个测试用例, 用于加载 HelloDDK。该测试用例分为以下三个步骤:

① 调用 LoadNTDriver, 加载驱动。如果调用成功, 则程序暂停住, 此时可以观测加载的情况。先观察注册表的前后变化, 再用 DebugView 查看驱动输出的 log, 是否运行了 DriverEntry 函数。同时再查看设备管理器中, 设备是否已经被加载。

② 按任意键继续, 运行编写的 TestDriver 函数, 此函数对驱动程序进行创建和关闭操作。此时观察 DebugView 是否有相应的输出 log。

③ 再次按任意键继续, 运行 UnloadNTDriver, 卸载驱动。此时再次观察注册表的变化, 会发现与注册表相关的项已经不存在了。另外, 观察设备管理器, 设备也不见了。同时, 观察 DebugView, 会发现驱动中 HelloDDKUnload 输出的 log 信息。

图 3-16 是控制台程序运行的情况。

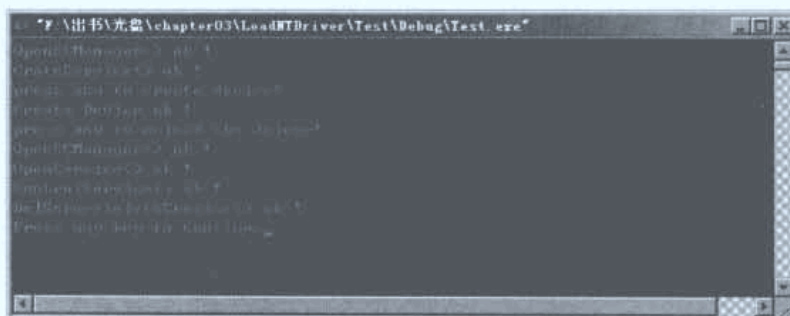


图 3-16 控制台运行情况

图 3-17 是 DebugView 输出 log 的情况。

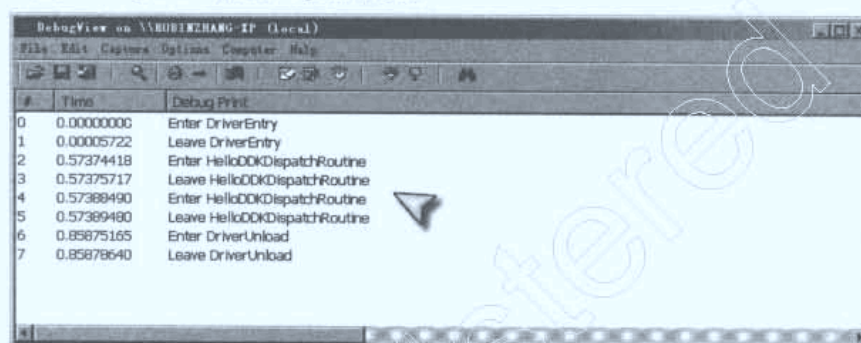


图 3-17 DebugView 输出情况

以下是相应的代码:

```
#001 int main(int argc, char* argv[])
#002 {
#003     //加载驱动
#004     BOOL bRet = LoadNTDriver(DRIVER_NAME, DRIVER_PATH);
#005     if (!bRet)
#006     {
#007         printf("LoadNTDriver error\n");
#008         return 0;
#009     }
#010     //这时候你可以通过注册表,或其他查看符号连接的软件验证
#011     printf("press any to unload the driver!\n");
#012     getch();
#013
#014     //卸载驱动
#015     UnloadNTDriver(DRIVER_NAME);
#016     if (!bRet)
#017     {
#018         printf("UnloadNTDriver error\n");
#019         return 0;
#020     }
#021
#022     return 0;
#023 }
```

此段代码可以在配套光盘中本章的 LoadNTDriver 目录下找到。


```
#001 void TestDriver()
#002 {
#003     //测试驱动程序
#004     HANDLE hDevice = CreateFile("\\\\.\\HelloDDK",
#005         GENERIC_WRITE | GENERIC_READ,
#006         0,
#007         NULL,
#008         OPEN_EXISTING,
#009         0,
#010         NULL);
#011     if( hDevice != INVALID_HANDLE_VALUE )
#012     {
#013         printf( "Open Driver ok ! \n" );
#014     }
#015     else
#016     {
#017         printf( "Open Driver failed %d ! \n", GetLastError() );
#018     }
#019     CloseHandle( hDevice );
#020 }
```

此段代码可以在配套光盘中本章的 LoadNTDriver 目录下找到。

3.7 WDM 式驱动的加载

和 NT 式驱动不同，WDM 式驱动程序不是被当做服务来加载的，因此不能简单地依靠修改注册表来加载驱动。WDM 式驱动的加载需要由一个以 INF 为扩展名的文本文件来描述安装的过程。

WDM 式驱动比 NT 式驱动增加了对即插即用的支持，这需要安装的时候提供一个 INF 文件进行配合。例如，在 PC 上新插入一个设备，系统会枚举到这个新设备，并报告操作系统这个设备的 VendorID 和 ProductID 等信息。

3.7.1 WDM 的手动安装

Windows 在安装的时候会提供很多 INF 文件，根据不同的 VendorID 和 ProductID，会找出合适这个设备的 INF 文件。如果系统中没有合适的 INF 文件，系统会向用户询问是否可以提供这个 INF 文件，如果不能则会到微软网站去寻找。经过一系列的定位 INF 文件步骤后，寻找到最终的 INF 文件。系统会根据 INF 文件上的指示，将驱动程序（.sys 文件）和相关文件复制到系统指定的目录下，并且修改注册表。同时会通知 PNP 管理器和 I/O 管理器，创建新设备，并运行驱动程序的入口程序 DriverEntry。INF 文件包含了 WDM 设备驱动程序需要的所有信息，这包括要创建或修改的注册表信息、复制的文件等。

在 Windows 里会默认安装大量的 INF 文件，这些 INF 文件记录了知名设备厂商提供的驱动程序。因此，大部分硬件安装在 Windows 的时候，是不需要安装驱动的。

当系统插入新设备时，如果没有找到相应驱动时，会提示用户安装方法，如图 3-18 所示。经过一系列的询问，系统会让用户定位 INF 文件的位置，如图 3-19 所示。

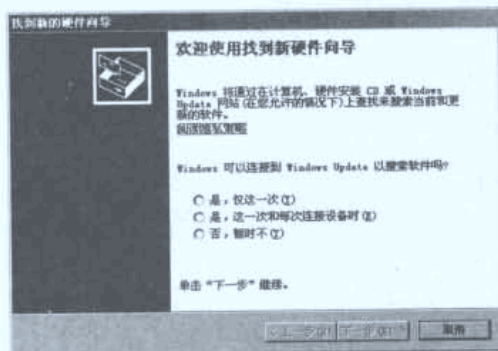


图 3-18 系统提示安装驱动程序

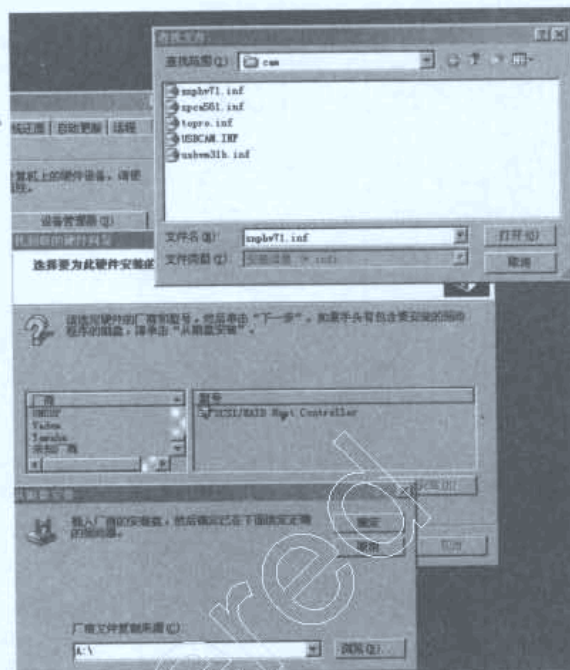


图 3-19 询问安装设备

3.7.2 简单的 INF 文件剖析

INF 文件是一个文本文件，由若干个节（Section）组成。每个节的名称用一个方括号指示，紧接着方括号后面的就是节内容。每一行就是一项内容，其形式都是类似 `SomeEntry=SomeValue`。每个项的顺序是可以颠倒的，但系统分析 INF 文件的时候，是顺序解析的。INF 中注释语句是用分号开头的。

下面以前面的 HelloWDM 的 INF 做简单分析，因为这是个虚拟设备，所以 `VernderID` 和 `ProductID` 用 `PCIVEN_9999&DEV_9999` 代替。

```
#001  ;; Win2K DDK 文档中有详细参考
#002
#003  ;----- 版本区域 -----
#004
#005  [Version]
#006  Signature="$CHICAGO$"
#007  Provider=Zhangfan_Device
#008  DriverVer=11/1/2007,3.0.0.3
#009
#010  ; 如果设备是一个标准类别，使用标准类的名称和 GUID
#011  ; 否则创建一个自定义的类别名称，并自定义它的 GUID
#012
#013  Class=ZhangfanDevice
#014  ClassGUID={EF2962F0-0D55-4bff-B8AA-2221EE8A79B0}
#015
#016
```



```

#017 ;----- 安装磁盘节-----
#018
#019 ; 这些节确定安装盘和安装文件的路径
#020 ;读者可以按照自己的需要修改.
#021
#022 [SourceDisksNames]
#023 1 = "HelloWDM",Disk1,,
#024
#025 [SourceDisksFiles]
#026 HelloWDM.sys = 1,MyDriver_Check,
#027
#028 ;----- ClassInstall/ClassInstall32 Section -----
#029
#030 ; 如果使用标准类别设备,下面的是不需要
#031
#032 ; 9X Style
#033 [ClassInstall]
#034 Addreg=Class_AddReg
#035
#036 ; NT Style
#037 [ClassInstall32]
#038 Addreg=Class_AddReg
#039
#040 [Class_AddReg]
#041 HKR,,,%DeviceClassName%
#042 HKR,,Icon,,*-5"
#043
#044 ;----- 目标文件节 -----
#045
#046 [DestinationDirs]
#047 YouMark_Files_Driver = 10,System32\Drivers
#048
#049 ;----- 制造商节 -----
#050
#051 [Manufacturer]
#052 %MfgName%=Mfg0
#053
#054 [Mfg0]
#055
#056 ; 在这里描述 PCI 的 VendorID 和 ProductID
#057 ; PCI\VEN_aaaa&DEV_bbbb&SUBSYS_cccccc&REV_dd
#058 ;改成你自己的 ID
#059 %DeviceDesc%=YouMark_DDI, PCI\VEN_9999&DEV_9999
#060
#061 ;----- DDInstall Sections -----
#062 ; ----- Windows 9X -----
#063
#064 ; 如果在 DDInstall 中的字符串超过 19, 将会导致严重的问题
#065 ;
#066
#067 [YouMark_DDI]
#068 CopyFiles=YouMark_Files_Driver
#069 AddReg=YouMark_9X_AddReg
#070
#071 [YouMark_9X_AddReg]
#072 HKR,,DevLoader,,*ntkern
#073 HKR,,NTMPDriver,,HelloWDM.sys
#074 HKR, "Parameters", "BreakOnEntry", 0x00010001, 0

```

```

#075
#076 ; ----- Windows NT -----
#077
#078 [YouMark_DDI.NT]
#079 CopyFiles=YouMark_Files_Driver
#080 AddReg=YouMark_NT_AddReg
#081
#082 [YouMark_DDI.NT.Services]
#083 Addservice = HelloWDM, 0x00000002, YouMark_AddService
#084
#085 [YouMark_AddService]
#086 DisplayName = %SvcDesc%
#087 ServiceType = 1 ; SERVICE_KERNEL_DRIVER
#088 StartType = 3 ; SERVICE_DEMAND_START
#089 ErrorControl = 1 ; SERVICE_ERROR_NORMAL
#090 ServiceBinary = %10%\System32\Drivers\HelloWDM.sys
#091
#092 [YouMark_NT_AddReg]
#093 HKLM, "System\CurrentControlSet\Services\HelloWDM\Parameters",\
#094 "BreakOnEntry", 0x00010001, 0
#095
#096
#097 ; ----- 文件节(common) -----
#098
#099 [YouMark_Files_Driver]
#100 HelloWDM.sys
#101
#102 ;----- 字符串节-----
#103
#104 [Strings]
#105 ProviderName="Zhangfan."
#106 MfgName="Zhangfan Soft"
#107 DeviceDesc="Hello World WDM!"
#108 DeviceClassName="Zhangfan_Device"
#109 SvcDesc="Zhangfan"

```

3.8 WDM 设备安装在注册表中的变化

WDM 式驱动程序的安装会在三个方面修改注册表，分别是硬件子键（Hardware）、类子键（Class）、服务子键（Service）。注册表从这三个方面的子键描述 WDM 设备。在安装好 WDM 驱动后，会根据 INF 的信息，在注册表中有所体现。

安装完 WDM 式驱动后，除了在注册表中得到体现外，在设备管理器中，设备会同时显示出来。在 INF 描述的各种信息，都可以从设备管理器中得到体现。请读者根据前面讲解 INF 文件的知识，对照下面的图示进行比较。读者可以根据自己的需要，修改前面给出的 INF 实例。

3.8.1 硬件子键

硬件子键，也称实例子键。其信息存储在注册表的 HKEY_LOCAL_MACHINE\SYSTEM\

CurrentControlSet\Enum 位置里, 见图 3-20。访问此子键必须拥有系统管理员的访问权限, 因此访问这个子键只能是运行在内核的程序或者拥有系统访问权限的应用程序, 如图 3-20 所示。

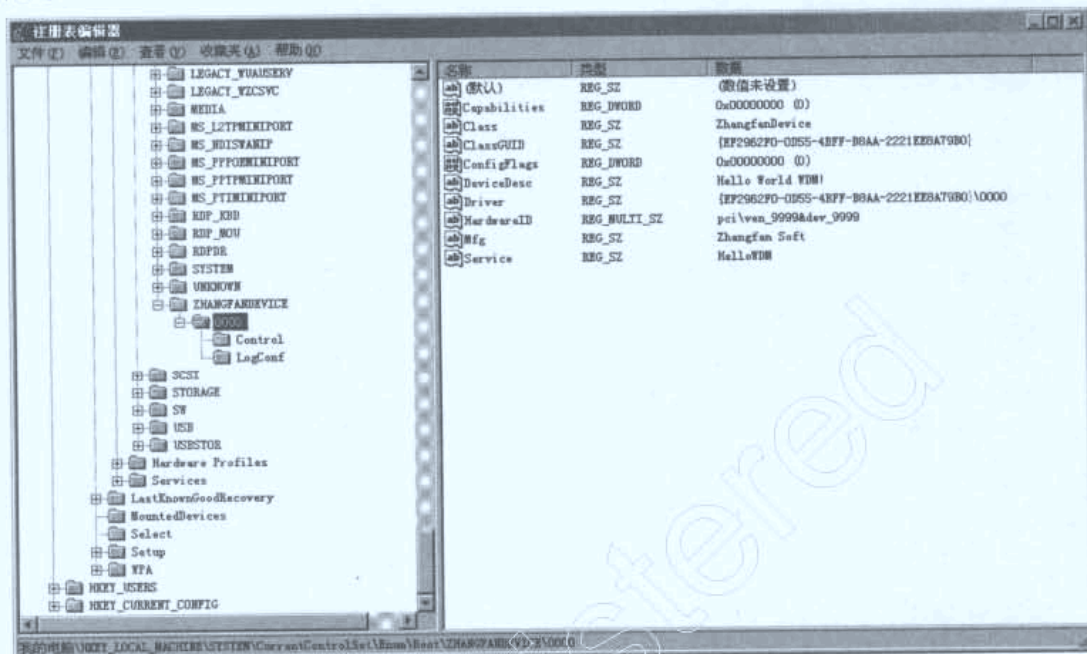


图 3-20 注册表中关于硬件子键的信息

HelloWDM 是个模拟设备, 所以位于 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum 目录下的 Root\ZHANGFANDEV\0000。这个可以根据设备管理中的详细信息中查到, 如图 3-21 所示。



图 3-21 硬件子键位置

可以想到, 如果 PC 中有多于一个的同类设备, 序号会顺序排列下去 0000、0001、0002...

观察注册表，会有如下内容，且这些内容会和 INF 中描述的一致。

```
"ClassGUID"="{EF2962F0-0D55-4BFF-B8AA-2221EE8A79B0}"
"Class"="ZhangfanDevice"
"HardwareID"= pci\ven_9999&dev_9999
"Driver"="{EF2962F0-0D55-4BFF-B8AA-2221EE8A79B0}\\0000"
"Mfg"="Zhangfan Soft"
"Service"="HelloWDM"
"DeviceDesc"="Hello World WDM!"
"ConfigFlags"=dword:00000000
"Capabilities"=dword:00000000
```

- ClassGUID: 指明此设备所属的类的 GUID。
- Class: 指明此设备所属于的设备类。
- HardwareID: 由设备的 ProductID 和 VernderID 联合组成。由于此设备是模拟设备，故为 pci\ven_9999&dev_9999。
- Driver: 指明驱动的位置。
- Mfg: 指明设备制造上的名称。
- Service: 指明此设备在服务子键的位置。
- DeviceDesc: 显示的是此设备在设备管理器中显示的名字。

图 3-22 显示的是在设备管理器中此种设备使用 HardwareID。图 3-23 显示的是设备管理器中所显示出来的设备名称，制造商名称等信息。



图 3-22 设备的 HardwareID

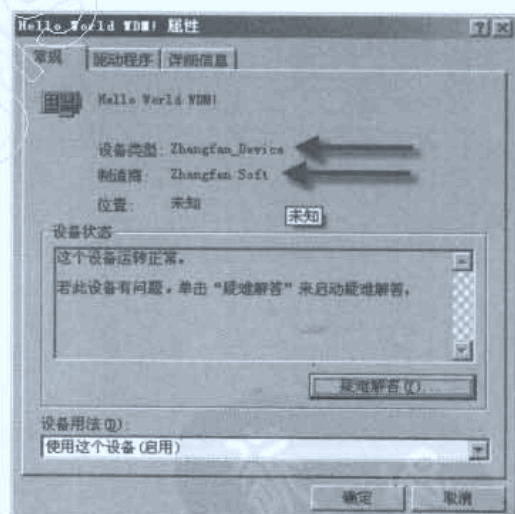


图 3-23 设备的制造商

3.8.2 类子键

每个设备都会从属于一个设备类。在 HelloWDM 中，该设备从属于 ZhangfanDevice 设备类。类子键负责记录这个类的信息。类子键位于 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Class 目录下。

这里记录了系统里所有的设备类，且每个设备类都是以一个 GUID 记录的。GUID 是一组 128 位的数字，形如{25DBCE51-6C8F-4A72-8A6D-B54C2B4FC835}。用户可以使用 GUIDGEN.exe 工具产生一组新的 GUID，且保证这个 GUID 号码不会与旧的 GUID 重复(是从小概率意义上保证的)。GUIDGEN.exe 的使用见图 3-24，它是 VC 提供的一个附加工具。

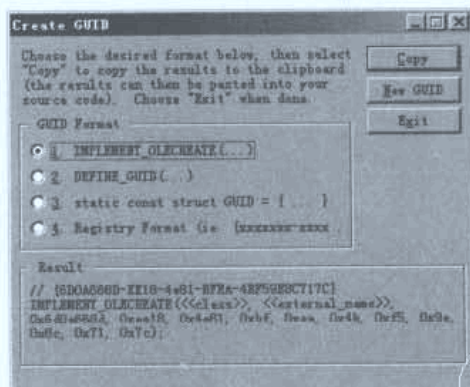


图 3-24 GUIDGEN 的使用

从 INF 文件中可以看出，HelloWDM 所属的类为 ZhangfanDevice，其 GUID 为 ClassGUID={E2962F0-0D55-4bff-B8AA-2221EE8A79B0}。所以可以在注册表中查到如下信息，如图 3-25 所示。

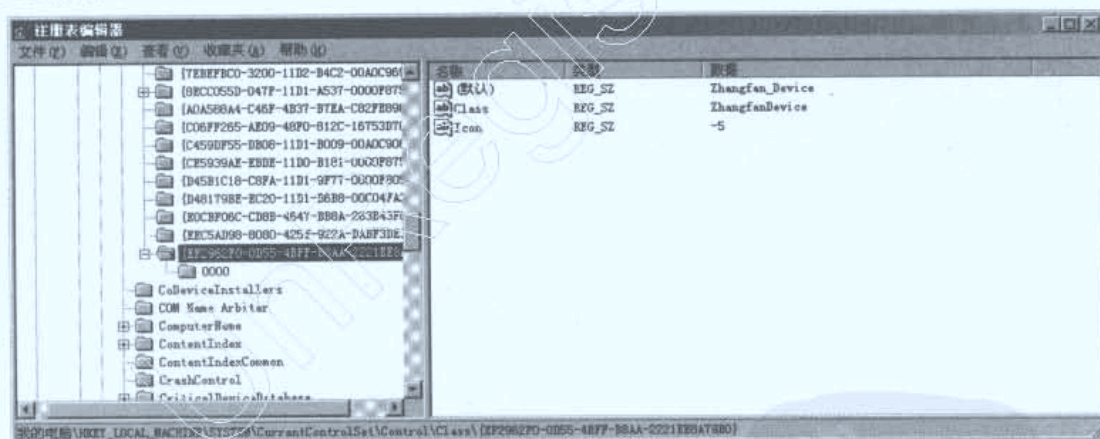


图 3-25 注册表中关于类子键的信息

其中，Icon 为 -5，这个图标是此类设备在设备管理器中的图标。读者可以更改此值，会得到很多有意思的图标。在此键的下面会有一个子键 0000，此子键的内容类似于硬件子键中的 0000。

```
"InfPath"="oem12.inf"
"InfSection"="YouMark_DDI"
"InfSectionExt"=".NT"
"ProviderName"="Zhangfan_Device"
"DriverDateData"=hex:00,00,44,25,1a,1c,c8,01
"DriverDate"="11-1-2007"
```

```
"DriverVersion"="3.0.0.3"
"MatchingDeviceId"="pci\ven_9999&dev_9999"
"DriverDesc"="Hello World WDM!"
```

这些内容，读者可以对照 INF 的片段更改，且同时作用于设备管理器中的内容。

下图是设备管理器中显示的设备类的名称和设备的名称，且图标标号为-5 的图标，读者可以更改此标号，会得到不同类型的图标，如图 3-26 所示。图 3-27 是在设备管理器中所显示出来的驱动供应商，驱动版本日期，数字签名等信息。

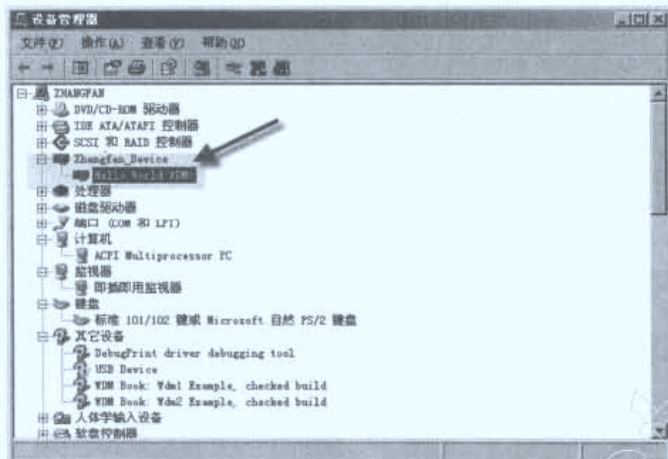


图 3-26 设备管理器中设备类和设备名称

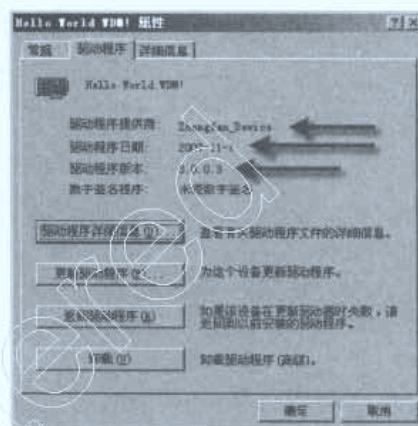


图 3-27 设备管理器中的驱动版本信息

3.8.3 服务子键

服务子键是为了兼容以前的 NT 式驱动程序，读者可以用同样的方法观察 NT 式驱动程序的服务子键。服务子键的位置在 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services 目录下。观察 HelloWDM 的服务子键，如图 3-28 所示。



图 3-28 注册表中设备的服务子键

```
"Type"=dword:00000001
"Start"=dword:00000003
```



```
"ErrorControl"=dword:00000001
"ImagePath"= System32\Drivers\HelloWDM.sys
"DisplayName"="Zhangfan"
```

- ImagePath 为 System32\Drivers\HelloWDM.sys，记录驱动程序的执行文件路径。
 - Type 为 1，指示该表描述一个内核模式的驱动。
 - Start 为 3，指示系统应动态装入这个驱动程序。此值会与 SERVICE_DEMOND_START 常量对应。
 - ErrorControl 为 1，指出如果装入驱动程序失败，系统应弹出一个错误提示对话框。
- 图 3-29 是设备管理器中此种设备所用的服务的名称。



图 3-29 设备管理器中的服务

3.9 小结

本章详细介绍了编译驱动程序的两种方法。一种方法是通过 build 脚本编译驱动程序，另一种方法是通过 VC 的 IDE 环境编写。另外，本章还介绍了 NT 驱动程序和 WDM 驱动程序的加载方式。NT 驱动程序是通过 Windows 服务进行加载，而 WDM 驱动程序需要借助 INF 文件进行加载。另外，本章还介绍了驱动程序加载后，注册表的变化情况。同时，本章还介绍了如何利用工具软件查看驱动程序输出的调试信息。最后，本章还介绍了 C 语言和 C++ 语言在编写驱动程序中需要注意的地方。

第4章 驱动程序的基本结构

本章首先对 Windows 驱动程序的两个重要数据结构进行介绍,分别是驱动对象和设备对象数据结构。另外还要介绍 NT 驱动程序和 WDM 驱动程序的入口函数、卸载例程、各种 IRP 派遣上函数等。

4.1 Windows 驱动程序中重要的数据结构

数据结构是计算机程序的核心, I/O 管理器定义了一些数据结构, 这些数据结构是编写驱动程序时所必须掌握的。驱动程序经常要创建和维护这些数据结构的实例。本节将简要讨论这些数据结构, 并且说明如何使用这些数据结构。

在 DDK 文档中, 这些数据结构都有非常详细的注释, 笔者这里重点讲述为什么要定义这些数据结构以及如何使用。

4.1.1 驱动对象 (DRIVER_OBJECT)

每个驱动程序会有唯一的驱动对象与之对应, 并且这个驱动对象是在驱动加载的时候, 被内核中的对象管理程序所创建的。

驱动对象用 DRIVER_OBJECT 数据结构表示, 它作为驱动的一个实例被内核加载, 并且内核对一个驱动只加载一个实例。确切地说, 是由内核中的 I/O 管理器负责加载的。驱动程序需要在 DriverEntry 中初始化。先了解一下驱动对象的数据结构。

```
typedef struct _DRIVER_OBJECT {
    CSHORT Type;
    CSHORT Size;
    PDEVICE_OBJECT DeviceObject;
    ULONG Flags;
    PVOID DriverStart;
    ULONG DriverSize;
    PVOID DriverSection;
```



```
PDRIVER_EXTENSION DriverExtension;
UNICODE_STRING DriverName;
PUNICODE_STRING HardwareDatabase;
PFAST_IO_DISPATCH FastIoDispatch;
PDRIVER_INITIALIZE DriverInit;
PDRIVER_STARTIO DriverStartIo;
PDRIVER_UNLOAD DriverUnload;
PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];
} DRIVER_OBJECT;
typedef struct _DRIVER_OBJECT *PDRIVER_OBJECT;
```

这里列出了完整的驱动对象的定义，其中有几个重要字段需要理解。

- **DeviceObject**: 每个驱动程序会有一个或多个设备对象。其中，每个设备对象都有一个指针指向下一个驱动对象，最后一个设备对象指向空。此处的 DeviceObject 指向驱动对象的第一个设备对象。通过 DeviceObject，就可以遍历驱动对象里的所有设备对象。设备对象是由程序员自己创建的，而非操作系统完成，在驱动被卸载的时候，遍历每个设备对象，并将其删除。
- **DriverName**: 顾名思义，DriverName 记录的是驱动程序的名字。这里用 UNICODE 字符串记录，该字符串一般为 \Driver[驱动程序名称]。
- **HardwareDatabase**: 这里记录的是设备的硬件数据库键名，这里同样用 UNICODE 字符串记录。该字符串一般为 \REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM。
- **DriverStartIo**: 记录 StartIO 例程的函数地址，用于串行化操作，后面会有具体讲解。
- **DriverUnload**: 指定驱动卸载时所用的回调函数地址。
- **MajorFunction**: MajorFunction 域记录的是一个函数指针数组，也就是 MajorFunction 是一个数组，数组中的每个成员记录着一个指针，每一个指针指向的是一个函数。这个函数就是处理 IRP 的派遣函数。
- **FastIoDispatch**: 文件驱动中用到的派遣函数，本书没有涉及文件驱动，请参阅 MSDN。

为了更加清楚地了解 DRIVER_OBJECT 结构，笔者将上述结构归纳成图表，如图 4-1 所示。

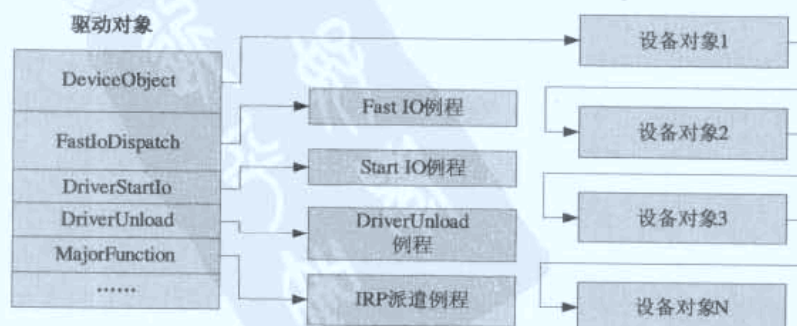


图 4-1 驱动对象的布局

4.1.2 设备对象 (DEVICE_OBJECT)

每个驱动程序会创建一个或多个设备对象，用 `DEVICE_OBJECT` 数据结构表示。每个设备对象都会有一个指针指向下一个设备对象，因此就形成一个设备链。设备链的第一个设备是由上一节介绍的 `DRIVER_OBJECT` 结构体中指明的。设备对象保存设备特征和状态的信息，其数据结构定义如下：

```
typedef struct _DEVICE_OBJECT {
    ...
    struct _DRIVER_OBJECT *DriverObject;
    struct _DEVICE_OBJECT *NextDevice;
    struct _DEVICE_OBJECT *AttachedDevice;
    struct _IRP *CurrentIrp;
    ULONG Flags;
    struct _DEVOBJ_EXTENSION *DeviceObjectExtension;
    ...
} DEVICE_OBJECT;
typedef struct _DEVICE_OBJECT *PDEVICE_OBJECT; // ntddis
```

下面对重要的字段进行说明。

- **DriverObject**: 指向驱动程序中的驱动对象。同属于一个驱动程序的驱动对象指向的是统一驱动对象。
- **NextDevice**: 指向下一个设备对象。这里指的下一个设备对象是同属于一个驱动对象的设备，也就是同一个驱动程序创建的若干设备对象。每个设备对象根据 `NextDevice` 域形成链表，从而可以枚举每个设备对象。
- **AttachedDevice**: 指向下一个设备对象。这里指的是，如果有更高一层的驱动附加到这个驱动的时候，`AttachedDevice` 指向的就是那个更高一层的驱动。
- **CurrentIrp**: 在使用 `StartIO` 例程的时候，此域指向的是的当前 IRP 结构。关于 `StartIO`，详见第 9 章。
- **Flags**: 此域是一个 32 位的无符号整型。每一个位有具体的含义，其主要位如表 4-1 表示。

表 4-1 设备对象中的 Flags

标 志	描 述
<code>DO_BUFFERED_IO</code>	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
<code>DO_EXCLUSIVE</code>	一次只允许一个线程打开设备句柄
<code>DO_DIRECT_IO</code>	读写操作使用直接方式（内存描述符表）访问用户模式数据
<code>DO_DEVICE_INITIALIZING</code>	设备对象正在初始化
<code>DO_POWER_PAGABLE</code>	必须在 <code>PASSIVE_LEVEL</code> 级上处理 <code>IRP_MJ_PNP</code> 请求
<code>DO_POWER_INRUSH</code>	设备上电期间需要大电流

- **DeviceExtension**: 指向的是设备的扩展对象。每个设备都会指定一个设备扩展对象，

设备扩展对象记录的是设备自己特殊定义的结构体，也就是由程序员自己定义的结构体。另外，在驱动程序中，应该尽量避免全局变量的使用，因为全局变量涉及不容易同步的问题。解决的办法，将全局变量存在设备扩展里。

➤ DeviceType: 指明设备的类型。常用的类型如表 4-1 所示。

表 4-2 DeviceType

设备类型	描 述
FILE_DEVICE_BEEP	蜂鸣器设备对象
FILE_DEVICE_CD_ROM	CD 光驱设备对象
FILE_DEVICE_CD_ROM_FILE_SYSTEM	CD 光驱文件系统设备对象
FILE_DEVICE_CONTROLLER	控制器设备对象
FILE_DEVICE_DATAINK	数据链设备对象
FILE_DEVICE_DFS	DFS 设备对象
FILE_DEVICE_DISK	磁盘设备对象
FILE_DEVICE_DISK_FILE_SYSTEM	磁盘文件系统设备对象
FILE_DEVICE_FILE_SYSTEM	文件系统设备对象
FILE_DEVICE_INPORT_PORT	输入端口设备对象
FILE_DEVICE_KEYBOARD	键盘设备对象
FILE_DEVICE_MAILSLOT	邮件槽设备对象
FILE_DEVICE_MIDI_IN	MIDI 输入设备对象
FILE_DEVICE_MIDI_OUT	MIDI 输出设备对象
FILE_DEVICE_MOUSE	鼠标设备对象
FILE_DEVICE_MULTI_UNC_PROVIDER	多 UNC 设备对象
FILE_DEVICE_NAMED_PIPE	命名管道设备对象
FILE_DEVICE_NETWORK	网络设备对象
FILE_DEVICE_NETWORK_BROWSER	网络浏览器设备对象
FILE_DEVICE_NETWORK_FILE_SYSTEM	网络文件系统设备对象
FILE_DEVICE_NULL	空设备对象
FILE_DEVICE_PARALLEL_PORT	并口设备对象
FILE_DEVICE_PHYSICAL_NETCARD	物理网卡设备对象
FILE_DEVICE_PRINTER	打印机设备对象
FILE_DEVICE_SCANNER	扫描仪设备对象
FILE_DEVICE_SERIAL_MOUSE_PORT	串口鼠标设备对象
FILE_DEVICE_SERIAL_PORT	串口设备对象
FILE_DEVICE_SCREEN	屏幕设备对象
FILE_DEVICE_SOUND	声音设备对象
FILE_DEVICE_STREAMS	流设备对象
FILE_DEVICE_TAPE	磁带设备对象
FILE_DEVICE_TAPE_FILE_SYSTEM	磁带文件系统设备对象
FILE_DEVICE_TRANSPORT	传输设备对象

续表

设备类型	描 述
FILE_DEVICE_UNKNOWN	未知设备对象
FILE_DEVICE_VIDEO	视频设备对象
FILE_DEVICE_VIRTUAL_DISK	虚拟磁盘设备对象
FILE_DEVICE_WAVE_IN	声音输入设备对象
FILE_DEVICE_WAVE_OUT	声音输出设备对象
FILE_DEVICE_8042_PORT	8042 端口设备对象
FILE_DEVICE_NETWORK_REDIRECTOR	网卡设备对象
FILE_DEVICE_BATTERY	电池设备对象
FILE_DEVICE_BUS_EXTENDER	总线扩展设备对象
FILE_DEVICE_MODEM	调制解调器设备对象
FILE_DEVICE_VDM	VDM 设备对象
FILE_DEVICE_MASS_STORAGE	大容量存储设备对象
FILE_DEVICE_SMB	SMB 设备对象
FILE_DEVICE_KS	内核流设备对象
FILE_DEVICE_CHANGER	充电设备对象
FILE_DEVICE_SMARTCARD	智能卡设备对象
FILE_DEVICE_ACPI	ACPI 设备对象
FILE_DEVICE_DVD	DVD 设备对象

根据设备的需要，需要填写相应的设备类型。当制作虚拟设备时，应选择 FILE_DEVICE_UNKNOWN 类型的设备。

- StackSize: 在多层驱动情况下，驱动与驱动之间会形成类似堆栈的结构。IRP 会依次从最高层传递到最底层。StackSize 描述的就是这个层数。
- AlignmentRequirement: 设备在大容量传输的时候，需要内存对齐，以保证传输速度。

为了更加清楚地了解 DEVICE_OBJECT 结构，笔者将上述结构归纳成图，如图 4-2 所示。

4.1.3 设备扩展

设备对象记录“通用”设备的信息，而另外一些“特殊”信息记录在设备扩展里。各个设备扩展由程序员自己定义，每个设备的设备扩展也不尽相同。设备扩展是由程序员指定内容和大小，由 I/O 管理器创建的，并保存在非分页内存中。

在驱动程序中，尽量避免使用全局函数，因为全局函数往往导致函数的不可重入性。重入性指的是，在多线程的程序中，多个函数并行运行，函数的运行结果不会根据函数的调用先后顺序而导致不同。解决的办法是，将全局变量以设备扩展的形式存储，并加以适当的同步保护措施。除此之外，在设备扩展中，还会记录下列一些内容：

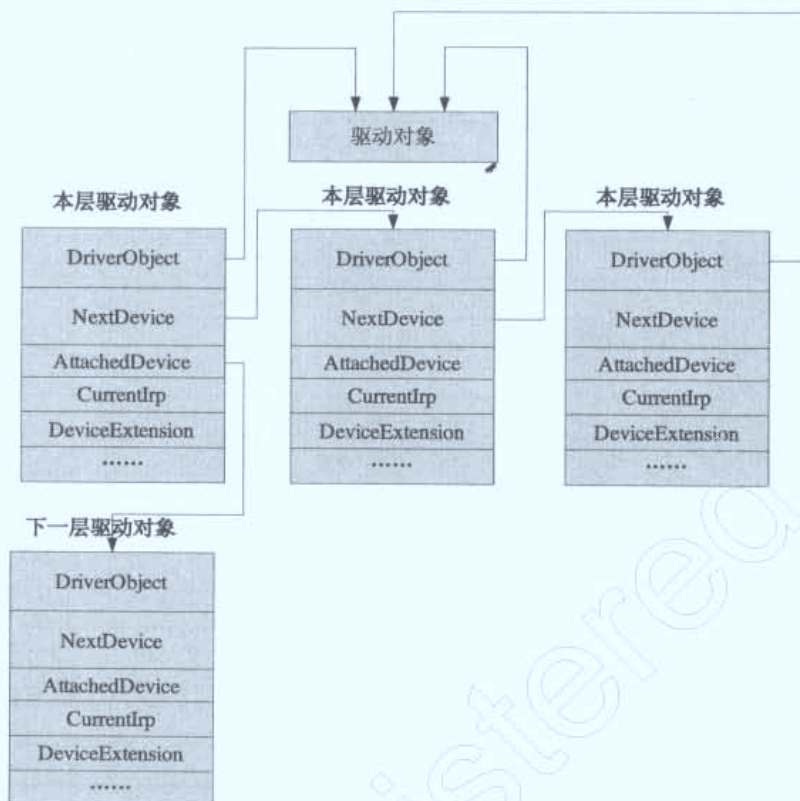


图 4-2 设备对象结构

- 设备对象的反向指针。
- 设备状态或驱动环境信息。
- 中断对象指针。
- 控制器对象指针。

由于设备扩展是驱动程序专用的，它的结构必须在驱动程序的头文件中定义。

4.2 NT 式驱动的基本结构

对于 NT 式驱动来说，主要的函数是 DriverEntry 例程、卸载例程及各个 IRP 的派遣例程。

4.2.1 驱动加载过程与驱动入口函数（DriverEntry）

和编写普通应用程序一样，驱动程序有个入口函数，也就是首先被执行的函数。这个函数通常被命名为 DriverEntry，读者可以指定另外的名字，但最好遵循这个名字。该函数的原型如下：

```
NTSTATUS DriverEntry(    IN PDRIVER_OBJECT pDriverObject,
                      IN PUNICODE_STRING pRegistryPath);
```

DriverEntry 主要是对驱动程序进行初始化工作,它是由系统进程所调用的。在 Windows 中有个特殊的进程叫做系统进程,打开进程管理器,里面有个名为 System 的进程就是系统进程。系统进程在系统启动的时候,就已经被创建了。

驱动加载的时候,系统进程启动新的线程,调用执行体组件中的对象管理器,创建一个驱动对象。这个驱动对象是一个 DRIVER_OBJECT 的结构体。另外,系统进程调用执行体组件中的配置管理程序,查询此驱动程序对应的注册表中的项。

系统线程调用驱动程序的 DriverEntry 例程时,同时传进两个参数,分别是 pDriverObject 和 pRegistryPath。其中,一个是指向刚才被创建驱动对象的指针,另外一个是指向设备服务键的键名字符串的指针。在 DriverEntry 中,主要功能是对系统进程创建的驱动对象进行初始化。(有关驱动对象的介绍,请参考下一节)。另外,设备服务键的键名有时候需要保存下来,因为这个字符串不是长期存在的(函数返回后可能消失)。如果以后想使用这个 UNICODE 字符串就必须先把它复制到安全的地方。

这个字符串的内容一般是\REGISTRY\MACHINE\SYSTEM\ControlSet\Services\[服务名]。在驱动程序中,字符串用 UNICODE 字符串来表示。UNICODE 是宽字符集,每个字符用 16 位表示。

其中,UNICODE 用数据结构 UNICODE_STRING 表示:

```
typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR Buffer;
} UNICODE_STRING *PUNICODE_STRING;
```

- Length: 记录这个字符串用多少字节记录。如果字符串有 N 个字符,那么 Length 将会是 N 的 2 倍。
- MaximumLength: 记录 buffer 的大小,也就是这个结构最大能记录的字节数。MaximumLength 要大于或等于 Length。
- Buffer: 记录字符串的指针。与 ASCII 字符串不同,这里的字符串每个字符都是 16 位。

在驱动中可以使用 KdPrint 打印 UNICODE 的信息。其语法是:

```
KdPrint(("S\n",pRegistryPath->Buffer));
```

或

```
KdPrint(("ws\n",pRegistryPath->Buffer));
```

DriverEntry 返回值是 NTSTATUS 的数据,NTSTATUS 是被定义为 32 位的无符号长整型。在驱动程序开发中,人们习惯用 NTSTATUS 返回状态。其中 0~0X7FFFFFFF,被认为是正确的状态,而 0X80000000~0xFFFFFFFF,被认为是错误的状态。有个非常有用的宏——

NT_SUCCESS, 被用来检测状态是否正确。

常用的 NTSTATUS 值有:

```
#define STATUS_SUCCESS                (NTSTATUS)0x00000000L)
#define STATUS_BUFFER_OVERFLOW        (NTSTATUS)0x80000005L)
#define STATUS_UNSUCCESSFUL           (NTSTATUS)0xC0000001L)
#define STATUS_NOT_IMPLEMENTED        (NTSTATUS)0xC0000002L)
#define STATUS_ACCESS_VIOLATION       (NTSTATUS)0xC0000005L)
#define STATUS_INVALID_HANDLE         (NTSTATUS)0xC0000008L)
#define STATUS_INVALID_PARAMETER      (NTSTATUS)0xC000000DL)
```

DriverEntry 的返回值如果表示成功, 则意味着加载驱动成功, 否则意味着加载驱动失败, 调用对象管理程序销毁驱动对象。

最后需要说明的是 DriverEntry 参数的修饰“IN”。“IN”、“OUT”、“INOUT”在 DDK 中都被定义成空串, 它们的功能类似于程序注释, 当看到一个“IN”参数时, 应该认定该参数是纯粹用于输入目的。“OUT”参数代表这个参数仅用于函数的输出参数。“INOUT”用于既可以输入又可以输出的参数。例如 DriverEntry 例程, 它的 DriverObject 指针是 IN 参数, 即使用者不能改变这个指针本身, 但完全可以改变它指向的对象。

另外需要注意的是在 C 语言和 C++ 编写时候的细微差别, 请参考第 3 章中关于 C 语言和 C++ 编译的讲解。这里再次将第 1 章中的 HelloDDK 的 DriverEntry 打开, 可以看出 DriverEntry 相对比较简单。

```
#001 extern "C" NTSTATUS DriverEntry (
#002     IN PDRIVER_OBJECT pDriverObject,
#003     IN PUNICODE_STRING pRegistryPath )
#004 {
#005     NTSTATUS status;
#006     KdPrint(("Enter DriverEntry\n"));
#007
#008     //注册其他驱动调用函数入口
#009     pDriverObject->DriverUnload = HelloDDKUnload;
#010     pDriverObject->MajorFunction[IRP_MJ_CREATE] = HelloDDKDispatchRoutine;
#011     pDriverObject->MajorFunction[IRP_MJ_CLOSE] = HelloDDKDispatchRoutine;
#012     pDriverObject->MajorFunction[IRP_MJ_WRITE] = HelloDDKDispatchRoutine;
#013     pDriverObject->MajorFunction[IRP_MJ_READ] = HelloDDKDispatchRoutine;
#014
#015     //创建驱动设备对象
#016     status = CreateDevice(pDriverObject);
#017
#018     KdPrint(("DriverEntry end\n"));
#019     return status;
#020 }
```

此段代码可以在配套光盘中本章的 NT_Driver 目录下找到。

在 DriverEntry 函数中, 一般设置卸载例程数和 IRP 的派遣函数, 另外还有一部分代码负责创建设备对象。设置卸载例程和设置派遣函数都是对驱动对象的设置。设备对象中的 MajorFunction 是一个函数指针数组, IRP_MJ_CREATE、IRP_MJ_CLOSE、IRP_MJ_WRITE 代表数组的第几个元素。在 HelloDDK 中, 所有派遣函数都设置为

HelloDDKDispatchRoutine。

4.2.2 创建设备对象

在 NT 式的驱动中，创建设备对象是由 IoCreateDevice 内核函数完成的。

```
NTSTATUS
IoCreateDevice(
    IN PDRIVER_OBJECT DriverObject,
    IN ULONG DeviceExtensionSize,
    IN PUNICODE_STRING DeviceName OPTIONAL,
    IN DEVICE_TYPE DeviceType,
    IN ULONG DeviceCharacteristics,
    IN BOOLEAN Exclusive,
    OUT PDEVICE_OBJECT *DeviceObject
);
```

- DriverObject: 输入参数，每个驱动程序中。会有唯一的驱动对象与之对应，但每个驱动对象会有若干个设备对象。DriverObject 指向的就是驱动对象的指针。
- DeviceExtensionSize: 输入参数，指定设备扩展的大小，I/O 管理器会根据这个大小，在内存中创建设备扩展，并与驱动对象关联。
- DeviceName: 输入参数，设置设备对象的名字。
- DeviceCharacteristics: 输入参数，设置设备对象的特征。
- Exclusive: 输入参数，设置设备对象是否为内核模式下使用，一般设置为 TRUE。
- DeviceObject: 输出参数，I/O 管理器负责创建这个设备对象，并返回设备对象的地址。
- 返回值: 返回此函数的调用状态。

设备名称用 UNICODE 字符串指定，并且字符串必须是“\Device\[设备名]”的形式。在 Windows 下的所有设备都是以类似名字命名的，例如，磁盘分区的 C 盘、D 盘、E 盘、F 盘就是被命名为“\Device\HarddiskVolume1”、“\Device\HarddiskVolume2”、“\Device\HarddiskVolume3”、“\Device\HarddiskVolume4”。

当然也可以不指定设备名字，如果在 IoCreateDevice 中没有指定设备对象的名字，I/O 管理器会自动分配一个数字作为设备的设备名，例如，“\Device\00000001”、“\Device\00000002”、“\Device\00000003”。

如果指定了设备名，只能被内核模式下的其他驱动所识别。但是在用户模式下的应用程序无法识别这个设备。让用户模式下的应用程序能识别设备有两种办法，第一种是通过符号链接找到设备，第二种是通过设备接口找到设备。设备接口的办法在 NT 驱动中很少使用，在后面介绍 WDM 驱动的时候再做介绍。

符号链接可以理解为设备对象起了一个“别名”。设备对象的名称只能被内核模式的驱动识别，而别名也可以被用户模式下的应用程序识别。例如，常说的 C 盘、D 盘就是符号链接。所谓的 C 盘，指的是名为“C:”的符号链接，其真正的设备对象是“\Device\

Windows 驱动开发技术详解

HarddiskVolume1”，而“D:”所代表的真正设备对象是“\Device\HarddiskVolume2”。创建符号链接的函数是 IoCreateSymbolicLink，其函数声明如下。

```
NTSTATUS
IoCreateSymbolicLink(
    IN PUNICODE_STRING SymbolicLinkName,
    IN PUNICODE_STRING DeviceName
);
```

- SymbolicLinkName: 输入参数，符号链接的字符串，用 UNICODE 字符串表示。
- DeviceName: 输入参数，设备对象名的字符串，用 UNICODE 字符串表示。
- 返回值: 返回创建符号链接是否成功。

在内核模式下，符号链接是以“\??\”开头的（或者是“\DosDevices\”开头的），如 C 盘就是“\??\C:”（或者“\DosDevices\C:”）。而在用户模式下，则是以“\\.”开头的，如 C 盘就是“\\.\C:”。在 HelloDDK 中，设备对象的符号链接是“\??\HelloDDK”。

在 HelloDDK 中，设备的创建被封装在一个函数中。

```
#001 NTSTATUS CreateDevice {
#002     IN PDRIVER_OBJECT pDriverObject)
#003 {
#004     NTSTATUS status;
#005     PDEVICE_OBJECT pDevObj;
#006     PDEVICE_EXTENSION pDevExt;
#007
#008     //创建设备名称
#009     UNICODE_STRING devName;
#010     RtlInitUnicodeString(&devName, L"\\Device\\MyDDKDevice");
#011
#012     //创建设备
#013     status = IoCreateDevice( pDriverObject,
#014                             sizeof(DEVICE_EXTENSION),
#015                             &(UNICODE_STRING)devName,
#016                             FILE_DEVICE_UNKNOWN,
#017                             0, TRUE,
#018                             &pDevObj );
#019     //判断设备对象是否创建成功
#020     if (!NT_SUCCESS(status))
#021         return status;
#022
#023     //将设备设置为缓冲区设备
#024     pDevObj->Flags |= DO_BUFFERED_IO;
#025     //得到设备扩展
#026     pDevExt = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
#027     //设置设备扩展的设备对象
#028     pDevExt->pDevice = pDevObj;
#029     //设置设备扩展中的设备名称
#030     pDevExt->ustrDeviceName = devName;
#031     //创建符号链接
#032     UNICODE_STRING symLinkName;
#033     RtlInitUnicodeString(&symLinkName, L"\\??\\HelloDDK");
#034     pDevExt->ustrSymLinkName = symLinkName;
#035     //创建符号链接
#036     status = IoCreateSymbolicLink( &symLinkName, &devName );
```

```

#037 //判断是否成功创建符号链接
#038 //if (INT_SUCCESS(status))
#039 {
#040     //删除符号链接
#041     IoDeleteDevice( pDevObj );
#042     return status;
#043 }
#044 return STATUS_SUCCESS;
#045 }

```

此段代码可以在配套光盘中本章的 NT_Driver 目录下找到。

其中在创建设备对象的时候,指定的设备类型为 FILE_DEVICE_UNKNOWN,说明此设备是常用设备之外的设备,一般虚拟设备常使用这个作为设备类型。

在创建设备对象后,对 Flags 的 DO_BUFFERED_IO 位进行设置,这里是将设备设置成“缓冲区设备”。关于将设备设置成“缓冲设备”或者“直接设备”,请参考第7章。随后设定设备的设备扩展,设备扩展是程序员自定义的,在 HelloDDK 中的设备扩展非常简单,如下:

```

typedef struct _DEVICE_EXTENSION {
    PDEVICE_OBJECT pDevice;           //通过 pDevice 指向设备对象
    UNICODE_STRING ustrDeviceName;    //设备名称
    UNICODE_STRING ustrSymLinkName;   //符号链接名
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

```

- pDevice: 设备对象中的 DeviceExtension 指向设备扩展, pDevice 可以指回设备对象。
- ustrDeviceName: 设备名。
- ustrSymLinkName: 设备的符号链接名。

在设备扩展中,记录以上几个信息,以备其他回调函数或者派遣函数使用。使用的时候,只需从驱动设备中获取,类似于以下代码:

```
pDevExt = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
```

4.2.3 DriverUnload 例程

在驱动对象中会设置 DriverUnload 例程,此例程在驱动被卸载的时候调用。在 NT 式驱动中,DriverUnload 一般负责删除在 DriverEntry 中创建的设备对象,并且将设备对象所关联的符号链接删除。另外,DriverUnload 还负责对一些资源进行回收。例如在 HelloDDK 中,就是做如下处置的。

```

#001 VOID HelloDDKUnload (IN PDRIVER_OBJECT pDriverObject)
#002 {
#003     PDEVICE_OBJECT pNextObj;
#004     KdPrint(("Enter DriverUnload\n"));
#005     //得到下一个设备对象
#006     pNextObj = pDriverObject->DeviceObject;
#007     //枚举所有设备对象

```



```
#008     while (pNextObj != NULL)
#009     {
#010         //得到设备扩展
#011         PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
#012             pNextObj->DeviceExtension;
#013
#014         //删除符号链接
#015         UNICODE_STRING pLinkName = pDevExt->ustrSymLinkName;
#016         IoDeleteSymbolicLink(&pLinkName);
#017         pNextObj = pNextObj->NextDevice;
#018         //删除设备
#019         IoDeleteDevice( pDevExt->pDevice );
#020     }
#021 }
```

此段代码可以在配套光盘中本章的 NT_Driver 目录下找到。

在 HelloDDKUnload 中，传进来驱动对象。前面讲过，根据驱动对象，就可以遍历所有由该驱动对象创建的设备对象。通过驱动对象的 DeviceObject 域，可以找到驱动对象的第一个设备对象，然后根据设备对象的 NextDevice 域，就可以找到随后的设备对象。在 HelloDDK 这个例子中，驱动对象其实只创建了一个设备对象。因此，在 HelloDDKUnload 的调用中，也只会删除一个设备对象。

删除设备对象的函数是 IoDeleteDevice，函数原型是：

```
VOID
IoDeleteDevice(
    IN PDEVICE_OBJECT DeviceObject
);
```

其参数就是要被删除的设备对象指针。在 DriverUnload 中，除了要删除设备对象，还要对设备对象关联的符号链接进行删除。删除符号链接的函数是 IoDeleteSymbolicLink，其函数原型是：

```
NTSTATUS
IoDeleteSymbolicLink(
    IN PUNICODE_STRING SymbolicLinkName
);
```

- SymbolicLinkName: 表示已经被注册了的符号链接。
- 返回值: 表示删除符号链接是否成功。

4.2.4 用 WinObj 观察驱动对象和设备对象

有一个非常实用的工具，可以方便地观察驱动对象和设备对象，这就是 WinObj。它由 Windows 内核专家 Mark Russinovich 编写。现在这个工具可以作为平台 SDK 工具包中的一个小工具，也可以单独从微软网站上直接下载。下载网址是：<http://www.microsoft.com/technet/sysinternals/utilities/WinObj.mspx>。

从 WinObj 这个名称可以看出，该工具是用于查看内核对象的。笔者主要介绍用它查

看驱动对象和设备对象，以及符号链接名等。

首先看一下驱动对象的情况。如图 4-3 所示，左边的列表框列出了 Windows 内核的几种对象。选择列表框里的驱动对象，右边会枚举出 Windows 已经安装过的所有驱动对象，其中包括第 1 章中安装的 HelloDDK 驱动和 HelloWDM 驱动。

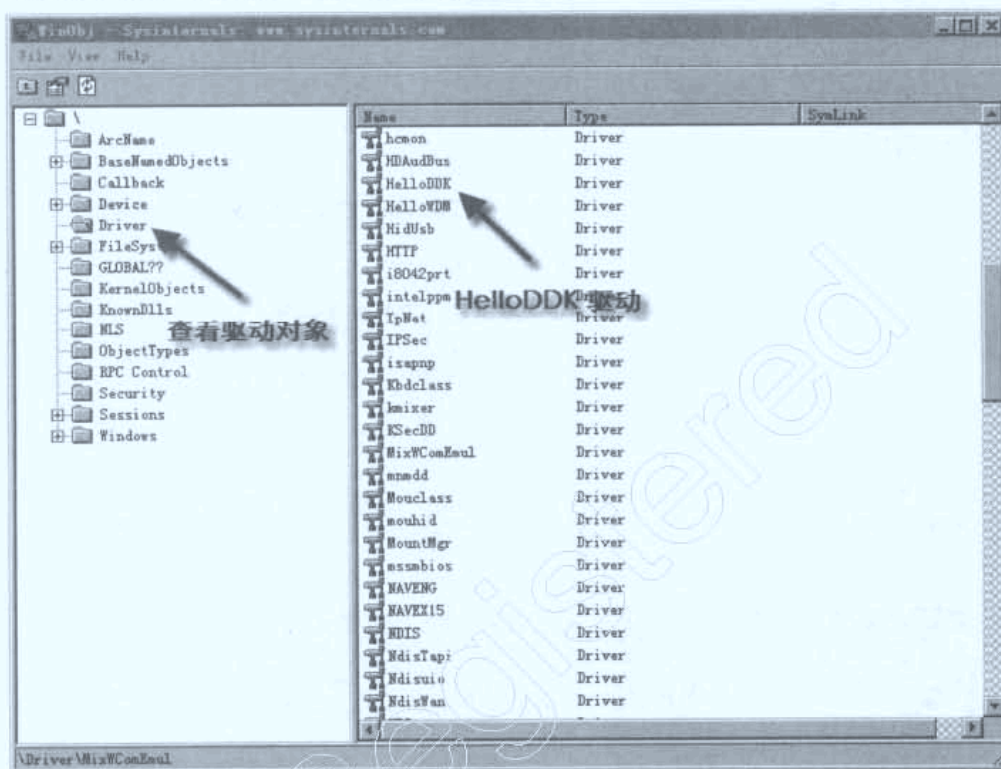


图 4-3 用 WinObj 查看驱动对象

然后，再查看设备对象。如图 4-4 所示，在左侧的列表框中选中 Device，右边会枚举出系统中所有的设备对象。其中有第 1 章中编写的 MyDDKDevice。MyDDKDevice 设备对象是由 HelloDDK 驱动对象所创建的。

前面曾经讲过，在创建设备对象时候，可以不指定设备对象的名称。也就是在使用 IoCreateDevice 函数时，将设备名称设为空字符串。没有设备名称的设备，I/O 管理器会自动建立默认名称。此名称以一个数字命名，且数字号随设备对象的增加而增加，如图 4-4 所示。

最后，用 WinObj 查看符号链接。前面已经介绍过符号链接是提供给用户程序用来设备设备的。符号链接类似于快捷方式，符号链接指向了真正的设备对象。在 HelloDDK 程序中，设备对象“\Device\MyDDKDevice”被链接成“\?\\HelloDDK”，如图 4-5 所示。

Windows 驱动开发技术详解

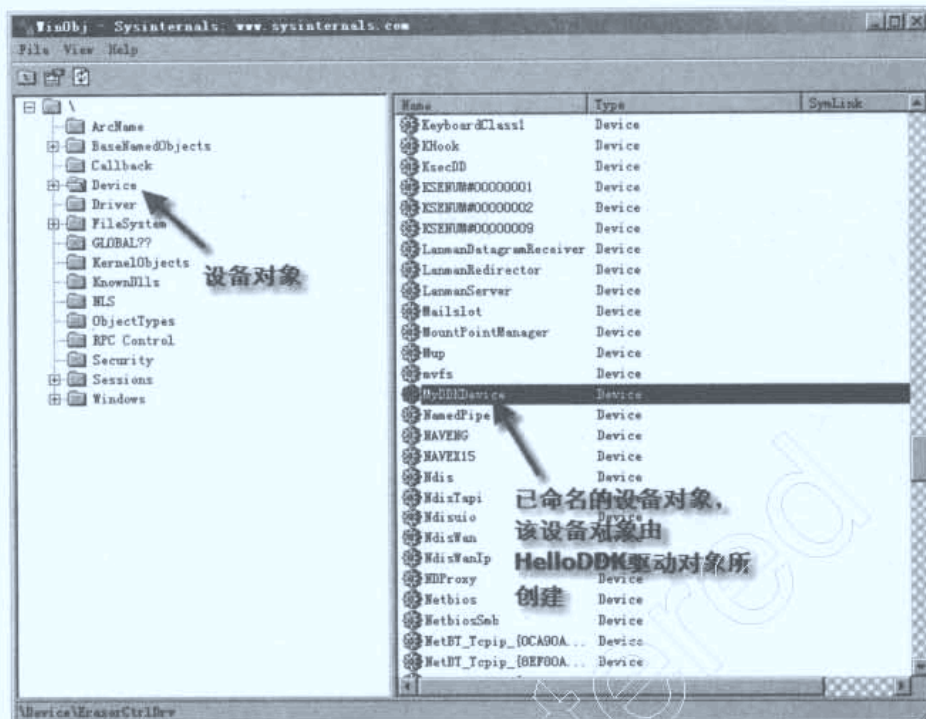


图 4-4 用 WinObj 查看设备对象

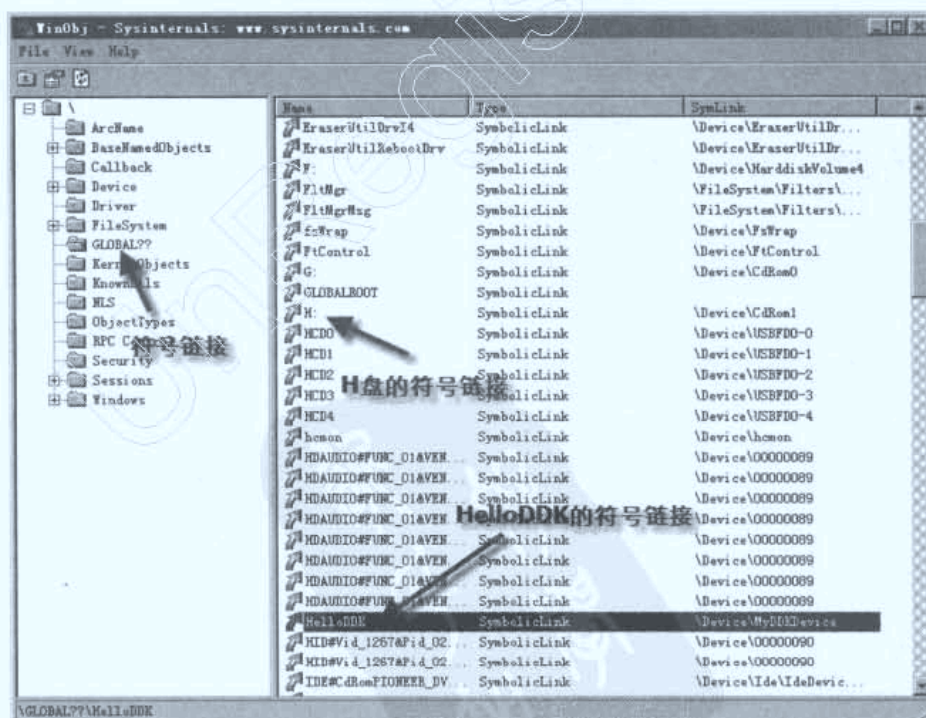


图 4-5 用 WinObj 查看符号链接

另外，还可以查看其他设备暴露的符号链接。例如，在笔者机器下，H 盘符其实是指

向第一个光驱设备，即“\\?\\H:指向\\Device\\CdRom1”，如图 4-5 所示。

4.2.5 用 DeviceTree 观察驱动对象和设备对象

在 DDK 工具中，提供了另外一个强有力的工具——DeviceTree。该工具可以比 WinObj 更加详细列举出驱动对象和设备对象的信息。

使用内核的各种查看工具，有助于读者快速地理解驱动程序中的各个重要概念。同时，也可以帮助读者调试自己的驱动程序。使用 DeviceTree 与 WinObj 有以下几个优点：

- 可以枚举出驱动对象创建的所有设备对象。
- 在分层驱动时，可以列出设备 I/O 堆栈。
- 列出驱动对象、设备对象的地址。
- 列出驱动对象支持的派遣函数。
- 列出驱动对象的 Unload 函数地址。
- 列出设备对象的堆栈大小。
- 列出设备对象的设备类型。
- 列出设备对象的 CurrentIrp 对象指针。
- 列出设备对象的下一个设备对象。
- 显示设备对象的分层结构。

从上述优点可以看出，DeviceTree 几乎将驱动对象和设备对象的相关信息全部显示了出来。首先，用 DeviceTree 查看驱动对象，如图 4-6 所示。



图 4-6 用 DeviceTree 查看驱动对象

其次，用设备对象查看设备对象，如图 4-7 所示。

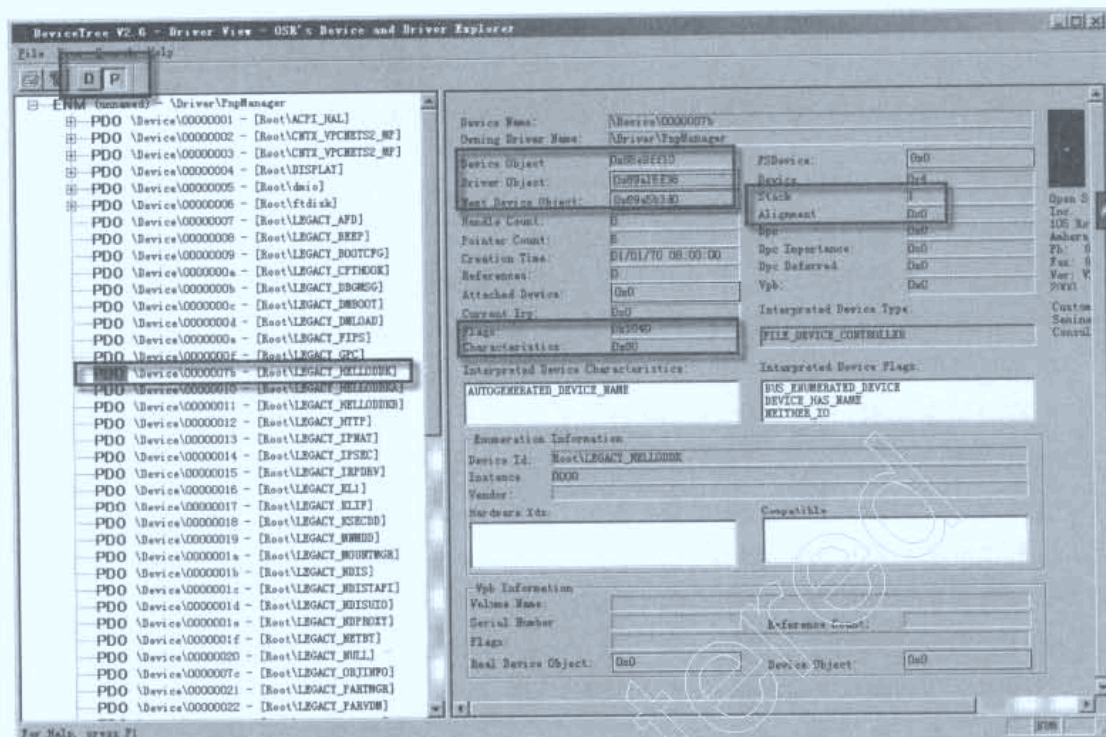


图 4-7 用 DeviceTree 查看设备对象

4.3 WDM 式驱动的基本结构

在 Windows 2000 以后，微软公司加入了新的驱动程序模型，这就是 WDM。WDM 模型是建立在 NT 式驱动程序模型基础之上的。因此，有了前面对 NT 式驱动程序的理解，相信读者也会很快掌握 WDM 的基本结构。

对于 WDM 驱动程序来说，一般都是基于分层的。也就是说，完成一个设备的操作，至少要由两个驱动设备共同完成。

4.3.1 物理设备对象与功能设备对象

在 WDM 模型中，完成一个设备的操作，至少有两个设备对象共同完成。其中，一个是物理设备对象 (Physical Device Object, 以下简称 PDO)，另一个是功能设备对象 (Function Device Object, 以下简称 FDO)。其关系是“附加”与“被附加”的关系。

当 PC 插入某个设备的时，PDO 会自动创建。确切地说，是由总线驱动创建的。PDO 不能单独操作设备，需要配合 FDO 一起使用。系统会提示检测到新设备，如图 4-8 所示，要求安装驱动程序。需要安装的驱动程序指的就是 WDM 程序，此驱动程序负责创建 FDO，并且附加到 PDO 之上。

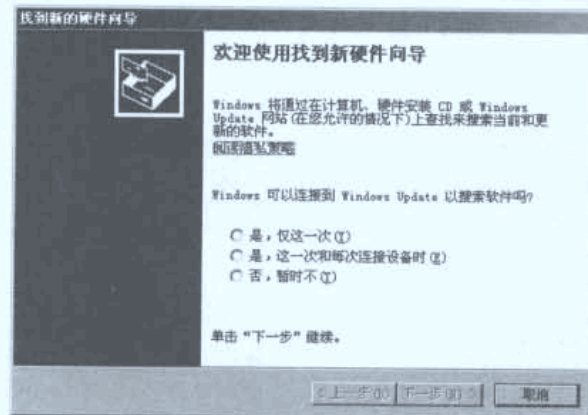


图 4-8 提示添加设备驱动

当一个 FDO 附加在 PDO 上的时候, PDO 设备对象的子域 AttachedDevice 会记录 FDO 的位置。PDO 被称作底层驱动或者下层驱动, 而 FDO 被称作高层驱动或者上层驱动。这里的“上”层指的是接近发出 I/O 请求的地方, 而“下”层指的是靠近物理设备的地方。

PDO 和 FDO 的关系可以从图 4-9 中得到更好地理解。

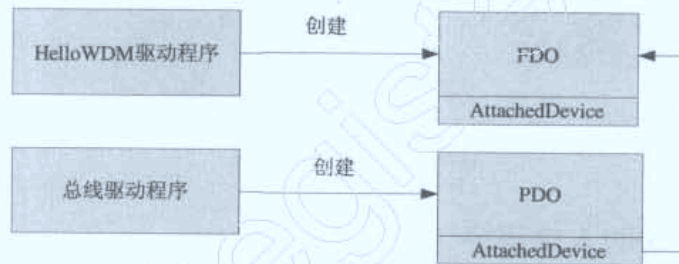


图 4-9 FDO 与 PDO

这是最简单的一种情况, 事实要比这个复杂一些。在 FDO 和 PDO 之间还会存在过滤驱动, 如图 4-10 所示。在 FDO 上面的过滤驱动被称作上层过滤驱动。在 FDO 的下层的驱动, 被称作下层过滤驱动。另外, 每个设备对象中, 有个 StackSize 子域, 表明操作这个设备对象需要几层才能到达最下层的物理设备。如图 4-10 所示, 最上层过滤设备对象的 StackSize 为 4, 也就是需要 4 个设备对象才能到达最底层的物理设备。

过滤驱动可以嵌套, 也就是说可以有很多个高层过滤驱动, 也可以有很多个底层过滤驱动。过滤驱动不是必须存在的。在 WDM 模型中 PDO 和 FDO 是必须的。

可以看出, NT 式驱动和 WDM 驱动在设计思路有所不同。NT 设备是被动装入的, 例如, 当有设备插入 PC 后, 系统不会有提示, 用户需要自己指定加载何种驱动。而 WDM 驱动则不是, 例如, 插入设备后, 系统会自动创建出 PDO, 并提示请求用户安装 FDO。

WDM 提示用户加载 FDO, 如果该设备已经由微软提供, 则会自动进行安装。例如, 当 USB 鼠标插入 PC 后, 系统会默认找到相应的驱动并加载。

这种设计思路, 导致了 WDM 模型支持即插即用功能。当然, 为了支持即插即用功能,

这些还远远不够。总线驱动创建的 PDO 为程序员提供了很多即插即用的服务，这在以后的章节会陆续介绍。

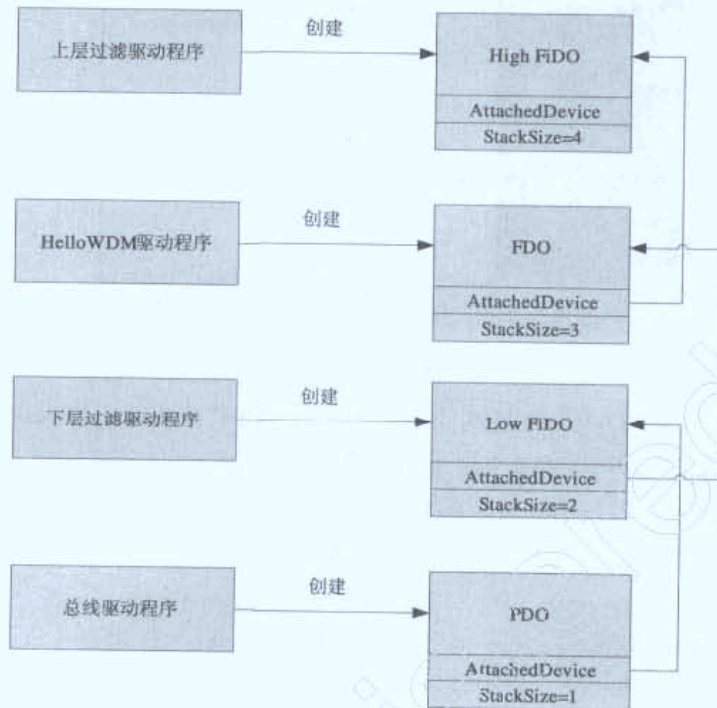


图 4-10 设备对象堆栈

4.3.2 WDM 驱动的入口程序

和 NT 驱动一样，WDM 驱动的入口程序也是 DriverEntry，但是初始化作用被分散到其他例程中。例如，创建设备对象的责任就不在 DriverEntry 中，而被放在了 AddDevice 例程中。同时，在 DriverEntry 中，需要设置对 IRP_MJ_PNP 处理的派遣函数。下面是第 1 章中的 HelloWDM 的 DriverEntry 函数。

```
#001 extern "C" NTSTATUS DriverEntry(IN PDRIVER_OBJECT pDriverObject,
#002                                  IN PUNICODE_STRING pRegistryPath)
#003 {
#004     KdPrint(("Enter DriverEntry\n"));
#005
#006     //设置 AddDevice 函数
#007     pDriverObject->DriverExtension->AddDevice = HelloWDMAddDevice;
#008     //设置各个 IRP 的派遣函数
#009     pDriverObject->MajorFunction[IRP_MJ_PNP] = HelloWDMPnp;
#010     pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
#011     pDriverObject->MajorFunction[IRP_MJ_CREATE] =
#012     pDriverObject->MajorFunction[IRP_MJ_READ] =
#013     pDriverObject->MajorFunction[IRP_MJ_WRITE] = HelloWDMDispatchRoutine;
#014     //设置卸载例程
#015     pDriverObject->DriverUnload = HelloWDMUnload;
#016 }
```

```
#017     KdPrint({"Leave DriverEntry\n"});
#018     return STATUS_SUCCESS;
#019 }
```

此段代码可以在配套光盘中本章的 WDM_Driver 目录下找到。

从上述代码可以看出，WDM 驱动的 DriverEntry 和 NT 式驱动的 DriverEntry 有以下几点不同：

- 增加了对 AddDevice 函数的设置。这是 WDM 驱动和 NT 驱动非常重要的不同点。因为 NT 驱动是主动加载设备的，也就是驱动一旦加载就创建设备。而 WDM 驱动是被动加载设备的，操作系统必须加载 PDO 以后，调用驱动的 AddDevice 例程，AddDevice 例程中负责创建 FDO，并且附加到 PDO 之上。
- 创建设备对象已经不在这个函数中了，而在 AddDevice 例程中创建。
- 必须加入 IRP_MJ_PNP 的派遣回调函数。IRP_MJ_PNP 主要是负责计算机中即插即用的处理，在 WDM 驱动中加入了很多即插即用的处理。

4.3.3 WDM 驱动的 AddDevice 例程

AddDevice 例程是 WDM 驱动所独有的，在 NT 驱动中没有该例程。在 DriverEntry 中，需要设置 AddDevice 例程的函数地址。设置的方式是驱动对象中有个 DriverExtension 子域，DriverExtension 中有个 AddDevice 子域，将该子域指向 AddDevice 例程的函数地址。

```
pDriverObject->DriverExtension->AddDevice = HelloWDMAddDevice;
```

和 DriverEntry 不同，AddDevice 例程的名字可以任意命名，程序员可以用更有意义的名字作为这个函数的名字。在 HelloWDM 例子中，使用的名称就是 HelloWDMAddDevice。

```
#001 NTSTATUS HelloWDMAddDevice(IN PDRIVER_OBJECT DriverObject,
#002                             IN PDEVICE_OBJECT PhysicalDeviceObject)
#003 {
#004     PAGED_CODE();
#005     KdPrint({"Enter HelloWDMAddDevice\n"});
#006
#007     NTSTATUS status;
#008     PDEVICE_OBJECT fdo;
#009     UNICODE_STRING devName;
#010     //初始化 UNICODE 字符串
#011     RtlInitUnicodeString(&devName,L"\\Device\\MyWDMDevice");
#012     //创建设备对象
#013     status = IoCreateDevice(
#014         DriverObject,
#015         sizeof(DEVICE_EXTENSION),
#016         &(UNICODE_STRING)devName,
#017         FILE_DEVICE_UNKNOWN,
#018         0,
#019         FALSE,
#020         &fdo);
#021     //判断是否成功创建设备对象
#022     if( !NT_SUCCESS(status))
#023         return status;
```



```

#024    //得到设备扩展
#025    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION)fdo->DeviceExtension;
#026    pdx->fdo = fdo;
#027    //将 FDO 附加在 PDO 上
#028    pdx->NextStackDevice = IoAttachDeviceToDeviceStack(fdo, PhysicalDevice
Object);
#029    UNICODE_STRING symLinkName;
#030    RtlInitUnicodeString(&symLinkName,L"\\DosDevices\\HelloWDM");
#031
#032    //用设备扩展记录设备名和符号链接
#033    pdx->ustrDeviceName = devName;
#034    pdx->ustrSymLinkName = symLinkName;
#035    //创建符号链接
#036    status = IoCreateSymbolicLink(&(UNICODE_STRING)symLinkName,&(UNICODE_
STRING)devName);
#037
#038    //如果创建符号链接失败,则删除设备对象
#039    if( !NT_SUCCESS(status))
#040    {
#041        IoDeleteSymbolicLink(&pdx->ustrSymLinkName);
#042        status = IoCreateSymbolicLink(&symLinkName,&devName);
#043        if( !NT_SUCCESS(status))
#044        {
#045            return status;
#046        }
#047    }
#048    //设置设备标志
#049    fdo->Flags |= DO_BUFFERED_IO | DO_POWER_PAGABLE;
#050    fdo->Flags &= ~DO_DEVICE_INITIALIZING;
#051
#052    KdPrint(("Leave HelloWDMAddDevice\n"));
#053    return STATUS_SUCCESS;
#054 }

```

这段代码可以在配套光盘中本章的 WDM_Driver 目录下找到。

从上述代码中可以看出, AddDevice 例程类似于 NT 驱动中 DriverEntry 创建设备对象的相关操作, 但是略有不同。AddDevice 函数有两个输入参数, 一个是驱动对象 DriverObject, 另一个是设备对象 PhysicalDeviceObject。驱动对象是 I/O 管理器创建的驱动对象。设备对象 PhysicalDeviceObject 就是底层总线驱动创建的 PDO 设备对象。传进该参数的目的就是 FDO 附加在 PDO 之上。

在 AddDevice 可以分为以下几个步骤:

① 在 AddDevice 通过 IoCreateDevice 等函数, 创建了设备对象, 该设备对象就是 FDO, 即功能驱动设备对象。和 NT 驱动一样, 可以设置驱动对象的设备名称, 也可以不设置。如果不设置设备名称, I/O 管理器会自动以一个数字作为该设备对象的名称。

② 创建完 FDO 后, 需要将 FDO 的地址保存下来, 以便以后使用。保存的位置是在设备扩展中, 前面介绍过, 在驱动程序中应该尽量避免使用全局变量, 而使用设备扩展。如果该电脑中存在多个同类设备, 例如, 插入两个型号相同的网卡, 操作系统会两次调用 AddDevice 例程。每个 AddDevice 例程创建各自的 FDO, 分别记录在各自的设备扩展中。

③ 驱动程序将创建的 FDO 附加在 PDO 上, 附加这个动作是依靠 IoAttachDeviceToDeviceStack 函数实现的。IoAttachDeviceToDeviceStack 的声明如下:

```
PDEVICE_OBJECT
IoAttachDeviceToDeviceStack(
    IN PDEVICE_OBJECT SourceDevice,
    IN PDEVICE_OBJECT TargetDevice
);
```

- SourceDevice: 要附加在别的设备之上的设备。将 FDO 附加在 PDO 之上时, 这个填写的是 FDO 的地址。
- TargetDevice: 被附加的设备。将 FDO 附加在 PDO 之上时, 这个填写的是 PDO 的地址。当 FDO 想附加在 PDO 上时, 有时会在 PDO 和 FDO 上附加过滤驱动。此时, FDO 其实是附加在过滤设备上, 而过滤设备附加在 PDO 上。
- 返回值: 附加以后, 返回附加设备的下层设备。如果中间没有过滤驱动的话, 返回值就是 PDO, 如果中间有过滤驱动, 返回的是过滤驱动。

前面介绍过, 当 FDO 附加到 PDO 上时, PDO 会通过 AttachedDevice 子域知道它上面的设备是 FDO (或者是过滤驱动)。但是 FDO 却不知道自己的下层是什么设备。解决的办法是, 通过设备扩展记录 FDO 下层的设备。下面是 HelloWDM 的设备扩展的定义。

```
typedef struct _DEVICE_EXTENSION
{
    PDEVICE_OBJECT fdo; //FDO
    PDEVICE_OBJECT NextStackDevice; //下层驱动设备
    UNICODE_STRING ustrDeviceName; // 设备名
    UNICODE_STRING ustrSymLinkName; // 符号链接
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

读者在自己编写驱动程序时, 可以根据自己的需要定制自己的设备扩展。子域 fdo 是为了保存 FDO 的地址, 以备后用。子域 NextStackDevice 是为了定位设备的下一层设备。

在附加操作完成后, 需要设定符号链接, 以便用户应用程序可以访问该设备。

④ 设置 fdo 的 Flags 子域。DO_BUFFERED_IO 是定义设备为“缓冲内存设备”。另外~DO_DEVICE_INITIALIZING, 是将 Flag 上的 DO_DEVICE_INITIALIZING 位清零。保证设备初始化完毕, 这一步是必需的。

4.3.4 DriverUnload 例程

在 NT 式驱动中, DriverUnload 例程主要负责做删除设备和取消符号链接。而在 WDM 驱动中, 这部分操作被 IRP_MN_REMOVE_DEVICE IRP 的处理函数所负责, 而 DriverUnload 例程显得变得相对简单。如果在 DriverEntry 中有申请内存的操作, 可以在 DriverUnload 例程中回收这些内存。在 HelloWDM 程序中, DriverUnload 例程除了打印两行 log 信息, 什么也没做。

4.3.5 对 IRP_MN_REMOVE_DEVICE IRP 的处理

关于 IRP 的介绍，后面第 7、8 章有详细的介绍，这里只是略微提一下。驱动程序内部是由 IRP 驱动的。创建 IRP 的原因有很多，IRP_MN_REMOVE_DEVICE 这个 IRP 是当设备需要被卸载的时候，由即插即用管理器创建，并发送到驱动程序中的。IRP 一般由两个号码指定该 IRP 的具体意义，一个是主 IRP 号 (Major IRP)，另一个是辅 IRP 号 (Minor IRP)。

每个 IRP 都由对应的派遣函数所处理，派遣函数是在 DriverEntry 中指定的。例如：

```
pDriverObject->MajorFunction[IRP_MJ_PNP] = HelloWDMPnp;
```

上面这句话就是将 IRP_MJ_PNP 的派遣函数指定为 HelloWDMPnp 函数。

当设备需要被卸载的时候，会先后发出多个 IRP_MJ_PNP。这些 IRP 的辅 IRP 号会有所不同。其中之一是 IRP_MN_REMOVE_DEVICE。

在 WDM 驱动程序中，对设备的卸载一般是在对 IRP_MN_REMOVE_DEVICE 的处理函数中进行卸载。在 HelloWDM 驱动程序中，负责处理 IRP_MN_REMOVE_DEVICE 的函数是 HandleRemoveDevice，其代码如下：

```
#001 NTSTATUS HandleRemoveDevice(PDEVICE_EXTENSION pdx, PIRP Irp)
#002 {
#003     PAGED_CODE();
#004     KdPrint(("Enter HandleRemoveDevice\n"));
#005     //设置 IRP 的完成状态
#006     Irp->IoStatus.Status = STATUS_SUCCESS;
#007     //将 IRP 请求向底层驱动转发
#008     NTSTATUS status = DefaultPnpHandler(pdx, Irp);
#009     //删除符号链接
#010     IoDeleteSymbolicLink(&(UNICODE_STRING)pdx->ustrSymLinkName);
#011
#012     //调用 IoDetachDevice()把 fdo 从设备栈中脱开
#013     if (pdx->NextStackDevice)
#014         //从设备栈中脱离
#015         IoDetachDevice(pdx->NextStackDevice);
#016
#017     //删除 fdo
#018     IoDeleteDevice(pdx->fdo);
#019     KdPrint(("Leave HandleRemoveDevice\n"));
#020     return status;
#021 }
```

此段代码可以在配套光盘中本章的 WDM_Driver 目录下找到。

在处理 IRP_MN_REMOVE_DEVICE 的函数中，它的功能类似于 NT 驱动中的 DriverUnload 函数。除了需要删除设备、取消符号链接外，在此函数中还需要将 FDO 从 PDO 上的堆栈中“摘除”下来。使用的函数是 IoDetachDevice，其函数原型如下：

```
VOID
IoDetachDevice(
    IN OUT PDEVICE_OBJECT TargetDevice
);
```

其中，TargetDevice 参数就是下层堆栈上的设备对象。在 HelloWDM 程序中，下层设备对象是记录在设备扩展中的，即：

```
IoDetachDevice(pdx->NextStackDevice);
```

此时，FDO 从设备链上被删除，但 PDO 还是存在。PDO 的删除不是由程序员负责，而是由操作系统负责。

4.3.6 用 Device Tree 查看 WDM 设备对象栈

FDO 附加在 PDO 上，同时可能还会有其他过滤设备也附加在其上。这样一层层的附加操作，从底层设备到高层设备形成了一个设备链，有时也被形象的称为设备对象堆栈。

Device Tree 是查看设备堆栈的优秀工具，它可以帮助程序员清楚地观察设备栈的形成过程。通过使用 Device Tree 工具，可以更好地理解 WDM 驱动程序的结构。首先，用 Device Tree 观察驱动对象和设备对象，这个和 NT 驱动基本上是类似的。

在图 4-11 中,不能反应设备堆栈的情况。因此,只能找出 PDO 创建的驱动对象,即总线驱动程序创建的驱动对象。在系统中,有很多种总线,不同设备的 PDO 是由不同总线驱动创建的。由于 HelloWDM 的驱动是虚拟设备,它的 PDO 是由“\Driver\PnpManager”所创建的,该驱动也被称为即插即用管理器。

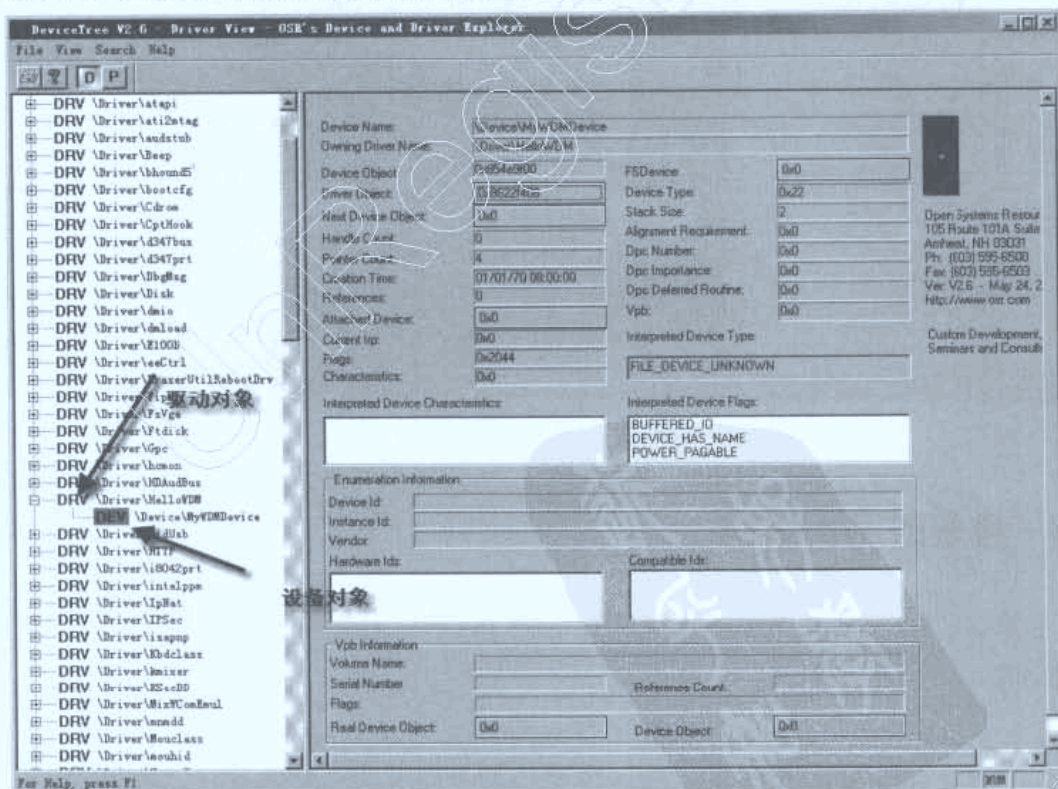


图 4-11 用 DeviceTree 查看 WDM 驱动对象

“\Driver\PnpManager”创建了多个设备对象，如图 4-12 所示，这些设备大部分是 PDO。由于属于同一个驱动对象，因此这些设备对象的子域 NextDevice 相互链接起来，形成一串设备链。笔者将这种关系称之为水平链接，以便和垂直关系（如 FDO 附加在 PDO 之上）相区分。

如图 4-12 所示，设备“\Device\MyWDMDevice”（也就是 FDO）“附加”到设备“\Device\00000041”（也就是 PDO）之上。其中图标“ATT”是“Attach”的缩写，这个列表可以很好地理解 WDM 模型。注意：因为 PDO 是未命名的设备对象，因此是以数字命名的设备名称。笔者里的数字和读者的数字会有所不同。

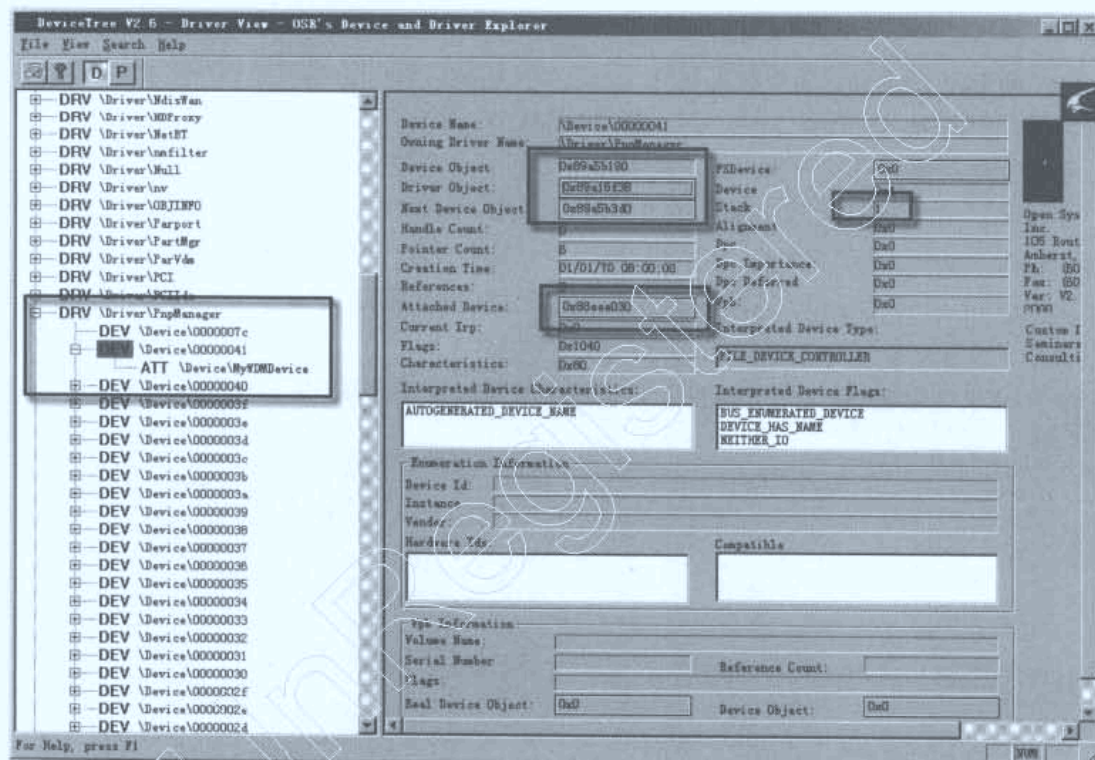


图 4-12 查看设备附加情况

4.4 设备的层次结构

在第 2 章中，曾介绍 Windows 操作系统是分层调用。其实在驱动程序中，也可以是分层调用的。

4.4.1 驱动程序的垂直层次结构

不仅是 WDM 驱动，NT 式驱动也可以分层，这主要是通过一个设备附加在另一个设备之上。因此，可以将 WDM 驱动模型看成是 NT 驱动模型的延伸。

设备的创建顺序是，先创建底层 PDO，再创建高层的 FDO，这也就是设备堆栈的生长方向，即从底层设备到高层设备。在 PDO 和 FDO 之间可能夹杂着各种过滤驱动。每层的设备对象由不同的驱动程序所创建，或者说每层的设备对应着不同的驱动程序。有的驱动程序是系统自带的，有的是需要程序员编写。底层设备对象寻找上一层的设备对象，是依靠底层设备对象的 AttachedDevice 来寻找的，如果某一设备的 AttachedDevice 为空，说明已经到了设备堆栈的顶部。

而高层设备寻找低一层的设备对象，设备对象没有相关子域可以使用。解决的办法是，通过程序员自定义设备扩展，在设备扩展记录低一层的设备对象。这样，从最底层的设备对象到达设备顶部，再从设备顶部到达设备堆栈底部，都有了相应的办法。

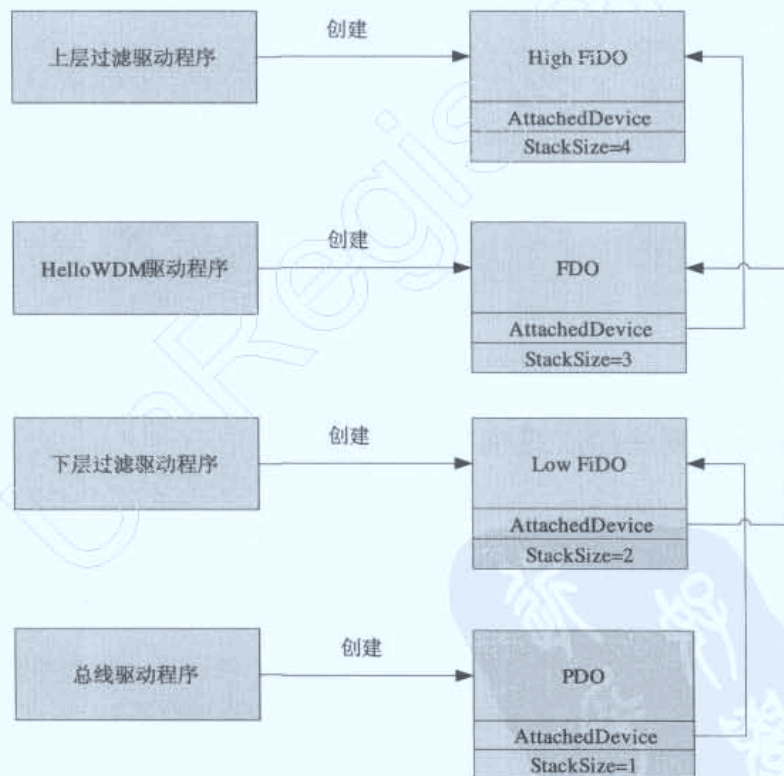


图 4-13 设备的垂直结构

4.4.2 驱动程序的水平层次结构

为了区分设备堆栈的垂直结构，笔者将同一驱动程序创建出来的设备对象的关系称之为水平层次关系。水平层次的第一个设备对象，由它的驱动对象所指定。每一个设备通过子域 NextDevice 可以找到水平层次的下一个设备对象。

例如，电脑中插入两块型号相同的网卡。当插入第一个网卡的时候，PCI 总线会检测到有 PCI 设备被插入到电脑中，创建 PCI 的物理设备对象（PDO），随后加载相应的 FDO。插入第二块网卡会有同样的过程，产生另外的 PDO 和 FDO。这样两个 PDO 之间就是同一水平层次的，而两个 FDO 也处于同一个水平层次上。

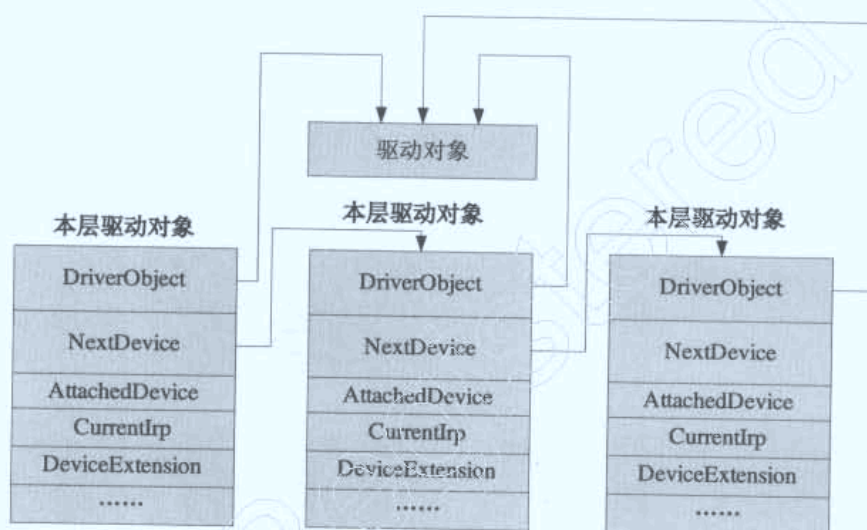


图 4-14 设备的水平结构

4.4.3 驱动程序的复杂层次结构

由设备对象的水平结构和垂直结构，组成了 Windows 设备的树形结构图。在 Windows 中初始的时候会有一个根设备，为了理解简单，我们将 PCI 总线想象成根总线（根总线其实不是 PCI 总线，只是为了理解方便）。插到 PCI 总线上的设备，PCI 总线会枚举每一个插在 PCI 总线上的设备，并为每个设备创建 PDO，然后每个 PDO 的上面必须有一个 FDO。比如网卡这样的设备，通过 FDO 和 PDO，就可以操作这个物理设备了。

PCI 还能枚举出诸如 PCI-ISA 的桥接设备，为它创建 PDO，并在上面加载 FDO。此时，桥接设备的 FDO 起到了类似根总线驱动的职能，遍历 ISA 总线上的所有设备。如果有 ISA 设备插入，则为之创建一个 PDO，并加载相应的 FDO。

USB 设备的枚举也十分类似。首先在 PCI 总线上，会枚举到 USB 控制器设备，并加

载 PDO 和 FDO。这个 FDO 会枚举在这个控制器上的 USB HUB 设备，为之创建 PDO，并加载相应的 FDO。该 FDO 枚举所有插在 USB HUB 上的 USB 设备，为之创建 PDO，并加载相应的 FDO，如图 4-15 所示。

为了更好地理解各个 PDO、FDO 的相互创建过程，图 4-16 列出了在笔者电脑中 USB 鼠标的 FDO。从图 4-16 中可以看出，为了创建 USB 鼠标的 PDO 和 FDO，中间经过了多个总线创建 PDO、加载相应 FDO 的过程。

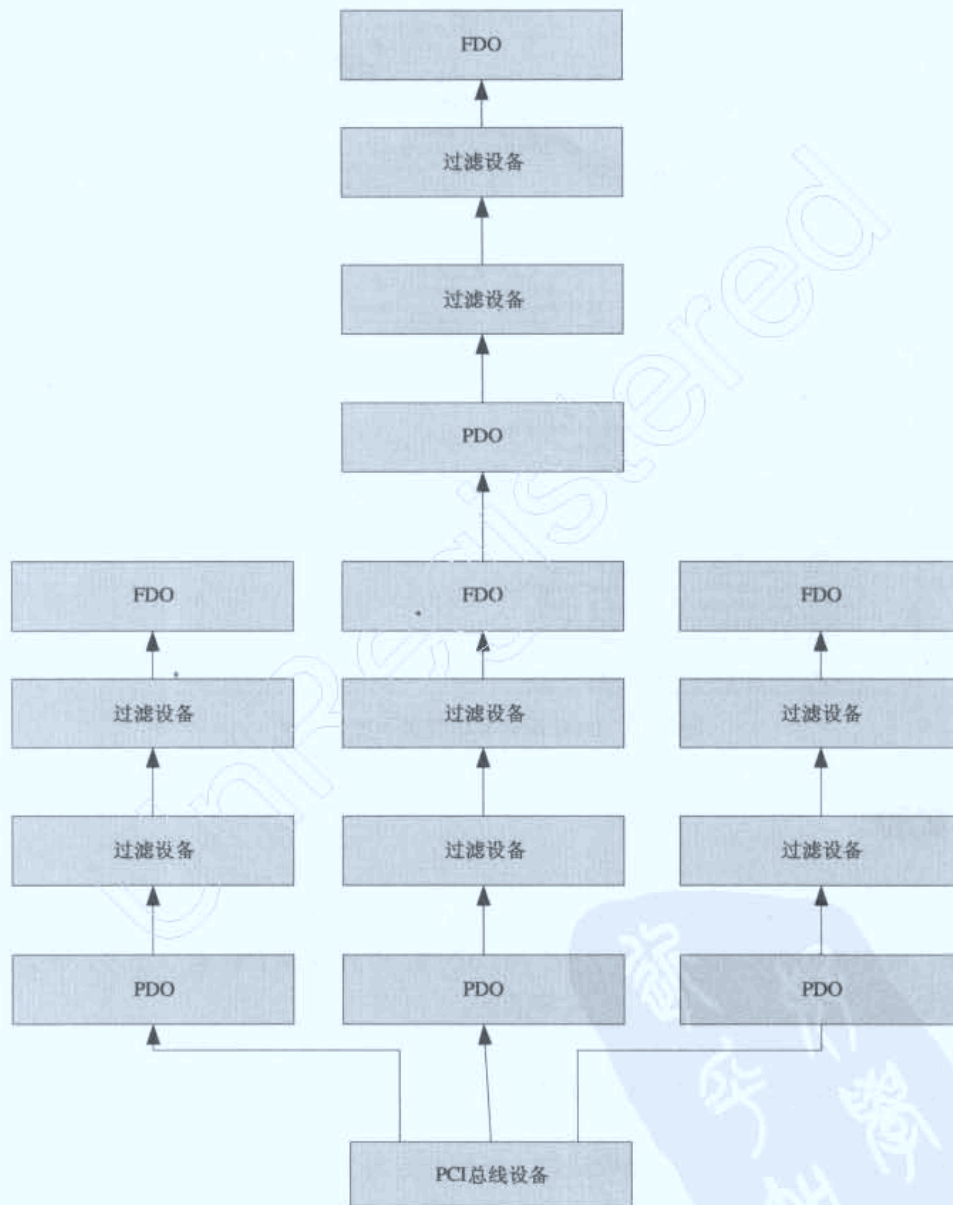


图 4-15 设备的复杂结构

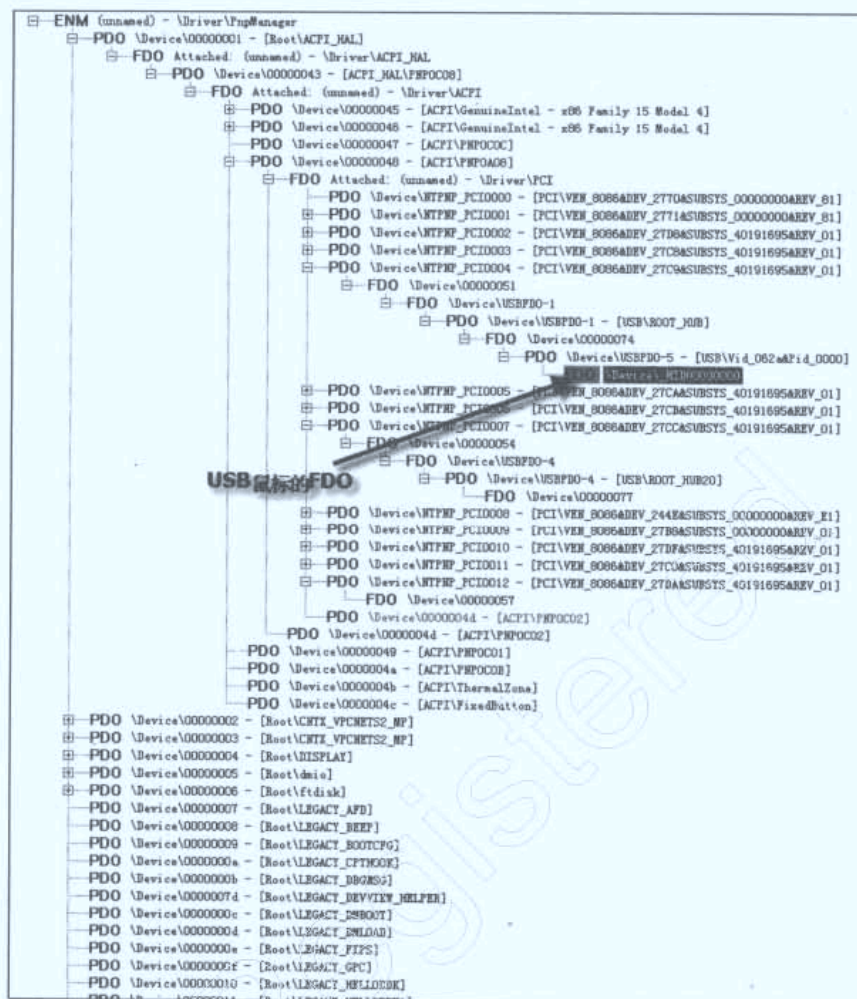


图 4-16 USB 鼠标驱动设备的加载过程

4.5 实验

本章详细讲解了 NT 式驱动和 WDM 式驱动的基本结构。在本章结束之前，为了复习本章所学，请读者做两个实验，试着改写 HelloDDK 和 HelloWDM 驱动程序，使其能详细打印出驱动对象、设备对象、设备堆栈等信息。

4.5.1 改写 HelloDDK 查看驱动结构

HelloDDK 驱动程序的结构比较简单。在垂直方向上，它没有挂载任何别的设备，在水平层次上也只有一个设备对象。为了增加程序的复杂性，改写的代码中，在 DriverEntry 中创建了两个设备对象，其关系是水平方向上的关系。用 CreateDevice 创建第一个设备对象，用 CreateDevice2 创建第二个设备对象。CreateDevice2 和 CreateDevice 的实现基本相同。

```

#001 //创建设备对象
#002 status = CreateDevice(pDriverObject);
#003 status = CreateDevice2(pDriverObject);

```

为了查看驱动对象的相关信息和水平方向的相关信息,编写了 Dump 函数,并在 DriverEntry 中调用,代码如下:

```

#001 void Dump(IN PDRIVER_OBJECT pDriverObject)
#002 {
#003     KdPrint(("-----\n"));
#004     KdPrint(("Begin Dump...\n"));
#005     //打印一些调试信息
#006     KdPrint(("Driver Address:0X%08X\n",pDriverObject));
#007     KdPrint(("Driver name:%S\n",pDriverObject->DriverName.Buffer));
#008     KdPrint(("Driver HardwareDatabase:%S\n",pDriverObject->Hardware
Database->Buffer));
#009     KdPrint(("Driver first device:0X%08X\n",pDriverObject->DeviceObject));
#010     //得到设备对象
#011     PDEVICE_OBJECT pDevice = pDriverObject->DeviceObject;
#012     int i=1;
#013     //枚举每一个设备
#014     for (;pDevice!=NULL;pDevice = pDevice->NextDevice)
#015     {
#016         KdPrint(("The %d device\n",i++));
#017         KdPrint(("Device AttachedDevice:0X%08X\n",pDevice->AttachedDevice));
#018         KdPrint(("Device NextDevice:0X%08X\n",pDevice->NextDevice));
#019         KdPrint(("Device StackSize:%d\n",pDevice->StackSize));
#020         KdPrint(("Device's DriverObject:0X%08X\n",pDevice->DriverObject));
#021     }
#022     KdPrint(("Dump over!\n"));
#023     KdPrint(("-----\n"));
#024 }

```

此段代码可以在配套光盘中本章的 NT_Driver 目录下找到。

经过改写后的代码,会创建出两个设备对象。改写后的驱动在被装入后,会打印出 log 信息,如图 4-17 所示。

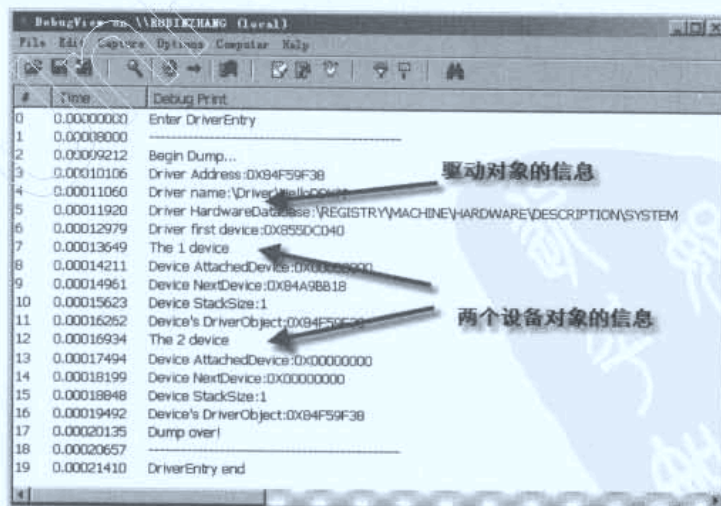


图 4-17 修改后的 HelloDDK 的 log

4.5.2 改写 HelloWDM 查看驱动结构

在 HelloWDM 中,可以观察 FDO 和 PDO 设备之间的关系,它们是处于垂直之间的关系。由于在 AddDevice 例程的后,FDO 创建完成,并且附加在 PDO 上。本例将罗列 PDO 和 FDO 的代码,将其添加在 AddDevice 的后面。

罗列 PDO 和 FDO 的代码被封装在函数 DumpDeviceStack 中。另外,在前面改写的 HelloDDK 中,查看水平方向的代码亦可使用。DumpDeviceStack 的代码如下:

```
#001 void DumpDeviceStack(IN PDEVICE_OBJECT pdo)
#002 {
#003     KdPrint(("-----\n"));
#004
#005     KdPrint(("Begin Dump device stack...\n"));
#006
#007     PDEVICE_OBJECT pDevice = pdo;
#008     int i=0;
#009     //枚举每一个附加的设备对象
#010     for (;pDevice!=NULL;pDevice=pDevice->AttachedDevice)
#011     {
#012         //打印一些调试信息
#013         KdPrint(("The %d device in device stack\n",i++));
#014         KdPrint(("Device AttachedDevice:0x%08X\n",pDevice->AttachedDevice));
#015         KdPrint(("Device NextDevice:0x%08X\n",pDevice->NextDevice));
#016         KdPrint(("Device StackSize:%d\n",pDevice->StackSize));
#017         KdPrint(("Device's DriverObject:0x%08X\n",pDevice->DriverObject));
#018     }
#019     KdPrint(("Dump over!\n"));
#020     KdPrint(("-----\n"));
#021 }
```

此段代码可以在配套光盘中本章的 WDM_Driver 目录下找到。

修改后的 HelloWDM 重新加载后,打印出来的 log 信息如图 4-18 所示。

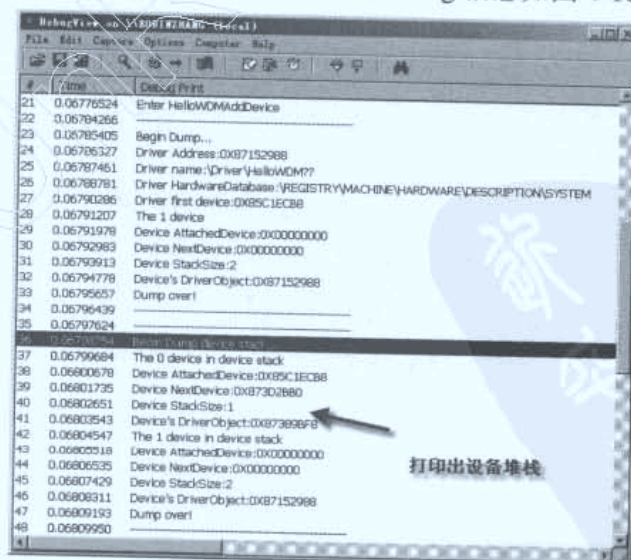


图 4-18 修改后的 HelloWDM 的 log

4.6 小结

本章重点介绍了 NT 式驱动和 WDM 驱动的结构，从中可以看出 WDM 驱动程序是在 NT 驱动程序的基础上变化过来的，可以将 WDM 驱动程序看成是 NT 驱动程序的特例。

同时，本章还介绍了驱动程序的入口函数、卸载函数、IRP 的派遣函数等。另外，本章还介绍了驱动对象、设备对象等主要数据结构。其中，对设备对象之间可以形成垂直结构和水平结构的框架。

UnRegistered



第5章 Windows 内存管理

作为开发 Windows 驱动程序的程序员，需要比普通程序员更多地了解 Windows 内部的内存管理机制，并在驱动程序中有效地使用内存。在驱动程序编写中，分配和管理内存不能使用熟知的 Win32 API 函数，取而代之的是 DDK 提供的高效的内核函数。程序员必须小心地使用这些内存相关的内核函数。因为在内核模式下，操作系统不会检查内存使用的合法性，稍有不慎就可能导致操作系统的崩溃。

另外，C 语言和 C++ 中大多数关于内存操作的运行时函数，大多在内核模式下是无法使用的。例如，C 语言中的 malloc 函数、C++ 中的 new 操作符等。本章将详细介绍 Windows 内存管理的有关技术，并且给出驱动程序中操作内存的一些实例。

5.1 内存管理概念

编写 Windows 驱动之前，需要读者进一步理解 Windows 操作系统是如何管理和使用内存的。在此只讨论 32 位平台下 Windows 操作系统的相关知识。本节先解释物理内存地址、虚拟内存地址、进程和驱动的关系，然后在后面给出操作内存的示例。

5.1.1 物理内存概念 (Physical Memory Address)

PC 上有三条总线，分别是数据总线、地址总线和控制总线。32 位的 CPU 的寻址能力为 4GB (2^{32}) 个字节。用户最多可以使用 4GB 的真实的物理内存。PC 中会拥有很多设备，其中很多设备都提供了自己的设备内存。例如，显卡就会提供自己的显存。这部分内存会映射到 PC 的物理内存上，也就是读写这段物理地址，其实会读写的设备内存地址，而不会读写物理内存地址。很多情况下，会广义地认为这也是物理内存地址。图 5-1 就是笔者 PC 中显卡的显存地址，一个设备可以有好几块设备内存映射到物理内存上。

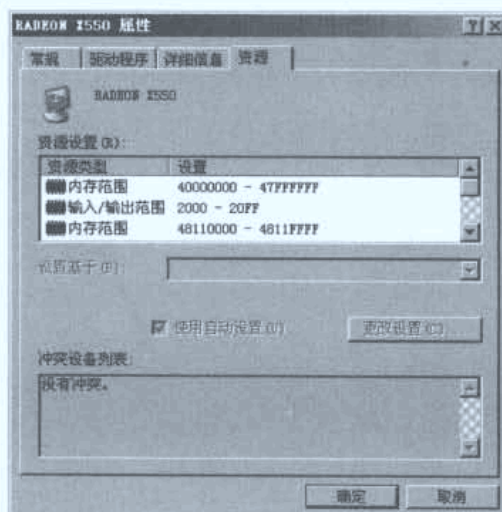


图 5-1 设备上的设备内存

5.1.2 虚拟内存地址概念 (Virtual Memory Address)

虽然可以寻址 4GB 的内存，而在 PC 里往往没有如此多的真实物理内存。操作系统和硬件（这里指的是 CPU 中的内存管理单元 MMU）为使用者提供了虚拟内存的概念。Windows 的所有程序（包括 Ring0 层和 Ring3 层的程序）可以操作的都是虚拟内存。之所以称为虚拟内存，是因为对它的所有操作，最终会变成一系列对真实物理内存的操作。

在此，笔者简单地介绍一下转换的过程，这有助于读者进一步理解 Windows 的内存管理。在 CPU 中有一个重要的寄存器 CR0，它是 32 位的寄存器，其中的一个位（PG 位）是负责告诉系统是否分页的。Windows 在启动前会将它的 PG 位置 1，即 Windows 允许分页。DDK 中有个宏 PAGE_SIZE 记录着分页大小，一般为 4KB。4GB 的虚拟内存会被分割成 1M 个（ $4GB/4KB=2^{20}$ ）分页单元。

其中，有一部分单元会和物理内存对应起来，即虚拟内存中第 N 个分页单元对应着物理内存第 M 个分页单元。这种对应不是一一对应，而是多对一的映射，多个虚拟内存页可以映射同一个物理内存页。还有一部分单元会被映射成磁盘上的文件，并标记为脏的（Dirty）。读取这段虚拟内存的时候，系统会发出一个异常，此时会触发异常处理函数，异常处理函数会将这个页的磁盘文件读入内存，并将标记设置为不脏。当让经常不读写的内存页，可以交换（Swap）成文件，并将此页设置为脏。还有一部分单元什么也没有对应，即空的。

笔者将上述过程简化成 5-2 示意图。从图 5-2 中可以看出，进程 1 和进程 2 虚拟内存映射的方式完全不同，有的物理内存块只映射到进程 1 上，因此无论对进程 2 怎样操作，也不会影响到那块内存。另外有些物理内存既映射到进程 1 上，也映射到进程 2 上。这样修改进程 2 的那段虚拟内存，进程 1 的相应的虚拟内存也会随之改变。这种方法就是所谓

的进程间共享内存。当然，这些都是严格控制在 Windows 操作系统之下的。大部分的虚拟内存是没有被映射到物理内存上的。

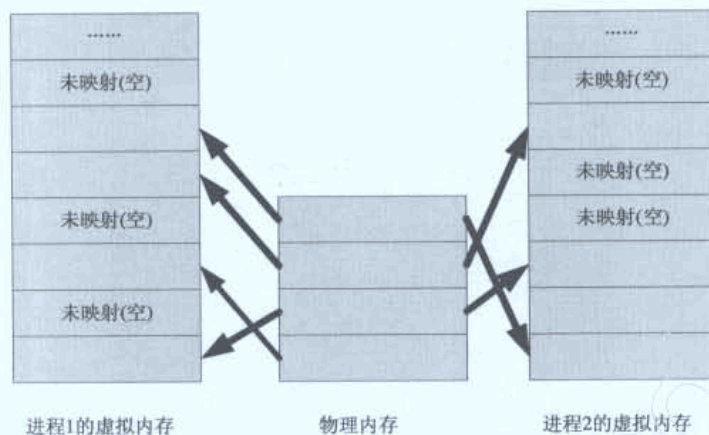


图 5-2 物理内存的映射

Windows 之所以如此设计，是基于以下两个原因。

- 第一是虚拟的增加了内存的大小。不管 PC 是否有足够的 4GB 的物理内存，操作系统总会有 4GB 的虚拟内存。这就允许使用者申请更多的内存，当物理内存不够时候，可以通过将不常用的虚拟内存页交换成文件，等需要的时候再去读取。
- 第二是使不同进程的虚拟内存互不干扰，为了让系统可以同时运行不同的进程，Windows 操作系统让每个进程看到的虚拟内存都不同。这个方法就使不同的进程会有不同的物理内存到虚拟内存的映射。例如进程 A 和进程 B 的内存地址 0x40000 会完全不同。修改 A 进程这个地址，不会影响到进程 B。因为 A 进程的这个地址可能映射的是一段物理内存地址，而 B 的这个地址映射的是另外一段物理内存地址。

5.1.3 用户模式地址和内核模式地址

虚拟地址在 0~0X7FFFFFFF 范围内的虚拟内存，即低 2GB 的虚拟地址，被称为用户模式地址。而 0X80000000~0xFFFFFFFF 范围内的虚拟内存，即高 2GB 的虚拟内存，被称为内核模式地址。Windows 规定运行在用户态（Ring3 层）的程序，只能访问用户模式地址，而运行在核心态（Ring0 层）的程序，可以访问整个 4GB 的虚拟内存，即用户模式地址和内核模式地址。

Windows 的核心代码和 Windows 的驱动程序加载的位置都是在高 2GB 的内核地址里，所以一般的应用程序是不能访问到这些核心代码和重要数据的。这大大提高了系统的稳健性。同时，Windows 操作系统在进程切换时，保持内核模式地址是完全相同的。也就是说，所有进程的内核地址映射完全一致，进程切换的时候，只改变用户模式地址的映射。

Windows 中内核模式地址和用户模式地址，如图 5-3 所示。

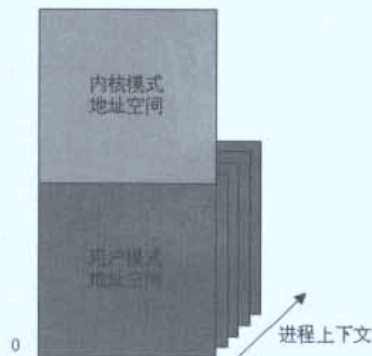


图 5-3 内核模式内存与用户模式内存

5.1.4 Windows 驱动程序和进程的关系

驱动程序可以看成是一个特殊的 DLL 文件被应用程序加载到虚拟内存中，只不过加载地址是内核模式地址，而不是用户模式地址。它能访问的只是这个进程的虚拟内存，而不能是其他进程的虚拟地址。需要指出的是，Windows 驱动程序里的不同例程运行在不同的进程中。DriverEntry 例程和 AddDevice 例程是运行在系统（System）进程中的。这个进程是 Windows 中非常重要的进程，也是 Windows 第一个运行的进程。当需要加载的时候，这个进程中会有一个线程将驱动程序加载到内核模式地址空间内，并调用 DriverEntry 例程。

而其他的一些例程，例如，IRP_MJ_READ 和 IRP_MJ_WRITE 的派遣函数会运行于应用程序的“上下文”中。所谓运行在进程的“上下文”，指的是运行于某个进程的环境中，所能访问的虚拟地址是这个进程的虚拟地址。

笔者有个技巧可以方便地看出代码是否运行于某个进程的上下文中。在代码中打印一行 log 信息，这行信息打印出当前进程的进程名。如果当前进程是发起 I/O 请求的进程，则说明在进程的“上下文”中。下面给出的函数可以显示出当前进程的进程名，读者可以在任意的例程中调用这个函数。读者可以用 DebugView 软件查看 log 信息。

```
#001 VOID DisplayItsProcessName()
#002 {
#003     //得到当前进程
#004     PEPROCESS pEProcess = PsGetCurrentProcess();
#005     //得到当前进程名称
#006     PTSTR ProcessName = (PTSTR)((ULONG)pEProcess + 0x174);
#007     KdPrint(("%%s\\n", ProcessName));
#008 }
```

此段代码可以在配套光盘中本章的 FileTest 目录下找到。

其中，PsGetCurrentProcess 函数是得到当前运行的进程，它是 EPROCESS 的结构体，然而 EPROCESS 这个结构体是微软没有公开的结构体，其中 0x174 偏移的位置，记录一

Windows 驱动开发技术详解

个字符串指针。有兴趣的读者，可以利用 WinDbg 工具，查看该结构体的内容。各版本 Windows 的该结构，可能略微有些差别，下面是笔者在 Windows XP SP2 的机器上用 WinDbg 查看该结构的结果。因为该结构十分庞大，只列出相关进程名的内容。

```
#001 lkd> dt _EPROCESS
#002 nt!_EPROCESS
#003      +0x000 Pcb                : _KPROCESS
#004      +0x06c ProcessLock       : _EX_PUSH_LOCK
#005      +0x070 CreateTime        : _LARGE_INTEGER
#006      (略)
#007      +0x170 Session           : Ptr32 Void
#008      +0x174 ImageFileName     : [16] UChar
#009      (略)
```

5.1.5 分页与非分页内存

前面已经介绍过了虚拟内存页和物理内存页之间的关系，Windows 规定有些虚拟内存页面是可以交换到文件中的，这类内存被称为分页内存。而有些虚拟内存页永远不会交换到文件中，这些内存被称为非分页内存。

当程序的中断请求级在 DISPATCH_LEVEL 之上时（包括 DISPATCH_LEVEL 层），程序只能使用非分页内存，否则将导致蓝屏死机。

在编译 DDK 提供的例程时，可以指定某个例程和某个全局变量是载入分页内存还是非分页内存，需要做如下定义：

```
#define PAGEDCODE code_seg("PAGE")
#define LOCKEDCODE code_seg()
#define INITCODE code_seg("INIT")

#define PAGEDDATA data_seg("PAGE")
#define LOCKEDDATA data_seg()
#define INITDATA data_seg("INIT")
```

如果将某个函数载入到分页内存中，我们需要在函数的实现中加入如下代码：

```
#001 #pragma PAGEDCODE
#002 VOID SomeFunction()
#003 {
#004     PAGED_CODE();
#005     //做一些事情
#006 }
```

其中，PAGED_CODE() 是 DDK 提供的宏，它只在 check 版本中生效。它会检验这个函数是否运行低于 DISPATCH_LEVEL 的中断请求级，如果等于或高于这个中断请求级，将产生一个断言。

如果让函数加载到非分页内存中，需要在函数的实现中加入如下代码：

```
#001 #pragma LOCKEDCODE
#002 VOID SomeFunction()
#003 {
```

```
#004    //做一些事情
#005 }
```

还有一种特殊情况，就是某个例程需要在初始化的时候载入内存，然后就可以从内存中卸载掉。这种情况指出现在 DriverEntry 情况下，尤其是 NT 式的驱动，DriverEntry 会很长，占据很大的空间，为了节省内存，需要及时地从内存中卸载掉。代码如下：

```
#001 #pragma INITCODE
#002 extern "C" NTSTATUS DriverEntry( IN PDRIVER_OBJECT pDriverObject,
#003                                  IN PUNICODE_STRING RegistryPath)
#004 {
#005
#006    //做一些事情
#007 }
```

5.1.6 分配内核内存

Windows 驱动程序使用的内存资源非常珍贵，分配内存时要尽量节约。和应用程序一样，局部变量是存放在栈（Stack）空间中的。但栈空间不会像应用程序那么大，所以驱动程序不适合递归调用或者局部变量是大型结构体。如果需要大型结构体，请在堆（Heap）中申请。

堆中申请内存的函数有以下几个，原型如下：

```
PVOID
ExAllocatePool(
    IN POOL_TYPE PoolType,
    IN SIZE_T NumberOfBytes
);
PVOID
ExAllocatePoolWithTag(
    IN POOL_TYPE PoolType,
    IN SIZE_T NumberOfBytes,
    IN ULONG Tag
);
PVOID
ExAllocatePoolWithQuota(
    IN POOL_TYPE PoolType,
    IN SIZE_T NumberOfBytes
);
PVOID
ExAllocatePoolWithQuotaTag(
    IN POOL_TYPE PoolType,
    IN SIZE_T NumberOfBytes,
    IN ULONG Tag
);
```

- PoolType 是个枚举变量，如果此值为 NonPagedPool，则分配非分页内存。如果此值为 PagedPool，则分配内存为分页内存。
- NumberOfBytes 是分配内存的大小，注意最好是 4 的倍数。
- 返回值分配的内存地址，一定是内核模式地址。如果返回 0，则代表分配失败。

以上四个函数功能类似，函数以 `WithQuota` 结尾的代表分配的时候按配额分配。函数以 `WithTag` 结尾的函数，和 `ExAllocatePool` 功能类似，唯一不同的是多了一个 `Tag` 参数，系统在要求的内存外又额外地多分配了 4 个字节的标签。在调试的时候，可以找出是否有标有这个标签的内存没有被释放。

以上四个函数都需指定 `PoolType`，分别可以指定如下几种。

- `NonPagedPool`: 指定要求分配非分页内存。
- `PagedPool`: 指定要求分配分页内存。
- `NonPagedPoolMustSucceed`: 指定分配非分页内存，必须成功。
- `DontUseThisType`: 未指定。
- `NonPagedPoolCacheAligned`: 指定要求分配非分页内存，而且必须内存对齐。
- `PagedPoolCacheAligned`: 指定分配分页内存，而且必须内存对齐。
- `NonPagedPoolCacheAlignedMustS`: 指定分配非分页内存，而且必须内存对齐，且必须成功。

将分配的内存，进行回收的函数是 `ExFreePool` 和 `ExFreePoolWithTag`，它们的原型是：

```
VOID
ExFreePool(
    IN PVOID P
);
NTKERNELAPI
VOID
ExFreePoolWithTag(
    IN PVOID P,
    IN ULONG Tag
);
```

- 参数 `P` 就是要释放的地址。

5.2 在驱动中使用链表

在驱动程序开发中，经常使用链表这种数据结构。DDK 为用户提供两种链表的数据结构，简化了对链表的操作。

链表中可以记录将整型、浮点、字符型或者程序员自定义的数据结构。链表通过指针将这些数据结构组成一条“链”，链中每个元素对应着记录的数据。对于单向链表，每个元素中有一个 `Next` 指针指向下一个元素。对于双向链表，每个元素有两个指针：`BLINK` 指针指向前一个元素，`FLINK` 指针指向下一个元素。

本节主要以使用双向链表为主，单向链表的操作类似。

5.2.1 链表结构

DDK 提供了标准的双向链表。双向链表可以将链表形成一个环。`BLINK` 指针指向前

一个元素，FLINK 指针指向下一个元素，如图 5-4 所示。

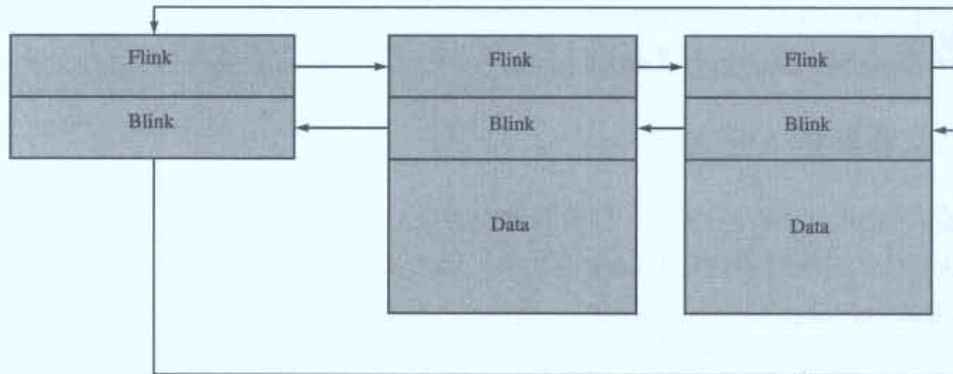


图 5-4 链表的基本结构

以下是 DDK 提供的双向链表的数据结构。读者可能会奇怪这个数据只有指向前后的指针，而没有数据。这个问题在下节进行介绍。

```
typedef struct _LIST_ENTRY {
    struct _LIST_ENTRY *Flink;
    struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY;
```

5.2.2 链表初始化

每个双向链表都是以一个链表头作为链表的第一个元素。初次使用链表头需要进行初始化，主要将链表头的 Flink 和 Blink 两个指针都指向自己。这意味着链表头所代表的链是空链，如图 5-5 所示。初始化链表头用 InitializeListHead 宏实现。

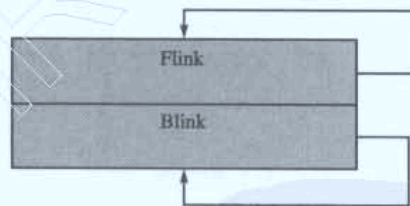


图 5-5 链表的初始化

如何判断链表是否为空，可以判断链表头的 Flink 和 Blink 是否指向自己。DDK 为程序员提供了一个宏简化这种检查，这就是 IsListEmpty。

```
IsListEmpty(&head);
```

程序员需要自己定义链表中每个元素的数据类型，并将 LIST_ENTRY 结构作为自定义结构的一个子域。LIST_ENTRY 的作用是将自定义的数据结构串成一个链表。

```
typedef struct _MYDATASTRUCT {
    // List Entry 需要作为_MYDATASTRUCT 结构体的一部分
```



```

LIST_ENTRY ListEntry;
// 下面是自定义的数据
ULONG x;
ULONG y;
} MYDATASTRUCT, *PMYDATASTRUCT;

```

5.2.3 从首部插入链表

对链表的插入有两种方式，一种是在链表的头部插入，一种是在链表的尾部插入。在头部插入链表使用语句 `InsertHeadList`，用法如下：

```
InsertHeadList(&head, &mydata->ListEntry);
```

其中，`head` 是 `LIST_ENTRY` 结构的链表头，`mydata` 是用户定义的数据结构，而它的子域 `ListEntry` 是包含其中的 `LIST_ENTRY` 数据结构。

假设链表中已经有一个元素 `Entry1`，如图 5-6 所示。现在将另外一个元素 `Entry2` 插入链表，用 `InsertHeadList` 插入。链表插入 `Entry2` 的结果如图 5-7 所示。

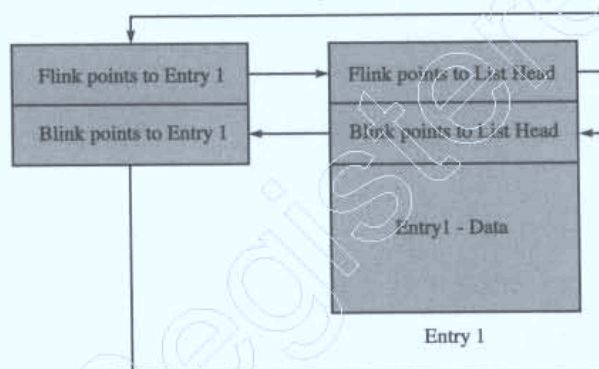


图 5-6 插入前链表状态

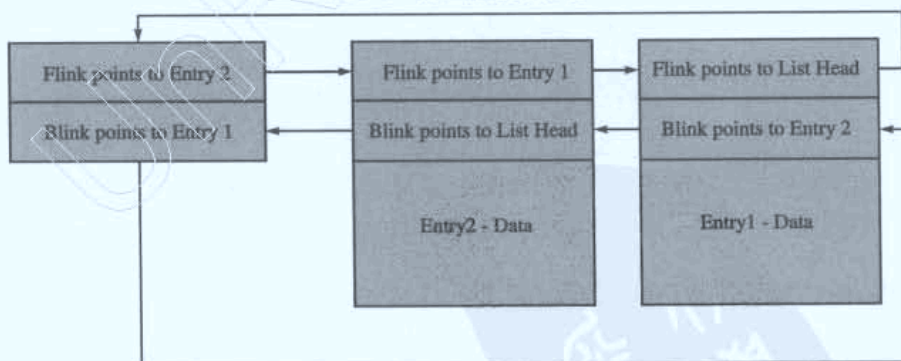


图 5-7 插入后链表状态

5.2.4 从尾部插入链表

另外一种插入方法是在链表的尾部进行插入。在尾部插入链表使用语句 `InsertTailList`，

用法如下：

```
InsertTailList (&head,&mydata->ListEntry);
```

其中，head 是 LIST_ENTRY 结构的链表头，mydata 是用户定义的数据结构，而它的子域 ListEntry 是包含其中的 LIST_ENTRY 数据结构。

假设链表中已经有一个元素 Entry1 和 Entry2，如图 5-8 所示。现在将另外一个元素 Entry3 插入链表，用 InsertTailList 插入。链表插入 Entry3 的结果如图 5-8 所示。

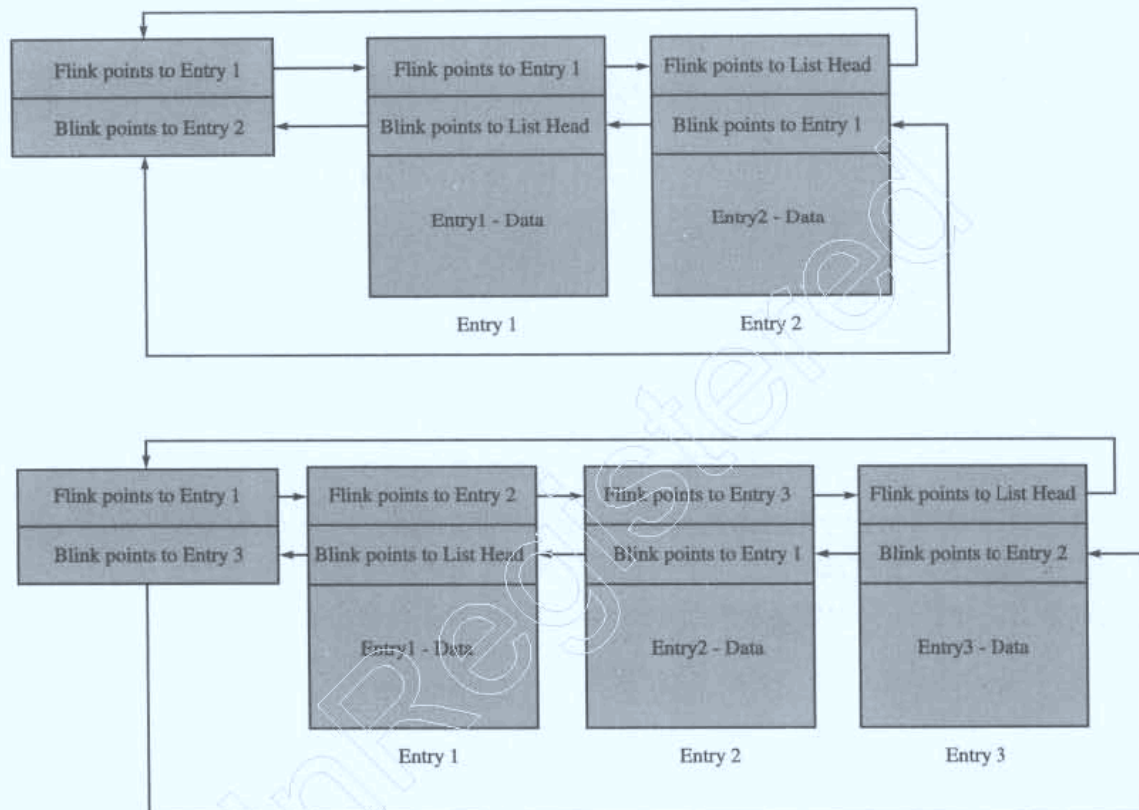


图 5-8 从链表的尾部插入链表

由此可见，InsertTailList 函数是将插入的元素插到了链表的尾部。

5.2.5 从链表删除

和插入链表一样，删除也有两种对应的方法。一种是从链表头部删除，一种是从链表尾部删除。分别对应 RemoveHeadList 和 RemoveTailList 函数。其使用方法如下：

```
PLIST_ENTRY pEntry = RemoveHeadList(&head);
```

及

```
PLIST_ENTRY pEntry = RemoveTailList (&head);
```


其中, head 是链表头, pEntry 是从链表删除下来的元素中的 ListEntry。这里有一个问题, 就是如何从 pEntry 得到用户自定义数据结构的指针。这里有以下两种情况。

(1) 当自定义的数据结构的第一个字段是 LIST_ENTRY 时, 如:

```
typedef struct _MYDATASTRUCT {
    LIST_ENTRY ListEntry;
    ULONG x;
    ULONG y;
} MYDATASTRUCT, *PMYDATASTRUCT;
```

此时, RemoveHeadList 返回的指针可以当做用户自定义的指针, 即:

```
PLIST_ENTRY pEntry = RemoveHeadList(&head);
PMYDATASTRUCT pMyData = (PMYDATASTRUCT) pEntry;
```

(2) 当自定义的数据结构的第一个字段不是 LIST_ENTRY 时, 如:

```
typedef struct _MYDATASTRUCT {
    ULONG x;
    ULONG y;
    LIST_ENTRY ListEntry;
} MYDATASTRUCT, *PMYDATASTRUCT;
```

此时, RemoveHeadList 返回的指针不可以当做用户自定义的指针, 即:

```
PLIST_ENTRY pEntry = RemoveHeadList(&head);
PMYDATASTRUCT pMyData = (PMYDATASTRUCT) pEntry; // (错误!!!!)
```

此时需要通过 pEntry 的地址逆向算出自定义数据的指针。一般通过 pEntry 在自定义数据中的偏移量, 用 pEntry 减去这个偏移量, 就会得到用户自定义结构的指针的地址。如图 5-9 所示, 指针 A 是自定义数据结构的地址, 指针 B 是自定义数据结构中的 pEntry。

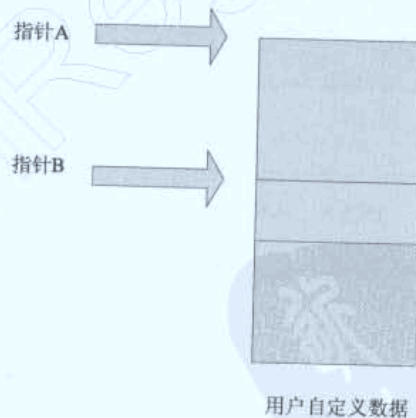


图 5-9 使用 CONTAINING_RECORD 宏

为了简化这个操作, DDK 为程序员提供了宏 CONTAINING_RECORD, 其用法如下:

```
#001 PLIST_ENTRY pEntry = RemoveHeadList(&head);
#002 PIRP pIrp = CONTAINING_RECORD(pEntry,
#003                               MYDATASTRUCT,
#004                               ListEntry);
```

CONTAINING_RECORD 的第一参数是相当于图 5-9 中指针 B，第二个参数是数据结构的名称，第三个参数是数据结构中的字段，返回的相当于图 5-9 中的指针 A。

DDK 建议无论自定义数据结构的第一个字段是否为 ListEntry，最好都使用 CONTAINING_RECORD 宏得到自定义数据结构的指针。

5.2.6 实验

在介绍完内存分配和链表操作后，下面进行一次实验。实验的目的：在驱动程序中，初始化链表，自定义链表数据结构。该结构包含一个整型数字，这个实验演示了向链表进行插入、删除等操作。其主要代码如下：

```
#001 VOID LinkListTest()
#002 {
#003     LIST_ENTRY linkListHead;
#004     //初始化链表
#005     InitializeListHead(&linkListHead);
#006
#007     PMYDATASTRUCT pData;
#008     ULONG i = 0;
#009     //在链表中插入 10 个元素
#010     KdPrint(("Begin insert to link list"));
#011     for (i=0 ; i<10 ; i++)
#012     {
#013         //分配分页内存
#014         pData = (PMYDATASTRUCT)
#015             ExAllocatePool(PagedPool, sizeof(MYDATASTRUCT));
#016         pData->number = i;
#017         //从头部插入链表
#018         InsertHeadList(&linkListHead, &pData->ListEntry);
#019     }
#020
#021     //从链表中取出，并显示
#022     KdPrint(("Begin remove from link list\n"));
#023     while(!IsListEmpty(&linkListHead))
#024     {
#025         //从尾部删除一个元素
#026         PLIST_ENTRY pEntry = RemoveTailList(&linkListHead);
#027         pData = CONTAINING_RECORD(pEntry,
#028                                 MYDATASTRUCT,
#029                                 ListEntry);
#030         KdPrint(("&d\n", pData->number));
#031         ExFreePool(pData);
#032     }
#033 }
```

此段代码可以在配套光盘中本章的 LinkList 目录下找到。

将上述代码插入到 HelloDDK 中的 DriverEntry 中。驱动加载后可以查看相应的 log 信息，如图 5-10 所示。

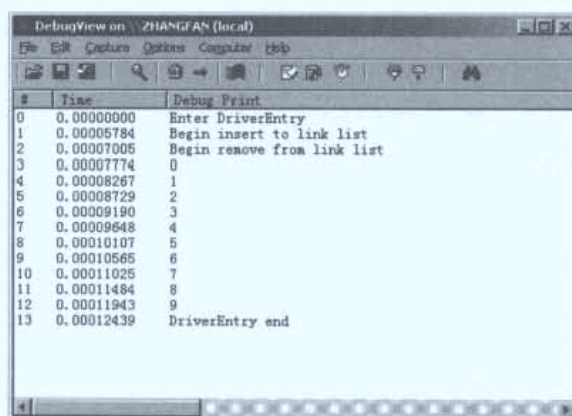


图 5-10 插入链表演示程序结果

5.3 Lookaside 结构

频繁申请和回收内存，会导致在内存上产生大量的内存“空洞”，从而导致最终无法申请内存。DDK 为程序员提供了 Lookaside 结构来解决这个问题。

5.3.1 频繁申请内存的弊端

频繁地申请内存，会导致一个问题，就是在内存中产生“空洞”。图 5-11 显示了这种情况，在内存中先后申请三块内存。最开始可用的内存是连续的。当某个时刻内存块 2 被回收以后，如果系统想分配一块略微大于原先内存块 2 的内存，这时候原先的内存 2 就不能被申请成功。因此，频繁地申请、回收内存会导致在内存上产生大量的内存“空洞”。

如果系统中存在大量的内存“空洞”，即使内存中有大量的可用内存，也会导致申请内存失败。在操作系统空闲的时候，系统会整理内存中的“空洞”，将内存中的“空洞”进行合并。

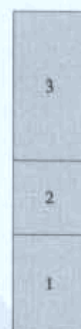


图 5-11 内存碎片的产生

5.3.2 使用 Lookaside

如果驱动程序需要频繁地从内存中申请、回收固定大小的内存，DDK 提供了一种机制来解决这个问题，这就是使用 Lookaside 对象。

可以将 Lookaside 对象想象成一个内存容器。在初始的时候，它先向 Windows 申请了一块比较大的内存。以后程序员每次申请内存的时候，不是直接向 Windows 申请内存，而是向 Lookaside 对象申请内存。Lookaside 对象会智能地避免产生内存“空洞”。如果

Lookaside 对象内部的内存不够用时，它会向操作系统申请更多的内存。当 Lookaside 对象内部有大量的未使用的内存时，它会自动让 Windows 回收一部分内存。总之，Lookaside 是一个自动的内存分配容器。通过对 Lookaside 对象申请内存，效率要高于直接向 Windows 申请内存。Lookaside 一般会在以下情况使用：

- (1) 程序员每次申请固定大小的内存。
- (2) 申请和回收的操作十分频繁。

如果程序员遇到上述两种情况，可以考虑使用 Lookaside 对象。驱动程序中的运行效率是程序员必须考虑的问题。如果驱动程序的运行效率低，会严重影响到操作系统的性能。使用 Lookaside 对象，首先要初始化 Lookaside 对象，有以下两个函数可以使用：

```
VOID
ExInitializeNPagedLookasideList(
    IN PNPAGED_LOOKASIDE_LIST Lookaside,
    IN PALLOCATE_FUNCTION Allocate OPTIONAL,
    IN PFREE_FUNCTION Free OPTIONAL,
    IN ULONG Flags,
    IN SIZE_T Size,
    IN ULONG Tag,
    IN USHORT Depth
);
```

和

```
VOID
ExInitializePagedLookasideList(
    IN PPAGED_LOOKASIDE_LIST Lookaside,
    IN PALLOCATE_FUNCTION Allocate OPTIONAL,
    IN PFREE_FUNCTION Free OPTIONAL,
    IN ULONG Flags,
    IN SIZE_T Size,
    IN ULONG Tag,
    IN USHORT Depth
);
```

这两个函数分别是对非分页和分页 Lookaside 对象进行初始化。

在初始化完 Lookaside 对象后，可以进行申请内存的操作了，有以下两个函数：

```
PVOID
ExAllocateFromNPagedLookasideList(
    IN PNPAGED_LOOKASIDE_LIST Lookaside
);
```

和

```
PVOID
ExAllocateFromPagedLookasideList(
    IN PPAGED_LOOKASIDE_LIST Lookaside
);
```

这两个函数分别是对非分页内存和分页内存的申请。

对 Lookaside 对象回收内存的操作，有以下两个函数：


```
VOID
ExFreeToNPagedLookasideList(
    IN PNPAGED_LOOKASIDE_LIST Lookaside,
    IN PVOID Entry
);
```

和

```
VOID
ExFreeToPagedLookasideList(
    IN PPAGED_LOOKASIDE_LIST Lookaside,
    IN PVOID Entry
);
```

这两个函数分别是对非分页内存和分页内存的回收。

在使用完 Lookaside 对象后，需要删除 Lookaside 对象，有以下两个函数：

```
VOID
ExDeleteNPagedLookasideList(
    IN PNPAGED_LOOKASIDE_LIST Lookaside
);
```

和

```
VOID
ExDeletePagedLookasideList(
    IN PPAGED_LOOKASIDE_LIST Lookaside
);
```

这两个函数分别是对非分页和分页 Lookaside 对象的删除。

5.3.3 实验

在介绍完 Lookaside 对象的使用后，下面做一个实验。这个实验利用 Lookaside 对象频繁地申请和回收内存。

```
#001 #pragma INITCODE
#002 VOID LookasideTest()
#003 {
#004     //初始化lookaside对象
#005     PAGED_LOOKASIDE_LIST pageList;
#006     ExInitializePagedLookasideList(&pageList, NULL, NULL, 0, sizeof
(MYDATASTRUCT), '1234', 0);
#007     #define ARRAY_NUMBER 50
#008     PMYDATASTRUCT MyObjectArray[ARRAY_NUMBER];
#009     //模拟频繁申请内存
#010     for (int i=0; i<ARRAY_NUMBER; i++)
#011     {
#012         MyObjectArray[i] = (PMYDATASTRUCT)ExAllocateFromPagedLookasideList
(&pageList);
#013     }
#014     //模拟频繁回收内存
#015     for (i=0; i<ARRAY_NUMBER; i++)
#016     {
#017         ExFreeToPagedLookasideList(&pageList, MyObjectArray[i]);
#018         MyObjectArray[i] = NULL;
```

```
#019     }
#020     ExDeletePagedLookasideList(&pageList);
#021     //删除 Lookaside 对象
#022 }
```

此段代码可以在配套光盘中本章的 Lookaside 目录下找到。

5.4 运行时函数

一般编译器厂商，在发布其编译器的同时，会将运行时函数一起发布给用户。运行时函数是程序运行的时候必不可少的，它由编译器提供。针对不同的操作系统，运行时函数的实现方法不同，但接口基本保持一致。例如，malloc 函数就是典型的运行时函数，所有编译器厂商都必须提供这个函数，它在不同操作系统上的实现方法就不尽相同。

5.4.1 内存间复制（非重叠）

在驱动程序开发中，经常用到内存的复制。例如，将需要显示的内容，从缓冲区复制到显卡内存中。DDK 为程序员提供了以下函数。

```
VOID
RtlCopyMemory(
    IN VOID UNALIGNED *Destination,
    IN CONST VOID UNALIGNED *Source,
    IN SIZE_T Length
);
```

- Destination: 表示要复制内存的目的地址。
- Source: 表示要复制内存的源地址。
- Length: 表示要复制内存的长度，单位是字节。

和 RtlCopyMemory 功能类似的函数有 RtlCopyBytes，这两个函数的参数全部一样，功能也完全一样，只是在不同的平台下有不同的实现。

例如，在 IA32（Inter 32 位 CPU）平台下，这两个函数其实是两个宏，分别定义为：

```
#define RtlCopyMemory(Destination,Source,Length) memcpy((Destination),(Source),\
(Length))
#define RtlCopyBytes RtlCopyMemory
```

这两个函数在 IA32 平台下，是依靠 memcpy 实现的。另外，Windows XP DDK 又添加了一个新的函数 RtlCopyMemory32，这个函数不是依靠 memcpy 实现的，因为是针对 32 位搬移，速度做了优化。

而在其他平台下（例如，Alpha 平台），RtlCopyMemory、RtlCopyMemory32 和 RtlCopyBytes 都有各自的实现方法。为了保证代码的可移植性，尽量不要使用 memcpy 函数，而使用 RtlCopyMemory 函数或者 RtlCopyBytes 函数。

5.4.2 内存间复制（可重叠）

用 `RtlCopyMemory` 可以复制内存，但其内部没有考虑内存重叠的情况。如图 5-12 所示，有三段等长的内存，ABCD 分别代表三段内存的起始地址和终止地址。如果需要将 A 到 C 段的内存复制到 B 到 D 这段内存上，这时 B 到 C 段的内存就是重叠的部分。

`RtlCopyMemory` 函数的内部实现方法是依靠 `memcpy` 函数实现的。根据 C99 定义，`memcpy` 没有考虑重叠的部分，因此它不能保证重叠部分是否被复制。

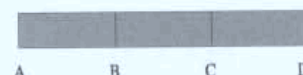


图 5-12 内存复制

为了保证重叠部分也被正确复制，C99 规定 `memmove` 函数完成这个任务。这个函数对两个内存是否重叠进行了判断，这种判断却牺牲了速度。

如果程序员能确保复制的内存没有重叠，请选择使用 `memcpy` 函数。如果不能保证内存是否重叠，请选择使用 `memmove` 函数。同样，为了保证可移植性，DDK 用宏对 `memmove` 进行了封装，名称变为 `RtlMoveMemory`。

```
VOID
RtlMoveMemory(
    IN VOID UNALIGNED *Destination,
    IN CONST VOID UNALIGNED *Source,
    IN SIZE_T Length
);
```

- Destination: 表示要复制内存的目的地址。
- Source: 表示要复制内存的源地址。
- Length: 表示要复制内存的长度。单位是字节。

5.4.3 填充内存

驱动程序开发中，还经常用到对某段内存区域用固定字节填充。DDK 为程序员提供了函数 `RtlFillMemory`。它在 IA32 平台下也是个宏，实际的函数是 `memset` 函数。

```
VOID
RtlFillMemory(
    IN VOID UNALIGNED *Destination,
    IN SIZE_T Length,
    IN UCHAR Fill
);
```

- Destination: 目的地址。
- Length: 长度。
- Fill: 需要填充的字节。

需要注意的是，该函数和 `memset` 函数的参数顺序不一致。`RtlFillMemory` 的第二个参数和第三个参数在 `RtlFillMemory` 中是相反的。

```
#define RtlFillMemory(Destination,Length,Fill) memset((Destination),(Fill),
(Length))
```

在驱动程序开发中，还经常要对某段内存填零，DDK 提供的宏是 `RtlZeroBytes` 和 `RtlZeroMemory`。

```
VOID
RtlZeroMemory(
    IN VOID UNALIGNED *Destination,
    IN SIZE_T Length
);
```

➤ `Destination`: 目的地址。

➤ `Length`: 长度。

它们在 IA32 平台下是依靠 `memset` 实现的。

```
#define RtlZeroMemory(Destination,Length) memset((Destination),0,(Length))
```

5.4.4 内存比较

驱动程序开发中，还会用到比较两块内存是否一致。该函数是 `RtlCompareMemory`，其声明是：

```
ULONG
RtlEqualMemory(
    CONST VOID *Source1,
    CONST VOID *Source2,
    SIZE_T Length
);
```

➤ `Source1`: 比较的第一个内存地址。

➤ `Source2`: 比较的第二个内存地址。

➤ `Length`: 比较的长度，单位为字节。

➤ 返回值: 相等的字节数。

`RtlEqualMemory` 通过判断返回值和 `Length` 是否相等，来判断两块内存是否完全一致。同时，DDK 还提供了一个宏，直接判断是否一致。

```
#define RtlEqualMemory(Destination,Source,Length) (!memcmp((Destination),(Source),
(Length)))
```

`RtlEqualMemory` 在两段内存一致的情况下返回非零值，在不一致的情况下返回零。

5.4.5 关于运行时函数使用的注意事项

以上介绍了内存相关的运行时函数，DDK 提供的标准的运行时函数名都是 `RtlXX` 形式。其中，大部分是以宏的形式给出，例如，`RtlCopyMemory` 就是一个宏，其定义为：

```
#define RtlCopyMemory(Destination,Source,Length) memcpy((Destination),(Source),
(Length))
```


memcpy 等这些函数都是由 ntoskrnl.exe 导出的函数。

下面通过 VC 提供的工具 Depends 查看这些导出函数, 如图 5-13 所示。

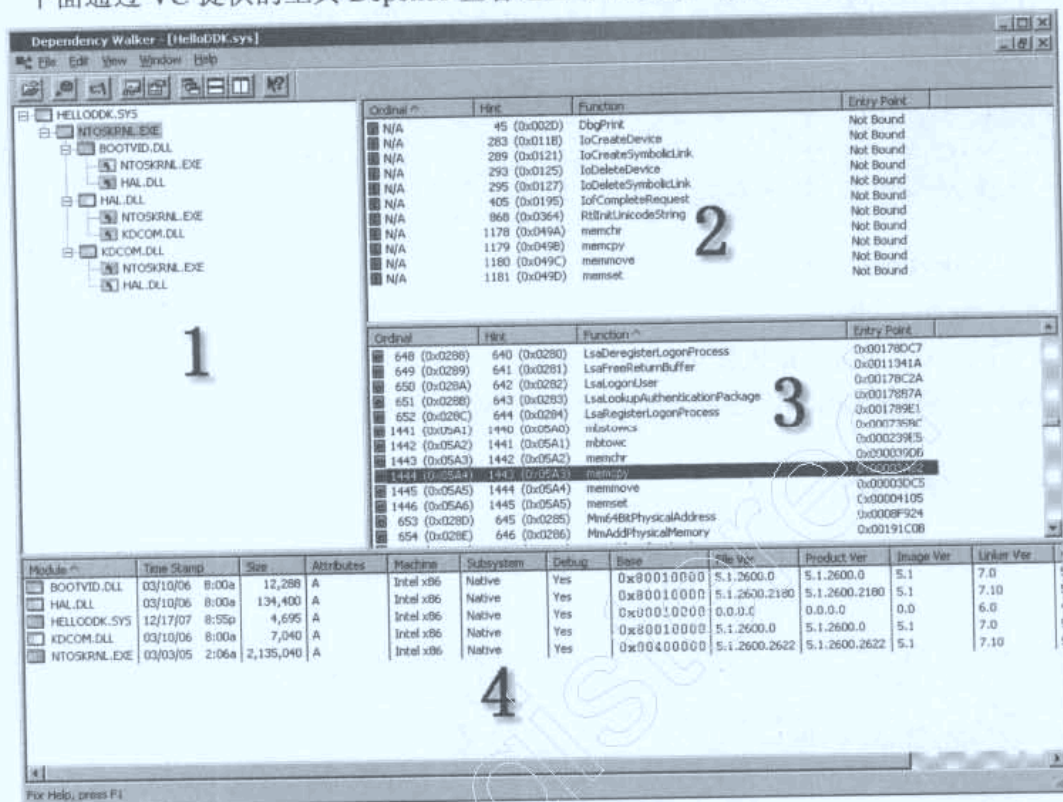


图 5-13 Depends 工具

用 Depends 工具打开编译的 DDK 驱动程序, 显示出以下 4 部分信息。

(1) 在标号为 1 的窗口中, 可以看到驱动程序 HelloDDK.sys 依赖的动态链接库 (ntoskrnl.exe)。一般的动态链接库都是以 DLL 文件形式存在的, 而 ntoskrnl.exe 却是以 EXE 文件形式存在的。但不管怎样, 它们都是 PE 格式的文件。ntoskrnl.exe 同时依赖于三个动态链接库。

(2) 在标号为 2 的窗口中, 列出了 HelloDDK.sys 中使用 ntoskrnl.exe 的导出函数。例如, 这个驱动程序中用到了 RtlCopyMemory 宏, 而这个宏的定义是 memcpy 函数, 因此在导出函数列表中可以找到 memcpy 函数。又例如, 驱动程序用到了 KdPrint 宏, 它其实是 DbgPrint 函数。DbgPrint 函数是 ntoskrnl.exe 的导出函数。

(3) 在标号为 3 的窗口, 列出了 ntoskrnl.exe 所有的导出函数。窗口 3 中列出的函数包含了窗口 2 中列出的函数。DDK 提供的函数基本都是由 ntoskrnl.exe 导出的函数。这些函数有的可以在 DDK 文档中, 找到使用说明, 如 IoCreateDevice 函数。而有的函数则在 DDK 中找不到使用说明, 比如刚才讲的 memcpy 函数。这些未文档化的函数, 往往是通过宏调用的。尽量避免使用这些未文档化的函数, 而使用 DDK 文档化的宏。

(4) 标号为4的窗口,列出了所有递归需要的动态链接库。同时,这里还列出了这些动态链接库需要加载的地址。

5.4.6 实验

在介绍完主要的运行时函数后,下面做一个实验。在这里,将前面介绍的运行时函数——进行验证。这个实验的目的是检验内存间复制、填充内存、内存比较等。

```
#001  #define BUFFER_SIZE 1024
#002  #pragma INITCODE
#003  VOID RtlTest()
#004  {
#005      PCHAR pBuffer = (PCHAR)ExAllocatePool(PagedPool,BUFFER_SIZE);
#006      //用零填充内存
#007      RtlZeroMemory(pBuffer,BUFFER_SIZE);
#008      PCHAR pBuffer2 = (PCHAR)ExAllocatePool(PagedPool,BUFFER_SIZE);
#009      //用固定字节填充内存
#010      RtlFillMemory(pBuffer2,BUFFER_SIZE,0xAA);
#011      //内存复制
#012      RtlCopyMemory(pBuffer,pBuffer2,BUFFER_SIZE);
#013      //判断内存是否一致
#014      ULONG ulRet = RtlCompareMemory(pBuffer,pBuffer2,BUFFER_SIZE);
#015      if (ulRet==BUFFER_SIZE)
#016      {
#017          KdPrint(("The two blocks are same.\n"));
#018      }
#019  }
```

此段代码可以在配套光盘中本章的 RtlTest 目录下找到。

5.5 使用 C++特性分配内存

在 C++语言中分配内存时,可以使用 new 操作符,回收内存时使用 delete 操作符。但是在驱动程序开发中,使用 new 和 delete 操作符,将得到错误的链接指示。如:

```
PCHAR p = new CHAR[1024];
```

编译的时候,会得到如下的错误报告:

```
-----Configuration: DriverDev - Win32 Driver Check Edition-----
Compiling...
Driver.cpp
Linking...
Driver.obj : error LNK2001: unresolved external symbol "void * __cdecl operator
new(unsigned int)" (??2@YAPAXI@Z)
MyDriver_Check/HelloDDK.sys : fatal error LNK1120: 1 unresolved externals
Error executing link.exe.
HelloDDK.sys - 2 error(s), 0 warning(s)
```

这说明在驱动程序中,微软编译器没有提供内核模式下的 new 操作符。如果读者希望在驱动程序中使用 new 和 delete 操作符,应该对 new 和 delete 操作符进行重载(overload)。

Windows 驱动开发技术详解

`new` 和 `delete` 操作符可以通过内核分配内存函数 `ExAllocatePool` 和回收函数 `ExFreePool` 实现。同时还可通过 `new` 操作符指定使用分页内存还是非分页内存。

重载 `new` 和 `delete` 操作符，有两种方法，一种是全局重载，一种是在类中重载。以下是笔者给出的示例代码。

```
#001 //全局 new 操作符
#002 void * __cdecl operator new(size_t size, POOL_TYPE PoolType=PagedPool)
#003 {
#004     KdPrint(("global operator new\n"));
#005     KdPrint(("Allocate size :%d\n",size));
#006     return ExAllocatePool(PagedPool,size);
#007 }
#008 //全局 delete 操作符
#009 void __cdecl operator delete(void* pointer)
#010 {
#011     KdPrint(("Global delete operator\n"));
#012     ExFreePool(pointer);
#013 }
#014
#015 class TestClass
#016 {
#017 public:
#018     //构造函数
#019     TestClass()
#020     {
#021         KdPrint(("TestClass::TestClass()\n"));
#022     }
#023
#024     //析构函数
#025     ~TestClass()
#026     {
#027         KdPrint(("TestClass::~~TestClass()\n"));
#028     }
#029
#030     //类中的 new 操作符
#031     void* operator new(size_t size, POOL_TYPE PoolType=PagedPool)
#032     {
#033         KdPrint(("TestClass::new\n"));
#034         KdPrint(("Allocate size :%d\n",size));
#035         return ExAllocatePool(PoolType,size);
#036     }
#037
#038     //类中的 delete 操作符
#039     void operator delete(void* pointer)
#040     {
#041         KdPrint(("TestClass::delete\n"));
#042         ExFreePool(pointer);
#043     }
#044 private:
#045     char buffer[1024];
#046 };
#047
#048 void TestNewOperator()
#049 {
#050     //测试 new 操作符
#051     TestClass* pTestClass = new TestClass;
```

```
#052 //测试 delete 操作符
#053 delete pTestClass;
#054
#055 pTestClass = new(NonPagedPool) TestClass;
#056 delete pTestClass;
#057
#058 char *pBuffer = new(PagedPool) char[100];
#059 delete []pBuffer;
#060
#061 pBuffer = new(NonPagedPool) char[100];
#062 delete []pBuffer;
#063 }
```

此段代码可以在配套光盘中本章的 new 目录下找到。

5.6 其他

在本章结束前，再向读者介绍一些驱动程序开发中的注意事项。

5.6.1 数据类型

用 C 语言或者 C++语言开发时，字符变量、短整型整数、长整型整数都有自己标准的数据类型。DDK 对这些数据结构进行了封装。在驱动程序中，既可以使用 C 语言的类型定义，也可以使用 DDK 提供的类型定义。表 5-1 列出了 C 语言的数据类型和 DDK 中对应的数据类型。

表 5-1 数据类型定义

C 语言的定义	DDK 中的定义
void	VOID
char	CHAR
short	SHORT
long	LONG
wchar_t	WCHAR
char*	PCHAR
wchar_t*	PWCHAR

在 C 语言中，整数类型有 8 位、16 位、32 位三种类型，而在 DDK 中又新添了一种 64 位的长整型整数。这种 64 位的整数只有无符号形式，表示范围从 0 到 $2^{64}-1$ ，用 LONGLONG 类型表示。64 位形整数不是标准 C 语言定义的，只有微软的编译器才识别这种类别。64 位整数的常量前面是一个数字，后面加上 i64 结尾。如：

```
LONGLONG llValue = 100i64;
```

这种 64 位整数支持加、减、乘、除等运算。

除了 LONGLONG 之外，DDK 还提供了另外一种的 64 位整数的表示方法，即

LARGE_INTEGER 数据结构。其区别是 LONGLONG 是基本的数据, 而 LARGE_INTEGER 是数据结构。LARGE_INTEGER 定义如下:

```
typedef union _LARGE_INTEGER {
    struct {
        ULONG LowPart;
        LONG HighPart;
    };
    struct {
        ULONG LowPart;
        LONG HighPart;
    } u;
    LONGLONG QuadPart;
} LARGE_INTEGER;
```

LARGE_INTEGER 是个联合体, 这种设计非常巧妙。联合体中的三个元素可以认为是 LARGE_INTEGER 的三个定义。

(1) LARGE_INTEGER 可以认为是由两个部分组成。一个是低 32 位的整数 HighPart, 一个是高 32 位的整数 HighPart。在 little endian 的情况下, 低 32 位数字在前, 高 32 位数字在后。

如果将这个 64 位数赋值为 100, 可以这样写:

```
#001    LARGE_INTEGER LargeValue;
#002    LargeValue.LowPart = 100;
#003    LargeValue.HighPart = 0;
```

(2) LARGE_INTEGER 可以认为是由两个部分组成。一个是低 32 位的整数 HighPart, 一个是高 32 位的整数 HighPart。在 big endian 的情况下, 高 32 位数字在前, 低 32 位数字在后。

如果想将这个 64 数赋值为 100, 可以这样写:

```
#001    LARGE_INTEGER LargeValue;
#002    LargeValue.u.LowPart = 100;
#003    LargeValue.u.HighPart = 0;
```

(3) LARGE_INTEGER 等价于 LONGLONG 数据。在这种情况下, 如果想将这个数据结构赋值 100, 可以这样写:

```
#001    LARGE_INTEGER LargeValue;
#002    liValue.QuadPart = 100i64;
```

5.6.2 返回状态值

DDK 大部分函数的返回值类型是 NTSTATUS 类型。查看 DDK.h 文件, 可以看到:

```
typedef LONG NTSTATUS;
```

NTSTATUS 的定义和 LONG 等价。为了函数的形式统一, 所有函数的返回值都是 NTSTATUS 类型。NTSTATUS 就是一个 32 位的整数, 其每位有着不同的含义, 如图 5-14 所示。

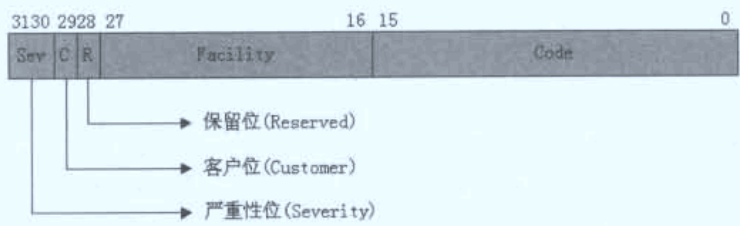


图 5-14 NTSTATUS 含义

在执行完内核函数后，应该查看该函数的返回状态。如果状态码高位为 0，无论其他位置是否设置，该状态代码代表成功。绝对不能用状态代码与 0 比较来判断操作是否成功，而应该使用 NT_SUCCESS 宏，其用法如下：

```
#001     status =Foo(...);
#002     if (NT_SUCCESS(status))
#003     {
#004         //该函数执行成功
#005     }
```

DDK 定义了大量的状态的返回值，都是形如 STATUS_XX 定义的宏，如表 5-1 所示。

表 5-2 常用 NTSTATUS 状态返回值

分 类	描 述
STATUS_SUCCESS	函数执行成功
STATUS_UNSUCCESSFUL	函数执行不成功
STATUS_NOT_IMPLEMENTED	函数未被实现
STATUS_INVALID_INFO_CLASS	输入参数是无效的类别
STATUS_INFO_LENGTH_MISMATCH	输入参数长度不匹配
STATUS_ACCESS_VIOLATION	不允许访问
STATUS_IN_PAGE_ERROR	发生页故障
STATUS_INVALID_HANDLE	输入是无效的句柄
STATUS_INVALID_PARAMETER	输入是无效的参数
STATUS_NO_SUCH_DEVICE	指定的设备不存在
STATUS_NO_SUCH_FILE	指定的文件不存在
STATUS_INVALID_DEVICE_REQUEST	无效的设备请求
STATUS_END_OF_FILE	文件已到结尾
STATUS_INVALID_SYSTEM_SERVICE	无效的系统调用
STATUS_ACCESS_DENIED	访问被拒绝
STATUS_BUFFER_TOO_SMALL	输入的缓冲区过小
STATUS_OBJECT_TYPE_MISMATCH	输入的对象类型不匹配
STATUS_OBJECT_NAME_INVALID	输入的对象名无效
STATUS_OBJECT_NAME_NOT_FOUND	输入的对象没有找到
STATUS_PORT_DISCONNECTED	需要链接的端口没有被连接
STATUS_OBJECT_PATH_INVALID	输入的对象路径无效

5.6.3 检查内存可用性

在驱动程序开发中，对内存的操作要格外小心。如果某段内存是只读的，而驱动程序试图去写操作，会导致系统的崩溃。同样，当某段内存是不可读的情况下，驱动程序试图去读，同样会导致系统的崩溃。

DDK 提供了两个函数，帮助程序员在不知道某段内存是否可读写的情况下，试探这段内存可读写性。这两个函数分别是 `ProbeForRead` 和 `ProbeForWrite`。

```
VOID
ProbeForRead(
    IN CONST VOID *Address,
    IN SIZE_T Length,
    IN ULONG Alignment
);
```

- **Address:** 需要被检查的内存的地址。
- **Length:** 需要被检查的内存的长度，单位是字节。
- **Alignment:** 描述该段内存是以多少字节对齐的。

```
VOID
ProbeForWrite (
    IN CONST VOID *Address,
    IN SIZE_T Length,
    IN ULONG Alignment
);
```

- **Address:** 需要被检查的内存的地址。
- **Length:** 需要被检查的内存的长度，单位是字节。
- **Alignment:** 描述该段内存是以多少字节对齐的。

这两个函数不是返回该段内存是否可读写，而是当不可读写的时候，引发一个异常（Exception）。这个异常需要用到微软编译器提供的“结构化异常”处理办法。“结构化异常”机制会轻松地检测到这种异常，进而做出相应的异常处理。关于结构化异常的介绍，见下一节。

5.6.4 结构化异常处理（try-except 块）

结构化异常处理（Structured Exception Handling）是微软编译器提供的独特处理机制，这种处理方式能在一定程度上在出现错误的情况下，免于程序崩溃。为了说明结构化异常，有两个概念需要说明一下。

（1）异常：异常的概念类似于中断的概念，当程序中某种错误触发一个异常，操作系统会寻找处理这个异常的处理函数。如果程序提供错误处理函数，则进入错误处理函数，如果没有提供处理函数，则由操作系统的默认错误处理函数处理。在内核模式下，操作系

统默认处理错误的办法往往很简单，直接让系统蓝屏，并在蓝屏上简单描述出错的信息，之后系统就进入死机状态。这当然不是程序员所希望的，程序员需要自己设置异常处理函数。

(2) 回卷：程序执行到某个地方出现异常错误时，系统会寻找出错点是否处于一个 `try{}` 块中，并进入 `try` 块提供的异常处理代码。如果当前 `try` 块没有提供异常处理，则会向更外一层的 `try` 块，寻找异常处理代码。直到最外层 `try{}` 块也没有提供异常处理代码，则交由操作系统处理。

这种向更外一层寻找异常处理的机制，被称为回卷。一般处理异常，是通过 `try-except` 块来处理的。

```
#001  __try
#002  {
#003  }
#004  __except(filter_value)
#005  {
#006  }
```

在被 `__try{}` 包围的块中，如果出现异常，会根据 `filter_value` 的数值，判断是否需要在 `__except{}` 块中处理。`filter_value` 的数值会有三种可能。

(1) `EXCEPTION_EXECUTE_HANDLER`，该数值为 1。进入到 `__except` 进行错误处理，处理完后不再回到 `__try{}` 块中，转而继续执行。

(2) `EXCEPTION_CONTINUE_SEARCH`，该数值为 0。不使用 `__except` 块中的异常处理，转而向上一层回卷。如果已经是最外层，则向操作系统请求异常处理函数。

(3) `EXCEPTION_CONTINUE_EXECUTION`，该数值为 -1。重复先前错误的指令，这个在驱动程序中很少用到。

`ProbeForRead` 和 `ProbeForWrite` 函数可以和 `try-except` 块配合，用来检查某段内存是否可读写。下面给出一段示例，这段代码探测空指针的地址是否可以写。这会引发一个异常，程序用 `try-except` 处理异常。

```
#001  VOID ProbeTest()
#002  {
#003      PVOID badPointer = NULL;
#004
#005      KdPrint(("Enter ProbeTest\n"));
#006
#007      __try
#008      {
#009          KdPrint(("Enter __try block\n"));
#010
#011          //判断空指针是否可读，显然会导致异常
#012          ProbeForWrite(badPointer, 100, 4);
#013
#014          //由于在上面引发异常，所以以后语句不会被执行！
#015          KdPrint(("Leave __try block\n"));
#016      }
#017      __except(EXCEPTION_EXECUTE_HANDLER)
#018      {
#019          KdPrint(("Catch the exception\n"));
```



```
#020         KdPrint(("The program will keep going\n"));
#021     }
#022
#023     //该语句会被执行
#024     KdPrint(("Leave ProbeTest\n"));
#025 }
```

此段代码可以在配套光盘中本章的 SEHTest 目录下找到。

除了读写内存外，try-except 块还可以处理一些异常。DDK 提供了一些函数触发异常，读者可以根据需要使用这些函数，如表 5-3 所示。

表 5-3 触发异常函数

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

5.6.5 结构化异常处理（try-finally 块）

结构化异常处理还有另外一种使用方法，就是利用 try-finally 块，强迫函数在退出前执行一段代码。

```
#001 NTSTATUS TryFinallyTest()
#002 {
#003     NTSTATUS status = STATUS_SUCCESS;
#004     __try
#005     {
#006         //做一些事情
#007         return STATUS_SUCCESS;
#008     }
#009     __finally
#010     {
#011         //程序退出前必然运行到此
#012         KdPrint(("Enter finally block\n"));
#013     }
#014 }
```

此段代码可以在配套光盘中本章的 SEHTest 目录下找到。

上面代码的__try{}块中，无论运行什么代码（即使是 return 语句或者触发异常），在程序退出前都会运行__finally{}块中的代码。这样的目的是，在退出前需要运行一些资源回收的工作，而资源回收代码的最佳位置就是放在这个块中。

除此之外，使用 try-finally 块还可以某种程度上简化代码。比较下面两段代码，其中第一个是没有使用 try-finally 块的代码，而第二段代码是使用 try-finally。可以看出，第二段代码比第一段代码简化。

```
#001 void FooTest()
#002 {
#003     NTSTATUS status = STATUS_SUCCESS;
#004 }
```

```

#005    //执行操作 1
#006    status = Foo1(...);
#007    //判断操作是否成功
#008    if (!NT_SUCCESS(status))
#009    {
#010        //回收资源
#011        return status;
#012    }
#013
#014    //执行操作 2
#015    status = Foo2(...);
#016    //判断操作是否成功
#017    if (!NT_SUCCESS(status))
#018    {
#019        //回收资源
#020        return status;
#021    }
#022
#023    //执行操作 n
#024    status = FooN(...);
#025    //判断操作是否成功
#026    if (!NT_SUCCESS(status))
#027    {
#028        //回收资源
#029        return status;
#030    }
#031
#032    //返回状态, 此状态一般是 STATUS_SUCCESS
#033    return status;
#034 }

```

此段代码可以在配套光盘中本章的 SEHTest 目录下找到。

以下是使用 try-finally 块的代码, 实现同样的功能, 但可以让代码简化。

```

#001 void FooTest()
#002 {
#003     NTSTATUS status = STATUS_SUCCESS;
#004
#005     __try
#006     {
#007         //执行操作 1
#008         status = Foo1(...);
#009         //判断操作是否成功
#010         if (!NT_SUCCESS(status))
#011         {
#012             return status;
#013         }
#014
#015         //执行操作 2
#016         status = Foo2(...);
#017         //判断操作是否成功
#018         if (!NT_SUCCESS(status))
#019         {
#020             return status;
#021         }
#022

```



```
#023         //执行操作 n
#024         status = FooN(...);
#025         //判断操作是否成功
#026         if (!NT_SUCCESS(status))
#027         {
#028             return status;
#029         }
#030
#031     }
#032     __finally
#033     {
#034         if (!NT_SUCCESS(status))
#035         {
#036             ///回收资源
#037         }
#038         return status;
#039     }
#040 }
```

此段代码可以在配套光盘中本章的 SEHTest 目录下找到。

5.6.6 使用宏需要注意的地方

DDK 提供了大量的宏。在使用这些宏的时候，要注意一种错误的发生，这就是“侧效”（Side Effect）。

宏一般由多行组成，如下面的形式，其中的“\”代表换行。

```
#001 #define PRINT(msg) KdPrint(("=====\n")); \
#002 KdPrint(msg); \
#003 KdPrint(("=====\n"));
```

在 C 语言中规定，for 或者 if 语句块中的内容如果只是一句，可以省略掉{}。如：

```
#001 if(bRet)
#002 {
#003     Foo();
#004 }
```

等价于

```
if(bRet)
    Foo();
```

但如果 Foo 是宏而非函数时，就会产生逻辑错误。如：

```
#001 if(bRet)
#002     PRINT (msg);
```

等价于

```
#001 if(bRet)
#002     PRINT(msg) KdPrint(("=====\n")); \
#003     KdPrint(msg); \
#004     KdPrint(("=====\n"));
```

这明显和我们开始的想法不一致。产生这样的错误，称为“侧效”错误。“侧效”错

误很难被发现，因为程序员很难判断 Foo 是函数还是宏。

解决这个问题，一般有两个办法。

(1) 对于 if、while、for 这样的语句，不省略{}。这样是最保险的做法，能完全保证不出现“侧效”错误。

(2) 在编写多行宏的时候，在宏的前后加上{}。如：

```
#001 #define PRINT(msg) {\
#002 KdPrint(("=====\n")); \
#003 KdPrint(msg); \
#004 KdPrint(("=====\n")); \
#005 }
```

这种方法能保证这个宏在调用时，不出现“侧效”错误。

5.6.7 断言

在驱动程序开发中，还有一个技巧，就是使用“断言”。在驱动程序使用“断言”，一般是通过使用 ASSERT 宏。例如：

```
#001 NTSTATUS Foo(PCHAR* str)
#002 {
#003     ASSERT(str!=NULL); //断言
#004     //对 str 的操作
#005 }
```

这段代码认为输入参数绝不可能是空指针，因此在函数的开头做一个断言(ASSERT)。一旦断言失败，会引发一个异常。

5.7 小结

本章围绕着驱动程序中的内存操作进行了介绍。在驱动程序开发中，首先要注意分页内存和非分页内存的使用。同时，还需要区分物理内存地址和虚拟内存地址这两个概念。

在驱动程序开发中，还会经常使用单向链表和双向链表等数据结构，本章对这些数据结构的使用进行了介绍。另外，在驱动程序开发中，内存复制、内存搬移、内存填充都和应用程序有所区别，要使用 DDK 提供专用的内核函数，而不能使用 C 语言提供的运行时函数。

第 6 章 Windows 内核函数

本章介绍了 Windows 内核中字符串处理函数、文件读写函数、注册表读写函数。这些函数是 DDK 提供的运行时函数，它们比标准 C 语言的运行时函数功能更丰富。普通的 C 语言运行时库是不能在内核模式下使用的，必须使用 DDK 提供的运行时函数。

6.1 内核模式下的字符串操作

和应用程序一样，驱动程序需要经常和字符串打交道。其中包括 ASCII 字符串、宽字符串，还有 DDK 定义的 ANSI_STRING 数据结构和 UNICODE_STRING 数据结构。

6.1.1 ASCII 字符串和宽字符串

在应用程序中，往往使用两种字符：一种是 char 型的字符串，负责记录 ANSI 字符集，它是指向一个 char 数组的指针，每个 char 型变量的大小为一个字节，字符串是以 0 标志字符串结束。还有一种是 wchar_t 型的宽字符串，负责描述 unicode 字符集的字符串，它是指向一个 wchar_t 数组的指针，wchar_t 字符大小为两个字节，字符串以 0 标志字符串结束。ANSI 字符的构造如下：

```
char* str1 = "abc";
```

str1 指针指向的内容是 61 62 63 00。

UNICODE 字符的构造如下：

```
wchar_t *str2 = L"abc";
```

str2 指针指向的内容是 6100 6200 6300 0000。在构造字符串的时候使用一个关键字“L”。编译器会自动生成所需要的宽字符。

在驱动程序开发中，DDK 将 char 和 wchar_t 类别，替换成 CHAR 和 WCHAR 类别。

对于这两类的字符串, DDK 提供相应的字符串操作函数, 例如, `strcpy`、`sprintf`、`strcat`、`strlen` 等。但 DDK 的帮助文档中, 不会查到这些函数的使用方法。微软公司不鼓励直接使用这些函数, 取而代之的是使用同样功能的宏。读者可以用 `Depends` 工具查看 `NTOSKRNL.EXE` 导出的函数, 如图 6-1 所示。

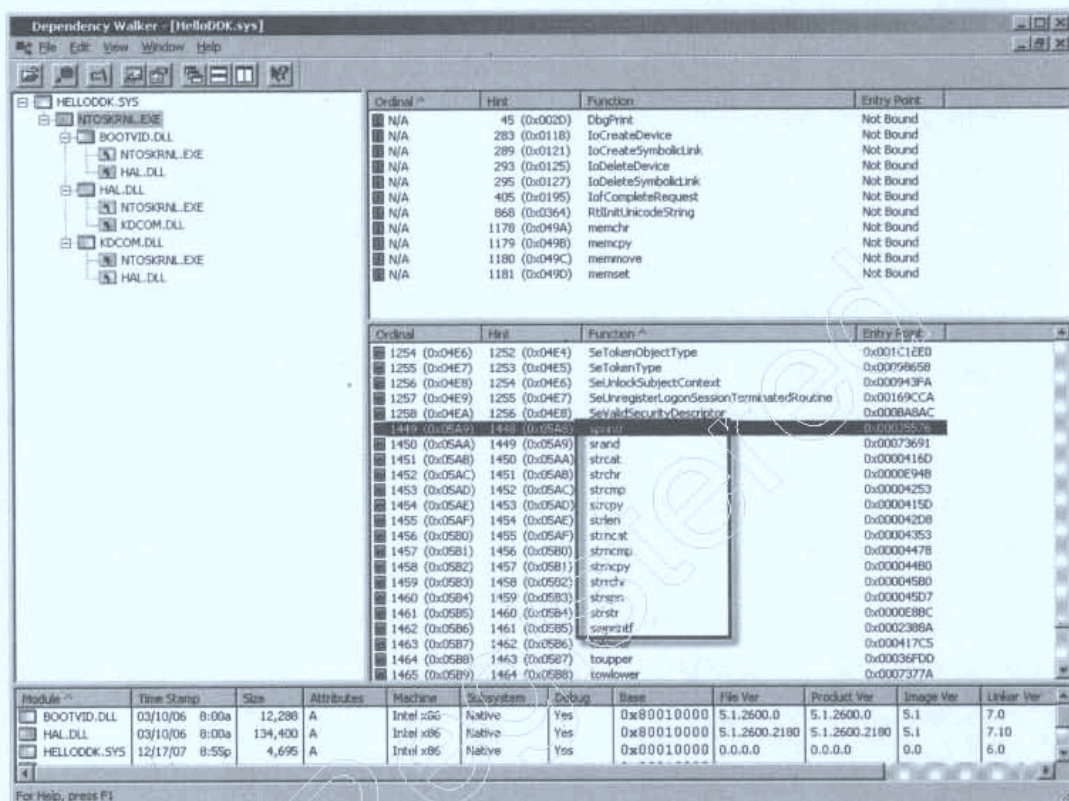


图 6-1 Depends 查看 NTOSKRNL.EXE 导出表

驱动程序可以用 `KdPrint` 打印 ASCII 字符串和宽字符串。`KdPrint` 类似于 C 语言的 `printf` 函数。例如, 打印一个 ASCII 字符串:

```
CHAR *string = "Hello";
KdPrint("%s\n", string); //注意是小写%s
```

打印一段宽字符串时需要:

```
WCHAR *string = L"Hello";
KdPrint("%S\n", string); //注意是大写%S
```

6.1.2 ANSI_STRING 字符串与 UNICODE_STRING 字符串

DDK 不鼓励程序员使用 C 语言的字符串, 主要是因为: 标准 C 的字符串处理函数容易导致缓冲区溢出等错误。如果程序员不对字符串的长度进行检验, 很容易导致这个错误, 从而导致整个操作系统的崩溃。DDK 鼓励程序员使用 DDK 自定义的字符串, 这种数据格

式的定义如下:

```
typedef struct _STRING {
    USHORT Length;
    USHORT MaximumLength;
    PCHAR Buffer;
} STRING;
typedef STRING ANSI_STRING;
typedef PSTRING PANSI_STRING;

typedef STRING OEM_STRING;
typedef PSTRING POEM_STRING;
```

这个数据结构对 ASCII 字符串进行了封装。

- Length: 字符的长度。
- MaximumLength: 整个字符串缓冲区的最大长度。
- Buffer: 缓冲区的指针。

注意: 和标准的字符串不同, STRING 字符串不是以 0 标志字符的结束。字符长度依靠 Length 字段。在标准 C 中的字符串中, 如果缓冲区长度是 N, 那么只能容纳 N-1 个字符的字符串, 这是因为要留一个字节存储 NULL。而在 STRING 字符串中, 缓冲区的大小 MaximumLength, 最大的字符串长度可以是 MaximumLength, 而不是 MaximumLength-1。

与 ANSI_STRING 相对应, DDK 将宽字符串封装成 UNICODE_STRING 数据结构。

```
typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR Buffer;
} UNICODE_STRING;
```

- Length: 字符的长度, 单位是字节。如果是 N 个字符, 那么 Length 等于 N 的 2 倍。
- MaximumLength: 整个字符串缓冲区的最大长度, 单位也是字节。
- Buffer: 缓冲区的指针。

和 ANSI_STRING 不同, UNICODE_STRING 的缓冲区是记录宽字符的缓冲区。每个元素是宽字符。和 ANSI_STRING 一样, 字符串的结束不是以 NULL 为标志, 而是依靠字段 Length。

关于 ANSI_STRING 字符串和 UNICODE_STRING 字符串, KdPrint 同样提供了打印 log 的方法。

```
ANSI_STRING ansiString;
//省去对 ansiString 初始化
KdPrint("%Z\n", &ansiString); //注意是%Z
```

而当打印一段宽字符的时候, 需要进行以下操作。

```
UNICODE_STRING uniString;
//省去对 uniString 初始化
KdPrint("%wZ\n", & uniString); //注意是%wZ
```

6.1.3 字符初始化与销毁

ANSI_STRING 字符串和 UNICODE_STRING 字符串使用前需要进行初始化。有两种办法构造这个数据结构。

(1) 方法一是使用 DDK 提供了相应的函数。

初始化 ANSI_STRING 字符串：

```
VOID
RtlInitAnsiString(
    IN OUT PANSI_STRING DestinationString,
    IN PCSZ SourceString
);
```

➤ DestinationString: 要初始化的 ANSI_STRING 字符串。

➤ SourceString: 字符串的内容。

初始化 UNICODE_STRING 字符串：

```
VOID
RtlInitUnicodeString(
    IN OUT PUNICODE_STRING DestinationString,
    IN PCWSTR SourceString
);
```

➤ DestinationString: 要初始化的 UNICODE_STRING 字符串。

➤ SourceString: 字符串的内容。

以 RtlInitAnsiString 为例，其使用方法是：

```
#001 ANSI_STRING AnsiString1;
#002 CHAR * string1= "hello";
#003 RtlInitAnsiString(&AnsiString1,string1);
```

这种办法是将 AnsiString1 中的 Buffer 指针等于 string1 指针。这种初始化的优点是操作简单，用完后不用清理内存。但带来另外一个问题，如果修改 string1，同时会导致 AnsiString1 字符发生变化。

看下面一段代码：

```
#001 ANSI_STRING AnsiString1;
#002 CHAR * string1= "hello";
#003 //初始化 ANSI_STRING 字符串
#004 RtlInitAnsiString(&AnsiString1,string1);
#005 KdPrint(("AnsiString1:%Z\n",&AnsiString1)); //打印 hello
#006 //改变 string1
#007 string1[0]='H';
#008 string1[1]='E';
#009 string1[2]='L';
#010 string1[3]='L';
#011 string1[4]='O';
#012 //改变 string1, AnsiString1 同样会导致变化
#013 KdPrint(("AnsiString1:%Z\n",&AnsiString1)); //打印 HELLO
```

此段代码可以在配套光盘中本章的 StringTest 目录下找到。

(2) 另外一种方法是程序员自己申请内存，并初始化内存，当不用字符串时，需要回收字符串占用的内存。

```
#001 #define BUFFER_SIZE 1024
#002 UNICODE_STRING UnicodeString1 = {0};
#003 //设置缓冲区大小
#004 UnicodeString1.MaximumLength = BUFFER_SIZE;
#005 //分配内存
#006 UnicodeString1.Buffer = (PWSTR)ExAllocatePool(PagedPool, BUFFER_SIZE);
#007 WCHAR* wideString = L"hello";
#008
#009 //设置字符串长度,因为是宽字符,所以是字符串长度的2倍
#010 UnicodeString1.Length = 2*wcslen(wideString);
#011
#012 //保证缓冲区足够大,否则程序终止
#013 ASSERT(UnicodeString1.MaximumLength>=UnicodeString1.Length);
#014 //内存复制
#015 RtlCopyMemory(UnicodeString1.Buffer, wideString, UnicodeString1.Length);
#016 //设置字符串长度
#017 UnicodeString1.Length = 2*wcslen(wideString);
#018
#019 KdPrint(("UnicodeString:%wZ\\n", &UnicodeString1));
#020
#021 //清理内存
#022 ExFreePool(UnicodeString1.Buffer);
#023 UnicodeString1.Buffer = NULL;
#024 UnicodeString1.Length = UnicodeString1.MaximumLength = 0;
```

此段代码可以在配套光盘中本章的 StringTest 目录下找到。

对于最后一步清理内存，DDK 同样给出了简化函数，分别是 `RtlFreeAnsiString` 和 `RtlFreeUnicodeString`。这两个函数内部调用了 `ExFreePool` 去回收内存。

6.1.4 字符串复制

DDK 提供针对 `ANSI_STRING` 字符串和 `UNICODE_STRING` 字符串的复制字符串命令，分别是：

ANSI_STRING 字符串复制函数

```
VOID
RtlCopyString(
    IN OUT PSTRING DestinationString,
    IN PSTRING SourceString OPTIONAL
);
```

- `DestinationString`: 目的字符串。
- `SourceString`: 源字符串。

UNICODE_STRING 字符串复制函数

```
VOID
RtlCopyUnicodeString(
    IN OUT PUNICODE_STRING DestinationString,
```

```
IN PUNICODE_STRING SourceString
};
```

- DestinationString: 目的字符串。
- SourceString: 源字符串。

下面的代码演示了如何使用 RtlCopyUnicodeString 函数, 与 RtlCopyString 函数的使用方法是类似的。

```
#001 //初始化 UnicodeString1
#002 UNICODE_STRING UnicodeString1;
#003 RtlInitUnicodeString(&UnicodeString1,L"Hello World");
#004
#005 //初始化 UnicodeString2
#006 UNICODE_STRING UnicodeString2={0};
#007 UnicodeString2.Buffer = (PWSTR)ExAllocatePool(PagedPool,BUFFER_SIZE);
#008 UnicodeString2.MaximumLength = BUFFER_SIZE;
#009
#010 //将初始化 UnicodeString2 复制到 UnicodeString1
#011 RtlCopyUnicodeString(&UnicodeString2,&UnicodeString1);
#012
#013 //分别显示 UnicodeString1 和 UnicodeString2
#014 KdPrint(("UnicodeString1:%wZ\n",&UnicodeString1));
#015 KdPrint(("UnicodeString2:%wZ\n",&UnicodeString2));
#016
#017 //销毁 UnicodeString2
#018 //注意!!UnicodeString1 不用销毁
#019 RtlFreeUnicodeString(&UnicodeString2);
```

此段代码可以在配套光盘中本章的 StringTest 目录下找到。

6.1.5 字符串比较

DDK 提供了对 ANSI_STRING 字符串和 UNICODE_STRING 字符串的相关字符串比较的命令, 分别是: ANSI_STRING 字符串比较函数和 UNICODE_STRING 字符串比较函数。

```
LONG
RtlCompareString(
    IN PSTRING String1,
    IN PSTRING String2,
    BOOLEAN CaseInsensitive
);
```

- String1: 要比较的第一个字符串。
- String2: 要比较的第二个字符串。
- CaseInsensitive: 是否对大小写敏感。
- 返回值: 比较结果。

```
LONG
RtlCompareUnicodeString(
    IN PUNICODE_STRING String1,
```



```
IN PUNICODE_STRING String2,  
IN BOOLEAN CaseInsensitive  
);
```

- String1: 要比较的第一个字符串。
- String2: 要比较的第二个字符串。
- CaseInsensitive: 是否对大小写敏感。
- 返回值: 比较结果。

这两个函数的参数形式相同, 以 `RtlCompareUnicodeString` 为例, 前两个参数分别是需要比较的字符串, 第三个参数指定是否对大小写敏感。如果函数返回值为 0, 表示两个字符串相等。如果小于零, 则表示第一个字符串小于第二个字符串。反之, 如果大于零, 则代表第一个字符串大于第二个字符串。

同时, DDK 又提供了 `RtlEqualString` 和 `RtlEqualUnicodeString` 函数, 其使用方法和上面两个函数类似。只是返回为非零代表相等, 零代表不相等。

下面的代码演示了如何使用 `RtlCompareUnicodeString` 函数。

```
#001 //初始化 UnicodeString1  
#002 UNICODE_STRING UnicodeString1;  
#003 RtlInitUnicodeString(&UnicodeString1,L"Hello World");  
#004  
#005 //初始化 UnicodeString2  
#006 UNICODE_STRING UnicodeString2;  
#007 RtlInitUnicodeString(&UnicodeString2,L"Hello");  
#008 //判断字符串是否相等  
#009 if (RtlEqualUnicodeString(&UnicodeString1,&UnicodeString2,TRUE))  
#010 {  
#011     KdPrint(("UnicodeString1 and UnicodeString2 are equal\n"));  
#012 }else  
#013 {  
#014     KdPrint(("UnicodeString1 and UnicodeString2 are NOT equal\n"));  
#015 }
```

此段代码可以在配套光盘中本章的 `StringTest` 目录下找到。

6.1.6 字符串转化成大写

DDK 提供了对 `ANSI_STRING` 字符串和 `UNICODE_STRING` 字符串的相关字符串大小写转化的函数,

(1) `ANSI_STRING` 字符串转化成大写。

```
VOID  
RtlUpperString(  
    IN OUT PSTRING DestinationString,  
    IN PSTRING SourceString  
);
```

- DestinationString: 目的字符串。
- SourceString: 源字符串。

(2) UNICODE_STRING 字符串转化成大写。

```

NTSTATUS
RtlUpcaseUnicodeString(
    IN OUT PUNICODE_STRING DestinationString OPTIONAL,
    IN PCUNICODE_STRING SourceString,
    IN BOOLEAN AllocateDestinationString
);

```

- DestinationString: 目的字符串。
- SourceString: 源字符串。
- AllocateDestinationString: 是否为目的字符串分配内存。
- 返回值: 返回转换是否成功。

RtlUpcaseUnicodeString 函数比 RtlUpperString 函数多一个参数 AllocateDestinationString。这个参数指定是否为目的字符串申请内存。目的字符串和源字符串可以是同一个字符串。

DDK 虽然提供了转化成大写的函数, 但却没有提供转化成小写的函数。下面的代码演示了如何使用 RtlUpcaseUnicodeString 函数。

```

#001 //初始化 UnicodeString1
#002 UNICODE_STRING UnicodeString1;
#003 RtlInitUnicodeString(&UnicodeString1,L"Hello World");
#004 //变化前
#005 KdPrint(("UnicodeString1:%wZ\n",&UnicodeString1));
#006 //转化成大写
#007 RtlUpcaseUnicodeString(&UnicodeString1,&UnicodeString1,FALSE);
#008 //变化后
#009 KdPrint(("UnicodeString1:%wZ\n",&UnicodeString1));

```

此段代码可以在配套光盘本章的 StringTest 目录下找到。

6.1.7 字符串与整型数字相互转换

DDK 提供了 UNICODE_STRING 字符串与整数相互转换的内核函数。

(1) 将 UNICODE_STRING 字符串转换成整数。

这个函数是 RtlUnicodeStringToInteger, 其声明是:

```

NTSTATUS
RtlUnicodeStringToInteger(
    IN PUNICODE_STRING String,
    IN ULONG Base OPTIONAL,
    OUT PULONG Value
);

```

- String: 需要转换的字符串。
- Base: 转换的数的进制 (如 2、8、10、16)。
- Value: 需要转换的数字。
- 返回值: 指明是否转换成功。

(2) 将整数转换成 UNICODE_STRING 字符串。

这个函数 `RtlIntegerToUnicodeString`，其声明是：

```
NTSTATUS  
RtlIntegerToUnicodeString(  
    IN ULONG Value,  
    IN ULONG Base OPTIONAL,  
    IN OUT PUNICODE_STRING String  
);
```

- Value: 需要转换的数字。
- Base: 转换的数的进制 (如 2、8、10、16)。
- String: 需要转换的字符串。
- 返回值: 指明是否转换成功。

以下是字符串与整型数字相互转换的实例。

```
#001 // (1) 字符串转换成数字  
#002 // 初始化 UnicodeString1  
#003 UNICODE_STRING UnicodeString1;  
#004 RtlInitUnicodeString(&UnicodeString1, L"-100");  
#005  
#006 ULONG lNumber;  
#007 NTSTATUS nStatus = RtlUnicodeStringToInteger(&UnicodeString1, 10,  
&lNumber);  
#008 if ( NT_SUCCESS(nStatus))  
#009 {  
#010     KdPrint(("Conver to integer succussfully!\n"));  
#011     KdPrint(("Result:%d\n", lNumber));  
#012 }else  
#013 {  
#014     KdPrint(("Conver to integer unsuccessfully!\n"));  
#015 }  
#016  
#017 // (2) 数字转换成字符串  
#018 // 初始化 UnicodeString2  
#019 UNICODE_STRING UnicodeString2={0};  
#020 UnicodeString2.Buffer = (PWSTR)ExAllocatePool(PagedPool, BUFFER_SIZE);  
#021 UnicodeString2.MaximumLength = BUFFER_SIZE;  
#022 nStatus = RtlIntegerToUnicodeString(200, 10, &UnicodeString2);  
#023  
#024 if ( NT_SUCCESS(nStatus))  
#025 {  
#026     KdPrint(("Conver to string succussfully!\n"));  
#027     KdPrint(("Result:%wZ\n", &UnicodeString2));  
#028 }else  
#029 {  
#030     KdPrint(("Conver to string unsuccessfully!\n"));  
#031 }  
#032  
#033 // 销毁 UnicodeString2  
#034 // 注意: UnicodeString1 不用销毁  
#035 RtlFreeUnicodeString(&UnicodeString2);
```

此段代码可以在配套光盘中本章的 `StringTest` 目录下找到。

6.1.8 ANSI_STRING 字符串与 UNICODE_STRING 字符串相互转换

DDK 提供了 ANSI_STRING 字符串与 UNICODE_STRING 字符串相互转换的相关函数。

(1) 将 UNICODE_STRING 字符串转换成 ANSI_STRING 字符串。

DDK 对于这种转换提供的函数是 RtlUnicodeStringToAnsiString，其声明是：

```
NTSTATUS
RtlUnicodeStringToAnsiString(
    IN OUT PANSI_STRING DestinationString,
    IN PUNICODE_STRING SourceString,
    IN BOOLEAN AllocateDestinationString
);
```

- DestinationString: 需要被转换的字符串。
- SourceString: 需要转换的源字符串。
- AllocateDestinationString: 是否需要对被转换的字符串分配内存。
- 返回值: 指明是否转换成功。

(2) 将 ANSI_STRING 字符串转换成 UNICODE_STRING 字符串。

DDK 对于这种转换提供的函数是 RtlAnsiStringToUnicodeString，其声明是：

```
NTSTATUS
RtlAnsiStringToUnicodeString(
    IN OUT PUNICODE_STRING DestinationString,
    IN PANSI_STRING SourceString,
    IN BOOLEAN AllocateDestinationString
);
```

- DestinationString: 需要被转换的字符串。
- SourceString: 需要转换的源字符串。
- AllocateDestinationString: 是否需要对被转换的字符串分配内存。
- 返回值: 指明是否转换成功。

以下的代码演示了如何实现 ANSI_STRING 字符串与 UNICODE_STRING 字符串的相互转换。

```
#001 // (1) 将 UNICODE_STRING 字符串转换成 ANSI_STRING 字符串
#002 // 初始化 UnicodeString1
#003 UNICODE_STRING UnicodeString1;
#004 RtlInitUnicodeString(&UnicodeString1, L"Hello World");
#005
#006 ANSI_STRING AnsiString1;
#007 NTSTATUS nStatus = RtlUnicodeStringToAnsiString(&AnsiString1, &UnicodeString1, TRUE);
#008
#009 if ( NT_SUCCESS(nStatus))
#010 {
#011     KdPrint(("Conver succussfully!\n"));
#012     KdPrint(("Result:%Z\n", &AnsiString1));
```



```
#013     }else
#014     {
#015         KdPrint(("Conver unsuccessfully!\n"));
#016     }
#017     //销毁 AnsiString1
#018     RtlFreeAnsiString(&AnsiString1);
#019     //(2) 将 ANSI_STRING 字符串转换成 UNICODE_STRING 字符串
#020     //初始化 AnsiString2
#021     ANSI_STRING AnsiString2;
#022     RtlInitString(&AnsiString2, "Hello World");
#023
#024     UNICODE_STRING UnicodeString2;
#025     nStatus = RtlAnsiStringToUnicodeString(&UnicodeString2, &AnsiString2, TRUE);
#026
#027     if ( NT_SUCCESS(nStatus))
#028     {
#029         KdPrint(("Conver succussfully!\n"));
#030         KdPrint(("Result:%wZ\n", &UnicodeString2));
#031     }else
#032     {
#033         KdPrint(("Conver unsuccessfully!\n"));
#034     }
#035
#036     //销毁 UnicodeString2
#037     RtlFreeUnicodeString(&UnicodeString2);
```

此段代码可以在配套光盘中本章的 StringTest 目录下找到。

6.2 内核模式下的文件操作

在驱动程序开发中，经常会对文件进行操作。与 Win32API 不同，DDK 提供另外一套对文件的操作函数。

6.2.1 文件的创建

对文件的创建或者打开都是通过内核函数 ZwCreateFile 实现的。和 Windows API 类似，这个内核函数返回一个文件句柄，文件的所有操作都是依靠这个句柄进行操作的。在文件操作完毕后，需要关闭这个句柄。

```
NTSTATUS
ZwCreateFile(
    OUT PHANDLE FileHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    IN PLARGE_INTEGER AllocationSize OPTIONAL,
    IN ULONG FileAttributes,
    IN ULONG ShareAccess,
    IN ULONG CreateDisposition,
    IN ULONG CreateOptions,
    IN PVOID EaBuffer OPTIONAL,
    IN ULONG EaLength
);
```

- **FileHandle**: 返回打开文件的句柄。
- **DesiredAccess**: 对打开文件操作的描述, 读、写或是其他。一般指定为 `GENERIC_READ` 或 `GENERIC_WRITE`。
- **ObjectAttributes**: 是 `OBJECT_ATTRIBUTES` 结构的地址, 该结构包含要打开的文件名。
- **IoStatusBlock**: 指向一个 `IO_STATUS_BLOCK` 结构, 该结构接收 `ZwCreateFile` 操作的结果状态。
- **AllocationSize**: 是一个指针, 指向一个 64 位整数, 该数指定文件初始分配时的大小。该参数仅关系到创建或重写文件操作, 如果忽略它 (如笔者在这里所做的), 那么文件长度将从 0 开始, 并随着写入而增长。
- **FileAttributes**: 0 或 `FILE_ATTRIBUTE_NORMAL`, 指定新创建文件的属性。
- **ShareAccess**: `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, 则可以与其他线程同时读取该文件。如果为写数据而打开文件, 可能不希望其他线程访问该文件。
- **CreateDisposition**: `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。
- **CreateOptions**: `FILE_SYNCHRONOUS_IO_NONALERT`, 指定控制打开操作和句柄使用的附加标志位。
- **EaBuffer**: 一个指针, 指向可选的扩展属性区。
- **EaLength**: 扩展属性区的长度。

这个函数需要填写 `CreateDisposition` 参数。如果想打开文件, `CreateDisposition` 参数设置成 `FILE_OPEN`。如果想创建文件, `CreateDisposition` 参数设置成 `FILE_OVERWRITE_IF`。此时, 无论文件是否存在, 都会创建新文件。

文件名的指定是通过设定第三个参数 `ObjectAttributes`。这个参数是一个 `OBJECT_ATTRIBUTES` 结构。DDK 提供对 `OBJECT_ATTRIBUTES` 结构初始化的宏 `InitializeObjectAttributes`。

```
VOID
InitializeObjectAttributes(
    OUT POBJECT_ATTRIBUTES InitialzedAttributes,
    IN PUNICODE_STRING ObjectName,
    IN ULONG Attributes,
    IN HANDLE RootDirectory,
    IN PSECURITY_DESCRIPTOR SecurityDescriptor
);
```

- **InitialzedAttributes**: 返回的 `OBJECT_ATTRIBUTES` 结构。
- **ObjectName**: 对象名称, 用 `UNICODE_STRING` 描述, 这里设置的是文件名。
- **Attributes**: 一般设为 `OBJ_CASE_INSENSITIVE`, 对大小写敏感。

Windows 驱动开发技术详解

- RootDirectory: 一般设为 NULL。
- SecurityDescriptor: 一般设为 NULL。

另外, 文件名必须是符号链接或者是设备名。符号链接的概念, 已经在第 4 章中介绍过。例如, 盘符“c:”就是一个符号链接。这里应该用“\\?\\c:”代替, “c:\\1.log”要写成“\\?\\c:\\1.log”。

其中, “\\?\\c:”是符号链接, 内核会将它转换成设备名“\\Device\\HarddiskVolume1”。下面的代码演示了如何在驱动程序中创建文件和打开文件。

(1) 创建文件:

```
#001 OBJECT_ATTRIBUTES objectAttributes;  
#002 IO_STATUS_BLOCK iostatus;  
#003 HANDLE hfile;  
#004 UNICODE_STRING logFileUnicodeString;  
#005  
#006 //初始化 UNICODE_STRING 字符串  
#007 RtlInitUnicodeString( &logFileUnicodeString,  
#008     L"\\?\\c:\\1.log");  
#009 //或者写成 "\\Device\\HarddiskVolume1\\1.LOG"  
#010  
#011 //初始化 objectAttributes  
#012 InitializeObjectAttributes(&objectAttributes,  
#013     &logFileUnicodeString,  
#014     OBJ_CASE_INSENSITIVE,  
#015     NULL,  
#016     NULL );  
#017  
#018 //创建文件  
#019 NTSTATUS ntStatus = ZwCreateFile( &hfile,  
#020     GENERIC_WRITE,  
#021     &objectAttributes,  
#022     &iostatus,  
#023     NULL,  
#024     FILE_ATTRIBUTE_NORMAL,  
#025     FILE_SHARE_READ,  
#026     FILE_OPEN_IF,  
#027     FILE_SYNCHRONOUS_IO_NONALERT,  
#028     NULL,  
#029     0 );  
#030 if ( NT_SUCCESS(ntStatus))  
#031 {  
#032     KdPrint(("Create file succussfully!\\n"));  
#033 }else  
#034 {  
#035     KdPrint(("Create file unsuccessfully!\\n"));  
#036 }  
#037  
#038 //文件操作  
#039 //.....  
#040  
#041 //关闭文件句柄  
#042 ZwClose(hfile);
```

此段代码可以在配套光盘中本章的 FileTest 目录下找到。

(2) 打开文件:

```
#001  OBJECT_ATTRIBUTES objectAttributes;
#002  IO_STATUS_BLOCK iostatus;
#003  HANDLE hfile;
#004  UNICODE_STRING logFileUnicodeString;
#005
#006  //初始化 UNICODE_STRING 字符串
#007  RtlInitUnicodeString( &logFileUnicodeString,
#008      L"\\??\\C:\\1.log");
#009  //或者写成 "\\Device\\HarddiskVolume1\\1.LOG"
#010
#011  //初始化 objectAttributes
#012  InitializeObjectAttributes(&objectAttributes,
#013      &logFileUnicodeString,
#014      OBJ_CASE_INSENSITIVE, //对大小写敏感
#015      NULL,
#016      NULL );
#017
#018  //创建文件
#019  NTSTATUS ntStatus = ZwCreateFile( &hfile,
#020      GENERIC_READ,
#021      &objectAttributes,
#022      &iostatus,
#023      NULL,
#024      FILE_ATTRIBUTE_NORMAL,
#025      FILE_SHARE_READ,
#026      FILE_OPEN, //打开文件, 如果不存在, 则返回错误
#027      FILE_SYNCHRONOUS_IO_NONALERT,
#028      NULL,
#029      0 );
#030  if ( NT_SUCCESS(ntStatus))
#031  {
#032      KdPrint(("Open file succussfully!\\n"));
#033  }else
#034  {
#035      KdPrint(("Open file unsuccessfully!\\n"));
#036  }
#037
#038  //文件操作
#039  //.....
#040
#041  //关闭文件句柄
#042  ZwClose(hfile);
```

此段代码可以在配套光盘中本章的 FileTest 目录下找到。

读者可以观察打开文件和创建文件的不同操作方法。需要注意的是, 文件被成功打开后, 一定要记得关闭句柄, 关闭句柄使用 ZwClose 内核函数。

6.2.2 文件的打开

除了使用 ZwCreateFile 函数可以打开文件, DDK 还提供了一个内核函数 ZwOpenFile。ZwOpenFile 内核函数的参数比 ZwCreateFile 的参数简化, 方便程序员打开文件。该函数的

Windows 驱动开发技术详解

声明如下:

```
NTSTATUS
ZwOpenFile(
    OUT PHANDLE FileHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    IN ULONG ShareAccess,
    IN ULONG OpenOptions
);
```

- FileHandle: 返回打开的文件句柄。
- DesiredAccess: 打开的权限, 一般设为 GENERIC_ALL。
- ObjectAttributes: objectAttributes 结构。
- IoStatusBlock: 指向一个结构体的指针。该结构体指明打开文件的状态。
- ShareAccess: 共享的权限。可以是 FILE_SHARE_READ 或者 FILE_SHARE_WRITE。
- OpenOptions: 打开选项, 一般设为 FILE_SYNCHRONOUS_IO_NONALERT。
- 返回值: 指明文件是否被成功打开。

下面的代码演示了如何使用 ZwOpenFile 打开文件。

```
#001 OBJECT_ATTRIBUTES objectAttributes;
#002 IO_STATUS_BLOCK iostatus;
#003 HANDLE hfile;
#004 UNICODE_STRING logFileUnicodeString;
#005
#006 //初始化 UNICODE_STRING 字符串
#007 RtlInitUnicodeString( &logFileUnicodeString,
#008     L"\\??\\C:\\l.log");
#009 //或者写成 "\\Device\\HarddiskVolume1\\l.LOG"
#010
#011 //初始化 objectAttributes
#012 InitializeObjectAttributes(&objectAttributes,
#013     &logFileUnicodeString,
#014     OBJ_CASE_INSENSITIVE,
#015     NULL,
#016     NULL );
#017
#018 //创建文件
#019 NTSTATUS ntStatus = ZwOpenFile( &hfile,
#020     GENERIC_ALL,
#021     &objectAttributes,
#022     &iostatus,
#023     FILE_SHARE_READ|FILE_SHARE_WRITE,
#024     FILE_SYNCHRONOUS_IO_NONALERT);
#025 if ( NT_SUCCESS(ntStatus))
#026 {
#027     KdPrint(("Create file succussfully!\\n"));
#028 }else
#029 {
#030     KdPrint(("Create file unsuccessfully!\\n"));
#031 }
```

```
#032
#033    //文件操作
#034    //.....
#035
#036    //关闭文件句柄
#037    ZwClose(hfile);
```

此段代码可以在配套光盘中本章的 FileTest 目录下找到。

6.2.3 获取或修改文件属性

获取和修改文件属性，包括获取文件大小、获取或修改文件指针位置、获取或修改文件名、获取或修改文属性（只读属性、隐藏属性）、获取或修改文件创建、修改日期等。DDK 提供了内核函数 `ZwSetInformationFile` 和 `ZwQueryInformationFile` 函数来进行获取和修改文件属性。

```
NTSTATUS
ZwSetInformationFile(
    IN HANDLE FileHandle,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    IN PVOID FileInformation,
    IN ULONG Length,
    IN FILE_INFORMATION_CLASS FileInformationClass
);
```

- `FileHandle`: 文件句柄。
- `IoStatusBlock`: 返回设置的状态。
- `FileInformation`: 依据 `FileInformationClass` 不同而不同。作为输入信息。
- `Length`: `FileInformation` 数据的长度。
- `FileInformationClass`: 描述修改属性的类型。

```
NTSTATUS
ZwQueryInformationFile(
    IN HANDLE FileHandle,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    OUT PVOID FileInformation,
    IN ULONG Length,
    IN FILE_INFORMATION_CLASS FileInformationClass
);
```

- `FileHandle`: 文件句柄。
- `IoStatusBlock`: 返回设置的状态。
- `FileInformation`: 依据 `FileInformationClass` 不同而不同。作为输出信息。
- `Length`: `FileInformation` 数据的长度。
- `FileInformationClass`: 描述修改属性的类型。

这两个函数的参数基本相同。其中 `FileInformationClass` 指定修改或者查询的类别。

(1) 当 `FileInformationClass` 是 `FileStandardInformation` 时，输入和输出的数据是 `FILE_STANDARD_INFORMATION` 结构体，描述文件的基本信息。


```
typedef struct FILE_STANDARD_INFORMATION {
    LARGE_INTEGER AllocationSize;    //为文件分配的大小（注意这个不是文件大小，而是占用簇所
    需要的大小）
    LARGE_INTEGER EndOfFile;        //距离文件结尾还有多少字节
    ULONG NumberOfLinks;            //有多少个链接文件
    BOOLEAN DeletePending;          //是否准备删除
    BOOLEAN Directory;              //是否为目录
} FILE_STANDARD_INFORMATION, *PFILE_STANDARD_INFORMATION;
```

(2) 当 FileInformationClass 是 FileBasicInformation 时，输入和输出的数据是 FILE_BASIC_INFORMATION 结构体，描述文件的基本信息。

```
typedef struct FILE_BASIC_INFORMATION {
    LARGE_INTEGER CreationTime;    //文件创建时间
    LARGE_INTEGER LastAccessTime;  //最后访问时间
    LARGE_INTEGER LastWriteTime;   //最后写时间
    LARGE_INTEGER ChangeTime;      //修改修改时间
    ULONG FileAttributes;          //文件属性
} FILE_BASIC_INFORMATION, *PFILE_BASIC_INFORMATION;
```

其中，时间参数是从一个 LARGE_INTEGER 的整数，该整数代表从 1601 年经过多少个 100ns（纳秒）。FileAttributes 描述文件属性。FILE_ATTRIBUTE_NORMAL 描述一般文件。FILE_ATTRIBUTE_DIRECTORY 描述是目录。FILE_ATTRIBUTE_READONLY 描述该文件为只读。FILE_ATTRIBUTE_HIDDEN 代表隐含文件。FILE_ATTRIBUTE_SYSTEM 代表系统文件。

(3) 当 FileInformationClass 是 FileNameInformation 时，输入和输出的数据是 FILE_NAME_INFORMATION 结构体，描述文件名信息。

```
typedef struct _FILE_NAME_INFORMATION {
    ULONG FileNameLength;    //文件名长度
    WCHAR FileName[1];      //文件名
} FILE_NAME_INFORMATION, *PFILE_NAME_INFORMATION;
```

注意：这里指定文件名的字符串是宽字符集字符串。

(4) 当 FileInformationClass 是 FilePositionInformation 时，输入和输出的数据是 FILE_POSITION_INFORMATION 结构体，描述文件名信息。

```
typedef struct FILE_POSITION_INFORMATION {
    LARGE_INTEGER CurrentByteOffset; //代表当前文件指针位置
} FILE_POSITION_INFORMATION, *PFILE_POSITION_INFORMATION;
```

下面的代码演示了如何使用 ZwQueryInformationFile 函数查询、修改文件属性。

```
#001 OBJECT_ATTRIBUTES objectAttributes;
#002 IO_STATUS_BLOCK iostatus;
#003 HANDLE hfile;
#004 UNICODE_STRING logFileUnicodeString;
#005
#006 //初始化 UNICODE_STRING 字符串
#007 RtlInitUnicodeString(&logFileUnicodeString,
```

```

#008     L"\\??\\C:\\l.log");
#009 //或者写成 "\\Device\\HarddiskVolume1\\l.LOG"
#010
#011 //初始化 objectAttributes
#012 InitializeObjectAttributes(&objectAttributes,
#013                             &logFileUnicodeString,
#014                             OBJ_CASE_INSENSITIVE, //对大小写敏感
#015                             NULL,
#016                             NULL );
#017
#018 //创建文件
#019 NTSTATUS ntStatus = ZwCreateFile( &hfile,
#020                                     GENERIC_READ,
#021                                     &objectAttributes,
#022                                     &iosstatus,
#023                                     NULL,
#024                                     FILE_ATTRIBUTE_NORMAL,
#025                                     0,
#026                                     FILE_OPEN, //打开文件, 如果不存在则返回错误
#027                                     FILE_SYNCHRONOUS_IO_NONALERT,
#028                                     NULL,
#029                                     0 );
#030 if (NT_SUCCESS(ntStatus))
#031 {
#032     KdPrint(("open file successfully.\n"));
#033 }
#034
#035 FILE_STANDARD_INFORMATION fsi;
#036 //读取文件长度
#037 ntStatus = ZwQueryInformationFile(hfile,
#038                                     &iosstatus,
#039                                     &fsi,
#040                                     sizeof(FILE_STANDARD_INFORMATION),
#041                                     FileStandardInformation);
#042 if (NT_SUCCESS(ntStatus))
#043 {
#044     KdPrint(("file length:%u\n", fsi.EndOfFile.QuadPart));
#045 }
#046
#047 //修改当前文件指针
#048 FILE_POSITION_INFORMATION fpi;
#049 fpi.CurrentByteOffset.QuadPart = 100i64;
#050 ntStatus = ZwSetInformationFile(hfile,
#051                                     &iosstatus,
#052                                     &fpi,
#053                                     sizeof(FILE_POSITION_INFORMATION),
#054                                     FilePositionInformation);
#055 if (NT_SUCCESS(ntStatus))
#056 {
#057     KdPrint(("update the file pointer successfully.\n"));
#058 }
#059
#060 //关闭文件句柄
#061 ZwClose(hfile);

```

此段代码可以在配套光盘中本章的 FileTest 目录下找到。

6.2.4 文件的写操作

DDK 提供了文件写操作的内核函数，其函数声明如下：

```
NTSTATUS
ZwWriteFile(
    IN HANDLE FileHandle,
    IN HANDLE Event OPTIONAL,
    IN PIO_APC_ROUTINE ApcRoutine OPTIONAL,
    IN PVOID ApcContext OPTIONAL,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    IN PVOID Buffer,
    IN ULONG Length,
    IN PLARGE_INTEGER ByteOffset OPTIONAL,
    IN PULONG Key OPTIONAL
);
```

- FileHandle: 文件打开的句柄。
- Event: 很少用到，一般设置为 NULL。
- ApcRoutine: 很少用到，一般设置为 NULL。
- ApcContext: 很少用到，一般设置为 NULL。
- IoStatusBlock: 记录写操作的状态。其中，IoStatusBlock.Information 记录实际写了多少字节。
- Buffer: 从这个缓冲区开始往文件里写。
- Length: 准备写多少字节。
- ByteOffset: 从文件的多少偏移地址开始写。
- Key: 很少用到，一般设置为 NULL。

下面的代码演示了如何对文件进行写操作。（为了使代码简化和阅读更加方便，这里省略了对写操作是否成功的判断）

```
#001 OBJECT_ATTRIBUTES objectAttributes;
#002 IO_STATUS_BLOCK iostatus;
#003 HANDLE hfile;
#004 UNICODE_STRING logFileUnicodeString;
#005
#006 //初始化 UNICODE_STRING 字符串
#007 RtlInitUnicodeString(&logFileUnicodeString,
#008     L"\\??\\C:\\1.log");
#009 //或者写成 *\\Device\\HarddiskVolume1\\1.LOG"
#010
#011 //初始化 objectAttributes
#012 InitializeObjectAttributes(&objectAttributes,
#013     &logFileUnicodeString,
#014     OBJ_CASE_INSENSITIVE, //对大小写敏感
#015     NULL,
#016     NULL );
#017
#018 //创建文件
#019 NTSTATUS ntStatus = ZwCreateFile(&hfile,
```

```

#020                                     GENERIC_WRITE,
#021                                     &objectAttributes,
#022                                     &iostatus,
#023                                     NULL,
#024                                     FILE_ATTRIBUTE_NORMAL,
#025                                     FILE_SHARE_WRITE,
#026                                     FILE_OPEN_IF, //即使存在该文件, 也创建
#027                                     FILE_SYNCHRONOUS_IO_NONALERT,
#028                                     NULL,
#029                                     0 );
#030 #define BUFFER_SIZE 1024
#031     PCHAR pBuffer = (PCHAR)ExAllocatePool(PagedPool, BUFFER_SIZE);
#032     //构造要填充的数据
#033     RtlFillMemory(pBuffer, BUFFER_SIZE, 0xAA);
#034
#035     KdPrint(("The program will write %d bytes\n", BUFFER_SIZE));
#036     //写文件
#037     ZwWriteFile(hfile, NULL, NULL, NULL, &iostatus, pBuffer, BUFFER_SIZE, NULL, NULL);
#038     KdPrint(("The program really wrote %d bytes\n", iostatus.Information));
#039
#040
#041     //构造要填充的数据
#042     RtlFillMemory(pBuffer, BUFFER_SIZE, 0xBB);
#043
#044     KdPrint(("The program will append %d bytes\n", BUFFER_SIZE));
#045     //追加数据
#046     LARGE_INTEGER number;
#047     number.QuadPart = 1024i64; //设置文件指针
#048     //对文件进行附加写
#049     ZwWriteFile(hfile, NULL, NULL, NULL, &iostatus, pBuffer, BUFFER_SIZE, &number,
NULL);
#050     KdPrint(("The program really appended %d bytes\n", iostatus.Information));
#051
#052     //关闭文件句柄
#053     ZwClose(hfile);
#054
#055     ExFreePool(pBuffer);

```

此段代码可以在配套光盘中本章的 FileTest 目录下找到。

6.2.5 文件的读操作

DDK 提供了文件读操作的内核函数, 其函数声明如下:

```

NTSTATUS
ZwWriteFile(
    IN HANDLE FileHandle,
    IN HANDLE Event OPTIONAL,
    IN PIO_APC_ROUTINE ApcRoutine OPTIONAL,
    IN PVOID ApcContext OPTIONAL,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    IN PVOID Buffer,
    IN ULONG Length,
    IN PLARGE_INTEGER ByteOffset OPTIONAL,
    IN PULONG Key OPTIONAL
);

```


Windows 驱动开发技术详解

- FileHandle: 文件打开的句柄。
- Event: 很少用到, 一般设置为 NULL。
- ApcRoutine: 很少用到, 一般设置为 NULL。
- ApcContext: 很少用到, 一般设置为 NULL。
- IoStatusBlock: 记录写操作的状态。其中, IoStatusBlock.Information 记录实际写了多少字节。
- Buffer: 从这个缓冲区开始从文件里读。
- Length: 准备写多少字节。
- ByteOffset: 从文件的多少偏移地址开始写。
- Key: 很少用到, 一般设置为 NULL。

下面的代码演示了如何在驱动程序中读文件。如果是读取整个文件, 需要知道文件的大小, 该功能可以通过内核函数 ZwQueryInformationFile 来实现。

下面的代码演示了如何在驱动程序中读取文件。

```
#001 OBJECT_ATTRIBUTES objectAttributes;  
#002 IO_STATUS_BLOCK iostatus;  
#003 HANDLE hfile;  
#004 UNICODE_STRING logFileUnicodeString;  
#005  
#006 //初始化 UNICODE_STRING 字符串  
#007 RtlInitUnicodeString( &logFileUnicodeString,  
#008 L"\\??\\C:\\1.log");  
#009 //或者写成 "\\Device\\HarddiskVolume1\\1.LOG"  
#010  
#011 //初始化 objectAttributes  
#012 InitializeObjectAttributes(&objectAttributes,  
#013 &logFileUnicodeString,  
#014 OBJ_CASE_INSENSITIVE, //对大小写敏感  
#015 NULL,  
#016 NULL );  
#017  
#018 //创建文件  
#019 NTSTATUS ntStatus = ZwCreateFile( &hfile,  
#020 GENERIC_READ,  
#021 &objectAttributes,  
#022 &iostatus,  
#023 NULL,  
#024 FILE_ATTRIBUTE_NORMAL,  
#025 FILE_SHARE_READ,  
#026 FILE_OPEN, //即使存在该文件, 也创建  
#027 FILE_SYNCHRONOUS_IO_NONALERT,  
#028 NULL,  
#029 0 );  
#030  
#031 if (NT_SUCCESS(ntStatus))  
#032 {  
#033     KdPrint(("The file is not exist!\\n"));  
#034     return;  
#035 }  
#036
```

```

#037 FILE_STANDARD_INFORMATION fsi;
#038 //读取文件长度
#039 ntStatus = ZwQueryInformationFile(hfile,
#040                                     &iostatus,
#041                                     &fsi,
#042                                     sizeof(FILE_STANDARD_INFORMATION),
#043                                     FileStandardInformation);
#044
#045 KdPrint(("The program want to read %d bytes\n", fsi.EndOfFile.QuadPart));
#046
#047 //为读取的文件分配缓冲区
#048 PCHAR pBuffer = (PCHAR)ExAllocatePool(PagedPool,
#049                                         (LONG) fsi.EndOfFile.QuadPart);
#050
#051 //读取文件
#052 ZwReadFile(hfile, NULL,
#053            NULL, NULL,
#054            &iostatus,
#055            pBuffer,
#056            (LONG) fsi.EndOfFile.QuadPart,
#057            NULL, NULL);
#058 KdPrint(("The program really read %d bytes\n", iostatus.Information));
#059 //关闭文件句柄
#060 ZwClose(hfile);
#061
#062 //释放缓冲区
#063 ExFreePool(pBuffer);

```

此段代码可以在配套光盘中本章的 FileTest 目录下找到。

6.3 内核模式下的注册表操作

在驱动程序的开发中，经常会用到对注册表的操作。与 Win32 的 API 不同，DDK 提供另外一套对注册表操作的相关函数。首先明确一下注册表里的几个概念，避免在后面混淆。

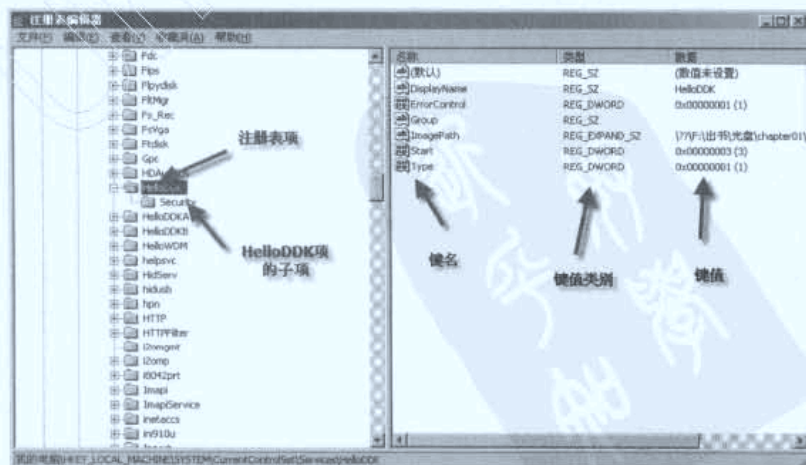


图 6-2 注册表概念

如图 6-2 所示，有五个概念需要重申一下：

- 注册表项：注册表中的一个项目，类似目录的概念。每个项中存储多个二元结构，键名—键值。每个项中，可以有若干个子项。
- 注册表子项：类似于目录中的子目录。
- 键名：通过键名可以寻找到相应的键值。
- 键值类别：每个键值存储的时候有不同的类别，可以是整型、字符串等数据。
- 键值：键名下对应存储的数据。

6.3.1 创建关闭注册表

和文件操作类似，对注册表操作首先要获取一个注册表句柄。对注册表的操作都需要根据这个句柄进行操作。可以通过 `ZwCreateKey` 函数获得打开的注册表句柄。这个函数打开注册表后，并返回一个操作句柄。其函数声明如下：

```
NTSTATUS
ZwCreateKey(
    OUT PHANDLE KeyHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    IN ULONG TitleIndex,
    IN PUNICODE_STRING Class OPTIONAL,
    IN ULONG CreateOptions,
    OUT PULONG Disposition OPTIONAL
);
```

- `KeyHandle`：获得的注册表句柄。
- `DesiredAccess`：访问权限，一般设置为 `KEY_ALL_ACCESS`。
- `ObjectAttributes`：`OBJECT_ATTRIBUTES` 数据结构。
- `TitleIndex`：很少用到，一般设置为 0。
- `Class`：很少用到，一般设置为 `NULL`。
- `CreateOptions`：创建时的选项，一般设置为 `REG_OPTION_NON_VOLATILE`。
- `Disposition`：返回是创建成功，还是打开成功。返回值是 `REG_CREATED_NEW_KEY` 或者是 `REG_OPENED_EXISTING_KEY`。
- 返回值：返回是否创建成功。

如果 `ZwCreateKey` 指定的项目不存在，则直接创建这个项目，并利用 `Disposition` 参数返回 `REG_CREATED_NEW_KEY`。如果该项目已经存在了，`Disposition` 参数返回 `REG_OPENED_EXISTING_KEY`。以下的代码演示了如何使用 `ZwCreateKey` 函数打开注册表。

```
#001 //创建或打开某注册表项目
#002 UNICODE_STRING RegUnicodeString;
#003 HANDLE hRegister;
#004
```

```

#005 //初始化 UNICODE_STRING 字符串
#006 RtlInitUnicodeString( &RegUnicodeString,
#007 MY_REG_SOFTWARE_KEY_NAME);
#008
#009 OBJECT_ATTRIBUTES objectAttributes;
#010 //初始化 objectAttributes
#011 InitializeObjectAttributes(&objectAttributes,
#012 &RegUnicodeString,
#013 OBJ_CASE_INSENSITIVE, //对大小写敏感
#014 NULL,
#015 NULL );
#016 ULONG ulResult;
#017 //创建或打开注册表项目
#018 NTSTATUS ntStatus = ZwCreateKey( &hRegister,
#019 KEY_ALL_ACCESS,
#020 &objectAttributes,
#021 0,
#022 NULL,
#023 REG_OPTION_NON_VOLATILE,
#024 &ulResult);
#025
#026 if (NT_SUCCESS(ntStatus))
#027 {
#028 //判断是被新创建, 还是已经被创建
#029 if(ulResult==REG_CREATED_NEW_KEY)
#030 {
#031 KdPrint(("The register item is created\n"));
#032 }else if(ulResult==REG_OPENED_EXISTING_KEY)
#033 {
#034 KdPrint(("The register item has been created, and now is opened\n"));
#035 }
#036 }
#037
#038 //创建或打开某注册表项目的子项
#039 UNICODE_STRING subRegUnicodeString;
#040 HANDLE hSubRegister;
#041
#042 //初始化 UNICODE_STRING 字符串
#043 RtlInitUnicodeString( &subRegUnicodeString,
#044 L"SubItem");
#045
#046 OBJECT_ATTRIBUTES subObjectAttributes;
#047 //初始化 subObjectAttributes
#048 InitializeObjectAttributes(&subObjectAttributes,
#049 &subRegUnicodeString,
#050 OBJ_CASE_INSENSITIVE, //对大小写敏感
#051 hRegister,
#052 NULL );
#053 //创建或打开注册表项目
#054 ntStatus = ZwCreateKey( &hSubRegister,
#055 KEY_ALL_ACCESS,
#056 &subObjectAttributes,
#057 0,
#058 NULL,
#059 REG_OPTION_NON_VOLATILE,
#060 &ulResult);
#061

```



```

#062     if (NT_SUCCESS(ntStatus))
#063     {
#064         //判断是被新创建, 还是已经被创建
#065         if (ulResult==REG_CREATED_NEW_KEY)
#066         {
#067             KdPrint(("The sub register item is created\n"));
#068         }else if (ulResult==REG_OPENED_EXISTING_KEY)
#069         {
#070             KdPrint(("The sub register item has been created,and now is opened\n"));
#071         }
#072     }
#073
#074     //关闭注册表句柄
#075     ZwClose(hRegister);
#076     ZwClose(hSubRegister);

```

此段代码可以在配套光盘中本章的 RegTest 目录下找到。

6.3.2 打开注册表

ZwCreateKey 函数既可以创建注册表项, 也可以打开注册表项。为了简化打开注册表项, DDK 提供了内核函数 ZwOpenKey, 以简化打开操作。如果 ZwOpenKey 指定的项不存在, 不会创建这个项目, 而是返回一个错误状态。该函数的声明如下:

```

NTSTATUS
ZwOpenKey(
    OUT PHANDLE KeyHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes
);

```

- KeyHandle: 返回被打开的句柄。
- DesiredAccess: 打开的权限, 一般设为 KEY_ALL_ACCESS。
- ObjectAttributes: OBJECT_ATTRIBUTES 数据结构, 指示打开的状态。
- 返回值: 返回是否打开成功。

ZwOpenKey 的参数比 ZwCreateKey 的参数简化, 程序员使用该函数会很方便。下面的代码演示了如何使用 ZwOpenKey 打开注册表项目。

```

#001     UNICODE_STRING RegUnicodeString;
#002     HANDLE hRegister;
#003
#004     //初始化 UNICODE_STRING 字符串
#005     RtlInitUnicodeString( &RegUnicodeString,
#006         MY_REG_SOFTWARE_KEY_NAME);
#007
#008     OBJECT_ATTRIBUTES objectAttributes;
#009     //初始化 objectAttributes
#010     InitializeObjectAttributes(&objectAttributes,
#011         &RegUnicodeString,
#012         OBJ_CASE_INSENSITIVE, //对大小写敏感
#013         NULL,
#014         NULL );

```

```
#015 //打开注册表
#016 NTSTATUS ntStatus = ZwOpenKey( &hRegister,
#017                                KEY_ALL_ACCESS,
#018                                &objectAttributes);
#019 //判断操作是否成功
#020 if (NT_SUCCESS(ntStatus))
#021 {
#022     KdPrint(("Open register successfully\n"));
#023 }
#024 //关闭句柄
#025 ZwClose(hRegister);
```

此段代码可以在配套光盘中本章的 RegTest 目录下找到。

6.3.3 添加、修改注册表键值

打开注册表的句柄后，就可以对该项进行设置和修改了。注册表是以二元形式存储的，即“键名”和“键值”。通过键名设置键值，而键值可以划分几个类，如表 6-1 所示。

表 6-1 键值的分类

分 类	描 述
REG_BINARY	键值用二进制存储
REG_SZ	键值用宽字符串，字符串以\0 的结尾
REG_EXPAND_SZ	键值用宽字符串，字符串以\0 的结尾,该字符串是扩展的字符
REG_MULTI_SZ	键值存储多个字符串，每个字符串以\0 隔开
REG_DWORD	键值用 4 字节整型存储
REG_QWORD	键值用 8 字节存储

在添加和修改注册表键值的时候，要分类进行添加和修改。DDK 提供了 ZwSetValueKey 函数来完成这个任务，其函数声明是：

```
NTSTATUS
ZwSetValueKey(
    IN HANDLE KeyHandle,
    IN PUNICODE_STRING ValueName,
    IN ULONG TitleIndex OPTIONAL,
    IN ULONG Type,
    IN PVOID Data,
    IN ULONG DataSize
);
```

- KeyHandle: 注册表句柄。
- ValueName: 要新建或者修改的键名。
- TitleIndex: 很少用，一般设为 0。
- Type: 在表 6-1 中选择一种类型。
- DataSize: 记录键值数据的大小。
- 返回值: 返回新建或者修改的结果。

使用 ZwSetValueKey 函数的时候，如果指定的键名不存在，则直接创建。如果指定键

名已经存在，则对已有键值进行修改。当新建或者修改键值的时候，根据表 6-1 不同的类别，Data 指向不同的数据结构，并且用 DataSize 指明数据大小。例如，REG_DWORD 类型，对应的数据大小就是 4，REG_QWORD 数据类型，数据大小就是 8，如果是 REG_SZ，数据长度是字符串长度的二倍加上两个字节。下面的代码演示了如何修改和设置键值。

```
#001 UNICODE_STRING RegUnicodeString;
#002 HANDLE hRegister;
#003
#004 //初始化 UNICODE_STRING 字符串
#005 RtlInitUnicodeString( &RegUnicodeString,
#006 MY_REG_SOFTWARE_KEY_NAME);
#007
#008 OBJECT_ATTRIBUTES objectAttributes;
#009 //初始化 objectAttributes
#010 InitializeObjectAttributes(&objectAttributes,
#011 &RegUnicodeString,
#012 OBJ_CASE_INSENSITIVE, //对大小写敏感
#013 NULL,
#014 NULL );
#015 //打开注册表
#016 NTSTATUS ntStatus = ZwOpenKey( &hRegister,
#017 KEY_ALL_ACCESS,
#018 &objectAttributes);
#019
#020 if (NT_SUCCESS(ntStatus))
#021 {
#022     KdPrint(("Open register successfully\n"));
#023 }
#024
#025 UNICODE_STRING ValueName;
#026 //初始化 ValueName
#027 RtlInitUnicodeString( &ValueName, L"REG_DWORD value");
#028
#029 //设置 REG_DWORD 子键
#030 ULONG ulValue = 1000;
#031 ZwSetValueKey(hRegister,
#032 &ValueName,
#033 0,
#034 REG_DWORD,
#035 &ulValue,
#036 sizeof(ulValue));
#037
#038 //初始化 ValueName
#039 RtlInitUnicodeString( &ValueName, L"REG_SZ value");
#040 WCHAR* strValue = L"hello world";
#041
#042 //设置 REG_SZ 子键
#043 ZwSetValueKey(hRegister,
#044 &ValueName,
#045 0,
#046 REG_SZ,
#047 strValue,
#048 wcslen(strValue)*2+2);
#049
#050 //初始化 ValueName
```

```

#051 RtlInitUnicodeString( &ValueName, L"REG_BINARY value");
#052
#053 UCHAR buffer[10];
#054 RtlFillMemory(buffer, sizeof(buffer), 0xFF);
#055
#056 //设置 REG_MULTI_SZ 子键
#057 ZwSetValueKey(hRegister,
#058               &ValueName,
#059               0,
#060               REG_BINARY,
#061               buffer,
#062               sizeof(buffer));
#063 //关闭注册表句柄
#064 ZwClose(hRegister);
#065 }

```

此段代码可以在配套光盘中本章的 RegTest 目录下找到。

6.3.4 查询注册表

驱动程序中有时需要对注册表的项进行查询，从而获取注册表的键值。DDK 提供的 ZwQueryValueKey 函数可以完成这个任务，其声明如下：

```

NTSTATUS
ZwQueryValueKey(
    IN HANDLE KeyHandle,
    IN PUNICODE_STRING ValueName,
    IN KEY_VALUE_INFORMATION_CLASS KeyValueInformationClass,
    OUT PVOID KeyValueInformation,
    IN ULONG Length,
    OUT PULONG ResultLength
);

```

- KeyHandle: 打开的注册表句柄。
- ValueName: 要查询的键名。
- KeyValueInformationClass: 根据 KeyValueInformation 的不同选择不同的查询类别。
- KeyValueInformation: 选择一种查询类别。选择 KeyValueBasicInformation、KeyValueFullInformation 或者 KeyValuePartialInformation。
- Length: 要查数据的长度。
- ResultLength: 实际查询数据的长度。
- 返回值: 表示查询数据是否成功。

使用 ZwQueryValueKey 函数查询注册表时，需要用 KeyValueInformationClass 选择一种查询方式。这可以是 KeyValueBasicInformation、KeyValueFullInformation 或者 KeyValuePartialInformation 中的一种。这分别代表查询基本信息，查询全部信息和查询部分信息，每种查询类型会有对应的一种数据结构获得查询结果。

一般情况下，选择 KeyValuePartialInformation 就可以查询键值的数据了，它对应的查询数据结构是 KEY_VALUE_PARTIAL_INFORMATION 的数据结构。


```
typedef struct _KEY_VALUE_PARTIAL_INFORMATION {
    ULONG TitleIndex;
    ULONG Type;           //数据的类型, 参考表 6-1
    ULONG DataLength;     //数据的长度
    UCHAR Data[1];       //数据指针, 这里是变长的数据
} KEY_VALUE_PARTIAL_INFORMATION, *PKEY_VALUE_PARTIAL_INFORMATION;
```

KEY_VALUE_PARTIAL_INFORMATION 的数据结构长度不固定, 所以首先要确定这个长度。一般使用 ZwQueryValueKey 分为 4 个步骤。

- ① 用 ZwQueryValueKey 获取这个数据结构的长度。
- ② 分配如此长度的内存, 用来查询。
- ③ 再次调用 ZwQueryValueKey, 获取键值。
- ④ 回收内存。

以下是针对查询注册表的示例代码。

```
#001  UNICODE_STRING RegUnicodeString;
#002  HANDLE hRegister;
#003
#004  //初始化 UNICODE_STRING 字符串
#005  RtlInitUnicodeString( &RegUnicodeString,
#006  MY_REG_SOFTWARE_KEY_NAME);
#007
#008  OBJECT_ATTRIBUTES objectAttributes;
#009  //初始化 objectAttributes
#010  InitializeObjectAttributes(&objectAttributes,
#011  &RegUnicodeString,
#012  OBJ_CASE_INSENSITIVE, //对大小写敏感
#013  NULL,
#014  NULL );
#015  //打开注册表
#016  NTSTATUS ntStatus = ZwOpenKey( &hRegister,
#017  KEY_ALL_ACCESS,
#018  &objectAttributes);
#019
#020  if (NT_SUCCESS(ntStatus))
#021  {
#022      KdPrint(("Open register successfully\n"));
#023  }
#024
#025  UNICODE_STRING ValueName;
#026  //初始化 ValueName
#027  RtlInitUnicodeString( &ValueName, L"REG_DWORD value");
#028
#029  //读取 REG_DWORD 子键
#030  ULONG ulSize;
#031  ntStatus = ZwQueryValueKey(hRegister,
#032  &ValueName,
#033  KeyValuePartialInformation,
#034  NULL,
#035  0,
#036  &ulSize);
#037
#038  if (ntStatus==STATUS_OBJECT_NAME_NOT_FOUND || ulSize==0)
#039  {
```

```

#040     ZwClose(hRegister);
#041     KdPrint(("The item is not exist\n"));
#042     return;
#043 }
#044 PKEY_VALUE_PARTIAL_INFORMATION pvpi =
#045     (PKEY_VALUE_PARTIAL_INFORMATION)
#046     ExAllocatePool(PagedPool, ulSize);
#047 //查询注册表
#048 ntStatus = ZwQueryValueKey(hRegister,
#049                             &ValueName,
#050                             KeyValuePartialInformation,
#051                             pvpi,
#052                             ulSize,
#053                             &ulSize);
#054 if (!NT_SUCCESS(ntStatus))
#055 {
#056     ZwClose(hRegister);
#057     KdPrint(("Read register error\n"));
#058     return;
#059 }
#060 //判断是否为 REG_DWORD 类型
#061 if (pvpi->Type==REG_DWORD && pvpi->DataLength==sizeof(ULONG))
#062 {
#063     PULONG pulValue = (PULONG) pvpi->Data;
#064     KdPrint(("The value:%d\n", *pulValue));
#065 }
#066
#067 ExFreePool(pvpi);
#068
#069 //初始化 ValueName
#070 RtlInitUnicodeString(&ValueName, L"REG_SZ value");
#071 //读取 REG_SZ 子键
#072 ntStatus = ZwQueryValueKey(hRegister,
#073                             &ValueName,
#074                             KeyValuePartialInformation,
#075                             NULL,
#076                             0,
#077                             &ulSize);
#078
#079 if (ntStatus==STATUS_OBJECT_NAME_NOT_FOUND || ulSize==0)
#080 {
#081     ZwClose(hRegister);
#082     KdPrint(("The item is not exist\n"));
#083     return;
#084 }
#085 //申请内存
#086 pvpi =
#087     (PKEY_VALUE_PARTIAL_INFORMATION)
#088     ExAllocatePool(PagedPool, ulSize);
#089 //查询注册表
#090 ntStatus = ZwQueryValueKey(hRegister,
#091                             &ValueName,
#092                             KeyValuePartialInformation,
#093                             pvpi,
#094                             ulSize,
#095                             &ulSize);
#096 if (NT_SUCCESS(ntStatus))
#097 {

```



```

#098      ZwClose(hRegister);
#099      KdPrint(("Read regsiter error\n"));
#100      return;
#101      }
#102      //判断是否为 REG_SZ 类型
#103      if (pvpi->Type==REG_SZ)
#104      {
#105          KdPrint(("The value:%S\n",pvpi->Data));
#106      }
#107      //关闭句柄
#108      ZwClose(hRegister);

```

此段代码可以在配套光盘中本章的 RegTest 目录下找到。

6.3.5 枚举子项

在注册表的操作中，还经常有另外两种操作，分别是枚举子项和枚举子键。枚举子项就是事先不知道该项中有多少个子项目，用某个函数将子项一一列举出来。而枚举子键是事先不知道该项中有多少个子键，用某个函数一一将子键列举出来。

DDK 提供了枚举子项的函数，它是 ZwQueryKey 函数（注意不是 ZwQueryValueKey 函数）和 ZwEnumerateKey。先看一下这两个函数的声明。

```

NTSTATUS
ZwQueryKey(
    IN HANDLE KeyHandle,
    IN KEY_INFORMATION_CLASS KeyInformationClass,
    OUT PVOID KeyInformation,
    IN ULONG Length,
    OUT PULONG ResultLength
);

```

- KeyHandle: 注册表项的句柄。
- KeyInformationClass: 查询的类别，一般选择 KeyFullInformation。
- KeyInformation: 查询的数据指针。如果 KeyInformationClass 是 KeyFullInformation，则该指针指向一个 KEY_FULL_INFORMATION 的数据结构。
- Length: 数据长度。
- ResultLength: 返回的数据长度。
- 返回值: 指示查询是否成功。

```

NTSTATUS
ZwEnumerateKey(
    IN HANDLE KeyHandle,
    IN ULONG Index,
    IN KEY_INFORMATION_CLASS KeyInformationClass,
    OUT PVOID KeyInformation,
    IN ULONG Length,
    OUT PULONG ResultLength
);

```

- KeyHandle: 注册表项句柄。
- Index: 很少用到，一般为 0。

- KeyInformationClass: 该子项的信息。
- Length: 子项信息的长度。
- ResultLength: 返回子键信息的长度。
- 返回值: 指明枚举是否成功。

ZwQueryKey 的作用主要是获得某注册表项究竟有多少个子项, 而 ZwEnumerateKey 的作用主要是针对第几个子项获取该子项的具体信息。

在使用 ZwQueryKey 时, 可以将参数 KeyInformationClass 指定为 KeyFullInformation。这样参数 KeyInformation 就对应一个 KEY_FULL_INFORMATION 的数据结构, 该数据结构中的 SubKeys 指明了项中有多少个子项。

KEY_FULL_INFORMATION 数据结构的大小是变长的, 所以要调用两次 ZwQueryKey。第一次获取 KEY_FULL_INFORMATION 数据的长度, 第二次真正获取 KEY_FULL_INFORMATION 数据。

在使用 ZwEnumerateKey 时, 需要将参数 KeyInformationClass 设置为 KeyBasicInformation, 这样其参数 KeyInformation 就能对应 KEY_BASIC_INFORMATION 的数据结构。

同理, KEY_BASIC_INFORMATION 也是变长的数据结构, 需要两次调用 ZwEnumerateKey。第一次获取 KEY_BASIC_INFORMATION 的长度, 第二次获取 KEY_BASIC_INFORMATION 数据。

下面的例子演示了如何枚举子键。

```
#001 UNICODE_STRING RegUnicodeString;
#002 HANDLE hRegister;
#003
#004 //初始化 UNICODE_STRING 字符串
#005 RtlInitUnicodeString( &RegUnicodeString,
#006 MY_REG_SOFTWARE_KEY_NAME);
#007
#008 OBJECT_ATTRIBUTES objectAttributes;
#009 //初始化 objectAttributes
#010 InitializeObjectAttributes(&objectAttributes,
#011 &RegUnicodeString,
#012 OBJ_CASE_INSENSITIVE, //对大小写敏感
#013 NULL,
#014 NULL );
#015 //打开注册表
#016 NTSTATUS ntStatus = ZwOpenKey( &hRegister,
#017 KEY_ALL_ACCESS,
#018 &objectAttributes);
#019
#020 if (NT_SUCCESS(ntStatus))
#021 {
#022     KdPrint(("Open register successfully\n"));
#023 }
#024
#025 ULONG ulSize;
#026 //第一次调用 ZwQueryKey, 为了获取 KEY_FULL_INFORMATION 数据的长度
#027 ZwQueryKey(hRegister,
#028 KeyFullInformation,
```



```

#029     NULL,
#030     0,
#031     &ulSize);
#032
#033     PKEY_FULL_INFORMATION pfi =
#034     (PKEY_FULL_INFORMATION)
#035     ExAllocatePool(PagedPool,ulSize);
#036
#037     //第二次调用 ZwQueryKey, 为了获取 KEY_FULL_INFORMATION 数据
#038     ZwQueryKey(hRegister,
#039     KeyFullInformation,
#040     pfi,
#041     ulSize,
#042     &ulSize);
#043
#044     for (ULONG i=0;i<pfi->SubKeys;i++)
#045     {
#046         //第一次调用 ZwEnumerateKey, 为了获取 KEY_BASIC_INFORMATION 数据的长度
#047         ZwEnumerateKey(hRegister,
#048             i,
#049             KeyBasicInformation,
#050             NULL,
#051             0,
#052             &ulSize);
#053
#054         PKEY_BASIC_INFORMATION pbi =
#055         (PKEY_BASIC_INFORMATION)
#056         ExAllocatePool(PagedPool,ulSize);
#057
#058         //第二次调用 ZwEnumerateKey, 为了获取 KEY_BASIC_INFORMATION 数据
#059         ZwEnumerateKey(hRegister,
#060             i,
#061             KeyBasicInformation,
#062             pbi,
#063             ulSize,
#064             &ulSize);
#065
#066         UNICODE_STRING uniKeyName;
#067         uniKeyName.Length =
#068         uniKeyName.MaximumLength =
#069         (USHORT)pbi->NameLength;
#070         uniKeyName.Buffer = pbi->Name;
#071         KdPrint(("The %d sub item name:%wZ\n",i,&uniKeyName));
#072         //回收内存
#073         ExFreePool(pbi);
#074     }
#075     //回收内存
#076     ExFreePool(pfi);
#077     //关闭句柄
#078     ZwClose(hRegister);

```

此段代码可以在配套光盘中本章的 RegTest 目录下找到。

6.3.6 枚举子键

和枚举子项类似,枚举子键是通过 ZwQueryKey 和 ZwEnumerateValueKey 两个函数的配合完成的。ZwEnumerateValueKey 函数的使用和 ZwEnumerateKey 函数的使用类似。

下面的例子演示了如何在驱动程序中枚举子键。

```
#001 UNICODE_STRING RegUnicodeString;
#002 HANDLE hRegister;
#003
#004 //初始化 UNICODE_STRING 字符串
#005 RtlInitUnicodeString( &RegUnicodeString,
#006 MY_REG_SOFTWARE_KEY_NAME);
#007
#008 OBJECT_ATTRIBUTES objectAttributes;
#009 //初始化 objectAttributes
#010 InitializeObjectAttributes(&objectAttributes,
#011 &RegUnicodeString,
#012 OBJ_CASE_INSENSITIVE, //对大小写敏感
#013 NULL,
#014 NULL );
#015 //打开注册表
#016 NTSTATUS ntStatus = ZwOpenKey( &hRegister,
#017 KEY_ALL_ACCESS,
#018 &objectAttributes);
#019 //判断操作是否成功
#020 if (NT_SUCCESS(ntStatus))
#021 {
#022 KdPrint(("Open register successfully\n"));
#023 }
#024
#025 ULONG ulSize;
#026 //查询注册表
#027 ZwQueryKey(hRegister,
#028 KeyFullInformation,
#029 NULL,
#030 0,
#031 &ulSize);
#032
#033 PKEY_FULL_INFORMATION pfi =
#034 (PKEY_FULL_INFORMATION)
#035 ExAllocatePool(PagedPool, ulSize);
#036 //查询注册表
#037 ZwQueryKey(hRegister,
#038 KeyFullInformation,
#039 pfi,
#040 ulSize,
#041 &ulSize);
#042
#043 for (ULONG i=0; i<pfi->Values; i++)
#044 {
#045 //枚举注册表
#046 ZwEnumerateValueKey(hRegister,
#047 i,
#048 KeyValueBasicInformation,
#049 NULL,
#050 0,
#051 &ulSize);
#052 PKEY_VALUE_BASIC_INFORMATION pvbi =
#053 (PKEY_VALUE_BASIC_INFORMATION)
#054 ExAllocatePool(PagedPool, ulSize);
#055 //枚举注册表
#056 ZwEnumerateValueKey(hRegister,
#057 i,
```



```

#058         KeyValueBasicInformation,
#059         pvbi,
#060         ulSize,
#061         &ulSize);
#062     UNICODE_STRING uniKeyName;
#063     uniKeyName.Length =
#064     uniKeyName.MaximumLength =
#065     (USHORT)pvbi->NameLength;
#066     uniKeyName.Buffer = pvbi->Name;
#067     KdPrint(("The %d sub value name:%wZ\n",i,&uniKeyName));
#068     if (pvbi->Type==REG_SZ)
#069     {
#070         KdPrint(("The sub value type:REG_SZ\n"));
#071     }else if (pvbi->Type==REG_MULTI_SZ)
#072     {
#073         KdPrint(("The sub value type:REG_MULTI_SZ\n"));
#074     }else if (pvbi->Type==REG_DWORD)
#075     {
#076         KdPrint(("The sub value type:REG_DWORD\n"));
#077     }else if (pvbi->Type==REG_BINARY)
#078     {
#079         KdPrint(("The sub value type:REG_BINARY\n"));
#080     }
#081     ExFreePool(pvbi);
#082 }
#083 //回收内存
#084 ExFreePool(pfi);
#085 //关闭句柄
#086 ZwClose(hRegister);

```

此段代码可以在配套光盘中本章的 RegTest 目录下找到。

6.3.7 删除子项

DDK 同样提供了删除子项的内核函数，它就是 ZwDeleteKey。其声明是：

```

NTSTATUS
ZwDeleteKey(
    IN HANDLE KeyHandle
);

```

➤ KeyHandle: 打开的文件句柄。

➤ 返回值: 指示是否删除成功。

需要指出，该函数只能删除没有子项的项目。如果项中还有子项，则不能删除。这时候需要先将该项中的所有子项全部删除后，再删除该项。下面的例子演示了如何在驱动程序中删除子项。

```

#001     UNICODE_STRING RegUnicodeString;
#002     HANDLE hRegister;
#003
#004     #define MY_REG_SOFTWARE_KEY_NAME1 L"\\Registry\\Machine\\Software\\Zhangfan\\SubItem"
#005     //初始化 UNICODE_STRING 字符串
#006     RtlInitUnicodeString(&RegUnicodeString,
#007         MY_REG_SOFTWARE_KEY_NAME1);
#008

```

```

#009  OBJECT_ATTRIBUTES objectAttributes;
#010  //初始化 objectAttributes
#011  InitializeObjectAttributes(&objectAttributes,
#012                                &RegUnicodeString,
#013                                OBJ_CASE_INSENSITIVE, //对大小写敏感
#014                                NULL,
#015                                NULL );
#016  //打开注册表
#017  NTSTATUS ntStatus = ZwOpenKey( &hRegister,
#018                                KEY_ALL_ACCESS,
#019                                &objectAttributes);
#020  //判断操作是否成功
#021  if (NT_SUCCESS(ntStatus))
#022  {
#023      KdPrint(("Open register successfully\n"));
#024  }
#025  //删除注册表键
#026  ntStatus = ZwDeleteKey(hRegister);
#027  if (NT_SUCCESS(ntStatus))
#028  {
#029      KdPrint(("Delete the item successfully\n"));
#030  }else if (ntStatus == STATUS_ACCESS_DENIED)
#031  {
#032      KdPrint(("STATUS_ACCESS_DENIED\n"));
#033  }else if (ntStatus == STATUS_INVALID_HANDLE)
#034  {
#035      KdPrint(("STATUS_INVALID_HANDLE\n"));
#036  }else
#037  {
#038      KdPrint(("Maybe the item has sub item to delete\n"));
#039  }
#040  //关闭句柄
#041  ZwClose(hRegister);
#042

```

此段代码可以在配套光盘中本章的 RegTest 目录下找到。

6.3.8 其他

以上介绍了部分内核模式下操作注册表的函数，使用方法都比较烦琐，往往一个操作需要若干个函数的共同配合。

为了简化注册表操作，DDK 还提供了一系列以 Rtl 开头的运行时函数，这些函数把前面介绍的函数进行了封装。往往一条函数就能实现前面介绍的若干条函数的功能。

表 6-2 RtlXX 关于注册表的操作

分 类	描 述
RtlCreatcRegistryKey	创建注册表
RtlCheckRegistryKey	查看某注册表项是否存在
RtlWriteRegistryValue	写注册表
RtlDeleteRegistryValue	删除注册表的子键

Windows 驱动开发技术详解

表 6-2 列出了 RtlXX 系列函数对注册表的相关操作。下面的代码演示了在驱动程序中如何使用这些函数。从代码中可以看出，其操作比 ZwXX 系列函数的操作要简单得多。

```
#001 //创建子项目
#002 NTSTATUS ntStatus =
#003     RtlCreateRegistryKey(RTL_REGISTRY_SERVICES,L"HelloDDK\\Zhangfan");
#004 if (NT_SUCCESS(ntStatus))
#005 {
#006     KdPrint(("Create the item successfully\n"));
#007 }
#008
#009 //检查某项是否存在
#010 ntStatus =
#011     RtlCheckRegistryKey(RTL_REGISTRY_SERVICES,L"HelloDDK\\Zhangfan");
#012 if (NT_SUCCESS(ntStatus))
#013 {
#014     KdPrint(("The item is exist\n"));
#015 }
#016
#017 //写入 REG_DWORD 的数据
#018 ULONG value1 = 100;
#019 //写注册表
#020 ntStatus =
#021     RtlWriteRegistryValue(RTL_REGISTRY_SERVICES,
#022                           L"HelloDDK\\Zhangfan",
#023                           L"DWORD_Value",
#024                           REG_DWORD,
#025                           &value1,
#026                           sizeof(value1));
#027 if (NT_SUCCESS(ntStatus))
#028 {
#029     KdPrint(("Write the DWORD value succuessfully\n"));
#030 }
#031
#032 PWCHAR szString = L"Hello DDK";
#033 //写注册表
#034 ntStatus =
#035     RtlWriteRegistryValue(RTL_REGISTRY_SERVICES,
#036                           L"HelloDDK\\Zhangfan",
#037                           L"SZ_Value",
#038                           REG_SZ,
#039                           szString,
#040                           wcslen(szString)*2+2);
#041 //判断操作是否成功
#042 if (NT_SUCCESS(ntStatus))
#043 {
#044     KdPrint(("Write the REG_SZ value succuessfully\n"));
#045 }
#046 //初始化变量
#047 RTL_QUERY_REGISTRY_TABLE paramTable[2];
#048 RtlZeroMemory(paramTable, sizeof(paramTable));
#049
#050 ULONG defaultData=0;
#051 ULONG uQueryValue;
#052 paramTable[0].Flags = RTL_QUERY_REGISTRY_DIRECT;
#053 paramTable[0].Name = L"DWORD_Value";
#054 paramTable[0].EntryContext = &uQueryValue;
```

```

#055     paramTable[0].DefaultType = REG_DWORD;
#056     paramTable[0].DefaultData = &defaultData;
#057     paramTable[0].DefaultLength = sizeof(ULONG);
#058
#059     //查询 REG_DWORD 的数据
#060     ntStatus = RtlQueryRegistryValues(RTL_REGISTRY_SERVICES,
#061                                     L"HelloDDK\\Zhangfan",
#062                                     paramTable,
#063                                     NULL,
#064                                     NULL);
#065     if (NT_SUCCESS(ntStatus))
#066     {
#067         KdPrint(("Query the item successfully\n"));
#068         KdPrint(("The item is :%d\n",uQueryValue));
#069     }
#070
#071     //删除子键
#072     ntStatus = RtlDeleteRegistryValue(RTL_REGISTRY_SERVICES,
#073                                     L"HelloDDK\\Zhangfan",
#074                                     L"DWORD_Value");
#075     if (NT_SUCCESS(ntStatus))
#076     {
#077         KdPrint(("delete the value successfully\n"));
#078     }

```

此段代码可以在配套光盘中本章的 RegTest 目录下找到。

6.4 小结

本章介绍了 Windows 内核模式下的一些常用内核函数, 这些函数在驱动程序的开发中将会经常用到。对这些内核函数的熟练掌握是十分必要的。本章介绍了字符串相关的一些内核函数。这些函数包括操作标准 C 语言字符串的, 也包括操作 UNICODE 字符串的。另外, 本章还介绍了与文件及注册表相关的内核函数。这些内核函数在后面的章节经常被用到。

第 7 章 派遣函数

派遣函数是 Windows 驱动程序中的重要概念。驱动程序的主要功能是负责处理 I/O 请求，其中大部分 I/O 请求是在派遣函数中处理的。

用户模式下所有对驱动程序的 I/O 请求，全部由操作系统转化为一个叫做 IRP 的数据结构，不同的 IRP 数据会被“派遣”到不同的派遣函数（Dispatch Function）中，这也是派遣函数名字的由来。本章将针对 IRP 和派遣函数进行详细的介绍。

7.1 IRP 与派遣函数

IRP 的处理机制类似 Windows 应用程序中的“消息处理”机制，驱动程序接收到不同类型的 IRP 后，会进入不同的派遣函数，在派遣函数中 IRP 得到处理。

7.1.1 IRP

在 Windows 内核中，有一种数据结构叫做 IRP（I/O Request Package），即输入输出请求包。它是与输入输出相关的重要数据结构。上层应用程序与底层驱动程序通信时，应用程序会发出 I/O 请求。操作系统将 I/O 请求转化为相应的 IRP 数据，不同类型的 IRP 会根据类型传递到不同的派遣函数内。

IRP 是一个很复杂的数据结构，后面会陆续介绍。在这里，读者需要先了解 IRP 的两个基本的属性，一个是 MajorFunction，另一个是 MinorFunction，分别记录 IRP 的主类型和子类型。操作系统根据 MajorFunction 将 IRP “派遣”到不同的派遣函数中，在派遣函数中还可以继续判断这个 IRP 属于哪种 MinorFunction。

前面介绍的 HelloDDK 和 HelloWDM 驱动程序，都是在入口函数 DriverEntry 里注册了 IRP 的派遣函数。一般来说，NT 式驱动程序和 WDM 驱动程序都是在 DriverEntry 函数中注册派遣函数的。以下是 HelloDDK 中的 DriverEntry 代码片段。

```

#001 #pragma INITCODE
#002 extern "C" NTSTATUS DriverEntry (
#003     IN PDRIVER_OBJECT pDriverObject,
#004     IN PUNICODE_STRING pRegistryPath )
#005 {
#006     NTSTATUS status;
#007     KdPrint(("Enter DriverEntry\n"));
#008
#009     //设置卸载函数
#010     pDriverObject->DriverUnload = HelloDDKUnload;
#011
#012     //设置派遣函数
#013     pDriverObject->MajorFunction[IRP_MJ_CREATE] = HelloDDKDispatchCreate;
#014     pDriverObject->MajorFunction[IRP_MJ_CLOSE] = HelloDDKDispatchClose;
#015     pDriverObject->MajorFunction[IRP_MJ_WRITE] = HelloDDKDispatchWrite;
#016     pDriverObject->MajorFunction[IRP_MJ_READ] = HelloDDKDispatchRead;
#017
#018     //创建驱动设备对象
#019     status = CreateDevice(pDriverObject);
#020
#021     KdPrint(("Leave DriverEntry\n"));
#022     return status;
#023 }

```

此段代码可以在配套光盘中本章的 DispatchTest 目录下找到。

在 DriverEntry 的驱动对象 pDriverObject 中，有个函数指针数组 MajorFunction。函数指针数组是个数组，每个元素都记录着一个函数的地址。通过设置这个数组，可以将 IRP 的类型和派遣函数关联起来。在上个例子中，只对四种类型的 IRP 设置了派遣函数，而 IRP 的类型并不只是这四种。对于其他没有设置的 IRP 类型，系统默认这些 IRP 类型与 _IopInvalidDeviceRequest 函数关联。

在进入 DriverEntry 之前，操作系统会将_IopInvalidDeviceRequest 的地址填满整个 MajorFunction 数组。IRP 与派遣函数的联系如图 7-1 所示。

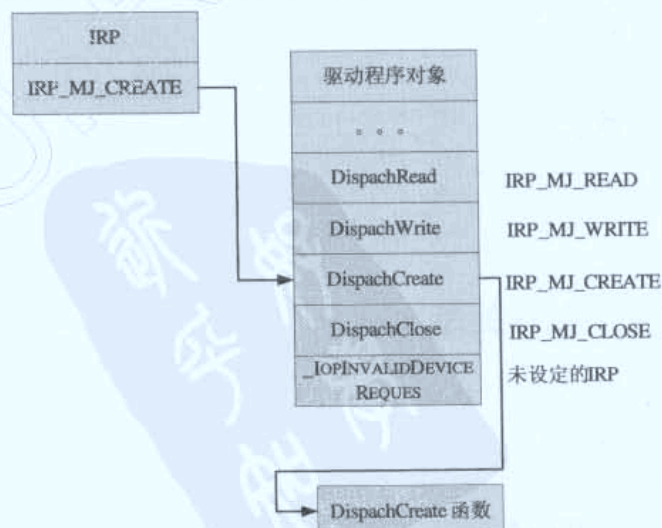


图 7-1 IRP 与派遣函数

7.1.2 IRP 类型

IRP 的概念类似 Windows 应用程序中“消息”的概念。在 Win32 编程中，程序是由“消息”驱动的。不同的消息，会被分发到不同的消息处理函数中。如果没有对应的处理函数，它会进入到系统默认的消息处理函数中。

IRP 的处理类似这种方式。文件 I/O 的相关函数，如 CreateFile、ReadFile、WriteFile、CloseHandle 等函数会使操作系统产生出 IRP_MJ_CREATE、IRP_MJ_READ、IRP_MJ_WRITE、IRP_MJ_CLOSE 等不同的 IRP，这些 IRP 会被传送到驱动程序的派遣函数中。

另外，内核中的文件 I/O 处理函数，如 ZwCreateFile、ZwReadFile、ZwWriteFile、ZwClose，它们同样会创建相应的 IRP_MJ_CREATE、IRP_MJ_READ、IRP_MJ_WRITE、IRP_MJ_CLOSE 等 IRP，并将 IRP 传送到相应驱动的相应派遣函数中。

还有些 IRP 是由系统的某个组件创建的，比如 IRP_MJ_SHUTDOWN 是在 Windows 的即插即用组件在即将关闭系统的时候发出的。表 7-1 列出了 IRP 的类型，并对其产生的来源做了说明。

表 7-1 IRP 类型

IRP 类型	来 源
IRP_MJ_CREATE	创建设备，CreateFile 会产生此 IRP
IRP_MJ_CLOSE	关闭设备，CloseHandle 会产生此 IRP
IRP_MJ_CLEANUP	清除工作，CloseHandle 会产生此 IRP
IRP_MJ_DEVICE_CONTROL	DeviceIoControl 函数会产生此 IRP
IRP_MJ_PNP	即插即用消息，NT 驱动不支持此种 IRP，只有 WDM 驱动才支持此种 IRP
IRP_MJ_POWER	在操作系统处理电源消息时，产生此 IRP
IRP_MJ_QUERY_INFORMATION	获取文件长度，GetFileSize 会产生此 IRP
IRP_MJ_READ	读取设备内容，ReadFile 会产生此 IRP
IRP_MJ_SET_INFORMATION	设置文件长度，GetFileSize 会产生此 IRP
IRP_MJ_SHUTDOWN	关闭系统前会产生此 IRP
IRP_MJ_SYSTEM_CONTROL	系统内部产生的控制信息，类似于内核调用 DeviceIoControl 函数
IRP_MJ_WRITE	对设备进行 WriteFile 时会产生此 IRP

7.1.3 对派遣函数的简单处理

大部分的 IRP 都源于文件 I/O 处理 Win32 API，如 CreateFile、ReadFile 等。处理这些 IRP 最简单的方法就是在相应的派遣函数中，将 IRP 的状态设置为成功，然后结束 IRP 的请求，并让派遣函数返回成功。结束 IRP 的请求使用函数 IoCompleteRequest。下面的代码演示了一种最简单的处理 IRP 请求的派遣函数。

```

#001 NTSTATUS HelloDDKDispatchRoutin(IN PDEVICE_OBJECT pDevObj,
#002                                     IN PIRP pIrp)
#003 {
#004     KdPrint(("Enter HelloDDKDispatchRoutin\n"));
#005     //对一般 IRP 的简单操作,后面会介绍对 IRP 更复杂的操作
#006     NTSTATUS status = STATUS_SUCCESS;
#007     //设置 IRP 完成状态
#008     pIrp->IoStatus.Status = status;
#009     //设置 IRP 操作了多少字节
#010     pIrp->IoStatus.Information = 0;    // bytes xfered
#011     //处理 IRP
#012     IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#013     KdPrint(("Leave HelloDDKDispatchRoutin\n"));
#014     return status;
#015 }

```

此段代码可以在配套光盘中本章的 DispatchTest 目录下找到。

在本例中,派遣函数设置了 IRP 的完成状态为 STATUS_SUCCESS。这样,发起 I/O 请求的 Win32 API (如 WriteFile) 将会返回 TRUE。相反,如果将 IRP 的完成状态设置为不成功,这时发起 I/O 请求的 Win32 API (如 WriteFile) 将会返回 FALSE。这种情况时,可以使用 GetLastError Win32 API 得到错误代码。所得的错误代码会和 IRP 设置的状态相一致。

除了设置 IRP 的完成状态,派遣函数还要设置这个 IRP 请求操作了多少字节。在本例中,将操作字节数简单地设置成了 0。如果是 ReadFile 产生的 IRP,这个字节数代表从设备读了多少字节。如果是 WriteFile 产生的 IRP,这个字节数代表对设备写了多少字节。最后,派遣函数将 IRP 请求结束,这是通过 IoCompleteRequest 函数完成的。IoCompleteRequest 的声明如下:

```

VOID
IoCompleteRequest(
    IN PIRP Irp,
    IN CCHAR PriorityBoost
);

```

- Irp: 代表需要被结束的 IRP。
- PriorityBoost: 代表线程恢复时的优先级别。

为了解释优先级的概念,需要了解一下与文件 I/O 相关的 Win32 API 的内部操作过程。这里以 ReadFile 为例,ReadFile 的内部操作大体是这样的:

- ① ReadFile 调用 ntdll 中的 NtReadFile。其中 ReadFile 函数是 Win32 API,而 NtReadFile 函数是 Native API。
- ② ntdll 中的 NtReadFile 进入到内核模式,并调用系统服务中的 NtReadFile 函数。
- ③ 系统服务函数 NtReadFile 创建 IRP_MJ_WRITE 类型的 IRP,然后它将这个 IRP 发送到某个驱动程序的派遣函数中。NtReadFile 然后去等待一个事件,这时当前线程进入“睡眠”状态,也可以说当前线程被阻塞住或者线程处于“Pending”状态。
- ④ 在派遣函数中一般会将 IRP 请求结束,结束 IRP 是通过 IoCompleteRequest 函数。在 IoCompleteRequest 函数内部会设置刚才等待的事件,“睡眠”的线程被恢复运行。

例如，在读一个很大的文件（或者设备）时，ReadFile 不会立刻返回，而是等待一段时间。这段时间就是当前线程“睡眠”的那段时间。IRP 请求被结束，标志这个操作执行完毕，这时“睡眠”的线程被唤醒。

IoCompleteRequest 函数中第二个参数 PriorityBoost 代表一种优先级，指的是被阻塞的线程以何种优先级恢复运行。一般情况下，优先级设置为 IO_NO_INCREMENT。对某些特殊情况，需要将阻塞的线程以“优先”的身份恢复运行。如键盘、鼠标等输入设备，它们需要更快地反应。

表 7-2 完成优先级

优先级	说 明
IO_NO_INCREMENT	不增加优先级
IO_CD_ROM_INCREMENT	光驱设备增加的优先级
IO_DISK_INCREMENT	磁盘设备增加的优先级
IO_KEYBOARD_INCREMENT	键盘设备增加的优先级
IO_MOUSE_INCREMENT	鼠标设备增加的优先级
IO_NAMED_PIPE_INCREMENT	命名管道增加的优先级
IO_NETWORK_INCREMENT	网络设备增加的优先级
IO_PARALLEL_INCREMENT	并口设备增加的优先级
IO_SERIAL_INCREMENT	串口设备增加的优先级
IO_SOUND_INCREMENT	声卡设备增加的优先级
IO_VIDEO_INCREMENT	视频设备增加的优先级
SEMAPHORE_INCREMENT	信号灯增加的优先级

本节只讨论了派遣函数中最简单的处理 IRP 的方法，在实际应用中往往要比这个复杂得多。

7.1.4 通过设备链接打开设备

本节介绍利用文件 I/O 相关的 Win32 API 对设备进行“打开”和“关闭”操作。要打开设备，必须通过设备的名字才能得到该设备的句柄。前面介绍过，每个设备都有设备名称，如 HelloDDK 驱动程序的设备名为“\Device\MyDDKDevice”，但是设备名无法被用户模式下的应用程序查询到，设备名只能被内核模式下的程序查询到。

在应用程序中，设备可以通过符号链接进行访问。驱动程序通过 IoCreateSymbolicLink 函数创建符号链接。HelloDDK 驱动程序的设备所对应的符号链接是“\??\HelloDDK”。在编写程序时，符号链接的写法需要稍微改一下，将前面的“\??\”改为“\\.”。因此符号链接“\??\HelloDDK”就变成了“\\.\HelloDDK”，写成 C 语言的字符串就是“\\\\.\HelloDDK”。

下面的代码演示了如何利用 CreateFile 来打开设备句柄，以及如何利用 CloseHandle 关闭设备句柄。在打开和关闭设备句柄的时候，操作系统内部会创建 IRP，并将 IRP 发送

到相应的派遣函数中。

```
#001 #include <windows.h>
#002 #include <stdio.h>
#003 int main()
#004 {
#005     //打开设备句柄, 会触发 IRP_MJ_CREATE
#006     HANDLE hDevice =
#007         CreateFile("\\\\.\\HelloDDK",
#008             GENERIC_READ | GENERIC_WRITE,
#009             0,           // 非共享
#010             NULL,       // 没有使用安全描述符
#011             OPEN_EXISTING,
#012             FILE_ATTRIBUTE_NORMAL,
#013             NULL ); // 没有模板
#014     //判断设备是否成功打开
#015     if (hDevice == INVALID_HANDLE_VALUE)
#016     {
#017         printf("Failed to obtain file handle to device: "
#018             "%s with Win32 error code: %d\n",
#019             "MyWDMDevice", GetLastError() );
#020         return 1;
#021     }
#022
#023     //关闭设备句柄, 会触发 IRP_MJ_CLEANUP 和 IRP_MJ_CLOSE
#024     CloseHandle(hDevice);
#025     return 0;
#026 }
```

此段代码可以在配套光盘中本章的 DispatchTest 目录下找到。

7.1.5 编写一个更通用的派遣函数

前面介绍的派遣函数处理过于简单, 下面笔者带领读者对派遣函数一步步进行扩充。首先介绍一个重要数据结构——IO_STACK_LOCATION, 即 I/O 堆栈, 这个数据结构和 IRP 紧密相连。

在第4章中, 笔者曾经介绍过驱动程序的层次结构。驱动对象会创建一个个的设备对象, 并将这些设备对象“叠”成一个垂直结构。这种垂直的结构很像栈, 因此被称为“设备栈”。

IRP 会被操作系统发送到设备栈的顶层, 如果顶层的设备对象的派遣函数结束了 IRP 的请求, 则这次 I/O 请求结束。如果没有将 IRP 的请求结束, 那么操作系统将 IRP 转发到设备栈的下一层设备处理。如果这个设备的派遣函数依然不能结束 IRP 请求, 则会继续向下层设备转发。

因此, 一个 IRP 可能会被转发多次。为了记录 IRP 在每层设备中做的操作, IRP 会有一个 IO_STACK_LOCATION 数组。数组的元素数应该大于 IRP 穿越过的设备数。每个 IO_STACK_LOCATION 元素记录着对应设备中做的操作。对于本层设备对应的 IO_STACK_LOCATION, 可以通过 IoGetCurrentIrpStackLocation 函数得到, 如:

Windows 驱动开发技术详解

```
PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
```

IO_STACK_LOCATION 结构中会记录 IRP 的类型，即 IO_STACK_LOCATION 中的 MajorFunction 子域。

下面的代码增加了派遣函数的难度，演示了派遣函数如何获得当前 IO_STACK_LOCATION，以及如何获得 IRP 的类型。在 DriverEntry 中将所有的 IRP 类型都和一个派遣函数相关联，下面是 DriverEntry 中的代码片段：

```
#001 pDriverObject->MajorFunction[IRP_MJ_CREATE] = HelloDDKDispatchRoutin;
#002 pDriverObject->MajorFunction[IRP_MJ_CLOSE] = HelloDDKDispatchRoutin;
#003 pDriverObject->MajorFunction[IRP_MJ_WRITE] = HelloDDKDispatchRoutin;
#004 pDriverObject->MajorFunction[IRP_MJ_READ] = HelloDDKDispatchRoutin;
#005 pDriverObject->MajorFunction[IRP_MJ_CLEANUP] = HelloDDKDispatchRoutin;
#006 pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = HelloDDKDispatch
Routin;
#007 pDriverObject->MajorFunction[IRP_MJ_SET_INFORMATION] = HelloDDKDispatch
Routin;
#008 pDriverObject->MajorFunction[IRP_MJ_SHUTDOWN] = HelloDDKDispatchRoutin;
#009 pDriverObject->MajorFunction[IRP_MJ_SYSTEM_CONTROL] = HelloDDKDispatch
Routin;
```

此段代码可以在配套光盘中本章的 DispatchTest 目录下找到。

然后在派遣函数中分辨出是 IRP 的类型，并打出相应的 log 信息。

```
#001 NTSTATUS HelloDDKDispatchRoutin(IN PDEVICE_OBJECT pDevObj,
#002                                  IN PIRP pIrp)
#003 {
#004     KdPrint(("Enter HelloDDKDispatchRoutin\n"));
#005
#006     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
#007     //建立一个字符串数组与 IRP 类型对应起来
#008     static char* irpname[] =
#009     {
#010         "IRP_MJ_CREATE",
#011         "IRP_MJ_CREATE_NAMED_PIPE",
#012         "IRP_MJ_CLOSE",
#013         "IRP_MJ_READ",
#014         "IRP_MJ_WRITE",
#015         "IRP_MJ_QUERY_INFORMATION",
#016         "IRP_MJ_SET_INFORMATION",
#017         "IRP_MJ_QUERY_EA",
#018         "IRP_MJ_SET_EA",
#019         "IRP_MJ_FLUSH_BUFFERS",
#020         "IRP_MJ_QUERY_VOLUME_INFORMATION",
#021         "IRP_MJ_SET_VOLUME_INFORMATION",
#022         "IRP_MJ_DIRECTORY_CONTROL",
#023         "IRP_MJ_FILE_SYSTEM_CONTROL",
#024         "IRP_MJ_DEVICE_CONTROL",
#025         "IRP_MJ_INTERNAL_DEVICE_CONTROL",
#026         "IRP_MJ_SHUTDOWN",
#027         "IRP_MJ_LOCK_CONTROL",
#028         "IRP_MJ_CLEANUP",
#029         "IRP_MJ_CREATE_MAILSLOT",
#030         "IRP_MJ_QUERY_SECURITY",
#031         "IRP_MJ_SET_SECURITY",
#032         "IRP_MJ_POWER",
```

```

#033     "IRP_MJ_SYSTEM_CONTROL",
#034     "IRP_MJ_DEVICE_CHANGE",
#035     "IRP_MJ_QUERY_QUOTA",
#036     "IRP_MJ_SET_QUOTA",
#037     "IRP_MJ_PNP",
#038     };
#039
#040     UCHAR type = stack->MajorFunction;
#041     if (type >= arraysize(irpname))
#042         KdPrint((" - Unknown IRP, major type %X\n", type));
#043     else
#044         KdPrint(("\\t%s\n", irpname[type]));
#045     //对一般 IRP 的简单操作, 后面会介绍对 IRP 更复杂的操作
#046     NTSTATUS status = STATUS_SUCCESS;
#047     // 设置 IRP 完成状态
#048     pIrp->IoStatus.Status = status;
#049     //设置 IRP 操作字节数
#050     pIrp->IoStatus.Information = 0;
#051     //结束 IRP 请求
#052     IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#053     KdPrint(("Leave HelloDDKDispatchRoutin\n"));
#054     return status;
#055 )

```

此段代码可以在配套光盘中本章的 DispatchTest 目录下找到。

将驱动程序成功加载后, 执行应用程序。应用程序就是上节介绍的“打开”和“关闭”设备。随后用 Dbgview 查看驱动程序输出的 log 信息。可以发现, 依次进入派遣函数的是 IRP_MJ_CREATE、IRP_MJ_CLEANUP 和 IRP_MJ_CLOSE。图 7-2 列出了用 Dbgview 输出的 log 信息。

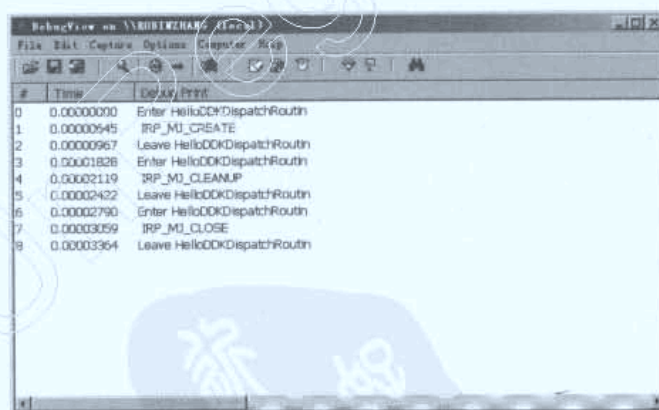


图 7-2 Dbgview 输出的 log 信息

7.1.6 跟踪 IRP 的利器 IRPTrace

IRP 是驱动程序中重要的数据结构, 可以说驱动程序的运行是由 IRP 所“驱动”的。在驱动程序中, 仅凭查看 log 信息有时候是不能满足调试需要的, 程序员往往需要更直观地跟踪 IRP 的传递、转发、结束等操作。

有时候 IRP 的处理非常复杂,跟踪 IRP 显得尤为重要。笔者这里介绍一个工具软件 IRPTrace,这个软件可以方便地跟踪 IRP 的各种操作。读者可以从网上下载试用版或购买完整版。以下简单介绍一下该软件的使用方法。

(1) IRPTrace 界面

这里以介绍跟踪 HelloDDK 中的 IRP 为例。首先确保已经成功加载了 HelloDDK 驱动程序,然后打开 IRPTrace,它的界面如图 7-3 所示。

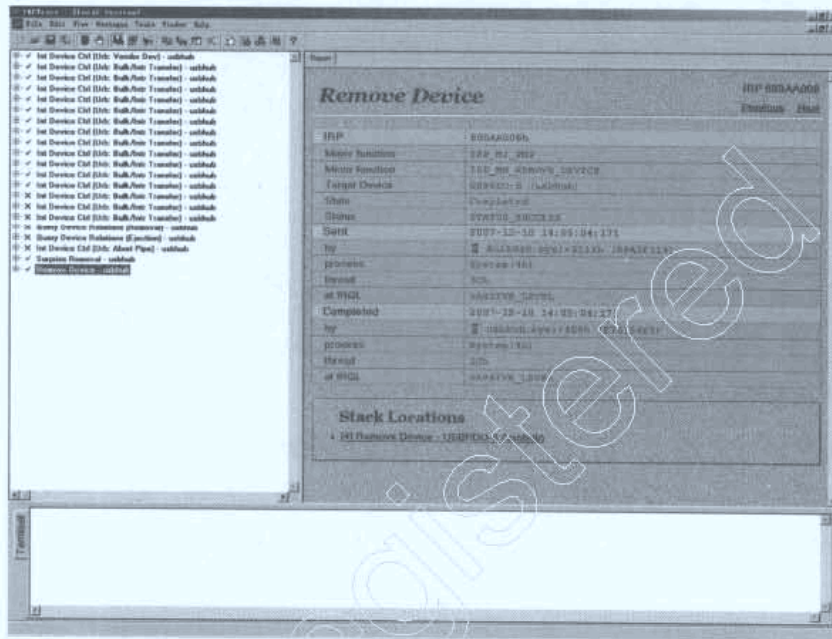




图 7-3 IRPTrace

(2) 用 IRPTrace 跟踪各类 IRP

选择图标 , 或者选择菜单 “Message” | “Hook Setup”。然后会弹出一个窗口,如图 7-4 所示。选中左边的 “Drivers” 选项卡,程序会枚举出系统加载的所有驱动程序。在本例中,选中驱动程序 HelloDDK。这时,右面会列出需要跟踪的 IRP 类型,选择 “ALL”。

选择图标 , 或者选择菜单 “Messages” | “Clear IRP log”, 将前面的 log 信息清除。至此,已经做好了跟踪前的准备。下面所要做的就是执行 7.1.4 小节中的应用程序,IRPTrace 会自动跟踪 IRP。

(3) 用 IRPTrace 观察 IRP 数据结构中的各项内容

将 IRPTrace 程序最小化,执行 7.1.4 小节中的应用程序。然后切换到 IRPTrace 程序中,会发现有三个 IRP 被跟踪下来,分别是 IRP_MJ_CREATE、IRP_MJ_CLEANUP 和 IRP_MJ_CLOSE,这和预期的完全一致。程序左边列出了跟踪到的三个 IRP,右边会列出具体的 IRP 跟踪信息。如图 7-5 所示。

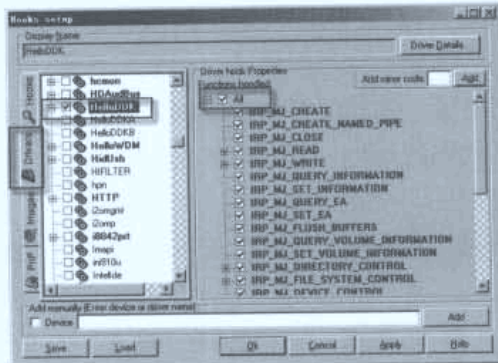


图 7-4 IRPTrace 中选择驱动

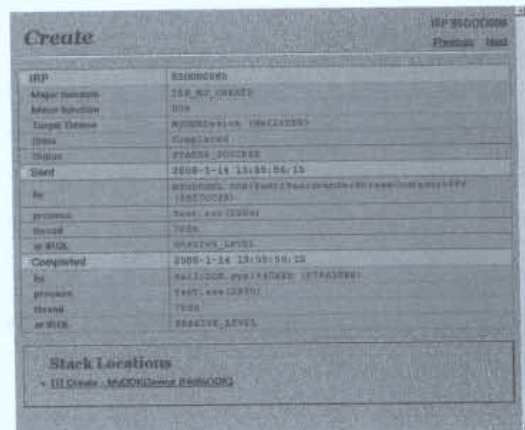


图 7-5 跟踪 IRP_MJ_CREATE

从图 7-5 中可以查看 IRP_MJ_CREATE 的详细信息。首先程序列出 IRP 数据结构的指针该地址是内核模式下的地址，所以是 4GB 空间中的高 2GB 部分，紧接着列出了 IRP 的主类型和子类型。然后列出的是 Target Device，这是 IRP 发送到设备栈中遇到的第一个设备。然后列出的是 IRP 的完成状态，这个例子中的完成状态是 STATUS_SUCCESS。然后列出的是该 IRP 是出自于哪个进程和哪个线程 在本例中 IRP 是出自 Test.exe 进程中的 0x7f8 号线程。

(4) 观察 IRP 的每层 I/O 堆栈

最后我们还能查看 IRP 请求是如何被结束的，本例中 IRPTrace 指出 IRP 是在 HelloDDK 驱动程序中被结束的，位置是 HelloDDK.sys!+40EEh。有兴趣的读者可以对 HelloDDK 进行反汇编，可以看出调用 IoCompleteRequest 的位置恰恰就是 40EEh 位置。

除了查看 IRP 相关信息，IRPTrac 还可以查看 IO_STACK_LOCATION 信息。在图 7-5 中单击 Stack Locations 的超链接，得到如图 7-6 所示的信息。

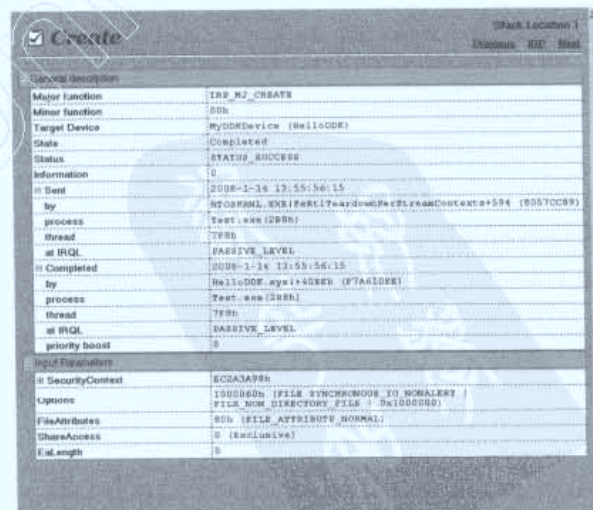



图 7-6 跟踪堆栈

(5) 设置符号表

需要指出的是, IRP 请求是在 HelloDDK.sys!+40EEh 的位置被结束的, 但很难知道 HelloDDK.sys!+40EEh 这个位置对应源代码的什么位置。如果想将这个地址与源程序中的位置对应起来查看, 需要正确设置驱动程序的符号表。微软的编译器在编译程序时会生成符号表文件, 它可以将行号、变量名等和二进制代码的偏移位置对应起来。符号表一般存储在 pdb 文件中。驱动程序被编译后, 都会伴随产生一个相应的 pdb 文件。

选择“Tools”|“Terminal Options”|“Symbols”, 出现如图 7-7 所示的界面。将此驱动对应的 pdb 的路径加入到符号表路径中。

这样就完成了对符号表的配置。HelloDDK.sys!+40EEh 被替换成了 HelloDDK.sys!HelloDDKDispatchRoutin+8D (F7A610EE)。

另外, IRPTrace 还可以在系统启动时就进行跟踪。选择图标 , 或者选择菜单“Tools”|“IRPTrace Driver Control”, 会弹出一个对话框如图 7-8 所示。如果需要在系统启动时就跟踪 IRP, 请选择“Startup Type”下拉菜单中“Boot”。

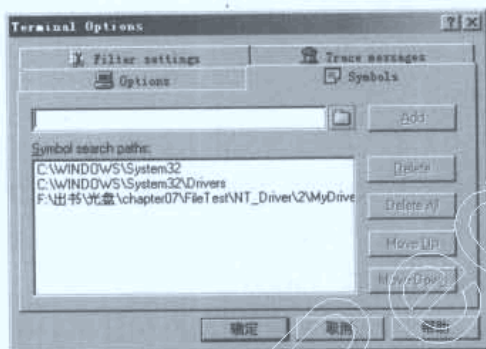


图 7-7 选择符号表

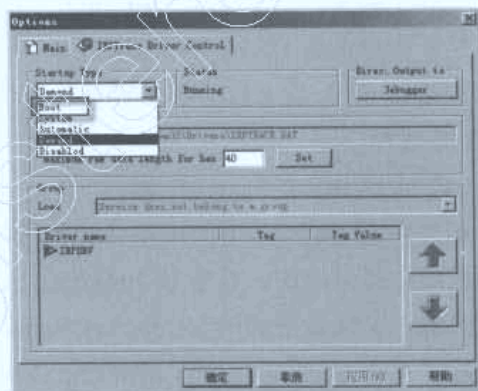


图 7-8 选择跟踪加载时机

IRPTrace 会非常详细的列出 IRP 的处理过程, 这有利于程序员跟踪 IRP。以后笔者还会介绍更多的跟踪工具, 利用这些工具, 程序员既可以方便的跟踪调试驱动, 同时又可以对内核的了解。

7.2 缓冲区方式读写操作

驱动程序所创建的设备一般会有三种读写方式, 一种是缓冲区方式, 一种是直接方式, 一种是其他方式。本节主要介绍缓冲区方式的读写。

7.2.1 缓冲区设备

在驱动程序创建设备对象的时候, 需要考虑好该设备是采用何种读写方式。当

IoCreateDevice 创建完设备后，需要对设备对象的 Flags 子域进行设置。设置不同的 Flags，会导致以不同的方式操作设备。

在以前的 HelloDDK 中是这样设置的：

```
#001 //创建设备
#002 status = IoCreateDevice( pDriverObject,
#003                          sizeof(DEVICE_EXTENSION),
#004                          &(UNICODE_STRING)devName,
#005                          FILE_DEVICE_UNKNOWN,
#006                          0, TRUE,
#007                          &pDevObj );
#008 //判断是否创建成功
#009 if (!NT_SUCCESS(status))
#010     return status;
#011 //设置读写方式
#012 pDevObj->Flags |= DO_BUFFERED_IO;
```

此段代码可以在配套光盘中本章的 ReadTest 目录下找到。

设备对象一共可以有三种读写方式，分别是缓冲区方式读写、直接方式读写、其他方式读写。这三种方式的 Flags 分别对应为 DO_BUFFERED_IO、DO_DIRECT_IO 和 0。缓冲区方式读写相对简单，本节先介绍缓冲区方式读写。

读写操作一般是由 ReadFile 或者 WriteFile 函数引起的，这里先以 WriteFile 函数为例进行介绍。WriteFile 要求用户提供一段缓冲区，并且说明缓冲区的大小，然后 WriteFile 将这段内存的数据传入到驱动程序中。

这段缓冲区内存是用户模式的内存地址，驱动程序如果直接引用这段内存是十分危险的。因为 Windows 操作系统是多任务的，它可能随时切换到别的进程。如果驱动程序需要访问这段内存，而这时操作系统可能已经切换到另外一个进程。如果这样，驱动程序访问的内存地址必定是错误的，这种错误会引起系统崩溃。

举个例子，进程 A 将 0x4000 地址传送到驱动程序中，而这时系统已经切换到进程 B。而驱动程序继续访问 0x4000 地址，而这时的 0x4000 地址是进程 B 的而不是进程 A 的地址。这肯定会引起非常严重的错误。

有很多方法可以解决这个问题，其中一个方法是使用缓冲区方式读写。对于这种方法，操作系统将应用程序提供缓冲区的数据复制到内核模式下的地址中。这样，无论操作系统如何切换进程，内核模式地址都不会改变。IRP 的派遣函数将会对内核模式下的缓冲区操作，而不是操作用户模式地址的缓冲区。

这样做的优点是，比较简单地解决了将用户地址传入驱动的问题。缺点是需要用户在用户模式和内核模式之间复制数据，影响了运行效率。在少量内存操作时，可以采用这种办法。

7.2.2 缓冲区设备读写

以缓冲区方式写设备时，操作系统将 WriteFile 提供的用户模式的缓冲区复制到内核模

式地址下。这个地址由 WriteFile 创建的 IRP 的 AssociatedIrp.SystemBuffer 子域记录。

以“缓冲区”方式读设备时，操作系统会分配一段内核模式下的内存。这段内存大小等于 ReadFile 或者 WriteFile 指定的字节数，并且 ReadFile 或者 WriteFile 创建的 IRP 的 AssociatedIrp.SystemBuffer 子域会记录这段内存地址。当 IRP 请求结束时（一般都是由 IoCompleteRequest 函数结束 IRP），这段内存地址会被复制到 ReadFile 提供的缓冲区中。

以缓冲区方式无论是“读”还是“写”设备，都会发生用户模式地址与内核模式地址的数据复制。复制的过程由操作系统负责。用户模式地址由 ReadFile 或者 WriteFile 提供，内核模式地址由操作系统负责分配和回收。

另外，在派遣函数中，也可以通过 IO_STACK_LOCATION 中的 Parameters.Read.Length 子域知道 ReadFile 请求多少字节。通过 IO_STACK_LOCATION 中的 Parameters.Write.Length 子域知道 WriteFile 请求多少字节。

然而，WriteFile 和 ReadFile 指定对设备操作多少字节，并不真正意味着操作了这么多字节。在派遣函数中，应该设置 IRP 的子域 IoStatus.Information。这个子域记录设备实际操作了多少字节。

ReadFile 和 WriteFile 分别通过各自的第四个参数得到真实操作了多少字节。

下面的代码演示了如何利用“缓冲区”方式读设备。本例的驱动程序返回给应用程序的数据都是 0XAA，因此应用程序会从设备中读到一连串的 0XAA。

其中，驱动程序中的派遣函数是这样的：

```
#001 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,  
#002                        IN PIRP pIrp)  
#003 {  
#004     KdPrint(("Enter HelloDDKRead\n"));  
#005  
#006     //对一般 IRP 的简单操作，后面会介绍对 IRP 更复杂的操作  
#007     NTSTATUS status = STATUS_SUCCESS;  
#008     //得到当前堆栈  
#009     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);  
#010     //得到需要读设备的字节数  
#011     ULONG ulReadLength = stack->Parameters.Read.Length;  
#012     // 完成 IRP  
#013     //设置 IRP 完成状态  
#014     pIrp->IoStatus.Status = status;  
#015     //设置 IRP 操作了多少字节  
#016     pIrp->IoStatus.Information = ulReadLength; // bytes xfered  
#017     //设置内核模式下的缓冲区  
#018     memset(pIrp->AssociatedIrp.SystemBuffer, 0xAA, ulReadLength);  
#019     //处理 IRP  
#020     IoCompleteRequest(pIrp, IO_NO_INCREMENT);  
#021     KdPrint(("Leave HelloDDKRead\n"));  
#022     return status;  
#023 }
```

此段代码可以在配套光盘中本章的 ReadTest 目录下找到。

应用程序使用 ReadFile 对设备进行读写：

```

#001 #include <windows.h>
#002 #include <stdio.h>
#003 int main()
#004 {
#005     //打开设备句柄
#006     HANDLE hDevice =
#007         CreateFile("\\\\.\\HelloDDK",
#008                 GENERIC_READ | GENERIC_WRITE,
#009                 0,
#010                 NULL,
#011                 OPEN_EXISTING,
#012                 FILE_ATTRIBUTE_NORMAL,
#013                 NULL );
#014     //判断是否成功打开设备
#015     if (hDevice == INVALID_HANDLE_VALUE)
#016     {
#017         printf("Failed to obtain file handle to device: "
#018               "%s with Win32 error code: %d\n",
#019               "MyWDMDevice", GetLastError() );
#020         return 1;
#021     }
#022
#023     UCHAR buffer[10];
#024     ULONG ulRead;
#025     //对设备读写
#026     BOOL bRet = ReadFile(hDevice,buffer,10,&ulRead,NULL);
#027     if (bRet)
#028     {
#029         printf("Read %d bytes:",ulRead);
#030         for (int i=0;i<(int)ulRead;i++)
#031         {
#032             printf("%02X ",buffer[i]);
#033         }
#034
#035         printf("\n");
#036     }
#037     //关闭设备句柄
#038     CloseHandle(hDevice);
#039     return 0;
#040 }

```

此段代码可以在配套光盘中本章的 ReadTest 目录下找到。

运行结果如图 7-9 所示。



图 7-9 运行结果

另外，可以用 IRP Trace 工具查看设备对象对应的 IO_STACK_LOCATION 结构，如图 7-10 所示。从图 7-10 中可以看出，IRP 请求是在 HelloDDK.sys!HelloDDKRead 中被结束的。一共读取 10 个字节，每个字节都是 0xAA。

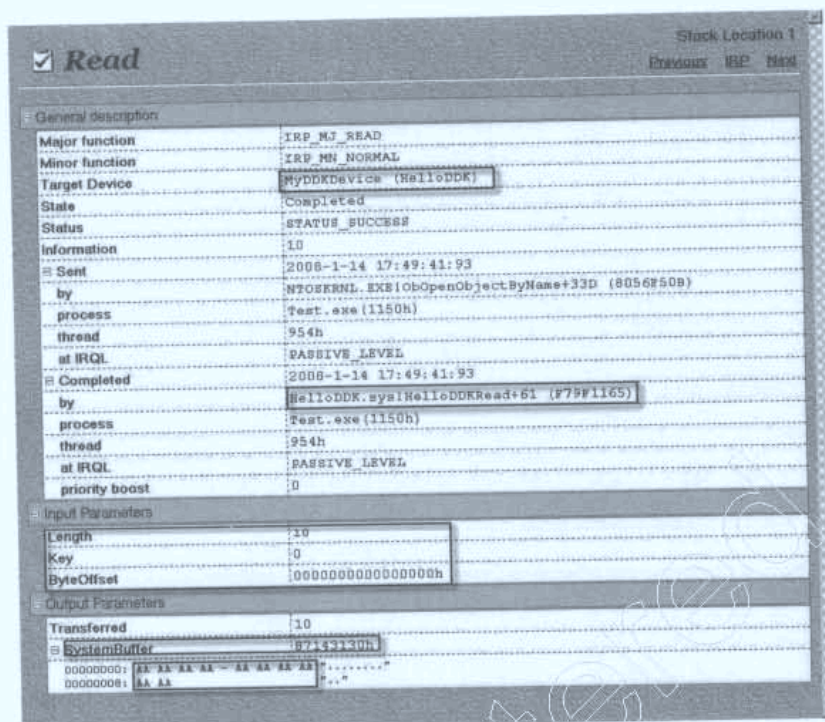


图 7-10 IRPTrace 跟踪结果

7.2.3 缓冲区设备模拟文件读写

相信读者已经明白了缓冲区方式的读写操作，下面根据这部分知识，来编写一个虚拟设备。这个设备来模拟一个文件，可以将这个设备想象成一个普通文件，可以进行读写操作。另外，每次写这个文件，文件的长度会增加，可以利用 GetFileSize 函数（API 函数）得到该文件的长度。

为了实现这个目的，需要编写三个 IRP 的派遣函数。它们分别对应着写操作、读操作、获取文件长度操作。下面分别进行介绍。

(1) 写操作。在应用程序中，通过 WriteFile 函数对设备进行写操作。下面程序片段是应用程序分配 10 字节的缓冲区，然后将缓冲区填写 0xBB，然后将这 10 个字节写入设备。

```
#001  UCHAR buffer[10];
#002  memset(buffer, 0xBB, 10);
#003  ULONG ulRead;
#004  ULONG ulWrite;
#005  BOOL bRet;
#006  //对设备写操作
#007  bRet = WriteFile(hDevice, buffer, 10, &ulWrite, NULL);
#008  if (bRet)
#009  {
#010      printf("Write %d bytes\n", ulWrite);
#011  }
```

此段代码可以在配套光盘中本章的 ReadTest 目录下找到。

WriteFile 内部会创建产生 IRP_MJ_WRITE 类型的 IRP，操作系统会将这个 IRP 传递给驱动程序。IRP_MJ_WRITE 的派遣函数需要将传送进来的数据保存起来，以便读取该设备的时候读取。在本例中，这个数据存储在一个缓冲区中，缓冲区的地址记录在设备扩展中。在设备启动的时候，驱动程序负责分配这个缓冲区，在设备被卸载的时候，驱动程序回收该缓冲区。

对于 IRP_MJ_WRITE 的派遣函数，主要任务是将写入的数据存储在这段缓冲区中。如果写入的字节数过大，超过缓冲区的大小，派遣函数将 IRP 的状态设置成错误状态。另外，在设备扩展中有一个变量记录着这个虚拟文件设备的文件长度。对设备的写操作会更改这个变量。

```
#001 NTSTATUS HelloDDWrite(IN PDEVICE_OBJECT pDevObj,
#002                        IN PIRP pIrp)
#003 {
#004     KdPrint(("Enter HelloDDWrite\n"));
#005     NTSTATUS status = STATUS_SUCCESS;
#006     PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
#007     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
#008     //获取存储的长度
#009     ULONG ulWriteLength = stack->Parameters.Write.Length;
#010     //获取存储的偏移量
#011     ULONG ulWriteOffset = (ULONG)stack->Parameters.Write.ByteOffset.QuadPart;
#012
#013     if (ulWriteOffset+ulWriteLength>MAX_FILE_LENGTH)
#014     {
#015         //如果存储长度+偏移量大于缓冲区长度，则返回无效
#016         status = STATUS_FILE_INVALID;
#017         ulWriteLength = 0;
#018     }else
#019     {
#020         //将写入的数据，存储在缓冲区内
#021         memcpy(pDevExt->buffer+ulWriteOffset,pIrp->AssociatedIrp.
SystemBuffer,ulWriteLength);
#022         status = STATUS_SUCCESS;
#023         //设置新的文件长度
#024         if (ulWriteLength+ulWriteOffset>pDevExt->file_length)
#025         {
#026             pDevExt->file_length = ulWriteLength+ulWriteOffset;
#027         }
#028     }
#029     pIrp->IoStatus.Status = status;           //设置 IRP 的完成状态
#030     pIrp->IoStatus.Information = ulWriteLength; //实际操作多少字节
#031     IoCompleteRequest( pIrp, IO_NO_INCREMENT ); //将 IRP 请求结束
#032     KdPrint(("Leave HelloDDWrite\n"));
#033     return status;
#034 }
```

此段代码可以在配套光盘中本章的 ReadTest 目录下找到。

(2) 读操作。在应用程序中，通过 ReadFile 从设备读取数据，代码如下。

```
#001 bRet = ReadFile(hDevice,buffer,10,&ulRead,NULL); //从设备读取 10 个字节
#002 if (bRet)
```


Windows 驱动开发技术详解

```
#003 {
#004     printf("Read %d bytes:",ulRead);
#005     //显示读取的数据
#006     for (int i=0;i<(int)ulRead;i++)
#007     {
#008         printf("%02X ",buffer[i]);
#009     }
#010     printf("\n");
#011 }
```

此段代码可以在配套光盘中本章的 ReadTest 目录下找到。

ReadFile 内部会创建 IRP_MJ_READ 类型的 IRP，操作系统会将这个 IRP 传递给驱动程序中 IRP_MJ_READ 的派遣函数。IRP_MJ_READ 的派遣函数的主要任务是把记录的数据复制到 AssociatedIrp.SystemBuffer 中。

```
#001 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#002                        IN PIRP pIrp)
#003 {
#004     KdPrint(("Enter HelloDDKRead\n"));
#005     PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
#006     NTSTATUS status = STATUS_SUCCESS;
#007     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
#008     ULONG ulReadLength = stack->Parameters.Read.Length;
#009     ULONG ulReadOffset = (ULONG)stack->Parameters.Read.ByteOffset.QuadPart;
#010     if (ulReadOffset+ulReadLength>MAX_FILE_LENGTH)
#011     {
#012         status = STATUS_FILE_INVALID;
#013         ulReadLength = 0;
#014     }else
#015     {
#016         //将数据存储在 AssociatedIrp.SystemBuffer，以便应用程序使用
#017         memcpy(pIrp->AssociatedIrp.SystemBuffer,pDevExt->buffer+ulReadOffset,
ulReadLength);
#018         status = STATUS_SUCCESS;
#019     }
#020     //设置 IRP 完成状态
#021     pIrp->IoStatus.Status = status;
#022     //设置 IRP 操作字节数
#023     pIrp->IoStatus.Information = ulReadLength;
#024     //结束 IRP
#025     IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#026     KdPrint(("Leave HelloDDKRead\n"));
#027     return status;
#028 }
```

此段代码可以在配套光盘中本章的 ReadTest 目录下找到。

(3) 读取文件长度。读取文件长度依靠 GetFileSize Win32API 获得。GetFileSize 内部会创建 IRP_MJ_QUERY_INFORMATION 类型的 IRP。这个 IRP 请求的作用是向设备查询一些信息，这包括查询文件长度、设备创建的时间、设备属性等。在本例中，IRP_MJ_QUERY_INFORMATION 派遣函数的主要任务是告诉应用程序这个设备的长度。

IRP_MJ_QUERY_INFORMATION 可以获得不同的信息，每种信息对应一种类别号。该类别号是一个 FILE_INFORMATION_CLASS 类型的枚举变量，可以读取 I/O 堆栈的

Parameters.QueryFile.FileInformationClass 子域。查询设备长度时，也就是调用 Win32 API GetFileSize 时，Parameters.QueryFile.FileInformationClass 应该为 FileStandard Information。

这时，IRP 的缓冲区数据为 FILE_STANDARD_INFORMATION 数据结构的数据。

```
typedef struct _FILE_STANDARD_INFORMATION {
    LARGE_INTEGER AllocationSize;
    LARGE_INTEGER EndOfFile;
    ULONG NumberOfLinks;
    BOOLEAN DeletePending;
    BOOLEAN Directory;
} FILE_STANDARD_INFORMATION, *PFILE_STANDARD_INFORMATION;
```

其中，EndOfFile 子域指明设备长度，修改这个子域会在 GetFileSize 的返回值中得到体现，IRP_MJ_QUERY_INFORMATION 派遣函数如下。

```
#001 NTSTATUS HelloDDKQueryInfomation(IN PDEVICE_OBJECT pDevObj,
#002                                     IN PIRP pIrp)
#003 {
#004     KdPrint(("Enter HelloDDKQueryInfomation\n"));
#005     //获得 IO 堆栈
#006     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
#007     //获得设备扩展
#008     PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
#009     //得到文件信息
#010     FILE_INFORMATION_CLASS info = stack->Parameters.QueryFile.FileInformation
Class;
#011     //判断是否是标准文件信息
#012     if (info==FileStandardInformation)
#013     {
#014         KdPrint(("FileStandardInformation\n"));
#015         PFILE_STANDARD_INFORMATION file_info =
#016             (PFILE_STANDARD_INFORMATION)pIrp->AssociatedIrp.SystemBuffer;
#017         file_info->EndOfFile = RtlConvertLongToLargeInteger(pDevExt->
file_length);
#018     }
#019     NTSTATUS status = STATUS_SUCCESS;
#020     //设置 IRP 完成状态
#021     pIrp->IoStatus.Status = status;
#022     //设置 IRP 操作字节数
#023     pIrp->IoStatus.Information = stack->Parameters.QueryFile.Length;
#024     //结束 IRP 请求
#025     IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#026     KdPrint(("Leave HelloDDKQueryInfomation\n"));
#027     return status;
#028 }
```

此段代码可以在配套光盘中本章的 ReadTest 目录下找到。

7.3 直接方式读写操作

上一节介绍了对设备的缓冲区方式读写，本节将介绍对设备的直接方式读写。

7.3.1 直接读取设备

除了“缓冲区”方式读写设备外，另外一种方式是直接方式读写设备。这种方式需要创建完设备对象后，在设置设备属性的时候，设置为 `DO_DIRECT_IO`，而不是设置 `DO_BUFFERED_IO` 属性，如：

```
#001 //创建设备
#002 status = IoCreateDevice( pDriverObject,
#003                          sizeof(DEVICE_EXTENSION),
#004                          &(UNICODE_STRING)devName,
#005                          FILE_DEVICE_UNKNOWN,
#006                          0, TRUE,
#007                          &pDevObj );
#008 if (!NT_SUCCESS(status))
#009     return status;
#010 pDevObj->Flags |= DO_DIRECT_IO;
```

此段代码可以在配套光盘中本章的 MDL_Test 目录下找到。

在应用程序进行读写的时候，例如，用 `WriteFile` 写设备时，会要求用户提供一段缓冲区，这段缓冲区里面是要写入设备的数据。操作系统在创建 `IRP_MJ_WRITE` 的时候，将需要写入的数据复制到 `IRP` 数据结构中的 `AssociatedIrp.SystemBuffer` 中，对于读设备操作也是类似的。

和缓冲区方式读写设备不同，直接方式读写设备，操作系统会将用户模式下的缓冲区锁住。然后操作系统将这段缓冲区在内核模式地址再次映射一遍。这样，用户模式的缓冲区和内核模式的缓冲区指向的是同一区域的物理内存。无论操作系统如何切换进程，内核模式地址都保持不变。

操作系统先将用户模式的地址锁定后，操作系统用内存描述符表（MDL 数据结构）记录这段内存。如图 7-11 所示，用户模式的这段缓冲区在虚拟内存上是连续的，但是在物理内存上可能是离散的。

MDL 记录这段虚拟内存，这段虚拟内存的大小存储在 `mdl->ByteCount` 里，这段虚拟内存的第一个页地址是 `mdl->StartVa`，这段虚拟内存的首地址对于第一个页地址的偏移量是 `mdl->ByteOffset`。因此，这段虚拟内存的首地址应该是 `mdl->StartVa+mdl->ByteOffset`。DDK 提供了几个宏方便程序员得到这几个数值。

```
#define MmGetMdlByteCount(Mdl) ((Mdl)->ByteCount)

#define MmGetMdlByteOffset(Mdl) ((Mdl)->ByteOffset)

#define MmGetMdlVirtualAddress(Mdl) \
    ((PVOID) ((PCHAR) ((Mdl)->StartVa) + (Mdl)->ByteOffset))
```

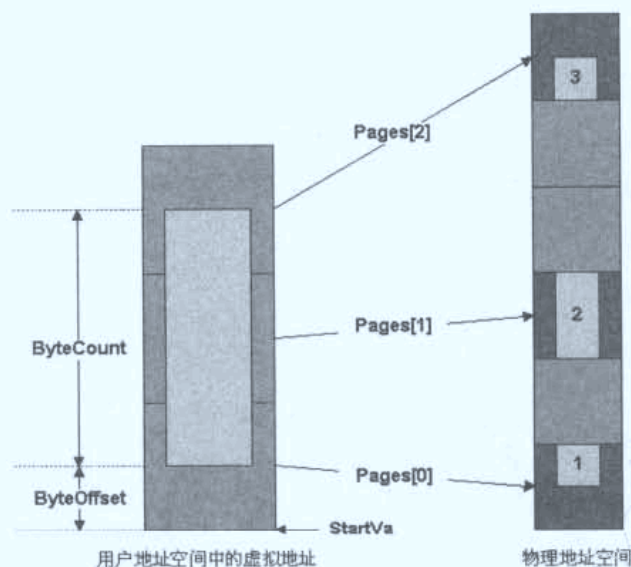


图 7-11 MDL 示意图

7.3.2 直接读取设备的读写

前面介绍了直接方式读写设备的原理，下面结合代码演示如何编写直接方式设备的派遣函数。本例演示了“IRP_MJ_READ”的派遣例程，读者可以根据这个例子写出“IRP_MJ_WRITE”的派遣例程。

应用程序调用 Win32 API ReadFile，操作系统将 IRP_MJ_READ 转发到相应的派遣函数中。派遣函数通过读取 I/O 堆栈的 stack->Parameters.Read.Length 来获取这次读取的长度。这个长度实际上就是 ReadFile 函数的第三个参数 nNumberOfBytesToRead。

派遣函数设置 pIrp->IoStatus.Information 告诉 ReadFile 实际读取了多少字节。这个数值对应着 ReadFile 的第 4 个参数。

```

BOOL ReadFile(
    HANDLE hFile,           // 文件句柄
    LPVOID lpBuffer,        // 缓冲区
    DWORD nNumberOfBytesToRead, // 希望读的字节数
    LPDWORD lpNumberOfBytesRead, // 实际读的字节数
    LPOVERLAPPED lpOverlapped // overlap 数据结构地址
);

```

通过 IRP 的 pIrp->MdlAddress 得到 MDL 数据结构，这个结构描述了被锁定的缓冲区内存。通过 DDK 的三个宏 MmGetMdlByteCount、MmGetMdlVirtualAddress、MmGetMdlByteOffset 可以得到锁定缓冲区的长度、虚拟内存地址、偏移量。

如果派遣函数的返回值是 STATUS_SUCCESS，则 ReadFile 返回 TRUE，表明读操作成功。如果派遣函数返回的不是 STATUS_SUCCESS，则 ReadFile 返回 FALSE，表明读操作失败。如果读取失败，可以通过 Win32 API GetLastError 得到错误代码，这个错误代码

Windows 驱动开发技术详解

和派遣函数返回值相对应。

通过上述介绍，读者可以基本明白直接方式读写设备的原理。下面给出这种派遣函数的一段示例代码。

```
#001 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#002                        IN PIRP pIrp)
#003 {
#004     KdPrint(("Enter HelloDDKRead\n"));
#005     //得到设备扩展
#006     PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
#007     NTSTATUS status = STATUS_SUCCESS;
#008     //得到当前 IO 堆栈
#009     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
#010     //获取指定的读字节数
#011     ULONG ulReadLength = stack->Parameters.Read.Length;
#012     KdPrint(("ulReadLength:%d\n", ulReadLength));
#013
#014     //得到锁定缓冲区的长度
#015     ULONG mdl_length = MmGetMdlByteCount(pIrp->MdlAddress);
#016     //得到锁定缓冲区的首地址
#017     PVOID mdl_address = MmGetMdlVirtualAddress(pIrp->MdlAddress);
#018     //得到锁定缓冲区的偏移量
#019     ULONG mdl_offset = MmGetMdlByteOffset(pIrp->MdlAddress);
#020
#021     KdPrint(("mdl_address:0X%08X\n", mdl_address));
#022     KdPrint(("mdl_length:%d\n", mdl_length));
#023     KdPrint(("mdl_offset:%d\n", mdl_offset));
#024     if (mdl_length != ulReadLength)
#025     {
#026         //MDL 的长度应该和读长度相等，否则该操作应该设为不成功
#027         pIrp->IoStatus.Information = 0;
#028         status = STATUS_UNSUCCESSFUL;
#029     } else
#030     {
#031         //用 MmGetSystemAddressForMdlSafe 得到 MDL 在内核模式下的映射
#032         PVOID kernel_address = MmGetSystemAddressForMdlSafe(pIrp->MdlAddress,
NormalPagePriority);
#033         KdPrint(("mdl_address:0X%08X\n", kernel_address));
#034         //填充内存
#035         memset(kernel_address, 0XAA, ulReadLength);
#036         pIrp->IoStatus.Information = ulReadLength; // bytes xfered
#037     }
#038     //设置完成状态
#039     pIrp->IoStatus.Status = status;
#040     //结束 IRP 请求
#041     IoCompleteRequest(pIrp, IO_NO_INCREMENT);
#042     KdPrint(("Leave HelloDDKRead\n"));
#043     return status;
#044 }
```

此段代码可以在配套光盘中本章的 MDL_Test 目录下找到。

图 7-12 是应用程序的运行结果。



图 7-12 运行结果

从运行结果可以看出，应用程序提供的缓冲区地址为 0X0012FF70。再来看一下驱动程序输出的 log 消息，如图 7-13 所示。从图 7-13 中可以看出，派遣函数从 MDL 得到的虚拟地址也为 0X0012FF70。调用 MmGetSystemAddressForMdlSafe 后，0X0012FF70 地址被重新映射为地址 0XF7D59F70。映射以后，驱动程序读写 0XF7D59F70 就相当于读写 0X0012FF70 地址了。最后，再用 IRPTrace 软件跟踪一下 IRP_MJ_READ，如图 7-14 所示。

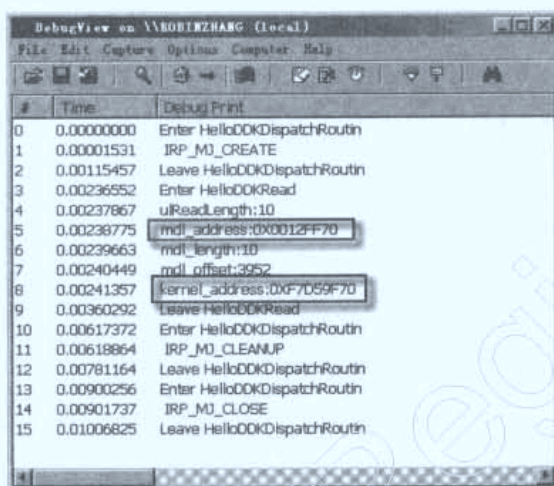


图 7-13 驱动程序显示的 log 信息

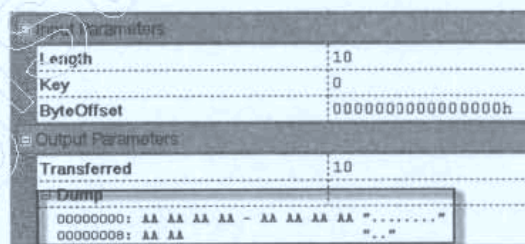


图 7-14 IRPTrace 跟踪结果

7.4 其他方式读写操作

除了缓冲区方式和直接方式外，还有一种读写设备的方法。DDK 文档中没有给这种方式命名，笔者称这种方式为其他方式读写操作。这种方式用得很少，本节就这种方式做一简单介绍。

7.4.1 其他方式设备

前面两节分别介绍了缓冲区方式和直接方式读写设备。除了上述两种方式，还有一种不常用的读写方式，笔者称这种方式为其他方式。这种方式很少被使用。

在调用 IoCreateDevice 创建设备后，对 pDevObj->Flags 既不设置 DO_BUFFERED_IO，

也不设置 `DO_DIRECT_IO`，此时采用的读写方式就是其他读写方式。

在使用其他方式读写设备时，派遣函数直接读写应用程序提供的缓冲区地址。在驱动程序中，直接操作应用程序的缓冲区地址是很危险的。只有驱动程序与应用程序运行在相同线程上下文的情况下，才能使用这种方式。

用其他方式读写时，`ReadFile` 或者 `WriteFile` 提供的缓冲区内存地址，可以在派遣函数中通过 `IRP` 的 `pIrp->UserBuffer` 字段得到。读取的字节数可以从 `I/O` 堆栈中的 `stack->Parameters.Read.Length` 字段中得到。

使用用户模式的内存时要格外小心，因为 `ReadFile` 有可能把空指针地址或者非法地址传递给驱动程序。因此，驱动程序使用用户模式地址前，需要探测这段内存是否可读或者可写。探测可读或者可写，应该使用 `ProbeForWrite` 函数和 `try` 块。

7.4.2 其他方式读写

下面给出一个例子来说明如何编写这种方式的派遣例程。这个例子演示了 `IRP_MJ_READ` 的派遣例程，读者可以参考 `IRP_MJ_READ` 派遣例程写出 `IRP_MJ_WRITE` 的派遣例程。

```
#001 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#002                          IN PIRP pIrp)
#003 {
#004     KdPrint(("Enter HelloDDKRead\n"));
#005     PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
#006     NTSTATUS status = STATUS_SUCCESS;
#007     //得到当前堆栈
#008     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
#009     //得到读的长度
#010     ULONG ulReadLength = stack->Parameters.Read.Length;
#011     //得到读的偏移量
#012     ULONG ulReadOffset = (ULONG)stack->Parameters.Read.ByteOffset.QuadPart;
#013     //得到用户模式地址
#014     PVOID user_address = pIrp->UserBuffer;
#015     KdPrint(("user_address:0x%0X\n",user_address));
#016     __try
#017     {
#018         KdPrint(("Enter __try block\n"));
#019         //判断空指针是否可写，显然会导致异常
#020         ProbeForWrite(user_address,ulReadLength,4);
#021         memset(user_address,0xAA,ulReadLength);
#022         //由于在上面引发异常，所以以后语句不会被执行
#023         KdPrint(("Leave __try block\n"));
#024     }
#025     __except(EXCEPTION_EXECUTE_HANDLER)
#026     {
#027         KdPrint(("Catch the exception\n"));
#028         KdPrint(("The program will keep going\n"));
#029         status = STATUS_UNSUCCESSFUL;
```

```

#030     }
#031     //设置 IRP 的完成例程
#032     pIrp->IoStatus.Status = status;
#033     //设置操作字节数
#034     pIrp->IoStatus.Information = ulReadLength;    // bytes xfered
#035     //结束 IRP 请求
#036     IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#037     KdPrint(("Leave HelloDDKRead\n"));
#038     return status;
#039 }

```

此段代码可以在配套光盘中本章的 Neither_Device_Test 目录下找到。

7.5 IO 设备控制操作

除了用 ReadFile（读设备）和 WriteFile（写设备）以外，应用程序还可以通过另外一个 Win32 API DeviceIoControl 操作设备。DeviceIoControl 内部会使操作系统创建一个 IRP_MJ_DEVICE_CONTROL 类型的 IRP，然后操作系统会将这个 IRP 转发到派遣函数中。

程序员可以用 DeviceIoControl 定义除读写之外的其他操作，它可以让应用程序和驱动程序进行通信。例如，要对一个设备进行初始化操作，程序员自定义一种 I/O 控制码，然后用 DeviceIoControl 将这个控制码和请求一起传递给驱动程序。在派遣函数中，分别对不同的 I/O 控制码进行处理。

7.5.1 DeviceIoControl 与驱动交互

除了 ReadFile 和 WriteFile 两个 Win32 API 可以与驱动程序通信，还有一个 Win32 API DeviceIoControl 可以与驱动程序相互通信。DeviceIoControl 的声明如下：

```

BOOL DeviceIoControl(
    HANDLE hDevice,           //已经打开的设备
    DWORD dwIoControlCode,   // 控制码
    LPVOID lpInBuffer,       // 输入缓冲区
    DWORD nInBufferSize,     // 输入缓冲区大小
    LPVOID lpOutBuffer,      // 输出缓冲区
    DWORD nOutBufferSize,    // 输出缓冲区大小
    LPDWORD lpBytesReturned,  // 实际返回字节数
    LPOVERLAPPED lpOverlapped // 是否 OVERLAP 操作
);

```

其中，lpBytesReturned 对应派遣函数中的 IRP 结构中的 pIrp->IoStatus.Information。

DeviceIoControl 的第二个参数是 I/O 控制码，控制码也称 IOCTL 值，是一个 32 位的无符号整型。IOCTL 需要符合 DDK 的规定，如图 7-15 所示。

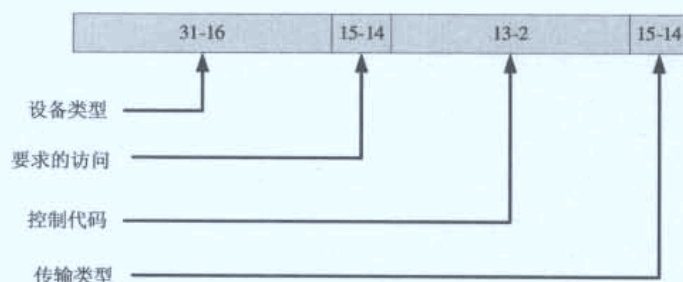


图 7-15 IOCTL 定义

DDK 特意提供了一个宏 CTL_CODE，其定义如下：

```
CTL_CODE( DeviceType, Function, Method, Access )
```

- DeviceType: 设备对象的类型，这个类型应和创建设备（IoCreateDevice）时的类型相匹配。一般是形如 FILE_DEVICE_XX 的宏。
- Function: 这是驱动程序定义的 IOCTL 码。其中：
 - 0X0000 到 0X7FFF: 为微软保留。
 - 0X800 到 0XFFF: 由程序员自己定义。
- Method: 这个是操作模式，可以是下列四种模式之一。
 - METHOD_BUFFERED: 使用缓冲区方式操作。
 - METHOD_IN_DIRECT: 使用直接写方式操作。
 - METHOD_OUT_DIRECT: 使用直接读方式操作。
 - METHOD_NEITHER: 使用其他方式操作。
- Access: 访问权限，如果没有特殊要求，一般使用 FILE_ANY_ACCESS。

7.5.2 缓冲内存模式 IOCTL

这一节介绍缓冲区模式的 IOCTL。在用 CTL_CODE 宏定义这种 IOCTL 时，应该指定 Method 参数为 METHOD_BUFFERED。前面曾经多次提到，在驱动中最好不要直接访问用户模式下的内存地址，缓冲区方式可以避免程序员访问内存模式下的内存地址。

在 Win32 API DeviceIoControl 的内部，用户提供的输入缓冲区的内容被复制到 IRP 中的 pIrp->AssociatedIrp.SystemBuffer 内存地址，复制的字节数是由 DeviceIoControl 指定的输入字节数。

派遣函数可以读取 pIrp->AssociatedIrp.SystemBuffer 的内存地址，从而获得应用程序提供的输入缓冲区数据。另外，派遣函数还可以写入 pIrp->AssociatedIrp.SystemBuffer 的内存地址，这被当做设备输出的数据。操作系统会将这个地址的数据再次复制到 DeviceIoControl 提供的输出缓冲区。复制的字节数由 pIrp->IoStatus.Information 指定。DeviceIoControl 也可以通过它的第七个参数得到这个操作字节数。

派遣函数先通过 `IoGetCurrentIrpStackLocation` 函数得到当前 I/O 堆栈 (`IO_STACK_LOCATION`)。派遣函数通过 `stack->Parameters.DeviceIoControl.InputBufferLength` 得到输入缓冲区大小, 通过 `stack->Parameters.DeviceIoControl.OutputBufferLength` 得到输出缓冲区大小。最后通过 `stack->Parameters.DeviceIoControl.IoControlCode` 得到 IOCTL。在派遣函数中通过 C 语言中的 `switch` 语句分别处理不同的 IOCTL。下面的例子演示了缓冲区方式 IOCTL 的派遣函数。

首先用 `CTL_CODE` 宏定义定义 IOCTL 码:

```
#define IOCTL_TEST1 CTL_CODE(\
    FILE_DEVICE_UNKNOWN, \
    0x800, \
    METHOD_BUFFERED, \
    FILE_ANY_ACCESS)
```

在驱动程序中使用 `CTL_CODE` 需要包含 `NTDDK.h` 头文件, 而在应用程序中使用 `CTL_CODE` 需要包含 `winioclt.h` 头文件。下面是 IOCTL 的派遣函数:

```
#001 NTSTATUS HelloDDKDeviceIOControl(IN PDEVICE_OBJECT pDevObj,
#002                                     IN PIRP pIrp)
#003 {
#004     NTSTATUS status = STATUS_SUCCESS;
#005     KdPrint(("Enter HelloDDKDeviceIOControl\n"));
#006     //得到当前堆栈
#007     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
#008     //得到输入缓冲区大小
#009     ULONG cbin = stack->Parameters.DeviceIoControl.InputBufferLength;
#010     //得到输出缓冲区大小
#011     ULONG cbout = stack->Parameters.DeviceIoControl.OutputBufferLength;
#012     //得到 IOCTL 码
#013     ULONG code = stack->Parameters.DeviceIoControl.IoControlCode;
#014     ULONG info = 0;
#015     switch (code)
#016     {
#017         // process request
#018         case IOCTL_TEST1:
#019         {
#020             KdPrint(("IOCTL_TEST1\n"));
#021             //缓冲区方式 IOCTL
#022             UCHAR* InputBuffer = (UCHAR*)pIrp->AssociatedIrp.SystemBuffer;
#023             for (ULONG i=0; i<cbin; i++)
#024             {
#025                 KdPrint(("0X\n", InputBuffer[i]));
#026             }
#027             //操作输出缓冲区
#028             UCHAR* OutputBuffer = (UCHAR*)pIrp->AssociatedIrp.SystemBuffer;
#029             memset(OutputBuffer, 0xAA, cbout);
#030             //设置实际操作输出缓冲区长度
#031             info = cbout;
#032             break;
#033         }
#034         default:
#035             status = STATUS_INVALID_VARIANT;
#036     }
```



```
#037    //设置 IRP 的完成状态
#038    pIrp->IoStatus.Status = status;
#039    //设置 IRP 请求操作的字节数
#040    pIrp->IoStatus.Information = info;    // bytes xfered
#041    //结束 IRP 请求
#042    IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#043    KdPrint(("Leave HelloDDKDeviceIOControl\n"));
#044    return status;
#045 }
```

此段代码可以在配套光盘中本章的 IOCTL_Test 目录下找到。

7.5.3 直接内存模式 IOCTL

这一节介绍直接方式的 IOCTL。在用 CTL_CODE 宏定义这种 IOCTL 时, 应该指定 Method 参数为 METHOD_OUT_DIRECT 或者 METHOD_IN_DIRECT。直接模式的 IOCTL 同样可以避免驱动程序访问用户模式的内存地址。

在调用 DeviceIoControl 时, 输入缓冲区的内容被复制到 IRP 中的 pIrp->AssociatedIrp.SystemBuffer 内存地址, 复制的字节数是按照 DeviceIoControl 指定输入字节数。这个步骤和缓冲区模式 IOCTL 的处理是一样的。

但是对于 DeviceIoControl 指定的输出缓冲区的处理, 直接模式的 IOCTL 和缓冲区模式的 IOCTL 却是以不同方式处理的。操作系统会将 DeviceIoControl 指定的输出缓冲区锁定, 然后在内核模式地址下重新映射一段地址。

派遣函数中的 IRP 结构中的 pIrp->MdlAddress 记录 DeviceIoControl 指定的输出缓冲区。派遣函数应该使用 MmGetSystemAddressForMdlSafe 将这段内存映射到内核模式下的内存地址。

另外在派遣函数中, 需要先通过 IoGetCurrentIrpStackLocation 函数得到当前 I/O 堆栈。然后通过 stack->Parameters.DeviceIoControl.InputBufferLength 得到输入缓冲区大小, 通过 stack->Parameters.DeviceIoControl.OutputBufferLength 得到输出缓冲区大小。最后通过 stack->Parameters.DeviceIoControl.IoControlCode 得到 IOCTL。在派遣函数中通过 C 语言中的 switch 语句分别处理不同的 IOCTL。

最后说一下 METHOD_IN_DIRECT 和 METHOD_OUT_DIRECT 的差别。其细微差别仅仅体现在打开设备的权限上。当以只读权限打开设备时, METHOD_IN_DIRECT 的 IOCTL 操作将会成功, 而 METHOD_OUT_DIRECT 的 IOCTL 将会失败。如果以读写权限打开设备时, METHOD_IN_DIRECT 和 METHOD_OUT_DIRECT 的 IOCTL 都将会执行成功。下面的例子演示了直接方式 IOCTL 的派遣函数。

首先用 CTL_CODE 宏定义定义 IOCTL 码:

```
#define IOCTL_TEST2 CTL_CODE(\
    FILE_DEVICE_UNKNOWN, \
    0x801, \
```

```
METHOD_IN_DIRECT, \
FILE_ANY_ACCESS)
```

在驱动程序中使用 CTL_CODE 需要包含 NTDDK.h 头文件，而在应用程序中使用 CTL_CODE 需要包含 winioctl.h 头文件。下面是 IOCTL 的派遣函数：

```
#001 NTSTATUS HelloDDKDeviceIOControl(IN PDEVICE_OBJECT pDevObj,
#002                                     IN PIRP pIrp)
#003 {
#004     NTSTATUS status = STATUS_SUCCESS;
#005     KdPrint(("Enter HelloDDKDeviceIOControl\n"));
#006     //得到当前堆栈
#007     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
#008     //得到输入缓冲区大小
#009     ULONG cbin = stack->Parameters.DeviceIoControl.InputBufferLength;
#010     //得到输出缓冲区大小
#011     ULONG cbout = stack->Parameters.DeviceIoControl.OutputBufferLength;
#012     //得到 IOCTL 码
#013     ULONG code = stack->Parameters.DeviceIoControl.IoControlCode;
#014     ULONG info = 0;
#015     switch (code)
#016     {
#017         // process request
#018         case IOCTL_TEST2:
#019             {
#020                 KdPrint(("IOCTL_TEST2\n"));
#021                 //缓冲区方式 IOCTL
#022                 //显示输入缓冲区数据
#023                 UCHAR* InputBuffer = (UCHAR*)pIrp->AssociatedIrp.SystemBuffer;
#024                 for (ULONG i=0;i<cbin;i++)
#025                 {
#026                     KdPrint(("0x%X\n",InputBuffer[i]));
#027                 }
#028                 //pIrp->MdlAddress 为 DeviceIoControl 输出缓冲区地址相同
#029                 KdPrint(("User Address:0x%08X\n",MmGetMdlVirtualAddress(pIrp->
MdlAddress)));
#030                 UCHAR* OutputBuffer = (UCHAR*)MmGetSystemAddressForMdlSafe
(pIrp->MdlAddress,NormalPagePriority);
#031                 //InputBuffer 被映射到内核模式下的内存地址，必定在 0x80000000-
0xFFFFFFFF 之间
#032                 memset(OutputBuffer,0xAA,cbout);
#033                 //设置实际操作输出缓冲区长度
#034                 info = cbout;
#035                 break;
#036             }
#037         default:
#038             status = STATUS_INVALID_VARIANT;
#039     }
#040     //设置 IRP 的完成状态
#041     pIrp->IoStatus.Status = status;
#042     //设置 IRP 请求的操作字节数
#043     pIrp->IoStatus.Information = info; // bytes xfered
#044     //将 IRP 请求结束
#045     IoCompleteRequest(pIrp, IO_NO_INCREMENT);
#046     KdPrint(("Leave HelloDDKDeviceIOControl\n"));
#047     return status;
}
```

此段代码可以在配套光盘中本章的 IOCTL_Test 目录下找到。

7.5.4 其他内存模式 IOCTL

这一节介绍其他模式的 IOCTL。在用 CTL_CODE 宏定义这种 IOCTL 时，应该指定 Method 参数为 METHOD_NEITHER。

这种方式的 IOCTL 很少被用到，因为它直接访问用户模式地址。使用用户模式地址必须保证调用 DeviceIOControl 的线程与派遣函数运行在同一个线程上下文中。

对于 DeviceIOControl 提供的输入缓冲区的地址，派遣函数可以通过 I/O 堆栈 (IO_STACK_LOCATION) 的 stack->Parameters.DeviceIoControl.Type3InputBuffer 得到。同时，DeviceIOControl 提供的输出缓冲区的地址，派遣函数可以由 IRP 的 pIrp->UserBuffer 得到。

派遣函数需要先通过 IoGetCurrentIrpStackLocation 得到当前 I/O 堆栈 (IO_STACK_LOCATION)。然后通过 stack->Parameters.DeviceIoControl.InputBufferLength 得到输入缓冲区大小，通过 stack->Parameters.DeviceIoControl.OutputBufferLength 得到输出缓冲区大小。最后，通过 stack->Parameters.DeviceIoControl.IoControlCode 得到 IOCTL。在派遣函数中通过 C 语言中的 switch 语句分别处理不同的 IOCTL。

由于驱动程序的派遣函数不能保证传递进来的用户地址是合法地址，所以最好对传入的用户模式地址进行可读写判断。可读写的判断在前面已经介绍过，一般通过 ProbeForRead 或者 ProbeForWrite 函数。下面的例子演示了其他方式 IOCTL 的派遣函数。

首先用 CTL_CODE 宏定义定义 IOCTL 码：

```
#define IOCTL_TEST3 CTL_CODE(\
    FILE_DEVICE_UNKNOWN, \
    0x802, \
    METHOD_NEITHER, \
    FILE_ANY_ACCESS)
```

在驱动程序中使用 CTL_CODE 需要包含 NTDDK.h 头文件，而在应用程序中使用 CTL_CODE 需要包含 winioctl.h 头文件。下面是 IOCTL 的派遣函数：

```
#001 NTSTATUS HelloDDKDeviceIOControl(IN PDEVICE_OBJECT pDevObj,
#002                                     IN PIRP pIrp)
#003 {
#004     NTSTATUS status = STATUS_SUCCESS;
#005     KdPrint(("Enter HelloDDKDeviceIOControl\n"));
#006     //得到当前堆栈
#007     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
#008     //得到输入缓冲区大小
#009     ULONG cbin = stack->Parameters.DeviceIoControl.InputBufferLength;
#010     //得到输出缓冲区大小
#011     ULONG cbout = stack->Parameters.DeviceIoControl.OutputBufferLength;
#012     //得到 IOCTL 码
#013     ULONG code = stack->Parameters.DeviceIoControl.IoControlCode;
#014     ULONG info = 0;
#015     switch (code)
#016     {
        // process request
```

```

#017         case IOCTL_TEST3:
#018         {
#019             KdPrint(("IOCTL_TEST3\n"));
#020             //缓冲区方式 IOCTL
#021             //显示输入缓冲区数据
#022             UCHAR* UserInputBuffer = (UCHAR*)stack->Parameters.Device
IoControl.Type3InputBuffer;
#023             KdPrint(("UserInputBuffer:0X%0X\n",UserInputBuffer));
#024             //得到用户模式地址
#025             PVOID UserOutputBuffer = pIrp->UserBuffer;
#026             KdPrint(("UserOutputBuffer:0X%0X\n",UserOutputBuffer));
#027             __try
#028             {
#029                 KdPrint(("Enter __try block\n"));
#030                 //判断指针是否可读
#031                 ProbeForRead(UserInputBuffer,cbin,4);
#032                 //显示输入缓冲区内容
#033                 for (ULONG i=0;i<cbin;i++)
#034                 {
#035                     KdPrint(("0X\n",UserInputBuffer[i]));
#036                 }
#037                 //判断指针是否可写
#038                 ProbeForWrite(UserOutputBuffer,cbout,4);
#039                 //操作输出缓冲区
#040                 memset (UserOutputBuffer,0xAA,cbout);
#041                 //由于在上面引发异常,所以以后语句不会被执行!
#042                 info = cbout;
#043                 KdPrint(("Leave __try block\n"));
#044             }
#045             __except (EXCEPTION_EXECUTE_HANDLER)
#046             {
#047                 KdPrint(("Catch the exception\n"));
#048                 KdPrint(("The program will keep going\n"));
#049                 status = STATUS_UNSUCCESSFUL;
#050             }
#051             info = cbout;
#052             break;
#053         }
#054         default:
#055             status = STATUS_INVALID_VARIANT;
#056     }
#057     // 设置 IRP 完成状态
#058     pIrp->IoStatus.Status = status;
#059     //设置 IRP 操作字节数
#060     pIrp->IoStatus.Information = info; // bytes xfered
#061     //结束 IRP 请求
#062     IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#063     KdPrint(("Leave HelloDDKDeviceIoControl\n"));
#064     return status;
#065 }

```

此段代码可以在配套光盘中本章的 IOCTL_Test 目录下找到。

7.6 小结

本章重点介绍了驱动程序中的处理 IRP 请求的派遣函数。所有对设备的操作最终将转化为 IRP 请求，这些 IRP 请求会被传送到派遣函数处理。本章主要介绍了 IRP_MJ_READ、IRP_MJ_WRITE、IRP_MJ_DEVICE_CONTROL 的派遣函数。这些 IRP 请求分别有缓冲区方式、直接方式和其他方式的操作。其中，缓冲区方式和直接方式是在驱动程序开发中经常用到的。

第 2 篇 进阶篇

第 8 章 驱动程序的同步处理

第 9 章 IRP 的同步

第 10 章 定时器

第 11 章 驱动程序调用驱动程序

第 12 章 分层驱动程序

第 13 章 让设备实现即插即用

第 14 章 电源管理

第 8 章 驱动程序的同步处理

在上一章中，笔者介绍了派遣函数对 IRP 的处理。但是这些处理没有考虑“同步”的问题，“同步”处理是驱动程序中特别重要的任务，本章将对驱动程序中的同步处理进行介绍。

Windows 是个多任务的操作系统，每个任务对应一个运行的进程。每个运行的进程中可以包含多个线程。如果没有同步机制的控制，所有的线程会任意运行。然而，多个线程可能会要求操作同一个资源，这时就需要同步处理。

如果驱动程序没有很好地处理同步问题，操作系统的性能就会下降，甚至出现死锁等现象。

8.1 基本概念

驱动程序中涉及同步处理的地方很多，本节先对一些基本概念做简单介绍。

8.1.1 问题的引出

在支持多线程的操作系统下，有些函数会出现不可重入的现象。所谓“可重入”，是指函数的执行结果不和执行顺序有关。反之，如果执行结果和执行顺序有关，则称这个函数是“不可重入”的。

“不可重入”的函数会对多线程操作系统下的程序带来错误，“不可重入”的根本原因是由于各个线程之间的切换导致的。下面是一个“不可重入”函数的例子。

```
#001 int number=0;
#002 void Foo()
#003 {
#004     number++;
#005     //做一些事情
#006     number--;
#007 }
```

其中, `number` 是全局函数, 这导致了 `Foo` 函数的不可重入性。`Foo` 的功能是先让 `number` 自增 1, 然后运行某些操作, 最后让 `number` 自减 1。运行 `Foo` 的前后应该保持 `number` 的数值不变。然而, 在多线程的环境下, 却不能保证这一点。将 `Foo` 函数用汇编代码展开:

```
#001 ;将 number++分解成如下
#002 mov     eax,[number]
#003 add     eax,1
#004 mov     [number],eax
#005 ;将 number--分解成如下
#006 mov     ecx,dword ptr [number]
#007 sub     ecx,1
#008 mov     dword ptr [number],ecx
```

如果单独运行函数会保证 `number` 前后一致。但在多线程的环境下, 同时运行两个 `Foo` 函数, 可能会导致汇编指令交错在一起, 从而导致函数的不可重入性。

为了解决函数的不可重入性, 使之变成可重入的函数, 需要对函数进行同步控制。

8.1.2 同步与异步

下面简单地介绍一下多线程运行的基本原理。如果 PC 中只有一个 CPU, CPU 将时间分成一个个时间片段, 然后 CPU 将这些时间片段分配给各个线程。当前的线程消耗完这个时间片段后, CPU 会转而执行其他的线程。由于 CPU 运行速度非常快, 每个线程仿佛是在同时运行一样。

这个时间片段不能设置太小, 否则每个线程没有运行多久就被强制切换到别的线程。同时, 这个时间片段也不能太大, 否则用户会感觉到各个线程不是同时运行的。另外, 不同线程分配的时间片可能会不同。

这时候, 各个线程之间的关系被称为是异步的, 也就是每个线程的运行不受其他线程的影响。

然而, 有些时候线程需要按一定顺序执行, 比如上一节的例子, 否则会出现错误。这就需要同步机制来处理这个问题。

举个例子, 两个人约好在某商场见面, 然后一起去购物。这两个人相当于两个线程, 约见的地方相当于同步点。先到的人需要等待另一个人的到达, 这种约定就是同步机制。如果没有这种同步机制, 两个人分别去购物, 这种方式就变成异步方式了。这仅仅是个比喻。

8.2 中断请求级

在设计 Windows 的时候, 设计者将中断请求划分为软件中断和硬件中断, 并将这些中断都映射成不同级别的中断请求级 (IRQL)。同步处理机制很大程度上依赖于中断请求级, 本节对中断请求级做简单的介绍。

8.2.1 中断请求（IRQ）与可编程中断控制器（PIC）

中断请求（IRQ）一般有两种，一种是外部中断，也就是硬件产生的中断，另一种由软件指令 `int n` 产生的中断。这里只介绍硬件产生的中断。

在传统 PC 中，一般可以接收 16 个中断信号，每个中断信号对应一个中断号。外部中断分为不可屏蔽（NMI）和可屏蔽中断，分别由 CPU 的两根引脚 NMI 和 INTR 来接收。

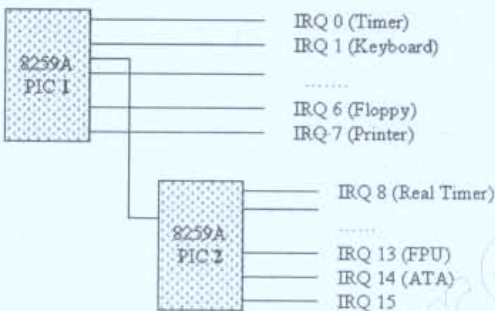


图 8-1 8259A 中断控制器

可屏蔽中断是通过可编程中断控制器（PIC）8259A 芯片向 CPU 发出请求的，如图 8-1 所示。通过 8259A 芯片，可以设置中断的优先级、是否被屏蔽等。CPU 的 INTR 连接主 8259A 芯片，同时主 8259A 芯片级联一片从 8259A 芯片。主从两个芯片一共可以接收 16 个中断信号。下面的表 8-1 列出了 16 个 IRQ 编号对应的用途。

表 8-1 PIC 的中断向量

IRQ 编号	设备名称	用 途
IRQ0	Time	计算机系统计时器
IRQ1	KeyBoard	键盘
IRQ2	Redirect IRQ9	与 IRQ9 相接，MPU-401 MDI 使用该 IRQ
IRQ3	COM2	串口设备
IRQ4	COM1	串口设备
IRQ5	LPT2	建议声卡使用该 IRQ
IRQ6	FDD	软驱传输控制用
IRQ7	LPT1	打印机传输控制用
IRQ8	CMOS Alert	即时时钟
IRQ9	Redirect IRQ2	与 IRQ2 相接。可设定给其他硬件使用
IRQ10	Reversed	建议保留给网卡使用该 IRQ
IRQ11	Reversed	建议保留给 AGP 显卡使用
IRQ12	PS/2 Mouse	接 PS/2 鼠标，若无也可设定给其他硬件使用
IRQ13	FPU	协处理器用，例如 FPU（浮点运算器）
IRQ14	Primary IDE	主硬盘传输控制用
IRQ15	Secondary Ide	从硬盘传输控制用

8.2.2 高级可编程控制器 (APIC)

传统 PC 一般使用 2 片 Intel 8259A 中断控制器。然而, 现在的 X86 计算机基本都是用高级可编程控制器, 即 Advanced Programmable Interrupt Controller (APIC)。

APIC 兼容 PIC, 并且 APIC 把 IRQ 的数量增加到了 24 个。读者可以用设备管理器查看这 24 个中断, 选择“查看”|“依连接排序资源”, 可以看到系统中的 0~23 号 IRQ, 如图 8-2 所示。

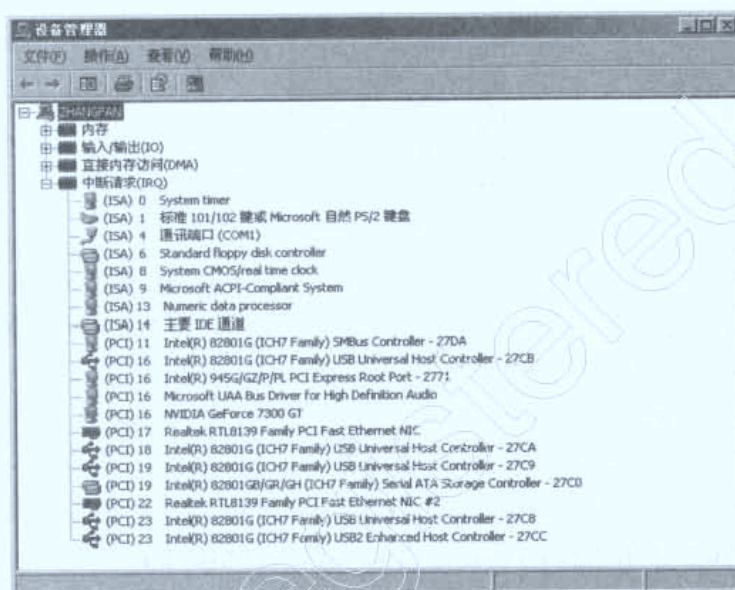


图 8-2 设备管理器

8.2.3 中断请求级 (IRQL)

在 APIC 中, IRQ 的数量被增加到了 24 个, 每个 IRQ 有各自的优先级别, 正在运行的线程随时可以被中断打断, 进入到中断处理程序。当优先级高的中断来临时, 处在优先级低的中断处理程序, 也会被打断, 进入到更高级别的中断处理函数。

Windows 将中断的概念进行了扩展, 提出一个中断请求级 (IRQL) 的概念。其中规定了 32 个中断请求级别, 分别是 0~2 级别为软件中断, 3~31 级为硬件中断 (这里包括 APIC 中的 24 个中断), 如图 8-3 所示。其中, 数字从 0 到 31, 优先级别逐次递增。

Windows 将 24 个 IRQ 映射到了从 DISPATCH_LEVEL 到 PROFILE_LEVEL 之间, 不同硬件的中断处理程序运行在不同的 IRQL 级别中。硬件的 IRQL 称为设备中断请求级, 或者简称 DIRQL。Windows 大部分时间运行在软件中断级别中。当设备中断来临时, 操作系统提升 IRQL 至 DIRQL 级别, 并且运行中断处理函数。当中断处理函数结束后, 操作系统把 IRQL 降到原来的级别。

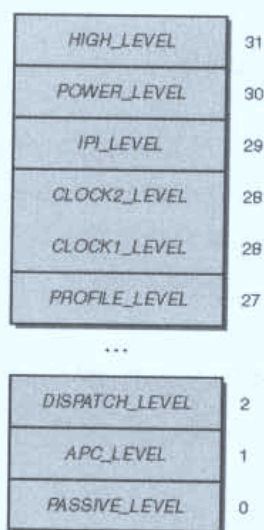


图 8-3 IRQL

用户模式的代码是运行在最低优先级的 `PASSIVE_LEVEL` 级别。驱动程序的 `DriverEntry` 函数、派遣函数、`AddDevice` 等函数一般都运行在 `PASSIVE_LEVEL` 级别，它们在必要时可以申请进入 `DISPATCH_LEVEL` 级别。

Windows 负责线程调度的组件是运行在 `DISPATCH_LEVEL` 级别，当前的线程运行完时间片后，系统自动从 `PASSIVE_LEVEL` 级别提升到 `DISPATCH_LEVEL` 级别。当线程切换完毕后，操作系统又从 `DISPATCH_LEVEL` 级别降到 `PASSIVE_LEVEL` 级别。

驱动程序的 `StartIO` 函数和 `DPC` 函数也运行在 `DISPATCH_LEVEL` 级别，这些函数会在后面陆续进行介绍。

在内核模式下，可以通过调用 `KeGetCurrentIrql` 内核函数来得到当前的 IRQL 级别。

8.2.4 线程调度与线程优先级

在应用程序的编程中，经常会听到线程优先级的概念。线程优先级和 IRQL 是两个容易混淆的概念。所有应用程序都运行在 `PASSIVE_LEVEL` 级别上，它的优先级别最低，可以被其他 IRQL 级别的程序打断。线程优先级只针对应用程序而言，只有程序运行在 `PASSIVE_LEVEL` 级别才有意义。

线程优先级是指某线程是否有更多的机会运行在 CPU 上，线程优先级高的线程有更多的机会被内核调度。负责调度线程的内核组件运行在 `DISPATCH_LEVEL` 级别的 IRQL 上，这时候所有应用程序的程都停止，等待着被调度。

`ReadFile` 内部创建 `IRP_MJ_READ`，然后这个 IRP 被传递到驱动程序的派遣函数中。这时候派遣函数运行于 `ReadFile` 所在的线程中，或者说 `ReadFile` 和派遣函数位于同一个线程上下文。

8.2.5 IRQL 的变化

为了更好地理解 IRQL 概念，笔者在下面详细描述了一个线程的运行过程。这个线程在运行中，被一个中断打断，并且在中断服务执行时，被更高级别的中断打断，运行的过程如图 8-4 所示。线程的运行分为以下几个阶段。

- 阶段 1：一个普通线程 A 正在运行。
- 阶段 2：这个时刻有一个中断发生，它的 IRQL 为 0xD。CPU 中断当前运行的线程 A，将 IRQL 提升至 0xD 级别。
- 阶段 3：这时候有一个更高优先级的中断发生，它的 IRQL 是 0x1A。这时候 CPU 将 IRQL 提升至 0x1A 级别。
- 阶段 4：这时候又有一个中断发生，但它的 IRQL 为 0x18，低于上一个中断优先级。CPU 不会理睬这个中断。
- 阶段 5：这时候 IRQL 为 0x1A 的中断结束，操作系统进入 IRQL 为 0x18 的中断服务。
- 阶段 6：这时候 IRQL 为 0x18 中断结束，于是进入 IRQL 为 0xD 的中断服务。
- 阶段 7：最后 IRQL 为 0xD 的中断结束，操作系统恢复线程 A。

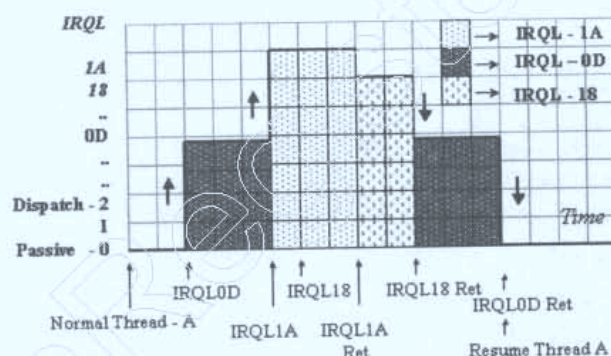


图 8-4 IRQL 的变化

线程运行在 PASSIVE_LEVEL 级别，这个时候操作系统随时可能将当前线程切换到别的线程。但是如果提升 IRQL 到 DISPATCH_LEVEL 级别，这时候不会出现线程的切换。这是一种很常用的同步处理机制，但这种方法只能使用于单 CPU 的系统。对于多 CPU 的系统，需要采用别的同步处理机制。

8.2.6 IRQL 与内存分页

这里还需要介绍一下 IRQL 与内存分页之间存在的问题。在使用分页内存时，可能会导致页故障。因为分页内存随时可能从物理内存交换到磁盘文件。读取不在物理内存中的分页内存时，会引发一个页故障，从而执行这个异常的处理函数。异常处理函数会重新将

磁盘文件的内容交换到物理内存中。

页故障允许出现在 `PASSIVE_LEVEL` 级别的程序中，但如果在 `DISPATCH_LEVEL` 或者更高级别 `IRQL` 的程序中会带来系统崩溃。

对于等于或者高于 `DISPATCH_LEVEL` 级别的程序不能使用分页内存，必须使用非分页内存。驱动程序的 `StartIO` 例程、`DPC` 例程、中断服务例程都运行在 `DISPATCH_LEVEL` 或者更高的 `IRQL`。因此，在这些例程中不能使用分页内存，否则会导致系统崩溃。

8.2.7 控制 IRQL 提升与降低

有些时候，驱动程序中需要提升 `IRQL` 级别。在运行一段时间后，再降回原来的 `IRQL` 级别。这样做的目的一般是基于同步处理的需要。

首先驱动程序程序需要知道当前状态是什么 `IRQL` 级别，可以通过 `KeGetCurrentIrql` 内核函数获取当前 `IRQL` 级别。

然后驱动程序使用内核函数 `KeRaiseIrql` 将 `IRQL` 提高。`KeRaiseIrql` 需要两个参数，第一个参数是提升后的 `IRQL` 级别，第二个参数保存提升前的 `IRQL` 级别。

最后，驱动程序在某个时刻需要将 `IRQL` 恢复到以前的 `IRQL` 级别，驱动程序可以调用 `KeLowerIrql` 内核函数。下面的代码演示了在驱动程序中如何提升与降低 `IRQL` 级别。

```
#001 VOID RaiseIRQL_Test()
#002 {
#003     KIRQL oldirql;
#004     //确保当前 IRQL 等于或小于 DISPATCH_LEVEL
#005     ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
#006     //提升 IRQL 至 DISPATCH_LEVEL, 并将先前的 IRQL 保存
#007     KeRaiseIrql(DISPATCH_LEVEL, &oldirql);
#008     //.....
#009     //恢复到先前的 IRQL
#010     KeLowerIrql(oldirql);
#011 }
```

8.3 自旋锁

自旋锁也是一种同步处理机制，它能保证某个资源只能被一个线程所拥有。这种保护被形象地称做“上锁”，本节主要对自旋锁的原理和使用方法进行介绍。

8.3.1 原理

在 Windows 内核中，有一种被称为自旋锁（Spin Lock）的锁，它可以用于驱动程序中的同步处理。初始化自旋锁时，处于解锁状态，这时它可以被程序“获取”。“获取”后的自旋锁处于锁住状态，不能被再次“获取”。锁住的自旋锁必须被“释放”以后，才能再次被“获取”。

如果自旋锁已经被锁住，这时有程序申请“获取”这个自旋锁，程序则处于“自旋”状态。所谓自旋状态，就是不停地询问是否可以“获取”自旋锁。自旋锁也因此而得名。

自旋锁不同于线程中的等待事件。在线程中如果等待某个事件（Event），操作系统会使这个线程进入休眠状态，CPU 会运行其他线程。而自旋锁原理则不同，它不会切换到别的线程，而是一直让这个线程“自旋”。因此，对自旋锁占用时间不宜过长，否则会导致申请自旋锁的其他线程处于自旋，这会浪费 CPU 宝贵的时间。

在单 CPU 的系统中，“获取”自旋锁仅仅是将当前的 IRQL 从 PASSIVE_LEVEL 级别提升到 DISPATCH_LEVEL 级别。但是在多 CPU 的系统中，自旋锁的实现方法会复杂得多。有兴趣的读者可以针对自旋锁的内核代码进行反汇编研究。需要注意的是，驱动程序必须在低于或者等于 DISPATCH_LEVEL 的 IRQL 级别中使用自旋锁。

8.3.2 使用方法

自旋锁的作用一般是为使各派遣函数之间同步。尽量不要将自旋锁放在全局变量中，而应该将自旋锁放在设备扩展里。自旋锁用 KSPIN_LOCK 数据结构表示。

```
typedef struct _DEVICE_EXTENSION {
    .....
    KSPIN_LOCK My_SpinLock; //在设备扩展中定义自旋锁
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

使用自旋锁前，需要先对其进行初始化，可以使用 KeInitializeSpinLock 内核函数。一般在驱动程序的 DriverEntry 或者 AddDevice 函数中初始化自旋锁。

申请自旋锁可以使用内核函数 KeAcquireSpinLock，它有两个参数，第一个参数为自旋锁指针，第二个参数记录获得自旋锁以前的 IRQL 级别。一般用下面的方法初始化和申请获得自旋锁。

```
#001 PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
#002 KIRQL oldIrql;
#003 KeAcquireSpinLock(&pdx->My_SpinLock,&oldIrql);
```

释放自旋锁使用 KeReleaseSpinLock 内核函数，它也是两个参数，第一个为自旋锁指针，第二个是释放自旋锁后应该恢复的 IRQL 级别。一般用下面方法释放自旋锁。

```
KeReleaseSpinLock(&pdx->My_SpinLock,oldIrql);
```

如果在 DISPATCH_LEVEL 级别申请自旋锁，不会改变 IRQL 级别。这时，申请自旋锁可以简单地使用 KeAcquireSpinLockAtDpcLevel 内核函数，而释放自旋锁使用 KeReleaseSpinLockFromDpcLevel 内核函数。

8.4 用户模式下的同步对象

在内核模式下可以使用很多种内核同步对象，这些内核同步对象和用户模式下的同步

对象非常类似。同步对象包括事件 (Event)、互斥体 (Mutex)、信号灯 (Semaphore) 等。为了和内核模式同步对象对比, 本节先介绍用户模式下同步对象的使用方法。

在学习完内核同步对象后, 读者会发现用户模式的同步对象都是借助内核模式的同步对象实现的。用户模式下的同步对象其实是内核模式下同步对象的再次封装。

8.4.1 用户模式的等待

在应用程序中, 可以使用 WaitForSingleObject 和 WaitForMultipleObjects 等待同步对象。其中, WaitForSingleObject 用于等待一个同步对象, 而 WaitForMultipleObjects 用于等待多个同步对象。WaitForSingleObject 函数声明如下:

```
DWORD WaitForSingleObject(  
    HANDLE hHandle,           // 同步对象句柄  
    DWORD dwMilliseconds     // 等待时间  
);
```

第二个参数 dwMilliseconds 是等待时间, 单位是 ms (毫米)。同步对象有两种状态, 一种是激发状态, 一种是未激发状态。如果同步对象处于未激发状态, WaitForSingleObject 则进入休眠, 等待同步对象被激发。如果同步对象在指定的等待时间内, 还没有处于激发状态, 则自动停止休眠。

dwMilliseconds 也可以设定为 INFINITE, 这表示无限期地等待下去。另外, dwMilliseconds 也可以为 0, 其作用是强迫操作系统将当前线程切换到其他线程。

WaitForMultipleObjects 函数声明如下:

```
DWORD WaitForMultipleObjects(  
    DWORD nCount,             // 同步对象数组元素个数  
    CONST HANDLE *lpHandles,  // 同步对象数组指针  
    BOOL bWaitAll,            // 是否等待全部同步对象  
    DWORD dwMilliseconds     // 等待时间  
);
```

和 WaitForSingleObject 函数略有不同, WaitForMultipleObjects 可以等待多个同步对象, 它需要提供一个同步对象的数组, 如果参数 bWaitAll 为真, 则等待所有的同步对象。否则, 只要有一个同步对象被激发, 则线程恢复运行。

8.4.2 用户模式开启多线程

等待同步对象一般出现在多线程的编程中, 因此这里介绍一下应用程序如何创建新线程。Win32 API CreateThread 函数负责创建新线程。

```
HANDLE CreateThread(  
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // 安全属性  
    SIZE_T dwStackSize, // 初始化堆栈大小  
    LPTHREAD_START_ROUTINE lpStartAddress, // 线程运行的函数指针
```

```

LPVOID lpParameter,           // 传入函数中的参数
DWORD dwCreationFlags,        // 开启线程时的状态
LPDWORD lpThreadId             // 返回线程 ID
);

```

CreateThread 用 lpStartAddress 参数指定新线程的运行地址, 用 dwStackSize 参数指定新线程的堆栈大小, 用 lpParameter 参数指定新线程的参数等信息。

一些旧的运行时函数不支持多线程。运行时函数中使用了大量的全局变量和静态变量, 如全局变量和静态变量没有用同步机制保护, 就会导致函数在多线程环境下运行错误。这种错误也可以说成函数是“不可重入”的。

幸运地是, 微软编写了一套多线程版本的运行时函数库。在使用这个版本的运行时库时, 需要指定链接这些库, 修改的方法如图 8-5 所示。

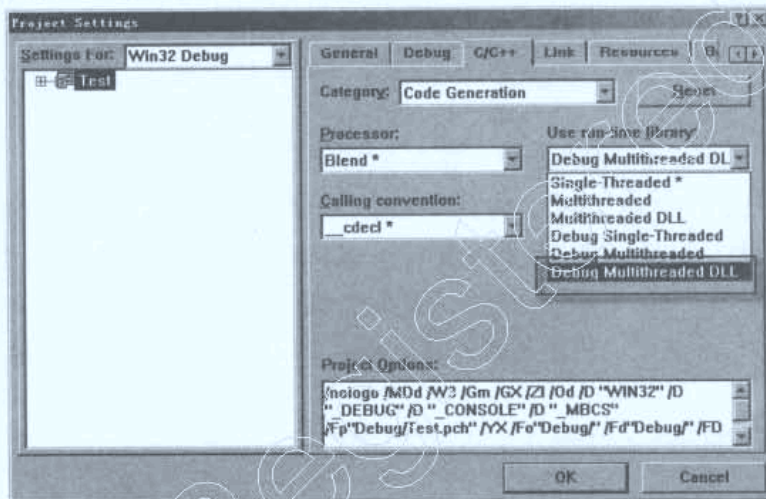


图 8-5 设置编译器参数

另外, 创建多线程的时候最好不使用 CreateThread 函数, 而使用 _beginthreadex 函数。_beginthreadex 函数是对 CreateThread 函数的封装, 其参数与 CreateThread 完全一致。_beginthreadex 函数的函数名前面有个下划线, 是因为它不是标准 C 语言提供的运行时函数。

另外, 许多第三方库都会对 CreateThread 函数进行封装, 从而达到方便程序员使用的目的。例如, MFC 库提供了 CWinThread 类, 这个类对线程的各个操作函数进行了封装。

8.4.3 用户模式的事件

事件是一种典型的同步对象。用户模式下的事件和内核模式的事件对象紧密相连。在使用事件之前, 需要对事件进行初始化, 使用 CreateEvent API 函数。

```

HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES lpEventAttributes, // 安全属性

```



```
    BOOL bManualReset,           // 是否设定为手动
    BOOL bInitialState,         // i 初始状态
    LPCTSTR lpName               // 命名
};
```

所有形如 CreateXXX 的 Win32 API 函数，如果它的第一个参数是 LPSECURITY_ATTRIBUTES 类型，那么这种 API 内部都会创建一个相应的内核对象。这种 API 返回一个句柄，操作系统可以通过这个句柄找到具体的内核对象。

CreateEvent 函数就是这样的函数，它内部会使操作系统创建一个内核事件对象。CreateEvent 返回的句柄值就代表了这个内核事件对象。应用程序无法获得这个内核事件对象的指针，而用一个句柄（一个 32 位的无符号整数）代表事件对象。

一般情况下，CreateEvent 的安全属性设置为 NULL。它的第二个参数 bManualReset 表示创建的事件是否是手动模式。如果是手动模式的事件，事件处于激发状态后，需要手动设置才能回到未激发状态。如果是自动模式，当事件处于激发状态后，遇到任意一个等待（如 WaitForSingleObject），则自动变回未激发状态。下面的例子综合演示了上面介绍的 API 函数，包括创建线程、创建事件、等待事件等。

在这个例子的主线程中，开启新的辅助线程，主线程把一个事件的句柄传递给子线程。同时，主线程等待该事件激发，辅助线程所做的事情就是显示一些信息，并设置该事件。这个例子非常简单，但也说明了同步处理的重要性。如果主线程不等待该事件，也是以异步的方式共同的和辅线程执行，这时很有可能主线程都退出来了，而辅助线程还在继续运行。

```
#001 #include <windows.h>
#002 #include <process.h> /* _beginthread, _endthread */
#003 #include <stddef.h>
#004 #include <stdlib.h>
#005 #include <conio.h>
#006
#007 UINT WINAPI Thread1(LPVOID para)
#008 {
#009     printf("Enter Thread1\n");
#010     HANDLE* phEvent = (HANDLE*)para;
#011     //设置该事件激发
#012     SetEvent(*phEvent);
#013     printf("Leave Thread1\n");
#014     return 0;
#015 }
#016 int main()
#017 {
#018     //创建同步事件
#019     HANDLE hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
#020     //开启新线程，并将同步事件句柄指针传递给新线程
#021     HANDLE hThread1 = (HANDLE) _beginthreadex (NULL, 0, Thread1, &hEvent, 0, NULL);
#022     //等待该事件激发
#023     WaitForSingleObject(hEvent, INFINITE);
#024     return 0;
#025 }
```

8.4.4 用户模式的信号灯

信号灯也是一种常用的同步对象，信号灯也有两种状态，一种是激发状态，另外一种则是未激发状态。信号灯内部有个计数器，可以理解信号灯内部有 N 个灯泡。如果有一个灯泡亮着，就代表信号灯处于激发状态，如果全部熄灭，则代表信号灯处于未激发状态。使用信号灯前需要先创建信号灯，CreateSemaphore 函数负责创建信号灯。它的声明如下：

```
HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,    // 安全属性
    LONG lInitialCount,    // 初始化计数个数
    LONG lMaximumCount,    // 计数器最大个数
    LPCTSTR lpName    // 命名
);
```

其中，第二个参数 lInitialCount 在指明初始化的时候，计数器的值为多少。第三个参数 lMaximumCount 指明该信号灯计数器最大的值为多少。如果初始值为 0，则处于未激发状态。如果初始值为非零，则处于激发状态。

另外，可以使用 ReleaseSemaphore 函数增加信号灯的计数器，其函数声明如下：

```
BOOL ReleaseSemaphore(
    HANDLE hSemaphore,    // 信号灯句柄
    LONG lReleaseCount,    // 本次操作增加的计数
    LPLONG lpPreviousCount    // 记录以前的计数
);
```

其中，第二个参数 lReleaseCount 是这次操作增加计数的数量，第三个参数 lpPreviousCount 获得执行本操作之前计数的大小。

另外，对信号灯执行一次等待操作，就会减少一个计数，相当于熄灭一个灯泡。当计数为零时，也就是所有灯泡都熄灭时，当前线程进入睡眠状态，直到信号灯变为激发状态。下面综合以上的 API，编写了信号灯同步对象的使用方法。

```
#001 #include <windows.h>
#002 #include <process.h>    /* _beginthread, _endthread */
#003 #include <stdio.h>
#004
#005 UINT WINAPI Thread1(LPVOID para)
#006 {
#007     printf("Enter Thread1\n");
#008     HANDLE* phSemaphore = (HANDLE*)para;
#009     //等待 5s
#010     Sleep(5000);
#011     printf("Leave Thread1\n");
#012     //将信号灯计数器加 1，使之处于激发状态
#013     ReleaseSemaphore(*phSemaphore, 1, NULL);
#014     return 0;
#015 }
#016
#017 int main()
```



```

#018  {
#019     //创建同步事件
#020     HANDLE hSemaphore = CreateSemaphore(NULL,2,2,NULL);
#021     //此时的信号灯计数为 2，处于触发状态
#022     WaitForSingleObject(hSemaphore,INFINITE);
#023     //此时的信号灯计数为 1，处于触发状态
#024     WaitForSingleObject(hSemaphore,INFINITE);
#025     //此时的信号灯计数为 0，处于未触发状态
#026     //开启新线程，并将同步事件句柄指针传递给新线程
#027     HANDLE hThread1 = (HANDLE) _beginthreadex (NULL,0,Thread1,&hSemaphore,
0,NULL);
#028     //等待该事件激发
#029     WaitForSingleObject(hSemaphore,INFINITE);
#030     return 0;
#031 }

```

8.4.5 用户模式的互斥体

互斥体也是一种常用的同步对象。互斥体可以避免多个线程争夺同一个资源。例如，多线程环境中，只能有一个线程占有互斥体。获得互斥体的线程如果不释放互斥体，其他线程永远不会获得这个互斥体。互斥体的概念类似于同步事件，所不同的是同一个线程可以递归获得互斥体。递归获得互斥体的意思是，得到互斥体的线程还可以再次获得这个互斥体，或者说互斥体对于已经获得互斥体的线程不产生“互斥”关系。而同步事件不能递归获取。

互斥体也有两种状态，激发态和未激发态。如果线程获得互斥体时，此时的状态是未激发态，当释放互斥体时，互斥体的状态为激发态。

初始化互斥体的函数是 `CreateMutex`，其函数声明如下：

```

HANDLE CreateMutex(
    LPSECURITY_ATTRIBUTES lpMutexAttributes, // 安全属性
    BOOL bInitialOwner, // 是否被占有
    LPCTSTR lpName // 命名
);

```

其中第二个参数 `bInitialOwner` 表明初始化时是否被占有。如果没有被占有，则为激发态，否则为未激发态。另外，获得互斥体的函数是 `WaitXX` 函数，而释放互斥体的函数是 `ReleaseMutex` 函数。

下面的例子演示了如何使用互斥体。例中主线程创建两个线程，且两个线程先各自运行一段时间。因为内核的调度，实际上两个线程的运行会交织起来。为了让两个线程不出现交织运行，本例使用互斥体作为同步处理机制。

```

#001 #include <windows.h>
#002 #include <process.h> /* _beginthread, _endthread */
#003 #include <stdio.h>
#004
#005 UINT WINAPI Thread1(LPVOID para)
#006 {

```

```

#007 HANDLE* phMutex = (HANDLE*)para;
#008 //得到互斥体
#009 WaitForSingleObject(*phMutex, INFINITE);
#010 //对于同一个线程，主要获得了互斥体，还可以获得多次
#011 WaitForSingleObject(*phMutex, INFINITE);
#012 printf("Enter Thread1\n");
#013 //强迫等待2s，如果没有同步处理，线程1和线程2会交织在一起
#014 Sleep(2000);
#015 printf("Leave Thread1\n");
#016 //释放互斥体
#017 ReleaseMutex(*phMutex);
#018 return 0;
#019 }
#020
#021 UINT WINAPI Thread2(LPVOID para)
#022 {
#023     HANDLE* phMutex = (HANDLE*)para;
#024     //得到互斥体
#025     WaitForSingleObject(*phMutex, INFINITE);
#026     printf("Enter Thread2\n");
#027     //强迫等待2s，如果没有同步处理，线程1和线程2会交织在一起
#028     Sleep(2000);
#029     printf("Leave Thread2\n");
#030     //释放互斥体
#031     ReleaseMutex(*phMutex);
#032     return 0;
#033 }
#034 int main()
#035 {
#036     //创建同步事件
#037     HANDLE hMutex = CreateMutex(NULL, FALSE, NULL);
#038     //开启新线程，并将同步事件句柄指针传递给新线程
#039     HANDLE hThread1 = (HANDLE) _beginthreadex (NULL, 0, Thread1, &hMutex, 0, NULL);
#040     HANDLE hThread2 = (HANDLE) _beginthreadex (NULL, 0, Thread2, &hMutex, 0, NULL);
#041     //强迫等待6s，让线程1和线程2运行完毕
#042     //下一节将介绍更好的等待方法
#043     Sleep(6000);
#044     return 0;
#045 }

```

程序运行结束后，观察程序的输出结果，如图 8-6 所示。结果表明两个线程的确没有出现交织运行情况。



图 8-6 运行结果

8.4.6 等待线程完成

还有一种同步对象，其实在前面已经介绍过了，这就是线程对象。每个线程同样有两个状态，激发状态和未激发状态。当线程处于运行之中的时候，是未激发状态。当线程终止后，线程处于激发状态。可以用 `WaitXX` 函数对线程句柄进行等待。

下面的例子演示了如何等待线程的结束。例子中，主线程创建两个线程，主线程用 `WaitForMultipleObjects` 等待两个子线程全部运行结束。

```
#001 #include <windows.h>
#002 #include <process.h>    /* _beginthread, _endthread */
#003 #include <stdio.h>
#004
#005 UINT WINAPI Thread(LPVOID para)
#006 {
#007     printf("Enter Thread\n");
#008     //等待 5s
#009     Sleep(5000);
#010     return 0;
#011 }
#012 int main()
#013 {
#014     HANDLE hThread[2];
#015     //开启两个线程
#016     hThread[0] = (HANDLE) _beginthreadex (NULL, 0, Thread, NULL, 0, NULL);
#017     hThread[1] = (HANDLE) _beginthreadex (NULL, 0, Thread, NULL, 0, NULL);
#018     //主线程等待两个子线程结束
#019     WaitForMultipleObjects(2, hThread, TRUE, INFINITE);
#020     return 0;
#021 }
```

8.5 内核模式下的同步对象

在内核模式下，有一系列的同步对象与用户模式下的同步对象相对应。在用户模式下，各个函数都是以句柄操作同步对象的。而在用户模式下，程序员无法获得真实同步对象的指针，而用一个句柄（其实是一个 32 位整数）代表这个对象。在内核模式下，程序员可以获得真实同步对象的指针。

每种同步对象在内核中都会对应一种数据结构。但是在内核模式下，程序员可以很自由地操作这些对象。但要注意仔细使用这些同步对象，否则会引起系统的死锁。

8.5.1 内核模式下的等待

在内核模式下，同样也有两个函数负责等待内核同步对象，分别是 `KeWaitForSingleObject` 和 `KeWaitForMultipleObjects` 函数。其实，用户模式下的 `WaitForSingleObject` 和 `WaitForMultipleObjects` 函数都是依靠内核模式下的 `KeWaitForSingleObject`

和 `KeWaitForMultipleObjects` 函数实现的。`KeWaitForSingleObject` 函数负责等待单个同步对象，其声明如下：

```
NTSTATUS
KeWaitForSingleObject(
    IN PVOID Object,
    IN KWAIT_REASON WaitReason,
    IN KPROCESSOR_MODE WaitMode,
    IN BOOLEAN Alertable,
    IN PLARGE_INTEGER Timeout OPTIONAL
);
```

内核模式下的 `KeWaitForSingleObject` 函数比用户模式下的 `WaitForSingleObject` 函数多了很多参数。

- 第一个参数 `Object` 是一个同步对象的指针，注意这里不是句柄。
- 第二个参数 `WaitReason` 表示等待的原因，一般设为 `Executive`。
- 第三个参数 `WaitMode` 是等待模式，说明这个函数是在用户模式下等待还是在内核模式下等待。此参数一般设为 `KernelMode`。
- 第四个参数 `Alertable` 指明等待是否是“警惕”的，一般设置为 `FALSE`。
- 最后一个参数是等待的时间。如果这个参数设为 `NULL`，则代表无限期的等待，直到等待的同步对象变为激发态。如果这个参数是一个指向 64 位的整数，这个整数则代表了需要等待的时间。当这个整数是正数时，这个数字代表未来的某一个时刻距离 1601 年 1 月 1 日所经历的时间。如果该整数是负数，这个数字代表从现在时刻算起经历的时间。时间以 100ns（纳米）为单位。

如果等待的同步对象变为激发态，这个函数会退出睡眠状态，并返回 `STATUS_SUCCESS`。如果这个函数是因为超时而退出，则会返回 `STATUS_TIMEOUT`。

`KeWaitForMultipleObjects` 负责在内核模式下等待多个同步对象，其声明如下：

```
NTSTATUS
KeWaitForMultipleObjects(
    IN ULONG Count,
    IN PVOID Object[],
    IN WAIT_TYPE WaitType,
    IN KWAIT_REASON WaitReason,
    IN KPROCESSOR_MODE WaitMode,
    IN BOOLEAN Alertable,
    IN PLARGE_INTEGER Timeout OPTIONAL,
    IN PKWAIT_BLOCK WaitBlockArray OPTIONAL
);
```

第一个参数 `Count` 为等待同步对象的个数，第二个参数 `Object` 是同步对象数组，第三个参数 `WaitType` 指示是等待任意一个同步对象还是等待所有的同步对象。剩下的参数 `WaitType`、`WaitReason`、`WaitMode`、`Alertable` 和 `Timeout` 的作用同 `KeWaitForMultipleObjects` 的参数完全一样。

如果这个函数是因为超时而退出，则返回 `STATUS_TIMEOUT`。如果是因为数组中其

中一个同步对象变为激发态，这个函数返回的状态码减去 STATUS_WAIT_0，就是激发的同步对象在数组中的索引号。下面的代码说了如何使用 KeWaitForMultipleObjects 函数。

```
#001 NTSTATUS status = KeWaitForMultipleObjects(...);
#002 if (NT_SUCCESS(status))
#003 {
#004     ULONG index = status - STATUS_WAIT_0;
#005 }
```

8.5.2 内核模式下开启多线程

一般在多线程中才需要同步处理机制，这里介绍一下如何在内核模式下创建新线程。内核函数 PsCreateSystemThread 负责创建新线程。该函数可以创建两种线程，一种是用户线程，一种是系统线程。用户线程是属于当前进程中的线程。当前进程指的是当前 I/O 操作的发起者。如果在 IRP_MJ_READ 的派遣函数中调用 PsCreateSystemThread 函数创建用户线程，新线程就属于调用 ReadFile 的进程。

系统线程不属于当前用户进程，而属于系统进程。系统进程是操作系统中一个特殊的进程。这个进程的 ID 一般为 4，读者可以用任务管理器查看该进程，如图 8-7 所示。



图 8-7 系统进程

驱动程序中的 DriverEntry 和 AddDevice 等函数都是被某个系统线程调用的。

PsCreateSystemThread 的函数声明如下：

```
NTSTATUS
PsCreateSystemThread(
    OUT PHANDLE ThreadHandle,
    IN ULONG DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
    IN HANDLE ProcessHandle OPTIONAL,
    OUT PCLIENT_ID ClientId OPTIONAL,
    IN PKSTART_ROUTINE StartRoutine,
    IN PVOID StartContext
);
```

- 第一个参数 ThreadHandle 用于输出, 这个参数得到新创建的线程句柄。
- 第二个参数 DesiredAccess 是创建的权限。
- 第三个参数 ObjectAttributes 是该线程的属性, 一般设为 NULL。
- 第四个参数 ProcessHandle 指定是创建用户线程还是系统进程。如果该值为 NULL, 则为创建系统线程。如果该值是一个进程句柄, 则新创建的线程属于这个指定的进程。DDK 提供的宏 NtCurrentProcess 可以得到当前进程的句柄。
- 第六个参数 StartRoutine 为新线程的运行地址。
- 第七个参数 StartContext 为新线程接收的参数。

在内核模式下, 创建的线程必须用函数 PsTerminateSystemThread 强制线程结束。否则该线程是无法自动退出的。

笔者这里介绍一种方法, 可以方便地让线程知道自己属于哪个进程。首先, 使用 IoGetCurrentProcess 函数得到当前线程。IoGetCurrentProcess 函数会得到一个 PEPROCESS 数据结构。PEPROCESS 数据结构记录进程的信息, 其中包括进程名。遗憾的是, 微软并没有在 DDK 中定义 PEPROCESS 结构, 但可以利用微软的符号表分析这个数据结构。笔者一般使用 WinDbg 查看这个数据结构, 关于 WinDbg 的用法, 第 23 章会给出详细的介绍。

在 Windows XP 中, PEPROCESS 结构的 0x174 偏移位置记录着进程名。因此, 可以用下面的办法查看当前进程名称。

```
#001     PEPROCESS pEProcess = IoGetCurrentProcess();
#002     PTSTR ProcessName = (PTSTR)((ULONG)pEProcess + 0x174);
#003     KdPrint(("This thread run in %s process!\n", ProcessName));
```

下面的代码演示了如何在驱动程序中创建线程。在例子中, 分别创建用户线程和系统线程。线程还显示出各自所属的进程名, 代码如下。

```
#001 VOID SystemThread(IN PVOID pContext)
#002 {
#003     KdPrint(("Enter SystemThread\n"));
#004     PEPROCESS pEProcess = IoGetCurrentProcess();
#005     PTSTR ProcessName = (PTSTR)((ULONG)pEProcess + 0x174);
#006     KdPrint(("This thread run in %s process!\n", ProcessName));
#007     KdPrint(("Leave SystemThread\n"));
#008     //结束线程
#009     PsTerminateSystemThread(STATUS_SUCCESS);
#010 }
#011
#012 VOID MyProcessThread(IN PVOID pContext)
#013 {
#014     KdPrint(("Enter MyProcessThread\n"));
#015     //得到当前进程
#016     PEPROCESS pEProcess = IoGetCurrentProcess();
#017     PTSTR ProcessName = (PTSTR)((ULONG)pEProcess + 0x174);
#018     KdPrint(("This thread run in %s process!\n", ProcessName));
#019
#020     KdPrint(("Leave MyProcessThread\n"));
#021     //结束线程
#022     PsTerminateSystemThread(STATUS_SUCCESS);
```



```
#023 }
#024 VOID CreateThread_Test()
#025 {
#026     HANDLE hSystemThread,hMyThread;
#027     //创建系统线程,该线程是 System 进程的线程
#028     NTSTATUS status = PsCreateSystemThread(&hSystemThread,0,NULL,NULL,NULL,
SystemThread,NULL);
#029     //创建进程线程,该线程是用户进程的线程
#030     status = PsCreateSystemThread(&hMyThread,0,NULL,NtCurrentProcess(),NULL,
MyProcessThread,NULL);
#031 }
```

第一个创建的线程是用户线程,它是在 IRP_MJ_DEVICE_CONTROL 的派遣函数中创建的。因此,这个线程属于 DeviceIOControl 所在的进程,也就是 Text.exe 进程。第二个创建的线程是系统线程,它属于系统进程。驱动程序的输出 log 信息如图 8-8 所示。

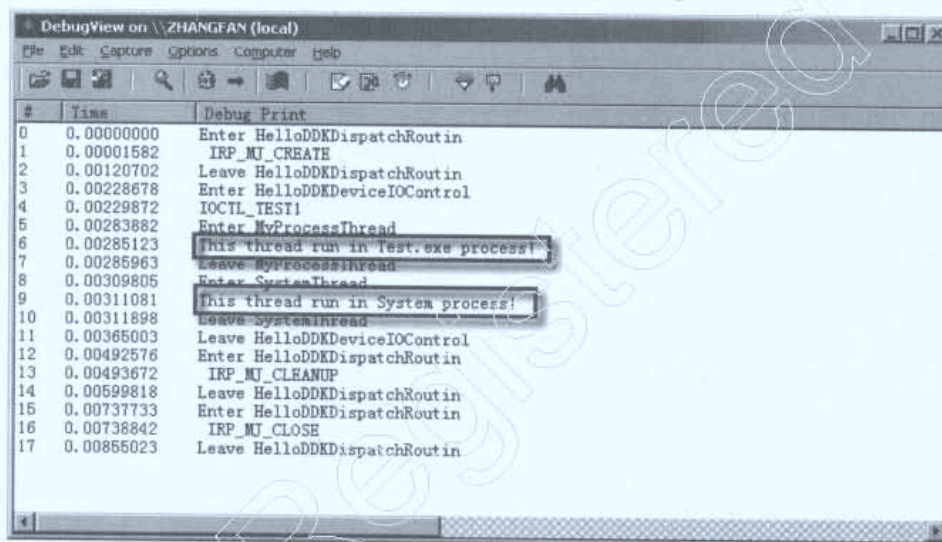


图 8-8 输出的 log

8.5.3 内核模式下的事件对象

在应用程序中,程序员只能操作事件句柄,无法得到事件对象的指针。在与事件相关的 Win32 API 函数的内部,会调用内核模式事件的相关函数。

在内核中,用 KEVENT 数据结构表示一个事件对象。在使用事件对象前,需要进行初始化。内核函数 KeInitializeEvent 负责对事件对象初始化,其声明如下:

```
VOID
KeInitializeEvent(
    IN PRKEVENT Event,
    IN EVENT_TYPE Type,
    IN BOOLEAN State
);
```

➤ 第一个参数 Event: 这个参数是初始化事件对象的指针。

- 第二个参数 **Type**: 这个参数是事件的类型。事件的类型分为两类，一类是“通知事件”，对应参数是 `NotificationEvent`。另一类是“同步事件”，对应参数是 `SynchronizationEvent`。
- 第三个参数 **State**: 这个参数如果为真，事件对象的初始化状态为激发状态。如果该参数为假，则事件对象的初始化状态为未激发状态。

如果创建的事件对象是“通知事件”，当事件对象变为激发态时，程序员需要手动将其改回未激发态。如果创建的事件对象是“同步事件”，当事件对象为激发态时，如遇到 `KeWaitForXX` 等内核函数，事件对象则自动变回未激发态。

下面的例子演示了如何在驱动程序中使用事件对象。这个例子首先创建一个事件对象，然后创建一个新线程，并将事件对象的指针传递给线程，主线程等待该事件对象。新创建的线程在完成任务后，将事件对象设置为激发状态，主线程得以继续运行。

```
#001 VOID MyProcessThread(IN PVOID pContext)
#002 {
#003     //获得事件指针
#004     PKEVENT pEvent = (PKEVENT)pContext;
#005     KdPrint(("Enter MyProcessThread\n"));
#006     //设置事件
#007     KeSetEvent(pEvent, IO_NO_INCREMENT, FALSE);
#008     KdPrint(("Leave MyProcessThread\n"));
#009     //结束线程
#010     PsTerminateSystemThread(STATUS_SUCCESS);
#011 }
#012 #pragma PAGEDCODE
#013 VOID Test()
#014 {
#015     HANDLE hMyThread;
#016     KEVENT kEvent;
#017     //初始化内核事件
#018     KeInitializeEvent(&kEvent, NotificationEvent, FALSE);
#019     //创建系统线程，该线程是 System 进程的线程
#020     NTSTATUS status = PsCreateSystemThread(&hMyThread, 0, NULL,
NtCurrentProcess(), NULL, MyProcessThread, &kEvent);
#021     //很重要，如果不等待，则 SystemThread 引用了本函数的栈上变量
#022     //当函数退出，同时栈上变量被回收，SystemThread 引用的参数会出现错误
#023     KeWaitForSingleObject(&kEvent, Executive, KernelMode, FALSE, NULL);
#024 }
```

8.5.4 驱动程序与应用程序交互事件对象

前面已经介绍过，应用程序中创建的事件和在内核中创建的事件对象，本质上是同一个东西。在用户模式时，它用句柄代表，在内核模式下，它用 `KEVENT` 数据结构代表。在应用程序中，所有内核对象都不会被用户看到，用户看到的只是代表内核对象的对象句柄。本节介绍的就是如何在应用程序和驱动程序中共同使用一个事件对象。

需要解决的第一个问题就是如何将用户模式下创建的事件传递给驱动程序。解决的办

Windows 驱动开发技术详解

法是采用 DeviceIoControl API 函数。在用户模式下创建一个同步事件，然后用 DeviceIoControl 把事件句柄传递给驱动程序。需要指出的是，句柄是与进程相关的，也就意味着一个进程中的句柄只能在这个进程中有效。句柄相当于事件对象在进程中的索引，通过这个索引操作系统就会得到事件对象的指针。DDK 提供了内核函数将句柄转化为指针，该函数是 ObReferenceObjectByHandle。

ObReferenceObjectByHandle 函数在得到指针的同时，会为对象的指针维护一个计数。每次调用 ObReferenceObjectByHandle 会使计数加 1。因此为计数平衡，在使用完 ObReferenceObjectByHandle 函数后，需要调用 ObDereferenceObject 函数。ObDereferenceObject 函数使计数减 1。

ObReferenceObjectByHandle 函数会返回一个状态值，表明是否成功得到指针。下面的例子演示了应用程序和驱动程序下如何交互事件对象。首先是用户模式的代码。

```
#001 Int main()
#002 {
#003     //打开设备
#004     HANDLE hDevice =
#005         CreateFile("\\\\.\\HelloDDK",
#006             GENERIC_READ | GENERIC_WRITE,
#007             0,
#008             NULL,
#009             OPEN_EXISTING,
#010             FILE_ATTRIBUTE_NORMAL,
#011             NULL );
#012     //判断设备是否成功打开
#013     if (hDevice == INVALID_HANDLE_VALUE)
#014     {
#015         printf("Failed to obtain file handle to device: "
#016             "%s with Win32 error code: %d\n",
#017             "MyWDMDevice", GetLastError() );
#018         return 1;
#019     }
#020     BOOL bSet;
#021     DWORD dwOutput;
#022     //创建用户模式同步事件
#023     HANDLE hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
#024     //建立辅助线程
#025     HANDLE hThread1 = (HANDLE) _beginthreadex(NULL, 0, Thread1, &hEvent, 0, NULL);
#026     //将用户模式的事件句柄传递给驱动
#027     bRet = DeviceIoControl(hDevice, IOCTL_TRANSMIT_EVENT, &hEvent,
sizeof(hEvent), NULL, 0, &dwOutput, NULL);
#028     //等待辅助线程结束
#029     WaitForSingleObject(hThread1, INFINITE);
#030     //关闭各个句柄
#031     CloseHandle(hDevice);
#032     CloseHandle(hThread1);
#033     CloseHandle(hEvent);
#034     return 0;
#035 }
```

下面是驱动程序的部分代码。

```

#001 NTSTATUS HelloDDKDeviceIOControl(IN PDEVICE_OBJECT pDevObj,
#002                                     IN PIRP pIrp)
#003 {
#004     NTSTATUS status = STATUS_SUCCESS;
#005     KdPrint(("Enter HelloDDKDeviceIOControl\n"));
#006     //获得当前 IO 堆栈
#007     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
#008     //获得输入参数大小
#009     ULONG cbin = stack->Parameters.DeviceIoControl.InputBufferLength;
#010     //获得输出参数大小
#011     ULONG cbout = stack->Parameters.DeviceIoControl.OutputBufferLength;
#012     //得到 IOCTL 码
#013     ULONG code = stack->Parameters.DeviceIoControl.IoControlCode;
#014     ULONG info = 0;
#015     switch (code)
#016     {
#017         // process request
#018         case IOCTL_TRANSMIT_EVENT:
#019         {
#020             KdPrint(("IOCTL_TEST1\n"));
#021             //得到应用程序传递进来的事件
#022             HANDLE hUserEvent = *(HANDLE*)pIrp->AssociatedIrp.SystemBuffer;
#023             PKEVENT pEvent;
#024             //由事件句柄得到内核事件数据结构
#025             status = ObReferenceObjectByHandle(hUserEvent, EVENT_MODIFY_STATE,
*ExEventObjectType,
#026             KernelMode, (PVOID*) &pEvent, NULL);
#027             //设置事件
#028             KeSetEvent(pEvent, IO_NO_INCREMENT, FALSE);
#029             //减小引用计数
#030             ObDereferenceObject(pEvent);
#031             break;
#032         }
#033         default:
#034             status = STATUS_INVALID_VARIANT;
#035     }
#036     // 设置 IRP 完成状态
#037     pIrp->IoStatus.Status = status;
#038     //设置 IRP 操作字节数
#039     pIrp->IoStatus.Information = info;
#040     //结束 IRP 请求
#041     IoCompleteRequest(pIrp, IO_NO_INCREMENT);
#042     KdPrint(("Leave HelloDDKDeviceIOControl\n"));
#043     return status;
#044 }

```

8.5.5 驱动程序与驱动程序交互事件对象

有时候，还需要在驱动程序与驱动程序中交互事件对象。例如，驱动程序 A 的某个派遣函数要与驱动程序 B 的派遣函数进行同步，就需要两个驱动程序之间交互事件对象。

关键的问题就是如何让驱动程序 A 获取驱动程序 B 中创建的事件对象。最简单的方法是驱动程序 B 创建一个有“名字”的事件对象，这样驱动程序 A 就可以根据“名字”寻找到事件对象的指针。

创建有名的事件可以通过 `IoCreateNotificationEvent` 和 `IoCreateSynchronizationEvent` 内核函数。其中, `IoCreateNotificationEvent` 函数创建“同步事件”对象, 而 `IoCreateSynchronizationEvent` 函数创建“通知事件”对象。“同步事件”和“同步对象”的区别已经在介绍 `KeInitializeEvent` 函数的时候介绍了。

这两个函数都用 `UNICODE_STRING` 字符串描述内核事件对象的名字, 如果内核中不存在指定名称的内核事件对象, 函数会创建该名称的内核事件对象。如果内核中存在指定名称的内核事件对象, 则函数会打开这个内核事件对象。这两个函数得到内核事件对象的句柄, 为了进一步得到内核事件对象的指针, 可以使用前面介绍的内核函数 `ObReferenceObjectByHandle`。

因此在不同的驱动程序中, 只要知道事件对象的名称, 就可以交互事件对象。给内核事件对象命名时, 最好不要用很简单的名字, 否则容易导致命名冲突。

8.5.6 内核模式下的信号灯

在内核中还有另一种同步对象, 这就是信号灯。和事件对象一样, 信号灯在用户模式下和内核模式下是完全统一的, 只不过操作方式不同。在用户模式下, 信号灯对象用句柄代表, 而在内核模式下, 信号灯对象用 `KSEMAPHORE` 数据结构表示。

在使用信号灯对象前, 需要对信号灯对象进行初始化。使用内核函数 `KeInitializeSemaphore` 对信号灯对象初始化, 其函数声明如下:

```
VOID  
KeInitializeSemaphore(  
    IN PKSEMAPHORE Semaphore,  
    IN LONG Count,  
    IN LONG Limit  
);
```

- 第一个参数 `Semaphore`: 这个参数获得内核信号灯对象指针。
- 第二个参数 `Count`: 这个参数是初始化时的信号灯计数。
- 第三个 `Limit`: 这个参数指明信号灯计数的上限值。

`KeReadStateSemaphore` 函数可以读取信号灯当前的计数。

释放信号灯会增加信号灯计数, 它对应的内核函数是 `KeReleaseSemaphore`。程序员可以用这个函数指定增量值。获得信号灯可以使用 `KeWaitXX` 系列函数, 如果能获得, 就将计数减一, 否则陷入等待。下面的代码演示了如何在驱动程序中使用信号灯对象。

```
#001 VOID MyProcessThread(IN PVOID pContext)  
#002 {  
#003     //得到信号灯  
#004     PKSEMAPHORE pkSemaphore = (PKSEMAPHORE)pContext;  
#005     KdPrint(("Enter MyProcessThread\n"));  
#006     KeReleaseSemaphore(pkSemaphore, IO_NO_INCREMENT, 1, FALSE);  
#007     KdPrint(("Leave MyProcessThread\n"));  
#008     //结束线程
```

```

#009     PsTerminateSystemThread (STATUS_SUCCESS);
#010 }
#011
#012 #pragma PAGEDCODE
#013 VOID Test()
#014 {
#015     HANDLE hMyThread;
#016     KSEMAPHORE kSemaphore;
#017     //初始化内核信号灯
#018     KeInitializeSemaphore(&kSemaphore,2,2);
#019     //读取信号灯状态
#020     LONG count = KeReadStateSemaphore(&kSemaphore);
#021     KdPrint(("The Semaphore count is %d\n",count));
#022     //等待信号灯
#023     KeWaitForSingleObject(&kSemaphore,Executive,KernelMode,FALSE,NULL);
#024     //读取信号灯状态
#025     count = KeReadStateSemaphore(&kSemaphore);
#026     KdPrint(("The Semaphore count is %d\n",count));
#027     KeWaitForSingleObject(&kSemaphore,Executive,KernelMode,FALSE,NULL);
#028     //读取信号灯状态
#029     count = KeReadStateSemaphore(&kSemaphore);
#030     KdPrint(("The Semaphore count is %d\n",count));
#031     //创建系统线程,该线程是 System 进程的线程
#032     NTSTATUS status = PsCreateSystemThread(&hMyThread,0,NULL,NtCurrentProcess(),
NULL,MyProcessThread,&kSemaphore);
#033
#034     //很重要,如果不等待,则 SystemThread 引用了本函数的栈上变量
#035     //当函数退出,同时栈上变量被回收, SystemThread 引用的参数会出现错误
#036     KeWaitForSingleObject(&kSemaphore,Executive,KernelMode,FALSE,NULL);
#037     KdPrint(("After KeWaitForSingleObject\n"));
#038 }

```

和事件对象一样,信号灯对象也可以在应用程序与驱动程序中交互。读者可以参考前面介绍的驱动程序与应用程序交互事件对象的例子。

8.5.7 内核模式下的互斥体

在内核中还有另一种同步对象,这就是互斥体对象。前面已经介绍了互斥体在应用程序中的使用,在内核中使用互斥体对象的方法是相似的。

互斥体在内核中的数据结构是 **KMUTEX**,使用前需要初始化互斥体对象。可以使用 **KeInitializeMutex** 内核函数初始化互斥体对象,其声明如下:

```

VOID
KeInitializeMutex(
    IN PRKMUTEX Mutex,
    IN ULONG Level
);

```

- 第一个参数 **Mutex**: 这个参数可以获得内核互斥体对象的指针。
- 第二个参数 **Level**: 保留值,一般设为 0。

初始化后的互斥体对象,就可以使线程之间互斥了。获得互斥体对象用 **KeWaitXX** 系

Windows 驱动开发技术详解

列内核函数，释放互斥体用 `KeReleaseMutex` 内核函数。

下面的例子演示了如何在驱动程序中使用互斥体对象。首先，这个例子创建两个线程，为了保证线程之间不并行运行，线程间使用了互斥体对象进行同步。

```
#001 VOID MyProcessThread1(IN PVOID pContext)
#002 {
#003     PKMUTEX pkMutex = (PKMUTEX)pContext;
#004     //获得互斥体
#005     KeWaitForSingleObject(pkMutex, Executive, KernelMode, FALSE, NULL);
#006     KdPrint(("Enter MyProcessThread1\n"));
#007     //强迫停止 50ms. 模拟执行一段代码, 模拟运行某段费时
#008     KeStallExecutionProcessor(50);
#009     KdPrint(("Leave MyProcessThread1\n"));
#010     //释放互斥体
#011     KeReleaseMutex(pkMutex, FALSE);
#012     //结束线程
#013     PsTerminateSystemThread(STATUS_SUCCESS);
#014 }
#015
#016 VOID MyProcessThread2(IN PVOID pContext)
#017 {
#018     PKMUTEX pkMutex = (PKMUTEX)pContext;
#019     //获得互斥体
#020     KeWaitForSingleObject(pkMutex, Executive, KernelMode, FALSE, NULL);
#021     KdPrint(("Enter MyProcessThread2\n"));
#022     //强迫停止 50ms. 模拟执行一段代码, 模拟运行某段费时
#023     KdPrint(("Leave MyProcessThread2\n"));
#024     //释放互斥体
#025     KeReleaseMutex(pkMutex, FALSE);
#026     //结束线程
#027     PsTerminateSystemThread(STATUS_SUCCESS);
#028 }
#029
#030 #pragma PAGEXCODE
#031 VOID Test()
#032 {
#033     HANDLE hMyThread1, hMyThread2;
#034     KMUTEX kMutex;
#035     //初始化内核互斥体
#036     KeInitializeMutex(&kMutex, 0);
#037     //创建系统线程, 该线程是 System 进程的线程
#038     PsCreateSystemThread(&hMyThread1, 0, NULL, NtCurrentProcess(), NULL,
MyProcessThread1, &kMutex);
#039     PsCreateSystemThread(&hMyThread2, 0, NULL, NtCurrentProcess(), NULL,
MyProcessThread2, &kMutex);
#040     PVOID Pointer_Array[2];
#041     //得到对象指针
#042     ObReferenceObjectByHandle(hMyThread1, 0, NULL, KernelMode, &Pointer_
Array[0], NULL);
#043     ObReferenceObjectByHandle(hMyThread2, 0, NULL, KernelMode, &Pointer_
Array[1], NULL);
#044     //等待多个事件
#045     KeWaitForMultipleObjects(2, Pointer_Array, WaitAll, Executive, KernelMode,
FALSE, NULL, NULL);
#046     //减小引用计数
```

```
#047 ObDereferenceObject(Pointer_Array[0]);
#048 ObDereferenceObject(Pointer_Array[1]);
#049 KdPrint(("After KeWaitForMultipleObjects\n"));
#050 }
```

运行此程序后，查看驱动程序输出的 log 信息，如图 8-9 所示。从图 8-9 中可以看出，两个线程是以串行方式运行的，这意味着成功地进行了同步。

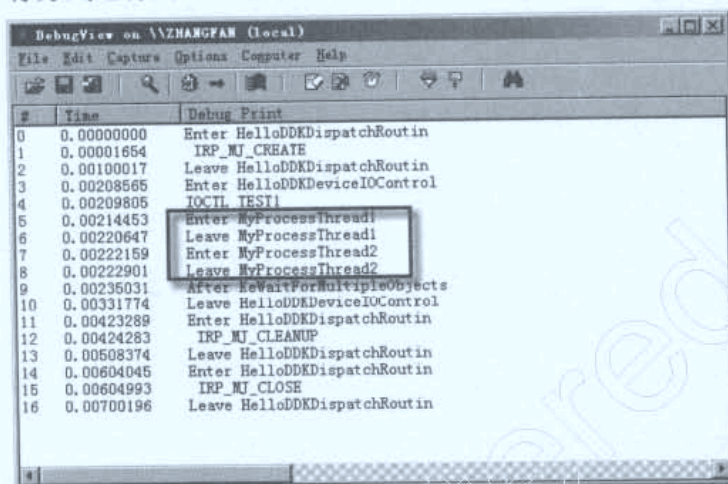


图 8-9 输出 log

另外，和事件对象一样，互斥体对象也可以在应用程序与驱动程序中进行交互。读者可以参考前面介绍的驱动程序与应用程序交互事件对象的例子。

8.5.8 快速互斥体

快速互斥体（Fast Mutex）是 DDK 提供的另外一种内核同步对象，它的特性类似前面介绍的普通互斥体对象。快速互斥体和普通互斥体起到的作用完全一样，之所以被称为快速互斥体，是因为执行的速度比普通互斥体速度快（这里指的是获取和释放的速度）。然而，快速互斥体比普通互斥体多了一个缺点，就是不能递归地获取互斥体对象。递归获取指的是，已经获得互斥体的线程，可以再次获得这个互斥体。换句话说，互斥体只互斥其他线程，而不互斥自己所在的线程。但是快速互斥体则不允许出现递归的情况。

普通互斥体在内核中是用 MUTEX 数据结构描述的，而快速互斥体在内核中是用 FAST_MUTEX 数据结构描述的。

除此之外，对快速互斥体的初始化、获取和释放对应的内核函数也和普通互斥体不同。初始化快速互斥体的内核函数是 `ExInitializeFastMutex`，获取快速互斥体的内核函数是 `ExAcquireFastMutex`，释放快速互斥体的内核函数是 `ExReleaseFastMutex`。下面的例子演示了如何在驱动程序中使用快速互斥体。

```
#001 VOID MyProcessThread1(IN PVOID pContext)
#002 {
#003     PFAST_MUTEX pFastMutex = (PFAST_MUTEX)pContext;
```



```

#004    //获得快速互斥体
#005    ExAcquireFastMutex(pFastMutex);
#006    KdPrint(("Enter MyProcessThread1\n"));
#007    //强迫停止 50ms, 模拟执行一段代码, 模拟运行某段费时
#008    KdPrint(("Leave MyProcessThread1\n"));
#009    //释放快速互斥体
#010    ExReleaseFastMutex(pFastMutex);
#011    //结束线程
#012    PsTerminateSystemThread(STATUS_SUCCESS);
#013    }
#014
#015    VOID MyProcessThread2(IN PVOID pContext)
#016    {
#017        PFAST_MUTEX pFastMutex = (PFAST_MUTEX)pContext;
#018        //获得快速互斥体
#019        ExAcquireFastMutex(pFastMutex);
#020        KdPrint(("Enter MyProcessThread2\n"));
#021        //强迫停止 50ms, 模拟执行一段代码, 模拟运行某段费时
#022        KdPrint(("Leave MyProcessThread2\n"));
#023        //释放快速互斥体
#024        ExReleaseFastMutex(pFastMutex);
#025        //结束线程
#026        PsTerminateSystemThread(STATUS_SUCCESS);
#027    }
#028
#029    #pragma PAGEDCODE
#030    VOID Test()
#031    {
#032        HANDLE hMyThread1, hMyThread2;
#033        FAST_MUTEX fastMutex;
#034        //初始化快速互斥体
#035        ExInitializeFastMutex(&fastMutex);
#036        //创建系统线程, 该线程是 System 进程的线程
#037        PsCreateSystemThread(&hMyThread1, 0, NULL, NtCurrentProcess(), NULL,
MyProcessThread1, &fastMutex);
#038        PsCreateSystemThread(&hMyThread2, 0, NULL, NtCurrentProcess(), NULL,
MyProcessThread2, &fastMutex);
#039        PVOID Pointer_Array[2];
#040        //获得对象指针
#041        ObReferenceObjectByHandle(hMyThread1, 0, NULL, KernelMode, &Pointer_
Array[0], NULL);
#042        ObReferenceObjectByHandle(hMyThread2, 0, NULL, KernelMode, &Pointer_
Array[1], NULL);
#043        KeWaitForMultipleObjects(2, Pointer_Array, WaitAll, Executive, KernelMode,
FALSE, NULL, NULL);
#044        //减小引用计数
#045        ObDereferenceObject(Pointer_Array[0]);
#046        ObDereferenceObject(Pointer_Array[1]);
#047        KdPrint(("After KeWaitForMultipleObjects\n"));
#048    }

```

8.6 其他同步方法

上一节主要介绍了内核中主要的同步对象。除了使用同步对象外, 还有几种方法可以

实现同步。本节将对这几种方法进行介绍。

8.6.1 使用自旋锁进行同步

在驱动程序中，经常使用自旋锁作为一种有效的同步机制。例如，在应用程序中打开一个设备后，有时需要开启多个线程去操作设备（例如，都调用 `ReadFile` 函数对设备进行读取操作）。这时候，`IRP_MJ_READ` 的派遣函数也会并发执行。但是大部分设备没有能力响应并发的读请求，必须完成一个读请求后再完成另一个读请求。这时候就需要进行同步处理，程序员可以选择采用前面介绍的事件、信号灯、互斥体等内核同步对象，但还有另外一种选择，这就是使用自旋锁。

对于要同步的代码，需要用同一把自旋锁进行同步。如果程序得到了自旋锁，其他程序希望获取自旋锁时，则不停地进入自旋状态。获得自旋锁的内核函数是 `KeAcquireSpinLock`。直到自旋锁被释放后，另外的程序才能获取到自旋锁。释放自旋锁的内核函数是 `KeReleaseSpinLock`。

如果希望同步某段代码区域，需要在这段代码区域前获取自旋锁，在代码区域后释放自旋锁。在单 CPU 的系统中，获取自旋锁是通过提升 `IRQL` 实现的。而在多 CPU 的系统中，实现方法比较复杂，有兴趣的读者可以自己研究一下。

无法获得自旋锁的线程会不停地自旋，这会浪费很多 CPU 时间。因此需要同步的代码区域不能过长，换句话说就是占有自旋锁的时间不宜过长。

下面的代码模拟了应用程序创建一个设备后，同时开启多个线程对设备进行请求的情况。这个例子采用的同步机制就是使用自旋锁。

```
#001 #include <windows.h>
#002 #include <process.h> /* _beginthread, _endthread */
#003 #include <stdio.h>
#004 #include <winioctl.h>
#005 #include "..\NT_Driver\Ioctl.h"
#006
#007 UINT WINAPI Thread1(LPVOID pContext)
#008 {
#009     BOOL bRet;
#010     DWORD dwOutput;
#011     //发送 IOCTL 码
#012     bRet = DeviceIoControl(*(PHANDLE)pContext, IOCTL_TEST1, NULL, 0, NULL, 0,
&dwOutput, NULL);
#013     return 0;
#014 }
#015
#016 int main()
#017 {
#018     //打开设备
#019     HANDLE hDevice =
#020         CreateFile("\\\\.\\HelloDDK",
#021             GENERIC_READ | GENERIC_WRITE,
#022             0,
#023             NULL,
```



```
#024             OPEN_EXISTING,
#025             FILE_ATTRIBUTE_NORMAL,
#026             NULL );
#027 //判断是否成功打开设备句柄
#028 if (hDevice == INVALID_HANDLE_VALUE)
#029 {
#030     printf("Failed to obtain file handle to device: "
#031           "%s with Win32 error code: %d\n",
#032           "MyWDMDevice", GetLastError() );
#033     return 1;
#034 }
#035
#036 HANDLE hThread[2];
#037 //开启两个新线程，每个线程都去执行 DeviceIoControl
#038 //因此在 IRP_MJ_DEVICE_CONTROL 的派遣函数会并行运行
#039 //为了让派遣函数不并行运行，而是串行运行，必须进行同步处理！
#040 //本例在派遣函数中采用自旋锁进行同步处理
#041 hThread[0] = (HANDLE) _beginthreadex (NULL,0,Thread1,&hDevice,0,NULL);
#042 hThread[1] = (HANDLE) _beginthreadex (NULL,0,Thread1,&hDevice,0,NULL);
#043 //等待两个进程全部运行完毕
#044 WaitForMultipleObjects(2,hThread,TRUE,INFINITE);
#045 //关闭句柄
#046 CloseHandle(hThread[0]);
#047 CloseHandle(hThread[1]);
#048 CloseHandle(hDevice);
#049 return 0;
#050 }
```

驱动程序的派遣函数需要进行同步处理，下面是示例代码。

```
#001 NTSTATUS HelloDDKDeviceIOControl(IN PDEVICE_OBJECT pDevObj,
#002                                     IN PIRP pIrp)
#003 {
#004     //为了避免多个派遣函数并行运行，所以进行同步处理
#005     //此处采用自旋锁处理同步
#006     //DeviceIoControl 调用，来源自用户线程，因此处于 PASSIVE_LEVEL
#007     ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);
#008
#009     PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
#010     KIRQL oldirql;
#011     KeAcquireSpinLock(&pdx->My_SpinLock,&oldirql); //获得自旋锁
#012     //A点=====
#013     //从A点到B点认为同步区域，不会被其他派遣函数
#014     NTSTATUS status = STATUS_SUCCESS;
#015     KdPrint(("Enter HelloDDKDeviceIOControl\n"));
#016     //使用自旋锁后，IRQL 提升至 DISPATCH_LEVEL
#017     ASSERT(KeGetCurrentIrql() == DISPATCH_LEVEL);
#018     //设置 IRP 完成状态
#019     pIrp->IoStatus.Status = status;
#020     //设置 IRP 操作字节数
#021     pIrp->IoStatus.Information = 0;
#022     //结束 IRP 请求
#023     IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#024     KdPrint(("Leave HelloDDKDeviceIOControl\n"));
#025     //B点=====
#026     KeReleaseSpinLock(&pdx->My_SpinLock,oldirql); //获得自旋锁
```

```
#027
#028     return status;
#029 }
```

因为正确使用自旋锁，派遣函数正确的处理了同步处理问题。

8.6.2 使用互锁操作进行同步

在本章开始曾经引出一个例子，来描述同步处理的必要性。回忆这个例子，C 语言中变量自增的语句，会被编译器编译成一段汇编指令。例如，下面的代码在多线程环境中，就存在“条件竞争”问题。语句 `number++` 不是执行的最小单位，最小的执行单位是汇编指令。每条汇编指令都有可能被打断。出现这个问题的原因是语句 `number++` 不是最小的执行单位。

```
#001 int number=0;
#002 void Foo()
#003 {
#004     number++;
#005     //做一些事情...
#006     number--;
#007 }
```

为了让 `number++` 称为最小的执行单位，保证运行的原子性，可以采用很多种办法。例如，可以使用自旋锁，下面的代码就是更改后的代码。

```
#001 int number=0;
#002 void Foo()
#003 {
#004     //获取自旋锁
#005     KeAcquireSpinLock(...);
#006     number++;
#007     //释放自旋锁
#008     KeReleaseSpinLock(...);
#009
#010     //做一些事情....
#011     //获取自旋锁
#012     KeAcquireSpinLock(...);
#013     number--;
#014     //释放自旋锁
#015     KeReleaseSpinLock(...);
#016 }
```

上面的办法采用自旋锁进行同步，需要同步的区域就是 `number++` 或者 `number--`。对于变量自增、自减需要同步时，可以使用 DDK 提供的 `InterlockedXX` 和 `ExInterlockedXX` 函数。再将上面的代码改成下面的样子。

```
#001 int number=0;
#002 void Foo()
#003 {
#004     //原子方式的自增
#005     InterlockedIncrement(&number);
#006     //做一些其他事情..
#007     //原子方式的子减
```



```
#008     InterlockedDecrement(&number);
#009 }
```

DDK 提供了两类互锁操作来提供简单的同步处理，一类是 InterlockedXX 函数，另一类是 ExInterlockedXX 函数。其中，InterlockedXX 系列的函数不通过自旋锁实现，而 ExInterlockedXX 系列函数通过自旋锁实现。InterlockedXX 系列函数不需要程序员提供自旋锁，内部不会提升 IRQL，因此 InterlockedXX 函数可以操作非分页的数据，也可以操作分页的数据。而 ExInterlockedXX 需要程序员提供一个自旋锁，内部依靠这个自旋锁实现同步，所有 ExInterlockedXX 不能操作分页内存的数据。

表 8-2 列出了 DDK 提供的 ExInterlockedXX 系列互锁函数及其功能。表 8-3 列出了 DDK 提供的 InterlockedXX 系列互锁函数及其功能。

表 8-2 ExInterlockedXX 互锁操作函数

内核函数	功 能
ExInterlockedAddLargeInteger	64 位整数加法互锁操作
ExInterlockedAddLargeStatistic	64 位整数加法互锁操作
ExInterlockedAddUlong	32 位整数加法互锁操作
ExInterlockedAllocateFromZone	分配互锁操作
ExInterlockedCompareExchange64	两个 32 位整数互换互锁操作
ExInterlockedDecrementLong	32 位整数减法互锁操作
ExInterlockedExchangeAddLargeInteger	64 位整数加法互锁操作
ExInterlockedExchangeUlong	两个整数交换互锁操作
ExInterlockedFlushSList	删除链表全部元素的互锁操作
ExInterlockedIncrementLong	32 位整数自增互锁操作
ExInterlockedInsertHeadList	插入双向链表互锁操作
ExInterlockedInsertTailList	插入双向链表互锁操作
ExInterlockedPopEntryList	删除单向链表互锁操作
ExInterlockedPopEntrySList	删除单向链表互锁操作
ExInterlockedPushEntryList	插入单向链表互锁操作
ExInterlockedPushEntrySList	插入单向链表互锁操作
ExInterlockedRemoveHeadList	插入双向链表互锁操作

表 8-3 InterlockedXX 互锁操作函数

内核函数	功 能
InterlockedCompareExchange	比较互锁操作
InterlockedCompareExchangePointer	比较互锁操作
InterlockedDecrement	整型自减互锁操作
InterlockedExchange	整型交换互锁操作
InterlockedExchangeAdd	两个整型相加互锁操作
InterlockedExchangePointer	为指针赋值互锁操作
InterlockedIncrement	整型自增互锁操作

8.7 小结

本章介绍了驱动程序中常用的同步处理办法，并且将内核模式下的同步处理方法和用户模式下的同步处理方法做了比较。另外，本章还介绍了中断请求级、自旋锁等同步处理机制。驱动程序编程中经常会用到同步处理方法，驱动程序员应该正确使用这些方法，避免系统的死锁。后面的章节还会对同步处理机制做进一步介绍。

UnRegistered



第 9 章 IRP 的同步

对设备的任何操作都会最终转化为 IRP 请求,而 IRP 一般都是由操作系统异步发送的。本章介绍了如何异步处理 IRP 请求。异步处理 IRP 有助于提高效率,但有时候异步处理会导致逻辑上的错误,这时候就需要将异步的 IRP 进行同步化。将 IRP 同步化有很多办法,这包括使用 StartIO 例程、使用中断服务例程等。本章将主要对 IRP 的同步化进行详细介绍。

9.1 应用程序对设备的同步异步操作

大部分 IRP 都是由应用程序的 Win32 API 函数发起的。这些 Win32 API 本身就支持同步和异步操作。例如, ReadFile、WriteFile 和 DeviceIOControl 等,这些都有两种操作方式,一种是同步操作,另一种是异步操作。本节将对这些操作方式进行介绍。

9.1.1 同步操作与异步操作原理

操作设备的 Win32 API 主要是三个函数,即 ReadFile 函数、WriteFile 函数和 DeviceIOControl 函数。以 DeviceIoControl 函数为例,它的同步操作如图 9-1 所示。

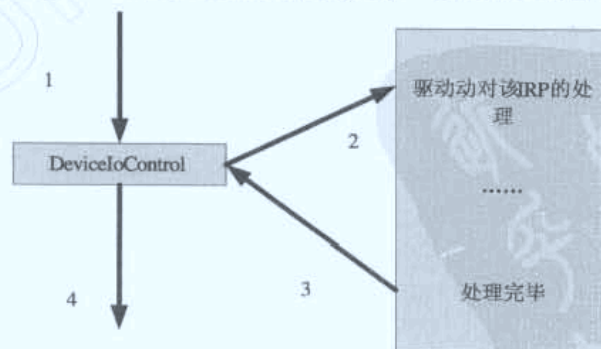


图 9-1 IRP 同步操作示意图

当应用程序调用 DeviceIoControl 函数时，它的内部会创建一个 IRP_MJ_DEVICE_CONTROL 类型的 IRP，并将这个 IRP 传送到驱动程序的派遣函数中。处理该 IRP 需要一段时间，直到 IRP 处理完毕后，DeviceIoControl 函数才会返回。

同步操作时，在 DeviceIoControl 的内部，会调用 WaitForSingleObject 函数去等待一个事件。这个事件直到 IRP 被结束时，才会被触发。如果通过反汇编 IoCompleteRequest 内核函数，就会发现在 IoCompleteRequest 内部设置了该事件。DeviceIoControl 会暂时进入睡眠状态，直到 IRP 被结束。而对于异步操作，它的处理过程如图 9-2 所示。

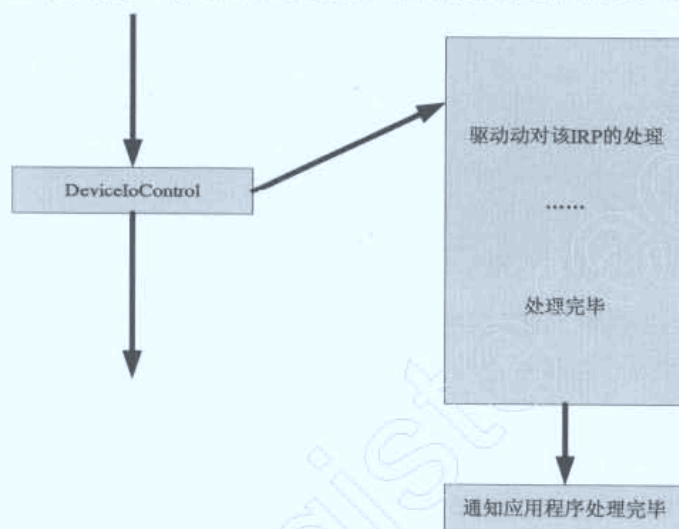


图 9-2 IRP 异步操作示意图

在异步操作的情况下，当 DeviceIoControl 被调用时，其内部会产生 IRP，并将该 IRP 传递给驱动内部的派遣函数。但此时 DeviceIoControl 函数不会等待该 IRP 结束，而是直接返回。当 IRP 经过一段时间被结束时，操作系统会触发一个 IRP 相关事件。这个事件可以通知应用程序 IRP 请求被执行完毕。

下面举个例子来比较同步与异步操作的区别。例如，DeviceIoControl 请求驱动程序去执行格式化磁盘的操作，这个操作假设为十分钟。如果 DeviceIoControl 采用同步操作时，DeviceIoControl 函数从被调用到返回之间需要十分钟的时间。因为，DeviceIoControl 中途一直在等待这个操作完成。

而如果 DeviceIoControl 函数采用异步操作时，执行 DeviceIoControl 的时间可能会忽略不计，应用程序的线程可以先执行别的操作。但这并不意味着 DeviceIoControl 的操作真正已经完毕，十分钟后操作系统会通过一个事通知应用程序，刚才的 DeviceIoControl 操作才真正完毕。

可以看出，异步操作比同步操作的效率高。其实对于整个 Windows 操作系统而言，它就是一个完全支持异步的操作系统。异步操作比同步操作在性能上优越了许多，但从编程的角度看，异步操作比同步操作复杂一些。

Windows 驱动开发技术详解

Win32 API 的异步操作还必须得到驱动程序的支持, 如果驱动程序不支持异步操作, Win32 API 的异步操作将会失败。

9.1.2 同步操作设备

如果需要同步操作设备, 在打开设备的时候就要指定以“同步”的方式打开设备。打开设备用 `CreateFile` 函数, 其函数声明如下:

```
HANDLE CreateFile(  
    LPCTSTR lpFileName,           // 设备名  
    DWORD dwDesiredAccess,        // 访问权限  
    DWORD dwShareMode,            // 共享模式  
    LPSECURITY_ATTRIBUTES lpSecurityAttributes, // 安全属性  
    DWORD dwCreationDisposition,  // 如何创建  
    DWORD dwFlagsAndAttributes,   // 设备属性  
    HANDLE hTemplateFile          // 文件模板  
);
```

`CreateFile` 函数在前面已经多次用到, 其中第六个参数 `dwFlagsAndAttributes` 是同步异步操作的关键。如果这个参数中没有设置 `FILE_FLAG_OVERLAPPED`, 则以后对该设备的操作都是同步操作, 否则所有操作为异步操作。在前面各章节介绍的例子都没有设置 `FILE_FLAG_OVERLAPPED`, 因此都是同步操作。

对设备操作的 Win32 API, 例如, `ReadFile`、`WriteFile` 和 `DeviceIoControl` 函数, 都会提供一个 `OVERLAP` 参数, 如 `ReadFile` 函数:

```
BOOL ReadFile(  
    HANDLE hFile,                 // 设备句柄  
    LPVOID lpBuffer,              // 读取的缓冲区  
    DWORD nNumberOfBytesToRead,   // 读取的大小  
    LPDWORD lpNumberOfBytesRead,  // 实际读取的大小  
    LPOVERLAPPED lpOverlapped    // overlapp 参数  
);
```

在同步操作设备时, 其 `lpOverlapped` 参数(也就是 `OVERLAP` 结构指针)设置为 `NULL`。下面的代码演示了应用程序如何对设备进行“同步”读取。这里以操作文件代替操作设备。文件可以看成是广义上的设备。

```
#001 int main()  
#002 {  
#003     //打开设备, 这里以打开文件作为演示  
#004     HANDLE hDevice =  
#005         CreateFile("test.dat",  
#006             GENERIC_READ | GENERIC_WRITE,  
#007             0,  
#008             NULL,  
#009             OPEN_EXISTING,  
#010             FILE_ATTRIBUTE_HIDDEN, //此处没有设置 FILE_FLAG_OVERLAPPED  
#011             NULL );
```

```

#012    //判断是否打开成功
#013    if (hDevice == INVALID_HANDLE_VALUE)
#014    {
#015        printf("Read Error\n");
#016        return 1;
#017    }
#018    UCHAR buffer[BUFFER_SIZE];
#019    DWORD dwRead;
#020    //读设备
#021    ReadFile(hDevice,buffer,BUFFER_SIZE,&dwRead,NULL); //这里没有设置 OVERLAP 参数
#022    //关闭设备句柄
#023    CloseHandle(hDevice);
#024    return 0;
#025 }

```

此段代码可以在配套光盘中本章的 SyncOperate 目录下找到。

9.1.3 异步操作设备（方式一）

异步操作设备时主要需要设置 OVERLAP 参数，Windows 中用一种数据结构 OVERLAPPED 表示。

```

typedef struct _OVERLAPPED {
    ULONG_PTR Internal;
    ULONG_PTR InternalHigh;
    DWORD Offset;
    DWORD OffsetHigh;
    HANDLE hEvent;
} OVERLAPPED;

```

- 第三个参数 Offset: 操作设备会指定一个偏移量，从设备的偏移量进行读取。该偏移量用一个 64 位整型表示，Offset 就是偏移量的低 32 位整型。
- 第四个参数 OffsetHigh: OffsetHigh 是偏移量的高 32 位整数。
- 第五个参数 hEvent: 这个事件用于该操作完成后通知应用程序。程序员可以初始化该事件为未激发，当操作设备结束后，即在驱动中调用 IoCompleteRequest 后，设置该设备为激发态。

在使用 OVERLAPPED 结构前，要先对其内部清零，并且为其创建事件。

下面的代码演示了如何在应用程序中使用异步操作。

```

#001 int main()
#002 {
#003     //打开设备（文件）
#004     HANDLE hDevice =
#005         CreateFile("test.dat",
#006             GENERIC_READ | GENERIC_WRITE,
#007             0,
#008             NULL,
#009             OPEN_EXISTING,
#010             FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, //注意此处必
//须加入 FILE_FLAG_OVERLAPPED，表示使用异常方式打开设备
#011     NULL );

```



```
#012     if (hDevice == INVALID_HANDLE_VALUE)
#013     {
#014         printf("Read Error\n");
#015         return 1;
#016     }
#017     UCHAR buffer[BUFFER_SIZE];
#018     DWORD dwRead;
#019     //初始化 overlap 使其内部全部为零
#020     OVERLAPPED overlap={0};
#021     //创建 overlap 事件
#022     overlap.hEvent = CreateEvent(NULL,FALSE,FALSE,NULL);
#023     //这里没有设置 OVERLAP 参数, 因此是异步操作
#024     ReadFile(hDevice,buffer,BUFFER_SIZE,&dwRead,&overlap);
#025
#026     //做一些其他操作, 这些操作会与读设备并行执行
#027     //等待读设备结束
#028     WaitForSingleObject(overlap.hEvent,INFINITE);
#029     //关闭设备句柄
#030     CloseHandle(hDevice);
#031     return 0;
#032 }
```

此段代码可以在配套光盘中本章的 AsyncOperate1 目录下找到。

9.1.4 异步操作设备（方式二）

除了 ReadFile 和 WriteFile 函数外, 还有两个 API 也可以实现异步读写, 这就是 ReadFileEx 和 WriteFileEx 函数。ReadFile 和 WriteFile 既可以支持同步读写操作, 又可以支持异步读写操作。而 ReadFileEx 和 WriteFileEx 函数是专门用于异步读写操作的, 不能进行同步读写。首先看一下 ReadFileEx 的声明。

```
BOOL ReadFileEx(
    HANDLE hFile,                                // 设备句柄
    LPVOID lpBuffer,                             // 读取缓冲区
    DWORD nNumberOfBytesToRead,                  // 读取的字节数
    LPOVERLAPPED lpOverlapped,                  // offset 指针
    LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine // 完成函数
);
```

- 第一个参数 hFile: 要操作的设备句柄。
- 第二个参数 lpBuffer: 读入数据的缓冲区。
- 第三个参数 nNumberOfBytesToRead: 需要读取的字节数。
- 第四个参数 lpOverlapped: 一个 OVERLAPPED 指针。
- 第五个参数 lpCompletionRoutine: 完成例程。

需要注意的是, 这里提供的 OVERLAPPED 不需要提供事件句柄。ReadFileEx 将读请求传递到驱动程序后立刻返回。驱动程序在结束读操作后, 会通过调用 ReadFileEx 提供的回调例程 (Call Back Function)。这类似一个软中断, 也就是当读操作结束后, 系统立刻回调 ReadFileEx 提供的回调例程。Windows 将这种机制称为异步过程调用 (APC Asynchronous

Procedure Call)。

然而，APC 的回调函数被调用是有条件的。只有线程处于警惕 (Alert) 状态时，回调函数才有可能被调用。有多个 API 可以使系统进入警惕状态，如 SleepEx、WaitForSingleObjectEx、WaitForMultipleObjectsEx 函数等。这些 Win32 API 都会有一个 BOOL 型的参数 bAlertable，当设置为 TRUE 时，就进入警惕模式。

当系统进入警惕模式后，操作系统会枚举当前线程的 APC 队列。驱动程序一旦结束读取操作，就会把 ReadFileEx 提供的完成例程插入到 APC 队列。

回调例程会报告本次操作的完成状况，比如是成功或是失败。同时会报告本次读取操作实际读取的字节数等。下面是一般回调例程的声明。

```
VOID CALLBACK FileIOCompletionRoutine(
    DWORD dwErrorCode,
    DWORD dwNumberOfBytesTransferred,
    LPOVERLAPPED lpOverlapped
);
```

- 第一个参数 dwErrorCode: 如果读取错误，会返回错误码。
- 第二个参数 dwNumberOfBytesTransferred: 返回实际读取操作的字节数。
- 第三个参数 lpOverlapped: OVERLAP 参数，指定读取的偏移量等信息。

下面的代码演示了如何在应用程序中 ReadFileEx 进行异步读操作。

```
#001 VOID CALLBACK MyFileIOCompletionRoutine(
#002     DWORD dwErrorCode,           // 对于此次操作返回的状态
#003     DWORD dwNumberOfBytesTransferred, // 告诉已经操作了多少字节,也就是在 IRP 里的
Infomation
#004     LPOVERLAPPED lpOverlapped // 这个数据结构
#005 )
#006 {
#007     printf("IO operation end!\n");
#008 }
#009
#010 int main()
#011 {
#012     //打开设备(文件)
#013     HANDLE hDevice =
#014         CreateFile("test.dat",
#015             GENERIC_READ | GENERIC_WRITE,
#016             0,
#017             NULL,
#018             OPEN_EXISTING,
#019             FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, //注意此处必
//须加入 FILE_FLAG_OVERLAPPED, 表示使用异常方式打开设备
#020             NULL );
#021     if (hDevice == INVALID_HANDLE_VALUE)
#022     {
#023         printf("Read Error\n");
#024         return 1;
#025     }
#026     UCHAR buffer[BUFFER_SIZE];
#027     //初始化 overlap 使其内部全部为零
#028     //不用初始化事件
```



```
#029     OVERLAPPED overlap={0};
#030     //这里没有设置 OVERLAP 参数, 因此是异步操作
#031     ReadFileEx(hDevice, buffer,
BUFFER_SIZE, &overlap, MyFileIOCompletionRoutine);
#032     //做一些其他操作, 这些操作会与读设备并行执行
#033     //进入 alterable
#034     SleepEx(0, TRUE);
#035     //关闭句柄
#036     CloseHandle(hDevice);
#037     return 0;
#038 }
```

此段代码可以在配套光盘中本章的 AsyncOperate2 目录下找到。

9.2 IRP 的同步完成与异步完成

在 9.1 节中介绍了如何在应用程序中对设备进行同步、异步操作。但是这些同步、异步操作必须得到驱动程序的支持。所有对设备的操作都会转化为 IRP 请求, 并传递到相应的派遣函数中。可以有两种方式处理 IRP 请求, 第一种是在派遣函数中直接结束 IRP 请求, 可以认为这是一种同步处理的方法, 前面介绍的例子都是基于这种处理方法, 也是最简单的处理方式。另外一种方法是在派遣函数中不结束 IRP 请求, 而是让派遣函数直接返回。IRP 在以后的某个时候再进行处理。异步处理的方式是本节介绍的重点。

9.2.1 IRP 的同步完成

下面将介绍 Win32 API 函数是如何一层层通过调用进入到派遣函数的。

- ① 在应用程序中调用 CreateFile Win32 API 函数, 这个函数用于打开设备。
- ② CreateFile Win32 API 函数内部调用了 ntdll.dll 中的 NtCreateFile 函数。
- ③ ntdll.dll 中的 NtCreateFile 函数进入内核模式, 然后调用 ntoskrnl.exe 中的 NtCreateFile 函数。
- ④ 内核模式中 ntoskrnl.exe 的 NtCreateFile 函数创建 IRP_MJ_CREATE 类型的 IRP, 让后调用相应驱动程序的派遣函数, 并将 IRP 的指针传递给该派遣函数。
- ⑤ 派遣函数调用 IoCompleteRequest, 将 IRP 请求结束。
- ⑥ 操作系统按照原路返回, 一直退到 CreateFile Win32 API 函数。至此 CreateFile 函数返回。
- ⑦ 如果需要读取设备, 应用程序应该调用 ReadFile Win32 API 函数。
- ⑧ ReadFile Win32 API 函数会调用 ntdll.dll 中的 NtReadFile 函数。
- ⑨ ntdll.dll 中的 NtReadFile 函数会进入内核模式, 调用 ntoskrnl.exe 中的 NtReadFile 函数。
- ⑩ ntoskrnl.exe 中的 NtReadFile 函数创建 IRP_MJ_READ 类型的 IRP, 并将其传入相应的派遣函数中。

对设备进行读取可以有三种方法，第一种方法是用 `ReadFile` 函数进行同步读取，第二种方法是用 `ReadFile` 函数进行异步读取，第三种方法是用 `ReadFileEx` 函数进行异步读取。

如果是用 `ReadFile` 函数进行同步读取时：

① `ReadFile` 函数内部会创建一个事件，这个事件连同 IRP 一起被传递到派遣函数中（这个事件是 IRP 的 `UserEvent` 子域）。

② 派遣函数调用 `IoCompleteRequest` 时，`IoCompleteRequest` 内部会设置 IRP 的 `UserEvent` 事件。

③ 操作系统按照原路一直返回到 `ReadFile` 函数，`ReadFile` 函数会等待这个事件。因为该事件已经被设置，所以无须等待。

④ 如果在派遣函数中没有调用 `IoCompleteRequest` 函数，该事件就没有被设置，`ReadFile` 会一直等 IRP 被结束。

如果是用 `ReadFile` 函数进行异步读取时：

① 这时，`ReadFile` 内部不会创建事件，但 `ReadFile` 函数会接收 `overlap` 参数。`overlap` 参数中会提供一个事件，这个事件被用作同步处理。

② `IoCompleteRequest` 内部会设置 `overlap` 提供的事件。

③ 在 `ReadFile` 函数退出前，它不会检测该事件是否被设置，因此可以不等待操作是否真的被完成。

④ 当 IRP 操作被完成后，`overlap` 提供的事件被设置，这个事件会通知应用程序 IRP 请求被完成。

如果是用 `ReadFileEx` 函数进行异步读取时，情况类似，但略有不同：

① `ReadFileEx` 不提供事件，但提供一个回调函数，这个回调函数的地址会作为 IRP 的参数传递给派遣函数。

② `IoCompleteRequest` 会将这个完成函数插入 APC 队列。

③ 应用程序只要进入警惕模式，APC 队列会自动出队列，完成函数会被执行，这相当于通知应用程序操作已经完成。

IRP 的同步处理就是在派遣函数中，将 IRP 处理完毕。这里指的处理完毕就是调用 `IoCompleteRequest` 函数。在第 7 章中介绍的所有例子都属于这种情况。下面重点介绍如何异步完成 IRP。

9.2.2 IRP 的异步完成

IRP 被“异步完成”指的就是不在派遣函数中调用 `IoCompleteRequest` 内核函数。调用 `IoCompleteRequest` 函数意味着 IRP 请求的结束，也标志着本次对设备操作的结束。

IRP 是被异步完成，而发起 IRP 的应用程序会有三种形式发起 IRP 请求，分别是用 `ReadFile` 函数同步读取设备，用 `ReadFile` 异步读取设备，用 `ReadFileEx` 异步读取函数。以下分别列出这三种形式。

- IRP 是由 ReadFile 的同步操作引起：当派遣函数退出时，由于 IoCompleteRequest 没有被调用，IRP 请求没有被结束。ReadFile 会一直等待，直到操作被结束。
- IRP 是由 ReadFile 的异步操作引起：当派遣函数退出时，由于 IoCompleteRequest 没有被调用，IRP 请求没有被结束。但 ReadFile 会立刻返回，返回值为失败，但代表操作没有完成。通过调用 GetLastError 函数，可以得到这时的错误代码是 ERROR_IO_PENDING。这不是真正的操作错误，而意味着 ReadFile 并没有真正完成操作，ReadFile 只是异步的返回。当 IRP 请求被真正的结束，即调用了 IoCompleteRequest，ReadFile 函数提供的 overlap 的事件才会被设置。这个事件可以通知了应用程序 ReadFile 的请求真正的被执行完毕。
- IRP 是由 ReadFileEx 的异步操作引起：和 ReadFile 的异步操作类似，ReadFileEx 会立即返回，但返回值是 FALSE，说明读操作没有成功。这时候如果调用 GetLastError 函数，会发现错误码是 ERROR_IO_PENDING，表明当前操作被“挂起”。当 IRP 被结束后，即调用了 IoCompleteRequest 后，ReadFileEx 提供的回调函数会被插入到 APC 队列中。一旦操作系统进入警惕状态时，线程的 APC 队列会自动出队列，进而 ReadFileEx 提供的回调函数被调用，这相当于操作系统通知应用程序操作真正的被执行完毕。

如果派遣函数不调用 IoCompleteRequest 函数，则需要告诉操作系统此 IRP 处于“挂起”状态。这需要调用内核函数 IoMarkIrpPending。同时，派遣函数应该返回 STATUS_PENDING。下面的代码演示了派遣函数异步处理 IRP。

```
#001 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,  
#002                        IN PIRP pIrp)  
#003 {  
#004     //.....  
#005     IoMarkIrpPending(pIrp);  
#006     //返回 pending 状态  
#007     return STATUS_PENDING;  
#008 }
```

为了演示异步处理 IRP，假设 IRP_MJ_READ 的派遣函数仅仅是返回“挂起”。应用程序关闭设备的时候会产生 IRP_MJ_CLEANUP 类型的 IRP。在 IRP_MJ_CLEANUP 的派遣函数中结束那些“挂起”的 IRP_MJ_READ。

为了能存储有哪些 IRP_MJ_READ IRP 被挂起，这里使用一个队列，也就把每个挂起的 IRP_MJ_READ 的指针都插入队列，最后 IRP_MJ_CLEANUP 的派遣函数将一个个 IRP 出队列，并且调用 IoCompleteRequest 函数将它们结束。

首先，要定义好队列的数据结构，该数据结构应有一个子域来记录 IRP 指针。

```
typedef struct _MY_IRP_ENTRY  
{  
    PIRP pIrp; //记录 IRP 指针  
    LIST_ENTRY ListEntry;  
} MY_IRP_ENTRY, *PMY_IRP_ENTRY;
```

在设备扩展中加入“队列”这个变量，这样驱动程序的所有派遣函数都可以使用该队列。在 DriverEntry 中初始化该队列，并在 DriverUnload 例程中回收队列。在 IRP_MJ_READ 的派遣函数中，将 IRP 插入堆栈，然后返回“挂起”状态。

下面的代码演示了异步处理 IRP 的派遣函数。

```
#001 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#002                        IN PIRP pIrp)
#003 {
#004     KdPrint(("Enter HelloDDKRead\n"));
#005     PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
#006         pDevObj->DeviceExtension;
#007     PMY_IRP_ENTRY pIrp_entry = (PMY_IRP_ENTRY)ExAllocatePool(PagedPool,
sizeof(MY_IRP_ENTRY));
#008     pIrp_entry->pIRP = pIrp;
#009     //插入队列
#010     InsertHeadList(pDevExt->pIRPLinkListHead, &pIrp_entry->ListEntry);
#011     //将 IRP 设置为挂起
#012     IoMarkIrpPending(pIrp);
#013     KdPrint(("Leave HelloDDKRead\n"));
#014     //返回 pending 状态
#015     return STATUS_PENDING;
#016 }
```

此段代码可以在配套光盘中本章的 PendingIRPTest 目录下找到。

在关闭设备的时候，会产生 IRP_MJ_CLEANUP 类型的 IRP。其派遣函数抽取队列中每一个“挂起”的 IRP，并调用 IoCompleteRequest 设置完成。

```
#001 NTSTATUS HelloDDKCleanUp(IN PDEVICE_OBJECT pDevObj,
#002                          IN PIRP pIrp)
#003 {
#004     KdPrint(("Enter HelloDDKCleanUp\n"));
#005     PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
#006         pDevObj->DeviceExtension;
#007
#008     //(1) 将存在队列中的 IRP 逐个出队列，并处理
#009     PMY_IRP_ENTRY my_irp_entry;
#010     while(!IsListEmpty(pDevExt->pIRPLinkListHead))
#011     {
#012         //删除队列元素
#013         PLIST_ENTRY pEntry = RemoveHeadList(pDevExt->pIRPLinkListHead);
#014         //得到元素入口
#015         my_irp_entry = CONTAINING_RECORD(pEntry,
#016             MY_IRP_ENTRY,
#017             ListEntry);
#018         //设置 IRP 完成状态
#019         my_irp_entry->pIRP->IoStatus.Status = STATUS_SUCCESS;
#020         //设置 IRP 操作字节数
#021         my_irp_entry->pIRP->IoStatus.Information = 0;
#022         //结束 IRP 请求
#023         IoCompleteRequest(my_irp_entry->pIRP, IO_NO_INCREMENT);
#024         //回收内存
#025         ExFreePool(my_irp_entry);
#026     }
#027
#028     //(2) 处理 IRP_MJ_CLEANUP 的 IRP
```



```
#029 NTSTATUS status = STATUS_SUCCESS;
#030 // 设置 IRP 完成状态
#031 pIrp->IoStatus.Status = status;
#032 pIrp->IoStatus.Information = 0;
#033 //设置 IRP 操作字节数
#034 IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#035 KdPrint(("Leave HelloDDKCleanUp\n"));
#036 return STATUS_SUCCESS;
#037 }
```

此段代码可以在配套光盘中本章的 PendingIRPTest 目录下找到。

在应用程序中异步操作该设备，先异步读两次，这样会创建两个 IRP_MJ_READ，这两个 IRP 被插入队列。在关闭设备的时候，会导致驱动程序调用 IRP_MJ_CLEANUP 的派遣函数。

```
#001 int main()
#002 {
#003     //打开设备句柄
#004     HANDLE hDevice =
#005         CreateFile("\\\\.\\HelloDDK",
#006             GENERIC_READ | GENERIC_WRITE,
#007             0,
#008             NULL,
#009             OPEN_EXISTING,
#010             FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, //注意此处必
                须加入 FILE_FLAG_OVERLAPPED，表示使用异常方式打开设备
                NULL );
#011
#012     //判断时候成功打开设备句柄
#013     if (hDevice == INVALID_HANDLE_VALUE)
#014     {
#015         printf("Open Device failed!");
#016         return 1;
#017     }
#018     //设置 OVERLAP 结构
#019     OVERLAPPED overlap1={0};
#020     OVERLAPPED overlap2={0};
#021     UCHAR buffer[10];
#022     ULONG ulRead;
#023     //异步读取设备
#024     BOOL bRead = ReadFile(hDevice,buffer,10,&ulRead,&overlap1);
#025     if (!bRead && GetLastError()==ERROR_IO_PENDING)
#026     {
#027         printf("The operation is pending\n");
#028     }
#029     //异步读取设备
#030     bRead = ReadFile(hDevice,buffer,10,&ulRead,&overlap2);
#031     if (!bRead && GetLastError()==ERROR_IO_PENDING)
#032     {
#033         printf("The operation is pending\n");
#034     }
#035
#036     //迫使程序中止 2 秒
#037     Sleep(2000);
#038     //创建 IRP_MJ_CLEANUP IRP
#039     CloseHandle(hDevice);
#040     return 0;
#041 }
```

此段代码可以在配套光盘中本章的 PendingIRPTest 目录下找到。

借助 IRPTrace 工具, 可以很方便地跟踪两个 IRP 从“挂起”到执行完毕的情形。图 9-3 是对 IRP “挂起”时的跟踪。

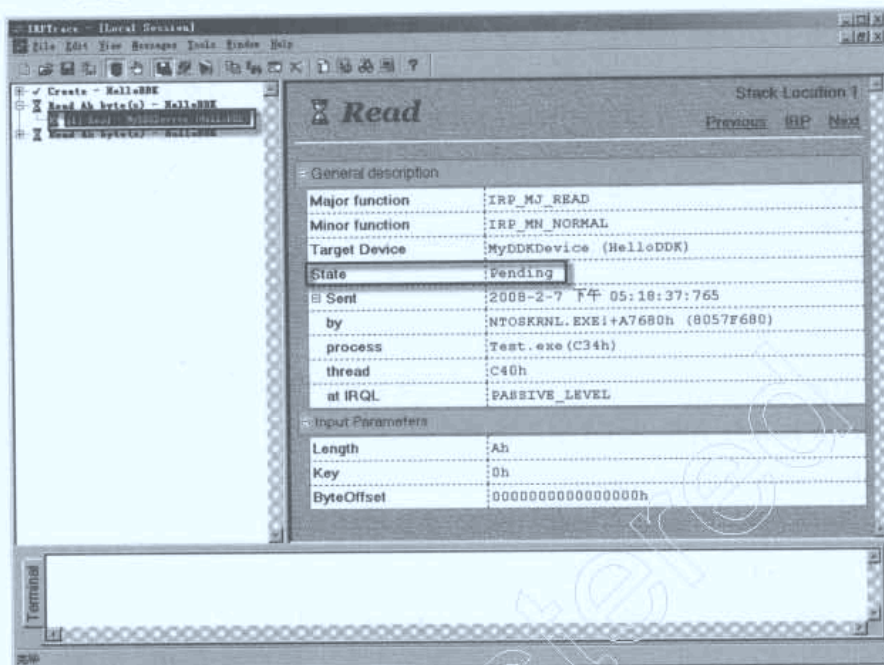


图 9-3 IRPTrace 跟踪 IRP

在应用程序关闭设备时, 被“挂起”的 IRP 从队列出来, 然后被结束, 如图 9-4 所示。

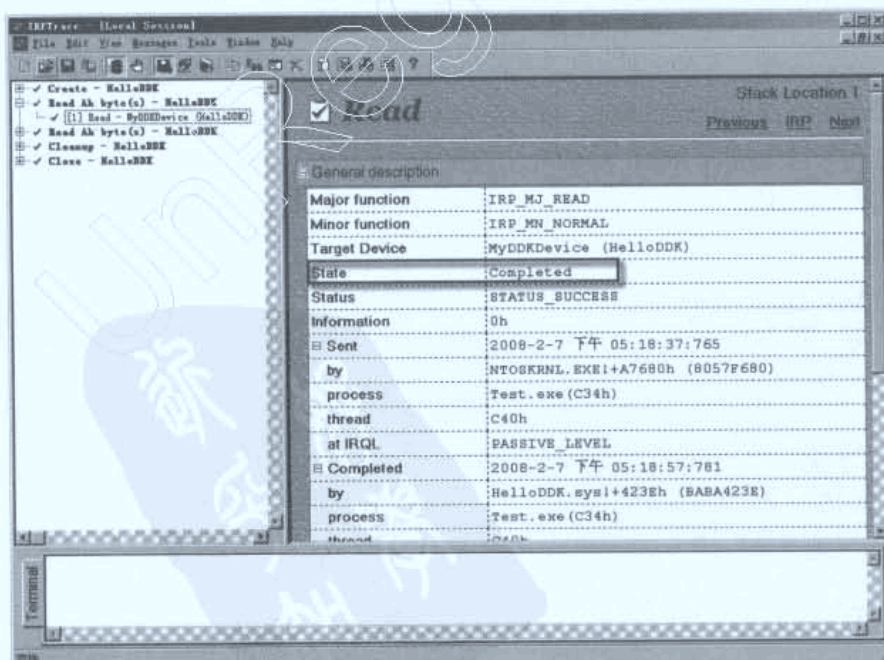


图 9-4 IRPTrace 跟踪 IRP

9.2.3 取消 IRP

9.2.2 节介绍了如何将“挂起”的 IRP 插入队列，并在关闭设备时，将“挂起”的 IRP 结束。还有另外一个办法可以将“挂起”的 IRP 逐个结束，这就是取消 IRP 请求。内核函数 `IoSetCancelRoutine` 可以设置取消 IRP 请求的回调函数，其声明如下：

```
PDRIVER_CANCEL
IoSetCancelRoutine(
    IN PIRP Irp,
    IN PDRIVER_CANCEL CancelRoutine
);
```

- 第一个参数 `Irp`：这个参数是需要取消的 IRP。
- 第二个参数 `CancelRoutine`：这个是取消函数的函数指针。一旦 IRP 请求被取消的时候，操作系统会调用这个取消函数。
- 返回值：标志是否操作成功。

`IoSetCancelRoutine` 可以将一个取消例程与该 IRP 关联，一旦取消 IRP 请求的时候，这个取消例程会被执行。`IoSetCancelRoutine` 函数也可以用来删除取消例程，当输入的 `CancelRoutine` 参数为空指针的时候，则删除原来设置的取消例程。

程序员可以用 `IoCancelIrp` 函数指定取消 IRP 请求。在 `IoCancelIrp` 内部，需要进行同步。DDK 在 `IoCancelIrp` 内部使用一个叫做 `cancel` 的自旋锁用来进行同步。

`IoCancelIrp` 在内部会首先获得该自旋锁，`IoCancelIrp` 会调用取消回调例程，因此，释放该自旋锁的任务就留给了取消回调例程。获得取消自旋的函数是 `IoAcquireCancelSpinLock` 函数，而释放取消自旋锁的函数是 `IoReleaseCancelSpinLock` 函数。

在应用程序中，可以调用 `CancelIo Win32 API` 函数取消 IRP 请求。在 `CancellIo` 的内部会枚举所有没有被完成的 IRP，然后依次调用 `IoCancelIrp`。另外，如果应用程序没有调用 `CancellIo` 函数，应用程序在关闭设备时同样会自动调用 `CancellIo`。下面的代码演示了如何编写取消例程。

```
#001 VOID
#002 CancelReadIrp(
#003     IN PDEVICE_OBJECT DeviceObject,
#004     IN PIRP Irp
#005 )
#006 {
#007     KdPrint(("Enter CancelReadIrp\n"));
#008     PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
#009         DeviceObject->DeviceExtension;
#010     //设置完成状态为 STATUS_CANCELLED
#011     Irp->IoStatus.Status = STATUS_CANCELLED;
#012     //设置 IRP 操作字节数
#013     Irp->IoStatus.Information = 0;
#014     //结束 IRP 请求
#015     IoCompleteRequest( Irp, IO_NO_INCREMENT );
#016     //释放 Cancel 自旋锁
```

```

#017 IoReleaseCancelSpinLock(Irp->CancelIrql);
#018 KdPrint(("Leave CancelReadIRP\n"));
#019 }
#020
#021 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#022                      IN PIRP pIrp)
#023 {
#024     KdPrint(("Enter HelloDDKRead\n"));
#025     //获得的设备扩展
#026     PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
#027         pDevObj->DeviceExtension;
#028     //设置完成例程
#029     IoSetCancelRoutine(pIrp, CancelReadIRP);
#030     //将 IRP 设置为挂起
#031     //挂起 IRP
#032     IoMarkIrpPending(pIrp);
#033     KdPrint(("Leave HelloDDKRead\n"));
#034     //返回 pending 状态
#035     return STATUS_PENDING;
#036 }

```

此段代码可以在配套光盘中本章的 CancelIRPTest 目录下找到。

下面我们在应用程序中调用 CancelIO win32API。这样可以导致 IRP 的取消例程被调用。

```

#001 int main()
#002 {
#003     //打开设备
#004     HANDLE hDevice =
#005         CreateFile("\\\\.\\HelloDDK",
#006                 GENERIC_READ | GENERIC_WRITE,
#007                 0,
#008                 NULL,
#009                 OPEN_EXISTING,
#010                 FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, //注意此处必
//须加入 FILE_FLAG_OVERLAPPED, 表示使用异常方式打开设备
#011                 NULL );
#012     //判断设备是否成功打开
#013     if (hDevice == INVALID_HANDLE_VALUE)
#014     {
#015         printf("Open Device failed!");
#016         return 1;
#017     }
#018     //设置 OVERLAP 结构
#019     OVERLAPPED overlapp1={0};
#020     OVERLAPPED overlapp2={0};
#021     UCHAR buffer[10];
#022     ULONG ulRead;
#023     //异步读取设备
#024     BOOL bRead = ReadFile(hDevice,buffer,10,&ulRead,&overlapp1);
#025     if (!bRead && GetLastError()==ERROR_IO_PENDING)
#026     {
#027         printf("The operation is pending\n");
#028     }
#029     //异步读取设备
#030     bRead = ReadFile(hDevice,buffer,10,&ulRead,&overlapp2);

```



```

#031     if (!bRead && GetLastError()==ERROR_IO_PENDING)
#032     {
#033         printf("The operation is pending\n");
#034     }
#035
#036     //迫使程序中止2秒
#037     Sleep(2000);
#038     //显式调用 CancelIo, 其实在关闭设备时会自动运行 CancelIo
#039     CancelIo(hDevice);
#040     //创建 IRP_MJ_CLEANUP IRP
#041     //关闭设备句柄
#042     CloseHandle(hDevice);
#043     return 0;
#044 }

```

此段代码可以在配套光盘中本章的 CancellRPTest 目录下找到。

在设置取消例程中要注意同步问题是, 当退出取消例程时, 一定要释放 cancel 自旋锁, 否则会导致系统崩溃。另外, cancel 自旋锁是全局自旋锁, 所有驱动程序都会使用这个自旋锁, 因此, 占用自旋锁时间不宜过长。

运行结果和 9.2.2 节类似, 在关闭设备前, 所有“挂起”的 IRP 会被执行完。

9.3 StartIO 例程

StartIO 例程能够保证各个并行的 IRP 顺序执行, 即串行化。

9.3.1 并行执行与串行执行

在很多情况时, 对设备的操作必须是串行执行, 而不能并行执行。例如, 对串口的操作, 假如有 N 个线程同时操作串口设备时, 必须将这些操作排队, 然后一一进行处理。如果不做串行化处理, 当一个操作没有完毕时, 新的操作又开始, 这必然会导致操作的混乱。

因此, 驱动程序有必要将并行的请求变成串行的请求, 这需要用队列, 如图 9-5 所示。假如有 8 个读请求先后到来, 其中每个方块代表一个请求。每个方块的左边代表发出请求的时间, 宽度代表处理这个操作所需要的时间。



图 9-5 IRP 并行运行

可以想象，如果不做处理，派遣函数的执行会交织在一起，也就是并行运行。如果想依次处理每个 IRP，必须采用队列将处理串行化。采用的原则是“先来先服务”。如图 9-6 所示，这是排队以后的请求。



图 9-6 IRP 运行的串行化

当一个新的 IRP 请求来临时，首先检查设备是否处于“忙”状态。设备在初始化的时候为“空闲”状态。当设备处在“空闲”的时候，可以处理一个 IRP 请求，并改变当前设备状态为“忙”状态。如果设备处于“忙”状态，则将新来的 IRP 插入队列，并立刻返回，IRP 留在以后处理。

当设备状态由“忙”转入“空闲”状态时，则从队列取出一个 IRP 进行处理，并重新将状态变为“忙”。这样，周而复始地将所有 IRP 的请求串行都处理了。

在 9.2.2 节的例子用到的就是上述原理，但是队列处理非常粗糙，没有考虑 IRP 取消函数、超时、同步插入队列等问题。程序员可以自己定义一个队列来实现“串行化”，但是 DDK 简化了程序员的工作，为程序员提供了一个内部队列，并将 IRP 用 StartIO 例程串行处理。

9.3.2 StartIO 例程

操作系统为程序员提供了一个 IRP 队列来实现串行，这个队列用 KDEVICE_QUEUE 数据结构表示。

```
typedef struct _KDEVICE_QUEUE {
    CSHORT Type;
    CSHORT Size;
    LIST_ENTRY DeviceListHead;
    KSPIN_LOCK Lock;
    BOOLEAN Busy;
} KDEVICE_QUEUE, *PKDEVICE_QUEUE, *RESTRICTED_POINTER PRKDEVICE_QUEUE;
```

这个队列的队列头保存在设备对象的 DeviceObject->DeviceQueue 子域中。插入和删除队列中的元素都是操作系统负责的。在使用这个队列的时候，需要向系统提供一个叫做 StartIo 的例程，并将这个例程的函数名传送给系统，代码如下：

```
#001 extern "C" NTSTATUS DriverEntry (
#002     IN PDRIVER_OBJECT pDriverObject,
#003     IN PUNICODE_STRING pRegistryPath )
#004 {
#005     ...
#006     //设置 StartIO 例程
#007     pDriverObject->DriverStartIo = HelloDDKStartIO;
#008     ...
#009 }
```


Windows 驱动开发技术详解

这个 StartIO 例程运行在 DISPATCH_LEVEL 级别, 因此这个例程是会被线程所打断的。StartIO 例程的参数类似于派遣函数, 只不过没有返回值。注意 StartIO 是执行在 DISPATCH_LEVEL 级别上, 因此在声明时要加上 #pragma LOCKEDCODE 修饰符。

```
#001 #pragma LOCKEDCODE
#002 VOID
#003 HelloDDKStartIO(
#004     IN PDEVICE_OBJECT DeviceObject,
#005     IN PIRP Irp
#006 )
#007 {
#008     ...
#009 }
```

派遣函数如果想把 IRP 串行化, 只需要加入 IoStartPacket 函数, 就可以将 IRP 插入队列了。并且 IoStartPacket 函数还可以让程序员指定其取消例程。IoStartPacket 首先判断当前设备是“忙”还是“空闲”。如果设备“空闲”, 则提升当前 IRQL 到 DISPATCH_LEVEL 级别, 并进入 StartIO 例程“串行”处理该 IRP 请求。如果设备“忙”, 则将 IRP 插入队列后返回。以下是 IoStartPacket 的伪代码。注意这只是示例代码, Windows 的源码要比这个复杂的多。

```
#001 VOID IoStartPacket(PDEVICE_OBJECT device, PIRP Irp, PULONG key, PDIRECTOR_CANCEL
cancel)
#002 {
#003     KIRQL oldirql;
#004     //获得自旋锁
#005     IoAcquireCancelSpinLock(&oldirql);
#006     //设置取消例程
#007     IoSetCancelRoutine(Irp, cancel);
#008     //设置 IRP
#009     device->CurrentIrp = Irp;
#010     //释放自旋锁
#011     IoReleaseCancelSpinLock(oldirql);
#012     //调用 StartIO 例程
#013     device->DriverObject->DriverStartIo(device, Irp);
#014 }
```

在 StartIO 例程结束前, 应该调用 IoStartNextPacket 函数, 其作用是从队列中抽取下一个 IRP, 并将这个 IRP 作为参数调用 StartIO 例程。以下是 IoStartNextPacket 的示例代码。

```
#001 VOID IoStartNextPacket(PDEVICE_OBJECT device, BOOLEAN cancel)
#002 {
#003     KIRQL oldirql;
#004     if (cancel)
#005         //获取自旋锁
#006     IoAcquireCancelSpinLock(&oldirql);
#007     //删除设备队列
#008     PKDEVICE_QUEUE_ENTRY p = KeRemoveDeviceQueue(&device->DeviceQueue);
#009     //获得 IRP 指针
#010     PIRP Irp = CONTAINING_RECORD(p, IRP, Tail.Overlay.DeviceQueueEntry);
#011     //设置 IRP
#012     device->CurrentIrp = Irp;
```

```

#013     if (cancel)
#014         //释放自旋锁
#015     IoReleaseCancelSpinLock(olDirql);
#016         //调用 StartIO 函数
#017     device->DriverObject->DriverStartIo(device, Irp);
#018 }

```

从上述代码可以看出，在调用 StartIO 之前，操作系统会将设备的 device->CurrentIrp 设置为当前 IRP，这意味这个 IRP 正准备由 StartIO 例程处理。

当处理 StartIO 例程时，最复杂的莫过于对取消例程的处理，正确使用 cancel 自旋锁是关键。在 StartIO 例程的开始处，应首先获得 cancel 自旋锁。然后判断当前的 IRP 是否等于 DeviceObject->CurrentIrp。如果是以上的情况，说明这个 IRP 正在或者即将被 StartIO 处理。StartIO 例程应立刻释放自旋锁，什么也不做立刻退出。

9.3.3 示例

在使用 StartIO 例程时，需要 IRP 的派遣例函数返回挂起状态，然后调用 IoStartPacket 内核函数。下面的代码演示了如何编写这样的派遣函数。

```

#001 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#002                        IN PIRP pIrp)
#003 {
#004     KdPrint(("Enter HelloDDKRead\n"));
#005     //获得设备扩展
#006     PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
#007         pDevObj->DeviceExtension;
#008
#009     //将 IRP 设置为挂起
#010     IoMarkIrpPending(pIrp);
#011
#012     //将 IRP 插入系统的队列
#013     IoStartPacket(pDevObj, pIrp, 0, OnCancelIRP);
#014
#015     KdPrint(("Leave HelloDDKRead\n"));
#016     //返回 pending 状态
#017     return STATUS_PENDING;
#018 }

```

此段代码可以在配套光盘中本章的 StartIOTest 目录下找到。

在派遣函数中调用 IoStartPacket 内核函数指定取消例程。下面的代码演示了如何编写取消例程。

```

#001 VOID
#002 OnCancelIRP(
#003     IN PDEVICE_OBJECT DeviceObject,
#004     IN PIRP Irp
#005 )
#006 {
#007     KdPrint(("Enter CancelReadIRP\n"));
#008     if (Irp==DeviceObject->CurrentIrp)
#009     {

```



```

#010      //当前 IRP 正由 StartIo 处理
#011
#012
#013      KIRQL oldirql = Irp->CancelIrql;
#014      //释放 Cancel 自旋锁
#015      IoReleaseCancelSpinLock(Irp->CancelIrql);
#016      //继续下一个 IRP
#017      IoStartNextPacket(DeviceObject, TRUE);
#018      //降低 IRQL
#019      KeLowerIrql(oldirql);
#020  }else
#021  {
#022      //从设备队列中将该 IRP 抽取出来
#023      KeRemoveEntryDeviceQueue(&DeviceObject->DeviceQueue, &Irp->Tail.Overlay.DeviceQueueEntry);
#024      //释放 Cancel 自旋锁
#025      IoReleaseCancelSpinLock(Irp->CancelIrql);
#026  }
#027
#028      //设置完成状态为 STATUS_CANCELLED
#029      Irp->IoStatus.Status = STATUS_CANCELLED;
#030      //设置 IRP 操作字节数
#031      Irp->IoStatus.Information = 0; // bytes xfered
#032      //结束 IRP 请求
#033      IoCompleteRequest(Irp, IO_NO_INCREMENT);
#034      KdPrint(("Leave CancelReadIRP\n"));
#035  }

```

此段代码可以在配套光盘中本章的 StartIOTest 目录下找到。

然后编写 StartIO 例程，注意 StartIO 运行在 DISPATCH_LEVEL 级别，因此不能使用分页内存，否则会引起页故障，从而导致系统崩溃。下面的代码演示了如何编写 StartIO 例程。

```

#001 #pragma LOCKEDCODE
#002 VOID
#003 HelloDDKStartIO(
#004     IN PDEVICE_OBJECT DeviceObject,
#005     IN PIRP Irp
#006 )
#007 {
#008     KIRQL oldirql;
#009     KdPrint(("Enter HelloDDKStartIO\n"));
#010
#011     //获取 cancel 自旋锁
#012     IoAcquireCancelSpinLock(&oldirql);
#013     if (Irp!=DeviceObject->CurrentIrp||Irp->Cancel)
#014     {
#015         //如果当前有正在处理的 IRP，则简单的入队列，并直接返回
#016         //入队列的工作由系统完成，在 StartIO 中不用负责
#017         IoReleaseCancelSpinLock(oldirql);
#018         KdPrint(("Leave HelloDDKStartIO\n"));
#019         return;
#020     }else
#021     {

```

```

#022      //由于正在处理该 IRP，所以不允许调用取消例程
#023      //因此将此 IRP 的取消例程设置为 NULL
#024      IoSetCancelRoutine(Irp, NULL);
#025      //释放自旋锁
#026      IoReleaseCancelSpinLock(olddirql);
#027  }
#028  KEVENT event;
#029  KeInitializeEvent(&event, NotificationEvent, FALSE);
#030  //等 3 秒
#031  LARGE_INTEGER timeout;
#032  timeout.QuadPart = -3*1000*1000*10;
#033  //定义一个 3 秒的延时，主要是为了模拟该 IRP 操作需要大概 3 秒左右时间
#034  KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, &timeout);
#035  //设置 IRP 状态
#036  Irp->IoStatus.Status = STATUS_SUCCESS;
#037  //设置 IRP 操作字节数
#038  Irp->IoStatus.Information = 0;
#039  //结束 IRP 请求
#040  IoCompleteRequest(Irp, IO_NO_INCREMENT);
#041  //在队列中读取一个 IRP，并进行 StartIo
#042  IoStartNextPacket(DeviceObject, TRUE);
#043  KdPrint(("Leave HelloDDKStartIO\n"));
#044  }

```

此段代码可以在配套光盘中本章的 StartIOTest 目录下找到。

最后编写应用程序的代码。这段代码首先打开设备，然后创建两个线程，每个线程都是异步读取。为了模拟真实的情况，笔者在 StartIO 中停顿了 3 秒钟。应用程序并发的发起两个线程同时读取设备，从而真实的模拟了并发发起 IRP 请求。

```

#001  UINT WINAPI Thread(LPVOID context)
#002  {
#003      printf("Enter Thread\n");
#004      //等待 3s
#005      OVERLAPPED overlap={0};
#006      //创建同步事件
#007      overlap.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
#008      UCHAR buffer[10];
#009      ULONG ulRead;
#010      //读取设备
#011      BOOL bRead = ReadFile(*(PHANDLE)context, buffer, 10, &ulRead, &overlap);
#012      //可以试验取消例程
#013      //CancelIo(*(PHANDLE)context);
#014      //等待事件
#015      WaitForSingleObject(overlap.hEvent, INFINITE);
#016      return 0;
#017  }
#018  int main()
#019  {
#020      //打开设备句柄
#021      HANDLE hDevice =
#022          CreateFile("\\\\.\\HelloDDK",
#023                  GENERIC_READ | GENERIC_WRITE,
#024                  FILE_SHARE_READ,
#025                  NULL,

```



```

#026         OPEN_EXISTING,
#027         FILE_ATTRIBUTE_NORMAL|FILE_FLAG_OVERLAPPED, //注意此处必
            //须加入 FILE_FLAG_OVERLAPPED, 表示使用异常方式打开设备
#028         NULL );
#029     //判断是否成功打开设备
#030     if (hDevice == INVALID_HANDLE_VALUE)
#031     {
#032         printf("Open Device failed!");
#033         return 1;
#034     }
#035     HANDLE hThread[2];
#036     //开启两个线程
#037     hThread[0] = (HANDLE) _beginthreadex (NULL, 0, Thread, &hDevice, 0, NULL);
#038     hThread[1] = (HANDLE) _beginthreadex (NULL, 0, Thread, &hDevice, 0, NULL);
#039     //主线程等待两个子线程结束
#040     WaitForMultipleObjects(2, hThread, TRUE, INFINITE);
#041
#042     //创建 IRP_MJ_CLEANUP IRP
#043     CloseHandle(hDevice);
#044     return 0;
#045 }

```

此段代码可以在配套光盘中本章的 StartIOTest 目录下找到。

9.4 自定义的 StartIO

系统定义的 StartIO 例程只能使用一个队列，这个队列会将所有的 IRP 进行处理化。例如，读、写操作都会混在一起进行串行化处理。然而，在有些情况下，需要将读、写分别进行串行化处理。这时候就需要自定义 StartIO 例程。

9.4.1 多个串行化队列

读者可能会发现，StartIO 虽然可以很方便地将 IRP 串行化，但是存在一个问题，这就是读、写操作都被一起串行化。有时候需要将读、写分开串行化。此时希望有两个队列，一个队列串行 IRP_MJ_READ 类型 IRP，另外一个线程负责串行 IRP_MJ_WRITE 类型 IRP。

但是很遗憾，DDK 提供的 StartIO 例程内部只有一个队列，无法实现上述的要求。但是 DDK 提供了一种更灵活的方式——自定义 StartIO。自定义 StartIO 类似于前面介绍的 StartIO 例程，不同的是程序员需要自己维护 IRP 队列。程序员可以灵活的维护多个队列，分别应用于不同类型的 IRP。

DDK 提供 KDEVICE_QUEUE 数据结构存储队列。

```

typedef struct _KDEVICE_QUEUE {
    USHORT Type;
    USHORT Size;
    LIST_ENTRY DeviceListHead;
    KSPIN_LOCK Lock;
    BOOLEAN Busy;
} KDEVICE_QUEUE, *PKDEVICE_QUEUE, *RESTRICTED_POINTER PKDEVICE_QUEUE;

```

队列中每个元素用 `KDEVICE_QUEUE_ENTRY` 数据结构表示。

```
typedef struct _KDEVICE_QUEUE_ENTRY {
    LIST_ENTRY DeviceListEntry;
    ULONG SortKey;
    BOOLEAN Inserted;
} KDEVICE_QUEUE_ENTRY, *PKDEVICE_QUEUE_ENTRY, *RESTRICTED_POINTER PKDEVICE_QUEUE_ENTRY;
```

在 `StartIO` 的内部也用到这种队列。但当使用 `StartIO` 时，程序员不用关心队列的“入队”和“出队”操作，这些都由操作系统负责。而当使用自定义 `StartIO` 时，程序员需要自己负责“入队”和“出队”操作。在使用队列前，应该初始化队列，用 `KeInitializeDeviceQueue` 函数初始化队列。队列应该存储在设备扩展中，在初始化设备的时候一起初始化该队列。

插入队列的内核函数是 `KeInsertDeviceQueue`，其声明如下。

```
BOOLEAN
KeInsertDeviceQueue(
    IN PKDEVICE_QUEUE DeviceQueue,
    IN PKDEVICE_QUEUE_ENTRY DeviceQueueEntry
);
```

- 第一个参数 `DeviceQueue`：这个参数是需要被插入的队列。
- 第二个参数 `DeviceQueueEntry`：这个参数是要被插入的元素。
- 返回值：返回值为 `BOOL` 值，如果当前设备不忙，则可以直接处理该 `IRP`，因此这时候不需要插入队列，返回 `FALSE`。如果设备正在处理，这时候需要将 `IRP` 插入队列，这时候返回 `TRUE`。

从队列中删除元素使用 `KeRemoveDeviceQueue` 函数，其声明如下。

```
PKDEVICE_QUEUE_ENTRY
KeRemoveDeviceQueue(
    IN PKDEVICE_QUEUE DeviceQueue
);
```

- 参数 `DeviceQueue`：指定从哪个队列中取出元素。
- 返回值：返回从队列中取出的元素指针。

9.4.2 示例

本节演示了如何编写自定义 `StartIO` 例程。

首先应该在设备扩展中加入 `KDEVICE_QUEUE` 数据结构存储队列，并且在 `DriverEntry` 中初始化该队列。如果程序员需要对读和写操作分别串行化，可以在设备扩展里创建两个队列，分别对应读和写。本例只自定义一个队列，读者可以试着为自己的驱动加入更多的队列。

然后是编写派遣例程，在派遣例程中首先用 `IoMarkIrpPending` 函数将该 `IRP` 设为挂起，然后准备将 `IRP` 进入队列。在进入队列之前，要先将当前的 `IRQL` 提升至 `DISPATCH_LEVEL` 级别。

Windows 驱动开发技术详解

插入队列使用 `KeInsertDeviceQueue` 函数，该函数的返回值指示是否需要立即执行。当该函数返回 `FALSE` 的时候，表明 IRP 没有插入到队列，而是需要被立刻结束。这时候需要调用自定义的 `StartIO` 例程。

```
#001 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#002                          IN PIRP pIrp)
#003 {
#004     KdPrint(("Enter HelloDDKRead\n"));
#005     //获得设备扩展
#006     PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
#007         pDevObj->DeviceExtension;
#008     //将 IRP 设置为挂起
#009     IoMarkIrpPending(pIrp);
#010     //设置取消例程
#011     IoSetCancelRoutine(pIrp, OnCancelIrp);
#012     KIRQL oldIrql;
#013     //提升 IRP 至 DISPATCH_LEVEL
#014     KeRaiseIrql(DISPATCH_LEVEL, &oldIrql);
#015     KdPrint(("HelloDDKRead irp :%x\n", pIrp));
#016     KdPrint(("DeviceQueueEntry:%x\n", &pIrp->Tail.Overlay.DeviceQueue
Entry));
#017     //插入设备队列
#018     if (!KeInsertDeviceQueue(&pDevExt->device_queue, &pIrp->Tail.Overlay.
DeviceQueueEntry))
#019         MyStartIo(pDevObj, pIrp);
#020     //将 IRP 降至原来的 IRQL 级别
#021     KeLowerIrql(oldIrql);
#022     KdPrint(("Leave HelloDDKRead\n"));
#023     //返回 pending 状态
#024     return STATUS_PENDING;
#025 }
```

接下来编写自定义 `StartIO` 例程。自定义 `StartIO` 的任务是首先处理传进 `StartIO` 中的 IRP，然后在枚举队列中的 IRP，然后依次出队列，再对其进行处理。

```
#001 #pragma LOCKEDCODE
#002 //让函数运行在非分页内存
#003 VOID
#004 MyStartIo(
#005     IN PDEVICE_OBJECT DeviceObject,
#006     IN PIRP pFistIrp
#007 )
#008 {
#009     KdPrint(("Enter MyStartIo\n"));
#010     //获得设备扩展
#011     PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
#012         DeviceObject->DeviceExtension;
#013     PKDEVICE_QUEUE_ENTRY device_entry;
#014     PIRP Irp = pFistIrp;
#015     do
#016     {
#017         KEVENT event;
#018         //初始化同步事件
#019         KeInitializeEvent(&event, NotificationEvent, FALSE);
#020         //等 3 秒
```

```

#021     LARGE_INTEGER timeout;
#022     timeout.QuadPart = -3*1000*1000*10;
#023     //定义一个3秒的延时,主要是为了模拟该IRP操作需要大概3秒左右时间
#024     KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, &timeout);
#025     KdPrint(("Complete a irp:%x\n", Irp));
#026     //设置IRP完成状态
#027     Irp->IoStatus.Status = STATUS_SUCCESS;
#028     //设置IRP操作字节数
#029     Irp->IoStatus.Information = 0;
#030     //结束IRP请求
#031     IoCompleteRequest(Irp, IO_NO_INCREMENT);
#032     //删除设备队列
#033     device_entry=KeRemoveDeviceQueue(&pDevExt->device_queue);
#034     KdPrint(("device_entry:%x\n", device_entry));
#035     if (device_entry==NULL)
#036     {
#037         break;
#038     }
#039     Irp = CONTAINING_RECORD(device_entry, IRP, Tail.Overlay.DeviceQueue
Entry);
#040     }while(1);
#041     KdPrint(("Leave MyStartIo\n"));
#042 }

```

此段代码可以在配套光盘中本章的 SpecialStartIOTest 目录下找到。

9.5 中断服务例程

中断服务例程是设备触发中断后进入的例程。当进入中断服务例程后, IRQL 会提升到设备对应的 IRQL 级别。本节针对中断服务例程的概念和编写方法做简单介绍。

9.5.1 中断操作的必要性

在介绍中断服务例程之前,先讨论一下中断操作的必要性,对中断原理熟悉的读者可以跳过本节。

在早期的 PC 中,很多设备都是“轮询”设备。CPU 发出指令去操作设备时,操作一般不能迅速完成。CPU 需要不停地读取设备的状态,从而判断操作是否结束。

后来,出现了“中断”设备。CPU 发出一个指令请求操作设备后, CPU 并不急于知道操作是否已经完毕。这时候 CPU 可以去做别的事情。当设备操作完成后,设备会向 CPU 发出中断请求。中断请求相当于通知设备的操作已经完毕。“中断”设备比“轮询”设备的效率高,不会浪费太多的 CPU 时间。

另外,中断请求可以嵌套。当 CPU 处理一个低优先级的中断时,它可以被更高优先级的中断打断,转而执行优先级高的中断处理程序。当执行完高优先级的中断处理服务程序后,刚才被打断的低优先级的中断恢复执行。

9.5.2 中断优先级

传统的 PC 用 2 片中断控制器(8259A)芯片级联, 可以连接 16 个中断信号源。每个中断信号分配一个中断号, 依次从 0 至 15, 多个设备可以共享一个中断号。新的 PC 中使用新的中断控制器, 并将中断信号扩展到了 24 个, 多个设备可以共享同一个中断向量, 如图 9-7 所示。

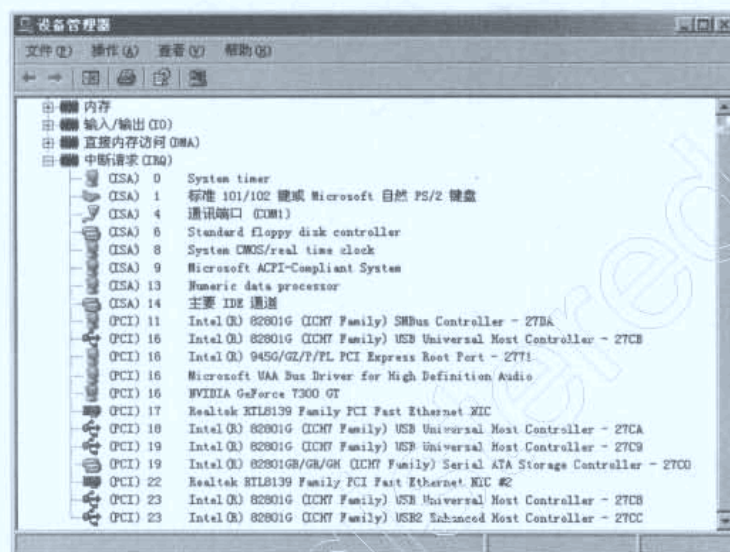


图 9-7 设备管理器中查看中断号

Windows 将中断的概念进行了扩展, 扩展为 32 个中断级别 (IRQL)。其中 0~2 级别, 即 PASSIVE_LEVEL 到 DISPATCH_LEVEL 级别为软件中断。从 3~31 级别为硬件中断。优先级从 0 到 31, 级别逐次升高。

一般线程运行在 PASSIVE_LEVEL 级别, 而负责调度线程的内核代码运行在 DISPATCH_LEVEL 级别。如果线程希望不被切换到别的线程, 可以将线程从 PASSIVE_LEVEL 提升到 DISPATCH_LEVEL 级别。硬件设备发出的中断信号, 都是硬件中断。硬件中断的优先级都高于软件优先级, 也就是说运行的线程会随时被硬件中断所打断。DDK 把硬件中断称为 DIRQL。

9.5.3 中断服务例程 (ISR)

当硬件设备的中断信号发生后, IRQL 会提升至相应的 DIRQL 级别, 操作系统会调用相应的中断服务例程 (ISR)。在驱动程序中使用 ISR, 首先要获得中断对象, 该中断对象是一个名为 INTERRUPT 的数据结构。对于 NT 式 DDK 驱动和 WDM 式驱动, 得到中断对象使用不同的方法, 这将在第 16 章进行介绍。

DDK 提供内核函数 `IoConnectInterrupt` 将中断对象和 ISR 联系起来，这样当中断信号来临时就会进入 ISR 处理。

ISR 运行在 `DIRQL` 级别，要高于普通线程的优先级。自选锁只能对 `DISPATCH_LEVEL` 以下的程序进行同步，所以这时自选锁无法满足同步的需要。派遣函数、`StartIO` 例程随时会被中断程序所打断。为了不让 ISR 打断，只需将 `IRQL` 提升至相应的 `DIRQL` 即可。DDK 提供了与 ISR 函数同步的内核函数 `KeSynchronizeExecution`，其声明如下。

```
BOOLEAN
KeSynchronizeExecution(
    IN PKINTERRUPT Interrupt,
    IN PKSYNCHRONIZE_ROUTINE SynchronizeRoutine,
    IN PVOID SynchronizeContext
);
```

- 第一个参数 `Interrupt`：这个参数是中断对象指针，ISR 和这个对象关联着。
- 第二个参数 `SynchronizeRoutine`：将当前 `IRQL` 提升至中断对象相应的 `DIRQL`，并执行 `SynchronizeRoutine` 函数。
- 第三个参数 `SynchronizeContext`：为 `SynchronizeRoutine` 提供的参数。

当运行到 `KeSynchronizeExecution` 的时候，ISR 不会打断 `KeSynchronizeExecution` 提供的同步函数，这需要将同步代码放入 `KeSynchronizeExecution` 提供的同步函数中。

9.6 DPC 例程

DPC 例程一般和中断服务例程配合使用。中断服务例程处于很高的 `IRQL`，会打断正常运行的线程。而 DPC 例程运行于相对较低的 `DISPATCH_LEVEL` 级别。因此，一般将不需要紧急处理的代码放在 DPC 例程中，而将需要紧急处理的代码放在中断服务例程中。

9.6.1 延迟过程调用例程（DPC）

延迟过程调用（DPC）广泛应用于驱动程序开发中，它运行在 `DISPATCH_LEVEL` 的 `IRQL` 级别。因此除了 ISR，其他例程是不会将其打断的。

一般说来，ISR 运行时间不宜过长。如果 ISR 运行时间过长，中断级别低的程序就无法得到响应。例如，鼠标、键盘的中断就不能及时响应。为了能响应更多的中断，应该让 ISR 代码尽量少。DDK 建议将一些不重要的响应代码从 ISR 移到 DPC 例程中。

ISR 会将 DPC 插入一个内部队列，这样当 `IRQL` 从高的 `IRQL` 恢复到 `DISPATCH_LEVEL` 时，DPC 会依次从队列中弹出，并执行相应的 DPC 例程。总结一下，ISR 中的代码应该尽量少，而将不太重要的代码放入 DPC 例程中。

9.6.2 DpcForISR

使用 DPC 例程，首先要初始化 DPC 对象，该动作使用内核函数 `KeInitializeDpc`，其

声明如下。

```
VOID  
KeInitializeDpc(  
    IN PRKDPC Dpc,  
    IN PKDEFERRED_ROUTINE DeferredRoutine,  
    IN PVOID DeferredContext  
);
```

- 第一个参数 Dpc: 需要初始化的 DPC 对象指针。
- 第二个参数 DeferredRoutine: 与 DPC 关联的 DPC 例程。
- 第三个参数 DeferredContext: 传入 DPC 例程的参数。

一般在驱动程序的 DriverEntry 或者 AddDevice 例程中初始化 DPC 例程。

```
#001 KeInitializeDpc( &pdx->DPC,  
#002 DpcForISR,  
#003 (PVOID) context );
```

ISR 中执行简短地处理, 然后将不重要的代码放到 DPC 例程中。

```
#001 BOOL OnInterrupt(PKINTERRUPT interrupt,PVOID context)  
#002 {  
#003     .....  
#004     IoRequestDpc(device,device->CurrentIrp,NULL);  
#005     .....  
#006 }
```

关于中断例程和 DPC 例程的详细介绍, 请参考第 16 章。

9.7 小结

对设备的操作主要分为同步操作和异步操作。所有对设备的操作最终将转化为 IRP 请求, 因此同步操作和异步操作问题就转化为对 IRP 的同步处理和异步处理。本章详细地介绍了 IRP 的同步处理方法和异步处理方法。另外, 本章还介绍了 StartIO 例程、中断服务例程、DPC 服务例程。

第 10 章 定时器

在驱动程序编程中，经常会用到定时器。定时器的应用很广泛，例如对于轮询设备，每间隔一定时间，操作系统就会向设备发送一个请求，这时就需要用到定时器。本章首先介绍如何在驱动程序中使用定时器。另外，本章还总结了驱动程序中四种等待的方法。最后，本章还对 IRP 的超时处理进行了讨论。

10.1 定时器实现方式一

在驱动程序中，一般有两种方法使用定时器，一种方法是使用 I/O 定时器例程，另一种方法是使用 DPC 例程。本节主要介绍如何使用 I/O 定时器例程，下一节介绍如何使用 DPC 例程实现定时器。

10.1.1 I/O 定时器

I/O 定时器是 DDK 提供的一种定时器。使用这种定时器时，每间隔 1s 钟系统会调用一次 I/O 定时器例程。程序员可以对这种定时器稍加改进，在定时器例程中记录一个计数，初始值设置为 N。每次进入定时器例程的时候计数减一，当计数变为 0 时，执行某项操作，并将计数还原为 N。这样，间隔 1s 的定时器就变化成间隔 Ns 的定时器。

I/O 定时器可以为间隔 Ns 做定时，但如果要实现 ms 级别间隔、 μ s 级别的间隔，程序员需要使用下一节介绍的 DPC 定时器。在使用 I/O 定时器前需要先进行初始化。初始化 I/O 定时器使用内核函数 IoInitializeTimer，其声明如下：

```
NTSTATUS
IoInitializeTimer(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIO_TIMER_ROUTINE TimerRoutine,
    IN PVOID Context
);
```


- 第一个参数 DeviceObject: 这个参数是 I/O 定时器关联的设备对象指针。
- 第二个参数 TimerRoutine: 这个参数是 I/O 定时器关联的定时器例程。
- 第三个参数 Context: 这个参数是传入定时器例程的参数。
- 返回值: 返回初始化的状态。

在初始化 I/O 定时器后, 可以开启和停止 I/O 定时器。开启定时器后, 每隔 1s 系统调用一次定时器例程。在停止定时器后, 系统就不会进入定时器例程。开启定时器的内核函数是 IoStartTimer, 停止 I/O 定时器的内核函数是 IoStopTimer。

I/O 定时器例程的形式如下:

```
VOID OnTimer(  
    IN PDEVICE_OBJECT DeviceObject,  
    IN PVOID Context)  
{  
}
```

需要指出的是, I/O 定时器例程运行在 DISPATCH_LEVEL 级别, 因此在这个例程中不能使用分页内存, 否则会引起页故障从而导致系统崩溃。另外 I/O 定时器是运行在任意线程的, 不一定是 IRP 发起的线程中, 因此不能直接使用应用程序的内存地址。

10.1.2 示例代码

下面的列子演示了如何在驱动程序中使用 I/O 定时器。这个例子每隔 1s 进入 I/O 定时器例程, 每隔 3s 输出一行调试信息。

首先在 DriverEntry 中对 I/O 定时器进行初始化。

```
#001 .....  
#002 IoInitializeTimer(pDevObj, OnTimer, NULL);  
#003 .....
```

然后在设备扩展中, 加入一个计数变量, 这个计数变量负责记录间隔的秒数。

```
#001 typedef struct _DEVICE_EXTENSION {  
#002 .....  
#003     LONG lTimerCount;  
#004 .....  
#005 } DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

然后定义两个 IOCTL 码, 分别是 IOCTL_START_TIMER 和 IOCTL_STOP。当应用程序发起这两个 IOCTL 请求后, 分别开启和关闭定时器。

```
#001 #pragma PAGEDCODE  
#002 NTSTATUS HelloDDKDeviceIOControl(IN PDEVICE_OBJECT pDevObj,  
#003                                     IN PIRP pIrp)  
#004 {  
#005     NTSTATUS status = STATUS_SUCCESS;  
#006     KdPrint(("Enter HelloDDKDeviceIOControl\n"));  
#007     //得到当前堆栈  
#008     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);  
#009     //得到输入缓冲区大小
```

```

#010     ULONG cbin = stack->Parameters.DeviceIoControl.InputBufferLength;
#011     //得到输出缓冲区大小
#012     ULONG cbout = stack->Parameters.DeviceIoControl.OutputBufferLength;
#013     //得到 IOCTL 码
#014     ULONG code = stack->Parameters.DeviceIoControl.IoControlCode;
#015     PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
#016         pDevObj->DeviceExtension;
#017     ULONG info = 0;
#018     //判别是哪个 IOCTL 码
#019     switch (code)
#020     {
#021         case IOCTL_START_TIMER:
#022         {
#023             KdPrint(("IOCTL_START_TIMER\n"));
#024             pDevExt->lTimerCount = TIMER_OUT;
#025             IoStartTimer(pDevObj);
#026             break;
#027         }
#028         case IOCTL_STOP:
#029         {
#030             KdPrint(("IOCTL_STOP\n"));
#031             IoStopTimer(pDevObj);
#032             break;
#033         }
#034         default:
#035             status = STATUS_INVALID_VARIANT;
#036     }
#037
#038     //设置 IRP 状态
#039     pIrp->IoStatus.Status = status;
#040     //设置 IRP 操作字节数
#041     pIrp->IoStatus.Information = info;
#042     //结束 IRP 请求
#043     IoCompleteRequest(pIrp, IO_NO_INCREMENT);
#044     KdPrint(("Leave HelloDKDeviceIoControl\n"));
#045     return status;
#046 }

```

此段代码可以在配套光盘中本章的 IoTimer_Test 目录下找到。

下面编写定时器例程，这个例程是运行在 DISPATCH_LEVEL 的 IRQL 级别。首先注意在这个例程中不能使用分页内存，另外在函数首部要使用 `#pragma LOCKEDCODE`。为了证明该线程可以运行在任意线程上下文，在这个例程的最后，我们得到当前线程的名称，并用 log 信息将其输出。

```

#001 #pragma LOCKEDCODE
#002 VOID OnTimer(
#003     IN PDEVICE_OBJECT DeviceObject,
#004     IN PVOID Context)
#005 {
#006     //得到设备扩展
#007     PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
#008         DeviceObject->DeviceExtension;
#009     KdPrint(("Enter OnTimer!\n"));
#010     //将计数器自锁减一
#011     InterlockedDecrement(&pDevExt->lTimerCount);
#012

```



```

#013 //如果计数器减到 0，重新编程 TIMER_OUT，整个过程是互锁运算
#014 LONG previousCount = InterlockedCompareExchange(&pDevExt->lTimerCount,
TIMER_OUT,0);
#015 //每隔 3s，计数器一个循环，输出以下 log
#016 if (previousCount==0)
#017 {
#018     KdPrint(("&d seconds time out!\n",TIMER_OUT));
#019 }
#020 //证明该线程运行在任意线程上下文的
#021 PEPROCESS pEProcess = IoGetCurrentProcess();
#022 PTSTR ProcessName = (PTSTR)((ULONG)pEProcess + 0x174);//即可得到用户进程
#023 KdPrint(("The current process is %s\n",ProcessName));
#024 }

```

此段代码可以在配套光盘中本章的 IoTimer_Test 目录下找到。

如图 10-1 所示为运行后 log 输出的结果，每隔 1s 进入一次 I/O 定时器例程，另外实现了每隔 3s 输出 1 行调试信息。I/O 定时器例程还列出了当前的进程名，在这个例子中显示的是 Idle 进程，而不是发起 IOCTL 请求的进程。

Idle 进程和系统进程一样，都是一种特殊进程，它没有相应的 exe 文件与之对应，在 Windows 启动后这两个进程就存在系统中了。如图 10-2 所示列出了任务管理器中的 Idle 进程。

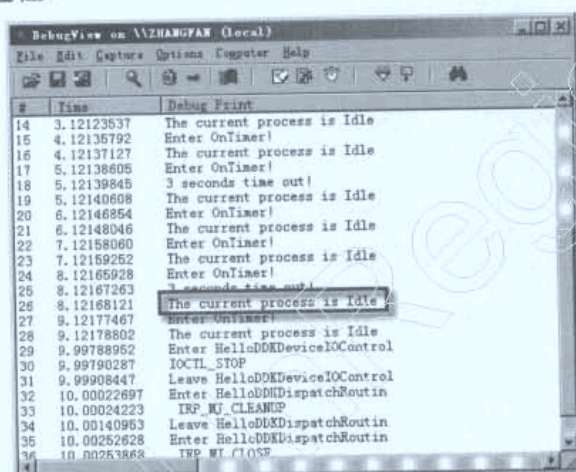


图 10-1 DebugView 输出 log



图 10-2 任务管理器中的 Idle 进程

10.2 定时器实现方式二

第二种定时器是 DPC 定时器，这种定时器的使用比 I/O 定时器更加灵活。

10.2.1 DPC 定时器

在驱动程序中第二种使用定时器的方法是使用 DPC 定时器，这种定时器更加灵活，可以对任意间隔时间进行定时。DPC 定时器内部使用定时器对象 KTIMER，当对定时器设

定一个时间间隔后，每隔这段时间操作系统就会将一个 DPC 例程插入 DPC 队列。当操作系统读取 DPC 队列时，对应的 DPC 例程会被执行。DPC 定时器例程相当于定时器的回调函数。

在使用 DPC 定时器前，需要初始化 DPC 对象和定时器对象。初始化定时器对象使用内核函数 `KeInitializeTimer`，其声明如下：

```
VOID
KeInitializeTimer(
    IN PKTIMER Timer
);
```

其中参数 `Timer` 是定时器的指针，这个初始化函数的名称和 `IoInitializeTimer` 函数很类似，读者注意不要搞混。

初始化 DPC 对象的内核函数是 `KeInitializeDpc`，其声明如下：

```
VOID
KeInitializeDpc(
    IN PRKDPC Dpc,
    IN PKDEFERRED_ROUTINE DeferredRoutine,
    IN PVOID DeferredContext
);
```

- 第一个参数 `Dpc`：这个参数是初始化的 DPC 对象指针。
- 第二个参数 `DeferredRoutine`：这个参数是与 DPC 关联的 DPC 例程。每当定时器间隔到时，操作系统就会将一个 DPC 对象插入队列，并触发该 DPC 例程运行。
- 第三个参数 `DeferredContext`：这个参数是传入 DPC 例程的参数。

开启定时器的内核函数是 `KeSetTimer`，其声明如下：

```
BOOLEAN
KeSetTimer(
    IN PKTIMER Timer,
    IN LARGE_INTEGER DueTime,
    IN PRKDPC Dpc OPTIONAL
);
```

- 第一个参数 `Timer`：这个参数是定时器对象指针。
- 第二个参数 `DueTime`：这个参数设定时间间隔。
- 第三个参数 `Dpc`：这个参数是传入 DPC 例程的参数。
- 返回值：表明是否成功设定了定时器。

取消定时器的内核函数 `KeCancelTimer`，其声明如下：

```
BOOLEAN
KeCancelTimer(
    IN PKTIMER Timer
);
```

在调用 `KeSetTimer` 函数后，经过 `DueTime` 的时间，操作系统调用 DPC 例程。如果 `DueTime` 是正数，则代表绝对时间，该时间是从 1601 年的 1 月 1 日到触发 DPC 例程的那

个时刻，时间单位是 100ns。如果 DueTime 是负数，意味着间隔多长时间，时间单位同样是 100ns。

在调用 KeSetTimer 后，只会触发一次 DPC 例程。如果想周期的触发 DPC 例程，需要在 DPC 例程被触发后，再次调用 KeSetTimer 函数。

10.2.2 示例代码

下面的代码演示了如何在驱动程序中使用 DPC 定时器。这个例子定义了 IOCTL_START_TIMER 和 IOCTL_STOP_TIMER 两个 IOCTL，分别对应开启定时器和关闭定时器。在传递 IOCTL_START_TIMER 时，将间隔的时间从应用程序传递到驱动程序中，这样时间间隔是由应用程序控制的。在应用程序中可以调用 DeviceIoControl 让驱动程序开启定时器和关闭定时器，并告诉定时器的间隔时间。

```
#001 .....
#002     DeviceIoControl(hDevice, IOCTL_START_TIMER, &dwMicroSeconds, sizeof(DWORD),
NULL, 0, &dwOutput, NULL);
#003     Sleep(10000);
#004     DeviceIoControl(hDevice, IOCTL_STOP_TIMER, NULL, 0, NULL, 0, &dwOutput,
NULL);
#005 .....
```

在驱动程序中，可以在设备扩展中加入 DPC 对象和定时器对象。

```
typedef struct _DEVICE_EXTENSION {
    PDEVICE_OBJECT pDevice;
    UNICODE_STRING ustrDeviceName;           //设备名称
    UNICODE_STRING ustrSymLinkName;          //符号链接名
    KDPC pollingDPC;                          // 存储 DPC 对象
    KTIMER pollingTimer;                      // 存储定时器对象
    LARGE_INTEGER pollingInterval;           // 记录定时器间隔时间
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

在初始化 DPC 对象时，需要将设备对象作为参数传递给 DPC 例程。这样在 DPC 例程中可以通过设备对象指针获得间隔时间，因为时间间隔记录在设备扩展中。

```
#001     KeInitializeTimer( &pDevExt->pollingTimer );
#002     KeInitializeDpc( &pDevExt->pollingDPC,
#003                     PollingTimerDpc,
#004                     (PVOID) pDevObj );
```

在 IRP_MJ_DEVICE_CONTROL 的派遣函数中，接收 IOCTL_START_TIMER 后，可以将应用程序传递进来的间隔时间记录下来，并且调用 KeSetTimer 函数开启定时器。在接收 IOCTL_STOP_TIMER 后，应该调用 KeCancelTimer 函数停止定时器。

```
#001     #pragma PAGEDCODE
#002     NTSTATUS HelloDDKDeviceIoControl(IN PDEVICE_OBJECT pDevObj,
#003                                       IN PIRP pIrp)
#004     {
#005         NTSTATUS status = STATUS_SUCCESS;
```

```

#006 KdPrint(("Enter HelloDDKDeviceIOControl\n"));
#007 //得到当前堆栈
#008 PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
#009 //得到输入缓冲区大小
#010 ULONG cbin = stack->Parameters.DeviceIoControl.InputBufferLength;
#011 //得到输出缓冲区大小
#012 ULONG cbout = stack->Parameters.DeviceIoControl.OutputBufferLength;
#013 //得到 IOCTL 码
#014 ULONG code = stack->Parameters.DeviceIoControl.IoControlCode;
#015 //得到设备扩展
#016 PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
#017     pDevObj->DeviceExtension;
#018 ULONG info = 0;
#019 switch (code)
#020 {
#021     case IOCTL_START_TIMER:
#022     {
#023         KdPrint(("IOCTL_START_TIMER!\n"));
#024         ULONG ulMicroSeconds = *(PULONG)pIrp->AssociatedIrp.SystemBuffer;
#025         //将 32 位整数转化为 64 位整数
#026         pDevExt->pollingInterval = RtlConvertLongToLargeInteger
(ulMicroSeconds * -10 );
#027         //设置计时器
#028         KeSetTimer(
#029             &pDevExt->pollingTimer,
#030             pDevExt->pollingInterval,
#031             &pDevExt->pollingDPC );
#032         break;
#033     }
#034     case IOCTL_STOP_TIMER:
#035     {
#036         KdPrint(("IOCTL_STOP_TIMER!\n"));
#037         //停止计时器
#038         KeCancelTimer(&pDevExt->pollingTimer);
#039         break;
#040     }
#041     default:
#042         status = STATUS_INVALID_VARIANT;
#043 }
#044 //设置 IRP 完成状态
#045 pIrp->IoStatus.Status = status;
#046 //设置 IRP 操作字节数
#047 pIrp->IoStatus.Information = info;
#048 //结束 IRP 请求
#049 IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#050 KdPrint(("Leave HelloDDKDeviceIOControl\n"));
#051 return status;
#052 }

```

此段代码可以在配套光盘中本章的 DPCTimer_Test 目录下找到。

最后是编写 DPC 例程，每次执行 KeSetTimer 只会触发一次 DPC 例程。为了能周期地调用 DPC 例程，应该在 DPC 例程中再次调用 KeSetTimer，并且时间间隔保持不变。需要注意的是，DPC 例程是运行在 DISPATCH_LEVEL 的 IRQL 级别，因此不能使用分页内存，并且在 DPC 例程的开始处用 #pragma LOCKEDCODE 修饰。最后，为了验证该例程可以运

行在任意线程上下文中，加入了获取当前进程的代码，当前进程的名字作为 log 信息输出。

```
#001 #pragma LOCKEDCODE
#002 VOID PollingTimerDpc( IN PKDPC pDpc,
#003                        IN PVOID pContext,
#004                        IN PVOID SysArg1,
#005                        IN PVOID SysArg2 )
#006 {
#007     //得到设备对象
#008     PDEVICE_OBJECT pDevObj = (PDEVICE_OBJECT)pContext;
#009     //得到设备扩展
#010     PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
#011     //开启计时器
#012     KeSetTimer(
#013         &pdx->pollingTimer,
#014         pdx->pollingInterval,
#015         &pdx->pollingDPC );
#016     KdPrint(("PollingTimerDpc\n"));
#017     //检验是运行在任意线程上下文
#018     PEPROCESS pEProcess = IoGetCurrentProcess();
#019     PTSTR ProcessName = (PTSTR)((ULONG)pEProcess + 0x174);
#020     KdPrint((" %s\n", ProcessName));
#021 }
```

此段代码可以在配套光盘中本章的 DPCTimer_Test 目录下找到。

本例的运行结果和 10.1.2 的例子运行结果类似。

10.3 等待

等待是驱动程序中经常用到的。笔者将 DDK 等待的方法总结了一下，共有四种办法，读者可以根据自己的需要进行选择。

10.3.1 第一种方法：使用 KeWaitForSingleObject

第一种方法是使用 KeWaitForSingleObject 函数。该函数在前面已经介绍过，其主要用来等待内核同步对象，但是也可以用来等待一段时间。

首先初始化一个内核同步对象，其初始状态为未激发状态。然后调用 KeWaitForSingleObject，并对其设置 timeout 参数，该参数是需要等待的时间。下面的代码演示了如何用 KeWaitForSingleObject 等待一段时间。

```
#001 #pragma PAGEDCODE
#002 VOID WaitMicroSecond1(ULONG ulMircoSecond)
#003 {
#004     KEVENT kEvent;
#005     KdPrint(("Thread suspends %d MircoSeconds...", ulMircoSecond));
#006     //初始化一个未激发的内核事件
#007     KeInitializeEvent(&kEvent, SynchronizationEvent, FALSE);
#008     //等待时间的单位是 100ns (纳秒)，将 us (微秒) 转换成这个单位
#009     //负数代表是从此刻到未来的某个时刻
#010     LARGE_INTEGER timeout = RtlConvertLongToLargeInteger(-10*ulMircoSecond);
```

```

#011 //在经过timeout后, 线程继续运行
#012 KeWaitForSingleObject(&kEvent,
#013     Executive,
#014     KernelMode,
#015     FALSE,
#016     &timeout);
#017 KdPrint(("Thread is running again!\n"));
#018 }

```

此段代码可以在配套光盘中本章的 Waiting_Test 目录下找到。

10.3.2 第二种方法: 使用 KeDelayExecutionThread

第二种方法是使用内核函数 KeDelayExecutionThread, 该内核函数和 KeWaitForSingleObject 类似, 都是强制当前线程进入睡眠状态。经过指定的睡眠时间后, 线程恢复运行。下面的代码演示了如何使用 KeDelayExecutionThread 等待一段时间。

```

#001 #pragma PAGEDCODE
#002 VOID WaitMicroSecond2(ULONG ulMircoSecond)
#003 {
#004     KdPrint(("Thread suspends %d MircoSeconds...", ulMircoSecond));
#005     //等待时间的单位是 100ns, 将 us 转换成这个单位
#006     //负数代表是从此刻到未来的某个时刻
#007     LARGE_INTEGER timeout = RtlConvertLongToLargeInteger(-10*ulMircoSecond);
#008     //此种方法类似于 KeWaitForSingleObject
#009     //将当前线程进入睡眠状态, 间隔时间到后转入运行状态
#010     KeDelayExecutionThread(KernelMode, FALSE, &timeout);
#011     KdPrint(("Thread is running again!\n"));
#012 }

```

此段代码可以在配套光盘中本章的 Waiting_Test 目录下找到。

10.3.3 第三种方法: 使用 KeStallExecutionProcessor

第三种方法是使用内核函数 KeStallExecutionProcessor, 该内核函数是让 CPU 处于忙等待状态, 而不是处于睡眠。经过指定时间后, 继续让线程运行。

这种方法是让 CPU 不停地等待, 而不是让线程进入休眠, 类似于自旋锁。因此这种方法浪费 CPU 时间, DDK 文档规定, KeStallExecutionProcessor 不宜超过 50us。由于没有将线程进入睡眠, 也不会发生线程的切换, 因此这种方法的延时比较精确。下面的代码演示了如何使用 KeStallExecutionProcessor 等待一段时间。

```

#001 #pragma PAGEDCODE
#002 VOID WaitMicroSecond3(ULONG ulMircoSecond)
#003 {
#004     KdPrint(("Thread suspends %d MircoSeconds...", ulMircoSecond));
#005     //忙等待, 此种方法属于忙等待, 比较浪费 CPU 时间
#006     //因此使用该方法不宜超过 50us
#007     KeStallExecutionProcessor(ulMircoSecond);
#008     KdPrint(("Thread is running again!\n"));
#009 }

```

此段代码可以在配套光盘中本章的 Waiting_Test 目录下找到。

10.3.4 第四种方法：使用定时器

第四种方法是使用定时器对象，注意这和前面讲的 DPC 定时器略有不同。这里没有用到 DPC 对象和 DPC 例程，因此当指定时间到后，不会进入 DPC 例程。

定时器对象和其他内核同步对象一样，也是有两个状态，一个是未激发状态，一个是激发状态。在初始化定时器的時候，定时器处于未激发状态。当使用 KeSetTimer 后，经过指定时间后，进入激发状态。这样就可以使用 KeWaitForSingleObject 函数对定时器对象进行等待，以下是一段演示代码：

```
#001  #pragma PAGEDCODE
#002  VOID WaitMicroSecond4(ULONG ulMircoSecond)
#003  {
#004  //使用定时器
#005  KTIMER kTimer; //内核定时器
#006  //初始化定时器
#007  KeInitializeTimer(&kTimer);
#008  LARGE_INTEGER timeout = RtlConvertLongToLargeInteger( ulMircoSecond * -10 );
#009  //注意这个定时器没有和 DPC 对象关联
#010  KeSetTimer(&kTimer, timeout, NULL);
#011  KdPrint(("Thread suspends %d MircoSeconds...", ulMircoSecond));
#012  //等待同步事件
#013  KeWaitForSingleObject(&kTimer, Executive, KernelMode, FALSE, NULL);
#014  KdPrint(("Thread is running again!\n"));
#015 }
```

此段代码可以在配套光盘中本章的 Waiting_Test 目录下找到。

10.4 时间相关的其他内核函数

除了上述介绍的定时器、等待函数外，DDK 中还提供了丰富的时间处理函数，本节针对一些常用的时间处理函数做简单介绍。

10.4.1 时间相关函数

获取当前系统时间的内核函数是 KeQuerySystemTime。这个时间是从 1601 年 1 月 1 日起经过的时间，单位是 100ns，并且以格林尼治时间计时为准。该函数的声明如下：

```
VOID
KeQuerySystemTime(
    OUT PLARGE_INTEGER CurrentTime
);
```

➤ 第一个参数 CurrentTime：这个参数返回当前系统时间，以格林尼治时间计时。

ExSystemTimeToLocalTime 函数将系统时间转换为当前时区对应的时间，当前时区可以在控制面板中设置。其函数的声明如下：

```

VOID
ExSystemTimeToLocalTime(
    IN PLARGE_INTEGER SystemTime,
    OUT PLARGE_INTEGER LocalTime
);

```

- 第一个参数 SystemTime: 这个参数输入系统时间, 以格林尼治时间为准。
- 第二个参数 LocalTime: 这个参数是当前时区的时间。

ExLocalTimeToSystemTime 函数可以将当前时区的时间转换为系统时间, 即格林尼治时间。ExLocalTimeToSystemTime 函数的声明如下:

```

VOID
ExLocalTimeToSystemTime(
    IN PLARGE_INTEGER LocalTime,
    OUT PLARGE_INTEGER SystemTime
);

```

- 第一个参数 LocalTime: 这个参数是输入的当前时区时间, 以控制面板中设置的时区为准。
- 第二个参数 SystemTime: 这个参数输入系统时间, 以格林尼治时间为准。

RtlTimeFieldsToTime 函数可以由当前的年月日得到系统时间, 其函数声明如下:

```

BOOLEAN
RtlTimeFieldsToTime(
    IN PTIME_FIELDS TimeFields,
    IN PLARGE_INTEGER Time
);

```

- 第一个参数 TimeFields: 这个参数是输入的年月日等信息。
- 第二个参数 Time: 这个参数输出转换的系统时间。
- 返回值: 返回表明是否转换成功。

输入的 TimeFields 是一个 TIME_FIELDS 的数据结构, 记录年月日等信息, 其声明如下:

```

typedef struct TIME_FIELDS {
    CSHORT Year;
    CSHORT Month;
    CSHORT Day;
    CSHORT Hour;
    CSHORT Minute;
    CSHORT Second;
    CSHORT Milliseconds;
    CSHORT Weekday;
} TIME_FIELDS;

```

- 第一个参数 Year: 年, 最小取值 1601。
- 第二个参数 Month: 月, 取值范围是 1 到 12。
- 第三个参数 Day: 日期, 取值范围是 1 到 31。
- 第四个参数 Hour: 小时, 取值范围是 0 到 23。
- 第五个参数 Minute: 分钟, 取值范围是从 0 到 59。

Windows 驱动开发技术详解

- 第六个参数 Second: s, 取值范围是从 0 到 59。
- 第七个参数 Milliseconds: ms, 取值范围是从 0 到 999。
- 第八个参数 Weekday: 礼拜几, 取值范围是从 0 到 6, 代表周日到周六。

RtlTimeToTimeFields 函数将系统时间得到是具体月日年等信息, 其声明如下:

```
VOID  
RtlTimeToTimeFields(  
    IN PLARGE_INTEGER Time,  
    IN PTIME_FIELDS TimeFields  
);
```

- 第一个参数 Time: 这个参数是输入的系统时间。
- 第二个参数 TimeFields: 这个参数返回月日年信息, 由 TIME_FIELDS 数据结构表示。

10.4.2 示例代码

下面的代码演示了如何在驱动程序中使用这些时间相关函数。这个例子首先得到当前的系统时间, 然后将当前的系统时间转换后得到当前时区的时间, 然后再从时区时间得到具体年月日等信息。

```
#001 #pragma PAGEDCODE  
#002 VOID Time_Test()  
#003 {  
#004     LARGE_INTEGER current_system_time;  
#005     //得到当前系统时间  
#006     KeQuerySystemTime(&current_system_time);  
#007  
#008     LARGE_INTEGER current_local_time;  
#009     //从系统时间转换成当地时区时间  
#010     ExSystemTimeToLocalTime(&current_system_time,&current_local_time);  
#011  
#012     TIME_FIELDS current_time_info;  
#013     //由当地时区时间得到月日年信息  
#014     RtlTimeToTimeFields(&current_local_time,&current_time_info);  
#015  
#016     //显示年月日等信息  
#017     KdPrint(("Current year:%d\n",current_time_info.Year));  
#018     KdPrint(("Current month:%d\n",current_time_info.Month));  
#019     KdPrint(("Current day:%d\n",current_time_info.Day));  
#020     KdPrint(("Current Hour:%d\n",current_time_info.Hour));  
#021     KdPrint(("Current Minute:%d\n",current_time_info.Minute));  
#022     KdPrint(("Current Second:%d\n",current_time_info.Second));  
#023     KdPrint(("Current Milliseconds:%d\n",current_time_info.Milliseconds));  
#024     KdPrint(("Current Weekday:%d\n",current_time_info.Weekday));  
#025 }
```

此段代码可以在配套光盘中本章的 Time_Test 目录下找到。

运行后输出的 log 信息如图 10-3 所示。

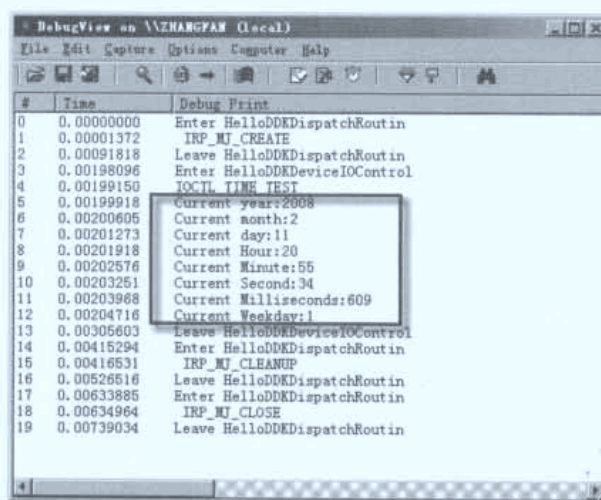


图 10-3 DebugView 输出 log

10.5 IRP 的超时处理

很多时候, IRP 被传送到底层驱动程序后, 由于硬件设备的问题, IRP 不能得到及时的处理, 甚至有可能永远都不会被处理。这时候需要对 IRP 超时情况做出处理, 一旦在规定时间内 IRP 没有被处理, 操作系统就会进入到 IRP 的超时处理函数中。

10.5.1 原理

在驱动程序编程中, 经常遇到一种情况, 对某一设备的操作很久没有反应。如果在规定的时间内没有完成操作, 则要取消该操作。取消例程需要程序员发出取消命令, 如在应用程序中执行 `CancellIO Win32 API` 函数, 或者在驱动程序中执行 `IoCancelIrp` 内核函数, 也可以通过设置 IRP 超时。当 IRP 超时后, 操作系统也会取消 IRP, 从而进入 IRP 的取消例程。

首先初始一个定时器对象和 DPC 对象, 并将 DPC 例程和定时器对象进行关联。在每次对 IRP 操作前, 开启定时器, 并设置好一定的超时。如果在指定时间内对 IRP 的处理没有结束, 那么操作系统就会进入 DPC 例程。

在 DPC 例程中取消还在继续处理的 IRP。如果驱动程序能在超时前结束 IRP 的操作, 则应该取消定时器, 从而保证不会再次取消 IRP。

10.5.2 示例代码

下面的代码演示了如何在驱动程序中对 IRP 的超时进行处理。为了演示超时的效果, 本例在 `IRP_MJ_READ` 的派遣函数中直接返回 pending 状态。这样 IRP 永远不会被结束。

Windows 驱动开发技术详解

在超时以后，会进入超时处理函数。

本例将超时设为 3s。在处理超时的 DPC 例程中，IRP 被强制结束。本例没有考虑 StartIO 例程和取消例程，有兴趣的读者可以将这个例子完善。

以下是派遣函数的代码。

```
#001 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#002                        IN PIRP pIrp)
#003 {
#004     KdPrint(("Enter HelloDDKRead\n"));
#005     //得到设备扩展
#006     PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
#007     pDevObj->DeviceExtension;
#008
#009     //将 IRP 设置为挂起
#010     IoMarkIrpPending(pIrp);
#011     //将挂起的 IRP 记录下来
#012     pDevExt->currentPendingIRP = pIrp;
#013     //定义 3s 的超时
#014     ULONG ulMicroSecond = 3000000;
#015
#016     //将 32 位整数转化成 64 位整数
#017     LARGE_INTEGER timeout = RtlConvertLongToLargeInteger(-10*ulMicroSecond);
#018     //开启计时器
#019     KeSetTimer(
#020         &pDevExt->pollingTimer,
#021         timeout,
#022         &pDevExt->pollingDPC );
#023
#024     KdPrint(("Leave HelloDDKRead\n"));
#025
#026     //返回 pending 状态
#027     return STATUS_PENDING;
#028 }
```

此段代码可以在配套光盘中本章的 IRPTimeout_Test 目录下找到。

以下是超时后进入的 DPC 例程。

```
#001 #pragma LOCKEDCODE
#002 VOID OnTimerDpc( IN PKDPC pDpc,
#003                IN PVOID pContext,
#004                IN PVOID SysArg1,
#005                IN PVOID SysArg2 )
#006 {
#007     //得到设备对象指针
#008     PDEVICE_OBJECT pDevObj = (PDEVICE_OBJECT)pContext;
#009     //得到设备扩展
#010     PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
#011     //得到当前挂起的 IRP
#012     PIRP currentPendingIRP = pdx->currentPendingIRP;
#013     KdPrint(("Cancel the current pending irp!\n"));
#014
#015     //设置完成状态为 STATUS_CANCELLED
#016     currentPendingIRP->IoStatus.Status = STATUS_CANCELLED;
#017     //设置 IRP 的操作字节数
```

```

#018    currentPendingIRP->IoStatus.Information = 0;
#019    //结束 IRP 请求
#020    IoCompleteRequest( currentPendingIRP, IO_NO_INCREMENT );
#021    }

```

此段代码可以在配套光盘中本章的 IRPTimeout_Test 目录下找到。

在应用程序中，发起了两次 ReadFile 请求，由于其派遣函数没有完成该 IRP，因此 ReadFile 会一直停在那里。直到 3s 超时后，ReadFile 才会返回。如图 10-4 所示为驱动程序输出的 log 信息。

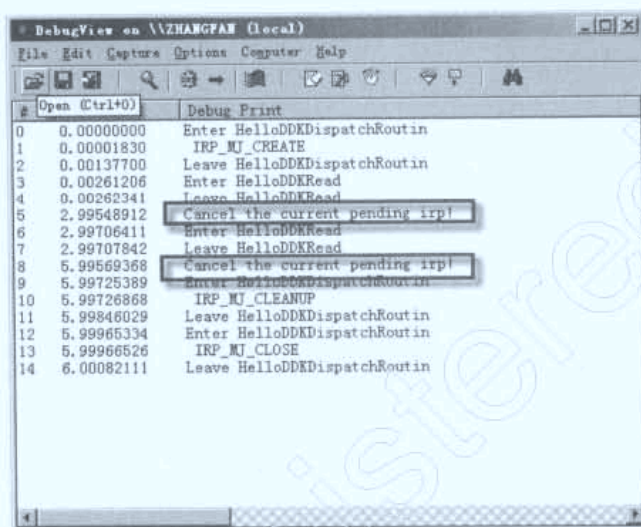


图 10-4 DebugView 输出 log

10.6 小结

定时器是驱动程序中经常用到的工具。本章介绍了两种常用的定时器，一种是 I/O 定时器，一种是 DPC 定时器。其中 I/O 定时器的最小间隔单位是 1s，但用法比较简单。而 DPC 定时器最小的时间间隔是 ms，但用法比 I/O 定时器复杂。

本章还总结了在内核模式下的四种等待方法，读者可以利用这些方法灵活地用在自己的驱动程序中。最后本章还介绍了如何对 IRP 的超时情况进行处理。

第 11 章 驱动程序调用驱动程序

在驱动程序开发中，经常需要一个驱动程序调用另外一个驱动程序。例如，虚拟串口转 USB 设备的驱动程序，这种驱动程序首先创建一个虚拟串口设备，对这个虚拟串口设备的读写请求会转发到一个 USB 设备上去。这时就需要在虚拟串口驱动程序中调用 USB 驱动程序。

类似的例子还很多，本章先从原理上介绍如何实现驱动程序调用另外的驱动程序，并给出相应的示例代码。另外，通过这部分知识的学习，将有助于读者更深地理解在 Windows 内核中 IRP 请求的构造、传递等内部特性。

11.1 以文件句柄形式调用其他驱动程序

本节介绍以文件句柄方式调用其他驱动程序的方法，这种方法类似于在应用程序中调用驱动程序。这种方法使用简单，不需要程序员对 Windows 底层了解过多的知识。

11.1.1 准备一个标准驱动

在介绍驱动程序调用其他驱动程序之前，首先准备一个标准驱动程序，作为“目标”驱动程序。本章后面介绍的所有驱动程序，都以不同方法调用这个“目标”驱动程序。

这个“目标”驱动程序创建一个模拟设备，模拟设备支持异步读取操作。事先规定每次对设备读取需要耗时 3s，因为这样可以很好地演示异步读取操作。

对于这样的驱动程序，应该设置一个定时器，定时器的间隔设置为 3s。另外在 IRP_MJ_READ 的派遣函数中不结束 IRP 请求，而是将 IRP 请求挂起，并且在派遣函数退出前开启定时器。这样 3s 后就会进入定时器的回调函数中，并在回调函数中结束 IRP 请求。

我们把这个“目标”驱动程序命名为 DriverA，调用 DriverA 的驱动程序被命名为

DriverB。以下列出的是 DriverA 的部分代码。

首先是 DriverA 的 IRP_MJ_READ 派遣函数。

```
#001 #pragma PAGEDCODE
#002 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#003                        IN PIRP pIrp)
#004 {
#005     KdPrint(("DriverA:Enter A HelloDDKRead\n"));
#006     //获取设备扩展
#007     PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
#008         pDevObj->DeviceExtension;
#009
#010     //将 IRP 设置为挂起
#011     IoMarkIrpPending(pIrp);
#012     //将挂起的 IRP 记录下来
#013     pDevExt->currentPendingIRP = pIrp;
#014
#015     //将定时器的间隔设置为 3s
#016     ULONG ulMicroSecond = 3000000;
#017     //将 32 位整数转化成 64 位整数
#018     LARGE_INTEGER timeout = RtlConvertLongToLargeInteger(-10*ulMicroSecond);
#019     //开启定时器
#020     KeSetTimer(
#021         &pDevExt->pollingTimer,
#022         timeout,
#023         &pDevExt->pollingDPC );
#024     KdPrint(("DriverA:Leave A HelloDDKRead\n"));
#025
#026     //返回 pending 状态
#027     return STATUS_PENDING;
#028 }
```

此段代码可以在配套光盘中本章的 Test1 目录下找到。

下面是 DriverA 的超时回调 DPC 例程。

```
#001 #pragma LOCKEDCODE
#002 VOID OnTimerDpc( IN PKDPC pDpc,
#003                 IN PVOID pContext,
#004                 IN PVOID SysArg1,
#005                 IN PVOID SysArg2 )
#006 {
#007     //从上下文中获取设备对象指针
#008     PDEVICE_OBJECT pDevObj = (PDEVICE_OBJECT)pContext;
#009     //从设备对象指针获取设备扩展指针
#010     PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
#011
#012     //从设备扩展中获得当前正在挂起的 IRP
#013     PIRP currentPendingIRP = pdx->currentPendingIRP;
#014
#015     KdPrint(("DriverA:complete the Driver A IRP_MJ_READ irp!\n"));
#016
#017     //设置完成状态为 STATUS_CANCELLED
#018     currentPendingIRP->IoStatus.Status = STATUS_SUCCESS;
#019     //设置 IRP 操作的字节数
#020     currentPendingIRP->IoStatus.Information = 0;
```



```
#021    //将 IRP 结束
#022    IoCompleteRequest( currentPendingIRP, IO_NO_INCREMENT );
#023    }
```

此段代码可以在配套光盘中本章的 Test1 目录下找到。

以上就是笔者准备的“目标”驱动程序 DriverA，本章后面将介绍几种不同的方法，使另外的驱动程序（以下称为 DriverB）可以调用 DriverA 的读取功能。为了演示 DriverB 调用 DriverA，并使读者可以看清楚 log 信息来源于哪个驱动程序，笔者让 DriverA 在输出 log 信息之前加上“DriverA:”这个前缀，而让 DriverB 在输出 log 信息之前加上“DriverB:”这个前缀。

在前 10 章的例子中，都是应用程序作为驱动程序的“客户端”。而本章 DriverA 的客户端不再是应用程序，而变成了另外的驱动程序（DriverB）。而 DriverB 作为应用程序的客户端，其调用关系如图 11-1 所示。



图 11-1 驱动调用驱动

DriverB 有多种方法调用 DriverA，这些方法可以是同步调用，也可是异步调用。

11.1.2 获得设备句柄

上一节笔者准备了一个目标驱动程序 DriverA，后面将陆续介绍如何编写 DriverB 调用 DriverA。下面回忆一下应用程序是如何调用驱动程序的，应该先用 CreateFile 函数打开设备，然后用 ReadFile 函数读取设备，最后用 CloseHandle 函数关闭设备。

在驱动程序中，打开设备使用 ZwCreateFile 内核函数，它会返回设备句柄。这里要讨论一下如何用 ZwCreateFile 内核函数打开“同步”设备和“异步”设备。ZwCreateFile 内核函数的声明如下：

```
NTSTATUS
ZwCreateFile(
    OUT PHANDLE FileHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    IN PLARGE_INTEGER AllocationSize OPTIONAL,
    IN ULONG FileAttributes,
    IN ULONG ShareAccess,
    IN ULONG CreateDisposition,
    IN ULONG CreateOptions,
    IN PVOID EaBuffer OPTIONAL,
    IN ULONG EaLength
);
```

ZwCreateFile 内核函数的大多数参数已经详细介绍过，这里只介绍打开“同步”设备和“异步”设备的区别。如果打开“同步”设备，第二个参数 DesiredAccess 需要设置为

SYNCHRONIZE, 并且倒数第三个参数 CreateOptions 需要指定为 FILE_SYNCHRONOUS_IO_NONALERT 或者 FILE_SYNCHRONOUS_IO_ALERT。

```
#001 //同步打开设备
#002 //设定了 FILE_SYNCHRONOUS_IO_NONALERT 或者 FILE_SYNCHRONOUS_IO_ALERT 为同步打
//开设备
#003 ntStatus = ZwCreateFile(&hDevice,
#004 FILE_READ_ATTRIBUTES|SYNCHRONIZE,
#005 &objectAttributes,
#006 &status_block,
#007 NULL,FILE_ATTRIBUTE_NORMAL,FILE_SHARE_READ,
#008 FILE_OPEN_IF,FILE_SYNCHRONOUS_IO_NONALERT,NULL,0);
```

如果打开“异步”设备,第二个参数 DesiredAccess 不能设置为 SYNCHRONIZE,并且倒数第三个参数 CreateOptions 不能指定 FILE_SYNCHRONOUS_IO_NONALERT 或参数 FILE_SYNCHRONOUS_IO_ALERT。

```
//异步打开设备
//没有设定了 FILE_SYNCHRONOUS_IO_NONALERT 和 FILE_SYNCHRONOUS_IO_ALERT 为异步打开设备
ntStatus = ZwCreateFile(&hDevice,
FILE_READ_ATTRIBUTES, //没有设 SYNCHRONIZE
&objectAttributes,
&status_block,
NULL,FILE_ATTRIBUTE_NORMAL,FILE_SHARE_READ,
FILE_OPEN_IF,0,NULL,0);
```

11.1.3 同步调用

在应用程序中, ReadFile 函数既可以用来读取文件,又可以用来读取设备。同样,在驱动程序中, ZwReadFile 内核函数既可以用来读取文件,又可以用来读取设备。

如果用 ZwReadFile 内核函数同步读取设备,操作对应的 IRP_MJ_READ 请求被结束后,函数才会返回,否则这个函数会一直等待 IRP_MJ_READ 请求被结束。如果用 ZwReadFile 内核函数异步读取设备,即使操作对应的 IRP_MJ_READ 请求没有被结束,函数也会立即返回。

本节介绍的 DriverB 是利用同步读取的方法调用 DriverA,如用 ZwReadFile 内核函数同步读取 DriverA 的设备,它的内部操作过程如下:

① 在 DriverB 中用 ZwReadFile 内核函数读取 DriverA 的设备对象。ZwReadFile 内核函数内部会创建 IRP_MJ_READ 类型的 IRP,然后将这个 IRP 当做参数传递给 DriverA 的派遣函数。

② DriverA 的派遣函数没有结束 IRP 请求,而是将 IRP 请求“挂起”。

③ ZwReadFile 函数会一直等待 IRP 中的一个事件,此时当前线程进入睡眠状态。

④ 3s 后,触发 DriverA 的定时器例程,这时 IRP 请求被结束,IRP 中的相关事件也被设置。

⑤ 由于相关事件被设置,刚才休眠的线程恢复运行,ZwReadFile 内核函数退出。

Windows 驱动开发技术详解

明白了上述过程后,使用 `ZwReadFile` 内核函数进行同步读取就变得很简单了。下面的代码演示了如何使用 `ZwReadFile` 内核函数同步读取 `DriverA` 的设备。

另外, `DriverB` 还需要做应用程序的客户端,因此, `DriverB` 的派遣函数还需要结束来自应用程序的 `IRP` 请求。

```
#001 #pragma PAGEDCODE
#002 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#003                        IN PIRP pIrp)
#004 {
#005     KdPrint(("DriverB:Enter B HelloDDKRead\n"));
#006     NTSTATUS ntStatus = STATUS_SUCCESS;
#007
#008     //初始化 DriverA 的设备名
#009     UNICODE_STRING DeviceName;
#010     RtlInitUnicodeString(&DeviceName, L"\\Device\\MyDDKDeviceA");
#011
#012     //初始化 objectAttributes
#013     OBJECT_ATTRIBUTES objectAttributes;
#014     InitializeObjectAttributes(&objectAttributes,
#015                               &DeviceName,
#016                               OBJ_CASE_INSENSITIVE,
#017                               NULL,
#018                               NULL);
#019
#020     HANDLE hDevice; //定义一个设备句柄
#021     IO_STATUS_BLOCK status_block; //定义一个 i/o 状态
#022     //同步打开设备
#023     //设定了 FILE_SYNCHRONOUS_IO_NONALERT 或者 FILE_SYNCHRONOUS_IO_ALERT 为同步
#024     //打开设备
#025     ntStatus = ZwCreateFile(&hDevice,
#026                             FILE_READ_ATTRIBUTES|SYNCHRONIZE,
#027                             &objectAttributes,
#028                             &status_block,
#029                             NULL, FILE_ATTRIBUTE_NORMAL, FILE_SHARE_READ,
#030                             FILE_OPEN_IF, FILE_SYNCHRONOUS_IO_NONALERT, NULL, 0);
#031     //判断是否成功打开设备
#032     if (NT_SUCCESS(ntStatus))
#033     {
#034         //对设备进行同步读取操作
#035         ZwReadFile(hDevice, NULL, NULL, NULL, &status_block, NULL, 0, NULL, NULL);
#036     }
#037     //关闭设备句柄
#038     ZwClose(hDevice);
#039     //设置 IRP 完成状态
#040     pIrp->IoStatus.Status = ntStatus;
#041     //设置 IRP 操作字节数
#042     pIrp->IoStatus.Information = 0;
#043     //将 IRP 请求结束
#044     IoCompleteRequest(pIrp, IO_NO_INCREMENT);
#045     KdPrint(("DriverB:Leave B HelloDDKRead\n"));
#046     return ntStatus;
#047 }
```

此段代码可以在配套光盘中本章的 `Test1` 目录下找到。

运行这段代码，应该首先加载 DriverA 的驱动程序，然后再加载 DriverB 的驱动程序，最后调用应用程序。如图 11-2 所示为 DriverA 和 DriverB 共同输出的 log 信息。在图中可以观察 DriverA 和 DriverB 输出的先后顺序。

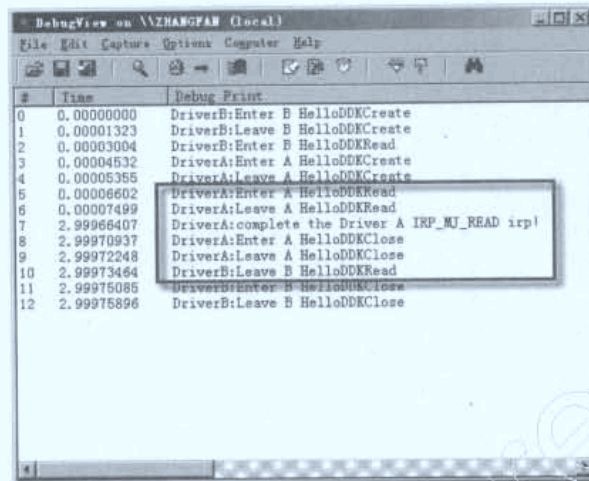


图 11-2 DebugView 输出 log

11.1.4 异步调用方法一

异步读取主要是指 ZwReadFile 内核函数在没有等待 DriverA 真正结束 IRP 请求时，就已经退出。如果用 ZwReadFile 内核函数异步读取 DriverA 的设备，它的内部操作过程如下：

- ① ZwReadFile 内核函数内部创建 IRP_MJ_READ 类型的 IRP，然后将这个 IRP 传递给 DriverA 的派遣函数。
- ② DriverA 派遣函数没有结束 IRP 请求，而是将 IRP “挂起”。
- ③ ZwReadFile 内核函数发现 DriverA 将 IRP_MJ_READ “挂起”，于是它直接返回，返回值是 STATUS_PENDING，这代表读取操作正在进行中。

ZwReadFile 内核函数退出后，无法得知“挂起”的 IRP 何时被结束。因此在调用 ZwReadFile 内核函数前，可以为 IRP 设置一个完成例程。当 IRP 结束时就触发这个完成例程。

在本例中，还将一个事件的指针传递给该完成例程，在完成例程中触发该事件。一旦这个事件被触发，就可以知道 IRP_MJ_READ 请求已经结束。

下面的代码演示了如何用 ZwReadFile 内核函数进行异步读取操作。

```
#001 #pragma PAGEDCODE
#002 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#003                        IN PIRP pIrp)
#004 {
#005     KdPrint(("DriverB:Enter B HelloDDKRead\n"));
#006     NTSTATUS ntStatus = STATUS_SUCCESS;
#007     //初始化 DriverA 的设备名
#008     UNICODE_STRING DeviceName;
```



```

#009   RtlInitUnicodeString( &DeviceName, L"\\Device\\MyDDKDeviceA" );
#010
#011   //初始化 objectAttributes
#012   OBJECT_ATTRIBUTES objectAttributes;
#013   InitializeObjectAttributes(&objectAttributes,
#014                               &DeviceName,
#015                               OBJ_CASE_INSENSITIVE,
#016                               NULL,
#017                               NULL );
#018
#019   HANDLE hDevice; //定义一个设备句柄
#020   IO_STATUS_BLOCK status_block; //定义一个 I/O 状态
#021   //异步打开设备
#022   //没有设定了 FILE_SYNCHRONOUS_IO_NONALERT 和 FILE_SYNCHRONOUS_IO_ALERT 为异步
//打开设备
#023   ntStatus = ZwCreateFile(&hDevice,
#024                           FILE_READ_ATTRIBUTES, //没有设 SYNCHRONIZE
#025                           &objectAttributes,
#026                           &status_block,
#027                           NULL, FILE_ATTRIBUTE_NORMAL, FILE_SHARE_READ,
#028                           FILE_OPEN_IF, 0, NULL, 0);
#029
#030   KEVENT event;
#031   //初始化事件, 用于异步读
#032   KeInitializeEvent(&event, SynchronizationEvent, FALSE);
#033   //将 32 位整数转换成 64 位整数
#034   LARGE_INTEGER offset = RtlConvertLongToLargeInteger(0);
#035   //判断是否成功打开设备
#036   if (NT_SUCCESS(ntStatus))
#037   {
#038       //用 ZwReadFile 函数进行异步读取
#039       ntStatus = ZwReadFile(hDevice, NULL, CompleteDriverA_Read, &event,
&status_block, NULL, 0, &offset, NULL);
#040   }
#041   //判断 ZwReadFile 函数是否返回=STATUS_PENDING
#042   if (ntStatus==STATUS_PENDING)
#043   {
#044       KdPrint(("DriverB:ZwReadFile return STATUS_PENDING!\n"));
#045       KdPrint(("DriverB:Waiting..."));
#046       //等待 IRP 被结束
#047       KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);
#048   }
#049
#050   //关闭设备句柄
#051   ZwClose(hDevice);
#052
#053   ntStatus = STATUS_SUCCESS;
#054   //设置 IRP 完成状态
#055   pIrp->IoStatus.Status = ntStatus;
#056   //设置 IRP 操作字节数
#057   pIrp->IoStatus.Information = 0;
#058   //结束 IRP 请求
#059   IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#060   KdPrint(("DriverB:Leave B HelloDDKRead\n"));
#061   return ntStatus;
#062 }

```

此段代码可以在配套光盘中本章的 Test2 目录下找到。

以下是完成例程的代码。完成例程主要是用一个事件通知 IRP 请求被结束。

```
#001 VOID CompleteDriverA_Read(PVOID context, PIO_STATUS_BLOCK pStatus_block,
ULONG)
#002 {
#003     KdPrint(("DriverB:The Driver A Read completed now!\n"));
#004     //设置事件, 这个事件通知 IRP 请求被结束
#005     KeSetEvent ( (PKEVENT) context, IO_NO_INCREMENT, FALSE);
#006 }
```

此段代码可以在配套光盘中本章的 Test2 目录下找到。

运行这段代码, 应该首先加载 DriverA 驱动, 然后再加载 DriverB 驱动, 最后调用应用程序。如图 11-3 所示为 DriverA 和 DriverB 共同的输出结果, 注意输出的先后顺序。

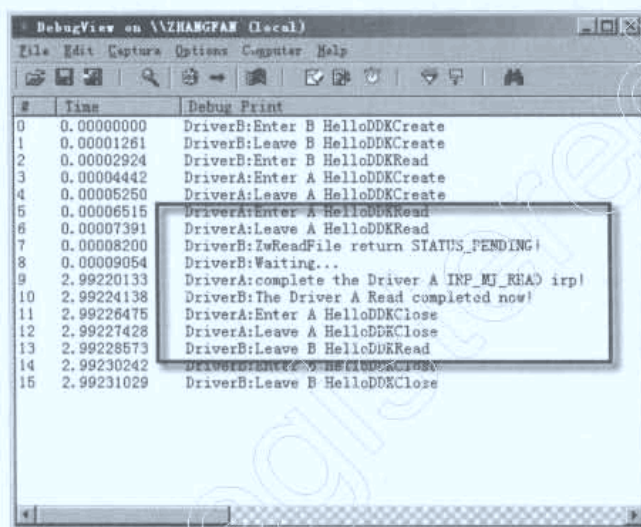


图 11-3 DebugView 输出 log

11.1.5 异步调用方法二

本节所介绍的异步调用方法, 和上一节的方法略有不同。上一节在异步读取时, 将一个事件的句柄传递给 ZwReadFile 内核函数, 这个事件可以用来通知读取操作何时完成。而本节介绍的方法不用将事件句柄传递给 ZwCreateFile 函数, 而是通过文件对象判断读取是否完毕。每打开一个设备, 都会伴随存在一个关联的文件对象 (FILE_OBJECT)。利用内核函数 ObReferenceObjectByHandle 可以获得和设备相关的文件对象指针。

当 IRP_MJ_READ 请求被结束后, 文件对象的子域 Event 会被设置, 因此用文件对象的 Event 子域可以当做同步点使用。本节就是利用文件对象的子域 Event, 即 FileObject->Event 作为同步点的, 以下是部分代码。

```
#001 #pragma PAGEDCODE
#002 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#003                        IN PIRP pIrp)
#004 {
```


Windows 驱动开发技术详解

```
#005     KdPrint(("DriverB:Enter B HelloDDKRead\n"));
#006     NTSTATUS ntStatus = STATUS_SUCCESS;
#007
#008     //构造设备名字串
#009     UNICODE_STRING DeviceName;
#010     RtlInitUnicodeString( &DeviceName, L"\\Device\\MyDDKDeviceA" );
#011
#012     //初始化 objectAttributes
#013     OBJECT_ATTRIBUTES objectAttributes;
#014     InitializeObjectAttributes(&objectAttributes,
#015                               &DeviceName,
#016                               OBJ_CASE_INSENSITIVE,
#017                               NULL,
#018                               NULL );
#019
#020     HANDLE hDevice;
#021     IO_STATUS_BLOCK status_block;
#022
#023     //异步打开设备
#024     ntStatus = ZwCreateFile(&hDevice,
#025                             FILE_READ_ATTRIBUTES, //没有设 SYNCHRONIZE
#026                             &objectAttributes,
#027                             &status_block,
#028                             NULL, FILE_ATTRIBUTE_NORMAL, FILE_SHARE_READ,
#029                             FILE_OPEN_IF, 0, NULL, 0);
#030     //将 32 位整数转换为 64 位整数
#031     LARGE_INTEGER offset = RtlConvertLongToLargeInteger(0);
#032     if (NT_SUCCESS(ntStatus))
#033     {
#034         ntStatus = ZwReadFile(hDevice, NULL, NULL, NULL, &status_block, NULL, 0,
&offset, NULL);
#035     }
#036     //判断 IRP 是否被挂起
#037     if (ntStatus==STATUS_PENDING)
#038     {
#039         KdPrint(("DriverB:ZwReadFile return STATUS_PENDING!\n"));
#040
#041         PFILE_OBJECT FileObject;
#042         //通过设备句柄找到文件对象指针
#043         ntStatus = ObReferenceObjectByHandle(hDevice, EVENT_MODIFY_STATE,
*ExEventObjectType,
#044                                             KernelMode, (PVOID*) &FileObject, NULL);
#045         if (NT_SUCCESS(ntStatus))
#046         {
#047             KdPrint(("DriverB:Waiting..."));
#048             //等待文件对象里的事件
#049             KeWaitForSingleObject(&FileObject->Event, Executive, KernelMode,
FALSE, NULL);
#050             KdPrint(("DriverB:Driver A Read IRP completed now!\n"));
#051             //减小文件对象引用计数
#052             ObDereferenceObject(FileObject);
#053         }
#054     }
#055     //关闭设备句柄
#056     ZwClose(hDevice);
#057
#058     ntStatus = STATUS_SUCCESS;
```

```

#059 // 将 IRP 状态设置为完成状态
#060 pIrp->IoStatus.Status = ntStatus;
#061 //设置 IRP 请求操作的字节数
#062 pIrp->IoStatus.Information = 0;
#063 //将 IRP 请求结束
#064 IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#065 KdPrint(("DriverB:Leave B HelloDDKRead\n"));
#066 return ntStatus;
#067 }

```

此段代码可以在配套光盘中本章的 Test3 目录下找到。

运行这段代码，首先加载 DriverA 的驱动，然后再加载 DriverB 的驱动，最后调用应用程序。由于是异步调用，因此 ZwReadFile 会在 DriverA 没有完成读 IRP 的时候就退出了。

如图 11-4 所示为 DriverA 和 DriverB 共同的输出结果，可以观察 DriverA 和 DriverB 输出的先后顺序。

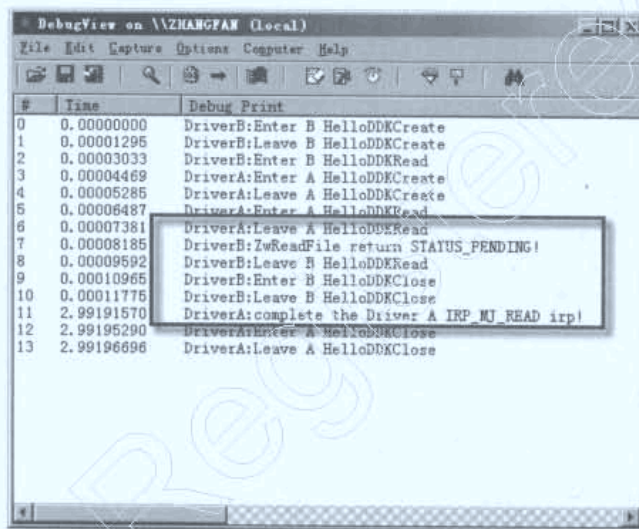


图 11-4 DebugView 输出 log

11.1.6 通过符号链接打开设备

前面介绍的例子都是通过设备名打开设备。但是有很多情况，使用者不容易知道具体的设备名，而只知道符号链接。例如“C:”代表第一个硬盘分区，而“C:”就是一个符号链接，它指向一个磁盘分区设备。尤其在 WDM 驱动程序中，通过符号链接打开设备是经常遇到的。

利用 ZwOpenSymbolicLinkObject 内核函数先得到符号链接的句柄，然后使用 ZwQuerySymbolicLinkObject 内核函数查找到设备名。通过设备名就可以方便地打开设备了。

下面的代码演示了如何通过符号链接打开设备。

Windows 驱动开发技术详解

```
#001 #pragma PAGEDCODE
#002 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#003                          IN PIRP pIrp)
#004 {
#005     KdPrint(("DriverB:Enter B HelloDDKRead\n"));
#006     NTSTATUS ntStatus = STATUS_SUCCESS;
#007     //构造符号链接字符串
#008     UNICODE_STRING DeviceSymbolicLinkName;
#009     RtlInitUnicodeString(&DeviceSymbolicLinkName, L"\\??\\HelloDDKA");
#010
#011     //初始化 objectAttributes
#012     OBJECT_ATTRIBUTES objectAttributes;
#013     InitializeObjectAttributes(&objectAttributes,
#014                               &DeviceSymbolicLinkName,
#015                               OBJ_CASE_INSENSITIVE|OBJ_KERNEL_HANDLE,
#016                               NULL,
#017                               NULL);
#018
#019     HANDLE hSymbolic;
#020     //得到符号链接句柄
#021     ntStatus = ZwOpenSymbolicLinkObject(&hSymbolic, FILE_ALL_ACCESS,
&objectAttributes);
#022     #define UNICODE_SIZE 50
#023     UNICODE_STRING LinkTarget;
#024     //申请 UNICODE_SIZE 大小的分页内存
#025     LinkTarget.Buffer = (PWSTR)ExAllocatePool(PagedPool, UNICODE_SIZE);
#026     //设定字符串长度
#027     LinkTarget.Length = 0;
#028     //设定字符串最大长度
#029     LinkTarget.MaximumLength = UNICODE_SIZE;
#030
#031     ULONG unicode_length;
#032     //通过符号链接得到设备名
#033     ntStatus = ZwQuerySymbolicLinkObject(hSymbolic, &LinkTarget, &unicode_length);
#034
#035     KdPrint(("DriverB:The device name is %wZ\n", &LinkTarget));
#036     //构造 objectAttributes
#037     InitializeObjectAttributes(&objectAttributes,
#038                               &LinkTarget,
#039                               OBJ_CASE_INSENSITIVE,
#040                               NULL,
#041                               NULL);
#042     HANDLE hDevice;
#043     IO_STATUS_BLOCK status_block;
#044     //打开设备
#045     ntStatus = ZwCreateFile(&hDevice,
#046                             FILE_READ_ATTRIBUTES|SYNCHRONIZE,
#047                             &objectAttributes,
#048                             &status_block,
#049                             NULL, FILE_ATTRIBUTE_NORMAL, FILE_SHARE_READ,
#050                             FILE_OPEN_IF, FILE_SYNCHRONOUS_IO_NONALERT, NULL, 0);
#051     //判断是否成功打开设备
#052     if (NT_SUCCESS(ntStatus))
#053     {
#054         //读取设备
#055         ZwReadFile(hDevice, NULL, NULL, NULL, &status_block, NULL, 0, NULL, NULL);
#056     }
```

```

#057 //关闭设备句柄
#058 ZwClose(hDevice);
#059 //关闭符号链接句柄
#060 ZwClose(hSymbolic);
#061 ExFreePool(LinkTarget.Buffer);
#062
#063 ntStatus = STATUS_SUCCESS;
#064 //设置 IRP 完成状态
#065 pIrp->IoStatus.Status = ntStatus;
#066 //设置 IRP 操作字节数
#067 pIrp->IoStatus.Information = 0;
#068 //结束 IRP 请求
#069 IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#070 KdPrint(("DriverB:Leave B HelloDDKRead\n"));
#071 return ntStatus;
#072 }

```

此段代码可以在配套光盘中本章的 Test4 目录下找到。

运行这段代码，首先加载 DriverA 的驱动，然后再加载 DriverB 的驱动，最后调用应用程序。如图 11-5 所示为输出结果，可以观察打开设备的符号链接、设备名等信息。

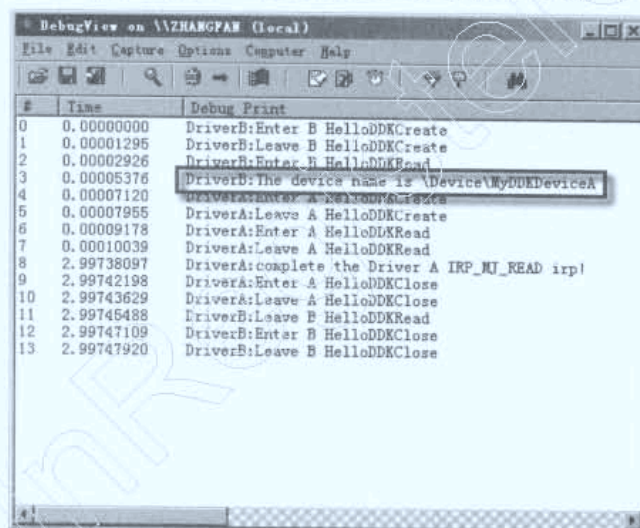


图 11-5 DebugView 输出 log

11.2 通过设备指针调用其他驱动程序

前面介绍了如何用 ZwCreateFile 内核函数打开设备，还介绍了如何用 ZwReadFile 内核函数读取设备。这些操作和应用程序中 CreateFile 和 ReadFile 函数的使用很类似。其实，CreateFile 和 ReadFile 这两个 Win32 API 函数内部分别调用了 ZwCreateFile 和 ZwReadFile 内核函数。

ZwReadFile 内核函数内部会创建 IRP_MJ_READ 类型的 IRP，然后将这个 IRP 传送到相应驱动的派遣函数中。本节介绍的驱动程序调用其他驱动程序的方法，不是借用

ZwCreateFile 和 ZwReadFile 等内核函数，而是“手动”构造各个 IRP，然后将 IRP 传递到相应驱动程序的派遣函数里。

通过本节的学习，读者可以从某种程度上了解 ZwCreateFile 和 ZwReadFile 内核函数是如何实现的。

11.2.1 用 IoGetDeviceObjectPointer 获得设备指针

每个内核中的句柄都会和一个内核对象的指针联系起来。例如，进程对象的句柄和进程对象的指针关联，线程对象的句柄和线程对象的指针关联，内核事件句柄和内核对象指针关联。

ZwCreateFile 内核函数可以通过设备名打开设备句柄，这个设备句柄和一个文件对象的指针关联。IoGetDeviceObjectPointer 内核函数可以通过设备名获得文件对象指针，而不是获得设备句柄，其函数声明如下：

```
NTSTATUS
IoGetDeviceObjectPointer(
    IN PUNICODE_STRING ObjectName,
    IN ACCESS_MASK DesiredAccess,
    OUT PFILE_OBJECT *FileObject,
    OUT PDEVICE_OBJECT *DeviceObject
);
```

- 第一个参数 ObjectName：设备名，用 UNICODE 字符串表示。
- 第二个参数 DesiredAccess：以什么样的权限得到设备指针。
- 第三个参数 FileObject：同时会返回一个和设备相关的文件对象指针。
- 第四个参数 DeviceObject：返回的设备对象指针。
- 返回值：是否成功得到设备指针。

Windows 内核会为每一个对象指针保存一个“引用计数”，当对象被创建的时候引用计数为 1。如果想引用这个对象时，计数就会加 1。如果删除对象时，Windows 先将引用计数减 1，如果引用计数不是 0，系统不会删除对象。只有引用计数被减到 0 时，系统才会真正删除对象。

当调用 IoGetDeviceObjectPointer 内核函数后，设备对象的引用计数就会加 1。当用完这个设备对象后，应该调用 ObDereferenceObject 内核函数，使其引用计数减 1。

```
ObDereferenceObject( FileObject );
```

当第一次调用 IoGetDeviceObjectPointer 内核函数时，会根据设备名打开设备，这时文件对象指针的引用计数为 1。此后如果再次调用 IoGetDeviceObjectPointer 打开设备，就不是真正地打开设备了，而只是将引用计数加 1。打开设备时，系统会创建一个 IRP_MJ_CREATE 类型的 IRP，并将这个 IRP 传递到驱动程序的派遣函数中。

每次调用 ObDereferenceObject 内核函数都会将“引用计数”减 1，如果减至 0 就会关

闭设备。关闭设备时，系统会创建一个 IRP_MJ_CLOSE 类型的 IRP，并将其传递到相应驱动的派遣函数中。

从上述内容可以看出，IoGetDeviceObjectPointer 和 ObDereferenceObject 内核函数完全可以代替 ZwCreateFile 和 ZwCloseFile 内核函数。另外，这种方法还能获得设备对象指针、关联的文件对象指针。

11.2.2 创建 IRP 传递给驱动的派遣函数

本节将介绍如何自己手动创建 IRP，并将其传递给相应的驱动程序。这样的好处是比 ZwReadFile 内核函数灵活。ZwReadFile 内核函数是针对设备句柄操作的，而传递 IRP 是通过设备对象的指针操作。

可以通过 IoBuildSynchronousFsdRequest 和 IoBuildAsynchronousFsdRequest 两个内核函数创建 IRP，它们分别用来创建同步类型的 IRP 和创建异步类型的 IRP。这两个内核函数可以创建 IRP_MJ_PNP、IRP_MJ_READ、IRP_MJ_WRITE、IRP_MJ_FLUSH_BUFFERS 和 IRP_MJ_SHUTDOWN 类型的 IRP。

可以通过 IoBuildDeviceIoControlRequest 内核函数创建 IRP_MJ_INTERNAL_DEVICE_CONTROL 和 IRP_MJ_DEVICE_CONTROL 两个类型的 IRP，这个内核函数只能创建同步类型的 IRP。

另外，还可以使用 IoAllocateIrp 内核函数，它可以创建任意类型的 IRP。IoBuildSynchronousFsdRequest、IoBuildAsynchronousFsdRequest、IoBuildDeviceIoControlRequest 这三个内核函数是属于靠近上层的内核函数。而 IoAllocateIrp 是比较底层的内核函数，以上三个内核函数都是通过调用 IoAllocateIrp 实现的。

创建完 IRP 后，还要构造 IRP 的 I/O 堆栈，每层 I/O 堆栈对应一个设备对象。由于示例程序 DriverA 是单层驱动程序，所以只需要构造 IRP 的第一层 I/O 堆栈。

最后是通过 IoCallDriver 内核函数调用相应的驱动。IoCallDriver 内核函数会根据 IRP 的类型，找到相应的派遣函数。

总结一下，手动创建 IRP 有以下几个步骤。

① 先得到设备的指针。一种方法是用 IoGetDeviceObjectPointer 内核函数得到设备对象指针。另外一种方法是用 ZwCreateFile 内核函数先得到设备句柄，然后调用 ObReferenceObjectByPointer 内核函数通过设备句柄得到设备对象指针。

② 手动创建 IRP，有 4 个内核函数可以选择，它们分别是 IoBuildSynchronousFsdRequest、IoBuildAsynchronousFsdRequest、IoBuildDeviceIoControlRequest 和 IoAllocateIrp。其中 IoAllocateIrp 内核函数是最灵活的，使用也最复杂。

③ 构造 IRP 的 I/O 堆栈。

④ 调用 IoCallDriver 内核函数，其内部会调用设备对象的派遣函数。

11.2.3 用 IoBuildSynchronousFsdRequest 创建 IRP

本节主要介绍如何使用 IoBuildSynchronousFsdRequest 内核函数创建同步类型的 IRP，其函数声明如下：

```
PIRP
IoBuildSynchronousFsdRequest(
    IN ULONG MajorFunction,
    IN PDEVICE_OBJECT DeviceObject,
    IN OUT PVOID Buffer OPTIONAL,
    IN ULONG Length OPTIONAL,
    IN PLARGE_INTEGER StartingOffset OPTIONAL,
    IN PKEVENT Event,
    OUT PIO_STATUS_BLOCK IoStatusBlock
);
```

- 第一个参数 MajorFunction: 这个参数是创建的 IRP 的主类型，IoBuildSynchronousFsdRequest 函数只支持 IRP_MJ_PNP、IRP_MJ_READ、IRP_MJ_WRITE、IRP_MJ_FLUSH_BUFFERS 和 IRP_MJ_SHUTDOWN。
- 第二个参数 DeviceObject: 这个参数是设备对象指针。IRP 将会传递给这个设备对象。
- 第三个参数 Buffer: 对于 IRP_MJ_READ 和 IRP_MJ_WRITE，Buffer 指的是输入和输出缓冲区。
- 第四个参数 Length: 这个参数是缓冲区的大小。
- 第五个参数 StartingOffset: 这个参数是偏移量。
- 第六个参数 Event: 这个参数是同步事件，这是创建同步类型 IRP 的关键，后面会有详细介绍。
- 第七个参数 IoStatusBlock: 这个参数是操作状态。
- 返回值: 返回一个 IRP 指针。

使用 IoBuildSynchronousFsdRequest 内核函数创建同步类型 IRP，关键在于第六个参数 Event。在调用 IoBuildSynchronousFsdRequest 之前，需要准备一个事件。这个事件会和 IRP 请求进行关联，当 IRP 请求被结束时该事件被触发。IoBuildSynchronousFsdRequest 和 IoBuildAsynchronousFsdRequest 内核函数之间的区别就是是否提供事件。

下面的代码演示了如何使用 IoBuildSynchronousFsdRequest 内核函数创建同步类型 IRP。

```
#001 #pragma PAGEDCODE
#002 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#003                          IN PIRP pIrp)
#004 {
#005     KdPrint(("DriverB:Enter B HelloDDKRead\n"));
#006     NTSTATUS ntStatus = STATUS_SUCCESS;
#007     //构造设备名字符串
#008     UNICODE_STRING DeviceName;
```

```

#009   RtlInitUnicodeString( &DeviceName, L"\\Device\\MyDDKDeviceA" );
#010
#011   PDEVICE_OBJECT DeviceObject = NULL;
#012   PFILE_OBJECT FileObject = NULL;
#013   //得到设备对象句柄, 计数器加 1
#014   //如果是第一次调用 IoGetDeviceObjectPointer, 会打开设备, 相当于调用 ZwCreateFile
#015   ntStatus = IoGetDeviceObjectPointer(&DeviceName, FILE_ALL_ACCESS,
&FileObject, &DeviceObject);
#016
#017   KdPrint(("DriverB:FileObject:%x\n", FileObject));
#018   KdPrint(("DriverB:DeviceObject:%x\n", DeviceObject));
#019
#020   //判断是否成功打开设备
#021   if (!NT_SUCCESS(ntStatus))
#022   {
#023       KdPrint(("DriverB:IoGetDeviceObjectPointer() 0x%x\n", ntStatus));
#024
#025       ntStatus = STATUS_UNSUCCESSFUL;
#026       //设置 IRP 的完成状态
#027       pIrp->IoStatus.Status = ntStatus;
#028       //设置 IRP 操作的字节数
#029       pIrp->IoStatus.Information = 0;
#030       //将 IRP 请求结束
#031       IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#032       KdPrint(("DriverB:Leave B HelloDDKRead\n"));
#033
#034       return ntStatus;
#035   }
#036
#037   KEVENT event;
#038   //初始化一个同步事件
#039   KeInitializeEvent(&event, NotificationEvent, FALSE);
#040   IO_STATUS_BLOCK status_block;
#041   //将 32 位整数转化为 64 位整数
#042   LARGE_INTEGER offset = RtlConvertLongToLargeInteger(0);
#043
#044   //创建同步 IRP
#045   PIRP pNewIrp = IoBuildSynchronousFsdRequest(IRP_MJ_READ,
#046                                               DeviceObject,
#047                                               NULL, 0,
#048                                               &offset, &event,
&status_block);
#049   KdPrint(("DriverB:pNewIrp:%x\n", pNewIrp));
#050
#051   //得到下一层的 I/O 堆栈
#052   PIO_STACK_LOCATION stack = IoGetNextIrpStackLocation(pNewIrp);
#053   //设置 I/O 堆栈的文件对象指针
#054   stack->FileObject = FileObject;
#055
#056   //调用 DriverA, 会一直调用到 DriverA 的派遣函数
#057   NTSTATUS status = IoCallDriver(DeviceObject, pNewIrp);
#058   //判断操作是否被挂起
#059   if (status == STATUS_PENDING) {
#060
#061       //如果 DriverA 的派遣函数没有完成 IRP, 则等待 IRP 完成
#062       status = KeWaitForSingleObject(
#063           &event,

```



```

#064         Executive,
#065         KernelMode,
#066         FALSE, // Not alertable
#067         NULL);
#068     status = status_block.Status;
#069 }
#070
#071 //将引用计数减 1, 如果此时计数器减为 0,
#072 //则将关闭设备, 相当于调用 ZwClose
#073 ObDereferenceObject( FileObject );
#074
#075 ntStatus = STATUS_SUCCESS;
#076 //设置 IRP 的完成状态
#077 pIrp->IoStatus.Status = ntStatus;
#078 //设置 IRP 的操作字节数
#079 pIrp->IoStatus.Information = 0; //
#080 //将 IRP 请求结束
#081 IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#082 KdPrint(("DriverB:Leave B HelloDDKRead\n"));
#083 return ntStatus;
#084 }

```

此段代码可以在配套光盘中本章的 Test5 目录下找到。

运行这段代码, 首先加载 DriverA 的驱动, 然后再加载 DriverB 的驱动, 最后调用应用程序。由于是异步调用, 因此 IoCallDriver 返回来的是 STATUS_PENDING, 并且这时候 IRP 并没有真正完成。3s 后 DriverA 将 IRP 真正完成。这时候事件被触发, 程序继续运行。

如图 11-6 所示为 DriverA 和 DriverB 共同的输出结果, 注意输出的先后顺序。

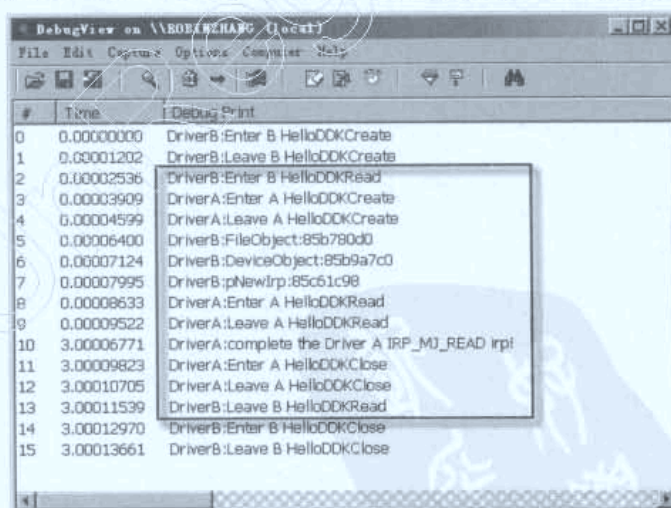


图 11-6 DebugView 输出 log

11.2.4 用 IoBuildAsynchronousFsdRequest 创建 IRP

本节主要介绍如何使用 IoBuildAsynchronousFsdRequest 内核函数创建异步类型的

IRP, 这个内核函数比 `IoBuildSynchronousFsdRequest` 内核函数少一个事件参数。

对于用 `IoBuildAsynchronousFsdRequest` 内核函数创建的 IRP, 当 IRP 请求被结束后, 操作系统不会通过事件进行通知。不过可以通过 IRP 的 `UserEvent` 子域来通知 IRP 请求的结束。

当执行 `IoCompleteRequest` 内核函数时, 操作系统会检查 IRP 的 `UserEvent` 子域是否为空。如果 `UserEvent` 子域非空, 则它代表一个事件指针, 这时候 `IoCompleteRequest` 会设置这个事件。

调用 `IoBuildAsynchronousFsdRequest` 内核函数创建 IRP 后, 如果希望进行同步处理, 也就是希望得到 IRP 被结束的通知, 则只需要设置 IRP 的子域 `UserEvent`。

以下的代码演示了如何使用 `IoBuildAsynchronousFsdRequest` 内核函数:

```
#001  #pragma PAGEDCODE
#002  NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#003                          IN PIRP pIrp)
#004  {
#005      KdPrint(("DriverB:Enter B HelloDDKRead\n"));
#006      NTSTATUS ntStatus = STATUS_SUCCESS;
#007      //构造设备名字符串
#008      UNICODE_STRING DeviceName;
#009      RtlInitUnicodeString(&DeviceName, L"\\Device\\MyDDKDeviceA");
#010
#011      PDEVICE_OBJECT DeviceObject = NULL;
#012      PFILE_OBJECT FileObject = NULL;
#013      //得到设备对象指针
#014      ntStatus = IoGetDeviceObjectPointer(&DeviceName, FILE_ALL_ACCESS,
&FileObject, &DeviceObject);
#015
#016      KdPrint(("DriverB:FileObject:%x\n", FileObject));
#017      KdPrint(("DriverB:DeviceObject:%x\n", DeviceObject));
#018      //判断是否成功打开设备
#019      if (!NT_SUCCESS(ntStatus))
#020      {
#021          KdPrint(("DriverB:IoGetDeviceObjectPointer() 0x%x\n", ntStatus));
#022
#023          ntStatus = STATUS_UNSUCCESSFUL;
#024          //设置 IRP 完成状态
#025          pIrp->IoStatus.Status = ntStatus;
#026          //设置 IRP 操作字节数
#027          pIrp->IoStatus.Information = 0;
#028          //结束 IRP 请求
#029          IoCompleteRequest(pIrp, IO_NO_INCREMENT);
#030          KdPrint(("DriverB:Leave B HelloDDKRead\n"));
#031
#032          return ntStatus;
#033      }
#034
#035      KEVENT event;
#036      //初始化同步事件
#037      KeInitializeEvent(&event, NotificationEvent, FALSE);
#038      IO_STATUS_BLOCK status_block;
#039      //将 32 位整数转化为 64 位整数
```



```

#040     LARGE_INTEGER offsert = RtlConvertLongToLargeInteger(0);
#041
#042     //创建异步 IRP
#043     PIRP pNewIrp = IoBuildAsynchronousFsdRequest(IRP_MJ_READ,
#044                                                     DeviceObject,
#045                                                     NULL, 0,
#046                                                     &offsert, &status_block);
#047     KdPrint(("pNewIrp->UserEvent :%x\n", pNewIrp->UserEvent));
#048     //设置 pNewIrp->UserEvent, 这样在 IRP 完成后可以通知该事件
#049     pNewIrp->UserEvent = &event;
#050
#051     KdPrint(("DriverB:pNewIrp:%x\n", pNewIrp));
#052     //得到下一层的 I/O 堆栈
#053     PIO_STACK_LOCATION stack = IoGetNextIrpStackLocation(pNewIrp);
#054     //设置 I/O 堆栈的文件句柄指针
#055     stack->FileObject = FileObject;
#056     //调用 DriverA 驱动程序
#057     NTSTATUS status = IoCallDriver(DeviceObject, pNewIrp);
#058     //判断操作是否被挂起
#059     if (status == STATUS_PENDING) {
#060         //等待 IRP 被结束
#061         status = KeWaitForSingleObject(
#062             &event,
#063             Executive,
#064             KernelMode,
#065             FALSE, // Not alertable
#066             NULL);
#067         status = status_block.Status;
#068     }
#069     //关闭文件句柄
#070     ZwClose(FileObject);
#071     //关闭设备句柄
#072     ObDereferenceObject( FileObject );
#073
#074     ntStatus = STATUS_SUCCESS;
#075     //设置 IRP 完成状态
#076     pIrp->IoStatus.Status = ntStatus;
#077     //设置 IRP 操作字节数
#078     pIrp->IoStatus.Information = 0;
#079     //将 IRP 结束
#080     IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#081     KdPrint(("DriverB:Leave B HelloDDKRead\n"));
#082     return ntStatus;
#083 }

```

此段代码可以在配套光盘中本章的 Test6 目录下找到。

运行这段代码，首先加载 DriverA 的驱动，然后再加载 DriverB 的驱动，最后调用应用程序。由于是异步调用，因此 IoCallDriver 返回来的是 STATUS_PENDING，并且这时候 IRP 并没有真正完成。3s 后 DriverA 将 IRP 真正完成。这时候事件被触发，程序继续运行。

如图 11-7 所示为 DriverA 和 DriverB 共同的输出结果，注意输出的先后顺序。

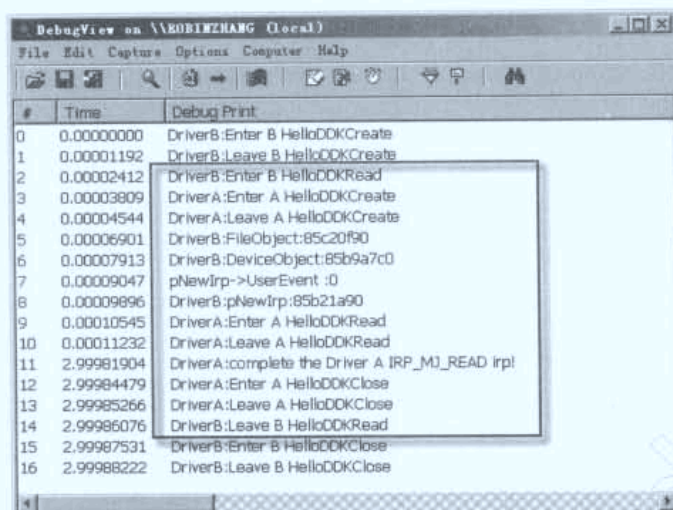


图 11-7 DebugView 输出 log

11.2.5 用 IoAllocateIrp 创建 IRP

有四个内核函数可以用来创建 IRP，分别是 IoBuildSynchronousFsdRequest、IoBuildAsynchronousFsdRequest、IoBuildDeviceIoControlRequest 和 IoAllocateIrp。前三个内核函数都是直接或者间接地调用 IoAllocateIrp 内核函数。

另外，ZwCreateFile、ZwReadFile、ZwWriteFile 等内核函数也都是间接地调用了 IoAllocateIrp。还有 CreateFile、ReadFile、DeleteFile 等 Win32 API 也都是间接地调用了 IoAllocateIrp。

所有对设备的操作都会转化为一个 IRP，IRP 会在驱动程序中被处理。IRP 请求被结束，就代表着对设备操作的结束，IRP 的完成状态就是操作的完成状态。而所有的 IRP 最终都是由 IoAllocateIrp 内核函数创建的。

整个 Windows 是个异步的框架，IRP 请求可以被异步地结束。如果对异步请求用同步事件进行同步时，异步请求就变成了同步请求。可以说同步请求是异步请求的一个特例，这如同静止状态是运动状态的一个特例一样。驱动程序开发者大部分的任务就是创建和处理 IRP 请求。

IoAllocateIrp 内核函数的声明如下：

```
PIRP
IoAllocateIrp(
    IN CCHAR StackSize,
    IN BOOLEAN ChargeQuota
);
```

- 第一个参数 StackSize：这个参数设置堆栈大小。
- 第二个参数 ChargeQuota：这个参数指示是否使用磁盘配额。

➤ 返回值：返回创建的 IRP。

可以看出 IoAllocateIrp 比 IoBuildSynchronousFsdRequest 等内核函数少了很多参数，例如，输入缓冲区、输出缓冲区、同步事件等参数。这些参数需要在 IoAllocateIrp 创建完 IRP 后，程序员自己“手动”填写完成。而 IoBuildSynchronousFsdRequest 等内核函数在内部就替程序员做好了。当然，“手动”填写这些 IRP 的子域可以很灵活地操作 IRP，但这需要对 IRP 结构有比较熟悉的理解。

IoBuildSynchronousFsdRequest、IoBuildAsynchronousFsdRequest、IoBuildDeviceIoControlRequest 内核函数在创建完 IRP 后，不需要程序员负责删除 IRP，操作系统会自动删除 IRP。而用 IoAllocateIrp 内核函数创建 IRP 时，需要程序员调用 IoFreeIrp 内核函数删除 IRP 对象。

下面的代码演示了如何使用 IoAllocateIrp 内核函数创建 IRP。

```
#001 #pragma PAGEDCODE
#002 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#003                        IN PIRP pIrp)
#004 {
#005     KdPrint(("DriverB:Enter B HelloDDKRead\n"));
#006     NTSTATUS ntStatus = STATUS_SUCCESS;
#007     //构造设备名字字符串
#008     UNICODE_STRING DeviceName;
#009     RtlInitUnicodeString(&DeviceName, L"\\Device\\MyDDKDeviceA");
#010
#011     PDEVICE_OBJECT DeviceObject = NULL;
#012     PFILE_OBJECT FileObject = NULL;
#013     //得到设备对象指针
#014     ntStatus = IoGetDeviceObjectPointer(&DeviceName, FILE_ALL_ACCESS,
&FileObject, &DeviceObject);
#015
#016     KdPrint(("DriverB:FileObject:%x\n", FileObject));
#017     KdPrint(("DriverB:DeviceObject:%x\n", DeviceObject));
#018     //判断设备是否被成功打开
#019     if (!NT_SUCCESS(ntStatus))
#020     {
#021         KdPrint(("DriverB:IoGetDeviceObjectPointer() 0x%x\n", ntStatus));
#022         ntStatus = STATUS_UNSUCCESSFUL;
#023         //设置 IRP 完成状态
#024         pIrp->IoStatus.Status = ntStatus;
#025         //设置 IRP 操作字节数
#026         pIrp->IoStatus.Information = 0;
#027         //结束 IRP 请求
#028         IoCompleteRequest(pIrp, IO_NO_INCREMENT);
#029         KdPrint(("DriverB:Leave B HelloDDKRead\n"));
#030
#031         return ntStatus;
#032     }
#033
#034     KEVENT event;
#035     //初始化同步事件
#036     KeInitializeEvent(&event, NotificationEvent, FALSE);
#037     //用 IoAllocateIrp 创建 IRP
#038     PIRP pNewIrp = IoAllocateIrp(DeviceObject->StackSize, FALSE);
```

```

#039     KdPrint(("pNewIrp->UserEvent :%x\n",pNewIrp->UserEvent));
#040     pNewIrp->UserEvent = &event;
#041
#042     IO_STATUS_BLOCK status_block;
#043     //设置新 IRP 的 UserIosb
#044     pNewIrp->UserIosb = &status_block;
#045     //设置新 IRP 相关的线程号
#046     pNewIrp->Tail.Overlay.Thread = PsGetCurrentThread();
#047
#048     //因为 DriverA 是 BUFFER IO 设备
#049     pNewIrp->AssociatedIrp.SystemBuffer = NULL;
#050
#051     KdPrint(("DriverB:pNewIrp:%x\n",pNewIrp));
#052     //取得下一层 I/O 堆栈
#053     PIO_STACK_LOCATION stack = IoGetNextIrpStackLocation(pNewIrp);
#054     //设置 I/O 堆栈的主 IRP 号
#055     stack->MajorFunction = IRP_MJ_READ;
#056     //设置 I/O 堆栈的子 IRP 号
#057     stack->MinorFunction=IRP_MN_NORMAL;//0
#058     //设置 I/O 堆栈的文件对象指针
#059     stack->FileObject = FileObject;
#060
#061     //调用 DriverA 驱动
#062     NTSTATUS status = IoCallDriver(DeviceObject,pNewIrp);
#063     //判断操作是否被挂起
#064     if (status == STATUS_PENDING) {
#065         //等待同步对象
#066         status = KeWaitForSingleObject(
#067             &event,
#068             Executive,
#069             KernelMode,
#070             FALSE, // Not alertable
#071             NULL);
#072         KdPrint(("STATUS_PENDING\n"));
#073     }
#074     //将文件对象指针的引用计数减 1
#075     ObDereferenceObject( FileObject );
#076
#077     ntStatus = STATUS_SUCCESS;
#078     // 设置 IRP 的完成状态
#079     pIrp->IoStatus.Status = ntStatus;
#080     //设置 IRP 的操作字节数
#081     pIrp->IoStatus.Information = 0;
#082     //结束 IRP 请求
#083     IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#084     KdPrint(("DriverB:Leave B HelloDDKRead\n"));
#085     return ntStatus;
#086 }

```

此段代码可以在配套光盘中本章的 Test7 目录下找到。

运行这段代码，首先加载 DriverA 的驱动，然后再加载 DriverB 的驱动，最后调用应用程序。由于是异步调用，因此 IoCallDriver 返回来的是 STATUS_PENDING，并且这时候 IRP 并没有真正完成。3s 后 DriverA 将 IRP 真正完成。这时候事件被触发，程序继续运行。

如图 11-8 所示为 DriverA 和 DriverB 共同的输出结果，注意输出的先后顺序。

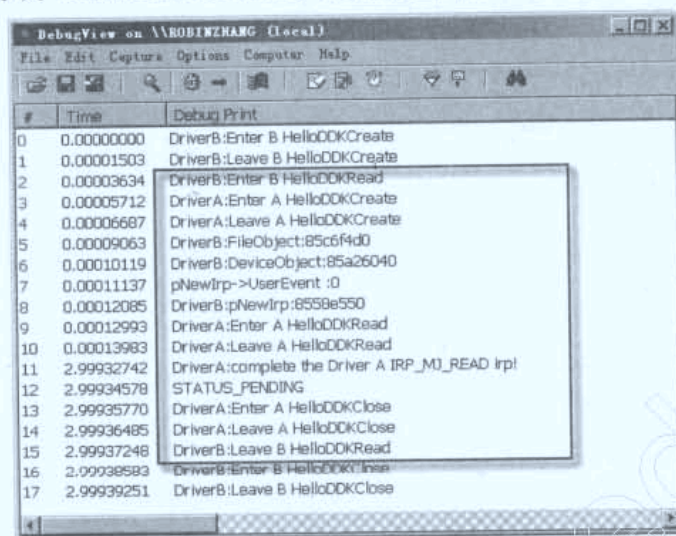


图 11-8 DebugView 输出 log

11.3 其他方法获得设备指针

前面几节中介绍了如何获得设备句柄、获得设备指针的方法，本节将较深入地介绍 Windows 内部如何获得设备对象指针。

11.3.1 用 ObReferenceObjectByName 获得设备指针

本节介绍一个 DDK 未公开的内核函数 ObReferenceObjectByName。之所以称其为未公开的内核函数，是因为在 DDK 的所有.h 文件中都没有关于这个函数的声明，但是通过 Dependency 工具查看，可以知道 ntoskrnl.exe 导出 ObReferenceObjectByName 函数，如图 11-9 所示。

由于 ObReferenceObjectByName 函数没有在 DDK 的.h 文件中声明，因此程序员需要自己“手动”声明这个函数。同时，ObReferenceObjectByName 内核函数用到的常量也需要程序员自己声明。需要注意的是，如果使用 C++编写驱动程序，还需要用 extern "C"来修饰。

下面的代码演示了如何在 C++程序中声明 ObReferenceObjectByName 内核函数。

```
#ifdef __cplusplus
extern "C"
{
#endif
#include <NTDDK.h>

NTKERNELAPI
```

```

NTSTATUS
ObReferenceObjectByName(
    IN PUNICODE_STRING ObjectName,
    IN ULONG Attributes,
    IN PACCESS_STATE PassedAccessState OPTIONAL,
    IN ACCESS_MASK DesiredAccess OPTIONAL,
    IN POBJECT_TYPE ObjectType,
    IN KPROCESSOR_MODE AccessMode,
    IN OUT PVOID ParseContext OPTIONAL,
    OUT PVOID *Object
);
extern POBJECT_TYPE IoDeviceObjectType;
#ifdef __cplusplus
}
#endif

```

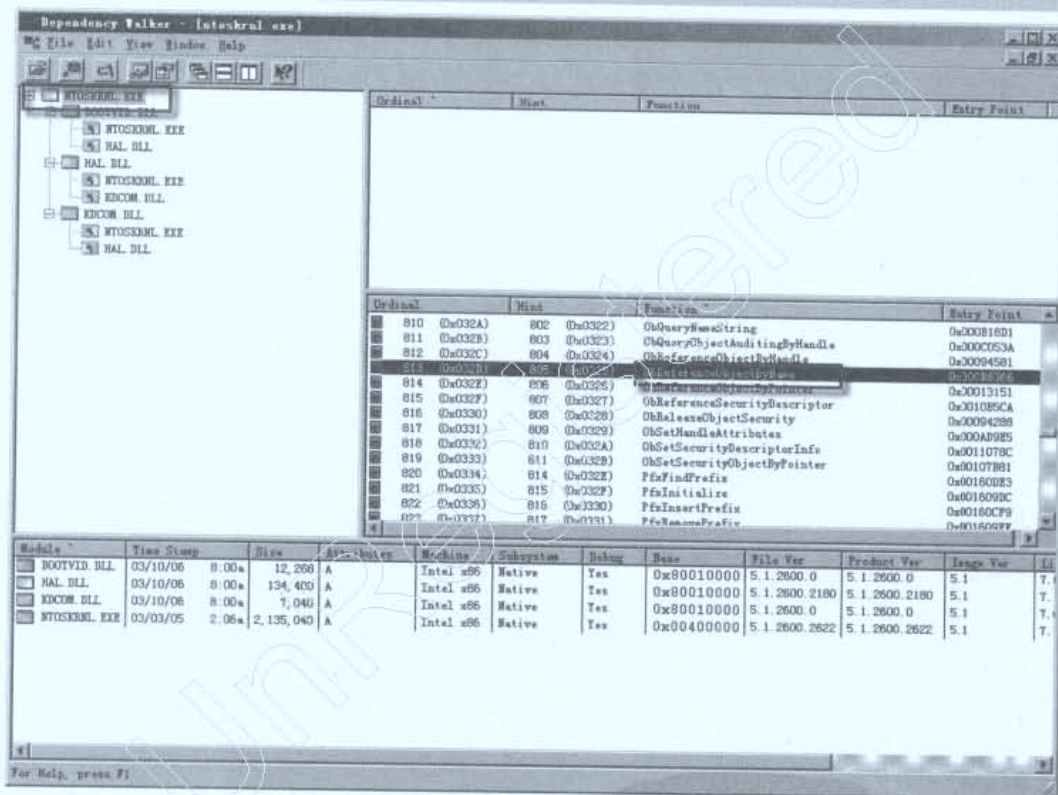


图 11-9 用 Depends 查看 ntoskrnl.exe 输出函数

- 第一个参数 ObjectName: 这个参数指定打开设备的设备名, 用 UNICODE 字符串表示。
- 第二个参数 Attributes: 这个属性一般设置为 OBJ_CASE_INSENSITIVE。
- 第三个参数 PassedAccessState: 这个参数很少被用到, 一般设为 NULL。
- 第四个参数 DesiredAccess: 这个参数是获取该设备的权限, 一般设为 FILE_ALL_ACCESS。
- 第五个参数 ObjectType: ObReferenceObjectByName 函数可以得到各类对象的指

针，如果要得到设备对象指针，需要指定这个参数为 `IoDeviceObjectType`。

- 第七个参数 `AccessMode`：如果要在内核中访问该指针，需要设定此参数为 `KernelMode`。
- 第八个参数 `ParseContext`：这个参数很少用到，一般设为 `NULL`。
- 第九个参数 `Object`：返回的内核对象的指针。
- 返回值：返回是否成功得到设备对象的指针。

`ObReferenceObjectByName` 和 `IoGetDeviceObjectPointer` 内核函数虽然都是获取设备对象指针，但是却有一些不同。`ObReferenceObjectByName` 内核函数仅仅是通过名字得到对象的指针而已，除了得到设备对象指针外，也可以得到别的内核对象的指针，比如内核事件、互斥体对象等。`ObReferenceObjectByName` 内核函数会增加对设备对象的引用计数，因此在需要使用设备对象完毕后，需要调用 `ObDereferenceObject` 内核函数将引用计数减 1。

而 `IoGetDeviceObjectPointer` 内核函数只能得到设备对象指针。在 `IoGetDeviceObjectPointer` 内核函数内部，得到设备对象指针后还会打开设备对象，也就是相当于向该设备对象发送 `IRP_MJ_CREATE` 的 `IRP`。同时，`IoGetDeviceObjectPointer` 内核函数还获得了与设备对象相关的文件对象句柄。在使用完设备对象后，需要对设备相关的文件对象调用 `ObDereferenceObject` 内核函数，这样相当于向设备发送 `IRP_MJ_CLOSE`。

下面的代码演示了如何使用 `ObReferenceObjectByName` 内核函数。

```
#001  #pragma PAGEDCODE
#002  NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#003                          IN PIRP pIrp)
#004  {
#005      KdPrint(("DriverB:Enter B HelloDDKRead\n"));
#006      NTSTATUS ntStatus = STATUS_SUCCESS;
#007      //构造设备名字符串
#008      UNICODE_STRING DeviceName;
#009      RtlInitUnicodeString(&DeviceName, L"\\??\\HelloDDKA");
#010
#011      PDEVICE_OBJECT DeviceObject = NULL;
#012      PFILE_OBJECT FileObject = NULL;
#013      //调用 ObReferenceObjectByName 内核函数打开设备
#014      ntStatus = ObReferenceObjectByName(&DeviceName, OBJ_CASE_INSENSITIVE,
NULL, FILE_ALL_ACCESS, IoDeviceObjectType, KernelMode, NULL, (PVOID*)&DeviceObject);
#015
#016      KdPrint(("ntStatus %x\n", ntStatus));
#017      KdPrint(("DeviceObject %x\n", DeviceObject));
#018      ntStatus = STATUS_SUCCESS;
#019      //设置 IRP 的完成状态
#020      pIrp->IoStatus.Status = ntStatus;
#021      //设置 IRP 的操作字节数
#022      pIrp->IoStatus.Information = 0;
#023      //结束 IRP 请求
#024      IoCompleteRequest(pIrp, IO_NO_INCREMENT);
#025      KdPrint(("DriverB:Leave B HelloDDKRead\n"));
#026      return ntStatus;
#027  }
```

此段代码可以在配套光盘中本章的 `Test7` 目录下找到。

11.3.2 剖析 IoGetDeviceObjectPointer

本节将介绍一种方法模拟 IoGetDeviceObjectPointer 内核函数的实现，可以通过以下几个步骤：

- ① 用 InitializeObjectAttributes 内核函数构造 OBJECT_ATTRIBUTES 结构体。
- ② 用 ZwOpenFile 内核函数打开设备对象的句柄。
- ③ 用 ObReferenceObjectByHandle 内核函数将设备对象的句柄转化成设备对象相关的文件对象句柄。
- ④ 用 IoGetBaseFileSystemDeviceObject 内核函数从设备相关文件对象指针得到设备对象指针。

下面的代码演示了如何模拟 IoGetDeviceObjectPointer 内核函数。

```
#001 //模拟 IoGetDeviceObjectPointer 实现
#002 NTSTATUS
#003 MyIoGetDeviceObjectPointer(
#004     IN PUNICODE_STRING ObjectName,
#005     IN ACCESS_MASK DesiredAccess,
#006     OUT PFILE_OBJECT *FileObject,
#007     OUT PDEVICE_OBJECT *DeviceObject
#008 )
#009 {
#010     PFILE_OBJECT fileObject;
#011     OBJECT_ATTRIBUTES objectAttributes;
#012     HANDLE fileHandle;
#013     IO_STATUS_BLOCK ioStatus;
#014     NTSTATUS status;
#015
#016     //设置要打开的设备的设备名
#017     InitializeObjectAttributes( &objectAttributes,
#018                                 ObjectName,
#019                                 OBJ_KERNEL_HANDLE,
#020                                 (HANDLE) NULL,
#021                                 (PSECURITY_DESCRIPTOR) NULL );
#022
#023     //ZwOpenFile 打开设备
#024     status = ZwOpenFile( &fileHandle,
#025                         DesiredAccess,
#026                         &objectAttributes,
#027                         &ioStatus,
#028                         0,
#029                         FILE_NON_DIRECTORY_FILE );
#030
#031     //判断操作是否被挂起
#032     if (NT_SUCCESS( status ))
#033     {
#034         //得到文件对象指针
#035         status = ObReferenceObjectByHandle( fileHandle,
#036                                             0,
#037                                             *IoFileObjectType,
#038                                             KernelMode,
#039                                             (PVOID *) &fileObject,
```



```
#040                                     NULL );
#041      //判断操作是否成功
#042      if (NT_SUCCESS( status ))
#043      {
#044          *FileObject = fileObject;
#045          //得到设备对象指针
#046          *DeviceObject = IoGetBaseFileSystemDeviceObject( fileObject );
#047      }
#048      //关闭设备句柄
#049      ZwClose( fileHandle );
#050  }
#051      return status;
#052 }
```

此段代码可以在配套光盘本章的 Test7 目录下找到。

11.4 小结

本章主要介绍了如何在驱动程序中调用其他驱动程序。最简单的方法就是使用 ZwCreateFile(NtZreateFile)、ZwOpenFile(NtOpenFile)、ZwReadFile(NtReadFile)、ZwWriteFile(NtWriteFile)等内核函数操作设备。这种方法类似于应用程序调用驱动程序的方法。

另外一种方法就是“手动”创建 IRP，然后将 IRP 传递给相应的设备对象。这种方法比上一种方法复杂，但灵活性却很高。创建 IRP 的内核函数有 IoBuildSynchronousFsdRequest、IoBuildAsynchronousFsdRequest、IoBuildDeviceIoControlRequest 和 IoAllocateIrp。其中 IoAllocateIrp 内核函数使用最为灵活。

第 12 章 分层驱动程序

本章介绍分层驱动程序的概念。分层驱动将一个驱动程序分解成若干个小驱动程序，每个小驱动程序完成一个简单的任务，这样有助于将复杂操作分解成多个简单操作。WDM 驱动程序就是在这个基础上发展而来的。

12.1 分层驱动程序概念

无论是 NT 式驱动还是 WDM 式驱动，都可以设计成分层的驱动程序。这种设计方法可以将功能复杂的驱动程序分解成多个简单的驱动程序。另外，对于一些标准的总线接口，操作系统提供总线驱动程序，程序员只需编写相对简单的驱动程序使其调用底层的总线驱动程序。这样有利于程序员避免重复开发，提高代码的重用性。

12.1.1 分层驱动程序的概念

分层驱动概念主要是为了区分前面介绍的单层驱动程序。分层驱动是指两个或两个以上的驱动程序，它们分别创建设备对象，并且形成一个由高到低的设备对象栈。IRP 请求一般会被传送到设备栈的最顶层的设备对象，顶层的设备对象可以选择直接结束 IRP 请求，也可以选择将 IRP 请求向下层的设备对象转发。如果是向下层设备对象转发 IRP 请求，当 IRP 请求结束时，IRP 会顺着设备栈的反方向原路返回。当得知下层驱动程序已经结束 IRP 请求时，本层设备对象可以选择继续将 IRP 向上返回，或者选择重新将 IRP 再次传递给底层设备驱动。

由此看来，分层驱动处理 IRP 要比单层驱动灵活得多。事实上，单层驱动也很少被用到，程序员大部分需要编写的是分层驱动程序。

分层驱动程序对应多个驱动程序，每个驱动程序创建一个设备对象，然后设备对象会一层一层地“挂载”在其他设备对象之上。这里所谓的挂载，指的是设备对象中的有个指

Windows 驱动开发技术详解

针指向了别的设备对象。为了搞清楚这个问题，首先来复习一下设备对象的数据结构。

```
typedef struct _DEVICE_OBJECT {  
    CSHORT Type; //这个数据结构类型  
    USHORT Size; //这个数据结构的大小  
    LONG ReferenceCount; //引用计数  
    struct _DRIVER_OBJECT *DriverObject; //驱动对象  
    struct _DEVICE_OBJECT *NextDevice; //下一个设备对象  
    struct _DEVICE_OBJECT *AttachedDevice; //附加的设备对象  
    struct _IRP *CurrentIrp; //当前的 IRP 指针  
    PIO_TIMER Timer; //计时器指针  
    ULONG Flags; //标记参数  
    ULONG Characteristics; //设备对象特性  
    PVOID DoNotUse1;  
    PVOID DeviceExtension; //设备扩展  
    DEVICE_TYPE DeviceType; //设备类型  
    CCHAR StackSize; //IO 堆栈大小  
    union {  
        LIST_ENTRY ListEntry; //IRP 链表  
        WAIT_CONTEXT_BLOCK Wcb;  
    } Queue;  
    ULONG AlignmentRequirement; //对齐属性  
    KDEVICE_QUEUE DeviceQueue; //设备队列  
    KDPC Dpc; //延迟过程调用  
    ULONG ActiveThreadCount; //当前线程的数量  
    PSECURITY_DESCRIPTOR SecurityDescriptor; //安全描述符表  
    KEVENT DeviceLock; //设备锁  
    USHORT SectorSize;  
    USHORT Spare1;  
    struct _DEVOBJ_EXTENSION *DeviceObjectExtension; //设备对象扩展  
    PVOID Reserved;  
} DEVICE_OBJECT;  
typedef struct _DEVICE_OBJECT *PDEVICE_OBJECT;
```

以上定义在 DDK.h 和 WDM.h 中都有。

其中有三个子域非常重要，分别是：

- **DriverObject**：这个参数是设备对象对应的驱动对象。
- **NextDevice**：这个参数记录下一个设备对象的指针。
- **AttachedDevice**：这个参数记录“自己”被哪一个设备对象所挂载。如果 DriverA 创建设备对象 DeviceA，DriverB 创建设备对象 DeviceB，并且 DeviceB 在设备栈的上层，而 DeviceA 在设备栈的下层，则 DeviceA 的 AttachedDevice 子域为 DeviceB 的地址，而 DeviceB 的 AttachedDevice 子域为 NULL。

分层驱动使用灵活，也是微软公司推荐使用的驱动模型，分层驱动程序一般会在以下的几种情况中用到。

(1) 大部分总线驱动程序（如 USB 总线驱动程序）已经由操作系统提供好了。如果想编写 USB 设备驱动程序，程序员一般不用了解太多 USB 总线协议的知识，只需要了解 USB 总线驱动程序提供的接口。

(2) 分层驱动程序使设计变得模块化。一个功能复杂的驱动程序可以划分为若干个小驱动程序，每个小驱动程序对上层驱动程序提供一个接口，上层驱动程序只关心下层驱动程序的接口，而不必关心底层驱动程序是如何实现的。

(3) 分层驱动程序可以对已经存在的驱动程序的功能进行修正，例如，某设备提供读写功能，而读写的大小没有限制。为了优化这个驱动程序的读写性能，可以在该驱动程序上层挂载一个新驱动程序，这个新驱动程序将读写请求分成大小相等的读写请求。

(4) 分层驱动程序还可以用于监视某个设备的操作情况。例如前面介绍的工具软件 IRPTrace，还有以后要介绍的 Bushound 软件（它可以用于监视 USB 设备的操作）。这些软件挂载在正常驱动程序之上，可以将各种 IRP 请求记录下来。

(5) 分层驱动程序还被用于某些计算机病毒。例如有些病毒挂载在键盘驱动程序之上，就可以监视计算机用户敲击键盘的所有信息。

(6) 在 Windows 2000 后，微软提出了 WDM（Windows Driver Model）驱动程序模型的概念。WDM 驱动程序就属于分层驱动程序。最简单的 WDM 驱动程序分为两层，一层是 PDO（物理设备对象 Physical Device Object），另一层是 FDO（功能设备对象 Function Device Object）。FDO 是挂载在 PDO 之上的，PDO 实现了即插即用的功能，FDO 完成逻辑功能，而将一些硬件相关的请求发往 PDO。关于 WDM 驱动程序与即插即用功能相关的内容将在第 13 章中进行介绍。

12.1.2 设备堆栈与挂载

挂载指的是将高一层的设备对象挂载在低一层的设备对象上，从而形成一个设备栈。实现挂载的内核函数是 IoAttachDeviceToDeviceStack，其声明如下：

```
PDEVICE_OBJECT
IoAttachDeviceToDeviceStack(
    IN PDEVICE_OBJECT SourceDevice,
    IN PDEVICE_OBJECT TargetDevice
);
```

- 第一个参数 SourceDevice：这个参数是设备栈中的一个设备指针。
- 第二个参数 TargetDevice：将 TargetDevice 挂载在设备栈的栈顶。
- 返回值：返回 TargetDevice 下面的低一层设备栈。

挂载操作如图 12-1 所示。假设在挂载前，设备栈中已经有了两个设备（设备 A 和设备 B）。IoAttachDeviceToDeviceStack 会将设备 C 挂载在设备 B 之上。

当驱动程序卸载的时候，驱动程序所创建的设备对象应该从设备栈中弹出。出栈的顺序必须和入栈的顺序相反。从设备栈弹出的内核函数是 IoDetachDevice，其函数声明如下：

```
VOID
IoDetachDevice(
    IN OUT PDEVICE_OBJECT TargetDevice
);
```


- **TargetDevice:** 它是位于被删除设备对象的低一层设备对象，如图 12-1 所示，如果要将设备 C 从设备栈中弹出，此时的 TargetDevice 为 DeviceB。

为了记住设备栈中的低一层驱动，一般会在设备扩展中记录这个对象指针。

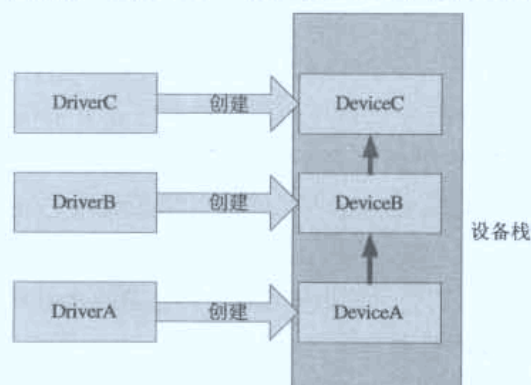


图 12-1 设备栈

12.1.3 I/O 堆栈

在 Windows 驱动模型中，还有一个概念叫做“I/O 堆栈”，用 `IO_STACK_LOCATION` 数据结构表示。它和设备堆栈紧密联合。IRP 一般会由应用程序的 `ReadFile` 或 `WriteFile` 创建，然后发送到设备堆栈的顶层。如果最上层的设备不处理 IRP，就会将 IRP 转发到下一层设备。每一层设备堆栈都有可能处理 IRP。

在 IRP 的数据结构中，存储着一个 `IO_STACK_LOCATION` 数组的指针。调用 `IoAllocateIrp` 内核函数创建 IRP 时，有一个 `StackSize` 参数，该参数就是 `IO_STACK_LOCATION` 数组的大小。

IRP 每穿越一次设备堆栈，就会用 `IO_STACK_LOCATION` 记录下本次操作的某些属性。I/O 堆栈和设备堆栈的关系如图 12-2 所示。

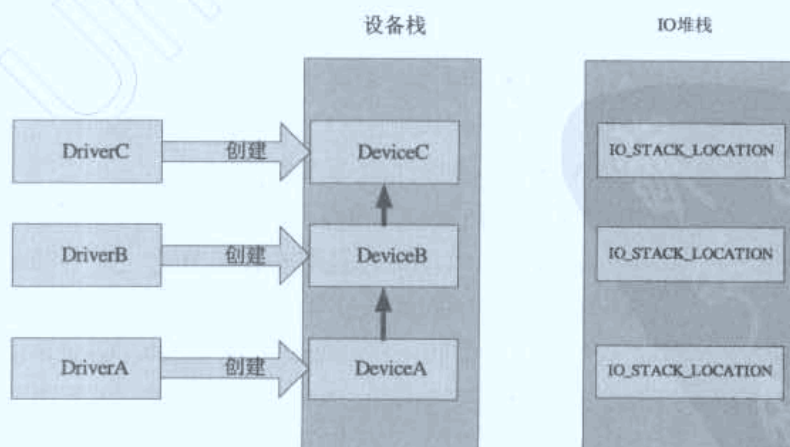


图 12-2 设备栈与 I/O 堆栈

12.1.4 向下转发 IRP

当顶层驱动的设备对象收到 IRP 请求并进入派遣函数后，有多种选择方式处理 IRP。

(1) 直接处理该 IRP，即调用 IoCompleteRequest 内核函数。

(2) 调用 StartIO，操作系统会将 IRP 请求串行化。除了当前运行的 IRP，其他的 IRP 请求进入 IRP 队列。

(3) 选择让底层驱动完成 IRP。

向下转发 IRP 涉及设备堆栈和 I/O 堆栈。一个设备堆栈对应着一个 I/O 堆栈元素 (IO_STACK_LOCATION)。IRP 内部有个指针指向当前正在使用的 IO_STACK_LOCATION，可以使用内核宏 IoGetCurrentIrpStackLocation 获得当前 I/O 堆栈。

每次调用 IoCallDriver 时，内核函数都会将 IRP 的当前指针下移，指向下一个 IO_STACK_LOCATION 指针。

但有时候，当前设备堆栈不对 IRP 做任何处理。因此，当前设备就不需要对应 I/O 堆栈。但是 IoCallDriver 已经将当前 I/O 堆栈向下移动了一个单位，所以 DDK 提供了内核宏 IoSkipCurrentIrpStackLocation，它的作用就是将当前 I/O 堆栈又往回 (上) 移动一个单位。

这样 IoCallDriver 和 IoSkipCurrentIrpStackLocation 对设备堆栈的移动就实现了平衡，也就是没有改变。这时 IoCallDriver 调用的低一层驱动所用到的 I/O 堆栈，实际上和一层用到的是同一个。因此，当本层驱动不需要用 I/O 堆栈时，可以做如下操作：

```
#001 PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
#002
#003 //调用底层驱动
#004 IoSkipCurrentIrpStackLocation (pIrp);
#005
#006 ntStatus = IoCallDriver(pdx->TargetDevice, pIrp);
```

在另外一种情况下，即当前 IRP 也参与操作时，就需要将当前 I/O 堆栈的参数复制到下一层，需要调用内核宏 IoCopyCurrentIrpStackLocationToNext，也就是做如下操作：

```
#001 PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
#002
#003 //调用底层驱动
#004 IoCopyCurrentIrpStackLocationToNext(pIrp);
#005
#006 ntStatus = IoCallDriver(pdx->TargetDevice, pIrp);
```

为了更清楚地了解 IoSkipCurrentIrpStackLocation 和 IoCopyCurrentIrpStackLocationToNext 两个内核宏的原理，笔者将 DDK 中两个宏的定义列出如下：

```
#001 #define IoSkipCurrentIrpStackLocation( Irp ) \
#002     (Irp)->CurrentLocation++; \
#003     (Irp)->Tail.Overlay.CurrentStackLocation++;
#004
#005 #define IoCopyCurrentIrpStackLocationToNext( Irp ) { \
#006     PIO_STACK_LOCATION irpSp; \
```


Windows 驱动开发技术详解

```
#007     PIO_STACK_LOCATION nextIrpSp; \
#008     IrpSp = IoGetCurrentIrpStackLocation( (Irp) ); \
#009     nextIrpSp = IoGetNextIrpStackLocation( (Irp) ); \
#010     RtlCopyMemory( nextIrpSp, IrpSp, FIELD_OFFSET(IO_STACK_LOCATION,
CompletionRoutine)); \
#011     nextIrpSp->Control = 0; }
```

程序员可以根据自己的需要,在适当的情况下选择使用 `IoSkipCurrentIrpStackLocation` 或者 `IoCopyCurrentIrpStackLocationToNext`。

12.1.5 挂载设备对象示例

下面的示例是将 `DriverB` 加载在 `DriverA` 上,并且在 `DriverB` 的派遣函数中不对当前设备栈处理,直接将 `IRP` 转发到下层设备上,即 `DriverA` 上。

`DriverA` 是实现读的普通驱动, `DriverB` 首先在 `DriverEntry` 上,用 `IoGetDeviceObjectPointer` 找到 `DriverA` 所创建的设备对象,并将自己的设备对象挂载在设备栈上,并用设备扩展将底层驱动的设备对象记录下来。

```
#001 #pragma INITCODE
#002 extern "C" NTSTATUS DriverEntry (
#003     IN PDRIVER_OBJECT pDriverObject,
#004     IN PUNICODE_STRING pRegistryPath )
#005 {
#006     NTSTATUS ntStatus;
#007     KdPrint(("DriverB:Enter B DriverEntry\n"));
#008
#009     //设置驱动的卸载函数
#010     pDriverObject->DriverUnload = HelloDDKUnload;
#011     //设置 IRP 派遣函数
#012     pDriverObject->MajorFunction[IRP_MJ_CREATE] = HelloDDKCreate;
#013     pDriverObject->MajorFunction[IRP_MJ_CLOSE] = HelloDDKClose;
#014     pDriverObject->MajorFunction[IRP_MJ_WRITE] = HelloDDKDispatchRoutine;
#015     pDriverObject->MajorFunction[IRP_MJ_READ] = HelloDDKDispatchRoutine;
#016     //初始化 UNICODE 字符串
#017     UNICODE_STRING DeviceName;
#018     RtlInitUnicodeString( &DeviceName, L"\\Device\\MyDDKDeviceA" );
#019
#020     PDEVICE_OBJECT DeviceObject = NULL;
#021     PFILE_OBJECT FileObject = NULL;
#022     //寻找 DriverA 创建的设备对象
#023     ntStatus = IoGetDeviceObjectPointer(&DeviceName, FILE_ALL_ACCESS,
&FileObject, &DeviceObject);
#024     //判断操作是否成功
#025     if (!NT_SUCCESS(ntStatus))
#026     {
#027         KdPrint(("DriverB:IoGetDeviceObjectPointer() 0x%x\n", ntStatus));
#028         return ntStatus;
#029     }
#030
#031     //创建自己的驱动设备对象
#032     ntStatus = CreateDevice(pDriverObject);
#033     //判断操作是否成功
#034     if (!NT_SUCCESS( ntStatus ) )
```

```

#035     {
#036         //将引用计数减一
#037         ObDereferenceObject( FileObject );
#038         DbgPrint( "IoCreateDevice() 0x%x!\n", ntStatus );
#039         return ntStatus;
#040     }
#041     //得到设备扩展
#042     PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) pDriverObject->DeviceObject->
DeviceExtension;
#043     //从设备扩展得到下层驱动
#044     PDEVICE_OBJECT FilterDeviceObject = pdx->pDevice;
#045
#046     //将自己的设备对象挂载在 DriverA 的设备对象上
#047     PDEVICE_OBJECT TargetDevice = IoAttachDeviceToDeviceStack(FilterDeviceOb
#048                                     ject, DeviceObject );
#049     //将底层设备对象记录下来
#050     pdx->TargetDevice = TargetDevice;
#051
#052     if ( !TargetDevice )
#053     {
#054         //将引用计数减 1
#055         ObDereferenceObject( FileObject );
#056         //删除设备对象
#057         IoDeleteDevice( FilterDeviceObject );
#058         DbgPrint( "IoAttachDeviceToDeviceStack() 0x%x!\n", ntStatus );
#059         return STATUS_INSUFFICIENT_RESOURCES;
#060     }
#061     //设置设备类型
#062     FilterDeviceObject->DeviceType = TargetDevice->DeviceType;
#063     //设置设备对象特性
#064     FilterDeviceObject->Characteristics = TargetDevice->Characteristics;
#065     //设置设备对象标记
#066     FilterDeviceObject->Flags &= -PO_DEVICE_INITIALIZING;
#067     FilterDeviceObject->Flags |= ( TargetDevice->Flags & ( DO_DIRECT_IO |
#068                                     DO_BUFFERED_IO ) );
#069     //将引用计数减 1
#070     ObDereferenceObject( FileObject );
#071     KdPrint(("DriverB:B attached A successfully!\n"));
#072     KdPrint(("DriverB:Leave B DriverEntry\n"));
#073     return ntStatus;
#074 }

```

此段代码可以在配套光盘中本章的 Test1 目录下找到。

12.1.6 转发 IRP 示例

在成功将 DriverB 挂载在 DriverA 之上后，以后所有向 DriverA 发送的 IRP 请求，首先会发送到 DriverA 创建的设备堆栈的栈顶，然后一层层下发。

以下是 Driver B 的派遣函数的示例，此派遣函数仅仅向下转发，而没有做任何处理，因此使用的是 IoSkipCurrentIrpStackLocation。

```

#001 #pragma PAGEDCODE
#002 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#003                        IN PIRP pIrp)

```



```

#004 {
#005     KdPrint(("DriverB:Enter B HelloDDKCreate\n"));
#006     NTSTATUS ntStatus = STATUS_SUCCESS;
#007     //将自己完成 IRP, 改成由底层驱动负责
#008     //得到设备扩展
#009     PDEVICE_EXTENSION pDx = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
#010     //略过当前 I/O 堆栈
#011     IoSkipCurrentIrpStackLocation (pIrp);
#012     //调用底层驱动
#013     ntStatus = IoCallDriver(pDx->TargetDevice, pIrp);
#014
#015     KdPrint(("DriverB:Leave B HelloDDKCreate\n"));
#016     return ntStatus;
#017 }

```

此段代码可以在配套光盘中本章的 Test1 目录下找到。

在实验此段代码时, 首先将 DriverA 的设备加载, 然后再加载 DriverB, 如果先加载 DriverB, 会导致 DriverB 的 DriverEntry 无法找到 DriverA 创建的设备。

之后调用应用程序, 此应用程序是对于 DriverA 设备的, 但却首先将 IRP 发送到 DriverB 的设备, 然后再发送到 DriverA 的设备。读者可以自己观察驱动输出的 log 信息, 如图 12-3 所示。

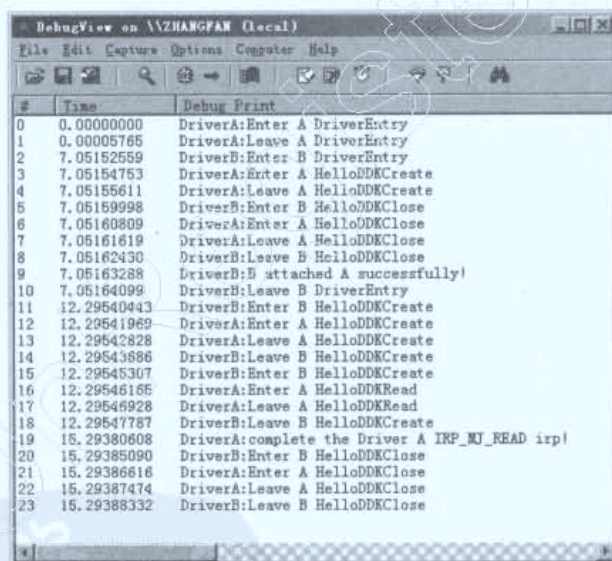


图 12-3 DebugView 输出 log

12.1.7 分析

在上述的例子中, 加载完 DriverA 及 DriverB 的驱动后, 可以使用 XP DDK 所提供的工具软件 DeviceTree 查看设备栈 (WinObj 工具也可以查看类似内容, 但功能没有 DeviceTree 强大)。如图 12-4 所示为 DeviceTree 观察 DriverA 和 DriverB 及相关设备驱动的直接关系。图中左边的字母“ATT”是 ATTACH 的缩写, 即设备 B 挂载在设备 A 上。

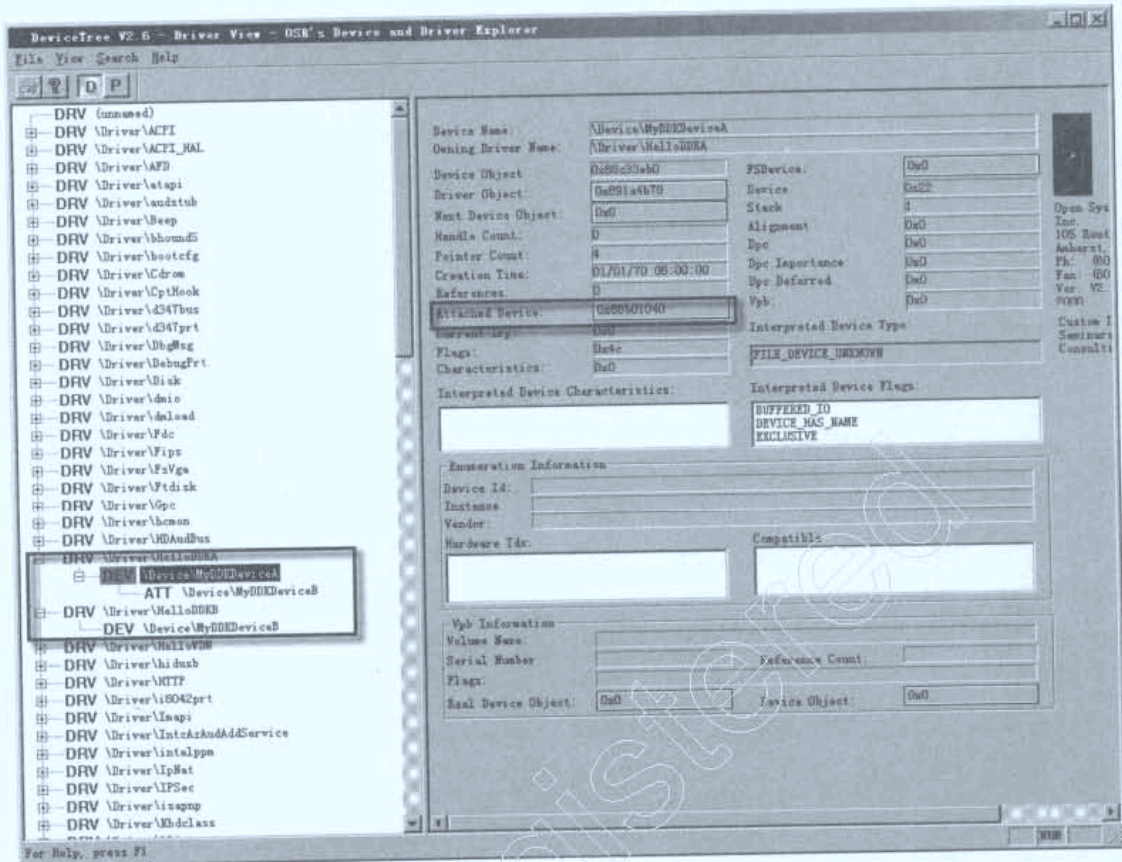


图 12-4 DeviceTree 查看设备栈

12.1.8 遍历设备栈

DeviceTree 和 WinObj 查看驱动对象、设备对象的原理都很简单。首先编写一个驱动程序，在该驱动程序中枚举所有驱动对象，通过驱动对象可以枚举到该驱动对象所创建的设备对象。通过设备对象可以枚举所有同类的设备对象的地址。通过设备对象还可以枚举到附加在此之上的设备对象，从而枚举出一整条设备堆栈。只要仔细分析设备对象、驱动对象的数据结构，同时再编写应用程序将这些信息通过驱动读取出来即可。以下是遍历设备的部分示例代码：

```
#001 VOID
#002 GetDeviceObjectInfo( PDEVICE_OBJECT DevObj )
#003 {
#004     POBJECT_HEADER ObjectHeader;
#005     POBJECT_HEADER_NAME_INFO ObjectNameInfo;
#006     //判断设备对象是否为空
#007     if ( DevObj == NULL )
#008     {
#009         DbgPrint( "DevObj is NULL!\n" );
#010     }
#011     return;
```


Windows 驱动开发技术详解

```
#011     }
#012     // 得到对象头
#013     ObjectHeader = OBJECT_TO_OBJECT_HEADER( DevObj );
#014     //判断对象头是否为空
#015     if ( ObjectHeader )
#016     {
#017         // 查询设备名称并打印
#018         ObjectNameInfo = OBJECT_HEADER_TO_NAME_INFO( ObjectHeader );
#019
#020         if ( ObjectNameInfo && ObjectNameInfo->Name.Buffer )
#021         {
#022             DbgPrint( "Driver Name:%wZ - Device Name:%wZ - Driver Address:0x%x - Device Address:0x%x\n",
#023                     &DevObj->DriverObject->DriverName,
#024                     &ObjectNameInfo->Name,
#025                     DevObj->DriverObject,
#026                     DevObj );
#027         }
#028
#029         // 对于没有名称的设备, 则打印 NULL
#030         else if ( DevObj->DriverObject )
#031         {
#032             DbgPrint( "Driver Name:%wZ - Device Name:%S - Driver Address:0x%x - Device Address:0x%x\n",
#033                     &DevObj->DriverObject->DriverName,
#034                     L"NULL",
#035                     DevObj->DriverObject,
#036                     DevObj );
#037         }
#038     }
#039 }
#040 VOID
#041 GetAttachedDeviceInfo( PDEVICE_OBJECT DevObj )
#042 {
#043     PDEVICE_OBJECT DeviceObject;
#044     //判断设备对象是否为空
#045     if ( DevObj == NULL )
#046     {
#047         DbgPrint( "DevObj is NULL!\n" );
#048         return;
#049     }
#050     //得到附加设备
#051     DeviceObject = DevObj->AttachedDevice;
#052     //枚举设备栈中的每一个设备
#053     while ( DeviceObject )
#054     {
#055         DbgPrint( "Attached Driver Name:%wZ, Attached Driver Address:0x%x, Attached DeviceAddress:0x%x\n",
#056                 &DeviceObject->DriverObject->DriverName,
#057                 DeviceObject->DriverObject,
#058                 DeviceObject );
#059         //从设备堆栈得到附加设备对象
#060         DeviceObject = DeviceObject->AttachedDevice;
#061     }
#062 }
#063 PDRIVER_OBJECT
```

```

#064 EnumDeviceStack( PWSTR pwszDeviceName )
#065 {
#066     UNICODE_STRING DriverName;
#067     PDRIVER_OBJECT DriverObject = NULL;
#068     PDEVICE_OBJECT DeviceObject = NULL;
#069     //初始化 UNICODE 字符串
#070     RtlInitUnicodeString( &DriverName, pwszDeviceName );
#071     //从设备对象得到设备指针
#072     ObReferenceObjectByName( &DriverName,
#073                             OBJ_CASE_INSENSITIVE,
#074                             NULL,
#075                             0,
#076                             ( POBJECT_TYPE ) IoDriverObjectType,
#077                             KernelMode,
#078                             NULL,
#079                             ( PVOID*)&DriverObject );
#080     //判断驱动对象是否为空
#081     if ( DriverObject == NULL )
#082     {
#083         return NULL;
#084     }
#085     //通过驱动对象得到设备对象
#086     DeviceObject = DriverObject->DeviceObject;
#087
#088     while ( DeviceObject )
#089     {
#090         //获取设备对象
#091         GetDeviceObjectInfo( DeviceObject );
#092
#093         // 判断当前设备上是否有过滤驱动 (Filter Driver)
#094         if ( DeviceObject->AttachedDevice )
#095         {
#096             GetAttachedDeviceInfo( DeviceObject );
#097         }
#098
#099         // 进一步判断当前设备上 VPB 中的设备
#100         if ( DeviceObject->Vpb && DeviceObject->Vpb->DeviceObject )
#101         {
#102             //获取设备对象
#103             GetDeviceObjectInfo( DeviceObject->Vpb->DeviceObject );
#104             if ( DeviceObject->Vpb->DeviceObject->AttachedDevice )
#105             {
#106                 GetAttachedDeviceInfo( DeviceObject->Vpb->DeviceObject );
#107             }
#108         }
#109         // 得到建立在此驱动上的下一个设备 DEVICE_OBJECT
#110         DeviceObject = DeviceObject->NextDevice;
#111     }
#112     return DriverObject;
#113 }

```

此段代码可以在配套光盘中本章的 Test2 目录下找到。

以/Driver/ACPI（高级电源管理驱动）驱动下的设备栈为例，如图 12-5 所示为驱动枚举出 ACPI 驱动的设备栈，可以看出，ACPI 的设备栈还是比较复杂的。

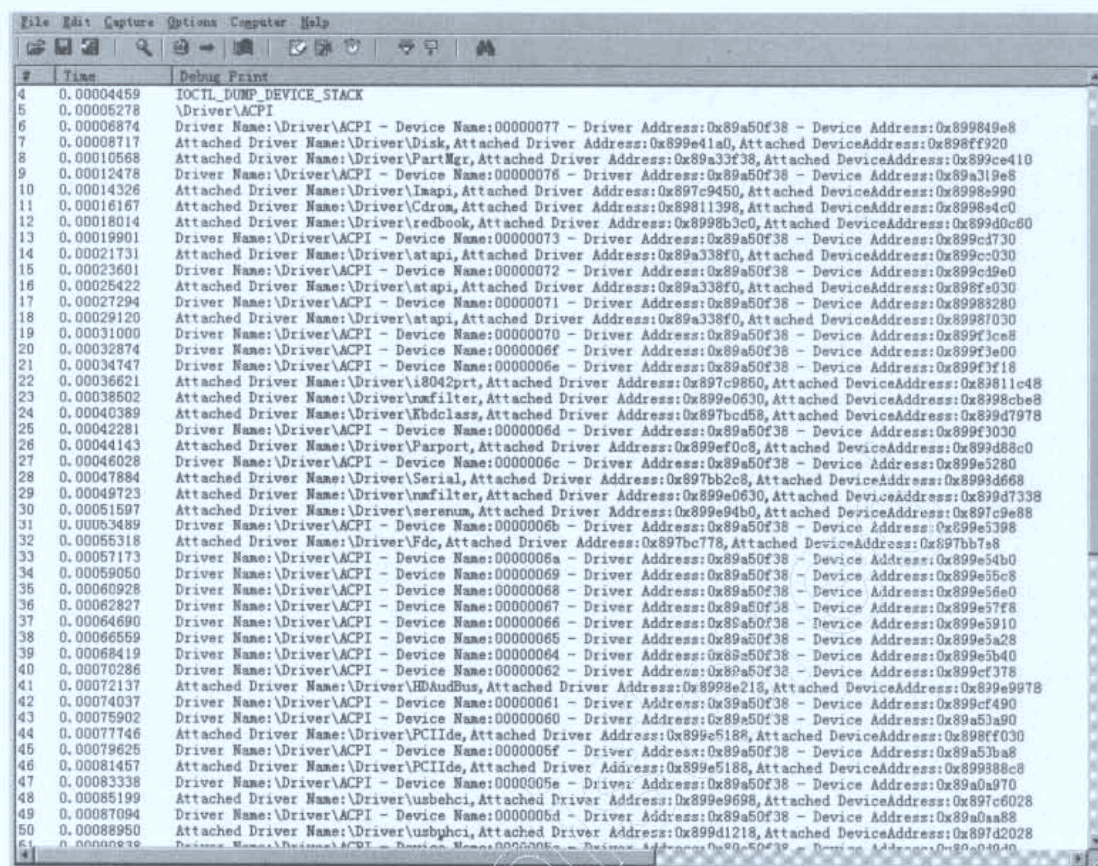


图 12-5 DebugView 输出 log

12.2 完成例程

在将 IRP 发送给底层驱动或者其他驱动前，可以对 IRP 设置一个完成例程。一旦底层驱动将 IRP 完成后，IRP 完成例程立刻被触发。通过设置完成例程可以方便地使程序员了解其他驱动对 IRP 进行的处理。

12.2.1 完成例程概念

完成例程是驱动程序编写中的一种常用技巧。当驱动程序调用自己的底层驱动，或者像 11 章中介绍的那样调用其他驱动的时候，可以通过完成例程获得通知。不管是调用自己的底层驱动或者调用其他驱动，都是使用内核函数 IoCallDriver。

当 IoCallDriver 将 IRP 的控制权交给被动驱动时，有两种情况。第一种情况，即调用的设备是同步完成这个 IRP 的，从 IoCallDriver 返回的时刻，即代表此 IRP 已经完成。第二种情况，就是调用的设备是异步操作，IoCallDriver 会立刻返回 IoCallDriver，但此时并

没有真正地完成 IRP。

在第二种情况下，调用 IoCallDriver 前，先对 IRP 注册一个完成例程，当底层驱动或者其他驱动完成此 IRP 时，此完成例程立刻被调用。其实注册 IRP 的完成例程就是在当前的堆栈（IO_STACK_LOCATION）中的 CompletionRoutine 子域。IRP 完成后，一层层堆栈向上弹出，如果遇到 IO_STACK_LOCATION 的 CompletionRoutine 非空，则调用这个函数，另外传进这个完成例程的是 IO_STACK_LOCATION 的子域 Context。

如果使用完成例程，就不能使用内核宏 IoSkipCurrentIrpStackLocation，即不能将本层 IRP 作为下层 I/O 堆栈。而必须使用 IoCopyCurrentIrpStackLocationToNext，将本层 I/O 堆栈拷贝到下一层的 I/O 堆栈中。对于初级程序员来说不必关心以上细节，DDK 已经提供了内核宏，即 IoSetCompletionRoutine，其声明如下：

```
VOID
IoSetCompletionRoutine(
    IN PIRP Irp,
    IN PIO_COMPLETION_ROUTINE CompletionRoutine,
    IN PVOID Context,
    IN BOOLEAN InvokeOnSuccess,
    IN BOOLEAN InvokeOnError,
    IN BOOLEAN InvokeOnCancel
);
```

- 第一个参数 Irp：表示要设置的 IRP。
- 第二个参数 CompletionRoutine：表示对 IRP 设置的完成例程。如果该值为 NULL 就意味着将完成例程取消。
- 第三个参数 Context：表示完成例程的上下文，也就是传送进完成例程的参数。
- 第四个参数 InvokeOnSuccess：指定是否 IRP 被成功完成后进入完成例程。
- 第五个参数 InvokeOnError：指定是否 IRP 被错误完成后进入完成例程。
- 第六个参数 InvokeOnCancel：指定是否 IRP 被取消完成后进入完成例程。

读者可以对照上述说明，并和 IoSetCompletionRoutine 宏的定义做一下比较，将会加深理解 IoSetCompletionRoutine 宏的意义。

```
#001 #define IoSetCompletionRoutine( Irp, Routine, CompletionContext, Success,
Error, Cancel ) { \
#002     PIO_STACK_LOCATION irpSp; \
#003     ASSERT( (Success) | (Error) | (Cancel) ? (Routine) != NULL : TRUE ); \
#004     irpSp = IoGetNextIrpStackLocation( (Irp) ); \
#005     irpSp->CompletionRoutine = (Routine); \
#006     irpSp->Context = (CompletionContext); \
#007     irpSp->Control = 0; \
#008     if ((Success)) { irpSp->Control = SL_INVOKE_ON_SUCCESS; } \
#009     if ((Error)) { irpSp->Control |= SL_INVOKE_ON_ERROR; } \
#010     if ((Cancel)) { irpSp->Control |= SL_INVOKE_ON_CANCEL; } }
```

当 IRP 处于某个设备栈时，IRP 被 IoCompleteRequest 完成的时候，会一层层出栈。出栈的时候如果遇到该栈有完成例程，则调用该栈的完成例程。因此，完成例程可以作为通知 IRP 完成的一个标志。并且，在完成例程中可以很清楚地知道 IRP 的完成情况，例如，

IRP 的 status、读写的情况，操作字节的数量等。另外，在完成例程中，提供了一个机会重新获取对 IRP 的“控制”。

当调用 IoCallDriver 之后，当前的驱动就失去了对 IRP 的控制，如果这时候设置 IRP 的属性，会引起系统的崩溃。完成例程只有两种返回的可能，一种是 STATUS_SUCCESS 或 STATUS_CONTINUE_COMPLETION。其实这两个状态数值一样，是等价的，STATUS_CONTINUE_COMPLETION 是 STATUS_SUCCESS 的别名。这种情况下，驱动不会再得到 IRP 的控制。另一种情况是完成例程返回 STATUS_MORE_PROCESSING_REQUIRED，这时候本层设备堆栈会重新获得 IRP 的控制权，并且设备栈不会向上弹出，也就是向上“回卷”设备栈停止。此时可以选择再次向底层发送 IRP。

12.2.2 传播 Pending 位

每当低级驱动完成 IRP 后，将 IRP 的堆栈向上回卷时，底层 I/O 堆栈中 Control 域的 SL_PENDING_RETURNED 位必须被传播到上一层。如果本层没有设置完成例程，那么这种传播是自动的，即不用程序员指定。如果本层堆栈设置了完成例程，则这种传播需要程序员自己实现。例如，调用 IoCallDriver 低层驱动时，返回的是 STATUS_PENDING。如果没有设置完成例程，需要如下编写代码：

```
#001 // 向下复制堆栈
#002 IoCopyCurrentIrpStackLocationToNext( Irp );
#003 // 向下传递 IRP
#004 status = IoCallDriver( nextDevice, Irp );
#005 // 直接返回底层驱动状态
#006 return status;
```

此时没有调用 IoMarkIrpPending，是因为没有设置完成例程，底层驱动会自动将堆栈的 Control 域的 SL_PENDING_RETURNED 位复制到本层堆栈。

然而如果是设置了完成例程，传播 Pending 位的任务就留给了程序员完成。对于设置了完成例程，下面的代码似乎是正确的：

```
#001 // 向下复制堆栈
#002 IoCopyCurrentIrpStackLocationToNext( Irp );
#003 // 向下传递 IRP
#004 status = IoCallDriver( nextDevice, Irp );
#005 // 下面的代码是有错误的！！
#006 // 因为调用 IoCallDriver 后，IRP 就不再归本层所有，因此再操作 IRP 会导致错误
#007 If (status == STATUS_PENDING) {
#008     IoMarkIrpPending( Irp );
#009 }
#010 // 直接返回底层驱动状态
#011 return status;
```

由于调用 IoCallDriver 后，IRP 就不再归当前堆栈所有，因此以上的代码有一点问题。为此，传播 Pending 位的任务就留给了完成例程。以下的代码是正确的：

```

#001 NTSTATUS
#002 CompletionRoutine( ... )
#003 {
#004     if (Irp->PendingReturned) {
#005         // 如果底层驱动是否有 pending 位
#006         // 1 则当前堆栈也设置 pending 位
#007         IoMarkIrpPending( Irp );
#008     }
#009 }
#010
#011 ... //完成例程的其他附加行为
#012 return STATUS_CONTINUE_COMPLETION;
#013 }

```

12.2.3 完成例程返回 STATUS_SUCCESS

当 IRP 被 `IoCompleteRequest` 完成时, IRP 就会沿着一层层的设备堆栈向上回卷。如果途经遇到某设备堆栈的完成例程, 则进入该完成例程。完成例程如果返回 `STATUS_SUCCESS` (别名是 `STATUS_CONTINUE_COMPLETION`), 则继续向上回卷。此时的完成例程仅仅就是一个通知, 表明 IRP 已完成。

以下是使用完成例程, 并在完成例程返回 `STATUS_SUCCESS` 的例子。

```

#001 NTSTATUS
#002 MyIoCompletion(
#003     IN PDEVICE_OBJECT DeviceObject,
#004     IN PIRP Irp,
#005     IN PVOID Context
#006 )
#007 {
#008     //进入此函数标志底层驱动设备将 IRP 完成
#009     KdPrint(("Enter MyIoCompletion\n"));
#010     if (Irp->PendingReturned)
#011     {
#012         //传播 pending 位
#013         IoMarkIrpPending( Irp );
#014     }
#015     return STATUS_SUCCESS; //同 STATUS_CONTINUE_COMPLETION
#016 }
#017
#018 #pragma PAGEDCODE
#019 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#020                      IN PIRP pIrp)
#021 {
#022     KdPrint(("DriverB:Enter B HelloDDKRead\n"));
#023     NTSTATUS ntStatus = STATUS_SUCCESS;
#024     //将自己完成 IRP, 改成由底层驱动负责
#025
#026     PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
#027
#028     //将当前 I/O 堆栈拷贝到底层堆栈
#029     IoCopyCurrentIrpStackLocationToNext(pIrp);
#030
#031     //设置完成例程

```



```

#032   IoSetCompletionRoutine(pIrp, MyIoCompletion, NULL, TRUE, TRUE, TRUE);
#033
#034   //调用底层驱动
#035   ntStatus = IoCallDriver(pdx->TargetDevice, pIrp);
#036
#037   //当 IoCallDriver 后, 并且完成例程返回的是 STATUS_SUCCESS
#038   //IRP 就不再属于派遣函数, 就不能对 IRP 进行操作了
#039   if (ntStatus == STATUS_PENDING)
#040   {
#041       KdPrint(("STATUS_PENDING\n"));
#042   }
#043   //将状态设置为挂起
#044   ntStatus = STATUS_PENDING;
#045
#046   KdPrint(("DriverB:Leave B HelloDDKRead\n"));
#047
#048   return ntStatus;
#049 }

```

此段代码可以在配套光盘中本章的 Test3 目录下找到。

本例子是将 DriverA 挂载在 DriverB 之上, 试验时首先加载 DriverA 驱动, 再加载 DriverB, 最后调用应用程序发起读请求, 运行结果参照图 12-6 所示。

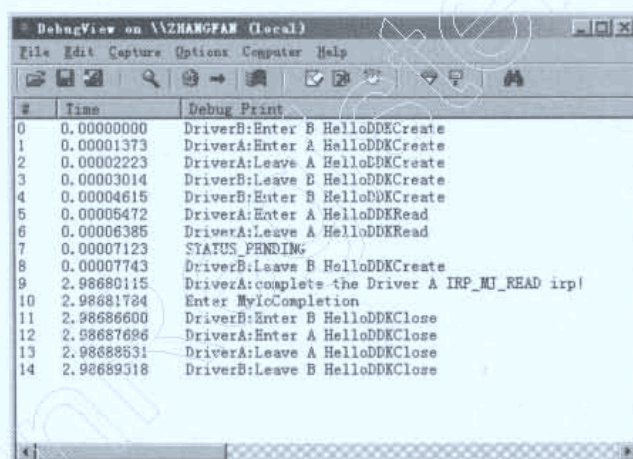


图 12-6 DebugView 输出 log

12.2.4 完成例程返回 STATUS_MORE_PROCESSING_REQUIRED

当 IRP 被 IoCompleteRequest 完成时, IRP 就会沿着一层层的设备堆栈向上回卷。如果途经遇到某设备堆栈的完成例程, 则进入该完成例程。完成例程如果返回 STATUS_MORE_PROCESSING_REQUIRED, 则停止向上回卷。这时的本层堆栈又重新获得 IRP 的控制, 并且该 IRP 从完成状态又变成了未完成状态, 需要再次完成, 即需要再次执行 IoCompleteRequest。

重新获得的 IRP 可以再次发往底层驱动, 也可以自己标志完成, 即调用 IoCompleteRequest。

以下给出了使用完成例程的例子，该完成例程返回 STATUS_MORE_PROCESSING_REQUIRED。在调用 IoCallDriver 之后，当前设备想等待底层驱动将设备完成后再继续执行。因此，IoCallDriver 初始化了个事件，并将事件指针传递给了完成例程。IRP 被完成后进入完成例程，并触发事件。

这种技巧被广泛应用于驱动程序的编写中，代码如下：

```
#001 NTSTATUS
#002 MyIoCompletion(
#003     IN PDEVICE_OBJECT DeviceObject,
#004     IN PIRP Irp,
#005     IN PVOID Context
#006 )
#007 {
#008     //判断是否是挂起返回
#009     if (Irp->PendingReturned == TRUE)
#010     {
#011         //设置事件
#012         KeSetEvent((PKEVENT)Context, IO_NO_INCREMENT, FALSE);
#013     }
#014     //返回 STATUS_MORE_PROCESSING_REQUIRED 说明 IRP 需要再次被结束
#015     return STATUS_MORE_PROCESSING_REQUIRED;
#016 }
#017
#018 #pragma PAGEDCODE
#019 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#020                       IN PIRP pIrp)
#021 {
#022     KdPrint(("DriverB:Enter B HelloDDKRead\n"));
#023     NTSTATUS ntStatus = STATUS_SUCCESS;
#024     //将自己完成 IRP，改成由底层驱动负责
#025     //得到设备扩展
#026     PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
#027     //将本层的 I/O 堆栈复制到底层堆栈
#028     IoCopyCurrentIrpStackLocationToNext(pIrp);
#029     KEVENT event;
#030     //初始化事件
#031     KeInitializeEvent(&event, NotificationEvent, FALSE);
#032     //设置完成例程
#033     IoSetCompletionRoutine(pIrp, MyIoCompletion, &event, TRUE, TRUE, TRUE);
#034     //调用底层驱动
#035     ntStatus = IoCallDriver(pdx->TargetDevice, pIrp);
#036     //判断 IRP 是否被挂起
#037     if (ntStatus == STATUS_PENDING)
#038     {
#039         KdPrint(("IoCallDriver return STATUS_PENDING,Waiting ...\n"));
#040         //等待同步事件
#041         KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);
#042         ntStatus = pIrp->IoStatus.Status;
#043     }
#044
#045     //虽然在底层驱动已经将 IRP 完成了，但是由于完成例程返回的是
#046     //STATUS_MORE_PROCESSING_REQUIRED，因此需要再次调用 IoCompleteRequest!
#047     IoCompleteRequest(pIrp, IO_NO_INCREMENT);
#048     KdPrint(("DriverB:Leave B HelloDDKRead\n"));
```



```
#049     return ntStatus;
#050 }
```

此段代码可以在配套光盘中本章的 Test4 目录下找到。

本例子是将 DriverA 挂载在 DriverB 之上，试验时首先加载 DriverA 驱动，再加载 DriverB，最后调用应用程序发起读请求，运行结果参照图 12-7 所示。

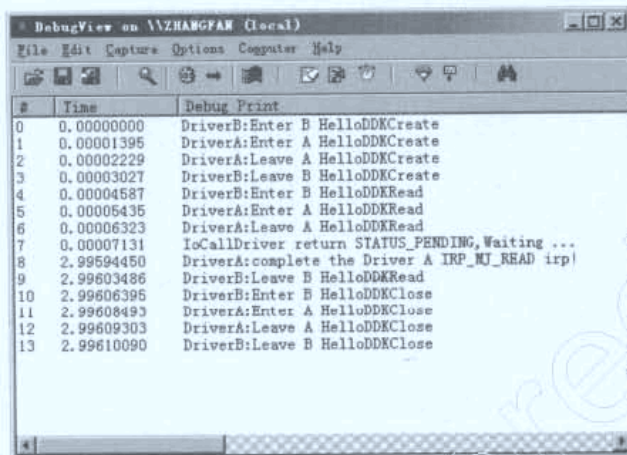


图 12-7 DebugView 输出 log

12.3 将 IRP 分解成多个 IRP

IRP 可以分解成多个小 IRP，例如，需要对某个设备读写大量的数据，但是设备所支持的一次物理读写只有很小的字节数，这样对设备的读写操作就可以分解成多个 IRP，每个 IRP 所读写的字节数都在设备所允许的范围内。

12.3.1 原理

在驱动编写中，经常会遇到这种需求，就是将 IRP 请求分成多个 IRP 请求。例如，某驱动（DriverA）实现了读取功能，但是对读取的字节只能在 1024 个字节以内，不支持更多的字节读取。这时候，应用程序每次的读请求只能是 1024 字节以内。

如果在已有原驱动（DriverB）的情况下，编写一个中间层驱动（DriverB）可以解决读取字节数限制的问题。DriverB 允许读取任意字节的操作，也就是应用程序可以请求任意字节的读取请求，DriverB 的操作如下：

- ① 如果读取字节数 N 是 1024 字节以内，就直接转发 IRP 给 DriverA。
- ② 如果读取字节数 N 是大于 1024 字节，则将当前 IRP 读取字节数设置为 1024，并设置一个完成例程，将 IRP 转发到 DriverA。
- ③ 一旦进入完成例程，就代表完成一个 1024 个字节的读取。这时候继续利用 IRP，并重新转发 IRP 给 DriverA。由于 IRP 还要继续转发，所以完成例程退出时返回 STATUS_

MORE_PROCESSING_REQUIRED。

④ 这样重复②、③步骤，周而复始，直到将所有字节数都传送完毕，这时候 IRP 操作才算真正完毕。

如图 12-8 所示，左边是应用程序直接调用 DriverA 的情形，右边是应用程序通过 DriverB 间接调用 DriverA 的情形。显然直接调用 DriverA 时，只能读取 1024 字节以内，而通过 DriverB 就可以读取任意字节。

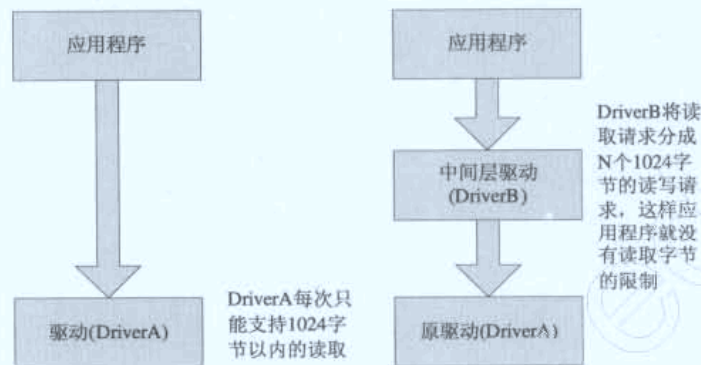


图 12-8 分层驱动模型

图 12-8 所示的情形在驱动程序中会经常遇到，例如在 USB 设备中，微软会提供一个 USB 总线驱动来负责和 USB 总线打交道，并读取数据，每次传输的字节数（每个包的大小）是事先确定的。程序员一般需要做的是编写一个驱动，去调用底层的 USB 总线驱动，将 IRP 分解成小数据的 IRP 传递给底层 USB 总线驱动。

12.3.2 准备底层驱动

如图 12-8 所示，本节重点讲述的是如何编写形如 DriverB 的驱动。但是必须准备一个底层驱动，形如 DriverA，这个驱动提供 1024 字节以内的读取操作。另外，为了众多总线驱动（如 USB 总线驱动），DriverA 采用的是直接操作方式，也就是在创建设备时采取 DO_DIRECT_IO 参数。读者可以通过下面的例子重新复习一下如何编写直接方式的设备驱动。

```
#001 #pragma PAGEDCODE
#002 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#003                          IN PIRP pIrp)
#004 {
#005     KdPrint(("DriverA:Enter A HelloDDKRead\n"));
#006     NTSTATUS status = STATUS_SUCCESS;
#007     //获得当前 I/O 堆栈
#008     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
#009
#010     ULONG totalLength;
#011     PVOID virtualAddress;
#012     //判断 MDL 是否为空
#013     if (!pIrp->MdlAddress)
#014     {
```



```

#015         status = STATUS_UNSUCCESSFUL;
#016         totalLength = 0;
#017         goto HelloDDKRead_EXIT;
#018     }
#019     //获得 MDL 的虚拟地址
#020     virtualAddress = MmGetMdlVirtualAddress(pIrp->MdlAddress);
#021     totalLength = MmGetMdlByteCount(pIrp->MdlAddress);
#022     //填充内存
#023     RtlFillMemory(virtualAddress,totalLength,0xFF);
#024
#025     KdPrint(("DriverA:virtualAddress:%x\n",virtualAddress));
#026     KdPrint(("DriverA:totalLength:%d\n",totalLength));
#027
#028 HelloDDKRead_EXIT:
#029     // 设置 IRP 完成状态
#030     pIrp->IoStatus.Status = status;
#031     //设置 IRP 操作字节数
#032     pIrp->IoStatus.Information = totalLength;
#033     //结束 IRP 请求
#034     IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#035     KdPrint(("DriverA:Leave A HelloDDKRead\n"));
#036     return status;
#037 }

```

此段代码可以在配套光盘中本章的 Test5 目录下找到。

12.3.3 读派遣函数

如图 12-9 所示,描述了应用程序、DriverA、DriverB 之间的调用关系,但是没有能具体表现 DriverB 的读派遣函数、DriverA 的读派遣函数、DriverB 的完成例程之间的调用关系。而是进一步描述了三者之间的关系。

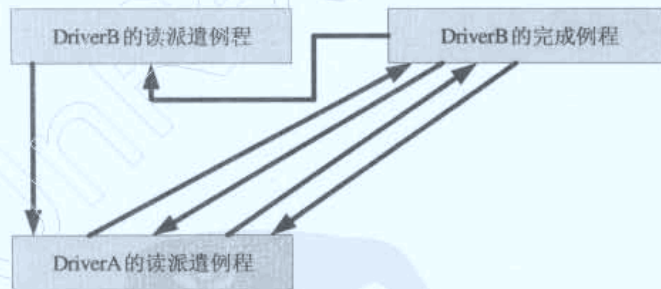


图 12-9 完成例程与派遣例程

上图中, DriverB 的派遣函数和 DriverB 的完成例程会反复多次调用 DriverA, 并多次调用 DriverA 的派遣例程。应用程序的读请求, 首先会来到 DriverB 的派遣函数, 由于读取的字节超过 1024 字节, 所以先向 DriverA 请求 1024 字节的读请求, 并将剩下的字节作为参数传递给 DriverB 的完成例程。

当 DriverA 的派遣例程操作完毕, 会调用 IoCompleteRequest。而因为上层堆栈注册了完成例程, 因此, IoCompleteRequest 在向上层堆栈回卷的时候, 会停止转而执行 DriverB 完成例程。

在 DriverB 的完成例程中会重新利用原先的 IRP，并根据未读取的字节数，向底层 (DriverA) 驱动发送读请求，因此会再次进入 DriverA 的派遣例程。这样周而复始，直到将所有数据读取完毕，回到 DriverB 的派遣例程。到此，整个 DriverB 的派遣例程结束，也标志着整个读取的结束。

在编写派遣函数之前要定义一种数据结构，这种数据结构主要是为了记录每次将 IRP 发往底层设备时，已经传送了多少字节、下次传输读取的起始地址。另外由于是直接读取，虚拟地址还会存储在 MDL 中，以下为这种数据结构的定义，其会作为参数传递给完成例程：

```
typedef struct _MyDriver_RW_CONTEXT
{
    PMDL          NewMdl;           //新 MDL
    PMDL          PreviousMdl;      //旧 MDL
    ULONG         Length;           //剩下没有读取的字节
    ULONG         Numxfer;          //已经传送过的字节数
    ULONG_PTR     VirtualAddress;    //后续传送的虚拟地址
    PDEVICE_EXTENSION DeviceExtension; //链接到设备扩展
} MYDRIVER_RW_CONTEXT, * PMYDRIVER_RW_CONTEXT;
```

以下是 DriverB 的派遣函数：

```
#001 #pragma PAGEDCODE
#002 NTSTATUS HelloDDKRead(IN PDEVICE_OBJECT pDevObj,
#003                        IN PIRP pIrp)
#004 {
#005     KdPrint(("DriverB:Enter B HelloDDKRead\n"));
#006     NTSTATUS status = STATUS_SUCCESS;
#007     //获得设备扩展
#008     PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
#009         pDevObj->DeviceExtension;
#010     //获得当前 I/O 堆栈
#011     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
#012
#013     ULONG totalLength;
#014     ULONG stageLength;
#015     PMDL mdl;
#016     PVOID virtualAddress;
#017     PMYDRIVER_RW_CONTEXT rwContext = NULL;
#018     PIO_STACK_LOCATION nextStack;
#019     //判断 MDL 是否为空
#020     if (!pIrp->MdlAddress)
#021     {
#022         status = STATUS_UNSUCCESSFUL;
#023         totalLength = 0;
#024         goto HelloDDKRead_EXIT;
#025     }
#026
#027     //获取 MDL 的虚拟地址
#028     virtualAddress = MmGetMdlVirtualAddress(pIrp->MdlAddress);
#029     //获取 MDL 的长度
#030     totalLength = MmGetMdlByteCount(pIrp->MdlAddress);
#031     KdPrint(("DriverB: (pIrp->MdlAddress)MmGetMdlVirtualAddress:%08X\n",
MmGetMdlVirtualAddress(pIrp->MdlAddress)));
```


Windows 驱动开发技术详解

```
#032    KdPrint(("DriverB: (pIrp->MdlAddress)MmGetMdlByteCount:%d\n",
MmGetMdlByteCount(pIrp->MdlAddress));
#033
#034    //将总的传输，分成几个阶段，这里设定每个阶段的长度
#035    if(totalLength > MAX_PACKAGE_SIZE)
#036    {
#037        stageLength = MAX_PACKAGE_SIZE;
#038    }else
#039    {
#040        stageLength = totalLength;
#041    }
#042
#043    //创建新的 MDL
#044    mdl = IoAllocateMdl((PVOID) virtualAddress,
#045                        totalLength,
#046                        FALSE,
#047                        FALSE,
#048                        NULL);
#049
#050    KdPrint(("DriverB: (new mdl)MmGetMdlVirtualAddress:%08X\n", MmGetMdl
VirtualAddress(mdl)));
#051    KdPrint(("DriverB: (new mdl)MmGetMdlByteCount:%d\n", MmGetMdlByteCount
(mdl)));
#052    //判断 MDL 是否为空
#053    if(mdl == NULL)
#054    {
#055        KdPrint(("DriverB:Failed to alloc mem for mdl\n"));
#056        status = STATUS_INSUFFICIENT_RESOURCES;
#057        goto HelloDDKRead_EXIT;
#058    }
#059
#060    //将 IRP 的 MDL 做重新映射
#061    IoBuildPartialMdl(pIrp->MdlAddress,
#062                    mdl,
#063                    (PVOID) virtualAddress,
#064                    stageLength);
#065    KdPrint(("DriverB: (new mdl)MmGetMdlVirtualAddress:%08X\n", MmGetMdl
VirtualAddress(mdl)));
#066    KdPrint(("DriverB: (new mdl)MmGetMdlByteCount:%d\n", MmGetMdlByteCount
(mdl)));
#067    //分配非分页内存
#068    rwContext = (PMYDRIVER_RW_CONTEXT)
#069                ExAllocatePool(NonPagedPool, sizeof(MYDRIVER_RW_CONTEXT));
#070
#071    rwContext->NewMdl = mdl;
#072    rwContext->PreviousMdl = pIrp->MdlAddress;
#073    rwContext->Length = totalLength - stageLength; //还剩下多少没读取
#074    rwContext->Numxfer = 0; //读了多少字节
#075    rwContext->VirtualAddress= ((ULONG_PTR)virtualAddress + stageLength);
//下一阶段开始读取的地址
#076    rwContext->DeviceExtension= pDevExt;
#077
#078    //复制到底层堆栈
#079    IoCopyCurrentIrpStackLocationToNext(pIrp);
#080    //得到下一层 I/O 堆栈
#081    nextStack = IoGetNextIrpStackLocation(pIrp);
#082    //根据底层驱动的实现，底层驱动有可能读取这个数值，也有可能读取 mdl 的 length。
#083    nextStack->Parameters.Read.Length = stageLength;
#084
```

```

#085     pIrp->MdlAddress = mdl;
#086
#087     //设定完成例程
#088     IoSetCompletionRoutine(pIrp,
#089                             (PIO_COMPLETION_ROUTINE)HelloDDKReadCompletion,
#090                             rwContext,
#091                             TRUE,
#092                             TRUE,
#093                             TRUE);
#094     //调用底层驱动
#095     IoCallDriver(pDevExt->TargetDevice,pIrp);
#096     //设置 MDL
#097     pIrp->MdlAddress = rwContext->PreviousMdl;
#098     //释放 MDL
#099     IoFreeMdl(rwContext->NewMdl);
#100
#101     HelloDDKRead_EXIT:
#102     //设置 IRP 完成状态
#103     pIrp->IoStatus.Status = status;
#104     //设置 IRP 操作字节数
#105     pIrp->IoStatus.Information = totalLength;
#106     //结束 IRP 请求
#107     IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#108     KdPrint(("DriverB:Leave B HelloDDKRead\r\n"));
#109     return status;
#110 }

```

此段代码可以在配套光盘中本章的 Test5 目录下找到。

12.3.4 完成例程

当派遣函数将第一个 1024 字节的读请求发送给底层驱动，并且将该请求完成后，会进入完成例程。完成例程会根据传递进来的参数，知道还有多少字节没有传送，下一次传送的地址是多少。完成例程根据这些信息，重新将 IRP 再次转发到底层驱动。

以下是 DriverB 的完成例程：

```

#001 #pragma PAGEDCODE
#002 NTSTATUS
#003 HelloDDKReadCompletion(
#004     IN PDEVICE_OBJECT DeviceObject,
#005     IN PIRP Irp,
#006     IN PVOID Context
#007 )
#008 {
#009     KdPrint(("DriverB:Enter B HelloDDKReadCompletion\n"));
#010     //获得上下文
#011     PMYDRIVER_RW_CONTEXT rwContext = (PMYDRIVER_RW_CONTEXT) Context;
#012     NTSTATUS ntStatus = Irp->IoStatus.Status;
#013
#014     ULONG stageLength;
#015     //判断是否成功得到上下文
#016     if(rwContext && NT_SUCCESS(ntStatus))
#017     {
#018         //已经传送了多少字节
#019         rwContext->Numxfer += Irp->IoStatus.Information;

```



```

#020
#021     if(rwContext->Length)
#022     {
#023         //设定下一阶段读取字节数
#024         if(rwContext->Length > MAX_PACKAGE_SIZE)
#025         {
#026             stageLength = MAX_PACKAGE_SIZE;
#027         }
#028         else
#029         {
#030             stageLength = rwContext->Length;
#031         }
#032         //重新利用 MDL
#033         MmPrepareMdlForReuse(rwContext->NewMdl);
#034         //重新设置 MDL
#035         IoBuildPartialMdl(Irp->MdlAddress,
#036                          rwContext->NewMdl,
#037                          (PVOID) rwContext->VirtualAddress,
#038                          stageLength);
#039         //设置上下文的虚拟地址
#040         rwContext->VirtualAddress += stageLength;
#041         //设置上下文的虚拟地址长度
#042         rwContext->Length -= stageLength;
#043         //将当前堆栈复制到底层堆栈
#044         IoCopyCurrentIrpStackLocationToNext(Irp);
#045         PIO_STACK_LOCATION nextStack = IoGetNextIrpStackLocation(Irp);
#046         //设置底层 I/O 堆栈参数
#047         nextStack->Parameters.Read.Length = stageLength;
#048         //设置完成例程
#049         IoSetCompletionRoutine(Irp,
#050                               HelloDDKReadCompletion,
#051                               rwContext,
#052                               TRUE,
#053                               TRUE,
#054                               TRUE);
#055         //调用底层驱动
#056         IoCallDriver(rwContext->DeviceExtension->TargetDevice,
#057                     Irp);
#058
#059         return STATUS_MORE_PROCESSING_REQUIRED;
#060     }
#061     else
#062     {
#063         //最后一次传输
#064         Irp->IoStatus.Information = rwContext->Numxfer;
#065     }
#066 }
#067 KdPrint(("DriverB:Leave B HelloDDKReadCompletion\n"));
#068 return STATUS_MORE_PROCESSING_REQUIRED;
#069 }

```

此段代码可以在配套光盘中本章的 Test5 目录下找到。

12.3.5 分析

将上节中编写的 DriverA 和 DriverB 先后加载，然后调用应用程序。通过前面的分析，应用程序发出的读请求本来是发往 DriverA 的，但由于 DriverB 挂载在 DriverA 之上，因

此读请求被 Driver B 的派遣函数所拦截，然后再将读请求发往底层驱动。

当 DriverA 和 DriverB 成功加载后，DriverB 创建的设备就会挂载在 DriverA 创建的设备上，通过 DDK 工具 Device Tree 可以清楚地看出两个驱动的关系，如图 12-10 所示。

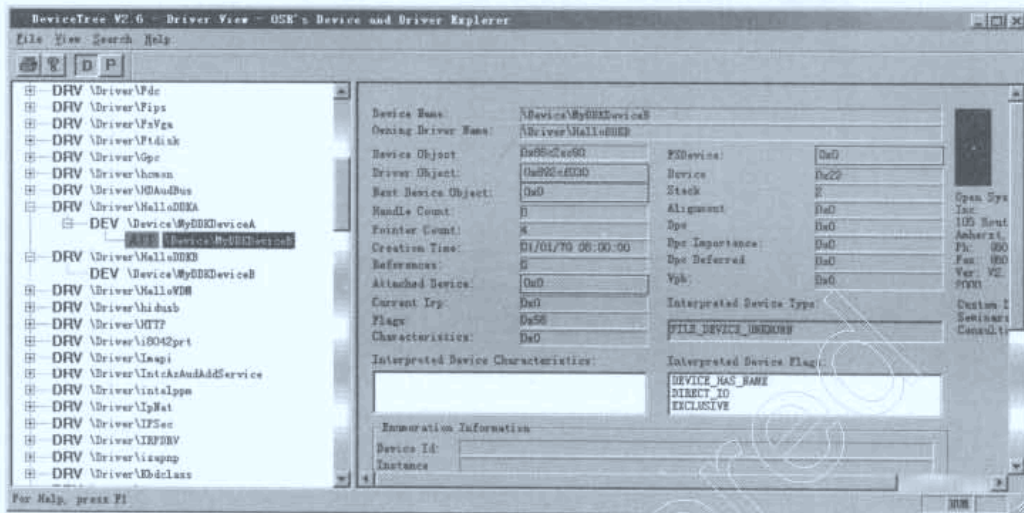


图 12-10 DeviceTree 查看设备栈

如图 12-11 所示为通过工具软件 IRPTrace 跟踪读 IRP 的全部过程，可以看出，读 IRP 被分作 4 次重复转发到底层驱动里。

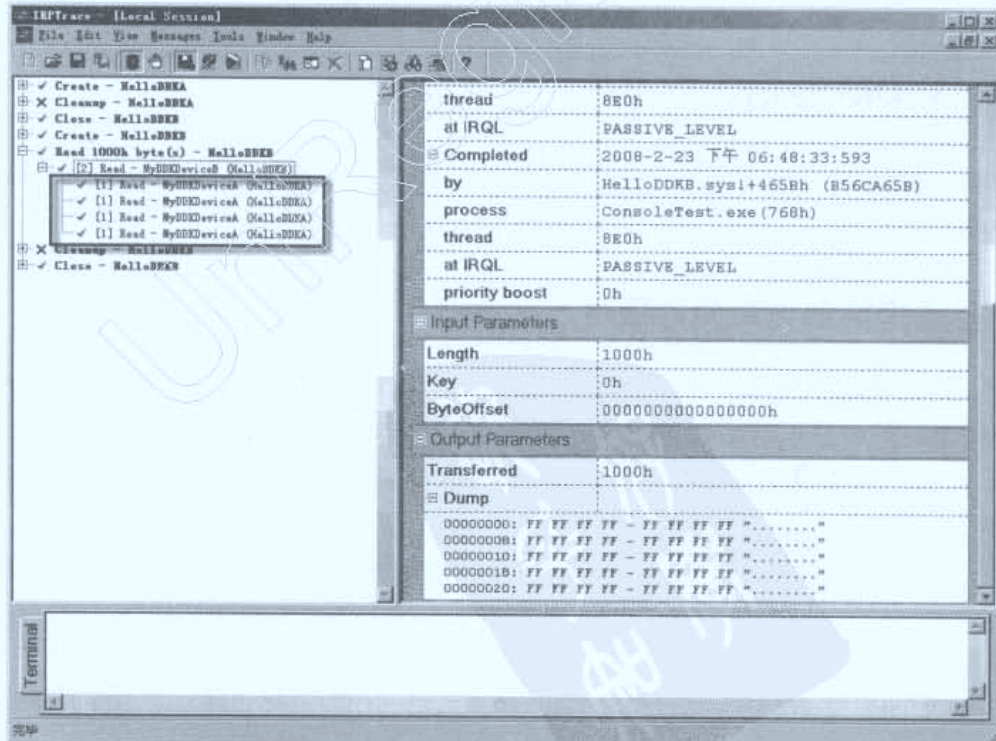


图 12-11 IRPTrace 跟踪 IRP

Windows 驱动开发技术详解

除此之外还可以通过 DebugView 查看两个驱动输出的 log 信息，如图 12-12 所示。读者可以将两个驱动输出 log 和 IRPTrace 跟踪 IRP 的结果，以进一步理解分层驱动模型。

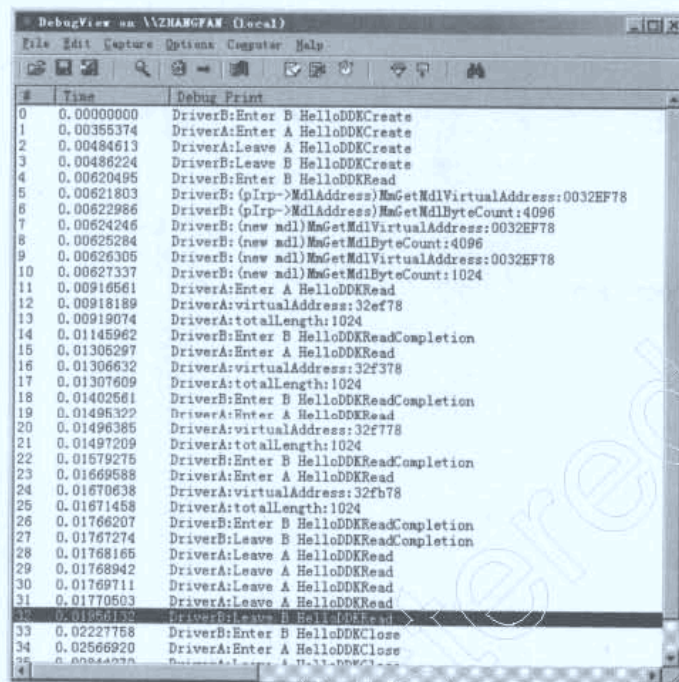


图 12-12 DebugView 查看 log

12.4 WDM 驱动程序架构

WDM 驱动是在 NT 式驱动概念之上发展而来的，它主要是对即插即用 IRP 的扩充。本节将对 WDM 驱动的架构进行介绍。

12.4.1 WDM 与分层驱动程序

在 Windows 2000 以后，微软公司提供了一种新型驱动模型，即 Windows 驱动模型 (WDM, Windows Driver Model)。其实这种新型的 WDM 驱动就是在分层的 NT 驱动之上发展而来的，如果读者已经对分层驱动有了较深入的了解，将会发现 WDM 模型原来是如此熟悉。

回忆第一章介绍的 HelloWDM 驱动，其就是一个最典型的 WDM 驱动。首先加载驱动的代码由 DriverEntry 变成两个，分别是 DriverEntry 和 AddDevice。其中 DriverEntry 依旧是作为入口函数，也就是驱动被加载的时候，首先调用 DriverEntry 函数。

但在 WDM 驱动中，DriverEntry 函数的功能被弱化，而创建设备等功能都被转移到 AddDevice 例程。AddDevice 例程是 WDM 驱动与老式 NT 式驱动的重要区别。

AddDevice 的函数如下所示，它是由 Windows 的即插即用部件负责调用的。

```
#001 #pragma PAGEDCODE
#002 NTSTATUS HelloWDMAddDevice(IN PDRIVER_OBJECT DriverObject,
#003                               IN PDEVICE_OBJECT PhysicalDeviceObject)
#004 {
#005     ...
#006 }
```

其中第二个参数 PhysicalDeviceObject 是一个设备对象的指针，这个设备对象是在 AddDevice 例程被调用时就已经创建了的设备对象，是物理设备对象。程序员编写的 AddDevice 例程会创建一个新的设备对象，并“挂载”在这个物理设备对象之上。

这个物理设备对象同样是由某个驱动创建，但该驱动绝大多数都是由微软公司提供的，针对不同总线的驱动，如虚拟总线驱动、PCI 总线驱动、USB 总线驱动。这些总线驱动将这种总线的基本实现了通用总线设备的抽象，并完成了对诸如电源处理、即插即用等处理。

此时程序员编写的驱动，就会将通用的处理转发给总线驱动，而只处理特殊的服务，因此程序员的工作量被大大简化了。例如即插即用的处理大多数都由底层的总线驱动完成。

12.4.2 WDM 的加载方式

另外，WDM 驱动加载的方式有别于老式 NT 驱动。老式的驱动不支持即插即用 (PNP) 功能。例如，在 Windows NT 操作系统上，当新设备插入计算机后，Windows 操作系统并不知道有新的硬件插入。而此时需要用户手动为该设备选择设备驱动。

而在 Windows 2000 以后，全面支持 WDM 驱动。当有设备插入电脑后，系统总线驱动（根总线）会枚举到有新设备被插入，这时候会通知 PNP 管理器寻找需要加载的设备驱动。

首先根据此种设备的总线，PNP 管理器会加载相应的总线驱动设备，并得到这种物理设备对象 (PDO)。

每个设备都有自己设备的 ProductID 和 VendorID 等信息，每种设备的这些信息是不同的，也就是说一种设备的驱动对应着一种 ProductID 和 VendorID。每个设备都是根据这两个信息选择需要加载的设备驱动程序。

PNP 管理器会根据 ProductID 和 VendorID 等信息加载这种设备的驱动，并将刚才创建好的 PDO 作为参数，传递给 AddDevice 例程。而 AddDevice 例程会将自己的设备对象挂载在 PDO 之上。至此设备栈创建完毕。

在某些时候，在物理设备对象和功能设备对象之间，会有一层或多层过滤驱动，从而形成一个负载的设备栈，如图 12-13 所示。

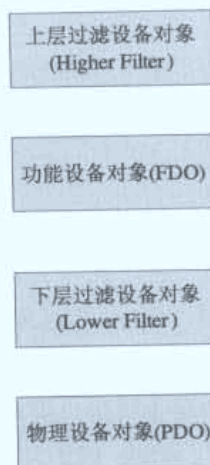


图 12-13 WDM 驱动模型

12.4.3 功能设备对象

功能设备对象，即 Function Device Object (FDO)，是 `DEVICE_OBJECT` 数据结构。这种设备对象一般由 `AddDevice` 例程创建，然后加载在物理设备对象之上。和老式的 NT 驱动一样，功能设备对象所属的驱动也会在 `DriverEntry` 里设置各个派遣函数的入口，例如，处理 `IRP_MJ_CREATE`、`IRP_MJ_CLOSE`、`IRP_MJ_WRITE`、`IRP_MJ_READ` 等 `IRP` 的派遣函数。

在处理的各个 `IRP` 的派遣函数中，有一种新的 `IRP` 被引进过来，这就是 `IRP_MJ_PNP`，这种 `IRP` 用来支持即插即用的功能。

`IRP_MJ_PNP` 一般不是来自于应用程序，而是来自于 I/O 管理器。在设备被插入、拔出、系统加载等情况下，I/O 管理器就会向这个设备发出各种 `IRP_MJ_PNP`，从而触发 `IRP_MJ_PNP` 的派遣函数。

`IRP_MJ_PNP` 的派遣函数一般会将 `IRP` 转发到底层的物理设备对象。底层的物理设备对象由总线驱动所创建，而总线驱动是由微软公司所提供，它封装了对即插即用功能的处理。这样就能很好地解决即插即用问题，并且在每次退出新版本的 Windows 时，总线驱动都会或多或少地进行修正，而功能驱动则不用进行修改，或进行很小的修改就可以在新版本的 Windows 上使用。

12.4.4 物理设备对象

物理设备对象是由微软公司提供的总线设备驱动所创建的，它完成了即插即用、电源管理等 `IRP`。在 Windows 中，有多个总线驱动，分别是 PCI 总线驱动、USB 总线驱动、ISA 总线驱动、虚拟总线驱动等。

第一章中介绍的 HelloWDM 驱动，其底层的 pdo 就是一个虚拟总线驱动，如图 12-14 所示，这是用一个叫做 Device Object Viewer 的工具，读者可以从网上下载该工具软件。

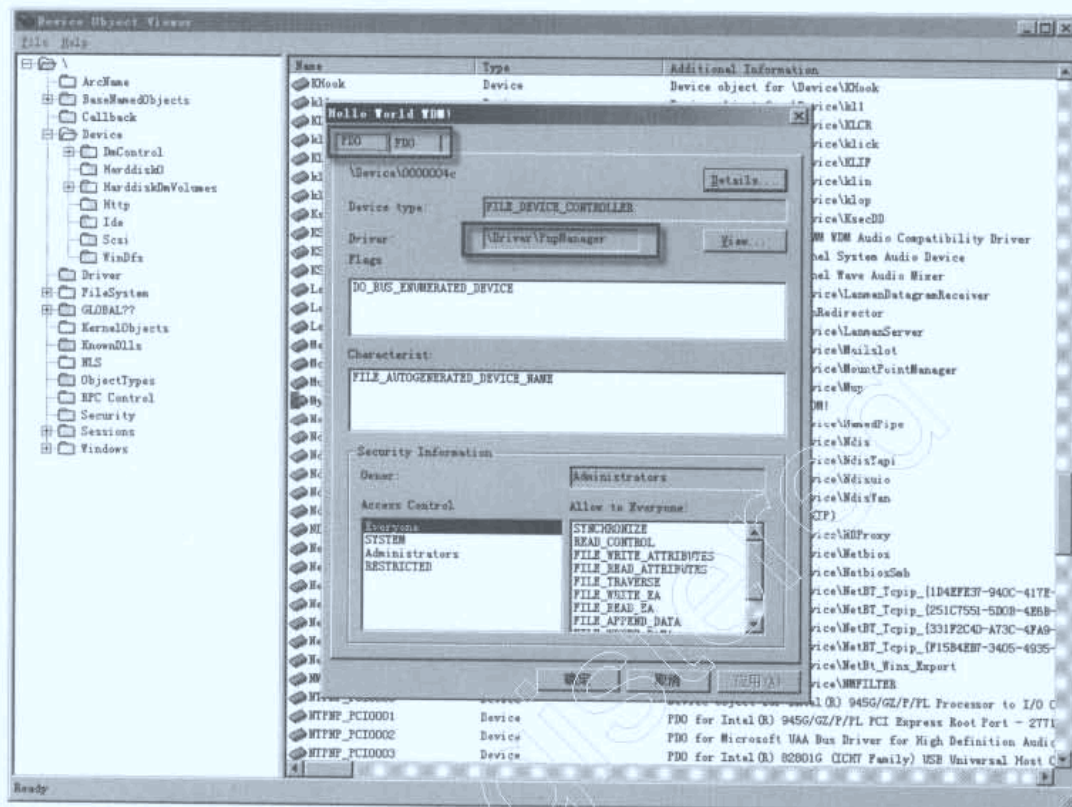


图 12-14 Device Object View 查看设备 FDO 与 PDO

HelloWDM 的例子在 WDM 驱动模型中应该算是最简单的，很多情况要比这个复杂得多。如图 12-15 所示为复杂的 WDM 模型。在大多数的 PC 中，PCI 总线是作为根部的总线。例如，ISA 总线、USB 总线整体可以理解为 PCI 总线上的一个设备存在。

当系统启动时，根总线驱动被加载，然后寻找挂载于根总线上的驱动设备。如果发现有 PCI 设备，就会加载 PCI 设备的 PDO，然后寻找适合的 FDO 进行加载。PCI-ISA 桥、USB 适配器会当做是 PCI 设备，被加载 PDO、FDO 等。

当 USB 适配器的驱动被成功加载后，它会枚举挂载在 USB 总线上的所有设备，并为之加载 PDO 和 FDO，如果有适合的 FDO，Windows 会提示用户安装一个合适的 FDO。

如图 12-15 所示，Windows 设备启动后，根总线驱动会生长成一个“倒挂”的树形结构，例如 USB 总线就是挂载在 PCI 总线上的。绝大多数的总线设备驱动都是由微软公司随着 Windows 提供的，单用户也可以定义自己的总线，例如，在 DDK 里的 toaster 的例子，它就是自己定义了一种总线驱动，并挂载在根总线驱动之上。

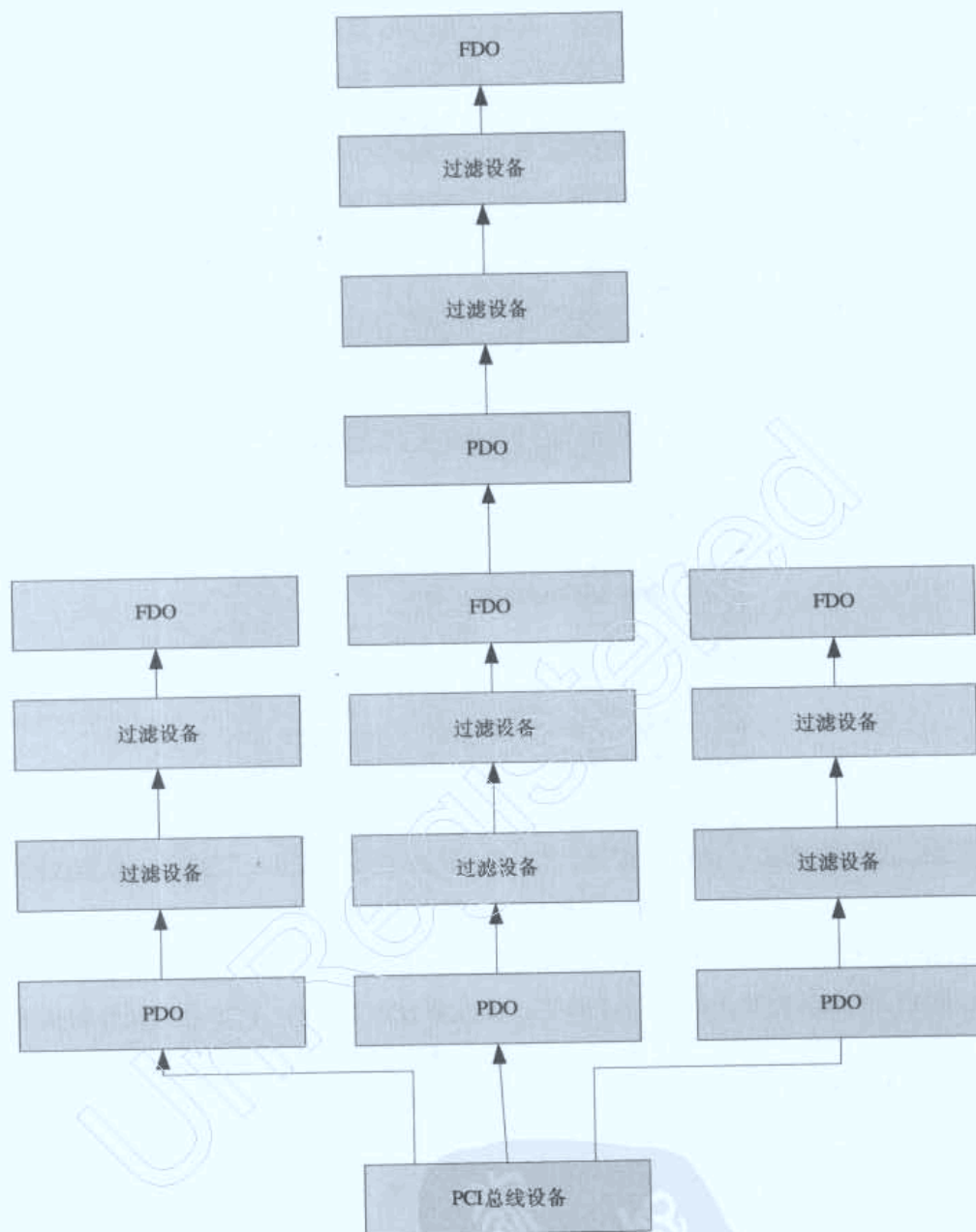


图 12-15 设备的复杂结构

12.4.5 物理设备对象与即插即用

物理设备对象一般都是由总线驱动所创建的设备，作为最底层的设备，它很好地支持了 PNP 功能。而程序员编写驱动只需要把精力重点放在如何操作设备功能上，把针对即插即用或者电源管理的问题都抛给底层的 PDO 处理，而 PDO 会很好地处理这些问题。

另外，在没有即插即用功能以前，某个板卡插入计算机后，其 I/O 空间地址、物理地址空间、中断号等资源都是固定的，这些固定的地址是在驱动程序中写死的。而在支持即插即用的驱动程序中，PDO 会根据总线枚举的情况，自动为设备分配合理的 I/O 空间地址、物理地址空间、中断号等资源信息，并且上报给 FDO。

而且，在 PDO 和 FDO 之间可以加入一个或多个过滤驱动，过滤驱动可以从中截取并修改即插即用相关的 IRP。

12.5 小结

本章主要介绍了分层驱动的概念。分层驱动可以将功能复杂的驱动程序分解为多个功能简单的驱动程序。多个分层的驱动程序形成一个设备堆栈，IRP 请求首先发送到设备堆栈的顶层，然后依次穿越每层的设备堆栈，最终完成 IRP 请求。

另外，WDM 驱动模型是基于分层驱动的概念发展而来的。WDM 驱动一般分为总线驱动程序和功能驱动程序。功能驱动程序挂载在总线驱动程序之上，功能驱动程序处理一部分 IRP，大部分的即插即用 IRP 和电源处理 IRP 被转发到总线驱动处理。

第 13 章 让设备实现即插即用

本章首先介绍即插即用的概念和驱动程序支持即插即用功能的必要性。另外，本章还介绍如何利用 WDM 驱动程序开发框架设计支持即插即用功能的驱动程序。

支持即插即用功能是 Windows 2000 及后续版本中 Windows 的重要特性，为此在 Windows 内核中专门有一个组件（即插用管理器），负责和 WDM 驱动程序配合来支持即插即用功能。

13.1 即插即用概念

现代 PC 上的设备基本都支持即插即用功能。即插即用功能能够通过操作系统协调自动分配设备上的资源，例如，中断号、I/O 地址、DMA 通道、设备物理内存等。

13.1.1 历史原因

在 PC 发展初期，PC 上的硬件资源需要用户手动配置，例如，中断号、I/O 地址、DMA 通道、设备物理地址等。只有正确配置这些硬件资源，用户才能让设备运行正常。然而，随着 PC 越来越普及，大部分用户是非专业人员，并且 PC 上的硬件也越来越层出不穷，因此用户经常面临一个问题，就是没有正确设置好 PC 上的硬件，导致两个或多个设备硬件冲突。

另外，安装新硬件必须关闭计算机，并且手动为硬件安装驱动程序，并为设备配置所需的硬件资源，PC 用户不得不知道所有硬件相关情况，并使它们不冲突。

随着 Windows 95 的出现和部分硬件厂商的支持，大多数新设备支持硬件资源的自动配置，而且有些设备支持热插拔，并可以不重启就自动配置。

Windows 95 因此取得了巨大成功，但毕竟 Windows 95、Windows 98 等并不是纯粹的 32 位操作系统，微软随后便推出了纯 32 位操作系统的 Windows 2000，它是基于 Windows

NT 架构，并引入了 WDM 驱动模型。WDM 驱动模型主要就是帮助开发驱动程序的程序员轻松简单地开发支持即插即用、电源管理等高级特性的驱动程序。

13.1.2 即插即用的目标

即插即用体系结构的主要目标是不需要用户介入，而直接对设备安装、拆除等操作进行自动配置。基于这个目标，即插即用有以下几种特征：

(1) 系统能够检测到新设备的插入，也能检测到设备被拔出。例如，当用户在 PCI 插槽处新插入一个 PCI 设备后，当下次开机时，系统会检测到有新硬件加载，并自动搜寻可安装的驱动程序。

(2) 如果总线接口允许，设备可以实现热插拔，并保证系统可以正常工作。例如 USB 设备，随着设备的插入，系统会自动加载其驱动，并且保证 USB 总线上的其他设备能正常工作。另外当系统运行时，拔掉 USB 设备不会引起系统的崩溃。

(3) 设备允许软件配置。硬件设备上的 I/O 端口资源、设备物理地址、DMA 通道、中断号可以接受软件配置，也就是操作系统可以更改它们。而在以前的 PC 中，更改这些设置都是依靠有经验的用户手动修改主板跳线（DIP 开关）来完成。

(4) 操作系统应该知道哪些是正确的驱动程序，并智能地加载。

13.1.3 Windows 中即插即用相关组件

在 Windows 2000 中是通过以下几个组件的相互配合，从而实现即插即用复杂的功能的，如图 13-1 所示。

(1) 即插即用管理程序：该程序包括两部分，即用户模式部分和内核模式部分。用户模式部分与应用接口组件相互允许交互程序查询和更改已安装的即插即用软件的配置。内核模式部分与硬件和其他内核模式组件交互管理硬件的正确检测和配置。

(2) 电源管理程序：该程序简化了设备电源的管理。依据它的特性对于一个在很长一段时间内不用的设备，可以暂时为它断开电源。该组件识别电源事件，并把它传递到合适的驱动程序。

(3) 注册表：Windows 2000 注册表维护已经安装的硬件和即插即用设备软件的数据库。注册表的内容帮助驱动程序和其他组件识别和定位设备使用的资源。

(4) INF 文件：每种设备必须由控制驱动程序安装期间使用的一个文件完全描述。每种设备、驱动程序组合必须提供一个专门格式化的 INF 文件。

(5) WDM 驱动程序：支持 WDM 框架的驱动程序，很方便地支持即插即用功能。这是因为 WDM 框架程序都是分层驱动，WDM 处于分层的高端，而总线驱动程序处于分层的低端。大部分的即插即用 IRP 都会被 WDM 驱动发送到底层的驱动程序处理。

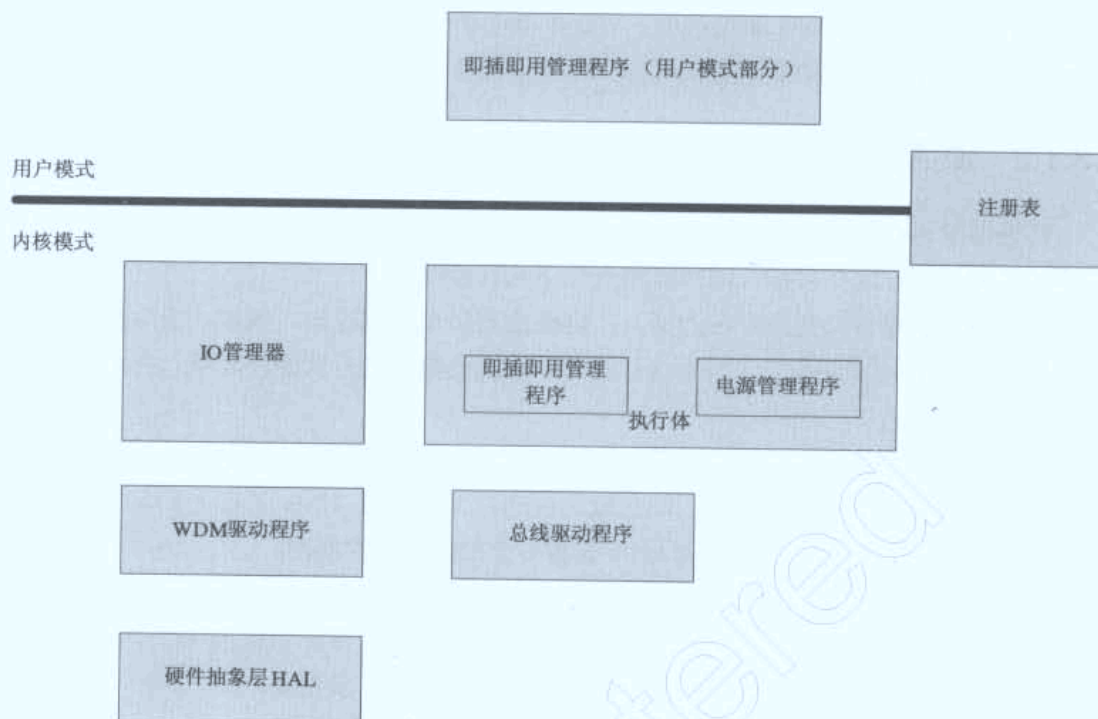


图 13-1 即插即用相关组件

（6）总线驱动程序：总线驱动程序主要负责处理总线通用的操作，并在此基础上加入了即插即用的功能。

13.1.4 遗留驱动程序

在 Windows 2000 以后的各个 Windows 版本中，除了 WDM 驱动，还大量存在着 NT 式驱动程序，这些驱动不实现即插即用功能，被称为遗留驱动程序，比较常见的有串口驱动程序、并口驱动程序等。

13.2 即插即用 IRP

即插即用 IRP 即 IRP_MJ_PNP，它一般是由即插即用管理器发送给 WDM 驱动程序的（NT 式驱动程序不会接到 IRP_MJ_PNP 的 IRP）。

不同情况下，即插即用管理器会发送不同子类型的 IRP_MJ_PNP IRP。例如，设备启动时，就会发送 IRP_MN_START_DEVICE 给 WDM 驱动程序，而设备被突然拔出后，会发送 IRP_MN_SURPRISE_REMOVAL 给 WDM 驱动程序。本节主要介绍 IRP_MJ_PNP 和它的派遣函数。

13.2.1 即插即用 IRP 的功能代码

IRP_MJ_PNP 类似于 Win32 编程中的 Windows 消息，而 IRP_MJ_PNP 的派遣函数则类似与 Win32 编程中的窗口函数。

所有的即插即用的事件，例如插入或拔出设备，都会引发即插即用管理器向 WDM 驱动程序发送一个 IRP_MJ_PNP IRP。IRP_MJ_PNP 是这个 IRP 的主功能代码，不同的即插即用事件会有不同的子功能代码。例如设备被突然拔出，其子功能代码就是 IRP_MN_SURPRISE_REMOVAL。如表 13-1 所示，列出了 WDM 驱动程序中会用到的子功能代码。

表 13-1 即插即用子功能代码

子功能代码	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

13.2.2 处理即插即用 IRP 的派遣函数

和其他派遣函数一样，IRP_MJ_PNP 派遣函数首先需要注册一下。驱动程序的派遣函

Windows 驱动开发技术详解

数需要在 DriverEntry 中注册。

```
#001 extern "C" NTSTATUS DriverEntry(IN PDRIVER_OBJECT pDriverObject,  
#002                                     IN PUNICODE_STRING pRegistryPath)  
#003 {  
#004     KdPrint(("Enter DriverEntry\n"));  
#005     //设置 AddDevice 例程  
#006     pDriverObject->DriverExtension->AddDevice = HelloWDMAddDevice;  
#007     //设置 IRP 派遣函数  
#008     pDriverObject->MajorFunction[IRP_MJ_PNP] = HelloWDMpnp;  
#009     pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =  
#010     pDriverObject->MajorFunction[IRP_MJ_CREATE] =  
#011     pDriverObject->MajorFunction[IRP_MJ_READ] =  
#012     pDriverObject->MajorFunction[IRP_MJ_WRITE] = HelloWDMDispatchRoutine;  
#013     //设置卸载例程  
#014     pDriverObject->DriverUnload = HelloWDMUnload;  
#015     KdPrint(("Leave DriverEntry\n"));  
#016     return STATUS_SUCCESS;  
#017 }
```

此段代码可以在配套光盘中本章的 Test1 目录下找到。

上面代码中，将 IRP_MJ_PNP 的派遣函数设置成了 HelloWDMpnp 函数。在 IRP_MJ_PNP 派遣函数中要处理不同子功能代码的 IRP，最简单的办法是采用一个巨大的 switch 和 case 语句组成的判别，但是这样会使该派遣函数变得很大。

本例采用函数指针的办法，首先初始化一个函数指针组成的数组，然后在派遣函数中判断是哪种子功能代码。根据这个子功能代码去寻找相应的函数指针，再通过指针找到针对具体子功能代码所做的操作函数。

```
#001 #pragma PAGEDCODE  
#002 NTSTATUS HelloWDMpnp(IN PDEVICE_OBJECT fdo,  
#003                       IN PIRP Irp)  
#004 {  
#005     PAGED_CODE{};  
#006  
#007     KdPrint(("Enter HelloWDMpnp\n"));  
#008     //设置派遣函数状态  
#009     NTSTATUS status = STATUS_SUCCESS;  
#010     //获得设备扩展  
#011     PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;  
#012     //获得当前 I/O 堆栈  
#013     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);  
#014     static NTSTATUS (*fctab[])(PDEVICE_EXTENSION pdx, PIRP Irp) =  
#015     {  
#016         DefaultPnpHandler,           // IRP_MN_START_DEVICE  
#017         DefaultPnpHandler,           // IRP_MN_QUERY_REMOVE_DEVICE  
#018         HandleRemoveDevice,          // IRP_MN_REMOVE_DEVICE  
#019         DefaultPnpHandler,           // IRP_MN_CANCEL_REMOVE_DEVICE  
#020         DefaultPnpHandler,           // IRP_MN_STOP_DEVICE  
#021         DefaultPnpHandler,           // IRP_MN_QUERY_STOP_DEVICE  
#022         DefaultPnpHandler,           // IRP_MN_CANCEL_STOP_DEVICE  
#023         DefaultPnpHandler,           // IRP_MN_QUERY_DEVICE_RELATIONS  
#024         DefaultPnpHandler,           // IRP_MN_QUERY_INTERFACE  
#025         DefaultPnpHandler,           // IRP_MN_QUERY_CAPABILITIES  
#026         DefaultPnpHandler,           // IRP_MN_QUERY_RESOURCES
```

```

#027     DefaultPnpHandler,           // IRP_MN_QUERY_RESOURCE_REQUIREMENTS
#028     DefaultPnpHandler,           // IRP_MN_QUERY_DEVICE_TEXT
#029     DefaultPnpHandler,           // IRP_MN_FILTER_RESOURCE_REQUIREMENTS
#030     DefaultPnpHandler,
#031     DefaultPnpHandler,           // IRP_MN_READ_CONFIG
#032     DefaultPnpHandler,           // IRP_MN_WRITE_CONFIG
#033     DefaultPnpHandler,           // IRP_MN_EJECT
#034     DefaultPnpHandler,           // IRP_MN_SET_LOCK
#035     DefaultPnpHandler,           // IRP_MN_QUERY_ID
#036     DefaultPnpHandler,           // IRP_MN_QUERY_PNP_DEVICE_STATE
#037     DefaultPnpHandler,           // IRP_MN_QUERY_BUS_INFORMATION
#038     DefaultPnpHandler,           // IRP_MN_DEVICE_USAGE_NOTIFICATION
#039     DefaultPnpHandler,           // IRP_MN_SURPRISE_REMOVAL
#040 };
#041
#042     ULONG fcn = stack->MinorFunction;
#043     if (fcn >= arraysize(fcntab))
#044     {
#045         // 未知的子功能代码
#046         status = DefaultPnpHandler(pdx, Irp);
#047     }
#048     //如果是调试版本则打印调试信息
#049     #if DBG
#050     static char* fcname[] =
#051     {
#052         "IRP_MN_START_DEVICE",
#053         "IRP_MN_QUERY_REMOVE_DEVICE",
#054         "IRP_MN_REMOVE_DEVICE",
#055         "IRP_MN_CANCEL_REMOVE_DEVICE",
#056         "IRP_MN_STOP_DEVICE",
#057         "IRP_MN_QUERY_STOP_DEVICE",
#058         "IRP_MN_CANCEL_STOP_DEVICE",
#059         "IRP_MN_QUERY_DEVICE_RELATIONS",
#060         "IRP_MN_QUERY_INTERFACE",
#061         "IRP_MN_QUERY_CAPABILITIES",
#062         "IRP_MN_QUERY_RESOURCES",
#063         "IRP_MN_QUERY_RESOURCE_REQUIREMENTS",
#064         "IRP_MN_QUERY_DEVICE_TEXT",
#065         "IRP_MN_FILTER_RESOURCE_REQUIREMENTS",
#066         "",
#067         "IRP_MN_READ_CONFIG",
#068         "IRP_MN_WRITE_CONFIG",
#069         "IRP_MN_EJECT",
#070         "IRP_MN_SET_LOCK",
#071         "IRP_MN_QUERY_ID",
#072         "IRP_MN_QUERY_PNP_DEVICE_STATE",
#073         "IRP_MN_QUERY_BUS_INFORMATION",
#074         "IRP_MN_DEVICE_USAGE_NOTIFICATION",
#075         "IRP_MN_SURPRISE_REMOVAL",
#076     };
#077
#078     KdPrint(("PNP Request (%s)\n", fcname[fcn]));
#079     #endif
#080     //进入 IRP 相应的处理函数
#081     status = (*fcntab[fcn])(pdx, Irp);
#082     KdPrint(("Leave HelloWDMpnp\n"));
#083     return status;
#084 }

```

此段代码可以在配套光盘中本章的 Test1 目录下找到。

13.3 通过设备接口寻找设备

在 WDM 驱动程序中，一般都是通过设备接口来定位一个驱动程序的。这有别于 NT 式驱动用设备名或者用符号链接来定位驱动程序。当然，为了兼容 NT 式驱动程序，WDM 驱动同样可以使用设备名、符号链接等定位设备。

13.3.1 设备接口

在 WDM 驱动中，一般很少用到设备名，也很少用到符号链接，而是用一个设备接口来指定设备。设备接口其实就是一组全局标识，它是一个 128 位组成的数字，并能保证在全世界范围内不会产生冲突。

引入设备接口的主要原因是避免设备名冲突，例如，不同的网卡厂商可能都将自己的设备名命名为“NetCardDevice”，当用户插入两块网卡时，就会引起命名冲突。而引入设备接口的概念，各个网卡厂商用自己定义的 128 位数字指定自己的设备，从小概率意义上，保证了各个设备的设备接口不会一样。

在 VC 的工具中有一个创建 GUID 的工具，叫 guidgen.exe，读者可以在命令行方式下键入 guidgen 进入该工具，如图 13-2 所示。

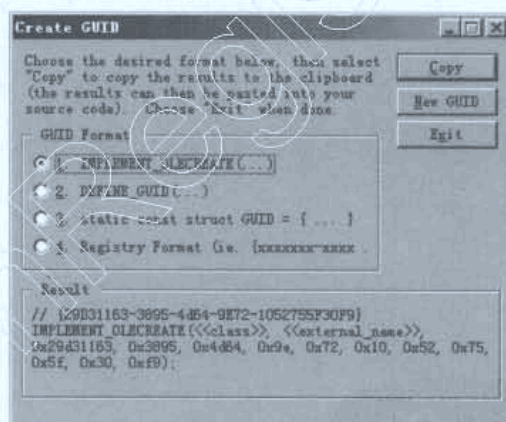


图 13-2 GUID 生成器

Guidgen.exe 为用户提供四种方式产生 guid，其实它们都是 128 位的数字，只是输出的形式不同而已，一般选择第二种。单击“New GUID”按钮会产生新的 GUID，单击“Copy”按钮，会将产生的 GUID 复制到内存里，在程序中“粘贴”即可。

```
// {C4D47B37-E8B2-48fa-B383-794F2C5EBD3B}
DEFINE_GUID(<<name>>,
0xc4d47b37, 0xe8b2, 0x48fa, 0xb3, 0x83, 0x79, 0x4f, 0x2c, 0x5e, 0xbd, 0x3b);
```

用 GUIDGEN.exe 产生的 GUID 从概率上可以保证全世界已存在的 GUID 不一样。

13.3.2 WDM 驱动中设置接口

在 WDM 驱动程序中使用设备接口，首先在创建设备的时候，不能指定设备名，这样系统会为设备自动创建一个设备名，该设备名就是一个数字，新设备加入的时候，数字依次递增。

另外不用 `IoCreateSymbolicLink` 创建设备，而是用 `IoRegisterDeviceInterface` 为设备创建设备链接。前面讲过设备链接是暴露给应用程序的，也就是应用程序可以通过符号链接来寻找到设备。而使用符号链接可以代替符号链接的作用，后面的内容中会告诉读者如何通过设备接口来寻找设备。

`IoRegisterDeviceInterface` 的声明如下：

```
NTSTATUS
IoRegisterDeviceInterface(
    IN PDEVICE_OBJECT PhysicalDeviceObject,
    IN CONST GUID *InterfaceClassGuid,
    IN PUNICODE_STRING ReferenceString OPTIONAL,
    OUT PUNICODE_STRING SymbolicLinkName
);
```

- `PhysicalDeviceObject`：物理设备对象，也就是 pdo。
- `InterfaceClassGuid`：设备接口的指针，这里是 GUID 类型的指针。
- `ReferenceString`：很少用，一般设置为 NULL。
- `SymbolicLinkName`：将 GUID 输出一串 UNICODE 字符串。
- 返回值：指明是否成功创建设备接口。

以下的代码是在 `AddDevice` 例程中，创建设备并创建设备接口，读者可以比较用符号链接和用设备接口的细微区别。

```
#001 #pragma PAGEDCODE
#002 NTSTATUS HelloWDMAddDevice(IN PDRIVER_OBJECT DriverObject,
#003                               IN PDEVICE_OBJECT PhysicalDeviceObject)
#004 {
#005     //确保当前函数运行在分页内存中
#006     PAGED_CODE();
#007     KdPrint(("Enter HelloWDMAddDevice\n"));
#008
#009     NTSTATUS status;
#010     PDEVICE_OBJECT fdo;
#011     //创建设备对象行
#012     status = IoCreateDevice(
#013         DriverObject,
#014         sizeof(DEVICE_EXTENSION),
#015         NULL, //没有指定设备名
#016         FILE_DEVICE_UNKNOWN,
#017         0,
#018         FALSE,
#019         &fdo);
#020     //判断操作是否公共
```



```
#021     if( !NT_SUCCESS(status))
#022         return status;
#023     //获得设备扩展
#024     PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION)fdo->DeviceExtension;
#025     //记录 FDO
#026     pdx->fdo = fdo;
#027     //将 FDO 挂载在设备堆栈上
#028     pdx->NextStackDevice = IoAttachDeviceToDeviceStack(fdo, PhysicalDevice
Object);
#029
#030     //创建设备接口
#031     status = IoRegisterDeviceInterface(PhysicalDeviceObject, &MY_WDM_DEVICE,
NULL, &pdx->interfaceName);
#032     //判断操作是否成功
#033     if( !NT_SUCCESS(status))
#034     {
#035         //删除设备
#036         IoDeleteDevice(fdo);
#037         return status;
#038     }
#039     KdPrint(("wZ\n", &pdx->interfaceName));
#040     //设置接口
#041     IoSetDeviceInterfaceState(&pdx->interfaceName, TRUE);
#042     ..判断操作是否成功
#043     if( !NT_SUCCESS(status))
#044     {
#045         if( !NT_SUCCESS(status))
#046         {
#047             return status;
#048         }
#049     }
#050     //设置操作为缓冲区模式
#051     fdo->Flags |= DO_BUFFERED_IO | DO_POWER_PAGABLE;
#052     fdo->Flags &= ~DO_DEVICE_INITIALIZING;
#053     KdPrint(("Leave HellowDMAddDevice\n"));
#054     return STATUS_SUCCESS;
#055 }
```

此段代码可以在配套光盘中本章的 Test1 目录下找到。

在上述代码中，读者可以将 `pdx->interfaceName` 通过 log 信息打印出来查看，其内容如图 13-3 所示。其实 `pdx->interfaceName` 就是暴露给应用程序的符号链接，也就是符号链接。而这种新符号链接由以下四部分组成：

- (1) 第一部分是何种总线的设备，如图 13-3 中的设备就是根总线上的设备，即 ROOT。
- (2) 第二部分是类设备的名称，如图 13-3 中的设备就是 ZHANGFANDEVICE。
- (3) 第三部分是这种设备的第几个设备，通过图 13-3 中可以看出这是同种设备的第 0 个设备。
- (4) 第四部分是指定的设备接口的 GUID。

在后面的讲述中可以看出，应用程序就是通过这组符号链接找到设备的。

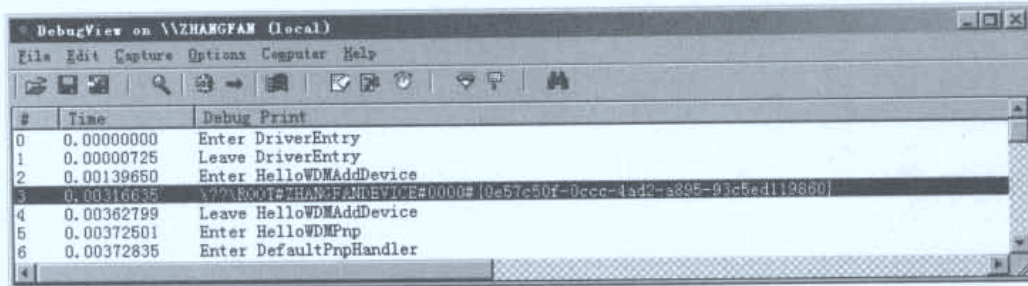


图 13-3 查看符号链接

另外，在卸载驱动的时候，需要将设备接口删除，如下面代码：

```
IoSetDeviceInterfaceState(&px->interfaceName, FALSE);
```

13.3.3 应用程序寻找接口

在应用程序中寻找设备，是通过设备接口和设备号决定的。这里的设备号是指如果 PC 中安装了两个相同驱动程序的网卡，第一个网卡就是第 0 号网卡，而第二个网卡就是第 1 号网卡。

在应用程序中主要是通过 SetupDiXX 系列函数得到设备接口，SetupDiXX 用法十分灵活，具体内容读者可以查看 MSDN 的相关资料，下面列出了寻找接口的程序。

```
#001 HANDLE GetDeviceViaInterface( GUID* pGuid, DWORD instance)
#002 {
#003     // 获取类信息
#004     HDEVINFO info = SetupDiGetClassDevs(pGuid, NULL, NULL, DIGCF_PRESENT |
DIGCF_INTERFACEDEVICE);
#005     //判断是否成功获得类信息
#006     if(info==INVALID_HANDLE_VALUE)
#007     {
#008         printf("No HDEVINFO available for this GUID\n");
#009         return NULL;
#010     }
#011
#012     // 获得设备接口
#013     SP_INTERFACE_DEVICE_DATA ifdata;
#014     ifdata.cbSize = sizeof(ifdata);
#015     if(!SetupDiEnumDeviceInterfaces(info, NULL, pGuid, instance, &ifdata))
#016     {
#017         printf("No SP_INTERFACE_DEVICE_DATA available for this GUID instance\n");
#018         SetupDiDestroyDeviceInfoList(info);
#019         return NULL;
#020     }
#021
#022     // 得到符号链接名
#023     DWORD ReqLen;
#024     SetupDiGetDeviceInterfaceDetail(info, &ifdata, NULL, 0, &ReqLen, NULL);
#025     PSP_INTERFACE_DEVICE_DETAIL_DATA ifDetail = (PSP_INTERFACE_DEVICE_DETAIL_
DATA)(new char[ReqLen]);
#026     if( ifDetail==NULL)
#027     {
```



```
#028         SetupDiDestroyDeviceInfoList(info);
#029         return NULL;
#030     }
#031
#032     //得到符号链接名
#033     ifDetail->cbSize = sizeof(SP_INTERFACE_DEVICE_DETAIL_DATA);
#034     if( !SetupDiGetDeviceInterfaceDetail(info, &ifdata, ifDetail, ReqLen, NULL,
NULL))
#035     {
#036         SetupDiDestroyDeviceInfoList(info);
#037         delete ifDetail;
#038         return NULL;
#039     }
#040
#041     printf("Symbolic link is %s\n",ifDetail->DevicePath);
#042     //通过符号链接名打开设备
#043     HANDLE rv = CreateFile( ifDetail->DevicePath,
#044         GENERIC_READ | GENERIC_WRITE,
#045         FILE_SHARE_READ | FILE_SHARE_WRITE,
#046         NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
#047     if( rv==INVALID_HANDLE_VALUE) rv = NULL;
#048
#049     delete ifDetail;
#050     //删除设备信息列表
#051     SetupDiDestroyDeviceInfoList(info);
#052     return rv;
#053 }
```

此段代码可以在配套光盘中本章的 Test1 目录下找到。

13.3.4 查看接口设备

因为创建设备时，没有指定具体的设备名称，而是系统自动分配给设备一个设备名。因此，在设备管理器中可以加载多个相同的 WDM 驱动，而每个驱动设备对应不同的设备号。

加载 WDM 驱动，可以通过工具软件 EzDriverInstaller 帮助加载，如图 13-4 所示。通过 EzDriverInstaller 程序，可以方便地帮助程序员加载、卸载、升级 WDM 驱动。



图 13-4 用 EzDriverInstaller 加载 WDM 驱动

由于没有设备名冲突的限制,因此可以加载任意多个,如图 13-5 演示了笔者在计算机上加载了三个 HelloWDM 驱动。

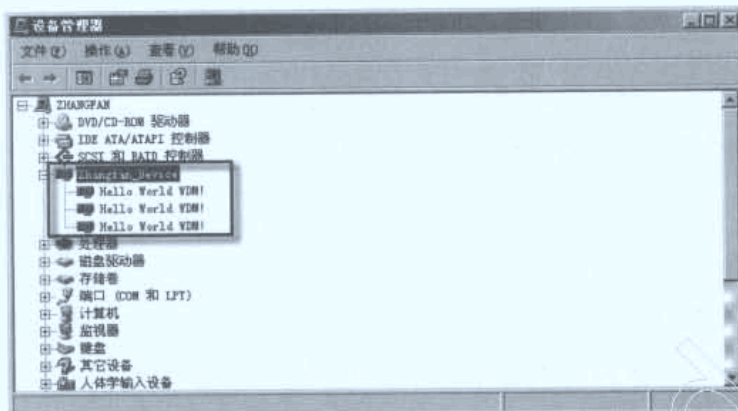


图 13-5 用设备管理器查看加载的 WDM 驱动

另外,通过工具软件 WinObj 可以观察设备接口的创建情况。如图 13-6 所示,通过设备接口其实间接地创建了设备链接。在通过 WinObj 查看设备链接时,可以发现三个相同驱动所创建的符号链接,它们的区别只是设备号不同而已。

另外虽然设备在创建时没有指定设备名,但系统会为设备默认指定一个设备名,即一组数字,如 \Device\0000000b、\Device\0000000b、\Device\0000000e。

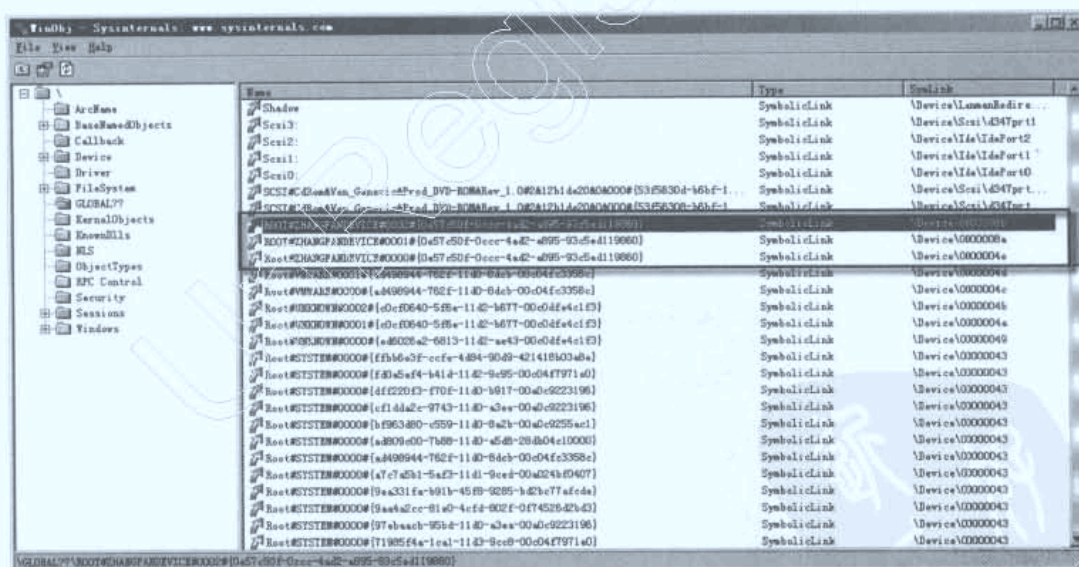


图 13-6 用 WinObj 查看设备

13.4 启动和停止设备

设备在启动和停止前,都会由即插即用管理器向 WDM 驱动发送相关的即插即用 IRP。

WDM 驱动通过这些即插即用 IRP 可以知道以下的两个重要信息:

- (1) 该设备目前处于何种状态, 即开启还是停止状态。
- (2) 得到相应的资源信息。

本节将重点讲述启动和停止相关的即插即用 IRP。

13.4.1 为一个实际硬件安装 HelloWDM

前面安装的 HelloWDM 都是为一个虚拟设备做驱动程序, 现在笔者带领读者略加修改就可以让一个真正的物理设备使用 HelloWDM。

在安装驱动的时候, 系统会搜索相关的 INF 文件, 如果 INF 文件中指定的 VendorID 和 ProduceID 信息与设备匹配上, 就会安装 INF 指定的驱动程序。例如, 以下是 HelloWDM.inf 其中的一段:

```
#001 [Mfg0]
#002
#003 ; PCI hardware Ids use the form
#004 ; PCI\VEN_aaaa&DEV_bbbb&SUBSYS_cccccc&REV_dd
#005 ;改成你自己的 ID
#006 %DeviceDesc%=YouMark_DDI, PCI\VEN_9999&DEV_9999
```

其中, PCI\VEN_9999&DEV_9999 就是 VendorID 和 ProductID 的组合, 这里指的就是一个虚拟设备的 VendorID 和 ProductID 的组合。如果这里指定的是一个真实物理设备的 VendorID 和 ProductID 的组合, 就会为这个设备加载 HelloWDM 驱动。

下面就告诉读者如何得知 VendorID 和 ProductID。以网卡设备为例, 通过设备管理器选择相应的物理设备, 在选择“详细信息”一栏, 如图 13-7 所示, 列出的就是 VendorID 和 ProductID 的组合, 通过“Ctrl+C”就可将其复制下来。这里的图 13-7 是已经将驱动安装完毕的效果, 对于新插入的未知设备同样可以用这种办法知道其 VendorID 和 ProductID 的组合。



图 13-7 查看 VendorID 和 ProductID

由上图可以看出, 该字符串是 PCI\VEN_10EC&DEV_8139&SUBSYS_813910EC&REV_10\4&1AF1648C&0&08F0, 但我们只需要前面的字符串 PCI\VEN_10EC&DEV_8139 即可。

修改 INF 文件, 以下是修改的相关片段。

```
#001 [Mfg0]
#002
#003 ; PCI hardware Ids use the form
#004 ; PCI\VEN_aaaa&DEV_bbbb&SUBSYS_cccccc&REV_dd
#005 ;改成你自己的 ID
#006 %DeviceDesc%=YouMark_DDIPCI\VEN_10EC&DEV_8139
```

此段代码可以在配套光盘中本章的 Test2 目录下找到。由于该目录下的 HelloWDM.inf 指定的设备不一定是读者 PC 中的设备, 请读者根据上述步骤修改。

下面就可以更新该网卡驱动了, 但需要注意的是, 更新网卡驱动后, HelloWDM 驱动并没有实现网卡驱动, 只是为了演示即插即用 IRP。有关网络的所有设置都会被清除, 所读读者更新的时候要慎重。显然, 用 HelloWDM 更新显卡驱动并不是一个好想法, 因为这样会破坏 Windows 的显示。笔者 PC 中有两块网卡, 一块是主板带的, 另一块是一块标砖的百兆网卡。笔者选择其中一个加载 HelloWDM 驱动。

读者也可以试着更新其他驱动程序, 例如, 网卡、声卡、显卡等, 最好选择 PCI 设备, 在后面的学习中读者会慢慢体会为什么这么做的原因。在设备管理器中选择原先设备, 并单击“更新驱动程序...”, 按照提示选择新的 INF 文件, 如图 13-8 所示。

注意, 安装的时候系统会提示不兼容, 或者没有数字验证等信息, 读者可以忽略这些错误。如图 13-9 所示, 网卡设备用 HelloWDM 驱动成功加载。但此时的网络是不能用的, 因为这里仅仅是一个演示普通设备的安装。

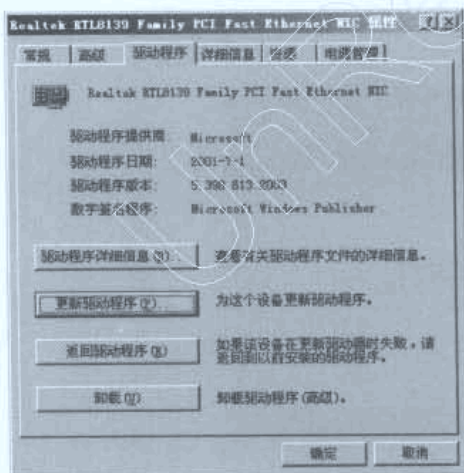


图 13-8 更新驱动



图 13-9 设备资源

当然, 可以随时将现在驱动切换到原有驱动, 这样网络就可以正常使用了, 同样是更新设备, 但不用寻找 INF, 操作系统会为用户记录下这一步, 如图 13-10 所示。

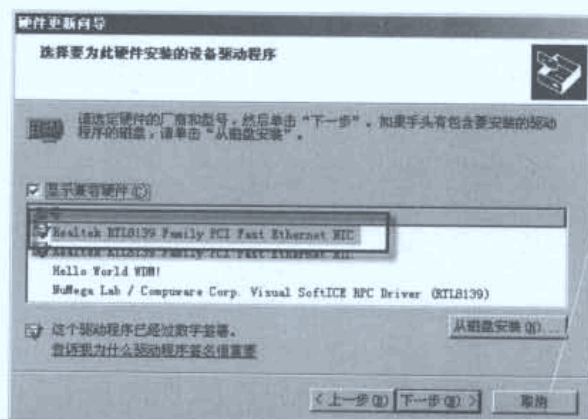


图 13-10 恢复以前的驱动

从图 13-9 中可以看出，HelloWDM 并没有申请任何资源，而关于网卡上的所有资源，都已经被枚举出来了，这就是即插即用的功劳了，也就是由启动相关的即插即用 IRP 和底层的物理设备对象（PDO）联合完成的。

13.4.2 启动设备

在设备启动的时候，设备管理器会向 WDM 驱动发送一个 IRP_MN_START_DEVICE 的子功能代码的即插即用 IRP。

这个 IRP 一般不需要 WDM 驱动来完成，而是将其转发到底层 PDO 来完成。IRP_MN_START_DEVICE 被传递到 PDO 后，PDO 会根据设备的类型枚举该设备的所有资源。如图 13-11 所示，该设备的所有资源都是通过 PDO 进行枚举的，而在传统的 NT 驱动中都是由驱动自己来枚举的。

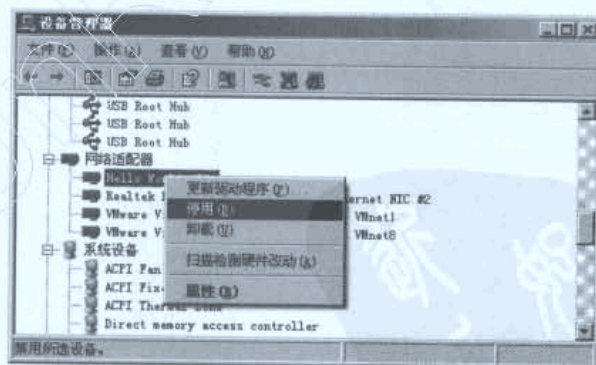


图 13-11 启动停止设备

因此通过这一点可以看出，WDM 简化了驱动的开发，很多通用并且烦琐的事情都交由操作系统来完成。其实，PDO 可以认为是一个 NT 式驱动。

观察 IRP 最好的工具莫过于 IRPTrace，让 IRPTrace 记录 HelloWDM 相关的 IRP。为

了观察 IRP_MN_START_DEVICE 的 IRP，此时驱动已经加载，必须重新开启该设备，如图 13-11 所示，先将设备停用，再开启设备。

用 IRPTrace 主要是为了查看 WDM 驱动将 IRP 传递给底层的 PDO 后，PDO 是如何处理 IRP_MN_START_DEVICE 的这个 IRP。如图 13-12 所示，PDO 完成此 IRP 之前，会得到所有的设备资源信息，如中断号、IO 地址、物理设备内存等。

在图 13-12 中，可以看到有两类资源，一类是分配资源（Allocated Resources），而另一类是翻译资源（Translated Resources）。简单地讲分配资源是 PC 得到的原始资源，而翻译资源是经过翻译后的资源，在驱动中应该使用翻译资源。

其实分配资源和翻译资源是基本相同的，只是中断号有所不同，在设备管理器中看到是分配资源中的中断号，而 WDM 驱动中使用的是翻译资源里的中断号。

除此之外，PDO 在完成该 IRP 的同时，还能知道此种设备属于何种总线上的设备，如图 13-12 中，就是 PCI 设备。

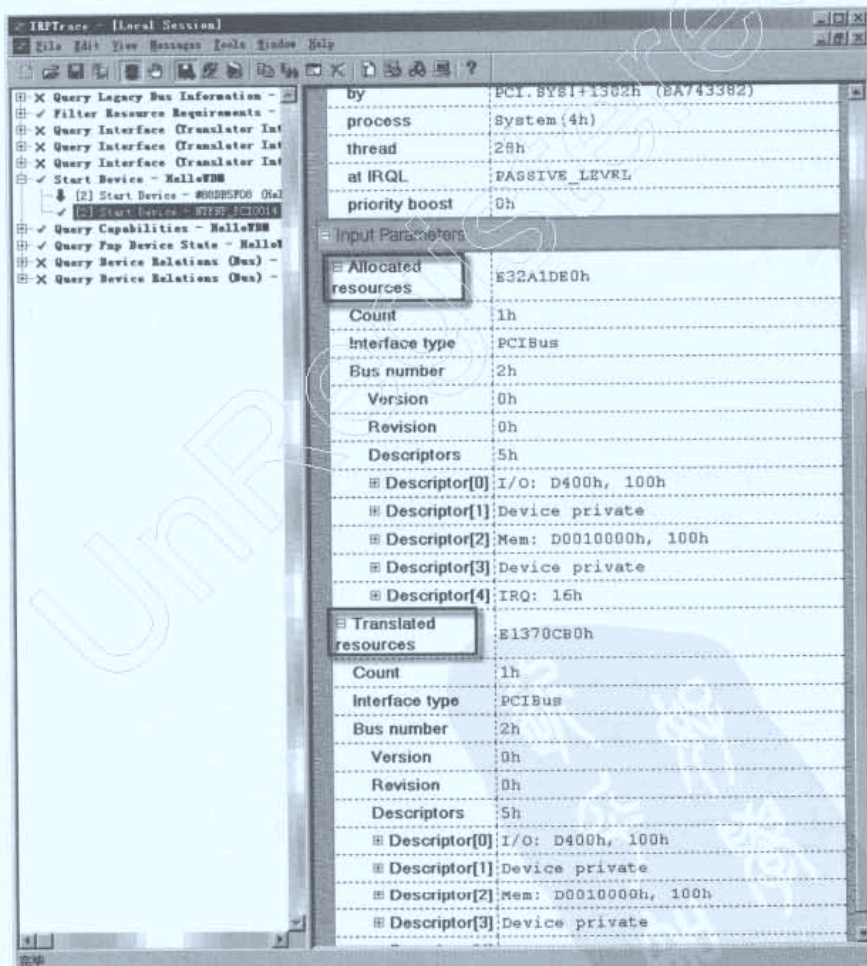


图 13-12 用 IRPTrace 查看 IRP_MN_START_DEVICE

13.4.3 转发并等待

为了能得到底层 PDO 处理 IRP 的结果，需要调用 PDO 后，能查询 IRP 的结果。但是，这里面临两个问题。

(1) 不知道 PDO 是基于同步完成还是异步完成。如果是同步完成，即 IoCallDriver (pdx->NextStackDevice, Irp) 的返回就标志着 PDO 处理 IRP 的完成。如果是异步完成，IoCallDriver (pdx->NextStackDevice, Irp) 的返回并不能代表 PDO 处理 IRP 完成。

(2) IRP 一旦被完成 (即 IoCompleteRequest (Irp, IO_NO_INCREMENT))，就不能再对 IRP 进行操作。但我们需要获得底层设备 PDO 对 IRP 的设置情况。

解决上述两个问题，需要采用完成例程，并归结为以下几个步骤：

- ① 即插即用 IRP 进入 WDM 的派遣函数。
- ② 派遣函数初始化一个事件，这个事件作为与 PDO 的同步之用。
- ③ 设置完成例程，并将事件作为参数传递给完成例程。
- ④ 调用底层驱动，即 PDO，并紧接着等待这个同步事件。
- ⑤ 底层驱动完成 IRP 时，会触发完成例程。
- ⑥ 在完成例程中，将事件设为有效。
- ⑦ 事件有效后，等待停止，进而转而下面的操作。

其中设置完成例程不能像以前的例子那样，掠过当前堆栈，如：

```
#001 IoSkipCurrentIrpStackLocation(Irp);
#002 return IoCallDriver(pdx->NextStackDevice, Irp)
```

而需要先复制当前堆栈，并设置完成例程，如：

```
#001 IoCopyCurrentIrpStackLocationToNext(Irp);
#002 //设置完成例程
#003 IoSetCompletionRoutine(Irp, (PIO_COMPLETION_ROUTINE) OnRequestComplete,
#004 (PVOID) &event, TRUE, TRUE, TRUE);
```

笔者将以上的诸多步骤，封装成一个函数 ForwardAndWait，在处理即插即用的派遣函数中调用这个函数即可。

```
#001 #pragma PAGEDCODE
#002 NTSTATUS ForwardAndWait(PDEVICE_EXTENSION pdx, PIRP Irp)
#003 { // ForwardAndWait
#004     PAGED_CODE();
#005
#006     KEVENT event;
#007     //初始化事件
#008     KeInitializeEvent(&event, NotificationEvent, FALSE);
#009
#010     //将本层堆栈复制到下一层堆栈
#011     IoCopyCurrentIrpStackLocationToNext(Irp);
#012     //设置完成例程
#013     IoSetCompletionRoutine(Irp, (PIO_COMPLETION_ROUTINE) OnRequestComplete,
#014 (PVOID) &event, TRUE, TRUE, TRUE);
```

```

#015
#016 //调用底层驱动, 即 PDO
#017 IoCallDriver(pdx->NextStackDevice, Irp);
#018 //等待 PDO 完成
#019 KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);
#020 return Irp->IoStatus.Status;
#021 } // ForwardAndWait

```

以下是 ForwardAndWait 中设置的完成例程。

```

#001 #pragma LOCKEDCODE
#002 NTSTATUS OnRequestComplete(PDEVICE_OBJECT junk, PIRP Irp, PKEVENT pev)
#003 { // OnRequestComplete
#004 //在完成例程中设置等待事件
#005 KeSetEvent(pev, 0, FALSE);
#006 //标志本 IRP 还需要再次被完成
#007 return STATUS_MORE_PROCESSING_REQUIRED;
#008 }

```

此段代码可以在配套光盘中本章的 Test3 目录下找到。

13.4.4 获得设备相关资源

有了上一节的方法, 就可以方便地知道 IRP_MN_START_DEVICE 被传递给 PDO 处理后的结果。在即插即用的派遣函数中, 会分析当前即插即用 IRP 子功能代码, 如果是 IRP_MN_START_DEVICE, 则会调用 HandleStartDevice。

在 HandleStartDevice 函数中, 先将该 IRP 通过 ForwardAndWait 转发到底层 PDO。其中 ForwardAndWait 的原理在上一节已经介绍过了, 主要是在完成例程中设置同步事件, 通过同步事件可以知道 PDO 处理完该 IRP。

由于在派遣函数中返回的是 STATUS_MORE_PROCESSING_REQUIRED, 所以该 IRP 可以继续被处理。于是在 HandleStartDevice 的后面, 需要再一次完成该 IRP, 即调用 IoCompleteRequest。

IRP_MN_START_DEVICE 这个 IRP 被 PDO 处理时, 会得到当前设备的很多资源信息, 并储存在 IRP 的堆栈里。以下是 HandleStartDevice 的代码:

```

#001 #pragma PAGEDCODE
#002 NTSTATUS HandleStartDevice(PDEVICE_EXTENSION pdx, PIRP Irp)
#003 {
#004 //确保当前函数运行在分页内存
#005 PAGED_CODE();
#006 KdPrint(("Enter HandleStartDevice\n"));
#007
#008 //转发 IRP 并等待返回
#009 NTSTATUS status = ForwardAndWait(pdx, Irp);
#010 //判断操作是否成功
#011 if (!NT_SUCCESS(status))
#012 {
#013 //设置 IRP 完成状态
#014 Irp->IoStatus.Status = status;
#015 //结束 IRP 请求

```



```

#016         IoCompleteRequest(Irp, IO_NO_INCREMENT);
#017         return status;
#018     }
#019     //得到当前堆栈
#020     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
#021     //从当前堆栈得到源信息
#022     PCM_PARTIAL_RESOURCE_LIST raw;
#023     if (stack->Parameters.StartDevice.AllocatedResources)
#024         raw = &stack->Parameters.StartDevice.AllocatedResources->List[0].
PartialResourceList;
#025     else
#026         raw = NULL;
#027
#028     KdPrint(("Show raw resources\n"));
#029     //显示 PCI 资源
#030     ShowResources(raw);
#031
#032     //从当前堆栈得到翻译信息
#033     PCM_PARTIAL_RESOURCE_LIST translated;
#034     if (stack->Parameters.StartDevice.AllocatedResourcesTranslated)
#035         translated = &stack->Parameters.StartDevice.AllocatedResources
Translated->List[0].PartialResourceList;
#036     else
#037         translated = NULL;
#038
#039     KdPrint(("Show translated resources\n"));
#040     //显示 PCI 资源
#041     ShowResources(translated);
#042
#043     //设置 IRP 完成状态
#044     Irp->IoStatus.Status = STATUS_SUCCESS;
#045     //结束 IRP 请求
#046     IoCompleteRequest(Irp, IO_NO_INCREMENT);
#047
#048     KdPrint(("Leave HandleStartDevice\n"));
#049     return status;
#050 }

```

此段代码可以在配套光盘中本章的 Test3 目录下找到。

13.4.5 枚举设备资源

PDO 完成 IRP_MN_START_DEVICE 后,会将获取到的设备资源存储在 IRP 的设备堆栈中,并且存储在 `stack->Parameters.StartDevice.AllocatedResourcesTranslated` 中。这是一个 `CM_PARTIAL_RESOURCE_LIST` 的结构指针。

```

typedef struct _CM_PARTIAL_RESOURCE_LIST {
    USHORT Version;
    USHORT Revision;
    ULONG Count;
    CM_PARTIAL_RESOURCE_DESCRIPTOR PartialDescriptors[1];
} CM_PARTIAL_RESOURCE_LIST, *PCM_PARTIAL_RESOURCE_LIST;

```

该结构如图 13-13 所示,会包含很多 `CM_PARTIAL_RESOURCE_DESCRIPTOR` 结构体。

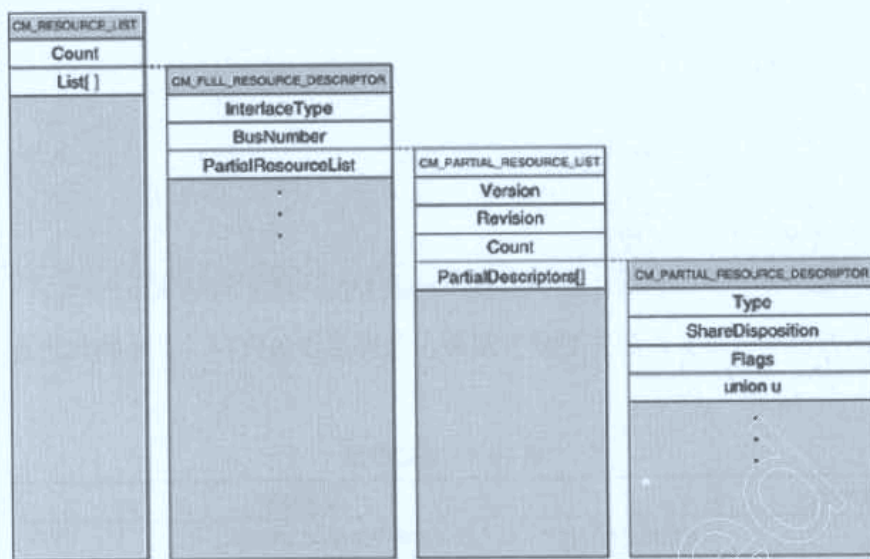


图 13-13 CM_PARTIAL_RESOURCE_DESCRIPTOR 结构

在 CM_PARTIAL_RESOURCE_DESCRIPTOR 中描述了设备中所需要的资源信息。

```
typedef struct _CM_PARTIAL_RESOURCE_DESCRIPTOR {
    UCHAR Type; //类别
    UCHAR ShareDisposition;
    USHORT Flags; // CM_PARTIAL_RESOURCE_DESCRIPTOR 参数
    union {
        struct {
            PHYSICAL_ADDRESS Start; //设备物理地址
            ULONG Length; //物理地址长度
        } Generic;
        struct {
            PHYSICAL_ADDRESS Start; //设备物理地址
            ULONG Length; //物理地址长度
        } Port;
        struct {
            ULONG Level; //中断级别
            ULONG Vector; //中断向量
            ULONG Affinity; //中断 CPU 亲缘关系
        } Interrupt;
        struct {
            PHYSICAL_ADDRESS Start; //物理地址
            ULONG Length; //物理地址长度
        } Memory;
        struct {
            ULONG Channel;
            ULONG Port; //设备物理端口
            ULONG Reserved1;
        } Dma;
        struct {
            ULONG Data[3];
        } DevicePrivate;
    }
};
```



```
        ULONG Start;        //总线号
        ULONG Length;       //长度
        ULONG Reserved;
    } BusNumber;
    struct {
        ULONG DataSize;
        ULONG Reserved1;
        ULONG Reserved2;
    } DeviceSpecificData;
    } u;
} CM_PARTIAL_RESOURCE_DESCRIPTOR, *PCM_PARTIAL_RESOURCE_DESCRIPTOR;
```

如表 13-2 所示，设备资源主要分为四类，分别是设备端口、设备物理内存、DMA、中断等。

表 13-2 设备资源

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）

CM_PARTIAL_RESOURCE_LIST 结构包含一个计数器和一个 CM_PARTIAL_RESOURCE_DESCRIPTOR 结构的数组，数组元素的描述见图 13-13。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。如果设备使用一个 IRQ 和一组 I/O 端口，那么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，不能提前知道这些描述符在数组中出现的位置。因此，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。

以下是枚举设备资源的代码，读者可以根据具体的设备将设备资源记录下来。

```
#001 #pragma PAGEDCODE
#002 VOID ShowResources(IN PCM_PARTIAL_RESOURCE_LIST list)
#003 {
#004     //枚举资源
#005     PCM_PARTIAL_RESOURCE_DESCRIPTOR resource = list->PartialDescriptors;
#006     ULONG nres = list->Count;
#007     ULONG i;
#008
#009     for (i = 0; i < nres; ++i, ++resource)
#010     {
#011         //对每种资源进行枚举
#012         ULONG type = resource->Type;
#013
#014         static char* name[] = {
#015             "CmResourceTypeNull",
#016             "CmResourceTypePort",
#017             "CmResourceTypeInterrupt",
#018             "CmResourceTypeMemory",
#019             "CmResourceTypeDma",
```

```

#020         "CmResourceTypeDeviceSpecific",
#021         "CmResourceTypeBusNumber",
#022         "CmResourceTypeDevicePrivate",
#023         "CmResourceTypeAssignedResource",
#024         "CmResourceTypeSubAllocateFrom",
#025     };
#026
#027     KdPrint((" type %s", type < arraysize(name) ? name[type] : "unknown"));
#028     //枚举 PCI 资源
#029     switch (type)
#030     {
#031     //物理内存资源
#032     case CmResourceTypePort:
#033     case CmResourceTypeMemory:
#034         KdPrint((" start %8X%8.8lX length %X\n",
#035             resource->u.Port.Start.HighPart, resource->u.Port.Start.
LowPart,
#036             resource->u.Port.Length));
#037         break;
#038     //中断资源
#039     case CmResourceTypeInterrupt:
#040         KdPrint((" level %X, vector %X, affinity %X\n",
#041             resource->u.Interrupt.Level, resource->u.Interrupt.Vector,
#042             resource->u.Interrupt.Affinity));
#043         break;
#044     //DMA 资源
#045     case CmResourceTypeDma:
#046         KdPrint((" channel %d, port %X\n",
#047             resource->u.Dma.Channel, resource->u.Dma.Port));
#048     }
#049     }
#050 }

```

此段代码可以在配套光盘中本章的 Test3 目录下找到。

如图 13-14 所示,就是用改造后的 HelloWDM 驱动为 RTL8139 网卡设备加载后枚举出来的设备资源,读者可以对照着这些信息和 IRPTrace 工具中的输出的信息进行比较。

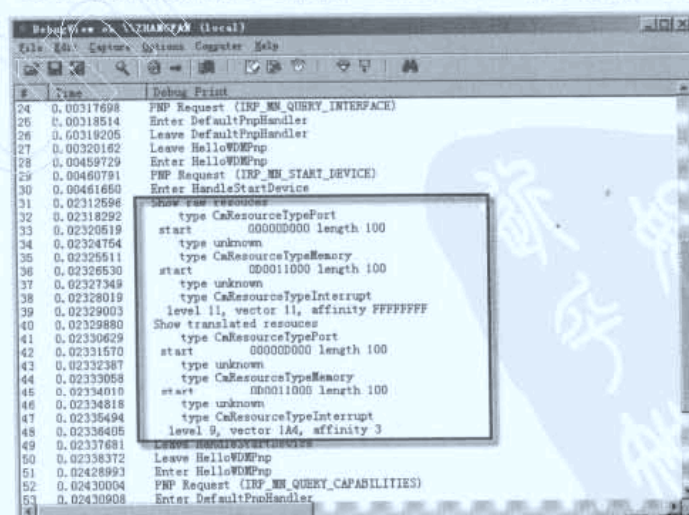


图 13-14 输出资源枚举信息

13.4.6 停止设备

设备在启动时，即插即用管理器会发送 IRP_MN_START_DEVICE 给 WDM 驱动。同样，在设备即将停止的时候，即插即用管理器也会发送一个叫做 IRP_MN_REMOVE_DEVICE。这个 IRP 标志着设备即将关闭，WDM 驱动在此时应该做一些资源回收工作。例如，删掉设备接口、从设备栈中弹出、删除 FDO 设备等。

以下是关于处理 IRP_MN_REMOVE_DEVICE 的代码示例：

```
#001 #pragma PAGEDCODE
#002 NTSTATUS HandleRemoveDevice(PDEVICE_EXTENSION pdx, PIRP Irp)
#003 {
#004     //保证此函数运行于分页内存
#005     PAGED_CODE();
#006     KdPrint(("Enter HandleRemoveDevice\n"));
#007
#008     Irp->IoStatus.Status = STATUS_SUCCESS;
#009     //将 IRP 转发到底层设备驱动处理
#010     NTSTATUS status = DefaultPnpHandler(pdx, Irp);
#011     //设置设备接口
#012     IoSetDeviceInterfaceState(&pdx->interfaceName, FALSE);
#013     //释放 UNICODE 字符串
#014     RtlFreeUnicodeString(&pdx->interfaceName);
#015
#016     //调用 IoDetachDevice() 把 fdo 从设备栈中脱开:
#017     if (pdx->NextStackDevice)
#018         //删除设备
#019         IoDetachDevice(pdx->NextStackDevice);
#020
#021     //删除 fdo
#022     IoDeleteDevice(pdx->fdo);
#023     KdPrint(("Leave HandleRemoveDevice\n"));
#024     return status;
#025 }
```

此段代码可以在配套光盘中本章的 Test3 目录下找到。

另外，WDM 驱动可以将 IRP_MN_REMOVE_DEVICE 设置为完成失败，即 IoCompleteRequest 时设置 IRP 参数为失败。这样会通知即插即用管理器 WDM 驱动不允许设备停止。同理，如果 IRP_MN_START_DEVICE 完成不成功，就是告诉即插即用管理器此设备不能正常启动。这时，在设备管理器中会看到此设备有一个惊叹号的图标，如图 13-15 所示。

除此之外，有些设备（如 USB 设备）支持热插拔，在拔出的瞬间即插即用管理器会向 WDM 驱动发送 IRP_MN_SURPRISE_REMOVAL 消息。此消息意味着即插即用管理器通知 WDM 驱动，此设备被突然拔出，WDM 做相应处理。

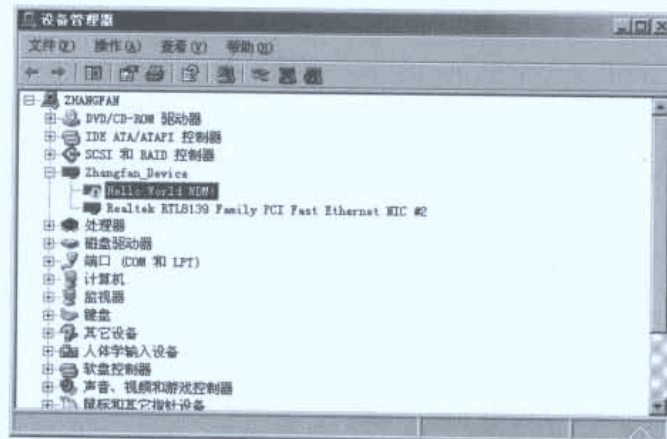


图 13-15 启动失败

13.5 即插即用的状态转换

为了支持即插即用功能，WDM 驱动程序内部必须实现类似状态机的机制。WDM 驱动程序根据即插即用管理器发送来的即插即用 IRP，转换内部状态机的状态。并根据当前状态，对即插即用 IRP 进行处理。

13.5.1 状态转换图

WDM 驱动加载和卸载时，即插即用管理器会向设备发送各类即插即用 IRP，如前面介绍的 IRP_MN_START_DEVICE 和 IRP_MN_REMOVE_DEVICE 等。

如图 13-16 所示，描述了一个设备的即插即用状态，这里很自然地联想到有限状态机。当一个设备被添加到系统时，Windows 查找正确的驱动程序，并调用它的 DriverEntry 例程。

即插即用管理器然后调用驱动程序的 AddDevice 例程，告诉它添加了一个新设备。此时，驱动程序创建自己的设备对象，即 FDO。

在处理的过程中，驱动程序会收到 IRP_MN_START_DEVICE 的 IRP，这里包括设备被分配的资源信息，然后它开始与设备硬件进行合适的对话。

如果一个设备要被拔出，Windows 使用 IRP_MN_REMOVE 的 IRP 查询驱动程序设备。如果驱动程序不希望设备被删除，它将拒绝删除请求，然后发送 IRP_MN_CANCEL_REMOVE 的 IRP，使它回到开始的状态。如果用户突然拔掉一个设备，在 Windows 中会发送 IRP_MN_SUPPRISE_REMOVE 的 IRP，驱动必须处理好中断了的传输。

当即插即用管理器希望重新分配驱动程序的一些资源时，会发生另一个主要的状态变化。如果某种新的设备插入系统，就会发生这种情况，这意味着资源分配需要调整。即插

即用管理器要求在资源重新分配时停止该驱动程序。类似的，在对删除请求的响应中，IRP_MN_QUERY_STOP_DEVICE 的 IRP 询问是否可以停止设备。如果可以停止设备，则发出一个 IRP_MN_STOP_DEVICE 的 IRP，使设备回到启动状态。当设备停止时，驱动程序不能访问它的设备，一个 IRP_MN_START_DEVICE 的 IRP 通知驱动程序设备的新资源并再次开启设备。

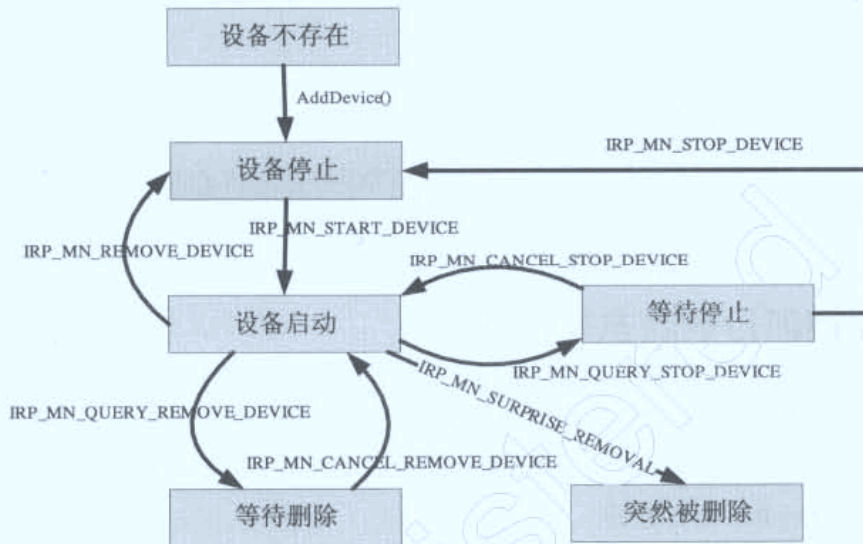


图 13-16 状态转换图

13.5.2 IRP_MN_QUERY_STOP_DEVICE

即插即用管理器在停止设备前总是先询问，得到允许后才向 WDM 驱动发送 IRP_MN_STOP_DEVICE 请求。询问以 IRP_MN_QUERY_STOP_DEVICE 请求的形式出现，WDM 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 WDM 驱动发送 IRP_MN_STOP_DEVICE，WDM 是否能立即停止设备”。

如果返回失败，则即插即用管理器会发送一个 IRP_MN_CANCEL_STOP_DEVICE，表示该 WDM 驱动目前不能停止设备。

13.5.3 IRP_MN_QUERY_REMOVE_DEVICE

与即插即用管理器在停止设备前向 WDM 驱动询问一样，它在删除设备前也向 WDM 驱动询问，即 IRP_MN_QUERY_REMOVE_DEVICE 请求。WDM 可以回答成功或失败。与停止查询相似，如果即插即用管理器中途改变想法，它就发送 IRP_MN_CANCEL_REMOVE_DEVICE 请求。

如果返回失败，则即插即用管理器会发送一个 IRP_MN_CANCEL_REMOVE_DEVICE，

表示该 WDM 驱动目前不能删除这个设备。

13.6 其他即插即用 IRP

本节将会继续介绍两个即插即用消息。

13.6.1 IRP_MN_FILTER_RESOURCE_REQUIREMENTS

有时，即插即用管理器会错误地报告 WDM 驱动程序获取的资源。这可能由于硬件或固件上存在着错误，也可能因为某个遗留设备的 INF 文件中出现错误，或者是其他原因。系统以 IRP_MN_FILTER_RESOURCE_REQUIREMENTS 请求的形式提供了一个切入点，它在即插即用管理器将要分配资源前给程序员提供一个机会来检测或修改资源列表。

当 WDM 驱动收到一个 IRP_MN_FILTER_RESOURCE_REQUIREMENTS 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，如果上层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，应该使用这个数据结构。如果想向当前的资源列表中加入资源，应该在派遣函数中做。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。当最后获得控制时，就可以修改列表中的任何资源描述。

其中 IO_RESOURCE_REQUIREMENTS_LIST 的数据结构定义如下，由于字段都是自解释型的，通过字段的名称读者基本可以分析出其基本含义。

```
typedef struct _IO_RESOURCE_REQUIREMENTS_LIST {
    ULONG ListSize;
    INTERFACE_TYPE InterfaceType;    // WDM 驱动程序不使用这个域
    ULONG BusNumber;                 // WDM 驱动程序不使用这个域
    ULONG SlotNumber;
    ULONG Reserved[3];
    ULONG AlternativeLists;
    IO_RESOURCE_LIST List[1];
} IO_RESOURCE_REQUIREMENTS_LIST, *PIO_RESOURCE_REQUIREMENTS_LIST;
```

和处理 IRP_MN_START_DEVICE 一样，只要从 IRP 的设备堆栈中抽取出 IO_RESOURCE_REQUIREMENTS_LIST 的数据即可。因此，本节没有给出相关代码，只是用 IRPTrace 查看了一下 IRP_MN_FILTER_RESOURCE_REQUIREMENTS 的处理情况，如图 13-17 所示。

ListSize	108h
InterfaceType	PCIBus
Bus Number	2h
Slot Number	5h
Alternative Lists	1h
List[0]	Descriptors: 7h
Output Parameters	
Resource list	E31719D0h
ListSize	108h
InterfaceType	PCIBus
Bus Number	2h
Slot Number	5h
Alternative Lists	1h
List[0]	Descriptors: 7h
Version	1h
Revision	1h
Count	7h
Descriptor[0]	I/O: 100h from D400h to D4FFh
Option	IO_RESOURCE_PREFERRED
Type	CmResourceTypePort
ShareDisposition	CmResourceShareDeviceExclusive
Flags	CM_RESOURCE_PORT_IO CM_RESOURCE_PORT_16_BIT_DECODE CM_RESOURCE_PORT_POSITIVE_DECODE
Length	100h
Alignment	1h
MinimumAddress	D400h
MaximumAddress	D4FFh
Descriptor[1]	I/O: 100h from 0h to FFFFFFFFh
Descriptor[2]	Device private
Descriptor[3]	Mem: 100h from D0010000h to D00100FFh
Descriptor[4]	Mem: 100h from 0h to FFFFFFFFh
Descriptor[5]	Device private
Descriptor[6]	IRQ: from 0h to FFh

图 13-17 跟踪 IRP_MN_FILTER_RESOURCE_REQUIREMENTS

13.6.2 IRP_MN_QUERY_CAPABILITIES

即插即用管理器发送这个即插即用 IRP 得到设备的功能，例如，设备是否可以被锁定或者弹出，以及各种电源管理特性。如果功能驱动程序和过滤驱动程序更改总线驱动程序支持的功能，则它们可以处理这个请求。总线驱动程序必须为它们的设备处理这个请求。

该 IRP 发送两次，在功能驱动程序装入并启动之前和之后各发送一次。驱动程序可以沿设备栈向下发送该 IRP，看总线驱动程序的功能是什么。

程序员可以从 IRP_MN_QUERY_CAPABILITIES 的 IRP 堆栈中提取 DEVICE_CAPABILITIES，这个数据结构记录着很多有关电源相关的数据。

和处理 IRP_MN_START_DEVICE 一样，只要从 IRP 的设备堆栈中抽取 DEVICE_CAPABILITIES 的数据即可。因此，本节没有给出相关代码，只是用 IRPTrace 查看了一下 IRP_MN_QUERY_CAPABILITIES 的处理情况，如图 13-18 所示。

process	System (4h)
thread	28h
at IRQL	PASSIVE_LEVEL
priority boost	0h
Input Parameters	
Size	40h
Version	1h
DeviceD1	0h
DeviceD2	0h
LockSupported	0h
EjectSupported	0h
Removable	0h
DockDevice	0h
UniqueID	0h
SilentInstall	0h
RawDeviceOK	0h
SurpriseRemovalOK	0h
Address	FFFFFFFFh
UINumber	FFFFFFFFh
SystemWake	PowerSystemUnspecified (0h)
DeviceWake	PowerDeviceUnspecified (0h)
D1Latency	0h
D2Latency	0h
D3Latency	0h
Output Parameters	
Size	40h
Version	1h
DeviceD1	1h
DeviceD2	1h
LockSupported	0h
EjectSupported	0h
Removable	0h
DockDevice	0h

图 13-18 跟踪 IRP_MN_QUERY_CAPABILITIES

13.7 小结

本章主要介绍了 WDM 驱动程序对即插即用功能的支持。一个设备一般需要加载两个 WDM 驱动程序，一个是总线驱动程序（它提供 PDO），另一个是功能驱动程序（它提供 FDO）。PDO 和 FDO 这两个设备对象形成一个设备堆栈。非即插即用功能的 IRP 一般在 FDO 中处理，而即插即用功能相关的 IRP 会被转发到 PDO 中处理。

WDM 驱动程序模型大大简化了 Windows 驱动程序的开发。其来源于 NT 式驱动的分层驱动模型，可以说 WDM 驱动程序是 NT 式驱动程序的一个特例。

第 14 章 电源管理

对电源管理和即插即用功能的支持，是 Windows 2000 的重大特色，WDM 驱动程序也必须支持这两大特性。上一章中已经介绍了如何在 WDM 驱动程序中支持即插即用管理特性，本章将介绍 WDM 驱动程序如何支持电源管理特性。

14.1 WDM 电源管理模型

14.1.1 概述

在移动或便携环境中，显然需要对系统上安装的设备电源进行管理，因为其能源是有限的。然而，即使在桌面和服务端环境中，也需要减少不必要的能量消耗，降低热量产生。这样提高了组件可靠性，也减少了电量消耗和费用。

作为计算机全局电源策略，操作系统支持一些用户接口，用户可以通过这些接口控制最终的电源管理策略。这些用户接口包括控制面板、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 请求，内核的电源管理组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。

14.1.2 热插拔

Windows 2000 以后的版本支持 WDM 驱动程序模型，它结合了总线和设备硬件的复杂电源管理。因此，本节从总线方面和设备方面进行介绍。

(1) 总线方面

在设备能够降低功率之前，它所插的总线必须在电力上提供一种电力下降的状态。例如，许多总线驱动程序要求电源确保总线打开（on-bus）和总线关闭（off-bus）状态。没有电源供给总线驱动程序芯片，驱动程序及其对其余总线的影响是不确定的。显然，为了

支持健壮电源管理，硬件总线规范必须允许电源功率渐降的驱动程序。

在某些情况下，总线设计必须满足物理约束。尤其在移动系统中，经常进行设备插入和拔出。如果设备和拆除在一个“热”系统上进行，则必须谨慎地设计物理总线特征。除了明确要求每种设备在物理上分开外，还要关注那些不太引人注意的总线信号首先“发出”的问题。通常，这种总线的接地引脚实际上要比电源要长一些，确保在插入和拆除期间接地引脚最先插入和最后离开。

（2）设备方面

除了满足给定总线要求外，还需要允许单个设备非常灵活地进行电源管理。例如调制解调器卡允许在 UART 和总线电路断电的情况下，仍然为环路探测电路供电，以触发一个唤醒序列。

当考虑可能分离设备组成部分的电源情况下，显然设备可以有多种状态来描述它的电源状态。在“完全开通”和“完全关闭”这两个极端情况之间，设备可以根据递减能力实现功率递减的电源状态。从一个状态到另一个状态的转变可以由系统或用户请求指定，也可以仅仅在超出一定空闲时间后发生状态转移。

14.1.3 电源状态

在 Windows 中，系统有 6 种状态，分别是 S0 到 S5，如表 14-1 所示。系统的总体状态最大程度地限制了个体设备的最大电源状态。

表 14-1 系统电源状态

系统电源状态	含 义
S0(Working)	CPU 全功率运行，设备可以占据任何电源状态
S1(Sleeping1)	CPU 停止，RAM 被刷新
S2(Sleeping2)	CPU 不通电，RAM 被刷新
S3(Sleeping3)	CPU 不通电，RAM 处于低速刷新模式，电源输出功率降低
S4(Hibernate)	系统停止，RAM 被保存到磁盘
S5(Shutdown)	系统停止并关闭，需要完全引导以恢复操作

14.1.4 设备状态

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。如表 14-2 所示，描述了 4 种设备状态。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。

表 14-2 设备状态

设备电源状态	含 义
D0	设备全功率运行
D1	设备运行于低功率模式，设备环境可能被保留
D2	设备运行于低功率模式，设备环境可能无效
D3	设备没有电源，环境丢失

Microsoft 规定了不同类型设备的类专用的电源需求。该需求规范可以在 <http://www.microsoft.com/hwdev/specs/PMref/> 中找到。例如，该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。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。

14.1.5 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“待机”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，然后驱动程序将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此查询，电源管理器仍旧发出这个 IRP_MN_SET_POWER 请求，但它将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。

14.2 处理 IRP_MJ_POWER

关于电源管理的 IRP 主要是 IRP_MJ_POWER, 和 IRP_MJ_PNP 一样, IRP_MJ_POWER 也有若干个子功能代码, 例如, IRP_MN_POWER_SEQUENCE、IRP_MN_QUERY_POWER、IRP_MN_SET_POWER 及 IRP_MN_WAIT_WAKE。

当需要改变系统电源状态时, 用一个 IRP_MJ_POWER 请求通知所有电源策略所有者。策略所有者激发即插即用管理器的 PoRequestPowerIrp 调用。PoRequestPowerIrp 内核函数的声明如下:

```
NTSTATUS
PoRequestPowerIrp(
    IN PDEVICE_OBJECT DeviceObject,
    IN UCHAR MinorFunction,
    IN POWER_STATE PowerState,
    IN PREQUEST_POWER_COMPLETE CompletionFunction,
    IN PVOID Context,
    OUT PIRP *Irp OPTIONAL
);
```

- DeviceObject: 设备对象指针。
- MinorFunction: IRP_MJ_POWER 的子功能代码。
- PowerState: 描述电源状态, 用 POWER_STATE 数据结构描述。
- CompletionFunction: 完成例程。
- Context: 传递给完成例程的上下文。
- Irp: 传进去的 IRP 指针。

对于 IRP_MJ_POWER 类型的 IRP, 基本都是被传送到底层 PDO 进行处理的。被传送到底层 PDO 处理的过程不能使用 IoCallDriver 函数, 而必须使用 PoCallDriver 或者 PoRequestPowerIrp 函数。

14.3 处理 IRP_MN_QUERY_CAPABILITIES

IRP_MN_QUERY_CAPABILITIES 也是电源相关的 IRP, 本节将介绍此内容。

14.3.1 DEVICE_CAPABILITIES

与电源有关的还有一个 IRP 就是 IRP_MN_QUERY_CAPABILITIES, FDO 通过向底层 PDO 请求此 IRP, 会得到 DEVICE_CAPABILITIES 数据结构。

DEVICE_CAPABILITIES 数据结构会指明驱动程序支持的电源特性等, 其内容如下:

```
typedef struct _DEVICE_CAPABILITIES {
    USHORT Size;           // 数据结构大小
```



```

USHORT Version; //数据结构版本号
ULONG DeviceD1:1;
ULONG DeviceD2:1;
ULONG LockSupported:1; //是否支持锁定
ULONG EjectSupported:1; //是否支持拔出
ULONG Removable:1; //是否支持可移动
ULONG DockDevice:1;
ULONG UniqueID:1; //是否是唯一 ID
ULONG SilentInstall:1; //是否支持默认安装
ULONG RawDeviceOK:1; //源数据
ULONG SurpriseRemovalOK:1; //是否支持突然拔出
ULONG WakeFromD0:1; //是否支持从 D0 状态苏醒
ULONG WakeFromD1:1; //是否支持从 D1 状态苏醒
ULONG WakeFromD2:1; //是否支持从 D2 状态苏醒
ULONG WakeFromD3:1; //是否支持从 D3 状态苏醒
ULONG HardwareDisabled:1; //硬件屏蔽
ULONG NonDynamic:1;
ULONG WarmEjectSupported:1; // 是否支持热插拔
ULONG NoDisplayInUI:1;
ULONG Reserved:14; //保留
ULONG Address; //设备地址
ULONG UINumber;
DEVICE_POWER_STATE DeviceState[PowerSystemMaximum];
SYSTEM_POWER_STATE SystemWake; //苏醒状态
DEVICE_POWER_STATE DeviceWake;
ULONG D1Latency;
ULONG D2Latency;
ULONG D3Latency;
} DEVICE_CAPABILITIES, *PDEVICE_CAPABILITIES;

```

这里很多字段是有关电源特性的，例如，其中的 SurpriseRemovalOK 字段代表是否支持热插拔，Removable 指明是否可以动态删除驱动，EjectSupported 指明是否支持直接插拔。

14.3.2 一个试验

本节带领读者做一个很有意思的试验。很多即插即用设备插入 PC 时，任务栏右下角总会出现一个绿色的箭头。当双击此绿色的箭头时，会弹出一个如图 14-1 所示的对话框。

本节的例子是使 HelloWDM 支持此项功能，这就需要正确处理 DEVICE_CAPABILITIES 请求。首先在 IRP_MN_QUERY_CAPABILITIES 的处理函数中，先将此 IRP 传递给底层 PDO，然后将获得的 DEVICE_CAPABILITIES 数据结构进行修改。主要分为以下几个步骤：

- ① SurpriseRemovalOK=TRUE，让设备支持热插拔。
- ② Removable=TRUE，让设备支持热插拔。
- ③ EjectSupported = TRUE，让设备支持热插拔。
- ④ WarmEjectSupported，这个字段保留，现在无意义。

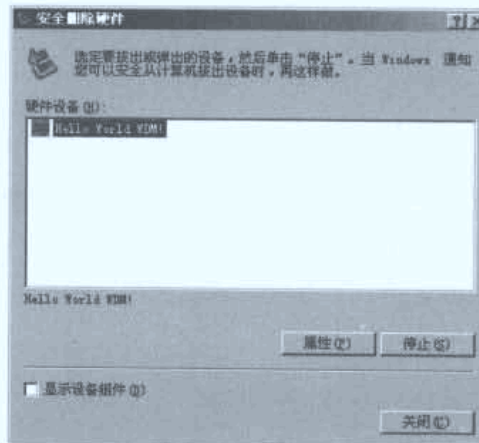


图 14-1 删除设备对话框

以下是示例代码：

```
#001 NTSTATUS PnpQueryCapabilitiesHandler( IN PDEVICE_EXTENSION pdx, IN PIRP Irp)
#002 {
#003     //将 IRP 传递到底层设备对象
#004     NTSTATUS status = ForwardAndWait( pdx, Irp);
#005     //判断是否操作成功
#006     if( NT_SUCCESS(status))
#007     {
#008         //得到当前 IO 堆栈
#009         PIO_STACK_LOCATION IrpStack = IoGetCurrentIrpStackLocation(Irp);
#010         PDEVICE_CAPABILITIES deviceCapabilities;
#011         //从 IO 堆栈中得到 DEVICE_CAPABILITIES
#012         deviceCapabilities = IrpStack->Parameters.DeviceCapabilities.
Capabilities;
#013         //枚举
#014         for(int ds=PowerSystemWorking;ds<PowerSystemMaximum;ds++)
#015             KdPrint(("Capabilities from bus: DeviceState[%d]=%d", ds, device
Capabilities->DeviceState[ds]));
#016         DEVICE_POWER_STATE dps;
#017
#018         SetMostPoweredState( PowerSystemWorking, PowerDeviceD0);
#019         SetMostPoweredState( PowerSystemSleeping1, PowerDeviceD3);
#020         SetMostPoweredState( PowerSystemSleeping2, PowerDeviceD3);
#021         SetMostPoweredState( PowerSystemSleeping3, PowerDeviceD3);
#022         SetMostPoweredState( PowerSystemHibernate, PowerDeviceD3);
#023         SetMostPoweredState( PowerSystemShutdown, PowerDeviceD3);
#024
#025         //重点就是这句话了
#026         deviceCapabilities->Removable=TRUE;
#027         for(ds=PowerSystemWorking;ds<PowerSystemMaximum;ds++)
#028             KdPrint(("Capabilities now: DeviceState[%d]=%d", ds, device
Capabilities->DeviceState[ds]));
#029     }
#030     //结束 IRP 请求
#031     return CompleteIrp( Irp, status, Irp->IoStatus.Information);
#032 }
```

此段代码可以在配套光盘中本章的 Test1 目录下找到。

14.4 小结

本章主要介绍了如何在 WDM 驱动程序中进行电源处理。电源处理主要是处理好电源状态和设备状态。有关电源的 IRP 有两个，一个是 IRP_MJ_POWER，另一个是 IRP_MN_QUERY_CAPABILITIES。WDM 驱动程序主要是通过处理这两个 IRP 来实现电源管理。在 WDM 驱动程序中，一般是将这些 IRP 转发到总线驱动处理。

第 3 篇

实用篇

第 15 章 I/O 端口操作

第 16 章 PCI 设备驱动

第 17 章 USB 设备驱动

第 18 章 SDIO 设备驱动

第 19 章 虚拟串口设备驱动

第 20 章 摄像头设备驱动程序

第 15 章 I/O 端口操作

很多 Windows 驱动程序都需要进行 I/O 端口操作。I/O 端口操作在 Window 操作系统中属于特权命令，必须在内核模式下运行，而不能在用户模式下运行。这也是 I/O 操作都必须在驱动程序中实现的原因。本章归纳了多种 I/O 操作的方法，并且给出了这些方法实现的代码。

15.1 概述

在 DOS 操作系统中，C 语言提供了操作 I/O 端口的函数接口。这些函数会直接或者间接调用 IN 和 OUT 汇编指令。而对于 Windows 操作系统而言，这些操作不能在用户模式下执行，而只能在内核模式下执行。

15.1.1 从 DOS 说起

I/O 端口的操作主要有两条，即 IN 指令和 OUT 指令。这两条指令在 DOS 环境中，可以随便使用。但是在 Windows 2000 以后的操作系统中，这两条指令被当做是“特权”指令。“特权”指令不能在 Ring3 上运行，而必须在 Ring0 上运行。DOS 时代的程序员可以使用 Turbo C 中的两组 I/O 端口操作的函数。

(1) 第一组函数与输入相关，分别是_inpb、_inpw 及_inpd 函数，分别对应着 8 位、16 位及 32 位输入。

(2) 第二组函数与输出相关，分别是_outpb、_outpw 及_outpd 函数，分别对应着 8 位、16 位及 32 位输出。

但是以上 6 个函数无法在 Windows 中使用。本章将给出几种不同的方法来实现 I/O 端口操作。

15.1.2 汇编实现

Turbo C 中提供了 6 个输入输出函数内部调用的汇编指令中的 IN 和 OUT 指令，为了解如何使用 IN 和 OUT 指令，本节将 IN 和 OUT 封装成 6 个函数。这 6 个函数起到了与_inpw、_inpw、_inpd、_outpw、_outpw 及_outpd 函数相同的功能。

在 VC 中，可以使用__asm 关键字直接在 C 代码中加入汇编代码，方便 C/C++ 程序员方便使用汇编代码的功能。例如：

```
#001 //插入一句 NOP 指令
#002 __asm
#003 {
#004     NOP;
#005 }
```

利用 VC 提供的__asm 关键字可以方便地在 C 代码中插入汇编代码，并且允许 C 代码和汇编代码混编。下面的代码利用__asm 关键字，用 C 语言将 I/O 端口操作封装成 6 个函数。

```
#001 //8 位 IO 输入
#002 UCHAR In_8 (PUCHAR Port)
#003 {
#004     UCHAR Value;
#005     __asm
#006     {
#007         //将 PORT 地址传入 EDX
#008         mov edx, Port
#009         //读取端口
#010         in al, dx
#011         //将读取端口的数值传入 Value 里
#012         mov Value, al
#013         //插入几个空指令
#014         nop
#015         nop
#016     }
#017     return(Value);
#018 }
#019 //16 位 IO 输入
#020 USHORT In_16 (PUSHORT Port)
#021 {
#022     USHORT Value;
#023     __asm
#024     {
#025     {
#026         mov edx, Port //将 PORT 传入 EDX
#027         in ax, dx //执行 16 位 IO 输入
#028         mov Value, ax
#029         //插入几个空指令
#030         nop
#031         nop
#032     }
#033     return(Value);
```



```

#034 }
#035 //32 位 IO 输入
#036 ULONG In_32 (PULONG Port)
#037 {
#038     ULONG Value;
#039     __asm
#040     {
#041         mov edx, Port    //将 PORT 传入 EDX
#042         in  eax, dx      //执行 32 位 IO 输入
#043         mov Value, eax
#044         //插入几个空指令
#045         nop
#046         nop
#047     }
#048     return(Value);
#049 }
#050 //32 位 IO 输出
#051 void Out_32(PULONG Port,ULONG Value)
#052 {
#053     __asm
#054     {
#055         mov edx, Port    //将 PORT 传入 EDX
#056         mov eax, Value   //将 Value 传入 EAX
#057         out dx,eax       //执行 32 位 IO 输出
#058         //插入几个空指令
#059         nop
#060         nop
#061     }
#062 }
#063 //16 位 IO 输出
#064 void Out_16 (PUSHORT Port,USHORT Value)
#065 {
#066     __asm
#067     {
#068         mov edx, Port    //将 PORT 传入 EDX
#069         mov ax, Value    //将 Value 传入 AX
#070         out dx,ax        //执行 16 位 IO 输出
#071         //插入几个空指令
#072         nop
#073         nop
#074     }
#075 }
#076 //8 位 IO 输出
#077 void Out_8 (PUCHAR Port,UCHAR Value)
#078 {
#079     __asm
#080     {
#081         mov edx, Port    //将 PORT 传入 EDX
#082         mov al, Value    //将 Value 传入 AL
#083         out dx,al        //执行 8 位 IO 输出
#084         //插入几个空指令
#085         nop
#086         nop
#087     }
#088 }

```

此段代码可以在配套光盘中本章的 Test1 目录下找到。

现在此 6 个函数还不能正确执行。这是因为在 Ring3 上, IN 和 OUT 指令是不允许运行的。例如执行下面的语句:

```
Out_8((PUCHAR)0x378,0);
```

该语句是对 0x378 端口进行 OUT 操作 (0x378 端口是打印机并口的一个端口)。如果在应用程序中运行这句话, 操作系统会引发一个异常, 如图 15-1 所示。



图 15-1 端口操作的异常通知

15.1.3 DDK 实现

DDK 同样提供了类似的端口操作函数。这些函数同样分为两组, 一组负责输入操作, 另一组负责输出操作。如表 15-1 所示, 列出了 DDK 中所有与端口输入、输出相关的函数。

表 15-1 DDK 中有关端口操作函数

函 数	描 述
READ_PORT_UCHAR	8 位输入
READ_PORT_USHORT	16 位输入
READ_PORT_ULONG	32 位输入
READ_PORT_BUFFER_UCHAR	8 位输入 (连续多个 8 位)
READ_PORT_BUFFER_USHORT	16 位输入 (连续多个 16 位)
READ_PORT_BUFFER_ULONG	32 位输入 (连续多个 32 位)
WRITE_PORT_UCHAR	8 位输出
WRITE_PORT_USHORT	16 位输出
WRITE_PORT_ULONG	32 位输出
WRITE_PORT_BUFFER_UCHAR	8 位输出 (连续多个 8 位)
WRITE_PORT_BUFFER_USHORT	16 位输出 (连续多个 16 位)
WRITE_PORT_BUFFER_ULONG	32 位输出 (连续多个 32 位)

下面的代码演示了如何使用这些函数。需要注意的是, 编译程序的时候需要加入 HAL.lib 库, 因为以上函数的实现代码位于 HAL.dll 中。

```
#001 //8 位的输入
#002 UCHAR Ret8 = READ_PORT_UCHAR((PUCHAR)0x379);
#003 //16 位的输入
#004 USHORT Ret16 = READ_PORT_USHORT((PUSHORT)0x379);
#005 //32 位的输入
#006 ULONG Ret32 = READ_PORT_ULONG((PULONG)0x379);
```



```
#007 //8 位的输出
#008 WRITE_PORT_UCHAR((PUCHAR)0x378,0);
#009 //16 位的输出
#010 WRITE_PORT_USHORT((PUSHORT)0x378,0);
#011 //32 位的输出
#012 WRITE_PORT_ULONG((PULONG)0x378,0);
```

此段代码可以在配套光盘中本章的 Test1 目录下找到。

15.2 工具软件 WinIO

WinIO 是一个专门操作 I/O 端口的第三方库，通过该库我们可以在 Windows 上方便地操作 I/O 端口。

15.2.1 WinIO 简介

WinIO 是由 Yariv Kaplan 编写，这个库有如下特点：WinIO 库通过使用内核模式下设备驱动程序和其他一些底层编程技巧绕过 Windows 安全保护机制，允许 32 位 Windows 程序直接对 I/O 口进行操作。通过使用一种内核模式的设备驱动器和其他几种底层编程技巧，它绕过了 Windows 系统的保护机制。

Windows NT/2000/XP 下，WinIO 函数库只允许被具有管理者权限的应用程序调用。如果使用者不是以管理者的身份进入，则 WinIO.DLL 不能被安装，也不能激活 WinIO 驱动程序。通过在管理者权限下安装驱动器软件就可以克服这种限制。然而，在这种情况下，ShutdownWinIO 函数不能在应用程序结束之前被调用，因为该函数会将 WinIO 驱动程序从系统注册表中删除。

15.2.2 使用方法

WinIO 会提供 5 个文件，分别是 WinIO.dll、WinIO.h、WinIO.lib、WinIO.sys、WINIO.VXD。其中 WINIO.VXD 是 Windows 95、Windows 98、Windows Me 的驱动程序，这里不会用到。WinIO.sys 是 Windows NT、Windows 2000 及以后版本 Windows 的驱动程序。

WinIO.dll 封装了驱动程序调用函数，WinIO.lib 是用来与应用程序链接编译的，WinIO.h 提供了封装函数的声明。使用时必须将 WinIO.sys 和应用程序放在同一个目录。

WinIO 封装了多个函数，读者可以参考 WinIO.h 中的内容。这里简单介绍一下其中的几个常用函数。

```
WINIO_API bool _stdcall InitializeWinIo();
```

此函数用来加载 WinIO 驱动程序。

```
WINIO_API void _stdcall ShutdownWinIo();
```

此函数用来卸载 WinIO 驱动程序。

```
WINIO_API bool _stdcall SetPortVal(WORD wPortAddr, DWORD dwPortVal, BYTE bSize);
```

此函数是用来对 I/O 端口进行输出操作，其中 bSize 表示输出的大小。bSize=1 时，代表输出 1 个字节，即 8 位。bSize=2 时，代表输出 2 个字节，即 16 位。bSize=4 时，代表输出 4 个字节，即 32 位。

```
WINIO_API bool _stdcall GetPortVal(WORD wPortAddr, PDWORD pdwPortVal, BYTE bSize);
```

此函数是用来对 I/O 端口进行输入操作，其中 bSize 表示输入的大小。bSize=1 时，代表输入 1 个字节，即 8 位。bSize=2 时，代表输入 2 个字节，即 16 位。bSize=4 时，代表输入 4 个字节，即 32 位。

下面的代码简单演示了如何使用 WinIO。

```
#001 #include <Windows.h>
#002 #include <stdio.h>
#003 #include "..\winiolib\WinIo.h"
#004
#005 int main()
#006 {
#007     //打开 WinIO 驱动
#008     bool bRet = InitializeWinIo();
#009     if (bRet)
#010     {
#011         printf("Load Dirver successfully!\n");
#012
#013         //对 0x378 端口进行输出操作,8 位操作
#014         SetPortVal(0x378,0,1);
#015
#016         //关闭 WinIO 驱动
#017         ShutdownWinIo();
#018     }
#019     return 0;
#020 }
```

此段代码可以在配套光盘中本章的 Test1 目录下找到。

15.3 端口操作实现方法一

本节介绍第一种操作 I/O 端口的方法，这种方法使用 DDK 提供的 6 个操作端口的内核函数。

15.3.1 驱动端程序

下面是驱动程序的主要代码，主要是对应用程序提供了两组 I/O 控制码 (READ_PORT 和 WRITE_PORT)。针对这两种 I/O 控制码，分别使用了 READ_PORT_UCHAR、READ_PORT_USHORT、READ_PORT_ULONG、WRITE_PORT_UCHAR、WRITE_PORT_BUFFER_USHORT 及 WRITE_PORT_ULONG6 个 DDK 函数的调用。

Windows 驱动开发技术详解

```
#001 #pragma PAGEDCODE
#002 NTSTATUS HelloDDKDeviceIOControl(IN PDEVICE_OBJECT pDevObj,
#003                                     IN PIRP pIrp)
#004 {
#005     NTSTATUS status = STATUS_SUCCESS;
#006     KdPrint(("Enter HelloDDKDeviceIOControl\n"));
#007
#008     //得到当前堆栈
#009     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
#010     //得到输入缓冲区大小
#011     ULONG cbin = stack->Parameters.DeviceIoControl.InputBufferLength;
#012     //得到输出缓冲区大小
#013     ULONG cbout = stack->Parameters.DeviceIoControl.OutputBufferLength;
#014     //得到 IOCTL 码
#015     ULONG code = stack->Parameters.DeviceIoControl.IoControlCode;
#016
#017     ULONG info = 0;
#018
#019     switch (code)
#020     {
#021         // process request
#022         case READ_PORT:
#023         {
#024             KdPrint(("READ_PORT\n"));
#025             //缓冲区方式 IOCTL
#026             //显示输入缓冲区数据
#027             PULONG InputBuffer = (PULONG)pIrp->AssociatedIrp.SystemBuffer;
#028             ULONG port = (ULONG)(*InputBuffer);
#029             InputBuffer++;
#030             UCHAR method = (UCHAR)(*InputBuffer);
#031
#032             //操作输出缓冲区
#033             PULONG OutputBuffer = (PULONG)pIrp->AssociatedIrp.SystemBuffer;
#034
#035             if (method==1) //8 位操作
#036             {
#037                 *OutputBuffer = READ_PORT_UCHAR((PUCHAR)port);
#038             }
#039             else if(method==2) //16 位操作
#040             {
#041                 *OutputBuffer = READ_PORT_USHORT((PUSHORT)port);
#042             }
#043             else if(method==4) //32 位操作
#044             {
#045                 *OutputBuffer = READ_PORT_ULONG((PULONG)port);
#046             }
#047
#048             //设置实际操作输出缓冲区长度
#049             info = 4;
#050             break;
#051         }
#052         case WRITE_PORT:
#053         {
#054             KdPrint(("WRITE_PORT\n"));
#055             //缓冲区方式 IOCTL
#056             //显示输入缓冲区数据
#057             PULONG InputBuffer = (PULONG)pIrp->AssociatedIrp.SystemBuffer;
#058             ULONG port = (ULONG)(*InputBuffer);
#059             InputBuffer++;
```

```

#058         UCHAR method = (UCHAR)(*InputBuffer);
#059         InputBuffer;
#060         ULONG value = (ULONG)(*InputBuffer);
#061
#062         //操作输出缓冲区
#063         PULONG OutputBuffer = (PULONG)pIrp->AssociatedIrp.SystemBuffer;
#064
#065         if (method==1)          //8 位操作
#066         {
#067             WRITE_PORT_UCHAR((PUCHAR)port, (UCHAR)value);
#068         }else if(method==2)    //16 位操作
#069         {
#070             WRITE_PORT_USHORT((PUSHORT)port, (USHORT)value);
#071         }else if(method==4)    //32 位操作
#072         {
#073             WRITE_PORT_ULONG((PULONG)port, (ULONG)value);
#074         }
#075         //设置实际操作输出缓冲区长度
#076         info = 0;
#077         break;
#078     }
#079
#080     default:
#081         status = STATUS_INVALID_VARIANT;
#082     }
#083
#084     //设置 IRP 完成状态
#085     pIrp->IoStatus.Status = status;
#086     //设置 IRP 操作字节数
#087     pIrp->IoStatus.Information = info;
#088     //结束 IRP 请求
#089     IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#090
#091     KdPrint(("Leave HelloDDKDeviceIOControl\n"));
#092
#093     return status;
#094 }

```

此段代码可以在配套光盘中本章的 Test3 目录下找到。

15.3.2 应用程序端程序

由于在驱动程序端已经提供好了两组 I/O 控制码，所以通过 DeviceIoControl 向驱动发出请求即可。以下是应用端的实例代码。

```

#001 #include <windows.h>
#002 #include <stdio.h>
#003 //使用 CTL_CODE 必须加入 winioctl.h
#004 #include <winioctl.h>
#005 #include "..\NT_Driver\Ioctl.h"
#006 int main()
#007 {
#008     //打开设备句柄
#009     HANDLE hDevice =
#010         CreateFile("\\\\.\\HelloDDK",

```



```

#011             GENERIC_READ | GENERIC_WRITE,
#012             0,           // 不共享
#013             NULL,
#014             OPEN_EXISTING,
#015             FILE_ATTRIBUTE_NORMAL,
#016             NULL );
#017             //判断是否成功打开设备句柄
#018             if (hDevice == INVALID_HANDLE_VALUE)
#019             {
#020                 printf("Failed to obtain file handle to device: "
#021                     "%s with Win32 error code: %d\n",
#022                     "MyWDMDevice", GetLastError() );
#023                 return 1;
#024             }
#025
#026             DWORD dwOutput ;
#027             DWORD inputBuffer[3] =
#028             {
#029                 0x378, //对 0x378 进行操作
#030                 0, //输出字节 0
#031                 1//1 代表 8 位操作, 2 代表 16 位操作, 4 代表 32 位操作
#032             };
#033
#034             //类似于 Out_8((PUCHAR)0x378,0);
#035             DeviceIoControl(hDevice, WRITE_PORT, inputBuffer, sizeof(inputBuffer),
NULL, 0, &dwOutput, NULL);
#036             //关闭句柄
#037             CloseHandle(hDevice);
#038
#039             return 0;
#040         }

```

此段代码可以在配套光盘中本章的 Test3 目录下找到。

15.4 端口操作实现方法二

本节介绍第二种操作 I/O 端口的方法。这种方法其实是第一种方法的一种变通。笔者将 DDK 提供的 6 个端口操作函数替换成笔者自己编写的 6 个函数。这 6 个函数使用内联汇编写成。

15.4.1 驱动端程序

在 15.3 节中, 驱动程序使用的是 DDK 提供的端口操作函数。而在 15.1.1 节中曾经给出 6 个函数 In_8、In_16、In_32、Out_8、Out_16、Out_32, 这 6 个函数其实就是笔者对于端口操作汇编语言进行的封装。这 6 个函数分别模拟了内核函数 READ_PORT_UCHAR、READ_PORT_USHORT、READ_PORT_ULONG、WRITE_PORT_UCHAR、WRITE_PORT_BUFFER_USHORT、WRITE_PORT_ULONG。

然而 In_8、In_16、In_32、Out_8、Out_16、Out_32 在应用程序中, 是无法实现的。

原因就是这 6 个函数用到了需要特权级别的汇编语句。因此，笔者在驱动程序中调用这 6 个函数，并对其进行封装，对应用程序提供了两个 IO 控制码的接口，使应用程序可以方便地调用端口操作。

以下是驱动端的主要代码：

```
#001 #pragma PAGEDCODE
#002 NTSTATUS HelloDDKDeviceIOControl(IN PDEVICE_OBJECT pDevObj,
#003                                     IN PIRP pIrp)
#004 {
#005     NTSTATUS status = STATUS_SUCCESS;
#006     KdPrint(("Enter HelloDDKDeviceIOControl\n"));
#007
#008     //得到当前堆栈
#009     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
#010     //得到输入缓冲区大小
#011     ULONG cbin = stack->Parameters.DeviceIoControl.InputBufferLength;
#012     //得到输出缓冲区大小
#013     ULONG cbout = stack->Parameters.DeviceIoControl.OutputBufferLength;
#014     //得到 IOCTL 码
#015     ULONG code = stack->Parameters.DeviceIoControl.IoControlCode;
#016
#017     ULONG info = 0;
#018
#019     switch (code)
#020     {
#021         // process request
#022         case READ_PORT:
#023         {
#024             KdPrint(("READ_PORT\n"));
#025             //缓冲区方式 IOCTL
#026             //显示输入缓冲区数据
#027             PULONG InputBuffer = (PULONG)pIrp->AssociatedIrp.SystemBuffer;
#028             ULONG port = (ULONG)(*InputBuffer);
#029             InputBuffer++;
#030             UCHAR method = (UCHAR)(*InputBuffer);
#031
#032             //操作输出缓冲区
#033             PULONG OutputBuffer = (PULONG)pIrp->AssociatedIrp.SystemBuffer;
#034
#035             if (method==1) //8 位操作
#036             {
#037                 *OutputBuffer = In_8((PUCHAR)port);
#038             }
#039             else if (method==2) //16 位操作
#040             {
#041                 *OutputBuffer = In_16((PUSHORT)port);
#042             }
#043             else if (method==4) //32 位操作
#044             {
#045                 *OutputBuffer = In_32((PULONG)port);
#046             }
#047
#048             //设置实际操作输出缓冲区长度
#049             info = 4;
#050             break;
#051         }
#052         case WRITE_PORT:
#053         {
#054             KdPrint(("WRITE_PORT\n"));
#055             //缓冲区方式 IOCTL
#056             //显示输出缓冲区数据
#057             PULONG OutputBuffer = (PULONG)pIrp->AssociatedIrp.SystemBuffer;
#058             ULONG port = (ULONG)(*OutputBuffer);
#059             OutputBuffer++;
#060             UCHAR method = (UCHAR)(*OutputBuffer);
#061
#062             //操作输入缓冲区
#063             PULONG InputBuffer = (PULONG)pIrp->AssociatedIrp.SystemBuffer;
#064
#065             if (method==1) //8 位操作
#066             {
#067                 *InputBuffer = Out_8((PUCHAR)port);
#068             }
#069             else if (method==2) //16 位操作
#070             {
#071                 *InputBuffer = Out_16((PUSHORT)port);
#072             }
#073             else if (method==4) //32 位操作
#074             {
#075                 *InputBuffer = Out_32((PULONG)port);
#076             }
#077
#078             //设置实际操作输入缓冲区长度
#079             info = 4;
#080             break;
#081         }
#082     }
#083     status = STATUS_SUCCESS;
#084     return status;
#085 }
```



```

#051         KdPrint(("WRITE_PORT\n"));
#052         //缓冲区方式 IOCTL
#053         //显示输入缓冲区数据
#054         PULONG InputBuffer = (PULONG)pIrp->AssociatedIrp.SystemBuffer;
#055         ULONG port = (ULONG)(*InputBuffer);
#056         InputBuffer++;
#057         UCHAR method = (UCHAR)(*InputBuffer);
#058         InputBuffer++;
#059         ULONG value = (ULONG)(*InputBuffer);
#060
#061         //操作输出缓冲区
#062         PULONG OutputBuffer = (PULONG)pIrp->AssociatedIrp.SystemBuffer;
#063
#064         if (method==1)          //8 位操作
#065         {
#066             Out_8((PUCHAR)port, (UCHAR)value);
#067         }else if(method==2)    //16 位操作
#068         {
#069             Out_16((USHORT)port, (USHORT)value);
#070         }else if(method==4)    //32 位操作
#071         {
#072             Out_32((PULONG)port, (ULONG)value);
#073         }
#074
#075         //设置实际操作输出缓冲区长度
#076         info = 0;
#077         break;
#078     }
#079     default:
#080         status = STATUS_INVALID_VARIANT;
#081 }
#082
#083 //设置 IRP 完成状态
#084 pIrp->IoStatus.Status = status;
#085 //设置 IRP 操作字节数
#086 pIrp->IoStatus.Information = info;
#087 //将 IRP 请求结束
#088 IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#089
#090 KdPrint(("Leave HelloDDKDeviceIOControl\n"));
#091
#092 return status;
#093 }

```

此段代码可以在配套光盘中本章的 Test4 目录下找到。

15.4.2 应用程序端程序

由于在驱动程序端已经提供好了两组 I/O 控制码，所以通过 DeviceIoControl 向驱动发出请求即可。以下是应用端的实例代码：

```

#001 #include <windows.h>
#002 #include <stdio.h>
#003 //使用 CTL_CODE 必须加入 winioctl.h
#004 #include <winioctl.h>

```

```

#005 #include "..\NT_Driver\Ioctl.h"
#006
#007 int main()
#008 {
#009     //打开设备句柄
#010     HANDLE hDevice =
#011         CreateFile("\\\\.\\HelloDDK",
#012                 GENERIC_READ | GENERIC_WRITE,
#013                 0,
#014                 NULL,
#015                 OPEN_EXISTING,
#016                 FILE_ATTRIBUTE_NORMAL,
#017                 NULL );
#018
#019     if (hDevice == INVALID_HANDLE_VALUE)
#020     {
#021         printf("Failed to obtain file handle to device: *
#022             "%s with Win32 error code: %d\n",
#023             "MyWDMDevice", GetLastError() );
#024         return 1;
#025     }
#026
#027     DWORD dwOutput ;
#028     DWORD inputBuffer[3] =
#029     {
#030         0x378, //对 0x378 进行操作
#031         1, //1 代表 8 位操作, 2 代表 16 位操作, 4 代表 32 位操作
#032         0, //输出字节 0
#033     }
#034     //类似于 Out_8((PUCHAR)0x378, 0);
#035     DeviceIoControl(hDevice, WRITE_PORT, inputBuffer, sizeof(inputBuffer),
NULL, 0, &dwOutput, NULL);
#036     //关闭设备句柄
#037     CloseHandle(hDevice);
#038     return 0;
#039 }

```

此段代码可以在配套光盘中本章的 Test4 目录下找到。

15.5 端口操作实现方法三

本节使用的主要方法有别于方法一和方法二。其思路是在应用程序的一个函数中直接对端口操作，然后通过某种办法将这个函数从用户模式提升至用内核模式。

15.5.1 驱动端程序

本节的方法主要是将应用程序的一个函数从用户模式提升至内核模式，这样在应用程序中该函数就可以像驱动程序一样，可以做一些特权级的操作。其中包括对端口操作，对高于 2GB 的内核内存的访问能力。

提升用户模式的方法是将应用程序的一个函数指针传递给驱动程序，在驱动程序接收

到这个函数指针后，在内核模式下执行此函数。

下面是驱动程序的主要代码：

```
#001 typedef void (*KernelFun)();
#002 //使此函数运行在分页内存
#003 #pragma PAGEDCODE
#004 NTSTATUS HelloDDKDeviceIOControl(IN PDEVICE_OBJECT pDevObj,
#005                                   IN PIRP pIrp)
#006 {
#007     NTSTATUS status = STATUS_SUCCESS;
#008     KdPrint(("Enter HelloDDKDeviceIOControl\n"));
#009
#010     //得到当前堆栈
#011     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
#012     //得到输入缓冲区大小
#013     ULONG cbin = stack->Parameters.DeviceIoControl.InputBufferLength;
#014     //得到输出缓冲区大小
#015     ULONG cbout = stack->Parameters.DeviceIoControl.OutputBufferLength;
#016     //得到 IOCTL 码
#017     ULONG code = stack->Parameters.DeviceIoControl.IoControlCode;
#018     ULONG info = 0;
#019
#020     switch (code)
#021     {
#022         // process request
#023         case IOCTL_KERNEL_FUNCTION:
#024         {
#025             KdPrint(("IOCTL_KERNEL_FUNCTION\n"));
#026             KernelFun* InputBuffer =
(KernelFun*)pIrp->AssociatedIrp.SystemBuffer;
#027             //将应用程序提供的函数地址提供给驱动程序
#028             KernelFun foo = *InputBuffer;
#029             //在内核模式下调用应用程序的函数
#030             foo();
#031             info = 0;
#032         }
#033         default:
#034             status = STATUS_INVALID_VARIANT;
#035     }
#036
#037     //设置 IRP 完成状态
#038     pIrp->IoStatus.Status = status;
#039     //设置 IRP 操作字节数
#040     pIrp->IoStatus.Information = info;
#041     //结束 IRP 请求
#042     IoCompleteRequest(pIrp, IO_NO_INCREMENT);
#043     KdPrint(("Leave HelloDDKDeviceIOControl\n"));
#044     return status;
#045 }
```

此段代码可以在配套光盘中本章的 Test5 目录下找到。

15.5.2 应用程序端程序

在驱动程序中，已经留好了一个接口，通过 I/O 控制码将应用程序的函数指针传递给

驱动程序。以下是应用程序中的主要代码：

```
#001 void KernelModeFunction()
#002 {
#003     //运行在 ring0
#004     //所以能执行 IO 操作
#005     Out_8((PUCHAR)0x378,0);
#006 }
#007 int main()
#008 {
#009     //打开设备句柄
#010     HANDLE hDevice =
#011         CreateFile("\\\\.\\HelloDDK",
#012                 GENERIC_READ | GENERIC_WRITE,
#013                 0,
#014                 NULL,
#015                 OPEN_EXISTING,
#016                 FILE_ATTRIBUTE_NORMAL,
#017                 NULL );
#018     //判断是否成功打开设备句柄
#019     if (hDevice == INVALID_HANDLE_VALUE)
#020     {
#021         printf("Failed to obtain file handle to device: "
#022             "%s with Win32 error code: %d\n",
#023             "MyWDMDevice", GetLastError() );
#024         return 1;
#025     }
#026
#027     DWORD dwOutput ;
#028     //类似于 Out_8((PUCHAR)0x378,0);
#029     DWORD Function_Address = (DWORD)KernelModeFunction;
#030     //将 KernelModeFunction 函数的函数地址传入驱动
#031     DeviceIoControl(hDevice, IOCTL_KERNEL_FUNCTION, &Function_Address, 4, NULL,
0, &dwOutput, NULL);
#032     //关闭设备句柄
#033     CloseHandle(hDevice);
#034     return 0;
#035 }
```

此段代码可以在配套光盘中本章的 Test5 目录下找到。

15.6 端口操作实现方法四

在 Windows 中还有一种不常用的方法，可以访问 I/O 端口，本节将对这种方法进行介绍。

15.6.1 原理

本书曾一再强调，Windows NT 及 Windows 2000 对 I/O 端口有着严格控制。它们与 Windows 95 和 Windows 98 有些不同，如果没有足够的权限读写端口，Windows NT/2000 将产生一个 exception privileged instruction 错误，不仅 Windows NT 如此，任何一款 386 或

更高档次的处理器在保护模式下运行时都会产生类似的错误。

在保护模式下存取 I/O 端口受两个条件限制，一个是 EFLAGS 寄存器中的 I/O privilege level (IOPL)，另一个是 Task State Segment (TSS) 任务状态段中的 I/O 允许位图设置。

I/O 允许位图设置可以让不具备足够权限的程序（如 USER 方式的程序）存取 I/O 端口。当一条指令被执行时，处理器首先检查这个任务是否具备存取端口的权限。如果具备权限，就允许其操作端口，如果任务不具备存取 I/O 的权限，处理器则检查 I/O 允许位图设置。

I/O 允许位图设置是利用一个位代表每个 I/O 地址。如果一个对应某端口的位被设置，则指令会产生一个 exception 错误。而如果这个位被清除，就会避免错误产生。这就提供了一种让程序存取特定端口的能力。通常一个任务只有一个 I/O 位图设置。

有两种方法解决 I/O 存取的问题，一是写一个运行在 ring0 级的设备驱动程序，也就是本章前面介绍的几个例子。

另一种可替换的方法是修改 I/O 允许位图设置，允许一个特定的任务存取特定的 I/O 端口。这种允许用户模式方式的程序在 Ring3 级按照 I/O 允许位图设置，可以不受限制地访问 I/O 端口。其不是一种最佳的方法，但其够让现有的程序运行在 windows NT/2000 下。

15.6.2 驱动端程序

为了修改 I/O 位图，使用了一个未文档化的内核函数 Ke386SetIoAccessMap。由于是未文档化，所以需要程序员自己声明。另外，由于本例是由 C++ 写成，所以声明此函数的时候必须按照 C 语言的方式声明，即用 extern "C" 将函数声明包起来。

以下是驱动程序修改 I/O 位图的主要代码：

```
#001  #pragma PAGEDCODE
#002  NTSTATUS HelloDDKDeviceIOControl(IN PDEVICE_OBJECT pDevObj,
#003                                     IN PIRP pIrp)
#004  {
#005      NTSTATUS status = STATUS_SUCCESS;
#006      KdPrint(("Enter HelloDDKDeviceIOControl\n"));
#007
#008      //得到当前堆栈
#009      PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
#010      //得到输入缓冲区大小
#011      ULONG cbin = stack->Parameters.DeviceIoControl.InputBufferLength;
#012      //得到输出缓冲区大小
#013      ULONG cbout = stack->Parameters.DeviceIoControl.OutputBufferLength;
#014      //得到 IOCTL 码
#015      ULONG code = stack->Parameters.DeviceIoControl.IoControlCode;
#016
#017      ULONG info = 0;
#018      //判断是哪种 IOCTL 码
#019      switch (code)
#020      {
#021          case IOCTL_ENABLEDIRECTIO:
```

```

#022     {
#023         KdPrint(("IOCTL_ENABLEDIRECTIO\n"));
#024         pIOPM = (UCHAR*)MmAllocateNonCachedMemory(IOPM_SIZE);
#025         if (pIOPM)
#026         {
#027             //将 IOPM 初始化为零
#028             RtlZeroMemory(pIOPM, IOPM_SIZE);
#029             Ke386IoSetAccessProcess(PsGetCurrentProcess(), 1);
#030             Ke386SetIoAccessMap(1, pIOPM);
#031         }
#032         else
#033             pIrp->IoStatus.Status = STATUS_INSUFFICIENT_RESOURCES;
#034     }
#035     case IOCTL_DISABLEDIRECTIO:
#036     {
#037         KdPrint(("IOCTL_DISABLEDIRECTIO\n"));
#038         if (pIOPM)
#039         {
#040             //设置 IOPM
#041             Ke386IoSetAccessProcess(PsGetCurrentProcess(), 0);
#042             Ke386SetIoAccessMap(1, pIOPM);
#043             //释放内存
#044             MmFreeNonCachedMemory(pIOPM, IOPM_SIZE);
#045             pIOPM = NULL;
#046         }
#047     }
#048     default:
#049         status = STATUS_INVALID_VARIANT;
#050 }
#051
#052 //设置 IRP 完成状态
#053 pIrp->IoStatus.Status = status;
#054 //设置 IRP 操作字节数
#055 pIrp->IoStatus.Information = info;
#056 //结束 IRP 请求
#057 IoCompleteRequest(pIrp, IO_NO_INCREMENT);
#058 KdPrint(("Leave HelloDDKDeviceIOControl\n"));
#059 return status;
#060 }

```

此段代码可以在配套光盘中本章的 Test6 目录下找到。

15.6.3 应用程序端程序

在上面的驱动程序中，为应用程序提供了两个 I/O 控制码接口，即 IOCTL_ENABLEDIRECTIO 和 IOCTL_DISABLEDIRECTIO。应用程序可以通过这两个 I/O 控制码和驱动程序进行通信。以下是应用程序中的主要代码：

```

#001 int main()
#002 {
#003     //打开设备句柄
#004     HANDLE hDevice =
#005         CreateFile("\\\\.\\HelloDDK",
#006                 GENERIC_READ | GENERIC_WRITE,
#007                 0, // share mode none

```



```

#008             NULL,           // no security
#009             OPEN_EXISTING,
#010             FILE_ATTRIBUTE_NORMAL,
#011             NULL );         // no template
#012             //判断是否成功打开设备句柄
#013             if (hDevice == INVALID_HANDLE_VALUE)
#014             {
#015                 printf("Failed to obtain file handle to device: "
#016                     "%s with Win32 error code: %d\n",
#017                     "MyWDMDevice", GetLastError() );
#018                 return 1;
#019             }
#020
#021             DWORD dwOutput ;
#022             //开启直接 IO
#023             DeviceIoControl(hDevice, IOCTL_ENABLEDIRECTIO, NULL, 0, NULL, 0, &dwOutput,
NULL);
#024             //进行 8 位输出操作
#025             Out_8((PUCHAR)0x378,0);
#026             //关闭直接 IO
#027             DeviceIoControl(hDevice, IOCTL_DISABLEDIRECTIO, NULL, 0, NULL, 0, &dwOutput,
NULL);
#028             //关闭设备句柄
#029             CloseHandle(hDevice);
#030             return 0;
#031         }

```

这段代码可以在配套光盘中本章的 Test6 目录下找到。

15.7 驱动 PC 喇叭

使 PC 喇叭发出声音，主要是正确设置 PC 喇叭所对应的一些寄存器。这些寄存器对应固定的 I/O 端口，本节利用前面讲解的对 I/O 端口操作的技巧，针对这些端口进行操作，从而达到使 PC 喇叭出声的效果。

15.7.1 可编程定时器

每个 PC 系统至少包含一个 8253 可编程时钟或等价的芯片。这个时钟包含三个独立的 16 位时钟。时钟 0 用于基本系统时钟，时钟 1 用于 PC 系统上的 DRAM 刷新，时钟 2 用于一般的应用程序，例如，扬声器音调控制。

本节主要介绍如何使用 PC 喇叭，对于可编程定时器，主要是使用时钟 2，如图 15-2 所示。

使用 8253 定时器，需要对其控制寄存器和数据寄存器进行设置。

(1) 控制寄存器

初始化编程时需要对控制寄存器进行初始化，以便确定时钟的运行状态。其中：

- 第 0 位：如果是 0，代表是二进制计数。如果是 1，代表用 BCD 计数。
- 第 1~3 位：000 代表时钟以方式 0 运行。

- 第 4~5 位：如果是 00 代表计数器锁存，如果是 01 代表只读写 8 位字节，如果是 10 代表只读写高 8 位字节，如果是 11 代表先读低 8 位，后读高 8 位。
- 第 6~7 位：如果是 00 代表选用计数器 0，如果是 01 代表选用计数器 1，如果是 10 代表选择计数器 2，如果是 11 代表非法。

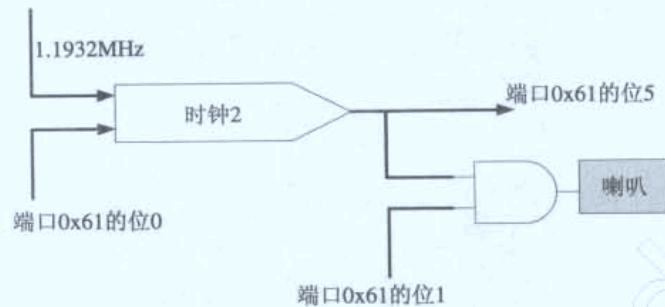


图 15-2 PC 喇叭原理图

(2) 数据寄存器

数据寄存器主要是对计数器的计数进行读写。在 PC 上，计数器 0 的数据寄存器端口是 0x40，计数器 1 的数据寄存器端口是 0x41，计数器 2 数据寄存器端口是 0x42。这三个计数器公用一个端口 0x43 作为其控制寄存器。另外，计数器 0 的输出会接到 8259A 芯片，作为第 0 号中断，如图 15-3 所示。

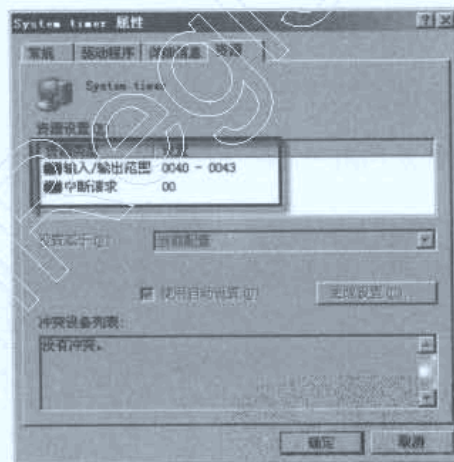


图 15-3 PC 计时器的端口

15.7.2 PC 喇叭

在 PC 中，喇叭（不是声卡）链接在计数器 2 的输出上，可以用软件的方式将其连接断开，后面会有所介绍。计数器 2 的输入频率是 1.932MHz。通过设置计数器 2 的分频计数，可以将 1.932MHz 进行分频。

分频后的频率输出被输出到 PC 喇叭上，通过设置不同的输出频率，喇叭就会产生不

Windows 驱动开发技术详解

同的声调。另外，可以通过软件的方式设置喇叭与输出的联通和断开。在 PC 中操作喇叭（不是声卡）的端口是 0x61，如图 15-4 所示。

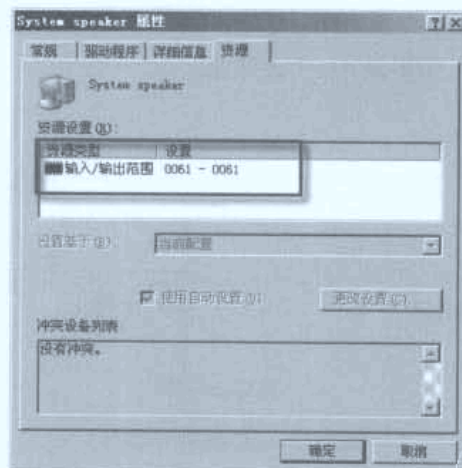


图 15-4 PC 喇叭的端口

端口 0x61 只有两位对 PC 喇叭有用，其中第 0 位是控制时钟 2 联通 PC 喇叭，如果为 0，则断开，如果为 1，则联通。其中第 1 位控制是否打开扬声器，如果是 0，则关闭扬声器，如果是 1，则打开扬声器：

15.7.3 操作代码

介绍完 PC 喇叭的工作原理以及各个相关的端口后，编写一段操作 PC 喇叭的测试代码就变得非常简单了。下面就是应用端口操作计时器 2 的相关端口，从而驱动 PC 喇叭的示例代码。

```
#001 //发音程序,参数 f 代表频率
#002 void Sound(HANDLE hDevice,int f)
#003 {
#004     //计数为 1193180/F
#005     USHORT B=1193180/f;
#006
#007     //从端口 0x61 取数
#008     UCHAR temp = In_8(hDevice,0x61);
#009     //两低位置 1
#010     temp = temp | 3;
#011     //输出到 0x61 端口
#012     Out_8(hDevice,0x61,temp);
#013
#014     //输出到 0x61 端口
#015     Out_8(hDevice,0x43,0xB6);
#016     //输出到 0x42 端口,写低 8 位
#017     Out_8(hDevice,0x42,B&0xFF);
#018     //输出到 0x42 端口,写高 8 位
#019     Out_8(hDevice,0x42,(B>>8)&0xFF);
```

```

#020 }
#021
#022 // 关闭声音
#023 void SoundOff(HANDLE hDevice)
#024 {
#025     //取端口 0x61 的字节
#026     UCHAR value = In_8(hDevice,0x61);
#027     //强制置最后两位为 0
#028     value = value & 0xFC;
#029     //返送端口 0x61
#030     Out_8(hDevice,0x61,value);
#031 }
#032
#033 int main()
#034 {
#035     //打开设备句柄
#036     HANDLE hDevice =
#037         CreateFile("\\\\.\\HelloDDK",
#038                 GENERIC_READ | GENERIC_WRITE,
#039                 0,
#040                 NULL,
#041                 OPEN_EXISTING,
#042                 FILE_ATTRIBUTE_NORMAL,
#043                 NULL );
#044     //判断设备是否成功打开
#045     if (hDevice == INVALID_HANDLE_VALUE)
#046     {
#047         printf("Failed to obtain file handle to device: "
#048             "%s with Win32 error code: %d\n",
#049             "MyWDMDevice", GetLastError() );
#050         return 1;
#051     }
#052     //产生 2KHz 频率的声音
#053     Sound(hDevice,2000);
#054     //持续 200ms
#055     Sleep(200);
#056     SoundOff(hDevice);
#057     //关闭设备句柄
#058     CloseHandle(hDevice);
#059     return 0;
#060 }

```

此段代码可以在配套光盘中本章的 Test7 目录下找到。

15.8 操作并口设备

操作 PC 上的并口设备，主要也是设置和读取并口设备对应的 I/O 端口。本节简单介绍 PC 并口所对应的一些 I/O 端口，并利用前面讲解的操作端口的方法，实现对 PC 并口进行操作。

15.8.1 并口设备简介

目前，在实验室和工业应用的各种控制系统中，串口是常用的计算机与外部控制系统

之间的数据传输通道。由于串行通信方便易行，所以应用广泛。但是使用串行通信，在实时性、速度、数据量等方面受到限制。而计算机的并行端口传输数据时是一次性传送 8 个位（一个字节）或更多，由于传输量较大，因此数据的传输速度要比串口快，在许多必须讲究传输速度的控制系统里，用 PC 并行端口与之连接就是一个很好的解决方案。

这里主要介绍计算机的标准配备并行端口即 25 针的母接头端口的应用，在此基础上可以运用相同的原理使用其他模式的并行端口。并行端口共有 25 支脚，但不是每支脚均被使用到。这些脚被区分为 3 种主要的功能，分别是用于数据的传送、检查打印机的状态及控制打印机，其接口如图 15-5 所示。

在 PC 机中，标准并行口使用 3 个 8 位的端口寄存器，PC 就是通过对这些寄存器，也就是所说的数据、状态、控制寄存器的读写访问并口的信号的。本文中使用一些通用的叫法，8 个数据位分别为 D0~D7，5 个状态位为 S3~S7，4 个控制位为 C0~C3。其中字母表示了端口寄存器，数字则表示该信号在寄存器中的位。

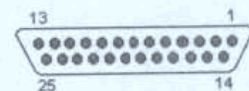


图 15-5 PC 打印机接口

15.8.2 并口寄存器

操作并口对于 PC 端来讲，主要操作三个寄存器，即数据寄存器、状态寄存器及控制寄存器。对于这三个寄存器的控制，可以完全控制并口。以下分别对这三个寄存器进行介绍。

1. 数据寄存器

数据端口或称数据寄存器（D0~D7）保存了写入数据输出端口的一字节信息。数据端口可以写入数据，也可以读出数据（即可擦写）；写进去的是希望从数据端口引脚输出的数据，不过读进来的也只是上次写进去的数据，或是原来保留在里面的数据，并不是从端口引脚输入 PC 的数据。数据端口引脚是 PIN2~PIN9，其定义如表 15-2 所示。

表 15-2 数据寄存器（即数据输出端口）

Bit	引脚: D-sub	信号名	信号源	是否在连接器处倒相
0	Pin2	D0	PC	否
1	Pin3	D1	PC	否
2	Pin4	D2	PC	否
3	Pin5	D3	PC	否
4	Pin6	D4	PC	否
5	Pin7	D5	PC	否
6	Pin8	D6	PC	否
7	Pin9	D7	PC	否

如果把这 8 支脚当做一般的数字输出的脚位看，则上述 8 支脚就相当于 8 个数字输

出的位置，就可以把它们当成是 8 个可以自由控制的输出点。当通过数据端口传送数据时，就是改变这 8 支脚的电平状态；而接受方也按照相同的编码原则解释，就可以获得传送的数据。

数据寄存器的 I/O 端口是 0x378 端口。

2. 状态寄存器

状态端口或称状态寄存器保存的是 5 个输入（S3～S7）的逻辑状态。S0～S2 位不出现在并口连接器中。除了 S0 以外，状态寄存器是只读的，读出的数据信息 是状态端口引脚上的逻辑状态。S0 是支持 EPP 传输并口的超时标志信息，可以用软件方法清零。在许多并口中，状态输入接有上拉电阻。状态端口引脚是 Pin10～Pin13、Pin15，其定义如表 15-3 所示。

表 15-3 状态寄存器（即状态输入端口）基地址+1

Bit	引脚: D-sub	信号名	信号源	是否在连接器处倒相
0		Time-Out		
1		未使用		
2		未使用		
3	Pin15	nError(nFault)	外设	否
4	Pin13	Select	外设	否
5	Pin12	PaperEnd	外设	否
6	Pin10	nAck	外设	否
7	Pin11	Busy	外设	是

上表中的基地址+1 指的是：如果我们的 LPT 地址是 378H，再加上 1 就是 379H；这个地址专门用来传递打印机的状态。和数据地址相比不同的是，这里的地址并非在连接器的脚位上均有对应点。此时的状态显示只有 5 个脚位有对应，位 S0～S2 是没有对应的——最起码是无法让计算机有对应的值可读取。

如果打印机接到并口上，那么打印机的状态将会通过这几支脚传送到 PC，程序只要去基地址+1 的位置读取数值即可知道现在打印机所处的状态。由于这几支 脚可以让打印机传送状态给 PC，那么可以把这几支脚位拿来当做数字输入的通道；可以让这几支脚位的状态发生电位的改变，而利用程序去读取这些脚位的数值，即可实现数据的输入。

3. 控制寄存器

控制端口或称控制寄存器保存了 C0～C3 的 4 位的控制信息。C4～C7 不出现在并口连接器中。一般来说，这些位被用来输出，然而大多数 SPP 中，控制位为集电极开路/漏极开路模式，也就是说，它们同样可以当做输入。要从控制位上读取外部逻辑信号，首先将向相应的输出写入“1”，然后读取控制寄存器的值即可。但是，为了提高交换速度，大多数支持 EPP 和 ECP 的接口中，控制位工作在不能当做输入的推拉模式下。在一些多模

式接口中，控制位采用的是改进型的推拉 模式，可以当做输入。控制端口引脚是 Pin1、Pin14、Pin16 和 Pin17，其定义如表 15-4 所示。

表 15-4 控制寄存器（即控制输出端口）基地址+2

Bit	引脚: D-sub	信号名	信号源	是否在连接器处倒相
0	Pin1	nStrobe	PC	是
1	Pin14	nAutoLF	PC	是
2	Pin16	nInit	PC	否
3	Pin17	nSelectIn	PC	是否
4		IRQ		
5		未使用		
6		未使用		
7		未使用		

上表中的基地址+2 指的是：如果我们的 LPT 地址是 378H，再加上 1 就是 37AH；这个地址专门用来控制打印机的动作。

如同数据的送出，程序只要将信息送往基地址+2 的地址中，就可以实现数据输出，接受端在相应引脚就可以接受到相应的逻辑电位状态。当控制端口的信号源为高电平时，这些引脚可以作为输入引脚，如同状态端口引脚一样。

在上述定义表格中，“是否在连接器处倒相”是指并口硬件将连接器与相应寄存器位之间的 4 个信号进行了倒相处理。具体说就是，S7、C0、C1、C3 信号的逻辑状态在连接器处是与相应寄存器位反相的。在对这些位进行写操作时，必须牢记写入的值应该与想在连接器处设置的值相反；当要对这些位进行读操作时，也必须记住所读取的值与连接器处的值相反。

计算机的标准配备并行端口除以上介绍的数据端口引脚 Pin2~Pin9、状态端口引脚 Pin15、Pin10~Pin13、控制端口引脚 Pin1、Pin14、pin16、Pin17 外，连接器上的其他引脚 Pin18~Pin25 是归地引脚 GND。

15.8.3 并口设备操作

在上一节介绍了并口寄存器的三个寄存器，其中每个寄存器占用一个 I/O 端口。并口设备的端口基地址是 0x378。也就是数据寄存器的 IO 地址是 0x378，状态寄存器的 IO 地址是 0x379，控制寄存器的 IO 地址是 0x37A。并口设备的 I/O 端口也可以通过设备管理器查询，单击打印机端口设备，查看属性，如图 15-6 所示。

通过上述的知识点，就完全可以操作并口设备了。下面的例子是为了演示控制并口设备的输出，其中数据寄存器的 IO 地址是 0x378，并且每隔 100ms 操作一次此端口，并将数据求反。

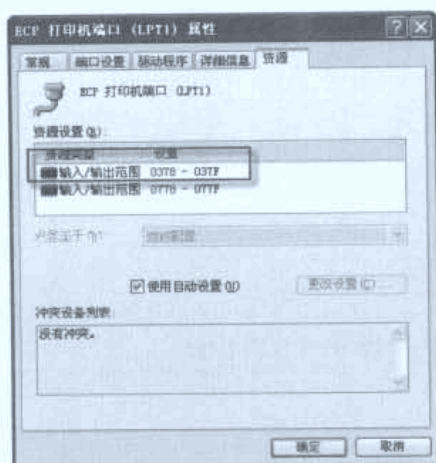


图 15-6 打印机接口端口

这样，在示波器查看第 2 管脚和第 3 管脚时，可以看到两个方波信号，周期是都是 200ms，并且两个方波是倒相的，即相差半个周期。以下是简单的演示代码：

```
#001 #include <windows.h>
#002 #include <stdio.h>
#003 //使用 CTL_CODE 必须加入 winioctl.h
#004 #include <winioctl.h>
#005 #include "..\NT_Driver\Ioctl.h"
#006 int main()
#007 {
#008     //打开设备句柄
#009     HANDLE hDevice =
#010         CreateFile("\\\\.\\HelloBDK",
#011             GENERIC_READ | GENERIC_WRITE,
#012             0,
#013             NULL,
#014             OPEN_EXISTING,
#015             FILE_ATTRIBUTE_NORMAL,
#016             NULL );
#017     //判断是否成功打开设备句柄
#018     if (hDevice == INVALID_HANDLE_VALUE)
#019     {
#020         printf("Failed to obtain file handle to device: "
#021             "%s with Win32 error code: %d\n",
#022             "MyWDMDevice", GetLastError() );
#023         return 1;
#024     }
#025
#026     DWORD dwOutput ;
#027     DWORD inputBuffer[3] =
#028     {
#029         0x378,        //对 0x378 进行操作
#030         1,            //1 代表 8 位操作，2 代表 16 位操作，4 代表 32 位操作
#031         2,            //输出字节
#032     }
#033
#034     //产生周期是 200ms 的周期信号
```


Windows 驱动开发技术详解

```
#035     for (;;)
#036     {
#037         DeviceIoControl(hDevice, WRITE_PORT, inputBuffer, sizeof(inputBuffer),
NULL, 0, &dwOutput, NULL);
#038         inputBuffer[1] = inputBuffer[1] ^3;    //让第 0、1 位求反
#039         Sleep(100);                            //暂停 100ms
#040     }
#041     //关闭设备句柄
#042     CloseHandle(hDevice);
#043     return 0;
#044 }
```

此段代码可以在配套光盘中本章的 Test8 目录下找到。

15.9 小结

本章总结了多种 I/O 端口操作的方法。这些方法本质上是一样的，都是将端口输入输出的汇编指令运行在内核模式中。读者可以对比这些例子，能不同角度考虑端口操作。其中有些方法模拟了 Windows 内核函数的实现，读者可以通过这些例子更深入地理解 Windows 操作系统的内核。I/O 端口操作是驱动程序中最基本的操作，在 PCI 驱动程序中，这些技巧将被再次用到。

第 16 章 PCI 设备驱动

PCI 总线标准是一种将系统外部设备连接起来的总线标准，它是 PC 中最重要的总线。其他总线如 ISA 总线、USB 等总线都挂在 PCI 总线之上。本章将主要介绍 PCI 协议和 PCI 设备的驱动程序开发。开发 PCI 设备驱动需要获取 PCI 配置空间的各个数据，本章总结了四种方法获取 PCI 配置空间，最后，本章给出一个具体的 PCI 驱动程序的实例。

16.1 PCI 总线协议

PCI 总线协议是 PC 上最基本的总线，它的传输速度很高，可以达到 133MB/s，一般显卡、网卡都被设计成 PCI 总线设备。其他总线都是挂在 PCI 总线上的，例如，ISA 总线是通过 PCI-ISA 桥设备挂在 PCI 总线上，而 USB 总线是通过 USB HOST 设备挂在 PCI 总线上的。

16.1.1 PCI 总线简介

PCI (Peripheral Component Interconnect) 总线是当前最流行的总线之一，是由 Intel 公司首先推出的一种局部总线。它定义了 32 位数据地址总线，并且可扩展为 64 位，其支持突发读写操作，也同时可以支持多组外围设备。

PCI 局部总线不能兼容在其之前出现的 ISA、EISA、MCA (Micro Channel Architecture) 等总线。但是大多数基于上述总线的设备可以通过自行设计 PCI 桥接电路挂载在 PCI 局部总线之上。实际上在当前的 PC 体系结构内，几乎所有外部设备采用的各种各样的接口总线，均是通过桥接电路挂载在 PCI 系统内的。在这种 PCI 系统中，Host/PCI 桥称为北桥，连接主处理器总线到基础 PCI 局部总线。PCI-ISA 桥称为南桥，连接基础 PCI 总线到 ISA 总线。其中南桥通常还含有中断控制器、IDE 控制器、USB 控制器和 DMA 控制器等设备。

南桥和北桥组成主板的芯片组。通过芯片组的扩展，实现了多种总线与基础 PCI 局部

总线的桥接，从而实现了多种总线对整个 PC 系统的挂接。在基础 PCI 局部总线或 PCI 插入卡上，可以嵌入一个或多个 PCI-PCI 桥，从而实现在基础 PCI 局部总线上挂接多个 PCI 设备。如图 16-1 所示为 PCI 总线、扩展总线、处理器和存储器总线间的基本关系。

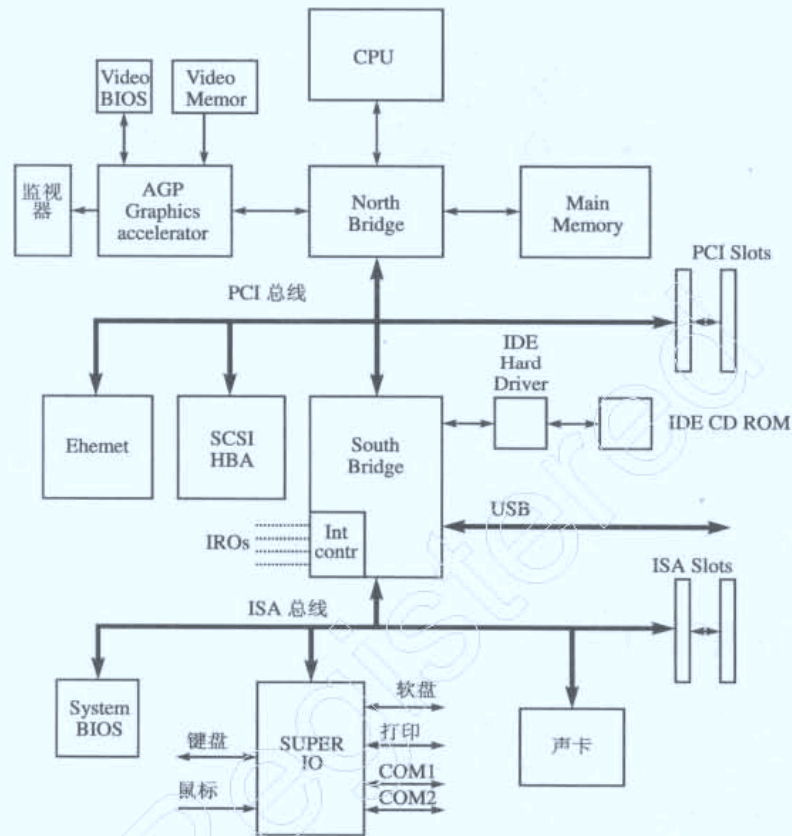


图 16-1 PCI 总线

16.1.2 PCI 配置空间简介

PCI 有三个相互独立的物理地址空间：设备存储器地址空间、I/O 地址空间和配置空间。配置空间是 PCI 所特有的一个物理空间。由于 PCI 支持设备即插即用，所以 PCI 设备不占用固定的内存地址空间或 I/O 地址空间，而是可以由操作系统决定其映射的基址。

系统加电时，BIOS 检测 PCI 总线，确定所有连接在 PCI 总线上的设备以及它们的配置要求，并进行系统配置。所以，所有的 PCI 设备必须实现配置空间，从而能够实现参数自动配置，实现真正的即插即用。

PCI 总线规范定义的配置空间总长度为 256 个字节，配置信息按一定的顺序和大小依次存放。前 64 个字节的配置空间称为配置头，对于所有的设备都一样，配置头的功能主要是用来识别设备，定义主机访问 PCI 卡的方式（I/O 访问或者存储器访问，还有中断信息）。其余的 192 个字节空间称为本地配置空间，主要定义卡上局部总线的特性、本地空

间基地址及范围等。

PCI2.2 规范定义了三种配置头格式：类型 0、类型 1 和类型 2。其中类型 1 适用于 PCI-to-PCI 桥设备，用于将两条 PCI 总线进行连接；类型 2 适用于 PCI-CardBus（主要用于笔记本的插卡式总线）桥，在 PC Card 规范中进行定义；类型 0 用于除类型 1 和类型 2 以外所有的 PCI 设备。一般设备都采用类型 0 的配置头。

如图 16-2 所示，描绘了类型 0 的 256 字节的配置空间。

31		16 15		0
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Rev ID	08h
BIST	Header Type	Latency Timer	Cache Line Size	0Ch
Base Address Register 0 (BAR0)				10h
Base Address Register 1 (BAR1)				14h
Base Address Register 2 (BAR2)				18h
Base Address Register 3 (BAR3)				1Ch
Base Address Register 4 (BAR5)				20h
Base Address Register 5 (BAR5)				24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address				30h
Reserved			CapPtr	34h
Reserved				38h
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line	3Ch
Reserved				40h-FFh

图 16-2 配置空间

PCI 大大地提高了系统的易配置性，这些是通过由 PCI 设备提供相应的功能来实现的。虽然寄存器的具体位的设置因设备而异，但是所有的寄存器必须能够被读写。这些寄存器的操作要能反映以下的信息：

(1) Vendor ID：厂商 ID。知名的设备厂商的 ID。FFFFh 是一个非法厂商 ID，它判断 PCI 设备是否存在。

(2) Device ID：设备 ID。某厂商生产的设备的 ID。操作系统就是凭着 Vendor ID 和 Device ID 找到对应驱动程序的。

(3) Class Code：类代码。共三个字节，分别是类代码、子类代码及编程接口。类代码不仅用于区分设备类型，还是编程接口的规范，这就是为什么会有通用驱动程序的原因。

(4) IRQ Line：IRQ 编号。PC 机以前是靠两片 8259 芯片来管理 16 个硬件中断。现在为了支持对称多处理器，有了 APIC（高级可编程中断控制器），它支持管理 24 个中断。

(5) IRQ Pin：中断引脚。PCI 有 4 个中断引脚，该寄存器表明该设备连接的是哪个引脚。

(6) 设备控制：这是由 Command 寄存器来实现的，提供了控制设备对于 PCI 访问的响应以及执行的能力，Command 寄存器的具体各位的信息可以通过 PCI 规范查得。

(7) 设备状态：Status 寄存器用来记录一个 PCI 设备当前的状态，Status 寄存器的具体各位的信息也可以通过 PCI 规范查到。

(8) 基地址：基地址寄存器用来存放 PCI 设备映射的存储器地址或者使用的 IO 空间的首地址。PCI 规范提供一种机制使得 I/O 和存储器分开，即在基地址寄存器的最低位上，如果是 0，表示该基地址寄存器指向的是一个存储器空间，而如果是 1，就是指向一个 I/O 空间。

16.2 访问 PCI 配置空间方法一

在驱动程序中，访问 PCI 的配置空间有多种办法，有的是从最基本的 I/O 端口读出整个配置空间，然后通过分析，从配置空间得出驱动的有用信息。除此之外，DDK 也提供了读取 I/O 资源的函数，这些函数将上述步骤进行封装，简化了程序员的工作。为了比较，笔者将这几种方法都呈现给读者，本节的方法是用最原始的方法，通过 I/O 端口直接读取 PCI 配置空间。

16.2.1 两个重要寄存器

访问 PCI 配置空间可通过访问两个寄存器，CONFIG_ADDRESS 寄存器和 CONFIG_DATA 寄存器。这两个寄存器在 PC 中分别对应着 CF8h 和 CFCh 端口，并且是 32 位端口，即读写要用 32 位的 IN 和 OUT 汇编指令。其中 CONFIG_ADDRESS 寄存器的内容如图 16-3 所示。

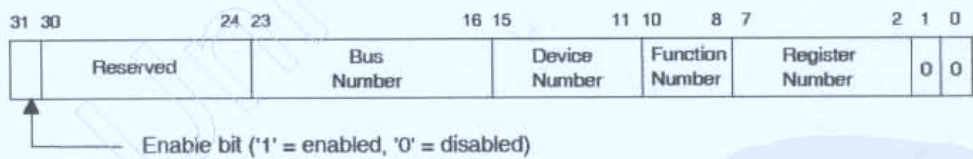


图 16-3 CONFIG_ADDRESS 寄存器

每个 PCI 设备可应用三个信息进行定位，即 Bus Number、Device Number 及 Function Number。另外，PCI 配置空间一共是 256 个字节，被分割成 64 个 4 字节的寄存器，从 0~63 编号。

每次要访问 PCI 配置空间时，先设置 CONFIG_ADDRESS 寄存器，这时 CONFIG_DATA 寄存器中的内容就对应着该 PCI 配置空间中的相应寄存器。

例如，要访问 Bus Number=0，Device Number=1，Function Number=2 的 PCI 配置空间中的 Status 和 Command。

首先查看 Status 和 Command 在 PCI 配置空间位于第 1 号寄存器（最开始的寄存器是 0 号），然后将 Bus Number, Device Number, Function Number 按照图 16-3 中的格式填好，即 0x80000A04（二进制是 1000 0000 0000 0000 0000 1010 0000 0100）。然后设置 CONFIG_ADDRESS 寄存器（0xCF8 端口）为 0x80000A04，然后查看 CONFIG_DATA 寄存器（0xCFC 端口），得出的内容即 Status 和 Command（Status 在高 16 位，Command 在低 16 位）。

16.2.2 示例

本例主要是根据 Bus Number、Device Number 及 Function Number 三个信息读取该 PCI 设备的 PCI 配置空间。本例中的端口操作是应用程序用 DeviceIOControl 函数向驱动程序申请实现的。

```
#001 //显示 PCI 配置空间
#002 void DisplayPCIConfiguration(HANDLE hDevice,int bus,int dev,int func)
#003 {
#004     DWORD    dwAddr;
#005     DWORD    dwData;
#006
#007     PCI_COMMON_CONFIG pci_config;
#008     PCI_SLOT_NUMBER SlotNumber;
#009
#010     SlotNumber.u.AsULONG = 0;
#011     //设置设备号
#012     SlotNumber.u.bits.DeviceNumber = dev;
#013     //设置功能号
#014     SlotNumber.u.bits.FunctionNumber = func;
#015     //得到物理地址
#016     dwAddr = 0x8000C000 | (bus <<16) | (SlotNumber.u.AsULONG<<8);
#017
#018     /* 256 字节的 PCI 配置空间 */
#019     for (int i = 0; i < 0x100; i += 4)
#020     {
#021         //进行 32 位 OUT 操作
#022         Out_32(hDevice,PCI_CONFIG_ADDRESS, dwAddr | i);
#023         dwData = In_32(hDevice,PCI_CONFIG_DATA);
#024         memcpy( ((PUCHAR)&pci_config)+i,&dwData,4);
#025     }
#026     //打印一些调试信息
#027     printf("bus:%d\tdev:%d\tfunc:%d\n",bus,dev,func);
#028
#029     printf("VendorID:%x\n",pci_config.VendorID);
#030     printf("DeviceID:%x\n",pci_config.DeviceID);
#031     printf("Command:%x\n",pci_config.Command);
#032     printf("Status:%x\n",pci_config.Status);
#033     printf("RevisionID:%x\n",pci_config.RevisionID);
#034     printf("ProgIf:%x\n",pci_config.ProgIf);
#035     printf("SubClass:%x\n",pci_config.SubClass);
#036     printf("BaseClass:%x\n",pci_config.BaseClass);
#037     printf("CacheLineSize:%x\n",pci_config.CacheLineSize);
#038     printf("LatencyTimer:%x\n",pci_config.LatencyTimer);
```


Windows 驱动开发技术详解

```
#039     printf("HeaderType:%x\n",pci_config.HeaderType);
#040     printf("BIST:%x\n",pci_config.BIST);
#041     for (i=0;i<6;i++)
#042     {
#043         printf("BaseAddresses[%d]:0X%08X\n",i,pci_config.u.type0.
BaseAddresses[i]);
#044     }
#045     printf("InterruptLine:%d\n",pci_config.u.type0.InterruptLine);
#046     printf("InterruptPin:%d\n",pci_config.u.type0.InterruptPin);
#047 }
```

此段代码可以在配套光盘中本章的 Test1 目录下找到。

本例在应用程序中引用了 DDK 中两个数据结构，分别是 PCI_COMMON_CONFIG 和 PCI_SLOT_NUMBER 数据结构。

笔者计算机的南桥芯片组属于 ICH7 系列，读者可以用软件 everest 查看自己计算机芯片组的型号，如图 16-4 所示。

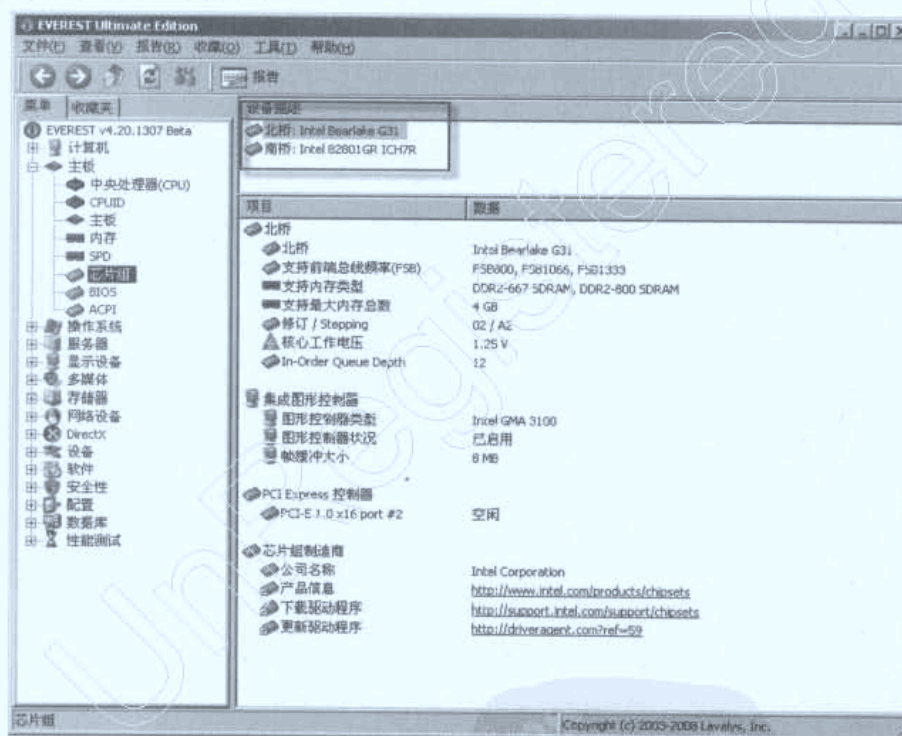


图 16-4 everest 查看设备

ICH7 芯片组的 DataSheet 参见 Intel 网站上的“Intel® I/O Controller Hub 7 (ICH7) Family” 30701303.pdf，该文档说明了南桥自带的 PCI 设备，如图 16-5 所示。

如果读者想查询南桥自带 PCI 设备的配置空间，如 USB UHCI Controller #1，可以使用以下的代码：

```
DisplayPCIConfiguration(hDevice,0,29,0);
```

该例子会显示出所有 256 字节的配置空间。可以看出，Intel 公司的 VendorID 都是 8086，

主板自带的 PCI 设备的 Bus 号都为 0。

Bus:Device:Function	Function Description
Bus 0:Device 30:Function 0	PCI-to-PCI Bridge
Bus 0:Device 30:Function 2	AC '97 Audio Controller (Desktop and Mobile Only)
Bus 0:Device 30:Function 3	AC '97 Modem Controller (Desktop and Mobile Only)
Bus 0:Device 31:Function 0	LPC Controller ¹
Bus 0:Device 31:Function 1	IDE Controller
Bus 0:Device 31:Function 2	SATA Controller (Desktop and Mobile Only)
Bus 0:Device 31:Function 3	SMBus Controller
Bus 0:Device 29:Function 0	USB UHCI Controller #1
Bus 0:Device 29:Function 1	USB UHCI Controller #2
Bus 0:Device 29:Function 2	USB UHCI Controller #3
Bus 0:Device 29:Function 3	USB UHCI Controller #4
Bus 0:Device 29:Function 7	USB 2.0 EHCI Controller
Bus 0:Device 28:Function 0	PCI Express [®] Port 1 (Desktop and Mobile Only)
Bus 0:Device 28:Function 1	PCI Express Port 2 (Desktop and Mobile Only)
Bus 0:Device 28:Function 2	PCI Express Port 3 (Desktop and Mobile Only)
Bus 0:Device 28:Function 3	PCI Express Port 4 (Desktop and Mobile Only)
Bus 0:Device 28:Function 4	PCI Express Port 5 (Intel [®] ICH7R, ICH7DH, and ICH7-M DH Only)
Bus 0:Device 28:Function 5	PCI Express Port 6 (Intel ICH7R, ICH7DH, and ICH7-M DH Only)
Bus 0:Device 27:Function 0	Intel [®] High Definition Audio Controller
Bus n:Device 8:Function 0	LAN Controller (Desktop and Mobile Only)

图 16-5 ICH7 芯片组南桥芯片

16.3 访问 PCI 配置空间方法二

第二种访问 PCI 配置空间的方法是通过 HalGetBusData 和 HalSetBusData 两个内核函数。这两个函数将方法一进行了封装，不需要程序员对 PCI 空间进行直接读取。

16.3.1 DDK 函数读取配置空间

DDK 提供了两个内核函数 HalGetBusData 和 HalSetBusData，分别用于读取 PCI 设备的配置空间和设置 PCI 配置空间。

这两个内核函数最早出现在 Windows2000 DDK 中，但是在 WindowsXP DDK 和 Windows2003 DDK 中，这两个函数已经被当做“过时”的内核函数。但为了考虑兼容性被保留下来。笔者认为这两个函数非常好用，可以很轻松地读取和设置 PCI 的配置空间。以 HalGetBusData 内核函数为例：

```
ULONG
HalGetBusData(
    IN BUS_DATA_TYPE BusDataType,
    IN ULONG BusNumber,
    IN ULONG SlotNumber,
    IN PVOID Buffer,
    IN ULONG Length
);
```


- BusDataType: 该参数指定总线类型。HalGetBusData 函数可以查询各个总线的情况, 对于 PCI 总线, 这里设置为 PCIConfiguration。
- BusNumber: 该参数指定 Bus 总线号。
- SlotNumber: 该参数由 Device 号和功能号共同组成。
- Buffer: 该参数是得到的 PCI 配置空间的首地址。
- Length: 该参数是 PCI 配置空间的大小。

用 HalGetBusData 和 HalSetBusData 两个函数访问 PCI 配置空间, 其内部原理和 15.3 中介绍的第一种方法基本一样。DDK 提供了这两个函数, 就不用程序员辛苦地去实现了, 直接利用这两个函数就可以了。HalGetBusData 和 HalSetBusData 两个函数内部也是通过操作端口实现的。

16.3.2 示例

下面的例子用第二种访问 PCI 配置空间的方法, 演示了枚举系统内的所有 PCI 设备。PCI 设备主要靠三个信息定位, 即 Bus 号、Device 号和功能号。只要分别对这三个数进行排列组合, 就能枚举出系统内的所有设备。以下是在内核中枚举所有 PCI 设备的主要代码, 枚举的同时会罗列出 PCI 配置空间的相关信息。

```
#001 #define PDI_BUS_MAX    0xFF
#002 #define PDI_DEVICE_MAX  0x1F
#003 #define PDI_FUNCTION_MAX  0x7
#004 #pragma PAGEDCODE
#005 VOID EnumeratePCI()
#006 {
#007     ULONG bus; //总线号
#008     ULONG dev; //设备号
#009     ULONG func; //功能号
#010     PCI_COMMON_CONFIG PciConfig;
#011     PCI_SLOT_NUMBER SlotNumber;
#012
#013     KdPrint(("Bus\tDevice\tFunc\tVendor\tDevice\tBaseCls\tSubCls\tIRQ\tPIN\n"));
#014     //枚举总线号
#015     for (bus = 0; bus <= PDI_BUS_MAX; ++bus)
#016     {
#017         //枚举设备号
#018         for (dev = 0; dev <= PDI_DEVICE_MAX; ++dev)
#019         {
#020             //枚举功能号
#021             for (func = 0; func <= PDI_FUNCTION_MAX; ++func)
#022             {
#023                 SlotNumber.u.AsULONG = 0;
#024                 //设置设备号
#025                 SlotNumber.u.bits.DeviceNumber = dev;
#026                 //设置功能号
#027                 SlotNumber.u.bits.FunctionNumber = func;
#028                 //内存拷贝
#029                 RtlZeroMemory(&PciConfig,
```

```

#030                                     sizeof(PCI_COMMON_CONFIG));
#031                                     //获得 PCI 配置空间
#032                                     ULONG Size = HalGetBusData(PCIConfiguration,
#033                                     bus,
#034                                     SlotNumber.u.AsULONG,
#035                                     &PciConfig,
#036                                     PCI_COMMON_HDR_LENGTH);
#037                                     if (Size==PCI_COMMON_HDR_LENGTH)
#038                                     {
#039                                     //打印调试信息
#040                                     KdPrint(("02X\t02X\t%x\t%x\t%x\t02X\t02X\t%d\t%d\n",
#041                                     bus,
#042                                     dev,
#043                                     func,
#044                                     PciConfig.VendorID,
#045                                     PciConfig.DeviceID,
#046                                     PciConfig.BaseClass,
#047                                     PciConfig.SubClass,
#048                                     PciConfig.u.type0.InterruptLine,
#049                                     PciConfig.u.type0.InterruptPin));
#050                                     }
#051                                     }
#052                                     }
#053                                     }
#054                                     }

```

此段代码可以在配套光盘中本章的 Test2 目录下找到。

如图 16-6 所示为通过以上程序，枚举出笔者计算机里的所有 PCI 设备。

Time	Channel	Computer	Message text	Bus	Device	Func	Vendor	Device	BaseCls	SubCls	IRQ	PIN
1063.546	Default		Enter HelloDDKDeviceIOControl									
1063.546	Default		Bus	00	00	0	8086	2770	06	00	0	0
1063.546	Default		Device	00	01	0	8086	2771	06	04	16	1
1063.546	Default		Func	00	1F	0	8086	27A8	04	03	16	1
1063.546	Default		Vendor	03	1D	0	8086	27C8	0C	03	23	1
1063.546	Default		Device	00	1D	1	8086	27C9	0C	03	19	2
1063.546	Default		Func	00	1D	2	8086	27CA	0C	03	18	3
1063.546	Default		Vendor	00	1D	3	8086	27CB	0C	03	16	4
1063.546	Default		Device	00	1D	7	8086	27CC	0C	03	23	1
1063.546	Default		Func	00	1E	0	8086	244E	06	04	255	0
1063.546	Default		Vendor	00	1F	0	8086	27B6	06	01	0	0
1063.546	Default		Device	00	1F	1	8086	27B7	01	01	0	1
1063.546	Default		Func	00	1F	2	8086	27B8	01	01	19	2
1063.546	Default		Vendor	00	1F	3	8086	27B9	0C	05	11	2
1063.546	Default		Device	01	00	0	104E	393	03	00	16	1
1063.546	Default		Func	02	01	0	104E	8139	02	00	17	1
1063.546	Default		Vendor	02	05	0	104E	8139	02	00	22	1
1063.582	Default		Leave HelloDDKDeviceIOControl									
1063.582	Default		Enter HelloDDKDispatchRoutine									

图 16-6 枚举 PCI 设备

16.4 访问 PCI 配置空间方法三

方法二适用于 NT 式驱动，但并不适用 WDM 驱动，因为在 WDM 驱动中上述的 HalGetBusData 和 HalSetBusData 两个函数虽然可以继续使用，但已经不被微软所推荐。DDK 保留了这两个函数，只是为了兼容以前的驱动。在 WDM 驱动程序中，应该使用第

三种方法，通过即插即用 IRP 获得 PCI 配置空间。

16.4.1 通过即插即用 IRP 获得 PCI 配置空间

在前面介绍的两个方法中，第一种是用最底层的方法获取 PCI 配置空间。程序员需要知道如何读取配置空间的 I/O 端口，并正确地操作它们。第二种方法是用 DDK 的提供的内核函数实现。这两种办法获取某个设备的 PCI 配置空间必须有三个输入信息，即总线号、设备号及功能号。

然而作为一个设备的驱动程序，以上这三个信息是应该在插入设备的时候就已经确定好的。WDM 模型为程序员简化了获取 PCI 配置空间的方法，程序员甚至不需要了解具体 PCI 配置空间的结构，这是因为 WDM 驱动程序已经帮助程序员从 PCI 配置空间中分析出有用的信息，并通知给程序员。

这种方法只能用在 WDM 驱动程序中，而不能用在 NT 驱动程序中。WDM 会为不同总线上的设备提供一个 PDO 设备，当程序员所写的功能驱动挂载在 PDO 之上的时候，就可以将 IRP_MN_START_DEVICE 传递给底层的 PDO 去处理。PCI 总线的 PDO 就会得到 PCI 配置空间，并从中得到有用信息，如中断号、设备物理内存及 IO 端口等信息。

这种方法是微软推荐的方法。因此在 WDM 驱动中，用 HalGetBusData 和 HalSetBusData 函数直接读取 PCI 配置空间的方法都是不被推荐的。微软鼓励程序员使用 PDO 处理 IRP_MN_START_DEVICE 的结果。

在处理完 IRP_MN_START_DEVICE 后，驱动程序会将处理结果存储在 IRP 的设备堆栈中，从 I/O 堆栈可以取出 CM_FULL_RESOURCE_DESCRIPTOR 数据结构，从 CM_FULL_RESOURCE_DESCRIPTOR 中可以取出 CM_PARTIAL_RESOURCE_LIST 数据结构，而在 CM_PARTIAL_RESOURCE_LIST 中又可以取出 CM_PARTIAL_RESOURCE_DESCRIPTOR 数据结构。

CM_PARTIAL_RESOURCE_DESCRIPTOR 数据结构就是 PDO 帮助程序员从 256 字节的 PCI 配置空间中获取的有用信息。这其中包括中断号、设备物理内存、I/O 端口等信息。

16.4.2 示例

由于这个例子必须是针对具体 PCI 设备，因此笔者将用 PCI 网卡做示范。笔者将自己 PC 中的一个网卡加载成修改后的 HelloWDM 驱动。读者可以根据需要，将 HelloWDM 加载在其他 PCI 设备上。

以下是获取 PCI 配置空间的相关示例代码：

```
#001 NTSTATUS HandleStartDevice(PDEVICE_EXTENSION pdx, PIRP Irp)
#002 {
#003     PAGED_CODE();
#004     KdPrint(("Enter HandleStartDevice\n"));
#005 }
```

```

#006    //转发 IRP 并等待返回
#007    NTSTATUS status = ForwardAndWait(pdx, Irp);
#008    //判断 IRP 是否在底层驱动中成功处理
#009    if (!NT_SUCCESS(status))
#010    {
#011        Irp->IoStatus.Status = status;
#012        IoCompleteRequest(Irp, IO_NO_INCREMENT);
#013        return status;
#014    }
#015    //得到当前堆栈
#016    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
#017
#018    //从当前堆栈得到源信息
#019    PCM_PARTIAL_RESOURCE_LIST raw;
#020    if (stack->Parameters.StartDevice.AllocatedResources)
#021        raw = &stack->Parameters.StartDevice.AllocatedResources->List[0].
PartialResourceList;
#022    else
#023        raw = NULL;
#024
#025    KdPrint(("Show raw resources\n"));
#026    //显示 PCI 卡的相关资源
#027    ShowResources(raw);
#028
#029    //从当前堆栈得到翻译信息
#030    PCM_PARTIAL_RESOURCE_LIST translated;
#031    if (stack->Parameters.StartDevice.AllocatedResourcesTranslated)
#032        translated =
#033        &stack->Parameters.StartDevice.AllocatedResourcesTranslated->
List[0].PartialResourceList;
#034    else
#035        translated = NULL;
#036
#037    KdPrint(("Show translated resources\n"));
#038    //显示 PCI 资源
#039    ShowResources(translated);
#040
#041    //设置 IRP 的完成状态
#042    Irp->IoStatus.Status = STATUS_SUCCESS;
#043    //结束 IRP 请求
#044    IoCompleteRequest(Irp, IO_NO_INCREMENT);
#045
#046    KdPrint(("Leave HandleStartDevice\n"));
#047    return status;
#048 }

```

此段代码可以在配套光盘中本章的 Test3 目录下找到。

16.5 访问 PCI 配置空间方法四

方法三介绍的方法适用于在 WDM 驱动程序中, 获得驱动程序本身对应的 PCI 设备的 PCI 配置空间。本节介绍的方法, 是获取其他 PCI 设备的配置空间方法。

16.5.1 创建 IRP_MN_READ_CONFIG

在方法三中，设备在启动前，即插即用管理器会向 PCI 驱动发送多个即插即用 IRP，其中包括 IRP_MN_START_DEVICE。方法三将即插即用管理器发过来的 IRP_MN_START_DEVICE 传递给 PDO，当返回的时候获取 PDO 从 PCI 配置空间分析出来的有用信息。

此种方法并没有完整地获取到 256 字节的 PCI 配置空间，而是 PDO 通过该 256 字节分析出的信息。如果程序员想完整地获得 PCI 配置空间，需要自己创建 IRP_MN_READ_CONFIG 或者 IRP_MN_WRITE_CONFIG，然后将创建好的即插即用 IRP 发送到底层的 PDO，并等待 PDO 的处理。

方法四与方法三的主要区别是：方法四获取完整的 PCI 配置空间，自己创建即插即用 IRP。

16.5.2 示例

和方法三的例子一样，本例也必须是 WDM 驱动，因为只有 WDM 驱动才可以支持即插即用 IRP。因此，试验本例就必须将此驱动作为一个真实的 PCI 设备的驱动加载，而不能作为一个虚拟设备的驱动来加载。

笔者将自己 PC 中的一个网卡的驱动程序卸载，换上了修改后的 HelloWDM 驱动程序。读者可以根据需要，修改 HelloWDM 加载在其他 PCI 设备上，同样可以获取 PDO 传递回来的 PCI 配置空间的相关信息。

以下是获取 PCI 配置空间的相关示例代码：

```
#001 NTSTATUS ReadWriteConfigSpace(  
#002     IN PDEVICE_OBJECT DeviceObject,  
#003     IN ULONG           ReadOrWrite, // 0 for read 1 for write  
#004     IN PVOID           Buffer,  
#005     IN ULONG           Offset,  
#006     IN ULONG           Length  
#007 )  
#008 {  
#009     KEVENT event;  
#010     NTSTATUS status;  
#011     PIRP irp;  
#012     IO_STATUS_BLOCK ioStatusBlock;  
#013     PIO_STACK_LOCATION irpStack;  
#014     PDEVICE_OBJECT targetObject;  
#015     //确保运行在分页内存中  
#016     PAGED_CODE();  
#017     //初始化同步事件  
#018     KeInitializeEvent( &event, NotificationEvent, FALSE );  
#019     //将新设备附加在设备堆栈上  
#020     targetObject = IoGetAttachedDeviceReference( DeviceObject );  
#021     //创建同步类型的 IRP  
#022     irp = IoBuildSynchronousFsdRequest( IRP_MJ_PNP,
```

```

#023         targetObject,
#024         NULL,
#025         0,
#026         NULL,
#027         &event,
#028         &ioStatusBlock );
#029 //判断 IRP 是否为空
#030 if (irp == NULL) {
#031     status = STATUS_INSUFFICIENT_RESOURCES;
#032     goto End;
#033 }
#034 //得到下一个 IO 堆栈
#035 irpStack = IoGetNextIrpStackLocation( irp );
#036
#037 if (ReadOrWrite == 0) {
#038     //如果是读则设置 IRP_MN_READ_CONFIG
#039     irpStack->MinorFunction = IRP_MN_READ_CONFIG;
#040 }else {
#041     //如果是写则设置 IRP_MN_WRITE_CONFIG
#042     irpStack->MinorFunction = IRP_MN_WRITE_CONFIG;
#043 }
#044 //设置 PCI 配置空间
#045 irpStack->Parameters.ReadWriteConfig.WhichSpace = PCI_WHICHSPACE_
CONFIG;
#046 //设置缓冲区
#047 irpStack->Parameters.ReadWriteConfig.Buffer = Buffer;
#048 //设置偏移量
#049 irpStack->Parameters.ReadWriteConfig.Offset = Offset;
#050 //设置操作长度
#051 irpStack->Parameters.ReadWriteConfig.Length = Length;
#052
#053 irp->IoStatus.Status = STATUS_NOT_SUPPORTED ;
#054
#055 //调用底层驱动
#056 status = IoCallDriver( targetObject, irp );
#057 //判断 IRP 是否别挂起
#058 if (status == STATUS_PENDING)
#059 {
#060     //如果 IRP 被挂起, 等待 IRP 被结束
#061     KeWaitForSingleObject( &event, Executive, KernelMode, FALSE, NULL );
#062     status = ioStatusBlock.Status;
#063 }
#064 End:
#065 //减小引用计数
#066 ObDereferenceObject( targetObject );
#067 return status;
#068 }

```

这段代码可以在配套光盘中本章的 Test4 目录下找到。

16.6 PCI 设备驱动开发示例

本节讲解一个具体的 PCI 设备开发实例,它综合了本章前面讲解的所有 PCI 总线知识、

WDM 框架的开发 PCI 驱动的知识。

16.6.1 开发步骤

PCI 设备有各种功能，因此开发每种 PCI 设备的驱动程序都是不同的。但是 PCI 设备又有其共同点，主要按照以下几个步骤进行：

① 和硬件工程师沟通：和前几章的例子不同，PCI 设备驱动需要和真实的物理设备打交道，它不是虚拟设备。因此，了解物理设备的原理图，和硬件工程师的沟通是必需的。

② 在驱动程序中获取设备的资源：在前几节中，笔者带领读者了解了多种获取 PCI 配置空间的办法。其中第三种方法是大多数程序员使用的方法。这种方法对于程序员不需要知道太多的 PCI 配置空间的知识，PCI 的总线驱动（PDO）已经将设备的资源从 PCI 配置空间中提取出来。作为一个 WDM 驱动程序员，开发 PCI 设备驱动最重要的一步是将 IRP_MN_START_DEVICE 中获取的设备资源记录下来，以便以后使用。

③ 对中断的操作：PCI 设备一般会有中断，在设备启动时 IRP_MN_START_DEVICE 会将中断的一切信息报告给程序员。程序员需要通过 IoConnectInterrupt 函数将中断注册到系统中，使 Windows 内核能够接收到这个中断，并在中断来临时转到中断程序中。

④ 操作：首先要知道该设备的主要使用目的。例如，笔者做个一个视频采集卡的驱动程序。视频卡每次采集一幅图像，会通过中断线发出一个中断信号通知 PC。在简单设置 I/O 端口后，驱动程序会读取设备上的物理内存，这段物理内存就是设备得到的图像。因此，作为一个软件工程师需要经常与硬件工程师商量，把设计方案设计好。

16.6.2 中断操作

PCI 设备直接挂在 PC 的中断控制器上，并且每个中断源都有一个固定的中断号，也就是 IRQ 号。8259A 芯片可以接收 8 个中断，因为 PC 有两个 8259A 芯片，所以老式 PC 可以接 15 个中断信号（其中有一个被级联占用）。

由于 PC 设备的多样化，即使 15 个中断信号也不能满足要求。目前有两种改善方法，一个是增加中断数量，由原来的 15 个提升至 24 个。另外一个方法是用多个设备共享一个中断号。

但是共享一个中断号会出现一个问题：就是不知道此中断号是由哪个设备触发的。因此，当获得中断后，操作系统会询问这条中断线上的所有设备。这时通过查询每个设备上的中断寄存器，就可以知道是哪一个设备请求了中断。

另外多个设备共享一个中断号，还会引起丢失中断的情况。老式的板卡多为 ISA 板卡，一般是边缘触发，假如同时有两个设备公用一个中断号，且都是边缘触发，很有可能正在处理一个中断的同时丢失另一个中断。目前，PCI 板卡的中断都是基于电平触发，这样可以解决同一中断号的不同设备之间丢失中断的情况。

在操作系统收到 IRP_MN_START_DEVICE 时, PDO 会获取设备的中断号。此时会获得两组中断号, 一组是原始中断号, 另一组是经过翻译的中断号。驱动程序应该使用经过翻译的中断号。获取中断号后, 驱动程序可以使用 IoConnectInterrupt 将中断与中断处理函数挂接。

```
NTSTATUS
IoConnectInterrupt(
    OUT PKINTERRUPT *InterruptObject,
    IN PKSERVICE_ROUTINE ServiceRoutine,
    IN PVOID ServiceContext,
    IN PKSPIN_LOCK SpinLock OPTIONAL,
    IN ULONG Vector,
    IN KIRQL Irql,
    IN KIRQL SynchronizeIrql,
    IN KINTERRUPT_MODE InterruptMode,
    IN BOOLEAN ShareVector,
    IN KAFFINITY ProcessorEnableMask,
    IN BOOLEAN FloatingSave
);
```

- InterruptObject: 输出参数, 得到一个 INTERRUPT 结构的中断内核对象。当设备停止或者退出时, 需要将中断与中断号分离, 这时候需要调用 IoDisconnectInterrupt, 而 InterruptObject 需要作为其参数。因此 InterruptObject 一般都需要记录在设备扩展中。
- ServiceRoutine: 驱动程序提供的中断例程, 此服务例程运行在 DIRQL 上, 所有分页的内存操作都不能在这个函数上使用。
- ServiceContext: 中断服务例程的上下文参数。
- SpinLock: 用于同步的自旋锁, 可选参数, 一般设为 NULL。
- Vector: 中断向量号, 这个是 PDO 返回时可以得到的。
- Irql: IRQL, 同样是 PDO 返回时可以得到的。
- SynchronizeIrql: 同 IRQL。
- InterruptMode: 中断的模式, 主要是电平触发 (Latched) 和边缘触发 (LevelSensitive)。对于 PCI 设备, 一般都是电平触发。
- ShareVector: 指定是否允许与其他 PCI 设备共享中断。
- ProcessorEnableMask: CPU 屏蔽位。

16.6.3 操作设备物理内存

对于物理内存, 一般程序是无法轻易读取的。一般程序所看到的内存指针都是虚拟内存, 如果想操作物理内存, 必须使用 DDK 提供的内核函数 WRITE_REGISTER_XX 系列函数和 READ_REGISTER_XX 系列函数, 如表 16-1 所示。

表 16-1 DDK 中有关物理内存操作函数

函 数	描 述
READ_REGISTER_UCHAR	8 位输入
READ_REGISTER_USHORT	16 位输入
READ_REGISTER_ULONG	32 位输入
READ_REGISTER_BUFFER_UCHAR	8 位输入 (连续多个 8 位)
READ_REGISTER_BUFFER_USHORT	16 位输入 (连续多个 16 位)
READ_REGISTER_BUFFER_ULONG	32 位输入 (连续多个 32 位)
WRITE_REGISTER_UCHAR	8 位输出
WRITE_REGISTER_USHORT	16 位输出
WRITE_REGISTER_ULONG	32 位输出
WRITE_REGISTER_BUFFER_UCHAR	8 位输出 (连续多个 8 位)
WRITE_REGISTER_BUFFER_USHORT	16 位输出 (连续多个 16 位)
WRITE_REGISTER_BUFFER_ULONG	32 位输出 (连续多个 32 位)

另外 DDK 提供了内核函数 `MmAllocateContiguousMemory`，它可以分配连续的物理内存，并映射成连续的虚拟内存。`MmAllocateContiguousMemory` 返回的是虚拟内存地址，可以用 `MmGetPhysicalAddress` 得到连续的物理内存地址，然后用前面介绍的 `WRITE_REGISTER_XX` 系列函数和 `READ_REGISTER_XX` 系列函数读写。

16.6.4 示例

最后，笔者给出一段操作 PCI 驱动的示例代码。由于各个 PCI 设备千差万别，作用也各有不同，因此很难写出一个通用的 PCI 驱动。笔者写出了—个比较通用的 PCI 驱动的程序，它能简单地枚举出 PCI 的各个资源并使用这些资源。读者可以根据自己的 PCI 设备，来开发自己的 PCI 设备驱动。以下是示例代码：

```
#001 NTSTATUS initMyPCI(IN PDEVICE_EXTENSION pdx, IN PCM_PARTIAL_RESOURCE_LIST list)
#002 {
#003     PDEVICE_OBJECT fdo = pdx->fdo;
#004
#005     ULONG vector;           //中断向量
#006     KIRQL irql;            //中断请求级
#007     KINTERRUPT_MODE mode;   //中断模式
#008     KAFFINITY affinity;     //CPU 的亲缘关系
#009     BOOLEAN irqshare;       //是否共享中断
#010     BOOLEAN gotinterrupt = FALSE;
#011
#012     PHYSICAL_ADDRESS portbase; //物理地址
#013     BOOLEAN gotport = FALSE;
#014
#015     PCM_PARTIAL_RESOURCE_DESCRIPTOR resource = &list->PartialDescriptors[0];
#016     ULONG nres = list->Count;
#017     BOOLEAN IsMem0 = TRUE;
#018     // 每次循环获取一种资源
```

```

#019     for (ULONG i = 0; i < nres; ++i, ++resource)
#020     {
#021         //判断是何种资源
#022         switch (resource->Type)
#023         {
#024             //I/O 端口资源
#025             case CmResourceTypePort:
#026                 //I/O 端口地址
#027                 portbase = resource->u.Port.Start;
#028                 //I/O 端口地址长度
#029                 pdx->nports = resource->u.Port.Length;
#030                 //是否需要地址映射
#031                 pdx->mappedport = (resource->Flags & CM_RESOURCE_PORT_IO) == 0;
#032                 //表示已经得到 I/O 端口资源
#033                 gotport = TRUE;
#034                 break;
#035             //物理内存资源
#036             case CmResourceTypeMemory:
#037                 if (IsMem0)
#038                 {
#039                     //获取基地址 0
#040                     pdx->MemBar0 = (PUCHAR)MmMapIoSpace(resource->u.Memory.Start,
#041                                                         resource->u.Memory.Length,
#042                                                         MmNonCached);
#043                     pdx->nMem0 = resource->u.Memory.Length;
#044                     IsMem0 = FALSE;
#045                 }else
#046                 {
#047                     //获取基地址 1
#048                     pdx->MemBar1 = (PUCHAR)MmMapIoSpace(resource->u.Memory.Start,
#049                                                         resource->u.Memory.Length,
#050                                                         MmNonCached);
#051                     pdx->nMem1 = resource->u.Memory.Length;
#052                 }
#053                 break;
#054             //中断资源
#055             case CmResourceTypeInterrupt:
#056                 //获得中断请求级
#057                 irq1 = (KIRQL) resource->u.Interrupt.Level;
#058                 //获得中断向量
#059                 vector = resource->u.Interrupt.Vector;
#060                 //获取 CPU 亲缘关系
#061                 affinity = resource->u.Interrupt.Affinity;
#062                 //获得中断模式
#063                 mode = (resource->Flags == CM_RESOURCE_INTERRUPT_LATCHED)
#064                       ? Latched : LevelSensitive;
#065                 //判断是否需要共享中断
#066                 irqshare = resource->ShareDisposition == CmResourceShareShared;
#067                 //表示已经得到中断
#068                 gotinterrupt = TRUE;
#069                 break;
#070             default:
#071                 KdPrint(("Unexpected I/O resource type %d\n", resource->Type));
#072                 break;
#073         }
#074     }

```



```

#075     if (!(TRUE&& gotport&& gotinterrupt    ))
#076     {
#077         KdPrint((" Didn't get expected I/O resources\n"));
#078         return STATUS_DEVICE_CONFIGURATION_ERROR;
#079     }
#080     //判断是否需要I/O端口映射
#081     if (pdx->mappedport)
#082     {
#083         //获得I/O端口地址
#084         pdx->portbase = (PUCHAR) MmMapIoSpace(portbase, pdx->nports, MmNonCached);
#085         if (!pdx->mappedport)
#086         {
#087             KdPrint(("Unable to map port range %I64X, length %X\n", portbase,
pdx->nports));
#088             return STATUS_INSUFFICIENT_RESOURCES;
#089         }
#090     }
#091     else
#092         //获得I/O端口地址
#093         pdx->portbase = (PUCHAR) portbase.QuadPart;
#094
#095     //连接中断
#096     NTSTATUS status = IoConnectInterrupt(&pdx->InterruptObject, (PKSERVICE_
ROUTINE) OnInterrupt,
#097         (PVOID) pdx, NULL, vector, irq1, irq1, LevelSensitive, TRUE, affinity,
FALSE);
#098     if (INT_SUCCESS(status))
#099     {
#100         KdPrint(("IoConnectInterrupt failed - %X\n", status));
#101         if (pdx->portbase && pdx->mappedport)
#102             MmUnmapIoSpace(pdx->portbase, pdx->nports);
#103         pdx->portbase = NULL;
#104         return status;
#105     }
#106
#107     #define IMAGE_LENGTH (640*480)
#108     //申请一段连续物理地址来读取图像
#109     PHYSICAL_ADDRESS maxAddress;
#110     //设置物理地址的低32位
#111     maxAddress.u.LowPart = 0xFFFFFFFF;
#112     //设置物理地址的高32位
#113     maxAddress.u.HighPart = 0;
#114
#115     //分配连续的物理地址
#116     pdx->MemForImage = MmAllocateContiguousMemory(IMAGE_LENGTH,maxAddress);
#117     //获取物理地址
#118     PHYSICAL_ADDRESS pycialAddressForImage = MmGetPhysicalAddress(pdx->
MemForImage);
#119
#120     WRITE_REGISTER_BUFFER_UCHAR((PUCHAR)pdx->MemBar0+0x10000,
#121         (PUCHAR)&pycialAddressForImage.u.LowPart,4);
#122
#123     return STATUS_SUCCESS;
#124 }

```

此段代码可以在配套光盘中本章的 Test5 目录下找到。

16.7 小结

本章主要介绍 PCI 设备的驱动开发。首先介绍了 PCI 总线协议。作为驱动程序员，开发 PCI 驱动程序首先要了解 PCI 配置空间。根据读取 PCI 配置空间，可以得到 PCI 设备的所有资源。本章总结了四种获取 PCI 配置空间的方法。这其中包括使用标准的内核函数获取 PCI 配置空间，也包括模拟内核函数的实现获取 PCI 配置空间。对比这些方法，相信读者可以更深入地理解 Windows 操作系统的内核。

除此之外，本章还介绍了在 Windows 驱动中如何使用中断向量、设备物理内存等方法。最后笔者给出一个 PCI 驱动程序例子，读者可以根据需要进行修改。

UnRegistered



第 17 章 USB 设备驱动

USB 设备驱动和 PCI 设备驱动是 PC 中最主要的两种设备驱动程序。与 PCI 协议相比，USB 协议更复杂，涉及面较多。本章将介绍 USB 设备驱动开发。首先介绍 USB 协议，使读者对 USB 协议有个整体认识。然后介绍 USB 设备在 WDM 中的开发框架。由于操作系统的 USB 总线驱动程序提供了丰富的功能调用，因此开发 USB 驱动开发变得相对简单，只需要调用 USB 总线驱动接口。

17.1 USB 总线协议

USB 总线协议比 PCI 协议复杂的多，涉及 USB 物理层协议，又涉及 USB 传输层协议等。对于 USB 驱动程序开发者来说，不需要对 USB 协议的每个细节都很清楚。本节概要地介绍 USB 总线协议，并对驱动开发者需要了解的地方进行详细介绍。

17.1.1 USB 设备简介

USB 即通用串行总线 (Universal Serial Bus)，是一种支持即插即用的新型串行接口。也有人称之为“菊链 (daisy_chaining)”，是因为在一条“线缆”上有链接 127 个设备的能力。USB 要比标准串行口快得多，其数据传输率可达 4Mb/s~12 Mb/s (而老式的串行口最多是每秒 115Kb)。除了具有较高的传输率外，它还能给外围设备提供支持。

需要注意的是，这不是一种新的总线标准，而是计算机系统连接外围设备 (如键盘、鼠标、打印机等) 的输入/输出接口标准。到现在为止，计算机系统连接外围设备的接口还没有统一的标准，例如，键盘的插口是圆的、连接打印机要用 9 针或 25 针的并行接口、鼠标则要用 9 针或 25 针的串行接口。USB 能把这些不同的接口统一起来，仅用一个 4 针插头作为标准插头，如图 17-1 所示。通过这个标准插头，采用菊链形式可以把所有的外设连接起来，并且不会损失带宽。USB 正在取代当前 PC 上的串口和并口。

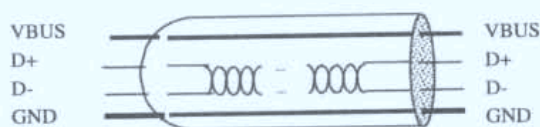


图 17-1 USB 的四条传输线

以 USB 连接设备时，所有的外设都在机箱外连接，连接外设不必再打开机箱；允许外设热插拔，而不必关闭主机电源。USB 采用“级联”方式，即每个 USB 设备用一个 USB 插头连接到另一个外设的 USB 插座上，而其本身又提供一个 USB 插座供下一个 USB 外设连接用。通过这种类似菊花链式的连接，一个 USB 控制器可以连接多达 127 个外设，而每个外设间距离（线缆长度）可达 5m。USB 能智能识别 USB 链上外围设备的插入或拆卸。

它可使多个设备在一个端口上运行，速度也比现在的串行口或并行口快得多，而且其总的连线在理论上说可以无限延长。对 PC 来说，以上这些都是一些难得的优点，因为不再需要 PS/2 端口、MIDI 端口等各种不同的端口了，还可以随时随地在各种设备上任意插拔。可以在一个端口上运行鼠标、控制手柄、键盘以及其他输入装置（例如数码相机），而且，也不必重新启动系统去做这些工作。现在 USB 设备正在快速增多，且由于操作系统已内置支持 USB 的功能，因而用户现在就可以方便地使用。显然，USB 为 PC 的外设扩充提供了一个很好的解决方案。

目前 USB 技术的发展，已经允许用户在不使用网卡、HUB 的情况下，直接通过 USB 技术将几台计算机连接起来组成小型局域网，用户只需要给各台计算机起个名字就可以开始工作。这种网络具备 Ethernet 网络的各种优点，同时少了 Ethernet 网络的许多限制。假设一位用户上班时使用笔记本电脑，回家时使用 PC 机，为实现数据传输，他可以通过采用 USB 技术的接口将两部电脑连接起来交换资源，其数据传输速度可达 12Mb/s，这是传统串行口无法比拟的。而且用户在组网的时候根本无须考虑 DIP、IRQ 等问题。此类技术除支持兼容 Ethernet 的软硬件外，也支持标准的网络通信协议，包括 IPX/SPX、NetBEUI 和 TCP/IP，这为通过 USB 技术组成的小局域网连接至大型网络或 Internet 提供了条件。

17.1.2 USB 连接拓扑结构

USB 设备的连接如图 17-2 所示，对于每个 PC 来说，都有一个或者多个称为 Host 控制器的设备，该 Host 控制器和一个根 Hub 作为一个整体。这个根 Hub 下可以接多级的 Hub，每个子 Hub 又可以接子 Hub。每个 USB 作为一个节点接在不同级别的 Hub 上。

(1) USB Host 控制器：每个 PC 的主板上都会有多个 Host 控制器，这个 Host 控制器其实就是一个 PCI 设备，挂载在 PCI 总线上。Host 控制器的驱动由微软公司提供，如图 17-3 所示，这是笔者 PC 中的 Host 控制器及 USB Hub 的驱动。值得注意的是，这里 Host 分别有两种驱动，一种是 1.0，另一种是 2.0，分别对应着 USB 协议 1.0 和 USB 协议 2.0。

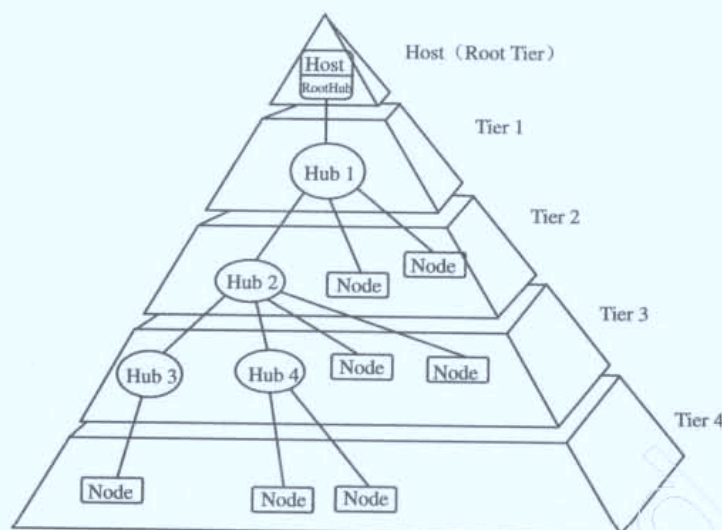


图 17-2 USB 连接拓扑结构

(2) USB Hub: 每个 USB Host 控制器都会自带一个 USB Hub, 被称为根 (Root) Hub。这个根 Hub 可以接子 (Sub) Hub, 每个 Hub 上挂载 USB 设备。一般 PC 有 8 个 USB 口, 通过外接 USB Hub, 可以插更多的 USB 设备。当 USB 设备插入到 USB Hub 或从上面拔出时, 都会发出电信号通知系统。这样可以枚举 USB 设备, 例如当被插入的时候, 系统就会创建一个 USB 物理总线, 并询问用户安装设备驱动。如图 17-4 所示为一个典型的 USB Hub 的示意图。

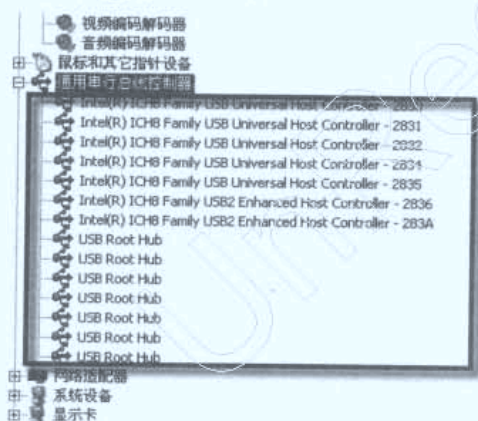


图 17-3 USB Host 和 USB Hub 驱动

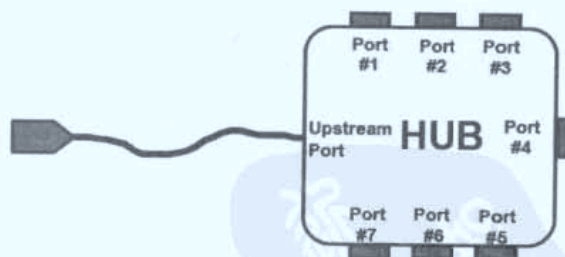


图 17-4 USB Hub 示意图

(3) USB 设备: USB 设备就是插在 USB 总线上工作的设备, 广义地讲 USB Hub 也算是 USB 设备。每个根 USB Hub 下可以直接或间接地连接 127 个设备, 并且彼此不会干扰。对于用户来说, 可以看成是 USB 设备和 USB 控制器直接相连, 之间通信需要满足 USB 的通信协议。

有的 USB 设备功能单一, 直接挂载在 USB Hub 上。而有的 USB 设备功能复杂, 会将

多个 USB 功能连在一起，成为一个复合设备，它甚至可以自己内部带一个 Hub，这个 Hub 下接多个 USB 子设备，其和多个子设备作为一个整体当做一个 USB 设备，如图 17-5 所示。

以上是 USB 的物理拓扑结构，但对于用户来说，可以略去 USB Hub 的概念，或者说 USB Hub 的概念对于用户可以看成是透明的。用户只需要将 USB 设备理解成一个 USB Host 连接多个逻辑设备。可能逻辑设备 1 和逻辑设备 2 是集中在第一个物理设备里，例如有的手机连接计算机后，系统会当做多个 USB 设备加载。因此，作为用户需要用如图 17-6 所示的逻辑拓扑结构理解 USB 拓扑结构。

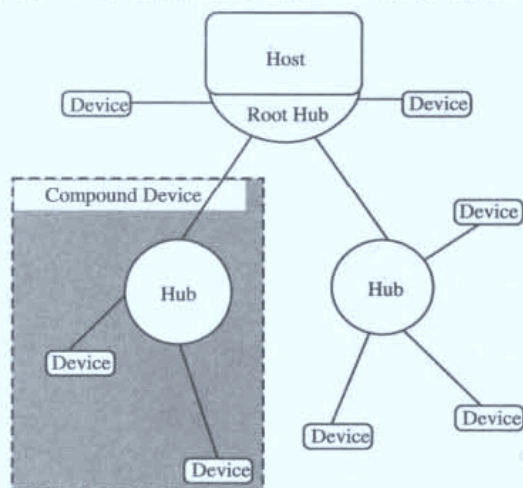


图 17-5 符合设备

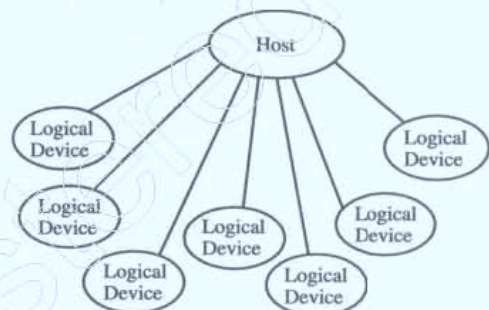


图 17-6 USB 逻辑拓扑结构

但对于具体 USB 设备来说，每个 USB 设备的传输绝对不会影响其他 USB 设备的传输。例如，在有 USB 设备传输的时候，其他 USB 设备的带宽不会被占用。对于 USB 设备来说，每个 USB 设备是直接连接到 USB Host 控制器上的。因此，应该用如图 17-7 所示的视角考虑 USB 设备的通信。

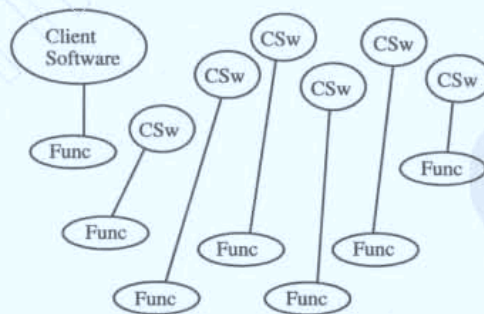


图 17-7 用户对 USB 设备的观察

17.1.3 USB 通信的流程

USB 的连接模式是 Host 和 Device 的连接模式，它不同于早期的串口和并口，所有的

请求必须是 Host 向 Device 发出，这就使 Host 端设计相对复杂，而 Device 端设计相对简单。Host 端会在主板的南桥设计好，而 Device 的厂商众多，厂商只需要遵循 USB 协议，重点精力可以放在设备的研发上，而与 PC 的通信不用过多考虑。

在 USB 的通信中，可以看成是一个分层的协议。分为三个层次，即最底层 USB 总线接口层、USB 设备层、功能通信层，如图 17-8 所示。

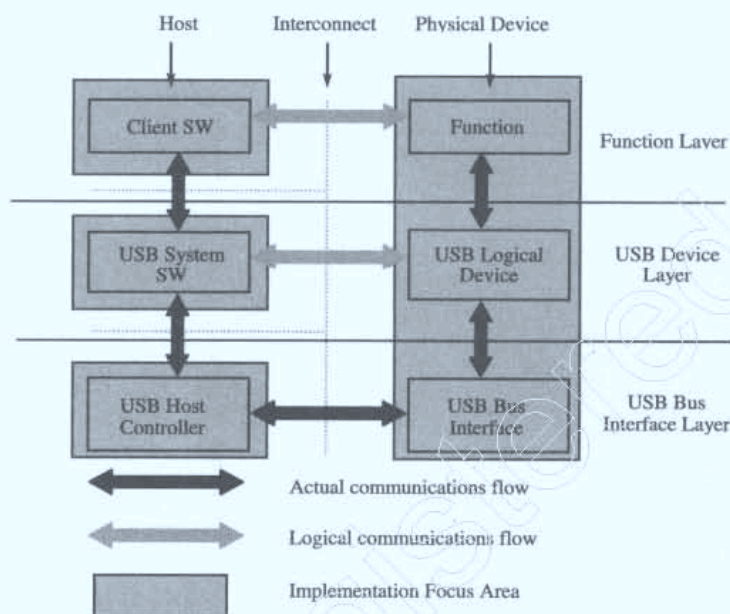


图 17-8 USB 协议

以 USB 摄像头设备为例，视频播放软件想通过 USB 总线得到 USB 摄像头捕捉的视频数据，这就相当于在功能层上。Client SW 是视频播放软件，Function 是 USB 摄像头。而这些数据的读取需要 USB 设备层提供的服务，在这一层上，主要是 USB 设备的驱动调度 Host 控制器向 USB 摄像头发出读请求。每个 USB 设备会有多个管道，使用哪个管道，传输的大小都需要指定。这个层次的 USB System SW 就是 USB 摄像头的驱动程序。而在 USB 设备一端一般会有小单片机或者处理芯片负责响应这种读请求，而这一层的传输又依赖于 USB 总线接口层的服务。在这一层，完全是 USB 的物理协议，包括如何分成更小的包 (packages) 传输，如何保证每次包传输不丢失数据等。

对于 USB 设备驱动程序员，主要是工作在 USB 设备层，向“上”对应用程序提供读写等接口，向“下”将读取某个管道的请求发往 USB Host 控制器驱动程序，它实现了最底层的传输请求。

对于每个 USB 设备，都有一个或者多个的接口 (Interface)，每个 Interface 都有多个端点 (Endpoints)，每个端点通过管道 (Pipes) 和 USB Host 控制器连接。每个 USB 设备都会有一个特殊的端点，即 Endpoint0，它负责传输设备的描述信息，同时也负责传输 PC 与设备之间的控制信息，如图 17-9 所示。

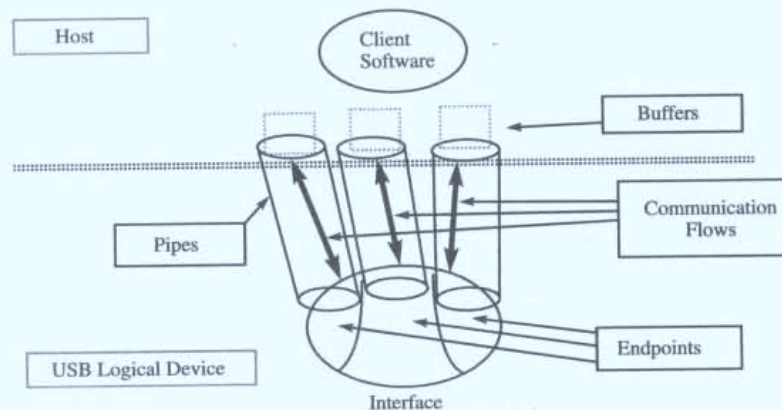


图 17-9 USB 管道与端点

17.1.4 USB 四种传输模式

当 USB 插入 USB 总线时，USB 控制器会自动为该 USB 设备分配一个数字来标示这个设备。另外，在设备的每个端点都有一个数字来表明这个端点。

USB 设备驱动向 USB 控制器驱动请求的每次传输被称为一个事务 (Transaction)，事务有四种类型，分别是 Bulk Transaction、Control Transaction、Interrupt Transaction 和 Isochronous Transaction。每次事务都会分解成若干个数据包在 USB 总线上传输。每次传输必须历经两个或三个部分，第一部分是 USB 控制器向 USB 设备发出命令，第二部分是 USB 控制器和 USB 设备之间传递读写请求，其方向主要看第一部分的命令是读还是写，第二部分有时候可以没有。第三部分是握手信号。以下针对这四种传输，分别进行讲解。

1. Bulk 传输事务

顾名思义，此种事务传输主要是大块的数据，传送这种事务的管道叫做 Bulk 管道。这种事务传输的时候分为三部分，如图 17-10 所示。第一部分是 Host 端发出一个 Bulk 的令牌请求，如果令牌是 IN 请求则是从 Device 到 Host 的请求，如果是 OUT 令牌，则是从 Host 到 Device 端的请求。

第二部分是传送数据的阶段，根据先前请求的令牌的类型，数据传输有可能是 IN 方向，也有可能是 OUT 方向。传输数据的时候用 DATA0 和 DATA1 令牌携带着数据交替传送。

第三部分是握手信号。如果数据是 IN 方向，握手信号应该是 Host 端发出，如果是 OUT 方向，握手信号应该是 Device 端发出。握手信号可以为 ACK，表示正常响应，也可以是 NAK 表示没有正确传送。STALL 表示出现主机不可预知的错误。

在第二部分，即传输数据包的时候，数据传送由 DATA0 和 DATA1 数据包交替发送。数据传输格式 DATA1 和 DATA0，这两个是重复数据，确保在 1 数据丢失时 0 可以补上，不至于数据丢失，如图 17-11 所示。

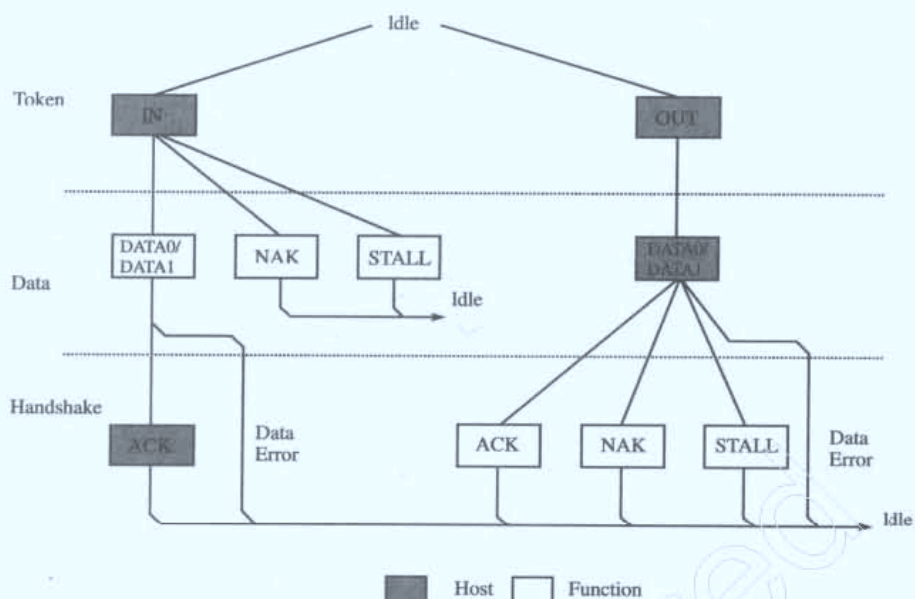


图 17-10 Bulk 传输

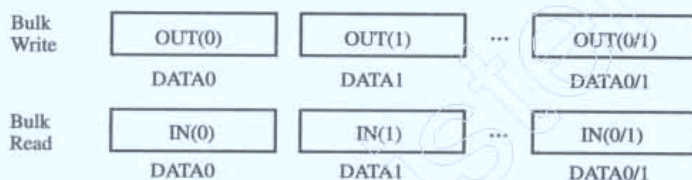


图 17-11 Bulk 传输时的令牌

2. 控制传输事务

控制传输是负责向 USB 设置一些控制信息，传送这种事务的管道是控制管道。在每个 USB 设备中都会有控制管道，也就是说控制管道在 USB 设备中是必须的。控制传输也分为三个阶段，即令牌阶段、数据传送阶段、握手阶段，如图 17-12 所示。

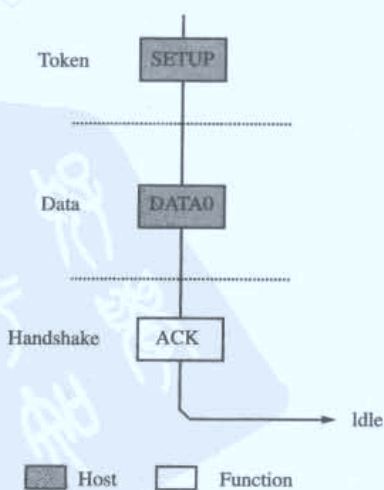


图 17-12 控制传输事务

3. 中断传输事务

在 USB 设备中，有种处理机制类似于 PCI 中断的机制，这就是中断事务。中断事务的数据量很小，一般用于通知 Host 某个事件的来临，例如 USB 鼠标，鼠标移动或者鼠标单击等操作都会通过中断管道来向 Host 传送事件。在中断事务中，也分为三个阶段，即令牌阶段、数据传输阶段、握手阶段，如图 17-13 所示。

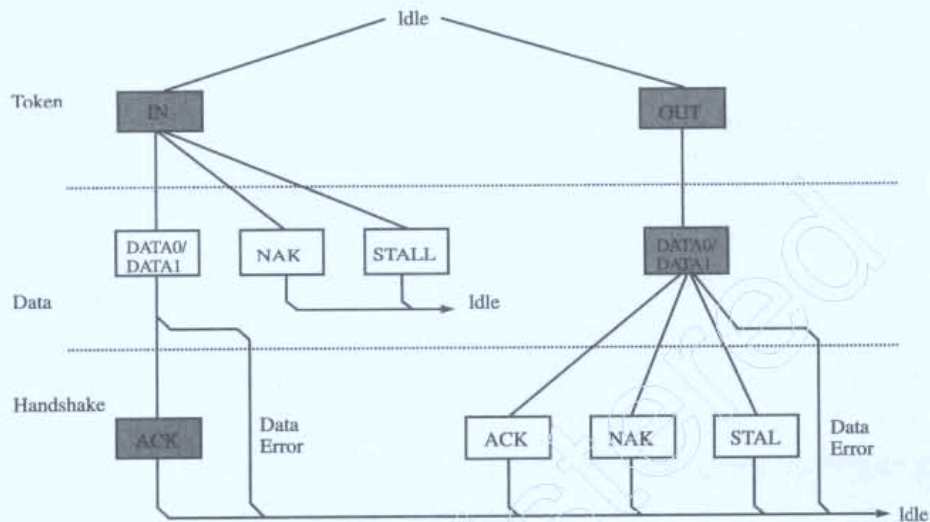


图 17-13 中断传输事务

4. 同步传输事务

USB 设备中还有一种事务叫同步传输事务，这种事务能保证传输的同步性。例如，在 USB 摄像头中传输视频数据的时候会采用这种事务，这种事务能保证每秒有固定的传输量，但与 Bulk 传输不同，它允许有一定的误码率，这样符合视频会议等传输的需求，因为视频会议首先要保证实时性，在一定条件下，允许有一定的误码率。同步传输事务有只有两个阶段，即令牌阶段、数据阶段，因为不关心数据的正确性，故没有握手阶段，如图 17-14 所示。

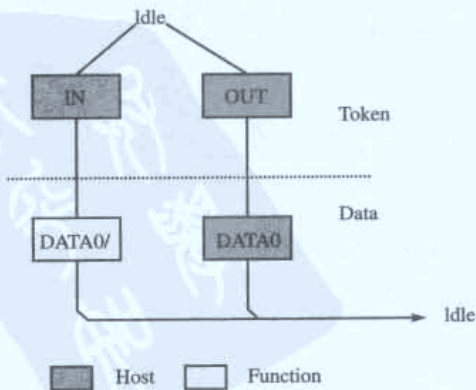


图 17-14 同步传输事务

17.2 Windows 下的 USB 驱动

在 Windows 上开发 USB 驱动相对来说比较简单，主要是因为微软已经提供了完备的 USB 总线驱动，程序员编写的设备驱动只需调用总线驱动即可。在 Windows 上还有一些工具软件可以帮助开发者查看 USB 的各类信息，包括设备描述符、配置描述符等。当然，这些描述符在驱动中也会用到。本节将介绍这些工具软件，并介绍这些描述符。

17.2.1 观察 USB 设备的工具

在学习编写 USB 驱动之前，有几个 USB 查看工具需要向读者介绍一下，通过用这些工具能方便地学习 USB 协议。

首先需要介绍的就是 DDK 中提供的工具，该工具叫 `usbview`，位于 DDK 的子目录 `src\wdm\usb\usbview` 下，需要用 DDK 编译环境进行编译。如图 17-15 所示为 `usbview` 的界面，在笔者的计算机里插入了一个 USB 移动硬盘，在这个软件中已经清楚地列举除了该 USB 设备的各个信息，如图设备描述符、管道描述符等。

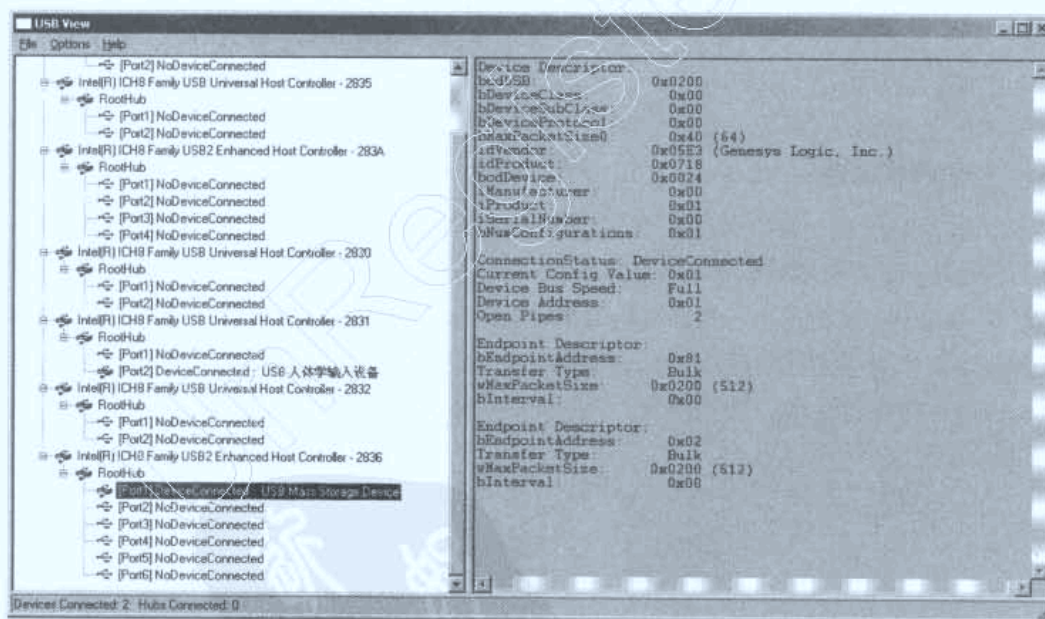


图 17-15 USBView

另一个有用的工具是 `BusHound`，如图 17-16 所示。`BusHound` 用于监视 USB 设备的传输数据，它的实现原理是在 USB 设备驱动之上加载一层过滤驱动程序，将 IRP 进行拦截，因此可以观察到所有 USB 数据的传输。使用该软件时需要指明监视哪种 USB 设备，如图 17-16 所示，在需要监视的设备上打钩。笔者这里监视的是一个 USB 移动硬盘。另外，在下面会列出该设备的基本信息，如管道 0 是控制管道，管道 1 是输出管道，管道 2 是输入管道。

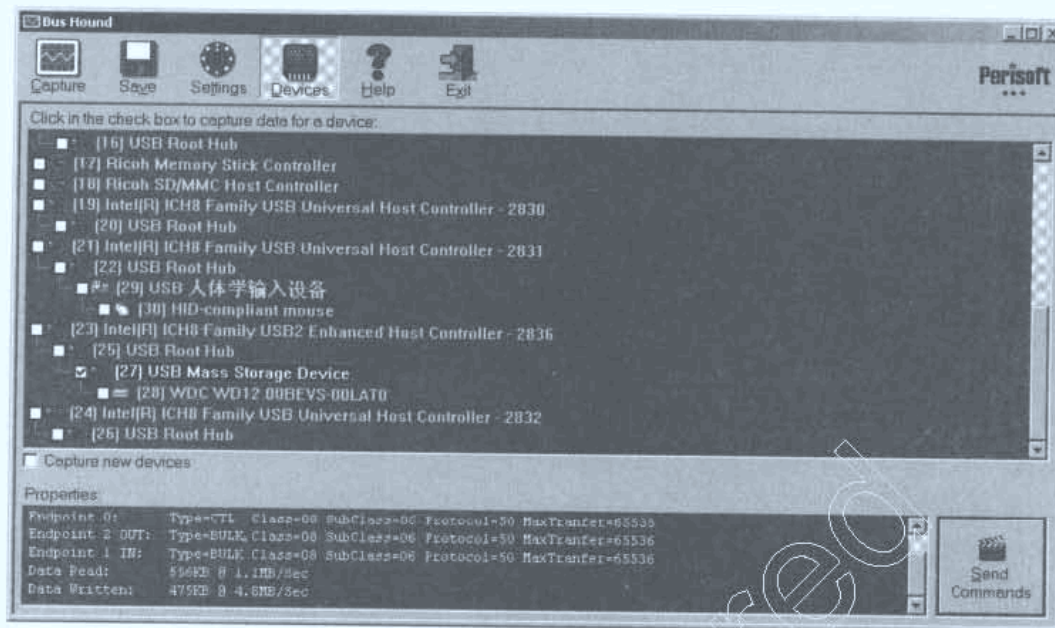


图 17-16 BusHound

该软件将 USB 的传输完全进行监视，包括每个 USB 的各个管道中的传输情况，都一一进行记录，非常有利于调试驱动。如图 17-17 所示，Device 一栏中标识是何种设备，例如 27.2 意味着第 27 号设备的第 2 号管道。Phase 一栏标识传输是输入还是输出。在 Data 一栏中记录着一次传输的具体内容。



图 17-17 BusHound 监视数据传输

17.2.2 USB 设备请求

USB 的请求是通过控制管道传输的，请求是 8 个字节，按照如表 17-1 所示的排列发送：

表 17-1

偏移量	变量	大小	数值	描 述
0	bmRequestType	1 字节	位图	第 7 位: 数据方向位 0 = 主机到设备 1 = 设备到主机 第 6-5 位: 类型 0 = 标准 1 = 类 2 = 厂商自定义 3 = 保留 第 4-0 位: 接收者 0 = 对设备的请求 1 = 对接口的请求 2 = 对管道（端点）的请求 3 = 其他 4-31 = 保留
1	bRequest	1 字节	数值	请求类别
2	wValue	2 字节	数值	不同请求含义不同
4	wIndex	2 字节	数值	不同请求含义不同
6	wLength	2 字节	数值	表示需要有多少数据返回

其中，bRequest 代表不同的 USB 请求，它们分别是以下的几种请求，定义在 DDK 的 usb100.h 文件中：

```
#define USB_REQUEST_GET_STATUS          0x00
#define USB_REQUEST_CLEAR_FEATURE      0x01
#define USB_REQUEST_SET_FEATURE        0x03
#define USB_REQUEST_SET_ADDRESS        0x05
#define USB_REQUEST_GET_DESCRIPTOR     0x06
#define USB_REQUEST_SET_DESCRIPTOR     0x07
#define USB_REQUEST_GET_CONFIGURATION  0x08
#define USB_REQUEST_SET_CONFIGURATION  0x09
#define USB_REQUEST_GET_INTERFACE      0x0A
#define USB_REQUEST_SET_INTERFACE      0x0B
```

17.2.3 设备描述符

在控制管道发起 USB 设备请求，其中很常见的请求是 USB_REQUEST_GET_DESCRIPTOR，即请求 USB 设备回答设备或者管道描述符。在请求描述符时，

bmRequestType 可以指定是针对设备还是针对管道的。

当请求设备描述符后，设备会回答主机该设备的设备描述符，设备描述符是一种固定的数据结构，它定义在 DDK 中的 usb100.h 文件中。

```
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;
```

- bLength: 设备描述符的 bLength 域应等于 18。
- bDescriptorType: bDescriptorType 域应等于 1，以指出该结构是一个设备描述符。
- bcdUSB: 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: 指出设备类型。
- bDeviceSubClass: 指出设备子类型。
- bDeviceProtocol: 指出设备类型所使用的协议。
- bMaxPacketSize0: 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。
- idVendor: 厂商代码。
- idProduct: 厂商专用的产品标识。
- bcdDevice: bcdDevice 指出设备的发行版本号（0x0100 对应版本 1.0）。
- iManufacturer、iProduct、iSerialNumber: 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。
- bNumConfigurations: 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）。

如图 17-18 所示为笔者用 BusHound 截获的 USB 移动硬盘的请求设备描述符，前面已经介绍过，在控制管道中，传输分为三个阶段。第一阶段是令牌阶段，这里 Host 向设备发送“80 06 00 01 00 00 12 00”8 个字节，可以参见表 17-1 中的解释。第二阶段是数据传

输阶段，方向是由设备传给主机，这个例子中设备给主机传递了 18 (0x12) 个字节，这 18 个字节对应着 USB_DEVICE_DESCRIPTOR 数据结构。第三阶段是握手阶段，在 BusHound 软件中没有体现出来。

27.9	CTL	80 06 00 01 00 00 12 00	GET_DESCRIPTOR	1.1.0
27.0	DI	12 01 00 02 00 00 00 40 e3 05 19 07 24 00 00 018....8...	1.2.0
		00 01	..	1.2.16

图 17-18 用 BusHound 抓取设备描述符

17.2.4 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: bLength 应该为 9。
- bDescriptorType: bDescriptorType 应该为 2，即是一个 9 字节长的配置描述符。
- wTotalLength: 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。
- bNumInterfaces: 指出该配置有多少个接口。
- bConfigurationValue: bConfigurationValue 域是该配置的索引值。
- iConfiguration: 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: bmAttributes 字节包含描述该配置中设备电源和其他特性的位掩码。
- MaxPower: MaxPower 域中指出要从 USB 总线上获取的最大电流。

如图 17-19 所示为笔者用 BusHound 截获的 USB 移动硬盘的请求配置描述符。其中第一行是 Host 向 Device 发送 Token 令牌，而第二行是 Device 向 Host 返回的数据。

29.0	CTL	80 06 00 02 00 00 09 00	GET_DESCRIPTOR	2.1.0
29.0	DI	09 02 20 00 01 01 04 e0 322	2.2.0

图 17-19 用 BusHound 抓取配置描述符

17.2.5 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。

```
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: bLength 应该为 9。
- bDescriptorType: bDescriptorType 域应为 4。
- bInterfaceNumber: bInterfaceNumber 是索引值。
- bAlternateSetting: bAlternateSetting 是索引值。用在 SET_INTERFACE 控制事务中以指定要激活的接口。
- bNumEndpoints: bNumEndpoints 域指出该接口有多少个端点，不包括端点 0，端点 0 被认为总是存在的，并且是接口的一部分。
- bInterfaceClass: bInterfaceClass 为接口类。
- bInterfaceSubClass: bInterfaceSubClass 为子接口类。
- bInterfaceProtocol: bInterfaceProtocol 为协议。
- iInterface: iInterface 是一个串描述符的索引，0 表示该接口无描述串。

如图 17-20 所示为笔者用 BusHound 截获的 USB 移动硬盘的请求配置描述符和接口描述符。其中第一行是 Host 向 Device 发送 Token 令牌，而第二行是 Device 向 Host 返回的数据。可以看出图中有一个配置描述符，后面紧接着一个接口描述符（“09 04 00 00 02 00 06 50 05”），后面还有两个接口描述符（下一节介绍）。

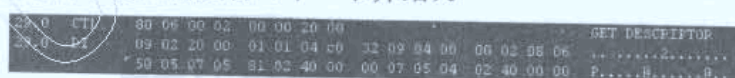


图 17-20 用 BusHound 抓取接口描述符

17.2.6 端点描述符

接口可以没有或多个端点描述符，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: bLength 应为 7。
- bDescriptorType: bDescriptorType 应该为 5。
- bEndpointAddress: bEndpointAddress 域编码端点的方向性和端点号。
- bmAttributes: bmAttributes 的低两位指出端点的类型。0 代表控制端点, 1 代表等时端点, 2 代表批量端点, 3 代表中断端点。
- wMaxPacketSize: wMaxPacketSize 值指出该端点在一个事务中能传输的最大数据量。
- bInterval: 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。

17.3 USB 驱动开发实例

本节具体介绍如何进行 USB 驱动的开发, 本节采用的源码来源于 DDK 的源程序, 其位置在 DDK 子目录的 src\wdm\usb\bulkusb 目录下。该示例很全面地支持了即插即用 IRP 的处理, 也很全面地支持了电源管理, 同时很好地支持了 USB 设备的 bulk 读写。如果从头开发 USB 驱动, 往往很难达到 USB 驱动的稳定性和兼容性, 所以强烈建议读者在此驱动修改的基础上进行 USB 驱动开发。

17.3.1 功能驱动与物理总线驱动

DDK 已经为 USB 驱动开发人员提供了功能强大的 USB 物理总线驱动 (PDO), 程序员需要做的事情是完成功能驱动 (FDO) 的开发。驱动开发人员不需要了解 USB 如何将请求转化成数据包等细节, 程序员只需要指定何种管道, 发送何种数据即可。

当功能驱动想向某个管道发出读写请求时, 首先构造请求发给 USB 总线驱动。这种请求是标准的 USB 请求, 被称为 URB (USB Request Block), 即 USB 请求块。这种 URB 被发送到 USB 物理总线驱动以后, 被 USB 总线驱动所解释, 进而转化成请求发往 USB HOST 驱动或者 USB HUB 驱动, 如图 17-21 所示。

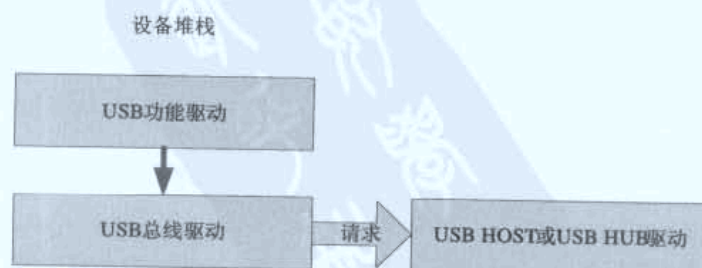


图 17-21 总线驱动与功能驱动的关系

可以看出，USB 总线驱动完成了大部分工作，并留给 USB 功能驱动标准的接口，即 URB 请求。USB 驱动开发人员只需要根据不同的 USB 设备的设计要求，在相应的管道中发起 URB 请求即可。

17.3.2 构造 USB 请求包

USB 驱动在与 USB 设备通信的时候，如在控制管道中获取设备描述符、配置描述符、端点描述符，或者在 Bulk 管道中获取大量数据，都是通过创建 USB 请求包（URB）来完成的。URB 中填充需要对 USB 的请求，然后将 URB 作为 IRP 的一个参数传递给底层的 USB 总线驱动。在 USB 总线驱动中，能够解释不同 URB，并将其转化为 USB 总线上的相应数据包。

DDK 提供了构造 URB 的内核函数 `UsbBuildGetDescriptorRequest`，其声明如下：

```
VOID
UsbBuildGetDescriptorRequest(
    IN OUT PURB Urb,
    IN USHORT Length,
    IN UCHAR DescriptorType,
    IN UCHAR Index,
    IN USHORT LanguageId,
    IN PVOID TransferBuffer OPTIONAL,
    IN PMDL TransferBufferMDL OPTIONAL,
    IN ULONG TransferBufferLength,
    IN PURB Link OPTIONAL
);
```

- `Urb`：用来输出的 URB 结构的指针。
- `Length`：用来描述该 URB 结构的大小。
- `DescriptorType`：描述该 URB 的类型。它可以是 `USB_DEVICE_DESCRIPTOR_TYPE`、`USB_CONFIGURATION_DESCRIPTOR_TYPE` 和 `USB_STRING_DESCRIPTOR_TYPE`。
- `Index`：用来描述设备描述符的索引。
- `LanguageId`：用来描述语言 ID。
- `TransferBuffer`：如果用缓冲区读取设备，`TransferBuffer` 是缓冲区内存的指针。
- `TransferBufferMDL`：如果用直接读取内存时，`TransferBufferMDL` 是直接读取内存时 MDL 的指针。
- `TransferBufferLength`：对于该 URB 所操作内存的大小。

在功能驱动中，所有与 USB 的通信，都需要用这个函数创建 URB，并通过 IRP 发送到底层 USB 总线驱动，以下是一个最基本的示例。

```
#001      UsbBuildGetDescriptorRequest(
#002          urb,
#003          (USHORT) sizeof(struct _URB_CONTROL_DESCRIPTOR_REQUEST),
#004          USB_DEVICE_DESCRIPTOR_TYPE,
#005          0,
```



```

#006         0,
#007         deviceDescriptor,
#008         NULL,
#009         siz,
#010         NULL);

```

17.3.3 发送 USB 请求包

功能驱动将 URB 包构造完毕后，就可以发送到底层总线驱动上了。URB 包要和一个 IRP 相关联起来，这就需要用 `IoBuildDeviceIoControlRequest` 创建一个 I/O 控制码的 IRP，然后将 URB 作为 IRP 的参数，用 `IoCallDriver` 将 URB 发送到底层总线驱动上。由于上层驱动无法知道底层驱动是同步还是异步完成的，因此需要做一个判断。if 语句判断当异步完成 IRP 时，用事件等待总线驱动完成这个 IRP。

```

#001 //该函数实现对发送 URB 到 USB 物理总线驱动
#002 NTSTATUS
#003 CallUSBd(
#004     IN PDEVICE_OBJECT DeviceObject,
#005     IN PURB            Urb
#006 )
#007 {
#008     PIRP            irp;
#009     KEVENT          event;
#010     NTSTATUS        ntStatus;
#011     IO_STATUS_BLOCK ioStatus;
#012     PIO_STACK_LOCATION nextStack;
#013     PDEVICE_EXTENSION deviceExtension;
#014
#015     //首先是变量初始化
#016     irp = NULL;
#017     deviceExtension = DeviceObject->DeviceExtension;
#018     //初始化事件
#019     KeInitializeEvent(&event, NotificationEvent, FALSE);
#020     //创建 IO 控制码相关的 IRP
#021     irp = IoBuildDeviceIoControlRequest(IOCTL_INTERNAL_USB_SUBMIT_URB,
#022                                         deviceExtension->TopOfStackDeviceObject,
#023                                         NULL,
#024                                         0,
#025                                         NULL,
#026                                         0,
#027                                         TRUE,
#028                                         &event,
#029                                         &ioStatus);
#030
#031     if(!irp) {
#032         //如果 IRP 创建失败则返回
#033         BulkUsb_DbgPrint(1, ("IoBuildDeviceIoControlRequest failed\n"));
#034         return STATUS_INSUFFICIENT_RESOURCES;
#035     }
#036     //得到下一层设备栈
#037     nextStack = IoGetNextIrpStackLocation(irp);
#038     ASSERT(nextStack != NULL);
#039     nextStack->Parameters.Others.Argument1 = Urb;
#040     BulkUsb_DbgPrint(3, ("CallUSBd:"));

```

```

#041     BulkUsb_IoIncrement(deviceExtension);
#042     //通过 IoCallDriver 将 IRP 发送到底层驱动
#043     ntStatus = IoCallDriver(deviceExtension->TopOfStackDeviceObject, irp);
#044     //如果 IRP 是异步完成时, 等待其结束
#045     if(ntStatus == STATUS_PENDING) {
#046         //等待 IRP 结束
#047         KeWaitForSingleObject(&event,
#048                               Executive,
#049                               KernelMode,
#050                               FALSE,
#051                               NULL);
#052         ntStatus = ioStatus.Status;
#053     }
#054     //调用结束
#055     BulkUsb_DbgPrint(3, ("CallUSBD::"));
#056     BulkUsb_IoDecrement(deviceExtension);
#057     return ntStatus;
#058 }

```

此段代码可以在配套光盘中本章的 sys 目录下找到。

17.3.4 USB 设备初始化

USB 驱动的初始化和一般驱动类似, 首先是进入入口函数 DriverEntry, 在 DriverEntry 函数中, 分别指定各个 IRP 的派遣函数地址、指定 AddDevice 例程函数地址、指定 Unload 例程函数地址等。

在 AddDevice 例程中, 创建功能设备对象, 然后将该对象挂载在总线设备对象之上, 从而形成设备栈。另外为设备创建一个设备链接, 便于应用程序可以找到这个设备。也可以根据具体需要, 从注册表中读取一些必要的设置。

17.3.5 USB 设备的插拔

由于 USB 设备驱动是基于 WDM 框架的, 因此需要对即插即用消息进行处理。BulkUSB 程序对即插即用 IRP 的支持非常完善, 具体可以参照其代码, 这里简单提一下其对插拔的处理。

插拔设备会设计 4 个即插即用 IRP, 包括 IRP_MN_START_DEVICE、IRP_MN_STOP_DEVICE、IRP_MN_EJECT 和 IRP_MN_SURPRISE_REMOVAL。其中, IRP_MN_START_DEVICE 消息是当驱动争取加载并运行时, 操作系统的即插即用管理器会将这个 IRP 发往设备驱动。因此, 当获得这个 IRP 后, USB 驱动需要获得 USB 设备类别描述符, 如设备描述符、配置描述符、接口描述符、端点描述符等。并通过这些描述符, 从中获取有用信息, 记录在设备扩展中。

IRP_MN_STOP_DEVICE 是设备关闭前, 即插即用管理器发的 IRP。USB 驱动获得这个 IRP 时, 应该尽快结束当前执行的 IRP, 并将其逐个取消掉。另外, 在设备扩展中还应该有表示当前状态的变量, 当 IRP_MN_STOP_DEVICE 来临时, 将当前状态记录成停止状态。

IRP_MN_EJECT 是设备被正常弹出, 而 IRP_MN_SURPRISE_REMOVAL 则是设备非自然弹出, 有可能意外掉电或者强行拔出等。在这种 IRP 到来的时候, 应该强迫所有未完成的读写 IRP 结束并取消。并且将当前的设备状态设置成设备被拔掉。

17.3.6 USB 设备的读写

USB 设备接口主要是为了传送数据, 80% 的传输是通过 Bulk 管道。在 BulkUSB 驱动中, Bulk 管道的读取是在 IRP_MJ_READ 和 IRP_MJ_WRITE 的派遣函数中, 这样在应用程序中就可以通过 ReadFile 和 WriteFile 等 API 对设备进行操作了。

在 IRP_MJ_READ 和 IRP_MJ_WRITE 的派遣例程中设置了完成例程, 如图 17-22 所示。其原理是将读写的大小分成单位为 BULKUSB_MAX_TRANSFER_SIZE 的若干块, 依次将请求发往底层 USB 总线驱动。第一个块是派遣例程先设置 BULKUSB_MAX_TRANSFER_SIZE 大小的读写, 并设置完成例程, 然后将请求发往 USB 总线驱动。当 USB 总线驱动完成 BULKUSB_MAX_TRANSFER_SIZE 大小的读写后, 会调用读写的完成例程。

这时候在完成例程中再次发起 BULKUSB_MAX_TRANSFER_SIZE 大小的读写, 并将请求发往底层 USB 总线驱动, 当 USB 总线驱动完成后, 又会进入完成例程。之后发送第三个数据块, 并且依此类推直到传送完毕。

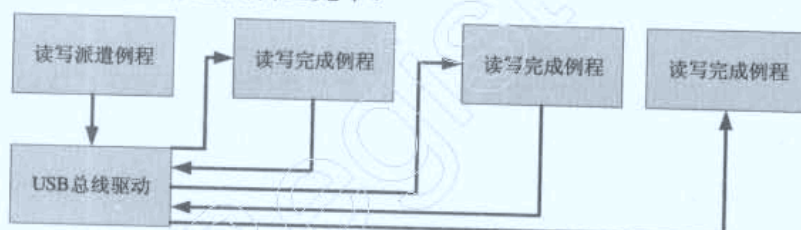


图 17-22 USB 读写派遣例程与完成例程

以下是 BulkUSB 的读写派遣函数的部分代码：

```
#001 NTSTATUS
#002 BulkUsb_DispatchReadWrite(
#003     IN PDEVICE_OBJECT DeviceObject,
#004     IN PIRP Irp
#005 )
#006 {
#007     PMDL mdl;
#008     PURB urb;
#009     ULONG totalLength;
#010     ULONG stageLength;
#011     ULONG urbFlags;
#012     BOOLEAN read;
#013     NTSTATUS ntStatus;
#014     ULONG_PTR virtualAddress;
#015     PFILE_OBJECT fileObject;
#016     PDEVICE_EXTENSION deviceExtension;
#017     PIO_STACK_LOCATION irpStack;
#018     PIO_STACK_LOCATION nextStack;
```

```

#019     PBULKUSB_RW_CONTEXT  rwContext;
#020     PUSHB_PIPE_INFORMATION pipeInformation;
#021
#022     //初始化变量
#023     urb = NULL;
#024     mdl = NULL;
#025     rwContext = NULL;
#026     totalLength = 0;
#027     irpStack = IoGetCurrentIrpStackLocation(Irp);
#028     fileObject = irpStack->FileObject;
#029     read = (irpStack->MajorFunction == IRP_MJ_READ) ? TRUE : FALSE;
#030     deviceExtension = (PDEVICE_EXTENSION) DeviceObject->DeviceExtension;
#031     //....略
#032
#033     //设置完成例程的参数
#034     rwContext = (PBULKUSB_RW_CONTEXT) ExAllocatePool(NonPagedPool,
#035                                                       sizeof(BULKUSB_RW_CONTEXT));
#036     //...略
#037     if(Irp->MdlAddress) {
#038         totalLength = MmGetMdlByteCount(Irp->MdlAddress);
#039     }
#040     //设置 URB 标志
#041     urbFlags = USBD_SHORT_TRANSFER_OK;
#042     virtualAddress = (ULONG_PTR) MmGetMdlVirtualAddress(Irp->MdlAddress);
#043
#044     //判断是读还是写
#045     if(read) {
#046         urbFlags |= USBD_TRANSFER_DIRECTION_IN;
#047     }
#048     else {
#049         urbFlags |= USBD_TRANSFER_DIRECTION_OUT;
#050     }
#051
#052     //设置本次读写的大小
#053     if(totalLength > BULKUSB_MAX_TRANSFER_SIZE) {
#054         stageLength = BULKUSB_MAX_TRANSFER_SIZE;
#055     }
#056     else {
#057         stageLength = totalLength;
#058     }
#059
#060     //建立 MDL
#061     mdl = IoAllocateMdl((PVOID) virtualAddress,
#062                        totalLength,
#063                        FALSE,
#064                        FALSE,
#065                        NULL);
#066     //将新 MDL 进行映射
#067     IoBuildPartialMdl(Irp->MdlAddress,
#068                      mdl,
#069                      (PVOID) virtualAddress,
#070                      stageLength);
#071
#072     //申请 URB 数据结构
#073     urb = ExAllocatePool(NonPagedPool, sizeof(struct _URB_BULK_OR_INTERRUPT_
TRANSFER));
#074
#075     //建立 Bulk 管道的 URB

```



```

#076     UsbBuildInterruptOrBulkTransferRequest(
#077                                     urb,
#078                                     sizeof(struct _URB_BULK_OR_INTERRUPT_TRANSFER),
#079                                     pipeInformation->PipeHandle,
#080                                     NULL,
#081                                     mdl,
#082                                     stageLength,
#083                                     urbFlags,
#084                                     NULL);
#085
#086     //设置完成例程参数
#087     rwContext->Urb           = urb;
#088     rwContext->Mdl           = mdl;
#089     rwContext->Length        = totalLength - stageLength;
#090     rwContext->Numxfer       = 0;
#091     rwContext->VirtualAddress = virtualAddress + stageLength;
#092     rwContext->DeviceExtension = deviceExtension;
#093     //设置设备堆栈
#094     nextStack = IoGetNextIrpStackLocation(Irp);
#095     nextStack->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;
#096     nextStack->Parameters.Others.Argument1 = (PVOID) urb;
#097     nextStack->Parameters.DeviceIoControl.IoControlCode =
#098                                     IOCTL_INTERNAL_USB_SUBMIT_URB;
#099     //设置完成例程
#100     IoSetCompletionRoutine(Irp,
#101                           (PIO_COMPLETION_ROUTINE)BulkUsb_ReadWriteCompletion,
#102                           rwContext,
#103                           TRUE,
#104                           TRUE,
#105                           TRUE);
#106     //将当前 IRP 阻塞
#107     IoMarkIrpPending(Irp);
#108     //将 IRP 转发到底层 USB 总线驱动
#109     ntStatus = IoCallDriver(deviceExtension->TopOfStackDeviceObject,
#110                             Irp);
#111     //...略去对不成功时的处理
#112     return STATUS_PENDING;
#113 }

```

此段代码可以在配套光盘中本章的 sys 目录下找到。

17.4 小结

本章介绍了 USB 总线协议的基本框架，其中包括 USB 总线的拓扑结构，USB 通信的流程，还有 USB 的四种传输模式。笔者用一些工具软件带领读者分析了各种 USB 令牌、设备描述符等。

USB 驱动程序的主要功能就是设置这些 USB 令牌，和获取 USB 设备描述符。USB 驱动程序将这些请求最终转化为 USB 请求包（URB 包），然后发往 USB 总线驱动程序。USB 总线驱动提供了丰富的功能，它封装了 USB 协议，提供了标准的接口。这使得 USB 驱动程序的编写变得简单，程序员不必过多地了解 USB 总线协议，就可以编写出功能强大的 USB 驱动程序。

第 18 章 SDIO 设备驱动

SDIO 卡是在 SD 内存卡接口基础之上发展起来的接口，SDIO 接口兼容以前的 SD 内存卡，并且可以连接 SDIO 接口的设备。SDIO 设备种类正在逐渐增加，目前有蓝牙设备、网卡设备、电视卡设备等。但是，有关介绍 SDIO 设备驱动开发的资料却很少。

本章首先介绍 SDIO 接口的协议，然后介绍 SDIO 设备的 WDM 的开发框架，最后给出 SDIO 驱动程序的开发实例。

18.1 SDIO 协议

SDIO 协议类似 USB 总线协议，但比 USB 总线协议简单。SDIO 协议是由 SD 卡的协议演化升级而来，很多地方保留了 SD 卡读写协议。同时，SDIO 协议又在 SD 卡协议之上添加了 CMD52 和 CMD53 命令。

18.1.1 SD 内存卡概念

SD 卡 (Secure Digital Memory Card) 是一种基于半导体快闪记忆器的新一代记忆设备。SD 卡由日本松下、东芝及美国 SanDisk 公司于 1999 年 8 月共同开发研制。大小犹如一张邮票的 SD 记忆卡，重量只有 2 克，但却拥有高记忆容量、快速数据传输率、极大的移动灵活性以及很好的安全性。

SD 卡在 24mm×32mm×2.1mm 的体积内结合了 SanDisk 快闪记忆卡控制与 MLC (Multilevel Cell) 技术和 Toshiba (东芝) 0.16u 及 0.13u 的 NAND 技术，通过 9 针的接口界面与专门的驱动器相连接，不需要额外的电源来保持其上记忆的信息。而且它是一体化固体介质，没有任何移动部分，所以不用担心机械运动的损坏。

SD 卡体积小巧，广泛应用在数码相机上，其最大的特点就是通过加密功能，保证数据资料的安全保密。SD 卡在外形上同 MultiMedia Card (MMC) 卡保持一致，并且兼容

MMC 卡接口规范。不过需要注意的是，在某些产品例如手机上，SD 卡和 MMC 卡是不能兼容的。SD 卡在售价方面要高于同容量的 MMC 卡。SD 内存卡的形状如图 18-1 所示。

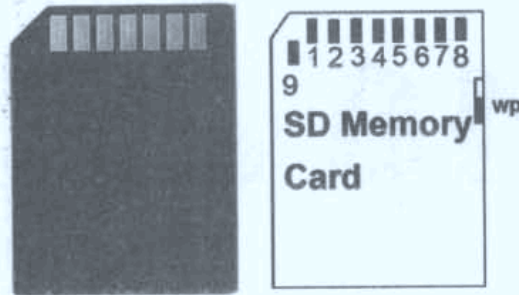


图 18-1 SD 内存卡

18.1.2 SDIO 卡概念

SDIO 卡与 SD 内存卡有着同样形状，并保持同样的接口。这样能保证支持 SDIO 卡的接口。

为了在个人计算机和 PDA 等中添加功能而使用 SD 卡的规格，由于可以使用比 PC 卡形状还小的卡和插槽，相比于笔记本电脑，更多采用于小巧的 PDA 中。主要销售的产品有，面向 PDA 的通信卡和蓝牙通信卡，无线网卡，小型数码相机等。

为了使用 SDIO，必须要有对应的插槽，数码相机等 memory card 的专用插槽不对应 SDIO 卡。SDIO 插槽中能够插入 SD memory card 来读写。

SDIO 在 SD 标准上定义了一种外设接口。目前，SDIO 有两类主要应用——可移动和不可移动。目前的可移动设备作为 Palm 和 Windows Mobile 的扩展设备，用来增加蓝牙、照相机、GPS 和 802.11b 功能。不可移动设备遵循相同的电气标准，但不要求符合物理标准。某些手机内包含通过 SDIO 连接 CPU 的 802.11 芯片。此举将“珍贵”的 I/O 管脚资源用于更重要的功能。

蓝牙、照相机、GPS 和 802.11b 设备有专为它们定义的应用规范。这些应用规范与为 PCI 和 USB 设备定义的类型规范很相像。它们允许任何宿主设备与任意外设“通话”，只要它们都支持应用规范。

SDIO 和 SD 卡规范间的一个重要区别是增加了低速标准。SDIO 卡只需要 SPI 和 1 位 SD 传输模式。低速卡的目标应用是以最小的硬件开支来支持低速 I/O 能力。低速卡支持类似调制解调器、条码扫描仪和 GPS 接收器等应用。对“组合”卡（存储器+SDIO）而言，全速和 4 位操作对卡内存储器和 SDIO 部分都是强制要求的。

18.1.3 SDIO 总线

和 USB 总线类似，SDIO 总线也有两端，其中一端是主机（HOST）端，另一端是设

备 (DEVICE) 端。采用 HOST-DEVICE 这样的设计是为了简化 DEVICE 的设计, 所有的通信都是由 HOST 端发出指令开始的。在 DEVICE 端只要能解析 HOST 的命令, 就可以和 HOST 进行通信。

SDIO 的 HOST 可以连接多个 DEVICE, 如图 18-2 所示。

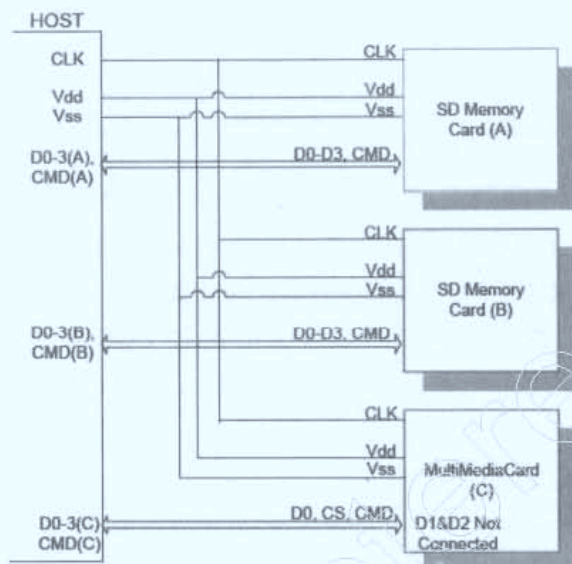


图 18-2 SDIO 总线

其中 SD 总线有如下几种信号：

- (1) CLK 信号：HOST 给 DEVICE 的时钟信号。
- (2) CMD 信号：双向的信号，用于传送命令和反应。
- (3) DAT0-DAT3 信号：四条用于传送的数据线。
- (4) VDD 信号：电源信号。
- (5) VSS1, VSS2：电源地信号。

18.1.4 SDIO 令牌

SDIO 总线上都是 HOST 端发起请求, 然后 DEVICE 端回应请求。其中请求和回应中会携带着数据信息。

(1) 命令 (Command): 用于开始传输的令牌, 是由 HOST 端发往 DEVICE 端的。其中命令是通过 CMD 信号线传送的。

(2) 回应 (Response): 回应是 DEVICE 返回 HOST 的令牌, 作为对命令的回应。回应是通过 CMD 信号线传送的。

(3) 数据 (Data): 数据是双向传送的。可以设置 1 线模式, 也可以设置 4 线模式。数据是通过 DAT0-DAT3 信号线传输的。

每次操作都是由 HOST 在 CMD 线上发起一个 CMD，对于有的 CMD，DEVICE 需要返回一个回应 (Response)，而对于有些 CMD 则不需要进行回应。如图 18-3 所示。

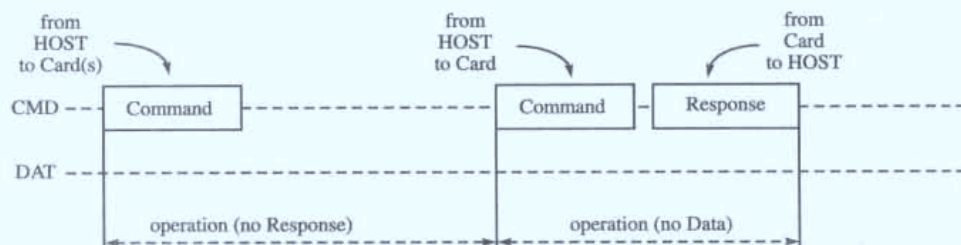


图 18-3 不带数据的 Command 和 Response

对于读命令，首先 HOST 会向 DEVICE 发送命令，紧接着 DEVICE 会返回一个握手信号。此时，当 HOST 收到回应的握手信号后，会将数据放在 4 位的数据线上，在传送数据的同时会跟随着 CRC 校验码。当整个读传送完毕后，HOST 会再次发送一个命令，通知 DEVICE 操作完毕，DEVICE 同时会返回一个响应。整个过程如图 18-4 所示。

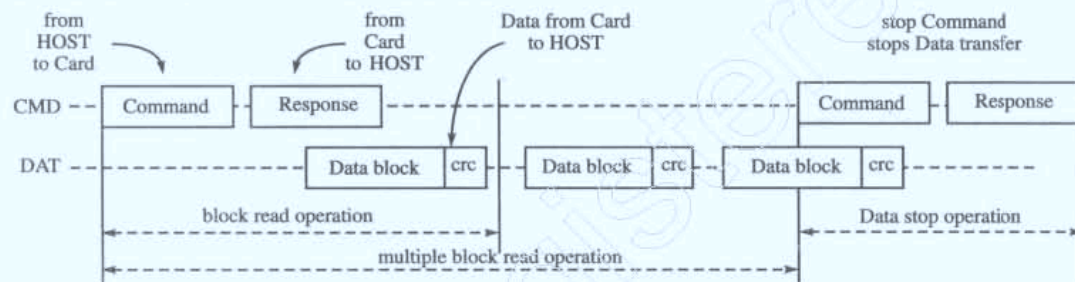


图 18-4 读命令流程图

对于写命令，首先 HOST 会向 DEVICE 发送命令，紧接着 DEVICE 会返回一个握手信号。此时，当 HOST 收到回应的握手信号后，会将数据放在 4 位的数据线上，在传送数据的同时会跟随着 CRC 校验码。当整个读传送完毕后，HOST 会再次发送一个命令，通知 DEVICE 操作完毕，DEVICE 同时会返回一个响应。整个过程如图 18-5 所示。

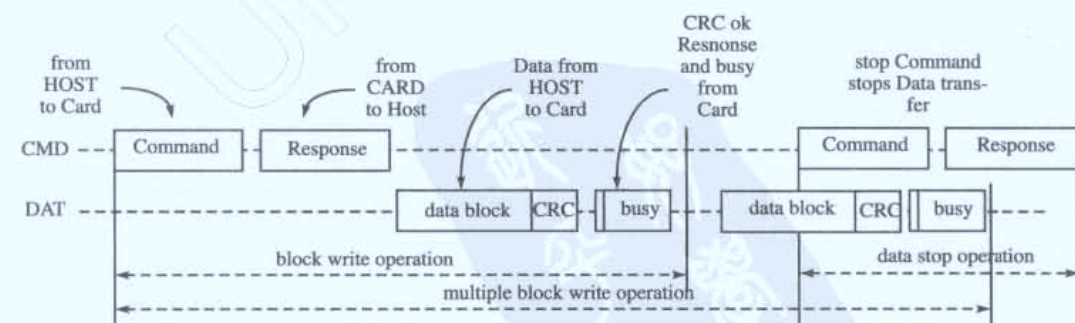


图 18-5 写命令流程图

18.1.5 SDIO 令牌格式

(1) 对于 Command 令牌，它是一个 48 位的令牌。第一位为 0，表示开始位。第二位为 1，代表从 HOST 发往 DEVICE。其后的 38 位为 Command 的内容，不同的数代表不同的 Command。再其后的 7 位为 CRC 校验码。最后一位为 1，代表结束位。Command 令牌的结构如图 18-6 所示。

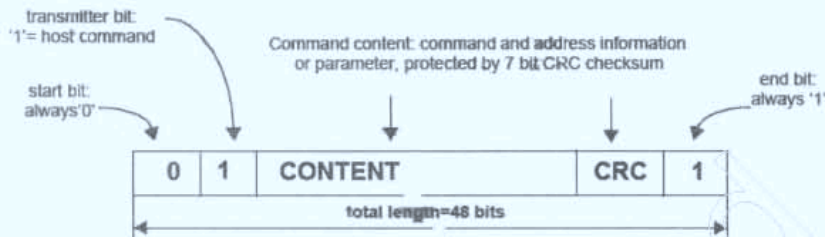


图 18-6 Command 令牌

(2) 对于 Response 令牌，有两种形式。其中一种如 R1、R3、R6 这样，是一个 48 位的令牌。而另外一种如 R2 这样的 Response 令牌，它是一个 136 位的令牌。Command 令牌的结构如图 18-7 所示。

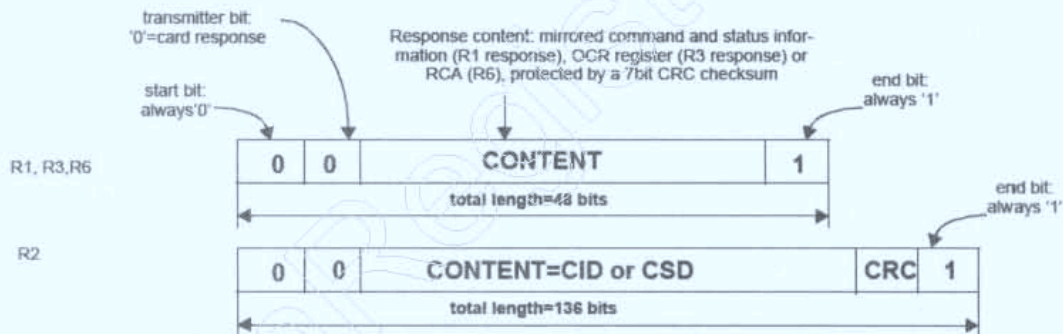


图 18-7 Response 令牌

(3) Data 令牌和前两种不同，前两种令牌是在 CMD 信号线上传送，而 Data 令牌是通过 DAT 线传送的。传送分为两种，一种是 1 位传送，只占用 DAT0 信号线。而另外一种一种是 4 位传送，需要用 DAT0-DAT34 条信号线传送。

当一线传送的时候，第一位为 0，代表开始位。紧接着就是需要传送的数据，其中高位在前，低位在后。当传送完毕后，紧接着是 7 位的 CRC 校验码，最后一位是 1，代表结束位。

当 4 线传送的时候，4 条线的第一位都是 0，代表开始位。然后紧接着是传送的数据。传送的数据按照 4 位并行发送一次，也是高位先传低位后传。传送完数据后，每条 DAT 线接着传送 7 位的 CRC 校验码。最后的一位都是 1，代表结束位。有关一线传送和 4 线传送的结构如图 18-8 所示。

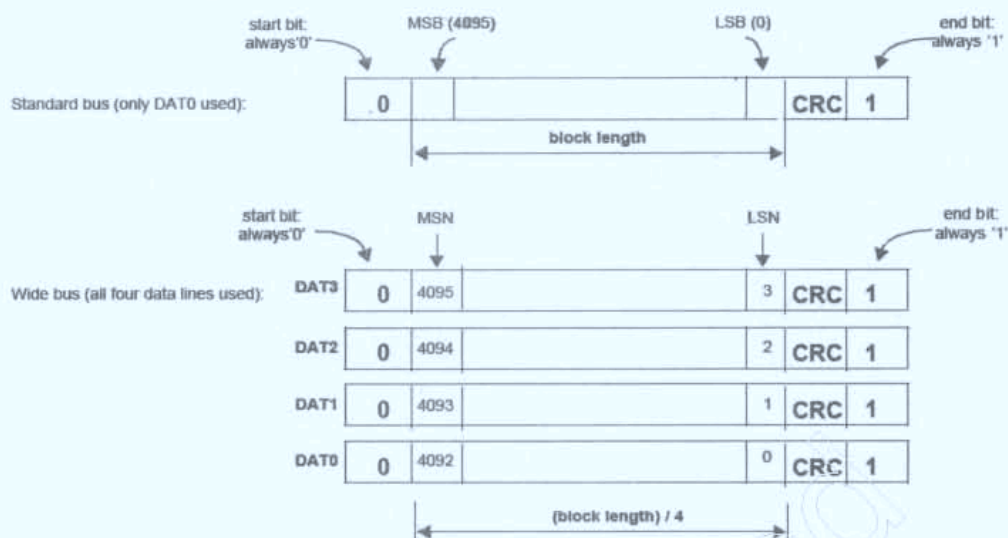


图 18-8 数据令牌

18.1.6 SDIO 的寄存器

SDIO 卡的设备驱动 80% 的任务就是操作 SDIO 卡上是有关的寄存器。SDIO 卡最多允许有 7 个功能 (Function)，一般的 SDIO 卡只有一个功能。每个功能都有一个功能号，它的取值范围是 1~7。每个功能都对应着一个 128K 字节大小的寄存器。功能号之所以取值范围是 1~7，而没有包含 0，是因为功能 0 并不代表真正的功能，而代表 CIA 寄存器组，即 Common I/O Area，这里记录着 SDIO 卡的一些特性，并且可以改写这些寄存器。关于 CIA 寄存器和各个功能寄存器如图 18-9 所示。

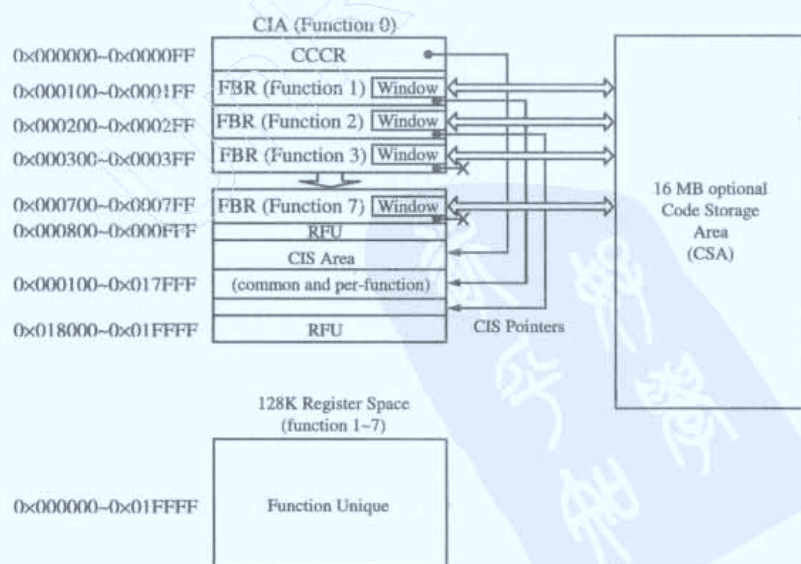


图 18-9 SDIO 寄存器

功能 0 寄存器组，即 CIA 寄存器组。其中 0x000000~0x0000FF 记录的是卡通用控制寄存器 (Card Common Control Registers)，即 CCCR 寄存器。在 CCCR 寄存器中包含了 CCCR 的版本号，对于某个功能是否允许中断等非常重要的信息。有关 CCCR 寄存器，请参见图 18-10 所示。

Address	Register Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x00	CCCR/SDIO Revision	SDIO bit 3	SDIO bit 2	SDIO bit 1	SDIO bit 0	CCCR bit 3	CCCR bit 2	CCCR bit 1	CCCR bit 0
0x01	SD Specification Revision	RFU	RFU	RFU	RFU	SD bit 3	SD bit 2	SD bit 1	SD bit 0
0x02	I/O Enable	IOE7	IOE6	IOE5	IOE4	IOE3	IOE2	IOE1	RFU
0x03	I/O Ready	IOR7	IOR6	IOR5	IOR4	IOR3	IOR2	IOR1	RFU
0x04	Int Enable	IEN7	IEN6	IEN5	IEN4	IEN3	IEN2	IEN1	IENM
0x05	Int Pending	INT7	INT6	INT5	INT4	INT3	INT2	INT1	RFU
0x06	I/O Abort	RFU	RFU	RFU	RFU	RES	AS2	AS1	AS0
0x07	Bus Interface Control	CD Disable	SCSI	ECSI	RFU	RFU	RFU	Bus Width 1	Bus Width 0
0x08	Card Capability	4BLS	LSC	E4MI	S4MI	SBS	SRW	SMB	SDC
0x09-0x0B	Common CIS Pointer	Pointer to card's common Card Information Structure (CIS)							
0x0C	Bus Suspend	RFU	RFU	RFU	RFU	RFU	RFU	BR	BS
0x0D	Function Select	DF	RFU	RFU	RFU	FS3	FS2	FS1	FS0
0x0E	Exec Flags	EX7	EX6	EX5	EX4	EX3	EX2	EX1	EXM
0x0F	Ready Flags	RF7	RF6	RF5	RF4	RF3	RF2	RF1	RFM
0x10-0x11	FN0 Block Size	I/O block size for Function 0							
0x12	Power Control	Reserved for Future Use (RFU)						EMPC	SMPC
0x13	High-Speed	RFU	RFU	RFU	RFU	RFU	RFU	EHS	SHS
0x14-0xEF	RFU	Reserved for Future Use (RFU)							
0xF0-0xFF	Reserved for Vendors	Area Reserved for Vendor Unique Registers							

图 18-10 CCCR 寄存器组

CIA 寄存器组在 0x000N00~0x000NFF，记录着第 N 号功能的功能基本寄存器 (Function Basic Registers)，即 FBR 寄存器组，N 的取值范围为 1~7。其中功能 1 的 FBR 如图 18-11 所示。

Address	7	6	5	4	3	2	1	0
0x100	Function 1 CSA enable	Function 1 supports CSA	RFU	RFU	Function 1 Standard SDIO Function interface code			
0x101	Function 1 Extended standard SDIO Function interface code							
0x102	RFU	RFU	RFU	RFU	RFU	RFU	EPS	SPS
0x103-0x108	Reserved for Future Use (RFU)							
0x109-0x10B	Pointer to Function 1 Card Information Structure (CIS)							
0x10C-0x10E	Pointer to Function 1 Code Storage Area (CSA)							
0x10F	Data access window to Function 1 Code Storage Area (CSA)							
0x110-0x111	I/O block size for Function 1							
0x112-0x1FF	Reserved for Future Use							
0x200-0x7FF	Function 2 to 7 Function Basic Information Registers (FBR)							
0x800-0xFFF	Reserved for Future Use							

图 18-11 FBR 寄存器组

CIA 作为功能 0 的寄存器组，以上已经做了简单的介绍。而关于功能 1~7 的寄存器组是由 SDIO 卡的硬件设计厂商所规定的。作为开发 SDIO 设备驱动的程序员，应该和硬件工程师共同制定某个功能的寄存器组。

18.1.7 CMD52 命令

SDIO 设备为了与 SD 内存卡兼容, SD 卡的所有 Command 和 Response 完全兼容, 同时加入了一些新的 Command 和 Response。例如, 初始化 SD 内存卡使用 ACMD41, 而 SDIO 卡设备则用 CMD5 通知 DEVICE 进行初始化。

但二者最重要的区别是, SDIO 卡比 SD 内存卡多了 CMD52 和 CMD53 命令, 这两个 Command 可以方便地访问某个功能的某个地址的寄存器。

首先介绍 CMD52 命令, CMD52 又称 IO_RW_DIRECT 命令。其命令格式如图 18-12 所示。

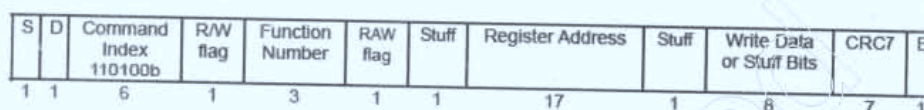


图 18-12 CMD52 格式

首先第一位应该为 0, 表明是起始位。第二位表示传送方向, 这里是 1, 代表方向是 HOST 向 DEVICE 设备传送。其后 6 位是命令索引, 这里是 110100B, 用十进制表示为 52, CMD52 的名字也由此而来。接着是读写标志, 如果是 0 则代表读, 如果是 1 则代表写。

然后是操作的功能号, 用 3 位指示, 可以是 0~7, 其中如果功能号为 0, 则指示为 CCCR 寄存器组, 如果是 1~7 则代表真正的功能号。紧接着的一位是填充位, 没有任何意义。

紧接着是寄存器地址, 用 17 位指示, 由于功能寄存器有 128K 字节, 17 位正好能寻址到这些地址。其后接着又是 1 位填充位。

再接着 8 位的意义是, 如果当前操作为写操作, 则这 8 位为写数据, 如果当前操作是读操作, 则这 8 位为填充位, 无意义。

其后紧接着是 7 位 CRC 校验码, 最后一位是结束位, 应该为 0。

对于 CMD52 的 Response 是 48 位, 如图 18-13 所示。

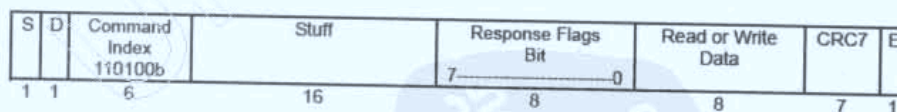


图 18-13 CMD52 的 Response

CMD52 的 Response 的第一位是开始位, 应该是 1。第二位是方向位, 这里是 0, 代表是从 DEVICE 到 HOST 方向。然后是命令索引, 这里是 110100B, 用十进制表示就是 52, 表示此 Response 是针对 CMD52 的。然后是 16 位的填充位, 无意义。

之后是 8 位的返回标志, 接着是 8 位的读写数据, 再之后是 7 位的 CRC 校验码, 最后是结束码, 应该为 0。

总结一下, CMD52 是由 HOST 发往 DEVICE 的, 它必须有 DEVICE 返回来的 Response。CMD52 不需要占用 DAT 线, 读写的数通过 CMD52 或者 Response 来传送。每次 CMD52

只能读或者写一个字节。

18.1.8 CMD53 命令

CMD52 每次只能读写一个字节，因此有 CMD53 对读写进行了扩展，CMD53 允许每次读取多个字节或者多个块 (Block)。CMD53 的结构如图 18-14 所示。

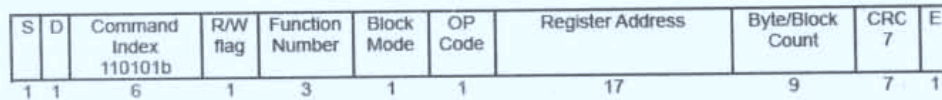


图 18-14 CMD53 结构

第一位是 1，表示开始位。然后是一位方向位，总是 1，代表此令牌是由 HOST 发往 DEVICE 方向。然后是 6 位命令索引，是 110101B，用十进制表示是 53，这也是 CMD53 名字的由来。

然后是 1 位的读写标志，如果是 0 代表读，如果是 1 代表写。然后是三位的功能号，可以是 0~7。然后 1 位的块模式指定。如果是 1 则代表传送是块为单位的，如果是 0 则代表传输是以字节为单位的。

然后是一位的操作位，如果这一位是 0，代表固定位置读写，如果是 1 代表是增量地址读写。例如，对地址 0 固定读 16 个字节，相当于 16 次读取的地址 0。而对地址 0 增量读 16 个字节，相当于读取 0~15 地址的数据。

然后是 17 位的地址寄存器，可以寻址到 128K 字节的地址。然后是 9 位的读写的计数，对于字节读取，读取大小就是这个计数，而对于块读写，读写的大小是计数乘以块的大小。

然后是 7 位的 CRC 校验码，然后是 1 位的计数码，此位为 0。

当读写操作是给予块操作时，块的大小可以通过设置 FBR 中的相关寄存器来设置。

18.2 SDIO 卡驱动开发框架

SDIO 卡的驱动开发，必须在 Windows 2003 DDK 或者后续版本中才有支持，但是编译出来的驱动可以运行在 Windows XP 中。

18.2.1 SDIO Host Controller 驱动

在装有 SDIO Host Controller 的 PC 中（一般是笔记本电脑配有这种 Host Controller），Windows 会为这种 Host Controller 加载驱动，一般是 SDBUS.sys。SDIO Host Controller 是一个 PCI 卡，因此会得到中断号和设备卡内存等资源，如图 18-15 所示。

SDIO Host Controller 会通过其中断线，来得知是否有 SD 内存卡或者 SDIO 卡插入。一旦有设备插入，就会自己创建一个 PDO (Physical Device Object)，并且寻找上层的 FDO。

Windows 驱动开发技术详解

如果是 SD 内存卡，则 FDO 由 sffdisk.sys 提供，不用程序员开发。如果是 SDIO 卡设备，则 FDO 需要这个硬件厂商提供，也就是驱动程序员所要编写的。如图 18-16 所示。



图 18-15 Host Controller 驱动

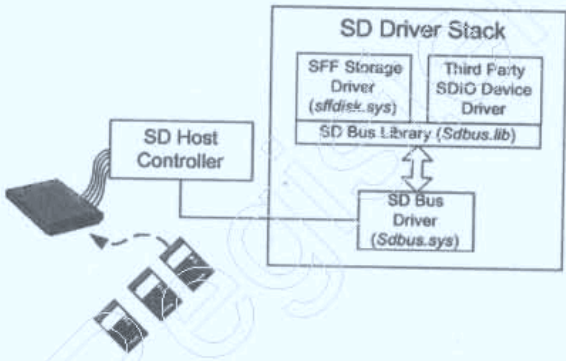


图 18-16 SDIO 设备堆栈

SDBUS.sys 提供的 PDO 提供了大量接口，以方便程序员开发 SDIO 卡设备驱动。例如，用这些接口可以向设备发送 CMD52、CMD53 命令等。其实作为 SDIO 卡驱动，80%的工作量就是用 CMD52 和 CMD53 向 DEVICE 读写数据。

18.2.2 SDIO 卡的初始化

SDIO 卡设备必须打开和初始化 SD 总线接口，用于 Host 与 Device 之间的通信。一般情况下，在 AddDevice 中需要首先调用 SD 库提供的内核函数 SdBusOpenInterface，其功能主要是为了得到 SDBUS_INTERFACE_STANDARD 数据结构。SDBUS_INTERFACE_STANDARD 结构如下：

```
typedef struct _SDBUS_INTERFACE_STANDARD {
    USHORT Size;
    USHORT Version;
    PVOID Context;
}
```

```

PINTERFACE_REFERENCE InterfaceReference;
PINTERFACE_DEREFERENCE InterfaceDereference;
PSDBUS_INITIALIZE_INTERFACE_ROUTINE InitializeInterface;
PSDBUS_ACKNOWLEDGE_INT_ROUTINE AcknowledgeInterrupt;
} SDBUS_INTERFACE_STANDARD, *PSDBUS_INTERFACE_STANDARD;

```

其中 SDBUS_INTERFACE_STANDARD 提供了几个函数地址, 如 InitializeInterface 是用来初始化 SD 总线的, InterfaceDereference 是用来初始化的反过程。另外, 还需要准备 SDBUS_INTERFACE_PARAMETERS 数据结构, 其内容如下:

```

typedef struct _SDBUS_INTERFACE_PARAMETERS {
    USHORT Size;
    USHORT Reserved;
    PDEVICE_OBJECT TargetObject;
    BOOLEAN DeviceGeneratesInterrupts;
    BOOLEAN CallbackAtDpcLevel;
    PSDBUS_CALLBACK_ROUTINE CallbackRoutine;
    PVOID CallbackRoutineContext;
} SDBUS_INTERFACE_PARAMETERS, *PSDBUS_INTERFACE_PARAMETERS;

```

这个数据结构主要是告诉总线驱动是否使用中断, 中断的回调函数, 中断回调函数的接收参数等。初始化 SD 总线的参考代码如下:

```

#001 status = SdBusOpenInterface (pDevExt->UnderlyingPDO,
#002     &pDevExt->BusInterface,
#003     sizeof(SDBUS_INTERFACE_STANDARD),
#004     SDBUS_INTERFACE_VERSION);
#005
#006 if (NT_SUCCESS(status)) {
#007     //设置 SD 总线接口参数
#008     SDBUS_INTERFACE_PARAMETERS interfaceParameters = {0};
#009     interfaceParameters.Size =
#010         sizeof(SDBUS_INTERFACE_PARAMETERS);
#011     //设置目标对象
#012     interfaceParameters.TargetObject =
#013         DeviceExtension->TargetObject;
#014     //允许中断
#015     interfaceParameters.DeviceGeneratesInterrupts = TRUE;
#016     //设置中断回调函数
#017     interfaceParameters.CallbackRoutine = pMyDriverCallback;
#018     status = STATUS_UNSUCCESSFUL;
#019     if (DeviceExtension->BusInterface.InitializeInterface) {
#020         status = (pDevExt->BusInterface.InitializeInterface)
#021             (pDevExt->BusInterface.Context, &interfaceParameters);
#022     }
#023 }

```

此段代码可以在配套光盘中本章的 SDIO_Driver 目录下找到。

18.2.3 中断回调函数

SDIO 设备会向 HOST 发送中断, 这是硬件工程师所设计的, 作为软件工程师主要是接收这个中断, 并进行相应的处理。

接收中断首先要开启中断, 在 CCCR 寄存器组中有 IENx 寄存器, 其中 x 代表不同的

Windows 驱动开发技术详解

Function, 需要将相应的 IENx 置一。另外还有一个寄存器 IENM, 这个寄存器管理是控制所有 Function 的中断, 因此这个寄存器也必须置一。除此之外, 相应的 Function 上的寄存器可能也需要开启, 这要看硬件工程师是如何设计的了。

硬件发送中断, Host Controller 首先得到这个中断, 在 SDBUS_INTERFACE_PARAMETERS 中应该指明是否要驱动处理中断, 并且回调函数是什么。当这些都指定好的时候, Host Controller 会在自己的中断处理函数中调用注册的中断回调函数。

在程序员提供的中断回调函数中需要 SD 总线提供的 AcknowledgeInterrupt 例程, 通知 SD 总线已经处理完中断, 之后返回 SD 总线的中断处理例程。以下是一段演示 SDIO 的中断处理程序:

```
#001 VOID
#002 MyDriverCallback(
#003     IN PVOID CallbackRoutineContext,
#004     IN ULONG InterruptType
#005 )
#006 {
#007     static ULONG i=0;
#008     NTSTATUS status;
#009     //得到设备扩展
#010     PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) CallbackRoutineContext;
#011
#012     KdPrint(("Enter MyDriverCallback\n"));
#013
#014     UCHAR ucRegVal = 0;
#015     //读取 SDIO 数据
#016     status = SdioReadWriteByte(pdx->FunctionalDeviceObject, pdx->
FunctionNumber, &ucRegVal, IFDevice_INT0_STATUS, FALSE);
#017     //判断操作是否成功
#018     if (!NT_SUCCESS(status))
#019     {
#020         KdPrint(("SdioReadWriteByte fail: %x\n", status));
#021         goto done;
#022     }
#023
#024     if (IFDevice_INT0_STATUS_MASK & ucRegVal)
#025     {
#026         KdPrint(("Enter MyDriverCallback %d\n", i++));
#027     }
#028
#029     done:
#030     pdx->InterfaceStandard.AcknowledgeInterrupt(pdx->InterfaceStandard.
Context);
#031 }
```

此段代码可以在配套光盘中本章的 SDIO_Driver 目录下找到。

18.2.4 获得和设置属性

SD 总线驱动提供了一些属性查询和属性设置的功能, 其实查询和设置这些属性就是通过 CMD52 或者 CMD53 查询和设置 DEVICE 上的相应寄存器, 只是 SD 总线驱动进行

了封装。

SD 总线提供了如下属性的支持，其中 SD 总线驱动在 2.0 版本中支持了更多的属性：

```
typedef enum {
    SDP_MEDIA_CHANGECOUNT = 0,
    SDP_MEDIA_STATE,
    SDP_WRITE_PROTECTED,
    SDP_FUNCTION_NUMBER,
    SDP_FUNCTION_TYPE,
    SDP_BUS_DRIVER_VERSION,
    SDP_BUS_WIDTH,
    SDP_BUS_CLOCK,
    SDP_BUS_INTERFACE_CONTROL,
    SDP_HOST_BLOCK_LENGTH,
    SDP_FUNCTION_BLOCK_LENGTH,
    SDP_FNO_BLOCK_LENGTH,
    SDP_FUNCTION_INT_ENABLE,
} SDRUS_PROPERTY;
```

此段代码摘抄自 ntddsd.h。

查询和设置属性主要是构造 SDBUS_REQUEST_PACKET 数据结构，并通过 SdBusSubmitRequest 发送到总线驱动。这里的 SDBUS_REQUEST_PACKET 和 IRP 有着类似的功能。笔者将查询和设置属性进行了封装，以下是其主要代码：

```
#001 NTSTATUS
#002 SdioGetProperty(IN PDEVICE_OBJECT deviceObject,
#003                 IN SDBUS_PROPERTY Property,
#004                 IN PVOID Buffer,
#005                 IN ULONG Length)
#006 {
#007     PDEVICE_EXTENSION pdx;
#008     SDBUS_REQUEST_PACKET sdrp;
#009
#010     pdx = (PDEVICE_EXTENSION)deviceObject->DeviceExtension;
#011     //将内存清零
#012     RtlZeroMemory(&sdrp, sizeof(SDBUS_REQUEST_PACKET));
#013
#014     //设置请求为获取属性
#015     sdrp.RequestFunction = SDRF_GET_PROPERTY;
#016     sdrp.Parameters.GetSetProperty.Property = Property;
#017     sdrp.Parameters.GetSetProperty.Buffer = Buffer;
#018     sdrp.Parameters.GetSetProperty.Length = Length;
#019     //读取属性
#020     return SdBusSubmitRequest(pdx->InterfaceStandard.Context, &sdrp);
#021 }
#022
#023 NTSTATUS
#024 SdioSetProperty(IN PDEVICE_OBJECT deviceObject,
#025                 IN SDBUS_PROPERTY Property,
#026                 IN PVOID Buffer,
#027                 IN ULONG Length)
#028 {
#029     PDEVICE_EXTENSION pdx;
#030     SDBUS_REQUEST_PACKET sdrp;
#031     //获取设备扩展
#032     pdx = (PDEVICE_EXTENSION)deviceObject->DeviceExtension;
```



```

#033    //对内存清零
#034    RtlZeroMemory(&sdrp, sizeof(SDBUS_REQUEST_PACKET));
#035
#036    sdrp.RequestFunction = SDRF_SET_PROPERTY;
#037    sdrp.Parameters.GetSetProperty.Property = Property;
#038    sdrp.Parameters.GetSetProperty.Buffer = Buffer;
#039    sdrp.Parameters.GetSetProperty.Length = Length;
#040    //调用 SD 总线的请求
#041    return SdBusSubmitRequest(pdx->InterfaceStandard.Context, &sdrp);
#042 }

```

此段代码可以在配套光盘中本章的 SDIO_Driver 目录下找到。

18.2.5 CMD52

CMD52 命令和 CMD53 命令在 SDIO 协议一节中已经讲述过了，SD 总线提供了其的封装。主要是设置 SDBUS_REQUEST_PACKET，并通过 SdBusSubmitRequest 向 SD 总线驱动进行调用。

CMD52 每次只能读取或者写入一个字节，并且需要指定操作的 Function 号、地址等信息。笔者将 CMD52 命令进行了函数封装，以下是主要代码：

```

#001 NTSTATUS
#002 SdioReadWriteByte(IN PDEVICE_OBJECT deviceObject,
#003                  IN ULONG Function,
#004                  IN PCHAR Data,
#005                  IN ULONG Address,
#006                  IN BOOLEAN WriteToDevice)
#007 {
#008 {
#009    //得到设备扩展
#010    PDEVICE_EXTENSION pdx =
#011        (PDEVICE_EXTENSION)deviceObject->DeviceExtension;
#012    NTSTATUS status;
#013    SDBUS_REQUEST_PACKET sdrp;
#014    SD_RW_DIRECT_ARGUMENT directArgument;
#015    //设置 CMD52 的读命令
#016    const SD_CMD_DESCRIPTOR ReadIoDirectDesc =
#017        {SDCMD_IO_RW_DIRECT, SDCC_STANDARD, SDTD_READ, SDTT_CMD_ONLY, SDRT_5};
#018    //设置 CMD52 的写命令
#019    const SD_CMD_DESCRIPTOR WriteIoDirectDesc =
#020        {SDCMD_IO_RW_DIRECT, SDCC_STANDARD, SDTD_WRITE, SDTT_CMD_ONLY, SDRT_5};
#021    //确保当前函数运行在分页内存中
#022    PAGED_CODE();
#023
#024    //将内存清零
#025    RtlZeroMemory(&sdrp, sizeof(SDBUS_REQUEST_PACKET));
#026    //设置 SDIO 请求
#027    sdrp.RequestFunction = SDRF_DEVICE_COMMAND;
#028
#029    directArgument.u.AsULONG = 0;
#030    //设置功能号
#031    directArgument.u.bits.Function = Function;
#032    //设置地址

```

```

#033     directArgument.u.bits.Address = Address;
#034     //判断是否是写操作
#035     if (WriteToDevice)
#036     {
#037         //设置为写操作
#038         directArgument.u.bits.WriteToDevice = 1;
#039         //设置传输数据
#040         directArgument.u.bits.Data = *Data;
#041         sdrp.Parameters.DeviceCommand.CmdDesc = WriteIoDirectDesc;
#042     } else {
#043         sdrp.Parameters.DeviceCommand.CmdDesc = ReadIoDirectDesc;
#044     }
#045
#046     sdrp.Parameters.DeviceCommand.Argument = directArgument.u.AsULONG;
#047     //调用 SD 总线驱动
#048     status = SdBusSubmitRequest(pdx->InterfaceStandard.Context, &sdrp);
#049     //判断操作是否成功
#050     if (!NT_SUCCESS(status))
#051     {
#052         KdPrint(("SdioReadWriteByte failed\n"));
#053     }
#054     //判断操作是否成功
#055     if (NT_SUCCESS(status) && !WriteToDevice)
#056     {
#057         *Data = sdrp.ResponseData.AsUCHAR[0];
#058     }
#059     return status;
#060 }

```

此段代码可以在配套光盘中本章的 SDIO_Driver 目录下找到。

18.2.6 CMD53

和 CMD52 命令类似, CMD53 主要是用来 HOST 和 DEVICE 之间进行通信的命令。SD 总线提供了其的封装。主要是设置 SDBUS_REQUEST_PACKET, 并通过 SdBusSubmitRequest 向 SD 总线驱动进行调用。

CMD53 每次可以读取多个字节或者多个 Block, 并且需要指定操作的 Function 号, 地址等信息。笔者将 CMD52 命令进行了函数封装, 以下是主要代码:

```

#001 NTSTATUS
#002 SdioReadWriteBuffer(IN PDEVICE_OBJECT deviceObject,
#003                     IN ULONG Function,
#004                     IN PMDL Mdl,
#005                     IN ULONG Address,
#006                     IN ULONG Length,
#007                     IN BOOLEAN WriteToDevice,
#008                     OUT PULONG BytesRead
#009                     )
#010 {
#011     //获取设备扩展
#012     PDEVICE_EXTENSION pdx =
#013         (PDEVICE_EXTENSION)deviceObject->DeviceExtension;
#014     //构造 SD 总线数据包

```


Windows 驱动开发技术详解

```
#015     SDBUS_REQUEST_PACKET sdrp;
#016     SD_RW_EXTENDED_ARGUMENT extendedArgument;
#017     NTSTATUS              status;
#018         //构造 CMD53 读命令令牌
#019     const SDCMD_DESCRIPTOR ReadIoExtendedDesc =
#020     {SDCMD_IO_RW_EXTENDED, SDCC_STANDARD, SDTD_READ, SDTT_SINGLE_BLOCK,
SDRT_5};
#021         //构造 CMD53 写命令令牌
#022     const SDCMD_DESCRIPTOR WriteIoExtendedDesc =
#023     {SDCMD_IO_RW_EXTENDED, SDCC_STANDARD, SDTD_WRITE, SDTT_SINGLE_BLOCK,
SDRT_5};
#024
#025     PAGED_CODE();
#026         //内存清零
#027     RtlZeroMemory(&sdrp, sizeof(SDBUS_REQUEST_PACKET));
#028
#029     sdrp.RequestFunction = SDRF_DEVICE_COMMAND;
#030         //设置 MDL
#031     sdrp.Parameters.DeviceCommand.Mdl = Mdl;
#032
#033     extendedArgument.u.AsULONG = 0;
#034         //设置功能号
#035     extendedArgument.u.bits.Function = Function;
#036         //设置递增操作模式
#037     extendedArgument.u.bits.OpCode = 1;
#038         //设置块模式
#039     extendedArgument.u.bits.BlockMode = pdx->BlockMode;
#040         //设置 SD 地址
#041     extendedArgument.u.bits.Address = Address;
#042
#043     if (WriteToDevice) {
#044         //设置读写方向
#045         extendedArgument.u.bits.WriteToDevice = 1;
#046         //设置写参数
#047         sdrp.Parameters.DeviceCommand.CmdDesc = WriteIoExtendedDesc;
#048     } else {
#049         //设置读参数
#050         sdrp.Parameters.DeviceCommand.CmdDesc = ReadIoExtendedDesc;
#051     }
#052
#053     if (pdx->BlockMode == 1) {
#054         sdrp.Parameters.DeviceCommand.CmdDesc.TransferType =
SDTT_MULTI_BLOCK_NC_CMD12;
#055     }
#056         //读取参数
#057     sdrp.Parameters.DeviceCommand.Argument = extendedArgument.u.AsULONG;
#058     sdrp.Parameters.DeviceCommand.Length = Length;
#059         //向底层 SD 总线驱动发起请求
#060     status = SdBusSubmitRequest(pdx->InterfaceStandard.Context, &sdrp);
#061     *BytesRead = (ULONG)sdrp.Information;
#062     return status;
#063
#064 }
```

此段代码可以在配套光盘中本章的 SDIO_Driver 目录下找到。

18.3 SDIO 开发实例

SDIO 卡的硬件千差万别，用途也各不相同，所以不能给出一个通用的 SDIO 驱动程序。因此本章的配套光盘给出的实例只是一个粗略的 SDIO 驱动框架，没有针对真正的某一种 SDIO 设备。

首先讲一下 DDK 的版本，对 SDIO 的支持，只有到了 Windows 2003 Sever DDK 以后才有支持，因此使用早期版本的 DDK 不能编译此驱动。但是编译出来的驱动可以安装在 Windows XP 的机器上。其次，要为 SDIO 设备提供一个 INF 文件，其中表示有一段标识固定硬件的一段字符串如下：

```
SD\VID_8096&PID_5510
```

其中 SD 代表 SD 总线上的设备，VID_8096 代表是此设备的 VendorID，PID_5510 代表是 ProductID。SDIO 卡驱动可以归结为以下几个方面：

(1) 硬件初始化：这部分需要在 DriverEntry 中设置各个派遣函数的入口地址，还有 AddDevice 和 Unload 历程。

另外在 AddDevice 中需要设置接口名，调用总线驱动提供的初始化历程。

最后在对 IRP_MN_START_DEVICE 的处理函数中，还可以做硬件上的初始化。

(2) 对中断的处理：当硬件发生中断后，首先得到通知的是总线驱动，这时候总线驱动会调用一个事先注册好了的回调函数。这个回调函数是由 SDIO 驱动提供的，在这个回调函数中，驱动可以进一步通知 Ring3 层的应用程序处理中断。

(3) 通过 CMD52 和 CMD53 访问 SDIO 设备上的各个寄存器，访问具体的寄存器，需要指定是哪个 Function 上的寄存器，寄存器的地址。CMD52 每次可以读写一个字节的寄存器，而 CMD53 可以一次读写多字节或多 Block 的寄存器。

(4) 编写针对此设备驱动的应用程序，这个应用程序是驱动程序的客户端，通过事先定义好的接口，如 ReadFile、WriteFile、DeviceIOControl 等。

本章并没有给出一个具体的 SDIO 卡的驱动，因为各个 SDIO 卡的驱动千差万别，在本章的相关配套光盘中，读者可以找到 SDIO_Driver，这是一个 SDIO 卡驱动的框架。还有一个 Test 程序，这是一个应用程序，作为 SDIO_Driver 的客户端。读者可以根据自己的需要，更改以上代码，并且会发现，开发 SDIO 驱动还是比较简单的。

18.4 小结

本章首先介绍了 SDIO 协议，讲述了 SD 内存卡和 SDIO 卡的兼容问题。然后介绍了

Windows 驱动开发技术详解

SDIO 协议中的发送命令、回应命令、传送数据等相关协议。随后，本章又介绍了 Windows 中，DDK 提供的对 SDIO 卡设备的支持。然后介绍了如何利用总线驱动，使 SDIO 设备初始化，接收中断，发送和接收数据等操作。

另外，本章的配套光盘给出了一个粗略的 SDIO 设备卡的开发框架，读者可以根据自己的设备进行简单改写，就可以完成对 SDIO 卡的驱动开发。

第 19 章 虚拟串口设备驱动

本章讲述 DDK 的开发串口驱动框架，并给出一个虚拟串口驱动的实例。虚拟串口驱动有很多实际用处，例如，可以开发一个虚拟串口，将读写请求传递给 USB 驱动，这样就可以利用现成的串口调试工具向 USB 设备读取了。

19.1 串口简介

RS-232、RS-422 与 RS-485 都是串行数据接口标准，最初都是由电子工业协会（EIA）制定并发布的，RS-232 在 1962 年发布，命名为 EIA-232-E，作为工业标准，以保证不同厂家产品之间的兼容。RS-422 由 RS-232 发展而来，它是为弥补 RS-232 之不足而提出的。为改进 RS-232 通信距离短、速率低的缺点，RS-422 定义了一种平衡通信接口，将传输速率提高到 10Mb/s，传输距离延长到 4000 英尺（速率低于 100kb/s 时），并允许在一条平衡总线上连接最多 10 个接收器。RS-422 是一种单机发送、多机接收的单向、平衡传输规范，被命名为 TIA/EIA-422-A 标准。为扩展应用范围，EIA 又于 1983 年在 RS-422 基础上制定了 RS-485 标准，增加了多点、双向通信能力，即允许多个发送器连接到同一条总线上，同时增加了发送器的驱动能力和冲突保护特性，扩展了总线共模范围，后命名为 TIA/EIA-485-A 标准。由于 EIA 提出的建议标准都是以 RS 作为前缀，所以在通信工业领域，仍然习惯将上述标准以 RS 作前缀称谓。

RS-232、RS-422 与 RS-485 标准只对接口的电气特性做出规定，而不涉及接插件、电缆或协议，在此基础上用户可以建立自己的高层通信协议。因此在视频界的应用，许多厂家都建立了一套高层通信协议，或公开或厂家独家使用。如录像机厂家中的 Sony 与松下对录像机的 RS-422 控制协议是有差异的，视频服务器上的控制协议则更多了，如 Louth、Odetis 协议是公开的，而 ProLINK 则是基于 Profile 上的。

目前 RS-232 是 PC 机与通信工业中应用最广泛的一种串行接口。RS-232 被定义为一

Windows 驱动开发技术详解

种在低速率串行通信中增加通信距离的单端标准。RS-232 采取不平衡传输方式,即所谓单端通信。

收、发端的数据信号是相对于信号地而言的,如从 DTE 设备发出的数据在使用 DB25 连接器时是 2 脚相对 7 脚(信号地)的电平。典型的 RS-232 信号在正负电平之间摆动,在发送数据时,发送端驱动器输出正电平在+5V~+15V,负电平在-5V~-15V 电平。当无数据传输时,线上为 TTL,从开始传送数据到结束,线上电平从 TTL 电平到 RS-232 电平再返回 TTL 电平。接收器典型的工作电平在+3V~+12V 与-3V~-12V。由于发送电平与接收电平的差仅为 2V~3V 左右,所以其共模抑制能力差,再加上双绞线上的分布电容,其传送距离最大为约 15m,最高速率为 20kb/s。RS-232 是为点对点(即只用一对收、发设备)通信而设计的,其驱动器负载为 3k Ω ~7k Ω 。所以 RS-232 适合本地设备之间的通信。

19.2 DDK 串口开发框架

DDK 对串口驱动提供了专门接口。只要编写的驱动满足这些接口,并按照串口标准的命名方法,不管是真实的串口设备,还是虚拟的设备,Windows 操作系统都会认为这个设备是一个标准的串口设备。用标准的串口调试工具都可以与这个设备进行通信。

19.2.1 串口驱动的入口函数

本章的实例程序是在 HelloWDM 驱动的基础之上修改而来,入口函数依然是 DriverEntry。在 DriverEntry 函数中指定各种 IRP 的派遣函数,以及 AddDevice 例程、卸载例程等。

```
#001 extern "C" NTSTATUS DriverEntry(IN PDRIVER_OBJECT pDriverObject,  
#002                                IN PUNICODE_STRING pRegistryPath)  
#003 {  
#004     KdPrint(("Enter DriverEntry\n"));  
#005     //设置 AddDevice 例程  
#006     pDriverObject->DriverExtension->AddDevice = HelloWDMAddDevice;  
#007     //设置 IRP 派遣函数  
#008     pDriverObject->MajorFunction[IRP_MJ_PNP] = HelloWDMNp;  
#009     pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = HelloWMDDispatch  
Controlp;  
#010     pDriverObject->MajorFunction[IRP_MJ_CREATE] = HelloWDMCreate;  
#011     pDriverObject->MajorFunction[IRP_MJ_CLOSE] = HelloWDMClose;  
#012     pDriverObject->MajorFunction[IRP_MJ_READ] = HelloWDMRead;  
#013     pDriverObject->MajorFunction[IRP_MJ_WRITE] = HelloWDMWrite;  
#014     //设置卸载例程  
#015     pDriverObject->DriverUnload = HelloWDMUnload;  
#016  
#017     KdPrint(("Leave DriverEntry\n"));  
#018     return STATUS_SUCCESS;  
#019 }
```

这段代码可以在配套光盘中本章的 Virtual_COM 目录下找到。

其中在 AddDevice 例程中,需要创建设备对象,这些都是和以前的 HelloWDM 驱动程序类似。在创建完设备对象后,需要将设备对象指定一个符号链接,该符号链接必须是 COM 开头,并接一个数字,如本例就采用了 COM7。因为 COM1 和 COM2 在有些计算机中有时会被占用,因此,当该设备对象在指定符号链接时,应该避免采用这些名称。

符号链接的作用主要是让客户程序能够识别出串口驱动程序。

```
#001 NTSTATUS HelloWDMAddDevice(IN PDRIVER_OBJECT DriverObject,
#002                               IN PDEVICE_OBJECT PhysicalDeviceObject)
#003 {
#004     PAGED_CODE();
#005     KdPrint(("Enter HelloWDMAddDevice\n"));
#006
#007     NTSTATUS status;
#008     PDEVICE_OBJECT fdo;
#009     UNICODE_STRING devName;
#010     //初始化设备名字符串
#011     RtlInitUnicodeString(&devName,L"\\Device\\MyWDMDevice");
#012     //创建设备对象
#013     status = IoCreateDevice(
#014         DriverObject,
#015         sizeof(DEVICE_EXTENSION),
#016         &(UNICODE_STRING)devName,
#017         FILE_DEVICE_UNKNOWN,
#018         0,
#019         FALSE,
#020         &fdo);
#021     //判断是否成功创建设备对象
#022     if( !NT_SUCCESS(status))
#023         return status;
#024     //获取设备扩展
#025     PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION)fdo->DeviceExtension;
#026     //获取 FDO
#027     pdx->fdo = fdo;
#028     //将设备对象挂载在设备堆栈上
#029     pdx->NextStackDevice = IoAttachDeviceToDeviceStack(fdo, PhysicalDeviceObject);
#030     //初始化 UNICODE 字符串
#031     UNICODE_STRING symLinkName;
#032     RtlInitUnicodeString(&symLinkName,L"\\DosDevices\\COM7");
#033     //记录设备名
#034     pdx->ustrDeviceName = devName;
#035     //记录符号链接
#036     pdx->ustrSymLinkName = symLinkName;
#037     //创建符号链接
#038     status = IoCreateSymbolicLink(&(UNICODE_STRING)symLinkName,&(UNICODE_
STRING)devName);
#039     //判断操作是否成功
#040     if( !NT_SUCCESS(status))
#041     {
#042         //删除符号链接
#043         IoDeleteSymbolicLink(&pdx->ustrSymLinkName);
#044         status = IoCreateSymbolicLink(&symLinkName,&devName);
#045         //创建符号链接
#046         if( !NT_SUCCESS(status))
#047     {
```


Windows 驱动开发技术详解

```
#048         return status;
#049     }
#050 }
#051 // 设置为缓冲区设备
#052 fdo->Flags |= DO_BUFFERED_IO | DO_POWER_PAGABLE;
#053 fdo->Flags &= ~DO_DEVICE_INITIALIZING;
#054
#055 KdPrint(("Leave HelloWDMAddDevice\n"));
#056 return STATUS_SUCCESS;
#057 }
```

此段代码可以在配套光盘中本章的 Virtual_COM 目录下找到。

在创建完符号链接后，还不能保证应用程序能找出这个虚拟的串口设备，还需要进一步修改注册表。具体位置是 HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\SERIALCOMM，可以在这里加入新项目。本例的项目名是 MyWDMDevice，类型为 REG_SZ，内容是 COM7。这里为了简便，没有将修改注册表写入驱动程序，而是直接手动修改注册表。读者可以将这一步在 AddDevice 例程中进行实现。修改注册表如图 19-1 所示。

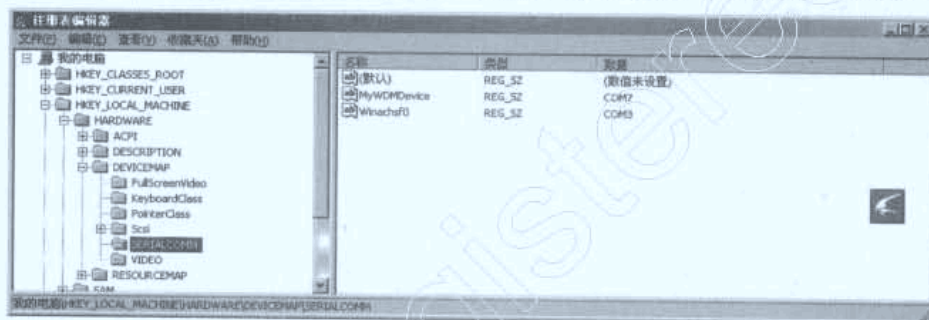


图 19-1 修改注册表

在上述步骤后，即在 AddDevice 例程中创建 COM7 的符号链接，并且在注册表中进行相应设置，系统就会认为有这个串口驱动。用任何一个串口调试软件，都可以枚举到该串口。笔者在本章中将使用一款常用的串口调试工具 sscom32，读者可以在网上下载到这个软件。用这个软件可以枚举到 COM7 这个虚拟串口设备，如图 19-2 所示。

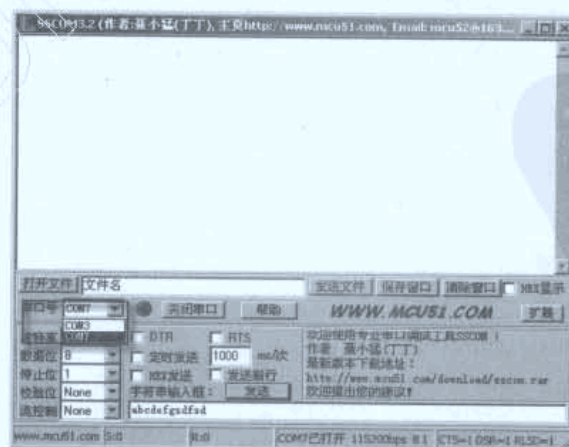


图 19-2 串口调试软件

19.2.2 应用程序与串口驱动通信

其实对于一个真实的串口驱动,或者这里介绍的虚拟串口驱动,都需要遵循一组接口。这组接口由微软事先定义好了。只要符合这组接口,Windows 就会认为这是一个串口设备。这里所指的接口就是应用程序发的 IO 控制码和读写命令,因此对于串口驱动只要对这些 IRP 的派遣函数编写适当,就能实现一个串口驱动。

首先用 IRPTrace 看一下,需要对哪些 IRP 进行处理。笔者加载本章已经介绍的虚拟串口驱动,并用 IRPTrace 拦截其 IRP 处理信息。在打开串口工具软件后,会发现 IRPTrace 立刻跟踪到若干个 IO 控制码,如图 19-3 所示。

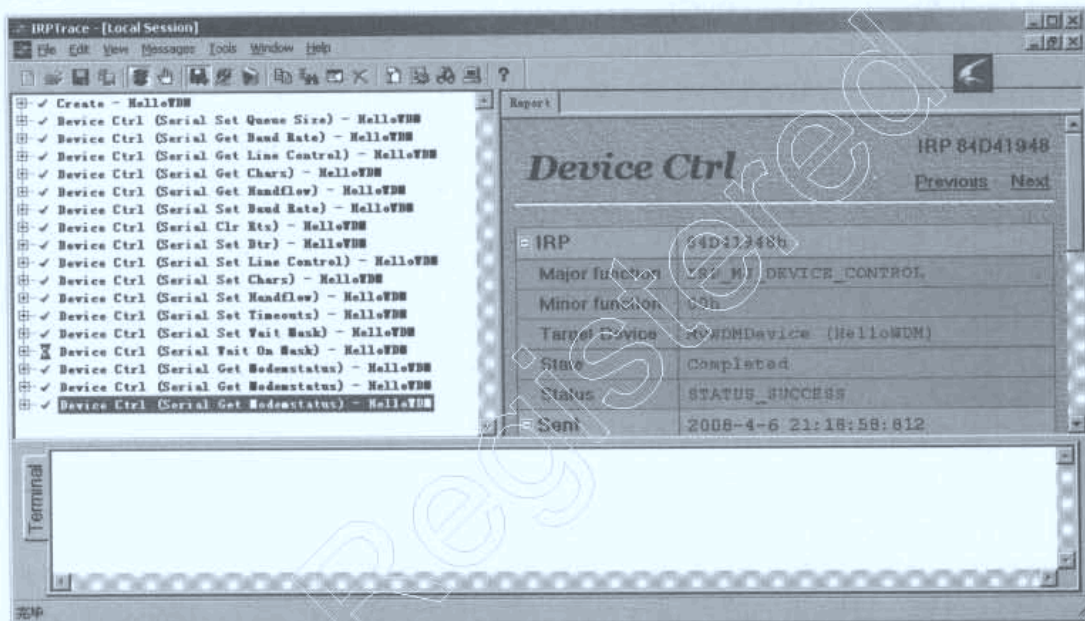


图 19-3 IRPTrace

下面依次解释这些 IO 控制码,理解这些 IO 控制码,并处理好这些控制码,是编写串口驱动的核心。关于这些 IO 控制码在 `ntddser.h` 文件中,都有相应的定义,并且还有相应的数据结构定义。

(1) `IOCTL_SERIAL_SET_QUEUE_SIZE`

这个控制码是应用程序向驱动请求设置串口驱动内部的缓冲区大小,它是向驱动传递 `SERIAL_QUEUE_SIZE` 数据结构来进行设置的。对于虚拟串口驱动来说,这是不需要关心的。用 IRPTrace 可以看出,串口调试工具会向驱动发送的请求是 0x400 大小的缓冲区大小。

(2) `IOCTL_SERIAL_GET_BAUD_RATE`

串口调试工具会接着向驱动发送 `IOCTL_SERIAL_GET_BAUD_RATE` 命令,这主要是询问驱动这个设备的波特率。驱动应该回应应用程序 `SERIAL_BAUD_RATE` 数据结构,来通知波特率的数值。

(3) IOCTL_SERIAL_GET_LINE_CONTROL

串口调试工具会接着向驱动发送 IOCTL_SERIAL_GET_LINE_CONTROL 命令, 这主要是为了返回串口的行控制信息, 行控制信息用 SERIAL_LINE_CONTROL 数据结构表示。

```
typedef struct _SERIAL_LINE_CONTROL {
    UCHAR StopBits;
    UCHAR Parity;
    UCHAR WordLength;
} SERIAL_LINE_CONTROL, *PSERIAL_LINE_CONTROL;
```

其中 StopBits 是停止位, 可以是 STOP_BIT_1、STOP_BITS_1_5、STOP_BITS_2 等取值。Parity 代表校验位, 可以是 NO_PARITY、ODD_PARITY、EVEN_PARITY、MARK_PARITY、SPACE_PARITY。WordLength 是数据位, 可以是 5、6、7、8。

(4) IOCTL_SERIAL_GET_CHARS

串口调试工具会接着向驱动发送 IOCTL_SERIAL_GET_CHARS 命令, 这个命令是应用程序向驱动请求特殊字符, 用来与控制信号握手, 用数据结构 SERIAL_CHARS 表示。

```
typedef struct _SERIAL_CHARS {
    UCHAR EofChar;
    UCHAR ErrorChar;
    UCHAR BreakChar;
    UCHAR EventChar;
    UCHAR XonChar;
    UCHAR XoffChar;
} SERIAL_CHARS, *PSERIAL_CHARS;
```

其中 EofChar 代表是否是传送结束、ErrorChar 代表是否传送中有错误、BreakChar 代表是否传送有停止等。

(5) IOCTL_SERIAL_GET_HANDFLOW

串口调试工具会接着向驱动发送 IOCTL_SERIAL_GET_HANDFLOW 命令, 这个命令是负责向驱动程序获得串口驱动的握手信号, 握手信号用 SERIAL_HANDFLOW 数据结构表示。

```
typedef struct _SERIAL_HANDFLOW {
    ULONG ControlHandShake;
    ULONG FlowReplace;
    LONG XonLimit;
    LONG XoffLimit;
} SERIAL_HANDFLOW, *PSERIAL_HANDFLOW;
```

(6) IOCTL_SERIAL_SET_WAIT_MASK

串口调试工具会接着向驱动发送 IOCTL_SERIAL_SET_WAIT_MASK 命令, 这个命令主要是设置串口驱动的某些事件发生时, 需要向应用程序通知。这些事件包括以下几种事件:

```
#define SERIAL_EV_RXCHAR      0x0001
#define SERIAL_EV_RXFLAG     0x0002
#define SERIAL_EV_TXEMPTY    0x0004
#define SERIAL_EV_CTS        0x0008
```

```

#define SERIAL_EV_DSR          0x0010
#define SERIAL_EV_RLSD        0x0020
#define SERIAL_EV_BREAK       0x0040
#define SERIAL_EV_ERR         0x0080
#define SERIAL_EV_RING        0x0100
#define SERIAL_EV_PERR        0x0200
#define SERIAL_EV_RX80FULL    0x0400
#define SERIAL_EV_EVENT1      0x0800
#define SERIAL_EV_EVENT2      0x1000

```

以上这些宏都是摘抄自 `ntddser.h`，每个宏基本都是自解释型的，读者可以得知其具体含义。

(7) IOCTL_SERIAL_WAIT_ON_MASK

这个 IO 控制码是最最重要的一个，当串口调试工具通过前面几个 IO 控制码初始化好后，就会发送这个请求。在驱动程序中，应该阻塞在那里，即返回 `PENDING` 状态，而不是完成这个 IRP。当 `IOCTL_SERIAL_SET_WAIT_MASK` 设置的事件中的一项发生时，阻塞状态改为完成，并通知应用程序究竟是哪种事件发生了。

本章介绍的虚拟串口驱动，就是一个简单的读写回传工作，即向虚拟串口发送写命令然后回写到这个设备上，通过读命令显示出来。在串口调试器中可以看到敲击键盘的字符，会在串口调试软件中显示出来。

19.2.3 写的实现

串口驱动除了需要完成处理 IO 控制码外，还需要对读写 IRP 进行处理。一般情况下，作为应用程序的串口调试工具会开启多个线程，其中主线程负责与串口驱动初始化的 IO 控制码通信，另外也负责应用程序的界面程序。

另外一个很重要的线程就是发送 `IOCTL_SERIAL_WAIT_ON_MASK` 请求，对于没有数据传输的情况下，这个 IO 控制码请求会 `PENDING` 在那里，即阻塞。当有传送的请求时，相应的事件被触发，刚才因为 `IOCTL_SERIAL_WAIT_ON_MASK` 的 IRP 被阻塞的线程会得以继续运行，如果应用程序得知该事件是被写入了一个字符，会去发出一个读请求，对于驱动则就是读的 IRP。如此循环过程，从而实现了一个虚拟摄像头回写的例子。

在对于写 IRP 的派遣函数中，主要是将写的数据存储和设备扩展中，以便以后读的时候将这些内容返回应用程序。另外一个很重要的内容，就是使阻塞的 IO 控制苏醒过来。在本例中调用 `DriverCheckEvent` 函数，该函数将阻塞的 IRP 完成，使应用程序的线程得以继续运行。并且这个线程还知道了 `SERIAL_EV_RXCHAR` 和 `SERIAL_EV_RX80FULL` 事件的到来，从而发起一个读请求，传送到驱动中就是读 IRP。

```

#001 NTSTATUS HelloWDMWrite(IN PDEVICE_OBJECT fdo,
#002                          IN PIRP Irp)
#003 {
#004     KdPrint(("HelloWDMWrite\n"));
#005     NTSTATUS ntStatus = STATUS_SUCCESS; // Assume success
#006     //获取设备扩展

```



```

#007 PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION)fdo->DeviceExtension;
#008 //获得当前 IO 堆栈
#009 PIO_STACK_LOCATION irpSp = IoGetCurrentIrpStackLocation( Irp );
#010 //获取当前 IO 堆栈的操作字节数
#011 ULONG DataLen = irpSp->Parameters.Write.Length;
#012 //从 IRP 的缓冲区中得到数据
#013 PCHAR pData = (PCHAR)Irp->AssociatedIrp.SystemBuffer;
#014 KIRQL OldIrql;
#015 PIRP pOldReadIrp = NULL;
#016 PDRIVER_CANCEL pOldCancelRoutine;
#017 //设置 IRP 的操作字节数
#018 Irp->IoStatus.Information = 0;
#019 ntStatus = STATUS_SUCCESS;
#020 if (DataLen == 0)
#021 {
#022     ntStatus = STATUS_SUCCESS;
#023 }else if (DataLen > COMBUFLEN)
#024 {
#025     ntStatus = STATUS_INVALID_PARAMETER;
#026 }
#027 else
#028 {
#029     //获取自选锁
#030     KeAcquireSpinLock(&pdx->WriteSpinLock, &OldIrql);
#031     //复制内存块
#032     RtlCopyMemory(pdx->Buffer, pData, DataLen);
#033     pdx->uReadWrite = DataLen;
#034     if (pdx->pReadIrp != NULL)
#035     {
#036         //记录 IRP
#037         pOldReadIrp = pdx->pReadIrp;
#038         //设置取消函数
#039         pOldCancelRoutine = IoSetCancelRoutine(pOldReadIrp, NULL);
#040         if (pOldCancelRoutine != NULL)
#041         {
#042             pOldReadIrp->IoStatus.Information = 0;
#043             pOldReadIrp->IoStatus.Status = STATUS_SUCCESS;
#044             pdx->pReadIrp = NULL;
#045         }
#046         else
#047         {
#048             pOldReadIrp = NULL;
#049         }
#050     }
#051     //检测事件
#052     DriverCheckEvent(pdx, SERIAL_EV_RXCHAR | SERIAL_EV_RX80FULL);
#053     //释放自选锁
#054     KeReleaseSpinLock(&pdx->WriteSpinLock, OldIrql);
#055     if (pOldReadIrp != NULL)
#056         IoCompleteRequest(pOldReadIrp, IO_NO_INCREMENT);
#057 }
#058 //设置 IRP 的完成状态
#059 Irp->IoStatus.Status = ntStatus;
#060 //设置 IRP 的操作字节数
#061 Irp->IoStatus.Information = DataLen;
#062 //结束 IRP 请求
#063 IoCompleteRequest( Irp, IO_NO_INCREMENT );

```

```
#064     return ntStatus;
#065 }
```

此段代码可以在配套光盘中本章的 Virtual_COM 目录下找到。

19.2.4 读的实现

对于虚拟串口的读的工作，就变得相对简单。因为写 IRP 会负责通知让阻塞的线程继续运行，并且通知是何种事件的来临。串口调试软件得知 SERIAL_EV_RXCHAR 这个事件，因此发起了读请求。在驱动中，就是进入读 IRP 的派遣函数。

在该派遣函数中，负责将存储在设备扩展中的数据通过 IRP 传送到应用程序。同时，还需要做一些同步处理。

```
#001 NTSTATUS HelloWDMRead(IN PDEVICE_OBJECT fdo,
#002                        IN PIRP Irp)
#003 {
#004     KdPrint(("HelloWDMRead\n"));
#005     //设置派遣函数的返回状态
#006     NTSTATUS ntStatus = STATUS_SUCCESS;
#007     //获得设备扩展
#008     PDEVICE_EXTENSION pExtension = (PDEVICE_EXTENSION)fdo->DeviceExtension;
#009     //获得当前 IO 堆栈
#010     PIO_STACK_LOCATION irpSp = IoGetCurrentIrpStackLocation( Irp );
#011     //获得此次操作的请求的字节数
#012     ULONG BufLen = irpSp->Parameters.Read.Length;
#013     //获得 IRP 的缓冲区
#014     PCHAR pBuf = (PCHAR)Irp->AssociatedIrp.SystemBuffer;
#015     KIRQL OldIrql;
#016     PDRIVER_CANCEL pOldCancelRoutine;
#017     Irp->IoStatus.Information = 0;
#018     if (BufLen == 0)
#019     {
#020         ntStatus = STATUS_SUCCESS;
#021     }
#022     else
#023     {
#024         //获取自旋锁
#025         KeAcquireSpinLock(&pExtension->WriteSpinLock, &OldIrql);
#026         //内存复制
#027         RtlCopyMemory(pBuf, pExtension->Buffer, BufLen);
#028         //设置 IRP 的操作字节数
#029         Irp->IoStatus.Information = BufLen;
#030         if (BufLen==0 && pExtension->pReadIrp==NULL) // nothing, store
#031         {
#032             //将读 IRP 保持
#033             pExtension->pReadIrp = Irp;
#034             //将 IRP 挂起
#035             Irp->IoStatus.Status = ntStatus = STATUS_PENDING;
#036             //设置取消函数
#037             IoSetCancelRoutine(Irp, DriverCancelCurrentReadIrp);
#038             if (Irp->Cancel)
#039             {
#040                 //重新设置取消函数
```



```
#041         pOldCancelRoutine = IoSetCancelRoutine(Irp, NULL);
#042         if (pOldCancelRoutine != NULL)
#043         {
#044             Irp->IoStatus.Status = ntStatus = STATUS_CANCELLED;
#045             pExtension->pReadIrp = NULL;
#046         }
#047         else
#048         {
#049             //标记 IRP 挂起
#050             IoMarkIrpPending(Irp);
#051         }
#052     }
#053     else
#054     {
#055         IoMarkIrpPending(Irp);
#056     }
#057 }
#058 KeReleaseSpinLock(&pExtension->WriteSpinLock, OldIrql);
#059 }
#060 Irp->IoStatus.Status = ntStatus;
#061 if (ntStatus != STATUS_PENDING)
#062     //结束 IRP 请求
#063     IoCompleteRequest( Irp, IO_NO_INCREMENT );
#064 return ntStatus;
#065 }
```

此段代码可以在配套光盘中本章的 Virtual_COM 目录下找到。

19.3 小结

本章介绍了串口开发的框架模型，在串口的 AddDevice 例程中需要暴露出一个串口的符号连接，另外在相应的注册表中需要进行设置。在串口与应用程序的通信中，主要是一组 DDK 定义的 IO 控制码，这些 IO 控制码负责由应用程序向驱动发出请求。除此之外，串口调试驱动程序需要提供读写 IRP 的派遣函数。

第 20 章 摄像头设备驱动程序

摄像头驱动程序的主要目的是通过硬件捕捉视频信号。微软公司提供了一套视频驱动的接口，可以满足这个接口的视频驱动程序。第三方厂商开发的软件，如 QQ 和 MSN 等软件，都可以通过这个接口读取摄像头采集的数据。由于硬件设计各不相同，本章将主要介绍虚拟的摄像头驱动。读者可以根据自己的需要，结合硬件设计出符合这个接口的 USB 或 PCI 摄像头驱动程序。

20.1 WDM 摄像头驱动框架

WDM 摄像头驱动分为两部分，一部分是类驱动程序，一部分是小驱动程序。类驱动由微软提供，而小驱动程序是由程序员提供的。本节介绍类驱动程序和小程序驱动的概念。并介绍如何编写摄像头的小驱动程序。

20.1.1 类驱动与小驱动

在介绍摄像头驱动之前，先为读者介绍一下类驱动（Class Driver）和小驱动（Mini Driver）的概念。

Class Driver 一般是由微软公司提供的，它为一类设备设计了标准接口，如摄像头驱动。各种摄像头虽然在硬件设计上不尽相同，但是为实现标准化，微软规范了接口标准。微软提供的每种类别的 Class Driver 就会封装这些接口。

还有一种驱动叫 Mini Driver。它是配合 Class Driver 针对不同硬件设备的，不同硬件会有不同的 Mini Driver。对于摄像头厂商来说，需要编写这种 Mini Driver。

微软之所以如此设计，就是为了将接口规范化。另外，摄像头厂商开发驱动程序也变得简单。一些通用的代码，被封装到 Class Driver 中。只在一些特殊的地方需要被写入 Mini Driver 中。因此驱动程序员的任务被大大减轻了。

在 WDM 驱动中, Mini Driver 首先将自己注册给 Class Driver, Class Driver 拥有设备对象。而 Mini Driver 不用创建设备对象, 利用 Class Driver 中的设备对象进行系统调用。

对于本章介绍的虚拟摄像头来说, 这里的 Class Driver 对应的是微软提供的。而 Mini Driver 是程序员所需要编写的。Class Driver-Mini Driver 这种配合的开发, 与先前介绍的单一 WDM 框架略有不同, 本章将一一进行介绍。

20.1.2 摄像头的类驱动与小驱动

摄像头采集的数据都是纯粹的数据文件, 然后源源不断地传送给 PC。因此得名 Stream (流设备)。编写摄像头驱动程序就是编写流设备的 Mini Driver, 但需要首先了解流设备的 Class Driver 与 Mini Driver 之间是如何配合的。

流设备的类驱动主要是控制请求。这需要通过调用小驱动的适配器来访问具体的硬件。在流设备的类驱动和小驱动都初始化以后, 小驱动需要被动地等待类驱动来调用。类驱动用 stream request block (SRB) 向小驱动发送标准的请求, 小驱动通过解析 SRB 后回答 Class Driver 的请求。

SRB 可以传送命令和数据。SRB 用数据结构 HW_STREAM_REQUEST_BLOCK 代表。下面分析类驱动程序和小驱动之间是如何初始化的。

(1) 当设备接入插口时, 即插即用管理器能够侦测到新的设备, 进而调用 Mini Driver 的 DriverEntry 例程。

(2) Mini Driver 在自己的 DriverEntry 中填充 HW_INITIALIZATION_DATA, 然后返回 StreamClassRegisterMinidriver。

(3) 在类驱动中初始化一个 SRB_INITIALIZE_DEVICE 类型的 SRB, 这个 SRB 中的 CommandData.ConfigInfo 记录着摄像头硬件设备的相关信息。

对于虚拟摄像头来说, 这里没有太多的可用信息。这个 SRB 会被传递给小驱动程序, 小驱动程序通过此或者一些硬件信息结束并且返回, 并通知类驱动, 告知小驱动已经初始化完毕。

(4) 类驱动程序会再发一个 SRB_GET_STREAM_INFO 类型的 SRB, 向小驱动程序询问 HW_STREAM_HEADER 数据信息和 HW_STREAM_INFORMATION 等信息。这些信息包含了摄像头驱动提供的视频图像大小、图像格式等信息。

(5) 类驱动会向小驱动再发一个 HW_STREAM_INFORMATION 的 SRB, 这次请求后, 小驱动应该做完所有的初始化操作, 并返回给类驱动程序。

20.1.3 编写小驱动程序

类驱动程序的目标是与操作系统交互, 其中包括处理同步、提供标准接口。而小驱动程序由类驱动调用, 主要负责具体硬件相关的操作。

程序员编写的小驱动程序会向类驱动注册一些回调函数。类驱动作为主程序，会在适当的时机调用小驱动程序提供的这些回调函数。

每个流的小驱动程序都会支持一种或多种数据格式。例如 DVD 播放器就支持一种声音流数据以及一种视频流数据。每种数据流都是从一个叫做 PIN 的接口输出的。

每种小驱动程序必须支持以下几种回调函数：

(1) StrMiniCancelPacket: 对 HW_STREAM_REQUEST_BLOCK 数据包进行取消的回调函数。

(2) StrMiniReceiveDevicePacket: 获取 HW_STREAM_REQUEST_BLOCK 数据包的回调函数。

(3) StrMiniRequestTimeout: 当 HW_STREAM_REQUEST_BLOCK 数据包超时的回调函数。

(4) StrMiniEvent: 使小驱动程序支持某一种事件。

(5) StrMiniInterrupt: 当驱动获得中断时进入的回调函数。

另外，对于小驱动程序中的不同数据流还应该支持以下几种回调函数：

(1) StrMiniReceiveStreamDataPacket: 对于获取数据流的回调函数。

(2) StrMiniReceiveStreamControlPacket: 对于控制数据流的回调函数。

(3) StrMiniEvent: 使数据流支持一种事件。

(4) StrMiniClock: 对于数据流时钟控制的回调函数。

20.1.4 小驱动流的流控制

小驱动中的典型流程是初始化、运行和反初始化。一般需要遵循以下几个步骤：

① 小驱动所支持的硬件插入设备能够被即插即用管理器所检测到，然后为这种设备创建一个 PDO，该 PDO 负责一些即插即用的 IRP。

② I/O 子系统加载小驱动并调用小驱动的 DriverEntry 入口函数。一般在 DriverEntry 中初始化 HW_INITIALIZATION_DATA 数据结构。

③ 在小驱动的 DriverEntry 中将初始化好的 HW_INITIALIZATION_DATA 数据结构作为参数，用 StreamClassRegisterMinidriver 函数传递给类驱动程序。在 HW_INITIALIZATION_DATA 中包含一些控制 SRB 的函数地址，以便在类驱动中回调这些函数。

④ 在类驱动程序中，类驱动将构造一个 SRB，SRB 的 Command 设置为 SRB_INITIALIZE_DEVICE。并将此 SRB 作为参数，调用 HW_INITIALIZATION_DATA 数据结构中已经被初始化的 HwReceivePacket。因此，小驱动提供的 HwReceivePacket 有必要处理 SRB_INITIALIZE_DEVICE。

⑤ 类驱动会构造 SRB_GET_STREAM_INFO 类型的 SRB，并调用 HW_INITIALIZATION_DATA 数据结构中的 HwReceivePacket。因此，小驱动提供的

HwReceivePackets 有必要处理 SRB_GET_STREAM_INFO。

⑥ 类驱动会继续调用 HwReceivePackets 函数, 并将 SRB_OPEN_STREAM 类型的 SRB 传递给该函数。SRB_OPEN_STREAM 的 SRB 会指定一个 HW_STREAM_OBJECT, 该数据结构描述一个流的实例。因此, 小驱动提供的 HwReceivePackets 有必要处理 SRB_OPEN_STREAM。

⑦ 类驱动会通过小驱动提供的 HwReceivePackets 用 SRB_READ_DATA 或 SRB_WRITE_DATA 发送或接收数据。因此, 小驱动程序有必要处理 SRB_READ_DATA 或 SRB_WRITE_DATA。

⑧ 类驱动为了获得或者设置小驱动的某项属性或者流的某项属性, 会向小驱动提供的 HwReceivePackets 函数发送 HW_STREAM_OBJECT 请求, 并伴随着 HW_STREAM_REQUEST_BLOCK 数据结构。

⑨ 当类驱动想关闭流的时候, 会通过小驱动提供的 HwReceivePackets 函数发送 SRB_CLOSE_STREAM 请求。因此, 小驱动程序应该在 HwReceivePackets 中处理 SRB_CLOSE_STREAM 请求。

20.2 虚拟摄像头开发实例

上节已经对类驱动和小驱动程序的概念进行了讲解, 本节结合这些概念, 编写一个虚拟的摄像头驱动。

20.2.1 编译和安装

本节将带领读者编写一个虚拟的摄像头驱动。读者可以根据自己的设备, 根据这个框架, 开发出自己的摄像头应用驱动程序, 如 USB 摄像头驱动、SDIO 摄像头驱动等。

这个例子是在 DDK 所带例子 testcap 进行修改后得来的。这里没有提供给读者使用 VC 编译的环境, 读者必须到 DDK 的编译环境下进行编译。

DDK 原先自带的例子略显复杂, 除了虚拟摄像头外, 还提供了电视台信号接收、调台等功能。笔者只保留了虚拟摄像头部分, 对其他部分进行了裁剪, 这将有助于读者更好地学习掌握摄像头驱动程序的开发。

进行编译的时候, 需要进入 DDK 提供的编译环境执行 build 脚本, 安装时根据 INF 文件, 通过设备管理器进行安装。这与 HelloWDM 虚拟设备安装类似。

在设备管理器安装完毕后, 可以通过 DDK 提供的工具软件 GraphEdit 软件 (DirectX 开发套件也提供此工具) 查看。摄像头驱动安装好以后, 系统会认为此驱动是 DirectShow 组件里的一个源。通过 GraphEdit 可以加入这个源。如图 20-1 所示。

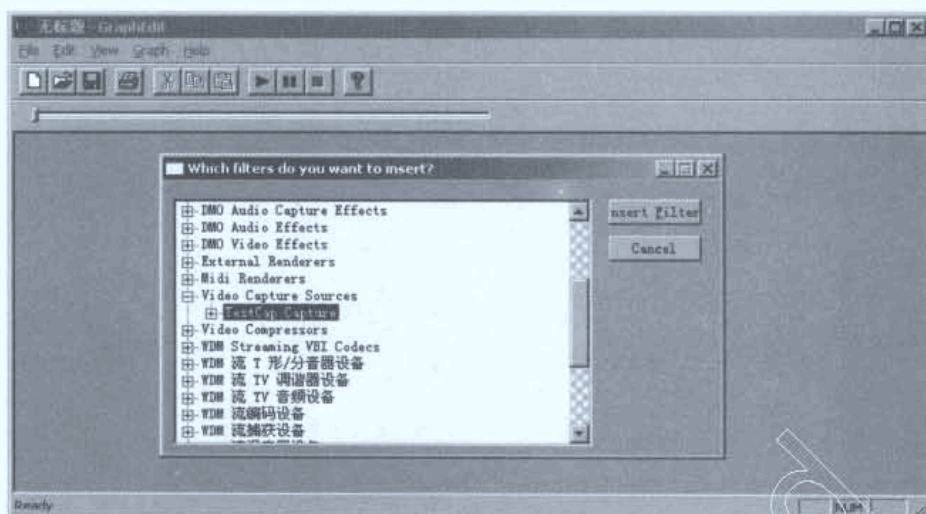


图 20-1 插入摄像头设备

在 GraphEdit 中插入摄像头源后，系统会自动寻找一条通路连接，直到显示在设备上。这个摄像头提供的输出格式是 RGB 格式，因此可以直接连接到一个显示设备上，而无需中间的解码插件。

单击 TestCap 的捕获或者预览管脚，右键单击鼠标并选择“Render”菜单，即可建立一条通路。如图 20-2 所示。

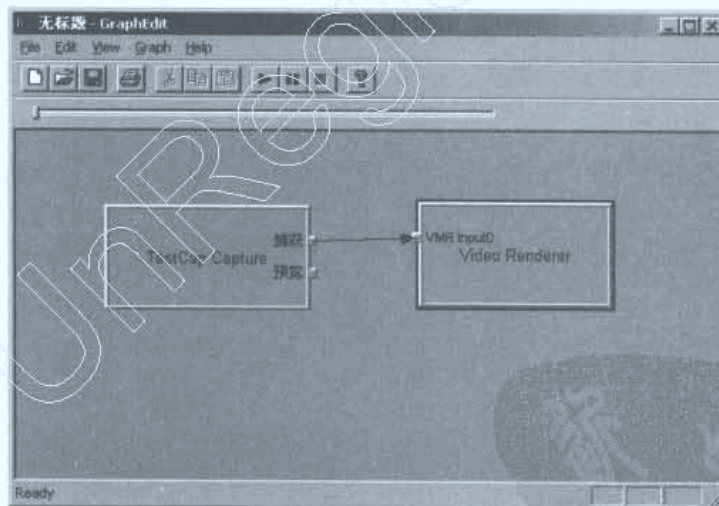


图 20-2 连接摄像头通路

在建立通路以后就可以测试摄像头驱动是否安装成功。单击“运行”按钮即可播放虚拟摄像头的图像。如图 20-3 所示。

测试好通路后，说明虚拟摄像头安装成功。所有支持微软摄像头的软件，如 QQ、MSN 等软件，会自动设置，并可以通过这些软件使用虚拟摄像头了。

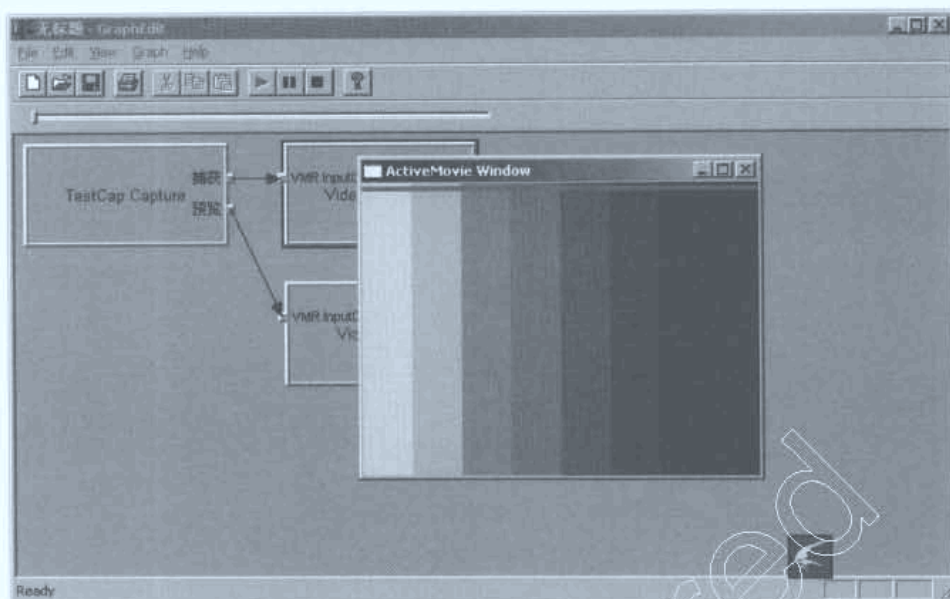


图 20-3 用 GraphEdit 软件播放

20.2.2 虚拟摄像头入口函数

这里所指入口函数是针对小驱动程序而言的，类驱动已经由微软提供，不需要程序员编写。在 DriverEntry 中所做的事情就是向类驱动程序提供一个 HW_INITIALIZATION_DATA 数据结构，这个数据结构提供了小驱动的若干个回调函数。DriverEntry 是由类驱动程序所调用的。

```
#001 ULONG
#002 DriverEntry (
#003     IN PDRIVER_OBJECT DriverObject,
#004     IN PUNICODE_STRING RegistryPath
#005 )
#006 {
#007     HW_INITIALIZATION_DATA HwInitData;
#008     ULONG ReturnValue;
#009
#010     DbgLogInfo(("TestCap: DriverEntry\n"));
#011     //对内存清零
#012     RtlZeroMemory(&HwInitData, sizeof(HwInitData));
#013     //初始化 HW_INITIALIZATION_DATA
#014     HwInitData.HwInitializationDataSize = sizeof(HwInitData);
#015
#016     //设置适配器的初始化指针
#017     HwInitData.HwInterrupt = NULL; // HwInterrupt;
#018     HwInitData.HwReceivePacket = AdapterReceivePacket;
#019     HwInitData.HwCancelPacket = AdapterCancelPacket;
#020     HwInitData.HwRequestTimeoutHandler = AdapterTimeoutPacket;
#021
#022     HwInitData.DeviceExtensionSize = sizeof(HW_DEVICE_EXTENSION);
#023     HwInitData.PerRequestExtensionSize = sizeof(SRB_EXTENSION);
```

```

#024     HwInitData.FilterInstanceExtensionSize = 0;
#025     HwInitData.PerStreamExtensionSize    = sizeof(STREAMEX);
#026     HwInitData.BusMasterDMA               = FALSE;
#027     HwInitData.Dma24BitAddresses          = FALSE;
#028     HwInitData.BufferAlignment            = 3;
#029     HwInitData.DmaBufferSize              = 0;
#030     //关闭同步机制
#031     HwInitData.TurnOffSynchronization     = TRUE;
#032     //注册类驱动
#033     ReturnValue = StreamClassRegisterAdapter(DriverObject, RegistryPath,
&HwInitData);
#034
#035     DbgLogInfo(("Testcap: StreamClassRegisterAdapter = %x\n", ReturnValue));
#036
#037     return ReturnValue;
#038 }

```

此段代码可以在配套光盘中本章的 testcap 目录下找到。

20.2.3 对 STREAM_REQUEST_BLOCK 的处理函数

在 DriverEntry 中设置的若干回调函数中，最重要的就是处理 STREAM_REQUEST_BLOCK 的回调函数。这个回调函数是在 DriverEntry 中设置 HW_INITIALIZATION_DATA 数据结构的 HwReceivePacket 的子域，在本例中就是 AdapterReceivePacket 函数。

在 AdapterReceivePacket 函数中主要要针对不同的 STREAM_REQUEST_BLOCK 进行处理。关于 STREAM_REQUEST_BLOCK，读者可以理解成 IRP，或者应用程序中的信息处理。这个函数主要部分就是一个大的 switch 结构，针对不同类型的 STREAM_REQUEST_BLOCK 分别处理。

```

#001 VOID
#002 STREAMAPI
#003 AdapterReceivePacket(
#004     IN PHW_STREAM_REQUEST_BLOCK pSrb
#005 )
#006 {
#007     PHW_DEVICE_EXTENSION pHwDevExt = ((PHW_DEVICE_EXTENSION)pSrb->HwDevice
Extension);
#008     BOOL Busy;
#009
#010     //这里采用一些同步措施，略
#011     //直到 SRB 处理完毕，循环才会结束
#012     while (TRUE) {
#013
#014         pSrb->Status = STATUS_SUCCESS;
#015         //根据 SRB 的 Command 进行不同处理
#016         switch (pSrb->Command)
#017         {
#018             case SRB_INITIALIZE_DEVICE:
#019                 //初始化小驱动程序
#020                 HwInitialize(pSrb);
#021                 break;
#022             case SRB_UNINITIALIZE_DEVICE:
#023                 //关闭小驱动程序

```



```

#024         HwUnInitialize(pSrb);
#025         break;
#026     case SRB_OPEN_STREAM:
#027         //打开视频流
#028         AdapterOpenStream(pSrb);
#029         break;
#030     case SRB_CLOSE_STREAM:
#031         //关闭视频流
#032         AdapterCloseStream(pSrb);
#033         break;
#034     case SRB_GET_STREAM_INFO:
#035         //获取视频流信息
#036         AdapterStreamInfo(pSrb);
#037         break;
#038     case SRB_GET_DATA_INTERSECTION:
#039         //设置本驱动支持的视频格式
#040         AdapterFormatFromRange(pSrb);
#041         break;
#042     case SRB_OPEN_DEVICE_INSTANCE:
#043     case SRB_CLOSE_DEVICE_INSTANCE:
#044         TRAP;
#045         pSrb->Status = STATUS_NOT_IMPLEMENTED;
#046         break;
#047     case SRB_GET_DEVICE_PROPERTY:
#048         //获取设备属性
#049         AdapterGetProperty (pSrb);
#050         break;
#051     case SRB_SET_DEVICE_PROPERTY:
#052         //设置设备属性
#053         AdapterSetProperty (pSrb);
#054         break;
#055     case SRB_PAGING_OUT_DRIVER:
#056         //驱动被换页出内存,被回调至此
#057         DbgLogInfo(("Testcap: Receiving SRB_PAGING_OUT_DRIVER --
SRB=%x\n", pSrb));
#058         break;
#059     case SRB_CHANGE_POWER_STATE:
#060         //电源状态切换,被回调至此
#061         DbgLogInfo(("Testcap: Receiving SRB_CHANGE_POWER_STATE -----
SRB=%x\n", pSrb));
#062         AdapterPowerState(pSrb);
#063         break;
#064     case SRB_INITIALIZATION_COMPLETE:
#065         //驱动初始化
#066         DbgLogInfo(("Testcap: Receiving SRB_INITIALIZATION_COMPLETE--
SRB=%x\n", pSrb));
#067         break;
#068     case SRB_UNKNOWN_DEVICE_COMMAND:
#069     default:
#070         pSrb->Status = STATUS_NOT_IMPLEMENTED;
#071     }
#072     //完成 SRB
#073     CompleteDeviceSRB (pSrb);
#074     //同步处理,略
#075 }
#076 }

```

此段代码可以在配套光盘中本章的 testcap 目录下找到。

20.2.4 打开视频流

在小驱动中有一个重要的步骤就是打开设备流。这个请求是由类驱动向小驱动发起的。小驱动程序需要根据它的请求，判断是本驱动是否支持请求的格式。

视频的格式用固定的 128 位 GUID 号码确定。

该程序用 AdapterVerifyFormat 函数帮助判断是否为本驱动所支持的视频格式，如果是，在 AdapterOpenStream 中会进行一些相关设置，并返回成功。

```
#001 VOID
#002 STREAMAPI
#003 AdapterOpenStream (
#004     PHW_STREAM_REQUEST_BLOCK pSrb
#005 )
#006 {
#007     PSTREAMEX pStrmEx = (PSTREAMEX)pSrb->StreamObject->HwStream
Extension;
#008     PHW_DEVICE_EXTENSION pHwDevExt = ((PHW_DEVICE_EXTENSION)pSrb->
HwDeviceExtension);
#009     int StreamNumber = pSrb->StreamObject->StreamNumber;
#010     PKSDataFormat pKSDDataFormat = pSrb->CommandData.OpenFormat;
#011     RtlZeroMemory(pStrmEx, sizeof(STREAMEX));
#012     //检测视频流号
#013     if (StreamNumber >= DRIVER_STREAM_COUNT || StreamNumber < 0) {
#014         pSrb->Status = STATUS_INVALID_PARAMETER;
#015         return;
#016     }
#017     //检测视频流实例
#018     if (pHwDevExt->ActualInstances[StreamNumber] >=
Streams[StreamNumber].hwStreamInfo.NumberOfPossibleInstances) {
#019         pSrb->Status = STATUS_INVALID_PARAMETER;
#020         return;
#021     }
#022     //检测请求的视频流格式是否合法
#023     if (!AdapterVerifyFormat (pKSDDataFormat, StreamNumber)) {
#024         pSrb->Status = STATUS_INVALID_PARAMETER;
#025         return;
#026     }
#027     //设置视频流格式
#028     if (!VideoSetFormat (pSrb)) {
#029         return;
#030     }
#031     ASSERT (pHwDevExt->pStrmEx [StreamNumber] == NULL);
#032     pHwDevExt->pStrmEx [StreamNumber] = (PSTREAMEX) pStrmEx;
#033     pSrb->StreamObject->ReceiveDataPacket =
(PVOID) Streams[StreamNumber].hwStreamObject.ReceiveDataPacket;
#034     pSrb->StreamObject->ReceiveControlPacket =
(PVOID) Streams[StreamNumber].hwStreamObject.ReceiveControlPacket;
#035     pSrb->StreamObject->Dma = Streams[StreamNumber].hwStreamObject.Dma;
#036     pSrb->StreamObject->Pio = Streams[StreamNumber].hwStreamObject.Pio;
#037     pSrb->StreamObject->StreamHeaderMediaSpecific =
Streams[StreamNumber].hwStreamObject.StreamHeaderMediaSpecific;
#038     pSrb->StreamObject->StreamHeaderWorkspace =
Streams[StreamNumber].hwStreamObject.StreamHeaderWorkspace;
#039
#040
#041
#042
#043
#044
```



```

#045     pSrb->StreamObject->HwClockObject =
#046         Streams[StreamNumber].hwStreamObject.HwClockObject;
#047         //将视频流实例号增加
#048     pHwDevExt->ActualInstances[StreamNumber]++;
#049     pStrmEx->pHwDevExt = pHwDevExt;           // 供定时器使用
#050     pStrmEx->pStreamObject = pSrb->StreamObject; // 供定时器使用
#051     pStrmEx->CompressionSettings.CompressionKeyFrameRate =
#052         pHwDevExt->CompressionSettings.CompressionKeyFrameRate;
#053     pStrmEx->CompressionSettings.CompressionPFramesPerKeyFrame =
#054         pHwDevExt->CompressionSettings.CompressionPFramesPerKeyFrame;
#055     pStrmEx->CompressionSettings.CompressionQuality =
#056         pHwDevExt->CompressionSettings.CompressionQuality;
#057     //初始化 VideoControl 属性
#058     pStrmEx->VideoControlMode = pHwDevExt->VideoControlMode;
#059     DbgLogInfo(("TestCap: AdapterOpenStream Exit\n"));
#060 }

```

此段代码可以在配套光盘中本章的 testcap 目录下找到。

20.2.5 对视频流的读取

前面介绍过的代码，基本上都是一些设置，而在 VideoReceiveCtrlPacket 函数中进行的代码则主要负责回应视频流的请求。

该函数的参数同样是 HW_STREAM_REQUEST_BLOCK 数据结构的指针。对于不同的 HW_STREAM_REQUEST_BLOCK 需要不同的处理。同样，该函数主要是由一个大的 switch 来处理不同类型的 HW_STREAM_REQUEST_BLOCK 请求。

在该函数处理中，会间接调用 ImageSynth 函数。ImageSynth 函数负责生成一幅图像，以便给虚拟摄像头定时提供视频。

```

#001 VOID
#002 STREAMAPI
#003 VideoReceiveCtrlPacket(
#004     IN PHW_STREAM_REQUEST_BLOCK pSrb
#005 )
#006 {
#007     PHW_DEVICE_EXTENSION pHwDevExt = ((PHW_DEVICE_EXTENSION)pSrb->
HwDeviceExtension);
#008     PSTREAMEX pStrmEx = (PSTREAMEX)pSrb->StreamObject->
HwStreamExtension;
#009     int StreamNumber = pStrmEx->pStreamObject->StreamNumber;
#010     BOOL Busy;
#011
#012     //同步处理，略
#013     while (TRUE) {
#014         pSrb->Status = STATUS_SUCCESS;
#015         switch (pSrb->Command)
#016         {
#017             case SRB_PROPOSE_DATA_FORMAT:
#018                 DbgLogInfo(("TestCap: Receiving SRB_PROPOSE_DATA_FORMAT SRB %p,
StreamNumber= %d\n", pSrb, StreamNumber));
#019                 if (!AdapterVerifyFormat (
#020                     pSrb->CommandData.OpenFormat,
#021                     pSrb->StreamObject->StreamNumber)) {

```

```

#022         pSrb->Status = STATUS_NO_MATCH;
#023         DbgLogInfo(("TestCap: SRB_PROPOSE_DATA_FORMAT FAILED\n"));
#024     }
#025     pSrb->Status = STATUS_NO_MATCH; // prevent dynamic format changes
#026     break;
#027     case SRB_SET_DATA_FORMAT:
#028         DbgLogInfo(("TestCap: SRB_SET_DATA_FORMAT\n"));
#029         if (!AdapterVerifyFormat (
#030             pSrb->CommandData.OpenFormat,
#031             pSrb->StreamObject->StreamNumber)) {
#032             pSrb->Status = STATUS_NO_MATCH;
#033             DbgLogInfo(("TestCap: SRB_SET_DATA_FORMAT FAILED\n"));
#034         } else {
#035             VideoSetFormat (pSrb);
#036             DbgLogInfo(("TestCap: SRB_SET_DATA_FORMAT SUCCEEDED\n"));
#037         }
#038         break;
#039     case SRB_GET_DATA_FORMAT:
#040         DbgLogInfo(("TestCap: SRB_GET_DATA_FORMAT\n"));
#041         pSrb->Status = STATUS_NOT_IMPLEMENTED;
#042         break;
#043     case SRB_SET_STREAM_STATE:
#044         //获取流状态
#045         VideoSetState(pSrb);
#046         break;
#047     case SRB_GET_STREAM_STATE:
#048         //设置流状态
#049         VideoGetState(pSrb);
#050         break;
#051     case SRB_GET_STREAM_PROPERTY:
#052         //获取流属性, 注意不是设备属性
#053         VideoGetProperty(pSrb);
#054         break;
#055     case SRB_SET_STREAM_PROPERTY:
#056         //设置流属性, 注意不是设备属性
#057         VideoSetProperty(pSrb);
#058         break;
#059     case SRB_INDICATE_MASTER_CLOCK:
#060         VideoIndicateMasterClock (pSrb);
#061         break;
#062     default:
#063         TRAP;
#064         pSrb->Status = STATUS_NOT_IMPLEMENTED;
#065     }
#066     CompleteStreamSRB (pSrb);
#067     //同步处理, 略
#068 }
#069 }

```

此段代码可以在配套光盘中本章的 testcap 目录下找到。

20.3 小结

本章主要介绍了微软提供的摄像头驱动框架。在该框架中, 微软提供了类驱动和小驱动的概念。这种设计有两种好处:

Windows 驱动开发技术详解

第一，将复杂的处理放在类驱动中，而这些都已经由微软提供好了，程序员只需将各种摄像头特殊的设置写进小驱动中就可以了。

第二，微软提供的类驱动将这种接口标准化，驱动程序员不用再负责定义和应用程序的接口。

类驱动和小驱动这样的开发模式在 DDK 中是很常见的，如本章介绍的摄像头驱动模型，还有 USB 驱动设计模型等。本章同时还给出一个具体的虚拟摄像头的实例，读者可以在这个驱动实例的基础上进行修改，以便实现特殊设备的摄像头驱动。

第 4 篇 提高篇

第 21 章 再论 IRP

第 22 章 过滤驱动程序

第 23 章 高级调试技巧

第 21 章 再论 IRP

不管在 WDM 驱动还是在 NT 式驱动中，驱动经常发送 IRP 给某一个设备对象。可能是自己创建 IRP 后，并将 IRP 发送给底层驱动，也可能是将上层 IRP 转发给底层驱动。可以看出正确灵活地使用 IRP 在驱动程序中是非常重要的环节。本章主要分为两部分：第一部分总结了转发 IRP 的各种情况，第二部分总结了在驱动中创建 IRP 的各种情况。

21.1 转发 IRP

21.1.1 直接转发

下面的代码演示了驱动将传送进来的 IRP 直接转发到底层驱动的过程，在此并不需要做进一步处理（如获得 IRP 的返回结果等），只是将 IRP 丢给底层驱动，这是最简单的转发 IRP 的方法。

```
#001 NTSTATUS
#002 DispatchRoutine_1(
#003     IN PDEVICE_OBJECT DeviceObject,
#004     IN PIRP Irp
#005 )
#006 {
#007     //略过当前 I/O 堆栈
#008     IoSkipCurrentIrpStackLocation (Irp);
#009     //调用底层驱动
#010     return IoCallDriver(TopOfDeviceStack, Irp);
#011 }
```

21.1.2 转发并且等待

下列代码演示了将传进来的 IRP 直接转发到底层驱动，并且等待底层驱动将 IRP 处理完毕。这时驱动程序重新获取 IRP 的控制权。这主要是依靠一个同步事件，并设置 IRP 的

完成例程。在完成例程中设置了这个同步事件。如果本层驱动想重新获取 IRP 的控制，必须在完成例程后返回 STATUS_MORE_PROCESSING_REQUIRED，这时候 IRP 需要重新被完成，即需要重新调用 IoCompleteRequest。这种情况多出于 IRP_MN_START_DEVICE IRP 等 IRP 的处理情况。

```
#001 NTSTATUS
#002 DispatchRoutine_2(
#003     IN PDEVICE_OBJECT DeviceObject,
#004     IN PIRP Irp
#005 )
#006 {
#007     KEVENT event;
#008     NTSTATUS status;
#009
#010     KeInitializeEvent(&event, NotificationEvent, FALSE);
#011     //复制当前 I/O 堆栈到下一个 I/O 堆栈
#012     IoCopyCurrentIrpStackLocationToNext(Irp);
#013
#014     //设置完成例程
#015     IoSetCompletionRoutine(Irp,
#016                           CompletionRoutine_2,
#017                           &event,
#018                           TRUE,
#019                           TRUE,
#020                           TRUE
#021     );
#022     //调用底层驱动
#023     status = IoCallDriver(TopOfDeviceStack, Irp);
#024
#025     if (status == STATUS_PENDING) {
#026         //等待内核事件
#027         KeWaitForSingleObject(&event,
#028                               Executive, // 等待原因
#029                               KernelMode, // 必须是 Kernelmode
#030                               FALSE,
#031                               NULL // 无限地等待下去
#032         );
#033         status = Irp->IoStatus.Status;
#034     }
#035     // 在这里可以设置需要做的工作
#036     //将 IRP 请求结束
#037     IoCompleteRequest(Irp, IO_NO_INCREMENT);
#038     return status;
#039 }
#040 NTSTATUS
#041 CompletionRoutine_2(
#042     IN PDEVICE_OBJECT DeviceObject,
#043     IN PIRP Irp,
#044     IN PVOID Context
#045 )
#046 {
#047     //判断是否挂起
#048     if (Irp->PendingReturned == TRUE)
#049     {
#050         KeSetEvent((PKEVENT) Context, IO_NO_INCREMENT, FALSE);
```



```
#051 }  
#052 return STATUS_MORE_PROCESSING_REQUIRED;  
#053 }
```

21.1.3 转发并且设置完成例程

这种情况是驱动设置了一个完成例程，将 IRP 向底层驱动转发，然后在这个派遣函数中直接返回底层驱动处理 IRP 的状态。这样做的目的就是在完成例程中有机会修改 IRP。

```
#001 NTSTATUS  
#002 DispatchRoutine_3(  
#003     IN PDEVICE_OBJECT DeviceObject,  
#004     IN PIRP Irp  
#005 )  
#006 {  
#007     NTSTATUS status;  
#008     //将当前 I/O 堆栈复制至下层 I/O 堆栈  
#009     IoCopyCurrentIrpStackLocationToNext(Irp);  
#010     //设置完成例程  
#011     IoSetCompletionRoutine(Irp,  
#012                           CompletionRoutine_31, // or CompletionRoutine_32  
#013                           NULL,  
#014                           TRUE,  
#015                           TRUE,  
#016                           TRUE  
#017     );  
#018     //调用底层驱动  
#019     return IoCallDriver(TopOfDeviceStack, Irp);  
#020 }
```

对于这样的完成例程，有两种编写方法，如 CompletionRoutine_31 或 CompletionRoutine_32。

```
#001 NTSTATUS  
#002 CompletionRoutine_31(  
#003     IN PDEVICE_OBJECT DeviceObject,  
#004     IN PIRP Irp,  
#005     IN PVOID Context  
#006 )  
#007 {  
#008     //判断是否需要挂起 IRP  
#009     if (Irp->PendingReturned)  
#010     {  
#011         //挂起 IRP  
#012         IoMarkIrpPending(Irp);  
#013     }  
#014     //IRP 已经完成  
#015     return STATUS_CONTINUE_COMPLETION;  
#016 }  
#017  
#018 NTSTATUS  
#019 CompletionRoutine_32(  
#020     IN PDEVICE_OBJECT DeviceObject,  
#021     IN PIRP Irp,  
#022     IN PVOID Context
```

```

#023     }
#024 {
#025     //判断是否需要挂起 IRP
#026     if (Irp->PendingReturned) {
#027         //挂起 IRP
#028         IoMarkIrpPending( Irp );
#029     }
#030     //挂起 IRP 请求
#031     IoCompleteRequest( Irp, IO_NO_INCREMENT);
#032     //指示 IRP 还需要进一步结束
#033     return STATUS_MORE_PROCESSING_REQUIRED;
#034 }

```

21.1.4 暂时挂起当前 IRP

下面代码演示了暂时不对 IRP 处理而是挂起当前 IRP。所谓挂起 IRP，就是将其状态设置成 PENDING。这种情况经常发生在多个 IRP 并行发出的状态。应当指出，在驱动程序运行中需要将其串行化处理。驱动可以采用 StartIo 例程，也可以自己将 IRP 依次插入队列，然后在适当时候依次处理这些 IRP。

```

#001 NTSTATUS
#002 DispatchRoutine_4(
#003     IN PDEVICE_OBJECT DeviceObject,
#004     IN PIRP Irp
#005 )
#006 {
#007     NTSTATUS status;
#008     //挂起 IRP 请求
#009     IoMarkIrpPending( Irp );
#010     //将当前 I/O 堆栈复制到下一层 I/O 堆栈
#011     IoCopyCurrentIrpStackLocationToNext( Irp );
#012     //设置 IRP 完成例程
#013     IoSetCompletionRoutine( Irp,
#014                             CompletionRoutine_41, // or CompletionRoutine_42
#015                             NULL,
#016                             TRUE,
#017                             TRUE,
#018                             TRUE
#019     );
#020     //调用底层驱动程序
#021     IoCallDriver( TopOfDeviceStack, Irp );
#022
#023     //返回挂起状态
#024     return STATUS_PENDING ;
#025 }

```

完成例程完成后可以返回 STATUS_CONTINUE_COMPLETION 或者 STATUS_MORE_PROCESSING_REQUIRED。如果返回 STATUS_MORE_PROCESSING_REQUIRED 时，可以重新获取 IRP 的控制权，并调用 IoCompleteRequest。

```

#001 NTSTATUS
#002 CompletionRoutine_41(
#003     IN PDEVICE_OBJECT DeviceObject,

```



```
#004     IN PIRP          Irp,
#005     IN PVOID         Context
#006     )
#007     {
#008         // 一旦返回 STATUS_CONTINUE_COMPLETION, 就不能再操作 IRP 了
#009         return STATUS_CONTINUE_COMPLETION ;
#010     }
#011
#012     NTSTATUS
#013     CompletionRoutine_42 (
#014         IN PDEVICE_OBJECT DeviceObject,
#015         IN PIRP          Irp,
#016         IN PVOID         Context
#017     )
#018     {
#019         // 如果返回 STATUS_MORE_PROCESSING_REQUIRED
#020         // IRP 需要再次被结束
#021         return STATUS_MORE_PROCESSING_REQUIRED ;
#022     }
```

21.1.5 不转发 IRP

还有一种处理 IRP 的情况，既直接在派遣中完成 IRP，而不向下层驱动转发。这需要设置 IRP 的状态、操作字节数，并调用 `IoCompletRequest` 来完成 IRP。

```
#001     NTSTATUS
#002     DispatchRoutine_5(
#003         IN PDEVICE_OBJECT DeviceObject,
#004         IN PIRP Irp
#005     )
#006     {
#007         // 添加处理 IRP 的代码
#008         Irp->IoStatus.Status = STATUS_XXX;
#009         // 设置 IRP 的操作字节数
#010         Irp->IoStatus.Information = YYY;
#011         // 结束 IRP 请求
#012         IoCompletRequest(Irp, IO_NO_INCREMENT);
#013         return STATUS_XXX;
#014     }
```

21.2 创建 IRP

创建 IRP 总体上说可以分为创建同步 IRP 和创建异步 IRP 两种。同步 IRP 指的是创建 IRP 后，用 `IoCallDriver` 将这个 IRP 传递给底层驱动处理，一定等到底层驱动处理完毕才会从 `IoCallDriver` 返回。

异步 IRP 指的是创建 IRP 后 `IoCallDriver` 会先返回，但这时候其并没有真正完成。这时可以通过设置完成例程，用来获知 IRP 真正完成。异步 IRP 要比同步 IRP 复杂很多，但异步 IRP 与同步 IRP 在用法上相比较为灵活。

21.2.1 IoBuildDeviceIoControlRequest

IoBuildDeviceIoControlRequest 可以创建 IRP_MJ_INTERNAL_DEVICE_CONTROL 和 IRP_MJ_DEVICE_CONTROL 两种 IRP。用 IoBuildDeviceIoControlRequest 创建的 IRP 是同步 IRP。

```

#001 NTSTATUS
#002 MakeSynchronousIoctl(
#003     IN PDEVICE_OBJECT    TopOfDeviceStack,
#004     IN ULONG              IoctlControlCode,
#005     PVOID                 InputBuffer,
#006     ULONG                 InputBufferLength,
#007     PVOID                 OutputBuffer,
#008     ULONG                 OutputBufferLength
#009 )
#010 {
#011     KEVENT                event;
#012     PIRP                  irp;
#013     IO_STATUS_BLOCK        ioStatus;
#014     NTSTATUS               status;
#015     //创建设备控制 IRP 给其他驱动
#016     //初始化同步事件
#017     KeInitializeEvent(&event, NotificationEvent, FALSE);
#018     //用 IoBuildDeviceIoControlRequest 创建 IRP
#019     irp = IoBuildDeviceIoControlRequest (
#020         IoctlControlCode,
#021         TopOfDeviceStack,
#022         InputBuffer,
#023         InputBufferLength,
#024         OutputBuffer,
#025         OutputBufferLength,
#026         FALSE, // External
#027         &event,
#028         &ioStatus);
#029     //判断 IRP 是否为空
#030     if (NULL == irp) {
#031         return STATUS_INSUFFICIENT_RESOURCES;
#032     }
#033     //调用底层驱动程序
#034     status = IoCallDriver(TopOfDeviceStack, irp);
#035     //判断 IRP 是否被挂起
#036     if (status == STATUS_PENDING) {
#037         //如果被挂起需要等待 IRP 结束
#038         status = KeWaitForSingleObject(
#039             &event,
#040             Executive, // 等待原因
#041             KernelMode, // 这里必须是 KernelMode
#042             FALSE,
#043             NULL);
#044
#045         status = ioStatus.Status;
#046     }
#047     return status;
#048 }
#049

```


21.2.2 创建有超时的 IOCTL IRP

有些时候，将 IRP 传递给底层驱动时，底层驱动由于种种原因，可能永远不会将 IRP 完成。例如，驱动会一直读取设备的某个状态寄存器，而由于硬件问题，这个寄存器永远也得不到正确代码。如果要解决这种问题，就需要设置一个超时。如果在超时内，没有将 IRP 完成，上层驱动就会调用 `IoCancelIrp` 将 IRP 设置完成。完成的状态是 `STATUS_TIMEOUT`。

```
#001 NTSTATUS
#002 MakeSynchronousIoctlWithTimeOut(
#003     IN PDEVICE_OBJECT   TopOfDeviceStack,
#004     IN ULONG             IoctlControlCode,
#005     PVOID                InputBuffer,
#006     ULONG                InputBufferLength,
#007     PVOID                OutputBuffer,
#008     ULONG                OutputBufferLength,
#009     IN ULONG             Milliseconds
#010 )
#011 {
#012     NTSTATUS status;
#013     PIRP irp;
#014     KEVENT event;
#015     IO_STATUS_BLOCK ioStatus;
#016     LARGE_INTEGER dueTime;
#017     IRPLOCK lock;
#018     //初始化同步事件
#019     KeInitializeEvent(&event, NotificationEvent, FALSE);
#020     //创建 IRP
#021     irp = IoBuildDeviceIoControlRequest (
#022         IoctlControlCode,
#023         TopOfDeviceStack,
#024         InputBuffer,
#025         InputBufferLength,
#026         OutputBuffer,
#027         OutputBufferLength,
#028         FALSE, // External ioctl
#029         &event,
#030         &ioStatus);
#031     //判断 IRP 是否为空
#032     if (irp == NULL) {
#033         return STATUS_INSUFFICIENT_RESOURCES;
#034     }
#035
#036     lock = IRPLOCK_CANCELABLE;
#037
#038     //设置完成例程
#039     IoSetCompletionRoutine(
#040         irp,
#041         MakeSynchronousIoctlWithTimeOutCompletion,
#042         &lock,
#043         TRUE,
#044         TRUE,
#045         TRUE
#046     );
#047     //调用底层驱动
```

```

#048     status = IoCallDriver(TopOfDeviceStack, irp);
#049     //判断是否 IRP 被挂起
#050     if (status == STATUS_PENDING) {
#051         //定义 1s 延时
#052         dueTime.QuadPart = -10000 * Milliseconds;
#053         //等待同步事件
#054         status = KeWaitForSingleObject(
#055             &event,
#056             Executive,
#057             KernelMode,
#058             FALSE,
#059             &dueTime
#060             );
#061         //如果是超时
#062         if (status == STATUS_TIMEOUT) {
#063             if (InterlockedExchange((PVOID)&lock, IRPLOCK_CANCEL_STARTED)
#064                 == IRPLOCK_CANCELABLE) {
#065                 //取消 IRP 请求
#066                 IoCancelIrp(irp);
#067                 if (InterlockedExchange(&lock, IRPLOCK_CANCEL_COMPLETE)
#068                     == IRPLOCK_COMPLETED) {
#069                     //结束 IRP 请求
#070                     IoCompleteRequest(irp, IO_NO_INCREMENT);
#071                 }
#072             }
#073             //等待同步时间
#074             KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);
#075             //设置 IRP 完成状态
#076             ioStatus.Status = status; // Return STATUS_TIMEOUT
#077         } else {
#078             status = ioStatus.Status;
#079         }
#080     }
#081     return status;
#082 }
#083 NTSTATUS
#084 MakeSynchronousIoctlWithTimeOutCompletion(
#085     IN PDEVICE_OBJECT DeviceObject,
#086     IN PIRP Irp,
#087     IN PVOID Context
#088 )
#089 {
#090     PLONG lock;
#091     lock = (PLONG) Context;
#092     if (InterlockedExchange((PVOID)&lock, IRPLOCK_COMPLETED) == IRPLOCK_
CANCEL_STARTED) {
#093         return STATUS_MORE_PROCESSING_REQUIRED;
#094     }
#095     return STATUS_CONTINUE_COMPLETION ;
#096 }

```

21.2.3 用 IoBuildSynchronousFsdRequest 创建 IRP

对于创建非 IOCTL 的 IRP，可以通过使用 IoBuildSynchronousFsdRequest 内核函数创建。应当指出该内核函数创建的 IRP 也是同步 IRP。

Windows 驱动开发技术详解

```
#001 NTSTATUS
#002 MakeSynchronousNonIoctlRequest (
#003     PDEVICE_OBJECT TopOfDeviceStack,
#004     PVOID WriteBuffer,
#005     ULONG NumBytes
#006 )
#007 {
#008     NTSTATUS status;
#009     PIRP irp;
#010     LARGE_INTEGER startingOffset;
#011     KEVENT event;
#012     IO_STATUS_BLOCK ioStatus;
#013     PVOID context;
#014
#015     startingOffset.QuadPart = (LONGLONG) 0;
#016     //分配非页内存
#017     context = ExAllocatePoolWithTag(NonPagedPool, sizeof(ULONG), 'ITag');
#018     if(!context) {
#019         return STATUS_INSUFFICIENT_RESOURCES;
#020     }
#021     //初始化同步事件
#022     KeInitializeEvent(&event, NotificationEvent, FALSE);
#023     //创建同步事件
#024     irp = IoBuildSynchronousFsdRequest(
#025         IRP_MJ_WRITE,
#026         TopOfDeviceStack,
#027         WriteBuffer,
#028         NumBytes,
#029         &startingOffset, // Optional
#030         &event,
#031         &ioStatus
#032     );
#033     //判断 IRP 是否为空
#034     if (NULL == irp) {
#035         //回收内存
#036         ExFreePool(context);
#037         //指示资源不够
#038         return STATUS_INSUFFICIENT_RESOURCES;
#039     }
#040     //设置完成 IRP 完成例程
#041     IoSetCompletionRoutine(irp,
#042         MakeSynchronousNonIoctlRequestCompletion,
#043         context,
#044         TRUE,
#045         TRUE,
#046         TRUE);
#047     //调用底层驱动
#048     status = IoCallDriver(TopOfDeviceStack, irp);
#049     //判断 IRP 是否被挂起
#050     if (status == STATUS_PENDING) {
#051         //等待 IRP 被结束
#052         status = KeWaitForSingleObject(
#053             &event,
#054             Executive,
#055             KernelMode,
#056             FALSE, // Not alertable
#057             NULL);
```

```

#058         status = ioStatus.Status;
#059     }
#060     return status;
#061 }
#062 NTSTATUS
#063 MakeSynchronousNonIoctlRequestCompletion(
#064     IN PDEVICE_OBJECT DeviceObject,
#065     IN PIRP            Irp,
#066     IN PVOID           Context
#067 )
#068 {
#069     if (Context) {
#070         //回收内存
#071         ExFreePool(Context);
#072     }
#073     //指示 IRP 被结束
#074     return STATUS_CONTINUE_COMPLETION ;
#075 }

```

21.2.4 关于 IoBuildAsynchronousFsdRequest

这里介绍如何使用 `IoBuildAsynchronousFsdRequest` 创建异步 IRP。异步 IRP 可以在任意上下文中创建，因为它没有和任何线程关联。程序员需要提供完成例程和释放 IRP 的相关内存等。

同步 IRP 不需要自己释放 IRP 占用的相关内存，有关操作由 I/O 管理器完成。异步 IRP 需要自己释放所占用的内存。

```

#001 NTSTATUS
#002 MakeAsynchronousRequest(
#003     PDEVICE_OBJECT TopOfDeviceStack,
#004     PVOID           WriteBuffer,
#005     ULONG           NumBytes
#006 )
#007 {
#008     NTSTATUS status;
#009     PIRP     irp;
#010     LARGE_INTEGER startingOffset;
#011     PIO_STACK_LOCATION nextStack;
#012     PVOID context;
#013
#014     startingOffset.QuadPart = (LONGLONG) 0;
#015     //创建 IRP
#016     irp = IoBuildAsynchronousFsdRequest(
#017         IRP_MJ_WRITE,
#018         TopOfDeviceStack,
#019         WriteBuffer,
#020         NumBytes,
#021         &startingOffset, // Optional
#022         NULL,
#023     );
#024     //判断 IRP 是否为空
#025     if (NULL == irp) {
#026
#027         return STATUS_INSUFFICIENT_RESOURCES;

```


Windows 驱动开发技术详解

```
#028     }
#029     //分配非分页内存
#030     context = ExAllocatePoolWithTag(NonPagedPool, sizeof(ULONG_PTR), 'ITag');
#031     if (NULL == context) {
#032         IoFreeIrp(irp);
#033         return STATUS_INSUFFICIENT_RESOURCES;
#034     }
#035     //设置 IRP 完成例程
#036     IoSetCompletionRoutine(irp,
#037         MakeAsynchronousRequestCompletion,
#038         context,
#039         TRUE,
#040         TRUE,
#041         TRUE);
#042     //得到下一层 I/O 堆栈
#043     nextStack = IoGetNextIrpStackLocation(irp);
#044     //设置下一层 I/O 堆栈的 IRP 为 IRP_MJ_SCSI
#045     nextStack->MajorFunction = IRP_MJ_SCSI;
#046     //调用底层驱动
#047     IoCallDriver(TopOfDeviceStack, irp);
#048     return STATUS_SUCCESS;
#049 }
#050 NTSTATUS
#051 MakeAsynchronousRequestCompletion(
#052     IN PDEVICE_OBJECT DeviceObject,
#053     IN PIRP Irp,
#054     IN PVOID Context
#055 )
#056 {
#057     PMDL mdl, nextMdl;
#058     //判断是否是缓冲区设备
#059     if(Irp->AssociatedIrp.SystemBuffer && (Irp->Flags & IRP_DEALLOCATE_BUFFER)){
#060         ExFreePool(Irp->AssociatedIrp.SystemBuffer);
#061     }
#062     else if (Irp->MdlAddress != NULL) {
#063         for (mdl = Irp->MdlAddress; mdl != NULL; mdl = nextMdl) {
#064             nextMdl = mdl->Next;
#065             //对 MDL 解锁
#066             MmUnlockPages( mdl ); IoFreeMdl( mdl );
#067         }
#068         Irp->MdlAddress = NULL;
#069     }
#070
#071     if(Context) {
#072         //释放内存
#073         ExFreePool(Context);
#074     }
#075     //释放 IRP 数据结构
#076     IoFreeIrp(irp);
#077     return STATUS_MORE_PROCESSING_REQUIRED;
#078 }
```

21.2.5 关于 IoAllocateIrp

除 IoBuildAsynchronousFsdRequest 之外，也可以用 IoAllocateIrp 创建异步 IRP。类似

于 IoAllocateIrp 创建的 IRP 需要由程序员自己回收内存, 这种情况一般是放在 IRP 的完成例程中。

回收 IRP 时, 如果是缓冲区 IRP 则需要清空缓冲区, 如果是直接操作内存 IRP 则使用 MDL 的 IRP, 这时需要依次对各个 MDL 进行释放。

```
#001 NTSTATUS
#002 MakeAsynchronousRequest2(
#003     PDEVICE_OBJECT TopOfDeviceStack,
#004     PVOID           WriteBuffer,
#005     ULONG           NumBytes
#006 )
#007 {
#008     NTSTATUS      status;
#009     PIRP          irp;
#010     LARGE_INTEGER startingOffset;
#011     KEVENT        event;
#012     PIO_STACK_LOCATION nextStack;
#013
#014     startingOffset.QuadPart = (LONGLONG) 0;
#015     //创建 IRP
#016     irp = IoAllocateIrp( TopOfDeviceStack->StackSize, FALSE );
#017     //判断 IRP 是否为空
#018     if (NULL == irp) {
#019         return STATUS_INSUFFICIENT_RESOURCES;
#020     }
#021
#022     //得到下一层 I/O 堆栈
#023     nextStack = IoGetNextIrpStackLocation( irp );
#024     //设置下一层的 I/O 堆栈
#025     nextStack->MajorFunction = IRP_MJ_WRITE;
#026     //设置下一层 I/O 堆栈的字节数
#027     nextStack->Parameters.Write.Length = NumBytes;
#028     //设置下一层 I/O 堆栈的偏移
#029     nextStack->Parameters.Write.ByteOffset = startingOffset;
#030
#031     if(TopOfDeviceStack->Flags & DO_BUFFERED_IO) {
#032         //设置 IRP 的缓冲区
#033         irp->AssociatedIrp.SystemBuffer = WriteBuffer;
#034         irp->MdlAddress = NULL;
#035     } else if (TopOfDeviceStack->Flags & DO_DIRECT_IO) {
#036         //申请 MDL
#037         irp->MdlAddress = IoAllocateMdl( WriteBuffer,
#038                                         NumBytes,
#039                                         FALSE,
#040                                         FALSE,
#041                                         (PIRP) NULL );
#042         if (irp->MdlAddress == NULL) {
#043             //回收 IRP
#044             IoFreeIrp( irp );
#045             return STATUS_INSUFFICIENT_RESOURCES;
#046         }
#047         try {
#048             //锁定 MDL 内存
```


Windows 驱动开发技术详解

```
#049         MmProbeAndLockPages( irp->MdlAddress,
#050                                     KernelMode,
#051                                     (LOCK_OPERATION) (nextStack->MajorFunction ==
IRP_MJ_WRITE ? IoReadAccess : IoWriteAccess) );
#052
#053     } except(EXCEPTION_EXECUTE_HANDLER) {
#054         //如果失败, 就回收 MDL
#055         if (irp->MdlAddress != NULL) {
#056             //回收 MDL
#057             IoFreeMdl( irp->MdlAddress );
#058         }
#059         //回收 IRP
#060         IoFreeIrp( irp );
#061         return GetExceptionCode();
#062     }
#063 }
#064 }
#065 //设置 IRP 完成例程
#066 IoSetCompletionRoutine(irp,
#067                         MakeAsynchronousRequestCompletion?,
#068                         NULL,
#069                         TRUE,
#070                         TRUE,
#071                         TRUE);
#072 //调用下层驱动程序
#073 (void) IoCallDriver(TargetDeviceObject, irp);
#074
#075 return STATUS_SUCCESS;
#076 }
#077
#078 NTSTATUS
#079 MakeAsynchronousRequestCompletion2(
#080     IN PDEVICE_OBJECT DeviceObject,
#081     IN PIRP Irp,
#082     IN PVOID Context
#083 )
#084 {
#085     PMDL mdl, nextMdl;
#086     //判断 MDL 是否为空
#087     if (Irp->MdlAddress != NULL) {
#088         for (mdl = Irp->MdlAddress; mdl != NULL; mdl = nextMdl) {
#089             nextMdl = mdl->Next;
#090             //对 MDL 解锁
#091             MmUnlockPages( mdl ); IoFreeMdl( mdl ); // This function will also
unmap pages.
#092         }
#093         Irp->MdlAddress = NULL;
#094     }
#095     //回收 IRP
#096     IoFreeIrp(Irp);
#097
#098     return STATUS_MORE_PROCESSING_REQUIRED;
#099 }
```

21.3 小结

本章将相关 IRP 的操作做了进一步的总结。首先是转发 IRP，归纳了几种不同的方式。其次总结了创建 IRP 的几种不同方法。创建 IRP 总的来说分为创建同步 IRP 和创建异步 IRP。对于创建同步 IRP，操作比较简单，I/O 管理器会负责回收 IRP 的相关内存，但是使用不够灵活。对于创建异步 IRP，操作比较复杂，程序员需要自己负责对 IRP 及相关内存回收，但使用十分灵活。



第 22 章 过滤驱动程序

本章主要介绍 WDM 和 NT 式过滤驱动程序开发。过滤驱动程序开发十分灵活，可以修改已有驱动程序的功能，也可以对数据进行过滤加密。另外，利用过滤驱动程序还能编写出很多具有相当神奇功能的程序来。

22.1 文件过滤驱动程序

文件过滤驱动是过滤驱动中典型的一种，它将挂载在磁盘驱动之上。它将所有发往磁盘驱动的 IRP 全部拦截，并有选择地过滤这些 IRP。

22.1.1 过滤驱动程序概念

对于 WDM 框架的过滤程序来说共有两种：一种是高层过滤驱动程序，一种是低层过滤程序。如果将过滤程序附在功能驱动（FDO）的下面，这样介于 FDO 和 PDO 之间的过滤驱动被称为低层过滤驱动程序，一般记为 Low FiDO。顺便说一下，之所以写成 FiDO 是为了与 FDO 相区别。

如果被附在功能驱动（FDO）的上面，则称为上层过滤驱动程序，一般表示为 High FiDO。在 WDM 框架中过滤驱动可以相互嵌套，层层叠加，即上层过滤驱动程序之上可以再附加更高层的过滤驱动程序。同理，低层过滤驱动程序可以被更低层的过滤驱动所过滤，如图 22-1 所示。

22.1.2 过滤驱动程序的入口函数

作为一个过滤驱动程序，一般都是将上层驱动程序传进来的 IRP 进行拦截以后并不做任何处理，并将此 IRP 再转发给底层驱动程序，这种方式被称为 Pass-Through。而对于个

别 IRP，过滤驱动程序需要对其处理。过滤驱动的作用是过滤 IRP，然后将 IRP 做进一步处理。不同过滤驱动会有不同的需求，这种处理也不会不同。一个最简单的过滤驱动程序，就是让 FDO 到 PDO 之间的请求完全通过，而不加以任何干涉。

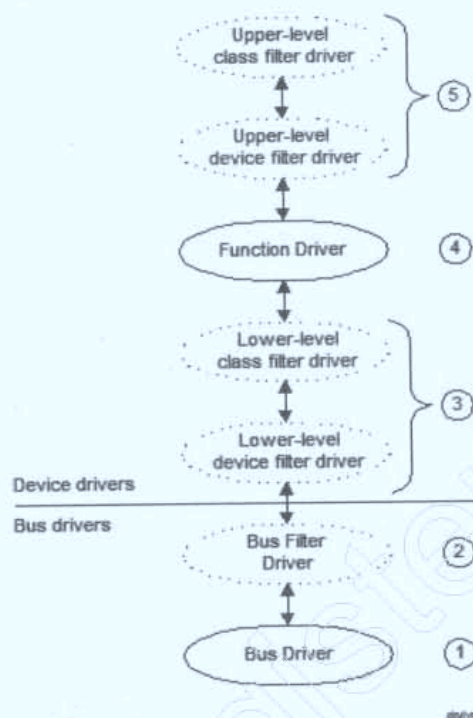


图 22-1 过滤驱动模型

在下面的实例代码中，驱动程序的入口函数用一个 for 循环将所有的 IRP 的派遣函数都置成了 DispatchAny 函数。而对于 IRP_MJ_POWER、IRP_MJ_PNP 及 IRP_MJ SCSI 则进行单独处理。当然，对于不同的过滤驱动，可以选择过滤不同的 IRP，而让其他的 IRP 只是穿过本层驱动。

```
#001 #pragma INITCODE
#002 extern "C" NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
#003     IN PUNICODE_STRING RegistryPath)
#004 { //驱动程序的入口函数
#005     KdPrint((DRIVERNAME " - Entering DriverEntry: DriverObject %8.8lX\n",
DriverObject));
#006     //设置卸载函数指针
#007     DriverObject->DriverUnload = DriverUnload;
#008     //设置 AddDevice 函数指针
#009     DriverObject->DriverExtension->AddDevice = AddDevice;
#010     //将所有 IRP 的派遣函数都设置为 DispatchAny
#011     for (int i = 0; i < arraysize(DriverObject->MajorFunction); ++i)
#012         DriverObject->MajorFunction[i] = DispatchAny;
#013     //重新设置三个 IRP 派遣函数
#014     DriverObject->MajorFunction[IRP_MJ_POWER] = DispatchPower;
#015     DriverObject->MajorFunction[IRP_MJ_PNP] = DispatchPnp;
```


Windows 驱动开发技术详解

```
#016     DriverObject->MajorFunction[IRP_MJ SCSI] = DispatchForSCSI;
#017     return STATUS_SUCCESS;
#018 } // 驱动程序的人口函数
```

此段代码可以在配套光盘中本章的 FileFilter 目录下找到。

对于不做处理的 IRP 只需要将其直接穿过本层驱动即可，主要是采用 IoSkipCurrentIrpStackLocation 函数将 IRP 达到，再用内核函数 IoCallDriver 调用更底层的驱动程序。运用下面的函数就可以达到此目的。在这个函数中还加入一些同步处理操作，并将 IRP 的相关信息作为调试信息打印出来。

```
#001 #pragma LOCKEDCODE
#002 NTSTATUS DispatchAny(IN PDEVICE_OBJECT fido, IN PIRP Irp)
#003 {
#004     //获得设备扩展
#005     PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
#006     //获得 I/O 堆栈
#007     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
#008     #if DBG
#009     static char* irpname[] =
#010     {
#011         "IRP_MJ_CREATE",
#012         "IRP_MJ_CREATE_NAMED_PIPE",
#013         "IRP_MJ_CLOSE",
#014         "IRP_MJ_READ",
#015         "IRP_MJ_WRITE",
#016         "IRP_MJ_QUERY_INFORMATION",
#017         "IRP_MJ_SET_INFORMATION",
#018         "IRP_MJ_QUERY_EA",
#019         "IRP_MJ_SET_EA",
#020         "IRP_MJ_FLUSH_BUFFERS",
#021         "IRP_MJ_QUERY_VOLUME_INFORMATION",
#022         "IRP_MJ_SET_VOLUME_INFORMATION",
#023         "IRP_MJ_DIRECTORY_CONTROL",
#024         "IRP_MJ_FILE_SYSTEM_CONTROL",
#025         "IRP_MJ_DEVICE_CONTROL",
#026         "IRP_MJ_INTERNAL_DEVICE_CONTROL",
#027         "IRP_MJ_SHUTDOWN",
#028         "IRP_MJ_LOCK_CONTROL",
#029         "IRP_MJ_CLEANUP",
#030         "IRP_MJ_CREATE_MAILSLOT",
#031         "IRP_MJ_QUERY_SECURITY",
#032         "IRP_MJ_SET_SECURITY",
#033         "IRP_MJ_POWER",
#034         "IRP_MJ_SYSTEM_CONTROL",
#035         "IRP_MJ_DEVICE_CHANGE",
#036         "IRP_MJ_QUERY_QUOTA",
#037         "IRP_MJ_SET_QUOTA",
#038         "IRP_MJ_PNP",
#039     };
#040     //获得 IRP 的主代码
#041     UCHAR type = stack->MajorFunction;
#042     if (type == IRP_MJ SCSI)
#043     {
#044         return DispatchForSCSI(fido, Irp);
#045     }
#046     #endif
```

```

#047
#048     NTSTATUS status;
#049     //获取自旋锁
#050     status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
#051     //判断是否成功获取自旋锁
#052     if (!NT_SUCCESS(status))
#053         //结束 IRP 请求
#054         return CompleteRequest(Irp, status, 0);
#055     //略过当前 I/O 堆栈
#056     IoSkipCurrentIrpStackLocation(Irp);
#057     //调用底层驱动程序
#058     status = IoCallDriver(pdx->LowerDeviceObject, Irp);
#059     //释放自旋锁
#060     IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
#061     return status;
#062 ) //驱动程序的派遣函数

```

此段代码可以在配套光盘中本章的 FileFilter 目录下找到。

22.1.3 U 盘过滤驱动程序

本节介绍的例子是将一个 U 盘的驱动程序进行过滤，让 U 盘成为一个只读存储器。当 U 盘插入计算机时，系统会枚举出一个 USB 的 PDO，并将一个叫做 USBSTOR 的驱动程序作为 FDO 加载到 PDO 之上。USBSTOR 会另外创建一个物理设备，上面挂载磁盘驱动，其上再挂载 PartMgr 驱动（分区驱动）。上述过程归结为图 22-2 所示。

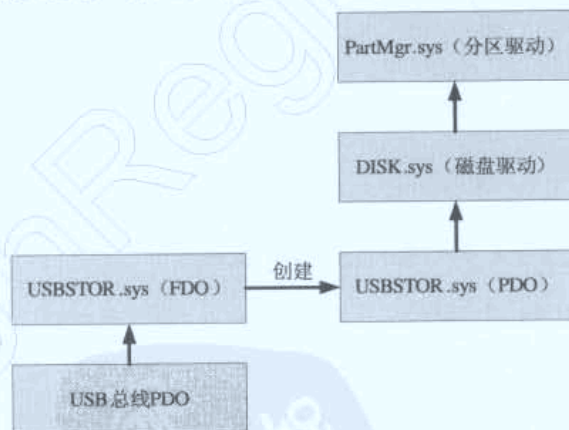


图 22-2 U 盘设备驱动拓扑

为了让编写的过滤驱动能让 U 盘变为只读状态，可以在 DISK.sys 和 USBSTOR.sys 驱动之间建立一个过滤驱动。笔者将这个驱动称为 MyFilter 驱动，和前面介绍的所有 WDM 驱动一样，该驱动需要在注册表中的 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\MyFilter 建立一系列表项。这里可以用 INF 文件来实现，和以往 INF 文件安装不同的是，只需用鼠标右键单击 INF 文件，并单击“安装”即可。

```

#001 [Version]
#002 Signature=$CHICAGO$

```



```

#003 Provider=%MFGNAME%
#004
#005 [DestinationDirs]
#006 DefaultDestDir=10,system32\drivers
#007 FiltJectCopyFiles=11
#008
#009 [SourceDisksFiles]
#010 MyFilter.sys=1
#011
#012 [SourceDisksNames]
#013 1=%INSTDISK%,,,MyFilter_Check
#014
#015 ;-----
#016 ; Windows 2000 Sections
#017 ;-----
#018
#019 [DefaultInstall.ntx86]
#020 CopyFiles=DriverCopyFiles,FiltJectCopyFiles
#021
#022 [DriverCopyFiles]
#023 MyFilter.sys,,,0x60 ; replace newer, suppress dialog
#024
#025 [DefaultInstall.ntx86.services]
#026 AddService=MyFilter,,FilterService
#027
#028 [FilterService]
#029 ServiceType=1
#030 StartType=3
#031 ErrorControl=1
#032 ServiceBinary=%10%\system32\drivers\MyFilter.sys
#033
#034 ;-----
#035 ; String Definitions
#036 ;-----
#037
#038 [Strings]
#039 MFGNAME="Zhangfan Software"
#040 INSTDISK="Zhangfan Disc"
#041 DESCRIPTION="Sample File Filter Driver"

```

此段代码可以在配套光盘中本章的 FileFilter 目录下找到。

22.1.4 过滤驱动程序加载方法一

为了加载过滤驱动程序，需要手动修改注册表。操作系统对于每插入一种 U 盘，都会在 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USBSTOR 位置上有相应的体现。笔者计算机里就曾插入过两种 U 盘。如图 22-3 所示。

在默认情况下，管理员不能修改这个位置的注册表项，但可以修改其权限。方法是：单击右键，选择“权限”菜单，将管理员（Administrator）加入安全列表，并选中“完全权限”复选框。如图 22-4 所示。

修改注册表权限后，就可以对该项目中的注册表进行修改了。根据不同的 U 盘，在不同的子目录下建立子键 LowerFilters。

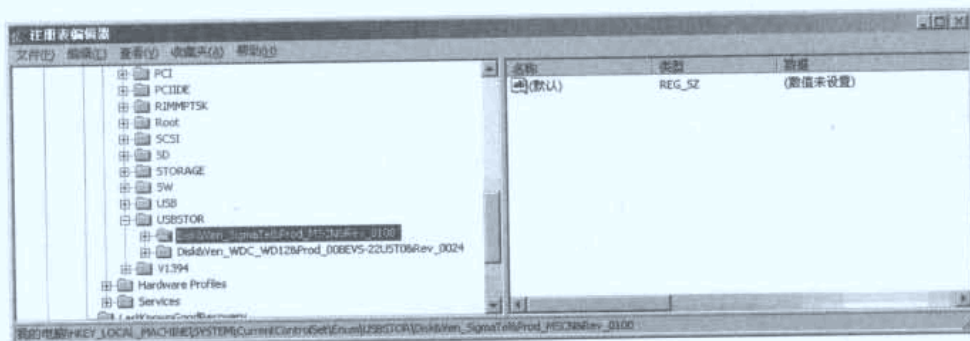


图 22-3 USBSTOR 所相关的注册表

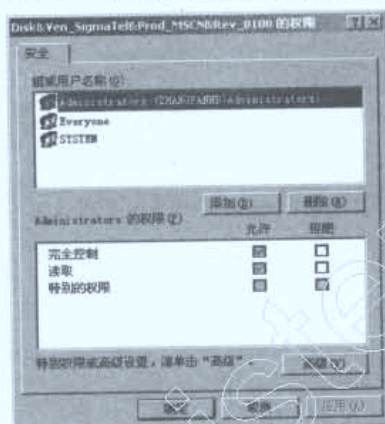


图 22-4 修改注册表权限

例如，笔者的子目录是 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USBSTOR\Disk&Ven_SigmaTel&Prod_MSCN&Rev_0100\0000A1054184410B&0 下，子键为过滤驱动的 Class 名称，这里是 MyFilter。如图 22-5 所示。

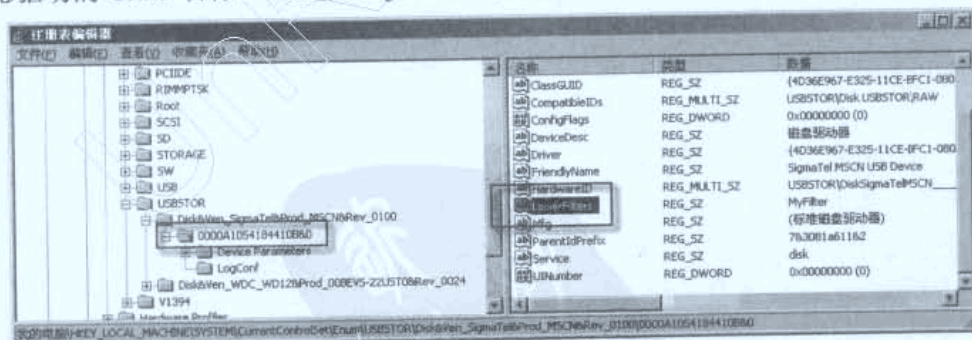


图 22-5 在 Enum 上加载过滤驱动

22.1.5 过滤驱动程序加载方法二

上面的方法是当特定 U 盘被插入时，操作系统就加载过滤驱动程序。一旦换了别的品牌 U 盘，还需要重新修改注册表。即使用同一个 U 盘，但如果换了别的插槽，还是需要

重新修改注册表。也就是说上述的方法只适用于特定的设备加载特定的过滤驱动程序。

还有另外一种方法，就是对同一类别驱动加载过滤驱动。例如，对于 U 盘这一类设备，其 ClassGUID 为 {4D36E967-E325-11CE-BFC1-08002BE10318}，可以在 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Class\{4D36E967-E325-11CE-BFC1-08002BE10318} 位置上进行设置，同样是创建子键 LowerFilters，子键为过滤驱动的 Class 名称，这里为 MyFilter。如图 22-6 所示。

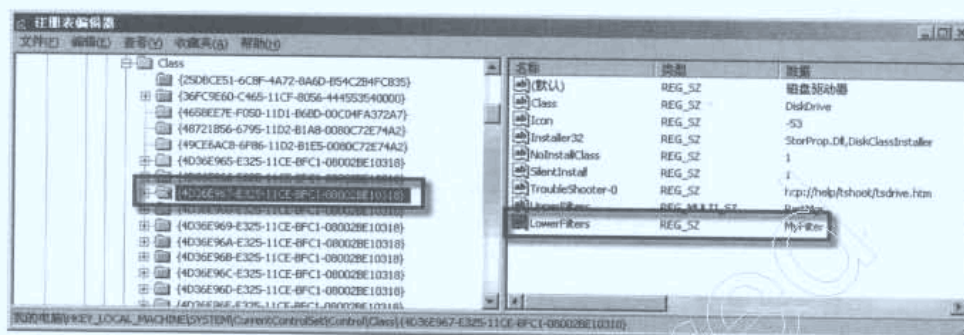


图 22-6 在 Class 项上加载过滤驱动

这样的好处是对于一类的设备进行了过滤，但同时带来一个问题，就是 ClassGUID 为 {4D36E967-E325-11CE-BFC1-08002BE10318} 代表所有的磁盘设备，也包括 IDE 硬盘，对于本过滤驱动一样会将其过滤，使 IDE 硬盘也变得只读，这当然不是大家所希望的。

本章后面的程序没有对 IDE 硬盘或者 USB 硬盘进行判断，读者可以对其稍加扩充。

22.1.6 过滤驱动程序的 AddDevice 例程

对于过滤驱动程序的 AddDevice 例程，需要创建一个过滤设备对象。这一步是通过 IoCreateDevice 内核函数实现的。然后将该过滤设备对象附加到设备栈上，并用设备扩展记录仅低于本层的驱动对象地址。这一步是通过调用 IoAttachDeviceToDeviceStack 内核函数实现的。

由于不知道底层驱动是直接读取设备还是缓冲区读取设备，因此，对于过滤设备的标志应该将两个都进行设置，即 DO_DIRECT_IO | DO_BUFFERED_IO。

```
#001 NTSTATUS AddDevice(IN PDRIVER_OBJECT DriverObject, IN PDEVICE_OBJECT pdo)
#002 {
#003     //确保当前函数运行在分页内存中
#004     PAGED_CODE();
#005     NTSTATUS status;
#006
#007     PDEVICE_OBJECT fido;
#008     //创建设备对象
#009     status = IoCreateDevice(DriverObject, sizeof(DEVICE_EXTENSION), NULL,
#010         GetDeviceTypeToUse(pdo), 0, FALSE, &fido);
#011     //判断是否成功创建设备对象
#012     if (!NT_SUCCESS(status))
```

```

#013 {
#014     KdPrint((DRIVERNAME " - IoCreateDevice failed - %X\n", status));
#015     //如果不能成功创建设备对象就返回
#016     return status;
#017 }
#018 //获得设备扩展
#019 PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
#020
#021 do
#022 {
#023     //初始化自旋锁
#024     IoInitializeRemoveLock(&pdx->RemoveLock, 0, 0, 0);
#025     pdx->DeviceObject = fido;
#026     pdx->Pdo = pdo;
#027     //将过滤驱动附加在底层驱动之上
#028     PDEVICE_OBJECT fdo = IoAttachDeviceToDeviceStack(fido, pdo);
#029     if (!fdo)
#030     {
#031         // “附加” 操作失败
#032         KdPrint((DRIVERNAME " - IoAttachDeviceToDeviceStack failed\n"));
#033         status = STATUS_DEVICE_REMOVED;
#034         break;
#035     }
#036     //记录底层驱动
#037     pdx->LowerDeviceObject = fdo;
#038     //由于不知道底层驱动是直接 I/O 还是 BufferIO, 因此将标志都置上
#039     fido->Flags |= fdo->Flags & (DO_DIRECT_IO | DO_BUFFERED_IO | DO_POWER_
PAGABLE);
#040     fido->Flags &= ~DO_DEVICE_INITIALIZING;
#041     while (FALSE);
#042     if (!NT_SUCCESS(status))
#043     {
#044         //如果没有成功, 则从设备堆栈中删除设备
#045         if (pdx->LowerDeviceObject)
#046             IoDetachDevice(pdx->LowerDeviceObject);
#047         IoDeleteDevice(fido);
#048     }
#049     return status;
#050 }

```

此段代码可以在配套光盘中本章的 FileFilter 目录下找到。

22.1.7 磁盘命令过滤

对于磁盘过滤的关键在于 IRP_MJ SCSI 的这个 IRP 的截获, 其实 IRP_MJ SCSI 就是 IRP_MJ_INTERNAL_DEVICE_CONTROL 的一个别名。在 DISK.sys 和 USBSTOR.sys 之间传递的是标准 SCSI 指令。

在 IRP_MJ SCSI 的派遣函数中, 首先将 IRP 发送到底层驱动, 然后设置完成例程。其主要是希望在完成例程中修改底层驱动所做的处理。

```

#001 #pragma LOCKEDCODE
#002 NTSTATUS DispatchForSCSI(IN PDEVICE_OBJECT fido, IN PIRP Irp)
#003 {

```



```

#004    //获得设备扩展
#005    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
#006    //获得当前 I/O 堆栈
#007    PIO_STACK_LOCATION irpStack = IoGetCurrentIrpStackLocation(Irp);
#008
#009    // 向下层设备发送 IRP
#010    NTSTATUS status;
#011    //获取自旋锁
#012    status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
#013    if (!NT_SUCCESS(status))
#014        return CompleteRequest(Irp, status, 0);
#015    //将当前 I/O 堆栈复制至下层 I/O 堆栈
#016    IoCopyCurrentIrpStackLocationToNext(Irp);
#017    //设置完成例程
#018    IoSetCompletionRoutine( Irp,
#019                            USBSCSICompletion,
#020                            NULL,
#021                            TRUE,
#022                            TRUE,
#023                            TRUE );
#024    //调用底层设备对象行
#025    status = IoCallDriver(pdx->LowerDeviceObject, Irp);
#026    //释放自旋锁
#027    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
#028    return status;
#029 }

```

对于 SCSI 命令，在 DDK 中的 srb.h 和 scsi.h 中相应定义，对于不同的 SCSI，分为如下的几种：

```

#define SCSIOP_COPY                0x18
#define SCSIOP_ERASE                0x19
#define SCSIOP_MODE_SENSE          0x1A
#define SCSIOP_START_STOP_UNIT     0x1B
#define SCSIOP_STOP_PRINT          0x1B
#define SCSIOP_LOAD_UNLOAD         0x1B
#define SCSIOP_RECEIVE_DIAGNOSTIC  0x1C
#define SCSIOP_SEND_DIAGNOSTIC     0x1D
#define SCSIOP_MEDIUM_REMOVAL      0x1E
#define SCSIOP_READ_FORMATTED_CAPACITY 0x23
#define SCSIOP_READ_CAPACITY       0x25
#define SCSIOP_READ                 0x28
#define SCSIOP_WRITE                0x2A
#define SCSIOP_SEEK                 0x2B
#define SCSIOP_LOCATE               0x2B
#define SCSIOP_POSITION_TO_ELEMENT  0x2B
#define SCSIOP_WRITE_VERIFY         0x2E
#define SCSIOP_VERIFY               0x2F
#define SCSIOP_SEARCH_DATA_HIGH     0x30
#define SCSIOP_SEARCH_DATA_EQUAL    0x31
#define SCSIOP_SEARCH_DATA_LOW      0x32
#define SCSIOP_SET_LIMITS           0x33

```

例如，如果想让设备变成只读，只需将 SCSI 命令中的 SCSIOP_WRITE 请求设置为失败。

所有对 U 盘的写请求，最终都会变成 SCSIOP_WRITE 请求。如果将该请求设置为失败，

该设备就会变成只读设备。然而在直接将 SCSIOP_WRITE 请求设置为失败，当对磁盘写一个文件时，会很久才会有系统提示失败。这大概是因为操作系统一旦发现 SCSIOP_WRITE 请求失败，会多次发送这个请求，直到超时为止。

还有一种更为简便的方法，即 SCSIOP_MODE_SENSE 请求时，设置 MODE_DSP_WRITE_PROTECT 位，使其变为只读设备。

这种操作类似只有只读开关的 U 盘。当只读开关关闭的时候，该 U 盘变得只读。

修改 SCSIOP_MODE_SENSE 请求，应该在 IRP_MJ SCSI 派遣例程的完成例程中实现，代码如下：

```
#001 NTSTATUS
#002 USBSCSICompletion( IN PDEVICE_OBJECT DeviceObject,
#003                     IN PIRP Irp,
#004                     IN PVOID Context )
#005 {
#006     //获取设备扩展
#007     PDEVICE_EXTENSION pdx = ( PDEVICE_EXTENSION )
#008                             DeviceObject->DeviceExtension;
#009     //获取自旋锁
#010     IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
#011     //获得当前 IO 堆栈
#012     PIO_STACK_LOCATION irpStack = IoGetCurrentIrpStackLocation( Irp );
#013     //获取当前 IRP 处理状态
#014     PSCSI_REQUEST_BLOCK CurSrb=irpStack->Parameters.Scsi.Srb;
#015     PCDB cdb = (PCDB)CurSrb->Cdb;
#016     //获取操作码
#017     UCHAR opCode=cdb->CDB6GENERIC.OperationCode;
#018     //判断是否是 SCSIOP_MODE_SENSE 操作
#019     if(opCode==SCSIOP_MODE_SENSE && CurSrb->DataBuffer
#020         && CurSrb->DataTransferLength >=
#021         sizeof(MODE_PARAMETER_HEADER))
#022     {
#023         KdPrint(("SCSIOP_MODE_SENSE coming!\n"));
#024         //得到模式参数头
#025         PMODE_PARAMETER_HEADER modeData =
(PMODE_PARAMETER_HEADER)CurSrb->DataBuffer;
#026         //设置 U 盘为只读磁盘
#027         modeData->DeviceSpecificParameter |= MODE_DSP_WRITE_PROTECT;
#028     }
#029     //判断是否需要挂起
#030     if ( Irp->PendingReturned )
#031     {
#032         //挂起 IRP
#033         IoMarkIrpPending( Irp );
#034     }
#035     //释放自旋锁
#036     IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
#037
#038     return Irp->IoStatus.Status ;
#039 }
```

此段代码可以在配套光盘中本章的 FileFilter 目录下找到。

如果上述过滤驱动加载成功,当写磁盘的时候,Windows 会报告错误,如图 22-7 所示。

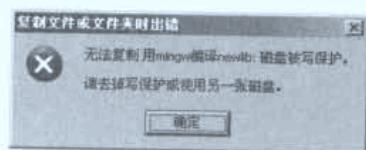


图 22-7 将 U 盘成功改成只读设备

22.2 NT 式过滤驱动程序

在上节中介绍的是 WDM 过滤驱动,安装它时需要修改注册表。本节介绍的 NT 式过滤驱动,无须修改注册表。它通过驱动名直接寻找到需要过滤的驱动设备的指针,然后将自己挂载在上面。本节主要针对 NT 式过滤驱动进行讲解。

22.2.1 NT 式过滤驱动程序

编写 NT 式过滤驱动更加简便。因为 WDM 是通过注册表指定挂载哪种驱动的。而对于 NT 式驱动程序可以通过寻找不同的设备对象指定。例如选择键盘驱动,再将自身的设备对象挂载在键盘驱动之上。

笔者前面曾多次提到,WDM 驱动无非是微软在 NT 式驱动之上进行了扩充,过滤驱动也不例外。

本节将介绍如何开发一种键盘过滤驱动,即对键盘的操作进行过滤。这在黑客或木马程序中经常被用到,例如可以通过用户的敲击键盘记录,用来分析用户密码等。

本节的例子来自 Mark Russinovich 编写的 ctrl2cap。他是一位资深的 Windows 内核的专家。ctrl2cap 程序主要是将自己的过滤驱动加载到键盘驱动之上,可以用来监视键盘记录,也可以改变键盘的输入。

在 Windows 中,键盘驱动所创建的设备对象叫做\Device\KeyboardClass0。过滤驱动自己首先创建一个设备对象,并设置派遣函数,然后将自己挂在\Device\KeyboardClass0 之上即可。键盘设备对象如图 22-8 所示。



图 22-8 在 WinObj 中观察键盘设备

22.2.2 NT 过滤驱动的入口函数

NT 式过滤驱动的入口函数和 WDM 的过滤驱动一样，都是将所有 IRP 进行过滤。为了保证能接收上层传下来所有的 IRP，要对所有 IRP 进行过滤。

```
#001 NTSTATUS DriverEntry(
#002     IN PDRIVER_OBJECT DriverObject,
#003     IN PUNICODE_STRING RegistryPath
#004 )
#005 {
#006     ULONG i;
#007     DbgPrint (("Ctrl2cap.SYS: entering DriverEntry\n"));
#008
#009     //对所有的 IRP 进行过滤
#010     for (i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
#011     {
#012         //将所有的 IRP 派遣函数都设置为 Ctrl2capDispatchGeneral
#013         DriverObject->MajorFunction[i] = Ctrl2capDispatchGeneral;
#014     }
#015     //设置 IRP_MJ_READ 的派遣函数
#016     DriverObject->MajorFunction[IRP_MJ_READ] = Ctrl2capDispatchRead;
#017
#018     //Ctrl2capInit 负责挂载过滤驱动
#019     return Ctrl2capInit( DriverObject );
#020 }
```

此段代码可以在配套光盘中本章的 KeyFilter 目录下找到。

22.2.3 挂载过滤驱动

挂载过滤驱动的步骤被封装在 Ctrl2capInit 函数里。这里通过构造一个 UNICODE 字符串，该字符串就是键盘驱动的设备名。然后创建一个新的设备对象，并将自己附加在键盘设备驱动之上。

一旦附加成功，所有对键盘的读写请求，都会经过这个过滤设备。

```
#001 NTSTATUS Ctrl2capInit(
#002     IN PDRIVER_OBJECT DriverObject
#003 )
#004 {
#005     CCHAR          ntNameBuffer[64];
#006     STRING          ntNameString;
#007     UNICODE_STRING ntUnicodeString;
#008     PDEVICE_OBJECT device;
#009     NTSTATUS        status;
#010     PDEVICE_EXTENSION devExt;
#011     WCHAR          messageBuffer[] = L"Ctrl2cap Initialized\n";
#012     UNICODE_STRING messageUnicodeString;
#013
#014     //构造 UNICODE 字符串
#015     sprintf( ntNameBuffer, "\\Device\\KeyboardClass0" );
#016     RtlInitAnsiString( &ntNameString, ntNameBuffer );
#017     RtlAnsiStringToUnicodeString( &ntUnicodeString, &ntNameString, TRUE );
```



```

#018
#019 //创建设备对象
#020     status = IoCreateDevice( DriverObject,
#021                             sizeof(DEVICE_EXTENSION),
#022                             NULL,
#023                             FILE_DEVICE_KEYBOARD,
#024                             0,
#025                             FALSE,
#026                             &device );
#027 //判断是否成功创建设备对象
#028     if( !NT_SUCCESS(status) ) {
#029         DbgPrint(("Ctrl2cap: Keyboard hook failed to create device!\n"));
#030         RtlFreeUnicodeString( &antUnicodeString );
#031         return STATUS_SUCCESS;
#032     }
#033 //将内存清零
#034     RtlZeroMemory(device->DeviceExtension, sizeof(DEVICE_EXTENSION));
#035 //获得设备扩展
#036     devExt = (PDEVICE_EXTENSION) device->DeviceExtension;
#037
#038 //设置对象标志
#039     device->Flags |= DO_BUFFERED_IO;
#040     device->Flags &= ~DO_DEVICE_INITIALIZING;
#041
#042 //这里是重点，附加过滤驱动
#043     status = IoAttachDevice( device, &antUnicodeString, &devExt->TopOfStack );
#044 //判断操作是否成功
#045     if( !NT_SUCCESS(status) ) {
#046         DbgPrint(("Ctrl2cap: Connect with keyboard failed!\n"));
#047         //删除设备
#048         IoDeleteDevice( device );
#049         //释放字符串
#050         RtlFreeUnicodeString( &antUnicodeString );
#051         return STATUS_SUCCESS;
#052     }
#053
#054 //释放 UNICODE 字符串所占用的内存
#055     RtlFreeUnicodeString( &antUnicodeString );
#056     DbgPrint(("Ctrl2cap: Successfully connected to keyboard device\n"));
#057     RtlInitUnicodeString ( &messageUnicodeString,
#058                             messageBuffer );
#059     ZwDisplayString( &messageUnicodeString );
#060     return STATUS_SUCCESS;
#061 }

```

此段代码可以在配套光盘中本章的 KeyFilter 目录下找到。

22.2.4 过滤键盘读操作

这个程序监视用户所有的键盘操作。每一次键盘操作，都会间接地向键盘驱动发送一个 IRP_MJ_READ 请求，因此可以监视 IRP_MJ_READ 来达到这个目的。

另外，当用户敲击大写键后，过滤驱动这个动作将被改写成敲击左 Ctrl 键盘，也就是说大写键被解释成了左 Ctrl 键。其过程是：首先在 IRP_MJ_READ 的派遣例程中设置完成

例程，通过完成例程改变 IRP 的设置。以下是 IRP_MJ_READ 的派遣例程：

```
#001 NTSTATUS Ctrl2capDispatchRead(
#002     IN PDEVICE_OBJECT DeviceObject,
#003     IN PIRP Irp )
#004 {
#005     PDEVICE_EXTENSION devExt;
#006     PIO_STACK_LOCATION currentIrpStack;
#007     PIO_STACK_LOCATION nextIrpStack;
#008     //获取设备扩展
#009     devExt = (PDEVICE_EXTENSION) DeviceObject->DeviceExtension;
#010     //获取当前 IO 堆栈
#011     currentIrpStack = IoGetCurrentIrpStackLocation(Irp);
#012     //得到下一层 IO
#013     nextIrpStack = IoGetNextIrpStackLocation(Irp);
#014
#015     *nextIrpStack = *currentIrpStack;
#016     //设置完成例程
#017     IoSetCompletionRoutine( Irp, Ctrl2capReadComplete,
#018                             DeviceObject, TRUE, TRUE, TRUE );
#019
#020     //调用底层驱动
#021     return IoCallDriver( devExt->TopOfStack, Irp );
#022 }
```

此段代码可以在配套光盘中本章的 KeyFilter 目录下找到。

在完成上述例程中，主要是为了修改已经从底层驱动返回来的结果。例如，当底层驱动返回来的键盘扫描码是大写键（Caps Lock 键）时，在完成例程中强迫将其改成 Ctrl 键，从而完成修改 IRP 的目的。

```
#001 NTSTATUS Ctrl2capReadComplete(
#002     IN PDEVICE_OBJECT DeviceObject,
#003     IN PIRP Irp,
#004     IN PVOID Context
#005 )
#006 {
#007     PIO_STACK_LOCATION IrpSp;
#008     PKEYBOARD_INPUT_DATA KeyData;
#009     int numKeys, i;
#010
#011     //获取当前设备堆栈
#012     IrpSp = IoGetCurrentIrpStackLocation( Irp );
#013     if( NT_SUCCESS( Irp->IoStatus.Status ) ) {
#014
#015         //获取 IRP 中的数据
#016         KeyData = Irp->AssociatedIrp.SystemBuffer;
#017         numKeys = Irp->IoStatus.Information / sizeof(KEYBOARD_INPUT_DATA);
#018
#019         for( i = 0; i < numKeys; i++ ) {
#020
#021             DbgPrint(("ScanCode: %x ", KeyData[i].MakeCode ));
#022             DbgPrint((" %s\n", KeyData[i].Flags ? "Up" : "Down" ));
#023             //修改键盘扫描码
#024             if( KeyData[i].MakeCode == CAPS_LOCK ) {
#025
#026                 KeyData[i].MakeCode = LCONTROL;
```



```
#027     }  
#028     }  
#029     }  
#030     //如果需要阻塞,调用 IoMarkIrpPending  
#031     if( Irp->PendingReturned )  
#032     {  
#033         //挂起 IRP  
#034         IoMarkIrpPending( Irp );  
#035     }  
#036     return Irp->IoStatus.Status;  
#037 }
```

此段代码可以在配套光盘中本章的 KeyFilter 目录下找到。

本节实例没有提供 VC 编译环境,请进入到 DDK 编译环境下进行编译。另外加载 NT 驱动可以利用以前介绍的多种方法,其中最简单的是使用工具 DriverMonitor 加载。

22.3 小结

本章主要是介绍过滤驱动程序的编写。过滤驱动可以是 WDM 驱动,也可以是 NT 式驱动。

如果是 WDM 驱动需要通过注册表记录指定加载的过滤驱动,操作系统会读取这些值完成加载,其可以是高层过滤,也可以是低层过滤驱动。而 NT 过滤驱动会更加灵活,不用设置注册表,直接在内存中寻找设备对象,然后自行创建过滤驱动并将自己附加在这个驱动之上。

过滤驱动的入口函数需要将所有的 IRP 都设置派遣例程。因为过滤驱动要保证所有上层传递下来的 IRP 都能够接收到。如果想修改某个 IRP 的处理,例如键盘读取输入的键盘扫描码,可以在派遣例程中设置完成例程,在完成例程中修改 IRP 的处理。

第 23 章 高级调试技巧

本章将介绍一些 Windows 开发驱动的高级调试技巧。在驱动程序的开发中，经常会遇到系统崩溃的情况，我们很难像用 VC 的调试器那样单步调试程序。但是还是有一些高级驱动程序调试技巧，可以帮助程序员找出驱动程序中的 Bug。另外，利用一些第三方工具软件，也可以帮助程序员找到驱动程序中的漏洞，从而提高开发效率。

23.1 一般性调试技巧

一般性调试技巧包括打印调试信息、查看 dump 文件等。这些方法在调试驱动中比较常见，使用也比较简单。

23.1.1 打印调试信息

对于开发驱动程序来说，最简单的调试方法就是打印出调试信息。因为驱动程序很难像应用程序那样由一般的调试器调试，因此，打印调试信息是运用最为广泛的调试技巧。

打印调试信息需要使用 DbgPrint 函数，使用方法类似于 printf 函数。笔者在前面已经有所介绍，但是这里推荐使用 KdPrint，它不是一个真正的函数，而是一个宏。在 Check 版本中，它对应着 DbgPrint 函数，而在 Free 版本中，它不做任何操作。因此，它很像 MFC 中的 TRACE 宏。

对于查看 log 信息，可以使用 Dbgview 工具软件，也可以使用 DriverMonitor 软件。

23.1.2 存储 dump 信息

打印调试信息，可以满足大部分的调试需要，然而有时驱动程序员经常遇到蓝屏死机的困扰。这样即便打印了调试信息，系统崩溃或重启了，驱动程序员也无法看到。这时程

Windows 驱动开发技术详解

程序员需要设置 Dump 信息。所谓 Dump 信息就是在系统崩溃之前，操作系统会将当前的调用堆栈记录成一个 Dump 文件，这个文件对于以后的分析极为有用。首先要确定计算机是否让操作系统存储了 Dump 信息。其方法如下。

右键单击“我的电脑”，选择“属性”，弹出“系统属性”对话框。在对话框中，单击“高级”选项卡，然后单击“设置”按钮（第三个），弹出一个对话框，设置写入调试信息。如图 23-1 所示。

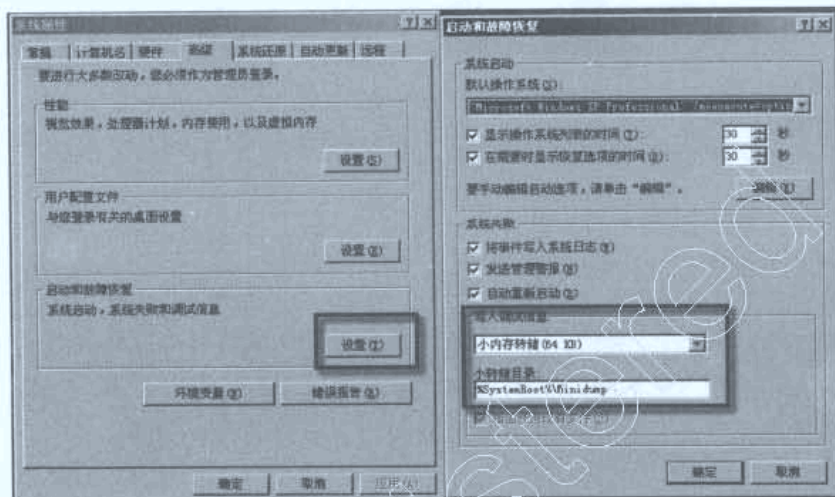


图 23-1 设置内存转储文件

读者可以选择“小内存转储”或者“完全内存转储”。其中小内存转储只存储 64KB 的存储信息，信息量少，而“完全内存转储”则将整个 4GB 的虚拟内存全部存储下来，这样虽然信息完整，但很耗时。对于一般情况来说，选择“小内存转储”即可。

引起蓝屏死机现象可能有多种情况。为了给读者演示蓝屏死机现象，笔者不得不做一点“牺牲”，编写一个错误示例，引起系统蓝屏。最简单的办法就是在程序代码中加入一行 `ASSERT(0)`，从而产生一个断言，继而会产生一个异常。可以说所有蓝屏都是由各类异常引起的，最常见的就是读写空指针、读写不可操作内存区域等。

笔者在程序代码中加入一行 `ASSERT(0)`，它在 check 版本中等效于调用 `RtlAssert` 函数，而在 Free 版本中不做任何操作。因此 check 版本的 `ASSERT(0)` 会引发蓝屏死机。本章示例光盘的 `ErrorTest` 就加入过这么一行，读者可以加载这个驱动，从而引发蓝屏。笔者在此声明：对进行此操作所引发的文件丢失等现象不承担任何法律责任。

23.1.3 使用 WinDbg 调试工具

Dump 信息，必须用专用的工具软件查看，如 WinDbg 等工具。在使用 WinDbg 工具时，最好下载系统的符号表。系统的 DLL、EXE、SYS、AX 等文件，都会有一个相应 PDB 文件，这就是符号表文件。这个 PDB 是微软在编译 Windows 时产生的符号表文件，有了

这些信息，就可以知道某段代码属于哪个系统文件中的函数了。

首先，在系统环境变量中设置 `_NT_SYMBOL_PATH` 环境变量，内容可以设置为 `SRV*d:\symcache*http://msdl.microsoft.com/download/symbols`。其含义是以 `http://msdl.microsoft.com/download/symbols` 作为符号表服务器，一旦需要就可以去这个网站下载符号表，下载后将其存储在 `d:\symcache` 目录里。读者可以根据需要修改这个环境变量。如图 23-2 所示。

其次，安装 WinDbg 软件。该软件是微软免费对程序员开放的，因此可以到微软的网站上下下载获取。WinDbg 软件可以调试应用程序，也可以调试内核程序，同时还可以观察蓝屏后 Dump 显示出来的信息。

这里重点讲述一下调试 Dump 信息。选择菜单“File”|“Open Crash Dump”，如图 23-3 所示。然后选择 Dump 的文件。这些文件一般存放在 `C:\WINDOWS\Minidump` 目录下。



图 23-2 设置 `_NT_SYMBOL_PATH` 环境变量

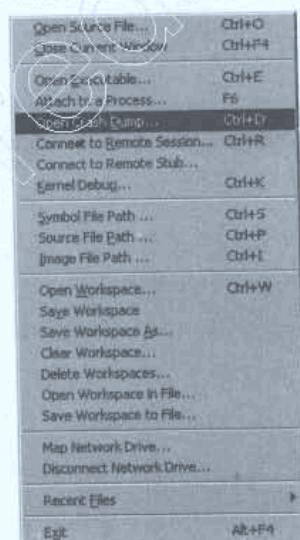


图 23-3 用 WinDbg 打开 Dump 文件

第一次使用 WinDbg 调试 Dump 文件，会等待很长一段时间。这主要是由于软件会自动到微软网站去下载符号表的缘故，即一些 PDB 文件。例如，内核重要文件 `ntkrnlpa.exe` 就会有对应文件 `ntkrnlpa.pdb`。这些文件下载的时间会相对较长，可是以后再次使用就方便了。程序员自己编写的驱动程序是无法到微软网站下载符号表的，这就需要程序员自己提供了。

一般情况下，不管是用 VC 提供的编译器，还是使用 DDK 提供的编译器，编译驱动程序的时候，在生成 `sys` 文件的同时，还会生成一个 PDB 文件，即符号表文件。初次打开 Dump 文件后，系统会提示是 `HelloWDM.sys` 文件中的某个地方出错，并且报告无法找到此文件。这时就需要 WinDbg 提供这个文件的位置，可以通过选择“File”/“Image File Path”来将 `sys` 文件的目录设置好。`sys` 文件会记录 PDB 文件的位置，从而找到符号表。正确加载符号表后，选择堆栈调用列表，如图 23-4 所示。

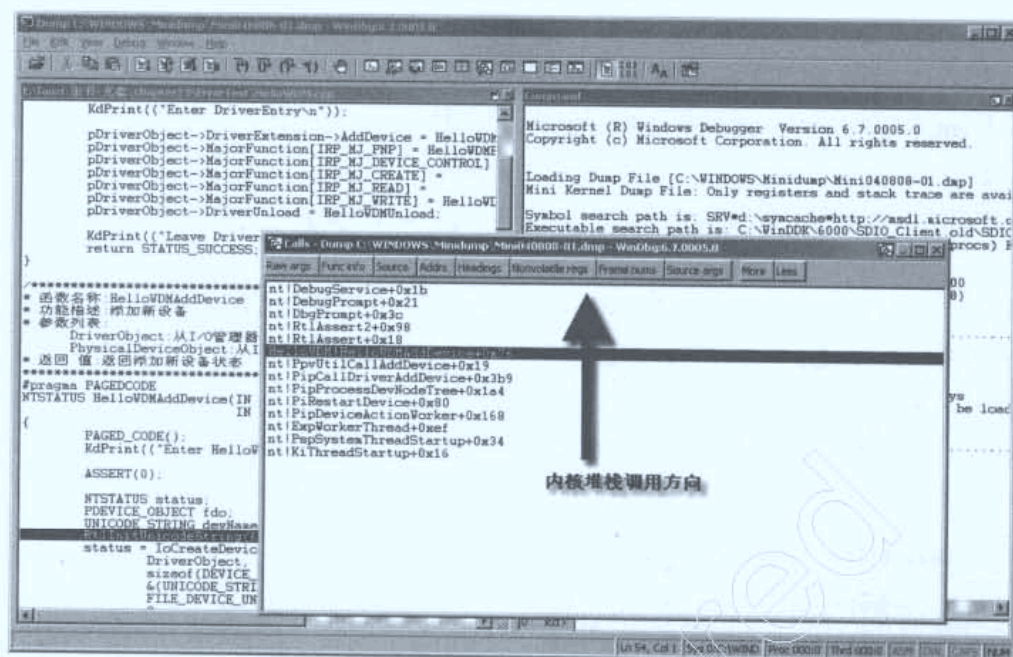


图 23-4 用 WinDbg 调试 Dump 文件

从图 23-4 中，可以清楚地看到在内核中的调用关系，这种调用关系是系统崩溃前调用关系的一个快照(Snap)，因此可以分析出是调用中的哪句话导致了系统崩溃。一般而言，系统的异常，如内存读写异常等错误往往导致系统崩溃。

此处是在内核中的 nt!KiThreadStartup 函数调用了 nt!PspSystemThreadStartup，又调用了 nt!ExpWorkerThread，然后一层层调用，直到 HelloWDM!HelloWDMAddDevice 函数。可以看出，HelloWDM!HelloWDMAddDevice 的代码偏移 0x76 处调用了 nt!RtlAssert+0x18。

双击“HelloWDM!HelloWDMAddDevice+0x76”，可以弹出具体对应的文件，这就是 WinDbg 强大的地方。很明显这里有一句 ASSERT(0)，它其实是一句宏，替换成 RtlAssert 内核函数。至此，调试找到了问题的根源！

当然，如果没有正确加载系统的符号表，就无法看到 RtlAssert 的名称，会被一堆数字所代替。同理，如果没有正确加载出错驱动程序的符号表，就无法看到 HelloWDMAddDevice 的名称，也会被一堆数字所代替。其实对于一个经验丰富的驱动程序开发人员，这些现象也能分析出来。

另外，在编译驱动程序的时候，还可以在编译器选项进行设置，使之编译完生成一个符号列表文件，这个文件列举了某个文件对应着十六进制的位置，因此可以分析出大概是哪个语句导致的错误。

23.2 高级内核调试技巧

高级内核调试技巧一般用 WinDbg 进行双机调试，一台作为主机，一台作为目标机。

如果使用虚拟机软件,如 VMWare 或 Virtual PC 等,可以在计算机中模拟出目标机,这样一台计算机就可以进行内核调试。本节主要介绍如何用 VMware 软件虚拟出目标机,并用 WinDbg 进行双机调试。

23.2.1 安装 VMWare

很多驱动或内核程序员都希望调试驱动程序时,能像 Ring3 层的程序一样单步执行,在断点的时候,观察变量,修改内存,完全知道程序的运行情况。这完全是可能的,但需要费一些周折。因为,驱动程序运行在 Ring0 层程序,调试软件不可能是 Ring3 层的程序,这与无法在二维空间观察三维空间是一样的道理。

著名软件公司 NuMega 有一款软件,叫 SoftICE,它曾经是无数内核前辈推崇的工具,可以在本机上直接进入内核状态,然后进行调试。但是它对 Windows XP SP2 的支持非常不好,经常容易导致系统运行不稳定。另外,该公司在 2006 年宣布停止 SoftICE 的开发,所有对 SoftICE 的技术支持也在 2007 年结束了。

这里要介绍的就是 WinDbg 软件,它可以调试应用软件(Ring3 层程序)、驱动程序(Ring0 层程序)和蓝屏存储文件。但是在调试驱动程序时,WinDbg 需要两台计算机调试,一台作为主机(HOST),一台作为目标机(TARGET),用一根串口线或者一条 1394 的电缆将两台计算机连接起来。对于有嵌入式开发经验的读者可能觉得很熟悉,因为很多嵌入式软件调试的模式都与这个类似。

但是对于一些软件开发人员来说,把两台计算机进行联机调试显得很麻烦。这里介绍一种简便的方法,即采用虚拟机软件虚拟出一个 PC 来,将虚拟的 PC 当做目标 PC,而真正的 PC 当做主 PC。虚拟机软件可以选择 VMware、Virtual PC 等。这里以 VMWare 为例进行讲解。

要安装尽量高版本的 VMWare。首先在 VMWare 中安装一个虚拟的 Windows,读者以安装 Windows XP SP2 为例。由于是虚拟的 PC,并且调试需要一个串口,因此利用虚拟机的虚拟特性,定制一个虚拟串口,其设置如图 23-5 所示。

使用“虚拟串口”使其建立一条虚拟的串口,一端是虚拟 PC 的串口 COM1,另一端是 Windows 上的管道。另外,在虚拟机里找到启动配置文件 boot.ini,在[operating systems]段中加入一行以调试模式启动的配置:

```
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional
Debug" /fastdetect /debugport=com1 /baudrate=115200
```

另外,创建一个新的 WinDbg 的快捷方式,其内容如下:

```
"C:\Program Files\Debugging Tools for Windows\windbg.exe" -b -k com:pipe,port=
\\.\pipe\com_1, resets=0
```

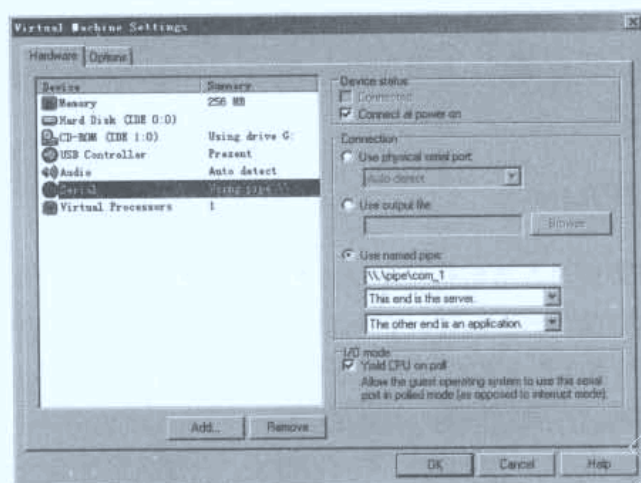



图 23-5 设置虚拟串口

23.2.2 在虚拟机上加载驱动程序

在进行上述设置后，就可以进行程序调试了。

首先运行 VMware 中的虚拟机软件，然后启动新建立的 WinDbg 快捷方式。不过 WinDbg 很快就会停止运行，整个虚拟机也会停止下来。这时就需要让虚拟机中的 Windows 暂停下来，以便可以用 Step in、Step out 等方式调试。由于此时在 Windows 的内核下，且没有 Windows 的源代码，所以看到的都是汇编命令，如图 23-6 所示。

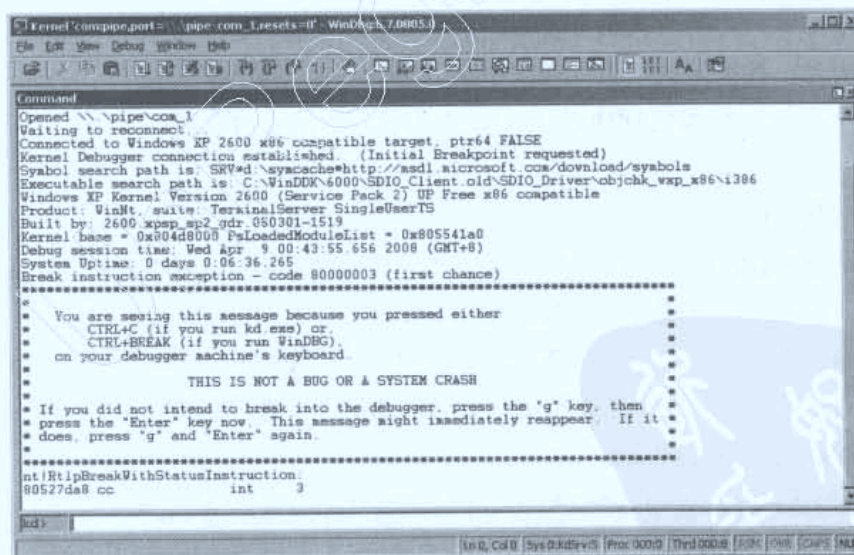


图 23-6 用 WinDbg 和 VMware 联合调试

这时候按 F5 键或者在命令行中输入 g，或者单击“运行”图标，都可以让系统运行起来。这里的操作非常类似于 VC 的调试界面。下一步需要做的事情是把以上编译的驱动程序放到虚拟机上运行。

这里首先需要解决的问题是如何将文件复制到虚拟机上。为此，VMWare 为用户提供了一个工具，叫做 VMWare Tools。在 VMWare 上选择菜单“Install VMWare Tools”，虚拟机中会自动安装一个配套软件插件。

然后在 VMWare 的工程中，选择“属性”菜单，单击“选项”对话框，如图 23-7 所示。然后选择“Add”按钮，可以为虚拟机设置一个“网络共享”，在图 23-7 中可以看出已经为虚拟机装好共享。

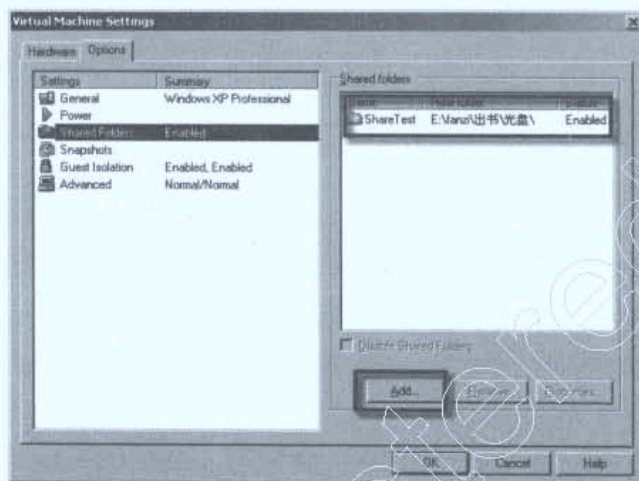


图 23-7 设置共享目录

当为虚拟机设置好共享后，就可以在虚拟机上使用了，如图 23-8 所示。

在虚拟机里的资源管理器中输入 \\host\，就可以找到共享目录。这样就可以将 PC 中的文件复制到虚拟机的目录里。

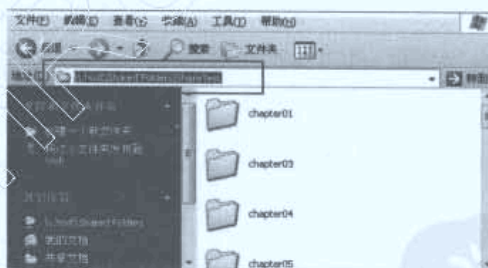


图 23-8 打开共享目录

此时读者就可以方便地在虚拟机里加载需要调试的驱动程序了。

下一节，笔者将介绍如何用 WinDbg 和 VMWare 联合进行驱动程序的单步调试及其他工作。

23.2.3 VMWare 和 WinDbg 联合调试驱动程序

在进行上述配置后，就可以用 VMWare 和 WinDbg 进行联合调试了。

Windows 驱动开发技术详解

这里首先需要了解一下 WinDbg 的简单使用方法。WinDbg 大部分调试都通过命令行方式,即通过输入命令实现。

(1) 断点设置: 对于设置的一般断点用 bp, 后面接函数名或者地址名。如 bp HelloWDM!DriveEntry 或者 bp 0x12345678。对于清除断点用 bc。例如, bc *代表清除所有断点, bc 1 代表第 1 号清除断点。对于没有加载符号表的驱动程序, 可以使用延迟设置断点, 用 bu。例如, bu HelloWDM!DriveEntry。

(2) 符号表设置: 如果符号表设置成功, 就不需要再设置了。但是符号表有时需要手动设置一下。重新加载符号表使用 .reload 命令。列出当前加载的符号表使用 .lm 命令。手动加载符号表使用 .ld 命令。

对于笔者而言, 是在虚拟机上运行下列一些列命令:

```
.reload //重新加载符号表
.lm//查看是否正确加载了驱动的符号表
.bu HelloWDM!DriverEntry//对于 HelloWDM 驱动中的 DriverEntry 函数设置断点
```

这时刚才加载的驱动程序会在虚拟机中暂停下来, 同时 WinDbg 也停在断点位置, 另外会有相应的源代码与之匹配。此时, 就可以单步执行源代码(按 F8 键)或者查看内存、变量等信息。如图 23-9 所示。

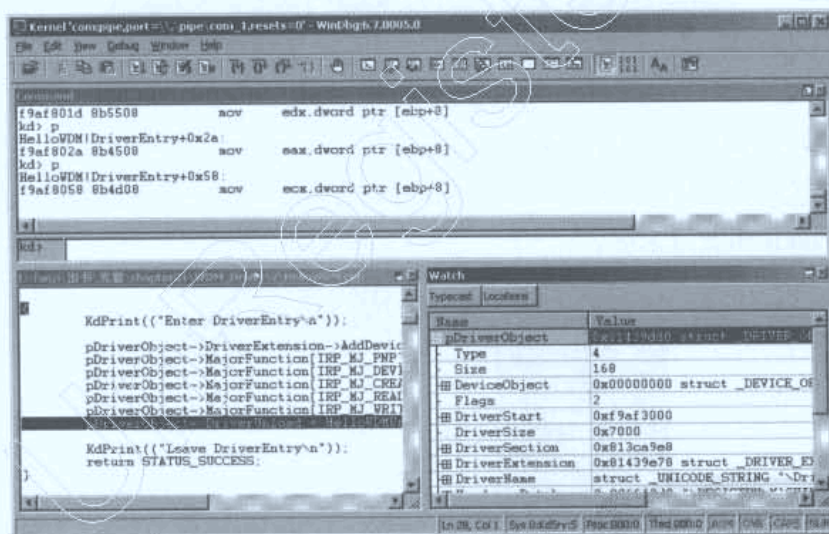


图 23-9 内核源码级调试

用 WMWare 和 WinDbg 联合调试最大的好处就是可以在单机的情况下实现驱动源码级调试, 这非常方便内核程序员对驱动程序进行跟踪、调试, 同时还可以学习更多的操作系统内核知识。

23.3 用 IRPTrace 调试驱动程序

用 WinDbg 和 VMWare 联合进行调试或者用两台 PC 联机调试, 都还略显复杂, 这里

介绍一种比较简单的方法,即用 IRPTrace 软件进行调试。对于大多数驱动程序来说,往往大多数问题出自 IRP 方面。而 IRPTrace 恰恰就适用于对 IRP 进行跟踪和调试。

IRPTrace 的原理是借助一个过滤驱动程序,并将其附加在需要观察的驱动程序之上。最后由这个过滤驱动程序对每个 IRP 的派遣函数都设置一个完成例程。在完成的例程中可以得到 IRP 的处理结果。这样对于需要观察的驱动程序所发出的所有 IRP,都会在过滤驱动中进行记录,以便程序员进行观察。其实大多数调试工具软件,例如,调试 USB 驱动的 BusHound 软件,都是利用这个原理编写的。

这里假设笔者编写一段错误代码。本段错误的代码将导致系统无法正常加载 HelloWDM 驱动,即加载驱动错误。如图 23-10 所示。

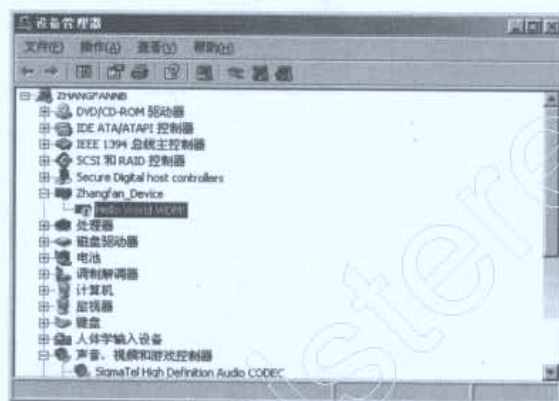


图 23-10 驱动加载出错

本段测试用的部分代码如下:

```
#001 #pragma PAGEDCODE
#002 NTSTATUS StartPnpDevice(PDEVICE_EXTENSION pdx, PIRP Irp)
#003 {
#004     //确保函数运行在分页内存
#005     PAGED_CODE();
#006     KdPrint(("Enter StartPnpDevice\n"));
#007     NTSTATUS status = STATUS_UNSUCCESSFUL;
#008     //设置 IRP 完成状态
#009     Irp->IoStatus.Status = status;
#010     //设置 IRP 操作字节数
#011     Irp->IoStatus.Information = 0;
#012     //结束 IRP 请求
#013     IoCompleteRequest(Irp, IO_NO_INCREMENT); //这句导致 IRP 返回失败
#014     KdPrint(("Leave StartPnpDevice\n"));
#015     return status;
#016 }
```

此段代码可以在配套光盘中本章的 ErrorTest2 目录下找到。

可以看出,由于 IRP_MN_START_DEVICE 这个 PNP 的派遣函数中将 IRP 设置成了 STATUS_UNSUCCESSFUL 状态,从而导致出现错误。

对于这个错误,可以用 IRPTrace 工具查看。

首先在加载驱动程序前，对 HelloWDM 驱动进行过滤。当加载这个驱动时，IRPTrace 会自动跟踪 HelloWDM 驱动向底层发送的所有 IRP。如图 23-11 所示。

IRP_MN_START_DEVICE 的 IRP 的相应图标出现红色的叉子，即说明这个 IRP 处理没有成功。需要进而查看 IRP 的状态。如果发现被设置为 STATUS_UNSUCCESSFUL 状态，即可知道是 IRP 完成时没有成功，因而可以将错误定位在 IRP_MN_START_DEVICE 的派遣函数上。这样就缩小了检查范围。

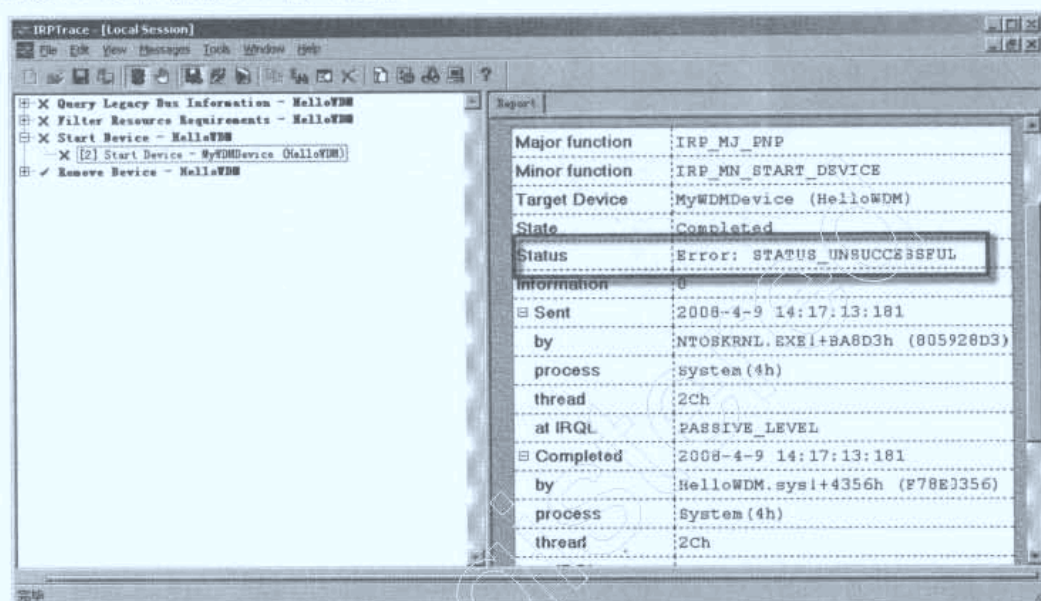


图 23-11 用 IRPTrace 跟踪错误

23.4 小结

本章介绍了一些常用驱动程序调试的高级技巧，包括用 WinDbg 查看蓝屏存储文件。蓝屏存储文件是在系统崩溃前对系统的一个快照。这样可以查看操作系统崩溃前驱动程序在内核中的调用堆栈。

如果调用堆栈中出现了自己编写的驱动程序中的相关函数，则说明其中的某个地方出现错误而导致了系统崩溃。通过正确加载符号表，可以具体定位发现到底是哪个函数中的哪行命令导致出现错误。

当通过两台 PC 或者使用虚拟机模拟另外一台 PC 时，都可以采用 WinDbg 的内核调试功能。通过这个功能，可以进行源码级的驱动程序调试。源码级调试可以跟踪驱动程序的整体运行，方便程序员查看运行的内部情况。

另外还可以通过其他工具软件查看驱动程序的运行情况。如用 IRPTrace、Dbgview、BusHound、Device Tree 等工具对其进行调试。