# 《Windows驱动开发技术详解》

## 图书基本信息

书名：《Windows驱动开发技术详解》

13位ISBN编号：9787121068461

10位ISBN编号：712106846X

出版时间：2008-1

出版社：电子工业出版社

作者：张帆等

页数：530

# 《Windows驱动开发技术详解》

## 内容概要

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。

# 《Windows驱动开发技术详解》

## 作者简介

张帆，毕业于北京理工大学电子工程系，曾就职于威盛电子有限公司，现就职于北京创毅视讯科技有限公司。长期从事PCI、USB、SDIO、串口、摄像头等设备的Windows驱动程序开发。对Windows操作系统内核有深入的研究，并且有丰富的Windows驱动程序开发经验。　　史彩成：博士后，北京理工大学信息科学技术学院副教授，资深电子系统专家，主要从事图像处理、激光信号处理、数据融合及ASIC设计等领域的研究工作。

# 《Windows驱动开发技术详解》

## 书籍目录

# 《Windows驱动开发技术详解》

## 章节摘录

第1篇　入门篇　　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BuildEnvironment，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。

# 《Windows驱动开发技术详解》

## 媒体关注与评论

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　　毛二可

# 《Windows驱动开发技术详解》

## 编辑推荐

原创经典，威盛一线工程师倾力打造。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。　　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，深入Windows操作系统的底层和内核，透析Windows驱动开发的本质。　　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　　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。

# 《Windows驱动开发技术详解》

## 精彩短评

1、比较详细
2、在被两本undocumented xx虐过以后在看这本，简直直下三千尺的流畅。
3、不会想读，除非回到底层领域
4、为了工作。 学习。 写windows驱动必看的书。
5、读了书中前8章，写的不错，收藏了

6、感觉写的顺序都点乱，初看的时候一头雾水。。。
7、想死啊。。。
8、挺好的,适合初学者
9、很多实用的例子，真的很有帮助。
10、很好的入门
11、即使开发了很久了 也可以用来查容易忘的细节
12、好书！知识讲解的很系统，很详细。
13、依旧是windows，依旧是张帆。
14、windows底层驱动入门指南，非常不错。
15、作者对概念理解不够深入，与PROGRAMMING WINDOWS DRIVER MODEL相比还是有差距。
16、深入浅出，学习windows驱动开发的好书
17、单位的一本书...抽空看看...{适合入门}...
18、非常好的书，中国人自己写的，把很多能遇到的，容易忽略的问题都描述了，很棒。
19、介绍这方面的书不多，想研究这方面的可以看，内容不错。但小女子我再也不想看这种高深技术书了。
20、还挺不错的书
21、驱动开发入门经典!通俗易懂!

## 精彩书评

1、art baker的nt驱动我是看过的，walter oney 的那本wdm看过3遍以上了，但是真正好的还是这本。就像作者说的那样，“ 写这本书，是为了一个梦 ”。我要感谢作者，感谢这本书。
2、P231第四行：主要获得了互斥体——只要获得了互斥体P254第12行 这里没有设置---这里设置P316的代码段里：构造设备名字符串，而下面的代码：RtlInitUnicodeString( &DeviceName, L"\\??\\HelloDDKA" );这是构造设备符号链接字符串，最后一行：此段代码在Test7目录下找到，实际在Test7目录下找不到，在Test8目录下的DriverDevB 可以找到，同理P318也也是这样，应改为在Test8目录下可以找到。
3、尊敬的博文视点的工作人员：您好!我是一名华中科技大学的研究生。最近在网上看到了博文视点推出的新书《windows设备驱动开发技术详解》。看过网上的介绍后，觉得该书脉络清晰，讲解循序渐进，细致入微，是一本国内不可多得的驱动开发的经典。恰好，我正在为学习驱动编程开发犯愁，就从卓越网上订购了一本。看过该书的总体内容后，我发现了该书与其它的驱动开发的书相比有以下优点：1.国内以前出版的许多将驱动开发的书入门要求比较高，没有考虑到现在越来越多非计算机专业的科研人员需要编写驱动程序，像搞搞工业控制的，搞机械装备的，数控开发的.....很多具体的科研项目都需要开发具体de 对应于这一项目的驱动程序，很多情况下，板卡生产商提供的通用的驱动程序并不能满足要求。而非计算机专业的科研人员又往往存在开发驱动基础知识和经验不足的瓶颈。常常导致科研课题一拖再拖，耽误了许多宝贵时间。例如我是材料加工工程专业的，研究方向是加工装备自动化。我一个课题需要开发几个板卡的数据采集驱动。而开发windows驱动程序需要具有一定的操作系统、计算机组成原理以及数据结构的基础知识，而这些我学习的不多，从而导致学起来付出很多，收效甚微。该书与其他书相比，降低了入门的门槛，特别适合非计算机专业的朋友学习驱动，而且由浅入深，一步步的将读者带入驱动开发的更深领域。而目前许多其他的驱动开发图书上来就给人以深不可测的感觉，就给人一“ 下马威 ”。2.我认为该书讲解细致入微，对将会困扰许多驱动开发人员的概念和术语讲解相当清楚，这也是许多其他书籍没有做到的。3.该书还有一个值得称道的地方是对于代码的讲解部分。这部分很好的体现了作者的良苦用心，对绝大部分代码，作者都进行了注释，体现了对读者负责的态度，而绝大部分其它驱动编程书中是大段代码罗列，却没有几行注解。总而言之，我觉得该书是我遇到的驱动开发书中为数不多精品，必将把更多学习windows驱动开发的人员带入驱动开发的大门。不过，在欣喜之余，我觉得也有些遗憾：就是我觉得该书应该再出版一个补充教材，或者再版时加以补充。我所指的是该书应该加上：一、对于用driver studio 软件开发驱动程序的讲解。因为越来越多的朋友用这个工具软件开发驱动。我觉的对于driver studio 及它附带的经典的调试工具softICE应该详细讲解，而该书涉及的比较少。二、我认为还应加上ISA 设备驱动一章，事实上，在工控领域，ISA总线系列的工控设备仍然占据相当的份额，许多的ISA板卡仍在使用，并没有过时。我认为作者应当详细的介绍ISA 设备的驱动开发。总之，该书是一本不可多得的好书，但毕竟是第一版。我认为的不完善之处，也许作者有别的考虑。但我想既然该书取名详解，而且讲解真的是很好，那就应当发挥出经典的作用，使内容更加全面。同时，我也期待着该书的修订和升级以及与作者的沟通。最后还是要感谢博文视点的工作人员，推出了这样一本好书。希望有更多这样优秀的新书出版。
4、再一次让我对大陆的IT书籍失望，看来这又是一本滥竽充数、混著作数评级的书。 本来看了目录觉得还不错，再加上说是“ via工程师的心得之作 ”，但书到手以后。。。 这回购书得到的经验：以后再买书，看目录的时候，不仅要看内容安排，还要看看他的页数。 现在看起来觉得自己很傻很天真，那么广的内容，怎可能寥寥数页就写完。 书写的不清不楚，没啥新意也就得了， 关键整书充斥着抄袭，比如大篇幅抄袭《USB 2.0原理与工程开发》，看过这两本书的读者都会有个比较，一字不差的抄，精简的抄，抄文字，抄图例，不一而足，真是让人无语。以上是我在当当的评论，书本身就很失望了，再看到其他人的好评如潮，更是失望，真正的好书没人读，这种烂书竟能获得那么多的好评，真是无语。忠告想买这本书的朋友，先去书店看看再说。
5、----------张帆的驱动开发论坛 http://bbs.kerneldev.com/张帆的驱动开发QQ群1# 49944346(满)张帆的驱动开发QQ群2# 64778681(目前开放中...)

# 《Windows驱动开发技术详解》

## 章节试读

1、《Windows驱动开发技术详解》的笔记-第48页

驱动程序被加载在内核模式下，它与windows操作系统其他组件进行密切互交。

windows特性：
1.可移植性------在尽可能多的平台上运行，于硬件密切相关的只有HAL层。windows被设计成为软件分层体系结构，引入面向对象编程思想。把操作系统各个组件抽象出来

2.兼容性------API保持一贯的命名和调用接口，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

4.可扩展性------内核从执行体中分离出来，操作系统内核只负责关于线程的调度工作。其他操作系统组件，如内存管理组件，进程管理组件等作为独立于内核的组件，统称为程序执行组件。

总体架构：
win32子系统将API函数转换为Native API函数。在Native API中，已经没有了子系统概念，它将这种调用转换为系统服务函数的调用。系统服务函数通过I/O管理器将消息传递给驱动程序。

Win32子系统
win32子系统是最纯正的windows子系统，提供大量的API。除此之外还有OS/2，POSIX，WOW，VDW子系统。

Native API
Native API的函数一般是win32 API前面加NT两个字母。所有Native API都是在Ntdll.dll中实现的。三个win32 子系统的核心dll文件都依赖于Ntdll.dll的。Native API是从用户模式进入内核模式的大门。

系统服务
win XP下是通过sysnter指令完成调用服务的。

执行程序组件
是内核模式下的一组服务函数，它们位于ntoskrnl.exe中。
1.对象管理程序-----所有服务几乎都是以对象的形式存在
2.进程管理程序------负责创建和终止进程
3.虚拟内存管理程序
4。I.O管理器--------负责发起I/O请求，并管理这些请求。无论对端口，键盘访问，磁盘文件的操作统一为IRP的请求形式。其中IRP包含了对设备操作的重要数据。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。
5.配置管理程序--------记录所有计算机软件，硬件配置信息。它使用一个被称为注册表的数据库保存这些数据

硬件抽象层
对各种硬件进行了抽象。在驱动程序中，应尽量避免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。

2、《Windows驱动开发技术详解》的笔记-第185页

DDK（过时了，现在是WDK）中，不鼓励使用ANSI字符串，因为标准的C字符串处理函数容易导致缓冲区溢出等错误。
字符串初始化与销毁
RtlUnicodeString------初始化Unicode字符串

字符串复制
RtlCopyUnicodeString-------Unicode字符串复制

字符串比较
RtlCompareUnicodeString-------比较Unicode字符串

字符串转换成大写
RtlUpcaseUnicodeString------将Unicode字符串转换成大写

字符串与整型数字相互转换
RtlUnicodeStringToInteger------将Unicode字符串转换为整数

内核模式下的文件操作
ZwCreateFile-------文件的创建和打开
ZwOpenFile--------打开文件
InitializeObjectAttributes---------对OBJECT_ATTRIBUTES初始化
ZwSetInformationFile---------设置（修改）文件属性
ZwQueryInformationFile--------获取文件属性
ZwWriteFile-------写文件
ZwReadFile------读文件

内核模式下的注册表操作
ZwCreateKey---------创建和打开注册表句柄（类似CreateFile）
ZwOpenKey-------打开注册表
ZwSetValueKey-------添加和修改注册表键值
ZwQueryValueKey-------对注册表的项进行查询
ZwQueryKey--------获得某注册表项究竟有多少个子项
ZwEnumerateKey---------针对第几个子项获取该子项具体信息
ZwEnumerateValueKey------针对第几个子键获取该子健具体信息
ZwDeleteKey-------删除子项

3、《Windows驱动开发技术详解》的笔记-第291页

I/O定时器
DDK提供的一种定时器。每间隔一秒钟就会调用一次I/O定时器例程
IoInitializeTimer-------初始化IO定时器
IoStartTimer------开启定时器
IoStopTimer------关闭定时器

IO定时器例程声明：

```
VOID OnTimer(IN PDEVICE_OBJECT pDevObj, IN PVOID Context)
{

}
```

I/O定时器运行在DISPACTH_LEVEL级别，因此在这个例程中不能使用分页内存，否则会引起系统崩溃蓝屏。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，因此不能直接使用应用程序的内存地址。

实例代码：

```
//在DriverEntry中先初始化I/O定时器
//然后在设备扩展中加如一个计数变量，负责记录事件间隔的秒数

NTSTATUS HelloDDKDeviceControl(PDEVICE_OBJECT pDevObj, PIRP pIrp)
{
NTSTATUS status = STATUS_SUCESS;
KdPrint(("Enter HelloDDKDeviceControl\n"));
PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
//获得输入参数大小
ULONG cbin = stack -&gt; Parameters.DeviceIoControl.InputBufferLength;
//获得输出参数大小
ULONG cbout = stack -&gt; Parameters.DeviceIoControl.OutputBufferLength;
//得到IOCTL码
ULONG code = stack -&gt; Parameters.DeviceIoControl.IoControlCode;
PDEVICE_EXTENSION pDecExt = (PDEVICE_EXTENSION)pDevObj -&gt; DeviceExtension;
ULONG info = 0;
switch(code)
{
case IOCTL_START_TIMER:
KdPrint(("IOTRL_START_TIMER\n"));
pDevExt -&gt; lTimerCount = TIME_OUT;
IoStartTimer(pDevObj);
break;
case: IOCTL_STOP:
KdPrint(("IOTRL_STOP_TIMER\n"));

IoStopTimer(pDevObj);
default:
status = STATUS_INVALID_VARINT;
break;
}
pIrp -&gt; IoStatus.Status = status;
pIrp -&gt; IoStatus.Information = info;
IoCompleteRequest(pIrp, IO_NO_INCREMENT);
KdPrint(("Leaver HelloDDKDeviceControl\n"));
return status;
}

VOID OnTimer(IN PDEVICE_OBJECT pDevObj, IN PVOID Context)
```

```
{
  PDEVICE_EXTENSION pDecExt = (PDEVICE_EXTENSION)pDevObj -&gt; DeviceExtension;

kdPrint(("Enter Timer"));
InterlockedDecrement(&amp;pDevExt -&gt; lTimerCount);
LONG previousCount = InterlockedCompareExchange(&amp;pDevExt -&gt; lTimerCount, TIME_OUT,0);

if(previousCount == 0) KdPrint(("%dsecs time out!\n" ,TIME_OUT));

PEPROCESS pEProcess = IoGetCurrentProcess();

PTSTR  ProcessName = (PTSTR )((ULONG)pEProcess + 0x174);
KdPrint(("The current Process name is %s\n", ProcessName));
}
```

DPC定时器这种定时器更加灵活，允许任意间隔事件进行定时。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。
KeInitializeTimer------初始化定时器对象
KeInitializeDpc------初始化DPC对象
KeSetTimer-------开启定时器
KeCancelTimer------取消定时器
在调用KeSetTimer后，只会触发一次DPC例程。如果想周期性触发DPC例程，需要在DPC例程触发后，再次调用KeSetTimer。
等待
KeWaitForSingleObject也可以用来等待时间。强制当前线程进入睡眠。经过指定时间恢复。KeDelayExecutionThread等待时间。强制当前线程进入睡眠。进过指定时间恢复。KeStallExecutionProcessor。让CPU不停的等待，而不是让线程进入休眠，类似自旋锁。浪费CPU时间。不宜超过50us。由于没有将线程进入睡眠，也不会发生线程切换，因此这种方法的延时比较精确。
时间相关的内核函数

KeQuerySystemTime------获取当前系统时间
ExSystemTimeToLocalTime---------将系统时间转换成当前时区对应的时间。
ExLocalTimeToSystemTime--------将当前时区对应的时间转换系统时间
RtlTimeFieldsToTime------由当前的年月日得到系统时间
RtlTimeToTimeFields------由系统时间得到当前的年月日

IRP的超时处理
在驱动程序编程中，有某种情况，对某设备的操作很久没有反应。如果在规定时间内没有完成操作，则要取消该操作。

4、《Windows驱动开发技术详解》的笔记-第216页

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，全部由操作系统转换为一个IRP的数据结构，不同的IRP数据被“派遣”到不同的派遣函数中。（其实也可以叫分发函数（Dispatch））.
IRPIRP是输入输出请求包，与输入输出相关的重要数据结构IRP的两个基本属性，MajorFunction

和MinorFunction，分别记录IRP的主类型和子类型。操作系统根据MajorFunction分发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数据哪种MinorFunction。一般来说，NT和WDM都是在DriverEntry中注册派遣函数的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。

IRP类型IRP类似win32应用程序中的“ 消息 ”概念。文件I/O相关函数如CreateFile CloseHandle等（包括内核ZwCreateFile）函数会使操作系统产生出IRP_MJ_CREATE,IRP_MJ_READ等不同的IRP，这些IRP会被传递到驱动程序的派遣函数中。
IoCompleteRequest--------结束IRP请求

通过设备链接打开设备
例如：CreateFile("\\\\.\\HelloDDK",...)；

编写一个更通用的派遣函数IO_STACK_LOCATION,即I/O堆栈，这个数据结构和IRP紧密相连。IRP会被操作系统发送到设备栈的栈顶，如果顶层设备对象的派遣函数结束了IRP请求，则这次I/O请求结束。如果没有结束，那么操作系统将IRP转发到设备栈的下一层设备处理。如果还没有结束，就由此一层层向下转发。

缓冲区方式读写操作驱动程序所创建的设备一般有三中读写方式，1.缓冲区方式，2.直接方式，3.其他方式
缓冲区设备读写操作一般是WriteFile和ReadFile引起的。WriteFile要求用户提供一段缓冲区，并且说明缓冲区大小，然后WriteFile将这段内存的数据传入驱动程序中。这段缓冲区是用户模式地址，驱动程序直接引用这段内存是很危险的，因为操作系统是多任务的，进程随时在切换。

缓冲区设备读写以缓冲区方式读写设备，都会发生用户模式地址与内核模式地址的数据复制，复制过程由操作系统负责。WriteFile和ReadFile指定对设备操作了多少字节，并不是真正意味着这么多字节，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。
直接方式读写
在创建完设备对象后，在设置设备属性的时候设置为DO_DIRECT_IO。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，这样用户模式的地址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。操作系统将用户模式的地址锁住后，操作系统用内存描述符表（MDL数据结构）记录这段内存。
其他方式读写
其他方式设备
既不设置DO_BUFFERED_IO，也不设置DO_DIRECT_IO，此时采用就是其他读写方式。只有驱动程序与应用程序运行在相同的线程上下文的情况下，才能使用这种方式。
IO设备控制操作除了用ReadFile和WriteFile读写设备以外，还可以用DeviceIoControl这个win32 API操作设备。它内部会创建一个IRP_MJ_DEVICE_CONTROL类型的IRP，然后系统将这个IRP转发到派遣函数中。它还可以让应用程序和驱动程序进行通信。（自己定义I/O控制码，用这个函数将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。）
缓冲内存模式IOCTL在win32 API DeviceIoControl中，用户提供的输入缓冲区的内容被复制到IRP中的pIrp -> AssociateIrp.SystemBuffer内存地址，复制的字节数由DeviceIoControl指定的输入字节数。派遣函数还可以写入pIrp -> AssociateIrp.SystemBuffer内存地址，被当作设备输出数据。
直接内存模式IOCTL
跟上文差不多（内核模式地址映射）

其他内存模式IOCTL
跟上文差不多（必须同一线程上下文）

5、《Windows驱动开发技术详解》的笔记-第249页

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。
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。
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，也就是硬件中断。另一种由软件指令int n产生的中断。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。可屏蔽中断是通过可编程中断控制器（PIC）8259A芯片向CPU发送请求的。
高级可编程控制器（APIC）
现在x86计算机基本都是用高级可编程控制器，即APIC。APIC兼容PIC，并且APIC把IRQ的数量增加到了24个（IRQ0~IRQ23）
中断请求级（IRQL）
24个IRQ。每个IRQ都有各自的优先级别。当优先级高的中断来临时，处于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程序中。windows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）。其中数字从0到31，优先级逐次递增。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上。当设备中断来临时，操作系统提升IRQL到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。
DISPATCH_LEVEL > APC_LEVEL > PASSIVE_LEVEL
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Driver_Entry函数，派遣函数，AddDevice等函数一般都运行在 PASSIVE_LEVEL级别，它们可以在必要时申请DISPATCH_LEVEL级别。windows负责线程调度的组件是运行在DISPATCH_LEVEL级别。
线程调度与线程优先级
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PASSVIE_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。
IRQL的变化
线程运行在PASSIVE_LEVEL级别，这时候操作系统随时可能将当前线程切换到另外的线程。但是如果提升IRQL到DISPATCH_LEVEL 级别，这个时候就不会出现线程的切换。这是一种很常用的同步机制，但这种方法只能使用于单CPU系统。对于多CPU系统，需要采取别的同步机制。
IRQL与分页
缺页中断允许出现在PASSIVE_LEVEL级别的程序中，但是如果在DISPATCH_LEVEL或者更高级的IRQL的程序中会带来系统崩溃。对于等于或者高于DISPATCH_LEVEL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或者更高的级别IRQL。因此使用这些例程不能使用分页内存，否则导致系统崩溃。
IRQL提升与降低

KeRaiseIrql--------提升当前IRQL
KeLowerIrql--------降低IRQL

自旋锁
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 ”之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 ”。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。而自旋锁不同，它不会切换到别的线程，而是让这个线程“ 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。
用户模式下的同步对象
同步对象包括事件，互斥体，信号灯等。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。
使用方法详情请查看《win32汇编程序设计》的读书笔记

内核模式下的同步对象
用户模式下，程序员无法获得真实同步对象的指针，而是句柄代表这个对象。在内核模式下，程序员可以获得真实同步对象的指针。
内核模式下的等待

KeWaitForMultipleObjects
KeWaitForSingleObject

内核模式下开启多线程
PsCreateSystemThread创建新线程。可以创建两种：1.用户线程。2.系统线程。 系统线程不属于当前用户进程，而属于系统进程。
PsTerminateSystemThread---------结束线程
IoGetCurrentProcess--------得到当前线程

内核模式下的事件对象

KeInitializeEvent---初始化事件对象
KeSetEvent-----设置事件

驱动程序与应用程序互交事件对象

```
int main()
{
   HANDLE hDevice = CreateFile("\\\\.\\HelloDDK",
                                    GENERIC_READ | GENERIC_WRITE,
                                     0, OPEN_ATTRIBUTE_NORMAL,
                                       NULL);

  if(hDevice == INVALID_HANDLE_VALUE)
  {
      printf("Fialed .Error Code is %s\n", GetLastError());
      return 1;
  }

  BOOL bRet;
```

```
  DWORD dwOutput;

  HANDLE hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

  HANDLE hThread1 = _beginthreadex(NULL, 0, Thread1, &amp;hEvent, 0, NULL);

  bRet = DeviceIoControl(hDevice, IOCTL_TRASMIT_EVENT, &amp;hEvent, sizeof(hEvent), NULL, 0,
&amp;dwOutput, NULL);

  WaitForSingleObject(hThread1, INFINITE);

 CloseHandle(hDevice);
  CloseHandle(hThread1);
   CloseHandle(hEvent1);

  return 0;
}


NTSTATUS HelloDDKDeviceControl(PDEVICE_OBJECT pDevObj, PIRP pIrp)
{
    NTSTATUS status = STATUS_SUCESS;

    KdPrint(("Enter HelloDDKDeviceControl\n"));

   PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);
   //获得输入参数大小
   ULONG cbin = stack -&gt; Parameters.DeviceIoControl.InputBufferLength;
  //获得输出参数大小
   ULONG cbout = stack -&gt; Parameters.DeviceIoControl.OutputBufferLength;

   //得到IOCTL码
   ULONG code = stack -&gt; Parameters.DeviceIoControl.IoControlCode;
   ULONG info = 0;
   switch(code)
  {
      case  IOCTL_TRASMIT_EVENT:
            KdPrint(("IOTRL_TEST\n"));
            HANDLE hUserEvent = *(HANDLE*)pIrp -&gt;       AssociatedIrp.SystemBuffer;
            PKEVENT pEvent;

            status = ObReferenceObjectByHandle(hUserEvent, EVENT_MODIFY_STATE, *ExEventObjectType,
KernelMode, (PVOID*)&amp;pEvent, NULL);
            KeSetEvent(pEvent, IO_NO_INCREMENT, FALSE);
            ObDereferenceObject(pEvent);
           break;
       default:
           break;
```

```
  }

  pIrp -&gt; IoStatus.Status = status;
  pIrp -&gt; IoStatus.Information = info;

  IoCompleteRequest(pIrp, IO_NO_INCREMENT);
  KdPrint(("Leaver HelloDDKDeviceControl\n"));

return status;

}
```

驱动程序与驱动程序互交事件对象

IoCreateNotificationEvent-----创建同步事件对象
IoCreateSynchronizationEvent--------创建通知事件对象
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，最好不要用很简单的名字，否则容易导致命名冲突。
内核模式下的信号灯

KeInitializeSemaphore-------对信号灯对象进行初始化
KeReadStateSemaphore-------读取信号灯当前计数
KeReleaseSemaphore----释放信号灯（会增加信号灯计数）

内核模式下的互斥体

KeInitializeMutex------初始化互斥体
KeReleaseMutex-------释放互斥体

快速互斥体
作用和互斥体一样。。但是快速互斥体执行速度比普通互斥体快（这里指获取和释放的速度）。缺点：不能递归地获取互斥体对象，就是互斥体只互斥其他线程，而不互斥自己所在的线程。
ExInitializeFastMutex-----初始化快速互斥体
ExAcquireFastMutex------获取快速互斥体
ExReleaseFastMutex-------释放快速互斥体

使用自旋锁进行同步

KeAcquireSpinLock-----获得自旋锁
KeReleaseSpinLock-------释放自旋锁

使用互锁操作进行同步

InterLockedXXXX-------这系列函数查WDK文档

6、《Windows驱动开发技术详解》的笔记-第147页

物理内存概念PC上有三条总线，数据总线，地址总线和控制总线。32bit的CPU寻址能力

为(2^32bytes=4GB)字节。用户最多可以使用4GB的真实的物理内存。
虚拟内存概念windows所有程序（ring0，ring3）可以操作的都是虚拟内存。之所以称为虚拟内存，是因为它的所有操作最终会变成对物理内存的操作。在CPU中有一个重要的寄存器CR0，32位的。其中一位PG位表示系统是否分页。windows启动前将它置为1，允许分页。页大小一般为4KB。4GB的虚拟内存会被分割为1M(4GB/4KB=2^20)个分页单元。
原因如下：
1.虚拟内存增加了内存的大小。不管PC是否有足够的4GB物理内存，操作系统总会有4GB虚拟内存。当物理内存不够用的时候，就把虚拟页换成文件，等需要时候再去读取。
2.不同进程的虚拟内存互不干扰。

用户模式地址和内核模式地址虚拟地址0x0~0x7FFFFFFF范围内的地址（即低2G的虚拟地址），被称为用户模式地址。而0x80000000~0xFFFFFFFF范围内的地址（高2G的虚拟地址），被称为内核模式地址。windows的核心代码和windows驱动程序加载的位置都在高2GB的内核地址里，一般应用程序不能访问到这些核心代码和数据windows操作系统在进行进程切换时，保持内核模式地址完全相同。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切换。
分页与非分页内存可以交换到磁盘文件上的虚拟页称为分页内存，不能交换到磁盘文件上的虚拟页称为非分页内存。当程序的中断请求级别在DISPATCH_LEVEL之上（包括DISPATCH_LEVEL），程序只能使用非分页内存，否则将导致蓝屏死机。（所以使用内核API时注意中断请求说明。）
分配内核内存
内核内存资源非常珍贵。内核栈空间比用户栈空间小得多。用户栈有1M（可以设置为2M），但是内核栈空间才64KB。大结构请在堆中申请。所以驱动程序要尽量避免递归。

运行时函数
RtlCopyMemory--------内存复制（没考虑重叠部分）
RtlMoveMemory-------内存复制（可重叠）
RtlFillMemory--------对某段内存区域用固定字节进行填充
RtlZeroMemory-------对某段内存区域填零
RtlEqualMemory------比较两个内存块是否一致

返回状态值
NTSTATUS有状态表。可以查。

检查内存可用性帮助程序员在不知道内存是否可写可读。ProbeForRead--------是否可读
ProbeForWrite--------是否可写如果不可读写，会引发一个异常。用结构化异常处理来检测。
结构化异常处理
__try{}块和__except(filter_value){}块处理。引发异常的语句后面的语句将不会执行。而跳到except块执行。

__try{}块和__finally{}块。强迫程序在退出前执行一段代码，也就是说，try块里面发生return或者是异常，在程序退出前都会运行finally块中的代码。

断言
ASSERT-------不满足某条件引发异常。

7、《Windows驱动开发技术详解》的笔记-第30页

windows驱动分两类：1，不支持即插即用功能的NT式驱动（反外挂，寒江独钓的书籍一般都是NT式驱动） 2，支持即插即用的WDM驱动（鼠标，键盘，PCIE等设备，真实存在的物理设备都需要）。NT式驱动的头文件一般是NTDDK.H。WDM驱动的头文件一般是WDM.HNT式驱动类似于windows服务程序，以服务的方式加载在系统中。成功加载的驱动，会在windows的设备管理器中。默认情况下，NT式驱动程序是隐藏的。AddDevice例程是WDM驱动特有的，NT式驱动是没有这个例程的。此回调函数的作用是创建设备对象，并由PNP管理器调用。AddDevice例程的参数，DriverObject是PNP管理器传递进来的驱动对象，其实就是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，这个概念是NT式驱动中没有的。
驱动的安装
WDM式驱动和NT式驱动的安装有很大不同，安装WDM驱动需要有相应的inf文件。inf文件描述了硬件信息和驱动的一些信息。

8、《Windows驱动开发技术详解》的笔记-第276页

对设备的任何操作最终都会转化成为IRP请求，而IRP一般都是由操作系统异步发送的。异步处理IRP有助于提高效率，但是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。

应用程序对设备的同步异步操作

操作设备的函数主要是：ReadFile，WriteFile，DeviceIoControl。
同步时，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IRP传送到驱动程序的派遣函数中。处理该IRP需要一段时间，知道IRP处理完毕后，DeviceIoControl才会返回。异步时，DeviceIoControl内部会产生IRP，并将IRP传递给驱动的派遣函数。但此时DeviceIoControl不会等待IRP是否结束，而是直接返回。当IRP经过一段事件被结束时，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。

同步操作设备
如果要同步操作设备，那么CreateFile的时候就要指定以“同步”方式打开。dwFlagsAndAttributes参数是关键。如果没有设置FILE_FLAG_OVERLAPPED则是同步，如果设置了，就为异步。
异步操作设备（几种方法）

```
int main()
{
   HANDLE hDevice = CreateFile(...) //以异步方式打开，参数就不写了

   if(hDevice == INVALID_HANDLE_VALUE)  {printf("Error!"); return 1;}

    UCHAR buffer[BUFFER_SIZE];
   DWORD dwRead;

   OVERLAPPED overlap = {0};
   overlap.hEvent = CreateEvent(NULL, FALSE,FALSE,NULL);

    ReadFile(hDevice, buffer, BUFFER_SIZE, &amp;dwRead, &amp;overlap);
```

```
    WaitForSingleObject(overlap, hEvent,INFINITE);

   CloseHandle(hDevice);
   return 0;
}
```

ReadFileEx-------专门异步读
WriteFileEx--------专门异步写
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。也就是说，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）。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。SleepEx，WaitForXXXXX系列函数，都有一个参数bAlerttable，当设置为TRUE，就进入警惕模式。当系统进入警惕状态时，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。

```
VOID CALLBACK MyCompleteRoutine(DWORD dwErrorCode,
DWORD dwNumberOfBytesTransfered,
LPOVERLAPPED lpoverlapped)
{
   printf("IO Complete end!\n")
}
int main()
{
   HANDLE hDevice = CreateFile(...) //以异步方式打开，参数就不写了

   if(hDevice == INVALID_HANDLE_VALUE)  {printf("Error!"); return 1;}

    UCHAR buffer[BUFFER_SIZE];


   OVERLAPPED overlap = {0};


    ReadFileEx(hDevice, buffer, BUFFER_SIZE, &amp;overlap，MyCompleteRoutine);

 SleepEx（0，TRUE）；
   CloseHandle(hDevice);
   return 0;
}
```

IRP的同步完成和异步完成
如果派遣函数不调用IoCompleteRequest，则需要告诉操作系统次IRP处于“挂起”状态。这需要调用IoMarkIrpPending。同时，派遣函数应该返回STATUS_PENDING。
取消IRP

IoSetCancelRoutine-------取消例程与IRP的关联，一旦取消IRP，这个取消例程会被执行
IoCancelIrp------取消IRP

StartIO例程
保证各个并行的IRP顺序执行，即串行化。
中断服务例程（ISR）
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。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。ISR运行在DIRQL级别，要高于普通线程的优先级。自旋锁只能对DISPACTH_LEVEL以下的程序进行同步。派遣函数和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提升到相应的DIRQL级别
KeSynchronizeExecution--------与ISR同步，它提供的函数不会被ISR打断

DPC例程
ISR处于很高的IRQL，会打断正常运行的线程。而DPC例程运行在PASSVIE_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ISR中。一般来说，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。
KeInitializeDpc--------初始化
IoRequestDpc------请求Dpc

9、《Windows驱动开发技术详解》的笔记-第4页

关于typedef struct XX {.....}XXX,XXX,XXX;的相关补充

struct和typedef struct
分三块来讲述：
1 首先：
在C中定义一个结构体类型要用typedef:

```
typedef struct Student
{
  int a;
}Stu;
```

于是在声明变量的时候就可：Stu stu1;
如果没有typedef就必须用struct Student stu1;来声明
这里的Stu实际上就是struct Student的别名。
另外这里也可以不写Student（于是也不能struct Student stu1;了）

```
typedef struct
{
  int a;
}Stu;
```

但在c++里很简单，直接

```
struct Student
{
  int a;
};
```

于是就定义了结构体类型Student，声明变量时直接Student stu2；
==========
2其次：
在c++中如果用typedef的话，又会造成区别：

```
struct  Student
{
  int  a;
}stu1;//stu1是一个变量
typedef  struct  Student2
{
  int  a;
```

}stu2;//stu2是一个结构体类型
使用时可以直接访问stu1.a
但是stu2则必须先 stu2 s2;
然后 s2.a=10;
==========
3 掌握上面两条就可以了，不过最后我们探讨个没多大关系的问题
如果在c程序中我们写：

```
typedef struct
{
  int num;
  int age;
}aaa,bbb,ccc;
```

这算什么呢？
我个人观察编译器（VC6）的理解，这相当于

```
typedef struct
{
  int num;
  int age;
}aaa；
typedef aaa bbb;
typedef aaa ccc;
```

也就是说aaa,bbb,ccc三者都是结构体类型。声明变量时用任何一个都可以,在c++中也是如此。但是你要注意的是这个在c++中如果写掉了typedef关键字，那么aaa，bbb，ccc将是截然不同的三个对象。

10、《Windows驱动开发技术详解》的笔记-第5页

#pragma INITCODE //将driverEntry设在分页内存中，当驱动加载成功，此函数在内存中移除。

11、《Windows驱动开发技术详解》的笔记-第4页

关于"#ifdef __cplusplus extern "C" " 收藏
Microsoft-Specific Predefined Macros __cplusplus Defined for C++ programs only.
意思是说，如果是C++程序，就使用extern "C"{，而这个东东，是指在下面的函数不使用的C++的名字修饰，而是用C的

The following code shows a header file which can be used by C and C++ client applications:

```
// MyCFuncs.h
#ifdef __cplusplus
extern "C" { //only need to export C interface if  used by C++ source code
#endif



__declspec( dllimport ) void MyCFunc();
__declspec( dllimport ) void AnotherCFunc();
```

```
#ifdef __cplusplus
}
#endif
```

当我们想从C++中调用C的库时，（注，驱动是用C写的，连new、delete也不能用,郁闷）不能仅仅说明 一个外部函数，因为调用C函数的编译代码和调用C++函数的编译代码是不同的。如果你仅说明一个外部函数， C++编译器假定它是C++的函数编译成功了，但当你连接时会发现很可爱的错误。解决的方法就是指定它为C函数：

```
extern "c" 函数描述
```

```
指定一群函数的话：
extern "C"{
n个函数描述
}
```

```
如果想C和C++混用的话：
#ifdef _cplusplus
extern "C"{
#endif
n个函数描述
#ifdef _cplusplus
}
#endif
```

extern "C"表示编译生成的内部符号名使用C约定。

C++支持函数重载，而C不支持，两者的编译规则也不一样。函数被C++编译后在符号库中的名字与C语言的不同。例如，假设某个函数的原型为：

```
void foo( int x, int y );
```

该函数被C编译器编译后在符号库中的名字可能为_foo，而C++编译器则会产生像_foo_int_int之类的名字（不同的编译器可能生成的名字不同，但是都采用了相同的机制，生成的新名字称为“ mangled name ”）。_foo_int_int这样的名字包含了函数名、函数参数数量及类型信息，C++就是靠这种机制来实现函数重载的。下面以例子说明，如何在C++中使用C的函数，或者在C中使用C++的函数。

```
//C++引用C函数的例子
//test.c
#include &lt;stdio.h&gt;
void mytest()
{
   printf("mytest in .c file ok\n");
```

```
}


//main.cpp
extern "C"
{
   void mytest();
}
int main()
{
   mytest();
   return 0;
}
```

//在C中引用C++函数
在C中引用C++语言中的函数和变量时，C++的函数或变量要声明在extern "C"{}里，但是在C语言中不能使用extern "C"，否则编译出错。

```
//test.cpp
#include &lt;stdio.h&gt;
extern "C"
{
  void mytest()
  {
    printf("mytest in .cpp file ok\n");
  }
}


//main.c
void mytest();
int main()
{
  mytest();
  return 0;
}
```

//综合使用
一般我们都将函数声明放在头文件，当我们的函数有可能被C或C++使用时，我们无法确定是否要将函数声明在extern "C"里，所以，我们应该添加：

```
#ifdef __cplusplus
extern "C"
{
#endif
//函数声明
```

```
#ifdef __cplusplus
}
#endif
```

如果我们注意到，很多头文件都有这样的用法，比如string.h，等等。

```
//test.h
#ifdef __cplusplus
#include &lt;iostream&gt;
using namespace std;
extern "C"
{
#endif
void mytest();
#ifdef __cplusplus
}
#endif
```

这样，可以将mytest()的实现放在.c或者.cpp文件中，可以在.c或者.cpp文件中include "test.h"后使用头文件里面的函数，而不会出现编译错误。

```
//test.c
#include "test.h"
void mytest()
{
  #ifdef __cplusplus
  cout &lt;&lt; "cout mytest extern ok " &lt;&lt; endl;
  #else
  printf("printf mytest extern ok n");
  #endif
}
//main.cpp
#include "test.h"
int main()
{
  mytest();
  return 0;
}
```

12、《Windows驱动开发技术详解》的笔记-第117页

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器所创建。确切的说是I/O管理器负责加载的。

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，因此形成一个设备链。

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。一般将全局变量以设备扩展的形式存储。

NT式驱动的基本结构
DriverEntry-------主要对驱动程序进行初始化工作，由系统进程所调用。
在驱动程序中UNICODE字符串用数据结构UNICODE_STRING来表示。

IoCreateDevice-------创建设备对象

IoCreateSymbolicLink---------创建符号链接
符号链接可以理解为设备对象的一个别名。设备对象的名称只能被内核模式的驱动识别，而别名也可以被应用程序识别。比如，C盘的符号链接为“C：\”，其实真正的设备对象是"HarddiskVolume1"。在内核模式下符号链接是以"\??\"开头的。而用户模式下则是以"\\.\"开头。

WDM式驱动的基本结构
WDM是建立在NT驱动模型的基础之上。
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。物理设备对象PDO和功能设备对象FDO。其关系是“附加”与“被附加”关系。

当PC插入某设备时，PDO会自动创建，由总线驱动创建。PDO不能单独操作设备，必须配合FDO。

PDO被称为底层驱动，FDO被称为上层驱动，“上”是指靠近发出I/O请求的地方，而“下”是指靠近物理设备的地方。FDO和PDO之间还可能存在过滤驱动（可能有很多个。）

WDM驱动提示用户加载FDO，如果该设备由微软提供，则会自动进行安装FDO。这种思路，导致了WDM模型支持即插即用的功能。

AddDevice是WDM驱动所独有的，NT没有。负责创建设备对象。

在NT式驱动中，DriverUnload主要负责删除设备和取消符号链接，而在WDM中，这部分操作被IRP_MN_REMOVE_DEVICE IRP处理函数所负责。

IoDetachDevice-------从设备链上摘除设备对象

驱动程序的垂直层次结构
其实NT驱动也可以分层。主要通过一个设备附加到另一个设备之上。形成设备栈。
设备的创建顺序是：先创建底层PDO，再创建高层FDO，这就是设备栈的生长方向，从底层到高层。中间可能夹杂这各种过滤驱动，每层的设备对象由不同的驱动程序所创建。有的系统自带，有的程序员自己编的。

驱动程序的水平层次结构
同一驱动程序所创建出来的设备对象的关系称为水平层次关系。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可以寻找水平层次的下一个设备对象。

驱动程序的复杂层次结构
将PCI总线想象成根总线（其实PCI不是根总线，为了方便理解），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，并为每一个设备创建PDO,然后每一个PDO上面必须有一个FDO。

# 《Windows驱动开发技术详解》

## 版权说明