**第一章 开始一个驱动项目**

在本章中我将对驱动开发的历史作一个简要的回顾。从80年代中期起，我开始涉足个人计算领域，那时也正是IBM刚刚开始出售搭配着MS-DOS操作系统的个人电脑。而就在今天，我们也依然能感受到当年IBM和Microsoft的决定所带来的影响。所以一定的历史背景知识会让你更好的理解如何编写驱动程序。

在本章中，我将简要地介绍一下可以运行WDM的两个截然不同的系统。Windows XP和Windows 2000,NT一样为WDM驱动提供了一个框架。在这个框架中，在完成应用程序或其它的驱动请求的I/0操作时，WDM驱动可以扮演的角色是该框架预先定义好的;而Windows Me和之前的Windows 9x以及Windows 3.x一样是一个相对“自由”的系统，所以在这里，驱动可以担当不同的角色。

如果确实有需要自己写一个驱动，那么在项目开始时要做的第一步是确定你需要什么样的驱动。本章介绍了很多不同的驱动类型，希望能帮助你在这个问题上有个明确的答案。

最后，本章的末节就项目中应当考虑到的一些事情，列出了一个参考清单。

**设备驱动的历史**

早期个人电脑中的英特尔处理器只能提供640KB“真正”内存的能力——之所以称之为“真正”的内存是因为这种内存以内存芯片的形式实实在在的存在在系统中，而处理器则通过一个20位的物理地址来访问这些内存。这种处理器只提供了一个操作模式，即所谓的“实模式”。在这“实模式”当中，处理器通过两个16位寄存器值组成一个20位的内存地址，并以此来处理每一个涉及内存操作的指令。从那时候开始，就已经有了扩展槽的概念。这样可以使得那些前卫的电脑用户可以购买独立的板卡，并把它们安装到系统中。这些板卡往往带有DIP开关（后来，这些DIP开关变成了跳线针），并附有使用说明，以便你调整I/O设置。那时，为了正确地设置I/O设置，你常常得记下系统中I/O的设置和中断使用的情况。后来，MS-DOS使用CONFIG.SYS文件来帮助简化这种设置。MS-DOS根据这个文件的配置自动载入系统中原有设备的实模式驱动以及那些扩展卡的实模式驱动。当然这些驱动都是用汇编语言编写的，所以经常通过INT指令来获得BIOS和MS-DOS的服务。最终用户也被迫学习怎样在一个命令行运行一个应用程序，而开发者当然只能学着在程序中直接操作显卡，键盘和鼠标等硬件，因为无论是MS-DOS，抑或BIOS都没有提供足够的功能来满足开发要求。

随后IBM又发布了AT架构的基于英特尔80286芯片的的个人电脑。这种286芯片新增了“保护模式”。保护模式使用一个24位的段地址（通过一个存放在16位的段寄存器中的段选择子来来确定这个段地址）和一个16的偏移地址，使得应用程序可以访问16MB的主内存和扩展内存。但是，MS-DOS本身还是一个实模式的操作系统，没有提供访问保护模式的能力，所以一些公司开始设计DOS扩展器，以使开发者把他们的实模式程序移植到保护模式下来，能够访问到当时市场上开始出现的大容量内存。正因为实模式的MS-DOS还统治着个人电脑市场，因此这个时候的驱动技术依然停留在原先的水平上。

个人认为，英特尔的80386芯片是个人电脑技术发展史上的一个分水岭。通过分页表，80386芯片允许程序可以访问多达4GB的内存，同时它提供了32位的数值运算能力和32位的内存寻址能力。伴随着80386芯片的问世，软件工具市场再度活跃起来。很多程序对内存和处理器的要求越来越高，各编译器厂商和做DOS 扩展器的公司纷纷推陈出新争夺市场份额。而那时的设备驱动还依然是用汇编语言写成的16位的实模式的驱动，依然得通过CONFIG.SYS文件来完成安装驱动，所以对最终用户来说还得手工配置各种扩展卡。

以后的日子里，处理器芯片基本上只在性能和能力上进一步提高。就在此时，在我写这一章的时候，运行速度在1GHz以上，拥有50GB容量的硬盘和512MB（甚至更多内存）的电脑已经成了普通配置，对大多数人来说价格上也完全能够承受。

伴随着硬件的发展，操作系统技术也在发展着。在文本模式和图形模式之间，大多数人，就算是系统软件的开发者，都更愿意于使用前者。在图形操作系统上，Microsoft虽然晚了一步——Apple公司早一步推出了它的Macintosh，但却用Windows系列操作系统统治了这个市场。早先的Windows系统只是实模式MS-DOS的一个图形“壳（shell）”。后来，针对常见的硬件，比如显示适配器，键盘，鼠标，出现了一些驱动程序集。这些驱动主要是由汇编语言开发的，带着.DRV扩展名的可执行文件。

在AT架构的电脑流行后，Microsoft开发了支持了保护模式的Windows。与此同时，Microsoft升级了实模式的驱动，使之可以运行在保护模式下。然而，除了那些“标准”的Windows硬件外——显示卡，键盘和鼠标等，其它的硬件依然由MS-DOS的实模式的驱动操纵管理。

随着386芯片的电脑普及，Microsoft发布了能够充分利用386芯片提供的虚拟内存管理能力的Windows 3.0。尽管如此，每一个新硬件还是需要一个实模式的驱动。而且现在一个新的重要的问题摆在了眼前。为了使MS-DOS程序能够支持多任务（一个能够让用户接受Windows的新特性），Microsoft设计了一个虚拟机的操作系统。和Windows的图形界面一样，每一个MS-DOS程序运行在一个属于它自己的虚拟机上。但是这些MS-DOS程序往往通过发出IN和OUT指令试图直接操作硬件，读取或写入设备内存以及处理来自硬件的中断等。更重要的是，两个或两个以上的共享处理器时间的程序很有可能对硬件发出相互冲突的指令。在显示卡，键盘和鼠标的使用上，这些程序都有可能引起冲突。

为了可以让多个应用程序共享硬件资源，Microsoft推出了所谓的虚拟设备技术。这种技术其主要目的在于“虚拟化”一个硬件设备。之所以这类驱动一般被称为VxD是因为它们大多数都有一个类似VxD.386的文件名，其中的x则用来表明该驱动支持的硬件类型。使用这项技术，Windows 3.0制造了这样一种假象，系统中的各个虚拟机似乎都运行各自独立的硬件设备上。但是这些设备本身，在很多时候还是依然由MS-DOS的实模式驱动所驱动。一个VxD驱动的任务在于在硬件和软件之间起到一个类似于中间人的作用，它首先会截取应用程序直接操作硬件的企图，然后为了运行MS-DOS驱动，暂时性地切换处理器的工作模式到一种和实模式非常类似的，所谓的虚拟8086模式（virtual 8086 mode）下。

这种技术其实只是权宜之计，它唯一的作用是可以允许硬件平台的平稳增长，也为操作系统的发展作了一个铺垫。它也是Windows 3.0的很多Bug的根本原因。Microsoft为此准备的答案便是OS/2系统。这个系统是Microsoft和IBM——以一种20世纪的“黓契”,合作开发的。

Microsoft版的OS/2后来成了Windows NT。NT的第一次发布是在90年代早期，比Windows 3.1的发布时间略微晚一点。Microsoft从零开始全新开发了NT，目的要使NT成为一个安全的，可持续服务的Windows平台。Windows NT下的驱动使用了全新的技术，和当时非常流行的另两种驱动技术可以说没有任何相同之处。NT的驱动几乎全部用C语言来编写。这样当新的CPU架构出现时，在无须更改驱动源代码的情况下，只要重新编译一次就可以了。

从Windows 3.0的时代开始，软件划分成了两个不同的世界：用户模式和核心模式，这一点对今天的我们的影响依然很大。人们购买电脑要使用的一般软件和游戏都属于用户模式的软件。但这些是不被“信任”的应用程序，它们被认为不能安全可靠地（甚至是可能有恶意的）和硬件或者其它的程序打交道。象你我编写的驱动和操作系统本身则属于核心模式的程序，它们是被“信任”的程序，可以随心所欲地使用系统资源。虽然Windows 3.0根据模式分离了不同的程序，但没有一款Windows[[1]](#footnote-1)[1]系统（甚至是Windows ME也一样）真正使用内存保护的手段来使系统运行的安全可靠。安全的责任[[A1]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_msocom_1)落到了Window NT和它的后继者身上，这些系统禁止用户模式的程序读取和修改由系统核心管理的资源。

然而，一直要到最近Windows NT才能在普通的个人电脑运行得很好，所以Microsoft不得不保持并继续Windows这条产品线。因此Windows 3.0之后，便有了3.1，3.11和95。从95开始，如果你要写一个设备驱动，你就得准备写一种称之为VxD的驱动了。这种驱动其实就是一个32位的保护模式的可执行文件。也就是从那时开始，用户终于可以丢掉他们的有关I/O的使用记录的备份了，因为从这时开始操作系统的新的即插即用的功能在某种程度上可以自动地识别和配置硬件设备了。可是作为一个硬件设备驱动开发者，你或许还得为你尚未升级到新系统的用户准备一个实模式的驱动。与此同时Windows NT也发展到了3.5，4.0。为了支持这些系统，“第三个驱动[[A2]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_msocom_2)”是必须的，而且之前你的开发经验大都并不适用于这些系统。

一个设备需要三个驱动，这真的太多了。为此Microsoft设计了一种新的设备驱动，Windows驱动模型（WDM），并在Windows 95的后继者Windows 98和Windows Me中加入了这种新特性。同时Microsoft也使Windows NT 4.0的后继者Windows XP和Windows 2000支持了这种新的驱动。在Windows Me中，MS-DOS的存在仅仅是出于“礼貌”而已，对硬件制造商来说再也不必为他们的硬件准备实模式驱动了。因为WDM——至少从设计的初衷来看，在所有的平台上几乎是一样的，这样使得只须写一个驱动成为可能。

就在今天，我们依然可以感受到早期的个人电脑的架构和第一版的MS-DOS带来的影响。今天的用户偶尔还是会打开他们的电脑，安装一些扩展卡。但是和以前相比，今天的电脑总线已经发生了很大的变化，功能强劲多了。即插即用和互联外围设备总线[[A3]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_msocom_3)（PCI）的存在基本上可以让用户无须过问I/0，内存和中断请求的使用情况。当然BIOS[[2]](#footnote-2)[2]还在，但是它的主要工作是负责启动系统，并将在这个过程检测到的配置情况通知操作系统（Windows XP或Windows Me）。最后，WDM驱动程序的后缀名还是沿用第一个实模式驱动的后缀名，.SYS。

**概述驱动开发者眼中的操作系统**

Windows驱动模型（WDM）为设备驱动提供了一个框架，使它们可以同时运行在两种不同的系统中——Windows 98/Windows Me和Windows 2000/Windows XP。正如上面的设备驱动的发展的简要回顾中提到的一样，这两套系统是两条产品线的并行发展的产物。在下面的章节中，我用“98/Me”来表示前一对系统，用XP表示后一对系统。尽管对普通用户来说，这两种系统看上起差不多，但其实它们内部的工作方式有很大的不同。在本节中，我将对这两种系统作一个简要的概述。

**Windows XP概述**

图1-1是关于Windows XP操作系统的相当简单的功能图，描述了我认为对驱动开发者来说很重要的特性。在每一个Windows XP运行的平台上都支持两种执行模式，应用程序要么运行在用户模式，要么就是运行在核心模式。如果一个用户模式的程序想从一个设备读取数据，就会调用一个相应的应用程序编程接口[[3]](#footnote-3)[3]（API），比如ReadFile。而另一个子系统，就象KERNEL32.DLL通过调用一个原生API[[4]](#footnote-4)[4]，NtReadFile功能来实现用户模式应用程序调用的API。有关原生API请参阅下面的补充说明[[A4]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_msocom_4)。

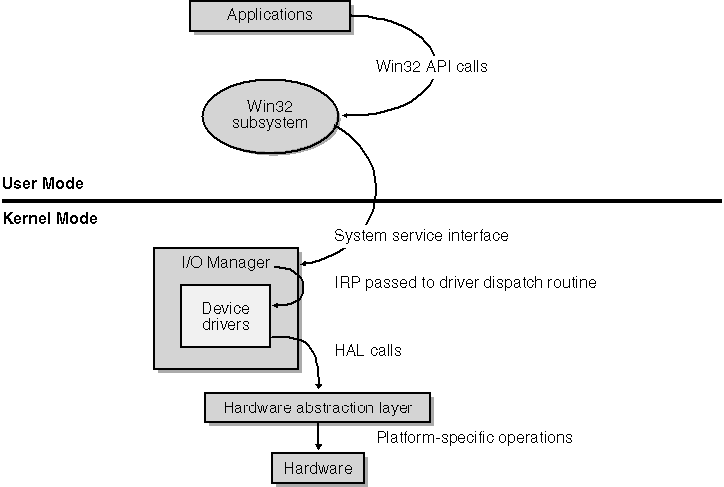


图1-1：Windows XP结构

我们经常说到NtReadFile是一个被称作I/O管理器[[A5]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_msocom_5)的系统组件的一个功能部分。然而，I/O管理器这个术语或许会有让人误解的地方，因为系统中没有这样一个有着同样名字的可执行模块。但是当我们在讨论为我们的驱动提供服务的一系列系统服务的时候，为这些系统服务准备一个名字是很有必要的，而我们经常使用的就是“I/O管理器”这个名字。

和NtReadFile类似，很多例程都提供这样的服务。为了响应一个应用程序的和一个设备交互的请求，它们工作在核心模式。这些例程首先对传入的参数作校验，确保它们的操作或者数据的存取发生在用户模式所允许的范围内，避免不经意的违背安全性的操作发生。然后，它们创建一个称作I/O请求包[[A6]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_msocom_6)（IRP）的结构，并把这个数据包传递到设备驱动的入口。在调用ReadFile这个例子中，NtReadFile创建一个IRP结构，设置主要功能标志[[A7]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_msocom_7)为IRP\_MU\_READ（DDK[[5]](#footnote-5)[5]头文件中的一个常量）。这个时候发生在例程中的处理细节往往有所不同，但是对一个象NtReadFile这样的例程来说一种可能的做法是返回到用户模式的呼叫者，并告之由IRP描述的操作尚未完成。由用户模式应用程序决定是继续执行业务流程，之后再等待请求的操作完成，或者立即进入等待状态。无论用户模式应用程序的决定怎样，驱动都将在独立于应用程序的情况下完成应用程序的请求。

|  |
| --- |
| **原生应用程序编程接口（原生API）**  NtReadFile是所谓的Windows XP的原生API的一部分。原生API的存在是有其历史原因的。在最初的Windows NT中包含了几个子系统，用以实现一些新的或已经存在的操作系统的语义上的功能。这里面有OS/2子系统，POSIX[[A8]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_msocom_8)子系统和Win32子系统。Windows NT通过用户模式的调用到原生API来实现这些子系统，而原生API本身是在核心模式中实现的。  一个名为NTDLL.DLL（我总感觉这是多余的）为Win32调用者实现原生API。这个DLL中的每个入口函数只是对一个核心模式函数调用的一层很浅的包装，实际的工作是由这个核心模式的函数来完成的。该调用过程利用平台相关的系统服务接口把控制权由用户模式传递到核心模式中。在较新的Intel处理器上，这个系统服务接口使用了SYSENTER指令。而在较老的Intel处理器上，这个接口使用功能标志为0x2E的INT指令。在其它的处理器上，则该接口使用其处理提供的相应的机制。作为驱动开发者，你和我没有必要理解这个机制的细节。我们只需要知道这个机制可以允许一个执行在用户模式的程序可以调用一个执行在核心模式的例程，而这个执行在核心模式的例程最终将返回到用户模式的调用者那里。在整个过程中，不存在线程上下文环境的切换，只有代码执行的特权级别发生了变化（还有一些其它的只有汇编程序员才需要注意和关心的细节问题）。  对大多数开发者来说Win32子系统是最熟悉的了，因为它实现了经常和Windows图形用户界面相关的函数。而其它的几个子系统随着时间的流逝慢慢被冷落了。但是，原生API依然存在，Win32子系统依然以本书例子所描述的方式依赖于它提供的服务。 |

为完成在IRP中所描述的任务，一个设备驱动最终需要操作到相关的硬件设备。在读取一个可编程I/O（P/O）设备的IRP\_MU\_READ的例子中，读取操作可能通过一个I/0端口读命令进行，也可能通过设备实现的一个内存寄存器[[A9]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_msocom_9)来完成。尽管执行在核心模式，可以直接和它们的设备联系，驱动利用硬件抽象层提供（HAL）的功能操作硬件设备。一个读取操作因此很可能调用READ\_PORT\_UCHAR从一个I/O端口读入一个数据字节，而硬件抽象层例程则使用平台相关的方法来真正完成这个操作。在一台x86的电脑上，硬件抽象层使用IN指令。或许在未来的Windows XP平台上，它会使用一个取内存操作。

等到驱动结束了一个I/O操作后，它调用一个特别的核心模式的服务例程来完成这个IRP。在处理一个IRP过程中，完成操作是其最后一步。之后，处于等待状态的应用程序就可以继续得以执行下去。

**Windows 98/Windows Me概述**

可以以图1-2展现的方式来观察Windows 98/Windows Me的架构。虚拟机管理器[[A10]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_msocom_10)（VMM）是操作系统的核心，它的主要工作是建立一个或更多的虚拟机，它们共享着一个物理机器上的硬件资源。Windows 3.0中的虚拟设备驱动的最初目的就是能够虚拟化一个硬件设备，使VMM能够创建一种每个虚拟机各自都拥有一套完整的硬件资源的“假象”。今天的Windows 98/Me的VMM，其架构还是采用了Windows 3.0的架构，其中增加了一些能够处理新硬件的功能和对32位程序的支持。

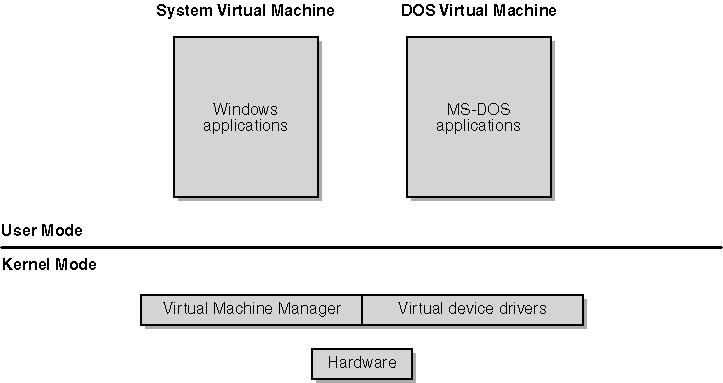


图1-2：Windows 98/Me架构

和Windows XP不同，Windows 98/Me在处理I/0操作是，因I/O对象的不同而不同。对磁盘操作，通讯端口操作，以及键盘操作等的处理方式上，Windows 98/Me有着很大的不同。而且Windows在对待32位应用程序和16位应用程序上也有着根本的不同。

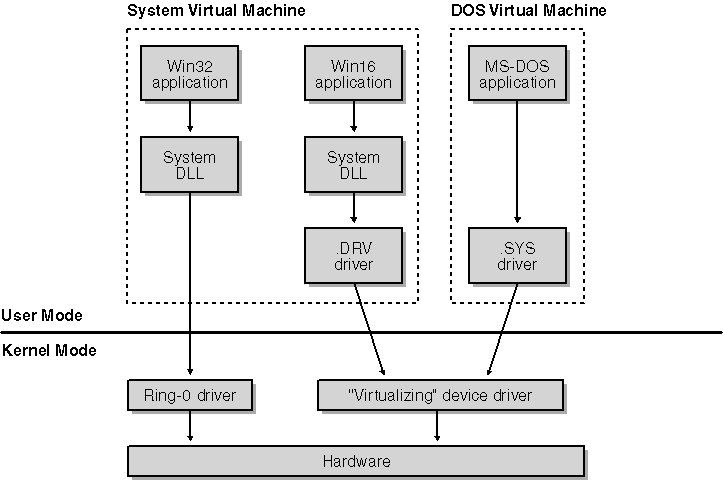


图1-3：Windows 98/Me中的I/O请求

图1-3的左半部分显示了系统是如何为32位的应用程序的I/O请求提供服务的。首先，一个程序调用一个Win32 API，比如ReadFile，而这个API是由一个系统DLL（KERNEL32.DLL）提供服务的。但是这里ReadFile的调用也是有限制的，当应用程序需要读取磁盘文件，通讯端口和其它的设备，而这些同时又拥有WDM驱动时，程序才能够调用ReadFile。如果针对的是其它的设备，程序就必须使用其它基于DeviceControl的特别的方式。而且就系统DLL的实现来看，Windows 98/Me和Windows XP的也是不同的。在ReadFile的用户模式的实现中就会判断参数的正确性——而在Windows XP中这一步是实现在核心模式中的，然后使用某种特别的方法转入核心模式的驱动中。这种方法因为对象的不同而不同，对磁盘文件是一种方法，对通讯端口又是一种方法，而对WDM的设备而言又有一种方法。这些机制都在从用户模式转入到核心模式时都利用到了中断30h，除此之外，它们间再无其它类似的地方了。

图1-3的中间部分显示了16位基于视窗的应用程序（Win 16应用程序）是如何进行I/O操作的。图的右半部分则说明了MS-DOS程序的控制流程。在这两种情况下，用户模式的程序直接或间接地调用用户模式驱动提供的服务，而这些提供的服务的驱动本身常常是可以运行在裸机系统[[A11]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_msocom_11)上的程序。COMM.DRV就是这样的一个例子。Win16程序在执行串口操作时都间接的用到这个16位的DLL提供的服务。（从Windows 95开始，COMM.DRV就已经是个独立的驱动，能够截取3号和4号中断请求信号，然后发出IN和OUT指令和串行芯片联系。）一个虚拟通讯设备驱动[[A12]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_msocom_12)（VCD）截取端口I/O操作，以防止两个虚拟机同时存取同一个端口的情况出现。类比也许不是很恰当，从某种角度来看这些用户模式的驱动通过截取I/O操作提供了一个类似于“API”的接口。而象VCD这样的“虚拟”驱动则通过模拟硬件的操作为这些伪API调用提供服务。

在Windows XP中所有的核心模式的I/O操作都用到了一个公共数据结构（就是IRP），相反在Windows 98/Me下就算一个程序的请求已经传递到了核心模式，这样的一致性也是不存在的。符合端口驱动功能调用的串口驱动是由VCOMM.VXD进行高度的。另一方面，磁盘驱动则工作在一个由IOS.VXD实现的数据包驱动的层次结构中。其它类型的设备依然使用各自不同的机制。

但是就WDM驱动来说，其在Windows 98/Me下的内部结构非常相似于Windows XP下的WDM。系统模块NTKERN.VXD包含了很多Windows NT内核支持的Windows 98的实现。NTKERN创建IRP，把它传送到WDM驱动的方式和Windows XP下完全一致。事实上，WDM驱动几乎不能分辨这两种不同的环境[[6]](#footnote-6)[6]。

当然，无论WDM在Windows XP下和Windows 98/Me下是如何的相似，它们之间还是有很重要的区别。在涉及到对驱动开发者来很重要的不同时，你会发现本书中有关的兼容性说明。附录A提供了这样一种方案，通过该方案你可以在Windows 2000/XP和Windows 98/Me之间使用一个编译好的驱动程序，尽管存在着这两种系统间的差异。

**你需要什么样的驱动？**

Windows XP包含了很多种驱动。图1-4显示了其中的一些驱动类别。

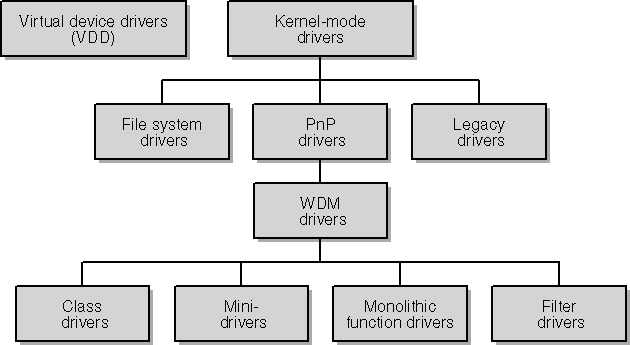


图1-4：Windows XP中的设备驱动类别

 虚拟设备驱动（VDD）。这是一种用户模式的驱动，它允许在Intel x86平台上MS-DOS应用程序的硬件操作。VDD利用I/O权限位来截取端口操作，进而为那些被编制为在裸机上直接和硬件打交道的程序模拟出硬件操作来。不要把Windows XP的VDD和Windows 98/Me的VxD混为一谈。它们都被称作虚拟设备驱动，而且它们的基本目的也都是虚拟硬件，但是它们却采用了截然不同的软件技术。

 在核心模式驱动这一类别下包含了几种子类别。其中PnP驱动是一种理解Windows XP的即插即用协议的核心模式的驱动类型。准确说来，本书所讲的并非是其它的驱动，而正是PnP驱动。

 WMD驱动。它也是一种PnP驱动，支持电源[[A13]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_msocom_13)管理协议，在源代码级上和Windows 98/Me以及Windows 2000/XP兼容。在WDM这个类别下，你可以发现存在着好几种子类别的驱动：类型驱动[[7]](#footnote-7)[7]，它管理的设备属于一些定义良好的设备类型；迷你驱动，生产商用来为类型驱动提供某些特定的功能；完全功能驱动。这种驱动在一个驱动中提供了支持一个硬件设备所需的全部功能；以及过滤驱动。为一个特定的设备“过滤”I/O操作，以达到增加或修改设备的某些行为的目的。

 文件系统驱动。这种驱动在本地硬盘上或者通过网络连接来实现标准的个人电脑的文件系统模型（这个模型包含了一个树形的目录结构，在这个目录结构中是一些命名文件）。它们也是核心模式驱动。

 老式设备驱动。这种核心模式驱动不需要其它驱动的帮助，直接控制一种硬件设备。比如Windows NT早期版本下的驱动，这些驱动在没有更改的情况下运行在Windows XP下面。

这些驱动类别间的区别并非总是显得很重要，就象我在我以前的《Windows 95的系统编程》（Microsoft出版社，1996年出版）一书中提到的那样，——当然这里我并不是在暗示你应该买全我的书。尤其是，我并没有老是按照上面的分类来严格区别WDM和PnP驱动。其实它们的区别更多的是体现在现象上的：是否能够同时运行在Windows 2000/XP和Windows 98/Me中。当然，尽管没有严格区别使用技术术语，当需要的时候我会很小心地说明和具体系统相关的问题。

有这么多类别的驱动可供选择，对驱动开发新手或者管理员来说很难就手上现有的事件选择一个合适的驱动类别。对有些设备而言，你根本不需要写驱动，因为Microsoft已经为你准备了一些通用驱动。这里就有这样的例子：

 SCSI兼容，ATAPI兼容的大容量存储设备。

 任何完全符合预先定义的规格的USB接入设备，当然前提是你愿意接受标准的Microsoft驱动附带的限制。

 标准串口和PS/2鼠标。

 标准键盘。

 没有加速或者其它特别功能的视频卡。

 标准并口和串口端口。

 标准软盘驱动器。

**WDM驱动**

对很多Microsoft没有直接提供支持的设备来说，你必须为之写一个WDM驱动。当然

你首先得确定你需要一个什么样类别的驱动：是一个完全功能驱动，还是一个过滤驱动，或者一个迷你驱动。对开发者来说，也许不需要你编写类型驱动。为了能够支持尽可能多的设备，Microsoft把这个特别的类型驱动开发留给了它自己。

**WDM迷你驱动**

一种普遍的做法是，如果Microsoft已经为你准备支持的设备准备了一个类型驱动，那么你应该定一个配合这个类型驱动工作的迷你驱动。你写的这个迷你驱动主要负责控制设备，而设备的管理工作则落到了类型驱动身上。你调用类型驱动中的例程，类型驱动也会回调到你的迷你驱动来完成设备相关的任务。迷你驱动的任务会因为设备类型的不同有很大的不同。

这里给出一些你应该计划为之写一个迷你驱动的设备类型：

* 非USB人机输入设备[[A14]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_msocom_14)（HID），包括鼠标，键盘，操纵杆，方向盘等。如果现有的Microsoft提供的通用的HIDUSB.SYS（这是Microsoft为USB HID设备提供的驱动）功能不能满足你的USB设备的要求，那么你也可以为之写一个HIDCLASS迷你驱动。这些设备的一个主要的特点是，它们能够通过一种描述数据结构来反映用户的输入。对这些设备来说，HIDCLASS.SYS扮演着类型驱动的角色，为Direct-Input和其它的处于高层的软件需要用到的功能，所以你的工作主要就是和HIDCALSS.SYS打交道。如何配合HIDCALSS.SYS工作是件很困难的事，本书的后面会有较大的篇幅来讨论这个问题。顺便一提，HIDUSB.SYS本身也是一个HIDCLASS的迷你驱动。
* Windows图像获取设备（WIA），包括扫描仪和摄像头。你应该为你的设备写一个WIA的迷你驱动——一个COM类型的接口库，以提供特定的功能。
* 流设备，比如音频，DVD和视频设备，以及针对多媒体数据流的纯软件过滤器[[A15]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_msocom_15)。你应该为这些设备写一个流迷你驱动。
* 非传统总线上的网络接口设备，比如USB或1394的网络设备。正如DDK文档中描述的一样，要获得和传统网络接口设备一样的处理过程，你必须为这些设备写一个符合网络驱动接口规格[[A16]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_msocom_16)的迷你端口的驱动[[A17]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_msocom_17)。但是这样的驱动在不同的操作系统间是不可移植的——尽管为之需要的改动并不是很大，你需要为不同的平台准备几个不同版本的这样的驱动。
* 视频卡。这种设备需要一个视频迷你驱动来配合视频端口的类型驱动工作。
* 打印机。打印机不需要核心模式的驱动，它要求用户模式的DLL库就行了。
* 电池。Microsoft已经有了针对电池的通用类型驱动。你需要写一个能够和BATTC.SYS协调工作的迷你驱动（在DDK文档中把这一类型的驱动称作迷你类型，但它们是同一回事）。

**WDM过滤驱动**

有时候，也许一个硬件的工作方式非常接近一种得到广泛承认的标准，于是一个Microsoft的通用驱动也许已经能够驱动这个硬件了。在某些情况下，你可以写一个过滤驱动以改变通用驱动的某些行为，使之能够驱动你的硬件。但是，这种情形并不经常出现，因为要改变一个通用驱动的操作硬件的方式并不容易。在十六章中，我将详细讨论这个问题。

**完全功能的WDM驱动**

撇开下一节我谈到的例外不说，大多数硬件都需要一个我称之为“完全功能”的驱动。这种驱动能够不依赖于其它的驱动，独立处理控制硬件的所有细节。

当形势需要完全功能的WDM驱动时，为了能在Intel的x86平台上的所有操作系统上使用同一个驱动，最好能遵循下面的步骤。首先，应该使用最新的DDK——本书的配套光盘的例子使用的是.NET DDK的测试版本。你可以用IoIsWdmVersionAvailable来检测当前使用的是何种操作系统。如果正好使用了Windows 2000或者Windows XP，那么你就可以调用MmGetSystemRoutineAddress来得到一个Windows XP特有的函数指针。在Windows 98/Me环境下，我建议最好在你的驱动中使用定义了一些重要的核心函数的WDMSTUB.SYS这个文件。否则，你的驱动在Windows 98/Me下面不可能成功运行，因为没有定义这些函数的引入库。

下面列出的一些设备你应该为它们编写一个完全功能的驱动：

* 智能卡读卡器——除了连接到串口的读卡器
* 数模转换器
* 用于支持专有身份标签读/写传感器的ISA卡

|  |
| --- |
| **驱动的二进制兼容性**  一开始，WDM的设计初衷就是要在各个Windows平台间做到驱动的二进制级别上的兼容性。后来随着版本的一再更新和新的设计思想的出现，越来越多的核心函数出现在从Windows 98以来的版本中，这些函数对驱动的健壮性和编程的便利性很重要，而且有时候这些函数的作用是本质性。在十四章中讨论到的IoXxxWorkItem系列函数就是这样的一个例子。它们是在Windows 2000中出现的，在应用时你必须要用到这些新的函数，而不是以前类似的，但健壮性差的ExXxxWorkItem系列函数。如果没有不采取额外的方法，那么你的驱动在Windows 98/Me下就不能够运行起来，原因很简单，操作系统没有提供这些必要的函数功能。MmGetSystemAddress是另外一个这样的例子，在Windows 98/Me下面不能使用这个函数。不幸的是，你甚至不能动态地选择你该使用哪个系列的函数。更糟的是，在WHQL[[c18]](http://writeblog.csdn.net/fckeditor/editor/fckeditor.html?InstanceName=Editor_Edit_FCKEditor&Toolbar=Default" \l "_msocom_18)的认证中有一项指标就是检查你的驱动中是否用到了ExXxxWorkItem的函数。  在Windows 98/Me下面，一个名为NTKERN的VxD驱动实现了WDM中提供核心支持函数的一个子集。就象在附录A中要提到的一样，NTKERN需要有新的输出符号，以便运行时符号装载器使用。你可以手工定义自己的输出符号，就象WDMSTUB所做的那样：为了能够开发我一直推崇的二进制兼容的驱动，WDMSTUB定义了一些在Windows 98/Me下没有的符号。  在本书的配套光盘中有一个WDMCHECK的实用工具，你可以在Windows 98/Me平台下用它来检测一个驱动是否要用到某些Windows 98/Me所没有的引入库。如果你开发了一个在Windows XP下运行良好的驱动，我建议你还应该把这个驱动复制到Windows 98/Me的系统中，然后用这个工具检测一下。如果WDMCHECK报告你的驱动用到某些不支持的函数，那么第二件你应该做的事是检查一下是否已经把WDMSTUB添加到你的驱动之中。如果还没有添加，那么就把它添加到驱动中去，而如果已经添加了这个文件，那么就手动修改你的驱动，或者可以给我发个电子邮件——由我来修改WDMSTUB。这样做的结果是，你可以开发一个二进制兼容的驱动程序。 |

**其它类型的驱动**

由于Windows 98/Me和Windows 2000/XP之间架构上的差别，在某些场合下只有一个完全功能的驱动是不够的。在下面讲到的例子中，你应该准备两个驱动：一个Windows 2000/XP下的WDM驱动，一个Windows 98/Me下的VxD驱动。

* 串口驱动。Windows 98/Me的串口驱动为一个VxD驱动，它提供了一个VCOMM端口驱动接口，而在Windows 2000/XP里边这样的驱动则是一个WDM驱动，它提供了功能丰富的，同时经过严格定义的IOCTL接口。在这两个驱动之间没有任何的相似之处。
* 连接到串口的设备驱动。在Windows 98/Me下它是一个VxD驱动，通过调用VCOMM来和串口通讯，而在Windows 2000/XP下它却成了一个WDM驱动，通过调用SERIAL.SYS或者其它实现了IOCTL接口的驱动来和串口通讯。
* 非标准的大容量USB设备驱动。在Windows 98/Me下面，你需要写一个VxD格式的驱动，这个驱动能够融入到一个由不同驱动构成的层叠的I/O管理的树形结构中去。而对Windows 2000/XP而言，你写的完全功能的WDM驱动应该能够接受SCSI请求块并同时和底层的USB设备进行通讯。

Microsoft早在WDM架构出现之前就为下面两种类型的设备设计了一个可移植的驱动架构：

* 小型计算机系统接口（SCSI）适配器使用一个称为“SCSI迷你端口”的驱动。该没有使用标准的核心支持函数，根据系统的不同它使用了由SCSIPORT.SYS或SCSIPORT.VXD输出的特别的API。而这个“迷你端口”驱动在Windows系统间具有良好的可移植性。
* 网络接口卡使用的是“NDIS迷你驱动”。根据系统的不同，这种驱动使用了由NDIS.SYS或者NDIS.VXD提供的特别的API。曾经，NDIS迷你端口驱动的移植性很好，但到今日这种良好的可移植性差不丧失殆尽。网络协议驱动和那些提供过滤功能的所谓的“中间层”驱动都是围绕着NDIS建立起来的。

**项目管理要旨和注意事项**

如果你是一位开发主管，又或者负责向市场发布一款硬件设备，那么这里有关设备驱动的几件事是你必须要知道的。首先，你得决定是否需要一个定制的设备驱动。如果是，那么你还得决定你所需要的是哪种类型的驱动。前面的章节所描述的应该对你的决定会有所帮助，当然你或许还需要一位这方面的专家，就这个问题提出一些建议。

假如事先的评估工作使你相信需要一个定制的设备驱动，那么你就需要寻找一位称职的程序员。而不幸的是，视窗驱动模型（WDM）的编程是件相当困难的事，只有非常有经验（当然雇用这样的程序员的费用也相当昂贵的）才能够胜任。有一些公司自己有这样一些的核心程序员，但是大多数公司则没有这样的条件。如果你的情况正和后者相同，那么你可能的选择不外乎有，从现有的人员中培养需要的程序员，聘请有相应技术的程序员，雇用顾问或者合同程序员，以及把项目外包给其它专门做这样的项目的公司做等。所有这些选择都各自有各自的长处和短处，你要做就是结合你独特的需要来左右权衡。

当就目标硬件如何工作的问题已经有了一个具体的描述之后，驱动的开发也就应该随之进行了。在驱动的开发过程中，你应该对发现的问题有清醒的准备，以根据需要调整这个规格描述，对于已有的硬件/固件和驱动的设计，也可能因为需要而再三考量和审查。灵活和愿意从头开始的准备对你将会有很大的帮助。

同时，对于驱动开发的时间和经费可能高于你先前的预测，你也应该事先有准备。这是因为很多的软件项目时间和经费到最后都往往超支。之所以有这样的超支出现主要原因在于，硬件设计人员和软件设计人员间的交流沟通困难，设计规格和DDK文档中说明上的模棱两可，以及各个部件中自身的缺陷，还有工程和制造方面的延迟等因素。

大多数情况下，你还想把你的硬件和软件一起送交到Windows硬件质量实验室进行检测，以便获得一个数字签证。它可以让你的产品的安装变得简单明了，同时给了你使用微软标识的权利。检测的大部分工作是自己完成的，你应该为这个检测准备相关的电脑设备，尽可能完善你的设备的检测要求，以免得到最后才发现项目中的不足之处。（举个例子说，如果要检测一个USB设备，你就应该在一个拓扑结构中准备一些不同种类的音频设备，哪怕是你的设备和音频或者其它类型的流媒体没有任何的关系。）

在准备Windows硬件质量实验室的论证过程中，还需要相关的属于业务框架里的一些准备工作。比如，最起码的一点，就是要求你从Dun和Bradstreet那里获得一个公司的邓氏编码[[8]](#footnote-8)[8]，一个Verisign颁发的数字签证。到目前为止，这个公司的邓氏编码是可以免费申请的，而Verisign的数字签字却是需要花钱的。所有这些工作都要求在不同的机构中进行，当然也是需要一定的时间。

及早注意最终用户是如何安装你的设备驱动的问题。很多扩展硬件的制造商往往都倾向于在光盘上带上一个定制的用户安装程序，而写这样一个安装程序可以让一个熟练的程序员花上几个星期的时间。眼下，基于网络的设备驱动库很流行，但是它要求你对这个软件的安装问题多花一份心思。

这里有两种方式可以让设备驱动程序为用户提供统计的和其它管理上的信息。Windows管理规范子系统为二进制数据间的信息交换提供了一种语言和信息交换机制独立的方式。对一些特定的设备，微软提供了标准的WMI类，而你的行业可能也会有自己的标准，你的程序应该和这些标准保持兼容。在第10章中，讲述了如何和微软的标准保持兼容性，但是对于如何遵循你自己行业的独有的标准，则是你们公司贸易部门代表的事了。

为用户提供管理信息的第二种方式是可以使用系统的事件日志功能。这个功能从Windows NT诞生起就已经存在了，它可以让系统管理员方便的了解到系统中最近发生的异常信息。你的设备驱动程序应该针对系统管理员可能感兴趣的事件作好事件日志，这可以让系统管理采取对应的措施。不管是谁在为你编写设备驱动，都应该资讯资深的系统管理员，以便确定哪些是应该在事件日志中记下来的，哪些是可以放弃的，这样就可以避免在事件中记录大量的日常的、非异常性的信息。如果你的设备驱动程序中还带有一个多国语言的资源包的话，那么对这些错误的语言信息的撰写，最好请一位经过培训过的专职人员来做。（我倒不是说，你的程序员没有这个能力来做这件事情，但也许在这件事上，他也许不是一个最好的选择。）

除了设备驱动外，你可能还需要相应的控制面板或者其它的程序。应该让一个用户交互方面的专家和设备驱动程序员一起来完成这件事情。而且因为所有这些额外的程序会和真正的设备驱动一起发布，所以它们很自然的成为Windows硬件质量实验室数字论证的一部分，它们应该和随设备驱动的完成而同时完成。

最后，不要低估设备驱动的重要性。拥有一个安装简单的设备驱动的重要性和产品本身相比一点也不差。简单来说，如意你的设备驱动使得操作系统崩溃了，那么检测者会对公众发出警告，而如果正好有人没有读到这样的信息，那么他会怒气冲冲地到商店去退货。那些因为你的设备驱动使得他们系统崩溃的顾客再也不会回头做你的生意。所以，在驱动开发中为节省成本的短视行为，将会在未来几年中对你的利益造成很大的负面影响。这对发展中国家的硬件制造者来说尤其重要，因为在那里开发主管总是想尽一切可想的办法来节约开发成本。我认为基于成本考虑的决定对设备驱动的开发来说是不合理的。

概括来说，在勾画设备驱动开发项目时，应该牢记下面这几条：

·评估所要开发的设备驱动，选择合格的开发人员

·编制完备的硬件规范可以使驱动开发得以进行

·准备好作为驱动调试用的硬件设备原型

·保证设备驱动和预想的那样能和相应的配件/固件协同工作

·在所有的操作系统上测试安装文件（INF）

·完成控制面板和其它附加程序的开发

·完成WMI（Windows管理规范）和事件日志功能，并对之进行测试

·内部根据Windows硬件质量实验室的要求首先进行自测，后送交官方论证

·定制的安装程序（这部分无须Windows硬件质量实验室认证）

·准备刻录光盘，出货

1. [1] 这里的Windows指的应是我们常说的Windows 9x系列产品 [↑](#footnote-ref-1)
2. [2] 基本输入/输出系统 [↑](#footnote-ref-2)
3. [3] application programming interface [↑](#footnote-ref-3)
4. [4] 原生API——native API [↑](#footnote-ref-4)
5. [5] Driver Development Kit [↑](#footnote-ref-5)
6. [6] Windows XP和Windows 98 [↑](#footnote-ref-6)
7. [7] class driver [↑](#footnote-ref-7)
8. [8] Data Universal Numbering System [↑](#footnote-ref-8)