4.5 USB 驱动程序开发工具

对于开发 WDM 驱动程序来说，我们有以下三个常用组合：

（1）直接使用 Windows DDK

（2）使用 DriverStudio

（3）使用 Windriver

下面我们分别比较三种方式的优缺点。

第一种：开发难度大一些，而且有很多烦琐的工作要作，大部分都是通用的基础性的工作。但如果选用这种方式的话你将对整个体系结构会有很好的理解和把握。

第二种：难度低一些，工具软件已经帮你作了很多基础性的工作。也封装了一些细节，你只要专心去作你需要的操作，但由于封装的问题，可能会带来一些 bug。有可能导致项目的失败。

第三种：几乎没有难度(从开发驱动的角度)。很容易，但只能开发硬件相关的驱动，事实上你写的只是定制和调用它提供的通用驱动而已。效率上有问题。工作频率不是很高。但开发花费的时间很少。是上面的几分之一乃至几十分之一。

一般建议用 windriver 开发驱动程序的原型，用 driverstudio 开发最终发行的驱动程序，如果驱动程序很复杂的话，则直接使用 ddk 开发。

上面的几种情况都需要 VC++作为辅助开发环境。前两种情况都需要 ddk。开发时间上，第一种最长，第三种最短，第二种可以认为是前面两种方案的折衷。

4.6 USB 驱动程序设计步骤

4.6.1 USB 驱动程序的编写

在 WDM 驱动程序设计中我们首先需要写一个 DriverEntry 过程，这是每一个设备驱动程序的入口，每次该程序启动时被系统自动调用。在 DriverEntry 中完成驱动程序的初始化工作。在一个 WDM 驱动程序中，初始化是唯一必不可少的。一个驱动程序可以只有初始化段，虽然这样的驱动程序什么也不能做，但它的的确确是一个驱动程序。若要使一个驱动程序能够实现对硬件设备的驱动，还应该有一些其他的回调例程和分发例程来处理各种 IRP，主要有：

DriverUnload

AddDevice

StartIo

MajorFunction[IRP_MJ_PNP]

MajorFunction[IRP_MJ_CREATE]

MajorFunction[IRP_MJ_CLOSE]

MajorFunction[IRP_MJ_READ]

MajorFunction[IRP_MJ_WRITE]

MajorFunction[IRP_MJ_DEVICE_CONTROL]

MajorFunction[IRP_MJ_POWER]

MajorFunction[IRP_MJ_SYSTEM_CONTROL]

……

而在初始化过程中，所要做的工作就是设置各回调例程的入口指针，使这些回调例程能够响应相应的 IRP。这些回调例程中包括以下几种比较常用的例程：

DriverUnload

系统在卸载设备时调用 DriverUnload，DriverUnload 例程负责在驱动程序被停止前做一些必要的处理。比如释放资源，记录最终状态等。

AddDevice

系统在发现新的硬件设备时调用 AddDevice，在 AddDevice 中主要完成以下方面的工作：

创建设备对象；注册一个或多个设备接口，以便应用程序能够发现设备的存在；把新设备对象放到设备栈上。

StartIo

驱动程序的分发例程必须是可重入的，通常采用的方法是使用 I/O 管理器的服务创建一个 IRP 设备队列，分发例程把 IRP 放在设备队列中，由 I/O 管理器调用 StartIo 一次处理一个 IRP。在 StartIo 中一般是处理具体的输入输出请求。当 StartIo 例程完成一个 IRP 时，它应调用内核，保证对下一个可用的 IRP 可再次调用。

MajorFunction[IRP_MJ_PNP]

当发生设备到达、硬件配置文件改变、设备被删除等情况时，PNP 管理器发出 PNP IRP，调用 MajorFunction[IRP_MJ_PNP]例程，MajorFunction[IRP_MJ_PNP]

例程对这些 PNP IRP 进行处理。需要指出的是，对于驱动程序分配的资源，如 I/O 端口、存储器地址、中断和 DMA 端口等，WDM 驱动程序是在收到“启动设备”PNP IRP 时被告知这些设备资源的。

MajorFunction[IRP_MJ_CREATE]和 MajorFunction[IRP_MJ_CLOSE]

这两个例程在用户调用 CreateFile 和 CloseHandle 时被调用，为即将到来的读写操作做准备，或做一些读写完成后的必要处理。

MajorFunction[IRP_MJ_READ]和 MajorFunction[IRP_MJ_WRITE]

当用户调用 ReadFile 从设备读取数据或 WriteFile 向设备写数据时，系统发出[IRP_MJ_READ]或[IRP_MJ_WRITE] IRP 调用这两个例程之一，在这两个例程中，或者将 IRP 挂接在相应的 IRP 队列上供 StartIo 处理，或者将这些 IRP 变成对硬件的实际的输入输出直接访问 I/O 端口、存储器地址、启动中断、DMA 等操作。

MajorFunction[IRP_MJ_DEVICE_CONTROL]

MajorFunction[IRP_MJ_DEVICE_CONTROL]对设备进行一些自定义的操作，比如更改设置等。它是在用户调用 DeviceIoControl 时被调用。它通过 IRP 获得用户的请求号，以及一个指向用户缓冲区的指针与用户程序进行通信，在这个例程中完成一些特定的 I/O 操作，如设备的设置等。

MajorFunction[IRP_MJ_POWER]

电源管理 IRP，可以对设备进行电源管理。如果不需要对设备进行电源管理，只需要把“电源管理”IRP 简单地传递给设备栈中下一层驱动程序即可。

MajorFunction[IRP_MJ_SYSTEM_CONTROL]

驱动程序通过处理“系统控制”IRP 来支持 WMI（Windows 管理诊断扩展）生成系统诊断和性能信息。与电源管理一样，可以简单地把这个 IRP 沿设备栈向下传递。

实际上还有许多其它的 IRP。不过，它们大多数都不需要进行特别的处理，只需象电源管理一样，把这个 IRP 沿设备栈向下传递就可以了。此外还有几个回调例程需要简单介绍一下：

ISR

中断服务例程，当与设备连接的中断产生时，调用此例程。

DpcForIsr

由于中断处理过程运行于较高的优先级上，它们能屏蔽许多级别小于或等于它们的过程的执行，如果它们占用 CPU 时间过长，很容易使系统性能下降，因此中断服务例程应尽可能快地执行完。然而有的中断服务例程需要完成很多任务，为不影响系统性能，除最紧迫的任务外，其他的部分放在一个被称为延迟过程调用（DPC）的例程中来完成。

至此，驱动程序设计的主要内容也就完成了。

USB 设备驱动程序的特殊之处是 USB 总线不允许 USB 功能驱动程序直接对硬件的访问。USB 设备驱动程序，必须使用 USB 总线驱动程序访问 USB 设备，而不能直接访问硬件。USB 设备驱动程序通常又称为 USB 客户驱动程序。对于 USB 客户驱动程序来说，总线枚举和信息如何传输这些细节由 USB 总线驱动程序来完成，而客户驱动程序并不需要知道。

客户驱动程序通过一个或多个管道访问功能设备，管道是主机之间单向或双向的数据传输通道，管道有 4 种类型：控制管道把命令传输给设备（包括 USB 设置和配置信息），含有沿任一方向的数据传输；中断管道传输设备特定的信息给主机；块管道传输更大量的数据；等时管道传输时间敏感的数据。

USB 客户驱动程序是支持即插即用功能的标准 WDM 驱动程序。当一个 USB 设备插入时，USB 总线驱动程序检测到设备的插入，PnP 管理器使用厂商 ID 或设备类信息选择要运行的驱动程序。首先调用驱动程序的 AddDevice 例程，并发出其他的 PnP IRP。这里需要特别说明的是 USB 客户驱动程序绝不会收到任何硬件资源，因为由 USB 总线驱动程序来处理所有的低层 I/O。

此外，USB 设备特别容易发生意外删除，如用户不小心（或者有意）拔掉了 USB 插头。在 Windows 2000 中，将收到一个“意外删除”PnP IRP，而在 Windows 98 中，只有“删除设备”PnP IRP。即使有对该设备打开的句柄，这些请求也必须成功。这时在处理过程中的任何传输必须中止。

USB 客户驱动程序在它的下沿进行对 USB 总线驱动程序的调用，但是它可以实现需要的上沿功能，具体的依照 USB 设备的用途而定。

USB 客户驱动程序对 USB 总线驱动程序的调用是通过使用表 4-2 所示的 USB 驱动程序接口（USBDI）内部 IOCTL 实现的。这些 IOCTL 都是内部的，只能用于内核的其他部分（如设备驱动程序），而不能用于用户态应用程序。这些

IOCTL 中最重要的内部 IOCTL 是 IOCTL_INTERNAL_USB_SUBMIT_URB，它发出 USB 请求块（URB）由 USB 总线类驱动程序处理。共有 30 多个不同的 URB 功能代码，USB 客户驱动程序使用 URB 做它们大多数的工作。USB 设备驱动程序的作用实际上就是进行各种 USBDI 的调用。

表 4-2　USBDI 的内部 IOCTL

IOCTL_INTERNAL_USB_SUBMIT_URB	发出 URB
IOCTL_INTERNAL_USB_RESET_PORT	复位并重新启用一个端口
IOCTL_INTERNAL_USB_GET_PROT_STATUS	得到端口状态
IOCTL_INTERNAL_USB_ENABLE_PORT	重新启用一个被禁止的端口
IOCTL_INTERNAL_USB_GET_HUB_COUNT	由集线器驱动程序在内部使用
IOCTL_INTERNAL_USB_CYCLE_PORT	模拟设备拔出和再次插入
IOCTL_INTERNAL_USB_GET_ROOTHUB_PDO	由集线器驱动程序在内部使用
IOCTL_INTERNAL_USB_GET_HUB_NAME	得到 USB 集线器的设备名
IOCTL_INTERNAL_USB_GET_BUS_INFO	填写 USB_BUS_NOTIFICATION 结构
IOCTL_INTERNAL_USB_GET_CONTROLLER_NAME	得到主机控制器设备名

USBDI 的调用采用如下方式：首先必须为内部的 IOCTL 创建一个新的 IRP，填写该 IRP，并沿设备栈向下把这个 IRP 发送给 USB 总线驱动程序。然后等待它，直到该 IRP 被处理。

创建一个新的 IRP，通常使用 IoBuildDeviceIoCountrolRequest 来构造一个新的内部 IRP，IOCTL 控制代码必须使用缓冲 I/O 方式。USB 内部 IOCTL 不使用标准输入和输出缓冲区，而必须设置下一个栈单元 Parameters.Others.Argument1 域，这个 Parameters.Others.Argument1 域指向一个 URB。因此，在这之前，还应该构造好一个 URB。设置完成后，调用 IoCallDriver 将 IRP 传递给下一层的总线驱动程序。

URB 作为 USB 内部 IOCTL 的参数传递给下一级的 USB 总线驱动程序，USB 客户驱动程序使用 URB 完成它的大部分工作，URB 在 USB 客户驱动程序有着重要的作用。为更容易地构造 URB，DDK 提供了很多例程来构造一个 URB，如

利用 UsbBuildGetDescriptorRequest 可以构造一个获得设备描述符 URB，UsbBuildInterruptOrBulkTransferRequest 用来构造块传输或中断传输 URB 等等。

总之，USB 客户驱动程序设计是非常复杂的。通常的作法是先构造 URB，然后创建 IRP，然后将 URB 指针填入下一个栈单元的 Parameters.Others.Argument1 域中，最后调用下一层的总线驱动程序。按照 USB 总线的规范，一次次地重复以上过程，即可实现 USB 设备的驱动。

最后，需要说的一点是：利用 DDK 设计驱动程序一般来说是很复杂的。在 DDK 的 SRC 目录下有大量的驱动程序模板，这些模板为我们提供了一个驱动程序框架，利用这些模板进行驱动程序的开发可以大大减少我们驱动程序设计的工作量。

注意事项：

在微软的 White papers 里称，所有在 Windows 95 之后的 Windows 操作系统都已实现了 WDM，包括 Windows 98、Windows 2000、Windows XP、Windows Me，并且在后续的操作系统中，也将继续支持 WDM。虽然，这是 WDM 主要的设计目标之一，但现在看来，微软在不断完善 WDM 的同时，也使这个问题变得有些混乱了，在微软的 White papers 和各操作系统的 DDK 里，得到的信息总是有些含糊。一开始，Microsoft 宣称 WDM 驱动程序会是 Windows98 和 Windows2000 x86 之间二进制兼容的，且与 Windows2000　Alpha 平台源代码兼容。但是，现在看来并不能保证二进制兼容，尽管 DDK 在这方面并不清楚。为了安全起见，建议在为 Windows98 构造驱动程序时，使用 Windows98 驱动程序开发工具包(DDK)，而对 Windows2000 使用 Windows2000 DDK.如果使用仅在 Windows2000 中出现的一些 WDM 功能，则就不能达到源代码兼容。例如，Windows2000 USB 系统驱动程序支持一些 Windows98 驱动程序不可用的功能。编写跨平台驱动程序时，微软对开发者的建议是，在不同的目标平台下，进行严格的测试。

现在我们有好几种 Windows 平台下的驱动程序开发技术了，虽然 WDM 提供了一个全新的解决方案，但为特定的设备和特定的平台选择一种合适的驱动程序类型来开发，仍然非常重要。如果你只需要写一个 Windows 95 的驱动程序，用 VxD。如果你只需要一个 Windows NT 4.0 的驱动程序，用 NT 内核模式驱动程序。如果系统里有一个 WDM 类驱动程序支持你的设备，而你所写的驱动程序不

需要与 Windows 95 和 Windows NT 4.0 兼容，用 WDM。如果你的主要目标是 Windows 98，那么你必须决定是否要与其他的操作系统兼容如果你关心与 Windows 95 兼容，那么你必须写一个 VxD。如果你关心与 Windows NT 兼容，那么写一个 WDM。注意，WDM 驱动程序不会在 Windows 95 和 Windows NT 4.0 下工作，但是，WDM 驱动程序与 NT 下的驱动程序有许多的相似之处，可以很容易的将 WDM 驱动程序改写成 NT 的内核模式驱动程序。如果你的主要目标是 Windows2000，那么你必须决定是否要与其他的操作系统兼容如果你需要与 Windows NT 兼容，那么你必须写一个 NT 内核模式驱动程序。如果你关心与 Windows98 兼容，那么写一个 WDM。注意，WDM 驱动程序不会在 Windows NT 4.0 下工作，但是，WDM 驱动程序与 NT 下的驱动程序有许多的相似之处，可以很容易的将 WDM 驱动程序改写成 NT 的内核模式驱动程序。Windows 98 虽然支持 WDM 和 VxD，但有些类型的设备，在 Windows 98 下必须用 VxD 来设计（比如文件系统驱动程序），其他的某些类型必须用 WDM 设计（比如 USB 驱动程序）。其他情况下，开发者可以根据需要选择写一个 VxD 或 WDM。Windows 2000 支持 VxD、内核模式驱动程序、WDM，但也有些限制，比如视频驱动程序需要用 NT 格式的内核模式驱动程序来编写。具体的细节必须查阅不同版本的 DDK。

4.6.2 USB 驱动程序的编译链接

安装 DDK 后，在 DDK 程序组下有 Check 和 Free 两个编译环境，Check 环境用于编译带调试信息的驱动程序，Free 则是编译正式发布版本的环境。通常情况下设备驱动程序的编译采用命令行的方式。通过一定的设置可以在 VC ++的集成环境下编译。

一般来说，成功编译一个最基本的设备驱动程序需要四个文件，第一个是驱动程序，即 C 语言源程序文件(例如 vdisk.c，注意下面所有的例子都是以 vdisk 来说明)；第二个是 RC 文件(例如 vdisk.rc)；第三个是 sources 文件；第四个文件是 makefile.rc 文件。sources 文件和 make 文件类似，用来指定需要编译的文件以及需要连接的库文件。这三个辅助文件都很简单，在 DDK samples 的每个例程里都有三个这样的文件，依样画瓢就能理解它们的结构和意义。举例分析 以下以 vdisk 程序为例，设 vdisk.rc 代码为：

```
/??vdisk.rc??/
```