第五卷 网络驱动程序设计指南

第一部分 网络驱动程序

网络驱动程序设计指南的向导
内核模式驱动程序的网络结构
网络驱动程序编程要点

第一章 网络驱动程序设计指南的向导

这一章为网络驱动程序设计指南提供了一个导航，它将以你将编写的内核模式网络驱动程序的类型为基础，告诉你需要参见这个指南的哪些部分。

微软的 Windows 2000 支持三种基本的内核模式网络驱动程序：

• 微端口 NIC 驱动程序

一个微端口的驱动程序直接控制一个网络接口卡(NIC)，并且为高层的驱动程序提供接口。

• 中间层驱动程序

一个中层协议驱动程序连接了上层协议，例如早期的传输驱动程序和一个微端口。开发中层协议驱动程序的一个普遍原因是用它在早期的传输驱动程序和一个微端口之间实现转换。一个微端口控制了一个 NIC，对于传输驱动程序来说，它是一个陌生的新介质类型。

• 协议驱动程序

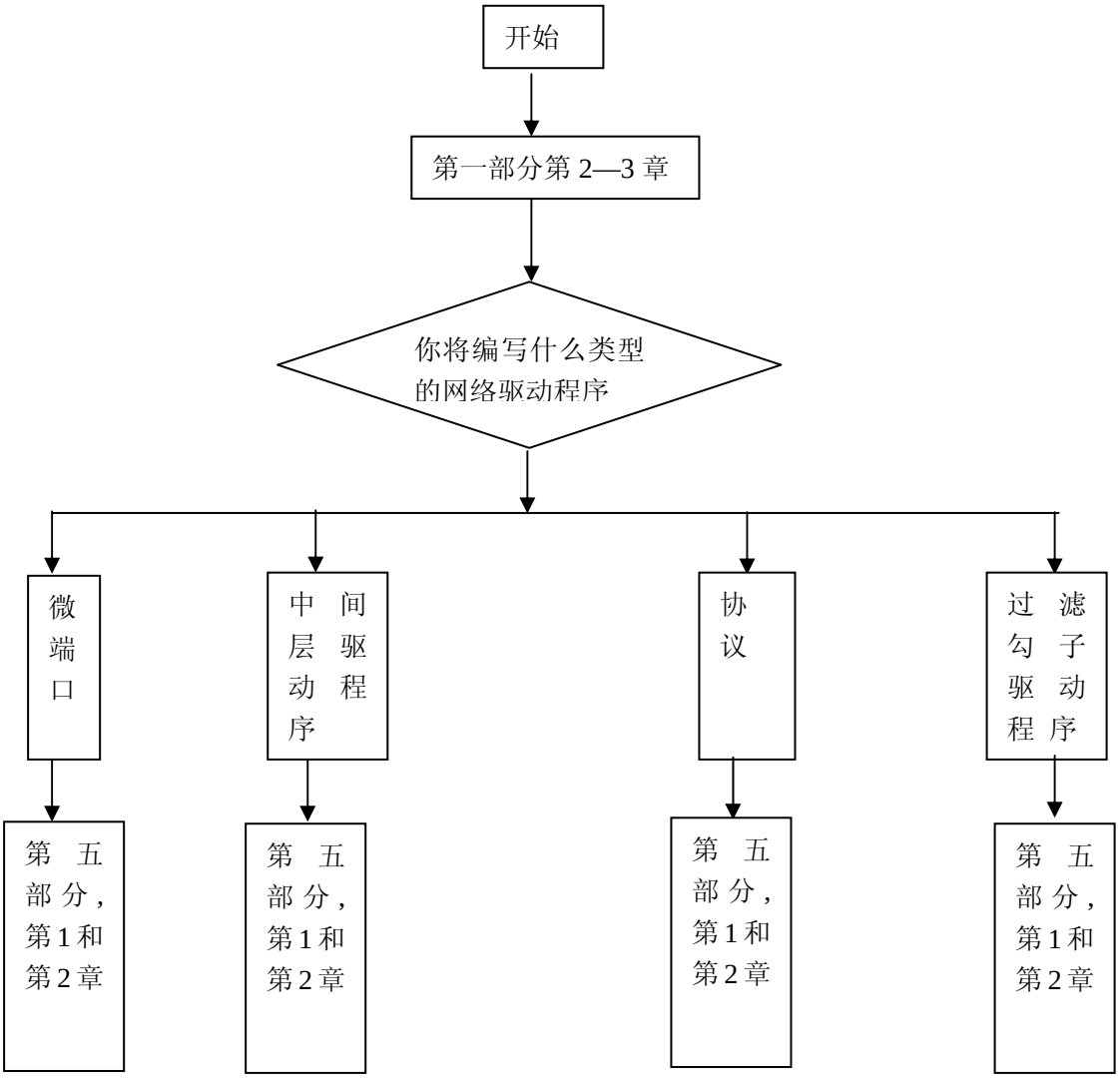
一个上层协议驱动程序向网络用户提供服务，它实现了 TDI 接口，或者也许是为另一个上一层的特殊应用而提供的接口。这种驱动程序在它的下边界提供了一个协议接口，用来向低层驱动程序发送和接收数据包。

另一种协议驱动程序是一种面向连接的呼叫管理器，一个呼叫管理器为面向连接的客户提供了呼叫建立和呼叫撤消服务，呼叫管理器也是协议驱动程序。

Windows 2000 支持的另外一种内核模式驱动程序类型是一过滤勾子驱动程序。一个过滤勾子驱动程序用来过滤数据包，它扩展了操作系统提供的 IP 过滤驱动程序的功能。

无论你将编写什么类型的驱动程序，你都应该参见第一部分“网络驱动程序设计指南”的 2—3 章。这些章节讨论了 Windows 2000 的网络结构和编程要素，你也应该参见第五部分的第 1 章。这一章讨论了网络 INF 文件，它被用来安装网络组件，如果你的网络驱动程序需要一个发布对象——例如控制绑定——请参见第五部分的第 2 章。

为了查寻需要参见哪些额外的章节，在在线文档的下表中键入恰当的驱动程序类型，它将跳转到某一部分，来限制下一步选择驱动程序类型的范围。



微端口驱动程序

选择你将编写的微端口驱动程序类型：

无连接的微端口

如果你要为无连接网络媒体，例如 Ethernet，FDDI，或 Token Ring 编写控制一个 NIC 的微端口，请参见第二部分的第 1—7 和第 9 章。

WAN 微端口

如果你要编写一个微端口来控制一个 WAN NIC 的话，请参见第二部分的第 1—9 章。

面向连接的微端口

如果你要为一个面向连接网络媒体，例如 ATM 或 ISDN，编写控制一个 NIC 的微端口，，请参见第二部分的第 1—7 和第 9 章及第四部分的第 1 章。

集成微端口呼叫管理器(MCM)

如果你要编写一个微端口来控制一个面向连接的 NIC 并且也提供呼叫管理服务的话，请参见第二部分的第 1—7 和第 9 章，第三部分的第 2 章和第四部分的第 1 章。

中间层驱动程序

选择你将编写的中间层驱动程序类型：

面向无连接的下层

如果你要编写一个中间层驱动程序，并在它的下层提供一个对于无连接微端口的接口，

请参见第三部分第 1 章。

面向连接的下层

如果你要编写一个中间层驱动程序，并在它的下层为面向连接微端口提供一个接口，请参见第三部分第 1 章和第四部分第 4 章。

协议驱动程序

选择你将编写的协议驱动程序类型：

面向一个无连接的下层

如果你要编写的协议驱动程序为下层的无连接微端口提供接口，请参见第三部分第 2 章。

面向 TDI 的上一层

如果你要编写一个拥有一个 TDI 的上层的协议，请参见第三部分的第 2，3，4 和第 5 章。

支持 Winsock

如果你要编写一个提供 Winsock 支持的协议，请参见第三部分的 2，3，4，5，6 章。

面向连接客户或呼叫管理器

如果你要编写一个面向连接的客户方，它对面向连接的微端口提供了接口，或者你要编写一个面向连接的呼叫管理器，请参见第三部分的第 2 章和第四部分的第 1 章。

过滤勾子驱动程序

请参见第六部分第 1 章。

第二章 内核模式驱动程序的网络结构

这一章讲述 Windows 2000 的内核模式驱动程序的网络结构，这章包含以下几个部分：

2.1 Windows 2000 网络结构和 OSI 模型
2.2 NDIS 网络驱动程序
2.3 TDI 驱动程序
2.4 网络驱动程序环境

2.1 Windows 2000 网络结构和 OSI 模型

Windows 2000 网络结构是以国际标准化组织(ISO)制定的七层网络模型为基础的，1978 年，ISO 制定的开放式系统(OSI)参考模型，将网络描述为一系列的协议层，在每个协议层中完成一系列的特定功能。每一层都向上一层提供明确的服务，同时将本层服务的实现封装起来.一个在相邻层之间完善的接口定义了下层对上层所提供的服务以及如何访问这些服务。

(应用层,表示层,会话层,传输层,网络层,链路层,物理层,物理介质　　图 2.1 OSI 参考模型)

Windows 2000 的网络驱动程序实现了这个网络结构的下面四层。

物理层

这是 OSI 模型中的最底层，这一层用来通过物理介质接收和发送的原始的没有结构的二进制数据流，它描述了对于物理介质的电/光，机械和功能的的接口，物理层为所有的上层传送信号，在 Windows 2000 中，物理层通过网络接口卡(NIC)来实现，物理层的收发是依缚于 NIC 介质的。使用串行端口的网络组件，物理层也同样包含了底层的网络软件，它定义了串行位流是如何分成数据包的。

数据链路层

电气电子工程师协会(IEEE)将该层进一步分成了两个子层，LLC 和 MAC。LLC 子层用于将数据帧从一个结点无错的传输到另一个结点。LLC 子层用来建立和终止逻辑链接，控制帧流，对帧排序，接收帧，并且对没有被接收的帧进行重发。LLC 子层使用帧应答和帧的重发为经过链路层的上层提供了真正的无错发送。MAC 子层控制物理介质的访问，检查帧的错误，并且管理接收帧的地址认证。在 Windows 2000 网络结构中，逻辑链路控制子层在传输驱动程序中实现，而介质访问控制子层则在网络接口卡(NIC)中实现，NIC 由一个名为 NIC 驱动程序的设备驱动程序软件控制。Windows 2000 的附带了许多常用的 NIC 的驱动程序。

网络层

这一层控制着子网的运作，它基于以下因素决定数据的物理路径：

- 网络状况
- 服务优先权
- 其他因素，包括路由、流量控制、帧的分解和重组、逻辑到物理地址的映射、用户帐号。

传输层

这一层确保信息传送的无错传输，连续传输和不丢失或不重复。它使得上层协议与上层

协议之间或与它同层的协议之间通讯不必关心数据的传输。传输层所在的协议栈至少应包括一个可靠的网络层，或在逻辑链路控制子层中提供一个虚电路。例如，因为 Windows 2000 NetBEUI 传输驱动程序包括一个与 OSI 兼容的 LLC 子层，它的传输层的功能就很小。如果协议栈不包括 LLC 子层，并且网络层不可靠，并且/或者支持自带地址信息(例如 TCP/IP 的 IP 层或 NWLINK 的 IPX 层)，那么传输层应能进行帧的顺序控制和帧的响应，同时要对未响应帧进行重发。

在 Windows 2000 网络结构中，逻辑链路层，物理层和传输层都是通过名为传输驱动程序的软件实现的，它有时也称作协议，协议驱动程序或协议模块。Windows 2000 附带了 TCP/IP，IPX/SPX，NetBEUI 和 AppleTalk 传输驱动程序。

2.2 NDIS 驱动程序

网络驱动程序接口说明(NDIS)库将网络硬件抽象为网络驱动程序。NDIS 也说明了网络驱动程序间的标准接口，因此它将用来管理硬件的底层驱动程序抽象为上层驱动程序，例如网络传输层。NDIS 也维护着状态信息和网络驱动程序的参数，包括指向函数的指针，句柄和链接时参数块的指针，以及其他系统参数。

正如图 2.2 所示，NDIS 支持以下几种类型的网络驱动程序：

- 微端口驱动程序
- 中间层驱动程序
- 协议驱动程序

(图 2.2 NDIS 驱动程序)

2.2.1 NDIS 微端口驱动程序

一个 NDIS 微端口驱动程序(也叫微端口 NIC 驱动程序)有两个基本功能：

- 管理一个网络接口卡(NIC)，包括通过 NIC 发送和接收数据。
- 与高层驱动程序相接，例如中间层驱动程序和传输协议驱动程序。

一个微端口驱动程序与它的 NIC 通信，并且通过 NDIS 库与高层驱动程序通信。NDIS 库对外提供了一整套的函数(NdisXXX 函数)，这些函数封装了微端口需要调用的所有操作系统函数。同时，微端口必须向外提供一组入口(MiniportXxx 函数)，使 NDIS 可以为了完成自己或高层驱动程序的任务而访问微端口。

发送和接收操作表明了 NDIS 与高层驱动程序和微端口 NIC 的相互作用。

- 当一个传输驱动程序发送一个包时，它调用一个 NDIS 库所提供的 NdisXxx 函数，NDIS 于是通过调用由微端口提供的合适的 MiniportXxx 函数将包传递给微端口，然后微端口驱动程序通过调用恰当的 NdisXxx 函数将包传递给 NIC 来发送包。
- 当一个 NIC 接收到由 NIC 发给它的包时，它将产生一个由 NDIS 或 NIC 的微端口处理的中断。NDIS 通过调用恰当的 MiniportXxx 函数指示 NIC 的微端口。微端口通过调用恰当的 NdisXxx 函数把数据从 NIC 传送到上层驱动程序，并且同时指示上层驱动程序接收包。

NDIS 既支持无连接环境下的微端口驱动程序，也支持面向连接的微端口驱动程序。

无连接的微端口为无连接网络介质，例如 Ethernt，FDDI 和 Token Ring，控制 NIC。可将无连接微端口进一步分为以下几种子类型：

- 串行化的驱动程序,它依靠 NDIS 对 MiniportXxx 函数调用进行串行化，并管理它们的发送队列。

- 非串行化驱动程序，它自己对 MiniportXxx 函数操作进行串行化，并且在内部对进入

的发送包进行排队。它的意义在于有很高的效率，例如驱动程序的临界区(在一个时间段内只有一个线程可访问的代码区)。面向连接的微端口为面向连接的网络介质，例如 ATM 和 ISDN，控制 NIC。面向连接的微端口经常非串行化的——它自己对 MiniportXxx 函数操作进行串行化，并且在内部对进入的发送抱进行排队。

一个 NDIS 微端口驱动程序可以有一个非 NDIS 的底层(参见图 2.3)。

（图 2.3 拥有非 NDIS 的底层 NDIS 微端口）

通过它的非 NDIS 底层，微端口使用某种类型的接口，例如通用串行总线构架(USB)或 IEEEE1394(火线)来控制总线上的设备。微端口通过直接发送 I/O 请求包(IRPs)到总线或是发送到连接到总线的其他远程设备上来与设备通信。在它的上层，微端口提供了标准的 NDIS 微端口接口，它使得微端口可以与上层 NDIS 驱动程序通信。

NDIS 也支持广域网(WAN)范围的控制 WAN NIC 的微端口，有关更多的 WAN 网络环境的信息，可参见 2.4.3 节。

2.2.2 NDIS 中间层驱动程序

正如图 2.4 所示，中间层驱动程序一般位于微端口驱动程序和传输协议驱动程序之间。

因为它在驱动程序层结构的中间位置，所以既与上层协议驱动程序通信又要与下层微端口驱动程序通信。

- 在它的下界，中间层驱动程序提供了协议入口点(ProtocolXxx 函数)，NDIS 调用这些函数传递下层微端口的请求。对于一个下层微端口驱动程序，一个中间层驱动程序这时就仿佛是一个协议驱动程序。
- 在它的上界，中间层驱动程序提供了微端口的入口指针(MiniportXxx 函数)，一个或多个上层协议驱动程序通过 NDIS 调用这些函数进行通信。对于上层协议驱动程序，一个中间层驱动程序这时就仿佛是一个微端口驱动程序。

（图 2.4 在微端口驱动程序和传输驱动程序之间的中间层驱动程序）

虽然向上层提供了 MiniportXxx 函数的一个子集，但是事实上中间层驱动程序并不管理物理的 NIC。它只是向上层协议提供了一个或多个可以绑定的虚拟适配器。对于一个协议驱动程序，一个由中间层驱动程序提供的适配器就如同一个物理的 NIC。当协议驱动程序发送包或向一个实际的适配器时发出请求时，中间层驱动程序将传递这些包和请求到下层的微端口。当下层微端口向上发出接收包的指示、响应协议请求的信息或指示状态信息时，中间层驱动程序将这些包、响应、和状态指示传递给绑定在虚拟适配器上的协议驱动程序。

中间层驱动程序有如下几种典型使用方法：

- 在不同的网络介质间进行转换

例如，处于 Ethernet 和 Token Ring 传输层及一个 ATM 微端口之间的中间层驱动程序的功能是将 Ethenet 和 Token Ring 的包转换为 ATM 包，反之亦然。

- 过滤包

一个包调度器是中间层驱动程序用作过滤驱动程序的很好例子。包调度器读出每个传输层传下来的发送数据包和每一个由微端口指示的接收包的优先权。然后包调度器以它的优先权调度抱的接收和发送顺序。

• 在多个 NIC 间平衡包的负载

负载平衡驱动程序向上层传输协议提供了一个虚拟的适配器，但实际上它却将包分配了给多个 NIC。

2.2.3 NDIS 协议驱动程序

一个网络协议在 NDIS 驱动程序层次结构中属于最高层驱动程序，而它经常在实现传输层协议的传输驱动程序中被用作最底层的驱动程序，例如 TCP/IP 或 IPX/SPX。一个传输协议驱动程序分配包，从应用程序中将数据拷贝到包中，并且通过调用 NDIS 函数将这些包发送到低层驱动程序中。协议驱动程序也为从下层驱动程序中接收包提供了接口。一个传输协议驱动程序将接收到的数据转换成相应的客户应用数据。

在它的下层，协议驱动程序与中层网络驱动程序和微端口 NIC 驱动程序相连接。协议驱动程序调用 NdisXxx 函数来发送包，读取和设置由低层驱动程序所维护的信息，以及使用操作系统服务。协议驱动程序也提供了一套入口点(ProtocolXxx 函数)， NDIS 代表下层驱动程序调用这些函数为自己或向上指示接收包、指示下层驱动程序的状态、以及与协议驱动程序进行通信。

对于上层，传输协议驱动程序对高层驱动程序提供了一个私有接口。

2.3 TDI 驱动程序

传输驱动程序接口(TDI)定义了一个内核模式的网络接口，它是为上层的传输协议栈提供的。(如图 2.5)

（图 2.5 TDI 客户和传输层）

TDI 客户是内核模式的驱动程序，例如 Redirector 和 Server，它通过 TDI 与传输层相接。TDI 简化了 TDI 的开发传输驱动程序的工作。同时它也通过减少必需编写的传输层相关代码的数量来简化开发客户的工作量。

传输驱动程序提供了仅可被 TDI 客户方所使用的 TDI 接口。为了提供更多的接入传输层的方法，Windows 2000 为当前较通用的网络接口，Windows Socket 和 NetBIOS 提供了仿真器模块，每个仿真器模块提供了它自己的一套函数，在用户模型中，可以通过标准的调用机制来调用它们。当调用它们时，仿真器模块将自己的函数和相关参数以及程序控制传递给 TDI 函数，然后通过 IDL 调用相关的传输驱动程序。

为了提高它的性能，TCP/IP 和 IPX/SPX 的实现也作为 TDI 传输驱动程序的一部分。

2.4 网络驱动程序环境

这一节将讲述微软 Windows 2000 基于以下几种类型的内核模式网络驱动程序的网络环境。

- 2.4.1 无连接的驱动程序
- 2.4.2 面向连接的驱动程序
- 2.4.3 WAN 驱动程序

2.4.1 无连接环境的网络驱动程序

图 2.6 显示了无连接网络程序的 NDIS 环境。

（图 2.6 无连接网络驱动程序环境）

无连接环境是为无连接介质例如 Ethernet 和 Token Ring 所提供的标准的网络驱动程序环境。参见 2.2 节中对这种环境下的驱动程序的描述。

2.4.2 面向连接环境下的网络驱动程序

NDIS 支持以下几种面向连接的驱动程序：

- 面向连接的微端口
- 面向连接的客户方
- 呼叫管理器
- 集成的微端口呼叫管理器(MCM)

图 2.7 显示了一个面向连接的客户方，一个呼叫管理器，和一个面向连接的微端口的配置。

（图 2.8 带 MCM 的面向连接环境）

面向连接的微端口控制着一个或多个网络接口卡(NIC)并且在面向连接协议(面向连接的客户方和呼叫管理器)和 NIC 硬件之间提供了一个接口。

呼叫管理器是一个 NDIS 协议，它对面向连接的客户方提供了呼叫的建立和断开服务。呼叫管理器使用面向连接微端口的发送和接收功能来与网络入口，例如网络交换机实体或远程的对等层交换信令消息。一个呼叫管理器支持一个或多个信令协议，例如，ATM 论坛所提供的 ATM UNI3.1。

MCM 驱动程序是一个面向连接的微端口，它也对面向连接的客户方提供了呼叫管理服务。虽然一个 MCM 对客户提供的面向连接的服务与一个呼叫管理器伴随一个面向连接的微端口提供的服务相同，但是呼叫管理器/微端口接口是在驱动程序内部的，因此对于 NDIS 是不透明的。

多个呼叫管理器和/或 MCM 可以共存在同一个环境下，并且每个呼叫管理器或 MCM 可以支持多个信令协议。

一个面向连接的客户方使用呼叫管理器或 MCM 提供的呼叫建立和断开服务。一个面向连接的客户方同样也使用面向连接的微端口或 MCM 的发送和接收功能，来发送和接收数据。

面向连接的客户方与一个无连接协议的共同点在于，对于它的上层应用，它们都提供了自己的网络和传输层服务，但是，不同于无连接协议的是，面向连接的客户方对于它的下层，它使用了呼叫管理器和面向连接微端口服务或它使用了 MCM 服务。

一个面向连接的客户方可以是一个在早期协议和面向连接 NDIS 之间的一个适配层(它可以是一个中间层驱动程序)。这种适配层的例子如 IP/ATM 和 LAN 仿真器(LANE)，它们都对下层连接使用了呼叫管理器服务，但对上层非连接协议都隐藏了面向连接特性的接口。鉴于这样的面向连接客户方的上层接口定义已超出了 NDIS 文档的范围，如果一个客户方正作为一个适配层服务的的话，那么它的上层接口应是由适应面向连接的 NDIS 的上层协议定义的。

关于更多的有关面向连接驱动程序的信息请见第四部分。

2.4.3 WAN 网络驱动程序的环境

Windows 2000 既支持广域网(WAN)通过无连接介质连接，也支持通过面向连接介质的连接。图 2.9 显示了 WAN 环境。

下图显示了远程访问服务（RAS）结构。

（图 2.9 Windows 2000 WAN 环境）

WAN 环境包括以下的组件：

• 远程访问服务(RAS)

RAS 允许用户模式的应用程序使用拨号连接，在一个 RAS 连接建立以后，一个用户的应用程序可以通过标准的网络接口，例如 Windows Sockets、NetBIOS、Named pipe 或 RPC 与网络服务相连。

• TAPI 服务提供者

TAPI 服务提供者是一个用户模式组件，它通过服务提供接口(SPI)发出的呼叫建立和断开请求，来接受来自 RAS 客户方和 TAPI-aware 应用的请求。并将 TAPI 请求传递给 NDIS TAPI（如果呼叫是通过无连接媒体）或 TAPI proxy（如果呼叫是通过面向连接媒体）。TAPI 服务提供者将把 SPI 请求转换为 TAPI 请求。

• **NDISTAPL**

NDISTAPI 是内核模式的组件，它向 TAPI 设备提供无连接的微端口。NDISTAPI 接受来自 TAPI 服务提供者的呼叫建立和断开请求，并且通过 NDISWAN 将请求传递给适当的微端口以建立，监听和断开线路及呼叫。

• **NDPROXY**

TAPI 代理是一个内核模式组件，它向 TAPI 设备提供面向连接的微端口。NOPROXY 对于 NDISWAN 相当于一个呼叫管理器，对于面向连接的微端口相当于一个面向连接的客户方。(更多的关于面向连接的网络环境信息请看 2.4.2 节)

• NDISWAN

NDISWAN 是一个中层 NDIS 驱动程序，它完成 PPP 协议/链接组帧，压缩和加密功能。NDISWAN 将来自上层传输驱动程序的 NDIS_PACKET 包转换成 NDIS_WAN_ PACKET 包并且将转换后的包传到下层的 WAN 微端口驱动程序。NDISWAN 既支持无连接也支持面向连接的微端口。

当工作在一个面向连接的环境下，NDISWAN 对于 TAPI 代理驱动程序相当于一个面向连接的客户方，对一个呼叫管理器提供了 NDISWAN 接口。

• WAN 微端口

一个 WAN 微端口和非 WAN NDIS 微端口调用许多相同的 NDIS 函数并提供许多相同的句柄。但是当一个 WAN 微端口发送包和向上层协议指示收到包时，它调用特殊的与 WAN 相关的 NDIS 函数。WAN 微端口也使用 NDIS_WAN_PACKET 结构而不是 NDIS_PACKET 结构。同时，WAN 微端口维护 WAN 相关信息并且对这种特殊的 WAN 的查询信息给予响应。一个 WAN 微端口支持无连接介质或面向连接的介质。

• 串行驱动程序

串行驱动程序是一个对内部串行口或多端口串行卡的标准化设备驱动程序。为 Windows 2000 内置的异步 WAN 微端口使用内部串行驱动程序来进行调制解调通信。

第三章 网络驱动程序编程要点

这一章将讲述当编写 Windows 2000 的任何网络驱动程序时通常要考虑的几点问题。

为 windows 2000 编写的网络驱动程序应完成许多共同的目标：

- 跨平台的可移植性
- 对于多处理器系统的可变性
- 简化软件硬件设置
- 基于对象的接口
- 支持异步 I/O

以下是几节为 Windows 2000 网络驱动程序编写者讲述设计目标的实现：

3.1 可移植性

3.2 多处理器支持

3.3 IRQLs

3.4 同步和指示

3.5 包结构

3.6 使用共享内存

3.7 异步 I/O 和完成函数

3.1 可移植性

NDIS 驱动程序应很容易在支持 Windows 2000 的平台间移植。一般说来，从一个硬件平台移植到另一个平台只需要将它在兼容系统的编译中重新编译即何。

驱动程序开发者应当避免调用与操作系统相关的函数，因为这将使他们的驱动程序不可移植。应用 NDIS 函数替换这些调用，只调用 NDIS 函数将使代码在支持 NDIS 的微软操作系统中可移植，NDIS 为编写驱动程序提供了很多支持函数，并且直接调用操作系统是不需要的。

驱动程序应用 C 来编写。更近一步，驱动程序代码应当限制在 ANSI C 标准并且应当避免使用不被其他兼容系统编译器支持的语言特性。驱动程序代码编写不应当使用任何 ANSI C 标准指明为“implementation defined”的特性来编写。

驱动程序应当避免使用任何在不同平台上大小和结构变化的数据类型，驱动程序代码不应该调用任何 C 的运行时库函数，而是限于调用 NDIS 提供的函数。

在内核模式下漂移指针指针是不允许的。试图运行这样的操作将是一个致命的错误。

如果驱动程序代码要支持特殊平台的特性，那么这些代码应当包含在#ifdef 和#endif 声明之间。

3.2 多处理器支持

编写的代码应当可以在多处理器系统中安全运行。这对于编写可移植的 Windows 2000 驱动程序是很重要的。一个网络驱动程序必须使用 NDIS 函数库提供的多处理器安全保证。

在单处理器环境下，在一个时刻单处理器只运行一条机器指令，既使这样，当包到达或时间中断发生时，对于 NIC 或其他设备执行的中断也可能发生。典型的，当正在操纵数据结构时，例如它的队列时，驱动程序对 NIC 发出停用中断来操纵数据，随后再发生可用中

断。许多线程在单处理器环境下表现的好像是在同一时刻运行的，但是实际上它们却是在独立的时间片上运行的。

在多处理器环境下，处理器同时可运行多条机器指令，一个驱动程序必须异步化，这使得当一个驱动程序函数正在操纵一个数据结构时，同样的在其他处理器运行的驱动程序函数不能修改共享的数据。在一个 SMP 机器中，所有的驱动程序代码都要重新装入，为了消除这种资源保护问题，Windows 2000 驱动程序使用自旋锁。对于 NDIS 微端口，NDIS 函数库可处理许多这些多处理器问题。NDIS 库对请求进行排队和串行化调用标准的微端口函数，除非有以下情况之一：

• 在设计和实现中，NIC 为无连接网络介质设计，并且微端口设计为非串行化的，这或许是因为 NIC 有支持包排队和同步化的部分，或者因为驱动程序的编写者喜欢来管理排队和在微端口内实现同步问题。

• NIC 为面向连接介质设计，NDIS 认为任何面向连接的 NIC 驱动程序是一个非串行化的微端口。

但是大多数无连接的 NIC 驱动程序都是串行化的微端口，因为他们依赖 NDIS 来管理包排队，同步化和串行化问题。

3.3 IRQL

所有 NDIS 调用的驱动程序函数都运行在系统决定的 IRQL 下，PASSIVE_LEVEL<DLSPATCH_LEVEL<DIRQL 中的一个。例如，一个微端口初始化函数、挂起函数、重启函数和有时的关闭函数通常都运行在 PASSIVE_LEVEL 下。中断代码运行在 DIRQL 下，所以 NDIS 中层或协议驱动程序从不运行在 DIRQL 下。所有其他的 NDIS 驱动程序函数运行在 IRQL<=DISPATCH_LEVEL 下。

驱动程序函数运行于的 IRQL 将影响调用什么样的 NDIS 函数。特定的函数只可在 IRQL PASSIVE_LEVEL 下调用，其他的函数可在 DISPATCH_LEVEL 或更低层调用。一个驱动程序的编写者应当检查每一个 NDIS 函数的 TRQL 限制。

任何与驱动程序的 ISR 共享资源的驱动程序函数必须能将它的 IRQL 升级到 DTRQL 来防止争用情况的发生，NDIS 提供了这种机质。

3.4 同步和指示

当两个线程共享可被同时访问的资源时，无论是单处理机还是 SMP，同步是必须的。例如，对于一个单处理机，如果一个驱动程序正在访问一个共享资源时，被一个运行在更高 IRQL（例如 ISR）的函数中断时，必须保护共享资源以阻止这种争用的发生而使资源处于不确定状态。在一个 SMP 中，两个线程可以在同一时刻运行，在不同处理器上并且试图来修改同一数据的访问必须同步。

NDIS 提供了自旋锁可以用来对在同一 IRQL 下运行的线程间访问共享资源实现同步。当两个线程在不同 IRQL 下访问共享资源时，NDIS 提供了一种机制来临时升高低 IRQL 代码的 IRQL，以使得访问共享资源串行化。

当一个线程依赖于一个外部事件的发生时，指示是必须的。例如，对于驱动程序，当经过一定周期后，必须指示它以使它可以检查它的驱动设备。或者一个 NIC 设备驱动程序要进行周期性的操作例如轮询，时钟提供了这种机质。

事件提供了一种两个执行的线程可进行同步操作的机质。例如，一个微端口驱动程序通过向设备写数据来测试 NIC 上的中断，它必须待等一种中断来指示驱动程序操作是成功的。

事件可以在等待中断完成的线程与处理中断的线程之间进行同步操作。

以下是这些 NDIS 机制的描述。

自旋锁

自旋锁提供了一个用来保护共享资源的同步机制，这种资源是单处理器或一个多处理机下的、运行在 IRQL>PASSIVE_LEVEL 下的、内核模式中的线程所共享使用的。一个自旋锁在同时运行在一个 SMP 机上不同的执行线程之间提供同步。一个线程在访问保护资源前获得一个自旋锁。自旋锁使得任务线程中只有持有自旋锁的线程可使用资源。一个等待自旋锁的线程将在试图获得锁时间内循环，直到持有锁的线程释放为止。

自旋锁的另一个特点是与 IRQL 有关。试图获得自旋锁将请求线程的 IRQL 临时升高为正在执行线程的 IRQL。这阻止了运行在同一处理器上的低 IRQL 线程优先获得执行线程，运行在高 IRQL 的线程可以优先获得执行线程，但这些线程不能获得自旋锁，因为自旋锁比它们的 IRQL 低。因此，其他运行在同一处理器上的线程只有到自旋锁被执行线程获得并释放后才可试图获得自旋锁。一个编写很好的网络驱动程序应该会减少自旋锁持有的时间。

一个典型的使用自旋锁的例子是保护一个队列。例如，微端口发送函数 MiniportSend 将协议驱动程序传来的包进行排队。因为其他驱动程序函数也使用这个队列，MiniportSend 必须用一个自旋锁保护这个队列使得在一个时刻只有一个线程可操纵这个队列。Miniport Send 获得自旋锁，添加包到队列后释放自旋锁。使用自旋锁保证持锁线程是唯一修改队列的线程，同时使得包被安全地添加到队列中。当 NIC 驱动程序从队列中取走包时，通过同样的自旋锁保护这个访问。当执行指令修改队列头或任何队列组成域时，驱动程序必须用自旋锁保护队列。

一个驱动程序不能过于保护一个队列。例如，一个驱动程序可以在网络驱动程序将包进行排队之前在包的预留字段进行操作(例如，填充包含长度的字段)。驱动程序可以在自旋锁保护区域之外进行这种操作，但必须是在进行包排队之前。一旦包排入队列并且执行线程释放了自旋锁，驱动程序应认为其他线程可以立即将包从队列中取出。

避免死锁问题

Windows 2000 并不限制网络驱动程序同时持有多于一个的自旋锁。但是，驱动程序的某部分在持有自旋锁 B 时，试图获得自旋锁 A，并且其他部分在持有锁 A 时，试图获得自旋锁 B 时，死锁就会发生。如果要获得多于一个的自旋锁，驱动程序应当通过强制以某一顺序获得锁来避免死锁，这就是说，如果一个驱动程序强制在获得自旋锁 A 之后才可获得锁 B，那么上述情况就不会发生。

总得来说，使用自旋锁将对系统性能带来负面效应，所以驱动程序不应当使用许多锁。有时，完全不同的函数(例如，发送和接收函数)有不太要紧的重叠可使用两个自旋锁。使用多于一个自旋锁是为了允许两个运行在独立处理器上的函数独立操作的折中方案。

时钟

时钟被用来轮询或进行超时操作的。一个驱动程序可以产生一个时钟并联合一个函数时钟上。当一个特定周期时钟期满时，调用相关函数。时钟可以是一次的或周期性的，一但设置了一个周期时钟，当每个周期结束时都会触发，直到它被完全清除掉为止。一次性时钟在触发后必须重新设置。

时钟通过调用 NdisMInitializeTimer 来产生和初始化，并且通过调用 NdisMsetTimer 来设置，也可调用 NdisMsetPeriodicTimer 设置周期时钟。如果使用了一个非周期时钟，那么通过调用 NdisMSetPeriodicTimer 重新设置时钟。通过调用 NdisMCancelTimer 可以清除时钟。

事件

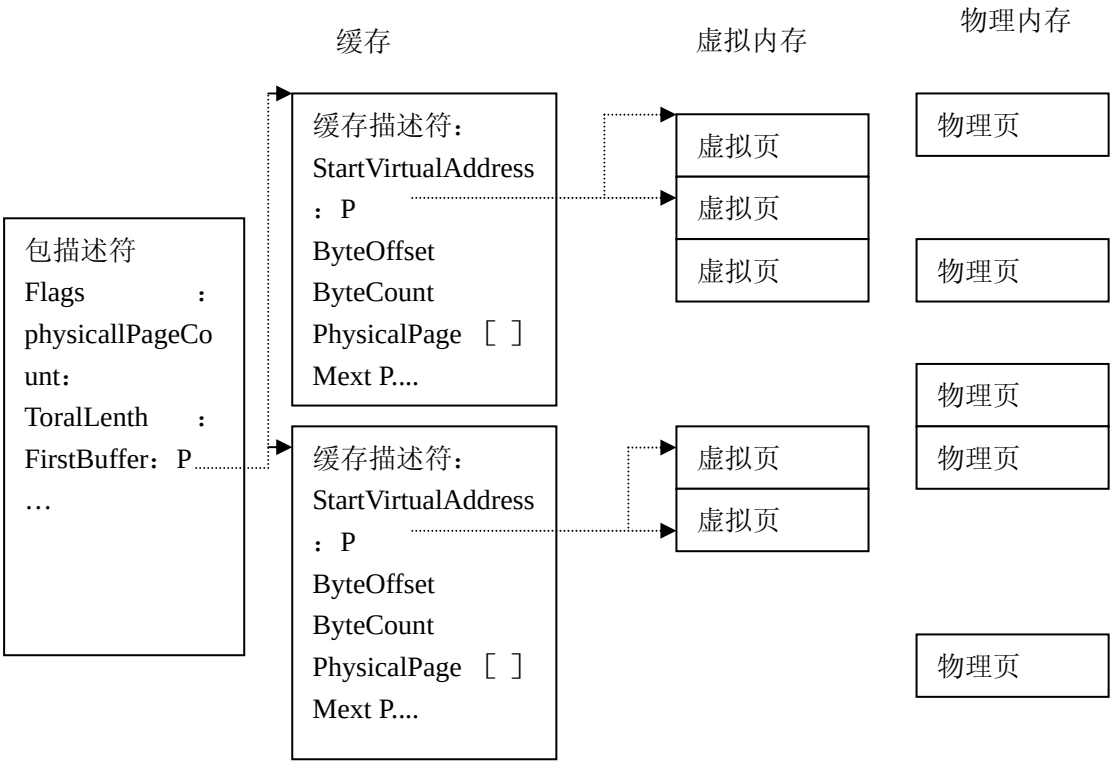
事件在两个执行线程之间实现同步操作。一个事件通过一个驱动程序装入并且通过调用 NdisInitializeEvent 初始化。一个运行在 IRQL PASSIVE_LEVEL 下的线程调用 NdisWaitEvent

来将自身转入等侯状态。当一个驱动程序线程等待一个事件时，它指定了最大等待时间即等待事件的时间。当调用 NdisSetEvent 使时间得到信号量，或最大等待时间段结束时，它们两个无论是谁先发生时都将结束线程等待状态。

典型的，事件是通过相互协调的线程调用 NdisSetEvent 来设置的。事件被创建时是没有信号量的，但为了指示等待线程，它必须要设置信号量，事件将一直处于保持有信号状态，直到 NdiResetEvent 调用后为止。

3.5 包结构

通过一个协议驱动程序可以分配 NDIS 包、填充数据，并且将它传递到下层的 NDIS 驱动程序，以便将数据发送到网络上。一些最底层的 NIC 驱动程序分配包用来保存接收到的数据，并将包传递到对应的高层驱动程序。有时，一个协议驱动程序分配一个包，并且通过一个请求将它传给 NIC 驱动程序，以使 NIC 驱动程序将接收到的数据拷贝到提供的包中。NDIS 提供函数用来分配和操纵构成包的子结构。



一个包的组成如下：

• 一个包描述符包含一个为微端口 NIC 和协议驱动程序提供的私有区域，一系列与包有关的、由相互合作的微端口协议驱动程序定义含义的标志量，包含包的物理页的长度，包的总长，和一个指向第一个缓存描述符的指针，而这个描述符映射了包中的第一个缓存。

• 一组缓存描述符，一个缓存描述符描述了每个缓存虚地址的起始位置，缓存区在页中

的偏移量（字节，通过虚地址给出），缓存中字节的总数和指向下一个缓存描述符的指针（如果有的话）。

• 虚拟范围，可以多于一页，它组成了由缓存描述符所描述的缓存。这些虚拟页映射到物理内存。

一个总线管理 NIC 驱动程序为接收包分配共享内存，或协议驱动程序为发送包分配内存时，必须保证用来存放接收或发出包的缓存与 cache 对齐，对于微端口在发送包以前刷新缓存确保一致性和在指示上层数据到来前刷新接收缓存，这很重要。

3.6 使用共享内存

用作总线管理 DMA 设备的微端口驱动程序必须为 NIC 和 NIC 驱动程序分配共享内存。当在一个驱动程序和它的 NIC 之间共享 cache 时，特别的预防是必须的。在某种结构下，必须采取特别步骤来保证内存一致，因为 NIC 可以直接访问共享的物理内存，而 NIC 驱动程序却要通过 cache 访问内存。这就引起驱动程序和 NIC 访问内存的不同，即使它们看起来在同一位置。

使用无 cache 共享可以避免许多与 cache 共享相关的问题。但是不被 cache 的内存是一个很稀少的系统资源，并且对这种内存的分配也有很多限制。当必须用写一个小的数据到物理内存时，没有 cache 的数据将不会很快地写到物理内存中，并且不使用 cache 不是推荐的也是不可能的。例如，NIC 驱动程序反复读数据应当使用 cache 来提高效率，或者，另一个例子，当要求 NIC 驱动程序将接收到的数据包传给许多不同的协议驱动程序时，极力推荐在这些情况下使用 cache。NIC 驱动程序也应当使用 cache，来通过协议驱动程序传递任何数据包。

NdisMAllocateSharedMemory 可由总线管理 NIC 驱动程序调用来为网络适配器和 NIC 驱动程序分配永久共享内存。这个函数返回了共享内存的虚地址和物理地址，这些地址是有效的，直到调用了 NdisMFreeSharedMemory 来释放内存为止。

当使用共享 cache 时，NDIS 提供了一个必须由 NIC 驱动程序调用的函数来确保 NIC 所见到的和 NIC 驱动程序所见到的是一致的。在为了发送或接收访问共享内存时，NIC 驱动程序必须调用 NdisFlushBuffer 和 NdisMUPdateSharedMemory 来确保 cache 一致性。

3.7 异步 I/O 和完成函数

因为在一些网络操作中有继承的因素，许多由 NIC 驱动程序提供的上层函数和协议驱动程序提供的下层函数被设计成支持异步操作，而不是用 CPU 消耗一定时间的循环来等待一个任务的完成或硬件事件的指示，网络驱动程序依赖处理许多异步操作的能力。

通过使用完成函数来支持异步网络 I/O。以下的例子将说明网络的 send 操作如何使用一个完成函数，同样的机制也存在一个协议或 NIC 驱动程序的其他操作中。

当协议驱动程序调用 NDIS 发送一个包时，NDIS 调用 NIC 驱动程序的 MiniportSend 函数发送请求，NIC 驱动程序试图立即完成这个请求并且返回一个恰当的状态值。对于异步操作，可能返回 NDIS_STATUS_SUCCESS 作为发送成功的标志，NDIS_STATUS_RESOURCES 和 NDIS_STATUS_FAILURE 表明有某些失败。

但是一个发送操作要花费一些时间来完成，此时 NIC 驱动程序(或 NDIS)可将包排队并且等侯 NIC 指示发送操作的结果。NIC 驱动程序的 MiniportSend 函数可以通过返回一个 NDIS_STATUS_PENDING 的状态值来异步处理这个操作，当 NIC 驱动程序完成了发送操作后，包调用完成函数 NdisMSendComplete 在调用中传递指向一个已被发送的包的描述符的

指针。这个信息会传给协议驱动程序，指示完成了操作。

许多需要一定时间来完成的驱动程序操作用完成数来完成支持异步的操作。这种函数有同一形式的名字 NidisMXxxComplete。不仅可用于发送和接收函数，完成函数也可用于查询、配置、重新设置硬件、状态指示、指示收到数据和传送收到数据。

第二部分 微端口 NIC 驱动程序

一 NDIS NIC 微端口驱动程序
二 NIC 微端口操作和函数概要
三 NIC 微端口驱动程序入口点和初始化
四 数据传输
五 获取和设置 WMI 的微端口信息及 NDIS 支持
六 微端口的电源管理
七 重置，停止和关闭
八 广域网微端口 NIC 驱动程序
九 任务卸载
十 负载平衡和失效替换
十一 快速转发路径
十二 带 WDM 低级接口的微端口驱动程序
十三 IrDA 微端口 NIC 驱动程序

第一章 NDIS NIC 微端口驱动程序

这一章介绍了 Windows 2000 的网络接口卡(NIC)微端口，以下是将要讨论的主题：

- 1.1 NIC 微端口驱动程序类型
- 1.2 网络接口卡支持
- 1.3 微端口驱动程序代码的重要特征
- 1.4 驱动程序例子

1.1 NIC 微端口驱动程序类型

NDIS 支持以下几种类型的网络接口卡(NIC)驱动程序：

·无连接微端口 为无连接网络介质控制 NIC，例如 Ethernet，FDDI 和 Token Ring。

无连接微端口进一步又可分为以下几种子类型：

·串行化驱动程序 依赖 NDIS 来串行调用 MiniportXxx 函数和管理发送队列。

·非串行化驱动程序 它通常运行在多处理器系统上来获得更高的操作性能，它串行化自己的 MiniportXxx 函数操作，并且在内部对所有发送包进行排队，而不是依赖 NDIS 来完成这些功能。这种结果显而易见地提高了效率，例如驱动程序的临界区(一个时间内只可一个线程客访问代码区)保持很小，但是，非串行化微端口必须满足额外的并且是极苛刻的设计要求，同时还要求有额外的调试和测试时间。

·面向连接的微端口 为面向连接的网络介质（例如 ATM）控制 NIC。

面向连接的微端口通常是非串行化的。这就是说，它们经常自己串行化自己的 MiniportXxx 函数操作，并且在内部进行发送包的排队。

NDIS 库继续支持早期的 NDIS3.0 NIC 驱动程序。但是，只有 NDIS4.0 和 NDIS5.0 微端口可以使用扩充功能和当前 NDIS 库所支持 NIC 驱动程序的特性。也只有 NDIS 微端口 NIC 驱动程序可以使用为 NDIS 库的将来版本所制定的功能。

1.2 网络接口卡支持

Windows 2000 支持以下几种类型的网络接口卡：

- Ethernet (802.3)
- Token Ring (802.5)
- FDDI
- LocalTalk
- Raw ARCNET （ARCNET 封装在 Ethernet)
- ARCNET878.2
- WAN(点对点和 WAN 卡)
- 面向连接的 WAN
- 无线
- ATM
- IrDA

当 NDIS 调用一个 NIC 驱动程序的 MiniportInitialize 函数时，它传递了一组 NDIS 支持的介质。如果 NIC 驱动程序仅支持一种介质类型，它选择它所支持的介质。或者，如果 NIC 驱动程序支持多于一种的介质类型，它选择它所喜欢的那一种并且将它的选择返回给 NDIS。

当一个高层 NDIS 协议驱动程序调用 NdisOpenAdapter 来绑定一个具体的 NIC 时，它提供了一系列它所能操作的介质类型，NDIS 使用来自 NIC 的信息和来自协议驱动程序的信息来正确绑定它们。这个绑定提供了一条路径，通过这条路径，包可以在协议栈中往下传递并通过网络传递出去，通过这条路径，接收到的包可以传到高层驱动程序。

为 ARCNET 定义的介质类型是 NdisMediumArcnet878_2，这种介质类的定义与为 RawARCNET 定义的 Novell 的 ARCNET 说明完全兼容。

Raw ARCNET 格式，由 Novell 出版的《ARCNET Packet Header Definition standard》中定义，它由 NetWare 使用，支持这种格式的传输驱动程序负责从这三种不同帧格式(short，long 或 exception)中选择格式，所选的格式依赖于它想送出数据的长度，传输驱动程序负责执行与使用这个标准相关的必要操作，包括组帧，填充数据和设置分隔标志和顺序号等。

NdisMediumArcnet878_2 格式允许传输驱动程序知道它是运行在 ARCNET 上的，但不需要处理帧的分段和组装细节。NdisMediumArcnet878_2 的 on-the-ware 格式与 Raw ARCNET 相同。但是，许多帧格式的细节对传输驱动程序都屏避了。NIC 驱动程序执行组帧操作，选择合式的帧格式，填入数据，设置分隔标志和队列号，以及其他相关字段。NDIS 库对发送的包从 802.3 格式转换成 ARCHET 格式，对从 ARCNET 微端口接收到的包从 ARCHET 格式转换成 802.3 格式。

Windows 2000 Tcpip 和 Nwlink 传输驱动程序使用 NdisMediumArcnet878_2。

Windows 2000 Nbf 传输(也叫 NetBEUI)则不使用它。

NdisMediumAtm 格式允许 NIC 驱动程序使用 NDIS5.0 的固有的 ATM 支持。

NdisMediumIrda 格式允许 NIC 驱动程序使用 NDIS5.0 的固有的红外线驱动器支持。

NIC 的类型

微端口用 NdisXxx 函数来初始化它的适配器和发送与接收包数据，这些函数依赖于它的 NIC 的如下特征。

1. 总线管理 DMA　NIC

这些 NIC 可以通过一个 DMA 控制器直接访问主机的内存，DMA 控制器用来管理在网络和主机内存之间不经过主机 CPU 进行数据传输。

当发送时，微端口驱动程序将 NIC 映射到发送的缓存中。然后微端口触发设备从这个内存开始传输。NIC 的 DMA 控制器将数据从共享的系统内存传输到网络，并且当传输结束后向 CPU 发出中断。当接收时，DMA 控制器将接收的数据传输到主机内存后，向主机发出一个中断指示。

一个总线管理 DMA NIC 一般地装有一个板上的环行缓冲区，微端口可以将它映射到一系列系统内存中的缓冲区。典型地，可对 NIC 进行编程来有效地处理好几个包。由于 NIC 可以有效地处理好几个包，所以管理这种 NIC 的微端口驱动程序通常支持多包的发送和接收，因此这样就可以提高 I/O 效率。

2. 非总线管理 DMA NIC

这包含三种基本的 NIC 类型。

• NIC 包含板上的共享内存。

微端口管理的 NIC 必须能将 NIC 的共享内存映射到主机内存，并且可将发送包拷贝到 NIC 内存，或者将收到的帧从 NIC 内存拷贝到由上层协议驱动程序或其他驱动程序提供的缓冲区中。这种微端口基本上不能通过支持多包发送和接收来提高它的能力。

• 从属的 DMA NIC

一个微端口管理的 NIC 使用系统的 DMA 控制器来管理从网络发送和接收的数据包。传输数据要求主机 CPU 的合作。

• 使用可编程的 I/O 的 NIC

微端口管理的 PIO 设备使用的 NDIS 函数以 byte， word，long 为单位将发送帧转移到设备寄存器，然后让设备发送数据，这种设备的驱动程序并没有从 NDIS 的多包发送和接收支持中受益。这种 NIC 类型的微端口只实现单包的发送和接收，如第四章所述。

1.3 微端口驱动程序代码的重要特征

这一节介绍微端口驱动程序代码的以下几个重要特征。

• 1.3.1 MiniportXxx 函数
• 1.3.2 与 NDIS 库链接
• 1.3.3 微端口适配器环境
• 1.3.4 VC 环境 (仅针对面向连接的微端口)
• 1.3.5 网络 OID

1.3.1 MiniportXxx 函数

典型的微端口驱动程序仅使用很少的函数来通过 NDIS 与上层和硬件进行通信。NDIS 为了完成自身的任务或协议驱动程序的服务,调用在 2.2.1 节和 2.2.2 节提供上层(MiniportXxx)函数列表与微端口通信；并不是所有的这些函数都是必须的。3.1.2.2 节将介绍哪些函数是可选的，哪些函数是必须的，并且给出了原因。

NDIS NIC 微端口和上层驱动程序使用 NDIS 库(ndis.sys)通过调用 NdisXxx 函数在彼此间通信。一个关于微端口可以调用的 NdisXxx 函数和宏的总结在 2.3 节提供。

许多微端口函数可以异步运行或同步运行。对于异步函数，当操作完成以后，必须调用 NdisXxx...Complete。例如，如果一个协议驱动程序调用 NdisReset 来复位一个 NIC，微端口的 MiniportReset 函数通过返回 NDIS_STATUS_PENDING 来挂起复位操作。最终，MiniportReset 函数必须调用 NdisMResetComplete 来指示复位请求的最终状态。参见第二章和在线 DDK 的“Network Drivers Reference”来获得更多的关于异步 MiniportXxx 函数以及相应的完成函数的信息。

1.3.2 与 NDIS 库链接

NDIS 库封装在 ndis.sys 中，这个内核模式库提供了一系列函数接口，它侧重于使功能最大化的宏。提供接口的库是一个.sys 文件，它的函数与动态链接库相似。NDIS 库函数在在线 DDK 的“Network Drivers Reference”中给予了讲述，包括协议和 NIC 驱动程序，以及 WAN NIC 驱动程序，它们都与 NDIS 库相链接。

Windows 2000 DDK 提供了 ndis.h 作为微端口的主要头文件。这个文件定义了微端口入口点，NDIS 库函数，和通常的数据结构。在线的 DDK 的“Network Drivers Reference”讲述了微端口，协议和 NdisXxx 函数，以及通常的数据结构和 OID。

1.3.3 微端口适配器环境

NDIS 使用叫一个逻辑适配器的软件对象来代表系统中每一个 NIC。这个对象由 NDIS 管理，并且对微端口和协议驱动程序是透明的。NDIS 将这个结构的句柄传递给 NIC 驱动程序的 MiniportInitialize 函数。以后微端口驱动程序在调用 NdisXxx 函数时，如果这个函数与

此句柄表示的 NIC 有关，它应在函数中传递此句柄。

当调用微端口 NIC 驱动程序来初始化一个它管理的 NIC 时，微端口驱动程序产生自己的内部数据结构来代表 NIC。驱动程序使用这个结构，把它当作微端口适配器环境，来维护特定设备的状态信息，这些信息是管理 NIC 所必须的。当驱动程序的 MiniportInitialize 函数调用 NdisMSetAttributes 或 NdisMSetAttributesEx 时，它将这个结构的句柄传递给 NDIS。当 NDIS 调用与 NIC 相关的微端口的 MiniportXxx 函数时，它将指向 NIC 的句柄传递给适当驱动程序。微端口适配器环境为微端口所拥用并且为微端口所管理，它对于 NDIS 和协议驱动程序是透明的。

1.3.4 VC 环境

在建立一个呼叫之前，面向连接的客户方请求面向连接的微端口建立一个虚拟连接(VC)，通过它可以发送和/或接收包。同样，在向面向连接的客户方指示一个入站呼叫之前，呼叫管理器或集成的微端口呼叫管理器(MCM)请求微端口为入站呼叫建立一个 VC。一个虚拟连接是两个面向连接实体间的一个逻辑连接。面向连接的发送和接收通常在一个特定的 VC 上进行。

面向连接微端口在微端口在为每个 VC 分配的环境区中保存状态信息。每一个 VC 的环境由微端口来管理，并且它对于 NDIS 和协议驱动程序是透明的。在它的 MiniportCoCreateVC 函数中，面向连接的微端口将 VC 环境区域的句柄传递给 NDIS，NDIS 将一个唯一指定的已存在的 VC 的 NdisVcHandle 回传给微端口、回传给适当的的面向连接的客户方、回传给呼叫管理器或集成微端口呼叫管理器(MCM)。

在 VC 上发送和接收数据之前，VC 必须被激话。呼叫管理器通过调用 Ndis（M）CmDeactivateVc 来初始化激活的 VC 并且向呼叫管理器传递呼叫参数，这包括用来激活 VC 的参数。作为响应，NDIS 调用微端口的 MiniportCoActivateVc 函数来激活 VC。

在一个呼叫结束或因其他原因不再需要 VC 时，呼叫管理器通过调用 Ndis(M)CmDeactivateVc 来去活 VC，这也将引起 NDIS 调用微端口的 MiniportCoDeactivateVc 函数。对于面向连接的客户方或呼收管理器，可以通过调用 NdisCoDeleteVc 来指示对 VC 的删除，它将引起 NDIS 调用微端口的 MiniportCoDeleteVc 函数。

更多在 VC 上进行微端口操作的信息，请参见第一部分的 “微端口驱动程序指南”。

1.3.5 网络 OID

微端口维护有关它的性能和当前状况的信息，以及有关它所控制的每个 NIC 的信息。每一个信息类型都由一个对象标识(OID)确认，OID 由系统定义。

NDIS 和高层驱动程序可以使用 OID 来查询信息以及在某些情况下设置信息。

• 对于无连接介质的高层驱动程序，它调用 NdisRequest 来查询或设置一个无连接微端口中的信息。为了执行查询操作，NDIS 调用微端口的 MiniportQueryInformation 函数。为了执行设置操作，NDIS 调用微端口的 MiniportSetInformation 函数。

• 对于面向连接介质的高层驱动程序，它调用 NdisCoRequest 来查询或设置面向连接微端口中的信息。为了执行查询和设置操作，NDIS 调用微端口的 MiniportCoRequest 函数。

NDIS 将许多系统为微端口定义的 OID 映射为全局的唯一标识(GUIDs)。NDIS 向内核模式的 Windows 管理检测器(WMI)来注册这些 GUIDS。WMI 支持用户模式下的基于 Web 的企业管理(WBEM)应用。当一个 WMI 客户方查询或设置这些 GUIDS 中的一个值时，NDIS 将这些请求转换为一个恰当的 OID 查询操作或一个 OID 设置操作，并且把信息和状态返回

给 WMI。驱动程序编写者可将自定义的 GUID 映射为客户 OID 或微端口状态。一个微端口必须在初始化时向 NDIS 注册客户的 GUID-to-OID 或 GUID-to-status 映射。

更多有关查询和设置 OID 信息、创建自定义 OID 以及 WMI 的 NDIS 支持，参见第五章。有关系统定义的 OID 的描述，参见在线的 DDK 的“Network Drivers Reference”。

1.4 驱动程序例子

DDK 包括了用来管理几种类型网络卡的微端口的代码例子。这些驱动程序的例子可以根据驱动程序开发者的要求进行组合。驱动程序的例子包含了可以改写为新的相似的驱动程序的函数。硬件相关的函数必须由驱动程序开发者重写。但是，许多函数都是很标准的。例如，与 NDIS 库而不是与网络接口卡通信的函数都是很标准的。对于这些驱动程序函数，在驱动程序例子中的代码可以经过很少的改动，有时甚至不经过修改就可使用。

第二章 NIC 微端口操作和函数概要

本章提供了一个关于 NIC 微端口操作的概要，例如初始化和发送数据。这些操作涉及 MiniportXxx 函数和 NdisXxx 函数。这一章也提供了一个关于 MiniportXxx 函数的概要（它是由 NIC 微端口驱动程序提供的），以及关于 NdisXxx 函数的概要（它是一个由 NIC 微端口驱动程序调用的）。

这一章包括以下内容：

- 2.1 NIC 微端口操作
- 2.2 微端口上层函数
- 2.3 微端口调用的 NDIS 函数

2.1 NIC 微端口操作

NIC 微端口所执行的操作如下所示：

- 2.1.1 初始化 NDIS 库和注册微端口驱动程序
- 2.1.2 注册网络接口卡
- 2.1.3 对查询和设置微端口信息作出响应
- 2.1.4 产生、激活、去活和删除虚连接(仅对面向连接的微端口)
- 2.1.5 发送数据
- 2.1.6 指示和传递接受的的数据
- 2.1.7 向上层指示硬件状态的变化
- 2.1.8 重新设置一个网络接口卡
- 2.1.9 卸载和注销驱动程序

2.1.1 初始化 NDIS 库和注册微端口驱动程序

当装载一个微端口驱动程序时，操作系统为微端口驱动程序产生一个驱动程序对象并且调用驱动程序的 DriverEntry 函数。在 DriverEntry 函数的环境中，微端口完成以下工作：

- 调用 NdisMInitializeWrapper，将驱动程序对象的地址和指向确定程序相关信息的注册表路径作为参数。
- 调用 NdisMRegisterMiniport 来将自己注册到 NDIS。包括他兼容的 NDIS 版本号以及提供他的上层（MinportXxx）函数的入口点。NDIS 库保存这些入口指针供以后调用微端口时使用。

如果 NdisMRegisterMiniport 或 NdisIMReqisterLayerdMiniportde 的调用没有返回 NDIS_STATUS_SUCCESS，那么 DriverEntry 将返回由 NdisMRegisterMiniport 或 NdisIMReqisterLayerdMiniport 返回的值，并且调用 NdisTerminateWrapper。对 NdisTerminateWrapper 的调用将使的 NDIS 清除微端口调用 NdisMInitializeWrapper 时分配的资源。

有关更详细的 DriverEntry 函数的描述，参见 3.1 节。

2.1.2 注册网络接口卡

在经过网络接收和发送包之前，NDIS 库必须注册它所管理的每一个网络接口卡(NIC)。

MiniportInitialize 函数做如下工作：

• 为维护 NIC 运行状态的适配器相关环境区域分配内存。

• 读取注册表配置数据库来确定它正初始化的 NIC 的参数。微端口在适配器相关环境区域，为 NIC 贮存这些信息。

• 如果可能的话，读取由 I/O 总线维护的总线相关信息。有关总线相关信息的例子包括 PCI 或 EISA 信息。

•通过调用 NdisMSetAttributes 或 NdisMSetAttributesEx 向 NDIS 注册 NIC。一个非串行化微端口必须通过设置 NDIS_ATTRIBUTE_DESERIALIZE 标志，来调用 NdiMSetAttributesEx 表明它被串行化了。

• 声明操作 NIC 所需的系统资源。这些资源包括包所需的内存和缓冲区，用来对设备的 I/O 端口地址进行读写操作，总线管理器要求的映射注册，等等。

• 初始化 NIC

• 如果 NIC 产生一个中断，注册一个中断

• 注册一个关闭句柄

• 初始化任何一个用来选取 NIC 的轮询时钟

有关 MiniportInitialize 函数的更多详情，参见 3.2 节。

2.1.3 对查询和设置微端口信息作出响应

为了查询无连接微端口的性能和统计信息，NDIS 调用微端口的 MiniportQueryInformation 函数。NDIS 可为完成自身任务或为上层的任务调用这个函数。查询一个面向连接微端口，NDIS 调用微端口的 MiniportCoRequest 函数。MiniportQueryInformation 或 MiniportCo Request 函数获取信息并且返回给 NDIS。

为了设置由一个无连接微端口维护的信息，NDIS 调用微端口的 MiniportSetInformation 函数，它设置指定的对象标识并且将操作状态返回给 NDIS。为了设置由一个面向连接微端口维护的对象标识（OID），NDIS 调用微端口的 MiniportCoRequest 函数。

微端口查询和设置函数可能是异步的。如果它们是同步完成的，它们立刻返回一个状态代码而不是 NDIS_STATUS_PENDING。如果它们是异步完成的，函数返回 NDIS_STATUS_PENDING。为了异步完成一个查询或设置，无连接微端口将分别为 MiniportQueryInformation 调用 NdisMQueryInformationComplete 或为 MiniportSetInformation 调用 NdisMSetInformationComplete。一个面向连接的微端口调用 NdisMCoRequestComplete 来完成异步查询或设置。

NDIS 库确保微端口不会有一个未完成的查询或设置请求，所以微端口不需要对查询排队。

更多有关查询和设置无连接微端口的内容，参见第五章，有关特定的 OID 信息，参见在线 DDK 的“Network Drivers Reference”。

2.1.4 产生，激活，去活，和删除虚连接

一个面向连接的微端口经常通过一个虚连接(VC)来发送和接收数据——无论这个包是为面向连接客户方的还是为一个呼叫管理器发消息。在由 VC 发送和接收数据之前，一个面向连接微端口必须为每一个 VC 建立环境，它包括分配和初始化内存，推荐用 Ndis...NpagedLookasideList 函数完成此项任务。然后微端口必须激活 VC，这包括进行 NIC 所要求的各种处理以满足请求需要。当一个呼叫不再需要 VC 时，微端口去活 VC 并且也许

会删除 VC。

在下列情况下，一个面向连接的微端口产生一个 VC。

•在呼叫管理器和微端口完成了它们的初始化操作后，呼叫管理器调用 NdisCoCreateVc 请求微端口建立一条呼叫管理器可用来发送信令消息的 VC。作为这个调用的响应，NDIS 调用 MiniportCoCreateVc 函数，它用来建立每个 VC 的环境。

• 当一个网络管理员手工添加一个永久 VC(PVC)时，呼叫管理器调用 NdisCoCreateVc 来请求微端口建立 VC。

•在请求一个呼叫管理器建立一个外出呼叫之前，面向连接客户方调用 NdisCoCreateVc 请求微端口建立一个 VC 用来发送呼叫数据。

•在向一个面向连接的客户方指示一个内入呼叫之前，呼叫管理器调用 NdisCoCreateVc 来请求微端口建立一条 VC 用来接收呼叫数据。

在一个呼叫结束或因其他原因不再需要 VC 时，呼叫管理器或集成微端口呼叫管理器(MCM)通过调用 NdisCmDeactivateVc 来去活 VC，它将引起 NDIS 调用微端口的 MiniportCoDeactivateVc 函数。对面向连接的客户方或呼叫管理器可以通过调用 NdisCoDeleteVc 来引发删除 VC 操作，它将引发 NDIS 调用微端口的 MiniportCoDeleteVc 函数。MiniportCoDeleteVc 断开由微端口分配的每个 VC 状态。

更多有关在 VC 上的微端口操作信息和由 NDIS 提供的面向连接的结构的概述，参见第四部分的第四章。

2.1.5 发送数据

一个无连接微端口可以有两种函数来发送包：

• MiniportSendPackets

NDIS 一次将一个或多个包作为一组指向包标识符的指针传给 MiniportSendPackets。

• MiniportSend

NDIS 通常一次传给 MiniportSend 一个包标识符。

面向连接的微端口必须有一个 MiniportCoSendPackets 函数用来发送包。NDIS 一次将一个或多个包作为一组指向包标识符的指针传给 MiniportCoSendPackets。

2.1.5.1 多包发送

无连接微端口的 MiniportSendPackets 或面向连接微端口的 MiniportCoSendPackets 接收一个指向一组指针的指针，这一组指针指向一个或多个将要发向网络的包。微端口必须保存包传递给 Miniport(Co)SendPackees 函数的顺序。

NIC 驱动程序可以使用 NDIS 宏从包标识符中取出带外(OOB)数据中的介质特性和权值信息，(参见 4.3 节)

从串行微端口多包发送

一个串行微端口可以同步或异步地完成发送操作。在它的 MiniportSendPackets 函数中(如果提供这样的函数)，串行化微端口必须在 MiniportSendPackets 返回之前，为每个包在 OOB 数据中设置发送完成状态。任何一种非 NDIS_STATUS_PENDING 的状态都表明了微端口处理了包(即就是，同步地完成了发送操作)并且返回了包标识符的所有权和与它相关的用来重新使用或将要释放的资源。如果微端口在一个包中返回 NDIS_STATUS_PENDING，那么驱动程序保持包资源的所有权，直到微端口调用 NdisMSendComplete 处理包为止。更多有关从串行微端口进行多包发送的操作参见 4.4.1.1 节。

从非串行化微端口多包发送

一个非串行化微端口的 MiniportSendPackets 函数并不能返回 NDIS_STATUS_RESOURCES，并且必须通过调用 NdisMSendComplete 来异步地完成对每一个包的发送。因此，NDIS 忽略由非串行微端口发送来的包中的 OOB 数据发送完成状态。当 MiniportSendPackets 没有足够资源发送一个给定的包时,非串行化微端口管理自己的内部包队列，而不是依赖 NDIS 来排队和重提交包，NDIS 也不会要求重新发包。一个非串行化微端口负责在其内部队列中保持发送包，直到它们可以发送到网络为止。并且，一个非串行化微端口必须同步它自身函数的执行，以便 MiniportSendPackets 可以与其他 MiniportXxx 函数保持一致性执行(MiniportPortReset 除外)。更多有关非串行化微端口信息参见 4.5 节。更多有关从非串行化微端口进行多包发送的细节参见 4.4.1.2 节。

通过面向连接微端口多包发送

如同一个非串行化微端口，一个面向连接的微端口：

- 不能返回 NDIS_STATUS_RESOURCES 作为多包发送的响应
- 在一个多包发送中并没有为每个包指示发送一完成状态
- 必须对 NDIS 传递给它的包进行内部排队
- 同步它自己的 MiniportXxx 函数的执行

面向连接微端口与无连接微端口不同点在于它使用 MiniportCoSendPackets 并且总是在一个虚连接(VC)上发送(接收)包。更多通过面向连接微端口多包发送的信息，参见 4.4.3 节。

2.1.5.2 单包发送

串行化和非串行化的非连接微端口可以使用 MiniportSend 函数，将单个包通过 NIC 发送到网络上。MiniportSent 将包转换成数据帧并且执行发送操作。在微端口实际发送包之前，它可以通过调用 NdisGetPacketFlags 来查询包标识标，从而读取协议驱动程序传递给 NIC 驱动程序的包标识符头结构中的 flags 成员。flags 成员包含着发送操作的信息（它不包括在包的数据中），这个信息被用于在相关的协议和定义标记的微端口驱动程序之间进行信息通信。如果驱动程序支持这个特性的话，驱动程序也可以查询带外(OOB)数据的权值信息和介质。

如果在返回之前 MiniportSend 完成了发送操作，它将返回一个状态代码而不是 NDIS_STATUS_PENDING 或 NDLS_STATUS_RESOURCES。完成发送操作意味着微端口驱动程序完成了包标识符和它所指向的资源，并且将资源返回给调用者以重新使用或等待释放。

在异步情况下，微端口返回 NDIS_STATUS_ RESOURCES，并且当它对包操作以后，必须调用 NdisMSendComplete，用来表明释放了包，以便重新使用或等待删除包。

串行化微端口可以返回 NDIS_STATUS_PENDING 来表明它没有足够的内部资源发送包。在这种情况下，NDIS 将返回的包标识符在内部排队，并且当有更多资源可用时，或微端口调用 NdisMSendComplete 或 NdisMSendResourcesAvailable 时，不管哪一个事件先发生，都促使 NDIS 将它再提交给 NIC 驱动程序。

当串行化微端口调用 NdisMSendComplete 或 NdisMSendResourcesAvailable 时，它向 NDIS 表明发送资源是可用的，并且微端口已作好了接收新包标识符的准备。

更多有关从无连接微端口单包发送的描述参见 4.4.2 节。

2.1.6 指示和传递接收的数据

一个无连接微端口可以调用以下函数中的一种来将到来的数据指示给上层：

- NdisMIndicateReceivePacket：参数为一个指向一组指向一个或多个接收数据包标识符的指针的指针。

•NdisMXxxIndicateReceive，参数为一个指向预先准备的缓存地址的指针，相关协议驱动程序将数据拷贝到这个缓存。

一个面向连接的微端口必须调用 NdisMCoIndicateReceivePacket，将到来的数据传送到上层。

2.1.6.1 多包接收

当一个无连接微端口调用 NdisMIndicateReceivePacket 或一个面向连接的微端口调用 NdisMCoIndicateReceivePacket 时，微端口将一个指向已完成包的标识符的指针上传给相关协议驱动程序。向上指示的每一个包标识符的返回状态都必须独自完成设置。任一个 NdisM(Co)IndicateReceivePacket 返回未定状态的包标识符都将为上层所保留。包标识符所描述的资源将在以后由微端口的 MiniportReturnPacket 函数返回。如果 NdisM(Co)IndicateReceivePacket 返回任意一个非 NDIS_STATUS_PENDING 的状态，它所描述的包标识符和资源的所有权将返回给微端口。

一组指向已指示包的指针的所有权在从 NdisM(Co)IndicateReceivePacket 返回时，就已返回给了微端口驱动程序，而不用考虑任一个包标识符的返回状态。

面向连接微端口应指出包从那个 VC 上接收到的。对于从网络接收内入呼叫的面向连接呼叫管理器或集成微端口管理器(MCM)的请求，要求微端口提前建立和激活 VC(这些操作是由 NDIS 调用微端口的 MiniportCoCreateVc 和 MiniportCoActivateVc 函数完成的)。微端口在 NdisMCoIndicateReceivePacket 中传递 NDIS 提供的 VC 句柄，有关多包接收的细节参见 4.6.1。

2.1.6.2 单包接收

无连接微端口调用特定协议的 NdisMXxxIndicateReceive 函数来向上层指示一个单独的包。当网络接口卡(NIC)从网络接收到一个数据包，它的微端口通过为网络介质调用恰当的函数向上指示一个包：

- 对于 ARCNET，微端口调用 NdisArcIndicateReceive
- 对于 Ethernet，微端口调用 NdisEthIndicateReceive
- 对于 FDDI，微端口调用 NdisFddiIndicateReceive
- 对于 Token Ring，微端口调用 NdisTrIndicateReceive

微端口向上指示一个部分包，如果包很小的话也许会是一个完整的包。NDIS 将这个指示传递给所有相关协议。如果它是一个部分包并且协议对完整包感兴趣，那么协议调用 NdisTransferData 请求包的剩余部分，它向下传递一个由分配协议分配的包描述符。

然后，NDIS 调用微端口的 MiniportTransferDate 函数通知微端口将数据传递到链接在包描述符中的协议分配缓存中，MiniportTransferData 将接收到的数据帧拷贝到缓存中。

如果 MiniportTransferData 同步完成，它仅仅将接收到的数据写入所提供的缓存中，并且返回非 NDIS_STATUS_PENDING 状态。如果 MiniportTransferData 在返回之前不能完成传递操作，它将返回 NDIS_STATUS_PENDING。当数据传送完毕，微端口必须调用 NdisMTransferDataComplete。

在 NIC 完成接收所有发送来的数据帧之后，微端口为网络媒体调用恰当的 NdisMXxxIndicateReceiveComplete 函数。这将导致对上层协议驱动程序的调用，指示上层协议驱动程序可以执行 post—receive 处理。

更多有关单包接收的描述，参见 4.6.2 节。

2.1.7 指示状态

一个无连接微端口调用 NdisMIndicateStatus，指示它管理的 NIC 的状态变化，面向连接微端口调用 NdisMCoIndicateStatus 来指示这种变化。

NdisM(Co)IndicateStatus 传递这个调用到所绑定的协议驱动程序，协议驱动程序将解释基本的状态代码。然后，协议驱动程序、NDIS 库、WMI 或其他 OS 入口可以记录状态变化，采取适当的动作，或者两者都进行。

无连接微端口在发送状态变化时，必须指示状态变化。在一个或多个 NdisMIndicateStatus 调用之后，无连接微端口调用 NdisMIndicateStatusComplete。NDIS 将完成调用传递给协议驱动程序使得它们可以进行任何必要的后续处理。当一个面向连接微端口发送状态变化时，它不用进行指示。

微端口不允许在它的 MiniportInitialize，MiniportISR，MiniportHalt，或 MiniportShutdown 函数的环境中指示状态。

有关指示状态的细节，参见 5.4 节。

2.1.8 复位网络接口卡

当下列情况发生时，NDIS 复位 NIC：

- NIC 驱动程序的 MiniportCheckForHang 函数返回 TRUE
- 当一个对微端口的请求(查询或设置)操作超时
- 当一个串行微端口发送操作超时

当 NDIS 库决定必须重置一个 NIC 时，它调用微端口的 MiniportReset 函数。MiniportReset 向 NIC 发布一个硬件重置，并且更新它自己的软件状态。

MiniportReset 以一个成功或失败状态代码同步的完成，或以 NDIS_STATUS_PENDING 异步的完成。如果函数是异步的，微端口随后调用 NdisMResetComplete 来允许后续处理。

当一个重置正在进行时，NIC 驱动程序必须保证 NDIS 不会发送任何请求，并且，NDIS 将完成任何目前未完成的请求或发送，所以微端口不需要明确地完成它们。

有关重置串行微端口管理的 NIC 的细节描述，参见 7.1 节。

2.1.9 终止一个微端口 NIC 驱动程序

NDIS 通过调用微端口的 MiniportHalt 函数来终止一个微端口。MiniportHolt 函数释放由微端口所声明的系统资源。例如，如果 MiniportInitialize 函数通过 NdisMRegisterInterrupt 注册了一个中断，那么微端口的 MiniportHalt 函数必须通过 NdisMDeregisterInterrupt 注销这个中断。

如果微端口初始化一个 NIC 或声明所需的系统资源失败，它的 MiniportInitialize 函数释放所有分配给微端口的资源。否则，微端口将在它的 MiniportHalt 函数的环境中释放它的资源。

由于一个微端口的 MiniportShutdown 函数不应释放任何已分配资源，MiniportShutdown 函数应当停止任何正在进行的数据传输并且将 NIC 置为初始状态。

有关终止一个微端口 NIC 驱动程序的更多信息，参见 7.2 节。

2.2 微端口上层函数

这一节提供了由微端口支持的上层函数的概述。NDIS 为完成自身任务或其他层的网络软件的任务而调用这些函数。例如，这个高层网络软件可以是一个绑定到由 NIC 驱动程序

管理的 NIC 上的高层协议驱动程序。一个微端口 NIC 驱动程序必须提供一些或所有的上层函数。

这里所讨论的微端口驱动程序函数都使用 Miniport 作为一个基本的前缀，但是为一个实际的微端口驱动程序编写函数，应当使用更多的描述前缀以使得容易调试。因为函数的地址(而不是名字)在初始化时传递到 NDIS 库，驱动程序开发者可自由命名 MiniportXxx 函数。

有一些 NIC 驱动程序函数本来就是同步的，而其余的可同步完成或异步完成。当一个微端口函数返回 NDIS_STATUS_PENDING 时，微端口必须通过调用恰当的 NdisM...Complete 函数来完成请求。NDIS 调用其他层的用来完成异步请求的函数，与这些层相协调操作。

2.2.1 无连接微端口的上层函数

下表中总结了 NDIS 定义的一个无连接微端口能提供或必须提供的微端口驱动程序函数。

函数	描述	异步选项
DriverEntry	由操作系统调用来激活和初始化微端口驱动程序	N/A
MiniportAllocateComple	调用它来指示以前调用的 NdisMAllocateMemoryAsync 已经完成	否
MiniportCheckForHang	检查 NIC 的内部状态	否
MiniportDisableInterrupt	禁止 NIC 产生中断	否
MiniportEnableInterrupt	允许 NIC 产生中断	否
MiniportHalt	重新分配和重新注册 NIC 占用的资源并且终止 NIC 以使它不再有用	否
MiniportHandleInterrupt	延期用来完成 I/O 中断函数的执行	否
MiniportInitialize	出始化 NIC	否
MiniportISR	作为 NIC 中断服务例程以高权值运行	否
MiniportQueryInformation	查询微端口驱动程序的性能和当前状态	否
MiniportReconfigure	未使用	否
MiniportReset	对 NIC 发出一个硬件重置	可以
MiniportReturnPacket	从上层接收一个包，这个包是在这之前通过调用 NdisMIndicateReceivePacket 函数传上去的包	否
MiniportSend	如果驱动程序没有 MiniportSendPadcets 或 MiniportWanSend 函数，它用来将一个包通过 NIC 传递到网络上	可以
MiniportSendPackets	通过 NIC 传递一组包到网络上	可以
MiniportSetInformation	变换(设置)关于微端口驱动程序或它的 NIC 的信息	可以
MiniportShutdown	当系统暂时关闭时，将 NIC 恢复到初始状态	否
MiniportSynchronizeIS	同步访问与 MiniportISR 或	否

R	MiniportDisableInterrupt 共享的资源。如果有运行在 DIRQL 下的 NIC 驱动程序函数与驱动程序的 MiniportISR 或 MiniportDisableInterrupt 函数共享资源，需要此函数	
MiniportTimer	如果微端口的 NIC 不产生中断，用于轮询 NIC 的状态	否
MiniportTransferData	将由 NIC 接收到的包的内容拷贝到一个给定的包缓存中	可以
MiniportWanSend	如果驱动程序控制着 WAN NIC，通过 NIC 发送一个包到网络上	否

下表总结了 NDIS 完成函数，它用来对每个无连接微端口的异步上层函数进行响应。

函数	异步完成函数
MiniportSend MiniportSendPackets	NdisMSendComplete
MiniportQueryInformation	NdisMQueryInformationComplete
MiniportReset	NdisMResetComplete
MiniportSetInformation	NdisMSetInformationComplete
MiniportTransferData	NdisMTransferDataComplete

2.2.2 面向连接微端口的上层函数

下表总结了一个面向连接微端口可以或必须提供的 NDIS 定义下的微端口驱动程序函数。

函数	描述	异步选项
DriverEntry	由操作系统调用来激活和初始化微端口驱动程序	N/A
MiniportAllocateComplete	调用它来指示以前调用的 NdisMAllocateMemoryAsync 已经完成	否
MiniportChecKForHang	检查 NIC 的内部状态	否
MiniportCoActivateVc	激活一个虚连接(VC)	可以
MiniportCoCreateVc	为一个 VC 建立 VC 状态	否
MiniportCoDeactivateVc	为一个 VC 释放 VC 的状态	可以
MiniportCoDeleteVc	删除一个 VC	否
MiniportCoRequest	查询微端口驱动程序的能力和当前状态或改变(设置)有关微端口驱动程序或它的 NIC 的信息	可以
MiniportCoSendPackets	通过一个 NIC 发送一组包到网络上	可以
MiniportDisableInterrupt	禁止一个 NIC 产生中断	否
MiniportEnableInterrupt	允许一个 NIC 产生中断	否
MiniporeHalt	收回和注销 NIC 占用的资源并且终止 NIC 以使它不再使用	否
MiniportHandIeInterrupt	延期用来完成 I/O 中断函数的执行	否
MiniportInitialize	初始化 NIC	否

MiniportISR	作为 NIC 中断服务例程以高权值运行	否
MiniportReconfigure	未使用	否
MiniporeReset	对 NIC 发出一个硬件重置	可以
MiniportReturnPacket	从上层接收一个包，这个包是在这以前通过调用 NdisMCoIndicateReceivePacket 传上去的	否
MiniportShutdown	一个可选择函数，当系统关闭时，将 NIC 恢复到它的初始状态	否
MiniportSynchronizpISR	同步访问与 MiniportISR 或 MiniportDisableInterrupt 共享的资源，如果有运行在 DIRQL 下的 NIC 驱动程序函数与驱动程序的 MiniportISR 或 MiniportDisableInterrupt 函数共享资源	否
MiniportTimer	如果微端口 NIC 不产生中断，轮询一个 NIC 状态	否
MiniportWanSend	如果驱动程序控制着 WAN NIC ，通过网络接口卡发送一个包到网络上	否

下表总结了 NDIS 完成函数，它用来对每个对面向连接微端口可能实现的异步上层函数进行响应。

函数	异步完成函数
MiniportCoActivateVc	NdisMCoActivateVcComplete
MiniportCoDeactivateVc	NdisMCoDeactivateVcComplet
MiniportCoRequest	NdisCoRequestComplete
MiniportCoSendPackets	NdisMCoSendComplete
MiniportReset	NdisMResetComplete

面向连接微端口与无连接微端口关于 MiniportXxx 函数的不同点如下：

• 面向连接微端口必须能产生，激活，去活和删除 VC。MiniportCoCreateVc，MiniportCoActivateVc，MiniporeCoDeactivateVc 和 MiniportCoDeleteVc 函数完成这些任务。

• 无连接微端口的 MiniportSend 或 MiniportSendPackets 函数在面向连接微端口中由 MiniportCoSendPackets 函数取代。

• 无连接微端口的 MiniportQueryInformation 和 MiniportSetInformation 函数在面向连接微端口中由 MiniportCoRequest 函数取代。

• 由于面向连接微端口总是将完整的包指示给绑定的协议，所以它们不需要 MiniportTransferData 函数。

2.3 由微端口调用的 NDIS 函数

当一个 NDIS 微端口需要（例如自旋锁或内存缓存区的）系统资源时，它必须调用 NDIS(NdisXxx)函数或宏。然后，NdisXxx 函数调用内核模式的支持例程来满足微端口的请求。

NDIS 定义了几百个函数和宏，它们中的许多都可被微端口调用。NDIS 库可以想像成标准的 C 库，它为编写一个 NIC 驱动程序提供了丰富的框架。仅有一个 NDIS 子集函数可能被任何给定的 NIC 驱动程序调用。

NDIS 库函数作为一个接口库(ndis.lib)被提供。NDIS 协议和微端口都可链接到这个接口库。

NDIS 库向微端口 NIC 驱动程序提供了以下几种类型的函数：

• 初始化和注册函数

• 硬件设置函数(为 ISA，EISA，PCI 和 PC 卡 NICS)

• I/O 端口函数

- 与 DMA 相关联的函数
- 中断处理函数
- 同步函数
- 状态函数
- 无连接微端口的发送和接收函数
- 面向连接微端口的发送和接收函数
- 带外数据宏
- 包和缓存处理函数
- 支持函数
- 媒体相关宏

以下几节总结了可被一个微端口驱动程序调用的 NDIS 库函数。

2.3.1 NDIS 提供的初始化和注册函数

在初始化当中，微端口驱动程序对自身进行设置，并且通过调用由 NDIS 提供的设置函数向 NDIS 进行注册。有些函数访问由微端口的 INF 文件写入注册表的设置信息。

下表中描述了由 NDIS 提供的初始化和注册函数。

函数	定义
NdisMInitializeWrapper	为微端口驱动程序初始化 NDIS 库数据结构
NdisMRegisterMiniport	向 NDIS 提供有关微端口驱动程序的信息
NdisMSetAttributes	向 NDIS 库报告微端口驱动程序所支持的 NIC 的类型并且传递指向微端口的环境区域的句柄。NDIS 将在接下来的调用中把这个句柄传递给 MiniportXxx 函数
NdisMSetAtteributesEx	除了如同 NdisMSetAttributes 一样传递同样的信息，这个函数改变了缺省的 NIC 超时动作，并且允许一个微端口指定一些附加的标志。一个驱动程序，例如一个 NDIS 中间层驱动程序或一个非 串行化微端口驱动程序，如果必须指定一个或多个这种标志的话，则必须调用这个函数而不是 NdisMSetAttributes
NdisMGetDeviceProperty	检索出设备对象，这个对象被用来通过一个总线驱动程序来建立一个与 NIC 的通信。例如，USB 和 IEEE 1394 总线驱动程序
NdisMQueryAdapterResources	返回一个 NIC 硬件资源列表
NdisMQueryInformationComplete	指示 NDIS 先前的 MiniporeQueryInformation 操作完成。仅被无连接微端口调用
NdisMSetInformationComplete	报告 NDIS 库，以前的 MiniportSetInofrmation 操作已经完成，仅被无连接微端口调用
NdisOpenConfiguration	提供一个调用者可以用来调用 NdisReadConfiguration，NdisWriteConfiguration，NdisOpenConfigurationByIndex 或 NdisOpenConfigurationByName 的句柄
NdisOpenCanfigurationByIndex	打开一个给定的已打开的注册表中的主键值的一个子键，而这个主键是由一个调用者所提供的句柄指定
NdisOpenConfigarationByName	打开一个给定的已打开的注册表中的主键值的一个指定的子键，这个主键是由一个调用者所提供的句柄指定

NdisReadConfiguration	使用由调用 NdisOpenConfiguration 所获得的参数句柄来读取存储在注册表中的指定键名的键值，键值将传给 NdisReadConfigration
NdisWriteConfiguration	使用由调用 NdisOpenConfiguration 所获得的参数句柄将键值写入注册表，键值传给 NdisWriteConfiguration
NdisCloseConfiguration	关闭一个通过调用 NdisOpenConfiguration 而打开的注册表句柄
NdisMRegisterAdapterShutdownHandler	注册一个 NIC 驱动程序提供的 Miniportshutdown 函数，它在系统将要关闭时被调用
NDIS_INIT_FUNCTION	注明一个驱动程序函数仅在初始化时开始运行
NDIS_PAGEABLE_FUNCTION	注明一个驱动程序函数是可分页代码

2.3.2 NDIS 提供的硬件设置函数

NDIS 库为 EISA，PCI 和 PC 卡提供了与总线相关的函数。这些函数由 MiniportInitialize 调用来获取初始化网络接口卡(NIC)所需的总线相关信息。

下表描述了由 NDIS 提供的硬件设置函数。

函数	定义
NdisReadEisaSlotInformation	读取 EISA NIC 插槽信息并且将它拷贝到 NIC 驱动程序所提供的一个缓存中
NdisReadEisaSlotInformationEx	为一个支持可选择设置的 EISA NIC 读取插槽信息和有关的一系列函数的信息，并且将这些设置拷贝到 NIC 驱动程序所提供的缓存中
NdisImmediateReadPciSlotInformation	从一个指定 PCI 设备的设置区间读取一个指定长度的字节
NdisImmediateWritePciSlotInformation	向一个指定 PCI 设备的设置区间写入一个指定长度的字节
NdisMPciAssignResources	返回一个指定 PCI 设备所声明的资源列表
NdisReadPciSlotInformation	从一个指定 PCI 设备的设置区间读取一个指定长度的字节
NdisWritePciSlotInformation	向一个指定 PCI 设备的设置区间写入一个指定长度的字节
NdisReadPcmciaAttributeMemory	为一个 PC 卡的 NIC 从属性内存中读取与总线相关的设置参数
NdisWritePcmciaAttributeMemory	为一个 PC 卡 NIC 向属性内存写入与总线相关的设置参数
NdisReadNetworkAddress	返回软件可设置的网络地址，它是在 NIC 安装在机器中时为 NIC 存储在注册表中的

2.3.3 NDIS 提供的 I/O 端口函数

NDIS 库提供了一系列微端口网络接口卡(NIC)的驱动程序，可以用来访问 I/O 端口的函数。这些调用提供了一个标准的轻便接口，它支持 NDIS 驱动程序在不同的操作环境下运行。提供的函数用来映射端口，声明 I/O 资源，并且用来对映射和未映射 I/O 端口进行读和写。

下表描述了由 NDIS 提供的 I/O 端口函数

函数	定义
NdisMRegisterIoPortRange	为使用 NdisRawReadPortXxx 和 NdisRawWriteportXxx 函数建立 I/O 访问端口
NdisMDeregisterIoProtRange	删除早期由 NdisMRegisterIoPortRange 注册的 I/O 访问端口
NdisMMapIospace	为随后的内存映射 I/O 操作映射一定范围的设备内存
NdisZeroMappedMemory	将早期调用 NdisMMapIoSpace 映射的内存块以 0 填充
NdisMoveFromMappedMemory	将在初始化时由 NdisMMapIoSpace 所映射的设备内存中的数据拷贝到一个系统空间的缓存中
NdisMoveToMappedMemory	将数据从一个系统空间缓存中拷贝到在初始化时由 NdisMMapIoSpace 所映射的设备内存中
NdisMUnmapIoSpace	释放调用 NdisMMapIoSpace 所映射的区域
NdisImmediateReadSharedMemory	在驱动程序调用 NdisMMapIoSpace 之前从共享内存地址中读取一块数据到缓存
NdisImmediateWriteSharedMemory	在驱动程序调用 NdisMMapIoSpace 之前向一个共享内存地址中写入缓存中的数据
NdisRawReadPortXxx	从一个 I/O 端口读取唯一的一条数据，具体的函数包括：NdisRawReadPortUchor， NdisRawReadportUlong， NdisRawReadportUshort
NdisRawReadPortBufferXxx	从一个 I/O 端口一次读取数据到缓存，具体函数包括：NdisRawReadPortBufferUchar，NdisRawReadPortBufferUlong， NdisRawReadPortBufferUshort
NdisRawWritePortXxx	向一个 I/O 端口写入一条数据，具体函数包括：NdisRawWritePortUchar， NdisRawWritePortUlong， NdisRawWritePortUshourt
NdisRawWritePortBufferXxx	将缓存中的数据写入 I/O 端口，具体函数包括：NdisRawWritePortBufferUch， NdisRawWritePortBufferUlong， NdisRawWritePortBufferUshort
NdisImmediateReadPortXxx	在驱动程序调用 NdisMRegisterIoPortRange 映射端口之前从一个 I/O 端口读取一条数据。具体函数包括：NdisImmediateReadPortUcha， NdisImmediateReadPortUshort， NdisImmeduateReadPortUlong
NdisImmediateWritePortXxx	在驱动程序调用 NdisMRegisterIoPortRange 映射端口之前向一个 I/O 端口写一条数据。具体函数包括：NdisImmediateWritePortUch， NdisImmediateWritePortUshort， NdisImmediateWritePortUlong

2.3.4 NDIS 数据的与 DMA 相关的函数

微端口驱动程序调用 NDIS 函数在主机和 NIC 之间进行直接内存访问(DMA)操作。例如，微端口 NIC 驱动程序为 DMA 传递分配和管理主机中的实际内存缓存。下表总结了由

NDIS 库提供的函数，它允许驱动程序分配，构建和检查这种缓存。

下表描述了由 NDIS 提供的与 DMA 相关的函数

函数	定义
NdisMAllocateMapRegisters	为使用总线管理器的 DMA 设备分配映射注册表
NdisMAllocateShareMemory	分配系统与一个总线管理器 DMA 和 NIC 共享的内存。它在驱动程序初始化时以 IRQL PASSIVE_LEVEL 调用
NdisMAllocateShareMemoryAsync	调用它提升 IRQL，例如一个微端口的 MiniportHandleInterrupt 函数为总线管理器 DMA NIC 分配共享内存
NdisMCompleteBufferPhysicalMapping	释放早期为一个总线管理器 DMA 操作而调用 NdisMStartBufferPhysicalMapping 所使用的映射注册表。仅在 DMA 初始化时由调用 NdisMAllocateMapRegisters 的驱动程序调用
NdisMCompleteDmaTransfer	指示 NDIS 库一个从属的 DMA 操作已完成
NdisMDeregisterDmaChannel	在微端口驱动程序的 DMA 通道上释放它的声明
NdisFlushBuffer	在发送数据到 NIC 或从 NIC 转移数据之前，调用它来确保在总线管理器 DMA 运行的 cache 和主机物理内存之间一致
NdisGetCacheFillSize	返回微处理器的以字节为单位的 cache 边界。DMA NIC 的驱动程序可以使用由这个函数返回的信息以避免在 DMA 传输时与 cache 断开
NdisMFreeMapRegisters	释放总线管理器 DMA 的映射注册表，它是早期由 NdisMAllocateMapRegisters 分配的
NdisMFreeSharedMemory	释放早期由 NdisMAllocateSharedMemory 或 NdisMAllocateSharedMemoryAsync 分配的内存
NdisMInitializeScatterGatherDma	为使用 DMA 驱动程序保留系统资源，仅被非串行化或面向连接微端口调用
NdisMReadDmaCounter	读取系统 DMA 管理器的计数器的当前值
NdisMRegistgerDmaChannel	为将来从属的 DMA 操作建立一个 DMA 通道控制
NdisMSetupDmaTransfer	为从属的 DMA 传递设置一个主机系统的 DMA 控制器
NdisMStarBufferPhysicalMapping	为一个总线管理器 DMA 操作产生一个物理地址映射，仅在 DMA 初始化时调用 NdisMAllocateMapRegisters 的驱动程序调用
NdisMUpdateSharedMemory	确保在总线管理器 DMA 操作时从共享内存区读取的数据是最新的
NdisQueryMapRegisterCount	返回所有可能的映射注册表数量。在调用 NdisMAllocateMapRegisters 之前驱动程序调用 NdisQueryMapRegisterCount 来选择实际分配多少注册表

2.3.5 NDIS 提供的中断处理函数

NDIS 库为多处理器环境提供了几种中断处理函数。这些函数与一个主机处理器上的网络接口卡(NIC)中断相关。

下表描述了由 NDIS 提供的中断处理函数。

函数	定义
NdisMDeregisterInterrupt	是一个中断处理程序停止接收中断。操作系统断开

	NIC 中断相连的中断服务函数的联系
NdisMRegisterInterrupt	连接微端口中断服务函数(MiniportISR)和由它的 NIC 所产生的中断
NdisMSynchronizeWithInterrupt	任何与 MiniportISR 或 MiniportDisableInterrupt 函数共享资源的 NIC 驱动程序函数必须同步访问这些资源以防止争用发生。通过调用 NdisMSynchronizeWithInterrupt 和传递一个 MiniportSynchronizeISR 函数的地址，来使函数与 MiniportISR 和 MiniportDisableInterrupt 同步。MiniportSynchronzeISR 运行在 DIRQL 上，所以它可以安全地访问共享资源

2.3.6 NDIS 提供的同步函数

时钟函数允许微端口实现一个在驱动程序所要求的间隔后发生的操作。例如，驱动程序可以使用时钟轮询它的 NIC。用来在两个线程之间进行同步操作的事件，其中至少要有一个运行在 IRQL PASSIVE_LEVEL 上。自旋锁被用来同步访问共享资源。

下表描述了由 NDIS 库为微端口驱动程序提供的同步函数。

函数	定义
NdisMCanceltimer	取消早期由 NdisMSetTimer 设置的一个时钟
NdisMInitializeTimer	初始化一个时钟对象并且将对象与一个 MiniportTimer 函数相连
NdisMSetTimer	设置一个时钟在一个指定间隔之后停止
NdisMSetPeriodicTimer	设置时钟每隔一个指定时间后停止，或直到调用 NdisMCancelTimer 后才停止
NdisMsleep	引起调用者的线程阻塞指定的间隔。驱动程序在初始化或当停止适配器时，调用 NdisMsleep。例如，当等待 NIC 完成初始化时。NdisMsleep 仅在 IRQL PASSIVEL_LEVEL 下被调用
NdisStallExecution	引起调用者的线程停止一个指定间隔，不超过 50 微秒。此时不能使用 NdisMSleep，NdisStallExecution 仅在升高的 IRQL 下被调用
NdisInitializeEvent	产生和初始化一个用来同步驱动程序操作的事件
NdisSetEvent	为指定事件设置信号量
NdisResetEvent	重新设置指定事件为无信号状态
NdisWaitEvent	引起调用者等待到指定事件被指示或指定时间间隔结束时为止
NdisAllocateSpinLock	初始化一个的 NDIS_SPIN_LOCK 类型变量，它被用来同步访问非 ISR 驱动程序函数之间共享的资源
NdisFreeSpinLock	释放一个在调用 NdisAllocateSpinLock 过程中初始化的自旋锁
NdisAcquireSpinLock	获得一个自旋锁来保护在一个 SMP 安全方式下运行的非 ISR 驱动程序函数之间的共享资源的访问。运行在 IRQL<DISPATCH_LEVEL 下的微端口调用这个函数来获得一个自旋锁
NdisReleaseSpinLock	释放一个早期调用 NdisAcquireSpinLock 获得的自旋锁
NdisDprAcquireSpinLock	获得一个在 IRQL DISPATCH_LEVEL 下的自旋锁。它保护在一个 SMP 安全模式下运行的非 ISR 驱动程序函数间的共享资源访问。它比为运行在 IRQL DISPATCH_LEVEL 上的驱动程序函数调用 NdisAcquireSpinLock 要快

NdisDprReleaseSpinLock	释放一个早期调用 NdisDprAcquireSpinLock 获得的自旋锁
NdisInitializeReadWriteLock	初始化一个 NDIS_RW_LOCK 类型变量。NDIS_RW_LOCK 变量用来限制对一个非 ISR 驱动程序线程的共享资源一次进行一个写访问。这个 NDIS_RW_LOCK 允许多个非 ISR 驱动程序线程同时读这些资源。这个读访问在写访问时是不允许的
NdisAcquireReadWriteLock	获得一个调用者用来在多个驱动程序线程的共享资源间进行写或读访问的锁。运行在 IRQL<DISPAICH_LEVEL 下的微端口调用这个函数来获得一个读—写锁。读—写锁适用经常进行读访问但很少进行写访问的资源
NdisReleaseReadWriteLock	释放一个在调用 NdisAcquireReadWriteLock 过程中获得的读—写锁
NdisMSynchronizeWithInterrupt	任何与 MiniportISR 或 MiniportDisableInterrupt 函数共享资源的 NIC 驱动程序函数必须与这两个函数同步访问资源，以防止争用发生。必须与 MiniportISR 和 MiniportDisableInterrupt 同步的函数通过调用 NdisMSynchronizeWithInterrupt，与一个也运行在 DLRQL 的 MiniportSynchronizeISR 函数同步访问共享资源

2.3.7 NDIS 提供的状态函数

下表总结了由 NDIS 为微端口提供的状态函数。

函数	定义
NdisMCoIndicateStatus	向绑定协议指示一个面向连接 NIC 的状态变化或一个在网络接口卡(NIC)上的指定虚连接(VC)的状态变化
NdisMIndicateStatus	向 NDIS 库指示 NIC 状态已变化，仅被无连接微端口调用
NdisMIndicateStatusComplete	向 NDIS 库指示状态变化已完成，仅被无连接微端口调用
NdisMQueryInformationComplete	指示早期的 MiniportQueryInformation 调用已完成
NdisMSetInformationComplete	指示早期的 MiniportSetInformation 调用已完成

2.3.8 NDIS 为无连接微端口提供的发送和接收函数

无连接介质的微端口或者从它的 MiniportSend 函数发送一个单包或者从它的 MiniportSendPackets 函数发送一组包到网络上。

一个无连结介质的微端口驱动程序调用 NdisMIndicateReceivePacket 向上层传递一个或多个完整的包，或者，微端口调用过滤相关的 NdisMXxxIndicateReceive 函数，强制相关上层驱动程序从一个单一的接收帧中拷贝数据。如果所有的包已作为预留数据传递了，那么它以 NdisMXxxIndicateReceive 方式调用，如果帧比预留缓存大，那么调用它的 MiniportTransferData 函数。

如果 NIC 驱动程序向上指示一个完整的包，NDIS 将调用微端口的 MiniportReturnPacket 函数返回包，下表描述了当发送和接收包时可被一个无连接微端口调用的 NDIS 库函数。

函数	定义
NdisMIndicateReceivePacket	向相关上层指示一个或多个包

NdisMArcIndicateReceive NdisMEthIndicateReceive NdisMFddiIndicateReceive NdisMTrIndicateReceive	指示一个指定介质类型的包正在接收
NdisMArcIndicateReceiveComplete NdisMEthIndicateReceiveComplete NdisMFddiIndicateReceiveComplete NdisMTrIndicateReceiveComplete	指示一个包接收操作已完成
NdisMsendComplete	指示早期包发送操作已完成，当早期 MiniportSend 函数异步操作时使用它
NdisMSendResourcesAvailable	指示 NDIS 库，微端口驱动程序有可用于发送操作的资源。如果微端口完成了一个同步发送，在它的 MiniportSend 函数中调用这个函数，或者当它检测出一个异步发送操作已完成时，在它的 MiniportHandleInterrupt 函数中调用。如果驱动程序没有为一个未完的发送操作调用 NdisMSendComplete 时，NdisMsendResourcesAvailable 才可被调用
NdisMTransferDataComplete	指示 NDIS 库早期的 MiniportTransferDate 请求已完成，当先前调用的 MiniportTransferData 函数返回 NDIS_STATUS_PENDING 时使用这个函数

2.3.9 NDIS 为面向连接微端口提供的发送和接收函数

面向连接微端口在它的 MiniportCoSendPackets 函数中发送一组包到网络上。

面向连接微端口 NIC 通过调用 NdisMCoIndicateReceivePacket 和传递一个完整的包或包组,向上层传递一个或多个接收包。NDIS 调用微端口的 MiniportReturnPacket 函数来返回包。

下表描述了当发送和接收包时，面向连接微端口所调用的 NDIS 库函数。

函数	定义
NdisMCoIndicateReceivePacket	向上层相关层指示一个或多个包，根据每一个 VC 指示包
NdisMCoSendComplete	指示早期通过调用 MiniportCoSendPackets 所发起的包发送操作已完成

2.3.10 NDIS 提供的带外数据宏

微端口可使用以下的宏来访问与一个包相关的带外(OOB)数据。有关 OOB 数据的更多信息，参见 4.3 节。

函数	定义
NDIS_OOB_DATA_FROM_PACKET	返回一个指向与一个给定包描述符相连的 OOB 数据块的指针
NDIS_GET_PACKET_MEDIA_SPECIFIC_INFO	从与一个给定包描述符相连的 OOB 数据块中返回 MediaSpecificInformation 指针和 SizeMediaSpecificInfo 值
NDIS_GET_PACKET_STATUS	从一个与给定包描述符相连的 OOB 数据块中返回状态值
NDIS_GET_PACKET_TI	从一个与给定包描述符相连的 OOB 数据块中返回

ME_TO_SEND	TimeToSend 值
NDIS_SET_PACKET_HEADER_SIZE	为随后的接收指示设置一个与给定包描述符相连的 OOB 数据块中的 HeaderSize 值
NDIS_SET_PACKET_MEDIA_SPECIFIC_LNFO	设置与一个给定包描述符相连的 OOB 数据块中的 MediaSpecificInformation 指针和 SizeMediaSpecificInfo 值
NDIS_SET_PACKET_STATUS	在一个驱动程序调用 NdisM(Co)IndicateReceivePacket 或在一个驱动程序的 Miniport(Co)SendPackets 函数返回控制之前，设置与一个给定包描述符相连的 OOB 数据块中的状态值。
NDIS_SET_PACKET_TIME_RECEIVED	设置与一给定包描述符相连的 OOB 数据块中的 TimeReceived 值
NDIS_SET_PACKET_TIME_SENT	设置与一给定包描述符相连的 OOB 数据块中的 TimeSent 值

2.3.11 NDIS 提供的包和缓存处理函数

微端口驱动程序必须分配缓存来接收内入的数据。向上层相关协议指示所有内入包的、总线管理器 DMA 设备的驱动程序必须为完成这一任务分配和管理包。如果从上层接收到的包是小的和分段的，NIC 驱动程序需要为发送操作分配阶段数据的缓存空间。对于一个发送，所有的微端口 NIC 驱动程序接收到一个完整的包，它必须调用 NDIS 函数来操纵这些包和提取出要发送的数据。一个将接收到的数据向上层驱动程序传递的微端口，从上层接收一个包并且用接收到的数据来填充这个包。NDIS 提供的函数使微端口驱动程序可以用来分配，构建和检测包的函数。

NDIS 把包定义成为一个拥有包含网络包数据的一个或多个缓存链的包描述符。NDIS 通过定义 NDIS_PACKET 结构来描述一个包描述符。描述符包含两部分：微端口驱动程序可见的公共成分和驱动程序不可见的私有成分。公共成分允许一个微端口驱动程序维护有关包的环境信息。微端口驱动程序可以使用这个组件来将包链入到传输队列中。

下表描述了由 NDIS 提供的处理包和缓存的函数。

函数	定义
NdisAllocatePacketPool	分配和初始化一个不分页包池块。调用者提供请求可存放包描述符数和以字节为单位的每个固定包的大小
NdisAllocatePacketPoolEx	分配和初始化一个不分页包池的块。并且提供请求包描述符的数量和以字节为单位的每个固定包的大小，调用者提供用来保存溢出状态下的包描述符的数量
NdisAllocatePacket	从由 NdisAllocatePacketPool 返回的包池中分配一固定大小的包描述符
NdisDprAllocatePacket	当调用者运行在 IRQL DISPATCH_LEVEL 下，分配和初始化一个包描述符
NdisAllocateBufferPool	返回一个调用者可用 NdisAllocateBuffer 来分配缓存描述符的句柄
NdisAllocateBuffer	在指定的已分配的不分页的内存块中创建一个映射虚存范围的缓存描述符。需给出由 NdisAllocateBufferPool 返回的句柄
NdisAdjustBufferLength	修改一个给定缓存描述符指定范围的长度
NdisBufferLength	返回一个给定映射缓存以字节为单位的长度
NdisBufferVirtualAddress	如果给定缓存描述符所给出的物理页并没有映射到系统空间的话,返回由一个给定缓存描述符映射的缓存的基本虚地址，并且将这物理页映射到系统空间，

NdisCopyBuffer	为指定的内存块创建缓存描述符
NdisCopyFromPacketToPacket	从一个包向另一个包拷贝指定数量的字节，给定一个指定的源包和一个目的包及每个包的起始位置
NdisCreateLookaheadBufferFromSharedMemory	返回一个与总线管理器 DMA NIC 共享内存块的缓存的虚地址。允许驱动程序以只读预分缓存方式将向上指示的接收数据的一部分映射到相关协议
NdisDestroyLookaheadBufferFromSharedMemory	释放由调用 NdisCreateLookaheadBufferFromSharedMemory 而获得的缓存
NdisQueryPacket	返回一组描述了包和链的大指针的信息
NdisQueryBuffer	返回由一个给定缓存描述符映射的一个缓存的虚地址的基址和长度
NdisQueryBufferOffset	返回在一个给定缓存描述符中映射的一个缓存的虚地址的基址和长度
NdisGetFirstBufferFromPacket	返回链在包上的第一个缓存的描述符及虚地址，以及第一个缓存的长度和所有缓存的总长（以防缓存使分段的）
NdisGetNextBuffer	返回给定缓存描述符链当前的缓存描述符
NdisGetBufferPhysicalArraySize	对一给定缓存描述符，返回不邻接的物理块的缓存数量。总线管理 DMA NIC 在调用 NdisMStarBufferPhysicalMapping 之前调用它来决定分配和填充多少 NDIS_PHYSICAL_ADDRESS 结构
NdisGetPacketFlags	如果有的话，返回在一个给定包中，由协议驱动程序设置的标志
NdisFreePacketPool	释放由调用 NdisAllocatePacketPool 所分配的一个不分页池的块
NdisPacketPoolUsage	返回一个包池中当前已分配的包描述符的数量
NdisFreePacket	释放调用 NdisAllocatePacket 所分配的包
NdisDprFreePacket	当调用者为访问包池提供内部同步，并且调用者运行在 IRQL DISPATCH_LEVEL 时，释放驱动程序分配的包描述符并且将它返回给空闲列表
NdisFreeBufferPool	释放调用 NdisAllocateBufferPool 所获得的句柄
NdisFreeBuffer	释放调用 NdisAllocateBuffer 所获得的缓存描述符
NdisChainBufferAtBack	将一给定缓存描述符链到一个给定包的缓存描述符链表的末尾
NdisUnchainBufferAtFront	从一给定包的缓存描述符链表中删除链表的表头并且返回指向这个缓存描述符的指针
NdisUnchainBufferAtBack	从一给定包的缓存描述符链表中删除表尾，并且返回指向这缓存描述符的指针
NdisReinitializePacket	从一给定包中删除所有相链的缓存并且重新初始化它以备重新使用
NdisRecalculatePacketCounts	重新设置多个给定包的有效个数以使下一个调用 NdisQueryPacket 重新计算个数
NDIS_BUFFER_LINKAGE	由一指向缓存描述符的指针，返回一个被链接的缓存的指针。这个宏允许一个驱动程序不必提供自己的缓存描述符链接情况而对已分配缓存描述符排队
NDIS_BUFFER_TO_SPAN_PAGES	确定给定的缓存取要使用多少物理页

2.3.12 NDIS 提供的支持函数

以下的 NdisXxx 函数类提供了支持微端口的操作：

- 内存函数
- 查询函数
- 系统信息函数
- 日志函数
- 字符串函数
- 文件函数
- 地址函数
- 变量函数
- 注册函数
- 工作项目函数

内存函数

函数	定义
NdisAllocateMemory	在指定地址范围地将驻留(不分页)的系统空间内存以物理连续分配方式分配和/或以非 cache 方式分配
NdisAllocateMemorywithTag	如同调用 NdisAllocateMemory，不同在于它允许调用者提供一个标记可用来跟踪驱动程序的内存分配
NdisEqualmemory	将一个内存块中的指定数量的字符与另一个内存块中相同数量的字符相比较
NdisFillMemory	用给定字符填充一个调用者所提供的缓存
NdisFreeMemory	释放早期由 NdisAllocateMemory 分配的内存块
NdisMoveMemory	从一调用者提供的位置将一指定数量的字节拷贝到另一地方
NdisZeroMeomory	用 0 填充内存块
NdisInitializeNPagedLookasideList	初始化一个预先分配的列表，在初始化成功以后，可以从预先分配列表中分配或向列表释放不分页的固定大小的块
NdisAllocateFromNPagedLookasideList	从一给定的预先分配的列表头中删除第一个项目。如果预先分配列表是空的，那么从不分页池中分配一个项目
NdisDeleteNPagedLookasideList	将一不分页预分配列表从系统中删除
NdisFreeToNPagedLookasideList	返回预分配列表的入口
NdisInitializeListHead	初始化一个顺序的，互锁的，单链表的表头
NdisInterlockedInsertTailList	插入一个项目，通常是一个包，到一个双链表的表尾，这使得可以在一个多处理器安全方式下同步访问列表
NdisInterlockedRemoveHeadList	从表头删除一个项目，这使得可以在一个多处理器安全方式下同步访问列表
NdisInitializeslistHead	初始化一个顺序的，互锁的，单链表的表头
NdisInterlockedPopEntrysList	从一个顺序的，单链表中删除第一个项目
NdisInterlockedPushEntrysList	在一个顺序的，单链表表头插入一个项目
NdisQueryDepthsList	返回一个给定的顺序的，单链表中当前项目数目

系统信息函数

函数	定义
NdisSystemProcessorCount	确定它的调用者运行在一个单处理器上还是多处理机上。一个 NDIS 驱动程序在分配资源之前的初始化中，调用这个函数
NdisGetCurrentProcessorCounts	返回驱动程序当前可使用的处理器的个数，驱动程序用它来确定 CPU 的利用情况
NdisGetCurrentProcessorCpuUsage	以百分比方式返回当前处理器使用率
NdisGetCurrentSystemTime	返回当前系统时间，用来设置时间戳
NdisGetSystemUptime	返回自从系统开始以来所经过的时间值，以微秒为单位

日志函数

函数	定义
NdisMCreateLog	分配和打开一个 NIC 微端口可写入数据的日志文件，这个文件由一个驱动程序提供的 Win32 应用程序显示
NdisMWriteLogData	为供驱动程序提供的 Win32 应用程序使用和显示日志，而将驱动程序提供的信息写入日志文件
NdisMFlushLog	清除日志文件
NdisMCloseLog	释放日志所使用的资源
NdisWriteErroLogEntry	向系统 I/O 出错日志文件写一条目

字符串函数

函数	定义
NdisAnsiStringToUnicodeString	将一给定数量 ANSI 字符串转化为一定数量的 Unicode 字符串
NdisEqualAnsiString	将两个 ANSI 字符串比较并且返回一个表明它们是否相等的值
NdisEqualString	在操作系统缺省的字符集下，比较两个字符串是否相等
NdisEqualUnicodeString	比较两个 Unicode 字符串并且返回一个表明它们是否相等的值
NdisFreeString	释放为缓存字符串分配的存储区
NdisInitAnsiString	初始化一 ANSI 字符串
NdisInitializeString	为一长度字符串分配存储区，并且以系统缺省的字符集初始化它
NdisInitUnicodeString	初始化一 Unicode 字符串
NdisPrintString	在调试窗口显示指定的字符串
NdisUnicodeStringToAnsiString	将一给定的 Unicode 字符串转换成 ANSI 字符串
NdisUpcaseUnicodeString	将一给定 Unicode 字符串副本转换为大写形式，并且返回转换后的字符串

文件函数

函数	定义
NdisOpenFile	返回一个已打开文件的句柄
NdisMapFile	如果文件未被映射，将一个已打开文件映射到一个调

	用者可访问的缓存中
NdisUnmapFile	释放一个由 NdisMapFile 建立的文件的虚地址映射
NdisCloseFile	释放由 NdisOpenFile 返回的句柄，并且释放文件打开时所分配给文件 contex 的内存

地址函数

函数	定义
NdisGetPhysicalAddressHigh	返回一个给定物理地址的高位部分
NdisGetPhysicalAddressLow	返回一个给定物理地址的低位部分
NdisSetPhysicalAddressHigh	将一个给定物理地址的高位部分设置为一个给定的值
NdisSetPhysicalAddressLow	将一个给定物理地址的低位部分设置为一个给定的值
NDIS_PHYSICAL_ADDRESS_CONST	初始化一个物理地址的静态常量类型

变量函数

函数	定义
NdisRetriveUlong	从源地址中取回一个 ULONG 值，避免对齐错误
NdisStoreUlong	存储一 ULONG 值到一指定地址，避免对齐错误
NdisInterlockedAddUlong	用一个原子操作将一个 Unsigned long 值与一个给定 Unsigned integer 相加，使用一个调用者提供的自旋锁来同步访问 integer 变量
NdisInterlockedDecrement	用一个原子操作减一
NdisInterlockedIncrement	用一个原子操作加一

注册函数

函数	定义
NdisReadRegisterUchar	从一个内存映射设备注册表中读取一个 UCHAR
NdisReadRegisterUlong	从一个内存映射设备注册表中读取一个 ULONG
NdisReadRegiseerUshort	从一个内存映射设备注册表中读取一个 USHORT
NdisWriteRegisterUchar	向一个内存映射设备写入一个 UCHAR
NdisWriteRegisterUlong	向一个内存映射设备写入一个 ULONG
NdisWriteRegisterUshort	向一个内存映射设备写入一个 USHORT

工作项目函数

函数	定义
NdisInitializeWorkItem	当一个系统工作线程获得控制时，用一个调用者提供的环境和回调例程，初始化一个工作队列来作为执行排队
NdisScheduleWorkItem	将一个给定工作项目插入到一个队列。一个系统工作线程删除项目，并且将控制给予驱动程序先前提供给 NdisInitializeWorkItem 的回调函数

2.3.13 NDIS 提供的媒体相关宏

一个微端口可使用这些宏来持行媒体相关操作。

函数	定义

ETH_COPY_NETWORK_ADDRESS	将一个给定 Ethernet 地址拷贝到一个给定位置
FDDI_IS_BROADCAST	将一个调用者提供的变量设置为一个布尔值，它表示一个给定的 FDDI 地址是否是一个广播地址
FDDI_IS_MULTICAST	将一个调用者提供的变量设置为一个布尔值，它表示一个给定的 FDDI 地址是否是多点传输地址
FDDI_IS_SMT	设置一个调用者提供的变量为一布尔值，它表示是否给定的 FDDI 帧是 SMT
TR_COMPARE_NETWORK_ADDRESS	将一个调用者提供的变量设置为一个布尔值，表明一个给定的 Token Ring 地址是大于、小于或等于另一个给定 Token Ring 地址的值
TR_COPY_NETWORK_ADDRESSES	将一个给定 Token Ring 地址拷贝到一给定位置
IR_IS_BROADCAST	将一个调用者提供的变量设置为一个布尔值，它表示一个给定的 Token Ring 地址是否是广播地址
TR_IS_FUNCTIONAL	将一个调用者提供的变量设置为一个布尔值，它表示一个给定 Token Ring 地址是否是函数地址
TR_IS_GROUP	将一个调用者提供的变量设置为一个布尔值，它表示一个给定 Token Ring 地址是否是一组地址
TR_IS_NOT_DIRECTED	将一个调用者提供的变量设置为一个布尔值，它表示一给定 Token Ring 地址是否既不是一个函数地址也不是一组地址
TR_IS_SOURCE_ROUTING	将一个调用者提供的变量设置为一个布尔值，它表示一给定 Token Ring 地址是否是一源路由地址

第三章 NIC 微端口驱动程序入口点和初始化

这一章讲述了微端口的 DriverEntry 函数和 MiniportInitialize 函数。任何 NDIS 微端口的驱动程序函数都需要它们。

这一章的内容包括：

•在 3.1 节讲述 DriverEntry 函数。在 DriverEntry 环境函数中，一个微端口将自身与 NDIS 相连，指定它所使用的 NDIS 版本，并且注册它的入口指针。

• 在 3.2 节讲述 MiniportInitialize 函数，在它的 MiniportInitialize 环境函数中，一个微端口注册它所管理的每个 NIC，声明系统资源(例如包，缓存，和 I/O 端口)，并且有可能地注册一个中断，关闭处理程序，和/或初始化轮询时钟。

3.1 NDIS 微端口驱动程序入口函数

每个 NIC 微端口驱动程序必须提供一个 DriverEntry 的函数。DriverEntry 由系统调用来装载驱动程序。DriverEntry 产生一个微端口 NIC 驱动程序和 NDIS 库的连接，并且向 NDIS 注册微端口的版本和入口指针。

（图 3.1 初始化 NDIS 库和注册一个微端口）

DriverEntry 函数的声明：

```
NTSTAUS
  DriverEntry {
    IN   PDRIVER_OBJECT   DriverObject，
    IN   PUNICODE_STRING   RegistryPath
    };
```

两个参数传递给 DriverEntry：

• 一个指向驱动程序对象的指针，它由 I/O 系统产生

• 一个指向注册表路径的指针，它讲明了指定驱动程序参数贮存的位置

正如图 3.1 所示，DriverEntry 调用 NdisMInitializeWrapper，然后调用 NdisMRegisterMiniport。DriverEntry 传递两个指针给 NdisMInitializeWrappper，它返回一个包裹句柄。DriverEntry 传递给 NdisMRegisterMiniport。

注册表包含系统重起时永久存在的数据，它同每次系统重起时重新产生的设置信息一样。在驱动程序安装时，描述驱动程序和 NIC 的数据存储在注册表，注册表包含 NIC 驱动程序可读取的，用来初始化自身和 NIC 的适配器的特性，参见《Plug and Play，Power Management，and Setup Design Guide》来获取更多有关注册信息、安装驱动程序和写注册表的 INF 文件信息。有关网络组件 INF 文件的更多信息参见第五章。

3.1.1 初始化包裹

DriverEntry 函数通过调用 NdisMInitializeWrapper 函数，（这个函数同调用 DriverEntry 时使用同样的参数）来使得微端口 NIC 驱动程序和 NDIS 相联系。这些参数是指向驱动程序对象的指针和指向注册表路径的指针。NdisMInitializeWrapper 分配一个结构来代表这个联

系，来存储 NDIS 库所需的微端口相关的信息，并且返回 NdisWrapperHandle，它是代表这个微端口 NIC 驱动程序结构的句柄。当驱动程序注册它的入口指针时，它必须保留和传递这个句柄到 NdisMRegisterMiniport。NDIS 将使用 NdisWrapperHandle 来辩别微端口，微端口必须保留这个句柄，但它不应试图去访问或解释这个句柄。

3.1.2 注册微端口

在 DriverEntry 函数调用 NdisMInitializeWrapper 产生与 NDIS 库的一个连接后，它必须初始化一个 NDIS_MINIPORT_CHARACTERISTICS 类型的结构。NDIS_MINIPORT_CHARACTERISTICS 结构指定了与微端口(见 3.1.2.1 节)兼容的 NDIS 版本以及要求的入口点和由微端口提供的可选上层函数(MiniportXxx) (见 3.1.2.2 节)。然后 DriverEntry 通过一个指向 NDIS_MINIPORT_CHARACTERISTICS 结构的指针调用 NdisMRegisterMiniport。

3.1.2.1 指定 NDIS 版本号

NDIS_MINIPORT_CHARACTERISTICS 结构中的 MajorNdisVersion 和 MinorNdisVersion 成员指定了与微端口兼容的 NDIS 版本。有效的 NDIS 版本号是 5.0，4.0，或 3.0。最新版本是 5.0。NDIS 继续支持早期为 4.0 和 3.0 版编写的驱动程序。

因此 NDIS 版本号，包括包含在 NDIS_MINIPORT_CHARACTERISTICS 中的，都必须在微端口源代码编释时指定。，下面显示的是驱动程序使用 DIS_MINIPORT_CHARACTERISTICS 5.0 版本条件下的包含在微端口源代码中的编译指令。

```
 #define NDIS_MINIPORT_DRIVER
#define NDISS0_MINIPORT     1
#include <ndis.h>      // AFTER Preceding directives
```

下面指令指定 NDIS 驱动程序版本：

```
#define NDISXX_MINIPORT
```

XX 是 NDIS 版本号。

这个定义语句必须在#include <ndis.h>语句之前。

除了使用预编译指令，同样的驱动程序可以在它的源文件中设置，以下构建指令：

```
...
C_FLAGS=$(C_FLAGS)-DNDIS_MINIPORT_DRIVER-DNDIS50_MINIPORT_DRIVER
...
```

NDIS_MINIPORT_CHARACTERISTICS 结构中指定的版本必须与编译器或为微端口设置的构建指令的版本一致。

3.1.2.2 注册 MiniportXxx 函数

NDIS_MINIPORT_CHARACTERISTICS 结构中每个地址成员必须被初始化——也就是说，它必须被设置成微端口提供的函数地址或 NULL。微端口必须为所有 MiniportXxx 函数提供入口指针;否则 NDIS 将不允许装载驱动程序。MiniportXxx 函数在 ndis.h 头文件中定义。

因为微端口提供 MiniportXxx 函数的地址，而不是名字，驱动程序开发者可以自由的以他们喜好命名。一个合理的微端口开发策略是给这函数命名以使调试容易。例如，微端口 NIC 驱动程序 mp.sys 命名入口指针为 MPISR，MpSend 等。

一些 MiniportXxx 函数是可选择的。例如，如果一个 NIC 不产生中断，微端口必须轮询

它的 NIC。这种微端口必须有一个 MiniportTimer 函数，但不需要 MiniportISR 函数、MiniportEnableInterrupt 函数或 MiniportDisableInterrupt 函数。

注册无连接微端口的 MiniportXxx 函数

无连接微端口提供的 MiniportXxx 函数在 2.1.1 节描述。

一个无连接微端口必须注册以下 MiniportXxx 函数：

- MiniportHalt
- MiniportInitialize
- MiniportQueryInformation
- MiniportReset
- MiniportReturnPacket 或 MiniportTransferData

如果 LAN 微端口通过调用 NdisMindicateReceivePacket 或 NdisMCoIndicateReceivePacket 支持多包接收的指示，并总是向上层驱动程序指示完整的网络包，那么它必须提供一个 MiniportReturnPacket 函数。如果一个 LAN 微端口在预分配缓存中指示接收网络包的初始部分，这个预分配缓存是相关上层拷贝接收数据的地方，那么它必须提供一个 MiniportTransferData 函数。WAN 微端口不需要提供了一个 MiniportReturnPacket 函数或一个 MiniportTransferData 函数。

- MiniportSend 或 MiniportSendPackets

LAN 微端口必须提供了一个 SendPacketsHandler 或一个 SendHandler。一个 WAN 微端口必须有一个 WanSendHandler。

- MiniportSetInformation

无连接微端口也可（在某些情况下必须）注册以下 MiniportXxx 函数：

- MiniportAllocateCommplete

如果微端口控制一个总线管理器 DMA NIC，并且用 NdisMAllocateSharedMemoryAsync 在升值的 IRQL 下分配内存，则需要这个函数。

- MiniportISR

如果由微端口管理的 NIC 不产生中断，微端口用微端口提供的时钟函数轮询 NIC 状态，或在其他条件下应用时，则不需要这个函数。更多信息参见 3.1.2.3 节。

- MiniportHandleInterrupt

如果由微端口管理的一个 NIC 产生中断，则需要这个函数。

- MiniportEnableInterrupt 和 MiniportDisableInterrupt

如果由微端口管理的 NIC 不产生中断，则不需要这个函数。

- MiniportCheckForHang

如果没有提供，微端口依靠 NDIS 来判断 NIC 是否被挂起，依据发送和请求的超时。

一个无连接微端口不能提供以下 MiniportXxx 函数：

- MiniportCoActivateVc
- MiniportCoCreateVc
- MiniportCoDeactivateVc
- MiniportCoDeleteVc
- MiniportCoRequest
- MiniportCoSendPackets

注册面向连接微端口的 MiniportXxx 函数

由一个面向连接微端口提供的 MiniportXxx 函数已在第二章中讲述了。面向连接的微端口必须注册以下 MiniportXxx 函数：

• MiniportCoActivateVc
• MiniportCoCreateVc
• MiniportCoDeactivateVc
• MiniportCoDeleteVc
• MiniportCoRequest
• MiniportCoSendPackets
• MiniportHalt
• MiniportInitialize
• MiniportReset
• MiniportReturnPacket

一个面向连接微端口可以，并且在某些情况下必须提供以下 MiniportXxx 函数：

• MiniportAllocateComplete

如果微端口控制一个总线管理器 DMA NIC，并且用 NdisMAllocateSharedMemoryAsync 在升值的 IRQL 下分配内存，则需要这个函数。

• MiniportISR

如果不由微端口管理的 NIC 产生中断，微端口用其提供的时钟函数轮询 NCI 状态，或其他条件下应用时，则不需要这个函数。更多信息参见 3.1.2.3 节。

• MiniportHandleInterrupt

如果由微端口管理的一个 NIC 产生中断，则需要这个函数。

• MiniportEnableInterrupt 和 MiniportDisableInterrupt

如果由微端口管理的 NIC 不产生中断，则不需要这个函数。

• MiniportCheckForHang

如果没有提供这个函数，微端口依据发送和请求的超时依靠 NDIS 来判断 NIC 是否被挂起。

一个面向连接的微端口不提供以下 MiniportXxx 函数：

• MiniportQueryInformation
• MiniportSetInformation
• MiniportSend
• MiniportSendPackers
• MiniportTransferData

下节讲述了注册可选 MiniportXxx 函数时的选择标准。

3.1.2.3 为中断支持的注册处理程序

这一节讲述了如何为微端口管理的一个 NIC 产生的中断，选择中断处理程序。有关中断处理的细节见 4.1 节。

当一个 NIC 中断时，NDIS 产生中断；即：当 NIC 中断时，NDIS 总是第一个被调用，NDIS 判断中断属于哪一个微端口，以及是应当调用微端口的 MiniportISR 函数，还是应当自己为微端口处理中断（如果中断是非共享的，且微端口在调用 NdisMRegisterInterrupt 时指示 NDIS 不能调用它的 MiniportISR 函数）。如果 NDIS 没有调用微端口 ISR 函数，NDIS 不能通过调用微端口的 MiniportDisableInterrupt 函数禁止进一步产生中断，然后对微端口的 MiniportHandleInterrupt DPC 函数进行排队。

如果微端口要处理它的设备中断，它必须在一个 NDIS_MINIPORT_CHARACTERISTICS 结构中注册它的 ISR 处理程序，并且通过将 NdisMRegisterInterrupe 的参数 RequestISR 设置为 TRUE 表明可以调用它的 ISR 处理程序。

注册一个 ISR 句柄

如果以下情况为真时，产生中断的 NIC 的微端口必须注册一个 ISR 处理程序。

- 微端口管理了一个在微端口初始化或微端口停止操作时能产生中断的 NIC
- 微端口的 NIC 与其他设备共享一个 IRQ
- 微端口必须在 DIRQL 下从 NIC 注册表获取中断相关数据

当调用 NdisMRegisterInterrupt 时，这种微端口必须也将 RequestIsr 设置为 TRUE。如果中断是共享的，当调用 NdisMRegisterInterrupt 时，必须设置 SharedInterrupt 为 FALSE。

如果上述情况都不成立，产生中断的 NIC 的微端口通过将 RequestISR 和 ShareInterrupt 置为 FALSE，调用 NdisMRegistInterrupt 指定由 NDIS 处理中断。

如果一个微端口管理的 NIC 不产生中断，微端口通常会用一个时钟来轮询 NIC，因此就不需要注册 ISR 处理程序。这种类型的驱动程序必须在 MiniportInitialize 函数中初始化必须的钟。参见 3.2.5 节。

注册中断处理程序

如果微端口管理的 NIC 产生中断，它必须注册一个 HandleInterrupt 句柄。既使 NDIS 完全处理中断，它也需要对微端口的 DPC 进行排队，引起微端口的 MiniportHandleInterrupt 句柄运行。

允许和不允许中断

如果 NIC 不产生中断，微端口不需要提供一个 MiniportEnableInterrupt 或 MiniportDisbleInterrupt 函数。如果微端口总是在 MiniportISR 函数中处理它的中断，并且调度 MiniportHandleInterrupt 函数，那么它不需要提供特定的 MiniportEnableInterrupt 或 MiniportDisableInterrupt 函数。这种微端口可以在它的 ISR 中选择中断，在它的 DPC 中选择允许中断，此时不需要提供特定的驱动程序函数完成这个任务。如果一个微端口指明 NDIS 处理它的中断，它必须提供一个 MiniportDisableInterrupt 函数，如果微端口不在它的 MiniportHandleInterrupt 函数中重新允许中断，它必须有 MiniportEnableInterrupt 函数。

3.1.2.4 为无连接微端口选择一个发送函数

无连接介质微端口 NIC 驱动程序可用以下两种方式处理发送：

- 提供一个 MiniportSendPackets 函数
- 提供一个 MiniportSend 函数

MiniportSendPackets 在一个呼叫中可以输入多个数据包。NDIS 以一组指向包描述符的指针的方式将包传递给 MiniportSendPackets。然而 MiniportSend 在一次呼叫中仅输入一个包。

基本上，如果微端口能同时向它的 NIC 进行多包发送，那么它提供了一个 MiniportSendPackets 函数。例如，一个管理着内置环的总线管理器的 DMA NIC 的微端口通常提供了一个 MiniportSendPackets 函数，对于这种 NIC 微端口可以用许多包建立一个环，从而在一次操作中发送它们。

即使 NIC 一次发送仅接收一个包，也可以通过在由 MiniportSendPackets 接收到的包中进行排队来提高性能。这种策略避免了早期重复调用 MiniportSend。

如果微端口 NIC 驱动程序在 NDIS_MINIPORT_CHARACTERISTICS 中为 MiniportSend 和 MiniportSendPackets 都提供了入口指针，那么 NDIS 总是调用 MiniportSendPackets 向微端口传递发送请求。

3.1.2.5 为无连接微端口选择接收函数

无连接微端口向上层传递接收包的方式决定了它是否必须注册一个 MiniportReturnPackets 函数或 MiniportTransferData 函数：

• 如果微端口通过调用 NdisMIdicateReceivePacket 来向上层指示包，那么它必须注册一个 MiniportReturnPacket 函数。

• 如果微端口通过调用 NdisMXxxIndicateReceive 函数来向上层指示包，那么，它必须注册一个 MiniportTransportData 函数，有关微端口接收操作参见 4.6 节。

3.1.2.6 注册一个分配完成处理程序

如果总线管理器 DMA NIC 微端口调用 NdisMAllocateShardMemoryAsync 来分配缓存空间，那么总线管理器 DMA NIC 微端口必须提供一个 MiniportAllocatelCOmplete 函数，。可以在升高 IRQL 的情况下调用 NdisMAllocateShardMemoryAsync—例如，从一个驱动程序的 MiniportHandleInterrupt 函数中。当一个驱动程序缺乏缓存时，可以调用 NdisMAllocateSharedMemoryAsync，而不是强制驱动程序分配在微端口初始化时分配的缓存空间。通常只有调用 NdisMIndicateReceivePacket 的微端口有这种需求。

由于向上指示完整包的微端口(典型的带 ring buffer 的 DMA 设备的微端口)放弃了向上层指示包的所有权，因此，它依靠上层迅速返回包。如果因为包不能迅速返回，微端口为接收数据导致缺乏缓存，那么它可以在升值的 IRQL 下调用 NdisMAllocateSharedMemoryAsync，在它的 MiniportHandleInterrapt 函数中分配一个缓存。NDIS 当异步分配完成时，它调用微端口的 MiniportAllocateComplete 函数。

3.1.2.7 注册一个挂起检测(CheckForHang) 处理程序

当微端口调用 NdisMRegisterMiniport 时，它可以有选择地提供 MiniportCheckHang 函数。如果一个微端口没有注册这个处理程序，它将依赖 NDIS 来检测 NIC 的挂起。

如果微端口提供了 MiniportCheckForHang 函数，NDIS 每两秒或以驱动程序要求的超时间隔调用这个函数。MiniportCheckForHang 函数判断 NIC 是否被挂起。如果它检测 NIC 挂起，那么 MiniportCheckForHang 返回 TRUE;否则，它返回 FALSE。

如果微端口没有注册 MiniportCheckForHang 函数，那么当下列情况出现时，NDIS 判定微端口被挂起：

• 一个发送包延缓了两倍超时周期。例如，向微端口传递一个包用来发送，它返回 NDIS_STATUS_PENDING，且在两倍超时时间内没有调用 NdisMSendComplete 函数。这仅对串行微端口适用。

• 一个向 MiniportQueryInformation 或 MiniportSetInformation 的发送请求在两倍超时时间内还没有完成。

一个微端口可以通过调用 NdisMSetAttributesEx 来改变超时参数，而不是用 MiniportInitialize 函数(见 3.2.1.3 节)中的 NdisMSetAtrributes。通过 NisMSetAttributeEx，微端口可以：

• 将缺省的 2 秒超时间隔改变为一个驱动程序指定的间隔。NDIS 将以这个新的间隔调用微端口的 MiniportCheckForHang 函数。慢速 NIC 的微端口可以指定适合 NIC 的时间超时间隔。

• 指定 NDIS 忽略包超时。

• 指定 NDIS 忽略请求超时。

如果微端口指定 NDIS 忽略包时间结束和请求时间结束，当 NIC 被挂起时，微端口负责检测 NIC 是否被挂起。

如果微端口 CheckForHang 返回 TRUE，或一个发送或请求已经超过了 2 倍超时间隔，NDIS 调用 MiniportReset。

3.2 NDIS 微端口初始化

当一个微端口 DriverEntry 函数返回时，NDIS 立即为每个微端口管理的 NIC，调用微端口的 MiniportInitialize 函数。

如果一个新的 NIC 插入到系统，将调用微端口的 MiniportInitialize 函数来初始化新安装的设备。因此， MiniportInitialize 函数和其他调用函数，以及所有运行在 IRQL PASSIVE_LEVEL 的函数都可被指定为可分页的。通过使用 NDIS_PAGABLE_FUNCTTON 宏来指定代码为可分页的。初始化代码不能用 NDIS_INIT_FUNCTION 宏来指定，这是由于标记这种方式的代码在初始化系统的启动结束之际，已不再映射了。仅有 DriverEntry 函数和从 DrireEntry 调用的函数可以传递到 NDIS_INTI_FUNCTLON 宏，有关指定代码为可分页的或仅用来初始化的代码的更多信息，参见《Driver Write's Guide》。

初始化函数的声明如下所示：

```
NDIS_STATUS
  MiniportInitialize {
    OUT   PNDIS_STATUS   OpenErrorStatus，
    OUT   PUINT          SelectMediumIndex，
    IN    PNDIS_MEDIUM   MediumArry，
    IN    UINT           MediumArraySize，
    IN    NDIS_HANDLE    MiniportAdapterHandle，
    IN    NDIS_HANDLE    ConfigurationHandle
  };
```

以下是传递给 MiniportInitialize 的参数。

•一组在 MediumArray 中的 NdisMediumXxx。微端口必须选择它所支持或兼容的介质，必须返回在 SelectedMediumIndex 中介质的索引。如果微端口没有在 MediumArray 中发现它所支持的介质，将返回一个失败状态 NDIS_STATUS_UNSUPPORTED_MEDIA。

• MiniportAdapterHandle，一个可被 NDIS 引用的参照微端口句柄。微端口必须保留这个句柄，以使它在以后的调用中可传递给 NDIS。——例如，在 NdisMRegisterAdapterShutdownHandler 和 NdisMInitializeTimer 调用中。

•一个配置句柄，它确定与这个微端口相连的指定 NIC 信息的注册表键。微端口必须保留和传递这个句柄给 NdisOpenConfiguration，来打开描述微端口 NIC 的注册表的 Parameters 子键。有关网络 INF 文件注册表中参数值的信息，参见第一部分。

实验介质类型

NDIS 介质类型和相关标识符包含在注册表的 HKEY_LOCAL_MACHINE \ SOFTWARE \ Microsoft \ NDIS \ MediaTypes 键中。这个键包含一个当前系统定义介质类型的列表。只有在此键中存在的介质，它才可在 MediumArray 中传递给 MiniportInitialize。为了注明一个不支持的实验介质类型，驱动程序编写者可以添加实验介质类型到第一部分第三章所讲述的注册表中。

3.2.1 注册一个 NIC

在微端口的 MiniportInitialize 函数中，微端口必须注册它的 NIC。此过程包括有以下主要的三步：

1.为适配器指定的环境区域分配内存，该区域典型地保存着 NIC 状态。

2.微端口在适配器指定的环境区域读取配置信息和存储这个信息。

3.微端口通过 NDIS 注册 NIC。

（图 3.2 注册一个网络接口卡）

3.2.1.1 分配一个适配器指定的环境区域

在微端口的 MiniportInitialize 函数中，微端口通常为驱动程序内部结构分配内存，在这个结构中存储有适配器指定的如下的环境：

- 向 MiniportInitialize 提供 NDIS 提供的的微端口适配器句柄
- 从调用 NdisReadConfiguration 得到的的配置信息
- 微端口所需要的其他状态信息

例如，一个在 MiniportInitialize 函数中为接收数据分配阶段缓存的驱动程序，可以将缓存与这个结构相连接。如果驱动程序对发送排队，那么这个队列的表头可以是这个结构的一部分。

一个微端口调用 NdisAllocateMemory 或 NdisAllocateMemoryWithTag，来为这个结构分配存储区。微端口在初始化它之前调用 NdisZeroMemory，清空这个存储区。然后，微端口传递一个指向这个结构的指针到 NdisMSetAttributes 或 NdisMsetAttributesEx，NDIS 保留指向这个与适配器有关的环境的指针，并且将它传递给微端口的 MiniportXxx 函数——例如，MiniportSend，MiniportISR 和 MiniportHandleInterrupt。微端口使用与适配器有关的环境，来保存所有与适配器有关的运行时状态，以处理发送，接收和请求。

3.2.1.2 读取配置信息

在分配和-初始化一个用来保存与适配器有关信息的结构后，微端口必须为它的 NIC 获得配置信息。在微端口 MiniportInitialize 环境函数中，微端口以以下方式为 NIC 获得配置信息：

1.微端口调用 NdisOpenConfigration 来打开和获得一个 Parameters 注册表键的句柄，这个注册表键包含有由安装脚本所提供的配置信息。

2.微端口调用 NdisReadConfiguration 来读取在 Parameters 键中与设备有关值的项目。微端口调用 NdisReadConfiguration 为每一个关键字检索相关的值，NdisReadConfiguration 发现与关键字匹配的值的项目名并且向微端口返回 NDIS_STATUS_SUCCESS。

3. 微端口可以调用 NdisOpenConfigurationKeyByIndex 或 NdisOpenConfigurationKeyByName 来打开和获得一个存储在子键中的信息的句柄。这个子键是在由 NdisOpenConfigurtion 打开的 Parameters 键的下部，然后，微端口调用 NdisReadConfiguration 来读取存储在已打开键的值。

4 如果与适配器相连的 I/O 总线类型中存在总线类型相关函数，那么微端口用它来获得总线类型指定信息。总线类型指定函数在 2.3.2 节列出。例如，如果总线类型是 EISA，微端口调用 NdisReadEisaSlotIntormation 来获得信息，例如适配器主要和次要的修正版本方式有关 IRQ 的信息。

5.如果 NIC 是可编程的，微端口调用 NdisReadNetWorkAddress 从注册表中读取 NIC 的网络地址随后将地址写入到适配器的 EPROM 中。

6.在微端口获得它向注册表所请求的值后，它调用 NdislloseConfiguration，当 NdisCloseConfiguration 返回时，注册表链句柄不再有效。

3.2.1.3 注册 NIC

在微端口读取它所需要的描述有关 NIC 特征的信息，并且按要求方式将此信息存储在它的微端口适配器环境中之后，微端口调用 NdisMSetAttributes 或 NdisMSetAttributeEx。在这个调用中，微端口作如下工作：

• 传递由 NDIS 提供的微端口适配器句柄到微端口的 MiniportInitialize 函数

• 传递微端口适配器环境句柄，当 NDIS 调用任何 MiniportXxx 函数时，NDIS 将这个适配器指定环境区域的句柄传递给微端口

• 指定它的 NIC 是否是一个总线管理器 DMA 设备

• 指定调用者的 NIC 的 I/O 总线接口类型

通过 NdisMSetAttributesEx，微端口可以指定有关超时和 Token Ring 错误的 NDIS 缺省行为的变化。

中间层驱动程序必须设置 NDIS_ATTRIBUTE_INTERMEDLATE_DRIVER 标志来调用 NdisMSetAttributerEx 。中间层驱动程序也必须设置 NDIS_ATTRIBUTE_NO_HALT_ON_SUSPEND 标志，来阻止微端口在系统转向低耗能(睡眠)状态时，停止驱动程序。

早期的管理一个 PM 不知（non-PM-aware）的 NIC 的微端口可以设置 NDIS_ATTRIBUTE_NO_HALT_ON_SUSPEND 标志，来阻止 NDIS 在系统转向睡眠(D3)状态之前停止微端口。早期微端口应设置这个标志，仅在以下情况：

• 它保存它所需要的所有硬件环境

• 它在睡眠状态(D3)之前将 NIC 置为合适的状态

• 它可以唤起 NIC 到工作状态(D0)

有关更多信息，见 6.6 节

一个非串行微端口必须设置 NDIS_ATTRIBUTE_DESERIALIIE 标志来调用 NdisMSetAttrributerEx。一个面向连接的微端口，总是非串行的，所以并不需要这么做。

3.2.2 声明资源

在微端口决定它的硬件资源并调用 NdisMSetAttribute(Ex)之后，注册它将要使用的系统资源。微端口必须在声明资源，包括初始时钟之前，调用 NdisMSetAttribates(Ex)。无论它所管理的 NIC 类型是什么，微端口必须为 NIC 管理设备分配内存。这个内存是附加在微端口为其适配器环境所分配的内存上的。例如，微端口应为接收到来的帧或为发送数据的阶段缓存分配内存。

下节将讲述微端口如何分配内存和注册 I/O 端口。这些操作以一般措辞讲述，然后分别针对总线管理器 DMA NICS、编程 I/O NICS、内存映射 NICS 和从属 DMA NICS 讲述。

3.2.2.1 分配内存

一般在初始化中，微端口应分配缓存、包描述符和用来请求发送和接收数据的缓存描述符。所分配内存的类型和数量依赖于 NIC 的类型，如同微端口所支持的接收样式类型。

内存可以共享或不共享。仅当内存由一个总线管理器 DMA NIC 访问时，驱动程序分配共享内存。总线管理器 DMA NIC 微端口必须为微端口和 NIC 都可访问的缓存分配共享内存。

内存可被 cache 或不被 cache。不可被 cache 的内存是一种稀有的系统资源，并且它很难分配。

管理总线管理器 DMA NIC 的微端口必须确保它所使用的任何可被 cache 的内存，在从缓存读取接收到的数据或从缓存发送数据之前是一致的。这种微端口必须更新它的缓存确保发送和接收中 cache 一致性。

管理从属 DMA NIC 或非 DMA NIC 的微端口不需要关心与 cache 一致性。由微端口为从属 DMA NIC 调用的 NdisXxx 函数为微端口更新了内存，非 DMA NIC 的微端口并不分配

cache。

以下函数是用来分配内存的。

• NdisAllocateMemory 或 NdisAllocateMemoryWithTag 分配一个长度由调用者指定的缓存。此内存不共享但是它可被 cache 或不被 cache，这依赖于调用者的请求。内存可以是连续的并且是在一指定物理地址界限内。

• NdisMAllocateSharedMemory 分配共享内存，可被 cache 或不被 cache，这个调用既返回一个 NIC 使用的物理地址指针，又返回一个微端口使用的虚地址指针。NdisMAllocateSharedMemory 必须在 NdisMSetAttributes(Ex)和 NdisMAllocateMapRegisters 或 NdisMInitializeScatterGatherDma 之后调用。

• NdisAllocatePacketPool 或 NdisAllocatePacketPoolEx 从已分配包描述符中分配和初始化不分页的系统内存快

•NdisAllocatePacket 从调用 NdisAllocatePacketPool(Ex)所分配的包池中分配一个包描述符

• NdisAllocateBufferPool 返回一个调用者可以用 NdisAllocateBuffer 分配的缓存描述符的句柄

• NdisAllocateBuffer 从由 NdisAllocateBufferPool 分配的缓存池中分配一个缓存描述符

3.2.2.2 注册端口

微端口调用 NdisMRegisterIoPortRange 声明 I/O 端口地址，微端口需要这个地址使用 NdisRawWrite(Read)PortXxx 向端口写入或读取数据。调用 NdisMRegisterIoPortRange 作如下两件事：

1.确定 NIC 试图注册的 I/O 端口是否已被分配给设备了。

2.映射 I/O 端口以使微端口可以向使用 NdisRawWrite(Read)PortXxx 注册的设备写或读。NdisRawWrite(Read)Xxx 函数直到 NdisRegisterIoPortRange 成功完成以后才不可以使用。

3.2.2.3 总线管理器 DMA 设备初始化

DMA 初始化的可选项依赖于微端口驱动程序的类型：

• 管理总线管理器 DMA NIC 的串行微端口必须在 MiniportInitialize 函数中调用 NdisMAllocateMapRegisters 为 DMA 操作分配映射注册表。

• 管理总线管理器 DMA NIC 的非串行或面向连接微端口应在它的 MiniportInitialize 函数中调用 NdisMInitializeScaterGatherDma 来初始化 DMA 资源。使用 NdisMInitializeScatterGatherDma 的优点在于，可以动态地为每个 DMA 传递分配或取消分配系统资源，例如映射注册表，并且优点还在于，通过获得映射实际范围的物理地址过程来提高 DMA 的性能。除了调用 NdisMInitializeScatterGatherDam，管理总线管理器 DMA NIC 的非串行或面向连接微端口，也可调用 NdisMAllocateMapRegisters 来静态分配映射注册表；但是，这导致 DMA 性能的降低和系统资源利用率的降低，

用 NdisMAllocateMapRegisters 分配映射注册表

在 DMA 初始化时，微端口调用的 NdisMAllocattMapRegisters 应当首先调用 NdisQueryMapRegisterrCout 来确定整个系统中有多少映射注册表可用。然后微端口调用 NdisMAllocateMapRegisters 来声明这些可用的系统映射注册表。微端口应仅分配一个单一 DMA 传递所需要的映射注册表。这是因为注册表是系统共享资源，声明过多的的注册表不利于系统整体性能，并且分配比一个单一传递所需注册表多的注册表仅仅使用了本来可被其他驱动程序使用的资源，并不会提高性能。

当微端口调用 NdisMAllocateMapRegisters 时，它必须指定映射的最大缓存(MaximumPhysicaMapping)，以适应 NIC 的最大 DMA 能力。给这个参数指定大于 0XF000

的值可能会引起 NdisMAllocateMapRegisters 失败。如果它运行在物理地址扩展(PAE)的 Microsoft Windows 2000 DataCenter Server 操作系统下，指定大于 0XF000 将无疑地导致 NdisMAllocateMapRegisters 失败。

在微端口初始化时，它只一次分配映射注册表。在注册表分配成功后，它属于微端口，直到当驱动程序卸载时，它们被收回为止。

用 NdisMInitializeScatterGatherDma 节省系统资源

管理总线管理器 DMA NIC 的非串行化或面向连接的微端口可以调用 NdisMInitializeScatterGatherDma，来为 DMA 操作初始化系统资源，这种驱动程序不需要为 DMA 操作分配映射注册表。

调用 NdisMInitializeScatterGatherDma 的驱动程序在 DMA 初始化时，使用合理的方法为 DMA 发送操作获取物理地址(见 4.4.5.1 节)。

NdisMInitializeScatterGatherDma 仅支持 32 位和 64 位 DMA 地址。NdisMinitializeScatterGatherDma 不支持 24 位 DMA 地址。

为缓冲区分配包池，缓存池和共享内存

为包池、缓存描述符和为缓存分配的共享内存的分配是由映射注册表（也许由微端口）是否运行在单处理器或多处理器上决定。微端口可以用 NdisSystemProcessorCount 来决定机器的类型。通常，多处理器的机器是一个服务器，由于服务器通常比单处理机(通常是客户端)处理更多的网络流量，所以基本上为服务器分配更多的资源。

除了为它的微端口适配器环境分配内存，一个总线管理器 DMA 微端口应当建立它用来指示包的包池，在初始化当中，微端口必须：

• 调用 NdisAllocatePacketPool 为包描述符分配包池。它必须调用 NdisAllocatePacket 从包池中分配一个包描述符。

• 调用 NdisAllocateBufferPool 为映射到组成包的缓存的缓存描述符分配缓存池。

•调用 NdisMAllocateSharedMemory 为用来从网络接收内入数据的一定量缓存分配共享内存，因为它可以被 NIC 和微端口共同访问，所以内存必须是共享内存。当内存用来发送和接收时，NIC 使用主机内存，NIC 使用内存的物理地址；微端口使用虚地址，如果调用没有先用 NdisMAllocateMapRegisters 分配映射注册表或用 NdisMInitializeScatterGatherDma 初始化系统资源，那么，调用 NdisMAllocateSharedMemory 会失败。

通常，DMA NIC 微端口应分配可被 cache 的内存。这也有一些例外，在 4.6.4 节注明。这种 NIC 微端口必须更新它的缓存，以确保在发送和接收中 cache 与内存的一致性。

如果微端口支持向 NDIS 指示多包接收，它必须为使用数组传递指向包的指针分配内存，这可以是各种类型。

3.2.2.4 可编程 I/O 设备初始化

管理可编程 I/O(PIO)设备的微端口通过向端口区域写或读，来进行发送和接收操作。微端口调用 NdisRawReadPortXxx 从接收缓存端口将数据读入临时缓存。如果缓存属于协议，那么在它的 MiniportTransferData 函数中调用此函数，否则在它的 MiniportHandleInterrupt 函数中调用。对于发送，微端口使用 NdisRaoWritePortXxx，来将数据写到 MiniportSend 或 MiniportSendPackets 函数的包中。

如果使用阶段缓存，微端口必须用 NdisAllocateMemory 或 NdisAllocateMemoryWithTag 分配缓存。除非驱动程序支持多包接收指示，任何它所需要管理的设备的附加内存都应是不可共享的，由 NdisAllocateMemory 分配的。如果微端口用 NdisMIndicateReceivePacket 指示，它必须至少分配一个如同 3.2.2.1 节所讲述的包。

为了建立端口范围，微端口必须以以下方式读取注册表：

1.用 NdisXxxConfiguration 调用或一个总线类型指定的 NdisXxx 调用。

2.通过调用 NdisMRegisterIoPortRange 来表明它所需求的 I/O 端口。

3.调用 NdisRead(Write)PortXxx 移动数据。

3.2.2.5 内存映射设备初始化

内存映射设备的微端口必须将 NIC 内置的内存映射到主机系统空间内存，以使它可以从卡上传递接收数据和向卡上传递发送的数据。微端口传递数据的内存是 NIC 的一部分；但是，驱动程序使用的是映射了的虚系统空间地址。

当接收数据时，微端口用缓存描述符所映射的虚范围指示接收数据，并且当/如果调用了 MiniportTransferDate 函数，微端口将数据从内置内存拷贝到协议驱动程序所提供的缓存中。如果没有协议驱动程序传递数据，当微端口从它的 MiniportHandleInterrapt 函数或它的轮询 MiniportTimerXxx 函数返回时，丢弃数据。如果微端口使用临时缓存，它用 NdisAllocateMemory 或 NdisAllocateMemoryWithTag 分配临时缓存。

如同一个 PIO 设备，拥有内置 RAM 的 NIC 的微端口调用 NdisMRegisterIoPortRange 注册它所声明的端口地址。这个调用返回一个映射了的虚端口地址，通过这个地址微端口可以访问注册了的适配器。这个函数映射的端口地址仅供 NdisRawWtite(Read)PortXxx(清除中断;设置和清除请求函数)来管理设备，但是它不能象 PIO 设备所做的那样传递数据。

将数据移入或移出 board，这种类型 NIC 的微端口必须映射 NIC 设备的内存到主机地址空间。在初始化当中，微端口通过调用 NdisReadConfiguration 或调用总线类型指定的 NdisXxx，来从注册表中读取物理地址和内置内存大小。一旦确定了配置信息，微端口必须调用：

• NdisSetPhysicalAddressHigh 和 NdisSetPhysicalAddressLow 设置 32 位物理地址，然后调用

• NdisMMapIoSpace 用这个物理地址把设备内存映射到微端口可访问的主机地址空间，NdisMMapIoSpace 返回一个微端口可用的虚地址指针来参照这个设备内存。

这个驱动程序使用 NdisRead(Write)RegisterXxx 对 NIC 的注册表进行读或写。

如果驱动程序用 NdisMIndicateReceivePacket 或 NdisMCoIndicateReceivePacket 指示请求的数据，它必须分配一个如 3.2.2.1 节所讲述的链状缓存包。

3.2.2.6 从属 DMA 设备初始化

管理 DMA NIC 的微端口调用 NdisAllocateMemory 来分配内存，在发送时，微端口将数据从它发送到系统 DMA 控制器，在接收时，将数据传给它。

管理从属 DMA 设备的微端口调用以下函数：

• NdisMRegisterDmaChannel 来声明一个通道用于发送和接收

• NdisMAllocateSharedMemory 分配内存，通过它，微端口可以向 DMA 控制器写出或读入数据

当发送和接收发生时，为 DMA 映射内存。如果驱动程序用 NdisMIndicateReceivePacket 或 NdisMCoIndicateReceivePacket 指示接收数据时，它必须分配至少一个如 3.2.2.1 所讲述的包。

3.2.3 注册一个中断

在调用 NdisMSetAttributes(Ex)、分配内存、声明资源之后，管理中断的 NIC 微端口必须调用 NdisMRegisterInterrupt，来确认中断并且声明中断的向量/等级。微端口为一个透明

的 NDIS_MINIPORT_INIERRUPT 类型变量分配永久存储区，并且传递指向这个向量的指针给 NdisMRegiterInterrupt。微端口传递有关中断的信息，例如如下：

- 总线相关的中断向量和中断等级
- 是否在每个中断调用 MiniportISR
- 是否共享中断
- 它的中断模式

由微端口指定的中断类型或是 NdisInterruptLevelSentsitive 或是 NdisInterruptLatched。NdisMRegisterInterrupt 代表微端口，以声明一个中断响应产生 HAL 调用，然后调用 I/O 系统与中断相连。当 NDIS 连接中断时，它首先向所有中断提供它自己的 ISR 函数。在 NDIS ISR 内部，做出是把中断传递给微端口的 MiniportISR 或 MiniportDisableInterrupt 函数还是在 NDIS 内部处理中断的决定(依据 NdisMRegisterInterrupt 提供的 RequestIsr 值)，然后对 MiniportHandleInterrupt 排队。

NdisMRegisterInterrupt 对 NIC 状态没有影响。但是，微端口在调用 NdisMRegisterInterrupt 时就可以接收中断了，即在 NdisMRegisterInterrupt 返回之前。即使 NIC 的中断是不可用的，微端口可以一直等待中断，直到它调用 NdisMDeregisterInterrupt 返回为止。在调用 NdisMRegisterInterrupt 之前，微端口应当进行任何适配器所指定的，用来处理中断时所要进行的准备操作，即，微端口必须产生和初始化任何它需要的结构，也许初始化它所需要用来进同步访问共享数据所需的自旋锁。当微端口注册它的中断和准备处理中断时，微端口可以允许设备指定的方式中断——例如，重置 NIC。

微端口在初始化当中，测试它的设备以确保设备有能力产生中断，是一个很好的习惯。当 NdisMRegisterInterrupt 返回时。微端口可以进行这个测试，如果设备运行不正常，微端口应当释放所有声明的资源并且从 MiniportInitialize 返回 NDIS_STATUS_FALLURE，以使微端口可以从系统中删除。

微端口可以通过调用 NdisMSetAttributesEx 改变缺省的超时设置。这个函数必须在 MiniportInitialize 中调用，如 3.1.2.3 节所讲，在 MiniportInitialize 返回之后，驱动程序不能改变缺省的超时特征。有关微端口的 CheckForHang 函数的讨论见 3.1.2.7 节。

3.2.4 注册一个关闭函数

微端口应当调用 NdisMRegisterAdapterShutdownHandler 用 NDIS 注册一个关闭函数。

微端口向 NdisMRegisterAdapterShutdownHandler 传递它的 MiniportShutdown 函数地址、由 NDIS 提供的微端口适配器的 MiniportInitialize 的句柄和当它被调用时，传递给 MiniportShutdown 的环境。NdisMRegisterAdapterShutdownHandler 通常在 MiniportInitialize 的最后，在微端口从 MiniportInitialize 返回之前被调用。有关关闭句柄的更多信息参见 7.3 节。

3.2.5 初始化轮询时钟

一些微端口控制的设备不产生中断，因此它必须被轮询。这种微端口不调用 NdisRegisterInterrupt，而是由小型端口分配一个或多个用于轮询设备的时钟对象。

MiniportInitialize 调用 NdisMIntializeTimer 来为每个微端口所需的时钟初始化一个时钟对象。NdisMinitializeTimer 将初始化的时钟与微端口提供的时钟函数相连，当时钟失效时，调用 MiniportTimer 函数。NDIS 提供了两种类型的时钟：一次性时钟和周期性时钟，当一次性时钟由 NdisMSetTimer 设置并且时钟终止时，它必须调用 NdisMSetTimer 重新设置。由于

在每个终止时重新设置时钟上花费时间，所以这种请求会引起等待时间堆积。周期性时钟，设置一次后它在每个要求的时间间隔内不停止终止。这种周期性时钟很适合在一个微端口进行轮询，一旦调用 NdisMSetPeriodicTimer 设置了时钟，它将不停地终止，引起相关的时钟函数的调用，直到时钟在以后的调用 NdisMsSetPeriodicTimer 被重新设置或调用 NdisMCancelTimer 清除时为止。

如果时钟函数与其他微端口函数共享资源，那么必须对这些资源进行保护。如果时钟函数与运行在同一 IRQL(DISPATCH_LEVEL)上的函数共享资源，那么用一个自旋锁来同步访问这些资源。为了获得早期已初始化的自旋锁，函数调用 NdisDprAcquireSpinLock 和 NdisDprReleaseSpinLock。这些函数要比 NdisAcquireSpinLock 和 NdisReleaseSpinLock 快。

如果时钟函数与 MiniportISR 或 MiniportDisableInterrupt 共享资源，那么必须调用 NdisMSynchronizeWithInterrupt 同步访问。微端口在它的 MiniportSynchroniczeISR 函数中执行任何与时钟相关的对共享资源的操作，通常是设备注册表，MiniportSychronicISR 函数的地址传递给了 NdisMSynchronizeWithInterrupt。

3.2.6 在初始化当中的同步

当微端口初始化它的 NIC 时，它在继续执行之前需要等待状态的改变：

• MiniportInitialize 调用 NdisMSleep 来延缓执行一个以微秒为单位的时间间隔，当 MiniportInitialize 调用 NdisMSleep 时，它将进入挂起状态。当睡眠间隔结束时，MiniportInitialize 恢复执行，微端口不需要调用 NdisStallExecution，函数所调用的时间间隔应小于 50 微秒。

• MiniportInitialize 可与事件同步。MiniportInitialize 可以通过调用 NdisInitializeEvent 产生一个事件，然后，在 MiniportInitialize 向它的 NIC 写入之后，它调用 NdisWaitEvent，当一个中断发生和 MiniportHandleInterrapt 运行时，它清除 NIC 上的中断并且通过调用 NdisSetEvent 设置事件，等待事件的 MiniportInitialize。等到事件后，调用 NdisResetEvert 清除事件继续它的执行。

3.2.7 在初始化时处理错误

如果在初始化中的任何时候，微端口返回一个出错状态——例如，如果它不能获得它所需要的资源或如果它不能发现它所支持的介质——它必须取消它对整个系统所进行的操作，尤其：

• 如果它已分配了内存，微端口必须调用 NdisFreeMemory 来收回分配内存。

• 如果它声明了 I/O 资源，微端口必须调用 NdisMDeregisterToPortRange 来释放这些资源。

• 如果它已调用了 NdisOpenConfiguration，微端口必须调用 NdisCloseConfiguration。

• 如果它已分配了映射注册表，微端口必须调用 NdisMFreeMapRegisters。

• 如果它已分配了一个 DMA 通道，微端口必须调用 NdisMDeregisterDmaChannel。

• 如果它已用 NdisMAllocateSharedMemory 分配了共享内存，它必须调用 NdisMFreeSharedMerory。

在初始化中失败的微端口可以调用 NdisWriteErrorLogEnery 在出错日志中用一个出错代码和出错类型指定的值记录问题。

3.3 查询微端口信息

如果 MiniportIntialize 返回 NDIS_STATUS_SUCCESS，NDIS 查询和保存驱动程序的操作特征。这些特征由微端口管理的对象标识符(OID)来解释。微端口必须支持所有基本命令介质指定的 OID，有关获取和设置微端口信息的更多信息见第五章。

3.4 减少微端口初始化时间

驱动程序初始化所花费的时间显然会影响 Windows 2000 启动所需的整个时间，因此，减小驱动程序初始化时间是很重要的，驱动程序初始化的时间应在 10 个毫秒内或更少。

为了减少初始化时间，NDIS 微端口应做如下操作：

•在检测链接速度之前，进行介质检测，如果介质检测失败(因为网线没有连接到 NIC)，那么微端口不进行链接速度率的检测

•如果介质检测成功，那么微端口应当使链接速率的检测服从于一个时钟函数，由于网络协议直到微端口被装载之后才被装载，微端口应当有充足的时间在一个协议发出一个 OID_GEN_LINK_SPEED 请求来请求链接速度之前获得链接速率

如果微端口在获得链接速率之前接收到 OID_GEN_LINK_SPEED 请求，OID_GEN_CURRENT_SPEED 请求或发送请求时，它将挂起请求或发送操作，在它获得链接速率之后完成这示或操作。微端口对这种请求或发送操作不应让其失败。为了防止一个挂起的请求或发送操作时间结束，微端口在它的 MiniportInitialize 的环境函数内，使用 NdisMSetAttributesEx 来设置恰当的超时间隔。

下列伪代码例子说明如何：

•在初始化当中处理介质检测和链接速率检测。

•使用一个时钟函数获得链接速率以及完成任何挂起的 OID 请求。

•如果链接速率检测没有完成时，如何挂起 OID_GEN_LINK_SPEED 和 OID_GEN_CURRENT_FILTER 请求。

•如果链接链速率检测没有完成时，如何挂已发送操作。

在初始化当中处理介质检测和链接速率检测

下列伪代码说明如何在 MinniportInitialize 的环境中处理介质检测和链接速率检测：

```
IF (there is no link pulse on the physical layer interface)
// The cable is not connected to NIC
 {
  // Do not link-speed detection
 }
  else
{
  Adapter→NegotiotionDone = FALSE;
  Command NIC to start the link-speed negotiation
  Start a timer to check for 完成 of link-speed dection
}
```

使用一个时钟函数来获得链接速率以及完成挂起 OID 请求和发送操作

为了减少初始化所花费的时间，微端口应当使链接速率的检测从属于一个时钟函数。下列伪代码说明如何使用一个时钟函数来获得链接速率。在时钟函数获得链接速率之后，它应当完成所有挂起的 OID 请求，并且重新启动所有挂起的发送操作。

```
IF (NIC has finished link-speed negotiation)
  {
```

```
Adapter→NegotiationDone = TRUE;
if     (Adapter→QueryPending)
{
// Complete the pending OID request after getting the    link sped，
NdisMQueryInformationComplete ();
// if the miniport exports an NdisMCoRequestComplete
// function，  call
//NdisMCoRequestComplete
//instead
}
if (Adapter->SetPending)
{
Send the set filter down to the adapter；
// The NIC generates an interrupt to indicate the  完成
// of the set-filter operation
NdisMSendResourceAvailable (); // Restars any    pending send operations
                                    // This timer dops not fire again
}
elsp
{
   Restart timer;
}
```

挂起 OID_GEN_LINK_SPEED 和 OID_GEN_CURRENT_PACKET_FILTER 请求

如果链接速率检测没有完成，如果一个微端口没有完成链接速率检测，它应挂起它所接收到的 OID_GEN_LINK_SPEEO 请求或 OID_GEN_CURRENT_PACKET_FILTER 请求。微端口在时钟函数完成链接速率检测之后，应当完成在它时钟函数中挂起的 OID 的异步请求。

以下伪代码说明如果链接速率检测没有完成时如何挂起一个在 MiniportQueryInformation 或 MiniportCoRequest 中的 OID_GEN_LINK_SPEED 的请求：

```
if (!Adapter->NegotiationDone && (OID==OID_GEN_LINK_SPEED))
{
 // Save the current OID request
   Adapter->QueryPending = TRUE; // Timer function tests this Value
   return NDIS_STATUS_PENDING;
```

以下伪代码说明如果链接速率检测没有完成时，如何挂起一个在 MiniportQueryInformation 或 MiniportCoRequest 中的 OID_GEN_CURRENT_PACKET_FILTER：

```
if (!Adapter->NegotiationDone && (OID==OID_GEN_CURRENT_PACKET_FILTER))
   {
   // Save the current OID request
   Adapter→Setpending = TRUE; // Timer function tests // this value
   return NDIS_STATUS_PENDING;
   }
```

如果链接速率检测失败延缓发送操作

如果一个串行化的微端口没有完成链接速率检测，它应当延缓任何它所接受的发送操

作，并且在时钟函数完成链接速率检测后，重新在一个时间函数（参见上部分）内启动这个操作。

以下伪代码说明如果链接速率检测没有完成时，如何在 MiniportSend 中延缓发送操作。

```
if (!Adapter_>NegotionDone)
return NDIS_STATUS_RESOURCES; //Force NDIS to resubmit the send
//when link-speed negotiation is
// finished.
```

以下伪代码说明如果链接速率检测没有完成时，如何在 MiniportSendPackets 中延缓发送操作。

```
if (!Adapter_>NegotionDone)
Set the Status member of NDIS_PACKET_OOB_DATA block
associated with the packet descriptor to NDIS_STATUS_RESOURCE;
return;
```

一个串行化或面向连接的微端口必须对发送操作在内部进行排队，并且在链接速率检测完成以后完成发送操作。

第四章 数据传输

本章描述微端口（Miniport）如何发送和接收数据。包括设计下述信息：

- 4.1 中断处理
- 4.2 延时处理程序
- 4.3 带外（OOB）数据包
- 4.4 数据包传输
- 4.5 非串行微端口
- 4.6 接收数据
- 4.7 保存统计量
- 4.8 802.1p 包优先性

4.1 中断处理

本节描述中断处理。更多的关于注册中断信息参见 3.1.2.3 节。

当网卡（NIC）产生一个中断时，NDIS 捕获它，并调用初始化时为这一中断注册过的每个微端口的 *MiniportISR* 例程。

一般来说，*MiniportISR*：

- 确定并记录中断的原因
- 拷贝任何易变的特殊中断信息到一个永久结构中（通常在 *MiniportAdapterContext* 某处传到 MiniportISR）
- 清除 NIC 上的中断

MiniportISR 的定义为：

```
VOID
 MiniportISR(
     OUT PBOOLEAN    InterruptRecognized,
     OUT PBOOLEAN    QueueMiniportHandleInterrupt,
     IN NDIS_HANDLE    MiniportAdapterContext
     );
```

在返回之前 *MiniportISR* 必须设置的两个传递参数：InterruptRecgnized 和 QueueMiniportHandleInterrupt。如果要识别出中断，*MiniportISR* 必须将 InterruptRecognized 设为 TRUE，否则设为 FALSE。若这一中断为共享的，并且此微端口没有识别出中断，则返回 FALSE。NDIS 就将中断传递给共享这一中断的下一个微端口的 *MiniportISR* 函数。NDIS 反复进行这一过程，直到有一个 *MiniportISR* 函数识别出该中断，否则 NDIS 将所有（共享这一中断的每个微端口的）*MiniportISR* 函数调用过一遍。

若 *MiniportISR* 识别出中断并希望其 MiniportHandleInterrupt 函数排序，就必须设置 *QueueMiniportHandle* 为 TRUE。当 *MiniportISR* 返回时，NDIS 实际上对 MiniportHandleInterrupt 函数进行了排序。若将微端口的 *InterruptRecognized* 设置为 FALSE，则 *QueueMiniportHandleInterrupt* 值假设为 FALSE。

在执行 ISR 的同时，处理器上所有其他在同一 IRQL 级别或较低 IRQL 级别的中断都被屏蔽掉了。因此，使一个微端口在 *MiniportISR* 上所花费时间最少是很重要的。否则，系统的总体运行性能，以及整个端口 I/O 性能都会降低。

对于管理一个与同一总线上其他设备共享一个中断的 NIC 的微端口来说，重要的是迅

速地检测出中断是否属于该微端口。若中断不属于该微端口，则微端口应将InterruptRecognized 设为 FALSE，并立即返回。

若中断共享且属于该微端口，或者未共享但却为该微端口识别（中断非假），微端口应在其 NIC 上禁止中断，然后从 NIC 的寄存器中得到所需要的特殊中断信息。如果需要，微端口立即从 MiniportISR 捕获中断特定的信息，或者更确切地说，微端口是在MiniportHandleInterrupt 函数中捕获中断特定信息，这一函数运行在较低的 IRQL 上。

MiniportISR 必须禁止中断，以便确保 MiniportHandleInterrupt 运行时，NIC 上没有信息被随后的中断所覆盖。MiniportISR 应当仅仅读取并保存那些在它返回后不能存取的信息，然后，MiniportISR 返回。

如果微端口不希望其 MiniportHandleInterrupt 函数进行排序，而且如果它以前在MiniportISR 中禁止中断，则在由 MiniportISR 返回之前，它必须使 NIC 重新启用中断。

ISR 函数同步

微端口的 MiniportISR 函数及其 MiniportDisableInterrupt 函数在 DIRQL 下执行。其他微端口代码则在 IRQL<=DISPATCH_LEVEL 下执行。为了防止竞争状态，任何与 MiniportISR 或 MiniportDisableInterrupt 共享资源的微端口函数必须同步地调用共享的资源。与MiniportISR 及 MiniportDisableInterrupt 同步的函数是通过调用 NdisMSychronizeWithInterrupt 来实现的，它提供一个 MiniportSychronizeISR 函数。MiniportSychronizeISR 在 DIRQL 状态下执行并保证安全调用共享的资源。

例如，**MiniportHandleInterrupt** 在清除中断状态寄存器之前，调用NdisMSynchronizeWithInterrupt。这一 NDIS 调用得到一个 *SynchronizeFunction* 参数。在**NdisMSynchronizeWithInterrupt** 内部，NDIS 在 DIRQL 状态下调用微端口的*SynchronizeFunction*，从而这一函数可以安全地清除中断状态寄存器，并确定 MiniportISR 或 MiniportDisableInterrupt 不会同时修改同一寄存器。SynchronizeFunction 应尽快执行，就象 MiniportISR 和 MiniportDisableInterrupt 函数一样。

MiniportISR 及 MiniportDisableInterrupt 都在 DIRQL 状态下运行，它们与中断对象的自旋锁同步。

中断与轮询的结合

某些时候，驱动程序开发人员可以通过将轮询 NIC 与中断的使用相结合的方法来改进驱动程序的运行。例如，一个驱动程序可以设计成对于接收指示使用中断，对于发送完整指示则禁止中断的模式。微端口可设一时钟，在时间停止时，检查发送完成状况。此外，当中断在接收时发生，微端口也要检查发送完成状态。当网络流通量很大时，这是一项合理的策略：中断处理量最小，并且没有遗漏数据包。

一个驱动程序可以抽样检测网络流通状况，以确定当前荷载下的最佳数据传输策略。若流通量很大，就应当采用刚刚描述的策略。如果网络上的流量低，驱动程序应启用“发送完成与接受”中断，因为所预期的中断数量要比高流通量环境下低，并且可能有对驱动程序发送性能不利的影响。

使用轮询和中断相结合的驱动程序，当使用轮询来管理其设备时禁止中断，而当使用中断来管理其设备时则启用中断。当 NDIS 被启用时，驱动程序也可用 NDIS 处理其设备的中断，但这要受到以上所描述内容的限制，即：

- 该中断不是共享的
- 端口所需的任何信息都可以在 MiniportHandleInterrupt 函数中捕获
- 在初始化期间及暂停过程中，其 NIC 从不产生中断

关于计时器的信息，参见 3.2.5 节。

4.2 DPC 处理程序

每个管理网卡（NIC）的微端口，其中断必须具有 MiniportHandleInterrupt 函数。即便 NDIS 完全处理中断，它也总是对这一微端口延时程序调用（DPC）排序，导致微端口 MiniportHandleInterrupt 处理程序的运行。若 MiniportISR 处理一中断，它通过在 InterruptRecognized 和 QueueMiniportHandleInterrupt 参数中的 MiniPortISR 返回的数值，来控制 MiniportHandleInterrupt 是否排序。若 MiniportISR 都设为 TRUE，则 MiniportHandleInterrupt 进行排序，以便在 IRQL DISPATCH_LEVEL 下执行；若将 InterruptRecognized 设为 FALSE，则 MiniportHandleInterrupt 不进行排序。

当 MiniportISR 或 MiniportDisableInterrupt 函数在 NIC 上禁止中断后，MiniportHandleInterrupt 被调用。MiniportHandleInterrupt 应读取它所需要的任何数据以便完成处理中断驱动的 I/O 操作。然后 MiniportHandleInterrupt 重新使 NIC 启用中断，或让 NDIS 在 MiniportHandleInterrupt 返回控制状态后，调用微端口的 MiniportEnableInterrupt 函数；或由 MiniportHandleInterrupt 内部启用中断，这样会快一些。NIC 上不会再发生中断，直到中断重新被启用。为了确保没有接收到的数据被遗漏，MiniportHandleInterrupt 应当尽快启用中断。

处理中断的微端口，完全在其 MiniportHandleInterrupt 函数的环境中处理数据接收和发送。例如，若 MiniportHandleInterrupt 确定中断的原因是一个接收任务，它通过调用在 DPC 环境内的 **NdisM(Co)IndicateReceive** 或 **NdisM*Xxx*IndicateReceiv**e 完成数据接收。4.6 节描述了微端口如何处理接收。

若无连接微端口确定产生中断的原因是完成了发送任务，它就调用 **NdisMSendComplete** 发出信号，表明对 MiniportSend 或 MiniportSendPacket 的数据包发送已经完成。类似地，若一面向连接的微端口确定中断原因是完成了发送，它就调用 **NdisMCoSendComplete** 发出信号，指示以前送到 NdisMCoSendCompletePackets 的数据包已完成，这一调用返回 NDIS_STATUS_PENDING 状态。微端口发送操作将在 4.4 节中描述。

当一微端口处理一项接收任务时，它必须不能再次发生中断，直到下述情况之一发生：

- 微端口已经完成整个微端口所分配的、包含接收数据的数据包或包序列
- 若任意协议驱动程序对数据感兴趣，微端口的 MiniportTransferData 函数将内入的数据传送到协议驱动程序所提供的包中（仅无连接微端口有 MiniportTransferData 函数）

若微端口分配的包传给了协议驱动程序，而且协议驱动程序保存此包，则微端口必须为下一次接收提供一个新的空包。当它已经处理完接收信息，并为下一次接收作好准备后，端口可以重新启用中断。

4.3 带外（OOB）数据包

每个 NDIS_PACKET 类型的描述符都具有一相关的带外（*Out-of-Band*）数据块，它传递了介质相关的和优先权的信息。这一子结构定义为：

```
typedef  struct  _NDIS_PACKET_OOB_DATA {
    Union {
        ULONGLONG      TimeToSend;
        ULONGLONG      TimeSent;
        }
```

```
    ULONGLONG           TimeReceived;
    UINT                HeaderSize;
    UINT                SizeMediaSpecificInformation;
    PVOID               MediaSpecificInformation;
    NDIS_STATUS         Status;
    }; NDIS_PACKET_OOB_DATA,   *PNDIS_PACKET_OOB_DATA
```

支持 OOB 数据的微端口驱动程序应当使用 NDIS 宏来存取这类数据。它不应计算偏移量，或者根据已知包的 OOB 数据块的确切结构另行采取行动。

OOB 数据的 NDIS 宏包括下述内容：

NDIS_OOB_FROM_PACKET(Packet)

返回 NDIS_PACKET_OOB_DATA 子结构的指针。这个一指针可以用来通过成员名来访问 NDIS_PACKET_OOB_DATA 的成员。

NDIS_SET_PACKET_STATUS (Packet, Status)

设置状态成员。

NDIS_SET_PACKET_TIME_TO_SEND (Packet, TimeSent)

设置 TimeToSend 成员。

NDIS_SET_PACKET_TIME_SENT (Packet, TimeSent)

设置 TimeSent 成员。

NDIS_SET_PACKET_TIME_RECEIVED (Packet, TimeReceived)

设置 TimeReceived 成员。

NDIS_SET_PACKET_HEADER_SIZE (Packet, HeaderSize)

设置 HeaderSize 成员。

NDIS_SET_PACKET_MEDIA_SPECIFIC_INFO (Packet, MediaSpecificInfo, Size-MediaSpecific)

设置 MediaSpecificInformation 及 SizeMediaSpecificInformation 成员。

NDIS_GET_PACKET_STATUS (Packet)

返回 **Status** 成员。

NDIS_GET_PACKET_TIME_TO_SEND (Packet)

返回 TimeToSend 成员。

NDIS_GET_PACKET_TIME_SENT (Packet)

返回 TimeSent 成员。

NDIS_GET_PACKET_TIME_RECEIVED (Packet)

返回 TimeReceived 成员。

NDIS_GET_PACKET_HEADER_SIZE (Packet)

返回 HeaderSize 成员。

NDIS_GET_PACKET_MEDIA_TO_SPECIFIC_INFO (Packet, pMediaSpecificInfo, pSizeMediaSpecificInfo)

返回介质相关信息到一个由调用程序提供的高速缓存中，并返回存储在调用程序变量中的信息字节大小。

4.3.1 等待发送的 OOB 数据

当一个支持介质特定信息以及/或者优先权的微端口接收到数据包时，它可以执行下列一个或多个步骤：

- 获取 **TimeToSend**

- 获取 **SizeMediaSpecificInformaton** 和 **MediaSpecificInfprmation**

在返回之前，串行微端口中的 MiniportSendPackets 为每个通过的包设置 **Status** 成员。若它完成了同步发送，它将 **Status** 设置为除 NDIS_STATUS_PENDING 之外的某一值。否则，MiniportSendPackets 将 **Status** 成员设置为 NDIS_STATUS_PENDING。此后，当发送完成时，发送状态必须在 **Status** 参数中返回给 **NdisMSendComplete**。MiniportSendPackets 返回后，OOB 数据的 **Status** 成员，以及所有其他成员，对于 NIC 驱动程序都是只读的。

在所有的提交给非串行驱动程序的 MiniportSendPackets 函数，或在面向连接驱动程序的 MiniportCoSendPackets 函数的包描述符中，NDIS 库都忽略了 OOB 块。NDIS 假定非串行微端口将每个包描述符同步地用 **NdisMSendComplete** 传输到 MiniportSendPackets，面向连接微端口驱动程序将每个包描述符用 **NdisCoSendComplete** 传输到 MiniportCoSendPackets。因此，非串行驱动程序的 MiniportSendPackets 函数，或面向连接驱动程序的 MiniportCoSendPackets 函数通常忽略了 NDIS_PACKET_OOB_DATA 块的 **Status** 成员，但它可以将这一成员设置为某个状态，这个状态与后来传给 **NdisM（Co）SendComplete** 的状态相同。

如果调用 MiniportSend，而且是同步发送，则 MiniportSend 总是把发送状态作为 MiniportSend 状态返回。OOB 数据从不用来返回 MiniportSend 操作的状态。

总之：

- 除 OOB 数据的 **Status** 成员以外，所有其他成员对于串行微端口的 MiniportSendPackets 函数都是只读的。通过写 **Status** 成员来为每个同步完成的包返回发送状态。
- NDIS 忽略了包描述符的 **Status** 成员，描述符将其传到非串行驱动程序的 MiniportSendPackets 函数或面向连接驱动程序的 MiniportCoSendPackets 函数。对于输入序列中的每个包，非串行驱动程序的 MiniportSendPackets 函数必须与 **NdisMSendComplete** 同步完成对每个包的处理。类似地，对于输入序列中的每个包，面向连接驱动程序的 MiniportCoSendPackets 函数必须与 **NdisMSendComplete** 同步完成对每个包的处理。
- 当包被同步发送时，或者由 MiniportSend 或 MiniportSendPackets 执行时，发送的最终状态总是作为一个参数返回给 **NdisMSendComplete**。当包被 MiniportCoSendPackets 同步发送时，最终状态总是作为参数返回给 **NdisMSendComplete**。
- 所有 OOB 数据对于 **MiniportSend** 都是只读的。MiniportSend 总是把发送状态作为 MiniportSend 状态返回，如果返回的状态是 NDIS_STATUS_PENDING，这个状态以后将作为一个参数返回给 **NdisMSendComplete**。

4.3.2 接收的 OOB 数据

如果微端口传送介质相关的信息以及/或者其优先权，它必须用 **NdisMIndicateReceivePacket** 指明一个或多个数据包来指示接收数据，或者，如果微端口是面向连接的，则由 **NdisMCoIndicateReceivePacket** 来指示。在指示包之前，微端口必须写带外（OOB）数据的 **Status** 成员。微端口必须设置 **Status** 成员的值：在不希望高层驱动程序保存此包时，将它设置为 NDIS_STATUS_RESOURCES，否则为 NDIS_STATUS_SCCESS。

此外，微端口可执行下述一个或多个步骤：

- 若时间可以确定（如 ATM NIC 驱动程序），则设定 **TimeSent**
- 设定 TimeReceived
- 设定 HeaderSize。典型地，当包描述符被初始化时，微端口只须对每一个包描述符

设定一次

- 设置指向 MediaSpecificInformation 及 SizeMediaSpecificInfo 的指针
- 若微端口指示接收包的时间戳，它必须应用 NDIS_SET_PACKET_TIME_RECEIVED 和/或 NDIS_SET_PACKET_TIME_SENT 宏在与包描述符相关的 NDIS_PACKET_OOB_DATA 中设置 TimeReceived 和/或 TimeSent 成员。若微端口应用系统时间作为时间戳，它可以调用一次 NdisGetCurrentSystemTime 来为指定的包序列中所有数据包设置接收时间戳。
- 若驱动程序通过接收指示额外 OOB 信息，它必须将 SizeMediaSpecificInfo 设为在执行 MediaSpecificInformation 时，调用者分配缓冲区中包含的信息的字节数。微端口可以用 NDIS_SET_PACKET_MEDIA_SPECIFIC_INFO 宏来设定这些数值。否则，SizeMediaSpecificInfo 应为零，而 MediaSpecificInfomation 应当为 NULL。

当 **NdisM(Co)IndicateReceivePacket** 返回时，OOB 数据对于该微端口为只读的。因为高层已经使用过 OOB，串行微端口应当读取 **Status** 成员，以确定哪个包可供重新使用。这类包所具有的状态与 NDIS_STATUS_PENDING 不同。任意带有 NDIS_STATUS_PENDING **Status** 成员的包属于更高层的驱动程序，当前不能为微端口所使用，随后将在其 **MiniportReturnPacket** 函数中返回给微端口。

若微端口在它所指示包序列的任意一个包中写了 NDIS_ATATUS_RESOURCES 状态，当 NdisMIndicateReceivePacket 返回时，这个包也将返回给微端口。

非串行或面向连接的微端口驱动程序在 **NdisMIndicateReceivePacket** 的返回时，不必检验指示包的 **Status** 成员。相反，非串行端口驱动程序在指明包描述符之前，必须以局部变量方式存储包的 **Status** 成员。当 **NdisMIndicateReceivePacket** 返回时，微端口驱动程序应当检查所存储包的 **Status** 成员。若微端口驱动程序在指明包描述符之前将 **Status** 成员设为 NDIS_STATUS_RESOURCES，则它在 **NdisMIndicateReceivePacket** 返回之后，应当立即收回包描述符，这可能通过调用它所拥有的 MiniportReturnPacket 函数来实现。这种情况下，NDIS 将不调用微端口驱动程序的 MiniportReturnPacket 函数来返回包描述符。若微端口驱动程序在指明包描述符之前将包的 **Status** 成员设为 NDIS_STATUS_SUCCESS，则微端口驱动程序在 NDIS 将包描述符返回给微端口驱动程序的 MiniportReturnPacket 之前，不能收回包描述符。

4.4 发送包

无连接微端口可以采用下列两种方式的任意一种来发送包：

- MiniportSendPackets 函数。NDIS 每次传送给 MiniportSendPackets 一个或多个包描述符指针序列形式的数据包。
- MiniportSend 函数。NDIS 总是每次传送一个包描述符给 MiniportSend。

无连接微端口要用 **NdisMRegisterMiniport** 注册 MiniportSendPackets 和 MiniportSend 两个函数，但 NDIS 仅调用 MiniportSendPackets。

面向连接微端口必须使用其 MiniportCoSendPackets 函数来发送包。NDIS 每次向 MiniportCoSendPackets 传送一个或多个包描述符指针序列形式的数据包。

一般来说，与分配大块不可缓冲的内存相比，为传输数据分配高速缓存更为可取。不可缓冲内存是一种稀少的系统资源。分配大块不可缓冲内存并非总是可行的，所以任何使用不可缓冲内存的微端口在驱动程序初始化期间必须分配内存。

微端口应当总是设想由协议驱动程序设定的缓冲区为可缓冲内存。

4.4.1 无连接微端口的多包传送

支持多包发送的无连接微端口声明了下述微端口函数：

```
VOID
    MiniportSendPackets (
        IN  NDIS_HANDLE    MiniportAdapterContext,
        IN  PPNDIS_PACKET  PacketArray,
        IN  UINT           NumberofPackets
        ) ;
```

PacketArray 是一个指针，它指向含有一个或多个包描述符的指针数组。微端口负责按协议驱动程序所确定的优先顺序发送包。NDIS 并不对从协议驱动程序接收到的数据包进行排序，而是以从协议驱动程序接收到的次序把它们传送给微端口。

微端口假定协议驱动程序从不发送太大的包，所以微端口不检查包的大小。协议驱动程序在初始化期间查询微端口以确定微端口所支持的最大包的大小。协议驱动程序负责只传送大小为微端口所接受的包。

4.4.1.1 串行微端口的多包传送

每个包描述符都有相应的带外（OOB）数据，如 4.3 节所述。OOB 块允许协议驱动程序给支持优先发送的协同操作微端口指定一个发送时间（TimeToSend）。在从 MiniportSendPackets 返回之前，串行微端口设置 OOB 块的 **Status** 成员，以指示调用的返回值。NDIS_PACKET_OOB_DATA 块的其他成员对于微端口是只读的。

为一个或多个包提供 MiniportSendPackets 和返回 NDIS_STATUS_PENDING 的串行微端口必须为各个包保存指针，因为将来还要返回到这种未决的状态。当 MiniportSendPackets 返回时，包含指针的数组将返回给调用者。

没有发送包资源的串行微端口可将 **Status** 成员设为 NDIS_STATUS_RESOURCES。这种情况下，当它通过调用 **NdisMSendComplete** 或调用 **NdisMSendResourcesAvailable** 完成发送时，无论哪一个先进行（见图 4.1），NDIS 都先将包排队，然后把它们传给微端口。

图 4.1　　串行微端口的多包传送——异步完成

NDIS 假定，若为一个单元设定了 NDIS_STATUS_RESOURCES，它也就为所有相应序列单元也设定了值。实际上，微端口暂时或永久地放弃了对包序列中所有包的所有权，在由 MiniportSendPackets 返回之前，它设定了不同于 NDIS_STATUS_PENDING 的 **Status** 成员。否则，微端口将拥有那些包资源，直到随后它调用 **NdisMSendComplete** 来放弃所有权，并将最后状态作为参数传给这一调用（见图 4.1）为止。

当串行微端口没有发送包资源时，它不返回 NDIS_STATUS_RESOURCES 而是在内部对这一请求排序。微端口可以使用微端口在包头中的保留区来链接排序的包，这可能链接到特定适配器环境结构。若微端口对包排序，它必须返回 NDIS_STATUS_PENDING 作为已排序包的状态。为了增加吞吐量，微端口可以对包进行排序，以避免当包返回给 NDIS 后再次传递给微端口的情况发生。

串行微端口在 MiniportSendPackets 返回之前，为每个包设定了 **Status** 域，无论包的完成是同步的或是异步的。

如果串行微端口同步完成发送（见图 4.2），应将包的 OOB 数据 **Status** 成员设定为发送状态，就是 NDIS_STATUS_SUCCESS，NDIS_STATUS_FALURE，NDIS_STATUS_RESOURCES，或者微端口所确定的状态。如果微端口保留了此包并将随后

发送此包，微端口将把 **Status** 成员设定为 NDIS_STATUS_PENDING。如果微端口对于数组中的包返回 NDIS_STATUS_RESOURCES 状态，则 NDIS 将对此包及数组中所有随后发送的包重新排序，并在微端口指示它有可以利用的资源时，将它们传出。

图 4.2　串行微端口的多包传送——同步完成

MiniportSendPackets 返回后，OOB 数据的 **Status** 域即禁止进入。这就意味着，当串行微端口完成一个先前已经返回了 NDIS_STATUS_PENDING 包时，微端口必须在调用 **NdisMSendComplete** 的 *Status* 参数中返回其发送状态。此微端口不必改写（通过调用 **NdisMSendComplete** 所完成的）任意包中 OOB 数据的 **Status** 成员。

4.4.1.2 非串行微端口的多包传送

每个包描述符有相应的带外（OOB）数据，如 4.3 节所述。OOB 块允许协议驱动程序为支持优先发送的协同操作微端口指定一发送时间（TimeToSend）。尽管非串行微端口可以在由 MiniportCoSendPackets 返回之前设定 OOB 数据块的 **Status** 成员，但 NDIS 忽略了这一成员。对于微端口，NDIS_PACKET_OOB_DATA 块的其他成员是只读的。

NDIS 库忽略它传给非串行微端口驱动程序的 MiniportSendPackets 函数的所有包描述符中的 OOB 数据块。NDIS 假定，非串行微端口与 **NdisMSendComplete** 异步地完成将每个包描述符输入到 MiniportSendPackets 函数中的操作。因此，非串行驱动程序的 MiniportSendPackets 函数通常忽略 NDIS_PACKET_OOB_DATA 块的 **Status** 状态成员，但它可以将这一成员设置为某个状态，这个状态与后来传给 **NdisMSendComplete** 的状态相同。

无论 MiniportSendPackets 多么缺乏足够的传输包的资源，也胜过依赖于 NDIS 对包进行排队，并重新发送数据包。非串行微端口管理其内部的包队列，驱动程序对其在内部队列中保持的传入数据包负责，直到它们通过网络传递出去为止。驱动程序也对协议检测的、传给 MiniportSendPackets 函数的包描述符的次序负责。非串行微端口必须用 **NdisMSendComplete** 来完成每一个传入的发送包（见图 4.3），非串行微端口不能调用 **NdisMSendResourcesAvaible**。

图 4.3　非串行微端口的多包传送

非串行微端口将永远不能（以一个最初提交给 MiniportSendPackets 函数的协议分配的包描述符的形式）把 NDIS_STATUS_RESOURCES 传给 **NdisMSendComplete** 。这种状态返回将导致协议请求的发送操作的失败，使得 NDIS 将以前传递过来的包描述符和所有相应资源返回给协议。

4.4.2　无连接微端口的单包发送

对于单包发送，无连接微端口声明了下述函数：

```
NDIS_STATUS
    MiniportSend (
        IN   NDIS_HANDLE    MiniportAdapterContext,
        IN   PNDIS_PACKET    Packet,
        IN   UINT    Flags
        );
```

参数 *Flags* 由共同操作的协议驱动程序和微端口定义；NDIS 不对此参数做出定义或任

何假定。然而，NDIS 提供一个函数 **NdisGetPacketFlags**，这一函数由微端口调用，回收标志值。微端口保留此包，直到发送完成。

微端口不必检查包的大小，然而微端口应假定协议驱动程序永远不会发送太大的包。协议驱动程序在初始化期间查询微端口以确定它所支持最大包的大小。协议负责只传送尺寸为微端口所能支持的包。

若微端口可即刻完成发送（见图 4.4），它返回 NDIS_STATUS_SUCCESS，如果因任何原因失败或可能的微端口确定的错误状态，返回 NDIS_STATUS_FAILURE，然后放弃包的所有权并返回给发送者。MiniportSend 总是把发送状态作为 MiniportSend 状态返回；它从不为这类包的 OOB 数据设定 **Status** 成员。

图 4.4　无连接微端口的单包发送——同步完成

若微端口不能立即完成发送任务（见图 4.5），它就返回 NDIS_STATUS_PENDING。这种情况下，微端口保留包的所有权，直到发送完成。例如，微端口在返回未决状态之前，在内部已经将包排序或者开始发送（但尚未完成）。当发送完成时，微端口必须调用 **NdisMsendComplete**，将一个指针传给已经发送完毕的包描述符，放弃包资源的所有权给发送者，以便其再次使用或者闲置它。

图 4.5　串行微端口的单包发送——异步完成

串行微端口如何处理资源问题

若串行微端口缺少发送资源，它可返回 NDIS_STATUS_RESOURCES。这种情况下，当微端口调用 **NdisMSengResourcesAvailable** 或 **NdisMSendComplete** 来表明它现在可以接受包（见图 4.5）时，NDIS 对包排序并将包重新送到 MiniportSend。串行微端口仅在返回 NDIS_STATUS_PENDING 和调用 **NdisMSendCommlete** 之间的间隙里，才可调用 **NdisMSendResourcesAvailable**。

非串行微端口如何处理资源问题

若非串行微端口缺少发送资源，它必须在内部对包排序并返回 NDIS_STATUS_PENDING（见图 4.6）。非串行微端口不能返回 NDIS_STATUS_RESOURCES。这种状态返回将导致协议请求的发送操作的失败，使得 NDIS 将以前传递过来的包描述符和所有相应资源返回给协议。

图 4.6　非串行微端口的单包发送——异步完成

无论 MiniportSend 多么缺乏足够的资源传送数据包，也胜过依赖于 NDIS 对包进行排队，并重新传送数据包，非串行微端口管理自己内部包的顺序。微端口在其内部队列中对处理内入的发送包负责，直到它们通过网络被传出。

4.4.3 面向连接微端口的多包发送

在发送数据之前，面向连接微端口通常总是激活一个虚连接（VC），依靠它来发送数据。然后，微端口使用 MiniportCoSendPackets 函数发出数据。数据送出后，微端口去活并有时删除在其上发送数据的 VC。关于建立、激活、去活、删除 VC 的信息，参见 2.1.4 节。

图 4.7 说明了面向连接微端口的多包发送。

图 4.7　面向连接微端口的多包发送

MiniportCoSendPackets 的定义为

```
VOID
MiniportCoSendPackets (
    IN   NDIS_HANDLE    MiniportVcContext,
    IN   PPNDIS_PACKET   PacketArray,
    IN   UINT    NumberofPackets
    );
```

MiniportVcContext 标识虚连接（VC），在其上进行包传输。VC 用面向连接客户的或者呼叫管理器/MCM 的调用 NdisCoCreateVc 来建立，**NdisCoCreateVc** 使得 NDIS 调用微端口的 *MiniportCoCreateVc* 函数。

每个 PacketArray 是指向一个或多个包描述符的指针数组的指针。微端口负责按照驱动程序设定的优先次序发送包。NDIS 不对从协议驱动程序接收到的包进行排序，但将它们以接收到的次序传给微端口。

微端口不必检查包的大小，然而微端口应假设协议驱动程序永远不会发送太大的包。协议驱动程序在初始化期间查询微端口以确定它所支持的最大包的大小。协议只传递大小为微端口所支持的数据包。

每个包描述符都有相应的带外（OOB）数据，如 4.3 节所述。OOB 块允许协议驱动程序给支持优先发送的共享微端口指定一发送时间（**TimeToSend**）。尽管面向连接微端口可在从 MiniportCoSendPacket 返回之前设定 OOB 块的 **Status** 成员，但这一成员为 NDIS 所忽略。NDIS_PACKET_OOB_DATA 块的其他成员对于微端口是只读的。

NDIS 库在提交给面向连接微端口 MiniportCoSendPackets 函数的所有包描述符中忽略了 OOB 数据块。NDIS 假定，面向连接微端口与 NdisMCoSendComplete 函数异步地将包描述符传给 **MiniportCoSendPackets** 函数。因此，MiniportCoSendPackets 经常忽略了 NDIS_PACKET_OOB_DATA 的 **Status** 成员，但它可以将这一成员设置为某个状态，这个状态与后来传给 **NdisMCoSendComplete** 的状态相同。

无论 MiniportCoSendPackets 多么缺乏发送包的资源，也胜过倚赖于 NDIS（对包）排序，并重新发送数据包。面向连接微端口管理自己的内部包队列。这种微端口对处理进入内部队列的内入包负责，直到它们通过网络发送出去，微端口也保留协议检测的、传给 MiniportCoSendPackets 函数的包描述符的次序。面向连接微端口必须用 **NdisMSendCoComplete** 来完成每一个内入的发送包（见图 4.3）。

面向连接微端口永远不能把 STATUS_INSUFFICIENT_RESOURCES 传给 **NdisMSendComplete**（以一个最初提交给 MiniportCoSendPackets 函数的由协议分配的包描述符的形式）。这种状态返回将导致协议请求的发送操作的失败，使得 NDIS 将以前传递过来的包描述符和所有相应资源返回给协议。

4.4.4　发送数据前的内存同步

如果微端口管理总线控制器 DMA NIC，则它必须保证在 NIC 从共享内存中读取之前，传递给 NIC 的、用于通过网络发送的任意数据处于物理内存之中。因此，微端口必须调用

NdisFlushBuffer 以强迫刷新处理器的高速缓存。刷新调用的时间并不是关键的，但必须在数据写到共享内存之后以及释放到 NIC 之前，由 MiniportSend 或 Miniport(Co)SendPackets 来完成。

例如，在网络上发送一个包，MiniportSendPackets 通常给出总线控制器包段在共享内存中的物理地址，然后调用 **NdisFlushBuffer** 和 **NdisMUpdateSharedMemory**。**NdisFlushBuffer** 调用确定 WriteToDevice 参数为 TRUE，这时表明传送方向是由主机到 NIC。微端口调用 **NdisFlushBuffer** 和 **NdisMUpdateSharedMemory** 之后，它通常将数据写到 NIC 寄存器中，释放给 NIC。

由于缓冲线断裂（cache-line tearing）对于数据发送来说不是问题，微端口可以刷新数据而不涉及高速缓存的大小。

管理可编程 I/O（PIO）设备的微端口保证传给 NIC 数据的正确性；因此，这种微端口将不调用 **NdisFlushBuffer**。

从属（Slave）DMA NIC 的微端口保证 NDIS 将确保高速缓存在 DMA 传送数据过程中的一致性。

一般来说，最好为发送数据分配可缓冲内存。不可缓冲内存是一种稀少的系统资源。分配大块不可缓冲内存并非总是可行的。任何使用不可缓冲内存的微端口在驱动程序初始化期间必须对它分配内存。

微端口应当总是假定协议驱动程序设置的缓冲区为可缓冲内存。

4.4.5 发送步骤

微端口驱动程序发送数据包的调用顺序，取决于微端口所管理的 NIC 的类型。本节描述总线控制器 DMA NIC，PIO 设备以及使用 NIC 板上内存来发送包的步骤。

4.4.5.1 在总线控制器 DMA NIC 上发送包

在总线控制器 DMA NIC 上发送包所涉及的步骤取决于在 DMA 初始化过程中微端口如何初始化或如何保存系统资源：

- 管理一个总线控制器 DMA NIC 的串行微端口必须从 MiniportInitialize 函数中调用 **NdisMAllocateMapRegisters**，以便为 DMA 操作分配映射寄存器。
- 管理总线控制器 DMA NIC 的非串行或面向连接的 NIC 应当在其 MiniportInitialize 函数中调用 **NdisMInitializeGatherDma**，以将 DMA 资源初始化。使用 **NdisMInitializeScatterGatherDma** 的优点在于系统资源，例如映射寄存器，是动态地为每个 DMA 传输分配和释放的（因而更好地利用系统资源），而且 DMA 通过流水线进程得到了映射到虚拟范围的物理地址，从而改进了运行性能。管理总线控制器 DMA NIC 的非串行或面向连接的微端口可调用 **NdisMAllocateMapRegisters** 来静态地分配映射寄存器，而不是调用 **NdisMInitializeScatterGather-Dma**，然而这导致 DMA 运行性能较差和系统资源利用较差。

用 NdisMAllocateMapRegisters 保存 DMA 资源的微端口

一般而言，管理总线控制器 DMA NIC 的无连接微端口（串行的或非串行的）输出 MiniportSendPackets 函数以使其 NIC 获取最佳性能。管理总线控制器 DMA NIC 的面向连接微端口总是输出 MiniportCoSendPackets。

在其 Miniport(Co)SendPackets 函数接收一个或多个包描述符之后，在 DMA 初始化期间，用 **NdisMAllocateMapRegisters** 分配映射寄存器（见 3.2.2.3 节）的微端口在用 DMA 发送包

时，执行下述步骤：

1. 微端口调用 **NdisQueryPacket** 来读入包的长度，获取第一个缓冲描述符的指针，并确定缓冲描述符的数目及具体段数。
2. 微端口需要确定它是否有可供利用的发送缓冲区和可供利用的发送缓冲区描述符，这种描述符映射将缓冲区映射为环形。若这些资源都不能利用，串行微端口将不能以 NDIS_STATUS_RESOURCES 为参数调用 MiniportSend 或 MiniportSendPackets，只能将数据包返回给 NDIS；或者在其内部对包进行排序，并返回 NDIS_STATUS_PENDING。非串行微端口或面向连接端口总是在内部对包排序并返回。
3. DMA 设备限制物理段数目的上限，这些段可映射到单个的 DMA 操作。若让微端口发送一个过于细小的数据片，它必须将连接到包的缓冲区拷贝到单一的发送缓冲区中。微端口应当从它的 MiniportInitialize 函数分配这一分段缓冲区和缓冲描述符，以便进行映射，微端口调用 **NdisMoveMemory** 从协议提供的缓冲区中将数据转移到这一分段缓冲区中。只有当包过于细小时，微端口才必须拷贝发送数据，因为拷贝到分段缓冲区是一项昂贵的操作，如果做的过于频繁会对运行性能会产生负面影响。一般而言，若待发送缓冲区小于 256 字节，微端口可通过将数据拷贝到分段缓冲区，并将微端口分配的描述符传给 **NdisMStartBufferPhysicalMapping** 来改进性能。
4. 微端口调用 **NdisMStartBufferPhysicalMapping** 映射以前分配的共享内存缓冲区，它的内容将被发送。这一调用产生一个布尔变量，WriteToDevice，它表明数据是否由主机内存转移到 NIC 或由 NIC 转移到主机内存。对于发送，WriteToDevice 设为 TRUE。微端口也必须提供用于映射缓冲区的映射寄存器的索引，而且这一索引必须是在微端口初始化期间调用 **NdisMAllocateMapRegisters** 返回的寄存器之一。
5. 若微端口为其发送缓冲区分配缓冲内存，它将调用 **NdisFlushBuffer**，为发送缓冲区提供地址并指定 WriteToDevice 参数为 TRUE，以表明传输方向是由主机到 NIC。**NdisFlushBuffer** 刷新处理器缓冲内存线，保证 NIC 所见的内存内容与微端口内容相同。因为高速缓存线破裂对于发送数据不构成影响，微端口可以刷新数据而不必涉及整个缓冲内存。为保证所有平台上缓冲内存的一致性，微端口也应调用 **NdisM-UpdateSharedMemory**。
6. 微端口为传输数据而对 NIC 编程，例如，写入控制寄存器，NIC 则进行 DMA 传输。
7. 当发送完成时，微端口调用 **NdisMCompleteBufferPhysicalMapping** 释放映射寄存器。如果 NIC 在发送完成时发生中断，那么这一调用由 MiniportHandleInterrupt 来执行。若 NIC 轮询，这一调用在时钟停止且发送结束时，由提供给 **NdisMSetPeriodTimer** 或 **NdisMSetTimer**（若端口不用周期时钟）的 MiniportTimer 函数来执行。
8. 微端口调用 NdisMSendComplete 或 NdisMCoSendComplete。

用 NdisMInitializeScatterGatherDma 初始化 DMA 资源的微端口。

一般而言，管理总线控制器 DMA NIC 的非串行微端口输出一个 MiniportSendPackets 函数以使其 NIC 达到最佳性能。管理总线控制器 DMA NIC 的面向连接微端口总是输出一个 MiniportCoSendPackets 函数。

在其 Miniport(Co)SendPackets 函数中接收一个或多个包描述符后，在 DMA 初始化期间，用 **NdisMInitializeScatterGatherDma** 初始化系统资源（见 3.2.2.3 节）的总线控制器 DMA 微端口在用 DMA 发送包时，执行下述步骤：

1. 微端口用 **ScatterGatherPacketInfo** 的一个 *InfoType* 调用 NDIS_PER_PACKET_INFO_

FROM_PACKET。NDIS_PER_PACKET_INFO_FROM_PACKET 将一指针返回给 SCATTER_GATHER_LIST 结构。此结构设定 SCATTER_GATHER_ELEMENT 结构的一个数组，数组的每一个值设定了 DMA 数据邻近范围的基本物理地址和长度，以便发送。

2. 微端口可能需要确定它是否有可供利用的发送缓冲区和可供利用的发送缓冲区描述符，这种描述符映射将缓冲区映射为环形。若这些资源都不能利用，非串行或面向连接的微端口在内部将包排序并返回。
3. DMA 设备具有物理段数目的上限，这些段可映射到单个的 DMA 操作。若让微端口发送一个过于细小的数据片，它必须将连接到包的缓冲区拷贝到单一的发送缓冲区中。微端口应由它的 *MiniportInitialize* 函数调用 **NdisMAllocateSharedMemory** 来分配这一分段缓冲区和缓冲描述符，以便进行映射。微端口调用 **NdisMoveMemory** 从协议提供的缓冲区将数据转移到这一分段缓冲区中。只有当包过于细小时，微端口才必须拷贝发送数据，因为拷贝到分段缓冲区是一项昂贵的操作，如果做的过于频繁会对运行性能产生负面影响。一般而言，若待发送缓冲区小于 256 字节，微端口可通过将数据拷贝到分段缓冲区来改进性能。
4. 微端口为其传输缓冲区分配缓冲内存，它调用 **NdisFlushBuffer**，来提供发送缓冲区的地址，并将 *WriteToDevice* 参数设为 TRUE 以指示传输的方向是从主机到 NIC，**NdisFlushBuffer** 刷新处理器缓冲内存线，保证 NIC 所见的内存内容与微端口内容相同。因高速缓存线破裂对于发送数据不构成影响，微端口可刷新数据而不必涉及整个缓冲内存。为保证所有平台上缓冲内存的一致性，微端口也应调用 **NdisMUpdateSharedMemory**。
5. 微端口为传输数据而给 NIC 编程，例如，写入控制寄存器，NIC 则进行 DMA 传输。
6. 微端口调用 **NdisMSendComplete** 或 **NdisCoSendComplete**。对 Ndis(Co)SendComplete 的调用释放用于 DMA 传输的映射资源。

4.4.5.2 在 PIO 设备上发送单包

对使用可编程 I/O（PIO）来发送数据的 NIC 来说，控制它的微端口在其发送函数中执行下列步骤：

1. 调用 **NdisQueryPacket** 得到包的长度，然后微端口检查是否提供有用于发送数据包的传输资源，如果没有，微端口返回 NDIS_STATUS_RESOURCE 或者内部将包排队，并从其发送处理程序返回。如果微端口中数据包排进了队列，微端口必须为这个包返回 NDIS_STATUS_PENDING。如果微端口支持带外（OOB）数据，它将使用 NDIS 宏从 OOB 块中读取任何有关的数据。如果提供了发送资源，NIC 驱动程序将继续。
2. 调用 **NdisQueryPacket** 得到第一个包描述符缓冲区的指针。
3. 调用 **NdisQueryBuffer** 获得包含待发送数据缓冲区的虚拟地址以及这个缓冲区的字节数。
4. 调用 **NdisRawWritePort***Xxx* 将缓冲区的长度写到端口上。管理 PIO NIC 的微端口保证将正确的数据传输给 NIC，所以，这样的微端口不调用 **NdisFlushBuffer**。
5. 调用 **NdisRawWritePortBuffer***Xxx* 写缓冲区——其地址是从 **NdisQueryBuffer** 返回的。
6. 调用 **NdisGetNextBuffer** 得到包的下一个缓冲区。
7. 重复 3 到 6，直到所有的包缓冲区都被成功地送出。
8. MiniportSend 总是把发送状态作为 MiniportSend 状态返回。如果这个状态不是

NDIS_STATUS_PENDING，微端口将包描述符和资源返回给调用者。串行微端口的 MiniportSendPackets 函数在包描述符的 OOB 块 **Send** 成员中返回发送的状态。如果微端口对任何一个包返回未决状态，在它使用包资源而且准备将包返回给调用者时，它必须调用 **NdisMSendComplete**。NDIS 库在它提交给非串行和面向连接微端口 Miniport(Co)SendPackets 函数的所有包描述符中忽略了 OOB 数据块。NDIS 假定这样的微端口（与 **NdisM(Co)SendComplete** 异步）将包描述符传递给 MiniportSendPackets 或 MiniportCoSendPackets 函数。因此，Miniport(Co)SendPackets 常常忽略 NDIS_PACKET_OOB_DATA 块的 **Status** 成员，但它可以将这一成员设置为某个状态，这个状态与后来传给 **NdisM(Co)SendComplete** 的状态相同。

4.4.5.3 使用板上内存发送包

对板上内存出现在主机的 I/O 空间的 NIC，支持它的微端口从 MiniportInitialize 中调用 **NdisMMapIoSpace**，以将适配器的板上内存映射到主机内存中，并给微端口一个可用于引用内存的虚拟地址。在这种类型的 NIC 上，微端口发送数据时，先将待发送的缓冲区转移到映射内存中，然后以一种设备独特的方式，调用 **NdisMRegisterIoRange** 写微端口声明的端口（或寄存器），这些动作将使得 NIC 把数据放到网络介质上。如果微端口缺乏发送资源，它就会或者将包排队，或者将包返回给 NDIS，就象以前所描述的那样。

如果微端口有可以利用的发送资源，它将调用：

1. **NdisQueryPacket**，得到包含待发送缓冲区的包的起始和长度。
2. **NdisQueryBuffer**，得到缓冲区描述符。
3. **NdisMoveToMappedMemory**，每次转移一个缓冲区到映射的适配器内存缓冲区。
4. **NdisGetNextBuffer**，从数据包中回收下一个缓冲区，然后再调用 **NdisMoveToMappedMemory**，重复这两个调用直到所有的缓冲区全部发送出去。
5. **NdisMRawWritePort***Xxx* 或 **NdisWriteRegister***Xxx* 引起设备特有方式发送的发生。

在开始拷贝之前，微端口用从 **NdisQueryPacket** 返回的长度来查明包含所有包缓冲区的板上内存是否有足够的空间。如果 NIC 内存对当前的发送没有足够的自由空间，串行微端口将返回一个 NDIS_STATUS_RESOURCE 状态，在这种情况下，NDIS 将包排队且在以后微端口指示有可供使用的资源时重新发送。从另一个方面来说，微端口也可以返回 NDIS_STATUS_PENDING，然后内部将包排队。在这种情况下，非串行微端口、面向连接的微端口以及 WAN 微端口必须返回 NDIS_STATUS_PENDING，内部对包进行排队。

从属（Slave）DMA 设备发送包

从属 DMA NIC 使用系统 DMA 控制器，将提供给发送处理程序（一般为 MiniportSend）的包发送到用于网络传输的 NIC 上，从属 DMA NIC 的微端口授权 NDIS 在 DMA 传输期间确保高速缓存的一致性。

对任何一个微端口有发送资源的数据包，从属 DMA MIC 的微端口进行下列步骤：

1. 调用 **NdisMSetupDmaTransfer** 建立用于发送的系统 DMA 控制器。微端口通过调用 **NdisMRegisterDmaChannel** 传递一个句柄给在微端口初始化期间分配的 DMA 通道，并传递一个指向包含待发送数据缓冲区的主机内存的指针，同时传递一个缓冲区的偏移量，并将数据的长度、指示传输方向的布尔值从主机传给设备。在 **NdisMSetupDmaTransfer** 返回中，控制器通过编程来传输数据，也包括确保缓冲区的高速缓存一致性。
2. 执行设备相关的步骤以引起传输（例如调用 **NdisRawWritePort***Xxx*）。

3. 当传输完成时，微端口调用 **NdisMCompleteDmaTransfer**。

一般来说，微端口为传输建立了完整的缓冲区。然而若缓冲区的大小大于 NIC 的 DMA 约束，微端口将用长度和偏移量参数来发送缓冲区的一段；如果待发送缓冲区小于 256 字节，微端口可通过将数据拷贝到分段缓冲区，并传递一个微端口分配的缓冲区包描述符到 **NdisMSetuoDmaTransfer** 以提高性能。

4.5 非串行微端口

通过对非串行微端口拥有的 Miniport*Xxx* 函数的操作进行串行化以及对内部队列接收到的发送包进行排序，非串行微端口就能较好地完成全双工工作。提高性能的代价是非串行微端口必须符合额外的、更严格的设计要求，非串行微端口也需要额外的调试和测试时间。

注意 NDIS 为非串行微端口保留了为接收包而打开适配器和过滤器的方法。

4.5.1 非串行微端口的 NDIS 要求

与 NDIS 接口的非串行微端口必须符合下列要求：

- 非串行微端口在初始化期间必须能象识别 NDIS 一样识别出自己。其 MiniportInitialize 函数必须用设置在 *AttributeFlag* 中的 NDIS_ATTRIBUTES_DESERIALIZE 标志调用 **NdisMSetAttributesEx**。
- 非串行微端口的 MiniportSendPackets 函数不能完全与多包传送同步，但它必须与接收包序列中每个包的 **NdisMSendComplete** 异步地完成处理每个包。从这一点来看，非串行微端口类似于面向连接的微端口，它总是用 **NdisMCoSendComplete** 来异步完成发送多包中的每一个包。
- NDIS 在它提交给非串行微端口 MiniportSendPackets 函数的所有包描述符中忽略了带外（OOB）数据块。因此，MiniportSendPackets 常常忽略 NDIS_PACKET_OOB_DATA 块的 **Status** 成员，但它可以将这一成员设置为某个状态，这个状态与后来传给 **NdisMSendComplete** 的状态相同。
- 如果非串行微端口不能立即传输它所接收到的数据包，它将不能把包返回给 NDIS 重新排队，而是微端口必须内部对它们排队，直到有足够的资源可用来传输数据包。非串行微端口的 MiniportSend 或者 MiniportSendPackets 函数不能返回 NDIS_STATUS_RESOURCE。

4.5.2 非串行微端口的驱动程序内部要求

非串行微端口必须符合下列的驱动程序内部要求：

- 非串行微端口必须用自旋锁保护其包队列，非串行微端口也应当保护其共享状态不被它所拥有的 Miniport*Xxx* 函数同时访问。关于使用自旋锁的更多信息，参见“*Kernel-Mode Drivers Design Guide*”。
- 非串行微端口的 Miniport*Xxx* 函数能运行在 IRQL<=DISPATCH_LEVEL 下，因此，驱动程序作者不能设想在它们处理的包序列中调用 Miniport*Xxx*，一个 Miniport*Xxx* 函数可以抢先于另一个运行于低 IRQL 下的 Miniport*Xxx*。
- 非串行微端口负责包的队列管理。当微端口察觉到一个资源问题时，它不能将发送包返回给 NDIS 去重新排队，而是将内入包在自己内部进行排队，等候有足够的资源时将它们发送出去。
- 非串行微端口必须以协议确定的次序来发送每一个包数组。也就是说，微端口必须在数组第一项中传输第一个包，在数组第二项中传输第二个包，如此等等。微端口

不能在给定的数组中对包进行重新排序。

4.6 接收数据

无连接的微端口使用下列方法之一给上层传递接收到的数据：

- 用一个指针调用 **NdisMIndicateReceivePacket**，这个指针指向一个或多个包含接收数据包的指针数组，微端口必须输出一个 MiniportReturnPacket 函数，用来处理接收协议返回的包。
- 用一个指向预先分配缓冲区的指针调用 **NdisM*Xxx*IndicateReceive**，有关的协议驱动程序会从缓冲区中拷贝数据。微端口必须输出一个 MiniportTransferData 函数，用来传输额外接收到的、不适合预先分配缓冲区的数据。

面向连接微端口必须调用 NdisMCoIndicateReceivePacket 来将接收到的数据传输给上层。

4.6.1 无连接和面向连接微端口的多包接收

要执行一个多包接收，微端口要分配和管理一个包描述符数组，用接收的数据填充链状缓冲区，然后通过函数调用，给有关协议驱动程序传输一个指针填充到包中，这个函数对无连接微端口是 NdisMIndicateReceivePacket，面向连接微端口是 NdisMIndicateReceivePacket。微端口应该为全部网络数据包而不是一个先行缓冲区分配和建立包描述符（见 3.2.2.1 节）。

在调用 **NdisMCoIndicateReceivePacket** 之前，面向连接的微端口总是要建立和激活一个在其上接收数据的虚连接（VC），当远程伙伴结束了接收调用时，微端口去活并有时是删除接收数据的 VC，关于 VC 创建、激活、去活以及删除的描述，参见 2.1.4 节。

图 4.8 说明了无连接微端口或面向连接微端口的多包接收

图 4.8 多包接收

微端口通过设置与每个包描述符相关的 OOB 数据块的成员，将状态（可能是包接收的时间、包发送的时间或其他介质特定信息）传递给上层，如果微端口允许协议保留数据包，并在以后将它返回给微端口的 MiniportReturnPacket 函数，则微端口必须将 OOB 块的 **Status** 成员设置为 NDIS_STATUS_SUCCESS。如果微端口要用尽包描述符或缓冲区，会强迫所有有关的上层驱动程序在 **NdisM(Co)IndicateReceivePacket** 返回之前拷贝包数据，此时微端口必须将 **Status** 成员设置为 NDIS_STATUS_RESOURCE，详情参见 4.3 节。

串行微端口必须在 NdisM(Co)IndicateReceivePacket 返回之前检查与每个包描述符有关的 OOB 块的 **Status** 成员：

- 如果 **Status** 是 NDIS_STATUS_PENDING 之外的任何一个，它所描述的包描述符和所有的资源将返回给微端口。
- 如果 **Status** 是 NDIS_STATUS_PENDING，微端口分配的包资源的所有权转给协议驱动程序或者中间协议驱动程序，直到包描述符返回给微端口的 MiniportReturnPacket 函数。
- 如果微端口在调用 **NdisM(Co)IndicateReceivePacket** 之前，将数组中每个包的 **Status** 成员设置为 NDIS_STATUS_RESOURCE，在高层驱动程序拷贝了指示包中的数据之后，包将返回给微端口，对所有这类包的 **Status** 成员将设置为 NDIS_STATUS_ SUCCESS。

非串行或面向连接的微端口驱动程序不必检查 **NdisMIndicateReceivePacket** 返回的指示包的 **Status**，而实际上，非串行微端口驱动程序在指示包描述符之前，将包的 **Status** 保存到一个本地变量。当 **NdisMIndicateReceivePacke**t 返回时，微端口驱动程序应当检查保存包的 **Status**。如果在指示包描述符之前微端口驱动程序将包的 **Status** 设置为 NDIS_STATUS_RESOURCES，它将在 NdisMIndicateReceivePacket 之后通过调用其自己的 MiniportReturnPacket 函数立即回收包描述符。在这种情况下，NDIS 不调用微端口驱动程序的 **MiniportReturnPacket** 函数来返回包描述符。如果微端口驱动程序在指示包描述符之前将包的 **Status** 设为 NDIS_STATUS_SUCCESS，直到 NDIS 将包描述符返回给微端口驱动程序的 MinportReturnPacket 函数，微端口驱动程序才收回包描述符。

管理总线控制器 DMA NIC 的微端口一般通过调用 Ndis(Co)IndicateReceivePacket 来指示包，因为其 NIC 通常有足够的链状缓冲区空间接收多个包。微端口通过一次处理几个包来提高性能。

通常，任何其他类型的网络接口卡（NIC），比如可编程 I/O（PIO）NIC 或适配器共享内存 NIC，在需要传递优先权、介质特定信息或者接收时间时，只调用 **NdisM(Co)-IndicateReceivePacket**。一个典型的非 DMA NIC 每次只接收一个包，在两个单次接收之间必须重置。这类设备的微端口每次只能用 NdisM(Co)IndicateReceivePacket 指示一个包。

因为微端口在它调用 **NdisM(Co)IndicateReceivePacket** 时传递出包的所有权，它必须管好接收缓冲区以确保网络上的新数据到达时有缓冲区可用。典型地，微端口在其 MiniportInitialize 函数中为链状缓冲区预分配了比所要求的多一些的缓冲区，分配了足够的缓冲区描述符以映射这些缓冲区和用来进行接收指示的包描述符。

在接收指示环境中处理返回包

当非串行或面向连接微端口用 **NdisM(Co)IndicateReceivePacket** 指示一个包时，有可能包在 **NdisM(Co)IndicateReceivePacket** 返回之前，返回给微端口的 MiniportReturnPacket 函数。下面就是微端口如何处理这种情形的一个例子：

1. 微端口在 NDIS_PACKET 的 **MiniportReserved** 成员中维护两个标志，其中一个称为 **InRcvDpc**，它指出指示线程是否处在微端口接收 DPC 的环境中；另一个标志，称为 **ReturnPacketCalled**，指出微端口的 MiniportReturnPacket 函数是否在接收指示环境中被调用。
2. 在调用 **Ndis(Co)IndicateReceivePacket** 指出一个包之前，微端口设置 **InRcvDpc** 为 TRUE，**ReturnPacketCalled** 为 FALSE，例如：
   ```
   NdisDpcAcquireSpinLock(&RcvLock);
   PacketReserved->fInRcvDpc=TRUE;
   PacketReserved->fReturnPacketCalled=FALSE;
   NdisDpcReleaseSpinLock(&RcvLock);
   ```
3. 微端口调用 **NdisM(Co)IndicateReceivePacket** 指示数据包。
4. **NdisM(Co)IndicateReceivePacket** 返回之后，微端口设置包的 **InRcvDpc** 标志为 FALSE，指示包不再处于接收指示的环境中。例如
   ```
   NdisDpcAcquireSpinLock(&RcvLock);
   Packet->fInRcvDpc=FALSE;
   NdisDpcReleaseSpinLock(&RcvLock);
   ```
5. 微端口然后检查返回状态和返回包的 **ReturnPacketCalled** 标志。如果其状态是 NDIS_STATUS_SUCCESS，NDIS_STATUS_RESOURCE 或者 NDIS_STATUS_

PENDING，而且 **ReturnPacketCalled** 为 TRUE，则表示微端口的 MiniportReturn-Packet 函数已经在接收指示环境中被调用了，微端口重新对它所用的包进行排队。（在这种情况下，MiniportReturnPacket 不对返回包重新排队。）例如：

```
Status = NDIS_GET_PACKET_STATUS(Packet);
If ((NDIS_STATUS_SUCCESS == Status) || (NDIS_STATUS_RESOURCE ==Status)
|| ((NDIS_STATUS_PENDING ==Status)&&Packet->fReturnPacketsCalled)
{
    ProcessReturnPacket(packet);
};
```

6. 在其每一个 MiniportReturn 函数中，微端口为每一个返回包测试 **InRcvDpc** 标志以检查包是否在接收指示环境被返回（也就是说，在接收指示完成以及微端口设置 I**nRcvDpc** 为 FALSE 之前）。如果 **InRcvDpc** 为 TRUE，微端口设置 **Return-PacketCalled** 为 TRUE 来指示协议已经完成对包的处理，微端口然后返回，以使得包能在接收指示环境中重新排队。例如：

```
NdisDpcAcquireSpinLock(&RcvLock);
If (Packet->fInRcvDpc)
{
    Packet->fReturnPacketsCalled=TRUE;
    NdisDprReleaseSpinLock(&RcvLock);
    Return;
}
NdisDpcReleaseSpinLock(&RcvLock);
ProcessReturnPacket(Packet);
```

如果 InRcvDpc 为 FALSE，表示在设备接收指示完成之后包被返回，MiniportReturnPacket 自己释放自旋锁，并对包重新排队。

处理资源问题

如果微端口检测到它运行于较少的接收缓冲区空间资源下，它将使用下列策略中的一个：

- 当它调用 **NdisM(Co)IndicateReceivePacket** 时，微端口将每一个不需要协议保留的包描述符的 OOB 块的 **Status** 成员设为 NDIS_STATUS_RESOURCE，这就告诉 NDIS，微端口不放弃指示包的所有权。NDIS 确保将包拷贝给协议驱动程序而不是传递给协议驱动程序。如果微端口是一个串行微端口，NDIS 调用 ProtocolReceive 函数而不是 ProtocolReceivePacket 函数，来强迫协议驱动程序拷贝包。如果微端口是一个非串行或面向连接的微端口，NDIS 调用协议驱动程序的 Protocol(Co)ReceivePacket 函数。当协议检测到 OOB 的 **Status** 成员设置为 NDIS_STATUS_ RESOURCE 时，它拷贝数据包。如果微端口支持所描述的多包接收范例，由于它不必提供 MiniportTransferData 函数，NDIS 将截取协议驱动程序的请求并执行传输。
- 微端口为其自由缓冲区维护一个“高水位”和一个“低水位”，当缓冲区的数目接近“低水位”时，微端口调用 **NdisMAllocateShareMemoryAsync** 分配更多的缓冲区，然后调用 **NdisAllocateBuffer** 为这些缓冲区分配缓冲区描述符。不象只能在

IRQL<DISPATCH_LEVEL 下调用的 **NdisMAllocateShareMemory**，**NdisM-AllocateShareMemoryAsync** 能在 IRQL DISPATCH_LEVEL 下被调用，即使是在 **MiniportHandleInterrupt** 函数中。当自由缓冲区集超过“高水位”时，微端口调用 **NdisMFreeShareMemory** 释放超出的缓冲区，然后调用 **NdisFreeBuffer** 释放映射到这些缓冲区的缓冲区描述符。

4.6.2 无连接微端口的单包接收

在初始化期间，无连接微端口的 MiniportQueryInformation 函数被调用来检测微端口或网络接口卡（NIC）的操作特征。这些特征中包含着微端口所指示的预先分配缓冲区的最小尺寸，微端口必须至少指示出这么多的数据量，但是如果 NIC 提供了有更多的数据，它也能指示出来。例如，由于 DMA 适配器将所有的数据直接传输进内存，它通常指出一个完整的接收缓冲区，这就使得协议驱动程序可能不必调用 **NdisTransferData** 得到其余数据，而是直接拷贝缓冲区。

在另一方面，管理可编程 IO（PIO）的无连接微端口传递一个最小预先分配缓冲区给协议驱动程序，其余数据直到微端口在 MiniportTransfer 函数调用微端口以前是不可读的。由于这种方式下传输数据比较慢，微端口对某些情况（例如没有协议接收包、以及调用 MiniportTransferData 请求最少协议数目等）进行了优化。

调用 **NdisM*Xxx*IndicateReceive** 的无连接微端口必须有一个 MiniportTransferData 函数，而且必须准备好被协议驱动程序调用，即拷贝内入的数据到一个链状缓冲区数据包中，MiniportTransferData 在微端口 **NdisM*Xxx*IndicateReceive** 调用的环境中被调用。在 **NdisM*Xxx*IndicateReceive** 返回控制之后，微端口能接收新的数据，也就是说，它能重新接收中断。

MiniportTransferData 传递一个标明单个缓冲区或缓冲区链的包描述符，微端口必须将收到的数据分配到这些缓冲区中，如果协议驱动程序提供的缓冲区不够，微端口将用一个适当的状态结束传输。然而由于微端口在其 **Ndis*Xxx*IndicateReceive** 调用中指出总的接收网络包的大小，协议驱动程序不能获得这个错误。如果所有的数据传给了 MiniportTransferData，数据的传输将同步地完成（参见图 4.9）。

图 4.9 无连接微端口的单包发送——同步完成

另外，MiniportTransferData 能返回 NDIS_STATUS_PENDING。如果微端口返回一个未决状态，在所有数据被传输之后，必须调用 **NdisMTransferDataComplete**（参见图 4.10）。而且微端口在数据传输完成前不能再中断，以使得已经接收的数据不会被新接收的覆盖掉。

图 4.10 无连接微端口的单包发送——异步完成

微端口必须周期性地调用 **NdisM*Xxx*ReceiveComplete**，这导致了 NDIS 调用每一个绑定协议驱动程序的 ProtocolReceiveComplete 函数，这个对 ProtocolReceiveComplete 的调用向协议驱动程序指示：微端口能处理已经接收到但是没有全部处理的数据。例如，微端口能为其客户指出接收的数据，微端口应当在每十个接收包之后，或者如果不足十个处理包而要退出 MiniportHandleInterrupt 之前，在其 MiniportHandleInterrupt 函数中调用 **NdisM*Xxx*ReceiveComplete**。

4.6.3 接收数据的高速缓存（Cache）因素

如果微端口分配了缓冲内存，它应当进行下列步骤：

- 生成一个 cache 定位的接收缓冲区。
- 在开始接收之前，调用 **NdisFlushBuffer** 和 **NdisMUpdateShareMemory** 刷新高速缓存。

管理总线控制器 DMA 设备的微端口将数据接收到与网络适配卡（NIC）共享的内存区间。微端口必须保证在它开始读数据之前，接收的数据处在共享内存，而不是在处理器内存之中。要完成这些工作，这种 DMA NIC 的微端口必须在指示接收或者其他引起数据读操作的因素之前，调用 **NdisFlushBuffer** 和 **NdisMUpdateShareMemory**（例如，在其 MiniportHandleInterrupt 函数中）。

微端口也能通过分配接收缓冲区的 cache 定位内存来防止高速缓存线断裂（cache-line tearing）。如果微端口分配了大块内存，然后重新将它为接收的数据分为较小的缓冲区，每一个接收缓冲区的开始必须是 cache 定位的，这样缓冲区才不会穿过高速缓存（cache）块，微端口在初始化期间调用 **NdisGetCacheFillSize**，在其适配器特定的环境区保存此值。微端口在分配这样的缓冲区时，使用这个值来计算接收缓冲区的大小，微端口也必须使用高速缓存块的大小计算在预分配内存的哪里开始分配接收缓冲区。

一般来说，总线控制器 DMA NIC 的微端口应当分配缓冲的共享内存，因为它是一种较快、更有用的资源。有两种情况微端口必须使用非缓冲内存：

- 如果管理 NIC 的微端口将连贯的数据包接收到连续的内存之中，一个包的结尾占用着下一个包开始的相同的缓冲块。因此，微端口必须分配非缓冲内存，因为第一个包的读取会造成第二个包开始的坏缓冲数据。
- 如果释放与写有关的缓冲区，例如接收缓冲区可以用一个包含所有权位的头连接起来，这个头用来表明缓冲区是属于适配器还是微端口的。在这种情况下，适配器能在释放这个缓冲区之前写这一位。使用这种技术的微端口必须从非缓冲内存中分配这些缓冲区为止。

4.6.4 接收数据的步骤

微端口怎样处理接收数据依赖于其所管理的 NIC 类型的一些问题。本节将讨论下列各个设备类型接收数据的处理过程：

- 总线控制器 DMA NIC
- PIO NIC
- 带板上内存的非 DMA NIC

对这些设备有一些公共的操作，它们包括：

- 微端口调用 **NdisRawWritePort***Xxx* 函数为其 NIC 编程。
- 指示一个缓冲区之前，指示一个完整包结构的微端口必须调用 **NdisAdjustBuffer-Length** 以调整描述缓冲区的缓冲区描述符的长度。微端口必须将缓冲区描述符的长度设为缓冲区中数据的长度。当包返回时，微端口必须再调用这个函数以将它重置为真实缓冲区的长度。
- 微端口不再启用更多的接收操作，直到当前数据已经被指示，并且一个新的缓冲区集已经被分配给接收环或者被有关的协议驱动程序拷贝为被指示的数据。

4.6.4.1 接收期间的包管理

一些微端口，比如以太网接口卡（NIC）的微端口，接收固定的、已知大小的帧。如果帧的大小总是相同的，微端口能用一个或多个链状缓冲区分配包，并能用相同包来进行接收。

然而，如果微端口控制着一个接收变长帧的 NIC，所需要处理接收的包的大小各不相同。如果微端口必须处理变长接收，它就必须进行下列步骤：

1. 对包调用 **Ndis(Un)ChainBufferAtBack** 或 **Ndis(Un)ChainBufferAtFront** 链接和短开包缓冲区，来建立较大的、并能将其改小的包指示。
2. 如果没有使用全部缓冲区，那么在指示数据包去设置链中最后一个缓冲区的缓冲区描述符大小之前，调用 **NdisAdjustBufferAtLength**。当包返回时，调用 **NdisAdjust-BufferLength** 将长度重置为缓冲区的真实长度。

从包描述符中拆除的缓冲区应当保存在一个空闲列表中，以确保它们没有“丢失”，并且当需要时它们可以再次使用。

4.6.4.2 在总线控制器 DMA NIC 上接收数据

当接收到一个或多个包时，微端口进行下列操作步骤：

1. 调用 **NdisFlushBuffer** 和 **NdisMUpdateShareMemory** 确保接收到的数据在微端口访问的内存区域中的一致性。
2. 调用 **NdisAdjustBufferLength** 调整缓冲区描述符域的大小（这个描述符域在指示之前映射网络接口卡（NIC）的接收/环状缓冲区）以匹配缓冲区中数据的实际长度。当包返回时，微端口必须再调用 **NdisAdjustBufferLength** 将接收/环状缓冲区回调到原来所分配的大小。
3. 如果微端口通过调用 **NdisMIndicateReceivePacket** 或 **NdisMCoIndicateReceive-Packet** 指示一个包数组，它必须设置与每个包描述符有关的 OOB 数据块的 **Status** 成员。如果微端口因为将要用尽接收缓冲区，而决定必须强迫协议驱动程序拷贝指示的数据时，它应当设置 OOB 块的 **Status** 成员为 NDIS_STATUS_RESOURCE。否则微端口将把 **Status** 成员设为 NDIS_STATUS_SUCCESS，NDIS_STATUS_ FAILURE，NDIS_STATUS_PENDING 或者由驱动程序确定的状态。
4. 如果无连接微端口通过调用 **NdisM*Xxx*IndicateReceive** 指示一个单包，那么微端口指示出所有接收数据，与原先大小一样指示出其整个大小。
5. 在包指示（第 3 步）或拷贝（第 4 步）之后，微端口代替环上的缓冲区描述符。缓冲区描述符被下列之一代替：
   - 一个先前分配，当前可用的包的缓冲区描述符。
   - 一个协议驱动程序强迫拷贝和返回的包的缓冲区描述符。
   - 一个通过调用 **NdisAllocateBuffer** 分配的缓冲区描述符，它是为另一个描述（调用 **NdisMAllocateShareMemoryAsync** 分配的）缓冲区的缓冲区描述符而使用的。

接收变长帧的 NIC 的微端口必须对帧进行链接和拆链，依靠接收数据的数量来构造不同大小的帧。一些微端口在使用固定长度帧的介质上接收数据，这种微端口能立即分配固定大小的缓冲区，将它们链接进包描述符，不必进行链接和拆除缓冲区就能使用它。

如果微端口调用 **NdisMAllocateSharedMemoryAsync** 来分配缓冲区，在分配完成时，将调用 *MiniportAllocateComplete* 函数。

4.6.4.3 在 PIO NIC 上接收数据

可编程 I/O 网络接口卡的微端口通过调用 **NdisRawReadPort*Xxx*** 和使用先前由 **NdisMRegisterIoPortRange** 声明的端口，从 NIC 端口读取数据。具有代表性的这样的一个设备通过调用具有过滤器特性的 **Ndis*Xxx*IndicateReceive** 函数，在 *MiniportTransferData* 函

数中向有关的协议驱动程序传输剩余部分数据，将接收包的预先分配部分指示给绑定协议驱动程序。在数据传输之后，接收中断重新启用，数据就可以被新的数据所覆盖了。

PIO NIC 的微端口不应因当前传输到上层协议驱动程序的时间延迟，而导致新的数据丢失。如果 NIC 有相对较小的板上 FIFO 缓冲区，微端口能在其 MiniportInitialize 函数预分配一个缓冲区，然后将接收数据转移到所指示的缓冲区，转移的过程试图保持 FIFO 尽可能地空，以利于接收新的数据。

换句话说，当调用微端口的 MiniportTransferData 函数时，微端口能指示数据直接从 NIC 端口拷贝，直接从 NIC 移动接收的剩余数据。这种接收技术招致了一个时间延迟——它等价于调用 **Ndis*Xxx*IndicateReceive** 指示接收数据到调用 MiniportTransferData，将接收的数据移进 MiniportTransferData 提供的缓冲区之间的时间。

如果微端口调用 **NdisMIndicateReceivePacket** 或者 **NdisMCoIndicateReceivePacket**，在它作调用之前，它将数据传输到链接到预分配包的缓冲区中。

4.6.4.4 在内存映射设备上接收数据

管理带板上内存的 NIC 的微端口通常直接从 NIC 将 *MiniportTransferData* 接收的数据传输到调用 MiniportTransferData 提供的缓冲区中。象可编程 I/O（PIO）NIC 微端口一样，如果因为直接从 NIC 向协议驱动程序提供的缓冲区传输数据而引起等待时间，使得可能有新传来的数据丢失，那么微端口就能进行这样的接收。

微端口调用 **NdisM*Xxx*IndicateReceive** 指出等价于预先分配数据大小的数据量，或者如果缓冲区较小，而且微端口也已经完成了这个缓冲区，它将指示出整个缓冲区。MiniportTransferData 调用 **NdisMoveFromMappedMemory** 从 NIC 的板上内存传输数据到协议驱动程序包或缓冲区中。数据被传输之后，微端口再启用中断，以使新的数据能被接收。

如果微端口调用 **NdisMIndicateReceivePacket** 或 **NdisMCoIndicateReceivePacket**，那么在调用之前，它将数据传输到链接至预分配包的缓冲区中。

4.7 保持统计量

每一个微端口网络接口卡（NIC）驱动程序应当维护强制性常规统计量和介质特性的统计对象标识符（OID）。关于更多的微端口处理查询和设置 OID 的信息，参见第五章。

无连接微端口的统计量

一般应用于所有无连接微端口 NIC 驱动程序的强制性统计量包括下列内容：

OID_GEN_XMIT_OK

OID_GEN_RCV_OK

OID_GEN_XMIT_ERROR

OID_GEN_RCV_ERROR

OID_GEN_RCV_NO_BUFFER

当接收到这样的一个 OID 查询时，MiniportQueryInformation 函数必须返回这些统计量的每一个积累的对应值。

面向连接微端口的统计量

一般应用于所有面向连接微端口 NIC 驱动程序的强制性统计量包括下列内容：

OID_GEN_CO_XMIT_PDUS_OK

OID_GEN_CO_RCV_PDUS_OK

OID_GEN_CO_XMIT_PDUS_ERROR

OID_GEN_CO_RCV_PDUS_ERROR

OID_GEN_CO_RCV_PDUS_NO_BUFFER

当接收到这样的一个 OID 查询时，MiniportCoRequest 函数必须返回它为这些统计量的每一个积累的对应值。

4.8 802.1P 包的优先权

802.1P 包的优先权是网络包中标准介质访问控制（MAC）头的一个扩展。在 802 网络介质中，这个扩展包含集线器和交换机用于建立包优先权的 3 比特的值。一般来说，当网络段拥塞时，集线器及交换机的负载导致了包的时间延迟或丢帧。在使用 802.1p MAC 扩展的网络上，高优先权的包优先接受处理，并且在它变成低优先级之前为它提供服务。

操作系统中负责服务质量（QoS）的部件给微端口提供 802.1p 优先权信息。描述每一个传输包的 NDIS_PACKET 结构用于发送优先权信息。QOS 部件通过映射服务类型到 IEEE 802.1p 优先级值来得到这个优先权信息。Internet 工程任务组（IETF）的集成服务工作组（Intserv）定义了这种服务类型的映射。这些 802.1p 优先权值的 Intserv 规范，也称为“用户优先权”对象，放在 IETF 的 Internet 站点上，在 http://search.ietf.org 执行下列步骤可以找到这些规范：

1. 单击 Internet 图标（Internet-Draft）
2. 单击 Internet 图标索引（Internet-Draft Index）
3. 单击特殊连接层的集成服务（Integrated Service over Specific Link Layers：issll）
4. 单击“IEEE 802 网络上的集成服务映射”规范(“Integrated Service Mappings on IEEE 802 Networks”specification）

注意 IETF 有可能用较新的草案或最终的规范取代这个规范。尽管 IEEE 802.1q 规范描述了怎样标明包的网络优先性，这个规范还是使用 IEEE802.1p 优先权值描述了带优先权的包。QOS 应用程序或者代表应用程序工作的 QOS 操作系统部件，能确定映射的 *Intserv* 服务类型。

除非主机正确地与网络协商了 QOS，主机将只标明用“最好的”802.1p 优先级值传输的包。如果主机有一个已经安装的包方案，主机将使用适当的 QOS 信号部件与高 802.1p 优先权的网络协商，包方案然后在 NDIS 包中传递一个适当的优先值给微端口驱动程序。

支持 802.1p 包优先级的微端口驱动程序使用从 NDIS_PACKET 得到的优先级值，然后在通过网络传输包的 MAC 头中生成对应的成员。这些微端口驱动程序也从接收包的 MAC 头中取出适当的信息。微端口驱动程序向高协议层指示包之前，优先值拷贝到 NDIS_PACKET 结构中。IEEE 802.1q 规范描述了包怎样标明特定的网络优先性。

注意到上层的驱动程序有解释 NDIS_PACKET 结构的优先信息的能力。这种能力可以使得这些驱动程序对特殊的介质使用适当的优先性信息。

支持 802.3 优先权的特性包含在 Windows NT4.0 的 NDIS 4.0 以及 Windows 95 及其后续版本之中，当 802.3 开始用于 NDIS 时，它用于 PACE 和 100-VG 的环境中。指定这个优先权的成员位于 NDIS 包描述符的 OOB 数据部分（NDIS_PACKET_OOB_DATA），这些成员仍然存在，但它们不使用 802.1p 优先权值。

下列章节将描述怎样实现一个支持 802.1p 包优先权的微端口驱动程序：

- 4.8.1 查询 802.1p 优先权支持
- 4.8.2 802.1 优先权的包支持
- 4.8.3 为发送和接收指定包的大小
- 4.8.4 默认时禁止 802.1p 优先权支持

4.8.1 查询 802.1p 优先权支持

上层驱动程序向下层 NIC 微端口驱动程序发送一个查询，来检测这些微端口驱动程序或其 NIC 的能力。如果微端口驱动程序支持 802.1p 包优先权，它们将完成这些查询操作，返回所指示的支持信息。上层驱动程序对无连接微端口调用 **NdisRequest** 函数，对面向连接微端口调用 NdisCoRequest，向查询返回一个填充着下列 OID_*XXX* 代码的 NDIS_REQUEST 结构。

OID_GEN_MAC_OPTION

OID_GEN_MAC_OPTION 返回一个位掩码给上层驱动程序，位掩码定义了下层驱动程序或微端口 NIC 的可选的性质。发出查询的协议能确定出位掩码中哪一位是由下层微端口驱动程序设置的，这样的协议也利用了微端口与这一位有关的性质。例如，位掩码设置为 NDIS_MAC_OPTION_8021P_PRIORITY 标志的微端口驱动程序，能从高层驱动程序的 NDIS_PACKET 结构中接收 802.1p 优先权值，然后在网络传输包的 MAC 头中生成适当地信息，这个微端口驱动程序也支持从网络上接收的包的 MAC 头中提取适当的信息。信息被提取之后，微端口驱动程序使用 NDIS_PACKET 结构向高层驱动程序指出信息。如果微端口驱动程序或微端口 NIC 支持其他可选的性质，微端口驱动程序也将在位掩码中为这些性质设置标志。Windows 2000 在线 DDK《*Network Drivers Reference*》中定义了 OID_GEN_MAC_OPTION 的所有位掩码标志。

4.8.2 802.1 优先权的包支持

上层驱动程序发送包给下层的 NIC 微端口驱动程序，微端口驱动程序也接收包，并将它们指示给连接层驱动程序。为了支持 802.1p 优先权，微端口驱动程序必须能访问 NDIS_PACKET 传输的优先权信息。

支持 802.1p 包优先权的微端口驱动程序从网络接收包含着优先权值的数据包，微端口驱动程序从包的 MAC 头中提取有关的信息，然后微端口驱动程序在将包指示给高层协议之前，在包描述符中插入优先权的信息。要插入一个 802.1p 优先权值到一个包描述符中，微端口驱动程序必须调用 NDIS_PER_PACKET_INFO_FROM_PACKET 宏，在这个调用中，微端口驱动程序传递一个指针到描述包的 NDIS_PACKET 中，指示这个 NDIS 的 **Ieee8021pPriority** NDIS_PER_PACKET_INFO 的值应当在包中插入 802.1p 优先权值。例如，在下列操作中，*pPacketDesc* 变量指向一个包，*UserPriority* 变量保持着插入到这个包的优先权值。要将优先权值插入包中，微端口驱动程序必须执行下列操作：

```
NDIS_PER_PACKET_INFO_FROM_PACKET(pPacketDesc,Ieee8021pPriority) =
                                        (PVOID)(ULONG_PTR)UserPriority
```

支持 802.1p 包优先权的微端口驱动程序在从上层传下的包中得到优先权值之后，这些微端口驱动程序应当在网络传输的包的 MAC 头中生成相应的成员。要从包描述符中得到 802.1p 优先权值，微端口驱动程序调用 NDIS_PER_PACKET_INFO_FROM_PACKET 宏，在这个调用中，微端口驱动程序传递一个指针到描述包的 NDIS_PACKET 中。在这个调用中，微端口驱动程序也传递一个指示这个 NDIS 的 **Ieee8021pPriority** NDIS_PER_PACKET_INFO 的值，这个值应当是从包中得到的 802.1p 优先权值。例如，为了从包含在由 *pPacketDesc* 变量指示的包中回收优先权值，然后将优先权值分配给 *UserPriority* 变量，微端口驱动程序必须执行下列操作：

```
UserPriority = (IEEE8021PPRIORITY)
    NDIS_PER_PACKET_INFO_FROM_PACKET(pPacketDesc,Ieee8021pPriority)
```

注意 NDIS 定义 IEEE8021PPRIORIY 数据类型为 UINT。

如果从 NDIS_PACKET 回收的优先权值为 0，微端口驱动程序在通过网络传输包之前，

不应当用 802.1p 优先权值标识这个包。在以太网（802.3）中，不标识的包就象用“最高”优先权值标明的包一样。不标识包是为了保证与网络上与支持 802.1p MAC 扩展的设备向后兼容。

如果微端口驱动程序从网络上接收到的 MAC 包头中含有“最高”优先权，那么在将它指示给高层驱动程序之前，微端口在其 NDIS_PACKET 中插入 0。

注意微端口驱动程序必须使用 NDIS_PER_PACKET_INFO_FROM_PACKET 插入或取得报包的 802.1p 优先权值。微端口驱动程序不必使用 NDIS_PACKET_OOB_DATA 结构中 **MediaSpecificInformation** 成员的 **NdisClass802_3Priority** NDIS_CLASS_ID 值，来插入或取得 802.1p 优先权值，NDIS_PACKET_OOB_DATA 结构包含着与包描述符有关的 OOB 信息。

4.8.3 为发送和接收指定包的大小

4.8.1 节所描述的确定支持 802.1p 包优先权的微端口驱动程序可以对上层协议发送给它的包设定不同的大小，但对通过网络接收的包不能，而且随后会将它们指示给上层驱动程序，步骤如下：

- 微端口驱动程序可以约束上层驱动程序发送给它们的包的大小。例如，微端口驱动程序可以限制上层驱动程序发送的包最大为 4 字节，所以它可以给这些包追加适当的优先权值，然后通过早期的网络将这些包传输出去。在另一方面，传输的包没有预料到会穿越早期的网络设备，微端口驱动程序将不要求上层驱动程序发送的包最大为 4 字节。微端口驱动程序限制包的最大尺寸为 4 字节，因为这是 IETF 定义的 802.1p 优先权值的大小。微端口驱动程序使用 OID_GEN_MAXIMUM_FRAME_SIZE 来确定其 NIC 支持接收上层驱动程序传来的包的最大尺寸。
- 微端口驱动程序必须确定它们从网络接收到的，然后指示给上层驱动程序的包的最大尺寸。即使微端口驱动程序在将这些通过网络接收到的包指示给上层驱动程序之前，去掉了它们的 4 字节优先权值，它也必须确定所有的可能指示给上层驱动程序的包的最大尺度。例如，微端口驱动程序可以接收没有标明优先权值的但仍然是下层介质支持的最大尺寸的网络包。微端口驱动程序使用 OID_GEN_MAXIMUM_TOTAL_SIZE 来确定其通过网络 NIC 接收到的包的最大尺寸。

4.8.4 默认情况下禁止 802.1p 的优先权支持

微端口驱动程序应当支持 NIC 在默认情况下禁止 802.1p 优先权的性质。不能在所有网络环境中传输特定的带 802.1p 优先权的包，当前也没有它所需要的标准检测机制。在默认情况下，微端口驱动程序必须禁止 802.1p 的优先权支持。但应当能够对微端口驱动程序进行设置，用户可以通过控制面板应用中的“Network”来启用 802.1p 优先权支持。

第五章 获取和设置 WMI 的微端口信息及 NDIS 支持

本章描述上层及 NDIS 怎样查询以及设置微端口信息，微端口怎样对上层及支持 Windows 管理设备（WMI）的 NDIS 报告硬件状态的变化。

这一章包含下列内容：

- 5.1 NDIS 管理信息和 OID
- 5.2 查询微端口信息
- 5.3 设置微端口信息
- 5.4 报告硬件状态
- 5.5 NDIS 对 WMI 的支持

5.1 NDIS 管理信息和 OID

每一个 MIC 微端口驱动程序都包含着它自己的 MIB，这是一个由驱动程序存储动态配置的信息和静态信息，并且它也可以是由管理实体查询或设置的信息块。以太网的广播地址列表是配置信息的一个例子，接收到的广播包数目是静态信息的一个例子。MIB 中每一个信息元素都被一个对象（*Object*）所引用。要引用这样的管理对象，NDIS 定义了一个对象标识符（*Object Identifier*）。所以，如果管理实体要查询或设置一个特殊的管理对象，它必须为这个对象提供一个特定的 OID。

MIB 包含三种对象：

- 那些对所有 NDIS NIC 微端口通用的对象
- 那些 NDIS NIC 微端口为所有特定的介质类型（如以太网或令牌环网）而提供的特殊对象
- 那些厂商实现的特定对象

那些通用的和强制性介质相关的 OID 在 DDK 的在线文档“*Network Drivers Reference*（网络驱动程序参考）”中组织说明，为特殊的 NIC 驱动程序实现特定的 OID 在驱动程序的文档中列示并且给予了说明。

除了被分为操作特性（*operational characteristics*）（例如广播地址列表）和统计量（*statistics*）（例如接收广播数据包）之外，对象还被分为必须的（*mandatory*）和可选的（*optional*）。对一般或介质特定类的所有操作特征的对象都是必须的，但只有一些统计性对象是必须的，所有的实现特性对象都被划分为必须的分类。

图 5.1 显示了 OID 的结构。

图 5.1 NDIS OID 的格式

OID 的前 3 个字节对 OID 的各种分类提供了主键。第 4 个字节标明了这种分类的特殊信息管理对象。

关于所有 NDIS 一般的和介质相关的 OID 的列表及其描述，参见 DDK 在线“*Network Drivers Reference*”。

5.2 查询微端口信息

协议驱动程序调用 **Ndis***Xxx* 函数来查询微端口信息，就象用来执行这个查询的 Miniport*Xxx* 一样，**Ndis***Xxx* 函数也依赖于微端口是无连接的还是面向连接微端口的。

5.2.1 无连接微端口的查询

要查询由无连接微端口维护的 OID，绑定协议调用 **NdisRequest**，传递一个标明了查询对象的 NDIS_REQUEST 类型的结构，它指向一个 NDIS 最终用来写请求信息的缓冲区。对 **NdisRequest** 的调用引起 NDIS 调用微端口的 MiniportQueryInformation 函数，它向 NDIS 返回请求信息。调用 *MiniportQueryInformation* 可以与调用 NdisMQueyInformationComplete 函数同步（或者异步）地完成（见图 5.2）。

图 5.2　无连接微端口的查询

NDIS 也能自己调用微端口的 MiniportQueryInformation 函数来查询微端口的能力，状态，或者统计信息。（例如，在微端口的 MiniportInitialize 函数返回 NDIS_STATUS_SUCCESS 之后。）

无连接微端口必须的通用 OID 包括：

OID_GEN_SUPPORTED_LIST
OID_GEN_HARDWARE_STATUS
OID_GEN_MEDIA_SUPPORTED
OID_GEN_MEDIA_IN_USE
OID_GEN_MAXIMUM_LOOKAHEAD
OID_GEN_MAXIMUM_FRAME_SIZE
OID_GEN_LINK_SPEED
OID_GEN_TRANSMIT_BUFFER_SPACE
OID_GEN_RECEIVE_BLOCK_SIZE
OID_GEN_VENDOR_ID
OID_GEN_VENDOR_DESRIPTION
OID_GEN_VENDOR_DRIVER_VERSION
OID_GEN_CURRENT_PACKET_FILTER
OID_GEN_CURRENT_LOOKHEAD
OID_GEN_DRIVER_VERSION
OID_GEN_MAXIMUM_TOTAL_SIZE
OID_GEN_MAC_OPTIONS
OID_GEN_MEDIA_CONNECT_STATUS
OID_GEN_MAXIMUM_SEND_PACKET

特别地，微端口的 MiniportQueryInformation 函数必须准备对 OID_GEN_MAXIMUM_LOOKAHEAD，OID_GEN_MAXIMUM_SEND_PACKETS 和 OID_GEN_MAC_OPTIONS 做出响应。

如果微端口通过调用 **Ndis*Xxx*IndicateReceive** 指示接收数据，它应当用 NIC 所能提供的先行数据的最大字节数来应答 OID_GEN_MAXIMUM_LOOKAHEAD。如果绑定协议支持的先行缓冲区大小不一，NDIS 将调用 MiniportSetInformation 设置微端口支持的预先缓冲区的大小为微端口和协议值的最小值。

如果驱动程序总是用 **NdisMIndicateReceivePacket** 指出完整的包，它将把这个值设置为整个包除去包头外的最大尺寸。

如果微端口注册了 MiniportSendPackets 函数，NDIS_GEN_MAXIMUM_SEND_PACKETS 将调用 MiniportQueryInformation 函数，微端口必须回答准备用于处理单包发送请求的最大包数目。因为没有资源（其设备被占满），微端口应当挑选一个这样的最大值，使

得返回 NDIS 或内部进行排队的包数目最少。总线控制器 DMA NIC 的微端口应当试图挑选一个使其 NIC 工作于预期负载下的值。

当 MinioprtQueryInformation 被 OID_GEN_MAC_OPTIONS 调用时，它必须返回一个标明微端口执行的可选操作的位掩码，标志集包括：

- NDIS_MAC_OPTION_COPY_LOOKAHEAD_DATA

指示空闲的协议驱动程序以任意方式访问指示的数据，如果微端口指示了超出板上共享内存范围的数据，它不必设置此标志。

- NDIS_MAC_OPTION_NO_LOOPBACK

如果进行了设置，则表示微端口不回送传输给 MiniportSend 的包——它直接传输到本机的接收器上，微端口希望 NDIS 执行这个回送。如果 NDIS 执行了包的回送，包不传递到微端口。微端口总是设置这个标志，除非其 NIC 执行了硬件回送。

- NDIS_MAC_OPTION_RECEIVE_SERIALIZED

如果设置了，那么微端口在前面接收到的包被处理完之前，不指示任何新的接收包，包括传输数据。大多数微端口，除了那些通过调用 **NdisMIndicateReceivePacket** 指示包之外，都要设置这一位。

- NDIS_MAC_OPTION_TRANSFERS_NOT_PEND

如果设置了，微端口永远不从 MiniportTransferData 返回 NDIS_STATUS_PENDING。

微端口必须永远不使用 NDIS 为其内部使用而保留的 NDIS_MAC_OPTION_RESERVED 标志。

MiniportQueryInformation 也被介质相关的 OID 查询以确定 NIC 的当前地址。例如，802.3 类型 NIC 的微端口用 OID_802_3_CURRENT_ADDRESS 来查询。

确定介质类型的微端口将接收介质相关的带外 OID。例如，802.3 类型 NIC 的微端口用 OID_802_3_MAXIMUM_LIST_SIZE 来查询。关于一般操作的和介质相关 OID 的信息的列表和描述，参见 DDK 在线的“*Network Drivers Reference*”。

5.2.2 面向连接微端口的查询

要查询由面向连接微端口维护的信息对象，绑定协议调用 **NdisCoRequest**，传递一个标明被查询对象的 NDIS_REQUEST 类型的结构，它提供一个 NDIS 最终可进行写请求信息的缓冲区。对 **NdisCoRequest** 的调用引起 NDIS 调用微端口的 *MiniportCoRequest* 函数，它对 NDIS 返回请求信息。MiniportCoRequest 与一个对 **NdisMCoRequestComplete** 的调用同步或异步地完成（见图 5.3）。

图 5.3 面向连接微端口的查询

NDIS 也能自己调用微端口的 MiniportCoRequest 函数来查询微端口的能力，状态，或者统计信息，例如，在微端口的 MiniportInitialize 函数返回 NDIS_STATUS_SUCCESS 之后。

面向连接微端口必须能为 NIC 特定的所有 VC（虚连接）提供全局及部分（perVC）的基础信息，例如，如果给 MiniportCoRequest 提供一个非空的 NdisVcHandle 用来查询 OID_GEN_CO_RCV_CRC_ERROR，那么微端口返回特定 VC 所有接收到的 CRC 错误数目。对一个用空 NdisVcHandle 的查询，微端口返回其 NIC 所有接收到的 CRC 错误数目。

面向连接微端口的一般操作 OID 集包括：

OID_GEN_SUPPORTED_LIST

OID_GEN_HARDWARE_STATUS

OID_GEN_MEDIA_SUPPORTED
OID_GEN_MEDIA_IN_USE
OID_GEN_LINK_SPEED
OID_GEN_VENDOR_ID
OID_GEN_VENDOR_DESRIPTION
OID_GEN_VENDOR_DRIVER_VERSION
OID_GEN_DRIVER_VERSION
OID_GEN_MAC_OPTIONS
OID_GEN_MEDIA_CONNECT_STATUS
OID_GEN_MINIMUM_SEND_PACKET

微端口的 MiniportCoRequest 函数必须准备对上述 OID 中任何一个的查询或设置做出响应。

当 OID_GEN_CO_MAC_OPTIONS 调用 MinioprtCoRequest 时，它必须返回一个标明微端口执行的可选操作的位掩码，标志集包括：

- NDIS_MAC_OPTION_NO_LOOPBACK

如果设置了，表示微端口不回送传输给 MiniportSend 的包——它直接传输到本机的接收器上，微端口希望 NDIS 执行这个回送。如果 NDIS 执行了包的回送，包不传递到微端口。微端口总是设置这个标志，除非其 NIC 执行了硬件回送。

- NDIS_MAC_ETOX_INDICATION

如果设置了，表示微端口只在包通过 NIC 传输之后指示发送完成。

微端口必须从不使用 NDIS 为其内部使用而保留的 NDIS_MAC_OPTION_ RESERVED 标志。

MiniportCoQequest 也被介质相关的 OID 查询以确定 NIC 的当前地址。例如，ATM 类型 NIC 的微端口用 OID_ATM_HW_CURRENT_ADDRESS 来查询。

关于面向连接微端口的一般操作和介质特定 OID 信息的列表和描述，参见 DDK 在线的“*Network Drivers Reference*”。

5.3 设置微端口信息

协议驱动程序调用 **Ndis***Xxx* 函数来设置微端口信息，就象用来进行一个设置操作的 Miniport*Xxx* 函数一样，根据微端口是无连接的或面向连接的不同而不同。

5.3.1 为无连接微端口设置信息

要设置一个由无连接微端口维护的 OID，绑定协议调用 **NdisRequest**，并且传递一个标明查询对象（OID）、且指向一个包含要设置对象值的缓冲区的 NDIS_REQUEST 类型的数据结构。对 **NdisRequest** 的调用引起 NDIS 调用微端口的 *MiniportSetInformation* 函数，它用所提供的值设置对象。

MiniportSetInformation 的调用可以同步或异步地完成。要异步地完成这个调用，微端口调用 **NdisMSetInformationComplete**。

图 5.4 为无连接微端口设置信息

5.3.2 为面向连接微端口设置信息

要设置一个由面向连接微端口维护的 OID，绑定协议调用 **NdisCoRequest**，传递一个标明被查询对象（OID）、且指向一个包含要设置对象值缓冲区的 NDIS_REQUEST 类型的数据结构。对 **NdisCoRequest** 的调用引起 NDIS 调用微端口的 *MiniportCoRequest* 函数，它用所提供的值设置对象。

MiniportCoRequest 的调用可以同步或异步地完成。要异步地完成这个调用，微端口调用 **NdisCoRequestComplete**。

图 5.5　为面向连接微端口设置信息

5.3.3　设置微端口信息的时机

除了在初始化期间被调用之外，无连接微端口的 *MinportSetInformation* 函数或面向连接微端口的 *MinportCoRequest* 函数在下列两种情况下被调用：

- 硬件重置期间
- 作为协议调用 **NdisCloseAdapter** 的结果

MinportSetInformation 或 MinportCoRequest 在硬件初始化操作期间的调用，参见 7.1 节。在这种情况下，调用 MinportSetInformation 或 MinportCoRequest 来将微端口重置为关于其地址的初始状态。

当协议调用微端口的 NdisCloseAdapter 来关闭 NIC 时，NDIS 也会调用 MinportSet-Information 或 MinportCoRequest。这样要求微端口更新其地址信息。

5.4　报告硬件状态

无连接微端口 NIC 驱动程序能通过调用 **NdisMIndicateStatus** 对上层指示硬件状态的变化，面向连接微端口则使用 **NdisMCoIndicateStatus** 来指示。

NdisM(Co)IndicateStatus 取得一个一般状态码和一个包含将来定义的用于标识状态改变原因的介质信息缓冲区。NDIS 给绑定协议驱动程序报告状态的变化，NDIS 不解释或中途截取这些状态代码。微端口 NIC 驱动程序可以发出一个或多个这样的调用，然而当一个微端口完成其发送状态时，它必须调用 **NdisMIndicateStatusComplete** 以使得有关的协议驱动程序知道已经报告了的状态；面向连接微端口不必指出它已经完成了发送状态。协议驱动程序或者配置管理器能记录其状态，校正其行为。

NdisMCoIndicateStatus 任取一个有效的 NDIS_STATUS_*XXX* 值。微端口 NIC 驱动程序负责指示协议或高层驱动程序状态码的含义。然而，协议驱动程序忽略了所有没有意义或在其操作环境中不可理解的状态值。

例如，微端口驱动程序使用 NDIS_STATUS_RING_STATUS 来报告环失败。**NdisMCo-IndicateStatus** 的 *StatusBuffer* 参数将详细说明环的状态，*StatusBuffer* 的内容可能包含表明状态报告原因的环状态集合，比如，NDIS_RING_SIGNAL_LOSS 和 NDI_RING_HARD_ ERROR。

上层驱动程序或 NDIS 可以向微端口 NIC 驱动程序查询微端口的硬件状态。当无连接微端口的 MiniportQueryInformation 函数或面向连接微端口的 MiniportCoRequest 函数接收到 OID_GEN_HARDWARE_STATUS 时，它用任何一个在 NDIS_HARDWARE_STATUS 中定义的硬件状态来回答，这些硬件状态包括：

- **NdisHardwareStatusReady**
- NdisHardwareStatusInitializing
- NdisHardwareStatusReset

- NdisHardwareStatusClosing
- NdisHardwareStatusNotReady

微端口能被查询，以至于 NDIS 可以在 NDIS 层之间同步地操作，例如，确定 NIC 是否已准备好接收包。

5.5 WMI 的 NDIS 支持

通过 NDIS，WMI 的客户可以查询或设置 NDIS 维护的 OID，WMI 客户也能注册用来接收微端口状态的指示。

NDIS 自动用 WMI 注册微端口和命名 VC，就象每一个微端口 OID 的标准集一样（参见 5.6.5 节）。象在 5.6.7 节中描述的一样，微端口能提供对定制 OID 和定制状态指示的支持。

NDIS 不为协议驱动程序提供 WMI 支持。协议，比如中间层驱动程序，能为它自己创建一个设备对象，并直接用 WMI 注册它。

关于 WMI 体系结构以及微端口驱动程序怎样适应这个体系的更多信息，参见 DDK 在线“*Kernel-Mode Drivers Design Guide*”。

5.5.1 用 WMI 注册与注销 NDIS 微端口

NDIS 自动向 WMI 注册微端口的每一个实例（即，每一个功能设备对象都是由 NDIS 响应一个成功的 MiniportInitialize 返回而创建）。微端口实例被注册为一个 WMI 提供的数据，WMI 客户能发送它的查询，设置以及注册接收微端口状态的指示。微端口不必明确地用 WMI 注册，微端口初始化之后，NDIS 将自动地为微端口进行这些步骤。

当微端口卸载时（即微端口的 *MiniportHalt* 函数返回之后），NDIS 自动向 WMI 注销微端口实例，使得 WMI 不再对微端口发送查询或设置命令。

对每一个向 WMI 注册的微端口实例，NDIS 注册每一个对应于特殊 OID 或微端口状态指示的 GUID（全局统一标识符）。对每一个微端口，NDIS 注册 GUID 为一个标准查询 OID 和微端口状态指示集（见 5.6.5 节和 5.6.6 节）。如果微端口支持定制的 OID 或状态指示（见 5.6.7 节），NDIS 也向 WMI 注册这些 GUID。

对面向连接微端口，NDIS 也注册任何命名 VC（见 5.6.3 节）。只有用 **NdisCoAssign-InstanceName** 命名的呼叫管理器，集成微端口呼叫管理器（MCM）或面向连接客户对 WMI 客户来说才是可见的。

5.5.2 OID 和微端口状态的 GUID 映射

当微端口发送请求或设置微端口时（即，当 NDIS 发送一个 IRP 到 NDIS 创建的功能性设备对象时），NDIS 截取这个查询或设置，映射 GUID 为微端口支持的 OID，然后查询或设置微端口维护的 OID。如果微端口是无连接微端口，NDIS 调用微端口的 *Miniport-QueryInformation* 函数或 *MiniportSetInformation* 函数。如果微端口是面向连接微端口，NDIS 调用微端口的 *MiniportCoRequest* 函数。NDIS 返回 WMI 查询或设置微端口 OID 的结果。

如果微端口用其客户注册接收的 **Ndis(Co)IndicateStatus** 发出一个状态指示，并将它传给 WMI。WMI 然后传递这个状态指示给所有已注册指示的客户。

5.5.3 支持命名 VC

对面向连接微端口，NDIS 允许 WMI 客户在每 VC（per-VC）基础上查询或设置 OID。WMI 也能列举 VC，在 WMI 客户查询或列举相关的特殊 VC OID 信息之前，呼叫管理器，

MCM，或者面向连接的客户必须用 **NdisCoAssignInstanceName** 来命名 VC。

在用 **NdisCoCreateVC** 建立了 VC 之后，呼叫管理器或者面向连接的客户用 **NdisCoAssignInstanceName** 命名 VC，NDIS 分配 VC 一个实例名，并用 WMI 注册这个实例名。WMI 客户然后列举这个 VC，查询或设置有关的 OID。

MCM 不能用 **NdisCoAssignInstanceName** 来命名 VC，然而 MCM 应当为 VC 创建一个定制的 GUID 和 OID，并向 NDIS 注册这个 GUID-OID 映射（见 5.5.7 节）。

5.5.4 NDIS 支持的 WMI 操作

NDIS 支持下列 WMI 操作：

- 列举适配器和列举 VC

NDIS 向 WMI 注册全局的 GUID（GUID_NDIS_ENUMERATE_ADAPTER 和 GUID_NDIS_ENUMERATE_VC），允许 WMI 客户列举所有的适配器（微端口实例）和所有命名 VC。由于 NDIS 自己保持所有载入的微端口和所有命名 VC 的路径，NDIS 不需要查询微端口的这些信息。

- 查询单个实例（QUERY SINGLE INSTANCE）和设置单个实例（SET SINGLE INSTANCE）

  通过 NDIS，WMI 客户能查询或设置数据块（对应于单个 OID）的单个实例。对于查询，NDIS 返回所有与 OID 相关的信息。WMI 客户不能查询或设置 OID 中的数据项。例如，GUID_NDIS_GEN_CO_LINK_SPEED 查询返回输出和输入的速度。WMI 客户不能查询输出速度，但是能查询输入速度。

- 查询所有数据（QUERY ALL DATA）

  NDIS 通过得到适当的数据以及向 WMI 返回所有 GUID 实例的混和数据，来满足特殊 GUID 的一个 QUERY ALL DATA 请求。例如，要响应 GUID_NDIS_ENUMERATE_ADAPTER 的 QUERY ALL DATA，NDIS 给 WMI 返回一个所有加载微端口的列表。对映射到 OID_GEN_CO_XMIT_PDUS_OK 的 GUID 上的 QUERY ALL DATA，微端口查询每一个面向连接微端口 VC 的 OID，并返回混和数据给 WMI。由于 QUERY ALL DATA 的开销太大，建议 WMI 客户只对列举适配器和 VC 使用 QUERY ALL DATA。

- 事件指示（EVENT NOTIFICATION）

WMI 客户能向 NDIS 注册以通知特殊微端口的状态指示。当这样的事件发生时，NDIS 用适当的 GUID 传递这些状态信息，分配给 WMI 客户。

5.5.5 向 WMI 注册标准微端口 OID

NDIS 对无连接微端口自动使用下面所列示的一般 OID 向 WMI 注册 GUID：

OID_GEN_HARDWARE_STATUS
OID_GEN_MEDIA_SUPPORTED
OID_GEN_MEDIA_IN_USE
OID_GEN_MAXIMUM_LOOKAHEAD
OID_GEN_MAXIMUM_FRAME_SIZE
OID_GEN_LINK_SPEED
OID_GEN_TRANSMIT_BUFFER_SPACE
OID_GEN_RECEIVE_BUFFER_SPACE
OID_GEN_TRANSMIT_BUFFER_SIZE
OID_GEN_RECEIVE_BUFFER_SIZE

OID_GEN_VENDOR_ID
OID_GEN_VENDOR_DESCRIPTION
OID_GEN_CURRENT_PACKET_FILTER
OID_GEN_CURRENT_LOOKAHEAD
OID_GEN_DRIVER_VERSION
OID_GEN_MAXIMUM_TOTAL_SIZE
OID_GEN_MAC_OPTION
OID_GEN_MEDIA_CONNECT_STATUS
OID_GEN_MAXIMUM_SEND_PACKETS
OID_GEN_VENDORD_RIVER_VERSION
OID_GEN_XMIT_OK
OID_GEN_RCV_OK
OID_GEN_XMIT_ERROR
OID_GEN_RCV_ERROR
OID_GEN_RCV_NO_BUFFER
NDIS 对面向连接微端口自动使用下面所列示的一般 OID 向 WMI 注册 GUID：
OID_GEN_CO_HARDWARE_STATUS
OID_GEN_CO_MEDIA_SUPPORTED
OID_GEN_CO_MEDIA_IN_USE
OID_GEN_CO_LINK_SPEED
OID_GEN_CO_VENDOR_ID
OID_GEN_CO_VENDOR_DESCRIPTION
OID_GEN_CO_DRIVER_VERSION
OID_GEN_CO_MAC_OPTION
OID_GEN_CO_MEDIA_CONNECT_STATUS
OID_GEN_CO_VENDORD_RIVER_VERSION
OID_GEN_CO_MINIMUM_LINK_SPEED
OID_GEN_CO_XMIT_PDUS_OK
OID_GEN_CO_RCV_PDUS_OK
OID_GEN_CO_XMIT_PDUS_ERROR
OID_GEN_CO_RCV_PDUS_ERROR
OID_GEN_CO_RCV_PDUS_NO_BUFFER
NDIS 自动使用下面所列以太网 OID 向 WMI 注册 GUID：
OID_802_3_PERMANENT_ADDRESS
OID_802_3_CURRENT_ADDRESS
OID_802_3_MULTICAST_LIST
OID_802_3_MAXIMUM_LIST_SIZE
OID_802_3_MAC_OPTIONS
OID_802_3_RCV_ERROR_ALIGNMET
OID_802_3_XMIT_ONE_COLLISIONS
OID_802_3_XMIT_MORE_COLLISIONS
NDIS 自动使用下面所列令牌环网 OID 向 WMI 注册 GUID：
OID_802_5_PERMANENT_ADDRESS
OID_802_5_CURRENT_ADDRESS

OID_802_5_CURRENT_FUNCTIONAL
OID_802_5_CURRENT_GROUP
OID_802_5_LAST_OPEN_STATUS
OID_802_5_CURRENT_RING_STATUS
OID_802_5_CURRENT_RING_STATE
OID_802_5_LINE_ERRORS
OID_802_5_LOST_FRAMES

NDIS 自动使用下面所列 FDDI 网络 OID 向 WMI 注册 GUID：

OID_FDDI_LONG_PERMANENT_ADDRESS
OID_FDDI_LONG_CURRENT_ADDRESS
OID_FDDI_LONG_MULTICAT_LIST
OID_FDDI_LONG_MAX_LIST_SIZE
OID_FDDI_SHORT_PERMANENT_ADDRESS
OID_FDDI_SHORT_CURRENT_ADDRESS
OID_FDDI_SHORT_MULTICAT_LIST
OID_FDDI_SHORT_MAX_LIST_SIZE
OID_FDDI_ATTACHMENT_TYPE
OID_FDDI_UPSREAM_NODE_LONG
OID_FDDI_DOWNSTREAM_NODE_LONG
OID_FDDI_FRAME_ERRORS
OID_FDDI_FRAME_LOST
OID_FDDI_RING_MGT_STATE
OID_FDDI_LCT_FAILURES
OID_FDDI_LEM_REJECTS
OID_FDDI_LCONNECTION_STATE

NDIS 自动使用下面所列 ATM 网络 OID 向 WMI 注册 GUID：

OID_ATM_SUPPORTED_VC_RATES
OID_ATM_SUPPORTED_SERVICE_GATEGORY
OID_ATM_SUPPORTED_AAL_TYPES
OID_ATM_HW_CURRENT_ADDRESS
OID_ATM_MAX_ACTIVE_VCS
OID_ATM_MAX_ACTIVE_VCI_BITS
OID_ATM_MAX_ACTIVE_VPI_BITS
OID_ATM_MAX_AAL0_PACKET_SIZE
OID_ATM_MAX_AAL1_PACKET_SIZE
OID_ATM_MAX_AAL34_PACKET_SIZE
OID_ATM_MAX_AAL5_PACKET_SIZE
OID_ATM_RCV_CELLS_OK
OID_ATM_XMIT_CELLS_OK
OID_ATM_RCV_CELLS_DROPPED

5.5.6 向 WMI 注册的标准微端口状态

NDIS 自动用下列 **NdisM(Co)IndicateStatus** 指示的 NDIS_STATUS 代码向 WMI 注册 GUID：

NDIS_STATUS_RESET_START
NDIS_STATUS_RESET_END
NDIS_STATUS_MEDIA_CONNECT
NDIS_STATUS_MEDIA_DISCONNECT
NDIS_STATUS_MEDIA_SPECIFIC_INDICATION
NDIS_STATUS_LINK_SPEED_CHANGE

5.5.7 定制 OID 与状态指示

驱动程序开发者可以创建一个定制的 OID 或者（映射到开发者创建的定制 GUID 上的）微端口状态指示。NDIS 用在微端口上起作用的 WMI 注册这个定制的 GUID，以便 WMI 客户能查询或注册这个 OID，或者接收定制的状态指示。

NDIS 在微端口完成初始化之后通过查询微端口而得到微端口的定制信息或状态指示。要得到这样的信息，NDIS 用 OID_GEN_SUPPORTED_GUID 查询无连接微端口，用 OID_GEN_CO_SUPPORTED_GUIDS 来查询面向连接微端口。

对 OID_GEN_(CO_)SUPPORTED_GUIDS 的查询，给 NDIS 返回一个 NDIS_GUID 结构的数组。每一个 NDIS_GUID 结构映射一个定制 GUID 到定制的 OID 或定制的状态指示。就象 5.5.7.1 节所描述的，涉及到支持定制 OID 和状态指示工作的主要部分是填充 NDIS_GUID。就象 5.5.7.2 节所描述的，其他主要的任务是创建一个描述 GUID 的 MOF 文件，以及对微端口编译这个文件。

5.5.7.1 填充 NDIS_GUID

NDIS_GUID 结构定义如下：

```
typedef  struct  _NDIS_GUID
{
    GUID                Guid;
    union
    {
        NDIS_OID        Oid;
        NDIS_STATUS     Status;
    };
    ULONG               Size;
    ULONG               Flags;
}NDIS_GUID,*PNDIS_GUID;
```

要得到这个结构 **Guid** 成员的 GUID，驱动程序开发者必须运行 uuidgen.exe 程序，关于这个工具更多的信息，参见 platform SDK。

Oid 或 **Status** 成员是一个表示 OID 代码或状态码的 ULONG，对这两种情况，代码中最重要的字节是 0xFF，其余是由厂商定义的。这个代码在微端口范围之内必须是唯一的。

如果 NDIS_GUID 结构映射到一个返回数据的 OID，那么返回数组的 **Size** 成员确定了每一个数据项的大小。如果返回数据项的大小是变化的，或者如果微端口不返回数据，或者 NDIS_ GUID 结构映射到一个状态指示，那么 **Size** 成员必须是-1。

下列 **Flags** 标志可以“或”在一起，以指示 GUID 是否映射到一个 OID 或 NDIS_STATUS 串，并指示出 GUID 所支持数据的类型：

NDIS_GUID_TO_OID

当设置了时，表示 NDIS_GUID 结构映射一个 GUID 到 OID 上。

NDIS_GUID_TO_STATUS

当设置了时，表示 NDIS_GUID 结构映射一个 GUID 到 NDIS_STATUS 串上。

NDIS_GUID_ANSI_STRING

当设置了时，表示为 GUID 提供一个 0 终结的 ANSI 串。

NDIS_GUID_UNICODE_STRING

当设置了时，表示为 GUID 提供一个 UNICODE 串。

NDIS_GUID_ARRAY

当设置了时，表示为 GUID 提供一个数据项数组。所确定的 Size 表示数组中每一个数据项的长度。

5.5.7.2 包括 MOF 文件

所有映射到微端口的定制 OID 和/或定制状态指示的定制 GUID 的描述必须包含进一个管理对象格式（MOF）文件中，这个文件必须编译和包含进微端口的资源文件（*.rc）。关于 MOF 文件格式的描述，参见《*Driver Writer's Guid*》。

MOF 源文件必须是一种 MOFDATA 类型，有一个.mof 的扩展名。MOF 源文件必须用 mofcomp.exe 编译进二进制文件，并用 wmimofck.exe 检查，这两个工具都包含在 DDK 之中。下面一行必须插入到微端口的*.rc 文件中以包含 MOF 二进制：

MofResource　MOFDATA　*filename.bmp*

在这里 *filename* 是 MOF 源文件的文件名。

第六章 微端口的电源管理

本章描述由 NDIS 为 NDIS 微端口提供的电源管理（PM）服务，以及为支持电源管理而对微端口的要求。本章将会讨论以下主题：

6.1 电源管理的需求与可选的 OID
6.2 网络设备电源状态
6.3 网络唤醒事件
6.4 处理 OID_PNP_QUERY_POWER
6.5 处理 OID_PNP_SET_POWER
6.6 早期微端口的电源管理

对电源管理的一般性讨论，参见“*plug and play*”，“*Power Management*”以及“*Setup Design Guide*”。

6.1 电源管理的需求与可选的 OID

对微端口来说，支持电源管理就包括支持电源管理对象标识符（OIDs）。无连接的微端口用它的 *MiniportQueryInformation* 函数来处理 OIDs，面向连接的微端口则用它的 *MiniportCoRequest* 来处理。关于微端口执行查询和设置 OIDs 的详细描述，参见第五章。

微端口的电源管理支持有两个级别：

- 微端口支持其 NIC 在电源状态之间的转换，这是电源管理支持的最低级别。对网络设备电源状态的描述，参见 6.2 节。
- 微端口也支持许多网络唤醒事件。对网络唤醒事件的描述，参见 6.3 节。

为支持其 NIC 电源状态间的转换，微端口必须支持以下状态标识符（OIDs）：

- OID_PNP_CAPABILITIES

NDIS 通过查询微端口的 OID，看它是否返回 NDIS_STATUS_SUCCESS，从而判断其电源管理性质，如果微端口返回 NDIS_STATUS_NOT_SUPPORTED，NDIS 就认为这个微端口是一个早期的不支持电源管理的微端口。

- OID_PNP_QUERY_POWER

这个 OID 指明了被请求转换 NIC 的网络设备电源状态。通过返回 NDIS_STATUS_SUCCESS，微端口保证不会危及 NIC 对网络设备电源状态转换的请求。如果微端口不能作出这样的保证，它就必须返回一个 NDIS_STATUS_SUCCESS 以外的状态码。

- OID_PNP_SET_POWER

这个 OID 表明 NIC 将要转换到指定的网络设备电源状态，微端口必须在返回 NDIS_STATUS_SUCCESS 之前为 NIC 特定状态的转换作好准备，微端口必须总是对它的 OID 返回 NDIS_STATUS_SUCCESS。

为支持网络唤醒事件，微端口也必须支持 OID_PNP_ENABLE_WAKE_UP，它包含双方的协议，NDIS 用这个 OID 来赋予 NIC 的网络唤醒能力，详细描述参见 6.3.4 节。

为支持网络唤醒帧（见 6.3.2 节），微端口也必须支持下列与唤醒事件有关的 OID：

- OID_PNP_ADD_WAKE_UP_PATTERN
- OID_PNP_REMOVE_WAKE_UP_PATTERN

协议使用这个 OID 来删除一个或多个以前对微端口指定的唤醒模式。

支持网络唤醒的 NDIS 微端口可选择地支持下列与唤醒事件有关的统计 OID：

- OID_PNP_WAKE_UP_ERROR

这个 OID 查询微端口 NIC 的失败唤醒信号的数目。

- OID_PNP_WAKE_UP_OK

这个 OID 查询微端口 NIC 的有效唤醒信号的数目。

6.2 网络设备电源状态

网络设备电源状态描述了 NIC 的电源消耗和计算能力的级别。

网络设备电源状态以下列标准术语定义：

- 电源消耗：NIC 用了多少电能？多少电能用于 NIC 和特殊总线的操作（时钟等）？
- 设备环境：NIC 在转换到低的电源状态之前保存多少可用的环境？
- 微端口和 NIC 行为：微端口请求传送包或指示接收包吗？NIC 能产生中断吗？
- 恢复时间：将 NIC 恢复到完全操作状态要花费多少时间？NIC 能用它的状态来唤醒系统吗？

有四种网络设备电源状态：D0，D1，D2 和 D3，D0 是完全状态，D1，D2 和 D3 是睡眠状态。状态数与电源消耗相反：较高数的状态使用较少的电能，在 D3 状态，电源供应可以全部从 NIC 中去掉。

有电源管理功能的微端口支持所有四种网络设备电源状态。

网络设备电源状态定义为：

D0

电源消耗：设备全力工作以提供全部功能与性能。

设备环境：维护硬件设备环境。

微端口和 NIC 行为：NIC 全部适应于访问网络的请求，NIC 或微端口上没有电源管理约束。

恢复时间：不用。

D1

电源消耗：D1 是睡眠状态的最高级，电源消耗比 D0 少，大于或等于 D2 状态。

设备环境：硬件设备环境也许会丢失。

微端口和 NIC 行为：微端口并不响应协议的传输请求。因为 NDIS 可以通知绑定协议转换到睡眠状态，而且对于早期不支持电源管理的协议，微端口禁止这样的协议请求，所以实际上，微端口不接收这种协议请求。

在这种状态下，微端口不指示任何 NIC 接收的包。

NIC 不产生中断，然而微端口必须能产生中断，因为共享的中断会在总线上产生。

恢复时间：NIC 恢复到 D0 状态的时间要少于 NIC 在 D2 状态需要的时间。

D2

电源消耗：D2 是中间睡眠状态，电源消耗小于 D1 状态，大于等于 D3 状态。

设备环境：硬件设备环境也许会丢失。

微端口和 NIC 行为：微端口并不响应协议的传输请求。因为 NDIS 可以通知绑定协议转换到睡眠状态，而且对于早期不支持电源管理的协议，微端口禁止这样的协议请求，所以

实际上，微端口不接收这种协议请求。

在这种状态下，微端口不指示任何 NIC 接收的包。

NIC 不产生中断，然而微端口必须能产生中断，因为共享的中断会在总线上产生。

恢复时间：NIC 恢复到 D0 状态的时间要大于 NIC 在 D1 状态需要的时间而少于 NIC 在 D3 状态需要的时间。

D3

电源消耗：D3 是最低的睡眠状态，在这一状态下，电源供应可以完全从 NIC 中除去。

设备环境：硬件设备环境假定已丢失。

微端口和 NIC 行为：微端口并不响应协议的传输请求。因为 NDIS 可以通知绑定协议转换到睡眠状态，而且对于早期不支持电源管理的协议，微端口禁止这样的协议请求，所以实际上，微端口不接收这种协议请求。

在这种状态下，微端口不指示任何 NIC 接收的包。

NIC 不产生中断，然而微端口必须能产生中断，因为共享的中断会在总线上产生。

恢复时间：NIC 到 D0 的恢复时间要大于 NIC 在 D2 状态需要的时间。

在 NIC 转换到 D3 状态之前，微端口必须关闭在其控制之下的所有事情：必须禁止中断，取消时钟，等等。在 NIC 被总线驱动设置到 D3 状态之后，微端口将不能访问到 NIC 硬件。

允许设备电源状态之间的转换：设备电源状态之间唯一允许的转换是从完全状态（D0）到睡眠状态（D1，D2，D3）或者是从睡眠状态到完全状态，NDIS 永远不会直接命令 NIC 直接从一个睡眠状态转换到另一个睡眠状态。

6.3 网络唤醒事件

网络唤醒事件是一个引起 NIC 唤醒系统的外部事件。NIC 发出一个总线特性的唤醒信号，以唤醒系统，最终导致系统从睡眠状态转换到工作状态。

NDIS 定义了三种网络唤醒事件：

- NIC 网线的重新连接
- 接收到网络唤醒帧，其中包含由绑定协议确定的模式
- 接收到魔包（Magic Packet ™）

在微端口完成初始化之后，NDIS 和绑定协议用 OID_PNP_CAPABILITIES 查询微端口，确定微端口 NIC 的唤醒能力。NIC 也支持任何网络唤醒事件的组合，包括无事件发生。如果微端口对于 OID_PNP_CAPABILITIES 查询返回 NDIS_STATUS_SUCCESS，NDIS 判断这个微端口是支持电源管理的，而不管微端口报告的唤醒能力。

依赖于 NIC 的能力，网络唤醒事件能从任何设备电源状态中产生，包括完全状态 D0，由于 D0 是完全电源状态，信号唤醒事件不唤醒系统而只用作一个运行时事件。

6.3.1 连接改变唤醒

连接改变唤醒事件是在 NIC 检测到其网线重新连接而发出唤醒信号时产生的，连接唤醒在缺省状态下是启用的，所以协议不必启动这样的唤醒能力。NIC 在电缆没连接时不产生唤醒事件，而只是在重新连接时才会产生。

6.3.2 网络唤醒帧

如果一个微端口，在它对 OID_PNP_CAPABILITIES 查询的响应中，指示它的 NIC 在接收到包含特定模式的包后产生唤醒信号，绑定协议将为 NIC 启用网络唤醒帧，它还标明了

一个或多个唤醒模式。协议通过在 OID_ENABLE_WAKE_UP 中设置 NDIS_WAKE_UP_PATTERN_MATCH 标志来启用这种类型的唤醒。协议用 OID_PNP_ADD_WAKE_UP_PATTERN 标明了一个或多个唤醒模式，并且同时给出了一个表明哪个接收的字节应与这个模式相比较的掩码。依赖于 NIC 的能力，唤醒模式可以保存在 NIC 或主机内存中。协议能用 OID_PNP_ REMOVE_WAKE_UP_PATTERN 移去一个或多个唤醒模式。

关于更多的网络唤醒帧的信息，参见微软的“网络设备类电源管理参考规范”站点：

http://www.microsoft.com/hwdev/onnow.htm

6.3.3 魔包唤醒

魔包是一个包含 16 个邻近接收网卡的以太网地址的数据包。

6.3.4 启用唤醒事件

用 OID_PNP_CAPABILITIES 查询了微端口的唤醒能力之后，协议驱动程序将发送 OID_PNP_ENABLE_WAKE_UP 以启用一个或多个网卡的唤醒能力。NDIS 并不立即由协议启用 OID_PNP_ENABLE_WAKE_UP 标明的唤醒能力，而是跟踪由协议启用的唤醒能力，只是在微端口转换到睡眠状态之后，给微端口发送 OID_PNP_WAKE_UP_ENABLE 来启用适当的唤醒事件。

在缺省情况下，如果 NIC 支持这样的唤醒事件，启用的是连接变化（电缆重连）的唤醒。协议不能通过在 OID_PNP_ENABLE_WAKE_UP 设置或清除 NDIS_PNP_WAKE_UP_LINK_CHANGE 标志来启用或禁止连接改变唤醒，然而，NDIS 可以设置或清除这个标志来启用或禁止连接改变唤醒，正如下面所描述的那样。

微端口转换到一个低能耗状态之后（即 NDIS 送一个 OID_PNP_SET_POWER 给微端口之后），NDIS 送一个 OID_PNP_ENABLE_WAKE_UP 给微端口以启用适当的唤醒能力，有三种基本的方案：

- 如果系统从完全工作状态转换到睡眠状态，NDIS 禁止连接改变唤醒；如果协议启用魔包唤醒和/或模式匹配唤醒，NDIS 也将启用魔包唤醒和/或模式匹配唤醒。这种启用/禁止唤醒事件的配置防止了连接改变在系统处于睡眠状态时不适当地唤醒系统，但允许 NIC 作为响应魔包或模式匹配唤醒系统。
- 如果 NIC 从完全工作状态因连接改变（有人从 NIC 上断开了电缆）转换到睡眠状态，NDIS 启用连接改变唤醒，NDIS 同时也禁止魔包和模式匹配唤醒。在这种情况下，NIC 的网络电缆连接导致 NIC 产生一个唤醒信号。
- 如果因为有人断开电缆，系统转换到低能耗状态而使得 NIC 能耗降低，NDIS 将允许连接改变唤醒，禁止魔包和模式匹配唤醒。在这种情况下，NIC 电缆的重新连接把 NIC 和系统两个都唤醒了。

  NDIS 永远不会为 NIC 启用所有三种唤醒事件，NDIS 或者允许连接改变唤醒，禁止魔包和模式匹配唤醒，或者禁止连接改变唤醒而允许其他由协议允许的唤醒能力（魔包唤醒和/或模式匹配唤醒）。

6.3.5 处理唤醒事件

微端口并不处理由它的 NIC 检测到的唤醒事件。当 NIC 检测到一个唤醒事件之后，它声明一个特殊总线信号以指示总线驱动程序，总线驱动程序再指示 NDIS，由它（NDIS）来指示 NIC 系统的电源管理系统，电源管理系统发送一个电源 IRP 给 NDIS，NDIS 然后给微端口发送一个 OID_PNP_SET_POWER，使其转入工作状态。

6.4 处理 OID_PNP_QUERY_POWER

OID_PNP_QUERY_POWER 请求微端口指明：在接受到 OID_PNP_SET_POWER 之后，它是否能保证其 NIC 转换到指定的睡眠状态。通过返回 NDIS_STATUS_SUCCESS，微端口承诺它将不做会危及到 NIC 转换到指定设备状态的任何事。

在 OID_PNP_QUERY_POWER 之后总是跟随着 OID_PNP_SET_POWER。OID_PNP_SET_POWER 也许是立即跟随 OID_PNP_QUERY_POWER 请求或者在 OID_PNP_QUERY_ POWER 请求后的一段不确定的时间间隔内到达。OID_PNP_SET_POWER 请求有效的特定 D0 设备状态会取消前面的 OID_PNP_QUERY_POWER 请求。

6.5 处理 OID_PNP_SET_POWER

NDIS 发送一个 OID_PNP_SET_POWER，向微端口指示其 NIC 从工作状态转换到睡眠状态或着从睡眠状态转换到工作状态。

6.5.1 转入睡眠状态

在两种情况下 NDIS 会送一个 OID_PNP_SET_POWER 给微端口，使其转入指定的睡眠状态：

- 系统转入睡眠状态
- NIC 处于 D0 状态时，其网线断开 20 秒以上

在第一种情况下，OID_PNP_SET_POWER 优先于 OID_PNP_QUERY_POWER。在第二种情况下，OID_PNP_SET_POWER 并不优先于 OID_PNP_QUERY_POWER。实际上，第二种情况，只是 NDIS 将 NIC 置于不接收系统电源 IRP 的睡眠状态的情形。

送出 OID_PNP_SET_POWER 给微端口后，如果微端口支持任何唤醒事件，NDIS 将再给微端口送一个 OID_PNP_ENABLE_WAKE_UP 请求（见 6.3.4 节）。

微端口必须成功地执行 OID_PNP_SET_POWER。

当支持连接改变唤醒事件的 NIC 处于 D0 状态，而且其网线断开时，微端口会检测到状态的变化，用 OID_STATUS_MEDIA_DISCONNECT 调用 **Ndis(Co)IndicateStatus** 来报告给 NDIS。如果微端口在 20 秒内没有用 OID_STATUS_MEDIA_CONNECT 调用 **Ndis(Co)-IndicateStatus，**说明电缆已被重新连接，NDIS 将送一个 OID_PNP_SET_POWER 给微端口，NDIS 将把 NIC 从能产生连接改变唤醒事件状态设置为最低能耗状态。

在 OID_PNP_SET_POWER 请求返回 NDIS_STATUS_SUCCESS 之前，微端口必须执行必要的设备相关操作，来为 NIC 转入睡眠状态作准备，微端口也必须保存所有的以前用 OID_GEN_CURRENT_PACKET_FILTER 设置的包过滤器。如果其 NIC 转换到 D3 状态，微端口必须关闭其控制下的所有东西：中断必须禁止，时钟必须取消，等等。微端口在 NIC 被总线驱动程序设置到 D3 状态后不能访问 NIC 硬件。

6.5.2 转入工作状态

NDIS 将 NIC 从睡眠状态（D1，D2 和 D3）转换到工作状态（D0）有两种情况：

- 系统从睡眠状态转换到工作状态（D0）
- NIC 产生一个唤醒事件

NDIS 给微端口送一个 OID_PNP_SET_POWER 来启动向工作状态 D0 的转换，然后微端口执行必须的设备相关操作以将 NIC 恢复到工作状态，对这个 OID 请求返回 NDIS_STATUS_SUCCESS 时，微端口及其 NIC 必须已经准备好了常规操作。

如果在其 NIC 转入睡眠状态以前，已经为微端口设置了任何一种过滤器，那么此时微

端口也必须恢复这些过滤器。对支持电源管理的微端口，NDIS 不恢复这样的过滤器。

6.6 早期微端口的电源管理

如果满足以下条件，NDIS 将判断这个微端口是不支持电源管理的早期微端口：

- 在初始化期间，总线驱动程序指出微端口的 NIC 或微端口不支持电源管理
- 微端口对 OID_PNP_CAPABILITES 查询返回 NDIS_STATUS_UNSUPPORTED

早期微端口只支持完全工作（D0）和完全关闭（D3）网络设备电源状态。

初始化期间，早期的微端口可以指示 NDIS 在系统转换到睡眠状态（D3）之前不能停止它。微端口是通过在传送给 **NdisMSetAttributesEx** 的 *AttributeFlags* 参数中设置 NDIS_ATTRIBUTE_NO_HALT_ON_SUSPEND 标志而作出这样的指示的。早期的微端口将只设置这个参数，如果：

- 它保存了所有需要的硬件环境
- 它能为睡眠状态（D3）而将其 NIC 置于一个适当的状态
- 它能将 NIC 重新激活至工作状态（D0）

在微端口不设置 NDIS_ATTRIBUTE_NO_HALT_ON_SUSPEND 标志时，NDIS 能从总线驱动程序确定 NIC（网卡）不支持电源管理，NDIS 将不查询微端口的电源管理能力。然而，如果微端口设置了 NDIS_ATTRIBUTE_NO_HALT_ON_SUSPEND 标志，NDIS 将对微端口发出一个 OID_PNP_CAPABILITES 请求。在这种情况下，微端口将用 NDIS_STATUS_SUCCESS 表示完成 OID_PNP_CAPABILITES 请求，在微端口返回的对应于这种请求的 NDIS_PM_WAKE_UP_CAPABILITIES 结构中，微端口必须为每一种唤醒能力指定 **NdisDeviceStateUnspecified** 的设备电源状态。

NDIS 为早期微端口提供下列电源管理支持：

- NDIS 完成所有（系统电源管理器送至代表 NIC 的设备对象）的 IRP_MN_QUERY_POWER 请求。此时，NDIS 保证微端口的 NIC 转换到 D3 状态以响应来自于系统的查询电源请求。
- 如果微端口在初始化期间没有设置 NDIS_ATTRIBUTE_NO_HALT_ON_SUSPEND 标志，NDIS 将在微端口 NIC 转换到 D3 状态之前调用微端口的 MiniportHalt 函数，微端口 NIC 将失去所有的硬件环境。
- 如果微端口在初始化期间设置了 NDIS_ATTRIBUTE_NO_HALT_ON_SUSPEND 标志，NDIS 将在系统转换到 D3 状态之前不停止微端口，而是给微端口发出一个 OID_PNP_SET_POWER 到 D3 状态的请求。微端口必须保存所有它使用的硬件环境，并且将其 NIC 置于一个适宜于 D3 状态的状态上。
- 系统转入 D0 状态之前，如果要停止微端口，NDIS 调用微端口的 MiniportInitialize 函数。如果 NDIS 不停止微端口，它就给微端口会发出一个到 D0 状态的 OID_PNP_SET_POWER 请求，微端口必须将其 NIC 置于一个适宜于 D0 状态的状态上。
- 如果在转入睡眠状态之前，已经为微端口设置了包过滤器，NDIS 将通过给微端口发送一个 OID_GEN_CURRENT_PACKET_FILTER 来恢复这些过滤器。

第七章　重置，停止和关闭

本章描述重置，停止和关闭函数。

7.1　硬件重置（Reset）

微端口必须用 **NdisMRegisterMiniport** 注册一个 MiniportReset 函数。MiniportReset 的定义为：

```
NDIS_STATUS
    MiniportReset(
        OUT PBOOLEAN AddressingReset,
        IN NDIS_HANDLE MiniportAdapterContext
    );
```

如果 NDIS 确定微端口的网络接口卡（NIC）因下列原因而被挂起，MiniportReset 就会被调用：

- 微端口的 MiniportcheckForHang 函数返回 TRUE
- NDIS 检测到一个未决的发送包（只对串行微端口）
- NDIS 检测到一个在指定的超时期间内未完成的未决请求

MiniportReset 也可以用调用 **NdisMResetComplete** 来同步或异步地完成（见图 7.1）。

图 7.1　网络接口卡的重置

MiniportReset 将：

- 禁止将要到来的中断。
- 清除进程中与任何发送有关的包，例如，对于总线控制器的直接内存访问（DMA）设备，发送缓冲区的指针必须被清除。串行微端口不顾其发送的状态，而放弃所拥有的未决包，对这样一个微端口，NDIS 以一个适当的状态完成一次未决的（给上层的）发送。非串行和面向连接的微端口必须为任何发送包的队列返回 NDIS_STATUS_REQUEST_ABORTED。
- 恢复硬件状态和微端口内部状态至与广播地址和注册过滤器有关的初始状态。微端口可以这样做，或者依赖于 NDIS。

如果微端口在 *AddressReset* 参数中返回 FALSE，则它将其地址值返回到其初始状态。如果微端口在 *AddressReset* 中返回 TRUE，NDIS 将调用无连接微端口的 MiniportSetInformation 函数或者面向连接的微端口的 MiniportCoRequest 函数，以对其初始状态设置地址和过滤器。注意，驱动程序的编写者应当确定微端口是否应保存这个地址状态，以至于在重置时能恢复到此状态，或者在重置时微端口是否依赖于 NDIS 以恢复其地址。

7.2　停止（Halt）处理程序

微端口必须对 **NdisMRegisterMiniport** 应用 MiniportHalt 函数。MiniportHalt 的定义为：

```
VOID
    MiniportHalt(
```

IN NDIS_HANDLE *MiniportAdapterContext*

);

在下列情况下，MiniportHalt 会被调用：

- 建立广播地址或设置 MAC_OPTIONS 的请求失败之后
- 所有绑定（bind）到微端口的协议解除之后
- 卸载操作期间，比如一个 NetStop 命令导致一个卸载操作执行
- 系统关闭（shutdown）期间

MiniportHalt 将撤销 MiniportInitialize 所作的任何事情，包括：

- 释放口
- 释放它所声明的所有硬件资源
- 调用 **NdisMDeregisterInterrupt** 释放中断资源
- 释放它所分配的任何内存
- 停止 NIC，除非 MiniportShutdown 函数已将 NIC 恢复到其初始状态。

MiniportHalt 应当在返回以前完成这些必须的卸载驱动程序的操作。图 7.2 显示了 MiniportHalt 所做的调用的集合，这些调用只是所能进行的调用的一部分。实际的调用集合依赖于先前微端口的行为。微端口如果因硬件问题，或不能得到所需的资源而不能成功地初始化网络适配卡时，也会在 MiniportInitialize 中作相同的调用。在这种情况下，MiniportInitialize 将会通过取消以前的操作来卸载驱动程序，否则，MiniportHalt 将取消 MiniportInitialize 的动作。

图 7.2　卸载微端口驱动程序

下面列出了请求撤消微端口所做动作的调用：

- 如果微端口注册了一个中断，它应调用 **NdisMDeregisterInterrupt**
- 如果微端口 NIC 驱动程序注册了关闭处理器（shutdown handler），它应调用 **NdisMDeregisterAdapterShutdownHandler**。
- 如果微端口建立了一个或多个时钟，它应为它所创建的每一个时钟调用 **NdisMCancelTimer**。
- 如果微端口用 **NdisAllocateMemory**、**NdisMAllocateShareMemory** 或 **NdisM-AllocateShareMemoryAsync** 分配了任何内存，它将调用 NdisFreeMemory 来释放这些内存。
- 如果微端口用 **NdisAllocatePacketPool** 分配并初始化包描述符池的存储，它将调用 NdisFreePacketPool 来释放这个存储。
- 如果微端口分配或保存了任何硬件资源，它们将调用 **NdisMDeregisterIoPortRange**；如果 NIC 是一个总线控制的 DMA NIC，则调用 **NdisMFree MapRegister**。

7.3　关闭（Shutdown）处理程序

微端口驱动程序应当在初始化期间注册关闭处理程序。**MiniportShutdown** 处理程序的定义为：

VOID

MiniportShutdown(

IN PVOID　*ShutdownContext*

);

驱动程序的关闭处理程序应当只：

- 在从关闭处理程序返回之后，确保硬件不再试图从内存中拷贝任何数据。也就是说，如果进程中存在直接内存访问（DMA）操作，这个操作应当被终止。
- 重置 NIC 硬件到其初始状态。

关闭处理程序的调用可以作为用户操作的结果，在这种情况下，它运行于 IRQL PASSIVE_LEVEL 下。它也可以因为不可重获的系统错误而被调用，如果这样的话，**MiniportShutdown** 能运行于任何 IRQL 之下。**MiniportShutdown** 将不调用 **Ndis***Xxx* 函数。

不象 *MiniportHalt*，**MiniportShutdown** 将不释放任何分配的资源。**MiniportShutdown** 只停止 NIC。

第八章 广域网微端口 NIC 驱动程序

本章将介绍在标准和面向连接的 NDIS（CoNDIS）环境中支持广域网 NIC 微端口驱动程序所需的 NDIS 结构和函数。同时将详细描述如何实现标准和 CoNDIS 广域网微端口驱动程序。

Microsoft® Window® 2000 同时支持标准和 CoNDIS 广域网微端口驱动程序的实现，但是我们鼓励驱动程序开发者编写 CoNDIS 广域网微端口驱动程序。

标准广域网微端口驱动程序：

- 大部分在非广域网无连接的微端口驱动程序中的 NDIS 函数仍被使用。
- 部分在非广域网无连接的微端口驱动程序中的 NDIS 函数被替换。
- 提供附加的广域网功能。

CoNDIS 广域网微端口驱动程序：

- 提供面向连接的微端口驱动程序提供的所有函数。
- 使用面向连接的微端口驱动程序所使用的 NDIS 函数。
- 提供附加的广域网功能。

标准和 CoNDIS 环境下的广域网 NIC 微端口驱动程序开发时需遵循第二，三，四章描述的要求和功能，除此之外还遵循本章的要求。

远程访问服务（Remote Access Servise , RAS）使远程工作站能够拨入局域网并且透明的访问局域网上的资源。

以下节描述在标准和面向连接的环境下，广域网卡的驱动程序如何通信。同时描述了如何实现两种环境下的广域网 NIC 微端口驱动程序及包的组帧：

- 8.1 RAS 体系结构
- 8.2 NDISWAN 概述
- 8.3 网卡、绑定、和连接
- 8.4 广域网微端口驱动程序的实现
- 8.5 广域网包组帧
- 8.6 标准 NDIS 上的电话服务扩展
- 8.7 使用支持电话服务的 CoNDIS 扩展

8.1 RAS 体系结构

本节说明远程访问服务（RAS）各部件之间的联系，以及广域网卡（如：ISDN, X.25, Switched 56 适配器等）的驱动程序如何使用 NDIS 和 NDISWAN 服务在标准广域网环境和面向连接的广域网环境中进行通信。

图 8.1 说明了 RAS 体系结构。后面的章节中将描述 NDISWAN 中间层 NDIS 驱动程序、NDISTAPI 驱动程序、及 NDPROXY 驱动程序，并且给出了整个 RAS 系统更详细的描述。

图 8.1 中所示的 WAN、RAS、和 TAPI 组件将在下文中讨论。

图 8.1　RAS 体系结构

RAS 函数

RAS 函数集允许用户模式的应用程序建立 RAS 连接。当 RAS 连接建立后，应用程序就可以使用标准网络接口（如：Windows Socket、NetBIOS、Named Pipes、和 RPC 等）连接到网络服务。

传输

RAS 系统部件提供传输服务如 PPP 验证（PAP，CHAP），网络配置协议（IPCP，IPXCP，NBFCP，LCP 等）。广域网微端口驱动程序仅实现 PPP 介质相关组帧。.

TAPI 服务

TAPI 服务（*tapisrv.exe*）将不同服务提供者的电话服务提供者接口（TSPI）提供给使用 TAPI 的应用程序。应用程序使用特定的服务提供者与特定类型的设备通信。这些服务提供者是在 TAPI 服务进程中运行的 DLLs。无论是标准的还是 CoNDIS 广域网微端口驱动程序，都可以使用操作系统提供的服务提供者与用户模式驱动程序通信。

KMDDSP

本部件是一个在 TAPI 服务进程中运行的服务提供者 DLL。*kmddsp.tsp* 部件为 TAPI 应用程序提供了一个 TSPI 接口，以便 NDISTAPI 能与用户模式应用程序通信。本部件为 NDISTAPI 将用户模式的请求转换为相关的 TAPI OIDs.

NDISTAPI

本部件实现 TAPI 接口的内核模式部分。*ndistapi.sys* 部件通过 **NdisRquest** 函数将 TAPI 专用的 OID 请求传递给适当的标准广域网微端口驱动程序，与广域网微端口进行通信。

NDPTSP

本部件是一个在 TAPI 服务进程中运行的服务提供者 DLL。*ndptsp.tsp* 为 TAPI 应用程序提供了一个 TSPI 接口，使 NDPROXY 能与用户模式应用程序通信，本部件为 NDPROXY 将用户模式的请求转换为相关的 TAPI-CO-related OIDs。

NDPROXY

在产生和接受呼叫时本部件将 TAPI 参数封装到 NDIS 结构中。*ndproxy.sys* 部件通过 NDPTSR 的 TSPI 接口与 TAPI 通信；通过 NDIS 与 NDISWAN 和 CoNDIS 广域网 NIC 微端口驱动程序进行通信。本部件为微端口驱动程序提供了一个客户接口，为 NDISWAN 提供呼叫管理器接口。NDISWAN 为本部件提供客户接口。微端口驱动程序为本部件提供呼叫管理器接口。本部件使用 TAPI-CO-related OIDs 调用 **NdisCoRequest** 函数列举 CoNDIS 广域网 微端口驱动程序的 TAPI 性能。同时，本部件注册特定的 TAPI 地址族、创建虚连接（VC）、产生和接受呼叫、并激活虚连接以便在虚连接上发送和接收数据。

NDISWAN

NDISWAN 中间层 NDIS 驱动程序支持 PPP 协议/链路组帧、压缩、和加密。NDISWAN

支持标准和面向连接的微端口驱动程序。*ndiswan.sys* 通过两个接口与标准广域网 NIC 驱动程序通信：

1. NDIS 广域网接口。
2. NDIS 微端口 NIC 驱动程序接口。

ndiswan.sys 通过面向连接的微端口 NIC 驱动程序接口与 CoNDIS 广域网微端口驱动程序通信。

串口驱动程序

本部件是内部串行端口或多端口串行卡的标准设备驱动程序。Windows® 2000 内置的异步广域网微端口驱动程序使用内部串口驱动程序进行调制解调器通信。输出与串口驱动程序相同的函数的任何驱动程序都与内置的异步 WAN 微端口驱动程序共同工作。

X.25 厂商可以选择为 X..25 卡实现串口驱动程序仿真程序。在这种情况下，每个 X.25 卡上的虚连接都表现为一个串口（有一个 X.25 PAD 连在上面）。连接接口必须能正确仿真串行信号如：DTR、DCD、RTS 和 DSR。

选择为 X..25 卡实现串口驱动程序仿真程序的厂商必须在 *pad.inf* 为 PAD 中提供入口。这个文件里包含通过 X.25 PAD 建立连接所需的命令/响应脚本。*pad.inf* 的详细信息请参见 *Remote Access Service Administrator's Guide* 。

广域网微端口 NIC 驱动程序

视情况而定，ISDN、Switched 56 和 X.25 厂商可以选择编写标准的或 CoNDIS 广域网微端口驱动程序。

8.2 NDISWAN 概述

NDISWAN 是一个中间层 NDIS 驱动程序，它可以运行在标准的和面向连接的环境中。NDISWAN 为上层协议驱动程序同时提供标准和 CoNDIS 802.3 微端口驱动程序接口。NDISWAN 同时为下层管理广域网卡的广域网微端口驱动程序提供协议驱动程序接口。对于 CoNDIS，这个广域网微端口驱动程序可以是一个集成的微端口呼叫管理器（MCM）。

NDISWAN 将外出包从局域网格式转换为 PPP 格式。NDISWAN 使用简单 HDLC 组帧。大多数介质相关的组帧必须在微端口驱动程序中完成。NDISWAN 从微端口驱动程序对 OID_WAN_MEDIUM_SUBTYPE 查询请求的响应来获得组帧的信息。

广域网微端口驱动程序可以在广域网上收发包之前必须要建立连接。建立连接的方法依赖于如下所描述的广域网微端口驱动程序是在标准还是 CoNDIS 环境下工作：

- 在标准环境中，应用程序必须建立一个由发送节点发起的连接，或通过产生或接受呼叫接受远程节点发起的连接。连接的建立、管理、拆卸是通过 TAPI 完成的。TAPI 请求和对 TAPI 的状态指示都通过 NDISWAN。在标准广域网微端口驱动程序上使用 TAPI 服务的详细叙述见 8.6 节。
- 在 CoNDIS 环境中，须建立虚连接（VC）。NDPROXY 为应用程序发起的外出呼叫创建虚连接。类似的，MCM 为内入呼叫指示 NDISWAN 和 NDPROXY 建立虚连接。MCM 必须通知远程团体虚连接参数，有时需与远程团体协商虚连接参数。连接的建立、管理、

拆卸是通过 TAPI 完成的。TAPI 请求和对 TAPI 的状态指示都通过 NDISWAN。NDPROXY 实现的 TAPI 服务详见 8.7 节。

在应用程序与远程节点建立连接后，就可以在广域网上交换数据包了。包的交换方式依赖于广域网微端口驱动程序是在标准还是在 CoNDIS 环境下运行：

- 在标准环境下，典型的从协议驱动程序到标准广域网微端口驱动程序的包传送路径是：协议驱动程序以包描述信息为参数调用 **NdisSend**；NDISWAN 检查包描述信息头以决定包将被送往哪个链接；NDISWAN 将一般不连续的数据包拷贝到连续的缓冲区中，此缓冲区带有标准广域网微端口驱动程序在初始化和注册时指定的报头和报尾填充。
- 在 CoNDIS 环境下，在协议驱动程序以一个 VC 句柄和一个包描述信息指针数组为参数调用 **NdisCoSendPackets** 函数之后，包通过 VC 从协议驱动程序传送到 CoNDIS 广域网微端口驱动程序，。NDISWAN 为指定 VC 接收包数组。

NDISWAN 接收到的包有几个可能的配置选项：报头压缩、数据压缩、和加密。这些配置选项是以一定顺序加到包上的。在接收方以相反的顺序解压和解密。任何时候启用这些如压缩、加密等的配置选项时，须通知特定的广域网微端口驱动程序。

在将包发送给广域网微端口驱动程序的发送函数以前，NDISWAN 进行简单 PPP HDLC 组帧。简单 PPP HDLC 组帧是指没有帧检验序列（FCS）、位/字填充、起始/结束标记的 PPP HDLC 组帧。有关介质相关组帧的内容详见 8.5 节广域网包组帧。最后，NDISWAN 发送包。然后 NDIS 调用广域网微端口驱动程序的发送函数。NDIS 根据广域网微端口驱动程序的类型（标准或 CoNDIS）调用不同的函数：

- 如果是标准广域网微端口驱动程序，发送函数是 *MiniportWanSend* ，而不是其他局域网微端口 NIC 驱动程序所实现的类似的函数。
- 如果是 CoNDIS 广域网微端口驱动程序，发送函数是 *MiniportCoSendPackets* ，所有面向连接的微端口驱动程序都实现这个发送函数。

广域网微端口驱动程序收到包后，它应在活动连接上发送包。对于标准环境，这个连接是指已经建立的链路。对于 CoNDIS 环境，这个连接是虚连接（VC）。广域网微端口驱动程序在每个连接上可以处理的未完成包的个数是由微端口驱动程序指定给协议驱动程序的。广域网微端口驱动程序指定这个数值的方法取决于它是标准或 CoNDIS 类型的：

- 标准广域网微端口驱动程序在 NDIS_WAN_INFO 结构的 **MaxTransmit** 成员中指定每个数据信道可以拥有的未完成包的缺省个数。微端口驱动程序用此结构响应协议驱动程序的 OID_WAN_GET_INFO 请求。然而，微端口驱动程序可以动态管理发送窗口，通过以 line-up 指示为每个线路调用 **NdisMIndicateStatus** 调整发送窗口。在这个调用中，微端口驱动程序为传递给函数的 NDIS_MAC_LINE_UP 结构的成员 **SendWindows** 提供一个新的非零值。如果微端口驱动程序将 **SendWindows** 置为零，NDISWAN 将为此线路使用 **MaxTransmit** 中保留的缺省值。

  标准广域网微端口驱动程序可能要修改或添加包的报头和报尾（如添加 FCS）。微端口驱动程序可以预先对包头和包尾进行适当填充。为了在广域网介质上发送数据，广域网微端口驱动程序可以用任何适当的方式改变包中的数据。
- CoNDIS 广域网微端口驱动程序在 NDIS_WAN_CO_INFO 结构的成员 **MaxSendWindow** 中指定每个 VC 可以拥有的未完成包个数。微端口驱动程序提供此结构来响应协议驱动程序的 OID_WAN_CO_GET_INFO 请求。然而，微端口驱动程序可以动态调整此数字，

以及为每个 VC 使用由微端口传输到 **NdisMCoIndiateStatues** 函数 WAN_CO_LINKPARAMS 结构的 **SendWindow** 成员动态调整这个数字。NDISWAN 使用当前的 **SendWindow** 值作为对未完成发送数的限制。微端口驱动程序通过将 **SendWindow** 置为 0 来表示它不能处理任何未完成的包，也就是说，如果 **SendWindow** 成员设置为“0”，发送窗口将关闭。

注意，如果 CoNDIS 广域网微端口驱动程序将 **SendWindow** 置零，它的关闭发送窗口将被关闭。如果标准广域网微端口驱动程序将 **SendWindow** 置零，它的发送窗口被设置为缺省值。

8.3 网络卡、绑定、和连接

广域网微端口驱动程序在每个绑定上可管理多个连接，因此每个 NIC 可有多个连接。对于标准环境，这些连接是已经建立的链路；对于 CoNDIS 环境，这些连接是虚连接（VC）。

与其他微端口驱动程序一样，每个广域网微端口驱动程序至少应有一个 NIC，为每个它分配并保留一个 NIC 专用的环境区域。此环境区域仅仅用于存储、检索和使用 NIC 硬件细节，如中断、总线类型、I/O 范围、和存储器，以及维护运行时状态。微端口 NIC 驱动程序应为系统中它所支持的每个网卡指定一个 NIC 专用的环境区域。

从广域网微端口驱动程序的观点看，每个 NIC 只有一个绑定，尽管在微端口驱动程序之上有许多协议被绑定到某个 NIC 上。因为这些协议对广域网微端口驱动程序是未知的，微端口驱动程序不保留对 NIC 的绑定。NDISWAN 处理微端口驱动程序的绑定以区分不同的协议。实际上，NDISWAN 绑定到一个或多个广域网微端口驱动程序上，一个或多个协议驱动程序绑定到 NDISWAN 上。

标准和 CoNDIS 广域网微端口驱动程序在 NIC 上使用不同类型的的连接：

- 标准广域网微端口驱动程序使用的连接是链路。链路是逻辑的、点对点的、双向的通信信道。每个 NIC 可以有多个链路。链路可以动态建立和拆除。每个连接的链路的速度和质量可以不同。然而，一个 NIC 上所有链路的填充和链接能力是通用的：如果一个标准广域网微端口驱动程序指定了 20 byte 的头填充和 4 byte 的尾填充，那么这个填充对微端口驱动程序的 NIC 上所有的链路都保持不变。
- CoNDIS 广域网微端口驱动程序与所有面向连接的微端口驱动程序相同，使用的是虚连接（VC）。每个 NIC 上可以有多个虚连接。每个虚连接的速度可以不同。但是同一 NIC 上所有虚连接的性能是相同的。如果一个 CoNDIS 广域网微端口驱动程序为它可以接收和发送的任何一个包指定了最大帧长度，则这个最大帧长度对 NIC 上的所有的 VC 都保持不变。

如果某个广域网微端口驱动程序指出它不需要 PPP 地址和控制字段压缩，则这对 NIC 上所有的连接都是有效的。

图 8.2 广域网微端口驱动程序的绑定

每个协议只绑定一次 NDISWAN 驱动程序
尽管 X.25 微端口只有一个绑定，NDISWAN 可以将以包传递给三个协议中的任何一个
ISDN 微端口有两个适配器，因此有两个绑定

协议只绑定到 NDISWAN 上一次，且不绑定到广域网微端口驱动程序上。这种绑定节省了内存并简化了广域网微端口驱动程序。因为在一个典型的系统中可能有多个协议和多个广域网微端口驱动程序，减少绑定数目可以节省内存。也就是说不必每个协议和驱动程序之间都有绑定。在上图中，如果每个协议都绑定到驱动程序上，会有 9 个绑定；如果每个协议只有一个广域网绑定，则这些协议可能被简化。

一个广域网微端口驱动程序一定是一个 NDIS 微端口 NIC 驱动程序。完全的 NIC 驱动程序不支持广域网 NIC。本章和第二部分有关微端口 NIC 驱动程序的章节给出了编写广域网微端口驱动程序所需的设计信息。

在源程序中加入以下行将驱动程序标识为微端口驱动程序：

C_DEFINES=/DNDIS_MINIPORT_DRIVER

支持通过 TAPI 连接的标准广域网微端口驱动程序，必须在源程序中加入以下行，指定所支持的 TAPI 版本号:

C_DEFINES=-DNDIS_TAPI_CURRENT_VERSION=0X00010003

是集成微端口呼叫管理器（MCM）驱动程序和支持 CoNDIS 地址族类型 CO_ADDRESS_FAMILY_TAPI_PROXY 的 CoNDIS 广域网微端口驱动程序，必须在源程序中加入下列行以指定所支持的 TAPI 版本号：

C_DEFINES=-DNDIS_TAPI_CURRENT_VERSION=0x00030000

广域网微端口驱动程序的包含路径须包含 *ndiswan.h* 和 *ndis.h*。

如果广域网微端口驱动程序支持通过 TAPI 的连接，则还要包含 *ndistapi.h*。

8.4 广域网微端口驱动程序的实现

为了支持像 ISDN、帧中继、Switch 56 等带远程访问服务（RAS）的介质，必须要提供一个广域网微端口驱动程序。微端口驱动程序应依据 NDIS 的当前版本（现为 5.0 版）编写。对于标准广域网微端口驱动程序，现存的根据 NDIS 3.0 和 NDIS 4.0 编写的驱动程序仍然可以使用。

标准广域网微端口驱动程序调用少数 NDIS 函数的广域网变种。标准广域网微端口驱动程序调用的部分 NDIS 函数与局域网微端口驱动程序调用的相同，部分局域网微端口驱动程序的 NDIS 调用被广域网专用的 NDIS 调用替换。另外，标准广域网微端口驱动程序提供了广域网专用的发送函数，它的定义与局域网微端口驱动程序实现的 MiniportSend 函数并不相同。本节后面将介绍这些广域网专用的调用和函数。

CoNDIS 广域网微端口驱动程序与所有面向连接的微端口驱动程序调用相同的 NDIS 函数集，它们提供的函数集也相同。

广域网微端口驱动程序通过 NDIS 库与 NDISWAN 中间层 NDIS 驱动程序通信。由于 NDISWAN 可以提供公用服务，如：数据压缩、加密、回送、简单 PPP 组帧等，因此广域网

微端口驱动程序只需要与介质相关的特征（例如：ISDN 需要 Q931 信令）。

广域网微端口驱动程序是这样的 NIC 微端口驱动程序：

- 执行从 NDISWAN 中间层 NDIS 驱动程序到广域网微端口驱动程序的连接管理操作。
- 接受发送请求，向 NDISWAN 驱动程序指示接受到的数据。

下面章节定义了 NDISWAN 驱动程序与广域网微端口驱动程序之间的接口，并详述了实现微端口驱动程序的细节。除了本章内容，广域网微端口驱动程序的开发者还需要参考第二部分有关局域网微端口驱动程序的章节，并参考 online DDK 内的 Network Drivers Reference 获得大多数关于 NDIS 函数的细节。

8.4.1 标准广域网微端口驱动程序与局域网微端口驱动程序的区别

在前面几章我们已经描述了标准广域网微端口驱动程序与局域网微端口驱动程序的一些区别，这些区别将影响标准广域网微端口驱动程序的实现。

- 标准广域网微端口驱动程序被禁止用 **NdisMRegisterMiniport** 注册 *MiniportTransferData* 处理程序。它总是将整个包传递给 **NdisMWanIndicateReceive** 函数。**NdisMWanIndicateReceive** 函数返回后，包已经被复制，此时广域网微端口驱动程序可以重用分配的包资源。
- 标准广域网微端口驱动程序提供函数 *MiniportWanSend* 代替 *MiniportSend* 函数。*MiniportWanSend* 函数接受 NDIS_WAN_PACKET 类型（不是 NDIS_PACKET 类型）的描述信息和一个指定包将要在上发送的数据信道的附加参数。
- 标准广域网微端口驱动程序从不返回 NDIS_STATUS_RESOURCES 作为 MiniportWanSend 或任何其他 Miniport*Xxx* 函数的执行情况，且不能调用 **NdisMSendResourcesAvaliable** 函数。
- 标准广域网微端口驱动程序支持一组广域网专用的 OID，这些 OID 可以用来设置和查询操作特性。
- 标准广域网微端口驱动程序支持一组传递给 **NdisMIndicateStatus** 的广域网专用的状态指示。这些状态指示报告链接状态的变化。
- 有两个广域网专用的 NDIS 函数可以被标准广域网微端口驱动程序调用，以完成用于发送和接收的广域网专用 NDIS 调用。这两个完成函数是：

  NdisMWanIndicateReceiveComplete

  NdisMWanSendComplete
- 标准广域网微端口驱动程序用 NDIS_WAN_PACKET 而不是 NDIS_PACKET 类型的描述信息。
- 标准广域网微端口驱动程序保留一组广域网专用的统计数据。
- 标准广域网微端口驱动程序从不回送，回送总是由 NDISWAN 提供。

8.4.2 CoNDIS 广域网微端口驱动程序的附加特性

CoNDIS 广域网微端口驱动程序必须实现以下附加特性：

- CoNDIS 广域网微端口驱动程序必须支持一组用来设置和查询操作特性的 CoNDIS 广域

网专用 OID。

- CoNDIS 广域网微端口驱动程序支持一组传递给 **NdisMCoIndicateStatus** 的 CoNDIS 广域网专用的状态指示。这些状态指示报告虚连接（VC）状态的变化
- CoNDIS 广域网微端口驱动程序应为驱动程序管理的 NIC 上的每个虚连接保留一组统计数据。
- CoNDIS 广域网微端口驱动程序使用 NDIS_PACKET 类型的描述符，而不是标准广域网微端口驱动程序使用的 NDIS_WAN_PACKET 类型的描述符。
- CoNDIS 广域网微端口驱动程序从不试图回送任何包；NDISWAN 提供回送支持。

8.4.3 广域网微端口驱动程序提供的服务

标准和 CoNDIS 广域网微端口驱动程序对 NDISWAN 驱动程序提供不同的接口。标准广域网微端口驱动程序提供的接口，除发送请求外，与局域网微端口驱动程序的相同。CoNDIS 广域网微端口驱动程序提供的接口与面向连接的微端口驱动程序相同。广域网微端口驱动程序接收并响应一组广域网专用 OID。对于标准广域网微端口驱动程序，用来查询和设置的 OID 被分别传递给 *MiniportQueryInformation* 和 *MiniportSetInformation* 函数。对于 CoNDIS 广域网微端口驱动程序，用来查询和设置的 OID 被传递给函数 *MiniportCoRequest* 。广域网微端口驱动程序在注册时将自己标识为广域网微端口驱动程序。

以下章节描述了广域网微端口驱动程序如何标识它自己，如何处理发送操作，以及用于查询和设置标准广域网微端口驱动程序、CoNDIS 广域网微端口驱动程序、和广域网 NIC 特性的 OID。

- 8.4.3.1 注册为广域网微端口驱动程序
- 8.4.3.2 查询广域网小段口驱动程序的信息
- 8.4.3.3 设置广域网小段口驱动程序的状态
- 8.4.3.4 在广域网微端口驱动程序上发送数据

8.4.3.1 注册为广域网微端口驱动程序

广域网微端口驱动程序在注册时可以将自己标识为标准或面向连接的微端口驱动程序。微端口驱动程序在它的函数 **DriverEntry** 内调用函数 **NdisMRigisterMiniport** 向 NDIS 注册。在此调用中，标准广域网微端口驱动程序通过在 NDIS*XX*_MINIPORT_CHARACTERISTICS 结构的成员 **Reserved** 中设置标志 NDIS_USE_WAN_WRAPPER 来指出它需要广域网服务。NDIS*XX*_MINIPORT_CHARACTERISTICS 结构的成员还指定了适当的 NDIS 版本号。NDIS 3.0、4.0 和 5.0 支持标准广域网微端口驱动程序。只有 NDIS 5.0 支持 CoNDIS 广域网微端口驱动程序。在此调用中，微端口驱动程序还为驱动程序提供的 MiniportXxx 函数指定了入口点。

为了从 NDIS 接收适当的 NDIS*XX*_MINIPORT_CHARACTERISTIC 结构，广域网微端口驱动程序还必须在源代码开始部分嵌入适当的编译器指令，或在源程序文件中设置适当的编译指令。这些指令取决于广域网微端口驱动程序运行的环境，是标准的还是 CoNDIS 环境：

- 对于标准广域网微端口驱动程序，将以下编译器指令被放在源代码开始部分表示微端口驱动程序使用了 NDIS_MINIPORT_CHARACTERISTICS 结构的 4.0 版：

  ```
  #define NDIS_MINIPORT_DRIVER
  #define NDIS40_MINIPORT 1
  #include <ndis.h>
  ```

  如果 NDIS40_MINIPORT 指令被忽略了，标准广域网微端口驱动程序将使用 3.0 版的

NDIS_MINIPORT_CHARACTERISTICS 结构。

- 对于 CoNDIS 广域网微端口驱动程序，将以下编译器指令被放在源代码开始部分表示微端口驱动程序使用了 NDIS_MINIPORT_CHARACTERISTICS 结构的 5.0 版：

  ```
  #define NDIS_MINIPORT_DRIVER
  #define NDIS50_MINIPORT 1
  #include <ndis.h>
  ```

  注意，在 NDIS*XX*_MINIPORT_CHARCTERISTICS 的成员中指定的 NDIS 的版本号必须与用编译器指令为驱动程序设置的版本号一致。

在广域网微端口驱动程序的 **DriverEntry** 函数执行完之后，微端口驱动程序已经为一个或多个 NIC 注册了它的微端口入口点。此后，微端口驱动程序必须为上层驱动程序（如：NDISWAN、NDPROXY）提供它及其 NIC 性能的信息。驱动程序提供性能信息的途径取决于广域网微端口驱动程序工作的环境，是标准的还是 CoNDIS 环境：

- 对于标准广域网微端口驱动程序，微端口驱动程序的 MiniportInitialize 函数首先为一个或多个 NIC 完成初始化。然后，NDISWAN 驱动程序使用 OID_WAN_GET_INFO 查询微端口驱动程序。然后 NDIS 调用微端口驱动程序的 MiniportQueryInformation 函数并且传递此查询请求。有 TAPI 能力的微端口驱动程序应当设置 NDIS_WAN_INFO 结构的成员 **FrameingBits** 的 TAPI_PROVIDER 位。设置 TAPI_PROVIDER 位通知上层驱动程序将 TAPI OID 发送给微端口驱动程序。标准广域网微端口驱动程序使用 TAPI OID 来建立、使用、拆卸 TAPI 连接。如何设置使用 TAPI 服务的标准广域网微端口驱动程序将在 8.6 节详细描述。
- 对于 CoNDIS 广域网微端口驱动程序，微端口驱动程序在函数 MiniportInitialize 内调用 **NdisCmRegisterAddressFamily** 函数，注册呼叫管理器入口点和地址族类型 CO_ADDRESS_FAMILY_TAPI_PROXY。通过这样做，微端口驱动程序通告上层驱动程序它实现 TAPI 服务。然后 NDIS 将新注册的地址族通知给 NDPROXY。NDPROXY 确定它可以使用呼叫管理器（集成在广域网微端口驱动程序之中）提供的 TAPI 服务。有 TAPI 能力的 CoNDIS 广域网微端口驱动程序可以是一个集成的微端口呼叫管理器（MCM）。NDPROXY 打开 TAPI-proxy 地址族，它联合微端口驱动程序，并向 NDIS 注册 NDPROXY 的面向连接的入口点，这些入口点被用于与微端口驱动程序的通信。一旦 NDPROXY 确定它可以使用微端口驱动程序的 TAPI 服务，它便列举微端口驱动程序的 TAPI 性能，然后发送封装在 NDIS 结构中的 TAPI 请求。 关于使用支持电话服务的 CoNDIS 扩展，详见 8.7 节。

8.4.3.2 查询广域网微端口驱动程序的信息

高层驱动程序用查询请求调用 **NdisRequest** 或 **NdisCoRequest** 函数，确定标准或 CoNDIS 广域网微端口驱动程序及其 NIC 的广域网性能和当前状态。在 NDISWAN 驱动程序转发查询请求之后，NDIS 调用广域网微端口驱动程序的查询请求函数。NDIS 调用的查询请求函数根据广域网微端口驱动程序的类型，是标准的或 CoNDIS，而有所不同：

- 对于标准广域网微端口驱动程序，这个查询请求函数是 *MiniportQueryInformation*。这个函数与局域网微端口驱动程序的相同，只是广域网微端口驱动程序可以识别标准广域网 OID。
- 对于 CoNDIS 广域网微端口驱动程序，这个查询请求函数是 *MiniportCoRequest*。这个函数与标准的面向连接的微端口驱动程序的相同，只是 CoNDIS 广域网微端口驱动程序可以识别 CoNDIS 广域网 OID。

若某个广域网微端口驱动程序需要通过返回状态 NDIS_STATUS_PENDING 异步地完成 **NdisRequest** 或 **NdisCoRequest**，它以后还必须通过调用一个完成函数（completion function）来完成查询。微端口驱动程序使用哪个完成函数取决于此驱动程序是标准还是 CoNDIS 类型的：

- 对于标准广域网微端口驱动程序，完成函数是 **NdisMQueryInformationComplete**。
- 对于 CoNDIS 广域网微端口驱动程序，完成函数是 **NdisCoRequestComplete**。

NDIS 在调用某个广域网微端口驱动程序的查询请求函数的时候，NDIS 传递一个指向 NDIS_REQUEST 结构的指针，这个结构包含了查询 OID 和一个用于保留从广域网微端口驱动程序获得的信息的缓冲区。广域网微端口驱动程序在完成请求之前控制此缓冲区。如果 NDIS_REQUEST 的成员 **InformationBufferLength** 中指定的字节数不足以存储此 OID 查询的信息，广域网微端口驱动程序将使此次查询请求失败，并置 NDIS_REQUEST 的成员 **BytesNeeded**（NDIS_REQUEST 的一个成员）为 OID 需要的字节数。

当前查询请求完成之前不会有其他请求再提交给特定的广域网微端口驱动程序。

广域网微端口驱动程序必须能够识别并正确的响应一组广域网查询 OID。这组 OID 取决于广域网微端口驱动程序的类型是标准还是 CoNDIS：

- 标准广域网查询 OID（Standard WAN query OIDs）。
- CoNDIS 广域网查询 OID（CoNDIS WAN query OIDs）。

标准广域网查询 OID

标准广域网微端口驱动程序必须能够识别并正确的响应的标准广域网查询 OID 有：

OID_WAN_MEDIUM_SUBTYPE 这个 OID 被用来查询标准广域网微端口驱动程序支持的介质子类型。标准广域网微端口驱动程序返回的信息应该是枚举 NDIS_WAN_MEDIUM_SUBTYPE 的子类型之一：

- **NdisWanMediumHub**
- **NdisWanMediumX_25**
- **NdisWanMediumIsdn**
- **NdisWanMediumSerial**
- **NdisWanMediumFrameRelay**
- **NdisWanAtm**
- **NdisWanSonet**
- **NdisWanMediumSW56K**
- **NdisWanMediumPPTP**
- **NdisWanMediumL2TP**
- **NdisWanMediumIrda**
- **NdisWanMediumParallel**

OID_WAN_GET_INFO 在标准广域网微端口驱动程序初始化之后，查询此 OID 来确定一些关于微端口驱动程序和它的 NIC 的信息。这个 OID 的结构的说明在 online DDK 的 *Network Drivers Reference* 中。标准广域网微端口驱动程序返回的信息包含在 NDIS_WAN_INFO 结

构的成员中。这些信息包括：

参数	内容
MaxFrameSize	驱动程序将此参数置为它所能发送和接收的最大帧长度（字节）。这个参数的典型值是 1500，但是驱动程序可以处理的字节数应比预测的最大值大 32 字节。这个值不包括驱动程序自己的组帧开销（framing overhead）或 PPP HDLC 开销。
MaxTransmit	广域网微端口驱动程序在一个线路上可以处理的未完成包(outstanding packet)数目的缺省最大值。NDISWAN 驱动程序确保微端口驱动程序不超过这个数字，具体来说，如果达到最大值，NDISWAN 将挂起发送包，除非微端口驱动程序在后来的 line-up 指示中重置此最大值。通过在 line-up 指示中传递一个新的非零 SendWindow 值，微端口驱动程序可以动态改变一条线路的最大传输值。
HeaderPadding	在最大包开始处填充所需缓冲区长度(字节)。
TailPadding	在最大包结尾处填充所需缓冲区长度(字节)。
Endpoint	NIC 最多所能支持的链接数。这是一个与适配器相关的特性。
MemoryFlags	如果 NIC 不是 DMA 设备则此标志为零。如果 NIC 支持 DMA，此标志可以被设为 NIDS_MEMORY_NONCACHED 或 / 和 NDIS_MEMORY_CONTIGUOUS。
HightestAcceptableAddress	如果广域网微端口驱动程序的 NIC 支持 DMA，置此标记为 NDIS_PHYSICAL_ADDRESS_CONST(-1,-1)；否则，如果 NIC 支持 DMA 且支持 24 位地址，置此标记为 NDIS_PHYSICAL_ADDRESS_CONST(0x1000000,0)。然后，指定范围内的内存将被分配给将要传送到 NIC 的包。这样可以通过 DMA 将内存的数据直接送到 NIC，而不必通过 CPU 在两块缓冲区之间复制。
FramingBits	驱动程序设置此标记置指定它支持的组帧类型。

OID_WAN_PERMANENT_ADDRESS 此 OID 用来查询 NIC 的硬件地址。
OID_WAN_CURRENT_ADDRESS 此 OID 用来查询 NIC 当前正使用的地址。

标准广域网微端口驱动程序为每个 NIC 返回一个唯一的地址。这个地址作为以太网地址被用于绑定协议。第一字节的最小有效位不能为 1，否则将被认为是以太网广播地址。如果供货商已经分配了一个以太网供货商编码，此编码应被用于确保地址不与其他供货商的地址冲突。

OID_WAN_GET_LINK_INFO 查询此 OID 来确定链接的当前状态。这个 OID 的结构的说明在 online DDK 的 *Network Drivers Reference* 中。标准广域网微端口驱动程序返回先前的 OID_WAN_SET_LINK_INFO 设置请求设置的值。如果先前没有设置请求发生，标准广域网微端口驱动程序将返回链接的当前状态。如果链接并不存在或处于非活动状态，将返回错误 NDIS_STATUS_INVALID_DATA。标准广域网微端口驱动程序返回的信息包含在结构 NDIS_WAN_GET_LINK_INFO 的成员中：

参数	内容

NdisLinkHandle	为此查询标识广域网微端口驱动程序的链接环境（link context）。
MaxSendFrameSize	广域网微端口驱动程序可以在此链接上发送的最大缓冲区长度（字节）。微端口驱动程序的 MiniportWanSend 函数可以拒绝接收大于此值的发送包。
MaxReceiveFrameSize	微端口驱动程序可以从网上接受的最大包（包括填充）。比这更大的包将被丢弃。
HeaderPadding	帧头填充长度（字节）。
TailPadding	帧尾填充的长度（字节）。
SendFramingBits	描述可以发送什么类型的帧的位掩码。
RecvFramingBits	描述可以接收什么类型的帧的位掩码。
SendCompressionBits	保留。
RecvCompressionBits	保留。
SendACCM	专用于异步介质类型。如果存在， 0-31 位分别指示需要字填充（byte stuff）的字。例如：如果位 0 被置为 1，ASCII 字符 0x00 应被字填充。
RecvACCM	同 SendACCM。

OID_WAN_GET_STATS_INFO 此 OID 用于查询标准广域网微端口驱动程序的 NIC 上的链接的状态。标准广域网微端口驱动程序应为它的链接保留统计量。这个 OID 的结构在 online DDK 的 *Network Drivers Reference* 中说明。标准广域网微端口驱动程序返回的信息包含在结构 NDIS_WAN_GET_STATS_INFO 的成员中，这些信息有：

成员	内容
NdisLinkHandle	为此查询标识广域网微端口驱动程序的链接环境(link context)。
BytesSent	发送了的字节数。
BytesRcvd	接收到的字节数。
FramesSent	发送的帧(广域网包)数。
FramesRcvd	接收到的帧数。
CRCErrors	链接遇到的 CRC 错误个数。CRC 错误是由循环冗余校验失败引起的。CRC 错误表明接收到的帧中的某些字节在传输过程中有错码。
TimeoutErrors	链接遇到的超时错误数。当预期的字节未被及时接收时，发生超时错误。
AlignmentErrors	链接遇到的定位错误数。当接收到的字节与预期的不同时，发生定位错误。这种错误发生的典型情况是有字节丢失或有超时错误发生。
SerialOverrunErrors	链接遇到的串行过速错误数。当广域网 NIC 不能以数据接收的速度处理数据时，发生串行过速错误。
FramingErrors	链接遇到的组帧错误数。当接收到有无效开始和结束位的异步字节时，发生组帧错误。
BufferOverrunErrors	链接遇到的缓冲过速错误数。当标准广域网微端口驱动程序不能以数据接收的速度处理数据时，发生缓冲过速错误。

CoNDIS 广域网查询 OID

CoNDIS 广域网微端口驱动程序必须能够识别并正确的响应面向连接的广域网查询

OID，概述如下：

OID_WAN_CO_GET_INFO 在 CoNDIS 广域网微端口驱动程序初始化之后，查询此 OID 以确定微端口驱动程序的 NIC 上所有 VC 的信息。这个 OID 的结构在 online DDK 的 *Network Drivers Reference* 中说明。CoNDIS 广域网微端口驱动程序返回的信息包含在 NDIS_WAN_CO_INFO 结构的成员中。这些信息包括：

成员	内容
MaxFrameSize	CoNDIS 广域网微端口驱动程序将此成员设为微端口驱动程序可以接收和发送的网络包的最大帧长度（字节）。此值应包括微端口驱动程序自己的组帧开销（framing overhead）和/或 PPP HDLC 开销。一般的，此值在 1500 左右，但微端口驱动程序应能处理比预期最大值大 32 字节的帧。
MaxSendWindow	CoNDIS 广域网微端口驱动程序将此成员置为驱动程序在每个 VC 上能处理的未完成包（outstanding packet）的最大数。NDISWAN 驱动程序确保微端口驱动程序不超过这个数字，也就是说，如果到达最大值，NDISWAN 将挂起发送包，除非微端口驱动程序在后继的链接参数（link-parameters）指示中重置此最大值。通过设置 WAN_CO_LINK_PARAMS 结构的成员 **SendWindow**，然后指示给 NDISWAN，微端口驱动程序可以为每个 VC 动态改变此值。
FramingBits	一个 32 位掩码，CoNDIS 广域网微端口驱动程序用此掩码指定微端口驱动程序支持的组帧类型。微端口驱动程序也可以指定它是否支持 TAPI 连接，以及是否执行它自己的组帧。
DesiredACCM	CoNDIS 广域网微端口驱动程序将此成员置为它需要的异步控制字符映射的类型。

OID_WAN_CO_GET_LINK_INFO 查询此 OID 以确定 VC 当前状态的 PPP 组帧（PPP framing）信息。这个 OID 的结构在 online DDK 的 *Network Drivers Reference* 中说明。CoNDIS 广域网微端口驱动程序返回先前用 OID_WAN_CO_SET_LINK_INFO 设置请求设置的值。如果先前没有设置请求发生，CoNDIS 广域网微端口驱动程序将返回 VC 的当前状态。如果 VC 并不存在或处于非活动状态，将返回错误 NDIS_STATUS_INVALID_DATA。CoNDIS 广域网微端口驱动程序返回的信息包含在结构 NDIS_WAN_CO_GET_LINK_INFO 的成员中。这些信息包括：

成员	内容
MaxSendFrameSize	微端口驱动程序在此 VC 上可以传输的最大缓冲区长度（byte）。微端口驱动程序的 *MiniportCoSendPackets* 函数可以拒绝接收大于此长度的发送包。
MaxRecvFrameSize	可以从网络上接收的最大包长度。微端口驱动程序可以丢弃大于此长度的包。
SendFramingBits	一个 32 位掩码，指定微端口驱动程序可以发送的组帧类型。
RecvFramingBits	一个 32 位掩码，指定微端口驱动程序可以接收的组帧类型。
SendACCM	专用于异步介质类型。如果存在，位 0-31 分别指示需要字填充（byte stuff）的字。例如：如果位 0 被置为 1，ASCII 字符 0x00 应被字填充。

RecvACCM	同 SendACCM。

OID_WAN_CO_GET_STATS_INFO 查询此 OID 以确定 VC 的状态。CoNDIS 广域网微端口驱动程序应为它的 VC 保留统计量。这个 OID 的结构在 online DDK 的 *Network Drivers Reference* 中说明。CoNDIS 广域网微端口驱动程序返回的信息包含在结构 NDIS_WAN_CO_GET_STATS_INFO 的成员中，这些信息有：

成员	内容
BytesSent	发送的字节数。
BytesRcvd	接收的字节数。
FrameSent	发送的帧（包）数。
FrameRcvd	接收的帧数。
CRCErrors	此 VC 遇到的 CRC 错误数。CRC 错误是由循环冗余校验失败引起的。CRC 错误表明接收到的帧中的某些字节在传输过程中有错码。
TimeoutErrors	VC 遇到的超时错误数。当预期的字节未被及时接收时，发生超时错误。
AlignmentErrors	VC 遇到的定位错误数。当接收到的字节与预期的不同时，发生定位错误。这种错误发生的典型情况是当字节丢失或有超时错误发生时。
SerialOverrunErrors	VC 遇到的串行过速错误数。当 WAN NIC 不能以数据接收的速度处理数据时，发生串行过速错误。
FramingErrors	VC 遇到的组帧错误数。当接收到有无效开始和结束位的异步字节时，发生组帧错误。
BufferOverrunErrors	VC 遇到的缓冲过速错误数。当 CoNDIS 广域网微端口驱动程序不能以数据接收的速度处理数据时，发生缓冲过速错误。

8.4.3.3 设置广域网小段口驱动程序的状态

高层协议使用设置请求调用 **NdisRequest** 或 **NdisCoRequest** 函数改变标准或 CoNDIS 广域网微端口驱动程序的 NIC 的信息。在 NDISWAN 驱动程序转发设置请求后，NDIS 调用广域网微端口驱动程序的设置请求函数。NDIS 调用哪个设置请求函数取决于广域网微端口驱动程序的类型，标准或 CoNDIS：

- 对于标准广域网微端口驱动程序，这个设置请求函数是 *MiniportSetInformation*。这个函数与局域网微端口驱动程序的相同，只是广域网微端口驱动程序可以识别标准广域网 OID。
- 对于 CoNDIS 广域网微端口驱动程序，这个设置请求函数是 *MiniportCoRequest*。这个函数与标准面向连接的微端口驱动程序的相同，只是 CoNDIS 广域网微端口驱动程序可以识别 CoNDIS 广域网 OID。

在当前设置请求完成之前，其他请求不会被提交给广域网微端口驱动程序。如果广域网微端口驱动程序不能立即完成设置请求，它必须在稍后调用一个完成函数（completion function）。广域网微端口驱动程序的设置请求函数返回 NDIS_STATUS_PENDING 表示它未完成设置请求。微端口驱动程序调用哪个完成函数依赖于广域网微端口驱动程序的类型，标

准或 CoNDIS：

- 对于标准广域网微端口驱动程序，这个完成函数是 **NdisMSetInformationComplete**。
- 对于 CoNDIS 广域网微端口驱动程序，这个完成函数是 **NdisCoRequestComplete**。

广域网微端口驱动程序可以使用所有的 NDIS 全局设置（global-set）OID。

广域网微端口驱动程序必须能够识别并正确的响应一组广域网设置 OID（WAN set OID）。这组广域网设置 OID 依赖于广域网微端口驱动程序的类型，标准或 CoNDIS：

标准广域网设置 OID

标准广域网微端口驱动程序必须能够识别并正确的响应以下标准广域网 OID。

OID_WAN_SET_LINK_INFO 此 OID 用于设置链接的特性。这个 OID 的结构在 online DDK 的 *Network Drivers Reference* 中说明。传递给标准广域网微端口驱动程序的信息包含在结构 NDIS_WAN_SET_LINK_INFO 的成员中。NDIS_WAN_SET_LINK_INFO 结构的成员与 NDIS_WAN_GET_LINK_INFO 的成员相同，这在标准广域网查询 OID 中有描述。

CoNDIS 广域网设置 OID

CoNDIS 广域网微端口驱动程序必须能够识别并正确的响应以下面向连接的广域网 OID。

OID_WAN_CO_SET_LINK_INFO 此 OID 用于为特定 VC 改变 PPP 组帧信息。这个 OID 的结构的说明在 online DDK 的 *Network Drivers Reference* 中。传送给 CoNDIS 广域网微端口驱动程序的信息包含在结构 NDIS_WAN_CO_SET_LINK_INFO 的成员中。这在 CoNDIS 广域网查询 OID 中有描述。

8.4.3.4 在广域网微端口驱动程序上发送数据

上层协议调用 **NdisSend** 或 **NdisCoSendPackets** 函数巴包发送给下层广域网微端口驱动程序。NDISWAN 驱动程序转发从上层驱动程序发送来的包。在转发包之前，NDISWAN 将之重新打包。对于标准广域网微端口驱动程序，在转发之前，NDISWAN 将包的结构从 NDIS_PACKET 变为 NDIS_WAN_PACKET 结构。对于 CoNDIS 广域网微端口驱动程序，NDISWAN 用新的 NDIS_PACKET 结构转发包。

NIDSWAN 转发包之后，NDIS 调用广域网微端口驱动程序的发送函数。NDIS 调用哪个发送函数取决于广域网微端口驱动程序的类型是标准还是 CoNDIS：

- 对于标准广域网微端口驱动程序，此发送函数是 *MiniportWanSend*。MiniportWanSend 函数将包通过 NIC 发送到广域网。
- 对于 CoNDIS 广域网微端口驱动程序，此发送函数是 *MiniportCoSendPacket*。关于 CoNDIS 广域网微端口驱动程序如何发送包的更多信息，参见第四部分的 1.6.7.1，面向连接的 NDIS。

不论是同步还是异步，包描述信息和包数据的所有权都被传递给广域网微端口驱动程序，直到包发送完毕为止。广域网微端口驱动程序的发送函数返回 NDIS_STATUS_PENDING，表示它还未完成包的传送，它需要调用一个完成函数完成发送。微端口驱动程序以后调用哪个完成函数取决于微端口驱动程序的类型是标准还是 CoNDIS：

- 对于标准广域网微端口驱动程序，此完成函数是 **NdisMWanSendComplete**。
- 对于 CoNDIS 广域网微端口驱动程序，此完成函数是 **NdisMCoSendComplete**。

这些完成函数的调用不一定表时包已经被传送，尽管一般来说包已经被传送（除智能NIC 外）。这些调用实际上是指广域网微端口驱动程序已经准备好释放包的所有权。

如果广域网微端口驱动程序返回一个不是 NDIS_STATUS_PENDING 的状态，则认为发送操作已经结束，包的所有权立刻返回到调用者。在这种情况下，微端口驱动程序不必调用完成函数。

在包被广域网微端口驱动程序接收到之后，应当在当前连接上将包发送出去。对于标准环境，此连接是一个已建立的链接。对于 CoNDIS 环境，此连接是一个 VC。

与局域网微端口驱动程序不同的是，广域网微端口驱动程序不可以返回状态 NDIS_STATUS_RESOURCES 以指示它没有足够的资源处理包的传送。取而代之的是，广域网微端口驱动程序应当将包放在发送队列中等待一段时间，并可能还要降低此连接的发送窗口的值。发送窗口的值是广域网微端口驱动程序在每个连接上可以处理的未完成包的个数。发送窗口的值由微端口驱动程序为上层驱动程序指定。广域网微端口驱动程序指定发送窗口值的方法取决于广域网微端口驱动程序的类型是标准还是 CoNDIS：

- 标准广域网微端口驱动程序通过 NDIS_WAN_INFO 结构的成员 **MaxTransmit** 指定每个数据信道可以处理的未完成包个数的缺省值。微端口驱动程序在响应协议驱动程序的 OID_WAN_GET_INFO 请求时提供此结构。然而，微端口驱动程序通过以 line-up 指示调用函数 **NdisMIndicateStatus**，可以在每条线路的基础上动态操纵发送窗口。在此调用中，微端口驱动程序为参数 NDIS_WAN_LINE_UP 结构的成员 **SendWindow** 提供一个新的非零值。如果微端口驱动程序将 **SendWindow** 值置为零，NDISWAN 将使用 **MaxTransmit** 中的缺省值作为此线路的值。标准广域网微端口驱动程序可能必须修改或添加包头，以及添加包尾（例如，添加 FCS）。微端口驱动程序预先提供在包的头和尾的适当填充。为了在广域网介质上发送包，广域网微端口驱动程序可以随意以它认为合适的方法改变包中的数据。
- CoNDIS 广域网微端口驱动程序通过 NDIS_WAN_CO_INFO 结构的成员 **MaxSendWindow**，来指定每个 VC 可以处理的未完成包的个数。微端口驱动程序在响应协议驱动程序的 OID_WAN_CO_GET_INFO 请求时提供此结构。然而，通过设置传送给函数 NdisMCoIndicateStatus 的 WAN_CO_LINKPARAMS 结构的成员 **SendWindow**，微端口驱动程序可以在每个 VC 的基础上动态调节此数值。NDISWAN 使用当前 **SendWindow** 值作为对未完成发送的限制。微端口驱动程序可以通过将 **SendWindow** 置零规定不可以处理任何未完成包。也就是说，把 **SendWindow** 成员置为零，关闭发送窗口并且 NDISWAN 停止为特定 VC 发送包。

广域网微端口驱动程序也不能调用 **NdisMSendResourcesAvailable** 函数。

如果已设置 PPP 组帧，传递给广域网微端口驱动程序的发送函数的包将包含简单 HDLC PPP 组帧。对于 SLIP 或 RAS 组帧，包只包含数据部分，不包含组帧数据。简单 HDLC PPP 组帧在后文中有消息讨论。

广域网 NIC 驱动程序不得试图提供软件回放或混杂模式回放（promiscuous-mode loopback）。NDISWAN 驱动程序全面支持这两种回放类型。

对于标准广域网微端口驱动程序，以下内容适用于广域网包。广域网包描述信息结构的详细信息见 online DDK 中 Network Drivers Reference 中的 NDIS_WAN_PACKET。

- NDIS_WAN_PACKET 结构的成员 **MacReserved**x 和 **WanPacketQueue** 对标准广域网微端口驱动程序是完全可用的。
- 可用的头填充只有 **CurrentBuffer-StartBuffer**。可用的尾填充是

EndBuffer-(CurrentBuffer+CurrentLength)。头和尾填充至少应满足所需要的数量，但可以更多。

8.4.4 广域网微端口驱动程序做出的指示

标准和 CoNDIS 广域网微端口驱动程序有不同的方法向 NDISWAN 指示状态和接收到的数据。

指示广域网专用的状态：

- 标准广域网微端口驱动程序与局域网微端口 NIC 驱动程序相同，调用 **NdisMIndicateStatus** 函数。
- CoNDIS 广域网微端口驱动程序与面向连接的微端口驱动程序相同，调用 **NdisMCoIndicateStatus** 函数。

指示从广域网接收到的数据：

- 标准广域网微端口驱动程序调用函数 **NdisMWanIndicateReceive**，而不是调用局域网微端口驱动程序的类似函数。
- CoNDIS 广域网微端口驱动程序与面向连接的微端口驱动程序相同，调用 **NdisMCoIndicateReceivePacket** 函数。

以下各节描述了标准广域网微端口驱动程序如何指示从广域网接收到的数据，并且描述了标准和 CoNDIS 广域网微端口驱动程序可以指示给 NDISWAN 驱动程序的广域网专用的状态类型。

- 8.4.4.1 指示从标准广域网微端口驱动程序接收数据
- 8.4.4.2 指示广域网微端口驱动程序状态

8.4.4.1 指示从标准广域网微端口驱动程序接收数据

标准广域网微端口驱动程序调用 **NdisMWanIndicateReceive** 函数指示包已到达且整个包已准备好检查。在此调用之后，NDISWAN 将包到达的信息指示给绑定的上层驱动程序的 ProtocolReceive 处理程序。可以检查整个包是因为微端口驱动程序不传递任何先行数据。

因为整个包总是向上传递，微端口驱动程序永远不会接收到传输数据的请求（数据被 NDISWAN 复制并向上传递到下一个高层驱动程序）。因为包可能被压缩和加密，所以整个包总是向上传递。同样，因为链接为点对点的，至少应有一个绑定的协议处理包。

包的报头中的数据与 NIC 接收的相同。NIC 驱动程序不会消除它接收到的任何报头或报尾数据。用来传送的驱动程序不能给包增加任何填充。

广域网微端口驱动程序调用函数 **NdisMWanIndicateReceiveComplete** 指示一个或多个接收指示的结束，以便协议加快处理接收到的包。作为结果，NDISWAN 调用绑定协议的 *ProtocolReceiveComplete* 处理程序通知每个协议可以开始处理接收到的数据了。在协议的接收完成处理程序中，协议的处理过程不受严格的时间限制，但在接受处理程序中受此限制。

协议应该假定 ProtocolReceiveComplete 调用是可以被中断的。在 SMP 计算机中，接受处理程序和接收完成处理程序可以在不同处理器中并行运行。

注意，标准广域网微端口驱动程序的 **NdisMWanIndicateReceive** 与 **NdisMWanIncicateReceiveComplete** 可以不一一对应。微端口驱动程序可以为多个接收指示发布一个接收完毕指示。例如：一个标准广域网微端口驱动程序可以在每完成十个包或退出处理程序时，调用一个 **NdisMWanIncicateReceiveComplete**。

8.4.4.2 指示广域网微端口驱动程序状态

广域网微端口驱动程序应该保存一组广域网专用的统计信息。广域网微端口驱动程序用唯一的状态代码和缓冲区内容指示状态。在广域网 NIC 的状态发生变化后，广域网微端口驱动程序调用特定的状态指示函数将这些变化指示给绑定的上层协议。广域网微端口驱动程序生成这些状态指示。绑定的协议驱动程序可以忽略这些指示。然而，一般来说执行这些指示可以改善协议和微端口驱动程序的性能。NDISWAN 驱动程序将状态指示传递给绑定协议的 *ProtocolStatus* 函数或配置管理器。这些协议或配置管理器能够记录指示并可能产生校正动作。广域网微端口驱动程序调用哪个状态指示函数取决于广域网微端口驱动程序的类型是标准还是 CoNDIS：

- 对于标准广域网微端口驱动程序，这个状态指示函数是 **NdisMIndicateStatus**。此函数与局域网微端口 NIC 驱动程序的相同，只是广域网微端口驱动程序指示的是一组广域网专用的统计数据。
- 对于 CoNDIS 广域网微端口驱动程序，这个状态指示函数是 **NdisMCoIndicateStatus**。此函数与标准面向连接的微端口 NIC 驱动程序中的相同，只是 CoNDIS 广域网微端口驱动程序为每个 VC 指示一组广域网专用的统计数据。微端口驱动程序使用一个显式的 VC 句柄调用 **NdisMCoIndicateStatus**，将变化指示给共享此 VC 的上层协议驱动程序。

每个状态指示提供两部分信息：

- 表示通用状态的状态码。已定义的通用状态码的个数是有限的；这些状态码在将来可以扩充。
- 保存状态信息的缓冲区。状态信息可以是某个特定的 NIC 的或 CoNDIS 广域网微端口驱动程序的，或是 NIC 上某个 VC 的。例如，缓冲区内保存一个 X.25 连接的新的传输速度。对于标准广域网微端口驱动程序，此缓冲区可能是指示一个新的链接变成活动的或一个已存在的链接变为空闲状态。

广域网微端口驱动程序做出的状态指示取决于广域网微端口驱动程序的类型是标准还是 CoNDIS，这将在下面两节描述：

- 标准广域网微端口驱动程序状态指示
- CoNDIS 广域网微端口驱动程序状态指示

标准广域网微端口驱动程序状态指示

标准广域网微端口驱动程序状态指示的一般形式是 NDIS_STATUS_WAN_*XXX*。三种状态指示是：

- NDIS_STATUS_WAN_LINE_UP

标准广域网微端口驱动程序调用 **NdisMIndicateStatus** 指示新的数据信道变为活动的或者是已存在的数据信道的参数发生改变。在此调用中，微端口驱动程序在参数 *GeneralStatus* 中传递 NDIS_STATUS_WAN_LINE_UP，在参数 *StatusBuffer* 中传递指向 NDIS_MAC_LINE_UP 结构的指针。NDIS_MAC_LINE_UP 为特定链接标识数据信道，并描述此链接的参数。在此指示发生之前，广域网 NIC 可以接受包并能够使它们成功或失败，但这样的包却未必可以被远程节点的进程接收到。在数据信道活动前，协议应当减少计时器并重新计数以便任何外出的连接企图迅速失败。

标准广域网微端口驱动程序必须在从 OID_TAPI_GET_ID 请求返回之前发出初始的 line-up 指示。此请求使用了 NDIS_TAPI_GET_ID 结构。如果 NDIS_TAPI_GET_ID 的成员 **DeviceClass** 的值是 NDIS，并且成员 **ulSelect** 的值是 LINECALLSELECT_CALL，必须进行 line-up 指示。在 OID_TAPI_GET_ID 请求完成之前，微端口驱动程序将 NDIS_TAPI_GET_ID 的成员 **DeviceID** 的值置为从 line-up 指示返回的 NDIS_MAC_LINE_UP 结构的成员

NdisLinkContext 的值。Line-up 指示的更多信息，参见 8.6.12 的 Line-up 指示。

- NDIS_STATUS_WAN_LINE_DOWN

当链接断开时标准广域网微端口驱动程序调用 **NdisMIndicateStatus**。在此调用中，微端口驱动程序在参数 *GeneralStatus* 中传递 NDIS_STATUS_WAN_LINE_DOWN，在参数 *StatusBuffer* 中传递指向 NDIS_MAC_LINE_DOWN 结构的指针。NDIS_MAC_LINE_DOWN 的成员 **NdisLinkContext** 指出不再有效的链接。

在此状态指示发生之后，绑定的协议应当减少计数器并重新计数，直到下一 line-up 指示出现。

- NDIS_STATUS_WAN_FRAGMENT

当从远程节点接收到不完全的包时，标准广域网微端口驱动程序调用 **NdisMIndicateStatus**。在此调用中，微端口驱动程序在参数 *GeneralStatus* 中传递 NDIS_STATUS_WAN_FRAGMENT，在参数 *StatusBuffer* 中传递指向 NDIS_MAC_FRAGMENT 结构的指针。NDIS_MAC_FRAGMENT 指出特定的链接并描述接收到不完全包的原因。

当从远程对等节点接收到不完全的包时，如果标准广域网微端口驱动程序不进行 FRAGMENT 指示，NDISWAN 将不会发现不完全包错误。在 FRAGMENT 指示发生后，协议应当发送帧到远程节点通知远程节点 FRAGMENT 指示，而不是等待超时发生。NDISWAN 通过计算链接上发生的 FRAGMENT 指示次数记录丢弃包的数量。

标准广域网微端口驱动程序还可以指示 TAPI 状态。微端口驱动程序调用 **NdisMIndicateStatus** 指示 TAPI 状态。在此调用中，微端口驱动程序在参数 *GeneralStatus* 中传递 NDIS_STATUS_TAPI_INDICATION，在参数 *StatusBuffer* 中传递指向 NDIS_MAC_EVENT 结构的指针。NDIS_MAC_EVENT 记述发生的 TAPI 线路或呼叫事件。微端口驱动程序开发者关心的 TAPI 状态指示类型有线路状态的改变、呼叫状态的改变、内入呼叫的到达、远程节点或微端口驱动程序关闭现存线路或呼叫的事件。详见 8.6 节。

CoNDIS 广域网微端口驱动程序状态指示

CoNDIS 广域网微端口驱动程序的 VC 状态指示的一般形式是 NDIS_STATUS_WAN_CO_*XXX*。两种 VC 状态指示类型是：

- NDIS_STATUS_WAN_CO_LINKPARAMS

CoNDIS 广域网微端口驱动程序调用 **NdisMCoIndicateStatus** 指示在 NIC 上活动的特定 VC 的参数发生改变。在此调用中，微端口驱动程序在参数 *NdisVcHandle* 中传递 VC 的句柄，在参数 *GeneralStatus* 中传递 NDIS_STATUS_WAN_CO_LINKPARAMS，在参数 *StatusBuffer* 中传递一个指向 WAN_CO_LINKPARAMS 结构的指针。WAN_CO_LINKPARAMS 中存放 VC 的新参数。

- NDIS_STATUS_WAN_CO_FRAGMENT

CoNDIS 广域网微端口驱动程序调用 **NdisMCoIndicateStatus** 指示它从 VC 的另一个端点接收到了不完全的包。在此调用中，微端口驱动程序在参数 *NdisVcHandle* 中传递 VC 的句柄，在参数 *GeneralStatus* 中传递 NDIS_STATUS_WAN_ CO_FRAGMENT，在参数 *StatusBuffer* 中传递指向 WAN_CO_FRAGMENT 结构的指针。WAN_CO_FRAGMENT 结构描述接收到不完全包的原因。

此指示发生后，面向连接客户应当向 VC 另一端的面向连接客户发送帧，这些帧通知对应端点它接收到了不完全包，而不是等超时发生。

NDISWAN 通过计算每个 VC 上发生的 FRAGMENT 指示次数记录丢弃包的数量。

8.5 广域网包的组帧

本节提供了广域网包帧结构的信息。

- 8.5.1 异步帧结构
- 8.5.2 X.25 帧结构
- 8.5.3 ISDN 和 switched-56K 帧结构

8.5.1 异步帧结构

DDK 提供了一个支持 SLIP 和 RAS 压缩的简单异步微端口 NIC 驱动程序

图 8.3 关闭压缩的异步帧结构

8.5.2 X.25 帧结构

图 8.4 X.25 帧结构（协议字段压缩开或关）

X.25 不能启用地址和控制字段压缩。如果协议字段压缩被打开，NLPID 将被去除。提供透明性、添加任何填充、添加 FCS 和添加结束标记是 X.25 驱动程序或硬件的责任。

8.5.3 ISDN 和 Switched-56K 帧结构

最初应使用 ISDN B 信道（不是 D 信道）。通过 NDISWAN 的多链接支持支持多个 B 信道。最初，应使有 NRZ 编码的位同步 HDLC 帧格式。驱动程序或 ISDN 硬件应提供透明性。ISDN 驱动程序或 ISDN 硬件还应提供 NRZ 编码、计算 FCS、添加 PPP 结束标记（0x7E）、以及插入帧间时间填充。Switched-56K 驱动程序以与 ISDN 的相同的方式组帧。

8.6 标准 NDIS 之上的电话服务扩展

本节描述了标准广域网 NIC 微端口驱动程序如何通过使用 NDISTAPI 驱动程序和 NDISWAN 驱动程序的功能实现电话服务。NDISTAPI 驱动程序通过一个电话服务提供者与电话应用程序通信。详见 Platform SDK 的电话应用编程接口。

为了在标准广域网 NIC 微端口驱动程序上实现电话服务，以下节更详细的描述了 NDISTAPI 驱动程序：

- NDISTAPI 概述
- 线路设备、地址和呼叫
- TAPI 设置和查询请求
- 保持关于呼叫状态的信息
- 建立呼叫句柄
- TAPI 注册
- TAPI 初始化
- 打开一个呼叫线路（call line）
- 接受内入呼叫

- 产生 TAPI 呼叫
- 主动事件处理
- 指示呼叫线路状态
- 关闭呼叫线路
- NDISTAPI 接口

8.6.1 NDISTAPI 概述

NDISTAPI 驱动程序定义了标准广域网 NIC 微端口驱动程序如何实现电话服务。它与 Windows Telephony 的服务提供接口有关，为兼容 TAPI 的 NDIS，微端口 NIC 驱动程序为用户模式 TAPI 服务提供者提供核心模式支持。本节介绍了兼容 TAPI 的标准广域网微端口驱动程序如何初始化、建立线路、使用 TAPI 和 WAN OIDs 建立呼叫和关闭呼叫。在这些介绍中简单的介绍了 Windows Telephony 中的一些概念，但读者应参考 Telephony SDK 中的文档得到更深入的了解。

兼容 TAPI 的 NDIS 微端口 NIC 驱动程序将自己同时注册和初始化为广域网和 TAPI 服务的用户。在注册和初始化完成之后，用户层的应用程序可以向用户模式的 KMDDSP 服务提供者模块提出电话请求，此模块将 SPI 请求转换为相应的 OID。KMDDSP 模块将 OID 传递给 TAPI 的内核模式组件——NDISTAPI 驱动程序。为了建立、监视、拆除线路和呼叫，NDISTAPI 将 OID 发送给恰当的 NIC 驱动程序。NDISTAPI 在 NdisRequest 调用中将 OIDs 传递给 NDISWAN 中间驱动程序。NDISWAN 继续传递这些 NDIS 请求，这导致了对兼容 TAPI 的标准广域网微端口驱动程序的 MiniportQueryInformation 函数或 MiniportSetInformation 函数的调用。

兼容 TAPI 的标准广域网微端口驱动程序通知线路和呼叫状态的变化，如从进行中状态变为连接状态、内入呼叫的到达、或者远程断连。通过将适当的 NDIS_STATUS_TAPI_*XXX* 和 NDIS_STATUS_WAN_*XXX* 状态消息传递给 NdisMIndicateStatus，微端口驱动程序将状态变化通知上层。

本节描述的各部件之间的关系参见 8.1 节的 RAS 体系结构图。内核模式通信的流动方向是从 NDISTAPI 到 NDISWAN，然后通过 NDIS 到兼容 TAPI 的标准广域网微端口驱动程序。

8.6.2 线路设备、地址和呼叫

兼容 TAPI 的标准广域网 NIC 微端口驱动程序模拟线路、地址和呼叫。线路设备是对一个拥有一个或多个通信信道（用于传送信号或/和信息）的物理设备的抽象。如何模拟线路设备由广域网微端口 NIC 驱动程序开发者自行决定。例如，考虑一个有两个 64K ”B” 信道的 BRI-ISDN 线路。标准广域网微端口驱动程序可以选择模拟以下的几种选择：

- 一个线路设备，它有两个信道
- 两个线路设备，每个线路设备都有一个信道
- 两个线路设备，每个都支持一个或两个信道，从公用池中选择
- 两个单信道线路设备和一个双信道线路设备

在后两种模型中，信道可以在不同时间被分配给不同线路设备。

每个线路设备可以分配一个或多个地址。每个地址相当于一个电话号码。地址 ID 是一个介于 0 到此线路上的地址数减一之间的数字。

从 TAPI 客户的观点来看，线路设备的数目和相应的地址数目始终保持在客户调用 **lineInitialize**（初始化 TAPI 客户会话）的次数和调用 **lineShutdown**（结束 TAPI 客户会话）的次数之间。然而，线路设备和地址的状态常常在客户的电话会话的生命期中改变。例如，

线路设备可能会因某些原因停止服务，并稍后又变正常。一些请求和通知消息允许客户应用追踪线路和地址的状况和性能。

一个呼叫代表两个或多个地址间的连接。与线路设备和地址不同的是，呼叫可以动态的创建和销毁。同线路设备和地址相同的是，它们都定义了请求和通知消息，允许客户应用程序追踪呼叫状态和信息的变化。

8.6.3 设置和查询请求

TAPI 请求以 OID_TAPI_*XXX* 的形式，连同相关的结构，传送给标准广域网微端口驱动程序。如果微端口 NIC 驱动程序应当给调用者返回信息，在 **NdisRequest** 调用中应指定 **NdisRequestQueryInformation** 请求类型。对于除完成状态外不返回任何信息的请求，应指定 **NdisRequestSetInformation** 请求类型。NDIS 库发送来自 NDISWAN 的 **NdisRequest** 调用给目标微端口驱动程序的 *MiniportQueryInformation* 函数或 *MiniportSetInformation* 函数。

所有发送给标准广域网微端口驱动程序的请求都既可以同步完成也可以异步完成。如果 NIC 驱动程序不能立即完成请求，它只返回 NDIS_STATUS_PENDING,并在它完成请求后调用 **NdisMQueryInformationComplete** 或 **NdisMSetInformationComplete**。

8.6.4 保持状态信息

标准广域网微端口驱动程序必须跟踪它管理的链接上的状态的变化，如当前呼叫、地址、线路的状态，并通过调用 **NdisMIndicateStatus** 向 NDISWAN 报告这些变化。

当呼叫状态改变时，微端口驱动程序必须总是发送呼叫状态改变消息。报告呼叫状态消息的格式是 LINECALLSTATE_*XXX* 和 LINECALLINFOSTATE_*XXX*。所有的呼叫状态消息都不会被过滤。

微端口驱动程序报告的一些典型的呼叫状态包括 LINECALLSTATE_IDLE、LINECALLSTATE_CONNECTED、LINE_CALLSTATE_OFFERING、LINECALLSTATE_DISCONNECT、LINECALLINFOSTATE_CALLERID 和 LINECALLINFOSTATE_MEDIAMODE。

只有在被 OID_TAPI_SET_STATUS_MESSAGE 请求使能时，微端口驱动程序才指示线路状态或地址状态的改变。

对于地址状态的改变，微端口驱动程序用 LINEADDRESSSTATE_*XXX* 格式表示，例如 LINEADDRESSSTATE_INUSEMANY 和 LINEADDRESSSTATE_FORWARD 状态。

对于线路状态的改变，微端口驱动程序用 LINEDEVSTATE_*XXX* 格式表示，例如 INEDEVSTATE_RISING、INEDEVSTATE_CONNECTED、INEDEVSTATE_OPEN、和 LINEADDRESSSTATE_CLOSED 状态。

标准广域网 NIC 微端口驱动程序可以接收一个或多个 OID_TAPI_SET_STATUS_MESSAGE 请求，此请求通知微端口驱动程序哪个状态的改变必须报告。最初，微端口驱动程序应当认为除 LINEDEVSTATE_REINIT（永远不会失效）外的所有线路和地址状态指示都时无效的。

当标准广域网微端口驱动程序调用 **NdisMIndicateStatus** 报告状态改变时，*GeneralStatus* 参数总是 NDIS_STATUS_TAPI_INDICATION。*StausBuffer* 参数是一个指向 NDIS_TAPI_EVENT 结构的指针。微端口驱动程序在 NDIS_TAPI_EVENT 结构的成员 **htLine** 和 **htCall** 中标识状态被改变的线路和呼叫。NDIS_TAPI_EVENT 结构的成员 **uMsg** 被置为像 LINE_CALLSTATE 这样的值，**ulParaml** 描述被报告的状态。

8.6.5 建立句柄

在 NDISTAPI 驱动程序将标准广域网微端口驱动程序初始化为提供者之后，微端口驱动程序可以接受一个打开线路的请求，此请求使用 OID_TAPI_OPEN 和适当参数调用 MiniportQueryInformation 函数。接收到打开请求后，微端口驱动程序一般来说应当作以下操作：

- 在微端口驱动程序的内部结构中将线路设备标记为开放的
- 保存 NDISTAPI 驱动程序的线路设备句柄（在 **htLine** 成员中传递的），这将在后续的事件通知中使用。
- 返回它自己的线路设备句柄（在 **hdLine** 成员中传回）。

再一次强调，此请求的发起者是调用 **lineOpen** 的客户进程。

在客户进程成功的打开线路后，它可以通过调用 **lineMakeCall** 在此线路上放置一个呼叫。这将导致 NDISTAPI 驱动程序用 OID_TAPI_MAKE_CALL 调用 **NdisRequest**，在此调用中，在 **htLine** 中指定 NDISTAPI 提供的线路句柄，在 **htCall** 中指定 NDISTAPI 提供的呼叫句柄。此请求被传递给微端口驱动程序的 MiniportQueryInformation 函数（指定 OID_LINE_MAKE_CALL）。如果 NIC 驱动程序能成功创建此呼叫，它在 **hdCall** 中传回它自己的呼叫句柄。

两级 **ht**Xxx/**hd**Xxx（“TAPI 对象 XXX 的句柄/驱动程序对象 XXX 的句柄”）句柄方案为 NDISTAPI 驱动程序和标准广域网微端口驱动程序提供了为同一个逻辑对象定义它们自己的含义的灵活性。到 NIC 驱动程序的请求指定驱动程序句柄（**hdLine** 和 **hdCall**），到 NDISTAPI 驱动程序的通知包含 TAPI 句柄（**htLine** 和 **htCall**）。在以上情况下微端口 NIC 驱动程序为某些对象定义了驱动程序句柄(例如：**hdLine** 和 **hdCall)**，一般来说指定一个指向适当的代表对象的驱动程序内部结构的指针作为句柄，这对 NIC 驱动程序是有利的。

当客户进程完成呼叫后，它将调用 **lineDrop** 丢弃呼叫，然后调用 **lineDeallocateCall** 收回呼叫实例。这将导致 NDISTAPI 驱动程序分别发出两个请求，OID_TAPI_DROP 和 OID_TAPI_CLOSE_CALL，这两个请求通过 NDISWAN 传递到标准广域网微端口驱动程序以丢弃呼叫和释放呼叫实例。微端口驱动程序也有可能收到 OID_TAPI_CLOSE_CALL 请求，而在这之前没有收到 OID_TAPI_DROP 请求，在这种情况下，微端口驱动程序应当丢弃呼叫和释放呼叫实例，就像两个 OID 都收到一样。

最后，当最后一个 TAPI 客户调用了 **lineClose**，NDISTAPI 驱动程序将在它的函数 *MiniportSetInformation* 中调用 **NdisRequest** 发送给微端口驱动程序一个 OID_TAPI_CLOSE 请求。OID_TAPI_CLOSE 也可以在一个活动线路上收到。在这种情况下，微端口驱动程序必须完成或中断此设备上的所有的未完成呼叫和异步请求，并关闭线路。带有有效的 **hdLine** 的 OID_TAPI_CLOSE 请求不会失败。

8.6.6 TAPI 注册

标准广域网微端口驱动程序通过两个步骤设置它自己可以接受 TAPI 命令。当微端口驱动程序在它的函数 **DriverEntry** 中通过调用 **NdisMRegisterMiniport** 向 NDIS 注册时，它通过在 NDIS*XX*_MINIPORT_CHARCTERISTICS 结构的成员 **Reserved** 中设置标记 NDIS_USE_WAN_WRAPPER 指出它需要广域网服务。

在运行 **DriverEntry** 函数和在 *MiniportInitialize* 函数完成一个或多个适配器的初始化之后，NDISWAN 驱动程序将用 OID_WAN_GET_INFO 请求，对标准广域网微端口驱动程序的 *MiniportQueryInformation* 函数进行 **NdisRequest** 调用。兼容 TAPI 的微端口驱动程序应将

给此调用中传递的 NDIS_WAN_INFO 结构的成员 **FramingBits** 的 TAPI_PROVIDER 位置位。

兼容 TAPI 的标准广域网微端口驱动程序将 NDIS_WAN_INFO 结构的成员 **MaxTransmit** 置为微端口驱动程序可接收未完成包的最大数。NDIS 在到达此最大值时挂起发送包。然而，兼容 TAPI 的微端口驱动程序可以动态提供一个不同于最初缺省值的新的发送窗口。在微端口驱动程序向 **NdisMIndicateStatus** 指示 line-up 状态时，它通过设置 NDIS_WAN_LINE_UP 结构的 **SendWindow** 可以为此线路提供一个新的非零值。如果微端口驱动程序传递了一个为零的 **SendWindow**，NDISWAN 将使用缺省值——在此线路初始化时设置的 **MaxTransimit**。

就像广域网驱动程序的 MiniportInitialize 函数被用来初始化它管理的每个适配器，NDISWAN 调用 NDISTAPI 注册每个 TAPI 兼容的适配器。这样就为特定的注册过的适配器建立了一个联系，NDISTAPI 可以通过 NDISWAN 向微端口驱动程序发送电话请求。

8.6.7 TAPI 初始化

在标准广域网微端口驱动程序注册、初始化、并指示自己是一个 TAPI 提供者之后，将使用 OID_TAPI_PROVIDER_INITIALIZE 请求调用 MiniportQueryInformation 函数。在提供者和 TAPI 兼容适配器之间有一对一关系。传递给调用的 NDIS_TAPI_PROVIDER_INITIALIZE 结构中的 **ulDeviceIDBase** 表示偏移量，若在后续的 OID 响应中涉及到设备，微端口驱动程序应当将它以零为基准的线路设备标识加上此偏移量作为设备标识。如果微端口驱动程序管理两个设备，并且 **ulDeviceIDBase** 为 n，这些设备在后续的调用中为 n 和 n+1。

在从 MiniportQueryInformation 返回之前，标准广域网微端口驱动程序应当填写 OID 结构的两个成员。**UlNumLineDevs** 被置为适配器支持的线路设备数。**ulProviderID** 应被置为一个唯一（每个适配器）的值。因为当前无法保证 **ulProviderID** 值的唯一性，将 NDIS 句柄作为 **ulProviderID** 传递给 *MiniportInitialize* 是一个好的策略。此结构的 **ulRequestID** 成员保留给 NDISTAPI，并且对微端口驱动程序不是透明的。

发送给已注册的标准广域网微端口驱动程序的 OID_TAPI_PROVIDER_INITIALIZE 请求总是由第一个客户进程载入和初始化的高层电话模块产生。对每个适配器，微端口驱动程序只在第一个客户进程作出 **lineInitialize** 请求时接收一次 OID_TAPI_PROVIDER_INITIALIZE 请求。

通信设备特性

在标准广域网微端口驱动程序返回它支持的设备数后，它的 MiniportQueryInformation 函数将以参数 OID_TAPI_GET_DEV_CAPS 被调用，以返回设备的性能；以参数 OID_TAPI_GET_DEV_CONFIG 被调用，以返回设备的配置。这些信息描述了广域网微端口驱动程序如何模拟它的设备。例如，同一 ISDN 适配器可被模拟为：

- 一个线路，此线路上有一个地址，每个地址支持两个呼叫
- 一个线路，此线路上有两个地址，每个地址支持一个呼叫
- 两个线路，每个线路上有一个地址，每个地址支持一个呼叫

使用这些 OID 可以查询微端口驱动程序的特性，每个从 OID_TAPI_PROVIDER_INITIALIZE 请求返回的设备都被查询一次。

8.6.8 打开线路

作为应用的 **lineOpen** 请求的结果，线路被打开。这个请求通过 TAPI 到 NDISTAPI 到 NDISWAN，再通过 NDIS 以 OID_TAPI_OPEN 请求的形式传递给标准广域网微端口驱动程序。此请求传递 NDIS_TAPI_OPEN 结构，成员 **ulDeviceID** 指定要被打开的线路的标识。另

外，此结构包含 **htLine**，NDISTAPI 内部涉及的线路句柄。微端口驱动程序必须保留此线路句柄并在对此线路的所有状态指示中和对查询的响应时返回，例如在用 **NdisMIndicateStatus** 报告状态改变时。

标准广域网微端口驱动程序创建一个结构来代表线路，此结构保存线路相关的环境。在从 MiniportQueryInformation 返回之前，微端口驱动程序应当在 OID_TAPI_OPEN 的成员 **hdLine** 中返回代表此环境的值。**hdLine** 中的线路相关值将在后续的有关此线路的调用中（如 OID_TAPI_MAKE_CALL）传递给微端口驱动程序。

设置线路的模式

如果应用程序用所有者特权发出 **lineOpen** 请求，应用表示它想接收内入呼叫。应用必须指定在此线路上它愿意接受的内入呼叫的有效类型。例如，一个应用可以请求接收这样的类型：g3fax、交互语音、datamodem 等。可以使用 OID_SET_DEFAULT_MEDIA_DETECTION 请求多次调用 MiniportSetInformation，以设置微端口驱动程序代表客户可以接受的类型。

如果多于一个的应用设置了介质类型，并且一个应用的类型列表与另一个的列表不同，两个应用以这些请求调用微端口驱动程序以设置介质检测类型。如果一个内入呼叫不在以前成功完成的 SET_DEFAULT_MEDIA_DETECTION 请求指定的介质模式类型中，微端口驱动程序将不会把此呼叫指示给 TAPI。如果微端口驱动程序没有收到对某个线路的 SET_DEFAULT_MEDIA_DETECTION 请求，它将不会指示任何内入呼叫；这个线路只用于外出呼叫。

标准广域网微端口驱动程序接收的 OID_TAPI_CONDITIONAL_MEDIA_DETECTION 请求用于判断微端口驱动程序是否支持对特定介质模式类型的内入呼叫指示。如果微端口驱动程序能够监控指示的介质模式集，能够支持 **LineCallParams** 中指出的性能，能够放置指定类型的呼叫，它将返回 NDIS_STATUS_SUCCESS。如果这样，微端口驱动程序很有可能会接收到后续请求如 OID_TAPI_OPEN，OID_SET_DEFAULT_MEDIA_DETECTION 等等。

8.6.9 接受内入呼叫

在标准广域网微端口驱动程序成功完成 OID_TAPI_PROVIDER_INITIALIZE、OID_TAPI_OPEN 和 OID_SET_DEFAULT_MEDIA_DETECTION 请求之后，它就可以在已经打开的线路上接收内入呼叫了。如果内入呼叫的介质类型与某个已设置的类型匹配，微端口驱动程序应调用 **NdisMIndicateStatus** 通知上层此呼叫，它在 *GeneralStatus* 中传递 NDIS_TAPI_INDICATION，在 *StatusBuffer* 中传递指向 NDIS_TAPI_EVENT 的指针。

NDIS_TAPI_EVENT 的 **ulMsg** 被置为 LINE_NEWCALL，**ulParam1** 被置为 **hdCall**——驱动程序的呼叫句柄。**htCall**、**ulParam2**、 **ulParam3** 被置为零。在从 **NdisMIndicateStatus** 返回时，NDISWAN 在 **ulParam2** 中放 **htCall**。微端口驱动程序应保留此句柄并在后续的属于此呼叫实例的指示中使用。例如，在向 NDISWAN 指示 line-up 时微端口驱动程序将 **ConnectionWrapperID** 置为此值。

微端口驱动程序给出 LINE_NEWCALL 状态指示之后，它应用 LINECALLSTATE_*XXX* 消息，一般是 LINECALLSTATE_OFFERING 消息，调用 **NdisMIndicateStatus**。此消息的 **ulParam3** 置为内入呼叫的介质模式。

如果呼叫被应用接受，应以 OID_TAPI_ACCEPT 调用 *MiniportSetInformation*，然后用一个 OID_TAPI_ANSWER 请求调用 *MiniportSetInformation*，或只用 OID_TAPI_ANSWER 请求调用 *MiniportSetInformation*。

如果微端口驱动程序接收到 OID_TAPI_ACCEPT 请求，它应为此呼叫分配并初始化一个呼叫状态环境，但在接收到 OID_TAPI_ANSWER 请求之前不能应答此呼叫。然后，如果微端口驱动程序接收到 OID_TAPI_ANSWER 请求，它应做如下操作：

- 应答此呼叫
- 将线路的状态置为已连接，若允许报告此状态则调用 **NdisMIndicateStatus** 报告此新状态。

如果微端口驱动程序在未收到 OID_TAPI_ACCEPT 请求时收到 OID_TAPI_ANSWER 请求，它应认为这两个请求被同时收到。

有可能广域网 NIC 驱动程序永远收不到对某一线路的 OID_TAPI_ANSWER 请求或 OID_TAPI_ACCEPT 请求。这种情况发生在客户忙于处理其他呼叫时，或客户对指示的呼叫类型不感兴趣（如对介质模式），等等。等待某一接受或应答请求一定时间后，或检测到远程部分断连，广域网 NIC 驱动程序应通知 NDISTAPI 驱动程序呼叫被转换为空闲状态。然后，一段时间后，一个 OID_TAPI_CLOSE_CALL 请求应被发送给广域网 NIC 驱动程序来释放呼叫实例。微端口驱动程序也有可能在新呼叫超时之前收到 OID_TAPI_CLOSE_CALL。

8.6.10 产生 TAPI 呼叫

拥有一个开放线路的标准广域网微端口驱动程序可以接收以下两个请求之一以发起呼叫：

- OID_TAPI_MAKE_CALL 请求，包含目标地址或使用预定义的缺省号码。
- OID_TAPI_MAKE_CALL 请求，不包含目标地址。这种情况下 OID_TAPI_MAKE_CALL 后紧跟一个包含目标地址的 OID_TAPI_DIAL 请求

当以 OID_TAPI_MAKE_CALL 请求调用 *MiniportQueryInformation* 时，NDISTAPI 在 NDIS_TAPI_MAKE_CALL 结构的成员 **hdLine** 中传递线路的标识，标识要产生呼叫的线路。**HdLine** 的值是微端口驱动程序从 OID_TAPI_OPEN 请求返回的值。

微端口驱动程序应保存通过请求传递的 **htCall** 值，此值在该呼叫实例后续的状态指示和响应中将被传递。例如，在向 NDISWAN 指示 line-up 时微端口驱动程序将此值作为 **ConnectionWrapperID**。

对于非拨号的地址，**ulDestAddressSize** 可以为零，例如热线电话总是自动连接到预定义的号码上；当用后续的 ID_TAPI_DIAL 请求拨号时，**ulDestAddressSize** 也可以为零。

若 NDIS_TAPI_MAKE_CALL 结构的成员 **bUseDefaultLineCallParams** 为 FALSE，微端口驱动程序应使用 **LineCallParams** 中的呼叫参数。若 **bUseDefaultLineCallParams** 为 TRUE，则使用预先设置的缺省呼叫参数。

微端口驱动程序必须检查在其上进行建立呼叫请求的线路的状态，以确定它是否可以生成请求的呼叫。若不能，微端口驱动程序应使此呼叫失败。

MiniportQueryInformation 在以下情况下返回：

- 它已产生了呼叫
- 否则，若在后续的 OID_TAPI_DIAL 请求中拨号，微端口驱动程序分配一个结构并将它自己设置为一个状态，可能是 dialtone 状态，以便以后完成此呼叫。

若 MiniportQueryInformation 返回 NDIS_STATUS_SUCCESS，微端口驱动程序必须在 **hdCall** 中返回一个呼叫相关的环境值。此唯一的 **hdCall** 值将在后续与此呼叫相关的请求中传递给微端口驱动程序。

若呼叫是通过两步完成的，第一步传递给 MiniportQueryInformation OID_TAPI_MAKE_CALL 请求；第二步传递给 *MiniportSetInformation* OID_TAPI_DIAL 请求，目标号码在 NDIS_TAPI_DIAL 结构中传递给微端口驱动程序。OID_TAPI_DIAL 请求需要在 NDIS_TAPI_DIAL 结构中设置 **hdCall**，它的值与微端口驱动程序从先前的 OID_TAPI_MAKE_CALL 请求返回的句柄相同。

8.6.11 主动事件处理

在电话会话过程中，标准广域网微端口驱动程序很可能会需要向 NDISTAPI 通知驱动程序主动事件。最典型的主动事件是在一个已打开线路上的进入呼叫，现有呼叫的状态转变（例如从振铃到连接），或线路设备状态的改变（从运行状态到运行结束，反之亦然）。

在向 NDISWAN 驱动程序自发地报告线路或呼叫上发生的事件时，使用 NDIS_STATUS_TAPI_INDICATION 状态。微端口驱动程序调用 **NdisMIndicateStatus**，指定 *GeneralStatus* 为 NDIS_STATUS_TAPI_INDICATION，*StatusBuffer* 参数指向一个已初始化的 NDIS_TAPI_WAN 结构。

此结构包含以下成员

结构成员	意义
htLine	事件发生的线路的 NDISTAPI 驱动程序线路句柄。
htCall	对于没有呼叫的线路相关事件，此参数应为零；否则，是与报告的事件相关的 NDISTAPI 驱动程序呼叫句柄。
ulMsg	指定事件类型，格式为 LINE_*XXX*。
ulParam1	与事件有关的参数，例如，指向描述特定事件的结构的指针。
ulParam2	同上
ulParam3	同上

与标准广域网微端口驱动程序开发者有关的通知消息有：

LINE_ADDRESSSTATE
LINE_CALLINFO
LINE_CALLSTATE
LINE_CLOSE
LINE_DEVSPECIFIC
LINE_NEWCALL
LINE_CALLDEVSPECIFIC

8.6.12 Line-Up 指示

从电话的观点看，呼叫的生命期就是一系列状态转变。对于外出呼叫，典型的状态序列是：拨号，进行中，振铃，连接，断连，最后是空闲。对于内入呼叫，这个序列是：提交，振铃，连接，断连，最后是空闲。无论在哪种情况下，对向连接状态的转化都特别感兴趣，因为连接状态是数据可以流过线路的状态。

标准广域网微端口驱动程序以参数 *GengealStatus* 为 NDIS_STATUS_WAN_LINE_UP 调用 **NdisMIndicateStatus** 函数，指示一个新的数据信道已变为活动的。*StatusBuffer* 指向的缓冲是 NDIS_MAC_LINE_UP 结构。在指示 line-up 之前，适配器可以接受帧，并能使他们成功或失败，但这些帧未必会被远程系统接收。在 line-up 指示前，协议应当减少计时器并重新计数以便任何外出的连接企图迅速失败。

NDIS_STATUS_WAN_LINE_UP 状态必须在从 OID_TAPI_GET_ID 请求返回之前完成。从 NDIS_STATUS_WAN_LINE_UP 指示返回的 **NdisLinkContext** 在 NDIS_TAPI_GET_ID 结构中作为 **DeviceID** 传递回去。

NDIS_STATUS_WAN_LINE_UP 必须在向 NDISWAN 指示 LINECALLSTATE_CONNECTED 之前完成。否则，连接的用户模式客户可能会在 NDISWAN 意识到此线路存在之前试图发送数据。

Lint-up 指示完成后，NDISWAN 和微端口驱动程序已经交换了句柄，其中一个句柄由 NDISWAN 用于以后直接向数据信道发送包，另一个由微端口驱动程序用于为到 NDISWAN 的发送和状态指示标识正确的数据信道。在 NDISWAN 得到 line-up 指示和为特定呼叫实例设立数据信道之前，NDISWAN 不会用微端口驱动程序的 *MiniportWanSend* 函数发送包。

如果下列某种情况发生，微端口驱动程序在 OID_TAPI_GET_ID 请求的环境之中指示 line-up 状态：

- NDIS_TAPI_GET_ID 结构的成员 **ulSelect** 被置为 LINECALLSELECT_CALL。并且
- 此请求是 NDIS_TAPI_GET_ID 结构成员 **hdCall** 的第一个 OID_TAPI_GET_ID 请求。

或

- NDIS_TAPI_GET_ID 结构的成员 **ulSelect** 被置为 LINECALLSELECT_CALL。并且
- 此请求不是 NDIS_TAPI_GET_ID 结构成员 **hdCall** 的第一个 OID_TAPI_GET_ID 请求，但 **DeviceClass** 不同于先前任何对此呼叫实例的 get-id 请求。并且
- 微端口驱动程序支持对不同设备类型有各自的数据信道。

传递给 **NdisMIndicateStatus** 的 *StatusBuffer* 参数应该指向 NDIS_MAC_LINE_UP 结构的缓冲区。此结构定义是：

```
typedef struct    _NDIS_MAC_LINE_UP{
        IN ULONG                        LinkSpeed;
        IN NDIS_WAN_QUALITY             Quality;
        IN USHORT                       SendWindow;
        IN NDIS_HANDLE                  ConnectionWrapperID;
        IN NDIS_HANDLE                   NdisLinkHandle;
        IN OUT NDIS_HANDLE              NdisLinkContext;
} NDIS_MAC_LINE_UP, *PNDIS_MAC_LINE_UP;
```

标准广域网微端口驱动程序应当将 NDIS_MAC_LINE_UP 的 **ConnectionWrapperID** 的值置为此呼叫的 **htCall**（TAPI 呼叫句柄），因为它对所有兼容 TAPI 的微端口驱动程序是唯一的。在微端口驱动程序发布 line-down 指示之前 **ConnectionWrapperID** 都是有效的。

NdisLinkHandle 表示在此呼叫上的数据信道的微端口驱动程序句柄。一般来说，它引用微端口驱动程序为此呼叫保存的状态。NDISWAN 保留此句柄，并且如果要在此数据信道发送，NIDSWAN 将它返回给微端口驱动程序。在微端口驱动程序调用 **NdisMIndicateStatus** 时，**NdisLinkContext** 被置为 NULL。NDISWAN 在从 **NdisMIndicateStatus** 函数返回前将 **NdisLinkContext** 置为 NDISWAN 提供的值。微端口驱动程序必须保留此不透明的句柄，并在随后的有关此链接的 NDISWAN 呼叫中提供此句柄，如 **NdisMWanIndicateReceive**。

当 MiniportQuerytInformation 从 OID_TAPI_GET_ID 请求返回时，必须在 NDIS_TAPI_GET_ID 的成员 **DeviceID** 中返回 **NdisLinkContext**。从 NDISTAPI 来看，此环境用一个句柄将呼叫连接和设备类型绑定起来，该句柄是 NDISWAN 可以识别并已绑定到特定数据信道的。NDISWAN 用此句柄确保 TAPI 客户的数据包直接通过 NDISWAN 发送到正确的微端口驱动程序数据信道。

改变发送窗口

在 line-up 指示中指定的 **SendWindow**（NDIS_MAC_LINE_UP 结构的成员）充当节流阀的作用，在开始对包排队之前对于特定线路，它控制 NDISWAN 可以向 MiniportWanSend 函数传递的包的数量。OID_WAN_GET_INFO 请求返回的 NDIS_WAN_INFO 结构的 **MaxTransmit** 成员表示适配器支持的所有线路的发送窗口的缺省值。当微端口驱动程序为一个新线路作第一次 line-up 指示时，它可以为此线路重置发送窗口值。它也可以通过在后续 line-up 指示中给 NDISWAN 一个新的非零 **SendWindow** 动态的为某个线路改变发送窗口值。也就是说，微端口驱动程序在没有内入或外出呼叫到达时也可以产生 line-up 指示。如果微端口驱动程序将 **SendWindows** 置为零，NDISWAN 将使用缺省的 **MaxTransmit** 值作为发送窗口的大小。

如果微端口驱动程序发出 line-up 指示改变 **SendWindow**，它必须填写 **NdisLinkHandle** 和 **NdisLinkContext** 成员。**LinkSpeed** 和 **Quality** 被置为零或者是一个新的值。通过作 line-up 指示并重置这些值，**LinkSpeed** 和 **Quality** 都可以被动态的独立的改变。

8.6.13 关闭呼叫线路

在标准广域网微端口驱动程序成功完成 OID_TAPI_PROVIDER_INITIALIZE 请求时一次会话开始。在一次会话中，可以通过 OID_TAPI_OPEN 请求打开一个或多个线路，并且在一个线路上，可以通过 OID_TAPI_MAKE_CALL、OID_TAPI_DIAL、OID_TAPI_ANSWER 请求建立一个或多个呼叫。在一个线路启动期间，许多呼叫可以被建立，然后关闭或放弃。在一个会话期间，一个或多个线路可以经历多次从开放到关闭的转变。微端口驱动程序如何处理这些转变描述如下：

关闭一个呼叫

一个正在运行的呼叫可以被本地或远程节点关闭。呼叫在本地节点被关闭，是因为拥有此呼叫的句柄的最后一个应用关闭了这个句柄，或因为微端口驱动程序的 *MiniportHalt* 或 *MiniportRest* 函数被调用。如果远程节点中止了一个正在运行的呼叫，微端口驱动程序必须向上层报告此状态改变。

如果本地节点的应用程序要关闭呼叫，应以 OID_TAPI_DROP 请求并随后以 OID_TAPI_CLOSE_CALL 调用 *MiniportSetInformation*，或仅仅发送一个 OID_TAPI_CLOSE_CALL 请求。

如果微端口驱动程序接收到 OID_TAPI_DROP 请求，它必须作以下操作：

1. 丢弃呼叫
2. 将呼叫的状态变为 LINECALLSTATE_IDLE 并调用 **NdisMIndicateStatus** 报告状态的改变。

已丢弃呼叫的句柄——**htCall** 和 **hdCall** 保持有效。微端口驱动程序不能释放、重新使用或者换句话说更改与 **hdCall**（为呼叫实例创建的）有关的状态，因为微端口驱动程序可能接收到与此呼叫有关的请求，例如有关此呼叫的统计数据的请求。如果 MiniportHalt 或 MiniportRest 函数被调用，微端口驱动程序应当将此呼叫断连，但为此呼叫实例保留状态直到接收到 OID_TAPI_CLOSE_CALL 请求。

当拥有此呼叫实例的打开句柄的最后一个应用关闭此句柄时，MiniportSetInformation 也可以接收 OID_TAPI_CLOSE_CALL 请求。微端口驱动程序不能使此关闭呼叫请求失败。在此请求从 MiniportSetInformation 返回后，**htCall** 将不再有效。

微端口驱动程序必须准备处理类似这样的时序：在接收到 OID_TAPI_DROP 之前接收 OID_TAPI_CLOSE_CALL。如果微端口驱动程序接收到一个没有 OID_TAPI_DROP 的 OID_TAPI_CLOSE_CALL 请求，它就像接收到了丢弃和关闭呼叫两个请求那样进行处理。如果微端口驱动程序在关闭呼叫请求后收到丢弃请求，因为呼叫的句柄不再有效，微端口驱

动程序应当返回适当的错误信息。

如果微端口驱动程序检测到远程系统已经丢弃了呼叫，它应将呼叫的状态变为 LINECALLSTATE_DISCONNECTED 并调用 **NdisMIndicateStatus**（传递此呼叫实例的 **htCall**）指示此状态改变。在此状态改变信息被传播到本地节点上的应用程序后，微端口驱动程序将接收到 OID_TAPI_DROP 随后是 OID_TAPI_CLOSE_CALL，或者仅仅接收到 OID_TAPI_CLOSE_CALL 请求。微端口驱动程序应以上面所讲的方法处理丢弃和关闭呼叫请求。

关闭线路

在拥有此线路的开放句柄的最后一个应用关闭了这个句柄后，线路将被关闭。当微端口驱动程序接收到 OID_TAPI_CLOSE 时，它应当拆除线路实例，也就是说，微端口驱动程序中所有涉及此线路的状态可以被重用或释放，因为这些信息不再有效。

如果传递给一个 OID_TAPI_CLOSE 调用的线路句柄 **htLine** 存在并依然有效，此调用必须成功。如果还有呼叫在线路上，微端口驱动程序不会接收到对此线路的 OID_TAPI_CLOSE 请求。在做出线路关闭请求前，应使用 OID_TAPI_CLOSE_CALL 关闭所有呼叫。

当线路被关闭时所有存在的呼叫都必须被关闭；微端口驱动程序应当丢弃这些呼叫并清除它们的状态。微端口驱动程序还应当将线路的状态改变为空闲状态。微端口驱动程序拥有的所有此线路的句柄都不再有效。如果微端口驱动程序在收到关闭请求后又收到传递 **htLine** 句柄或 **htCall** 句柄的请求，它必须使这些请求失败。

关闭会话

上层或广域网微端口 NIC 驱动程序都可以发起会话的终止操作。当最后一个客户进程与高层电话模块分离时，NDISTAPI 驱动程序将被通知它需要终止每一个已注册适配器的会话。NDISTAPI 通过用 OID_TAPI_PROVIDER_SHUTDOWN 调用 **NdisRequest** 完成上述行为。一接收到此请求，NDISWAN 就调用微端口驱动程序的 MiniportSetInformation 函数，这将结束指定适配器上所有进行中的活动并释放所有相关资源。

驱动程序发起的会话终止可以在微端口 NIC 驱动程序被它的函数 MiniportHalt 卸载时发生。一般来说，微端口 NIC 驱动程序应完成 NDISTAPI 驱动程序的所有未完成请求并通过调用 **NdisMIndicateStatus** 通知 NDISWAN 线路设备正在关闭。然后 NDIWAN 应调用 NDISTAPI 驱动程序注销适配器。若微端口 NIC 驱动程序在后来又被重新加载，它应经历与先前描述相同的初始化过程。

若标准广域网微端口驱动程序经历了一些迫使所有客户和设备完成重新初始化的动态重配置，它也有可能发起会话的终止。例如，如果一个适配器的线路设备模型（如支持的线路设备数量）被改变，微端口驱动程序应设置 LINEDEVICESTATE_REINIT 标记并用 LINE_LINEDEVICESTATE 调用 **NdiaMIndicateStatus**。

8.6.14 NDISTAPI 接口

本节将描述定义 NDISTAPI 驱动程序、NDISWAN 和之下的标准广域网 NIC 微端口驱动程序之间接口的 OID 请求。此处描述的每个 OID 都在 Platform SDK 中的 Telephony Software Development Kit 中定义了一个相应的 SPI 功能。

标准广域网微端口驱动程序开发者在设计和实现驱动程序时应参考 SPI 规范。注意 NDISTAPI 规范和 SPI 规范的函数定义和参数可能有一些不同，在这种情况下开发者应遵守 NDISTAPI 驱动程序接口规范。

下表总结了定义 NDISTAPI 接口的 OID 请求。

OID	可选或必需
OID_TAPI_ACCEPT	可选

OID_TAPI_ANSWER	必需
OID_TAPI_CLOSE	必需
OID_TAPI_CLOSE_CALL	必需
OID_TAPI_CONDITIONAL_MEDIA_DETECTION	必需
OID_TAPI_CONFIG_DIALOG	可选
OID_TAPI_DEV_SPECIFIC	可选
OID_TAPI_DIAL	可选
OID_TAPI_DROP	必需
OID_TAPI_GET_ADDRESS_CAPS	必需
OID_TAPI_GET_ADDRESS_ID	必需
OID_TAPI_GET_ADDRESS_STATUS	必需
OID_TAPI_GET_CALL_ADDRESS_ID	必需
OID_TAPI_GET_CALL_INFO	必需
OID_TAPI_GET_CALL_STATUS	必需
OID_TAPI_GET_DEV_CAPS	必需
OID_TAPI_GET_DEV_CONFIG	可选
OID_TAPI_GET_EXTENSION_ID	可选
OID_TAPI_GET_ID	必需
OID_TAPI_GET_LINE_DEV_STATUS	必需
OID_TAPI_MAKE_CALL	必需
OID_TAPI_NEGOTIATE_EXT_VERSION	可选
OID_TAPI_OPEN	必需
OID_TAPI_PROVIDER_INITIALIZE	必需
OID_TAPI_PROVIDER_SHUTDOWN	必需
OID_TAPI_SECURE_CALL	可选
OID_TAPI_SELECT_EXT_VERSION	可选
OID_TAPI_SEND_USER_USER_INFO	可选
OID_TAPI_SET_APP_SPECIFIC	必需
OID_TAPI_SET_CALL_PARAMS	必需
OID_TAPI_SET_DEFAULT_MEDIA_DETECTION	必需
OID_TAPI_SET_DEV_CONFIG	可选
OID_TAPI_SET_MEDIA_MODE	必需
OID_TAPI_SET_STATUS_MESSAGES	必需

这些 OID 的定义在 *ntddndis*.h 中。NDIS_TAPI_EVENT 消息的定义在 *ndistapi*.h 中。

一个给定的标准广域网微端口驱动程序支持此接口的范围，主要依赖于开发者的判断和下层硬件的性能。一些请求的功能要求非常简单，例如 OID_TAPI_SET_APP_SPECIFIC 请求只要求驱动程序联合一个 ULONG 值到特定呼叫实例，此值可以被后来的 OID_TAPI_GET_CALL_INFO 请求检索到。

NIC 驱动程序开发者决定不支持的请求应当以 NDIS_STATUS_TAPI_OPERATIONUNAVIL 状态码结束。NIC 驱动程序可以通过对存在于某些 OID_TAPI_GET_*XXX* 请求中的结构的成员（例如，LINE_ADDRESS_CAPS 结构的成员 **ulAddrCapFlags**）置位或清零，以指示它支持的请求类型（依赖于线路、地址、或呼叫状态，在不同时间有所变化）。

以下节将描述 NDISTAPI 接口上的查询和设置请求：

- 8.6.14.1 查询信息请求
- 8.6.14.2 设置信息请求

8.6.14.1 查询信息请求

查询信息类型的请求通过在结构定义中加 OUT 前缀，表示它与设置信息类型的请求在请求结构的一个或多个成员上的差异，OUT 前缀表示要传递回调用进程的信息。有 IN 前缀的成员应被认为是只读的。所有查询信息请求的第一个成员，**ulRequestID**，为将来的使用保留，可以被忽略。

以下结构的定义可以在 *ndistapi*.h 中找到：

OID_TAPI_DEV_SPECIFIC

此请求用于产生扩展机制，使广域网 NIC 驱动程序提供对其他请求未描述特征的访问。扩展的意义与设备有关，应用程序要完全知道这些扩展才能利用它们。此函数使用 NDIS_TAPI_DEV_SPECIFIC 结构。

OID_TAPI_GET_ADDRESS_CAPS

此请求查询指定线路设备上的指定地址，以测定它的电话性能。此请求使用 NDIS_TAPI_GET_ADDRESS_CAPS 结构。

OID_TAPI_GET_ADDRESS_ID

此请求返回指定线路设备上与地址相关的地址 ID（因线路不同而有不同的格式）。此请求使用 NDIS_TAPI_GET_ADDRESS_ID 结构。

OID_TAPI_GET_ADDRESS_STATUS

此请求查询指定地址，以得到它的当前状态。使用 NDI _TAPI_GET_ADDRESS_STATUS 结构。

OID_TAPI_GET_CALL_ADDRESS_ID

此请求为指定呼叫检索地址 ID，使用 NDIS_TAPI_GET_CALL_ADDRESS_ID 结构。

OID_TAPI_GET_CALL_INFO

此请求返回指定呼叫的详细信息，使用 NDIS_TAPI_GET_CALL_INFO 结构。

OID_TAPI_GET_CALL_STATUS

此请求返回指定呼叫的当前状态，使用 NDIS_TAPI_GET_CALL_STATUS 结构。

OID_TAPI_GET_DEV_CAPS

此请求检索指定的线路以测定它的电话性能。返回信息对此线路设备上所有的地址都有效。使用 NDIS_TAPI_GET_DEV_CAPS 结构。

OID_TAPI_GET_DEV_CONFIG

此请求返回一个数据结构对象，它的内容与线路（广域网微端口 NIC 驱动程序）和设备类型有关，给出与线路设备一对一联系的设备的当前配置。此请求使用 NDIS_TAPI_GET_DEV_CONFIG 结构

OID_TAPI_GET_EXTENSION_ID

此请求为指定线路设备返回广域网微端口 NIC 驱动程序支持的扩展 ID。此请求使用 NDIS_TAPI_GET_DEV_CONFIG 结构。

OID_TAPI_GET_ID

此请求为联合选定线路、地址、或呼叫的指定设备类型返回设备 ID。使用 NDIS_TAPI_GET_ID 结构。

OID_TAPI_GET_LINE_DEV_STATUS

此请求查询指定的已打开线路设备的当前状态。返回的信息对此线路上所有地址是公用的。此请求使用 NDIS_TAPI_GET_LINE_DEV_STATUS 结构。

OID_TAPI_MAKE_CALL

此请求在指定线路上向指定目标地址发出呼叫。如果未要求使用缺省呼叫建立参数，可以指定呼叫参数。此请求使用 NDIS_TAPI_MAKE_CALL 结构。

OID_TAPI_NEGOTIATE_EXT_VERSION

此请求返回微端口设备希望在其上运行的最高版本扩展号，给出可能的扩展版本范围。此请求使用 NDIS_TAPI_NEGOTIATE_EXT_VERSION 结构。

OID_TAPI_OPEN

此请求打开指定的线路设备（给定设备 ID），返回此设备的 NIC 驱动程序句柄。NIC 驱动程序必需保留此设备的 NDISTAPI 驱动程序句柄。此请求使用 NDIS_TAPI_OPEN 结构。

OID_TAPI_PROVIDER_INITIALIZE

此请求初始化广域网微端口驱动程序，以便处理 TAPI 操作。此请求使用 NDIS_TAPI_PROVIDER_INITIALIZE 结构。

8.6.14.2 设置信息请求

除完成状态外，设置信息请求不返回任何信息。请求结构的所有成员应被认为是只读的，不能被标准广域网微端口驱动程序修改。所有查询信息请求的第一个成员，**ulRequestID**，保留在将来使用，可以被忽略。

以下结构的定义可以在 *ndistapi*.h 中找到：

OID_TAPI_ACCEPT

此请求接受指定的呼叫。它可以向呼叫方发送特定的用户到用户信息。此请求使用 NDIS_TAPI_ACCEPT 结构。

OID_TAPI_ANSWER

此请求应答指定的呼叫。它可以向呼叫方发送特定的用户到用户信息。此请求使用 NDIS_TAPI_ANSWER 结构。

OID_TAPI_CLOSE

此请求在完成或中止设备上所有未完成呼叫和异步请求后关闭指定的线路设备。此请求使用 NDIS_TAPI_CLOSE 结构。

OID_TAPI_CLOSE_CALL

此请求在完成或中止呼叫上所有未完成异步请求后释放指定的呼叫。此请求使用 NDIS_TAPI_CLOSE_CALL 结构

OID_TAPI_CONDITIONAL_MEDIA_DETECTION

当客户应用程序使用 LINEMAPPER 作为 **dwDeviceID** 调用 **lineOpen** 函数请求驱动程序搜索线路，以找出一个支持期望的介质模式和呼叫参数的线路时， NDISTAPI 发出此请求。

NDISTAPI 驱动程序的搜索基于期望的介质模式和此线路上其他正被监视的介质模式的并集，如果微端口驱动程序不能同时监视所有请求的介质模式，这就给微端口驱动程序指示的机会。在那时，如果微端口驱动程序能监视指定的介质模式集，支持 **LineCallParams** 中指定的性能，支持指定类型的呼叫，它将以成功指示回复。微端口驱动程序保持线路的活动介质监视模式不变。此请求使用 NDIS_TAPI_CONDITIONAL_MEDIA_DETECTION 结构。

OID_TAPI_DIAL

此请求在指定呼叫上拨指定号码。此请求使用 NDIS_TAPI_DIAL 结构。

OID_TAPI_DROP

此请求丢弃或断连指定的呼叫。可以选择用户到用户信息作为呼叫断连的一部分传送。应用可以在任何时间发起此请求。成功返回 OID_TAPI_DROP 后，此呼叫变为空闲状态。此请求使用 NDIS_TAPI_DIAL 结构。

OID_TAPI_PROVIDER_SHUTDOWN

此请求关闭微端口驱动程序。微端口驱动程序应终止所有正在进行的活动，并为所有开放线路向 NDISWAN 作 line-down 指示。此请求使用 NDIS_TAPI_PROVIDER_SHUTDOWN 结构。

OID_TAPI_SECURE_CALL

此请求使呼叫免受所有可能影响呼叫的介质流的中断或冲突的影响。此请求使用 NDIS_TAPI_SECURE_CALL 结构。

OID_TAPI_SELECT_EXT_VERSION

此请求为指定线路设备选择指定的扩展版本。后续的请求操作应符合此扩展版本。此请求使用 NDIS_TAPI_SELECT_EXT_VERSION 结构。

OID_TAPI_SEND_USER_USER_INFO

此请求向指定呼叫的远程方发送用户到用户信息。此请求使用 NDIS_TAPI_SEND_USER_USER_INFO 结构。

OID_TAPI_SET_APP_SPECIFIC

此请求设置指定呼叫的 LINECALLINFO 结构的应用相关成员。此请求使用 NDIS_TAPI_SET_APP_SPECIFIC 结构。

OID_TAPI_SET_CALL_PARAMS

此请求为一个已存在的呼叫设置某些呼叫参数。此请求使用 NDIS_TAPI_SET_CALL_PARAMS 参数。

OID_TAPI_SET_DEFAULT_MEDIA_DETECTION

此请求通知微端口驱动程序新的介质模式集（替换原先的集合）以监测指定的线路。此请求使用 NDIS_TAPI_SET_DEFAULT_MEDIA_DETECTION 结构。

OID_TAPI_SET_DEV_CONFIG

此请求用先前用 OID_TAPI_GET_DEV_CONFIG 获得的数据恢复与线路设备一对一联系的设备的配置。此数据结构的内容与线路（广域网 NIC 驱动程序）和设备类型有关。此请求使用 NDIS_TAPI_SET_DEV_CONFIG 结构。

OID_TAPI_SET_MEDIA_MODE

此请求改变存储在呼叫的 LINE_CALL_INFO 结构中的呼叫的介质模式。此请求使用 NDIS_TAPI_SET_DEV_CONFIG 结构。

OID_TAPI_SET_STATUS_MESSAGE

此请求指定 NIC 驱动程序应为指定线路或它的任何地址的哪些状态改变事件产生通知信息。在缺省情况下，一个线路的地址和线路状态报告最初是停用的。此请求使用 NDIS_TAPI_SET_STATUS_MESSAGE 请求。

OID_TAPI_CONFIG_DIALOG

此请求检索用户模式动态连接库（用于配置指定设备）的名字。这个配置 DLL 应导出以下函数：

```
LONG
WINAPI
ConfigDialog(
    IN   HWND   hwndOwner,
    IN   ULONG  ulDeviceID,
    IN   LPCSTR lpszDeviceClass
    );
```

此请求使用 NDIS_TAPI_CONFIG_DIALOG 结构。

8.7 使用支持电话服务的 CoNDIS 扩展

本节描述了在面向连接的环境中，CoNDIS 广域网 NIC 微端口驱动程序如何使用 NDIS 函数实现电话服务。CoNDIS 广域网微端口驱动程序通过 NIDS 与 NDPROXY 和 NDISWAN 驱动程序通信。NDISPROXY 驱动程序通过电话服务提供者与电话应用程序通信。详见 Platform SDK 中的电话应用程序编程接口（TAPI）。

以下节更全面的描述了 NDISPROXY 驱动程序。以下节还描述了 CoNDIS 广域网微端口驱动程序如何注册和列举它的 TAPI 性能、建立线路、建立和关闭由 TAPI 请求发起的呼叫。这些描述中简单的提及了 TAPI 中的一些概念，读者应参阅 Platform SDK 以获得有关 TAPI 更详细的信息。TAPI 如何模拟线路设备以及广域网微端口驱动程序如何保持连接的状态在 8.6.2 和 8.6.4 节中已有详细的描述。

- 8.7.1 NDPROXY 概述
- 8.7.2 CoNDIS TAPI 注册
- 8.7.3 CoNDIS TAPI 初始化
- 8.7.4 建立外出呼叫
- 8.7.5 接受内入呼叫
- 8.7.6 CoNDIS TAPI 关闭
- 8.7.7 语音流对呼叫管理器的要求
- 8.7.8 在面向连接 NDIS 之上支持电话服务的非广域网专用的扩展

8.7.1 NDPROXY 概述

NDPROXY 驱动程序是 Windows Telephony 服务的提供者。应用程序发出 TAPI 请求，Windows Telephony 服务将此请求传递给 NDPROXY。NDPROXY 为 Windows Telephony 服务实现电话服务提供者接口（TSPI），NDPROXY 通过 NDIS 与 NIDSWAN 驱动程序和 CoNDIS 广域网微端口驱动程序通信。这些 CoNDIS 广域网微端口驱动程序必需提供 TAPI 能力。

NDPROXY 驱动程序通过将 TAPI 参数封装在 NDIS 结构中传递 TAPI 请求。NDPROXY 向 CoNDIS 广域网微端口驱动程序呈现一个客户接口，向 NDISWAN 呈现一个呼叫管理者接口。NDISWAN 向 NDPROXY 呈现一个客户接口。CoNDIS 广域网微端口驱动程序向 NDPROXY 呈现一个呼叫管理者接口，向 NDISWAN 呈现一个 CoNDIS 微端口接口。

一个有 TAPI 能力的 CoNDIS 广域网微端口驱动程序将自己注册并初始化为广域网和 TAPI 服务二者的用户。在注册和初始化完成之后，用户层的应用就可以向 Windows 电话服务模块发出电话请求，此模块将 TAPI 请求转化为 TSPI 请求。Windows Telephony 服务模块将这些请求传递给 NDPROXY 驱动程序。NDPROXY 将 TAPI 请求的参数封装到 NDIS 结构中，并将请求发送给适当的 CoNDIS 广域网微端口驱动程序。传送这些请求的目的是建立、监视、拆卸线路和呼叫。

一个有 TAPI 能力的 CoNDIS 广域网微端口驱动程序还可以通知线路和呼叫状态的改变，例如内入呼叫的到达或远程断连。

8.7.2 CoNDIS TAPI 注册

本节讨论 CoNDIS 广域网微端口驱动程序如何注册以及如何建立与 NDISWAN 和

NDPROXY 驱动程序的 TAPI 相关通信。

在 CoNDIS 广域网微端口驱动程序为一个或多个 NIC 注册微端口驱动程序入口后，以下操作将使 NDISWAN 和 NDPROXY 驱动程序与这些 NIC 以 TAPI 相关方式联合起来。

- CoNDIS 广域网微端口驱动程序在它的 *MiniportInitialize* 函数内部调用 **NdisMCmRegisterAddressFamily** 函数，注册它的呼叫管理器入口指针和地址族类型 CO_ADDRESS_FAMILY_TAPI_PROXY。通过这个操作，微端口驱动程序通知它实现 TAPI 服务。
- NDIS 调用 NDPROXY 的 *ProtocolCoAfRegisterNotify* 函数通知 NDPROXY 新注册的地址族。NDPROXY 的 ProtocolCoAfRegisterNotify 检查地址族数据并确定它可以使用集成到 CoNDIS 广域网微端口驱动程序内的呼叫管理者提供的 TAPI 服务。有 TAPI 能力的 CoNDIS 广域网微端口驱动程序是一个集成微端口呼叫管理器（MCM）。
- NDPROXY 调用 **NdisClOpenAddressFamily** 函数打开联合 CoNDIS 广域网微端口驱动程序的 TAPI-proxy 地址族。**NdisClOpenAddressFamily** 向 NDIS 注册 NDPROXY 的面向连接的入口指针。这些入口指针用于与 CoNDIS 广域网微端口驱动程序通信。
- NDPROXY 调用 **NdisCmRegisterAddressFamily** 函数注册它的呼叫管理器入口指针和地址族类型 CO_ADDRESS_FAMILY_TAPI。通过这样做，NDPROXY 通知它实现 TAPI 服务。
- NDIS 调用 NDISWAN 的 ProtocolCoAfRegisterNotify 函数通知 NDISWAN 新注册的地址族。NDISWAN 的 ProtocolCoAfRegisterNotify 函数检查地址族数据并确定 NDISWAN 可以使用 NDPROXY 提供的 TAPI 服务。
- NDISWAN 调用 **NdisClRegisterSap** 函数通知 NDPROXY，NDISWAN 可以在特定服务访问点（SAP）上接受内入呼叫。在此调用中，NDISWAN 传递一个描述 SAP 的 CO_SAP 结构。NDISWAN 将 CO_SAP 结构的成员 **SapType** 置为 AF_TAPI_SAP_TYPE，指定 SAP 将用于 TAPI 呼叫。为一个特定的 TAPI 设备，NDISWAN 将 CO_SAP 结构的成员 **Sap** 置为一个字符串。在 TAPI 应用程序调用 lineGetID 函数时，应用程序提供此串。NDPROXY 应向 NDISWAN 通知所有定位到 SAP 上的内入呼叫。

8.7.3 CoNDIS TAPI 初始化

本节讨论 CoNDIS 广域网微端口驱动程序如何为应用列举它的 TAPI 性能。这些 TAPI 性能包括：

- 微端口驱动程序支持的线路设备数。如 modem、fax 卡、以及 ISDN 卡等。
- 特定线路的信息。例如线路标识符和线路支持的信道地址（电话号码）等。
- 特定信道地址的信息。例如呼叫者的标识符（Caller ID）和活动呼叫的数目。

为了得到线路和信道地址的性能，NDPROXY 发出请求检索底层硬件的信息。也就是说，NDPROXY 查询 CoNDIS 广域网微端口驱动程序的 TAPI 性能。NDPROXY 驱动程序调用 **NdisCoRequest** 函数查询微端口驱动程序的 TAPI 性能。在此调用中，NDPROXY 传递一个 NDIS_REQUEST 结构。NDPROXY 在 NDIS_REQUEST 结构中指定以下信息：

- 在 **RequestType** 成员中保存值 **NdisRequestQueryInformation**。
- 在 **Oid** 成员保存对象标识符（OID），指定要从微端口驱动程序检索哪个 TAPI 性能。
- 返回 TAPI 性能的缓冲区保存在 **InformationBuffer** 成员中。

NDPROXY 发送给 CoNDIS 广域网微端口驱动程序的所有请求都可以被异步或同步的完成。如果 CoNDIS 广域网微端口驱动程序确定不能立即结束查询，它将仅返回一个 NDIS_STATUS_PENDING，并在完成请求后在它的 ProtocolCoRequest 函数内调用 **NdisMCmRequestComplete** 函数。

在 CoNDIS 广域网微端口驱动程序通知 NDPROXY 关于一个新地址族注册的消息后，NDPROXY 查询以下 OID 以确定 CoNDIS 广域网微端口驱动程序和微端口驱动程序的 NIC 的 TAPI 相关性能。

- NDPROXY 用 OID_CO_TAPI_CM_CAPS 查询微端口驱动程序，以确定微端口驱动程序的设备（提供 TAPI 服务的设备）支持的线路数。此 OID 还要求微端口驱动程序指出这些线路是否有不同的性能。
- NDPROXY 然后用 OID_CO_TAPI_LINE_CAPS 查询微端口驱动程序，以确定指定线路的电话性能。此 OID 还要求微端口驱动程序指出此线路上的地址是否有不同的地址性能。
  - 若先前的 OID_CO_TAPI_CM_CAPS 查询发现微端口驱动程序的设备只支持一个线路，或设备支持具有相同性能的多条线路，NDPROXY 只需要查询一次 OID_CO_TAPI_LINE_CAPS 以获得设备的线路性能。在此情况下，微端口驱动程序返回的线路性能适用于此设备上所有的线路。
  - 若设备支持有不同性能的多条线路，NDPROXY 必须为每条线路查询一次 OID_CO_TAPI_LINE_CAPS 以获得每条线路的线路性能。
- 最后，NDPROXY 用 OID_CO_TAPI_ADDRESS_CAPS 查询微端口驱动程序，以确定指定线路上指定地址的电话性能。
  - 若先前的 OID_CO_TAPI_LINE_CAPS 查询发现线路只支持一个地址，或此线路上的所有地址具有相同的地址性能，NDPROXY 只需要查询一次 OID_CO_TAPI_ADDRESS_CAPS 以获得此线路上所有地址的性能。
  - 若线路支持有不同性能的多个地址，NDPROXY 必须为此线路上的每个地址查询一次 OID_CO_TAPI_LINE_CAPS。

NDPROXY 驱动程序使用由 TAPI 列举 OID 获得的信息做以下工作：

- 为后续 TAPI 呼叫创建 TAPI 参数。
- 决定应接受还是拒绝后续的内入 TAPI 呼叫。
- 注册一个或多个 TAPI 服务访问点（SAP）。

8.7.4 建立外出呼叫

如果一个应用程序试图发起外出呼叫，它首先应打开一个线路。应用调用 TAPI **lineOpen** 函数打开线路。应用调用 TAPI **lineMakeCall** 函数并传递一个指向指定目标地址的指针，将一个电话呼叫放到先前打开的线路上。如不要求使用缺省的呼叫建立参数，应用还要传递一个指向 LINECALLPARAMS 结构的指针。如应用使用缺省的呼叫建立参数，这些参数由 **lineMakeCall** 在 LINECALLPARAMS 结构中返回。此结构的成员指定电话呼叫应如何建立。

这些 TAPI 函数调用导致 NDPROXY 作如下操作：首先通过 CoNDIS 广域网微端口驱动程序建立虚连接（VC），然后为了产生外出呼叫将 TAPI 参数封装在 NDIS 结构中。微端口驱动程序将使用这些 TAPI 参数建立外出呼叫。外出呼叫的建立过程如下：

- NDPROXY 调用 **NdisCoCreateVc** 指示微端口驱动程序建立 VC。在 NDPROXY 调用 **NdisCoCreateVc** 后，作为一个同步操作，NDIS 调用集成在微端口驱动程序中的呼叫管理器的 ProtocolCoCreateVc 函数。NDIS 向 ProtocolCoCreateVc 传递一个代表 VC 句柄。如果 **NdisCoCreateVc** 调用成功，NDIS 会返回此句柄。ProtocolCoCreateVc 执行所有必需的动态资源和结构的分配，微端口驱动程序的呼叫管理器需要这些结构以执行此 VC 上的后续操作。这些资源包括（并不只限于）内存缓冲区、数据结构、事件、以及其他类似资源。

- NDPROXY 在 CO_AF_TAPI_MAKE_CALL_PARAMETERS 结构中为外出呼叫指定 TAPI 参数。NDPROXY 用 TAPI **lineMakeCall** 函数传递来的信息填充此结构的成员：
  - **DestAddress** 中存放目标地址。
  - **ulLineID** 中存放线路标识符。
  - **LineCallParams** 中存放 LINECALLPARAMS 结构。
- NDPROXY 将 CO_AF_TAPI_MAKE_CALL_PARAMETERS 结构放到 CO_SPECIFIC_PARAMETERS 结构的成员 **Parameters** 中，并将 CO_SPECIFIC_PARAMETERS 结构的成员 **Length** 置为 CO_AF_TAPI_MAKE_CALL_PARAMETERS 结构的长度。
- NDPROXY 将 CO_SPECIFIC_PARAMETERS 结构放到 CO_MEDIA_PARAMETERS 结构的成员 **MediaSpecific** 中。
- NDPROXY 将一个指向 CO_MEDIA_PARAMETERS 结构的指针放到 CO_CALL_PARAMETERS 结构的成员 **MediaParameters** 中。
- 一旦 NDPROXY 封装完 TAPI 参数，它就调用 **NdisClMakeCall** 函数指示外出呼叫。在此函数调用中，NDPROXY 传递了一个指向填充了的 CO_CALL_PARAMETERS 结构的指针。NDIS 然后调用 CoNDIS 广域网微端口驱动程序的呼叫管理器的 *ProtocolCmMakeCall* 函数。微端口驱动程序只需要检查嵌入在 CO_CALL_PARAMETERS 结构中的 CO_AF_TAPI_MAKE_CALL_PARAMETERS 结构，其他呼叫参数都没有意义。如果微端口驱动程序随后为外出呼叫激活了 VC，它调用 NdisMCmActiveVc 函数并传递一个指向填充了的 CO_CALL_PARAMETERS 结构的指针。
- 微端口驱动程序与网络协商，为 VC 确定电话呼叫参数并为那些呼叫参数设置 NIC 之后，它调用 **NdisCmMakeCallComplete** 函数指出它已经准备好在 VC 上进行数据传输。在此调用中，微端口驱动程序必需传递此 VC 的句柄和对电话呼叫参数的修改。
- 微端口驱动程序必需修改 CO_CALL_PARAMETERS 结构的成员 **CallMgrParameters**，以指定传输包的服务质量（QoS），例如带宽。为了设置 **CallMgrParameters**，微端口驱动程序填充 CO_CALL_MANAGER_PARAMETERS 结构的成员并将 **CallMgrParameters** 指向此结构。例如，为了标识 VC 的传输和接收速度，微端口驱动程序必需 CO_CALL_MANAGER_PARAMETERS 结构的成员 **Transmit** 和 **Receive** 的 **PeakBandWidth** 成员。**Transmit** 和 **Receive** 的类型是 FLOWSPEC 结构。FLOWSPEC 结构的详细信息参见 Platform SDK。
- 如果微端口驱动程序修改了电话呼叫参数，它必须将 CO_CALL_PARAMETERS 结构的 **Flags** 置为 CALL_PARAMETERS_CHANGED。作为微端口驱动程序调用 **NdisCmMakeCallComplete** 函数的结果，NDIS 调用 NDPROXY 的 ProtocolClMakeCallComplete 函数完成由 **NdisClMakeCall** 发起的异步操作。
- 在微端口驱动程序成功完成外出呼叫后，NDPROXY 通知 TAPI 应用程序呼叫已连接。然后此 TAPI 应用程序调用 TAPI **lineGetID** 函数通知 NDPROXY 定位适当的 CoNDIS 客户。在此 **lineGetID** 调用中，TAPI 应用程序为特定 TAPI 设备（应用程序需要它的句柄）提供了一个字符串。NDPROXY 使用此字符串定位 CoNDIS 客户（已经为特定 TAPI 注册了 SAP）。如果这个 CoNDIS 客户是 NDISWAN，此字符串是 NDIS。如果 NDPROXY 定位到一个与 TAPI 应用程序传递的字符串匹配的 SAP，它就调用 **NdisMCmCreateVc** 建立一个连接边界点，在此边界点上它可以发出外出呼叫通知。NDIS 然后调用 NDISWAN 的 ProtocolCoCreateVc 函数并传递一个代表 VC 的句柄。
- 在 NDPROXY 建立连接边界点后，它调用 **NdisCmDispatchIncomingCall** 函数通知

NDISWAN 外出呼叫。在此调用中，NDPROXY 传递了一个包含外出呼叫参数的 CO_AF_MAKE_CALL_PARAMETER 结构。NDIS 然后调用 NDISWAN 的 ProtocolClIncomingCall 函数，在此调用中 NDISWAN 可以接受或拒绝请求的连接。如果 NDISWAN 改变了传递给它的呼叫参数，它必须将 CO_CALL_PARAMETERS 结构的 **Flags** 设置为 CALL_PARAMETERS_CHANGED。

- 在决定是否接受连接并可能改变呼叫参数后，NDISWAN 调用 **ProtocolClIncomingCallComplete** 函数。NDIS 然后调用微端口驱动程序的 ProtocolCmIncomingCallComplete 函数。微端口驱动程序调用 **NdisCmDispatchCallConnected** 还是 **NdisCmDispatchIncomingCloseCall** 函数，取决于 NDISWAN 是否接受了外出呼叫以及微端口驱动程序是否接受 NDISWAN 对呼叫参数的改变，。**NdisCmDispatchCallConnected** 通知 NDISWAN 可以在 NDISWAN 为外出呼叫建立的 VC 上开始数据传输。**NdisCmDispatchIncomingCloseCall** 通知 NDISWAN 和 NDISWAN 拆卸外出呼叫。
- 在 NDISWAN 接受外出呼叫之后，NDPROXY 调用 **NdisCoGetTapiCallId** 函数检索标识 VC 的 NDISWAN 环境的字符串。NDPROXY 将此字符串传递回 TAPI 应用程序。TAPI 应用程序用此 VC 环境字符串完成对 **lineGetID** 的调用。

8.7.5 接受内入呼叫

在应用可以接受内入呼叫之前，它必须要打开一个线路。应用调用 TAPI **lineOpen** 函数打开线路。对此 TAPI 函数的调用导致下层驱动程序为了准备接受内入呼叫将 TAPI 参数封装到 NDIS 结构中。CoNDIS 广域网微端口驱动程序接收到一个内入呼叫后，首先微端口驱动程序必须与 NDPROXY 驱动程序建立一个虚连接（VC），然后向 NDPROXY 通知内入呼叫。NDPROXY 然后通过 TAPI 通知应用程序。内入呼叫过程如下：

- NDPROXY 在 CO_AF_TAPI_SAP 结构中为内入连接指定 TAPI 参数。NDPROXY 用从 TAPI **lineOpen** 函数传递来的以下信息填充此结构：
  - **ulLineID** 中存放开放线路的标识。
  - **ulAddressID** 中存放内入连接的地址。
  - **ulMediaModes** 中存放内入连接的信息流的介质模式。
- NDPROXY 用 CO_AF_TAPI_SAP 结构覆盖 CO_SAP 结构的成员 **Sap**，并置 CO_SAP 结构的成员 **SapLength** 为 CO_AF_TAPI_SAP 结构长度。NDPROXY 还应将 CO_SAP 结构的 **SapType** 置为 AF_TAPI_SAP_TYPE。
- 在 NDPROXY 封装完 TAPI 参数后，NDPROXY 调用 **NdisClRegisterSap** 函数使它自己准备好接收内入呼叫。在此函数调用中，NDPROXY 传递一个指向 CO_SAP 结构的指针，此结构指定 NDPROXY 可以在其上接收内入呼叫的服务访问点（SAP）。NDIS 将此 CO_SAP 结构传递给 CoNDIS 广域网微端口驱动程序的呼叫管理器（MCM）的 ProtocolCmRegisterSap 函数。ProtocolCmRegisterSap 根据需要与网络控制设备或其他介质相关代理通信，来为 NDPROXY 向网络注册 SAP。在微端口驱动程序注册完 SAP 后，它就可以接受直接向此 SAP 提出的内入呼叫。
- 从网络来的信号消息通知 CoNDIS 广域网微端口驱动程序内入呼叫。微端口驱动程序为呼叫从这些信号消息中提取呼叫参数，包括内入呼叫寻址的 SAP。
- 在通知 NDPROXY 内入呼叫之前，微端口驱动程序调用 **NdisMCmCreateVc** 函数通知 NDPROXY 建立 VC。NDPROXY 分配并初始化 VC 需要的资源并存储 VC 的句柄。
- CoNDIS 广域网微端口驱动程序在 CO_AF_TAPI_INCOMING_CALL_PARAMETERS 结构中为内入呼叫设置 TAPI 参数。微端口驱动程序用从信号消息中提取的信息填充此结构的成员：

- **ulLine** 中存放线路标识符。
- **ulAddressID** 中存放内入呼叫的地址。
- **ulFlags** 中的 CO_TAPI_FLAG_INCOMING_CALL 位。**ulFlags** 的其他位被保留且必需被置为 0。
- **LineCallInfo** 中存放 LINECALLPARAMS 结构。LINECALLPARAMS 的成员为内入呼叫指定 TAPI 呼叫参数。

- 微端口驱动程序用 CO_AF_TAPI_INCOMING_CALL_PARAMETERS 结构覆盖 CO_SPECIFIC_PARAMETERS 结构的 **Parameters** 成员，并置 CO_SPECIFIC_PARAMETERS 结构 **Length** 成员为 CO_AF_TAPI_INCOMING_CALL_PARAMETERS 的长度。
- 微端口驱动程序将 CO_SPECIFIC_PARAMETERS 结构放到 CO_MEDIA_PARAMETERS 结构的成员 **MediaSpecific** 中。
- 微端口驱动程序将一个指向 CO_MEDIA_PARAMETERS 结构的指针放到 CO_CALL_PARAMETERS 结构的 **MediaParameters** 成员中。
- 微端口驱动程序还必须设置 CO_CALL_PARAMETERS 结构的 **CallMgrParameters** 成员以指定传送包的服务质量（QoS），如带宽。为了设置 **CallMgrParameters**，微端口驱动程序填充 CO_CALL_MANAGER_PARAMETERS 结构的成员并将 **CallMgrParameters** 指向此结构。例如，为了标识 VC 的传输和接收速度，微端口驱动程序必需 CO_CALL_MANAGER_PARAMETERS 结构的成员 **Transmit** 和 **Receive** 的 **PeakBandWidth** 成员。**Transmit** 和 **Receive** 的类型是 FLOWSPEC 结构。FLOWSPEC 结构的详细信息参见 Platform SDK。
- 在微端口驱动程序封装 TAPI 参数和填充 CO_CALL_MANAGER_PARAMETERS 的 **CallMgrParameters** 成员后，它调用 **NdisMCmDispatchIncomingCall** 函数向 NDPROXY 指示内入呼叫。在此调用中，微端口驱动程序传递以下参数：
  - 标识内入呼叫寻址的 SAP 的句柄。
  - 标识 VC 的句柄。
  - 指向 CO_CALL_PARAMETERS 结构的指针。
- NDPROXY 返回微端口驱动程序 NDIS_STATUS_PENDING，这样 NDPROXY 就可以异步的完成 **NdisMCmDispatchIncomingCall**。
- 在 TAPI 应用程序用 **lineAnswer** 函数应答内入呼叫后，NDPROXY 调用 **NdisClIncomingCallComplete** 函数。NDIS 然后调用微端口驱动程序的 ProtocolCmIncomingCallComplete 函数。NDPROXY 返回 NDIS_STATUS_SUCCESS 表示接受呼叫参数。如果 NDPROXY 发现呼叫参数不可接受，它可以通过将 CO_CALL_PARAMETERS 结构的 **Flags** 置为 CALL_PARAMETERS_CHANGED 并提供修改了的呼叫参数，请求改变呼叫参数。如果 NDPROXY 接受内入呼叫，微端口驱动程序应发送信号消息指示正在呼叫的实体呼叫被接受。否则，微端口驱动程序应发送信号消息指示呼叫被拒绝。如果 NDPROXY 请求改变呼叫参数，微端口驱动程序发送信号消息请求对呼叫参数的改变。
- 微端口驱动程序激活 VC 并调用 **NdisMCmActivateVc** 函数通知 NDPROXY 微端口驱动程序已准备好在 VC 上传输数据。
- 如果拒绝呼叫，微端口驱动程序调用 **NdisMCmDeactivateVc** 函数去活微端口驱动程序为内入呼叫创建的 VC。然后微端口驱动程序调用 **NdisMCmDeleteVc** 函数删除 VC。
- 根据 NDPROXY 是否接受内入呼叫和端到端连接是否被成功建立，微端口驱动程序调用 **NdisMCmDispatchCallConnected** 或 **NdisMCmDispatchIncomingCloseCall** 函数。注

意，如果远程正在呼叫的实体拆卸了呼叫，它将发送信号消息指示端到端连接没有被成功建立。**NdisMCmDispatchCallConnected** 通知 NDPROXY 可以在 VC 上开始数据传输。**NdisMCmDispatchIncomingCloseCall** 通知 NDPROXY 拆除内入呼叫。

- 如果 NDPROXY 被指示拆除内入呼叫，它调用 **NdisClCloseCall** 函数确认它不会在此 VC 上试图发送或接收数据。NDIS 然后调用微端口驱动程序的 ProtocolCmCloseCall 函数。微端口驱动程序然后调用 **NdisMCmDeactivateVc** 函数去活 VC。在 VC 被去活后，微端口驱动程序调用 **NdisMCmDeleteVc** 函数删除 VC。
- 在 TAPI 应用程序接受内入呼叫和 NDPROXY 通知应用程序呼叫已连接后，应用调用 TAPI **lineGetID** 函数通知 NDPROXY 定位适当的 CoNDIS 客户。在此 **lineGetID** 调用中，TAPI 应用程序为特定 TAPI 设备（应用程序需要它的句柄）提供了一个字符串。NDPROXY 使用此字符串定位 CoNDIS 客户（已经为特定 TAPI 注册了 SAP）。如果这个 CoNDIS 客户是 NDISWAN，则此字符串是 NDIS。如果 NDPROXY 定位到一个与 TAPI 应用程序传递的字符串匹配的 SAP，它就调用 **NdisMCmCreateVc** 建立一个连接边界点，在此边界点上它可以发出内入呼叫通知。NDIS 然后调用 NDISWAN 的 ProtocolCoCreateVc 函数并传递一个代表 VC 的句柄。
- 在 NDPROXY 建立连接边界点后，它调用 **NdisCmDispatchIncomingCall** 函数向 NDISWAN 通知内入呼叫。在此调用中，NDPROXY 传递了一个包含内入呼叫参数的 CO_AF_MAKE_CALL_PARAMETER 结构。NDIS 然后调用 NDISWAN 的 ProtocolClIncomingCall 函数，在此调用中 NDISWAN 可以接受或拒绝请求的连接。
- 在决定是否接受连接并可能改变呼叫参数后，NDISWAN 调用 **ProtocolClIncomingCallComplete** 函数。NDIS 然后调用微端口驱动程序的 ProtocolCmIncomingCallComplete 函数。根据 NDISWAN 是否接受了内入呼叫以及微端口驱动程序是否接受 NDISWAN 对呼叫参数的改变，微端口驱动程序调用 **NdisCmDispatchCallConnected** 或 **NdisCmDispatchIncomingCloseCall** 函数。**NdisCmDispatchCallConnected** 通知 NDISWAN 可以在 NDISWAN 为内入呼叫建立的 VC 上开始数据传输。**NdisCmDispatchIncomingCloseCall** 通知 NDISWAN 和 NDISWAN 拆卸内入呼叫。
- 在 NDISWAN 接受内入呼叫之后，NDPROXY 调用 **NdisCoGetTapiCallId** 函数检索标识 NDISWAN 的 VC 环境的字符串。NDPROXY 将此字符串传递回 TAPI 应用程序。TAPI 应用程序用此 VC 环境字符串完成对 **lineGetID** 的调用。

8.7.6 CoNDIS TAPI 关闭

TAPI 会话在 CoNDIS 广域网微端口驱动程序为应用程序列举完它的 TAPI 性能后开始。在一个会话中，可以打开一个或多个线路，并且可以发布一个或多个呼叫。在一个线路的开放期内，可以发布，然后关闭或丢弃多个呼叫。在会话期间，一个或多个线路可以经历多次从开放到关闭的转换。本节描述了微端口驱动程序如何处理这样的转换。

关闭呼叫

一个进行中的呼叫可以被本地或远程节点关闭。呼叫在本地节点被关闭，是因为拥有此呼叫的句柄的最后一个应用关闭了这个句柄，或可能因为微端口驱动程序的 *MiniportHalt* 或 *MiniportRest* 函数被调用。如果远程节点终止了一个正在运行的呼叫，微端口驱动程序必须通知上层拆卸此呼叫。

如果本地节点的应用程序要关闭呼叫，它必须断开呼叫。应用程序调用 TAPI **lineDrop** 函数断开呼叫。此 TAPI 函数调用导致 NDPROXY 调用 **NdisClCloseCall** 函数并传递一个代表 VC 的句柄。NDIS 然后调用 CoNDIS 广域网微端口驱动程序的 ProtocolCmCloseCall 函数。

微端口驱动程序应向 NDPROXY 返回 NDIS_STATUS_PENDING，这样微端口驱动程序就可以异步的完成 **NdisClCloseCall**。

微端口驱动程序的 ProtocolCmCloseCall 必须与网络控制设备通信以终止本地节点和远程节点间的连接。然后微端口驱动程序必需调用 **NdisMCmDeactivateVc** 函数指示用于此呼叫的 VC 的失效。

在微端口驱动程序终止连接后，ProtocolCmCloseCall 调用 **NdisMCmCloseCallComplete** 函数完成呼叫关闭。

如果远程节点中止了一个正在运行的呼叫，微端口驱动程序调用 **NdisCmDispatchIncomingCloseCall** 函数，通知 NDISWAN 和 NDPROXY 拆除内入呼叫。

关闭线路

在拥有此线路的开放句柄的最后一个应用关闭了这个句柄后，线路将被关闭。应用程序调用 TAPI **lineClose** 函数关闭线路。此 TAPI 函数调用导致 NDPROXY 发起对此线路上所有呼叫的关闭操作。微端口驱动程序应丢弃这些呼叫并清除它们的状态。

关闭会话

上层模块或 CoNDIS 广域网微端口驱动程序都可以发起会话的终止。在最后一个进程与高层电话模块分离后，应通知 NDPROXY 终止每个注册适配器的会话。为了完成这些操作，NDPROXY 驱动程序调用 **NdisClCloseAddressFamily** 函数并传递 TAPI 地址族的句柄。NDIS 然后调用微端口驱动程序的 ProtocolCmCloseAf 函数。微端口驱动程序应终止指定适配器上进行的相关活动并释放所有相关资源。在调用 **NdisClCloseAddressFamily** 后，客户应认为 TAPI 地址族的句柄已无效。

驱动程序发起的会话终止可以在微端口驱动程序被 MiniportHalt 函数卸载时发生。一般情况下，微端口驱动程序应完成所有未完成的 NDPROXY 请求并通知 NDISWAN 呼叫正在被关闭。如果后来微端口驱动程序又被重新加载，它应经历与先前描述相同的初始化过程。

如果 CoNDIS 广域网微端口驱动程序进行某些迫使所有客户和驱动程序完全重新初始化的动态重配置，它也可能会发起会话终止。例如，如果适配器的线路设备模块（如支持的线路设备数）被改变。

8.7.7 语音流对呼叫管理器的要求

本节描述了呼叫管理器或集成的微端口呼叫管理器（MCM）为了在面向连接介质上支持语音流必需达到的要求。这些需求可被分为以下几类：

- 8.7.7.1 响应 OID_CO_TAPI_LINE_CAPS 查询
- 8.7.7.2 为外出呼叫指定参数
- 8.7.7.3 为内入呼叫指定参数

8.7.7.1 响应 OID_CO_TAPI_LINE_CAPS 查询

为了响应 OID_CO_TAPI_LINE_CAPS 查询，呼叫管理器或 MCM 返回一个包含 LINE_DEV_CAPS 的 CO_TAPI_LINE_CAPS 结构。为了支持语音流，呼叫管理器或 MCM 必须在 LINE_DEV_CAPS 结构中指定下列值：

- **ulMediaModes**

此字段应包含 LINEMEDIAMODE_AUTOMATEDVOICE，它在 TAPI 3.0 中映射为 TAPIMEDIAMODE_AUDIO。

- **ulAddressTypes**

必须适当地填充此字段。参见 Platform SDK 中对 **dwAddressTypes** 的描述。此字段不能

为零。

- **ulGenerateDigitModes**

用 LINEDIGITMODE 常数的按位或填充此字段。LINEDIGITMODE 常数指定可以在线路上产生的数字模式。LINEDIGITMODE 常数的详细信息参见 Platform SDK 中对 **dwGenerateDigitModes** 的描述。

- **ulMonitorDigitModes**

用 LINEDIGITMODE 常数的按位或填充此字段。LINEDIGITMODE 常数指定可以在线路上被检测到的数字模式。LINEDIGITMODE 常数的详细信息参见 Platform SDK 中对 **dwMonitorDigitModes** 的描述。

8.7.7.2 为外出呼叫指定参数

在产生外出呼叫时，支持语音流的呼叫管理器或 MCM 必须为 CO_CALL_MANGET_PARAMETERS 结构提供下列值：

- 最大发送 SDU 长度（CallMgrParameters->Transmit.MaxSduSize）。
- 最大接收 SDU 长度（CallMgrParameters->Receive.MaxSduSize）。

一个支持除 CO_ADDRESS_FAMILY_TAPI_PROXY 之外其他地址族的呼叫管理器或 MCM，在响应 OID_CO_TAPI_TRANSLATE_TAPI_CALLPARAMS 请求将 TAPI 呼叫参数转换成 NDIS 呼叫参数时，填充这些值。

支持 CO_ADDRESS_FAMILY_TAPI_PROXY 地址族的呼叫管理器或 MCM，在函数 *ProtocolCmMakeCall* 的中将这些值写入 CO_CALL_MANGER_PARAMETERS 结构。注意传递给 ProtocolCmMakeCall 函数的最大 SDU 长度是错误的。ProtocolCmMakeCall 必须用正确值覆盖相应的错误值。

8.7.7.3 为内入呼叫指定参数

在用 **Ndis(M)CmDispatchIncomingCall** 指示内入呼叫时，支持语音流的呼叫管理器或 MCM 必须在 CO_CALL_MANGET_PARAMETERS 结构中指定以下值：

- 最大发送 SDU 长度（CallMgrParameters->Transmit.MaxSduSize）。
- 最大接收 SDU 长度（CallMgrParameters->Receive.MaxSduSize）。

另外，呼叫管理器或 MCM 在 LINE_CALL_INFO 结构中指定以下值：

- **ulMediaMode**

此字段应包含 LINEMEDIAMODE_AUTOMATEDVOICE，它在 TAPI 3.0 中变换成 TAPIMEDIAMODE_AUDIO。

- **ulCallerIDFlags**
- **ulCallerIDSize**
- **ulCallerIDOffset**
- **ulCallerIDNameSize**
- **ulCallerIDNameOffset**
- **ulCalledIDFlags**
- **ulCalledIDSize**
- **ulCalledIDOffset**
- **ulCalledIDNameSize**

- **ulCalledIDNameOffset**
- **ulCallerIDAddressType**
- **ulCalledIDAddressType**

一个支持除 CO_ADDRESS_FAMILY_TAPI_PROXY 之外其他地址族的呼叫管理器或 MCM，在响应 OID_CO_TAPI_TRANSLATE_TAPI_CALLPARAMS 请求时，指定上述 LINE_CALL_INFO 成员。

一个支持 CO_ADDRESS_FAMILY_TAPI_PROXY 地址族的呼叫管理器或 MCM，在提供给 **Ndis(M)CmDispatchIncomingCall** 的 CO_CALL_MANGER_PARAMETERS 结构的介质相关部分，指定上述 LINE_CALL_INFO 成员。

对 LINE_CALL_INFO 成员的描述详见 Platform SDK 中的 LINECALLINFO 结构。

8.7.8 在面向连接 NDIS 之上支持电话服务的非广域网专用的扩展

本节描述了在面向连接 NDIS 之上提供 TAPI 的非广域网专用的扩展。此扩展是一些 NDIS/TAPI 转换 OID。这些扩展允许非广域网专用的呼叫管理器和集成微端口呼叫管理器（MCM）将 TAPI 参数转换成 NDIS 参数，或将 NDIS 参数转换成 TAPI 参数。例如，这些扩展允许支持 ATM 的呼叫管理器和 MCM 提供面向连接介质上 TAPI 访问。在面向连接 NDIS 上提供 TAPI 支持的广域网专用的扩展的信息参见 8.7 使用支持电话服务的 CoNDIS 扩展中的 8.7.2 到 8.7.6。

这些 NDIS/TAPI 转换 OID 不能用于分别用 NdisCmRegisterAddressFamily 或 NdisMCmRegisterAddressFamily 注册 CO_ADDRESS_FAMILY_TAPI_PROXY 的呼叫管理器或 MCM。这些呼叫管理器和 MCM，与 TAPI 客户一样，应将 TAPI 参数封装到面向连接结构中。参见 8.7 使用支持电话服务的 CoNDIS 扩展。

NDIS/TAPI 转换 OID 有：

- OID_CO_TAPI_TRANSLATE_TAPI_CALLPARAMS

此 OID 请求呼叫管理器或 MCM 将客户提供的 TAPI 呼叫参数转换成 NDIS 呼叫参数。客户一般使用呼叫管理器或 MCM 返回的 NDIS 呼叫参数作为 **NdisClMakeCall** 的输入（以 CO_CALL_PARAMETERS 格式）。客户使用 **NdisClMakeCall** 发起面向连接的呼叫。

- OID_CO_TAPI_TRANSLATE_NDIS_CALLPARAMS

此 OID 请求呼叫管理器或 MCM 为一个内入呼叫将 NDIS 呼叫参数（在 CO_CALL_PARAMETERS 结构中传递给客户的 *ProtocolClIncomingCall* 函数）转换为 TAPI 呼叫参数。客户使用呼叫管理器或 MCM 返回的 TAPI 呼叫参数决定是否接受内入呼叫。

- OID_CO_TAPI_TRANSLATE_SAP

此 OID 请求呼叫管理器或 MCM 用客户提供的 TAPI 呼叫参数准备一个或多个 NDIS SAP。客户一般将呼叫管理器或 MCM 返回的 NDIS SAP 作为 **NdisClRegisterSap** 的输入（以 CO_SAP 格式），客户通过此函数注册一个接受内入呼叫的 SAP。

第九章　任务卸载

为了显著的提高性能，Microsoft TCP/IP 传输层可以将以下任务推卸给拥有适当任务卸载能力的 NIC：

- 校验和任务

TCP/IP 传输层能够卸载对 IP 和/或 TCP 校验和的计算和/或验证。Window® 2000 的最初版本不支持卸载 UDP 校验和；将来 Window® 2000 的服务包和升级版本可能会支持卸载 UPD

校验和。

- IP 安全性任务

TCP/IP 传输层能够卸载对身份验证报头（authentication headers，AH）和/或封装安全负载（encapsulating security payloads，ESP）的加密校验和的计算和/或验证。TCP/IP 传输层还能够卸载对 ESP 负载的加密和解密。

- 大 TCP 包分段

TCP/IP 传输层能够卸载对大 TCP 包的分段处理。

Window® 2000 中的 TCP/IP 传输层可以处理 IPv4 包（但不能处理 IPv6 包），因此只对 IPv4 支持任务卸载。

从 TCP/IP 传输层卸载任务到 NIC 包括两个主要操作：

- 查询并启动 NIC 的任务卸载功能（参加 9.1 和 9.2）。
- 传递涉及卸载任务的 per-packet 信息（扩展的带外数据）。参见 9.4、9.5、9.6、9.7 节。

9.1 查询任务卸载能力

TCP/IP 传输层通过查询 OID_TCP_TASK_OFFLOAD 确定 NIC 的任务卸载能力。在此查询中，TCP/IP 传输层在 *InformationBuffer* 中提供一个 NDIS_TASK_OFFLOAD_HEADER，定义如下：

```
typedef  struct  _NDIS_TASK_OFFLOAD_HEADER
{
    ULONG   Version;
    ULONG   Size;
    ULONG   Reserved;
    UCHAR   OffsetFirstTask;
    NDIS_ENCAPSULATION_FORMAT   EncapsulationFormat;
}  NDIS_TASK_OFFLOAD，*PNDIS_TASK_OFFLOAD;
```

NDIS_TASK_OFFLOAD_HEADER 结构指定以下内容：

- TCP/IP 传输层支持的任务卸载版本。
- NDIS_TASK_OFFLOAD_HEADER 结构的长度（用于检查此结构的版本）。
- 从 *InformationBuffer* 的开始到第一个 NDIS_TASK_OFFLOAD 结构的偏移量（字节）。
- 在 TCP/IP 传输层和微端口之间传递的发送和接收包的封装格式。

使用 OID_TASK_OFFLOAD_HEADER 提供的信息，微端口或它的 NIC 能够在发送或接收包中定位第一个 IP 报头的开始。这是执行卸载任务的先决条件。

作为对 OID_TCP_TASK_OFFLOAD 查询的响应，微端口在 *InformationBuffer* 中返回 NDIS_TASK_OFFLOAD_HEADER，后跟一个或多个 NDIS_TASK_OFFLOAD 结构。NDIS_TASK_OFFLOAD 结构定义如下：

```
typedef  struct  _NDIS_TASK_OFFLOAD
{
    ULONG   Version;
    ULONG   Size;
    NDIS_TASK   task;
    ULONG   OffsetNextTask;
```

```
    ULONG    TaskBufferLength;
    UCHAR    TaskBuffer[1];
}   NDIS_TASK_OFFLOAD，*PNDIS_TASK_OFFLOAD;
```

NDIS_TASK_OFFLOAD 结构指定以下内容：

- 指定卸载任务的版本。
- NDIS_TASK_OFFLOAD 结构的长度（用于检查此结构的版本）。
- 要卸载的任务。
- 从 *InformationBuffer* 的开始到下一个 NDIS_TASK_OFFLOAD 结构的偏移量（字节）。
- **TaskBuffer** 的长度。
- 一个包含任务相关信息的变长数组。

每个 NDIS_TASK_OFFLOAD 结构指定一个微端口的 NIC 支持的任务卸载能力。如果微端口的 NIC 支持特定任务卸载能力的多个版本，它将为每个版本返回一个 NDIS_TASK_OFFLOAD 结构。

9.1.1 报告 NIC 的校验和性能

如果微端口的 NIC 能够计算和/或验证 IP，TCP，和/或 UDP 校验和，微端口将在 NDIS_TASK_TCP_IP_CHECKSUM 结构中指出上述性能。微端口用 NDIS_TASK_TCP_IP_CHECKSUM 结构覆盖 NDIS_TASK_OFFLOAD 结构的 **TaskBuffer**，并在 *InformationBuffer* 中返回 NDIS_TASK_OFFLOAD 结构作为对 OID_TCP_TASK_OFFLOAD 查询的响应。

NDIS_TASK_TCP_IP_CHECKSUM 结构的定义如下：

```
typedef   struct   _NDIS_TASK_TCP_IP_CHECKSUM
{
    struct
    {
     ULONG    IpOptionSupported:1’
     ULONG    TcpOptionSupported:1;
     ULONG    TcpChecksum:1;
     ULONG    UdpChecksum:1;
     ULONG    IpChecksum:1;
    }V4Transmit;
    struct
    {
     ULONG    IpOptionSupported:1’
     ULONG    TcpOptionSupported:1;
     ULONG    TcpChecksum:1;
     ULONG    UdpChecksum:1;
     ULONG    IpChecksum:1;
    }V4Receive;
    struct
    {
     ULONG    IpOptionSupported:1’
     ULONG    TcpOptionSupported:1;
```

```
    ULONG    TcpChecksum:1;
    ULONG    UdpChecksum:1;
  }V6Transmit;
  struct
  {
    ULONG    IpOptionSupported:1'
    ULONG    TcpOptionSupported:1;
    ULONG    TcpChecksum:1;
    ULONG    UdpChecksum:1;
  }V6Receive;
```

微端口为 IPv4 发送包和 IPv4 接收包指示以下校验能力：

- NIC 是否能够为在 IP 报头中包含 IP 选项的包计算和/或验证校验和。
- NIC 是否能够为在 TCP 报头中包含 TCP 选项的包计算和/或验证校验和。
- NIC 为发送包计算为接收包验证的校验和（IP，TCP，和/或 UDP）的类型。Window® 2000 的最初版本不支持卸载 UDP 校验；将来 Window® 2000 的服务包和升级版本可能会支持卸载 UPD 校验。

微端口可以为 IPv6 接收包和发送包指示类似的校验和能力；但是因为 Window 2000 的最初版本只能处理 IPv4 包，它从不为 IPv6 包激活校验能力。

9.1.2 报告 NIC 的 IP 安全性性能

微端口在 NDIS_TASK_IPSEC 结构中指定它的 NIC 的安全性性能。微端口用此结构覆盖 NDIS_TASK_OFFLOAD 结构的 **TaskBuffer**，然后为了响应 OID_TCP_TASK_OFFLOAD 查询，微端口在 *InformationBuffer* 中返回 NDIS_TASK_OFFLOAD 结构。

NDIS_TASK_IPSEC 结构定义如下：

```
typedef   struct   _NDIS_TASK_IPSEC
{
  struct
  {
    ULONG    AH_ESP_COMBIND:1;
    ULONG    TRANSPORT_TUNNEL_COMBINE:1;
    ULONG    V4_OPTION:1;
    ULONG    RESERVED:1;
  }Supported;

  struct
  {
    ULONG    MD5:1;
    ULONG    SHA_1:1;
    ULONG    Transport:1;
    ULONG    Tunnel:1;
    ULONG    Send:1;
    ULONG    Receive:1;
```

```
    }V4AH;

    struct
    {
     ULONG    DES:1;
     ULONG    RESERVED:1;
     ULONG    TRIPLE_DES:1;
     ULONG    NULL_ESP:1;
     ULONG    Transport:1;
     ULONG    Tunnle:1;
     ULONG    Send:1;
     ULONG    Receive:1;
    }V4ESP
}NDIS_TASK_IPSEC，*PNDIS_TASK_IPSEC；
```

微端口在 NDIS_TASK_IPSEC 结构中指定下列通用性能：

- 它的 NIC 是否能够在包上执行组合的 IP 安全操作。也就是说，处理既包含身份验证报头（AH）又包含封装安全负载（ESP）的包，这个包的格式是[IP][AH][ESP][包的剩余部分]。
- 它的 NIC 是否在接收和发送包的传输模式和隧道模式两部分都可以进行 IP 安全处理。包的传输模式部分属于端到端安全。包的隧道模式部分属于隧道安全。
- 它的 NIC 是否能够在 IP 报头中包含 IP 选项的包上执行 IP 安全操作。

为了给 AH 负载和身份确认信息计算和/或验证加密校验和，微端口为其 NIC 指定以下性能：

- NIC 可以使用的完全性算法（MD5 或 SHA 1）。
- NIC 是否可以为包的传输模式部分处理 AH 安全性负载。
- NIC 是否可以为包的隧道模式部分处理 AH 安全性负载。
- NIC 是否可以为发送包处理 AH 安全性负载。
- NIC 是否可以为接收包处理 AH 安全性负载。

为了处理 ESP 负载，微端口为其 NIC 指定以下性能：

- NIC 可以使用的机密性算法（DES 和/或三重 DES）。
- NIC 是否支持空加密（null encryption），即没有加密但有身份验证散列的 ESP 负载。
- NIC 能否为包的传输层模式部分进行 ESP 处理。
- NIC 能否为包的隧道模式部分进行 ESP 处理。
- NIC 能否为发送进行 ESP 处理。
- NIC 能否为接收进行 ESP 处理。

9.1.3 报告 NIC 的 TCP 包分段性能

微端口在 NDIS_TASK_TCP_LARGE_SEND 结构中指定它的 NIC 的包分段性能。微端口用此结构覆盖 NDIS_TASK_OFFLOAD 结构的 **TaskBuffer**，然后为了响应 OID_TCP_TASK_OFFLOAD 查询，微端口在 *InformationBuffer* 中返回 NDIS_TASK_OFFLOAD 结构。

NDIS_TASK_TCP_LARGE_SEND 结构的定义如下：

```
typedef     struct    _NDIS_TASK_TCP_LARGE_SEND
{
    ULONG         MaxOffloadSize;
    ULONG         MinSegmentCount;
    BOOLEAN    TcpOptions;
    BOOLEAN    IpOptions;
} NDIS_TASK_TCP_LARGE_SEND，*PNDIS_TASK_TCP_LARGE_SEND;
```

微端口必需在 NDIS_TASK_TCP_LARGE_SEND 结构中指定以下信息：

- **MaxOffLoadSize**，在一个大 TCP 包中，TCP/IP 传输层可以传递给微端口的用户数据的最大字节数。
- **MinSegmentCount**，在 TCP/IP 传输层可以将分段推卸给 NIC 之前，大 TCP 必须被分割成的最小段数。
- NIC 是否可以分割包含 TCP 选项的 TCP 包。
- NIC 是否可以分割包含 IP 选项的 TCP 包。

9.2 启用任务卸载能力

TCP/IP 传输层通过设置 OID_TCP_TASK_OFFLOAD 启用 NIC 的任务卸载能力。在此设置操作中，TCP/IP 传输层在 *InformationBuffer* 中传递一个 NDIS_TASK_OFFLOAD_HEADER 结构，在此结构后紧跟若干 NDIS_TASK_OFFLOAD 结构，每个 NDIS_TASK_OFFLOAD 对应要卸载的任务。

微端口必需检查 *InformationBuffer* 中的每个 NDIS_TASK_OFFLOAD 以确定哪个卸载任务正被启用。微端口还必需检查每个 NDIS_TASK_OFFLOAD 的 **TaskBuffer** 中的任务相关结构（NDIS_TASK_TCP_IP_CHECKSUM，NDIS_TASK_IPSEC，或 NDIS_TASK_TCP_LARGE_SEND），以确定正在启用特定卸载任务的哪个能力。

TCP/IP 传输层只用一个 OID_TCP_TASK_OFFLOAD 设置为 NIC 启用所有卸载能力。微端口必需使后续所有设置 OID_TCP_TASK_OFFLOAD 的试图失败。

9.3 停用任务卸载能力

为了停用 NIC 的所有任务卸载能力，TCP/IP 传输层设置 OID_TCP_TASK_OFFLOAD，在 *InformationBuffer* 中只传递一个 NDIS_TASK_OFFLOAD_HEADER 结构。TCP/IP 传输层将 NDIS_TASK_OFFLOAD_HEADER 结构的 **OffsetFirstTask** 成员置为 0。

9.4 访问 Per-Packet 信息

TCP/IP 传输层和微端口之间用包的 Per-Packet 信息互相传递关于任务卸载操作的信息。Per-Packet 信息是用 NDIS_PACKET_EXTENSION 结构表示的带外信息，此结构的定义如下：

```
typedef  struct  _NDIS_PACKET_EXTENSION
{
  PVOID  NdisPacketInfo[MaxPerPacketInfo];
} NDIS_PACKET_EXTENSION，*PNDIS_PACKET_EXTENSION;
```

用 **NdisAllocatePacket** 分配的每个包描述信息都有一个相关的 NDIS_PACKET_EXTENSION 结构。此结构包含一个指针数组（PVOID），每个数组包含或指向一个与包描述信息相关的 Per-Packet 信息的特定类型。

驱动程序使用 NDIS_PACKET_EXTENSION_FROM_PACKET 宏或 NDIS_PER_PACKET_INFO_FROM_PACKET 宏访问 Per-Packet 信息。

NDIS_PACKET_EXTENSION_FROM_PACKET 宏的定义如下：

```
PNDIS_PACKET_EXTENSION
      NDIS_PACKET_EXTENSION_FROM_PACKET（
            IN  PNDIS_PACKET  Packet
            ）;
```

此宏返回一个指向与给定包描述信息相关的 NDIS_PACKET_EXTENSION 结构的指针。当驱动程序需要访问与包描述信息相关的多个类型的 Per-Packet 信息时，它应调用此宏。在获得指向 NDIS_PACKET_EXTENSION 结构的指针后，驱动程序可以使用适当的数组索引访问特定类型的 Per-Packet 信息。属于卸载任务的索引值有：

TcpIpCheckSumPacketInfo

索引一个指向 NDIS_TCP_IP_CHECKSUM_PACKET_INFO 结构的指针。此结构为卸载的校验和操作向微端口指定 Per-Packet 信息。

IpSecPacketInfo

索引一个指向 NDIS_IPSEC_PACKET_INFO 结构的指针。此结构为 IP 安全性操作向微端口指定 Per-Packet 信息。

TcpLargeSendPacketInfo

索引指向一个 ULONG 值的指针。此值用于卸载大 TCP 包分段。

NDIS_PER_PACKET_INFO_FROM_PACKET 宏定义如下：

```
PVOID
    NDIS_PER_PACKET_INFO_FROM_PACKET（
          IN  PNDIS_PACKET  Packet，
          IN  NDIS_PER_PACKET_INFO  InfoType
              ）;
```

此宏为指定包返回一个指向或包含 Per-Packet 信息的特定类型的指针。属于卸载任务的 *InfoType* 值有 **TcpIpCheckSumPacketInfo**、**IpSecPacketInfo**，和 **TcpLargeSendPacketInfo**。这些值与上面描述的索引值一致。例如，若指定的 *InfoType* 是 **TcpIpCheckSumPacketInfo**，此宏返回一个指向与给定包描述信息相关的 NDIS_TCP_IP_CHECKSUM_PACKET_INFO 结构的指针。在驱动程序需要访问 Per-Packet 信息的一个类型时，它应调用 NDIS_PER_PACKET_INFO_FROM_PACKET 宏。

9.5 卸载 TCP/IP 校验和任务

负载平衡微端口不能卸载 TCP/IP 校验和任务。负载平衡的更多信息参见第十章。

在向微端口传递一个包描述信息（微端口将要在此包上执行一个或多个校验任务）之前，TCP/IP 传输层指定与此包描述信息相关的校验信息。此信息用 NDIS_TCP_IP_CHECKSUM_PACKET_INFO 结构表示，它是与包描述信息相关的 Per-Packet 信息（扩展的带外信息）的一部分。

NDIS_TCP_IP_CHECKSUM_PACKET_INFO 结构的定义如下：

```
typedef  struct  _NDIS_TCP_IP_CHECKSUM_PACKET_INFO
{
    union
    {
      struct
      {
        ULONG     NdisPacketChecksumV4:1;
        ULONG     NdisPacketChecksumV6:1;
        ULONG     NdisPacketTcpChecksum;
        ULONG     NdisPacketUdpChecksum;
        ULONG     NdisPacketIpChecksum;
      } Transmit;

      struct
      {
        ULONG     NdisPacketTcpChecksumFailed:1;
        ULONG     NdisPacketUdpChecksumFailed:1;
        ULONG     NdisPacketIpChecksumFailed:1;
        ULONG     NdisPacketTcpChecksumSucceeded:1;
        ULONG     NdisPacketUdpChecksumSucceeded:1;
        ULONG     NdisPacketIpChecksumSucceeded:1;
        ULONG     NdisPacketLoopback:1;
      } Receive;

      ULONG    Value;
     };
}NDIS_TCP_IP_CHECKSUM_PACKET_INFO，
 *PNDIS_TCP_IP_CHECKSUM_PACKET_INFO;
```

在将对 TCP 包的校验和的计算卸载之前，TCP/IP 传输层为 TCP 虚拟报头计算校验和。TCP/IP 传输层的虚拟报头的校验和计算跨越了虚拟报头的所有字段，包括源 IP 地址、目标 IP 地址、协议字段、和 TCP 包的 TCP 长度。TCP/IP 传输层将虚拟报头校验和放到 TCP 报头的校验和字段中。

TCP/IP 传输层提供的虚拟报头校验和是 NIC 为发送包计算的实 TCP 校验和的提前注入量。为了计算实 TCP 校验和，NIC 计算 TCP 校验和（对 TCP 报头和负载）的可变部分，将

此校验和加到 TCP/IP 传输层计算的虚拟报头校验和上，然后为此校验和计算 16 位的补码。（计算这样的校验和的更多信息，参见 RFC 793 和 RFC 1122）。

在从 *MiniportSend*、*MiniportSendPacket*、和 *MiniportCoSendPacket* 函数收到包描述信息后，微端口一般进行以下校验和处理：

1. 微端口调用 NDIS_GET_PACKET_PROTOCOL_TYPE 宏确定包的协议。
2. 如果宏返回的协议 ID 不是 NDIS_PROTOCOL_ID_TCP_IP，微端口的 NIC 不应在此包上进行任何校验操作。
3. 如果返回的协议 ID 是 NDIS_PROTOCOL_ID_TCP_IP，微端口调用 NDIS_PER_PACKET_INFO_FROM_PACKET 宏（其中 *InfoType* 值为 TcpIpCheckSumPacketInfo），获得一个指向 NDIS_TCP_IP_CHECKSUM_PACKET_INFO 结构的指针。（或者微端口调用 NDIS_PACKET_EXTENSION_FROM_PACKET 宏获得一个指向 NDIS_PACKET_EXTENSION 结构的指针。然后微端口使用 TcpIpCheckSumPacketInfo 数组索引获得一个指向 NDIS_TCP_IP_CHECKSUM_PACKET_INFO 结构的指针。）
4. 微端口检查 NDIS_TCP_IP_CHECKSUM_PACKET_INFO 结构中的 NdisPacketChecksumV4 标记。若未设置 NdisPacketChecksumV4 标记，微端口的 NIC 不应在此包上进行任何校验操作。
5. 若已设置 NdisPacketChecksumV4 标记，微端口检查 MdisPacketTcpChecksum、MdisPacketUdpChecksum、和 MdisPacketIpChecksum 标记，确定它的 NIC 应为包计算哪些校验和。
6. 微端口将包传递给它的 NIC，NIC 将为包计算适当的校验和。如果包既有一个隧道 IP 报头，又有一个传输 IP 报头，那么支持 IP 校验卸载的 NIC 只在隧道报头上执行 IP 校验任务。TCP/IP 传输层在传输 IP 报头上执行 IP 校验任务。

在指示一个接收包之前，微端口验证适当的校验和并在 NDIS_TCP_IP_CHECKSUM_PACKET_INFO 结构中置 **NdisPacket*Xxx*ChecksumFailed** 和/或 **NdisPacket*Xxx*ChecksumSucceeded** 标记。如果此包是一个回送包，微端口设置 **NdisPacketLoopback** 标记和所有的 **NdisPacket*Xxx*ChecksumSucceeded** 标记。

9.6 卸载 IP 安全任务

本节假定读者对在 RFC 中和在 Internet Engineering Task Force（IETF）的 IP Security Working Group 发布的草稿中指定的 IP 安全性有一定了解。

IPSec 体系结构：

- *Security Architecture for the Internet Protocol*（RFC 2401）

身份验证报头（AH）：

- *IP Authentication Header*（RFC 2402）
- *The Use of HMA-MD5-96 within ESP and AH*（RFC 2403）
- *The Use of HMA-SHA-1-96 within ESP and AH*（RFC 2404）
- *HMAC-MD5 IP Authentication with Replay Prevention*（RFC 2085）

封装安全负载（ESP）：

- *IP Encapsulating Security Payload*（ESP）（RFC 2406）

- *The ESP CBC-Mode Cipher Algorithms*（RFC 2451）
- *The ESP DES-Mode Cipher Algorithms with Explicit IV*（RFC 2405）
- *The NULL Encryption Algorithm and its Use with IPSec*（RFC 2410）

对 IP 安全性卸载的需求和限制

对 IP 安全性卸载有以下需求和限制：

- NIC 必须保持一个安全性联系表（Security Association SA）。这使在发送包中不需要包含密钥或其他 AH 和 ESP 需要的信息，因此改善了性能。
- NIC 应能同时为单个包处理 AH 和 ESP 负载。在此情况下，NIC 必须为 AH 和 ESP 支持以下完整性（身份验证）算法的组合。

AH	ESP
MD5	MD5
SHA 1	SHA 1
MD5	SHA 1
SHA 1	MD5
MD5	Null（只有在 NIC 支持空加密时）
SHA 1	Null（只有在 NIC 支持空加密时）

- 支持 DES 算法的 NIC 必须生成这些算法需要的 IV。
- NIC 执行的 IP 安全性任务处理加密的 AH 校验和和/或 ESP 校验和，以及加密和解密 ESP 负载。对于发送包，TCP/IP 传输层创建所有的报头、填充、和重发编号，并选择只有目标地址/IP 安全性协议对才有的 SPI 值。对于接收包，TCP/IP 传输层执行内入策略检查，处理重发检测和预防，并处理审计事件。
- 对于发送包，TCP/IP 传输层不提供显式的偏移量——例如，指示加密数据的开始的偏移量——因为卸载驱动程序能够非常容易的从用于处理包的特定安全性联系（SA）中确定这些信息。
- 使用 IP 安全协议的包在身份验证报头（AH）和/或封装安全负载（ESP）中必须有验证信息。决不允许 IP 安全包不带验证信息的。
- 不能为需要 IP 分段的发送包，或为需要装配 IP 分段的接收包卸载 IP 安全性任务。
- 不能为通过负载平衡微端口发送和接收的包卸载 IP 安全性任务。负载平衡的详细信息参见第十章。

为 NIC 添加安全性联系

在 TCP/IP 传输层确定一个 NIC 可以执行 IP 安全性操作后（参见 9.1.2），传输层必需请求 NIC 的微端口在传输层将 IP 安全性任务推卸给 NIC 之前为 NIC 添加一个或多个内入和外出安全性联系（SA）。TCP/IP 传输层用 OID_TCP_TASK_IPSEC_ADD_SA 请求微端口为它的 NIC 添加一个或多个 SA。

从 NIC 删除安全性联系

如果必要，TCP/IP 传输层设置 OID_TCP_TASK_IPSEC_DELETE_SA 以请求微端口从它的 NIC 删除安全性联系（SA）。

为了给 NIC 上其他 SA 腾出空间，微端口可以为一个接收包在 NDIS_IPSEC_PACKET_INFO 结构中设置 **SA_DELETE_REQ**。TCP/IP 传输层随后发出 OID_TCP_TASK_IPSEC_DELETE_SA 删除内入安全性联系（SA），然后再一次发出 OID_TCP_TASK_IPSEC_DELETE_SA 删除与内入 SA 对应的外出 SA。在接收到相应的 OID_TCP_TASK_IPSEC_DELETE_SA 请求之前，微端口的 NIC 不得删除任何 SA。微端口

对 **SA_DELETE_REQ** 的设置与 **CRYPTO_DONE** 无关。

卸载发送方的 IP 安全性任务

在为一个包向微端口发送一个包描述信息之前，TCP/IP 传输层更新与此包描述信息联系的 IP 安全性信息。TCP/IP 传输层在一个 NDIS_IPSEC_PACKET_INFO 结构中指定此信息，NDIS_IPSEC_PACKET_INFO 结构是与此包描述信息联系的 Per-Packet 数据（扩展的带外数据）。

NDIS_IPSEC_PACKET_INFO 结构的定义如下：

```
typedef   struct   _NDIS_IPSEC_PACKET_INFO
{
    union
    {
        struct
        {
            NDIS_HANDLE   OffloadHandle;
            NDIS_HANDLE   NextOffloadHandle;
        }   Transmit;

        struct
        {
            ULONG   SA_DELETE_REQ:1;
            ULONG   CRYPTO_DONE:1;
            ULONG   NEXT_CRYPTO_DONE:1;
            ULONG   CryptoStatus;
         }   Receive;
    };
} NDIS_IPSEC_PACKET_INFO，*PNDIS_IPSEC_PACKET_INFO ；
```

TCP/IP 传输层提供一个 *OffloadHandle*，为发送包的传输（端到端连接）部分指定外出 SA 的句柄。如果包将通过隧道传输，TCP/IP 传输层还提供 *NextOffloadHandle*，为发送包的隧道部分指定外出 SA 的句柄。

在微端口从它的 *MiniportSend*、*MiniportSendPacket*、或 *MiniportCoSendPacket* 函数接收到包描述信息之后，它以参数 *InfoType* 为 **IpSecNdisTask** 调用 NDIS_PER_PACKET_INFO_FROM_PACKET 宏，以获得与此包描述信息联系的 NDIS_PACKET_EXTENSION 结构。然后微端口就可以用 IpSecPacketInfo 数组索引获得指向 NDIS_IPSEC_PACKET_INFO 结构的指针。

当在发送包上执行 IP 安全性处理时，如果此包包含一个 ESP 负载并还要加密此包，NIC 为包计算 AH 和/或 ESP 加密校验和。TCP/IP 传输层已经构建、填充了此包，并且在必要时为它分配了顺序号和 SPI。

卸载接收方的 IP 安全任务

当在接收包上执行 IP 安全性处理时，NIC 为包含 ESP 负载的包解密并为包计算 AH 和/或 ESP 加密校验和。在向 TCP/IP 传输层指示此包前，微端口以参数 *InfoType* 为 **IpSecNdisTask** 调用 NDIS_PER_PACKET_INFO_FROM_PACKET 宏以获得一个指向与此包相关的 NDIS_IPSEC_PACKET_INFO 结构的指针，或调用 NDIS_PACKET_EXTENSION_FROMPACKET 宏以获得一个指向与包描述信息相关的

NDIS_PACKET_EXTENSION 结构的指针。

微端口通过在 NDIS_IPSEC_PACKET_INFO 结构中设置 CRYPTO_DONE 标记指出它的 NIC 至少在接收包的一个 IP 安全性负载上执行了 IP 安全检查。如果微端口的 NIC 在接收包的隧道和传输部分都执行了 IP 安全检查，微端口还要在 NDIS_IPSEC_PACKET_INFO 结构中设置 **NEXT_CRYPTO_DONE** 标记。只有在包同时具有隧道和传输 IP 安全性负载时才能设置 **NEXT_CRYPTO_DONE** 标记；否则，**NEXT_CRYPTO_DONE** 被置为零。微端口还必须在 NDIS_IPSEC_PACKET_INFO 结构中提供 **CryptoStatus** 的值，表示 IP 安全检查的结果。如果 NIC 求校验和或解密时发生错误，无论此包的格式如何，微端口必须向上指示此接收包，并指定适当的 **CryptoStatus** 值。

在微端口将包指示给 TCP/IP 传输层后，传输层检查由 NIC 执行的 IP 安全性检查的结果、为包检查顺序号、并决定如何处理校验失败和/或顺序号检查失败的包。

网络接口的变化对 IP 安全卸载的影响

网络接口的以下改变会影响 IP 安全任务的卸载：

- NIC 被删除

在从系统中删除一个执行被卸载任务的 NIC 之前，它的微端口应从 NIC 上删除所有的安全性联系（SA）。微端口不必请求 TCP/IP 传输层删除 SA。

- 路由接口被改变

当网络通信通过一个新的接口发送时，TCP/IP 栈暂时由它自己执行 IP 安全性任务，直到它在新的接口上将适当的 SA 添加到 NIC 上。TCP/IP 栈通过发布 OID_TCP_TASK_IPSEC_ADD_SA 将 SA 添加到 NIC 上。NIC 上用于旧接口的 SA 失效后，TCP/IP 传输层发布多次 OID_TCP_TASK_IPSEC_DELETE_SA，请求 NIC 的微端口从 NIC 上删除 SA。

9.7 卸载大 TCP 包分段

支持对大 TCP 包（大于网络介质的最大传输单元的 TCP 包）进行分割的 NIC 必须能够：

- 为包含 IP 选项的发送包计算 IP 校验和。
- 为包含 TCP 选项的发送包计算 TCP 校验和。

如果微端口指出它的 NIC 支持对大 TCP 包进行分段，TCP/IP 栈假定 NIC 可以对包含 IP 选项和/或 TCP 选项的大 TCP 包执行此分段操作。

TCP/IP 传输层只对符合以下标准的大 TCP 包卸载分段操作：

- 必须是一个 TCP 包。TCP/IP 传输层不卸载大 UDP 包的分段操作。
- 包必需至少应被分为微端口指定的最小分段数。（参见 9.1.3）
- 不能是回送包。
- 此包不能通过隧道发送。
- 此包不能被送往负载平衡微端口。负载平衡参见第十章。

在为大 TCP 包卸载分段操作之前，TCP/IP 传输层做以下操作：

- 更新与包描述信息有关的大包分段信息。此信息是一个 ULONG 值，是与包描述信息有关的 Per-packet 信息的一部分。（关于 Per-packet 信息的详见 online DDK 中 *Network Driver Reference* 中的 NDIS_PACKET_EXTENSION）。TCP/IP 传输层将此 ULONG 值置为最大段长度（MSS），相当于当前的最大传输单元（MTU）。MTU 是微端口当前可以在一个包内包含的用户数据的最大字节数。

- 将大 TCP 包的总长度写入包的 IP 报头的总长度字段。总长度包括：IP 报头的长度、IP 选项的长度（如果存在）、TCP 报头的长度、TCP 选项的长度（如果存在）、TCP 负载的长度。
- 为 TCP 虚拟报头计算校验和，并将此虚拟报头校验和写入 TCP 报头的校验和字段。TCP/IP 传输层在虚拟报头的以下字段上计算虚拟报头校验和：源 IP 地址、目标 IP 地址、和协议字段。TCP/IP 传输层提供的虚拟报头校验和为 NIC 对每个包进行的实 TCP 校验和计算提供了一个提前注入量，这样 NIC 就直接从 TCP 包计算而不必检查 IP 报头。
- 将当前的顺序号写入 TCP 报头的顺序号字段。顺序号标识 TCP 负载的第一个字节。

在微端口从它的 *MiniportSend*、*MiniportSendPacket*、或 *MiniportCoSendPacket* 函数中获得包描述信息之后，它调用 NDIS_PER_PACKET_INFO_FROM_PACKET 宏，其中将 *InfoType* 置为 **TcpLargeSendPacketInfo**，以获得 TCP/IP 传输层写入的 MSS 值。或者，微端口调用 NDIS_PACKET_EXTENSION_FROM_PACKET 宏获得一个指向 NDIS_PACKET_EXTENSION 结构的指针，然后微端口就可以使用 **TcpLargeSendPacketInfo** 数组索引获得 MSS 值。

微端口从包的 IP 报头获得包的总长度，并使用 MSS 值将大 TCP 包分段为较小的包。每个较小的包都包含 MSS 或较少的数据字节。

微端口将 IP 和 TCP 报头附加到每个从大包分出的段上。微端口必须为这些包的报头计算 IP 和 TCP 校验和。为了计算 TCP 校验和，NIC 计算 TCP 校验和的可变部分（为 TCP 报头和 TCP 负载），将此校验和加到 TCP/IP 传输层计算的虚拟报头校验和上，然后为此校验和计算 16 位的补码。（计算这样的校验和的更多信息，参见 RFC 793 和 RFC 1122）。

以下假定和限制应用于对 IP 和 TCP 报头的处理：

- TCP/IP 传输层卸载的大 TCP 包的 IP 报头中的 MF 位不应被置位，且 IP 报头中的分段偏移量应为 0。
- 大 TCP 包的 IP 报头中的 URG、RST 和 SYN 标记不应被置位，且 IP 报头的紧急偏移量应为 0。
- 如果大 TCP 包的 IP 报头中的 FIN 位被置位，微端口必须在从大 TCP 包创建的最后一个包的 TCP 报头中设置此位。
- 如果大 TCP 包的 IP 报头中的 PUSH 位被置位，微端口必须在从大 TCP 包创建的最后一个包的 TCP 报头中设置此位。
- 如果大 TCP 包包含 IP 选项和/或 TCP 选项，微端口将把这些选项原封不动的复制到从大 TCP 包分出的每个包中。特别是，NIC 不会使时间戳选项增值。

微端口必须以从 TCP/IP 传输层接收的顺序发送包。然而，从大 TCP 包分出来的包可以与其他包交错。

微端口还必须保持 ACK（对从大 TCP 包分出的包的应答）的顺序。微端口必须以接收的顺序向上层指示包含 ACK 的包，这样 TCP/IP 传输层可以正确执行快速中继算法、快速恢复算法、和选择 ACK 算法。

在为大包完成发送操作（例如，使用 **NdisMSendComplete** 或 **NdisMCoSendComplete** 完成发送操作）之前，微端口将表示发送的用户数据总字节数的 ULONG 值（大 TCP 包负载的 per-packet 信息）写入从大 TCP 包分出的所有包中。

9.8 卸载组合

对特定包而言，TCP/IP 传输层实际上将哪些任务推卸给了 NIC 依赖于 NIC 的微端口报告的卸载能力以及 IP 安全策略在系统中是否有效。

- 如果在系统中没有有效的 IP 安全策略，TCP/IP 栈将卸载校验和任务和/或大 TCP 包分段任务（如果 NIC 的微端口表示 NIC 支持这样的任务）。
- 如果系统中有一个 IP 安全策略有效，TCP/IP 栈不卸载校验和任务和/或大 TCP 包分段。TCP/IP 栈将卸载 IP 安全任务（如果 NIC 的微端口表示 NIC 支持这样的任务）。
- 如果系统中有一个 IP 安全策略变为有效，TCP/IP 栈立即停止卸载校验和任务和/或大 TCP 包分段并开始卸载 IP 安全任务。
- 如果系统中的 IP 安全策略变为无效，TCP/IP 栈恢复卸载校验和任务和/或大 TCP 包分段。

Windows 2000 的将来版本和服务包可能会对同一包同时支持校验和卸载和 IP 安全卸载。因此，如果 NIC 支持校验和卸载和 IP 安全卸载，它必须能够在同一包上同时执行校验和和 IP 安全操作。

下表总结了在不同的任务卸载能力集合下可以卸载的任务：

卸载能力				**卸载的任务**			
IP 校验和	TCP 校验和	大 TCP 包	IP 安全性	IP 校验和	TCP 校验和	大 TCP 包	IP 安全性
Off	Off	Off	Off	No	No	No	No
Off	Off	Off	**On**	No	No	No	**Yes**[2]
Off	Off	**On**	Off	No	No	**Yes**[1]	No
Off	Off	**On**	**On**	No	No	**Yes**[1]	**Yes**[2]
Off	**On**	Off	Off	No	**Yes**[1]	No	No
Off	**On**	Off	**On**	No	**Yes**[1]	No	**Yes**[2]
Off	**On**	**On**	Off	No	**Yes**[1]	**Yes**[1]	No
Off	**On**	**On**	**On**	No	**Yes**[1]	**Yes**[1]	**Yes**[2]
On	Off	Off	Off	**Yes**	No	No	No
On	Off	Off	**On**	**Yes**[1]	No	No	**Yes**[2]
On	Off	**On**	Off	**Yes**	No	**Yes**[1]	No
On	Off	**On**	**On**	**Yes**[1]	No	**Yes**[1]	**Yes**[2]
On	**On**	Off	Off	**Yes**	**Yes**[1]	No	No
On	**On**	Off	**On**	**Yes**[1]	**Yes**[1]	No	**Yes**[2]
On	**On**	**On**	Off	**Yes**	**Yes**[1]	**Yes**	No
On	**On**	**On**	**On**	**Yes**[1]	**Yes**[1]	**Yes**[1]	**Yes**[2]

1. 如果系统中没有有效的 IP 安全策略，任务被卸载。
2. 对需要 IP 分段的发送包和需要组装 IP 分段的接收包不卸载 IP 安全。只有在 NIC 激活了适当的 IP 安全能力时才卸载 IP 安全。例如，如果系统和对等节点之间的 ISAKMP/Oakley（IKE）协商为 NIC 不支持的安全性联系指定了一个算法组合，则不卸载 IP 安全性。详见 9.6 节。

9.9 使用注册表键值激活和禁止任务卸载

在调试驱动程序的任务卸载功能时，激活或停用任务卸载可能是有用的。有两个注册表键控制此功能：

- HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\TCPIP\Parameters\DisableTaskOffload

  将此键置为 0 会禁止从 TCP/IP 传输层卸载所有任务。将它置为 1 激活所有任务卸载。
- HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Ipsec\EnableOffload

  将此键置为 0 禁止从 TCP/IP 传输层卸载 IP 安全。但不影响对 TCP/IP 校验和任务和大 TCP 包分段任务的卸载。将此键置为 1 激活 IP 安全卸载。

第十章　负载平衡和失效替换

负载平衡和失效替换（LBFO）功能可以提高微端口驱动程序的网络适配器的可靠性和性能。微端口驱动程序可以使用 LBFO 将工作量分布到它的适配器束中，从而平衡工作量。也就是说微端口驱动程序可以使用 LBFO 将任务从主适配器推卸给任何次适配器。例如，假设一个协议驱动程序请求微端口驱动程序发送包，为了平衡包发送工作量，微端口驱动程序可以使用它的主适配器传送一些包，而将另一些包的传送卸载给次适配器。LBFO 功能还可以使微端口驱动程序在主适配器发生故障时用一个次适配器接管包传输和信息请求。

以下部分描述了 LBFO 以及如何实现支持 LBFO 的微端口驱动程序：

- 10.1　关于 LBFO
- 10.2　指定对 LBFO 的支持
- 10.3　在微端口驱动程序上实现 LBFO

关于在微端口驱动程序上添加 LBFO 功能的信息，参见 Passthru driver 的自述文件。这个微端口驱动程序是 Microsoft® Windows® 2000　DDK 中的一个微端口驱动程序示例。

10.1 关于 LBFO

微端口驱动程序可以实现对 LBFO 的支持。这样的微端口驱动程序可以将工作量分布到它的微端口实例束中，以平衡包传输的工作量，并可以在主适配器发生故障时，用一个次适配器接管包传输和信息请求。微端口驱动程序管理的微端口实例束可以包括单端口和多端口 NIC。

NDIS 只向传输层暴露微端口驱动程序的主微端口。如果传输层请求向微端口驱动程序发送包，或请求设置或查询微端口驱动程序的信息，NDIS 将这些请求传递给主微端口。

只有被同一微端口驱动程序初始化的微端口实例才能成为 LBFO 方案的一部分。为了使多个被不同微端口驱动程序初始化的微端口实例成为一个 LBFO 方案的一部分，必须实现一个中间层微端口驱动程序为这些微端口实例提供 LBFO。

次微端口只能使用主微端口的句柄向绑定的传输层指示包。

主微端口处理传输层产生的一切信息请求。然而，NDIS 可以将指定请求发送给次微端口。例如，NIDS 可能会查询所有微端口的介质连接性（media connectivity）。次微端口必须

处理并响应此查询。

在安装时微端口驱动程序必须指定它是否实现了 LBFO。为了指定 LBFO 支持，微端口驱动程序的信息文件（*.INF）必须包含 *BuddleId* 关键字和一个字符串值以标识微端口的适配器束。注册表就是用这些信息配置的。

在初始化期间，微端口驱动程序必须确定正在初始化的微端口实例是否属于一个已存在的束。为了确定微端口是否是束的成员，微端口驱动程序从微端口的注册表键下的 BundleId 关键字中检索一个字符串值。每个微端口实例都应将此 BuddleId 值复制到微端口的内部结构中。

然后微端口应搜索所有已初始化的微端口的内部结构，查找与它有相同束标识（BuddleId）值的微端口。如果没有找到，它就成为束的主微端口。在此微端口是束中第一个被初始化的微端口时，发生上述情况。如果微端口找到了一个与它有相同束标识值的微端口，它应将自己置为主微端口对应的次微端口。微端口驱动程序可以设置多个次微端口，也就是说，一个束内可以有多个次微端口。

微端口驱动程序必须为包传输和信息请求提供适当的 LBFO。以下描述了提供 LBFO 的微端口的例子：

- 微端口驱动程序将包传输和信息请求重新选择给次微端口。
- 如果束的主微端口失效，微端口驱动程序可以从束中将主微端口去除，并提升次微端口担任主要任务。

10.2 指定对 LBFO 的支持

微端口驱动程序在安装时指定它实现 LBFO。为了指定对 LBFO 的支持，微端口驱动程序的信息文件（INF）必须包含一个 BundleId 关键字和一个字符串值（REG_SZ）。此字符串值标识微端口驱动程序的适配器束。束标识符信息还用于配置注册表。

微端口驱动程序的束标识符值必须作为适配器属性暴露给网络应用程序、控制面板应用程序，所以如果用户改变了此属性，NIC 将被停机然后重新启动。用户可以通过改变束标识符属性调整从属关系。

以下是一个微端口的 INF 文件中的 *add-registry-section*，它在 **Ndi\params** 键下添加 **BundleId** 子键，并给 **BundleId** 的 **ParamDesc**（参数说明）设置一个字符串值”Bundle1”。

[a1.params.reg]

HKR，Ndi\params\BundleId，ParamDesc，0，”Bundle1”

向微端口的 INF 文件添加键和值的信息详见 1.2.7 节。指定适配器配置参数和值的信息详见 1.2.7.11 和 1.2.7.12。

10.3 在微端口驱动程序上实现 LBFO

为了支持 LBFO，微端口驱动程序必须能够执行以下操作：

- 10.3.1 初始化微端口束
- 10.3.2 平衡微端口驱动程序的工作量
- 10.3.3 在主微端口失效后提升一个次微端口

10.3.1 初始化微端口束

NDIS 为驱动程序管理的每个网络适配器调用一次微端口驱动程序的 MiniportInitialize 函数。为了支持 LBFO，MiniportInitialize 必须确定被初始化的微端口是否属于一个已存在的束。

为了确定微端口是否是束的成员，微端口首先调用 **NdisOpenConfiguratiom** 函数获得注册表键（存储着微端口的配置参数）的句柄。然后微端口调用 **NdisReadConfiguration** 函数从微端口的注册表键下的 *BundleId* 关键字中检索它的字符串值（REG_SZ）。每个微端口都应将此束标识符复制到它的内部结构中。此内部结构还被认为是一个 *MiniportAdapterContext*。每个微端口还应将它的 *MiniportAdapterHandle* 复制到 *MiniportAdapterContext* 中。NDIS 使用 *MiniportAdapterHandle* 引用相关的微端口。

在微端口检索束标识符后，微端口应搜索所有已初始化微端口的内部结构，查找与它有相同束标识符值的微端口。为了执行此搜索，微端口检查每个微端口的 *MiniportAdapterContext*。微端口驱动程序可以将此微端口置为主微端口或次微端口：

- 此微端口是束中第一个被初始化的微端口。换句话说，此微端口没有查找到与它有相同束标识符值的微端口。在此情况下，此微端口默认成为主微端口。
- 此微端口查找到一个相同的束标识符值。它应调用 **NdisMSetMiniportSecondary** 函数将它自己置为与主微端口关联的次微端口。在此调用中，微端口在 *MiniportAdapterHandle* 参数中传递他自己的句柄，在 *PrimaryMiniportAdapterHandle* 参数中传递已被初始化的主微端口的句柄。微端口从主微端口的 *MiniportAdapterContext* 中检索主微端口的句柄。

一个束中可以存在多个次微端口。在每个实例初始化的过程中，微端口驱动程序可以为多个微端口调用 **NdisMSetMiniportSecondary**，将它置为与主微端口关联的次微端口。

只有在微端口实例潜在的属于一个被同一微端口驱动程序初始化的实例束时，它才能调用 **NdisMSetMiniportSecondary**。在微端口实例调用 **NdisMSetMiniportSecondary** 之前，它不能被认为是束的一部分。只有在成功调用 **NdisMSetMiniportSecondary** 之后，微端口实例才能成为束的一部分。

10.3.2 平衡微端口驱动程序的工作量

先前初始化了多个微端口并使它们成为一个束的成员的微端口驱动程序，可以使用这些微端口平衡驱动程序的工作量。

NDIS 只将主微端口实例暴露给传输层。如果传输层请求向微端口驱动程序发送包或请求查询或设置微端口驱动程序的信息，NDIS 应将这些请求传递给主微端口实例。为了平衡微端口驱动程序的工作量，驱动程序应实现将一些请求卸载给次微端口实例的功能。微端口驱动程序可以通过将这些请求发送给次微端口请求次微端口执行与请求相关的任务。不管实际上是主微端口还是次微端口执行了请求，在完成这些请求时微端口驱动程序只应用主微端口的句柄。

然而，如果 NDIS 向次微端口发送了一个请求，那个次微端口应使用它自己的句柄完成请求。

10.3.3 在主微端口失效后提升一个次微端口

支持 LBFO 的微端口驱动程序可以在主适配器失效时提升一个次微端口担任主要角色。如果微端口驱动程序的主适配器失效，驱动程序可以调用 **NdisMRemoveMiniport** 函数将主微端口从束中去除。然后驱动程序调用 **NdisMPromoteMiniport** 函数将次微端口提升为主微

端口。NDIS 应使用新的微端口作为主微端口将传输层的后续请求传递给微端口驱动程序。

第十一章　快速转发路径

快速转发路径（Fast Forwarding Path，FFP）是网络接口卡（NIC）的一个特性，它可以改善网络的性能。FFP 用来与路由和过滤代码协作，从同一个 NIC（或类似的 NIC）上的一个适配器端口到另一个适配器端口转发和过滤包，而不用通过主机处理器发送包。

协议驱动程序可以将转发和过滤包的任务推卸给支持 FFP 的 NIC。为了卸载这些任务，协议驱动程序请求 NIC 的微端口驱动程序在 NIC 上激活 FFP。NIC 的微端口驱动程序可以设置它的 NIC 以使此 NIC 执行这些任务，一般情况下这些任务是由协议驱动程序的路由和过滤代码执行的。

以下节描述了 FFP 和如何实现支持 FFP 的 NIDS 驱动程序：

- 11.1　关于 FFP
- 11.2　NIDS 中的 FFP 支持
- 11.3　为 IP 转发在微端口实现 FFP

11.1　关于 FFP

NIC 微端口驱动程序可被实现成通过快速路径转发包，而不中断主机处理器。例如，一个 NIC 有两个适配器端口，分别叫 NET1 和 NET2。假设最初的包到达 NET1，NIC 的微端口驱动程序取回包并将它指示给路由代码。然后路由代码检查包，让它通过包过滤代码，并通过微端口驱动程序将它发送到 NET2。NIC 的微端口驱动程序能将发送给 NET2 的包的报头信息同从 NET1 取回的包的报头信息相比较，以确定它们是否相同。利用此路由和比较的方法，微端口驱动程序和它的 NIC 确定是否它们应把后来从 NET1 接收的、与最初包有相同报头信息的包快速转发到 NET2，而不中断主机处理器。

包中的头信息也被称为包标识元组（*packet-identifier tuple*）。包标识元组是唯一地标识包的包元素。

如果 NIC 的微端口驱动程序确定从 NET1 取回的初始包与发送给 NET2 的包相同，微端口驱动程序可以将包的标识元组，连同外出包的硬件地址，添加到 NIC 的快速转发缓存（fast-forward cache，FFC）中。随后，每当有包到达 NET1，NIC 将此包的标识元组与 FFC 中所有的标识元组进行比较。如果发现匹配其中一个，它将快速转发此包。

以下节更详细的描述了 FFP：

- 11.1.1　使用一个 NIC 的 FFP
- 11.1.2　使用多个 NIC 的 FFP
- 11.1.3　IP 转发
- 11.1.4　FFP 和包过滤

11.1.1　使用一个 NIC 的 FFP

图 11.1 描述了使用一个 NIC 的 FFP。

图 11.1　使用一个网络卡的 FFP

在使用一个 NIC 的 FFP 中，第一个包最初通过慢速路径路由；也就是说，NET1 的第一个包通过微端口驱动程序、绑定的协议驱动程序、和路由代码，到达 NET2。然后，与第一个包有相同的报头信息的后续包通过快速路径路由；也就是说，它不通过微端口驱动程序和路由代码直接从 NET1 到 NET2。

11.1.2　使用多个 NIC 的 FFP

NIC 微端口驱动程序可被实现成通过跨越多个相同类型的 NIC 的快速路径转发包。例如，NIC1 有一个适配器端口 NET1，NIC2 有一个适配器端口 NET2。假设最初的包到达 NET1，这些 NIC 的微端口驱动程序取回此包并将它指示给路由代码。同只有一个 NIC 时的方法相同，路由代码检查并将此包传递给 NET2，然后微端口驱动程序进行包比较。这些 NIC 的微端口驱动程序确定发送给 NET2 的包是否与从 NET1 取回的初始包相同。如果两个包相同，微端口驱动程序可以将包的标识元组添加到两个 NIC 共享的 FFC 中。随后，每当有包到达 NET1，NIC1 将此包的标识元组与共享的 FFC 中所有的标识元组进行比较。如果发现一个匹配，它将通过到 NET2 的共享 BUS 快速转发此包，而不中断主机处理器。此 FFP 方案需要微端口驱动程序保持一个多个 NIC 共享的内入包缓存。本节不描述如何实现微端口驱动程序，以在多个 NIC 之间共享内入包缓存。

图 11.2　使用两个或更多网络卡的 FFP

图 11.2 描述了使用多个 NIC 的 FFP。第一个包最初通过慢速路径路由；也就是说，NIC1 上 NET1 的第一个包通过微端口驱动程序、绑定的协议驱动程序、和路由代码，到达 NIC2 上的 NET2。然后，与第一个包有相同的报头信息的后续包通过快速路径路由；也就是说，它不通过微端口驱动程序和路由代码直接从 NET1 到 NET2。

11.1.3　IP 转发

如果包的包标识元组与任何存储在 NIC 的 FFC 中的元组匹配，NIC 可以通过快速路径转发包。

如果到达 NIC 的适配器端口的包是通过使用 IP 协议的网络传输的，则那些包在包标识元组中包含以下信息：

IP 源（SRC）地址	（IP 报头偏移量 12，4 字节）
IP 目标（DEST）地址	（IP 报头偏移量 16，4 字节）
IP 协议类型	（IP 报头偏移量　9，1 字节）

如果到达 NIC 的适配器端口的包是通过使用 IP 高层协议的网络传输的，则那些包可能会在包标识元组中包含追加信息。高层协议包括传输控制协议（TCP）、用户数据报协议（UDP）、和网间控制报文协议（ICMP）等。

如果包使用 TCP/IP 传输，那些包应在包标识元组中包含以下关于 TCP 的追加信息：

TCP 源端口	（TCP 报头偏移量 0，2 字节）
TCP 目标端口	（TCP 报头偏移量 2，2 字节）

如果包使用 UDP/IP 传输，那些包应在包标识元组中包含以下关于 UDP 的追加信息：

UDP 源端口　　（UDP 报头偏移量 0，2 字节）
UDP 目标端口　　（UDP 报头偏移量 2，2 字节）

如果包使用 ICMP/IP 传输，那些包应在包标识元组中包含以下关于 ICMP 的追加信息：

ICMP 类型　　（ICMP 消息偏移量 0，1 字节）
ICMP 编码　　（ICMP 消息偏移量 1，1 字节）

IP 的包标识元组的定义如下：

<IPSRC，IPDEST，IPPROTOCOL>

TCP/IP 的包标识元组的定义如下：

<IPSRC，IPDEST，IPPROTOCOL，[TCPSRC+TCPDEST]>

UDP/IP 的包标识元组的定义如下：

<IPSRC，IPDEST，IPPROTOCOL，[UDPSRC+UDPDEST]>

ICMP/IP 的包标识元组的定义如下：

<IPSRC，IPDEST，IPPROTOCOL,[ICMPTYPE+ICMPCODE]>

11.1.4　FFP 和包过滤

被要求过滤某些包的 NIC 微端口驱动程序，应将这些包指示给绑定协议驱动程序的包过滤逻辑。微端口驱动程序将包传送给过滤逻辑会消除 FFP 的优势，因为 NIC 将不得不中断主机处理器将包发送到路由代码。然而，包标识元组可以包含一些信息，NIC 通过检查这些信息确定是否过滤此包。这样 NIC 只需向绑定的协议驱动程序指示那些包标识元组先前未经过过滤逻辑的包。另外，NIC 可被设为快速转发或滤出带有特定包标识元组的包。

使用包过滤的高层协议驱动程序可以请求在 NIC 的 FFC 中把有特定包标识元组类型的缓存项目类型置为 FFP_DISCARD_PACKET。这样,NIC 就可以执行包过滤，而不中断主机处理器。也就是说，NIC 将丢弃那些有被置为 FFD_DISCARD_PACKET 的包标识元组的包。如果 NIC 发现一个内入包的包标识元组与一个在 FFC 中被置为 FFD_DISCARD_PACKET 的包标识元组匹配，NIC 将立即丢弃此包而不用将它指示给绑定的协议驱动程序。详见 online DDK 中的 Network Drivers Reference，FFP Objects 中的 OID_FFP_DATA。

如果协议驱动程序不使用基于高层协议的活动 IP 过滤，协议驱动程序可以通知 NIC。这些高层驱动程序包括，传输控制协议（TCP）、用户数据报协议（UDP）、和网间控制报文协议（ICMP）等。NIC 一旦接收到这个信息，它就可以全速快速转发每个包，而不必考虑每个包内高层协议报头信息。在这种情况下，NIC 检查的转发信息可以用以下元组表示：

IP 源地址，IP 目标地址，IP 协议

详见 online DDK 中的 Network Drivers Reference，FFP Objects，OID_FFP_SUPPORT 或 OID_FFP_CONTROL。

11.2　NIDS 中的 FFP 支持

为了在 NIC 上支持 FFP，协议驱动程序必须可以执行以下操作：

- 协议驱动程序必须能够获得一个拥有特定适配器端口的 NIC 微端口驱动程序的句柄。协议驱动程序首先调用 **NdisOpenAdapter** 函数得到一个绑定适配器端口的句柄。然后调用 **NdisGetDriverHandle** 函数得到拥有绑定适配器端口的 NIC 微端口驱动程序的句柄。协议驱动程序可以使用 **NdisGetDriverHandle** 函数检查一个特定的微端口驱动程序是否拥有两个或多个被协议驱动程序打开的绑定适配器端口。如果一个协议驱动程序需

要对一个 NIC 微端口驱动程序的所有适配器端口的 FFP 状态进行刷新，它可以只给 NIC 微端口驱动程序发送一个 OID_FFP_FLUSH。

- 协议驱动程序必须能够使用包含 FFP 对象标识符（OID）的请求为支持 FFP 的 NIC 设置或取出操作特性。包含 FFP 对象标识符（OID）的请求，详见 online DDK 中的 Network Drivers Reference 中的 FFP Objects。

为了在 NIC 上支持 FFP，NIC 微端口驱动程序必须可以执行以下操作：

- NIC 微端口驱动程序必须能处理 FFP OID。
- NIC 微端口驱动程序必须能够基于协议驱动程序中的过滤器设置它的 NIC 使它丢弃包。微端口驱动程序通过，在 NIC 的快速转发缓存中，将包含特定包标识元组的包的缓存项目类型置为 FFD_DISCARD_PACKET，使 NIC 丢弃包。
- 如果没有基于高层协议的活动 IP 过滤，NIC 微端口驱动程序必须能够设置它的 NIC 使它忽略每个包的高层协议报头中的附加信息。然后 NIC 就可以全速的快速转发每个接收包。

11.3 为 IP 转发在微端口实现 FFP

本节将描述在 IP 网络中如何在 NIC 微端口驱动程序上实现 FFP。在实现这样的微端口驱动程序时必须要考虑一下项目：

- 如果到达 NIC 的适配器端口的包包括任何选项，微端口驱动程序和驱动程序的 NIC 不应当快速转发此包。此包应被指示给绑定的协议驱动程序。
- 如果微端口驱动程序和驱动程序的 NIC 快速转发特定包，在转发之前必须要将这些包的 IP 报头中的存活时间（time to live，TTL）减 1。由于那些包没有被指示给路由代码，因此没有被路由代码减 1，所以微端口驱动程序和驱动程序的 NIC 必须减少那些包的 TTL。一个包的 IP 报头通过 **IPHeader** 结构描述。TTL 的成员是 **iph_ttl**。**IPHeader** 结构的定义参见 online DDK 中 Network Drivers Reference 中的 FFP Objects 中的 OID_FFP_DATA。
- 如果微端口驱动程序和驱动程序的 NIC 改变了一个包的 **IPHeader** 结构的 **iph_ttl**，它们还必须重新计算包的 IP 报头的校验和。计算完此校验和后必须将它放到 **IPHeader** 结构的 **iph_xsum** 中。微端口驱动程序应使用增量技术计算此校验和。
- 如果微端口驱动程序和驱动程序的 NIC 不能处理一个包，它们不能丢弃或快速转发此包，但必须将此包指示给绑定的协议驱动程序。以下列举了一些微端口的 NIC 不能处理包的原因：
  - **IPHeader** 中的 **iph_xsum** 成员的值不是 0xffff。
  - **IPHeader** 中的 **iph_ttl** 被置为 0。
- 正如在 FFP 和包过滤中描述的，一个协议驱动程序可以使用高层协议信息设置 NIC 使它过滤包而不中断主机处理器。高层协议协议包括，传输控制协议（TCP）、用户数据报协议（UDP）、和网间控制报文协议（ICMP）等。通过 IP 网络传输的包能回分段到达 NIC 的适配器端口。包的报头中的高层协议信息只存在于第一个传输分段中。NIC 应能从到达的包中过滤出第一个分段以及所有的后续分段。在包的第一个分段到达后，NIC 应保持一些状态信息。NIC 将这些状态信息用于确定是否过滤或快速转发包的后续分段。

第十二章　带 WDM 低级接口的微端口驱动程序

本章描述如何实现一个 NDIS-WDM 微端口驱动程序，这样的微端口驱动程序：

- 使用 Windows 驱动程序模型（WDM）低级接口
- 能调用 NDIS 和非 NDIS 函数
- 能初始化用来控制连接到一个特殊总线的远程设备，而且通过总线与远程设备通信的微端口实例

例如，一个控制通用串行总线（USB）或 IEEE 1394 总线的微端口要求一个对上层提供标准的 NDIS 微端口接口，在下层为特殊总线使用类接口的驱动程序。这样的微端口就可以通过发送 I/O 请求包（IRP）给总线或直接给与总线相连接的远程设备，而完成与远程设备的相互通信。

以下章节描述如何用 WDM 低层实现一个微端口驱动程序：

- 一个带 WDM 低级接口的微端口驱动程序范例的图解
- WDM 低层微端口驱动程序的注册函数
- 用 WDM 低层初始化微端口驱动程序
- 发布命令与远程设备通信
- WDM 低层微端口驱动程序的实现要点
- WDM 低层微端口驱动程序的编译标志

12.1　WDM 低层微端口

图 12.1 展示了一个与 USB 驱动程序接口的微端口驱动程序

图 12.1　USB 的微端口驱动程序

12.2　注册 WDM 低层的微端口函数

WDM 低层的微端口驱动程序必须在其 **DriverEntry** 例程中指定明确的入口点函数。然后 **DriverEntry** 调用 **NdisMRegisterMiniport** 函数用 NDIS 库注册这些入口点，这些入口点函数组成了微端口驱动程序的上层，其描述如下：

- *MiniportInitialize*

微端口驱动程序初始化函数建立一个用于输入输出（I/O）操作的微端口实例，“*Initializing a miniport driver with a WDM lower edge*”描述了它必须执行初始化函数，并通过使用 WDM 低层的微端口来建立通信操作。

- *MiniportHalt*

微端口驱动程序的停止（Halt）函数释放资源，停止 I/O 操作。实际上，MiniHalt 将恢复由 MiniportInitialize 为特殊微端口实例所作的所有事情。

- *MiniportReset*

微端口驱动程序的重置（Reset）函数将驱动程序的软件状态重置。

- *MiniportReturnPacket*

微端口驱动程序的返回包（return-packet）函数为随后的接收指示准备返回包。较高层的驱动程序调用这个函数向微端口驱动程序返回包的所有权。

- 微端口驱动程序的发送例程

这些函数通过网络传送协议提供的包。对无连接的微端口驱动程序，这个发送函数是

MiniportSend，对面向连接的微端口驱动程序，这个发送函数是 *MiniportCoSendPackets*。微端口驱动程序同时只能传送单个包描述符。

■ 执行信息请求的微端口驱动程序例程

这些函数处理协议发起的请求，从微端口驱动程序得到信息，或设置微端口驱动程序的信息。无连接的微端口驱动程序为查询或设置信息提供两个独立的函数：*MiniportQueryInformation* 和 *MiniportSetInformation*，而面向连接的微端口驱动程序提供 *MiniportCoRequest* 函数来处理查询和设置两种操作。

WDM 低层微端口驱动程序在其 DriveEntry 或初始化例程中不设置确定的入口点函数。例如，下列入口点函数会以下述原因而不设置：

■ *MiniportISR*，*MiniportHandleInterrupt*，*MiniportEnableInterrupt*，*MiniportDisable*：

这些入口点在 **DriveEntry** 中没有确定。因为微端口驱动程序不从物理 NIC 接收中断，它将不请求这些例程，当包到达微端口驱动程序想要的总线时，特定总线的驱动程序接收这些中断，总线驱动程序然后指示微端口驱动程序。

■ *MiniportTransferData*

这个入口点在 **DriveEntry** 中没有确定。当微端口驱动程序指出数据包时，它整个将它们都通过。高层的驱动程序从来不会调用这些例程来传递保留包的内容，所以它不是必须的。

■ *MiniportAllocateComplete*

这个入口点在 **DriveEntry** 中没有确定。由于微端口驱动程序不需要为物理 NIC 分配内存，以及将内存映射到物理内存。所以也不需要一个完整的例程。

■ *MiniportCheckForHang*

这个入口点在 **DriveEntry** 中没有确定。微端口驱动程序只能依赖于 NDIS，根据它的发送和请求超时以判断其微端口是否挂起，所以这个例程也是不必要的。

■ *MiniportShutdown*

微端口驱动程序在其初始化例程期间并不建立关闭（shut-down）例程，因为微端口实例没有必须关掉的硬件。

12.3 初始化 WDM 低层微端口

微端口驱动程序在被操作系统装载之后，NDIS 调用微端口驱动程序的 *MiniportInitalize* 函数初始化其管理的微端口实例。要通过 WDM 低层的微端口驱动程序通信，微端口驱动程序必须收回建立联系的特殊的信息。在微端口实例的初始化期间，微端口驱动程序必须调用 **NdisMGetDeviceProperty** 函数，以回收其在与微端口实例建立联系（通过 WDM 接口）时请求的设备对象。在这个调用中，微端口驱动程序将处理以 *MiniportAdapterHandle* 参数和回收 DEVICE_OBJECT 结构的缓冲区的形式传递给微端口实例。这些结构代表了微端口（或许也包括微端口驱动程序控制的总线的设备对象）。微端口驱动程序使用回收的设备对象指针创建 IRP，并将它们提交给接口。更多的创建和提交 IRP 的信息，参见《*Kernel-mode Drivers Design Guide*》。

WDM 低层微端口驱动程序必须是一个非串行微端口驱动程序，也就是说，这样的微端口驱动程序管理着它自己内部的发送请求队列，无论它有没有足够的资源，立即传输所接收到的发送包。在微端口实例的初始化期间，非串行微端口驱动程序应当控制 NDIS 不将微端口实例的未决发送请求排入队列。要确定这个特征，微端口驱动程序应在调用 NdisMSetAttributesEx 回收时，设置 NDIS_ATTRIBUTE_DESERIALIZE 标志。带 WDM 低层的微端口驱动程序必须确定微端口是非串行的，因为它是在 NDIS 环境以外接收和发送数据包，也就是说，包是通过总线接口传送和接收的；NDIS 不牵扯到传输过程。

12.4 发布命令与远程设备通信

控制总线的微端口驱动程序对总线接口发布命令以及与远程设备通信。为了给总线接口发布命令，这样的微端口驱动程序使用特殊总线类的 IOCTL_*Xxx* 控制码，发布 IRP_MJ_INTERNAL_DEVICE_CONTROL 中断请求包（IRP）。例如微端口驱动程序对 USB 发布控制码 IOCTL_INTERNAL_USB_SUBMIT_URB，对 IEEE 1394 发布控制码 IOCTL_1394_CLASS。

微端口驱动程序将命令的参数打包成一个与特殊总线相联系的请求块，并且通过中断请求包（IRP）I/O 堆栈位置（IO_STACK_LOCATION）的 **Parameters.Others.Argument1** 数据成员，将指针传递给请求块。请求块的函数代码或者识别出特殊的请求，或者确定操作的类型，请求块的其他成员描述了请求和操作。例如，微端口驱动程序将 IEEE 1394 的命令参数打包为 IEEE1394 请求块（IRB），将 USB 的打包为 USB 请求块（URB）。

更多的关于创建和提交 IRP 的信息，参见《*Kernel-mode Drivers Design Guide*》。更多关于特殊总线的函数代码和请求块格式的信息，参见总线驱动程序接口的文档。

微端口驱动程序能发布命令以发送数据包给远程设备，也可以指示自己远程设备所发送的如下所述的数据包：

- 在总线上发送包
- 在总线上接收包

12.4.1 在总线上发送包

给微端口驱动程序传送包的传输请求导致 NDIS 微端口调用微端口驱动程序的发送例程，这个例程发布一个命令给特定总线或者给与总线相连的远程设备，这个命令了确定了通过网络发送的数据包。

12.4.2 在总线上接收包

微端口驱动程序为了能接收到通过网络传输过来的包，必须给特定总线发布命令。这个命令确定总线在接收到要传给微端口驱动程序的数据包后，指示微端口驱动程序。微端口驱动程序被指示后，它从传输中指示出那些数据包。为了指示数据包，无连接的微端口驱动程序需要调用 **NdisMIndicateReceivePacket** 函数，面向连接的微端口驱动程序需要调用 **NdisMCoIndicateReceivePacket** 函数。

12.5 WDM 低层的实现要点

这一节讨论用 WDM 低层接口实现一个 NDIS-WDM 微端口驱动程序的要点。这样的一个微端口驱动程序能调用 NDIS 和非 NDIS 两种函数。这些非 NDIS 函数包括，WDM 内核模式支持的特定总线驱动接口的函数和例程。NDIS-WDM 微端口驱动程序仅限于使用 WDM 内核模式支持的例程（它只是 Microsoft Windows 2000 内核模式支持例程的一个子集）。要得到更多的 WDM 内核模式支持例程的信息，参见《*Windows 2000 Drivers Development Reference*》的第二卷，在这一卷中将 WDM 内核模式支持的例程等同于 *wdm.h* 头文件中所描述的内核模式例程，如果内核模式例程只出现在 *ntddk.h* 头文件中，它不是一个 WDM 内核模式支持的例程。

当实现一个 NDIS-WDM 微端口驱动程序时，必须考虑到下列问题：

- 构造一个 NDIS-WDM 微端口驱动程序要求在包括（Include）*ndis.h* 头文件以前，NDIS_WDM 标志已被定义。NDIS_WDM 标志的定义将导致自动包括 *wdm.h* 头文件。NDIS_WDM 标志应当嵌入微端口驱动程序源代码的开始，或者放在微端口驱

动程序的源（source）文件中。NDIS_WDM 微端口驱动程序需要 *wdm.h* 来调用象 **IoCallDriver** 和 **IoAllocateIrp** 这样的内核模式的函数。

- 特定总线驱动接口的函数调用需要总线驱动程序的头文件。
- 建议不要将 NDIS 和非 NDIS 头文件包括（include）进同一个源文件，因为它们不能很好地在一起工作，也就是说，调用 NDIS 函数的代码应当被创建为一个独立的源文件，调用非 NDIS 的也一样。
- NDIS_WDM 微端口驱动程序应当调用适当的 NDIS 函数分配和释放资源，除非为下列之一的情况：
  - 资源，典型的如内存资源，由 NDIS_WDM 微端口驱动程序分配，后来被一个非 NDIS 实体如总线驱动程序接口释放。
  - 资源，典型的如内存资源，由非 NDIS 实体分配，后来被一个 NDIS-WDM 微端口驱动程序释放。

对上述情形，NDIS_WDM 微端口驱动程序将调用适当的 WDM 函数为非 NDIS 实体分配或释放资源。

12.6 WDM 低层的编译标志

本节列出了为建造 NDIS-WDM 微端口驱动程序而必须包括进源文件的编译标志，下列编译标志必须被包括：

- -DNDIS_WDM=1

指导 NDIS 包含 *wdm.h* 头文件。

- -DUSE_KLOCKS=1

指导 NDIS 为自旋锁（spin lock）使用 KSPIN_LOCK 类型而不是 NDIS_SPIN_LOCK 的变量，使用 WDM 内核模式函数初始化、分配和释放这些自旋锁。

作为一种选择，编译标志能在 *ndis.h* 头文件被包括之前，嵌入到微端口驱动程序源代码的开始。

第十三章　IrDA 微端口 NIC 驱动程序

红外数据协会（the Infrared Data Association：IrDA）已经制定了一种用红外线脉冲对数据编码的半双工协议。制造红外收发器用来传输和接收这些红外线脉冲，这些红外收发器由光发射二极管（LED）和接收二极管组成。接收二极管对一定距离内红外 LED 发出的红外线传播很敏感，发射 LED 必须向同样距离之外的远程接收二极管发送红外线脉冲。

红外适配器是一种由支持硬件对帧进行编码和解码的红外收发器组成的网络接口卡（NIC）。IrDA 微端口 NIC 驱动程序是实现用来控制红外 NIC。

下面的章节描述 IrDA 微端口 NIC 驱动程序的体系结构、IrDA 微端口驱动程序的特征以及 IrDA 微端口驱动程序所必须支持的 IrDA 介质种类的帧格式：

- 13.1　IrDA 微端口驱动程序简述
- 13.2　IrDA 体系结构
- 13.3　IrDA 协议驱动程序
- 13.4　IrDA 介质特征
- 13.5　IrDA 帧格式
- 13.6　IrDA 微端口驱动程序包编码方案
- 13.7　接收和发送包序列
- 13.8　即插即用

13.1　IrDA 微端口驱动程序简述

IrDA 硬件（红外收发器）和核心 IrDA 协议栈之间的接口是通过网络驱动程序接口规范（NDIS）和 IrDA 微端口驱动程序来实现的，IrDA 协议栈期待 IrDA 硬件和 IrDA 微端口驱动程序来处理编帧、透明、错误检测、支持感知介质行为的命令以及改变速度。IrDA 微端口驱动程序负责丢弃坏的循环冗余校验（CRC）的接收包，这些帧永远不会提交到 IrDA 协议栈。

操作系统提供了一个核心 IrDA 协议栈的实现，在 IrDA 微端口驱动程序安装时，IrDA 协议栈被即插即用的系统动态地载入。IrDA 协议栈的 IrDA 小传输协议（TinyTP）层通过支持用户空间部件 *wshirda.dll* 的 Windows 套接字（WinSock）来支持应用程序。应用程序只能使用 Winsock 与 IrDA 微端口驱动程序通信。关于 WinSock 的更多信息，参见 platom SDK 文档。

为达到更多的用途，操作系统将 IrDA 微端口驱动程序看作只绑定于 IrDA 协议栈上的 NIC 微端口驱动程序。IrDA 协安装程序将这些驱动程序（象 IrDA 适配器、处理特殊 IrDA 用户接口和设置等）问题显示给用户。

不象其他介质的 NDIS 驱动程序，IrDA 微端口驱动程序不能同时绑定一个以上的协议，而且它总是绑定到同一个协议。这是因为，不象 NDIS 其他介质，IrDA 协议栈只实现了介质控制算法的一部分，这种算法排除了多个控制器。

系统提供的和厂商提供的 INF 文件用来安装 IrDA 微端口驱动程序和 IrDA 协议栈，这些 INF 文件必须符合 Net INF 和 IrDA INF 协定。

操作系统支持的一个 IrDA 微端口驱动程序：*irsir.sys*，是用于外部连接 SIR 适配器的通用异步收发器（UART）和内部 SIR IrDA 适配器间的通信的。厂商应使它们的 SIR 硬件正确地被 *irsir.sys* 支持。

高质量的 IrDA 微端口驱动程序和 INF 文件能被操作系统包含在内。应当鼓励这种习惯，

因为它改进了用户的 IrDA 安装经验。

IrDA 协议栈能同时支持多个 IrDA 网络接口卡（NIC），希望开发这种特征的厂商应当通过其 IrDA 微端口驱动程序来支持多个 IrDA NIC。每一个 NIC 应代表一个独立的收发器和支撑硬件——能进行独立的红外连接。

在为多种 IrDA 硬件配置实现 IrDA 微端口驱动程序时，有多个即插即用的问题应当被考虑到。13.8 节，“即插即用”，将讨论这些问题。

13.2 IrDA 体系结构

图 13.1 显示了 IrDA 协议层的组成，以及 IrDA 微端口驱动程序是怎样通过 NDIS 库与其 NIC 及应用程序通信的。

图 13.1 IrDA 体系结构

13.3 IrDA 协议驱动程序

IrDA 协议驱动程序，*irda.sys*，是打开 IrDA 微端口驱动程序的唯一的协议驱动程序。这个协议驱动程序是 IrDA 红外连接访问协议（IrLAP）的一个实现，协议驱动程序负责检测红外设备、在本地和远程设备之间建立连接以及可靠地在本地和远程设备之间传送数据。IrDA 协议驱动程序使用 IrDA 微端口驱动程序，在本地和远程设备之间收发数据。

IrDA 协议驱动程序的服务类似于 TCP/IP 协议表现出来的服务。使用这些服务，运行于两个不同计算机上的应用程序能很容易地打开多个用于发送和接收数据的可靠连接。IrDA 协议驱动程序提供下列服务：

- 小传输协议（TinyTP）是通过 Winsock 提供给应用程序的。应用程序能使用 TinyTP 建立连接和收发数据。
- 红外连接管理协议（IrLMP）为 IrLAP 增加多个会话支持。多个应用程序能使用 IrLMP 来监视相互没有接口的连接。单个应用程序也能使用 IrLMP 同时打开一个控制连接和一个数据连接。IrLMP 也能为单个 IrDA 连接提供的流控制增加每个连接流的控制。IrLMP 使得应用程序能得到传给 IrDA 协议栈的大块数据，IrDA 协议栈然后将这些数据以优化的速度送出。使用 IrLMP，应用程序不需要监视丢失的数据或者进行流控制。
- 信息访问服务（IAS）直接在 IrLMP 的顶部运行。IAS 将一个 ASCII 服务名映射为一个 *LSAP-SEL*。*LSAP-SEL* 是用于从运行于服务器上多个可能的应用程序中挑选出一个应用程序的协议元素。这个协议的名字是提供给应用程序的一个友好的抽象。
- 红外连接打印传输（IrLPT）与打印机通信。TinyTP 不用于 IrLPT。IrLPT 使用一种只支持单个 IrLMP 连接的专有模式，这个单个 IrLMP 连接使用了 IrLAP 的流控制特性。IrLMP 定义了这种专有模式的服务。

IrDA 协议驱动程序绑定到了所有支持 **NdisMediumIrDA** 介质类型的 IrDA 微端口驱动程序。IrDA 微端口驱动程序只绑定到 IrDA 协议驱动程序，因为，不象其他 NDIS 介质，IrDA IrDA 协议层实现了介质控制算法的一部分，这种算法排除了多个控制器。

13.4 IrDA 介质特征

与早期的 NDIS 介质比起来，通过 IrDA 介质控制和通信，**NdisMediumIrDA** 会涉及到一些独特的问题。IrDA 微端口驱动程序解决了这些问题。下列各节将描述红外介质的特征：

- 13.4.1 通信连接速度
- 13.4.2 通信连接回转时间
- 13.4.3 接收器同步

13.4.1 通信连接速度

不象典型的 NDIS 介质，红外介质支持许多不同的传输和接收比特的速度。当前定义的操作速度从 2 400 比特每秒（bps）直到 4 M 比特每秒（Mbps）。在将来，IrDA 将会定义更高的速度。各种不同速度的设计目的导致了三种不同的帧编码方式：SIR，MIR 和 FIR。这些不同的帧编码方式必须都由 IrDA 微端口驱动程序处理，然后传给协议。

当前定义的 IrDA 速度及帧编码方式列表如下：

速度（bps）	帧编码方式
2 400	SIR
9 600	SIR
19 200	SIR
38 400	SIR
57 600	SIR
115 200	SIR
576 000	MIR
1.152 Mbps	MIR
4 Mbps	FIR

关于帧编码方式的定义，参见 13.6 节，“IrDA 微端口驱动程序包编码方案”。

特定的红外适配器不需要支持上表列出的所有速度。然而，对于两个相互通信的 IrDA 设备，在支持的速度上必须有重叠部分。IrDA 标准规定所有的 IrDA 设备必须支持 9 600 bps 的速率。这样的 IrDA 规范保证了所有的 IrDA 设备至少能以 9 600bps 通信。通过以 9 600bps 速率通信，IrDA 设备可能协商达到一个更高的速度，例如，两个 IrDA 设备必须以 9 600 bps 的速度协商，但如果两个设备都支持 115.2 k 比特每秒（Kbps），它们就会以这个较高的速度来传输数据。注意在很少的情况下，两个 IrDA 设备以 9 600 bps 协商，但两个 IrDA 设备共享的通信速度是唯一的 2 400bps 的低速。

IrLAP 协议能通过给驱动程序发送一个 OID_IRDA_SUPPORTED_SPEEDS 查询以检测 IrDA 微端口驱动程序支持什么速度。

IrLAP 协议在传输包以前，使用 OID_IRDA_LINK_SPEED 和 OID_IRDA_SUPPORTED_SPEEDS 的返回值来设置 IrDA 微端口驱动程序的连接速度。连接速度指红外适配器向外传输帧的速度以及可能接收传入帧的速度。关于怎样设置 OID_IRDA_ LINK_SPEED 来控制传入帧的速度，参见 OID_IRDA_RATE_SNIFF 的描述信息。

13.4.2 通信连接回转时间

红外适配器由支持硬件对帧编码和解码的红外收发器组成。红外收发器包含许多靠得很近的发射 LED 和接收二极管。接收二极管对红外线相当敏感，因为它必须接收到至少一米之外的远程红外 LED 传输来的信号。发射 LED 必须是大功率的，因为它必须向同样距离之外的远程接收二极管传输信息。

在传输期间，本地的 LED 必须发射足以使本地接收二极管饱和的光线。就象人们盯着太阳看时看不清楚一样，本地 LED 向外发出传输帧之后，本地接收二极管很难立即正确地接收到传过来的帧。

本地 LED 接收二极管从饱和状态变得能再接收来帧所允许的时间，即为 IrDA 协议定

义的回转时间。回转时间确定了以毫秒计的接收二极管从饱和状态恢复的时间。对一些 IrDA 设备，回转时间可以忽略不计，有些 IrDA 设备则是相对较长的一段时间。

本地接收二极管的回转时间将不会影响本地收发器的行为。然而本地接收二极管的回转时间会影响远程收发器的期望行为。例如，本地收发器从其 LED 完成传输到其接收二极管能接收信号需要 1 毫秒的时间延迟，在开始传输一个新帧之前，远程的工作站在接收到上一帧的最后一位之后就必须等待 1 毫秒的时间，远程工作站执行这个等待以兑现本地收发器的回转时间。

IrDA 微端口驱动程序必须被告知其本地收发器的回转时间。IrDA 微端口驱动程序使用 OID_IRDA_TURNAROUND_TIME 来确定这个时间值，IrLAP 能在任何时候给 IrDA 微端口驱动程序发送 OID_IRDA_TRUNAROUND_TIME 查询来得到本地收发器的回转时间。

考虑到远程收发器的回转时间，IrDA 协议有时会规定延迟包的传输，如果要这样做，IrLAP 协议须确定包传输后的时间长短。IrDA 微端口驱动程序在等待请求的时间数之前不必传输数据包，尽管驱动程序可以等待较长的时间。IrDLAP 协议在数据包的带外（OOB）数据块的介质相关成员中规定了包的传输延迟时间。关于更多的信息，参见 13.7 节，“发送和接收帧序列”。

13.4.3 接收器同步

一些红外收发器使用自己红外信号中的信息来与信号同步，以便能接收到传过来的帧。由于红外收发器使用不同的硬件，这些收发器同步到它们能可靠地接收数据的位置也需要不同的时间。典型的情况，红外收发器在接收到位于每一个传入包起始的帧开始（BOF）标志时与红外信号同步。由于一些硬件需求，红外收发器也许需要一个或多个 BOF 标志以完成接收器同步。

IrDA 微端口驱动程序必须一直持有（其收发器在接收的红外信号中请求同步时所使用的）额外的 BOF 标志数目的信息。IrDA 微端口驱动程序使用 OID_IRDA_EXTRA_RCV_BOFS 来确定请求额外的 BOF 标志。IrLAP 可以在任何时候给 IrDA 微端口驱动程序发送一个 OID_IRDA_EXTRA_RCV_BOFS 来得到本地收发器的额外 BOF 标志的数目。IrDA 微端口驱动程序也必须被告知远程接收工作站所请求的额外 BOF 标志。在 IrLAP 与远程接收工作站协商连接速度并设置 IrDA 微端口驱动程序的连接速度之后，IrDA 微端口驱动程序必须使用下列换算表来检测额外 BOF 标志的请求数目。

在 OID_IRDA_EXTRA_RCV_BOFS 中规定的值应用于 115.2 Kbps 的串行红外（SIR）连接速度，其他 SIR 连接速度的值由此值换算而来。IrDA 微端口驱动程序使用下列换算表为那些驱动程序从连接速度中确定所需要的额外 BOF 标志。

速度	请求的额外 BOF 标志							
2 400	1	0	0	0	0	0	0	0
9 600	4	2	1	0	0	0	0	0
19 200	8	4	2	1	0	0	0	0
38 400	16	8	4	2	1	0	0	0
57 600	24	12	6	3	1	1	0	0
115 200	48	24	12	6	3	2	1	0

为确定 OID_IRDA_EXTRA_RCV_BOFS 的值，IrDA 微端口驱动程序应当指示 IrDA 协议驱动程序：

1. 在最左一列中找到其 NIC 收发器的连接速度；
2. 在这一行，找出收发器以这种速度工作时的额外 BOF 标志的请求数目。
3. 往下找到 115.2 Kbps 行的那一列，换算表为 115.2Kbps 规定的额外 BOF 标志数就是微端口驱动程序应当指示给 IrDA 协议驱动程序的额外 BOF 标志数。

下面的例子展示了 IrDA 微端口驱动程序怎样使用上述换算表确定适当的 OID_IRDA_EXTRA_RCV_BOFS 值的：

- 以 2 400bps 速度传输，如果 NIC 的收发器为了同步请求 1 个额外 BOF 标志，IrDA 微端口驱动程序必须将 OID_IRDA_EXTRA_RCV_BOFS 的值确定为 48；
- 以 57.6 Kbps 速度传输，如果 NIC 的收发器为了同步请求 8 个额外 BOF 标志，IrDA 微端口驱动程序必须将 OID_IRDA_EXTRA_RCV_BOFS 的值确定为 24，因为 8 处于表中的 6 和 12 之间，IrDA 微端口驱动程序使用较大的 12 来确定 OID_IRDA_EXTRA_RCV_BOFS 的值。
- 以 115.2 Kbps 速度传输，如果 NIC 的收发器为了同步请求 1 个额外 BOF 标志，IrDA 微端口驱动程序必须将 OID_IRDA_EXTRA_RCV_BOFS 的值确定为 1；

注意 MIR 和 FIR 红外收发器的连接速度必须以此速度下 BOF 标志准确的数目来工作。

为了兑现远程工作站所要求的额外 BOF 标志数，IrLAP 必须在送往 IrDA 微端口驱动程序传输的每一个数据包中确定额外 BOF 标志的数目，这个值总是为当前连接速度下的额外 BOF 标志数。因为 IrLAP 在每个包中确定了额外 BOF 标志数，IrDA 微端口驱动程序不需要在每次准备发送包之前使用上述的换算表。IrDLAP 协议在数据包的带外（out-of-band OOB）数据块的介质相关成员中规定了额外 BOF 标志数目。关于更多的信息，参见 13.7 节，“发送和接收帧序列”。

13.5 IrLAP 帧格式

IrDA 协议栈的协议最低层指的是红外连接访问协议（IrLAP）。IrLAP 定义了红外介质上发送和接收的帧的格式。下列各节将描述 IrLAP 帧格式的内容：

- 13.5.1 帧格式简述
- 13.5.2 帧信息的使用
- 13.5.3 地址成员

13.5.1 帧格式简述

IrLAP 定义了红外介质上发送和接收的帧的格式。每一个 IrLAP 帧由下列元素组成：

- 一个或多个表示帧开始的帧起始（BOF）标志。BOF 成员长度大小依赖于速度而不同，就象在“接收器同步”节描述的那样。
- 识别第二个联系地址的地址（A）成员。地址成员为 8 位，地址成员确定了属于特定 IrDA 微端口驱动程序设备的地址，IrDA 微端口驱动程序通过这个设备传输或接收包含其地址的帧。更多的地址成员的信息，参见 13.5.3 节“地址成员”。
- 确定特殊帧功能的控制（C）成员。控制成员为 8 位。
- 包含信息数据的可选信息（I）成员。信息成员是八位字节的整数。
- 帧检查序列（FCS）允许接收站检查帧的传输准确性。FCS 依赖于速度为 16 位或 32 位。
- 帧结束（EOF）标志表示帧的结束。EOF 成员的大小随随速度变化而不同。

下列 IrLAP 帧的例子显示了上述元素的顺序：

BOF	A	C	I	FCS	EOF

包的编码和 BOF、EOF 以及 FCS 成员随红外介质的操作速度而变化。关于物理帧的完整文档能在 IrDA 的发布的《*Infrared Data Association Serial Physical Layer Link Specification*》中找到。13.6 节 “IrDA 微端口驱动程序包编码方案”提供了各种不同编码方案的简介。

13.5.2 帧信息的使用

IrDA 协议驱动程序调用 NdisSendPackets 函数发送一个包描述符序列（NDIS_PACKET 结构）到下层 IrDA 微端口驱动程序。这些包描述符只给出了帧从“地址元素”到“信息元素”的信息，IrDA 在传输此帧之前必须给它加上其他所有的帧元素。同样地，IrDA 微端口驱动程序在将包指示给 IrDA 协议驱动程序之前，必须将接收包的“地址元素”到“信息元素”之外的所有元素剥离。

IrDA 微端口驱动程序必须进行一个用来转换传递到红外介质上的包的编码操作，同样的，IrDA 微端口驱动程序也必须进行解码操作，将从红外介质上接收的包转换为其原始形式。

如果从红外介质上接收的包是无效的，IrDA 微端口驱动程序将不不把它提交给 IrDA 协议驱动程序。一个包可以因为下列原因而变得无效：

- 包不完整
- 包的循环冗余校验（CRC）无效
- 包包含一个拒绝序列

13.5.3 地址成员

下列例子显示了 IrLAP 帧中的地址成员的 8 比特信息：

C/R	A	A	A	A	A	A	A

地址成员的第一位用于命令与响应（C/R）。如果这一位被设置了，表示这个包从主站送到从站。如果 C/R 位被清除，表示包从从站送往主站。因为 IrDA 微端口驱动程序不需要 C/R 位的信息，它并不监视这一位。注意主站启动了到从站的连接，并运行使这些连接有效的时钟。

地址成员中剩下第 7 个信号位表示连接访问协议（LAP）的地址，LAP 的地址是一个属于特殊 IrDA 微端口驱动程序的设备地址。IrDA 微端口驱动程序通过这个设备传送和接收数据包。

IrLAP 协议能指定一个能接收包的设备地址清单，从而限制所有其他的设备接收数据包。为了指定这个清单，IrLAP 协议请求 IrDA 微端口驱动程序用 OID_IRDA_UNICAST_LIST 在 UNICAST 列表中设置地址，UNICAST 列表是一个 IrDA 微端口驱动程序能通过它接收数据包的设备地址的数组。如果 UNICAST 列表设置了，IrDA 微端口驱动程序将只能从指定地址接收有效的红外数据帧；如果 UNICAST 列表没有设置或包含 0 元素，IrDA 微端口驱动程序将必须接收所有有效的红外帧，然后将数据包提交给 IrLAP 协议。

13.6 IrDA 微端口驱动程序包编码方案

实际在红外介质上传输的包的格式是随着包传输和接收的速度不同而变化的，IrDA 微端口驱动程序形成包的方式依赖于其红外 NIC 的硬件实现。下列各节将简要描述不同类型的红外 NIC 的帧编码的方法：

- 13.6.1 SIR 编码
- 13.6.2 MIR 编码
- 13.6.3 FIR 编码

如果红外 NIC 的硬件允许开发者进行红外介质的编码，他们能得到出版物：《*Infrared Data Association Serial Physical Layer Link Specification*》和《*Infrared Data Association Serial Infrared Link Access Protocol* （*IrLAP*）》，这些出版物详细地解释 IrDA 提供的所有类型的红外 NIC 帧的编码。

13.6.1 SIR 编码

本节描述 IrDA 微端口驱动程序或者其红外 NIC 怎样对串行 IrDA（SIR）连接速度下的传输帧进行编码。SIR 规范用 1 个起始位、8 个数据位和 1 个结束位定义了一个小范围的红外异步串行传输模式。最大的数据传输率为 115.2 Kbps（半双工）。以 SIR 对帧进行编码的最大益处是已存在的串行硬件非常便宜，使用串行硬件的低花费是红外 SIR 设备广泛应用的一个原因。

SIR 速度的 BOF 标志定义为 0xC0，EOF 的值定义为 0xC1。为了避免包含 BOF 和 EOF 的帧含糊不清，为帧中的其他部分出现的 0xC0 和 0xC1 定义了一个转义序列，转义字符定义为 0x7D。

对遇到的每一个象 BOF、EOF 或转义序列这样的字符，发送器将执行下列步骤：

1. 在每一个这样的字节之前插入一个控制序列字节（0x7D）

2. 对每一个象 BOF、EOF 或转义序列这样的字符的第五位求反，也就是用 0x20 对其进行一个或（OR）运算。

在驱动程序插入任何一个控制序列字节之前，IrDA 微端口驱动程序计算出帧的 FCS 成员。

在 FCS 测定之前，接收站也将测试 BOF 和 EOF 之间的整个帧环境。接收站对遇到的每一个转义字节执行下列步骤：

1. 丢弃转义字节

2. 对转义字节以后字节的第五位求反

IrDA 微端口驱动程序用地址（A）、控制（C）和信息（I）成员来计算帧的 FCS 序列。对 SIR 连接速度，具有代表性的，是用 IrDA 微端口驱动程序而不是驱动程序的硬件来计算 CRC 值。计算 16 位 CRC 的算法可参见 IrDA 发布的《*Infrared Data Association Serial Physical Layer Link Specification*》和《*The Internet Working Group Request For Comments*（*RFC*）1171》。

对 SIR 连接速度，IrDA 微端口驱动程序也能在包中编入一个拒绝序列。拒绝序列指示红外收发器接收到这样一个包后丢弃它，拒绝序列为 0x7D 0xC1。

13.6.2 MIR 编码

本节讨论 IrDA 微端口驱动程序或者其红外 NIC 对以中间 IrDA（MIR）连接速度传输的帧的编码。MIR 速率为 0.576 Mbps 和 1.152 Mbps（半双工）。

对 MIR 连接速度，BOF 和 EOF 的定义是相同的，都定义为 0x7E。为了避免带 BOF 和 EOF 的帧含糊不清，MIR 速率在 BOF 和 EOF 之外的所有成员中每 5 个连续的“1”之后插入 1 个“0”，而不是象 SIR 那样使用转义序列。因为在比特级插入和去除“0”的过程是很消耗处理器的，所以强烈推荐在硬件中实现这个逻辑。在 MIR 连接速度中，每一个帧需要两个 BOF 标志。

对 MIR 连接速度，象 SIR 速度一样也使用了 CRC，也就是说，对 MIR 连接速度，是 IrDA 微端口驱动程序而不是驱动程序的 NIC 来计算 CRC 值。

13.6.3 FIR 编码

本节讨论 IrDA 微端口驱动程序或者其红外 NIC 对以快 IrDA（FIR）连接速度传输的帧的编码。FIR 规范定义了一个 4Mbps 下的小范围低能耗的操作（半双工）。所有的 FIR 设备都需要支持 SIR 操作。

对 FIR 连接速度，使用了一个完全不同的编码方案——称为脉冲位置调制（PPM）。PPM 为 BOF 和 EOF 定义了一个特殊的标志。PPM 编码方案应当在硬件实现。

IrDA 微端口驱动程序也需要计算 CRC 以检验帧的有效性。对 FIR 连接速度，使用了一

个 32 位的 CRC。计算 32 位 CRC 的算法在 IrDA 出版的《*Infrared Data Association Serial Physical Layer Link Specification*》中提供。

13.7 发送和接收帧序列

IrDA 微端口驱动程序应当支持以下所描述的发送和接收帧序列。

IrDA 微端口驱动程序应当实现一个 *MiniportSendPackets* 函数来通过网络传输帧的序列。IrLAP 调用 **NdisSendPackets** 函数发送一个帧标识符的序列给下层的 IrDA 微端口驱动程序。每一个帧标识符都关联着带外（OOB）信息。IrDA 微端口驱动程序能从 OOB 数据块回收和使用（NDIS_PACKET_OOB_DATA）指定给红外介质的信息。

包从远程站传过来后，NIC 的收发器接收这些数据包，然后发出中断给 NDIS，再依次调用特殊的例程处理这些中断。NIC 的 IrDA 微端口驱动程序应当实现 *MiniportIsr* 和 *MiniportHandleInterrupt* 函数为接收到的包处理中断。为将数据包指示给 IrDA 微端口驱动程序，IrDA 微端口驱动程序为每一个包设置（已在 OOB 数据块中指定的）红外介质的信息，然后调用 **NdisMIndicateReceivePacket** 函数。

IrDA 微端口驱动程序应当使用 NDIS 提供的设置和回收 OOB 数据的宏。关于 OOB 数据和访问 OOB 数据的信息，参见第 4 章 4.3 节“带外（OOB）数据包”。

OOB 数据块的 **MediaSpecificInformation** 成员指向一个 MEDIA_SPECIFIC_INFORMATION 类型的结构，而这种结构有一个为 NDIS_IRDA_PACKET_INFO 类型结构的 **ClassInformation** 成员。NDIS_IRDA_PACKET_INFO 定义如下：

```
typedef struct _NDIS_IRDA_PACKET_INFO{
    UINT ExtraBOFs;
    UINT MinTurnAroundTime;
}NDIS_IRDA_PACKET_INFO,*PNDIS_IRDA_PACKET_INFO;
```

ExtraBOFs 成员表示增加到传输包中的额外帧起始（BOF）数目，关于更多的信息，参见 13.4.3 节“接收器同步”。

MinTurnAroundTime 表示帧的最小延迟时间，关于更多的信息，参见 13.4.2 节“通信连接回转时间”。

13.8 即插即用

当为各种各样硬件配置的 NIC 红外收发器实现 IrDA 微端口驱动程序时，应该考虑到几个即插即用问题。这些配置包括，支持即插即用、所连接系统的类型以及连接速度。下列章节将描述这些问题：

- 13.8.1 非即插即用外部串行连接 SIR 适配器
- 13.8.2 非即插即用内部 SIR 适配器或者象串口一样错误地呈现于外的内部 SIR 适配器
- 13.8.3 即插即用外部串行连接 SIR 适配器
- 13.8.4 即插即用内部 SIR 适配器
- 13.8.5 非即插即用总线连接 FIR 适配器
- 13.8.6 即插即用总线连接 FIR 适配器

13.8.1 非即插即用外部串行连接 SIR 适配器

连接到外部串口并且不支持任何即插即用功能的红外 SIR 设备由操作系统提供的 IrDA 微端口驱动程序 *irsir.sys* 来支持。系统提供的 IrDA 微端口驱动程序使用串口驱动程序提供的服务来访问串口。串口驱动程序可以是操作系统提供的一个 *serial.sys*，或者是一个兼容的

第三方串口驱动程序。

外部串口在“控制面板/设备管理”列表的“端口（*COM&LPT*）”单元中对用户可见。系统提供的 IrDA 微端口驱动程序忽略了串口的特殊设置。所以，用户不能正确地配置串口以至于连接到外部串口的红外设备不能工作。

用户应当通过“硬件向导（Wizard）”的“增添新硬件”功能手工安装这些设备。用户应当选择：

1. IrDA 适配器设备类型
2. 从支持适配器类型的列表中找到的 IrDA 适配器设备
3. 适配器连接的串口，串口通过其名字来选择

系统提供的 IrDA 微端口驱动程序打开用户通过其名字而选择的串口。请求支持这个端口的资源是不可见的，用户可以通过 IrDA 适配器的用户接口来配置它。用户只能配置串口名。

需要操作系统支持硬件厂商应当请求这样的支持、提供特定设备初始化和速度变化的程序。

13.8.2 非即插即用内部 SIR 适配器或者象串口一样错误地呈现于外的内部 SIR 适配器

内部连接的和任何非即插即用功能的红外 SIR 设备不象即插即用的 SIR 设备那样通过系统 BIOS 呈现于外。这些设备被检测到以后再进行安装，或者它们被象常规串口那样手工安装。

如果红外 SIR 设备的 IrDA UART 只象常规的串行设备那样通过 BIOS 呈现于外，这个设备就象这样安装。在这种情况下，用户的感觉就象在《*Non-Plug ang Play External Serial-Attached SIR Adapters*》中描述的那样。

推荐不要使用这种配置。

13.8.3 即插即用外部串行连接 SIR 适配器

当前没有连接到外部串口的红外 SIR 设备支持串行计算，所以不支持这种配置。

13.8.4 即插即用内部 SIR 适配器

内部连接的和支持任何即插即用功能的红外 SIR 设备，象即插即用的 SIR 设备那样通过系统 BIOS（PNP0510 或 PNP0511）呈现于外。这些设备象常规的 UART 一样呈现给 IrDA 微端口驱动程序，这是由操作系统的 *irsir.sys* 和一个串口驱动程序提供的。这个串口驱动程序可以是操作系统提供的一个 *serial.sys*，或者是一个兼容的第三方串口驱动程序。

这些设备并不在“控制面板/设备管理”列表的“端口（*COM&LPT*）”单元可见。因为它们象串行设备那样并不是通过 BIOS 呈现出来，这就意味着在这个端口上串口驱动程序不能自动调入，另外，这些设备象常规串口一样对用户和应用程序不可见。

一旦安装了这些设备通过即插即用，它们就在“设备管理器”列表的“红外适配器”单元中呈现给用户，而且，安装了系统提供的 IrDA 微端口驱动程序，它使用下层 IrDA UART 描述的资源。为了使用 IrDA UART 资源，系统象低层过滤驱动程序一样通过 IrDA 微端口驱动程序载入串口驱动程序。

SIR IrDA UART 请求的资源必须以常规即插即用的方式呈现出来。即插即用试图找到一个相互不冲突的资源的集合。为了使得所有的系统硬件一起工作，内置的或 OEM 包装的硬件必须总是支持一定的资源集。操作系统不允许中断共享，所以，SIR UART 硬件必须配置到 3 和 4 以外的中断上工作，这两个中断是早期的程序中断。这个中断配置必须通过 BIOS 提供出来。

注意，在很多情况下，提供给 SIR IrDA UART 的中断 3 和 4 的请求不能工作。

13.8.5 非即插即用总线连接 FIR 适配器

本节讨论连接到总线而且不支持任何即插即用功能的红外 FIR 设备。

在这种情况下，即插即用支持的缺乏表明了 FIR 硬件不响应总线计算或不通过系统 BIOS 呈现出来。如果用户试图安装多个原始的 FIR 硬件，它们将不可能成功。例如，多个原始 FIR 硬件包括，带许多硬件跳线的 FIR 硬件。

所有必需的 FIR 硬件也要支持 SIR 协议和速度。FIR 硬件除 FIR 的 DMA 机制以外通常为 SIR 提供一个 UART，来满足这个要求。这种配置通常要求 IrDA 微端口驱动程序和安装 INF 文件都必须复合在一起。SIR 和 FIR 部件两者都必须都是可编程的。而且，SIR 必须支持中断 3 和 4 以外的中断，必须有能力至少使用早期的 COM 3 或 COM 4 的 I/O 空间。这种资源需求对避免与早期的串口相冲突是十分必要的。这些冲突对诊断也许是微妙的和困难的。

这些 FIR 设备必须由 IrDA 微端口驱动程序和安装 INF 文件来提供。INF 文件必须列出它所支持的资源组合，必须当用户选择了一个不冲突的资源集时就能对 FIR 硬件进行编程。INF 文件必须描述所需要的其他任何一个参数，而且用户必须能配置这些参数。其他参数包括，红外收发器的类型。

一旦安装了这些 FIR 设备，它们就在“控制面板”“设备管理器”列表的“红外适配器”单元中呈现给用户，而且，安装了 FIR 设备的 IrDA 微端口驱动程序，这些驱动程序使用 FIR 资源以及下层 IrDA UART 描述的资源。

为了使用 IrDA UART 资源，IrDA 微端口驱动程序能象低层过滤驱动程序一样载入串口驱动程序，*serial.sys*，但是 IrDA 微端口驱动程序不要求这么做。

建议不要使用这种配置。

13.8.6 即插即用总线连接 FIR 适配器

本节讨论连接到总线而且支持即插即用功能的红外 FIR 设备。

同样的应用于非即插即用 FIR 设备的限制也应用于即插即用的 FIR 设备。也就是说，SIR 和 FIR 部件必须都是可编程的，而且 SIR 必须支持诊断 3 和 4 以外的中断，必须至少使用 COM3 和 COM4 的早期 I/O 空间。这些资源必须或者通过系统 BIOS 正确报告或者在安装 INF 文件中被描述。

支持即插即用的 FIR 设备必须报告一个唯一的精确描述芯片组和收发器组合的即插即用标识符。即插即用标识符的标准当前并不存在。请留意能发布已知的 FIR 硬件组合及其适当的即插即用标识符的组织，这个组织可能是 IrDA、芯片组或收发器厂商集团。

这些FIR设备必须由IrDA微端口驱动程序和安装INF文件来提供。INF文件必须为IrDA微端口驱动程序列出能支持即插即用的 SIR 和 FIR 资源的组合。

一旦 FIR 设备被安装，它们就在“控制面板”“设备管理器”列表的“红外适配器”单元中呈现给用户，而且，安装了 FIR 设备的 IrDA 微端口驱动程序，这些驱动程序使用 FIR 资源以及下层 IrDA UART 描述的资源。

FIR 设备的 IrDA 微端口驱动程序象低层过滤驱动程序一样，不能调入系统提供的串口驱动程序 *serial.sys*，FIR 设备通常提供 FIR 功能扩充了的标准 UART 寄存器。使用过滤器一样的串口驱动程序排除了对硬件或 IRQ 直接访问。

支持即插即用的 FIR 设备能以一个唯一的即插即用标识符和标准 SIR 即插即用别名呈现出来，如果操作系统能将 FIR 即插即用标识符与 IrDA 微端口驱动程序匹配起来，操作系统将安装这个驱动程序；否则，操作系统将安装串口驱动程序。

象常规串口一样不正确呈现出来的 FIR 设备只支持手工安装。在这种情况下，用户的经历如同 13.8.2 节“非即插即用内部 SIR 适配器或者象串口一样错误地呈现于外的内部 SIR 适配器”所描述的。

第三部分 NDIS 中间层驱动程序和 TDI 驱动程序

第一章 NDIS 中间层驱动程序

NDIS（网络驱动器接口标准）中间层驱动程序在其上边界导出 MiniportXxx 函数，在其下边界导出 ProtocalXxx 函数。该驱动程序在其上边界仅提供面向无连接通信支持，而在其下边界，则即可支持面向无连接通信，也可支持面向连接通信（关于面向连接通信方面更多的信息，可参阅本书的第四部分）。

中间层驱动程序的微端口部分（上边界）必须是非串行的，系统将依赖这些非串行驱动程序，而不是 NDIS 对 MiniportXxx 函数的操作进行串行化处理和对内部生成的输出包进行排队操作，这样驱动程序只要保持很小的临界区（每次只能有一个线程执行该代码）就能提供性能良好的全双工操作。但是这些非串行 Miniport 要受到更多也更严格的设计要求的限制，往往要为此付出更多的调试和测试时间（关于非串行驱动程序的更多的信息可参阅第二部分第四章 4.5 节）。

中间层驱动程序是一种典型的层次结构程序，它基于一个或多个 NDIS NIC 驱动程序，其上层是一个向上层提供 TDI（传输驱动程序接口）支持的传输驱动程序（也可能是多层结构）。从理论上讲，一个中间层驱动程序也可以是基于其他中间层驱动程序或作为其他中间层驱动程序的低层出现的，尽管这种方案未必能展现更好的性能。

中间层驱动程序的一个示例是 LAN 仿真中间层驱动程序，其上层是一个早期传输驱动程序，下层是一个非 LAN 介质的微端口 NIC 驱动程序。该驱动程序从上层接收 LAN 格式的数据包并将其转换为本地网卡的介质格式，然后将其发送到那个 NIC 的 NDIS 微端口。接收数据时，该驱动程序将低层网卡驱动程序送来的数据包转换为 LAN 兼容格式，最后向上层传输驱动程序提交这些转换过的数据包。

例如，NDISWAN 就具有一些上述特征。NDISWAN 将数据包从上层的传输 LAN 格式转换为 WAN 数据包格式，或者将数据包从低层的网卡驱动 WAN 格式转换为 LAN 数据包格式。另外，如果低层 NIC 硬件不支持这些功能，那么 NDISWAN 也可提供诸如压缩、加密和端对端协议（PPP）等的数据格式化功能。NDISWAN 为在 NDISAPI 和网卡驱动程序之间进行通信提供了一个专用接口，同时，NDISWAN 也将协议绑定映射为活动连结请求。

另一个中间层驱动程序的例子是 ATM LANE （LAN 仿真）驱动程序，它将数据包从上层无连接的传输格式转换为下层面向连接的网卡支持的 ATM 格式。

图 1.1 说明了中间层驱动程序结构

图 1.1 中间层驱动程序结构

NDIS 中间层驱动程序在 NDIS 中起着转发上层驱动程序送来的数据包，并将其向下层驱动程序发送的接口功能。当中间层驱动程序从下层驱动程序接收到数据包时，它要么调用

NdisMXxxIndicateReceive 函数，要么调用 NdisMindicateReceivePacket 函数向上层指示该数据包。

中间层驱动程序通过调用 NDIS 打开和建立一个对低层 NIC 驱动程序或者 NDIS 中间层驱动程序的绑定。中间层驱动程序提供 MiniportSetInformation 和 MiniportQueryInformation 函数来处理高层驱动程序的设置和查询请求，某些情况下，可能还要将这些请求向低层 NDIS 驱动程序进行传递，如果其下边界是面向无连接的可通过调用 NidsRequest 实现这一功能，如果其下边界是面向连接的则通过调用 NidsCoRequest 实现该功能。

中间层驱动程序通过调用 NDIS 提供的函数向网络低层 NDIS 驱动程序发送数据包。例如，下边界面向无连接的中间层驱动程序必须调用 NdisSend 或 NdisSendPackets 来发送数据包或者包数组，而在下边界面向连接的情况下就必须调用 NdisCoSendPackets 来发送包数组数据包。如果中间层驱动程序是基于非 NDIS NIC 驱动程序的，那么在调用中间层驱动程序的 MiniportSend 或 Miniport(Co)SendPackets 函数之后，发送接口对 NDIS 将是不透明的。

NDIS 提供了一组隐藏低层操作系统细节的 NdisXxx 函数和宏。例如，中间层驱动程序可以调用 NdisMInitializeTimer 来创建同步时钟，可以调用 NdisInitializeListHead 创建链表。中间层驱动程序使用符合 NDIS 标准的函数，来提高其在支持 Win32 接口的微软操作系统上的可移植性。

可分页和可丢弃代码

就像在第一部分中所描述的，每一个 MiniportXxx 函数或 ProtocolXxx 函数都运行在一个特定的 IRQL 上，在中间层驱动程序中这些函数可使用的 IRQL 从 PASSIVE_LEVEL 一直到 DISPATCH_LEVEL(包括 DISPATCH_LEVEL)。

总是运行在 IRQL PASSIVE_LEVEL 上的中间层驱动程序函数可通过调用 NDIS_PAGEABLE_FUNCTION 宏将其标记为可分页代码。驱动程序设计者应尽可能的将程序代码设计为可分页的，为那些必须驻留内存代码释放系统空间。运行在 IRQL PASSIVE_LEVEL 的驱动程序函数，当其既不调用运行在 IRQL >=DISPATCH_LEVEL 的任何函数，也不被运行在 IRQL >=DISPATCH_LEVEL 的任何函数调用时，可将其标注为可分页的。例如，一个获取自旋锁的函数，而获取自旋锁将促使获取线程提升到 IRQL DISPATCH_LEVEL 上运行。一个运行在 IRQL PASSIVE_LEVEL 的函数，如 ProtocolBindAdapter，如果被标注为可分页的，就不能再调用运行在 IRQL >=DISPATCH_LEVEL NDIS 的函数。关于运行在 IRQL 上的 NDIS 函数的更多信息，请参阅在线 DDK 的“*Network Drivers Reference*”，其中列出了每一个 NdisXxx 函数的 IRQL。

中间层驱动程序的 DriverEntry 函数以及只在 DriverEntry 中调用的代码，应该用 NDIS_INIT_FUNCTION 宏将其设定为仅用作系统初始化功能。假定 NDIS_INIT_FUNCTION 宏标识的代码仅在系统初始化时运行，这样该部分代码将只有在初始化时才会被映射，在 DriverEntry 返回后，NDIS_INIT_FUNCTION 宏标识的这部分代码将被丢弃。

共享资源的访问同步

如果驱动程序分配的资源能够被两个驱动程序函数同时共享，或者中间层驱动程序能够运行在 SMP（对称多处理）机器上，这样相同的驱动程序函数能够从多个处理器同时访问该资源，那么对这些共享资源的访问必须进行同步。例如，驱动程序维持一个共享队列，使用自旋锁来对队列的访问进行同步，自旋锁在队列创建之前调用 NdisAllocateSpinLock 进行初始化。

当然，也不必过分地保护共享资源，例如，对于队列，一些读操作不进行串行化也是可以成功执行的。但任何针对队列链接的操作都必须进行同步。自旋锁应该尽量少使用，并且

每次都要尽可能缩短其使用的时间。关于自旋锁更深入的讨论可参阅“*Kernel_Mode Drivers Design Guide*”。

1.1 中间层驱动程序的 DriverEntry 函数

为了使加载程序能够准确地识别，必须将中间层驱动程序的初始入口点明确地指定为 DriverEntry 的形式。所有其他的驱动程序导出函数，像这里所描述的 MiniportXxx 和 ProtocolXxx 函数，由于其地址传给了 NDIS，因此在设计时可由开发者任意指定名称。任何内核模式驱动程序 DriverEntry 的定义具有以下形式：

```
NTSTAUS
DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
);
```

如果除了 ProtocolXxx 之外，驱动程序还导出一组标准的内核模式驱动程序函数，那么必须将这些标准函数的地址写入要传给 DriverEntry 的驱动程序对象中。

在中间层驱动程序中，DriverEntry 至少应该完成以下工作：

1、调用 NdisMInitializeWrapper 并保存在 *NdisWrapperHandle* 中返回的句柄；

2、传递上一步保存的句柄，调用 NdisIMRegisterLayeredMiniport 注册驱动程序的 MiniportXxx 函数；

3、如果驱动程序随后要绑定到低层 NDIS 驱动程序上，则调用 NdisRegisterProtocol 注册驱动程序的 ProtocolXxx 函数；

4、如果驱动程序导出了 MiniportXxx 和 ProtocolXxx 函数，那么调用 NdisIMAssociateMiniport 向 NDIS 通告有关驱动程序的微端口低边界和协议高边界信息；

DriverEntry 能够为中间层驱动程序分配的所有共享资源初始化自旋锁，例如驱动程序用于跟踪运行中的连接和发送任务的构件和内存区。

当 DriverEntry 不能为驱动程序分配用于网络 I/O 操作的所有资源时，它就应该释放先前已经分配的任何资源并返回一个适当的错误状态。例如，如果 DriverEntry 已经调用了 NdisMInitializeWrapper 函数，那么当后续操作出错时必须调用 NdisTerminateWrapper 复位系统状态。

中间层驱动程序的 DriverEntry 函数能够执行一些全局初始化操作。然而，如果驱动程序提供了在 1.2 节所描述的，实现对低层设备的打开和绑定功能的 ProtocolBindAdapter 函数，那么驱动程序就能够让 ProtocolBindAdapter 来分配绑定相关的系统资源，ProtocolBindAdapter 根据要求为 *DeviceName* 设备进行资源分配和绑定操作。DriverEntry 必须初始化该包裹程序并注册微端口驱动程序。如果中间层驱动程序导出了一组 ProtocolXxx 函数的话，也要注册协议驱动程序。

如果中间层驱动程序仅向 NDIS 导出了一组 MiniportXxx 函数，只要向 NDIS 库注册这些函数即可，如下所述。

1.1.1 注册 NDIS 中间层驱动程序

NDIS 中间层驱动程序必须在 DriverEntry 函数的环境中向 NDIS 注册其 MiniportXxx 函数和 ProtocolXxx 函数。驱动程序通过调用 NdisIMRegisterLayeredMiniport 对 MiniportXxx

函数进行注册，该调用导出中间层驱动程序的 MiniportXxx 函数。当虚拟 NIC 被初始化时，以及随后当驱动程序将要基于该 NIC 接收和发送数据包时，注意驱动程序的控制过程。

对于指定的虚拟 NIC，NDIS 将在 NdisIMInitializeDeviceInstance 的环境中调用中间层驱动程序的 MiniportInitialize 函数对其进行初始化操作。如果中间层驱动程序导出了多个虚拟 NIC，那么为使其可用于网络请求，驱动程序必须为每一个 NIC 调用 NdisIMInitializeDeviceInstance 函数进行初始化。这样可根据网络业务量，生成相应的数量的虚拟 NIC。

如果 NDIS 中间层驱动程序也导出了一组 ProtocolXxx 函数，则必须调用相应的 NdisRegisterProtocol 函数向 NDIS 库注册这些函数。

1.1.1.1 注册中间层驱动程序的 Miniport

中间层驱动程序通过调用 NdisIMRegisterLayeredMiniport 导出 MiniportXxx 函数。

NdisIMRegisterLayeredMiniport 以如下方式进行声明：

```
NDIS_STATUS
NdisIMRegisterLayeredMiniport(
    IN NDIS_HANDLE NdisWrapperHandle,
    IN PNDIS_MINIPORT_CHARACTERISTICS MiniportCharacteristics,
    IN UINT CharacteristicsLength,
    OUT PNDIS_HANDLE DriverHandle
);
```

中间层驱动程序必须保存 NdisIMRegisterLayeredMiniport 返回的 *DriverHandle* 句柄，并且在驱动程序中调用 NdisIMInitializeDeviceInstance 函数，请求中间层驱动程序的 MiniportInitialize 函数对虚拟 NIC 进行初始化时，将该句柄输入 NDIS。当中间层驱动程序成功的绑定到一个或多个低层 NIC 驱动程序上或者当其绑定在一个非 NIC 设备驱动程序上后，将调用 NdisIMInitializeDeviceInstance 函数，使得中间层驱动程序可以初始化 Miniport 组件来接受虚拟 NIC 上的 I/O 请求。

NdisWrapperHandle 句柄是由先前的 NdisMInitializeWrapper 函数返回的。

中间层驱动程序必须完成以下操作：

1. 用 NdisZeroMemory 函数零初始化一个 NDIS_MINIPORT_CHARACTERISTICS 类型的构件；
2. 保存所有驱动程序导出的强制性的和非强制的 MiniportXxx 函数的地址，并将所有非强制的 MiniportXxx 入口指针设为 NULL；

当其他类型的 NDIS 驱动程序的有效主版本是 0x03、0x04、0x05 时，如果要导出任何新的 V4.0 或 V5.0 的 MiniportXxx 函数，中间层驱动程序的主版本必须是 4.0 并提供 4.0 或 5.0 版的 *MiniportCharacteristics* 构件。

对于 *MiniportCharacteristics*，如果驱动程序不用导出 MiniportXxx 函数，则必须将其设为 NULL，但是如果要导出函数的话，则必须将其设为某个有效的 MiniportXxx 函数地址值。

HaltHandler

当低层 NIC 超时并且 NDIS 已经中止了网卡驱动程序时，或者操作系统正在执行一个可控的系统关闭操作时，NDIS 将调用该函数。

InitializeHandler

作为中间层驱动程序调用 NdisIMInitializeDeviceInstance 初始化微端口的结果，调用该函数对虚拟网卡进行初始化。

QueryInformationHandler

该函数接收 OID_XXX 请求，这个请求来自于高层驱动程序（用 NdisRequestQueryInformation 请求类型作参数，调用 NdisRequest）。

ResetHandler

在高层协议驱动程序调用 NdisReset 的指示下，NDIS 能够调用中间层驱动程序的 MiniportReset 函数。然而，协议驱动程序并不启动复位功能，通常，NDIS 启动低层网卡驱动程序复位操作并调用中间层驱动程序的 ProtocolStatus 和 ProtocolStatusComplete 函数通知中间层驱动程序低层微端口正在复位网卡。

SetInformationHandler

该函数接收 OID_XXX 请求，这个请求来自于高层驱动程序（用 NdisRequestSetInformation 请求类型作参数，调用 NdisRequest）。

SendHandler

NDIS 调用该函数向低层网卡（或设备）驱动程序发送单个数据包。如果中间层驱动程序不支持 MiniportSendPackets 函数，那么 MiniportSend 函数（或 MiniportWanSend 函数）是必须提供的。除非中间层驱动程序总是基于那些每次只发送单一数据包或将自身绑定到低层 WAN NIC 的驱动程序，否则 MiniportSendPackets 函数应该总是以 SendPacketsHander 方式提供，而不是以该 SendHander 处理程序方式提供，关于这方面的更多讨论请参阅 1.6 节。

SendPacketsHander

该函数接收用于指定网上传输数据包的包描述符指针数组。除非中间层驱动程序绑定到低层 WAN NIC 驱动程序上并提供了 MiniportWanSend 函数，否则驱动程序应提供对 MiniportSendPackets 而不是 MiniportSend 函数支持。换句话说，不管中间层驱动程序是基于每次只能传送单个数据包的网卡驱动程序；还是基于每次可以传送多个数据包的网卡驱动程序，也不管中间层驱动程序是基于每次只能传送单个数据包的协议驱动程序；还是基于每次可以传送多个数据包的协议驱动程序，MiniportSendPackets 函数都能实现最好的性能，关于这方面的更多讨论可参阅 1.6 节。

TransferDataHandler

该函数用于传输在前视缓冲区中没有指示的接收数据包的剩余部分，该前视缓冲区由中间层驱动程序传递给 NdisMXxxIndicateReceive 函数。这个被指示的数据包可以是中间层驱动程序的 ProtocolReceive 函数或者是 ProtocolReceivePackets 处理程序接收的转换数据包。如果中间层驱动程序通过调用（除 NdisMWanIndicateReceive 之外）NdisMXxxIndicateReceive 函数向上层驱动程序指示接收数据包，那么该处理程序是必须提供的。如果中间层驱动程序总是通过调用 NdisMIndicateReceivePacket 向上层驱动程序指示接收数据包，则不必要提供 MiniportTransferData 函数。

ReturnPacketHandler

该函数接收返回包描述符（该包描述符先前通过 NdisMIndicateReceivePacket 调用向高层指示），从而释放指示给高层驱动程序的资源的控制权。在高层驱动程序处理完所有指示之后，中间层驱动程序分配的描述符及所描述的资源将返回给 MiniportReturnPacket 函数。当然，如果中间层驱动程序总是通过调用介质相关的 NdisMXxxIndicateReceive 函数向上层指示数据包，或者在调用 NdisMIndicateReceivePacket 之前总是将 OOB 数据块（与每一个描述符相关的）状态设置为 NDIS_STATUS_RESOUCES，则不必提供 MiniportReturnPacket 函数。

CheckForHangHandler

该函数以 NDIS 规定的时间间隔调用，或者以中间层驱动程序规定的时间间隔运行，二者只居其一。如果提供了该处理程序，那么 MiniportCheckForHang 函数每两秒钟被调用一次（或者按驱动程序要求的间隔调用）。关于 MiniportCheckForHang 函数的更多信息可参见

在线 DDK 的“*Network Drivers Reference*”或者该手册的第二部分。通常情况下，由于驱动程序无法确定低层网卡是否悬挂，NDIS 中间层驱动程序并不提供对 MiniportCheckForHang 函数的支持。如果驱动程序基于状态不能到达 NDIS 的非 NDIS 驱动程序，中间层驱动程序可能会提供对该处理程序的支持。

由于驱动程序并不管理中断设备，不为使用中的 IRQL 分配缓冲区，或者因为 NDIS 并不调用 MiniportReconfigure 函数（在 ReconfigureHandler 情况下），因此中间层驱动程序不会提供以下的几个微端口处理程序函数。

- DisableInterruptHandler
- EnableInterruptHandler
- HandleInterruptHandler
- ISRHandler
- AllocateCompleteHandler
- ReconfigureHandler

1.1.1.2 注册中间层驱动程序的协议

中间层驱动程序通过调用 NdisRegisterProtocol 向 NDIS 注册 ProtocolXxx 函数。

NdisRegisterProtocol 以如下方式进行声明：

```
VOID
NdisRegisterProtocol(
    OUT PNDIS_STATUS Status,
    OUT PNDIS_HANDLE NdisProtocolHandler,
    IN NDIS_PROTOCOL_CHARACTERISTICS ProtocolCharacteristics,
    IN UINT CharacteristicsLength
);
```

在调用 NdisRegisterProtocol 函数之前，中间层驱动程序必须完成以下操作：

1. 零初始化一个 NDIS_PROTOCOL_CHARACTERISTICS 类型的结构。中间层驱动程序能够使用 4.0 或者 5.0 版的 *ProtocolCharacteristics* 结构。协议驱动程序必须支持即插即用功能，因此 NDIS 不再支持 3.0 版的协议驱动程序。
2. 保存所有驱动程序支持的强制性的和非强制的 ProtocolXxx 函数的地址；

该调用的返回句柄 *NdisProtocolHandler* 对中间层驱动程序是不透明的，中间层驱动程序必须保存该句柄，并在将来 NDIS 中间层驱动程序的协议部分的函数调用中作为输入参数传递，例如，打开低层适配器的函数调用。

注册下边界面向无连接的中间层驱动程序的 ProtocolXxx 函数

下边界面向无连接中间层驱动程序能够导出的（包括可选的和必须的）协议处理程序函数如下所列：

BindAdapterHandler

这是一个必须提供的函数。NDIS 调用该函数请求中间层驱动程序绑定到低层 NIC 或虚拟 NIC 上，NIC 名作为该处理程序的一个参数传递。关于动态绑定的更多的信息参见 1.2 节.

UnbindAdapterHandler

这是一个必须提供的函数。NDIS 调用 ProtocolUnbindAdapter 释放对低层 NIC 或虚拟 NIC 的绑定，网卡名作为该处理程序的一个参数传递。当绑定成功关闭时，ProtocolUnbindAdapter 将调用 NdisCloseAdapter 函数并释放相关资源。

OpenAdapterCompleteHandler

这是一个必须提供的函数。如果中间层驱动程序对 NdisOpenAdapter 的调用返回 NDIS_STATUS_PENDING，则接着调用 ProtocolOpenAdapterComplete 来完成绑定。

CloseAdapterCompleteHandler

这是一个必须提供的函数。如果中间层驱动程序对 NdisCloseAdapter 的调用返回 NDIS_STATUS_PENDING，则接着调用 ProtocolCloseAdapterComplete 来完成绑定释放。

ReceiveHandler

这是一个必须提供的函数。ProtocolReceive 函数以指向包含网络接收数据的前视缓冲区的指针为参数被调用执行。如果该缓冲区包含的不是完整的网络数据包，ProtocolReceive 以数据包描述符作为参数，调用 NdisTransferData 接收该数据包的剩余部分。如果低层驱动程序调用 NdisMIndicateReceivePacket 指示接收数据包，那么传给 ProtocolReceive 函数的前视缓冲区将总是完整的网络数据包。

ReceivePacketHandler

这是一个可选函数。如果中间层驱动程序所基于的 NIC 驱动程序指示的是数据包描述符指针数组，或者调用 NdisMIndicateReceivePacket 函数指示接收带外数据，那么驱动程序应提供 ProtocolReceivePacket 函数。如果开发者不能确定中间层驱动程序的执行环境，也应提供函数，因为在能够产生多包指示的低层 NIC 驱动程序上，中间层驱动程序将获得更好的性能。

ReceiveCompleteHandler

这是一个必须提供的函数。当先前指示给 ProtocolReceive 函数的数据包被处理后，将调用 ProtocolReceiveComplete 函数。

TransferDataCompleteHandler

如果 ProtocolReceive 要调用 NdisTransferData 函数，则必须提供该处理程序。如果复制接收数据包剩余部分的 NdisTransferData 函数调用返回 NDIS_STATUS_PENDING，那么当传输操作完成后，将调用 ProtocolTransferDataComplete 函数。

ResetCompleteHandler

这是一个必须提供的函数。当 NdisReset 函数（返回 NDIS_STATUS_PENDING）调用启动的复位操作完成时，ProtocolResetComplete 函数将被调用。通常情况下，中间层驱动程序并不调用 NdisReset，但由于上层驱动程序可能会调用该函数，所以中间层驱动程序可能只是向低层 NDIS 驱动程序转发复位请求而已。

RequestCompleteHandler

这是一个必须提供的函数。当 NdisRequest 函数（返回 NDIS_STATUS_PENDING）调用启动的查询或设置操作完成时，ProtocolRequestComplete 函数将被调用。

SendCompleteHandler

这是一个必须提供的函数。对每一个调用 NdisSend 函数传输的数据包，当其返回 NDIS_STATUS_PENDING 作为发送状态时，将调用 ProtocolSendComplete 函数完成发送操作。如果调用 NdisSendPackets 发送一组数据包，那么对于每一个传送给 NdisSendPackets 的数据包，ProtocolSendComplete 将被调用一次。中间层驱动程序仅仅根据送给 ProtocolSendComplete 的状态参数就能确定调用 NdisSendPackets 函数的发送操作的状态。

StatusHandler

这是一个必须提供的函数。NDIS 用低层 NIC 驱动程序发起的状态通知来调用 ProtocolStatus 函数。

StatusCompleteHandler

这是一个必须提供的函数。NDIS 调用 ProtocolStatusComplete 函数来指示状态改变操作已经完成，该状态先前被指示给 ProtocolStatus 函数。

PnPEventHandler

这是一个必须提供的函数。NDIS 调用 ProtocolPnPEvent 来指示即插即用事件或电源管理事件。更多信息可参见 1.7 节。

UnloadHandler

这是一个可选函数。NDIS 调用 ProtocolUnload 函数来响应用户卸载中间层驱动程序的请求。对于每一个绑定的适配器，NDIS 在调用 ProtocolUnbindAdapter 之后，将调用 ProtocolUnload 函数卸载驱动程序。ProtocolUnload 执行驱动程序决定的清除操作。

注册下边界面向连接的中间层驱动程序的 ProtocolXxx 函数

下边界面向连接的中间层驱动程序必须注册如下的面向无连接的和面向连接的微端口所共有的协议处理程序函数：

- BindHandler
- UnbindHandler
- OpenAdapterCompleteHandler
- CloseAdapterCompleteHandler
- ReceiveCompleteHandler
- ResetCompleteHandler
- RequestCompleteHandler
- StatusCompleteHandler
- PnPEventHandler

这些函数已经在上一小节作了汇总。

下边界面向连接的中间层驱动程序还必须注册如下的面向连接协议函数：

CoSendCompleteHandler

这是一个必须提供的函数。对于传给 NdisCoSendPackets 的每一个数据包都要调用一次 ProtocolCoSendComplete 函数。中间层驱动程序仅仅根据送给 ProtocolCoSendComplete 的状态参数就能确定调用 NdisCoSendPackets 的发送操作的状态。

CoStatusHandler

这是一个必须提供的函数。NDIS 用低层 NIC 驱动程序发起的状态通知来调用 ProtocolCoStatus 函数。

CoReceivePacketsHandler

这是一个必须提供的函数。当绑定的面向连接 NIC 驱动程序或者集成微端口呼叫管理器（MCM）通过调用 NdisMCoIndicateReceivePackets 指示一个指针数组时，NDIS 将调用中间层驱动程序的 ProtocolCoReceivePacketHandler 函数。

CoAfRegisterNotifyHandler

如果中间层驱动程序是一个使用呼叫管理器或 MCM 驱动程序的呼叫管理服务的面向连接客户，那么就必须注册 ProtocolCoAfRegisterNotify 函数，该函数用来确定中间层驱动程序能否使用呼叫管理器或 MCM（已经通过注册地址族，公布其服务）的服务。

1.2 中间层驱动程序的动态绑定

中间层驱动程序必须提供 ProtocolBindAdapter 和 ProtocolUnbindAdapter 函数以支持对低层 NIC 的动态绑定。当 NIC 可用时，NDIS 调用中间层驱动程序（能够绑定到 NIC）的 ProtocolBindAdapter 函数实现动态绑定操作。

```
VOID
ProtocolBindAdapter(
```

OUT PNDIS_STATUS *Status*,
IN NDIS_HANDLE *BindContext*,
IN PNDIS_STRING *DeviceName*,
IN PVOID SystemSpecific1,
IN PVOID *SystemSpecific2*
);

BindContext 句柄代表绑定请求的 NDIS 环境，中间层驱动程序必须保存该句柄，并且在中间层驱动程序完成绑定相关操作，并准备接受发送请求时，将该句柄作为 NdisCompleteBindAdapter 的参数返回 NDIS。

绑定时的操作包括为该绑定分配 NIC 相关的环境区域并进行初始化，接着调用 NdisOpenAdapter 绑定到 *DeviceName* 参数指定的适配器。*DeviceName* 可以是低层 NIC 驱动程序管理的 NIC，也可以是介于被调用的中间层驱动程序和管理适配器的 NIC 驱动程序之间，由中间层 NDIS 驱动程序导出的控制传输请求的虚拟 NIC。通常情况下，可能仅有一个基于 NIC 驱动程序的中间层 NDIS 驱动程序，实现早期的高层协议驱动程序支持的介质格式和低层 NIC 驱动程序支持的介质格式之间的转换。

注意，中间层驱动程序向 NdisOpenAdapter 传递的 *DeviceName* 必须与先前向 ProtocolBindAdapter 函数传递的 *DeviceName*（一个 Unicode 字符串缓冲区指针）相同，驱动程序不能复制该指针并将指针副本传递给 NdisOpenAdapter 函数。

中间层驱动程序能够在已分配的绑定相关环境区域或者另一个驱动程序可访问位置保存 *BindContext*。如果 NdisOpenAdapter 返回 NDIS_STATUS_PEDDING，则必须保存 *BindContext* 值，在这种情况下，中间层驱动程序直到打开适配器的操作完成，并且其 ProtocolOpenAdapterComplete 函数已调用完成，才能调用 NdisCompleteBindAdapter 函数完成绑定工作。*BindContext* 必须从某个已知位置获得，并由 ProtocolOpenAdapterComplete 传递给 NdisCompleteBindAdapter 函数。

如果中间层驱动程序将适配器相关信息存入注册表，那么 *SystemSpecific1* 将指向注册表路径，该值将被传给 NdisOpenProtocolConfiguration 函数以获取用于读写适配器信息的句柄。

SystemSpecific2 预留系统使用。

如果中间层驱动程序要将内入数据包从一种介质格式转化为另一种格式，那么 ProtocolBindAdapter 能够分配数据包描述符池和每一个绑定所需的缓冲描述符。关于分配和管理数据包的要求可参阅 1.3 节。另外，如果中间层驱动程序仅仅用 Protocol(Co)Receive 函数接收内入数据，那么驱动程序应该分配数据包池和缓冲池来复制接收的数据。

1.2.1 打开中间层驱动程序下层的适配器

ProtocolBindAdapter 函数通过 *DeviceName* 参数值打开低层 NIC 或虚拟网卡，从而建立到低层 NIC 驱动程序的绑定，它也能够从注册表中读取所要求的附加配置信息。NdisOpenProtocolConfiguration 用于获取指向中间层驱动程序存储适配器相关信息的注册表主键句柄。中间层驱动程序通过调用 NdisOpenConfigurationKeyByIndex 函数或者 NdisOpenConfigurationKeyByName 函数打开并获取主键（由 NdisOpenProtocolConfiguration 函数打开）下的子键句柄。然后，中间层驱动程序能够调用 NdisRead(Write)Configuration 函数读写注册表主键或子键下的相关信息。NdisRead(Write)Configuration 函数在在线 DDK 的“*Network Drivers Reference*”中有详细的描述。

典型地，ProtocolBindAdapter 使用环境区域（代表对 *DeviceName* 绑定）存储所有绑定相关信息（与绑定适配器相关联的）。

绑定操作最终由 NdisOpenAdapter 函数调用来实现，该函数声明如下：

```
VOID
NdisOpenAdapter（
OUT PNDIS_STATUS Status,
OUT PNDIS_STATUS OpenErrorStatus,
OUT PNDIS_HANDLE NdisBindingHandle,
OUT PUNIT SelectedMediumIndex,
IN PNDIS_MEDIUM MediumArray,
IN UINT MediumArraySize,
IN NDIS_HANDLE NdisProtocolHandle,
IN NDIS_HANDLE ProtocolBindingContext,
IN PNDIS_STRING AdapterName,
IN UINT OpenOptions,
IN PSTRING AddressingInformation
);
```

中间层驱动程序在 *ProtocolBindingContext* 中传递代表绑定相关环境区域（已经分配并初始化）的句柄。NDIS 在未来与绑定相关的调用中，将向中间层驱动程序返回该环境。例如，在对 Protocol(Co)Receive 或 Protocol(Co)Status 函数的调用中。相似地，当 NdisOpenAdapter 调用返回时，NDIS 将向中间层驱动程序传递该 *NdisProtocolHandle* 句柄。驱动程序必须保存该句柄，通常保存在绑定相关的环境区域，在以后与该绑定相关的调用中，中间层驱动程序将向 NDIS 传送该句柄，例如 NdisSend 或 Ndis(Co)SendPackets 函数调用。

ProtocolBindAdapter 也能够通过 *MediumArray* 传递所支持的介质类型。如果 NdisOpenAdapter 函数调用成功，低层 NIC 驱动程序将选择一种其所支持的介质类型，并通过 *SelectedMediumHandle* 返回其从 *MediumArray* 中所选介质的索引。

ProtocolBindAdapter 可以通过 *NdisProtocolHandle* 传递前面对 NdisRegisterProtocol 函数成功调用所返回的值。

如果 NdisOpenAdapter 返回了一个错误，那么中间层驱动程序应该收回为绑定相关环境区域分配的内存空间，并释放为绑定分配的其他所有资源。典型地，ProtocolBindAdapter 通过调用 NdisWriteErrorLogEntry，用适当的描述信息记录任何失败的绑定操作。

1.2.2 微端口初始化

当成功打开低层 NIC 并准备在虚拟 NIC 或 NIC 上接收请求和发送数据之后，ProtocolBindAdapter 将对 NdisIMIntializeDeviceInstance 进行一次或多次调用来请求对一个或多个网卡进行初始化操作。NdisIMIntializeDeviceInstance 调用中间层驱动程序的 MiniportInitialize 函数执行指定网卡的初始化。当 MiniportInitialize 函数返回后，上层 NDIS 驱动程序就能够进行对中间层驱动程序的虚拟 NIC(s)的绑定操作了。

MiniportInitialize 函数必须分配和初始化虚拟 NIC 相关的环境区域。作为初始化操作的一部分，MiniportInitialize 必须用相应的环境句柄调用 NdisMSetAttributeEx 函数，NDIS 将在以后对 MiniportXxx 函数调用中传递该环境句柄。MiniportInitialize 也必须设置 AttributeFlags 参数（将被传递给 NdisMSetAttributeEx 函数）中的 NDIS_ATTRIBUTE_INTERMEDIATE_DRIVER 标记。中间层驱动程序通过设置 NDIS_ATTRIBUTE_INTERMEDIATE_DRIVER 标记来标识 NDIS 驱动程序类型。

另外，如果当中间层驱动程序队列中的发送和请求操作超时时，不想让 NDIS 调用 MiniportCheckForHang(或 MiniportReset)函数，那么 MiniportInitialize 必须对 AttributeFlags

参数（将被传递给 NdisMSetAttributeEx 函数）中的 NDIS_ATTRIBUTE_IGNORE_PACKET_TIMEOUT 和 NDIS_ATTRIBUTE_IGNORE_REQUEST_TIMEOUT 标记进行设置。通过设置超时标记来通知 NDIS 将由中间层驱动程序负责处理虚拟 NIC 超时操作。因为中间层驱动程序并不操纵低层 NIC，因此它无法控制到底花了多长时间完成未决发送和请求操作，驱动程序通常既不提供 MiniportCheckForHang 函数也不处理虚拟 NIC 超时。

然而，如果中间层驱动程序已经注册了 CheckForHangHandler 句柄的入口点，并且没有请求 NDIS 忽略数据包和请求超时，也没有改变超事间隔，那么，在默认情况下，将每隔两秒对 MiniportCheckForHang 函数进行一次调用。如果 MiniportCheckForHang 函数返回 TRUE，驱动程序将调用 MiniportReset 函数。如果驱动程序支持 MiniportCheckForHang 函数，那么可以通过调用 NdisMSetAttributeEx 函数来明确指定一个不同的 *TimeInSeconds* 值，改变默认的两秒调用间隔设置。

中间层驱动程序必须像一个非串行驱动程序那样进行操作，并且通过设置将要传递给 NdisMSetAttributeEx 函数的 AttributeFlags 参数中的 NDIS_ATTRIBUTE_DESERIALIZE 标记来进行注册。非串行驱动程序对 MiniportXxx 函数的操作进行串行化，并且对所有引入的发送数据包在内部进行排队，而不是依靠 NDIS 来保存发送队列。关于非串行驱动程序的更多信息可参阅第二部分第四章 4.5 节。

中间层驱动程序也必须设置 AttributeFlags 参数（将被传给 NdisMSetAttributeEx 函数）中的 NDIS_ATTRIBUTE_NO_HALT_SUSPEND 标记，防止 NDIS 在低层微端口过渡到低功耗状态之前中断驱动程序。

中间层驱动程序要确保所保持的状态信息是完全初始化过的。如果中间层驱动程序请求发送相关的资源（例如 MiniportSend 或 MiniportSendPackets 将要向相邻低层发送的数据包的包描述符），那么如果在调用 NdisIMInitializeDeviceInstance 之前，ProtocolBindAdapter 还没有分配数据包池，则分配数据包池。

1.2.3 中间层驱动程序查询和设置操作

当成功绑定到低层 NIC 并完成虚拟 NIC(s)的初始化操作之后，中间层驱动程序就可以查询低层 NIC 驱动程序的操作特性，设置其内部状态，也可以协商一些参数（如为低层 NIC 驱动程序预留的缓冲区大小等）。下边界面向无连接的中间层驱动程序通过调用 NdisRequest 实现该功能，下边界面向连接的中间层驱动程序则通过调用 Ndis(Co)Request 函数实现该功能。

中间层驱动程序也能够接收来自协议驱动程序的 MiniportQueryInformation 和 MiniportSetInformation 函数的查询和设置请求，它要么响应这些请求要么将这些请求传给低层驱动程序。

在在线 DDK 的“*Network Drivers Reference*”中包含了中间层驱动程序开发者所关心的全部通用的、面向连接的、非介质相关的 OID，以及介质相关的 OID 的详细信息。接下来将讨论几个标准且常用的通用分类 OID、面向连接 OID 以及一些介质相关 OID。

发布设置和查询请求

典型地，下边界面向无连接的中间层驱动程序通过发布 OID_GEN_MAXIMUM_FRAME_SIZE 请求，查询低层 NIC 驱动程序所支持的帧最大长度，该请求返回值不包括帧头部分的长度。

下边界面向无连接的中间层驱动程序能够用 OID_GEN_MAXIMUM_TOTAL_SIZE 请求，查询绑定从而确定低层 NIC 驱动程序所管理的 NIC 所能接纳的最大数据包，中间层驱

动程序必须对发送数据包进行设置，使其满足这一尺寸要求。如果上层驱动程序发送一个超出 NIC 驱动程序（中间层驱动程序所绑定的）所能支持尺寸的数据包，那么将会出现错误。

下边界面向无连接的中间层驱动程序能够用 OID_GEN_CURRENT_LOOKAHEAD 请求，查询前视数据缓冲区的大小。如果中间层驱动程序提交这一查询请求，NDIS 将返回对低层 NIC 驱动程序的给定绑定的最新前视缓冲区尺寸。如果中间层驱动程序进行相应的设置请求，那么它将指示所提出的前视缓冲区尺寸，但中间层驱动程序并不能保证低层 NIC 驱动程序能够按照所指示尺寸设置前视缓冲区。

下边界面向无连接的中间层驱动程序用 OID_GEN_LINK_SPEED 请求，查询低层 NIC 驱动程序的链接速率，并用该请求的返回值修改其保存的内部超时设置。下边界面向连接的中间层驱动程序用 OID_GEN_CO_LINK_SPEED 请求，查询低层 NIC 驱动程序的链接速率，并且也能够用 OID_GEN_CO_LINK_SPEED 请求，设置低层 NIC 驱动程序的链接速率。

如果中间层驱动程序绑定到 WAN NIC 驱动程序上的，那么直到接收到一个连结指示（指示本地节点和远程节点连结建立）时才能确定链接速率。关于链接指示的描述请参阅第二部分第八章的“广域网微端口驱动程序做出的指示”。

中间层驱动程序也必须确定低层 NIC 驱动程序的操作特性的设置，下边界面向无连接的中间层驱动程序用 OID_GEN_MAC_OPTIONS 请求来实现这一功能，下边界面向连接的中间层驱动程序用 OID_GEN_CO_MAC_OPTIONS 请求来实现这一功能。

下边界面向无连接的中间层驱动程序通常发布的是 OID_GEN_MAXIMUM_SEND_PACKETS 查询（特别是在中间层驱动程序导出了 MiniportSendPackets 函数情况下），驱动程序能够在以后响应高层驱动程序的 OID_GEN_MAXIMUM_SEND_PACKETS 查询时，向上层传递该查询的返回值。

中间层驱动程序也能够通过介质相关 OID 查询相关介质的当前地址，例如，下边界面向无连接的中间层驱动程序可以发布 OID_WAN_CURRENT_ADDRESS 、OID_802_3_CURRENT_ADDRESS 、OID_802_5_CURRENT_ADDRESS 或者 OID_FDDI_LONG_CURRENT_ADDRESS 查询，下边界面向连接的中间层驱动程序可以 OID_ATM_WAN_CURRENT_ADDRESS 查询。

如果必要，中间层驱动程序能够发布一个设置请求，来通知 NDIS 其操作特性的有关信息。下边界面向无连接的中间层驱动程序用 OID_GEN_PROTOCOL_OPTIONS 调用 NdisRequest 函数来实现这一功能，而下边界面向连接的中间层驱动程序用 OID_GEN_CO_PROTOCOL_OPTIONS 调用 NdisRequest 来实现这一功能。

绑定到支持 WAN 的 NIC 的中间层驱动程序同时必须完成以下设置信息请求：

- 用 OID_WAN_PROTOCOL_TYPE 请求，通知低层 NIC 驱动程序其协议的类型，该类型以单字节的网络层协议标识符形式提供；
- 用 OID_WAN_HEADER_FORMAT 请求，通知低层 NIC 驱动程序其发送数据包的头格式。

响应设置和查询请求

因为 NDIS 中间层驱动程序可被高层 NDIS 驱动程序绑定，所以它也可以接收 MiniportQueryInformation 和 MiniportSetInformation 函数的查询和设置请求。在某些情况下，中间层驱动程序所起的作用仅仅是将这些请求传递给低层驱动程序。另外，当这些请求是关于其在上边界导出的介质时，也能够对这些查询和设置请求进行响应。注意中间层驱动程序必须将其从上层 NDIS 驱动程序接收到的 OID_PNP_XXX 请求，传递给低层 Miniport 驱动程序处理。

通常情况下，中间层驱动程序所接收到的通用 OID，与其向低层 NIC 驱动程序提交的

OID 是相似甚至相同的，中间层驱动程序所接收到的介质相关 OID 将是高层驱动程序所期望的介质类型。

1.2.4 作为面向连接客户程序注册中间层驱动程序

下边界面向连接的中间层驱动程序必须作为面向连接客户程序进行注册。面向连接客户程序使用呼叫管理器或集成微端口呼叫管理器（MCM）的安装调用（call-setup）和卸载（tear-down）服务完成相关功能，也可以使用面向连接的微端口或 MCM 的接收和发送功能进行发送和接收数据操作。关于面向连接通信的更多信息请参阅第一部分第四章。

当呼叫管理器或 MCM 从 ProtocolBindAdapter 函数中调用 Ndis(M)CmRegisterAddressFamily 注册地址族时，NDIS 将调用绑定上的所有协议驱动程序的 ProtocolCoRegisterAfNotify 函数。如果中间层驱动程序在注册协议时调用了 ProtocolCoRegisterAfNotify 函数，那么 NDIS 将对中间层驱动程序的 ProtocolCoRegisterAfNotify 函数进行调用。

如果 ProtocolCoRegisterAfNotify 函数确定中间层驱动程序能够使用呼叫管理器或者 MCM（注册地址族的）的服务，那么它将为客户的每一个 AF 分配相应的环境区域并调用 NdisClOpenAddressFamily 函数注册一组客户提供的函数。

NdisClOpenAddressFamily 定义如下：

```
NDIS_STATUS
NdisClOpenAddressFamily(
IN NDIS_HANDLE NdisBindingHandle,
IN PCO_ADDRESS_FAMILY AddressFamily,
IN NDIS_HANDLE ProtocolAfContext,
IN PNDIS_CLIENT_CHARACTERISTICS ClCharacteristics,
IN UINT SizeOfClCharactertistics,
OUT PNDIS_HANDLE NdisAfHandle
);
```

在调用 NdisClOpenAddressFamily 之前，中间层驱动程序必须完成以下操作：

1. 将 PNDIS_CLIENT_CHARACTERISTICS 类型的 *ClCharacteristics* 结构体置零，该结构的最新版本为 5.0；
2. 存储驱动程序支持的 ProtocolXxx 客户函数的地址。

该调用的返回值 *NdisAfHandle* 对中间层驱动程序是不透明的，中间层驱动程序必须保存该句柄并在以后中间层驱动程序的协议部分调用中作为参数传递给 NDIS。例如，注册 SAP 的 NdisClRegisterSap 函数调用。

中间层驱动程序必须用 NdisClOpenAddressFamily 注册的客户函数有：

ClCreateVcHandler

设定呼叫器的 ProtocolCoCreateVc 函数的入口点。

ClDeleteVcHandler

设定呼叫器的 ProtocolCoDeleteVc 函数的入口点。

ClRequestHandler

设定呼叫器的 ProtocolCoRequest 函数的入口点。

ClRequestCompleteHandler

设定呼叫器的 ProtocolCoRequestComplete 函数的入口点。

ClOpenAfCompleteHandler

设定呼叫器的 ProtocolClOpenAfComplete 函数的入口点。

ClCloseAfCompleteHandler

设定呼叫器的 ProtocolClCloseAfComplete 函数的入口点。

ClRegisterSapCompleteHandler

设定呼叫器的 ProtocolClRegisterSapComplete 函数的入口点，客户程序用该函数接收远程机器的呼叫。

ClDeRegisterSapCompleteHandler

设定呼叫器的 ProtocolClDeRegisterSapComplete 函数的入口点。

ClMakeCallCompleteHandler

设定呼叫器的 ProtocolClMakeCallComplete 函数的入口点，客户程序用该函数对远程机器作外出呼叫。

ClModifyCallQoSCompleteHandler

设定呼叫器的 ProtocolClModifyCallQoSComplete 函数的入口点，客户程序用该函数对已经建立的 VC 服务质量进行动态修改，或者当准备一个内入呼叫时，用该函数和呼叫管理器协商建立 QoS。

ClCloseCallCompleteHandler

设定呼叫器的 ProtocolClCloseCallComplete 函数的入口点。

ClAddPartyCompleteHandler

设定呼叫器的 ProtocolClAddPartyComplete 函数的入口点，客户程序用该函数为对远程机器的外出呼叫建立点对多点的 VCs。

ClDropPartyCompleteHandler

设定呼叫器的 ProtocolClDropPartyComplete 函数的入口点。

ClIncomingCallHandler

设定呼叫器的 ProtocolClIncomingCall 函数的入口点，客户程序用该函数接收远程机器的呼叫。

ClIncomingCallQoSChangeHandler

设定呼叫器的 ProtocolClIncomingCallQoSChange 函数的入口点，客户程序用该函数接收来自远程机器的调用，在该远程机器上发送客户程序可以动态地改变 QoS。

ClIncomingCloseCallHandler

设定呼叫器的 ProtocolClIncomingCloseCall 函数的入口点。

ClIncomingDropPartyHandler

设定呼叫器的 ProtocolClIncomingDropParty 函数的入口点。

ClCallConnectedHandler

设定呼叫器的 ProtocolClCallConnected 函数的入口点，客户程序用该函数接收远程机器的呼叫。

即使中间层驱动程序不支持内入呼叫、外出呼叫或者“点—多点”连接，当调用 NdisClOpenAddressFamily 时，也必须设置 ProtocolCl/CoXxx 函数调用的 NDIS_CLEINT_CHARACTERISTICS 结构的每一个 ClXxx 参数成员，对于那些中间层驱动程序不支持的面向连接函数子集，ProtocolCl/CoXxx 函数将只是简单地返回 NDIS_STATUS_NOT_SUPPORTED 值。

1.3 中间层驱动程序数据包管理

中间层驱动程序从高层驱动程序接收数据包描述符，并在网络上发送，该包描述府与一

个或多个链式数据缓冲区相关联。中间层驱动程序能够对数据进行重新打包，并使用新的数据包描述符进行数据传输，也可以直接将数据包传递给低层驱动程序，如果驱动程序下边界面向无连接，可调用 NdisSend 或 NdisSendPackets 函数完成该功能，如果驱动程序下边界是面向连接的，可调用 NdisCoSendPackets 函数完成此项功能。中间层驱动程序也可以进行一些操作改变链式缓冲区的内容，或者调整内入数据包相对于其他发送任务的发送次序或发送定时。但是，即使中间层驱动程序只是向下层传递上层引入的数据报，例如，仅仅只是对数据包进行计数，也必须分配新的数据包描述符，并且要管理部分或者全部新的包结构。

每一个中间层驱动程序都必须分配自己的包描述符来代替高层的数据包描述符。如果中间层驱动程序要把数据包从一种格式转化为另一种格式，也必须分配缓冲区描述符来映射用于复制转配数据的缓冲区，该缓冲区由中间层驱动程序进行分配。如果有与复制的包描述符相关的 OOB 数据，那么可以将这些数据复制到与包描述符（中间层驱动程序分配的）相关的新 OOB 数据块，其过程是，首先，用 NDIS_OOB_DATA_FROM_PACKET 宏获取 OOB 数据区的指针，然后，调用 NdisMoveMemory 将其内容移入与新包描述符相关的 OOB 数据区。该驱动程序也能够用 NDIS_GET_PACKET_XXX 或 NDIS_SET_PACKET_XXX 宏从与老的包描述符相关的 OOB 数据区中，读取相关的内容，并写入与新包描述符相关的 OOB 数据区。

包描述符通过调用以下 NDIS 函数进行分配：

1. 调用 NdisAllocatePacketPool 或者 NdisAllocatePacketPoolEx，为固定尺寸包描述符（由呼叫器指定数量）分配并初始化一组非可分页池；
2. 调用 NdisAllocatePacket 函数，从 NdisAllocatePacketPool(Ex)已经分配的池中分配包描述符；

根据中间层驱动程序目的的不同，驱动程序能够对引入包描述符连接的缓冲区进行重新打包。例如，中间层驱动程序可以在接下来的情况下分配包缓冲池、对引入包数据重新打包：

- 如果中间层驱动程序从高层协议驱动程序接收到的数据缓冲区，比低层介质能够发送的单个缓冲区更大，那么中间层驱动程序必须将引入的数据缓冲分割成更小的、满足低层发送要求的数据缓冲。
- 中间层驱动程序在将发送任务转交低层驱动程序之前，可以通过压缩或加密数据方式来改变内入数据包的长度。

调用以下 NDIS 函数分配上面所要求的缓冲区：

1. 用 NdisAllocateBufferPool 获取用于分配缓冲区描述符的句柄；
2. 用 NdisAllocateMemory 或 NdisAllocateMemoryWithTag 分配缓冲区；
3. 用 NdisAllocateBuffer 分配和设置缓冲区描述符，映射由 NdisAllocateMemory(WithTag)分配的缓冲区，并链接到 NdisAllocatePacket 分配的包描述符上。

驱动程序可以通过调用 NdisChainBufferAtBack 或 NdisChainBufferAtFront 函数，将缓冲区描述符和包描述符进行链接。调用 NdisAllocateMemory(WithTag)返回的虚拟地址和缓冲区长度，将被传递给 NdisAllocateBuffer 函数来初始化其所映射的缓冲区描述符。

符合典型要求的包描述符能够在驱动程序初始化时根据要求进行分配，也可以通过 ProtocolBindAdapter 函数调用来实现。如果必要或者出于性能方面的考虑，中间层驱动程序开发者可以在初始化阶段，分配一定数量的包描述符和由缓冲区描述符映射的缓冲区，这样，就为 ProtocolReceive 复制内入数据（将向高层驱动程序指示）预先分配了资源，也为 MiniportSend 或 MiniportSendPackets 向相邻低层驱动程序传递引入的发送数据包，准备了可用的描述符和缓冲区。

如果在中间层驱动程序复制接收/发送数据到一个或多个缓冲区时，最末的一个缓冲的实

际数据长度比缓冲区的长度小，那么，中间层驱动程序将调用 NdisAdjustBufferLength 把该缓冲区描述符调节到数据的实际长度。当该包返回到中间层驱动程序时，应再次调用该函数将其长度调节到完整缓冲区的实际尺寸。

下边界面向无连接的中间层驱动程序能够通过 ProtocolReceivePacket 函数，从低层 NIC 驱动程序以完整数据包形式接收内入数据，该数据包由 NDIS_PACKET 类型的包描述符指定，也能够通过将内入数据指示给 ProtocolReceive 函数，并将数据复制到中间层驱动程序提供的数据包中。下边界面向连接的中间层驱动程序总是用 ProtocolCoReceivePacket 函数，从低层 NIC 驱动程序接收数据作为一个完整的数据包。

在如下情况下，中间层驱动程序能够保持对接收数据包的所有权：

- 当下边界面向无连接的中间层驱动程序向 ProtocolReceivePacket 函数指示完整数据包时；
- 当下边界面向连接的中间层驱动程序向 ProtocolCoReceivePacket 函数指示数据包时，其中 NDIS_PACKET_OOB_DATA 的 Status 成员设置为除 NDIS_STATUS_RESOURCES 以外的任何值。

在这些情况下，中间层驱动程序能够保持对该包描述符和其所描述的资源的所有权，直到所接收数据处理完毕，并调用 NdisReturnPackets 函数将这些资源返还给低层驱动程序为止。如果 ProtocolReceivePacket 向高层驱动程序传递其所接收的资源，那么至少应该用中间层驱动程序已经分配的包描述符替代引入包描述符。

根据中间层驱动程序目的的不同，当其从低层驱动程序接收完整数据包时，将有几种不同的包管理策略。例如，以下是几种可能的包管理策略：

- 复制缓冲区内容到中间层驱动程序分配的缓冲区中，该缓冲区被映射并链接到一个新的包描述符，向低层驱动程序返回该输入包描述符，然后可以向高层驱动程序指示新的数据包；
- 创建新的包描述符，将缓冲区（与被指示包描述符相关联）链接到新的包描述符，然后将新的包描述符指示给高层驱动程序。当高层驱动程序返回包描述符时，中间层驱动程序必须拆除缓冲区与包描述符间的链接，并将这些缓冲区链接到最初从低层驱动程序接收到的包描述符，最后向低层驱动程序返还最初的包描述符及其所描述的资源。

即使下边界面向无连接的中间层驱动程序支持 ProtocolReceivePacket 函数，它也提供 ProtocolReceive 函数。当低层驱动程序不释放包描述符所指示资源的所有权时，NDIS 将调用 ProtocolReceive 函数，当这类情况出现时，中间层驱动程序必须复制所接收的数据到它自己的缓冲区中。如果中间层驱动程序同时也指示了带外数据，ProtocolReceive 函数能够用 NDIS_GET_ORIGINAL_PACKET 调用 NdisGetReceivePacket，获取接收指示关联的带外数据。

对于下边界面向连接的中间层驱动程序，当低层驱动程序不释放包描述符所指示资源的所有权时，则将数据包的 NDIS_PACKET_OOB_DATA 的 Status 成员设为 NDIS_STATUS_RESOURCES，然后驱动程序的 ProtocolCoReceivePacket 函数必须将接收到数据复制到自己的缓冲区中。

重用数据包

前面已经讲过，NdisSend 为数据包描述符（中间层驱动程序提交的）返回 NDIS_STATUS_SUCCESS 状态标志后，不是将包描述符返回给 ProtocolSendComplete 函数，就是将中间层驱动程序分配的数据包返回给 MiniportReturnPacket 函数。包描述符的所有权及其所描述的资源都将返回给中间层驱动程序。如果高层驱动程序提供缓冲区（链到返回包

描述符），那么中间层驱动程序应当能够准确地向分配资源的驱动程序返回这些资源。

如果最初是中间层驱动程序分配了包描述符和链接的缓冲区，那么它就能够回收这些资源，并可以在接下来的发送和数据接收过程中使用这些资源。对中间层驱动程序来说，比起先释放这些资源，然后在需要时再进行重新分配，重新初始化并重用其所分配的包描述符、重用任何中间层驱动程序分配的链接缓冲区描述符和缓冲区，将是一种更为有效的方法。

中间层驱动程序通过调用 NdisReinitializePacket 函数对包描述符进行重新初始化，然而，中间层驱动程序必须首先确定已经调用 NdisUnchainBufferAtXxx 移去了所有链接缓冲区及其缓冲描述符。因为 NdisReinitializePacket 将清除指向缓冲区链的成员，如果没有预先释放或存储链接缓冲区，该调用将导致内存泄漏。同样地，如果与包描述符相关的 MediaSpecificInformation 中包含 OOB 数据，在重新初始化该包描述符之前，也必须回收内存。

1.4 中间层驱动程序的限制

前面各章节已经描述了中间层驱动程序必须按序执行的各项操作，现总结如下：

1. 当中间层驱动程序在 MiniportInitialize 函数中调用 NdisMSetAttributesEx 时，必须设定 NDIS_ATTRIBUTE_INTERMEDIATE_DRIVER 标识，NDIS 将仅仅通过该标识的当前值鉴别中间层驱动程序类型，并采取一定的措施确保延期操作不会导致死锁，例如向中间层驱动程序传送内部发送队列数据包。
2. 中间层驱动程序至少应该用其分配的新数据包描述符代替内入数据包描述符，不管该数据包是要向下传递给低层驱动程序进行发送，还是要向上传递给高层驱动程序进行接收。其后，还必须用原始包描述符（最初与传递给中间层驱动程序的数据包相关联）替换其自己的包描述符，例如当完成一个发送或完成一个接收指示时。另外，中间层驱动程序还必须通过返回包描述符及其所指定的资源，及时地返还其从高层或低层驱动程序借入的资源。
3. 特别地，中间层驱动程序必须遵循接下来的准则，该准则适用于所有非串行微端口驱动程序：

如果驱动程序的所有内部资源被中间层驱动程序发送函数和其他的 MiniportXxx 函数所共享，那么该资源必须通过自旋锁保护，这些函数包括 MiniportSend 或 MiniportSendPackets 以及其他 MiniportXxx 函数，唯一的例外是 MiniporReset 函数，它由 NDIS 进行串行化。

对于非串行微端口驱动程序，中间层驱动程序将其每个绑定环境区域的共享资源（仅受自旋锁保护）组织为离散的专用接收区、专用发送区和共享区域，这样相对于那种过分保护那些分布没有规律的发送、接收和共享变量的驱动程序，将能够获得更好的性能。

1.5 中间层驱动程序接收数据

这一节将讨论下边界面向无连接以及下边界面向连接的中间层驱动程序如何进行数据接收。

1.5.1 下边界面向无连接的中间层驱动程序接收数据

低层面向无连接的 NIC 驱动程序可通过下面两种方式指示数据包:

- NIC 驱动程序调用非过滤相关的 NdisMIndicateReceivePacket，传递指向数据包描述符的指针数组的指针，向高层驱动程序转让所指示包资源的所有权。当高层驱动程序处理

完相应数据后将向 NIC 驱动程序返回那些包描述符及其所指向的资源。

- NIC 驱动程序调用过滤相关的 NdisMXxxIndicateReceive 函数，传递前视缓冲区指针及数据包的大小值。

下边界面向无连接的中间层驱动程序必须提供 ProtocolReceive 函数，另外，它也可包含 ProtocolReceivePacket 函数，这依赖于其具体的运行环境而定。

- 对于下边界面向无连接的中间层驱动程序来说， ProtocolReceive 函数是必须提供的。该函数传递前视缓冲区指针，如果面向无连接中间层驱动程序检查完前视数据之后认为该数据包是高层驱动程序所需要的，那么必须将数据复制到已分配的数据包，该数据包将指示给高层驱动程序。如果前视缓冲区尺寸小于接收到的包大小，中间层驱动程序必须首先在 ProtocolReceive 的环境中调用 NdisTransferData 复制接收数据包的其余部分。

  为了让 ProtocolReceive 尽可能快的执行，中间层驱动程序应该为此目的预分配包描述符、缓冲区及缓冲区描述符。ProtocolReceive 被调用通常是因为低层驱动程序调用了 NdisMXxxIndicateReceive 函数，然而，如果低层 NIC 驱动程序用 NdisMIndicateReceivePacket 函数指示接收数据包时，被指示的数据包描述符的 OOB 数据块状态设为 NDIS_STATUS_RESOUCES，那么 ProtocolReceive 也可以被调用。因为那些 NdisMIndicateReceivePacket 指示的数据包被传递给 ProtocolReceive 函数，所以前视缓冲的大小总是与数据包的大小相等，因此中间层驱动程序不会为那些指示，调用 NdisTransferData 进行大小不相等包的处理。但是，如果低层 NIC 驱动程序也指示 OOB 数据，那么中间层驱动程序在 ProtocolReceive 处理中必须以 NIDS_GET_ORIGINAL_PACKET 为参数调用 NdisGetReceivePacket 找到这些信息。
- 对于下边界面向无连接的中间层驱动程序来说， ProtocolReceivePacket 函数是可选的。ProtocolReceivePacket 接收描述完整网络数据包的包描述符指针。如果低层面向无连接的 NIC 是 DMA 总线控制设备，驱动程序也必须相应的调用非过滤相关的 NdisMIndicateReceivePacket 函数指示接收数据包，并且，如果驱动程序绑定到低层 NIC 驱动程序，那么中间层驱动程序应提供 ProtocolReceivePacket 函数。另外，支持 OOB 数据的低层 NIC 驱动程序在大多数情况下，会在接收数据过程中向 NdisMIndicateReceivePacket 函数传递包描述符，以使中间层驱动程序能够访问与该描述符相关的 OOB 数据。

ProtocolReceivePacket 对数据包进行检查，如果它认为该包是高层驱动程序所需要的，那么它能够通过返回一个非零值的方式保持该包的所有权。如果一个非零值被返回，中间层驱动程序接下来必须以包描述符指针为参数调用 NdisReturnPackets，而且，对于一个特定的包描述符，该函数被调用的次数应与接收指示时 ProtocolReceivePacket 函数返回的非零值的数目相等。

当中间层驱动程序按指定次数调用 NdisReturnPackets 之后，将向低层驱动程序转让最初指示接收数据的包描述符的所有权及相关的缓冲区，所以中间层驱动程序应尽可能快的调用 NdisReturnPackets 函数进行相应处理。

另一方面，如果中间层驱动程序从 NdisReturnPackets 中返回零值，这表示将立即释放数据包及相关资源。例如，如果中间层驱动程序复制指示数据到自己的缓冲区，并且在向高层驱动程序提交之前内部进行数据的相应处理，这种情况将会发生。

1.5.1.1 在中间层驱动程序中实现 ProtocolReceivePacket 处理程序

当低层 NIC 驱动程序通过调用 NdisMIndicateReceivePacket 函数指示可能包含 OOB 数据的包数组时，NDIS 通常将以每一个包描述符为参数调用中间层驱动程序的 ProtocolReceivePacket 函数，并且在返回之前允许中间层驱动程序保持包描述符指定的资源，

并对数据进行相关的处理。以包数组为参数调用 NdisMIndicateReceivePacket 函数的两类典型的 NIC 驱动程序如下：

- 管理能够接收多个网络数据包到缓冲环的 DMA 总线控制适配器的 NIC 驱动程序；
- 在与包描述符相关的 NDIS_PACKET_OOB_DATA 数据块中，向高层驱动程序提供包含介质相关信息的带外数据（如包优先级等）的 NIC 驱动程序。当然，这些驱动程序并不一定非要是 DMA 总线控制设备的驱动程序。

如果中间层驱动程序被绑定到 NIC 驱动程序，就像前面提及的，那么它应该提供 ProtocolReceivePacket 函数。这使得驱动程序可以完成接下的操作：

1. 在每一个接收指示中，接收完整数据包；
2. 调用 NDIS 宏，读取与包描述符相关的 OOB 数据，而不是调用 NdisGetReceivePacket 和 NDIS_GET_ORIGINAL_PACKET 接收和复制数据;
3. 保持对内入数据包描述符的所有权，并通过这些包描述符对缓冲数据进行直接读访问，然后，在对每一个包描述符的处理中，可能要为客户程序制作该数据的多个副本；
4. 通过 NdisReturnPackets 函数返回包描述符及其所描述资源，另外还有其他可能的包描述符。

即使中间层驱动程序提供了 ProtocolReceivePacket 处理程序，NIC 驱动程序对 NdisMIndicateReceivePacket 函数的调用也可能导致对中间层驱动程序 ProtocolReceivePacket 函数的调用。当 NIC 驱动程序调用 NdisMIndicateReceivePacket 暂时释放了驱动程序分配资源的所有权之后，将依赖于这些包使用者调用 NdisReturnPackets 及时向低层驱动程序返还该部分资源。换句话说，NIC 驱动程序能够撇开接收资源运行，像 NIC 中的接收缓冲空间等。NIC 驱动程序通过向 OOB 数据块（与被传给 NdisMIndicateReceivePacket 的包数组中的包描述符相关联）写入 NDIS_STATUS_RESOURCES 状态标识来完成该项功能，该状态指示的数据包将导致 NDIS 以该包及数组中相继的其他包为参数调用高层驱动程序的 ProtocolReceive 函数，这将强制中间层驱动程序复制包数据而不是获取包的所有权。

如果中间层驱动程序想要通过调用 ProtocolReceive 函数来获取与包描述符关联的 OOB 数据，那么它必须用 NDIS_GET_ORIGINAL_PACKET 调用 NdisGetReceivePacket，复制特定的介质信息到中间层驱动程序所分配的缓冲区，另外，如果低层 NIC 驱动程序提供了时间戳，那么还必须复制 TimeSent 和 TimeReceived 信息。

1.5.1.2 在中间层驱动程序中实现 ProtocolReceive 处理程序

如果 NIC 驱动程序调用了 NdisMXxxIndicate 函数，那么驱动程序将总是调用 ProtocolReceive 函数来处理接收数据包。如果中间层驱动程序接受该数据包的话，ProtocolReceive 必须以包描述符为参数调用 NdisTransferData 函数，复制前视缓冲区及包的其余部分到中间层驱动程序已分配的缓冲区。NdisTransferData 必须在 ProtocolReceive 环境中调用，而且只能调用一次。中间层驱动程序应该设置拥有足够尺寸的链式缓冲区包描述符来保存所有的接收数据。当 NdisTransferData 返回之后，从低层 NIC 驱动程序接收来的数据将不再有用。

如果传给 ProtocolReceive 的数据是通过调用 NdisMXxxIndicateReceive 进行指示的，那么传给 ProtocolReceive 函数的前视缓冲区尺寸将不会超过用 OID_GEN_CURRENT_LOOKAHEAD 调用 NdisRequest 返回的值。对于中间层驱动程序来说，所有的前视缓冲区中的数据都是只读的。如果对 ProtocolReceive 函数的调用是由于在调用 NdisMIndicateReceivePacket 之前，低层 NIC 驱动程序把包数组中的一个或多个包状态设置为 NDIS_STATUS_RESOURCES，那么前视缓冲区的尺寸将总是等于整个网络数据包的大小，所以中间层驱动程序将不必再调用 NdisTransferData 函数。

ProtocolReceive 函数必须尽可能快的返回资源的控制权，因此在中间层驱动程序收到接

收指示之前，应确保拥有可利用的包描述符、缓冲区及缓冲区描述符。如果中间层驱动程序检查前视数据后认为该包不是其要复制的那个，驱动程序应返回 NDIS_STATUS_NOT_ACCEPT 标识。

在接收包被复制时，ProtocolReceive 函数不能处理接收数据，因为这将严重影响系统性能以及低层 NIC 从网络中接收内入数据包的能力。作为替代，中间层驱动程序在以后的 ProtocolReceiveComplete 函数中对接收数据包进行处理，该函数在随后能够进行数据包后期处理时被调用。典型地，当低层 NIC 驱动程序已经指示了 NIC 驱动程序确定的所有数据包时或者在退出 DPC 层接收句柄之前，以上操作将发生。中间层驱动程序必须对 ProtocolReceive 复制的数据包进行排队，以使 ProtocolReceiveComplete 函数能够过它们进行后期处理。

1.5.1.3 下边界面向无连接中间层驱动程序接收 OOB 数据信息

如果接收到的网络数据包被指示给 ProtocolReceive 函数，那么驱动程序必须将接收数据复制到中间层驱动程序所提供的缓冲区。如果包描述符相关的数据包 OOB 数据中包含特定介质信息和（或）时间戳信息，中间层驱动程序将调用 NdisGetReceivePacket 和 NDIS_GET_ORIGINAL_PACKET 获取介质信息以及 TimeSent 和 TimeReceived 时间戳（如果低层 NIC 驱动程序提供了那些信息）。

如果接收数据包被传给 ProtocolReceivePacket 函数，那么中间层驱动程序必须以下面的方式用 NDIS 宏保存与包关联的 OOB 数据信息：

- 用 NDIS_GET_MEDIA_SPECIFIC_INFO 读取介质相关信息，用 NDIS_SET_MEDIA_SPECIFIC_INFO 写入介质相关信息;
- 用 NDIS_GET_TIME_SENT 读 TimeSent 信息,用 NDIS_SET_TIME_TO_SEND 写 TimeSent 信息;
- 用 NDIS_GET_TIME_RECEIVED 读 TimeReceived 信息。

TimeSent 时间戳是远程节点 NIC 发送数据包的时间，如果可能的话，它将被低层 NIC 驱动程序获取并保存。TimeReceived 时间戳是内入数据包被低层 NIC 驱动程序接收的时间。

1.5.2 下边界面向连接的中间层驱动程序接收数据

面向连接的 NIC 驱动程序通过调用 NdisMIndicateReceivePacket 指示数据包，传递参数为指向数据包描述符的指针数组的指针。如果中间层驱动程序基于 NIC 驱动程序之上，NDIS 接下来将调用中间层驱动程序的 ProtocolCoReceivePacket 函数。

ProtocolReceivePacket 接收描述完整网络数据包的包描述符指针。ProtocolReceivePacket 检查该数据包，如果认为该包是高层驱动程序需要的，将通过返回非零值的方式保持对该包的所有权。如果对该包返回了非零值，中间层驱动程序接着必须以相应包描述符指针为参数调用 NdisReturnPackets 函数，而且，对于一个特定的包描述符，该函数被调用的次数应与接收指示时 ProtocolCoReceivePacket 函数返回的非零值的个数相等。

当中间层驱动程序以指定次数调用 NdisReturnPackets 之后，将向低层驱动程序转让最初指示接收数据的包描述符的所有权及相关的缓冲区，所以中间层驱动程序应尽可能快地调用 NdisReturnPackets 函数进行相应处理。

另一方面，如果中间层驱动程序从 NdisReturnPackets 中返回零值，这表示将立即释放数据包及相关资源。例如，如果中间层驱动程序复制指示数据到自己的缓冲区，并且在向高层驱动程序提交之前内部进行数据的相应处理，该种情况将会发生。

1.5.2.1 在中间层驱动程序中实现 ProtocolCoReceivePacket 处理程序

当低层 NIC 驱动程序通过调用 NdisMCoIndicateReceivePacket 函数指示可能包含 OOB 数据的包数组时，NDIS 以每一个包描述符为参数调用中间层驱动程序的

ProtocolCoReceivePacket 函数。为了获得相关包的 OOB 数据块状态，ProtocolCoReceivePacket 必须对每一个包描述符调用一次 NDIS_GET_PACKET_STATUS 宏。

如果 NIC 驱动程序在调用 NdisMCoIndicateReceivePacket 之前，将数据包描述符相关的 OOB 数据块状态设为 NDIS_ STATUS_SUCCESS，NIC 驱动程序将暂时释放与该包描述符相关的资源的所有权。在这种情况下，NIC 驱动程序将依赖于这些包的使用者及时返还该部分资源。换句话说，NIC 驱动程序能够撇开接收资源运行，像 NIC 中的接收缓冲空间等。当撇开这些资源运行时，NIC 驱动程序向与包描述符相关的 OOB 数据块写入 NDIS_STATUS_RESOURCES 标识，该状态指示的数据包将强制中间层驱动程序的 ProtocolCoReceivePacket 函数立即复制包数据，而不是保持 NIC 驱动程序分配的包资源。在这种情况下，ProtocolCoReceivePacket 必须返回零。

如果低层 NIC 驱动程序没有将 OOB 数据块（与被传给 NdisMCoIndicateReceivePacket 的包数组中的包描述符相关的）设置为 NDIS_STATUS_RESOURCES，那么将允许中间层驱动程序保持该包描述符及其指定的资源，直到中间层驱动程序、高层协议及高层协议的客户程序处理完数据并返回包描述符为止。

1.5.2.2 在下边界面向连接的中间层驱动程序中接收 OOB 数据信息

中间层驱动程序必须以下面的方式用 NDIS 宏保存与包关联的 OOB 数据信息：

- 用 NDIS_GET_MEDIA_SPECIFIC_INFO 读取介质相关信息，用 NDIS_SET_MEDIA_SPECIFIC_INFO 写入介质相关信息;
- 用 NDIS_GET_TIME_SENT 读 TimeSent 信息,用 NDIS_SET_TIME_TO_SEND 写 TimeSent 信息;
- 用 NDIS_GET_TIME_RECEIVED 读 TimeReceived 信息。

TimeSent 时间戳是远程节点 NIC 发送数据包的时间，如果可能的话，它将被低层 NIC 驱动程序获取并保存。TimeReceived 时间戳是导入数据包被低层 NIC 驱动程序接收的时间。

1.5.3 向高层驱动程序指示接收数据包

在面向连接的中间层驱动程序处理完所接收的数据（可能已将其转化为高层驱动程序所要求的格式，并已将相关数据复制到链向中间层驱动程序分配的包描述符的缓冲区）之后，可以通过调用 NdisMIndicateReceivePacket，将数据包指示给相邻的高层驱动程序，即使中间层驱动程序的微端口是面向无连接的。

1.6 通过中间层驱动程序传输数据包

在 1.1.1 节已经提到，中间层驱动程序必须提供 MiniportSendPackets 函数，并通过 NdisIMRegisterLayeredMiniport 函数进行注册。如果中间层驱动程序是位于两个支持多包发送的 NDIS 驱动程序之间的，那么 MiniportSendPackets 函数能够导入包数组的转发。如果驱动程序下边界是面向无连接的，通过调用 NdisSendPackets 实现转发；如果驱动程序下边界是面向连接的，那么通过调用 NdisCoSendPackets 实现转发。如果驱动程序是位于一次只能用 NdisSend 发一个数据包的传输器下的，MiniportSendPackets 函数能够用 NdisSend 或 Ndis(Co)SendPackets 进行单一数据包传输（对系统性能没有任何负面影响）。

上述的中间层驱动程序不会成为性能的瓶颈。如果中间层驱动程序位于可向 MiniportSendPackets 发送包数组的传输器和一次只能处理一个包的微端口之间，不考虑低层微端口的性能，MiniportSendPackets 可用 NdisSendPackets 发送接收到的包数组。NDIS 先对数组中的数据包进行排队，然后当微端口可以接受发送请求时，将数据包独立的逐个发送给

低层面向无连接的微端口的 MiniportSend 函数，当然这些操作对中间层驱动程序是透明的。

因为对 MiniportXxx 的调用是由 NDIS 管理的，同步是有保证的，因此当高层驱动程序对低层驱动程序的转发请求产生时，不必像 1.4 节描述的那样进行 NdisIMXxx 的调用。

作为最起码的要求，必须用中间层驱动程序自己的包描述符代替每一个内入包描述符。驱动程序必须保留高层驱动程序的原始的描述符（和链接的缓冲区，如果它们被复制到了新的缓冲区的话），当发送完成、返回高层驱动程序之前，驱动程序应该返回原始的包描述符和用于发送包的数据缓冲区。关于如何分配包资源以及将信息从一个包复制到另一个包的更多的信息，请参阅 1.3 节。

MiniportSendPackets 接收按序组织的包描述符数组，该顺序由 NdisSendPackets 的调用者确定。在多数情况下，当将这些引入包数组传给低层 NIC 驱动程序时，中间层驱动程序应该维持该包描述符顺序，仅仅在将它们传给低层驱动程序之前，中间层驱动程序向内入数据包增加带外信息时才可能重排该引入包数组的顺序。

当向 NdisSendPackets 传递包数组时，NDIS 将一直保持包描述符指针的顺序。低层 NIC 驱动程序假定：传给 MiniportSendPackets 函数的包指针数组隐含包将以同样的顺序发送。

下边界面向无连接的中间层驱动程序能够以包描述符指针为参数调用 NdisSend 传输单个数据包，也可以通过调用 NdisSendPackets 并传递指向包描述符（驱动程序已经分配的）指针数组的指针来传输单个或多个数据包。下边界面向连接的中间层驱动程序能够通过调用 NdisCoSendPackets 并传递指向包描述符指针数组的指针，传输单个或多个数据包。

通常情况下，下边界面向无连接的中间层驱动程序开发者，应该根据驱动程序的自身要求和低层面向无连接 NIC 驱动程序的已知特征，决定是使用 NdisSend 还是 NdisSendPackets 函数。下边界面向无连接的中间层驱动程序可以通过以 NdisQueryInformation（RequeryType 类型）和 OID_GEN_MAXIMUM_SEND_PACKETS 为参数调用 NdisRequest，获取低层驱动程序能够接受的发送包数组的最大包数量。如果低层驱动程序返回值‘1’或 NDIS_STATUS_NOT_SUPPORTED，那么中间层驱动程序可以选择使用 NdisSend 而不是 NdisSendPackets 函数。如果低层驱动程序返回大于‘1’的值，那么在发送包数组时使用 NdisSendPackets 函数会使两种驱动程序的性能都变得更好。

如果低层面向无连接的 NIC 驱动程序返回大于‘1’的值，那么下边界面向无连接的中间层驱动程序也应该提供 MiniportSendPackets 函数。另外，中间层驱动程序应该用与低层 NIC 驱动程序相等大小的值响应 OID_GEN_MAXIMUM_SEND_PACKETS 查询。

如果 OOB 数据在中间层驱动程序和 NIC 驱动程序之间传递，两个发送函数都可能被调用，因为在任何一种情况下，低层驱动程序都能使用 NDIS 提供的宏读取与包描述符相关的 OOB 数据。

如果中间层驱动程序发送超出低层 NIC 驱动程序内部资源承受能力的数据包：

- 非串行或面向连接的 NIC 驱动程序将在其内部队列中对超量的数据包进行排队，在条件允许的时候再发送它们；
- 串行 NIC 驱动程序可以对超量的数据包在内部队列中进行排队，在条件允许的时候再发送它们，也可以用 NDIS_STATUS_RESOURCE 状态返回超量数据包。在后一种情况下，NDIS 将在内部队列中保存那些数据包并在 NIC 驱动程序下一次调用 NdisMSendResourceAvailable 或 NdisMSendComplete 时，重新提交这些数据包。

当下边界面向无连接的中间层驱动程序从 MiniportSend 调用 NdisSend 函数时，将以同步或异步方式转让包描述符的所有权及其所描述的所有资源（直到发送操作完成为止）。如果 NdisSend 返回的状态不是 NDIS_STATUS_PENDING，则调用是以同步方式完成的，并且在 NdisSend 返回时包资源的所有权就将返还给中间层驱动程序。中间层驱动程序应该返还所有高层驱动程序分配的发送资源，并传送 NdisSend 调用的结果作为 MiniportSend 的返回

状态。

如果 NdisSend 返回的状态是 NDIS_STATUS_PENDING，当接下来发送操作完成的时候，发送操作的最终状态和包描述符将返回给 ProtocolSendComplete。中间层驱动程序应该从 ProtocolSendComplete 函数中调用 NdisSendComplete 传送发送状态给相邻的高层驱动程序。

当下边界面向无连接的中间层驱动程序通过调用 NdisSendPackets 发送一个或多个数据包时，或者当下边界面向连接的中间层驱动程序通过调用 NdisCoSendPackets 发送一个或多个数据包时，发送操作默认是异步的。直到每一个包描述符和该报发送的最终状态都返回给 ProtocolCoSendComplete 之后，呼叫器才释放这些包描述符的所有权。ProtocolCoSendComplete 必须像前一段描述的那样，向相邻的高层驱动程序传送每一个包的发送状态。

作为一个结论，如果中间层驱动程序用 NdisSendPackets 发送数组中的数据包，那么在 NdisSendPackets 返回时，它不能企图读取相关的 OOB 数据块的 Status 成员。该成员被 NDIS 用于跟踪转换中的发送请求的进程，它是易变的。中间层驱动程序仅仅能通过检查传送给 Protocol(Co)SendComplete 函数的 Status 参数来获取传送请求的状态。

如果在发送之前，中间层驱动程序通过重组从上层驱动程序接收的数据包，请求不同优先级的包数组传送，那么应将最高优先级的数据包放在数组的开始位置。当将这些数据包传给低层 NIC 驱动程序的 MiniportSend 或 Miniport(Co)SendPackets 函数时，NDIS 将保持该顺序，即使在内部要对一些数据包进行排队。对于每一个 Ndis(Co)SendPackets 调用，NDIS 将维持该顺序。

NDIS 永远不会企图对与包描述符相关的 OOB 数据块进行排队或检查。除非中间层驱动程序对 NIC 驱动程序处理包优先级的方式有特别的了解，否则，应假定 NIC 驱动程序按接收到的顺序发送数据包，保持接收时的顺序。

NDIS_PACKET 类型描述符的私有（Private）成员的结构对中间层驱动程序是不透明的，并且允许使用 NDIS 提供的宏或函数对其进行读访问，在某些情况下，也可进行写访问。例如，在发送一个数据包之前，中间层驱动程序可以调用 NdisSetPacketFlags 设置描述符的 NDIS 私有部分的中间判定标识。这些标识并不是 NDIS 定义的，而是协议和低层 NIC 驱动程序合作定义的。

1.6.1 传递介质相关信息

中间层驱动程序可以通过与每一个 NDIS_PACKET 描述符相关的 OOB 数据块，传送更多的介质相关信息。以下是 OOB 数据块的定义：

```
typedef struct _ NDIS_PACKET_OOB_DATA{
    union{
        ULONGLONG TimeToSend;
        ULONGLONG TimeSend;
    };
    ULONGLONG TimeReceived;
    UINT      HeaderSize;
    UINT      SizeMediaSpecificInfo;
    PVOID     MediaSpecificInformation;
    NDIS_STATUS      Status;
 } NDIS_PACKET_OOB_DATA,   * PNDIS_PACKET_OOB_DATA;
```

缓冲区中的单个记录结构 MediaSpecificInformation 定义如下：

```
typedef struct MediaSpecificInformation{
```

```
    UINT      NextEntryOffset;
    NDIS_CLASS_ID  ClassId;
    UINT      Size;
    UCHAR  ClassInformation[1];
}MEDIA_SPECIFIC_INFORMATION;
```

ClassId 成员是 NDIS 定义的枚举变量（ClassInformation[1]中发现的信息类型）。目前，支持 win32 的微软操作系统中为介质提供了四个 Class Ids：NdisClass802_3Priority、NdisClassWirelessWanMbxMailbox、NdisClassIrdaPacketInfo 和 NdisClassAtmAALInfo。关于更详细的信息请参阅在线“*the Network Drivers Reference*”。

如果中间层驱动程序知道发送数据包的低层 NIC 驱动程序要使用 OOB 数据，它就能够设定相应的 OOB 结构成员。例如，中间层驱动程序能够完成以下操作：

- 通过使用 NDIS_SET_PACKET_TIME_TO_SEND 宏设置 TimeToSend 成员，请求数据包在特定的时间发送。该宏在系统时间单元中传递请求时间。驱动程序可以调用 NdisGetCurrentSystemTime 获取系统时间，可用该值计算请求发送时间。
- 可以使用 NDIS_PACKET_SET_MEDIA_SPECIFIC_INFO 宏在 MediaSpecificInformation 中传递呼叫器缓冲区中的介质相关信息。例如，如果中间层驱动程序绑定到要求优先级的低层 NIC，那么可以设置 MediaSpecificInformation 结构的 ClassId 成员为 NdisClass802_3Priority，并通过 ClassInformation 传递优先级相关信息以及该信息的字符大小值。

1.7 处理中间层驱动程序的 PnP 事件和 PM 事件

中间层驱动程序必须能够处理即插即用（**PnP**）事件和电源管理事件（PM）。特别地：

- 中间层驱动程序必须在传给 NdisMSetAttributeEx 的 AttributeFlags 参数中设置标识，参阅 1.2.2 节;
- 中间层驱动程序的微端口部分必须处理 OID_PNP_XXX 请求;
- 中间层驱动程序的协议部分必须传送准确的 OID_PNP_XXX 请求给低层的微端口。中间层驱动程序的微端口部分必须向最初产生请求的协议驱动程序传递低层微端口对该请求的响应;
- 中间层驱动程序的协议部分必须提供 ProtocolPnPEvent 处理程序。

1.7.1 处理 OID_PNP_XXX 查询和设置

中间层驱动程序的上边界必须导出 MiniportQueryInformation 函数和 MiniportSetInformation 函数。当上层驱动程序（绑定到中间层驱动程序导出的虚设备实例的）调用 NdisRequest 设置和查询对象信息（OID_XXX）时，NDIS 将相应地调用 MiniportQueryInformation 或 MiniportSetInformation 函数实现该功能。NDIS 也可为实现自己的某项功能调用 MiniportQueryInformation 或 MiniportSetInformation。关于微端口设置和查询处理的更多信息请参阅第二部分第五章。

中间层驱动程序能够使用 NdisRequest（如果中间层驱动程序下边界是面向无连接的）或 NdisCoRequest（如果中间层驱动程序下边界是面向连接的）查询或设置低层微端口保存的 OID_XXX。

在绑定到低层 NIC 之后，中间层驱动程序应该通过查询 OID_PNP_CAPABILITIES 确定低层 NIC 的电源管理能力。如果 NIC 能够进行电源管理，低层微端口对

OID_PNP_CAPABILITIES 查询返回 NDIS_STATUS_SUCCESS。Miniport 也能够设定 NIC 唤醒能力。如果 NIC 不具备电源管理能力，低层微端口对 OID_PNP_CAPABILITIES 查询返回 NDIS_STATUS_NOT_SUPPORTED。

中间层驱动程序处理查询和设置上边界保存的即插即用对象信息的方式，在一定程度上依赖于低层 NIC 是否具有电源管理能力:

- OID_PNP_CAPABILITIES

  如果低层 NIC 具有电源管理能力，中间层驱动程序对 OID_PNP_CAPABILITIES 查询必须返回 NDIS_STATUS_SUCCESS 。在该 OID 返回的 OID_PM_WAKE_UP_CAPABILITIES 结构中，中间层驱动程序必须为每一个唤醒功能指定 NdisDeviceStateUnspecified 的设备电源状态。该响应表示中间层驱动程序具有电源管理能力但不能唤醒系统。

  如果低层 NIC 不具备电源管理能力，低层微端口对 OID_PNP_CAPABILITIES 查询返回 NDIS_STATUS_NOT_SUPPORTED。

- OID_PNP_QUERY_POWER 和 OID_PNP_SET_POWER

  中间层驱动程序对 OID_PNP_QUERY_POWER 查询和 OID_PNP_SET_POWER 设置返回 NDIS_STATUS_SUCCESS。中间层驱动程序不能将这些 OID 请求传递给低层微端口驱动程序。

- Wake-up OIDs

  如果低层 NIC 具有电源管理能力，中间层驱动程序向低层微端口传递（调用 Ndis(Co)Request）如下的关于唤醒事件的 OID_PNP_XXX：

  OID_PNP_ ENABLE_WAKE_UP
  OID_PNP_ ADD_WAKE_UP_PATTERN
  OID_PNP_ REMOVE_WAKE_UP_PATTERN
  OID_PNP_ WAKE_UP_PATTERN_LIST
  OID_PNP _WAKE_UP_ERROR
  OID_PNP _WAKE_UP_OK

  中间层驱动程序也必须把低层微端口对这些 OID 的响应传递给上边的协议驱动程序。

  如果 NIC 不具备电源管理能力，中间层驱动程序对这些 OID 查询和设置应该返回 NDIS_STATUS_NOT_SUPPORTED。

1.7.2 中间层驱动程序 ProtocolPnPEvent 处理程序的实现

如果操作系统向代表 NIC 的目标设备对象发布即插即用 IRP 或电源管理 IRP，NDIS 截取该 IRP，然后通过调用驱动程序的 ProtocolPnPEvent 处理程序向每一个绑定的中间层驱动程序和协议驱动程序指示该事件，NDIS 传递描述被指示的 PnP 事件或 PM 事件的 NET_PNP_EVENT 结构的指针。

就像 NET_PNP_EVENT 结构中 NetEvent 节点所指示的，有六个可能的 PnP 和 PM 事件：

- NetEventSetPower

  指示电源设置请求，该请求指定将 NIC 过渡到特定电源状态。中间层驱动程序应该总是通过返回 NDIS_STATUS_SUCCESS 说明成功处理该事件。关于处理电源设置请求的更多信息请参阅 1.7.3。

- NetEventQueryPower

  指示电源查询请求，该请求查询 NIC 能否过渡到特定电源状态。中间层驱

动程序应该总是通过返回 NDIS_STATUS_SUCCESS 说明成功处理该事件。中间层驱动程序应该总是能够成功执行 NetEventQueryPower。不应通过使 NetEventQueryRemoveDevice 失败的方式阻止系统过渡到睡眠状态。注意 NetEventQueryPower 之后总是要调用 NetEventSetPower。对设备当前电源状态的 NetEventSetPower 调用实际上相当于撤销了 NetEventQueryPower。

- NetEventQueryRemoveDevice

  指示设备删除查询请求，该请求查询 NIC 能否在不中断操作的情况下删除 NIC。如果中间层驱动程序不能释放设备（例如，因为设备正在使用），那么必须以返回 NDIS_STATUS_FAILURE 的方式使 NetEventQueryRemoveDevice 操作失败。

- NetEventCancelRemoveDevice

  指示取消设备删除操作请求，该请求取消 NIC 的删除操作。中间层驱动程序应该总是通过返回 NDIS_STATUS_SUCCESS 说明成功处理该事件。

- NetEventReconfigure

  指示网络部件的配置已经改变。例如，如果用户改变了 TCP/IP 的 IP 地址，NDIS 用 NetEventReconfigure 向 TCP/IP 协议指示该事件。中间层驱动程序应该总是通过返回 NDIS_STATUS_SUCCESS 说明成功完成该事件。

- NetEventBindList

  向 TDI 客户指示绑定列表已经改变。绑定列表是传输协议导出的一个或多个设备的列表，TDI 客户可绑定到上面。

- NetEventBindComplete

  指示中间层驱动程序已经绑定到所有可以绑定的 NIC 上。除非有即插即用 NIC 装入系统，否则 NDIS 不会向中间层驱动程序指示更多的 NIC。

NET_PNP_EVENT 的 Buffer 成员指向包含被指示事件特定信息的缓冲区。更多的信息请参阅在线 DDK“*Network Drivers Reference*”。

中间层驱动程序能够以 NdisCompletePnPEvent 异步地完成 ProtocolPnPEvent 调用。

1.7.3 处理规定的电源请求

中间层驱动程序处理电源设置请求的方式依赖于低层 NIC 是否具有电源管理能力。

睡眠状态的电源设置请求

1. NDIS 调用 IM 驱动程序和每一个绑定到虚设备（IM 驱动程序导出的）的协议驱动程序的 ProtocolPnPEvent 函数。该调用指定 NetEventSetPower 为睡眠状态（D1、D2、D3 网络设备的电源状态）。绑定的协议驱动程序停止向中间层驱动程序发送数据包和进行 OID 请求。中间层驱动程序停止向下边的微端口发送数据包和进行 OID 请求。
2. NDIS 向中间层驱动程序的微端口(上边)(如果该 NIC 支持电源管理，还包括低层的 NIC 微端口)发出 OID_PNP_SET_POWER 请求。当成功完成请求时，中间层驱动程序和低层微端口都返回 NDIS_STATUS_SUCCESS。中间层驱动程序不能向低层微端口传递 OID_PNP_SET_POWER 请求。
3. 通常情况下，中间层驱动程序不会清除虚 NIC 的初始化。尤其是，如果低层 NIC 具有电源管理能力并支持唤醒事件，中间层驱动程序不能清除虚 NIC 的初始化。如果中间层驱动程序调用 NdisIMDeinitialDeviceInstance 清除虚 NIC 的初始化，那么 NDIS 调用每一个绑定的协议驱动程序的 ProtocolUnbindAdapter 函数解除到虚 NIC 的绑定。在 TCP/IP 协议解除绑定之前，通过向低层微端口发送 OID_PNP_

REMOVE_WAKE_UP_PATTERN 清除所有低层微端口的唤醒模式，这实际上禁止了唤醒匹配模式。

工作状态的电源设置请求

- 如果低层 NIC 支持电源管理，NDIS 可向低层微端口发出 OID_PNP_SET_POWER 请求设置工作状态。NDIS 也可向中间层驱动程序的微端口(上边)发出 OID_PNP_SET_POWER 请求。对电源设置请求，中间层驱动程序仅仅返回 NDIS_STATUS_SUCCESS。中间层驱动程序不能向低层微端口传递 OID_PNP_SET_POWER 请求。
- 如果中间层驱动程序清除了虚 NIC 的初始化，那么必须调用 NdisIMInitialDeviceInstance 重新初始化该 NIC。在这种情况下，为了将协议驱动程序绑定到虚 NIC 上，NDIS 调用每一个上层协议驱动程序的 ProtocolBindAdapter 函数。
- NDIS 调用 IM 驱动程序和每一个绑定到虚设备（IM 驱动程序导出的）的协议驱动程序的 ProtocolPnPEvent 函数。该调用指定 NetEventSetPower 为工作状态（D0 网络设备的电源状态）。绑定的协议驱动程序可以开始向中间层驱动程序发送数据包和进行 OID 请求。中间层驱动程序可以开始向下边的微端口发送数据包和进行 OID 请求。

注意，在向低层微端口发送 OID_PNP_SET_POWER 之前，NDIS 可向中间层驱动程序微端口发出该 OID 设置 D0 状态。在调用中间层驱动程序的 ProtocolPnPEvent 函数指示 D0 状态之前，NDIS 也可调用绑定协议的 ProtocolPnPEvent 函数指示 NIC 过渡到 D0 状态。

在上述情况下，在低层 NIC 真正完全行使职能之前，中间层驱动程序的微端口将在功能上完全代表 NIC，中间层驱动程序必须具备处理这种情况的能力，例如，如果绑定的协议驱动程序向中间层驱动程序的微端口发送一个数据包，中间层驱动程序应该立即在内部对该包进行排队直到低层 NIC 完全行使职能。

1.8 中间层驱动程序复位操作

中间层驱动程序必须提供由于低层 NIC 复位而导致的发送（低层驱动程序未处理的）撤销能力。

典型地，低层驱动程序复位 NIC 是由于当 NDIS 队列发送超时或有 NIC 的绑定请求时，NDIS 调用 NIC 驱动程序的 MiniportReset 函数。如果高层驱动程序调用 NdisReset，微端口也可复位 NIC。如果低层 NIC 要复位，NDIS 以 NDIS_STATUS_RESET_START 状态调用每一个绑定协议和中间层驱动程序的 Protocol(Co)Status 函数，然后调用同一个绑定驱动程序的 ProtocolStatusComplete 函数。当 NIC 驱动程序复位完成时，NDIS 用 NDIS_STATUS_RESET_END 再一次调用 Protocol(Co)Status 函数，并接着调用 ProtocolStatusComplete 函数。

当 NIC 复位时，如果中间层驱动程序有任何已发送而该 NIC 又未处理掉的数据包，NDIS 将向中间层驱动程序返回这些包的准确状态并结束这些数据包。复位完成后，中间层驱动程序必须重新发送这些数据包。

当中间层驱动程序收到 NDIS_STATUS_RESET_START 状态时，将进行以下操作：

- 挂起发送就绪的数据包直到 Protocol(Co)Status 收到 NDIS_STATUS_RESET_END 通知并且已调用 ProtocolStatusComplete 函数为止;
- 挂起已经准备好向高层驱动程序指示的接收数据包直到 Protocol(Co)Status 收到

NDIS_STATUS_RESET_END 通知并且已调用 ProtocolStatusComplete 函数为止;

- 清除其保持的所有处理中的操作的内部状态和 NIC 状态。

在 Protocol(Co)Status 收到 NDIS_STATUS_RESET_END 并且 ProtocolStatusComplete 函数已被调用之后，中间层驱动程序可继续发送数据包、产生请求、向高层驱动程序作指示。

因为中间层驱动程序通常在 NDIS 调用 NdisMSetAttributeEx 时，就将发送和请求超时禁止掉了，所以 MiniportReset 很少被调用。如果 MiniportReset 被调用，有可能是因为高层驱动程序调用了 NdisReset，那么如果有必要的话，中间层驱动程序应复位内部状态并应该在返回之前设置 AddressingReset 为 TRUE。当低层 NIC 复位时，如果低层 NIC 处在继续发送和接收数据包状态，NDIS 将请求 MiniportSetInformation，为该 NIC 复位中间层驱动程序的内部地址状态。MiniportReset 不必完成为未处理完的发送，NDIS 将以适当的方式使高层驱动程序的未完成发送失效。不过，中间层驱动程序保存的一些相关状态可能应该清除。

中间层驱动程序能够通过调用 NdisReset 启动复位操作。如果复位请求返回 NDIS_STATUS_PENDING，那么当低层 NIC 或虚 NIC 复位时调用了 ProtocolResetComplete 并且驱动程序为 NIC 调用了 ProtocolMResetComplete。除非中间层驱动程序知道低层 NIC 不能正常运行，否则很少调用 NdisReset。例如，如果中间层驱动程序发现对于相当数量的发送或请求没有收到完成调用，而且如果有足够的关于低层 NIC 的知识断定有问题发生，则可以调用 NdisReset。然而，通常情况下，NIC 复位要求的检测和启动是通过 NDIS 和 NIC 驱动程序使用超时逻辑来完成的。

1.9 中间层驱动程序拆除绑定操作

当低层 NIC 不再可用时，中间层驱动程序从 NDIS 调用的 NdisUnbindAdapter 函数中调用 NdisCloseAdapter 来从低层 NIC 驱动程序解除绑定。例如，如果低层 NIC 超时并且 NDIS 为故障 NIC 调用了 NIC 驱动程序的 MiniportHalt 函数，那么接下来 NDIS 将调用上层的中间层驱动程序的 NdisUnbindAdapter 函数。

中间层驱动程序也能发起解除绑定的操作。例如，当在 ProtocolStatus 中接收到的 GeneralStatus 状态值为 NDIS_STATUS_CLOSING 时。如果初始化期间某个操作并发地使用适配器，例如分配所要求的资源失败，那么中间层驱动程序也必须解除到适配器的绑定，因为绑定时不能执行网络操作。

中间层驱动程序的 NdisUnbindAdapter 函数要调用 NdisCloseAdapter。中间层驱动程序不必释放预绑定资源，除非 NdisCloseAdapter 函数返回 NDIS_STATUS_SUCCESS 或者驱动程序调用 NdisCompleteUnbindAdapter。如果 NdisCloseAdapter 函数返回 NDIS_STATUS_PENDING，那么中间层驱动程序不能够释放预绑定的资源（直到 NdisCloseAdapterComplete 函数被调用为止）。

在该对绑定上的所有请求处理完后，NDIS 才能调用 NdisCloseAdapterComplete。当 NdisCloseAdapterComplete 返回控制权后，中间层驱动程序分配的代表绑定的 ProtocolBindingContext 句柄将变成无效的。

在同步地从 NdisUnbindAdapter 返回之前，或者用 NdisCompleteUnbindAdapter 异步地完成解除绑定操作之前，中间层驱动程序必须完成以下操作：

- 清除其为绑定保存的任何状态值；
- 释放其分配的用于建立绑定的所有资源；
- 调用 NdisCloseAdapter。

如果正在被结束的绑定被映射到中间层驱动程序导出的设备，并且该设备是通过调用

NdisIMInitializeDeviceInstance 初始化的，那么中间层驱动程序能够通过调用 NdisIMDeInitializeDeviceInstance 或 NdisIMInitializeDeviceInstanceEx 函数关闭这些设备。这样做的结果将导致对高层驱动程序来的发送和请求，中间层驱动程序的虚 NIC 将不再响应。

当中间层驱动程序调用 NdisCloseAdapter 之后，应该以合适的错误状态值使该绑定的所有发送请求失效。

1.10 中间层驱动程序状态指示

下边界面向无连接的中间层驱动程序被要求提供 ProtocolStatus 和 ProtocolStatusComplete 函数。当低层面向无连接的 NIC 驱动程序调用 NdisMIndicateStatus 报告硬件状态时，NDIS 调用 ProtocolStatus。当状态改变开始的时候，ProtocolStatus 被调用。如果当 ProtocolStatus 调用时状态节点指示的操作没有完成，低层 NIC 驱动程序紧跟着将调用 NdisMIndicateStatusComplete。当上面情况发生时，调用 ProtocolStatusComplete 为状态改变执行后期操作。

面向连接的中间层驱动程序被要求提供 ProtocolCoStatus 函数和 ProtocolStatusComplete 函数。当低层面向连接的 NIC 驱动程序调用 NdisMCoIndicateStatus 报告硬件状态时，NDIS 调用 ProtocolCoStatus。当状态改变开始的时候，ProtocolCoStatus 被调用。

被送给 ProtocolStatus 的 GeneralStatus 例子包括：

- NDIS_STATUS_CLOSING

  在 1.9 节讨论了该状态和 Protocol(Co)Status 的操作;
- NDIS_STATUS_RESET_START 和 NDIS_STATUS_RESET_END

  就像 1.8 节中所说的，这两个状态都将送给 Protocol(Co)Status 和 ProtocolStatusComplete 函数；
- NDIS_STATUS_LINE_UP

  该状态表明中间层驱动程序是位于已经与远程节点建立连接的支持 WAN 功能的 NIC 驱动程序之上。关于 WAN 驱动程序的更多信息请参阅第二部分第八章；
- NDIS_STATUS_RING_STATUS

  对于该状态 StatusBuffer 提供了更详细的信息，例如关于令牌环介质的问题。

当中间层驱动程序收到状态指示的时候，如果状态指示导致中间层驱动程序以影响 MiniportXxx 函数操作的方式改变其内部状态，那么它将通过调用 NdisMIndicateStatus 向上层驱动程序指示该状态。即，如果指示给中间层驱动程序的状态使得驱动程序的发送或请求失效，那么中间层驱动程序能够向高层驱动程序（可能暂停提交发送和请求的）指示该状态。然而，如果中间层驱动程序通过内部等待队列继续接收发送和请求，那么不应向上传递该状态信息。

第二章　NDIS 协议驱动程序

本章描述在下边界导出一组 ProtocolXxx 函数的 NDIS 驱动程序的特征。协议驱动程序和 NDIS 进行通讯来发送和接收网络数据包，并绑定和使用低层微端口 NIC 驱动程序或中间层 NDIS 驱动程序（它在上边界导出 MiniportXxx 接口）。

上述的 NDIS 协议驱动程序可能在上边界支持 TDI，或者可以通过驱动程序的传送栈，其中包括栈顶支持 TDI 的栈，向高层核心模式驱动程序导出私有接口。例如，NDIS 协议驱动程序可能是多模块传输实现的标准协议的最低层模块，如最高层模块支持 TDI 的 TCP/IP 协议。

和低层 NDIS 驱动程序通信来发送和接收数据包的协议驱动程序总是使用 NDIS 提供的函数来进行通信。例如，下边界面向无连接的协议驱动程序（和面向无连接介质的低层驱动程序进行通信，如以太网、令牌环网）必须调用 NdisSend 或者 NdisSendPackets 向低层 NDIS 驱动程序发送数据包，也必须调用 NdisRequest 来产生或传送查询，以及用网络相关的低层面向无连接驱动程序支持的 OID_XXX 的设置信息请求。

NDIS 也提供一组隐藏低层操作系统细节的 NdisXxx 函数。例如，协议驱动程序调用 NdisInitializeEvent 为同步目的创建事件，调用 NdisInitializeListHead 创建链表。使用那些函数的 NDIS 版本的协议驱动程序在支持 WIN32 接口的微软操作系统中可移植性更好。协议驱动程序也可调用 OS 专用的核心模式支持的例程，像用 KeInitializeEvent 创建事件、用 KeWaitForSingleObject 同步两个执行线程。核心模式支持例程的文档可查阅“Kernel-Mode Drivers Reference”。

可分页代码和可丢弃代码

每一个 ProtocolXxx 运行的 IRQL 是从 PASSIVE_LEVEL 到 DISPATCH_LEVEL。

独占地运行在 IRQL PASSIVE_LEVEL 下的函数应该用 NDIS_PAGEABLE_FUNCTION 宏标识为页面模式。驱动程序开发者应尽可能地将代码设计为可分页的，释放内存驻留代码的系统空间。运行在 IRQL PASSIVE_LEVEL 下的驱动程序函数只要其既不调用也不被运行在 IRQL〉= DISPATCH_LEVEL 下的任何函数调用——例如要求自旋锁的函数，就可设计为可分页。获取自旋锁导致获取线程的 IRQL 被提升到 IRQL PASSIVE_LEVEL。运行在 IRQL PASSIVE_LEVEL 下的驱动程序函数，如 ProtocolBindAdapter，如果被设计为可分页的，那么就不能调用任何运行在 IRQL〉= DISPATCH_LEVEL 下的 NdisXxx 函数。关于每一个 NdisXxx 函数的 IRQL，请参阅在线 DDK 的“the Network Drivers Reference”。

协议驱动程序的 DriverEntry 函数以及其中调用的代码，应该用 NDIS_INIT_FUNCTION 宏指定为仅仅用于初始化的代码。该宏标识的代码默认情况下只在系统初始化时运行一次，并且作为结果，该部分代码只在那时被映射。在函数代码被标识为仅用于初始化时，该代码将成为可丢弃的。

如果驱动程序分配的共享资源可以被两个驱动程序函数同时访问，或者如果协议驱动程序能够运行在 SMP 机器上，使得同一个协议驱动程序函数可以从多个处理器上同时访问某些资源，那么这些访问必须被同步。例如，如果驱动程序维护共享队列，自旋锁可用来串行对该队列的访问，该自旋锁应在队列创建的时候初始化。

然而，也不应过分保护共享资源，如内部驱动程序队列。有些只读操作对队列访问可不必串行化，但任何变换队列连接的操作都必须串行化。自旋锁应该尽可能少地使用，并尽可能地减少其存在时间。关于自旋锁的深入讨论请参阅“the Kernel-Mode Drivers Design Guide”。

2.1 协议 DriverEntry 及其初始化

对于驱动程序初始化要求的入口点，为了使引导程序能够识别，必须被明确地命名为 DriverEntry 形式。所有其他的被描述为 ProtocolXxx 的导出函数，由于其地址被传给了 NDIS，则可由开发者指定任何确定的名字。任何内核模式驱动程序的 DriverEntry 定义具有以下形式：

```
NTSTAUS
DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
);
```

如果除了 NDIS 定义的 ProtocolXxx 之外，驱动程序还导出一组标准的内核模式驱动程序例程，如（Tdi）DispatchXxx 和卸载例程，那么协议驱动程序必须将这些标准例程的地址以 *DriverObject* 的形式传入 DriverEntry，就像任何其他的内核模式中间层驱动程序一样。

为了和 NDIS 库建立通信，协议的 DriverEntry 必须调用 NdisRegisterProtocol 作为协议驱动程序注册，稍候将作详细描述。

DriverEntry 也能够初始化协议要求的自旋锁，如保护用于跟踪连接和运行中的发送任务或驱动程序分配的队列等的状态变量。

如果 DriverEntry 分配某个协议要求的资源失败，那么它应该释放先前为其分配的任何资源，如果必要，甚至可以包括调用 NdisDeregisterProtocol，然后返回一个适当的错误状态。

当协议驱动程序能够分配 DriverEntry 中要求的所有资源时，如果驱动程序提供了像 2.3 节描述的 ProtocolBindAdapter 函数，那么可以推迟对低层 NIC（或虚 NIC）驱动程序的断开和绑定，并保留系统资源管理绑定直到出现初始的网络 I/O 请求。然后，当 ProtocolBindAdapter 调用 NdisOpenAdapter 时，将分配低层 NIC（打开的）要求的资源。

2.1.1 注册 NDIS 协议驱动程序

驱动程序在 DriverEntry 环境中通过调用 NdisRegisterProtocol 向 NDIS 注册 ProtocolXxx 函数。NdisRegisterProtocol 定义如下：

```
VOID
NdisRegisterProtocol(
    OUT PNDIS_STATUS Status,
    OUT PNDIS_HANDLE NdisProtocolHandler,
    IN NDIS_PROTOCOL_CHARACTERISTICS ProtocolCharacteristics,
    IN UINT CharacteristicsLength
);
```

该调用的返回句柄 *NdisProtocolHandler* 对协议驱动程序是透明的，协议驱动程序必须保存该句柄并在将来对 NDIS 的调用中作为输入参数传递，例如，打开低层适配器。

在调用 NdisRegisterProtocol 之前，DriverEntry 必须完成以下操作：

1. 零初始化一个 NDIS_PROTOCOL_CHARACTERISTICS 类型的结构，例如调用 NdisZeroMemory 函数。这将确保可选入口点的尚未使用的成员设置为 NULL。如果该结构没被置零，那么在调用 NdisRegisterProtocol 之前，任何不再使用的成员必须置为 NULL;
2. 像 2.1.1.1 节中描述的那样，在 NDIS_PROTOCOL_CHARACTERISTICS 结构中指

定协议兼容的 NDIS 版本;

3. 像 2.1.1.2 节和 2.1.1.3 节中描述的那样，在 NDIS_PROTOCOL_CHARACTERISTICS 结构中保存驱动程序导出的强制性的和非强制的 ProtocolXxx 函数的地址。

2.1.1.1 为协议驱动程序指定 NDIS 版本号

NDIS_PROTOCOL_CHARACTERISTICS 结构中的 MajorNdisVersion 和 MinorNdisVersion 成员指定了协议兼容的 NDIS 版本。对协议有效的 NDIS 版本号为 5.0 和 4.0。最新的版本是 5.0。NDIS 继续支持早期的为 4.0 版设计的驱动程序。

协议驱动程序必须支持即插即用功能，所以 NDIS 不再支持 3.0 版的协议驱动程序。

2.1.1.2 注册下边界面向无连接的协议驱动程序的 ProtocolXxx 函数

面向无连接驱动程序必须和可能要求的导出协议函数如下所列：

BindAdapterHandler

这是一个必须提供的函数。NDIS 调用该函数请求协议驱动程序绑定到低层网卡或虚拟网卡上，网卡名作为该处理程序的参数传递。关于动态绑定的更多的信息参见 2.3 节.

UnbindAdapterHandler

这是一个必须提供的函数。NDIS 调用 ProtocolUnbindAdapter 释放对低层网卡或虚拟网卡的绑定，网卡名作为参数传递。当绑定成功解除时，ProtocolUnbindAdapter 函数调用 NdisCloseAdapter 并释放资源。

OpenAdapterCompleteHandler

这是一个必须提供的函数。如果协议驱动程序对 NdisOpenAdapter 的调用返回 NDIS_STATUS_PENDING，则接着调用 ProtocolOpenAdapterComplete 来完成绑定操作。

CloseAdapterCompleteHandler

这是一个必须提供的函数。如果协议驱动程序对 NdisCloseAdapter 的调用返回 NDIS_STATUS_PENDING，则接着调用 ProtocolCloseAdapterComplete 来完成解除绑定操作。

ReceiveHandler

这是一个必须提供的函数。ProtocolReceive 函数以前视缓冲区的指针为参数被调用执行。如果该缓冲区包含的不是完整的接收到的网络数据包，ProtocolReceive 以协议分配的数据包描述符作为参数，调用 NdisTransferData 指定协议分配缓冲区接收数据包的其余部分。

ReceiveCompleteHandler

这是一个必须提供的函数。ProtocolReceiveComplete 用来指出：以前指示给 ProtocolReceive 的接收数据包现在可以延期处理。

TransferCompleteHandler

这是一个必须提供的函数，除非协议排它地绑定到低层 NIC 驱动程序（以 NdisMIndicateReceivePacket 指示数据包）上。当 NdisTransferData 调用返回 NDIS_STATUS_PENDING，并且剩余数据已复制到与给定包描述符相连的协议分配的缓冲区时，ProtocolTransferDataComplete 函数被调用。

ReceivePacketHandler

这是一个可选函数。如果协议驱动程序绑定到 NIC 驱动程序（通过调用 NdisMIndicateReceivePacket 指示包数组），那么 ProtocolReceivePacket 函数应被提供。

SendCompleteHandler

这是一个必须提供的函数。对用 NdisSend 函数（返回 NDIS_STATUS_PENDING 作为发送状态）调用来传输的每一个数据包，ProtocolSendComplete 函数被调用。如果包数组被发送，那么对于每一个传给 NdisSendPackets 的数据包，ProtocolSendComplete 被调用一次，不管是否返回了 NDIS_STATUS_PENDING。

ResetCompleteHandler

这是一个必须提供的函数。当返回值为 NDIS_STATUS_PENDING 的 NdisReset 函数调用开始的启动协议复位操作完成时，ProtocolResetComplete 被调用。

RequestCompleteHandler

这是一个必须提供的函数。当以返回为 NDIS_STATUS_PENDING 的 NdisReset 函数调用开始的启动协议查询和设置操作完成时，ProtocolRequestComplete 被调用。

StatusHandler

这是一个必须提供的函数。ProtocolStatus 函数用来处理低层 NIC 驱动程序指示的状态改变。

StatusCompleteHandler

这是一个必须提供的函数。NDIS 调用 ProtocolStatusComplete 函数和 ProtocolStatus 函数来报告发起的 NDIS 驱动程序和 NIC 驱动程序复位操作的开始和结束。

PnPEventHandler

这是一个必须提供的函数。NDIS 调用 ProtocolPnPEvent 来指示即插即用事件或电源管理事件。更多信息参见 2.6 节。

UnloadHandler

这是一个可选函数。NDIS 调用 ProtocolUnload 函数来响应用户卸载中间层驱动程序的请求。对于每一个绑定的适配器，在调用 NDIS ProtocolUnbindAdapter 之后，调用 ProtocolUnload 函数卸载驱动程序。ProtocolUnload 执行驱动程序决定的清除操作。

协议驱动程序应该设置 ProtocolCharacteristics 的 TranslateHandler 成员为 NULL，该成员预留未来使用。

2.1.1.3 注册面向连接协议驱动程序的 ProtocolXxx 函数

面向连接的协议可以或者必须注册如下的，对于面向无连接的和面向连接的协议所共有的协议处理程序函数：

- BindHandler(必须)
- UnbindHandler(必须)
- OpenAdapterCompleteHandler(必须)
- CloseAdapterCompleteHandler(必须)
- ReceiveCompleteHandler(必须)
- ResetCompleteHandler(必须)
- RequestCompleteHandler(必须)
- StatusCompleteHandler(必须)
- PnPEventHandler(必须)

这些函数已经在 2.1.1.2 节已作了总结。

面向连接的协议也必须注册如下的面向连接的协议函数，说明这些入口点需要 ProtocolCharacteristics 类型的 5.0 版的结构。

CoSendCompleteHandler

这是一个必须提供的函数。对于传给 NdisCoSendPackets 的每一个数据包 ProtocolCoSendComplete 都要被调用一次。中间层驱动程序仅仅根据送给 ProtocolCoSendComplete 的状态参数就能确定调用 NdisCoSendPackets 的发送操作的状态。

CoStatusHandler

这是一个必须提供的函数。NDIS 用低层 NIC 驱动程序初始化的状态标志信息来调用 ProtocolCoStatus 函数。

CoReceivePacketsHandler

这是一个必须提供的函数。当一个绑定的面向连接的 NIC 驱动程序或 MCM 通过调用

NdisMCoIndicateReceivePackets 指示一个指针数组时，NDIS 调用中间层驱动程序的 ProtocolCoReceivePacketHandler。

CoAfRegisterNotifyHandler

如果协议驱动程序是一个使用呼叫管理器或 MCM 驱动程序的呼叫管理服务的面向连接的客户程序，则必须注册 ProtocolCoAfRegisterNotify 函数。当呼叫管理器或 MCM 驱动程序调用 Ndis(M)CmRegisterNotify 注册地址族时，NDIS 调用呼叫绑定的每一个协议的 ProtocolCoAfRegisterNotify 函数。ProtocolCoAfRegisterNotify 确定协议驱动程序是否可以使用已经发布其服务的呼叫管理器或 MCM 驱动程序的服务。独立于呼叫管理器的协议驱动程序也必须注册 CoAfRegisterNotifyHandler 处理程序，当被 NDIS 调用呼叫时，呼叫管理器的 ProtocolCoAfRegisterNotify 函数简单的返回控制权。

2.1.2 打开中间层驱动程序低层的适配器

协议驱动程序读取安装期间存储的注册信息，建立将要绑定的适配器名列表。注册信息包括网络配置时写入的绑定信息。DriverEntry 读取这些注册信息，其中包括可以绑定的适配器名。

协议驱动程序调用 NdisOpenProtocolConfiguration 获取协议驱动程序能够存储适配器相关信息的注册表主键句柄。然后，协议驱动程序可以调用 NdisOpenConfigurationKeyByIndex 或者 NdisOpenConfigurationKeyByName 函数，获取由 NdisOpenProtocolConfiguration 函数打开的主键下的子键句柄。

在注册表中存储适配器相关信息的协议驱动程序，必须在 *ProtocolName* 下存储该信息并使用 NDIS 函数访问这些信息。一旦获取了该键的句柄，协议驱动程序就可以调用 NdisRead(Write)Configuration 函数读写这些信息。NdisRead(Write)Configuration 函数在在线 DDK 的“*Network Drivers Reference*”中有详细的描述。

如果协议驱动程序提供 ProtocolBindAdapter 函数，那么传给该函数的 SystemSpecific1 参数将在 ProtocolSection 中被连续地传给 NdisOpenProtocolConfiguration。如果协议驱动程序没有提供 ProtocolBindAdapter 函数，那么假定驱动程序知道其打开的适配器的名字和 ProtocolSection 中传递的字符串值。

在协议驱动程序已经从注册表中获得了其要求的信息，并已调用 NdisRegisterProtocol 完成注册之后，在协议驱动程序能够发送数据包和接收导入数据包之前，必须绑定到微端口或中间层 NDIS 驱动程序管理的低层 NIC。协议通过调用 NdisOpenAdapter 函数将自身绑定到低层 NIC 和管理它的微端口。NdisOpenAdapter 函数描述如下：

```
VOID
NdisOpenAdapter（
OUT PNDIS_STATUS Status,
OUT PNDIS_STATUS OpenErrorStatus,
OUT PNDIS_HANDLE NdisBindingHandle,
OUT PUNIT SelectedMediumIndex,
IN PNDIS_MEDIUM MediumArray,
IN UINT MediumArraySize,
IN NDIS_HANDLE NdisProtocolHandle,
IN NDIS_HANDLE ProtocolBindingContext,
IN PNDIS_STRING AdapterName,
IN UINT OpenOptions,
IN PSTRING AddressingInformation
```

);

协议驱动程序向用于为绑定存储状态信息的适配器相关环境区域传递 *ProtocolBindingContext* 句柄，在接下来从属于绑定的调用中，NDIS 将向协议驱动程序返回该句柄，例如，对 ProtocolReceive 或 ProtocolStatus 函数的调用。NDIS 返回 *NdisProtocolHandle* 句柄给协议驱动程序，该句柄必须被协议所保存，通常放在该适配器相关环境区域，在未来与该绑定相关的调用中，协议驱动程序将向 NDIS 传送该句柄。

协议驱动程序以 *AdapterName* 形式向 NdisOpenAdapter 传送从注册表读取的或者通过 DeviceName 输入到 ProtocolOpenAdapter 函数中的适配器名。在 *MediumArray* 中传送协议驱动程序支持的介质类型。如果对 ProtocolOpenAdapter 函数的调用成功，低层 NIC 驱动程序将从 *MediumArray* 中选择一种介质并用 *SelectedMediumIndex* 参数返回其索引。成功调用 NdisRegisterProtocol 返回的值以 *NdisProtocolHandle* 形式返回给协议驱动程序。

如果协议驱动程序没能成功绑定到低层适配器，将释放以前为该适配器分配的任何资源。如果协议驱动程序不能打开可能存在的适配器中的任一个，应该释放协议以前分配的所有全局资源并返回恰当的错误状态，在这种情况下，协议驱动程序将被卸载。典型地，DriverEntry 或者 ProtocolBindAdapter 函数应该用恰当的描述信息，记录失败的绑定操作，这可以通过 NdisWriteErrorLogEntry 或者 OS 专门支持的例程（像 IoWriteErrorLogEntry）来实现。

2.1.3 协议驱动程序查询和设置操作

当协议驱动程序确定打开哪一个适配器并已成功地绑定到一个或多个适配器上之后，就可以调用 Ndis(Co)Request 查询低层 NDIS 驱动程序特征信息，也可以设置协议自身的操作特征。协议驱动程序也可协商绑定的某些参数。

驱动程序可以进行通用的 OID_GEN_(CO_)XXX 类型的独立于低层介质的 OID 请求。驱动程序也能够进行为介质相关类型定义的介质相关的 OID 请求。在线 DDK 的“*Network Drivers Reference*”有关于这两种类型的 OID 的详细描述。有一些 OID 是强制要求的，低层驱动程序必须支持它们，而其他的则是可选的。

设置和查询请求的处理

面向无连接协议调用 NdisRequest 来查询或设置低层微端口的特征信息。

面向无连接协议驱动程序使用通用的 OID_GEN_MAXIMUM_FRAME_SIZE，查询用字节表示的低层 NIC 驱动程序支持的最大帧尺寸，返回的尺寸大小不包括包头部分。

面向无连接协议驱动程序使用通用的 OID_GEN_MAXIMUM_TOTAL_SIZE，查询绑定确定的低层 NIC 驱动程序管理的 NIC 所能提供的最大数据包尺寸，协议驱动程序必须限制发送包大小以满足这一尺寸要求。如果协议驱动程序向低层 NIC 驱动程序发送一个超出其所能支持的尺寸的数据包，将会产生错误。

面向无连接协议驱动程序应该使用 OID_GEN_MAXIMUM_SEND_PACKETS，确定低层 NIC 驱动程序每次调用 NdisSendPackets 所能接受的发送包的数量。关于该请求怎样影响协议驱动程序的下一次发送操作，请参阅 2.5 节关于这方面的讨论。

面向无连接协议驱动程序可以查询或设置低层驱动程序提供的前视缓冲区尺寸（协议驱动程序的 ProtocolReceive 函数被调用时），使用的 OID 是 OID_GEN_CURRENT_LOOKAHEAD。如果协议驱动程序将其作为查询请求提交，NDIS 将返回为低层 NIC 驱动程序的给定绑定分配的最新前视缓冲区尺寸。如果协议驱动程序用该 OID 产生设置请求，将设定所指定的前视缓冲区，但协议驱动程序并不保证低层驱动程序能够与该请求相一致。如果对于先前的 OID_GEN_MAXIMUM_SEND_PACKETS 查询低层驱

动程序返回大于‘1’的值，那么该驱动程序将总是用包含完整网络数据包的前视缓冲区指示接收数据包。

面向无连接协议驱动程序可以用 OID_GEN_LINK_SPEED 来向低层 NIC 驱动程序查询链接速率，并用返回值设定其维护的内部超时值设置。如果协议驱动程序绑定到 NDISWAN，那么直到从 WAN NIC 驱动程序接收到一个指示本地节点和远程节点连结建立的连结指示，才能确定链接速率。

面向无连接协议驱动程序也必须通过 OID_GEN_MAC_OPTIONS 查询，确定低层 NIC 驱动程序的操作特性，该查询返回低层驱动程是否支持双工操作等方面的信息。关于该查询可以返回的 NDIS 定义的标识组请参阅在线 DDK 的“*the Network Drivers Reference*”。

如果必要，面向无连接协议驱动程序能够通过 OID_GEN_PROTOCOL_OPTIONS 产生设置请求，向 NDIS 通知它的有关操作特性。

2.1.3.2 面向连接协议驱动程序的查询和设置

面向连接协议驱动程序调用 NdisCoRequest 查询和设置低层面向连接微端口的特征信息。

面向连接协议驱动程序可以用 OID_GEN_CO_LINK_SPEED 来向低层 NIC 驱动程序查询链接速率，并用返回值设定其维护的内部超时值设置。如果协议驱动程序绑定到 NDISWAN，那么直到从 WAN NIC 驱动程序接收到一个指示本地节点和远程节点连结建立的连结指示，才能确定链接速率。面向连接协议驱动程序也可以用 OID_GEN_CO_LINK_SPEED 设置低层 NIC 的速率。

面向连接协议驱动程序也必须通过 OID_GEN_MAC_OPTIONS 查询，确定低层 NIC 驱动程序的操作特性。关于该查询可以返回的 NDIS 定义的标识组，请参阅在线 DDK 的“*the Network Drivers Reference*”。

如果必要，面向连接协议驱动程序可以通过 OID_GEN_PROTOCOL_OPTIONS 产生设置请求，向 NDIS 通知它的有关操作特性。

2.1.3.3 对 WAN Miniports 的查询和设置

绑定到支持 WAN 功能 NIC 上的面向无连接或者面向连接的协议驱动程序也必须产生如下的设置信息请求:

- OID_WAN_PROTOCOL_TYPE 通知 NDISWAN 驱动程序关于协议驱动程序的类型，该类型值以单字节网络层协议标识符形式给出。
- OID_WAN_HEADER_FORMAT 通知 NDISWAN 驱动程序关于发送数据包的头格式。

2.1.3.4 介质相关查询

协议驱动程序用介质相关 OID 查询相关介质的当前地址，例如，面向无连接协议驱动程序能够处理 OID_WAN_CURRENT_ADDRESS、OID_802_3_CURRENT_ADDRESS、OID_802_5_CURRENT_ADDRESS 或者 OID_FDDI_LONG_CURRENT_ADDRESS 查询。面向连接协议驱动程序能够处理 OID_WAN_CURRENT_ADDRESS 查询或者 OID_ATM_HW_CURRENT_ADDRESS 查询。

2.1.4 作为呼叫管理器或者面向连接客户程序进行注册

面向连接协议驱动程序必须作为一个呼叫管理器或者面向连接客户程序进行注册。关于呼叫管理器和面向连接客户程序的描述请参阅第一部分第四章。

2.1.4.1 作为呼叫管理器注册

协议驱动程序通过调用 NdisCmRegisterAddressFamily 为呼叫管理器注册。当驱动程序调用 NidsOpenAdapter 建立到低层微端口的绑定之后，协议驱动程序从 ProtocolBindAdapter 函

数中调用 NdisCmRegisterAddressFamily 进行注册。每当协议驱动程序的 ProtocolBindAdapter 函数被调用时，它都要调用 NdisCmRegisterAddressFamily 函数，为每一个向低层面向连接客户程序提供呼叫管理服务的 NIC 有效地注册地址族。

通过调用 NdisCmRegisterAddressFamily 函数，协议驱动程序注册其呼叫管理函数并通告为绑定的每一个客户提供的具体的信号服务。当呼叫管理器的 ProtocolBindAdapter 函数成功注册独立呼叫管理器并返回之后，NDIS 调用绑定上的所有面向连接客户的 ProtocolCoRegisterAfNotify 函数（参见 2.1.4.2 节）。

NdisCmRegisterAddressFamily 定义如下：

NDIS_STATUS

NdisCmRegisterAddressFamily(

IN NDIS_HANDLE *NdisBindingHandle*,

IN PCO_ADDRESS_FAMILY *AddressFamily*,

IN PNDIS_CLIENT_CHARACTERISTICS *ClCharacteristics*,

IN UINT *SizeOfClCharactertistics*,

);

在调用 NdisCmRegisterAddressFamily 之前，中间层驱动程序必须完成以下操作：

1. 对一个 PNDIS_CLIENT_CHARACTERISTICS 类型的结构体置零，*ClCharacteristics* 结构体的最新版本为 5.0；
2. 存储驱动程序支持的 ProtocolXxx 呼叫管理函数的地址。

协议驱动程序必须用 NdisCmRegisterAddressFamily 注册的呼叫管理函数有：

CmCreateVcHandler

设定呼叫器的 ProtocolCoCreateVc 函数的入口点。

CmDeleteVcHandler

设定呼叫器的 ProtocolCoDeleteVc 函数的入口点。

CmOpenAfHandler

设定呼叫器的 ProtocolCmOpenAf 函数的入口点。

CmCloseAfHandler

设定呼叫器的 ProtocolCmCloseAf 函数的入口点。

CmRegisterSapHandler

设定呼叫器的 ProtocolCmRegisterSap 函数的入口点。

CmDeRegisterSapHandler

设定呼叫器的 ProtocolCmDeRegisterSap 函数的入口点。

CmMakeCallHandler

设定呼叫器的 ProtocolCmMakeCall 函数的入口点。

CmCloseCallHandler

设定呼叫器的 ProtocolCmCloseCall 函数的入口点。

CmIncomingCallCompleteHandler

设定呼叫器的 ProtocolCmIncomingCallComplete 函数的入口点。

CmAddPartyHandler

设定呼叫器的 ProtocolCmAddParty 函数的入口点。

CmDropPartyHandler

设定呼叫器的 ProtocolCmDropParty 函数的入口点。

CmActivateVcCompleteHandler

设定呼叫器的 ProtocolCmActivateVcComplete 函数的入口点。

CmDeActivateVcCompleteHandler

设定呼叫器的 ProtocolDeCmActivateVcComplete 函数的入口点。

CmMakeCallQoSHandler

设定呼叫器的 ProtocolCmMakeCallQoS 函数的入口点。

CmRequestHandler

设定呼叫器的 ProtocolCoRequest 函数的入口点。

CmRequestCompleteHandler

设定呼叫器的 ProtocolCoRequestComplete 函数的入口点。

即使协议驱动程序不支持内入呼叫、外出呼叫或者“点—多点”连接，协议驱动程序也必须设置 NDIS_CLEINT_CHARACTERISTICS 结构中的每一个 CmXxx 成员，对于那些呼叫管理器不支持的面向连接函数子集，占位符 ProtocolCmXxx 函数只是简单地返回 NDIS_STATUS_NOT_SUPPORTED。

当独立呼叫管理器成功调用 NdisCmRegisterAddressFamily 函数之后，NDIS 将忽略 CM 以前在 NDIS_CLEINT_CHARACTERISTICS 结构（传给 NdisRegisterProtocol 的）的 RequestHandler 和 RequestCompleteHandler 成员中设定的每一个入口点。

2.1.4.2 作为面向连接客户注册

当呼叫管理器或 MCM 从 ProtocolBindAdapter 函数中调用 Ndis(M)CmRegisterAddressFamily 注册地址族时，NDIS 将调用绑定的所有协议驱动程序的 ProtocolCoRegisterAfNotify 函数。如果当协议驱动程序作为协议注册时调用了 ProtocolCoRegisterAfNotify 函数，那么 NDIS 将调用协议驱动程序的 ProtocolCoRegisterAfNotify 函数。

如果 ProtocolCoRegisterAfNotify 函数判定协议驱动程序能够使用呼叫管理器或 MCM 注册到地址族的服务，它将为客户的每一个 ProtocolCoRegisterAfNotify 函数分配环境区域，并调用 NdisClOpenAddressFamily 函数为中间层驱动程序注册客户支持的函数集。

NdisClOpenAddressFamily 定义如下：

```
NDIS_STATUS
NdisClOpenAddressFamily(
IN NDIS_HANDLE NdisBindingHandle,
IN PCO_ADDRESS_FAMILY AddressFamily,
IN NDIS_HANDLE ProtocolAfContext,
IN PNDIS_CLIENT_CHARACTERISTICS ClCharacteristics,
IN UINT SizeOfClCharactertistics,
OUT PNDIS_HANDLE NdisAfHandle
);
```

在调用 NdisClOpenAddressFamily 之前，中间层驱动程序必须完成以下操作：

1. 对一个 PNDIS_CLIENT_CHARACTERISTICS 类型的结构体置零，*ClCharacteristics* 结构体的最新版本为 5.0；
2. 存储驱动程序支持的 ProtocolXxx 客户函数的地址。

该调用的返回值 *NdisAfHandle* 对中间层驱动程序是不透明的，这个句柄必须由该驱动程序保存，并在以后驱动程序协议部分调用时作为输入参数传递给 NDIS，例如，用 NdisClRegisterSap 函数注册一个 SAP 的调用。

必须用 NdisClOpenAddressFamily 注册的协议驱动程序客户函数有：

ClCreateVcHandler

设定呼叫器的 ProtocolCoCreateVc 函数的入口点。

ClDeleteVcHandler

设定呼叫器的 ProtocolCoDeleteVc 函数的入口点。

ClRequestHandler

设定呼叫器的 ProtocolCoRequest 函数的入口点。

ClRequestCompleteHandler

设定呼叫器的 ProtocolCoRequestComplete 函数的入口点。

ClOpenAfCompleteHandler

设定呼叫器的 ProtocolClOpenAfComplete 函数的入口点。

ClCloseAfCompleteHandler

设定呼叫器的 ProtocolClCloseAfComplete 函数的入口点。

ClRegisterSapCompleteHandler

设定呼叫器的 ProtocolClRegisterSapComplete 函数的入口点，客户程序用该函数接收远程机器的调用。

ClDeRegisterSapCompleteHandler

设定呼叫器的 ProtocolClDeRegisterSapComplete 函数的入口点。

ClMakeCallCompleteHandler

设定呼叫器的 ProtocolClMakeCallComplete 函数的入口点，客户程序用该函数对远程机器进行输出调用。

ClModifyCallQoSCompleteHandler

设定呼叫器的 ProtocolClModifyCallQoSComplete 函数的入口点，客户程序用该函数对已经建立的 VC 服务质量进行动态修改，或者当准备建立一个内入呼叫时，用该函数和呼叫管理器协商建立 QoS。

ClCloseCallCompleteHandler

设定呼叫器的 ProtocolClCloseCallComplete 函数的入口点。

ClAddPartyCompleteHandler

设定呼叫器的 ProtocolClAddPartyComplete 函数的入口点，客户程序用该函数为对远程机器的输出调用建立点对多点的 VC。

ClDropPartyCompleteHandler

设定呼叫器的 ProtocolClDropPartyComplete 函数的入口点。

ClIncomingCallHandler

设定呼叫器的 ProtocolClIncomingCall 函数的入口点，客户程序用该函数接收远程机器来的调用。

ClIncomingCallQoSChangeHandler

设定呼叫器的 ProtocolClIncomingCallQoSChange 函数的入口点，客户程序用该函数接收远程机器来的调用，在该远程机器上，发送客户程序可以动态修改 QoS。

ClIncomingCloseCallHandler

设定呼叫器的 ProtocolClIncomingCloseCall 函数的入口点。

ClIncomingDropPartyHandler

设定呼叫器的 ProtocolClIncomingDropParty 函数的入口点。

ClCallConnectedHandler

设定呼叫器的 ProtocolClCallConnected 函数的入口点，客户程序用该函数接收远程机器来的调用。

即使中间层驱动程序不支持内入呼叫、外出呼叫或者“点—多点”连接，当调用 NdisClOpenAddressFamily 时，协议驱动程序也必须为呼叫器的 ProtocolCl/CoXxx 函数设置

NDIS_CLEINT_CHARACTERISTICS 结构的每一个 ClXxx 参数成员，对于那些中间层驱动程序不支持的面向连接函数子集，ProtocolCl/CoXxx 函数仅仅返回 NDIS_STATUS_NOT_SUPPORTED。

2.2 协议驱动程序数据包管理

协议驱动程序从客户程序接收一个或多个网上传送数据的缓冲区。协议驱动程序必须分配并初始化客户数据缓冲区链接的包描述符。包描述符必须像接下来描述的这样从包池中分配：

1. 在驱动程序初始化或者每个绑定初始建立时，调用 NdisAllocatePacketPool 或 NdisAllocatePacketPoolEx 为固定大小的包描述符（呼叫器其指定数量）分配并初始化一组非可分页池；
2. 调用 NdisAllocatePacket 从 NdisAllocatePacketPool 已经分配的池中分配包描述符；

通过调用 NdisChainBufferAtBack 或者 NdisChainBufferAtFront 函数，将映射到缓冲区的缓冲区描述符链接到包描述符。如果协议驱动程序从客户程序接收到必须以几个更小缓冲区发送的数据缓冲区，协议驱动程序能够将这些数据复制到协议已经分配的缓冲区，用以前已经分配的缓冲区描述符映射这些缓冲区，然后将这些缓冲区描述符链接到协议已经分配的包描述符。这些缓冲区可以通过调用核心模式支持的例程，如 NdisAllocateMemory 或者 NdisAllocateMemoryWithTag 来分配，并可像以下描述的这样，映射到协议驱动程序已经分配的缓冲区描述符：

1. 在驱动程序初始化或者每个绑定初始建立时，调用 NdisAllocateBufferPool 获取用于分配缓冲区描述符的句柄；
2. 调用 NdisAllocateMemory（WithTag）分配缓冲区，并将其链接到 NdisAllocatePacket 分配的包描述符上；
3. 调用 NdisAllocateBuffer 分配和设置一个缓冲区描述符，该描述符映射到由 NdisAllocateMemory(WithTag)调用分配缓冲区。

调用 NdisAllocateMemory(WithTag) 返回的基准虚地址和长度将传递给 NdisAllocateBuffer，来初始化缓冲区描述符。

满足典型传送要求的包描述符可以在驱动程序初始化和（或）绑定时根据需要分配。协议驱动程序开发者可以在初始化时分配一定数量的包描述符和链接的缓冲区描述符，以保存协议绑定到低层 NIC 驱动程序时出现的接收数据包，接着，ProtocolReceive 应尽可能快地返回控制权给低层驱动程序，否则，紧接着的接收数据包就可能丢失。就像稍后将要讲的，面向无连接协议驱动程序要么在 ProtocolReceivePackets 函数中以包描述符（指定完整网络数据包）从低层 NIC 驱动程序接收内入数据，要么向 ProtocolReceive 函数指示接收数据，这样就必须将指示数据复制到缓冲区（链到协议分配的包描述符）。而面向连接协议驱动程序总是用 ProtocolCoReceivePackets 函数接收内入数据。

每一个面向无连接的协议驱动程序都必须提供 ProtocolReceive 函数。当该函数被调用时，协议驱动程序必须将所指示的前视数据复制到协议驱动程序分配的已链到预分配包描述符的缓冲区，该包描述符将传给 NdisTransferData 函数。如果除了指示的前视数据之外还有其他接收数据，以上调用将向低层驱动程序传递包描述符。

如果协议驱动程序只是绑定到用 NdisM(Co)IndicateReceivePackets 指示包数组的低层 NDIS 驱动程序，那么 Protocol(Co)ReceivePacket 函数不必为内入数据提供包描述符、缓冲区和缓冲区描述符。当协议驱动程序向 Protocol(Co)ReceivePacket 函数指示完整数据包时，

那么可以赋予客户对内入数据包描述符描述的缓冲数据直接只读访问能力，直到该数据处理完毕并释放了其所指定的包描述符及所有的相关资源为止。当协议驱动程序获得了该数据包的所有权后就不必再将这些数据复制到预分配包，也不必再调用 NdisTransferData 函数获取数据包的剩余部分。

如果协议驱动程序向 NdisTransferData、NdisSend(Packets)或者 NdisCoSendPackets 函数提供链向多个缓冲区的包描述符，并且最末的一个缓冲区的实际数据长度小于分配的缓冲区空间，那么协议驱动程序应该调用 NdisAdjustBufferLength 函数在缓冲区描述符中设置实际数据长度。当包描述符返回给协议驱动程序时，驱动程序应重新将缓冲区描述符指定的长度调整到完整缓冲区的大小。

重用数据包

当 Ndis(Co)Send 返回除 NDIS_STATUS_PENDING 之外的任何值或者当驱动程序的 Ndis(Co)SendComplete 被调用时，协议驱动程序分配的用于发送的包资源的所有权将返还给协议驱动程序。然后协议驱动程序可以随后的发送请求，重定义这些返回的包资源，或者如果其是无连接的驱动程序，可以重定义这些资源，来复制 ProtocolReceive 接收的数据包。

比起释放那些返还资源并在随后的发送或者数据传输操作中重新分配资源，协议驱动程序重新初始化并重用这些包描述符和缓冲区将是一种更为有效的方式。如果保存这些没被链接的缓冲区描述符和缓冲区以供将来需要时重用，而不是释放并重新分配相应的资源，那么协议驱动程序通常能够展现更好的性能。

协议驱动程序通过调用 NdisReinitializePacket 函数重新初始化包描述符。首先，协议驱动程序应该确定：已经通过调用 NdisUnchainBufferAtXxx 释放缓冲描述符及其映射的缓冲区的方式，删除了所有的链接缓冲区及它们的缓冲描述符，换句话说，NdisReinitializePacket 将设置所有指向链接缓冲区的成员为 NULL，所以如果没有预先释放存储链接缓冲区，那么重新初始化包描述符将导致内存泄漏。同样地，如果协议驱动程序和低层驱动程序使用带外信息，那么与每一个包描述符相关的 OOB 数据块所指定的资源在调用 NdisReinitializePacket 函数之前也必须被收回。

2.3 协议驱动程序的动态绑定

支持即插即用功能的协议驱动程序通过提供 ProtocolBindAdapter 函数和 ProtocolUnbindAdapter 函数就可以支持对低层 NIC 的动态绑定。

如果驱动程序向 NDIS 注册了这些函数，那么将可以延迟打开和绑定低层 NIC 而不必在 DriverEntry 函数中实现该功能，只要用 ProtocolBindAdapter 函数就可代替执行该操作。如果协议驱动程序提供了这些函数，那么只要低层 NIC 可用，NDIS 将调用能够将自己绑定到该适配器的任何协议驱动程序的 ProtocolBindAdapter 函数。并且只要低层 NIC 被关闭，那么 NDIS 将可以调用互逆的 ProtocolUnbindAdapter 函数。

ProtocolBindAdapter 函数定义如下

```
VOID
ProtocolBindAdapter(
OUT PNDIS_STATUS Status,
IN NDIS_HANDLE BindContext,
IN PNDIS_STRING DeviceName,
IN PVOID SystemSpecific1,
IN PVOID SystemSpecific2
```

);

NDIS 通过 *DeviceName* 向 ProtocolBindAdapter 函数提供最近可用的适配器名。NDIS 在 *BindContext* 中为绑定请求传递代表其环境的句柄。协议驱动程序必须保存该句柄并在驱动程序完成绑定相关的操作，并准备传输和接收数据包时将该句柄作为 NdisCompleteBindAdapter 的参数传递给 NDIS。

绑定时的操作包括为绑定分配适配器相关的环境空间并初始化，以 *DeviceName* 为参数调用 NdisOpenAdapter 打开适配器。*DeviceName* 可以指低层 NIC，或者也可以是介于协议驱动程序和 NIC 驱动程序之间，由 NDIS 中间层驱动程序导出的管理控制传输请求的适配器的虚拟 NIC。例如，NDIS 中间层驱动程序在旧版本协议驱动程序支持的介质格式和低层 NIC 驱动程序支持的介质格式之间进行转换。*BindContext* 值应该存储在协议驱动程序分配的环境区域或者其他类似的位置。

因为 NdisOpenAdapter 调用可能返回 NDIS_STATUS_PEDDING，所以 *BindContext* 值必须保存。如果 NdisOpenAdapter 调用返回 NDIS_STATUS_PEDDING，那么直到适配器打开操作完成并且协议驱动程序的 ProtocolOpenAdapterComplete 函数已被调用之后，协议驱动程序才能调用 NdisCompleteBindAdapter 函数。在这种情况下，*BindContext* 必须从某个已知位置获取并传给 NdisCompleteBindAdapter 函数。

SystemSpecific1 是一个必须传给 NdisOpenProtocolConfiguration 函数的字符串。该字符串对协议驱动程序是不透明的。

SystemSpecific2 预留系统使用。

关于动态绑定的更多信息请参阅 2.8 节。

2.4 协议驱动程序接收数据

这一节将讨论面向无连接以及面向连接的协议驱动程序是如何接收数据的。

2.4.1 下边界面向无连接的中间层驱动程序接收数据

低层面向无连接的 NIC 驱动程序可以下面两种方式指示数据包:

- NIC 驱动程序调用 NdisMIndicateReceivePacket 函数，传递指向完整数据的包描述符数组指针并且向高层驱动程序（在稍后能够处理数据并能返还包资源）转让这些数据包资源的所有权。
- NIC 驱动程序调用过滤相关的 NdisMXxxIndicateReceive 函数，传递前视缓冲区指针、该缓冲区尺寸及其接收数据包的总尺寸值。

每一个面向无连接的协议驱动程序必须至少提供这两个接收句柄中的一个，同时也可能提供如下的两个函数：

1. ProtocolReceive 函数是必须提供的。该函数通过前视缓冲区指针调用。如果 ProtocolReceive 函数检查前视数据并认为该数据包是客户程序所需要的，那么必须将该数据复制到协议驱动程序分配的可能链到包描述符的缓冲区。如果前视缓冲区大小小于接收数据包的总尺寸，那么 ProtocolReceive 函数必须以该包描述符为参数调用 NdisTransferData 函数，获取低层驱动程序复制到协议驱动程序提供的缓冲区中的接收数据包的其余部分。
2. ProtocolReceivePacket 函数是可选的，接收描述完整网络数据包的包描述符指针。ProtocolReceive 函数也检查该数据包，并确定该数据包是否是客户程序所需要的。如果该数据包客户程序所需要的，那么协议驱动程序能够以从 ProtocolReceivePacket 函数返

回一个非零值（协议驱动程序向其转发接收指示的客户数量）的方式，赋予客户程序对被指示包资源的所有权，包括对缓冲区网络数据的直接只读访问能力。当然，以后协议驱动程序的客户必须向低层驱动程序返还包描述符及其所指定的资源。对于该接收指示，除非所有客户程序的调用从 ProtocolReceivePacket 函数返回的值总计非零，否则每个客户都必须返回该包描述符。

当协议驱动程序的客户程序调用返回包描述符达到要求的次数之后，将向最初指示接收数据包的低层 NIC 驱动程序转让包资源的所有权。

另一方面，如果协议驱动程序从 NdisReturnPackets 函数中返回零值，那么将立即释放数据包及相关资源。例如，如果对于那个数据包，连接已关闭或者处于其他的某种不可用方式，或者如果在向客户指示之前已将指示数据复制到自己的缓冲区，并在内部处理了数据，那么立即释放该接收数据包的情况将出现。

2.4.1.1 在中间层驱动程序中实现 ProtocolReceivePacket 处理程序

当低层面向无连接的 NIC 驱动程序通过调用 NdisMIndicateReceivePacket 函数指示包描述符数组时，NDIS 通常将以每一个包描述符为参数调用绑定的协议驱动程序的 ProtocolReceivePacket 函数，从而让协议驱动程序（或者其客户）保存包描述符及其所有的资源，直到协议驱动程序或者其客户处理完数据并返回包描述符。典型的以包数组为参数调用 NdisMIndicateReceivePacket 函数的两类面向无连接 NIC 驱动程序如下：

1. 管理能够一次接收多个网络数据包到缓冲环的 DMA 总线控制适配器的 NIC 驱动程序；
2. 在与包描述符相关的 NDIS_PACKET_OOB_DATA 块中提供包含介质西相关信息的带外数据，像包优先级等的 NIC 驱动程序。

当然，一个驱动程序并不要求必须是 DMA 总线控制设备驱动程序。

如果协议驱动程序意识到被绑定（或者可能被绑定）到上面所说的 NIC 驱动程序上，那么它就应该提供 ProtocolReceivePacket 函数。这使得协议驱动程序可以完成接下的所有操作：

- 从低层 NIC 驱动程序接收完整数据包的指示；
- 用 NDIS 提供的宏，读取与包描述符相关的 OOB 数据;
- 保持对内入数据包描述符的所有权和对这些包描述符指定的缓冲数据的直接读访问能力，或者甚至在选择了对客户重要的包数据的范围之后向客户转发所指示的包描述符；然后，在对每一个包描述符的处理中，可能要为客户程序制作该数据的多个副本；
- 如果协议驱动程序保存着内入数据包描述符的所有权，那么当协议驱动程序处理完指示数据之后，通过 NdisReturnPackets 函数返回包描述符及所描述资源，另外可能还有其他被保存的包描述符。换句话说，即要客户（协议驱动程序向其转发接收指示）返回包描述符。

即使协议驱动程序提供了 ProtocolReceivePacket 处理程序，NIC 驱动程序对 NdisMIndicateReceivePacket 函数的调用也可能导致对协议驱动程序 ProtocolReceivePacket 函数的调用。因为 NIC 驱动程序调用 NdisMIndicateReceivePacket 函数时暂时释放了驱动程序分配资源的所有权，所以低层驱动程序将依赖于这些数据包的使用者及时返还该部分资源。换句话说，NIC 驱动程序能够撇开接收资源运行，像 NIC 中的接收缓冲空间等。NIC 驱动程序通过向与传给 NdisMIndicateReceivePacket 的包数组中的包描述符相关的 OOB 数据块写入 NDIS_STATUS_RESOURCES 状态标识，来完成该项功能，该状态指示的数据包将导致 NDIS 以该包及数组中相继的其他包为参数，调用高层驱动程序的 ProtocolReceive 函数，这将强制协议驱动程序立即复制包数据而不是保持 NIC 驱动程序分配的这些包资源。

如果协议驱动程序想要通过调用 ProtocolReceive 函数来获取与包描述符关联的 OOB 数据，那么它必须调用 NdisGetReceivePacket 及 NDIS_GET_ORIGINAL_PACKET 复制介质相

关信息到中间层驱动程序分配的缓冲区，另外，如果低层 NIC 驱动程序提供了时间戳，那么还必须复制 TimeSent 和 TimeReceived 信息。

2.4.1.2 在协议驱动程序中实现 ProtocolReceive 处理程序

如果低层面向无连接的 NIC 驱动程序调用了与过滤相关的 NdisMXxxIndicateReceive 函数，那么驱动程序将总要调用每一个绑定协议驱动程序的 ProtocolReceive 函数。

如果协议驱动程序的一个或者多个客户成为数据包的传送目标，而其前视缓冲区尺寸小于该包总尺寸，那么 ProtocolReceive 就必须完成以下操作：

1. 复制前视缓冲区数据到协议驱动程序分配的包描述符映射的内部缓冲区；
2. 将映射到足够容纳网络包数据的其余部分的缓冲区（由协议驱动程序分配的）的缓冲区描述符链接到协议驱动程序分配的包描述符；
3. 以包描述符为参数调用 NdisTransferData 函数，使低层驱动程序复制接收数据包的其余部分到协议驱动程序缓冲区。

当 NdisTransferData 函数返回 STATUS_SUCCESS 或者 ProtocolTransferDataComplete 函数被调用时，前视缓冲区可以被链接包含该传输数据的包描述符，并指示给任何对其感兴趣的客户。

对每一个接收指示，NdisTransferData 仅能被调用一次。协议驱动程序应创建包含链式缓冲区的包描述符，该缓冲区应能够容纳完整网络数据包。当 NdisTransferData 返回之后，从低层 NIC 驱动程序接收来的数据将不再有用。

传给 ProtocolReceive 函数的前视缓冲区长度是用 OID_GEN_CURRENT_LOOKAHEAD 调用 NdisRequest 的返回尺寸和数据包总长度中较小的一个。对于 ProtocolReceive 函数来说，所有的前视缓冲区中的数据都是只读的。

如果前视缓冲区长度大于或者等于与过滤相关的 NdisMXxxIndicateReceive 函数指示的数据包总尺寸，那么 ProtocolReceive 应该调用 NdisTransferData 函数将前视缓冲数据复制到内部缓冲区。

如果对 ProtocolReceive 函数的调用是由于在调用 NdisMIndicateReceivePacket 之前，低层 NIC 驱动程序设置了包数组中的一个或多个包状态为 NDIS_STATUS_RESOURCES，那么前视缓冲区的尺寸将总是等于整个网络数据包的大小。在这种情况下，由于 ProtocolReceive 可以立即复制完整指示到内部缓冲区，所以协议驱动程序不会再调用 NdisTransferData 函数。

如果在响应 OID_GEN_MAC_OPTIONS 请求时，NDIS_MAC_OPTIONS_COPY_LOOKAHEAD_DATA 被置位，那么协议驱动程序可以使用任何方法将前视数据复制到内部缓冲区，如 NdisMoveMemory 等。如果以上标识没有被置位，那么协议驱动程序就必须调用 NdisCopyLookaheadData 函数来复制指示数据，否则复制操作的结果将是不确定的。

ProtocolReceive 函数必须尽可能快地执行。协议驱动程序必须确保在接收内入指示之前有可用的由其分配的包资源。在协议驱动程序检查数据包确定其不是要复制的数据包之后，ProtocolReceive 函数应立即返回 NDIS_STATUS_NOT_ACCEPT 标识。

在复制接收包时，ProtocolReceive 函数不能处理接收数据，因为这将严重影响系统性能以及低层 NIC 接收导入数据包的能力。作为替代，中间层驱动程序在以后的 ProtocolReceiveComplete 函数中对接收数据包进行处理，该函数在随后能够进行数据包后期处理时被调用。典型地，当低层 NIC 驱动程序已经接收并指示了微端口确定的所有数据包时，或在其退出接收处理程序之前，以上操作将发生。协议驱动程序必须对 ProtocolReceive 接收的数据包进行排队，以使 ProtocolReceiveComplete 函数能够过它们进行后期处理。

2.4.1.3 从面向无连接协议驱动程序中访问 OOB 数据信息

当接收到的网络数据包指示给 ProtocolReceive 函数时，将强制驱动程序复制接收数据到协议驱动程序提供的缓冲区。如果该指示在包描述符相关的数据包 OOB 数据中包含介质相关信息和（或）时间戳信息，那么协议驱动程序可以调用 NdisGetReceivePacket 和 NDIS_GET_ORIGINAL_PACKET 获取介质相关信息以及 TimeSent 和 TimeReceived 时间戳。

如果接收数据包传给 ProtocolReceivePacket 函数，那么协议驱动程序可以以如下的方式使用 NDIS 提供的宏，保存相关的 OOB 数据信息：

- 用 NDIS_GET_MEDIA_SPECIFIC_INFO 复制介质相关信息;
- 用 NDIS_GET_TIME_SENT 复制 TimeSent 信息;
- 用 NDIS_GET_TIME_RECEIVED 复制 TimeReceived 信息。

TimeSent 时间戳是远程节点 NIC 发送数据包的时间，并且如果可能的话，它将被本地节点的低层 NIC 驱动程序获取并保存。TimeReceived 时间戳是内入数据包被本地节点的低层 NIC 驱动程序接收的时间。

如果协议驱动程序将以上时间戳转换为另一种格式，那么当前系统时间可以通过 NdisGetCurrentSystemTime 或者 KeQuerySystemTime 函数确定。

2.4.2 面向连接协议驱动程序接收数据

低层面向连接的 NIC 驱动程序通过调用 NdisMIndicateReceivePacket 指示数据包，传递参数为指向完整数据包的包描述符指针数组的指针，向在稍后能够处理数据并返还包资源的上层驱动程序转让这些数据包资源的所有权。

面向连接的协议驱动程序通过调用 ProtocolCoReceivePacket 函数接收指示数据包。该函数接收指定缓冲的完整网络数据包的包描述符指针。

ProtocolCoReceivePacket 函数也检查该数据包，并确定该数据包是否是客户程序所需要的。如果该数据包是客户程序所需要的，那么协议驱动程序能够以从 ProtocolCoReceivePacket 函数返回一个非零值（协议驱动程序向其转发接收指示的客户数量）的方式赋予客户程序对被指示包资源的所有权，包括对缓冲区网络数据的直接只读访问能力。当然，以后协议驱动程序的客户必须向低层驱动程序返还包描述符及其所指定的资源。对于该接收指示，除非所有客户程序的调用从 ProtocolCoReceivePacket 函数返回的值总计非零，否则每个客户都必须返回该包描述符。

当协议驱动程序的客户程序调用返回包描述符达到要求的次数之后，将向最初指示接收数据包的低层 NIC 驱动程序转让包资源的所有权。

另一方面，如果协议驱动程序从 ProtocolCoReceivePacket 函数中返回零值，那么将立即释放数据包及相关资源。例如，如果对于那个数据包，连接已关闭或者处于其他的某种不可用方式，或者如果在向客户指示之前已将指示数据复制到自己的缓冲区并在内部处理了数据，那么将立即释放该接收数据包。

2.4.2.1 ProtocolCoReceivePacket 处理程序实现

当低层面向连接的 NIC 驱动程序通过调用 NdisMCoIndicateReceivePacket 函数指示包数组时，NDIS 以每一个包描述符为参数，调用绑定的协议驱动程序的 ProtocolCoReceivePacket 函数。为了获得相关包的 OOB 数据块状态，ProtocolCoReceivePacket 必须对每一个包描述符调用一次 NDIS_GET_PACKET_STATUS 宏。

如果 NIC 驱动程序在调用 NdisMCoIndicateReceivePacket 之前，将数据包描述符相关的 OOB 数据块状态设为 NDIS_ STATUS_SUCCESS，NIC 驱动程序将暂时释放该包描述符相

关的资源的所有权。在这种情况下，NIC 驱动程序将依赖于这些包的使用者及时返还该部分资源。换句话说，NIC 驱动程序能够撇开接收资源运行，像 NIC 中的接收缓冲空间等。

当撇开这些资源运行时，NIC 驱动程序设置包描述符相关的 OOB 数据块状态为 NDIS_STATUS_RESOURCES，该状态指示的数据包将强制协议驱动程序的 ProtocolCoReceivePacket 函数立即复制包数据而不是保持 NIC 驱动程序分配的包资源。在这种情况下，ProtocolCoReceivePacket 必须返回零值。

如果低层 NIC 驱动程序没有将 OOB 数据块（与传给 NdisMCoIndicateReceivePacket 的包数组中的包描述符相关）置为 NDIS_STATUS_RESOURCES，那么将允许协议驱动程序保持该包描述符及其指定的资源，直到协议驱动程序或者其客户程序处理完数据并返回包描述符为止。

2.4.2.2 从面向连接协议驱动程序中访问 OOB 数据信息

当接收到的网络数据包指示给 ProtocolReceive 函数时，将强制驱动程序复制接收数据到协议驱动程序提供的缓冲区。如果该指示在包描述符相关的数据包 OOB 数据中包含介质相关信息和（或）时间戳信息，那么协议驱动程序可以调用 NdisGetReceivePacket 和 NDIS_GET_ORIGINAL_PACKET 获取介质相关信息以及 TimeSent 和 TimeReceived 时间戳。

面向连接协议驱动程序可以以如下的方式使用 NDIS 提供的宏，复制指示接收包相关的 OOB 数据块信息：

- 用 NDIS_GET_MEDIA_SPECIFIC_INFO 复制介质相关信息;
- 用 NDIS_GET_TIME_SENT 复制 TimeSent 信息;
- 用 NDIS_GET_TIME_RECEIVED 复制 TimeReceived 信息。

TimeSent 时间戳是远程节点 NIC 发送数据包的时间，并且如果可能的话，它将被本地节点的低层 NIC 驱动程序获取并保存。TimeReceived 时间戳是内入数据包被本地节点的低层 NIC 驱动程序接收的时间。

如果协议驱动程序将以上时间戳转换为另一种格式，那么当前系统时间可以通过 NdisGetCurrentSystemTime 函数确定。

面向连接微端口能够以 NIC 本地时间指示数据包 TimeReceived 时间戳。为了查询面向连接微端口或者 MCM 的本地定时能力，协议驱动程序通过 OID_GEN_CO_TIME_CAPS 调用 NdisCoRequest 函数。为了获得 NIC 的本地时间，协议驱动程序通过 OID_GEN_CO_GET_NETCARD_TIME 调用 NdisCoRequest 函数。

2.5 发送协议驱动程序创建的数据包

这一节来讨论从面向无连接协议驱动程序和面向连接驱动程序如何发送数据包。

2.5.1 从面向无连接协议驱动程序发送数据包

面向无连接协议驱动程序可以通过传递包描述符指针来调用 NdisSend 函数传输单个数据包，该包描述符指向链式缓冲区描述符（映射要发送缓冲数据）。作为一种替代方案，面向无连接协议驱动程序也可以通过传递指向包描述符指针数组的指针来调用 NdisSendPackets 函数传输多个数据包。

通常，面向无连接协议驱动程序开发者应根据驱动程序的需求和低层 NIC 驱动程序的特征，选择是调用 NdisSend 还是 NdisSendPackets 函数进行数据包传输。

当绑定到低层 NIC 驱动程序时，面向无连接协议驱动程序应通过

OID_GEN_MAXIMUM_SEND_PACKETS 调用 NdisRequest，获取低层驱动程序能够接受的发送包数组的最大包数量。如果 NIC 驱动程序仅支持单个包的发送，不管是通过 NdisSend 函数还是 NdisSendPackets 函数，那么以上调用的返回值将是‘1’或者 NDIS_STATUS_NOT_SUPPORTED，这两个返回之中的任一个都可能表示协议驱动程序将要调用的是很可能是 NdisSend 而不是 NdisSendPackets 函数。如果低层驱动程序返回大于‘1’的值，那么在发送包数组时使用 NdisSendPackets 函数会使两种驱动程序的性能都变得更好。如果 OOB 数据在协议驱动程序和 NIC 驱动程序之间传递，那么该两个发送函数都可以被调用，因为在任何一种情况下，低层驱动程序都能使用 NDIS 提供的宏读取相关的 OOB 数据。

当协议驱动程序调用 NdisSend 函数时，将以同步或异步方式释放给定包资源的所有权直到发送操作完成为止。如果 NdisSend 返回状态不是 NDIS_STATUS_PENDING，则调用以同步方式完成，并且将包资源的所有权返还协议驱动程序。如果 NdisSend 返回状态是 NDIS_STATUS_PENDING，则当调用完成时，发送操作的最终状态和协议驱动程序分配的包描述符将传给 NdisSendComplete 函数。

当协议驱动程序通过调用 NdisSendPackets 发送一个或多个数据包时，发送操作总是异步的。协议驱动程序释放其分配的包资源的所有权，直到每一个包描述符和对该包发送操作的最终状态都返回 ProtocolCoSendComplete 为止。

作为一个结论，在 NdisSend(Packets)函数调用返回时，协议驱动程序不能读取包描述符相关的 OOB 数据块的 Status 成员。协议驱动程序不能够以这种方式知道发送请求的状态，因为该成员被 NDIS 用于跟踪转换中的发送请求的进程，在包描述符返回给 ProtocolSendComplete 函数之前它是易变的。协议驱动程序要么通过检查 NdisSend 函数的返回值，要么通过传给 Protocol(Co)SendComplete 函数的 Status 参数来获取传送请求的状态信息。

如果在发送之前，中间层驱动程序通过重组从客户程序接收的数据包，请求不同优先级的包数组传送，那么应将最高优先级的数据包放在数组的开始位置。NDIS 将保持这些包数组（将被传递给 NdisSendPackets 函数）中的数据包的顺序，即使在内部要对一些数据包进行排队。

NDIS 不能企图对与传给 NdisSendPackets 函数（或者 NdisSend 函数）的包描述符相关的任何 OOB 数据块进行排队或检查。除非协议驱动程序对 NIC 驱动程序处理包优先级或 TimeToSend 时间戳的方式有特别的了解，否则，应假定 NIC 驱动程序按接收到的顺序发送数据包，保持接收时的顺序。因此，协议驱动程序应该根据数据包将在网络上传输的顺序，对将要发送的包数组进行排序。

2.5.1.1 面向无连接协议驱动程序传递介质相关信息

在发送数据包之前，协议驱动程序能够调用 NdisSetPacketsFlags 函数在包描述符的 NDIS 私有部分设置协议规定的标识。该标识并不是 NDIS 定义的，而是协议和低层 NIC 驱动程序合作定义的。NDIS_PACKET 的私有（Private）成员的结构对所有的 NDIS 驱动程序都是可访问的，允许使用 NDIS 提供的宏或函数对其进行读访问，在某些情况下，也可进行写访问。

协议驱动程序可以通过 OOB 数据块（与每一个 NDIS_PACKET 类型描述符相关的）传送更多的介质相关信息。以下是 OOB 数据块的定义：

```
typedef struct _ NDIS_PACKET_OOB_DATA{
    union{
        ULONGLONG TimeToSend;
        ULONGLONG TimeSend;
```

```
        };
    ULONGLONG TimeReceived;
    UINT      HeaderSize;
    UINT      SizeMediaSpecificInfo;
    PVOID     MediaSpecificInformation;
    NDIS_STATUS      Status;
} NDIS_PACKET_OOB_DATA,  * PNDIS_PACKET_OOB_DATA;
```

驱动程序缓冲区中的单个记录结构 MediaSpecificInformation 定义如下：

```
typedef struct MediaSpecificInformation{
    UINT      NextEntryOffset;
    NDIS_CLASS_ID  ClassId;
    UINT      Size;
    UCHAR  ClassInformation[1];
}MEDIA_SPECIFIC_INFORMATION;
```

ClassId 成员是 NDIS 定义的枚举变量（ClassInformation[1]中发现的信息类型）。目前，支持 win32 的微软操作系统中为无连接介质提供了三个 Class Id：NdisClass802_3Priority、NdisClassWirelessWanMbxMailbox 和 NdisClassIrdaPacketInfo。关于更详细的信息请参阅在线“*the Network Drivers Reference*”。

如果协议驱动程序知道要发送数据包的低层 NIC 驱动程序使用 OOB 数据，就能够设定相应的 OOB 结构成员：

- 通过使用 NDIS_SET_PACKET_TIME_TO_SEND 宏设置 TimeToSend 成员，请求数据包在特定的时间发送。该宏在系统时间单元中传递请求时间。协议驱动程序可以调用 NdisGetCurrentSystemTime 获取用于计算请求发送时间的系统时间；
- 使用 NDIS_ PACKET_SET_ MEDIA_SPECIFIC_INFO 宏设置该成员为缓冲区地址，在 MediaSpecificInformation 中传递协议缓冲区中的介质相关信息。例如，如果协议驱动程序绑定到要求优先级的低层 NIC，那么可以设置 MediaSpecificInformation 结构的 ClassId 成员为 NdisClass802_3Priority，并通过 ClassInformation 传递优先级相关信息以及通过 Size 传递该信息的字符大小值。协议驱动程序应该分配缓冲区来保存任何介质相关数据记录，也应该通过 MediaSpecificInformation 设置该缓冲区的指针。

2.5.2 面向连接协议驱动程序发送数据包

面向连接协议驱动程序可以通过向 NdisCoSendPackets 传送指向包描述符指针数组的指针的方式进行函数调用，传送一个或者多个数据包。

当协议驱动程序通过调用 NdisCoSendPackets 发送一个或多个数据包时，发送操作总是异步的。协议驱动程序释放其分配的包资源的所有权，直到每一个包描述符和对该报发送操作的最终状态都返回 ProtocolCoSendComplete 为止。

作为一个结论，在 NdisCoSendPackets 函数调用返回时，协议驱动程序不能读取包描述符相关的 OOB 数据块的 Status 成员。协议驱动程序不能够以这种方式知道发送请求的状态，因为该成员被 NDIS 用于跟踪转换中的发送请求的进程，在包描述符返回给 ProtocolCoSendComplete 函数之前它是易变的。协议驱动程序总是通过传给 ProtocolCoSendComplete 函数的 Status 参数来获取传送请求的状态信息。

如果在发送之前，中间层驱动程序通过重组从客户程序接收的数据包，请求不同优先级的包数组传送，那么应将最高优先级的数据包放在数组的开始位置。NDIS 将保持这些将要

传给 NdisCoSendPackets 函数的包数组中的数据包的顺序，即使在内部要对一些数据包进行排队。

NDIS 不能企图对与传给 NdisCoSendPackets 函数（或者 NdisSend 函数）的包描述符相关的任何 OOB 数据块进行排队或检查。除非协议驱动程序对 NIC 驱动程序处理包优先级或 TimeToSend 时间戳的方式有特别的了解，否则，应假定 NIC 驱动程序按接收到的顺序发送数据包，保持接收时的顺序。因此，协议驱动程序应该根据数据包将在网络上传输的顺序对将要发送的包数组进行排序。

2.5.2.1 面向连接协议驱动程序传递介质相关信息

在发送数据包之前，协议驱动程序能够调用 NdisSetPacketsFlags 函数在包描述符的 NDIS 私有部分设置协议规定的标识。该标识并不是 NDIS 定义的，而是协议和低层 NIC 驱动程序合作定义的。NDIS_PACKET 的私有（Private）成员的结构对所有的 NDIS 驱动程序都是可访问的，允许使用 NDIS 提供的宏或函数对其进行读访问，在某些情况下，也可进行写访问。

协议驱动程序可以通过与每一个 NDIS_PACKET 类型描述符相关的 OOB 数据块，传送更多的介质相关信息。以下是 OOB 数据块的定义：

```
typedef struct _ NDIS_PACKET_OOB_DATA{
    union{
        ULONGLONG TimeToSend;
        ULONGLONG TimeSend;
        };
    ULONGLONG TimeReceived;
    UINT      HeaderSize;
    UINT      SizeMediaSpecificInfo;
    PVOID     MediaSpecificInformation;
    NDIS_STATUS      Status;
 } NDIS_PACKET_OOB_DATA,   * PNDIS_PACKET_OOB_DATA;
```

驱动程序缓冲区中的单个记录结构 MediaSpecificInformation 定义如下：

```
typedef struct MediaSpecificInformation{
    UINT      NextEntryOffset;
    NDIS_CLASS_ID  ClassId;
    UINT      Size;
    UCHAR  ClassInformation[1];
}MEDIA_SPECIFIC_INFORMATION;
```

ClassId 成员是 NDIS 定义的枚举变量（ClassInformation[1]中发现的信息类型）。目前，支持 win32 的微软操作系统中提供了一个 Class Id：NdisClassAtmAALInfo。关于更详细的信息请参阅在线“*the Network Drivers Reference*”。

如果协议驱动程序知道要发送数据包的低层 NIC 驱动程序使用 OOB 数据，就能够设定相应的 OOB 结构成员：

- 通过使用 NDIS_SET_PACKET_TIME_TO_SEND 宏设置 TimeToSend 成员，请求数据包在特定的时间发送。该宏在系统时间单元中传递请求时间。协议驱动程序可以调用 NdisGetCurrentSystemTime 获取用于计算请求发送时间的系统时间；
- 使用 NDIS_ PACKET_SET_ MEDIA_SPECIFIC_INFO 宏设置该成员为缓冲区地址，在 MediaSpecificInformation 中传递协议缓冲区中的介质相关信息。

面向连接协议驱动程序能够用 NIC 本地时间调度发送任务并标记发送包时间。为了查询面向连接微端口或者 MCM 的本地定时能力，协议驱动程序通过 OID_GEN_CO_TIME_CAPS 调用 NdisCoRequest 函数。为了获得 NIC 的本地时间，协议驱动程序通过 OID_GEN_CO_GET_NETCARD_TIME 调用 NdisCoRequest 函数。

2.6 处理协议驱动程序的 PnP 事件和 PM 事件

当操作系统向代表 NIC 的目标设备对象发出即插即用 IRP 或电源管理 IRP 时，NDIS 将截取该 IRP，然后通过调用驱动程序的 ProtocolPnPEvent 处理程序向每一个绑定的协议驱动程序和中间层驱动程序指示该事件。在对 ProtocolPnPEvent 的调用中，NDIS 传递描述被指示的 PnP 事件或 PM 事件的 NET_PNP_EVENT 结构的指针。

就像 NET_PNP_EVENT 结构中 NetEvent 节点所指示的，有六个可能的 PnP 和 PM 事件：

- NetEventSetPower

  指示电源设置请求，该请求指定 NIC 过渡到特定电源状态。一个支持电源管理的协议驱动程序应该总是通过返回 NDIS_STATUS_SUCCESS 说明该事件成功处理。早期的协议驱动程序可以通过 NDIS_STATUS_NOT_SUPPORT 指示 NDIS 应该解除到 NIC 的绑定。

- NetEventQueryPower

  指示电源查询请求，该请求查询 NIC 能否过渡到特定电源状态。协议驱动程序应该总是能够成功执行 NetEventQueryPower。在建立一个有效连接之后，协议驱动程序可以调用 PoRegisterSystemState 注册一个持续忙状态，只要状态注册还有效，电源管理器就不会企图让系统进入睡眠状态。当连接不可用时，协议驱动程序可以调用 PoUnregisterSystemState 撤消状态注册。协议驱动程序不应通过使 NetEventQueryRemoveDevice 失败的方式阻止系统过渡到睡眠状态。注意 NetEventQueryPower 之后总是要调用 NetEventSetPower。对设备当前电源状态的 NetEventSetPower 调用实际上相当于撤销了 NetEventQueryPower。

- NetEventQueryRemoveDevice

  指示一个删除设备查询请求，该请求查询 NIC 能否在不中断操作的情况下删除 NIC。如果中间层驱动程序不能释放设备（例如，因为设备正在使用），那么必须以返回 NDIS_STATUS_FAILURE 的方式使 NetEventQueryRemoveDevice 操作失败。

- NetEventCancelRemoveDevice

  指示撤销删除设备请求，该请求撤销 NIC 删除操作。协议驱动程序应该总是通过返回 NDIS_STATUS_SUCCESS 说明该事件成功处理。

- NetEventReconfigure

  指示网络部件的配置已经改变。例如，如果用户改变了 TCP/IP 的 IP 地址，NDIS 用 NetEventReconfigure 向 TCP/IP 协议指示该事件。协议驱动程序应该总是通过返回 NDIS_STATUS_SUCCESS 说明成功完成该事件。

- NetEventBindList

  向客户指示绑定列表已经改变。绑定列表是传输协议导出的客户程序可绑定的一个或多个设备的列表。

- NetEventBindComplete

  指示协议驱动程序已经绑定到所有可以绑定的 NIC 上。除非有即插即用 NIC

装入系统，否则 NDIS 不会向协议驱动程序指示更多的 NIC。

NET_PNP_EVENT 的 Buffer 成员指向包含被指示事件特定信息的缓冲区。更多的信息请参看 NET_PNP_EVENT。

协议驱动程序能够以 NdisCompletePnPEvent 异步地完成 ProtocolPnPEvent 调用。

2.7 协议驱动程序复位操作

协议驱动程序可以通过调用 NdisReset 的方式启动复位操作。如果复位请求返回 NDIS_STATUS_PENDING，那么当复位操作完成时，ProtocolResetComplete 函数将被调用。

除非协议驱动程序知道低层 NIC 不能正常运行，否则很少调用 NdisReset。例如，如果协议驱动程序发现对于相当数量的发送或请求没有收到完成调用，而且如果有足够的关于低层 NIC 的知识断定有硬件问题发生，那么就可以调用 NdisReset 启动复位操作。然而，通常情况下，NIC 复位要求的检测和启动是通过 NDIS 和 NIC 驱动程序使用超时逻辑来完成的。任何绑定到以 NdisMediumWan 方式报告介质类型的低层 NDIS 驱动程序的协议驱动程序，都不能调用 NdisReset 函数。

典型地，低层驱动程序复位 NIC 是因为发送或请求操作中 NIC 超时，这导致 NDIS 调用 NIC 驱动程序的 MiniportCheckForHang 函数，并随后调用其 MiniportReset 函数，或者因为 NIC 驱动程序确定 NIC 接收能力异常。如果 NDIS 启动了复位操作并且 NdisReset 函数返回 NDIS_STATUS_PENDING，那么首先 NDIS 将以 NDIS_STATUS_RESET_START 状态调用每一个绑定协议驱动程序的 Protocol(Co)Status 函数，然后调用相同绑定驱动程序的 ProtocolStatusComplete 函数。当 NIC 驱动程序调用 ProtocolResetComplete 函数时， NDIS 将以 NDIS_STATUS_RESET_END 状态再一次调用 Protocol(Co)Status 函数并接着调用 ProtocolStatusComplete 函数。

协议驱动程序必须处理未完成的发送任务（在对低层 NIC 的绑定上）因为 NIC 复位而被撤销的情况。如果绑定的协议驱动程序有任何未完成的发送数据包，NDIS 将向协议驱动程序返回这些包的准确状态并结束这些数据包。当复位操作完成时，假设 NIC 恢复可用状态，那么协议驱动程序必须重新发送这些数据包。

当中间层驱动程序收到 NDIS_STATUS_RESET_START 状态时，应进行以下操作：

- 挂起发送就绪的数据包直到 Protocol(Co)Status 收到 NDIS_STATUS_RESET_END 通知;
- 不做任何对低层微端口操作的 NDIS 调用，除了那些返回资源的调用，像 NdisReturnPackets 返回数据包调用。

在 Protocol(Co)Status 收到 NDIS_STATUS_RESET_END 并且 ProtocolStatusComplete 函数已被调用之后，协议驱动程序可以继续发送数据包和处理请求。

2.8 协议驱动程序拆除绑定操作

协议驱动程序可以通过调用 NdisCloseAdapter 函数拆除对低层 NIC 驱动程序的绑定。如果协议驱动程序注册了 NdisUnbindAdapter 函数，那么当低层 NIC 不再可用时，NDIS 将调用该函数。例如，如果低层 NIC 超时，复位 NIC 的尝试不能使其正常工作，并且 NDIS 为故障 NIC 调用了 NIC 驱动程序的 MiniportHalt 函数，那么接下来 NDIS 将调用上层协议驱动程序的 NdisUnbindAdapter 函数。

如果协议驱动程序没有注册该函数，作为在 GeneralStatus 中接收到 NDIS_STATUS_CLOSING 状态值的结果，拆除绑定操作也可由驱动程序的

Protocol(Co)Status 函数产生。

中间层驱动程序也能发起解除绑定的操作。例如，当在 ProtocolStatus 中接收到的 GeneralStatus 状态值为 NDIS_STATUS_CLOSING 时。如果初始化期间某个操作并发地使用适配器，例如分配所要求的资源失败，那么协议驱动程序应该拆除绑定，因为该绑定上已不能执行网络操作。

如果协议驱动程序注册了 NdisUnbindAdapter 函数，那么该函数将要调用 NdisCloseAdapter 函数。无论如何，如果协议驱动程序调用了 NdisCompleteUnbindAdapter 函数，那么必须释放所有与绑定相关的资源。如果协议驱动程序没有提供 NdisUnbindAdapter 函数，但调用了 NdisCloseAdapter 函数，那么当 NdisCloseAdapter 函数返回 NDIS_STATUS_SUCCESS 或者如果 NdisCloseAdapter 函数返回了 NDIS_STATUS_PENDING，当调用 NdisCloseAdapterComplete 函数时，可以释放以前分配的资源。

NDIS 推迟调用 NdisCloseAdapterComplete 函数，直到绑定上的所有未处理请求都完成为止。一旦关闭请求完成，协议驱动程序分配的代表绑定的 ProtocolBindingContext 句柄将无效。

在同步地从 NdisUnbindAdapter 返回之前或者用 NdisCompleteUnbindAdapter 异步地完成解除绑定操作之前，中间层驱动程序必须完成以下操作：

- 清除其为绑定保存的任何状态值；
- 释放其分配用于建立绑定的所有资源；
- 调用 NdisCloseAdapter。

2.9 协议驱动程序状态指示

面向无连接协议驱动程序被要求提供 ProtocolStatus 和 ProtocolStatusComplete 函数。面向连接协议驱动程序被要求提供 ProtocolCoStatus 和 ProtocolCoStatusComplete 函数。

当低层面向无连接的 NDIS 驱动程序调用 NdisMIndicateStatus 报告 NIC 上的改变时，NDIS 将调用 ProtocolStatus 函数。当低层面向连接的 NDIS 驱动程序调用 NdisMCoIndicateStatus 时，NDIS 将调用 ProtocolCoStatus 函数。

当状态改变开始的时候，Protocol(Co)Status 被调用。如果当 ProtocolStatus 函数或者面向无连接协议被调用时，状态信息指示的操作没有完成，在这种情况下低层面向无连接 NIC 驱动程序紧接着将调用 NdisMIndicateStatusComplete 函数，那么调用 ProtocolStatusComplete 执行状态改变所指示的任何延期操作。

被送给 ProtocolStatus 的 GeneralStatus 实例包括：

- NDIS_STATUS_CLOSING

  在 2.8 节已经讨论了该状态和 Protocol(Co)Status 的操作;
- NDIS_STATUS_RESET_START 和 NDIS_STATUS_RESET_END

  就像 2.7 节中所说的，这两个状态都将送给 Protocol(Co)Status 和 ProtocolStatusComplete 函数；
- NDIS_STATUS_LINE_UP

  该状态表明协议驱动程序是基于已经与远程节点建立连接的支持 WAN 功能的 NIC 驱动程序的。协议驱动程序交换活动连接的句柄，在该连接上，可以进行数据发送和接收，也可查询连接获取相关呼叫信息。关于 WAN 驱动程序的更多信息请参阅第二部分第八章；

- NDIS_STATUS_RING_STATUS 或者其他介质相关的状态值

  对于该状态 StatusBuffer 提供了更详细的信息，例如关于令牌环介质的问题。

第三章　TDI 传输器及其客户

总的来说，这一章主要描述 TDI 传输驱动程序和 TDI 核心模式客户之间的联系，如使用系统定义的 TDI 接口和传输协议栈通信的高层网络接口仿真器、转向器和服务器。即本章将提供对以下问题的综述：

- 传输驱动程序接口（TDI）
- 支持即插即用的 TDI 传输器如何使用代表传输器和 NIC 绑定的设备对象，以及它们如何用 TDI 通过这些绑定动态注册客户可用的网络地址
- TDI 如何使用代表网络实体的文档对象，包括以下的三种基本 TDI 实体：
  1. 传输地址
  2. 连接端点
  3. 控制信道
- TDI 要求的传输驱动程序调度例程集合
- TDI 客户和其低层传输器间的交互
- TDI 请求和事件之间的区别

3.1 传输驱动程序接口（TDI）

图 3.1 展示了 TDI 客户和传输驱动程序之间的总体关系。

图 3.1　TDI 客户和传输提供者

如图中所示，传输驱动程序接口（TDI）定义了在传输协议栈上边界提供的核心模式网络接口。上述每一个栈中的最高层协议驱动程序支持为更高层核心模式网络客户提供的 TDI 接口。

该接口包括以下内容：

- 一组由每个传输驱动程序导出的标准核心模式中间层驱动程序的调度例程，客户通过调用支撑例程向传输驱动程序提交 I/O 请求（IRP），如 Zw..File 例程和(或)IoCallDriver；
- 一组由每个 TDI 客户导出的 ClientEventXxx 回调例程，通过向低层传输驱动程序注册这些例程来接收特定的网络事件通知；
- 一组由 TDI 客户导出的 ClientPnPXxx 回调例程，通过向 TDI 注册这些例程来接收有关动态绑定、网络地址和电源状态改变等来自支持 PnP 的 Windows 2000 传输器的通知；
- 与 TDI 传输、ClientEventXxx 例程和 ClientPnPXxx 例程相关的参数、结构、IOCTL 和程序规则；
- 一组系统提供的 TdiXxx 函数，通过调用这些函数，传输器和客户进行相互通信；
- 一组系统提供的 TdiBuildXxx 宏和函数，客户用它们生成提交给低层传输器的 I/O 请求。

要求所有传输驱动程序提供单一公共接口（TDI）简化了传输驱动程序开发的任务，这样所有传输器只要支持这些预定的单一接口就行了。由于减少了必须编写的传输器相关的代码量，这也简化了客户开发的任务。

Windows 2000 提供了几种通用网络接口的接口模块，如 Windows Sockets 和 NetBIOS。

每个接口模块对外提供一组本地的原语函数，在用户代码中通过标准的调用就可以对其进行访问，接口模块将这些本地的原语函数及其相关参数和程序规则映射为对低层 TDI 传输驱动程序的一个或者多个服务的调用。

TDI 的主要特征包括：

■ 高粒度级别

因为在本质上有粒度的相关性，TDI 可以现存的通用网络接口为标准，用几个 TDI 定义的请求（它们可以进行混合和匹配调节现存的网络接口函数的映射），调整所有的原语和约定。

■ 异步操作

多数核心模式 TDI 操作是异步的，用客户提供的回调例程指示异步网络事件，异步地完成大部分客户发起的作为 IRP 提交的操作。

■ 32 位的地址和值

就像所有的 Windows 2000 核心模式组件一样，TDI 传输器和其客户是以 32 位编码的。TDI 定义的结构和参数使用 32 位指针和值。

■ 灵活的寻址模式

TDI 并不要求什么特定的地址格式，如以前操作系统定义的 16 字符的 NetBIOS 名称。相反其使用一种扩展的机制使得多种地址格式都能被识别和使用。

■ 扩展通信

TDI 定义了 TDI_ACTION_IOCTL 请求，通过使用它，TDI 传输器可以支持一组由传输器确定的由其客户的请求发起的操作。这允许客户向低层传输驱动程序（TDI 没有明确定义）提交特定传输请求。

■ 事件通知

TDI 定义了一种模式，当其在网络上出现的时候（不要求客户提交显式的 I/O 请求），通过它传输器能够向客户通知其感兴趣的事件。对于某些种类的事件，像连接、断连和接收事件等，当传输驱动程序通知客户时，可以将相关数据一起指示给客户。

■ 即插即用事件通知

TDI 定义了一种模式，通过它 Windows 2000 传输器能够通知其客户相关的 PnP 事件，如低层 NIC 的可用性以及基于本地网址的连接创建和删除。

■ 系统电源状态改变通知

TDI 定义了一种模式，通过它 Windows 2000 传输器能够通知客户其所提名的系统电源状态改变事件，这样使得客户有可能维持有用连接。

接下来的特征被 TDI 传输驱动程序作为附加选择支持：

■ 数据传输方式

TDI 支持以离散消息（消息模式）方式或者字节流（字节流模式）的方式发送和接收数据。支持哪一种模式（或者两种都支持）由传输驱动程序设计者和（或）协议的性质确定。

■ 内部缓冲

TDI 传输器能够在内部对其客户的发送和接收进行缓冲。传送其内部缓冲，从而允许 TDI 客户查询和设置驱动程序内部缓冲区的尺寸、请求非块式发送操作、接收可用缓冲区空间的通告和在接收之前浏览缓冲数据。

■ 管理选项

所有的传输器都维持着一些关于其各自的特征、限制和运行时统计信息的由 TDI 定义的状态。这允许每个客户动态地查询，在某些情况下也可设置传输提供者的静态信息、统计信息和配置参数。另外，前面提到的可扩展 TDI_ACTION IOCTL 允许 TDI 传输驱动程序实现独特的网络管理特性（可以被其客户通过对传输器的请求操作访问的）。

■ 服务质量（QoS）

TDI 传输器能够提供网络连接建立时的 QoS 协商。另外，该驱动程序也能对面向无连接的数据报传输提供 QoS 支持。为了提供上述 QoS 支持，TDI 定义的连接建立和数据报发送请求包括 Options 和 OptionsLength 参数，这使得 TDI 客户可以包括一个传输相关的可变长字符串来指定选项。

事实上，Options 和 OptionsLength 参数不仅仅可以指定 QoS，而且能够用于向传输驱动程序传递由驱动程序设计者决定的传输器相关连接或者有关数据报的选项。

■ 加速传送

当发送消息时，客户可以标识特定消息为要加速的。在发送端，低层传输器在非加速消息之前发送这些被标识的消息。在接收端，标识为加速的消息被提前和（或者）独立于非加速消息指示给客户

■ 链式接收指示

如果低层 NDIS 微端口支持多包接收指示，那么 TDI 传输器可以赋予其客户在单个调用中对完整 TSDU 的直接只读访问能力，并且客户能够保持包含 TSDU 的缓冲区的控制，直到其处理完指示数据。该特性通过降低接收指示的调用开销，提高了传输器和客户的性能。

3.2 TDI 设备对象

Windows 2000 传输器通过调用代表各自对低层 NIC 绑定的设备对象，支持即插即用和系统电源管理。每个那样的传输器在系统安装时作为一个网络提供者向 TDI 注册，这使得在低层网络硬件联机时，潜在客户可以使用即插即用传输器。

这些具有即插即用功能的传输器的潜在客户向 TDI 注册各自的 ClientPnPXxx 回调函数，来接收关于传输器对 NIC 活动绑定的可用性的通告和关于传输器建立的网络地址的通告，当该网络地址有效时，客户就可以在网络上发送和接收数据。

当传输器通过调用由传输栈中的最低层模块（或者整体式驱动程序）向 NDIS 注册的 ProtocolBindAdapter 函数创建对低层 NIC 的绑定时，TDI 传输驱动程序创建指定的设备对象，用来代表对低层 NDIS 微端口的绑定。在系统安装期间，支持 PnP 的传输器向 TDI 注册其创建的每个指定的设备对象，并设置每个绑定的状态。传输器也向 TDI 注册每个绑定上其已知的网络地址，并设置每个注册地址的状态。当 NDIS 在传输栈底部通过 PnP 事件代码 NetEventBindsComplete 调用 ProtocolPnPEvent 函数时，如果网络上至少有一个传输器-NIC 绑定准备好传送数据，那么接着该传输器将通知 TDI。

TDI 使得每个已经注册了 ClientPnPXxx 处理程序的网络客户有可能通过从注册客户的 ClientPnPBindingChange 例程调用 ZwCreateFile 绑定到传输器创建的设备对象。接着 TDI 对于每个网络地址（低层传输器已经为其设备对象注册的）一次或者多次调用 ClientPnPAddNetAddress 例程。当某个特定的传输器指示其准备好进行网络发送时，并且当所有安装时传输器-NIC 绑定都已建立时，TDI 将通过调用已注册的 ClientPnPBindingChange 例程通知其客户。

运行时，当动态绑定发生改变时，不管改变是由于 NIC 被启动或禁止，还是由于终端用户导致了绑定改变，或者当传输器像如下所述的那样创建或断开一个对远程节点网络地址的连接时，这些支持 PnP 的传输器（或者 NDIS）都将继续调用 TDI。

- 如果一个新的 NIC 被启动，NDIS 调用传输栈底部的 ProtocolBindAdapter 函数，使得传输器有可能绑定到新的 NIC 并创建新的设备对象代表其新的绑定，然后向 TDI 注册新的设备对象。当每个传输器创建的设备对象注册时，TDI 将调用最新注册的

ClientPnPBindingChange 例程并提供客户打开新的绑定的机会。

- 在注册新的设备对象后，传输器也将向 TDI 注册绑定上的已知的网络地址。当传输器注册新的绑定（或者已存在的绑定）上已知网络地址时，TDI 将调用 ClientPnPAddNetAddress 例程。该调用也使得客户开启对已注册地址（甚至绑定）的连接成为可能。当传输器断开对远程节点的连接时，将调用 TDI 注销相应的网络地址。
- 在现存的 NIC 被禁止之前，典型地，NDIS 将调用传输栈底部的 ProtocolPnPEvent 函数询问删除 NIC 是否安全。接着，该传输器通知 TDI 关于该删除 NIC 的请求，然后 TDI 通过调用已注册的 ClientPnPPowerChange 例程通知客户。如果没有客户反对 NIC 的删除，那么接下来 NDIS 将调用传输栈底部的 ProtocolUnbindAdapter 函数并且传输器将拆除该绑定。在这个过程中，传输器调用 TDI 注销以前正式注册的绑定上的所有网络地址，它也调用 TDI 注销以前创建的代表绑定的设备对象。按顺序，接下来 TDI 通过调用 ClientPnPDelAddress 例程和 ClientPnPBindingChange 例程通知客户关于这些拆除绑定操作。
- 如果终端客户促使网络连接文件夹中的绑定发生改变，NDIS 将直接调用 TDI，把相关绑定的改变通知注册网络客户。如果没有客户反对（即用户并不企图取消绑定改变操作），那么 NDIS 将选择传输栈底部 ProtocolPnPEvent、ProtocolBindAdapter 或者 ProtocolUnbindAdapter 中的适当函数进行相关的处理。

TDI 以相似的方式把低层 NIC 的电源状态改变通知给支持 PnP 的传输器的客户。当系统电源管理器调用 NDIS 来提供 NIC 电源或者切断 NIC 电源时，NDIS 将调用每个当前绑定该 NIC 的协议栈底部的 ProtocolPnPEvent 函数。接着，每个支持 PnP 的传输器通知 TDI 相关电源状态的改变，并且 TDI 将向传输器的客户转发该通知以征求赞成或拒绝意见。

3.3 TDI 文件对象

所有 I/O 请求都被直接指示给一个打开的文件对象，该文件对象代表一个物理的或虚拟的设备、一个数据文件或者任何 I/O 请求的逻辑目标，该文件对象能够通过特定步骤打开。打开文件对象是进程相关的。例如，I/O 管理器为每个打开特定数据文件的进程创建一个文件对象。

TDI 用文件对象代表所有网络通信环境中的实体。特别地，TDI 用文件对象代表如下 TDI 定义实体类型：

- 代表传输地址（address），该地址由网络节点上的特定进程或者一组进程打开；
- 代表连接端点，该端点标识与传输地址相关的特定端点，它被打开用于设置和远程节点的对等进程的端端连接；
- 代表控制信道，该信道标识传输服务提供者，打开它来查询和设置低层传输器维持的全局配置、特性、限制和（或）统计信息。

TDI 传输驱动程序和低层 NDIS 驱动程序提供了一种机制，使得数据可以从网络节点的一个进程传送到本节点或者其他网络节点上的一个或多个进程。该传输以打开文件对象所代表的前述地址和连接端点实体的方式出现。

例如，Windows 2000 转向器进程打开两个文件对象，来向远程节点上的服务器进程发送 SMB 消息和接收来自远程节点上服务器进程的返回消息，该两个文件对象其中一个代表传输地址，另一个代表传输地址相关的连接端点。

3.3.1 代表传输地址的文件对象

TDI 支持不可靠的面向无连接传输和可靠的面向连接传输两种数据传输。不可靠的面向无连接数据可被发送到远程节点上已经打开特定传输地址的一个或多个进程。当作为数据报发送不可靠的面向无连接数据时，发送者仅需要标识接收数据报的进程或者进程组的远程节点地址。

如果端端连接（也叫虚电路）在两个进程间已经建立，那么可靠的面向连接的数据就能够在该两个进程间传送。一个端端连接是二个且仅有两个进程之间的一对一联系。为了建立那样一个连接，一个进程必须标识其要建立连接的进程，每个进程都必须在各自的网络节点打开一个传输地址和一个连接端点，并用打开的传输地址联系连接端点。关于连接端点的更多信息请参阅 3.2.2 节。

用于标识特定进程或进程组的 TDI 实体是一个或多个打开的代表特定传输地址的进程相关文件对象。该文件对象包含传输器提供的指向驱动程序维持的标识特定进程和进程驻留节点状态的指针。这些具有路由能力的传输器，像 TCP/IP、Mcsxns、AppleTalk 和 NWLink，保存的关于那些地址的状态也包括指定节点所在的网络（或者子网）的信息。

TDI 定义的传输地址类型能够容纳关于地址标识单个进程（单值地址）还是一组进程（组地址）的显式的或者隐式的指示。在组地址情况下，TDI 定义的地址能够包含标识相关进程即进程驻留节点和节点所在网络（或者子网）的信息。

TDI 支持许多地址类型，接下来描述三种通用 TDI 地址类型的格式和用法：

- TDI_ADDRESS_NETBIOS

NetBIOS 类型地址包括标准 16 字符的 NetBIOS 名和所指示的成员。如果该名以单值名称注册，则指示该名标识单个进程；或者如果该名作为组名注册，则指示该名标识一组进程。

因为传输驱动程序能够确保每次网络上仅有一个进程使用单值名，所以该名称不仅能够标识进程，而且能够隐含地标识进程驻留节点。

如果被注册的是一个组名，那么该地址对多个终端的进程都是可用的，这样该组名标识了注册该名的所有进程以及进程驻留节点。

- TDI_ADDRESS_IP

IP 类型地址包含一个端口号和标准的 IP 地址。由于 TCP/IP 允许相同的端口号可被多个节点上的进程注册使用，所以要求 IP 地址能够独立地标识节点并且要求端口号能够独立地标识该节点上的进程。

对于无连接数据传输（使用 TCP/IP 的 UDP 协议），相同的端口号可以被同一节点上的多个进程注册使用。另外，某些 IP 地址也能被多个节点使用。送给 TDI 地址的包含端口号和 IP 地址的数据可以被该 IP 地址标识的所有节点接收，并且这些数据将被送给注册指定端口号的所有进程。

- TDI_ADDRESS_IPX

IPX 类型地址包含四字节的网络号、六字节的节点号和两字节的端口号。因为 IPX 允许多个节点上进程注册相同的端口号，所以要求 IPX 地址能够独立地标识网络和节点，而端口号用来独立地标识该节点上的进程。

3.3.2 代表连接端点的文件对象

代表打开连接端点的文件对象标识一个特定的连接，在该连接上，本地和远程节点的对等进程通过网络相互进行通信，或者当本地节点进程建立和远程节点对等进程的端端连接之后在二者之间进行通信。

以上所述的任何文件对象必须建立与其他打开的代表特定传输地址的文件对象之间的联系。3.2.1 节已经讨论过，传输地址可以标识进程。

相似地，传输器使用代表本地连接端点的文件对象，保存关于远程节点对等进程的状态，

像远程节点传输地址，本地节点客户可以建立到它的端端连接。

客户进程能够和远程节点进程建立多个端端连接。例如，转向器和远程 Windows 2000 服务器建立独立的连接进行相互通信。这样客户就能够拥有多个打开且相互独立的与打开的传输地址相关的连接端点。

当某个进程想在端端连接上发送数据时，必须指定数据要发送到现存的哪一个连接上。每一个打开的代表连接端点的文件对象用于区分同一个进程已经建立的多个端端连接。

3.3.3 代表控制信道的文件对象

网络管理进程能够查询和设置与特定传输提供者相关的全局配置或统计信息。为了进行上述查询和设置，该进程必须在多个可能的传输服务提供者之间指定特定的传输器。

为了查询相应的传输器，客户应该打开代表控制信道的文件对象。在实际运行中，当客户打开控制信道时，客户将通过调用 ZwCreateFile 或者 IoGetDeviceObjectPointer，打开代表低层传输驱动程序创建的设备对象的文件对象。

通过网络发送和接收数据的其他 TDI 客户也能够打开控制信道，对低层传输器进行查询。例如，客户能够生成一个查询，确定传输器对数据报长度的限制，使得客户可以有效地分配用于发送和接收数据报的缓冲区大小。

总之，TDI 传输器维持的是关于其特征和最新的统计值的全局状态信息，而不是依赖于特定打开文件对象的进程相关状态信息。例如，客户传递指向代表该地址的特定客户文件对象的指针，来查询其打开的传输地址的最新状态，传输器返回当前所有打开相同地址的客户的公共信息。

3.4 TDI 传输驱动程序例程

每个 TDI 兼容的传输驱动程序都必须是一个导出许多可以被 I/O 管理器调用的入口点的标准中间层驱动程序。

在 TDI 传输器的标准驱动程序例程中，一部分用于完成初始化和卸载驱动程序的工作，其他的是标准的调度例程，当 TDI 客户调用系统支持例程时，像 ZwCreateFile 或者 IoCallDriver，I/O 管理器可以调用这些调度例程。

像其他的核心模式驱动程序一样，TDI 传输器的 DriverEntry 例程设置一个或多个调度例程来处理各种类型的作为 IRP 传入的 I/O 请求。TDI 驱动程序能够导出单个调度例程处理所有引入的 IRP，或者导出独立的 DispatchXxx 例程处理驱动程序支持的 IRP_MJ_XXX。调度例程的一般要求的详细讨论可参阅 Kernel-mode Driver Guide。调度例程的 TDI 相关要求在第四章将进行汇总，其详细描述也可参阅在线 DDK 的“*the Network Drivers Reference*”。

当 TDI 完成其客户的请求操作时，I/O 管理器调用客户提供的 IoComplete 例程，其在该客户向低层传输器提交该请求之前在 IRP 中设置。

另外，当特定网络事件出现时，上述传输驱动程序必须在 TDI 客户代码中的预注册入口点调用 TDI 客户。这些由客户提供的事件处理程序也将在第四章进行汇总，其详细描述也可参阅在线 DDK 的“*the Network Drivers Reference*”。

在下边界，整体式的 TDI 驱动程序必须导出一组 ProtocolXxx 函数供代表低层 NDIS 中间层驱动程序和（或）NIC 驱动程序的 NDIS 库调用。这些 NDIS 驱动程序下边界函数前面已在该本书中进行了描述，也可参阅在线 DDK 的“*the Network Drivers Reference*”中的函数相关方法。

3.5 TDI 核心模式客户交互

下图显示了 TDI 客户是如何向其低层 TDI 传输器发送 I/O 请求的，以及传输器是怎样回调其客户的。

图 3.2　TDI 客户/传输器交互

TDI 客户以以下方式和其低层传输驱动程序进行交互：

- 创建 TDI 文件对象

客户调用 ZwCreateFile 创建或打开代表传输地址、连接端点或者控制信道的文件对象。该调用使得 I/O 管理器分配 IRP，将客户提供的参数整理到 IRP，向低层传输驱动程序的 TdiDispatchCreate 例程传送 IRP。当传输驱动程序为新创建的文件对象设置了其维护的所有状态之后，将以 IRP 和 STATUS_SUCCESS 调用 IoCompleteRequest（或者 TdiCompleteRequest）。IoCompleteRequest 将向 TDI 客户返回文件对象句柄。

每个客户进程对 ZwCreateFile 的调用将创建一个独立的文件对象，即使两个客户在对 ZwCreateFile 的调用中已指定了相同的传输地址。

对 ZwCreateFile 的成功调用依赖于调用中客户传递的 EaXxx 参数打开传输地址、连接端点或者控制信道。

- 提交请求

文件对象创建之后，客户就可以提交引用这些对象的请求。例如，在打开代表特定传输地址的文件对象之后，该客户就可以提交地址信息查询或者从该地址发送数据报的请求。

上述客户用标准 I/O 系统机制和约定提交如下请求：

- 客户用 IRP_MJ_XXX 操作码（指定客户希望传输驱动程序执行的操作）设置 IRP。客户为给定的 IRP_MJ_XXX 设置所有相关参数，并且作为一种选择也可设置 IoComplete 例程，该例程在传输器完成请求时被调用。

  Windows 2000 DDK 包含一组 TdiBuildXxx 宏（在 tdikrnl.h 中），该宏可被链入客户代码，也可用于为 TDI 定义的 IOCTL 请求设置 IRP，另外，也提供了用于分配 IRP 的 TdiBuildInternetDeviceControl 函数。
- 当 IRP 设置好后，客户用指向 IRP 的指针、指向代表地址的文件对象、连接端点或者控制信道的指针、以及指向传输驱动程序设备对象的指针调用 IoCallDriver。I/O 管理器直接向传输驱动程序的适当 TdiDispatch（Xxx）例程传递该 IRP。
- 当传输驱动程序完成请求操作之后，将调用 TdiCompleteRequest 或者 IoCompleteRequest。然后 I/O 管理器调用客户提供的 IoCompletion 例程，如果对该 IRP 客户提供了这个例程的话。

- 处理事件通知

如果客户已预先为一个或多个事件处理程序注册了入口点，那么当相应网络事件发生时，传输驱动程序将调用相关例程。例如，客户在端端连接相关的地址上注册了 ClientEventReceive 处理程序，那么当远程节点对等进程发送的数据被本地节点接收时，传输器将调用该处理程序。

- 删除 TDI 对象

当地址、连接端点或者控制进程不再为客户需要时，该客户调用 ZwClose 删除相关的文件对象。删除请求被转发给传输器的 TdiDispatchCleanUp，接着转发给 TdiDispatchClose 例程。

关于以上所提的核心模式支持例程的更多信息，请参阅在线 DDK 的“*the Kernel-Mode*

Drivers Reference”。

3.6 TDI 请求及事件

许多低层网络接口，如 NetBIOS 和 Windows Sockets，主要都是单行道的。客户在任何时候都可以调用低层传输驱动程序，反之，传输驱动程序就不能调用其客户。传输驱动程序调用其客户的唯一方式是通过从客户发起的请求返回特定错误码。例如，在 NetBIOS 接口中，传输器通过随特定的错误代码（例如，对会话关闭用 0x0A）一起返回未决的 NCB_Receive (或者 NCB_Receive_Any)通知其客户关于连接挂起信息。

TDI 提供一种事件通知机制，在特定网络事件（例如，数据报的接收）发生时，允许 TDI 传输器调用其受到影响的核心模式客户，只要每个客户为该类型事件都已向传输器注册了 ClientEventXxx 处理程序。然后 TDI 客户提供的回调执行相应的操作，并将控制返回传输驱动程序。

TDI 也为支持 PnP 的 Windows 2000 传输器定义了一种方式，当该传输器绑定到低层 NIC 或拆除绑定时，当传输器在特定绑定上建立或断开到远程节点的连接时，或者当电源管理器改变（和/或影响）系统电源状态时，允许传输器通知其客户。就像在 3.2 节中所说的，该传输器的客户向 TDI 注册一组 ClientPnPXxx 处理程序来接收这些种类的通知。

当 TDI 传输驱动程序调用客户注册的 ClientEventXxx 处理程序时，能够传递一定数量的数据作为该调用的参数。该特征使得客户不必分配缓冲区就可接收来自传输器的消息。

例如，转向器就利用了该 TDI 特征。转向器向服务器发送的许多 SMB 请求，仅要求用标准 SMB 报头存放相应的 SMB 响应信息，该报头很小，典型的只有 100 字节，报头存放信息包括状态指示器、多路复用 ID 等。

当低层 TDI 传输驱动程序用先前发送的 SMB 响应信息调用转向器注册的 ClientEventReceive 处理程序时，该转向器仅仅需要察看（不必复制）指示消息，记录 SMB 响应的状态指示器并返回传输驱动程序。在该处理中，转向器不必分配缓冲区就可接收 SMB 响应信息。

第四章　TDI 例程、宏和回调

本章讨论 TDI 传输驱动程序的例程和请求和 TDI 核心模式客户导出的回调例程，以及系统提供给传输器和客户使用的宏和 TDI 函数。

以下主题是对每种信息的汇总：

- 传输驱动程序

4.1、 TDI 驱动程序初始化

4.2、 TDI 驱动程序调度例程

- TDI 定义的核心模式客户能够向其低层传输器发布的一组 IOCTL

4.3、 TDI IOCTL 请求

- 客户能够向低层传输器注册的事件处理程序

4.4、TDI 客户回调

- TDI 传输器和其客户能够使用的由系统提供的一组函数和宏

4.5、TDI 库函数和宏

关于 TDI 客户和传输期间的交互序列（当使用前面所述的宏、函数和例程时）的更多信息请参阅第五章。

要想获取关于每个请求、宏或例程的详细信息，请参阅在线 DDK 的“*the Network Drivers Reference*”。

4.1 TDI 驱动程序初始化

每个 TDI 传输驱动程序必须提供 DriverEntry 显式命名的初始化例程。另外，传输驱动程序应提供足够核心模式客户 I/O 请求需要的 TdiDispatchXxx 例程和内部驱动程序函数。每个 TDI 传输驱动程序或者传送栈中的某个低层协议驱动程序也必须提供前面第二章所描述的 NDIS 驱动程序下边界函数，关于其详细信息可参阅在线 DDK 的“*the Network Drivers Reference*”。

每个传输驱动程序必须提供在传输器载入时供 I/O 管理器调用的 DriverEntry 例程。DriverEntry 必须在驱动程序代码中声明，使得传输器能够自动装载。

当载入驱动程序时，I/O 管理器创建代表传输器的驱动程序对象，并在调用 TDI 传输器的 DriverEntry 例程时传递该驱动程序对象的指针。然后，DriverEntry 例程完成如下的工作：

- 如果可能，读取注册表获取传输器 INF 文件写入注册表的配置信息。根据需要，用注册表配置信息进行相关配置。
- 在驱动程序对象中设置驱动程序所有的 TdiDispatchXxx 入口点。以后，I/O 管理器处理 TDI 客户请求时将会调用这些 TdiDispatchXxx 例程。
- 调用 TdiRegisterProvider 通知 TDI 关于传输器的即插即用和电源管理支持能力。
- 使用 IoCreateDevice 为自己创建至少一个设备对象。每个驱动程序设备对象的名字就是存在注册表 Export 入口中的 Linkage 键下的设备名。每个传输驱动程序的 Export 入口决定了系统安装时每个特定的传输驱动程序创建的设备对象。

  有些传输驱动程序，像 NWLink，不管其绑定的 NIC 有多少，都只创建一个设备对象。有些传输驱动程序，像 TCP/IP，可以创建一个或者多个设备对象并且对每个设备对象导出一组设备接口。其他的传输驱动程序则为每个绑定的 NIC 创建一个独立的设备对象，如 Nbf_Elnkkii1，Nbf_Elnkkii2 等。

- 执行其他必须的初始化任务，像绑定到低层 NDIS 中间层驱动程序和（或）NIC 驱动程序。
- 当创建对每个低层 NIC 的绑定时，支持即插即用的 Windows 2000 传输器还要用传输器创建的设备对象调用 TdiRegisterDeviceObject。接着，一次或者多次调用 TdiRegisterNetAddress 注册所有已知的与设备对象（传输器为其最近建立的绑定创建的）相关的网络地址（包括本地和远程节点）。

当该支持 PnP 的传输器已至少完成了一个绑定的初始化，从而可以在网络上传输数据时，传输器必须调用 TdiProviderReady。典型地，作为 NDIS 在传输栈低边通过 NET_PNP_EVNET 编码 NetEventBindsComplete 调用 ProtocolPnPEvent 函数的结果，支持 PnP 的传输器调用 TdiProviderReady。

TDI 传输驱动程序必须提供卸载例程，除非系统可用时驱动程序不能卸载。多数 TDI 传输驱动程序声明了卸载例程并且 DriverEntry 例程在驱动程序对象中设置其入口点。关于驱动器卸载例程的更多信息请参阅 4.1.2 节。

关于 TDI 驱动程序所用的核心模式支持例程的更多信息，请参阅在线 DDK 的“*the Kernel-Mode Drivers Reference*”。

4.1.1 注册 TDI 传输驱动程序

当 TDI 载入时，配置记录获取每个 TDI 传输驱动程序和核心模式客户在特定网络上的操作入口。使用注册表中的绑定信息，操作系统导入网络驱动程序。

接着，I/O 管理器创建驱动程序对象并调用每个 TDI 传输驱动程序的 DriverEntry 例程直到网上所有 TDI 驱动程序都被载入并注册为止。

如果要绑定到低层 NDIS NIC 驱动程序，那么 TDI 传输驱动程序或者传送栈的低层协议组件就必须调用 NdisRegisterProtocol 向 NDIS 库注册。当 PnP 管理器调用 NDIS 初始化本地节点上的每个 NIC 时，NDIS 在相应 TDI 传输驱动程序或传送栈的下边界调用已注册的 ProtocolBindAdapter 函数，使得每个传输器都能够绑定到相应的 NDIS 微端口。

当 TDI 传输驱动程序绑定到 NDIS 微端口或者使自己位于已绑定到该微端口的 NDIS 协议驱动程序之上时，TDI 驱动程序应准备好响应已注册的 TDI 客户，该客户向 TDI 传输器已经向该绑定注册的网络地址提交请求。

最后，当 TDI 传输器（支持 PnP）为每个初始化时调用（TdiRegisterDeviceObject 和 TdiRegisterNetAddress）设置了各个设备对象状态和各个地址状态之后，必须调用 TdiProviderReady。

注意上述 TDI 传输器每次初始化时只调用 TdiRegisterProvider 和 TdiProviderReady 一次。然而，传输器可以对 Windows 2000 TDI 提供的 PnP 和电源管理函数进行多次运行时调用，例如，TdiRegister/DeregisterDeviceObject、TdiRegister/DeregisterNetAddress 和 TdiPnPPowerRequest 等。

关于 Windows 2000 PnP 和电源管理支持的更多信息，请参阅 *Plug and Play，Power Management，and Setup Design Guide* 以及 the Network Drivers Guide 中关于 NDIS 库如何指示 PnP 和电源管理事件的详细资料。

4.1.2 卸载和注销 TDI 传输驱动程序

在 NIC 从机器中删除之前，绑定到该 NIC 的 Windows 2000 TDI 传输器能够首先通知其客户：NIC 将被禁止或删除。即，除非意外删除 NIC，NDIS 将在每个绑定到该 NIC 的传输器的下边界用 NetEventQueryRemoveDevice 输入码，调用 ProtocolPnPEvent 函数，使得每个 TDI 传输器都能够通过调用 TdiPnPPowerRequest 向其客户转发该通知。

直到 NDIS 在传送栈下边界调用 ProtocolUnbindAdapter 函数，上述支持 PnP 的传输器才真正拆除绑定并调用 TdiDeregisterNetAddress 和 TdiDeregisterDeviceObject。而且，也直到释放了所有对低层 NIC 的绑定之后，传输驱动程序才被卸载。

不管怎样，当 NIC 从机器中删除之后，系统就可以卸载 TDI 传输驱动程序了，这些驱动程序要么直接绑定到上述低层 NIC 驱动程序，要么建立在绑定到 NIC 驱动程序的 NDIS 协议驱动程序之上。为了实现该功能，系统调用 TDI 传输器的卸载例程。

当服务控制器从内存卸载该驱动程序之后，I/O 管理器将调用传输器的卸载例程。传输驱动程序的卸载例程释放所有保留的驱动程序分配的资源。

支持 PnP 的 TDI 传输驱动程序必须在其卸载例程中进行互逆的 TdiDeregisterProvider 调用。如果 TDI 驱动程序导出了一组 NDIS 定义的 ProtocolXxx 函数，那么在其卸载例程中也必须调用 TdiDeregisterProtocol。

关于标准中间层驱动程序卸载例程的更多信息，请参阅 the Kernel-Mode Drivers Design Guide。关于 Windows 2000 PnP 和电源管理支持的更多信息，请参阅 *Plug and Play，Power Management，and Setup Design Guide* 以及 the Network Drivers Guide 中关于 NDIS 库如何指示 PnP 和电源管理事件的详细资料。

4.2 TDI 驱动程序调度例程

I/O 请求由 I/O 管理器或者由 TDI 客户显式地格式化为 IRP，并通过调用 IoCallDriver 提交给传输驱动程序。当传输驱动程序调用 IoCompleteRequest 或 TdiCompleteRequest 时，完成的 IRP 返回给呼叫器。在调用 IoCallDriver 之前，任何核心模式 TDI 客户都可以为 IRP 设置 IoCompletion 例程。

有五个 IRP_MJ_XXX 编码用于向 TDI 传输驱动程序发送 I/O 请求。传输器处理的引入 IRP，其中的 MajorFunctionCode 是以下之一：

- IRP_MJ_CREATE

打开一个传输器创建的指定的设备对象，接着打开低层传输器中的传输地址、连接端点和控制信道。

该请求由客户对 ZwCreateFile 的调用发起。

- IRP_MJ_INTERNEL_DEVICE_CONTROL

指定核心模式客户请求（TDI IOCTL），对于该请求，内部传输函数处理除打开和关闭文件对象之外的操作。

该请求通常由客户对 TdiBuildXxx 宏的调用发起，该宏之后紧接着调用 IoCallDriver。

- IRP_MJ_ DEVICE_CONTROL

指定用户可见模式 IOCTL 请求，该请求可由传输器专门应用发起。除了传输驱动程序定义的私有 IOCTL 之外，前述请求通常都能被转发给相同的处理内部设备控制请求的内部驱动程序函数。

该请求由传输器专门用户模式应用对 DeviceIoControl 的调用引起。

- IRP_MJ_CLEANUP

当 NT 执行程序关闭相应文件对象的最后一个句柄时，关闭打开的相应地址、连接端点和控制信道。

该请求由客户对 ZwClose 的调用引起。

- IRP_MJ_CLOSE

如果 NT 执行程序取消对文件对象句柄的最后的引用，关闭打开的相应地址、连接端点

和控制信道。当未完成的文件对象句柄都已关闭时，关闭传输驱动程序创建的设备对象。

在同一个文件对象上，该请求紧跟着 IRP_MJ_CLEANUP 请求发起。

TDI 驱动程序中的入口点是一个或者多个处理 IRP_MJ_XXX 请求的调度例程。因为 TDI 客户通过 IRP 和驱动程序进行通信，所以驱动程序要有一个或多个确定执行什么操作的 TdiDispatchXxx 例程。通常，为了更进一步的处理，这些 TdiDispatchXxx 例程向相应的内部驱动程序函数传递客户请求。

TDI 传输驱动程序通过在传给 DriverEntry 例程的驱动程序对象中进行设置，导出其所有 TdiDispatchXxx 例程的入口点。当客户提交 I/O 请求时，I/O 管理器调用 DriverEntry 例程。传输驱动程序能够用分离的 TdiDispatchXxx，处理每个可能的 IRP_MJ_XXX 操作码，用单个 TdiDispatchXxx 例程处理所有可能 IRP_MJ_XXX 操作码的 IRP，或者用一组 TdiDispatchXxx 处理 IRP_MJ_XXX 操作码的分立子集。

既然所有调度入口点都是通过驱动程序对象中的地址而不是名字导出的，因此 TDI 传输驱动程序对这些例程的命名是不受限制的。在该 TDI 文档中，相对于前述的 IRP_MJ_XXX ，TdiDispatchXxx 例程有如下的后缀名，每一个都描述了其基本的功能：

IRP_MJ_CREATE

TdiDispatchCreate

IRP_MJ_INTERNEL_DEVICE_CONTROL

TdiDispatchInternelDeviceControl

IRP_MJ_ DEVICE_CONTROL

TdiDispatchDeviceControl

IRP_MJ_CLEANUP

TdiDispatchCleanUp

IRP_MJ_CLOSE

TdiDispatchClose

4.3 TDI IOCTL 请求

前一节已经提到，核心模式客户通过调用 IoCallDriver 在 IRP 中向低层传输驱动程序传递 TDI IOCTL。这些请求通过 IRP_MJ_INTERNEL_DEVICE_CONTROL 的 MajorFunctionCode 和指定如下 TDI 定义的操作之一的 MinorFunctionCode 进行传递：

TDI_ACCEPT

允许本地节点客户接受来自远程节点对等实体的引入连接提议，使得能够在网络上发送和接收面向连接数据。

TDI_ACTION

为地址、连接端点或者控制信道启动传输相关的扩展操作。

TDI_ASSOCIATE_ADDRESS

用打开的本地传输地址联系开放连接端点，为了和远程节点对等实体建立端端连接这是必要的。

TDI_CONNECT

启动从一端到远程节点的另一端的连接提议。

TDI_DISASSOCIATE_ADDRESS

对一个无效连接，使连接端点与相连的地址对象分离。

TDI_DISCONNECT

为本地节点客户终止一个已建立的端对端连接，或者证实远程节点引发的连接中断。

TDI_LISTEN

被动地侦听远程节点对等进程的引入连接提议。

TDI_QUERY_INFORMATION

查询低层 TDI 传输器的特定类型信息。

TDI_RECEIVE

返回端端连接上接收的 TSDU 部分或全体。

TDI_RECEIVE_DATAGRAM

返回打开传输地址上的数据报 TSDU。

TDI_SEND

向端端连接上的远程节点对等实体发送标准的或者加速的 TSDU。

TDI_SEND_DATAGRAM

向指定的远程节点地址发送数据报。

TDI_SET_EVENT_HANDLER

注册 ClientEventXxx 处理程序。当相应网络事件发生时，传输器将调用该处理程序。

TDI_SET_INFORAMTION

在低层传输器中设置特定类型信息。

4.4 TDI 客户回调

多数 TDI 客户例程是易变的并且是依赖于环境的。然而，TDI 定义了一组由客户向低层 TDI 传输器注册的回调例程，用于在相关网络事件发生时接收通知。如果 TDI 驱动程序要通知客户关于所发生的特定类型网络事件，那么核心模式 TDI 客户能够注册上述的所有 ClientEventXxx 例程。作为一种选择，客户甚至能够使用注册的 ClientEventXxx 处理程序向驱动程序发布确定的 TDI_XXX IOCTL 请求。

就像前面所描述的 TdiDispatchXxx 例程一样，客户提供的事件处理程序任意选择其名称。这儿所说的每个 ClientEventXxx 注册为一个接口，该接口被传递给由客户提供的向 TdiDispatchInternelDeviceControl 例程提交的 IRP。

一个 TDI 客户可以有一个或者多个以下所列的回调例程，这些回调例程在打开传输地址上的网络操作开始时，通过发布连续的 TDI_SET_EVENT_HANDLER 请求，向 TDI 传输驱动程序注册，并用 TdiBuildSetEventHandler 进行设置。

ClientEventConnect

通知客户关于远程节点对等进程的连接提议。

ClientEventDisconnect

通知本地节点客户：远程节点对等实体正在终止它们之间的端端连接。

ClientEventError 和/或 ClientEventErrorEx

通知客户关于其低层传输驱动程序、传送栈的某个低层协议或者低层 NDIS 驱动程序的错误状态，这使得随后在客户打开传输地址上的 I/O 请求变得不可靠或者不可能。

ClientEventReceive

通知客户：传输器已从端端连接的远程节点对等实体上接收到标准数据，并且该接收数据的部分或全体已可被客户复制。

ClientEventChainReceive

通知客户：传输器已从端端连接的远程节点对等实体上接收到标准 TSDU，并且客户已

可处理该完整 TSDU。

ClientEventReceiveExpedited

通知客户：传输器已从端端连接的远程节点对等实体上接收到加速数据，并且该接收数据的部分或全体已可被客户复制。

ClientEventChainReceiveExpedited

通知客户：传输器已从端端连接的远程节点对等实体上接收到加速 TSDU，并且客户已可处理该完整 TSDU。

ClientEventReceiveDatagram

通知客户：TDI 驱动程序已在客户打开的传输地址上接收到数据报。

ClientEventChainReceiveDatagram

通知客户：TDI 驱动程序已在客户打开的传输地址上接收到数据报，并且客户已可处理该完整 TSDU。

ClientEventSendPossible

通知客户：传输器（在内部对发送数据进行缓冲）重新拥有发送缓冲空间。

Windows 2000 TDI 客户也可向 TDI 注册一组 ClientPnPXxx 回调例程来接收支持 PnP 的 TDI 传输器的动态 PnP 绑定改变、网络地址改变和电源状态改变事件的通知。就像 ClientPnPXxx 处理程序一样，这些客户提供的 PnP 事件处理程序也可以任意选择其名称。

该 Windows 2000 TDI 客户有以下所列的一组回调例程，这些回调例程在客户初始化是通过调用 TdiRegisterPnPHandlers 进行注册。

ClientPnPBindingChange

系统启动时，TDI 调用该例程通知客户所有可用的由传输器创建的、代表传输器对 NIC 绑定的设备对象。当每个支持 PnP 的 TDI 传输器指示其绑定已准备好进行网络数据传输时，再一次调用该例程。当所有可能的 TDI 传输器对 NIC 的绑定都已建立之后，也要调用该例程。

运行期间，当支持 PnP 的传输器注册另一个代表最近建立的传输器对 NIC 绑定的设备对象时，或者当注销现存设备对象时，TDI 将调用该客户例程。另外，如果终端用户使网络连接文件夹中的绑定列表顺序发生改变，也要调用该例程。

ClientPnPAddNetAddress

系统启动时，TDI 调用该例程通知客户其绑定上的所有当前网络地址，该网络地址由支持 PnP 的传输器向 TDI 注册。接着，当支持 PnP 的传输器在绑定上注册新的网络地址（通常是为新近建立的对远程节点的连接注册的）时，TDI 将调用该客户。当所有可能的 TDI 传输器对 NIC 的绑定都已建立之后，也要调用该例程。

ClientPnPDelNetAddress

当支持 PnP 的传输器从注册表中撤销绑定上的现存网络地址（通常在断开对远程节点连接时发生）时，TDI 将调用该例程。

ClientPnPPowerChange

TDI 调用该例程通知客户相关 NIC 的删除和相关系统电源状态的改变，这将影响特定绑定上的客户在网络上进行传输数据的能力。

典型地，向 TDI 注册这组 ClientPnPXxx 处理程序的客户将保存关于所有有效网络地址的动态状态，通常作为一个或多个内部列表，这些网络地址是依附于用于网上通信的所有 TDI 传输器的绑定的。

4.5 TDI 库函数和宏

Windows 2000 以核心模式动态连接库的方式提供了一组 TDI 库函数，通过这些库函数，TDI 驱动程序和核心模式客户可以进行链接。然而，系统提供的供核心模式客户调用的多数 TdiBuildXxx 是作为宏在 tdikrnl.h 头文件中实现的，该头文件可被客户和传输器包含。

就像 4.3 节已经提到的，任何核心模式 TDI 客户在准备 TDI_XXX IOCTL IRP 时，都可以根据需要使用 TdiBuildXxx 宏。当客户使用 TdiBuildXxx 宏设置好了该 IRP 之后，将通过向 IoCallDriver 传递 IRP 的方式，向低层 TDI 驱动程序提交其请求。这些宏将填充客户提供的 IRP 的相关成员，除了 Status 和 Information 成员之外，这两个成员在处理完客户请求之后由低层传输器填充。在使用 TdiBuildXxx IOCTL 宏之前，如果高层网络组件还没有分配并赋予客户相应的 IRP，客户能够调用 TdiBuildInternetDeviceControlIrp 分配 IRP。

传输驱动程序不能使用 TdiBuildXxx 宏。然而，在处理客户请求过程中，作为辅助，TDI 驱动程序能够使用其余的 TDI 库中的某些例程，例如 TdiCompleteRequest 和 TdiCopyBufferToMdl 等。

TDI 库例程和宏包括以下所列：

TdiBuildInternetDeviceControlIrp

分配 IRP，如果客户没有收到较高网络层的 IRP 的话。

TdiBuildAccept

为客户提交 TDI_ACCEPT 请求设置 IRP。

TdiBuildAction

为客户提交 TDI_ACTION 请求设置 IRP。

TdiBuildAssociateAddress

为客户提交 TDI_ASSOCIATE_ADDRESS 请求设置 IRP。

TdiBuildConnect

为客户提交 TDI_CONNECT 请求设置 IRP。

TdiBuildDisassociateAddress

为客户提交 TDI_DISASSOCIATE_ADDRESS 请求设置 IRP。

TdiBuildListen

为客户提交 TDI_LISTEN 请求设置 IRP。

TdiBuildQueryInformation

为客户提交 TDI_QUERY_INFORMAITON 请求设置 IRP。

TdiBuildReceive

为客户提交 TDI_RECEIVE 请求设置 IRP。

TdiBuildReceiveDatagram

为客户提交 TDI_RECEIVE_DATAGRAM 请求设置 IRP。

TdiBuildSend

为客户提交 TDI_SEND 请求设置 IRP。

TdiBuildSendDatagram

为客户提交 TDI_SEND_DATAGRAM 请求设置 IRP。

TdiBuildSetEventHandler

为客户提交 TDI_SET_EVENT_HANDLER 请求设置 IRP。

TdiBuildSetInformation

为客户提交 TDI_SET_INFORMATION 请求设置 IRP。

TdiBuildNetbiosAddress

为客户设置 NetBIOS 地址。

TdiBuildNetbiosAddressEa

设置缓冲的 NetBIOS 地址，客户接下来将其传递给 ZwCreateFile 来打开该地址。

TdiReturnChainReceives

在客户处理完接收的 TSDU 之后，释放传给 ClientEventChainReceive（Xxx）处理程序的缓冲区的控制权。

TdiCopyBufferToMdl

复制一定范围的缓冲数据到给定 MDL 映射的目标缓冲区。

客户和传输器都可使用该函数。

TdiCopyMdlToBuffer

从给定 MDL 映射的缓冲区复制数据到呼叫器提供的目标缓冲区。

客户和传输器都可使用该函数。

TdiCopyLookaheadData

安全地复制 NIC 驱动程序指示给传输器的接收数据，不管 NIC 正在使用的内存（包括映射的设备内存）的特征是什么。

TdiMapUserRequest

如果 TdiMapUserRequest 认可输入 IRP 中指定的局部函数代码，将传给传输器 DisPatchDeviceControl 例程的 IRP 转换为 TDI_XXX IOCTL IRP。

TdiCompleteRequest

以系统为传输驱动程序定义的网络相关优先权作为辅助完成 IRP。

TdiRegisterPnPHandler

向 TDI 注册一组 ClientPnPXxx 例程接收来自低层支持 PnP 的 TDI 传输器的 PnP 绑定改变、网络地址改变和电源状态改变事件的通知。

TdiDeregisterPnPHandler

注销一组 ClientPnPXxx 例程，该例程接收关于 PnP 绑定改变、网络地址改变和电源状态改变事件通知。

TdiRegisterProvider

系统启动时，向 TDI 注册支持 Windows 2000 PnP 的传输驱动程序。

TdiProviderReady

指示支持 PnP 的传输驱动程序准备好执行一个或多个新近建立的传输器-NIC 绑定上的网络 I/O。

TdiDeregisterProvider

卸载之前注销支持 PnP 的传输驱动程序。

TdiRegisterDeviceObject

注册一个指定的由传输器创建的设备对象，该支持 PnP 传输器的客户能够为网络 I/O 打开该设备对象。该支持 PnP 的传输器调用该 TDI 函数通知其潜在客户它刚刚建立了一个新的传输器-NIC 的绑定。

TdiDeregisterDeviceObject

当支持 PnP 的传输器拆除绑定时，注销传输器创建的设备对象。

TdiRegisterNetAddress

注册一个有效网络地址，该地址由支持 PnP 的传输器为本地机器创建或者是在特定绑定上建立对远程节点链接时获得的。支持 PnP 的传输器调用该 TDI 函数通知潜在客户：以后可使用该协议相关地址进行网络通信。

TdiDisregisterNetAddress

从特定的支持 PnP 的传输器-NIC 绑定的有效地址列表中，注销一个网络地址。

TdiPnPPowerRequest

向支持 PnP 的传输器客户转发确定的 NET_PNP_EVENT 类型通知。支持 PnP 的传输器调用该 TDI 函数，使其客户有可能通知它们的更高层客户传输器依赖的低层 NIC 的绑定发生改变或电源状态发生改变。

TdiPnPPowerComplete

指示客户已经完成了对绑定改变或电源状态改变通知（先前返回了 STATUS_PENDING）的处理。

TdiEnumerateAddresses

向调用客户返回所有有效网络地址的列表。该 TDI 函数很少被调用，因为该调用对调用程序的性能有严重的负面影响。

第五章　TDI 操作

本章讨论 TDI 运行时的基本操作。当支持 PnP 的 Windows 2000 传输器的客户开始这里所说的操作时，它们将使用当前已向 TDI 注册的协议相关网络地址，在网络上建立连接。关于支持 PnP 的传输器和其客户的更多信息，可参阅前面的第三章和第四章。

很多 TDI 运行操作都包括本地节点客户和远程节点客户之间的端端连接，这些 TDI 操作是面向连接的并且包括以下内容：

- 打开本地节点连接断点
- 请求连接
- 接受连接
- 发送面向连接数据
- 接收面向连接数据
- 断开连接
- 关闭本地节点连接断点

TDI 也允许网络节点间的无连接通信。这种类型的操作不要求本地节点客户建立和远程节点客户的端端连接。它们比面向连接通信速度更快但可靠性降低。TDI 无连接操作包括以下内容：

- 发送数据报
- 接收数据报

对面向连接和面向无连接通信，以下 TDI 操作是公用的：

- 打开本地节点传输地址，该地址用一个文件对象作为代表
- 打包和发送请求
- 设置和查询信息
- 接收错误通知
- 如果低层传输器对其客户支持扩展 TDI 接口，请求相关传输操作
- 关闭前面打开的传输地址

接下来这一节讨论 TDI 的主要操作，其内容将尽可能按这些操作可能出现的顺序安排。关于这里所提的 TDI、传输器和客户例程请参阅在线 DDK 的“*the Network Drivers Reference，Part 3*”。

关于这里所说的支撑例程，如 ZwCreateFile 和 IoCallDriver 等，请参阅在线 DDK 的“*the Kernel-Mode Drivers Reference*”。

5.1 打开传输地址

图 5.1 显示了核心模式客户是怎样在低层传输驱动程序中打开传输地址的。

图 5.1　　打开传输地址

当为绑定打开传输器创建的设备对象之后，TDI 客户通常通过打开代表传输地址的文件对象和本地节点传输器进行通信。为了实现该功能，客户调用 ZwCreateFile 并在其 EA（扩展属性）缓冲区参数中指定相应地址。在 EA 缓冲区中，客户设置 EaName 成员为系统定义的 TdiTransportAddress，其后紧跟的是 TDI 定义的 TRANSPORT_ADDRESS 类型 EA 值，该值由客户设置指定将被打开的相关传输地址。

当调用 ZwCreateFile 时，客户可以指定特定传输地址是可与其他客户共享的还是该客户自己独占使用的。当初始客户为独占使用成功打开特定传输地址之后，其他客户直到该客户关闭该地址时才能再次打开该地址。

为了在网上进行无连接通信，假设传输器支持广播数据报，客户能够在 EA 缓冲区中指定低层传输器的广播地址。任何客户都不能独占使用该地址。

客户对 ZwCreateFile 的调用促使 I/O 管理器创建代表该地址的客户进程相关文件对象，并用包含客户向 ZwCreateFile 提供的参数的 IRP，调用 TDI 传输驱动程序的 TdiDispatchCreate 例程。TdiDispatchCreate 对 EA 信息进行分析，并且如果调用成功，传输器将为该打开地址和客户设置相应的内部状态。

当 ZwCreateFile 调用成功并向客户返回一个文件句柄，并且该客户也通过 ObReferenceObjectHandler 获得了该文件句柄的指针之后，就能够向低层传输器提交确定的 TDI_XXX IOCTL 请求。例如，客户能够在该打开地址上发送和接收数据报。

通常，客户必须首先确定是否要注册 ClientEventXxx 处理程序，是否将使用该打开地址和远程节点对等进程进行通信，如果要进行通信，客户也必须打开一个连接断点。

如果客户想接收不同网络事件的通知，可以通过向低层传输器提交 TDI_SET_EVENT_HANDLER 请求的方式来为每类事件注册相关 ClientEventXxx 处理程序，该请求用 TdiBuildSetEventHandler 进行设置。关于设置 IOCTL 请求和向低层传输器提交该请求的更多信息请参阅 5.3 节。

当客户在打开的传输地址上注册了 ClientEventXxx 处理程序之后，就能够像 5.2 节所说的那样打开一个连接端点进行面向连接通信，也可像 5.7 节中将要讨论的进行无连接通信。

当打开的传输地址上的网络通信完成后，客户必须关闭代表该传输地址的文件对象，该操作通过传递 ZwCreateFile 返回句柄调用 ZwClose 来完成，后面 5.13 节有详细描述。

5.2 打开连接端点

图 5.2 显示了核心模式客户是怎样在低层传输驱动程序中打开连接端点的。

图 5.2　打开连接端点

如果本地节点客户请求和远程节点对等进程进行面向连接通信，那么本地节点客户必须首先在低层传输驱动程序中打开一个传输地址，并且接着打开一个连接端点。即，该客户要两次调用 ZwCreateFile，第一次打开一个地址，其次打开一个连接端点。

当打开连接端点时，客户在 EA（扩展属性）缓冲区参数中向 ZwCreateFile 传递客户提供的连接环境。在 EA 缓冲区中，客户设置 EaName 成员为系统定义的 TdiConnectionContext，其后紧跟的 EA 值通常是客户分配的环境区域地址，客户将在该区域保存要建立的端端连接的状态。

当 ZwCreateFile 调用成功并向客户返回一个文件句柄，并且该客户也通过 ObReferenceObjectHandler 获得了该文件句柄的指针之后，就能够向低层传输器提交确定的 TDI_XXX IOCTL 请求和远程节点对等实体建立端端连接。

首先，客户必须通过向低层传输器提交 TDI_ASSOCIATION_ADDRESS 请求的方式，在连接端点和其打开的传输地址之间建立联系，该请求用 TdiBuildAssociation 进行设置。

关于设置 TDI_XXX IOCTL IRP 的更多信息可参阅 5.3 节。关于如何建立端端连接可参阅 5.5 节。

当端端连接已经断开并且客户不再进行任何面向连接通信时，客户必须关闭连接端点，

该操作通过传递 ZwCreateFile 返回的文件句柄，调用 ZwClose 来完成，后面 5.12 节有详细描述。

5.3 打包并提交 IOCTL 请求

图 5.3 显示了 TDI 客户是如何设置并提交对其低层 TDI 传输器的 TDI_XXX IOCTL 请求的。

图 5.3　准备 IOCTL IRP 并产生请求

核心模式客户使用第四章讲到的 TdiBuildXxx 宏，为与低层传输驱动程序进行通信，准备其 IOCTL IRP。客户能够从网络高层获取 IRP，也可以调用 TdiBuildInternalDeviceControlIrp 为自己分配 IRP。

接下来调用 TdiBuildXxx 宏在 IRP 的 MinorFunctionCode 中设置合适的 TDI_XXX，并将 MajorFunctionCode 设为 IRP_MJ_INTERNEL_DEVICE_CONTROL ，另外，也要为每个 TdiBuildXxx 宏调用设置 IRP 中的客户提供的以及与 IOCTL 相关的其他参数。例如，为了建立前面提到的 TDI_ASSOCIATION_ADDRESS 请求，当使用 TdiBuildAccept 时，客户将要传递代表连接端点的文件对象指针和代表传输地址的文件对象句柄。

IRP 设置好后，客户调用 IoCallDriver 向其低层传输器提交该 IOCTL 请求。传输器的 TdiDispatchInternelDeviceControl 例程接收客户通过 IoCallDriver 提供的 IRP，通常，为了进行进一步的处理，还要向内部驱动程序函数转发该客户的请求。传输器处理请求操作，用操作结果设置 IRP 的 I/O 状态区，并在客户请求已被满足时用 IRP 调用 TdiCompleteRequest（或者 IoCompleteRequest）。

如果客户在调用 TdiBuildXxx 宏时提供了 IoComplete 例程的入口点，那么当传输器完成请求操作时，将调用客户的 IoComplete 例程并随同客户提供的 TDI_XXX IOCTL 请求一起调用 TdiCompleteRequest（或者 IoCompleteRequest）。

5.4 设置和查询信息

图 5.4 显示了 TDI 客户是如何查询其低层传输器的特征的，以及如何请求其低层传输器将其状态数据设为客户指定值的。

图 5.4　设置和查询传输器信息

TDI 客户能够查询传输驱动程序信息，像连接状态信息、特定传输地址的活动性、传输相关的数据报大小和数量限制、驱动程序统计数据以及内部发送接收缓冲区大小（如果传输器在内部对数据进行缓冲的话）。客户也能设置低层传输器的一些状态信息，尽管客户不能请求设置超出低层传输器确定的尺寸限制的值。

如果客户想要查询关于打开传输地址和连接端点的信息，可以向低层传输器提交 TDI_QUERY_INFORMAION 请求，该请求用 TdiBuildQueryInformation 进行设置。当使用该宏时，客户将传递指向代表地址的文件对象或者连接端点的指针，以及指向客户提供缓冲区的指针，传输器在该缓冲区中返回请求信息。

为了设置传输器信息，客户发布 TDI_SET_INFORMAION 请求，该请求用 TdiBuildSetInformation 进行设置。

对于上述操作，客户也可将系统定义的 TDI_QUERY_XXX 值作为 TdiBuild..Information

宏的 Qtype 或者 Stype 参数传递，来区分查询和设置的信息类型。

要查询和设置传输地址或连接端点以外的信息，客户在提交其查询或设置信息请求之前必须首先打开 TDI 驱动程序中的控制信道。例如，查询低层传输器的用于无连接数据通信的广播地址信息或者查询传输器的最新统计数据信息的客户要使用控制信道。

为了打开控制信道，客户用空的 EaBuffer 指针（NULL）调用 ZwCreateFile。该调用促使 I/O 管理器创建客户进程相关的代表控制信道的文件对象，并用包含客户向 ZwCreateFile 提供的参数的 IRP 来调用 TDI 传输驱动程序的 TdiDispatchCreate 例程。TdiDispatchCreate 给出空缺 EA 信息，并且如果调用成功，传输器将为打开的控制信道和该客户设置内部状态。

作为打开代表控制信道的文件对象的另一选择，客户可以调用 IoGetDeviceObjectPointer，该调用返回指向指定设备对象和相关文件对象的指针。

当 ZwCreateFile 调用成功向客户返回一个文件句柄，并且该客户也通过 ObReferenceObjectHandler 获得了该文件句柄的指针之后，就能够向低层传输器提交 TDI_QUERY_INFORMAION 或者 TDI_QUERY_SET_INFORMAION IOCTL 类型的请求。

当查询和设置信息操作完成之后，客户必须关闭打开的控制信道，该操作通过传递 ZwCreateFile 返回的文件句柄调用 ZwClose 来完成，也可通过向 ObDereferenceObject 传递 IoGetDeviceObjectPointer 返回的文件对象指针来实现，在后面 5.13 节进行讨论。

5.5 建立端端连接

网络上分立的 TDI 客户和其各自的传输器相互合作在它们之间建立端端连接。在连接建立之前，各自节点上的客户必须完成以下操作：

1、 打开一个传输地址
2、 打开一个连接端点
3、 通过向低层传输器提交 TDI_ASSOCIATION_ADDRESS IOCTL 请求的方式在打开的连接和打开的传输地址之间建立联系

然后，通常当一个客户正在侦听（被动地）即将建立的连接提议时，另一个客户向该侦听客户发送建立连接提议。客户也可以向其低层传输器注册 ClientEventConnect 处理程序（参阅 5.1 节）。当从远程节点发向客户开放传输地址的连接提议到达时，传输驱动程序将调用 ClientEventConnect。

请求对远程节点的连接

图 5.5 显示了本地节点 TDI 客户是如何发起对远程节点对等进程的连接提议的。

图 5.5　请求连接

本地节点客户通过向低层传输器提交 TDI_CONNECT 请求，向远程节点对等进程发送连接提议，该请求通过 TdiBuildConnect 进行设置。

本地节点传输器根据其客户的连接请求确定远程节点目标地址并向相应的远程节点传动器发送该连接提议。

远程节点传输器通知客户关于内入连接提议信息，这可以通过响应客户提交的 TDI_LISTEN 请求或者调用已注册的 ClientEventConnect 处理程序来实现。

如果远程节点没有侦听或者没有响应连接请求，本地节点传输器将使该连接 IRP 失效。如果两边的传输器都支持连接延迟接受功能，远程节点将既可以接受该连接提议也可以拒绝该连接提议。

接受远程节点的连接提议

图 5.6 显示了本地节点客户是如何侦听远程节点对等进程的连接提议的。

图 5.6　接受连接提议（侦听操作）

为了建立端端连接，一个客户发布连接提议，另一个客户则指示其低层传输器它正在等待连接提议。

本地节点客户可以通过向低层传输器提交 TDI_LISTEN 请求的方式被动地侦听引入连接提议，该请求通过 TdiBuildListen 进行设置。当设置侦听 IRP 时，客户能够指定发送连接提议的远程节点传输地址。如果传输器支持连接延迟接受功能，那么客户能够指示传输器要么立即接受特定远程节点地址来的连接提议，要么允许客户检查该提议并决定是够接受。

当传输器收到 TDI_LISTEN 请求之后，将监视来自远程节点的连接提议，该连接提议被发向客户打开的传输地址。当收到指定远程节点地址的连接提议时，传输器将复制提议信息到相应的缓冲区，该缓冲区是本地节点客户通过 TDI_LISTEN 请求提供的，同时结束 IRP 返回正在侦听客户。

如果客户指示传输器立即接受引入连接提议，那么当其传输器通知远程节点传输器，和远程节点客户的端端连接已经建立时，侦听请求就完成了。实际上，甚至在知道侦听 IRP 完成之前，本地节点客户就已经建立了和远程节点对等实体的端端连接，并且该客户立刻就可在该端端连接上接收数据。换句话说，传输器的侦听请求完成操作通知本地节点客户关于连接提议信息，客户要么通过发布 TDI_ACCEPT 请求接受该提议，要么通过发布 TDI_DISCONNECT 请求拒绝该提议。本地节点客户使用 TdiBuildAccept 或者 TdiBuildDisconnect 为连接延迟接受操作，设置将要提交给低层传输器的 IRP。

如果本地节点客户通过事件处理和低层 TDI 传输驱动程序进行通信，接受连接提议操作甚至将会更加简单。当传输器接收到一个远程节点客户发给本地客户打开的传输地址的连接提议时，传输器调用本地节点客户注册的 ClientEventConnect 处理程序，并传递提议客户的传输地址和接收到的连接数据。然后，ClientEventConnect 将决定立即接收还是拒绝该连接提议。

5.6 发送和接收面向连接数据

网络分立节点上的 TDI 客户必须首先建立端端连接，才能发送和接收面向连接数据。在一个客户向另一客户发送数据之前，各自节点的每个客户必须完成以下操作：

1、 打开一个传输地址
2、 打开一个连接端点
3、 通过向低层传输器提交 TDI_ASSOCIATION_ADDRESS IOCTL 请求的方式，在打开的连接和打开的传输地址之间建立联系
4、 一个客户发布连接提议，另一个接受连接提议，建立两者之间的端端连接

事实上，在 TDI_LISTEN IRP 完成并返回之前，只要本地节点传输器向远程节点传输器发送连接接受帧，接受连接提议的客户就可以从远程节点对等实体接收数据了。

当低层传输器建立端端连接之后，客户就能够在网络相互发送数据了。

在端端连接上发送数据

图 5.7 显示了核心模式客户是如何通过其低层传输器在端端连接上向远程节点对等实体

发送数据的。

图 5.7　向远程节点对等实体发送数据

本地节点 TDI 客户通过发布发送请求，实现从其连接端点向远程节点连接端点的数据传送。为实现该功能，本地节点客户向其传输器提交 TDI_LISTEN IOCTL 请求。该 IRP 由客户调用 TdiBuildSend 进行设置，其中包含有一个指向客户提供缓冲区的指针，缓冲区容纳面向流或面向消息的 TSDU，其长度可以是小于 TDI 传输器允许的最大尺寸的任意值。如果传输器支持加速传送，那么其客户就能够请求将 TSDU 作为加速数据传输，该加速数据将先于已经提交的但当前又尚未完成的标准数据进行发送。如果传输驱动程序支持内部缓冲，那么其客户可以请求非块式发送。

客户用 TDI_LISTEN IRP 调用 IoCallDriver，向低层传输器的 TdiDispatchInternelDeviceControl 例程转发 IRP。该例程检查传输器 I/O 栈中 IRP 的 MinorFunction 代码，通常要调用驱动程序内部与发送相关的函数对该 IRP 进行进一步的处理。对于发送请求，如果此前客户已经提交了其他的发送请求，而该请求又尚未通过网络向远程节点传输，那么驱动程序内部函数通常会对新的 IRP 进行排队，同时传输器总是将加速数据发送请求排在标准 TSDU 发送请求之前。不管是标准的还是加速的发送请求，传输器总是以先入先出（FIFO）的原则传输队列中客户请求的发送任务。在完成客户提交的 TDI_SEND IRP 之前，传输器要么将客户提供的数据复制到内部缓冲区，要么在网上发送指定的数据。

如果由于内部缓冲区空间不足使得非块式发送请求失败，那么当传输器又有可用的发送缓冲空间时，驱动程序将调用客户注册的 ClientEventSendPossible 处理程序。ClientEventSendPossible 将再次提交前面失败的 TDI_SEND 请求。

从端端连接上接收数据

图 5.8 显示了核心模式客户是如何通过其低层传输器从端端连接上的远程节点对等实体接收数据的。

图 5.8　从远程节点对等实体接收数据

本地节点客户可以通过向低层传输器提交 TDI_RECEIVE 请求接收标准的或者加速的 TSDU。该 IRP 由客户调用 TdiBuildReceive 进行设置，其中包含有一个指向客户提供缓冲区的指针，该缓冲区用来复制接收来自远程节点对等实体的 TSDU 的全部或部分数据，其长度可以是小于 TDI 传输器允许的最大尺寸的任意值。

客户用 TDI_RECEIVE IRP 调用 IoCallDriver，向低层传输器的 TdiDispatchInternelDeviceControl 例程转发该 IRP。该例程检查传输器 I/O 栈中 IRP 的 MinorFunction 代码，通常要调用驱动程序内部与接收相关的函数对该 IRP 进行进一步的处理。驱动程序内部函数向客户提供的缓冲区复制接收数据，直到缓冲区满或者接收的 TSDU 数据复制完毕。

接收时，加速数据优先于标准数据。如果当传输器处理客户提交的对标准数据的接收请求时，有远程节点的加速 TSDU 到达，那么传输器将结束该 IRP 并和已经复制到客户提供缓冲区的标准 TSDU 一起返回客户，接着传输器对接收的加速数据进行处理，加速数据处理完后，客户必须发布新的 TDI_RECEIVE 请求获取标准 TSDU 的其余部分。

客户也可以以事件通知的方式从低层 TDI 传输驱动程序接收来自远程节点对等实体的数据。对这些通知，驱动程序删除从远程节点接收到的 TSDU 的传输层报头，并调用客户注册的 ClientEventReceive、ClientEventChainedReceive、ClientEventReceiveExpedited 或者

ClientEventChainedReceiveExpedited 处理程序。然后，客户的事件处理程序能够尽可能的复制更多的数据。如果 ClientEventReceive 或者 ClientEventReceiveExpedited 没有接收完全部数据，可以执行以下的某一个操作：

- 立即返回没被接收状态，实际上是告诉传输器接收到的 TSDU 不是客户需要的。
- 产生另一个 TDI_RECEIVE 请求获取 TSDU 数据的其余部分。
- 依靠接下来的驱动程序接收事件通知获取接收数据的其余部分。

低层传输器总是只赋予 ClientEventChainedReceive 和 ClientEventChainedReceiveExpedited 处理程序对完整 TSDU 的只读访问能力，因此这些例程不必向低层传输器发布一系列的 TDI_RECEIVE 请求，也不必处理接收到的 TSDU 的局部指示。然而客户应该尽可能快的调用 ClientReturnChainedReceives 向 NDIS 微端口返还该指示相关的资源。

关于注册 ClientEventXxx 处理程序的更多信息，请参阅 5.1 节。

5.7 发送和接收无连接数据

只要核心模式客户成功地打开了低层 TDI 传输器的一个传输地址，就能够在网上进行无连接通信。

为了接收无连接数据，客户必须向低层传输器注册 ClientEventReceiveDatagram 处理程序和可能的 ClientEventChainedReceiveDatagram 处理程序，或者向低层传输器发布显式的 TDI_RECEIVE_DATAGRAM 请求。

关于打开传输地址和注册该地址的事件处理程序的更多信息，请参阅 5.1 节。

发送数据报

图 5.9 显示了 TDI 客户是如何向远程节点地址发送数据报的。

图 5.9　发送数据报

如图中所示，发送数据报和发送面向连接数据是相似的。然而，为发送数据，本地节点客户将向其低层传输器提交依附于一个开放传输地址的 TDI_SEND_DATAGRAM 请求，代替前面 5.6 节所述的在端端连接上的发送请求。

TDI_SEND_DATAGRAM IRP 请求低层传输器向特定远程节点传输地址上的数量未知的客户发送客户提供的数据报。

本地节点客户（拥有打开的传输地址）能够使用 TdiBuildSendDatagram 对上述请求进行设置。客户提供的缓冲数据长度可以是小于 TDI 传输驱动程序为数据报发送提供的最大缓冲空间的任意值（包括目标节点地址）。客户能够通过像 5.4 节所述的那样，查询低层传输器来确定该最大值限制。

接收数据报

图 5.10 显示了核心模式客户是如何通过其低层传输器接收来自远程节点的数据报的。

图 5.10　接收数据报

如图中所示，接收数据报与接收面向连接数据是相似的。然而，为接收数据，本地节点客户将向其低层传输器提交依附于一个开放传输地址的 TDI_RECEIVE_DATAGRAM 请求，代替前面 5.6 节所述的在端端连接上的接收请求。

然而，本地客户能够接收任何远程节点客户发送的数据报，该远程客户指定本地客户打开的传输地址作为数据报的发送目标，其他打开相同传输地址的客户也能够接收该数据报。

TDI_RECEIVE_DATAGRAM IRP 请求低层传输器返回来自远程节点客户的数据报，该远程客户向本地节点客户打开的传输地址发送了数据，本地节点客户能够使用 TdiBuildReceiveDatagram 对该请求进行设置。客户提供的缓冲区长度可以是小于 TDI 传输驱动程序为数据报接收提供的最大缓冲空间的任意值(包括代表打开的传输地址的文件对象指针)。因为 TDI 传输器不会对数据报进行分段，所以本地节点客户通常只要发布一次接收数据报请求就能接收一个数据报。

当上述请求完成时，传输驱动程序复制接收数据报到客户提供的缓冲区，并随同完成的 IRP 返回远程节点发送数据报的客户的传输地址。如果接收数据报长度超出缓冲区大小，那么传输器将返回尽可能多的数据并丢弃剩余数据。

客户也能够以低层 TDI 传输驱动程序的事件通知方式接收来自远程节点的数据报。对这些通知，驱动程序删除从远程节点接收的 TSDU 报头并调用客户注册的 ClientEventReceiveDatagram 或者可能的 ClientEventChainedReceiveDatagram 处理程序。ClientEventReceiveDatagram 能够执行以下的某一个操作：

- 立即返回没被接收状态，实际上是告诉传输器接收到的 TSDU 不是客户需要的。
- 向内部缓冲区复制尽可能多的数据并返回控制，如果传输器仅仅提供了部分数据，则为 TSDU 的其余部分再生成一个 TDI_RECEIVE_DATAGRAM 请求。
- 复制 TSDU 的全部数据到内部缓冲区并返回控制。

如果接收数据报指示发生之后，客户没有复制数据或者没有返回 IRP，那么非缓冲式传输驱动程序能够丢弃接收数据，而缓冲式传输驱动程序则保存一定数量的数据报信息，客户随后可通过显式的 TDI_RECEIVE_DATAGRAM 请求获取该信息。

低层传输器总是赋予 ClientEventChainedReceiveDatagram 处理程序对完整 TSDU 的只读访问能力，因此该例程不必向低层传输器发布 TDI_RECEIVE_DATAGRAM 请求，也不必处理接收到的 TSDU 的局部指示。然而客户应该尽可能快的调用 ClientReturnChainedReceives 向 NDIS 微端口返还该指示相关的资源。

关于注册 ClientEventXxx 处理程序的更多信息，请参阅 5.1 节。

5.8 面向连接和面向无连接传输

TDI 在传输层提供面向连接服务，以最小的代价提供最高质量的服务。根据相关驱动程序协议的特点，该服务包括客户数据打包、排序、应答、重发、流量控制和错误恢复。

TDI 也提供无连接的数据报服务。然而，该服务是轻量的：不提供纠错重发和流量控制功能。通常，传输层一般不对数据报进行分割或重组。仅就 TDI 传输器来说，无连接数据传输效率是最高的。

数据报发送本质上是一种不可靠数据通信方式。发送客户无法确定有多少或者哪一个远程节点客户已经打开了远程节点目标地址，也无法确定当前是否有远程节点传输器在接收数据报。此外，本地节点 TDI 驱动程序可以任由传输驱动程序设计者确定对数据报进行丢弃或者复制操作。相反，当端端连接有效时，本地节点传输器要负责在端端连接上重发数据，直到发送数据被远程节点接收并且已由远程节点传输器进行确认为止。

就像数据报发送一样，数据报接收操作也是不可靠的。即本地节点 TDI 驱动程序可以任由传输驱动程序设计者确定对数据报进行丢弃或者复制操作。相反，当端端连接有效时，本地节点传输器要负责提供端端连接上的接收数据，直到发送数据被其客户接收（或者拒绝）

并向远程节点传输器进行确认为止。

端端连接将一直保持有效，直到出现后面 5.11 节要讨论的断连操作或者低层传输器的超时逻辑确定远程节点不能响应为止。

5.9 请求传输相关操作

图 5.11 显示了 TDI 客户是如何向低层 TDI 驱动程序进行传输相关请求的，如果传输器定义了这些 TDI 扩展操作的话。

图 5.11　请求传输相关扩展操作

TDI 客户能够向 TDI 传输驱动程序发送特定的或专有的扩展请求，TDI 传输驱动程序为这些扩展操作定义了一组传输相关操作代码。这些关于打开地址、开放连接端点或者打开的控制信道的扩展操作仅仅适用于呼叫客户，对其他的任何 TDI 传输器客户或驱动程序都是不适用的。

为了请求上面的传输器定义操作，客户首先必须像 5.1 节、5.2 节或者 5.4 节所说的那样打开相关地址、连接端点或者控制信道，然后，客户使用 TdiBuildAction，与客户提供的包含传输器定义操作代码和请求操作的相关参数一起，设置 TDI_ACTION 请求。

5.10 接收错误通知

在对客户提交的 IRP 进行响应时，传输驱动程序在返回的状态码中将出错情况通知 TDI 客户。当向客户提供相关请求的错误信息时，客户将很难保存失败 IRP 的计数，也很难使用内部逻辑确定低层驱动程序是否处在一种将会使客户以后的网络 I/O 操作不可靠的状态。

为了接收关于低层驱动程序或者低层物理介质的意外错误状态的通知，客户可以像 5.1 节所说的那样向低层传输器注册 ClientEventError 或者 ClientEventErrorEx 处理程序。然后，如果传输器自身或者传输器依赖的任何低层驱动程序进行网上客户通信时发生错误，传输器将调用 ClientEventError(Ex)。该调用将通知其客户，客户打开的传输地址上的网络 I/O 操作不再可靠（或者有时是可能变得不可靠）。接着，客户能够通知其更高层客户关于网络失效情况，同时释放所有 TDI 客户为相关传输地址上的未决操作分配的资源。

5.11 断开端端连接

图 5.12 显示了 TDI 客户是如何释放端端连接的。

图 5.12　缺

断开连接操作本质上是与特定传输器相关的。当面相连接 TDI 发起断连请求时，连接上的两个节点可能都要参与断连操作。即，当一个客户发起断连请求时，远程节点客户可能必须响应该请求。

断连操作期间，TDI 传输器通常将拒绝该开放连接端点上的内入请求，并终止指定连接端点上的所有活动，除非两个传输器都支持可控断连功能，并且发起客户请求向其中一个提交断连请求。

如图 5.12 所示，端端连接上的一个客户能够向低层传输器通过提交 TDI_DISCONNECT

请求的方式发起断连操作，该请求可用 TdiBuildDisconnect 进行设置。当传输器完成对该客户请求的处理时，将通知远程节点传输驱动程序进行断连操作，同时该传输器将对端端连接上的客户提交的 I/O 请求返回一个合适的状态码。

如果响应客户注册了 ClientEventDisconnect 处理程序，那么当断连请求发生时，将调用该处理程序通知其客户。然后，ClientEventDisconnect 通过向低层传输器提交 TDI_DISCONNECT 请求来确认断连操作。该通知使得客户能够及时地清除为端端连接分配的状态。

然而，断连操作并没有关闭客户打开的连接端点或者传输地址，当 TDI_DISCONNECT 请求完成之后，客户能够重用这些代表其低层传输器打开资源的文件对象，客户也可以像 5.5 节所说的那样再次向网络上的远程节点发送连接提议。除非客户像 5.12 节和 5.13 节所说的那样，关闭了代表其各自连接端点的文件对象和相关的传输地址，否则对客户向低层传输器提交的 IOCTL 请求来说，相关资源仍然是可用的。

5.12 关闭连接端点

图 5.13 显示了 TDI 客户是如何关闭连接端点的。

图 5.13　关闭本地节点的连接端点

当端端连接断开后，客户就可以关闭连接端点了。

当客户不再使用连接端点时，必须进行如下所述操作关闭连接端点：

- 用 ObReferenceObjectByHandler 返回的文件对象指针来调用 ObDereferenceObject。
- 用在连接端点打开时 ZwCreateFile 返回的文件句柄调用 ZwClose。

然后，I/O 管理器向传输器的 TdiDispatchCleanUp 例程和随后的 TdiDispatchClose 例程提交 IRP。

这些传输器例程将立即关闭连接端点，并释放所有相关的传输驱动程序资源。TdiDispatchCleanUp 也将通过向相应的远程节点发送断连请求的方式终止与该连接端点相关的任何活动连接。

就像前一段讨论所表示的，在产生关闭连接端点请求之前，TDI 客户没有必要将连接端点和相关传输地址分离。如果必要，低层传输器可以模拟该分离效果。

然而，在关闭连接端点之前，客户也能够通过向传输器提交 TDI_DISCONNECT_ADDRESS 请求的方式，显式的对连接端点和相关传输地址进行分离，该请求可通过 TdiBuildDisconnectAddress 进行设置。

例如，客户可以生成分离地址请求，并将开放连接端点与其他打开的传输地址进行重新联合。

5.13 关闭传输地址和控制信道

在关闭所有相关连接之后，TDI 客户就可以关闭当前打开的传输地址了。

当客户不再使用打开的传输地址或者控制信道时，必须像下面所述的这样释放该对象：

- 用 ObReferenceObjectByHandler 返回的文件对象指针来调用 ObDereferenceObject。
- 用在连接端点打开时 ZwCreateFile 返回的文件句柄来调用 ZwClose。

相似的，TDI 客户也能够关闭低层传输驱动程序中打开的任何控制信道。

如果客户是调用 IoGetDeviceObjectPointer 来打开代表控制信道的文件对象的，那么就必

须向 ObDereferenceObject 调用传递 ObReferenceObjectByHandler 返回的文件对象指针，从而释放该文件对象。

然后，I/O 管理器向传输器的 TdiDispatchCleanUp 例程和随后的 TdiDispatchClose 例程提交 IRP。

这些传输器例程将立即关闭传输地址或者控制信道，并释放所有相关的传输驱动程序资源。例如，TdiDispatchCleanUp 撤销将要关闭的传输地址上的所有未决请求，注销该地址上的所有 ClientEventXxx 处理程序，清除该地址的客户相关状态。如果该客户释放了为该传输地址分配的所有文件对象句柄，那么传输器也将释放该传输地址的内部状态。

当 ZwClose 返回客户之后，就不能再为前面打开的传输地址或者控制信道向低层传输器提交请求了。代表该地址或者控制信道的客户相关文件对象也将不复存在。

第六章 Windows　Sockets 的传输助手 DLLS

作为 TCP / IP 的应用程序编程接口，Windows Sockets 已经成为 Windows 2000 网络传输的主要用户模式接口。除了 TCP / IP，Windows Sockets 动态链接库还与其他传输形式相连系，这些传输器以传输器特有的用户模式形式向 Windows Sockets Helper DLLS(WSH)提供扩展。

本章描述了一个新的传输驱动程序所支撑的 WSH DLLS，通过 Windows Sockets 支持应用程序的调用：

- Windows Sockets、传输驱动程序和特殊传输 WSH DLL 之间的结构关系总览。
- Windows Sockets 和特殊传输 WSH DLLS 通信概况。
- 建立 WSH DLLS 的配置注册。
- 管理 WSH DLLS 的同步。
- WSH DLLS 在连接和断开时如何支持网络中发送的额外信息。
- 必需的 WSHxx 和可选的 **WSH**xx 功能总结。

6.1 Windows Sockets Helper DLL 结构

WSH DLLS 是由传输驱动程序开发者为方便 Windows Sockets 的使用而提供的用户模式组件。图 3.1 显示了这个结构的总貌。

图 3.1　Windows Sockets Helper DLL 结构

Windows 2000 提供了一个 DLL,msafd.dll,它是 Socket 服务的提供者。当传输驱动程序安装时，同时也会安装特殊传输 WSH DLL，网络设置将自动把 msafd.dll 配置为该 WSH DLL 的服务提供者。在应用程序调用 Windows Sockets 函数时(如 **WSASocket**)，作为服务提供者 ,MSAFD 将处理这个调用,并调用合适的 WSH DLL 以寻求必要的帮助。一些 Windows Sockets 函数不需要任何 WSH DLL 的帮助。例如，在连接已建立的情况下发送和接受数据不需要 WSH DLL。在此情况下，MSAFD 通过调用 Win32 函数和传输驱动程序进行通信。

然而对于依赖于特殊传输特性的函数调用，或传输驱动的实现不统一时，传输器专用的 WSH DLL 可以用来消除这些不确定性。例如，**WSAJoinLeaf** 将一个 Socket 加入到已建立的多点会话中。每个传输器各自实现多点会话中一个连接的加入。结果，MSAFD 调用合适的 WSH DLL，该 DLL 根据特殊传输实现支持 Socket 接口，为了支持对 **WSAJoinLeaf** 的调用，MSAFD 调用 WSH DLL 的 **WSAJoinLeaf** 函数验证请求的选项，并用合适的信息调用传输器，从而把一个 socket 增加到多点会话中。

如果一个传输器支持 Windows Sockets2 的新特性，包括多点会话 Socket，Socket 集群，和 Sockets 地址的逻辑表示，相应的 WSH DLL 必须输出 Windows Sockets2 函数，这将在 6.3 节中总结。

6.2 用 WSH DLL 通信

每个支持 Windows Sockets 应用程序的传输器都支持一个用户模式的 WSH DLL，用以解释网络地址和处理 6.1 节所列结构的 Socket 选项。例如，一个应用程序调用 **Socket** 时，它说明了地址族、Socket 类型和协议。这三个参数必须确定唯一的传输驱动程序来支持此

Socket。Windows Sockets 搜索应用程序提供的参数和注册表中 WSH DLL 的标准化配置是否匹配，如果它发现一个匹配，Windows Sockets 通过 MSAFD 调用由 WSH DLL 输出的 **WSH**Xxx 函数，否则，应用程序调用 **Socket** 失败。当找到一个匹配时，Windows Sockets 调用 WSH DLL 中的 **LoadLibrary**,然后调用 **GetProcAddress** 取回 **WSH**Xxx 函数的入口。在查询 WSH DLL 支持的协议时，这些 **WSH**Xxx 函数被调用，用以翻译地址，并调用 **getsockopt** 和 **setsockopt** 处理 WSH 支持选项。例如，当一个应用程序将选项传递给 **setsockopt**（Windows Sockets 不显式支持）时，将会调用 WSHDLL 处理这个选项。WSHDLL 作必要的操作，对每个选项提供相应的传输器支持。

下面的协定允许 socket 地址（即应用程序定义的 SOCKADDR 结构）和传输驱动程序使用的 TDI-defined 地址转变。

• TDI 地址类型，由 TA_ADDRESS 结构中的 **AddressType** 说明，它等同于 socket 地址结构中的地址族。

• TA_ADDRESS 结构中的 **Address** [] 和 socket 地址结构中的 **sa_family** 的值相等。

6.3 配置 WSH DLL

注册表中有两个地方必须设置成 WSH DLL 的标准配置信息。

1. 在 **HKEY_LOCAL_MACHINE \ SYSEM \ CurrentControlSet \ Services \ Winsock \ Parameters** 下是说明协议(传输驱动程序)列表的 REG_MULTI_SZ 入口值。存储在列表中的值和**..\CulrentControlSet\Services** 下相应的传输驱动程序的键名相匹配。

2.对每个传输驱动程序来说,**HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Sevices\TransportDriverName\Parameters** 有一个 **Winsock** 子键。Windows Sockets 从 **Service** 键下它自己的键的路径中，**..\Winsock\Parameters** 下列表中描述的传输器名字中，和传输器的**..Parameters\Winsock** 键中形成一个名字串，从而打开那个键，该键必须包括以下值的项：

• **Mapping** 一个描述地址族，socket 类型和协议参数三元组的 REG_BINARY 值。这个二进制数据格式是 WINSOCK_MAPPING 结构，在 wsahelp.h 中定义。

• **HelperDllName** 一个描述 WSH DLL 路径的 REG_EXPAND_SZ 值。

• **MinSockaddrLength** 以字节为单位，说明 SOCKADDR 的最小效值，它的类型是 REG_DWORD。

• **MaxSockaddrLength** 以字节为单位,说明 SOCKADDR 的最大有效值, 它的类型是 REG_DWORD。

把这个信息存储在**..\CurrentControlSet\Services** 下两个不同的地方。将位于驱动程序自己的 **Parameters** 键下的信息本地化。**..\Services\Winsock\Paramters** 下的信息提供了每个传输 WSH DLL 信息的指针，使得安装程序更容易地相互合作。

Windows 2000 安装工具提供了一些函数，这些函数为传输驱动程序建立 WSH DLL 注册信息做了绝大多数工作。这些例程包括 **AddWinsockInfo** 和 **RemoveWinsockInfo**.传输驱动程序安装脚本能调用 **AddWinsockInfo**,传送传输驱动程序 **Services** 下的键名、特殊传输 WSH DLL 的完全路径名、SOCKADDR 的最小和最长长度。**AddWinsockInfo** 存储注册表中的标准信息。传输器调用安装函数的例子，请看 *oemnxptc.inf*,这是 TCP / IP 的安装脚本，也位于 System 目录下。

6.4WSH DLL 同步

对每个 socket，Windows Sockets 都保持与 WSH DLL 的同步，也就是说，两个线程不能对同一个 socket 同时执行 **WSH**Xxx 函数。从而，WSH DLL 本身不需要做任何同步工作。

如果一个 WSH DLL 支持对全局状态信息的存取进行同步，它必须在内部对状态信息存取进行同步。Windows 2000 提供了几个用户模式的同步机制，包括信号量，事件，和关键段。具

体哪个同步机制取决于 WSH DLL 开发者。

6.5 用 WSH DLL 支持连接和断开数据

一些传输器，如 DECNet 和 OSITP4，支持连接和断开数据：不在正常的网络数据流中的额外数据。它随着连接和断开请求而发生。通常，连接和断开数据被用于应用程序级的版本协商。

TDI 用 TDI_CONNETION_INFORMATION 结构支持连接和断开数据的传送，该结构包括 UserDataLength,UserData,OptionsLength 和 Options。该结构在由核心模式的 TDI 客户发出的 TDI 连接、接受和断开请求中被传输。

然而，Windows Sockets 不向 connect 函数提供连接数据的输入参数，也不向 Shutdown 和 closesocket 函数提供断开数据的输入参数。WSH DLL 为调用 Windows Sockets 的 setsockopt 和 getsockopt 函数的应用程序,增加对连接和断开数据的支持。

setsockopt 函数用来说明发送到远程结点的连接和断开数据。getsockopt 函数，取到来自远程结点的连接和断开数据。详情请参见 *Windows.h* 中的 SO_XXX 选项。

一个应用程序如何在 getsockopt 和 setsockopt 中使用这些选项,取决于应用程序是个服务器还是客户(参见 6.4.1 节和 6.4.2 节)，以及应用程序如何使用连接数据。对于传输器而言,如 DECNet 有一个与上面不同的、预置的连接数据定义,WSH DLL 对传输语义和 Windows Sockets 语义进行互译。

从 Windows Sockets 来看，连接和断开选项与连接和断开数据一样有效。这实际上是在连接和断开之前传送给传输器的数据缓冲区，这个缓冲区包括随着连接和断开而得到的数据。结果，应用程序常常把连接或断开选项和连接或断开数据以同一方式使用。仅有的不同是选项名多被作为传送给 getsockopt 和 setsockopt 的参数。例如，SO_CONNOPT 代替了 SO_CONNDATA。

6.5.1 客户应用程序和连接数据

一个应用程序使用 Windows Sockets 和支持连接数据的传送器通信，此 Windows Sockets 使用支持连接数据的传输器，该应用程序在调用 setsockopt 时，应使用 SOL_SOCKET 作为级别，SO_CONNDATA 作为选项名，一个指向连接数据缓冲区的指针作为选项值，这个连接数据是由传输器传送到远程结点的，数据与连接请求一同发送。所以在一个应用程序连接之前必须调用 setsockopt。当应用程序调用 connect 时，这些连接数据将被发送。

1．在调用 connect 前，应用程序必须通知 Windows Sockets 为预期的连接响应数据预留多大空间。为做到这一点，应用程序使用选项级别 SOL_SOCKET，选项名字 SO_CONNDATALEN，和指向一个整数的选项来调用 setsockopt,该整数说明了应用程序所期望的最大字节数。WindowsSockets 分配这么大的缓冲区以容纳来自服务器应用程序的连接数据。

2．在 connect 成功完成后，应用程序能以选项组别 SOL_SOCKET，选项名字 SO_CONNDATA，和指向容纳响应连接数据的缓冲区的指针，调用 getsockopt。然后，Windows Sockets 将响应连接数据拷入应用程序缓冲区。

在应用程序创建响应连接缓冲区之后，Windows Sockets 在此 socket 的生命周期中一直使用那个缓冲区。那就是说，在一个 socket 连接后和关闭前，应用程序在任何时候都可以取到该 socket 的响应连接数据。

6.5.2 服务器应用程序和连接数据

服务器应用程序以两种重要方式使用来自客户应用程序的连接数据。

1．在发送响应之前，服务器应用程序接收客户的连接数据。

2．因为 Windows Sockets 不支持延迟连接接受(TDI 可以)，服务器应用程序必须向所有客

户发送相同的响应连接数据。

为了取到来自客户的连接数据，服务器应用程序用 SO_CONNDATALEN 选项调用 **setsockopt**，使用 Windows Sockets 为连接数据保留缓冲区空间。连接数据缓冲区的大小对所有连接都是一样的，但 Windows Sockets 为每个连接分配独立的缓冲区。

在一个新连接的 socket 被 **accept** 后，服务器应用程序用 SO_CONNDATA 选项和一个足以容纳期望数据的、由服务器分配的缓冲区来调用 **getsockopt**。Windows Sockets 用来自客户的连接数据填充这个缓冲区。

为向客户发送响应连接数据，服务器应用程序用 SO_CONNDATA 选项和一个数据缓冲区，在连接被接受之前调用 **setsockopt**(通常在 **listen** 之前)。此后，连接服务器的客户都会收到服务器提供的响应连接数据。服务器的响应连接数据在任何时候都可以用 **setsockopt** 来改变，用新的数据代替老数据。

6.5.3 断连（disconnect）数据

客户和服务器应用程序以类似于发送和接收连接数据的方式发送断连数据。

客户应用程序使用 SO_DISCDATA 选项和一个断开数据（发送到远程结点服务器）的缓冲区，来调用 **setsockopt**。客户应用程序调用 **shutdown** 断开 socket 的发送方时，或者应用程序调用 **closesocket** 时，发送这个数据。

为了取到断开数据，客户应用程序在远端关闭连接以前，必须用 SO_DISCDATALEN 选项调用 **setsockopt**。这个选项说明了应用程序所期望的断开数据的数量。

在远程结点断开后，服务器提供的断开数据能以 SO_DISCDATA 选项调用 **getsockopt** 来获得。客户应用程序传递一个足以容纳断开数据的缓冲区，Windows Sockets 将收到的断开数据拷入该缓冲区。

6.6WSH DLL 函数总览

现在总结每个 WSH DLL 都必须提供的 **WSH**Xxx 函数集。每个新传输器的供应者能在与传输特性相符的 WSH DLL 中实现这些函数。Windows Sockets 信任所有 WSH DLL。所以，**WSH**Xxx 函数中的 bug 会表现为 Windows Sockets 的 bug.

下面概括了所有传输器专用的 WSHDLL 必须提供的函数集：

- **WSHEnumProtocols** 返回 WSH DLL 支持的协议列表(Windows Sockets PROTOCOL_INFO 结构))
- **WSHGetsockAddrType** 解析一个 socket 地址。
- **WSHGetsocketInformation** 在向 **getsockopt** 传递一个 Windows Sockets 不显式支持的选项时被调用。
- **WSHGetWildCardSockAddr** 在 Windows Sockets 需要执行 socket 自动捆绑时被调用。
- **WSHGetWinsockMapping** 返回地址族，socket 类型和 WSH DLL 支持的协议参数三元组。
- **WSHNotify** 把状态转换通知助手 DLL。
- **WSHOpenSocket** 在一个 socket 打开时调用。
- **WSHSetsocketInfomation** 在 **setsockopt** 被传送一个 Windows sockets 不显式支持的选项时调用。为支持新的 Windows Sockets2 的叶\根、Socket、Socket 群和 Socket 的逻辑表示，应实现下面可选的函数：
- **WSHAddressToString** 返回一个用来显示的 socket 地址的逻辑字符串表示。
- **WSHGetBroadcastSockaddr** 得到 socket 的有效广播地址。
- **WSHGetProviderGuid** 返回助手 DLL 的协议的 GUID

- **WSHGetWSAProtocolInfo** 返回助手 DLL 支持的协议信息的指针
- **WSHIoctl** 得到基于一般特殊控制码的操作的执行信息。
- **WSHJoinJeaf** 执行任何协议专用的操作，这些操作在把一个 socket 加入多点会话时必须执行。
- **WSHOpenSocket2** 为创建一个新 Socket 而执行协议专用的操作。如果这个函数被输出，它将代替 **WSHOpenSocket** 函数。
- **WSHStringToAddress** 将一个 socket 的逻辑表示转成 SOCKADDR 结构。WSHXxx 函数的更多信息，请参见在线 *"Network Drivers Reference"*。

第四部分面向连接的网络驱动程序接口标准(NDIS)

第一章 面向连接的网络驱动程序接口标准(NDIS)

网络驱动程序接口标准(NDIS)提供了对面向连接介质的支持，如异步传输模式(ATM)和ISDN。下面几节描述面向连接的NDIS：

- 1.1 面向连接环境
- 1.2 使用 AFs，VCs，SAPs 和 Parties
- 1.3 服务质量
- 1.4MCM 与呼叫管理有何不同
- 1.5 面向连接的时间特性
- 1.6 面向连接的操作

1.1 面向连接环境

NDIS 支持如下面向连接的驱动程序：

- 面向连接客户
- 呼叫管理器
- 集成微端口呼叫管理器(MCM)
- 面向连接的微端口

图 1.1 显示了面向连接客户，呼叫管理器和微端口的结构。

图 1.1 呼叫管理器的面向连接环境

图 1.2 显示了面向连接客户和集成微端口呼叫管理器(MCM)的结构

图 1.2 集成微端口呼叫管理器的面向连接环境

一个面向连接微端口控制一个或多个网络接口卡(NICs)，并提供了面向连接协议(面向连接客户和呼叫管理器)和 NIC 硬件的接口。面向连接微端口执行的面向连接操作的总结，请参见 1.6.1.3 节。

呼叫管理器是为面向连接客户提供呼叫建立和断开服务的 NDIS 协议驱动程序。呼叫管理器使用面向连接微端口的发送和接收功能在网络实体之间交换信令消息，这些实体包括交换机和远程对等体。呼叫管理器支持 1 个或多个的信令协议，如由 ATM 论坛规范的 ATM UNI3.1。呼叫管理器执行的面向连接的操作总结，请参见 1.6.1.2 节。

集成微端口呼叫管理器(MCM)是一个向面向连接客户提供呼叫管理器服务的面向连接微端口。虽然 MCM 向客户提供的功能和基于面向连接微端口的呼叫管理器一样，但由于 MCM 的呼叫管理器／微端口接口是内置的，因此对于 NDIS 来说是不可见的。MCM 和呼叫管理的详细比较参见 1.4 节。

多个呼叫管理器和／或MCM在同一环境中并存，每个呼叫管理器或MCM能支持多个信令协议。一个面向连接的客户使用呼叫管理器或 MCM 的呼叫建立和断开服务。面向连接的客户也能用

面向连接的微端口或 MCM 的发送和接收功能发送和接收数据，由面向连接客户执行的面向连接的操作的总结，参见 1.6.1.1。
一个面向连接的客户类似于无连接协议，因为它能在其上边界，向更高层应用程序提供它自己的网络和传输层服务。然而，不象无连接协议，面向连接的客户在其低层使用呼叫管理器和面向连接的微端口的服务，或者使用 MCM 的服务。
一个面向连接的客户可以是位于合法协议和面向连接 NDIS 之间的适配层。如 IP / ATM 和 LAN Emulation(LANE)，两者调用呼叫管理器的服务建立连接特性。注意如此一个面向连接客户上层接口的定义已超过了 NDIS 文献的范围范畴。如果客户用作适配层，它的上层接口由协议定义，该协议需要由其提供对 NDIS 的适配功能。

1.2 使用 AFs，VCs，SAP 和 Parties

面向连接的驱动程序创建和使用
• 1.2.1 地址族(AFs)
• 1.2.2 虚连接(VCs)
• 1.2.3 服务访问点(SAPs)
• 1.2.4Parties
当一个面向连接驱动程序注册一个 AF 或创建一个 VC，SAP 或 Party 时，它将为那个 NDIS 实体传递本地环境区的指针。NDIS 向这个驱动程序，返回一个新注册的 AF 或新创建的 VC，SAP 或 Party 的句柄。

1.2.1 地址族

一个地址族(AF)表示了如下驱动程序集之间的联系。
• 面向连接客户，呼叫管理器，和下层的面向连接的微端口
• 面向连接客户和信令协议专用的的 MCM
地址族也说明了特殊信令协议，如 ATM UNI3.1。
呼叫管理器或 MCM 通过在 NDIS 中注册地址族，向特殊信令协议公布其呼叫管理器服务。在能使用呼叫管理器或 MCM 提供的呼叫管理器服务前，面向连接的客户必须用呼叫管理器或 MCM 打开地址族。
地址族操作的更多信息，请参见 1.6.2.1 节和 1.6.2.4 节。

1.2.2 虚连接

在一个本地系统中，虚连接(VC)是客户，呼叫管理器或 MCM，微端口之间的一个端点。网络中，一个 VC 指两个通信端点的连接，如两个面向连接客户。
一块网络接口卡上同时可有许多 VCs 处于激活状态，允许网络接口卡同时向多个呼叫提供服务。每个连接可连到不同计算机的不同端点。
网络中的 VCs 在向客户提供的服务的类型上是不同的。例如，一个 VC 能提供单向或双向服务。服务质量(QoS)参数保证了特殊的性能阈值，如带宽和响应时间。在信令协议基础上 VC 的 QoS 是可协商的。更多的 NDIS 对 QoS 的支持信息，请参见 1.3 节。网络中的 VC 可以是交换 VC(SVC)或永久 VC(PVC)
• SVC 在一个特殊呼叫需要时生成。例如，面向连接客户为了建立外出呼叫而创建一个 VC。类似地，呼叫管理器或 MCM 为了向面向连接客户指示一个内入呼叫而建立一个 VC。呼叫管理器或 MCM 必须和远端通信并协商 VC 参数。
• PVC 由手动建立，并最终由使用配置工具的操作员删除，NDIS 并不支持这个配置工具，。

监视 PVCS 手动生成和删除的客户能使用 OID_CO_ADD_PVC 和 OIO_CO_DELETE_PVC 请求呼叫管理器或 MCM 从 PVCs 配置列表中增加或删除一个 PVC。PVC 的 QoS 由操作员定义，网络中是不可协商的。
在 NDIS 中，一个 VC 由微端口分配的资源组成，该资源保持了网络中一个 VC 的状态信息。这些资源包括存储缓冲区，事件和数据结构,但并不限于此。经面向连接客户请求，微端口为外出呼叫建立一个 VC 的环境;经呼叫管理器请求,微端口为内入呼叫建立一个 VC 的环境。更多的 VCs 生成信息，请参见 1.6.3.1 节。

在一个 VC 能传输数据前，它必须由呼叫管理器或 MCM 激活。为激活一个 VC，微端口或 MCM 为 VC 建立资源，并与 NIC 进行通信，使 NIC 准备接收或传送 VC 中的数据。更多的 VC 激活信息，请参见 1.6.3.2 节。

当断开一个呼叫时，呼叫管理器或 MCM 使该呼叫使用的 VC 去活。VC 去活的更多信息，参见 1.6.3.3 节。

在一个呼叫断开后，VC 的生成者(面向连接客户，呼叫管理器，或 MCM)能或者删除ＶＣ，或者将其用于其他呼叫。关于删除ＶＣ的更多信息，请参见 1.6.3.4 节。

1.2.3 SAPs

服务访问点(SAP)定义了面向连接客户感兴趣的内入呼叫集，通过向呼叫管理器或 MCM 注册一个 SAP，客户指示呼叫管理器或 MCM 向客户通知那个 SAP 的所有内入呼叫。
SAP 总是用客户已开放的地址族注册。客户能向一个呼叫管理器或 MCM 注册多个 SAPs。
注册或注销 SAPs 的更多信息，参见 1.6.2.2 节和 1.6.2.3 节

1.2.4 Parties

一个 Party 代表了一对多连接中多个叶节点的 1 个。当进行一个外出呼叫时，面向连接客户可以说明一个 Party。这使得该呼叫成为一个多点呼叫，客户作为呼叫根，远程 Party 作为叶节点。然后，客户能请求其他远程 Parties 作为叶节点结点加入该呼叫。
关于增加 Parties 到一对多呼叫的更多信息,参见 1.6.6.1 节。从一对多呼叫中删除 Parties 的信息，参见 1.6.6.2 节。

1.3 服务质量

SVC 上一个呼叫发起者能说明该呼叫的服务质量(QoS)参数，根据所用的信令协议，呼叫管理器或 MCM 能协商网络实体的 QoS，如网络交换机或远程客户。如果信令协议允许，面向连接客户也可以在决定，是否在接受一个内入呼叫时，请求改变 QoS。

呼叫的 QoS 参数是 CO_CALL_PARAMETRS 结构。CO_CALL_PARAMETERS 指向两个其他结构：
• CO_CALL_MANAGER_PARAMETERS,说明呼叫管理器参数，呼叫管理器和 MCM 用这个参数建立呼叫。
• CO_MEDIA_PARAMETERS，说明介质参数，呼叫管理器和 MCM 用这个参数建立呼叫。

CO_CALL_MANAGER_PARAMETERS 和 CO_MEDIA_PARAMETERS 都包含了种类参数(flags),此参数适用于所有使用这些参数的驱动程序。这些结构也都指向一个 CO_SPECIFIC_PARAMETERS 结构，这个结构说明了特殊呼叫管理器参数(在 CO_CALL_MANAGER_PARAMETERS 结构指向它时)或者特殊介质参数(在 CO_MEDIA_PARAMETERS 指向它时)。
QoS 操作的更多信息，参见 1.6.5.1 节和 1.6.5.2 节。

1.4MCM 和呼叫管理器有何不同

集成微端口(MCM) 也是一个面向连接微端口，同时向面连接客户提供呼叫管理器服务。MCM 执行了面向连接微端口和呼叫管理器的所有面向连接功能。象所有的微端口一样，MCM 必须调用 NdisXxx 和 NIC 硬件通信。MCM 主要在两个方面不同于呼叫管理器:

• 呼叫管理器是一个增加了呼叫管理器功能的 NDIS 面向连接协议驱动程序。MCM 是增加了呼叫管理器功能的 NDIS 面向连接微端口驱动程序。

• 呼叫管理器和微端口完全暴露于 NDIS--那就是说，所有呼叫管理器和微端口之间的通信都经过 NDIS。除了客户 VCs 的激活去活(VCs 用来传送外出和内入的客户数据)，MCM 的呼叫管理器部分和微端口部分的接口对于 NDIS 来说是不可见的。客户 VCs 的激活和去活必须通过 NDIS 来完成，因为 NDIS 保持着客户 VCs 的跟踪。

MCM 和呼叫管理器的差异在下面几节中作进一步描述。

1.4.1 初始化的不同

呼叫管理器是一个 NDIS 协议，它遵守面向连接协议的初始化顺序，并有一个额外的步骤：在它的 ProtocolBindAdapter 处理程序中,在完成面向连接协议初始化步骤后(见第三部分)，呼叫管理器必须立即通过调用 NdisCmRegisterAddressFamily 注册地址族。在 NdisCmRegisterAddressFamily 函数中，呼叫管理器必须为每个绑定的 NIC 注册地址族。

由于 MCM 是一个微端口，它遵守面向连接微端口的初始化顺序，并有一个额外的步骤：在它的 *MiniportInitialize* 函数中，在完成微端口初始序列后(参见第二部分)，MCM 必须通过调用 NdisMCmRegisterAddressFamily 来注册地址族。在 NdisMCmRegisterAddressFamily 中，MCM 注册了它的呼叫管理器功能，将 MCM 与通常的面向连接微端口区分出来。虽然，在初始化过程中，MCM 在调用 NdisMCmRegesterMiniport 时，仅对微端口处理程序注册一次，但是，它必须为每个 NIC 调用 NdisMcmRegisterAddressFamily 一次。

1.4.2 对 NdisXxx 函数调用的不同

呼叫管理器和 MCM 调用不同的呼叫管理器函数集。呼叫管理器调用 NdisCmXxx 函数，而 MCM 调用 NdisMCmXxx 函数。MCM 不调用面向连接客户和呼叫管理器调用的 NdisCoXxx 函数。相反，MCM 调用下面的 NdisMCmXxx 函数：

• NdisMCmCreateVc 而不是 NdisCoCreateVc
• NdisMCmDeleteVc 而不是 NdisCoDeleteVc
• NdisMCmRequest 而不是 NdisCoRequest
• NdisMCmRequestComplete 而不是 NdisCoRequestComplete

MCM 不需类似 NdisCoSendPackets 的调用,由于呼叫管理器和微端口的发送界面内置于 MCM，因而对 NDIS 来说不可见。

1.4.3 虚连接的不同

呼叫管理器用信令 VCs 向/从网络实体(如交换机)，发送 / 接收信令消息。呼叫管理器的信令 VCs 对 NDIS 是可见的。呼叫管理器必须通过调用 NDIS 创建，激活，去活，删除 VCs。然而，一个 MCM 的信令 VCs 对 NDIS 是不可见的。MCM 在内部执行这个操作。MCM 必须调用 NDIS 执行 VCs 上的操作，从而发送和接受客户数据。NDIS 保持着对客户 VCs 的追踪。

MCM 既是呼叫管理器又是微端口的事实，意味着某些面向连接操作是冗余的。*MiniportCoCreateVC* 和 *MiniportCoDeleteVC* 是冗余的，因此 MCM 不支持。对 VC 的操作通

过如下函数进行：
• 客户请求 VC 的生成和删除时，调用 MCM 的 *ProtocolCoCreateVC* 和 *ProtocolCoDeleteVC* 函数。
• MCM 创建或删除 VC 时，调用 **NdisMCmCreateVC** 和 **NdisMCmDeleteVC**
• MCM 激活或去活 VC 时，调用 **NdisMCmActivateVc** 和 **NdisMCmDeactivateVc**
MCM 必须支持 *MiniportCoRequest* 函数以使客户可以查询或设置微端口信息，同时支持 *MiniportCoSendPackets* 函数处理来自客户的发送操作。

1.5 面向连接的时间特性

面向连接的 NDIS 使用 NIC 的本地时间安排包传输和发送与接收包的时间戳。
面向连接的协议能使用 OID_GEN_CO_GET_TIME_CAPS 调用 MiniportCoRequest,查询面向连接微端口或 MCM 的本地计时能力。作为对此查询的反应，微端口或 MCM 返回如下信息：
• NIC 上是否有个就绪的时钟
• NIC 是否从网络连接中得到它的时间
• 本地时钟的精确度
• NIC 是否能用本地时间对接受到的包加时间戳
• NIC 能否根据本地时间安排发送包的传输。
• NIC 能否用本地时间对传送的包加盖时间戳。
为得到 NIC 的本地时间，面向连接协议可以用 OID_GEN_CO_GET_NETCARD_TIME 调用 **NdisCoRequest** 来查询面向连接微端口或 MCM。面向连接微端口或 MCM 同步的返回本地时间，面向连接协议可以用来安排包传送。
发送或接收包的时间信息包含在包的带外(OOB)数据中。更多的信息，参见在线 *“Network Drivers Reference”* 中的 *NDIS_PACKET_OOB_DATA*。

1.6 面向连接操作

这节从全局角度描述面向连接操作。在每个面向连接操作描述中，包含了所有牵涉到的面向连接驱动程序，并总结了在那个操作中，驱动程序和 NDIS 调用的所有面向连接函数。

1.6.1 面向连接操作总结

这节描述了由面向连接客户，呼叫管理器，微端口执行的面向连接操作。集成微端口呼叫管理器(MCM)执行呼叫管理器和面向连接微端口两者的面向连接操作。

1.6.1.1 由客户执行的面向连接操作

面向连接客户执行如下的面向连接操作
• 打开和关闭地址族
接收到来自 NDIS 的通知(表明呼叫管理器或 MCM 已经注册了一个地址族)时，面向连接驱动程序能用呼叫管理器或 MCM 打开那个地址族(参见 1.6.2.1 节)。客户然后就能使用呼叫管理器或 MCM 提供的呼叫管理器服务了。客户通过关闭地址族,释放它和呼叫管理器或 MCM 的联系（参见 1.6.2.4 节)。
• 注册和注销 SAPs
在用呼叫管理器或 MCM 打开一个地址族后,面向连接客户能用呼叫管理器或 MCM 注册一个或更多的 SAPs(参见 1.6.2.2 节)。呼叫管理器或 MCM 将向客户指示所有与该 SAP 相关的内入呼叫。客户通过注销(deregister)来释放 SAPs(参见 1.6.2.3 节)。

• 增加和删除 PVCs

当一个操作员手动配置或解除一个永久 VC(PVC)时，面向连接客户能进行监测。作为对这个操作的响应，客户请求呼叫管理器或 MCM 将 PVCs 加入 PVCs 列表，或者从列表中删去一个 PVC(参 见 在 线 DDK “ *Network Drivers Reference* ” 的 *OID_CO_ADD_PVC* ， *DDK_OID__CO_ADD_PVC_NR* 和 *OID_CO_DELETE_PVC*)。

• 进行外出呼叫

在进行外出呼叫之前，客户必须为此呼叫建立一个 VC(参见 1.6.3.1 节)。然后客户可以进行外出呼叫(见 1.6.4.1 节)。为了进行点-多点呼叫，客户在进行呼叫时必须说明一个 party。

• 从一对多呼叫中增加或删除一个 party

客户能将一个 party 加入一对多呼叫(参见 1.6.6.1 节)，并能从一对多呼叫中删除一个 party(参见 1.6.6.2 节)。客户也能对一个内入呼叫作出反应，从而从一对多呼叫中删除一个 party(参见 1.6.6.3 节)。

• 接受或拒绝内入呼叫

客户能接受或拒绝某个 SAP 的内入呼叫，该 SAP 是客户先前向呼叫管理器或 MCM 注册的(参见 1.6.4.2 节)。

• 为一个激活的 VC 协商呼叫参数

在信令协议允许的基础上，客户可以为一个激活的 VC 协商呼叫参数。客户能请求改变服务质量(QoS)(参见 1.6.5.1 节)，也能响应一个激活 VC 的 QoS 的改变请求。客户也能对来自远程 party 的 QoS 改变请求作出响应（参见 1.6.5.2 节)。

• 发送和接受包

客户能通过面向连接微端口或 MCM 发送包，(参见 1.6.7.1 节)

客户能通过面向连接微端口或 MCM 接收包，(参见 1.6.7.2 节)

• 发起 VC 的删除

客户能发起删除它所创建的 VC。

• 发起断开呼叫

客户能发起断开外出呼叫或它所接收的内入呼叫（参见 1.6.8.1 节)。

• 查询或设置信息

客户通过一个绑定的呼叫管理器或 MCM 的呼叫管理器部分，来查询或设置信息。客户也能对来自呼叫管理器或 MCM 的查询和设置作出响应。(参见 1.6.9.1 节)

另外，客户可以通过绑定的微端口或 MCM 的微端口部分来查询或设置信息（参见 1.6.9.1 节)。

• 输入微端口状态指示

客户能输入由面向连接微端口或 MCM 驱动程序指示的状态。(参见 1.6.9.2 节)

• 重置 NIC

面向连接客户可以使面向连接微端口或 MCM 重置它的 NIC(参见 1.6.10 节)。

1.6.1.2 由呼叫管理器执行的面向连接操作

呼叫管理器执行如下的面向连接操作

• 注册和注销 1 个或多个地址族(参见 1.6.2.1 节)。通过注册一个地址族，呼叫管理器用 NDIS 注册了它的呼叫管理器函数入口，并且向面向连接客户公布了它的呼叫管理器服务(一个信令协议)。

• 在面向连接客户的请求下注册和注销 SAPs

呼叫管理器接收来自绑定的面向连接客户的请求，这些请求要求注册 SAPs(参见 1.6.2.2 节)和注销 SAPs(参见 1.6.2.3 节)。呼叫管理器代表客户在网络上发送信令消息以注册或注销

SAPs。

·在面向连接客户的请求下创建一个外出呼叫

当面向连接客户作一个外出呼叫时，呼叫管理器为了建立一个连接，在必要时同网络控制设备进行通信。(参见 1.6.4.1 节)。如果远程 party 接受呼叫，呼叫管理器激活 VC(参见 1.6.3.2 节)。

·向面向连接客户创建和指示一个内入呼叫

呼叫管理器向面向连接客户，指示所有该客户注册的 SAP 的呼叫。(参见 1.6.4.2 节)。在向客户指示内入呼叫前，呼叫管理器为这呼叫发起创建 VC(参见 1.6.3.1 节)，然后发起这个 VC 的激活(参见 1.6.3.2 节)。

·传递改变 QoS 的请求

在信令协议基础上，呼叫管理器能传递来自本地客户的请求，该请求要求改变外出或内入呼叫的 QoS(参见 1.6.5.1 节)，也能传递来自远程 party 关于呼叫 QoS 的改变请求(参见 1.6.5.2 节)。

·传递增加或删除 parties 的请求

呼叫管理器传递本地客户请求，该请求要求从一个一对多呼叫中增加或删除一个 party(参见 1.6.6.1 节和 1.6.6.2 节)。呼叫管理器也传递远程 party 的内入请求，该请求要求将其从一对多呼叫中删除。(参见 1.6.6.3 节)。

·断开呼叫

应面向连接客户请求，呼叫管理器与网络控制设备进行通信来关闭呼叫。(参见 1.6.8.1 节)。应远程 party 请求，呼叫管理器向面向连接客户指示远程 party 关于关闭一个呼叫的请求(参见 1.6.8.2 节)。在断开呼叫过程中，呼叫管理器使 VC 去活(参见 1.6.3.3 节)。如果是呼叫管理器创建的 VC(对一个内入呼叫而言)，呼叫管理器也能删除此 VC(参见 1.6.3.4 节)

·查询和设置信息

呼叫管理器能查询或设置面向对象客户支持的信息(参见 1.6.9.1 节)。呼叫管理器也能响应来自绑定的面向连接客户的查询和设置操作。

另外，呼叫管理器也能查询或设置由微端口或 MCM 的微端口部分支持的信息。(参见 1.6.9.1 节)。

·输入微端口状态指示

呼叫管理器能从一个面向连接微端口输入状态信息(参见 1.6.9.2 节)。

·重置 NIC

呼叫管理器可以使面向连接微端口重置它的 NIC(参见 1.6.10 节)。

1.6.1.3 由微端口执行的面向连接操作

除了控制 NIC 硬件，面向连接微端口执行如下的面向连接功能：

·发送和接收包

面向连接微端口代表面向连接客户或呼叫管理器发送和接收包(参见 1.6.7.1 节和 1.6.7.2 节)。

·创建(建立)VCs

应面向连接客户请求，面向连接微端口为一个外出呼叫 VC 分配和初始化资源。(参见 1.6.3.1 节)。应呼叫管理器请求，面向连接微端口为内入呼叫 VC，或呼叫管理器用于发送 / 接收信令消息的 VC，分配并初始化资源(参见 1.6.3.1 节)。

·激活 VCs

应呼叫管理器请求，面向连接微端口与它的 NIC 通信，使 NIC 准备好通过 VC 接受或传递数据(参见 1.6.3.2 节)。

• 去活 VCs

应呼叫管理器请求，面向连接微端口和它的 NIC 通信，使 NIC 终止一个 VC 的所有通信(参见 1.6.3.3 节)。

• 删除 VCs

应面向连接客户请求，面向连接微端口收回客户建立的 VC 的资源(参见 1.6.3.4 节)。应呼叫管理器请求,面向连接微端口收回由呼叫管理器发起建立的 VC 的资源(参见 1.6.3.4 节)。

• 查询或设置信息

面向连接微端口对面向连接客户或呼叫管理器的查询或设置操作作出响应。(参见 1.6.9.1 节)。

• 指示状态

面向连接微端口能向面向连接客户和呼叫管理器指示它的状态变化或是 NIC 的状态变化。（参见 1.6.9.2 节）

• 重置 NIC

应面向连接客户，呼叫管理器，或 NDIS 的请求，面向连接微端口重置它的 NIC(参见 1.6.10 节)。

1.6.2 地址族和 SAPs 上的操作

呼叫管理器或 MCM 驱动程序必须用 NDIS 注册它的呼叫管理器入口,并且向面向连接客户发布呼叫管理器服务。呼叫管理器或 MCM 通过注册地址族来实现上述操作。

为了用呼叫管理器或 MCM 的呼叫管理器服务，面向连接客户必须用那个呼叫管理器或 MCM 打开一个地址族。为了接收内入呼叫，客户必须用呼叫管理器 MCM 注册一个或多个 SAPs.

下面是关于地址族和 SAPs 的面向连接操作：

• 1.6.2.1 注册并打开一个地址族
• 1.6.2.2 注册一个 SAP
• 1.6.2.3 注销一个 SAP
• 1.6.2.4 关闭一个地址族

1.6.2.1 注册并打开一个地址族

呼叫管理器必须为每个 NIC 注册一个地址族,通过 NIC 向面向连接客户提供呼叫管理器服务。类似地，MCM 驱动程序必须向它管理的 NIC 注册一个地址族。通过注册一个地址族，呼叫管理器或 MCM 用 NDIS 注册了入口，并使得 NDIS 向所有绑定在这块适配卡上的客户，公布呼叫管理器或 MCM 服务。

如果面向连接客户可以使用呼叫管理器或 MCM 公布的服务,它就能用呼叫管理器或 MCM 打开一个地址族。

从呼叫管理器中注册地址族。

在呼叫管理器的 ProtocolBindAdapter 函数用 NdisOpenAdapter 绑定 NIC 微端口后,呼叫管理器调用 NdisCmRegisterAddressFamily 为此绑定注册一个地址族。(见图 1.3)。

图 1.3 用呼叫管理器注册并打开一个地址族

调用 NdisCmRegisterAddressFamily 为呼叫管理器注册呼叫管理器入口,并公布管理器的特殊信令服务。每当呼叫管理器在调用 ProtocolBindAdapter 函数时，并用 NdisOpenAdapter 成功绑定一块 NIC 时，必须注册地址族。

呼叫管理器在它绑定的所有 NIC 驱动程序中，可以支持多个的地址族。呼叫管理器也能在它所绑定的单个 NIC 上绑定多个的地址族。在这种情况下，呼叫管理器必须为该绑定上的每个

地址注册相同的入口。对于一个绑定某个 NIC 驱动程序的客户，只有一个呼叫管理器能为其提供某一种特殊地址族。

从 MCM 驱动程序中注册地址族

MCM 驱动程序在用 **NdisMCmRegisterMiniport** 注册入口后，从 *MiniportInitiallize* 函数中调用 **NdisMCmRegisterAddressFamily** 一次，用 NDIS 注册它的呼叫管理器入口，并且发布它对面向连接客户提供的服务(参见图 1.4)。

图 1.4 用 MCM 注册和打开地址族

NIC 微端口驱动程序有一个内置的面向连接信令支持，即使呼叫管理器是可利用的,也能把自己注册为一个 MCM 驱动程序。这时，MCM 驱动程序取代那个 NIC 的呼叫管理器。

打开一个地址族

呼叫管理器或 MCM 调用 **Ndis(M)CmRegisterAddressFamily**，使得 NDIS 调用面向连接客户的 *ProtocolCoAfRegisterNotify* 函数(如图 1.3 和 1.4)。

ProtocolCoAfRegisterNotify 检查地址族以确定是否客户能使用 CM 或 MCM 服务,以及在呼叫管理器或 MCM 支持的协议基础上，客户是否能修改地址族数据。

如果客户可以利用呼叫管理器服务，*ProtocolCoAfRegisterNotify* 为这个客户的每个地址族分配一个环境并调用 **NdisClOpenAddessFamily**。**NdisClOpenAddessFamily** 向 NDIS 注册了客户的面向连接入口点。

对 **NdisClOpenAddressFamily** 的调用,使 NDIS 调用呼叫管理器或 MCM 的 *ProtocolCmOpenAf* 函数(如图 1.3、1.4 所示)。ProtocolCmOpenAf 保证客户传递一个有效的地址族，并分配和初始化资源，该资源对客户是必须的。*ProtocolCmOpenAf* 也存储了一个由 NDIS 支持的 *NdisAfHandle*,它代表了呼叫管理器和打开地址族的客户之间的联系。

ProtocolCmOpenAf 能同步或异步地完成。为异步完成，呼叫管理器 ProtocolCmOpenAf 函数调用 **NdisCmOpenAddressFamilyComplete**。MCM 的 ProtocolCmOpenAf 函数调用 **NdisMCmRegisterAddressFamilyComplete**。对于 **Ndis(M)CmRegisterAddressFamilyComplete** 的调用,使 NDIS 调用客户的 ProtocolClOpenAfComplete 函数，进而使该客户调用 **NdisClOpenAddressFamily**。

如果客户成功调用 **NdisClOpenAddressFamily**，NDIS 将一个 *NdisAfHandle* 返回给客户，该 *NdisAfHandle* 代表了呼叫管理器和客户之间的联系。

如果一客户接受了内入呼叫，在 ProtocolClOpenAfComplete 函数中，它通常在成功调用 **NdisCIOpenAddressFamily** 之后，调用 **NdisClRegisterSap** 来注册一个或多个 SAPs(参见 1.6.6.2 节)。

如果一客户作外出呼叫，根据外出呼叫的请求，在其 ProtocolClOpenAfComplete 函数中创建一个或多个 VCs(参见 1.6.3.1 节)。

1.6.2.2 注册一个 SAP

如果一个客户接受了内入呼叫，它的 ProtocolClOpenAfComplete 函数(参见 1.6.2.1 节)通常通过 **NdisClRegisterSap**，向呼叫管理器注册一个或多个 SAPs。

图 1.5 呼叫管理器的客户注册一个 SAP

图 1.5 用呼叫管理器注册一个 SAP

图 1.6MCM 的客户注册一个 SAP

图 1.6 用 MCM 注册一个 SAP

通过调用 NdisClRegisterSap，客户请求在某个 SAP 上的内入呼叫的指示。NDIS 将客户支持的 SAP 信息传给呼叫管理器或 MCM 的 *ProtocolCmRegisterSap* 函数来验证其有效性。如果某个 SAP 已经正在使用，或者呼叫管理器或 MCM 客户不支持这个 SAP 说明，请求就会失败。

ProtocolCmRegisterSap 与网络控制设备或其他特殊介质代理通信，为面向连接客户注册 SAP。ProtocolCmRegisterSap 也存储了一个代表 SAP 的 *NdisSapHandle*（由 NDIS 支持）。

ProtocolCmRegisterSap 能异步或同步完成。为了能异步完成，呼叫管理器的 ProtocolCmRegisterSap 函数调用 NdisCmRegisterSapComplete 。MCM 的 ProtocolCmRegisterSap 调用 NdisMCmRegisterSapComplete. 对函数 Ndis(M)CmRegisterSapComplete 的调用，使 NDIS 调用了客户的 ProtocolClRegisterSapcomplete 函数。

如果客户调用 NdisClRegisterSap 成功,NDIS 向客户返回一个代表该 SAP 的 NdisSapHandle。在呼叫管理器注册了一个代表面向连接客户的 SAP 之后，将通过调用 NdisCmDispatchIncomingCall, 向客户通知那个 SAP 的内入呼叫。MCM 调用 NdisMCmDispatchIncomingCall（参见 1.6.4.2 节）。当 SAP 注册未决时（那是调用 ProtocolClRegisterSapcomplete 函数以前）客户就能接收那个 SAP 的内入呼叫了。

1.6.2.3 注销 SAP

面向连接客户用 NdisClDeregisterSap 注销一个 SAP。

图 1.7 显示了一个呼叫管理器的客户注销 SAP

图 1.7 用呼叫管理器注销 SAP

图 1.8 展示一个 MCM 客户注销 SAP.

图 1.8 用 MCM 注销 SAP

对 NdisClDeregisterSap 的调用使得 NDIS 调用呼叫管理器或 MCM 的 *ProtocolClDeregisterSap* 函数。ProtocolClDeregisterSap 与网络控制设备和其他特殊介质代理通信以注销 SAP。另外，ProtocolClDeregisterSap 必须释放所有动态分配给该 SAP 的资源。ProtocolClDeregisterSap 可以同步或异步完成。为了能异步完成，呼叫管理器的 ProtocolClDeregisterSap 调用 NdisCmDeregisterSapComplete 。MCM 的 ProtocolClDeregisterSap 调用 NdisMCmDeregisterSapComplete 。Ndis(M)CmDeregisterSapComplete 通知 NDIS 和客户,呼叫管理器已经完成注销 Sap 的请求，这时 ProtocolClDeregisterSap 返回 NDIS_STATUS_PENDING。

调用 Ndis(M)CmDeregisterSapComplete 使得 NDIS 调用客户的 ProtocolClDeregisterSapComplete 函数。调用 ProtocolClDeregisterSapComplete 指示，客户先前对 NdisCmDeregisterSap 的调用已经被呼叫管理器或 MCM 处理。

注意，客户可以注销 SAP,而不影响那个 SAP 已被接受的内入呼叫，也不影响那个呼叫的 VC。

1.6.2.4 关闭一个地址族

面向连接客户调用 NdisClCloseAddressFamily 删除自身、呼叫管理器、NIC 之间的联系。

图 1.9 显示了一个呼叫管理器客户关闭一个地址族。

图 1.9 通过呼叫管理器关闭一个地址族。

图 1.10 显示了 MCM 客户关闭一个地址族。

图 1.10 通过 MCM 关闭一个地址族。

客户在关闭某个绑定上所有打开的 VCs(参见 1.6.8.1 节)和注销所有 SAPs(参见 1.6.2.3 节)后，客户将从 ProtocolUnbindAdapter 函数中调用 **NdisClCloesAddressFamily**。无论何时，当客户绑定的 NIC 被删除或重新配置时，NDIS 调用客户的 ProtocolUnbindAdapter 函数。调用 **NdisClCloesAddressFamily** 使得 NDIS 调用呼叫管理器或 MCM 的 *ProtoclCmCloseAf* 释放所有资源和 / 或使其去活，这些资源由呼叫管理器在其 *ProtocllCmOpenAf* 函数中分配。在客户关闭那个地址族后，呼叫管理器也应该撤消所有代表面向连接客户所作的操作。ProtocllCmCloseAf 可以同步或异步完成。为能异步完成，呼叫管理器的 ProtocllCmCloseAf 函数调用了 **NdisCmCloesAddressFamilyComplete**。MCM 的 ProtocllCmCloseAf 函数调用 **NdisMCmCloesAddressFamily**。如果在一个地址族上，有请求或连接处于打开状态，呼叫管理器对客户关闭地址族的要求作以下两种响应：

- 呼叫管理器或 MCM 用 NDIS_STATUS_NOT_ACCEPTED 驳回请求。
- 呼叫管理器或 MCM 返回 NDIS_STATUS_PENDING。在客户关闭所有呼叫且注销的所有 SAPs 后，呼叫管理器或 MCM 可以关闭地址族，并调用 **Ndis(M)CmCloesAddressFamilyComplete** 通知客户。这是用户所期望的响应。

对 **Ndis(M)CmCloesAddressFamilyComplete** 的调用，使 NDIS 调用面向连接客户的 *ProtocolClCloseAfComplete* 函数，该客户先前调用了 **NdisClCloseAddressFamily**。

1.6.3 VCs 上的操作

网络上的所有面向连接的通信都发生在虚连接（VCs）之上。VC 操作在下述章节中描述：

- 1.6.3.1 创建 VC
- 1.6.3.2 激活 VC
- 1.6.3.3 使 VC 去活
- 1.6.3.4 删除 VC

1.6.3.1 创建 VC

在作一个外出呼叫前，面向连接客户发起一个虚连接(VC)的创建。在向面向连接客户指示一个内入呼叫时，呼叫管理器或 MCM 也发起一个 VC 的创建。在 VC 创建并激活后，客户数据能在此 VC 上传递和 / 或接收。

呼叫管理器和 MCM 也能发起创建一个 VC，在此 VC 上，网络成员(如网络交换机)之间可以交换信令消息。

由客户发起的 VC 创建

在调用 **NdisClMakeCall**(参见 1.6.4.1 节)前，面向连接客户调用 **NdisCoCreateVC** 以发起 VC 的创建。图 1.11 显示了呼叫管理器的客户发起一个 VC 的创建。

图 1.11 由客户发起的 VC 的创建（呼叫管理器）

图 1.12 显示了一个 MCM 客户发起 VC 的创建。

图 1.12 由客户发起的 VC 的创建。(MCM)

当面向连接客户调用 **NdisCoCreateVC** 时，作为一个同步操作，NDIS 调用呼叫管理器的 *ProtocolCoCreateVC* 函数和微端口 *MiniportCoCreateVC* 函数。(见图 1.11)。NDIS 向 ProtocolCoCreateVC 和 MiniportCoCreateVC 传递一个代表 VC 的 *NdisVCHandle*.如果调用 **NdisCoCreateVc** 成功，NDIS 向 **NdisCoCreateVc** 返回 NdisVCHandle。

ProtocolCoCreateVC 分配并初始化任何动态资源和结构，这些资源和结构被呼叫管理器用于即将激活的 VC 的后续操作上。MiniportCoCreateVc 分配并初始化任何资源，这些资源被微端口用来维持 VC 的状态信息。ProtcolCoCreateVc 和 MiniportCoCreateVc 贮存 *NdisVcHandle*。

一个 MCM 的面向连接客户对 **NdisCoCreateVc** 的调用,使得 NDIS 调用 ProtocolCoCreateVC 函数(见图 1.12)。在这种情况下，ProtocolCoCreateVC 执行了 VC 资源的分配和初始化。这里没有 MiniportCoCreateVC 的调用(也有可能是内部的)，MCM 不支持这个函数。

呼叫管理器发起的 VC 创建

在用**NdisCmDispatchIncomingCall** 向面向连接客户指示一个内入呼叫前(参见 1.6.4.2 节)，呼叫管理器调用 **NdisCoCreateVC** 以发起一个 VC 的创建(参见图 1.3)。

图 1.3 呼叫管理器发起的 VC 创建

当呼叫管理器调用 **NdisCoCreateVC** 时，作一个同步操作，NDIS 调用注册 SAP(呼叫正是通过这个 SAP 接收的)的面向连接客户的 *ProtocolCoCreateVC* 函数。NDIS 也调用了微端口的 *MiniportCoCreateVC* 函数。NDIS 向 *ProtocolCoCreateVC* 和 *MiniportCoCreateVC* 传递一个代表 VC 的 *NdisVCHandle*。如果 **NdisCoCreateVC** 的调用成功，NDIS 向 **NdisCoCreateVC** 返回 *NdisVCHandle*。

MCM 发起的 VC 创建

在用 **NdisMcmDispatchIncomingCall** 向面向连接客户指示一个内入呼叫前(参见 1.6.4.2 节)，MCM 调用 **NdisMCmCreateVC** 以发起 VC 的创建(见图 1.14)

图 1.14 MCM 发起的 VC 创建

当 MCM 调用 **NdisMCmCreateVC** 时，作为 **NdisMCmCreateVC** 返回前的同步操作，NDIS 将调用面向连接客户的 *ProtocolCoCreateVC* 函数。NDIS 向 *ProtocolCoCreateVC* 传递一个代表 VC 的 *NdisVCHandle*。如果对 **NdisMCmCreateVc** 的调用成功，NDIS 向 **NdisMCmCreateVc** 返回一个 *NdisVCHandle*。

ProtocolCoCreateVc 分配并初始化所有动态资源和结构，客户用这些资源和结构来执行 VC 上的后续操作。*ProtocolCoCreateVC* 也贮存 *NdisVCHandle*。

注意，当 MCM 创建一个 VC,用于网络成员间交换信令消息时，它并不使用 **NdisXxx** 创建 VC。实际上，MCM 不用 **NdisXxx** 来创建、激活，去活，或删除这样的 VCs。相反，MCM 在内部执行这些操作。这样的 VCs 对 NDIS 来说是不可见的。

1.6.3.2 激活 VC

在虚连接创建后(参见 1.6.3.1 节)，必须在能进行数据传递和接收前，激活该 VC。呼叫管理器调用 **NdisCmActivateVC** 发起的激活(见图 1.15)。

图 1.15 呼叫管理器发起 VC 的激活

MCM 通过调用 **NdisMCmActivateVC** 发起 VC 的激活(见图 1.16)。

图 1.16 MCM 发起 VC 的激活

如果本地客户和远程 party 成功地商定了 VC 参数的改变，呼叫管理器或 MCM 可以再次激活一个活动 VC。(参见 1.6.5.1 节和 1.6.5.2 节)。为了改变活动 VC 的呼叫参数，呼叫管理器或 MCM 可以为单个 VC 多次调用 **Ndis(M)CmActivateVc**。

对于一个客户发起的外出呼叫,呼叫管理器或MCM在和远程party交换证实商定协议的包后，或者在交换机上成功建立呼叫之后，将立即调用 **Ndis(M)CmActivateVC**。在调用

Ndis(M)CmMakeCallComplete 通知 NDIS(和客户)外出呼叫已完成之前(参见 1.6.4.1 节),呼叫管理器或 MCM 先调用 Ndis(M)CmActivateVC。对于一个内入呼叫，呼叫器或 MCM 通常在成功调用 **NdisCo(M)CmCreateVC** 之后，以及调用 **Ndis(M)cmDispatchIncomingCall** 之前(参见 1.6.4.2)，调用 **Ndis(M)CmActivateVC**。

呼叫管理器对 **NdisCmActivateVc** 的调用，使 NDIS 调用 NIC 微端口函数 *MiniportCoActivateVc* 函数。*MiniportCoActivateVc* 函数必须检查该 VC 的呼叫参数,以证实适配器能够支持请求的呼叫。如果呼叫参数是可接受的，*MiniportCoActivateVc* 必须和适配器通信，从而使其准备好(例如，程序的接收缓冲区)通过虚连接传送或接收数据。如果呼叫参数不被支持，微端口驳回请求。

MiniportCoActivateVc 可以同步或异步地完成。调用 **NdisMCoActivateVCComplete** 使得 NDIS 调用呼叫管理器的 *ProtocolCmActivateVcComplete* 函数，*ProtocolCmActivateVcComplete* 必须检查 **NdisMCoActivateVcComplete** 返回的状态,以确认虚连接已成功激活。如果微端口没有成功激活 VC，呼叫管理器无法在 VC 上通信。*ProtocolCmActivateVcComplete* 必须处理网络介质请求，在将控制返回给 NDIS 之前，确保虚连接准备好传送数据。

MCM 对 **NdisMCmActivateVC** 的调用通知 NDIS，它已在一个新生成的 VC 上创建了呼叫和介质参数，或者改变了一个已存在 VC 的呼叫参数。这通知 NDIS，MCM 已使 NIC 准备好传送该 VC 的数据。NDIS 通过调用 MCM 的 *ProtocolCmActivateVcComplete* 函数，完成激活序列。

MCM 调用 **NdisMCmActivateVC** 激活传送和 / 或接收客户数据的 VCs，但不能激活用于 MCM 和网络成员(如交换机)之间交换信令消息的 VCs。MCM 激活一个信令 VC 时，不调用任何 **NdisXxx** 函数，所以任何用于信令交换的 VC 对于 NDIS 来说是不可见的。

1.6.3.3 使 VC 去活

作为关闭一个外出或内入呼叫的基本步骤，通常在和断开呼叫的网络成员进行包交换之后，呼叫管理器调用 **NdisCmDeactivateVC**（参见 1.6.8.1 和 1.6.8.2 节）。MCM 调用 **NdisMCmDeactivateVC** 来做相同的工作。

对 **NdisCmDeactivateVC** 的调用使得 NDIS 调用 NIC 驱动程序的 *MiniportCoDeactivateVC*(参见图 1.17）. *MiniportCoDeactivateVC* 与它的网络适配器通信,以终止这个 VC 上的所有通信(例如，清除该适配器上的发送和接收缓冲区)。

图 1.17 呼叫管理器发起的 VC 去活

在使 VC 去活之前，微端口必须先完成该 VC 上未完成的传输。那就是说，得等到所有正在发送和已经指示的包的接收完成。在 VC 去活后，微端口不能再在这个 VC 上指示接收或传输发送。

注意，*MiniportCoDeactivateVC* 并不删除 VC。一个不再使用的 VC 的生成者(如客户，呼叫管理器，或 MCM)调用 **NdisCoDeleteVC** 来删除 VC(参见 1.6.3.4 节)。去活的 VC 可以被面向连接客端，呼叫管理器，或 MCM 重新激活。(参见 1.6.3.2 节)。

MiniportCoDeactivateVC 能同步或异步完成。对 **NdisMCoDeactivateVcComplete** 的调用，使 NDIS 调用呼叫管理器的 *ProtocolCmDeactivateVcComplete* 函数。去活的完成意味着 VC 激活时所用的所有参数不再有效。除非用新的参数集重新激活 VC，否则无法使用此 VC。

MCM 对 **NdisMCmDeactivateVC** 的调用通知 NDIS,它已经使一个 VC 去活，或改变了一个 VC 的呼叫参数（见图 1.8)。NDIS 通过调用 MCM 的 *ProtocolCmDeactivateVcComplete* 完成去活操作序列。

图 1.18 MCM 发起的 VC 去活

1.6.3.4 删除 VC

只有发起 VC 创建的面向连接客户、呼叫管理器和 MCM 才能启动那个 VC 的删除操作。客户删除先前为一个外出呼叫而创建的 VC，呼叫管理器或 MCM 删除先前为一个内入呼叫而创建的 VC，呼叫管理器删除先前为交换信令消息而创建的 VC。(MCM 不调用 NDIS 来删除它先前为交换信令消息而创建的 VC。MCM 用内部操作删除该 VC，这对 NDIS 是不可见的。)面向连接客户或呼叫管理器用 NdisCoDeleteVC 发起一个 VC 的删除。图 1.19 显示了呼叫管理器的客户发起 VC 的删除。

图 1.19 面向连接客户通过呼叫管理器发起的 VC 的删除

图 1.20 显示了 MCM 客户发起的 VC 删除。

图 1.20 面向连接客户通过 MCM 发起的 VC 的删除。

图 1.21 显示了呼叫管理器发起的 VC 删除。

图 1.21 呼叫管理器发起的 VC 删除。

当客户或呼叫管理器调用 **NdisCoDeleteVC**，或者当 MCM 调用 **NdisMCmDeleteVC** 时，在此 VC 上肯定已经不存在呼叫，而且 VC 肯定已经去活了(参见 1.6.3.3 节)。达到这些要求表示下面的条件已经满足：

•如果呼叫断开由本地客户发起，那么那个客户已经用 *NdisVCHandle* 调用 **NdisClCloseCall** 并且已成功地完成(参见 1.6.8.1 节)。

• 如果呼叫断开由远程客户发起，那么呼叫管理器已经用 *NdisVCHandle* 调用了 **NdisCmDeactivateVC**，或者 MCM 已经用 *NdisVCHandle* 调用了 **NdisMCmDeactivateVC**。并且已成功地完成(参见 1.6.8.2 节)。

客户或呼叫管理器调用 **NdisCoDeleteVC**,使 NDIS 调用 NIC 驱动程序的 *MiniportCoDeleteVC* 函数和客户或呼叫管理器的 *ProtocolCoDeleteVC* 函数。调用者与客户或呼叫管理器共享同一个 *NdisVCHandle*(参见图 1.19，1.20 和 1.21)。*MinportCoDeleteVC* 释放所有分配给 VC 的资源，包括该 VC 的微端口环境。*ProtocolCoDeleteVC* 释放客户或呼叫管理器用来执行操作或跟踪 VC 状态的所有资源。

MinportCoDeleteVC 和 *ProtocolCoDeleteVC* 都是不可能返回 NDIS_STATUS_PENDING 的同步函数。

MCM 用 **NdisMCmDeleteVC** 发起一个 VC 删除(参见图 1.22)。

图 1.22MCM 发起 VC 的删除

MCM 对 **NdisMCmDeleteVC** 的调用使得 NDIS 调用客户的 *ProtocolCoDeleteVC* 函数，该客户和 MCM 分享同一个 *NdisVCHandle*。

当 **NdisCoDeleteVC** 或 **NdisMCmDeleteVC** 返回控制时，*NdisVCHandle* 将不再有效。

1.6.4 创建呼叫

在作一个外出呼叫时，面向连接客户发起一个呼叫的创建。当向面向连接客户指示一个内入呼叫时，呼叫管理器或 MCM 发起呼叫的创建。

1.6.4.1 进行呼叫

图 1.23 显示一个客户通过呼叫管理器作一个外出呼叫。

图 1.23 通过呼叫管理器作一个呼叫

图 1.24 显示一个客户通过 MCM 作一个外出呼叫

图 1.24 通过 MCM 作一个呼叫

在作外出呼叫之前，面向连接客户必须完成以下操作：

• 在类型 CO_CALL_PARAMETERS 结构中初始化呼叫参数。呼叫管理器或 MCM 通常用客户说明的呼叫参数，创建呼叫并衍生出微端口使用的介质参数。

• 用 NdisCoCreateVC 发起 VC 的创建（参见 1.6.3.1）。

在 NdisCoCreateVC 成功返回时，客户调用 NdisClMakeCall 创建呼叫(参见图 1.23 和 1.24)。在对 NdisClMakeCall 调用中，客户传递一个初始化过的指向 CO_CALL_PARAMETERS 结构的指针。客户同时也传递一个 *NdisVCHandle*(由 NdisCoCreateVC 返回的)，这个 *NdisVCHandle* 指出了客户用来为呼叫传送数据的 VC。如果客户作一个多点呼叫(具有多个远程 party 呼叫)，它也传递一个 *PotocolPartyContext*，此参数以说明客户分配的常驻环境区的句柄，在这个环境区中客户为多点 VC 上的每个 party 维护状态信息。

对 NdisClMakeCall 的调用使 NDIS 将这个请求传给呼叫管理器或 MCM 的 *ProtocolCmMakeCall* 函数，呼叫管理器或 MCM 与客户共享这个 *NdisVCHandle*。*ProtocolCmMakeCall* 必须验证由客户创建的呼叫参数。*ProtocolCmMakeCall* 同网络控制设备进行通信(交换信令消息)，来创建一个连接。呼叫管理器调用 NdisCoSendPackets 发起这个交换(参见 1.6.7.1 节)。而 MCM 不会调用 NdisCoSendPackets。相反，它通过网络直接送数据。

呼叫管理器或 MCM 在与相关网络成员协商过程中，可以修改客户支持的呼叫参数，返回不同于客户最初传给 NdisClMakeCall 的传输参数。

一个传给 *ProtocolCmMakeCall* 的 *NdisPartyHandle* 指示客户创建的 VC 将用于多点呼叫。呼叫管理器或 MCM 必须分配并初始化的所有必需的资源，那些资源用于维护每个 Party 的状态信息和多点呼叫控制。

在呼叫管理器已经作了介质所要求的所有与网络硬件的通信后，它必须调用 NdisCmActivateVc 发起 VC 的激活，呼叫数据将在这个 VC 上发送或接收。(参见 1.6.3.2 节)。MCM 必须调用 NdisMCmActivateVc。

当 NIC 微端口准备好在一个 VC 上作数据传输时(在 VC 被激活之后)，呼叫管理器调用 NdisCmMakeCallComplete，MCM 调用 NdisMCmMakeCallComplete。在这时，呼叫管理器或 MCM 应已经完成了 VC 的激活。

在对 Ndis(M)CmMakeCallComplete 的调用中，呼叫管理器或 MCM 把 VC 的呼叫参数，以指向结构类型 CO_CALL_PARAMETERS 的指针来传递。如果呼叫管理器已经修改了呼叫参数，它会在 CO_CALL_PARAMETERS 结构中设置 CALL_PARAMETERS_CHANGED 标志通知客户。

对 Ndis(M)CmMakeCallComplete 的调用，使 NDIS 调用发出外出呼叫的客户的 *ProtocolClMakeCallComplete* 函数。*ProtocolClMakeCallComplete* 指示:呼叫管理器已经完成了客户建立虚连接的请求(用 NdisClMakeCall 提出的)。

如果客户建立外出呼叫成功，*ProtocolClMakeCallComplete* 应该检查 CALL_PARAMETERS_CHANGED 标志，确定客户最初说明的呼叫参数是否已被修改。如果标志被设置，说明呼叫参数已被修改，*ProtocolClMakeCallComplete* 应该检查返回的参数，以确定参数是否可接受。如果呼叫参数是可接受的，*ProtocolClMakeCallComplete* 返回控制。如果呼叫参数不可接受并且信令协议允许再协商，客户可以调用 NdisClModifyCallQoS 来请求改变呼叫参数（参见 1.6.5.1 节）。如果信令协议不允许再协商，则 *ProtocolClMakeCallComplete* 必须用 NdisClCloseCall 断开呼叫(参见 1.6.8.1 节)。

1.6.4.2 指示内入呼叫

来自网络的信令消息提醒呼叫管理器或 MCM 有一个内入呼叫。呼叫管理器或 MCM 从这些信令

消息中,抽取呼叫参数，包括内入呼叫的 SAP。
图 1.25 显示呼叫管理器指示一个内入呼叫

图 1.25 通过呼叫管理器指示内入呼叫

图 1.26 显示了 MCM 指示内入呼叫

图 1.26 通过 MCM 指示内入呼叫

如果对于呼叫管理器或 MCM 来说,内入呼叫参数是不可接受的,那么如果信令协议允许协商，它就可以和远程 party 协商改变呼叫参数。内入呼叫所导向的客户在收到来自呼叫管理器或 MCM 的呼叫指示后，也可以协商呼叫参数。(参见 1.6.5.1 节)。如果呼叫中管理器或 MCM 无法和远程 party 商定一个可接受的呼叫参数，可能会拒绝这次呼叫。在上述情况下，根据具体的信令协议进行相应处理。

在向客户指示内入呼叫前，呼叫管理器或 MCM 必须签别呼叫所导向的 SAP。这个 SAP 必须已被客户注册(参见 1.6.2.2 节)。呼叫管理器或 MCM 必须发起虚连接的创建(参见 1.6.3.1 节)并发起激活此 VC (1.6.3.2 节)。然后，呼叫管理器或 MCM 向注册这个 SAP 的客户指示内入呼叫。呼叫管理器用 **NdisCmDispatchIncomingCall** 指示一个内入呼叫。在调用 **Ndis(M)CmDispatchIncomingCall** 时，呼叫管理器或 MCM 传递了如下的信息：

- 用于指出呼叫所针对的 SAP 的 *NdisSapHandle*
- 用于指出内入呼叫的 VC 的 *NdisVCHandle*
- 指向结构类型 CO_CALL_PARAMETERS 的指针，该结构中包含了呼叫参数。

对 **Ndis(M)CmDispatchInComingCall** 的调用使得 NDIS 调用客户的 *ProtocolClInComingCall* 函数，在这个函数中，客户接受或者拒绝该连接请求。*ProtocolClInComingCall* 应该验证 SAP、VC 和呼叫参数的有效性。

ProtocolClInComingCall 能同步完成，或者，它能用 **NdisClInComingCallComplete** 异步完成，并返回 NDIS_STATUS_PENDING。对 **NdisClInComingCallComplete** 的调用,使 NDIS 调用呼叫管理器或 MCM 的 *ProtocolCmInComingCallComplete* 函数。

由 *ProtocolClInComingCall* 同步完成所返回的 NDIS_STATUS 码，或是由 *NdisClInComingCallComplete* 支持的 NDIS_STATUS 码,都指示了客户对内入呼叫的接受或拒绝。客户也在缓冲区的 CO_CALL_PARAMETERS 结构中，返回呼叫参数。如果客户发现呼叫参数不可接受，只要信令协议允许，它能在 CO_CALL_PARAMETERS 结构中设置 CALL_PARAMETERS_CHANGED 标志来请求改变呼叫参数,同时在缓冲区的 CO_CALL_PARAMETERS 结构中提供修订过的呼叫参数。

如果客户接受一个内入呼叫，呼叫管理器或 MCM 应发送信令消息，以指示呼叫实体，此次呼叫已被接受。否则，呼叫管理器或 MCM 应该发送信令消息指示呼叫已被拒绝。如果客户正在请求呼叫参数，呼叫管理器或 MCM 发送信令消息请求改变呼叫参数。

如果客户接受呼叫，或者远程 party 接受客户改变呼叫参数的请求，呼叫管理器调用 **NdisCmDispatchCallConnected**,MCM 调用 **NdisMCmDispatchCallConnected** 。调用 **Ndis(M)CmDispatchCallConnected** 使 NDIS 调用客户的 *ProtocolClCallConnnected* 函数。

如果客户拒绝呼叫，并且呼叫管理器或 MCM 已经为此次内入呼叫激活 VC，那么呼叫管理器或 MCM 调用 **Ndis(M)CmDeactivateVC** 使 VC 去活。呼叫管理器用 **NdisCoDeleteVC** 发起 VC 的删除。MCM 用 **NdisMCmDeleteVC** 发起 VC 的删除。

如果客户接受呼叫，但端到端的连接没有成功建立(例如，远程 party 断开链接)，那么呼叫管理器或 MCM 不调用 **Ndis(M)CmDispatchCallConnected** 。相反，它调用 **Ndis(M)CmDispatchIncomingCloseCall**, 这使得 NDIS 调用客户的 *PtotocolClIncomingCloseCall* 函数。然后，客户必须调用 **NdisClCloseCall**，完成呼叫的断开。接着，呼叫管理器或 MCM 调用 **Ndis(M)CmDeactivateVC** 使 VC 去活(参见 1.6.3.3 节)。

呼叫管理器用 NdisCoDeleteVC 发起 VC 的删除，MCM 用 NdisMCmDeleteVC 发起 VC 的删除(参见 1.6.3.4 节)。

1.6.5 改变活动 VC 的 QoS

面向连接客户能请求改变一个活动 VC 的 QoS。远程客户也能作如此请求。在这种情况下，呼叫管理器或 MCM 驱动程序指示远程客户的 QoS 改变请求。

1.6.5.1 客户发起的改变呼叫参数请求。

客户用 NdisClModifyCallQoS 请求改变活动 VC 的 QoS。
图 1.27 显示了呼叫管理器的改变 QoS 的客户请求。
图 1.28 显示了 MCM 的客户的改变 QoS 的请求。

图 1.27 客户通过呼叫管理器发起呼叫参数的改变请求

图 1.28 客户通过 MCM 发起呼叫参数的改变请求

在对 NdisClModifyCallQoS 的调用中，客户提供了如下的信息：
• 指明 VC 的 *NdisVCHandle*
• 指向 CO_CALL_PARAMETERS 结构的指针，该结构容纳了客户正在请求的呼叫参数。
客户可否请求改变 QoS 的环境是由信令协议决定的。
对 NdisClModifyCallQoS 的调用使得 NDIS 调用呼叫管理器或 MCM 的 *ProtocolCmModifyQoS* 函数，该函数输入 *NdisVCHandle* 和由客户传给与 NdisClModifyCallQoS 的缓冲区的 CO_CALL_PARAMETERS 结构。*ProtocolCmModifyQoS* 与网络控制设备或其他介质成员通信，为一个虚连接修改介质呼叫参数。
在和网络通信并断定改变成功之后，呼叫管理器必须调用 NdisCmActivateVC(MCM 必须调用 NdisMCmActivateVC)，用新的呼叫参数激活 VC。
如果网络不接收新的呼叫参数或者微端口不接受参数，呼叫管理器或 MCM 将 VC 状态恢复到修改以前，并返回 NDIS_STATUS_FAILURE。
为了指示客户改变 QoS 请求的状态，呼叫管理器调用 NdisCmModifyQoSComplete,MCM 调用 NdisMCmModifyCallQoSComplete.
在调用中，呼叫管理器或 MCM 传递如下信息：
• 指示请求状态的 NDIS_STATUS
• 指明 VC 的 *NdisVCHandle*
• 指向 CO_CALL_PARAMETERS 结构的指针，该结构容纳了 VC 的参数。
如果信令协议允许，呼叫管理器或 MCM 可以将修改过的呼叫参数返回给客户。这些修改可以是网络协商的结果，或者也可由呼叫管理器和 MCM 本身提供。呼叫管理器或 MCM 在 CO_CALL_PARAMETERS 结构中设置 CALL_PARAMETERS_CHANGE 标志，指示已修改了呼叫参数。
对 Ndis(M)CmModifyCallQoSComplete 的调用使 NDIS 调用客户的 *ProtocolClModifyCallQoSComplete* 函数。NDIS 向 *ProtocolClModifyCallQoSComplete* 传递信息如下：
• 指示客户改变 QoS 请求状态的 NDIS_STATUS
• 指明 VC 的 *ProtocolVCContext* 的句柄
• 指向 CO_CALL_PARAMETERS 结构的指针，该结构包含了呼叫管理器或 MCM 传给 Ndis(M)CmModifyCallQoSComplete 的呼叫参数。

如果在 CO_CALL_PARAMETERS 结构中，CALL_PARAMETERS_CHANGED 标志被设置，客户必须检查返回的呼叫参数，并确定是否接受修改。如果客户的 NdisClModifyCallQOS 调用成功，*ProtocolClModifyCallQoSComplete* 接收 QoS 修改并返回控制。否则，在信令协议允可的条件下，并且客户开发者已对协商次数作了合理限制时，*ProtocolClModifyCallQoSComplete* 和呼叫管理器作进一步协商。当呼叫管理器拒绝改变 QoS 的请求，并且以前的 QoS 不再可接受时，*ProtocolClModifyCallQoSComplete* 调用 NdisClCloseCall 断开一个呼叫(参见 1.6.8.1 节)。

1.6.5.2 改变呼叫参数的内入请求

网络信令消息提醒呼叫管理或 MCM，有远程 party 的内入请求，该请求要求改变活动 VC 的呼叫参数。信令协议来决定是否呼叫管理器或 MCM 支持活动 VC、QoS 的动态改变。例如，ATM UNI3.1 不支持活动 VC 的动态 QoS 改变。

图 1.29 显示一个通过呼叫管理器修改呼叫参数的请求

图 1.29 通过呼叫管理器改变 QoS 的内入请求

图 1.30 显示一个通过 MCM 改变呼叫参数的内入请求

图 1.30 通过 MCM 改变呼叫参数的内入请求

在收到改变呼叫参数的内入请求之后，呼叫管理器将修改过的呼叫参数传给 NdisCmActivateVC,把 QoS 的变化通知给 NIC 驱动程序。MCM 将修改过的呼叫参数传给 NdisMCmActivateVC(参见 1.6.3.2 节)。如果 NIC 驱动程序接受改变后的呼叫参数，呼叫管理器调用 NdisCmDispatchIncomingCallQoSChange(参见图 1.29)。MCM 调用 NdisMCmDispatchIncomingCallQoSChange(参见图 1.30)。呼叫管理器或 MCM 向 Ndis(M)CmDispatchIncomingCallQoSChange 传递 *NdisVCHandle* 和缓冲区的 CO_CALL_PARAMETERS 结构。

客户接受 QoS 的改变，并更新它所维护的 QoS 的状态信息，然后返回控制。如果不接受修改，在信令协议允可的条件下，客户可用 NdisClModifyCallQoS 来重新协商呼叫参数。(参见 1.6.5.1 节)否则，客户用 NdisClCLoseCall 断开呼叫。(参见 1.6.8.1 节)。在 *ProtocolClIncomingCallQoS* 返回后，呼叫管理器或 MCM 把客户的接受或拒绝反应传给远程 party。

1.6.6 增加和删除 Parties

面向连接客户可以在一个由其通过 NdisClMakeCall 发起的外出多点呼叫中增加一个 Party。面向连接客户可以自己或应远程 party 请求，从多点呼叫中删去一个 Party。

1.6.6.1 把一个 Party 加入到多点呼叫

客户用 NdisClAddParty 请求将 Party 加入到多点呼叫中。客户只能将一个 Party 加入到已存在的多点各叫中。

图 1.31 显示了呼叫管理器客户请求将一个 Party 加入到多点呼叫中的过程。

图 1.31 通过呼叫管理器将一个 Party 加入到多点呼叫中。

图 1.32 显示了 MCM 客户请求将一个 Party 加入到多点呼叫中的过程。

图 1.32 通过 MCM 将一个 Party 加入到多点呼叫中。

在调用 NdisClAddParty 之前，客户必须为加入的 Party 分配和初始化环境区。在调用 NdisClAddParty 时，客户传递指向环境区的指针 *ProtocolPartyContext* 和指向环境区内一

个变量的指针 *NdisPatyHandle*.除了 *NdisVCHandle* 和 *ProtocolPartyContext*，客户向 **NdisClAddParty** 传递呼叫参数(缓冲区的 CO_CALL_PARAMETERS)。下面的网络介质决定了，客户能是否需要说明多点呼叫中每个 party 的传输参数。例如，多点 VC 中的传输参数对于 ATM 中的所有 parties 来说都是相等的。

对 **NdisClAddparty** 调用，使 NDIS 将此请求传给呼叫管理器或 MCM 的 *PotocolCmAddParty* 函数，用此函数共享 *NdisVCHandle*.NDIS 将下边的信息传给 *ProtocolCmAddParty*:

- 向呼叫指示 VC 的 *CallMgrVCContext*
- 包含由客户传给 **NdisClAddParty** 的呼叫参数的 CO_CALL_PARAMETERS 结构的指针。
- 指出所增加 Party 的 *NdisPartyHandle*

*ProtocolCmAddParty*分配并初始化增加Party所需的任何动态资源。从*ProtocolCmAddParty* 中，呼叫管理器或 MCM 和网络控制设备或其他网络介质通信，从而在多点呼叫中增加一个 Party.如果客户传递的呼叫参数并不与已创建的多点 VC 相匹配，呼叫管理器或 MCM 会：

- 只要网络介质支持，就为每个 Party 单独创建传输参数。
- 重置已创建的 VC 的传输参数
- 改变呼叫参数
- 拒绝客户增加一个 Party

ProtocolCmAddParty 可以同步完成。在使用呼叫管理器的情况下，可以用 **NdisCmAddPartyComplete**异步完成;在使用MCM的情况下,可以用**NdisMCmAddPartyComplete** 异步完成。无论是呼叫管理器或 MCM 采用同步或异步方式完成操作，都要向 NDIS 传递缓冲区中的呼叫参数。

对 **Ndis(M)CmAddPartyComplete** 的调用，使 NDIS 调用客户的 *ProtocolClAddPartyComplete* 函数。如果客户增加 Party 的请求成功，并且如果信令协议允许呼叫管理器或 MCM 修改呼叫参数，*ProtocolClAddPartyComplete* 应该检查缓冲区中的 CO_CALL_PARAMETERS 结构的 CALL_PARAMETERS_CHANGED 标志，从而确定是否呼叫参数已改变。如果发现 CO_CALL_PARAMETERS 的改变是不可接受的，则由信令协议决定客户应采取的措施。在这种情况下，客户通常调用 **NdisClDropParty**(参见 1.6.6.2 节)

1.6.6.2 从多点呼叫中删除 Party

作为多点呼叫根的面向连接客户,最后必须用 **NdisClDropParty** 或 **NdisClCloseCall** 删除呼叫中的每一个 Party.

在如下情况下，客户从呼叫中删除一个 Party：

- 在用 **NdisClCloseCall** 发起多点呼叫的关闭操作前(1.6.8.1 节)，客户必须使用一系列 **NdisClDropParty** 的调用,将 Party 删到只剩一个。客户用 **NdisClCloseCall** 删除最后一个 Party。
- 作为对远程 Party 从多点呼叫中删除请求的响应（参见 1.6.6.3），客户在它的 *ProtocolClIncomingDropParty* 函数中调用 **NdisClDropParty**。
- 在客户增加一个 Party 的尝试失败后，客户必须调用 **NdisClDropParty**。

客户对 **NdisClDropParty** 的调用，使 NDIS 调用呼叫管理器或 MCM 的 *ProtocolCmDropParty* 函数，该呼叫管理器或 MCM 与多点 VC 共享同一 *NdisVCHandle*。

图 1.33 显示了呼叫管理器的客户请求删除一个 Party

图 1.33 通过呼叫管理器从多点呼叫中删除一个 Party

图 1.34 显示了 MCM 的客户请求删除一个 Party

图 1.34 通过 MCM 从多点呼叫中删除一个 Party

ProtocolCmDropParty 和网络控制设备通信，从现存的多点呼叫中删除一个 Party。NDIS 向

ProtocolCmDropParty 传递一个含有数据的缓冲区的指针(这些数据在对 NdisClDropParty 的调用中，由客户提供)。在连接删除前，*ProtocolCmDropParty* 必须发送这些数据。

ProtocolCmDropParty 可以同步完成，在使用呼叫管理器情况下，也可用 NdisCmDropPartyComplete 异步完成,在使用 MCM 情况下,也可用 NdisMCmDropPartyComplete 异步完成。对 Ndis(M)CmDropPartyComplete 的调用，使 NDIS 调用客户的 *ProtcolClDropPartyComplete* 函数。如果客户处于断开多点 VC 的过程中，*PorotcolClDropPartyComplete* 可以用任何有效的 *NdisPartyHandle* 来调用 NdisClDropParty,该 *NdisPartyHandle* 指向活动多点 VC 的一个 Party.如果多点 VC 中只有一个 Party 了，客户将其 *NdisPartyHandle* 传给 NdisClCloseCall,删除这个 Party(参见 1.6.8.1 节)。

1.6.6.3 从多点呼叫中删除一个 Party 的内入请求

来自网络的信令消息提醒呼叫管理器或 MCM，有一个来自远程 party 的内入请求，该请求要求从多点呼叫中删除那个 Party。呼叫管理器或 MCM 检测到数据在 VC 上传输有网络阻碍时，会发出内入请求删除一个 Party.

如果删除的 Party 不是 VC 中的最后一个，呼叫管理器调用 NdisCmDisPatchIncomingDropParty。MCM 调用 NdisMCmDisPatchIncomingDropParty。如果删除的 Party 是 VC 中的最后一个，呼叫管理器调用 NdisCmDisPatchIncomingCloseCall，MCM 调用 NdisMCmDisPatchIncomingCloseCall(参见 1.6.8.2 节)。

对 Ndis(M)CmDisPatchIncomingDropPartY 的调用使得 NDIS 调用客户的 *ProtocolClIncomingDropParty* 函数。

图 1.35 显示了一个通过呼叫管理器删除 Party 的内入请求

图 1.35 通过呼叫管理器从多点呼叫中删除 Party 的内入请求

图 1.36 显示了一个通过 MCM 删除 Party 的内入请求

图 1.36 通过 MCM 从多点呼叫中删除 Party 的内入请求

ProtocolClIncomingDropParty 为从客户的多点 VC 中删除 Party 而执行所有协议确定的操作。如果被删去的 Party 不是 VC 中最后一个 Party，*ProtocolClIncomingDropParty* 必须调用 NdisClDropParty(参见 1.6.6.2 节)。如果被删除的 Party 是 VC 中的最后一个 Party, ProtocolClIncomingDropParty 必须调用 NdisClCloseCall(参见 1.6.8.1 节)。

1.6.7 发送并接收数据

在一个已创建并激活的 VC 上传输数据包括发送接收包。

1.6.7.1 在 VC 上发送包

为了在网络上发送包，面向连接客户或呼叫管理器调用 NdisCoSendPackets.与 MCM 相连的面向连接客户也调用 NdisCoSendPackets。因为呼叫管理器和 MCM 的接口在 MCM 的内部,所以 MCM 不调用 NdisCoSendPackets，MCM 不经 NDIS，而是直接将包传给 NIC。

图 1.37 显示了客户或呼叫管理器通过微端口发送包

图 1.37 客户或呼叫管理器通过微端口发送包

图 1.38 显示了客户通过 MCM 发送包

图 1.38 客户通过 MCM 发送包

在调用 NdisCoSendPackets 之前,客户或呼叫管理器应该按相应的发送顺序创建一个包指针数组。然后客户或呼叫管理器调用 NdisCoSendPackets，同时传递一个指向包指针数组的指针和指明所用 VC 的 *NdisVCHandle*。

当客户或呼叫管理器调用 NdisCoSendPackets 时，它就放弃了以下资源的所有权：
• 包数组中所包描述符
• 与这些描述符相链的缓冲区指示器所指的缓冲区
• 与包描述符相关的带外数据块(OOD)，包括 OOB 块中所有介质相关缓冲区。
对 NdisCoSendPackets 的调用，使 NDIS 调用微端口的 *MiniportCoSendPackets* 函数。*MiniportCoSendPackets* 根据数组中原有的包次序，顺序发送数组中的每一个数据包。*MiniportCoSendPackets* 调用 NdisQueryPacket 抽取信息，如缓冲区描述符的数目和要求传输的字节总数。
MiniportCoSendPackets 可以调用 *NdisGetFirstBufferFromPacket*,*NdisQueryBuffer* 或 *NdisQueryBufferOffset* 来抽取单个缓冲区的信息。*MiniportCoSendPackets* 通过使用 NDIS_GET_PACKET_XXX 宏，取回协议支持的每个包的 OOB 信息。*MiniportCoSendPackets* 函数通常忽略 NDIS_PACKET_OOB_DATA 块的 status 成员，但它可以把这个成员设置为传给 NdisMCoSendComplete 的状态。
当 MiniportCoSendPackets 没有足够的资源传送包时，面向连接微端口自己管理内部包排队，而不靠 NDIS 排队和重新提交发送包。微端口总是在它的内部队列中保存发送包，一直到包被发送为止。队列保存着协议确定的包描述符次序，这些包描述符将被传给 MiniportCoSendPackets 函数。
面向连接微端口必须用 NdisMCoSendComplete 来完成每个发送包的发送。它不可调用 NdisMSendResourcesAvailable.
面向连接微端口不能用协议分配的包描述符向 NdisMCoSendComplete 传送 STATUS_INSUFFICIENT_RESOVRCES,这些包描述符最初被传给 *MiniportCoSendPackets* 函数。
对 NdisMCoSendPackets 的调用,使 NDIS 调用客户的 *ProtocolCoSendComplete* 函数。*ProtocolCoSendComplete* 为完成传输操作执行所有善后处理，如通知客户。
发送操作的完成表示 NIC 驱动程序已经完成包的传输。然而，“智能”NIC 驱动程序在把包下放到 NIC 时就认为发送已完成。
在调用 *ProtocolCoSendComplete* 后，驱动程序重新获得了协议分配的资源的所有权：
• 包描述符
• 与包描述符相链的缓冲区器，这些包描述符映射容纳包数据的缓冲区，所有协议分配的缓冲区都由这些描述符映射。
• 所有与包描述符相关的 OOB
• 所有由 OOB 描述的协议分配缓冲区
虽然在对 NdisCoSendPackets 的调用中，NDIS 总按协议确定的顺序向微端口提交协议支持的包数组，但是下面的驱动程序可以随机地发送。那就是说每个绑定协议依靠 NDIS，以 FIFO 次序低层驱动程序提交包，但协议不能指望这个驱动程序以同样的次序调用 NdisMCoSendComplete。

1.6.7.2 接收 VC 上的包

面向连接微端口或 MCM 调用 NdisMCoIndicateReceivePacket,向面向连接客户或呼叫管理器指示接收到的包。如果微端口或 MCM 处理中断，它在 *MiniportHandleInterrupt* 函数中调用 NdisMCoIndicateReceivePacket。
图 1.39 显示了一个指示接收包的微端口

图 1.39 通过微端口接收包

图 1.40 显示了一个指示接收包的 MCM

图 1.40 通过 MCM 接收包

在对 NdisMCoIndicateReceivePacket 的调用中，微端口或 MCM 将传递一个指向包描述符指针数组的指针。微端口或 MCM 也要传送指示收到包的 VC 的 *NdisVCHandle*。在调用 NdisMCoIndicateReceivePacket 之前，微端口或 MCM 必须创建一个包数组(参见第二部分，4.6 节)。

对 NdisMCoIndicateReceivePacket 的调用，使 NDIS 调用协议驱动程序的 *ProtocolCoReceiveComplete* 函数，这个驱动程序和微端口共享 VC。*ProtocolCoReceiveComplete* 函数处理接收指示。

在调用 NdisMCoIndicateReceivePacket 之后，微端口必须调用 NdisMCoReceiveComplete 来指示接收的完成，这些接收是由 NdisMCoIndicateReceivePacket 指示的。对 NdisMCoReceiveComplete 的调用，使 NDIS 调用面向连接客户或呼叫管理器的 *ProtocolReceiveComplete* 函数。

如果协议不立即返还微端口分配的资源，微端口或 MCM 可以用 NDIS_STATUS_RESOURCES 调用 NdisMCoIndicateStatus，来提醒协议：微端口或 MCM 缺少可利用的包或缓冲区描述符。

接收包的详细情况，参见第二部分，4.6 节。

1.6.8 断开呼叫

面向连接客户能发起一个呼叫的断开操作，这个呼叫或者是它发出的，或者是接受的内入呼叫。类似地，远程 party 也可以发起一个呼叫的断开。

1.6.8.1 客户发起的关闭呼叫请求

如果客户正在关闭含有一个或多个 party 的多点呼叫，它首先必须多次调用 NdisClDropParty，删除除最后一个 Party 外的所有 Party（参见 1.6.6.2）.

客户调用 NdisClCloseCall 发起呼叫的关闭。图 1.4.1 显示了客户通过呼叫管理器发起呼叫的关闭。图 1.42 显示了客户通过 MCM 发起呼叫关闭。

图 1.41 客户通过呼叫管理器发起呼叫关闭请求。

图 1.42 客户通过 MCM 发起呼叫关闭请求。

面向连接客户在如下情况中，调用 NdisClCloseCall

- 关闭一个已创建的外出或内入呼叫
- 从 *ProtocolClIncomingCloseCall* 函数中断开一个呼叫(参见 1.6.8.2 节)
- 如果远程 party 提出的 QoS 的改变不能被接受，则在 *ProtocolClIncomingCallQoSChange* 函数中断开呼叫。
- 如果客户提出的 QoS 改变，远程 party 不能接受，则在函数 *ProtocolClModifyCallQoSComplete* 中断开呼叫。

客户对 NdisClCloseCall 的调用使得 NDIS 调用呼叫管理器或 MCM 的 *ProtocolCmCloseCall* 函数。*ProtocolCmCloseCall* 必须与网络控制设备通信,以终止本地结点与远程结点之间的通信。

如果传送给 *ProtocolCmCloseCall* 的是一个显式的 *CallMgrPartyContext*,那么终止的呼叫是一个多点呼叫。呼叫管理器或 MCM 必须与网络硬件通信，来终止这个多点呼叫。

在调用 NdisClClose 过程中，NDIS 向 *ProtocolCmCloseCall* 传递一个缓冲区指针，该缓冲区中包含客户提供的数据。这些数据可以是指明为什么关闭呼叫的诊断数据，也可以是信令协议需要的其他数据。在完成关闭呼叫之前，*ProtocolCmCloseCall* 必须把这些数据送出去。如果不能在关闭连接的同时发送数据，呼叫管理器 MCM 返回 NDIS_STATUS_INVALID_DATA。

ProtocolCmCloseCall 可以同步或异步完成，呼叫管理器用 NdisCmCloseCallComplete 异步

完成,MCM 用 NdisMCmCloseCallComplete 异步完成。对 Ndis(M)CmCallComplete 的调用使 NDIS 调用客户的 *ProtocolClCloseComplete* 函数。
通过调用 NdisCmDeactivateVC 或 NdisMCmDeacticateVC，呼叫管理器或 MCM 使 VC 去活(参见 1.6.3.3 节)。然后,VC 的创建者(客户呼叫管理器或 MCM)可以有选择地删除 VC(参见 1.6.3.4 节)。

1.6.8.2 关闭呼叫的内入请求

当远程客户调用 NdisClCloseCall 关闭一个呼叫时,本地呼叫管理器或 MCM 必须向本地客户指示这个请求。为此，呼叫管理器用被设置为 NDIS_STATUS_SUCCESS 的 CloseStatus 来调用 NdisCmDispatchIncomingCloseCall.(参见图 1.43)。

图 1.43 通过呼叫管理器关闭呼叫的内入请求

MCM 调用 NdisMCmDispatchIncomingCloseCall，来指示关闭呼叫的请求(参见图 1.44)。

图 1.44 通过 MCM 关闭呼叫的内入请求

呼叫管理器或 MCM 也能在如下情况下，调用 Ndis(M)CmDispatchIncomingCloseCall：
•如果 MCM 确定面向连接客户正在请求改变呼叫参数，而这个改变是不可接受的，那么可以从 *ProtocolCmIncomingCallComplete* 中调用。
• 如果异常网络条件迫使呼叫管理器断开活动呼叫。对 Ndis(M)CmDispatchIncomingCloseCall 的调用,使NDIS 调用 *ProtocolClIncomingCloseCall* 函数。*ProtocolClIncomingCloseCall* 执行所有协议确定的操作，诸如通知客户连接已经断开。如果关闭的呼叫是由客户创建的多点呼叫，*ProtocolCmIncomingCloseCall* 必须调用 NdisClCloseCall(用指向 VC 中最后一个 Party 的包柄)，来确认客户不再在这个 VC 上发送或接收数据。如果是呼叫管理器或 MCM 创建了这个 VC，*ProtocolCmIncomingCloseCall* 在调用 NdisClCloseCall 之后，返回控制。呼叫管理器或 MCM 必须使 VC 去活(参见 1.6.3.3 节)。如果是客户创建的 VC，并且 CloseStatus 是 NDIS_STATUS_SUCCESS，那么 *ProtocolClIncomingCloseCall* 可以有选择地用 NdisCoDeleteVC 断开 VC(参见 1.6.3.4 节)。或者将 VC 重用于另一呼叫。如果 CloseStatus 不是 MDIS_STATUS_SUCCESS，*ProtocolCmIncomingCloseCall* 必须调用 NdisCoDeleteVC.
如果是呼叫管理器或 MCM 创建的 VC，呼叫管理器或 MCM 分别调用各自的 NdisCmDeleteVC 或 NdisMCmDeleteVC,有选择地删除 VC(参见 1.6.3.4 节)。

1.6.9 获取并设置信息

呼叫管理器,面向连接客户或 MCM 可以查询或设置绑定上其他驱动程序所维护的信息。另外，微端口或 MCM 可以向其他驱动程序指示微端口状态。

1.6.9.1 查询或设置信息

面向连接客户或呼叫管理器调用 NdisCoReqeuest 来查询或设置信息，这些信息由绑定上的其他驱动程序维护或由低层的微端口维护。
在调用 NdisCoReqeuest 之前，客户或呼叫管理器为其分配了一个缓冲区并初始化了一个 NDIS_REQUEST 结构。这个结构说明了请求类型(查询或设置)，指明了查询或设置的信息(OID)，以及用于传送 OID 数据的缓冲区指针。
如果面向连接客户或呼叫管理器传递了一个有效的 *NdisAfHandle*(参见 1.2.1 节),那么 NDIS 将调用每个协议驱动程序的 *ProtocolCoRequest* 函数(参见图 1.45)。

图 1.45 查询或设置客户或呼叫管理器信息

如果低层微端口是 MCM，那么 NDIS 将调用 MCM 的 *ProtocolCoReqcuest*(参见图 1.46)

图 1.46 查询或设置微端口的呼叫管理器信息

如果面向连接客户或呼叫管理器传递空的地址族句柄，NDIS 调用微端口或 MCM 的 *MiniportCoReqcuest*。图 1.47 显示了客户或呼叫管理器查询或设置微端口信息。

图 1.47 查询或设置微端口信息。

图 1.48 显示了客户查询或设置 MCM 的微端口信息。

图 1.48 查询或设置 MCM 的微端口信息。

为查询客户维护的信息，MCM 调用 **NdisMCmReqeuest**(参见图 1.49)。

图 1.49MCM 发起的客户信息查询

通过说明 VC 句柄或 Party 句柄，**NdisCoReqeuest** 或 **NdisMCmReqeuest** 的调用者可以缩小请求范围。无论请求本身是给客户、呼叫管理器、微端口还是 MCM 的，传递一个空的 *NdisVcHandle*，都将使请求成为成为全局的。*ProtocolCoReqcuest* 或 *MiniportCoReqcuest* 能同步完成，或者也可用 **NdisCoReqcuestComplete** 异步完成。对 **NdisCoReqcuestComplete** 地调用使 NDIS 调用驱动程序的 *ProtocolCoRequestComplete* 函数，正是这个驱动程序调用了 **NdisCoReqeuest**。

1.6.9.2 指示微端口状态

为了报告面向连接 NIC 的状态变化或是 NIC 上某个活动 VC 的状态变化，面向连接微端口调用 **NdisMCoIndicateStatus**.如果微端口报告某个 VC 的状态变化，它须提供指明此 VC 的 *NdisVCHandle*.图 1.50 显示了指示一个微端口状态.图 1.51 显示了指示一个 MCM 状态

图 1.50 指示微端口状态

图 1.51 指示 MCM 的微端口状态

如果没有提供 *NdisVCHandle*,对 **NdisMcoIndicateStatus** 的调用，使 NDIS 调用了所有绑定到该 NIC 上的协议驱动程序的 *ProtocolCoStatus* 函数。如果微端口提供 *NdisVCHandle*,NDIS 调用共享该 VC 的协议驱动程序的函数。当 MCM 调用 **NdisMCoIndicateStatus** 时，由于这里没有呼叫管理器,NDIS 仅调用绑定的面向连接客户的 *ProtocolCoStatus* 函数。

ProtocolCoStatus 函数能记录状态变化并采取正确的行动。例如，在微端口调用 **NdisMCoIndicateSatus** 报告 NIC 中操作问题时,协议可以调用 **NdisReset**(或 NDIS 自己调用微端口的 *MiniportReset* 函数)，使 NIC 恢复到工作状态(参见 1.6.10 节)。

1.6.10 重置

绑定到面向连接微端口或 MCM 的协议，能调用 **NdisReset**，从而使微端口或 MCM 重置 NIC。NDIS 自己也可以调用微端口或 MCM 的 *MiniportReset* 重置 NIC。

图 1.5.2 显示了一个客户向微端口提出重置请求的过程。

图 1.5.2 通过微端口提出重置 NIC 请求。

图 1.5.3 显示了一个客户向 MCM 提出重置请求的过程。

图 1.5.3 通过 MCM 提出重置 NIC 请求。

当面向连接驱动程序重置 NIC 时，NDIS 用 NDIS_STATUS_RESET_START 调用协议的 *ProtocolCoStatus* 函数，通知每个绑定的协议。NDIS 接着调用 *ProtocolStatusComplete*.在微端口或 MCM 的 NIC 重置时，NDIS 不接受协议的发送请求和对微端口或 MCM 的信息请求。当重置进行时，协议驱动程序无法用 **NdisCoSendPackets** 发送给微端口，也不能用

NdisCoRequest 向微端口请求信息。

MiniportReset 执行所有与设备相关的操作，这些操作对重置 NIC 是必须的。*MiniportReset* 可以同步完成，也能用 NdisMResetComplete 异步完成。

· 如果同步完成重置，NDIS 用 NDIS_STATUS_RESET_END 调用每个绑定协议的 *ProtocolCoStatus* 函数。然后,NDIS 调用协议的 *ProtocolStatusComplete* 函数。

· 如果异步完成重置，NDIS 用 NDIS_STATUS_RESET_END 调用每个绑定协议的 *ProtocolCoStatus* 函数，然后 NDIS 调用协议的 *ProtocolStatusComplete* 函数。如果重置由协议发起，NDIS 也调用该协议的 *ProtocolResetComplete* 函数。

第五部分 安装网络组件

第一章安装网络组件
第二章网络组件的通知对象

第一章　安装网络组件

这章包括：
• 涉及网络组件安装的组件和文件总结
• 关于创建网络组件信息(INF)文件的细节化信息

1.1 用于安装网络组件的组件和文件

Windows 2000 网络组件的安装涉及如下几个方面：
• 类安装器和协作安装程序
网络组件由 Windows 2000 网络类安装器，或者供应商创建的定制类安装器实现安装。类安装器是一个动态链接库，用于安装、配置或删除某个类的设备。如果网络类安装器没有提供所需的特性，供应商自己可以写一个协作安装程序来定制安装过程。协作安装程序可以是一个 Win32 DLL,它在 Windows 2000 系统中协助设备安装。协作安装程序作为类安装器的助手或过滤器，由设备安装器调用。
• 信息(INF)文件
每个网络组件必须有信息(INF)文件，网络类安装器可以用来安装组件。网络 INF 文件基于通用的 INF 文件格式。
为网络组件创建 INF 文件的细节化信息，参见 1.2 节
• 可选的通知对象
软件组件，如网络协议、客户或服务，可以有一个通知对象。通知对象实现一个用户接口，向组件通知绑定事件，使组件实现对某个绑定过程的控制，并提供条件安装和条件删除。通知对象在第二章中描述。
网络适配器也能支持一个用户接口，并对绑定事件、条件安装和条件删除实现一些控制。这些通过 INF 文件或协作安装程序来实现。
• 可选的网络移植 DLL 和相关文件
如果网络供应商的驱动程序没有随 Windows 2000 一起发布，供应商应该提供这些组件的升级支持。网络升级处理将网络组件的参数只从 Windows NT 3.51 或 Windows NT 4.0 移到 Windows 2000 上。关于升级网络组件的更多信息，参见在线 DDK *“NetWork Drivers Reference”*的第六部分，第一章。
除了以上组件外，供应商也提供如下文件：
• 设备的驱动程序
驱动程序通常由驱动程序印象(.sys 文件)和驱动程序库(.dll 文件)构成。
• 可选的驱动程序目录文件
供应商向 Windows 硬件质量库(WHQL)提交驱动程序，用来测试和签名，进而得到一个数字签名。WHQL 将一个目录文件(.cat 文件)随同包返回。供应商必须在设备的 INF 文件中列出所有目录文件。

• 可选的文本模式安装信息文件(txtsetup.oem).
如果网络设备要求启动机器，操作系统包中必须包括驱动程序，或者该设备供应商必须提供一个 txtsetup.oem 文件。txtsetup.oem 文件包含一些信息，在启动处理的早期阶段，系统安装组件用这些信息来安装设备(在文本模式安装期间)。

1.2 创建网络 INF 文件

网络 INF 文件基于标准 INF 文件格式，但也包括网络特殊项，如网络特殊节、说明、节项和值。下面对于网络 INF 文件的描述，假设读者已经理解基本 INF 文件。在尝试创建网络 INF 文件之前，应先读一下基本 INF 文件的描述。
微软 Windows 2000 的网络 INF 文件与 Windows NT 4.0 和更早版本的 NT 的网络 INF 文件是不兼容的。为了使网络组件可以在 Windows 2000 和 NT 4.0 平台上安装，应分别创建不同的 INF 文件。
同样的 INF 文件可以在 Windows 2000 和 Windows 95/98 上用来安装网络组件。参见*《Windows 2000 Driver Development Reference》*第一卷的 INF 文献。
INF 文件需求因网络类型不同而变化。不同类型网络对 INF 需求的总结，参见 1.2.11。

1.2.1 网络 INFS 文件名的约定

和 Windows 2000 一同出售的 INF 文件的文件名不得超过 8 个字符。其他 INF 文件不受这个限制。所有 INF 文件的扩展名为.inf。所有网络 INF 文件名以 net 开头，用来指示由网络类安装器处理这个文件。IrDA 设备的 INF 文件以 ir 开头。
文件名的其余部分用来指示网络组件生产商或描述这些组件。下面是一个有效的网络 INF 文件名的例子：

网络 INF 文件名	文件名组件
netMst	net+制造商
netDlc	net+产品描述
net999	net+产品型号

1.2.2 网络 INF 文件的版本节

网络 INF 文件的版本节有一些网络特有性质的描述，如下：
Class
类版本节中应该包括类项，使读者容易确定该文件要安装的网络组件的类别。
有 4 个网络类
• Net
说明物理或虚拟的网络适配器。NDIS 中间层驱动程序包含在 Net 类中，该程序输出虚拟网络适配器。
• NetTrans
说明网络协议，如 TCP/IP，IPX，面向连接客户或面向连接呼叫管理器。
• NetClient
说明网络客户，如微软网络客户或 NetWare 客户。NetClient 组件被认为是网络提供者，如果它提供网络打印服务，它也被认为是一个打印提供者。
• NetService
说明网络服务，如文件服务或打印服务。
虽然 IrDA 设备由类安装器安装，但 IrDA 并不归于上面 4 个网络类中的任何一个。用来安装

IrDA 设备的 INF 文件应该有一个 Infrared 的 Class 值。这个类包括 Serial-IR 和 Fast-IR 设备。

ClassGuid

版本节中必须包含 ClassGuid 项。网络类安装器用 ClassGuid 项确定安装的网络组件类。有 4 个 ClassGuid 值，每个值对应于一个网络类：

网络类	ClassGuid
Net	{4D36E972-E325-11CE-BEC1-08002BE10318}
NetTrans	{4D36E973-E325-11CE-BFC1-08002BE10318}
NetClient	{4D36E974-E325-11CE-BFC1-08002BE10318}
NetService	{4D36E975-E325-11CE-BFC1-08002BE10318}

IrDA 设备的 INF 文件应该有一个 **ClassGuid**:{66ddlfc5-81do-bec7-08002be2092f}。

签名和操作系统项

签名项有三个值：

- $Windows 95$
- $Windows NT$
- $Chicago$

如果 INF 文件仅用于 Windows 95/98，正确的签名应力$Windows 95$。有$Windows 95$签名的 INF 文件不能运行于 Windows 2000.

如 INF 仅用于 Windows 2000，正确的签名是$Windows NT$。有$Windows NT$签名的 INF 文件不能在 Windows 95/98 上运行。

如 INF 在 Windows 95/98 以及 Windows 2000 上运行，正确的签名是$Chicago$。有$Chicago$签名的网络 INF 文件必须在版本节中有如下一行：

Compatible=1

如果没有这一行，就不能在 Windows2000 上运行。

版本节例子

下面是安装网络适配器的 INI 文件的版本节：

```
[Version]
Signature=$Chicago$
Compatible=1
Class=Net
ClassGuid={4D36E972-E325-11CE-BEC1-08002BE0318}
Provider=%Msft%
Driverver=08/20/1999
```

Provider 项指示谁开发了 INF 文件,不是谁开发了 INF 文件安装的组件。

1.2.3 网络 INF 文件的模型节

INF 文件的模型节为每个安装的组件,包含如下格式的项：

[device-description=install-section.name,hw-id[,compatible-id...]

对这个项的细节化描述，参见*《Windows 2000 Driver Development Reference》*第 1 卷的 INF 文档。

网络适配器的 hw-id(也称为设备、硬件、或组件 ID)必须和适配器提供给 PnP 管理器的硬件 ID 相匹配。网络软件组件的 hw-id 应由提供者名字组成，后面跟着下划线和制造商名称或

产品名称。提供者名称指示了 INF 文件的提供者。制造商的名字指示了软件组件的制造商。产品名字指明了软件组件。例如，下面的 hw-id 包含了后跟产品名称的提供者名称：

MS_DLC

MS_IBMDLC

1.2.4 INF 文件的 DDInstall 节

网络 INF 文件的 DDInstall 节有如下的网络特殊属性：

Characteristics

每个 *DDInstall* 节必须有 **Characteristics** 项。**Characteristics** 项说明了网络组件的某个特征，可能限制用户对组件所作的操作。例如，**Characteristics** 可以说明组件是否支持用户接口，是否可删除，或是否对子用户隐藏。

Characteristics 项可以有 1 个或多个如下的值(多值应计算总和)：

十六进制值	名字	描述
ox1	NCF_VIRTUAL	说明组件是个虚拟适配器
ox2	NCF_SOFTWARE_ENUMERATED	说明组件是一个软件模拟的适配器
ox4	NCF_PHYSICAL	说明组件是一个物理适配器
ox8	NCF_HIDDEN	说明组件不显示用户接口
ox10	NCF_NO_SERVICE	说明组件没有相关的服务(设备驱动程序)
ox20	NCF_NOT_USER_REMOVABLE	说明不能被用户删除(例如，通过控制面板或设备管理器)
ox40	NCF_MULTIPORT_INSTANCED_ADAPTER	说明组件有多个端口,每个端口作为单独的设备安装。每个端口有自己的 hw_id(组件 ID)并可被单独安装，这只适合于 EISA 适配器
ox80	NCF_HAS_UI	说明组件支持用户接口(例如，Advanced Page 或 Customer Properties Sheet)
ox400	NCF_FILTER	说明组件是一个过滤器

下面 **Characteristics** 值的组合是不被允许的：

• NCF_VIRTUAL,NCF_SOFTWARE_ENUMERATED,和 NCF_PHYSICAL 是相互排斥的。

•NCF_NO_SERVICE 不能和 NCF_VIRTUAL,NCF_SOFTWARE_ENUMERATED,或 NCF_PHYSICAL 同时使用。虚拟的、软件仿真的适配器或物理适配器必须有一个相关的服务(设备驱动程序)。

下面是支持用户接口的物理适配器的 **Characteristics** 项例子：

Characteristics =0x84;NCF_PHYSICAL,NCF_HAS_UI

BusType

物理适配器的 *DDInstall* 节必须包含 **BusType** 项，该项说明了总线类型(PCI，ISA，等等)。

BusType 项可能的值由 INTERFACE_TYPE 枚举说明：

总线类型	值

ISA	1
EISA	2
MicroChannel	3
TurboChannel	4
PCIBus	5
VMBus	6
NuBus	7
PCMCIABus	8
Cbus	9
MPIBus	10
MPSABus	11
PNPISABus	14
PNPBus	15

如果适配器可用于多种总线类型,那么安装该适配器的 INF 文件对每种总线类型都应有一个 *DDInstall* 节。例如，如果适配器可用于 ISA 总线和 PnPISA 总线，INI 文件应该有 ISA 的 *DDInstall* 节和 PnPISA 的 *DDInstall* 节。在 *DDInstall* 节中的 **BusType** 说明该节的总线类型。例子如下：

```
[a1.isa]
BusType=1
[a1.pnpisa]
BusType=14
```

EisaCompressedId 和 AdapterMask

安装 EISA 网络适配器的 INF 文件的 *DDInstall* 节必须包括 **EisaCompressedId** 项,该项说明 EISA 压缩 ID 和适配器掩码。例子如下：

```
EisaCompressedId=0x24322432
AdapterMask=0xffff
```

Port1DeviceNumber 和 Port1FunctionNumber

安装多端口网络适配器的 INF 文件的 *DDInstall* 节，包括 **Port1DeviceNumber** 项或 **Port1FunctionNumber** 项。说明了这个项，当鼠标在网络适配器名字或图标上时，适配器端口信息显示于连接属性对话框（这个通过网络和拨号文件夹来访问）中。如果适配器端口号映射到 PCI 设备号，则要使用 **Port1DeviceNumber** 项。将 **Port1DeviceNumber** 设置为第一个 PCI 设备号。例如，如果 PCI 设备号 4 映射到端口 1，PCI 设备号 5 映射到端口 2，PCI 设备号 6 映射到端口 3。用如下项：

```
Port1DeviceNumber=4
```

如果适配器端口号顺序映射到 PCI 函数号，则要使用 **Port1FunctionNumber** 项。将 **Port1FunctionNumber** 设置成第一个 PCI 函数号。例如，如 PCI 函数号 2 映射到端口 1，PCI 函数号 3 映射到 port2,PCI 函数号 4 映射于端口 3，等等。用如下项：

```
Port1FunctionNumber=2
```

PCI 设备号或 PCI 函数号到端口的映射被认为是静态的。同时也认为适配器端口是顺序编码的。

Port1DeviceNumber 和 **Port1FunctionNumber** 项是互相排斥的。如果两个项存在于同一 *DDInstall* 节中，则只使用 **Port1DeviceNumber** 项。

1.2.5 删除节

删除 NetClient,NetTrans,和 NetService 组件支持的节，但不删除 Net 组件(适配器)。网络类安装器不对适配器实例进行跟踪。删除节导致删除其他网络适配器和适配器的其他实例共享的文件，使适配器或适配器实例无法工作。

如果必须删除一个 Net 组件所使用的驱动程序文件,则要用协作安装程序跟踪所有使用该文件的驱动程序。这种协作安装程序既可跟踪同一设备的多个实例，也可跟踪多个设备的驱动程序。协作安装程序的更多信息，参见*《Windows 2000 Driver Development Reference》*的第一卷的 INF 文献。

1.2.6 ControlFlags 节

ControlFlags 节通常有 1 个或多个 **ExcludeFromSelect** 项。每个 **ExcludeFromSelect** 项说明一个网络组件，该组件不作为手动安装的一个选项显示给最终用户。*ControlFlags* 节必须为如下项包括一个 **ExcludeFromSelect** 项：

- 每个安装的即插即用适配器
- 每个由程序自动增加（而非用户手动增加）的软件组件

非即插即用的适配器必须由用户手动增加，因此不应在 *ControlFlags* 节中列出。例如，ISA 和 EISA 适配器必须由用户手动安装。

ExcludeFromSelect 项的功能和 *DDInstall* 节中的 **Characteristics** 项的 NCF_HIDDEN 的功能是不同的。

ExcludeFromSelect 项防止适配器或软件组件在选择安装组件对话框中显示。然而，适配器或组件仍能在连接对话框中列出。NCF_HIDDEN 防止适配器或组件在任何用户界面中出现，包括连接对话框。更多的信息，参见*《Windows 2000 Driver Development Reference》*的第一卷。

1.2.7 网络 INF 文件的 add-registry-sections

INF 对每个安装的组件都包含 1 个或多个 *add-registry-sections*。*Add-registry-section* 向注册表增加键和值。INF 的 *DDInstall* 节包含 **AddReg** 说明，它引用 1 个或多个 *add-registry-sections*。关于 *add-registry-section* 和 **AddReg** 说明,参见*《Windows 2000 Driver Development Reference》*卷 1 中的 INF 文档。

向组件的实例键中增加键和值

一个或多个 *add-registry-sections* 能向实例键中增加键和值：

- 为组件设置静态参数(不能通过用户接口修改的配置参数)。参见 1.2.7.1 节
- 说明端口数目(如信道，电路或 bearer 信道)参见 1.2.7.2 节
- 说明 ISDN 适配器的键和值。参见 1.2.7.3 节
- 请求另一个网络组件的安装。参见 1.2.7.4 节
- 说明支持定制属性页的值。参见 1.2.7.12

向 NetClient 组件增加键和值

一个 NetClient 组件的 *add-registry-section* 必须将一个 **NetworkProrider** 键加入到该组件的 *service* 键中。

NetworkProvider 键有 2 个值：说明网络提供者名字的 **Name** 和描述网络提供者 DLL 完全路径的 **ProviderPath**。参见 1.2.7.6。

生成 Ndi 键

每个网络 INF 文件必须包括至少一个 *add-registry-section*,用来为该文件安装的组件增加

Ndi 键。**Ndi** 键是一个特殊网络键，在组件的实例键内。加入到 **Ndi** 键中的键和值根据网络组件类型和相容性而不同。**Ndi** 键支持如下信息：

- 为 NetTrans,NetClient,或 NetService 组件说明 HelpText 值。参见 1.2.7.7。
- 为通知对象说明值。参见 1.2.7.8
- 说明相关服务值。参见 1.2.7.9
- 说明绑定接口。参见 1.2.7.10
- 为高级页说明适配器配置参数。参见 1.2.7.11
- 为过滤器服务说明值。参见 1.2.7.13。
- 说明束成员关系。参见 1.2.7.14

在 Windows 95/98 中可用，在 Window 2000 中不再使用的 Ndi 注册键和值的列表参见 1.2.7.15。

1.2.7.1 设置静态参数

静态参数只能用 INF 文件设置一次，不能通过属性页重新配置。

add-registry-section 将一个 REG_S2 值，作为静态参数加入到组件的实例键中。下面是一个例子，将两个静态参数加入到组件实例键中。

```
[al.staticparams.reg]
HKR,,MediaType,0,"1"
HKR,,InternalId,0,"232"
```

add-registry-section 可以将一些供应商定义的静态参数加入到组件实例键中。

1.2.7.2 为 WAN 适配器说明 WAN 端点

WAN 适配器的 INF 文件必须向适配器实例键中增加 **WanEndpoionts** 值。**WanEndpoints** 是一个 REG_DWORD 值，说明了 WAN 适配器支持的端点数目(如信道，电路或 bearer channels)。例如，BRI(基本速率接口)ISDN 适配器的 **WanEndPoints** 值是 2，而 PRI(主速率)ISDN 适配器的 **WanEndPoints** 值是 23。

下面是 *add-registry-section* 的例子，为一个 BRIISDN 适配器增加 **WanEnelpoionts**,其值为 2。

```
[al.reg]
HKR,,WanEndpoints,0x00010001,2
```

1.2.7.3 为 ISDN 适配器说明 ISDN 键和值

除了 **WanEndpoints** 值(参见 1.2.7.2 节)，ISDN 适配器的 INF 文件还必须向适配器实例键中加入如下键和值(通过 *add-registry-section*)。

IsdnNumDChannels

这个 REG_DWORD 类型的值说明了 ISDN 适配器支持的 D-Channels 的数目。

IsdnAutoSwitchDetect(Optional)

这个可选的 REG_DWORD 类型值说明,是否 ISDN 适配器支持自动交换检测。值 1 表示支持，值 0 不支持。

IsdnSwitchType

这个 REG_DWORD 值说明了 ISDN 适配器支持的交换类型：

ISDN_SWITCH_AUTO	Auto Detect(NorthAmericalonly)
ISDN_SWITCH_AUTO	ESS5(AT&T,NorthAmerica)
ISDN_SWITCH_NI1	National ISDN1(NI_1)

ISDN_SWITCH_NI2	National ISDN2(NI_2)
ISDN_SWITCH_NT1	Northern Telecom DMS 100(NT_1)
ISDN_SWITCH_INS64	NTT INS64(Japan)
ISDN_SWITCH_ITR6	Germal National(ITR6).此类型很少使用
ISDN_SWITCH_VN3	French National(VN3).此类型很少使用
ISDN_SWITCH_NET3	European ISDN(DSS1)
ISDN_SWITCH_DSS1	European ISDN(DSS1)
ISDN_SWITCH_AUS	Australian National 此类型很少使用
ISDN_SWITCH_BEL	Belgium National 此类型很少使用
ISDN_SWITCH_VN4	French National(VN4)
ISDN_SWITCH_SWE	Swedish National
ISDN_SWITCH_ITA	Italian National
ISDN_SWITCH_TWN	TaiWan National

为说明多个交换类型，只要将交换类型值相加即可。ISDN 向导（在安装 ISDN 组件时自动运行）允许用户选择由 **IsdnSwithTypes** 说明的交换类型。选择的交换类型决定了随后显示的 ISDN 配置参数。这些参数包括电话号码，SPID(service profile identifier),子地址和多用户号。

IsdnNumBchannels(增加到 D-Chanel 键)

D-Channel 值是一个 0～9 的索引，用于指明 D-Channel。

IsdaNumBChannels 是一个 REG_DWORD 值，该值被添加到 *D-channel* 键中。**IsdnNumBchannels** 说明了由 D-Channel 支持的 B-Channel 的数目。

下面是向 ISDN 适配器实例键增加 ISDN 键和值的例子。说明了两个 D-Channels，每个 D-Channel 中说明了两个 B-Channel。

```
[ISDNadapter,reg]
HKR,,WanEndpoint,0x00010001,4
HKR,,IsdnNumDChannels,0x00010001,2
HKR,,IsdnAutoSwitchDetect,0x00010001,1
HKR,,IsdnSwitchType,0x00010001,0x00000004;NI1
HKR,0,IsdnNumBChannels,0x00010001,2
HKR,1,IsdnNumBChannels,0x00010001,2
```

ISDN 向导自身也根据用户指定的参数值向 ISDN 适配器的实例键中增加 ISDN 键和值。ISDN 向导增加如下的键和值：

- **IsdnSwitchType**

这个 REG_DWORD 值指示用户选择的交换类型

- 对每个 D-Channel,有一个 **IsdnMultiSubscriberNumbers** 值.这个 REG_MULTI_S2 值指明了用户说明的 multi_subscriber 数目。
- 每个 B-Channel 有一个 *B-Channel* 键、**IsdnSpid**、**IsdnPhoneNumber** 和/或一个 **IsdnSubaddress** 值。
- *B-Channel* 键是指示 B-Channel 的由 0 开始的索引。B-Channel 键值的最大值比 **IsdnNumBChannels** 的值小 1。
- **IsdnSpid** 是指示 SPID 的一个 REG_SZ 值。如果有，就由用户来说明。
- **IsdnPhoneNumber** 是电话号码.如果有，由用户说明

下面是 ISDN 适配器注册表节的布局示例。每个注册键由方括号括起，例如：[keyname]。粗体 ISDN 键和值由 ISDN 适配器的 INF 文件添加。非粗体的 ISDN 键和值由 ISDN 向导添加。

[Enum*emumeratorID*|*device-instance-id*] ;ISDN 适配器实例键
WanEndPoints=4
IsdnNumDChannels=2
IsdnAutoSwitchDetect=1
IsdnSwitchType=0x4 ;NationalISDN1
[Enum*emumeratorID*|*dervice-instance-id*\0] ;D-Channel0
IsdnNumBChannels=2
IsdnMultiSubscriberNumbers=123456723456783456789
[Enum*emumeratorID*|*device-instance-id*\0\0] ;D-channel0 的 B-Channel0
IsdnSpid=00555121200
IsdnPhoneNumber=5551212
IsdnSubaddress=
[Enum*emumeratorID*|*device-instance-id*\0\1] ;D-Channel0 的 B-Channel1
IsdnSpid=00555121300
IsdnPhoneNumber=5551213
IsdnSubaddress=
[Enum*emumeratorID*|*dervice-instance-id*\1] ;D-Channel1 键
IsdnNumBChannels=2
IsdnMultiSubscriberNumbers=867530923901257658156
[Enum*emumeratorId*|*dervice-instance-id*\1\0] ;D-Channel1 的
B-Channel0
IsduSpid=00555987600
IsdnPhoneNumber=5559876
IsdnSubaddress=
[Enum*emumeratorID*|*dervice-instance-id*\1\0] ;D-Channel1 的 B-Channel1
Isdnspid=00555876500
IsdnPhoneNumber=5558765
IsdnSubaddress=

1.2.7.4 安装多协议 WAN NICs

多协议 WAN NIC 提供多于 1 种的 WAN 协议。例如，NIC 可能允许用户选择 ISDN，帧中继或 T1 信道。在安装 NIC 或配置 NIC 期间，用户可以选择 WAN 协议。
多协议 WAN NIC 的供应商必须提供协作安装程序，以安装向导页。(协作安装程序详细信息，参见*《Plug and Play, Power Management, and Setup Design Guide》*，以及*《Windows 2000 Driver Development Reference》*卷 1)。向导页提示用户选择 WAN 协议。
•如果用户选择 ISDN，则显示 ISDN 向导。ISDN 向导提示用户输入 ISDN 交换类型和其他 ISDN 参数值。(参见 1.2.7.3 节)
• 如果用户选择 WAN 协议，向导在 WAN NIC 的实例键中增加 ShowIsdnPages 值。在这种情况中，向导将 ShowIsdnPages 设置成 0，从而阻止 ISDN 向导的显示。
在安装 WAN NIC 后，用户可以使用 NIC 的属性页重新配置 NIC。
• 如果用户将协议从 ISDN 改为 WAN 协议，属性页将 ShowIsdnPage 值加入 WAN NIC 的实例键中。属性页设置 ShowIsdnPages 为 0，阻止 ISDN 向导的显示。
• 如果用户将协议改为 ISDN，WAN NIC 的属性页显示一个对话框提示用户确认改变。当用户确认改变后，属性页将 ShowIsdnPages 设置成 1。当用户再次打开属性页时，显示 ISDN 向

导。如果多协议 WAN NIC 支持 ISDN，绑定接口中的 LowerRange 必须设置成 isdn(参见 1.2.7.10 节)。如果 showIsdnPage 注册值不存在，且 NIC 的 LowerRanger 被设置为 isdn，安装和配置时显示 ISDN 向导。如果 showIsdnPages 设置成 0，ISDN 向导不显示。ShowIsdnPage 设置成 1，ISDN 向导在 NIC 配置时显示。

1.2.7.5 请求安装另一个网络组件

为正常运作，网络组件可能需要安装 1 个或多个其他网络组件。网络 INF 文件用 RequiredAll 值说明这些依赖性。RequiredAll 值被加入到需要安装其他组件的组件的 Ndis 键中。(通过 *add-registry-section*)。

下面的例子显示了 *add-registry-section* 的 RequiredAll 项：

```
[ndi.reg]
HKR,Ndi,RequiredAll,0,"Component id"
```

组件 ID 是所需要的网络组件的 hw-id(参见 1.2.3 节)。如果网络组件要安装多个其他组件，对每个组件都使用一个 RequiredAll 项，如下所示：

```
[HKR,Ndi,RequireAll],0,"component2 id"
```

RequiredAll 仅用于安装那些不能由用户安装的隐藏网络组件。这种组件不为用户接口所支持。

由 RequiredAll 说明的组件只有在通过 RequiredAll 请求安装该组件的组件被删除后，才能被删除。例如，如果组件 A 的 INF 文件使用 RequiredAll 说明了对组件 B 的依赖，组件 B 在组件 A 删除时才能删除。

RequiredAll 仅安装在组件运行时，必须需要的其他组件。例如，Net 组件(适配器)的 INF 文件中，使用 RequiredAll 说明安装 TCP/IP，用户在适配器删除前不能删除 TCP/IP。由于适配器可以不需要 TCP/IP，适配器的 INF 文件不应该使用 RequiredAll 说明对 TCP/IP 的依赖。

说明 RequiredAll 依赖性的 INF 文件，必须确保所需网络组件的 INI 文件在 inf 目录下。它通常有一个 *CopyFiles* 节。*CopyFiles* 节的更多信息，参见《*Plug and Play,Power Management,and Setup Design Guide*》，以及《*Windows 2000 Driver Development Reference*》卷 1。

如果由 RequiredAll 说明的网络组件安装失败，依赖于该组件的网络组件也无法成功安装。

1.2.7.6 说明 NetCLient 组件的名字和提供者

在用户接口中可见的安装 NetClient 组件的 INF 文件必须在该组件的 *Service* 键中增加一个 NetworkProvider

键。INF 文件通过 *add-registry-section* 增加 NetworkProvider 键，这在该组件的 *Service-install* 节中用 AddReg 引用。

NetWorkProvider 键有 2 个值：一个描述网络提供者字的 Name 和描述网络提供者 DLL 完全路径的 ProviderPath。

下面是一个 *add-registry-section* 的例子，用以向组件实例键增加 NetworkProvider 键。

```
[NWCWorkStation.AddReg]
HKR,NetworkProvider,Name,0,"NetWare or Compatible Network"
HKR,NetworkProvider,ProviderPath,0x2000,"%11%\nwprovau.dll"
```

安装 NetClient 组件的 INF 文件不必修改组件的...Control\Network\Provider\Order 键下的 ProviderOrder 值。这由网络类安装器自动完成。

1.2.7.7 增加 HelpText 值

NetTrans,NetClient,NetService 网络组件的 INF 文件应该在组件 Ndi 键中增加 **HelpText** 值(REG_SZ)。**HelpText** 值是一个字符串，说明组件的功用。例如，NetClient 组件的 **HelpText** 值不应只简单地指明这个客户，而且还需指出客户允许用户和什么连接。在连接属性对话框的 Genenal 页中，当页中组件被选择时，**HelpText** 值出现在页底部。Net 组件(适配器)和 IrDA 组件不支持 **HelpText** 值。下面是 *add-registry-section* 的例子，用来向 **Ndi** 键中增加 **HelpText** 值：

```
[ms_Protocol.ndi_reg]
HKR,Ndi,HelpText,0,%MyTransport_Help%
```

HelpText 值是一个*%strkey%*形式的标志，这个在 INF 文件的 **Strings** 节中定义。**Strings** 节的更多信息参见《*Plug and Play,Power Management,and Setup Design Guide*》，以及《*Windows 2000 Driver Development Reference*》卷 1。

1.2.7.8 为通知对象增加注册值

NetTrans,NetClient 或 NerService 组件可以有一个通知对象，用来实现以下功能：

- 显示组件用户接口
- 将绑定事件通知组件，使组件能实现绑定过程上的一些控制。
- 提供条件安装。

详情参见第 2 章。

Net 组件(适配器)不支持通知对象，它使用的是协作安装程序。协作安装程序的详情，参见《*Plug and Play,Power Management,and Setup Design Guide*》,以及《*Windows 2000 Driver Development Reference*》的卷 1 的协作安装程序文档。

如果组件提供通知对象，则此组件的 INF 文件必须将下面两个值加进组件 **Ndi** 键。

- **ClsID**

说明组件对象的 GUID。通过运行 uuidgen.exe 得到 GUID。详情参见 Platform SKD。

- **ComponentDll**

说明通知对象 DLL 的路径。如果 DLL 不在 Window\system32 目录下，则应说明为完全路径。

下面是将 **ClsID** 和 **ComponentDll** 加入到 **Ndi** 键中的 *add-registry-section* 例子：

```
[MS_Protocol.ndi.reg]
HKR,Ndi,ClsID,0, “GUID”
HKR, Ndi,ComponentDll,0,"notifyobject.dll"
```

有通知对象的组件的 DDInstall 节必须包含 **CopyFiles** 指令，来引用 *file-list-section*, 该节将通知对象 DLL 复制到 *DirectionDirs* 节说明的目的目录中。关于 **CopyFiles** 指令和 *DirectionDirs* 节的详情参见《*Plug and Play,Power Management,and Setup Design Guide*》，以及《*Windows 2000 Driver Development Reference*》的卷 1 的 INF 文档。

1.2.7.9 向 Ndi 键增加服务相关值

如果组件有一相连的服务(设备驱动程序)，*add-registry-section* 必须将 **Service** 值加入到 **Ndi** 键中。该值是个 REG_SZ 值，说明了与组件相连系的主要服务。**Service** 值必须和 **AddService** 指令中的 *ServiceName* 参数一致，该指令引用了 *Serivce-install-section*(参见 1.2.8 节)。

如果组件有 1 个或多个相连系的服务，由 *DDInstall* 节引用的 *add-registry-section* 必须将 **CoServices** 值加入到 **Ndi** 键中。**CoService** 值是 MULTI_SZ 值，说明了组件安装的所有服务，包括 **Service** 值说明的主要服务。所有 NetTrans,NetClient 和 NetService 组件都需要 **CoServices** 值。由于仅有 1 个服务能连系于 1 个适配器，Net 组件(适配器)不应支持

CoServices 值。除了关闭服务，所有服务相关的操作按它们在 Coservices 中所列的顺序执行。例如，服务按它们所列的顺序开始，关闭时却是逆序的。只有当服务在 CoServices 中列出时，服务相关的操作才能被执行。

如果在 CoServices 中列出的服务不想在组件安装时启动，这些服务应该在 ExcludeSetupStartServices 值(MULTI_SZ)中列出，该值被加入到 Ndi 键中。

下面是一个 *add-registry-section*,用来将服务相关值加入到 Ndi 键中：

```
[Ms_Protocol.ndi.reg]
HKR,Ndi,Service,0,"MYT3"
HKR,Ndi,CoService,0x10000,"MYT3","MYT3CO"
HKR,Ndi,ExcludeSetupStartService,0x10000,"MYT3CO"
```

1.2.7.10 说明绑定接口

对安装的每个网络组件，网络 INF 文件必须为此组件说明向上和向下的接口，这可以通过向 NDi 键中加入 Interface 键来达到。

Interface 键至少有 2 个值：

- 1 个 UpperRange 值(REG_SZ)，用以定义组件可以绑在其上边界的接口。
- 1 个 LowerRage 值(REG_SZ)，用以定义组件可以绑定在其下边界的接口。(对物理适配器来说，这个接口是网络介质，如以太网或信令环网)。

Windows 95/98 的网络 INF 文件中的 DefUpper 和 DefLower，Windows 2000 的 INF 文件不支持。

下表列出了微软支持的 UpperRange 值：

UpperRange 值	描述
Netbios	NetBIOS
Ipx	IPX
Tdi	TCP/IP 的 TDI 接口
ndis5	NDIS 5.x(ndis2,ndis3 和 ndis4 不应再用)。对于非 ATM 网络组件这个值必须说明，如非 ATM 适配器，它的上边界与 NDIS 接口。
Ndisatm	ATM 支持的 NDIS 5.x。这个值在 ATM 网络组件中是必须的，如 ATM 适配器，它的上边界和 NDIS 接口相接。
ndiswan	WAN 适配器的上边界。这个值的说明导致操作系统自动使 WAN 适配器用于 RAS。
Ndiscowan	WAN 适配器的上边界，面向连接的 NDIS 在上面运行。
Noupper	所有不暴露上边界给绑定用途的组件的上边界。例如有一个 private 接口的组件。
Winsock	Windows socket 接口
ndis5_atalk	NDIS 5.x Net 组件(适配器)的上边界，在其上边界仅绑定 AppleTalk 接口
ndis5_dlc	NDIS5.xNet 组件适配器的上边界，在其上边界仅绑定 DLC 接口
ndis5_ip	NDIS5.xNet 组件(适配器)的上边界，在其上边界仅绑定 TCP/IP 接口
ndis5_ipx	NDIS5.xNet 组件(适配器)的上边界，在其上边界仅绑定

	IPX 接口
ndis5_nbf	NDIS 5.x Net 组件(适配器)的上边界，在其上边界仅绑定 NetBEUI 接口
ndis5_strems	NDIS5.xNet 组件(适配器)的上边界，在其上边界仅绑定 streams 接口

下表是微软支持的 LowerRange 值的列举：

LowerRange 值	描述
ethernet	以太网适配器的下边界
atm	ATM 适配器的下边界
tokenring	令牌环网的下边界
serial	serial 适配器的下边界
fddi	FDDI 适配器的下边界
baseband	baseband 适配器的下边界
arcnet	Arcinet 适配器的下边界
localtalk	LocalTalk 适配器的下边界
isdn	ISDN 适配器的下边界
wan	WAN 适配器的下边界
nolower	某些组件的下边界,这些组件都不将下边界暴露给绑定用途。
ndis5	NDIS 5.x(ndis2,ndis3,ndis4 不再使用)。对于所有下边界通过 NDIS 与非 ATM 组件接口的组件都必须说明。
Ndisatm	由 ATM 支持的 Ndis5.x。所有下边界通过 NDIS 与 ATM 组件接口的组件都必须说明。

UpperRange 和 LowerRange 说明了组件可绑定的接口类型，而不是实际的组件。绑定引擎将网络组件绑定到所有提供相应接口(在适当的边界上)的组件。例如，LowerRange 值为 ndis5 的协议，绑定到所有 UpperRange 值为 ndis5 的组件，如物理或虚拟适配器。

如果 NDIS 5.x Net 组件(适配器)和一个或多个协议工作，那么它的 UppeRange 应该赋予一个或多个协议值，如 ndis5_atalk,nids5_dlc,ndis5_ipx,ndis5_nbf 或 ndis5_streams。这样的 Net 类组件不应将其 UpperRange 值赋为 ndis_5，因为这将使该组件绑定于所有提供 ndis5 下边界的协议。INF 文件开发者可以对 private 绑定接口使用供应商的特定 UpperRange 和 LowerRange 值。例如，如果供应商只想将其适配器绑定到自己的私有协议驱动程序，那么 INF 文件开发者可以将适配器的 UpperRange 说明为 XXX，私有协议的 LowerRanger 说明为 XXX。Windows 2000 绑定引擎将所有 UpperRange 值为 XXX 的组件(此例中是适配器)绑定到所有 LowerRange 值为 XXX 的组件(此例中是私有协议)上。

下面是 *add-registry-section* 的一个例子，用于为 ATM 适配器增加 UpperRange 和 LowerRange。

```
[addreg-section]
HKR,Ndi\Interfaces,UpperRange,0,"ndisATM"
```

HKR,Ndi\Interfaces,LowerRange,0,"atm"

1.2.7.11 为高级属性页说明配置参数

安装 Net 组件(适配器)的 INF 文件可以说明适配器配置参数,这些参数将在组件的高级属性页中显示。用户在高级属性页中说明的配置值被写到此组件的根实例键中。

如果适配器支持高级属性页，适配器的 *DDInstall* 节中的 **Characteristics** 项必须包括 NCF_HAS_UI 值。网络 INF 文件通过 *add-registry-section* 说明在高级属性页中显示的配置参数，*add-registry-section* 在该组件的 *DDInstall* 节中引用。这种 *add-registry-section* 在 **Ndi\params** 键中增加一个或多个子键。配置参数的子键格式为 **Ndi\params***SubkeyName*，*SubkeyName* 是说明供应商参数名字的 REG_SZ 值。例如，说明 transceiver 类型参数的键，可以命名为 **Ndi\params\TransceiverType**。

以下的键保留，不能作为 **Ndi\params***SubkeyName* 使用。这些键包括 BundleId,Characteristics,ComponentId,Description,DriverDesc,InfPath,InfSection,InfSectionExt,Manufacturer,NetCfgInstanceId,Provider 和 ProviderName。

对于每个加入到 **Ndi\params** 中的参数子键，*add-registry-section* 必须加入 **ParamDesc**(参数描述)和 **Type** 值。*add-registry-section* 也可以为这个参数增加 **Default** 和 **Optional** 值，并且如果参数式数字的，也可增加 **Min**,**Max** 和 **Step** 值。下表描述了可被加入到 **Ndi\params** 键中的值:

值名	值	描述
ParamDesc	String	说明在高级属性页中显示的参数名
Type	int,long,word,dword, edit,或 enum	说明参数类型 int,long,word 说明是数字参数。edit,enum 说明是文本参数。
Default	默认值	为参数说明默认值。对数字参数，默认值必须是数值(int,long,word 或 dword)。对文本参数，默认值必须是字符串。必选参数必须设默认值，可选参数不应设默认值。
optional	0 或 1	0 说明参数是必需的。1 说明参数是可选的。在高级属性页中用户可以将可选参数设置为 Not Present。但对于必需参数，用户必须说明一个值或使用默认值。
Min	数字值	说明数字参数的最小值
Max	数字值	说明数字参数的最大值
Step	数字值	说明数字参数有效值之间的步长。以最小值为起点。

enum 参数的值域由如下形式的子键说明：

Ndi\params*SubkeyName***\enum**

每个枚举值都必须提供这个键。每个 **enum** 子键说明一个数字值(从 0 开始)和对该值的描述。

下面是 *add-registry-section* 的例子，用来增加名为 TransType 的配置参数。

```
[a1.params.reg]
HKR,Ndi\params\TransType,ParamDesc,0,"TranseiverType"
HKR,Ndi\params\TransType,Type,0,"enum"
HKR,Ndi\params\TransType,Default,0,"0"
HKR,Ndi\params\TransType,optional, 0,"0"
HKR,Ndi\params\TransType\enum,"0",0"Auto-Connector"
```

HKR,Ndi\params\TransType\enum,"1",0,"Thick Net(AUI/DIX)"
HKR,Ndi\params\TransType\enum,"2",0,"Thin Net(BNC/COAX)"
HKR,Ndi\params\TransType\enum,"3",0,"Twisted-Pair(TPE)"

1.2.7.12 为网络适配器说明定制属性页

如果高级属性页没有为 Net 组件(适配器)提供合适的配置选择,可以生成一个或多个定制属性页，如下所示：

1.生成 Microsoft Win32 属性页。参见 Platform SDK

2.生成属性页扩展 DLL,该 DLL 提供 AddPropSheetPageProc 和 ExtensionPropSheetPageProc 回调函数。参见 Platform SDK。

3.用 *add-registry-section* 将 EnumPropPages32 键加入到适配器的实例键中。EnumPropPages32 键有两个 REG_SZ 值：输出 AddPropSheetPageProc 函数的 DLL 名字和 ExtensionPropSheetPageProc 函数的名字。下面是一个 *add-registry-section* 例子，用来增加 EnumPropPages32 键：

HKR, EnumPropPages32,0,"DLL name,ExtensionPropSheetPageProc function name"

4.在适配器的 INF 文件中，包括一个 *CopyFiles* 节，将属性页扩展 DLL 复制到 Windows/systems32 目录下。关于 *Copyfile* 节的详情，参见《*Plug and Play,Power Management,and Setup Design Guide*》，以及《*Windows 2000 Driver Development Reference*》的卷 1 的 INF 文档。

5.在适配器的 *DDInstall* 节中，将 NCF_HAS_UI 说明为一个 Characteristics 值(参见 1.2.4 节)，用以指明适配器支持的一个用户接口。

6.当用户确认属性页中所作的改动后，属性页扩展 DLL 必须调用 SetupDiGetDeviceInstallParams, 并在在参数 SP_DEVINSTALL_PARAMS 结构中设置 DI_FLAGEX_PRORCHANGE_PENDING 标志,然后用该结构调用 SetupDiSetDeviceInstallParams, 重装驱动程序，从而读入改变后的参数值。

1.2.7.13 说明过滤器服务值

安装网络过滤器组件的 INF 文件必须说明过滤器服务值。本节将描述怎样定义一个过滤器服务。网络过滤器组件包括以下两个部分：

- 过滤器服务
- 过滤器设备

一个网络过滤器的服务和设备属于同一过滤器驱动程序。安装网络过滤器既需要过滤器服务的 INF 文件也需要过滤器设备的 INF 文件。过滤器服务 INF 文件必须说明 Ndi 键下的过滤器组件的 Service 值。过滤器设备 INF 文件，包含了 AddService 指令，该指令引用了 *Service-install* 节，在该节中说明了过滤器服务在何时、怎样被装载。AddService 指令的 ServiceName 值必须和过滤器组件的服务（Service）名相匹配。过滤器设备 INF 文件中的 *Service-install* 节的 ServiceBinary 项说明了过滤器驱动程序的二进制路径。在过滤器安装过程中，用过滤器服务 INF 文件的 *CopyFiles* 指令，将过滤器驱动程序传输到目标计算机中。参见 1.2.7.9 节和 1.2.8 节。

下面是过滤器服务 INF 文件说明过滤器组件服务名字的一个例子：

HKR, Ndi,service,,sfilter

这个过滤器组件服务名字也就是要提交给 NDIS 的名字。过滤器组件服务名字不需要和过滤器驱动程序的二进制名字相同，但在通常情况下它们是相同的。

当 INF 文件增加过滤器组件服务时,过滤器设备 INF 文件是怎样引用过滤器组件服务名的呢?

示例如下：

```
[SFilterMP.ndi.Services]
AddService=Sfilter,2,SfilterMP.AddService
［SFilterMP.AddService］
DisplayName=%SFilter_Desc%;"Sample Filter Miniport"
ServiceType=1;SERVICE_KERNEL_DRIVER
StartType=3;SERVICE_DEMAND_START
ErrorControl=1;SERVICE_ERROR_NORMAL
SericeBinary=%12%\passthru.sys;filterdriver
LoadOrderGroup=PNP_TDI
```

过滤器服务 INF 文件的 **Ndi** 键必须支持如下注册表项，用以定义过滤器服务：

FilterClass

FilterClass 说明了过滤器服务类。过滤器类可以是下表值中的一个：

值	含义
scheduler	包调度过滤服务。这个过滤器类在链中是最高的。包调度程序检测 802.1p 优先权，这优先权是由 QoSsignaling 组件赋给包的。包调度程序根据优先级将这些包发送给低层的驱动程序。
loadbalance	负载平衡过滤服务。这个过滤器类在包调度和失败结束过滤器之间。负载平衡过滤器通过将工作负载分配给微端口实例束，平衡包传输的工作负载。
failover	失败结束过滤器。这个过滤器类是链中最低的。那就是说没有其他过滤器可以处于这个过滤器设备和适配器之间。

FilterClass 值确定过滤器在过滤器栈中的位置。

注意：每个过滤器服务类只有一个过滤器存在于过滤器分层栈中。例如，两个调度过滤器不能同时在栈中出现。

FilterDeviceInfFile

FilterDeviceInfFile 为过滤器设备说明 INF 文件的名字。过滤器设备 INF 文件定义了过滤器设备，过滤器服务用它为某个网络适配器过滤信息。过滤器设备是一个虚拟 NIC。

FilterDeviceInfId

FilterDeviceInfId 说明了过滤器设备的标识符。标识符在过滤器设备的 INF 文件的 *Models* 节中列出。过滤器设备是由过滤器驱动程序为每个物理适配器初始化的微端口实例。

FilterMediaTypes

FilterMediaTypes 说明了过滤器服务过滤的介质类型。介质类型列表，参见 1.2.7.10 节中的微软支持的 **LowerRange** 值列表。过滤器服务 INF 在 **Ndi** 键下 **Interfaces** 键中说明了这些介质类型。

注意：为 **FilterMediaTypes** 说明合适的介质类型是很关键的。过滤器设备只能安装在这种物理适配器上，即该适配器所连的网络介质至少要和过滤器介质类型中的一个相匹配。物理适配器的 **LowerRonge** 项说明了网络介质，如以太网或信令网。

下面是过滤器服务 INF 文件定义过滤器服务的值的一个例子：

```
HKR,Ndi,FilterClass,failover
HKR,Ndi,FilterDeviceInfFile,netsf_m.inf
```

HKR,Ndi,FilterDevviceInfId,ms_sfiltermp
HKR,Ndi,\Interfaces,FilterMediaTypes,"ethernet,tokenring,fddi"
虽然其他网络 INF 文件可能为安装的组件说明向上和向下绑定接口，但是过滤器服务 INF 文件必须说明,它不是出于绑定目的而暴露上、下边界。过滤器服务 INF 文件在 **Interface** 键中说明这个特性。下面是说明这个特性的例子：
HKR,Ndi\Interfaces,UpperRange,,noupper
HKR,Ndi\Interfaces,LowerRange,,noloner

1.2.7.14 说明束成员关系

安装 Net 组件(物理或虚拟适配器)的 INF 文件可说明这些适配器属于同一个适配器束。NDIS 中间层驱动程序和过滤器驱动程序(输出虚拟网络适配器)包括在 Net 类中。NDIS 驱动程序通过在适配器束上分配工作负载，来平衡负载。
关于负载平衡的更多信息，参见*《Miniport NIC Driver》*的第 2 部分第 10 章。
为了说明适配器属于某个适配器束，安装适配器的驱动程序 INF 文件必须包含 BundleId 键和一个大小写敏感的字符串值(REG_SZ)。这个字符串值说明了适配器的驱动程序束。这个注册值用束标志符信息来配置。
下面是一个驱动程序 INF 文件的 *add-registry-section* 例子,用来将 **BundleId** 子键加入到 **Ndi\params** 键中，并赋给 **BundleId** 的 **ParamDesc**(参数描述)一个字符串值“Bundle1”。
[al.params.reg]
HKR,Ndi\Params\BundeId,ParamDesc,0,"Bundle1"

1.2.7.15 Window 2000 中不用的 Window 95/98 Ndi 值和键

下面列出的 **Ndi** 注册键和值在 Windows 95/98 中使用，但在 Windows 2000 中不再使用。这些信息帮助开发者把驱动程序从 95/98 上移植到 Windows 2000 上。

- **DevLoader**
- **DeviceVxD**
- **DriverDesc**
- **Ndi\DeviceID**
- **Ndi\NdiInstaller**
- **InfFile**
- **InfSelection**
- **Ndi\InstallInf**
- **Ndi\CardType**
- **StaticVxD**
- **Ndi\Interfaces\DefUpper**
- **Ndi\Interfaces\DefLower**
- **Ndi\Interfaces\RequireAny**
- **Ndi\Compability**
- **Ndi\Install**
- **Ndi\Remove**
- **Ndi\Params***Param_key_name*\flag**
- **Ndi\Params***Param_key_name*\location**
- Ndi**param_key_name**\resc
- **Ndi***filename*\...

• NDIS\...
Windows 2000 不支持 Ndi*param_key_name*\resc 和 Ndi\Params*param_key_name*\flag 值。这意味着用户不能通过高级页配置适配器资源。

1.2.8 DDInstall.Service 节

DDInstall.Services 节包括一个或多个 AddService 指令，每个指令引用了 INF 作者定义的 *Service-install-section*,它说明了组件的驱动程序何时以及如何加载。
在安装 Net 组件(适配器)的 INF 文件中必需一个 *DDInstall*.Service*s* 节。即在安装 NetTrans,NetClient 或 NetService 组件的 INF 文件中该节是可选的。
DDInstallService 节中的 AddService 指令也能引用为组件安装错误日志的 *error-log-install-section*。错误日志对所有网络组件是可选的。*DDInstall*.Services 节和 AddService 指令的细节化描述参见《*Plug and Play, Power Management, and Setup Design Guide*》以及《*Windows 2000 Driver Development Reference*》。第 1 卷的 INF 文件格式描述，。
下面是 *DDInstall*.Services 节，*service-install-section*,*error-log-install-section* 和 *add-registry-section*(被 *error-log-install-section* 中的 AddReg 指令引用)的例子：

```
[a1.ndi.NT.Services]
AddService=a1,2,a1.AddService,a1.AddEventLog
[a1.AddService]
DisplayName=%Adapter1.DispName%
ServiceType=1;SERVICE_KERNEL_DRIVER
StartType=2;SERVICE_AUTO_START
ErrorControl=1;SERVICE_ERROR_NORMAL
ServiceBinary=%12%\a1.sys
LoadOrderGroup=NDIS
[a1.AddEventlog]
AddReg=a1.AddEventLog.reg
[a1.AddEventlog.reg]
HKR,,EventMessageFile,0x00020000,"%%SystemRoot%%\system32\netevent.dll"
HKR,TypeSupported,0x00010001,7
```

AddService 指令的 *ServiceName* 参数(上例中的 a1),必须和组件的 Ndi\Service 值相匹配。参见 1.2.7.9 节。

1.2.9 NetworkProrider 和 PrintProvider 节

由于 NetClient 组件向用户提供网络服务，它们被认为是网络提供者。NetClient 组件如 Microsoft Client for Networks 和 NetWare Client。
除了作为网络提供者外，NetClient 组件还可以是一个打印提供者。打印提供者向网络中的用户应用程序提供打印服务。
NetClient 组件总是作为网络提供者来安装。安装 NetClient 组件的 INF 文件不需要 *NetworkProvider* 节，除非下述条件之一成立：
• 说明了这个组件的替代设备名字。
• 为使用 net view 命令，为组件指定短名(参见 1.2.9.1 节)。
安装作为打印提供者的 NetClient 组件的 INF 文件，必须为该组件包含一个 *PrinterProvider* 节(参见 1.2.9.2 节) 。安装 NetClient 组件的 INF 文件也必须包含一个

add-registrg-section(通过 *service-install-section* 中的 AddReg 指令引用)，用来向组件 service 键中增加一个 NetworkProvider 键(参见 1.2.7.6 节)。

1.2.9.1 包含一个 NetworkProvider 节

NetworkProvider 节说明了 1 个或两个如下信息：
NetClient 组件的替代设备名和用于 net view 命令的短名。为了创建一个 NetworkProrider 节，将.NetworkProvider 扩展名加入到 *DDInstall* 节中，示例如下：

```
[DDInstall]                              ;InstallSection
[DDInstall.NetworkProvider]              ;NetworkProvidersection
```

说明一个设备名
正常情况下，网络类安装器通过将组件的 Ndi\Service 值(参见 1.2.7.9 节)复制到组件 service 键下的 NetworkProvider 键中，来创建网络提供者的设备名。为了给组件说明不同的设备名，在 *NetworkProvider* 节中包含 DeviceName 项，如下例所示：

```
[DDInstall-section.NetworkProvider]
DeviceName="nwrdr"
```

DeviceName 是可选的，仅在 Ndi\Service 值作为网络提供者的设备名不够用时，才会说明。
说明一个短名
为使用 NetWare 的 net view 命令而为网络提供者说明一个短名，将在 *NetworkProvider* 节中包含一个 ShortName 项，如下例所示：

```
[DDInstall-section.NetworkProvider]
ShortName="nw"
```

下面是 net view 命令所用短名的例子：

```
net view /n:nw
```

ShortName 比网络提供者的全名容易记，也容易打印。
ShortName 是可选的，仅在需要的时候说明。

1.2.9.2 包括一个 PrintProvider 节

安装 NetClient 组件（这里是一个打印提供者）的 INF 文件必须包括 *PrintProvider* 节。为了生成一个 *PrintProvider* 节，在 *DDInstall* 节中加入.PrintProvider 扩展名，如下例所示：

```
[DDInstall-section] ;InstallSection
[DDInstall-section.printProvider] ;PrintProvidersection
```

PrintProvider 节必须包括如下项：

- PrintProviderName

一个说明打印提供者的非本地字符串

- PrintProviderDll

打印提供者 DLL 的文件名

- DisplayName

说明打印提供者名字的本地字符串。DisplayName 可以不同于 PrintProviderName。
PrintProviderName 和 PrintProviderDll 项提供一些信息，这些信息被用作 AddPrintProvider 函数的输入值(PROVIDER_INFO_1 结构)。AddPrintProvider 函数将打印提供者组件作为打印提供者加入。关于 AddPrintProvider 函数的更多信息，参见 Platform SDK。
下面是 PrintProvider 节的一个例子：

[DDInstall-section.PrintProvider]
PrintProviderName="NetWare or Compatible Network"
PrintProviderDll="nwprovau.dll"
DisplayName="%NWC_Network_Display_Name%"

1.2.10 Winsock 节

提供 Winsock 接口的 NetTrans 组件的 INF 文件必须说明 Winsock 的依赖性。这个 INF 文件必须包含一个 *Winsockinstall* 节。为创建一个 *Winsockinstall* 节，将.Winsock 扩展名增加到该协议的 *DDInstall* 节名中。例如，如果 *DDInstall* 节命名为 *Ipx*，该协议的 *Winsockinstall* 节必须命名为 Ipx.Winsock。

Winsock 安装节必须包含 **AddSock** 指令。**AddSock** 指令说明供应商命名的节，该节中包括了增加到组件的 **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControl\Services***TransportDriver-Name***Params\Winsock** 键中的值。

由 **AddSock** 指令引用的供应商命名的节必须包含如下值：

值名	**描述**
TransportService	说明协议服务名的 REG_SZ 值。这必须和该协议 Ndi\Servce 值相同。(参见 1.2.7.9 节)
HelperDllName	一个 REG_EXPAND_SZ 值，说明该协议的 WindowsSocketsHelper(WSH)DLL。参见第 3 部分。
MaxSockAddLength	一个 REG_DWORD 值，说明了最大有效 SOCKADDR 的尺寸，以字节为单位。
MinSockAddLength	一个 REG_DWORD 值，说明了最小有效 SOCKADDR 的尺寸，以字节为单位

如果说明了名字空间提供的可选 **ProviderId**，下面的值也必须说明：

值名	**描述**
ProviderId	说明 GUID 的 REG_SZ 值，用来签别名字空间提供者。GUID 作为所有名字空间提供者的键。通过运行 uuidgen.exe 得到 GUID。详情参参见 platform SDK。
LibraryPath	一个 REG_EXPAND_SX 值,说明名字空间提供者 DLL 的完全路径。
DisplayString	一个本地化的字符串,说明名字空间提供者在用户接口中的显示名。
SupportedSpace	一个 REG-DWORD 值,说明一个由名字空间提供者支持的名字空间。下面是在 Winsock2.h 中定义的名字空间值：

名字空间	值
NS_ALL	0
NS_SAP	1
NS_NDS	2
NS_PEER_BROWSE	3
NS_TCPIP_LOCAL	10
NS_TCPIP_HOSTS	11

NS_DNS	12
NS_NETBT	13
NS_WINS	14
NS_NBP	20
NS_MS	30
NS_STDA	31
NS_CAIRO	32
NS_X500	40
NS_NIS	41
NS_WRQ	50

Version	一个可选的 REG_DWORD 值，说明了名字空间提供者的版本号。如果此值没有说明，则用缺省值(1)。名字空间提供者的详情参见 Platform SDK。

下例显示了 IPX 协议的 Winsock 节：

```
[Ipx.Winsock]
AddSock=Install.IpxWinsock
[Install.IpxWinsock]
TransportService=nwlinkpipx
HelperDllName="%%SystemRoot%%\System32\wshisn.dll"
MaxSockAddrLength=0x10
MinSockAddrLength=0xe
ProviderId="GUID"
LibraryPath="%SystemRoot%\\System32\\nwprovaa.dll"
DisplayString=%NwlnkIpx_Desc%
SupportedNameSpace=1
Version=2
```

通过包括一个 *Winsockremove* 节，INF 文件为一个协议删除 Winsock 依赖性。为创建一个 *Winsock-remove* 节，将.Winsock 扩展名加入到协议的 *Remove* 节名中。例如，协议的 *Remove* 节名是 Ipx.Remove,那么该 *Winsock-remove* 节名必须为 Ipx.Remove.Winsock。*Winsock-remove* 节包含一个 **DelSock** 指令，该指令说明了一个 *INF-writer-named* 节。*INF-Writer-named* 节必须说明需删除的传输服务。如果 **ProviderId** 已经注册，*vendor-named* 节也必须说明要删除的 **ProviderId**.下例显示了删除 Ipx 协议 Winsock 依赖性的两个节：

```
[Ipx.Remove.Winsock]
DelSock=Remove.IpxWinsock
TransportService=nwlinkipx
ProviderId="GUID"
```

1.2.11 网络组件安装需求总结

本节总结如下类型的网络组件安装需求：

- 网络适配器
- 网络协议驱动程序

- 中间层网络驱动程序
- 网络过滤器驱动程序
- 网络客户
- 网络服务

1.2.11.1 网络适配器的安装需求

本节总结了网络适配器的安装需求。

一般需求

INF 文件节	状态	注释
Version	需要	Class=Net ClassGUID={4D36E972-E325-11CE-08002BE10318}
SourceDisksNames 和 SourceDiskFiles	如果..，则需要	如果 INF 文件没有随 Windows 2000 发布时需要。如果 INF 文件随 Windows 2000 发布，在 version 节中必须说明一个 LayoutFile 项，并且不用 SourceDiskNames 和 SourceDiskFiles 节。
DestinationDirs	需要	没有特殊网络需求。
ControlFlags	需要	在安装即插即用(PnP)适配器的 INF 文件中必须包含 ExcludeFromSelect 项。对非 PnP 适配器，无需列出。
Manufacturer	需要	无特殊网络需求
Models	需要	hw-id 必须和 PnP 管理器的适配器提供的硬件 ID 相匹配。
DDInstall	需要	Characteristics 项可用的值： NCF_VIRTUAL,NCF_SOFTWARE_ENUMERATED,NCF_PHYSICAL,NCF_MULTIPORT_INSTANCED_ADAPTER,NCF_HAS_UI,NCF_HIDDEN,NCF_NOT_USER_REMOVABLE。 NCF_VIRTUAL,NCF_SOFTWARE_ENUMERATED 和 NCF_PHYSICAL 是相互排斥的。 对于物理适配器,BusType 项是必需的。对 EISA 适配器，EisaCompressedID 项是必需的。这个项说明了 EISA 压缩 ID 和适配器掩码。多端口网络适配器需要 Port1DeviceNumber 或 Port1FunctionNumber 项。
DDInstall.Services	需要	无特殊网络需求

add_reg_sections	需要	需要： 创建 *Ndi* 键 说明服务相关值 说明束成员关系(仅对 LBFO 微端口) 说明绑定接口 可用的绑定接口： **UpperRange**: ndis5,ndisatm,ndiswan,ndiscowan,noupper,ndis5_atalk,ndis5_dlc,nids5_ip,ndis5_ipx,ndis5_nbf,ndis5_streams **LowerRange**: ethernet,atm,tokenring,serial,fddi,baseband,broadbond,arcnet,isdn,localtalk,Wan 可选项： 为组件设置静态参数 需要其他网络组件的安装 为高级属性页说明配置参数 生成定制属性页
Strings	需要	无特殊网络需求

WAN 适配器的额外需求

以下部分描述 WAN 适配器的额外安装需求：

- 说明端点数目(如信道，电路或 bearer Channels)
- 为 ISDN 适配器说明键和值
- 安装多协议 WAN 适配器

不支持

Remove 节

通知对象

1.2.11.2 网络协议安装要求

本节总结了网络协议的安装需求

一般需求

INF 文件节	状态	注释
Version	需要	**Class**=NetTrans **classGuid**={4D36E973-E325-11CE-BFC1-08002BE10318}
SourceDiskNames 和 **SourceDiskFiles**	如果..，则需要，	如果 INF 文件没有随 Windows 2000 发布则需要。如果 INF 文件随 Windows 2000 发布，在 **Version** 节中必须说明一个 **LayoutFile** 项，不使用 **SourceDiskNames** 和 **SourceDisksFiles** 节。
DestinationDirs	需要	无特殊网络需求
ControlFlags	可选	无特殊网络需求
Manufacturer	需要	无特殊网络需求
Model	需要	hw_id 由提供者名字，下划线和制造商名或产品名组成如：MS_DLC。
DDInstall	需要	**Characteristics** 项值： NCF_HIDDEN,NCF_NO_SERVICE,NCF_NOT_USER_REMOVABLE,NCF_HAS_UI

DDInstall.Services	可选	无特殊网络需求
add_reg_sections	需要	需要： 创建 Ndi 键 说明绑定接口 允许的绑定接口： UpperRange: netbios,ipx,tdi,Winsock,noupper LowerRange: ndis5,ndisatm,nolower 可选： 为组件设置静态参数 需要其他网络组件的安装 说明相关服务值 说明 HelpText 值 为一个通知对象说明值
Remove	可选	
WinsockSections	可选	对于一个提供 Winsock 接口的协议，*Winsock_install* 是必需的，*Winsock-remove* 节是可选的。
Strings	需要	无网络特殊需求

1.2.11.3 中间层网络驱动程序的安装需求

本节总结了中间层网络驱动程序的安装需求。

INF 文件节	状态	注释
Version	需要	Class=Net ClassGuid={4D36E972-E325-11CE-BFC10802BE-10318}
SourceDiskNames 和 SourceDisksFiles	如..，则需要	如果 INF 文件没有随 Windows 2000 发布，这个节是必需的。 如果 INF 文件随 Windows 2000 发布，Version 节中的 Layoutfile 项必须说明，不使用 SourceDiskNames 和 SourceDisksFiles 节。
DestinationDirs	需要	无特殊网络需求
ControlFlags	可选	无特殊网络需求
Manafacturer	需要	无特殊网络需求
Models	需要	hw_id 由提供者名字，下划线和制造商名字或产品名字组成 如：MS_DLC
DDInstall	需要	Characteristics 项： NCF_VIRTUAL 是必需的。 NCF_HIDDIN 和 NCF_NOT_USER_REMOVABLE 是可选的。
DDInstall.Services	需要	无网络特殊需求
Add-reg-sections	需要	需要： 创建 Ndi 键 说明相关服务值 说明绑定接口 可用的绑定接口：

		UpperRange: ndis5,ndisatm,ndiswan,ndiscowan,noupper,nids5_atalk, ndis5_dlc,ndis5_ip,ndis5_ipx,ndis5_dbf,ndis5_streams LowerRange: ethernet,atm,tokenring,serial,fddi,baseband,broadban d,arcnet,isdn,localtalk,wan 可选: 为组件设置静态参数 需要安装其他网络组件
strings	需要	无网络特殊需求

1.2.11.4 网络过滤器驱动程序的安装需求

本节总结了网络过滤器驱动程序的 INF 文件需求。安装网络过滤器驱动程序需要 2 个 INF 文件:

- 一个是为过滤器服务提供的(Class=NetService)
- 一个是为过滤器设备提供的(Class=Net)

网络过滤器驱动程序的 ServiceINF 文件

INF 文件节	状态	注释
Version	需要	Class=NetService Classguid={4D36E975-E325-11CE-BFC1-08002BE10318}
SourceDiskNames 和 SourceDisksFiles	需要，如果..	如果 INF 不随 Windows 2000 发布，该节是必需的。如果 INF 文件随 Windows 2000 发布，Version 节中的 LayoutFile 项必须说明，不使用 SourceDisNames 和 SourceDisksFiles 节。
DestinationDirs	需要	无特殊网络需求
ControlFlags	可选	无特殊网络需求
Manufacturer	需要	无特殊网络需求
Models	需要	hw_id 由提供者名字,下划线和制造商名字或产品名组成。如:MS-DLC
DDInstall	需要	Characteristics 项: NCF-FILTER 是必需的。NCF_HAS_UI 和 NCF_NO_SERVICE 是可选的。
DDInstall.Services	可选	需要: 生成 Ndi 键 说明过滤器服务值: FilterClass,FilterDeviceInfFile,FilterDeviceInfId,FilterMediaTypes 说明绑定接口: UpperRange:noupper LowerRange:nolower 可选: 为组件设置静态参数

		需要安装其他网络组件 说明一个 HelpText 值 说明一个通知对象的值
Remove	可选	
Strings	需要	无特殊网络需求

网络过滤器驱动程序的 DeviceINF 文件

INF 文件节	状态	注释
Version	需求	Class=Net ClassGuid={4D36E972-E325-11CE-BFC1-08002BE10318}
ControlFlags	需要	这个节必须包含 ExcludeFromSelect 项
Manufacturer	需要	无特殊网络要求
Models	需要	hw_id 由提供者名字，下划线和制造商名字或产品名组成。如：MS_DLC。
DDInstall	需要	Characteristics 项： NCF_VIRTUAL 是必须的。 NCF_HIDDEN 和 NCF_NOT_USER_REMOVABLE 是可选的。
DDInstall.Service	需要	AddService 指令的 *ServiceName* 值必须和 Ndi 键下的过滤器组件服务值相匹配。
add_reg_sections	需要	需要： 创建 Ndi 键 说明服务相关值 可选： 为组件设置静态参数 需要其他网络组件的安装
Strings	需要	无特殊网络需求

1.2.11.5 网络客户的安装需求

本节总结了网络客户的安装需求

INF 文件节	状态	注释
Version	需要	Class=NetClient ClassGuid={4D36E974-11CE0-BFC1-08002BE10318}
SourceDisksNames 和 SourceDisksFiles	需要，如果..	如果 INF 文件没有随 Windows 2000 发布，该节是必需的。如果 INF 文件随 Windows 2000 发布，Version 节中的 LayoutFiles 项必须说明，不使用 SourceDiskNames 和 SourceDisksFiles 节。
DestinationDirs	需要	无特殊网络需求
ControlFlags	可选	无特殊网络需求
Manufacturer	需要	无特殊网络需求
Models	需要	hw_id 由提供者名字，下划线和制造商名字或产品名组成。如：MS_DLC。

DDInstall	需要	Characteristics 项可用值： NCF_HIDDEN ， NCF_NO_SERVICE ， NCF_NOT_USER_REMOVABLE，NCF_HAS_UI
DDInstall.Services	可选	无特殊网络需求
Add_reg_section	需要	需要： 创建 Ndi 键 说明绑定接口 可用的绑定接口： UpperRange:noUpper LowerRange:ipx,tdi,Winsock,netbios,nolower 可选： 为组件设置静态参数 需要安装其他组件 说明服务相关值 说明 HelpText 值 说明通知对象的值
Remove	可选	
NetworkProvider	如..，则需要。	如果说明了网络客户的可替代设备名字，或者为使用 net view 命令而为组件说明了一个短名时，该节是必需的。
PrintProvider	如..，则需要。	如网络客户是一个打印提供者，该节名是必须的。
Strings	需要	无特殊网络需求。

1.2.11.6 网络服务的安装请求

本节总结了网络服务的安装请求。

INF 文件节	状态	注释
SourceDisksNames 和 SourceDisksFiles	如果.., 则需要.	如果 INF 文件没有随 Windows 2000 发布，该节是必需的。 如果 INF 文件随 Windows 2000 发布，在 Version 节中的 LayoutFile 项必须说明，不使用 SourceDiskNames 和 SourceDisksFiles 节。
DestionationDirs	需要	无特殊网络需求
ControlFlags	可选	无特殊网络需求
Manufacturer	需要	无特殊网络需求
Models	需要	hw_id 由提供者名字，下划线和制造商名字或产品名组成。如：MS_DLC。
DDInstall	需要	Characteristics 项可用值： NCF_HIDDEN,NCF_NO_SERVICE,NCF_NOT_USER_REMOVABLE, NCF_HAS_UI
DDInstall.Service	可选	无网络特殊需求
add-reg-sections	需要	需要： 创建 Ndi 键 说明绑定接口 可用的绑定接口： UpperRane:noupper LawerRange:ipx,tdi,Winsock,netbios,nolower Optional:

		为组件设置静态参数
		需要安装其他网络组件
		说明服务相关值
		说明 HelpText 值
		说明通知对象的值
Remove	可选	
Strings	需要	无特殊网络需求

第二章 网络组件的通知对象

通知对象处理通知信息，这些通知是由网络配置子系统发送给代表相关网络组件的对象的。通知对象由一个 DLL 实现。通知对象被用来显示网络组件的属性页并给出组件网络配置的编程控制。

注意：如果下述两个条件同时成立，网络组件可能不需要通知对象。

- 如果网络组件通过它的信息(INF)文件安装和删除。
- 如果对网络配置信息的反应不是必需的。

下面章节描述了通知对象并解释怎样开放。

下面章节也提供了支持这种对象的接口方法的参考信息。

- 2.1 关于通知对象
- 2.2 创建通知对象
- 2.3 通知对象参考

2.1 关于通知对象

通知对象处理通知，这些通知是由网络配置子系统，发送给代表某个网络组件（它拥有通知该对象）的对象。可以拥有通知对象的网络组件有：

- 传输，如协议驱动程序
- 服务，如中间层驱动程序
- 客户，如拨号网络

注意网卡不支持也不能拥有通知对象。如果物理和虚拟网卡需要涉及安装/删除过程或配置网络，可以使用 INF 文件或设备协作安装程序机制。(协作安装程序的更多信息，参见*《Plug and Play, Power Management, and Setup Design Guide》*和*《Windows 2000 Driver Development Reference 》*卷 1 的 INF 文档)。

通知对象是一个组件对象模型(com)类对象，由作为 com 组件服务器的 DLL 实现。每种类型的网络组件和一个类安装器相连系。那个类安装器用来安装这种类型的网络组件，并注册网络组件拥有的 com 类对象。注册在网络组件的主要安装阶段结束之后进行。类安装器调用对象的入口函数，注册一个 COM 类对象。

当操作系统安装、升级或删除网络功能模块时，或当应用程序配置网络时，操作系统或该应用程序必须启动网络配置子系统。一旦网络配置子系统启动，创建一个通知对象实例，这个通知对象执行某些特定操作。

以下部分描述了通知对象接收的通知类型和通知对象执行的操作：

- 2.1.1 通知对象图
- 2.1.2 通知类型
- 2.1.3 网络组件的安装
- 2.1.4 网络组件的删除

• 2.1.5 升级网络组件
• 2.1.6 显示并改变属性
• 2.1.7 网络配置

2.1.1 通知对象图

图 2.1 显示了安装或控制网络的客户应用程序是如何调用网络配置子系统的。子系统调用网络类安装器安装网络组件并为这些组件注册通知对象。通知对象代表这些组件，回调子系统来配置网络。

图 2.1 网络配置子系统

2.1.2 通知类型

网络配置子系统向通知对象发送通知：
• 在网络安装期间。包括安装操作系统，在先前没有网络功能的操作系统上安装网络，升级操作系统和删除网络。
• 在网络配置期间。包括增加、删除、激活、去活网络组件，改变网络组件，改变网络配置子系统和网络组件绑定的方式。
• 在应用程序指导子系统显示网络组件（它拥有通知对象）的属性之后。

2.1.3 网络组件的安装

网络配置子系统安装网络组件如下：
1.网络配置子系统为某个组件类型调用类安装器。
2.类安装器调用 Setup API，从组件 INF 文件中获取信息，并安装组件。
3.如果组件拥有通知对象，那么类安装器取回实现通知对象的 DLL 名。这个 DLL 在组件 INF 文件中出现，如下所示：
HKR,Ndi,ComponentDll,0,"notifyobject.dll"
4.类安装器调用 DLL 的入口函数以注册通知对象。
5. 网络配置子系统创建一个通知对象的实例，并调用该对象的 INetCfgComponentControl::Initialize 方法。
6.网络配置子系统调用通知对象的 INetCfgComponentSetup::Install 方法，执行安装组件所需的操作。

7. 如果组件的安装是自动的，网络配置子系统调用通知对象的 InetCfgComponentSefup::ReadAnswerFile 方法,从一个自动安装文件中取出组件参数，该文件称为 *answer* 文件。
8.一旦网络配置子系统创建一个实例，并初始化这个通知对象，那么子系统将调用通知对象的 INetCfgComponentNotifyGlobal::GetSupportedNotifications 方法,来取回对象所需的通知类型。用这个信息，子系统将通知发送给对象。对象可以用这些通知控制网络安装和配置，这些安装和配置可能会影响拥有这个对象的组件。例如，如果子系统调用 INetCfgComponentNotifyGlobal::SysNotifyComponent 方法来通知对象:子系统要安装或删除其他网络组件，对象就有机会执行相关的操作。
9.一旦网络配置子系统创建通知对象的一个实例并将其初始化，子系统也将调用通知对象 INetCfgComponentNofifyBinding 接口中的任一方法,把有关变化（子系统将其他组件绑定到拥有通知对象的组件的绑定方式的变化）通知给对象。

10. 当网络配置子系统准备将组件属性应用到操作系统中时，它调用对象的 INetCfgComponentControl::ApplyRegisryChanges 方法给组件注册键下的组件参数赋值。通知对象调用它组件的 INetCfgComponent::OpenParamKey 方法来打开和取回组件注册键。
11. 为配置组件的驱动程序，网络配置子系统调用通知对象的 INetCfgComponentControl::ApplyPnpChanges 方法,并传递 INetCfgPnpReconfigCallback 接口。通知对象调用 INetCfgPnpReconfigCallback:：SendPnpRecconfig 方法,将配置信息发送到其组件驱动程序。

2.1.4 删除网络组件

网络配置子系统以如下描述的方式删除一个网络组件：
子系统创建通知对象的一个实例并调用对象的 INetCfgComponentControl::Initialize 方法。该方法初始化对象并提供对组件和网络配置的访问功能。
为执行删除组件所需的操作，网络配置子系统调用通知对象的 INetCfgComponentControl::Removing 方法。Removing 方法执行清除操作以准备删除组件。

为删除组件，网络配置子系统调用通知对象的 INetCfgComponentControl::ApplyRegistryChange 方法从注册表中删除网络组件信息。

2.1.5 升级网络组件

网络配置子系统升级网络组件，描述如下。
子系统创建一个通知对象的实例，并调用通知对象的 INetCfgComponentControl::Initialize 方法。这个方法初始化对象并提供对组件和网络配置访问功能。
当操作系统改变为新的、改进的或其他版本时，网络配置子系统调用通知对象的 INetCfgComponentSefup::Upgrade 方法，执行所需的操作。
为了升级组件，网络配置子系统调用通知对象的 INetCfgComponentSefup：：ApplyRetistryChanges 方法,来修改注册表中网络组件的信息。为了用升级后的信息配置组件驱动程序，网络配置子系统调用通知对象的 InetCfgComponentSefup::ApplyPnpReconfigCallback 方法，并传递 INetCfgPnpReconfigCallback 接口。

2.1.6 显示并改变属性

网络配置子系统显示网络组件的属性页，并改变组件的参数，如下所述。
一个用户打开网络控制面板应用程序,来显示和修改组件的属性。当用户打开该应用程序时，应用程序启动网络配置子系统。然后，网络配置子系统创建通知对象的一个实例，并调用对象的 INetCfgComponentControl::Initialize 方法。这个方法初始化对象并提供对组件和网络配置的访问功能。
为显示组件属性，网络控制面板应用程序调用组件的 INetCfgComponent::RaisePropertyUi 方法。RaisePropertyUi 方法调用通知对象的如下方法：
• INetCfgComponentPropertyUi::QueryPropertyUi。确定某一个环境对显示组件属性是否合适。
• INetCfgComponentPrpertyUi::SetContext 方法。指导组件的通知对象在某环境中显示组件属性。

• INetCfgComponentPrpertyUi::MergePropPages 方法。创建和归并定制属性页。
如果用户在一个定制属性页中改变了组件的某个属性，RaisePropertyUi 调用通知对象的 INetCfgComponentPropertyUi::ApplyProperties 方法,在内存中保存此变化。
为使改变生效，网络配置子系统调用通知对象的 INetCfgComponentControl::ApplyRetistryChanges 方法,来修改注册表中网络组件的信息。为了用修改后的信息配置驱动程序，网络配置子系统调用通知对象的 INetCfgComponentControl::ApplyPnpChanges 方法，同时传递 INetCfgPnpReconfigCallback 接口。

2.1.7 网络配置

通知对象向拥有它的网络组件，提供网络配置的编程控制，描述如下。
用户可以打开网络控制面板应用程序，配置网络的某些特性。打开后，应用程序启动网络配置子系统。然后该子系统创建一个通知对象的实例，并调用对象的 INetCfgComponentControl::Initialize 方法。这个方法初始化对象并提供对组件和网络配置的存取。一旦网络配置子系统创建并初始化这个通知对象，子系统调用通知对象的 INetCfgComponentNotifyGlobal::GetSupportedNotifications 方法,来取回对象所需的通知类型。用这个信息，子系统向对象发送所需的通知。对象用这些通知控制网络安装和配置，这些安装或配置能够影响拥有对象的组件。例如，如果子系统调用通知对象的 INetCfgComponentNotifyGlobal::SysQueryBindingPath 方法通知对象:子系统增加其他网络组件的绑定路径。从而使这个对象有机会请求子系统：去活这个绑定路径。另外，子系统可以调用通知对象 INetCfgComponentNotifyBinding 接口的所有方法。这些方法将绑定方式变化（子系统将其他网络组件绑定到拥有通知对象的组件上）通知对象。

2.2 创建通知对象

如果网络组件要执行如下操作，则应为网络组件创建通知对象：
• 需要在网络安装和配置上进行一些控制。
• 需要显示定制属性页，用户能用定制属性页修改组件属性。以下部分描述了创建通知对象的过程：
.2.2.1 装载通知对象 DLL 和类对象
.2.2.2 定义通知对象
.2.2.3 创建并初始化通知对象的一个实例
.2.2.4 安装、升级、删除组件
.2.2.5 为组件创建属性页
.2.2.6 为显示属性设置环境
.2.2.7 评价网络配置的变化
.2.2.8 将组件变化加到注册表中
.2.2.9 配置组件驱动程序

2.2.1 装载通知对象 DLL 和类对象

网络组件的通知对象应该用 Com 类对象实现，它位于作为 Com 服务器的 DLL 中。
特定通知对象的 DLL 应该提供一些入口函数
DllMian 函数允许网络配置子系统将 DLL 装载进子系统的虚地址空间。
DllRegisrterServer 和 DllUnregisterServer 函数将 DLL 类对象的信息放入注册表中。网络配置子系统用注册表信息定位并装载网络组件的通知对象。

DLLCanUnloadNow 函数允许网络配置子系统确定 DLL 是否正在使用。如果当前 DLL 不在使用，子系统可以安全地将其从内存中卸载。
为使通知对象的 DLL 成为一个 Com 服务器，通知对象类必须暴露一个类工厂。这个类工厂允许网格配置子系统创建一个通知对象的实例。类工厂应该从 IClassFactory 接口中继承。

2.2.2 定义通知对象

通知对象类从 INetCfgComponentControl 接口继承。然而，如果通知对象执行某些特殊操作，它们的通知类也必须从如下接口中继承实现：
. 如果通知对象执行一些操作，这些操作与拥有该对象的组件的安装、升级和删除相关，则通知类必须从 INetCfgComponentSetup 接口中继承。
. 如果通知对象为拥有对象的组件显示定制属性页，则通知类必须从 INetCfgComponentPropertyUi 接口中继承。
. 如果一个通知对象评价一些变化（网络配置子系统把组件（拥有对象的）绑定到其他组件的方式的变化），则相应通知类必须从 INetCfgComponentNotifyBinding 接口中继承。
• 如果通知对象评价影响拥有对象的组件的网络配置的变化，则相应的通知对象类必须从 INetCfgComponentNotifyGlobal 接口中继承。
在通知类中的某些数据成员对所有通知对象是一样的。而另一些数据成员随组件不同而变化的。
所有通知对象都应定义的数据成员包括：
• 指向网络组件实例的 INetCfgComponent 接口指针，该组件拥有通知对象。通知对象实例用此指针存取和控制组件
•指向网络配置对象实例的 INetCfg 接口指针。通知对象实例用这个指针存取网络配置各个特性。
• 为拥有通知对象的组件贮存参数信息的变量。
• 说明通知对象先前执行的操作的变量。定义常量来指示通知对象可能执行的不同操作。当网络配置子系统调用通知对象的 INetCfgComponentControl::ApplyRegistryChanges 方法，将配置变化加入注册表。ApplyRegistryChanges 用这个变量确定注册表的变化。例如，如果通知对象在它的 INetCfgComponentSetup::Install 方法中，执行与安装组件相关的操作(该组件拥此通知对象)，Install 将这个变量设置为 install.
• HKEY 类型的注册键。通知对象调用组件的 INetCfgComponent:: OpenParamKey 方法打开和获取注册键，它包含组件参数。通知对象然后将该键设置为 HKEY 成员。
为通知类定义构造函数和析构函数。也要考虑定义只有通知类能用的私有方法。
通知类应该实现 IUnknown 接口的所有方法。如果通知类从前面列表中提到的任一所选接口继承而来，那么该接口的所有方法都应实现。对通知对象接口的任一方法，E_NOTIMPL 都不是一个有效返回值。如果通知对象不需要某个方法的实现，则该方法只需返回 S_OK 即可。

2.2.3 创建并初始化通知对象实例

在子系统能够做如下操作之前,网络配置子系统必须创建一个通知对象实例并初始化对象：
• 提醒通知对象网络配置发生变化
• 为拥有对象的组件显示定制属性页
子系统通过 DLL 的类工厂中创建通知对象实例。类工厂调用通知类的构造函数。
类构造函数应该将初始值分配给类数据成员。构造函数应该分配的初始值包括：
• 构造函数应该将指向网络组件实例 INetCfgComponent 的接口指针赋为 NULL 值。
• 构造函数应该将指向网络配置对象实例的接口（INetCfg）指针赋为 NULL 值，

·构造函数应该将一个变量设置成表示未知操作的常量，该变量用来说明通知对象先前所执行的操作。此变量的详情，参见 2.2.2 节的“定义通知对象”。

一旦网络配置子系统创建一个通知对象实例，这个子系统将调用 INetCfgComponentControl::Initialize 方法来初始化对象实例。在这个调用中，子系统传递了 INetCfgComponent 接口指针。INetCfgComponent 向通知对象提供对象组件的实例，对象能用这个实例访问和控制组件。在这个调用中，子系统也传递 INetCfg 接口指针,向通知对象提供网络配置对象实例，通知对象可用来访问网络配置的所有特性。

Initialize 方法应该将网络配置子系统提供的 INetCfgComponent 和 INetCfg 接口指针赋给通知类的数据成员。Initialize 然后调用：

· INetCfg::AddRef 方法，用来增加网络配置对象的引用数。

· INetCfgComponent::AddRef 方法，用来增加拥有通知对象的网络组件的引用数。

在 Initialize 返回前，不可调用其他的通知对象的其他接口方法。

2.2.4 安装，升级和删除组件

当网络配置子系统安装、升级或删除网络组件时,子系统也调用组件的通知对象来完成安装、升级和删除操作。实现组件的通知对象，来执行组件所需的操作。例如：

· 当子系统安装组件时，组件通知对象安装组件需要的其他网络组件。通知对象的 INetCfgComponentSetup::Install 方法调用网络配置的 INetCfgClassSetup::Install 方法,安装一个网络组件。在这个调用中，组件通知对象的 Install 方法传递所要安装的组件的标识符。

· 通知对象将删除其他网络组件。为删除一个网络组件，组件通知对象的 INetCfgComponentSetup:: Removing 方法调用网络配置的 INetCfgClassSetup:: DeInstall。在这个调用中 Removing 传递要删除组件的 INetCfgComponent 接口指针。

·实现组件通知对象，当子系统升级一个组件时，组件通知对象可以改变组件绑定路径的顺序。为改变这个顺序，通知对象的 INetCfgComponentSetup::Upgrade 方法将调用 INetCfgComponentBinding::MoreBefore 或 INetCfgComponentBinding::MoreAfter 方法。

2.2.5 为组件生成属性页

在网络配置子系统调用通知对象的 INetCfgComponentPropertyUi::MergePropPages 方法之后，通知对象生成定制属性页。可以用 MergePropPages 方法将定制属性页并到组件属性单的页缺省集中。MergePropPages 返回定制属性页合并后的缺省页数目。

为了生成定制属性页，MergePropPages 调用 COM 的 CoTaskMemAlloc 函数为 PRORSHEETPAGE 结构的句柄分配空间。每个句柄代表了要创建的属性页。如果 CoTaskMemAlloc 为句柄成功分配了空间，MergePropPages 将为每个属性页声明和填写 PRORSHEETPAGE 结构。一旦 MergePropPages 填写了这些结构，它将为每个属性页调用 Win 32 的 CreatePropertySheetPage 函数。在这个调用中，MergePropPages 传递要生成的 PROPSHEETPAGE 结构地址。

MergePropPages 生成的每个属性页都实现一个对话框回调函数。对话框回调函数处理操作系统发送给与其相联系的属性页的消息。为将属性页和对话框函数连系起来，MergePropPages 应该把每个属性页的 PROPSHEETPAGE 结构中的 pfnDlgProc 成员指向该页对话框函数。

对话框函数处理如下信息：

· WM_INITDIALOG 消息，在操作系统显示属性页之前，把这个消息发送给对话框函数。对话框函数通常用这个消息初始化属性页，并执行影响属性页外观的任务。

· WM_NOTIFY 消息，把属性页中发生的一个事件发送到对话框函数。随该消息发送的其他信息用来鉴别发生了什么事件。事件信息包含在指向 NMHDR 结构的指针中。NMHDR 中包含的信息如下：

· PSN_APPLY 事件，指示用户点击了属性页上的 OK、Apply 或 Close 按钮。如果用户点击 OK、Close 或 Apply,那么对话框函数将调用 PropSheet_Changed 宏通知属性单，页中信息已经改变。在这个调用中，对话框函数传递属性单和属性页的句柄。对话框函数可以把页句柄作为参数，调用 Win 32 的 GetParent 函数，来获取属性单句柄。

在对话框函数将变化通知属性单后，网络配置子系统调用 INetCfgComponentPropertyUi::ValidateProperties 方法检查变化的合法性。如果变化是有效的，子系统调用通知对象的 INetCfgComponentPropertyUi::ApplyProperties 方法，使改变生效。网络配置子系统在操作系统关闭对话框之前,调用 ApplyProperties。

ApplyProperties 获取用户输入的信息，并给通知对象的数据成员设置信息。

· PSN_RESET 事件，指示操作系统将销毁(destroy)属性页。用户可以点击属性页的 Cancel，发起这个操作。如果用户点击了 Cancel,网络配置子系统调用 INetCfgComponentPropoertyUi::CancelProperties 方法，丢弃所有修改。在对话框关闭以前，网络配置子系统调用 CancelProperties。

· PSN_KILLACTIVE 事件，指示属性页将变成非活动的。这个事件在用户激活其他属性页或点击 OK 时发生。

属性页回调函数也可以由 MergePropPages 创建的不同属性页分别实现。该回调函数执行该页的初始化和消除操作。为了将属性页和属性页回调函数连系起来，MergePropPages 应该在 PRORSHEETPAGE 结构中，把 pfnCallback 成员指向该页的回调函数。

2.2.6 设置环境来显示属性

通知对象可以设置环境，在这个环境中为拥有该对象的网络组件显示属性。在网络配置子系统调用对象的 INetCfgComponentPropertyUi::SetContext 方法之后，且在子系统调用对象的 INetCfgComponentPropertyUi::MergePropPages 方法之前，通知对象设置显示环境。当网络配置子系统调用 SetContext 时，它传递一个 IUnknown 接口。SetContext 调用该接口的 QueryInterface 方法来确定子系统提供的特定对象的接口。

例如，网络配置子系统在调用 SetContext 时，能够提供 INetLanConnectionUiInfo 接口。SetContext 可以用 INetLanConnectionUIInfo 的 GetDeviceGuid 方法,取回 LAN 设备的 GUID。通知对象随后在 LAN 设备的环境中,显示网络组件的属性。例如，TCP/IP 协议的通知对象在适配器的环境中，显示一个和 LAN 适配器连系在一起的 IP 地址。这样用户可以为这个适配器配置 IP 地址。

2.2.7 评价网络配置的变化

在网络配置子系统调用通知对象的 INetCfgComponentNotifyGlobal 和 INetCfgComponentNotifyBinding 接口后，通知对象应该评价子系统发送的网络配置的变化，并执行与这些变化相关的操作。通知对象的 INetCfgComponentNotifyGlobal 和 INetCfgComponentNotifyBinding 接口的方法，处理对拥有对象的组件有影响的变化。

例如，网络配置子系统调用通知对象的

INetCfgComponentNotifyGlobal::SysNotifyComponent 方法并传递 NCN_ADD,提醒通知对象：子系统安装了其他网络组件。如果拥有对象的组件要绑定指定的安装组件，则通知对象将执行有助于绑定的操作。

2.2.8 将组件变化加入注册表

在网络配置子系统调用通知对象的 InetCfgComponentControl::ApplyRegistryChanges 方法后，通知对象根据先前做的操作，从注册表中设置、修改或删除信息。在通知对象执行特定操作后(这个操作与拥有对象的组件的安装，删除或参数修改有关)，通知对象设置某个数据成员来指示已执行的操作。在子系统调用 ApplyRegistryChanges 将配置变化加入注册表后，ApplyRegistryChanges 用这个数据成员确定如何修改注册值。

例如：如果通知对象先前执行了与安装组件（拥有该对象的）有关的操作，通知对象将数据成员设置为"install"。在子系统调用 ApplyRegistryChanges 将配置变化加入到注册表时，ApplyRegistryChanges 在注册表中设置组件信息。

如果通知对象先前执行有关删除组件（拥有对象的）的操作，通知对象应该将数据成员设置为"remove"。在子系统调用 ApplyReyistryChanges 时，ApplyRegistryChanges 从注册表中删除该组件的信息。

如果一个用户在组件的定制属性页中修改了一个属性，组件通知对象应该将数据成员设置为"modifyparameter"。在子系统调用 ApplyRegistryChanges 时，ApplyRegistryChanges 在注册表中修改组件参数信息。

为了打开并取回组件的注册表键，修改组件信息，ApplyRegistryChanges 方法应该调用组件的 INetCfgComponent::OpenParamKey 方法。为了在组件的注册键下设置值，ApplyRegistryChanges 用 Win 32 函数向注册表写数据。例如，ApplyRegistryChanges 可以调用 RegCreateKeyEX 函数来生成子键，用 RegSetValueEx 函数创建并设置值。

2.2.9 配置组件驱动程序

在网络配置子系统调用通知对象的 INetCfgComponentControl::ApplyPnpChanges 方法后，通知对象应该将配置信息发送给组件（拥有该对象的）的驱动程序。在网络配置子系统调用 INetCfgComporentControl::ApplyRegistryChanges 方法后，并且网络组件的服务和驱动程序已经启动后，网络配置子系统调用 ApplyPnpChanges。在 ApplyPnpChanges 调用中，网络配置子系统传递了 INetCfgPnpReconfigCallback 接口。组件的通知对象可以用 INetCfgPnpReconfigCallback 接口,把配置信息发送给组件驱动程序。这个驱动程序必须是 TDI 提供者或 NDIS 微端口。通知对象在它的 ApplyPnpChanges 中调用 INetCfgPnpReconfigCallback::SendPnpReconfig,将配置信息发送给组件驱动程序。SendPnpReconfig 向驱动程序传送配置信息。

通知对象也能调用 Win 32 的 CreateFile 函数打开与组件驱动程序的连接。通知对象可以调用 Win 32 DeviceIoControl 函数,向其组件驱动程序直接发送控制码和输入数据。

并不要求通知对象一定要使用 INetCfgPnpReconfigCallback。但是，如果通知对象使用 INetCfgPnpReconfigCallback,则用户无需重新启动操作系统，就可以使配置变化在驱动程序中生效。