目 录

表格 1 缩略语表

项目	英文描述	中文描述
NDIS	Network Driver Interface Specification	网络驱动程序接口标准
IMD	Intermediate Drivers	中间层驱动
TDI	Transport driver Interface	传输驱动程序接口
NIC	Network Interface Card	网络接口卡
SP	Service Pack	服务包
LAN	Local Area Network	局域网
LAN-E	LAN Emulation	局域网仿真
NAT	Network Address Translation	网络地址转换
LBFO	Load Balancing And Fail-Over	负载平衡和失效替换
DDK	Device Drivers Kit	设备驱动程序开发包
SMP	Symmetry Multiprocessing	对称多处理
OS	Operating System	操作系统
IDE	Integrated Development Environment	集成开发环境

1 NDIS中间层驱动程序

1.1 NDIS 中间层驱动程序(NDIS Intermediate Drivers)概述

微软 Windows 网络驱动程序接口标准（NDIS 4.0）和 Windows NT 4.0（SP3）引入了一种新的 NDIS 驱动程序，它可以嵌在 NDIS 传输驱动程序 TDI（如，TCP/IP）和底层的 NDIS 网络接口驱动程序的中间。这种新类型的驱动程序被称为 NDIS 中间层驱动，如图表 1。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。

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。

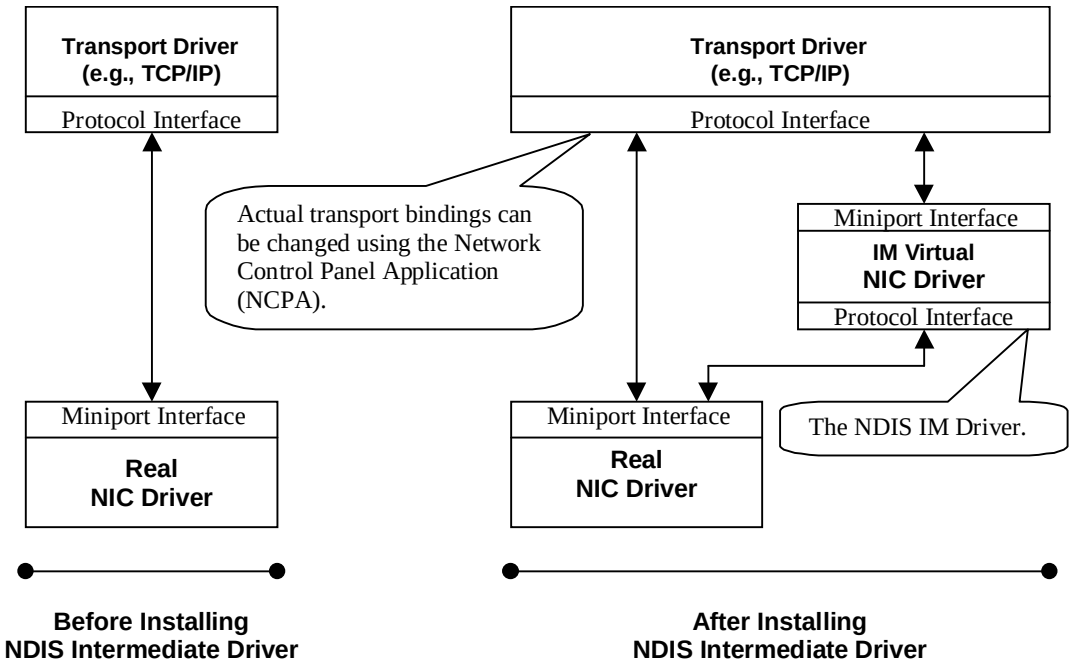
中间层驱动程序是一种典型的层次结构程序，它基于一个或多个 NDIS NIC 驱动程序，其上层是一个向上层提供 TDI（传输驱动程序接口）支持的传输驱动程序（也可能是多层结构）。从理论上讲，一个中间层驱动程序也可以是基于其他中间层驱动程序或作为其他中间层驱动程序的低层出现的，尽管这种方案未必能展现更好的性能。

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 NDIS API 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接请求。

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



图表 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。

1.2 NDIS 中间层驱动程序的用途

NDIS 中间层驱动有几个方面的用途，包括：

- **局域网仿真（LAN Emulation）** – NDIS 中间层驱动可以使一个非局域网 NIC 驱动（如，ATM）犹如一个局域网 NIC 驱动（如，Ethernet）。
- **包过滤（Packet Filtering）** - 可以拦截和修改高层 TDI（传输驱动程序）和底层 NIC 驱动程序之间的网络包（Packets）：
  - 通过或过滤掉（Pass/Drop Packets）
  - 延迟或重新排序（ Delay/Reorder Packets）
  - 加密或解密（Packet Encryption/Decryption）
  - 压缩或解压（Packet Compression/Decompression）
  - 路由包（Route Packets）：
    - NAT 网络地址转换（Network Address Translation）
    - LBFO 负载平衡和失效替换（Adapter Load Balancing And Fail-Over）

1.3 NDIS 中间层驱动程序的开发环境

OS : Microsoft Windows 2000 Server
IDE : Microsoft Visual C++ V6.0
DDK : Windows 2000 Device Drivers Kit

2 NDIS 中间层驱动程序的开发

2.1 可分页和可丢弃代码

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。

2.2 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*”。

2.3 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 2.4 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。

2.3.1 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。

2.3.1.1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 2.9 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 2.9 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

2.3.1.2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。

2.3.1.2.1 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 2.4 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 2.9 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。

2.3.1.2.2 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。

2.4 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 2.5 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。

2.4.1 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。

2.4.2 微端口(Miniport)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。

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。

2.4.3 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。

2.4.3.1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。

2.4.3.2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。

2.4.4 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。

2.5 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。

2.5.1.1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。

2.6 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。

2.7 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。

2.7.1 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。

2.7.1.1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 ProtocolReceive 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。

2.7.1.2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。

2.7.1.3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。

2.7.2 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。

2.7.2.1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。

2.7.2.2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。

2.7.3 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。

2.8 通过中间层驱动程序传输数据包

在 2.3.1 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 2.5 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 2.5 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包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。

2.8.1 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。

2.9 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 2.4.2 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。

2.9.1 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。

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。

2.9.2 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 2.9.3。

- 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。

2.9.3 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。

2.9.3.1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。

2.9.3.2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。

2.10 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。

2.11 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）。

在对该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。

2.12 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 2.11 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 2.10 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。

3 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：

- 关于 LBFO
- 指定对 LBFO 的支持
- 在微端口驱动程序上实现 LBFO

3.1 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。

3.2 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”*NDIS Network Drivers Design Guide*”第五部分的 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。

3.3 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：

- 初始化微端口束
- 平衡微端口驱动程序的工作量
- 在主微端口失效后提升一个次微端口

3.3.1 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。

3.3.2 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。

3.3.3 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。

4 安装网络组件

这部分包括：
· 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

4.1 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)。

4.2 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。

4.2.1 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

4.2.2 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-BECl-08002BE10318}
NetTrans	{4D36E973-E325-11CE-BFCl-08002BE10318}
NetClient	{4D36E974-E325-11CE-BFCl-08002BE10318}
NetService	{4D36E975-E325-11CE-BFCl-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-BECl-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。

4.2.3 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

4.2.4 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 =Ox84;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：

```
[al.isa]
BusType=1
[al.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。

4.2.5 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。

4.2.6 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。

4.2.7 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,Ox00010001,2
```

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,Ox00010001,4
HKR,,IsdnNumDChannels,Ox00010001,2
HKR,,IsdnAutoSwitchDetect,Ox00010001,1
HKR,,IsdnSwitchType,Ox00010001,Ox00000004;NI1
HKR,0,IsdnNumBChannels,Ox00010001,2
HKR,1,IsdnNumBChannels,Ox00010001,2
```

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=Ox4                                        ;NationalISDN1
[Enum\emumeratorID\device-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\device-instance-id\1]                  ;D-Channel1 键
IsdnNumBChannels=2
IsdnMultiSubscriberNumbers=867530923901257658156
[Enum\emumeratorId\device-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,Ox2000,"%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,O,“GUID”
HKR，Ndi,ComponentDll,O,"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,O,"MYT3"
HKR,Ndi,CoService,Ox10000,"MYT3","MYT3CO"
HKR,Ndi,ExcludeSetupStartService,Ox10000,"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 非 ATM	NDIS 5.x(ndis2,ndis3 和 ndis4 不应再用)。对于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 须	ATM 支持的 NDIS 5.x。这个值在 ATM 网络组件中是必的，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 仅绑	NDIS 5.x Net 组件(适配器)的上边界，在其上边界定 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。

```
[al.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：

```
[al.ndi.NT.Services]
AddService=a1,2,al.AddService,al.AddEventLog
[a1.AddService]
DisplayName=%Adapter1.DispName%
ServiceType=1;SERVICE_KERNEL_DRIVER
StartType=2;SERVICE_AUTO_START
ErrorControl=1;SERVICE_ERROR_NORMAL
ServiceBinary=%12%\al.sys
LoadOrderGroup=NDIS
[a1.AddEventlog]
AddReg=al.AddEventLog.reg
[a1.AddEventlog.reg]
HKR,,EventMessageFile,Ox00020000,"%%SystemRoot%%\system32\netevent.dll"
HKR,TypeSupported,Ox00010001,7
```

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 GUID	说明 GUID 的 REG_SZ 值，用来签别名字空间提供者。 作为所有名字空间提供者的键。通过运行 uuidgen.exe 得到 GUID。详情参见 platform SDK。
LibraryPath 全路	一个 REG_EXPAND_SX 值，说明名字空间提供者 DLL 的完 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 字	一个 REG-DWORD 值，说明一个由名字空间提供者支持的名 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 号。	一个可选的 REG_DWORD 值，说明了名字空间提供者的版本 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=Ox10
MinSockAddrLength=Oxe
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,broadband,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