An NDIS intermediate driver exports MiniportXxx functions at its upper edge and ProtocolXxx functions at its lower edge.

An NDIS intermediate driver can support only connectionless communication at its upper edge. At its lower edge, however, an NDIS intermediate driver can support either connectionless communication or connection-oriented communication. (For more information on connection-oriented communication, see Part 4, Chapter 1).

The miniport section (upper edge) of an intermediate driver must be deserialized. Deserialized drivers serialize the operation of their own MiniportXxx functions and queue internally all incoming send packets rather than relying on NDIS to perform these operations. This results in significantly better full-duplex performance, provided that the driver's critical sections (code which only a single thread at a time can execute) are kept small. However, deserialized miniports must meet additional and more stringent design requirements and also require additional debugging and testing time. See Part 2, Chapter 4, Section 4.5 for more information on deserialized drivers.

An intermediate driver is typically layered over one or more NDIS NIC drivers and under a transport driver (possibly multilayered) that supports TDI at its upper edge. Theoretically, an intermediate driver could be layered above or below another intermediate driver, although such an arrangement is unlikely to exhibit good performance.

An example of an intermediate drivers is a LAN-emulator intermediate driver layered below a legacy transport driver and above a miniport NIC driver for a non-LAN medium.

Such a driver receives packets in a LAN format at its upper edge, translates them to another NIC-native medium format and sends them on to an NDIS miniport for that NIC. On receives, this intermediate driver translates packets indicated up from the underlying NIC driver to a LAN-compatible format and indicates these converted packets to the upper level transport driver.

For example, NDISWAN has many of these properties. NDISWAN translates packets from the overlying transports' LAN format to WAN packet format and packets from the underlying NIC drivers' WAN packet format to LAN packet format. NDISWAN also optionally performs compression, encryption and PPP formatting if this is not supported by the underlying NIC hardware. NDISWAN contains a private interface for communication between NDISTAPI and the NIC driver. It also maps protocol bindings to active call connections.

Another example of an intermediate driver is an ATM LANE (LAN emulation) driver that translates packets from the overlying connectionless transport's LAN format to the ATM format used by the underlying connection-oriented NIC.

Figure 1.1 shows an intermediate driver.
Figure 1.1 Supported intermediate driver configuration

An NDIS intermediate driver interfaces to NDIS to forward packets sent by a driver above and to pass them to a driver below. When an intermediate driver receives packets from an underlying driver, it indicates them to the driver above either by calling a filter-specific `NdisMIndicateReceive` function or `NdisMIndicateReceivePacket`.

An intermediate driver calls NDIS to open and establish a binding to an underlying NIC driver or intermediate NDIS driver that exports a set of Miniport functions at its upper edge. An intermediate driver provides MiniportSetInformation and MiniportQueryInformation functions to process set and query requests from higher level driver(s) and, perhaps, to pass them through to a lower level NDIS driver by calling `NdisRequest` if it has a connectionless lower edge or `NdisCoRequest` if it has a connection-oriented lower edge.

An intermediate driver calls NDIS-provided functions to send packets on to still lower level NDIS drivers to the net. For instance, an intermediate driver with a connectionless lower edge must call `NdisSend` or `NdisSendPackets` to send a packet or array of packets. An intermediate driver with a connection-oriented lower edge must call `NdisCoSendPackets` to send an array of packets. If the intermediate driver is layered over a nonNDIS NIC driver, the send interface is opaque to NDIS after it calls the MiniportSend or Miniport(Co)SendPackets function of the intermediate driver.

NDIS provides a set of `NdisYxx` functions and macros that hides the details of the underlying...
NDIS provides a set of NdisXxx functions and macros that hides the details of the underlying operating system. For instance, an intermediate driver can call NdisMInitializeTimer to create a timer for synchronization purposes and NdisInitializeListHead to create a linked list. Intermediate drivers use NDIS functions in order to be more portable across Microsoft operating systems that support the Win32® interface.

**Pageable and Discardable Code**

As explained in Part 1, every MiniportXxx or ProtocolXxx function runs at a particular IRQL. The possible IRQLs for these functions range between PASSIVE_LEVEL up to and including DISPATCH_LEVEL in intermediate drivers.

Intermediate driver functions that always run at IRQL PASSIVE_LEVEL can be marked as pageable using the NDIS_PAGABLE_FUNCTION macro. Driver developers are encouraged to designate code as pageable whenever possible, freeing system space for code that must be memory-resident. A driver function that runs at IRQL PASSIVE_LEVEL can be made pageable as long as it neither calls nor is called by any function that runs at IRQL >= DISPATCH_LEVEL—for instance, a function that acquires a spin lock. Acquiring a spin lock causes the IRQL of the acquiring thread to be raised to IRQL DISPATCH_LEVEL. A function, such as ProtocolBindAdapter, that runs at IRQL PASSIVE_LEVEL, must not call NDIS functions that run at IRQL >= DISPATCH_LEVEL if ProtocolBindAdapter is marked as pageable code. For more information about NDIS functions that run at raised IRQL, see the Network Driver Reference, which specifies the IRQL for each of the NdisXxx functions.

The DriverEntry function of an intermediate driver, as well as code that is called only from DriverEntry, should be specified as initialization-only code, using the NDIS_INIT_FUNCTION macro. Code identified with this macro is assumed to only run once at system initialization time, and as a result, is only mapped during that time. After DriverEntry returns, code marked with the NDIS_INIT_FUNCTION macro is discarded.

**Synchronizing Access to Shared Resources**

Access to any driver-allocated shared resource must be synchronized if the resource can be simultaneously shared by two driver functions or if the intermediate driver can run on an SMP machine such that the same driver function can be attempting to simultaneously access the resource from more than one processor. For instance, if a driver maintains a shared queue, a spin lock can be used to synchronize access to that queue. The spin lock should be initialized before the queue is created by calling NdisAllocateSpinLock.

However, care should be taken not to overprotect a shared resource, such as a queue. For example, some read operations can be done without serialization. Any operation that manipulates the queue links, however, must be synchronized. Spin locks always should be used sparingly and held for as short a time as possible. See the Kernel-Mode Driver Design Guide for an in-depth discussion of spin locks.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK
1.1 Intermediate Driver DriverEntry Function

An intermediate driver’s initial required entry point must be explicitly named `DriverEntry` so that the loader can properly identify it. All other exported driver functions, described here as MiniportXxx and ProtocolXxx, can have any developer-specified name since they are passed as addresses to NDIS. The definition of `DriverEntry` is that of any kernel-mode driver:

```c
NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
);
```

If the driver exports a set of standard kernel-mode driver functions in addition to the ProtocolXxx functions, it must write the addresses of those standard functions in the driver object passed to `DriverEntry`.

In an intermediate driver, `DriverEntry` must at a minimum:

1. Call `NdisMInitializeWrapper` and save the handle returned at `NdisWrapperHandle`.
2. Call `NdisIMRegisterLayeredMiniport` to register the driver’s MiniportXxx functions, passing the handle obtained in Step 1.
3. Call `NdisRegisterProtocol` to register the driver’s ProtocolXxx functions if the driver subsequently binds itself to an underlying NDIS driver.
4. Call `NdisIMAssociateMiniport` to inform the NDIS library about the driver’s miniport lower edge and protocol upper edge if the driver exports both MiniportXxx and ProtocolXxx functions.

`DriverEntry` can initialize spin locks for any globally shared resources that the intermediate driver allocates, such as structures and memory areas that the driver uses to track connections and to track sends in progress.

If `DriverEntry` fails to allocate any resources that the driver needs to carry out network I/O operations, it should release any previously allocated resources and return an appropriate error status. For example, if `DriverEntry` has called `NdisMInitializeWrapper`, it must call `NdisTerminateWrapper`.

An intermediate driver’s `DriverEntry` function can perform some global initialization steps. However, if an intermediate driver provides a ProtocolBindAdapter function, which opens and binds to an underlying device as described in Section 1.2, such a driver can defer allocating binding-related system resources to ProtocolBindAdapter. ProtocolBindAdapter performs the bind and allocates resources as needed for the device passed at `DeviceName`. `DriverEntry` must always initialize the wrapper and register as a miniport driver. It also must register as a protocol driver if the intermediate driver also exports a set of ProtocolXxx functions.

If the intermediate driver only exports to NDIS a set of MiniportXxx functions, it registers only those functions with the NDIS library, as described next.

Built on Wednesday, June 28, 2000
1.0 NDIS Intermediate Drivers

1.1.1 Registering as an NDIS Intermediate Driver

An NDIS intermediate driver must register its Miniport\texttt{Xxx} functions and its Protocol\texttt{Xxx} operations, if any, with NDIS in the context of its \texttt{DriverEntry} function. To register its Miniport\texttt{Xxx} functions, an intermediate driver must make call \texttt{NdisIMRegisterLayeredMiniport}. This call exports the intermediate driver’s Miniport\texttt{Xxx} functions. Note that the intermediate driver controls when its virtual NIC is initialized and, thus, when the driver is ready to accept sends and requests on that NIC.

NDIS then calls the intermediate driver’s Miniport\texttt{Initialize} function in the context of \texttt{NdisIMInitializeDeviceInstance} for the specified virtual NIC. If the intermediate driver exports more than one virtual NIC, the driver must call \texttt{NdisIMInitializeDeviceInstance} for each NIC that it makes available for network requests. An intermediate driver could use this capability to make more or fewer of its virtual NICs available, depending on network traffic.

If the NDIS intermediate driver also exports a set of Protocol\texttt{Xxx} functions, it must also register these functions with the NDIS library by calling \texttt{NdisRegisterProtocol}.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

1.1.1.1 Registering an Intermediate Driver as a Miniport

An intermediate driver calls \texttt{NdisIMRegisterLayeredMiniport} to export its Miniport\texttt{Xxx} functions. \texttt{NdisIMRegisterLayeredMiniport} is declared as follows:

\begin{verbatim}
NDIS_STATUS
NdisIMRegisterLayeredMiniport(
    IN NDIS_HANDLE NdisWrapperHandle,
    IN PNDIS_MINIPORT_CHARACTERISTICS MiniportCharacteristics,
    IN UINT CharacteristicsLength,
    OUT PNDIS_HANDLE DriverHandle
);
\end{verbatim}

The \texttt{DriverHandle} returned by \texttt{NdisIMRegisterLayeredMiniport} must be retained by the intermediate driver and input to NDIS when the driver calls \texttt{NdisIMInitializeDeviceInstance} to request a call to the intermediate driver’s Miniport\texttt{Initialize} function, which then initializes a virtual NIC. An intermediate driver calls \texttt{NdisIMInitializeDeviceInstance} after it either has successfully bound to one or more underlying NIC drivers or has layered itself above the driver of a nonNIC device so the intermediate driver is ready to initialize its miniport component to accept I/O requests on its virtual NIC.

\texttt{NdisWrapperHandle} was returned by a previous call to \texttt{NdisMInitializeWrapper}.

The intermediate driver must:

1. Zero-initialize a structure of type NDIS MINIPORT CHARACTERISTICS with
1. Zero-initialize a structure of type NDIS_MINIPORT_CHARACTERISTICS with NdisZeroMemory.

2. Store the addresses of the mandatory MiniportXxx functions, as well as any optional MiniportXxx functions the driver exports. Any other MiniportXxx entry points must be set to NULL.

While valid major versions are 0x03, 0x04, or 0x05 for other types of NDIS drivers, an intermediate driver must provide a major version of 4.0 and provide a version 4.0 or 5.0 MiniportCharacteristics structure whether it exports any new-for-V4.0 and/or new-for-V5.0 MiniportXxx functions.

The following entries in MiniportCharacteristics must be set to a valid MiniportXxx function address if the driver exports the function or to NULL if the driver does not export the function:

**HaltHandler**
This function is called by NDIS—for instance, if the underlying NIC has timed out and NDIS has halted the NIC driver or perhaps because the operating system is performing a controlled shutdown of the system.

**InitializeHandler**
This function is called as a result of the intermediate driver calling NdisIMInitializeDeviceInstance to initialize its miniport operations for the virtual NIC being initialized.

**QueryInformationHandler**
This function receives OID_XXX requests originating from or passed through by an overlying driver that has called NdisRequest with a request type of NdisRequestQueryInformation.

**ResetHandler**
NDIS can call an intermediate driver’s MiniportReset function at the behest of a higher protocol driver that has called NdisReset. Typically, however, a protocol driver does not initiate resets. Generally, NDIS initiates a reset of an underlying NIC driver and calls an intermediate driver’s ProtocolStatus and ProtocolStatusComplete functions to inform such an intermediate driver that the underlying miniport is resetting its NIC.

**SetInformationHandler**
This function processes OID_XXX requests made by or passed through an overlying driver that has called NdisRequest with a request type of NdisRequestSetInformation.

**SendHandler**
NDIS calls this function to transmit a single packet to the underlying NIC (or device) driver. A MiniportSend function (or a MiniportWanSend function) is required if the intermediate driver does not supply a MiniportSendPackets function. A MiniportSendPackets function should always be provided at SendPacketsHandler, rather than this handler, unless the intermediate driver will always be layered between drivers that transmit a single packet at a time or will bind itself to underlying WAN NIC driver(s). See Section 1.6 for more discussion of this topic.

**SendPacketsHandler**
This function receives an array of one or more pointers to packet descriptors specifying packets for transmission over the network. Every intermediate driver should supply a MiniportSendPackets function, rather than supplying a MiniportSend function, unless the intermediate driver will bind itself to underlying WAN NIC driver(s) and must supply a MiniportWanSend function. Otherwise, a MiniportSendPackets function provides the best performance whether the intermediate driver is layered over a NIC driver that can transmit several packets at a time or only one packet at a time and whether it is layered under a protocol driver that sends one packet at a time or several packets at a time. See Section 1.6 for a more complete discussion.
TransferDataHandler
This function is called to transfer the remaining part of a received packet that was not previously indicated up in the lookahead buffer passed by the intermediate driver to NdisM\*xxIndicateReceive. This indicated packet can be a converted packet previously received at the intermediate driver’s ProtocolReceive function or ProtocolReceivePacket handler. This handler is required if the intermediate driver indicates received packets to its overlying driver by calling any medium-specific NdisM\*xxIndicateReceive function except for NdisMWanIndicateReceive. If the intermediate driver always indicates packets to the overlying driver by calling NdisMIndicateReceivePacket it need not provide a MiniportTransferData function.

ReturnPacketHandler
This function receives a returned packet descriptor that was previously indicated to a higher level driver by calling NdisMIndicateReceivePacket, thereby relinquishing control of the resources indicated to the higher level driver. After each such indication is consumed by the higher level driver, the intermediate-allocated packet descriptor and the resources it describes will be returned to the MiniportReturnPacket function. A MiniportReturnPacket function is not supplied if the intermediate driver always indicates packets up by calling a medium-specific NdisM\*xxIndicateReceive function or if it always sets the status in the OOB data block associated with each packet descriptor to NDIS_STATUS_RESOURCES before calling NdisMIndicateReceivePacket.

CheckForHangHandler
This function is called at an NDIS-determined or, alternatively, at an intermediate-driver-determined interval. If supplied, the MiniportCheckForHang function is called every 2 (or at a driver-requested interval) seconds. See the Network Driver Reference for more information about the MiniportCheckForHang function or Part 2 of this manual. Generally, an NDIS intermediate driver does not provide a MiniportCheckForHang function because such a driver has no way to determine if an underlying NIC is hung. An intermediate driver probably would provide this handler if it was layered over a non-NDIS driver whose state is inaccessible to NDIS.

The following miniport handler functions are never supplied by an intermediate driver, because such drivers do not manage interrupting devices, do not allocate buffers at raised IRQL, or in the case of the ReconfigureHandler, because NDIS currently does not call a MiniportReconfigure function:

- DisableInterruptHandler
- EnableInterruptHandler
- HandleInterruptHandler
- ISRHandler
- AllocateCompleteHandler
- ReconfigureHandler

1.1.1.2 Registering an Intermediate Driver as a Protocol

An intermediate driver registers its Protocol\*xx functions with NDIS by calling NdisRegisterProtocol, which is declared as follows:

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK
The \texttt{NdisRegisterProtocol} function is declared as follows:

\begin{verbatim}
VOID NdisRegisterProtocol(
    OUT PNDIS_STATUS Status,
    OUT PNDIS_HANDLE NdisProtocolHandle,
    IN NDIS_PROTOCOL_CHARACTERISTICS ProtocolCharacteristics,
    IN UINT CharacteristicsLength
);
\end{verbatim}

Before calling \texttt{NdisRegisterProtocol}, the intermediate driver must:

1. Zero a structure of type \texttt{NDIS_PROTOCOL\_CHARACTERISTICS}. An intermediate driver can use a version 4.0 or 5.0 \texttt{ProtocolCharacteristics} structure. Protocol drivers must be PnP-ready. NDIS therefore no longer supports version 3.0 protocol drivers.
2. Store the addresses of the mandatory \texttt{ProtocolXxx} functions, as well as any optional \texttt{ProtocolXxx} functions that the driver supports.

The \texttt{NdisProtocolHandle} returned by this call is opaque to an intermediate driver. The handle must be retained by the intermediate driver and provided as an input parameter in future calls made by the protocol part of the intermediate driver to NDIS—for instance, to open underlying adapter.

\textbf{Registering the ProtocolXxx Functions of an Intermediate Driver with a Connectionless Lower Edge}

The possible optional and required protocol handler functions that an intermediate driver with a connectionless lower edge can export are listed below:

\begin{description}
\item[\texttt{BindAdapterHandler}] This is a required function. NDIS calls this function to request that the intermediate driver bind to an underlying NIC or virtual NIC whose name is passed as a parameter to this handler. See Section 1.2 for more information on dynamic binding.
\item[\texttt{UnbindAdapterHandler}] This is a required function. ProtocolUnbindAdapter is called by NDIS to close a binding to the underlying NIC or virtual NIC whose name is passed to this function. ProtocolUnbindAdapter calls \texttt{NdisCloseAdapter} and deallocates resources when the binding is successfully closed.
\item[\texttt{OpenAdapterCompleteHandler}] This is a required function. If an intermediate driver’s call to \texttt{NdisOpenAdapter} returns \texttt{NDIS\_STATUS\_PENDING}, \texttt{ProtocolOpenAdapterComplete} is subsequently called to complete the binding.
\item[\texttt{CloseAdapterCompleteHandler}] This is a required function. If an intermediate driver’s call to \texttt{NdisCloseAdapter} returns \texttt{NDIS\_STATUS\_PENDING}, \texttt{ProtocolCloseAdapterComplete} is subsequently called to complete the unbinding.
\item[\texttt{ReceiveHandler}] This is a required function. ProtocolReceive is called with a pointer to a lookahead buffer containing data received over the network. If this buffer contains less than the full net packet, ProtocolReceive calls \texttt{NdisTransferData} with a packet descriptor to get the remainder of the net packet. If the underlying driver calls \texttt{NdisMIndicateReceivePacket} to indicate receives, the lookahead buffer passed to ProtocolReceive will always contain a full net packet.
\end{description}
This is an optional function. A ProtocolReceivePacket function is provided if the intermediate driver will be layered over any NIC driver that indicates an array of pointer(s) to one or more packet descriptors or that supplies out-of-band data with its receive indications by calling `NdisMIndicateReceivePacket`. If a developer is unsure of the environment in which the intermediate driver will execute, this function should be provided because the intermediate driver will achieve better performance over any underlying NIC driver that makes multipacket receive indications.

**ReceiveCompleteHandler**

This is a required function. ProtocolReceiveComplete is called when any packets previously indicated to ProtocolReceive can be postprocessed.

**TransferDataCompleteHandler**

This is a required function if the ProtocolReceive function ever calls `NdisTransferData`. If a previous call to `NdisTransferData` to copy the remainder of a received packet returned NDIS_STATUS_PENDING, ProtocolTransferDataComplete is called when the transfer operation is done.

**ResetCompleteHandler**

This is a required function. ProtocolResetComplete is called when a reset operation, begun with a call to `NdisReset` that returned NDIS_STATUS_PENDING, is done. Usually, intermediate drivers do not call `NdisReset`, but the drivers above them might, so an intermediate driver could forward such a reset request to the underlying NDIS driver.

**RequestCompleteHandler**

This is a required function. ProtocolRequestComplete is called upon completion of a query/set operation, begun with a call to `NdisRequest` that returned NDIS_STATUS_PENDING, is done.

**SendCompleteHandler**

This is a required function. ProtocolSendComplete is called for each packet transmitted with a call to `NdisSend` that returned NDIS_STATUS_PENDING as the status of the send operation. If an array of packets is sent by calling `NdisSendPackets`, ProtocolSendComplete is called once for each packet passed to `NdisSendPackets`. The intermediate driver can determine the status of a send operation that calls `NdisSendPackets` only from the status argument input to ProtocolSendComplete.

**StatusHandler**

This is a required function. ProtocolStatus is called by NDIS with status notifications initiated by an underlying NIC driver.

**StatusCompleteHandler**

This is a required function. ProtocolStatusComplete is called by NDIS to indicate that a status change, previously indicated to ProtocolStatus, is now complete.

**PnPEventHandler**

This is a required function. NDIS calls ProtocolPnPEvent to indicate a Plug and Play event or a Power Management event. See Section 1.7 for more information.

**UnloadHandler**

This is an optional function. NDIS calls ProtocolUnload in response to a user request to uninstall an intermediate driver. NDIS calls ProtocolUnload after calling ProtocolUnbindAdapter once for each bound adapter. ProtocolUnload performs driver-determined clean-up operations.

---

**Registering the ProtocolXxx Functions of an Intermediate Driver with a Connection-Oriented Lower Edge**
An intermediate driver with a connection-oriented lower edge must register the following protocol handler functions that are common to both connectionless and connection-oriented miniports:

- BindHandler
- UnbindHandler
- OpenAdapterCompleteHandler
- CloseAdapterCompleteHandler
- ReceiveCompleteHandler
- ResetCompleteHandler
- RequestCompleteHandler
- StatusCompleteHandler
- PnPEventHandler

These functions are summarized above in the previous subsection.

An intermediate driver with a connection-oriented lower edge must also register the following connection-oriented protocol functions:

**CoSendCompleteHandler**
This is a required function. ProtocolCoSendComplete is called once for each packet passed to NdisCoSendPackets. The intermediate driver can determine the status of a send operation that calls NdisCoSendPackets only from the status argument input to ProtocolCoSendComplete.

**CoStatusHandler**
This is a required function. ProtocolCoStatus is called by NDIS with status notifications initiated by an underlying NIC driver.

**CoReceivePacketHandler**
This is a required function. When a bound connection-oriented NIC driver or integrated miniport call manager (MCM) indicates an array of pointer(s) by calling NdisMCoIndicateReceivePacket, NDIS calls the intermediate driver’s ProtocolCoReceivePacketHandler.

**CoAfRegisterNotifyHandler**
If the intermediate driver is a connection-oriented client that uses the call manager services of a call manager or an MCM driver, it must register a ProtocolCoAfRegisterNotify function. ProtocolCoAfRegisterNotify determines whether the intermediate driver can use the services of a call manager or MCM that has advertised its services by registering an address family.

1.2 Dynamic Binding in an Intermediate Driver
An intermediate driver must support dynamic binding to underlying NICs by providing both a ProtocolBindAdapter and a ProtocolUnbindAdapter function. NDIS calls the ProtocolBindAdapter function of any intermediate driver that can bind to a NIC when that NIC becomes available.
1.0 NDIS Intermediate Drivers

```c
OUT PNDIS_STATUS Status,
IN NDIS_HANDLE BindContext,
IN PNDIS_STRING DeviceName,
IN PVOID SystemSpecific1,
IN PVOID SystemSpecific2
);
```

The handle at `BindContext` represents NDIS’ context for the bind request. The intermediate driver must retain this handle and pass it back to NDIS as a parameter to `NdisCompleteBindAdapter` when the intermediate driver has completed its binding-related activities and is ready to accept transmit requests.

Binding-time actions include allocating and initializing a NIC-specific context area for the binding, and calling `NdisOpenAdapter` to bind to the adapter passed at `DeviceName`. `DeviceName` can refer to a NIC managed by an underlying NIC driver, or it can be the name of a virtual NIC exported by an intermediate NDIS driver that is layered between the called intermediate driver and the NIC driver managing the adapter to which transmit requests are directed. Usually, only one intermediate NDIS driver is layered over a NIC driver, perhaps to translate between the format of the medium supported by an overlying legacy protocol driver and the format of the medium supported by an underlying NIC driver.

Note that the intermediate driver must pass to `NdisOpenAdapter` the same `DeviceName` (a pointer to a buffered Unicode string) that was previously passed to the driver’s `ProtocolBindAdapter` function. The driver must not copy the pointer and pass a copy of the pointer to `NdisOpenAdapter`.

The intermediate driver can store the `BindContext` in the allocated binding-related context area or in another driver-accessible location. The `BindContext` value must be stored if `NdisOpenAdapter` can return NDIS_STATUS_PENDING. In this case, the intermediate driver cannot call `NdisCompleteBindAdapter` until the open-adapter operation is complete and the intermediate driver is called at its `ProtocolOpenAdapterComplete` function. `BindContext` must then be retrieved from a known location and passed by `ProtocolOpenAdapterComplete` to `NdisCompleteBindAdapter`.

`SystemSpecific1` points to a registry path if the intermediate driver stores adapter-specific information in the registry. This value is passed to `NdisOpenProtocolConfiguration` to obtain a handle used to read and possibly write such information.

`SystemSpecific2` is reserved for system use.

If an intermediate driver converts incoming packets from one medium format to another, `ProtocolBindAdapter` can allocate a pool of packet descriptors and buffer descriptors that it will need on a per-binding basis. See Section 1.3 for information about the requirement to allocate and manage packets. Also, if an intermediate driver receives incoming data only at its `Protocol(Co)Receive` function, it should allocate packet pool and buffer pool to copy received data.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

1.2.1 Opening an Adapter Underlying an Intermediate Driver
ProtocolBindAdapter uses the value at `DeviceName` to open an underlying NIC or virtual NIC, thereby establishing a binding to the underlying NIC driver. It can also read additional configuration information that it requires from the registry. `NdisOpenProtocolConfiguration` is called to obtain a handle to the registry key where the intermediate driver stores any adapter-specific information. The intermediate driver can call `NdisOpenConfigurationKeyByIndex` or `NdisOpenConfigurationKeyByName` to open and get a handle to a subkey below the key opened with `NdisOpenProtocolConfiguration`. The intermediate driver can then call `NdisRead(Write)Configuration` functions to read and write registry information from an opened key or subkey. The `NdisRead(Write)Configuration` functions are described in the *Network Driver Reference*.

Typically, ProtocolBindAdapter uses the context area that it allocates to represent its binding to `DeviceName` as storage for any binding-specific information associated with the bound adapter.

Binding is accomplished by calling `NdisOpenAdapter`, which is declared as follows:

```c
VOID
NdisOpenAdapter(
    OUT PNDIS_STATUS Status,
    OUT PNDIS_STATUS OpenErrorStatus,
    OUT PNDIS_HANDLE NdisBindingHandle,
    OUT PUINT 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
);```

An intermediate driver passes a handle at `ProtocolBindingContext` that represents the binding-specific context area that it allocated and initialized. NDIS will pass this context back to the intermediate driver in future calls pertaining to the binding — for instance, in calls to Protocol(Co)Receive or Protocol(Co)Status. Similarly, NDIS will pass a handle at `NdisBindingHandle` to the intermediate driver when `NdisOpenAdapter` returns. This handle must be retained by the intermediate driver, usually in its binding-specific context area. The intermediate driver passes this handle to NDIS in future calls relating to this binding, such as calls to `NdisSend` or `Ndis(Co)SendPackets`.

ProtocolBindAdapter also passes the type(s) of medium(s) that it supports at `MediumArray`. If `NdisOpenAdapter` is successful, the underlying NIC driver will select a medium that it can support and return the index of the selected medium from `MediumArray` at `SelectedMediumIndex`.

ProtocolBindAdapter passes the value returned from a previous successful call to `NdisRegisterProtocol` at `NdisProtocolHandle`.

If `NdisOpenAdapter` returns an error, the intermediate driver should deallocate the memory used for the context area and release any other resources that it allocated for the binding. Typically, ProtocolBindAdapter logs any failed binding operations with appropriate descriptive information by calling `NdisWriteErrorLogEntry`.

Built on Wednesday, June 28, 2000
1.2.2 Initializing as a Miniport

ProtocolBindAdapter, after it has successfully opened an underlying NIC and is ready to accept requests and sends on its virtual NIC or NICs, calls NdisIMInitializeDeviceInstance one or more times to request initialization of one or more NICs. NdisIMInitializeDeviceInstance calls the intermediate driver’s MiniportInitialize function to perform the initialization of the specified virtual NIC. After MiniportInitialize returns, overlying NDIS drivers can bind to the intermediate driver’s virtual NIC(s).

MiniportInitialize must allocate and initialize a virtual-NIC-specific context area. As part of initialization, MiniportInitialize must call NdisMSetAttributesEx and pass a handle to that context area. NDIS will pass this context handle in subsequent calls to the MiniportXxx functions. MiniportInitialize also must set the NDIS_ATTRIBUTE_INTERMEDIATE_DRIVER flag the AttributeFlags passed to NdisMSetAttributesEx. An intermediate driver sets the NDIS_ATTRIBUTE_INTERMEDIATE_DRIVER flag to identify its driver type to NDIS.

In addition, MiniportInitialize must set the NDIS_ATTRIBUTE_IGNORE_PACKET_TIMEOUT and NDIS_ATTRIBUTE_IGNORE_REQUEST_TIMEOUT flags in AttributeFlags passed to NdisMSetAttributesEx if it does not want NDIS to call the MiniportCheckForHang (or MiniportReset) function whenever NDIS times out sends and requests that it is holding queued to the intermediate driver. Setting the timeout flags instructs NDIS that the intermediate driver will be responsible for timing out its own virtual NIC. Since the intermediate driver does not control the underlying NIC and therefore has no control over how long it takes to complete pending sends and requests, it generally does not provide a MiniportCheckForHang function nor does it time-out its virtual NIC.

However, if the intermediate driver registers an entry point at CheckForHangHandler, does not request that NDIS ignore packet and request timeouts, and does not change the time-out interval, its MiniportCheckForHang function will be called, by default, every 2 seconds. If MiniportCheckForHang returns TRUE, the MiniportReset function will be called. If the driver supplies a MiniportCheckForHang function, it can change the default 2-second interval by explicitly specifying a different TimeInSeconds when it calls NdisMSetAttributesEx.

An intermediate driver must operate as a deserialized driver and must register itself as such by setting the NDIS_ATTRIBUTE_DESERIALIZE flag in the AttributeFlags parameter passed to NdisMSetAttributesEx. A deserialized driver serializes the operation of its MiniportXxx functions and queues internally all incoming send packets rather than relying on NDIS to maintain its send queue. (See Part 2, Chapter 4, Section 4.5 for more information on deserialized drivers.)

An intermediate driver must also set the NDIS_ATTRIBUTE_NO_HALT_ON_SUSPEND flag in the AttributeFlags parameter passed to NdisMSetAttributesEx. Setting this flag prevents NDIS from halting the driver before the underlying miniport transitions to a low-power state.

An intermediate driver should make sure that the state information it maintains is properly initialized. If the driver requires send-related resources—for instance, new packet descriptors for packets that
MiniportSend or MiniportSendPackets will transmit to the next lower layer—the packet pool can be allocated at this time if it was not done in ProtocolBindAdapter before calling NdisIMInitializeDeviceInstance.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

1.2.3 Intermediate Driver Query and Set Operations

After it has successfully bound to an underlying NIC and initialized its virtual NIC(s), an intermediate driver queries the operating characteristics of the underlying NIC driver and sets its internal state, as well as negotiates such parameters as lookahead buffer size for the binding with the underlying NIC driver, if appropriate. An intermediate driver with a connectionless lower edge accomplishes this by calling NdisRequest. An intermediate driver with a connection-oriented lower edge accomplishes this by calling NdisCoRequest.

An intermediate driver can also receive query and set requests from protocol drivers at its MiniportQueryInformation and MiniportSetInformation functions, respectively, and either respond to those requests or pass them down to the underlying driver.

The Network Driver Reference contains information about all the general, connection-oriented, non-media-specific OIDs and about required media-specific OIDs of interest to intermediate driver developers. The following mentions typical and commonly used general category OIDs, connection-oriented OIDs, and a few media-specific OIDs.

Issuing Set and Query Requests

An intermediate driver with a connectionless lower edge typically issues an OID_GEN_MAXIMUM_FRAME_SIZE request to query the maximum frame size, in bytes, supported by an underlying NIC driver. The size returned does not include the header.

An intermediate driver with a connectionless lower edge can query a binding with OID_GEN_MAXIMUM_TOTAL_SIZE to determine the largest packet an underlying NIC driver can accommodate on the NIC that it manages. The intermediate driver must always set up send packets that conform to this size. It is an error for an overlying driver to submit a larger packet than a NIC driver to which it is bound can support.

An intermediate driver with a connectionless lower edge can query the size of the lookahead data buffer. The OID used is OID_GEN_CURRENT_LOOKAHEAD. If the intermediate driver issues this as a query, NDIS returns the current lookahead buffer size for the given binding to the underlying NIC driver. If the intermediate driver makes a set request, it indicates its preferred lookahead buffer size but the intermediate driver is not assured that the underlying NIC driver will conform to this.

An intermediate driver with a connectionless lower edge queries the underlying NIC driver for its link speed with OID_GEN_LINK_SPEED and uses the response to set any internal time-out values that it maintains. An intermediate driver with a connection-oriented lower edge queries the underlying NIC
driver for its link speed with OID_GEN_CO_LINK_SPEED and can also set the link speed of the underlying NIC driver with OID_GEN_CO_LINK_SPEED.

If an intermediate driver is bound to a WAN NIC driver, it cannot determine the link speed until it receives a line-up indication, indicating that a connection has been established between the local node and a remote node. See Indications that WAN miniport drivers make in Chapter 8 of Part 2, for a description of the line-up indication.

An intermediate driver must also determine the set of operating characteristics of the underlying NIC driver. An intermediate driver with a connectionless lower edge accomplishes this by issuing an OID_GEN_MAC_OPTIONS. An intermediate driver with a connection-oriented lower edge accomplishes this by issuing an OID_GEN_CO_MAC_OPTIONS.

An intermediate driver with a connectionless lower edge usually issues an OID_GEN_MAXIMUM_SEND_PACKETS query, particularly if the intermediate driver exports a MiniportSendPackets function. Such an intermediate driver can propagate the returned value when it, in turn, responds to an OID_GEN_MAXIMUM_SEND_PACKETS query from a still higher level driver.

An intermediate driver also can query the medium-dependent current address with a medium-specific OID. For example, an intermediate driver with a connectionless lower edge might issue an OID_WAN_CURRENT_ADDRESS, OID_802_3_CURRENT_ADDRESS, OID_802_5_CURRENT_ADDRESS, or OID_FDDI_LONG_CURRENT_ADDRESS. An intermediate driver with a connection-oriented lower edge might issue an OID_ATM_HW_CURRENT_ADDRESS.

If needed, the intermediate driver issues a set request to inform NDIS of its operating characteristics. An intermediate driver with a connectionless lower edge accomplishes this by passing OID_GEN_PROTOCOL_OPTIONS to NdisRequest. An intermediate driver with a connection-oriented lower edge accomplishes this by passing OID_GEN_CO_PROTOCOL_OPTIONS to NdisCoRequest.

An intermediate driver bound to a WAN-capable NIC must also make the following set-information requests:

- OID_WAN_PROTOCOL_TYPE to inform the underlying NIC driver of the protocol’s type. The type is supplied as a single-byte, network-level protocol identifier.
- OID_WAN_HEADER_FORMAT to inform the underlying NIC driver of the header format of the packets it sends.

Responding to Sets and Queries

Because an intermediate NDIS driver is bound to by an overlying NDIS driver, it can also receive queries and sets at its MiniportQueryInformation and MiniportSetInformation functions. In some cases, the intermediate driver just passes such requests through to the underlying NIC driver. Otherwise, it can respond to these queries and sets as appropriate to the medium it exports at its upper edge. Note that an intermediate driver must always pass through any OID_PNP_XXX requests that it receives from an overlying NDIS driver to the underlying miniport driver.
Typically, the general OIDs received by an intermediate driver will be the same or similar to those that the intermediate driver makes to the underlying NIC driver. The medium-specific OIDs received by an intermediate driver will be the type of the medium expected by the overlying driver.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

1.2.4 Registering an Intermediate Driver as a Connection-Oriented Client

An intermediate driver with a connection-oriented lower edge must register as a connection-oriented client. A connection-oriented client uses the call-setup and tear-down services of a call manager or integrated miniport call manager (MCM). A connection-oriented client also uses the send and receive capabilities of a connection-oriented miniport or an MCM to send and receive data. (For more information on connection-oriented communication, see Part 1, Chapter 4).

When a call manager or MCM, from its ProtocolBindAdapter function, calls Ndis(M) CmRegisterAddressFamily to register an address family, NDIS calls the ProtocolCoRegisterAfNotify function of all protocol drivers on the binding. If an intermediate driver registered a ProtocolCoAfRegisterNotify function when it registered as a protocol, NDIS calls the intermediate driver’s ProtocolCoAfRegisterNotify function.

If ProtocolCoAfRegisterNotify determines that the intermediate driver can use the services of the call manager or MCM that registered the address family, it allocates a per-AF context area for the client and calls NdisClOpenAddressFamily. With the call to NdisClOpenAddressFamily, an intermediate drivers registers a set of client-supplied functions

NdisClOpenAddressFamily is declared as follows:

```c
typedef NDIS_STATUS

NdisClOpenAddressFamily(
    IN NDIS_HANDLE NdisBindingHandle,
    IN PCO_ADDRESS_FAMILY AddressFamily,
    IN NDIS_HANDLE ProtocolAfContext,
    IN PNDIS_CLIENT_CHARACTERISTICS ClCharacteristics,
    IN UINT SizeOfClCharacteristics,
    OUT PNDIS_HANDLE NdisAfHandle
);
```

Before calling NdisClOpenAddressFamily, the intermediate driver must:

1. Zero a structure of type NDIS_CLIENT_CHARACTERISTICS. The current version of the ClCharacteristics structure is 5.0.
2. Store the addresses of the ProtocolXxx client functions that that driver supports.

The NdisAfHandle returned by this call is opaque to an intermediate driver. The handle must be retained by the intermediate driver and provided as an input parameter in future calls made by the protocol part of the intermediate driver to NDIS—for instance, to register a SAP with NdisClRegisterSap.
The client functions that an intermediate driver must register with `NdisClOpenAddressFamily` are listed below:

- **ClCreateVcHandler**
  Specifies the entry point of the caller’s ProtocolCoCreateVc function.

- **ClDeleteVcHandler**
  Specifies the entry point of the caller’s ProtocolCoDeleteVc function.

- **ClRequestHandler**
  Specifies the entry point of the caller’s ProtocolCoRequest function.

- **ClRequestCompleteHandler**
  Specifies the entry point of the caller’s ProtocolCoRequestComplete function.

- **ClOpenAfCompleteHandler**
  Specifies the entry point of the caller’s ProtocolClOpenAfComplete function.

- **ClCloseAfCompleteHandler**
  Specifies the entry point of the caller’s ProtocolClCloseAfComplete function.

- **ClRegisterSapCompleteHandler**
  Specifies the entry point of the caller’s ProtocolClRegisterSapComplete function. A client uses this function to accept incoming calls from remote machines.

- **ClDeregisterSapCompleteHandler**
  Specifies the entry point of the caller’s ProtocolClDeregisterSapComplete function.

- **ClMakeCallCompleteHandler**
  Specifies the entry point of the caller’s ProtocolClMakeCallComplete function. A client uses this function to make outgoing calls to remote machines.

- **ClModifyCallQoSCompleteHandler**
  Specifies the entry point of the caller’s ProtocolClModifyCallQoSComplete function. A client uses this function to make changes in the quality of service on an established VC dynamically or to negotiate with the call manager to establish the QoS when setting up an incoming call.

- **ClCloseCallCompleteHandler**
  Specifies the entry point of the caller’s ProtocolClCloseCallComplete function.

- **ClAddPartyCompleteHandler**
  Specifies the entry point of the caller’s ProtocolClAddPartyComplete function. A client uses this function to establish point-to-multipoint VCs for outgoing calls to remote machines.

- **ClDropPartyCompleteHandler**
  Specifies the entry point of the caller’s ProtocolClDropPartyComplete function.

- **ClIncomingCallHandler**
  Specifies the entry point of the caller’s ProtocolClIncomingCall function. A client uses this function to accept incoming calls from remote machines.

- **ClIncomingCallQoSChangeHandler**
  Specifies the entry point of the caller’s ProtocolClIncomingCallQoSChange function. A client uses this function to accept incoming calls from remote machines on which the sending client can change the QoS dynamically.

- **ClIncomingCloseCallHandler**
  Specifies the entry point of the caller’s ProtocolClIncomingCloseCall function.

- **ClIncomingDropPartyHandler**
  Specifies the entry point of the caller’s ProtocolClIncomingDropParty function.

- **ClCallConnectedHandler**
  Specifies the entry point of the caller’s ProtocolClCallConnected function. A client uses this function to accept incoming calls from remote machines.
The intermediate driver must set every CIxcli member in the NDIS_CLIENT_CHARACTERISTICS structure to a caller-supplied ProtocolCI/CoXxx function when it calls NdisClOpenAddressFamily, even if it does not support incoming calls, outgoing calls, or point-to-multipoint connections. For whatever subset of connection-oriented functionality that such an intermediate driver does not support, its ProtocolCI/CoXxx functions simply return NDIS_STATUS_NOT_SUPPORTED.

1.3 Intermediate Driver Packet Management

An intermediate driver receives a packet descriptor with one or more chained buffers of data from a higher level driver to send over the network. The intermediate driver can repackage the data to be transmitted in a fresh packet descriptor and simply pass the packet through to the underlying driver by calling NdisSend or NdisSendPackets if the driver has a connectionless lower edge or NdisCoSendPackets if the driver has a connection-oriented lower edge. Alternatively the intermediate driver can take some actions to modify either the contents of the chained buffer(s) or perhaps the ordering or timing of the incoming packet(s) relative to other transmissions. Even if the intermediate driver does nothing other than simply pass on incoming packet(s) — for instance, if it simply counts the packets — it must allocate a fresh packet descriptor and manage some or all of a new packet structure.

Every intermediate driver must allocate its own packet descriptors to replace those of the overlying driver. If an intermediate driver converts the packet from one format to another, it also can allocate buffer descriptors to map intermediate-allocated buffers into which the converted data is copied. If there is OOB data associated with the packet descriptor being copied, this data can be copied to the new OOB block associated with the intermediate-allocated packet descriptor, using the macro NDIS_OOB_DATA_FROM_PACKET, to obtain a pointer to the OOB data area and, then, calling NdisMoveMemory to move the contents into the OOB area associated with the new packet descriptor. Alternatively, such an intermediate driver can use the NDIS_GET_PACKET_XXX and NDIS_SET_PACKET_XXX macros to read specific items from the OOB data associated with the old packet descriptor and to write the OOB data for the new packet descriptor.

Packet descriptors must be allocated by calling the following NDIS functions:

1. NdisAllocatePacketPool or NdisAllocatePacketPoolEx to allocate and initialize a block of nonpaged pool for a caller-specified number of fixed-size packet descriptors
2. NdisAllocatePacket to allocate a packet descriptor from the pool allocated by NdisAllocatePacketPool(Ex)

Depending on the purpose of the intermediate driver, such a driver can repackage buffers chained to incoming packet descriptors. For instance, an intermediate driver allocates buffer pool and repackage incoming packet data in the following circumstances:

- The intermediate driver receives a larger data buffer from an overlying protocol driver than can be sent in a single buffer over the underlying medium. Consequently, the intermediate driver
be sent in a single buffer over the underlying medium. Consequently, the intermediate driver must divide the incoming data into smaller buffers.

- The intermediate driver changes the length of the incoming packet data by compressing or encrypting the data before forwarding each send to the underlying driver.

Such buffers can be allocated by calling the following NDIS functions:

1. NdisAllocateBufferPool to obtain a handle with which to allocate buffer descriptors
2. NdisAllocateMemory or NdisAllocateMemoryWithTag to allocate a buffer
3. NdisAllocateBuffer to allocate and set up a buffer descriptor to map the buffer allocated by calling NdisAllocateMemoryWithTag and to chain to the packet descriptor allocated by calling NdisAllocatePacket

Buffer descriptors are chained to a packet descriptor by calling NdisChainBufferAtBack or NdisChainBufferAtFront. The virtual address and the length of the buffer returned by NdisAllocateMemoryWithTag are passed in the call to NdisAllocateBuffer to initialize the buffer descriptor that maps the buffer.

Packet descriptors to meet typical needs can be allocated as needed, at driver initialization time, or in the ProtocolBindAdapter function. An intermediate driver developer can, if necessary and for performance reasons, allocate a number of packet descriptors and possibly buffers mapped by buffer descriptors at initialization time so that ProtocolReceive has preallocated resources into which to copy incoming data for indicating to higher level driver and so that MiniportSend or MiniportSendPackets has available descriptors (and possibly buffers) for passing on incoming send packets to the next lower driver.

If an intermediate driver copies send data or received data to a new buffer or buffers, and the length of actual data in the last buffer is less than the allocated length of the buffer, the intermediate driver calls NdisAdjustBufferLength to adjust the buffer descriptor to the actual length of the data. When the packet is returned to the intermediate driver, it should again make this call to readjust the length to the full length of its buffer.

An intermediate driver with a connectionless lower edge can either receive incoming data from an underlying NIC driver either as a complete packet, specified by a packet descriptor of type NDIS_PACKET, at its ProtocolReceivePacket function or have the data indicated to its ProtocolReceive function and copy it into an intermediate-driver-supplied packet. An intermediate driver with a connection-oriented lower edge always receives incoming data from an underlying NIC driver as a complete packet at its ProtocolCoReceivePacket function.

An intermediate driver can retain ownership of a received packet in the following circumstances:

- When an intermediate driver with a connectionless lower edge has a complete packet indicated to its ProtocolReceivePacket function
- When an intermediate driver with a connection-oriented lower edge has a packet indicated to its ProtocolCoReceivePacket function with the Status member of NDIS_PACKET_OOB_DATA set to anything other than NDIS_STATUS_RESOURCES

In these cases, the driver can retain ownership of the packet descriptor and the resources that it describes until the received data is consumed and return these resources to the underlying driver later by calling NdisReturnPackets. If ProtocolReceivePacket passes the resources that it received to a
higher level driver, it must, at a minimum, replace the input packet descriptor with one that the intermediate driver has allocated.

Depending on an intermediate driver’s purpose when it receives a full packet from an underlying driver, several packet-management strategies are possible. For example, it can do either of the following:

- Copy the buffered contents into an intermediate driver-allocated buffer mapped and chained to a fresh packet descriptor, return the input packet descriptor to the underlying driver, and then indicate the new packet to the higher level driver.
- Create a new packet descriptor and chain the buffers from the indicated packet descriptor to this new descriptor. This new packet descriptor can then be indicated up to the higher level driver. When this higher level driver returns the packet descriptor, the intermediate driver must unchain the buffers from its packet descriptor, chain them to the packet descriptor it originally received from the underlying driver, and return the original packet descriptor and the resources that it describes to the underlying driver.

Even if an intermediate driver with a connectionless lower edge has a ProtocolReceivePacket function, it must also have a ProtocolReceive function. NDIS calls ProtocolReceive whenever the underlying driver will not relinquish ownership of the resources it indicates with a packet descriptor, and the intermediate driver must copy the received data into its own buffers when such an indication occurs. If the underlying driver also indicates out-of-band data with receives, ProtocolReceive can call NdisGetReceivedPacket and NDIS_GET_ORIGINAL_PACKET to get the out-of-band data associated with the receive indication.

When an underlying connection-oriented NIC driver will not relinquish ownership of the resources that it indicates with a packet descriptor, it sets the Status member of the packet’s NDIS_PACKET_OOB_DATA to NDIS_STATUS_RESOURCES. The ProtocolCoReceivePacket function of the intermediate driver then must copy the received data into its own buffers.

**Reusing Packets**

As already mentioned, after NdisSend returns NDIS_STATUS_SUCCESS for a packet descriptor submitted by an intermediate driver, either the packet descriptor is returned to ProtocolSendComplete, or an intermediate-driver-allocated packet is returned to MiniportReturnPacket. The ownership of the packet descriptor and all the resources that it describes are returned to the intermediate driver. If a higher level driver supplied the buffers chained to such a returned packet descriptor, the intermediate driver should return these resources back to the allocating driver promptly.

If the intermediate driver originally allocated the packet descriptor and/or the chained buffers, it can reclaim its resources and use them in subsequent sends or use them for received data that is indicated to ProtocolReceive. It is more efficient for the intermediate driver to reinitialize and reuse the packet descriptors that it allocates, and to reuse any intermediate driver-allocated chained buffer descriptors and buffers, than to deallocate and then later reallocate these resources.

An intermediate driver reinitializes a packet descriptor by calling NdisReinitializePacket. However, the intermediate driver should first take care to remove any chained buffers and their buffer descriptors by calling NdisUnchainBufferAtXxx. Otherwise, since NdisReinitializePacket will
1.0 NDIS Intermediate Drivers

1.4 Restrictions on Intermediate Drivers

Previous sections have described specific actions an intermediate driver must follow to perform correctly, summarized as follows:

1. An intermediate driver must set the NDIS_ATTRIBUTE_INTERMEDIATE_DRIVER flag when it calls NdisMSetAttributesEx in its MiniportInitialize function. NDIS identifies that a driver is an intermediate type only through the presence of this flag and takes special steps to ensure that deferred actions, such as passing internally-queued send packets on to the intermediate driver, occur without deadlocks.

2. An intermediate driver must always, at a minimum, replace an incoming packet descriptor for a packet that it passes down to an underlying driver on a send or up to a higher level driver on a receive with a fresh packet descriptor, allocated by the intermediate driver. It must also then replace its own packet descriptor with the one that originally was associated with the packet when it was passed to the intermediate driver—for instance, when completing a send or completing a receive indication. The intermediate driver must return the resources that it has "borrowed" from a higher or lower driver promptly by returning that driver’s packet descriptor and the resources that it specifies.

3. An intermediate driver must follow the rules that apply to any deserialized miniport driver in particular, the following guideline:

   If any internal driver resources are shared between an intermediate driver’s send function and any other MiniportXxx function, that resource must be protected by a spin lock. The only exception to this is in the MiniportReset path, which is serialized by NDIS with respect to the MiniportSend or MiniportSendPackets function and all other MiniportXxx functions.

   As for any deserialized miniport driver, an intermediate driver that organizes its per-binding context area into discrete receive-specific, send-specific, and shared ranges, with only the shared resource(s) protected by spin lock(s), will exhibit far better performance than a driver that must overprotect its context area because send-specific, receive-specific, and shared variables are scattered throughout.

1.5 Receiving Data in an Intermediate Driver
This section describes how intermediate drivers with a connectionless lower edge receive data and how intermediate drivers with a connection-oriented lower edge receive data.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

### 1.5.1 Receiving Data in an Intermediate Driver with a Connectionless Lower Edge

Underlying connectionless NIC drivers can indicate packets in two ways:

- A NIC driver calls the non-filter-specific `NdisMIndicateReceivePacket`, passing a pointer to an array of pointer(s) to one or more packet descriptors, relinquishing ownership of the indicated packet resources to higher level drivers. When the higher level driver(s) have consumed the data, they return the packet descriptor(s) (and the resources they specify) to the NIC driver.

- A NIC driver calls a filter-specific `NdisMXxxIndicateReceive` function, passing a pointer to a lookahead buffer and the total size of the packet.

An intermediate driver with a connectionless lower edge must have a `ProtocolReceive` function, and it can also have a `ProtocolReceivePacket` function, depending on the environment in which it runs.

- `ProtocolReceive` is a required function for an intermediate driver with a connectionless lower edge. `ProtocolReceive` is passed a pointer to a lookahead buffer. If, after the connectionless intermediate driver examines the lookahead data, it determines that the packet is intended for one or more of its overlying drivers, it must copy the data into a previously allocated packet which it will indicate to its overlying driver(s). If the size of the lookahead buffer is less than the total size of the received packet, the intermediate driver must first call `NdisTransferData` in the context of `ProtocolReceive`, to copy the rest of the received data.

So that `ProtocolReceive` executes as quickly as possible, the intermediate driver should have packet descriptors, buffers, and buffer descriptors preallocated for this purpose. `ProtocolReceive` is usually called because the underlying driver called `NdisMXxxIndicateReceive`. However, `ProtocolReceive` is also called if the underlying NIC driver indicated the received data with `NdisMIndicateReceivePacket` but set the status in the OOB block for an indicated packet descriptor to NDIS_STATUS_RESOURCES. For such packets indicated with `NdisMIndicateReceivePacket`, but passed to `ProtocolReceive`, the size of the lookahead buffer is always equal to the total size of the packet. Consequently, the intermediate driver does not call `NdisTransferData` for such an indication to `ProtocolReceive`. However, if the underlying NIC driver also indicates OOB data, the intermediate driver must retrieve this information in `ProtocolReceive` by calling `NdisGetReceivedPacket` and `NDIS_GET_ORIGINAL_PACKET`.

- `ProtocolReceivePacket` is an optional function for an intermediate driver with a connectionless lower edge. `ProtocolReceivePacket` receives a pointer to packet descriptor that always describes a full network packet. If any underlying connectionless NIC might be a busmaster DMA device, its driver will most likely indicate received packets by calling the non-filter-specific `NdisMIndicateReceivePacket`, and the intermediate driver should have a
ProtocolReceivePacket function if it might bind itself to such an underlying NID driver. In addition, any underlying NIC driver that supports OOB data will most likely pass a packet descriptor on receives to NdisMIndicateReceivePacket so that the intermediate driver can access the OOB data associated with the packet descriptor.

ProtocolReceivePacket examines the packet and if it determines that the packet is intended for one or more overlying drivers, it can retain ownership of the packet descriptor by returning a nonzero value. If it does return a nonzero value, the intermediate driver must subsequently call NdisReturnPackets with a pointer to this packet descriptor. The driver must make this call for a particular packet descriptor as many times as the nonzero value it returned from ProtocolReceivePacket when it received that indication.

When the intermediate driver calls NdisReturnPackets this number of times, it relinquishes ownership of the packet descriptor and associated buffers back to the underlying driver that originally indicated the receive. Therefore, an intermediate driver should call NdisReturnPackets as quickly as possible.

If, on the other hand, an intermediate driver returns zero from ProtocolReceivePacket, it indicates that it is relinquishing the packet and associated resources immediately. This could occur, for instance, if the intermediate driver copies the indicated data into buffers of its own and processes the data internally before indicating up to still higher level drivers.

1.5.1.1 Implementing a ProtocolReceivePacket Handler in an Intermediate Driver

When an underlying NIC driver indicates an array of one or more packets, possibly with associated OOB data, by calling NdisMIndicateReceivePacket, NDIS will usually call a bound intermediate driver’s ProtocolReceivePacket with each packet descriptor, allowing the intermediate driver to retain the resources specified by the packet descriptor and to consume the data before returning it. Two kinds of NIC drivers typically call NdisMIndicateReceivePacket with an array of packets:

- A NIC driver managing a busmaster DMA adapter that is capable of receiving several network packets into a ring of buffers.
- A NIC driver that provides out-of-band data containing media-specific information, such as packet priority, to higher level drivers in the NDIS_PACKET_OOB_DATA block associated with the packet descriptor. Such a driver need not be a driver for a busmaster DMA device.

If an intermediate driver is aware that it is (or might be) bound to such a NIC driver, it should, as mentioned earlier, have a ProtocolReceivePacket function. This allows the intermediate driver to all of the following:

1. To receive a full network packet at every receive indication
2. To use NDIS macros to read the OOB data associated with the packet descriptor, rather than calling NdisGetReceivedPacket and NDIS_GET_ORIGINAL_PACKET to receive and copy the data.
3. To retain ownership of incoming packet descriptors and direct read-only access to the buffered data specified by these descriptors, possibly making copies of the data for its clients, and then, when done with each packet descriptor
4. To return the packet descriptor and the resources it describes, possibly along with other retained packet descriptors, with NdisReturnPackets.

Even when an intermediate driver provides a ProtocolReceivePacket handler, there are cases when a call by a NIC driver to NdisMIIndicateReceivePacket results in a call to an intermediate driver’s ProtocolReceive handler. Since a NIC driver temporarily relinquishes ownership of driver-allocated resources when it calls NdisMIIndicateReceivePacket, the underlying driver is dependent on the consumers of those packets to return them with NdisReturnPackets in a timely manner. Otherwise, such a NIC driver can run short of receive resources, such as receive buffer space in the NIC. When it does, the NIC driver writes a status of NDIS_STATUS_RESOURCES into the OOB block associated with a packet descriptor in the packet array it passes to NdisMIIndicateReceivePacket. Indicating a packet with this status causes NDIS to call the overlying driver’s ProtocolReceive function with such a packet and with any subsequent packets in the array, thus forcing the intermediate driver to copy the packet data rather than taking ownership.

If the intermediate driver requires the OOB data associated with a packet descriptor but is called at ProtocolReceive, it must call NdisGetReceivedPacket and NDIS_GET_ORIGINAL_PACKET to copy the media-specific information into an intermediate-driver-supplied buffer and possibly the TimeSent and the TimeReceived if the underlying NIC driver provides these timestamps.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

1.5.1.2 Implementing a ProtocolReceive Handler in an Intermediate Driver

If the NIC driver calls NdisMXxxIndicate, ProtocolReceive is always called, and if the intermediate driver accepts the packet, ProtocolReceive must call NdisTransferData with a packet descriptor and driver-allocated buffers into which the lookahead buffer has been copied and into which the rest of the packet is copied. NdisTransferData must be called in the context of ProtocolReceive, and can only be called once. It is the responsibility of the intermediate driver to set up a packet descriptor with chained buffers of a sufficient size to contain all the received data. After NdisTransferData returns, the received data is no longer available from the underlying NIC driver.

If the data passed to ProtocolReceive was indicated by a call to NdisMXxxIndicateReceive, the size of the lookahead buffer passed to ProtocolReceive is <= the size returned by a call to NdisRequest with OID_GEN_CURRENT_LOOKAHEAD. All data in the lookahead buffer is read-only to the intermediate driver. If the call to ProtocolReceive occurred because the underlying NIC driver set the status of one or more packets in a packet array to NDIS_STATUS_RESOURCES before calling NdisMIIndicateReceivePacket, the size of the lookahead buffer will always be equal to the size of the full network packet so the intermediate driver need not call NdisTransferData.

ProtocolReceive must return control as quickly as possible. The intermediate driver should ensure that it has packet descriptors, buffer descriptors, and buffers available before it gets receive indications. If the intermediate driver examines the lookahead data and determines that the packet is not one it will copy, the intermediate driver should return NDIS_STATUS_NOT_ACCEPTED.
not one it will copy, the intermediate driver should return NDIS_STATUS_NOT_ACCEPTED.

ProtocolReceive must not process received data as it is copied since that would adversely impact system performance, as well as the ability of the underlying NIC to accept incoming receives from the network. Instead, the intermediate driver processes the received data later in its ProtocolReceiveComplete function, which is called subsequently when the packets can be postprocessed. Typically, this occurs when the underlying NIC driver has received and indicated a NIC-driver-determined number of packets or before it exits its DPC-level receive handler. The intermediate driver must queue the copied net packets in ProtocolReceive so that they are available to ProtocolReceiveComplete for postprocessing.

1.5.1.3 Accessing OOB Information From an Intermediate Driver with a Connectionless Lower Edge

If a received network packet is indicated to ProtocolReceive, it forces the driver to copy the received data into an intermediate-driver-supplied buffer. If the packet contains media-specific and/or timestamp information in the OOB data associated with that packet descriptor, an intermediate driver calls NdisGetReceivedPacket and NDIS_GET_ORIGINAL_PACKET to retrieve the media-specific information, as well as TimeSent and TimeReceived if such information is provided by the underlying NIC driver.

If a received packet is passed to ProtocolReceivePacket, the intermediate driver must obtain the information from the OOB data associated with the packet using NDIS-supplied macros as follows:

- Media-specific information is read using NDIS_GET_MEDIA_SPECIFIC_INFO and written using NDIS_SET_MEDIA_SPECIFIC_INFO
- TimeSent is read using NDIS_GET_TIME_SENT and written using NDIS_SET_TIME_TO_SEND
- TimeReceived is read using NDIS_GET_TIME_RECEIVED

TimeSent is the time a packet was sent by the NIC on the remote node, and is retrieved and stored by the underlying NIC driver if available. TimeReceived is the time that the incoming packet was received on the underlying NIC.

1.5.2 Receiving Data in an Intermediate Driver with a Connection-Oriented Lower Edge

An underlying connection-oriented NIC driver indicates packets by calling NdisMCoIndicateReceivePacket, passing a pointer to an array of pointers to one or more packet descriptors. If an intermediate driver is layered above the NIC driver, NDIS then calls the...
descriptors. If an intermediate driver is layered above the NIC driver, NDIS then calls the intermediate driver’s ProtocolCoReceivePacket function.

ProtocolReceivePacket receives a pointer to a packet descriptor that always describes a full network packet. ProtocolReceivePacket examines the packet and if it determines that the packet is intended for one or more of its overlying drivers, it can retain ownership of the packet by returning a nonzero value. If it does return a nonzero value, the intermediate driver must subsequently call NdisReturnPackets with a pointer to the corresponding packet descriptor. The driver must make this call for a particular packet descriptor as many times as the nonzero value it returned from ProtocolCoReceivePacket when it received that packet.

When the intermediate driver calls NdisReturnPackets this number of times, it relinquishes ownership of the packet descriptor and associated buffers back to the underlying driver that originally indicated the receive. Therefore, an intermediate driver should call NdisReturnPackets as quickly as possible.

If, however, an intermediate driver returns zero from ProtocolReceivePacket, the driver indicates that it is relinquishing the packet immediately. This could occur, for instance, if the intermediate driver copies the indicated data into buffers of its own and processes the data internally before indicating up to still higher level drivers.

Network Drivers: Windows 2000 DDK

1.5.2.1 Implementing a ProtocolCoReceivePacket Handler in an Intermediate Driver

When an underlying connection-oriented NIC driver indicates an array of one or more packets, possibly with associated OOB data, by calling NdisMCoIndicateReceivePacket, NDIS calls a bound intermediate driver’s ProtocolCoReceivePacket with each packet descriptor. ProtocolCoReceivePacket must call the NDIS_GET_PACKET_STATUS macro once for each packet descriptor to obtain the OOB block Status for the associated packet.

If a NIC driver, before calling NdisMCoIndicateReceivePacket, set the OOB block Status associated with a packet descriptor to NDIS_STATUS_SUCCESS, the NIC driver temporarily relinquished ownership of the resources associated with the packet descriptor. In this case, the NIC driver depends on the consumers of those packets to return them in a timely manner. Otherwise, the NIC driver can run short of receive resources, such as receive buffer space in the NIC.

When it does run short of resources, a NIC driver sets the OOB block Status associated with a packet descriptor to NDIS_STATUS_RESOURCES. Indicating a packet with this status forces a bound intermediate driver’s ProtocolCoReceivePacket function to copy the packet data immediately rather than retaining the NIC-driver-allocated packet resources. ProtocolCoReceivePacket must return zero in such a situation.

It the underlying NIC driver does not set NDIS_STATUS_RESOURCES in the OOB block associated with a packet descriptor in the packet array that it passes to NdisMCoIndicateReceivePacket, it allows the intermediate driver to retain the packet descriptor and all the resources it specifies until the intermediate driver, overlying protocols, and clients of...
and all the resources it specifies until the intermediate driver, overlying protocols, and clients of overlying protocols have consumed the data and returned the packet descriptor.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

1.5.2.2 Accessing OOB Information from an Intermediate Driver with a Connection-Oriented Lower Edge

An intermediate driver must obtain the information from the OOB data associated with the packet using NDIS-supplied macros as follows:

- Media-specific information is read using NDIS_GET_MEDIA_SPECIFIC_INFO and written using NDIS_SET_MEDIA_SPECIFIC_INFO
- TimeSent is read using NDIS_GET_TIME_SENT and written using NDIS_SET_TIME_TO_SEND
- TimeReceived is read using NDIS_GET_TIME_RECEIVED

TimeSent is the time a packet was sent by the NIC on the remote node, and is retrieved and stored by the underlying NIC driver if available. TimeReceived is the time that the incoming packet was received on the underlying NIC.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

1.5.3 Indicating Receive Packets to Higher Level Drivers

After a connection-oriented intermediate driver has processed a received packet, perhaps converted it to the format expected by a higher level driver, and copied relevant data into buffers chained to an intermediate-driver-allocated packet descriptor, the packet is indicated to the next higher driver by calling NdisMIndicateReceivePacket as if the intermediate driver were a connectionless miniport.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

1.6 Transmitting Packets Through an Intermediate Driver

As discussed in Section 1.1.1, an intermediate driver should always provide a MiniportSendPackets function when it registers with NdisIMRegisterLayeredMiniport. If the intermediate driver is layered between two NDIS drivers that support multipacket sends, the MiniportSendPackets function can forward incoming packet arrays by calling NdisSendPackets if the driver has a connectionless lower edge or NdisCoSendPackets if the driver has a connection-oriented lower edge. If the
intermediate driver is layered under a transport that sends one packet at a time with \texttt{NdisSend}, the MiniportSendPackets function can transmit the single packet with \texttt{NdisSend} or \texttt{Ndis(Co) SendPackets} without any negative performance impact.

Such an intermediate driver does not become a performance bottleneck. If the intermediate driver is layered between a transport that sends a packet array to MiniportSendPackets and a connectionless miniport that only handles a single packet at a time, MiniportSendPackets can send the array of packets it receives with \texttt{NdisSendPackets} without regard to the capabilities of the underlying miniport. NDIS, transparently to the intermediate driver, queues the packets in the array and sends each individual packet to the underlying connectionless miniport’s MiniportSend function when the miniport is able to accept the send request.

Because calls down to the MiniportXxx functions are managed in NDIS, synchronization is guaranteed, so an intermediate driver need not make any \texttt{NdisIMXxx} calls as described in Section 1.4, when forwarding requests that originate in higher level drivers to underlying drivers.

An intermediate driver will, at a minimum, replace the packet descriptor for each incoming send with its own. It must retain the original descriptor (and chained buffers if it copies them to new buffers) from the higher level driver and, when the send completes, the intermediate driver must return the original packet descriptor and the data buffers of the as-sent packet before completing the send back to the higher level driver. For more information on how to allocate packet resources and copy information from one packet to another, see Section 1.3.

MiniportSendPackets receives an array of packet descriptors arranged in an order determined by the caller of \texttt{NdisSendPackets}. In most cases, the intermediate driver should maintain this ordering as it passes an incoming array of packets on to the underlying NIC driver. Only an intermediate driver that adds out-of-band information to incoming packets before passing them on to the underlying driver is likely to reorder an incoming array.

NDIS always preserves the ordering of packet descriptor pointers as passed as an array to \texttt{NdisSendPackets}. The underlying NIC driver also assumes that an array of packet pointers passed in to its MiniportSendPackets function implies the packets should be transmitted in the same order.

An intermediate driver with a connectionless lower edge can transmit a single packet by calling \texttt{NdisSend} with a pointer to the packet descriptor. An intermediate driver with a connectionless lower edge can transmit several packets or just one using \texttt{NdisSendPackets} and passing a pointer to an array of pointer(s) to one or more packet descriptors that the driver has allocated. An intermediate driver with a connection-oriented lower edge can transmit several packets or just one using \texttt{NdisCoSendPackets} and passing a pointer to an array of pointer(s) to one or more packet descriptors.

The driver developer of an intermediate driver with a connectionless lower edge, in general, should determine whether to use \texttt{NdisSend} or \texttt{NdisSendPackets} based upon the driver’s own requirements and any known characteristics of the underlying connectionless NIC driver. An intermediate driver with a connectionless lower edge can call \texttt{NdisRequest} with a \texttt{RequestType} of \texttt{NdisQueryInformation} and an OID\_GEN\_MAXIMUM\_SEND\_PACKETS to obtain the maximum number of send packets that the underlying driver will accept in a packet array. If the underlying driver returns a value of one or \texttt{NDIS\_STATUS\_NOT\_SUPPORTED}, the intermediate driver can choose to use \texttt{NdisSend} rather than \texttt{NdisSendPackets}. If the underlying driver returns a value greater
If any underlying connectionless NIC driver returns a value greater than one, the intermediate driver with a connectionless lower edge should also have a MiniportSendPackets function. In addition, such an intermediate driver should respond to an OID_GEN_MAXIMUM_SEND_PACKETS query with a comparable value to that of the underlying NIC driver.

If OOB information is passed between the intermediate driver and the NIC driver, either send function can be called, since in either case the underlying driver can read the OOB data associated with the packet descriptor using NDIS-supplied macros.

If an intermediate driver sends more packets to an underlying NIC driver than that driver has the internal resources to transmit:

- A deserialized or connection-oriented NIC driver will queue the excess packets in an internal queue and transmit them when it can.
- A serialized NIC driver will either queue the excess packets in an internal queue and transmit them when it can or it will return the excess packet(s) with a status of NDIS_STATUS_RESOURCES. In the latter case, NDIS will hold such packet(s) in an internal queue and resubmit them when the NIC driver next calls NdisMSendResourcesAvailable or NdisMSendComplete, whichever occurs first.

When an intermediate driver with a connectionless lower edge calls NdisSend from MiniportSend, it relinquishes ownership of the packet descriptor and of all the resources that it describes until the send completes, either synchronously or asynchronously. If the status returned by NdisSend is other than NDIS_STATUS_PENDING, the call completes synchronously, and ownership of the packet resources reverts to the intermediate driver on return from NdisSend. The intermediate driver should return any send resources allocated by a higher level driver and propagate the result of its call to NdisSend as the return status from MiniportSend.

If the status returned by NdisSend is NDIS_STATUS_PENDING, when the send subsequently completes, the final status of the send and the packet descriptor will be returned to ProtocolSendComplete. The intermediate driver should call NdisSendComplete from its ProtocolSendComplete function to propagate the status of the send of each packet back to the next higher driver.

When an intermediate driver with a connectionless lower edge transmits one or more packet(s) by calling NdisSendPackets or when an intermediate driver with a connection-oriented lower edge transmits one or more packet(s) by calling NdisCoSendPackets, the send operation is implicitly asynchronous. The caller relinquishes ownership of its packet descriptor(s) and all the resources that they describe until each packet descriptor and the final status of the send for that packet is returned to Protocol(Co)SendComplete. Protocol(Co)SendComplete must propagate the status of the send of each packet back to the next higher driver as described in the previous paragraph.

As a consequence, if an intermediate driver sends packets in an array using NdisSendPackets, the intermediate driver should not attempt to read the Status member(s) of the associated OOB block(s) on return from NdisSendPackets. This member is in use by NDIS to track the progress of an in-transition send request and is volatile. An intermediate driver can only obtain the status of a transmit request by examining the Status argument passed to its Protocol(Co)SendComplete function.
request by examining the Status argument passed to its Protocol(Co)SendComplete function.

If an intermediate driver requests the transmission of an array of packets of different priorities by rearranging the packets that it receives from an overlying driver before transmitting them, it should place the highest priority packets at the beginning of the array. NDIS retains this ordering when it passes the packets to either the MiniportSend or to the Miniport(Co)SendPackets function of the underlying NIC driver, even if NDIS queues some of the packets internally. This ordering is maintained by NDIS for each call to Ndis(Co)SendPackets.

NDIS never attempts to examine and make queuing decisions based on the OOB block associated with a packet descriptor. Unless an intermediate driver has special knowledge of the manner in which a NIC driver handles packet priority, it should assume that the NIC driver transmits the packets in the order in which it receives them, preserving the as-received order.

The structure of the Private member of an NDIS_PACKET-type descriptor is opaque to an intermediate driver and is accessed to read and, in some cases, to write using NDIS-supplied macros or functions. For example, before sending a packet, an intermediate driver can call NdisSetPacketFlags to set the intermediate-defined flags in the NDIS-private portion of the descriptor. Such flags are not defined by NDIS, but are defined by a pair of cooperating protocol and underlying NIC drivers.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

1.6.1 Passing Media-Specific Information

More media-specific information can be passed by an intermediate driver in the OOB block associated with each NDIS_PACKET descriptor. The definition of the OOB block is:

typedef struct _NDIS_PACKET_OOB_DATA {
    union {
        ULONGLONG    TimeToSend;
        ULONGLONG    TimeSent;
    };
    ULONGLONG    TimeReceived;
    UINT             HeaderSize;
    UINT             SizeMediaSpecificInfo;
    PVOID            MediaSpecificInformation;
    NDIS_STATUS   Status;
} NDIS_PACKET_OOB_DATA, *PNDIS_PACKET_OOB_DATA;

The structure of individual records within the buffer at MediaSpecificInformation is defined as follows:

typedef struct MediaSpecificInformation {
    UINT             NextEntryOffset;
    NDIS_CLASS_ID    ClassId;
    UINT             Size;
    UCHAR            ClassInformation[1];
} MEDIA_SPECIFIC_INFORMATION;

The ClassId member is an NDIS-defined enumeration that defines the type of information found at
1.0 NDIS Intermediate Drivers

Currently, there are four class IDs for media in use across Microsoft operating systems that support Win32: NdisClass802_3Priority, NdisClassWirelessWanMbxBMailbox, NdisClassIrdaPacketInfo, and NdisClassAtmAALInfdo. See the Network Driver Reference for details.

If the intermediate driver knows that the underlying NIC driver to which it is sending packets uses OOB data, it can set certain OOB structure members. For instance, an intermediate driver can:

- Request that the packet be sent at a specific time by writing the TimeToSend member using the NDIS_SET_PACKET_TIME_TO_SEND macro. This macro passes the requested time in system time units. The driver can call NdisGetCurrentSystemTime in order to obtain the current system time with which to calculate a requested send time.
- Pass media-specific information in a caller-allocated buffer at MediaSpecificInformation using the NDIS_PACKET_SET_MEDIA_SPECIFIC_INFO macro. For instance, if an intermediate driver is bound to an underlying NIC that requires priority, it sets the ClassId member of the MediaSpecificInformation structure to NdisClass802_3Priority, and pass priority-related information in the ClassInformation member and the size in bytes of this information in Size.

1.7 Handling PnP Events and PM Events in an Intermediate Driver

An intermediate driver must be capable of handling Plug and Play events and Power Management (PM) events. Specifically:

- An intermediate driver must set the NDIS_ATTRIBUTE_NO_HALT_ON_SUSPEND flag in the AttributeFlags parameter passed to NdisMSetAttributesEx (see Section 1.2.2).
- The miniport portion of an intermediate driver must handle OID_PNP_XXX requests.
- The protocol section of an intermediate driver must propagate the appropriate OID_PNP_XXX requests to the underlying miniport. The miniport section of the intermediate driver must pass the underlying miniport’s responses to these requests back to the protocol driver that originated the requests.
- The protocol portion of an intermediate driver must supply a ProtocolPnPEvent handler.

1.7.1 Handling OID_PNP_XXX Queries and Sets

The upper edge of an intermediate driver must export MiniportQueryInformation and MiniportSetInformation functions. When an upper level driver bound to the virtual device instance
exported by the intermediate driver calls \texttt{NdisRequest} to set or query information objects (OID\_XXX), NDIS calls the intermediate driver’s MiniportQueryInformation or MiniportSetInformation function, respectively. NDIS can also call MiniportQueryInformation or MiniportSetInformation on its own behalf. (For more information on miniport handling of sets and queries to information objects, see Chapter 5.)

The intermediate driver itself can query or set OID\_XXX maintained by the underlying miniport with \texttt{NdisRequest} (if the intermediate driver has a connectionless lower edge) or with \texttt{NdisCoRequest} (if the intermediate driver has a connection-oriented lower edge).

After binding to the underlying NIC, the intermediate driver should determine the power-management capabilities of the underlying NIC by querying \texttt{OID\_PNP\_CAPABILITIES}. If the NIC is PM-aware, the underlying miniport returns NDIS\_STATUS\_SUCCESS in response to a query of OID\_PNP\_CAPABILITIES. The miniport also specifies the wake-up capabilities of its NIC. If the NIC is not PM-aware, the underlying miniport returns NDIS\_STATUS\_NOT\_SUPPORTED in response to a query of OID\_PNP\_CAPABILITIES.

How an intermediate driver itself handles queries and sets to Plug and Play information objects (OID\_PNP\_XXX) maintained by its own upper edge depends in part on whether the underlying NIC is PM-aware:

- \texttt{OID\_PNP\_CAPABILITIES}

  If the underlying NIC is PM-aware, the intermediate driver must return NDIS\_STATUS\_SUCCESS to a query of OID\_PNP\_CAPABILITIES. In the NDIS\_PM\_WAKE\_UP\_CAPABILITIES structure returned by this OID, the intermediate driver must specify a device power state of \texttt{NdisDeviceStateUnspecified} for each wake-up capability. Such a response indicates that the intermediate driver is PM-aware but is incapable of waking up the system.

  If the underlying NIC is not PM-aware, the intermediate driver must return NDIS\_STATUS\_NOT\_SUPPORTED to a query of OID\_PNP\_CAPABILITIES.

- \texttt{OID\_PNP\_QUERY\_POWER} and \texttt{OID\_PNP\_SET\_POWER}

  The intermediate driver must always return NDIS\_STATUS\_SUCCESS to a query of OID\_PNP\_QUERY\_POWER or a set of OID\_PNP\_SET\_POWER. The intermediate driver must never propagate either of these OID request to the underlying miniport driver.

- "Wake-up OIDs"

  If the underlying NIC is PM-aware, the intermediate driver must pass to the underlying miniport (by calling \texttt{Ndis(Co)Request}) the following OID\_PNP\_XXX relating to wake-up events:

  - \texttt{OID\_PNP\_ENABLE\_WAKE\_UP}
  - \texttt{OID\_PNP\_ADD\_WAKE\_UP\_PATTERN}
  - \texttt{OID\_PNP\_REMOVE\_WAKE\_UP\_PATTERN}
  - \texttt{OID\_PNP\_WAKE\_UP\_PATTERN\_LIST}
  - \texttt{OID\_PNP\_WAKE\_UP\_ERROR}
  - \texttt{OID\_PNP\_WAKE\_UP\_OK}
The intermediate driver must also propagate the underlying miniport’s response to these OIDs to the overlying protocol drivers.

If the underlying NIC is not PM-aware, the intermediate driver should return NDIS_STATUS_NOT_SUPPORTED in response to a query or set of these OIDs.

1.7.2 Implementing a ProtocolPnPEvent Handler in an Intermediate Driver

When the operating system issues a Plug and Play IRP or a Power Management IRP to a target device object that represents a NIC, NDIS intercepts the IRP and then indicates the event to each bound intermediate driver and each bound protocol driver by calling the driver’s ProtocolPnPEvent handler. In the call to ProtocolPnPEvent, NDIS passes a pointer to a NET_PNP_EVENT structure that describes the Plug and Play event or Power Management event being indicated.

There are six possible Plug and Play and Power Management events, as indicated by the NetEvent code in the NET_PNP_EVENT structure:

- **NetEventSetPower**
  
  Indicates a Set Power request, which specifies that the NIC transition to a particular power state. An intermediate driver should always succeed this event by returning NDIS_STATUS_SUCCESS. For more information on handling a Set Power request, see Section 1.7.3.

- **NetEventQueryPower**
  
  Indicates a Query Power request, which queries whether the NIC can a transition to a particular power state. An intermediate driver should always succeed this event by returning NDIS_STATUS_SUCCESS. An intermediate driver should always succeed a NetEventQueryPower. An intermediate driver should never try to prevent the system from transitioning to the sleeping state by failing a NetEventQueryRemoveDevice. Note that a NetEventQueryPower is always followed by a NetEventSetPower. A NetEventSetPower to the device’s current power state in effect cancels the NetEventQueryPower.

- **NetEventQueryRemoveDevice**
  
  Indicates a Query Remove Device request, which queries whether the NIC can be removed without disrupting operations. If an intermediate driver cannot release a device (for example, because the device is in use) it must fail a NetEventQueryRemoveDevice by returning NDIS_STATUS_FAILURE.

- **NetEventCancelRemoveDevice**
  
  Indicates a Cancel Remove Device request, which cancels the removal of the NIC. The intermediate driver should always succeed this event by returning NDIS_STATUS_SUCCESS.
1.0 NDIS Intermediate Drivers

1. NetEventReconfigure

Indicates that the configuration has changed for a network component. For example, if a user changes the IP address for TCP/IP, NDIS indicates this event to the TCP/IP protocol with the NetEventReconfigure code. The intermediate driver should always succeed this event by returning NDIS_STATUS_SUCCESS.

2. NetEventBindList

Indicates to a TDI client that its bind list has changed. The bind list is a list of one or more device(s) exported by transport protocol(s) to which the TDI client can bind itself.

3. NetEventBindComplete

Indicates that the intermediate driver has bound to all the NICs that it can bind to. NDIS will not indicate any more NICs to the intermediate driver unless a Plug and Play NIC is plugged into the system.

The Buffer member of the NET_PNP_EVENT points to a buffer that contains information specific to the event being indicated. See NET_PNP_EVENT for more information.

An intermediate driver can complete the call to ProtocolPnPEvent asynchronously with NdisCompletePnPEvent.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

1.7.3 Handling a Set Power Request

The way in which an intermediate driver handles a Set Power request depends on whether the underlying NIC is PM-aware.

Handling a Set Power Request to a Sleeping State

1. NDIS calls the ProtocolPnPEvent function of the IM driver and of each protocol driver bound to the virtual device exported by the IM driver. The call to ProtocolPnPEvent specifies a NetEventSetPower to a sleeping state (a network device power state of D1, D2, or D3). Bound protocol drivers stop sending packets and making OID requests to the intermediate driver. The intermediate driver stops sending packets and making OID requests to the underlying miniport.

2. NDIS issues an OID_PNP_SET_POWER request to the miniport (upper edge) of the intermediate driver and to the underlying miniport for the NIC if the NIC is PM-aware. Both the intermediate driver and the underlying miniport succeed the request by returning NDIS_STATUS_SUCCESS. The intermediate driver must not propagate the OID_PNP_SET_POWER request to the underlying miniport.

3. Typically, an intermediate does not deinitialize its virtual NIC. In particular, if the underlying NIC is PM-aware and supports wake-up events, the intermediate driver should not deinitialize its virtual NIC. If the intermediate driver does deinitialize its virtual NIC by calling NdisIMDeinitializeDeviceInstance, NDIS calls the ProtocolUnbindAdapter function of each bound protocol driver to unbind it from the virtual NIC. Before the TCP/IP protocol unbinds, it removes all wake-up patterns from the underlying miniport by issuing an...
OID_PNP_REMOVE_WAKE_UP_PATTERN to the underlying miniport. This, in effect, disables pattern-match wake-up.

**Handling a Set Power Request to the Working State**

1. If the underlying NIC is PM-aware, NDIS issues the underlying miniport an **OID_PNP_SET_POWER** to the working state. NDIS also issues an **OID_PNP_SET_POWER** to the miniport (upper edge) of the intermediate driver. The intermediate driver simply returns NDIS_STATUS_SUCCESS to the Set Power request. The intermediate driver must not propagate the **OID_PNP_SET_POWER** request to the underlying miniport.

2. If the intermediate driver deinitialized its virtual NIC, it must reinitialize the NIC with **NdisIMInitializeDeviceInstance**. In this case, NDIS calls the ProtocolBindAdapter function of each overlying protocol driver so that the protocol driver binds to the virtual NIC.

3. NDIS calls the **ProtocolPnPEvent** function of the IM driver and of each protocol driver bound to the virtual device exported by the IM driver. The call to ProtocolPnPEvent specifies a **NetEventSetPower** to the working state (a network device power state of D0). Bound protocol drivers can start sending packets to the intermediate driver, and the intermediate driver can start sending packets to the underlying miniport.

Note that NDIS can issue the miniport of the intermediate driver an **OID_PNP_SET_POWER** to D0 before sending such an OID to the underlying miniport. NDIS can also call the ProtocolPnPEvent function of bound protocols to indicate the NIC’s transition to D0 before calling the ProtocolPnPEvent function of the intermediate driver to indicate the same event.

In such a case, the miniport of the intermediate driver represents a fully functional NIC before the underlying NIC is indeed fully functional. The intermediate driver must be prepared to handle this situation. If a bound protocol driver sends the miniport of the intermediate driver a packet, the intermediate driver could, for example, queue the packet internally until the underlying NIC is fully functional.

**1.8 Intermediate Driver Reset Operations**

An intermediate driver must be prepared to handle the situation where its outstanding sends on a binding to an underlying driver can be dropped because the underlying NIC is reset.

An underlying driver typically resets its NIC because NDIS calls the NIC driver’s **MiniportReset** function when NDIS times out queued sends or requests bound for the NIC. A miniport can also reset its NIC because a higher level driver called **NdisReset**. If an underlying NIC is reset, NDIS calls the Protocol(Co)Status function of each bound protocol and intermediate driver with a status of **NDIS_STATUS_RESET_START**, and then calls the same bound driver’s **ProtocolStatusComplete** function. When the NIC driver completes the reset, NDIS again calls Protocol(Co)Status with a status of **NDIS_STATUS_RESET_END** followed by a call to ProtocolStatusComplete.

When a NIC is reset, if a bound intermediate driver has any transmit packets pending to that NIC
When a NIC is reset, if a bound intermediate driver has any transmit packets pending to that NIC, NDIS completes those packets back to the intermediate driver with an appropriate status. The intermediate driver must resubmit these packets again when the reset is completed.

When an intermediate driver receives a status of NDIS_STATUS_RESET_START, it should:

- Hold any packets ready to be transmitted until Protocol(Co)Status receives an NDIS_STATUS_RESET_END notification and ProtocolStatusComplete is called.
- Hold any received packets that are ready to be indicated up to the next higher driver until Protocol(Co)Status receives an NDIS_STATUS_RESET_END notification and ProtocolStatusComplete is called.
- Clean up any internal state it maintains for in-progress operations and NIC status.

After Protocol(Co)Status receives NDIS_STATUS_RESET_END and ProtocolStatusComplete is called, the intermediate driver can resume sending packets, making requests and making indications to higher level drivers.

Because an intermediate driver usually disables timing out of sends and requests by NDIS when it calls NdisMSetAttributesEx, its MiniportReset function is seldom called. If MiniportReset is called, possibly because a higher level driver called NdisReset, the intermediate driver should reset its internal state if necessary and it should always set AddressingReset to TRUE before it returns. When the underlying NIC is reset, NDIS will send requests to MiniportSetInformation to reset the intermediate driver’s internal address state for the underlying NIC if the NIC is in a condition to continue transmitting and receiving packets. MiniportReset does not have to complete outstanding sends; NDIS will fail any outstanding sends from the higher level driver appropriately. However, associated state kept by the intermediate driver possibly should be cleaned up.

An intermediate driver can initiate a reset operation by calling NdisReset. If the reset request returns NDIS_STATUS_PENDING, ProtocolResetComplete is called when the underlying NIC or virtual NIC is reset and the driver for the NIC calls NdisMResetComplete. An intermediate driver seldom calls NdisReset unless it has special knowledge that the underlying NIC is not functioning correctly. For instance, if an intermediate driver detects that it has not received completion calls for an unusually large number of sends or requests, and if it has enough knowledge of the underlying NIC to conclude that there is a problem, it can call NdisReset. Usually, however, the need for a NIC reset is detected and initiated by NDIS and the NIC driver using their time-out logic.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

1.9 Intermediate Driver Unbinding Operations

An intermediate driver unbinds from the underlying NIC driver by calling NdisCloseAdapter from its ProtocolUnbindAdapter function which NDIS calls when the underlying NIC is no longer available. For instance, if the underlying NIC timed out, and NDIS called the MiniportHalt function of the NIC driver for the malfunctioning NIC, NDIS will subsequently call the overlying intermediate driver’s ProtocolUnbindAdapter function.

An intermediate driver can also originate the unbind operation, for instance in ProtocolStatus, if the...
An intermediate driver can also originate the unbind operation, for instance in ProtocolStatus, if the status it receives at GeneralStatus is NDIS_STATUS_CLOSING. An intermediate driver also must unbind from an adapter during initialization if some action subsequent to opening an adapter, such as allocating a required resource, fails, and the intermediate driver needs to unbind because it cannot carry out network operations on the binding.

An intermediate driver’s ProtocolUnbindAdapter function calls NdisCloseAdapter. The intermediate driver must not release the per-binding resources until NdisCloseAdapter returns NDIS_STATUS_SUCCESS or the driver calls NdisCompleteUnbindAdapter. If NdisCloseAdapter returns NDIS_STATUS_PENDING, the intermediate driver cannot deallocate any per-binding resources until its ProtocolCloseAdapterComplete function is called.

NDIS does not call ProtocolCloseAdapterComplete until all outstanding requests on the binding have completed. When ProtocolCloseAdapterComplete returns control, the ProtocolBindingContext handle that the intermediate driver allocated to represent the binding is invalid.

Before returning synchronously from ProtocolUnbindAdapter or before completing the unbind operation asynchronously with NdisCompleteUnbindAdapter, the intermediate driver must do the following:

1. Finish cleaning up any state that it maintains for the binding
2. Release any resources that it allocated to establish the binding
3. Call NdisCloseAdapter

If the underlying NIC binding that is being closed is mapped to a device exported by the intermediate driver and if that device was initialized by calling NdisIMInitializeDeviceInstance, the intermediate driver can close these devices by calling NdisIMDeInitializeDeviceInstance or NdisIMInitializeDeviceInstanceEx. The result is that the intermediate driver’s virtual NIC becomes no longer available for sends or requests made by higher level drivers.

After an intermediate driver calls NdisCloseAdapter, it should fail any send requests for that binding with an appropriate error status.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

1.10 Status Indications in an Intermediate Driver

An intermediate driver with a connectionless lower edge is required to supply a ProtocolStatus function and a ProtocolStatusComplete function. NDIS calls ProtocolStatus when an underlying connectionless NIC driver calls NdisMIndicateStatus to report a change in its hardware status. ProtocolStatus is called when the status change begins. If the action indicated by the status code is not complete when ProtocolStatus is called, NdisMIndicateStatusComplete is subsequently called by the underlying NIC driver. When this occurs, ProtocolStatusComplete is called to carry out any postprocessing for the status change.
A connection-oriented intermediate driver is required to supply a ProtocolCoStatus function and a ProtocolStatusComplete function. NDIS calls ProtocolCoStatus when an underlying connection-oriented NIC driver calls NdisMCoIndicateStatus to report a change in its hardware status. ProtocolCoStatus is called when the status change begins.

Examples of *GeneralStatus* reported to ProtocolStatus include:

- NDIS_STATUS_CLOSING. This status and the actions of Protocol(Co)Status is discussed in Section 1.9.
- NDIS_STATUS_RESET_START and NDIS_STATUS_RESET_END. Both these are reported to both Protocol(Co)Status and ProtocolStatusComplete as explained in Section 1.8.
- NDIS_STATUS_LINE_UP, if the intermediate driver is layered over a WAN-capable NIC driver that has established a connection with a remote node. See Part 2, Chapter 8, for more information on WAN drivers.
- NDIS_STATUS_RING_STATUS, for which StatusBuffer provides more detailed information, for instance on problems specific to a Token Ring medium.

When an intermediate driver receives a status indication, it indicates it up to the higher level driver(s) by calling NdisMIndicateStatus only if the status indication causes the intermediate driver to change its internal state in a way that affects the operation of its MiniportXxx functions. That is, if the status indicated to the intermediate driver causes the driver to fail sends or requests, it can pass the status indication to the higher level driver(s) which presumably will then hold off on submitting sends and requests. If, however, the intermediate driver continues to accept sends and requests, perhaps queuing them internally, it should not pass the status indication up.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

**Chapter 2 NDIS Protocol Drivers**

This describes the features of an NDIS driver that exports a set of ProtocolXxx functions at its lower edge. Such a protocol driver communicates with NDIS to send and receive network packets and to bind to and use an underlying miniport NIC driver or intermediate NDIS driver that exports a MiniportXxx interface at its upper edge.

Such an NDIS protocol driver might support TDI at its upper edge, or it might export a private interface to a higher level kernel-mode driver, possibly through a transport stack of drivers, including one that supports TDI at the top of the stack. For instance, an NDIS protocol driver can be the lowest module of a multimodule transport implementation of a standard protocol such as TCP/IP with TDI support in the highest module.

Protocol drivers that communicate with lower level NDIS drivers to send and receive packets always use NDIS-provided functions to communicate with the lower level NDIS drivers. For instance, a protocol driver that has a connectionless lower-edge (which communicates with underlying drivers for connectionless media such as Ethernet or token ring) must call NdisSend or NdisSendPackets to send a packet or packets to a lower level NDIS driver and must call NdisRequest to make or pass through query- or set-information requests with network related OID XXXs that are supported by
through query–set information requests with network-related OID_XXXs that are supported by underlying connectionless drivers. A protocol driver that has a connection-oriented lower edge (which communicates with underlying drivers for connection-oriented media such as ATM and ISDN) must call **NdisCoSendPackets** to send a packet or packets to a lower-level NDIS driver and must call **NdisCoRequest** to query or set OID_XXXs that are supported by underlying connection-oriented drivers.

NDIS also provides a set of **NdisXxx** functions that hide the details of the underlying operating system. For instance, a protocol driver can call **NdisInitializeEvent** to create an event for synchronization purposes and **NdisInitializeListHead** to create a linked list. Protocol drivers that use the NDIS versions of such functions are more portable across Microsoft operating systems that support the Win32® interface. However, protocol drivers can also call OS-specific kernel-mode support routines, such as **KeInitializeEvent** to create an event or **KeWaitForSingleObject** to synchronize two threads of execution. The kernel-mode support routines are documented in the *Kernel-Mode Drivers Reference*.

**Pageable and Discardable Code**

Every ProtocolXxx function runs at an IRQL in the range from PASSIVE_LEVEL to DISPATCH_LEVEL.

Functions that run exclusively at IRQL PASSIVE_LEVEL should be marked as pageable using the **NDIS_PAGABLE_FUNCTION** macro. Driver developers should designate code as pageable whenever possible, freeing system space for code that must be memory-resident. A driver function that runs at IRQL PASSIVE_LEVEL can be made pageable as long as it neither calls nor is called by any function that runs at IRQL >= DISPATCH_LEVEL—for instance a function that acquires a spin lock. Acquiring a spin lock causes the IRQL of the acquiring thread to be raised to IRQL DISPATCH_LEVEL. A driver function, such as ProtocolBindAdapter, that runs at IRQL PASSIVE_LEVEL must not call any **NdisXxx** functions that run at IRQL >= DISPATCH_LEVEL if that driver function is marked as pageable code. See the *Network Driver Reference*, which specifies the IRQL for each **NdisXxx** function.

The **DriverEntry** function of a protocol driver, as well as code that is called only from **DriverEntry**, should be specified as initialization-only code, using the **NDIS_INIT_FUNCTION** macro. Code identified with this macro is assumed to only run once at system initialization time, and, as a result, is only mapped during that time. After a function marked as initialization-only returns, it is discarded.

Access to any driver-allocated shared resource must be synchronized if the resource can be simultaneously shared by two driver functions or if the protocol driver can run on an SMP machine such that the same protocol driver function can be attempting to simultaneously access a resource from more than one processor. For instance, if a driver maintains a shared queue, a spin lock can be used to serialize access to that queue. The spin lock should be initialized when such a queue is created.

However, care should be taken not to overprotect a shared resource, such as an internal driver queue. Some read-only operations can be done without serializing access to a queue, but any operation that manipulates the queue links must be serialized. Spin locks always should be used sparingly and held as short a time as possible. See the *Kernel-Mode Drivers Design Guide* for an in-depth discussion of spin locks.
2.1 Protocol DriverEntry and Initialization

A protocol driver’s initial required entry point must be explicitly named DriverEntry so that the loader can identify it. All other exported functions, described here as ProtocolXxx, can have any developer-specified name since they are passed as addresses to NDIS. The definition of DriverEntry is that of any kernel-mode driver.

```c
NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
);
```

If the protocol driver exports a set of standard kernel-mode driver routines, such as any (Tdi) DispatchXxx and Unload routines, in addition to the NDIS-defined ProtocolXxx functions, the protocol driver must set the entry points of its standard routines in the DriverObject passed to DriverEntry, like any other kernel-mode intermediate driver.

To set up communication with the NDIS library, a protocol’s DriverEntry must register as a protocol driver by calling NdisRegisterProtocol, as described later.

DriverEntry also can initialize any spin locks the protocol requires — for example, to protect state variables that track connections and sends in progress or driver-allocated queues.

If DriverEntry fails to allocate any resources the protocol needs to operate, it should release any previously allocated resources, including making a call to NdisDeregisterProtocol, if necessary, and return an appropriate error status.

While a protocol driver can allocate all the resources it requires in DriverEntry, if the driver provides a ProtocolBindAdapter function as described in Section 2.3, it can defer opening and binding to an underlying NIC (or virtual NIC) driver, as well as reserving system resources to manage such a binding until the initial network I/O request occurs. Then, when ProtocolBindAdapter calls NdisOpenAdapter, it allocates resources as needed for the underlying NIC it opens.
VOID NdisRegisterProtocol(
    OUT PNDIS_STATUS Status,
    OUT PNDIS_HANDLE NdisProtocolHandle,
    IN PNDIS_PROTOCOL.Characteristics ProtocolCharacteristics,
    IN UINT CharacteristicsLength
);

The NdisProtocolHandle returned by this call is opaque to a protocol driver. The handle must be retained by the protocol driver and provided as an input parameter in future calls to NDIS — for example, to open an adapter:

Before making this call, DriverEntry must do the following:

1. Zero-initialize a structure of type NDIS_PROTOCOL.Characteristics — for instance, with a call to NdisZeroMemory. This assures that unused members for optional entry points are set to NULL. If the structure is not zeroed, any unused members must be set to NULL before calling NdisRegisterProtocol.
2. Specify in the NDIS_PROTOCOL.Characteristics structure the NDIS version with which the protocol is compatible, as described in Section 2.1.1.1.
3. Store the addresses of the mandatory ProtocolXxx functions, as well as any optional ProtocolXxx functions the driver exports, in the NDIS_PROTOCOL.Characteristics structure, as described in Section 2.1.1.2 and Section 2.1.1.3.

2.1.1.1 Specifying the NDIS Version Number for a Protocol Driver

The MajorNdisVersion and MinorNdisVersion members in the NDIS_PROTOCOL.Characteristics structure specify the NDIS version with which the protocol is compatible. Valid NDIS version numbers for protocols are 5.0 and 4.0. The current version is 5.0. NDIS continues to support legacy drivers written for version 4.0.

Protocol drivers must be PnP-ready. NDIS therefore no longer supports version 3.0 protocol drivers.

2.1.1.2 Registering the ProtocolXxx Functions of a Connectionless Protocol Driver

The possible and required protocol functions that a connectionless protocol driver can export are listed below:

**BindAdapterHandler**

This is a required function. NDIS calls this function to request that the protocol driver bind to an underlying NIC or virtual NIC whose name is passed as a parameter to this handler. See
Section 2.3 for more information on dynamic binding.

**UnbindAdapterHandler**
This is a required function. ProtocolUnbindAdapter is called by NDIS to close a binding to the underlying NIC or virtual NIC whose name is passed to this function. ProtocolUnbindAdapter calls NdisCloseAdapter and deallocates resources when the binding is successfully closed.

**OpenAdapterCompleteHandler**
This is a required function. If a protocol driver’s call to NdisOpenAdapter returns NDIS_STATUS_PENDING, ProtocolOpenAdapterComplete is subsequently called to complete the binding operation.

**CloseAdapterCompleteHandler**
This is a required function. If a protocol driver’s call to NdisCloseAdapter returns NDIS_STATUS_PENDING, ProtocolCloseAdapterComplete is subsequently called to complete the unbinding operation.

**ReceiveHandler**
This is a required function. ProtocolReceive is called with a pointer to a lookahead buffer. If this buffer contains less than the full, received network packet, ProtocolReceive calls NdisTransferData with a protocol-allocated packet descriptor specifying protocol-allocated buffer(s) to obtain the remainder of the received packet.

**ReceiveCompleteHandler**
This is a required function. ProtocolReceiveComplete is called to indicate that any received packets previously indicated to ProtocolReceive can now be postprocessed.

**TransferCompleteHandler**
This is a required function unless the protocol binds itself exclusively to underlying NIC driver(s) that indicate packets with NdisMIndicateReceivePacket. ProtocolTransferDataComplete is called when a previous call to NdisTransferData returned NDIS_STATUS_PENDING and the remaining data has been copied into the protocol-supplied buffers chained to a given packet descriptor.

**ReceivePacketHandler**
This is an optional function. A ProtocolReceivePacket function should be provided if the protocol driver might be bound to a NIC driver that indicates an array of one or more packets by calling NdisMIndicateReceivePacket.

**SendCompleteHandler**
This is a required function. ProtocolSendComplete is called for each packet transmitted with a call to NdisSend that returned NDIS_STATUS_PENDING as the status of the send operation. If an array of packets is sent, ProtocolSendComplete is called once for each packet passed to NdisSendPackets, whether or not it returned pending.

**ResetCompleteHandler**
This is a required function. ProtocolResetComplete is called when a protocol-initiated reset operation, begun with a call to NdisReset that returned NDIS_STATUS_PENDING, is completed.

**RequestCompleteHandler**
This is a required function. ProtocolRequestComplete is called when a protocol-initiated query or set operation, begun with a call to NdisRequest that returned NDIS_STATUS_PENDING, is completed.

**StatusHandler**
This is a required function. ProtocolStatus is called to handle status changes indicated by the underlying NDIS driver.

**StatusCompleteHandler**
This is a required function. ProtocolStatusComplete is called by NDIS, along with
ProtocolStatus, to report the start and end of an NDIS- or NIC-driver-initiated reset operation.

**PnPEventHandler**
This is a required function. NDIS calls ProtocolPnPEvent to indicate a Plug and Play event or a Power Management event. See Section 2.6 for more information.

**UnloadHandler**
This is an optional function. NDIS calls ProtocolUnload in response to a user request to uninstall a protocol driver. NDIS calls ProtocolUnload after calling ProtocolUnbindAdapter once for each bound adapter. ProtocolUnload performs driver-determined clean-up operations.

A protocol driver should set the *ProtocolCharacteristics TranslateHandler* member to NULL. The TranslateHandler member is reserved for future use.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

**2.1.1.3 Registering the ProtocolXxx Functions of a Connection-Oriented Protocol Driver**

A connection-oriented protocol can or must register the following protocol handler functions that are common to both connectionless and connection-oriented protocols:

- BindHandler (required)
- UnbindHandler (required)
- OpenAdapterCompleteHandler (required)
- CloseAdapterCompleteHandler (required)
- ReceiveCompleteHandler (required)
- ResetCompleteHandler (required)
- RequestCompleteHandler (required)
- StatusCompleteHandler (required)
- PnPEventHandler (required)

These functions are summarized in Section 2.1.1.2.

A connection-oriented protocol must also register the following connection-oriented protocol functions. Specifying these entry points requires a V5.0 structure at *ProtocolCharacteristics*:

**CoSendCompleteHandler**
This is a required function. ProtocolCoSendComplete is called once for each packet passed to *NdisCoSendPackets*. The protocol driver can determine the status of a send operation that calls *NdisCoSendPackets* only from the status argument input to ProtocolCoSendComplete.

**CoStatusHandler**
This is a required function. ProtocolCoStatus is called by NDIS with status notifications initiated by an underlying NIC driver.

**CoReceivePacketHandler**
This is a required function. When a bound connection-oriented NIC driver or MCM indicates an array of pointer(s) by calling *NdisMCoIndicateReceivePacket*, NDIS calls the protocol driver’s ProtocolCoReceivePacketHandler.
CoAfRegisterNotifyHandler

If the protocol driver is a connection-oriented client that uses the call manager services of a call manager or an MCM driver, it must register a ProtocolCoAfRegisterNotify function. NDIS calls the ProtocolCoAfRegisterNotify function of each protocol on a binding when a call manager or MCM driver registers an address family with Ndis(M)CmRegisterNotify. ProtocolCoAfRegisterNotify determines whether the protocol driver can use the services of a call manager or MCM that has advertised its services. Protocol drivers that are standalone call managers must also register a CoAfRegisterNotifyHandler. When called by NDIS, the ProtocolCoAfRegisterNotify function of a call manager simply returns control.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

2.1.2 Opening an Adapter Underlying a Protocol Driver

The protocol driver reads the registry information stored during setup to build a list of names of adapters to which it will bind. The registry contains binding information written when the network is configured. DriverEntry reads this information, including the names of the adapter or adapters to which it can bind.

A protocol driver calls NdisOpenProtocolConfiguration to obtain a handle to the registry key where the protocol driver can store adapter-specific information. A protocol driver can then call NdisOpenConfigurationKeyByIndex or NdisOpenConfigurationKeyByName to obtain a handle to a subkey below the registry key opened with NdisOpenProtocolConfiguration.

A protocol driver that stores adapter-specific information in the registry must store it under the ProtocolName key and use NDIS functions to access the information. Once a handle to this key is obtained, a protocol driver calls the NdisRead/WriteConfiguration functions to read and write such information. The NdisRead/WriteConfiguration functions are described in the Network Driver Reference.

If a protocol driver provides a ProtocolBindAdapter function, the SystemSpecific1 parameter passed to this function is passed uninterpreted to NdisOpenProtocolConfiguration at ProtocolSection. If a protocol driver does not provide a ProtocolBindAdapter function, the driver is assumed to know both the name of the adapter it is opening and the string value to be passed at ProtocolSection.

After a protocol driver has retrieved the information it requires from the registry and has registered by calling NdisRegisterProtocol, and before the protocol driver can send packets and receive incoming data, it must bind itself to one or more underlying NICs managed by a miniport or an intermediate NDIS driver. The protocol binds itself to an underlying NIC and the miniport that controls it by calling NdisOpenAdapter, declared as follows:

```c
VOID
NdisOpenAdapter(
    OUT PNDIS_STATUS Status,
    OUT PNDIS_STATUS OpenErrorStatus,
    OUT PNDIS_HANDLE NdisBindingHandle
)
```
1.0 NDIS Intermediate Drivers

A protocol driver passes a handle at ProtocolBindingContext to an adapter-specific context area that it allocated to store state information for the binding. NDIS will return this handle to the protocol driver in subsequent calls pertaining to the binding — for instance, in calls to ProtocolReceive or ProtocolStatus. NDIS returns a handle at NdisBindingHandle to the protocol driver. This handle must be retained by the protocol, usually in its adapter-specific context area. The protocol driver must pass this handle to NDIS in future calls relating to this binding.

The protocol driver passes the name of the adapter that it has read from the registry or that was input to the ProtocolBindAdapter function at DeviceName to NdisOpenAdapter at AdapterName. It passes the type(s) of medium(s) it supports at MediumArray. If the call to NdisOpenAdapter succeeds, the underlying NIC driver will select a medium from MediumArray and return its index at SelectedMediumIndex. The value at NdisProtocolHandle is the value returned to the protocol driver from a successful call to NdisRegisterProtocol.

If a protocol driver cannot successfully bind to an underlying adapter, it deallocates any resources it previously allocated for that adapter. If the protocol driver cannot successfully open any of the possible adapters, it should deallocate any global resources the protocol has previously allocated and return an appropriate failure status. In this case, the protocol driver will subsequently be unloaded. Typically, its DriverEntry or ProtocolBindAdapter function should log failed binding attempts with appropriate descriptive information, either by calling NdisWriteErrorLogEntry or an OS-specific support routine, such as IoWriteErrorLogEntry.

Network Drivers: Windows 2000 DDK

2.1.3 Protocol Driver Query and Set Operations

After a protocol driver has determined which adapters to open and has successfully bound to one or more adapters, it can call Ndis(Co)Request to query the underlying NDIS driver for its characteristics and to set the protocol’s own operating characteristics. The protocol can also negotiate certain parameters for the binding.

The driver can make requests with general OID_GEN_(CO_)XXX types of OIDs, which are independent of the underlying medium. The driver can also make requests with medium-specific OIDs that are defined for a particular medium type. The Network Driver Reference documents both types of OIDs. Some OIDs are mandatory, meaning an underlying driver must support them; others are optional.
2.1.3.1 Queries and Sets from Connectionless Protocol Drivers

A connectionless protocol calls NdisRequest to query or set the characteristics of an underlying miniport.

A connectionless protocol driver uses the general OID_GEN_MAXIMUM_FRAME_SIZE to query the maximum frame size, in bytes, supported by an underlying NIC driver. The size returned does not include the packet header.

A connectionless protocol driver queries a binding with the general OID_GEN_MAXIMUM_TOTAL_SIZE to determine the largest packet the underlying NIC driver can accommodate on the NIC it manages. The protocol driver must restrict the size of the packets it subsequently sends to this size. It is an error for a protocol driver to submit a larger packet to the underlying NIC driver than the NIC driver has indicated it can support.

A connectionless protocol driver should use OID_GEN_MAXIMUM_SEND_PACKETS to determine the number of send packets that an underlying driver can accept each time the protocol calls NdisSendPackets. See Section 2.5 for a discussion of how this request can affect the protocol driver’s subsequent send operations.

A connectionless protocol driver can make a query request and/or a set request concerning the size of the lookahead data buffer the underlying driver supplies whenever the protocol’s ProtocolReceive function is called. The OID used is OID_GEN_CURRENT_LOOKAHEAD. If the protocol driver issues this as a query request, NDIS returns the current lookahead buffer size for the given binding to the underlying NIC driver. If the protocol driver makes a set request with this OID code, it indicates its preferred lookahead buffer size, but the protocol is not assured that an underlying driver will conform to this request. If the underlying driver returned a value greater than one for a preceding OID_GEN_MAXIMUM_SEND_PACKETS query, that driver always indicates receives with a lookahead buffer containing a full network packet.

A connectionless protocol driver can query the underlying NIC driver for its link speed with OID_GEN_LINK_SPEED and use the results to set any internal time-out values it maintains. If the protocol driver is bound to NDISWAN, it cannot determine the link speed until it receives a line-up indication from the WAN NIC driver, indicating that a connection has been established between the local node and a remote node.

A connectionless protocol driver must also issue an OID_GEN_MAC_OPTIONS query to determine the operating characteristics of the underlying NIC driver. This query returns such information as whether the underlying driver supports full-duplex operations. See the Network Driver Reference for the set of NDIS-defined flags that can be returned for this query.

If needed, a connectionless protocol driver issues a set request to inform NDIS of its own operating characteristics, passing OID_GEN_PROTOCOL_OPTIONS.
2.1.3.2 Queries and Sets from Connection-Oriented Protocol Drivers

A connection-oriented protocol calls NdisCoRequest to query or set the characteristics of an underlying connection-oriented miniport.

A connection-oriented protocol driver can query the underlying NIC driver for its link speed with OID_GEN_CO_LINK_SPEED and use the results to set any internal time-out values it maintains. If the protocol driver is bound to NDISWAN, it cannot determine the link speed until it receives a line-up indication from the WAN NIC driver, indicating that a connection has been established between the local node and a remote node. A connection-oriented protocol driver can also set the speed of the underlying NIC with OID_GEN_CO_LINK_SPEED.

A connection-oriented protocol driver must also issue an OID_GEN_CO_MAC_OPTIONS query to determine the operating characteristics of the underlying NIC driver. See the Network Driver Reference for the set of NDIS-defined flags that can be returned for this query.

If needed, a connection-oriented protocol driver issues a set request to inform NDIS of its own operating characteristics, passing OID_GEN_CO_PROTOCOL_OPTIONS.

2.1.3.3 Queries and Sets to WAN Miniports

A connectionless or connection-oriented protocol driver bound above a WAN-capable NIC must also make the following set-information requests:

- OID_WAN_PROTOCOL_TYPE to inform the underlying NDISWAN driver of the type of the protocol driver. The type is supplied as a single-byte, network-level protocol identifier.
- OID_WAN_HEADER_FORMAT to inform the underlying NDISWAN driver of the header format of the packets it sends.

2.1.3.4 Medium-Dependent Queries

A protocol driver queries the medium-dependent current address with a medium-dependent OID. For example, a connectionless protocol might query one of the following: OID_WAN_CURRENT_ADDRESS, OID_802_3_CURRENT_ADDRESS, OID_802_5_CURRENT_ADDRESS, or OID_FDDI_LONG_CURRENT_ADDRESS. A connection-oriented protocol might query OID_WAN_CURRENT_ADDRESS or...
2.1.4 Registering as a Call Manager or a Connection-Oriented Client

A connection-oriented protocol must register either as a call manager or a connection-oriented client. See Part 1, Chapter 4 for an description of call managers and connection-oriented clients.

2.1.4.1 Registering as a Call Manager

A protocol driver registers itself as a call manager by calling NdisCmRegisterAddressFamily. The protocol driver calls NdisCmRegisterAddressFamily from its ProtocolBindAdapter function after the driver has established a binding to the underlying miniport with NdisOpenAdapter. The protocol driver calls NdisCmRegisterAddressFamily each time that its ProtocolBindAdapter function is called, effectively registering an address family for each NIC on which it provides call-management services to connection-oriented clients.

By calling NdisCmRegisterAddressFamily, a protocol driver both registers its call manager functions and advertises its specific signaling services for all clients on the binding. After the call manager’s ProtocolBindAdapter function returns control with a successful registration as a stand-alone call manager, NDIS calls the ProtocolCoRegisterAfNotify functions of all connection-oriented clients on the binding (see Section 2.1.4.2).

NdisCmRegisterAddressFamily is declared as follows:

```c
NDIS_STATUS
NdisCmRegisterAddressFamily(
    IN NDIS_HANDLE NdisBindingHandle,
    IN PCO_ADDRESS_FAMILY AddressFamily,
    IN PNDIS_CALL_MANAGER_CHARACTERISTICS CmCharacteristics,
    IN UINT SizeOfCmCharacteristics
);
```

Before calling NdisCmRegisterAddressFamily, the protocol driver must:

1. Zero a structure of type NDIS_CALL_MANAGER_CHARACTERISTICS. The current version of the CmCharacteristics structure is 5.0.
2. Store the addresses of the ProtocolXxx call manager functions that that driver supports.
The call manager functions that a protocol driver must register with

**NdisCmRegisterAddressFamily** are listed below:

- **CmCreateVcHandler**
  Specifies the entry point of the caller’s ProtocolCoCreateVc function.

- **CmDeleteVcHandler**
  Specifies the entry point of the caller’s ProtocolCoDeleteVc function.

- **CmOpenAfHandler**
  Specifies the entry point of the caller’s ProtocolCmOpenAf function.

- **CmCloseAfHandler**
  Specifies the entry point of the caller’s ProtocolCmCloseAf function.

- **CmRegisterSapHandler**
  Specifies the entry point of the caller’s ProtocolCmRegisterSap function.

- **CmDeregisterSapHandler**
  Specifies the entry point of the caller’s ProtocolCmDeregisterSap function.

- **CmMakeCallHandler**
  Specifies the entry point of the caller’s ProtocolCmMakeCall function.

- **CmCloseCallHandler**
  Specifies the entry point of the caller’s ProtocolCmCloseCall function.

- **CmIncomingCallCompleteHandler**
  Specifies the entry point of the caller’s ProtocolCmIncomingCallComplete function.

- **CmAddPartyHandler**
  Specifies the entry point of the caller’s ProtocolCmAddParty function.

- **CmDropPartyHandler**
  Specifies the entry point of the caller’s ProtocolCmDropParty function.

- **CmActivateVcCompleteHandler**
  Specifies the entry point of the caller’s ProtocolCmActivateVcComplete function.

- **CmDeactivateVcCompleteHandler**
  Specifies the entry point of the caller’s ProtocolCmDeactivateVcComplete function.

- **CmModifyCallQoSHandler**
  Specifies the entry point of the caller’s ProtocolCmModifyCallQoS function.

- **CmRequestHandler**
  Specifies the entry point of the caller’s ProtocolCoRequest function.

- **CmRequestCompleteHandler**
  Specifies the entry point of the caller’s ProtocolCoRequestComplete function.

A protocol driver must set every **CmXxx** member in the

**NDIS_CALL_MANAGER_CHARACTERISTICS** structure, even if it does not support incoming calls, outgoing calls, or point-to-multipoint connections. For whatever subset of connection-oriented functionality that such a call manager does not support, its placeholder ProtocolCmXxx functions simply return **NDIS_STATUS_NOT_SUPPORTED**.

After a stand-alone call manager calls **NdisCmRegisterAddressFamily** successfully, NDIS ignores any entry point(s) that the CM previously specified in the **RequestHandler** and **RequestCompleteHandler** members of the **NDIS_PROTOCOL_CHARACTERISTICS** structure that it passed to **NdisRegisterProtocol**.
2.1.4.2 Registering as a Connection-Oriented Client

When a call manager or MCM, from its ProtocolBindAdapter function, calls \texttt{Ndis(M)CmRegisterAddressFamily} to register an address family, NDIS calls the ProtocolCoRegisterAfNotify function of all protocol drivers on the binding. If an protocol driver registered a ProtocolCoAfRegisterNotify function when it registered as a protocol, NDIS calls the protocol driver’s ProtocolCoAfRegisterNotify function.

If ProtocolCoAfRegisterNotify determines that the protocol driver can use the services of the call manager or MCM that registered the address family, it allocates a per-AF context area for the client and calls \texttt{NdisClOpenAddressFamily}. With the call to \texttt{NdisClOpenAddressFamily}, a protocol driver registers a set of client-supplied functions.

\texttt{NdisClOpenAddressFamily} is declared as follows:

\begin{verbatim}
NDIS_STATUS
NdisClOpenAddressFamily(
    IN NDIS_HANDLE NdisBindingHandle,
    IN PCO_ADDRESS_FAMILY AddressFamily,
    IN NDIS_HANDLE ProtocolAfContext,
    IN PNDIS_CLIENT_CHARACTERISTICS ClCharacterisitics,
    IN UINT SizeOfClCharacteristics,
    OUT PNDIS_HANDLE NdisAfHandle
);
\end{verbatim}

Before calling \texttt{NdisClOpenAddressFamily}, the protocol driver must:

1. Zero a structure of type NDIS\_CLIENT\_CHARACTERISTICS. The current version of the \texttt{ClCharacteristics} structure is 5.0.
2. Store the addresses of the ProtocolXxx client functions that that driver supports.

The \texttt{NdisAfHandle} returned by this call is opaque to a protocol driver. The handle must be retained by the driver and provided as an input parameter in future calls made by the protocol part of the driver to NDIS — for instance, to register a SAP with \texttt{NdisClRegisterSap}.

The client functions that a protocol driver must register with \texttt{NdisClOpenAddressFamily} are listed below:

- **ClCreateVcHandler**
  Specifies the entry point of the caller’s ProtocolCoCreateVc function.

- **ClDeleteVcHandler**
  Specifies the entry point of the caller’s ProtocolCoDeleteVc function.

- **ClRequestHandler**
  Specifies the entry point of the caller’s ProtocolCoRequest function.

- **ClRequestCompleteHandler**
  Specifies the entry point of the caller’s ProtocolCoRequestComplete function.

- **ClOpenAfCompleteHandler**
  Specifies the entry point of the caller’s ProtocolClOpenAfComplete function.
1.0 NDIS Intermediate Drivers

1.1 Protocol Driver Characteristics

Specifies the entry point of the caller’s ProtocolClOpenAfComplete function.

ClCloseAfCompleteHandler
Specifies the entry point of the caller’s ProtocolClCloseAfComplete function.

ClRegisterSapCompleteHandler
Specifies the entry point of the caller’s ProtocolClRegisterSapComplete function. A client uses this function to accept incoming calls from remote machines.

ClDeregisterSapCompleteHandler
Specifies the entry point of the caller’s ProtocolClDeregisterSapComplete function.

ClMakeCallCompleteHandler
Specifies the entry point of the caller’s ProtocolClMakeCallComplete function. A client uses this function to make outgoing calls to remote machines.

ClModifyCallQoSCompleteHandler
Specifies the entry point of the caller’s ProtocolClModifyCallQoSComplete function. A client uses this function to make changes in the quality of service on an established VC dynamically or to negotiate with the call manager to establish the QoS when setting up an incoming call.

ClCloseCallCompleteHandler
Specifies the entry point of the caller’s ProtocolClCloseCallComplete function.

ClAddPartyCompleteHandler
Specifies the entry point of the caller’s ProtocolClAddPartyComplete function. A client uses this function to establish point-to-multipoint VCs for outgoing calls to remote machines.

ClDropPartyCompleteHandler
Specifies the entry point of the caller’s ProtocolClDropPartyComplete function.

ClIncomingHandler
Specifies the entry point of the caller’s ProtocolClIncomingCall function. A client uses this function to accept incoming calls from remote machines.

ClIncomingCallQoSChangeHandler
Specifies the entry point of the caller’s ProtocolClIncomingCallQoSChange function. A client uses this function to accept incoming calls from remote machines on which the sending client can change the QoS dynamically.

ClIncomingCloseCallHandler
Specifies the entry point of the caller’s ProtocolClIncomingCloseCall function.

ClIncomingDropPartyHandler
Specifies the entry point of the caller’s ProtocolClIncomingDropParty function.

ClCallConnectedHandler
Specifies the entry point of the caller’s ProtocolClCallConnected function. A client uses this function to accept incoming calls from remote machines.

The protocol driver must set every ClXxx member in the NDIS_CLIENT_CHARACTERISTICS structure to a caller-supplied ProtocolCl/CoXxx function when it calls NdisClOpenAddressFamily, even if it does not support incoming calls, outgoing calls, or point-to-multipoint connections. For whatever subset of connection-oriented functionality that such an intermediate driver does not support, its ProtocolCl/CoXxx functions simply return NDIS_STATUS_NOT_SUPPORTED.

Network Drivers: Windows 2000 DDK

2.2 Protocol Driver Packet Management
A protocol driver receives one or more buffers of data from a client to transmit over the network. A protocol driver must, at a minimum, allocate and initialize a packet descriptor to which the client’s data buffers are chained. Packet descriptors must be allocated from packet pool, as follows:

1. Call `NdisAllocatePacketPool` or `NdisAllocatePacketPoolEx` to allocate and initialize a block of nonpaged pool for a caller-specified number of fixed-size packet descriptors during driver initialization or when each binding is first established.
2. Call `NdisAllocatePacket` to allocate a packet descriptor from the pool allocated by `NdisAllocatePacketPool`.

Buffer descriptor(s) mapping such buffer(s) are chained to a packet descriptor by calling `NdisChainBufferAtBack` or `NdisChainBufferAtFront`. If a protocol driver receives a data buffer from a client that must be sent in several smaller buffers, the protocol driver can copy the data into protocol-allocated buffers, map its buffers with previously allocated buffer descriptors, and chain these buffer descriptors to protocol-allocated packet descriptors. Such buffers can be allocated by calling a kernel-mode support routine, `NdisAllocateMemory`, or `NdisAllocateMemoryWithTag` and can be mapped with protocol-allocated buffer descriptors as follows:

1. Call `NdisAllocateBufferPool` to obtain a handle with which to allocate buffer descriptors during driver initialization or when each binding is first established.
2. Call `NdisAllocateMemoryWithTag` to allocate a buffer to chain to a packet descriptor allocated by calling `NdisAllocatePacket`.
3. Call `NdisAllocateBuffer` to allocate and set up a buffer descriptor that maps the buffer allocated by calling `NdisAllocateMemoryWithTag`.

The base virtual address and the length returned by `NdisAllocateMemoryWithTag` are passed in the call to `NdisAllocateBuffer` to initialize a buffer descriptor.

Packet descriptors to meet typical transmit needs can be allocated as needed, when the driver initializes, and/or at binding time. A protocol driver developer can allocate a number of packet descriptors with chained buffer descriptors at initialization time to hold receives that can occur as soon as the protocol binds itself to an underlying NIC driver. Then ProtocolReceive can return control to the underlying driver as quickly as possible. Otherwise, subsequently received data can be lost. As described later, a connectionless protocol driver can either receive incoming data from an underlying NIC driver at its ProtocolReceivePacket function as a packet descriptor specifying a full network packet or have data indicated to its ProtocolReceive function, which must copy the indicated data into a protocol-supplied buffer chained to a protocol-allocated packet descriptor. A connection-oriented protocol always receives incoming data at its ProtocolCoReceivePacket function.

Every connectionless protocol driver must provide a ProtocolReceive function. When this function is called, the protocol driver must copy the indicated lookahead data into a protocol-allocated buffer chained to a preallocated packet descriptor, which it must pass to `NdisTransferData`. This call passes the packet descriptor to the underlying NIC driver to be filled if there is additional data received beyond the indicated lookahead data.

If a protocol driver binds itself only to underlying NDIS drivers that indicate arrays of packet(s) with `NdisM(Co)IndicateReceivePackets`, Protocol(Co)ReceivePacket need not provide packet descriptors, buffers, and buffer descriptors for incoming data. When the protocol driver has a full network packet indicated to its Protocol(Co)ReceivePacket function, it can give client(s) direct read-
only access to the buffered data described by an input packet descriptor until the data is consumed and the client(s) release the packet descriptor and all the resources it specifies. When the protocol driver takes ownership of such packet resources, it has no need to copy data into a preallocated packet and call `NdisTransferData` to obtain the remaining packet data.

If a protocol driver supplies a packet descriptor with more than one chained buffer to `NdisTransferData`, `NdisSend(Packets)`, or `NdisCoSendPackets` and the length of actual data in the last buffer is less than the allocated length of its buffer, the protocol driver should call `NdisAdjustBufferLength` to set the actual length of the data in the buffer descriptor. When the packet descriptor is returned to the protocol driver, the driver should readjust the buffer descriptor’s specified length to the full length of the buffer.

**Reusing Packets**

Ownership of protocol-allocated packet resources for sends reverts to the protocol driver when `Ndis (Co)Send` returns anything other than `NDIS_STATUS_PENDING` or when the driver’s Protocol(Co)SendComplete function is called. Then the protocol driver can reclaim the returned packet resources for subsequent sends or, if it’s a connectionless driver, for copying received data in ProtocolReceive.

It is more efficient for a protocol driver to reinitialize and reuse the packet descriptor and buffers than to deallocate these resources and then later reallocate them again for a subsequent send or data-transfer operation. A protocol usually exhibits better performance if it saves unchained buffer descriptors and buffers for reuse rather than deallocating and reallocating such resources.

A protocol driver reinitializes a packet descriptor by calling `NdisReinitializePacket`. First, the protocol driver should take care to remove any chained buffers and their buffer descriptors by calling `NdisUnchainBufferAtXxx` to release the buffer descriptors and the buffers mapped by these descriptors. Otherwise, `NdisReinitializePacket` sets the member that points to the chained buffer descriptor(s) to NULL, so reinitializing the packet descriptor without first releasing and saving the chained buffer descriptors will cause a memory leak. Similarly, if the protocol and underlying driver use out-of-band information, the resources specified in the OOB data block associated with each packet descriptor must be reclaimed before a call to `NdisReinitializePacket`.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

## 2.3 Dynamic Binding in a Protocol Driver

A PnP-ready protocol driver can support dynamic binding to underlying NICs by providing both a `ProtocolBindAdapter` function and a `ProtocolUnbindAdapter` function.

If a driver registers these functions with NDIS, it indicates that it will defer opening and binding to an underlying NIC as is otherwise done in the `DriverEntry` function and instead perform these actions in its `ProtocolBindAdapter` function. If a protocol provides these dynamic-bind functions, whenever an underlying NIC becomes available, NDIS will call the `ProtocolBindAdapter` function of any protocol driver that can bind itself to that adapter, and whenever an underlying NIC is closed, NDIS will call the reciprocal `ProtocolUnbindAdapter` function.
ProtocolBindAdapter is declared as follows:

```c
VOID
ProtocolBindAdapter(  
    OUT PNDIS_STATUS   Status,  
    IN NDIS_HANDLE    BindContext,  
    IN PNDIS_STRING   DeviceName,  
    IN PVOID          SystemSpecific1,  
    IN PVOID          SystemSpecific2)
```

NDIS supplies the name of the newly available adapter for ProtocolBindAdapter to open at `DeviceName`. NDIS passes a handle at `BindContext` that represents its context for the bind request. The protocol driver must retain this handle and pass it back to NDIS as a parameter to `NdisCompleteBindAdapter` when the driver has completed its binding-related activities and is ready to transmit and receive packets.

Binding-time actions include allocating and initializing an adapter-specific context area for the binding and calling `NdisOpenAdapter` to open the adapter passed at `DeviceName`. `DeviceName` might refer to an underlying NIC or it can be the name of a virtual NIC exported by an NDIS intermediate driver that is layered between the protocol driver and the NIC driver managing the adapter to which transmit requests are directed. For instance, an intermediate NDIS driver can translate between the format of the medium supported by a legacy protocol driver and the format of the medium supported by an underlying NIC driver. The protocol-allocated context area or a comparable location should be used to store the `BindContext`.

The `BindContext` value must be stored because `NdisOpenAdapter` can return `NDIS_STATUS_PENDING`. If `NdisOpenAdapter` returns a pending status, the protocol driver cannot call `NdisCompleteBindAdapter` until the open operation is complete and the protocol driver is called at its ProtocolOpenAdapterComplete function. In this case, `BindContext` must be retrieved from a known location and passed to `NdisCompleteBindAdapter`.

`SystemSpecific1` is a string that must be passed to `NdisOpenProtocolConfiguration`. This string is opaque to protocol drivers.

`SystemSpecific2` is reserved for system use.

For more information about dynamic unbinding, see Section 2.8.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

### 2.4 Receiving Data in a Protocol Driver

This section describes how data is received in a connectionless protocol driver and also in a connection-oriented protocol driver.
2.4.1 Receiving Data in a Connectionless Protocol Driver

Underlying connectionless NIC drivers can indicate packets in two ways:

- A NIC driver calls `NdisMIndicateReceivePacket`, passing a pointer to an array of packet descriptor pointer(s) to one or more full packets and relinquishing ownership of the resources for these packets to the overlying drivers, which can consume the data and return the packet resources at a later time.

- A NIC driver calls a filter-specific `NdisMXX::IndicateReceive` function, passing a pointer to a lookahead buffer, the size of the lookahead buffer, and the total size of the received net packet.

Every connectionless protocol driver must have at least one of two possible receive handlers and can have both of the following:

1. **ProtocolReceive** is a required function that is called with a pointer to a lookahead buffer.

   After ProtocolReceive examines the lookahead data and determines that the packet is intended for one or more of its clients, it must copy the lookahead data into a protocol-allocated buffer, possibly to be chained to a protocol-allocated packet descriptor. If the size of the lookahead buffer is less than the total size of the received packet, ProtocolReceive must call `NdisTransferData` with the protocol-allocated packet descriptor to obtain the rest of the received data, which the underlying driver copies into the protocol-supplied buffer.

2. **ProtocolReceivePacket** is an optional function that receives a pointer to packet descriptor specifying a buffered full network packet.

   ProtocolReceivePacket also examines the packet data and determines whether the packet is intended for one or more of its clients. If so, the protocol can give its client(s) ownership of the indicated packet resources, including direct read-only access to the buffered net packet data, by returning a nonzero value (a count of clients to which the protocol forwarded the receive indication) from ProtocolReceivePacket. The protocol driver’s client(s) must subsequently return the packet descriptor and all the resources it specifies to the underlying driver. Each client must return the packet descriptor until all clients’ calls total the nonzero value returned from ProtocolReceivePacket for that receive indication.

   When the protocol driver’s client(s) call return the packet descriptor the required number of times, they relinquish ownership of the returned packet resources to the underlying NIC driver that originally indicated the receive.

   On the other hand, if a protocol driver returns zero from ProtocolReceivePacket, it relinquishes ownership of the packet resources when ProtocolReceivePacket returns. Relinquishing such a received packet immediately could occur, for instance, if the client for the packet has closed the connection or in some manner become unavailable, or if the protocol copies the indicated data into buffers of its own and processes the data internally before indicating up to its clients.
2.4.1.1 Implementing a ProtocolReceivePacket Handler in a Protocol Driver

When an underlying connectionless NIC driver indicates an array of one or more packet descriptors with \texttt{NdisMIndicateReceivePacket}, NDIS will usually call a bound protocol driver’s ProtocolReceivePacket function with each packet descriptor, thereby allowing the protocol driver (or its clients) to retain the packet descriptor and all the resources it specifies until the protocol or its client(s) have consumed the data and returned the packet descriptor. Two kinds of connectionless NIC drivers typically call \texttt{NdisMIndicateReceivePacket} with an array of packet descriptors:

1. A NIC driver managing a busmaster DMA adapter that is capable of receiving several packets at a time into a ring of buffers
2. A NIC driver that provides out-of-band data containing media-specific information, such as packet priority, in the NDIS_PACKET_OOB_DATA block associated with the packet descriptor

Such a driver need not be the driver of a busmaster DMA device.

If a protocol driver is aware that it is (or might be) bound to such a NIC driver, it should have a ProtocolReceivePacket function. This allows the protocol driver to do all of the following:

- To receive indications of full network packets from the underlying NIC driver
- To read the OOB data associated with each packet descriptor using NDIS-supplied macros
- To retain ownership of the input packet descriptor and direct read-only access to the buffered data specified by the descriptor if the protocol makes copies of the data for its clients, or even to forward the indicated packet descriptor to its client(s) after selecting the range of packet data of interest to the client(s)
- To return the packet descriptor and the resources it described, possibly along with other retained packet descriptors, with \texttt{NdisReturnPackets} when the protocol has consumed the indicated data if it retains ownership of the input packet descriptor; otherwise, to have clients to which the protocol forwarded a receive indication return the packet descriptor.

Even when a protocol driver provides a ProtocolReceivePacket handler, there are cases when a call by a NIC driver to \texttt{NdisMIndicateReceivePacket} results in a call to the protocol driver’s ProtocolReceive function. Since a NIC driver temporarily relinquishes ownership of driver-allocated resources when it calls \texttt{NdisMIndicateReceivePacket}, the underlying driver depends on the consumers of those packets to return them in a timely manner. Otherwise, the NIC driver can run short of receive resources, such as receive buffer space in the NIC. When it does, the NIC driver sets NDIS_STATUS_RESOURCES in the OOB block associated with a packet descriptor in the packet array it passes to \texttt{NdisMIndicateReceivePacket}. Indicating a packet with this status causes NDIS to call the overlying driver’s ProtocolReceive function with such a packet and with any subsequent packets in the array, thus forcing the protocol driver to copy the packet data immediately, rather than retaining the NIC-driver-allocated packet resources.

If the protocol driver requires the OOB data associated with a packet descriptor but is called at ProtocolReceive, it must call \texttt{NdisGetReceivedPacket} and \texttt{NDIS_GETORIGINAL_PACKET} to
copy the media-specific information into a protocol-supplied buffer and possibly the `TimeSent` and `TimeReceived` values if the underlying NIC driver provides these timestamps.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

### 2.4.1.2 Implementing a ProtocolReceive Handler in a Protocol Driver

If the underlying connectionless NIC driver calls a filter-specific `NdisMXXxIndicateReceive` function, NDIS always calls the ProtocolReceive function of each bound protocol driver.

If one or more of such a protocol driver’s clients are the target of the packet and the lookahead buffer size is less than the total packet size, ProtocolReceive must do the following:

1. Copy the lookahead data into an internal buffer mapped by a protocol-allocated packet descriptor.
2. Chain buffer descriptor(s) mapping sufficient protocol-allocated buffers to contain the rest of the network packet data to a protocol-allocated packet descriptor.
3. Call `NdisTransferData` with the packet descriptor, so the underlying driver copies the rest of the received data into the protocol’s buffer(s).

When `NdisTransferData` returns STATUS_SUCCESS or the ProtocolTransferDataComplete function is called, the lookahead buffer can be chained to the packet descriptor containing this transferred data and indicated up to any interested client(s).

`NdisTransferData` can only be called once for each receive indication. It is the responsibility of the protocol driver to set up its packet descriptor with chained buffers of a sufficient size to contain the full network packet. After `NdisTransferData` returns, the received data is no longer available from the underlying NIC driver.

The size of the lookahead buffer passed to ProtocolReceive is the minimum of the size returned by a call to `NdisRequest` with OID_GEN_CURRENT_LOOKAHEAD and the total packet size. All data in the lookahead buffer is read-only to the ProtocolReceive.

If the lookahead buffer size is greater than or equal to the size of the total packet indicated by a filter-specific `NdisMXXxIndicateReceive` function, ProtocolReceive should call `NdisTransferData` to copy the lookahead data into an internal buffer. Instead, the

If the call to ProtocolReceive occurred because the underlying NIC driver set the status for one or more packets in a packet array to NDIS_STATUS_RESOURCES before calling `NdisMIndicateReceivePacket`, the size of the lookahead buffer will always be equal to the size of the full network packet. In these circumstances, the protocol driver never calls `NdisTransferData` because ProtocolReceive can copy the full indication into an internal buffer immediately.

If the NDIS_MAC_OPTION_COPY_LOOKAHEAD_DATA bit is set in response to an OID_GEN_MAC_OPTIONS request, a protocol driver can use any means to copy lookahead data into an internal buffer, such as calling `NdisMoveMemory`. If this flag was not set, the protocol driver
1.0 NDIS Intermediate Drivers

into an internal buffer, such as calling NdisMoveMemory(). If this flag was not set, the protocol driver must call NdisCopyLookaheadData to copy the indicated data; otherwise, the results from a copy operation are indeterminate.

ProtocolReceive must execute as quickly as possible. The protocol driver must ensure that it has protocol-allocated packet resources available before it receives incoming indications. After the protocol driver examines the packet and determines that the packet is not one it will copy, ProtocolReceive should simply return NDIS_STATUS_NOT_ACCEPTED.

ProtocolReceive must not process the received data as soon as it is copied since that would adversely impact system performance, as well as possibly causing dropped receives in the underlying NIC driver. Instead, the protocol driver processes the received data later in its ProtocolReceiveComplete function. Following one or more receive indications, ProtocolReceiveComplete is called when the underlying NIC driver calls NdisMxxIndicateReceiveComplete. Typically, this occurs when the underlying NIC driver has received and indicated a miniport-determined number of packets or before it exits its receive handler. The protocol driver must queue the received data in ProtocolReceive so that it is available to ProtocolReceiveComplete for postprocessing.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

2.4.1.3 Accessing OOB Information From a Connectionless Protocol Driver

When a received network packet is indicated to ProtocolReceive, it forces the driver to copy the received data into a protocol-supplied buffer. If such an indication has associated media-specific and/or timestamp information in the OOB block associated with the packet descriptor, a protocol driver can call NdisGetReceivedPacket and NDIS_GET_ORIGINAL_PACKET to copy the media-specific information, as well as any TimeSent and TimeReceived timestamps.

If a received packet is passed to ProtocolReceivePacket, the protocol driver can copy the information from the associated OOB block using NDIS-supplied macros as follows:

- Media-specific information is copied using NDIS_GET_MEDIA_SPECIFIC_INFO.
- TimeSent is copied using NDIS_GET_TIME_SENT.
- TimeReceived is copied using NDIS_GET_TIME_RECEIVED.

TimeSent is the time a packet was sent by the NIC driver on the remote node, and is retrieved and stored by the underlying NIC driver on the local node if available. TimeReceived is the time that the incoming packet was received on the underlying NIC on the local node.

The current system time can be determined using NdisGetCurrentSystemTime or KeQuerySystemTime if the protocol converts these timestamps into another format.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

2.4.2 Receiving Data in a Connection-Oriented Protocol Driver
Underlying connection-oriented NIC drivers indicate packets by calling `NdisMCoIndicateReceivePacket`, passing a pointer to an array of packet descriptor pointer(s) to one or more full packets and relinquishing ownership of the resources for these packets to the overlying drivers, which can consume the data and return the packet resources at a later time.


`ProtocolCoReceivePacket` also examines the packet data and determines whether the packet is intended for one or more of its clients. If so, the protocol can give its client(s) ownership of the indicated packet resources, including direct read-only access to the buffered net packet data, by returning a nonzero value (a count of clients to which the protocol forwarded the receive indication) from `ProtocolCoReceivePacket`. The protocol driver’s client(s) must subsequently return the packet descriptor and all the resources it specifies to the underlying driver. Each client must make this call with the packet descriptor until all clients’ calls total the nonzero value returned from `ProtocolCoReceivePacket` for that receive indication.

When the protocol driver’s client(s) return the packet descriptor the required number of times, they relinquish ownership of the returned packet resources to the underlying NIC driver that originally indicated the receive.

On the other hand, if a protocol driver returns zero from `ProtocolCoReceivePacket`, it relinquishes ownership of the packet resources when `ProtocolCoReceivePacket` returns. Relinquishing such a received packet immediately could occur, for instance, if the client for the packet has closed the connection or in some manner become unavailable, or if the protocol copies the indicated data into buffers of its own and processes the data internally before indicating up to its clients.

2.4.2.1 Implementing a `ProtocolCoReceivePacket` Handler

When an underlying connection-oriented NIC driver indicates an array of one or more packet descriptors with `NdisMCoIndicateReceivePacket`, NDIS calls a bound protocol driver’s `ProtocolReceivePacket` function with each packet descriptor. `ProtocolCoReceivePacket` must call the `NDIS_GET_PACKET_STATUS` macro once for each packet descriptor to obtain the OOB block `Status` for the associated packet.

If a NIC driver, before calling `NdisMCoIndicateReceivePacket`, set the OOB block `Status` associated with a packet descriptor to `NDIS_STATUS_SUCCESS`, the NIC driver temporarily relinquished ownership of the resources associated with the packet descriptor. In this case, the NIC driver depends on the consumers of those packets to return them in a timely manner. Otherwise, the NIC driver can run short of receive resources, such as receive buffer space in the NIC.

When it does run short of resources, a NIC driver sets a packet’s OOB block `Status` to `NDIS_STATUS_RESOURCES`. Indicating a packet with this status forces the...
1.0 NDIS Intermediate Drivers

NDIS_STATUS_RESOURCES. Indicating a packet with this status forces the ProtocolCoReceivePacket function of the protocol driver to copy the packet data immediately rather than retaining the NIC-driver-allocated packet resources. ProtocolCoReceivePacket must return zero in such a situation.

It the underlying NIC driver does not set NDIS_STATUS_RESOURCES in the OOB block associated with a packet descriptor in the packet array that it passes to NdisMCoIndicateReceivePacket, it allows the protocol driver (or its clients) to retain the packet descriptor and all the resources it specifies until the protocol or its client(s) have consumed the data and returned the packet descriptor.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

2.4.2.2 Accessing OOB Information From a Connection-Oriented Protocol

A connection-oriented protocol driver can copy the information from the associated OOB block of an indicated receive packet using NDIS-supplied macros as follows:

- Media-specific information is copied using NDIS_GET_MEDIA_SPECIFIC_INFO.
- TimeSent is copied using NDIS_GET_TIME_SENT.
- TimeReceived is copied using NDIS_GET_TIME_RECEIVED.

TimeSent is the time a packet was sent by the NIC driver on the remote node, and is retrieved and stored by the underlying NIC driver on the local node, if available. TimeReceived is the time that the incoming packet was received on the underlying NIC on the local node.

The current system time can be determined using NdisGetCurrentSystemTime if the protocol converts these timestamps into another format.

A connection-oriented miniport may be able to indicate the TimeReceived for a packet in the NIC’s local time. To query a connection-oriented miniport’s or MCM’s local timing capabilities, a protocol calls NdisCoRequest with OID_GEN_CO_GET_TIME_CAPS. To obtain the NIC’s local time, a protocol calls NdisCoRequest with OID_GEN_CO_GET_NETCARD_TIME.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

2.5 Sending Protocol Driver-Originated Packets

This section describes sending packets from a connectionless protocol driver and also from a connection-oriented protocol driver.

Built on Wednesday, June 28, 2000
2.5.1 Sending Packets from a Connectionless Protocol Driver

A connectionless protocol driver can transmit a single packet by calling NdisSend, passing in a pointer to a packet descriptor with chained buffer descriptors mapping the buffered data to be sent. Alternatively, a connectionless protocol driver can transmit several packets using NdisSendPackets, passing in a pointer to an array of pointer(s) to one or more packet descriptors.

In general, a connectionless protocol driver developer should choose whether to call NdisSend or NdisSendPackets based upon the driver’s own requirements and on the characteristics of the underlying NIC driver.

When it has bound itself to an underlying NIC driver, a connectionless protocol driver should call NdisRequest with an OID_GEN_MAXIMUM_SEND_PACKETS query to determine the maximum number of send packets that the underlying driver will accept in a packet array. If the NIC driver supports only single-packet sends, either at its MiniportSend or its MiniportSendPackets function, the return value will be one or, possibly NDIS_STATUS_NOT_SUPPORTED. Either of these returns implies the protocol driver is likely to call NdisSend rather than NdisSendPackets. If the underlying driver returns a value greater than one, both drivers' performance will be better if the protocol driver sends an array of packets with NdisSendPackets. If OOB information is passed between the protocol driver and the NIC driver, either NDIS function can be called, since, in either case, the underlying driver can read the OOB data using NDIS-supplied macros.

Whenever a protocol driver calls NdisSend, it relinquishes ownership of the given packet resources until the send completes, either synchronously or asynchronously. If the status returned by NdisSend is something other than NDIS_STATUS_PENDING, the send completes synchronously and ownership of the protocol-allocated packet resources reverts to the protocol driver. If the status returned by NdisSend is NDIS_STATUS_PENDING, when the send subsequently completes, the final status of the send and the protocol-allocated packet descriptor will be passed in to the ProtocolSendComplete function.

When a protocol driver transmits one or more packet(s) by calling NdisSendPackets, send operations are always asynchronous. The protocol driver relinquishes ownership of the packet resources that it allocated until each packet descriptor and the final status of the send for that packet is returned to ProtocolSendComplete.

As a consequence, a protocol driver never reads the Status member in the OOB block associated with a packet descriptor on return from NdisSend(Packets). The protocol cannot learn the status of its send request in this manner because this member is in use by NDIS to track the progress of an in-transition send request and is volatile until the packet descriptor is returned to ProtocolSendComplete. A protocol driver always obtains the status of a transmit request either by examining the value returned by NdisSend or from the Status parameter passed to ProtocolSendComplete.

If a protocol driver requests the transmission of an array of packets of different priorities by arranging the packets it receives from clients before transmitting them, the protocol should place the highest
NDIS always preserves the ordering of packets in any array passed to NdisSendPackets, even if NDIS queues some of the packets internally.

NDIS does not attempt to examine and make queueing decisions based on any of the OOB data associated with the packet descriptors given to NdisSendPackets (or to NdisSend). Unless a protocol driver has special knowledge of the manner in which the underlying NIC driver handles packet priorities or the TimeToSend timestamps, the protocol should assume that the underlying NIC driver transmits all packets in the order in which it receives them, preserving the as-received order. Consequently, a protocol should order the packet arrays it sends according to the order in which those packets should be transmitted over the network.

Network Drivers: Windows 2000 DDK

2.5.1.1 Passing Media-Specific Information from a Connectionless Protocol Driver

Before sending a packet, a protocol driver can call NdisSetPacketFlags to set protocol-defined flags in the NDIS-private portion of the packet descriptor. Such flags are not defined by NDIS, but are defined for use by a cooperating pair of protocol and lower level NDIS drivers. The structure of the Private member of an NDIS_PACKET is opaque to all NDIS drivers and is accessed to read and, in some cases, to write using NDIS-supplied functions or macros.

More media-specific information can be passed by a protocol driver in the OOB block associated with each NDIS_PACKET-type descriptor. The definition of the OOB block is:

```c
typedef struct NDIS_PACKET_OOB_DATA {
    union {
        ULONGLONG TimeToSend;
        ULONGLONG TimeSent;
    };
    ULONGLONG TimeReceived;
    UINT HeaderSize;
    UINT SizeMediaSpecificInfo;
    PVOID MediaSpecificInformation;
    NDIS_STATUS Status;
} NDIS_PACKET_OOB_DATA, *PNDIS_PACKET_OOB_DATA;
```

The structure of individual records within a driver-allocated buffer at MediaSpecificInformation is defined as follows:

```c
typedef struct MediaSpecificInformation {
    UINT NextEntryOffset;
    NDIS_CLASS_ID ClassId;
    UINT Size;
    UCHAR ClassInformation[1];
} MEDIA_SPECIFIC_INFORMATION;
```

The ClassId member is an NDIS-defined enumeration that defines the type of information found at ClassInformation[1]. Currently, there are three class IDs for connectionless media in use across Microsoft operating systems that support Win32: NdisClass802_3Priority, NdisClassWirelessWanMbxMailbox, and NdisClassIrdaPacketInfo. See the NetworkReference for details.
If the protocol driver knows that the underlying NIC driver to which it is sending packets uses OOB data, the protocol can set the following OOB structure members:

- Request that the packet be sent at a specific time by setting the `TimeToSent` member using the `NDIS_SET_PACKET_TIME_TO_SEND` macro. This macro passes the requested time in system time units. The protocol can call `NdisGetCurrentSystemTime` in order to obtain the current system time with which to calculate a requested send time.
- Pass media-specific information in a protocol-allocated buffer at `MediaSpecificInformation` using the `NDIS_PACKET_SET_MEDIA_SPECIFIC_INFO` macro to set this member to the address of the buffer. For instance, if a protocol driver is bound to an underlying NIC that requires priority, it will set the `ClassId` member of the `MediaSpecificInformation` structure to `NdisClass802_3Priority`, and pass priority-related information in the `ClassInformation` member and the size in bytes of this information in `Size`. The protocol driver is responsible for allocating a buffer to contain any media-specific data record(s) and for setting up the pointer to this buffer at `MediaSpecificInformation`.

2.5.2 Sending Packets from a Connection-Oriented Protocol Driver

A connection-oriented protocol driver can transmit one or more packets with `NdisCoSendPackets`, passing in a pointer to an array of pointer(s) to one or more packet descriptors.

When a protocol driver transmits one or more packet(s) by calling `NdisCoSendPackets`, send operations are always asynchronous. The protocol driver relinquishes ownership of the packet resources that it allocated until each packet descriptor and the final status of the send for that packet is returned to `ProtocolCoSendComplete`.

As a consequence, a protocol driver never reads the `Status` member in the OOB block associated with a packet descriptor on return from `NdisCoSendPackets`. The protocol cannot learn the status of its send request in this manner because this member is in use by NDIS to track the progress of an in-transition send request and is volatile until the packet descriptor is returned to `ProtocolCoSendComplete`. A protocol driver always obtains the status of a transmit request from the `Status` parameter passed to `ProtocolCoSendComplete`.

If a protocol driver requests the transmission of an array of packets of different priorities by arranging the packets it receives from clients before transmitting them, the protocol should place the highest priority packets at the beginning of the array. NDIS always preserves the ordering of packets in any array passed to `NdisCoSendPackets`.

NDIS does not attempt to examine and make queueing decisions based on any the OOB data associated with the packet descriptors given to `NdisCoSendPackets`. Unless a protocol driver has...
associated with the packet descriptors given to NdisCoSendPackets. Unless a protocol driver has special knowledge of the manner in which the underlying NIC driver handles packet priorities or the TimeToSend timestamps, the protocol should assume that the underlying NIC driver transmits all packets in the order in which it receives them, preserving the as-received order. Consequently, a protocol should order the packet arrays it sends according to the order in which those packets should be transmitted over the network.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

2.5.2.1 Passing Media-Specific Information from a Connection-Oriented Protocol Driver

Before sending a packet, a protocol driver can call NdisSetPacketFlags to set protocol-defined flags in the NDIS-private portion of the packet descriptor. Such flags are not defined by NDIS, but are defined for use by a cooperating pair of protocol and lower level NDIS drivers. The structure of the Private member of an NDIS_PACKET is opaque to all NDIS drivers and is accessed to read and, in some cases, to write using NDIS-supplied functions or macros.

More media-specific information can be passed by a protocol driver in the OOB block associated with each NDIS_PACKET-type descriptor. The definition of the OOB block is:

```c
typedef struct _NDIS_PACKET_OOB_DATA {
    union {
        ULONGLONG TimeToSend;
        ULONGLONG TimeSent;
    };
    ULONGLONG TimeReceived;
    UINT HeaderSize;
    UINT SizeMediaSpecificInfo;
    PVOID MediaSpecificInformation;
    NDIS_STATUS Status;
} NDIS_PACKET_OOB_DATA, *PNDIS_PACKET_OOB_DATA;
```

The structure of individual records within a driver-allocated buffer at MediaSpecificInformation is defined as follows:

```c
typedef struct MediaSpecificInformation {
    UINT NextEntryOffset;
    NDIS_CLASS_ID ClassId;
    UINT Size;
    UCHAR ClassInformation[1];
} MEDIA_SPECIFIC_INFORMATION;
```

The ClassId member is an NDIS-defined enumeration that defines the type of information found at ClassInformation[1]. Currently, there is one class ID in use across Microsoft operating systems that support Win32:NdisClassAtmAALInfo. See the NetworkReference for details.

If the protocol driver knows that the underlying NIC driver to which it is sending packets uses OOB data, the protocol can set the following OOB structure members:

- Request that the packet be sent at a specific time by setting the TimeToSend member using the
A connection-oriented protocol can use a NIC’s local time for scheduling the transmission of packets and for time-stamping send packets. To query a connection-oriented miniport’s or MCM’s local timing capabilities, a protocol calls NdisCoRequest with OID_GEN_CO_GET_TIME_CAPS. To obtain the NIC’s local time, a protocol calls NdisCoRequest with OID_GEN_CO_GET_NETCARD_TIME.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

2.6 Handling PnP Events and Power Management Events in a Protocol Driver

When the operating system issues a Plug and Play IRP or a Power Management IRP to a target device object that represents a NIC, NDIS intercepts the IRP and then indicates the event to each bound protocol and each bound intermediate driver by calling the driver’s ProtocolPnPEvent handler. In the call to ProtocolPnPEvent, NDIS passes a pointer to a NET_PNP_EVENT structure that describes the Plug and Play event or Power Management event being indicated.

There are six possible Plug and Play and Power Management events, as indicated by the NetEvent code in the NET_PNP_EVENT structure:

- **NetEventSetPower**
  
  Indicates a Set Power request, which specifies that the NIC transition to a particular power state. A PM-aware protocol should always succeed this event by returning NDIS_STATUS_SUCCESS. A legacy protocol can return NDIS_STATUS_NOT_SUPPORTED to indicate that NDIS should unbind it from the NIC.

- **NetEventQueryPower**
  
  Indicates a Query Power request, which queries whether the NIC can a transition to a particular power state. A protocol should always succeed a NetEventQueryPower. After establishing an active connection, a protocol can call PoRegisterSystemState to register a continous busy state. As long as the state registration is in effect, the Power Manager does not attempt to put the system to sleep. After the connection becomes inactive, the protocol cancels the state registration by calling PoUnregisterSystemState. A protocol should never try to prevent the system from transitioning to the sleeping state by failing a NetEventQueryRemoveDevice. Note that a NetEventQueryPower is always followed by a NetEventSetPower. A NetEventSetPower to the device’s current power state in effect cancels the...
1.0 NDIS Intermediate Drivers

10/18/02

NetEventSetPower to the device’s current power state in effect cancels the NetEventQueryPower.

- NetEventQueryRemoveDevice

Indicates a Query Remove Device request, which queries whether the NIC can be removed without disrupting operations. If a protocol cannot release a device (for example, because the device is in use) it must fail a NetEventQueryRemoveDevice by returning NDIS_STATUS_FAILURE.

- NetEventCancelRemoveDevice

Indicates a Cancel Remove Device request, which cancels the removal of the NIC. The protocol should always succeed this event by returning NDIS_STATUS_SUCCESS.

- NetEventReconfigure

Indicates that the configuration has changed for a network component. For example, if a user changes the IP address for TCP/IP, NDIS indicates this event to the TCP/IP protocol with the NetEventReconfigure code. The protocol should always succeed this event by returning NDIS_STATUS_SUCCESS.

- NetEventBindList

Indicates to a client that its bind list has changed. The bind list is a list of one or more device(s) exported by transport protocol(s) to which the client can bind itself.

- NetEventBindsComplete

Indicates that a protocol driver has bound to all the NICs that it can bind to. Indicates that a protocol driver has bound to all the NICs that it can bind to. NDIS will not indicate any more NICs to the protocol unless a Plug and Play NIC is plugged into the system.

The Buffer member of the NET_PNP_EVENT points to a buffer that contains information specific to the event being indicated. See NET_PNP_EVENT for more information.

A protocol can complete the call to ProtocolPnPEvent asynchronously with NdisCompletePnPEvent.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

2.7 Protocol Driver Reset Operations

A protocol driver can initiate a reset operation by calling NdisReset. If its reset request returns NDIS_STATUS_PENDING, ProtocolResetComplete is called when the reset operation is complete.

A protocol driver seldom calls NdisReset unless it has special knowledge that the underlying NIC is not functioning correctly. For instance, if a protocol driver detects that it has not received completion calls for an unusually large number of sends or requests, and it has enough knowledge of the underlying NIC to conclude that this indicates hardware problems, it can call NdisReset. Typically, however, the need for a reset is detected and initiated by NDIS and/or the NIC driver itself using time-out logic. Any protocol driver bound to an underlying NDIS driver that reports its medium type.
time-out logic. Any protocol driver bound to an underlying NDIS driver that reports its medium type
as NdisMediumWan cannot call NdisReset.

Typically, an underlying NIC driver resets its NIC because the NIC is timing out during send or
request operations, which causes NDIS to call the NIC driver’s MiniportCheckForHang and
subsequently MiniportReset function, or because the NIC driver determines the NIC’s receive
capability is disfunctional. If a reset is initiated by NDIS and MiniportReset returns
NDIS_STATUS_PENDING, NDIS first calls the Protocol(Co)Status function of each bound protocol
driver with a status of NDIS_STATUS_RESET_START, and then calls the same bound protocol
driver’s ProtocolStatusComplete function. When the NIC driver call NdisResetComplete, NDIS
again calls Protocol(Co)Status with a status of NDIS_STATUS_RESET_END followed by a call to
ProtocolStatusComplete.

A protocol driver must handle the possibility that outstanding sends on a binding to an underlying
NIC can be cancelled because the NIC is reset. If a bound protocol driver has any transmit packets
pending, NDIS will complete those packets back to the protocol driver with an appropriate status.
The protocol driver must resubmit the packets again when the reset operation is completed, assuming
the NIC becomes operational again.

When a protocol driver receives a status of NDIS_STATUS_RESET_START, it should do the
following:

- Hold any packets ready to be transmitted until Protocol(Co)Status receives an
  NDIS_STATUS_RESET_END notification.
- Not make any NDIS calls directed to the underlying miniport, except calls to return resources
  such as returning packets with NdisReturnPackets.

After Protocol(Co)Status receives an NDIS_STATUS_RESET_END message and
ProtocolStatusComplete is called, the protocol driver can resume sending packets and requests.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

2.8 Protocol Driver Unbinding Operations

A protocol driver unbinds from an underlying NIC by calling NdisCloseAdapter. If a protocol driver
registers a ProtocolUnbindAdapter function, NDIS will call this function if the underlying NIC is no
longer available. For instance, if the NIC timed out, an attempt to reset the NIC failed to make it
operational, and NDIS called the MiniportHalt function of the NIC driver for the malfunctioning
NIC, NDIS will subsequently call the overlying protocol driver’s ProtocolUnbindAdapter function.

If a protocol driver does not register this function, an unbind operation can originate in the driver’s
Protocol(Co)Status function, as the result of receiving a status of NDIS_STATUS_CLOSING at
GeneralStatus. A protocol driver also should unbind from an adapter during initialization if some
action subsequent to opening an adapter, such as allocating a required resource, fails, and the driver
needs to unbind because it cannot carry out network operations on the binding.

If a protocol driver registers a ProtocolUnbindAdapter, this function calls NdisCloseAdapter.
1.0 NDIS Intermediate Drivers

If a protocol driver registers a ProtocolUnbindAdapter, this function calls NdisCloseAdapter. However, the protocol must not deallocate all its binding-specific resources until it calls NdisCompleteUnbindAdapter. If the protocol driver does not provide a ProtocolUnbindAdapter function, but calls NdisCloseAdapter, it can deallocate any previously allocated resources when either NdisCloseAdapter returns NDIS_STATUS_SUCCESS, or, if NdisCloseAdapter returns NDIS_STATUS_PENDING, when ProtocolCloseAdapterComplete is called.

NDIS delays calling ProtocolCloseAdapterComplete until all outstanding requests on the binding have completed. Once the close request completes, the ProtocolBindingContext handle the protocol driver allocated to represent the binding becomes invalid.

Before returning synchronously from ProtocolUnbindAdapter or before completing the unbind operation asynchronously with NdisCompleteUnbindAdapter, the driver must do the following:

- Finish cleaning up any state that it maintains for the binding
- Release any resources that it allocated to establish the binding
- Call NdisCloseAdapter

2.9 Status Indications in a Protocol Driver

A connectionless protocol driver is required to supply both ProtocolStatus and ProtocolStatusComplete functions. A connection-oriented protocol driver is required to supply both ProtocolCoStatus and ProtocolStatusComplete functions.

NDIS calls ProtocolStatus when an underlying connectionless NDIS driver calls NdisMIndicateStatus to report a change in its NIC. NDIS calls ProtocolCoStatus when an underlying connection-oriented NDIS driver calls NdisMCoIndicateStatus.

Protocol(Co)Status is called when the status change begins. If the action indicated by the status message is not complete when the ProtocolStatus function or a connectionless protocol is called, NdisMIndicateStatusComplete is subsequently called by an underlying connectionless NIC driver. When this occurs, ProtocolStatusComplete is called to carry out any postprocessing indicated by the status change. Note that a connection-oriented miniport does not have to indicate that it has finished sending status and therefore does not call ProtocolStatusComplete.

Examples of GeneralStatus reported to ProtocolStatus include:

- NDIS_STATUS_CLOSING — This status and the actions of Protocol(Co)Status were already discussed in Section 2.8.
- NDIS_STATUS_RESET_START and NDIS_STATUS_RESET_END — In this case, both of these statuses are reported to both Protocol(Co)Status and ProtocolStatusComplete as explained already in Section 2.7.
- NDIS_STATUS_LINE_UP — This status is indicated if the protocol driver is layered over
NDISWAN while an underlying WAN-capable NIC driver has an established connection with a remote node. The protocol driver exchanges handles for an active connection on which it can send and receive as well as query the connection for call-specific information. See WAN Miniport NIC Drivers in Chapter 8 of Part 2, for more information on WAN drivers.

- NDIS_STATUS_RING_STATUS or any other medium-specific status value, for which StatusBuffer provides more detailed information about the status, for instance, problems specific to a Token Ring medium.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

Chapter 3 TDI Transports and Their Clients

This describes, in general, the relationship between TDI transport drivers and TDI kernel-mode clients, such as higher level network-interface emulators, redirectors, and servers, that communicate with any transport protocol stack using the system-defined TDI interface. That is, this provides an overview of the following:

- The Transport Driver Interface (TDI) in Section 3.1
- How PnP-aware TDI transports use device objects to represent their transport-to-NIC bindings and how they dynamically register network addresses available to clients through these bindings with TDI in Section 3.2
- How TDI uses file objects to represent network entities in Section 3.3, including the following three basic TDI entities:
  1. Transport Addresses in Section 3.3.1
  2. Connection Endpoints in Section 3.3.2
  3. Control Channels in Section 3.3.3
- The set of transport driver Dispatch routines required by TDI in Section 3.4
- The interactions between TDI clients and their underlying transports in Section 3.5
- The distinction between TDI requests and events in Section 3.6

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

3.1 The Transport Driver Interface (TDI)

Figure 3.1 shows the general relationship between TDI clients and transport drivers.
1.0 NDIS Intermediate Drivers

Figure 3.1 TDI clients and transport providers

As this figure shows, the Transport Driver Interface (TDI) defines a kernel-mode network interface that is exposed at the upper edge of all transport protocol stacks. The highest level protocol driver in every such stack supports the TDI interface for still higher level kernel-mode network clients.

This interface includes the following:

- A set of standard kernel-mode intermediate driver Dispatch routines exported by each TDI transport driver to which clients can submit I/O requests (IRPs) by making calls to support routines, such as the Zw..File routines and/or IoCallDriver
- A set of ClientEventXxx callback routines exported by each TDI client that can be registered with the underlying transport driver to receive notifications of specific network events when they occur
- A set of ClientPnPXxx callback routines exported by TDI clients that can be registered with TDI to receive notifications of dynamic binding, network address, and power-state changes from PnP-aware Windows® 2000 transports
- Parameters, structures, IOCTLs, and procedural rules associated with these TDI transport and ClientEventXxx and ClientPnPXxx routines
- A set of system-supplied TdiXxx functions that transports and clients can call to communicate with each other
- A set of system-supplied TdiBuildXxx macros and functions that clients can use to set up I/O requests to be submitted to their underlying transports

Requiring that all transport drivers expose a single common interface (TDI) simplifies the task of transport driver development in that all transports need code to support only a single defined interface. It also simplifies the task of client development by minimizing the amount of transport-specific code that must be written.

Windows® 2000 includes interface modules for several popular network interfaces, such as Windows Sockets and NetBIOS. Each of these interface modules exposes a native set of primitive functions, which are accessible through standard calls from user mode. When called, the interface module maps the native (frequently user-mode) function and its associated parameters and procedural rules, to one or more calls to the underlying TDI transport driver.

Key features of TDI include the following:

**High Level of Granularity**
TDI accommodates all primitives and conventions from existing popular network interfaces because it is relatively granular in nature, with several small TDI-defined requests that can be mixed and matched to accommodate mapping from existing network-interface functions.

**Asynchronous Operation**
Most kernel-mode TDI operations are asynchronous, using client-supplied callback routines to
indicate asynchronous network events as they occur and completing most client-initiated operations submitted as IRPs asynchronously as well.

32-bit Addressing and Values
Like all Windows 2000 kernel-mode components, TDI transports and their clients are 32-bit code. TDI-defined structures and parameters use 32-bit pointers and values.

Flexible Addressing Scheme
TDI does not mandate any particular address format, such as the 16-character NetBIOS name defined for legacy operating systems. Instead, it features an extensible mechanism by which many address formats can be identified and used.

Extensible Communication
TDI defines a TDI_ACTION IOCTL request that any TDI transport can use to support a set of transport-determined operations initiated by requests from its clients. This allows the client to submit transport-specific requests to the underlying transport driver that are not expressly defined by TDI.

Event Notification
TDI defines a scheme by which transports can notify their clients of interesting events that occur on the network without requiring the client to submit an explicit I/O request. For certain types of events, such as connects, disconnects, and receives, the transport driver can indicate data along with the event notification when it calls its client with such a notification.

Plug-and-Play Event Notifications
TDI defines a scheme by which Windows 2000 transports can notify their clients of certain PnP events, such as the availability of underlying NICs and the creation/deletion of connections on local network addresses.

System Power-State Change Notifications
TDI defines a scheme by which Windows 2000 transports can notify their clients of proposed system power-state changes, thereby giving the client an opportunity to keep an active connection powered up.

The following features can be supported by a TDI transport driver as additional options:

Data Transfer Modes
TDI supports sending and receiving data as discrete messages (message mode) or as a stream of bytes (byte-stream mode). Support for either mode (or both) depends on the transport driver writer and/or on the nature of the protocol.

Internal Buffering
TDI transports can buffer their client’s sends and receives internally. Transport-internal buffering allows the TDI client to set and query driver internal buffer sizes, request nonblocking send operations, receive notification of available buffer space, and look at (peek) buffered data before receiving it.

Management Options
All transports maintain some TDI-defined state about their respective features, limits, and runtime statistics. This allows each client to dynamically query and, in some cases to set, transport-provider static information, statistics, and configuration parameters. In addition, the extensible TDI_ACTION IOCTL, mentioned previously, allows any TDI transport driver to implement unique network-management features that can be accessed by its clients through action requests to the transport.

Quality of Service (QOS)
TDI transports can provide QOS negotiation upon establishment of a network connection. In addition, such a driver can support QOS for connectionless datagram transmission. To support
either, TDI-defined connection-establishment and datagram-send requests include Options and OptionsLength parameters that allow a TDI client to include a transport-specific, variable-length counted string specifying QOS options.

In fact, the Options and OptionsLength parameters can be used to pass any transport-specific connection- or datagram-related options to the transport driver at the discretion of the driver writer, not just QOS specifications.

**Expedited Delivery**
When sending messages, the client can flag particular messages as expedited. At the sending side, the underlying transport sends these messages ahead of non-expedited messages. At the receiving side, expedited messages are indicated to the client before and/or separate from non-expedited messages.

**Chained Receive Indications**
If the underlying NDIS miniport(s) support multipacket receive indications, a TDI transport can give its clients direct read-only access to a full TSDU in a single call, and the client can retain control of the buffer containing the TSDU until it has consumed the indicated data. This feature improves the performance of both transport and client by cutting down on call overhead for receive indications.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

### 3.2 TDI Device Objects

Windows 2000 transports support Plug and Play and system Power Management by creating device objects to represent their respective bindings to underlying NICs. Each such transport registers itself with TDI during system startup as a network provider to make the PnP-aware transport available to potential clients when the underlying network hardware comes online.

Potential client(s) of these PnP-aware transports register their respective ClientPnPXxx callbacks with TDI to receive notifications of the availability of active transport-to-NIC bindings and of transport-established network addresses on which these clients can subsequently send and receive data over the network while these network addresses remain valid.

As such a transport establishes a binding to an underlying NIC from the ProtocolBindAdapter function that the lowest module in the transport stack (or monolithic transport driver) registered with NDIS, the TDI transport driver creates a named device object to represent its binding to the underlying NDIS miniport. During system startup, such a PnP-aware transport registers each named device object it creates with TDI and sets up its per-binding state. The transport also registers any known network addresses on each such binding with TDI and sets up its per-registered-address state. When NDIS calls the ProtocolPnPEvent function at the bottom of the transport stack with the PnP event code NetEventBindsComplete, such a transport then notifies TDI if it is ready to transfer data over the network on at least one such transport-to-NIC binding.

TDI gives each network client that has registered its ClientPnPXxx handlers the opportunity to bind itself to each such transport-created device object by calling ZwCreateFile from the client’s registered ClientPnPBindingChange routine. TDI subsequently calls its registered
ClientPnPAddNetAddress routine one or more times with each network address that the underlying transport has already registered for its device object. TDI also notifies these clients with calls to their registered ClientPnPBindingChange routines when any particular transport indicates it is ready to carry out network transfers and again when all startup-time transport-to-NIC bindings have been established.

At runtime, these PnP-aware transports (or NDIS) continue to call TDI whenever a dynamic binding change occurs, whether because a NIC has been enabled/disabled or because the end user has initiated a binding change, and whenever such a transport makes or breaks a connection to a remote-node network address as follows:

- If a new NIC is enabled, NDIS calls the ProtocolBindAdapter function at the bottom of each transport stack, thereby giving that transport an opportunity to bind itself to the new NIC, to create another device object to represent its new binding, and to register its new device object with TDI. As each such transport-created device object is registered, TDI calls the set of currently registered ClientPnPBindingChange routines and offers these clients the opportunity to open the new binding(s).

- After registering a new device object, the transport also registers any known network addresses on its new binding with TDI. TDI calls the set of registered ClientPnPAddNetAddress routines when transports register known network addresses on their new (or existing) bindings. These calls also give clients the opportunity to open a connection to such a registered address (and even to a binding). Whenever such a transport breaks a connection to a remote node, it calls TDI to deregister the corresponding network address.

- Before an existing NIC is disabled, NDIS typically calls the ProtocolPnPEvent function at the bottom of the transport stack to ask whether it is safe to remove the NIC. The transport, in turn, notifies TDI of such a request to remove that NIC, and TDI notifies the clients with calls to their registered ClientPnPPowerChange routines. If no client objects to the NIC’s removal, NDIS subsequently calls the ProtocolUnbindAdapter function at the bottom of the transport stack and the transport tears down its binding. In this process, the transport calls TDI to deregister all the network addresses it formerly registered on the binding, and it also calls TDI to deregister the device object that it created to represent that binding. In turn, TDI notifies clients of these unbinding operations by making calls to their registered ClientPnPDelAddress and ClientPnPBindingChange routines.

- If the end user initiates a binding change in the network connections folder, NDIS calls TDI directly to notify registered network clients of the proposed binding change. If no client objects (and, therefore, the user is not prompted to cancel the binding-change operation), NDIS proceeds to call the appropriate ProtocolPnPEvent, ProtocolBindAdapter, or ProtocolUnbindAdapter function(s) at the bottom of the transport stack(s).

In a similar manner, TDI notifies the clients of PnP-aware transports about proposed system power-state changes for underlying NICs. When the system Power Manager calls NDIS to power down (or up) a NIC, NDIS calls the ProtocolPnPEvent function at the bottom of each transport stack that is currently bound to the NIC. Each such PnP-aware transport, in turn, notifies TDI of the proposed power-state change and TDI forwards the notification to the transport’s clients for their approval or rejection.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK
3.3 TDI File Objects

All I/O requests are directed to an open file object, which can represent a physical or virtual device, a data file, or any logical target of I/O requests, that can be opened by a particular process. Open file objects are process-specific. For example, the I/O Manager creates a file object for each process that opens a particular data file.

TDI uses file objects to represent entities that exist in any network communication environment. In particular, TDI uses file objects to represent the following types of TDI-defined entities:

- To represent transport addresses opened by specific processes or groups of processes on a network node
- To represent connection endpoints identifying specific endpoints associated with particular transport addresses, opened to set up endpoint-to-endpoint connections with remote-node peer processes
- To represent control channels identifying transport providers, opened for the purpose of setting and querying global configuration, features, limits, and/or statistics information maintained by the underlying transport

A TDI transport driver and the underlying NDIS drivers provide a mechanism by which data can be transferred from a process on one network node to one or more processes that reside either on the same node or other network nodes. The preceding address and connection endpoint entities represented by open file objects are the means by which such transfers occur.

For example, the Windows® 2000 redirector process opens two file objects, one representing a transport address and the other a connection endpoint associated with that address, to send SMB (Server Message Block) messages to a server process on a remote node and to receive messages back from the server process on the remote node.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

3.3.1 File Objects Represent Transport Addresses

TDI supports both unreliable connectionless and reliable connection-oriented data transfers.

Unreliable connectionless data can be sent to one or a group of processes that have opened a particular transport address on a remote node. When sending unreliable connectionless data, as datagrams, the sender need only identify the target remote-node address for the process or group of processes for them to receive each such datagram.

Reliable connection-oriented data transfers can be sent between two processes if an endpoint-to-endpoint connection (also called a virtual circuit) has been established between them. An endpoint-to-endpoint connection is a one-to-one association between two, and only two, processes. To establish such a connection, one process must identify the process with which it wants to establish the
connection. Each such process must have opened a transport address and a connection endpoint on its respective network node and associated its connection endpoint with the open transport address. For more information about connection endpoints, see Section 3.2.2.

The TDI entity used to identify a specific process, or a specific group of processes is one or more open process-specific file object(s) that represent a particular transport address. Such a file object contains transport-supplied pointers to driver-maintained state that identifies the specific process and the node on which that process resides. The state that a "routable" transport, such as TCP/IP, Mcsxns, AppleTalk and NWLink, maintains about such an address also contains information that identifies the network (or subnet) on which the node resides.

Certain TDI-defined transport address types accommodate explicit or implicit indications of whether the address identifies a single process (unique address), or can identify a group of processes (group address). In the case of a group address, the TDI-defined address can contain information that identifies the specific processes, that is, the node on which the process resides, and the network (or subnet) on which the node resides.

A number of address types are supported by TDI. The following describes the format and usage of three commonly used TDI address types:

- **TDI_ADDRESS_NETBIOS** – The NetBIOS-type address consists of a standard 16-character NetBIOS name, and a member that indicates if the name is registered (or to be registered) as a *unique* name (indicating that the name identifies a *single* process), or as a *group* name (indicating that the name can identify a *group* of processes).

  Since the transport driver ensures that a unique name is in use by only one process on the network at a time, such a name not only identifies the process, it also implicitly identifies the node on which that process resides.

  If the registered name is a group name, it can be an address available to many processes on many stations. Thus a group name identifies all processes that registered the name as well as the nodes on which those processes reside.

- **TDI_ADDRESS_IP** – The IP-type address consists of a port number and a standard Internet Protocol (IP) address. Since TCP/IP allows the same port number to be registered by processes on many nodes, the IP address is required to uniquely identify the node and the port number is required to uniquely identify the process on that node.

  For connectionless data transfer (using the TCP/IP UDP protocol), the same port number can be registered by many processes on the same station. In addition, certain IP addresses can be used by more than one node. Data sent to a TDI address that consists of this type of port number and IP address will be received by all nodes to which the IP address applies; on those nodes, the data will be passed to all processes that have registered the specified (UDP) port number.

- **TDI_ADDRESS_IPX** – The IPX-type address consists of a four-byte network number, a six-byte node number, and a two-byte port number. Since IPX allows the same port number to be registered by processes on many nodes, an IPX address is required to uniquely identify the network and the node, while the port number can be used to uniquely identify the process on that node.
3.3.2 File Objects Represent Connection Endpoints

A file object that represents an open connection endpoint identifies a specific connection on which local and remote-node peer processes are communicating with each other across the network or will communicate with each other when the local-node process has established an endpoint-to-endpoint connection with a remote-node peer.

Any such file object must be associated with another open file object that represents a particular transport address. The transport address identifies the process, as already described in Section 3.2.1.

In a similar manner, the transport uses the file object representing the local connection endpoint to maintain state about the remote-node peer process, such as the remote-node transport address, with which its local-node client establishes an endpoint-to-endpoint connection.

A client process can establish many endpoint-to-endpoint connections between itself and other processes on remote nodes. For example, the redirector establishes separate connections between itself and each remote Windows 2000 server with which it communicates. Such a client can have several open connection endpoints all separately associated with an open transport address.

When a process wants to send data over an endpoint-to-endpoint connection, it must have a means of identifying which of all existing connections the data is to be sent over. Each open file object that represents a connection endpoint is used to discriminate among established endpoint-to-endpoint connections of the same process.

3.3.3 File Objects Represent Control Channels

A network management process can query or set global configuration or statistical information with respect to a specific transport provider. To do so, such a process must have a means of identifying a particular transport among several possible transport providers.

Such a client opens a file object that represents a control channel so that it can query the appropriate transport. In effect, the client opens a file object that represents the device object created by the underlying transport driver when the client opens a control channel, either by calling ZwCreateFile or IoGetDeviceObjectPointer.

Other TDI clients that send and receive data across the network also can open a control channel to query their underlying transports. For example, such a client might issue a query to determine the transport’s limit on datagram size so the client could effectively size the buffers that it will allocate to send and receive datagrams subsequently.
In general, a TDI transport maintains global state information about its features and current statistics, rather than process-specific state information tied to a particular open file object. For example, a client that queries the current state of its open transport address passes a pointer to a client-specific file object representing that address, but the transport returns information common to all clients that currently have the same address open.

Network Drivers: Windows 2000 DDK

### 3.4 TDI Transport Driver Routines

Every TDI-compliant transport driver is a standard intermediate driver that must export a number of entry points called by the I/O Manager.

Some of a TDI transport’s standard driver routines initialize and unload the driver itself. Others are standard Dispatch routines that the I/O Manager calls when TDI clients make calls to system support routines, such as `ZwCreateFile` and `IoCallDriver`.

Like other kernel-mode drivers, a TDI transport’s `DriverEntry` routine sets up one or more driver-supplied Dispatch routines to handle various types of I/O requests passed in as IRPs. A TDI driver can export a single Dispatch routine to handle all incoming IRPs or a separate DispatchXxx routine to handle each IRP_MJ_XXX the driver must support. General requirements for Dispatch routines are discussed in detail in the *Kernel-mode Driver Design Guide*. TDI-specific requirements for Dispatch routines are summarized later in Chapter 4 and described in the *Network Driver Reference*.

As a TDI transport completes an operation requested by its client, the I/O Manager calls any client-supplied IoCompletion routine that the client set in the IRP before that client submitted it to the underlying transport.

In addition, such a transport driver must call the TDI client at preregistered entry points within the TDI client’s code when specific network events occur. These client-supplied event handlers also are summarized later in Chapter 4 and described in the *Network Driver Reference*.

At its lower edge, a TDI driver that is monolithic must export a set of ProtocolXxx functions to be called by the NDIS library on behalf of underlying NDIS intermediate and/or NIC drivers. These NDIS driver lower-edge functions are described earlier in this manual and in a function-specific manner in the *Network Driver Reference*.

Network Drivers: Windows 2000 DDK

### 3.5 TDI Kernel-Mode Client Interactions

The following figure shows how TDI clients make I/O requests to their underlying TDI transports and...
The following figure shows how TDI clients make I/O requests to their underlying TDI transports and how transports make callbacks to their clients.

Figure 3.2 TDI client/transport interactions

A TDI client interacts with its underlying transport driver as follows:

- **Creating TDI File Objects** – A client calls `ZwCreateFile` to create or open a file object that represents a transport address, a connection endpoint, or a control channel. This call causes the I/O Manager to allocate an IRP, to marshal the client-supplied parameters into the IRP, and to pass the IRP to the underlying transport driver’s `TdiDispatchCreate` routine. When the transport driver has set up all the state it maintains for the newly created file object, it calls `IoCompleteRequest` (or `TdiCompleteRequest`) with the IRP and STATUS_SUCCESS. `ZwCreateFile` then returns to the TDI client with a handle to the file object.

Each client process’s call to `ZwCreateFile` creates a separate file object, even if two clients have specified the same transport address in their calls to `ZwCreateFile`.

A successful call to `ZwCreateFile` opens a transport address, a connection endpoint, or a control channel, depending on the `EaXxx` parameters the client passes in its call.

- **Submitting Requests** – After the appropriate file object(s) have been created, the client can submit requests that reference those objects. For example, after it opens a file object that represents a particular transport address, the client can submit an address-information query or "send datagram from this address" request.

Such a client uses standard I/O system mechanisms and conventions to submit all such requests:

- The client prepares an IRP with an IRP_MJ_XXX opcode that identifies what operation the client wants the transport driver to perform. The client supplies all appropriate parameters for the given IRP_MJ_XXX, and, optionally, sets up its own `IoCompletion` routine that will be called when the request is completed by the transport.
The Windows® 2000 DDK includes a set of TdiBuildXxx macros (in tdikrnl.h) that can be linked into client code and used to prepare IRPs for TDI-defined IOCTL requests, as well as the TdiBuildInternalDeviceControl function to allocate such an IRP.

When the IRP has been set up, the client calls IoCallDriver with pointers to the IRP, to the file object that represents the address, connection endpoint, or control channel, and to the transport driver’s device object. The I/O Manager passes the IRP directly to the appropriate TdiDispatch(Xxx) routine of the transport driver.

When the transport driver has completed the requested operation, it calls TdiCompleteRequest or IoCompleteRequest. The I/O Manager then calls the client-supplied IoCompletion routine if the client supplied one for the IRP.

Handling Event Notifications – If a client has preregistered its entry point(s) for one or more event handlers, the transport driver calls each such routine when the corresponding network event occurs. For example, if the client has registered a ClientEventReceive handler on an address associated with an endpoint-to-endpoint connection, the transport calls this handler when data sent by the remote-node peer process is received on the local node.

Deleting TDI Objects – The client calls ZwClose to delete a file object when the address, connection endpoint, or control channel is no longer needed by the client. The close request is forwarded to the transport’s TdiDispatchCleanup and, subsequently, TdiDispatchClose routines.

For more information about the kernel-mode support routines mentioned here, see the Kernel-Mode Driver Reference. For detailed information about the TdiXxx functions, TdiBuildXxx macros, and ClientEventXxx routines mentioned here, see the Network Driver Reference.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

3.6 TDI Requests versus Events

Many low-level network interfaces, such as NetBIOS and Windows Sockets, are primarily one-way. A client can call the underlying transport driver whenever it wants, but the transport driver cannot call the client. The only way such a transport driver can "say anything" to its client is by returning a specific error code on return from a client-initiated request. In the NetBIOS interface, for example, a transport informs its clients of a connection "hangup" by returning pending NCB_Receives (or an NCB_Receive_Any) with a specific error code (0x0A for a session being closed, for example).

TDI provides an event-notification mechanism that allows a TDI transport to call its affected kernel-mode clients whenever a specific network event occurs (receipt of a datagram, for example) if each such client has registered its ClientEventXxx handler with the transport for that type of event. The TDI client-supplied callback then takes appropriate action and returns control to the transport driver.

TDI also defines a means for PnP-aware Windows 2000 transports to notify their clients whenever such a transport binds itself to or unbinds itself from an underlying NIC, whenever the transport establishes or breaks a connection to a remote node on any particular binding, and whenever a system power-state change is proposed (and/or effected) by the Power Manager. The clients of such a transport register a set of ClientPnPXxx handlers with TDI to receive these types of notifications, as
transport register a set of Client PnP handlers with TDI to receive these types of notifications, as already mentioned in Section 3.2.

When a TDI transport driver calls a client’s registered ClientEventXxx handler, it can pass a limited amount of data as a parameter of that call. This feature allows the client to receive messages from the transport without having to allocate a buffer.

For example, the redirector takes advantage of this TDI feature. Many of the SMB requests that the redirector sends to the server (such as write SMBs) require little more than the standard SMB header to be present in the corresponding SMB response. The length of this header, which includes status indicators, multiplex ID, etc., is quite small, typically less than 100 bytes.

When the underlying TDI transport driver calls the redirector’s registered ClientEventReceive handler with such an SMB response message to a preceding send, the redirector need only view (not necessarily copy) the indicated message, note the SMB response status indicator, and return to the transport driver. In such a transaction, the redirector receives the SMB response message without having to allocate a buffer.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

Chapter 4 TDI Routines, Macros, and Callbacks

This chapter summarizes the routines and requests particular to TDI transport drivers. It also summarizes callback routines exported by TDI kernel-mode clients, as well as the macros and TDI functions provided by the system for transports and clients to use.

See the following topics for summaries about each category of information:

- Transport Drivers
  - TDI Driver Initialization
  - TDI Driver Dispatch Routines
- The set of TDI-defined IOCTLs that kernel-mode clients can issue to their underlying transports
  - TDI IOCTL Requests
- The event handlers that clients can register with their underlying transports
  - TDI Client Callbacks
- The set of system-supplied functions and macros that TDI transports and their clients can use
  - TDI Library Functions and Macros
For more information about the sequence of interactions between TDI clients and transports when the preceding are used, see Chapter 5.

For detailed information about each request, macro, or routine, see the Network Driver Reference.

Network Drivers: Windows 2000 DDK

4.1 TDI Driver Initialization

Every TDI transport driver must provide an initialization routine explicitly named DriverEntry. In addition, a transport driver should have as many TdiDispatchXxx routines and internal driver functions as it needs to satisfy the I/O requests of its kernel-mode clients. A TDI transport driver or one of the underlying protocol drivers in its transport stack also must provide the NDIS driver lower-edge functions described earlier in Chapter 2 and in the Network Driver Reference.

Every transport driver must have a DriverEntry routine to be called by the I/O Manager when the transport is loaded. DriverEntry must be declared in the driver code to make the transport automatically loadable.

When it loads the driver, the I/O Manager creates a driver object to represent the transport and passes a pointer to the driver object when it calls the TDI transport’s DriverEntry routine. Then, its DriverEntry routine does the following:

- Possibly, reads the registry (using kernel-mode support routines) to retrieve configuration information written into the registry by the transport’s INF file. Using the registry information, configure itself as needed.
- Sets all the driver’s TdiDispatchXxx entry points in the driver object. The driver’s TdiDispatchXxx routines will be called later by the I/O Manager to handle requests from TDI clients.
- Calls TdiRegisterProvider to notify TDI of the transport’s Plug and Play and power management support.
- Creates at least one named device object for itself with IoCreateDevice. The names of each driver’s device objects are the device names stored under the transport driver’s registry section Linkage key in the Export entry. Each transport driver’s Export entry determines what device object(s) will be created by any particular transport driver during system setup.

Some transport drivers, such as NWLink, create only one named device object no matter how many NICs they bind themselves to. Some transports, such as TCP/IP, create one or more named device objects and "export" a set of device interfaces for each such device object. Other transport drivers create a separate named device object for each bound NIC, such as Nbf_Elnki1, Nbf_Elnki2, and so forth.
- Performs any other necessary initialization tasks, such as binding to underlying NDIS intermediate and/or NIC drivers.
- As it establishes bindings to each underlying NIC, a PnP-aware Windows 2000 transport also calls TdiRegisterDeviceObject with the named device object that the transport creates. Then,
it makes one or more calls to TdiRegisterNetAddress to register all known network addresses (whether of the local machine or of remote nodes) associated with the transport-created device object for its newly established binding.

When such a PnP-aware transport has completed its initialization on at least one but possibly more bindings so that it is capable of transferring data over the network, the transport must call TdiProviderReady. Typically, such a PnP-aware TDI transport calls TdiProviderReady as a consequence of an NDIS call to the ProtocolPnPEvent function at the lower edge of the transport stack with the input NET_PNP_EVENT code NetEventBindsComplete.

A TDI transport driver must provide an Unload routine unless that driver cannot be unloaded without rendering the system unusable. Most TDI transport drivers declare an Unload routine, and their DriverEntry routines set its entry point in the driver object. For more information about transport Unload routines, see Section 4.1.2.

For more information about the kernel-mode support routines and objects used by TDI drivers, see the Kernel-mode Driver Reference.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

4.1.1 Registering a TDI Transport Driver

When TDI is loaded, the configuration registry has entries for each TDI transport driver and kernel-mode client operating on a particular network. Using binding information in the registry, the operating system loads network drivers.

Then, the I/O Manager creates driver objects and calls the DriverEntry routine of each TDI transport driver. The I/O Manager calls each transport driver’s DriverEntry routine until all TDI drivers are loaded and registered on the network.

A TDI transport driver or a lower protocol component in its transport stack must register with the NDIS library by calling NdisRegisterProtocol if it will bind itself to an underlying NDIS NIC driver. As NDIS is called by the PnP Manager to initialize each NIC found in the machine, NDIS calls the registered ProtocolBindAdapter functions at the lower edge of the appropriate TDI transport driver(s) or transport stack(s) to give each transport the opportunity to bind itself to the appropriate NDIS miniport(s).

When a TDI transport driver is bound to an NDIS miniport or has layered itself above an NDIS protocol driver that has bound itself to such a miniport, the TDI driver should be ready to respond to any registered TDI client that submits a request to open any network address that the TDI transport has already registered with TDI on that binding.

Consequently, a PnP-aware TDI transport must not call TdiProviderReady until it has set up its per-device-object and per-address state for each initialization-time call that it makes to TdiRegisterDeviceObject and TdiRegisterNetAddress.
Note that such a TDI transport calls \texttt{TdiRegisterProvider} and \texttt{TdiProviderReady} only once each during system startup. However, the transport can make many runtime calls to the Windows 2000 TDI-provided PnP and power-management functions, such as \texttt{TdiRegister/DeregisterDeviceObject}, \texttt{TdiRegister/DeregisterNetAddress}, and \texttt{TdiPnPPowerRequest}.

For more information about Windows 2000 PnP and power-management support, see the \textit{Setup, Plug and Play, & Power Management Design Guide}, as well as the particulars of how the NDIS library indicates PnP and power events earlier in this \textit{Network Driver Design Guide}.

### 4.1.2 Unloading and Deregistering a TDI Transport Driver

Before a NIC is removed from the machine, Windows® 2000 TDI transports bound to that NIC are usually given the opportunity to first notify their clients that the NIC might be disabled and removed. That is, unless a "surprise" removal of a NIC occurs, NDIS calls the \texttt{ProtocolPnPEvent} function at the lower edge of each transport bound to that NIC with the input \texttt{NetEventQueryRemoveDevice} code so that each such TDI transport can forward this notification to its clients by calling \texttt{TdiPnPPowerRequest}.

Such a PnP-aware TDI transport does not actually tear down its binding or make calls to \texttt{TdiDeregisterNetAddress} and \texttt{TdiDeregisterDeviceObject} until NDIS has called the \texttt{ProtocolUnbindAdapter} function at the lower edge of the transport stack. Furthermore, a transport driver cannot be unloaded until it has released all its bindings to all underlying NICs.

However, after a NIC is removed from the machine, the system can unload a TDI transport driver that was either bound directly only to that underlying NIC’s driver or layered over an NDIS protocol driver that was bound to that NIC driver. To accomplish this, the system calls the TDI transport’s Unload routine.

The transport’s Unload routine will be called by the I/O Manager when the Service Controller unloads the driver from memory. A transport driver’s Unload routine frees all remaining driver-allocated resources.

A PnP-aware TDI transport driver must make a reciprocal call to \texttt{TdiDeregisterProvider} from its Unload routine. If the TDI driver exports a set of NDIS-defined ProtocolXxx functions, it also must call \texttt{NdisDeregisterProtocol} from its Unload routine.

For more information about standard intermediate driver Unload routines, see also the \textit{Kernel-Mode Driver Design Guide}. For more information about Windows 2000 PnP and power-management support, see the \textit{Setup, Plug and Play, & Power Management Design Guide}, as well as the particulars of how the NDIS library indicates PnP and power events earlier in this \textit{Network Driver Design Guide}.
4.2 TDI Driver Dispatch Routines

I/O requests are formatted as IRPs either by the I/O Manager or explicitly by a TDI client and submitted to the transport driver by calling IoCallDriver. Completed IRPs are returned to the caller when the transport driver calls IoCompleteRequest or TdiCompleteRequest. Any kernel-mode TDI client can set its IoCompletion routine for an IRP before it calls IoCallDriver.

There are five IRP_MJ_XXX codes used to send I/O requests to TDI transport drivers. These transports handle incoming IRPs in which the MajorFunctionCode is one of the following:

**IRP_MJ_CREATE**
- Opens a named device object created by the transport and, subsequently, opens a transport address, connection endpoint, or control channel in the underlying transport.

  This request originates in a client’s call to ZwCreateFile.

**IRP_MJ_INTERNAL_DEVICE_CONTROL**
- Specifies kernel-mode client requests (TDI IOCTLs), for which internal transport functions handle operations other than opening and closing file objects.

  This request usually originates in a client’s call to TdiBuildXxx macro followed by its call to IoCallDriver.

**IRP_MJ_DEVICE_CONTROL**
- Specifies user-mode-visible IOCTL requests issued by a transport-dedicated application. Except for any transport-defined "private" IOCTLs, such requests usually are forwarded to the same internal driver functions that handle internal-device control requests.

  This request originates in a call by a transport-dedicated user-mode application to DeviceIoControl.

**IRP_MJ_CLEANUP**
- Closes an open address, connection endpoint, or control channel when the NT executive is closing the last handle for the corresponding file object.

  This request originates in a client’s call to ZwClose.

**IRP_MJ_CLOSE**
- Closes an address, connection endpoint, or control channel if the executive is removing its last reference to the file object handle. When no outstanding file object handles are open, closes a transport-created device object.

  This request follows an IRP_MJ_CLEANUP request on the same file object.

The entry points in the TDI driver are one or more Dispatch routines that handle these IRP_MJ_XXX requests. Because a TDI client communicates with the driver only through IRPs, the driver has one or more TdiDispatchXxx routines that determine what operation to carry out. Usually, these TdiDispatchXxx routines pass the client requests to appropriate internal driver functions for further processing.

A TDI transport driver exports all its TdiDispatchXxx entry points by setting them in the driver object.
A TDI transport driver exports all its transport-specific entry points by setting them in its driver object passed in to its **DriverEntry** routine. The I/O Manager calls a TdiDispatchXxx routine whenever a client makes an I/O request. A transport driver can have a separate TdiDispatchXxx to handle each of the possible IRP_MJ_XXX opcodes, a single TdiDispatch routine that processes IRPs with all possible IRP_MJ_XXX opcodes, or a number of TdiDispatchXxx that handle discrete subsets of the IRP_MJ_XXX opcodes.

Since all Dispatch entry points are exported by address in the driver object, not by name, a TDI transport driver writer can name these routines anything. In this TDI documentation, TdiDispatchXxx routines have the following metanames, each describing their basic functionality, to correspond to the preceding IRP_MJ_XXX:

- **IRP_MJ_CREATE**
  - TdiDispatchCreate
- **IRP_MJ_DEVICE_INTERNAL_CONTROL**
  - TdiDispatchInternalDeviceControl
- **IRP_MJ_DEVICE_CONTROL**
  - TdiDispatchDeviceControl
- **IRP_MJ_CLEANUP**
  - TdiDispatchCleanup
- **IRP_MJ_CLOSE**
  - TdiDispatchClose

4.3 TDI IOCTL Requests

As already mentioned in the preceding section, kernel-mode clients pass TDI IOCTLs in IRPs to the underlying transport driver with calls to **IoCallDriver**. These requests are passed with a **MajorFunctionCode** of IRP_MJ_INTERNAL_DEVICE_CONTROL and **MinorFunctionCode** that specifies one of the following TDI-defined operations:

- **TDI_ACCEPT**
  - Allows the local-node client to accept an incoming connection offer from a remote-node peer so that it can send and receive connection-oriented data on the network.
- **TDI_ACTION**
  - Initiates transport-specific extension operations for an address, a connection endpoint, or a control channel.
- **TDI_ASSOCIATE_ADDRESS**
  - Associates an open connection endpoint with an open local transport address, which is a prerequisite to establishing an endpoint-to-endpoint connection with a remote-node peer process.
- **TDI_CONNECT**
  - Initiates a connection offer from one endpoint to another on a remote node.
- **TDI_DISASSOCIATE_ADDRESS**
  - Disassociates a connection endpoint for an inactive connection from its associated address.
Disassociates a connection endpoint for an inactive connection from its associated address object.

**TDI_DISCONNECT**
Terminates an established endpoint-to-endpoint connection for the local-node client or acknowledges connection termination initiated by a remote-node peer.

**TDI_LISTEN**
Passively listens for an incoming connection offer from a remote-node peer process.

**TDI_QUERY_INFORMATION**
Queries the underlying TDI transport for a specific type of information.

**TDI_RECEIVE**
Returns all or part of a TSDU received on an endpoint-to-endpoint connection.

**TDI_RECEIVE_DATAGRAM**
Returns a datagram TSDU on an open transport address.

**TDI_SEND**
Sends a normal or expedited TSDU to the remote-node peer on an endpoint-to-endpoint connection.

**TDI_SEND_DATAGRAM**
Sends a datagram to a specified remote-node address.

**TDI_SET_EVENT_HANDLER**
Registers a ClientEventXxx handler that the transport will call whenever the corresponding network event occurs.

**TDI_SET_INFORMATION**
Sets a specific type of information in the underlying transport.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

### 4.4 TDI Client Callbacks

Most TDI clients' routines are highly variable and environment-dependent. However, TDI defines a set of callback routines that clients can register with the underlying TDI transport to receive notifications when network events of interest occur. A kernel-mode TDI client can register any of these ClientEventXxx routines if the TDI driver is to notify it of a particular type of network event occurrence. The client can even use such a registered ClientEventXxx handler as an alternative to issuing certain TDI_XXX IOCTL requests to the driver.

Like the TdiDispatchXxx routines described already, the client-supplied event handlers can have any names the client writer chooses. Each ClientEventXxx mentioned here is registered as an entry point passed in a client-supplied IRP submitted to the TdiDispatchInternalDeviceControl routine.

A TDI client can have one or more of the callbacks listed next, registered at the beginning of network operations on an open transport address with the TDI transport driver by issuing successive TDI_SET_EVENT_HANDLER requests, set up with TdiBuildSetEventHandler:

- **ClientEventConnect**
  Notifies the client of a connection offer from a remote-node peer process.

- **ClientEventDisconnect**
  Notifies the local-node client that its remote-node peer is terminating the endpoint-to-endpoint
Notifies the local-node client that its remote-node peer is terminating the endpoint-to-endpoint
connection between them.

**ClientEventError** and/or **ClientEventErrorEx**
Notifies the client of an error condition in its underlying transport driver, in one of the lower
protocol layers of the transport stack, or in a still lower NDIS driver that has made subsequent
I/O on the client’s open transport address unreliable or even impossible.

**ClientEventReceive**
Notifies the client that its transport has received normal data from the remote-node peer on an
endpoint-to-endpoint connection and makes all or part of the received data available to be
copied by the client.

**ClientEventChainedReceive**
Notifies the client that its transport has received a normal TSDU from the remote-node peer on an
endpoint-to-endpoint connection and makes the full TSDU available for consumption by the
client.

**ClientEventReceiveExpedited**
Notifies the client that its transport has received expedited data from the remote-node peer on an
endpoint-to-endpoint connection and makes all or part of the received data available to be
copied by the client.

**ClientEventChainedReceiveExpedited**
Notifies the client that its transport has received an expedited TSDU from the remote-node peer on an
endpoint-to-endpoint connection and makes the full TSDU available for consumption by the
client.

**ClientEventReceiveDatagram**
Notifies the client that the TDI driver has received a datagram directed to a transport address
that the client has opened.

**ClientEventChainedReceiveDatagram**
Notifies the client that the TDI driver has received a datagram directed to a transport address
that the client has opened and makes the full TSDU available for consumption by the client.

**ClientEventSendPossible**
Notifies the client that the transport, which buffers send data internally, again has buffer space
available for sends.

Windows 2000 TDI clients also can register a set of **ClientPnPXxx** callbacks with TDI to receive
notifications of dynamic PnP binding-change, net-address-change, and power-state-change events
from PnP-aware TDI transports. Like the **ClientEventXxx** handlers, these client-supplied PnP event
handlers can have any names the client writer chooses.

Such a Windows 2000 TDI client has the set of callbacks listed next, which are registered with a call
to **TdiRegisterPnPHandlers** when the client initializes itself:

**ClientPnPBindingChange**
During system startup, TDI calls this routine to notify the client of all available transport-
created named device objects that represent transport-to-NIC bindings, again when each such
PnP-aware TDI transport indicates its readiness to carry out network data transfers on its
binding(s), and again when all possible TDI transport-to-NIC bindings have been established.

During runtime, TDI calls this client routine whenever a PnP-aware transport registers another
device object to represent a newly established transport-to-NIC binding or deregisters an
existing device object while tearing down a binding. TDI also calls this client routine if the end
user makes binding-list-order changes in the network connections folder.


ClientPnPAddNetAddress

During system startup, TDI calls this routine to notify the client of all network addresses currently registered with TDI by PnP-aware transports on their established bindings. Subsequently, TDI calls this client whenever a PnP-aware transport registers a new network address (usually for a newly established connection to a remote node) on an established binding.

ClientPnPDelNetAddress

TDI calls this routine to notify the client whenever a PnP-aware TDI transport deregisters an existing network address (usually when breaking a connection to a remote node) on an established binding.

ClientPnPPowerChange

TDI calls this routine to notify the client of proposed NIC removals and of proposed system-power-state changes that could affect that client’s ability to transfer data over the network on a particular binding.

A client that registers such a set of ClientPnPxXx handlers with TDI typically maintains dynamic state, usually as one or more internal lists, about all valid network addresses on the bindings of all TDI transports it uses to communicate over the network.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

4.5 TDI Library Functions and Macros

Windows® 2000 provides a small set of TDI library functions as a kernel-mode dynamic-link library (the tdi.sys export library) with which TDI drivers and kernel-mode clients link themselves. However, most of the system-supplied TdiBuildxx called by kernel-mode clients are implemented as macros in the tdikrn.h header that both clients and transports include.

Any kernel-mode TDI client can use the TdiBuildxXx macros as needed in preparing the TDI XXX IOCTL IRPs, already mentioned in Section 4.3. After the client has set up such an IRP with a TdiBuildxx macro, it submits the request to the underlying TDI driver by passing the IRP to IoCallDriver. Each of these macros fills in the relevant members of the client-provided IRP, except for the Status and Information members, which the underlying transport fills in after processing its client’s request. Before using one of TdiBuildxXx IOCTL macros, a client can call TdiBuildInternalDeviceControlIrp to allocate the IRP if it has not already been allocated by a still higher level network component and given to the client.

A transport driver never uses the TdiBuildxXx macros. However, a TDI driver can use some of the remaining TDI library routines, such as TdiCompleteRequest and TdiCopyBufferToMdl, for assistance in processing client requests.

TDI library routines and macros include the following:

TdiBuildInternalDeviceControlIrp

Allocates an IRP if the client does not receive an IRP from a higher network layer.

TdiBuildAccent
NDIS Intermediate Drivers

TdiBuildAccept
Sets up an IRP for a client-submitted TDI_ACCEPT request.

TdiBuildAction
Sets up an IRP for a client-submitted TDI_ACTION request.

TdiBuildAssociateAddress
Sets up an IRP for a client-submitted TDI_ASSOCIATE_ADDRESS request.

TdiBuildConnect
Sets up an IRP for a client-submitted TDI_CONNECT request.

TdiBuildDisassociateAddress
Sets up an IRP for a client-submitted TDI_DISASSOCIATE_ADDRESS request.

TdiBuildListen
Sets up an IRP for a client-submitted TDI_LISTEN request.

TdiBuildQueryInformation
Sets up an IRP for a client-submitted TDI_QUERY_INFORMATION request.

TdiBuildReceive
Sets up an IRP for a client-submitted TDI_RECEIVE request.

TdiBuildReceiveDatagram
Sets up an IRP for a client-submitted TDI_RECEIVE_DATAGRAM request.

TdiBuildSend
Sets up an IRP for a client-submitted TDI_SEND request.

TdiBuildSendDatagram
Sets up an IRP for a client-submitted TDI_SEND_DATAGRAM request.

TdiBuildSetEventHandler
Sets up an IRP for a client-submitted TDI_SET_EVENT_HANDLER request.

TdiBuildSetInformation
Sets up an IRP for a client-submitted TDI_SET_INFORMATION request.

TdiBuildNetbiosAddress
Sets up a NetBIOS address for a client.

TdiBuildNetbiosAddressEa
Sets up a buffered NetBIOS address that a client can pass subsequently to ZwCreateFile to open the address.

TdiReturnChainedReceives
Relinquishes control of the buffer that was passed to a ClientEventChainedReceive(Xxx) handler after the client has consumed the received TSDU.

TdiCopyBufferToMdl
Copies a range of buffered data into a destination buffer mapped by a given MDL.

Both clients and transports can use this function.

TdiCopyMdlToBuffer
Copies data from buffer(s) mapped by a given MDL into a caller-supplied destination buffer.

Both clients and transports can use this function.

TdiCopyLookaheadData
Safely copies received data indicated to the transport by a NIC driver, whatever the nature of the memory (including mapped device memory) that the NIC is using.

TdiMapUserRequest
Converts an IRP passed in to a transport’s TdiDispatchDeviceControl routine into a TDIXXX IOCTL IRP if TdiMapUserRequest recognizes the minor function code specified in the input IRP.

TdiCompleteRequest
1.0 NDIS Intermediate Drivers

Completess an IRP with the system-defined network-specific priority boost for a transport driver.

**TdiRegisterPnPHandlers**
Registers a set of ClientPnPXX routines with TDI to receive notifications of PnP binding-change, network-address-change, and power-state-change events from underlying PnP-aware TDI transports.

**TdiDeregisterPnPHandlers**
Removes a set of ClientPnPXX routines from being notified about subsequent PnP binding-change, network-address-change, and power-state-change events.

**TdiRegisterProvider**
Registers a Windows 2000 PnP-aware transport driver with TDI during system startup.

**TdiProviderReady**
Indicates a PnP-aware transport driver’s readiness to carry out network I/O on one or more newly established transport-to-NIC bindings.

**TdiDeregisterProvider**
Deregisters a PnP-aware transport driver just before it unloads.

**TdiRegisterDeviceObject**
Registers a named, transport-created device object that the clients of such a PnP-aware transport can subsequently "open" for network I/O. In effect, such a transport’s call to this TDI function notifies all potential clients of this transport that it has just established a new transport-to-NIC binding.

**TdiDeregisterDeviceObject**
Deregisters a transport-created device object when a PnP-aware transport is tearing down a binding.

**TdiRegisterNetAddress**
Registers a valid network address that a PnP-aware transport has just created for the local machine or has just obtained when establishing a connection to a remote node on a particular binding. In effect, a PnP-aware transport’s call to this TDI function notifies its potential clients that they (or their remote-node peers) can subsequently use this protocol-specific address for network communications.

**TdiDeregisterNetAddress**
Removes a network address from the list of those valid on a particular PnP-aware transport-to-NIC binding.

**TdiPnP PowerRequest**
Forwards certain NET_PNP_EVENT-type notifications (by NDIS to the lower edge of the transport stack) to a PnP-aware transport’s clients. In effect, a PnP-aware transport’s call to this TDI function gives its clients the opportunity to notify their own still higher level clients about binding changes or power-state changes to the underlying NIC on which the transport depends.

**TdiPnP PowerComplete**
Indicates that a client has completed its processing of a binding-change or power-state-change notification for which it previously returned STATUS_PENDING.

**TdiEnumerateAddresses**
Returns a list of all valid network addresses to the calling client. (This TDI function is almost never called because it has a severe adverse effect on the caller’s performance.)

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK
Chapter 5 TDI Operations

This describes the basics of TDI runtime operations. Clients of Windows 2000 PnP-aware transports use the protocol-specific network addresses currently registered with TDI to establish connections across the network when they begin the operations described here. For more information about PnP-aware TDI transports and their clients, see also the preceding Chapters 3 and 4.

Many TDI runtime operations involve an endpoint-to-endpoint connection between a local-node client and a remote-node client. These TDI operations are connection-oriented and include the following:

- Opening a local-node connection endpoint
- Requesting a connection
- Accepting a connection
- Sending connection-oriented data
- Receiving connection-oriented data
- Disconnecting
- Closing a local-node connection endpoint

TDI also permits connectionless communications between nodes on a network. This type of operation does not require the local-node client to establish an endpoint-to-endpoint connection with a remote-node client. They are faster but less reliable than connection-oriented communications. TDI connectionless operations include the following:

- Sending a datagram
- Receiving a datagram

The remaining TDI operations are common to connection-oriented and connectionless communication, including the following:

- Opening a local-node transport address represented by a file object
- Packaging and submitting a request
- Setting and querying information
- Receiving error notification
- Requesting transport-specific actions if the underlying transport supports any extensions to the TDI interface for its clients
- Closing a previously opened transport address

The following sections describe the main TDI operations, as much as possible in the same order as these operations are likely to occur. See the Network Driver Reference, Part 3, for more detailed information about the TDI, transport, and client routines mentioned here.

See the Kernel-Mode Driver Reference for more detailed information about the support routines, such as ZwCreateFile and IoCallDriver, mentioned here.

Built on Wednesday, June 28, 2000
5.1 Opening a Transport Address

Figure 5.1 shows how a kernel-mode client opens a transport address in its underlying transport driver.

After opening a transport-created device object for a binding, a TDI client usually begins communicating with its local-node transport by opening a file object that represents a transport address. To do this, the client calls ZwCreateFile, passing the address specification in the EA (extended attributes) buffer parameter to ZwCreateFile. Within the EA buffer, the client sets the EaName member to the system-defined value TdiTransportAddress followed by an EA value of the TDI-defined type TRANSPORT_ADDRESS, set up by the client to specify the transport-specific address to be opened.

The client can specify that a particular transport address be either sharable with other clients or exclusive to the client when it calls ZwCreateFile. The initial client to successfully open a particular transport address for exclusive use prevents other clients from opening the same address until the original client closes that address.

To do subsequent connectionless communication over the network, the client can specify the underlying transport’s broadcast address in the EA buffer, assuming the transport supports broadcast datagrams. Such an address cannot be opened for exclusive use by any client.

The client’s call to ZwCreateFile causes the I/O Manager to create a client-process-specific file object to represent the address and to call the TDI transport driver’s TdiDispatchCreate routine with an IRP containing the client-supplied parameters to ZwCreateFile. TdiDispatchCreate parses the EA information, and the transport sets up internal state for the open address and for this client if the call succeeds.

After its successful call to ZwCreateFile returns a file handle to the client and the client has obtained a pointer to the file object with ObReferenceObjectByHandle, it is ready to make certain TDI_XXX IOCTL requests to its underlying transport. For example, the client can send or receive datagrams on such an open address.
Usually, a client must first decide whether to register one or more of its ClientEventXxx handlers and whether it will use the open address to communicate with a remote-node peer process, in which case the client also must open a connection endpoint.

If the client wants to receive notifications of various network events, it can register its ClientEventXxx handler for each type of event by submitting one or more TDI_SET_EVENT_HANDLER requests, set up with TdiBuildSetEventHandler, to the underlying transport. For more information about setting up IOCTL requests and submitting them to an underlying transport, see Section 5.3.

After the client has registered its ClientEventXxx handlers, if any, on the open transport address, it is ready to make preparations for connection-oriented communication by opening a connection endpoint, as described in Section 5.2, or to begin connectionless communication, as described in Section 5.7.

When it is done with network communications on the opened transport address, the client must close the file object that represents that transport address by calling ZwClose with the handle returned by ZwCreateFile, as described later in Section 5.13.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

## 5.2 Opening a Connection Endpoint

Figure 5.2 shows how a kernel-mode client opens a connection endpoint in its underlying transport driver.

![Figure 5.2 Opening a connection endpoint](Image)

If a local-node client requires connection-oriented communication with a remote-node peer process, the local-node client must first open a transport address and then open a connection endpoint in its underlying transport. The client makes two calls to ZwCreateFile, first to open an address and, then, a connection endpoint.

When opening a connection endpoint, the client passes a client-supplied context for the connection in
the EA (extended attributes) buffer parameter to ZwCreateFile. Within the EA buffer, the client sets the EaName member to the system-defined value TdiConnectionContext followed by an EA value that is usually the address of a client-allocated context area in which the client will subsequently maintain state about the endpoint-to-endpoint connection to be established.

The client’s call to ZwCreateFile causes the I/O Manager to create a client-process-specific file object to represent the connection endpoint and to call the TDI transport driver’s TdiDispatchCreate routine with an IRP containing the client-supplied parameters to ZwCreateFile. TdiDispatchCreate parses the EA information, and the transport sets up internal state for the open connection endpoint of this client if the call succeeds.

After its successful call to ZwCreateFile returns a file handle to the client and the client has obtained a pointer to the file object with ObReferenceObjectByHandle, it is ready to make certain TDI XXX IOCTL requests to the underlying transport to establish an endpoint-to-endpoint connection with a remote-node peer.

First, the client must associate the connection endpoint with its open transport address by submitting a TDI_ASSOCIATE_ADDRESS request, set up with TdiBuildAssociateAddress, to its underlying transport.

For more information about setting up TDI XXX IOCTL IRPs, see Section 5.3, next. For more information about how to establish an endpoint-to-endpoint connection, see Section 5.5.

When the endpoint-to-endpoint connection has been disconnected and the client no longer will use any connection-oriented communications, the client must close the connection endpoint by passing the file handle that was returned by ZwCreateFile to ZwClose, as described later in Section 5.12.

5.3 Packaging and Submitting IOCTL Requests

Figure 5.3 shows how TDI clients set up and submit TDI XXX IOCTL requests to their underlying TDI transports.
A kernel-mode client prepares its own IOCTL IRPs for communication with the underlying transport driver, using one of the `TdiBuildXxx` macros summarized in Chapter 4. The client can either obtain an IRP from a still higher layer of the network or call `TdiBuildInternalDeviceControlIrp` to allocate an IRP for itself.

A subsequent call to a `TdiBuildXxx` macro sets the appropriate TDI_XXX code in the `MinorFunctionCode` of the IRP and sets the `MajorFunctionCode` to `IRP_MJ_INTERNAL_DEVICE_CONTROL` for the underlying transport driver, as well as setting the client-supplied and IOCTL-specific parameters to each `TdiBuildXxx` macro in the IRP. For example, to set up the TDI_ASSOCIATE_ADDRESS request, mentioned in the preceding section, the client passes both a pointer to the file object that represents its connection endpoint and the handle to the file object that represents its transport address when it uses `TdiBuildAccept`.

When the IRP is set up, the client calls `IoCallDriver` to submit its IOCTL request to its underlying transport. The transport’s `TdiDispatchInternalDeviceControl` routine receives the client-supplied IRP from `IoCallDriver` and usually forwards the client’s request to an internal driver function for further processing. The transport completes the requested operation, sets the I/O status block in the IRP with the results of the operation, and calls `TdiCompleteRequest` (or `IoCompleteRequest`) with the IRP when the client’s request has been satisfied.

If the client supplied the entry point of its `IoCompletion` routine when it called the `TdiBuildXxx` macro, the client’s `IoCompletion` routine is called when the transport completes the requested operation and calls `Io/TdiCompleteRequest` with the client-supplied TDI_XXX IOCTL request.

5.4 Setting and Querying Information

Figure 5.4 shows how a TDI client can make queries about the features of its underlying transport or request its underlying transport to set its state data to client-specified values.
A TDI client can query certain transport driver information, such as connection-status information, activity on a particular transport address, transport-specific limits on datagram size and number, driver statistics, and internal send/receive buffer sizes if the transport buffers data internally. A client also can set some state information in the underlying transport, although it cannot override the underlying transport-determined values for certain size limits by issuing set-information requests.

If a client wants to query information concerning an open transport address or a connection endpoint, it submits a TDI_QUERY_INFORMATION request, set up with TdiBuildQueryInformation, to the underlying transport. When it uses this macro, the client passes pointers to the file object that represents the address or connection endpoint and to a client-supplied buffer in which the transport returns the requested information.

To set transport information, the client issues a TDI_SET_INFORMATION request, set up with TdiBuildSetInformation.

For each of these operations, the client also passes a system-defined TDI_QUERY_XXX value as the QType or SType parameter to the TdiBuild..Information macros to discriminate among the types of information that can be queried and set.

To query or set information that does not specifically concern an address or a connection endpoint, a client first must open a control channel in the TDI driver before submitting its query/set-information request(s). For example, a client querying information about the underlying transport’s broadcast address for connectionless-data transmission or the transport’s current statistics uses a control channel.

To open a control channel, the client calls ZwCreateFile with a NULL EaBuffer pointer. The client’s call to ZwCreateFile causes the I/O Manager to create a client-process-specific file object to represent the control channel and to call its underlying transport’s TdiDispatchCreate routine with an IRP containing the client-supplied parameters to ZwCreateFile. TdiDispatchCreate notes the absence of EA information, and the transport sets up internal state for the open control channel and for this client if the call succeeds.

As an alternative for opening a file object that represents a control channel, the client can call IoGetDeviceObjectPointer, which returns pointers to the named device object and corresponding file object.

After its successful call to ZwCreateFile returns a file handle to the client and the client has obtained a pointer to the file object with ObReferenceObjectByHandle, it is ready to make certain types of TDI_QUERY_INFORMATION or TDI_QUERY_SET_INFORMATION IOCTL requests to its underlying transport.
When its query- and/or set-information operations are complete, the client must close the open control channel by passing the file handle that was returned by `ZwCreateFile` to `ZwClose` or passing the file object pointer that was returned by `IoGetDeviceObjectPointer` to `ObDereferenceObject`, as described later in Section 5.13.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

5.5 Making an Endpoint-to-Endpoint Connection

TDI clients on separate nodes of the network and their respective transports cooperate to establish an endpoint-to-endpoint connection between them. Before such a connection can be attempted, each client on its own node must do the following:

1. Open a transport address, as already described in Section 5.1.
2. Open a connection endpoint, as already described in Section 5.2.
3. Associate its open connection with the open address by submitting a `TDI_ASSOCIATE_ADDRESS` IOCTL request to its underlying transport, as mentioned in Section 5.2 and described in Section 5.3.

Then, one of the clients makes a connection offer to the other, while the other usually listens (passively) for the connection offer to come in.

Either client also might have registered its `ClientEventConnect` handler with its underlying transport (see Section 5.1). The transport driver calls `ClientEventConnect` when any connection offer directed to its client’s open transport address comes in from a remote node.

**Requesting a Connection to a Remote Node**

Figure 5.5 shows how a local-node TDI client initiates a connection offer to a remote-node peer process.

![Diagram of TDI connection process](image)
Figure 5.5 Requesting a connection

The local-node client makes a connection offer to a remote-node peer process by submitting a TDI_CONNECT request, set up with TdiBuildConnect, to its underlying transport.

The local-node transport determines the client-specified target remote-node address from its client’s connect request and transmits the connection offer to the corresponding remote-node transport.

The remote-node transport notifies its client of the incoming connection offer, either by satisfying a TDI_LISTEN request previously submitted by its client or by calling the previously registered ClientEventConnect handler.

The local-node transport fails the connect IRP if the remote-node client is not listening or if the remote-node transport does not respond. The remote-node client can accept or reject the offered connection if both transports support delayed-connection acceptance, as described next.

Accepting a Connection Offer from a Remote Node

Figure 5.6 shows how a local-node client listens for a connection offer from a remote-node peer process.

![Figure 5.6 Accepting a connection (listen operation)](image)

To establish an endpoint-to-endpoint connection, one client makes a connection offer, and the other indicates to its underlying transport that it is waiting for a connection offer to occur.

A local-node client can passively listen for such an incoming connection offer by submitting a TDI_LISTEN request, set up with TdiBuildListen, to its underlying transport. The client can specify the remote-node transport address from which an acceptable offer can occur when it sets up the listen IRP. If the transports support delayed-connection acceptances, the client can direct the transport to either accept an offer from a particular remote-node address immediately or allow the client to inspect the offer and decide whether to accept it.

When the transport receives a TDI_LISTEN request, it monitors connection offers coming from
remote nodes that are directed to the transport address opened by the client. When it receives such a connection offer from an acceptable remote-node address, the transport copies information transmitted with the offer into a buffer the local-node client supplied with its TDI_LISTEN request and completes the IRP back to the listening client.

If the client directed the transport to accept the incoming connection offer immediately, completion of the listen request occurs as the transport notifies the remote-node transport that an endpoint-to-endpoint connection has been established with the remote-node client. In effect, the local-node client has made an endpoint-to-endpoint connection with its remote-node peer even before it "knows" that its listen IRP has been completed, and the local-node client can receive data on that endpoint-to-endpoint connection at once. Otherwise, the transport’s completion of the listen request notifies the local-node client of the offer to connect, and the client either accepts or rejects the request by issuing a TDI_ACCEPT or TDI_DISCONNECT request, respectively. The local-node client uses TdiBuildAccept or TdiBuildDisconnect to set up the IRP it submits to its underlying transport for a delayed-connection acceptance operation.

Accepting a connection is even simpler if the local-node client is using event handling to communicate with the underlying TDI transport driver. When its transport receives a connection offer from a remote-node client that is directed to the open transport address of the local-node client, the transport calls the registered ClientEventConnect handler of the local-node client, passing in the transport address of the offering client and any connect data the transport received with the connection offer. ClientEventConnect then accepts or rejects the connection offer immediately.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

5.6 Sending and Receiving Connection-Oriented Data

TDI clients on separate nodes of the network must establish an endpoint-to-endpoint connection between them to send and receive connection-oriented data. Before either client sends data to the other, each client on its own node must do the following:

1. Open a transport address, as already described in Section 5.1.
2. Open a connection endpoint, as already described in Section 5.2.
3. Associate its open connection with the open address by submitting a TDI_ASSOCIATE_ADDRESS IOCTL request to its underlying transport, as mentioned in Section 5.2 and described in Section 5.3.
4. Establish the endpoint-to-endpoint connection between them, one by making a connection offer and the other by accepting that offer, as described in Section 5.5.

In fact, the client that accepts a connection offer can receive data from its remote-node peer as soon as the local-node transport sends a connection-acceptance frame to the remote-node transport, which can occur before the accepting client’s TDI_LISTEN IRP has been fully completed back to that client.

After an endpoint-to-endpoint connection has been made by the underlying transports, either client
After an endpoint-to-endpoint connection has been made by the underlying transports, either client can send data to the other across the network.

### Sending Data on an Endpoint-to-Endpoint Connection

Figure 5.7 shows how a kernel-mode client sends data on an endpoint-to-endpoint connection through its underlying local-node transport to a remote-node peer.

*Figure 5.7 Sending data to a remote-node peer*

A local-node TDI client issues a send request to transmit data from its connection endpoint to the remote-node connection endpoint. To do this, the local-node client submits a `TDI_SEND` IOCTL request to its transport. This IRP, which the client sets up with `TdiBuildSend`, contains a pointer to a client-supplied buffer containing a stream-oriented or message-oriented TSDU. This buffer can be any size up to the maximum the TDI transport driver allows. If the transport supports expedited sends, the client can request that the TSDU be transmitted as expedited data, ahead of any preceding normal sends it has already submitted that are currently pending in its underlying transport. If the transport driver supports internal buffering, its client can issue a non-blocking send.

The client’s call `IoCallDriver` with the TDI_SEND IRP forwards the IRP to the underlying
transport’s TdiDispatchInternalDeviceControl routine. This routine checks the MinorFunction code in the transport’s I/O stack location of the IRP and usually calls a send-specific internal driver function to process the IRP further. For send requests, the internal driver function usually queues the IRP if the client has already submitted other send requests that have not yet been transmitted over the network to the remote node. The transport always queues requests to send expedited data ahead of requests to send normal TSDUs to the client’s remote-node peer. Whether normal or expedited, the queuing transport always transmits client-requested sends over the network in FIFO order. The transport either copies the client-supplied data into its internal buffers or sends the specified data on the network before it completes each client-submitted TDI_SEND IRP.

If the underlying transport has failed a non-blocking send request due to insufficient internal buffer space in the transport, the driver calls its client’s registered ClientEventSendPossible handler when the transport again has available buffer space for sends. Then, ClientEventSendPossible can resubmit the TDI_SEND request that the transport previously failed.

**Receiving Data on an Endpoint-to-Endpoint Connection**

Figure 5.8 shows how a kernel-mode client receives data on an endpoint-to-endpoint connection through its local-node transport from its remote-node peer.
Figure 5.8 Receiving data from a remote-node peer

A local-node client can receive a TSDU, either normal or expedited, on a connection by making a TDI_RECEIVE request to the underlying transport. This IRP, which the client sets up with TdiBuildReceive, contains a pointer to a client-supplied buffer into which the transport copies all or part of the TSDU data it received from the client’s remote-node peer. This buffer can be any size up to the maximum the TDI transport driver allows.

The client’s call to IoCallDriver with the TDI_RECEIVE IRP forwards the IRP to the underlying transport’s TdiDispatchInternalDeviceControl routine. This routine checks the MinorFunction code in the transport’s I/O stack location of the IRP and usually calls a receive-specific internal driver function to process the IRP further. The internal driver function transfers received data into the client-supplied buffer until it is full or until the received TSDU data is exhausted.

However, expedited data takes precedence over normal data during receive operations. If an expedited TSDU comes in from the remote node while the transport is processing a client-submitted receive request for normal data, the transport completes the IRP back to its client with whatever normal TSDU data has already been transferred into the client-supplied buffer. Then, the transport processes the expedited receive to completion, and the client must issue another TDI_RECEIVE request to obtain the remainder of the normal TSDU.

A client can also receive data from its remote-node peer as an event notification from the underlying TDI transport driver. For these notifications, the driver removes the transport layer header from the TSDU that it receives from the remote node and calls the client’s registered ClientEventReceive, ClientEventChainedReceive, ClientEventReceiveExpedited, or ClientEventChainedReceiveExpedited handler. The client’s event handler can then copy as much of the data as possible. If ClientEventReceive or ClientEventReceiveExpedited does not receive all the data, it can do one of the following:

- Return a not-accepted status immediately, effectively telling the transport that the received TSDU is not of interest to the client.
- Make another TDI_RECEIVE receive request to obtain the remainder of the TSDU data.
- Rely on subsequent driver receive-event notifications to obtain the remainder of the data.

The ClientEventChainedReceive and ClientEventChainedReceiveExpedited handlers are always given read-only access to a full TSDU by the underlying transport. Consequently, these routines have no need to issue a sequence of TDI_RECEIVE requests to the underlying transport or to process partial indications of received TSDUs. However, the client is responsible for calling TdiReturnChainedReceives promptly to return the resources associated with such an indication to the NDIS miniport that originally allocated them.

For more information about registering ClientEventXxx handlers, see Section 5.1.
5.7 Sending and Receiving Connectionless Data

A kernel-mode client can engage in connectionless communication by sending data over the network as soon as it has successfully opened a transport address in the underlying TDI transport.

To receive data in this manner, the client must register its ClientEventReceiveDatagram and, possibly, ClientEventChainedReceiveDatagram handlers with the underlying transport or issue an explicit TDI_RECEIVE_DATAGRAM request to the underlying transport.

For more information about opening a transport address and registering an event handler on that address, see Section 5.1.

Sending a Datagram

Figure 5.9 shows how a TDI client sends a datagram to a remote-node address.

As this figure shows, sending a datagram is similar to sending connection-oriented data. However, the local-node client submits a TDI_SEND_DATAGRAM request to its underlying transport on an open transport address, instead of a send request on an established endpoint-to-endpoint connection as already described in Section 5.6.

A TDI_SEND_DATAGRAM IRP requests the underlying transport to transmit a client-supplied datagram to an unknown number of remote-node clients at a particular remote-node transport address.

The local-node client with an open transport address can use TdiBuildSendDatagram to set up such a request. Along with the target remote-node address, the client provides buffered data that can be any size up to the maximum the TDI transport driver allows for datagram sends. The client can determine this limit by querying the underlying transport, as described already in Section 5.4.
Receiving a Datagram

Figure 5.10 shows how a kernel-mode client receives a datagram through its underlying transport from a remote node.

**Figure 5.10 Receiving a datagram**

As this figure shows, receiving a datagram is similar to receiving connection-oriented data. However, the local-node client submits a `TDI_RECEIVE_DATAGRAM` request to its underlying transport on its open transport address, instead of a receive request on an established endpoint-to-endpoint connection as already described in [Section 5.6](#).

However, the client can receive a datagram sent by any remote-node client that specifies the local client’s open transport address as the destination of the datagram. Other local clients that have opened the same transport address can also receive the same datagram.

A `TDI_RECEIVE_DATAGRAM` IRP requests the underlying transport to return a datagram from any remote-node client that has sent one to the local-node client’s open transport address. The local-node client can use `TdiBuildReceiveDatagram` to set up such a request. Along with a pointer to the file object that represents its open transport address, the client provides a buffer that can be any size...
up to the maximum the TDI transport driver allows for received datagrams. Because TDI transports
never fragment datagrams, the local-node client usually issues one receive-datagram request to accept
one datagram.

When it satisfies such a request, the transport driver copies a received datagram into the client-
supplied buffer and returns the transport address of the remote-node client that sent the datagram with
the completed IRP. If the received datagram is too large for the buffer, the transport returns as much
data as possible and discards the remaining data.

A client can also receive a datagram from a remote node as an event notification by the underlying
TDI transport driver. For these notifications, the transport removes its header from the TSDU that it
receives from the remote node and calls the client’s registered ClientEventReceiveDatagram, or
possibly ClientEventChainedReceiveDatagram, handler. ClientEventReceiveDatagram can do one of
the following:

- Return a not-accepted status immediately, effectively telling the transport that the TSDU is not
  of interest to the client.
- Copy as much data as possible into an internal buffer and set up a
  TDI_RECEIVE_DATAGRAM request for the remainder of the TSDU if the transport
  provided only part of the data and return control.
- Copy all of the TSDU into an internal buffer and return control.

Non-buffering transport drivers can discard data after a receive-datagram event indication if their
clients do not copy the data or return an IRP. Buffering transport drivers retain a certain amount of
datagram information that their clients can retrieve later by making explicit
TDI_RECEIVE_DATAGRAM requests.

The ClientEventChainedReceiveDatagram handler is always given read-only access to a full TSDU
by the underlying transport. Consequently, this routine has no need to issue a
TDI_RECEIVE_DATAGRAM request to the underlying transport or to process partial indications of
received TSDUs. However, the client is responsible for calling TdiReturnChainedReceives
promptly to return the resources associated with such an indication to the NDIS miniport that
originally allocated them.

For more information about registering ClientEventXxx handlers, see Section 5.1.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

5.8 Connection-Oriented Versus
Connectionless Transfers

TDI provides connection-oriented services at the transport layer to provide the highest quality of
service at the lowest cost. Depending on the specific driver protocol, this can involve packetizing of
client data, sequencing, acknowledgment, retransmission, flow-control, and error recovery.
client data, sequencing, acknowledgment, retransmission, flow control, and error recovery.

TDI also provides connectionless services for datagrams. However, these services are lightweight: they do not provide error-free delivery or flow control. The transport layer generally does not segment or reassemble datagrams. Delivery of connectionless data is best-effort only for TDI transports.

A datagram send is an inherently unreliable form of network communication. The sending client has no way of determining how many or which remote-node clients have opened the target remote-node address or whether the remote-node transport is currently accepting datagrams. Furthermore, the local-node TDI driver can lose or duplicate a datagram at the discretion of the transport driver writer. By contrast, the local-node transport is responsible for retrying sends on an established endpoint-to-endpoint connection until the transmitted data is accepted on the remote node and acknowledged by the remote-node transport while the endpoint-to-endpoint connection is active.

Like a datagram-send, a datagram receive operation can be unreliable. That is, the local-node TDI driver can lose or duplicate a received datagram at the discretion of the transport driver writer. By contrast, the local-node transport is responsible for delivering a receive on an established endpoint-to-endpoint connection until the transmitted data is accepted (or rejected) by its client and acknowledged to the remote-node transport while the endpoint-to-endpoint connection is active.

An endpoint-to-endpoint connection remains active until it is broken by a disconnect operation, as described later in Section 5.11 or until the underlying transport’s time-out logic determines that the remote node is failing to respond.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

5.9 Requesting Transport-Specific Actions

Figure 5.11 shows how a TDI client can make transport-specific action requests to the underlying TDI driver if that transport defines such extensions to TDI.

![Figure 5.11 Requesting transport-specific extensions](image)

A TDI client can send special or proprietary extension requests to a TDI transport driver that defines
A TDI client can send special or proprietary extension requests to a TDI transport driver that defines a set of transport-specific action codes for these operations. These extensions, which can concern an open address, an open connection endpoint, or an open control channel, apply only to the calling client and not to any other TDI transport clients or drivers.

To request such a transport-defined action operation, the client first must open the address, connection endpoint, or control channel, as already described in Section 5.1, Section 5.2, or Section 5.4, respectively. Then, the client can use `TdiBuildAction` to set up a TDI_ACTION request with a client-supplied buffer containing the transport-defined action code and corresponding action parameter block for the requested operation.

**5.10 Receiving Error Notifications**

In response to client-submitted IRPs, the transport driver notifies a TDI client of error conditions in the status codes that it returns. While this provides a client with request-specific errors, it would be difficult for the client to maintain a count of failed IRPs and apply internal logic to determine whether the underlying drivers are in a state that renders the client’s subsequent network I/O operations unreliable.

To receive a notification of unexpected error conditions in an underlying driver or in the underlying physical medium, the client can register its `ClientEventError` or `ClientEventErrorEx` handler with its underlying transport, as already described in Section 5.1. Then, if the transport itself or any lower driver that the transport depends on to carry out client communications over the network encounters such an error condition, the transport calls `ClientEventErrorEx`. This call notifies its client that network I/O on the client’s open transport address is no longer reliable (or, sometimes, even possible). Then, the client can notify its own higher level clients of the network failure and clean up all TDI-client-allocated resources for pending operations on the affected open transport address.

**5.11 Disconnecting an Endpoint-to-Endpoint Connection**

Figure 5.12 shows how TDI clients release an endpoint-to-endpoint connection.

![Diagram](image-url)
Figure 5.12 Disconnecting an endpoint-to-endpoint connection

Disconnection behavior is transport-specific in nature. When a connection-oriented TDI client initiates a disconnect between nodes, both nodes might need to participate in the disconnection operation. That is, when one client initiates a disconnect, the remote-node client possibly must respond to it.

During a disconnect operation, the TDI transport driver usually refuses incoming requests on the open connection endpoint and stops all activity at the specified connection endpoint unless both transports support controlled disconnects and the initiating client requests one.

As Figure 5.12 shows, one client on an endpoint-to-endpoint connection can initiate a disconnection operation by submitting a TDI_DISCONNECT request, set up with TdiBuildDisconnect, to its underlying transport. When that transport finishes processing the initiating client’s request, it notifies the remote-node transport driver that a disconnection is in progress, and this transport begins returning an appropriate status code for client-submitted I/O requests on the endpoint-to-endpoint connection.

If the responding client registered its ClientEventDisconnect handler, the TDI transport driver notifies the client when the disconnect occurs by calling this handler. Then, ClientEventDisconnect acknowledges the disconnection by making a TDI_DISCONNECT request to its underlying transport. This notification allows the responding client to clean up client-allocated state for the endpoint-to-endpoint connection promptly.

However, a disconnection operation does not close either client’s open connection endpoints or transport addresses. After TDI_DISCONNECT requests have been satisfied, both clients can reuse...
the file objects representing these open resources in their underlying transports. For example, either client might make a subsequent connection offer to another remote node on the network, as already described in Section 5.5. Until each client closes the file objects representing its respective connection endpoint and the associated transport address as described in Section 5.12 and Section 5.13, respectively, these resources remain allocated to the client and available for client-submitted IOCTL requests to the underlying transport.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

5.12 Closing a Connection Endpoint

Figure 5.13 shows how a kernel-mode client closes a connection endpoint.

![Diagram of closing a connection endpoint]

Figure 5.13 Closing a local-node connection endpoint

After an endpoint-to-endpoint connection has been disconnected, as already described in Section 5.11, a client can close the connection endpoint.

When a client no longer has any use for an open connection endpoint, it must close that connection endpoint as follows:

- Pass the file object pointer returned by `ObReferenceObjectByHandle` to `ObDereferenceObject`.
- Pass the file handle that was returned by `ZwCreateFile` when the connection endpoint was opened to `ZwClose`.

Then the I/O Manager's IRP is dispatched to `TDI_DISPATCH_TDI` and `TDI_CLOSE` is handled.
Then, the I/O Manager submits IRPs to the transport’s `TdiDispatchCleanup` and, subsequently, `TdiDispatchClose` routines.

These transport routines immediately close the connection endpoint and free all associated transport driver resources. `TdiDispatchCleanup` also terminates any active connection involved with the endpoint by sending a disconnect notification to the corresponding remote-node transport.

As the preceding sentence implies, it is unnecessary for a TDI client to disassociate the connection endpoint from its associated transport address before making a close-connection-endpoint request. If necessary, the underlying transport driver simulates the effects of a disassociation.

However, a client can explicitly disassociate a connection endpoint from an open transport address before closing the connection endpoint by making a `TDI_DISSOCIATE_ADDRESS` request to the transport, set up with `TdiBuildDisassociateAddress`.

For example, a client might make a disassociate-address request and reassociate the open connection endpoint with another open transport address.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

5.13 Closing a Transport Address or Control Channel

After closing any associated connection(s), a TDI client is ready to close an open transport address.

When a client no longer has any use for an open transport address or control channel, it must release that object as follows:

- Pass the file object pointer returned by `ObReferenceObjectByHandle` to `ObDereferenceObject`.
- Pass the file handle that was returned by `ZwCreateFile` when the connection endpoint was opened to `ZwClose`.

In a similar manner, a TDI client also can close any control channel it has opened in its underlying transport driver.

If the client opened the file object representing a control channel with a call to `IoGetDeviceObjectPointer`, it must pass the returned file object pointer to `ObDereferenceObject` to release the file object.

Then, the I/O Manager submits IRPs to the transport’s `TdiDispatchCleanup` and `TdiDispatchClose` routines.

These transport routines immediately close the transport address or control channel and free all
These transport routines immediately close the transport address or control channel and free all associated client-specific transport driver resources. For example, TdiDispatchCleanup cancels any pending requests on a transport address that is being closed, deregisters any ClientEventXxx handlers on that address, and cleans up client-specific state for that address. If this client has released the last file handle for a particular transport address, the transport also releases internal state for that transport address.

After ZwClose returns to the client, it cannot submit a request to the underlying transport for the transport address or control channel it previously had open. The client-specific file object representing that address or control channel no longer exists.

Chapter 6 Transport Helper DLLs for Windows Sockets

Originally an application programming interface specific to TCP/IP, Windows Sockets has since become the primary user-mode interface to Windows® 2000 network transports. In addition to TCP/IP, the Windows Sockets dynamic-link library works in conjunction with several other transports that provide extensions in the form of transport-specific user-mode Windows Sockets helper (WSH) DLLs.

This describes the WSH DLL to be supplied with a new transport driver to support application calls through Windows Sockets, as follows:

- An overview of the architectural relationships among Windows Sockets, transport drivers, and transport-specific WSH DLLs in Section 6.1
- An overview of how Windows Sockets communicates with transport-specific WSH DLLs in Section 6.2
- Setting up the configuration registry for WSH DLLs in Section 6.3
- Managing synchronization for WSH DLLs in Section 6.4
- How WSH DLLs can support additional information sent on the network with connections and disconnections in Section 6.5
- A summary of the required and optional WSHXxx functions in Section 6.6

For information about how applications interface with Windows Sockets 2, see the Platform SDK.
writers to facilitate the usage of Windows Sockets with their transports. Figure 6.1 shows an overview of this architecture.

Figure 6.1 Windows Sockets Helper DLL architecture

Windows 2000 provides a dynamic-link library, msafd.dll, which is a sockets service provider. When a transport driver is installed in the system and it installs a transport-specific WSH DLL, network setup will automatically configure msafd.dll to be the service provider for that WSH DLL. When an application makes a call to a Windows Sockets function, such as WSAsocket, MSAFD as the service provider will resolve the call and make a call to the appropriate WSH DLL for assistance as necessary.

Some Windows Sockets functions require no assistance from any WSH DLL. For example, sending or receiving data does not require the WSH DLL as the connections have already been established. In cases such as these, MSAFD can communicate directly with the transport by calling Win32® functions.

However, for function calls that rely on transport-specific features or where the implementation can vary from transport to transport, the transport-specific WSH DLL will be used to resolve these ambiguities. For example, WSAJoinLeaf adds a socket to an established multipoint session. Each transport implements the addition of new connections to a multipoint session differently. Consequently, MSAFD calls the appropriate WSH DLL to support the sockets interface according to the transport-specific implementation. To support application calls to WSAJoinLeaf, MSAFD calls the WSH DLL' WSHJoinLeaf function to validate the options requested and calls the transport with the appropriate information to add a new socket to a multipoint session.

If a transport supports the new features of Windows Sockets 2, including multipoint session sockets, socket grouping, and logical representations of socket addresses, the corresponding WSH DLL must export the Windows Sockets 2 functions, summarized later in Section 6.6.
6.2 Communicating with a WSH DLL

Each transport that supports Windows Sockets applications supplies a user-mode Windows Sockets helper dynamic-link library to interpret network addresses and to process socket options in the architecture as outlined in Section 6.1.

For example, when any application calls `socket`, it specifies an address family, a socket type, and a protocol. These three arguments must uniquely identify a transport driver to support the socket. Windows Sockets searches for a match between these application-supplied arguments and the standardized configuration information stored in the registry for WSH DLLs. If it finds a match, Windows Sockets subsequently calls down through MSAFD to the `WSHXxx` functions exported by the WSH DLL as necessary. Otherwise, the application’s call `socket` fails.

When a match is found, Windows Sockets calls `LoadLibrary` on the WSH DLL, and then calls `GetProcAddress` to retrieve the entry point for each exported `WSHXxx` function. These `WSHXxx` functions are called whenever necessary to query for protocols supported by the helper DLL, to translate addresses, and to process WSH-supported options as `getsockopt` and `setsockopt` calls occur. For example, when an application passes an option to `setsockopt` that is not explicitly supported by Windows Sockets, the appropriate WSH DLL is called to process the option. The WSH DLL takes any necessary action to support each such option the corresponding transport supports.

The following conventions permit mechanical transformations between socket addresses, application-specified as `SOCKADDR` structures, and the TDI-defined addresses used by underlying transport drivers.

- TDI address types, specified as values in the `AddressType` member of `TA_ADDRESS` structures, are identical to the address family values of socket address structures.
- The `Address[ ]` array of a `TA_ADDRESS` structure has exactly the same format as the data following `sa_family` in a socket address structure for a given address type.

6.3 Configuring a WSH DLL

Two places in the registry must be set up with standardized configuration information for each WSH DLL:

1. Under `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Winsock\Parameters` is a value entry of type `REG_MULTI_SZ` specifying a list of protocols (transport drivers) that each have a corresponding WSH DLL. The names stored in
1.0 NDIS Intermediate Drivers

protocols (transport drivers) that each have a corresponding WSH DLL. The names stored in this list match the key names for the corresponding transport drivers under ..\CurrentControlSet\Services.

2. Under HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TransportDriverName\Parameters for each such transport driver is a Winsock subkey. Windows Sockets opens this key by forming a name string from the path to its own key under the Services key, the transport name specified in the list under ..\Winsock\Parameters, and the transport’s Parameters\Winsock key, which must contain the following value entries:

- **Mapping** — a REG_BINARY value that describes the address family, socket type, and protocol parameter triples supported by the WSH DLL. The format for this binary data is the WINSOCK_MAPPING structure, as defined in wsahelp.h.
- **HelperDllName** — a REG_EXPAND_SZ value that specifies the path to the WSH DLL itself.
- **MinSockaddrLength** — a REG_DWORD value that specifies the smallest valid SOCKADDR size, in bytes, for the WSH DLL.
- **MaxSockaddrLength** — a REG_DWORD value that specifies the largest valid SOCKADDR size, in bytes, for the WSH DLL.

Storing this information in two distinct places under ..\CurrentControlSet\Services helps to localize information about a transport driver under that driver’s own Parameters key. The information under ..\Services\Winsock\Parameters provides a pointer to the actual information for each transport’s WSH DLL, allowing setup programs to interoperate more easily.

Windows 2000 setup utilities provide functions that perform most of the necessary work to set up WSH DLL registry information for a transport driver. These routines are AddWinsockInfo and RemoveWinsockInfo in the setup file utility.inf, which can be found in the system directory. A transport driver’s installation script can call AddWinsockInfo, passing in the key name under Services of the transport driver, the full path name of the transport-specific WSH DLL, and the minimum and maximum SOCKADDR lengths. AddWinsockInfo then stores the standardized information already described in the registry.

For an example of a transport’s call to this setup function, see oemmxtc.inf, the installation script for TCP/IP, also located in the system directory.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

### 6.4 WSH DLL Synchronization

Windows Sockets guarantees synchronization to a WSH DLL on a per-socket basis. That is, two threads cannot be executing concurrently in WSH\xxx functions for the same socket. Consequently, a WSH DLL that maintains its state information on a per-socket basis need not do any synchronization of its own.

If a WSH DLL maintains global state or per-process state, it must synchronize access to such state information internally. Windows® 2000 provides several user-mode synchronization mechanisms such a WSH DLL might use, including semaphores, events, and critical sections. Which type of
such a WSH DLL might use, including semaphores, events, and critical sections. Which type of synchronization mechanism to use is up to the WSH DLL developer.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

6.5 Supporting Connect and Disconnect Data with a WSH DLL

Some transports, such as DECNet and OSI TP4, support *connect and disconnect data*: additional data, not in the normal network data stream, that is sent on the wire along with connect or disconnect requests. Typically, connect and disconnect data is used for operations like application-level version negotiation.

TDI supports the transmission of connect and disconnect data in the `TDI_CONNECTION_INFORMATION` structure, which has members specifying the `UserDataLength`, `UserData`, `OptionsLength`, and `Options`. This structure is passed in TDI connect, accept, and disconnect requests by kernel-mode TDI clients.

However, Windows Sockets does not provide input parameters for connect data to the `connect` function or for disconnect data to the `shutdown` and `closesocket` functions. WSH DLLS can add support for connect and disconnect data for applications that call the Windows Sockets `setsockopt` and `getsockopt` functions.

The `setsockopt` function can be called to specify connect and disconnect data to be sent to a remote node, and `getsockopt` can be called to retrieve connect and disconnect data sent from the remote node. To support this, see the `SO_XXX` options in the header file `winsock.h`:

How an application uses these options in conjunction with `getsockopt` and `setsockopt` depends on whether the application is a server or client (see Section 6.4.1 or 6.4.2, respectively), and on how the application uses the transmitted connect data. For transports, such as DECNet, that have a preexisting definition of connect data different from what is described here, a WSH DLL translates between the transport’s semantics and the semantics expected by Windows Sockets, making calls to `getsockopt` and `setsockopt`.

Connect and disconnect options are effectively the same as connect and disconnect data from the standpoint of Windows Sockets. These are actually a buffer of data passed to the transport prior to the connect or disconnect, and a buffer containing data that is retrieved following a connect or disconnect. Consequently, an application uses connect or disconnect options in much the same manner as connect or disconnect data. The only difference is the option names used for the parameter passed to `getsockopt` and `setsockopt`; for example, `SO_CONNOPT` in place of `SO_CONNDATA`.

Built on Wednesday, June 28, 2000

Network Drivers: Windows 2000 DDK

6.5.1 Client Applications and Connect Data
6.5.1 Client Applications and Connect Data

An application written to communicate through Windows Sockets with a transport that supports connect data should call `setsockopt` with SOL_SOCKET as the level, SO_CONNDATA as the option name, and an option value that is a pointer to a buffer containing the connect data the transport should send to the remote node with the connection request. Such an application must call `setsockopt` before it calls `connect`, and the application-supplied connect data is transmitted when the application makes its call to `connect`.

Besides making a call to `connect`, retrieving response-connect data requires two additional steps:

1. Before calling `connect`, the application must inform Windows Sockets of how much space to reserve for the expected response-connect data. To do this, the application calls `setsockopt` with option level SOL_SOCKET, option name SO_CONNDATALEN, and option pointing to an integer that specifies the maximum number of bytes the application expects. Windows Sockets allocates a buffer of this size to contain connect data sent by the server application from the remote node.

2. After `connect` has completed successfully, the application can call `getsockopt` with an option level of SOL_SOCKET, option name of SO_CONNDATA, and option pointing to a large enough buffer to hold the response-connect data. Then, Windows Sockets copies the response-connect data, if any, into the application’s buffer.

After an application has set up a response-connect buffer, Windows Sockets uses that buffer for the lifetime of the socket. That is, an application can retrieve response-connect data for a particular socket at any time after the socket is connected and before it is closed.

6.5.2 Server Applications and Connect Data

Server applications use connect data differently from client applications in two important ways:

1. A server application receives the client’s connect data before it sends back its own.
2. Because Windows Sockets does not support delayed-connection acceptances as TDI does, a server application must send the same response-connect data to all clients.

To receive connect data sent by clients, a server application calls `setsockopt` with the SO_CONNDATALEN option to force Windows Sockets to reserve buffer space for receiving connect data for incoming connect requests. The specified amount of connect-data buffer space is used subsequently for all incoming connects, but Windows Sockets allocates a separate buffer for each connect.

After a newly connected socket is accepted with `accept`, the server application calls `getsockopt` with the SO_CONNDATA option and a server-allocated buffer large enough to hold the expected data. Windows Sockets fills in this buffer with the connect data sent by the client.
To send response-connect data to clients, a server application calls `setsockopt` with the `SO_CONNDATA` option and a buffer of data before any incoming connections are accepted (typically before `listen`). Thereafter, any client that connects to the server receives that server-supplied connect-response data. The connect-response data for a server can be changed at any time with another `setsockopt` call, which discards the previous data and replaces it with the new data.

6.5.3 Disconnect Data

Client and server applications send and receive disconnect data in a manner similar to the way each sends and receives connect data.

To send disconnect data, a client application calls `setsockopt` with the `SO_DISCDATA` option and a buffer of disconnect data to be transmitted to the server on the remote node. This data is sent either when the client application calls `shutdown` to disconnect the send side of the socket or when the application calls `closesocket` on the socket.

To receive disconnect data, a client application must call `setsockopt` with the `SO_DISCDATALEN` option prior to a remote’s closure of the connection. This option specifies the amount of disconnect data the application expects to receive.

After the remote has disconnected, any server-supplied disconnect data can be retrieved with the `SO_DISCDATA` option to `getsockopt`. The client application passes in a buffer large enough to hold all the disconnect data, and Windows Sockets copies the received disconnect data into this buffer.

6.6 WSH DLL Function Summary

This summarizes the set of `WSHXXx` functions that every WSH DLL must export. Each supplier of a new transports can implement these functions in a corresponding WSH DLL according to the features supported by each transport. Windows Sockets treats all WSH DLLs as trusted code. Consequently, any bug in a `WSHXXx` function manifests itself as bug(s) in Windows Sockets.

The following summarizes the set of functions that must be exported by any transport-specific WSH DLL:

- `WSHEnumProtocols` returns a list of protocols (Windows Sockets `PROTOCOL_INFO` structures) that the WSH DLL supports.
WSHGetSockAddrType parses a socket address.

WSHGetSocketInformation is called whenever getsockopt is passed an option that Windows Sockets does not explicitly support.

WSHGetWildCardSockAddr is called when Windows Sockets needs to perform an "automatic" bind of a socket.

WSHGetWinsockMapping returns information about the address family, socket type, and protocol parameter triples supported by the WSH DLL.

WSHNotify is called to notify the helper DLL of a state transition.

WSHOpenSocket is called when a socket is opened.

WSHSetSocketInformation is called whenever setsockopt is passed an option that Windows Sockets does not explicitly support.

To support the new Windows Sockets 2 extensions for leaf/root sockets, socket groups, and logical representation of sockets, the following optional functions should be implemented:

WSHAddressToString returns a logical string representation of a socket address that can be used for display purposes.

WSHGetBroadcastSockaddr obtains a valid broadcast address for a socket.

WSHGetProviderGuid returns the GUID that identifies the protocols supported by a helper DLL.

WSHGetWSAProtocolInfo returns a pointer to protocol information for the protocol(s) supported by a helper DLL. This function is only used during setup.

WSHIoctl obtains information or performs actions based on a unique control code.

WSHJoinLeaf performs any protocol-specific actions that must be taken to add a socket to a multipoint session.

WSHOpenSocket2 performs the protocol-specific actions for creating a new socket. If this function is exported, it replaces the WSHOpenSocket function.

WSHStringToAddress converts a logical string representation of a socket address to a SOCKADDR structure.

For more detailed information about these these WSHXxx functions, see the Network Driver Reference.

Built on Wednesday, June 28, 2000