3Com/Microsoft LAN Manager
Network Driver Interface Specification

Version 2.01 (FINAL)

Published 5 October 1990. Printed in the U.S.A.

Table of Contents

Chapter 1 - Introduction

This document describes the LAN Manager network driver architecture and interfaces that let a DOS or OS/2 system support one or more network adapters and protocol stacks. This architecture provides a standardized way for writing drivers for network adapters and communications protocols. It also solves the problem of how to configure and bind multiple drivers into the desired set of layered protocol stacks.

Drivers written to the interfaces defined here will function concurrently in a system with other networking and protocol drivers, and will operate correctly with the LAN Manager software for DOS and OS/2.

Definition of Terms

To simplify the job of supporting multiple adapters and protocols, the architecture defines four kinds of drivers.

- Media Access Control (MAC) drivers, which provide low-level access to network adapters. The main function of a MAC driver is to support transmitting and receiving packets, plus some basic adapter management functions. MAC drivers are device drivers that are loaded during system initialization and remain permanently in memory. Since they cannot be unloaded, they are called "static".

- Protocol drivers, which provide higher-level communication services from data link to application (depending on the driver). An example is a NetBIOS driver that provides a NetBIOS interface at the top and talks to a MAC driver at the bottom. Protocol drivers can be device drivers, TSRs, or transient DOS applications. A protocol driver is called "static" if it cannot be unloaded. A protocol driver is called "dynamic" if it can be loaded and unloaded on demand.

- MAC-layer entities, which bind to real MAC drivers and expose a new MAC-like layer interface on top. Possible examples are MAC bridges, test tools, or interface mappings which change the NDIS interface to meet some environment-specific administrative requirement.

- The Protocol Manager driver. This is a special driver that provides a standardized way for multiple MAC and protocol drivers to get configuration information and bind together into the desired protocol hierarchy. The Protocol Manager gets all configuration information from a central file, PROTOCOL.INI.

Scope of this Document

This document defines:

1. Protocol Manager functions and interfaces for configuration and binding of MAC and protocol drivers.

2. The software interface between MAC and protocol drivers.

Separate documents will specify the configuration and interface details for other kinds of protocol drivers, including data link and transport drivers.

Changes for this Version

The major highlights of this version compared to the last (1.0) are:

1. Support for dynamic binding/unbinding of protocol modules, allowing protocols to be swapped in and out of memory as needed. No changes are required of MAC drivers to support the dynamic bind/unbind features. In particular NDIS 1.0.1 conformant MACs will support dynamically binding protocol modules.

2. Additional Protocol Manager functions to support dynamic binding and future administrative requirements.

3. Some adjustments to the Reset MAC function, StartReset, and EndReset primitives were made to correct some inconsistencies and keep the logic out of the criticial paths.

4. Additional fields were added to certain tables to provide additional information. The presence or absence of these fields can be determined by examining the length field in each table.

5. Some new recommendations and clarifications on such issues as double-word alignment of data blocks, the use of the permanent station address, the copying of DS and entry points, the use of 80386 32-bit registers, the release of internal resources before confirmations, the handling of 0 length data blocks, the formatting of MAC headers, the use of zero handles, new transmit error codes for Token Ring to support source-routing, and various other points that needed additional clarifications.

6. A standard for protocol service-specific characteristics tables.

7. The inclusion of additional 802.3 and 802.5 specific information and added statistics definitions.

8. Additional information and caveats to help developers.

9. The Protocol Manager now has a transient component (in some configurations) called PROTMAN.EXE. This is now described with certain restricitions imposed on Protocol Manager primitives.

10. Some new error response codes were defined.

11. A new appendix, Appendix E, was added to describe some helpful bind and configuration management utilities provided with Protocol Manager.

12. Selected statistics designated as manditory for both service-specific and media specific statistics(802.3 and 802.5).

13. Extended 802.3 statistics to include Number_of_Underruns.

14. OpenAdapter function expanded to permit driver return of vendor specified warning errors and/or hardware error codes.

It is not expected that any of these changes will result in incompatibilities with protocol and MAC drivers written to previous versions of this specification. Great care was taken to avoid creating incompatibilities. It is the protocol's responsibility to identify and interoperate with older NDIS version driver implementations that may not have implemented support for statistics. Older network drivers will co-exist with network drivers written to this specification. However, to take advantage of new features (such as dynamic binding), developers may wish to update their protocol drivers to be NDIS 2.0.1 compliant.

Chapter 2 - Configuration and Binding

A network server or workstation includes at least one Media Access Control (MAC) and one protocol driver, plus the Protocol Manager driver. More complex configurations may have multiple MAC and protocol drivers.

The Protocol Manager is always defined in CONFIG.SYS to load before any MAC or protocol drivers. Its job is to read the configuration information out of the PROTOCOL.INI file and make this available to MAC and protocol drivers which load later.

MAC and protocol drivers use this information to set initialization parameters and allocate memory appropriately. For example, a NetBIOS driver may use the configuration information provided by the Protocol Manager to determine its maximum number of names and sessions.

As each driver configures and initializes itself, it identifies itself to the Protocol Manager using a driver-defined "module name" and "characteristics table". The module name defines a kind of logical name for the communication service provided by the driver. The characteristics table provides specific parameters about the service and the set of entry points the driver uses to communicate with other drivers. A single driver may identify itself to the Protocol Manager as multiple logical modules if, for example, it implements more than one layer of protocol interface (such as transport and data link).

Before two modules can communicate, they must be bound together. Binding is the process of two modules exchanging characteristics tables so that they can access each other's entry points. This establishes the linkage they need to make requests of one another and indicate asynchronous request completion. Binding is controlled by the Protocol Manager based on information from PROTOCOL.INI. Binding can be either static or dynamic for protocol drivers. If a protocol driver is static, then its binding is static. If it is dynamic, then its binding is dynamic. A dynamic protocol driver can be unbound from its bound drivers prior to unloading itself from memory. This unbinding process is also controlled through the Protocol Manager.

Configuration and Binding Process

In the typical case of a system with one MAC driver and a NetBIOS driver, the set of drivers load and initialize as follows:

1. Protocol Manager loads, initializes, and reads PROTOCOL.INI.

2. MAC driver loads. It calls GetProtocolManagerInfo to get any needed configuration information, like its DMA channel.

3. MAC driver initializes and calls RegisterModule to identify itself as the module named e.g. "ETHERCARD." This call passes ETHERCARD's characteristics table to Protocol Manager.

4. NetBIOS driver loads. It calls GetProtocolManagerInfo to get any needed configuration information, like the maximum number of names, sessions, and commands to support.

5. NetBIOS driver initializes and calls RegisterModule to identify itself as the module named "NetBIOS". This call passes NetBIOS's characteristics table to Protocol Manager and indicates that NetBIOS wants to bind to ETHERCARD.

6. After all device drivers have loaded, Protocol Manager determines from the information supplied on previous RegisterModule requests that NetBIOS must bind to ETHERCARD. Using a defined dispatch address in the characteristics table for NetBIOS, Protocol Manager calls NetBIOS and instructs it to bind to ETHERCARD. The call, InitiateBind, includes the characteristics table for ETHERCARD.

7. NetBIOS calls ETHERCARD, requesting to Bind. The modules exchange characteristics tables with each other. They now have each other's entry points and are bound.

8. NetBIOS may now call ETHERCARD at its defined entry points for transmitting and receiving packets (see next section).

If the example NetBIOS driver was dynamically loadable, the binding to the ETHERCARD MAC would be done through the Protocol Manager's VECTOR facility (see Chapter 7). The Vector shields the static MAC driver from the details of dynamic binding.

Chapter 3: Protocol to MAC Interface Description

The interface between a protocol and MAC driver provides for the transmission and reception of network packets, called frames. The interface includes other functions for controlling and determining the status of the network adapter controlled by the MAC.

To allow for efficient use of memory and to minimize buffer copies, frames being transmitted and received are passed between protocol and MAC using a scatter/gather buffer description convention. This passes an array of pointers/lengths called a frame buffer descriptor. There are three types of these descriptors, one for describing frames being transmitted (TxBufDescr) and two for frames being received (RxBufDescr and TDBufDescr).

Overall, the calls at the protocol/mac interface are grouped into categories of transmission, reception, indication control, status indications, and general requests. An additional category of function, system requests, is generic to all drivers.

Transmission

Transmitting data can work either synchronously or asynchronously, at the option of the MAC. Protocols must be able to handle both cases. Primitives are TransmitChain and TransmitConfirm.

Protocol		MAC
Transmit Chain	—CALL—>	Call passes TxBufDescr and unique handle. MAC may copy data now or later.
	<—RETURN—	Return indicates if data has been copied. If not, MAC now owns frame data blocks and will copy them asynchronously.

Later on, after data is copied by MAC:

TransmitConfirm	<—CALL—	Call supplies unique handle from Transmit.
	—RETURN—>	Data block ownership returned to protocol.

NOTE: If the MAC transmits the frame synchronously, it indicates this on the return from TransmitChain and will not generate a TransmitConfirm.

Reception

Receiving data can work in either of two ways, depending on the MAC. Protocols must be able to handle both cases.

- The MAC generates a ReceiveLookahead indication that points to part or all of the received frame in contiguous storage. This is called the "lookahead" data. The protocol may issue a TransferData call back to the MAC if it wants the MAC to copy all or part of the received frame to protocol storage. The protocol may, of

course, copy the look ahead data itself. In some implementations, this may be the entire frame.

- The MAC generates a ReceiveChain indication that points to a RxBufDescr that describes the entire frame received. The protocol may copy the data immediately or later. If later, it releases the frame buffer areas back to the MAC via a call to ReceiveRelease.

Generally, the first approach will be implemented by MAC drivers for non-host buffered network adapters, while drivers for host buffered network adapters will implement the second. Non-host buffered adapters that use programmed I/O or DMA will generally provide a small leading portion of the received frame as look ahead data, whereas those using a single memory mapped buffer will usually provide the whole frame.

In either case, the protocol must validate the received packet very rapidly (within a few instructions) and to reject it if necessary. This is very important to performance in a multi-protocol environment.

The following sections illustrate the non host-buffered adapter versus host-buffered adapter receive scenarios:

Non Host-Buffered Adapter

MAC		Protocol
ReceiveLookahead	—CALL—>	Call passes pointer to lookahead data. Protocol examines this data.

If protocol wants the frame and look ahead wasn't the whole frame, the protocol can ask MAC to transfer the frame:

TransferData	<—CALL—	Passes TDBufDescr indicating where to put the received data.
	—RETURN—>	
	<—RETURN—	

Upon return from protocol, MAC re-enables the hardware.

IndicationComplete	—CALL—>	MAC calls protocol to allow interrupt-time post processing.
	<—RETURN—	

Host-Buffered Adapter

MAC		Protocol
ReceiveChain	—CALL—>	Call passes pointer to RxDataDescr.

	<—RETURN—	Return tells if protocol accepts the frame, and if so, whether it copied the data. If accepted but not copied, ownership of data blocks passes to the protocol which copies the data asynchronously.
IndicationComplete	—CALL—>	MAC calls protocol to allow interrupt-time post processing.
	<—RETURN—	

Later, if protocol deferred copying the data (this may occur during IndicationComplete)

	<—CALL—	ReceiveRelease. The call supplies the unique handle from ReceiveChain.
	—RETURN—>	Data block ownership returned to MAC.

Indication Control

Two primitives let a protocol selectively control when it can be called with indications from the MAC. These are IndicationOn and IndicationOff.

Before calling an indication routine, the MAC implicitly disables indications. This means, for example, that if another frame arrives while the protocol is processing the indication for the previous one, the protocol will not be reentered. Likewise, if the protocol issues a TransmitChain for loopback data from within the ReceiveLookahead indication routine, it will not be reentered to process the loopback data reception.

Protocols can re-enable indications upon returning from ReceiveLookahead, ReceiveChain or Status indications or by calling IndicationOn within the IndicationComplete routine.

Status Indication

Status indications are calls from a MAC to protocol that convey a change in adapter or network status.

A status indication works much like a reception indication. The status indication handler is entered with indications disabled and there is a mechanism which will leave indications disabled.

MAC		Protocol
Status	—CALL—>	Call passes status type and information.
	<—RETURN—	
IndicationComplete	—CALL—>	MAC calls protocol to allow interrupt-time post processing.
	<—RETURN—	

General Requests

General requests are calls from a protocol to a MAC, asking it to do a general function such as open or close the network adapter or change the station address.

General requests work much like a TransmitChain request, except the primitives are Request and RequestConfirm.

Protocol		**MAC**
Request	—CALL—>	Issue request to MAC with unique handle.
	<—RETURN—	Return indicates if request completed.

Later, if request completed asynchronously:

	<—CALL—	RequestConfirm. The call supplies unique handle from Request.
	—RETURN—>	

If the MAC satisfies the request synchronously, it indicates this on the return from Request and will not generate a RequestConfirm.

System Requests

System requests support module binding and management functions. They are usually made by the Protocol Manager to a MAC or protocol module, but can also be made by a protocol to another protocol or MAC module.

System requests work much like general requests except that all are synchronous and the requests are not module specific.

Upper Module		**Lower Module**
System	—CALL—>	Issue request to lower module.
	<—RETURN—	Return indicates request completed.

Protocol Manager Primitives

Protocol Manager primitives are requests from protocol or MAC modules to the Protocol Manager for various Protocol Manager services. These requests are always synchronous.

Protocol or MAC Module		**Protocol Manager**
Primitive	—CALL—>	Issue request to Protocol Manager
	<—RETURN—	Return indicates request completed

Chapter 4 - Data Structures

Module Characteristics

Protocol and Media Access Control (MAC) modules are described by a data structure called a characteristics table. Each characteristics table consists of several sections: a master section called the common characteristics table and four subtables. The common characteristics table contains module-independent information, including a dispatch address for issuing system commands like InitiateBind to the module. The four module-specific subtables are chained off the common characteristics table. These define module-specific parameters and the entry points used for inter-module communication (such as the MAC/protocol interface functions). When two modules bind together, they exchange pointers to their common characteristics tables, so that each gets access to the other's descriptive information and entry points.

NOTE: NDIS drivers must copy the Module DS and entry point addresses (from the Common Characteristics and Upper/Lower Dispatch Tables) to their local data segment at Bind time. In future versions of this specification, this information may be volatile. Having this information directly accessible will also improve performance. This information must not be copied prior to the Bind call and must not be used unless the Bind completes successfully.

NOTE: The information in the characteristics table for a module is primarily informational, in support of network management and configuration tools. Upper modules binding to lower ones will NOT need to parse this information to adapt their behavior at the interface. They will generally just use the information to validate that they have been bound to the correct type of module. Most of the other information is provided in the structure to support information utilities.

Some new fields have been added to some of the characteristics tables for V2.0.1. The size/length fields at the start of the tables can be checked to see if the new fields are available in the table.

Common Characteristics

The format of this information is identical for all modules. Note that all information in this section of the table is static.

WORD	Size of common characteristics table (bytes)
BYTE	Major NDIS Version (2 BCD digits - 02 for this version)
BYTE	Minor NDIS Version (2 BCD digits - 00 for this version)
WORD	Reserved
BYTE	Major Module Version (2 BCD digits)
BYTE	Minor Module Version (2 BCD digits)
DWORD	Module function flags, a bit mask : 0 - Binding at upper boundary supported 1 - Binding at lower boundary supported 2 - Dynamically bound (i.e., this module can be swapped out) 3-31 - Reserved, must be zero
BYTE[16]	Module name - ASCIIZ format

BYTE	Protocol level at upper boundary of module: 1 - MAC 2 - Data link 3 - Network 4 - Transport 5 - Session -1 - Not specified
BYTE	Type of interface at upper boundary of module: For MAC's: 1 => MAC For Data Links: To be defined For Transports: To be defined For Session: 1 => NCB For any level: 0 => private (ISV defined)
BYTE	Protocol level at lower boundary of module 0 - Physical 1 - MAC 2 - Data link 3 - Network 4 - Transport 5 - Session -1 - Not specified
BYTE	Type of interface at lower boundary of module: For MAC: 1 => MAC For Data Link: To be defined For Transport: To be defined For Session: 1 => NCB For any level: 0 => private (ISV defined)
WORD	Module ID filled in by Protocol Manager on return from RegisterModule
WORD	Module DS
LPFUN	System request dispatch entry point
LPBUF	Pointer to service-specific characteristics (NULL if none)
LPBUF	Pointer to service-specific status (NULL if none)
LPBUF	Pointer to upper dispatch table (see below; NULL if none)
LPBUF	Pointer to lower dispatch table (see below; NULL if none)
LPBUF	Reserved for future expansion, must be NULL
LPBUF	Reserved for future expansion, must be NULL

NOTE:	LPSZ	Long pointer to an ASCIIZ string
	LPBUF	Long pointer to a data buffer
	LPFUN	Long pointer to a function

In V1.0.1, the 2 bytes after the first WORD were required to be set to 0. For compatibility with V1.0.1, an NDIS spec major version number of 00 is interpreted the same as major version number 01.

The module function flags bit mask must accurately specify the capabilities of the module. The Protocol Manager uses these fields; e.g. the "Dynamically bound" (bit 2) flag when set indicates that this module is a dynamically loadable and unloadable module. Such a module can only be used in the Protocol Manager dynamic mode.

The upper and lower boundary protocol level and interface type bytes must accurately specify the protocol level and interface type. The Protocol Manager uses these fields. If an interface does not support NDIS bindings or a protocol level is undefined at the interface, a value at OxFF must be used. In this case the corresponding interface type is undefined.

In addition to the above common characteristics table, a given module will typically have a set of sub-tables that are chained off the common table:

- Service-specific characteristics table:
  This table contains descriptive information and parameters about the module.

- Service-specific status table:
  This table contains runtime operating status and statistics for the module.

- Upper dispatch table:
  This table contains dispatch addresses for the upper boundary of the module — i.e., the entry points it exports as a service provider.

- Lower dispatch table:
  This table contains dispatch addresses for the lower boundary of the module — i.e., the entry points it exports as a service client.

NOTE: Under OS/2 dispatch addresses and data segments are Ring 0 selectors. This field is usually set at Ring 3 INIT time even though the selector set must be Ring 0.

MAC Service-Specific Characteristics

All MAC's use the following format for this table. This table contains volatile information (like the current station address) which may be updated by the MAC during the course of operation. Other modules may read this table directly during execution.

WORD	Length of MAC service-specific characteristics table
BYTE [16]	Type name of MAC, ASCIIZ format: 802.3 802.4 802.5 802.6 DIX DIX+802.3 APPLETALK ARCNET FDDI SDLC BSC HDLC ISDN
WORD	Length of station addresses in bytes
BYTE [16]	Permanent station address
BYTE [16]	Current station address
DWORD	Current functional address of adapter (0 if none)
LPBUF	Multicast Address List (structure defined below)
DWORD	Link speed (bits/sec)
DWORD	Service flags, a bit mask: 0 - broadcast supported 1 - multicast supported 2 - functional/group addressing supported 3 - promiscuous mode supported

4 - software settable station address
5 - statistics are always current in service-specific status table
6 - InitiateDiagnostics supported
7 - Loopback supported
8 - Type of receives
0 - MAC does primarily ReceiveLookahead indications
1 - MAC does primarily ReceiveChain indications
9 - IBM Source routing supported
10 - Reset MAC supported
11 - Open / Close Adapter supported
12 - Interrupt Request supported
13 - Source Routing Bridge supported
14 - GDT virtual addresses supported
15 - Multiple TransferDatas permitted during a single indication (V2.01 and later)
16 - Mac normally sets FrameSize = 0 in ReceiveLookahead (V2.01 and later)
17-31 - Reserved, must be 0

WORD	Maximum frame size which may be both sent and received
DWORD	Total transmission buffer capacity in the driver (bytes)
WORD	Transmission buffer allocation block size (bytes)
DWORD	Total reception buffer capacity in the driver (bytes)
WORD	Reception buffer allocation block size (bytes)
CHAR[3]	IEEE Vendor code
CHAR	Vendor Adapter code
LPSZ	Vendor Adapter description
WORD	IRQ Interrupt level used by adapter (V2.0.1 and later)
WORD	Transmit Queue Depth (V2.0.1 and later)
WORD	Maximum number of data blocks in buffer descriptors supported (V2.0.1 and later)

Remaining bytes in table (based on Length) are vendor-specific

In interpreting these tables the implementer must always bear in mind that additional functions may be added to future MAC's and that the support of functions that the protocol does not need must not prevent the protocol from accepting a bind for the MAC.

The type name describes to the protocol the type of MAC protocol header that the MAC driver supports. In general, protocol stacks must be prepared to support the types "802.3", "802.5", "DIX" and "DIX+802.3". If the native media of the MAC is not one of these types (for example, ARCNET) then it is recommended that the MAC developer must consider claiming support for one of the above types and doing a transparent internal mapping between the private header format of the media and the public header format being claimed. Without support for one of the above header formats, general protocol compatibility cannot be guaranteed. The list specified above is not exhaustive. New names may be added in the future or a vendor may provide special MAC type names for use with protocols that interoperate with special MACs provided by that vendor. In the latter case it is recommended that a vendor use a MAC type name that does not start with an alphanumeric character to avoid conflicts with NDIS MAC type names that might be specified in future versions of this specification.

The normal type name of an ethernet MAC would be "DIX+802.3." See Appendix B for references on IEEE 802.3 and DIX.

In the various parts of this specification, all station and multicast addresses for a given MAC have the length specified in the "Length of Station Address" field.

The permanent station address must be obtained from the hardware if at all possible, as it may be used by LAN Manager for security or administrative purposes. If a PROTOCOL.INI entry is used to override the current station address, the permanent station address must not be affected. Only if there is no hardware based addressing scheme will it be possible to override the permanent station address by configuration parameters. The current station address will always reflect the current address as set via parameter or by calling Request SetSetationAddress.

The functional address DWORD represents the functional address bit pattern present in the last 4 bytes of an IBM compatible functional address. This excludes the first 2 bytes 0xC0, 0x00. The functional address DWORD represents the logical OR of all functional addressess currently registered to the adapters.

Multicast Address List is a buffer formatted as follows:

WORD	Maximum number of multicast addresses
WORD	Current number of multicast addresses
BYTE[16]	Multicast address 1
BYTE[16]	Multicast address 2
	. . .
BYTE[16]	Multicast Address N

The Multicast Address List is kept packed by the MAC so that the current multicast addresses occur first in the list.

Service flags indicate which optional functions are supported by an NDIS driver. If a particular function bit is set, that function is supported. When attempts are made to invoke unsupported functions, NDIS MAC drivers must respond properly by returning INVALID_FUNCTION (0x0008).

If loopback is supported in the network adapter hardware, then bit 7 of the MAC service flags must be set.

If loopback is not supported in hardware (bit 7 of the MAC service flags is not set), the protocol driver must handle this function itself by some loopback delivery of the frame to be transmitted.

The following criteria must be met for loopback:

1. The destination address is the same as the local station's current station address or the destination is a broadcast, multicast or functional address which would have been received by this station if sent by another.

2. The frame must qualify for reception according to the current packet filter.

The loopback mechanism must cause the Receive indication to occur at interrupt time (and it must be delayed by IndicationOff)

If IBM source routing is used (bit 9 is set) it is the protocol module's responsibility to encode and interpret appropriate source routing information. This bit merely implies that

the device is capable of sending packets with the "source routing bit" set in the source address so that a protocol may recognize such a packet.

While the ResetMAC function (bit 10) is optional, it is strongly recommended that those implementing the NDIS MAC driver support this function. Some protocol drivers may rely on this function to recover from hardware error conditions.

If the service flags indicate that OpenAdapter is supported (bit 11 is set), then the protocol driver must ensure that the adapter is open. The open status of an adapter can be determined by examining bit 4 of the MAC status in the MAC service-specific status table. If the adapter is not open, then the protocol must issue an OpenAdapter Request (normally during bind-time processing).

If Source Routing Bridge is set (bit 13) then it is implied that the MAC is capable of receiving all packets on the network which have the source routing bit set.

If GDT virtual addresses are supported (bit 14 is set) then Ring 0 GDT virtual addresses may be used to describe frames. All MAC's must support the use of physical addresses to describe frames; however, for some MAC's it is preferable to supply a GDT address if one is readily available. The GDT address must remain valid throughout the scope of its use by the MAC.

If bit 16 of the service flags field is set, then the MAC driver does not normally provide the total frame size of received data. In this case the MAC normally calls RecieveLookahead with the FrameSize parameter equal to 0. Setting this bit is optional. It is left to the MAC implementor to determine how frequently returning FrameSize equals 0 constitutes sufficient grounds to set this bit. However, this bit must be reset if the MAC always calls ReceiveLookahead with the FrameSize parameter non-zero or if a 0 FrameSize parameter is returned only for intermittent erroneous packet reception. For V1.0.1 compatibility, bit 16 reset gives no indication whether the MAC will return a zero FrameSize parameter or not. Some MAC and higher layer protocols do not support "length" fields within their encoding. Such protocols rely on knowing the size of valid frame data received at the MAC interface and then deduce the amount of data at their layer by stripping off the lower layer protocol headers. This bit warns such protocols that the required received frame size is not normally available at the MAC interface and that receive frames might not be able to be processed. Such protocols can refuse to bind to such MACs.

The maximum frame size must reflect the maximum size packet that can be both transmitted and received by the MAC client. This size must reflect only the bytes which actually cross the NDIS boundary. For Ethernet, this value is typically 1514, since the client does not specify the CRC bytes. Token Ring values vary but do not include the non-data SD, ED and FS bytes or the FCS.

The network adapter RAM is characterized by four parameters. The first is the number of bytes available for storing packets to be transmitted, usually one or two full-size packets in size. The second parameter is the allocation granularity, typically about 256 bytes, indicating how much memory would be occupied by a one byte packet pending transmission. The next two parameters are the number of bytes available for storing received packets and the receive packet granularity. Often these parameters will be affected by PROTOCOL.INI keywords (for example, specifying two transmit buffers rather than one), and it is required that these numbers accurately reflect the current adapter configuration. Protocol drivers may use these numbers to determine reasonable window sizes, and incorrect values may impact performance.

The intent of the IEEE Vendor and Vendor Adapter Codes is that, when used in combination, they uniquely identify this MAC driver for this adapter. The IEEE Vendor Code uniquely defines the vendor providing the MAC driver. The use of the IEEE Vendor Code avoids the need for any global registry of Vendor Adapter Codes. The IEEE Vendor Code is assigned by the IEEE and becomes the first three bytes of a six-byte IEEE 802 address. The Vendor Adapter Code specifies a particular MAC driver provided by the Vendor for an adapter. If the IEEE Vendor Code is assigned to the Vendor, the Vendor assigns a unique Vendor Adapter Code to each MAC driver provided. For those without an IEEE Vendor Code, a value of 0xFFFFFF is required. In this case, the Vendor Adapter Code is undefined.

The Vendor Adapter description string is an ASCIIZ string containing a description of the adapter that could be used to format an error message (for example, "3Com EtherLink II Adapter").

The transmit queue depth specifies the maximum number of TransmitChain requests the MAC can buffer internally. This number must be set to one if the TransmitChain implementation in the MAC is synchronous. Each queued TransmitChain request requires that the MAC driver copy at least the chain descriptor and immediate data, so this parameter is generally configurable through a PROTOCOL.INI keyword called MAXTRANSMITS. The protocol driver can use this queue depth to compute the amount of time a transmit might be queued up within the MAC.

The maximum number of data buffer blocks is the maximum number of blocks supported in Transmit, TransferData, and ReceiveChain buffer desciptors. For V1.0.1 backward compatibility this must be a minimum of 8. For V1.0.1 compatible MACs for which this field is absent, the maximum number assumed is 8.

The size of the NDIS defined part of the MAC specific characteristics table may increase in subsequent versions of the specification. If vendor specific information follows the NDIS defined information, a protocol using it must check the NDIS spec version number in the Common Characteristics table to determine where the NDIS specified information ends and the vendor specified information begins.

MAC Service-Specific Status Table

The MAC service-specific status and media-specific statistics tables provide information about the status of and traffic through a MAC. Since these tables can be updated by the MAC driver at interrupt time, a protocol must ensure that these tables are read with interrupts disabled. The MAC must update this table (and the media-specific statistics table if present) atomically.

WORD	Length of status table
DWORD	Date/time when diagnostics last run (0xFFFFFFFF if not run). Format is seconds since 12:00 Midnight January 1, 1970
DWORD	MAC status, a 32-bit mask: 0-2 - Opcoded as follows: 0 - Hardware not installed 1 - Hardware failed startup diagnostics 2 - Hardware failed due to configuration problem 3 - Hardware not operational due to hardware fault 4 - Hardware operating marginally due to soft faults 5-6 Reserved

7 - Hardware fully operational
3 - If set, MAC is bound, else not bound
4 - If set, MAC is open, else not open (if adapter doesn't support open/close function, set to 1 if hardware is functional)
5 - If set, adapter diagnostics are in progress (V2.0.1 and later)
6-31 - Reserved, must be zero

WORD Current packet filter, a bit mask:
0 - directed and multicast or group and functional
1 - broadcast
2 - promiscuous
3 - all source routing
4-15 - Reserved, must be zero

Statistics for MAC's

Statistics in **bold** are mandatory, all others are strongly recommended.
0xFFFFFFFF means not supported.
Reserved slots should return as 0xFFFFFFFF (unsupported).

LPBUF	Pointer to media specific statistics table (may be NULL)
DWORD	Date/time when last ClearStatistics issued (0xFFFFFFFF if not kept) format is seconds since 12:00 Midnight January 1, 1970
DWORD	**Total frames received OK**
DWORD	**Total frames with CRC error**
DWORD	Total bytes received
DWORD	Total frames discarded - no buffer space
DWORD	Total multicast frames received OK
DWORD	Total broadcast frames received OK
DWORD	Reserved (Obsolete statistic)
DWORD	Reserved (Obsolete statistic)
DWORD	Reserved (Obsolete statistic)
DWORD	Reserved (Obsolete statistic)
DWORD	Reserved (Obsolete statistic)
DWORD	Total frames discarded - hardware error
DWORD	**Total frames transmitted OK**
DWORD	Total bytes transmitted OK
DWORD	Total multicast frames transmitted
DWORD	Total broadcast frames transmitted
DWORD	Reserved (Obsolete statistic)
DWORD	Reserved (Obsolete statistic)
DWORD	Total frames not transmitted - time-out
DWORD	Total Frames not transmitted - hardware error

Remaining bytes (based on Length) in table are vendor specific.

All statistics counters are 32-bit unsigned integers that wrap to zero when the maximum count is reached after which the counters will continue to count. When updating these counters, a frame is counted in all the supported counters that apply. The case of an unsupported counter (0xFFFFFFFF) can be distinguished from the situation wherby a counter is about the wrap around if the values of the counters are checked at bind times. If the initial value of the counter is 0xFFFFFFFF, then the counter is not supported. Otherwise the counter is supported and 0xFFFFFFFF at a later time means the counter is about to wrap around.

SERVICE SPECIFIC STATISTICS DEFINITIONS:

Total frames received ok
(NumberOfFramesReceivedOK) - corresponding 802.3 statistics

This contains a count of frames that are successfully received. It does not include "frames with errors", as listed in non-media specific statistics item 7.

Frames received with CRC error
(NumberOfFramesReceivedWithFrameCheckSequenceErrors)

This contains a count of frames that are an integral number of bytes in length and do not pass the FCS check. Reports on CRC errors "as the station sees it".

Total bytes received ok

This contains a count of bytes in frames that are successfully received. It includes bytes from received multicast and broadcast frames. This number should include everything, starting from destination address up to but excluding FCS. Source address destination address, length (or type) and pad are included. It should exclude FCS and the preambles.

According to this definition, this NDIS statistics is not exactly the same as 802.3's NumberOfBytesReceivedOK, which does not include the octets in the address and length/type fields.

Frames discarded - no buffer space

Frames discarded by MAC driver due to a lack of buffer space.

Multicast frames received ok.
(NumberOfMulticastFramesReceivedOK)

This includes all of the multicast frames the MAC driver received successfully.

It does not include "frames with errors" as listed in non-media specific statistics item 7.

Broadcast frames received ok.
(NumberOfBroadcastFramesReceivedOK)

This includes all of the broadcast frames the MAC driver receives successfully.

It does not include "frames with errors" as listed in non-media specific statistics item 7.

Frames discarded - hardware error

Frames discarded due to hardware error.

Definition of this statistic should be adapter specific.

Total frames transmitted ok.
(NumberOfFramesTransmittedOK)

Total number of frames transmitted successfully.

Total bytes transmitted ok.

Total number of bytes transmitted successfully.

This number should include everything, starting from destination address up to but excluding FCS. Source address destination address, length (or type) and pad are included. It should exclude FCS and the preambles.

Multicast frames transmitted ok.
(NumberOfMulticastFramesTransmittedOK)

Number of frames transmitted successfully to non-broadcast group address.

Broadcast frames transmitted ok.
(NumberOfBroadcastFramesTransmittedOK)

Number of frames transmitted successfully to broadcast address.

Frames not transmitted - time-out

This contains a count of frames that could not be transmitted due to the hardware not signaling transmission completion for an excessive period of time.

Frames not transmitted - hardware error

This contains a count of frames that could not be transmitted due to a hardware error. This count should exclude DMA underrun error which itself is a separate counter (Frames transmitted with underun). Definition of this statistic should be adapter specific.

MAC Upper Dispatch Table

The number and meaning of dispatch addresses provided here apply to the boundary between a MAC and a protocol. This may differ at other protocol boundaries. Note that each upper/lower module binding may have its own unique set of dispatch addresses that is set up when the modules exchange characteristics tables. This can be achieved by exchanging copies of the common characteristics table, where the copy has the desired pointers to the specific dispatch tables for the binding.

LPBUF Back pointer to common characteristics table

LPFUN	Request address
LPFUN	TransmitChain address
LPFUN	TransferData address
LPFUN	ReceiveRelease address
LPFUN	IndicationOn address
LPFUN	IndicationOff address

NOTE: No dispatch address is allowed to be NULL.

Protocol Service-Specific Characteristic Table

For compatibility with future versions of this specification, all protocols must provide a protocol service-specific characteristics table which starts with the following fields:

WORD	Length of protocol service-specific characteristics table
BYTE [16]	Type name of protocol, ASCIIZ format:
WORD	Protocol type code

This may be followed by protocol-specific information.

The protocol type name will be used in future versions of this specification. Specific type names for different protocol types will be defined later. Protocol type codes will also be defined later. For the moment these two fields are simple place holders and must be set to null string and zero respectively.

Protocol Lower Dispatch Table

The protocol lower dispatch table is specified in the characteristics table for the protocol binding to the MAC. The characteristics table for the MAC actually does not supply a lower dispatch table (the pointer to it is NULL).

LPBUF	Back pointer to common characteristics table
DWORD	Interface flags (used by Vector frame dispatch): 0 - Handles non-LLC frames 1 - Handles specific-LSAP LLC frames 2 - Handles non-specific-LSAP LLC frames 3-31 - Reserved must be zero
LPFUN	RequestConfirm address
LPFUN	TransmitConfirm address
LPFUN	ReceiveLookahead indication address
LPFUN	IndicationComplete address
LPFUN	ReceiveChain indication address
LPFUN	Status indication address

NOTE: No dispatch address is allowed to be NULL.

Characteristic Tables for NetBIOS Drivers

NetBIOS drivers written to the existing LAN Manager Ring0 NetBIOS specification can be adapted to fit into the Protocol Manager structure by defining a common characteristics table for them shown below. Note that such a NetBIOS driver must still respond to the

existing LAN Manager NetBIOS Linkage binding mechanism; these drivers will only use Protocol Manager binding at their lower boundary (to the MAC). A variant kind of NetBIOS module will be defined in the future that takes advantage of Protocol Manager binding at both boundaries.

Common characteristics for NetBIOS drivers:

WORD	Size of common characteristics table (bytes)
BYTE	Major NDIS Version (2 BCD digits)
BYTE	Minor NDIS Version (2 BCD digits)
WORD	Reserved
BYTE	Major Module Version (2 BCD digits)
BYTE	Minor Module Version (2 BCD digits)
DWORD	Module function flags, 0x00000002 (binds lower)
BYTE[16]	NetBIOS Module name
BYTE	Protocol level at upper boundary of module: 5 = Session
BYTE	Type of interface at upper boundary of module: 1 = LANMAN NCB
BYTE	Protocol level at lower boundary of module: 1 = MAC
BYTE	Type of interface at lower boundary of module: 1 = MAC
WORD	NetBIOS Module ID
WORD	NetBIOS Module DS
LPFUN	System request dispatch entry point
LPBUF	Pointer to service-specific characteristics (see below)
LPBUF	Pointer to service-specific status, must be (NULL)
LPBUF	Pointer to upper dispatch table (see below)
LPBUF	Pointer to lower dispatch table (see below)
LPBUF	Reserved, must be NULL
LPBUF	Reserved, must be NULL

Upper dispatch table for a NetBIOS module:

LPBUF	Back pointer to common characteristics table
LPFUN	Request address
LPFUN	NetBIOS NCB handler (LANMAN calling conventions)

Lower dispatch table for a NetBIOS module:

LPBUF	Back pointer to common characteristics table
DWORD	Interface flags (used by Vector frame dispatch): 0 - Handles non-LLC frames 1 - Handles specific-LSAP LLC frames 2 - Handles non-specific-LSAP LLC frames 3-31 - Reserved must be zero
LPFUN	RequestConfirm address
LPFUN	TransmitConfirm address
LPFUN	ReceiveLookahead indication address
LPFUN	IndicationComplete address
LPFUN	ReceiveChain indication address
LPFUN	Status indication address

Service-specific characteristics for a NetBIOS module:

WORD	Length of NetBIOS module service-specific characteristics table
BYTE [16]	Type name of NetBIOS module, ASCIIZ format:

WORD NetBIOS module type code

This may be followed by module-specific information.

The protocol type name will be used in future versions of this specification. Specific type names for different protocol types will be defined later. Protocol type codes will also be defined later. For the moment these two fields are simple place holders and must be set to null string and zero respectively.

Frame Data Description

The MAC describes frame data with a data structure called a buffer descriptor. The descriptor is composed of pointers and lengths which describe a logical frame. Buffer descriptors are ephemeral objects. A descriptor is valid only during the scope of the call that references it as a parameter. The called routine must not modify the descriptor in any way. If the called routine needs to refer to the described data blocks after returning from the call, it must save the information contained in the descriptor.

Data blocks described by descriptors are long-lived. Ownership of the data blocks is implicitly passed to the module that is called with the descriptor. The called module relinquishes ownership back to the caller either via setting a return argument, or by later issuing a call back to the supplying module. Under OS/2, some pointers may be either GDT virtual addresses or physical addresses. In this case the pointer has an associated pointer type opcoded field. Defined values are 0 for physical address and 2 for GDT virtual addresses. GDT virtual addresses may be supplied to the MAC only if bit 14 of the service flags in the MAC service specific characteristics table is set. The GDT address must remain valid throughout the scope of its use by the MAC.

Under DOS there is no distinction between physical and virtual addresses. All addresses in this case are segment: offset. Care must be taken to ensure that the segment offset plus data length do not exceed the 64K segment boundary. The pointer type field if present is always encoded as a 0.

For performance reasons,it is recommended that data blocks used for transmission and reception be double-word aligned where possible. Both MAC and protocol NDIS drivers may choose to perform byte, word or dword memory movement without first ensuring proper alignment. This will result in reduced performance in combination with drivers which do not guarantee such alignment.

A buffer descriptor may contain one or more data blocks of length zero. In this case the other fields in the data block (Data Ptr and Data Type) may not be valid and must be ignored.

Transmit Buffer Descriptor

All transmit data is passed using a far pointer to a transmit buffer descriptor, TxBufDescr. The format of this descriptor is:

```
WORD    TxImmedLen    ;Byte count of immediate data; max is 64
LPBUF   TxImmedPtr    ;Virtual address of immediate data
WORD    TxDataCount   ;Count of remaining data blocks; max is configurable
```

Followed by TxDataCount instances of:

```
BYTE    TxPtrType    ;Type of pointer (0=Physical, 2=GDT)
BYTE    TxResByte    ;Reserved Byte (must be 0)
WORD    TxDataLen    ;Length of data block
LPBUF   TxDataPtr    ;Address of data block
```

In a TxBufDescr structure, the immediate data described by the first two fields is ephemeral and may be referenced only during the scope of the call that supplies it. Such immediate data is always transmitted before data described by TxDataLen and TxDataPtr pairs. If the called routine needs to refer to the immediate data after returning from the call, it must copy the data. The maximum size of immediate data is 64 bytes. For V2.0.1 MACS or later the maximum TxDataCount is specified in the MAC specific characteristics table. For V1.0.1 MACs the maximum count is 8.

Transfer Data Buffer Descriptor

Transfer data can be described by a far pointer to a transfer data buffer descriptor, TDBufDescr. Transfer data buffer descriptors have the following format:

```
WORD    TDDataCount    ; Count of transfer data blocks; max is configurable
```

Followed by TDDataCount instances of:

```
BYTE    TDPtrType    ;Type of pointer (0=Physical, 2=GDT)
BYTE    TDResByte    ;Reserved Byte (must be 0)
WORD    TDDataLen    ;Length of data block
LPBUF   TDDataPtr    ;Address of data block
```

For V2.0.1 MACs or later the maximum TDDataCount is specified in the MAC specific characteristics table. For V1.0.1 MACs the maximum count is 8.

Receive Chain Buffer Descriptor

Receive chain data can be passed by a far pointer to a receive chain buffer descriptor, RxBufDescr. Receive chain buffer descriptors have the following format:

```
WORD    RxDataCount    ;Count of receive data blocks; max is configurable
```

Followed by RxDataCount instances of:

```
WORD    RxDataLen    ;Length of data block
LPBUF   RxDataPtr    ;Virtual address of data block
```

For V2.0.1 MACs or later the maximum receive data block count is specified in the MAC specific characteristics table. For V1.0.1 MACs the maximum count is 8.

For received frames that are larger than 256 bytes, the first data block of the frame must be at least 256 bytes long. Frames less than or equal to 256 bytes will be passed up with RxDataCount equal to 1.

PROTOCOL.INI

The PROTOCOL.INI file stores configuration and binding information for all the protocol and MAC modules in the system. The file uses the same general format as the LANMAN.INI file. It consists of a series of named sections, where the section name is in fact the module name from a module characteristics table. Below the bracketed module name is a set of configuration settings for the module in name=value format. For example:

```
[MYNetBIOS]
Drivername = NetBIOS$
Bindings = ETHERCARD
MaxNCBs = 16
MaxSessions = 32
MaxNames = 16
```

The rules for PROTOCOL.INI contents are:

- Bracketed module name. Must be the name of a protocol or MAC module, e.g. [MYNetBIOS]. This is the name of the module as defined in that module's characteristics table. The name must be 15 characters or less (not counting the brackets). Mixed case may be used but the Protocol Manager will convert it to uppercase when it reads the file into memory.

- Drivername = <device driver name>. This parameter is required for all device driver modules. It defines the name of the OS/2 or DOS device driver that the module is contained in. Note that a single device driver name may be mentioned by several sections of the PROTOCOL.INI file, if the driver contains multiple logical modules. The Drivername parameter is the recommended method by which a module searches for its module section in the PROTOCOL.INI file to get its configuration parameters. This allows the module to find all relevant module sections based on a single name intrinisic to the module independent of the particular bracketed module name used in the PROTOCOL.INI file. This keyword is also required for DOS dynamic modules like TSRs or transient application modules. Although there is no driver name instrinsically assigned to such modules it is required that a unique name be assigned to this keyword for such modules anyway. In this way the same search mechanism used by device drivers can be used by dynamic DOS modules to find their relevant module sections in PROTOCOL.INI.

- Bindings = <module name> | <module name>,<module name>, ... This parameter is optional for protocol modules. It is not valid for MAC modules. If present, it is used by the protocol module to determine what MAC modules it will ask to bind to. (In other words, changing this parameter in the PROTOCOL.INI file can reconfigure a protocol to bind to a different MAC.). The Bindings parameter may be omitted if the protocol driver software is preconfigured to bind to a particular MAC, or if the system will only contain one MAC and one static protocol module. In the latter case (only in static mode), the Protocol Manager by default will ask the one static protocol to bind to the one MAC.

- Other keywords and parameters. Any other keyword = value statements are module specific. Keyword names must be 15 characters or less. They may be mixed case but are converted to uppercase when read by the Protocol Manager.

Note that keyword names are unique within the scope of each <module name> section and can appear within the section in any order.

- Whitespace around the equals sign is not significant, nor is trailing white space on the line. Except for this leading and trailing white space, all other characters of the value string are taken verbatim.

- A list of 0 or more parameters can appear to the right of the equals sign. If there are no parameters the equals sign can be optionally omitted. A parameter is terminated by a space, tab, comma, or semicolon. No parameters are interpreted by the Protocol Manager.

- A parameter can either be up to a 31-bit signed numeric value or a string of any length.

- A numeric parameter can be expressed either in decimal or hexadecimal format. All numeric parameters must start with the characters '0' through '9' or by a + or - followed by the '0' to'9' character. A hexadecimal parameter must start with '0x' or '0X' and use valid hexadecimal digits. A non-hexadecimal numeric parameter is treated as decimal integer. A parameter not surrounded by quotes and starting with 0 to 9 or + and - followed by 0 to 9 will be assumed to be a numeric parameter.

- A string is a parameter which either starts with a non-numeric character or is surrounded with quotes ("...."). The string is preserved in the memory image as it appears in PROTOCOL.INI.

- A line starting with a semicolon in column 1 is a comment and is ignored. Blank lines are ignored too.

- Lines may be as long as required. Continuation lines are not supported. Lines end with CR LF.

- Tabs, formfeeds, and spaces are considered to be white space.

The Protocol Manager supports an optional section with optional keywords defined below:

```
[PROTMAN]
Drivername = PROTMAN$
Dynamic = YES or NO
PRIORITY = prot1, prot2, ...
Bindstatus = YES or NO
```

The bracketed module name can be any valid name as long as it is unique within this PROTOCOL.INI. Drivername is required and must be assigned PROTMAN$, identifying the section as belonging to the Protocol Manager. None of the entries are case-sensitive.

The DYNAMIC keyword is optional. It defaults to NO if not present. If set to NO, the Protocol Manager operates only in the static mode and does not support dynamic protocol drivers. If set to YES, the Protocol Manager operates in the dynamic mode and supports both static and dynamic binding.

The PRIORITY keyword is optional. If absent, then the VECTOR uses default demultiplexing priority if multiple protocol drivers are bound to the same MAC (see Vector Demultiplexing in Chapter 7). If present, the parameters on the right-hand side are

presumed to be a list of protocol module names, highest priority first. The VECTOR prioritizes protocol drivers for demultiplexing (if necessary) according to their order in the list, and packets are offered to the first protocol driver listed first. Protocol drivers not listed are assigned default priority AFTER those listed. It is not necessary that a protocol driver ever bind for it to be listed here.

The BINDSTATUS keyword is optional. If absent, then the BindStatus command is not supported by the Protocol Manager. If set to YES, then BindStatus is supported by the Protocol Manager. The default disable condition is a memory optimization feature primarily for DOS environments.

When syntax errors are detected in processing the PROTOCOL.INI commands, by convention, all NDIS drivers should:

1) Display a error message detail exact syntax problem.
2) Assume some *non-fatal* value for the parameter associated with the error and complete processing.

Configuration Memory Image

When the Protocol Manager initializes, it reads PROTOCOL.INI and parses it into a memory image that it makes available to MAC and protocol modules via the Get Protocol Manager Info call. The parsed image is formatted to make it easy for run-time modules to interpret. All information contained in PROTOCOL.INI is present in the memory image in the same order as in the file. (Comments and white space are of course not present in the image). Note that in static mode the image is only available during device driver initialization time. In dynamic mode the image may additionally be created by a utility which then registers it with the Protocol Manager.

The structure definitions defined below do not conform rigorously to C language syntax. They provide a pseudo C-like language to define the data structures encoded in the configuration memory image.

ConfigMemoryImage

The ConfigMemoryImage data structure defines the complete memory image for all logical devices read from the PROTOCOL.INI configuration file. It is a doubly linked list of ModuleConfig structures. Each ModuleConfig structure corresponds to one module. The ConfigMemoryImage structure is defined as follows:

```
struct ConfigMemoryImage
{
        struct Module Config(1) Module(1);
        struct Module Config(2) Module(2);
        . . .
        struct ModuleConfig(N) Module(N);
};
```

where:

N=the number of modules encountered by the Protocol Manager when parsing the configuration file PROTOCOL.INI.

ModuleConfig

The ModuleConfig(i) structure defines the memory image for configuration parameters corresponding to one (bracketed name) module. For the (i)th module specified in PROTOCOL.INI it is defined as follows:

```
struct ModuleConfig(i)
{
        struct ModuleConfig(i+1) far *NextModule;
        struct ModuleConfig(i-1) far *Prev Module;
        unsigned char Module Name [16];
        struct KeywordEntry(1) KeywordEntry(1);
        struct KeywordEntry(2) KeywordEntry(2);
        . . .
        struct KeywordEntry(N) KeywordEntry(N);
};
```

where:

N = the number of keyword entries encountered in the PROTOCOL.INI file for this module.

NextModule = a FAR pointer to the next module configuration structure. NULL if this is the structure for the last module. For OS/2 the selector is a Ring 3 selector. For DOS the pointer is a segment:offset pair.

PrevModule = a FAR pointer to the previous module configuration structure. NULL if this is the structure for the first module. For OS/2 the selector is a Ring 3 selector. For DOS the pointer is a segment:offset pair.

ModuleName = array containing the characters of the module name (given in brackets in the configuration file). This is an ASCIIZ string consisting of a maximum of 15 non-null uppercase characters.

KeywordEntry

For each keyword line in the configuration file for the module a memory image structure is created specifying the keyword and the parameter values. The (j)th keyword encountered in the PROTOCOL.INI file for the module is defined as follows:

```
struct KeywordEntry(j)
{
        struct KeywordEntry(j+1) far *NextKeywordEntry;
        struct KeywordEntry(j-1) far *PrevKeywordEntry;
        unsigned char Keyword[16];
        unsigned NumParams;
        struct Param(1) Param(1);
        struct Param(2) Param(2);
        . . .
        struct Param(N) Param(N);
};
```

where:

N = the number of parameters entered with the keyword. If N =0 the parameters are not present.

NextKeywordEntry = a FAR pointer to the next keyword entry structure in the memory image. NULL if this is the last keyword entry. For OS/2 the selector is a Ring 3 selector. For DOS the pointer is a segment:offset pair.

PrevKeywordEntry = a FAR pointer to the previous keyword entry structure in the memory image. NULL if this is the first keyword entry. For OS/2 the selector is a Ring 3 selector. For DOS the pointer is a segment:offset pair.

Keyword = the array containing the characters of the keyword found in the configuration file. This is an ASCIIZ string consisting of a maximum of 15 non-null characters. The case of alphabetic characters will be uppercase in the memory image.

NumParams = the number (N) of parameters entered with the keyword each parameter described by a param structure. The value is 0 if no parameters were present.

Param(k) = the (k)th parameter structure to specify the value of one parameter in a list of parameters for a keyword. "Param(k+1)" follows Param(k) in sequence within the memory image. Each parameter is delimited by a length field for the parameter. It is assumed that a keyword's fields will be parsed sequentially.

Param

For the (k)th parameter defined in a parameter list for a specific keyword the following structure defines its value and attributes:

```
struct Param(k)
{
        unsigned ParamType;
        unsigned ParamLen;
        union ParamValue
        {
                long Numeric;
                unsigned char String[STRINGLEN];
        };
};
```

where:

STRINGLEN = length of the ASCIIZ parameter string (including the terminating NULL) for string parameters.

ParamType = The type of parameter. The following types are supported:
0 - signed integer supporting up to 31 bits least significant byte first.
1 - a string of characters.

ParamLen = The length of the parameter value. The length could be one of the following either be 4 for numeric parameters or STRINGLEN for string parameters where STRINGLEN is the length of the string (including the terminating NULL).

Numeric = a 31-bit signed numeric value.

String = an ASCIIZ character string. The case of alphabetic characters in the string is preserved from that in PROTOCOL.INI.

The size of the Param (k) structure is thus ParamLen + 4.

BindingsList

For each module that registers with the Protocol Manager a BindingsList structure may be given to the Protocol Manager specifying the set of modules that the given module wishes to bind to. The current module will require services from these other modules. This structure is defined as follows:

```
struct BindingsList
{
        unsigned NumBindings;
        struct Module
        {
                char ModuleName[16];

        } BoundDriver[NUMBINDINGS];
};
```

where:

NumBindings = the number (NUMBINDINGS) of modules that the specified module wants to be bound to it from below. In the static default binding mode of one static protocol and one MAC, a value of 0 in this field means for the protocol that it will bind to the MAC. Otherwise in the non-default binding mode, a value of 0 in this field means that the module has no lower bindings.

ModuleName = an ASCIIZ string specifying the logical name of a module which the current module wishes to have bound to it from below. Maximum of 15 non-null characters. The Protocol Manager will convert all alphabetic characters to uppercase.

BoundDriver = an array of NUMBINDINGS module names specifying the list of modules to which the current module wants to be bound.

The order of the modules in the list is significant in that InitiateBind requests will be issued to the protocol module in this order.

Chapter 5 - Specification of Primitives

Implementers should obey the following general guidelines:

- All primitives specified in this section can be called in protected mode in either interrupt or task context under OS/2. Since any primitive may be called in interrupt context it is illegal to block during the execution of a primitive.

- All routines must run (as much as possible) with interrupts enabled. Interrupt handlers must dismiss the interrupt at the 8259 as soon as possible.

- An indication handler will normally be entered with interrupts enabled. The handler may enable or disable interrupts if it chooses and on return the MAC must assume that the interrupt state may have been changed.

- Under MS-DOS indication handlers must assume they have only 200 bytes of stack space. If more stack space is needed then the handler must supply a stack.

- Confirmation and IndicationComplete handlers must be fully re-entrant and are always entered with interrupts enabled. Under DOS Confirmation and IndicationComplete handlers must assume they are entered on whatever stack the interrupt occurred on.

- A confirmation handler may be entered with the confirmation for a request before the request has returned.

- It is recommended that a MAC release the internal resources associated with either TransmitChain or a request before calling the confirmation handler. This allows the protocol to submit a new TransmitChain or request from the confirmation handler. Failing to do so may have a significant impact on performance.

- A protocol must assume whenever it gives control to a MAC that interrupts may be enabled by the MAC unless otherwise explicitly specified.

- When passing a virtual address to one of these primitives under OS/2 the address must be a Ring 0 GDT address unless otherwise specified. The interrupt service routine portion of the MAC must handle the fact that this address may not be valid if an interrupt occurs in real mode.

- All primitives have a set of specific error codes defined. In general, MAC's and protocols must return these specific codes. However it is acceptable to return GENERAL_FAILURE for any non-recoverable failure. NDIS developers must be aware that new error codes may be added in the future and must design their code to allow for this.

- If a particular entry point or function is not supported by an NDIS protocol or MAC driver, the entry point must still be exposed and an error (INVALID_FUNCTION 0x0008) returned if it is called. Crashing when an unsupported request is made is unacceptable.

- Parameters are passed on the stack compatible with Microsoft C FAR Pascal calling conventions. On entry to any routine the called module must save the caller's DS before setting its DS from the "dataseg" parameter. At exit the caller's DS must be restored. Furthermore the called module must follow standard Microsoft C conventions about saving "register variable" SI and DI registers if these are used. Modules which use the 80386 registers EDI, ESI and EBP must preserve these registers also. The direction bit is assumed to be clear on entry and must be clear upon exit. These conventions apply for calls in both directions across the NDIS MAC interface.

- Direct calls return in AX a return code specifying the status of function invocation. Those functions specified as using IOCTLs return this in the status field of the request block.

- Before calling a module in OS/2 it is the caller's responsibility to ensure that it is currently executing in protected mode. If it is running in real mode it must do an OS/2 "RealToProt" DevHlp call before calling the inter-module interface function. Furthermore in OS/2 the inter-module call can only be made at post CONFIG.SYS INIT time since all selectors are Ring 0 selectors.

- A MAC starts with packet reception disabled. A protocol must call SetPacketFilter to enable reception of packets.

- It is recommended that the number of Request commands which can be simultaneously queued by the MAC be configurable. The suggested keyword in the configuration file is "MaxRequests." The recommended default is 6. The suggested range is 1 to 10.

- The number of TransmitChain commands which can be simultaneously queued by the MAC must be configurable. The suggested keyword in the configuration file is "MaxTransmits". The recommended default is 6. The suggested range is 1 to 50.

- On a DIX or 802.3 network, packet buffers received may have been padded to the minimum packet size for short packets. It is the responsibility of the MAC client to examine the length field if present and strip off the padding.

- For DIX or 802.3 networks the MAC client can transmit a buffer with packet length smaller than the minimum. It is the responsibility of the MAC to provide the required padding bytes before transmission on to the wire. The content of the padding bytes is undefined.

- Protocol drivers conforming to this specification are expected to format and interpret MAC headers for the MAC driver types supported. Generally, protocols are expected to support 802.3, DIX, and 802.5 MAC headers. It is recommended that MAC drivers for other media types consider claiming to be one of the above types and doing a transparent internal mapping between that and its own private MAC header format. In doing so, the MAC will be able to claim interoperability (assuming the appropriate testing is done) with most protocol drivers developed for LAN Manager.

- In the absence of any such conversion, the MAC header is passed protocol-to-MAC or MAC-to-protocol in exactly the format in which it exists on the medium. The CRC and non-data fields are not passed across this boundary. Therefore the Ethernet CRC and the Token Ring SD, FCS, ED and FS fields are not passed and

will not be included in the packet length. The protocol must convert header fields found in the header buffer passed up to whatever format is required to conveniently store them in local memory. For example multi-byte fields (e.g., 802.3 length) may not be received in the byte order that is normally used by the CPU for storing multi-byte parameters. For exact format of the MAC header refer to the appropriate standards document (see Appendix B).

- For performance reasons, it is recommended that PhysToGDT be used whenever possible instead of PhysToVirt.

- Commonly Used Parameters

ProtID — The unique module ID of the protocol, assigned at bind time by the Protocol Manager.

MACID — The unique module ID of the MAC, assigned at bind time by the Protocol Manager.

ReqHandle — A handle assigned by the protocol to identify this request. If the request is implemented asynchronously by the MAC driver in question, this handle is returned on the confirmation call used to indicate completion of the request. A ReqHandle of 0 indicates that the confirmation be unconditionally suppressed. For example, the request may still be handled asynchronously but there will be no notification of completion. A ReqHandle of 0 must not change the immediate return code.

ProtDS — DS value for called protocol module, obtained from the module's dispatch table at bind time.

MACDS — DS value for called MAC module, obtained from the module's dispatch table at bind time.

Direct Primitives

TransmitChain

Purpose: Initiate transmission of a frame

```
PUSHWORD    ProtID      ;Module ID of protocol
PUSHWORD    ReqHandle   ;Unique handle for this request or 0
PUSHLPBUF   TxBufDescr  ;Pointer to framebufferdescriptor
PUSHWORD    MACDS       ;DS of called MAC module
CALL TransmitChain
```

Returns:		
	0x0000	SUCCESS
	0x0002	REQUEST_QUEUED
	0x0006	OUT_OF_RESOURCE
	0x0007	INVALID_PARAMETER
	0x0008	INVALID_FUNCTION
	0x000A	HARDWARE_ERROR
	0x000B	TRANSMIT_ERROR
	0x000C	NO_SUCH_DESTINATION

0x00FF GENERAL_FAILURE

TxBufDescr Far pointer to the buffer descriptor for the frame.

Description:

This call asks the MAC to transmit data. The MAC may either copy the data described by TxBufDescr before returning, or queue the request for later (asynchronous) processing. The MAC indicates which option it is taking by setting the appropriate return code.

In the asynchronous case, ownership of the frame data blocks passes to the MAC until the transmission is complete; the protocol must not modify these areas until then. Ownership of the data blocks is returned to the protocol when the MAC either returns a status code which implies completion of the original request or calls its TransmitConfirm entry with the ReqHandle from TransmitChain. If a request handle of zero was used and therefore TransmitConfirm will not be called, then ownership must not be considered returned until the protocol receives a message that implies the transmission has occurred (e.g., receiving an ACK to the transmitted message).

Note that when doing asynchronous transmission, the MAC must retain any needed information from TxBufDescr, since the pointer to that structure becomes invalid upon returning from TransmitChain. Also, if the TxImmedLen of the descriptor is non-zero, the MAC must retain a copy of the immediate data at TxImmedPtr, since the immediate data area becomes invalid upon returning from TransmitChain.

The MAC header must fit entirely in the immediate data, if present, or in the first non-immediate element described in TxBufDescr if there is no immediate data.

A MAC must be prepared to handle a TransmitChain request at anytime, including from within interrupt-time indication routines.

The return code REQUEST_QUEUED will cause a TransmitConfirm to be called from the MAC back to the protocol if the ReqHandle on the TransmitChain call is not 0. All other return codes from TransmitChain imply that no TransmitConfirm will occur.

The TRANSMIT_ERROR and NO_SUCH_DESTINATION error codes are intended to allow a protocol to recreate the frame status byte on a Token Ring network. Thus, NO_SUCH_DESTINATION implies that the address recognized bits were not set (and therefore the frame was not copied), while TRANSMIT_ERROR merely means that the frame was not copied. Protocols which make use of Source Routing may need the NO_SUCH_DESTINATION error code to be completely conformant. Token Ring MAC driver writers must make every attempt to return these error codes properly.

TransmitConfirm

Purpose: Imply the completion of transmitting a frame.

```
PUSH WORD       ProtID      ;Module ID of Protocol
PUSH WORD       MACID       ;Module ID of MAC
PUSH WORD       ReqHandle   ;Unique handle from TransmitChain
PUSH WORD       Status      ;Status of original TransmitChain
PUSH WORD       ProtDS      ;DS of called protocol module
CALL TransmitConfirm
```

Returns:	0x0000	SUCCESS
	0x0007	INVALID_PARAMETER
	0x00FF	GENERAL_FAILURE

Description:

This routine is called by a MAC to indicate completion of a previous TransmitChain. The purpose of this is to return ownership of the transmitted data blocks back to the protocol.

The ProtID parameter must be the value passed by the protocol on the previous TransmitChain to identify the requestor.

The ReqHandle is the value passed by the protocol on the previous TransmitChain which identifies the original request.

TransmitConfirm does not necessarily imply that the packet has been transmitted, though it generally will have been (with the exception of some intelligent adapter implementations). If the packet has been transmitted, Status must indicate the final transmit status:

0x0000	SUCCESS
0x000A	HARDWARE_ERROR
0x000B	TRANSMIT_ERROR
0x000C	NO_SUCH_DESTINATION
0x00FF	GENERAL_FAILURE

See TransmitChain for more details.

ReceiveLookahead

Purpose: Indicate arrival of a received frame and offer lookahead data.

```
PUSH  WORD      MACID       ;Module ID of MAC
PUSH  WORD      FrameSize   ;Total size of frame (0 if not known)
PUSH  WORD      BytesAvail  ;Bytes of lookahead available in Buffer
PUSH  LPBUF     Buffer      ;Virtual address of lookahead data
PUSH  LPBYTE    Indicate    ;Virtual address of indicate flag
PUSH  WORD      ProtDS      ;DS of called protocol module
CALL  ReceiveLookahead
```

Returns:	0x0000	SUCCESS
	0x0003	FRAME_NOT_RECOGNIZED
	0x0004	FRAME_REJECTED
	0x0005	FORWARD_FRAME
	0x0006	OUT_OF_RESOURCE
	0x0007	INVALID_PARAMETER
	0x00FF	GENERAL_FAILURE

FrameSize — The total size, in bytes, of the received frame. A value of 0 indicates that the MAC does not know the total frame size at this time.

BytesAvail — The number of bytes available in the lookahead buffer. This is guaranteed to be at least as large as the lookahead size established with the SetLookahead request. For frames which are smaller than the lookahead size, the lookahead buffer will contain the whole frame.

Buffer — Virtual address of contiguous lookahead buffer. The buffer contains the leading BytesAvail octets of the frame. This buffer is ephemeral; it is addressable to the protocol only during the scope of the Receive call.

Indicate — Virtual address of indication flag byte. This byte is set to 0xFF by the MAC prior to this call. If the protocol clears the byte to zero prior to returning then indications will be left disabled until IndicationOn is called from IndicationComplete.

Description:

This routine is called by a MAC to indicate reception of a frame and to offer frame lookahead data. The protocol is expected to inspect this information very rapidly to determine if it wants to accept the frame or not. If it wants to accept the frame, it may call TransferData to ask the MAC to copy the frame data to a specified buffer described by a TDBufDescr. The protocol can indicate that it is rejecting or does not recognize the frame by returning an appropriate error code. Note that the frame not recognized error has special significance to the Vector function. If the protocol is accepting the frame and if the lookahead buffer contains the whole frame, the protocol can simply copy the data itself before returning from Receive. The protocol may determine that it has the whole frame if BytesAvail equals FrameSize, or if the lookahead information includes a protocol header with the frame length, and this matches BytesAvail.

It is strongly recommended that MACs provide a non-zero FrameSize whenever possible. Some protocols might not be able to process frames unless the frame size given by this parameter is known. A MAC can optionally indicate that it does not normally provide a non-zero frame size by setting bit 16 of the service flags in the MAC specific characteristics table.

The MAC implicitly disables indications (IndicationOff) before calling Receive Lookahead. The Indicate flag byte instructs the MAC on whether to reenable indications or leave them disabled on the return. If the protocol chooses to leave indications disabled, it can enable them within IndicationComplete by calling IndicationOn.

The protocol must absolutely minimize its processing time within the ReceiveLookahead handler. This is necessary to let certain MAC's re-enable the hardware to avoid loss of incoming frames. Shortly after returning from ReceiveLookahead, the MAC will call the protocol back at its IndicationComplete entry point. The protocol can do any needed post-processing of the received frame at that time. The MAC does not guarantee to provide one IndicationComplete call for each indication. It can choose to issue a single IndicationComplete for several indications that have occurred.

TransferData

Purpose: Transfer received frame data from the MAC to a protocol.

```
PUSH LPWORD     BytesCopied   ;Number of bytes copied
PUSH WORD       FrameOffset   ;Starting offset in frame for transfer
PUSH LPBUF      TDBufDescr    ;Virtual address of transfer data description
PUSH WORD       MACDS         ;DS of called MAC module
CALL TransferData
```

Returns: 0x0000 SUCCESS

	0x0007	INVALID_PARAMETER
	0x0008	INVALID_FUNCTION
	0x00FF	GENERAL_FAILURE

BytesCopied — Virtual address of buffer for returning number of bytes copied during transfer data operation.

FrameOffset — Starting offset in received frame where data transfer must start. The value of FrameOffset must be less than or equal to the value of BytesAvail from the corresponding ReceiveLookahead.

TDBufDescr — Virtual address of transfer descriptor describing where to store the frame data.

Description:

A protocol calls this synchronous routine from within its ReceiveLookahead handler before return, to ask the MAC to transfer data for a received frame to protocol storage. The protocol can specify any starting frame offset and byte count for the transfer, so long as these don't exceed the frame's length. If bit 15 of the MAC service flags is set, multiple TransferDatas may be called during a single ReceiveLookahead indication. If this bit is reset, only one TransferData per ReceiveLookahead indication is permitted. In the latter case subsequent calls within the same indication will return an error.

For MACs with bit 15 of the MAC service flags reset, a protocol intending to call TransferData must do so only if it has decided to accept the incoming packet. Since the MAC driver may be shared by multiple protocols, a protocol's failure to follow this restriction in this case jeopardizes other coexisting protocol drivers from receiving these packets. When a protocol is bound to a MAC with bit 15 set, this restriction does not apply as a mandatory requirement. However, it is still recommended in such cases for performance reasons that a protocol call TransferData only if it has decided to accept the incoming packet. A protocol module must set the Lookahead size large enough to determine if the packet is intended for it by examining ony the Lookahead bytes presented by ReceiveLookahead.

It is recommended that the multiple TransferData feature with bit 15 set be implemented in MAC drivers whenever it is reasonable.to do so with the adapter hardware.

IndicationComplete

Purpose: Allow protocol to do post-processing on indications.

```
PUSH  WORD      MACID      ;Module ID of MAC
PUSH  WORD      ProtDS     ;DS of called protocol module
CALL  IndicationComplete
```

Returns:	0x0000	SUCCESS
	0x0007	INVALID_PARAMETER
	0x00FF	GENERAL_FAILURE

Description:

A MAC calls this entry point to enable a protocol to do post-processing after an indication. The MAC will always generate an IndicationComplete subsequent to an indication

regardless of the return code of the indication. Although still in interrupt context and subject to the normal OS/2 guidelines for interrupt processing, the protocol is not under the severe time constraints of the indication. The MAC must minimize stack usage before calling this routine and, under DOS, must have swapped off of any special "interrupt" stack.

This routine is always entered with interrupts enabled and with the network adapter interrupt dismissed from the interrupt controller. Therefore, it may be reentered at the completion of another indication. Also no one-to-one correspondence is guaranteed between indications and IndicationComplete. A MAC may generate one IndicationComplete for several indications. A protocol may enforce a one-to-one correspondence by leaving indications disabled until the return from IndicationComplete.

If indications are explicitly disabled by a protocol on return from an indication, it is the protocol's responsibility to invoke IndicationOn as soon possible during IndicationComplete.

MAC developers must avoid simply serializing each indication with IndicationComplete as this can negatively affect performance. The MAC must be designed to allow an indication to occur during IndicationComplete processing. Of course, if this occurs, another IndicationComplete call will be necessary.

ReceiveChain

Purpose: Indicate reception of a frame in MAC-managed buffers.

```
PUSH WORD     MACID      ;Module ID of MAC
PUSH WORD     FrameSize  ;Total size of frame (bytes)
PUSH WORD     ReqHandle  ;Unique handle for this request
PUSH LPBUF    RxBufDescr ;Virtual address of receive descriptor
PUSH LPBYTE   Indicate   ;Virtual address of indicate flag
PUSH WORD     ProtDS     ;DS of called protocol module
CALL ReceiveChain
```

Returns:		
	0x0000	SUCCESS
	0x0001	WAIT_FOR_RELEASE
	0x0003	FRAME_NOT_RECOGNIZED
	0x0004	FRAME_REJECTED
	0x0005	FORWARD_FRAME
	0x0006	OUT_OF_RESOURCE
	0x0007	INVALID_PARAMETER
	0x00FF	GENERAL_FAILURE

FrameSize — Total size of received frame, in bytes.

RxBufDescr — Virtual address of receive descriptor describing the received frame.

Indicate — Virtual address of indication flag byte. This byte is set to 0xFF by the MAC prior to this call. If the protocol clears the byte to zero prior to returning then indications will be left disabled until IndicationOn is called from IndicationComplete.

Description:

A MAC calls this routine to indicate the reception of a frame in MAC-managed storage. Ownership of this storage is implicitly passed to the protocol when this call is made. At its option, the protocol may copy the data right away and indicate this via the return code (in which case ownership reverts to the MAC); or the protocol may queue the request and copy the frame later, in which case it retains ownership of the frame's storage until it calls ReceiveRelease. Since the protocol may queue data received in this manner, it is possible that the MAC may run low on available frame buffers. The MAC may elect to call ReceiveLookahead instead of ReceiveChain while it is low on frame buffers. This allows the MAC to retain control of its remaining buffers until the protocol releases the buffers it is holding.

Note that for frames longer than 256 bytes, the MAC must guarantee that the first data block of the frame is at least 256 bytes long. Frames less than or equal to 256 bytes in length must be completely specified with a single data block. This allows the protocol to parse packet headers out of the first data block and greatly facilitates protocol processing efficiency.

Like ReceiveLookahead, a protocol's processing within ReceiveChain is time critical. At some point after return from ReceiveChain the MAC will generate an IndicationComplete to allow post-processing of the indication.

The MAC implicitly disables indications (IndicationOff) before calling ReceiveChain. The Indicate flag byte instructs the MAC on whether to reenable indications or leave then disable on the return. If the protocol chooses to leave indications disabled, it can enable them within IndicationComplete by calling IndicationOn.

ReceiveRelease

Purpose: Return frame storage to the MAC that owns it.

```
PUSH WORD       ReqHandle    ;Unique handle from ReceiveChain
PUSH WORD       MACDS        ;DS of called MAC module
CALL ReceiveRelease
```

Returns:	0x0000	SUCCESS
	0x0007	INVALID_PARAMETER
	0x0009	NOT_SUPPORTED
	0x00FF	GENERAL_FAILURE

Description:

A protocol uses this call after it has copied frame data provided by a ReceiveChain call. ReceiveRelease returns ownership of the frame data blocks to the MAC.

IndicationOff

Purpose: Disable MAC indications

```
PUSH WORD       MACDS        ;DS of called MAC module
CALL IndicationOff
```

Returns:	0x0000	SUCCESS
	0x0008	INVALID_FUNCTION
	0x00FF	GENERAL_FAILURE

Description:

A protocol may use this call to prevent the generation of ReceiveLookahead, ReceiveChain and Status indications from the MAC. This is similar in concept to disabling interrupts. When indications are off, a MAC must queue events that would cause it to generate indications to the protocol. A MAC implicitly disables indications just before calling the ReceiveLookahead, ReceiveChain or Status indication entry point of a protocol.

The only legal use of IndicationOff is to bracket a call or calls to the MAC. For example, the following sequence is valid:

IndicationOff
TransmitChain
IndicationOn

In this situation the protocol must not block while indications are off and must call IndicationOn as soon as possible. The protocol must ensure that all calls to IndicationOff are paired up with a corresponding call to IndicationOn. If the protocol issues an IndicationOff call from a timer tick handler, or from a ReceiveLookahead, ReceiveChain or Status indication handler it must issue the IndicationOn call before returning.

Note that IndicationComplete may still occur even though indications are disabled. Disabling indications has no effect on a MAC's ability to call IndicationComplete.

This function always returns with interrupts disabled. It is the responsibility of the caller to re-enable them.

IndicationOn

Purpose: Enable MAC indications

Called from protocol to MAC.

```
PUSH WORD       MACDS       ;DS of called MAC module
CALL IndicationOn
```

Returns:	0x0000	SUCCESS
	0x0008	INVALID_FUNCTION
	0x00FF	GENERAL_FAILURE

Description:

A protocol must use this call to re-enable indications after having disabled them. Note that a MAC may optionally defer the actual re-enabling of indications.

It is possible that IndicationOff and IndicationOn pairs will nest. Therefore the MAC must maintain a reference count to enable it to determine when to actually re-enable indications. The protocol must not assume that a call to IndicationOn will immediately enable indications.

IndicationOn may be called from an IndicationComplete handler after leaving indications disabled on return from an indication handler. IndicationOn may also be used, paired with IndicationOff, to bracket a call or calls to the MAC.

This function always returns with interrupts disabled. It is the responsibility of the caller to re-enable them. No indications will be generated until after the call has returned.

General Requests

General requests are commands from a protocol to a MAC directing it to do adapter management operations like setting the station address, running diagnostics, and changing operating parameters or modes. A MAC may choose to implement any of the Request functions synchronously or asynchronously. A MAC returns the REQUEST_QUEUED return code to inform the protocol that a given request will be processed asynchronously. When this is the case, the MAC will call back to the protocol's RequestConfirm entry point to indicate when processing of the request is complete. If a request handle of zero is used then the RequestConfirm call is suppressed. It is the caller's responsibility to make certain that any data referenced by the request remains valid until the request is guaranteed to have completed. If a protocol makes a general MAC request when executing its InitiateBind startup function and the MAC returns REQUEST_QUEUED, the protocol must wait for the correspondingRequestConfirm to be returned before exiting from the InitiateBind function. Any other return code from a general request implies that no RequestConfirm will occur.

All general requests have the following common calling convention:

```
PUSH  WORD      ProtID      ;Module ID of Protocol or 0
PUSH  WORD      ReqHandle   ;Unique handle for this request or 0
PUSH  WORD      Param1      ;Request dependent word parameter or 0
PUSH  DWORD     Param2      ;Request dependent dword parameter or 0
PUSH  WORD      Opcode      ;Opcode of request
PUSH  WORD      MACDS       ;DS of called MAC module
Call  Request
```

InitiateDiagnostics

Purpose: Start runtime diagnostics.

```
PUSH  WORD      ProtID      ; Module ID of Protocol
PUSH  WORD      ReqHandle   ; Unique handle for this request or 0
PUSH  WORD      0           ; Pad parameter - must be 0
PUSH  DWORD     0           ; Pad parameter - must be 0
PUSH  WORD      1           ; Initiate Diagnostics Request
PUSH  WORD      MACDS       ; DS of called MAC module
Call  Request
```

Returns:		
	0x0000	SUCCESS
	0x0002	REQUEST_QUEUED
	0x0006	OUT_OF_RESOURCE
	0x0007	INVALID_PARAMETER
	0x0008	INVALID_FUNCTION
	0x0009	NOT_SUPPORTED
	0x000A	HARDWARE_ERROR
	0x00FF	GENERAL_FAILURE

Description:

Causes a MAC to run hardware diagnostics and update its status information in the MAC-specific status section of the characteristics table. A MAC must return an error if it does not support run time diagnostics. While the diagnostics are in progress, the MAC must set the diagnostics in progress bit (bit 5) in the MAC status field in the MAC service-specific status table. If HARDWARE_ERROR is returned, the protocol may examine the various fields in the service-specific status table for an indication as to the cause of the problem.

ReadErrorLog

Purpose: Return error log.

```
PUSH WORD     ProtID      ; Module ID of Protocol
PUSH WORD     ReqHandle   ; Unique handle for this request or 0
PUSH WORD     LogLen      ; Length of log buffer
PUSH LPBUF    LogAddr     ; Buffer for returning log
PUSH WORD     2           ; Read Error Log Request
PUSH WORD     MACDS       ; DS of called MAC module
Call Request
```

Returns:		
	0x0000	SUCCESS
	0x0002	REQUEST_QUEUED
	0x0006	OUT_OF_RESOURCE
	0x0007	INVALID_PARAMETER
	0x0008	INVALID_FUNCTION
	0x0009	NOT_SUPPORTED
	0x00FF	GENERAL_FAILURE

Description:

Causes a read error log to be issued to adapter. This command is implemented on the IBM token ring adapter and possibly other adapters. The format of the information returned is adapter specific and not specified here.

SetStationAddress

Purpose: Set the network address of the station.

```
PUSH WORD     ProtID      ; Module ID of Protocol
PUSH WORD     ReqHandle   ; Unique handle for this request or 0
PUSH WORD     0           ; Pad parameter - must be 0
PUSH LPBUF    AdaptAddr   ; Buffer containing the adapter address
PUSH WORD     3           ; SetStationAddress Request
PUSH WORD     MACDS       ; DS of called MAC module
Call Request
```

Returns:		
	0x0000	SUCCESS
	0x0002	REQUEST_QUEUED
	0x0006	OUT_OF_RESOURCE
	0x0007	INVALID_PARAMETER
	0x0008	INVALID_FUNCTION
	0x0009	NOT_SUPPORTED
	0x00FF	GENERAL_FAILURE

Description:

There is only a single station address. Each time it replaces the current station address in the MAC service-specific characteristics table and will reconfigure the hardware to receive on that address if required. The station will be initially configured with the address specified in the permanent station address field of the MAC service-specific characteristics table (which this call does not modify).

The adapter address buffer contains only the bytes of the address to be set. The length of the address must be equal to the length specified in the MAC service characteristics table.

If the hardware does not support a mechanism to modify its station address then the current station address buffer is not updated and this function returns INVALID_FUNCTION. In this case the MAC continues to use the permanent station address to recognize incoming directed packets.

If a MAC does not support the OpenAdapter and CloseAdapter commands (bit 11 of the MAC service flags is reset), then the SetStationAddress command can be issued by the protocol at any time. However, if the MAC supports the Open Adapter and CloseAdapter commands (bit 11 of the MAC service flags is set), then this command is valid only either during system initialization time or while the MAC is in a closed state. The protocol driver must issue an Open Adapter call after issuing the SetStationAddress call for the SetStationAddress command to take effect.

OpenAdapter

Purpose: Issue open request to network adapter.

```
PUSH  WORD      ProtID        ; Module ID of Protocol
PUSH  WORD      ReqHandle     ; Unique handle for this request or 0
PUSH  WORD      OpenOptions   ; Adapter specific open options
PUSH  DWORD     ExtendedRet   ; Optional pointer to a DWORD extended return code
                              (vendor-specific or warning level)
PUSH  WORD      4             ; Open Adapter Request
PUSH  WORD      MACDS         ; DS of called MAC module
Call  Request
```

Returns:		
	0x0000	SUCCESS
	0x0002	REQUEST_QUEUED
	0x0006	OUT_OF_RESOURCE
	0x0007	INVALID_PARAMETER
	0x0008	INVALID_FUNCTION
	0x0009	NOT_SUPPORTED
	0x0024	HARDWARE_FAILURE
	0x002A	NETWORK_MAY_NOT_BE_CONNECTED
	0x00FF	GENERAL_FAILURE

Where:

Optional vendor-specific information can be returned through the ExtendedRet pointer. A caller supporting this would push a pointer to a DWORD. The DWORD would have been initialized to 0xFFFFFFFF (unsupported). If there is any extended return information this value would be changed. A caller not supporting this would simply push a NULL (0) pointer. The OpenAdapter routine which supports this would verify the ExtendedRet

pointer is not NULL (0) and then write the information. The OpenAdapter routine which does not support this would simply ignore the pointer.

The purpose of ExtendedRet is to provide ***warning*** messages on a SUCCESS return without requiring additional testing for those callers not supporting warnings, to provide additional information on GENERAL_FAILURE and HARDWARE_FAILURE, and to pass vendor-specific codes on any return to provide for active functional experimentation and evolution without inconveniencing other vendor's components.

Description:

The purpose of the OpenAdapter function is to activate an adapter's network connection. This may involve making an electrical connection for some adapters like token ring adapters. This also implies that a considerable delay may occur between submittal of this request and its confirmation. If the MAC indicates that OpenAdapter is supported (by setting bit 11 of the service flags in the MAC service-specific characteristics table), then the protocol driver must ensure the adapter is open during bind-time processing. Since OpenAdapter can only be called when the adapter is closed, even in a VECTOR configuration, the protocol must first check if the adapter is already open by examining bit 4 of the MAC status in the MAC service-specific status table.

While an adapter is closed the following functions are guaranteed to operate: SetLookahead, SetPacketFilter, SetStationAddress, Interrupt, Indicationoff, IndicationOn.

Since this function is adapter specific it is expected that any necessary parameters are either known a priori by the MAC or can be recovered from the PROTOCOL.INI file. The format of the information is highly adapter specific and left up to the implementer to define.

The OpenOptions parameter is adapter specific. For IBM TokenRing and compatible adapters, these are defined in the IBM Token Ring Technical Reference Manual.

CloseAdapter

Purpose: Issue close request to network adapter.

```
PUSH WORD       ProtID      ; Module ID of Protocol
PUSH WORD       ReqHandle   ; Unique handle for this request or 0
PUSH WORD       0           ; Pad parameter - must be 0
PUSH DWORD      0           ; Pad parameter - must be 0
PUSH WORD       5           ; Close Adapter Request
PUSH WORD       MACDS       ; DS of called MAC module
Call Request
```

Returns:		
	0x0000	SUCCESS
	0x0002	REQUEST_QUEUED
	0x0006	OUT_OF_RESOURCE
	0x0007	INVALID_PARAMETER
	0x0008	INVALID_FUNCTION
	0x0009	NOT_SUPPORTED
	0x00FF	GENERAL_FAILURE

Description:

This function closes an adapter. This causes it to decouple itself from a network so that packets cannot be sent or received. CloseAdapter resets the functional or multicast addresses currently set.

Since this function is adapter specific it is expected that any necessary parameters are either already known by the MAC or can be recovered from the PROTOCOL.INI file. The format of the information is highly adapter specific and left up to the implementer to define.

ResetMAC

Purpose: Reset the MAC software and adapter hardware.

```
PUSH  WORD      ProtID      ; Module ID of Protocol
PUSH  WORD      ReqHandle   ; Unique handle for this request or 0
PUSH  WORD      0           ; Pad parameter - must be 0
PUSH  DWORD     0           ; Pad parameter - must be 0
PUSH  WORD      6           ; Reset MAC Request
PUSH  WORD      MACDS       ; DS of called MAC module
Call  Request
```

Returns:		
	0x0000	SUCCESS
	0x0006	OUT_OF_RESOURCE
	0x0007	INVALID_PARAMETER
	0x0008	INVALID_FUNCTION
	0x0009	NOT_SUPPORTED
	0x0024	HARDWARE_FAILURE
	0x002A	NETWORK_MAY_NOT_BE_CONNECTED
	0x00FF	GENERAL_FAILURE

Description:

The function causes the MAC to issue a hardware reset to the network adapter. The MAC may discard without confirmation any pending requests and abort operations in progress. For compatibility with some current protocols which do not properly handle resets, it is suggested the MAC complete pending requests, returning INVALID_FUNCTION on all confirmations which result. The MAC must preserve the current station address, LOOKAHEAD length, packet filter, multicast address list, functional address and indication on/off state.

For MAC's that support the OpenAdapter function, the Reset MAC command leaves the adapter in the opened state if it was opened prior to the reset. The adapter open parameters that were in effect prior to the reset must be the same ones in effect after the reset.

When the reset is initiated, the MAC must generate a StartReset status indication back to the protocol. For some MAC's a considerable delay can elapse between the start of the reset and its completion. All MAC's must subsequently issue an EndReset indication when the reset is complete. During the time between the StartReset indication and the corresponding EndReset indication, the MAC must return INVALID_FUNCTION for any request it receives while a reset is in progress. The EndReset indication notifies the protocol that the MAC can handle new requests. As always, an IndicationComplete follows these indications. MACs written to V1.0.1. of this spec will not issue the End Reset. They must issue the IndicationComplete to signal the end of the reset.

Note that the completion (i.e. the return from this command or the request confirm) of the Reset MAC request itself does not signal the start or end of the reset.

There can be no guarantee that this function will succeed, though the NDIS MAC developer must make every attempt. An error return from this call can be considered fatal. If the reset fails, the adapter may no longer be in the same state. For example, if the adapter was open before a failed ResetMAC, it may now be closed.

ResetMac must not be queued.

SetPacketFilter

Purpose: Select received packet general filtering parameters.

```
PUSH WORD       ProtID      ; Module ID of Protocol
PUSH WORD       ReqHandle   ; Unique handle for this request or 0
PUSH WORD       FilterMask  ; Bit mask for packet filter
PUSH DWORD      0           ; Pad parameter - must be 0
PUSH WORD       7           ; Set Packet Filter Request
PUSH WORD       MACDS       ; DS of called MAC module
Call Request
```

FilterMask bit
0 directed and multicast or group and functional
1 broadcast packets
2 any packet on LAN (promiscuous)
3 any source routing packet on LAN
4-15 Reserved, must be zero

Returns:		
	0x0000	SUCCESS
	0x0002	REQUEST_QUEUED
	0x0006	OUT_OF_RESOURCE
	0x0007	INVALID_PARAMETER
	0x0008	INVALID_FUNCTION
	0x00FF	GENERAL_FAILURE

Description:

This command tells the MAC which kinds of received packets must generate indications to the protocol invoking this command. A FilterMask of 0 indicates that the MAC must not indicate received packets to that protocol. If a FilterMask bit is set, then this indicates that the MAC must indicate that type of packet to the protocol. Except for a 0 FilterMask, a filter bit of 0 does not require the MAC to suppress indications for that type of packet. For example the FilterMask used by the MAC may or may not correspond to the capabilities of the hardware adapter. For example a MAC may be designed to receive multicast frames by promiscuously receiving all frames and discarding those that do not match the filter. It is optional for the MAC to support such software filtering. If the MAC can suppress such indications, it is strongly recommended that it do so. However, if the MAC does not suppress such indications, then the protocol must be prepared to receive these and discard the incoming packet if necessary.

If this request returns SUCCESS, then the hardware is enabled to receive the types of packets requested and will generate Indications to the protocol for those types of packets.

If the MAC does not support the receiving of packets of the type specified, then it will return GENERAL_FAILURE. In this case the FilterMask is left in its previous state.

AddMulticastAddress

Purpose: Allow adapter to respond to a multicast address.

```
PUSH  WORD      ProtID      ; Module ID of Protocol
PUSH  WORD      ReqHandle   ; Unique handle for this request or 0
PUSH  WORD      0           ; Pad parameter - must be 0
PUSH  LPBUF     MultiAddr   ; Buffer containing multicast address
PUSH  WORD      8           ; Add Multicast Address Request
PUSH  WORD      MACDS       ; DS of called MAC module
Call  Request
```

Returns:		
	0x0000	SUCCESS
	0x0002	REQUEST_QUEUED
	0x0006	OUT_OF_RESOURCE
	0x0007	INVALID_PARAMETER
	0x0008	INVALID_FUNCTION
	0x0009	NOT_SUPPORTED
	0x00FF	GENERAL_FAILURE

Description:

This function allows the addition of multicast addresses. The term multicast address also implies 802.5 group addresses. This function allows the addition of only one address at a time but can be repeated to add more multicasts.

It is the MAC's responsibility to return an error if too many multicast addresses have been added (OUT_OF_RESOURCE or INVALID_FUNCTION) or if an address of the wrong type has been added (INVALID_PARAMETER).

Multicast addresses are never over written and will return an error (INVALID_PARAMETER) if they already exist no matter what their type. They must be explicitly deleted.

The multicast address buffer contains only the bytes of the multicast address to be added. The length of the multicast address must be equal to the length specified in the MAC service characteristics table.

DeleteMulticastAddress

Purpose: Forbid adapter to respond to a multicast address.

```
PUSH WORD       ProtID      ; Module ID of Protocol
PUSH WORD       ReqHandle   ; Unique handle for this request or 0
PUSH WORD       0           ; Pad parameter - must be 0
PUSH LPBUF      MultiAddr   ; Buffer containing multicast address
PUSH WORD       9           ; Delete Multicast Address Request
PUSH WORD       MACDS       ; DS of called MAC module
Call Request
```

Returns:	0x0000	SUCCESS
	0x0002	REQUEST_QUEUED
	0x0006	OUT_OF_RESOURCE
	0x0007	INVALID_PARAMETER
	0x0008	INVALID_FUNCTION
	0x0009	NOT_SUPPORTED
	0x00FF	GENERAL_FAILURE

Description:

This function removes a previously added multicast address. The term multicast address also implies 802.5 group addresses. INVALID_PARAMETER is returned if the address was not in the table.

The multicast address buffer has the same format as in the AddMulticastAddress command.

UpdateStatistics

Purpose: Cause MAC statistics to be updated.

```
PUSH WORD       ProtID      ; Module ID of Protocol
PUSH WORD       ReqHandle   ; Unique handle for this request or 0
PUSH WORD       0           ; Pad parameter - must be 0
PUSH DWORD      0           ; Pad parameter - must be 0
PUSH WORD       10          ; Update Statistics request
PUSH WORD       MACDS       ; DS of called MAC module
Call Request
```

Returns:	0x0000	SUCCESS
	0x0002	REQUEST_QUEUED
	0x0006	OUT_OF_RESOURCE
	0x0007	INVALID_PARAMETER
	0x0008	INVALID_FUNCTION
	0x00FF	GENERAL_FAILURE

Description:

Causes the MAC to atomically update the statistics in its characteristics table. The requester can then read the table when this operation is complete. Those statistics which are not always current will remain the same until the next UpdateStatistics call is performed. If all of the statistics in the table are always current this function must return SUCCESS.

ClearStatistics

Purpose: Cause MAC statistics to be cleared.

```
PUSH WORD       ProtID      ; Module ID of Protocol
PUSH WORD       ReqHandle   ; Unique handle for this request or 0
PUSH WORD       0           ; Pad parameter - must be 0
PUSH DWORD      0           ; Pad parameter - must be 0
PUSH WORD       11          ; Clear Statistics request
PUSH WORD       MACDS       ; DS of called MAC module
Call Request
```

Returns:	0x0000	SUCCESS
	0x0002	REQUEST_QUEUED
	0x0006	OUT_OF_RESOURCE
	0x0007	INVALID_PARAMETER
	0x0008	INVALID_FUNCTION
	0x00FF	GENERAL_FAILURE

Description:

Causes the MAC to reset its statistics counters. This implies that all statistics must be reset to zero in an atomic operation.

InterruptRequest

Purpose: Request asynchronous indication.

```
PUSH WORD       ProtID      ; Module ID of Protocol
PUSH WORD       0           ; Pad parameter - must be 0
PUSH WORD       0           ; Pad parameter - must be 0
PUSH DWORD      0           ; Pad parameter - must be 0
PUSH WORD       12          ; InterruptRequest
PUSH WORD       MACDS       ; DS of called MAC module
Call Request
```

Returns:	0x0000	SUCCESS
	0x0006	OUT_OF_RESOURCE
	0x0008	INVALID_FUNCTION
	0x0009	NOT_SUPPORTED
	0x00FF	GENERAL_FAILURE

Description:

This function requests the MAC to generate an asynchronous Interrupt Status indication back to the protocol. The protocol may control the generation of this Interrupt Status indication by disabling and later enabling indications. The MAC may at its discretion suppress the generation of this indication if there is another indication pending which may be issued in place of the Interrupt status indication. This request is intended to be used for MAC's which can generate a hardware interrupt on demand. This function must be implemented if at all possible. Interrupt request will substantially improve the performance of some protocols (particularly DLC).

SetFunctionalAddress

Purpose: Cause adapter to change its functional address.

```
PUSH WORD       ProtID      ; Module ID of Protocol
PUSH WORD       ReqHandle   ; Unique handle for this request or 0
PUSH WORD       0           ; Pad parameter - must be 0
PUSH LPBUF      FunctAddr   ; Buffer containing functional address
PUSH WORD       13          ; Set Functional Address Request
PUSH WORD       MACDS       ; DS of called MAC module
Call Request
```

Returns:	0x0000	SUCCESS
	0x0002	REQUEST_QUEUED
	0x0006	OUT_OF_RESOURCE
	0x0007	INVALID_PARAMETER
	0x0008	INVALID_FUNCTION
	0x0009	NOT_SUPPORTED
	0x00FF	GENERAL_FAILURE

Description:

This sets the IEEE802.5 functional address to the passed functional address. The adapter will use the functional address to discern packets intended for it. For more information on functional addresses see the IEEE 802.5 specification.

The functional address buffer contains only the bytes of the new functional address bit pattern. It represents the logical OR of all functional addresses to be registered with the adapter. The length of the functional address buffer is 4 bytes.

Multiple protocols can set or reset their functional address bit if required by each protocol by first reading the current functional address DWORD bit pattern from the MAC service characteristics table, then ORing in or ANDing out the required functional bit and passing the new functional address pattern in this command.

SetLookahead

Purpose: Set length of lookahead information for ReceiveLookahead.

```
PUSH WORD       ProtID      ; Module ID of Protocol
PUSH WORD       ReqHandle   ; Unique handle for this request or 0
PUSH WORD       Length      ; Minimum length of lookahead info
PUSH DWORD      0           ; Pad parameter - must be 0
PUSH WORD       14          ; Set Lookahead Request
PUSH WORD       MACDS       ; DS of called MAC module
Call Request
```

Returns:	0x0000	SUCCESS
	0x0002	REQUEST_QUEUED
	0x0007	INVALID_PARAMETER
	0x00FF	GENERAL_FAILURE

Description:

This request sets the minimum length in bytes of lookahead information to be returned in a Receive Lookahead indication. Until SetLookahead is initially called, a value of 64 bytes is assumed for the lookahead length. When first called, SetLookahead sets the lookahead length value equal to the Length parameter of the request. After the first SetLookahead request, the lookahead length is changed only if the value of the Length parameter is larger than the current lookahead length. If the length parameter value is smaller, the current Lookahead length remains unchanged and SUCCESS is returned. SetLookahead may be called at any time and the lookahead length is preserved during a reset. The maximum value for the lookahead length is 256 bytes. MAC's which never call Receive Lookahead or always return lookahead information of length greater than or equal to 256 bytes may return SUCCESS without any internal action. MAC's must support 256 bytes of lookahead data if requested.

General Request Confirmation

Purpose: Confirm completion of a previous General Request.

```
PUSH  WORD      ProtID      ; Module ID of Protocol
PUSH  WORD      MACID       ; Module ID of MAC
PUSH  WORD      ReqHandle   ; Unique handle of original request
PUSH  WORD      Status      ; Final status of original request
PUSH  WORD      Request     ; Original Request opcode
PUSH  WORD      ProtDS      ; DS of called Protocol module
Call  RequestConfirm
```

Returns:	0x0000	SUCCESS
	0x0006	OUT_OF_RESOURCE
	0x0007	INVALID_PARAMETER
	0X0024	HARDWARE_FAILURE
	0x00FF	GENERAL_FAILURE

Description:

Notify a protocol that an asynchronous MAC control Request has completed after previous Request had returned a REQUEST_QUEUED. It is possible that a RequestConfirm can be returned to the protocol before the protocol's corresponding Request function has completed.

The ProtID parameter must be the value passed by the protocol on the previous general request to identify the requestor.

If a protocol had made a general MAC request when executing its InitiateBind startup function and the MAC returned REQUEST_QUEUED, the protocol must wait for the corresponding RequestConfirm to be returned before exiting from the InitiateBind function.

Status Indications

Status indications are spontaneous calls from a MAC to a protocol, typically at interrupt time. They inform the protocol of changes in MAC status.

All status indications have the following common calling convention:

```
PUSH WORD       MACID       ; Module ID of MAC
PUSH WORD       Param1      ; Opcode dependent word parameter or 0
PUSH LPBYTE     Indicate    ; Virtual address of indicate flag
PUSH WORD       Opcode      ; Opcode of status indication
PUSH WORD       ProtDS      ; DS of called Protocol module
Call Status
```

Indicate is the virtual address of the indication flag byte. This byte is set to 0xFF by the MAC prior to this call. If the protocol clears the byte to zero prior to returning then indications will be left disabled until IndicationOn is called from IndicationComplete.

RingStatus

Purpose: Return a change in ring status.

```
PUSH WORD       MACID       ; Module ID of MAC
PUSH WORD       Status      ; New Ring Status
PUSH LPBYTE     Indicate    ; Virtual address of indicate flag
PUSH WORD       1           ; Ring Status Indication
PUSH WORD       ProtDS      ; DS of called protocol module
Call Status
```

Returns: 0x0000 SUCCESS

Description:

Called by 802.5-style MAC drivers to indicate a change in ring status. The status codes for 802.5-style drivers are encoded as a 16-bit mask, where the bits in the mask are defined as follows:

Bit	Meaning
15	Signal Loss
14	Hard Error
13	Soft Error
12	Transmit Beacon
11	Lobe Wire Fault
10	Auto-Removal Error 1
9	Reserved
8	Remove Received
7	Counter Overflow
6	Single Station
5	Ring Recovery
4-0	Reserved

For certain ring status changes, the adapter may already have been removed from the ring. The protocol driver must check whether the adapter has been closed (by examining bit 4 fo the MAC status field in the MAC service-specific status table). For additional information, consult the *IBM Token Ring Technical Reference Manual.* If the status condition caused the adapter to close, the MAC must return confirmations with non-SUCCESS status codes for all pending TransmitChain and general requests.

AdapterCheck

Purpose: Return hardware status.

```
PUSH  WORD      MACID      ; Module ID of MAC
PUSH  WORD      Reason     ; Reason for Adapter Check
PUSH  LPBYTE    Indicate   ; Virtual address of indicate flag
PUSH  WORD      2          ; Adapter Check Indication
PUSH  WORD      ProtDS     ; DS of called protocol module
Call  Status
```

Returns: 0x0000 SUCCESS

Description:

Called to indicate a fatal adapter error. If this function is called the protocol must issue a ResetMAC call (if supported) before communications can resume. Note that a MAC may choose to tolerate some number of errors before issuing an AdaperCheck indication. For example, a MAC may want to accept the occasional receive DMA overrun, and only issue the AdapterCheck for this condition if it occurs excessively.

For 802.5 MAC's the Reason code is defined as follows (NOT a bit mask):

0x8000	Adapter Inoperative
0x1000	Illegal Opcode
0x0800	Local Bus Parity Error
0x0400	Parity Error
0x0100	Internal Parity Error
0x0080	Parity Error, Ring Transmit
0x0040	Parity Error, Ring Receive
0x0020	Transmit Overrun
0x0010	Receive Overrun
0x0008	Unrecognized Interrupt
0x0004	Unrecognized Error Interrupt
0x0003	Adapter Detected No PC System Service
0x0002	Unrecognized Supervisory Request
0x0001	Program Request

All 802.5 values not defined above are reserved.

The MAC must always return confirmations with non-SUCCESS status codes for all pending TransmitChain and general requests.

For 802.3 MAC's the Reason code is defined as follows (NOT a bit mask):

0x8000	Adapter Inoperative (Adapter did not respond to command or could not be found)
0x4000	Command Timed Out (Adapter did not complete command within acceptable time interval)
0x2000	SQE Test Failure (No heartbeat detected on previous transmission)
0x1000	Excessive Collisions (Transmission failed due to excessive collisions)
0x0800	Lost Carrier Sense (Adapter lost carrier during transmission)
0x0400	TDR Failure (TDR test detected a short or open on the link)
0x0020	Transmit Underrun (DMA underrun occurred on transmission)
0x0010	Receive Overrun (DMA overrun occurred on reception)

All 802.3 values not defined above are reserved.

StartReset

Purpose: Imply that adapter has started a reset.

```
PUSH  WORD      MACID       ; Module ID of MAC
PUSH  WORD      0           ; Pad parameter must be zero
PUSH  LPBYTE    Indicate    ; Virtual address of indicate flag
PUSH  WORD      3           ; Start Reset Indication
PUSH  WORD      ProtDS      ; DS of called protocol module
Call  Status
```

Returns: 0x0000 SUCCESS

Description:

Called to indicate that the adapter has started a reset. This will generally be due to a call to ResetMAC (perhaps by another protocol driver in a VECTOR configuration) but can be unsolicited. The protocol must assume when it gets this indication that all requests outstanding to the MAC have been discarded without notification. The end of the reset will be signalled by an EndReset indication. The reset process may take a significant amount of time. While it is in progress, the MAC may reject any requests it cannot handle with INVALID_FUNCTION (0x0008). As with any other indication, StartReset is entered with indications implicitly disabled. To protect itself from other indications the protocol may choose to modify the Indicate flag to keep indications disabled on return. This will not prevent the EndReset indication from being generated however.

StartReset is affected by IndicationOn and IndicationOff.

EndReset

Purpose: Imply that adapter has finished a reset.

```
PUSH  WORD      MACID      ; Module ID of MAC
PUSH  WORD      Status     ; MAC error information
PUSH  LPBYTE    Indicate   ; Virtual address of indicate flag
PUSH  WORD      5          ; End Reset Indication
PUSH  WORD      ProtDS     ; DS of called protocol module
Call  Status
```

Returns:	0x0000	SUCCESS
	0x0008	INVALID_FUNCTION

Description:

Called to indicate that the adapter has finished a reset and follows the StartReset indication. The protocol may return INVALID_FUNCTION if it was written to the 1.0.1 version of this specification, where it assumes end of reset on IndicationComplete. To ensure compatibility with 1.0.1 protocol drivers, the MAC must ensure the IndicationComplete is called after EndReset and before any other indications.

EndReset will pass up a success/fail code for ResetMAC in the Status parameter.

0x0000	SUCCESS
0x0024	HARDWARE_ERROR
0x002A	NETWORK_MAY_NOT_BE_CONNECTED
0x00FF	GENERAL_FAILURE

As with any other indication, EndReset is entered with indications implicitly disabled. To protect itself from other indications the protocol may choose to modify the Indicate flag to keep indications disabled on return. MAC drivers must be prepared for the possibility that both StartReset and EndReset allow the protocol to modify this flag.

EndReset is **not** affected by IndicationOn and IndicationOff. In other words, if the protocol modifies the indicate flag during StartReset to disable indications, this will not prevent the EndReset indication from being generated.

If both StartReset and EndReset disable indications, the IndicationOff depth is 2, requiring two calls to IndicationOn in order to enable indications. For example, if protocol A disables indications during StartReset and protocol B disables indications during EndReset, both protocols must issue IndicationOn before indications are re-enabled. The same is true if the same protocol issues IndicationOff twice.

Interrupt

Purpose: Imply that an interrupt has occurred as the result of a interrupt request.

```
PUSH  WORD      MACID      ; Module ID of MAC
PUSH  WORD      0          ; Pad parameter must be 0
PUSH  LPBYTE    Indicate   ; Virtual address of indicate flag
PUSH  WORD      4          ; Interrupt indication
PUSH  WORD      ProtDS     ; DS of called protocol module
Call  Indication
```

Returns: 0x0000 SUCCESS

Description:

The MAC calls this function to indicate to a protocol that an interrupt requested by an Interrupt request has occurred. Since this indication may be deferred by disabling indications, a protocol may use this mechanism to implement a simple scheduling scheme to allow it to regain control once outside of a critical code region. The MAC may at its discretion suppress the generation of this indication if there is another indication pending which may be issued in place of the Interrupt status indication.

System Requests

All MAC and protocol modules implement a set of system request functions that support module-independent functions such as binding. The caller of these functions is usually the Protocol Manager. The entry point for system requests is defined in the common characteristics table for the module. All system requests are implemented synchronously. Note that all pointers in system requests are Ring 0 GDT virtual addresses.

All system requests have the following common calling convention:

```
PUSH DWORD      Param1      ; Request dependent dword parameter or 0
PUSH DWORD      Param2      ; Request dependent dword parameter or 0
PUSH WORD       Param3      ; Request dependent word parameter or 0
PUSH WORD       Opcode      ; Opcode of request
PUSH WORD       TargetDS    ; DS of called module
Call System
```

InitiateBind

Purpose: Instruct a module to bind to another module.

```
PUSH DWORD      0           ; Pad parameter must be 0
PUSH LPBUF      CharTab     ; Characteristics of module to bind
PUSH WORD       LastBind    ; Non-zero if last InitiateBind
PUSH WORD       1           ; Initiate Bind Request
PUSH WORD       ProtDS      ; DS of called Protocol module
CALL System
```

Returns:		
	0x0000	SUCCESS
	0x0008	INVALID_FUNCTION
	0x0021	INCOMPLETE_BINDING
	0x0022	DRIVER_NOT_INITIALIZED
	0x0023	HARDWARE_NOT_FOUND
	0x0024	HARDWARE_FAILURE
	0x0025	CONFIGURATION_FAILURE
	0x0026	INTERRUPT_CONFLICT
	0x0027	INCOMPATIBLE_MAC
	0x0028	INITIALIZATION_FAILED
	0x002A	NETWORK_MAY_NOT_BE_CONNECTED
	0x002B	INCOMPATIBLE_OS_VERSION
	0x00FF	GENERAL_FAILURE

Description:

This call is issued by the Protocol Manager to an upper protocol module. It passes the address of the characteristics table of the lower module that the upper module must issue a Bind call to. If the upper module specified a BindingsList including more than one lower module, then InitiateBind's will be issued for those modules in the order the lower modules are listed in the BindingsList structure. LastBind is used to indicate the last Initiate Bind request so the module may perform any final initialization prior to returning. In the static default binding case of one static protocol and one MAC, the Protocol Manager will issue an InitiateBind passing the characteristics table of the MAC even if no bindings list was specified. In this case LastBind will be non-zero. In the non-default case if a module other than a MAC does not have lower bindings (having a Bindlist with a NumBindings count = 0), the Protocol Manager will still issue an Initiate Bind to the module to allow final initialization. In this case CharTab will be NULL and LastBind will be non-zero.

If the Bind operation fails then the Initiate Bind operation must also fail returning the same return code as the failing Bind call.

If a module returns a non-SUCCESS code on InitiateBind, in the dynamic mode the Protocol Manager will automatically deregister that module and remove all reference to it in its bind tables. In particular any other module that had registered (via RegisterModule) its intention to bind with the failed module will get an InitiateBind call with the "CharTab" pointer far NULL and "LastBind" non-zero. A module that has lower bindings and receives an InitiateBind with a NULL bind "CharTab" must generate a non-SUCCESS return code in order to force the Protocol Manager to deregister it. In DOS it is recommended that a dynamic module that failed its bind deinstall itself. In OS/2 it is recommended that the dynamic driver that failed its bind leave its dynamic segments unlocked.

Bind

Purpose: Exchange module characteristic table information.

```
PUSH  LPBUF     CharTab     ; Pointer to caller's table
PUSH  LPBUF     TabAddr     ; Address where to return a pointer
                            ; to called module's characteristics
PUSH  WORD      0           ; Pad parameter must be zero
PUSH  WORD      2           ; Bind Request
PUSH  WORD      TargetDS    ; DS of called module
CALL  System
```

Returns:		
	0x0000	SUCCESS
	0x0008	INVALID_FUNCTION
	0x0022	DRIVER_NOT_INITIALIZED
	0x0023	HARDWARE_NOT_FOUND
	0x0024	HARDWARE_FAILURE
	0x0025	CONFIGURATION_FAILURE
	0x0026	INTERRUPT_CONFLICT
	0x0027	INCOMPATIBLE_MAC
	0x0028	INITIALIZATION_FAILED
	0x002A	NETWORK_MAY_NOT_BE_CONNECTED
	0x002B	INCOMPATIBLE_OS_VERSION
	0x00FF	GENERAL_FAILURE

Description:

Used by one module to bind to another. It exchanges pointers to characteristics tables between the two modules. A MAC will accept only one bind and will not accept additional bind attempts.

For compatibility with Remote Initial Program Load, MAC drivers must not manipulate the network adapter at INIT time. The MAC driver is free to determine if the network adapter is present, but must leave any hardware manipulation to Bind time processing.

InitiatePrebind (OS/2 only)

Purpose: In OS/2 dynamic bind mode instruct a module to restart its prebind initialization.

```
PUSH DWORD      0               ; Pad parameter (must be zero)
PUSH LPBUF      0               ; Pad parameter (must be zero)
PUSH WORD       0               ; Pad parameter (must be zero
PUSH WORD       3               ; Initiate Prebind Request
PUSH WORD       ProtDS          ; DS of called protocol module
CALL System
```

Returns:	0x0000	SUCCESS
	0x00FF	GENERAL_FAILURE

Description:

In the OS/2 dynamic mode, this call is issued by the Protocol Manager to a dynamically loadable protocol driver when the Protocol Manager InitAndRegister is called. This function is available for the protocol driver to restart its prebind initialization when it is dynamically reloaded.

An OS/2 dynamic protocol driver is assumed to be made up of static and transient segments. When the protocol is not needed, the transient segments are unlocked (using the DevHlp Unlock command) to allow them to be swapped out. When the protocol is needed again, InitiatePrebind is issued. During InitiatePrebind, the driver needs to Lock down its dynamic segments (using the DevHlp Lock command, type 1) to force them back into memory and make them addressable again. The protocol must save the lock handle returned by this call for later Unlock'ing. Also, the prebind initialization sequence is initiated in this call and consists of re-reading the PROTOCOL.INI memory image, configuration initialization, prebind memory allocations, and registration with the Protocol Manager. The protocol module typically carries out here the same functions that are performed by a static protocol module when a strategy routine INIT command is given.

InitiateUnbind

Purpose: Instruct a module to unbind from another module.

```
PUSH DWORD      0               ; Pad parameter (must be zero)
PUSH LPBUF      CharTab         ; Char's of module to unbind
PUSH WORD       LastUnbind      ; Non-zero if last Init'Unbind
PUSH WORD       4               ; Initiate Unbind Request
PUSH WORD       ProtDS          ; DS of called protocol module
CALL System
```

Returns:

	0x0000	SUCCESS
	0x00FF	GENERAL_FAILURE

Description:

This call is issued by the Protocol Manager in dynamic mode to an upper protocol module. It passes the address of the characteristics table of the lower module that the upper module must issue an Unbind command to (this would be an entry into the VECTOR if the lower module is a MAC). LastUnbind is used to indicate the last InitiateUnbind request, so the module may perform any final cleanup before returning.

If a protocol module does not have lower bindings (having a BindingsList with a NumBindings count = 0), InitiateUnbind will still be issued with CharTab set to NULL and LastUnbind set to non-zero in order to allow the module to terminate.

Unbind

Purpose: An unbind request from an upper protocol module to a lower module.

```
PUSH  LPBUF     CharTab         ; Caller's characteristics table
PUSH  DWORD     0               ; Pad parameter (must be zero)
PUSH  WORD      0               ; Pad parameter (must be zero)
PUSH  WORD      5               ; Unbind Request
PUSH  WORD      TargetDS        ; DS of called module
CALL  System
```

Returns:

	0x0000	SUCCESS
	0x0008	INVALID_FUNCTION
	0x00FF	GENERAL_FAILURE

Description:

Used by one protocol module to unbind from another. The caller's characteristics table is passed to permit the called module to identify the upper module. If the Unbind is to a MAC, the VECTOR does the Unbind cleanup on behalf of the MAC. Thus MAC drivers themselves do not need to support this call.

Protocol Manager Primitives

Since the Protocol Manager primitives may be accessed via an IOCTL in OS/2, a request block is defined as follows:

```
struct ReqBlock
{
  unsigned Opcode;   /*Opcode for Protocol Manager request      */
  unsigned Status;   /*Status at completion of request         */
  char far *Pointer1; /*First parameter Ring 0 GDT pointer      */
  char far *Pointer2; /*Second parameter Ring 0 GDT pointer     */
  unsigned Word1;    /*Parameter word                          */
};
```

Direct calls are made to the Protocol Manager with a pointer to the ReqBlock on the stack. For IOCTL requests, the parameter buffer contains a pointer to the ReqBlock. The direct calling sequence is as follows:

```
PUSH  LPBUF       ReqBlock    ; Ring 0 GDT Address of ReqBlock
PUSH  WORD        TargetDS    ; DS of Protocol Manager
Call  ProtManEntry
```

Note that under OS/2 the direct entry cannot be used at CONFIG.SYS initialization time since the driver is still in Ring 3 context.

Note also that if the Protocol Manager is in dynamic mode, these primitives can be invoked by other modules after system initialization. Dynamic OS/2 Ring 0 device drivers issuing these primitives post INIT time must use the direct entry interface since the IOCTL interface is illegal at this time.

GetProtocolManagerInfo

Purpose: Retrieve pointer to configuration image.

Opcode - 1

Status - On return contains request status

Pointer1 - On return contains a FAR pointer to structure memory image representing the parsed user configuration file PROTOCOL.INI. For static OS/2 device drivers, the selector of the pointer returned here is valid only at device INIT time. For dynamic OS/2 device drivers, the selector returned is always valid and will be a valid LDT selector for the process under which this primitive is called. For DOS this is a segment:offset pair.

Pointer2 - Unused

Word1 - On return contains the BCD-encoded major (low byte in memory) and minor (high byte in memory) version of the specification on which this Protocol Manager driver is based. (2.0 for this specification)

Returns:		
	0x0000	SUCCESS
	0X0008	INVALID_FUNCTION
	0x0002F	INFO_NOT_FOUND
	0x00FF	GENERAL_FAILURE

Description:

This request is used by a module to obtain the configuration information parsed from the user-defined protocol configuration file PROTOCOL.INI. Modules invoke this function during device driver initialization to obtain this information for initializing configuration variables and making dynamic memory allocations and to determine their inter-module bindings.

In DOS dynamic mode, INFO_NOT_FOUND is returned if the Protocol Manager detects that the structured memory image is not valid. This can occur if prior to loading a dynamic module the structured configuration memory image was not registered with the Protocol Manager via a RegisterProtocolManagerInfo command or if the memory image got corrupted between registering it and getting it via the current primitive. The corruption

might occur if another DOS program is loaded between the memory image registrations and the memory image read operation by a dynamic protocol invoking the GetProtocolManagerInfo primitive.

This request is valid in both the static and dynamic modes of Protocol Manager operation. In the static mode, this request is only valid prior to binding and starting. Invoking this primitive in static mode after all modules are bound and started will cause INVALID_FUNCTION to be returned by Protocol Manager.

RegisterModule

Purpose: Register a module and its bindings.

Opcode - 2

Status - On return contains request status

Pointer1 - Contains a FAR pointer to the module's common characteristics table. The module must have all information in that table filled in except for the Module ID which is filled in by the Protocol Manager on return.

Pointer2 -Contains a FAR pointer to a BindingsList structure of the modules to which this module wishes to be bound to. The Protocol Manager will use only the information passed in the BindingsList to determine the relevant module bindings. This pointer can be FAR NULL to indicate that this module will not currently bind to any module. This latter option is useful for dynamic OS/2 modules that need to register their module name with the Protocol Manager but do not wish to remain fully resident (and therefore bind) at the current time. This non-bindable registration permits the dynamic driver to reregister with a BindingsList when it is later reloaded and made operational.

Word1 -Unused

Returns:		
	0x0000	SUCCESS
	0x0008	INVALID_FUNCTION
	0x002C	ALREADY_REGISTERED
	0x00FF	GENERAL_FAILURE

Description:

This request is used by a driver or dynamically loadable executable to identify one of its contained modules to the Protocol Manager. After calling RegisterModule, a static driver must remain installed and respond to system requests. A dynamic OS/2 driver must leave its system entry function code permanently locked in memory. A dynamic DOS module must remain installed and respond to system requests until it is unbound and unloaded. This registration is accomplished by passing a pointer to the module's characteristics table to the Protocol Manager. The driver also passes a bindings list requested by the module. The bindings list contains the one or more module names which the module wishes to bind to as a client. This bindings information is later used by the Protocol Manager to determine the necessary sequence of InitiateBind commands to issue. This bindings list must persist while the protocol is operational. In the static default bindings case of one static protocol and one MAC, the bindings list pointer provided in this request can be NULL indicating that a protocol module by default will bind to the single underlying MAC. Otherwise in the

non-default bindings case, a NULL bindings list pointer provided in this request will indicate that this module will not bind to any other module at the current time and is not ready to initialize. In this latter case the Protocol Manager will not call the module's InitiateBind system function. A NULL binding list pointer is particuarly useful for dynamic OS/2 drivers that register their module name at INIT time, but are not to remain fully resident at startup time. This is called a non-bindable registration. A protocol module can also pass a non-NULL bindings list with a 0 number of bindings count. In the default bindings case, this is interpreted by the Protocol Manager to bind the protocol to the single underlying MAC. In the non-default bindings configuration this means that a protocol is registering without any lower bindings, but is required to be initialized by an InitiateBind call.

A driver which contains multiple modules can call RegisterModule multiple times, once for each module. The Protocol Manager responds to each request by assigning each module a module ID. The module ID is returned in the module's characteristics table on completion of the RegisterModule request.

If a module name is currently registered with the Protocol Manager, an attempt to register the same module name will fail and a status code of ALREADY_REGISTERED will be returned. A dynamic OS/2 driver is considered currently registered if it had previously registered with a non-NULL bindings list indicating a requirement to bind and/or start and it had not yet unbound. Thus a dynamic OS/2 driver can reregister with the Protocol Manager under the same module names if it either had unbound or had not previously made a bindable registration.

This request is valid in both the static and dynamic modes of Protocol Manager operation. In the static mode, this request in only valid prior to binding and starting. Invoking this primitive in static mode after all modules are bound and started will cause INVALID FUNCTION to be returned by the Protocol Manager. A registration of a dynamic module (bit 2 set of the "module function flags" in the Common Characteristics table) in static Protocol Manager mode is invalid and will generate INVALID_FUNCTION. It is mandatory that all static DOS and static and dynamic OS/2 device drivers invoke this function at least once at INIT time.

BindAndStart

Purpose: Initiate the binding process.

Opcode - 3

Status - On return contains request status

Pointer1 - Caller's virtual address of FailingModules structure. This structure in the caller's address space is filled in by the Protocol Manager prior to returning from BindAndStart. If BindAndStart reports an error, it contains the module names in ASCIIZ format of the upper module and lower module (may be a NULL string) reporting the error. If BindAndStart is successful then both are NULL strings.

```
struct FailingModules {
        char UpperModuleName[16]; /* Upper failing module */
        char LowerModuleName[16]; /* Lower failing module */
};
```

Pointer2 - Unused

Word1 - Unused

Returns:		
	0x0000	SUCCESS
	0x0007	INVALID_PARAMETER
	0x0008	INVALID_FUNCTION
	0x0020	ALREADY_STARTED
	0x0021	INCOMPLETE_BINDING
	0x0022	DRIVER_NOT_INITIALIZED
	0x0023	HARDWARE_NOT_FOUND
	0x0024	HARDWARE_FAILURE
	0x0025	CONFIGURATION_FAILURE
	0x0026	INTERRUPT_CONFLICT
	0x0027	INCOMPATIBLE_MAC
	0x0028	INITIALIZATION_FAILED
	0x0029	NO_BINDING
	0x0002D	PATH_NOT_FOUND
	0x0002E	INSUFFICIENT_MEMORY
	0x00FF	GENERAL_FAILURE

Description:

This is used to trigger the Protocol Manager bind and start sequence. This permits an application program (e.g., executing from a DOS batch or OS/2 command file) to trigger the bind sequence. The bind sequence is invoked by the Protocol Manager's calling each module's inter-module InitiateBind function. If an InitiateBind fails then BindAndStart will fail with same return code as the failing InitiateBind.

In the static mode of Protocol Manager operation, this request can be invoked only once to bind and start all static drivers. Successive invocations return INVALID_FUNCTION.

In the dynamic mode, this command tells the Protocol Manager to issue the IntitiateBind primitive to all dynamically loaded protocol drivers that have registered since the last InitiateBind (or since the beginning of time for the first call).

In DOS, the caller is required to invoke this primitive via the direct entry point rather than the DOS IOCTL method. The Protocol Manager will generate an INVALID_FUNCTION error if this function is invoked by an IOCTL. This will permit the protocol modules to make DOS function calls during their bind and start sequence initiated by this primitive (when the Protocol Manager calls the InitiateBind system entry point of the protocol). If the IOCTL were used, the bind/start sequence would be carried out inside of a DOS call and protocols would not be able to make further DOS calls within their initialization sequence in order to prevent DOS reentrancy.

In DOS the Protocol Manager loads PROTMAN.EXE to execute this commnad. The caller must have previously guaranteed that at least 20k of memory is available to load PROTMAN.EXE prior to invoking the BindAndStart primitive. In static VECTOR configurations (Chapter 7) PROTMAN.EXE will remain resident after BindAndStart completes. In such cases it is strongly recommended that the caller free as much memory as possible prior to calling BindAndStart so the PROTMAN.EXE will reside in the lowest memory possible. This will prevent large unusable gaps in DOS memory when the calling function terminates.

A utility, NETBIND.EXE, that invokes the BindAndStart primitive is provided with the Protocol Manager and is described in Appendix E.

GetProtocolManagerLinkage

Purpose: Retrieve Protocol Manager Dispatch and DS Value.

Opcode - 4

Status - On return contains request status

Pointer1 - On return contains the Protocol Manager Dispatch point.

Pointer2 - Unused

Word1 - On return contains the Protocol Manager DS.

Returns: -

0x0000	SUCCESS
0x0008	INVALID_FUNCTION
0x00FF	GENERAL_FAILURE

Description:

This request is used by a module to obtain the dispatch entry point and DS of the Protocol Manager. Direct calls may then be made by DOS & OS/2 Ring 0 drivers and DOS utilities to the dispatch entry point.

All dynamically reloaded OS/2 protocol drivers must issue this command to the Protocol Manager at CONFIG.SYS INIT time using the IOCTL mechanism and must save the Ring 0 Protocol Manager dispatch entry point and DS. When the driver subsequently re-registers with the Protocol Manager on reload at post INIT time, it must do so via the direct entry interface using the saved entry point and DS (since an IOCTL would be illegal at that time).

Any DOS utility that intends to invoke the BindAndStart or UnbindAndStop Protocol Manager primitives must first invoke this primitive to get the Protocol Manager's direct entry point.

This request is valid in both the static and dynamic modes of Protocol Manager operation. In the static mode, this request is only valid prior to binding and starting. Invoking this primitive in static mode after all modules are bound and started will cause INVALID_FUNCTION to be returned by the Protocol Manager.

GetProtocolIniPath

Purpose: A command to obtain the path to the PROTOCOL.INI file read by the Protocol Manager when it initialized.

Opcode - 5

Status - On return contains request status

Pointer1 - The virtual FAR pointer to a buffer, which will contain the returned PROTOCOL.INI pathname in ASCIIZ format on completion.

Pointer2 - Unused

Word1 - The length of the provided buffer on input.

Returns: -

0x0000	SUCCESS
0x0007	INVALID_PARAMETER
0x0008	INVALID_FUNCTION

Description:

This primitive can be called by an application program or dynamically loadable protocol that will read and parse the PROTOCOL.INI file to obtain the original location of the PROTOCOL.INI file used by the Protocol Manager when it initialized. This permits such a program to use the same file read by the Protocol Manager. The Protocol Manager returns only the pathname to the subdirectory containing the PROTOCOL.INI file, excluding the string "\PROTOCOL.INI", which may be up to 60 characters in length. This string will include the drive identifier and be fully qualified relative to the root. The buffer must be large enough to hold the returned string. If not, the contents of the buffer are undefined and the INVALID_PARAMETER error returned.

This request is valid in both the static and dynamic modes of Protocol Manager operation. In the static mode, this request is only valid prior to binding and starting. Invoking this primitive in static mode after all modules are bound and started will cause INVALID_FUNCTION to be returned by the Protocol Manager.

RegisterProtocolManagerInfo

Purpose: A command valid only in the dynamic mode to register the current starting address of the PROTOCOL.INI memory image with the Protocol Manager.

Opcode - 6

Status - On return contains request status

Pointer1 - The virtual FAR pointer to the structured memory image representing the parsed user configuration file, PROTOCOL.INI.

Pointer2 - Unused

Word1 - Length of structured memory image

Returns: -

0x0000	SUCCESS
0x0008	INVALID_FUNCTION

Description:

In dynamic mode, this command registers with the Protocol Manager the address of the PROTOCOL.INI memory image. It is assumed that prior to dynamically loading a protocol module, the PROTOCOL.INI file is re-read and re-parsed in some memory image. The pointer to the memory image is given to the Protocol Manager, so that it is available for the "GetProtocolManagerInfo" primitive of the dynamic initializing module that reads its configuration parameters.

In static mode, this command is illegal and the INVALID_FUNCTION error code is returned.

A utility, READPRO.EXE, that reads and parses the PROTOCOL.INI file into a memory image and registers this with the Protocol Manager is provided with the Protocol Manager and is described in Appendix E.

InitAndRegister

Purpose: An optional dynamic OS/2 command to dynamically restart the prebind initialization of a dynamically reloadable protocol driver.

Opcode - 7

Status - On return contains request status

Pointer1 - Unused

Pointer2 - FAR virtual pointer to an ASCIIZ buffer containing the name of the module to be prebind initialized.

Word1 - Unused

Returns:

0x0000	SUCCESS
0x0007	INVALID_PARAMETER
0x0008	INVALID_FUNCTION
0x00FF	GENERAL_FAILURE

Description:

In OS/2 dynamic mode, this reactivates the transient portions of a protocol driver previously statically loaded at system startup, but for which the transient portions of the driver were not locked down. The command causes the Protocol Manager to invoke the system entry point of the specified module with the function"InitiatePrebind" in order for the driver to restart its prebind initialization. The prebind initialization functions are driver specific. However, it is expected that such functions might include

- locking down its dynamic segments using the DevHlp Lock command (lock type 1) and saving the returned lock handle.
- getting its PROTOCOL.INI configuration information
- doing its prebind initialization,
- and finally, re-registering with the Protocol Manager.

In static mode, this command is illegal and the INVALID_FUNCITON error code is returned.

UnbindAndStop

Purpose: A dynamic binding command to terminate a transient previously dynamically bound protocol module and to terminate its bindings.

Opcode - 8

Status - On return contains request status

Pointer1 - Failing modules as for the "BindAndStart" command

Pointer2 - If non-NULL, FAR virtual pointer to an ASCIIZ buffer containing the name of the module to be unbound.

If NULL, then terminates a set of previously dynamically bound protocol modules as defined below. Valid only for DOS.

Word1 - Unused

Returns: -

0x0000	SUCCESS
0x0007	INVALID_PARAMETER
0x0008	INVALID_FUNCTION
0x0002D	PATH_NOT_FOUND
0x0002E	INSUFFICIENT_MEMORY

Description:

This is used in the dynamic mode to terminate either a specific protocol module or a set of previously dynamically bound protocol modules and to terminate their binds. A "set" is the collection of protocol modules previously loaded or reloaded between two successive "BindAndStart" calls or between the last "BindAndStart" and this call. Successive "UnbindAndStop" commands with NULL Pointer2 arguments terminate protocol sets in the reverse order in which they were bound. The Protocol Manager removes reference to the protocols from its VECTOR (for MAC unbindings) table and general binding tables. The Protocol Manager issues an "InitiateUnbind" command to each protocol to be unbound so that the protocol can issue an "Unbind" command to the modules it is bound to. For MAC unbindings, the "Unbind" is issued back to the Protocol Manager VECTOR. The NULL Pointer2 option is used in DOS environments for TSR protocol modules in which the unbind sequence usually proceeds in reverse order of the bind sequence. The non-NULL Pointer2 option must be used in OS/2 environments. The NULL Pointer2 option is invalid for OS/2.

In DOS, the caller is required to invoke this primitive via the direct entry point method rather than the DOS IOCTL method. The Protocol Manager will generate an INVALID_FUNCTION error if this function is invoked by an IOCTL. This will permit the protocol modules to be terminated to make DOS function calls during their unbind/stop sequence initiated by this primitive (when the Protocol Manager calls the InitiateUnbind system entry point of the protocol). If the IOCTL were used, the unbind/stop sequence would be carried out inside of a DOS call, and protocols would not be able to make further DOS calls within their termination sequence in order to prevent DOS reentrancy.

In DOS the Protocol Manager loads PROTMAN.EXE to execute this command. The caller must have previously guaranteed that at least 20K of memory is available to load PROTMAN.EXE prior to invoking the UnbindAndStop primitive.

A utility, UNBIND.EXE, that invokes the UnbindAndStop primitive is provided with the Protocol Manager and is described in Appendix E.

In static mode, this command is illegal and the INVALID_FUNCTION error code is returned.

BindStatus

Purpose: A command to obtain information from the Protocol Manager about the current set of bound modules.

Opcode - 9

Status - On return contains request status

Pointer1 - On input, under OS/2 only, if the caller is in Ring 3, this must be a FAR virtual pointer to a buffer where the returned information will be stored.
- On input, under DOS or in OS/2, if the caller is in Ring 0, this pointer must be NULL.
- On output, Pointer1 points to the root tree.

Pointer2 - NULL

Word1 - only used in OS/2
- Length of buffer (input) and bytes copied (output).

Returns:

-	0x0000	SUCCESS
	0x0008	INVALID_FUNCTION
	0x000D	BUFFER_TOO_SMALL

Description:

If enabled by the Protocol Manager's BINDSTATUS=YES parameter in PROTOCOL.INI, this command can be called at any time to obtain information from the Protocol Manager about the current set of bound modules. If this command is disabled, an attempt to invoke this command will return INVALID_FUNCTION.

The following characteristics tables are returned for the modules which qualify:

Common Characteristics

Service-Specific Characteristics (including the Multicast Address List for MAC modules)

Service-Specific Status

Media-Specific Statistics (for MAC modules only)

The tables are linked together into a bind tree using a new structure:

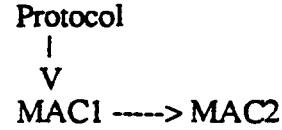
```
struct BindNode {
        struct cctable far *commonptr;
        struct BindNode far *down;
        struct BindNode far *right;
};
```

NOTE: There may be additional fields added to BindNodes in the future, so do not rely on its exact size.

A BindNode is linked to its Common Characteristics Table (CCT) by the CommonPtr field. The CCT's are then linked into a bind tree using the Right and Down pointers. Down points to the first BindNode bound below this one, and Right points to the next. At the top of the tree (the uppermost level), the Right pointers also link together the BindNodes as if they are bound to a virtual root BindNode.

A simple example might help illustrate this better:

which would be represented by the following bind tree:



where the BindNodes have been hidden to keep the diagram simple--only their Down and Right pointers are shown. The remaining Down and Right pointers would be NULL.

One option when making this call is to pass a NULL buffer pointer (in Pointer1), in which case the root BindNode pointer will be returned in Pointer1. The Protocol Manager uses BindNodes internally to build the bind tree. The caller can then run the current bind tree to obtain information. This is the only method supported under DOS. Under OS/2, this method will only work for Ring 0 drivers.

Under OS/2, Ring 3 programs must use a second method by providing a pointer to a buffer (in Pointer1) of a specified size (in Word1) to copy the characteristics tables into. In this case, the Protocol Manager will copy the qualifying tables into the buffer provided. The first entry in the buffer will be the root BindNode. The order of the remaining BindNodes and tables within the buffer is undefined. The BindNodes and their various tables are linked together by pointers which will be fixed up by the Protocol Manager to use the same selector as the buffer itself (i.e., Ring 3 if the buffer is Ring 3). Specifically, the Protocol Manager will fixup the following entries:

BindNode:
- CommonPtr
- Down
- Right

Common Characteristics:
- Pointer to service-specific characteristics
- Pointer to service-specific status

Service-Specific Characteristics
- Pointer to multicast address list (MAC's only)

Service-Specific Status
- Pointer to media-specific statistics (MAC's only)

The remaining pointers (e.g., dispatch tables and entry points) will be in an undefined state and must not be relied upon.

If the buffer was too small, BUFFER_TOO_SMALL will be returned, the pointers to tables which were not copied will be NULL, and the bytes copied return parameter (Word1) will indicate where the information was truncated.

The information returned is merely a snapshot at a particular point of time. The Protocol Manager will disable interrupts while copying individual status and media-specific statistics tables to guarantee their internal integrity. The caller cannot assume that all tables were copied in the same atomic operation however.

In the case of OS/2, if two or more modules are bound to the same lower module, the lower module's table is duplicated in the tree. Therefore, the Ring 3 caller will have to provide larger amount of buffer space for the returned information.

The number of nodes in the bind tree does not necessarily reflect the number of modules bound.

RegisterStatus

Purpose: A command to query whether a specific logical module is currently registered with the Protocol Manager.

Opcode: - 0x0A

Status: - On return contains request status

Pointer1 - NULL

Pointer2 - FAR virtual pointer to a 16-byte ASCIIZ module name

Word1 - NULL

Returns:

- 0x0000	SUCCESS
0x0008	INVALID_FUNCTION
0x002C	ALREADY_REGISTERED

Description:

This command can be called in either the static or dynamic mode to determine whether a specific logical module is currently registered with the Protocol Manager. This can be used by the caller to determine whether a specified module has already registered with the Protocol Manager to prevent duplicate registration. A SUCCESS status returned means that the specified module is not currently registered with the Protocol Manager. An ALREADY_REGISTERED status means that the module is currently registered.

In the static mode, this request is only valid prior to binding and starting. Invoking this primitive in static mode after all modules are bound and started will cause INVALID_FUNCTION to be returned by the Protocol Manager.

Chapter 6 - Protocol Manager

Protocol Manager Initialization

The Protocol Manager is loaded and initialized in both the OS/2 and DOS environment via the operating system CONFIG.SYS INIT sequence. It must be loaded before any protocol or MAC driver is loaded. In DOS, the Protocol Manager will be provided in a file called PROTMAN.DOS. For OS/2, the file is PROTMAN.OS2. The device name for the Protocol Manager is PROTMAN$ under DOS and \DEV\PROTMAN$ under OS/2 (the \DEV format is required by versions 1.2 and later of OS/2).

In DOS to save memory, an additional dynamically loadable component of the Protocol Manager called PROTMAN.EXE is provided. This file must reside in the same directory as the static device driver component, PROTMAN.DOS, itself. This file is called for execution by the Protocol Manager device driver component whenever the Protocol Manager primitives BindAndStart and UnbindAndStop are to be executed. In the static VECTOR mode (Chapter 7) PROTMAN.EXE will remain resident after BindAndStart executes.

The Protocol Manager reads the PROTOCOL.INI file at INIT time and parses it to create the configuration memory image passed to the protocol modules. The file is located in the \LANMAN directory of the boot drive or the directory given by the /I: parameter on the DEVICE=PROTMAN.xxx line in CONFIG.SYS. Under DOS, this image is relocated to just below the memory ceiling, where it must remain untouched until all binding has completed. The Protocol Manager computes a checksum of this image and checks it at bind time to guarantee that the image has not been modified in the interim. Note that this memory is not reserved by the Protocol Manager.

If the Protocol Manager CONFIG.SYS initialization is successful it is ready to support the initialization of the other drivers. However the initialization can be aborted for either of the following reasons:

1. The Protocol Manager did not have enough memory to hold the PROTOCOL.INI configuration memory image.

2. The Protocol Manager encountered a syntax error while parsing the PROTOCOL.INI file. This could have been an illegal hex or decimal parameter value, an overflow condition (numeric value could not fit into 32 bits) was encountered or a string was encountered with missing end quotes.

These conditions are flagged as fatal errors to prevent erroneous configuration parameters from propagating to the drivers for their operation.

Static Binding Sequence

The Protocol Manager can be configured to operate either in the static binding mode or in the dynamic binding mode. In the static binding mode, only statically loadable device drivers can be loaded and bound once at system initialization time. In the dynamic binding mode, dynamically loadable protocol drivers can be loaded and dynamically bound and unbound during system operation on a demand basis. Static drivers can also be loaded at

INIT time in dynamic mode. The static binding sequence is described in this section. The dynamic binding sequence is described in Chapter 7, "VECTOR and Dynamic Binding."

To determine the binding sequence, the Protocol Manager builds a tree representing the bindings for all the modules in the system. MAC drivers are at the bottom, and the highest level (for example, NetBIOS) protocol layers at the top. It then binds module pairs together from the bottom up. To do this, it issues an InitiateBind to the upper module in the pair, passing it the characteristics table of the lower module. The upper module is expected to issue a Bind to the lower module (if it is acceptable) and return. This continues with the next higher up module. If there is a module which is not bound to anything else, it receives an InitiateBind with a NULL characteristics table pointer.

To be more formal, the definitions listed below are required:

- A MAC driver is a protocol module with an upper layer interface level of one (MAC layer) and a lower layer interface level of zero (physical). It must support binding at its upper boundary.

- A MAC-layer entity is a protocol module with both upper and lower layer interface levels of one. It must support binding at its lower boundary.

- A standalone protocol module is one which has a lower layer interface level of zero and which does not support binding at its upper boundary.

The Protocol Manager builds a tree with multiple branches. Each MAC driver is at the base of a branch, with the protocol layers bound to it above it. Standalone modules are also considered branches by themselves. The left-to-right order is defined by the order in which the modules register with the Protocol Manager. The Protocol Manager does a pre-order transversal of the tree, issuing InitiateBinds to all of the nodes except the MAC drivers.

An important aspect of the binding scheme is that it allows for modules to specify that they only do binding from above or below. This is a requirement in cases where a monolithic module exposes several interfaces, such as a NetBIOS, TLI, and DLC. The TLI could be presented as a logical module that had an upper interface (the TLI) but no lower interface (since it uses a private internal interface to its DLC). Such a module would have a characteristics table with the following settings:

DWORD	Module function flags, a bit mask (hints only): Bit 0 - set (binds at upper boundary) Bit 1 - clear (doesn't bind at lower boundary)
BYTE	Protocol level at upper boundary of module: 4 - Transport
BYTE	Type of interface at upper boundary of module: 1 => TLI
BYTE	Protocol level at lower boundary of module -1 - Not specified
BYTE	Type of interface at lower boundary of module: For any level: 0 => private (ISV defined)
LPBUF	Pointer to upper dispatch table
LPBUF	Pointer to lower dispatch table (NULL)

Sequence for non-VECTOR configurations:

1. Protocol Manager driver (PROTMAN.OS2 for OS/2 or PROTMAN.DOS for DOS) is loaded during CONFIG.SYS initialization. The Protocol Manager must be configured ahead of any MAC or protocol drivers in CONFIG.SYS.

2. Protocol Manager initializes and reads PROTOCOL.INI to build the configuration memory image.

3. MAC and protocol drivers are loaded by the operating system. During its initialization processing, each driver optionally does the following:

   a. Open the PROTMAN$ device

   b. Invoke the GetProtocolManagerInfo primitive to PROTMAN$ to get a pointer to the configuration memory image.

   c. Read configuration parameters from the image and use these to finish initialization and build characteristics tables.

   d. Use the RegisterModule function once for each module to be defined to the Protocol Manager.

4. CONFIG.SYS processing ends and applications are started.

5. An application opens the PROTMAN$ device and issues the BindAndStart IOCTL. Such an application utility called NETBIND.EXE is provided with the Protocol Manager driver and is defined in Appendix E.

6. The Protocol Manager uses information passed on previous RegisterModule calls to determine the module binding hierarchy.

7. Proceeding from bottom to top of the binding hierarchy, the Protocol Manager uses InitiateBind to cause each module to bind to the module below it in the hierarchy. Each module getting this call responds by issuing a Bind call to the module specified by the Protocol Manager on InitiateBind.

8. When all modules have been bound, the Protocol Manager returns from BindAndStart.

The system is now fully operational. Vector configurations are similar, with the VECTOR being automatically inserted between layers one and two, if necessary (on top of the MAC driver as well as any MAC-layer entities which are present).

OS/2 Calling Convention

All of the Protocol Manager requests are supported by a single OS/2 IOCTL function. The services are demultiplexed via a function code specified in the ReqBlock structure.

This IOCTL has the following IOCTL request packet parameters:

1. Block Device Unit Code: Undefined since the Protocol Manager is a character device.

2. Command Code: 16 for Generic IOCTL.

3. Status: If the IOCTL corresponds to one of the Protocol Manager commands then the status field is returned with the ERR bit cleared signifying IOCTL successful completion. However the final status of the command is returned in the "status" field of the ReqBlock buffer as defined below. Note that if the command is recognized the ERR bit is always cleared regardless of the status returned in "status". However if the command is not recognized an IOCTL status UNKNOWN_COMMAND (3) is returned with the ERR bit set. Finally all of the commands return with the status "DON" bit set.

4. Category code: 0x81 which is the LAN Manager category code.

5. Function code: 0x58 for Protocol Manager command type.

6. Parameter buffer: Pointer to ReqBlock structure.

7. Data buffer: Unused and therefore the pointer is NULL.

By using the GetProtocolManagerLinkage request a module may obtain the Protocol Manager dispatch point and DS. Once a module obtains the Protocol Manager's entry point and data segment it passes the a request to the Protocol Manager via the following function call:

```
int (far pascal *ProtManEntry)(ReqBlockPtr, DataSeg);
struct ReqBlock far *ReqBlockPtr;
unsigned DataSeg;
```

where:

ReqBlockPtr = a FAR pointer to the request block

DataSeg = the Protocol Manager's data segment base.

The Protocol Manager returns in AX the same return code that is returned in the ReqBlock "status".

DOS Calling Convention

All of the Protocol Manager requests are supported by a single DOS IOCTL function. The services are demultiplexed via a function code specified in the ReqBlock. This IOCTL should be requested via Interrupt 21 with general registers loaded with the following contents:

AH = 44H for IOCTL request
AL = 02H for device input
DS:DX = Pointer to ReqBlock structure
CX = 14 for the size of the ReqBlock structure
BX = Handle from DOS Open of "PROTMAN$"

This IOCTL generates the following IOCTL request packet parameters:

1. Block Device Unit Code: Undefined since the Protocol Manager is a character device.

2. Command Code: 3 for IOCTL input.

3. Status: If the IOCTL corresponds to one of the Protocol Manager commands then the status field is returned with the ERR bit cleared signifying IOCTL successful completion. However the final status of the command is returned in the "status" field of the ReqBlock buffer as defined below. Note that if the command is recognized the ERR bit is always cleared regardless of the status returned in "status". However if the command is not recognized an IOCTL status UNKNOWN_COMMAND (3) is returned with the ERR bit set. Finally all of the commands return with the status "DON" bit set.

4. Media Descriptor Byte: Unused

5. Transfer Address: Pointer to ReqBlock structure.

6. Byte/Sector Count: 14

7. Starting Sector Number: Unused

By using the GetProtocolManagerLinkage request a module or application may obtain the Protocol Manager dispatch point and DS. It then makes a request to the Protocol Manager via the same direct calling mechanism as OS/2.

Chapter 7 - VECTOR and Dynamic Binding

In static mode, the VECTOR is a function that is implemented within the Protocol Manager that allows more than one protocol stack to drive a single MAC. In this mode, the Protocol Manager uses the VECTOR function only if it detects that more than one protocol is using the same MAC. If more that one MAC is attached to multiple protocol stacks, then an instantiation of the VECTOR is created for each MAC so attached.

In dynamic mode, the VECTOR function is always present unconditionally for protocol/MAC intermodule communications. There can be zero, one, or more protocol stacks that bind to a MAC, but the VECTOR function is still present. There can be zero protocols if there is only one dynamic protocol stack being used in the system and that stack is not currently loaded. In the dynamic mode, the VECTOR shields all static binding MACs from the interactions of dynamic binding and unbinding protocol modules.

Static VECTOR Binding

The Protocol Manager will modify the normal binding process if it detects that multiple protocols have requested the use of the same MAC in the PROTOCOL.INI file.

1. At INIT time from RegisterModule the Protocol Manager has determined the bind hierarchy and has found some MACs that bind to 2 or more protocols, signaling the insertion of VECTOR.

2. To a MAC that will support multiple protocol stacks, the Protocol Manager issues Bind passing a Protocol Manager characteristics table with entry points into the VECTOR module. The MAC starts itself and returns, passing back to the Protocol Manager a pointer to the MACs characteristic table.

3. For a protocol that is part of a multiple protocol stack binding to the single MAC that was issued the previous Bind command, the Protocol Manager issues InitiateBind passing as the bind inter-module entry point, an entry point within the VECTOR module inside of the Protocol Manager.

4. The protocol module responds by issuing a Bind request back to the Protocol Manager through its VECTOR entry point. The protocol module passes its characteristics table to the Protocol Manager VECTOR. The Protocol Manager returns a characteristics table within the VECTOR which is copied from the associated MAC's characteristics tables, substituting the VECTOR entry points for the real MAC's entry points.

5. The protocol starts itself and returns from InitiateBind.

6. The Protocol Manager then issues subsequent InitiateBind's to other protocol modules as described above. If these other protocols are bound to a MAC through the VECTOR, the VECTOR procedure is repeated. Otherwise the non-VECTOR procedure is used.

At the conclusion of the binding process the VECTOR is in a position to filter calls as appropriate going in either direction across the MAC/protocol interface.

Dynamic VECTOR Binding

A dynamic protocol module can be loaded and bound after system initialization time on a demand basis. This dynamic loading and binding takes place in three phases:

1. The PROTOCOL.INI file is re-read.

2. The dynamic protocol module does some prebind initialization including getting its PROTOCOL.INI configuration parameters and registering with the Protocol Manager.

3. The dynamically loaded protocol module dynamically binds to other modules given in its bind specification. If these other modules are MAC's, the bind takes place through the Protocol Manager VECTOR facility.

At some point the dynamic protocol module is no longer required. The protocol module unbinds itself, terminates, and unloads itself from memory.

The mechanisms for dynamically binding and unbinding are carried out somewhat differently between DOS and OS/2. The procedures are briefly described below.

Dynamic Binding/Unbinding in the DOS Environment

1. In dynamic mode, both static and dynamic protocol modules can be supported. At startup time, the Protocol Manager performs initialization and binding of static modules as described in section "Static Binding Sequence." However, in the dynamic mode, the VECTOR function is always inserted.

2. At some point after system startup, a dynamic loadable protocol module (that can be a transient application program or a TSR) is demand loaded. For the dynamic protocol module to have its configuration parameters at initialization, the PROTOCOL.INI file must be re-read. Either an application program or the protocol module itself reads and parses the PROTOCOL.INI file into the configuration memory image. It is suggested that the application or protocol module obtain the location of the PROTOCOL.INI file using the "GetProtocolIni" primitive. A pointer to this memory image is passed to the Protocol Manager via the "RegisterProtocolManagerInfo" primitive. This is required since the configuration memory image created by the Protocol Manager at INIT time is not valid at post INIT time. An application utility, READPRO.EXE, that reads and parses PROTOCOL.INI is provided with the Protocol Manager and is described in Appendix E.

3. After loading, the protocol module initializes. Minimally, the protocol gets its PROTOCOL.INI configuration information from the Protocol Manager via "GetProtocolManager Info," does its prebind initialization, and registers with the Protocol Manager via "RegisterModule."

4. Either an application or the dynamic protocol module itself requests that the Protocol Manager initiate the binding sequence via the "BindAndStart" primitive. This causes the bind sequence described in steps 3 to 5 of the section "Static VECTOR Binding" to be executed. After the bind, the dynamic protocol is ready for use. An application utility, NETBIND.EXE, to initiate the binding sequence is provided with the Protocol Manager and is described in Appendix E.

5. During operation, all protocol commands to the MAC go through the VECTOR

6. When the dynamic protocol module is ready to terminate, either it or an application program issues the "UnbindAndStop" command to the Protocol Manager. This causes the Protocol Manager to call the protocol's "InitiateUnbind" system entry point. In turn, this allows the protocol to issue "Unbinds" to other modules it was bound to and to do final cleanup before terminating. On return from the "UnbindAndStop" command, the protocol can be removed from memory. An application utility, UNBIND.EXE, to initiate the unbinding sequence is provided with the Protocol Manager and is described in Appendix E.

Dynamic Binding/Unbinding in the OS/2 Environment

1. In OS/2, all dynamic protocol modules are multi-segment OS/2 device drivers. A dynamic OS/2 protocol differs from a static one in that the dynamic module has code and/or data segments that may be swapped out of virtual memory when not needed. These extra code and data segments must be specified with IOPL in the module's .DEF file so that they are marked as movable/swappable and not discardable by OS/2. In a static protocol module all segments are permanently locked in memory. A dynamic protocol module uses the OS/2 DevHlp Lock and Unlock calls (using a lock type of 1) to lock and free its code and/or data segments as needed. A dynamic protocol module is able to re-register multiple times with the Protocol Manager and to dynamically bind with other configured modules. When no longer required, the dynamic module can unbind and the dynamic memory segments can be Unlock'ed to free up the memory. Static OS/2 protocol modules register and bind only at system initialization time. They do not unbind.

2. Since all OS/2 dynamic protocol modules are OS/2 device drivers they may perform some INIT time initialization. The protocol must always register at INIT time with the Protocol Manager via "RegisterModule". A protocol that is not required at system startup must still register with the Protocol Manager at INIT time passing a NULL BindingsList pointer in the "RegisterModule" primitive. This is called a non-bindable registration. In this case the protocol need not lock down its extra code and data segments. It does, however, need to save the selector values for its dynamic code and data segments. The device driver's device header, strategy routine, and the NDIS system entry routine must reside in the driver's main code and data segments (the first ones in the driver) which are permanently locked down. A driver required at system startup must pass a non-NULL BindingsList pointer if it has modules it is required to bind to (a bindable registration). A driver required at system startup must go ahead and DevHlp Lock its other segments at INIT time, making sure to save the lock handle returned by the call. Also at INIT time, the protocol module must invoke the "GetProtocolManagerLinkage" primitive to get and save the Protocol Manager's Ring 0 direct entry point and DS.

3. Assuming that the protocol was not required at system startup time, at some point in time later it needs to be dynamically bound. At this point the module needs to get its PROTOCOL.INI configuration parameters, lock down its code and data segments, and perform its bindings. If the configuration parameters are not retained in the base data segment, the protocol must re-read the PROTOCOL.INI file. This is done in a similar fashion to that described for DOS. The "InitAndRegister" primitive is the standard facility that lets the Protocol Manager request the protocol to reload its dynamic segments and perform its prebind initialization. Upon receiving the "InitAndRegister" primitive, the Protocol Manager calls the protocol driver's system entry point with "InitiatePrebind", allowing the protocol to perform its prebind initialization. The protocol module uses this opportunity to issue DevHlp Lock calls (lock type 1) on it's dynamic segments to bring them back into memory. The handle returned from the Lock call must be saved for later unlocking. Also at this juncture, the protocol can get its PROTOCOL.INI memory image from the Protocol Manager via the direct entry point "GetProtocolManagerInfo" function. It may also do other prebind initialization and finally register with the Protocol Manager via the direct entry point "RegisterModule" function. If the protocol module had previously made a non-bindable registration at system startup, then the current registration affords it the opportunity to specify its bindings to the Protocol Manager.

4. The bind and postbind initialization step is similar to that described for DOS. Again, any protocol binds to MAC's are performed through the VECTOR.

5. During protocol operation, any protocol commands to a MAC go through the VECTOR.

6. When the protocol is no longer required, an application or the protocol itself can issue the "UnbindAndStop" command to the Protocol Manager. The sequence is similar to that described for DOS. The OS/2 driver, however, issues DevHlp Unlock commands against all of its dynamic segments so that these may be swapped out from memory. The previously saved Lock handle is required on this call.

VECTOR Demultiplexing

The Vector dispatches incoming frames to protocol stacks using either a preprogrammed default or user statically defined priority polling mechanism. The default mechanism is based on the "Interface Flags" variable in the protocol's lower dispatch table. These flags describe the protocol according to the kinds of frames it handles. Protocols that handle:

- Non-LLC frames
- LLC frames with specific LSAPs
- LLC frames with non-specific LSAPs

According to default dispatch priority, VECTOR polls protocols in that order (and within that order, in the order they registered) until it finds one that does not return FRAME_NOT_RECOGNIZED or FORWARD_FRAME in the indication. For specific protocols, this default may be overridden by specifying the bracketed name of the protocol with the Protocol Manager PROTOCOL.INI keyword PRIORITY. Protocols with static priorities specified in this manner are polled by the VECTOR before any protocol not so specified. Protocols with static priorities are themselves polled in the order in which their bracketed names appear in the PRIORITY keyword parameter list. Of course, a protocol

appearing in the static list is only polled if it is registered with the Protocol Manager and has bound to the MAC offering up the frame.

Appendix A - System Return Codes

This appendix lists return codes used in this version of the NDIS specification. Note that new error codes may be added in the future. Both protocol and MAC driver developers must design their code to allow for this.

0x0000 SUCCESS: The function completed successfully.

0x0001 WAIT_FOR_RELEASE: The ReceiveChain completed successfully but the protocol has retained control of the data buffer. ReceiveRelease will be called to release the data buffers.

0x0002 REQUEST_QUEUED: The current request has been queued. If the request handle is non-zero the module will call TransmitConfirm or RequestConfirm when the request completes.

0x0003 FRAME_NOT_RECOGNIZED: Returned from the protocol when a MAC does an Indication and the frame does not make sense to the protocol. This will be interpreted by the VECTOR to mean that the next protocol in line ought to be called with the Indication.

0x0004 FRAME_REJECTED: A received frame was recognized but it was discarded. The buffer may be immediately re-used.

0x0005 FORWARD_FRAME: A protocol wishes the received frame to be offered to other protocols but wishes to receive an IndicationComplete. This will be interpreted by the VECTOR to mean that the next protocol in line ought to be called with the Indication.

0x0006 OUT_OF_RESOURCE: The module is in a transient out of resource condition. The current request was not completed.

0x0007 INVALID_PARAMETER: One or more parameters was invalid.

0x0008 INVALID_FUNCTION: A command function was requested when it was not legal to do so or a invalid request was made.

0x0009 NOT_SUPPORTED: A valid request which is not supported by the Module was issued.

0x000A HARDWARE_ERROR: A hardware error occurred during the execution of this request. The request was not completed successfully and this can be considered non-fatal.

0x000B TRANSMIT_ERROR: The packet was not transmitted. May indicate a local resource problem, excessive collisions, or a remote resource problem. On Token Ring networks, this would be returned if the destination address was recognized but the receiver was out of buffers. This is a non-fatal error and can be taken as a hint that the packet should be retransmitted.

0x000C NO_SUCH_DESTINATION: The destination address was not recognized by any adapter on the local ring. This error is Token Ring specific and can be interpreted to mean that source routing must be invoked to reach the destination.

0x000D BUFFER_TOO_SMALL: The buffer provided was too small for the information being returned. Some commands may still return partial information.

0x0020 ALREADY_STARTED: The Protocol Manager has already started the network drivers. This error occurs when BindAndStart is called more than once.

0x0021 INCOMPLETE_BINDING: This bind-time error occurs when the Protocol cannot complete all of the bindings described in the bindings list, most probably due to missing modules.

0x0022 DRIVER_NOT_INITIALIZED: This bind-time error occurs when the MAC does not initialize properly during system boot, and a subsequent request is made to the MAC.

0x0023 HARDWARE_NOT_FOUND: This bind-time error occurs when the network adapter is not found by the MAC.

0x0024 HARDWARE_FAILURE: This error occurs in the following cases: network adapter reset failed, network adapter diagnostics failed, network adapter is not responding, network adapter is not found by the MAC. This error can be considered fatal.

0x0025 CONFIGURATION_FAILURE: This bind-time error occurs when the configuration is unacceptable to the network adapter.

0x0026 INTERRUPT_CONFLICT: This bind-time error occurs in OS/2 only, when an interrupt from some other device in the computer conflicts with the network adapter's.

0x0027 INCOMPATIBLE_MAC: This bind-time error occurs when a Protocol determines a MAC is not compatible for the binding operation. Thus, binding cannot proceed.

0x0028 INITIALIZATION_FAILED: This bind-time error occurs when a Protocol fails its initialization.

0x0029 NO_BINDING: This bind-time error occurs to indicate that the binding was not performed. This error can occur if a protocol driver took an error exit during its initialization or if a protocol driver has its upper level incorrectly specified as a MAC.

0x002A NETWORK_MAY_NOT_BE_CONNECTED: This bind-time error indicates that the adapter may not be connected to a network. Intended to be suggestive of corrective action by the user.

0x002B INCOMPATIBLE_OS_VERSION: This bind-time error indicates that a protocol or MAC driver does not support the version of DOS or OS/2 being used.

0x002C ALREADY_REGISTERED: This error is returned by the Protocol Manager if an attempt is made to register a module with a module name already registered with the Protocol Manager. It is also returned from a "RegisterStatus" primitive to indicate that the name is already registered.

0x002D PATH_NOT_FOUND: This error is returned by the DOS Protocol Manager if PROTMAN.EXE could not be found when attempting to execute a BindAndStart or UnBindAndStop command.

0x002E INSUFFICIENT_MEMORY: This error is returned by the DOS Protocol Manager if PROTMAN.EXE could not be loaded due to insufficient DOS memory when attempting to execute a BindAndStart or UnbindAndStop command.

0x002F INFO_N0T_FOUND: This error is returned by the DOS Protocol Manager in a GetProtocolManagerInfo command if the PROTOCOL.INI structured configuration memory image is not present or previously invalidated due to being overwritten or corrupted.

0x00FF GENERAL_FAILURE: Unspecified failure during execution of the function

0xF000 - 0xFFFF: Reserved for vendor defined error returns. These errors are treated as GENERAL_FAILURE.

Appendix B - Reference Material

OS/2 Device Drivers Guide

DOS Technical Reference

ANSI/IEEE standard 802.2 - 1985 (ISO/DIS 8802/2) Logical link control standard.

ANSI/IEEE standard 802.5 - 1985 (ISO/DIS 8802/5) Token ring local area network standard.

ANSI/IEEE standard 802.3 - 1985 (ISO/DIS 8802/3) Carrier Sense Multiple Access with Collision Detection local area network standard.

The Ethernet. A Local Area Network. Data Link Layer and Physical Layer Specifications, V2.0, November 1982. Also known as the "Ethernet Blue Book"

IBM Token Ring Network PC Adapter Technical Reference (69X7830)

IBM Token Ring Network Architecture Reference - November 1985 (6165877)

Information processing systems - Open Systems Interconnection - Basic Reference Model, (ISO 7498) The OSI reference model.

Appendix C - 802.3 Media Specific Statistics

MEDIA SPECIFIC STATISTICS TABLE STRUCTURE:

The 802.3 media specific statistics structure is defined as follows:

Statistics in **bold** are manditory, all others are strongly recommended.
Reserved slots should return as 0xFFFFFFFF (unsupported).

WORD	Length of 802.3 statistics structure, including this field
WORD	802.3 statistics structure version level (1)
DWORD	**Total frames with alignment error**
DWORD	Reserved (Obsolete statistic)
DWORD	Total frames with overrun error
DWORD	Reserved (Obsolete statistic)
DWORD	Total frames transmitted after deferring
DWORD	Total frames not transmitted - max (16) collisions
DWORD	Reserved (Obsolete statistic)
DWORD	Total late (out of window) collisions
DWORD	**Total frames transmitted after exactly one(1) collision**
DWORD	**Total frames transmitted after multiple collisions**
DWORD	Total frames transmitted, CD heartbeat
DWORD	Reserved (Obsolete statistic)
DWORD	Total carrier sense lost during transmission
DWORD	Reserved (Obsolete statistic)
DWORD	Total number of underruns (V2.0.1 and later)

When updating the statistics counters, a frame is counted in all the supported counters that apply.

Examples:

(a) A 'Multicast frame received ok' is counted in the the following statistics counters:

- Total multicast frames received ok
- Total frames received ok

(b) A 'Transmit Broadcast frame with one collision' is counted in all the following statistics counters :

- Frames transmitted with only one collision.
- Total broadcast frames transmitted.
- Total frames transmitted ok.

MEDIA SPECIFIC STATISTICS DEFINITIONS:

Frames received with alignment error
(NumberOfFramesReceivedWithAlignmentErrors)

This contains a count of frames that are not an integral number of bytes in length and do not pass FCS check. Reports on alignments errors "as the station sees it".

Frames received with overrun errors

This contains a count of frames which could not be accepted due to a DMA overrun error.

Frames transmitted after deferring (NumberOfFramesWithDeferredTransmission)

This counter does not include frames involved in collisions.

Frames not transmitted - max collisions exeeeded. (NumberOfFramesAbortedDueToExcessiveCollision)

This contains a count of the frames that are not transmitted successfully due to excessive collisions.

Frames transmitted with late (out-of-window) collision. (NumberOfLateCollisions)

This contains a count of frames that are involved in a out-of-window collision.

Frames transmitted after exactly one collision (NumberOfSingleCollisionFrames)

This contains a count of frames that are transmitted after exactly one collision.

Frames transmitted after multiple collisions (NumberOfMultipleCollsionFrames)

This conatins a count of frames that are transmitted after multiple number of collisions.

Frames transmitted, CD heartbeat (NumberOfSQETestErrors)

This contains a count of frames transmitted with CD(collision detection) signal missing.

Frames with carrier sense lost during transmission (NumberOfCarrierSenseErrors)

This contains a count of frames that experienced carrier sense lost(carrier sense signal not present at the receive pair of the controller) during transmission.

Frames transmitted with underrun error (V2.0.1 and later)

This contains a count of frames which could not be transmitted due to a DMA underrun error.

Appendix D - 802.5 Media Specific Statistics

MEDIA SPECIFIC STATISTICS TABLE STRUCTURE:

The 802.5 media specific statistics structure is defined as follows:

Statistics in **bold** are mandatory, all others are strongly recommended. Reserved slots should return as 0xFFFFFFFF (unsupported).

WORD	Length of 802.5 Statistics structure, including this field
WORD	802.5 Statistics structure version level (1)
DWORD	**FCS or code violations detected in repeated frame**
DWORD	Reserved (Obsolete statistic)
DWORD	Number of 5 half-bit time transition absences detected
DWORD	A/C errors
DWORD	Frames transmitted with abort delimiter
DWORD	**Frames transmitted that failed to return**
DWORD	Frames recognized, no buffer available
DWORD	Frame copied errors
DWORD	Number of frequency errors detected
DWORD	Number of times active monitor regenerated
DWORD	Reserved
DWORD	Reserved
DWORD	Reserved
DWORD	Reserved (Obsolete statistic)
DWORD	Number of underruns

When updating the statistics counters, a frame is counted in all the supported counters that apply.

MEDIA SPECIFIC STATISTICS DEFINITIONS:

FCS or code violations detected in repeated frame

This counter is incremented for every repeated frame that has a code violation or fails the Frame Check Sequence (FCS) cyclic redundancy check.

Number of 5 half-bit time transition absences detected

Also known as Burst Error, this counter is incremented every time 5 half-bit time transitions are not detected between SDEL and EDEL in a repeated frame.

A/C errors

Also known as **ARI/FCI set error**, this counter is incremented when a station receives more than one AMP or SMP MAC frames with AC (ARI/FCI) equal to zero without first receiving an intervening AMP MAC frame. This counter

indicates that the upstream Adapter is unable to set its AC (ARI/FCI) bits in a frame that it has copied.

Frames transmitted with abort delimiter

This counter is incremented each time the Adapter transmits an abort delimiter. This indicates that the frame was aborted in mid-transmission.

Frames transmitted that failed to return

This counter is incremented when a transmitted frame fails to return from around the ring due to time-out or the reception of another frame.

Frames recognized, no buffer available

Also known as **Receiver congestion**, this counter is incremented when a ring station is receiving/repeating a frame and recognizes a frame addressed to it, but has no buffer space available for the frame.

Frame copied errors

This counter is incremented when a ring station receives or repeats a frame from the ring with the ring stations's individual address, but with A = C = 1, indicating a possible duplicate address.

Number of frequency errors detected

This counter is incremented when a ring stations detects a signal frequency problem.

Number of times active monitor regenerated

This counter is incremented each time the active monitor is lost and regenerated.

Number of underruns

This counter is incremented each time a DMA underrun is detected.

Appendix E - Utilities Provided with the Protocol Manager

To save system integrators the effort to read and parse the PROTOCOL.INI file, to register it with the Protocol Manager, to invoke the binding and unbinding Protocol Manager primitives, and to report various Protocol Manager error conditions, 3 utilities are provided with the Protocol Manager in both the DOS and OS/2 environments and one utility is provided exclusively for the OS/2 environment:

1. NETBIND.EXE - Initiates the binding and operational startup of a set of modules previously loaded. It issues to the Protocol Manager the BindAndStart primitive and reports to the console any binding/initialization errors detected by the modules bound. This utility can be used in either the static or dynamic Protocol Manager modes of operation. In the static mode it should be invoked after all device driver modules are loaded (e.g. from AUTOEXEC.BAT in DOS or STARTUP.CMD in OS/2). In the dynamic mode it can be invoked either at system startup time as in static mode or after a set of dynamically loadable modules have been loaded and are ready to be run. There are no command line parameters associated with this utility.

2. UNBIND.EXE - Initiates the unbinding and termination sequence of a set of dynamically loadable modules previously loaded and bound. It issues to the Protocol Manager the UnbindAndStop primitive and reports to the console any unbinding/termination errors detected by the modules being unbound. The utility can be used only in the dynamic Protocol Manager mode of operation. Invocation in the static mode will generate an error. It should be invoked when it is desired to terminate (and release from memory) a set of dynamically loadable modules that have been previously loaded and bound. In DOS each invocation will terminate and unbind the last set of modules previously bound via the NETBIND.EXE utility. Modules can be bound and unbound in groups if required by invoking NETBIND.EXE for each group of modules to be bound together and later invoking UNBIND.EXE. UNBIND.EXE will unbind the groups only in the reverse order in which the groups were previsoulsy bound. If protocols are implemented so that they free themsleves from memory at the end of the unbind sequence, then this utility will free up the memory of all such protocols unbound. This utility has no effect on MAC drivers which are always static device drivers. In OS/2 the utility takes an argument string specifying the name of the module being unbound. In DOS there are no command line parameters associated with this utility.

3. READPRO.EXE - Reads the PROTOCOL.INI file, parses it into a memory image and registers this memory image with the Protocol Manager so that the image is available to dynamically loadable protocols when they request their configuration memory image information. By invoking the GetProtocolIniPath Protocol

Manager primitive, this utility assures that the PROTOCOL.INI file is read from the same subdirectory as that used by the Protocol Manager when it had initialized. The memory image is registered with the Protocol Manager via the RegisterProtocolManagerInfo primitive. This utility can be used only in the Protocol Manager dynamic mode of operation. The utility reports any detected error condtions on the console. It should be invoked prior to the loading of any dynamic modules. There are no command line parameters associated with this utility.

4. RELOAD.EXE.- Initiates the prebind initialization of an OS/2 dynamically loadable module. It issues to the Protocol Manager the InitAndRegister primitive containing the module name that was given as a command line parameter. The Protocol Manager calls the system entry point of the named module with the InitiatePrebind system function. The modules is required to reinitialize, which may include locking down swappable segments, requesting and parsing the PROTOCOL.INI image, and reregistering with the Protocol Manager in preparation for a subsequent NETBIND.EXE invocation. This utility reports any detected error to the console. It applies only to OS/2.

If the system integrator requires more functionality than that provided by these utilities, the integrator can write an application utility directly that performs the desired functionality and invokes the required Protocol Manager primitives described in Chapter 5. For example if in DOS a more flexible unbind facility to unbind in a user specified order is required, UNBIND.EXE can be replaced by a user written utility that invokes the UnbindAndStop primitive in which Pointer2 points to the name of the module to be unbound.

3TECH

The 3Com Technical Journal

ISSN1051-9637

NDIS CONCEPTS

The following article originally appeared in the Winter 1991 Issue of *3TECH, The 3Com Technical Journal.*

3TECH is published quarterly by 3Com Corporation, Santa Clara, CA 95052.

Subscriptions to to *3TECH* are available at a rate of $35 per calendar year. To order subscriptions to *3TECH* write to 3TECH Journal, 3Com, P.O. Box 58145, Santa Clara, CA 95052-9953. All orders must be prepaid and reference 3C2869.

3TECH, 3Com's Technical Journal, is published quarterly by 3Com Corporation, Santa Clara, CA 95052.
Officers: L. William Krause, Chairman; Eric A. Benhamou, President and Chief Executive Officer; Bob Finnochio, Executive Vice President, Field Operations; Christopher B. Paisley, Vice President and Chief Financial Officer; Debra Engel, Vice President, Corporate Services; Andy Verhalen, Vice President and General Manager, Network Adapter Division; John Hart, Vice President and Chief Technical Officer.

SUBSCRIPTIONS
Subscription rate: $35 per calendar year. To order subscriptions for 3TECH, or to report a change of address, write to 3TECH Journal, 3Com, P.O. Box 58145, Santa Clara CA 95052-9953, ATTN: Dept. CSI. All orders must be prepaid and reference 3C2869. Subscriptions may be entered or cancelled by 3Com employees by sending e-mail requests to: MPS USER:FO:3Com

SUBMISSIONS
Manuscript submissions, inquiries, and all other correspondence should be addressed to 3TECH's Editor: Marianne Cohn, 3Com Corporation, 5400 Bayfront Plaza, Santa Clara CA 95052-8145. Articles in 3TECH are primarily authored by 3Com employees, however, articles from non-3Com authors dealing with 3Com-related research or solutions to technical problems are encouraged for publication.

NDIS Concepts

By Rex Allers

The Network Driver Interface Specification (NDIS) is a standardized interface for OS/2 or DOS network platforms. NDIS provides access to network services at the Data Link layer and is especially useful if the access must be shared. Software developers who need to employ their own network protocol implementations can program to the NDIS interface and utilize NDIS-compliant drivers provided by network hardware vendors. This frees the protocol developer from programming directly to various network interface cards and solves compatibility problems on machines with multiple protocols.

This article explains why the Network Driver Interface Specification was developed, and describes its organization and operation. Some of the information is general and will be of interest to a broad group of people involved with networks. A good deal of the information is quite detailed and technical. It is intended primarily to give network programmers an overview of NDIS and what is involved in binding a protocol to a vendor-supplied MAC driver for a network adapter board.

The Old Way

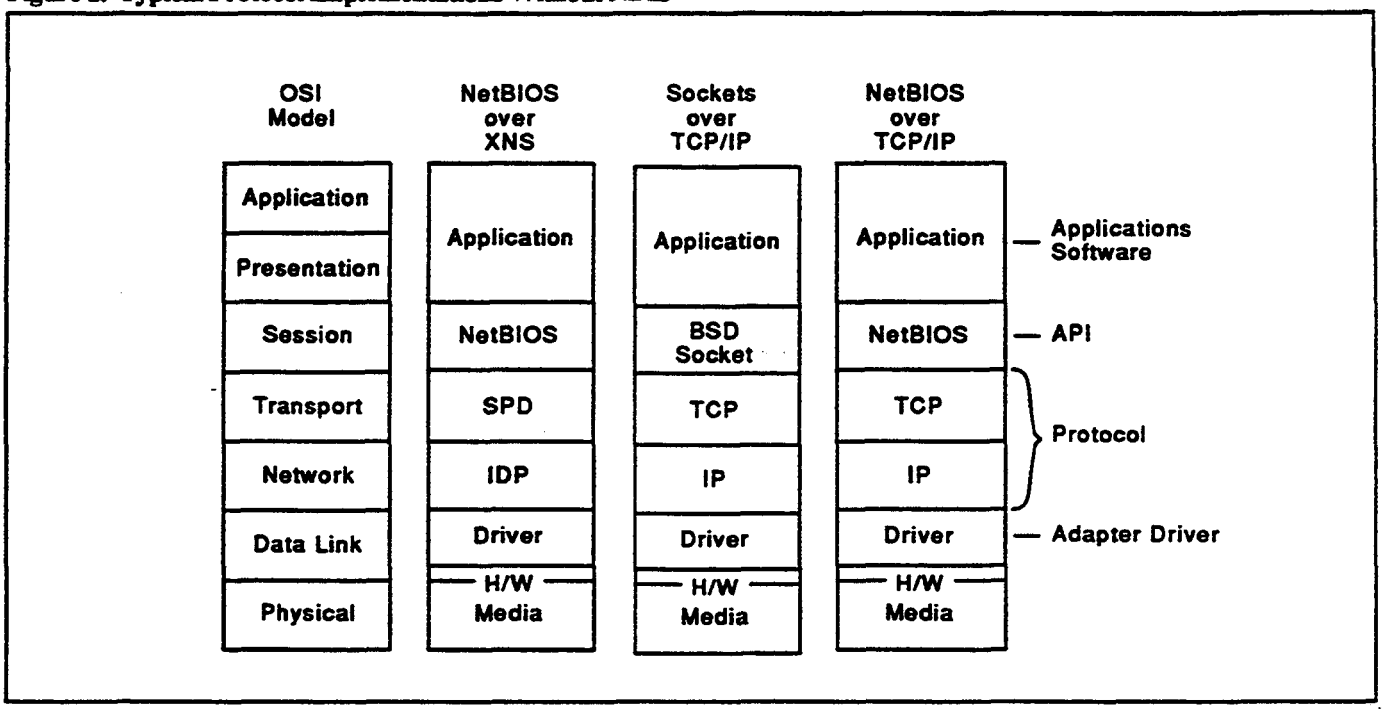
Traditionally, network software vendors for the MS-DOS environment have used ad hoc methods to implement the protocols and drivers that link applications to their resident network hardware. The entity that performs these network functions and provides communication between applications is usually referred to as a protocol stack. In the OSI Reference Model, the stack would correspond to the Data Link, Network, Transport, and Session layers, with some stacks possibly including higher layers. At the stack's top end is a user interface or some type of applications programming interface (API) and, at the bottom end, are the interfacing routines that control the network adapter hardware. In implementation, a stack might consist of one system driver, multiple drivers, a program, or a combination of drivers and programs.

Figure 1 shows three alternative stacks that could be used to perform equivalent network functions. In a typical implementation (for example, NetBIOS over XNS in Figure 1), the Data Link, Network, and Transport layers might be implemented as three separate system drivers, and the Session layer implemented as a TSR program. The interface between the layers would usually be accomplished through a proprietary interface developed by the vendor, and the application would communicate to NetBIOS via software interrupts. This works well in a homogeneous network environment but, as networks grow more complex, it is becoming desirable to have the flexibility to utilize mixtures of different protocols, application interfaces, and network media.

The Problem to Be Solved

Compatibility issues between various networking implementations can make it difficult or impossible to accomplish some seemingly simple tasks. For example, assume that we have an Ethernet network that has two types of file servers attached. One server runs an XNS protocol with NetBIOS at the Session layer, the other server runs a TCP/IP protocol with a Berkeley Socket Session interface.

Figure 1. Typical Protocol Implementations Without NDIS



We would like to write a program to run on a workstation that can copy a file from the first server to the second. In this example, let's say we have two sets of software from two vendors that are designed to communicate with each of the servers, and that the vendors have defined a programmer's interface that should allow us to write a program that talks to the two stacks. Assuming that we have enough memory to load both stacks at one time, we will probably find that our biggest configuration problem occurs at the bottom of the stacks.

At the Data Link layer, each of the vendors has supplied us with a driver for use with their protocol stack that can control the EtherLink II adapter board that we have in our station. Most likely, we will find that each of these drivers expects to have exclusive ownership and control of the EtherLink II. As one of the drivers tries to control the board, it interrupts or corrupts the functions being attempted by the other driver. What is needed is one driver that can control the adapter and be shared by the two protocols.

In May 1988, 3Com and Microsoft released NDIS, which was jointly developed in conjunction with LAN Manager. The NDIS specification is a standard designed to alleviate compatibility issues for both OS/2 and DOS network platforms. The NDIS specification should be beneficial to both the protocol-level network software developer, who now has a standard interface available, and the user, who gains from the flexibility and interoperability advantages of protocols using NDIS.

NDIS Organization

All network software components compliant with NDIS definitions are drivers. These drivers can be classified into two types: protocol drivers, and Media Access Control (MAC) drivers. NDIS allows protocol drivers to be device drivers, TSRs, or DOS applications; however, in this discussion, it is assumed that all NDIS drivers are device drivers—the simplest and most common implementation.

The MAC driver forms the bottom layer of the stack and is the driver that directly controls the network hardware. The remaining higher layers of the protocol stack are implemented in one or more protocol drivers.

The MAC layer is a sublayer within the OSI Data Link layer that is defined by IEEE 802 specifications. This

layer is the appropriate point for a driver that manages the network hardware and implements the transmission and reception of network data packets. NDIS MAC drivers are provided by 3Com and many other network hardware vendors (see Table 1 for a partial list) and can be used with any vendor's NDIS-compliant protocol drivers.

NDIS Stacks

All NDIS drivers, both MAC and protocol, share a common modular structure. Each driver has an upper and lower boundary. The drivers are linked to form a stack by connecting, or binding, the upper boundary of one driver to the lower boundary of another driver during the binding portion of driver initialization. This binding process can be repeated multiple times, linking several drivers, daisychain fashion, to form the stack. The MAC driver at the bottom of the stack always has its lower boundary connected to the physical layer—the network hardware.

The simplest configuration of drivers is one MAC driver supporting one network adapter card bound to a single protocol driver spanning from the MAC layer to the Session layer (see Figure 2). This forms a single protocol stack of two drivers. Optionally, the protocol part of this stack might be made using two or more protocol drivers to form a single stack of three or more drivers. NDIS also allows us to have two completely parallel stacks in one machine, each with its own adapter card and MAC driver, to implement two different protocols.

Table 1. Companies Supporting NDIS

Companies with NDIS MAC Drivers

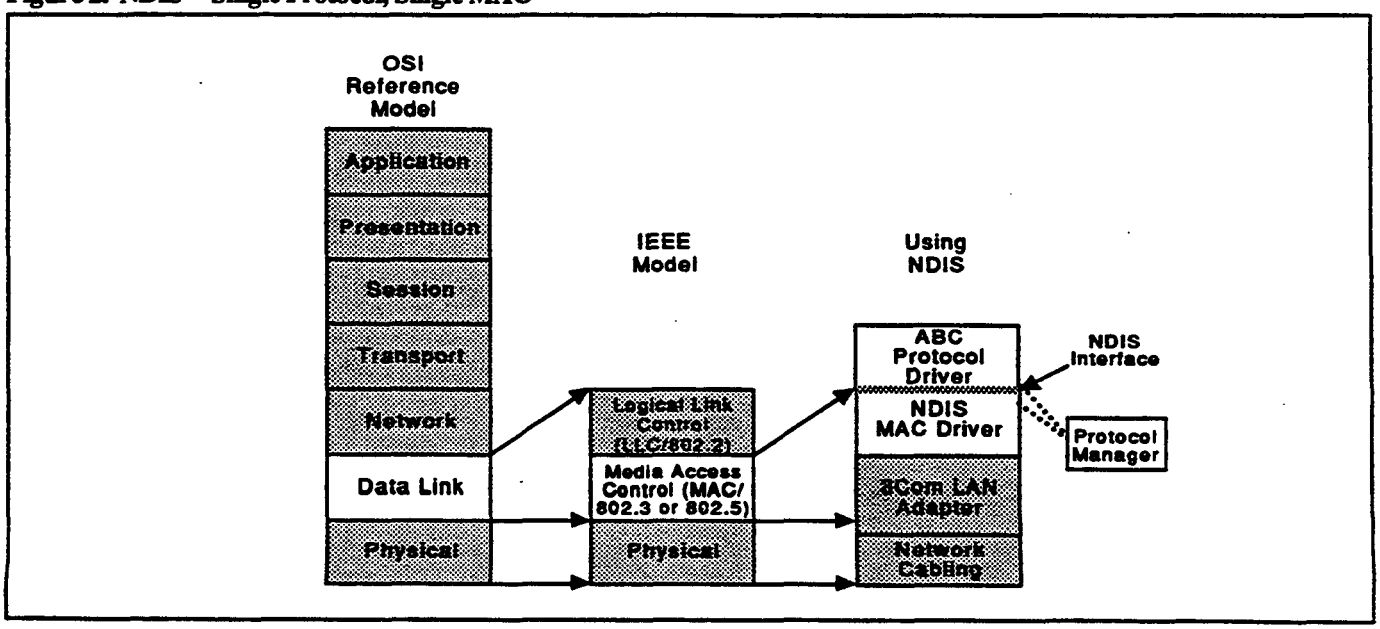
Note: The entire list of companies shipping NDIS MAC drivers is too long for this article, but includes:

3Com	Interlan
AST Research	Proteon
AT&T	Tiara
Compaq	Ungermann-Bass
Excelan	Western Digital
IBM	

Companies with OS or Applications Supporting NDIS

3Com	— LAN Manager
AT&T	— LAN Manager
Banyan	— Vines (workstation)
DEC	— LAN Works
FTP	— PC/TCP
IBM	— LANServer, OS/2 Extended
Microsoft	— LAN Manager
Pacer	— Pacerlink
Sun	— PC-NFS

Figure 2. NDIS—Single Protocol, Single MAC



More importantly, NDIS lets you have just one adapter card and a single MAC driver with the MAC driver bound to two separate protocol drivers (see Figure 3). Therefore, two protocols (for instance, XNS and TCP/IP) can share the same MAC driver and adapter card. This configuration solves the problem discussed earlier in which files need to be copied from two servers running different protocols.

To complete the picture, we can also have two adapter cards and MAC drivers, with both the MAC drivers bound to one protocol driver (Figure 4). This configuration could be used to create a network bridge with one protocol connected to two networks.

Figure 3. NDIS—Multiple Protocols, Single MAC

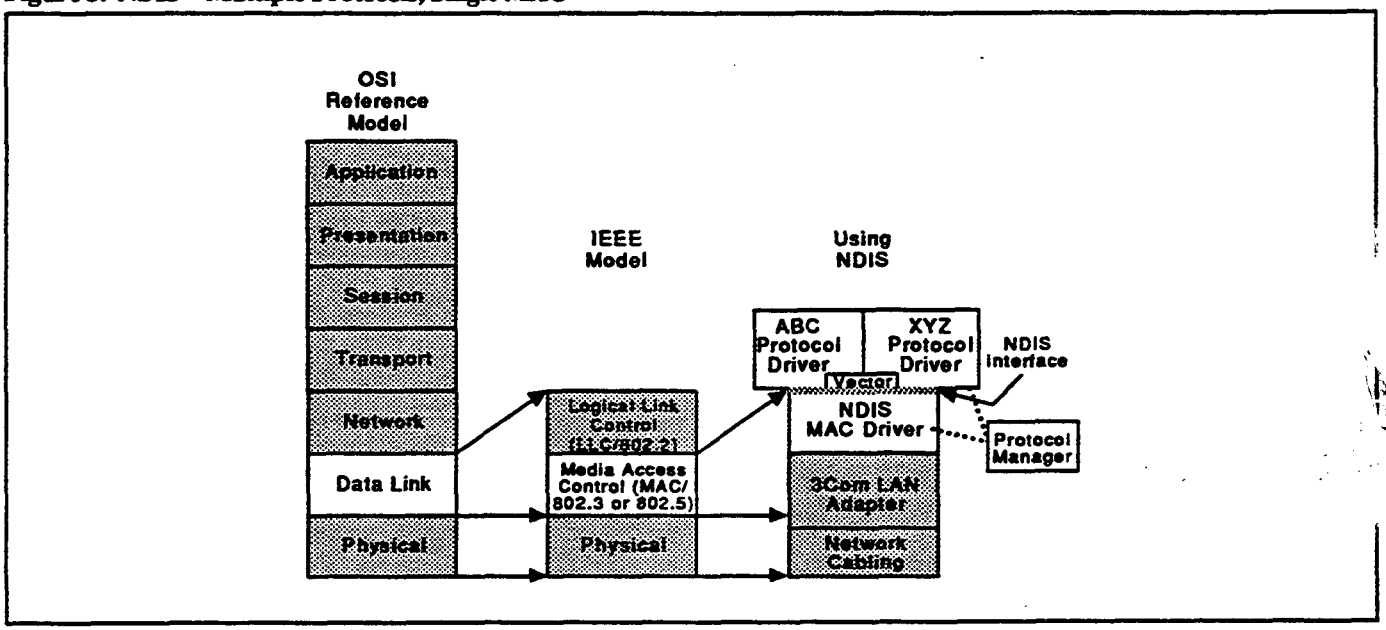
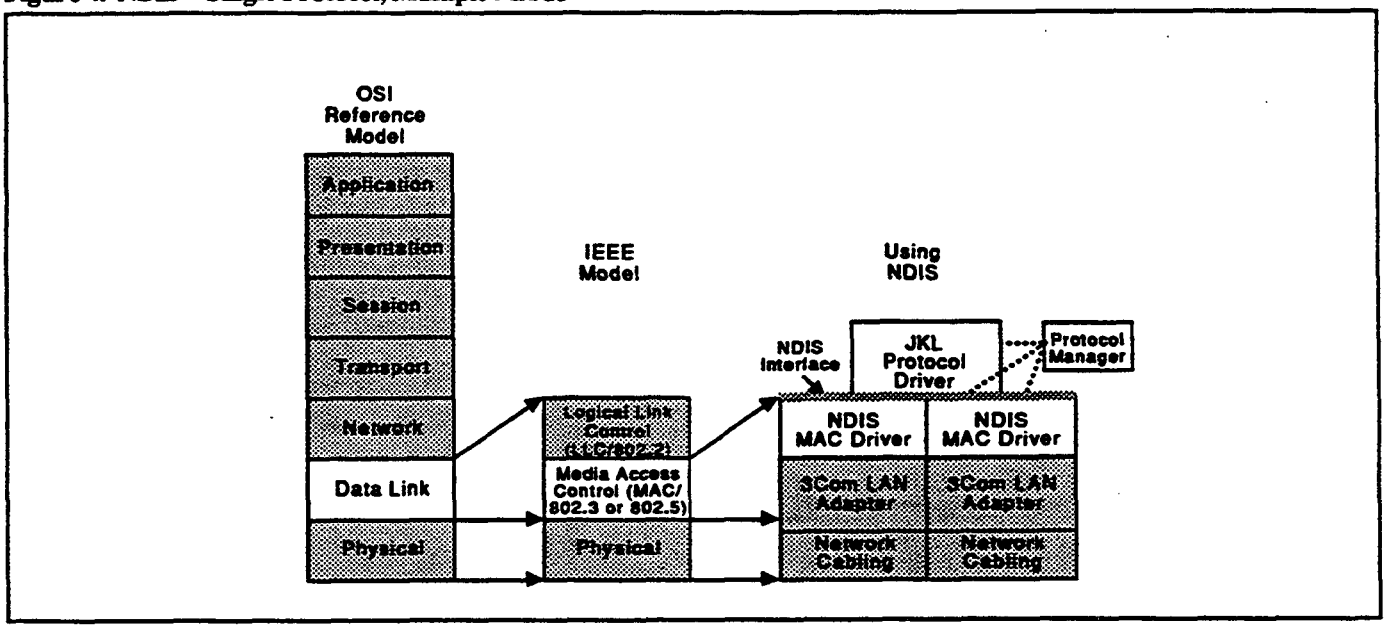


Figure 4. NDIS—Single Protocol, Multiple MACs



Driver Structures

The drivers communicate with each other by a defined set of primitives. The NDIS document has clearly specified a set of primitives for the interface between the MAC driver and protocol driver and for managing the NDIS driver binding process. Although it is possible to implement a protocol stack with multiple protocol drivers, currently no primitives are defined by NDIS for these upper layers. This is not a serious limitation, because a stack with multiple protocol drivers would generally have all of the protocol drivers common to one vendor. There is less need for a standardized interface between protocol drivers than there is at the MAC layer where sharing resources and multiple vendors are more likely.

Each driver contains a series of module-characteristic data structures that provide information about the purpose and capabilities of the driver, and that manage the linkage and operation of the driver during and after initialization.

The main structure is called the Common Characteristics table and contains the name of the driver and version information. This is the highest-level table for a driver; other types of characteristics tables are located from pointers in this table. The Common Characteristics table also contains basic information about what type of binding is supported at the upper and lower boundaries of the driver. The binding information is in the form of a byte identifying the OSI layer that is supported for the boundary. This byte can be examined by other drivers to determine if it is appropriate to bind to the driver.

The Common Characteristics table contains pointers to the other module characteristics tables—the Service-Specific Characteristics table, Service Specific Status table, and Upper and Lower Dispatch tables. These tables give specific information related to the service that the driver performs, manage its operation, and record linkage points to other drivers after the driver is bound.

Managing Binding and Initialization

To form the protocol stacks from the individual drivers we need to get the right drivers connected in the desired sequence. This is accomplished in the initialization and binding process. Three components are used to manage and control the process—PROTOCOL.INI (an ASCII configuration parameter file), PROTMAN.DOS or PROTMAN.OS2 (the protocol manager—a special driver), and NETBIND.EXE (a program that initiates the final driver binding process.)

The initialization and binding process is essentially the same whether the operating system is DOS or OS/2. Some minor adjustments need to be made (for instance, selecting either PROTMAN.DOS or PROTMAN.OS2) and some different parameters may be required, but the discussion that follows applies to either environment.

The ASCII file PROTOCOL.INI contains the instructions for assembling the protocol stack or stacks from the NDIS network drivers. It also contains parameters that are needed to configure the individual drivers. At CONFIG.SYS initialization time, the Protocol Manager Driver reads this file. The file is created—much as CONFIG.SYS is created—either directly, by the administrator typing the information with an editor, or by some type of installation program. The PROTOCOL.INI information is grouped into a number of logical sections of the form:

```
[module name]
   parameter=value
```

The module name is the name of the NDIS driver as contained in the Common Characteristics table for the driver. There will be one module section for each of the NDIS drivers that describes the driver's configuration. Each section can have multiple parameters, but must have at least one, the DRIVERNAME.

Figure 5 illustrates the contents of a simple PROTOCOL.INI file that has entries for three drivers. The first is Protocol Manager, the special driver that controls the binding process—more about its purpose later. In PROTOCOL.INI the Protocol Manager entry is currently optional, but it may be required in the future,

so it's a good idea to include it. The second module section is for the EtherLink II adapter's MAC driver, and the last section is for an arbitrary protocol driver.

Notice that in each section, the first parameter is DRIVERNAME=. This parameter must be included and must specify a name that uniquely defines the NDIS module. In most cases, it will be the driver name that the driver registers to the operating system during initialization. The driver determines the name that must be used, because it uses the DRIVERNAME entry as a key when searching PROTOCOL.INI data for its relevant module section.

Figure 5. PROTOCOL.INI File Contents

```
;*****************************
; Example PROTOCOL.INI file
;*****************************


[PROTMGR]
        DRIVERNAME=PROTMAN$

[ETHERLINKII]
        DRIVERNAME=ELNKII$
        INTERRUPT=3
        TRANSCEIVER=EXTERNAL

[PROTOTST]
        DRIVERNAME=PROTO$
        BUFFSIZE=2048
        BINDINGS=ETHERLINKII
```

Any number of additional, optional parameter entries can be included in a module section. One purpose of these parameters is to allow control of the driver configuration. A set of valid configuration options will be defined for any particular driver. In the case of the ETHERLINK II section in Figure 5, we have selected two of the possible options for this driver. INTERRUPT=3 tells the driver to use hardware interrupt channel 3, and TRANSCEIVER=EXTERNAL tells the driver to configure the adapter for its external transceiver. In the PROTOTST protocol driver, the BUFFSIZE parameter might direct a protocol to use a particular size for its internal buffers.

The BINDINGS= parameter is a special parameter that is valid only for protocol drivers and specifies the module name of the driver with which the protocol should attempt to bind on its lower boundary. This parameter determines which drivers will be bound together to form the stack or stacks. In Figure 5, PROTOTST is bound to the ETHERLINKII driver.

As mentioned earlier, the component of the NDIS environment that manages the binding process is the Protocol Manager, which has the file name PROTMAN.DOS or PROTMAN.OS2. Protocol Manager has two main functions: it keeps and manages common data for the NDIS drivers, and it controls the binding sequence. Functions of the Protocol Manager are needed by the NDIS drivers during their system level initialization, so the Protocol Manager driver must be loaded in CONFIG.SYS before any of the other NDIS drivers.

The Protocol Manager driver was written by 3Com and is available from both 3Com and Microsoft. Protocol Manager or NETBIND.EXE is available for any vendor for use in the initialization of their network software products. They are also a standard part of LAN Manager as shipped by 3Com and Microsoft.

Driver Initialization

The Protocol Manager gathers NDIS-related information during the system CONFIG.SYS initialization of the drivers. During initialization, the Protocol Manager driver reads the PROTOCOL.INI file and parses the information into a set of structures, called the Configuration Memory Image, which is accessible by the other NDIS drivers. Because other NDIS drivers use this information, the Protocol Manager must be the first to initialize.

As CONFIG.SYS processing continues, the operating system directs the other drivers to initialize. During initialization, they must open the Protocol Manager device (PROTMAN$) and then issue a GetProtocolManagerInfo primitive to obtain a pointer to the Configuration Memory Image (the PROTOCOL.INI data). The drivers find the section of this data that pertains to them and use any parameters found there to adjust their initialization

process. One result of this is that drivers may modify their loaded size, based on parameter requirements, to optimize host memory consumption. If the driver is a protocol and it finds a BINDINGS= parameter, it will assess whether or not this binding is valid. Finally, the driver must issue a RegisterModule primitive to the Protocol Manager to register itself. During this register, the driver passes a pointer to its Common Characteristics table and, for a protocol driver, a list of modules to which it wants to bind, based on the BINDINGS= parameter.

After CONFIG.SYS processing completes, the Protocol Manager has a list of the active NDIS drivers, their characteristics (including entry points), and the desired bindings.

Driver Binding

The actual binding of NDIS drivers starts when some program issues the BindAndStart primitive call to the PROTMAN$ device. For all current Microsoft OS implementations, this call will come from the execution of NETBIND.EXE within a BAT or CMD file.

After receiving the BindAndStart directive, Protocol Manager will take the binding information from the module registrations and build a binding hierarchy tree. Starting at the bottom of this tree (the MAC end), the Protocol Manager works up the tree and issues an InitiateBind primitive to each protocol module that needs a driver bound on its lower boundary. As part of the InitiateBind call, the driver is passed a pointer to the Common Characteristics table of the module to be bound. The protocol driver that was instructed to initiate the bind will then issue a Bind primitive directly to the driver that it wishes to bind. When the bind completes, each driver will have a pointer to the Common Characteristics of the other, and therefore to its entry points.

After the Protocol Manager has processed all of the binding tree, all the appropriate network drivers will be bound to each other. The protocol stack is then fully operational, and the drivers can access each other by calling the dispatch entry points for communication. Figure 6 summarizes the binding process.

Figure 6. Initialization and Binding Process

A. `CONFIG.SYS` Initialization begins.

1. Protocol Manager driver does its initialization
   a. Protocol Manager reads the `PROTOCOL.INI` file and builds the Configuration Memory Image.
2. Other NDIS drivers do their initialization.
   a. Open the `PROTMAN$` device.
   b. Issue `GetProtocolManagerInfo` to gain access to ProtMan Configuration Image.
   c. Read config parameters from the Image and use them to complete initialization.
   d. Issue `RegisterModule` to register characteristics info with Protocol Manager

B. `CONFIG.SYS` processing ends.

C. Binding process starts when the `NETBIND.EXE` program opens the `PROTMAN$` device and issues a `BindAndStart` to Protocol Manager.

1. Protocol Manager builds a binding tree from `RegisterModule` info.
2. Protocol Manager starts at the bottom and calls drivers with `InitiateBind`.
   a. Each called driver issues `Bind` to the specified module to complete binding.

D. When all modules are bound, Protocol Manager returns from `BindAndStart`.

One more factor is involved in the binding process if more than one protocol is to be bound to a MAC driver. MAC drivers can only have one binding at their upper boundary. To link one MAC to multiple protocols, the Protocol Manager inserts a component, called Vector, between the MAC and the protocols (see Figure 3). Vector is part of the Protocol Manager and will be bound between the MAC and each of the protocols. To do this, the Protocol Manager first binds the MAC driver to Vector by issuing a Bind call to the MAC driver, then it issues an InitiateBind call to each of the protocols directing them to bind to a Vector entry rather than to the MAC entry.

The basic function of Vector is to route incoming packets between the protocols. When a packet is received by a MAC driver, it will issue a notification of the event to its upper boundary. When vector is involved, it will pass this notification, first to one, then to the other protocol, until one protocol accepts the packet or all have rejected it. Other functions can be passed, essentially directly, between protocols and the MAC, but this vectoring of incoming packets is key to implementing multiple protocols on one MAC.

MAC-to-Protocol Interface and Operation

The main purpose of the NDIS interface is to let the bound drivers communicate with each other. To that end, NDIS specification is largely concerned with defining a set of functions that dictate how the MAC driver will communicate with the protocol bound on its upper layer. Table 2, NDIS Primitives, lists the primitives that are defined for this MAC-to-Protocol communication. All communication between the MAC and its bound protocol will be accomplished using these primitives.

In Table 2, individual primitives are grouped into the main functional categories that they perform. In the first group are functions for the transmission of network packets from the protocol through the MAC and onto the network, and for the reception of packets in the reverse direction. In the Control group are all the functions that the protocol uses to control or modify the operation of the adapter and MAC driver. The Asynchronous Status group contains functions that the MAC uses to report events to the protocol. Finally, the Binding group has the functions used to accomplish the driver binding process. The most important of these have already been described. The remainder are extensions to allow binding and unbinding of dynamically loadable protocols. More on this subject later.

The center column of the primitive table has a symbol that indicates the direction in which the primitive calls are passed. To submit a primitive, the caller driver pushes a series of parameters on the system stack and calls an entry point in the called driver. The entry points are known to the calling driver as a result of the binding process. The Common Characteristics table, whose address was passed during binding, has Dispatch tables chained off of it. The defined entry points for the driver are in a Dispatch table.

The MAC driver's Upper Dispatch table and the addresses it contains are shown below:

MAC Upper Dispatch Table
- GeneralRequest
- TransmitChain
- TransferData
- ReceiveRelease
- IndicationOn
- IndicationOff

These are entry points that the protocol driver will call to request primitive execution by the MAC. All except GeneralRequest are called direct primitives because they serve only one primitive function. The GeneralRequest entry serves the remainder of the primitive calls, other than Binding, that are passed to the MAC. In Table 2, the GeneralRequest primitives are all the primitives in the group labeled CONTROL, except IndicateOn and IndicateOff, which have their own direct entries.

The Direct Primitives have their own entry points because they are performance-critical functions. The primitives employing the GeneralRequest entry, which are less critical, can share a common entry. The GeneralRequests identify themselves by passing an opcode as one of their parameters.

On the protocol driver side, the Protocol Lower Dispatch table defines the entry points from the MAC to the protocol. The table and its contents are as follows:

Protocol Lower Dispatch Table
- GeneralRequestConfirm
- TransmitConfirm
- ReceiveLookahead*
- IndicationComplete
- ReceiveChain*
- Status*

* denotes Indications

Table 2. NDIS Primitives

TRANSMIT AND RECEIVE		
TransmitChain	v	Initiate transmission of a frame
TransmitConfirm	^	Imply completion of frame transmit
ReceiveLookahead	^	Indicate arrival of received frame and offer lookahead data
TransferData	v	Request transfer of received frame from MAC to protocol
IndicationComplete	^	Allow protocol to do post-processing on indication
ReceiveChain	^	Indicate reception of a frame in MAC managed buffers
ReceiveRelease	v	Return frame buffer to the MAC that owns it
CONTROL		
IndicationOff	v	Disable indications from the MAC
IndicationOn	v	Enable indications from the MAC
InitiateDiagnostics	v	Start MAC runtime diagnostics
ReadErrorLog	v	Get error log info from MAC
SetStationAddress	v	Set network address of the station
OpenAdapter	v	Issue open request to network adapter
CloseAdapter	v	Issue close request to network adapter
ResetMAC	v	Reset MAC software and adapter hardware
SetPacketFilter	v	Specify filtering params for received packets
AddMulticastAddress	v	Specify multicast address for adapter
DeleteMulticastAddress	v	Remove previously added multicast address
UpdateStatistics	v	Cause MAC to update statistics counters
ClearStatistics	v	Cause MAC to clear statistics counters
InterruptRequest	v	Protocol requests later async indication from MAC
SetFunctionalAddress	v	Cause adapter to change its functional address
SetLookahead	v	Set length of visible data for ReceiveLookahead
GeneralRequestConfirmation	^	Confirm completion of previous General Request (see text for more explanation)
ASYNCHRONOUS STATUS		
RingStatus	^	Indicate a change in ring status
AdapterCheck	^	Indicate error from adapter
StartReset	^	Indicate adapter has started a reset
EndReset	^	Indicate adapter has completed reset
InterruptIndication	^	MAC response due to InterruptRequest
BINDING		
InitiateBind	p>m	Instruct a module to bind to another module
Bind	m>m	Exchange Characteristic Table info with another module
InitiatePrebind	p>m	In OS/2 dynamic bind mode, instruct a module to restart its prebind initialization
InitiateUnbind	p>m	Instruct a module to unbind from another module
Unbind	m>m	Delete linkage info with another module
GetProtocolManagerInfo	m>p	Retrieve pointer to Configuration Image
RegisterModule	m>p	Register Characteristics and Bindlist with Protocol Manager
BindAndStart	e>p	Initiate the binding process
GetProtocolManagerLinkage	m>p	Get entry point for Protocol Manager
GetProtocolIniPath	d>p	Get file path for the PROTOCOL.INI file
RegisterProtocolManagerInfo	d>p	Dynamic mode, register new Configuration Image
InitAndRegister	d>p	Dynamic mode OS/2, restart prebind initialization
UnbindAndStop	d>p	Dynamic mode, Unbind and terminate a module
BindStatus	e>p	Retrieve info on current bindings
RegisterStatus	e>p	Query if a specific module is registered

KEY		
	^	- MAC to protocol
	v	- protocol to MAC
	p>m	- Protocol Manager to driver module
	m>p	- driver module to Protocol Manager
	e>p	- ? to Protocol Manager; ? is normally an executable program
	d>p	- dynamic protocol or control program to Protocol Manager

All of these entries except Status are direct. Status is the entry for the Asynchronous Status group in the primitive table, and these primitive calls also have an opcode that identifies the type so they can share one entry. In a sense, GeneralRequestConfirm can also be viewed as a shared entry. There is only one primitive for this entry, but it is generated by the MAC at the end of any of the GeneralRequests to the MAC, and contains the request handle of the original primitive request to the MAC. Therefore, it is shared functionally in response to all of the GeneralRequest primitives.

Some of the entries in this list are marked with an asterisk to show that they are Indications. Indications are a special class of requests that imply some special handling. In implementation, they are usually associated with notifications to the protocol that are made from the MAC while it is in interrupt context. Because of this, the protocol is required to handle Indications as efficiently as possible. For example, it might put the received frame on a queue. After the protocol processes the indication, it returns control to the caller, the MAC. The MAC will enable as much interrupt processing as possible and then call IndicationComplete to give the protocol an opportunity to perform more processing on the Indication in a less critical mode (e.g., to decode the previously queued receive frame).

Transmit and Receive

The information in Table 2 identifies the basic purpose of each NDIS Primitive. Of these, transmit and receive are the key functions at the MAC-to-protocol interface level, so let's examine the primitives serving these functions in a bit more detail.

Network packets, or frames, are the data units that are transferred between the MAC and the protocol by the transmit and receive primitives. These packets include all the information, other than hardware-related functions such as preamble and checksum, that will be sent out on the network medium. As a result, the protocol, on transmit, must build the entire packet, including Data Link fields such as source and destination addresses. Likewise, on receive, the protocol must process the packet down to these levels.

The passing of packet data across the interface between the protocol and MAC is accomplished, whenever possible, by exchanging pointers to buffers or to a descriptor that in turn, points to several data buffers. The objective is to avoid unnecessary, time-consuming copying of data between buffers.

In transmit, the packet data buffers are owned by the host system and managed by the protocol driver. NDIS defines a structure, the Transmit Data Buffer Descriptor, that allows the packet data to be contained either in one buffer or in a series of chained buffers. To initiate a transmit, the protocol assembles the packet data into buffers, puts the buffer addresses in the buffer descriptor structure, and calls the MAC with the TransmitChain primitive. The primitive contains a pointer to the buffer descriptor.

The MAC has two options for processing the transmit. It will choose one or the other at its own discretion; the protocol must be capable of handling either. In the first, called synchronous transmission, the MAC copies all the packet data and returns to the protocol with a code signifying that the data buffers are free and the transmit is complete. Optionally, the MAC can return with a code signifying that the transmit is queued (this is called asynchronous transmission). It implies that the buffers are not free and the transmit has not yet completed. Later, after the MAC has copied all the transmit data, it will call the protocol with a TransmitConfirm primitive (the asynchronous response) to inform the protocol that the buffers are now free and the transmit is complete.

An additional feature available in transmit is immediate data. The protocol has the option of beginning the transmit buffers with up to 64 bytes of immediate data. This immediate data, if present, is always the first data to be transmitted. The MAC must fully process or copy this data before returning from the TransmitChain call, even if it will asynchronously process any remaining data buffers for the call. This feature allows the protocol to have a small, locally managed buffer that only needs to be valid during the TransmitChain primitive call. A protocol might use this for building packet header information, or for entire small protocol-generated packets, such as acknowledgements.

For receiving packets, the process can work in one of two ways: using ReceiveLookahead and TransferData primitives or using the ReceiveChain primitive. The method that is used is determined by the MAC, depending on how the MAC and adapter can handle data buffering. It is generally a function of whether the adapter has on-board receive buffers that use I/O or DMA to transfer the data, or whether the adapter buffers are memory-mapped and accessible directly by the host.

For buffers on the adapter that use programmed I/O or DMA to transfer the data, the reception process will use a ReceiveLookahead and TransferData pair of primitives. When the MAC has received a packet that it wants to present to the protocol, it indicates this by calling the ReceiveLookahead primitive of the protocol. The ReceiveLookahead call contains a pointer to a short portion of the data at the beginning of the packet. This usually means that the MAC must have first DMA'ed this lookahead data to a buffer in the host.

At this point, the protocol driver can examine the lookahead data to determine if it wants the packet. In some cases the packet may not be of interest to the protocol. If the packet is not needed, the protocol can return to the MAC indicating a reject and stating that the receive is complete. If the packet is needed, the protocol calls the TransferData primitive of the MAC, which results in the MAC transferring the remainder of the data to a protocol buffer.

The purpose of the Receive Lookahead implementation is to avoid unnecessary data transfers between the MAC and the protocol. This technique improves the efficiency of the network stack.

If the adapter has receive buffers that are accessible as host memory, receive will be implemented with the ReceiveChain primitive. For this type of buffering, the MAC will own and manage the receive buffers. For flexibility, this mode has a ReceiveChain buffer descriptor structure, similar to the transmit structure, that lets multiple separate buffers be joined for one packet transfer. When the MAC has a received packet to present to the protocol, it builds a buffer descriptor for the packet and calls the protocol with ReceiveChain.

When the protocol gets the ReceiveChain Indication, it has two options. In the simplest case, the protocol can copy all of the packet data and return to the MAC specifying that the receive is complete and the buffers are free. In the other case, the protocol can defer copying the buffers and return to the MAC specifying that the buffers are still in use. The protocol will later complete the copying of the buffers and then call the MAC with a ReceiveRelease primitive to indicate that the buffers are now free and the receive is done.

In all of these receive scenarios, the primitive calls issued to the protocol are indications. This means that the protocol drivers need to observe certain rules and that the MAC must issue an IndicationComplete call to the protocol as part of the process. For more information about these indication issues, refer to the NDIS specification document.

Dynamic Binding

As mentioned earlier, some primitives are provided to support Dynamic Binding. Dynamic Binding is a new concept that has been added in Version 2.0.1 of the NDIS document. It allows a protocol to be added to or removed from an existing network configuration after the initialization process has completed. The dynamic protocol driver must be written for this purpose and normally will be implemented as a TSR or transient program module.

The main advantage of Dynamic Binding is freeing system memory until it is actually needed for a particular protocol. This is most useful in the DOS environment where there is a 640K memory limit and it is difficult to have multiple protocol stacks loaded simultaneously.

DOS Versus OS/2

NDIS has all the features needed to allow writing network drivers that will run in either the DOS or the OS/2 environment. A driver can only be used with one of these operating systems (the same driver can't be used for both), but the structure of the driver can be identical for both environments. We have found that one set of

source code can be used to make versions for both DOS and OS/2. Where different techniques are required, a small piece of code can be selected via conditional compile or assembly statements for the two versions. An example of such a difference is that a certain call might need to use Interrupt 21h in a DOS-based driver, versus an IOCTL call for OS/2. A conditional selection of a few lines of code can implement one or the other. The make process for the driver will select the correct environment.

Network device drivers are not very different in structure from other types of device drivers. The device driver must be written to conform to the architecture in which it will run—DOS or OS/2. All of the normal issues apply in writing NDIS drivers for both of these environments. Several books and articles are available that explain general driver development issues. See the list of references at the end of this article for suggested reading.

Summary

One of the main goals of the Network Driver Interface Specification is to save network software developers from "reinventing the wheel" for each new version of network adapter hardware. A protocol that is written with an NDIS interface at its base should be able to function unchanged with many different types of adapter hardware. In addition, the manufacturers of the network hardware should be in the best position to write efficient and bug-free MAC drivers for their own boards. 3Com and many other vendors have NDIS MAC drivers available to support their network hardware.

Using the standardized NDIS interface allows a new level of sharing of network resources in a machine. Multiple protocols and multiple hardware adapters can coexist—even those from different vendors. Input was sought from many leaders in the network industry to guarantee that the specification is flexible enough to meet most networking needs. Further, 3Com carefully defined the functions so that performance was not sacrificed for this flexibility.

If you are a software developer who would like to evaluate NDIS for your specific needs, the next step is to obtain a copy of the Network Driver Interface Specification. The NDIS document can be obtained from 3Com by writing to the following address:

3Com Corporation
Network Adapter Division
Software Product Marketing
5400 Bayfront Plaza
P.O. Box 58145
Santa Clara, CA, 95052-8145 ■

Rex Allers is a Systems Engineer in the Technical Services Organization at 3Com, specializing in support for developers. Rex has worked in engineering and support in the computer industry for longer than he cares to admit, and has been at 3Com since 1986.

Suggested Reading:

1. *Microsoft/3Com LAN Manager Network Driver Interface Specification*, 3Com/Microsoft, 1990

2. *The Open Book*, Marshall Rose, Prentice Hall, 1990

3. *Advanced MS-DOS*, Ray Duncan, Microsoft Press, 1986

4. *Writing MS-DOS Device Drivers*, Robert S. Lai, Addison-Wesley, 1987

5. *OS/2 Programmer's Guide*, Ed Iacobucci, McGraw-Hill, 1988

6. *Writing OS/2 Device Drivers*, Raymond Westwater, Addison-Wesley, 1989