Microsoft/3Com LAN Manager Network Driver Interface Specification

Version 1.0.2 Preliminary Draft

3Com Corporation
3165 Kifer Road
Santa Clara, California 95052-8145
Printed in the U.S.A.

Manual Part No. 6406-00
Published January 1989

Disclaimer

Contents

Chapter 5: Specification of Primitives

Chapter 1: Introduction

This document describes the LAN Manager driver architecture and interfaces that let a DOS 3 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 .

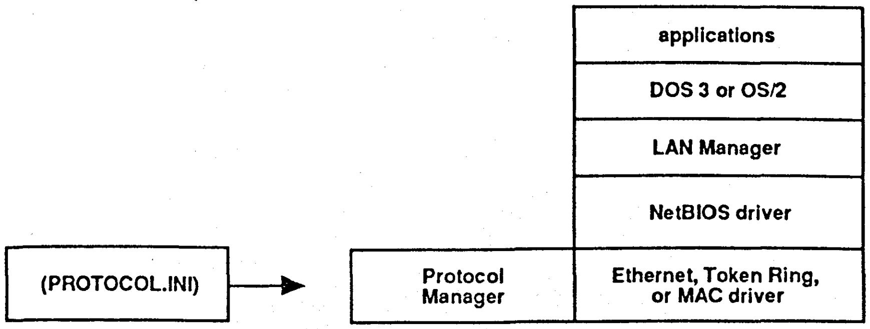
Definition of Terms

To simplify the job of supporting multiple adapters and protocols, the architecture defines three 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.

- Protocol drivers, which provide higher-level communication services from data link to session (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.

- 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.

3. The software interface provided specifically by NetBIOS drivers that will work with LAN Manager.

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

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.

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 "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 should 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).

The configuration and binding steps and API calls are defined in the supplement, "Protocol Manager Binding: NETBIND and the PROTOCOL.INI file."

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 are calls from the Protocol Manager to a MAC or protocol module. Their purpose is to support module binding and management functions.

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.

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. Static information which needs to be referenced frequently (such as entry points) should be copied into the local data area for efficient access. This information should not be copied prior to the Bind call and should 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 generally 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 and provide hints.

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)
WORD	Level of common characteristics table (zero this version)
WORD	Level of service-specific subtables (zero for MAC and NetBIOS definitions in the following sections)
BYTE	Major Module Version (2 BCD digits)
BYTE	Minor Module Version (2 BCD digits)
DWORD	Module function flags, a bit mask (hints only): 0 - Binding at upper boundary supported 1 - Binding at lower boundary supported 2-31 - 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 -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 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 Ring0 selectors. This field is usually set at Ring 3 INIT time even though the selector set in must be Ring 0 (obtained from the device header filled in byOS/2).

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 retrieve this information.

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, (provided as hints only): 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
	15-31 - Reserved, must be zero
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	Interrupt level used by adapter

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

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.

In interpreting these tables the implementer should 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 should not prevent the protocol from accepting a bind for the MAC.

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

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 frame; 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 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.

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

If a MAC does not support loopback (bit 7 is set) a protocol must handle this function itself. In other words, if the source and destination addresses of a frame are the same, or the frame is a broadcast frame or multicast frame to a local multicast address, then the protocol must handle the loopback delivery of the frame. MAC's that support loopback must do so either by actually transmitting the frame to the media, or via a mechanism that lets the receive indication be done at interrupt time. MAC's should support loopback if at all possible. Loopback support will substantially improve the performance of some protocols (particularly DLC).

MAC Service-specific Status Table

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) 5-31 - 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 (0xFFFFFFFF means not kept):

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
DWORD	Frames with CRC error
DWORD	Total bytes received
DWORD	Frames discarded - no buffer space
DWORD	Multicast frames received
DWORD	Broadcast frames received
DWORD	Frames received with errors
DWORD	Frames exceeding maximum size
DWORD	Frames smaller than minimum size
DWORD	Multicast bytes received

DWORD	Broadcast bytes received
DWORD	Frames discarded - hardware error
DWORD	Total frames transmitted
DWORD	Total bytes transmitted
DWORD	Multicast frames transmitted
DWORD	Broadcast frames transmitted
DWORD	Broadcast bytes transmitted
DWORD	Multicast bytes transmitted
DWORD	Frames not transmitted - time-out
DWORD	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. When updating these counters, a frame is counted in all the supported counters that apply.

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 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)
WORD	Level of common characteristics table: 0
WORD	Level of service-specific subtables: 0
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 (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 (LANMAN NetBIOS Linkage table):

WORD	Bytes of data returned in this table
WORD	Bytes of data actually available
BYTE	LANA number
WORD	Net driver type (1 = NCB)
WORD	Network Status: Bit 0: Reserved, must be zero Bit 1: Cleared = normal driver Set = loopback driver Bit 2-15: Reserved, must be zero
DWORD	Network bandwidth (bits/s)
WORD	Maximum sessions
WORD	Maximum number of NCB's
WORD	Maximum number of names
WORD	NetBIOS driver's DS value
DWORD	NetBIOS NCB Handler address
BYTE	Number of commands in OEM extension table

OemExtTable:

WORD	ExtStruct
WORD	ExtStruct

where

ExtStruct structure:

Cmd	BYTE Extended NCB Command Opcode Value		
CmdInfo	WORD Command descriptor bits		
	bit	0 == 1:	standard buffer used
		1 == 1:	second buffer used
		2 == 1:	lock buffers
		3 == 1:	asynchronous option allowed
		4 == 1:	command can be cancelled
		6/7 ==	1: if regular command
			2: if privileged command
			3: if exclusive command
		8 == 1:	Uses LSN field
		9 == 1:	Uses NamNum field
		10 == 1:	Uses Local Name Field (ncb_name)
		11 == 0:	Buffer segments must be read-write
		1:	Buffer segments may be read-only
		12-15 == 0:	Reserved, must be 0

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 may 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.

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 8
```

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      ;Physical 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.

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:

```
WORDTDDataCount; Count of transfer data blocks; max is 8
```

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      ;Physical address of data block
```

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 8
```

Followed by RxDataCount instances of:

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

For received frames that are larger than 256 bytes, the first data block of the frame must be at least 256 bytes long.

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 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.

- 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 should 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 protocol module. In the latter case, the Protocol Manager by default will ask the one 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.

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 the image is only available during device-driver initialization time.

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 the Ring 3 INIT time 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 the Ring 3 INIT time 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 the Ring 3 INITtime 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 the Ring 3 INIT time 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 to31 bit values 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.

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. A value of 0 in this field is equivalent to passing a NULL bindingslist pointer in the Register Module command.

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.

Chapter 5: Specification of Primitives

MAC 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 should run (as much as possible) with interrupts enabled. Interrupt handlers should dismiss the interrupt at the 8259 as soon as possible.
- An indication handler will normally be entered with interrupts enabled but an indication handler should not enable interrupts. The handler may disable interrupts if it chooses and on return the MAC must assume that interrupts may have been disabled. 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.
- An indication handler may only issue TransmitChain and TransferData primitives. Any MAC interface request may be issued from a confirmation or IndicationComplete handler.
- 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. This means they must usually provide their own stack.
- A confirmation handler may be entered with the confirmation for a request before the request has returned.
- 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 0GDT 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 should return these specific codes. However it is acceptable to return GENERAL_FAILURE for any non-recoverable failure.

- Parameters are passed on the stack compatible with Microsoft C FAR Pascal calling conventions. On entry to this 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 should follow standard Microsoft C conventions about saving "register variable" SIDI registers if these are used. The direction bit is assumed to be clear on entry and must be clear upon exit.

- The function returns in AX a return code specifying the status of function invocation.

- 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.

- The number of Request commands which can be simultaneously queued by the MAC should be configurable. The suggested keyword in the configuration file is "MaxRequests." If it is not configurable then the minimum value for this parameter is 6.

- The number ofTransmitChain commands which can be simultaneously queued by the MAC must be configurable. The suggested keyword in the configuration file is "MaxTransmits". The suggested minimum default value for this parameter is 6.

- 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.

- For performance reasons, frame data buffers used for transmission and reception should be word aligned.

- The MAC header is passed protocol-to-MAC or MAC-to-protocol in exactly the format in which it exists on the medium. The protocol should convert header fields found in the header buffer passed up to whatever format is required to conveniently store the mind 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).

- 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	Unique handle assigned by the protocol to identify this request. If asynchronous request is being done, the MAC will return this ReqHandle on the completion call it later makes to indicate completion. A ReqHandle of 0 indicates that the completion routine should be unconditionally suppressed.
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
	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 should 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.

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
	0x000A	HARDWARE_ERROR

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.

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.

The MAC implicitly disables indications (IndicationOff) before calling Receive Lookahead, and by default will re-enable them on return from this routine. 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 desc
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 transfershould 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. TransferData may be called only once per ReceiveLookahead indication. Subsequent calls within the same indication will return an error.

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 eventually after 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 should minimize stack usage before calling this routine and, under DOS, should 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.

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. This requirement 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, and by default will re-enable them on return from this routine. 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 should 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. Additionally a protocol may not call IndicationOff from within its ReceiveLookahead, ReceiveChain or Status indication handler.

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.

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 may use this call to re-enable indications after having disabled them. Note that a MAC may optionally defer the actual re-enabling of indications. Additionally a protocol may not call IndicationOn from within its ReceiveLookahead, ReceiveChain or Status indication handler.

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 called is guaranteed to have completed.

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
	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 may return an error if it does not support run time diagnostics.

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. This command can be called as often as desired. Each time it replaces a current station address buffer internally maintained by the MAC 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 a default address either read out of hardware or obtained from the PROTOCOL.INI file if not available from hardware. That initial address will be maintained in the permanent station address field of the MAC service-specific 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.

On hardware which supports OpenAdapter, SetStationAddress may not take effect while the adapter is in a closed state.

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   0            ; Pad parameter - must be 0
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
	0x00FF	GENERAL_FAILURE

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.

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 priority 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
            0x0002      REQUEST_QUEUED
            0x0006      OUT_OF_RESOURCE
            0x0007      INVALID_PARAMETER
            0x0008      INVALID_FUNCTION
            0x0009      NOT_SUPPORTED
            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. The MAC must preserve the current station address, LOOKAHEAD length, packet filter, multicast address list and functional address. The MAC also enables indications.

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 should be the same ones in effect after the reset.

When the reset is initiated, the MAC should 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 should subsequently issue an IndicationComplete when the reset is complete. During the time between the StartReset indication and reset IndicationComplete the MAC should generate INVALID_FUNCTION for any Requests it cannot handle while the reset is in progress. The IndicationComplete notifies the protocol that the MAC can continue handling Requests.

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 function implies to the MAC the kind of packets it should allow indications to be generated. A packet filter of 0 indicates that the MAC should not indicate received packets.

NOTE: The packet filter 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 does not support the receiving packets of the type specified it will return GENERAL_FAILURE. In this case the packet filter is left in its previous state.

If this Request returns SUCCESS the hardware is enabled to receive the types of packets requested and will generate Indications to the protocol for those types of packets. Filtering implied by the bits in Filtermask is optional. For example, a protocol may receive broadcast packets even if bit 1 is not set in the filter.

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 (INVALID_FUNCTION error) 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 error) if they already exist no matter what their type. They must be explicitly deleted.

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.

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 counters in its characteristics table. The requester can read that table when this operation completes. If the statistics table are always current this function should 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 should be reset to zero in an atomic operation.

Interrupt

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 should be implemented if at all possible. Interrupt request will substantially mprove 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.

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. 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.

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.

Status Indication

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  Indication
```

Returns:	0x0000	SUCCESS

Description:

Called by 802.5-style drivers to indicate a change in ring status. The status codes can be found in the IBM Token Ring Technical Reference Manual.

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  Indication
```

Returns:	0x0000	SUCCESS

Description:

Called to indicate an adapter error. This is to be considered fatal. For 802.5, error definitions are defined in the IBM Token Ring Technical Reference Manual. Note that these definitions are merely adhered to for ease of DLC implementation; MAC's other than 802.5 can generate this indication.

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  Indication
```

Returns: 0x0000 SUCCESS

Description:

Called to indicate that adapter has started a reset. This function always returns SUCCESS.

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
	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 should issue a Bind call to. LastBind is used to indicate the last Initiate Bind request so the module may perform any final initialization prior to returning. If a module other than a MAC does not have lower bindings, 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 should also fail returning the same return code as the failing Bind call.

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
	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.

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.

GetProtocolManagerInfo

Purpose: Retrieve pointer to configuration image.

Opcode	- 1
Status	- On return contains request status
Pointer1	- On return contains a FAR pointer to the structured memory image representing the parsed user configuration file PROTOCOL.INI. For OS/2 the selector of the pointer returned here is valid only at device INIT time. This pointer cannot be used later. 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. (1.0 for this specification)

Returns:	0x0000	SUCCESS
	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.

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 should 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.
Word1	-Unused

Returns:	0x0000	SUCCESS
	0x00FF	GENERAL_FAILURE

Description:

This request is used by a driver to identify one of its contained modules to the Protocol Manager. After calling register module, a driver must remain installed and respond to system requests. This register 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 binding information is later used by the Protocol Manager to determine necessary sequence of InitiateBind commands to issue.

A driver which contains multiple modules can call Register Module 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 Register Module request.

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
	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 an AUTOEXEC.BAT or STARTUP.CMD 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.

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
		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 Ring 0 drivers to the dispatch entry point.

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 PROTOMAN.DOS. For OS/2 the file is PROTMAN.OS2. This device header name for the Protocol Manager device driver is PROTMAN$.

For OS/2 and DOS the PROTOCOL.INI file is read at INIT time by the Protocol Manager and parsed into a memory image.

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.

Binding Sequence

The Protocol Manager works from the bottom to the top of the protocol hierarchy, telling each upper module to bind to the appropriate lower module. The command used to do this is called InitiateBind. In response, the upper module initiates a Bind command to the specified lower module, which serves to exchange characteristics tables directly between the two modules.

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. Use the GetProtocolManagerInfo call 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.

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.

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

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

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 MAC's 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 MAC's 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 the characteristics table of the associated MAC, substituting the VECTOR entry points for the real MAC's entry points. The protocol then starts itself and returns from InitiateBind.

5. The Protocol Manager then issues subsequent Initiatebind 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.

Vector Demultiplexing

The Vector dispatches incoming frames to protocol stacks 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 frame
- Protocols that handle LLC frames with specific LSAP's
- Protocols that handle LLC frames with non-specific LSAP's

The Vector will poll protocols in that order (and within that, in the order they Registered) until it finds one that will accept the frame.

Appendix A: System Return Codes

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 may 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 may 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.

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 bind-time 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.

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.

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

The 802.3 media specific statistics structure is defined as follows:

WORD	Length of 802.3 Statistics structure, including this field
WORD	802.3 Statistics structure version level (1)
DWORD	Frames with alignment error
DWORD	Receive error failure mask
DWORD	Frames with overrun error
DWORD	Frames transmitted after at least 1 collision
DWORD	Frames transmitted after deferring
DWORD	Frames not transmitted - max (16) collisions
DWORD	Total collision during transmission attempts
DWORD	Late (out of window) collisions
DWORD	Frames transmitted after exactly 1 collision
DWORD	Frames transmitted after multiple collisions
DWORD	Frames transmitted, CD heartbeat
DWORD	Jabber errors
DWORD	Carrier sense lost during transmission
DWORD	Transmit error failure mask

The 802.3 failure masks are defined as follows:

Receive error failure mask:
- Bit 0 = CRC error
- Bit 1 = Framing error
- Bit 2 = Frame size exceeds maximum
- Bit 3-31 reserved, must be zero

Transmit error failure mask:
- Bit 0 = Excessive collisions error
- Bit 1 = Carrier check failed error (SQE test failed, CD heartbeat)
- Bit 2 = Short circuit error
- Bit 3 = Open circuit error
- Bit 4 = Frame too long error
- Bit 5 = Remote failure to defer error (out of window collision)
- Bit 6-31 reserved, must be zero

On initialization the failure masks are 0. A bit is set the first time the corresponding error occurs.

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

Examples:

(a) A 'Receive frame with a CRC error' is counted in all the the following statistics counters :

- Total Frames received,
- Frames with CRC errors, and
- Frames with error counters.

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

- Total Frames transmitted,
- Frames transmitted with one or more collision, and
- Frames transmitted with only one collision.

Appendix D: OS/2 NETBIOS Device Driver Interface

Introduction

NetBIOS drivers install as standard OS/2 device drivers. All programming considerations for device drivers apply to NetBIOS drivers, with the additions noted herein.

Because NetBIOS drivers get called by the operating system to do I/O, a direct Ring 0 linkage mechanism has been defined. This allows OS/2 to request network I/O services via a direct call to the NetBIOS driver, as opposed to going through standard device driver interfaces.

OS/2 performs a standard DosDevIoctl to the NetBIOS driver to get the far address of the direct I/O routine. This routine is analogous to a real mode INT 5C handler-i.e., it gets called with a pointer to an NCB. All NCB's passed to this routine contain physical buffer addresses only and the NCB and buffers have already been locked if appropriate. A convention is defined to support OEM-defined NCB's such that the operating system can still perform appropriate locking on behalf of the extended NCB's.

To support NetBIOS drivers, the operating system provides two standardized routines, PhysNCBDone and VirtNCBDone, for handling NCB completion. NetBIOS drivers need not process synchronous or asynchronous NetBIOS options; they just call an NCBDone routine when processing is complete. The linkage to the NCBDone routines is defined below.

We recommend that NetBIOS drivers for LanMan support a configurable "call" time-out. This parameter specifies the maximum amount of time that the driver will wait for a response from the remote machine when handling Call, AddName, and AddGroupName NCB's. The parameter should be settable from the DEVICE= line in the CONFIG.SYS file.

NetBIOS Driver Configuration

NetBIOS drivers are normal OS/2 device drivers and are installed by inserting a DEVICE= statement in CONFIG.SYS. Note that NetBIOS drivers depend on the Microsoft-supplied NETWKSTA.SYS driver. This must be installed AFTER all NetBIOS drivers.

Multiple NetBIOS drivers can be installed in a single system. Each is described by a set of parameters in a configuration file, LANMAN.INI, as follows:

```
[networks]
net1 = toknet$,0,LM10,32,32,16
net2 = ethnet$,1,LM10,32,32,16
```

The format of each NetBIOS network definition is:

<network name> = <DN>,<LN>,<type>,<NCB's>,<Sess>,<Name>

<DN>	= OS/2 driver name	Required
<LN>	= LANA Number	Not required, default is 0
<type>	= driver type	Not required, default is "LM10"
<NCB's>	= Number NCB's	Not required, default is driver default
<Sess>	= Number Sessions	Not required, default is driver default
<Name>	= Number Names	Not required, default is driver default

When the workstation is initialized, it will initialize each driver using the <NCB's>, <Sess>, and <Name> parameters. This configuration information is processed by the NETWKSTA.SYS driver in support of the protected mode NetBIOS API. It allows applications to open NetBIOS drivers by logical name, independent of the actual OS/2 driver name and LANA number. The NETWKSTA.SYS driver also handles OS/2 related processing that is common to all NetBIOS drivers, including translation of NCB's from virtual to physical form and synchronization between interrupt time and task time asynchronous NCB completion.

NetBIOS Driver Initialization

Network drivers perform device initialization at load time just like any other device drivers. They must also support a DosDevIoctl command that is issued from OS/2 to the NetBIOS driver during system initialization. The driver initialization sequence is:

1. Respond to the device Init command at driver load time by allocating packet buffers, setting up tables, and running startup diagnostics as appropriate.

   Note that GDT selectors (allocated via the AllocGDTSelector Device Help) are not valid to use during initialization. Any access to memory that normally would be done via an allocated GDT selector must be done via PhysToVirt.

2. Respond to the sequence DosOpen, DosDevIoctl: NetBIOSLinkage, DosClose. Via this mechanism, the redirector passes in the entry points of PhysNCBDone and VirtNCBDone routines and the lana number requested. The driver is responsible for filling in a table parameters that the redirector needs.

Note that a driver must be prepared to respond to multiple initialization sequences of DosOpen/DosDevIoctl/DosClose. For a driver supporting multiple LANA's, this would happen once for each LANA defined to the system.

NetBIOSLinkage: DosDevIoctl Category 81, Subfunction 62

Purpose: Get NetBIOS driver linkage table.

Request packet 13-BYTE Request header

BYTE	Function category = 81h
BYTE	Function code = 62h
DWORD	Parameter buffer address
DWORD	Data buffer address (returned)

Parameter Packet

WORD	Parameter Packet length in Bytes
WORD	Data Packet length in Bytes
DWORD	NCBDone handler address NOTE: This address is only valid after completion of CONFIG.SYS processing by OS/2. Hence it may not be called until the first NCB has been received through the NetBIOS NCB Handler entry point.
BYTE	LANA Number (as specified in the NetBIOS network definition in LANMAN.INI)
BYTE	Pad
DWORD	VirtNCBDone handler address NOTE: This address is only valid after completion of CONFIG.SYS processing by OS/2. Hence it may not be called until the first NCB has been received through the NetBIOS NCB Handler entry point.

The driver can determine if the given version of NETWKSTA.SYS supports the VirtNCBDone entry point by checking the Parameter Packet length. If the length is less than 14 bytes, the VirtNCBDone entry point is not available.

The data buffer is filled in by the NetBIOS driver to contain the following NetBIOSLinkage structure:

NetBIOSLinkage structure:

WORD	Bytes of data returned in this table
WORD	Bytes of data actually available
BYTE	Reserved do not change.
WORD	Net driver type (1=NCB, 2=MCB)
WORD	Network Status: Bit 0: Reserved, must be zero Bit 1: Cleared = normal driver Set = loopback driver Bit 2-15: Reserved, must be zero
DWORD	Network bandwidth (bits/s)
WORD	Maximum sessions
WORD	Maximum number of NCB's
WORD	Maximum number of names
WORD	NetBIOS driver's DS value
DWORD	NetBIOS NCB Handler address
BYTE	Number of commands in OEM extension table

OemExtTable:

```
WORD        ExtStruct
...
WORD        ExtStruct
```

where

ExtStruct structure:

```
Cmd         BYTE        Extended NCB Command Opcode Value
CmdInfo     WORD        Command descriptor bits
                            bit   0 ==       1:   standard buffer used
                                  1 ==       1:   second buffer used
                                  2 ==       1:   lock buffers
                                  3 ==       1:   async option allowed
                                  4 ==       1:   command is cancelable
                                  6/7 ==     0:   invalid encoding
                                             1:   if regular command
                                             2:   if privileged command
                                             3:   if exclusive command
                                  8 ==       1:   Uses LSN field
                                  9 ==       1:   Uses NamNum field
                                  10 ==      1:   Uses Local Name Field (ncb_name)
                                  11 ==      0:   Buffer segments must be read-write.
                                             1:   Buffer segments may be read-only.
                                  12-15 ==   0:   Reserved must be zero
```

The OemExtTable is optional (can have 0 entries). It defines NCB commands that are OEM-defined extensions to the standard NetBIOS interface. The CmdInfo parameter describes the extended NCB's usage of buffers and asynchronous notification. It also indicates whether the command is cancelable and whether the requesting application needs to have opened the NetBIOS driver in privileged or exclusive mode to issue the command. The operating system needs this information to provide correct addressing and protection in making application-level buffers available to the NetBIOS driver in physical address form. The highest three bits are used to protect Server and Redir Sessions and Names from interference from user submitted NCB's.

NetBIOS NCB Handler

The NetBIOS NCB handler is entered via a far call. The calling conventions are essentially identical to those for real-mode INT 5C, except that ES:BX is the virtual address of the NCB to be executed. The physical address of the NCB is available in DX:AX.

Entry:	ES:BX	Virtual address of NCB
	DX:AX	Physical address of NCB
	DS	NetBIOS driver's DS
	TOS:DWORD	Return address
	TOS+4:DWORD	Virtual address of NCB
	TOS+8:DWORD	Physical address of NCB
Return:	AX	NetBIOS-defined result code
Errors:		NetBIOS-defined immediate result codes

The virtual address of the NCB (passed in ES:BX and TOS+4) will be a GDT based address and will be accessible by the driver (both at task time and interrupt time) until driver has completed the NCB (or rejected it via the immediate return code).

Parameters are passed both in registers and on the stack. The NCB Handler must always exit via a far return, with parameters removed from the stack and an immediate NCB return code in AX. These conventions allow the handler to be coded in high level languages.

Note that if the immediate result code returned is non-zero, the NCB is considered to be completed, an NCBDone routine must NOT be called.

The addresses within the NCB are physical addresses that are guaranteed to be valid until the invocation of PhysNCBDone or VirtNCBDone. OS/2 will handle locking and unlocking of these data regions.

NOTE: When transferring blocks of data via REP instructions, a maximum size of 2k is allowed. See the OS/2 Device driver guidelines.

The values of the NCB_POST and the NCB_CMD_CPLT fields are reserved and must not be modified during the course of processing

The NetBIOS NCB handler should not distinguish between synchronous and asynchronous NCB's. It need just start the necessary I/O going and return. NCB completion and synchronization is handled by a Microsoft supplied routine, NCBDone.

NOTE: The NCB Handler MUST NOT block at any time.

The interrupt routine must issue a far call to either PhysNCBDone or VirtNCBDone upon completing processing of an NCB (usually at Interrupt time). The address of the NCB being completed must be passed to the given NCBDone routine on the stack (the far addresses of PhysNCBDone and VirtNCBDone are passed to the driver via initialization-time DosDevIoctl described in this appendix, "NetBIOS Driver Initialization"). Entry conditions for PhysNCBDone and VirtNCBDone:

1. Mode = ProtMode. See DevHlp GoProt and GoReal documentation.

2. SS == SS at driver entry point (either interrupt handler or at task time). (Note: if GoProt is used in an interrupt handler, SS == SS upon return from GoProt.)

3. Entry for PhysNCBDone

TOS:DWORD	Return address
TOS+4:DWORD	Physical address of NCB

   Entry for VirtNCBDone

TOS:DWORD	Return address
TOS+4:DWORD	Virtual address of NCB as passed to the drivers NCB handler in ES:BX.

4. The NCB_RETCODE field has the correct Return Code value.

5. The Following fields must contain the values present at NCB submission time: NCB_POST, NCB_CMD_CPLT.

6. All NetBIOS processing of this NCB must be complete. The internals of NCBDone reserve the right to modify the NCB in any way.

Common Problems and Hints

- For performance reasons, it is recommended that PhysToGDT be used whenever possible instead of PhysToVirt.
- When a Reset NCB is received, the driver MUST complete (call PhysNCBDone or VirtNCBDone) all outstanding NCB's with a canceled or name deleted status. It is recommended that for the driver's own protection, the driver should not accept any NCB's while a reset is in progress. The handler should complete all other outstanding NCB's before completing the reset NCB.
- The NCB Handler MUST NOT block at any time.
- When a session is aborted (as opposed to hung-up), the driver should be return error code 18H (Session Ended Abnormally). The error code 0AH (Session Closed) should only be returned when either end hangs up the session. The redirector needs this information for its session handling.
- The virtual address passed (on the stack) to VirtNCBDone must be the SAME address that was passed to the drivers NCB handler in ES:BX (and on the stack).
- The driver may use either PhysNCBDone and VirtNCBDone, but must only call ONE of the routines for any given NCB.