Novell ODI Specification: 16-Bit DOS Client HSMs

(Intel 80x86 Assembly Language)

HSM Specification Version 4.00
Document Version 4.03
Part Number: 107-000054-001
February 2, 1996

Disclaimer

Novell, Inc. makes no representations or warranties with respect to the contents or use of this manual, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc. reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes.

Novell, Inc.
122 East 1700 South
Provo, Utah 84606

Trademarks

Novell has made every effort to supply trademark information about company names, products, and services mentioned in this document. Trademarks indicated below were derived from various sources.

Novell and NetWare are registered trademarks of Novell, Inc.

Internetwork Packet Exchange, ODI, Open Data-Link Interface, LSL, Link Support Layer, MLID, Multiple Link Interface Driver, MLI, Multiple Link Interface, MPI, Multiple Protocol Interface, MSM, Media Support Module, TSM, Topology Support Module, HSM, Hardware Support Module, RX-Net, NE1000, NE2000, NE/2, NE2-32, and NTR2000 are trademarks of Novell, Inc.

Table Of Contents

List of Tables

List of Figures

Preface

This document tells you how to use the LAN driver toolkit to develop the Hardware Specific Module (HSM) of a DOS client MLID that conforms to Novell's® Open Data-Link Interface™ (ODI™) specification. ODI allows multiple protocols to operate in the NetWare® v3.1*x* (and higher), DOS, and OS/2 environments. Writing a LAN driver that conforms to the ODI specification ensures compatibility with any protocol that is also written to the ODI specification (for example, TCP/IP, ISO, IPX, etc.).

The ODI specification provides many powerful features. Because of this complexity, we have provided a LAN driver toolkit to help in LAN driver development. This toolkit provides two modules, the Media Support Module™ (MSM™) and the Topology Specific Module™ (TSM™). These modules contain some of the major LAN driver portions that the developer previously wrote. This means that you (the developer) are now responsible only for that portion of the LAN driver that is dependent upon your specific LAN adapter. This module is called the Hardware Specific Module™ (HSM™). The MSM and TSM allow the portion of the LAN driver that you write (the HSM) to be as simple as possible and still retain all the features the ODI specification requires. Linked together, these three modules (MSM, TSM, and HSM) comprise an ODI LAN driver (also referred to in this document as an MLID™, Multiple Link Interface Driver™).

The HSM interfaces with the MSM and TSM. However, the HSM can also interface directly with the Link Support Layer™ (LSL™) when the need arises. The majority of developers will find the MSM and TSM adequate for their needs. However, some developers might find they need to modify the MSM and/or TSM to better suit their particular needs. In either case, we strongly recommend using the LAN driver toolkit because it greatly simplifies the task of writing a driver.

Note Unless otherwise specified, all references in this document to the MSM include both the MSM and the TSM. The developer's kit contains sample source code for the HSM. We recommend that you study this code before writing your HSM. ▲

Document Organization

You can use the following table to help locate the information you need. If you cannot find the subject or term you are looking for, please check the index.

If you want to know about:	Look in:	
Assembling and linking	Appendix A	Creating the DOS ODI LAN driver
Custom configuration keywords	Appendix B	The NET.CFG Configuration File
Data structures and variables	Chapter 4	HSM Data Structures and Variables
General theory		
HSM	Chapter 2	Designing the DOS ODI HSM
	Chapter 3	DOS ODI HSM Overview
ODI	Chapter 1	Introduction to ODI
Routines the HSM must provide	Chapter 5	Developer-Written HSM Routines
Support routines		
LSL support routines	Chapter 7	Support Routines Provided by the LSL
MSM support routines	Chapter 6	Support Routines Provided by the MSM
Supporting PCMCIA boards	Appendix C	Supporting PCMCIA Boards

Supplementary Documents Referenced

This document refers to the following ODI Specification Supplements:

The MLID Installation Information File
Part number 107-000056-001
Source Routing
Part number 107-000058-001
Canonical and Noncanonical Addressing
Part number 107-000059-001
Frame Types and Protocol IDs
Part number 107-000055-001

This document also refers to the *NESL Specification: 16-Bit DOS Programmer's Interface.*

Prerequisites

The developer should be experienced with assembly language programming for the Intel family of microprocessors and have a sound understanding of event-driven systems, interrupt-driven drivers and DOS device driver development.

Manual Conventions

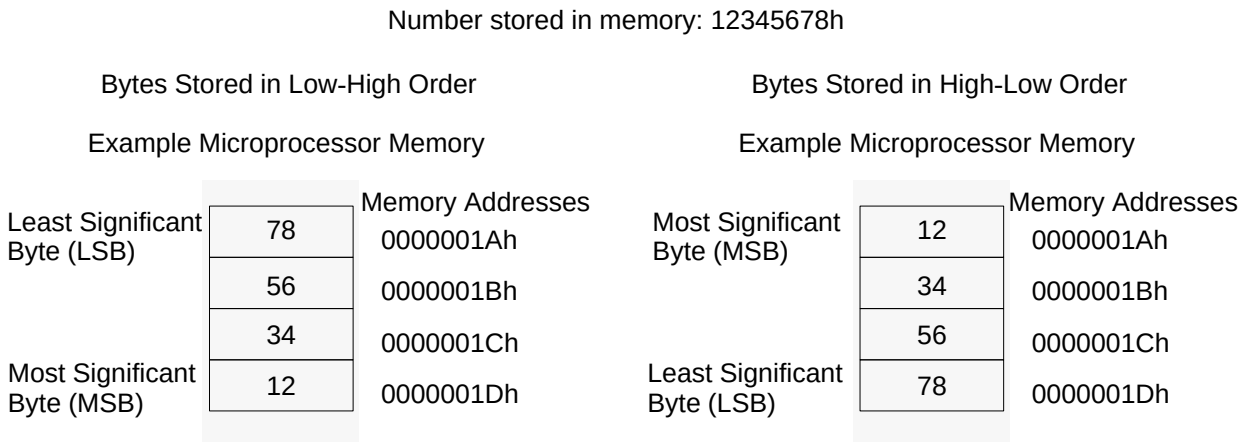
All numbers in this document are decimal unless otherwise specified. Hexadecimal numbers are identified by a trailing 'h' (for example, 0FFh). Where bit fields within a byte are specified, bit 0 is assumed to be the low-order bit.

The following data types are defined:

byte	1-byte unsigned integer
char	1-byte ASCII character
offset	4-byte near offset of an Intel 80386/80486 address

Numeric fields composed of more than one byte can be in one of two formats: high-low or low-high. High-low numbers contain the most significant byte in the first byte (the byte with the lowest address) of the field, the next most significant byte in the second byte, and so on, with the least significant byte appearing last (in the highest address). Low-high numbers are stored in the opposite order. The Intel 80X86 microprocessors store numbers in low-high order. See Figure Prf.1.

Figure Prf.1 How Bytes Are Stored in Memory



Names of procedures referred to in the text and set in *italics* are listed in the Table of Contents.

❑

Chapter 1 Introduction to ODI

Chapter Overview

This chapter briefly describes the Open Data-Link Interface™ (ODI™) specification. It describes the functions of Multiple Link Interface Drivers, protocol stacks, and the LSL. This chapter also contains a brief description of data flow through the ODI model.

Because the ODI specification provides for communications between a variety of protocols and media, LAN drivers are called *Multiple Link Interface Drivers* (*MLIDs*). The Link Support Layer™ (LSL™) handles the transfer of information between MLIDs and protocol stacks.

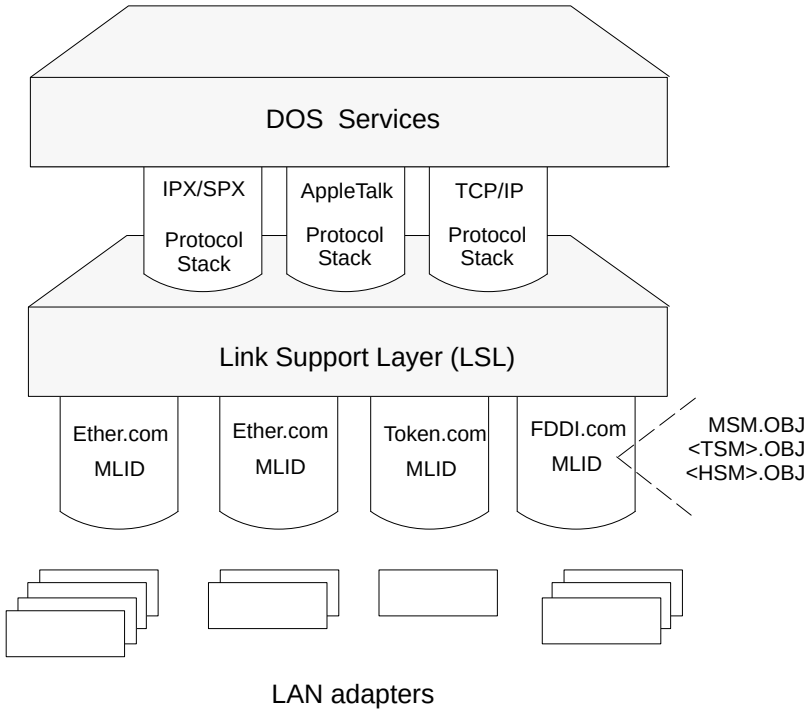
Note The terms *MLID* and *LAN driver* can be interchanged. ▲

You should read this chapter if you are not familiar with the basic concepts involved in the ODI specification.

Open Data-Link Interface (ODI)

DOS client MLIDs and protocol stacks must conform to the ODI specification. Figure 1.1 illustrates the elements that make up the ODI specification.

Figure 1.1
The ODI Specification Diagram



The ODI specification allows multiple network protocols and LAN adapters (physical boards) to be used concurrently on the same workstation or file server. It provides a flexible, high-performance Data Link Layer interface to Network Layer protocol stacks. The ODI specification is comprised of the three elements listed below and illustrated above in Figure 1.1 .

- Protocol Stacks
- Link Support Layer (LSL)
- Multiple Link Interface Drivers (MLIDs)

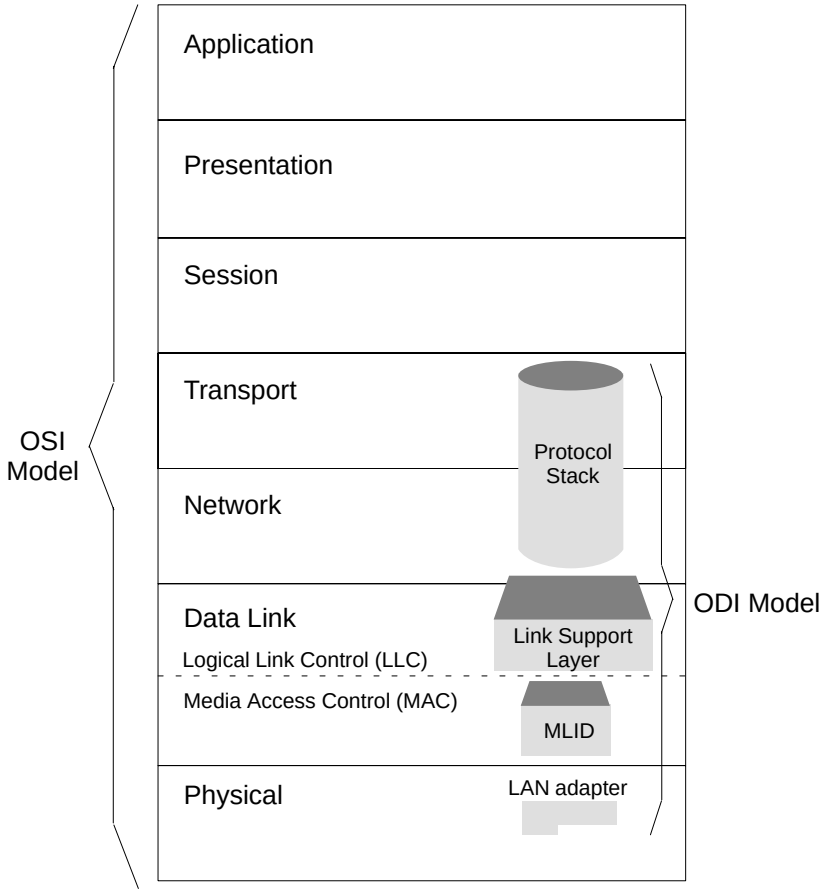
Protocol Stacks

Protocol Stack Functionality

Network Layer protocol stacks transmit and receive data over a logical or physical network. They also handle routing, connection services, and APIs, and provide an interface to allow higher layer protocols or applications access to the protocol stack's services. As a general rule, protocol stacks written to the ODI specification provide OSI (Open Systems Interconnection) Network Layer functionality; however, they

are not limited to this. Figure 1.2 illustrates the ODI/OSI correspondence.

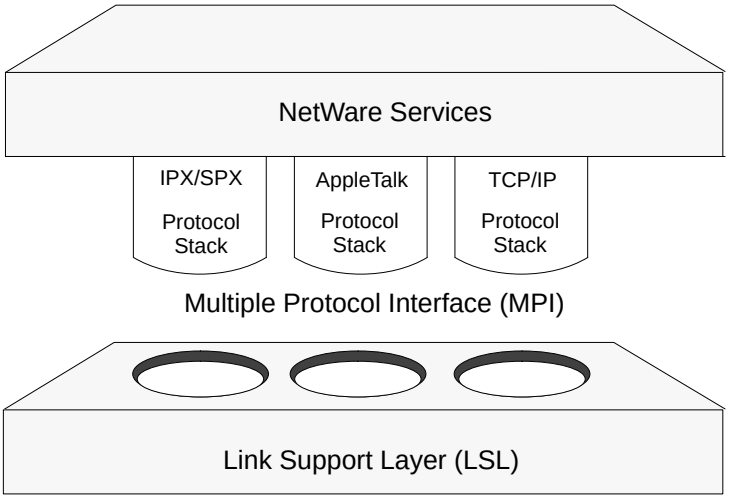
Figure 1.2
How ODI Fits into the OSI Model



The Multiple Protocol Interface (MPI)

Protocol stacks communicate with the LSL through the Multiple Protocol Interface™ (MPI™). The MPI is an interface that resides between the protocol stack and the LSL (see Figure 1.3). The MPI provides protocol stacks with all the APIs that are necessary for the protocol stack to communicate over the network.

Figure 1.3
The Multiple Protocol Interface (MPI)



Link Support Layer (LSL)

The LSL handles the communication between protocol stacks and MLIDs. Because the ODI allows the physical topology to support many different types of protocols, the MLID receives packets destined for different protocol stacks that might be present in the system. For example, one Ethernet network might support all of the following protocols: IPX, TCP/IP, AppleTalk, and LAT (a Digital Equipment Corporation protocol). The LSL then determines which protocol stack is to receive the packet. Next, the protocol stack determines what should be done with the packet or where it should be sent. When the protocol stack transmits a packet, it hands the packet to the LSL. The LSL then directs the packet to the appropriate MLID.

The LSL also tracks the various protocols and MLIDs that are currently loaded in the system and provides a consistent method of finding and using each of the loaded modules.

In addition, the LSL performs the following services:

- Allows a protocol stack to obtain and return *Event Control Blocks (ECBs)*. (ECBs are control structures that are used to send or receive packets or to schedule events.)
- Queues and recovers ECBs for later use.
- Registers and deregisters the protocol stack.
- Allows protocol stacks to obtain timing services.
- Allows protocol stacks to determine Stack and Protocol IDs.
- Allows protocol stacks to obtain MLID statistics.
- Allows protocol stacks to bind with MLIDs.

- Allows protocol stacks to transmit and receive packets through an MLID.
- Maintains lists of all active protocol stacks and MLIDs.
- Allows protocol stacks to obtain information about MLIDs and other protocol stacks.
- Allows protocol stacks to change the operational state of MLIDs. (For example, the protocol stack could cause the MLID to shut down or reset.)

Multiple Link Interface Drivers (MLIDs)

MLID Functionality

MLIDs are device drivers that handle the sending and receiving of packets to and from a physical or logical topology (for example, Ethernet SNAP is a logical topology). MLIDs interface with a LAN adapter (also referred to as Network Interface Card [NIC] or physical board) and handle frame header appending and stripping. MLIDs also help determine the packet's frame type.

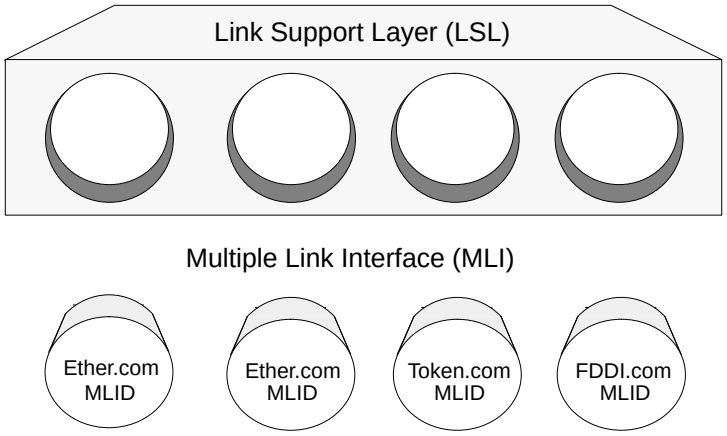
Each MLID's interface with the LAN adapter is determined by that adapter's hardware.

All MLIDs can handle packets from various protocols because the MLID does not interpret the packet. Instead, it passes received packets to the Link Support Layer (LSL) using Event Control Blocks (ECBs). ECBs are data structures that the MLID uses to send or receive packets or to schedule events.

The Multiple Link Interface (MLI)

The MLID communicates with the LSL through the Multiple Link Interface™ (MLI™). The MLI is the interface between the LSL and the MLID (see Figure 1.4). This interface contains the APIs necessary to facilitate communication between these two modules.

Figure 1.4
The Multiple Link Interface (MLI)



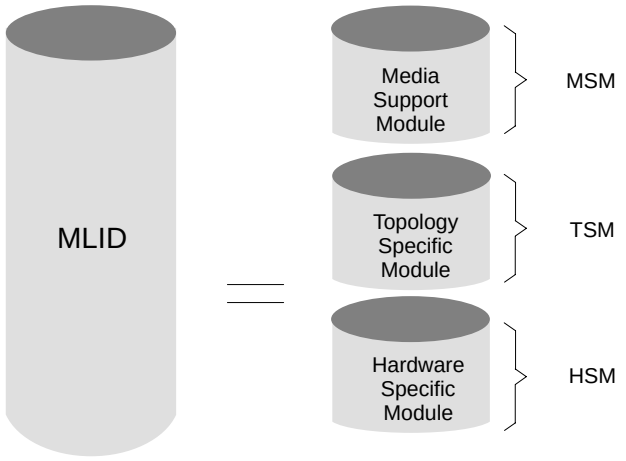
LAN Driver Toolkit

Novell has simplified the task of ODI LAN driver development by furnishing a set of support modules that provide all the tools necessary to interface a LAN driver to the LSL. These modules are:

- Media Support Module (MSM)
- Topology Specific Modules (TSM)

These support modules are a collection of procedures, macros, structures, and variables that simplify driver development. When using these modules, LAN driver development is reduced to creating the Hardware Specific Module (HSM). The HSM handles all hardware interactions. (Figure 1.5 illustrates the relationship of these modules to an MLID.)

Figure 1.5
MLID Modules



Media Support Module. The Media Support Module (MSM) contains general functions that are common to all drivers.

Topology Specific Module. The Topology Specific Module (TSM) manages the operations unique to a specific topology.

TSMs provide support for the standardized media types of Ethernet, Token-Ring, and FDDI. Multiple frame support is implemented in the TSM so that all frame types for a given topology are supported. The possible topology modules are listed below.

- ETHERNET.OBJ
- TOKEN.OBJ
- FDDITSM.OBJ

Source code for each TSM is provided with the *Novell LAN Driver Developer's Guide*. Although not recommended, you can create proprietary topology modules by modifying an existing TSM to meet your requirements or by creating a new module that provides the same functionality contained in the standard TSMs; however, your proprietary TSM must conform to the functionality defined in this specification and have a unique name.

Hardware Specific Module. You create the Hardware Specific Module for a specific LAN adapter. The HSM handles all hardware interactions. Its primary functions include adapter initialization, reset, and shutdown, as well as packet reception and transmission. Additional procedures provide support for timeout detection, multicast addressing, and promiscuous mode reception. When you use the LAN driver toolkit to develop an MLID, the HSM is the only module that you write. This document uses the term "HSM" to refer to that portion of the MLID that you develop with this toolkit.

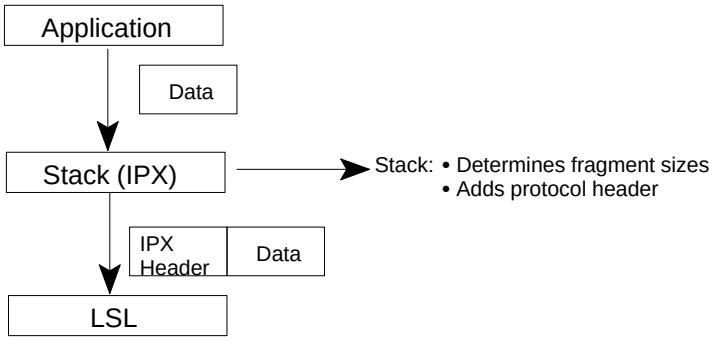
Data Flow

When messages are sent and received, the various protocols or layers add and remove their own information at each layer. The following diagrams illustrate basic data flow.

Send Data Flow

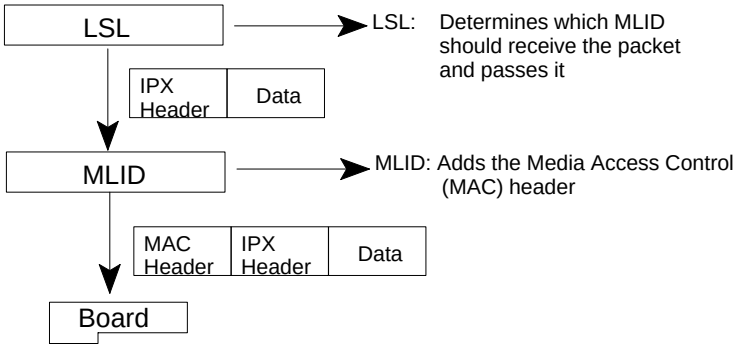
As Figure 1.6 illustrates, the protocol stack receives data from the application above it, determines whether the packet must be split into fragments, determines the size of the fragments, adds the appropriate protocol header to the data packet, and sends it to the LSL. The LSL isolates the protocol stack from the topology and LAN medium below it. The protocol stack simply passes data to the LSL. The LSL directs the packet to the appropriate MLID, which then takes care of the topology-specific information. This is the reason ODI protocol stacks are known as being media and frame-type independent.

Figure 1.6
Data Flow from Application to LSL



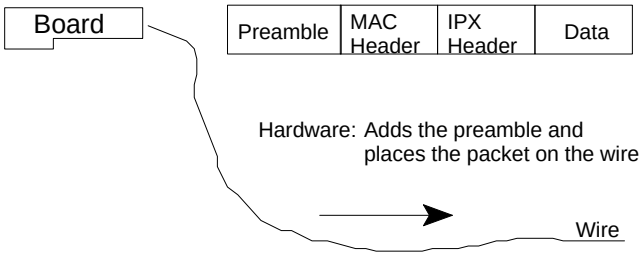
As illustrated by Figure 1.7, the LSL directs the packet to the appropriate MLID. The MLID then adds the MAC header to the packet and hands the packet to the LAN adapter.

Figure 1.7
Data Flow from the LSL to the Board



In Figure 1.8 the hardware adds the preamble to the packet and places the packet on the wire.

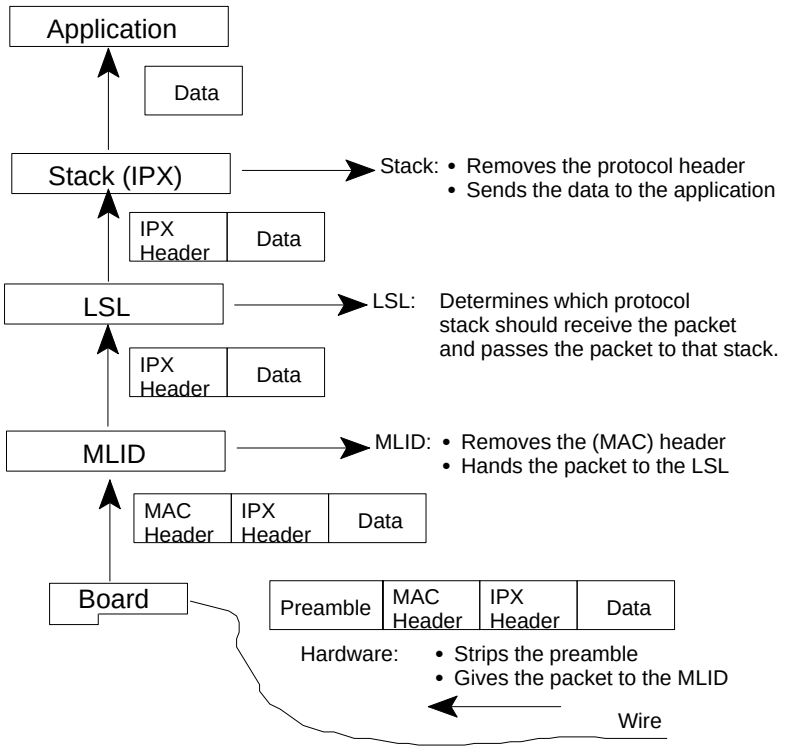
Figure 1.8
Data Flow from the Board to the Wire



Receive Data Flow

Figure 1.9 shows the LAN adapter receiving the packet off the wire and stripping the preamble from the packet. The LAN adapter then hands the packet to the MLID, which discards the MAC header from the packet and hands the packet to the LSL. The LSL directs the packet to the appropriate protocol stack, which then removes the protocol header from the packet and hands the data to the application.

Figure 1.9
Receive Data Flow
from Wire to Application



❑

Chapter 2 Designing the DOS ODI HSM

Chapter Overview

This chapter briefly describes the design, programming, and functionality factors you must understand to write DOS ODI client HSMs. This chapter discusses the LAN driver in the DOS environment.

You should read this chapter if you have never written a DOS HSM.

The DOS Environment

Because your HSM is to operate in the DOS environment, you must be aware of the following issues:

- DOS does not multitask
- DOS ODI LAN drivers are not reentrant
- DOS ODI LAN drivers are *.COM* files
- DOS ODI LAN drivers are Terminate and Stay Resident (TSR) programs

Multitasking Issues

DOS is not a multitasking operating system. The minimum environment in which your LAN driver is required to operate is a simple real mode DOS environment. Usually this means the operating system can only run one task to completion (nonmultitasking). However, you should keep in mind that your LAN driver might also be operating with DOS multitasking products.

LAN drivers in the DOS environment are not reentrant. Each LAN adapter located in the workstation contains its own code and data images in memory. The boards do not share code images.

.COM File Issues

DOS ODI LAN drivers are *.COM* files, which means they have only one 64K segment group. (See the "Hardware and Programming Issues" section below for more information.)

TSR Program Issues

DOS ODI drivers are TSRs. When the DOS ODI LAN driver completes its initialization process, the driver terminates and stays resident (TSR) in memory. Because ODI is a dynamic specification that allows LAN driver modules to be loaded and unloaded as they are needed, the LAN driver must be fully unloadable. The LAN driver toolkit provides this capability.

Programming and Hardware Issues

Programming Issues

Driver's Code and Init Sections

A DOS ODI LAN driver is a *.COM* file, thus limiting the driver to the use of one segment. The driver is broken into two parts, and these are joined to form one segment. The following are the two parts:

- *Code* Contains all the run time code and data.
- *Init* Contains all the code and data which are used only at initialization time. This part is discarded after initialization.

These parts are stored in memory in the same order as they are listed above.

CGroup

The MSM groups the two segment parts into a group called *CGroup*. All offsets are relative to *CGroup* and not to their individual parts. When referencing relative offsets, always explicitly specify *CGroup* so that the assembler generates the correct offset. For example:

```
Code            segment para public 'CODE'
    MyVariable  dw  ?
Code            ends

Init            segment para public 'CODE'
    mov     si, offset CGroup:MyVariable
Init            ends
```

Important Be aware that placing code or data into your driver in the *Code* portion of the segment after initialization wastes memory. Furthermore, do not place code or data into the *Init* portion of the segment that will be needed after the driver has installed itself. The driver will only use this portion of code and data during driver initialization. After initialization is complete, the driver discards the *Init* portion of code and data. ▲

Segment Attributes

When the driver is setting up a portion of the segment, it should have the following segment attributes:

para public 'CODE'

The HSM should include DRIVER.INC at the top of the HSM's source file:

DRIVER.INC includes several other include files.

Before you write the HSM, you should go through all include files, MSM source modules, and sample drivers shipped to you with this kit. This will enhance your understanding of the system and help you to write a better driver.

Necessary Routines and Structures

Every LAN driver also needs to have initialization, send, receive, reset and shutdown routines. These routines contain

code that is located in the HSM as well as in the TSM. The TSM's portion of the code is typically executed before and after an HSM routine is called (all calls made to the HSM by the MSM are *near* calls). In addition, LAN drivers also need two data structures: the HSM configuration table and the HSM statistics table.

(See Appendix A for instructions on assembling and linking your LAN driver.)

Hardware Issues

Before writing the HSM, you should have a thorough understanding of the adapter. Knowing the characteristics of the hardware, bus type, and data transfer mode allows you to create a more efficient driver.

Network Interface Controllers

You should be familiar with the Network Interface Controller (NIC) integrated circuit. You should make every effort to obtain and use current data books and application notes from the manufacturer. In addition, the manufacturer's support engineers can provide up-to-date information on hardware quirks and modifications.

Data Transfer Mode

The MSM and TSM provide certain support procedures that are optimized for use with a specific data transfer mode. For example, the development of the HSM's packet reception and transmission routines will be affected by the adapter's transfer mode. The following are the data transfer modes:

- Programmed I/O
- Shared RAM (Memory Mapped I/O)
- Direct Memory Access (DMA)
- Bus Master

Bus Type

You should also consider the bus type and size in creating optimized HSM operations. The HSM's initialization process will be affected by the bus type when it initializes and registers the hardware configuration with the MSM and Link Support Layer. The following are the bus types:

- Industry Standard Architecture (ISA)
- Micro Channel Architecture
- Extended Industry Standard Architecture (EISA)

- Peripheral Component Interconnect (PCI)
- Personal Computer Memory Card International Association (PCMCIA)
- Video Electronics Standard Association (VESA) Local Bus

❑

Chapter 3 DOS ODI HSM Overview

Chapter Overview

This chapter provides an overview of writing a DOS ODI client HSM. It also discusses the procedures and functions that the HSM should provide. Depending on the hardware and media of the physical board, your HSM may not need to meet all the requirements discussed in this chapter.

You should read this chapter if you have never developed an HSM for a DOS ODI LAN driver.

DOS LAN Driver Modules

The Modules That Comprise the LAN Driver

An DOS ODI LAN driver is created by linking the following three modules together:

- MSM.OBJ (Media Support Module)
- <TSM>.OBJ (Topology Specific Module)
- <HSM>.OBJ (Hardware Specific Module)

Media Support Module (MSM)

The Media Support Module (MSM) is common to all DOS ODI LAN drivers that use the LAN driver toolkit. The MSM handles all generic initialization and run-time issues.

Topology Specific Module (TSM)

The <TSM>.OBJ module is specific to the topology you are using. (Replace the <TSM> with the appropriate module name.) For example, LAN drivers that drive Ethernet LAN adapters use the ETHERNET.OBJ TSM. The TSM is responsible for handling frame header appending and stripping, multicasting, source routing, and any other issues unique to a particular topology. We provide the following three TSMs that you can use to develop your LAN driver:

- ETHERNET.OBJ (for Ethernet boards)
- TOKEN.OBJ (for 802.5 token-ring boards)
- FDDITSM.OBJ (for FDDI boards)

Note This list is not exclusive and may change. ▲

Other TSMs are created as new media are supported. Generally, we do not provide TSMs for proprietary topology types. If you are supporting a proprietary topology, you must create your own Topology Specific Module.

Hardware Specific Module (HSM)

The <HSM>.OBJ file contains the hardware-specific code you will create for a specific physical board. This module is responsible for handling all hardware interactions.

We have provided the source code to the MSM.OBJ and <TSM>.OBJ in the files MSM.ASM and <TSM>.ASM. You can modify the code to suit the particular requirements of your LAN adapter. However, keep these changes to a minimum and carefully document them, because Novell updates these modules as the need arises. If you have changed these modules,

you must reincorporate those changes into the new MSM.ASM and <TSM>.ASM modules when we supply them.

Developing The Hardware Specific Module (HSM)

HSM Data Structures

Every HSM must contain

- One configuration table data structure.
- One statistics table data structure.

Configuration Table

The configuration table is a data structure that defines the configuration of the physical board and MLID. The fields in this table are primarily used during initialization and are referred to by the LSL and the MLID. The requirements for configuration tables are explained in detail in Chapter .4 The MSM copies and maintains the configuration table for each logical board.

Statistics Table

The statistics table is a data structure that contains data on the operation of the physical board and the MLID. Both the LSL and the MLID look at fields in this table. Chapter 4 contains a detailed description of this data structure.

HSM Routines

The MSM calls all of the following routines. Therefore, the HSM must be aware of the entry conditions that exist and must meet any execution and return conditions the MSM requires. Chapter 6 describes these conditions.

Important The names of your routines must exactly match the italicized names of the routines below. ▲

Routines that Must Be Fully Written

You must write the following routines:

- *DriverInit* (initializes the HSM).
- *DriverISR* (services interrupts and receives packets).
- *DriverMulticastChange* (updates the multicast table). If your hardware does not support multicast addressing, you may stub this routine.
- *DriverReset* (resets the hardware).
- *DriverSend* (sends packets).
- *DriverShutdown* (shuts down the hardware).

Routines that Can Be Stubbed

Even if you choose not to support any of the following routines in your HSM, you must still include the routine, but you can stub it so that it contains only a return instruction.

- *DriverChangeLookAheadSize* (changes the look-ahead size).
- *DriverManagement* (manages HSM specific features).
- *DriverPoll* (polls the driver).
- *DriverPromiscuousChange* (changes the promiscuous mode).

MSM Support for the HSM

Chapter 6 describes the MSM support calls available to the HSM. These calls handle common driver interactions with the MSM. The calls are:

- HSMProvideTCB
- HSMShutDownMSM
- MSMBuildTransmitControlBlock
- MSMCallNESL
- MSMClearSendQueue
- MSMGenerateNESLChangeEvent
- MSMGenerateNESLEvent
- MSMGenerateNESLResumeEvent
- MSMGenerateNESLSuspendEvent
- MSMGetNextSend
- MSMGetRCB
- MSMMediaConfigUpdate
- MSMPrintStringZero
- MSMRcvComplete
- MSMRcvCompleteStatus
- MSMReturnRCB
- MSMSendComplete
- MSMSetIRQ
- MSMUnSetIRQ
- MSMUpdateMulticast

LAN Driver Capabilities

Your LAN driver must provide the following capabilities when feasible. In some instances, this manual recommends certain ways of implementing these capabilities to allow the HSM to be as versatile as possible, to run in as many environments as possible and to coexist successfully with any additional TSRs. However, you can implement these capabilities as you choose.

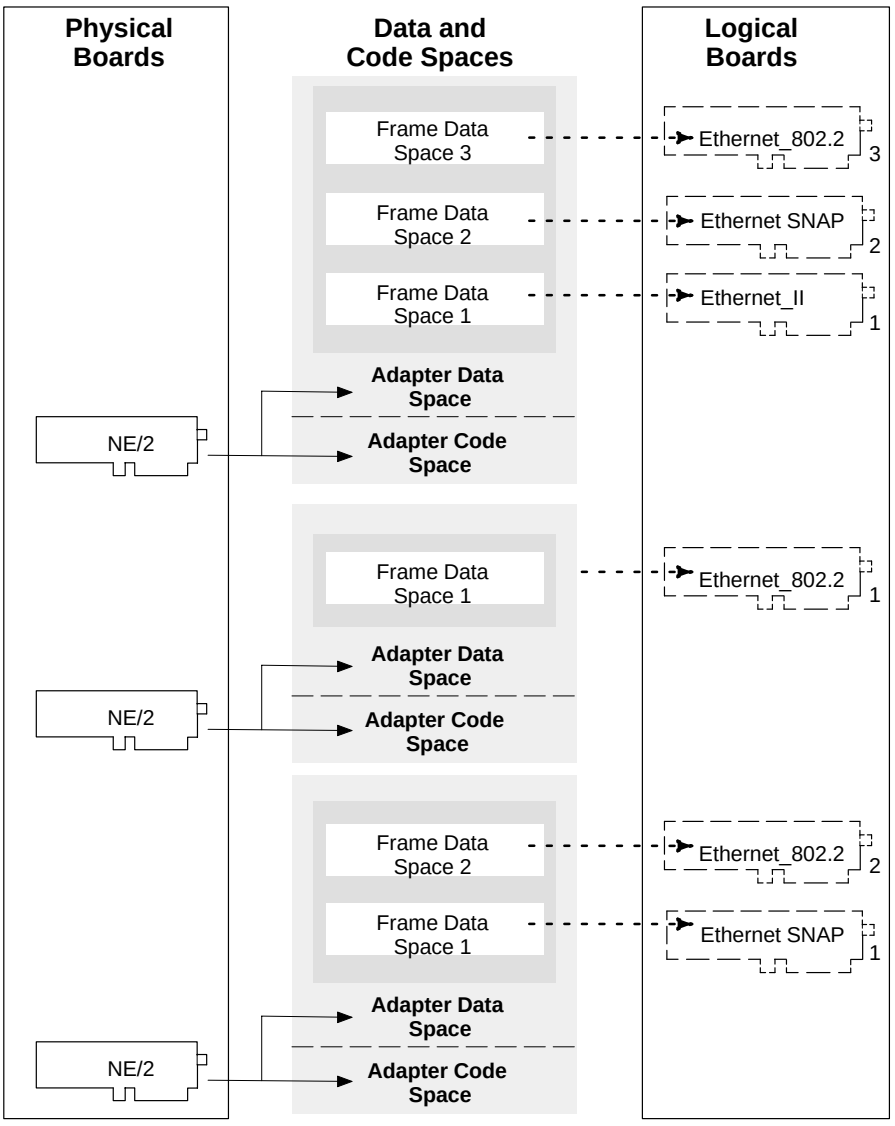
Multiple Frame Support

The MSM provides multiple frame support capability. Therefore, your HSM automatically supports multiple frame

types. (You can allocate frame types in the NET.CFG file. See Appendix B.)

The default frame type for the physical board is 802.2. If your LAN adapter runs on a topology that supports multiple frame types, the MSM supports each frame type by using *logical boards* (see Figure 3.1). A logical board is merely a piece of code written to emulate a physical board that uses a specific frame type.

Figure 3.1 Implementation of Multiple Frame Support in Ethernet Topology



The NetWare operating system does not concern itself with distinguishing between logical boards that have exclusive use of a LAN adapter and logical boards that might share the same LAN adapter with other logical boards. Only the MLID makes this distinction.

The MSM creates a logical board for each frame type the MLID supports. Each logical board uses the same code image and the same *adapter data space* loaded into memory. However, the MSM maintains a separate *frame data space* for each logical board.

Adapter Data Space

Because DOS ODI LAN drivers are not reentrant, the MSM allocates in memory only one adapter driver space and one code space for each physical board. In addition, the MSM also allocates one frame data space for each frame type (logical board) supported. The HSM statistics table is included in the adapter data space.

The ODI specification defines the statistics table (see Chapter 4).

Frame Data Space

Every logical board has one frame data space associated with it. The frame data space is a structure that contains the frame specific information necessary for the MSM to support a specific frame type. The configuration table is part of this structure. The MSM creates and maintains a separate frame data space for each frame format that is loaded.

The MSM fills in the configuration table by reading a configuration file called NET.CFG (see Appendix B). If this file is not present, the MSM uses the default frame type specified by the <TSM>.OBJ for that particular topology. If the NET.CFG file is present, you can specify multiple frame types in it. In that case, the MSM allocates one configuration table for each frame type (logical board) by creating a copy of the configuration table in the MLID's data space and modifying the fields appropriately for that logical board.

The ODI specification defines the configuration table (see Chapter 4).

Multicast Address Support

If the LAN adapter is physically capable of supporting multicast addressing, the HSM must support it. The HSM I/O control routine *DriverMulticastChange* implements multicast support. (See Chapter 5.)

Source Routing Support

The MSM provides source routing capability. Therefore, your HSM automatically supports source routing.

Only token-ring and FDDI drivers support source routing. In these drivers, the MSM handles the source routing support; therefore, the HSM transparently supports it.

Promiscuous Mode

We strongly recommend that your HSM support promiscuous mode. When MLIDs operate in *promiscuous mode,* the HSM configures the hardware to receive all packets on the LAN medium. This includes bad packets, if possible. Because various monitoring functions operate in promiscuous mode, we strongly recommend that your HSM support promiscuous mode if your adapter is capable of such support. The HSM enables or disables promiscuous mode upon request by using the *DriverPromiscuousChange* routine described in Chapter 5.

❑

Chapter 4 HSM Data Structures and Variables

Chapter Overview

This chapter describes general structures and variables that the HSM uses. Because each structure and variable name in this chapter is predefined, you must use these exactly as they appear in this document.

- **Required Variables and Constants**

  This section describes the variables and constants that must be present in the HSM in order to avoid linker errors.

- **Variables and Constants Provided by the MSM**

  This section describes some of the variables and constants the MSM provides for your optional use.

- **Structures Required by the MSM**

  This section describes the structures and tables that must be present in the HSM in order to avoid linker errors.

- **Structures Provided by the MSM**

  This section describes some of the structures and tables the MSM provides for your optional use.

This chapter contains useful reference material for the developer.

Required Variables and Constants

The variables and constants listed must be present in the HSM.

Code Segment: General Variables and Constants

DriverConfigTable : label

DriverConfigTable is a label that must immediately precede the HSM configuration table. The MSM uses this label to locate the configuration table that follows.

Example

```
DriverConfigTable   label   byte
    MSignature              db  'HardwareDriverMLID     '  ;Note the 8
                                                           ;spaces at
                                                           ;the end of
                                                           ;the string
    MConfigTableMajorVer    db  01
    MConfigTableMinorVer    db  13  ;v1.13 for this table
    MNodeAddress            db  6 dup (0ffh)
        .
        .
        .
    MDMALine2               db  0
```

DriverStatTable : label

DriverStatTable is a label that must immediately precede the statistics table. The MSM uses this label to locate the various fields in the table that follows it. See the definition of the HSM statistics table on page 4-26 and in the ODI.INC file.

Example

```
DriverStatTable label   byte
DriverStatMajorVer      db      01
DriverStatMinorVer      db      01;v1.01 for this table
NumberGenericCounters   dw      14
ValidCountersMask       dd      0
.
.
.
```

Init Segment: General Variables and Constants

DriverMainSectionText : byte

The MSM uses the *DriverMainSectionText* string variable to locate the HSM's section text inside the NET.CFG file. The string in *DriverMainSectionText* must be in upper case and the 'LINK DRIVER' string must be the first part, followed by the driver name. The driver name is usually the name of the MLID's executable file. This string must be zero-terminated.

Example

```
DriverMainSectionText   db  'LINK DRIVER NE1000',0
```

DriverSignOnMessage : byte

The MSM displays the *DriverSignOnMessage* string variable when it loads. The HSM defines this string to contain the company name, board name, two spaces, version, one space, and date string, followed by the appropriate copyright messages. The string must be zero-terminated. Furthermore, the string 'VeRsIoN=' must immediately precede the *DriverSignOnMessage*. The string 'VeRsIoN=' must use the mixed case exactly as it appears in this document.

Important In order to identify which version of this specification an HSM conforms to, a version string (the "specification version string") must be embedded into the HSM. The specification version string number (4.00 for this specification) is the actual version number of the specification. The following is the specification version string for this specification; it must be added to the HSM where the global variable declarations are made exactly as shown:

```
HSMSPEC                 db  'HSM_SPEC_VERSION: 4.00',0
```

▲

Example

```
HSMSPEC                 db  'HSM_SPEC_VERSION: 4.00',0
                        db  'VeRsIoN='
DriverSignOnMessage     db  'Novell  NE2 Ethernet  MLID'
                        db  '  v1.00 (950831)',0Dh,0Ah
                        db  '(C) Copyright 1991–1995 Novell, '
                        db  'Inc. All Rights Reserved.',0
```

Init Segment: Driver Keywords

DriverKeywordText : word

DriverKeywordText is a word table of pointers to strings defining the text for each parameter that the MSM uses to parse the NET.CFG file. This variable must be present to avoid linking errors, regardless of whether the HSM implements custom keywords.

Example

```
DriverKeywordText   dw  offset CGroup:MaxPacketText
                    dw  offset CGroup:CableTypeText
```

DriverKeywordTextLen : word

DriverKeywordTextLen is a table of words defining the length of each keyword string defined in the *DriverKeywordText* table.

This variable must be present to avoid linker errors, regardless of whether the HSM implements this feature.

Example

```
DriverKeywordTextLen  dw  MaxPacketTextLen
                      dw  CableTypeTextLen
```

DriverNumKeywords : abs

DriverNumKeywords is a constant that specifies the number of driver defined NET.CFG keywords. This variable must be present to avoid linker errors, regardless of whether the HSM implements this feature. HSMs set this to 0, if the HSM has no custom NET.CFG keywords.

Example

```
DriverNumKeywords     equ     2
```

DriverProcessKeywordTab : word

DriverProcessKeywordTab is a table of word pointers to near procedures the MSM calls when it encounters the associated keyword parameter in the NET.CFG file. Whether or not the HSM uses custom keywords, this variable must be present to avoid linker errors.

Example

```
DriverProcessKeywordTab   dw  offset CGroup:ProcMaxPack
                          dw  offset  CGroup:ProcCableType
```

Defining and Using Driver KeyWords

The example below illustrates how the driver defines and uses *DriverKeywordText, DriverKeywordTextLen, DriverProcessKeywordTab,* and *DriverNumKeywords.* The example assumes the driver has two unique keywords.

Example

```
DriverNumKeywords       equ             2

MaxPacketText           db          'MAX PACKET SIZE'
MaxPacketTextLen        equ             $–MaxPacketText
CableTypeText           db          'CABLE TYPE THICK'
CableTypeTextLen        equ             $–CableTypeText

DriverKeywordText       dw offset  CGroup:MaxPacketText
                        dw offset  CGroup:CableTypeText

DriverKeywordTextLen    dw          MaxPacketTextLen
                        dw          CableTypeTextLen

DriverProcessKeywordTab dw offset  CGroup:ProcessMaxPacketSize
```

```
                                        dw offset  CGroup:ProcessCableType
;
;       DS  CGroup
;       SI  –> buffer holding any parms
;       BX  –> Configuration Table
;       Interrupts enabled
;       CLD is in effect
ProcessMaxPacketSize    proc    near
    call   MSMEatWhite              ; Point SI at parameter
    jnz    RanOutOfFile
    call   MSMAtoI                  ; Get the desired packet size value
    mov    MyMaxPacketSize, ax

RanOutOfFile:
    ret
ProcessMaxPacketSize    endp
ProcessCableType   proc    near
    mov    CableType, 1             ; Signal cable type
    ret
ProcessCableType   endp
```

Variables and Constants Provided by the MSM

Code Segment: General Variables and Constants

LSLSupport : dword

LSLSupport is a double word variable that holds the far address of the LSL's MLID entry point. This variable is used to invoke the MLID support functions. Remember that DS must equal CGroup when using this variable unless a segment override prefix is used as shown in the example below. For example, to get the LSL's millisecond interval marker, you would execute the following code:

Example

```
mov bx, MLIDSUP_GET_INTERVAL_MARKER
call  cs:LSLSupport
```

MSMEOIFlag : byte

MSMEOIFlag is a byte variable that signals whether an EOI operation should be issued to the slave programmable interrupt controller (PIC) when processing a hardware interrupt in the HSM. The HSM uses this flag when issuing EOI commands inside its *DriverISR* procedure. This flag is set to a nonzero value when it is necessary for the HSM to issue EOI to the slave PIC (port A0h) in addition to the master PIC. If this flag is 0, the HSM need only issue EOI to the master PIC (port 20h). This byte is valid after calling *MSMSetIRQ*.

Example

```
cmp MSMEOIFlag, 0       ;Do we need to EOI the slave?
JNE EOISlavePIC         ;NE=YES, EOI the slave.
```

MSMIntMaskOff : byte

MSMIntMaskOff is a byte variable that holds an appropriate mask value that when *OR*ed with the current 8259 mask value disables the 8259's interrupt line specified in the configuration table's *MIRQLine1* field. This interrupt is usually the LAN adapter's interrupt. The MSM sets up this variable during the call to *MSMSetIRQ*.

Example

```
;Disable LAN adapter's interrupt line
    mov  dx, MSMIntMaskPort
    in   al, dx             ;Read current mask value
    slow                    ;Add at least 500ns delay
    or   al, MSMIntMaskOff
    out  dx, al             ;Output new mask value
```

MSMIntMaskOn : byte

MSMIRQMaskOn is a byte variable that holds an appropriate mask value that when *AND*ed with the current 8259 mask value will enable the 8259's interrupt line specified in the configuration table's *MIRQLine1* field. This interrupt is usually the LAN adapter's interrupt. The MSM sets up this variable during the call to *MSMSetIRQ*.

Example

```
;Enable LAN adapter's interrupt line
    mov  dx, MSMIntMaskPort
    in   al, dx             ;Read current mask value
    slow                    ;Add at least 500ns delay
    and  al, MSMIntMaskOn
    out  dx, al             ;Output new mask value
```

The slow macro is defined as follows:

```
slow    macro
        push    ax
        in      al, 61h
        in      al, 61h
        in      al, 61h
        pop     ax
    endm
```

MSMIntMaskPort : word

MSMIntMaskPort is a word variable that holds the port address of the 8259 mask register applicable to the interrupt line specified in the configuration table's *MIRQLine1* field. The MSM sets up this variable when it calls *MSMSetIRQ*. The HSM uses *MSMIntMaskPort* whenever the HSM must enable or disable a LAN adapter's interrupt line. (See the example for *MSMIntMaskOn* and *MSMIntMaskOff*.)

MSMLookAheadSegment : word

MSMLookAheadSegment is a word variable that the HSM sets during the *DriverInit* routine. The MSM uses the *MSMLookaheadSegment* variable to initialize some of its internal data structures at initialization time instead of run time, thereby increasing performance. The HSM sets this variable to the value of the segment where the receive look ahead data is located. Shared memory based HSMs usually set this variable to the segment address of the LAN adapter's shared memory. I/O and DMA based LAN adapters set this variable to CGroup because they provide look ahead data using an internally allocated buffer. The HSM sets this variable when initialization has been successful and before the HSM returns from *DriverInit*.

The HSM calls *MSMMediaConfigUpdate* if it must modify this variable after returning from *DriverInit*.

Example

```
mov ax, cs
mov MSMLookAheadSegment, ax    ;Where AX contains
                               ;the segment value.
```

MSMMaxMulticastAddr : equate

MSMMaxMulticastAddr is a decimal equate indicating the maximum number of multicast addresses the MSM can handle. For example, the MSM might be able to handle 16 entries in the multicast table. This equate would then be set to 16.

MSMPhysNodeAddress : 6–byte

MSMPhysNodeAddress is a 6-byte variable that provides the MLID's node address in the format required by the hardware.

MSMPriorityQSupportPtr : word

MSMPriorityQSupportPtr is a word variable that holds the near address of a function supplied by the HSM. This function enables the HSM to queue packets according to priority.

MSMSystemFlags : byte

MSMSystemFlags is a byte variable that holds information concerning the HSM's environment. Check the MSM.INC file for the current bit definitions. Note that you can set more than one bit. For example, if the machine were 80386 based, the *ATFlag* bit and *I386Flag* bit would both be set.

The following bits are defined:

Example

```
ATFlag      01h     ;Set if this machine's processor is
                    ;a 80286 or higher.
MCAFlag     02h     ;Set if this machine contains a
                    ;Micro Channel bus.
EISAFlag    04h     ;Set if this machine contains a
                    ;EISA bus.
I386Flag    08h     ;Set if this machine's processor is
                    ;an 80386 or higher.
PCIFlag     10h     ;Set if this machine contains a Peripheral
                    ;Component Interface BIOS.
CSFlag      20h     ;Set if PCMCIA Card Services is present.
NESLFlag    80h     ;Set if the NetWare Event Service Layer (NESL) is
                    ;present.
```

MSMTxFreeCount : byte

MSMTxFreeCount is a byte variable the HSM initializes to hold the number of hardware transmit resources the driver currently has available. The HSM increments this variable whenever a transmit resource becomes available. The HSM also sets this variable accordingly whenever the HSM is doing an internal hardware reset. *MSMGetNextSend* decrements this variable.

For example, if the HSM has two maximum size transmit buffers available on its LAN adapter, it sets *MSMTxFreeCount* equal to 2. If the LAN adapter supports hardware queuing and a large number of possible outstanding transmits, the HSM sets the variable to a value that represents the number of transmits the LAN adapter can process.

Note The MSM allocates *MAX_TCB_ALLOCATED* Transmit Control Blocks (TCBs). Therefore, *MSMTxFreeCount* cannot exceed *MAX_TCB_ALLOCATED*.▲

Example

```
inc  MSMTxFreeCount   ;After Tx resource becomes available.
```

MSMTxMonPtr : dword

MSMTxMonPtr is a double word variable that holds the far address of a registered transmit monitor. The *MSMSendComplete* routine calls the transmit monitor.

Init Segment: General Variables and Constants

No required variables or constants.

Structures Required by the MSM

Frame Data Space

The MSM calls *DriverInit* which initializes the adapter and the configuration table. After returning from *DriverInit*, the MSM

allocates the frame data space and creates a copy of the configuration table in this area. The MSM allocates a separate frame data space containing a separate configuration table for each frame the driver supports. If the HSM modifies any configuration table field after it returns from *DriverInit*, it must call *MSMMediaConfigUpdate*.

Configuration Table
Code segment

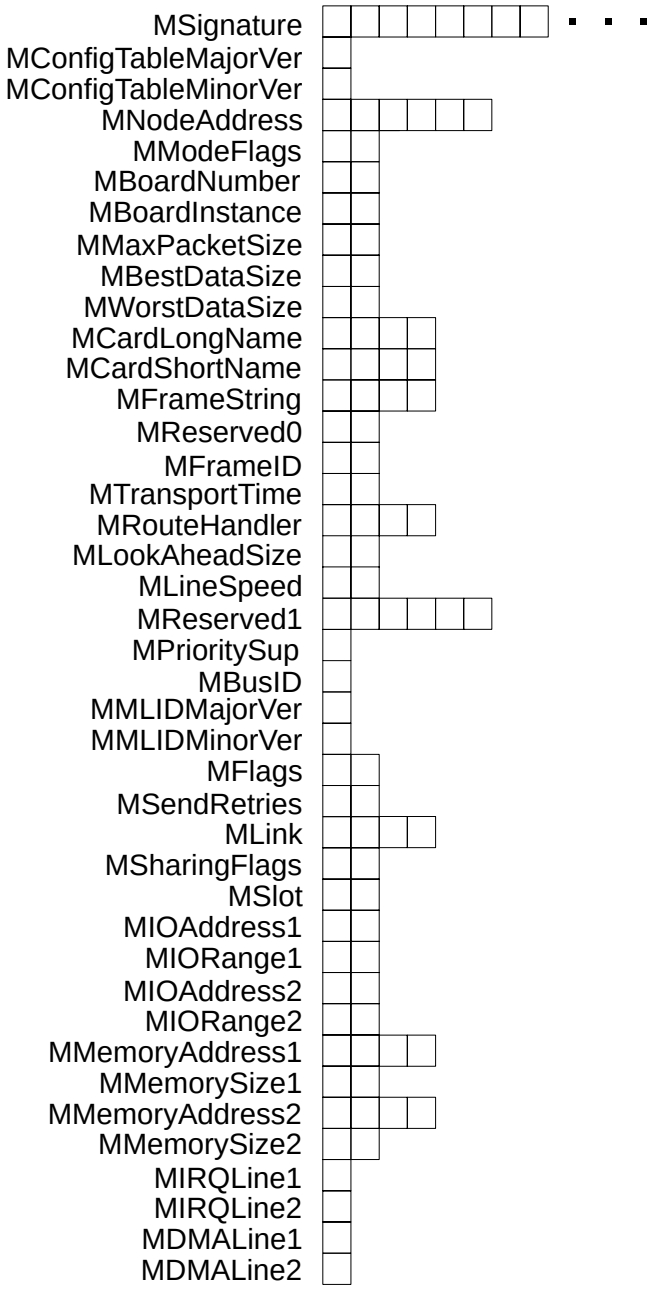
The HSM must define one configuration table containing information about the HSM and its configuration. The table must be defined by the fields shown below with each entry filled accordingly.

The following figures and tables contains the field names, descriptions, and other necessary information about the configuration table. This structure is defined in the ODI.INC file.

Figure 4.1
Configuration Table
Sample Source Code

```
MLIDConfigurationStructure      struc
    MSignature                  db  'HardwareDriverMLID',8 dup (' ')
    MConfigTableMajorVer        db   01
    MConfigTableMinorVer        db   13     ;v1.13 for this version
    MNodeAddress                db    6  dup (?)
    MModeFlags                  dw    0    ;Unused bits set to 0
    MBoardNumber                dw    ?    ;Assigned by the LSL
    MBoardInstance              dw    ?    ;Determined by the MSM
    MMaxPacketSize              dw    0
    MBestDataSize               dw    0
    MWorstDataSize              dw    0
    MCardLongName               dd    0
    MCardShortName              dd    0
    MFrameString                dd    0
    MReserved0                  dw    0    ;Must be set to 0
    MFrameID                    dw    0
    MTransportTime              dw    1    ;Nominally set to 1
    MRouteHandler               dd    0    ;Only for source routing
                                           ;support
    MLookAheadSize              dw   18    ;Default
    MLineSpeed                  dw    0
    MReserved1                  db    6  dup (0)
    MPrioritySup                db    0    ;Number of Priorities
                                           ;(Max=7)
    MBusID                      db   –1    ;Indicates the adapter's bus
                                           ;type.
    MMLIDMajorVer               db    0
    MMLIDMinorVer               db    0
    MFlags                      dw    0    ;Unused bits set to 0
    MSendRetries                dw    0
    MLink                       dd    0    ;HSM must not modify
    MSharingFlags               dw    1    ;Initially a driver is shut
                                           ;down
    MSlot                       dw    0    ;Default: HSM scans slots
    MIOAddress1                 dw    0    ;If not used, set to 0
    MIORange1                   dw    0
    MIOAddress2                 dw    0
    MIORange2                   dw    0
    MMemoryAddress1             dd    0
    MMemorySize1                dw    0
    MMemoryAddress2             dd    0
    MMemorySize2                dw    0
    MIRQLine1                   db  0FFh   ;if not used, set to 0FFh
    MIRQLine2                   db  0FFh
    MDMALine1                   db  0FFh
    MDMALine2                   db  0FFh
MLIDConfigurationStructure      ends
```

Figure 4.2
Graphic Representation of the Configuration Table



Description of HSM Configuration Table Fields

The following table describes the MLID configuration table fields and how they are initialized.

Table 4.1
HSM Configuration Table

HSM Configuration Table			
Offset	**Name**	**Size (in bytes)**	**Description**
0h	MSignature	26	This field contains the string 'HardwareDriverMLID' with eight spaces appended.
1Ah	MConfigTableMajorVer	1	This field defines the current major version of the configuration table structure. As changes are made to this structure, the revision level will be altered. For this specification, set this field to 01.
1Bh	MConfigTableMinorVer	1	This field defines the current minor version of the configuration table structure. For this specification, set this field to 13 (v1.13).
1Ch	MNodeAddress	6	This field holds the card's node address. The HSM sets this field during the initialization routine. (See the *ODI Specification Supplement: Canonical and Noncanonical Addressing*.)
22h	MModeFlags	2	This field contains flags the HSM sets by using the definitions which follow this table.
24h	MBoardNumber	2	During initialization, the MSM sets this field to the board number returned by *LSLRegisterMLID*.
26h	MBoardInstance	2	The MSM sets this field to the relative board number for this driver. A value of 0 indicates the initial frame type for this MLID—for example, if the MLID loads three frame types, the third frame type will have its configuration table *MBoardInstance* field set to 2 (zero based count).

HSM Configuration Table *(continued)*

Offset	Name	Size (in bytes)	Description
28h	MMaxPacketSize	2	This field defines the largest possible packet size that the driver/LAN adapter combination can transmit and/or receive. This value includes all headers. Typically, Ethernet HSMs set this field to 1514 decimal. For example, because token-ring drivers can send and receive a number of different packet sizes, a token-ring driver must determine the appropriate packet size during *DriverInit* and place that value in this field. Token-ring HSMs should support 4KB (4096 [data] + 30 [source routing] + 22 [MAC] + 74 [protocol header] = 4222) packet sizes whenever possible and practical. The value in this field cannot be less than 638 decimal (512 + 30 [source routing] + 22 [MAC] + 74 [protocol header] = 638).
2Ah	MBestDataSize	2	The MSM sets this field during *MSMMediaConfigUpdate*. The MSM subtracts the length of the *smallest* media header(s) from the value in the *MMaxPacketSize* field. For example, the Ethernet MSM sets this field to 1500 decimal (1514 – 14 [MAC] = 1500) if the HSM runs the Ethernet_II packet type. The token-ring MSM sets this field to *MMaxPacketSize* – 14 [MAC] – 3 [802.2 UI] if the HSM's packet type is token-ring.
2Ch	MWorstDataSize	2	The MSM sets this field during *MSMMediaConfigUpdate*. The MSM subtracts the length of the *largest* media headers(s) from the *MMaxPacketSize* field. For example, the token-ring MSM sets this field to *MMaxPacketSize* – 14 [MAC] – 3 [802.2 UI] – 30 [source routing] – 5 [SNAP] if the HSM's packet type is token-ring SNAP. An Ethernet_II driver sets this field to 1500. (MMax–14MAC=1500.) **Note:** Protocol stacks use the value in this field to determine the largest packet size this driver can send or receive.
2Eh	MCardLongName	4	This field holds a far pointer to a length-preceded, zero-terminated string that contains a full description of the LAN adapter.

HSM Configuration Table *(continued)*

Offset	Name	Size (in bytes)	Description
32h	MCardShortName	4	This field holds a far pointer to a length-preceded, zero-terminated string that holds a single descriptive name. The maximum length of this string is 8 characters, not including the length byte or zero terminator. The legal characters are 0–9 and upper- and lower-case A–Z. We recommend that this string contains the HSM's filename.
36h	MFrameString	4	This field holds a far pointer to a length-preceded, zero-terminated string describing the frame and media type being used by this logical board. (See the *ODI Specification Supplement: Frame Types and Protocol IDs*.)
3Ah	MReserved0	2	Set this field to 0.
3Ch	MFrameID	2	This field describes the frame and media type the logical board is using. (See the *ODI Specification Supplement: Frame Types and Protocol IDs*.)
3Eh	MTransportTime	2	The HSM sets this field to define the number of milliseconds it takes the HSM and LAN adapter to transmit a 512 byte data packet. The HSM cannot set this field to 0. Most HSMs set this field to a value of 1. If the HSM drives a board on a slow asynchronous line, it sets this field according to a representative value.
40h	MRouteHandler	4	MLIDs that support source routing use this field in conjunction with ROUTE.COM. HSMs initialize this field to 0 and then do not modify it. (See the *ODI Specification Supplement: Source Routing* for a discussion of source routing.)

HSM Configuration Table *(continued)*

Offset	Name	Size (in bytes)	Description
44h	MLookAheadSize	2	This field holds the configured look ahead size as set by protocol stacks. The HSM initializes this field to a default value of 18 bytes. When it receives a packet, the HSM uses this value and the maximum possible media header to determine the amount of look ahead data it must pass to the *MSMGetRCB* routine. The value in this field can be changed at any time. Therefore, the HSM must refer to this field for every packet it receives. The MLID can implement *DriverChangeLookAheadSize* to avoid checking on each packet. The maximum value this field can be set to is 128 bytes.
46h	MLineSpeed	2	This field holds the data rate used by the LAN adapter's medium (usually specified in megabits per second). The HSM sets this field to an appropriate value. This value is normally specified in megabits per second (Mbps). If the line speed is less than 1 Mbps or if it is a fractional number, the value of this field can be defined in kilobits per second (Kbps) by setting the most significant bit (bit 15) to 1. This field is undefined if it is set to 0. For example: If the speed of the line driver is 10 Mbps, put 10 (decimal) in this field. If the speed is 2.5 Mbps, then the value of this field is 2500 (decimal) logically ORed with 8000h (most significant bit is 1 for Kbps). If the line speed can be selected, as with token-ring, the HSM determines the selected line speed and places that value in this field. Some common values are listed below: Ethernet 10Mbps 000Ah Token-Ring 4Mbps 0004h Token-Ring 16Mbps 0010h FDDI 100 Mbps 0064h ISDN 64 Kbps 8040h
48h	MReserved1	6	This field is reserved. Set this field to 0.

HSM Configuration Table *(continued)*

Offset	Name	Size (in bytes)	Description
4Eh	MPrioritySup	1	This field contains the number of priority levels that the HSM can handle. This field has a maximum of 7 priorities (1–7). Zero indicates no priority packet support. Therefore, the HSM can set this field to a value of 0 through 7.
4Fh	MBusID	1	If the HSM supports multiple bus types, it checks this field during initialization to determine which bus it should be initialized for. This field is defined as follows: BUS_ID_ISA equ 0 BUS_ID_MCA equ 1 BUS_ID_EISA equ 2 BUS_ID_PCMCIA equ 3 BUS_ID_PCI equ 4 BUS_ID_VESA equ 5 BUS_ID_HSM_DEFAULT equ –1 If *MBusID* is set to –1, the HSM searches each of the machine's busses for a supported LAN adapter and initializes the first LAN adapter it finds. The HSM determines the order to search the busses in. If *MBusID* is initially set to default, the HSM sets it to the appropriate Bus ID. **Note:** If the user has set the *BUS ID* keyword for an unsupported bus, the HSM should exit *DriverInit* and return an error. (For information about the *Bus ID* keyword, see Appendix B.) ▲
50h	MMLIDMajorVer	1	This field defines the current major revision level of the driver. This field must match the revision level displayed in the *DriverSignOnMessage* string. For example, if the MLID's current major version is 2, this field would also be 2.
51h	MMLIDMinorVer	1	This field defines the current minor revision level of the driver. This field must match the revision level displayed by the *DriverSignOnMessage* string. For example, if the MLID's current minor version were .02, this field would also be .02. (If the current major and minor version level displayed by the MLID is 2.02, these fields should reflect that version of 2.02.)
52h	MFlags	2	This field contains flags the HSM sets by using the definitions which follow this table.

HSM Configuration Table *(continued)*

Offset	Name	Size (in bytes)	Description
54h	MSendRetries	2	The HSM initializes this field to an appropriate value that represents the number of times the HSM will retry an errored transmission operation before giving up. **Note:** 10 is a nominal default.
56h	MLink	4	The HSM sets this field to 0 and does not modify it.
5Ah	MSharingFlags	2	This field contains flags the HSM sets by using the definitions which follow this table.
5Ch	MSlot	2	The HSM initializes this field to 0. HSMs that control slot-based LAN adapters (for example, the Micro Channel Architecture boards) use this field. If the HSM is for an ISA board, it can ignore this field. If the HSM is for a Micro Channel Architecture, PCI, or EISA type board, it sets the slot number of the LAN adapter it is driving. Slot numbers are 1-based. An initial value of zero implies that the HSM scans for the board. The user can override this value with the NET.CFG file.
5Eh	IOAddress1	2	The HSM initializes this field to the default I/O port base address. If the HSM is self-configurable, it determines the appropriate value for the LAN adapter and places that value in this field before it returns from initialization. If the HSM does not use I/O ports, it sets this field to 0. The user can override this value with the NET.CFG file.
60h	IORange1	2	This field defines the number of I/O ports decoded by the LAN adapter at *MIOAddress1*. Set this field to 0 if the LAN adapter does not use I/O ports.
62h	IOAddress2	2	This field allows the HSM to define two I/O port base addresses. The definition is the same as *MIOAddress1*. Set this to 0 if the LAN adapter does not have a second range of I/O ports. The user can override this value with the NET.CFG file.
64h	IORange2	2	This field defines the number of I/O ports decoded by the LAN adapter at *MIOAddress2*. Set this field to 0 if the LAN adapter does not use I/O ports.

HSM Configuration Table *(continued)*

Offset	Name	Size (in bytes)	Description
66h	MMemoryAddress1	4	The HSM initializes this field to the LAN adapter's default base memory address. If the HSM is self-configurable, it determines the appropriate value for the LAN adapter and places that value in this field before returning from initialization. If the LAN adapter does not use or define shared RAM or ROM, the HSM sets this field to 0. This value is an absolute physical address. For example, if a LAN adapter's RAM were located at C000:0, the value in this field would be C0000. The user can override this value with the NET.CFG file.
6Ah	MMemorySize1	2	This field defines the number of paragraphs (16 bytes) decoded at *MMemoryAddress1*. If *MMemoryAddress1* is not defined, the HSM sets this field to 0.
6Ch	MMemoryAddress2	4	This field allows the HSM to define a second memory address range for the HSM's LAN adapter to use. For example, *MemoryAddress1* could define the starting address of the LAN adapter's RAM, and this field could define the starting address of the LAN adapter's ROM. Set this field to 0 if the LAN adapter does not define a second memory range. If the HSM is self-configurable, it determines the appropriate value for the LAN adapter and places that value into this field before returning from initialization. The user can override this value with the NET.CFG file.
70h	MMemorySize2	2	This field defines the number of paragraphs (16 bytes) decoded at *MemoryAddress2*. If *MemoryAddress2* is not defined, the HSM sets this field to 0.

HSM Configuration Table *(continued)*

Offset	Name	Size (in bytes)	Description
72h	MIRQLine1	1	The HSM initializes this field to the LAN adapter's default interrupt request line (IRQ). If the HSM is self-configurable, it determines the appropriate value for the LAN adapter and places that value into this field before returning from initialization. If the LAN adapter does not use an interrupt line, the HSM sets this field to 0FFh (unused). If the HSM's LAN adapter supports IRQ 2 or 9, the HSM sets the value to be consistent with the LAN adapter's documentation. For example, if the LAN adapter's documentation specifies the default jumper setting as IRQ 2, the HSM places a value of 2 in this field. If the LAN adapter's documentation specifies a default jumper setting as IRQ 9, the HSM places a value of 9 in this field. The HSM sets this field to 0FFh if the field is not needed. The user can override this value with the NET.CFG file.
73h	MIRQLine2	1	The HSM uses this field if the HSM's LAN adapter uses a second IRQ line. Set this field to 0FFh if it is not needed. The user can override this value with the NET.CFG file.
74h	MDMALine1	1	The HSM initializes this field to the LAN adapter's default DMA channel number. If the HSM is self-configurable, it determines the appropriate value for the LAN adapter and places that value in this field before returning from initialization. If the LAN adapter does not use DMA, the HSM sets this field to 0FFh (unused). The user can override this value with the NET.CFG file.
75h	MDMALine2	1	The HSM uses this field if the HSM's LAN adapter uses a second DMA channel. Set this field to 0FFh if the field is not needed. The user can override this value with the NET.CFG file.

Configuration Table Flags

This section contains bit maps that describe the bits in each of the configuration table flags.

MModeFlags

Table 4.2
***MModeFlags* Bit Map**
Offset 22h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			0	0	0	0	0	1					0		1
Default Values															

***MModeFlags* Bit Map**

Bit #	Description
0	*Reserved*. Set this bit to 1 for backward compatibility.
1	*UsesDMABit*. The HSM sets this bit if it uses DMA or bus-mastering.
2	*Reserved*. Set to 0.
3	*MulticastBit*. The HSM sets this bit if it supports multicast addressing. The HSM must support multicast addressing, if the hardware supports it.
4	*PointToPointBit*. Set this bit to allow the HSM to bind with a protocol stack without providing a network number. No network number exists in point-to-point connections. The HSM must set this bit if the HSM supports dynamic call setup or teardown. Typically, asynchronous or X.25 HSMs set this bit.
5	*NeedsPollingBit*. Setting this bit causes the system to call the HSM every timer tick (55 ms) and whenever a protocol stack relinquishes control. Only HSMs that do not have interrupt capabilities use this bit. Do not use this feature to implement a watchdog function; instead, use the *ScheduleAESEvent* function available through the LSL. (See Chapter 7).
6	*RawSend*. The HSM sets this bit to 1 if it supports raw sends.
7	*Reserved*. Set to 1 for backward compatibility.
8	*Reserved*. Set to 0.
9	*Reserved*. Set to 0.
10	*Reserved*. Set to 0.
11	*Reserved*. Set to 0.
12	*Reserved*. Set to 0.
13	*PromiscuousModeBit*. The HSM sets this bit if it supports promiscuous mode.

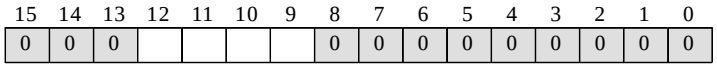
***MModeFlags* Bit Map** *(continued)*

Bit #	Description
15, 14	The MLID sets these bits to indicate whether the *MNodeAddress* field of the configuration table contains a canonical or a noncanonical address. Bit 15 This bit indicates whether the node address format is configurable. Bit 14 indicates whether the configuration table *MNodeAddress* field contains the node address in canonical or noncanonical form. The state of bit 14 is only defined when bit 15 is set. The bit 15/bit 14 combinations are: 00 = *MNodeAddress* format is unspecified. The node address is assumed to be in the physical layer's native format. 01 = This is an illegal value and must not occur. 10 = *MNodeAddress* is canonical. 11 = *MNodeAddress* is noncanonical. (See the *ODI Specification Supplement: Canonical and Noncanonical Addressing*.)

MFlags

The HSM sets the bits in this field to indicate different support mechanisms, such as multicast filtering and multicast address format.

Table 4.3
***MFlags* Bit Map**
Offset 52h



Default Values

MFlags Bit Map	
Bit #	**Description**
0	*Reserved*. Set to 0.
1	*Reserved*. Set to 0.
2	*Reserved*. Set to 0.
3	*Reserved*. Set to 0.
4	*Reserved*. Set to 0.
5	*Reserved*. Set to 0.
6	*Reserved*. Set to 0.
7	*Reserved*. Set to 0.
8	*Reserved*. Set to 0.

***MFlags* Bit Map** *(continued)*

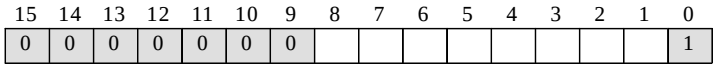
Bit #	Description
10,9	These bits indicate different support mechanisms for multicast filtering. These bits are only valid if bit 3 of the *MModeFlags* is set, indicating that the HSM supports multicast addressing. The HSM sets bit 10 if it has specialized adapter hardware (such as hardware that utilizes CAM memory). **Note:** If an HSM that usually defaults to using functional addresses also supports group addressing and sets bit 10, it receives both functional and group addresses. The state of bit 9 is defined only if bit 10 is set. Bit 9 is set if the adapter completely filters group addresses and the TSM does not need to perform any checking. The HSM can dynamically set and clear bit 9. For example, if the adapter utilizes CAM memory, but has temporarily run out of memory, the TSM must temporarily filter the group addresses. In this case, the HSM would reset bit 9. The bit 10/bit 9 combinations are: 00 = The format of the multicast address defaults to that of the topology: Ethernet => Multicast addressing, in other words, Group addressing Token-Ring => Functional addressing/Group addressing FDDI => Group addressing 01 = Illegal value and must not occur. 10 = A specialized adapter supports group addressing, but the TSM filters the addresses. 11 = A specialized adapter supports group addressing, and the TSM is not required to filter the addresses. (See the *ODI Specification Supplement: Canonical and Noncanonical Addressing*.)
11	NESL_REQUIRED_BIT. If this bit is set, the LAN driver requires the NetWare Event Service Layer. For more information about the NetWare Event Service Layer (NESL), see the *NESL Specification: 16-Bit DOS Client Programmer's Interface.*
12	*PrioritySupportBit*. The MSM sets this bit during the *MSMMediaConfigUpdate* routine, if the following conditions are met: • The HSM has provided a pointer to the *DriverPriorityQSupport* routine. • The HSM has set *MPrioritySup* to something other than zero. **Note:** The HSM may temporarily clear this bit to disable priority support.
13	*Reserved*. Set to 0.
14	*Reserved*. Set to 0.
15	*Reserved*. Set to 0.

MSharingFlags

This field informs the system which hardware resources a driver/LAN adapter can share with other driver/LAN adapters. The first bit indicates when the HSM is shutdown. The MSM sets and clears this bit. If the HSM supports shareable interrupts, it must set the *CanShareIRQ* bit.

Note The HSM initially sets the shutdown bit to 1, as the driver is shut down. ▲

Table 4.4
***MSharingFlags* Bit Map**
Offset 5Ah



Default Values

MSharingFlags Bit Map	
Bit #	**Description**
0	*ShutDownBit*. Set to 1 if the LAN adapter is currently shut down.
1	*CanShareIO1*. Set to 1 if the LAN adapter can share I/O port 1.
2	*CanShareIO2*. Set to 1 if the LAN adapter can share I/O port 2.
3	*CanShareMemory1*. Set to 1 if the LAN adapter can share memory range 1.
4	*CanShareMemory2*. Set to 1 if the LAN adapter can share memory range 2.
5	*CanShareIRQ1*. Set to 1 if the LAN adapter can share interrupt 1.
6	*CanShareIRQ2*. Set to 1 if the LAN adapter can share interrupt 2.
7	*CanShareDMA1*. Set to 1 if the LAN adapter can share DMA channel 1.
8	*CanShareDMA2*. Set to 1 if the LAN adapter can share DMA channel 2.
9	*Reserved*. Set to 0.
10	*Reserved*. Set to 0.
11	*Reserved*. Set to 0.
12	*Reserved*. Set to 0.
13	*Reserved*. Set to 0.
14	*Reserved*. Set to 0.
15	*Reserved*. Set to 0.

Adapter Data Space

The adapter data space contains data specific to a particular LAN adapter. This space contains the statistics table and other internal hardware-specific variables.

Statistics Table

Code segment

The ODI specification requires the HSM to define one statistics table that contains network management information. The table must be defined by the fields shown below with each entry filled accordingly.

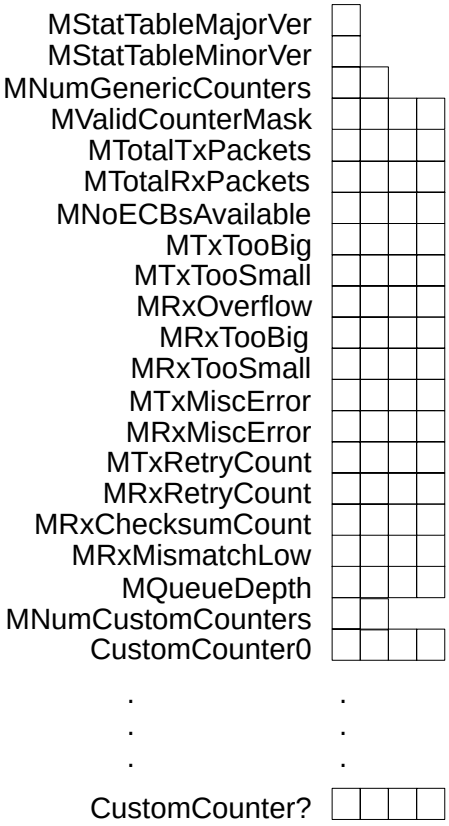
The following figures and tables contains the field names, descriptions, and other necessary information about the statistics table.

Figure 4.3
MLID Statistics Table
Sample Source Code

```
MLIDStatStructure       struc
   MStatTableMajorVer       db  01
   MStatTableMinorVer       db  01        ;1.01
   MNumGenericCounters      dw  14
   MValidCounterMask        dd  ?         ;0 = used, 1 = not used
   MTotalTxPackets          dw  2 dup (0)
   MTotalRxPackets          dw  2 dup (0)
   MNoECBsAvailable         dw  2 dup (0)
   MTxTooBig                dw  2 dup (0)
   MTxTooSmall              dw  2 dup (0)
   MRxOverflow              dw  2 dup (0)
   MRxTooBig                dw  2 dup (0)
   MRxTooSmall              dw  2 dup (0)
   MTxMiscError             dw  2 dup (0)
   MRxMiscError             dw  2 dup (0)
   MTxRetryCount            dw  2 dup (0)
   MRxChecksumError         dw  2 dup (0)
   MRxMismatchError         dw  2 dup (0)
   MQueueDepth              dw  2 dup (0)
   MNumCustomCounters       dw  ?
   CustomCounter0           dd  0
         .
         .
   CustomCounter?           dd  0
                            dw  offset CGroup:CustomCounterStr0
                            dw  segment CGroup:CustomCounterStr0
                             .
                             .
                            dw  offset CGroup:CustomCounterStr?
                            dw  segment CGroup:CustomCounterStr?
MLIDStatStructure       ends
Message    CustomCounterStr0      'Custom Counter Text for Counter 0'
   .
   .
Message    CustomCounterStr?      'Custom Counter Text for Counter ?'
```

Note *Message* is a macro that length-prepends and zero-terminates a text string. ▲

Figure 4.4
Graphic Representation of the MLID Statistics Table



Description of HSM Statistics Table Fields

The following table describes the fields of the HSM statistics table.

Table 4.5
Statistics Table Field Descriptions

HSM Statistics Table			
Offset	**Name**	**Size (in bytes)**	**Description**
00h	MStatTableMajorVer	1	This field defines the current major version of the statistics table. For this specification, set this field to 1.
01h	MStatTableMinorVer	1	This field defines the current minor version of the statistics table. For this specification, set this field to 01. (The current version of the statistics table is 1.01; *MDriverStatMajorVer* = 1, *MDriverStatMinorVer* = 01.)
02h	MNumGenericCounters	2	This field defines the number of generic counters defined in the statistics table. Currently this number is 14.
04h	MValidCounterMask	4	This bit field is used to signal which generic counters the HSM is actually using. The bit field is 32 bits long and the most significant bit corresponds to the first generic counter, *MTotalTxCount*. A bit value of 1 disables the counter; a bit value of 0 enables the counter.
08h	MTotalTxPackets	4	This field contains the total number of packets that the HSM requested to transmit. Whether or not they were actually transmitted depends upon the HSM. The MSM increments this field.
0Eh	MTotalRxPackets	4	This field contains the total number of incoming packets for which the HSM received an RCB. The MSM increments this field.
10h	MNoECBsAvailable	4	This field is used to count the number of incoming packets that were not received or wanted either because no RCBs were available, or because the protocol stack did not want the packets. The MSM increments this field.
14h	MTxTooBig	4	This field has the number of requested packets for transmission that were too big to send. Normally, the HSM does not use this field. The MSM increments this field.

HSM Statistics Table *(continued)*

Offset	Name	Size (in bytes)	Description
18h	MTxTooSmall	4	This field contains the number of requested packets for transmission that were normally too small to be transmitted. The packets might still have been sent if the HSM does padding. Normally, the HSM does not use this field.
1Ch	MRxOverflow	4	The HSM increments this field when the LAN adapter runs out of internal receive buffers.
20h	MRxTooBig	4	This field has the number of incoming packets that were bigger than the value in *MMaxPacketSize*. The MSM increments this field.
24h	MRxTooSmall	4	This field contains the number of incoming packets that were smaller than the minimum legal size for the media. The MSM increments this field.
28h	MTxMiscError	4	This field contains the number of transmission requests that were not sent because of errors other than those explicitly listed in this table.
2Ch	MRxMiscError	4	This field has the number of incoming packets that were lost because of errors other than those explicitly listed in this table.
30h	MTxRetryCount	4	The HSM increments this field when the HSM must retransmit because of a hardware failure (for example, too many collisions).
34h	MRxChecksumError	4	This field has the total number of incoming packets which were lost due to checksum/CRC errors.
38h	MRxMismatchError	4	This field contains the total number of incoming packets which were lost due to conflicting information given by the hardware and the media-specific header.
3Ch	MQueueDepth	4	The MSM increments this field whenever it queues a packet. It decrements this field whenever it removes a transmit packet from the transmit queue.

HSM Statistics Table *(continued)*

Offset	Name	Size (in bytes)	Description
41h	MNumCustomCounters	2	This field has the total number of custom variables which follow this WORD. The statistics table allows the HSM to define a number of custom counters. These counters are HSM specific and can count any interesting event which you think would be useful for a system administrator. Each custom counter allows you to define a corresponding descriptive text string that is length-preceded and zero-terminated. Keep the number of custom counters to a minimum to conserve DOS memory. When a custom counter corresponding to a receive or transmit error event is incremented, the HSM also increments the appropriate *MiscCount* counter (for example, *TxMiscCount*). The miscellaneous counters total all custom defined error events. We recommend that counter increments be done as follows: add RxOverflowCount+0, 1 adc RxOverflowCount+2, 0 **Note:** You can define other counters for debugging purposes, but remove them when the driver is shipped to an end user.

Structures Provided by the MSM

AES Event Control Block (AES ECB)

Structure

The LSL support call *ScheduleAESEvent* uses AES Event Control Blocks (AES ECBs) to schedule a driver-defined event to occur at the end of a specified time interval. Below is a description of each field in the AES ECB.

```
AESECBStruc     struc
    AESLink         dd  0
    MSecondValue    dd  0
    AESStatus       dw  0
    AESESR          dd  0
AESECBStruc     ends
```

Table 4.6 AES ECB Field Descriptions

AES ECB			
Offset	**Name**	**Size (in bytes)**	**Description**
00h	AESLink	4	The LSL uses this field to manage a linked list of AES ECBs. The HSM cannot modify this field while the LSL controls the AES ECB.
04h	MSecondValue	4	The HSM sets this field when calling *ScheduleAESEvent*. This field defines the number of milliseconds the LSL will wait before calling the event service routine defined in the *AESESR* field. The value in this field will be invalid upon event completion or cancellation.
08h	AESStatus	2	The HSM cannot modify this field while the LSL controls the AES ECB. This field is set to 0 when the specified timeout has expired and the AES (Event Service Routine) ESR is called.
0Ah	AESESR	4	This field contains a far pointer to an HSM routine that will be called at the end of the specified time interval. This field must contain the address of a valid routine. The LSL does not modify this field.

Receive Control Block (RCB)

Structure

The HSM uses Receive Control Blocks (RCBs) to transfer data to the upper layers of the ODI model. Each RCB can contain pointers to multiple fragments. Once notified of a received packet, the HSM obtains an RCB from the MSM and fills in the fragmented buffers. The RCB is passed back to the MSM to be processed. Below is a description of each field in the RCB.

```
RCBStructure      struc
    RCBDriverWS         db   8  dup (0)
    RCBReserved         db 36  dup (0)
    RCBFragCount        dw  0
    RCBFrag0Addr        dd  0
    RCBFrag0Len         dw  0
RCBStructure      ends

;Additional fragments for an RCBFragCount > 1
    RCBFrag?Addr    dd  0
    RCBFrag?LEN     dw  0
```

Table 4.7
Receive Control Block Field Descriptions

Receive Control Block (RCB)			
Offset	**Name**	**Size (in bytes)**	**Description**
00h	RCBDriverWS	8	The HSM can use this field for any purpose.
08h	RCBReserved	36	The HSM must not modify this field.
2Ch	RCBFragCount	2	This field has the number of fragment descriptors that follow; this number cannot be 0.
30h	RCBFrag0Addr	4	This field contains a far pointer to a receive buffer fragment.
32h	RCBFrag0Len	2	This field contains the length of the receive buffer pointed to by the previous field. This field can be 0.
?	RCBFrag?Addr RCBFrag?Len	4 2	The fragment structure (address and length) is repeated for additional fragments if RCBFragCount > 1.

Transmit Control Block (TCB)

Structure

The MSM constructs a TCB to describe the data it receives from a protocol stack. The TCB structure includes a pointer to a separate *FragmentStructure* as well as the entire media header. The MSM gives the TCB to the HSM, which collects the header and packet fragments and transmits the packet. Below are descriptions of the fields in the TCB and *FragmentStructure*.

```
TCBStructure        struc
    TCBDriverWS         db  6 dup (0)
    TCBDataLength       dw  0
    TCBFragStrucPtr     dd  0
    TCBMediaHeaderLen   dw  0
    TCBMediaHeader      db  0?
```

Table 4.8 Transmit Control Block Field Descriptions

Transmit Control Block (TCB)			
Offset	**Name**	**Size (in bytes)**	**Description**
00h	TCBDriverWS	6	The HSM can use this field for any purpose.
06h	TCBDataLength	2	This field contains the length of the frame, as described by the data fragments, plus the media header. This value will never be 0.
08h	TCBFragStrucPtr	4	This field contains a far pointer to a list of fragments as defined by the *FragmentStructure* below.
0Ch	TCBMediaHeaderLen	2	This field is the length of the media header which follows this field in memory. This value can have a value of 0.
0Eh	TCBMediaHeader	1	This is the start of the media header (the media header buffer is part of the TCB).

The *FragmentStructure* is defined as follows:

```
FragmentStructure       struc
    FFragmentCount      dw  0   ;number of fragment descriptors
    FFrag0Address       dd  0   ;1st fragment buffer
    FFrag0Length        dw  0   ;1st fragment buffer length
FragmentStructure       ends

;Additional fragments for FFragment Count > 1
    FFrag?Address       dd  0
    FFrag?Length        dw  0
```

Table 4.9
Fragment Structure
Field Descriptions

Fragment Structure			
Offset	**Name**	**Size (in bytes)**	**Description**
00h	FFragmentCount	2	This field contains the number of fragment descriptors following this field. This field cannot be set to 0.
02h	FFrag0Address	4	This field contains a far pointer to a buffer that contains part of the frame data.
06h	FFrag0Length	2	This field contains the length of the buffer that was pointed to by the previous field. This can be set to 0.
08h	FFrag?Address FFrag?Len	4 2	The fragment structure (address and length) is repeated for additional fragments if FFragmentCount > 1.

❑

Chapter 5 Developer-Written HSM Routines

Chapter Overview

This chapter describes the routines you must write for your HSM. The following table lists the routines described in this chapter.

Table 5.1 Summary of HSM Routines

Summary of HSM Routines	
Summary	**Function Name**
Routines that must be fully written.	
Initializing the HSM.	DriverInit
Resetting the hardware.	DriverReset
Sending packets.	DriverSend
Servicing interrupts and receiving packets.	DriverISR
Shutting down the hardware.	DriverShutdown
Updating the multicast table (Can be stubbed with a near ret if the hardware does not support it).	DriverMulticastChange
Routines that can be stubbed. (These routines must exist so that the HSM will link properly with the MSM.)	
Changing promiscuous mode.	DriverPromiscuousChange
Changing the look-ahead size.	DriverChangeLookAheadSize
Allows other modules to manage the HSM and/or hardware.	DriverManagement
Polling the driver.	DriverPoll

You should review this chapter before writing these HSM routines.

DriverChangeLookAheadSize

Located in the Code segment
Can be stubbed with a near ret

Description

Informs the HSM of an increase to the look-ahead size.

Entry State

DS
CGroup

CX
NewLookAheadSize

Interrupts
are disabled.

Return State

Preserved
DX, SI, ES, SS, SP.

Interrupts
are disabled.

Remarks

The MSM calls *DriverChangeLookAheadSize* to inform the HSM that the look-ahead size has increased. HSMs that can program the LAN adapter to have the *LookAheadSize* plus *MaxHeaderSize* number of bytes to be available prior to the ISR can utilize this function to adjust the hardware setting. Pipelined LAN adapters are good examples of HSMs that would call this routine.

DriverInit

Located in the Init segment
Must be fully written

Description

Initializes the LAN adapter.

Entry State

DS
CGroup.

ES
CGroup.

Interrupts
are enabled.

Note
CLD is in effect.

Return State

AX
completion code.

DX
pointer to a zero-terminated error message, if AX is nonzero.

Interrupts
are disabled.

Flags
set according to AX.

Preserved
no other flags or registers.

Completion Codes (AX)

0000h *MSM_SUCCESSFUL*
The hardware and HSM are operational.

Completion codes 0001 – 0004 are accompanied by the offset of a zero-terminated error message that describes the problem in DX.

0001h *MSMERR_HSM_FAILED*
One of the following conditions exist:
- The hardware is present, but it failed to initialize
- The HSM experienced a fatal error.

0002h *MSMERR_TSM_FAILED*
The TSM found a problem and didn't load.

0003h *MSMERR_MSM_FAILED*
The MSM found a problem and didn't load.

0004h *MSMERR_CARD_NOT_PRESENT*
The HSM initialized, but the LAN adapter is missing. PCMCIA drivers are the most likely to use this completion code.

DriverInit *continued*

Remarks

The MSM calls *DriverInit* during system initialization.

Important If your HSM supports PCMCIA adapters, refer to Appendix C for information about *DriverInit*. ▲

The HSM must initialize the hardware before it can send and receive packets on the network. It is during initialization, that the HSM brings the hardware into operation. If a nonfatal error occurs, the HSM can use the *MSMPrintStringZero* routine (see Chapter 6) to display an appropriate warning message to the user.

Note The HSM can check MSMSystemFlags to find information regarding the environment it is initializing in. ▲

Setting the Variables

The HSM sets the *MSMTxFreeCount* variable to the maximum number of transmits the HSM can process at one time (see Chapter 4). The HSM also sets the *MSMLookAheadSegment* word variable to the value of the segment the receive look-ahead data is provided in. Shared memory LAN adapters set this variable to the LAN adapter's shared memory segment address. If the LAN adapter does not use shared memory, this value is usually the driver's CS value.

The HSM sets *MSMPriorityQueueSupport* with a near pointer to an HSM priority queue support function. (See Appendix D.) If the *MSMPhysNodeAddress* variable was previously set to FFFFFFFFFFFFh, the HSM sets it to the default hardware address. Otherwise, the HSM programs the LAN adapter to the node address override.

Initializing the HSM Configuration Table Hardware Options

The HSM initializes the hardware to the values in the configuration table, unless the hardware is capable of communicating different values to the HSM. HSMs that drive ISA-type LAN adapters should validate the configuration table values, if they can.

If the user desires settings other than the default, he/she should specify those settings in the NET.CFG file. The MSM reads the new settings from the NET.CFG file and then sets those values into the HSM's configuration table before the MSM calls *DriverInit*.

If the hardware or system is capable of communicating parameters to the HSM, the HSM should write those values to

DriverInit *continued*

the HSM configuration table. In all cases, the HSM ensures that the settings in the configuration table mirror the settings that the LAN adapter is initialized to.


Important After the HSM has initialized the variables and the configuration table, it must call *MSMMediaConfigUpdate* to inform the MSM. ▲

Scanning the Slots. If the HSM drives a Micro Channel, PCI, or EISA type LAN adapter, it can read the configuration information from the system. Usually, the HSM scans the slots and uses the first LAN adapter of the correct type it finds.

If the user fills in the NET.CFG *Slot* keyword (see Appendix B) to specify which machine slot the LAN driver is to use.

The HSM checks the value in the *Slot* field and responds as follows:

0	the HSM scans the slots to find a LAN adapter
Nonzero	the HSM uses the number it finds in the *Slot* field as the LAN adapter's slot number. The HSM checks this slot for a LAN adapter and does not scan.

If the HSM does not find a LAN adapter in the specified slot, it aborts the initialization process and returns an appropriate error message. If the HSM finds a LAN adapter, it reads the hardware configuration and places the values in the HSM configuration table.

Supporting Multiple Bus Types. If the HSM supports multiple bus types, it checks the driver configuration table's *MBusID* field to determine whether the user specified in the NET.CFG file which bus the adapter is located in.

If *MBusID* is set to –1, the HSM searches each of the machine's busses for a supported LAN adapter and initializes the first LAN adapter it finds. The HSM determines the order to search the busses in.

If *MBusID* is initially set to default, the HSM sets it to the appropriate Bus ID.

Note If the user has set the *BUS ID* keyword for an unsupported bus, the HSM should exit *DriverInit* and return an error. (For information about the *Bus ID* keyword, see Appendix B.) ▲

DriverInit *continued*

Overriding the Node Address. The user can also override the node address and assign a locally administered node address for the LAN adapter to replace its hard-coded node address. The following table outlines this procedure.

Table 5.2
Specifying an Alternate Node Address

Specifying an Alternate Node Address	
Module	**Action**
NET.CFG	1. Specifies a locally administered address with the *Node Address* keyword (see Appendix B).
MSM	2. Adjusts the address specified in step 1 for LSB or MSB format. 3. Changes the *MSMPhysNodeAddress* variable's default value, FFFFFFFFFFFFh, to the address specified in step 1. (For the node address 02008C001600L in an Ethernet environment, *MSMPhysNodeAddress* is set to 02008C001600h; in a token-ring MSB environment *MSMPhysNodeAddress* is set to 400031006800h.) 4. Calls *DriverInit*.
HSM	5. Checks the value of the *MSMPhysNodeAddress* variable. 6. Executes one of the following steps, depending on the value found in step 5. a) If the *MSMPhysNodeAddress* is equal to FFFFFFFFFFFFh, sets *MSMPhysNodeAddress* to the LAN adapter's hard-coded address. The HSM now uses the LAN adapter's hard-coded address as the node address. b) If *MSMPhysNodeAddress* is not equal to FFFFFFFFFFFFh, sets the LAN adapter's node address to the address provided in *MSMPhysNodeAddress*.
MSM	7. Sets the HSM configuration table *NodeAddress* field to the LSB/MSB adjusted value found in *MSMPhysNodeAddress*.

If the HSM does not allow the user to override the node address, it ignores the value already in *MSMPhysNodeAddress* and sets it to the LAN adapter's hard-coded address. The HSM then notifies the user that it has ignored the node address override. If the user has not specified a locally administered address, the HSM sets the *MSMPhysNodeAddress* variable to the LAN adapter's hard-coded address before it returns from the initialization routine.

DriverInit *continued*

Setting the Interrupt Vector. Once the HSM and adapter are ready to service interrupts, the HSM can use the MSM support routine *MSMSetIRQ* to set up the HSM interrupt

vector specified in the configuration table *IRQLine1* field. If the HSM sets the vector and then fails to initialize the LAN adapter, the HSM removes the vector by using *MSMUnSetIRQ* (see Chapter 6).

Interrupts in Timing Loops. Remember to enable the interrupts when using the LSL *GetIntervalMarker* routine for timing loops. This routine's return value will never change if interrupts are disabled (see Chapter 7).

Completing Initializing

After the LAN adapter successfully initializes, *DriverInit* either calls or jumps to the *MSMMediaConfigUpdate* routine.

All memory containing code and data in the Init portion of the segment is freed after returning from *DriverInit*. If the HSM returns successfully, the MSM installs the driver as a DOS terminate and stay resident (TSR) program.

Providing Netware Event Service Layer (NESL) Events

If the HSM provides NetWare Event Service Layer (NESL) events, it must register with the NESL any event classes that the MSM does not provide. The MSM registers the following event classes with the NESL:

- Suspend Notification
- Resume Notification
- Service/Status Change

See Also

Configuration Table (Chapter 4)
MSMCallNESL (Chapter 6)
MSMLookAheadSegment (Chapter 6)
MSMMediaConfigUpdate (Chapter 6)
MSMPhysNodeAddress (Chapter 4)
MSMPrintStringZero (Chapter 6)
MSMPriorityQSupport (Chapter 4)
MSMSetIRQ (Chapter 6)
MSMSystemFlags (Chapter 4)
MSMTxFreeCount (Chapter 4)
MSMUnSetIRQ (Chapter 6)
Appendix C (for information on PCMCIA support)
Appendix D (for information on priority queues)

DriverInit: An Outline

This pseudocode is intended to illustrate a flow of events and does not necessarily describe complete or optimized code.

Disable system interrupts (CLI).

Set the *MSMLookAheadSegment* variable to the appropriate segment value.
Set the *MSMTxFreeCount* variable to the appropriate value. *(determined by the hardware)*

IF using a LAN adapter in a slot aware bus,
    IF the LAN adapter is present in one of the slots,
        get the slot number,
        interpret the bus configuration table.
    ELSE
        set DX to point to error string & AX = MSMERR_HSM_FAILED,
        return
    ENDIF
ENDIF

IF there is not a node address override,
    read in the node address from the LAN adapter,
    copy the node address to the *MSMPhysNodeAddress* variable.
ELSE
    use the node address already present in the *MSMPhysNodeAddress* variable to program the adapter
ENDIF

Initialize the LAN adapter. *(this is the bulk of the INIT)*

IF there was an error initializing the LAN adapter,
    set DX to point to error string & AX = MSMERR_HSM_FAILED
    return
ENDIF

IF necessary,
    setup DMA channels
ENDIF

IF the driver is interrupt driven, install interrupt vector,
    CALL *MSMSetIRQ*
ENDIF

IF the driver has need of starting an AES event, *(typically used to timeout transmits)*
    set up the AES block,
    move MLIDSUP_SCHEDULE_AES_EVENT into BX,
    CALL *LSLSupport*.
ENDIF

CALL *MSMMediaConfigUpdate*

Return (return code from *MSMMediaConfigUpdate*)

DriverISR

Located in the Code segment
Must be fully written

Description

Handles transmission and reception interrupts.

Entry State

DS
CGroup.

Interrupts
are disabled, but might be enabled by HSM.

Note
CLD is in effect.

Note
A critical section has been started.

Return State

DS
CGroup

Interrupts
are disabled.

Preserved
SS, SP.

Remarks

The MSM gains control and calls *DriverISR* whenever the LAN adapter interrupts the system. The MSM preserves all 16-bit processor registers, sets DS equal to CGroup, enters a critical section, and then calls *DriverISR*.

Important The system interrupts are enabled for the majority of *DriverISR*. ▲

Note Pipelined adapters use a slightly different reception process. The specifics of this process are described later in this section. ▲

Entering *DriverISR*

When the HSM enters *DriverISR*, it must do the following:

1. Disable the LAN adapter's interrupt line;
2. Use the *MSMEOIFlag* global variable to determine whether to issue an EOI to the slave PIC;
3. Issue an EOI to the slave PIC (I/O Port 0A0h), if the HSM determined in step 2 that it should;
4. Issued an EOI to the master PIC (I/O Port 20h);

DriverISR *continued*

5. Enable system interrupts (STI);
6. Determine why the interrupt occurred;
7. Service the interrupt.

Leaving DriverISR

Before leaving *DriverISR*, the HSM must do the following:

1. Disables system interrupts (CLI);
2. Enables the LAN adapter's interrupt line.

Disabling the Interrupt

The HSM disables the LAN adapter's interrupt line using one of the methods described below.

Preferred Method. Disable the LAN adapter's interrupt line by using the interrupt mask port.

Alternate Method. Mask the LAN adapter's interrupt line by using the PIC mask register, if the LAN adapter does not have an interrupt mask register. For example, the HSM can execute the following code to disable the adapter's interrupt at the PIC:

```
;Disable LAN adapter's interrupt line
    mov  dx, MSMIntMaskPort
    in   al, dx             ;Read current mask value
    slow                    ;Add at least 500ns delay
    or   al, MSMIntMaskOff
    out  dx, al             ;Output new mask value
```

Enabling the Interrupt

To enable interrupts, the HSM reverses the process it used to disable them. In other words, if the HSM disabled the interrupts with the interrupt mask port, it enables interrupts with the interrupt mask port. And, if the HSM disabled interrupts with the PIC mask register, it enables interrupts with the PIC mask register.

The HSM can execute the following code to enable the adapter's interrupt at the PIC:

```
;Enable LAN adapter's interrupt line
    mov  dx, MSMIntMaskPort
    in   al, dx             ;Read current mask value
    slow                    ;Add at least 500ns delay
    and  al, MSMIntMaskOn
    out  dx, al             ;Output new mask value
```

The slow macro is defined as follows:

DriverISR *continued*

```
slow    macro
        push        ax
        in    al, 61h
        in    al, 61h
        in    al, 61h
        pop ax
        endm
```

Note The MSM has simplified PIC interrupt disabling/enabling by providing the developer with the following 3 variables:

- *MSMIntMaskOff*
- *MSMIntMaskOn*
- *MSMIntMaskPort*

(See Chapter 4 for instructions on using these variables.) ▲

Processing the Completed Event

The LAN adapter's interrupt request invokes the *DriverISR* whenever the LAN adapter has finished processing event(s). *Transmission* and *Reception Complete* are examples of events that require the *DriverISR* routine..

Transmission Complete Event

A transmission complete event occurs whenever the LAN adapter finishes executing a transmission event started by the HSM's *DriverSend* routine. To handle a transmission complete, the HSM

1. Checks for transmission errors and handles accordingly;
2. Calls *MSMSendComplete* to return any TCBs not previously returned during *DriverSend* (see Chapter 6);
3. Increments the value in the *MSMTxFreeCount* variable;
4. Calls *MSMGetNextSend* to check for pending TCBs (see Chapter 6);
5. Calls *DriverSend* to send the frame, if pending TCBs exist.

The MSM internally uses the *MSMTxFreeCount* variable to determine whether the HSM can handle another transmission request. The MSM decrements this variable every time it gives a *Transmit Control Block* (TCB) to the HSM. Because the MSM cannot tell when a hardware transmit resource is freed, the HSM increments the *MSMTxFreeCount* variable whenever a transmit resource becomes available. When the HSM receives an unrecoverable error, it resets the internal hardware and then sets the *MSMTxFreeCount* variable back to the initial value.

DriverISR *continued*

Reception Complete Event

A reception complete event occurs whenever the LAN adapter accepts a packet from the LAN medium. The HSM uses receive look-ahead data in order to receive a packet. When the HSM receives a packet, it

1. Uses the value in the configuration table *LookAheadSize* field (plus the *MaxMediaHeaderSize*) to determine how much look-ahead data to provide when it asks the MSM for a receive buffer;
2. Calls *MSMGetRCB* (see Chapter 6) with a pointer to a look-ahead buffer holding the first *n* bytes of the packet (see note below);
3. Chooses one of the following actions:
   a. If an RCB is returned, copies the received packet from the LAN adapter into the RCB,
   b. If an RCB is not returned, discards the packet,
   c. If it cannot fill in the RCB, calls *MSMReturnRCB*;
4. Calls either *MSMRcvComplete* or *MSMRcvCompleteStatus* to hand the RCB to the MSM (see Chapter 6). The MSM then completes the receive operation.

Note *n* in step 2 represents the amount of look-ahead data plus the maximum media header size. The amount of look-ahead data can be configured and can be any value between 18 and 128 bytes. The HSM always uses the current look-ahead size. ▲

The following table outlines the steps accomplished by the call to *MSMGetRCB*.

Table 5.3
Actions Accomplished by *MSMGetRCB*

Actions Accomplished by *MSMGetRCB*	
Module	**Method**
MSM	1. Builds a *LookAhead* structure which points to the *LookAhead* buffer. 2. Passes the *LookAhead* structure to the LSL.
LSL	3. Passes the *LookAhead* structure to the appropriate prescan, bound, or default stack for the LAN adapter.

Actions Accomplished by ***MSMGetRCB*** *(continued)*	
Module	**Method**
Protocol Stack	4. Interrogates the *LookAhead* structure to determine if the protocol stack should accept the packet. If the protocol stack decides to receive the packet: 5. Determines which receive buffer to move the packet into. 6. Returns to the MSM an event control block defining which buffers to move the packet into.
MSM	7. Gives the HSM an RCB to fill in, if one is available.

Determining the Amount of Look-Ahead Data

The HSM's configuration table *LookAheadSize* field has a default value of 18. However, this value can change at any time. Therefore, the HSM must either reference this value for every frame it receives or implement *DriverChangeLookAheadSize* to ensure that *LookAheadSize* number of bytes + max headers are available for every frame. (*DriverChangeLookAheadSize* is documented earlier in this chapter.)

MaxMediaHeaderSize for Ethernet = (14 [MAC] + 3 [802.2 UI] + 5 [SNAP] = 22 bytes)

MaxMediaHeaderSize for Token-Ring = (14 [MAC] + 30 [Source Routing] + 3 [802.2 UI] + 5 [SNAP] = 52 bytes).

MaxMediaHeaderSize for FDDI = (13 [MAC] + 30 [Source Routing] + 3 [802.2 UI] + 5 [SNAP] = 51 bytes).

Methods of Providing Look-Ahead Data

The HSM must provide the look ahead data before the MSM can obtain a receive buffer. Several methods exist for moving receive data from the LAN adapter to host memory. This document describes the following three methods.:

- Shared Memory
- Programmed I/O
- DMA

The process of providing look ahead data differs with each method.

Shared Memory. This is the simplest method of providing look ahead data. The HSM provides a pointer to the start of the received packet in shared memory.

DriverISR *continued*

Programmed I/O. To provide look ahead data, the HSM copies the first portion of packet data into an internal buffer and then points to this buffer. If the MSM returns a receive buffer to the HSM, the HSM resets the LAN adapter's internal RAM pointer to the start of the packet plus the offset value returned by the MSM. The HSM then copies the data from the data port into the receive buffers.

DMA. For this method to work, the HSM must have an internal receive buffer large enough to hold the largest possible packet to be received. After the HSM has placed the data into the internal buffer, the HSM simply points to that buffer.

Packet Reception for Pipelined Adapters

Pipelined adapters can be configured to interrupt prior to receiving a complete packet. In this case, the adapter must be able to be configured to wait until it has received at least the *LookAheadSize* and the *MaxHeaderSize* before it interrupts. These HSMs call *MSMGetRCB* with BP=–1 before they have received the entire packet. Therefore, the MSM cannot fill in all the *LookAhead* fields with definitive values. The error bits and length fields are set at an initial best guess until the packet has been completely received. After the RCB is completely filled in, pipelined adapters return it by calling *MSMRcvCompleteStatus*.

See Also

Configuration Table (Chapter 4)
MSMEOIFlag (Chapter 4)
MSMGetRCB (Chapter 6)
MSMIntMaskOff (Chapter 4)
MSMIntMaskOn (Chapter 4)
MSMIntMaskPort (Chapter 4)
MSMRcvComplete (Chapter 6)
MSMRcvCompleteStatus (Chapter 6)
MSMTxFreeCount (Chapter 4)

DriverISR: An Outline

This pseudocode is intended to illustrate a flow of events and does not necessarily describe complete or optimized code.

Make sure that all necessary actions are taken to service the LAN adapter hardware, reset its IRQ line, and
prepare it to generate further interrupts. Program carefully so that no hardware events are overlooked.

```
Disable LAN adapter's interrupts.
Issue EOI to interrupt controller(s).          use the MSMEOIFlag variable
Enable system interrupts (STI).                run with interrupts enabled

LoopOnReceive:

Get the LAN adapter Status.

IF receive event,
    check for receive errors if necessary.
    put pointer to LookAhead frame area in ES:SI,
    get the hardware reported frame length and put in BP,
    CALL MSMGetRCB

    IF RCB is NOT available.
        goto TransmitEvent.
    ENDIF

    copy data into the RCB.
    CALL MSMRcvComplete
goto LoopOnReceive.

TransmitEvent:
ELSE IF transmit event,
    IF transmit error,
        decrement retry counter

        IF retry count = 0,
            quit trying to send the packet and continue.
        ELSE increment error counter,
            mark start time of current send,
            attempt to send the packet again,
            goto ExitDriverISR.
        ENDIF
    ENDIF

    reset retry counter to Maximum value.
    increment MSMTxFreeCount.
    return TCB  if it has not already been returned.
    disable system interrupts (CLI),
    CALL MSMGetNextSend
    enable system interrupts (STI),

    IF TCB was available,
        CALL to InternalDriverSend procedure.
    ENDIF
ENDIF

ExitDriverISR:
    Disable system interrupts (CLI).
    Enable LAN adapter's interrupts.
Return.                                        Don't IRET
```

DriverManagement

Located in the Code segment
Can be stubbed with a near ret.

Description

Allows other modules to manage the HSM and or LAN adapter.

Entry State

AX
LSLERR_BadCommand

ES:SI
pointer to a management ECB.

Interrupts
are disabled.

Return State

AX
has a completion code.

ES:SI
pointer to a management ECB.

Interrupts
state is disabled. (The HSM could temporarily enable the interrupts.)

Completion Codes (AX)

0000h *MSM_SUCCESSFUL*
0001h *MSM_PENDING_SUCCESS*
The routine was successful, but is holding onto the ECB.

8002h *LSLERR_BAD_PARAMETER*
The value in the first byte of the ECB's *ProtID* field is not higher than 40h.

8008h *LSLERR_BAD_COMMAND*
The HSM does not support driver management.

800Ah *LSLERR_NO_SUCH_HANDLER*
The Protocol ID is not supported.

Remarks

On entry to this routine, the HSM checks the ECB *ProtID* field to verify that the value is his identifier. The valid values in the *ProtID* field are defined by the HSM.

If the HSM must respond asynchronously to this call, it queues the ECB and returns the value MSM_PENDING_SUCCESS (01) in the AX register. When the ECB is complete, the HSM calls the ESR address in the ECB.

The MSM places the LSLERR_BAD_COMMAND return code in AX before it calls this routine. This enables the HSMs that do not support driver manager to stub this routine with a ret instruction.

DriverMulticastChange

Located in the Code segment
Must be fully written
(if the hardware supports it)

Description

Updates the multicast address table.

Entry State

DS
CGroup.

DS:SI
pointer to the multicast address table.

DS:DI
pointer to the address that was added or deleted.

BX
has the state of the address at DS:DI as follows:

zero	the address at DS:DI is invalid; flush the LAN adapters internal multicast address table and reset the LAN adapter to use the active addresses in the multicast table.
nonzero	DS:DI contains a pointer to the address that caused this function to be called. BX equals either ADD_MULTICAST_ADDRESS (2) or DEL_MULTICAST_ADDRESS (3).

CX
maximum number of entries in the multicast table.

DX:AX
used only with token-ring HSMs. This register pair has the new token-ring functional address (double word variable).

Interrupts
are disabled, but might be enabled by the driver.

Note
CLD is in effect

Call
at process time only.

Return State

Preserved
DS.

Remarks

DriverMulticastChange updates the LAN adapter's list of enabled multicast/group or functional addresses.

The MSM calls this routine whenever its list of enabled multicast/group or functional addresses has changed. *DS:SI* points to the table of multicast/group addresses which contains 8-byte entries. The first 6 bytes are the address, and the last 2

DriverMulticastChange *continued*

bytes are a use flag. If the use flag is nonzero, the entry contains a valid multicast/group address. In token-ring topologies, DX:AX contains the current enabled functional address.

On entry to this routine, *CX* contains the maximum number of entries in the multicast table. The HSM cycles through the entire table and outputs each used entry to the LAN adapter.

The HSM can assume that all addresses passed to this routine are valid multicast/group addresses. In token-ring topologies, DX:AX contains the lower two words of a valid functional address. The HSM must provide the upper word of C000h. An empty multicast address table implies that multicast/group reception is disabled in all cases.

The MSM does the multicast filtering on received packets for LAN adapters that cannot guarantee 100% multicast filtering on received packets.

Note If the hardware does not support multicast/group or functional addresses, you can stub this function with a near ret. Otherwise, this routine must be fully written. ▲

See Also

DriverPromiscuousChange
MSMUpdateMulticast (Chapter 6)
MFlags configuration table bit map (Chapter 4)

DriverPoll

Located in the Code segment
Can be stubbed with a near ret

Description

Assists polled drivers.

Entry State

DS
CGroup.

Interrupts
interrupts are disabled, but might be enabled by the driver.

Note
CLD is in effect.

Return State

Preserved
No registers or flags.

Remarks

DriverPoll is an optional routine the LSL calls periodically to assist polled drivers every timer tick (55 ms). The LSL also calls *DriverPoll* every time a protocol stack relinquishes control to the LSL. The HSM written for LAN adapters that do not have interrupt capabilities use this call; therefore, most HSMs will not use it. If an HSM does need polling, it sets the *NeedsPolling* bit inside the HSM configuration table *ModeFlags* field (bit 5).

Important *DriverPoll* should not be used for watchdog or timeout functions; instead, the HSM should schedule a reoccurring AES event that has a relatively long timeout (for example, 1 second) for this purpose. ▲

DriverPoll generally behaves in the same manner as an interrupt service routine. However, a critical section is not set up before the MSM invokes *DriverPoll*. Therefore, if this routine runs with its interrupts enabled, the HSM must explicitly enter and exit a critical section using the same routines the MSM uses for its interrupt service routine.

Note This routine must complete quickly because it is usually called from a timer interrupt. ▲

DriverPriorityQSupport

Located in the Code Segment

Description

Called by the MSM when it is about to queue a priority packet.

Entry State

AL
the priority number.

DS:SI
pointer to a Transmit ECB.

Interrupts
are disabled.

Return State

Interrupts
are disabled.

Preserved
DS, BP, SS, SP.

This routine should either transmit the packet immediately or queue the ECB. The HSM must be able to service the Priority Queue and handle priority level detection issues. Because this routine is called with interrupts disabled, it should process only essential items and return as quickly as possible.

The HSM must set *MSMPriorityQSupportPtr* with a word (near) pointer to this routine during *DriverInit*. The HSM can set or reset PrioritySupportBit as the HSM changes from supporting to not supporting priority packet states. PrioritySupportBit is checked on a per packet basis.

The AL value is the actual priority of the packet. The HSM is not concerned at this point with whether the ECB has a raw send packet or not. The value in AL will never be larger than the value in the MPrioritySup field of *DriverConfigTable*.

Note This routine is **only** used by HSM's that have transmit priority support. (See Appendix D for more details.) ▲

DriverPromiscuousChange

Located in the Code Segment
Can be stubbed with a near ret

Description

Enables and disables promiscuous mode on the LAN adapter.

Entry State

CX

Nonzero turns on promiscuous mode
Zero turns off promiscuous mode
Bit mask for promiscuous frame reception:
Bit 0 MAC frames
Bit 1 Data frames
Bit 2 SMT Frames

Interrupts

are disabled.

Note

CLD is in effect.

Call

only at process time.

Return State

Preserved

DS, BP, SS. SP.

Interrupts

are disabled.

Remarks

DriverPromiscuousChange enables or disables promiscuous mode on the HSM's adapter. The value of the CX register determines whether promiscuous mode is enabled or disabled. If the LAN topology or adapter does not distinguish between MAC and non-MAC frames (for example, Ethernet does not), any nonzero value in the *CX* register enables promiscuous mode.

The MSM calls *MSMUpdateMulticast* after *DriverPromiscuousChange* returns. This guarantees that active multicast addresses are enabled on the LAN adapter after promiscuous mode is turned off.

All adapters that have promiscuous mode enabled should be able to pass up bad packets, if possible.

Important We strongly recommend that your HSM support promiscuous mode, because various monitoring functions operate in promiscuous mode. However, if you choose not to support promiscuous mode or your hardware does not support it, this routine can be stubbed with a near ret.. ▲

DriverReset

Located in the Code segment
Must be fully written

Description

Resets the LAN adapter.

Entry State

DS
CGroup.

Interrupts
are disabled, but might be enabled by the driver.

Note
CLD is in effect.

Call
only at process time.

Return State

Preserved
No registers or flags.

Remarks

DriverReset issues a hardware reset to the LAN adapter and returns. If the HSM was *temporarily* shut down with the *DriverShutdown* routine, an application can call this routine to bring the LAN adapter out of the shutdown condition and back into full operation.

After returning from this routine, the MSM calls *MSMUpdateMulticast*, which then calls *DriverMulticastChange* (or *DriverPromiscuousChange*, if the HSM is in promiscuous mode). This ensures that the proper multicast addresses are enabled after the hardware reset.

The HSM then resets the *MSMTxFreeCount* variable to the initial value set by the HSM during *DriverInit*. After returning from this routine, the MSM assumes that all hardware transmit resources are available.

The MSM generates a *NESL_MLID_Reset* after *DriverReset* returns.

See Also

DriverMulticastChange
DriverPromiscuousChange
MSMUpdateMulticast (Chapter 6)
NESL Specification: 16-Bit DOS Programmer's Interface

DriverSend

Located in the Code segment
Must be fully written

Description

Sends a packet.

Entry State

DS
CGroup.

DS:SI
pointer to the Transmit Control Block (TCB).

CX
hardware frame length (Ethernet only; otherwise it is undefined).

Interrupts
are disabled.

Note
CLD is in effect.

Note
A critical section has been started.

Return State

DS
CGroup

Interrupts
must be disabled.

Preserved
no other flags or registers.

Remarks

The MSM calls *DriverSend* to transmit a frame onto the LAN medium. The MSM passes *DriverSend* a pointer to a TCB.

The TCB contains a *FragmentStructure* that controls the transmitted packet fragments by providing addresses and other information which pertain to the frame data.

The HSM assumes that it has the resources necessary to handle the transmit operation; it does not need to check for an available transmit hardware resource because the MSM handles the flow control for the HSM. The MSM uses the value set in the *MSMTxFreeCount* variable during *DriverInit* to determine how many outstanding transmits the HSM can manage.

DriverSend executes with the system interrupts enabled. To purge reentrancy problems, the HSM disables the LAN adapter's interrupt line when *DriverSend* is first called. This

DriverSend *continued*

keeps the driver from being interrupted by the LAN adapter. After the driver has disabled the LAN adapter's interrupt line, it enables system interrupts (STI).

Note The HSM assumes that the TCB is valid for its LAN medium. The HSM should not do consistency checking on the TCB fields. ▲

The HSM uses *MSMSendComplete* to return the TCB to the MSM when the transmission operation is complete (see Chapter 6). Some HSMs can return the TCB before they exit the *DriverSend* routine. Other HSMs must wait for a transmission complete confirmation from the driver's LAN adapter before they can return the TCB.

Note In the event of a hardware failure (for example, too many collisions), the HSM tries to retransmit the packet for the configured number of times set in the configuration table *SendRetries* field. ▲

After processing the TCB and before returning, the HSM must disable interrupts (CLI) and enable the LAN adapter's interrupt line.

Ethernet Drivers. The MSM calls the *DriverSend* routine with an additional parameter in the CX register. This parameter contains the length of the entire frame as it appears on the LAN medium. Ethernet HSMs gives this value to the LAN adapter which defines the number of bytes to transmit. The *TCBDataLength* field only describes the amount of data being passed in the TCB. In the case of Ethernet, the frame might have been padded or evenized. For example, if the HSM uses DMA to transfer the TCB data to the LAN adapter's memory, the *TCBDataLength* field would tell the LAN adapter how many bytes to transfer.

Note If you would like to implement priority packet transmission, please refer to Appendix D. ▲

See Also

DriverISR
MSMSendComplete (Chapter 6)
MSMTxFreeCount (Chapter 4)
TCB discussion (Chapter 4)
Priority packet discussion (Appendix D)

DriverSend: An Outline

This pseudocode is intended to illustrate a flow of events and does not necessarily describe optimized code.

```
    Disable LAN adapter interrupts.
    Enable system interrupts (STI).                    (run with interrupts enabled)

InternalDriverSend:

    Copy the MediaHeader from the TCB into a transmit buffer.
    Copy the fragmented data from the TCB fragment structure into a transmit buffer.

    Give the command to send the packet.

    CALL MSMSendComplete
    IF  entered through InternalDriverSend
        Return.
    ELSE
        Disable system interrupts (CLI).
        Enable LAN adapter's interrupts.
        Return.
```

DriverShutdown

Located in the Code segment
Must be fully written

Description

Shuts down LAN adapter hardware

Entry State

DS
CGroup.

CX

Zero	if permanent shutdown
Nonzero	if temporary shutdown

Interrupts
are disabled, but might be reenabled by the driver.

Note
CLD is in effect.

Call
only at process time.

Return State

Preserved
No registers or flags.

Remarks

DriverShutdown places the LAN adapter into a safe, nonactive state.

The MSM does not call *DriverSend* when the MLID is temporarily shut down.

The MSM generates a NESL_MLID_ShutDown after *DriverShutdown* returns.

Permanent Shut Down

If *CX*=0, the MLID is about to be removed from memory, and the memory returned to DOS. Therefore, the HSM must cancel or return all outstanding AES events and system resources, including any hooked interrupt vectors) before returning from this routine.

The HSM must remove and disable the LAN adapter's interrupt prior to returning.

Temporary Shut Down

If this is a temporary shut down (*CX*<>0), the HSM places the LAN adapter into a nonactive state so that it is not sending or receiving packets. However, the HSM does not need to cancel outstanding AES events or to return system resources. A call to

DriverShutdown *continued*

DriverReset by an application must be capable of bringing the HSM/LAN adapter back into full operation. If the hardware configuration has been changed in the meantime, the HSM must update the configuration table by calling *MSMMediaConfigUpdate*.

When the driver has been temporarily shut down, the only valid configuration table fields are:

- MSignature
- MConfigTableMajorVer
- MConfigTableMinorVer
- MBoardNumber
- MBoardInstance
- MFrameString
- MFrameID
- MRouteHandler
- MLookAheadSize
- MBusID
- MMLIDMajorVer
- MMLIDMinorVer
- MSharingFlags (bit 0 only)

Note The Canonical Settings bits in the ModeFlags field are valid only after the *Temporary Shutdown* function has been called. The Canonical Settings bits are not changed by a call to *MSMMediaConfigUpdate*.

Permanent and Temporary Shut Down

The HSM returns all TCBs and RCBs it has in its control before it returns.

See Also

DriverReset
MSMMediaConfigUpdate (Chapter 6)
MSharingFlags bit map (Chapter 4)
NESL Specification: 16-Bit DOS Programmer's Interface

❑

Chapter 6 Support Routines Provided by the MSM

Chapter Overview

The MSM contains routines for the HSM's use at initialization and run time. This chapter provides useful reference material.

All routines documented in this chapter have been defined as external inside the DRIVER.INC file.

This chapter discusses the following routines:

- HSMProvideTCB
- HSMShutDownMSM
- MSMBuildTransmitControlBlock
- MSMCallNESL
- MSMClearSendQueue
- MSMGenerateNESLChangeEvent
- MSMGenerateNESLEvent
- MSMGenerateNESLResumeEvent
- MSMGenerateNESLSuspendEvent
- MSMGetNextSend
- MSMGetRCB
- MSMMediaConfigUpdate
- MSMPrintStringZero
- MSMRcvComplete
- MSMRcvCompleteStatus
- MSMReturnRCB
- MSMSendComplete
- MSMSetIRQ
- MSMUnSetIRQ
- MSMUpdateMulticast

Summary of Support Routines

The following table summarizes the support routines available in the MSM.

Table 6.1 Summary of MSM Support Routines

Summary of MSM Support Routines	
Summary	**Function Name**
Interrupt Management Routines	
Unhook the interrupt vector.	MSMUnsetIRQ
Hook the specified interrupt vector.	MSMSetIRQ
Multicast Routines	
Refresh enabled addresses.	MSMUpdateMulticast
NetWare Event Service Layer (NESL) Routines	
Generate a NESL event.	MSMGenerateNESLEvent

Summary of MSM Support Routines *(continued)*

Summary	Function Name
Generate a NESL Service Resume Class event.	MSMGenerateNESLResumeEvent
Generate a NESL Service/Status Change Class event.	MSMGenerateNESLChangeEvent
Generate a NESL Service Suspend Class event.	MSMGenerateNESLSuspendEvent
Provide a generic front end to NESL.	MSMCallNESL
Output Routines	
Output a zero-terminated string.	MSMPrintStringZero
Packet Reception Routines	
Obtain an RCB for a received packet.	MSMGetRCB
Return the RCB to the MSM to complete the packet reception.	MSMRcvComplete
Return the RCB to the MSM and cancel packet reception.	MSMReturnRCB
Return the RCB to the MSM to complete packet reception (pipelined adapters).	MSMRcvCompleteStatus
Packet Transmission Routines	
Cancel queued transmit events.	MSMClearSendQueue
Convert ECB to TCB (priority transmission only).	MSMBuildTransmitControlBlock
Determine if a TCB is in the transmit queue and unqueue it.	MSMGetNextSend
Return the TCB to the MSM.	MSMSendComplete
Update Routines	
Update the MLID tables, structures, and variables.	MSMMediaConfigUpdate

Completion Codes

These MSM routines return the following completion codes:

- 0000h SUCCESSFUL
- 0002h MSMERR_TSM_FAILED
- 0003h MSMERR_MSM_FAILED
- 8002h LSLERR_BAD_PARAMETER
- 8005h LSLERR_FAIL
- 8008h LSLERR_BAD_COMMAND

HSMProvideTCB

Located in the Code Segment

Description

Called by the HSM to increase the number of TCB's that the MSM has available.

Entry State

DS
CGroup.

DS:SI
TCB buffer.

Interrupts
are disabled.

Return State

CX
is destroyed.

Interrupts
are disabled.

Preserved
DS, BP, SS, SP.

The HSM can provide extra TCBs by linking extra TCBs into the TCB's free list during *DriverInit*, prior to calling *MSMMediaConfigUpdate*, by calling *HSMProvideTCB*. The allocated memory for each TCB must be included with the TCB size in MaxFrameHeaderLen and must reside within CGroup. Because TCBs must be allocated from within the CGroup segment, *HSMProvideTCB* can only be called during initialization and from within *DriverInit*.

Note This routine is **only** used by HSM's that have transmit priority support. (See Appendix D for more details.) ▲

HSMShutDownMSM

Located in the Code Segment

Description

Called by an HSM to shut down the MSM.

Entry State

DS
CGroup.

Interrupts
are disabled.

Note
CLD is in effect.

Return State

AX
zero.

Interrupts
are disabled.

Preserved
DS, BP, SS, SP.

Remarks

This routine is used by the HSM to shut down the MSM (for example, PCMCIA card removal). It is the HSM's responsibility to place itself in a safe state before making this call. To bring the MSM back into operation, the HSM must call *MSMMediaConfigUpdate*.

See Also

MSMMediaConfigUpdate.

MSMBuildTransmitControlBlock

Located in the Code Segment

Description

Converts a Transmit ECB to a TCB.

Entry State

ES:DI
pointer to a Transmit ECB.

DS
CGroup.

Interrupts
are disabled.

Return State

SI
equal to zero if no TCB is available.

DS:SI
pointer to a TCB if SI is not equal to zero.

CX
hardware frame length (Ethernet only).

Interrupts
are disabled.

Preserved
DS, SS, SP.

Remarks

HSMs that support priority queues call *MSMBuildTransmitControlBlock* to convert an ECB to a TCB.

The HSM should be aware of the number of TCBs in the MLID. The TSM will allocate MAX_TCB_ALLOCATED number of TCBs. If the HSM makes this call when there are no TCB's available, the SI register will be set to zero.

Note This routine is **only** used by HSM's that have transmit priority support. (See Appendix D for more details.) ▲

MSMCallNESL

Located in the Code Segment

Description

Provides an interface to the NetWare Event Service Layer (NESL).

Entry State

DS
CGroup

BX
NESL function number.

ES:SI
pointer to NESL control block.

Interrupts
unspecified.

Return State

AX
completion code.

Flags
set according to AX.

Interrupts
state is preserved, but were disabled during call.

Preserved
DS, BP, SS, SP.

Completion Codes (AX)

0000h	*SUCCESSFUL*
8002h	*LSLERR_BAD_PARAMETER*
8005h	*LSLERR_FAIL*
8008h	*LSLERR_BAD_COMMAND*

Remarks

MSMCallNESL provides a generic front end to the NetWare Event Service Layer (NESL).

The MSM maintains the NESL entry point. This routine simply passes the register parameters to the NESL.

See Also

NESL Specification: 16-Bit DOS Client Programmer's Interface.

MSMClearSendQueue

Located in the Code segment

Description

Cancels queued transmit events.

Entry State

DS
CGroup.

Interrupts
are disabled and remain disabled.

Return State

Interrupts
are disabled.

Preserved
Direction flag, BP, DS, SS, SP.

Remarks

MSMClearSendQueue cancels any transmit events that the MSM queues.

After the HSM calls this routine, all send events queued inside the MSM are passed back to the LSL and placed on the LSL's internal queue. After making this call, the HSM must call *LSLServiceEvents* (see Chapter 7) in order to properly process the canceled transmit events. The HSM also sets the *MSMTxFreeCount* variable back to its default value.

MSMGenerateNESLChangeEvent

Located in the Code Segment

Description

Generates a NESL Service/Status Change Class event for each logical board.

Entry State

DS
CGroup

ES:SI
pointer to a NESL Event Parameter Block (EPB).

Interrupts
unspecified.

Return State

AX
completion code.

Flags
set according to AX.

Interrupts
state is preserved, but were disabled during call.

Preserved
DS, BP, SS, SP.

Completion Codes (AX)

0000h	*SUCCESSFUL*
8002h	*LSLERR_BAD_PARAMETER*
8005h	*LSLERR_FAIL*
8008h	*LSLERR_BAD_COMMAND*

Remarks

MSMGenerateChangeEvent generates a NESL Service/Status Change Class event for each logical board. The MSM places a pointer to the correct configuration table into the Event Parameter Block *EPB_DataPtr0* field prior to calling the NESL

See Also

NESL Specification: 16-Bit DOS Client Programmer's Interface.

MSMGenerateNESLEvent

Located in the Code Segment

Description

Generates a NESL event for each logical board.

Entry State

DS
CGroup

ES:SI
pointer to a NESL Producer Event Control Block (PECB).

Interrupts
unspecified.

Return State

AX
completion code.

Flags
set according to AX.

Interrupts
state is preserved, but were disabled during call.

Preserved
DS, BP, SS, SP.

Completion Codes (AX)

0000h	*SUCCESSFUL*
8002h	*LSLERR_BAD_PARAMETER*
8005h	*LSLERR_FAIL*
8008h	*LSLERR_BAD_COMMAND*

Remarks

MSMGenerateNESLEvent provides a generic front end to the NetWork Event Service Layer (NESL). This routine generates an event for each logical board.

The HSM calls *MSMGenerateNESLEvent* if the following two criteria are met:

1. The MSM did not register as a producer for a particular event class.
2. The HSM called *MSMCallNESL,* which allowed the HSM to register directly for the particular event class as an event producer.

The MSM places a pointer to the correct configuration table into the Event Parameter Block *EPB_DataPtr0* field, pointed to by the PECB_DataPtr pointer prior to calling the NESL.

See Also

NESL Specification: 16-Bit DOS Client Programmer's Interface.

MSMGenerateNESLResumeEvent

Located in the Code Segment

Description

Generates a NESL Service Resume Class event for each logical board.

Entry State

DS
CGroup

ES:SI
pointer to a NESL Event Parameter Block (EPB).

Interrupts
unspecified.

Return State

AX
completion code.

Flags
set according to AX.

Interrupts
state is preserved, but were disabled during call.

Preserved
DS, BP, SS, SP.

Completion Codes (AX)

0000h	*SUCCESSFUL*
8002h	*LSLERR_BAD_PARAMETER*
8005h	*LSLERR_FAIL*
8008h	*LSLERR_BAD_COMMAND*

Remarks

MSMGenerateNESLResumeEvent generates a NESL Service Resume Class event for each logical board. The MSM places a pointer to the correct configuration table into the Event Parameter Block *EPB_DataPtr0* field prior to calling the NESL.

See Also

NESL Specification: 16-Bit DOS Client Programmer's Interface.

MSMGenerateNESLSuspendEvent

Located in the Code Segment

Description

Generates a NESL Service Suspend Class event for each logical board.

Entry State

DS
CGroup

ES:SI
pointer to NESL Event Parameter Block (EPB).

Interrupts
unspecified.

Return State

AX
completion code.

Flags
set according to AX.

Interrupts
state is preserved, but were disabled during call.

Preserved
DS, BP, SS, SP.

Completion Codes (AX)

0000h	*SUCCESSFUL*
8002h	*LSLERR_BAD_PARAMETER*
8005h	*LSLERR_FAIL*
8008h	*LSLERR_BAD_COMMAND*

Remarks

MSMGenerateNESLSuspendEvent generates a NESL Service Suspend Class event for each logical board. The MSM places a pointer to the correct configuration table into the Event Parameter Block *EPB_DataPtr0* field prior to calling the NESL.

See Also

NESL Specification: 16-Bit DOS Client Programmer's Interface.

MSMGetNextSend

Located in the Code segment

Description

Returns a TCB if a transmit request is in the queue.

Entry State

DS
CGroup.

Interrupts
are disabled and remain disabled.

Note
CLD is in effect.

Return State

DS:SI
pointer to a Transmit Control Block (TCB), if the z flag is set.

CX
hardware packet length (Ethernet only).

Interrupts
state is preserved.

Flags
Z flag set if a TCB is available.

Preserved
Direction flag, DS, SS, SP.

Remarks

MSMGetNextSend returns a Transmit Control Block (TCB) if a transmit request is available.

The HSM calls this routine whenever a hardware transmit resource is available. This routine checks the MSM's internal send queue and returns a TCB if another send event is available. If a TCB is returned, the HSM initiates the transmission. After the HSM is finished with the TCB, it calls *MSMSendComplete* to return the TCB to the MSM.

If *MSMGetNextSend* returns a TCB, the MSM decrements the *MSMTxFreeCount* variable before returning a TCB in *MSMGetNextSend*.

See Also

DriverISR (Chapter 5)
DriverSend (Chapter 5)
MSMSendComplete

MSMGetRCB

Located in the Code segment

Description

Returns an RCB for a received packet if available.

Entry State

AX
the known status of the received packet:
Zero packet is good
Nonzero packet is bad, AX contains the bits as described in the bit map below.

DS
CGroup.

BP
either the packet length reported from the hardware, or –1 if the packet length is unknown (pipelining adapters).

ES:SI
pointer to the look-ahead buffer.

Interrupts
are enabled.

Note
CLD is in effect.

Return State

AX
number of bytes to skip from the start of the frame.

BP
either the number of bytes of the frame to move into the receive buffers starting from offset AX, or –1 if the packet length is unknown.

ES:SI
pointer to a Receive Control Block (RCB).

Interrupts
are enabled, but will be disabled for a period of time by the MSM during the call.

Flags
Z flag set if RCB is available.

Note
CLD is in effect.

Preserved
DS, SS, SP.

MSMGetRCB *continued*

Remarks

MSMGetRCB supports standard and pipelined adapters. Under normal conditions *MSMGetRCB* is handled as described in the section "Stand Adapters." *MSMGetRCB* can also implement pipelined adapter support as described in the "Pipelined Adapters" section.

Standard Adapters. Usually an HSM calls *MSMGetRCB* when the hardware has received an entire packet. *MSMGetRCB* obtains receive buffers (Receive Control Blocks [RCBs]) for a waiting packet.

On entry ES:SI points to a *LookAheadBuffer*. The MSM uses this buffer to determine if the HSM should process this frame. If the HSM is to process this packet, and an RCB is available, *MSMGetRCB* returns a pointer to an RCB.

If *MSMGetRCB* does not return an RCB, the HSM discards the frame. If *MSMGetRCB* returns an RCB for the frame, the HSM moves the frame into the waiting fragment buffers described by the RCBs. The HSM starts copying the frame data AX bytes from the start of the frame (in other words, AX bytes from the start of the MAC layer header). The MSM determines the number of frame bytes the HSM can copy without overflowing the receive buffer(s) and puts that number into the BP register for the HSM's use.

Note The HSM need not worry about overflowing the receive buffer(s) if it only copies the number of bytes specified in register BP. ▲

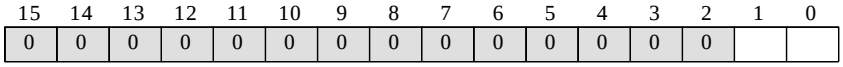
After the HSM has moved the frame data into the receive fragment buffers, it calls *MSMRcvComplete* to return the *RCB* to the MSM.

If the HSM could not complete the receive operation, it calls *MSMReturnRCB* to return the RCB.

The "on entry" AX bit map is illustrated below.

MSMGetRCB *continued*

Table 6.2
MSMGetRCB
AX Bit Map



Default Values

AX Bit Map	
Bit #	**Description**
0	CRC error, if set to 1.
1	CRC alignment error, if set to 1.
2	*Reserved*. Set to 0.
3	*Reserved*. Set to 0.
4	*Reserved*. Set to 0.
5	*Reserved*. Set to 0.
6	*Reserved*. Set to 0.
7	*Reserved*. Set to 0.
8	*Reserved*. Set to 0.
9	*Reserved*. Set to 0.
10	*Reserved*. Set to 0.
11	*Reserved*. Set to 0.
12	*Reserved*. Set to 0.
13	*Reserved*. Set to 0.
14	*Reserved*. Set to 0.
15	*Reserved*. Set to 0.

Pipelined Adapters. Pipelined adapters can be configured to interrupt prior to receiving a complete packet. In this case, the adapter must be able to be configured to wait until it has received at least the *LookAheadSize* plus *MaxHeaderSize* bytes before it interrupts. These HSMs call *MSMGetRCB* before they have received the entire packet. Therefore, the MSM cannot fill in all the *LookAhead* fields with definitive values. The error bits and length fields are set at an initial best guess until the packet has been completely received. After the RCB is completely filled in, the HSM calls *MSMRcvCompleteStatus*.

In the case of a pipelined adapter, the *MSMGetRCB* entry state is similar to the entry state document on the previous pages, except that BP contains a –1 and not the hardware-reported data length.

MSMGetRCB *continued*

The *MSMGetRCB* return state is similar to the return state document on the previous pages, except for the following:

- BP equals –1 and not the number of data bytes to copy.
- The HSM fills in the RCB buffers and must not overflow the buffers.
- The HSM must call *MSMRcvCompleteStatus* to complete the receive process.

See Also

DriverISR (Chapter 5)
MSMRcvCompleteStatus
MSMReturnRCB
MSMRcvComplete

MSMMediaConfigUpdate

Located in the Code Segment

Description

Updates the MLID's configuration.

Entry State

DS
pointer to CGroup.

Interrupts
are disabled.

Return State

AX
completion code.

DX
offset of an error message, if AX is not equal to 0. This offset is only valid during Init time.

Interrupts
are disabled.

Preserved
DS, BP, SS, SP.

Completion Codes (AX)

0000h	*Successful*
0002h	*MSMERR_TSM_FAILED*
0003h	*MSMERR_MSM_FAILED*

Remarks

The HSM calls *MSMMediaConfigUpdate* after the hardware is fully functional and the configuration variables are filled in.

The following variables must be initialized before the HSM can call this routine:

- DriverConfigTable
- MSMTxFreeCount
- MSMLookAheadSegment
- MSMPhysNodeAddress
- MSMPriorityQSupportPtr

MSMPrintStringZero

Located in the Init segment

Description

Outputs an ASCIIZ string.

Entry State

DS
CGroup.

DX
the offset to a zero-terminated string to print.

Interrupts
are unspecified.

Return State

Interrupts
state is preserved.

Preserved
Direction flag and all registers.

Remarks

MSMPrintStringZero outputs the ASCIIZ string specified in DS:DX to the standard I/O (*STDIO*).

If the first word of the string is less than 1000, a prefix is prepended to the string. The general format of the string looks like: CardName-DOS-MessageNumber.

Important Because this procedure is located in the Init segment, the HSM can only use this procedure during initialization. After initialization, the Init segment is discarded. ▲

NumberMessage Macro

The *NumberMessage* macro is used to generate message numbers, which precede each message printed by *MSMPrintStringZero*.

NumberMessage name,num,stringgg,parm1,parm2,parm3

name
the name of the Label used in declaring the string.

num
the message number that precedes the message. The minimum value for this parameter is 100. If parameter is left blank, the macro will automatically assign numbers starting at 100.

stringgg
an ASCII string.

parm1, parm2, parm3
up to three parameters can be on this line (for example, 0Ah, 0Dh, 0). The macro places the parameters at the end of the

message. If there are no parameters, the macro zero terminates the message.

Example:

```
NumberMessage FirmWareStartErrMsg, 237,
        'The FirmWare cannot be initialized.',0Ah,0Dh,0
```

Example

```
NoNE2000InMachine       db   50,0,'The board cannot be found.',0Dh,0Ah,0

    mov dx, offset CGroup:NoNe2000InMachine
    call MSMPrintStringZero
```

The following error message is output to the monitor as the result of executing the above code:

NE2000-DOS-050: The board cannot be found.

MSMRcvComplete

Located in the Code segment

Description

Passes control of a filled in RCB to the MSM.

Entry State

DS
CGroup.

ES:SI
pointer to a Receive Control Block (RCB)

Interrupts
are disabled and remain disabled.

Return State

Interrupts
are disabled.

Preserved
Direction flag, DS, BP, SS, SP.

Remarks

MSMRcvComplete returns the RCB to the MSM. The HSM calls *MSMRcvComplete* when it has completed moving frame data into an RCB receive buffer.

MSMSendComplete queues the event on the LSL's *EventHoldQueue*. If the HSM calls this routine outside of *DriverSend* or *DriverISR*, the HSM must also call the *LSLServiceEvents* routine to process the queued event. If the HSM calls this routine inside of its *DriverSend* or *DriverISR* routines, the MSM will call the *LSLServiceEvents* routines, saving the HSM this step.


Important

The HSM must not modify the RCB in any way after it calls this routine. ▲

See Also

EventHoldQueue
LSLServiceEvents
MSMGetRCB
MSMRcvCompleteStatus
MSMReturnRCB

MSMRcvCompleteStatus

Description

Passes control of a filled in RCB to the MSM.

Entry State

AX
has the status of the received packet:

Zero	packet is good
Nonzero	packet is bad, AX contains the bits as described in the bit map in Table 6.2.

DS
CGroup.

BP
has the packet length reported from the hardware.

ES:SI
pointer to the RCB.

Interrupts
are disabled, but might be re-enabled.

Return State

Interrupts
are disabled.

Preserved
Direction flag, DS, BP, SS, SP.

Remarks

If BP = –1 when the HSM called *MSMGetRCB*, the HSM calls *MSMRcvCompleteStatus* when it has finished placing the frame data into the RCB fragment buffers. *MSMRcvCompleteStatus* passes control of the RCB to the MSM. The HSM must not modify any values in the RCB after it has called this routine.

MSMReturnRCB

Located in the Code segment

Description

Passes control of the RCB to the MSM.

Entry State

DS
CGroup.

ES:SI
pointer to a Receive Control Block (RCB).

Interrupts
are disabled and remain disabled.

Return State

Interrupts
are disabled.

Preserved
Direction flag, BP, DS, SS, SP.

Remarks

If the HSM cannot complete the receive operation, *MSMReturnRCB* returns the Receive Control Block (RCB) to the MSM. This routine cancels the RCB.

See Also

MSMGetRCB
MSMRcvComplete
MSMRcvCompleteStatus

MSMSendComplete

Located in the Code segment

Description

Passes control of the TCB to the MSM.

Entry State

DS
CGroup.

DS:SI
pointer to a Transmit Control Block (TCB).

Interrupts
are disabled and remain disabled.

Return State

Interrupts
are disabled.

Preserved
Direction flag, DS, BP, SS, Sp.

Remarks

MSMSendComplete returns control of a TCB to the MSM.

The HSM calls *MSMSendComplete* when it is finished with a Transmit Control Block (TCB).

MSMSendComplete queues the event on the LSL's *EventHoldQueue*. If the HSM calls this routine outside of *DriverSend* or *DriverISR*, the HSM must also call the *LSLServiceEvents* routine to process the queued event. If the HSM calls this routine inside of its *DriverSend* or *DriverISR* routines, the MSM will call the *LSLServiceEvents* routines, saving the HSM this step.


Important

The MLID should not be in a critical section when the HSM calls *LSLServiceEvents*.

After calling *MSMSendComplete*, the HSM must not reference or modify any values in the TCB. ▲

See Also

LSLServiceEvents (Chapter 7)

MSMSetIRQ

Located in the Code segment

Description

Installs the interrupt vector and enables the IRQ specified in the driver configuration table *MIRQLine1* field.

Entry State

DS
CGroup.

Interrupts
are disabled and remain disabled.

Return State

Interrupts
are disabled.

Preserved
Direction flag, BP, BX, DS, SS, SP.

Remarks

MSMSetIRQ installs and enables the IRQ specified in the HSM configuration table *MIRQLine1* field. The HSM calls this routine during *DriverInit* to install the HSM's hardware interrupt vector. This routine also properly enables the IRQ line and sets up the proper interrupt vector on any machine type (for example, IRQ 2 or 9 will set up for the proper vector.).

During this call, the MSM initializes the *MSMEOIFlag* variable to the appropriate value, depending upon the value of *MIRQLine1* and the machine type.

The MSM front ends all hardware interrupts from the LAN adapter. After the MSM has entered a critical section, it calls the *DriverISR* routine. After the HSM has returned from *DriverISR*, the MSM checks for queued transmits, starts one, if possible, and then exits the critical section. However, the HSM must still issue the End Of Interrupt (EOI) commands to the programmable interrupt controllers.

The MSM does not support shareable interrupts. However, if you want your driver to support shareable interrupts, you can configure the HSM to do so, keeping the following points in mind:

- You must write all routines dealing with interrupts.

MSMSetIRQ *continued*

- You will not be able to use internal interrupt routines (for example, *MSMSetIRQ* and *MSMUnSetIRQ*).
- Your HSM must set the appropriate bit in the *MSharingFlags* field of the HSM configuration table.
- We recommend that your HSM follow the IBM specification for shareable interrupts (see the *IBM AT Technical Reference Manual* and/or the *IBM PS/2 Technical Reference Manual*).

Note The MSM routines do not support a second interrupt. If you wish to use more than one interrupt, you must write a complete ISR and the corresponding routines to *MSMSetIRQ* and *MSMUnSetIRQ* for the second interrupt. You are welcome to copy these routines from the MSM source code, rename them, and modify them for the additional interrupts. ▲

See Also

MSMEOIFlag (Chapter 4)
MSMIntMaskPort (Chapter 4)
MSMMaskBitOn (Chapter 4)
MSMMaskBitOff (Chapter 4)
MSMUnSetIRQ

MSMUnSetIRQ

Located in the Code segment

Description

Removes the interrupt vector and disables the IRQ that *MSMSetIRQ* installed.

Entry State

DS
CGroup.

Interrupts
are disabled.

Note
CLD is in effect.

Return State

Interrupts
are disabled.

Preserved
Direction flag, BP, BX, DS SS, SP.

Remarks

MSMUnsetIRQ disables and removes the specified interrupt and interrupt vector, set up by *MSMSetIRQ*.

Note The MSM routines do not support a second interrupt. If you wish to use more than one interrupt, you must write the complete ISR and the corresponding routines to *MSMSetIRQ* and *MSMUnSetIRQ* for the second interrupt. You are welcome to copy these routines from the MSM source code, rename them, and modify them for the additional interrupts. ▲

See Also

MSMSetIRQ
MSMEOIFlag (Chapter 4)
MSMIntMaskPort (Chapter 4)
MSMIntMaskOn (Chapter 4)
MSMIntMaskOff (Chapter 4)

MSMUpdateMulticast

Located in the Code segment

Description

References the list of enabled multicast addresses.

Entry State

DS
CGroup.

Interrupts
are disabled.

Note
CLD is in effect.

Return State

Preserved
DS, SS, SP

Remarks

MSMUpdateMulticast refreshes the MSM's list of enabled multicast addresses by calling *DriverMulticastChange*.

The HSM calls this routine after executing an internal hardware reset. Sometimes resetting the hardware disables all previously enabled multicast addresses. In these cases, after the HSM issues a hardware reset, it calls *MSMUpdateMulticast* to enable the proper multicast addresses. Some HSMs do not call this routine because a hardware reset does not disable multicast addresses.

The MSM calls this routine after *DriverReset* is called.

Important We recommend that the HSM call this routine after it resets all internal hardware. If the HSM was in promiscuous mode, *MSMUpdateMulticast* will refresh promiscuous mode by calling *DriverPromiscuousChange* instead of calling *DriverMulticastChange*.▲

See Also

DriverMulticastChange (Chapter 5)
DriverPromiscuousChange (Chapter 5)
DriverReset (Chapter 5)

❑

Chapter 7 Support Routines Provided by the LSL

Chapter Overview

The MSM handles most of the HSM's interactions with the Link Support Layer (LSL). The LSL provides the Multiple Link Interface (MLI). Writing HSMs enables you to write a simpler driver because the HSM utilizes the MSM, masking much of the LSL's complexity. However, in some cases, you might need a function directly from the LSL because some LSL support services do not exist in the MSM.

Calling the LSL Support Routines

The HSM invokes LSL services by loading the specified registers as defined and making a far indirect call through the *LSLSupport* variable. For example, to invoke the *GetIntervalMarker* function, the HSM executes the following code:

```
mov     bx, MLIDSUP_GET_INTERVAL_MARKER   ;Equate defined in
                                          ;ODI.INC
call    LSLSupport
```

When the function returns a completion code, the Z flag is set according to the value in register AX.

Use the following equates in the BX register to call the appropriate LSL support routine:

```
MLIDSUP_CANCEL_AES_EVENT        equ     4
MLIDSUP_GET_INTERVAL_MARKER     equ     5
MLIDSUP_SCHEDULE_AES_EVENT      equ     3
MLIDSUP_SERVICE_EVENTS          equ     11
```

The HSM can freely use any of the LSL services listed below.

- CancelAESEvent
- GetIntervalMarker
- ScheduleAESEvent
- ServiceEvents

Completion Codes

The LSL returns the following completion codes:

- 0000h LSLERR_SUCCESSFUL
- 8001h LSLERR_OUT_OF_RESOURCES
- 8002h LSLERR_BAD_PARAMETER
- 8003h LSLERR_NO_MORE_ITEMS
- 8004h LSLERR_ITEM_NOT_PRESENT
- 8005h LSLERR_FAIL
- 8006h LSLERR_RX_OVERFLOW
- 8007h LSLERR_CANCELLED

- 8008h LSLERR_BAD_COMMAND
- 8009h LSLERR_DUPLICATE_ENTRY
- 800Ah LSLERR_NO_SUCH_HANDLER
- 800Bh LSLERR_NO_SUCH_DRIVER
- 800Ch LSLERR_PACKET_ERRORED

This chapter contains useful reference material.

CancelAESEvent

Description

Cancels AES event.

Entry State

BX
is equal to MLIDSUP_CANCEL_AES_EVENT.

ES:SI
pointer to the AES ECB to be cancelled.

Interrupts
are disabled and remain disabled.

Return State

AX
completion code.

Interrupts
are disabled.

Preserved
BP, ES, DS, SI, SS, SP.

Note
CLD is in effect.

Completion Codes (AX)

0000h *SUCCESSFUL*
The cancel command was successfully completed.

8004h *LSLERR_ITEM_NOT_PRESENT*
The AES ECB was not found.

Remarks

CancelAESEvent attempts to cancel the AES event to which ES:SI is pointing. The AES ECB *AESStatus* field will be set to LSLERR_CANCELLED (ODI.INC). This routine will not call the LSL Event Service Routine.

The AES ECB is defined below:

```
AESECBStruc     struc
    AESLink         dd  0
    MSecondValue    dd  0
    AESStatus       dw  0
    AESESR          dd  0
AESECBStruc     ends
```

See Also

ScheduleAESEvent

GetIntervalMarker

Description

Returns a timing marker.

Entry State

BX
is equal to MLIDSUP_GET_INTERVAL_MARKER.

Interrupts
are unspecified.

Return State

DX:AX
has the current interval marker value.

Interrupts
state is preserved.

Preserved
all other registers.

Note
CLD is in effect.

Remarks

GetIntervalMarker returns a timing marker in milliseconds that could be used for timing retry events, for example. The value of this marker has no relation to any real-world, absolute time. When time marker values are compared with each other, the difference is elapsed time in milliseconds.

If the HSM is using the interval marker for time measurement, the interrupts must be enabled. The interval marker is updated using the PC's timer interrupt approximately every 55 milliseconds.

ScheduleAESEvent

Description

Schedules driver-defined events.

Entry State

BX
is equal to MLIDSUP_SCHEDULE_AES_EVENT.

ES:SI
pointer to an AES Event Control Block (AES ECB).

Interrupts
are unspecified.

Return State

Interrupt
state is preserved.

Preserved
BP, DS, ES, SI, SS, SP.

Note
CLD is in effect.

Remarks

ScheduleAESEvent schedules a driver-defined event to occur at the end of a specified time interval.

Before scheduling an AES event, the HSM sets the AES ECB *MSecondValue* field to the desired timeout value in milliseconds. The HSM also sets the AES ECB *AESESR* field to an internal routine to handle the completed event. (The AES ECB is defined below.)

Event service routines are invoked with the following parameters:

Entry State

ES:SI pointer to an AES ECB.

Interrupts are disabled.

Return State

Interrupts are disabled.

Preserved *BP, DS, SS, SP.*

Definition of the AES ECB Structure

```
AESECBStruc     struc
    AESLink         dd  0
    MSecondValue    dd  0
    AESStatus       dw  0
    AESESR          dd  0
AESECBStruc     ends
```

ScheduleAESEvent *continued*

Example: DriverTimeOut

The HSM could use a *DriverTimeOut* routine to detect any send-failure timeouts. This procedure is usually set up as an AES event that periodically reschedules itself at defined intervals to check for transmit failures.

DriverTimeOut regularly inspects the LAN adapter to determine if the board has failed to complete a send. If the timeout check routine encounters a send timeout, it discards the packet being sent, resets the board, and sends the next packet (if one is waiting on the send queue).

DriverTimeOut: An Outline

This pseudocode is intended to illustrate a flow of events and does not necessarily describe complete or optimized code.

```
Push DS, BP.
Set DS to CGroup.

IF transmit is in progress,
    IF transmit has timed out,
        increment appropriate error counter,
        return timed out TCB,
        reset the LAN adapter/controller.
        move MLIDSUP_SERVICE_EVENTS into BX,; (process queued
                                                ; receives)
        CALL LSLSupport,
        CALL MSMGetNextSend.                    ;(check the send
                                                ; queue)
        IF TCB was available,
            call DriverSend procedure.
        ENDIF
    ENDIF
ENDIF

reset the MSecondValue field in the AES ECB,    ; (reschedule AES
                                                ; event)
move MLIDSUP_SCHEDULE_AES_EVENT into BX,
CALL LSLSupport.

Pop BP, DS.
Return.
```

See Also CancelAESEvent

ServiceEvents

Description

Completes the processing of queued send and receive events.

Entry State

BX
is equal to MLIDSUP_SERVICE_EVENTS.

Interrupts
are disabled, but could be enabled inside the routine.

Return State

Interrupts
are disabled.

Preserved
BP, DS.

Remarks

The HSM invokes *ServiceEvents* to complete the processing of send and receive events only if the HSM called *MSMSendComplete*, *MSMRcvComplete*, or *MSMRcvCompleteStatus* outside of *DriverSend* or *DriverISR*.

During this routine, protocol stacks can enable interrupts and call the *DriverSend* routine.

Caution If the HSM does not call *ServiceEvents* when queued events are outstanding, the system may halt. ▲

See Also

LSLSendComplete
LSLRcvComplete
LSLRcvCompleteStatus

❑

Appendix A Creating the DOS ODI LAN Driver

Appendix Overview

This appendix explains the details involved in creating the DOS ODI LAN driver with the MSM and TSM modules. This includes an explanation of the files and tools needed to accomplish this task.

Read this appendix if you have never used the MSM and TSM modules to create a LAN driver.

Required Software Tools

Assembler Package

Supported assemblers and their corresponding linkers are:

- Microsoft Assembler (MASM) v5.1 or above
- Borland's Turbo Assembler (TASM)

MASM has an additional utility which converts an .EXE file into a .COM file.

Operating Environment

You must be running DOS version 3.x or above at the workstation. You must also be running NetWare version 2.x or above at the file server.

Debugging Software (Optional)

Make sure that your debugging software is compatible with the assembler you choose.

Required Files

So that you can create a working DOS ODI LAN driver, the developer's kit includes the following files:

Source files

```
<TSM>.ASM       ;Provided by Novell
MSM.ASM         ;Provided by Novell
```

Include files

```
DRIVER.INC      ;Provided by Novell
MSM.INC         ;Provided by Novell
ODI.INC         ;Provided by Novell
MSMTXT.INC      ;Provided by Novell
NESLNAME.INC    ;Provided by Novell
NESL.INC        ;Provided by Novell
```

The <TSM>.ASM file you use with your MLID depends upon which topology the MLID is using (for example, Ethernet, token-ring, FDDI).

You must also have the following .COM files in order to bring up a functional workstation.

```
LSL.COM                 ;v2.11 or above
IPXODI.COM              ;v2.20 or above
NETX.COM or VLM.EXE     ;v1.11 or above
```

Assembling and Linking the Driver

A sample batch file that illustrates the process of assembling and linking the MLID with the appropriate files is shown below:

Using MASM

```
masm        msm;
masm        ethernet;
masm        ne2.asm,ne2.obj,ne2.lst;
link        /m msm+ethernet+ne2,ne2;
exe2bin ne2.exe ne2.com
```

Using TASM

```
tasm        msm;
tasm        ethernet;
tasm        ne2;
tlink       /t msm+ethernet+ne2, ne2,;
```

The above examples assume that all source modules and include files are located in the current directory. You must link the MSM, <TSM>, and <HSM> modules in the order shown above.

Other Files

The NET.CFG file is used to specify hardware and system configuration information. If you use it, you must generate this ASCII file.

The following steps are involved in creating a DOS ODI LAN driver:

1. Edit the <HSM> portion of the MLID.
2. Edit the necessary parts of the MSM and/or <TSM> module if necessary.
3. Assemble the source code and appropriate include files.
4. Link the modules in the following order: MSM, <TSM>, and HSM.
5. If you are using the MASM assembler, convert the .EXE file into a .COM file.
6. Using the other .COM files supplied with NetWare, load the workstation as follows:

   ```
   LSL.COM
   <driver>.COM
   IPXODI.COM
   VLM.COM
   ```

❑

Appendix B The NET.CFG Configuration File

Appendix Overview

ODI modules (including the LAN driver) use the NET.CFG file to obtain the network system configuration information at initialization time. The MSM parses the NET.CFG, evaluates the driver parameters, and then sets those parameters in the HSM driver configuration table.

Each node (workstation) in a network system contains at least one network LAN adapter, the LAN adapter's MLID, the Link Support Layer, and the protocol stacks.

As each module of the network system loads into the computer system, that module reads the NET.CFG file to find configuration information concerning its operation.

The NET.CFG Configuration File

The NET.CFG configuration file consists of main section headings and keywords for ODI modules.

NET.CFG File Main Section Headings

Main section headings must be flush left and typically begin with one of the following labels: Link Driver, Protocol, or Link Support. The main section heading enables an ODI module to locate its configuration section. During initialization, the ODI module parses the NET.CFG for its main section header. The module then parses and interprets the configuration entries until the parser reaches the end of the file or until another main section header is encountered.

All configuration entries following the main section heading must be preceded with one or more white spaces. The end of the configuration information for a specific module is signaled by the occurrence of another main section heading or the end of the file. The text in the NET.CFG is not case sensitive, and the text parameters are delimited by white space. Any comments should be preceded by a semicolon (;).

MLID Main Section Headings and Keywords

MLID Main Section Headings

MLIDs use the main section heading defined by the *DriverMainSectionText* variable. Typically, this is "Link Driver" followed by the name of the MLID to which the information in the section following the heading refers. The following example illustrates a sample NET.CFG file entry for an MLID:

Example

```
link driver NTR2000                     ;Main section header for
                                        ;Novell token-ring MLID
    Node address 400000001234M          ;Node address of the MLID
```

An MLID's main section header should use an intuitive name such as the MLID's executable filename (for example, NTR2000).

MLID Subsidiary Keywords and Parameters

Bus ID <name> <ID>

Bus ID allows the user to specify the bus to look for the adapter on, if the MLID supports multiple bus types.

The currently defined bus IDs are defined below.

name	ID
ISA	0
MCA	1
EISA	2
PCMCIA	3
PCI	4
VL	5
Default	–1 (0FFh)

Note This list is not inclusive. You can obtain an updated list of identifiers from Novell Labs. ▲

The default value indicates that the MLID should search each of the machine's busses for a supported card. It should then initialize the first supported card it finds and update the *MBusID* field in the MLID's configuration table. The MLID determines the order in which the busses will be searched.

DMA [#n] channel

If the MLID uses multiple DMA channels, *DMA* configures DMA *#n* to be *channel*. *#n* can be #1 or #2 (it is assumed to be #1 if this parameter is absent). You might need multiple *DMA* entries.

Example

```
; Example NET.CFG file
; Configure DMA 1 to be channel 7
Link driver NE2100
    DMA #1 7
```

Frame <name> [Addressing Mode.]

At initialization time, the MLID uses this keyword to create a logical board for "<name>" frame type. You can load multiple frame types concurrently

You can configure the addressing mode on some MLIDs on the basis of frame type. The following keywords determine the address mode:

- *LSB* Canonical addressing mode
- *MSB* Noncanonical addressing mode

In the following example, token-ring is in MSB mode, and token-ring SNAP is in LSB mode. The configuration table ModeFlags field indicates the mode the logical board is operating in. See the *ODI Specification Supplement: Canonical and Noncanonical Addressing* for more details regarding canonical and noncanonical addressing.

Example

```
; Example NET.CFG file
; Configure the MLID to support 4 frame types
Link driver NE2000
    Frame ETHERNET_802.2
    Frame ETHERNET_802.3
    Frame ETHERNET_II
    Frame ETHERNET_SNAP
Link driver NTR2000
    Node address 020012345678L
    Frame Token-Ring MSB            ;noncanonical addressing
    Frame Token-Ring_SNAP LSB       ;canonical addressing
```

IRQ [#n] line

IRQ configures the interrupt #n to be the interrupt *line*. #n can be #1 or #2 (it is assumed to be #1 if this parameter is absent. If the MLID uses multiple interrupt lines, you can have multiple IRQ entries.

Example

```
; Example NET.CFG file
; Configure first interrupt to be interrupt 9
link driver NE2100
    Int #1 9
```

MEM [#n] address [length]

MEM configures the #nth memory address and range at *address* for *length* paragraphs. *Address* must be an absolute physical address. #n can be #1 or #2 (it is assumed to be #1 if this parameter is absent). You can have multiple *MEM* entries.

Example

```
; Example NET.CFG file
; Configure memory address and range
link driver NE2000
    MEM #1 C0000 80
```

Node address h [format]

Node address overrides any hard-coded node address in the MLID's hardware, if the hardware allows it. The new address h is used to program the LAN adapter.

You can use either canonical or noncanonical format.

Example

```
Node address   0800005A656BL    ;Node address in canonical
                                ;form
Node address   1000005AA6D6M    ;Node address in
                                ;noncanonical form
```

If *M* or *L* is not specified, the default for the node address is the Physical Layer form of the address.

Note Even though FDDI is a noncanonical topology at the physical layer, all addresses are passed in canonical (LSB) format. ▲

Note A node address labeled with an "M", which indicates a noncanonical (MSB) address, is legal even when the media is canonical; the MLID simply bit-swaps the provided address to obtain the appropriate canonical (LSB) address. ▲

Example

```
; Example NET.CFG file
; Set up the NTR2000 card for noncanonical form address.
Link driver NTR2000
    Node address 1000005AA6D6
; Set up the NTR2000 card for address in canonical form.
Link driver NTR2000
    Node address 0800005A656BL
; Set up the NTR2000 card for address in noncanonical form.
Link driver NTR2000
    Node address 1000005AA6D6M
```

Port[#n] address [length]

Port configures the #nth I/O port address and length at *address* for *length* ports. #n can be #1 or #2 (it is assumed to be #1 if this parameter is absent). You could need multiple *Port* entries.

Example

```
; Example NET.CFG file
; Configure the I/O port
link driver NE1000
    port 320
```

Protocol <name> h <frame>

Protocol tells the MLID that the named protocol has a protocol IO of h, for frame type *frame*. This enables new protocols (or overrides the default protocol ID) to be handled by existing MLIDs. The MLID uses this information to call the LSL *AddProtocolID* routine during initialization. The LSL maintains a table containing protocol IDs, protocol names, and frame types for the protocols to use.

Example

```
; Example NET.CFG file
; Identify IPX as E0 on Ethernet_802.2, identify IP as 800h,
; and ARP as 806h on Ethernet_II
Link driver NE1000
    Frame Ethernet_II
    Frame Ethernet_802.2
    Protocol IPX E0 Ethernet_802.2
    Protocol IP 800 Ethernet_II
    Protocol ARP 806 Ethernet_II
```

Note While the protocol keyword provides the LSL with information that protocol stacks can use, it does not imply an implicit binding. ▲

Slot n and Slot ?

Slot n indicates which slot contains the card for the MLID. *n* is 1 based; that is, the first slot is one (1).

Slot ? indicates that the MLID is to scan the slots for the adapter. This is the default mode if this keyword is not present.

Example

```
; Example NET.CFG file
Link driver NE2
     slot 3
```

❑

Appendix C Supporting PCMCIA Boards

Appendix Overview

PCMCIA adapters present new challenges to HSM developers. This appendix describes some suggestions for implementing PCMCIA support into the HSM.

This information is useful if you are developing HSMs that support PCMCIA LAN adapters. However, as this appendix only describes changes required by ODI and does not cover everything that Card Services required, you will also need to refer to the *PCMCIA Card Services Specification* Release 2.1.

Supporting PCMCIA Card Services

Existing HSMs can support PCMCIA LAN adapters with the following changes.

During *DriverInit*

The *DriverInit* routine should perform the following.

1. Check the value of the configuration table *MBusID* field for either *BUS_ID_PCMCIA* or *BUS_ID_DEFAULT*.
2. Check the *MSMSystemFlags* for a set *CSFlag*.
3. Return with DX pointing to an error message and AX equal to MSMERR_HSM_FAILED, if Card Services are not loaded.
4. Register any classes that the HSM generates events for and that are not provided by the MSM with the NetWare Event Service Layer (NESL). (For more information about the NESL, see *NESL Specification: 16-Bit DOS Client Programmer's Reference*.) The MSM registers the following event classes with NESL:

   - NESL_Service_Suspend
   - NESL_Service_Resume
   - NESL_ServiceStatus_Change

   (These class names are defined in NESLNAME.INC.)
5. Register with Card Services. The HSM provides the call back handler required by Card Services.
6. If a card is present and successfully initialized, *DriverInit* returns with AX equal to MSM_SUCCESSFUL. If a card is not present, the registration routine returns with AX equal to MSMERR_CARD_NOT_PRESENT and DX pointing to a message indicating that the driver loaded, but the Network Interface Controller (NIC) was not present. If an error occurred during the initialization of the card, the registration routine returns with DX pointing to the offset of an error message and AX equal MSMERR_HSM_FAILED.

The Callback Handler and Callback Handler Subroutines

Card Services requires the HSM to provide a callback handler to support a PCMCIA card. (See the *PCMCIA Card Services Specification* Release 2.1 for details.)

The following section briefly outlines the logic for the card insertion and removal subroutines.

Card Insertion Subroutine

This subroutine performs the following:

1. If the PCMCIA card that the driver is loaded for is inserted:
   - Negotiates hardware resources with Card Services.
   - Activates the card.
   - Calls *CardInit* to initialize hardware and driver software.

     CardInit is HSM-implementation specific. It is identified here to point out that the code dealing with the hardware initialization must be relocated to the Code segment (runtime segment). This code was once included in *DriverInit* in the Init segment of code.

     Note Code dealing with initialization will be called every time the desired PCMCIA Card is inserted.

     Once everything at the HSM level has been initialized successfully *MSMMediaConfigUpdate* must be called.

   - If *CardInit* is successful, generate the NESL_Card_Insertion event.
   - Return to Card Services.
2. If this card insertion event is not from the desired card, return to Card Services

Card Removal Subroutine

This subroutine performs the following:

If the PCMCIA Card that the driver initialized for is removed, the HSM should take the following steps before returning to Card Services:

- Call *HSMShutDownMSM*.
- Release the interrupt vector by calling *MSMUnSetIRQ*.
- Return the Hardware resources through the Card Services.
- Generate a NESL_Card_Removal event.

❑

Appendix D Transmitting Priority Packets

Appendix Overview

Priority packet transmission is a new capability in the DOS ODI specification. The new toolkit provides an interface that allows the HSM to support priority packet transmission. This appendix describes this procedure. (The functions used in priority packet transmission are documented in Chapter 5 and 6.)

Priority Support Algorithm

The following is the algorithm used for priority support.

1) During *DriverInit*, the HSM sets the following:

   - The MSMPriorityQSupportPtr variable with a word pointer (near) to the *DriverPriorityQSupport* routine(Chapter 5).
   - PrioritySupportBit in the MFlags field of *DriverConfigTable*.
   - The MPrioritySup field in *DriverConfigTable* to indicate the number of levels available.

   The HSM can set or reset the PrioritySupportBit as the HSM changes from Priority Queue Support Enabled to Disabled state. The PrioritySupportBit is checked on a per queued packet basis.

2) The protocol sets the ProtoNum (StackID) field to a value greater than or equal to 0FFF0h. The following are the valid ProtoNum values:

0FFFFh	Raw Send packets (no priority).
0FFFEh – 0FFF8h	Raw Send packets (priority levels 1–7; 7 is the highest priority).
0FFF7h	Non-Raw Send packets (no priority).
0FFF6h – 0FFF0h	Non-Raw Send packets (priority levels 1–7; 7 is the highest priority).

   Priority levels are defined as 0 for no priority, 1 for low priority, and 7 for high priority. To extract the priority level, NEG (2's complement) the ProtoNum (StackID) field and AND it with 07h. The result is a number from 0 to 7.

3) The MSM normally gives the packet to the HSM directly through *DriverSend* as a TCB. If MSMTxFreeCount is zero and the transmit ECB is a priority transmit ECB, the MSM calls *DriverPriorityQSupport*, which gives the HSM to the transmit ECB. This HSM provided routine will either queue it in the HSM for transmission soon, or transmit the packet out a priority channel by first calling *MSMBuildTransmitControlBlock* (Chapter 6) to build a TCB.

4) The HSM calls *MSMBuildTransmitControlBlock* to build a TCB whenever a priority transmit resource becomes

available and there is a Transmit ECB queued in the HSM's priority queue. The MSM checks if a TCB is available during *MSMBuildTransmitControlBlock*. The HSM may only use MAX_TCB_ALLOCATED minus the maximum value of MSMTxFreeCount number of TCBs. Non-priority packets have the original MSMTxFreeCount number of TCB's reserved exclusively for their use.

5) The HSM can provide extra TCBs by increasing the MAX_TCB_ALLOCATED equate and reassembling the TSM module. The HSM can also provide extra TCBs by linking extra TCBs into the TCB's free list during *DriverInit*, prior to calling *MSMMediaConfigUpdate*, by calling *HSMProvideTCB* (Chapter6). The allocated memory for each TCB must be included with the TCB size in MaxFrameHeaderLen and must reside within CGroup. Because TCBs must be allocated from within the CGroup segment, *HSMProvideTCB* can only be called during initialization and from within *DriverInit*.

6) After the HSM has transmitted the TCB returned from *MSMBuildTransmitControlBlock*, the HSM calls *MSMSendComplete*. This takes care of incrementing the counters, calling the TxMonitor, placing the TCB back on the free list, and returning the ECB to its original owner.

❑

Glossary

Abort

To execute an orderly termination of a process whenever the process cannot or should not complete.

Adapter

A circuit board driven by software. In the context of this document an adapter refers to a physical board. See also *NIC, MLID, Driver*.

Address

A unique group of characters that correspond either to a selected memory location, an input/output port, or a device on the network. See also *Node address*.

AES—Asynchronous Event Scheduler

An auxiliary service that measures elapsed time and triggers events at the conclusion of measured time intervals.

API—Application Programming Interface

A defined set of routines that enables two software modules to pass information between them.

ARP—Address Resolution Protocol

The protocol used by TCP/IP to locate nodes on a network.

Asynchronous process

A process that does not depend upon occurrence of a timing signal.

Bit

A binary digit that can only be 0 or 1.

Broadcast

A simultaneous transmission of data from a single source to all destinations.

Buffer

A data area used for the temporary storage of data being moved between processes.

Bus

The hardware interface upon which data is transferred.

Byte

A sequence of 8 bits.

CAM—Content Addressable Memory

Memory that resides on the adapter. In the context of this specification, this memory is used to hold the group addresses that the adapter is to filter.

Completion code

A code returned by a routine to indicate that the routine has completed either successfully or unsuccessfully.

Control Block

A data structure that is used by a process to store control information. See also *ECB*.

Destination Address

A field that identifies the physical location to which data is to be sent.

Driver

The software module that operates a circuit board. In the context of this document, driver refers to a software module that drives a network board (or adapter) and enables a device to communicate over a LAN. See also *Adapter, NIC, MLID*.

ECB—Event Control Block

A data structure that contains the information required to coordinate the scheduling and activation of certain operations. All ODI layers and AES functions act upon *ECBs*.

EISA—Extended Industry Standard Architecture

A 32-bit bus standard, a superset of the ISA standard.

EOI—End of Interrupt

A command issued to the interrupt controller (PIC) indicating an end of interrupt.

ESR—Event Service Routine

An application-defined procedure that is called after an event occurs. An event can be the completion of a send request, the completion of a listen request, or the recurrence of an event that rescheduled itself with the AES.

Ethernet

A wire medium usually used in a bus topology.

FDDI—Fiber Distributed Data Interface

A dual ring topology.

Frame

The unit of transmission on the network. The frame includes the associated addresses and control information in the Media Access Control (MAC) Layer and the transmitted data.

HSM—Hardware Specific Module

One of three modules comprising the LAN driver toolkit. The developer writes the HSM to handle all hardware interactions for a specific physical board.

Interrupt

A hardware signal that causes the orderly suspension of the currently executing process in order to execute a special program (or interrupt handler).

IOCTL—I/O Control

MLID procedures that perform specific actions (for example, add multicast address, reset, shut down, etc.).

IP—Internet Protocol

The protocol used by TCP/IP. IP is connectionless and was designed to handle a large number of WANs and LANs on an internetwork.

IPX—Internet Packet eXchange

An implementation of the Internetwork Datagram Packet (IDP) protocol from Xerox. It allows applications running on NetWare workstations to take advantage of NetWare communications drivers to communicate directly with other workstations, servers, or devices on the internetwork.

ISA—Industry Standard Architecture

An 8/16-bit bus standard used with Intel's microprocessors.

ISR—Interrupt Service Routine

Routine that is executed to handle a hardware or software interrupt request.

LAN—Local Area Network

At least two computers (usually located in the same building) connected together in such a way as to allow them to communicate and share resources.

LSL—Link Support Layer

An ODI layer through which multiple protocol packets are directed from the MLID to a designated protocol stack, and vice versa. The LSL directs incoming and outgoing packets.

MAC Header—Media Access Control Header

Controls the transmission of packets through a network. The MAC header includes source and destination data.

Medium

The physical carrier of a signal.

Micro Channel Architecture

A bus standard defined by IBM.

MLI—Multiple Link Interface

The interface between the MLID and the LSL that allows multiple MLIDs to exist concurrently.

MLID—Multiple Link Interface Driver

The ODI layer that receives and transmits packets to a hardware device. This acronym refers to ODI LAN drivers.

MMIO—Memory Mapped I/O

An architecture for input and output that allows I/O ports to be accessed as though they were memory locations.

MPI™—Multiple Protocol Interface

The interface between the LSL and a Network Layer protocol stack that allows different communication protocols to operate concurrently.

MSM—Media Support Module

One of three modules comprising the LAN driver toolkit. The MSM standardizes and manages the generic details of interfacing ODI MLIDs to the LSL and the operating system.

Multicast

The simultaneous transmission of data from a single source to a selected group of destination addresses on the network.

NIC—Network Interface Controller/Card

The physical network board installed in workstations and file servers.

NLM—NetWare Loadable Module

Applications that are loaded dynamically and integrated with all the NetWare server operating systems starting with NetWare 3.

Node

Any network device that transmits and/or receives data. The device must have a physical board and a unique address. See also *Node Address.*

Node Address

A unique combination of characters that corresponds to a physical board on the network. Each adapter must have a unique node address.

ODI—Open Data–Link Interface

The model that allows multiple network protocols, physical boards, and frame types to coexist on a single workstation or server.

OSI—Open Systems Interconnection

A standard communications model that defines communications between computer systems.

Packet

The unit of transmission on the network. The packet includes the associated addresses and control information.

PID—Protocol Identification

A stamp containing a globally administered value (1 to 6 bytes in length) that reflects the protocol stack in use (for example, E0h=IPX 802.2). The PID located in every packet is a stamp that uniquely identifies the packet as belonging to a specific protocol.

Protocol

The set of rules and conventions that determines how data is to be transmitted and received on the network.

Pseudocode

Describes computer program algorithms generically without using the specific syntax of any programming language.

RAM—Random Access Memory

The computer's (or physical board's) storage area into which data can be entered and retrieved nonsequentially.

RCB—Receive Control Block

A data structure used by the MLID to receive data.

ROM—Read Only Memory

The portion of the computer's (or physical board's) storage area that can be read only (write operations are ignored).

Shared RAM

The RAM on some physical boards that can be accessed by either the computer or the physical board on which the RAM is installed.

Source Address

A field that identifies the physical location that is sending the data.

SPX—Sequenced Packet Exchange

A Session Layer protocol that uses IPX. SPX provides connection oriented services and guarantees packet delivery.

Stubbed Routine

A routine that contains only a return (ret) instruction.

Synchronous Process

A process that depends upon the occurrence of another event such as a timing signal.

TCB—Transmit Control Block

The data structure used by the MLID to transmit data.

TCP—Transmission Control Protocol

Allows a process on one machine to send a stream of data to a process on another machine.

Token–Ring

A network that utilizes a ring topology and passes a token from one device to another. A node that is ready to send data can capture the token and send the data for as long as it holds the token.

TSM—Topology Specific Module

One of three modules comprising the LAN driver toolkit. The <TSM>.OBJ manages the operations unique to a specific media type.

TSR—Terminate-and-Stay-Resident

A DOS program or routine that remains in memory after being loaded and subsequently exited.

VAP—Value Added Process

A process that runs "on top" of the NetWare 2 network operating system (in much the same way a word processing or spreadsheet application runs on top of DOS). VAPs tie in with the network operating system so that additional enhancements can provide services without interfering with the network's normal operation.

Virtual Machine

An illusion of multiple processes, each executing on its own processor with its own memory. The resources of the physical computer can be used to share the CPU and make it appear that each process has its own processor. The virtual machine is created with an interface that appears to be identical to the underlying hardware.

WAN—Wide Area Network

At least two computers remotely connected together in such a way as to allow them to communicate over wide distances and to share resources.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W