



**Novell ODI Specification:  
NetWare 16-Bit DOS  
Protocol Stacks and MLIDs**

ODI Specification Version 4.00

Document Version 1.03

Part Number: 107-00078-001

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.

© Copyright 1995 and 1996 by Novell, Inc. All rights reserved. This document may be freely copied and distributed as long as it is reproduced in its entirety and for the benefit of network product developers. Portions of this document may be included with other material as long as authorship is attributed to Novell, Inc. and appropriate copyright notices are included.

Novell, Incorporated  
122 East 1700 South  
Provo, Utah 84606

## Trademarks

Novell, Inc. has attempted to supply trademark information about company names, products, and services mentioned in this manual.

Novell and NetWare are registered trademarks of Novell, Inc.

Internetwork Packet Exchange, IPX, Link Support Layer, LSL, MAC, Multiple Link Interface, MLI, Multiple Link Interface Driver, MLID, Multiple Protocol Interface, MPI, NE1000, NE2000, NE2100, NE/2, NE2-32, NTR2000, NetWare System Interface, NSI, NetWare Access Services, NetWare Core Protocol, NCP, NetWare Directory Services, NDS, NetWare DOS Requester, NDR, NetWare Express, NetWare Loadable Module, NLM, NetWare Management Agent, NetWare Requester, NetWare Runtime, Novell Embedded Systems Technology, NEST, Novell Labs, Open Data-Link Interface, ODI, Packet Burst, RX-Net, SFT, Transactional Tracking System, TTS, Virtual Loadable Module, VLM are trademarks of Novell, Inc.

IBM is a registered trademark of International Business Machines Corporation. AppleTalk is a registered trademark of Apple Computer, Inc. LAT is a trademark of Digital Equipment Corporation.



# Table of Contents

<b>Table of Contents</b> .....	<b>iii</b>
<b>Preface</b> .....	<b>x</b>
Document Organization .....	x
Referenced Documents .....	x
Equates and Structures .....	xi
LSL Initialization Entry Function Equates .....	xi
Registration Structures .....	xii
MemStatStruc Structure .....	xii
StackChainStruc Structure .....	xii
LookAheadStruc Structure .....	xiii
LDestType Bit Definitions .....	xiii
LPacketAttrib Bit Definitions .....	xiii
Promiscuous State Flag Bit Definitions .....	xiii
<b>ODI Architecture</b> .....	<b>I-1</b>
Section Overview .....	I-2
<b>Introduction to ODI</b> .....	<b>1-1</b>
Chapter Overview .....	1-2
Open Data-Link Interface (ODI) .....	1-3
Protocol Stacks .....	1-3
Link Support Layer (LSL) .....	1-5
Multiple Link Interface Drivers (MLIDs) .....	1-5
Data Flow .....	1-6
Send Data Flow .....	1-6
Receive Data Flow .....	1-7
<b>Protocol Stacks</b> .....	<b>II-1</b>
Section Overview .....	II-2
<b>Overview of Protocol Stacks</b> .....	<b>2-1</b>
DOS Environment .....	2-2
Protocol Stack Performance .....	2-2
Network Interface Card Utilization .....	2-2
Hardware/Media Independence .....	2-3
Protocol Stack Multiplexing .....	2-4
Packet Transmission and Reception .....	2-4
Binding Protocol Stacks .....	2-4
Receiving Packets .....	2-6

---

Priority Packet Support .....	2-6
Stack Filtering .....	2-7
<b>Protocol Stack Data Structures .....</b>	<b>3-1</b>
Overview .....	3-2
Protocol Stack Configuration Table .....	3-3
Protocol Stack Configuration Table Structure Sample Code .....	3-3
Protocol Stack Statistics Table .....	3-4
Protocol Stack Statistics Table Structure Sample Code .....	3-4
<b>Protocol Stack Initialization .....</b>	<b>4-1</b>
Overview .....	4-2
Locating the LSL .....	4-3
Registering with the LSL .....	4-5
Determining Which Logical Boards to Service .....	4-6
Explicit Method .....	4-6
Dynamic Method .....	4-6
Adding Protocol IDs .....	4-7
Multiple Board Support .....	4-8
Obtaining Protocol ID Value(s) .....	4-8
Customizing a Protocol Stack .....	4-8
Line Speed .....	4-9
Measuring Effective Network Performance .....	4-9
Maximum Packet Size .....	4-9
Multicast Support .....	4-10
Receive Look Ahead .....	4-11
Bind to Logical Board(s) .....	4-11
Final Initialization .....	4-12
Unhooking the Protocol Stack .....	4-12
Unload Module Algorithm .....	4-12
<b>Protocol Stack Packet Reception .....</b>	<b>5-1</b>
Protocol Stack Packet Receive Operation .....	5-2
Receive Routine Events .....	5-2
Receive Look Ahead .....	5-2
Receive Handler .....	5-2
LookAheadStruc Structure .....	5-3
Receive Look Ahead Size .....	5-5
Protocol Stack Packet Reception Methods .....	5-6
Bound Protocol Stack .....	5-6
Prescan Protocol Stack .....	5-6
Default Protocol Stack .....	5-6
StackChainStruc Structure .....	5-7
Protocol Receive Handler .....	5-9
Protocol Receive Complete Handler .....	5-11
ECB Resubmit Procedures .....	5-14
<b>Protocol Stack Packet Transmission .....</b>	<b>6-1</b>
Overview .....	6-2

Send Routine Events .....	6-2
Starting the Packet Send .....	6-3
Supporting Multiple Outstanding Transmit Requests .....	6-3
Sending the Packet .....	6-3
Event Control Blocks .....	6-3
Transmit Complete .....	6-7
Protocol Transmit Complete Handler .....	6-8
Prescan Transmit Protocol Stack Handler .....	6-9
<b>Protocol Stack Control Routines .....</b>	<b>7-1</b>
Overview .....	7-2
BindToMLID .....	7-3
GetProtocolStackConfiguration .....	7-4
GetProtocolStackStatistics .....	7-5
MLIDDeregistered .....	7-6
ProtocolManagement .....	7-7
ProtocolPromiscuousChange .....	7-9
UnbindFromMLID .....	7-10
<b>Link Support Layer (LSL) .....</b>	<b>III-1</b>
Section Overview .....	III-2
<b>Overview of the LSL .....</b>	<b>8-1</b>
Overview .....	8-2
Link Support Layer (LSL) .....	8-3
LSL Completion Codes .....	8-3
Specification Version String .....	8-4
LSL Commandline Switches .....	8-4
Custom Configuration Files .....	8-5
<b>LSL Data Structures .....</b>	<b>9-1</b>
Overview .....	9-2
LSL Configuration Table .....	9-3
LSL Configuration Table Structure .....	9-3
LSL Statistics Table .....	9-4
LSL Statistics Table Structure .....	9-5
<b>LSL Protocol Stack Support Routines .....</b>	<b>10-1</b>
Overview .....	10-2
AddProtocolID .....	10-5
BindStack .....	10-7
CancelAESEvent .....	10-8
DefragmentECB .....	10-9
DeregisterDefaultStackChain .....	10-10
DeregisterPrescanRxChain .....	10-11
DeregisterPrescanTxChain .....	10-12
DeregisterRPLBootROM .....	10-13
DeregisterStack .....	10-14

EndCriticalSection	10-15
GetBoundBoardInfo	10-16
GetCriticalSectionStatus	10-17
GetECB	10-18
GetHeldPacket	10-19
GetIntervalMarker	10-21
GetLSLConfiguration	10-22
GetLSLStatistics	10-23
GetMLIDControlEntry	10-24
GetPIDFromStackIDBoard	10-25
GetProtocolControlEntry	10-27
GetStackIDFromName	10-28
GetStartOfChain	10-29
GetTickMarker	10-30
HoldEvent	10-31
HoldPacket	10-32
ModifyStackFilter	10-33
RegisterDefaultStackChain	10-34
RegisterPrescanRxChain	10-36
RegisterPrescanTxChain	10-38
RegisterRPLBootROM	10-40
RPLBootROMInfoStruc Structure	10-41
RegisterStack	10-42
StackInfoStruc Structure	10-42
RelinquishControl	10-44
ResubmitDefault	10-45
ResubmitPrescanRx	10-47
ResubmitPrescanTx	10-49
ReturnECB	10-50
ScanPacket	10-51
ScheduleAESEvent	10-52
AESECB Structure	10-52
SendPacket	10-54
ServiceEvents	10-55
StartCriticalSection	10-56
UnbindStack	10-57
<b>LSL MLID Support Routines</b>	<b>11-1</b>
Overview	11-2
AddProtocolID	11-4
CancelAESEvent	11-5
ControlStackFilter	11-6
DefragmentECB	11-8
DeregisterMLID	11-9
EndCriticalSection	11-10
GetCriticalSectionStatus	11-11

---

GetECB .....	11-12
GetIntervalMarker .....	11-13
GetStackECB .....	11-14
LookAheadStruc Structure .....	11-14
HoldReceiveEvent .....	11-16
ReturnECB .....	11-17
ScheduleAESEvent .....	11-18
AESECB Structure .....	11-18
ServiceEvents .....	11-20
StartCriticalSection .....	11-21
SendComplete .....	11-22
<b>LSL Initialization Routines .....</b>	<b>12-1</b>
Overview .....	12-2
GetEntryPoints .....	12-3
LSLInitEntryPointBlock Structure .....	12-3
GetMLIDSupportEntry .....	12-4
GetProtocolSupportEntry .....	12-5
RegisterMLID .....	12-6
Information Block on Entry .....	12-6
Information Block on Return .....	12-7
<b>LSL General Services .....</b>	<b>13-1</b>
Overview .....	13-2
AddGeneralService .....	13-3
GenServiceControlBlock Structure .....	13-4
AddMemoryToPool .....	13-6
AllocateMemory .....	13-7
FreeMemory .....	13-8
GetNETCFGPath .....	13-9
GetServiceChain .....	13-10
MemoryStatistics .....	13-11
MemStatStruc Structure .....	13-11
ReallocateMemory .....	13-12
RemoveGeneralService .....	13-13
<b>Multiple Link Interface Drivers (MLIDs) .....</b>	<b>IV-1</b>
Section Overview .....	IV-2
<b>Overview of the MLID .....</b>	<b>14-1</b>
Overview .....	14-2
ODI MLID .....	14-3
MLID Procedures .....	14-3
MLID Initialization .....	14-4
Board Service Routine .....	14-4
Packet Transmission .....	14-4
Control Routines .....	14-4
Timeout Detection .....	14-5

Driver Remove .....	14-5
Events .....	14-5
MLID Data Structures and Variables .....	14-5
MLID Configuration Table .....	14-5
MLID Statistics Table .....	14-5
MLID Functionality .....	14-6
Multiple Frame Support .....	14-6
Source Routing Support .....	14-7
Promiscuous Mode Support .....	14-7
Multicast Addressing Support .....	14-8
MLID Design Considerations .....	14-8
Hardware Issues .....	14-8
<b>MLID Data Structures .....</b>	<b>15-1</b>
Overview .....	15-2
Frame Data Space .....	15-3
MLID Configuration Table .....	15-3
MLID Configuration Table Sample Code .....	15-4
Configuration Table Flags .....	15-12
Adapter Data Space .....	15-17
MLID Statistics Table .....	15-17
MLID Statistics Table .....	15-18
<b>MLID Initialization .....</b>	<b>16-1</b>
Overview .....	16-2
MLID Initialization .....	16-3
<b>MLID Packet Reception and Transmission .....</b>	<b>17-1</b>
Overview .....	17-2
MLID Packet Reception .....	17-3
Lookahead Buffer .....	17-4
Shared RAM .....	17-5
Programmed I/O .....	17-5
DMA .....	17-5
MLID Packet Transmission .....	17-5
<b>MLID Control Routines .....</b>	<b>18-1</b>
Overview .....	18-2
AddMulticastAddress .....	18-3
Ethernet Multicasts .....	18-4
Token-Ring Multicasts .....	18-4
Number of Supported Multicast Addresses Supported .....	18-5
DeleteMulticastAddress .....	18-6
DriverManagement .....	18-7
DriverPoll .....	18-9
GetMLIDConfiguration .....	18-10
GetMLIDStatistics .....	18-11
GetMulticastInfo .....	18-12

MLIDReset .....	18-14
MLIDShutdown .....	18-15
PromiscuousChange .....	18-17
SetLookAheadSize .....	18-19
RegisterTxMonitor .....	18-20
Transmit Monitor .....	18-21
Transmit Control Block (TCB) .....	18-21
<b>Appendixes .....</b>	<b>V-1</b>
<b>Event Control Blocks (ECBs) .....</b>	<b>A-1</b>
Overview .....	A-2
Event Control Block Structure Sample Code .....	A-3
<b>Compatibility with Multitasking DOS Products .....</b>	<b>B-1</b>
Globally Accessible Data Buffers .....	B-2
Microsoft Windows 386 Enhanced Mode .....	B-2
<b>The 802.2 Type II Frame Header .....</b>	<b>C-1</b>
Overview .....	C-2
Support of the 802.2 Type II Frame .....	C-3
Packet Transmission .....	C-3
Packet Reception .....	C-3
<b>Promiscuous Mode .....</b>	<b>D-1</b>
Overview .....	D-2
Implementing Promiscuous Mode .....	D-3
Implementation Considerations .....	D-3
Multitasking .....	D-3
Packet Sequencing .....	D-3
ECB Does not Contain the Entire Packet .....	D-4
Error Packets .....	D-4
Logical vs. Physical .....	D-4
<b>The NET.CFG Configuration File .....</b>	<b>E-1</b>
Overview .....	E-2
Main Section Headings .....	E-3
Locating the NET.CFG File's Directory .....	E-3
LAN Driver Keywords and Parameters .....	E-4
DMA [#selection] Channel .....	E-4
IRQ [#selection] Interrupt .....	E-4
MEM [#selection] Address [length] .....	E-4
Port [#selection] Address [length] .....	E-4
Slot n .....	E-5
Node Address h [format] .....	E-5
Protocol <name> h <frame type> .....	E-5
Frame <name> Address Mode .....	E-6
Custom Keywords .....	E-6
<b>Glossary .....</b>	<b>Gloss-1</b>
<b>Index .....</b>	<b>Index-1</b>



## Preface

### Document Organization

This document describes the ODI architecture, which consists of three main elements: protocol stacks, the LSL and the LAN driver (also called Multiple Link Interface Driver or MLID). This document is organized into sections that discuss each element of the architecture individually. The document contains the following sections.

#### **Section I Introduction**

Introduces the ODI architecture and discusses the design issues relevant to the ODI architecture as it applies to the NetWare environment.

#### **Section II Protocol Stacks**

Explains the architecture of an ODI protocol stack and discusses the design issues relevant to a stack. This section also discusses protocol stack data structures, initialization, packet reception and transmission, and control routines.

#### **Section III LSL**

Presents a brief overview of the LSL and describes its configuration and statistics tables. This section also includes descriptions of the general LSL support routines, the Multiple Protocol Interface (MPI) support routines, and Multiple Link Interface (MLI) support routines.

#### **Section IV MLIDs**

Explains the architecture of an ODI MLID and discusses the design issues relevant to an ODI MLID. This section also discusses MLID data structures, initialization, packet reception and transmission, and control routines.

#### **Section V Appendixes**

### Referenced Documents

This document refers to the following Novell products and documents.

*LAN Driver Developer's Kit*, Part number 172-000029-001

*NESL Specification: 16-Bit DOS Client Programmer's Interface*, part number 107-000066-001

*Novell ODI Specification: 16-Bit DOS Client HSMs*, part number 107-000054-001

*ODI Specification Supplement: Canonical and Noncanonical Addressing*, part number 107-000059-001

*ODI Specification Supplement: Frame Types and Protocol IDs*, part number 107-000055-001

*ODI Specification Supplement: Source Routing*, part number 107-000058-001

*ODI Specification Supplement: Standard MLID Message Definitions*, part number 107-000060-001

*ODI Specification Supplement: The Hub Management Interface*, part number 107-000023-001

*ODI Specification Supplement: The MLID Installation Information File*, part number 107-000056-001

## Equates and Structures

MinNumECBs	equ 0	; LSL default
MaxNumECBs	equ 50	; Server's max /40
MinECBDataSize	equ 512 + 74 + 52	; 512 data + 74 protocol header + 52 MAC headers.
MaxECBDataSize	equ 17954	; Largest Token-Ring setting
DefaultNumMaxBoards	equ 4	; LSL default
DefaultNumMaxStacks	equ 4	; LSL default
DefaultECBDataSize	equ 1514	; 1514 = 1388 data + 74 protocol headers + 52 MAC headers

## LSL Initialization Entry Function Equates

LSLINIT_MLID_REG	equ 1	; Registers MLID, ; DS:DI → MLIDRetInfoBlockStruc
LSLINIT_GET_PROTSUP_ENTRY	equ 2	; ES:SI → LSL entry point Info Block
LSLINIT_GET_MLID_ENTRY	equ 3	; ES:SI → MLID support entry point
LSLINIT_GET_ENTRY_POINTS	equ 4	; ES:SI → Extended EntryPointInfoBlock (V2.10 and above.)

## Registration Structures

```

MLIDInfoBlockStruc  struc
    MIBS_SendEntry   dd  ?
    MIBS_ControlEntry dd  ?
    MIBS_ConfigTable dd  ?
MLIDInfoBlockStruc  ends

MLIDRetInfoBlockStruc struc
    MRIBS_MLID_SUP   dd  ?
    MRIBS_BoardNum   dd  ?
    MRIBS_ECB_DataSize dd ?
MLIDRetInfoBlockStruc ends

LSLInitEntryPointBlock  struc
    LSLProtSupEntryPt   dd  0   ; Protocol Support Entry (MPI API)
    LSLGenSupEntryPt    dd  0   ; General Services Entry
    LSLMLIDSupEntryPt   dd  0   ; MLID Support Entry Point (MLI API)
LSLInitEntryPointBlock  ends
    
```

## MemStatStruc Structure

```

MemStatStruc  struc
    MemAvail      dw  0   ; in paragraphs
    MemInUse      dw  0   ; in paragraphs
    LargestAvailBlk dw  0   ; in paragraphs
    NumAvailBlocks dw  0
    MemOverhead   dw  0   ; in bytes per allocation
    MinAllocation dw  0   ; in bytes
MemStatStruc  ends
    
```

## StackChainStruc Structure

```

StackChainStruc  struc
    StkChnLink      dd  0   ; Link Field
    StkChnBoardNum  dw  -1   ; Logical Board Number
    StkChnPositReq  dw  STACK_REQ_DEPENDS ; Chain
                                                Position Requested
    StkChnHandler   dd  ?   ; Stack's Tx or Rx Handler
    StkChnControl   dd  ?   ; Stack's Control Entry Point
    StkChnID        dw  0   ; Stack's ID Number
    StkChnMask      dw  83h  ; Default (broadcast, multicast, and
                                                direct)
    StkChnReserved  dd  0   ; Reserved
StackChainStruc  ends ; (Size 24 bytes)

STACK_REQ_FIRST      equ  0
STACK_REQ_NEXT_FIRST equ  1
STACK_REQ_DEPENDS    equ  2
STACK_REQ_NEXT_LAST  equ  3
STACK_REQ_LAST       equ  4
    
```

## LookAheadStruc Structure

```

LookAheadStruc          struc
    LMediaHeaderPtr     dw    2    dup (?)
    LookAheadPtr        dw    2    dup (?)
    LookAheadLen        dw    ?
    LProtID             db    6    dup (?)
    LBoardNum           dw    0
    LDataSize           dw    ?
    LImmAddress         db    6    dup (?)
    LPacketAttrib       dw    0
    LDestType           dw    0
    LStartCopyOffset   dw    0
    LPriorityLevel      db    0
    LRESERVED           db    3    dup (?)
LookAheadStruc          ends

```

## LDestType Bit Definitions

```

DEST_MULTICAST          equ 0001h
DEST_BROADCAST          equ 0002h
DEST_REMOTE_UNICAST     equ 0004h
DEST_REMOTE_MULTICAST  equ 0008h
DEST_SOURCE_ROUTE      equ 0010h ; source route info
DEST_ERRORED           equ 0020h ; super exclusive bit
DEST_MAC_FRAME         equ 0040h ; exclusive bit
DEST_DIRECT            equ 0080h
RX_NOT_8022            equ 0000h ; non-802.2 packet
RX_8022_TYPE1          equ 0100h ; 802.2 type 1 packet
RX_8022_TYPE2          equ 0200h ; 802.2 type 2 packet
RX_PRIORITY_FRAME      equ 0400h ; priority level is indicated
DEST_PROMISCUOUS      equ 0FFFh ; all packets. (filter mask
                                set by protocols to get all
                                packets, including errors)

```

## LPacketAttrib Bit Definitions

```

PKT_CRC_ERR            equ 0001h ; CRC error / (FCS error)
PKT_ALIGN_ERR         equ 0002h ; CRC/frame alignment error
PKT_RUNT_ERR          equ 0004h ; runt packet
PKT_BIG_ERR           equ 0010h ; packet larger than media
                                allowed
PKT_TYPE_ERR          equ 0020h ; packet for unsupported
                                frame type
PKT_MALFORM_ERR       equ 0040h ; malformed packet
IMM_ADDR_MSB_FORM     equ 8000h ; immediate address is MSB
                                format.

```

## Promiscuous State Flag Bit Definitions

```

PROM_OFF               equ 00h
PROM_MAC               equ 01h
PROM_NON_MAC           equ 02h
PROM_STAT              equ 04h

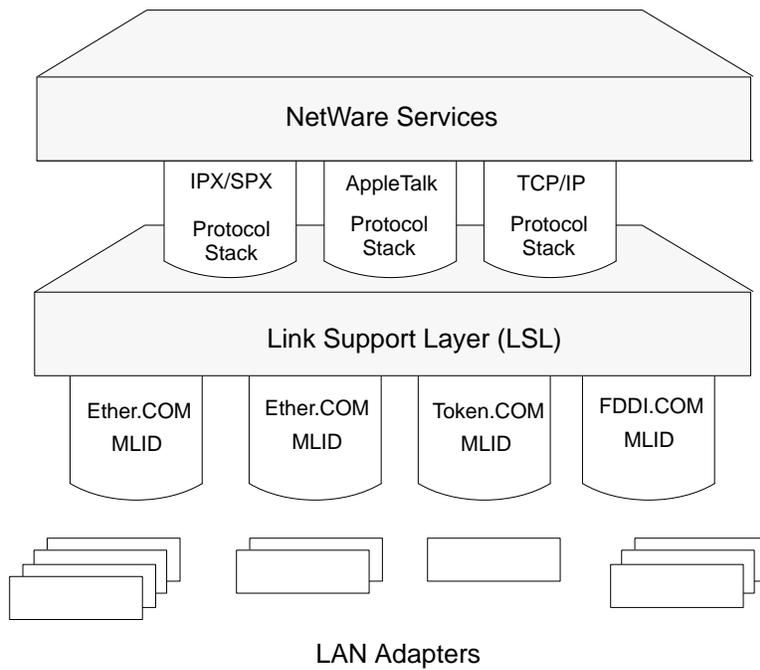
```

□





Section I **ODI Architecture**

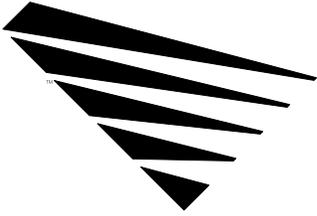


## Section Overview

This introduction briefly describes the Open Data-Link Interface (ODI) specification. It also briefly describes the design, programming, and functionality factors you must understand to write MLIDs and protocol stacks.

*Chapter 1: Introduction to ODI* briefly describes the Open Data-Link Interface (ODI) specification. You should read this chapter if you are not familiar with the basic concepts involved in the ODI specification.





*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.

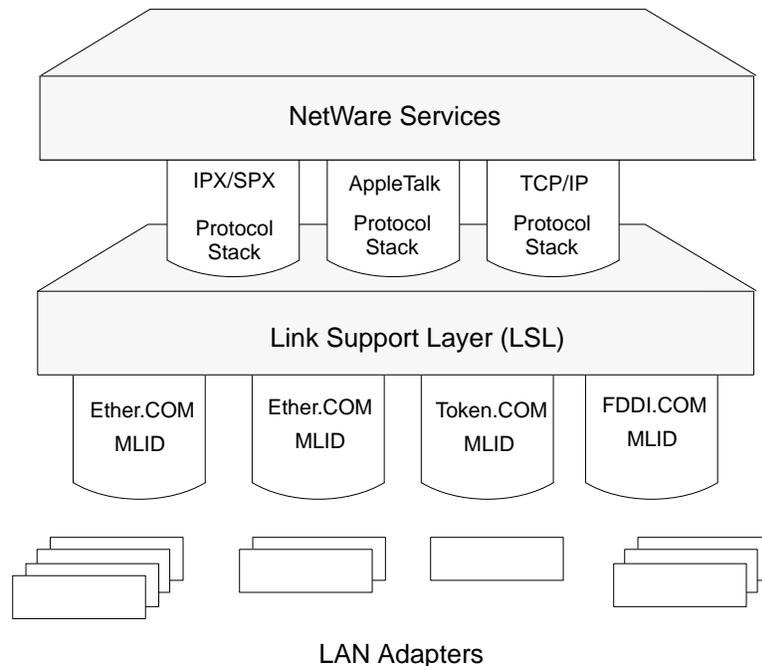
**Note** The terms *MLID* and *LAN driver* can be interchanged.

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

## Open Data-Link Interface (ODI)

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

**Figure 1.1**  
The ODI Specification



The ODI specification allows multiple network protocols and adapters (physical boards) to be used concurrently on the same client 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 .

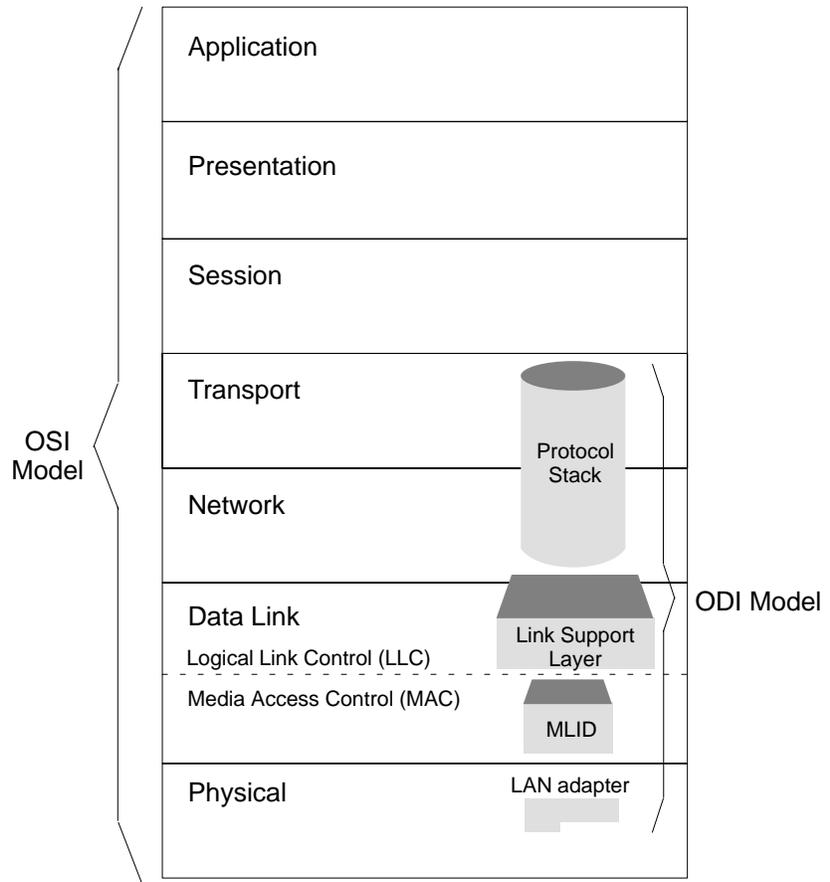
- Protocol Stacks
- Link Support Layer (LSL)
- Multiple Link Interface Drivers (MLIDs)

### 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.

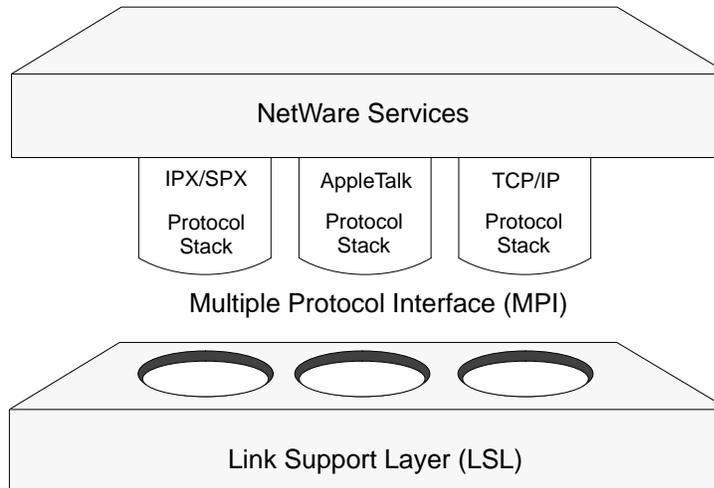
**Figure 1.2**  
**How ODI Fits into**  
**the OSI Model**



### **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.

**Note** The term “LAN adapter” applies to any network controller that provides access across a network. This network controller is as likely to be present directly on the motherboard of a computer in an embedded system as it is on a network interface card that inserts into a computer bus.

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.

## 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.

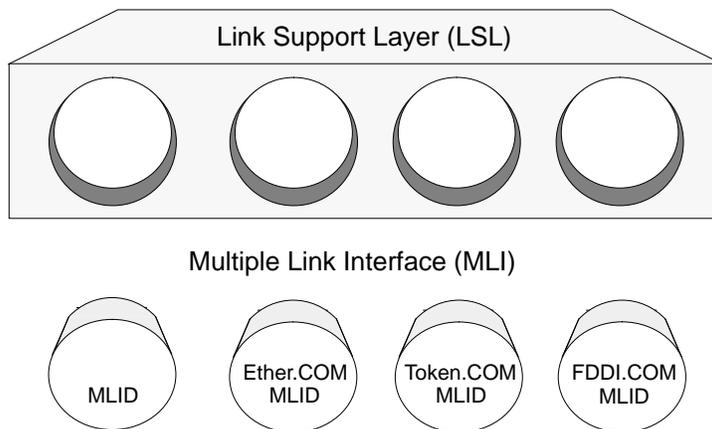
Each MLID's interface with the LAN adapter is determined by that adapter's hardware.

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)



## Data Flow

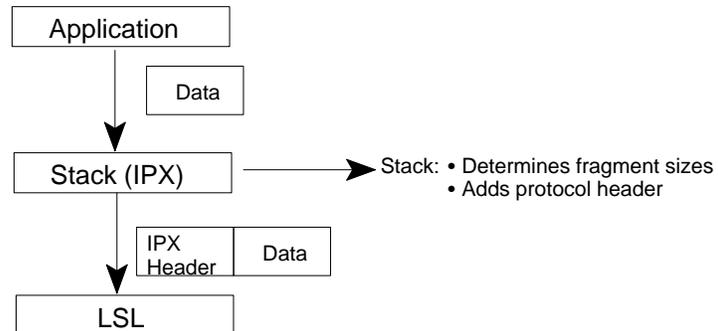
When messages are sent and received, the various protocols or layers add and remove their own information at each layer. The following diagrams illustrate basic data flow.

### Send Data Flow

As Figure 1.5 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

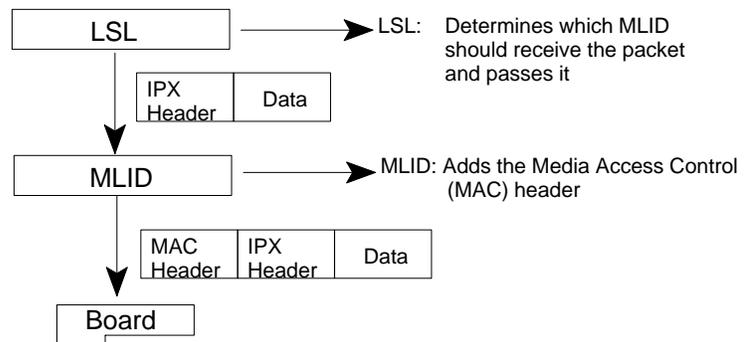
the appropriate MLID, which then takes care of the topology-specific information. This is the reason ODI protocol stacks are known as being media and frame-type independent.

**Figure 1.5**  
Data Flow from  
Application to LSL



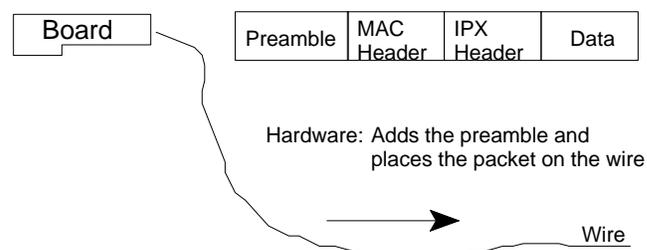
As illustrated by Figure 1.6 , the LSL directs the packet to the appropriate MLID. The MLID then adds the MAC header to the packet and hands the packet to the LAN adapter.

**Figure 1.6**  
Data Flow from the  
LSL to the Board



In Figure 1.7 the hardware adds the preamble to the packet and places the packet on the wire.

**Figure 1.7**  
Data Flow from the  
Board to the Wire

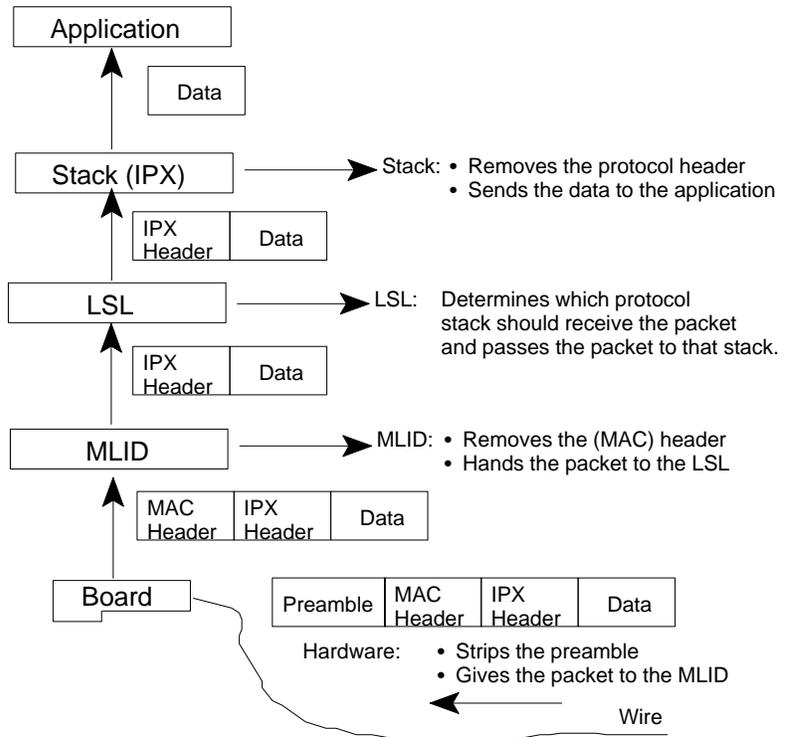


## Receive Data Flow

Figure 1.8 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

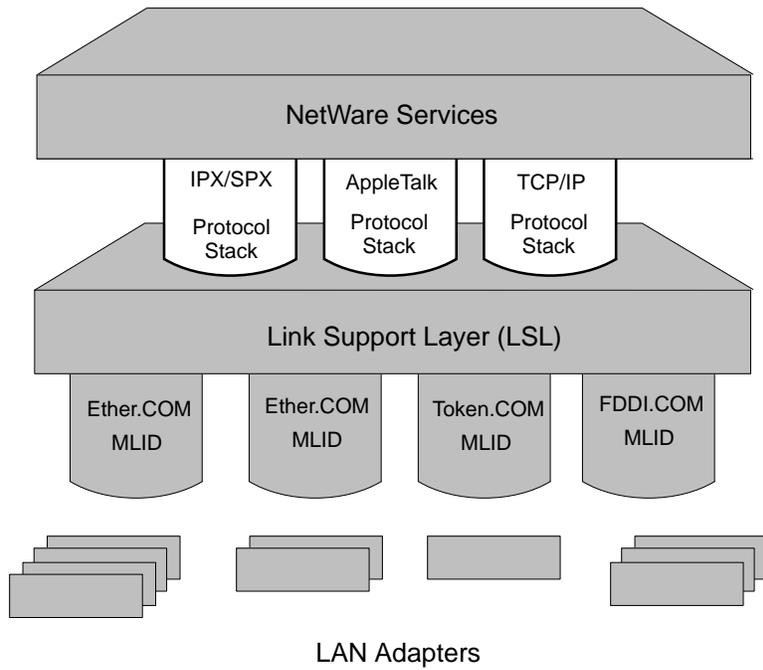
LSL. The LSL directs the packet to the appropriate protocol stack, which then removes the protocol header from the packet and hands the data to the application.

**Figure 1.8**  
Receive Data Flow  
from Wire to Application





Section II **Protocol Stacks**



## Section Overview

This section describes protocol stack initialization, transmission, and reception routines and it also describes protocol stack control routines.

*Chapter 2: Overview of Protocol Stacks* provides an overview of NetWare protocol stack operation. You should read this chapter if you have not developed an ODI protocol stack.

*Chapter 3: Protocol Stack Data Structures* describes the protocol stack's configuration and statistics tables. This chapter provides useful reference material for protocol stack developers.

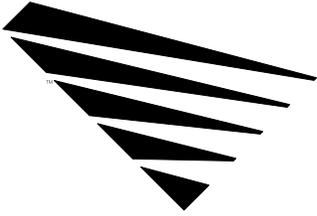
*Chapter 4: Protocol Stack Initialization* discusses registering, binding, and chaining prescan, bound, and default protocol stacks. You should review this chapter before writing the protocol stack initialization routine.

*Chapter 5: Protocol Stack Packet Reception* describes the protocol stack receive routine and details the prescan and default protocol stack receive methods. You should review this chapter before writing the protocol stack receive routine.

*Chapter 6: Protocol Stack Packet Transmission* describes the send operation of the protocol stack. You should review this chapter before writing the protocol stack transmission routine.

*Chapter 7: Protocol Stack Control Routines* describes the commands that your protocol stack must provide to support the MPI interface. This chapter provides useful reference material for protocol stack developers and should be reviewed before writing the protocol stack control routines.

□



*Chapter 2* **Overview of Protocol Stacks**

## DOS Environment

The minimum environment in which your protocol stack is required to operate is a simple real mode DOS environment. However, you should keep in mind that your protocol stack might also be operating with DOS multitasking products such as Microsoft Windows. Therefore, your protocol stack usually needs to be compatible with DOS multitasking products (see *Appendix B: Compatibility with Multitasking DOS Products*).

## Protocol Stack Performance

A protocol stack's performance is extremely important. To achieve the best possible performance, develop the transmit and receive portions of the protocol stack in assembly language.

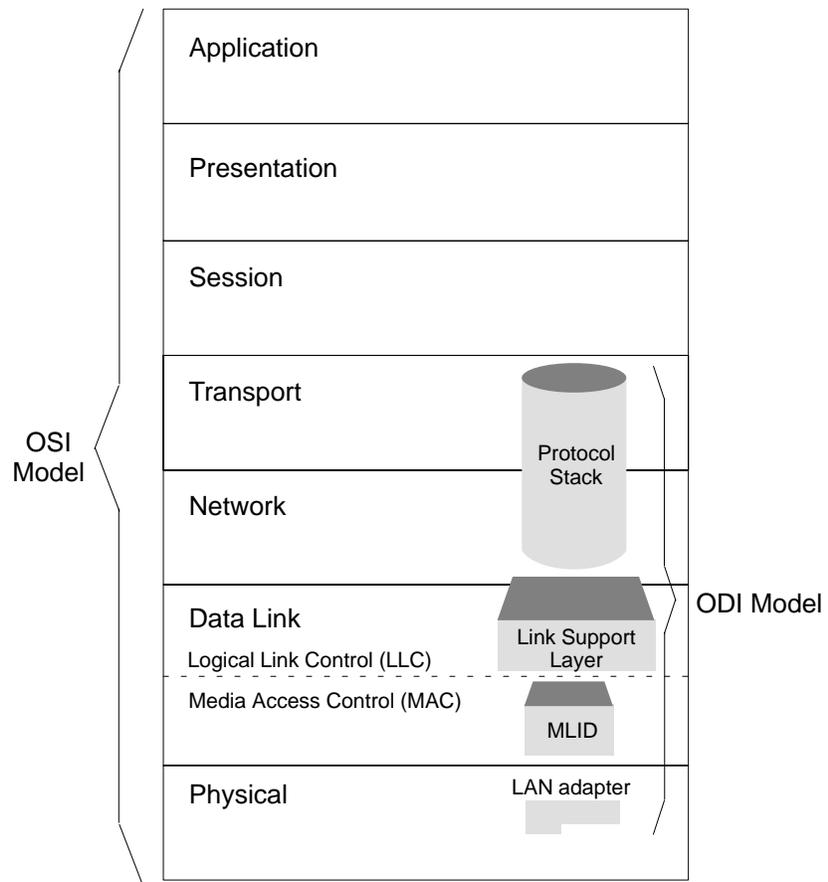
## Network Interface Card Utilization

Each protocol stack minimally utilizes one Network Interface Card (NIC). (The terms NIC, board, adapter, and card can be interchanged.) The protocol stack sees each NIC as a logical board with a corresponding board number. This board number is a handle the protocol stack uses to refer to the NIC when the protocol stack requests the LSL to perform a function on the NIC. Each board number represents a logical NIC operating on a physical board. (A physical board might support more than one logical NIC.) To the protocol stack, logical boards appear identical to physical boards. For example, in some installations, the NIC might be another protocol stack which is nesting your protocol stack inside another protocol which is then transferred onto a physical network. Regardless of the installation, a protocol stack deals with the physical network by using a board number handle.

The ODI specification allows multiple LAN adapters and multiple frame formats on each adapter. Therefore, your protocol stack should be able to service multiple LAN adapters when necessary. As a minimum requirement, your protocol stack will service one logical LAN adapter in the workstation.

Because ODI is a dynamic specification that allows protocol stacks and LAN driver modules to be loaded and unloaded as they are needed, we strongly recommend that your protocol stack be fully unloadable. *Chapter 4: Protocol Stack Initialization* discusses module unloading in detail.

**Figure 2.1**  
How ODI Fits into  
the OSI Model



## Hardware/Media Independence

MLIDs are written to be independent of the network protocol, which allows MLIDs to be compatible with any protocol. Protocol stacks require the same independence; they must be written so that they are independent of the LAN medium and frame type.

The protocol stack's interface to the LSL (the OSI Data Link Layer) is what allows the protocol stack to be independent of the underlying LAN medium and frame type. In other words, the protocol stack can be used on any adapter with any frame format without the need for frame specific code. This allows the protocol stack to be used in environments that traditionally have not supported it. There are some specialized protocol stacks that are written to be frame type aware, but these are exceptions.



### **Important**

We strongly recommend that you develop your protocol stack so that it is LAN medium and frame type independent.

## Protocol Stack Multiplexing

Multiplexing different protocol types on the same frame type requires each frame's protocol type to be uniquely identified. The LSL returns a unique value known as a Protocol ID (PID) for each protocol that uses that frame type. The MLID places this PID value in every transmitted frame, allowing the physical LAN medium to be viewed as many logical networks with each protocol communicating on its own wire.

The location and format of the PID in the frame header is LAN medium and frame type dependent and should not concern the protocol stack. The LAN driver receives the frames and passes them to the LSL which then uses the PID to determine which protocol stack receives the frame.

## Packet Transmission and Reception

A protocol stack uses two system handles to concurrently utilize and service multiple boards in a system:

- The board number specifies the physical network and the frame type.
- The Protocol ID (PID), together with the board number, represents the logical wire used on the physical network.

To send a frame on a network, the protocol stack only needs to know the board number of the NIC that services that network and the PID, which should be embedded into the media frame header.

Frame reception is more involved than frame transmission and requires a protocol stack to be bound to a board in the system. Binding enables the LSL to route incoming frames to the protocol stack by providing the address of the protocol stack's receive handler routine to the LSL. This routine can be called at interrupt time and is entered with interrupts disabled. This routine must not enable interrupts.

## Binding Protocol Stacks

The ODI specification defines three types of protocol stacks:

- Bound protocol stacks
- Prescan protocol stacks
- Default protocol stacks

### Bound Protocol Stacks

Bound protocol stacks are the most common. A bound protocol stack requires that frames received from the LSL have a

unique, registered Protocol ID (PID) embedded in the frame header. (The system administrator can use the appropriate entries in the NET.CFG file to register Protocol IDs for each protocol stack. See *Appendix E: NET.CFG Configuration File*) The appropriate PIDs for a given protocol are usually different for each frame type. *ODI Specification Supplement: Frame Types and Protocol IDs* lists the normal IPX and TCP/IP protocol stack PID values for some of the more common frame types.

The LSL uses the embedded PID in the packet header to locate the appropriate protocol stack to receive the packet. A bound protocol stack receives only the packets that have the registered PID for that stack and that pass the stack filter.

Protocol stacks that contain multiple Network Layer protocols using different PIDs (for example, TCP/IP = IP, ARP, RARP) must be registered to the LSL as separate and distinct protocols. In other words, if a protocol uses more than one Protocol ID, that protocol stack should be logically fragmented and each fragment registered with the LSL as a separate protocol stack. However, these fragments can still be located in the same module and can specify the same receive handler routine. The receive handler routine then determines for which subprotocol the frame is intended by examining the *StackID* field of the frame's Event Control Block (ECB).

The bound protocol stack method allows multiple protocol stacks to service and share a single LAN adapter. This method also minimizes protocol cross-talk, because the protocol type of the packet is not determined by parsing the protocol header.

### **Prescan Protocol Stacks**

Prescan protocol stacks receive all incoming packets from a particular LAN adapter before the packet is routed to the appropriate bound protocol stack. The protocol stack either consumes the frame or allows other protocols to process the frame.

Special purpose protocol stacks such as diagnostic utilities use this method.

### **Default Protocol Stacks**

A default protocol stack receives every frame not claimed by any other protocol stack (prescan or bound). In other words, a default protocol stack receives all leftover frames.

Protocol stacks that provide an alternate Data-Link Layer solution use this method.

## Receiving Packets

A bound protocol stack must be registered with the LSL and be bound to an MLID in order to receive packets from that MLID. Prescan and default stacks must be registered with the LSL in order to receive packets from the MLID. Registration provides the LSL with the information required to route packets from MLIDs to protocol stacks. The following table describes the steps involved in packet reception.

<b>Table 2.1 Packet Reception</b>	
<b>Actor/Agent</b>	<b>Action</b>
MLID	<ol style="list-style-type: none"> <li>1. Upon receipt of a packet, fills out the <i>LookAheadStruc</i> structure.</li> <li>2. Passes the <i>LookAheadStruc</i> structure to the LSL.</li> </ol>
LSL	<ol style="list-style-type: none"> <li>3. Calls registered prescan stacks for the board number. Each prescan stack in turn gets an opportunity to consume the packet.</li> <li>4. In the absence of a prescan stack or if the prescan stack did not consume the packet, uses the board number and Protocol ID to route the packet to the correct, bound protocol stack.</li> <li>5. In the absence of a bound stack or if the bound stack did not consume the packet, calls the first stack in the default stack chain. This procedure continues until the packet is consumed or there are no more stacks.</li> <li>6. In the absence of a default stack or if no ECB was provided, ignores the packet and returns the <i>LookAheadStruc</i> structure to the MLID with instructions to discard the received packet.</li> </ol>

Note, a protocol stack consumes a packet by providing an ECB.

A protocol stack can be bound to any number of MLIDs. An MLID can be bound to multiple stacks.

## Priority Packet Support

The MLID sets the *MPrioritySup* field in the MLID configuration table to indicate the number of priority levels available. The MLID indicates that priority support is active by setting or clearing the *PrioritySupportBit* bit in the *MFlags* field of the MLID configuration table. The MLID can set or reset the *PrioritySupportBit* bit as the MLID changes from Priority Queue Support Enabled to Disabled states. The *PrioritySupportBit* bit is checked prior to queueing a packet by the MLID.

The protocol sets the *ProtoNum* (StackID) field to a value greater than or equal to 0FFF0h. The values have the following meanings:

0FFFFh	raw send packets; no priority.
0FFFEh – 0FFF8h	raw send packets; priority level 1–7.
0FFF7h	non-raw send packets; no priority.
0FFF6h – 0FFF0h	non-raw send packets; priority level 1–7.

Priority levels are defined as 0 = no priority and 7 as high priority. To extract the priority level, NEG (2's complement) the ProtNum (StackID) field, and AND it with 07h. The resultant will be a number from 0 to 7 with 0 = No Priority, and 7 being the highest Priority.

The MLID will normally send the packet directly. If the MLID is busy and the transmit ECB to be sent is a priority transmit ECB, the MLID will either queue it in a priority queue for transmission soon, or transmit the packet out a priority channel.

After the MLID has transmitted the priority ECB, the MLID calls the transmit monitor (if it is registered), increments the necessary counters, and calls *SendComplete* to return the ECB to its original owner.

## Stack Filtering

As a protocol registers with the LSL, a default filter is assigned to only allow directed, multicast, and broadcast frames to be presented to the protocol stack. Old MLIDs do not use the current *LookAheadStruc* structures, which have the filter information included, so the LSL will use the old rules for this type of MLID; in other words, only these three types of frames would ever be presented by the LSL. A protocol wanting to see only errored frames would make the call to the modify stack filter function and set the pass filter to accept errored packets.

Protocols written to handle only custom types of frames can reliably use the filter to screen for desired frames. The following packet types can be selected:

- Multicast
- Broadcast
- Remote unicast
- Remote broadcast
- Source route
- MAC frames
- Direct

- Priority frames





*Chapter 3* **Protocol Stack Data Structures**

## **Overview**

This chapter describes the protocol stack's configuration and statistics tables.

This chapter provides useful reference material for protocol stack developers.

## Protocol Stack Configuration Table

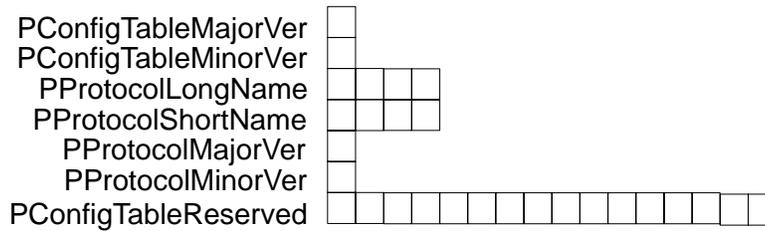
The following describes the protocol stack configuration table in detail. It includes sample code and a description of the protocol stack configuration table fields.

### Protocol Stack Configuration Table Structure Sample Code

```

ProtocolConfigStructure  struc
    PConfigTableMajorVer  db  01
    PConfigTableMinorVer  db  00
    PProtocolLongName     dw  ?,?
    PProtocolShortName    dw  ?,?
    PProtocolMajorVer     db  ?
    PProtocolMinorVer     db  ?
    PConfigTableReserved  db  16 dup (0)
ProtocolConfigStructure  ends
    
```

**Figure 3.1**  
Graphic Representation  
of the Protocol Stack  
Configuration Table



**Table 3.1 Protocol Stack Configuration Table Field Descriptions**

Offset	Name	Size (in bytes)	Description
00h	PConfigTableMajorVer	01	This field has the major version number of the configuration table (1 for this specification).
01h	PConfigTableMinorVer	01	This field number has the minor version number of the configuration table (0 for this specification).
02h	PProtocolLongName	04	This field contains a far pointer to a length-preceded ASCII protocol stack description string and cannot be 0.
06h	PProtocolShortName	04	This field contains a far pointer to a length-preceded ASCII string used to register the protocol stack. This string cannot have more than 15 characters (not including the length byte or zero terminator) and cannot be 0.
0Ah	PProtocolMajorVer	01	This field has the major version number of the protocol stack. The number in this field is a decimal number (0 through 99).
0Bh	PProtocolMinorVer	01	This field has the minor version number of the protocol stack. The number in this field is a decimal number (0 through 99).
0Ch	PProtocolReserved	16	This field is reserved for future use and must be initialized to 0.

## Protocol Stack Statistics Table

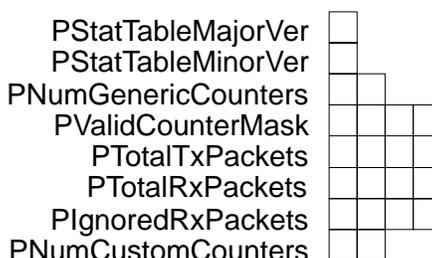
All protocol stacks must keep a statistics table for the purpose of network management. The following contains a sample of the statistics table code and a description of each of the fields in the statistics table.

### Protocol Stack Statistics Table Structure Sample Code

```

ProtocolStatStructure      struc
    PStatTableMajorVer     db  1
    PStatTableMinorVer     db  00
    PNumGenericCounters    dw  3
    PValidCounterMask      dd  0001111111111111111111111111111b
    PTotalTxPackets        dw  2 dup (0)
    PTotalRxPackets        dw  2 dup (0)
    PIgnoredRxPackets      dw  2 dup (0)
    PNumCustomCounters     dw  0
ProtocolStatStructure      ends
    
```

**Figure 3.2**  
Graphic Representation  
of the Protocol Stack  
Statistics Table



**Table 3.2 Protocol Stack Statistics Table Field Descriptions**

Offset	Name	Size (in bytes)	Description
00h	PStatTableMajorVer	01	This field has the major version number of the statistics table (0 through 99 decimal); 1 for this specification.
01h	PStatTableMinorVer	01	This field has the minor version number of the statistics table (0 through 99 decimal); 0 for this specification.
02h	PNumGenericCounters	02	This field has the number of dword counters in the static portion of this table. This field should be set to 0003h for this specification.
04h	PValidCountersMask	04	This field contains a bit mask that indicates which generic counters are used. The value 0 indicates Yes; the value 1 indicates No. The bit/counter correlations are determined by shifting left as you move down the counters in the table. For example, bit 31 represents <i>TotalTxPackets</i> .
08h	PTotalTxPackets	04	This field has the total number of <i>SendPacket</i> requests made to the LSL.
0Ch	PTotalRxPackets	04	This field contains the total number of incoming packets that were consumed by the protocol stack.

**Table 3.2 Protocol Stack Statistics Table Field Descriptions**

Offset	Name	Size (in bytes)	Description
10h	PIgnoredRxPackets	04	This field has the total number of times the protocol receive handler was called with look ahead data, and the protocol stack did not return a receive ECB.
14h	PNumCustomCounters	02	This field contains the total number of custom variables that follow this word.

There are *PNumCustomCounters* dwords that start at offset 16h and that correspond to the custom statistics for the protocol stack. Following these dwords, *PNumCustomCounters* pointers (4 bytes each) point to length-preceded ASCII strings that describe each custom counter.

### Example

If *PNumCustomCounters* = 2:

```

CustomCounter1    dd 0
CustomCounter2    dd
                  dd offset: Segment pointer to custom counter
                  string
                  dd offset: Segment pointer to custom counter
                  string
CustomCounterString1 db "This is Counter 1"
CustomCounterString2 db "This is Counter 2"

```

□





*Chapter 4* **Protocol Stack Initialization**

## **Overview**

Initializing the protocol stack involves the following general steps:

1. Locate the LSL.
2. Register the protocol stack.
3. Determine which logical board(s) to service.
4. Obtain the Protocol ID value(s).
5. Customize the protocol stack.
6. Bind the protocol stack to the logical board(s).

If your protocol stack is to become resident, it should free all the memory it used to hold the initialization code and initialization data before turning resident.

## Locating the LSL

The LSL module must reside in the system before the user can load any protocol stacks.

The process usually used to boot an ODI system is outlined below:

```

LSL.COM           ;Load LSL
NE1000.COM        ;Load MLID(s)
IPXODI.COM        ;Load protocol(s)
VLM.EXE
    
```

A protocol stack must first obtain the LSL API entry points in order to initialize. Table 4.1 describes the procedure to find these entry points.

**Table 4.1 Finding LSL API Entry Points**

Actor/Agent	Action
Protocol Stack	1. Scans the DOS interrupt 2F multiplex slots for a signature string. (Scanning is required because the actual interrupt 2F slot used by the LSL is dynamic.)
LSL	2. Returns an error to the operating system if the protocol stack fails to find the API entry points.
Protocol Stack	3. Returns a far pointer to the LSL's initialization entry point after the protocol stack locates the correct interrupt 2F slot.
LSL	4. Calls the LSL's initialization entry point to request support entry points.
Protocol Stack	5. Returns far pointers to the LSL's protocol stack API entry point and to the LSL's general services API entry point. (The API specification for both of these entry points can be found in <i>Chapter 10: LSL Protocol Stack Support Routines</i> .)

The following sample code illustrates how to locate the LSL and also shows how to obtain the protocol stack support and general services API entry points after the LSL is located.

### Sample Code for Locating the LSL

```

Signature          db    'LINKSUP$'
LSLInitHandler     dd    0
LSLEntryPoints     dd    0          ; Protocol Support API entry point
                  dd    0          ; General Services API entry point
    
```

```

FindLSL          proc    near
    sub          ax, ax          ;    Make sure vector is valid
    mov         es, ax
    mov         ax, word ptr es:[2Fh*4]
    mov         cx, word ptr es:[2Fh*4+2]
    or          ax, cx          ;    Vector 0?
    jz          LSLIsNotLoaded
    mov         ax, 0C000h      ;    AH = Slot (Start after DOS reserved)

LookForLSLLoop:
    push        ax
    push        ds
    int         2Fh
    cmp         al, 0FFh       ;    Slot used?
    pop         ds
    pop         ax
    je          SlotInUse

TryNextSlot:
    inc         ah              ;    Next slot (go until wrap)
    jnz        LookForLSLLoop
    jmp        LSLIsNotLoaded

;    If LSL then ES:SI points at 'LINKSUP$' string and
;    DX:BX points at LSL initialization entry point

SlotInUse:
    ;          ;    See if this is the LSL's slot
    mov         di, si         ;    See if signature present
    mov         si, offset Signature
    mov         cx, 4          ;    Compare 8 bytes
    rep        cmpsw
    jnz        TryNextSlot

;    Found the LSL's slot. Get the Protocol Stack support and General Services API entry points.
    mov         word ptr LSLInitHandler, bx
    mov         word ptr LSLInitHandler+2, dx
    push        ds
    pop         es             ;    ES:SI points at buffer to hold two far pointers
    mov         si, offset LSLEntryPoints
    mov         bx, LSLINIT_GET_PROTSUP_ENTRY (2);    Function #2-Request support entry points
    call        LSLInitHandler ;    BX destroyed (all other registers preserved)
    sub         ax, ax         ;    Return success
    ret

LSLIsNotLoaded:
    mov         ax, Error      ;    Return error
    ret
FindLSL          endp

```

The protocol stack calls the LSL initialization entry point to obtain the LSL services entry points for the MLID and the protocol stack. These entry points depend upon the input parameters contained in the BX register.

**Table 4.2 Input and Return Parameters of Register BX**

If BX Equals:	Action	Return Parameters
LSLINIT_MLID_REG (01h)	Request MLID registration.	DS:DI has a pointer to the <i>MLIDRetInfoBlockStruc</i> structure.
LSLINIT_GET_PROTSUP_ENTRY (02h)	Request protocol stack and general service API entry points.	ES:SI has a pointer to the LSL information block.
LSLINIT_GET_MLID_ENTRY (03h)	Request MLID API entry point.	ES:SI has a pointer to the MLID support entry point.
LSLINIT_GET_ENTRY_POINTS (04h)	Request all the LSL's API entry points.	ES:SI points to <i>LSLInitEntryPointBlock</i> .

Note, the LSL information block is a subset of the *LSLInitEntryPointBlock* structure.

## Registering with the LSL

After the protocol stack has located the LSL, the protocol stack must register itself with the LSL. This accomplishes the following two items:

- Gives the LSL pointers to the protocol stack's receive handler and to the protocol stack's control handler
- Dynamically assigns a unique stack ID to the protocol stack

The following table illustrates how the receive and control handlers and the stack ID are used.

**Table 4.3 Using the Receive and Control Handlers and the Stack ID**

Actor/Agent	Action
LSL	1. Calls the protocol stack's receive handler whenever a packet is received that is destined to that particular protocol stack.
Applications and LSL	2. Call the protocol stack's control handler to obtain configuration information and to issue defined control functions.
LSL	3. Assigns the stack ID when the protocol stack registers. The LSL uses the stack ID to track the protocol stack.

The bound protocol stack registers by invoking the *RegisterStack* function as defined in *Chapter 10: LSL Protocol Stack Support Routines*.

If the protocol stack is using the prescan or default receive methods (see *Chapter 5: Protocol Stack Packet Reception*), it should register using *RegisterPrescanRxChain* or *RegisterDefaultStackChain* respectively. The LSL assigns a

stack ID to default and prescan protocol stacks and pointers to the protocol's receive and control handlers are still necessary.

## Determining Which Logical Boards to Service

Because an ODI system can have multiple LAN adapters and each adapter can have multiple frame types enabled, protocol stacks must determine to which boards to bind and service. For example, a user might have two LAN adapters with each adapter enabled for four frame types; this translates into eight logical boards registered with the LSL. The user must then tell the protocol stack to which boards to bind. Protocol stacks can determine to which boards to bind by using either the explicit method or the dynamic method; they should support both methods.

### Explicit Method

Using this method, the user explicitly specifies to which logical boards the protocol stack should bind. We suggest that for each protocol entry the user specify a "BIND" entry in the NET.CFG file that looks like the following:

```
BIND #1  
BIND #2
```

Board numbers are displayed when each MLID is loaded.

**Note** Remember to decrement the board numbers obtained from the NET.CFG by one before using the number internally.

The protocol stack should first verify whether a specified board exists and whether a Protocol ID (PID) is available for the protocol that uses that particular board. The protocol stack can verify that a board number exists by calling the LSL protocol support function, *GetMLIDControlEntry*. If the board is valid, the protocol stack should determine whether a PID exists for the protocol on that particular board by calling the *GetPIDFromStackIDBoard* function. If a PID is not present for that protocol, the protocol stack should either add a PID to use or abort the initialization procedure.

### Dynamic Method

If no bind information is specified in the NET.CFG, the protocol stack should scan for a boards to which to bind. The protocol stack should scan through all the possible board numbers, starting with board 0, and call *GetMLIDControlEntry*, which returns whether or not the specified board number exists. The protocol stack should continue scanning and calling

*GetMLIDControlEntry* until the message *LSLERR\_NO\_MORE\_ITEMS* (8003h) is returned. The protocol stack then knows that no boards exist at any higher board numbers. When the protocol stack encounters an active board, the stack should query the LSL for a PID by calling the *GetPIDFromStackIDBoard* support function. If the protocol stack cannot find a board that has a PID for it, the protocol stack should either add a PID to be used or abort the initialization procedure.

## Adding Protocol IDs

You should write your protocol stacks so that they are LAN medium and frame type unaware. Because Protocol ID (PID) values are determined by the frame type and LAN medium on which they are used, your protocol stack should not interpret the PID. Usually, the user of your protocol stack will enter into the NET.CFG file your protocol stack's PID for each frame type board combination. The MLIDs then register each link driver MLIDname . . . protocol entry to the LSL. As discussed in the "Explicit Method" and the "Dynamic Method" sections above, the protocol stack obtains the PID by calling *GetPIDFromStackIDBoard*. Your protocol stack can register an appropriate PID for each board it binds to. This procedure eases the system configuration for the user because the user does not need to enter any PID values for your protocol stack in the NET.CFG file.

To add a PID, the protocol stack should know the common PID value for each of the frames currently defined—for example, ETHERNET\_802.2, TOKEN\_RING, NOVELL\_RX-NET, etc.. See *ODI Specification Supplement: Frame Types and Protocol IDs* for a list of the current frame types.

Before the PID is added, the protocol stack should determine whether a PID has previously been registered for that stack on that particular board. The protocol stack determines this by calling *GetPIDFromStackIDBoard*. If this call returns a PID, the protocol stack should use it. If a PID is not returned, the protocol stack should look at the MLID's configuration table *FrameType* field (see *Chapter 15: MLID Data Structures*) to determine whether the protocol stack has a known (default) PID for that frame type. If the protocol stack does have a known PID for that frame type, the protocol stack calls *AddProtocolID*. If the protocol stack does not have a known Protocol ID for that frame type (for example, perhaps a new frame type is being used), the protocol stack should return an

error to the user stating that a PID must be entered into the NET.CFG file.

In summary, a protocol stack can add a PID to the LSL for a particular board if the following two conditions are true:

- A PID for the protocol stack on a particular board has not been previously registered (determined by *GetPIDFromStackIDBoard*).
- The protocol stack is internally aware of a PID for the board's frame type. (For example, IPXODI's PID on frame type ETHERNET\_802.2 is usually E0h, and TCP/IP's IP PID on frame type Ethernet\_II is usually 800h.)

Checking for a PID before registering your own PID allows the protocol stack's default PID to be overridden. This is an important feature because some users might want your protocol stack to use a different PID.

## Multiple Board Support

The ODI specification allows a protocol stack to be simultaneously bound to multiple boards. Whether or not your protocol stack supports multiple boards or is limited to servicing only one board is for you to decide.

## Obtaining Protocol ID Value(s)

The protocol stack usually handles this step when the stack is determining to which board it can bind. The *GetPIDFromStackIDBoard* function returns the assigned Protocol ID (PID) for that protocol stack on the specified board. A protocol stack only needs the PID values when it sends packets and when it registers with the LSL. A protocol stack should not interpret the PID in any way so that LAN medium and frame type independence is maintained. The protocol stack should simply save the obtained PIDs for later use when transmitting packets.

## Customizing a Protocol Stack

One of the ODI specification's goals is to keep the protocol stack interface to the LSL and the underlying MLIDs as general and independent of issues specific to LAN adapters as possible. However, there are still a number of issues that must be dealt with during initialization. This means that your protocol stack must be customized to the particular capabilities of the underlying MLIDs and the associated LAN adapters. (If the protocol is interested in supporting NESL events, see *NESL Specification: 16-Bit DOS Client Programmer's Interface*.)

## Line Speed

Most LAN media provide high speed data transfer rates (for example, 1M to 100M bits per second). Protocol stacks that retry transmit operations when they do not receive an expected acknowledgement within a specific period of time might have to customize timeout values so that they are appropriate to the speed of the underlying physical LAN medium. Timeout values can usually be small because acknowledgement transmission and reception on most LAN media is very fast. However, keep in mind that the underlying medium might have relatively low data rates (for example, 2400 baud). Unless the protocol stacks increase their internal timeout values when they are using a slow network, excessive and unneeded transmit retries will occur and adversely affect operation.

## Measuring Effective Network Performance

Protocol stacks can use two fields in the MLID's configuration table to measure the effective performance of a particular network: *TransportTime* and *LineSpeed* (see *Chapter 15: MLID Data Structures* for more detail regarding these fields).

### **TransportTime**

The *TransportTime* field specifies the time to transmit a 586 byte packet in milliseconds. This field is usually set to a value of 1 or 2 by all higher speed MLIDs. Lower speed LAN media must set this field to a higher value.

### **LineSpeed**

The *LineSpeed* field specifies the effective bits per second data rate of the underlying LAN medium. This field can be specified either in megabits per second or kilobits per second.

## Maximum Packet Size

Each physical LAN medium has a defined maximum packet size that it can transmit and receive. Protocol stacks must, therefore, configure themselves for the maximum amount of packet data that they can send and receive when using a particular board. The logical board's configuration table contains three maximum packet size fields: *MaxDataSize*, *BestDataSize* and *WorstDataSize*.

### **MaxDataSize**

*MaxDataSize* represents the absolute maximum packet size. The maximum packet size includes all low-level headers with the exception of the leaders and trailers managed by the hardware.

### **BestDataSize**

*BestDataSize* represents the maximum number of data bytes that the protocol stack can send and receive when it does not use certain low-level headers (for example, the source routing headers in Token-Ring).

### **WorstDataSize**

*WorstDataSize* represents the maximum number of data bytes the protocol stack can transmit and receive regardless of any low-level headers managed by the MLID. Protocol stacks should always use the *WorstDataSize* value when they determine the maximum data packet they can send and receive. The value of *WorstDataSize* includes the protocol stack's header information.

For example, if the *WorstDataSize* was set to 1500 bytes and a protocol stack appends a 16-byte header to all the data it transmits, the effective maximum amount of data that an application using that particular protocol stack can transmit and receive is 1484 (1500 – 16) bytes.

**Note** A prescan transmit stack can only increase the packet size if the stack fragments the packet. At no time should the size of the data being handed to the MLID exceed *WorstDataSize*.

## **Multicast Support**

A number of protocol stacks take advantage of multicast transmission, a LAN medium specific capability. Multicast transmission operates in a similar way to broadcast transmission: transmitted packets can be targeted to more than one node. The difference is that broadcast packets are received by all nodes on a network while multicast packets are received by a defined subset of all nodes. This allows the protocol broadcast information to only preempt the resources on the nodes that will actually receive the protocol stack's packets, significantly reducing the performance impact on the nodes that are not to receive the broadcast packets.

In order for a LAN adapter to become a member of a multicast group, the group's multicast address has to be enabled on the adapter so that any packets received by it will be passed to the host computer and not discarded at the hardware level. Protocol stacks determine whether an MLID supports multicast by examining the *MModeFlags* field in the MLID configuration table (see *Chapter 15: MLID Data Structures*).

Multicast support and the format of the multicast addresses is LAN medium dependent, and some LAN media do not support

any type of multicast capability. A protocol stack that utilizes multicasting must determine whether the MLID is using noncanonical or canonical addressing by examining the MLID configuration table entry *MModeFlags* (see *Chapter 15: MLID Data Structures*). (Canonical addressing is a “generic” form of addressing that is media independent. See *ODI Specification Supplement: Canonical and Noncanonical Addressing*.) If the MLID is using noncanonical addressing, the protocol stack must determine the LAN medium type of the underlying LAN adapter and use the appropriate multicast address. MLIDs have control functions that add and remove multicast addresses.

If a protocol stack does not know the format of the LAN medium’s multicast address, or the LAN medium does not support multicasts, the protocol stack should simply use real broadcasts (FF FF FF FF FF FF) instead of multicasts.

Protocol stacks that use multicast addresses should also allow the user to specify the multicast addresses that the protocol stack will use for a particular board. This capability is usually accomplished in the NET.CFG file by using a custom keyword (group addressing for example) underneath the protocol stack’s Main Section header. This allows the protocol stack to use correctly formatted multicast addresses for LAN medium other than the LAN medium for which the protocol’s multicast code was originally written.

## Receive Look Ahead

As part of the customization, your protocol stack should inform the underlying MLID about the amount of receive look ahead data it must have in order to properly process received packets.

This is done by calling the MLID’s control entry point function *SetLookAheadSize*.

## Bind to Logical Board(s)

One of the last things a protocol stack must do before it becomes fully operational is to bind to the predetermined board(s). Binding the protocol stack enables the LSL to route incoming packets destined for that protocol stack to its receive handler. You should note that a protocol stack can send packets without being bound to any board(s).

A protocol stack must be prepared to have its receive handler invoked after calling the *BindStack* support function.

If the protocol stack uses the prescan or default receive methods, this step is not done. Packet reception for these types of protocol stacks begins when the *RegisterPrescanRxChain* or *RegisterDefaultChain* commands are issued. Therefore, prescan and default protocol stacks must wait to register until they are ready to be fully operational.

## Final Initialization

At this point the protocol stack should be fully operational. Many protocol stacks will exit to DOS as a TSR (Terminate and Stay Resident program). It is recommended that your module display to the user the logical board(s) to which it is bound and servicing. The display should also include the MLID's short name and any other information that the user would find useful.

## Unhooking the Protocol Stack

The DOS ODI specification allows protocol stacks and MLIDs to unhook themselves from the LSL and purge any bindings between the MLID and the protocol stack. This dynamic capability allows DOS ODI modules to be loaded and unloaded as the need arises. Protocol suites that are compliant to the DOS ODI specification must be fully unloadable by the user. This chapter describes an algorithm for unloading a TSR (Terminate and Stay Resident) module.

### Unload Module Algorithm

Unloading a module usually entails reloading the transient module with a "U" command line option, signalling that the transient module should find its like module (the resident module) in the TSR chain and then take the actions necessary to unhook the resident module from the LSL and other modules. Finally, the unloading algorithm must free up the resident module's memory.

In order to unload a resident module, the transient module must perform the following general steps:

1. Find the last resident module.
2. Check that the modules are the same version.
3. Check that the module can unload safely.
4. Unhook the resident module.
5. Free the resident module.

#### Find the Last Resident Module

In order to unload a TSR module, there must be a method of finding and recognizing the module. You can find TSR modules

that interface directly with the LSL by using the appropriate LSL support routine. If the module does not interface directly with the LSL, you must use another method.

Before you can unload a module, you must obtain the module's PSP (Program Segment Prefix) value. This is usually easy because most TSRs use one segment which, typically, is equal to its PSP.

### Check that Modules are the Same Version

To prevent system corruption, the transient module should verify that the resident module is the same version (or at least looks enough like the transient module) so that it can unload safely. If the transient module has hooked interrupt vectors, the transient module should verify that these vectors have not been hooked by another TSR and that the offset value is the same as the transient module's ISR offset.

### Check that the Module Can Unload Safely

To prevent the user from corrupting the system by unloading a resident module that is being actively used by other modules, the transient module should verify that the resident TSR is the last module in the TSR chain. This assures that no other TSRs are hooked into the resident module. The sample code below illustrates the method we suggest you use. You should note that if the resident TSR is loaded above 640K bytes, the "last TSR" check code will not work with most LOAD HI applications. Therefore, if the resident module is above A000:0, skip the "last TSR" code and just allow the unload portion if all of the protocol's other safety checks pass.

The following contains sample code for unloading TSRs.

### Sample Code for Unhooking TSRs

```

mov     ax, ResModulePSPLocation    ;   Get installed module's segment value
cmp     ax, 0A000h                  ;   If loaded high don't do MCB checking
jae     SkipMCBChecking
;
;   Now make sure there are no other TSR's above the installed module.
;   Scan past all unused MCB's or MCB's in use by DOS. Then check the
;   in-use MCB to see if it is our program.
dec     ax                            ;   ES points at res modules MCB
mov     es, ax
CheckNextMCB:
mov     ax, es                        ;   ES:0 -> TSR to remove
add     ax, word ptr es:[0003h]      ;   -> next MCB in chain
inc     ax
mov     es, ax
cmp     word ptr es:[0001h], 0        ;   MCB in use?
je      CheckNextMCB

```

```
inc    ax                      ; Is it our environment block?
mov    dx, word ptr cs:[2Ch]
cmp    dx, ax
je     CheckNextMCB           ; Skip environment blocks
mov    cx, word ptr es:[0001h] ; Is this MCB owned by a shell?
mov    ds, cx
cmp    word ptr ds:[0016h], cx ; Shell's parent PSP -> self
je     CheckNextMCB           ; Allow shell MCB's to be between
; Found a true life MCB used by a TSR. The next MCB must be us or
; another TSR is loaded above the installed TSR.
mov    cx, cs                  ; Segments match (Is this us?)
cmp    cx, ax
jne    TSRAboveResident       ; Jump if error
; No one above resident module
SkipMCBChecking:
```

### **Unhook Resident Module**

Unhooking the resident module entails deregistering it from the LSL, restoring any hooked interrupt vectors, and unhooking the module from any other service it might be using.

### **Free Resident Module**

To free the memory a resident TSR is using, you must pass the resident module's PSP segment value in register ES and invoke the DOS *FreeMemory* function (#49h). After the resident module has been freed, the transient module simply exits to DOS without staying resident. Multi-segmented TSRs must make more than one call to the *FreeMemory* to free their memory. The sample code below illustrates this:

```
mov    es, ResModulePSPLocation ; DOS FreeMemory function
mov    ah, 49h
int    21h
```

□



*Chapter 5* **Protocol Stack Packet Reception**

## Protocol Stack Packet Receive Operation

When a protocol stack registers with the LSL, the protocol stack specifies a routine for the LSL to call when an MLID receives a packet destined for that protocol stack. This routine is the protocol stack's receive routine.

## Receive Routine Events

The events listed in Table 5.1 must occur during a protocol stack receive routine.

<b>Table 5.1 Protocol Stack Receive Routine</b>	
<b>Actor/Agent</b>	<b>Action</b>
MLID	1. When a packet is received, fills out a <i>LookAheadStruc</i> structure and calls the MLID LSL support routine <i>GetStackECB</i> to obtain from a protocol stack a receive buffer for the packet data. (For more information on <i>GetStackECB</i> , see <i>Chapter 11: LSL MLID Support Routines</i> .)
LSL	2. Determines which bound, prescan, or default protocol stacks will be receiving the packet. 3. Calls the protocol stack that is to receive the data and passes to the protocol stack a pointer to the <i>LookAheadStruc</i> structure describing the received packet.
Protocol Stack	4. Determines whether to receive the packet. 5. Builds an ECB describing a set of receive buffers into which the packet should be dispersed. 6. Signals to the LSL that this protocol stack will consume the packet and passes the ECB to the LSL, which passes the ECB to the MLID.
MLID	7. Copies the packet data into the provided data buffers. 8. Places the ECB into the LSL hold queue. 9. Calls <i>ServiceEvents</i> .
LSL	10. Dispatches the defined ESR (Event Service Routine), signaling that packet reception is complete.

## Receive Look Ahead

The receive method known as receive look ahead entails passing the beginning portion of the packet up to the protocol stack. In most cases, this allows the receive packet data to be dispersed directly into the application buffers. This is the optimal situation because the receive data only crosses the host's bus once.

## Receive Handler

Regardless of whether the protocol stack is bound, prescan or default, the protocol stack is passed look ahead data whenever

its receive handler is invoked. This data should be used to determine into which receive buffers (if any) the data should be placed. (Receive buffers can be fragmented.) If the protocol stack determines that it will consume the packet, it must build an ECB that describes the receive buffers and then return that ECB to the MLID. The MLID uses the ECB's description of the receive buffers to move the data from the network adapter into the described protocol receive buffers. When the MLID has completed the data move, it passes the ECB to the LSL for event completion. Event completion occurs when the MLID issues the LSL support command *ServiceEvents*. This calls the ECB's ESR and allows the protocol stack to process the packet.

## LookAheadStruc Structure

The *LookAheadStruc* structure that is given to the protocol stack's receive handler is defined below. The *LookAheadStruc* structure is valid only in the context of the receive handler. You should treat the *LookAheadStruc* structure as read-only, with the exception of the *LStartCopyOffset* field.

```

LookAheadStruc          struc
    LMediaHeaderPtr      dw    2    dup (?)
    LookAheadPtr         dw    2    dup (?)
    LookAheadLen         dw    ?
    LProtID              db    6    dup (?)
    LBoardNum            dw    0
    LDataSize            dw    ?
    LImmAddress          db    6    dup (?)
    LPacketAttrib        dw    0
    LDestType            dw    0
    LStartCopyOffset     dw    0
    LPriorityLevel        db    0
    LRESERVED            db    3    dup (?)
LookAheadStruc          ends

```

### *LMediaHeaderPtr*

This is a far pointer to a buffer containing the complete low-level media header. The protocol stack typically should not look at the low-level header information.

### *LookAheadPtr*

This is a far pointer to a buffer containing the start of the protocol's header (for example, IPX header). The *LMediaHeaderPtr* buffer is guaranteed to immediately precede the *LookAheadPtr* buffer (for example, the 802.3/802.2/IPX headers are contiguous).

### *LookAheadLen*

This contains the length of the *LookAheadPtr* buffer. This value is normally the MLID's currently configured look ahead size. Adapters that use shared RAM can set this field to the length of the data packet because the *LookAheadPtr* points at the entire packet located in shared RAM. If the received packet's data

length is less than the MLID's configured look ahead size, this field will be set to the actual packet data length. A protocol stack should verify that this field is at least the minimum length required for the protocol stack.

*LProtID*

This contains the Protocol ID value that was embedded in the low-level media header. This is the protocol's assigned Protocol ID value in the case of a bound stack.

*LBoardNum*

This contains the logical board number that received this packet. Remember that the logical board value specifies a LAN adapter and frame type combination.

*LDataSize*

The total number of data bytes in the received packet. If the MLID does not know the size, it will be set to -1, which typically only occurs on a pipeline adapter.

*LImmAddress*

This contains the address of the node that the packet was received from. Note, this address might not be the address of the node that originated the packet.

*LPacketAttrib*

Defined by the following bits:

PKT_CRC_ERR	equ 0001h	; CRC error / (FCS error)
PKT_ALIGN_ERR	equ 0002h	; CRC/frame alignment error
PKT_RUNT_ERR	equ 0004h	; runt packet
PKT_BIG_ERR	equ 0010h	; packet larger than media allowed
PKT_TYPE_ERR	equ 0020h	; packet for unsupported frame type
PKT_MALFORM_ERR	equ 0040h	; malformed packet
IMM_ADDR_MSB_FORM	equ 8000h	; immediate address is MSB format.

All bits in *LPacketAttrib* that are not defined are reserved.

*LDestType*

*LDestType* and *StkChnMask* have the following bit assignments:

DEST_MULTICAST	equ 0001h	
DEST_BROADCAST	equ 0002h	
DEST_REMOTE_UNICAST	equ 0004h	
DEST_REMOTE_MULTICAST	equ 0008h	
DEST_SOURCE_ROUTE	equ 0010h	; source route info
DEST_ERRORED	equ 0020h	; super exclusive bit
DEST_MAC_FRAME	equ 0040h	; exclusive bit
DEST_DIRECT	equ 0080h	
RX_NOT_8022	equ 0000h	; non-802.2 packet
RX_8022_TYPE1	equ 0100h	; 802.2 type 1 packet
RX_8022_TYPE2	equ 0200h	; 802.2 type 2 packet
RX_PRIORITY_FRAME	equ 0400h	; priority level is indicated
DEST_PROMISCUOUS	equ 0FFFFh	; all packets. (filter mask set by protocols to get all packets, including errors)

A protocol stack sets the bits for those types of packets that it wanted to see in its stack filter mask (see *ModifyStackFilter*).

MLIDs can set more than one bit if a packet fits into more than one category according to the following rules: (Tests are made to determine the type of packet and appropriate bits are set.)

- *RX\_PRIORITY\_FRAME* is set in conjunction with other destination type bits if the MAC indicates that the frame is a priority frame. This is only valid for those topologies that support a distinction in priority levels. When this bit is set, the *LPriority* field in the *LookAheadStruc* structure will hold the priority level of the frame. This bit is not set if the frame is received at the normal priority level of the topology.
- *RX\_8022\_TYPE1* or *RX\_8022\_TYPE2* is set in conjunction with other destination bits according to the 802.2 type designation.
- *DEST\_SOURCE\_ROUTE* is set in conjunction with other destination type bits if the packet has source routing information in the packet (for example, the RII bit is set). If a source routing module (for example, ROUTE.COM) is not loaded and the length of the source route field is greater than two bytes (packet is from a remote ring), all other bits will be cleared.
- *DEST\_MAC\_FRAME* is set if the packet is a MAC frame. All other bits will be cleared.
- *DEST\_ERRORED* is set if an error bit is set in the *LPacketAttrib* field. All other bits will be cleared. *DEST\_ERRORED* has the highest precedence.

#### *LStartCopyOffset*

This is set by the protocol stack to indicate where to start the copy process into the provided ECB buffers. Do not set this greater than *LookAheadDataLen*. It is an offset from the *LLookAheadPtr* location.

#### *LPriorityLevel*

If the *RX\_PRIORITY\_FRAME* bit is set, this field is set to the received packet's priority.

#### *LRESERVED*

This field is reserved.

**Note** In 802.2 frame types, the received packet's DSAP is returned in byte 5 of the *ProtocolIID* field; bytes 0 – 4 are set to 0.

## Receive Look Ahead Size

The amount of receive look ahead data needed by a protocol's receive handler is usually different for each type of protocol stack (prescan, bound, or default). The protocol stack can configure the amount of receive look ahead data the MLID provides by invoking *SET\_LOOK\_AHEAD\_SIZE* MLID control function as part of the protocol's initialization.

*SET\_LOOK\_AHEAD\_SIZE* informs the MLID that a protocol stack needs the specified number of packet data bytes to properly determine into which receive buffers a packet should be placed. The look ahead size value can be any value between 0 and 128 bytes inclusive. The requested size should not include any room for possible media headers, because the LAN driver will internally adjust the look ahead size value to include the LAN medium's worst case low-level media header size. If the requested size is larger than the current look ahead size, the MLID will use the new value. However, if the requested size is smaller than the current size, the MLID will not decrease the current size. The look ahead size is set to a default size of 18.

## Protocol Stack Packet Reception Methods

The ODI specification defines three types of protocol stacks:

- bound
- prescan
- default

The following describes each type of protocol stack.

### Bound Protocol Stack

Bound protocol stacks receive packets with the appropriate Protocol ID (PID) in the *LookAheadStruc* structure's *LProtID* field. The PID is obtained from the low-level frame header. A bound protocol stack can choose to consume or reject a packet. If the protocol stack rejects the packet and no default protocol stack exists for this board, the packet is discarded from the system.

### Prescan Protocol Stack

Prescan protocol stacks use the look ahead method (as do bound and default stacks) to look at all packets received by a particular logical board (adapter and frame type combination). The protocol stack can consume select packets and allow others to pass onto other prescan stacks or to the appropriate bound or default protocol stack.

### Default Protocol Stack

Default protocol stacks receive packets not consumed by the prescan and bound protocol stacks. A default protocol stack can choose to consume or reject a packet. If the packet is rejected, it is shown to other default stacks in the chains. If no stack consumes the packet, it is discarded from the system.

We discourage you from using the prescan or default method of packet reception. This is because a protocol stack parses the packet header to determine if the packet is the correct type. Therefore, a protocol stack might receive a packet that passes the protocol stack's acceptance tests, but in reality is not the correct type. This could cause unpredictable results in the network station.

The prescan and default reception methods should only be used for specialized protocol stacks that must receive packets having a large range of Protocol IDs. For example, the 802.2 protocol stack must receive packets with any destination SAP. Protocol stacks that provide a Data-Link Layer interface to the network layer protocol stacks are candidates for using prescan or default receive methods.

## StackChainStruc Structure

```
StackChainStruc  struc
    StkChnLink      dd  0      ; Link Field
    StkChnBoardNum  dw -1      ; Logical Board Number
    StkChnPositReq  dw  STACK_REQ_DEPENDS  ; Chain
                                     Position Requested
    StkChnHandler   dd  ?      ; Stack's Tx or Rx Handler
    StkChnControl   dd  ?      ; Stack's Control Entry Point
    StkChnID        dw  0      ; Stack's ID Number
    StkChnMask      dw  83h    ; Default (broadcast, multicast, and
                                     direct)
    StkChnReserved  dd  0      ; Reserved
StackChainStruc  ends          ; (Size 24 bytes)
```

*StkChnMask* bits are the same as those defined for *LDestType* in the *LookAheadStruc* structure.

The following are the *StkChnPositReq* equates:

```
STACK_REQ_FIRST      equ 0    ; must be first stack in chain
STACK_REQ_NEXT_FIRS  equ 1    ; load at next available position
                                     from front of chain
STACK_REQ_DEPENDS    equ 2    ; chain position depends on load
                                     order
STACK_REQ_NEXT_LAST  equ 3    ; load at next available position
                                     from end
STACK_REQ_LAST       equ 4    ; must be last stack in chain
```

These bits achieve the following prescan chain picture:

LSL Tx ->0 ->1a->1b ->2c->2b->2a ->3b->3a ->4 -> MLID (NIC)

LSL Rx <-4 <-3a<-3b <-2c<-2b-<2a <-1b<-1a <-0 <- MLID (NIC)

This picture depicts the flow of the packet from the LSL transmit call to the MLID, and back. "a" would be the first "x"

position requested protocol stack. A protocol stack that is interested in both send and receive, a compression/decompression stack for example, might need to register as a transmit stack as position "1" (STACK\_REQ\_NEXT\_FIRST). The stack must then register as a receive stack position "3" (STACK\_REQ\_NEXT\_LAST) in order to ensure that the protocol stack is in the same order relative to other stacks in the chain. That way the stack would be assured that the algorithm applied to the packet would be able to be undone by the receiving station, and visa versa. If a protocol stack chooses to register for a position of "2" (STACK\_REQ\_DEPENDS), the LSL will do the swapping to ensure that the packet is seen in the proper order. Using the load order dependant position "2" (STACK\_REQ\_DEPENDS) provides the greatest flexibility. It is the stack's responsibility to request the proper position when registering for both transmit and receive protocol chains.

Default stacks have the following chain picture:

(Last Stack) 4 <-3a<-3b <-2c<-2b<-2a <-1b<-1a <-0 <- LSL RX (Last Bound stack)

## Protocol Receive Handler

**Description** The protocol receive handler is invoked when a packet has been received and the LSL has determined that the packet is intended for the protocol stack. This definition applies to bound, prescan, and default protocol stacks.

**Entry State**

*DS:DI*  
has a pointer to the *LookAheadStruc* structure.

*ES:SI*  
has a pointer to a *StackChainStruc* structure if the stack is a prescan or default stack.

Interrupts  
are disabled.

**Note** CLD is cleared.

**Return State**

*AX*  
has a completion code.

*ES:SI*  
has a pointer to an ECB if *AX* equals *LSL\_SUCCESSFUL* (0000h).

Flags  
Z flag is set according to *AX*.

Interrupts  
are disabled.

*DS, DI, BP, SS, SP*  
are preserved.

**Note** CLD is cleared.

**Completion Codes (AX)**

*LSL\_SUCCESSFUL* (0000h)  
The protocol stack will consume the packet (*ES:SI* valid).

*LSLERR\_OUT\_OF\_RESOURCES* (8001h)  
The protocol stack will not receive the packet.

**Remarks**

The protocol stack examines the look ahead data as described by the *LookAheadStruc* structure and returns an ECB when appropriate.

The *LookAheadStruc* structure and its fields are only valid in the context of this function. This routine must complete quickly, and interrupts must remain disabled. LAN driver functions should not be invoked inside this function (for example, *SendPacket* should not be invoked inside this function).

The ECB must have the following fields and descriptors set before it returns: *EventServiceRoutine*, *FragmentCount*, and fragment descriptors. You can specify more than one fragment descriptor. The *FragmentCount* field cannot be set to 0 and should not exceed 16.

If the protocol stack also requires the ECB *BoardNumber* field to be filled in, the protocol stack should fill it in with the board number supplied in the *LookAheadStruc* structure.

If an ECB is returned from this function, the MLID will call the ESR at a later time, signaling that the packet data has been transferred to the described receive buffers either successfully or with an error.

The protocol receive handler can be invoked multiple times before a previous ECB's ESR will be called. Therefore, the protocol stack should allocate and maintain multiple ECBs.

The following fields in the ECB structure must be set by the protocol stack prior to providing the ECB to the MLID.

- ESR
- FragCount
- Frag descriptors
- BoardNumber (required only if you want board numbers)

**Note** Prescan and default receive protocol stacks should set the *StackID* field if they provide an ECB.

## Protocol Receive Complete Handler

**Description** The LSL invokes this Event Service Routine (ESR) function after the MLID has dispersed the receive packet's data (either with or without error) into the previously provided ECB's data buffers and has placed the ECB on the LSL's holding queue. When this function is called, the LSL transfers ownership of the ECB and its associated data buffers back to the protocol stack.

**Entry State** *ES:SI*  
has a pointer to an ECB.

Interrupts  
are disabled.

**Note** CLD is cleared.

**Return State** Interrupts  
are disabled.

*DS, BP, SS, SP*  
are preserved.

**Note** CLD is cleared.

**Remarks** The following table illustrates the event sequence of the receive complete handler.

**Table 5.2 Receive Complete Handler**

<b>Actor/ Agent</b>	<b>Action</b>
MLID	1. Disperses the receive packet data into the receive buffers supplied by the protocol stack. 2. Gives the ECB to the LSL for temporary holding. 3. Invokes the LSL's service events routine, after the MLID has finished servicing the network adapter.
LSL's Service Events Routine	4. Calls each of the previously queued ECB's Event Service Routines (ESRs). (The protocol stack set the address of the ESR before it returned an ECB to the MLID.) 5. Transfers ownership of the ECB and its associated data buffers back to the protocol stack when the ECB's ESR is called.

When this function is called, the following ECB fields are set:

- PrevLink
- Status

- ESR
- StackID
- ProtID
- ImmediateAddress
- DriverWS
- DataLength
- FragCount
- fragment descriptions

**Note** If the MLID is using canonical addressing, the address in the *ImmediateAddress* field will be in canonical form.

The ECB *Status* field can be set to one of the following values:

*LSL\_SUCCESSFUL* (0000h)

Packet was received successfully.

*LSLERR\_RX\_OVERFLOW* (8006h)

The supplied data buffers were not large enough to contain the entire packet but were filled with valid packet data.

*LSLERR\_CANCELLED* (8007h)

The MLID could not complete the receive operation. The data buffers do not contain valid data.

The ECB *StackID* field will be set to the Stack ID of the protocol stack receiving this packet. The LSL assigned this value when the protocol stack registered.

The *LDestType* field is stored in the *DriverWS* field and will use the following bit assignments:

<i>DEST_MULTICAST</i>	equ 0001h	
<i>DEST_BROADCAST</i>	equ 0002h	
<i>DEST_REMOTE_UNICAST</i>	equ 0004h	
<i>DEST_REMOTE_MULTICAST</i>	equ 0008h	
<i>DEST_SOURCE_ROUTE</i>	equ 0010h	; source route info
<i>DEST_ERRORED</i>	equ 0020h	; super exclusive bit
<i>DEST_MAC_FRAME</i>	equ 0040h	; exclusive bit
<i>DEST_DIRECT</i>	equ 0080h	
<i>RX_NOT_8022</i>	equ 0000h	; non-802.2 packet
<i>RX_8022_TYPE1</i>	equ 0100h	; 802.2 type 1 packet
<i>RX_8022_TYPE2</i>	equ 0200h	; 802.2 type 2 packet
<i>RX_PRIORITY_FRAME</i>	equ 0400h	; priority level is indicated
<i>DEST_PROMISCUOUS</i>	equ 0FFFFh	; all packets. (filter mask set by protocols to get all packets, including errors)

If the MLID configuration table is v1.12 or greater, the MLID supports 802.2 Type II (see *Chapter 15: MLID Data Structures*).

In most cases, the protocol receive complete handler is invoked from the context of a hardware interrupt. The LAN driver has

finished servicing the network adapter and is ready to restore the processor registers and exit from its interrupt handler. This routine might enable interrupts and process for an extended period of time. However, the routine must switch to an internal stack and guard against being reentered because the network hardware is fully functional at this point. MLID control routines should not be invoked from this routine because they can be called only at process time. However, the protocol stack may freely make requests to the LSL (for example, by using the protocol's *SendPacket* function).

## ECB Resubmit Procedures

Prescan and default protocol stacks must be able to place ECBs back into the chain at the next point from where the ECB was provided to consume the packet. This allows a protocol stack that is spell checking, for example, to correct the spelling in a packet and then send the corrected ECB onto the rest of the system. Another example is a prescan protocol stack that was converting packets with EBCDIC characters to ASCII characters. In both cases, the protocol stack needs to operate on the ECB and then resubmit the packet to the LSL.

This resubmission is done with the appropriate *ResubmitXXX* through the protocol support entry point. Prescan transmit stacks use *ResubmitPrescanTx*; prescan receive stacks use *ResubmitPrescanRx*; default stacks use *ResubmitDefault*. Calling these functions should only be done with data received through the protocol's receive/transmit handler.

Prescan transmit protocol stacks send packets that originate from inside a protocol or from an application to the LSL as part of a *SendPacket* function call. This packet traverses the prescan transmit chain and is eventually presented to the prescan transmit protocol as a transmit.

For proper syntax, see the respective function descriptions in *Chapter 10: LSL Protocol Stack Support Routines*.

□



*Chapter 6* **Protocol Stack Packet Transmission**

## Overview

In the ODI specification, packet transmission is an asynchronous operation that entails building an Event Control Block (ECB) and calling the *SendPacket* protocol support routine (see *Chapter 10: LSL Protocol Stack Support Routines*). Packets sent through the LSL are connectionless and, if the conditions warrant, are neither guaranteed to reach their destination, nor to be placed onto the LAN medium. Protocol stacks typically do not need to use checksums because the underlying MLID and LAN adapters guarantee a high degree of data integrity; however, your protocol stack can use checksums if you desire.

**Note** Some protocol stacks must provide guaranteed packet delivery to the upper layers. If this is the case, your protocol stack must contain the necessary timeouts, retries, and packet acknowledgments to realize a guaranteed delivery system.

## Send Routine Events

The events listed in the following table must occur during a protocol stack send routine:

<b>Table 6.1 Protocol Stack Send Routine</b>	
<b>Actor/ Agent</b>	<b>Event</b>
Protocol Stack	1. Hands the ECB to the LSL for transmission.
LSL	2. Passes the ECB to any prescan transmit stack registered with the ECB's board number. If consumed, the prescan transmit stack must resubmit the ECB or cancel it later. If not consumed or if there is no prescan transmit stack, the LSL calls the underlying MLID transmit handler with a pointer to the ECB.
MLID	3. Passes ownership of the ECB and its associated packet data buffers to the MLID. 4. Transmits the packet. 5. Passes ownership of the ECB and its associated packet data buffers to the LSL, regardless of whether the packet transmission was completed successfully or with an error. 6. Notifies the transmit monitor of packet transmission.
LSL	7. Calls the Event Service Routine specified in the ECB.

**Note** The ECB and its associated data buffers must not be modified until ownership is returned to the protocol stack.

## Starting the Packet Send

A protocol stack can usually transmit packets at any time.



**Caution** A protocol stack should not poll for a transmit complete inside an interrupt handler (for example, IRQ 0). Polling for a transmit complete inside of an interrupt handler can create a dead-lock. A protocol stack should also not issue transmit requests inside its packet look ahead receive handler routine.

**Note** For information about priority packet transmission, see “Priority Packet Support” in *Chapter 2: Overview of Protocol Stacks*.

## Supporting Multiple Outstanding Transmit Requests

The underlying MLIDs generally support multiple outstanding transmit requests from protocol stacks. While the adapter transmits one packet onto the LAN medium, the MLID loads the next transmit packet’s data onto the LAN adapter. The number of transmits an MLID can give an adapter before the MLID must queue the ECBs varies, because this number is MLID-dependent.

When the protocol stack must transmit bursts of packets, it will achieve its best performance by passing multiple transmit requests to an underlying MLID. In theory, an MLID can handle any number of outstanding transmit requests. However, three outstanding requests have been proven to be adequate. An MLID that handles more than three active transmit requests generally does not have higher throughput (three active transmit requests saturate most LAN adapters). A protocol stack should be able to have at least three transmits outstanding on a particular board.

## Sending the Packet

To send a packet, the protocol stack must provide data buffers and an Event Control Block (ECB) describing the data to be sent. The protocol stack can specify from 1 to 16 data buffers per transmit request. The underlying MLID will then combine the buffers to form a single data packet.

## Event Control Blocks

An Event Control Block (ECB) is a general purpose request control block used for transmit and receive events in the ODI

specification. The ECB *ProtocolWorkspace* field can be used for any purpose by the protocol stack because the *ProtocolWorkspace* field is not modified by the LSL or the MLIDs. (See *Appendix A: Event Control Blocks (ECBs)* for a more detailed discussion of ECBs.) However, the protocol stack cannot count on any field or value in the ECB while other parts of the system own the ECB.

You must set the ECB fields and descriptors listed below before the protocol stack gives the ECB to the LSL for transmission:

- `EventServiceRoutine`
- `StackID`
- `BoardNumber`
- `ProtocolID`
- `ImmediateAddress`
- `DataLength`
- `FragmentCount`
- `Fragment Descriptors`

These fields are treated as read-only by the LSL and the MLIDs. Therefore, each field does not have to be reinitialized after a transmit operation unless that field's value must be changed.

### **EventServiceRoutine**

The *EventServiceRoutine* field contains a far pointer to a routine the LSL calls when the underlying MLID has finished with the ECB and its data buffers. (See the "Transmit Complete" section at the end of this chapter.)

### **StackID**

The *StackID* field should be initialized with the protocol's assigned stack ID. Raw sends and priority packets are also indicated in this field.

### **Raw Sends**

The ODI specification specifies an optional capability (raw send) in MLIDs that allows protocol stacks to specify the complete low-level header when sending a packet. Because Raw sends force a protocol stack to be LAN medium and frame type aware, protocol stacks do not generally use raw sends unless absolutely necessary.

Because a raw send is an optional capability, some MLIDs do not support it. To determine whether a particular board supports raw sends, the protocol stack should check the

*RawSendBit* bit (bit 6 [0040h]) in the MLID configuration table *ModeFlags* field. If this bit is set, the MLID supports raw sends.

A protocol stack signals a raw send to the MLID by placing 0FFFFh, instead of its stack ID, in the *StackID* field. The underlying MLID checks this field for 0FFFFh. If this value is 0FFFFh, the MLID will skip over the code to build the low-level header. The first fragment of the ECB must then contain the entire low-level header information.

The first data fragment must contain the complete MAC header, including the source address, for the media in use. However, in some cases this address will not be used; some adapters automatically insert the source node address in the low-level header.

The protocol stack must be completely aware of frame characteristics. Usually, however, minimum packet length padding and evenization are handled by the MLID.

### **BoardNumber**

The *BoardNumber* field specifies on which logical board the packet should be transmitted. The board number specifies which physical adapter and which low-level frame format will be used.

### **ProtocolID**

The *ProtocolID* field specifies which Protocol ID value will be embedded into the frame header. This value stamps the packet as a particular protocol type (for example, IPX, TCP/IP, etc.).

For example, the Ethernet 802.2 frame ECB's *ProtocolID* field contains the Destination Service Access Point (DSAP). The Source SAP (SSAP) is set equal to the Destination SAP (DSAP or Protocol ID) when the MLID builds the frame header. The MLID will also set the 802.2 control byte equal to 03h (UI).

To allow a protocol stack to specify the complete 802.2 header (for example, DSAP, SSAP, Control 0, Control 1), MLIDs that support the 802.2 frame allow a special flag in the transmit ECB *ProtocolID* field. When this flag is present, the MLID uses the specified 802.2 header instead of setting SSAP equal to DSAP and Control equal to 03h (the usual method).

If an explicit 802.2 header needs to be specified, set the *ProtocolID* field to the following values:

**Byte 0 x** This byte is normally zero. However, if a non-zero number is specified, the MLID will look for the explicit header information. x = a zero-based number of bytes in the explicit number (for example, 00h signifies DSAP, 02h signifies DSAP, SSAP and Control 0—802.2 Type I header, and 03h signifies DSAP, SSAP, Control 0, Control 1—802.2 Type II header).

**Bytes 1 through 5** These bytes are set by the protocol stack for an explicit 802.2 header. Unused bytes should be set to 0.

**Note** If the MLID configuration table version is v1.11, the MLID will support only 802.2 Type 1 and the additional control byte for the type number could be present in some 802.2 packets. A protocol that uses this extra control byte should include this byte in the first data fragment of the send ECB. (See *ODI Specification Supplement: Frame Types and Protocol IDs* for more information on 802.2 Type II Frame Header Support.) If the MLID configuration table version is v1.12, or greater, the MLID supports the flag value in Byte 0.

### **ImmediateAddress**

The *ImmediateAddress* field contains the destination address that specifies to which node on the local network the packet should be sent. This can be a direct, multicast, or broadcast address. If your protocol stack must receive its own sends, it must emulate loopback capabilities. The address is expected to be in the form specified by the MLID *MModeFlags* field in the MLID configuration table. Bits 14 and 15 indicate the appropriate format. (See *Chapter 15: MLID Data Structures and ODI Specification Supplement: Canonical and Noncanonical Addressing* for more information.)

**Note** The address FF FF FF FF FF FF always indicates a broadcast packet. (All adapters on the physical network will receive the packet.)

### **DataLength**

The *DataLength* field holds the total length of all fragment descriptor length fields.

### **FragmentCount**

The *FragmentCount* field specifies the number of fragment descriptor data structures that follow the *FragmentCount* field. This field must contain a value greater than 0 and less than or equal to 16 ( $0 < \textit{FragmentCount} \leq 16$ ).



### **Important**

*FragmentCount* must never equal 0.

### Fragment Descriptors

Each fragment descriptor contains the location and length of a contiguous section of RAM memory (segment:offset). The protocol stack can specify a maximum of 16 fragment descriptors. The MLID will combine the fragments together to form one contiguous packet.

**Note** The length field of a fragment descriptor can be 0.

A frame containing only an 802.2 Type II header can be transmitted by setting the length fields of the fragment descriptors for the ECB containing the transmit information to 0. The *FragmentCount* field must be equal to at least 01h. The ECB *ProtocolID* field should contain the entire explicit 802.2 Type II header.

### Transmit Complete

The events in the following table occur to complete transmitting a packet.

<b>Table 6.2 Completing Packet Transmission</b>	
<b>Actor/ Agent</b>	<b>Action</b>
Protocol Stack	1. Gives the ECB to the MLID (through the LSL).
MLID	2. Transmits the ECB. 3. Returns the ECB to the LSL.
LSL	4. Places the ECB into a temporary event queue.
MLID	5. Calls the LSL <i>ServiceEvents</i> routine after the MLID has finished servicing the hardware.
LSL's Service Events Routine	6. Removes each ECB from the queue in turn. 7. Calls the ESR defined in the ECB Event Service Routine (ESR). In the case of a transmit complete, the ESR will be the protocol stack's transmit complete handler.

**Note** The MLID can invoke the protocol stack's transmit complete handler before the call to *SendPacket* returns.

## Protocol Transmit Complete Handler

**Description** The protocol transmit complete handler is called when a previous transmit request has completed successfully or with an error.

---

**Entry State** *ES:SI*  
has a pointer to the completed ECB.  
  
Interrupts  
are disabled.

**Note** CLD is cleared.

**Return State** Interrupts  
state is preserved.  
  
*DS, BP, SS, SP*  
are preserved.

**Note** CLD is cleared.

---

**Remarks** When this routine is invoked, the LSL returns ownership of the ECB and its data buffers to the protocol stack.

A protocol must not use ECBs that it does not own.

This routine must complete quickly because it is usually invoked from an Interrupt Service Routine. Transmit requests may be issued from this routine, but the protocol stack must switch to its own stack.

The ECB *Status* field is set to one of the following:

*LSL\_SUCCESSFUL* (0000h)

The MLID determined that the transmit was successful. Because the transmit was connectionless, this completion code does not mean that the destination received the packet.

*LSLERR\_NO\_SUCH\_DRIVER* (800Bh)

The LAN adapter specified in the ECB *BoardNumber* field cannot be found. This usually means that the MLID has been removed from memory.

*LSLERR\_PACKET\_ERRORED* (800Ch)

The MLID was unable to transmit the packet. The packet was probably too big.

## Prescan Transmit Protocol Stack Handler

**Description** Ownership of the ECB and its associated data buffer is passed to the protocol stack when the LSL calls the protocol's transmit handler. The protocol's transmit handler can be called either at process or interrupt time.

**Entry State** *ES:SI*  
pointer to transmit ECB.

Interrupts  
are disabled.

**Return State** *AX*  
Zero the protocol stack consumed the packet.  
Non-zero the protocol stack returns an ECB the LSL should route.

*ES:SI*  
has a pointer to an ECB the LSL should route if *AX* is non-zero.

Interrupts  
are disabled.

Flags  
are set according to *AX*.

*DS, BP, SS, SP*  
are preserved.

**Remarks** This routine can enable interrupts and can process for an extended period of time if the routine is switched to an internal stack and has guarded against reentry (the network hardware can be fully functional at this point, hence packet transmission ordering must be maintained), or it can treat this as a run-to-completion event (be aware that this can degrade performance). MLID control routines must not be invoked from this routine because they can only be called at process time. However, the protocol stack can freely make requests to the LSL (such as *GetECB*, to obtain another ECB buffer). If the protocol stack consumes the ECB, after it has finished with it, it should return the ECB to the LSL using the *HoldEvent* support function.

If the protocol stack processes the transmit ECB by queuing it and servicing the ECB at process time, it can resubmit the ECB for further processing by other transmit prescan protocol stacks and for eventual transmission by using the *ReSubmitPreScanTx* function.

Transmitting prescan protocol stacks should treat an ECB handed to it, whose data it is going to modify, as read-only and should create a copy of the ECB and process the copy, installing its Event Service Routine (ESR) in the ECB etc. The reasons for this are due to the originating protocol stack (bound stacks for example) can manipulate data in the original ECB when it's ESR is called and if the prescan stack manipulates the data—for example, compression, the data will be incomprehensible to the original stack. When the prescan stack's ESR is called, it in turn should call the ESR in the original ECB with ES:SI pointing to the original ECB.

**Note** Data transmitted by prescan stacks are still limited by the transmitting MLID configuration table's *WorstDataSize* (offset 2Ch) value. Also calling the LSL function *SendPacket* from a transmit prescan stack can cause the prescan stack's protocol transmit handler to be called from itself.

Prescan stacks call resubmit for transmitting data originated external to the prescan stack. Data originating from the prescan stack should be sent by calling the LSL *SendPacket* function. The LSL will handle calling all the prescan transmit handlers for data being transmitted on a board with prescan stacks registered.

□



*Chapter 7* **Protocol Stack Control Routines**

## Overview

When a protocol stack registers with the LSL, one of the parameters it passes is a pointer to the protocol stacks control entry point. Applications and other protocol stacks can obtain this entry point from the LSL (see *Chapter 10: LSL Protocol Stack Support Routines, GetProtocolControlEntry*) and then invoke it to obtain the protocol's configuration.

The following is an alphabetical list of the protocol stack control routines along with their entry point and function number.

<b>Descriptive Name</b>	<b>Function Name</b>	<b>Funct. No.</b>
BindToMLID	BIND_TO_MLID	2
GetProtocolStackConfiguration	GET_STACK_CONFIGURATION	0
GetProtocolStackStatistics	GET_STACK_STATISTICS	1
MLIDDeregistered	INFORM_MLID_DEREGISTERED	4
ProtocolPromiscuousChange	INFORM_PROMISCUOUS_CHANGE	5
ProtocolManagement	PROTOCOL_MANAGEMENT	8
UnbindFromMLID	UNBIND_FROM_MLID	3
Reserved	RESERVED	6
Reserved	RESERVED	7

The following is a function number ordered list of the protocol stack control routines along with their entry point and function number.

<b>Descriptive Name</b>	<b>Function Name</b>	<b>Funct. No.</b>
GetProtocolStackConfiguration	GET_STACK_CONFIGURATION	0
GetProtocolStackStatistics	GET_STACK_STATISTICS	1
Bind to MLID	BIND_TO_MLID	2
UnbindFromMLID	UNBIND_FROM_MLID	3
MLIDDeregistered	INFORM_MLID_DEREGISTERED	4
ProtocolPromiscuousChange	INFORM_PROMISCUOUS_CHANGE	5
Reserved	RESERVED	6
Reserved	RESERVED	7
ProtocolManagement	PROTOCOL_MANAGEMENT	8

## BindToMLID

<b>Description</b>	Informs the protocol stack that it should bind to a specific board.
<hr/>	
<b>Entry State</b>	<p><i>BX</i> is equal to <i>BIND_TO_MLID</i> (0002h).</p> <p><i>CX</i> has the board number to which the protocol stack should bind.</p> <p><i>ES:SI</i> has a pointer to an optional parameter string that is protocol implementation-dependent.</p> <p>Interrupts are enabled.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p>Flags Z flag set according to <i>AX</i>.</p> <p>Interrupts are enabled.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The protocol stack is successfully bound.</p> <p><i>LSLERR_BAD_COMMAND</i> (8008h) The protocol does not support this function.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h) The board number was invalid or the parameter pointed to by <i>ES:SI</i> was invalid.</p> <p><i>LSLERR_DUPLICATE_ENTRY</i> (8009h) The protocol is already bound to a board.</p> <p>Other protocol implementation-dependent codes</p>
<hr/>	
<b>Remarks</b>	<p>Because most protocol stacks bind to a board or boards before staying resident, it is not necessary to implement this function. However, the protocol stack can implement this function if the protocol stack must have an external utility invoke the binding process after the protocol stack has stayed resident or to provide dynamic linking. The protocol stack must return <i>LSLERR_BAD_COMMAND</i> (8008h) if it chooses not to support this function after staying resident.</p>

## GetProtocolStackConfiguration

**Description** Returns a pointer to the protocol stack configuration table.

---

**Entry State**

*BX*

is equal to *GET\_STACK\_CONFIGURATION* (0000h).

Interrupts

are enabled.

**Return State**

*AX*

is always equal to *LSL\_SUCCESSFUL* (0000h).

*ES:SI*

has a pointer to the protocol stack configuration table.

Flags

Z flag is set according to *AX*.

Interrupts

are enabled.

*DS, BP, SS, SP*

are preserved.

---

**Remarks**

All protocol stacks must support this function.

**See Also**

See *Chapter 3: Protocol Stack Data Structures* for the format of the protocol stack configuration table.

---

## GetProtocolStackStatistics

**Description** Returns a pointer to the protocol stack internal statistics table.

---

**Entry State**

*BX*

is equal to *GET\_STACK\_STATISTICS* (0001h).

Interrupts

are enabled.

**Return State**

*AX*

is always equal to *LSL\_SUCCESSFUL* (0000h).

*ES:SI*

has a pointer to the protocol stack statistics table.

Flags

Z flag is set according to *AX*.

Interrupts

are enabled.

*DS, BP, SS, SP*

are preserved.

---

**Remarks**

All protocol stacks must support this function.

**See Also**

See *Chapter 3: Protocol Stack Data Structures* for the format of the protocol stack statistics table.

## MLIDDeregistered

**Description** Informs the protocol stacks that the specified board is no longer available.

---

**Entry State**

*BX*  
is equal to *INFORM\_MLID\_DEREGISTERED* (0004h).

*CX*  
has a board number that has deregistered from the LSL.

Interrupts  
are enabled.

**Return State**

*AX*  
has a completion code.

Interrupts  
are enabled.

*DS, BP, SS, SP*  
are preserved.

**Completion Codes (AX)**

*LSL\_SUCCESSFUL* (0000h)  
The call was acknowledged and might have been acted upon.

*LSLERR\_BAD\_COMMAND* (8008h)  
The protocol does not support this function.

---

**Remarks**

The LSL invokes *MLIDDeregistered* whenever the logical board that a protocol stack is using has deregistered. The protocol stack can use this information in any way it chooses and can even discard it. However, the specified board will not be available for packet transmission or reception. Also, if the protocol stack discards the information, the stack must be able to handle the call and return with *AX* equal to *LSLERR\_BAD\_COMMAND* (8008h).

**See Also** *DeregisterMLID* in Chapter 11: *LSL MLID Support Routines*.

## ProtocolManagement

<b>Description</b>	Provides a generic way of allowing protocol dependent functions to be defined.
<hr/>	
<b>Entry State</b>	<p><i>BX</i> is equal to <i>PROTOCOL_MANAGEMENT</i> (8).</p> <p><i>ES:SI</i> is a pointer to the management ECB.</p> <p>Interrupts are disabled.</p>
<b>Return State</b>	<p><i>AX</i> has completion code.</p> <p><i>ES:SI</i> is a pointer to the management ECB.</p> <p>Flags are set according to <i>AX</i>.</p> <p>Interrupts are disabled but might have been temporarily enabled by the protocol stack.</p> <p><i>DS, ES, BP, SI, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The requested operation was successful. The ECB is returned to the caller.</p> <p><i>LSL_PENDING_SUCCESS</i> (0001h) The requested operation was successfully started but will complete asynchronously. The ECB is not returned. The ESR will be called after the operation completes.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h) The first byte of the ECB <i>ProtocolID</i> field is not valid. The first byte must be greater than 41h (A) and less than 7Eh (~) inclusive.</p> <p><i>LSLERR_BAD_COMMAND</i> (8008h) Protocol management support is not provided.</p> <p><i>LSLERR_NO_SUCH_HANDLER</i> (800Ah) The Protocol ID value is not supported.</p>
<hr/>	
<b>Remarks</b>	This control function is provided to allow the protocol a generic interface to protocol dependant management functions. The implementation of this function is optional. If this call is not implemented, a call to this function must return <i>LSLERR_BAD_COMMAND</i> (8008h).

The management ECB is of the form of an ECB, but all fields below the *ProtocolID* field can be redefined by the protocol.

The *ProtocolID* field is defined as a 6-byte string that uniquely identifies the protocol. The first character of the string must be greater than or equal to 41h ("A") and less than or equal to 7Eh ("~"). The remaining characters are defined by the protocol. If the first character is not greater than or equal to 41h and less than or equal to 7Eh, the protocol should return with the completion code *LSLERR\_BAD\_PARAMETER* (8002h).

If the protocol does not recognize the value in the *ProtocolID* field, the protocol returns a completion code of *LSLERR\_NO\_SUCH\_HANDLER* (800Ah).

If the protocol must respond asynchronously to the management request, it should queue the ECB internally and return a status of *LSL\_PENDING\_SUCCESS* (0001h). When the queued request is complete, the protocol should place the ECB on the LSL hold event queue by calling *HoldEvent*. The LSL will then process the ECB during the next call to service events.

**See Also**

*DriverManagement*

## ProtocolPromiscuousChange

<b>Description</b>	Called before a MLID changes the promiscuous state.
<b>Entry State</b>	<hr/> <i>BX</i> is equal to <i>INFORM_PROMISCUOUS_CHANGE</i> (5).  <i>CX</i> has the board number.  <i>SI</i> has the promiscuous state.  PROM_OFF (0000h) = Promiscuous mode off PROM_MAC (0001h) = All MAC frames to be received. PROM_NORMAL (0002h) = All non-MAC frames to be received. PROM_SMT (0004h) = All SMT frames to be received.  Interrupts are disabled.
<b>Return State</b>	 <i>AX</i> contains a completion code.  Interrupts are unchanged.  <i>DS, BP, SS, SP</i> are preserved.
<b>Completion Codes (AX)</b>	 <i>LSL_SUCCESSFUL</i> (0000h) The requested operation was completed successfully.  <i>LSLERR_BAD_COMMAND</i> (8008h) This function is not supported.
<b>Remarks</b>	<hr/> <p>A protocol stack implements this control command if it expects that the MLID is only passing up qualified packets and needs to adjust its filtering methods if promiscuous mode has been enabled by another protocol stack. This command will also be called when an MLID is turning promiscuous mode off. The MLID is responsible for notifying all of the logical boards using a particular physical board of each change.</p> <p>The protocol will only receive the packets that have not been filtered out by the currently active filter for the protocol.</p>

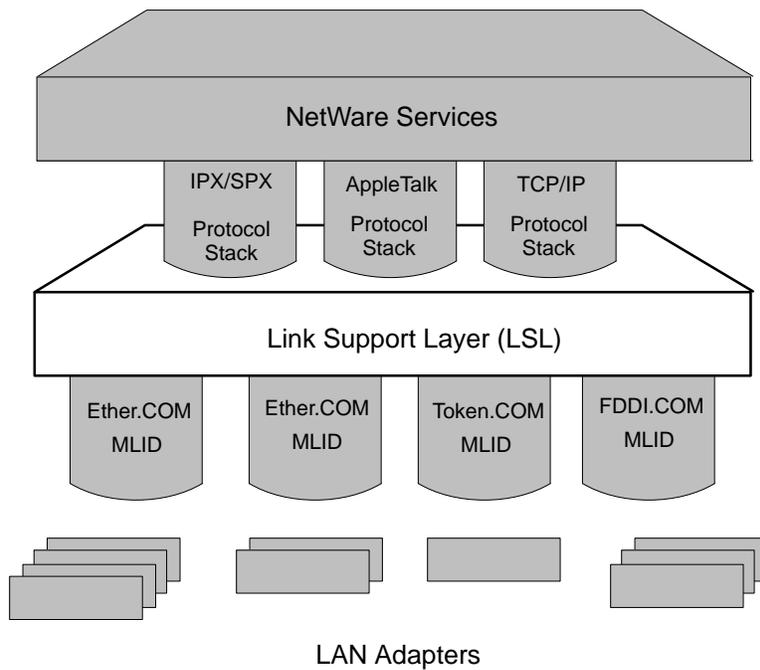
## UnbindFromMLID

<b>Description</b>	Informs the protocol stack that it should unbind from the specified board.
<b>Entry State</b>	<p><i>BX</i> is equal to <i>UNBIND_FROM_MLID</i> (0003h).</p> <p><i>CX</i> has the board number from which the protocol stack should unbind.</p> <p><i>ES:SI</i> has a pointer to a pointer to an optional parameter string that is protocol implementation-dependent.</p> <p>Interrupts are enabled.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p>Flags Z flag set according to <i>AX</i>.</p> <p>Interrupts are disabled.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The protocol stack unbound successfully.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h) The protocol was not bound to the board number in <i>CX</i>.</p> <p><i>LSLERR_BAD_COMMAND</i> (8008h) The protocol does not support this function.</p> <p>Other protocol implementation-dependent codes.</p>
<b>Remarks</b>	<p>Because most protocol stacks only need to unbind when they are unloaded, they usually do not implement this function. However, the protocol stack can implement this function if the protocol needs to support a specialized application.</p> <p>Protocols supporting mobility or dynamic binding should implement this function.</p> <p>The protocol stack must return <i>LSLERR_BAD_COMMAND</i> (8008h) if it chooses no to support this function after staying resident.</p>

□



Section III **Link Support Layer (LSL)**



## Section Overview

This section provides a chapter discussing the Link Support Layer (LSL) configuration and statistics table. It also includes chapters describing the LSL calls that are used by both the protocol stacks and the MLID.

*Chapter 8: Overview of the LSL* provides a brief overview of the LSL and its functions and a table of completion codes.

*Chapter 9: LSL Data Structures* describes the fields in the LSL configuration and statistics table.

*Chapter 10: LSL Protocol Stack Support Routines* describes the LSL functions available to protocol stacks.

*Chapter 11: LSL MLID Support Routines* describes the support routines provided by the LSL for MLIDs.

*Chapter 12: LSL General Services* describes general services provided by the LSL for protocol stacks and MLIDs.





*Chapter 8* **Overview of the LSL**

## **Overview**

This chapter provides a brief overview of the Link Support Layer (LSL) and its functions. It also lists the completion codes the LSL returns in the support routines.

## Link Support Layer (LSL)

The LSL handles the communication between protocol stacks and MLIDs. Because the ODI specification allows the physical topology to support many different types of protocols, every MLID sends and receives packets of different frame types that are destined for different protocol stacks. The LSL acts as a demultiplexer, or switchboard, and determines the protocol stack or MLID that receives the packet.

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:

- Optionally allows a protocol stack to obtain and return Event Control Blocks (ECBs). (ECBs are control structures that are used to send or receive packets and to schedule events.)
- Queues and recovers ECBs for later use.
- Allows protocol stacks to obtain timing services.
- Allows protocol stacks to determine stack IDs and Protocol IDs (PIDs).
- 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 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 can cause the MLID to shut down or reset.)

**Note** Versions of the LSL are available for the NEC, Fujitsu, IBM, and compatible computers.

## LSL Completion Codes

The following are the completion codes used by the LSL and other modules (defined in the *ODI.INC* file).

- *LSL\_SUCCESSFUL* (0000h)
- *LSL\_PENDING\_SUCCESS* (0001h)

- `LSLERR_OUT_OF_RESOURCES` (8001h)
- `LSLERR_BAD_PARAMETER` (8002h)
- `LSLERR_NO_MORE_ITEMS` (8003h)
- `LSLERR_ITEM_NOT_PRESENT` (8004h)
- `LSLERR_FAIL` (8005h)
- `LSLERR_RX_OVERFLOW` (8006h)
- `LSLERR_CANCELLED` (8007h)
- `LSLERR_BAD_COMMAND` (8008h)
- `LSLERR_DUPLICATE_ENTRY` (8009h)
- `LSLERR_NO_SUCH_HANDLER` (800Ah)
- `LSLERR_NO_SUCH_DRIVER` (800Bh)
- `LSLERR_PACKET_ERRORED` (800Ch)

## Specification Version String

In order to identify which version of this specification an LSL conforms to, a version string (the “specification version string”) is embedded within the LSL. 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 is located in the LSL’s initialization variable; it is not available at runtime.

```
ODISPEC db 'ODI_SPEC_VERSION: 4.00',0
```

## LSL Commandline Switches

Valid switches: U, R, F, ?, H, V, C=, S.

Only the “U, ?, C=” are documented in the help.

'U' or 'R' Unloads the LSL if possible.

'F' Forces the unload.

'H' an equivalent to '?', and is used by many of Novell’s Other utilities.

'V' Shows the signon and version information.

'C=' Used to change the path and/or filename of the configuration file and is the only 2 letter switch that is valid.

'S' System check switch that will display the internals of the LSL.

The following information is provided:

The LSL Version is vX.XX

The LSL Configuration Table Version is vX.XX

The LSL Machine type is set to XXX machines.

LSL Initial Entry point = Seg:Offset

LSL protocol Support Entry point = Seg:Offset

LSL General Services Entry point = Seg:Offset

LSL MLID Support Entry Point = Seg:Offset

LSL Config Table pointer = Seg:Offset

LSL Stat Table pointer = Seg:Offset

The Configuration File used was [Path\Filename].

Buffers, XX, Buffer size XXXX bytes, Memory Pool X Bytes.

In addition if there is a MEMPOOL value > zero, then the MEMPOOL Memory stat structure will also be displayed

Memory Pool

Memory Available: X paragraphs

MemoryInuse: X paragraphs

Largest Available Block	X paragraphs
Number Available Blocks	X
Memory Management Overhead	X bytes
Minimum Allowed Allocation	X bytes

## Custom Configuration Files

When using the “C=” commandline switch, the LSL first tries the [path]\filename as given. If the file is not found, the filename is searched for in the current working directory. If the filename still cannot be found, the LSL looks in the load directory. The filename is considered valid even if the path was incorrect. After parsing the configuration file, the LSL displays the relative path and filename of the configuration file that was parsed.

All MLIDs using an unmodified LAN driver toolkit will use the same configuration file that the LSL used. It is recommended and strongly suggested that all protocol stacks and shells (or requesters) query the LSL and use the same configuration file unless the user explicitly gives a commandline parameter to require a different configuration file.







*Chapter 9* **LSL Data Structures**

## **Overview**

This chapter describes the LSL configuration and statistics tables and each of the fields in these structures.

## LSL Configuration Table

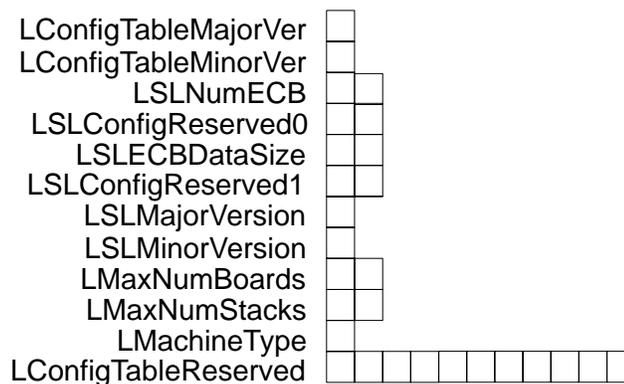
The following describes the LSL configuration table in detail; specifically, it includes the configuration table structure definition and a description of each of the configuration table fields.

### LSL Configuration Table Structure

```

LSLConfigurationStructure      struc
    LConfigTableMajorVer      db    01
    LConfigTableMinorVer      db    11
    LSLNumECB                  dw    0
    LSLConfigReserved0        dw    0
    LSLECBDataSize            dw    0
    LSLConfigReserved1        dw    0
    LSLMajorVersion           db    ?
    LSLMinorVersion           db    ?
    LMaxNumBoards              dw    ?
    LMaxNumStacks              dw    ?
    LMachineType               db    0
    LConfigTableReserved      db    11 dup (0)
LSLConfigurationStructure      ends
    
```

**Figure 9.1**  
Graphic Representation  
of the LSL  
Configuration Table



**Table 9.1 LSL Configuration Table**

Offset	Name	Size (in bytes)	Description
00h	LConfigTableMajorVer	01	This field has the major version number of the configuration table (0 through 99 decimal). For this specification, the major version number is 1.
01h	LConfigTableMinorVer	01	This field has the minor version number of the configuration table (0 through 99 decimal). For this specification, the minor version number is 11.
02h	LSLNumECB	02	The LSL optionally provides ECBs; this is a count of the total available ECBs.
04h	LSLConfigReserved0	02	Reserved
06h	LSLECBDataSize	02	The LSL optionally provides ECBs, this is the size of the data buffer associated with the ECB.

**Table 9.1 LSL Configuration Table**

Offset	Name	Size (in bytes)	Description
08h	LSLConfigReserved1	02	Reserved
0Ah	LSLMajorVersion	01	This field contains the major version of the LSL (0 through 99 decimal).
0Bh	LSLMinorVersion	01	This field contains the minor version of the LSL (0 through 99 decimal).
0Ch	LMaxNumBoards	02	This field has the maximum number of boards for which the LSL is configured.
0Eh	LMaxNumStacks	02	This field contains the maximum number of protocol stacks for which the LSL is configured. In other words, the value of this field is the maximum number of Protocol IDs identifying protocol stacks that the LSL can handle—for example, TCP/IP usually uses 3 PIDs: one for IP, one for ARP, and one for RARP.
	LMachineType	01	This field is defined specifically for each machine version of the 16-bit DOS ODI LSL. Currently there are three versions of the LSL available (IBM, NEC, Fujitsu, and their compatible computers). For each machine type, a language translation has been bound on for the default language.
10h	LConfigTableReserved	12	Reserved.

## LSL Statistics Table

The following describes the LSL statistics table in detail; specifically, it includes the LSL statistics table structure definition and a description of each of the statistics table fields.

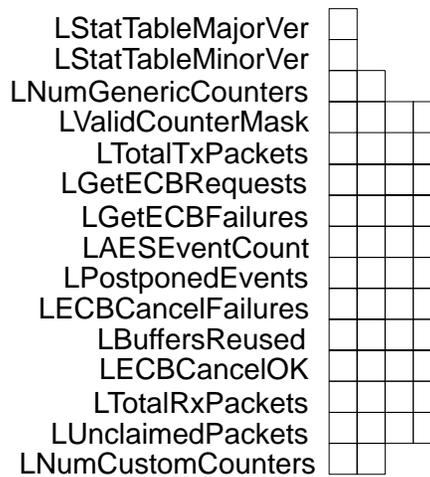
The LSL keeps a statistics table for the purpose of network management.

## LSL Statistics Table Structure

```

LSLStatStructure      struc
  LStatTableMajorVer  db 01
  LStatTableMinorVer  db 01
  LNumGenericCounters dw 10
  LValidCounterMask   dd 00000001001111111111111111111111b
  LTotalTxPackets     dw 2 dup (0)
  LGetECBRequests     dw 2 dup (0)
  LGetECBFailures     dw 2 dup (0)
  LAESEventCount      dw 2 dup (0)
  LPostponedEvents    dw 2 dup (0)
  LECBCancelFailures  dw 2 dup (0)
  LBuffersReused      dw 2 dup (0)
  LECBCancelOK        dw 2 dup (0)
  LTotalRxPackets     dw 2 dup (0)
  LUnclaimedPackets   dw 2 dup (0)
  LNumCustomCounters  dw 0
LSLStatStructure      ends
  
```

**Figure 9.2**  
Graphical Representation of the  
LSL Statistics Table



**Table 9.2 LSL Statistics Table Field Descriptions**

Offset	Name	Size (in bytes)	Description
00h	LStatTableMajorVer	01	This field contains the major version number of the statistics table (0 through 99 decimal). For this specification, the major version number is 1.
01h	LStatTableMinorVer	01	This field contains the minor version number of the statistics table (0 through 99 decimal). For this specification, the minor version number is 1.
02h	LNumGenericCounters	02	This field has the number of dword counters in the static portion of this table. For this specification, the number is 10.

**Table 9.2 LSL Statistics Table Field Descriptions**

Offset	Name	Size (in bytes)	Description
04h	LValidCountersMask	04	This field contains the bit mask that indicates which generic counters are used. The value 0 indicates "Used;" the value 1 indicates "Unused." The bit-to-counter correlations are determined by shifting left as you move down the counters in the table. For example, bit 31 corresponds to <i>TotalTxPackets</i> .
08h	LTOTALTxPackets	04	This field has the total number of <i>SendPacket</i> requests made to the LSL.
0Ch	LGetECBRequests	04	This field contains the number of times the LSL was requested to provide an ECB.
10h	LGetECBFailures	04	This field contains the number of times no ECB was provided by the LSL.
14h	LAESEventsCount	04	This field contains the number of completed AES events.
18h	LPostponedEvents	04	This field has the number of AES, send, and receive events that were postponed because of critical sections inside the MLIDs.
1Ch	LECBCancelFailures	04	This field contains the number of times <i>CancelAESEvent</i> was called and failed to find and cancel the specified AES ECB.
20h	LBuffersReused	04	This field contains the number of buffers that were reused.
24h	LECBCancelOK	04	This field has the number of successful calls to <i>CancelEvent</i> to cancel a receive or transmit ECB.
28h	LTOTALRxPackets	04	This field has the total number of <i>GetStackECB</i> requests the MLID has made to the LSL.
2Ch	LUnclaimedPackets	04	This field has the total number of times a packet was received and was not consumed by a protocol stack.
30h	LNumCustomCounters	02	This field has the total number of custom variables that follow this word.

There are *NumberCustom* dwords that start at offset 32h and that correspond to the custom statistics for the LSL. Following these dwords, *NumberCustom* pointers (4 bytes each) point to length preceded ASCII strings that describe each custom counter. Starting with this specification, there are no custom counters in use.





*Chapter 10* **LSL Protocol Stack Support Routines**

## Overview

The LSL contains a number of services that are available to protocol stacks. You invoke these services by calling the protocol stack support entry point obtained when the protocol stack locates the LSL.

The following is an alphabetical list of the LSL protocol stack support routines along with their entry point and function number.

<b>Descriptive Name</b>	<b>Function Name</b>	<b>Function No.</b>
AddProtocolID	PROTSUP_ADD_PID	23 (17h)
BindStack	PROTSUP_BIND_STACK_TO_MLID	21 (15h)
CancelAESEvent	PROTSUP_CANCEL_EVENT	4
DefragmentECB	PROTSUP_DEFRAG_ECB	2
DeregisterDefaultStackChain	PROTSUP_DEREGISTER_DEFAULT_CHAIN	30 (1Eh)
DeregisterPrescanRxChain	PROTSUP_DEREGISTER_PRESCAN_RX_CHAIN	31 (1Fh)
DeregisterPrescanTxChain	PROTSUP_DEREGISTER_PRESCAN_TX_CHAIN	32 (20h)
DeregisterRPLBootROM	PROTSUP_DEREGISTER_RPL_BOOTROM	9
DeregisterStack	PROTSUP_DEREGISTER_STACK	7
EndCriticalSection	PROTSUP_END_CRITICAL_SECTION	39 (27h)
GetBoundBoardInfo	PROTSUP_GET_BOUND_BOARD_INFO	42 (2Ah)
GetCriticalSectionStatus	PROTSUP_CRITICAL_SECTION_STATUS	40 (28h)
GetECB	PROTSUP_GET_ECB	0
GetHeldPacket	PROTSUP_GET_HELD_PACKET	14 (0Eh)
GetTickMarker	PROTSUP_GET_TICK_MARKER	26 (1Ah)
GetLSLConfiguration	PROTSUP_GET_LSL_CONFIG	25 (19h)
GetLSLStatistics	PROTSUP_GET_LSL_STATS	20 (14h)
GetMLIDControlEntry	PROTSUP_GET_MLID_CTL_ENTRY	18 (12h)
GetPIDFromStackIDBoard	PROTSUP_GET_PID_PROTNUM_MLIDNUM	17 (11h)
GetProtocolControlEntry	PROTSUP_GET_PROTO_CTL_ENTRY	19 (13h)
GetStackIDFromName	PROTSUP_GET_PROTNUM_FROM_NAME	16 (10h)
GetStartOfChain	PROTSUP_GET_START_CHAIN	33 (21h)
GetIntervalMarker	PROTSUP_GET_INTERVAL_MARK	5
HoldEvent	PROTSUP_HOLD_EVENT	37 (25h)
HoldPacket	PROTSUP_HOLD_PACKET	13 (0Dh)
ModifyStackFilter	PROTSUP_MODIFY_STACK_FILTER	43 (2Bh)
RegisterDefaultStack	PROTSUP_REGISTER_DEFAULT_CHAIN	27 (1Bh)
RegisterPrescanRxChain	PROTSUP_REGISTER_PRESCAN_RX_CHAIN	28 (1Ch)
RegisterPrescanTxChain	PROTSUP_REGISTER_PRESCAN_TX_CHAIN	29 (1Dh)
RegisterRPLBootROM	PROTSUP_REGISTER_RPL_BOOTROM	8
RegisterStack	PROTSUP_REGISTER_STACK	6
RelinquishControl	PROTSUP_RELINQUISH_CONTROL	24 (18h)
ResubmitDefault	PROTSUP_RESUBMIT_DEFAULT	34 (22h)
ResubmitPrescanRx	PROTSUP_RESUBMIT_PRESCAN_RX	35 (23h)
ResubmitPrescanTx	PROTSUP_RESUBMIT_PRESCAN_TX	36 (24h)
ReturnECB	PROTSUP_RETURN_ECB	1
ScanPacket	PROTSUP_SCAN_PACKET	15 (0Fh)
ScheduleAESEvent	PROTSUP_SCHEDULE_AES_EVENT	3

SendPacket	PROTSUP_SEND_PACKET	12 (0Ch)
ServiceEvents	PROTSUP_SERVICE_EVENTS	41 (29h)
StartCriticalSection	PROTSUP_START_CRITICAL_SECTION	38 (26h)
UnbindStack	PROTSUP_UNBIND_STACK_FROM_MLID	22 (16h)

The following is a function number ordered list of the LSL protocol stack support routines along with their entry point and function number.

<b>Descriptive Name</b>	<b>Function Name</b>	<b>Function No.</b>
GetECB	PROTSUP_GET_ECB	0
ReturnECB	PROTSUP_RETURN_ECB	1
DefragmentECB	PROTSUP_DEFRAG_ECB	2
ScheduleAESEvent	PROTSUP_SCHEDULE_AES_EVENT	3
CancelAESEvent	PROTSUP_CANCEL_EVENT	4
GetIntervalMarker	PROTSUP_GET_INTERVAL_MARK	5
RegisterStack	PROTSUP_REGISTER_STACK	6
DeregisterStack	PROTSUP_DEREGISTER_STACK	7
RegisterRPLBootROM	PROTSUP_REGISTER_RPL_BOOTROM	8
DeregisterRPLBootROM	PROTSUP_DEREGISTER_RPL_BOOTROM	9
SendPacket	PROTSUP_SEND_PACKET	12 (0Ch)
HoldPacket	PROTSUP_HOLD_PACKET	13 (0Dh)
GetHeldPacket	PROTSUP_GET_HELD_PACKET	14 (0Eh)
ScanPacket	PROTSUP_SCAN_PACKET	15 (0Fh)
GetStackIDFromName	PROTSUP_GET_PROTNUM_FROM_NAME	16 (10h)
GetPIDFromStackIDBoard	PROTSUP_GET_PID_PROTNUM_MLIDNUM	17 (11h)
GetMLIDControlEntry	PROTSUP_GET_MLID_CTL_ENTRY	18 (12h)
GetProtocolControlEntry	PROTSUP_GET_PROTO_CTL_ENTRY	19 (13h)
GetLSLStatistics	PROTSUP_GET_LSL_STATS	20 (14h)
BindStack	PROTSUP_BIND_STACK_TO_MLID	21 (15h)
UnbindStack	PROTSUP_UNBIND_STACK_FROM_MLID	22 (16h)
AddProtocolID	PROTSUP_ADD_PID	23 (17h)
RelinquishControl	PROTSUP_RELINQUISH_CONTROL	24 (18h)
GetLSLConfiguration	PROTSUP_GET_LSL_CONFIG	25 (19h)
GetTickMarker	PROTSUP_GET_TICK_MARKER	26 (1Ah)
RegisterDefaultStack	PROTSUP_REGISTER_DEFAULT_CHAIN	27 (1Bh)
RegisterPrescanRxChain	PROTSUP_REGISTER_PRESCAN_RX_CHAIN	28 (1Ch)
RegisterPrescanTxChain	PROTSUP_REGISTER_PRESCAN_TX_CHAIN	29 (1Dh)
DeregisterDefaultStackChain	PROTSUP_DEREGISTER_DEFAULT_CHAIN	30 (1Eh)
DeregisterPrescanRxChain	PROTSUP_DEREGISTER_PRESCAN_RX_CHAIN	31 (1Fh)
DeregisterPrescanTxChain	PROTSUP_DEREGISTER_PRESCAN_TX_CHAIN	32 (20h)
GetStartOfChain	PROTSUP_GET_START_CHAIN	33 (21h)
ResubmitDefault	PROTSUP_RESUBMIT_DEFAULT	34 (22h)
ResubmitPrescanRx	PROTSUP_RESUBMIT_PRESCAN_RX	35 (23h)
ResubmitPrescanTx	PROTSUP_RESUBMIT_PRESCAN_TX	36 (24h)
HoldEvent	PROTSUP_HOLD_EVENT	37 (25h)
StartCriticalSection	PROTSUP_START_CRITICAL_SECTION	38 (26h)
EndCriticalSection	PROTSUP_END_CRITICAL_SECTION	39 (27h)
GetCriticalSectionStatus	PROTSUP_CRITICAL_SECTION_STATUS	40 (28h)
ServiceEvents	PROTSUP_SERVICE_EVENTS	41 (29h)

GetBoundBoardInfo	PROTSUP_GET_BOUND_BOARD_INFO	42 (2Ah)
ModifyStackFilter	PROTSUP_MODIFY_STACK_FILTER	43 (2Bh)

## AddProtocolID

<b>Description</b>	Allows a protocol stack to register a Protocol ID for a given frame type and protocol stack combination
<b>Entry State</b>	<p><i>AX</i> has the frame type ID to which the new Protocol ID applies (for example, the frame type ID of ETHERNET_II is 2). See <i>ODI Specification Supplement: Frame Types and Protocol IDs</i> for a list of defined frame type IDs and commonly used Protocol IDs.</p> <p><i>BX</i> is equal to <i>PROTSUP_ADD_PID</i> (23 [017h]).</p> <p><i>CX:DI</i> has a pointer to a length-preceded, zero-terminated string containing the protocol's short name to which the new Protocol ID applies.</p> <p><i>ES:SI</i> has a pointer to a 6-byte Protocol ID to register.</p> <p>Interrupts are enabled.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p>Flags Z flag set according to <i>AX</i>.</p> <p>Interrupts are enabled.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The specified Protocol ID was successfully registered with the LSL.</p> <p><i>LSLERR_OUT_OF_RESOURCES</i> (8001h) The LSL has no resources to register another Protocol ID for the specified frame type.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h) The length of the specified protocol short name is either equal to 0 or is greater than 15.</p> <p><i>LSLERR_DUPLICATE_ENTRY</i> (8009h) A Protocol ID for the specified protocol stack and frame type has already been registered with the LSL.</p>
<b>Remarks</b>	This routine allows a protocol stack to register a Protocol ID for a given frame type and protocol stack combination. A protocol

stack should invoke the *GetPIDFromStackIDBoard* function before it calls this function because a Protocol ID might have been previously registered for the specified protocol and frame type combination. The protocol should use the previously registered PID.

## BindStack

<b>Description</b>	Binds the specified stack ID to the specified board.
<hr/>	
<b>Entry State</b>	<p><i>AX</i> has the protocol stack's assigned stack ID value.</p> <p><i>BX</i> is equal to <i>PROTSUP_BIND_STACK_TO_MLID</i> (21 [015h]).</p> <p><i>CX</i> has the logical board number to which to bind.</p> <p>Interrupts are unspecified.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p>Flags Z flag is set according to <i>AX</i>.</p> <p>Interrupts are unchanged.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The protocol stack bound successfully to the specified board.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h) Either the specified stack ID or the board number was invalid.</p> <p><i>LSLERR_DUPLICATE_ENTRY</i> (8009h) The specified binding already exists.</p>
<hr/>	
<b>Remarks</b>	<p>This routine binds a protocol stack to an adapter and frame type combination (logical board) that allows and enables packet reception. When this routine returns successfully, the specified binding has occurred. The bound protocol stack will receive the packets that contain the registered Protocol ID for that stack, which are received by the specified board, and that pass the current stack filters, which were setup during registration.</p>

## CancelAESEvent

### Description

Called by a protocol stack to cancel an AES ECB.

---

### Entry State

*BX*

is equal to *PROTSUP\_CANCEL\_EVENT* (4).

*ES:SI*

has a pointer to an ECB to be canceled (the ESR will not be called).

Interrupts

are disabled.

### Return State

*AX*

has a completion code.

Flags

Z flag is set according to *AX*.

*DS, BP, ES, SI, SS, SP*

are preserved.

### Completion Codes (AX)

*LSL\_SUCCESSFUL* (0000h)

The specified AES event was canceled.

*LSLERR\_ITEM\_NOT\_PRESENT* (8004h)

The specified ECB is not currently scheduled.

---

### Remarks

If the AES ECB was canceled the AES ECB's *Status* field will be set to *LSLERR\_CANCELLED* (8007h). The defined ESR will not be called.

### See Also

*ScheduleAESEvent*

## DefragmentECB

### Description

Copies a fragmented ECB into a continuous ECB.

---

### Entry State

*AX*

has an offset beyond the end of the destination ECB to where the ECB data will start to be copied. If *AX* is  $-1$  (0FFFFh), the ECB itself will not be copied, and the data pointed to by the source ECB will be copied into the destination buffer.

*BX*

is equal to *PROTSUP\_DEFRAG\_ECB* (2).

*ES:SI*

is a pointer to the data destination (an ECB or data buffer).

*CX:DI*

is a pointer to the ECB source data.

Interrupts

are unspecified.

### Return State

*AX*

is equal to *LSL\_SUCCESSFUL* (0000h).

Flags

are set according to *AX*.

Interrupts

state is preserved.

*DS, ES, SI, BP, SS, SP*

are preserved.

---

### Remarks

This function is used to defragment an ECB into a contiguous ECB. This function can be used to move only the data pointed to by the ECB data buffer pointers to a contiguous buffer by setting *AX* equal to  $-1$  (0FFFFh). Setting *AX* to a value other than  $-1$  will cause the ECB to be copied as well as the data pointed to by the ECB. The value in *AX* will be used to reserve space between the ECB and the start of the first data fragment. It is assumed that the data buffer pointed to by *ES:SI* is large enough to hold the source ECB, data, and the offset that will be introduced by *AX*. If *AX* is equal to  $-1$ , the destination buffer needs to be only as large as the source ECB.*DataLength* value.

## DeregisterDefaultStackChain

<b>Description</b>	Deregisters the default receive protocol stack described by the <i>StackChainStruc</i> structure.
<b>Entry State</b>	<p><i>BX</i></p> <p>is equal to <i>PROTSUP_DEREGISTER_DEFAULT_CHAIN</i> (30 [1Eh]).</p> <p><i>ES:SI</i></p> <p>is a pointer to the <i>StackChainStruc</i> structure for the protocol stack that describes the default protocol stack.</p> <p>Interrupts</p> <p>are disabled.</p>
<b>Return State</b>	<p><i>AX</i></p> <p>has a completion code.</p> <p>Flags</p> <p>Z flag set according to <i>AX</i>.</p> <p>Interrupts</p> <p>are unchanged.</p> <p><i>DS, BP, SS, SP</i></p> <p>are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h)</p> <p>The command was successfully executed.</p> <p><i>LSLERR_ITEM_NOT_PRESENT</i> (8004h)</p> <p>The <i>StackChainStruc</i> structure was not found in the chain for the board number described in the <i>StackChainStruc</i> structure.</p>
<b>Remarks</b>	<p>This function is called by protocol stacks that are to be removed from the protocol stack default chain. The LSL will release the <i>StackChainStruc</i> structure back to the calling protocol stack.</p> <p>After successfully returning from this function, the <i>StackChainStruc</i> structure is released back to the originating stack. After this function returns, the LSL will no longer present data to this stack.</p>

## DeregisterPrescanRxChain

<b>Description</b>	Deregisters the prescan receive protocol stack described by the <i>StackChainStruc</i> structure.
<hr/>	
<b>Entry State</b>	<p><i>BX</i> is equal to <i>PROTSUP_DEREGISTER_PRESCAN_RX_CHAIN</i> (31 [1Fh]).</p> <p><i>ES:SI</i> is a pointer to <i>StackChainStruc</i> structure for the deregistering protocol.</p> <p>Interrupts are disabled.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p>Flags Z flag set according to <i>AX</i>.</p> <p>Interrupts are unchanged.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The command was successfully executed.</p> <p><i>LSLERR_ITEM_NOT_PRESENT</i> (8004h) The <i>StackChainStruc</i> structure was not found in the chain for the board number described in the <i>StackChainStruc</i>.</p>
<hr/>	
<b>Remarks</b>	<p>This function is called by protocol stacks that are to be removed from the protocol stack prescan receive chain.</p> <p>After successfully returning from this function, the <i>StackChainStruc</i> structure is released back to the originating stack. After this function returns, the LSL will no longer present data to this stack.</p>

## DeregisterPrescanTxChain

**Description** Removes a prescan protocol stack from the prescan chain for the board number described in the *StackChainStruc* structure.

---

**Entry State**

*BX*  
is equal to *PROTSUP\_DEREGISTER\_PRESCAN\_TX\_CHAIN* (32 [20h]).

*ES:SI*  
Pointer to the prescan transmit protocol stack's *StackChainStruc* structure.

Interrupts  
are disabled.

**Return State**

*AX*  
has a completion code.

Flags  
Z flag set according to *AX*.

Interrupts  
are unchanged.

*DS, BP, SS, SP*  
are preserved.

**Completion Codes (AX)**

*LSL\_SUCCESSFUL* (0000h)  
The command was successfully executed.

*LSLERR\_ITEM\_NOT\_PRESENT* (8004h)  
The *StackChainStruc* structure was not found in the chain for the board number described in *StackChainStruc*.

---

**Remarks** Called by protocol stacks to be removed from the protocol stack prescan transmit chain.

After successfully returning from this function, the *StackChainStruc* structure is released back to the originating stack. After this function returns, the LSL will no longer present data to this stack.

## DeregisterRPLBootROM

<b>Description</b>	Deregisters an RPL boot ROM stack from the LSL.
<b>Entry State</b>	<p><i>AX</i> has the board number to deregister.</p> <p><i>BX</i> is equal to <i>PROTSUP_DEREGISTER_RPL_BOOTROM</i> (9).</p> <p>Interrupts are disabled.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p>Flag is set according to <i>AX</i>.</p> <p>Interrupts are disabled.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The protocol stack was successfully deregistered.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h) The specified board does not exist.</p> <p><i>LSLERR_ITEM_NOT_PRESENT</i> (8004h) The protocol stack is not presently registered.</p>
<b>Remarks</b>	This routine deregisters the protocol stack used by the RPL boot ROM from the specified board. After this call, the protocol stack will not receive any more incoming packets and must make the <i>RegisterRPLBootROM</i> call again to start receiving packets.

## DeregisterStack

**Description** Removes a bound protocol stack from the LSL's list of known protocol stacks.

---

**Entry State**

*AX* has the protocol stack ID value.

*BX* is equal to *PROTSUP\_DEREGISTER\_STACK* (7).

Interrupts are disabled.

**Return State**

*AX* has a completion code.

Flag Z flag set according to *AX*.

Interrupts are disabled.

*DS, BP, SS, SP* are preserved.

**Completion Codes (AX)**

*LSL\_SUCCESSFUL* (0000h)  
The protocol stack was successfully deregistered.

*LSLERR\_ITEM\_NOT\_PRESENT* (8004h)  
The protocol stack is not presently registered.

---

**Remarks**

After this call, the protocol stack will not receive any more incoming packets and must make the *RegisterStack* and *BindStack* calls again to start receiving packets.

This command implicitly unbinds the protocol stack from all the MLIDs to which it was bound.

## EndCriticalSection

**Description** Ends a critical section.

---

**Entry State** *BX*  
is equal to *PROTSUP\_END\_CRITICAL\_SECTION* (39 [27h]).  
Interrupts  
are disabled.

**Return State** Interrupts  
disabled but might have been enabled during the routine.  
*DS, BP, SS, SP*  
are preserved.  
All other registers are destroyed.

---

**Remarks** This function is called by stacks that have previously started a critical section by a call to *StartCriticalSection*.  
*StartCriticalSection* should always be used in conjunction with a subsequent call to *EndCriticalSection*.

## GetBoundBoardInfo

<b>Description</b>	Used to obtain information about a board.
<b>Entry State</b>	<p><i>AX</i> is equal to the board number.</p> <p><i>BX</i> is equal to <i>PROTSUP_GET_BOUND_BOARD_INFO</i> (42 [2Ah]).</p> <p>Interrupts are in any state.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p><i>DX:BX</i> has a StackID mask indicating which stacks are bound to the given board number if <i>AX</i> equals <i>LSL_SUCCESSFUL</i> (0000h).</p> <p>Flags Z flag is set according to <i>AX</i>.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p> <p><i>CX, DI, SI</i> are destroyed.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The StackID mask is in <i>DX:BX</i>.</p> <p><i>LSLERR_ITEM_NOT_PRESENT</i> (8004h) The board number is invalid.</p>
<b>Remarks</b>	<p>If successful this function returns in <i>DX:BX</i> a mask of stack IDs for the stacks that are bound to the board identified by the board number in <i>AX</i>—for example, if a given board has two stacks bound to it, one with stack ID 0 and the other with stack ID 1, then bits 0 and 1 would be set (<i>BX</i> = 3).</p> <p><b>Note</b> Currently, the maximum number of stacks is 16, therefore <i>DX</i> will always return 0.</p>

## GetCriticalSectionStatus

<b>Description</b>	Returns the number of critical sections that are currently active.
<b>Entry State</b>	<p><i>BX</i></p> <p>is equal to <i>PROTSUP_CRITICAL_SECTION_STATUS</i> (40 [28h]).</p> <p>Interrupts are unspecified.</p>
<b>Return State</b>	<p><i>BX</i></p> <p>has the total number of outstanding calls to start critical section.</p> <p>Interrupts are unchanged.</p> <p>All other registers are preserved.</p>

---

## GetECB

### Description

Allocates one of the LSL's ECB buffers.

---

### Entry State

*BX*

is equal to *PROTSUP\_GET\_ECB* (0).

Interrupts

are disabled.

### Return State

*AX*

has a completion code.

*ES:SI*

is a pointer to the ECB if *AX* is *LSL\_SUCCESSFUL* (0000h).

*DS, BP, SS, SP*

are preserved.

### Completion Codes (AX)

*LSL\_SUCCESSFUL* (0000h)

The ECB was allocated successfully.

*LSLERR\_OUT\_OF\_RESOURCES* (8001h)

Out of ECBs.

---

### Remarks

When you call *GetECB*, it tries to obtain an ECB from the LSL's pool of ECBs. The end user must place "BUFFERS xx yy" under the link support section heading in the NET.CFG file. ECBs obtained from the LSL should be returned with the *ReturnECB* function. If the LSL was configured with no ECBs or there are no ECBs left in the pool, this function returns *LSLERR\_OUT\_OF\_RESOURCES* (8001h).

## GetHeldPacket

**Description** Allows a protocol stack to remove an ECB from the hold queue.

### Entry State

*BX*

is equal to *PROTSUP\_GET\_MLID\_PACKET* (14 [0Eh]).

*ES:SI*

If *ES:SI* equals 0:0

*AX* stack ID

*CX* match word

The first ECB that satisfies the following two conditions is removed from the hold queue:

The ECB's stack ID matches the value in *AX*.

The first word of the protocol workspace matches the value in *CX*.

Note, if *CX* = 0FFFFh, the match on the protocol workspace will be ignored.

If *ES:SI* is not equal to 0:0

*AX* stack ID

The ECB indicated in *ES:SI* is removed from the hold queue.

Interrupts

are disabled.

### Return State

*AX*

has a completion code.

*ES:SI*

has a pointer to the ECB if *AX* equals *LSL\_SUCCESSFUL* (0000h).

Interrupts

are disabled.

Flags

are set according to *AX*.

*DS, BP, SS, SP*

are preserved.

### Completion Codes (AX)

*LSL\_SUCCESSFUL* (0000h)

The pointer to the ECB was found.

*LSLERR\_ITEM\_NOT\_PRESENT* (8004h)

The ECB was not found or the hold queue was empty.

*LSLERR\_BAD\_COMMAND* (8008h)

The function was not available.

### Remarks

This function is called to retrieve an ECB that was previously placed on the hold queue by a call to *HoldPacket*.

**Note** ECBs placed on the hold queue might be used by other protocols in the system. In this case, this function returns *LSLERR\_ITEM\_NOT\_PRESENT* (8004h).

**Note** This function is not available if the LSL is not providing ECBs.

## GetIntervalMarker

**Description** Returns a timing marker in milliseconds. The timing marker is used for machine-independent time measurement.

---

**Entry State** *BX*  
is equal to *PROTSUP\_GET\_INTERVAL\_MARK* (5).

Interrupts  
state is unspecified.

**Return State** *DX:AX*  
has the current dword interval time in milliseconds.

Interrupts  
state is unchanged; interrupts are not enabled inside the routine.

All other registers are preserved.

---

**Remarks** The actual value returned has no relation to any real-world absolute time. However, when time marker values are compared with each other, the difference is elapsed time in milliseconds.

On an IBM compatible computers, the marker is incremented every 1/18th of a second (55 milliseconds). On a Japanese Fujitsu (FMR) computer, the marker is incremented every 10 milliseconds. On a Japanese NEC or compatible computer, the marker is incremented every 33 milliseconds.

## GetLSLConfiguration

<b>Description</b>	Obtains a pointer to the LSL configuration table.
<b>Entry State</b>	<i>BX</i> is equal to <i>PROTSUP_GET_LSL_CONFIG</i> (25 [19h]). Interrupts are unspecified.
<b>Return State</b>	<i>AX</i> is always equal to <i>LSL_SUCCESSFUL</i> (0000h). <i>ES:SI</i> has a pointer to the LSL configuration table. Flags Z flag is set according to <i>AX</i> . Interrupts state is unchanged. <i>DS, BP, SS, SP</i> are preserved.
<b>Remarks</b>	This routine returns a pointer to the LSL configuration table. The configuration table is normally used to obtain the LSL's current version number (for example, 2.10). The version number can be used to determine if certain LSL features are present. (See <i>Chapter 9: LSL Data Structures</i> for the LSL configuration table.)

## GetLSLStatistics

<b>Description</b>	Obtains a pointer to the LSL statistics table.
<b>Entry State</b>	<i>BX</i> is equal to <i>PROTSUP_GET_LSL_STATS</i> (20 [14h]). Interrupts are unspecified.
<b>Return State</b>	<i>AX</i> is always equal to <i>LSL_SUCCESSFUL</i> (0000h). <i>ES:SI</i> has a pointer to the LSL statistics table. Flags Z flag set according to <i>AX</i> . Interrupts are unchanged. <i>DS, BP, SS, SP</i> are preserved.
<b>Remarks</b>	This routine returns a pointer to the LSL statistics table. (See <i>Chapter 9: LSL Data Structures</i> for the LSL statistics table.)

## GetMLIDControlEntry

**Description** Returns the MLID control entry point for the specified logical board.

---

**Entry State** *AX* has the logical board number for which to return the entry point.

*BX* is equal to *PROTSUP\_GET\_MLID\_CTL\_ENTRY* (18 [12h]).

**Interrupts** are unspecified.

**Return State** *AX* has a completion code.

*ES:SI* has a pointer to an MLID control handler routine if *AX* equals *LSL\_SUCCESSFUL* (0000h).

**Flags** are set according to *AX*.

**Interrupts** are unchanged.

*DS, BP, SS, SP* are preserved.

**Completion Codes (AX)** *LSL\_SUCCESSFUL* (0000h)  
A pointer to an MLID control handler routine has been returned.

*LSLERR\_NO\_MORE\_ITEMS* (8003h)  
The board number does not exist and there are no boards at higher *AX* values.

*LSLERR\_ITEM\_NOT\_PRESENT* (8004h)  
The board number does not exist, but there might be boards at higher *AX* values.

---

**Remarks** This routine returns the specified MLID's control handler routine.

The MLID control handler routine can be called directly by a protocol or an application to obtain configuration information and to issue defined commands. (See *Chapter 18: MLID Control Routines* for the defined MLID control routines.)

## GetPIDFromStackIDBoard

**Description** Returns a Protocol ID that corresponds to a protocol and frame type combination. Protocol stacks use this value to fill in the *ProtocolID* field of all send ECBs.

---

**Entry State**

*AX* has the stack ID of a protocol for which to find the Protocol ID.

*BX* is equal to *PROTSUP\_GET\_PID\_PROTNUM\_MLIDNUM* (17 [11h]).

*CX* has the logical board number.

*ES:SI* has a pointer to a 6-byte buffer which will hold the Protocol ID if *AX* equals *LSL\_SUCCESSFUL* (0000h) when this routine returns.

Interrupts are unspecified.

**Return State**

*AX* has a completion code.

*ES:SI* has a pointer to a 6-byte buffer which will hold the Protocol ID if *AX* equals *LSL\_SUCCESSFUL* (0000h) when this routine returns.

Flag  
Z flag set according to *AX*.

Interrupts are unchanged; interrupts are not enabled inside this routine.

*DS, BP, SS, SP* are preserved.

**Completion Codes (AX)**

*LSL\_SUCCESSFUL* (0000h)  
A Protocol ID was found that corresponds to a protocol and frame type combination.

*LSLERR\_BAD\_PARAMETER* (8002h)  
The stack ID or the board number does not exist.

*LSLERR\_ITEM\_NOT\_PRESENT* (8004h)  
A Protocol ID has not been registered for the specified combination.

---

**Remarks** The returned Protocol ID is the value assigned to the protocol for the frame type (for example, ETHERNET\_II) that is

represented by the board number in *CX*. The Protocol ID is registered by MLIDs according to the “link driver . . . Protocol” keyword entries in the NET.CFG file. If a Protocol ID is not present, a protocol stack can add its own Protocol ID.

**Note** A protocol should not hard-code the Protocol ID because a user might want to change it through the NET.CFG file. The protocol stack should use this function to obtain it.

## GetProtocolControlEntry

<b>Description</b>	Returns the protocol's control entry point for the specified bound stack ID.
<hr/>	
<b>Entry State</b>	<p><i>AX</i> has the Stack ID to locate.</p> <p><i>BX</i> is equal to <i>PROTSUP_GET_PROTO_CTL_ENTRY</i> (19 [13h]).</p> <p>Interrupts are unspecified.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p><i>ES:SI</i> has a pointer to a protocol stack's control handler routine if <i>AX</i> equals <i>LSL_SUCCESSFUL</i> (0000h).</p> <p>Flags Z flag is set according to <i>AX</i>.</p> <p>Interrupts are unchanged.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The control handler routine of the specified bound protocol stack has been returned.</p> <p><i>LSLERR_NO_MORE_ITEMS</i> (8003h) The Stack ID does not exist and there are no more bound Stack IDs at a higher value.</p> <p><i>LSLERR_ITEM_NOT_PRESENT</i> (8004h) The Stack ID does not exist but there might be more bound Stack IDs at a higher value.</p>
<hr/>	
<b>Remarks</b>	The protocol control handler routine can be called directly by a protocol or an application to obtain configuration information and to issue defined commands. (See <i>Chapter 7: Protocol Stack Control Routines</i> for the defined protocol control routines.)
<b>See Also</b>	For prescan receive prescan and default stacks, see <i>GetStartOfChain</i> .

## GetStackIDFromName

<b>Description</b>	Allows a protocol stack or an application to obtain its own or any other stack ID.
<b>Entry State</b>	<p><i>BX</i></p> <p>is equal to <i>PROTSUP_GET_PROTNUM_FROM_NAME</i> (16 [10h]).</p> <p><i>ES:SI</i></p> <p>has a pointer to a length-preceded, zero-terminated string containing the short name of the protocol stack.</p> <p>Interrupts</p> <p>are unspecified.</p>
<b>Return State</b>	<p><i>AX</i></p> <p>has a completion code.</p> <p><i>BX</i></p> <p>has the stack ID for the specified protocol stack if <i>AX</i> equals <i>LSL_SUCCESSFUL</i> (0000h).</p> <p>Flag</p> <p>Z flag is set according to <i>AX</i>.</p> <p>Interrupts</p> <p>are unchanged.</p> <p><i>DS, BP, SS, SP</i></p> <p>are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h)</p> <p>The protocol stack or application has obtained a stack ID, which was returned in <i>BX</i>.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h)</p> <p>The length of the stack name is greater than 15 or equal to 0.</p> <p><i>LSLERR_ITEM_NOT_PRESENT</i> (8004h)</p> <p>A stack ID is not presently registered.</p>
<b>Remarks</b>	The stack name is not case sensitive.

## GetStartOfChain

**Description** Obtains the start of each type of protocol stack chain. The caller must treat the chain as **read-only**.

**Entry State** *BX* is equal to *PROTSUP\_GET\_START\_CHAIN* (33 [21h]).

*CX* is the board number.

Interrupts are disabled.

**Return State** *AX* has a completion code.

*ES:BX* is a pointer to a pointer to the start of the default protocol stack chain.

*ES:DI* is a pointer to a pointer to the start of the prescan transmit protocol stack chain.

*ES:SI* is a pointer to a pointer to the start of the prescan receive protocol stack chain.

Flags  
Z flag is set according to *AX*.

Interrupts are unchanged.

*DS, BP, SS, SP* are preserved.

**Note** The pointers point to a pointer equal to 0 (NULL) if there is nothing in the chain.

**Completion Codes (AX)** *LSL\_SUCCESSFUL* (0000h)  
The requested operation was completed successfully.

*LSLERR\_BAD\_PARAMETER* (8002h)  
The board does not exist.

**Remarks** This function provides a method to search for registered stacks through its *StackChainStruc* structures. All values in each *StackChainStruc* must be treated as read-only.

**See Also** *ModifyStackFilter*

## GetTickMarker

**Description** Returns the number of ticks that have occurred since the LSL was loaded.

---

**Entry State** *BX*  
is equal to *PROTSUP\_GET\_TICK\_MARKER* (26 [1Ah]).  
Interrupts  
are unspecified.

**Return State** *AX*  
is equal to the number of ticks since the LSL was loaded.  
Interrupts  
are unchanged.  
All registers are preserved except *AX*.

---

**Remarks** The tick count is only updated when interrupts are enabled.  
  
On an IBM compatible computers, the marker is incremented every 1/18th of a second (55 milliseconds). On a Japanese Fujitsu (FMR) computer, the marker is incremented every 10 milliseconds. On a Japanese NEC or compatible computer, the marker is incremented every 33 milliseconds.

## HoldEvent

**Description**

Places an ECB on the LSL's event hold queue.

---

**Entry State**

*BX*

is equal to *PROTSUP\_HOLD\_EVENT* (37 [25h]).

*ES:SI*

is a pointer to a completed ECB.

Interrupts

are disabled.

**Return State**

*AX*

has a completion code.

Flags

Z flag is set according to *AX*.

Interrupts

are unchanged.

*DS, BP, SS, SP*

are preserved.

---

**Remarks**

ECBs are placed on the event hold queue. The ECB's ESR routines are called by the LSL during a call to *ServiceEvents*.

This function is called when an ECB is filled in. Protocol stacks that have processed a prescan ECB and obtained a new ECB will fill in the new ECB and then call this function with *ES:SI* pointing to the new ECB. When the ESR of the new ECB is called, the ESR of the prescan ECB should be called, or the prescan ECB should be placed on the hold event queue (by calling this function) at that time.

## HoldPacket

**Description** Places the specified ECB on the end of the LSL's ECB hold queue.

---

**Entry State**

*BX*  
is equal to *PROTSUP\_HOLD\_PACKET* (13 [0Dh]).

*ES:SI*  
is a pointer to the ECB to be placed in the queue.

Interrupts  
are disabled.

**Return State**

*AX*  
has a completion code.

Interrupts  
are disabled.

*DS, SS, SP, BP, ES, SI*  
are preserved.

**Completion Codes (AX)**

*LSL\_SUCCESSFUL* (0000h)  
The requested operation was completed successfully.

*LSLERR\_BAD\_COMMAND* (8008h)  
The LSL is not configured to support this function.

---

**Remarks** Protocols that are not managing their own ECB buffers can use this function to hold an ECB on the LSL's hold queue. It might be taken by another stack if the ECB is from the original pool of ECBs provided by the LSL.

**Note** This function is not valid if no ECBs are provided by the LSL.

## ModifyStackFilter

<b>Description</b>	Called by protocol stacks to modify or read its <i>StkChnMask</i> .
<hr/>	
<b>Entry State</b>	<p><i>AX</i> is the board number.</p> <p><i>BX</i> is equal to <i>PROTSUP_MODIFY_STACK_FILTER</i> (43 [2Bh]).</p> <p><i>CX</i> has the stack ID (bound stacks) or chain ID (prescan and default stacks).</p> <p><i>DX</i> has the new <i>StkChnMask</i> filter bits (if equal to 0, it reads the current value).</p>
<b>Return State</b>	<p><i>DX</i> has the current <i>StkChnMask</i> value for this board and stack combination.</p> <p>Interrupts are unchanged.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The requested operation was completed successfully.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h) The board Value is incorrect.</p> <p><i>LSLERR_ITEM_NOT_PRESENT</i> (8004h) The stack/chain ID was not found.</p>
<hr/>	
<b>Remarks</b>	<p>This function is used to change the <i>StackChainStruc</i> structure that the protocol stack registered with the LSL. Changes should only be made by the LSL until the protocol deregisters. Bound stacks have an internal filter that the LSL maintains; this function is used to change that filter. A program can call this to determine what a particular stack is filtering on.</p> <p>By default, the <i>StkChnMask</i> field is set to (DEST_MULTICAST + DEST_BROADCAST + DEST_DIRECT = 83h) when a protocol registers.</p> <p>This function is called to change or read the <i>StkChainMask</i> value of the protocol identified by the stack ID passed in <i>CX</i>. Protocols should not modify the filter mask directly.</p>

## RegisterDefaultStackChain

**Description** Called to place the protocol in the default protocol stack chain.

---

**Entry State** *BX*  
is equal to *PROTSUP\_REGISTER\_DEFAULT\_CHAIN* (27 [1Bh]).

*ES:SI*  
has a pointer to the *StackChainStruc* structure.

Interrupts  
are enabled; call only at process time.

**Return State** *AX*  
has a completion code.

Flags  
Z flag set according to *AX*.

Interrupts  
are enabled, but might have been disabled in the routine.

*DS, BP, SS, SP*  
are preserved.

**Completion Codes (AX)** *LSL\_SUCCESSFUL* (0000h)  
The command was successfully executed.

*LSLERR\_BAD\_PARAMETER* (8002h)  
The board does not exist or a bad position requested.

*LSLERR\_DUPLICATE\_ENTRY* (8009h)  
Requested chain position already occupied. Since there can only be one first and one last in the chain, this error code only occurs if *STACK\_REQ\_FIRST* (0) or *STACK\_REQ\_LAST* (4) is requested. The other position requests are limited only by system memory.

---

**Remarks** Default protocols are generally protocols that have a number of Protocol IDs it is servicing. Default stacks are used for 802.2 protocol stacks—for example, where the DSAP and SSAP are not constant values.

A default stack makes this call after it has fully initialized itself. After this call returns, packets not consumed by a previous stack will start to be presented. The default stack's receive handler can be called any time after this call is initialized. The default stack's receive handler is called with *DS:DI* pointing at a *LookAheadStruc* structure.

The relative position in a default stack chain is determined by the values in the *StackChainStruc* structure that is passed in.

The *StkChnMask* field is set to (DEST\_MULTICAST + DEST\_BROADCAST + DEST\_DIRECT). PROTSUP\_MODIFY\_STACK\_FILTER is used to change the *StkChnMask* value.

Each packet received by *BoardNumber*, listed in the *StackChainStruc* structure, that meets one of the *StkChnMask* field requirements, and has not been consumed by a prior prescan or bound stack receive handler, is handed to the receive handler pointed to by *StkChnHandler*.

The *StackChainStruc* structure passed in becomes linked in to the LSL's chain structure and must remain available to the LSL until after the deregister call is made.

## RegisterPrescanRxChain

<b>Description</b>	Called to place the protocol in the prescan receive protocol stack chain.
<b>Entry State</b>	<p><i>BX</i></p> <p>is equal to <i>PROTSUP_REGISTER_PRESCAN_RX_CHAIN</i> (28 [1Ch]).</p> <p><i>ES:SI</i></p> <p>has a pointer to the <i>StackChainStruc</i> structure.</p> <p>Interrupts</p> <p>are enabled; call only at process time.</p>
<b>Return State</b>	<p><i>AX</i></p> <p>has a completion code.</p> <p>Flags</p> <p>Z flag is set according to <i>AX</i>.</p> <p>Interrupts</p> <p>are enabled but might have been disabled in the routine.</p> <p><i>DS, BP, SS, SP</i></p> <p>are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h)</p> <p>The command was successfully executed.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h)</p> <p>The board number does not exist or a bad position requested.</p> <p><i>LSLERR_DUPLICATE_ENTRY</i> (8009h)</p> <p>Requested chain position already occupied. Since there can only be one first and one last in the chain, this error code only occurs if <i>STACK_REQ_FIRST</i> (0) or <i>STACK_REQ_LAST</i> (4) is requested. The other position requests are limited only by system memory.</p>
<b>Remarks</b>	<p>A prescan stack makes this call after it has fully initialized itself. After this call returns, packets not consumed by a previous stack will start to be presented. The prescan stack's receive handler can be called at any time after this call is initialized. The prescan stack's receive handler is called with <i>DS:DI</i> pointing at a <i>LookAheadStruc</i> structure.</p> <p>The relative position in a prescan receive stack chain is determined by the values in the <i>StackChainStruc</i> structure that is passed in. The <i>StkChnMask</i> field is set to (<i>DEST_MULTICAST</i> + <i>DEST_BROADCAST</i> +</p>

DEST\_DIRECT). PROTSUP\_MODIFY\_STACK\_FILTER is used to change the *StkChnMask* value.

Each packet received by *BoardNumber*, listed in the *StackChainStruc* structure, that meets one of the *StkChnMask* field requirements, and has not been consumed by a prior prescan or bound stack receive handler, is handed to the receive handler pointed to by *StkChnHandler*.

The *StackChainStruc* structure passed in becomes linked in to the LSL's chain structure and must remain available to the LSL until after the deregister call is made.



The *StackChainStruc* structure passed in becomes linked in to the LSL's chain structure and must remain available to the LSL until after the deregister call is made.

All transmit ECBs that are not consumed by an up chain prescan transmit stack will be presented to the prescan transmit stack.

## RegisterRPLBootROM

**Description** Allows the RPL boot ROM protocol stack the ability to process packets by registering a default protocol stack with the LSL for the specified board.

---

**Entry State** *AX* has the logical board number for which the boot ROM protocol stack should register as a default stack.

*BX* is equal to *PROTSUP\_REGISTER\_RPL\_BOOTROM* (8).

*ES:SI* has a pointer to an 8-byte table.

Interrupts are unspecified.

**Return State** *AX* has a completion code.

Flags Z flag is set according to *AX*.

Interrupts are unchanged; interrupts are not enabled inside of this routine.

*DS, BP, SS, SP* are preserved.

**Completion Codes (AX)** *LSL\_SUCCESSFUL* (0000h)  
The protocol stack registered successfully.

*LSLERR\_BAD\_PARAMETER* (8002h)  
The board does not exist.

*LSLERR\_DUPLICATE\_ENTRY* (8009h)  
An RPL boot ROM stack is already registered for the given board number.

---

**Remarks** The protocol stack can receive packets immediately after this routine returns.

The 8-byte table passed in *ES:SI* must contain the information in *RPLBootROMInfoStruc*.

Only one RPL boot ROM can be registered per board.

---

## RPLBootROMInfoStruc Structure

```
RPLBootStackInfoStruc          struc
    RPLBootStackReceiveHandler  dd ?
    RPLBootStackControlHandler  dd ?
RPLBootStackInfoStruc          ends
```

### *RPLBootStackReceiveHandler*

This field has a far pointer to the protocol stack's receive handler routine. This routine is called when the MLID receives a packet and neither the prescan nor the bound stacks have accepted the packet.

### *RPLBootStackControlHandler*

This field contains a far pointer to the RPL boot ROM protocol stack's control handler routine. Applications might call this routine.

**Note** The *RPLBootStackInfoStruc* structure does not need to be present after *RegisterRPLStack* returns.

## RegisterStack

**Description** Registers a bound protocol stack with the LSL and returns an LSL-assigned handle for the stack (the stack ID).

---

**Entry State**

*BX*  
is equal to *PROTSUP\_REGISTER\_STACK* (6).

*ES:SI*  
has a pointer to a 12-byte table.

Interrupts  
are unspecified.

**Return State**

*AX*  
has a completion code.

*BX*  
has the assigned stack ID if the call was successful.

Flags  
Z flag is set according to *AX*.

Interrupts  
are unchanged; interrupts are not enabled inside this routine.

*DS, BP, SS, SP*  
are preserved.

**Completion Codes (AX)**

*LSL\_SUCCESSFUL* (0000h)  
The stack has been successfully registered.

*LSLERR\_OUT\_OF\_RESOURCES* (8001h)  
The maximum number of stacks is already registered.

*LSLERR\_BAD\_PARAMETER* (8002h)  
The stack name length was greater than 15 or is equal to 0.

*LSLERR\_DUPLICATE\_ENTRY* (8009h)  
The specified stack is already registered.

---

**Remarks**

The 12-byte table passed in *ES:SI* must contain the *StackInfoStruc* structure.

### StackInfoStruc Structure

```
StackInfoStruc    struc
    StackNamePtr    dd    ?
    StackReceiveHandler dd    ?
    StackControlHandler dd    ?
StackInfoStruc    ends
```

*StackNamePtr*

This field contains a far pointer to the protocol stack's short name (for example, IPX). This string is length-preceded and

zero-terminated. (The length byte does not include the length byte or zero termination byte.)

*StackReceiveHandler*

This field has a far pointer to the protocol's receive handler routine. This routine is called when a packet has been received by an MLID with the protocol stack's Protocol ID and has passed the stack's packet filtering. The filter is set by the LSL to **83h (DEST\_MULTICAST + DEST\_BROADCAST + DEST\_DIRECT)**.

*StackControlHandler*

This field has a far pointer to the protocol stack's control handler routine. Applications call this routine.

The *StackInfoStruc* structure and the string pointed to by *StackNamePtr* do not need to be present after *RegisterStack* returns. The protocol stack's stack receive handler will not be called until the protocol has bound to a board (see *BindStack*).

The stack name must uniquely identify the registering stack.

A protocol stack uses *ModifyStackFilter* to modify its filter from the default value of **83h (DEST\_MULTICAST + DEST\_BROADCAST + DEST\_DIRECT)**.

## RelinquishControl

**Description** Allows a protocol stack to yield control to the LSL, allowing the LSL to perform any necessary background processing.

---

**Entry State** *BX*  
is equal to *PROTSUP\_RELINQUISH\_CONTROL* (24 [18h]).  
Interrupts  
are enabled.

**Return State** Interrupts  
are enabled.  
*DS, BP, SS, SP*  
are preserved.

---

**Remarks** Protocol stacks that are waiting for an event (such as *SendECB* to complete) to occur should make this call. MLIDs that are not interrupt driven and, therefore, rely on polling from the LSL to process network events, will be polled when a protocol stack invokes this function.

## ResubmitDefault

<b>Description</b>	Allows default chained protocol stacks that originally queued and later processed at process time receive ECB's to be passed back to the LSL for further processing by the default stack. The LSL passes the ECB to the next protocol stack in the chain.
<b>Entry State</b>	<hr/> <i>BX</i> is equal to <i>PROTSUP_RESUBMIT_DEFAULT</i> (34 [22h]).  <i>DS:DI</i> has a pointer to the <i>LookAheadStruc</i> structure.  <i>ES:SI</i> has a pointer to the current (your) stack chain node structure.
<b>Return State</b>	<i>ES:SI</i> has a pointer to the ECB to fill. The ECB fragment count and fragment descriptor fields describe the buffers to be filled if AX equals <i>LSL_SUCCESSFUL</i> (0000h).  Flags Z flag set according to AX.  Interrupts are enabled but might have been disabled during the routine.  <i>DS, BP, SS, SP</i> are preserved.
<b>Completion Codes (AX)</b>	<i>LSL_SUCCESSFUL</i> (0000h) The command was successfully executed. EI:SI points to an ECB that is to be filled.  <i>LSLERR_OUT_OF_RESOURCES</i> (8001h) The LSL was unable to obtain an ECB for this packet. EI:SI is not valid. The packet can be discarded.
<b>Remarks</b>	<hr/> A default stack makes this call to send the data to the next protocol in the chain.  A protocol that is resubmitting an ECB must provide the same level of look ahead service that the MLID provides if an ECB is returned; it is the protocol's responsibility to fill the ECB's buffer and place it on the LSL's hold queue.  Protocols are in effect logically MLIDs when they call this function. The <i>LookAheadStruc</i> structure must be fully filled out, and if an ECB is returned, the calling protocol must fill in the data buffer and all of the following fields:

- Status
- StackID
- ImmediateAddress
- DriverWorkspace
- PrevLink
- DriverWS
- DataLength

## ResubmitPrescanRx

<b>Description</b>	Allows prescan receive chained protocol stacks that originally queued and later processed at process time receive ECB's to be passed back to the LSL for further processing; the LSL passes it to the next protocol stack in the chain.
<b>Entry State</b>	<hr/> <i>BX</i> is equal to <i>PROTSUP_RESUBMIT_PRESCAN_RX</i> (35 [23h]).  <i>DS:DI</i> has a pointer to the <i>LookAheadStruc</i> structure.  <i>ES:SI</i> has a pointer to the current (your) stack chain node structure if <i>AX</i> equals <i>LSL_SUCCESSFUL</i> (0000h).
<b>Return State</b>	<i>ES:SI</i> has a pointer to the ECB to fill. The ECB fragment count and fragment descriptor fields describe the buffers to be filled.  Flag Z flag is set according to <i>AX</i> .  Interrupts are enabled but might have been disabled during the routine.  <i>DS, BP, SS, SP</i> are preserved.
<b>Completion Codes (AX)</b>	<i>LSL_SUCCESSFUL</i> (0000h) The command was successfully executed, and <i>ES:SI</i> points to the ECB to be filled.  <i>LSLERR_OUT_OF_RESOURCES</i> (8001h) The LSL was unable to obtain an ECB for this packet, and <i>ES:SI</i> is invalid.
<b>Remarks</b>	<hr/> A prescan receive protocol stack makes the call to send the data to the next protocol in the stack.  A protocol that is resubmitting an ECB must provide the same level of look ahead service that the MLID provides if an ECB is returned; it is the protocol's responsibility to fill the ECB's buffer and place it on the LSL's hold queue.  Protocols are in effect logically MLIDs when they call this function. The <i>LookAheadStruc</i> structure must be fully filled out, and if an ECB is returned, the calling protocol must fill in the data buffer and all of the following fields:

- Status
- StackID
- ImmediateAddress
- DriverWorkspace
- PrevLink
- DriverWS
- DataLength

## ResubmitPrescanTx

**Description** Allows prescan transmit chained protocol stacks that originally queued and later processed at process time transmit ECB's to be passed back to the LSL for further processing; the LSL passes it to the next protocol stack in the chain.

### Entry State

*BX*

is equal to *PROTSUP\_RESUBMIT\_PRESCAN\_TX* (36 [24h]).

*DS:DI*

has a pointer to the current (your) stack chain node structure.

*ES:SI*

has a pointer to the transmit ECB if *AX = LSL\_SUCCESSFUL* (0000h).

### Return State

Flags

Z flag is set according to AX.

Interrupts

are enabled but might have been disabled during the routine.

*DS, BP, SS, SP*

are preserved.

### Completion Codes (AX)

*LSL\_SUCCESSFUL* (0000h)

The command was successfully executed.

*LSLERR\_BAD\_PARAMETER* (8002h)

The board does not exist.

### Remarks

Prescan transmit stacks use this function to transmit ECBs it originally queued and has processed. If a prescan transmit stack has data to transmit, it should use the *SendPacket* function. This function should only be used for ECBs originating external to the prescan transmit stack.

If a new ECB is created, the old ECB can be placed on the LSL event hold queue prior to making this call or when the ESR of the new ECB is called.

## ReturnECB

### Description

Returns an LSL provided ECB.

---

### Entry State

*BX*

is equal to *PROTSUP\_RETURN\_ECB* (1).

*ES:SI*

is a pointer to the ECB to be returned.

Interrupts

are disabled.

### Return State

*AX*

has a completion code.

Interrupts

are disabled.

*DS, BP, SS, SP*

are preserved.

### Completion Codes (AX)

*LSL\_SUCCESSFUL* (0000h)

The ECB was returned successfully.

*LSLERR\_BAD\_PARAMETER* (8002h)

The ECB is not the LSL's.

---

### Remarks

This function is called to return an ECB obtained from the LSL *GetECB* function. ECBs provided by other modules should not be passed to this function.

**Note** This function is not available if the LSL is not providing ECBs.

## ScanPacket

<b>Description</b>	Searches for a specific ECB on the LSL's ECB hold queue.
<hr/>	
<b>Entry State</b>	<p><i>AX</i> has the stack ID of the requesting stack.</p> <p><i>BX</i> is equal to <i>PROTSUP_SCAN_PACKET</i> (15 [0Fh]).</p> <p><i>ES:SI</i> is a pointer to the ECB to begin searching from. If equal to 0, the search starts from the beginning of the list.</p> <p><i>CX</i> 0FFFh ignore <i>ProtoWS</i> field. xxxx use in matching <i>ProtoWS</i> field.</p> <p>Interrupts are disabled.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p><i>ES:SI</i> is a pointer to the ECB if <i>AX</i> equals <i>LSL_SUCCESSFUL</i> (0000h).</p> <p>Interrupts are disabled.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The searched for ECB was found.</p> <p><i>LSLERR_NO_MORE_ITEMS</i> (8003h) The hold queue was empty or the ECB could not be found.</p> <p><i>LSLERR_BAD_COMMAND</i> (8008h) The LSL is not configured to support this function.</p>
<hr/>	
<b>Remarks</b>	This function is not available if the LSL is not providing ECBs.

## ScheduleAESEvent

**Description** Schedules an asynchronous event scheduler (AES) event.

---

**Entry State**

*BX*

is equal to *PROTSUP\_SCHEDULE\_AES\_EVENT* (3).

*ES:SI*

has a pointer to an *AESECB* structure (the ESR must be valid).

Interrupts

are unspecified.

**Return State**

*AX*

is always equal to *LSL\_SUCCESSFUL* (0000h).

Flags

Z flag is set according to *AX*.

Interrupts

state is unchanged; interrupts are disabled inside this routine.

*DS, BP, ES, SI, SS, SP*

are preserved.

---

**Remarks**

### AESECB Structure

```

AESECB      struc
    AESLink          dd      ?
    MSecondValue    dd      ?
    AESStatus        dw      ?
    AESESR           dd      ?
AESECB      ends
    
```

*AESLink*

This field is used by the LSL for list management.

*MSecondValue*

This field specifies the number of milliseconds to wait before invoking the defined *AESESR* routine. This field must be initialized each time the ECB is passed to *ScheduleAESEvent*.

*AESStatus*

This field is set to 0 when the *AESESR* is invoked.

*AESESR*

This field specifies a routine that is to be invoked when the specified time has expired. This field must point to a valid routine and only needs to be initialized once. The ESR must complete quickly because it is executing in the context of a timer interrupt.

The ESR can reschedule itself after it resets the *MSecondValue*, thus creating a simple polling function.

An ECB that is already in use by the LSL AES system must not be passed again to *ScheduleAESEvent*. To reset the AES event time, use *CancelAESEvent* and then issue a new *ScheduleAESEvent* call.

## SendPacket

**Description** Sends a packet, as described by an ECB, to the specified MLID for transmission.

---

**Entry State**

*BX*  
is equal to *PROTSUP\_SEND\_PACKET* (12 [0Ch]).

*ES:SI*  
has a pointer to a send ECB.

Interrupts  
are unspecified.

**Return State**

Interrupts  
are disabled.

*DS, BP, SS, SP*  
are preserved.

---

**Remarks**

The ECB's ESR might be invoked before this function returns.

Any module might use this function to send a packet. Protocols do not need to be registered or bound to use this function.

**Note** Prescan transmit stacks do not use this function to send data they received through their transmit handler.

## ServiceEvents

<b>Description</b>	This routine is invoked to complete the processing of network events queued on the LSL's holding queue. The ECBs in the queue are processed in the same order that they arrived in the queue (FIFO).
<b>Entry State</b>	<p><i>BX</i> is equal to <i>PROTSUP_SERVICE_EVENTS</i> (41 [29h]).</p> <p>Interrupts are unspecified.</p>
<b>Return State</b>	<p>Interrupts are disabled.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Remarks</b>	<p>An MLID's ISR calls <i>ServiceEvents</i> immediately before the ISR restores the registers. The MLID should have completed all hardware processing, and the ISR should be ready to accept a new interrupt.</p> <p>The <i>ServiceEvents</i> routine calls each queued ECB's ESR, to pass ECBs to the appropriate protocol stacks.</p> <p>While inside this routine, interrupts might be enabled and the MLID's send handler routine could be called.</p> <p> <b>Caution</b> If this routine is not called when queued events are outstanding, the events may never complete.</p> <p> <b>Caution</b> The MLID must not be in a critical section, or have any internal lock-out semaphores set, when this routine is called because this routine could call a protocol ESR which could issue a transmit request to the MLID and then wait for the transmit request to finish. If the MLID postpones the transmit request, the machine will deadlock.</p>

## StartCriticalSection

<b>Description</b>	Starts a critical section.
<b>Entry State</b>	<p><i>BX</i> is equal to <i>PROTSUP_START_CRITICAL_SECTION</i> (38 [26h]).</p> <p>Interrupts are disabled.</p>
<b>Return State</b>	<p><i>BX</i> is destroyed.</p> <p>Interrupts are unchanged.</p> <p>All other registers and direction flags are preserved.</p>
<b>Remarks</b>	Whenever this function is called, <i>EndCriticalSection</i> must also be called after the reason for the critical section no longer exists.

## UnbindStack

**Description** Unbinds a protocol stack from a logical board (an adapter and frame type combination).

### Entry State

*AX* has the protocol stack's assigned *StackID* value.

*BX* is equal to *PROTSUP\_UNBIND\_STACK\_FROM\_MLID* (22 [16h]).

*CX* has the logical board number from which to unbind.

Interrupts are unspecified.

### Return State

*AX* has a completion code.

Flag Z flag is set according to *AX*.

Interrupts are unchanged; interrupts are not enabled inside this routine.

*DS, BP, SS, SP* are preserved.

### Completion Codes (AX)

*LSL\_SUCCESSFUL* (0000h)  
The protocol stack was unbound from a logical board (adapter and frame type combination).

*LSLERR\_BAD\_PARAMETER* (8002h)  
The specified stack ID or the board number is invalid.

*LSLERR\_ITEM\_NOT\_PRESENT* (8004h)  
The specified binding does not exist.

### Remarks

After this routine successfully returns, packet reception between the specified protocol stack and logical board is disabled. *DeregisterStack* performs this operation implicitly.







*Chapter 11* **LSL MLID Support Routines**

## Overview

This chapter describes the LSL support routines that are used by the MLID and that are referred to in this document. A protocol stack may sometimes need to use these routines.

The following is an alphabetical list of the LSL MLID support routines along with their entry point and function number.

<b>Descriptive Name</b>	<b>Function Name</b>	<b>Function No</b>
AddProtocolID	MLIDSUP_ADD_PID	15 (0Fh)
CancelAESEvent	MLIDSUP_CANCEL_AES_EVENT	4
ControlStackFilter	MLIDSUP_CONTROL_STACK_FILTER	17 (11h)
DefragmentECB	MLIDSUP_DEFRAG_ECB	2
DeregisterMLID	MLIDSUP_DEREGISTER_MLID	6
EndCriticalSection	MLIDSUP_END_CRITICAL_SECTION	9
GetCriticalSectionStatus	MLIDSUP_CRITICAL_SECTION_STATUS	10 (0Ah)
GetECB	MLIDSUP_GET_ECB	0
GetIntervalMarker	MLIDSUP_GET_INTERVAL_MARKER	5
GetStackECB	MLIDSUP_GET_STACK_ECB	16 (10h)
HoldReceiveEvent	MLIDSUP_HOLD_RECV_EVENT	7
ReturnECB	MLIDSUP_RETURN_ECB	1
ScheduleAESEvent	MLIDSUP_SCHEDULE_AES_EVENT	3
SendComplete	MLIDSUP_SEND_COMPLETE	14 (0Eh)
ServiceEvents	MLIDSUP_SERVICE_EVENTS	11 (0Bh)
StartCriticalSection	MLIDSUP_START_CRITICAL_SECTION	8
Reserved	RESERVED	12 (0Ch)
Reserved	RESERVED	13 (0Dh)

The following is a function number ordered list of the LSL MLID support routines along with their entry point and function number.

<b>Descriptive Name</b>	<b>Function Name</b>	<b>Function No</b>
GetECB	MLIDSUP_GET_ECB	0
ReturnECB	MLIDSUP_RETURN_ECB	1
DefragmentECB	MLIDSUP_DEFRAG_ECB	2
ScheduleAESEvent	MLIDSUP_SCHEDULE_AES_EVENT	3
CancelAESEvent	MLIDSUP_CANCEL_AES_EVENT	4
GetIntervalMarker	MLIDSUP_GET_INTERVAL_MARKER	5
DeregisterMLID	MLIDSUP_DEREGISTER_MLID	6
HoldReceiveEvent	MLIDSUP_HOLD_RECV_EVENT	7
StartCriticalSection	MLIDSUP_START_CRITICAL_SECTION	8
EndCriticalSection	MLIDSUP_END_CRITICAL_SECTION	9
GetCriticalSectionStatus	MLIDSUP_CRITICAL_SECTION_STATUS	10 (0Ah)
ServiceEvents	MLIDSUP_SERVICE_EVENTS	11 (0Bh)
Reserved	RESERVED	12 (0Ch)
Reserved	RESERVED	13 (0Dh)

SendComplete	MLIDSUP_SEND_COMPLETE	14 (0Eh)
AddProtocolID	MLIDSUP_ADD_PID	15 (0Fh)
GetStackECB	MLIDSUP_GET_STACK_ECB	16 (10h)
ControlStackFilter	MLIDSUP_CONTROL_STACK_FILTER	17 (11h)

## AddProtocolID

**Description** Allows an MLID to register a Protocol ID for a given frame type and protocol stack combination

---

**Entry State**

*AX*

has the frame type ID to which the new Protocol ID applies (for example, the frame type ID of ETHERNET\_II is 2). See *ODI Specification Supplement: Frame Types and Protocol IDs* for a list of defined frame type IDs and commonly used Protocol IDs.

*BX*

is equal to *MLIDSUP\_ADD\_PID* (15 [0Fh]).

*CX:DI*

has a pointer to a length-preceded, zero-terminated string containing the protocol's short name to which the new Protocol ID applies.

*ES:SI*

is a pointer to a 6-byte Protocol ID to register.

Interrupts

are enabled.

**Return State**

*AX*

has a completion code.

Flags

Z flag set according to *AX*.

Interrupts

are enabled.

*DS, BP, SS, SP*

are preserved.

**Completion Codes (AX)**

*LSL\_SUCCESSFUL* (0000h)

The specified Protocol ID was successfully registered with the LSL.

*LSLERR\_OUT\_OF\_RESOURCES* (8001h)

The LSL has no resources to register another Protocol ID for the specified frame type.

*LSLERR\_BAD\_PARAMETER* (8002h)

The length of the specified protocol short name is either equal to 0 or is greater than 15.

*LSLERR\_DUPLICATE\_ENTRY* (8009h)

A Protocol ID for the specified protocol stack and frame type has already been registered with the LSL.

---

**Remarks**

This function is called by the MLID to add Protocol IDs specified in the NET.CFG file. The "Protocol x y" keyword with x equal to the Protocol ID and y equal to the frame type.



## ControlStackFilter

<b>Description</b>	Called by the MLID to pass a protocol control function call to each of the protocol stacks registered with a particular board number.
<b>Entry State</b>	<p><i>AX</i> has the protocol control function.</p> <p><i>BX</i> is equal to <i>MLIDSUP_CONTROL_STACK_FILTER</i> (17 [11h]).</p> <p><i>CX</i> has the protocol control parameters (passed on to each protocol stack unchanged).</p> <p><i>DX</i> has the <i>StkChnMask</i> filter bits.</p> <p><i>ES:SI</i> has the parameter buffer (optionally), which is passed on to each protocol stack unchanged.</p> <p><i>BP</i> has the logical board number.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p><i>BX</i> has the protocol control function.</p> <p><i>DI</i> is destroyed.</p> <p>Flags set according to <i>AX</i>.</p> <p>Interrupts are unchanged.</p> <p>All other registers are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The requested operation was completed successfully.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h) Board number is bad.</p>

---

The LSL calls all of the protocol stacks that have registered with the calling MLID board number and have a bit matching in *StkChnMask*. If the MLID wishes to send the control function to all of the registered protocol stacks, a '-1' is passed in for the *StkChnMask* bits. No return codes are available from

the called stacks. Each call is made assuming that the parameters for the control function are correct.

## DefragmentECB

### Description

Copies a fragmented ECB into a continuous ECB.

---

### Entry State

*AX*

has an offset beyond the end of the destination ECB to where the ECB data will start to be copied. If *AX* is  $-1$  (0FFFFh), the ECB itself will not be copied, but the data pointed to by the source ECB will be copied into the destination buffer.

*BX*

is equal to *MLIDSUP\_DEFRAG\_ECB* (2).

*ES:SI*

is a pointer to the data destination.

*CX:DI*

is a pointer to the source ECB.

Interrupts

are unspecified.

### Return State

*AX*

is always equal to *LSL\_SUCCESSFUL* (0000h).

Flags

are set according to *AX*.

Interrupts

are preserved.

*DS, ES, BP, SI, SS, SP*

are preserved.

---

### Remarks

This function is used to defragment an ECB into a contiguous ECB. This function can be used to move only the data pointed to by the ECB data buffer pointers to a contiguous buffer by setting *AX* equal to  $-1$  (0FFFFh). Setting *AX* to a value other than  $-1$  will cause the ECB to be copied as well as the data pointed to by the ECB. The value in *AX* will be used to reserve space between the ECB and the start of the first data fragment. It is assumed that the data buffer pointed to by *ES:SI* is large enough to hold the source ECB, data, and the offset that will be introduced by *AX*. If *AX* is equal to  $-1$ , the destination buffer needs to be only as large as the source ECB.*DataLength* value.

---

## DeregisterMLID

<b>Description</b>	Deregisters an MLID from the LSL.
<b>Entry State</b>	<p><i>AX</i> has the board number to deregister.</p> <p><i>BX</i> is equal to <i>MLIDSUP_DEREGISTER_MLID</i> (6).</p> <p>Interrupts are enabled.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p>Interrupts are enabled.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The MLID was successfully deregistered.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h) The board number is invalid.</p>
<b>Remarks</b>	<p>The MLID should clear its send queue before calling this procedure.</p> <p>The LSL will notify all protocols still bound to the board number the MLID deregistered.</p>

## EndCriticalSection

<b>Description</b>	Ends the critical section started by an MLID.
<b>Entry State</b>	<p><i>BX</i> is equal to <i>MLIDSUP_END_CRITICAL_SECTION</i>(9).</p> <p>Interrupts are disabled.</p>
<b>Return State</b>	<p>Interrupts disabled but might have been enabled during the routine.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p> <p>All other registers are destroyed.</p>
<b>Remarks</b>	<p>This function is called by MLIDs that have previously started a critical section by a call to <i>StartCriticalSection</i>. <i>StartCriticalSection</i> should always be used in conjunction with a subsequent call to <i>EndCriticalSection</i>.</p>

## GetCriticalSectionStatus

<b>Description</b>	Returns the number of critical sections that are currently active.
<b>Entry State</b>	<i>BX</i> is equal to <i>MLIDSUP_GET_CRITICAL_SECTION_STATUS</i> (10 [0Ah]). Interrupts are unspecified.
<b>Return State</b>	<i>BX</i> has the total number of outstanding calls to <i>StartCriticalSection</i> . Interrupts are unchanged. All other registers are preserved.
<b>Remarks</b>	MLIDs use this routine to avoid waiting for completion of events—for example, an AES that has been posted while inside a critical section.

## GetECB

### Description

Allocates one of the LSL's ECB buffers.

---

### Entry State

*BX*

is equal to *MLIDSUP\_GET\_ECB* (0).

Interrupts

are disabled.

### Return State

*AX*

has a completion code.

*ES:SI*

is a pointer to the ECB if *AX* is *LSL\_SUCCESSFUL* (0000h).

*DS, BP, SS, SP*

are preserved.

### Completion Codes (AX)

*LSL\_SUCCESSFUL* (0000h)

The ECB was allocated successfully.

*LSLERR\_OUT\_OF\_RESOURCES* (8001h)

Out of ECBs.

---

### Remarks

MLIDs call *GetStackECB* to obtain ECBs to pass receive information up. This function is provided for completeness in the specification.

When you call *GetECB*, it tries to obtain an ECB from the LSL's pool of ECBs. The end user must place "BUFFERS xx yy" under the link support section heading in the NET.CFG file. ECBs obtained from the LSL should be returned with the *ReturnECB* function. If the LSL was configured with no ECBs or there are no ECBs left in the pool, this function returns *LSLERR\_OUT\_OF\_RESOURCES* (8001h).

MLIDs that obtain ECBs from the LSL are not passed on to the system. A *LookAhead* structure and a call to *GetStackECB* must be made to pass a receive packet to the receiving protocol.

## GetIntervalMarker

**Description** Returns a timing marker in milliseconds. The timing marker is used for computer independent time measurements.

---

**Entry State** *BX*  
is equal to *MLIDSUP\_GET\_INTERVAL\_MARKER* (5).  
Interrupts  
are unspecified.

**Return State** *DX:AX*  
has the current dword interval time in milliseconds.  
Interrupts  
state is unchanged; interrupts are not enabled inside the routine.  
All other registers are preserved.

---

**Remarks** The actual value returned has no relation to any real-world absolute time. However, when time marker values are compared with each other, the difference is elapsed time in milliseconds.

On an IBM compatible computers, the marker is incremented every 1/18th of a second (55 milliseconds). On a Japanese Fujitsu (FMR) computer, the marker is incremented every 10 milliseconds. On a Japanese NEC or compatible computer, the marker is incremented every 33 milliseconds.

## GetStackECB

**Description** This routine is called to obtain a communications buffer.

---

**Entry State**

*BX*  
is equal to *MLIDSUP\_GET\_STACK\_ECB* (16 [10h]).

*DS:DI*  
has a far pointer to the *LookAheadStruc* structure.

Interrupts  
are disabled.

**Return State**

*AX*  
has a completion code.

*ES:SI*  
has a far pointer to an ECB if *AX* equals *LSL\_SUCCESSFUL* (0000h).

Flags  
Z flag set according to *AX*.

Interrupts  
are disabled.

*DS, BP, DI, SS, SP*  
are preserved.

**Completion Codes (AX)**

*LSL\_SUCCESSFUL* (0000h)  
The function was successful, and *ES:SI* points to a valid ECB.

*LSLERR\_OUT\_OF\_RESOURCES* (8001h)  
The LSL was unable to obtain an ECB for this packet.

---

**Remarks**

A far pointer to the *LookAheadStruc* structure will be passed to this call from a protocol stack. (For more information on the *LookAheadStruc* structure, see *Chapter 5: Protocol Stack Packet Reception*.)

### LookAheadStruc Structure

LookAheadStruc	struc			
LMediaHeaderPtr	dw	2	dup (?)	
LookAheadPtr	dw	2	dup (?)	
LookAheadLen	dw	?		
LProtID	db	6	dup (?)	
LBoardNum	dw	0		
LDataSize	dw	?		
LImmAddress	db	6	dup (?)	
LPacketAttrib	dw	0		
LDestType	dw	0		
LStartCopyOffset	dw	0		

```
LPriorityLevel      db    0
LRESERVED           db    3  dup (?)
LookAheadStruc     ends
```

## HoldReceiveEvent

**Description** Places an ECB on the LSL's event hold queue.

---

**Entry State**

*BX*  
is equal to *MLIDSUP\_HOLD\_RECV\_EVENT* (7).

*ES:SI*  
is a pointer to a completed received ECB.

Interrupts  
are disabled.

**Return State**

Interrupts  
are unchanged.

*DS, BP, SS, SP*  
are preserved.

---

**Remarks**

ECBs are placed on the event hold queue. The ECB's ESR routines are called by the LSL during a call to *ServiceEvents*.

This function is called when an ECB is filled in by the MLID.

---

## ReturnECB

**Description** Return's an LSL provided ECB.

---

**Entry State**

*BX*

is equal to *MLIDSUP\_RETURN\_ECB* (1).

*ES:SI*

is a pointer to the ECB to be returned.

Interprets

are disabled.

**Return State**

*AX*

has a completion code.

Interrupts

are disabled.

*DS, BP, SS, SP*

are preserved.

**Completion Codes (AX)**

*LSL\_SUCCESSFUL* (0000h)

The ECB was returned successfully.

*LSLERR\_BAD\_PARAMETER* (8002h)

The ECB is not the LSL's.

---

**Remarks**

This function is called to return an ECB obtained from the LSL *GetECB* function. ECBs provided by other modules should not be passed to this function.

**Note** This function is not available if the LSL is not providing ECBs.

## ScheduleAESEvent

<b>Description</b>	Schedules an asynchronous event scheduler (AES) event.
<b>Entry State</b>	<p><i>BX</i> is equal to <i>MLIDSUP_SCHEDULE_AES_EVENT</i> (3).</p> <p><i>ES:SI</i> has a pointer to an <i>AESECB</i> structure. ESR must be valid.</p> <p>Interrupts are unspecified.</p>
<b>Return State</b>	<p><i>AX</i> is always equal to <i>LSL_SUCCESSFUL</i> (0000h).</p> <p>Flags Z flag set according to <i>AX</i>.</p> <p>Interrupts state is unchanged; interrupts are not enabled inside this routine.</p> <p><i>DS, BP, ES, SI, SS, SP</i> are preserved.</p>

---

### Remarks

#### AESECB Structure

```
AESECB    struc
    AESLink    dd    ?
    MSecondValue dd    ?
    AESStatus  dw    ?
    AESESR    dd    ?
AESECB    ends
```

##### *AESLink*

This field is used by the LSL for list management.

##### *MSecondValue*

This field specifies the number of milliseconds to wait before invoking the defined *AESESR* routine. This field must be initialized each time the ECB is passed to *ScheduleAESEvent*.

##### *AESStatus*

This field is set to 0 when the *AESESR* is invoked.

##### *AESESR*

This field specifies a routine that is to be invoked when the specified time has expired. This field must point to a valid routine and only needs to be initialized once. The ESR must complete quickly because it is executing in the context of a timer interrupt.

The ESR can reschedule itself after it resets the *MSecondValue*, thus creating a simple polling function.

An ECB that is already in use by the LSL AES system must not be passed again to *ScheduleAESEvent*. To reset the AES event time, use *CancelAESEvent* and then issue a new *ScheduleAESEvent* call.

## ServiceEvents

**Description** This routine is invoked to complete the processing of network events queued on the LSL's holding queue. The ECBs in the queue are processed in the same order that they arrived in the queue (FIFO).

---

**Entry State** *BX*  
is equal to *MLIDSUP\_SERVICE\_EVENTS* (11 [0Bh]).

Interrupts  
are unspecified.

**Return State** Interrupts  
are disabled.  
*DS, BP, SS, SP*  
are preserved.

---

**Remarks** An MLID's ISR calls *ServiceEvents* immediately before the ISR restores the registers. The MLID should have completed all hardware processing, and the ISR should be ready to accept a new interrupt.

The *ServiceEvents* routine calls each queued ECB's ESR, to pass ECBs to the appropriate protocol stacks.

While inside this routine, interrupts might be enabled and the MLID's send handler routine could be called.

 **Caution** If this routine is not called when queued events are outstanding, the events may never complete.

 **Caution** The MLID must not be in a critical section, or have any internal lock-out semaphores set, when this routine is called because this routine could call a protocol ESR which could issue a transmit request to the MLID and then wait for the transmit request to finish. If the MLID postpones the transmit request, the machine will deadlock.

## StartCriticalSection

**Description** Starts a critical section.

---

**Entry State** *BX*  
is equal to *MLIDSUP\_START\_CRITICAL\_SECTION* (8).  
Interrupts  
are disabled.

**Return State** *BX*  
is destroyed.  
Interrupts  
are unchanged.  
All other registers and direction flags are preserved.

---

**Remarks** Whenever this function is called, *EndCriticalSection* must also be called after the reason for the critical section no longer exists.





*Chapter 12* **LSL Initialization Routines**

## Overview

This chapter describes the LSL initialization routines.

The following is an alphabetical list of the LSL initialization routines along with their entry point and function number.

<b>Descriptive Name</b>	<b>Function Name</b>	<b>Funct. No.</b>
GetEntryPoints	LSLINIT_GET_ENTRY_POINTS	4
GetMLIDSupportEntry	LSLINIT_GET_MLID_ENTRY	3
GetProtocolSupportEntry	LSLINIT_GET_PROTSUP_ENTRY	2
RegisterMLID	LSLINIT_MLID_REG	1

The following is a function number ordered list the LSL initialization routines along with their entry point and function number.

<b>Descriptive Name</b>	<b>Function Name</b>	<b>Funct. No.</b>
RegisterMLID	LSLINIT_MLID_REG	1
GetProtocolSupportEntry	LSLINIT_GET_PROTSUP_ENTRY	2
GetMLIDSupportEntry	LSLINIT_GET_MLID_ENTRY	3
GetEntryPoints	LSLINIT_GET_ENTRY_POINTS	4

## GetEntryPoints

<b>Description</b>	Used to obtain all the LSL provided entry points.
<b>Entry State</b>	<p><i>BX</i> is equal to <i>LSLINIT_GET_ENTRY_POINTS</i> (4).</p> <p><i>ES:SI</i> has a buffer for a LSL initialization entry point block (<i>LSLInitEntryPointBlock</i>).</p>
<b>Return State</b>	<p><i>BX</i> is destroyed.</p> <p><i>ES:SI</i> has the LSL initialization entry point block buffer filled with the LSL support entry points.</p> <p>All other registers are preserved.</p>
<b>Remarks</b>	The functional equates that are used with each entry point are named according to the entry point: <i>PROTSUP_xxx</i> equates are used with the <i>LSLProtSupEntryPt</i> entry point, <i>GENSERV_xxx</i> equates are used with the <i>LSLGenSupEntryPt</i> , <i>MLIDSUP_xxx</i> equates are used with the <i>LSLMLIDSupEntryPt</i> entry point.

### LSLInitEntryPointBlock Structure

```

LSLInitEntryPointBlock    struc
    LSLProtSupEntryPt      dd 0
    LSLGenSupEntryPt       dd 0
    LSLMLIDSupEntryPt     dd 0
LSLInitEntryPointBlock    ends

```

*LSLProtSupEntryPt*  
This field has the protocol support entry point.

*LSLGenSupEntryPt*  
This field has the general services support entry point.

*LSLMLIDSupEntryPt*  
This field has the MLID support entry point.

## GetMLIDSupportEntry

<b>Description</b>	Obtains the MLID support entry point.
<b>Entry State</b>	<i>BX</i> is equal to <i>LSLINIT_GET_MLID_ENTRY</i> (3).
<b>Return State</b>	<i>BX</i> is destroyed. <i>ES:SI</i> has the MLID support entry point. All other registers are preserved.
<b>Remarks</b>	The value returned in <i>ES:SI</i> is a far pointer to the MLID support entry point. All of the functional equates whose names start with "MLIDSUP" use this entry point.

## GetProtocolSupportEntry

<b>Description</b>	Obtains the protocol stack support and general services entry points.
<b>Entry State</b>	<p><i>BX</i> is equal to <i>LSLINIT_GET_PROTSUP_ENTRY</i> (2).</p> <p><i>ES:SI</i> has a pointer to an 8-byte buffer.</p>
<b>Return State</b>	<p><i>BX</i> is destroyed.</p> <p><i>ES:SI</i> has a pointer to an 8-byte buffer that is filled in with a far pointer to the protocol stack and to the general services entry point.</p> <p>All other registers are preserved.</p>
<b>Remarks</b>	<p>If all three entry points (protocol stack, general services, and MLID) are needed, use <i>GetEntryPoints</i>.</p> <p>The 8-byte buffer is defined as follows:</p> <ul style="list-style-type: none"><li>• protocol support API entry point</li><li>• general services support API entry point</li></ul>

## RegisterMLID

<b>Description</b>	Registers an MLID with the LSL.
<hr/>	
<b>Entry State</b>	<p><i>BX</i> is equal to <i>LSLINIT_MLID_REG</i> (1).</p> <p><i>DS:DI</i> is a buffer supplied by the MLID for returning information (<i>MLIDRetInfoBlock</i>).</p> <p><i>ES:SI</i> is a pointer to the MLID information block (<i>MLIDInfoBlockStruc</i>).</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p><i>DS:DI</i> has a pointer to the returned information block (<i>MLIDRetInfoBlock</i>).</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The MLID is successfully registered.</p> <p><i>LSLERR_OUT_OF_RESOURCES</i> (8001h) Exceeded maximum number of boards supported in the LSL.</p>
<hr/>	

**Remarks**

### Information Block on Entry

```

MLIDInfoBlockStruc    struc
    MIBS_SendEntry      dd ?
    MIBS_ControlEntry   dd ?
    MIBS_ConfigTable    dd ?
MLIDInfoBlockStruc    ends
    
```

**Field descriptions:**

*MIBS\_SendEntry*

This contains a far pointer to the MLID's transmit entry point. When the LSL has a packet for the MLID to send, it will be sent through this address.

*MIBS\_ControlEntry*

This has a far pointer to the MLID's control entry point. See *Chapter 18: MLID Control Routines* for a description of the control functions.

*MIBS\_ConfigTable*

---

This contains a far pointer to the MLID's configuration table.

## Information Block on Return

```
MLIDRetInfoBlockStruc  struc
    MRIBS_MLID_SUP      dd ?
    MRIBS_BoardNum      dw ?
    MRIBS_ECB_DataSize  dw ?
MLIDRetInfoBlockStruc  ends
```

### Field descriptions:

#### *MRIBS\_MLID\_SUP*

This contains a far pointer to the LSL's MLID support entry point.

#### *MRIBS\_BoardNum*

This contains the LSL assigned board number for this logical board.

#### *MRIBS\_\_ECBDataSize*

This is the size of the ECB's provided by the LSL if the LSL is configured to provide them. If the LSL is not providing ECBs, this field is set to 1514.

For additional information, see *MLIDInitialization* in *Chapter 16: MLID Initialization*.

□





*Chapter 13* **LSL General Services**

## Overview

The LSL provides a number of services to all modules in a system. You can invoke the LSL's general services by calling the general service entry point that the protocol stack obtains when it locates the LSL. This interface also allows applications to register custom general services with the LSL.

**Note** General services provided by applications are documented by the providing application.

The following is an alphabetical list of the LSL general service routines along with their entry point and function number.

<b>Descriptive Name</b>	<b>Function Name</b>	<b>Funct. No.</b>
AddGeneralService	GENSERV_ADD_GENERAL_SERVICE	5
AddMemoryToPool	GENSERV_ADD_MEMORY_TO_POOL	4
AllocateMemory	GENSERV_ALLOC_MEMORY	0
FreeMemory	GENSERV_FREE_MEMORY	1
GetNETCFGPath	GENSERV_GET_NETCFG_PATH	7
GetServiceChain	GENSERV_GET_SERVICE_CHAIN	9
MemoryStatistics	GENSERV_MEMORY_STATISTICS	3
ReallocateMemory	GENSERV_REALLOC_MEMORY	2
RemoveGeneralService	GENSERV_REMOVE_GENERAL_SERVICE	6
Reserved	GENSERV_RESERVED	8

The following is a function number ordered list of the general service routines along with their entry point and function number.

<b>Descriptive Name</b>	<b>Function Name</b>	<b>Funct. No.</b>
AllocateMemory	GENSERV_ALLOC_MEMORY	0
FreeMemory	GENSERV_FREE_MEMORY	1
ReallocateMemory	GENSERV_REALLOC_MEMORY	2
MemoryStatistics	GENSERV_MEMORY_STATISTICS	3
AddMemoryToPool	GENSERV_ADD_MEMORY_TO_POOL	4
AddGeneralService	GENSERV_ADD_GENERAL_SERVICE	5
RemoveGeneralService	GENSERV_REMOVE_GENERAL_SERVICE	6
GetNETCFGPath	GENSERV_GET_NETCFG_PATH	7
Reserved	GENSERV_RESERVED	8
GetServiceChain	GENSERV_GET_SERVICE_CHAIN	9

## AddGeneralService

**Description** This routine allows protocol stacks, MLIDs, and applications to add new commands to the LSL's general services entry point.

**Entry State**

*BX* is equal to GENSERV\_ADD\_GENERAL\_SERVICE (5).

*ES:SI* has a pointer to a general service control block.

Interrupts are enabled.

**Return State**

*AX* has a completion code.

Flags Z flag is set according to AX.

Interrupts are enabled.

*DS, BP, SS, SP* are preserved.

**Completion Codes (AX)**

*LSL\_SUCCESSFUL* (0000h)  
The protocol stack was able to register a new command with the LSL.

*LSLERR\_DUPLICATE\_ENTRY* (8009h)  
An LSL general service with the requested command code already exists.

**Remarks**

The LSL owns the memory for the general service control block until the service is removed by the *RemoveGeneralService* call. The application should not change any of the fields after *AddGeneralService* is called.

The new command's entry will be called whenever the LSL's general service entry point is entered with a command code that matches the command code in the control block.

*AddGeneralService* is especially useful to enable a process to locate other pieces of itself. For example, a protocol stack could register itself to allow another piece of the protocol stack, which might not always be loaded, to find and communicate with the master piece of the protocol stack.

Before the protocol stack can add a new general service, the protocol stack must locate an available command code in the

range 8000h–FFFFh. To locate the available command code, the protocol stack calls the general service entry point with the desired command code in BX and AX equal to 0. If the command code is already in use, AX is returned still equal to 0, and ES:SI has a pointer to a description record (detailed below). The description record can be examined to determine what general service is installed for this command code. If the command code is not in use, AX will contain *LSLERR\_BAD\_COMMAND* (8008h).

New general services must support AX = 0 as an incoming parameter and must return AX = 0. In addition, ES:SI must point to the service’s description record described below. Optionally, DX:BX can contain an entry point for the service.

The general service description record is shown in the following table.

**Table 13.1 General Service Description Record**

Offset	Name	Size (in bytes)	Description
00h	GenServiceName	11	This field has the ASCII general service name. This name has no leading length byte.
0Bh	GenServiceMonth	1	This field contains the decimal value that matches the month the service’s executable code was generated (for example, 12 = December).
0Ch	GenServiceDay	1	This field contains the decimal value that matches the day of the month that the service’s executable code was generated.
0Dh	GenServiceYear	1	This field has the decimal value matching the year the service’s executable code was generated (for example, 91 = year 1991).
0Eh	MajorVersion	1	This field has the decimal major version of the general service.
0Fh	MinorVersion	1	This field contains the decimal minor version of the general service.

### GenServiceControlBlock Structure

```

GenServiceControlBlock  struc
GenServiceLink          dd      ?
GenServiceAddress       dd      ?
GenServiceCommand       dw      ?
GenServiceControlBlock  ends
    
```

#### *GenServiceLink*

This field is used by the LSL to manage a list of general service control blocks.

*GenServiceAddress*

This field contains the application's entry point that is invoked when the defined general service is invoked. This must be set by the application adding the new general service.

*GenServiceCommand*

This field has the general service command code that is used to invoke this function. The application must set this field before the application adds the new general service.

## AddMemoryToPool

**Description** Allows a protocol stack or Terminate and Stay Resident (TSR) application to give more memory to the buffer pool.

---

**Entry State**

*BX* is equal to *GENSERV\_ADD\_MEMORY\_TO\_POOL* (4).

*ES* has the segment address to add to the pool.

*CX* is the number of paragraphs to add to the pool.

Interrupts are enabled.

**Return State**

*AX* always returns *LSL\_SUCCESSFUL* (0000h).

Interrupts are enabled.

*DS, BP, SS, SP* are preserved.

---

**Remarks** Once memory is given to the pool, it cannot be removed.

## AllocateMemory

<b>Description</b>	Allocates a block of memory to the protocol stack.
<b>Entry State</b>	<p><i>BX</i> is equal to <i>GENSERV_ALLOC_MEMORY</i> (0).</p> <p><i>CX</i> is the number of bytes to allocate.</p> <p>Interrupts are enabled.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p><i>ES:SI</i> is a pointer to the allocated memory if <i>AX</i> equals <i>LSL_SUCCESSFUL</i> (0000h).</p> <p>Interrupts are enabled.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) Memory was available and <i>ES:SI</i> will point to the allocated memory.</p> <p><i>LSLERR_OUT_OF_RESOURCES</i> (8001h) The memory pool does not have enough memory to satisfy a request.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h) A request needs more memory than allowed. The maximum number is 65516 bytes.</p>

---

## FreeMemory

<b>Description</b>	Returns a piece of memory that was previously allocated with the <i>AllocateMemory</i> function.
<b>Entry State</b>	<i>BX</i> is equal to <i>GENSERV_FREE_MEMORY</i> (1). <i>ES:SI</i> is a pointer to the block of memory to free. Interrupts are enabled.
<b>Return State</b>	<i>AX</i> has a completion code. Interrupts are enabled. <i>DS, BP, SS, SP</i> are preserved.
<b>Completion Codes (AX)</b>	<i>LSL_SUCCESSFUL</i> (0000h) The memory was successfully returned to the pool. <i>LSLERR_BAD_PARAMETER</i> (8002h) The pointer returned did not come from the memory pool.

---

---

## GetNETCFGPath

<b>Description</b>	Returns a fully formed path specification to the NET.CFG file as found by the LSL when the LSL initialized.
<b>Entry State</b>	<p><i>BX</i></p> <p>is equal to <i>GENSERV_GET_NETCFG_PATH</i> (7 [7h]).</p> <p>Interrupts</p> <p>state is unspecified.</p>
<b>Return State</b>	<p><i>AX</i></p> <p>is always equal to <i>LSL_SUCCESSFUL</i> (0000h).</p> <p><i>DS:DX</i></p> <p>has a pointer to an ASCII path and configuration filename.</p> <p>Flags</p> <p>Z flag is set according to <i>AX</i>.</p> <p>Interrupts</p> <p>are unchanged.</p> <p><i>ES, CX, DI, SI, BP, SS, SP</i></p> <p>are preserved.</p>
<b>Remarks</b>	<p>If the LSL could not find a NET.CFG file when it loaded, the returned string will be "NET.CFG", which is adequate for attempting an open in the calling module's current working directory. LSL.COM searches for the NET.CFG in the following order:</p> <ol style="list-style-type: none"><li>1. The current working directory at the time the LSL was loaded.</li><li>2. The directory from which LSL.COM was loaded (possibly a search path).</li></ol> <p>Because all ODI modules use this function, the users are able to define one NET.CFG file that all modules will use as they are loaded.</p>

---

## GetServiceChain

**Description** Returns the start of the general services chain. The chain is of all general services that have been added to the LSL since it was loaded.

---

**Entry State** *BX*  
is equal to *GENSERV\_GET\_SERVICE\_CHAIN* (9).

**Return State** *AX*  
is equal to *LSL\_SUCCESSFUL* (0000h).

*ES:BX*  
Head of general services chain.

Interrupts  
are unchanged.

All other registers are preserved.

---

**Remarks** This function returns the start of a chain to scan for a free service number or to determine what a function is.

The chain and all values should be regarded as read-only.

## MemoryStatistics

**Description** Returns information about the LSL's memory system.

**Entry State**

*BX*

is equal to *GENSERV\_MEMORY\_STATISTICS* (3).

*ES:SI*

is a pointer to a 6-word block that information will be returned in (*MemStatStruc*).

Interrupts

are unspecified.

**Return State**

*AX*

is equal to *LSL\_SUCCESSFUL* (0000h).

*ES:SI*

is a pointer to a *MemStatStruc* block.

Interrupts

are unchanged.

All other registers are preserved.

**Remarks**

### MemStatStruc Structure

```
MemStatStruc    struc
    MemAvail      dw    0
    MemInUse      dw    0
    LargestAvailBlk  dw    0
    NumAvailBlocks  dw    0
    MemOverhead   dw    0
    MinAllocation  dw    0
MemStatStruc    ends
```

*MemAvail*

the number of paragraphs of memory available.

*MemInUse*

the number of paragraphs of memory in use.

*LargestAvailBlk*

the number of paragraphs in the largest block of memory.

*NumAvailBlocks*

the number of available blocks of memory.

*MemOverhead*

the number of bytes overhead per allocation.

*MinAllocation*

the number of bytes minimum allocation.

## ReallocateMemory

<b>Description</b>	Allows reduction of the size of the allocated memory block, returning some of the memory to the pool.
<b>Entry State</b>	<p><i>BX</i> is equal to <i>GENSERV_REALLOC_MEMORY</i> (2).</p> <p><i>ES:SI</i> has a pointer to the memory block to be resized.</p> <p><i>CX</i> is the requested new memory block size; it must be less than the original memory block size. If <i>CX</i> is equal to 0FFFFh, the memory size of the block is returned.</p> <p>Interrupts are enabled.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p><i>CX</i> is the size of the memory block.</p> <p><i>ES:SI</i> has a pointer to the resized memory block.</p> <p>Interrupts are enabled.</p> <p><i>DS, SS, SP, BP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The sizing was completed and/or the the size was returned.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h) More memory than was in the original block of memory is requested.</p>

---

## RemoveGeneralService

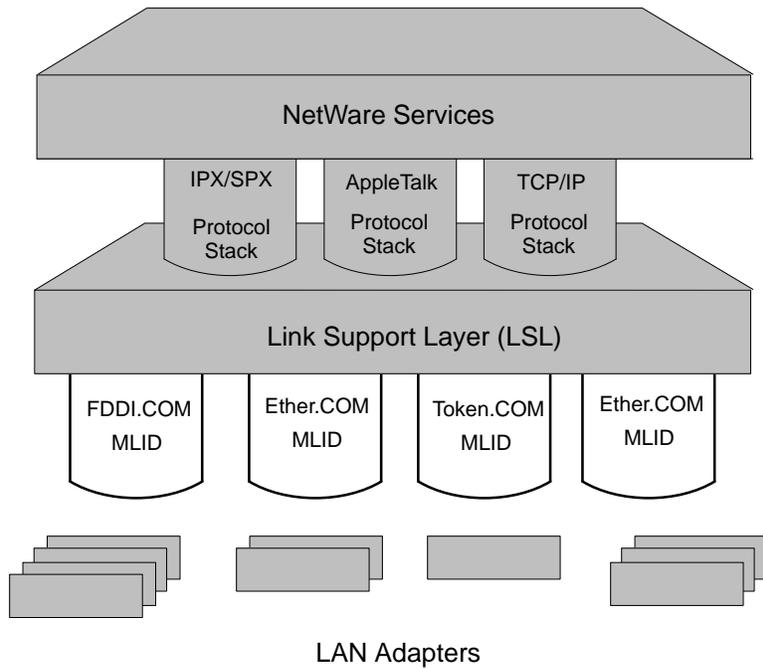
<b>Description</b>	Removes a general service that was previously added using <i>AddGeneralService</i> .
<b>Entry State</b>	<p><i>BX</i> is equal to <i>GENSERV_REMOVE_GENERAL_SERVICE</i> (6).</p> <p><i>ES:SI</i> has a pointer to a general service control block described in the <i>AddGeneralService</i> definition; this must be the same general service control block passed to the <i>AddGeneralService</i> and not a copy of it.</p> <p>Interrupts are enabled.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p><i>ES:SI</i> has a pointer to the general services control block that was returned.</p> <p>Flags Z flag is set according to <i>AX</i>.</p> <p>Interrupts are enabled.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The general service was removed.</p> <p><i>LSLERR_ITEM_NOT_PRESENT</i> (8004h) The general service could not be found.</p>
<b>Remarks</b>	This function is invoked to remove a general service that the caller previously installed. Prior to exiting the system, all added general services should be removed by the providing module.







Section IV **Multiple Link Interface Drivers (MLIDs)**



## Section Overview

This section introduces Multiple Interface Link Drivers (MLIDs) and describes MLID initialization, transmission, and reception routines. This section also provides chapters describing MLID control routines.

*Chapter 14: Overview of the MLID* describes the procedures and functions that the MLID should provide. You should read this chapter if you have never written an ODI MLID before.

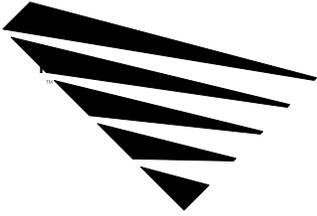
*Chapter 15: MLID Data Structures* describes the data structures and variables that the MLID must define. This chapter contains useful reference material for the developer.

*Chapter 16: MLID Initialization* describes the steps required to initialize an MLID. You should review this chapter before writing the MLID initialization routine.

*Chapter 17: MLID Packet Reception and Transmission* describes the packet reception and transmission methods available to the developer. You should review this chapter before writing the MLID packet reception and/or transmission routine.

*Chapter 18: MLID Control Routines* describes the control procedures that an MLID is required to provide.





*Chapter 14* **Overview of the MLID**

## Overview

This chapter describes the procedures and functionality that the MLID provides. However, depending on the hardware and supported topology of your LAN adapter, your MLID might not need to meet all of the requirements discussed in this chapter.

You should read this chapter if you have never written an ODI MLID before.

**Note** As an alternative to writing a complete MLID, you may want to write a Hardware Specific Module (HSM) by getting the *LAN Driver Developer's Kit* toolkit. This toolkit provides most of the code for the MLID and only requires you to write the HSM, which is easier to do than writing a complete MLID.

## ODI MLID

MLIDs handle the sending and receiving of packets on the network. MLIDs drive a LAN adapter (also referred to as Network Interface Card or NIC) and handle frame header appending and stripping. They also determine the packet's frame type.

The requirements of your LAN adapter dictate how you write your MLID.

## MLID Procedures

The ODI specification defines the following procedures:

- MLID initialization routine (required)
- Board service routine (one or both are required)
  - Interrupt Service Routine (ISR)
  - Driver polling routine
- Packet transmission routine (required)

The MLID also supports the following control procedures:

- Control procedures for ODI IOCTLs
  - AddMulticastAddress (required if hardware supports multicast addressing)
  - DeleteMulticastAddress (required if hardware supports multicast addressing)
  - GetMLIDConfiguration (required)
  - GetMLIDStatistics (required)
  - PromiscuousChange (recommended)
  - SetLookAheadSize (required)
  - RegisterTxMonitor (required)
  - DriverManagement (optional)
  - DriverPoll (optional)
  - MLIDReset (required)
  - MLIDShutdown (required)
- Timeout detection
  - Interrupt call back routine (optional)
  - AES call back routine (optional)
- MLID removal routine (required)

The specific hardware requirements of a LAN adapter can require that you write additional procedures; however, the procedures listed above represent the generic code elements found in every MLID.

A brief description of each procedure is presented throughout the rest of this chapter. These descriptions are high-level generalizations only and are not true in every case, nor do they describe every possible case or the optimal algorithm for implementation.

## MLID Initialization

In general terms, the MLID's initialization routine must perform the following actions:

- Allocate memory for the MLID's variables and structures.
- Parse the standard command line options.
- Parse NET.CFG entries for the MLID.
- Process custom command line parameters and custom firmware.
- Register the MLID with the LSL.
- Provide a hook for the MLID's board service routine by allocating an interrupt or by establishing a polling procedure.
- Schedule callback events for timeout detection and recovery.
- Initialize the LAN adapter.

## Board Service Routine

The board service routine generally needs to detect and handle the following events on the LAN adapter:

- Received packet
- Packet reception error
- Completed transmission
- Packet transmission error

The MLID can be notified of these events by an interrupt service routine (ISR), a polling procedure, or a polling procedure with interrupt backup.

## Packet Transmission

The MLID's packet transmission routine is called whenever a packet needs to be transmitted onto the wire. The MLID must build the necessary frame and media headers and then send the packet. The MLID can implement priority packet transmission if applicable.

## Control Routines

Among the control procedures that the MLID must provide are control procedures to support multicast addressing (if the

hardware supports it) and procedures to reset and shut down the hardware. The MLID can also supply a control procedure to support promiscuous mode.

MLIDs might also implement the driver management support routine.

## Timeout Detection

The MLID can schedule an AES event that it uses at specified intervals. For example, the MLID might need to be called regularly to inspect the LAN adapter and determine if the adapter has failed to complete a transmission. If a timeout error had occurred, the procedure discards the packet being sent, resets the board, and begins transmitting the next packet in the send queue.

## Driver Remove

Every MLID must have a remove procedure that allows the user to unload the MLID from the operating system. This procedure must shut down the LAN adapter and return any resources that the MLID allocated from the operating system.

## Events

The MLID should be implemented so it is NESL (NetWare Event Service Layer Event) compliant. See *NESL Specification: 16-Bit DOS Client Programmer's Interface* for more information.

## MLID Data Structures and Variables

In addition to the procedures discussed above, the MLID must also contain certain data structures and variables. The following are the primary structures:

- MLID configuration table
- MLID statistics table

### MLID Configuration Table

The MLID configuration table is a data structure that defines the configuration of the LAN adapter and MLID. The fields in this table are primarily used during initialization and are referred to by the LSL and MLID. The requirements for MLID configuration tables are described in detail in *Chapter 15: MLID Data Structures*

### MLID Statistics Table

The MLID statistics table is a data structure that contains data on the operation of the LAN adapter and the MLID. *Chapter*

15: *MLID Data Structures* contains a detailed description of this data structure.

## MLID Functionality

The MLID should include the following support if possible:

- multiple frame
- source routing
- promiscuous mode
- multicast addressing (required if the adapter can support multicast addressing)

**Note** In some instances this specification makes recommendations on how to implement certain functionality, but these are only recommendations, and it is up to you to implement the functionality correctly.

### Multiple Frame Support

If the LAN adapter runs on a topology that supports multiple frame types, the MLID should support all the frame types for that particular topology. You can implement multiple frame support by using logical boards.

#### Multiple Frame Support and Logical Boards

Your MLID creates a “logical board” for each frame type configured. Each logical board looks like a separate physical board to the other modules in the system. The MLID should provide this feature.

#### Adapter Data Space

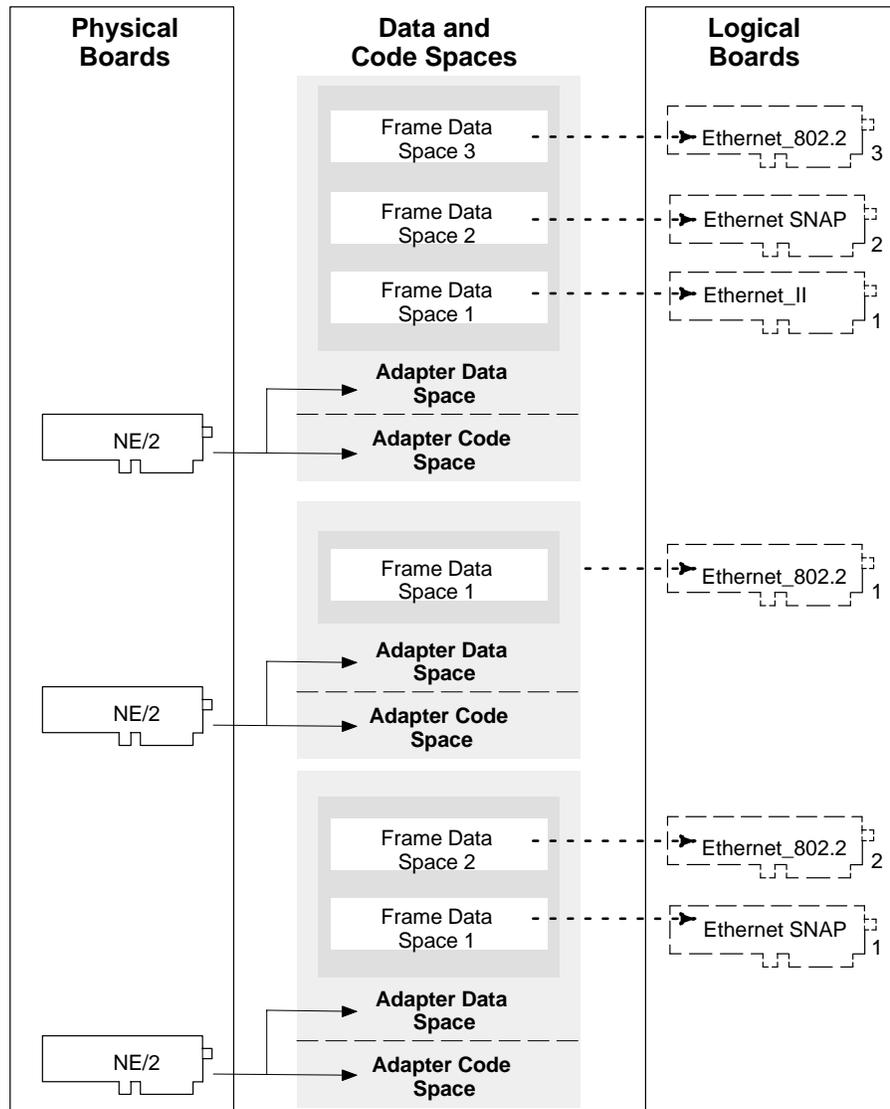
The adapter data space contains the hardware specific information that the MLID needs to drive the LAN adapter (interrupt number, beginning memory address, etc.). The statistics table required by the ODI specification is contained in the adapter data space. The MLID allocates one adapter data space for each LAN adapter, regardless of the number of logical boards (frame types) it supports.

#### Frame Data Space

Every logical board has a frame data space associated with it. The frame data space contains the frame-specific information the MLID needs to support that frame type. The MLID allocates a frame data space for each logical board. The MLID then copies the configuration table template for that logical board into its frame data space.

**Note** The MLID must create a frame data space for every frame type that is loaded.

**Figure 14.1**  
Implementation of  
Multiple Frame Support  
in Ethernet Topology



## Source Routing Support

The *ODI Specification Supplement: Source Routing* describes how to add and configure source routing in the MLID. If your LAN adapter is capable of supporting source routing—for example, Token-Ring and FDDI topologies, it should do so.

## Promiscuous Mode Support

When MLIDs operate in promiscuous mode, they pass all of the packets they receive to the upper layers. This includes bad packets, if possible. Because various monitoring functions

operate in promiscuous mode, your MLID should support promiscuous mode if your adapter is capable of such support.

## Multicast Addressing Support

If your LAN adapter is capable of supporting multicast addressing, your MLID must support it. The *AddMulticastAddress* and *DeleteMulticastAddress* IOCTLs implement multicast support. These control procedures are discussed in more detail in *Chapter 18: MLID Control Routines*.

## MLID Design Considerations

The following section discusses hardware and coding issues you must consider when developing the MLID.

### Hardware Issues

Every type of LAN adapter, such as the NE1000 and NE2000, has different hardware and data transfer characteristics. A thorough understanding of your LAN adapter and LAN topology will allow you to create a more efficient driver. Keep in mind that the board and chip manufacturer's support engineers can provide you with up-to-date information regarding their hardware.

#### Data Transfer Mode

The LAN adapter's mode of data transfer is a primary consideration in MLID development. To achieve the highest performance, you must select support procedures matched to the data transfer mode. The data transfer modes are:

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

#### Bus Type

You must also consider the LAN adapter's bus type and size. The following are common bus types:

- Industry Standard Architecture (ISA)
- Micro Channel Architecture
- Extended Industry Standard Architecture (EISA)
- Personal Computer Memory Card International Association (PCMCIA)
- Peripheral Component Interconnect (PCI)

□



*Chapter 15* **MLID Data Structures**

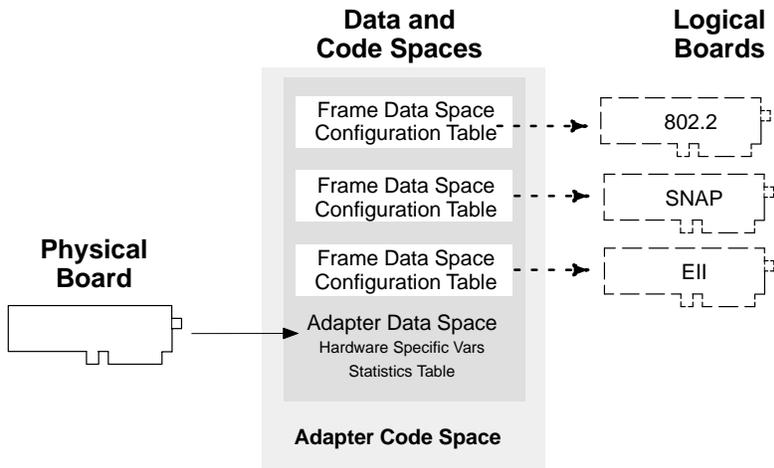
## **Overview**

This chapter describes the data structures and variables that the MLID must define. All the data structures defined in this chapter must be present in the CGroup segment of the MLID.

## Frame Data Space

The ODI specification requires that every MLID have a configuration table as part of the frame data space. The MLID keeps a copy of the configuration table template in the CGroup segment. The MLID uses the configuration table in the CGroup segment as the working configuration table for the default logical board and as a template for the configuration tables it must copy for each loaded logical board. When the MLID allocates the frame data space for each logical board (frame type) that loads, it copies the configuration table template for that logical board into that logical board's frame data space. Because external processes can also access this table, the ODI specification defines this table's format strictly.

**Figure 15.1**  
Frame and Adapter  
Data Space



## MLID Configuration Table

The MLID configuration table contains information about the MLID and its configuration. The MLID must define one configuration table for each logical board number assigned by the LSL. Variables in this structure include the interrupt number, port I/O address, node address, and other MLID specific parameters.

The MLID must define the configuration table to contain the LAN adapter's default configuration and any other information about that configuration. The table must be defined by the fields described in this chapter with each entry filled in accordingly. Certain variables in the configuration table are specific to your MLID. Other variables are specific to the LAN adapter the MLID is running.

**Note** All data strings in the configuration table are length preceded and NULL terminated. These strings consist of a one-byte

length (not counting the length byte or NULL), the data string itself, and a terminating 0 (NULL) byte.



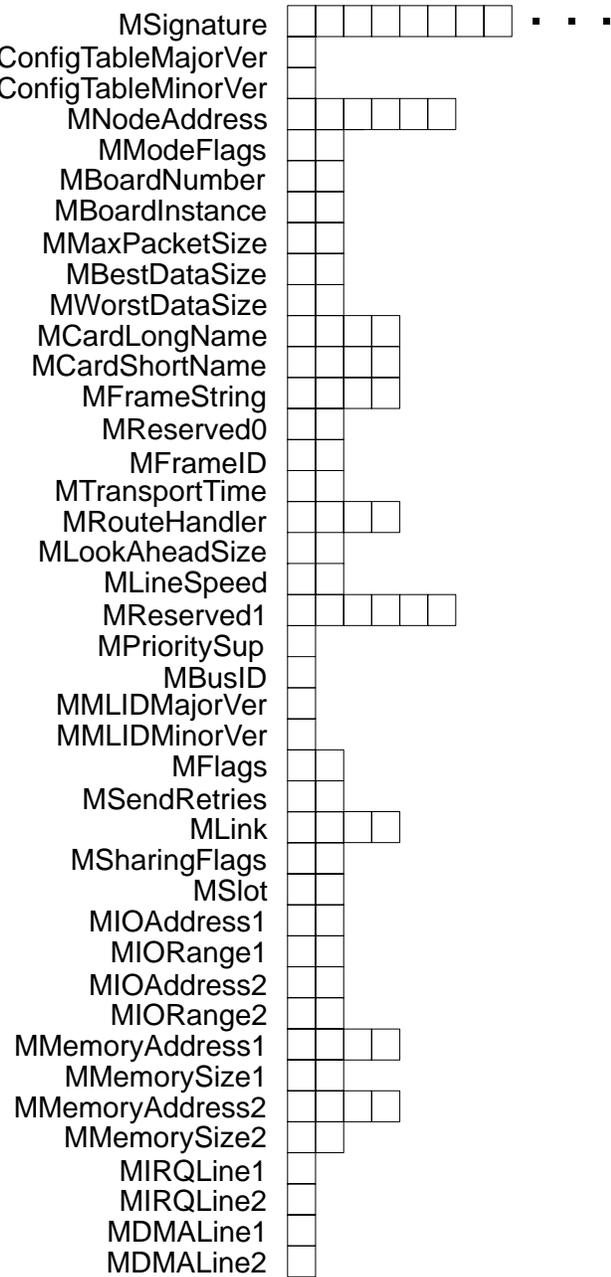
**Important** A protocol stack treats the MLID configuration table as read-only!

### MLID Configuration Table Sample Code

```

MLIDConfigurationStructure  struc
    MSignature                db  'HardwareDriverMLID',8 dup ( ' ')
    MConfigTableMajorVer     db   01
    MConfigTableMinorVer     db   13
    MNodeAddress              db    6 dup (?)
    MModeFlags                dw  81h
    MBoardNumber              dw    ?
    MBoardInstance           dw    ?
    MMaxPacketSize           dw    0
    MBestDataSize            dw    0
    MWorstDataSize          dw    0
    MCardLongName            dd    0
    MCardShortName          dd    0
    MFrameString             dd    0
    MReserved0               dw    0
    MFrameID                 dw    0
    MTransportTime           dw    1
    MRouteHandler            dd    0
    MLookAheadSize           dw   18
    MLineSpeed               dw    0
    MReserved1               db    6 dup (0)
    MPrioritySup              db    0
    MBusID                   db   -1
    MMLIDMajorVer            db    0
    MMLIDMinorVer           db    0
    MFlags                   dw    0
    MSendRetries             dw    0
    MLink                    dd    0
    MSharingFlags            dw    1
    MSlot                    dw    0
    MIOAddress1              dw    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
    MIRQLine2                db  0FFh
    MDMALine1                db  0FFh
    MDMALine2                db  0FFh
MLIDConfigurationStructure  ends
    
```

**Figure 15.2**  
**Graphic Representation of the**  
**MLID Configuration Table**



**Table 15.1 MLID Configuration Table Field Descriptions**

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.
1Ch	MNodeAddress	6	This field holds the card's node address. The MLID sets this field during the initialization routine. (See <i>ODI Specification Supplement: Canonical and Noncanonical Addressing</i> for more information.)
22h	MModeFlags	2	This field contains flags, which are defined in Table 15.2.
24h	MBoardNumber	2	During initialization, the MLID sets this field to the board number returned by <i>RegisterMLID</i> .
26h	MBoardInstance	2	The MLID sets this field to the logical board instance for the MLID. There is a <i>BoardInstance</i> number (zero based) for each logical board. With Each physical board, the <i>BoardInstance</i> number for the logical boards restarts at 0.  For example, you have two physical boards with two frame types loaded on each of them. The <i>BoardInstance</i> number for the first logical board on each of the physical adapters is 0; the <i>BoardInstance</i> for each of the second logical boards on each physical adapter is 1.
28h	MMaxPacketSize	2	This field defines the largest possible packet size that the MLID and LAN adapter combination can transmit and/or receive. This value includes all headers. Typically, Ethernet MLIDs 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 MLID must determine the appropriate packet size during initialization and place that value in this field.  Token-Ring MLIDs 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).

**Table 15.1 MLID Configuration Table Field Descriptions** (continued)

Offset	Name	Size (in bytes)	Description
2Ah	MBestDataSize	2	The MLID sets this field during initialization. The MLID subtracts the length of the <i>smallest</i> media header(s) from the value in the <i>MMaxPacketSize</i> field.  For example, an Ethernet MLID sets this field to 1500 decimal ( $1514 - 14 [\text{MAC}] = 1500$ ) if the MLID runs the Ethernet_II packet type. A Token-Ring MLID sets this field to $MMaxPacketSize - 14 [\text{MAC}] - 3 [802.2 \text{ UI}]$ if the MLID's packet type is Token-Ring.
2Ch	MWorstDataSize	2	The MLID sets this field during initialization. The MLID subtracts the length of the largest media headers(s) from the <i>MMaxPacketSize</i> field.  For example, a Token-Ring MLID sets this field to $MMaxPacketSize - 14 [\text{MAC}] - 3 [802.2 \text{ UI}] - 30 [\text{source routing}] - 5 [\text{SNAP}]$ if the MLID's packet type is Token-Ring SNAP. An Ethernet_II MLID sets this field to 1500 ( $MMaxPacketSize - 14 [\text{MAC}] = 1500$ ). 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.
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, upper- and lower-case A–Z, and the underscore.  We recommend that this string contain the MLID'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 <i>ODI Specification Supplement: Frame Types and Protocol IDs</i> .)
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 <i>ODI Specification Supplement: Frame Types and Protocol IDs</i> .)
3Eh	MTransportTime	2	The MLID sets this field to define the number of milliseconds it takes the MLID and LAN adapter to transmit a 512-byte data packet.  The MLID cannot set this field to 0. Most MLIDs set this field to a value of 1.  If the MLID drives a board on a slow asynchronous line, it sets this field according to a representative value.

**Table 15.1 MLID Configuration Table Field Descriptions** (continued)

Offset	Name	Size (in bytes)	Description															
40h	MRouteHandler	4	MLIDs that support source routing use this field in conjunction with ROUTE.COM. MLIDs initialize this field to 0 and then do not modify it. (See the <i>ODI Specification Supplement: Source Routing</i> for a discussion of source routing.)															
44h	MLookAheadSize	2	This field holds the configured look ahead size as set by protocol stacks. The MLID initializes this field to a default value of 18 bytes.  When it receives a packet, the MLID uses this value and the maximum possible media header to determine the amount of look ahead data it must pass to the <i>GetStackECB</i> routine. The value in this field can be changed at any time. Therefore, the MLID must refer to this field for every packet it receives.  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. The MLID 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 MLID determines the selected line speed and places that value in this field. Some common values are listed below:  <table border="0"> <tr> <td>Ethernet</td> <td>10Mbps</td> <td>000Ah</td> </tr> <tr> <td>Token-Ring</td> <td>4Mbps</td> <td>0004h</td> </tr> <tr> <td>Token-Ring</td> <td>16Mbps</td> <td>0010h</td> </tr> <tr> <td>FDDI</td> <td>100 Mbps</td> <td>0064h</td> </tr> <tr> <td>ISDN</td> <td>64 Kbps</td> <td>8040h</td> </tr> </table>	Ethernet	10Mbps	000Ah	Token-Ring	4Mbps	0004h	Token-Ring	16Mbps	0010h	FDDI	100 Mbps	0064h	ISDN	64 Kbps	8040h
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.															
4Eh	MPrioritySup	1	This field contains the number of priority levels that the MLID can handle. This field has a maximum of 7 priorities (1–7). Zero indicates no priority packet support. Therefore, the MLID can set this field to a value of 0 through 7.															

**Table 15.1 MLID Configuration Table Field Descriptions** (continued)

Offset	Name	Size (in bytes)	Description																					
4Fh	MBusID	1	<p>If the MLID supports multiple bus types, it checks this field during initialization to determine which bus it should be initialized for.</p> <p>This field is defined as follows:</p> <table> <tr> <td>BUS_ID_ISA</td> <td>equ</td> <td>0</td> </tr> <tr> <td>BUS_ID_MCA</td> <td>equ</td> <td>1</td> </tr> <tr> <td>BUS_ID_EISA</td> <td>equ</td> <td>2</td> </tr> <tr> <td>BUS_ID_PCMCIA</td> <td>equ</td> <td>3</td> </tr> <tr> <td>BUS_ID_PCI</td> <td>equ</td> <td>4</td> </tr> <tr> <td>BUS_ID_VESA</td> <td>equ</td> <td>5</td> </tr> <tr> <td>BUS_ID_HSM_DEFAULT</td> <td>equ</td> <td>-1</td> </tr> </table> <p>If <i>MBusID</i> is set to -1, the MLID searches each of the machine's busses for a supported LAN adapter and initializes the first LAN adapter it finds. The MLID determines the order to search the busses in.</p> <p>If <i>MBusID</i> is initially set to default, the MLID sets it to the appropriate bus ID.</p>	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
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																						
50h	MMLIDMajorVer	1	This field defines the current major revision level of the MLID. This field must match the revision level displayed in the <i>DriverSignOnMessage</i> string. For example, if the MLID's current version is 2.01, this field is set to 2.																					
51h	MMLIDMinorVer	1	This field defines the current minor revision level of the MLID. This field must match the revision level displayed by the <i>DriverSignOnMessage</i> string. For example, if the MLID's current version is 2.01, this field is set to 01.																					
52h	MFlags	2	This field contains flags, which are defined in Table 15.3.																					
54h	MSendRetries	2	<p>The MLID initializes this field to an appropriate value that represents the number of times the MLID will retry an errored transmission operation before giving up.</p> <p><b>Note:</b> 10 is a nominal default.</p>																					
56h	MLink	4	The MLID sets this field to 0 and does not modify it.																					
5Ah	MSharingFlags	2	This field contains flags, which are defined in Table 15.4.																					
5Ch	MSlot	2	<p>The MLID initializes this field to 0.</p> <p>MLIDs that control slot-based LAN adapters (for example, the Micro Channel Architecture bus boards) use this field. If the MLID is for an ISA board, it can ignore this field. If the MLID is for a Micro Channel Architecture, PCI, or EISA type board, it sets the slot number of the LAN adapter it is driving.</p> <p>Slot numbers are 1-based. An initial value of 0 implies that the MLID scans for the board.</p> <p>The user can override this value with the NET.CFG file.</p>																					

**Table 15.1 MLID Configuration Table Field Descriptions** (continued)

Offset	Name	Size (in bytes)	Description
5Eh	MIOAddress1	2	<p>The MLID initializes this field to the default I/O port base address.</p> <p>If the MLID 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 MLID does not use I/O ports, it sets this field to 0.</p> <p>The user can override the default value using the NET.CFG file.</p>
60h	MIORange1	2	<p>This field defines the number of I/O ports decoded by the LAN adapter at <i>MIOAddress1</i>. Set this field to 0 if the LAN adapter does not use I/O ports.</p>
62h	MIOAddress2	2	<p>This field allows the MLID to define two I/O port base addresses. The definition is the same as <i>MIOAddress1</i>. Set this to 0 if the LAN adapter does not have a second range of I/O ports.</p> <p>The user can override the default value using the NET.CFG file.</p>
64h	MIORange2	2	<p>This field defines the number of I/O ports decoded by the LAN adapter at <i>MIOAddress2</i>. Set this field to 0 if the LAN adapter does not use I/O ports.</p>
66h	MMemoryAddress1	4	<p>The MLID initializes this field to the LAN adapter's default base memory address.</p> <p>If the MLID is self-configurable, it determines the appropriate value for the LAN adapter and places that value in this field before returning from initialization.</p> <p>If the LAN adapter does not use or define shared RAM or ROM, the MLID sets this field to 0.</p> <p>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.</p> <p>The user can override the default value using the NET.CFG file.</p>
6Ah	MMemorySize1	2	<p>This field defines the number of paragraphs (16 bytes) decoded at <i>MMemoryAddress1</i>. If <i>MMemoryAddress1</i> is not defined, the MLID sets this field to 0.</p>
6Ch	MMemoryAddress2	4	<p>This field allows the MLID to define a second memory address range for the MLID's LAN adapter to use.</p> <p>For example, <i>MemoryAddress1</i> 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.</p> <p>If the MLID is self-configurable, it determines the appropriate value for the LAN adapter and places that value into this field before returning from initialization.</p> <p>The user can override the default value using the NET.CFG file.</p>

**Table 15.1 MLID Configuration Table Field Descriptions** *(continued)*

<b>Offset</b>	<b>Name</b>	<b>Size (in bytes)</b>	<b>Description</b>
70h	MMemorySize2	2	This field defines the number of paragraphs (16 bytes) decoded at <i>MemoryAddress2</i> . If <i>MemoryAddress2</i> is not defined, the MLID sets this field to 0.
72h	MIRQLine1	1	<p>The MLID initializes this field to the LAN adapter's default interrupt request line (IRQ).</p> <p>If the MLID is self-configurable, it determines the appropriate value for the LAN adapter and places that value into this field before returning from initialization.</p> <p>If the LAN adapter does not use an interrupt line, the MLID sets this field to 0FFh (unused). If the MLID's LAN adapter supports IRQ 2 or 9, the MLID sets the value to be consistent with the LAN adapter's documentation.</p> <p>For example, if the LAN adapter's documentation specifies the default jumper setting as IRQ 2, the MLID places a value of 2 in this field. If the LAN adapter's documentation specifies a default jumper setting as IRQ 9, the MLID places a value of 9 in this field.</p> <p>The MLID sets this field to 0FFh if the field is not needed. The MLID must handle the IRQ 2 – IRQ 9 overlap correctly.</p> <p>The user can override the default value using the NET.CFG file.</p>
73h	MIRQLine2	1	<p>The MLID uses this field if the MLID's LAN adapter uses a second IRQ line. Set this field to 0FFh if it is not needed.</p> <p>The user can override the default value using the NET.CFG file.</p>
74h	MDMALine1	1	<p>The MLID initializes this field to the LAN adapter's default DMA channel number.</p> <p>If the MLID is self-configurable, it determines the appropriate value for the LAN adapter and places that value in this field before returning from initialization.</p> <p>If the LAN adapter does not use DMA, the MLID sets this field to 0FFh (unused).</p> <p>The user can override the default value using the NET.CFG file.</p>
75h	MDMALine2	1	<p>The MLID uses this field if the MLID's LAN adapter uses a second DMA channel. Set this field to 0FFh if the field is not needed.</p> <p>The user can override the default value using the NET.CFG file.</p>

## Configuration Table Flags

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

### MModeFlags

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

**Table 15.2**  
**MModeFlags Bit Description**

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

---

**MModeFlags Bit Description**

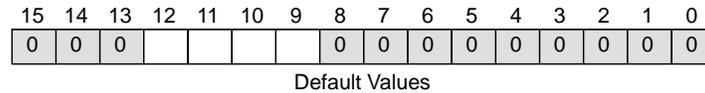
---

<b>Bit #</b>	<b>Description</b>
15, 14	<p>The MLID sets these bits to indicate whether the <i>MNodeAddress</i> field of the configuration table contains a canonical or a noncanonical address.</p> <p>Bit 15 indicates whether the node address format is configurable.</p> <p>Bit 14 indicates whether the configuration table <i>MNodeAddress</i> field contains the node address in canonical or noncanonical form. The state of bit 14 is only defined when bit 15 is set.</p> <p>The bit 15/bit 14 combinations are:</p> <ul style="list-style-type: none"><li>00 = <i>MNodeAddress</i> format is unspecified. The node address is assumed to be in the physical layer's native format.</li><li>01 = This is an illegal value and must not occur.</li><li>10 = <i>MNodeAddress</i> is canonical.</li><li>11 = <i>MNodeAddress</i> is noncanonical.</li></ul> <p>(See the <i>ODI Specification Supplement: Canonical and Noncanonical Addressing</i>.)</p>

---

### MFlags

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



**Table 15.3**  
MFlags Bit Description

<b>MFlags Bit Description</b>	
<b>Bit #</b>	<b>Description</b>
0	<i>Reserved.</i> Set to 0.
1	<i>Reserved.</i> Set to 0.
2	<i>Reserved.</i> Set to 0.
3	<i>Reserved.</i> Set to 0.
4	<i>Reserved.</i> Set to 0.
5	<i>Reserved.</i> Set to 0.
6	<i>Reserved.</i> Set to 0.
7	<i>Reserved.</i> Set to 0.
8	<i>Reserved.</i> Set to 0.
10,9	<p>These bits indicate different support mechanisms for multicast filtering and multicast address format. These bits are only valid if bit 3 of the <i>MModeFlags</i> is set, indicating that the MLID supports multicast addressing.</p> <p>The MLID sets bit 10 if it has specialized adapter hardware (such as hardware that utilizes CAM memory).</p> <p><b>Note:</b> If an MLID that usually defaults to using functional addresses also supports group addressing and sets bit 10, it receives both functional and group addresses.</p> <p>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 MLID does not need to perform any checking. The MLID can dynamically set and clear bit 9. For example, if the adapter utilizes CAM memory but has temporarily run out of memory, the MLID must temporarily filter the group addresses. In this case, the MLID would reset bit 9.</p> <p>The bit 10/bit 9 combinations are:</p> <ul style="list-style-type: none"> <li>00 = The format of the multicast address defaults to that of the topology:                  Ethernet   =&gt; Multicast addressing, in other words, Group addressing                  Token-Ring =&gt; Functional addressing/Group addressing                  FDDI       =&gt; Group addressing</li> <li>01 = Illegal value and must not occur.</li> <li>10 = Filter group address in MLID. Group addressing is supported by the specialized adapter hardware.</li> <li>11 = Adapter filtered group address. MLID software checking is not required. Group addressing is supported by the specialized adapter hardware.</li> </ul> <p>(See the <i>ODI Specification Supplement: Canonical and Noncanonical Addressing</i>.)</p>
11	<p>NESL_REQUIRED_BIT. If this bit is set, the LAN driver requires the NetWare Event Service Layer. The MLID should not load if this bit is set and the NESL is unavailable. For more information about the NetWare Event Service Layer (NESL), see the <i>NESL Specification: 16-Bit DOS Client Programmer's Interface</i>.</p>

---

**MFlags Bit Description**

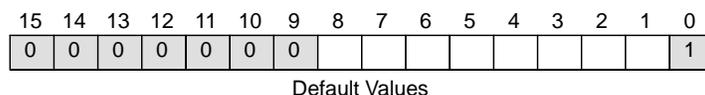
---

<b>Bit #</b>	<b>Description</b>
12	<i>PrioritySupportBit</i> . The MLID sets this bit during initialization if the following conditions are met: <ul style="list-style-type: none"><li>• The MLID has provided priority service support.</li><li>• The MLID has set the <i>MPrioritySup</i> field to something other than 0.</li></ul> <b>Note:</b> The MLID may temporarily clear this bit to disable priority support.
13	Reserved. Set to 0.
14	<i>Reserved</i> . Set to 0.
15	<i>Reserved</i> . Set to 0.

---

### MSharingFlags

This field informs the system which hardware resources an MLID and LAN adapter combination can share with other MLID and LAN adapter combinations. The first bit indicates when the MLID is shutdown. The MLID sets and clears this bit. If the MLID supports shareable interrupts, it must set the *CanShareIRQ* bit.



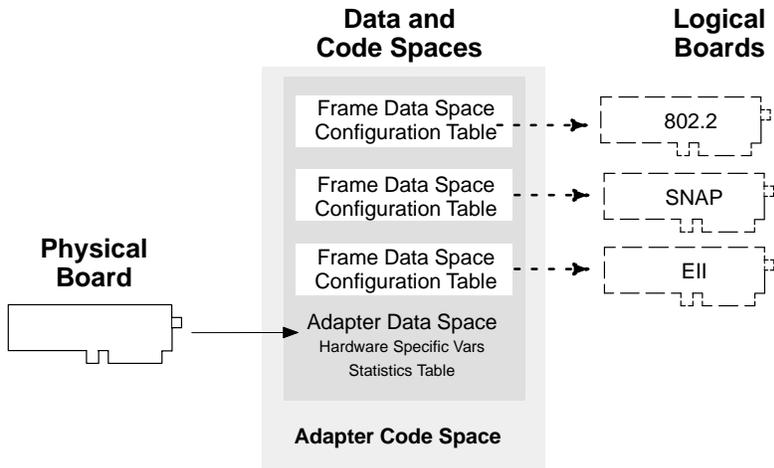
**Table 15.4**  
**MSharingFlags Bit Description**

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

## Adapter Data Space

The MLID must allocate and initialize a space called *AdapterDataSpace*. This space must contain the data that is specific to a particular LAN adapter. You must determine what hardware-specific fields the MLID needs in this space in order to drive its particular LAN adapter. But keep in mind that this space must also contain the MLID statistics table.

**Figure 15.3**  
Driver Frame and  
Adapter Data Space



### MLID Statistics Table

This section describes the MLID statistics table in detail. This section includes the statistics table definition and a description of each of the statistics table fields.

All MLIDs must keep a statistics table for each physical adapter for the purpose of network management. The following is the format of an MLID statistics table.



#### **Important**

A protocol stack treats this table as read only!

The statistics table contains various diagnostic counters. All statistics counters listed must be present in the table. These counters can be grouped into the following categories.

- Generic Statistics Counters
- Custom Statistics Counters

## MLID Statistics Table

```

MLIDStatStructure    struc
    MStatTableMajorVer    db    01
    MStatTableMinorVer    db    01    ;1.01
    MNumGenericCounters    dw    14
    MValidCounterMask    dd    ?    ;
    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 ?'

```

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

```

Message macro    name, string
    Local Tail
    name          db    (tail-2) - name
                  db    string, 0
    tail equ $
endm

```

**Figure 15.4**  
**Graphic Representation**  
**of the MLID Statistics Table**

MStatTableMajorVer				
MStatTableMinorVer				
MNumGenericCounters				
MValidCounterMask				
MTotalTxPackets				
MTotalRxPackets				
MNoECBsAvailable				
MTxTooBig				
MTxTooSmall				
MRxOverflow				
MRxTooBig				
MRxTooSmall				
MTxMiscError				
MRxMiscError				
MTxRetryCount				
MRxRetryCount				
MRxChecksumCount				
MRxMismatchLow				
MQueueDepth				
MNumCustomCounters				
CustomCounter0				
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
CustomCounter?				

**Table 15.5 MLID Statistics Table Field Descriptions**

<b>Offset</b>	<b>Name</b>	<b>Size (in bytes)</b>	<b>Description</b>
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; <i>MDriverStatMajorVer</i> = 1, <i>MDriverStatMinorVer</i> = 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 MLID is actually using. The bit field is 32 bits long and the most significant bit corresponds to the first generic counter, <i>MTotalTxCount</i> . 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 MLID requested to transmit. Whether or not they were actually transmitted depends upon the MLID.
0Eh	MTotalRxPackets	4	This field contains the total number of incoming packets for which the MLID received an ECB.
10h	MNoECBsAvailable	4	This field is used to count the number of incoming packets that were not received or wanted because an ECB was not provided by a protocol stack after calling <i>GetStackECB</i> .
14h	MTxTooBig	4	This field has the number of requested packets for transmission that were too big to send.
18h	MTxTooSmall	4	This field contains the number of requested packets for transmission that were normally too small to be transmitted.
1Ch	MRxOverflow	4	The MLID 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 <i>MMaxPacketSize</i> .
24h	MRxTooSmall	4	This field contains the number of incoming packets that were smaller than the minimum legal size for the media.
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 MLID increments this field when the MLID 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 and/or 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.

**Table 15.5 MLID Statistics Table Field Descriptions** *(continued)*

<b>Offset</b>	<b>Name</b>	<b>Size (in bytes)</b>	<b>Description</b>
3Ch	MQueueDepth	4	The MLID increments this field whenever it queues a packet. It decrements this field whenever it removes a transmit packet from the transmit queue.
41h	MNumCustomCounters	2	<p>This field has the total number of custom variables which follow this WORD. The statistics table allows the MLID to define a number of custom counters. These counters are MLID 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.</p> <p>When a custom counter corresponding to a receive or transmit error event is incremented, the MLID also increments the appropriate <i>MiscCount</i> counter (for example, <i>TxMiscCount</i>). The miscellaneous counters total all custom defined error events. We recommend that counter increments be done as follows:</p> <pre>add RxOverflowCount+0, 1 adc RxOverflowCount+2, 0</pre> <p><b>Note:</b> You can define other counters for debugging purposes, but remove them when the driver is shipped to an end user.</p>

□





*Chapter 16* **MLID Initialization**

## **Overview**

This chapter covers the steps involved in initializing and registering the MLID.

You should review this chapter before writing the MLID initialization routine.

## MLID Initialization

If your MLID is to be resident, it should free all the memory it used to hold the initialization code and data before turning resident. (The process of freeing initialization code and data is a design implementation decision.)

Under the ODI specification, the MLID has two essential functions:

- to take packets off the LAN adapter and pass them to the LSL and
- to take packets from the LSL and place them on the board.

Before the MLID can perform its functions, it must be initialized. The initialization process of an MLID under DOS occurs in the following stages:

- 1) The MLID registers with the LSL. An MLID locates the LSL in the same way that a protocol stack does; therefore, see “Locating the LSL” in *Chapter 4: Protocol Stack Initialization* for the steps in locating the LSL.
- 2) The MLID reads the NET.CFG file and fills in the MLID configuration table with the necessary information.
- 3) The MLID calls the LSL initialization entry point with the following information:

```

BX    LSLINIT_MLID_REG
      MLID initialization function code
ES:SI Points to the MLIDInfoBlockStruc

```

```

MLIDInfoBlockStruc struc
    MIBS_SendEntry    dd ?
    MIBS_ControlEntry dd ?
    MIBS_ConfigTable  dd ?
MLIDInfoBlockStruc ends

```

### *MIBS\_SendEntry*

Address of the MLID send entry point. All packets to be sent on the network will be sent through this address.

### *MIBS\_ControlEntry*

Address of the MLID control entry point.

### *MIBS\_ConfigTable*

Address of the MLID configuration table valid at the time this call is made.

```

DS:DI    Pointer to MLIDRetInfoBlockStruc for the
          LSL to return configuration information.

```

```
MLIDRetInfoBlockStruc struc
    MRIBS_MLID_SUP    dd ?
    MRIBS_BoardNum    dd ?
    MRIBS_ECB_DataSize dd ?
MLIDRetInfoBlockStruc ends
```

*MRIBS\_MLID\_SUP*

Address of the MLID support entry point of the LSL.

*MRIBS\_BoardNum*

Board number assigned to the MLID.

*MRIBS\_ECB\_DataSize*

Maximum buffer size of receive ECBs in the system.

- 4) At this point, the MLID should initialize the hardware. If the hardware fails, make the *DeRegisterMLID* call to the MLID support entry point to remove the MLID from the LSL's list of MLIDs. The process should then be terminated and an error message sent to the user.
- 5) The MLID informs the LSL about the protocols the MLID can process. The protocols are processed using the *AddProtocolID* call. Only Protocol IDs mentioned in NET.CFG should be added, since there are a limited number of protocol stacks supported by the LSL.
- 6) The MLID terminates execution in the operating system and remains resident. At this point, the driver is installed in the computer's system and is able to begin sending and receiving packets.

The MLID repeats Step 3 for each frame type. The LSL assigns a unique board number on each call to *RegisterMLID*.

□



*Chapter 17* **MLID Packet Reception and  
Transmission**

## **Overview**

This chapter describes the steps involved in MLID packet reception and transmission. This chapter also describes how to provide a receive look ahead buffer. You should read this chapter before writing a packet transmission and/or packet reception routine.

## MLID Packet Reception

When the physical board obtains a packet from the network, it generates an interrupt to which the MLID responds by calling its packet receive handler or board service routine. Generally, the receive handler requests an ECB from the LSL and then fills it in with information about the incoming packet. When the ECB is filled out, the MLID passes the address of the ECB to the LSL. The LSL then transfers the information to the correct protocol stack.

When the MLID receives a packet, it generally performs the following steps.

- 1) Saves the state and disables system interrupts if not already disabled.
- 2) Starts an internal critical section.
- 3) Enables system interrupts so external processes can occur. The MLID performs the following, order sensitive, procedures:
  - a) Disables the physical board's interrupts.
  - b) Clears the interrupt from the appropriate PICs.
  - c) Enables system interrupts.
- 4) Fills in the *LookAheadStruc* structure. Make sure the appropriate bits are set.
- 5) If the packet contains errors, sets the *DEST\_ERRORED* bit in the *LDestType* field of the *LookAheadStruc* structure and fill out the appropriate error bit in the *LPacketAttrib* field.
- 6) Requests a receive buffer (ECB) by calling the LSL support routine *GetStackECB* with a pointer to the *LookAheadStruc* structure.
- 7) Performs one of the following steps, depending on whether an ECB was available:
  - a) If a receive ECB is not available, discards the packet.
  - b) If a receive ECB is available, copies the data into the buffer described by the ECB. This buffer can be fragmented. The media header is not included as part of the data.
- 8) Sets the following fields in the ECB:

- PreviousLink
- Status
- ImmediateAddress
- DataLength
- DriverWorkspace

For a description of these fields, see the ECB field descriptions in *Appendix A: Event Control Blocks (ECBs)*.

- 9) Transfers the data.
- 10) Calls *HoldReceiveEvent*; this call places the specified ECB on the LSL's holding queue for processing.
- 11) Increments the appropriate counters.
- 12) Checks if the board has received another packet. If the board has received another packet, initiates the packet reception process again.
- 13) Returns interrupts to their original state before exiting the procedure by performing the following, order sensitive steps:
  - a) Disables system interrupts.
  - b) Enables the physical board's interrupts.
  - c) Enables system interrupts.
- 14) Terminates the internal critical section.
- 15) Calls *ServiceEvents*, which must be called prior to exiting the procedure in order to process any queued ECBs that have been placed on the hold queue.
- 16) Returns control back to the calling procedure by restoring the registers and doing an IRET.

The board service routine might also receive transmit complete interrupts. In this case, this procedure must be able to handle the additional overhead involved in completing and re-issuing send requests.

## Lookahead Buffer

The MLID must be able to provide the look ahead data before it can obtain a receive buffer. You can use one of three methods to move receive data from the physical board to host memory:

- Shared Memory
- Programmed I/O

- DMA

The method you chose is determined by the physical board.

## Shared RAM

If the board uses shared RAM, *LkAhd\_MediaHeaderPtr* and *LkAhd\_DataLookAheadPtr* of the *LookAheadStruc* structure point to the appropriate point in the shared RAM area.

## Programmed I/O

If the board uses port I/O to transfer packet data to system memory, you must create a buffer in the MLID that is large enough to hold the look ahead portion of the packet. The maximum size of this buffer is 128 bytes plus the size of the media header. For example, the maximum buffer an Ethernet packet needs is 150 bytes (128 bytes data + 22 bytes [14 MAC header + 3 802.2 UI (AAAA03) + 5 SNAP header] media header). After reading in the minimum number of bytes necessary to provide the look ahead data, set the *LkAhd\_MediaHeaderPtr* pointer to point to the start of the data in this buffer and set the *LkAhd\_DataLookAheadPtr* data pointer to point to the appropriate point in this buffer.

## DMA

If the board uses DMA, you should create a continuous, intermediate buffer that is capable of holding the largest possible packet. The MLID will transfer the entire packet off the board into this buffer. You should set the *LkAhd\_MediaHeaderPtr* pointer and the *LkAhd\_DataLookAheadPtr* data pointer to point to the appropriate points in this buffer

## MLID Packet Transmission

The MLID transmits packets through the physical board. When a packet is ready to be sent, the protocol stack prepares an Event Control Block (ECB) and calls the LSL's send packet routine. The LSL inspects the ECB board number and calls the associated MLID send handler. (The send handler entry point is exchanged with the LSL during initialization time.)

In order to prepare a packet for transmission when called by the LSL, the MLID generally performs the following steps:

- 1) Starts an internal critical section.
  - a) Disable system interrupts.

- b) Disable the physical board's interrupts.
  - c) Enable system interrupts.
- 2) Checks if the hardware is busy with a send. If it is, the MLID queues the send and rechecks to see if the hardware is busy.
- 3) Sets the busy flag and inspects the ECB for raw sends, if the hardware is not busy sending.
- 4) Determines whether the packet is a raw send.
  - a) If the packet is a raw send, the MLID does not need to generate the media header.
  - b) If the packet is not a raw send, the MLID must generate a media header.
- 5) Enables system interrupts in order to allow external processes to occur. The following steps explain the order sensitive step.
- 6) Begins transmission of the header and data by issuing a send request to the hardware.
- 7) Increments appropriate counters.
- 8) Performs one of the following, depending upon whether the transmission is lying or non-lying:

**Non-lying sends**

After the send operation has completed, the MLID sets the *ECB\_Status* field to 0 if the send was successful or to an appropriate error code if the send was unsuccessful. (This usually takes place in the Interrupt Service Routine (ISR) after receiving a send complete interrupt.)

**Lying sends**

Immediately following the send request to the hardware, the MLID sets the *ECB\_Status* field to 0 as if the send had completed successfully.

**Note** If priority support is provided, the MLID should transmit the frame by the corresponding priority means.

- 9) Returns the ECB by calling *SendComplete*. If a transmit monitor is registered, the completed MLID creates a TCB to pass to the monitor for its inspection prior to calling *SendComplete*.

**Note** If the MLID is doing non-lying sends, it must maintain a pointer to the ECB. *SendComplete* requires a pointer to the ECB to be returned. If the MLID is doing non-lying sends, the packet is usually returned in the ISR.

10) The MLID checks its internal queue for pending transmits and initiates the next send if any pending transmits are found.

11) Returns interrupts to their original state before exiting the procedure. The following steps give the order sensitive steps.

- a) Disable system interrupts.
- b) Enable the physical board's interrupts.
- c) Enable system interrupts.

12) Terminates the internal critical section and calls *ServiceEvents*, which must be called prior to exiting the procedure to process any queued ECBs that have been placed on the send queue.

13. Returns control to the calling procedure.

**Note** The entity that made the transmit request should not poll for completion of the transmit request but should wait until it's ESR in the transmit ECB is called. Polling for completion of the transit request can cause dead-locks to occur and the system to fail.

□





*Chapter 18* **MLID Control Routines**

## Overview

The ODI specification requires an MLID to provide a number of control procedures to protocol stacks. Protocol stacks can obtain the MLID control entry point by calling *GetMLIDControlEntry* and then invoking the returned entry point with the proper registers set. These functions can only be called at process time.

The following lists the MLID control routines along with their entry point and function number in alphabetical order.

<b>Descriptive Name</b>	<b>Function Name</b>	<b>Funct. No.</b>
AddMulticastAddress	ADD_MULTICAST_ADDRESS	2
DeleteMulticastAddress	DELETE_MULTICAST_ADDRESS	3
DriverManagement	DRIVER_MANAGEMENT	14
DriverPoll	DRIVER_POLL	12
GetMLIDConfiguration	GET_MLID_CONFIGURATION	0
GetMLIDStatistics	GET_MLID_STATISTICS	1
GetMulticastInfo	GET_MULTICAST_INFO	15
MLIDReset	MLID_RESET	6
MLIDShutdown	MLID_SHUTDOWN	5
PromiscuousChange	PROMISCUOUS_CHANGE	10
RegisterTxMonitor	REGISTER_TX_MONITOR	4
SetLookAheadSize	SET_LOOK_AHEAD_SIZE	9
Reserved	RESERVED	7
Reserved	RESERVED	8
Reserved	RESERVED	11
Reserved	RESERVED	13

The following lists the MLID control routines along with their entry point and function number in function number order.

<b>Descriptive Name</b>	<b>Function Name</b>	<b>Funct. No.</b>
GetMLIDConfiguration	GET_MLID_CONFIGURATION	0
GetMLIDStatistics	GET_MLID_STATISTICS	1
AddMulticastAddress	ADD_MULTICAST_ADDRESS	2
DeleteMulticastAddress	DELETE_MULTICAST_ADDRESS	3
RegisterTxMonitor	REGISTER_TX_MONITOR	4
MLIDShutdown	MLID_SHUTDOWN	5
MLIDReset	MLID_RESET	6
Reserved	RESERVED	7
Reserved	RESERVED	8
SetLookAheadSize	SET_LOOK_AHEAD_SIZE	9
PromiscuousChange	PROMISCUOUS_CHANGE	10
Reserved	RESERVED	11
DriverPoll	DRIVER_POLL	12
Reserved	RESERVED	13
DriverManagement	DRIVER_MANAGEMENT	14
GetMulticastInfo	GET_MULTICAST_INFO	15

## AddMulticastAddress

**Description** Adds the specified node address to the multicast address table.

**Entry State**

*AX*  
has the number of a logical board.

*BX*  
is equal to *ADD\_MULTICAST\_ADDRESS* (2).

*ES:SI*  
has a pointer to the 6-byte multicast buffer holding the multicast address.

Interrupts  
are enabled.

**Return State**

*AX*  
has a completion code.

Flags  
Z flag is set according to *AX*.

Interrupts  
are enabled.

*DS, BP, SS, SP*  
are preserved.

**Completion Codes (AX)**

*LSL\_SUCCESSFUL* (0000h)  
Multicast was successfully enabled.

*LSLERR\_OUT\_OF\_RESOURCES* (8001h)  
The MLID has insufficient resources to enable the multicast address.

*LSLERR\_BAD\_PARAMETER* (8002h)  
The specified multicast address is invalid for the MLID's media type, or the board number is invalid.

*LSLERR\_BAD\_COMMAND* (8008h)  
Multicast addressing is not supported by the MLID and/or the underlying hardware device.

**Remarks**

Protocol stacks that will enable multicast reception should first check the *MulticastBit* bit in the MLID configuration table's *MModeFlags* field. Some LAN media (for example, RX-Net) do not support multicast. When an underlying MLID/adaptor does not support multicasting, the protocol stack should use broadcast transmission instead.

The MLID will keep a count of the times a specified address is added. When an address is deleted by *DeleteMulticastAddress*,

the count is decremented. When the count is 0, the address is disabled. This behavior allows two or more protocols to safely use the same multicast address.

The MLID manages enabled multicast addresses according to the physical adapter. The format of a multicast address is LAN medium dependent. The two most common formats are for Ethernet (Ethernet\_II/IEEE 802.3) and Token-Ring (802.5), which are summarized below. Proprietary LAN media that support multicast can have alternate address encoding methods. Therefore, a protocol stack should allow a multicast address that can be configured by the user (for example, a NET.CFG parameter). This allows the protocol stack to work correctly on proprietary LAN media.

## Ethernet Multicasts

Ethernet multicast addresses must have bit 0 of byte 0 set to 1 (for example, x1 xx xx xx xx xx). The address is value based; each value is unique and separate from other values. Most adapters for Ethernet create a multicast hash table to filter incoming packets destined to a multicast group. Hashing is not usually a guaranteed filter; therefore, more than one multicast address might be received by the adapter. This will cause the underlying MLID to receive unwanted multicast packets. The MLID will complete the filtering so that only addresses enabled through this command are actually passed to protocol stacks.

## Token-Ring Multicasts

Token-Ring multicast addresses in an ODI system are usually Token-Ring functional addresses. However, support for group addresses has been included in some Token-Ring hardware. New MLIDs should determine if the hardware will support both functional and group addressing and provide as much support as possible for both types of addressing. These addresses are bit based: each bit position in the address signifies an unique address (in other words, more than one address can be specified by simply setting multiple bits). Addresses always begin with C0-00, leaving 32 bits (4 bytes) for functional addresses. However, four of the 32 possible bits are reserved by IBM, leaving 28 unique multicast addresses available.

More than one multicast address can be added when you invoke the *AddMulticastAddress* command. For example, if C0 00 00 01 00 00 and C0 00 00 02 00 00 need to be enabled, *AddMulticastAddress* can be called twice (once for each

address) or simply called once with C0 00 00 03 00 00. Both methods are equivalent.

Token-Ring MLIDs keep a use count for each functional address bit. Token-Ring MLIDs can also implement global addressing.

**Note** Functional addresses should never be sent on the medium with more than one function bit set. If more than one function bit is set, the address will not work on all media. For example, Token-Ring accepts a functional address that has more than one function bit set but PCN\_II does not.

### Number of Supported Multicast Addresses Supported

The number of multicast addresses supported by an underlying MLID/LAN medium is not specified by ODI specification. In the case of Token-Ring, the maximum number supported is specified by the definition of the address, with 28 being the maximum. Ethernet, however, has an almost infinite number of possible addresses. The maximum supported by the server MLID will usually be high (32 or more) and low (2 to 16) in a client MLID. The *Novell ODI Specification: 16-Bit DOS Client HSMs* toolkit allows for a maximum of 16 multicast addresses.

#### See Also

*GetMulticastInfo*

*ODI Specification Supplement: Canonical and Noncanonical Addressing* for information regarding canonical and noncanonical addressing

## DeleteMulticastAddress

<b>Description</b>	Disables reception of a previously enabled multicast address.
<b>Entry State</b>	<p><i>AX</i> has the number of a logical board.</p> <p><i>BX</i> is equal to <i>DELETE_MULTICAST_ADDRESS</i> (3).</p> <p><i>ES:SI</i> has a pointer to the 6-byte multicast address to delete from the multicast address list.</p> <p>Interrupts are enabled.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p>Interrupts state is preserved.</p> <p>Flags Z flag set according to <i>AX</i>.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) One instance of the address was successfully deleted.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h) The specified multicast address is invalid for the MLID's media type, or the board number is invalid (see <i>AddMulticastAddress</i>).</p> <p><i>LSLERR_ITEM_NOT_PRESENT</i> (8004h) The specified address is not presently enabled in the MLID.</p> <p><i>LSLERR_BAD_COMMAND</i> (8008h) Multicast addressing is not supported by the MLID and/or the underlying hardware device.</p>
<b>Remarks</b>	<p>This routine disables reception of a previously enabled multicast address. This command decrements the MLID's use count for the specified address. When the use count becomes 0, response of that address is disabled. (See <i>AddMulticastAddress</i> for a discussion of multicast address formats.)</p>

## DriverManagement

<b>Description</b>	Provides a generic way of allowing protocol dependent functions to be defined.
<hr/>	
<b>Entry State</b>	<p><i>AX</i> has the number of a logical board.</p> <p><i>BX</i> is equal to <i>MLID_MANAGEMENT</i> (14).</p> <p><i>ES:SI</i> is a pointer to the management ECB.</p> <p>Interrupts are disabled.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p><i>ES:SI</i> is a pointer to the management ECB.</p> <p>Flags are set according to <i>AX</i>.</p> <p>Interrupts are disabled but might have been temporarily enabled by the MLID stack.</p> <p><i>DS, ES, SI, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The requested operation was successful. The ECB was returned to the caller.</p> <p><i>LSL_PENDING_SUCCESS</i> (0001h) The requested operation was successfully started but will complete asynchronously. The ECB is not returned. The ESR will be called after the operation completes.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h) The board number is invalid, or the first byte of the ECB <i>ProtocolID</i> field is invalid. The first byte must be greater than 41h (A) or less than 7Eh (~) inclusive.</p> <p><i>LSLERR_BAD_COMMAND</i> (8008h) MLID management support is not provided.</p> <p><i>LSLERR_NO_SUCH_HANDLER</i> (800Ah) The Protocol ID value is not supported.</p>
<hr/>	
<b>Remarks</b>	This control function is provided to allow the MLID a generic interface to MLID dependant management functions. The

implementation of this function is optional. If not implemented, a call to this function must return *LSLERR\_BAD\_COMMAND* (8008h).

The management ECB is of the form of an ECB, but all fields below the *ProtocolID* field can be redefined by the MLID.

The *ProtocolID* field is defined as a 6-byte string that uniquely identifies the MLID. The first character of the string must be greater than or equal to 41h (“A”) and less than or equal to 7Eh (“~”). The remaining characters are defined by the MLID. If the first character is not greater than or equal to 41h and less than or equal to 7Eh, the MLID should return with the completion code *LSLERR\_BAD\_PARAMETER* (8002h).

If the MLID does not recognize the value in the *ProtocolID* field, the MLID returns a completion code of *LSLERR\_NO\_SUCH\_HANDLER* (800Ah).

If the MLID must respond asynchronously to the management request, it should queue the ECB internally and return a status of *LSL\_PENDING\_SUCCESS* (0001h). When the queued request is complete, the MLID should place the ECB on the LSL hold event queue by calling *HoldReceiveEvent*. The LSL will then process the ECB during the next call to service events.

**See Also**

*ProtocolManagement*

Refer to *ODI Specification Supplement: The Hub Management Interface* and *ODI Specification Supplement: Brouter Support* for an implementation of this procedure.

## DriverPoll

<b>Description</b>	Assists polled MLIDs.
<b>Entry State</b>	<p><i>AX</i> is a board number.</p> <p><i>BX</i> is equal to <i>DRIVER_POLL</i> (12 [0Ch]).</p> <p>Interrupts interrupts are disabled but might be enabled by the driver.</p>
<b>Return State</b>	<p>Interrupts are preserved.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Remarks</b>	<p><i>DriverPoll</i> is an optional routine the LSL calls periodically to assist polled drivers every timer tick. The LSL also calls <i>DriverPoll</i> every time a protocol stack relinquishes control to the LSL. MLIDs written for adapters that do not have interrupt capabilities use this call; most MLIDs do not use it. If an MLID does need polling, it sets the <i>NeedsPolling</i> bit inside the MLID configuration table <i>ModeFlags</i> field (bit 5).</p> <p> <b>Important</b> <i>DriverPoll</i> should not be used for watchdog or timeout functions; instead, the MLID should schedule a reoccurring AES event that has a relatively long timeout (for example, 1 second) for this purpose.</p> <p><i>DriverPoll</i> generally behaves in the same manner as an interrupt service routine. However, a critical section is not set up before the MLID invokes <i>DriverPoll</i>. Therefore, if this routine runs with its interrupts enabled, the MLID must explicitly enter and exit a critical section.</p> <p> <b>Important</b> This routine must complete quickly because it is usually called from a timer interrupt.</p> <p>Calling <i>DriverPoll</i> with an invalid board number can cause <i>DriverPoll</i> to abort before completing function execution.</p>

## GetMLIDConfiguration

<b>Description</b>	Returns a pointer to the MLID configuration table for the specified logical board.
<b>Entry State</b>	<p><i>AX</i> has the number of a logical board.</p> <p><i>BX</i> is equal to <i>GET_MLID_CONFIGURATION</i> (0).</p> <p>Interrupts are enabled.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p><i>ES:SI</i> has a pointer to the MLID configuration table.</p> <p>Flags Z flag set according to <i>AX</i>.</p> <p>Interrupts are enabled.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) ES:SI has a pointer to the MLID configuration table.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h) The board number is invalid. ES:SI returned as an invalid pointer.</p>
<b>Remarks</b>	This command is supported by all MLIDs. A separate configuration table is maintained by the MLID for each adapter and frame type combination. (See <i>Chapter 15: MLID Data Structures</i> for the format of the MLID configuration table.)

## GetMLIDStatistics

<b>Description</b>	Returns a pointer to the MLID statistics table for the specified board.
<hr/>	
<b>Entry State</b>	<p><i>AX</i> has the number of a logical board.</p> <p><i>BX</i> is equal to <i>GET_MLID_STATISTICS</i> (1).</p> <p>Interrupts are enabled.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p><i>ES:SI</i> has a pointer to the MLID statistics table.</p> <p>Flags Z flag set according to <i>AX</i>.</p> <p>Interrupts are enabled.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) ES:SI has a pointer to the MLID statistics table.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h) The board number is invalid. ES:SI returned as an invalid pointer.</p>
<hr/>	
<b>Remarks</b>	<p>All MLIDs support this command.</p> <p>The MLID maintains one statistics table for each physical adapter. Each frame type (or logical board) present for that physical adapter uses the same table. The board number in <i>AX</i> can be any of the logical board values present for a physical adapter. Regardless of the logical board number, <i>GetMLIDStatistics</i> will return the same table for each logical board associated with the MLID. (See <i>Chapter 15: MLID Data Structures</i> for the format of the MLID statistics table.)</p>

## GetMulticastInfo

**Description** Allows various management entities to obtain the location of the group (multicast) and /or functional address table that the MLID is using.

---

**Entry State**

*AX* has the number of a logical board.

*BX* is equal to *GET\_MULTICAST\_INFO* (15).

Interrupts are enabled.

**Return State**

*AX* has the completion code.

*CX* the number of entries in the group (multicast) table.

*ES:DI* has a pointer to the functional address buffer in use.

*ES:SI* has a pointer to the group (multicast) table in use.

Flags Z flag is set according to *AX*.

Interrupts are enabled.

*DS, BP, SS, SP* are preserved.

**Completion Codes (AX)**

*LSL\_SUCCESSFUL* (0000h)  
ES:SI and ES:DI are valid pointers.

*LSLERR\_BAD\_PARAMETER* (8002h)  
An invalid board number was specified.

*LSLERR\_BAD\_COMMAND* (8008h)  
Multicast addressing is not supported by the MLID and/or the underlying hardware device.

**Remarks**

The functional address pointer (DI) will be equal to 0 if the MLID and/or topology does not support functional addressing. The functional address buffer is a 6-byte buffer with all active functional bits set. The address will be in the topology dependant bit order.

The group (multicast) table consists of a number of multicast table entries; the number of entries is indicated in *CX* on return. The format of each group (multicast) table entry is defined as follows:

```
MulticastTableEntry  struc
    MulticastAddress  db 6 dup (?)
    MulticastUseage   dw 0
MulticastTableEntry  ends
```

*MulticastAddress*

The multicast address in the topology transmission format. For example, Token-Ring is in MSB (noncanonical) format, and Ethernet is in LSB (canonical) format.

*MulticastUsage*

A counter indicating the number of calls made to activate this address. A value of 0 indicates that this address is not currently active.

**See Also**

*AddMulticast, DeleteMulticast*

See *ODI Specification Supplement: Canonical and Noncanonical Addressing* for information regarding canonical and noncanonical addressing.

## MLIDReset

<b>Description</b>	Causes the MLID to totally reinitialize the physical adapter.
<b>Entry State</b>	<p><i>AX</i> has the number of a logical board.</p> <p><i>BX</i> is equal to <i>MLID_RESET</i> (6).</p> <p>Interrupts are enabled.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p>Flag Z flag set according to <i>AX</i>.</p> <p>Interrupts are enabled.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The physical card has been reactivated.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h) The board number is invalid.</p> <p><i>LSLERR_FAIL</i> (8005h) The MLID was unable to reset its hardware. This might indicate a hardware failure or system corruption.</p>
<b>Remarks</b>	<p>This command also brings an MLID back into active operation if it was temporarily shut down.</p> <p>The <i>ShutDownBit</i> bit in each logical board's MLID configuration table <i>MSharingFlags</i> field will be reset to 0 when this function returns.</p> <p>This function leaves enabled any multicast addresses that were previously enabled.</p>

## MLIDShutdown

<b>Description</b>	Allows an application to shut down a physical adapter.				
<b>Entry State</b>	<p><i>AX</i> has the number of a logical board.</p> <p><i>BX</i> is equal to <i>MLID_SHUTDOWN</i> (5).</p> <p><i>CX</i></p> <table border="0"> <tr> <td style="padding-right: 20px;">0000h</td> <td>shut down hardware and deregister with the LSL (permanent shutdown).</td> </tr> <tr> <td>non-zero</td> <td>shut down hardware only (temporary shutdown).</td> </tr> </table> <p>Interrupts are enabled.</p>	0000h	shut down hardware and deregister with the LSL (permanent shutdown).	non-zero	shut down hardware only (temporary shutdown).
0000h	shut down hardware and deregister with the LSL (permanent shutdown).				
non-zero	shut down hardware only (temporary shutdown).				
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p>Flags Z flag set according to <i>AX</i>.</p> <p>Interrupts are enabled.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>				
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The MLID was successfully shut down.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h) The board number is invalid.</p> <p><i>LSLERR_FAIL</i> (8005h) The MLID was unable to shutdown its hardware. This might indicate a hardware failure or system corruption.</p>				
<b>Remarks</b>	<p>If the MLID is permanently shutdown, a subsequent call to <i>MLIDReset</i> will not be successful. Permanent shutdowns are normally only used to completely disable the hardware and restore any hooked interrupt vectors and return all system resources. A permanent shutdown causes the MLID to release all memory resources, including memory used for code and any data.</p> <p>MLIDs that are temporarily shut down can be brought back into operation by invoking the <i>MLIDReset</i> control command. All the adapter's logical boards represented by the logical board number in <i>AX</i> are affected by this command. All logical board</p>				

configuration tables that are affected by this command will have their *ShutDownBit* bit set in the *MSharingFlags* field.

Any outstanding protocol transmit and receive ECBs will be returned before this command is completed.

A NESL (NetWare Event Service Layer) event should be generated by this routine for each logical board.

## PromiscuousChange

<b>Description</b>	Invoked by protocol stacks to enable or disable promiscuous mode on the MLID's adapter.
<b>Entry State</b>	<p><i>AX</i> contains the board number.</p> <p><i>BX</i> is equal to <i>PROMISCUOUS_CHANGE</i> (10 [0Ah]).</p> <p><i>CX</i> has the promiscuous state.</p> <p>PROM_OFF (0000h) = Promiscuous mode off  PROM_MAC (0001h) = All MAC frames to be received.  PROM_NONMAC (0002h) = All non-MAC frames to be received.  PROM_SMT (0004h) = All SMT frames to be received.</p> <p>Interrupts are disabled.</p> <p><b>Note</b> CLD is in effect.</p>
<b>Return State</b>	<p><i>AX</i> contains a completion code.</p> <p><i>CX</i> has the current promiscuous state.</p> <p>Interrupts are disabled.</p> <p>Flags are set according to the value in <i>AX</i>.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The requested operation was completed successfully.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h) The board number is invalid.</p> <p><i>LSLERR_BAD_COMMAND</i> (8008h) This MLID does not support promiscuous mode.</p>
<b>Remarks</b>	A protocol stack can enable promiscuous mode multiple times without error; however, only the current call is in effect. The current value of the <i>CX</i> register determines whether promiscuous mode is enabled or disabled. If the LAN medium or adapter does not distinguish between MAC and non-MAC

frames (for example, Ethernet does not differentiate between MAC or non-MAC frames), non-zero values in the CX register enable promiscuous mode.

Call this function only at process time.

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

MLIDs that support promiscuous mode set bit 13 in the *MModeFlags* field of the MLID configuration table.

All stacks bound to each logical board provided by the MLID will be notified of a change to the promiscuous status through the LSL .

## SetLookAheadSize

<b>Description</b>	Tells the MLID the amount of look ahead data that is needed by the caller to properly process received packets.
<hr/>	
<b>Entry State</b>	<p><i>AX</i> has a logical board number.</p> <p><i>BX</i> is equal to <i>SET_LOOK_AHEAD_SIZE</i> (9).</p> <p><i>CX</i> has the requested look ahead size (0–128).</p> <p>Interrupts are enabled.</p>
<b>Return State</b>	<p><i>AX</i> has a completion code.</p> <p>Flags Z flag set according to <i>AX</i>.</p> <p>Interrupts are enabled.</p> <p><i>DS, BP, SS, SP</i> are preserved.</p>
<b>Completion Codes (AX)</b>	<p><i>LSL_SUCCESSFUL</i> (0000h) The LookAhead size is now at least as large as the requested size.</p> <p><i>LSLERR_BAD_PARAMETER</i> (8002h) The board number is invalid. LookAhead size was not changed.</p>
<hr/>	
<b>Remarks</b>	<p>As part of a protocol stack's initialization, this function should be invoked to properly configure the MLID specified in <i>AX</i> for the amount of look ahead data a protocol stack needs for packet reception. If the requested size is less than the MLID's current look ahead size value, the MLID will use the larger value. In other words, it is impossible to adjust the size downward.</p> <p>The default look ahead size is 18 bytes.</p>

## RegisterTxMonitor

**Description** Registers a transmit monitor with the MLID.

---

**Entry State**

*AX*

is the logical board number.

*BX*

is equal to *REGISTER\_TX\_MONITOR* (4).

*CX*

zero            Disable transmit monitoring.  
non-zero       Enable transmit monitoring.

*ES:SI*

Dword pointer to transmit monitor routine.

Interrupts

are disabled.

**Return State**

*AX*

has a completion code.

Flags

are set according to *AX*.

Interrupts

are disabled.

*DS, ES, SI, BP, SS, SP*

are preserved.

**Completion Codes (AX)**

*LSL\_SUCCESSFUL* (0000h)

The requested operation was completed successfully.

*LSLERR\_OUT\_OF\_RESOURCES* (8001h)

A transmit monitor is already registered for this logical board number.

*LSLERR\_BAD\_PARAMETER* (8002h)

The board number is invalid or unable to deregister because *ES:SI* pointed to a different transmit monitor routine than was previously registered.

---

**Remarks**

Protocols invoke *RegisterTxMonitor* when they want to monitor the packets the adapter is transmitting. The MLID will call the transmit monitor routine pointed to by *ES:SI* after the packet is sent to the adapter for transmission. This provides the protocol with the exact bytes being transmitted on the media.

## Transmit Monitor

The transmit monitor is passed a TCB in DS:SI. The transmit monitor can copy part or all of the packet described by the ECB but cannot modify it.

### Entry State

*DS:SI*  
has a pointer to an TCB.  
Interrupts  
are disabled.

### Return State

Interrupts  
are disabled and remained disabled.  
All registers must be preserved.

### Remarks

An MLID calls this function after a transmit has completed. This provides the transmit monitor a look at the actual transmitted packet.

## Transmit Control Block (TCB)

The MLID 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. 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?
TCBStructure      end
```

### Transmit Control Block (TCB)

Offset	Name	Size (in bytes)	Description
00h	TCBDriverWS	6	The MLID 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.

<b>Transmit Control Block (TCB) (continued)</b>			
<b>Offset</b>	<b>Name</b>	<b>Size (in bytes)</b>	<b>Description</b>
08h	TCBFragStrucPtr	4	This field contains a far pointer to a list of fragments as defined by the <i>FragmentStructure</i> .
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
    
```

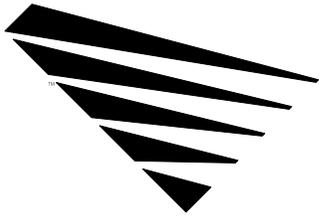
Additional fragments for FFragment Count > 1

```

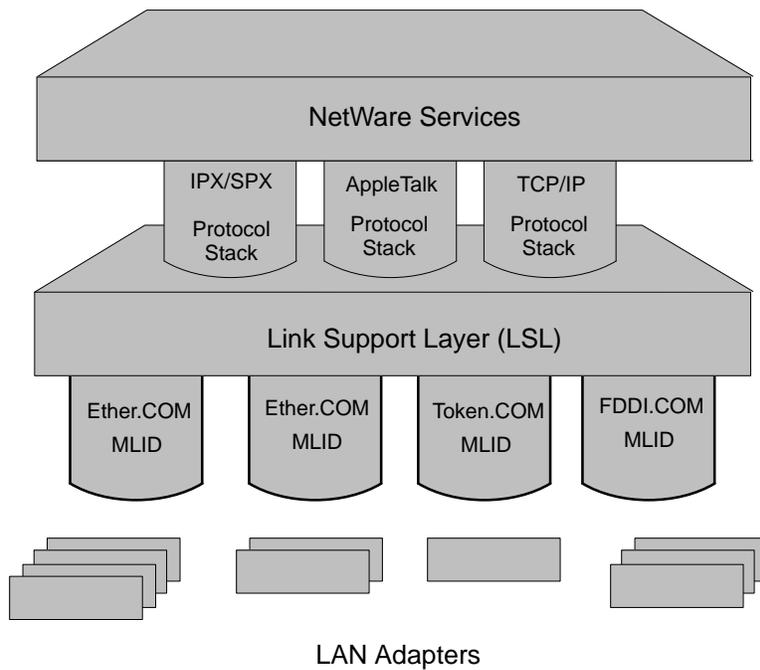
    FFrag?Address   dd 0
    FFrag?Length    dw 0
    
```

<b>Fragment Structure</b>			
<b>Offset</b>	<b>Name</b>	<b>Size (in bytes)</b>	<b>Description</b>
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	4	The fragment structure (address and length) is repeated for additional fragments if FFragmentCount > 1.
	FFrag?Len	2	

□



Section V **Appendixes**







*Appendix A* **Event Control Blocks (ECBs)**

## **Overview**

The ODI system uses Event Control Blocks (ECBs) for two purposes:

- to describe the protocol data during packet transmission and
- to describe the protocol buffers during packet reception

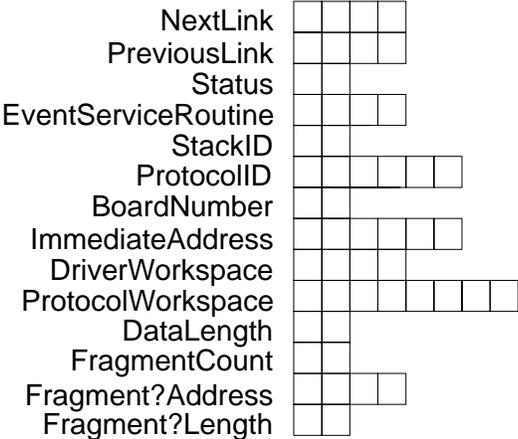
The format of the ECB is the same regardless of whether it is a send or a receive ECB.

This appendix includes the ECB structure in sample code, a graphic representation of the ECB, and a description of the ECB fields.

### Event Control Block Structure Sample Code

```
ECB struct
  NextLink      dd 0
  PrevLink      dd 0
  Status        dw 0
  ESR           dd ?
  StackID       dw ?
  ProtID        db 6 dup (?)
  BoardNum      dw ?
  ImmAddr       db 6 dup (?)
  DriverWS      db 4 dup (?)
  ProtocolWS    dw 4 dup (?)
  DataLen       dw 0
  FragCount     dw 1
  Frag1Addr     dd ?
  Frag1Len      dw ?
ECB ends
```

**Figure 18.1**  
**Graphical Representation**  
**of Event Control Block**



**Table G.1 Event Control Block Field Descriptions**

Offset	Name	Size (in bytes)	Description
00h	NextLink	4	This field is typically used as a forward link to manage a list of ECBs. The current owner of the ECB (the protocol stack, in this case) uses this field.
04h	PreviousLink	4	This field is typically used as a back link to manage a list of ECBs. The current owner of the ECB (the protocol stack, in this case) uses this field. This field is also used to pass receive information when in a receive ECB.  Word pointer <i>PreviousLink+0</i> holds the same value as in the <i>LPacketAttrib</i> field of the <i>LookAheadStruc</i> structure that was presented to obtain the ECB. The first byte ( <i>PreviousLink+0</i> ) is restricted to error conditions.
08h	Status	2	This field indicates the completion status of an ECB. This field is invalid until the associated Event Service Routine is called. If the routine is successful, this field will be equal to 0000h.  <i>Status</i> is 0 for error free reception. If the size of the ECB's buffers is less than the <i>FrameDataLength</i> value, <i>Status</i> will be set to <i>LSLERR_RX_OVERFLOW</i> (8006h). The <i>Data</i> buffers will be filled with as much of the data as possible. If any of the error bits in the <i>PreviousLink+0</i> field are set, the ECB will be returned with <i>Status</i> set to <i>LSLERR_CANCELLED</i> (8007h). The <i>DEST_ERRORED</i> bit would be set in word pointer <i>DriverWS+0</i> . <i>DEST_ERRORED</i> is an exclusive bit and thus will be the only bit set in an error case.
0Ah	EventServiceRoutine	4	The protocol stack sets this field to point to an appropriate routine that is to be called when the send or receive event is complete (either successfully or with an error). This field must point to a valid handler.
0Eh	StackID	2	When a packet is transmitted, the protocol stack sets this field to the protocol stack's assigned stack ID before the protocol stack sends the ECB to the LSL. When a packet is being received, the LSL sets this field to the stack ID assigned to the protocol stack that is receiving the packet. If a packet is being transmitted as a raw send, the protocol stack can set this field to 0FFFFh as a signal to the underlying MLID that this is a raw send. This gives the protocol stack the ability to specify the complete packet, including all low-level headers. See "Priority Packet Support" in <i>Chapter 2: Overview of Protocol Stacks</i> for a complete list of priority values.

**Table G.1 Event Control Block Field Descriptions**

Offset	Name	Size (in bytes)	Description
10h	ProtocolID	6	This field contains the Protocol ID (PID) value for sends and receives. If the ECB is a send ECB, the protocol stack sets this field before calling <i>SendPacket</i> . In a send ECB, the Protocol ID is embedded into the low-level packet header by the underlying MLID and is used to uniquely identify the packet as the caller's protocol type. In a receive ECB, the MLID stores in this field the Protocol ID embedded in the low-level packet header. 802.2 frames store only the DSAP in the Protocol ID field. The Protocol ID is stored in high-low order. See <i>ODI Specification Supplement: Frame Types and Protocol IDs</i> for an explanation of explicitly defining the 802.2 header to use in a transmit ECB.
16h	BoardNumber	2	When an MLID registers with the LSL, the MLID is given a logical board number. The <i>BoardNumber</i> field of the configuration table contains that board number. On sends, a protocol stack fills in this field to indicate the target logical board.
18h	ImmediateAddress	6	If the ECB is a send ECB, the protocol stack sets this field before calling <i>SendPacket</i> . The immediate address is the destination address of the packet on the physical network. If the ECB is a receive ECB, the underlying MLID sets this field to the packet's source node address before the MLID returns the ECB to the protocol stack. This source address is the node on the same physical network that just sent the packet. If the MLID is utilizing canonical addressing, the immediate address should be in canonical form.
1Eh	DriverWorkspace	4	This field is generally reserved for use by the MLID. Protocol stacks should not modify this field, unless the protocol stack currently owns the ECB. The first word of the workspace is defined as the <i>DestinationAddressType</i> field for the receive packets. The MLID sets this word to the same value as <i>LDestType</i> in the <i>LookAheadStruc</i> structure when it is a receive ECB. Word pointer <i>DriverWorkspace+2</i> holds the <i>FrameDataLength</i> value of the packet.
22h	ProtocolWorkspace	8	This field is reserved for use by the originating protocol stack and must not be modified by the LSL or MLIDs.
2Ah	DataLength	2	This field holds the amount of valid data contained in the ECB's buffers.
2Ch	FragmentCount	2	This field contains the number of fragment buffer descriptors immediately following this field. This value cannot be 0 or larger than 16.

**Table G.1 Event Control Block Field Descriptions**

Offset	Name	Size (in bytes)	Description
2Eh	Fragment?Address	4	This field specifies a far pointer to a data buffer of <i>Fragment?Length</i> .
32h	Fragment?Length	2	This field specifies the length of the buffer pointed to by <i>Fragment?Address</i> . This field can be 0, in which case the MLID will skip over it when transmitting or receiving data.

□



*Appendix B* **Compatibility with Multitasking DOS  
Products**

## Globally Accessible Data Buffers

The INTEL80386 microprocessor enables an 80386 protected mode application to create and maintain multiple virtual 8086 machines, or DOS boxes. Each DOS box can be preempted so that more than one DOS application can run at once (multitasking). DOS ODI modules can coexist with these products if all data structures and code entry points used by the LSL and underlying MLIDs are globally accessible from the context of any DOS box.

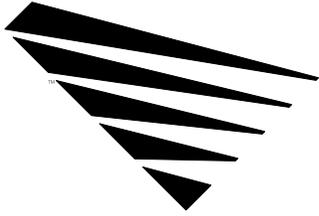
DOS applications (for example, *NETX.COM*) running on top of ODI modules send and receive packets through a network protocol that uses data buffers provided by the application programs. Unfortunately, this means that the application can only access the buffers in the real mode of the DOS box in which the application is operating. If the application accesses a buffer from another DOS box, that buffer will not be the intended buffer, but a random piece of memory that happens to exist at the same SEGMENT:OFFSET address. If this happens, the intended results of the application will not be achieved.

Before Network Layer protocols can send or receive using an ECB, the ECB, the ECB's ESR, and the associated data buffers must be globally accessible. These data structures and routines must be accessible at interrupt time or after a context switch to another DOS box.

## Microsoft Windows 386 Enhanced Mode

When DOS device drivers (including TSRs) run under Microsoft Windows, they should interface to a Window's Virtual Device Driver (VxD) that has been written to conform with multitasking issues (for example, memory globalization). If your protocol stack will be operating under Microsoft Windows, you will probably have to develop a VxD for your protocol stack. Refer to the Microsoft Windows SDK for more information.

□



*Appendix C* **The 802.2 Type II Frame Header**

## **Overview**

This appendix describes the changes that enable a protocol stack to receive packets with 802.2 Type II headers.

## Support of the 802.2 Type II Frame

Protocol stacks previously supported only the 802.2 Type I frame. When a Type II packet was received, the stacks considered the second control byte of the header as part of the packet's data.

### Packet Transmission

Previously, if a protocol stack needed to specify the complete 802.2 Type I header, the protocol stack would place a value of 02h in Byte 0 of the *ProtocolID* field of the transmit ECB. Now, in order to allow a protocol stack to support 802.2 Type II headers, the *ProtocolID* field of the transmit ECB can be filled out according to the following table:

<b>Protocol Stacks that:</b>	<b>Byte 0</b>	<b>Byte 1</b>	<b>Byte 2</b>	<b>Byte 3</b>	<b>Byte 4</b>	<b>Byte 5</b>
Are not 802.2 aware (for example, IPX)	0	0	0	0	0	DSAP
Specifically specify the 802.2 Type I header	2	0	0	DSAP	SSAP	Ctrl0
Explicitly specify the 802.2 Type II header	3	0	DSAP	SSAP	Ctrl0	Ctrl1

Byte 0 contains the number of extra bytes in the *ProtocolID* field defining the explicit 802.2 header. The protocol stack should use byte 0 as a zero-based count of how many bytes of the 802.2 header is contained in the *ProtocolID* field. For example, if the value in byte 0 is 02h, the 802.2 header starts with the third byte following the *ProtocolID* field.

The value in byte 0 is a count and should not be compared for a specific value except to check for the special case of 0. If the value in byte 0 is equal to 0, byte 5 of the *ProtocolID* field contains the DSAP field of an 802.2 Type I header. (The 802.2 header DSAP = SSAP, and UI = 03. For example, DSAP = E0 generates E0 E0 03, an 802.2 Type I UI header.)

### Packet Reception

Bits 0 and 1 of the *Ctrl0* field indicate whether an 802.2 header is Type I or Type II. If both bit 0 and bit 1 are set, the header is 802.2 Type I. Any other combination (00, 01, 10) indicates an 802.2 Type II header. The MLID no longer uses the first byte of

the first fragment buffer as the *Ctrl1* field for an 802.2 Type II header; the first byte of the first fragment buffer now contains the first byte of the received packet's data. The *ProtocolID* field of the receive ECB contains the DSAP value of the 802.2 header regardless of whether the header is Type I or Type II.

If a packet contains an 802.2 header, the *RX\_8022\_TYPEx* bit will be set in the *LDestType* field of the *LookAheadStruc* structure. After the MLID has filled in the ECB, the first word of the *DriverWS* field of the ECB will be filled in with the same value. If *RX\_8022\_TYPE2* bit is set, the *Ctrl0* and *Ctrl1* fields exist in the MAC header. If a protocol needs the 802.2 Type 1 or Type 2 header information, it should be saved at look ahead time.

□



*Appendix D* **Promiscuous Mode**

## **Overview**

This appendix describes how promiscuous mode can be implemented and some of the issues involved in implementation.

## Implementing Promiscuous Mode

An MLID can receive a packet in one of four ways:

- The packet is routed specifically to the MLID.
- The packet is a broadcast packet and is received by all MLIDs on the LAN.
- The packet is a multicast packet, and the MLID's address is in the multicast table. (The multicast table is typically limited to 16 in DOS ODI.)
- The MLID has enabled promiscuous mode and is receiving all packets on the wire.

When the MLID receives a packet, it must determine the packet type. If the LAN adapter is capable of receiving errored frames in promiscuous mode, it should do so.

Protocols that wish to receive and/or monitor promiscuous traffic need to do the following things.

1. Obtain the entry point for the logical board you wish to monitor.
2. Register with the LSL as a prescan receive or default stack.
3. Call the LSL's *ModifyStackFilter* function to adjust to see desired packet types.
4. Call the appropriate routine to enable promiscuous mode.

After these steps, packets that pass the filter will start being presented to the protocol's receive handler.

## Implementation Considerations

Prescan and default protocol stacks need to consider the following when looking at operations on ECBs.

### Multitasking

Global buffers should be provided to the LSL and lower layers. Task switching and real-protected mode switching must also be handled appropriately.

### Packet Sequencing

Prescan and default protocol stacks must ensure that receive and transmit packets are kept in sequence. While a protocol stack is operating on a packet, all subsequent packets must be queued to maintain the proper sequence.

## ECB Does not Contain the Entire Packet

If a bound protocol stack supplies the ECB for a received packet and does not need the entire packet, the ECB might not contain all of the received packet's data. For example, the packet might contain 100 bytes, but the protocol stack might only need to transfer 80 bytes to its supplied buffers. While this might be a rare occurrence, it can be detected if the protocol stack examines the *DataLength* field of the ECB, and compares this field to the *LookAheadStruc* structure's *LDataSize* field. Prescan and default stacks should always supply full sized ECBs for packets that they are consuming and that they will be resubmitting to the LSL.

## Error Packets

When in promiscuous mode, error packets can be received by setting the appropriate bit through calling the *ModifyStackFilter* function in the LSL protocol support entry point. The *LDestType* field in the *LookAheadStruc* structure will have only the *DestError* bit set, and the actual error type will be defined in the *LPacketAttrib* field in the *LookAheadStruc* structure. All error packets that are received without identifying a board number will be received on the first logical board for the physical board; usually, this is board number 0 if there is only one LAN adapter in the machine.

## Logical vs. Physical

Promiscuous mode is enabled on a physical board basis, but stack chaining is only available on a logical board basis. In other words, when you enable promiscuous mode on one logical board, promiscuous mode is enabled on the physical board basis, and all logical boards on that physical adapter will have promiscuous mode enabled also. This does not present a problem since the filtering provided at the LSL precludes sending any packet to protocols other than those that have a filter set appropriately.

If a frame type is not loaded, or there is an error that causes the MLID to not be able to determine the frame type, it is received on the first logical (default) board of the MLID in question.

□



*Appendix E* **The NET.CFG Configuration File**

## **Overview**

The NET.CFG file is used by the various ODI modules (including the protocol stack and MLID) to obtain the network system configuration information at initialization time. The MLID parses the NET.CFG, evaluates the MLID parameters, and then sets those parameters in the MLID configuration table.

Each node (workstation) in a network system contains at least one network interface card (NIC), the NIC's MLID, the LSL, and the protocol stacks.

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

## Main Section Headings

Main section headings must be flush left and typically begin with one of the following labels: Protocol, Link Support, or Link Driver. Protocol stacks use the “Protocol” main section heading. The “Protocol” entry is followed by the name of the protocol stack to which the information in the section following the heading refers. This enables an ODI module to locate its configuration section. All configuration entries following the Main Section Heading must be preceded with 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 following example illustrates a sample NET.CFG file entry:

```
protocol ipx
    bind #2

protocol abc
    multicast 011234567890 ;Fictitious keyword

link driver ne1000
    int 3
    port 330
```

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 (;). For example:

```
protocol ipx          ;Main section header for IPX
    bind #1          ;Inform IPX that it should bind
                    ;to logical board 1
```

A protocol’s main section header should use an intuitive name such as the protocol’s executable file name—for example, TCP/IP.

During initialization, the protocol stack parses the NET.CFG for its main section header. The protocol then parses and interprets the configuration entries until the parser reaches the end of the file or until another main section header is encountered.

## Locating the NET.CFG File’s Directory

The LSL general service function *GetNetcfgPath* (see *Chapter 12: LSL General Services*) is available to the protocol stack to enable it to locate the directory in which the LSL found the NET.CFG file. All ODI modules should use this API function to open the NET.CFG file so that all modules use a common NET.CFG file.

## LAN Driver Keywords and Parameters

### DMA [#selection] Channel

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

#### Example

```
; Example NET.CFG file; this file is not functional  
  
; Configure DMA selection 1 to be channel 7  
Link driver 3c505  
DMA #1 7
```

### IRQ [#selection] Interrupt

*IRQ* configures the *#selection n* to be IRQ *n*. *#Selection* 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; this file is not functional  
  
; Configure selection 1 to be interrupt 9  
Link driver NE2000  
IRQ #1 9
```

### MEM [#selection] Address [length]

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

#### Example

```
; Example NET.CFG file; this file is not functional  
  
; Configure memory address and range  
Link driver TRXNET  
MEM #1 C0000 80
```

### Port [#selection] Address [length]

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

### Example

```
; Example NET.CFG file; this file is not functional

; Set up the card. Note that these settings are fictitious.
Link driver NE1000
    port 320
```

### Slot n

*Slot* indicates which slot contains the card for the MLID. N is 1 based; that is, the first slot is one (1), not zero (0).

### Slot ?

Slot ? indicates that the MLID is to scan the slots for the adapter. This is the default mode.

### Example

```
; Example NET.CFG file; this file is not functional

Link driver ne2
    slot 3
```

### Node Address h [format]

*Node Address* overrides any hard-coded node address in the MLID's hardware, if the hardware will allow it.

You can use either canonical or noncanonical format for any media when overriding the node address. For example:

```
Node Address 080000A5646BL    Node address in canonical form.
Node Address 100005A26D6M    Node address in noncanonical form.
```

If *M* or *L* is not specified, the default for overriding the node address is the physical layer form of the address.

### Example

```
; Example NET.CFG file; this file is not functional

; Set up the ne2 card. Note that all settings are fictitious.
Link driver ne2
    node address 1234abcd

;Set up the ne2 card for address in canonical form.
Link driver ne2
    node address 1234abcdL

;Set up the ne2 card for address in noncanonical form.
Link driver ne2
    node address 1234abcdM
```

### Protocol <name> h <frame type>

*Protocol* tells the MLID that the named protocol has a protocol identification number of h. This enables new protocols (or

overrides the default) to be handled by existing MLIDs. The MLID uses this information to make the *AddProtocolID* call to the LSL during its initialization.

### Example

```
; Example NET.CFG file; this file is not functional

; Identify IPX as 8137h on Ethernet_II, identify IP
as 800h, identify ARP as 806h
Link driver NE1000
    Frame Ethernet_II
    Protocol IPX 8137 Ethernet_II
    Protocol IP 800 Ethernet_II
    Protocol ARP 806 Ethernet_II
```

## Frame <name> Address Mode

*Frame* configures the link-level frame type to be <name>. The MLID will create this link-level frame if there is a possibility of more than one type. Multiple frame types can be loaded concurrently.

The following keywords set the OBR mode.

**LSB** Addresses are presented in canonical mode to the hardware.

**MSB** Addresses are presented in noncanonical mode to the hardware.

### Example

```
; Example NET.CFG file; this file is not functional

; Configure the MLID to support 4 frame types
Link driver NE2000
    Frame Ethernet_II
    Frame Ethernet_802.3
    Frame Ethernet_802.2
    Frame Ethernet_SNAP

; Configure the MLID for OBR support by frame type
Link Driver TOKEN
    Node Address 020012345678L
    Frame Token-Ring MSB
    Frame Token-Ring_Snap LSB
```

Where Token-Ring is in MSB format and Token-Ring\_SNAP is in LSB format. The appropriate configuration tables for each frame will indicate the MSB/LSB configuration.

## Custom Keywords

A protocol stack can define custom configuration keywords as the keywords are needed. All protocol modules should at least

parse and understand the “Bind” keyword with its associated logical board number. The board number specified must be preceded by a pound sign (#) and must be one-based (meaning that the first board number must start with one). You must keep in mind, however, that internally, board numbers are zero-based. Therefore, when the protocol stack obtains a board number from the NET.CFG file, it must subtract one from that number so that the board number is converted to a zero-based number for internal use. Remember, also, that before the protocol stack displays the board number on the monitor, the protocol stack must add one to the board number, converting it to a one-base number. Additional board number entries must be entered on another line. Following is an example NET.CFG file for a fictitious protocol stack:

```
protocol ABC
  bind #1
  bind #5

protocol XYZ
  bind #5
```







## **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 destination addresses on a network.

**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 a LAN 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 data-link protocol that specifies how data is placed on and retrieved from a common transmission medium.

**FDDI--Fiber Distributed Data Interface**

A cable interface capable of transmitting data at 100 Mbps. FDDI can operate over fiber lines or twisted-pair cable.

**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.

**Peripheral Component Interconnect--PCI**

A 32-bit or 64-bit bus standard with multiplexed address and data lines.

**Personal Computer Memory Card International Association--PCMCIA**

A 16-bit bus standard.

**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 in a frame that identifies the physical location of a node that is sending the packet.

**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 an instruction to return to the caller of the routine.

**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 an HSM to transmit data. The structure is typically used by MLIDs using the LAN Driver Toolkit. This structure is also used by MLIDs to pass information to transmit monitors.

**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

## Numbers

802.2 header, DriverWorkspace field, 5-12

802.2 header, specifying, 6-5

802.2 Type II frame, specifying, C-3

## A

adapter data space, 15-17  
defined, 14-6

AddGeneralService, defined, 13-3

Adding Protocol ID, process, 4-7

adding Protocol IDs, 4-7

AddmemoryToPool, defined, 13-6

AddMulticastAddress, defined, 18-3

AddProtocolID, defined, 10-5, 11-4

AESECB structure, 10-52, 11-18

AESESR, AESECB field defined, 10-52, 11-18

AESLink, AESECB field, defined, 10-52, 11-18

AESStatus, AESECB field defined, 10-52, 11-18

AllocateMemory, defined, 13-7

## B

BestDataSize, 4-10

HSM configuration table field, defined, 15-7

Binding

dynamic, 4-6

NET.CFG file entry, 4-6

overview, 2-4

process, 4-6

binding to logical boards, 4-11

BindStack, defined, 10-7

BindToMLID, defined, 7-3

bit maps

MFlags, 15-14

MModeFlags, 15-12

MSharingFlags, 15-16

board, service routine, overview, 14-4

Board number

defined, 2-2

using, 2-4

board support, multiple, 4-8

Board, logical. See Logical board, 2-2

BoardInstance, HSM configuration table field,  
defined, 15-6

BoardNumber

Event Control Block field, defined, A-5

HSM configuration table field, defined, 15-6

BoardNumber field, 6-5

Bound protocol stack, defined, 2-4, 5-6

Brouter, document, 18-8

buffer, lookahead, 17-4

bus type

Extended Industry Standard Architecture  
(EISA), 14-8

Industry Standard Architecture (ISA), 14-8  
listed, 14-8

Micro Channel Architecture, 14-8

Peripheral Component Interconnect (PCI),  
14-8

Personal Computer Memory Card Interna-  
tional Association (PCMCIA), 14-8

## C

CancelAESEvent, defined, 10-8, 11-5

canonical and noncanonical addressing, docu-  
ment, xi, 4-11, 6-6, 18-5, 18-13

commandline switches, LSL, 8-4

Completion codes, LSL. See LSL completion  
codes, 8-3

ConfigMajorVersion, LSL configuration table  
field, defined, 9-3

ConfigMinorVersion, LSL configuration table  
field, defined, 9-3

ConfigTableLink, HSM configuration table  
field, defined, 15-9

ConfigTableMajorVer, HSM configuration table  
field, defined, 15-6

ConfigTableMinorVer, HSM configuration table  
field, defined, 15-6

configuration files, custom, 8-5

configuration table

MLID, 14-5, 15-3

major version, 15-6

minor version, 15-6

protocol stack, 3-3

control, routines, MLID, overview, 14-4

Control commands for protocol stacks, Bind-  
ToMLID, 7-3

control procedure  
 required, 14-3  
 supported, 14-3

ControlStackFilter, defined, 11-6

custom configuration files, 8-5

customizing protocol stacks, 4-8

## D

data, transfer mode, methods, 14-8

data flow  
 receive, 1-7  
 send, 1-6

data space  
 adapter, 15-17  
 frame, 15-3

data structures, MLID, 14-5

DataLength, Event Control Block field, defined, A-5

DataLength field, 6-6

Default protocol stack, defined, 2-2, 2-5, 5-6

DefaultStackControlHandler, DefaultStackInfo structure field, defined, 10-41

DefaultStackInfo structure, defined, 10-41

DefaultStackReceiveHandler, DefaultStackInfo structure field, defined, 10-41

DefragmentECB, defined, 10-9, 11-8

DeleteMulticastAddress, defined, 18-6

DeregisterDefaultStackChain, defined, 10-10

DeregisterMLID, defined, 11-9

DeregisterPrescanRxChain, defined, 10-11

DeregisterPrescanTxChain, defined, 10-12

DeregisterRPLBootROM, defined, 10-13

DeregisterStack, defined, 10-14

destination, determining packet, 8-3

determining, packet destination, 1-5, 8-3

DMA, lookahead, 17-5

DMA keyword, E-4

DMALine, HSM configuration table field, defined, 15-11

document

Brouter supplement, 18-8  
 canonical and noncanonical addressing supplement, xi, 4-11, 6-6, 18-5, 18-13  
 frame types, xi, 2-5, 4-7, 6-6, 10-5, 11-4, A-5  
 hub management interface, xi, 18-8  
 installation information file, xi  
 MLID message definition, xi  
 NESL specification, xi, 4-8, 14-5  
 other, x  
 protocol IDs (PIDs), xi, 2-5, 4-7, 6-6, 10-5, 11-4, A-5  
 source routing, xi  
 supplement, x

DOS Environment, protocol stack, 2-2

DriverMajorVer, HSM configuration table field, defined, 15-9

DriverManagement, defined, 18-7

DriverMinorVer, HSM configuration table field, defined, 15-9

DriverPoll, defined, 18-9

DriverWorkspace, Event Control Block field, defined, A-5

## E

ECB. See Event Control Block (ECB), A-2

EndCriticalSection, defined, 10-15, 11-10

entry point  
 LSLGenSupEntryPt, 12-3  
 LSLMLIDSupEntryPt, 12-3  
 LSLProtSupEntryPt, 12-3

Event Control Block (ECB)  
 AESECB, defined, 10-52, 11-18  
 defined, A-3  
 overview, 6-3, A-2  
 ProtocolWorkspace field, 6-4  
 sample code, A-3  
 sending packets, 6-4  
 StackID ECB field, values of, 5-12  
 Status ECB field, values of, 5-12

EventServiceRoutine, Event Control Block field, defined, A-4

EventServiceRoutine field, 6-4

Explicit, 4-6

Extended Industry Standard Architecture (EISA) bus, 14-8

## F

FFrag?Address, FragmentStructure field, defined, 18-22

FFrag?Len, FragmentStructure field, defined, 18-22

FFrag0Address, FragmentStructure field, defined, 18-22

FFrag0Length, FragmentStructure field, defined, 18-22

FFragmentCount, FragmentStructure field, defined, 18-22

files, custom configuration, 8-5

filtering, stack, 2-7

Flags, HSM configuration table field, defined, 15-9

flags

- MFlags, bit map, 15-14
- MLIDModeFlags, OBR support, bits 14 and 15, 15-13
- MSharingFlags, bit map, 15-16

flow of data

- receive, 1-7
- send, 1-6

Fragment descriptors, defined, 6-7

Fragment?Address, Event Control Block field, defined, A-6

Fragment?Length, Event Control Block field, defined, A-6

FragmentCount, Event Control Block field, defined, A-5

FragmentCount field, 6-6

FragmentStructure, structure provided by MSM, defined, 18-22

frame

- data space, 15-3
- defined, 14-6
- supporting multiple types, 14-6
- type, relation of to logical board, 14-6

Frame keyword, E-6

Frame Reception. See Receiving packets, 5-2

Frame transmission. See Sending packets, 6-2

frame type, document, xi, 2-5, 4-7, 6-6, 10-5, 11-4, A-5

FrameTypeID, HSM configuration table field, defined, 15-7

FrameTypeString, HSM configuration table field, defined, 15-7

FreeMemory, defined, 13-8

## G

General Service Description Record, defined, 13-4

GenServiceControlBlock structure, 13-4

GetBoundBoardInfo, defined, 10-16

GetCriticalSectionStatus, defined, 10-17, 11-11

GetECB, defined, 10-18, 11-12

GetEntryPoints, defined, 12-3

GetHeldPacket, defined, 10-19

GetIntervalMarker, defined, 10-21, 11-13

GetLSLConfiguration, defined, 10-22

GetLSLStatistics, defined, 10-23

GetMLIDConfiguration, defined, 18-10

GetMLIDControlEntry, defined, 10-24

GetMLIDStatistics, defined, 18-11

GetMLIDSupportEntry, defined, 12-4

GetMulticastInfo, 18-12

GetNETCFGPath, defined, 13-9

GetPIDFromStackIDBoard, defined, 10-25

GetProtocolControlEntry, defined, 10-27

GetProtocolStackConfiguration, defined, 7-4

GetProtocolStackStatistics, defined, 7-5

GetProtocolSupportEntry, defined, 12-5

GetServiceChain, defined, 13-10

GetStackECB, defined, 11-14

GetStackIDFromName, defined, 10-28

GetStartOfChain, defined, 10-29

GetTickMarker, defined, 10-30

## H

hardware

- bus type, listed, 14-8
- data transfer, 14-8

Hardware/Media independence, 2-3

HoldEvent, defined, 10-31

HoldPacket, defined, 10-32

HoldReceiveEvent, defined, 11-16  
 hub management interface, document, xi, 18-8

## I

ImmediateAddress, Event Control Block field, defined, A-5  
 ImmediateAddress field, 6-6  
 Industry Standard Architecture (ISA) bus, 14-8  
 Initialization, protocol stack  
   overview, 4-2  
   process, 4-2  
 initializing, MLID, 14-4, 16-3  
 installation information file, document, xi  
 Interrupt handler, polling from, 6-3  
 IntLine, HSM configuration table field, defined, 15-11  
 IRQ keyword, E-4

## K

Keywords, NET.CFG file, E-4

## L

LAESEventsCount, LSL statistics table field defined, 9-6  
 LAN adapters, multiple allowed by ODI, 2-2  
 LBoardNum, Receive LookAhead structure field, defined, 5-4  
 LDataSize, Receive LookAhead structure field, defined, 5-4  
 LDestType field, xiii, 5-4  
 LECBCancelFailures, LSL statistics table field defined, 9-6  
 LECBCancelOK, LSL statistics table field defined, 9-6  
 LImmAddress field, 5-4  
 line speed, protocol stack, 4-9  
 LineSpeed, HSM configuration table field, defined, 15-8  
 Link Support Layer (LSL)  
   defined, 1-5, 8-3  
   locating, 4-3

  registering with, 4-5  
 Link Support Layer API entry points, locating, 4-3  
 LMediaHeaderPtr, Receive LookAhead structure field, defined, 5-3  
 LNumCustomCounters, LSL statistics table field defined, 9-6  
 LNumGenericCounters, LSL statistics table field defined, 9-5  
 loading the LSL, 4-3  
 Locating the LSL, locating LSL API entry points, 4-3  
 logical, board, relation of to frame type, 14-6  
 Logical board, defined, 2-2  
 logical board  
   binding to, 4-11  
   servicing, 4-6  
 Logical network, defined, 2-4  
 look ahead, 4-11  
 Look ahead data  
   configuring, 5-5  
   defined, 5-2  
 look ahead structure, xiii, 5-3, 11-14  
 lookahead  
   buffer, 17-4  
   DMA, 17-5  
   programmed I/O, 17-5  
   shared RAM, 17-5  
 LookAhead structure, defined, 5-3  
 LookAheadLen, Receive LookAhead structure field, defined, 5-3  
 LookAheadPtr, Receive LookAhead structure field, defined, 5-3  
 LookAheadSize, HSM configuration table field, defined, 15-8  
 LookAheadStruc structure, xiii, 5-3, 11-14  
 LPacketAttrib field, xiii, 5-4  
 LPostponedEvents, LSL statistics table field defined, 9-6  
 LPriorityLevel field, 5-5  
 LProtID, Receive LookAhead structure field, defined, 5-4  
 LSL, loading, 4-3  
 LSL API entry points, locating, 4-3  
 LSL commandline switches, 8-4

LSL completion codes, list of, 8-3

LSL protocol stack services

- BindStack, 10-7
- CancelAESEvent, 10-8
- DeregisterRPLBootROM, 10-13

LSLGenSupEntryPt entry point, 12-3

LSLInitEntryPointBlock structure, 12-3

LSLMajorVersion, LSL configuration table field, defined, 9-4

LSLMinorVersion, LSL configuration table field, defined, 9-4

LSLMLIDSupEntryPt entry point, 12-3

LSLProtSupEntryPt entry point, 12-3

LStartCopyOffset field, 5-5

LStatTableMajorVer, LSL statistics table field defined, 9-5

LStatTableMinorVer, LSL statistics table field defined, 9-5

LTotalRxPackets, LSL statistics table field defined, 9-6

LTotalTxPackets, LSL statistics table field defined, 9-6

LUnclaimedPackets, LSL statistics table field defined, 9-6

LValidCountersMask, LSL statistics table field defined, 9-6

lying send, 17-6

## M

MaxBoardsNum, LSL configuration table field, defined, 9-4

MaxDataSize, 4-9

Maximum packet size, configuring, 4-9

MaxPacketSize, HSM configuration table field, defined, 15-6

MaxStacksNum, LSL configuration table field, defined, 9-4

MBusID, HSM configuration table field, defined, 15-9

MEM keyword, E-4

MemoryAddress, HSM configuration table field, defined, 15-10

MemorySize, HSM configuration table field, defined, 15-10, 15-11

MemoryStatistics, defined, 13-11

MemStatStruc structure, 13-11

MFlags, bit map, 15-14

Micro Channel Architecture bus, 14-8

MIOAddress, HSM configuration table field defined, 15-10

MIORange

- HSM configuration table field defined, 15-10
- HSM configuration table field, defined, 15-10

MLI (Multiple Link Interface), defined, 1-6

MLID

- configuration table
- major version, 15-6
- minor version, 15-6
- statistics table, 15-18
- major version, 15-20
- minor version, 15-20

MLID (Multiple Link Interface Driver)

- configuration table, 15-3
- control routines, overview, 14-4
- defined, 1-5
- functions of, 16-3
- message definition, document, xi
- multiple frame support. *See* frame, supporting multiple types
- removing, overview, 14-5
- timeout detection, 14-5

MLIDDeregistered, defined, 7-6

MLIDInfoBlockStruc structure, xii, 16-3

MLIDReset, defined, 18-14

MLIDRetInfoBlockStruc structure, xii, 16-4

MLIDShutdown, defined, 18-15

MModeFlags, bit map, 15-12

MNoECBsAvailable, HSM statistics table field, defined, 15-20

MNumCustomCounters, HSM statistics table field, defined, 15-21

MNumGenericCounters, HSM statistics table field, defined, 15-20

ModeFlags, HSM configuration table field, defined, 15-6

ModifyStackFilter, defined, 10-33

MPrioritySup, HSM configuration table field, defined, 15-8

MQueueDepth, HSM statistics table field, defined, 15-21

MRxChecksumError, HSM statistics table field, defined, 15-20

MRxMiscCount, HSM statistics table field, defined, 15-20

MRxMismatchError, HSM statistics table field, defined, 15-20

MRxOverflow, HSM statistics table field, defined, 15-20

MRxTooSmall, HSM statistics table field, defined, 15-20

MSecondValue, AESECB field, defined, 10-52, 11-18

MSharingFlags, bit map, 15-16

MStatTableMajorVer, HSM statistics table field, defined, 15-20

MStatTableMinorVer, HSM statistics table field, defined, 15-20

MTotalRxPackets, HSM statistics table field, defined, 15-20

MTotalTxPackets, HSM statistics table field, defined, 15-20

MTxMiscError, HSM statistics table field, defined, 15-20

MTxRetry, HSM statistics table field, defined, 15-20

MTxTooBig, HSM statistics table field, defined, 15-20

MTxTooSmall, HSM statistics table field, defined, 15-20

multicast addressing, support, MLID, 14-8

Multicast transmission

- defined, 4-10
- Ethernet addresses, 18-4
- maximum number of addresses, 18-5
- media dependence of, 4-10
- NET.CFG file, 4-11
- specifying support of, 4-10
- Token-Ring addresses, 18-4
- unknown addresses, 4-11

multiple board support, protocol stack, 4-8

multiple frame support, MLID, 14-6

Multiple Link Interface Driver (MLID)

- configuration table, 14-5
- data structures, 14-5
- definition, 14-3
- design considerations, 14-8
- initializing, 14-4, 16-3
- multicast addressing support, 14-8
- multiple frame support, 14-6

packet reception, 17-3

packet transmission, 17-5

promiscuous mode support, 14-7

recommended functionality, 14-6

source routing support, 14-7

statistics table, 14-5

Multiple outstanding transmits, supporting, 6-3

Multiple Protocol Interface (MPI), defined, 1-4

Multiplexing protocol stacks, procedure, 2-4

Multitasking

- compatibility, B-2
- data buffer access, B-2
- running under Microsoft Windows, B-2

MValidCounterMask, HSM statistics table field, defined, 15-20

## N

NESL, document, xi, 4-8, 14-5

NET.CFG file

- "Bind" entry, 4-6
- custom configuration keywords, E-6
- example of, E-7
- keywords, LAN driver, E-4
- locating the directory, E-3
- main section headers, E-3
- minimum keyword required, E-6
- multicast addresses, 4-11
- parsing, E-3
- sample entry, E-3
- text in, E-3

Network Interface Card, utilization of, 2-2

Network logical. See Logical network, 2-4

NextLink, Event Control Block field, defined, A-4

NIC. See Network Interface Card, 2-2

NICLongName, HSM configuration table field, defined, 15-7

NICShortName, HSM configuration table field, defined, 15-7

Node Address keyword, E-5

NodeAddress, HSM configuration table field, defined, 15-6

non-lying send, 17-6

Novell, Inc., ii

## O

ODI, OSI, correspondence to, 2-2

ODI (Open Data-Link Interface) specification  
defined, 1-3  
illustrated, 1-3  
OSI, correspondence to, 1-3

ODI system, booting, 4-3

Outstanding transmit requests, number of, 6-3

## P

### Packet

receiving, overview, 2-4, 5-2  
receiving, process, 5-2  
sending, completion of, 6-7  
sending, overview, 2-4, 6-2  
sending, process, 6-3  
size, configuring maximum for MLID, 4-9  
transmitting, 6-2

### packet

destination, determining, 1-5, 8-3  
flow, 1-6  
reception, MLID, 17-3  
transmission, 14-4

Packet bursts, sending, 6-3

Packet receive routine, process, 5-2

packet reception, 2-6

### Packet reception methods

bound protocol stack, 5-6  
default protocol stack, 5-6  
prescan protocol stack, 5-6

### Packet send routine

process, 6-2, 6-3  
transmit complete, 6-7

Packet size, maximum size for MLID, configuring, 4-9

packet transmission, MLID, 17-5

Packets, sending bursts of, 6-3

PConfigTableMajorVer, protocol stack configuration table field defined, 3-3

PConfigTableMinorVer, protocol stack configuration table field defined, 3-3

Peripheral Component Interconnect (PCI) bus, 14-8

Personal Computer Memory Card International Association (PCMCIA) bus, 14-8

PID. See Protocol ID (PID), 2-4

PIgnoredRxPackets, protocol stack statistics table field, 3-5

PNumCustomCounters, protocol stack statistics table field, 3-5

PNumGenericCounters, protocol stack statistics table field, 3-4

Port keyword, E-4

PProtocolLongName, protocol stack configuration table field defined, 3-3

PProtocolMajorVer, protocol stack configuration table field defined, 3-3

PProtocolMinorVer, protocol stack configuration table field defined, 3-3

PProtocolReserved, protocol stack configuration table field defined, 3-3

PProtocolShortName, protocol stack configuration table field defined, 3-3

Prescan protocol stack, defined, 2-5, 5-6

prescan transmit protocol stack handler, 6-9

PreviousLink, Event Control Block field, defined, A-4

priority packet support, 2-6

priority sends, 2-6

Programmed I/O, lookahead, 17-5

### promiscuous mode

defined, 14-7  
support, MLID, 14-7

PromiscuousChange, defined, 18-17

### Protocol ID (PID)

adding, conditions for, 4-8  
adding, procedure for, 4-7  
bytes defined, C-3  
defined, 2-4  
location and format of in frame header, 2-4  
obtaining, 4-8  
overriding, 4-8

Protocol IDs, adding, 4-7

Protocol keyword, E-5

protocol receive complete handler, 5-11

protocol receive handler, 5-9

### Protocol stack

bound, 2-4, 5-6  
default, 2-2, 2-5, 5-6  
initializing, 4-2  
minimum environment required, 2-2  
minimum number of LAN adapters, 2-2  
multiplexing, 2-4

performance, 2-2  
 prescan, 2-5, 5-6  
 unloading, 4-12  
 utilizing multiple boards, 2-4  
 protocol stack  
   configuration table, 3-3  
   customizing, 4-8  
   defined, 1-3  
   handler, prescan transmit, 6-9  
   line speed, 4-9  
   measuring performance, 4-9  
   multiple board support, 4-8  
   statistics table, 3-4  
   transport time, 4-9  
 protocol transmit complete handler, 6-8  
 ProtocolConfigStructure, structure, 3-3  
 ProtocolID  
   802.2 header, 6-5  
   Event Control Block field, defined, A-5  
 ProtocolID field, 6-5  
 ProtocolManagement, defined, 7-7  
 ProtocolPromiscuousChange, defined, 7-9  
 ProtocolReceiveCompleteHandler, ECB fields to set, 5-11  
 ProtocolReceiveHandler, ECB fields to set, 5-10  
 ProtocolTransmitCompleteHandler  
   Status ECB field, setting, 6-8  
   transmitting from, 6-8  
 ProtocolWorkspace  
   Event Control Block field, defined, A-5  
   using, 6-3  
 PStatTableMajorVer, protocol stack statistics table field, 3-4  
 PStatTableMinorVer, protocol stack statistics table field, 3-4  
 PTotalRxPackets, protocol stack statistics table field, 3-4  
 PTotalTxPackets, protocol stack statistics table field, 3-4  
 PValidCountersMask, protocol stack statistics table field defined, 3-4

## Q

QueueDepth, HSM configuration table field, defined, 15-8

## R

Raw send  
   addressing, 6-5  
   defined, 6-4  
   requirements, 6-5  
   specifying, 6-4  
 ReallocateMemory, defined, 13-12  
 receive complete handler, protocol, 5-11  
 receive handler, protocol, 5-9  
 receive look ahead, 4-11  
 Receive Look Ahead structure, 5-3  
 Receive Look Ahead, defined, 5-2  
 receive lookahead, buffer, 17-4  
 Receive routine, 5-2  
 Receiving frames. See Receiving packets, 5-2  
 Receiving packets  
   ECB fields to set, 5-10, 5-11  
   overview, 2-4, 5-2  
   process, 5-2  
   protocol stack's role in, 2-4, 5-6  
   receive routine, 6-2  
 RegisterDefaultStackChain, defined, 10-34  
 registering  
   MLID, with LSL, 16-3  
   transmit monitor, 18-20  
 Registering with LSL  
   bound protocol stack, 4-5  
   default protocol stack, 4-5  
   prescan protocol stack, 4-5  
   receive and control handlers role in, 4-5  
   Stack ID field's role in, 4-5  
 registering with the LSL, 4-5  
 RegisterMLID, defined, 12-6  
 RegisterPrescanRxChain, defined, 10-36  
 RegisterPrescanTxChain, defined, 10-38  
 RegisterRPLBootROM, defined, 10-40  
 RegisterStack, defined, 10-42  
 RegisterTxMonitor, defined, 18-20  
 RelinquishControl, defined, 10-44  
 RemoveGeneralService, defined, 13-13  
 removing, MLID, overview, 14-5  
 required control procedures, 14-3  
 ResubmitDefault, defined, 10-45  
 ResubmitPrescanRx, defined, 10-47

ResubmitPrescanTx, defined, 10-49  
ReturnECB, defined, 10-50, 11-17  
RPLBootROMInfoStruc structure, 10-41  
RxTooBigCount, HSM statistics table field, defined, 15-20

## S

ScanPacket, defined, 10-51  
ScheduleAESEvent, defined, 10-52, 11-18  
SendComplete, defined, 11-22  
Sending packets  
  ECB fields to set, 6-4  
  ECB's role in, 6-2  
  ECBs provided by protocol stack, 6-3  
  outstanding transmit requests, 6-3  
  overview, 6-2  
  packet bursts, 6-3  
  polling for transmit complete, 6-3  
  process, 6-2  
  protocol stack's role in, 2-4  
  transmit complete, 6-7  
SendPacket, defined, 10-54  
SendRetries, HSM configuration table field, defined, 15-9  
ServiceEvents, defined, 10-55, 11-20  
servicing, logical board, 4-6  
SetLookAheadSize  
  configuring look ahead data, 5-5  
  defined, 18-19  
Shared RAM, lookahead, 17-5  
SharingFlags, HSM configuration table field, defined, 15-9  
Signature, HSM configuration table field, defined, 15-6  
Slot, HSM configuration table field, defined, 15-9  
Slot keyword, E-5  
source routing, document, xi  
source routing support, MLID, 14-7  
SourceRouteHandler, HSM configuration table field, defined, 15-8  
specification version number, 8-4  
specification version string, 8-4  
stacj filtering, 2-7

StackChainStruc structure, xii, 5-7  
StackID, Event Control Block field, defined, A-4  
StackID field, 6-4  
StackInfoStruc structure, 10-42  
StartCriticalSection, defined, 10-56, 11-21  
statistics counter  
  custom, 15-17  
  generic, 15-17  
statistics table  
  HSM, sample code, 15-18  
  MLID, 14-5, 15-18  
  major version, 15-20  
  minor version, 15-20  
  protocol stack, 3-4  
Status, Event Control Block field, defined, A-4  
Status ECB field, defined, 5-12  
structure  
  AESECB, 10-52, 11-18  
  GenServiceControlBlock, 13-4  
  LookAheadStruc, xiii, 5-3, 11-14  
  LSLInitEntryPointBlock, 12-3  
  MemStatStruc, 13-11  
  RPLBootROMInfoStruc, 10-41  
  StackChainStruc, xii, 5-7  
  StackInfoStruc, 10-42  
  TCB, 18-21  
structures provided by MSM, FragmentStructure, 18-22  
switches, LSL commandline, 8-4

## T

TCB (Transmit Control Block), defined, 18-21  
TCBDataLength, TCB field, defined, 18-21  
TCBDriverWS, TCB field, defined, 18-21  
TCBFragStrucPtr, TCB field, defined, 18-22  
TCBMediaHeader, TCB field, defined, 18-22  
TCBMediaHeaderLen, TCB field, defined, 18-22  
timeout, MLID, overview, 14-5  
timeout detection, MLID, 14-5  
trademark, ii  
transmit complete handler, protocol, 6-8  
Transmit complete, process, 6-7  
Transmit monitor, defined, 18-21

transmit monitor, registering, 18-20

transmitting, packets, 14-4

TransportTime, HSM configuration table field,  
defined, 15-7

## **U**

UnbindFromMLID, defined, 7-10

UnbindStack, defined, 10-57

Unloading protocol stack, algorithm for, 4-12

## **V**

version string, specification, 8-4

## **W**

WorstDataSize, 4-10

HSM configuration table field, defined, 15-7

