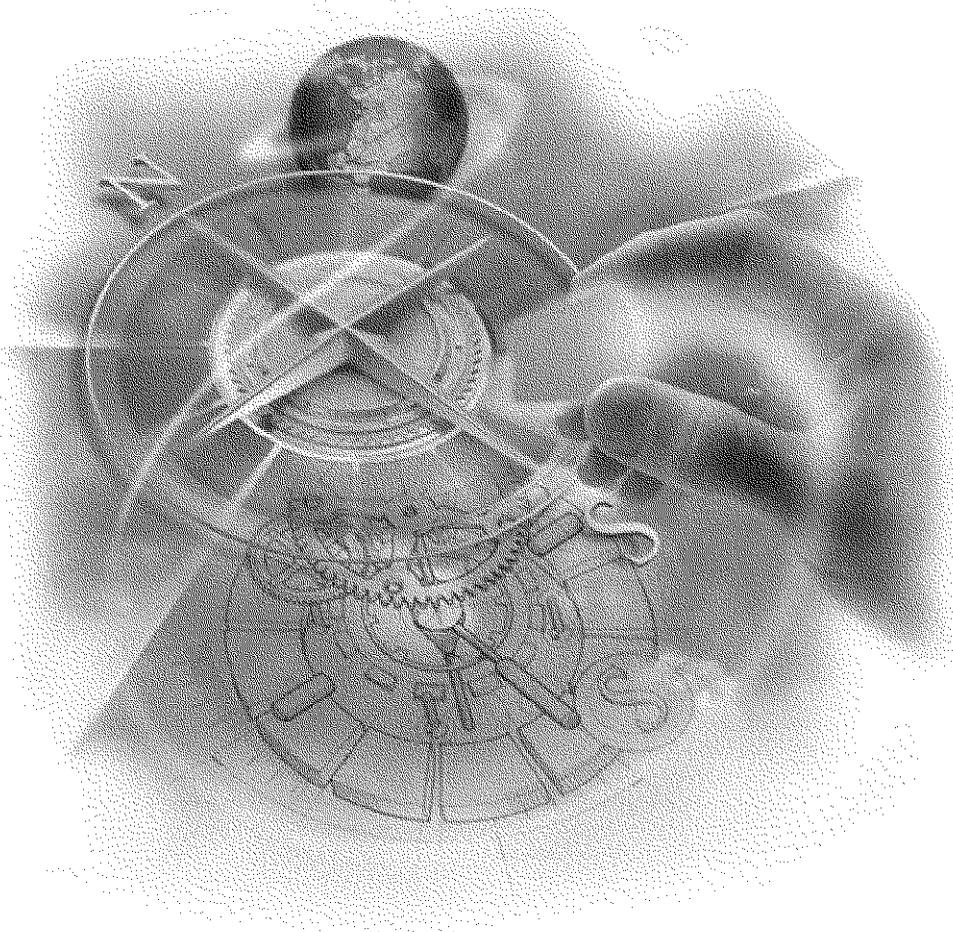


ODI™ SPECIFICATION:

Protocol Stacks and MLIDs™

(Intel 32-Bit Assembly Language)



Novell Developer Kit

Novell®

Legal Notices

Novell, Inc. makes no representations or warranties with respect to the contents or use of this documentation, 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.

Further, Novell, Inc. makes no representations or warranties with respect to any software, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc. reserves the right to make changes to any and all parts of Novell software, at any time, without any obligation to notify any person or entity of such changes.

This product may require export authorization from the U.S. Department of Commerce prior to exporting from the U.S. or Canada.

Copyright © 1993-2000 Novell, Inc. All rights reserved. No part of this publication may be reproduced, photocopied, stored on a retrieval system, or transmitted without the express written consent of the publisher.

U.S. Patent Nos 5,553,139; 5,553,143; 5,677,851; 5,758,069; 5,784,560; 5,818,936; 5,864,865; 5,903,650; 5,905,860; 5,910,803 and other Patents Pending.

Novell, Inc.
122 East 1700 South
Provo, UT 84606
U.S.A.

www.novell.com

Protocol Stacks and MLIDs (Intel 32-bit Assembly Language)
May 2000
104-000194-001

Online Documentation: To access the online documentation for this and other Novell developer products, and to get updates, see developer.novell.com/ndk. To access online documentation for Novell products, see www.novell.com/documentation.

Novell Trademarks

AppNotes is a registered trademark of Novell, Inc.

AppTester is a trademark of Novell, Inc., in the United States.

ArcNet 68 is a trademark of Novell, Inc.

BorderManager is a trademark of Novell, Inc.

C3PO is a trademark of Novell, Inc.

Client 32 is a trademark of Novell, Inc.

ConsoleOne is a trademark of Novell, Inc.

Controlled Access Printer is a trademark of Novell, Inc.

Custom 3rd-Party Object is a trademark of Novell, Inc.

DeveloperNet is a registered trademark of Novell, Inc.

DeveloperNet 2000 is a trademark of Novell, Inc.

GroupWise is a registered trademark of Novell, Inc., in the United States and other countries.

GroupWise 5 is a trademark of Novell, Inc.

Hardware Specific Module is a trademark of Novell, Inc.

HostPublisher is a trademark of Novell, Inc.

Hot Fix is a trademark of Novell, Inc.

HSM is a trademark of Novell, Inc.

InForms is a trademark of Novell, Inc.

Internetwork Packet Exchange is a trademark of Novell, Inc.

IPX is a trademark of Novell, Inc.

IPX/SPX is a trademark of Novell, Inc.

LANalyzer is a registered trademark of Novell, Inc., in the United States and other countries.

Link Support Layer is a trademark of Novell, Inc.

LSL is a trademark of Novell, Inc.

ManageWise is a registered trademark of Novell, Inc., in the United States and other countries.

Mirrored Server Link is a trademark of Novell, Inc.

MLI is a trademark of Novell, Inc.

MLID is a trademark of Novell, Inc.

MSL is a trademark of Novell, Inc.

Multiple Link Interface is a trademark of Novell, Inc.

Multiple Link Interface Driver is a trademark of Novell, Inc.

NControl is a trademark of Novell, Inc.

NCP is a trademark of Novell, Inc.

NDebug is a trademark of Novell, Inc.

NDPS is a registered trademark of Novell, Inc.

NDR is a trademark of Novell, Inc.

NDS is a registered trademark of Novell, Inc in the United States and other countries.

NDS Manager is a trademark of Novell, Inc.

NetWare is a registered trademark of Novell, Inc., in the United States and other countries.

NetWare 386 is a trademark of Novell, Inc.

NetWare Aware is a trademark of Novell, Inc.

NetWare Connect is a registered trademark of Novell, Inc, in the United States.

NetWare Core Protocol is a trademark of Novell, Inc.

NetWare DOS Requester is a trademark of Novell, Inc.

NetWare Loadable Module is a trademark of Novell, Inc.

NetWare Management Portal is a trademark of Novell, Inc.

NetWare MHS is a trademark of Novell, Inc.

NetWare Name Service is a trademark of Novell, Inc.

NetWare Peripheral Architecture is a trademark of Novell, Inc.

NetWare Print Server is a trademark of Novell, Inc.

NetWare Requester is a trademark of Novell, Inc.

NetWare SFT and NetWare SFT III are trademarks of Novell, Inc.

NetWare SQL is a trademark of Novell, Inc.

NetWare Telephony Services is a trademark of Novell, Inc.

NetWare Tools is a trademark of Novell, Inc.

NLM is a trademark of Novell, Inc.

Novell is a registered trademark of Novell, Inc., in the United States and other countries.

Novell Application Launcher is a trademark of Novell, Inc.

Novell Authorized Service Center is a service mark of Novell, Inc.

Novell BorderManager is a trademark of Novell, Inc.

Novell Certificate Server is a trademark of Novell, Inc.

Novell Client is a trademark of Novell, Inc.

Novell Directory Services is a registered trademark of Novell, Inc.

Novell Distributed Print Services is a trademark of Novell, Inc.

Novell Embedded Systems Technology is a registered trademark of Novell, Inc., in the United States and other countries.

Novell HostPublisher is a trademark of Novell, Inc.

Novell, Yes, Tested & Approved logo is a trademark of Novell, Inc.

ODI is a trademark of Novell, Inc.

Open Data-Link Interface is a trademark of Novell, Inc.

Packet Burst is a trademark of Novell, Inc.

Personal NetWare is a trademark of Novell, Inc.

Printer Agent is a trademark of Novell, Inc.

Public Access Printer is a trademark of Novell, Inc.

QuickFinder is a trademark of Novell, Inc.

Remote Console is a trademark of Novell, Inc.

RX-Net is a trademark of Novell, Inc.

Sequenced Packet Exchange is a trademark of Novell, Inc.

SFT, SFT III, and SFT NetWare are trademarks of Novell, Inc.

SMS is a trademark of Novell, Inc.

SMSTSA is a trademark of Novell, Inc.

SoftSolutions is a registered trademark of SoftSolutions Technology Corporation, a wholly owned subsidiary of Novell, Inc.

SPX is a trademark of Novell, Inc.

Storage Management Services is a trademark of Novell, Inc.

SVR4 is a trademark of Novell, Inc.

System V is a trademark of Novell, Inc.

Topology Specific Module is a trademark of Novell, Inc.

Transaction Tracking System is a trademark of Novell, Inc.

TSM is a trademark of Novell, Inc.

TTS is a trademark of Novell, Inc.

Universal Component System is a trademark of Novell, Inc.

ViewMAX is a trademark of Novell, Inc.

ZENworks is a trademark of Novell, Inc.

Third-Party Trademarks

All third-party trademarks are the property of their respective owners.

Java is a trademark or registered trademark of Sun Microsystems, Inc., in the United States and other countries.

Contents

Contents

Preface

Overview	19
Introduction	19
Protocol Stacks	19
Link Support Layer (LSL)	20
Multiple Link Interface Driver (MLID)	20
Appendices	20
References	20
Prerequisites	21
Manual Conventions	21

1 Introduction to ODI

Chapter Overview	23
Open Data-Link Interface (ODI)	23
Protocol Stacks	24
Multiple Protocol Interface (MPI)	25
Link Support Layer (LSL)	26
Multiple Link Interface Drivers (MLIDs)	26
Data Flow	27

2 ODI Module Design

Chapter Overview	31
NetWare Environment	31
NetWare Loadable Modules (NLMs)	31
NLM Design and Programming Issues	32
Memory Protection	38
Memory Management	39
Hardware/Media Independence	40
Development Process	40
Related Files	41

3 Overview of Protocol Stacks

Chapter Overview	43
The NetWare Server Protocol Stack	43

Protocol Stack Multiplexing	43
Packet Flow with Multiple Protocol Stacks	46

4 Protocol Stack Data Structures

Chapter Overview	53
Protocol Stack Configuration Table	53
Protocol Stack Statistics Table	54
Event Control Blocks	57
Receive Event Control Block	58
Transmit Event Control Block	60
Event Control Block Field Descriptions	62

5 Protocol Stack Initialization

Chapter Overview	67
Protocol Stack Initialization	67
LAN Boards and Auto-binding	68
Binding	68
Initialization	70

6 Protocol Stack Packet Reception

Chapter Overview	79
Protocol Stack Packet Receive Operation	79
Protocol Stack Promiscuous Mode	79
Receive Routine Events	80
Prescan and Default Protocol Stack Packet Reception	80
Receive ECBs	82
The Protocol Stack Receive Handler	82
Setting the ECB BLink Field	84
Setting the ECB DriverWorkspace Field	85
Chained Protocol Stacks and Resubmission	87

7 Protocol Stack Packet Transmission

Chapter Overview	89
Protocol Stack Packet Transmission	89
Transmission Routine Events	89
Starting the Packet Transmission	90
Supporting Multiple Outstanding Transmission Requests	90
Sending the Packet	91
Handling a Transmit Event Control Block	92
Raw Sends	94
The Prescan Protocol Stack Transmission Handler	97

8 Protocol Stacks and MLIDs (Intel 32-bit Assembly Language)

Chained Prescan Transmission Protocol Stacks and Resubmission	98
Transmission Complete	99
Protocol Transmission Complete Handler.	99

8 Protocol Stack Control Routines

Ctl0_GetProtocolStackConfiguration	103
Ctl1_GetProtocolStackStatistics.	105
Ctl2_Bind	107
Ctl3_Unbind	109
Ctl4_MLIDDeRegistered	111
Ctl5_ProtocolPromiscuousChange	113
Ctl100_GetProtocolStringForBoard	115
Ctl101_GetBoundNetworkInfo	117

9 Overview of the LSL

Chapter Overview	119
Link Support Layer (LSL)	119

10 The LSL Statistics Table

Chapter Overview	121
LSL Statistics Table	121
The LSL Logical Board Statistics Structure	124

11 LSL Support Routines (Assembly Language)

LSLAdapterMutexLock	133
LSLAdapterMutexTryLock.	135
LSLAdapterMutexUnlock	137
LSLAddPollingProcedure	138
LSLAddProtocolID.	140
LSLAddTimerProcedure.	143
LSLAllocatePhysicalBoardID	145
LSLAssignMutexToInstance.	147
LSLBindStack	149
LSLControlStackFilter	152
LSLDeFragmentECB	154
LSLDeRegisterDefaultChain	156
LSLDeRegisterMLID	158
LSLDeRegisterPreScanRxChain	160
LSLDeRegisterPreScanTxChain	162
LSLDeRegisterStack	164
LSLDeRegisterStackSMPSafe	166

LSLFastRcvEvent	168
LSLFastSendComplete.	172
LSLFreePhysicalBoardID	173
LSLGetBoundBoardInfo	174
LSLGetLinkSupportStatistics	176
LSLGetMaximumPacketSize	177
LSLGetMLIDControlEntry	178
LSLGetMultipleSizedRcvECBRTag	180
LSLGetPhysicalAddressOfECB	182
LSLGetPIDFromStackIDBoard	183
LSLGetProtocolControlEntry	185
LSLGetSizedRcvECBRTag	187
LSLGetStackIDFromName	189
LSLGetStartChain	191
LSLHoldRcvEvent	194
LSLModifyStackFilter	198
LSLRegisterDefaultChain	200
LSLRegisterMLIDRTag.	203
LSLRegisterPreScanRxChain	205
LSLRegisterPreScanTxChain	208
LSLRegisterStackRTag	211
LSLRegisterStackSMPSafe	213
LSLRemoveMutexFromInstance	215
LSLRemovePhysicalMutex.	217
LSLRemovePollingProcedure	219
LSLRemoveTimerProcedure	221
LSLReSubmitDefaultECB	223
LSLReSubmitPreScanRxECB	225
LSLReSubmitPreScanTxECB	226
LSLReturnRcvECB	227
LSLSendComplete	228
LSLSendPacket	229
LSLServiceEvents	233
LSLSMPGetSendQ.	235
LSLSMPReaderLock	237
LSLSMPReaderToWriterLock	239
LSLSMPReaderUnLock	241
LSLSMPWriterLock	243
LSLSMPWritertoReaderLock.	245
LSLSMPWriterUnLock	247
LSLUnbindStack	249
LSLUnBindThenDeRegisterMLID	251

12 LSL Support Routines (C Language)

CLSLAddProtocolID	255
CLSLBindStack	257
CLSLControlStackFilter	259
CLSLDeRegisterDefaultChain	261
CLSLDeRegisterPreScanRxChain	263
CLSLDeRegisterPreScanTxChain	265
CLSLDeRegisterStack	267
CLSLGetBoundBoardInfo	269
CLSLGetMLIDControlEntry	271
CLSLGetPIDFromStackIDBoard	273
CLSLGetProtocolControlEntry	275
CLSLGetStackIDFromName	277
CLSLGetStartChain	279
CLSLModifyStackFilter	281
CLSLRegisterDefaultChain	283
CLSLRegisterPreScanRxChain	286
CLSLRegisterPreScanTxChain	289
CLSLRegisterStackRTag	292
CLSLReSubmitDefaultECB	294
CLSLReSubmitPreScanRxECB	295
CLSLReSubmitPreScanTxECB	296
CLSLReturnRcvECB	297
CLSLSendPacket	298
CLSLUnbindStack	302

13 Overview of the MLID

Chapter Overview	303
The NetWare Server MLID	303
MLID Procedures	303
MLID Initialization	305
Board Service Routine	305
Packet Transmission	306
Multiple Operating System Support	306
Control Procedures	306
Timeout Detection	306
Driver Remove	306
MLID Data Structures and Variables	307
Configuration Table	307
Statistics Table	307
MLID Functionality	307
Reentrancy	307
Multiple Frame Support	308
Multiple Frame Support in Reentrant Code	308

MLID Design Considerations	313
--------------------------------------	-----

14 MLID Data Structures

Chapter Overview	315
Frame Data Space	315
The MLID Configuration Table	315
Configuration Table Flags	327
Deriving the Maximum Packet Size	332
Adapter Data Space	333
MLID Statistics Table	334
MLID Statistics Table Field Descriptions	336
CounterMask Bit Maps	340
Topology-specific Counters	342
Event Control Blocks	350
Receive Event Control Block	351
Transmit Event Control Block	352
Event Control Block Field Descriptions	353
Driver Firmware	358
Reading Driver Firmware: Example Code	358

15 The MLID Initialization Routine

Chapter Overview	361
The MLID Initialization Routine	361
Loading the MLID	362
Requirements of the Calling Routine	362
Initialization Parameters Passed on the Stack	362
Adapter Data Space and Frame Data Space	364
Resource Tags	364
Determining Hardware Options	366
Parsing the Command Line	366
Registering Hardware Options	368
Setting Up A Board Service Routine	368
Initializing the LAN Adapter	369
Registering with the LSL	370
Scheduling a Hardware Time Out Check	370
Error Handling	371
Pseudocode for DriverInitialize	371

16 The MLID Packet Reception Routine

Chapter Overview	377
The Packet Reception Routine	377
Reception Methods	378

12 Protocol Stacks and MLIDs (Intel 32-bit Assembly Language)

Front Ends for the Board Service Routine	381
The Board Service Routine	383
Handling Receive Errors	391
Transmission Complete Interrupt	392
Transmission Error	392
Using Shared Interrupts	392
Pseudocode for the Board Service Routine	393

17 The MLID Packet Transmission Routine

Chapter Overview	397
Packet Transmission	397
General Transmission Method	397
Sending a Packet	398
Queuing Sends	398
Multiple Frame Support	399
Raw Sends	399
Priority Sends	399
Packet Length	399
Pseudocode for MLID Packet Transmission Routine	400
Pseudocode for Packet Transmission Routine for RX-Net MLIDs	401

18 MLID Timeout Procedure

Chapter Overview	403
Establishing a Timeout Procedure	403
Scheduling an Interrupt Time Callback	403
Determining the Wait Interval	404
Identifying a Timeout Error	404
Using System Alerts	404
Reinitializing the LAN adapter	404
Pseudocode for TimeOutCheck	405
Pseudocode for the Timeout Procedure for RX-Net MLIDs	406

19 Remove MLID Procedure

Chapter Overview	407
Removing the MLID	407
DeRegistering Logical Boards	407
Canceling Polling Procedures and Timer Events	408
Shutting Down the LAN Adapter	408
Removing Data Spaces	408
Pseudocode for Remove MLID	409

20 MLID Control Routines

Preliminary Information	412
Ctl0_GetMLIDConfiguration	416
Ctl1_GetMLIDStatistics	418
Ctl2_AddMulticastAddress	420
Ctl3_DeleteMulticastAddress	424
Ctl4_Reserved	428
Ctl5_MLIDShutdown	429
Ctl6_MLIDReset	431
Ctl7_Reserved	433
Ctl8_Reserved	434
Ctl9_SetLookAheadSize	435
Ctl10_MLIDPromiscuousChange	437
Ctl11_RegisterMonitor	440
Ctl12_Reserved	442
Ctl13_Reserved	443
Ctl14_Driver Management	444
Ctl15_Reserved	447
Ctl16_RemoveNetworkInterface	448
Ctl17_ShutdownNetworkInterface	449
Ctl18_ResetNetworkInterface	451
Pseudocode for DriverControl	452

21 Operating System Support Routines

AddPollingProcedureRTag	466
Alloc	468
AllocateMappedPages	470
AllocateResourceTag	473
AllocBufferBelow16Meg	476
AllocNonMovableCacheMemory	478
BindProtocolToBoard	480
BusInterruptClear	482
BusInterruptEOI	483
BusInterruptSetup	484
CancelInterruptTimeCallBack	488
CancelNoSleepAESProcessEvent	490
CancelSleepAESProcessEvent	492
CFindResourceTag	494
ClearHardwareInterrupt	496
ClearSymmetricInterrupt	498
CPSemaphore	500
CRescheduleLast	502
CYieldWithDelay	503
CVSemaphore	504

DeAllocateMappedPages	505
DeRegisterHardwareOptions	506
DisableHardwareInterrupt	508
DoEndOfInterrupt	510
DoRealModeInterrupt	512
EnableHardwareInterrupt	515
Free	517
FreeBufferBelow16Meg	519
FreeNonMovableCacheMemory	521
GetCurrentTime	522
GetFileServerMajorVersionNumber	523
GetFileServerMinorVersionNumber	524
GetHardwareBusType	525
GetNumberOfLANs	527
GetRealModeWorkSpace	528
GetServerConfigurationType	532
GetSuperHighResolutionTimer	533
ImportPublicSymbol	534
MapAbsoluteAddressToDataOffset	536
MapDataOffsetToAbsoluteAddress	538
NetWareAlert	540
NVMAlloc	546
NVMAllocIO	548
NVMFree	551
OutputToScreen	552
ParseDriverParameters	554
QueueSystemAlert	561
ReadEISAConfig	565
ReadPhysicalMemory	567
ReadRoutine	569
RegisterForEventNotification	572
RegisterHardwareOptions	577
RemovePollingProcedure	579
ScheduleInterruptTimeCallBack	580
ScheduleNoSleepAESProcessEvent	582
ScheduleSleepAESProcessEvent	584
SetHardwareInterrupt	586
SetSymmetricInterrupt	589
SMPDoEndOfInterrupt	592
UnRegisterEventNotification	593
WritePhysicalMemory	595

22 Assembling and Linking NLMs

Overview	597
NetWare Loadable Modules (NLMS)	597
Creating an NLM	598
The NetWare Linker	599
The Definition File	599
Loading and Unloading.	604

23 Debugging NLMs

Overview	605
The NetWare Debugger	605
Setting Breakpoints.	606
Changing Memory	608
Dumping Memory	609
Register Manipulations.	610
I/O	611
Miscellaneous	611
Grouping Operators	615
Unary Operators	615
Ternary Operator.	616

24 Server Command Line Parameters and Keywords

Overview	617
MLID Keywords	617
DMA	617
SLOT	618
PORT.	618
MEMORY ADDRESS	618
MEMORY LENGTH	618
INTERRUPT NUMBER.	618
NODE	619
RETRIES.	619
FRAME.	619

25 Writing Protocol Stacks for NetWare SFT III

Overview	621
Introduction to NetWare SFT III	621
Mirrored Server Implementation	621
Primary and Secondary Servers	622
MSEngine and IOEngine	622
Events and Requests on Mirrored Servers.	623
Mirrored Servers and PC Clients.	624
NetWare SFT III and Existing Applications.	625

Protocol Stacks and NetWare SFT III	625
NetWare SFT III Basic Architecture	625
Inter-Engine Support Layer	625
Protocol Stacks and the Inter-Engine Support Layer	626
The Protocol Stack NLM	627
Additional Protocol Stack Capabilities	627
IPX Protocol Stack Communication	628
Developing Protocol Stacks for SFT III	628
Protocol IDs for the VIRTUAL_LAN Frame Type	628
Nonrouting Protocol Stacks on SFT III	629
LSL Routines, IOCTLs, and OS Routines for SFT 11 Protocol Stacks	631
LSLSendProtocolInfoToOtherEngine	632
LSLSendProtocolInfoToPartner	633
Ctl6SFTIIIExchange	635

26 Revision History

May 2000 Release - Doc Version 1.21	639
---	-----

27 Glossary

Preface

This is Spec Version 4.10, Doc Version 1.21.

Overview

This document describes the ODI specification in the NetWare server environment and tells you how to write protocol stacks, and network communications drivers (LAN drivers) for NetWare file servers (NetWare 3 and NetWare 4). ODI allows multiple protocols to operate in the NetWare 3 and NetWare 4, DOS/WIN, OS/2, WIN95, and NT environments. Writing a LAN driver that conforms to the ODI specification ensures compatibility with any protocol that is also written to the ODI specification (for example, TCP/IP, ISO, IPX, etc.).

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

Introduction

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

Protocol Stacks

Explains the architecture of an ODI protocol stack and discusses the design issues relevant to an ODI protocol stack for the NetWare server. This section

also discusses protocol stack data structures, initialization, packet reception, and transmission and control routines.

Link Support Layer (LSL)

Presents a brief overview of the LSL and describes its statistics table. 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.

Multiple Link Interface Driver (MLID)

Explains the architecture of an ODI MLID and discusses the design issues relevant to an ODI MLID for the NetWare server. This section also discusses MLID data structures, initialization, packet reception and transmission, MLID unloading, time-out, and the control routines.

Appendices

Contains information that is sometimes necessary to build an ODI MLID. This section also provides more information about particular subjects. The appendix section contains information such as operating system support routines that the MLID might need to access, NLM assembling and linking, and the NetWare debugger.

References

This document refers to the following ODI Specification Supplements:

- ◆ The MLID Installation Information File
- ◆ The Hub Management Interface
- ◆ Source Routing
- ◆ Canonical and Noncanonical Addressing
- ◆ Frame Types and Protocol IDs
- ◆ Standard MLID Message Definitions

Prerequisites

The NetWare 3 and NetWare 4 operating systems run in both real mode and protected mode. MLID and protocol stack developers must understand protected mode issues.

Manual Conventions

All numbers in this document are decimal unless noted otherwise. Hexadecimal numbers are followed by a lowercase h (for example, 00FFh). Where byte fields are defined, byte 0 is assumed to be the low-order bit.

The data types in this document are defined as follows:

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

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

Table 1 How Bytes Are Stored in Memory (Example Number: 123456788h) – Low-high Order

Byte	Address	Number
Least Significant Byte	0000001Ah	78
	0000001Bh	56
	0000001Ch	34
Most Significant Byte	0000001Dh	12

Table 2 How Bytes Are Stored in Memory (Example Number: 123456788h) – High-low Order

Byte	Address	Number
Most Significant Byte	0000001Ah	12

Byte	Address	Number
	0000001Bh	34
	0000001Ch	56
Least Significant Byte	0000001Dh	78

1

Introduction to ODI

Chapter Overview

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

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

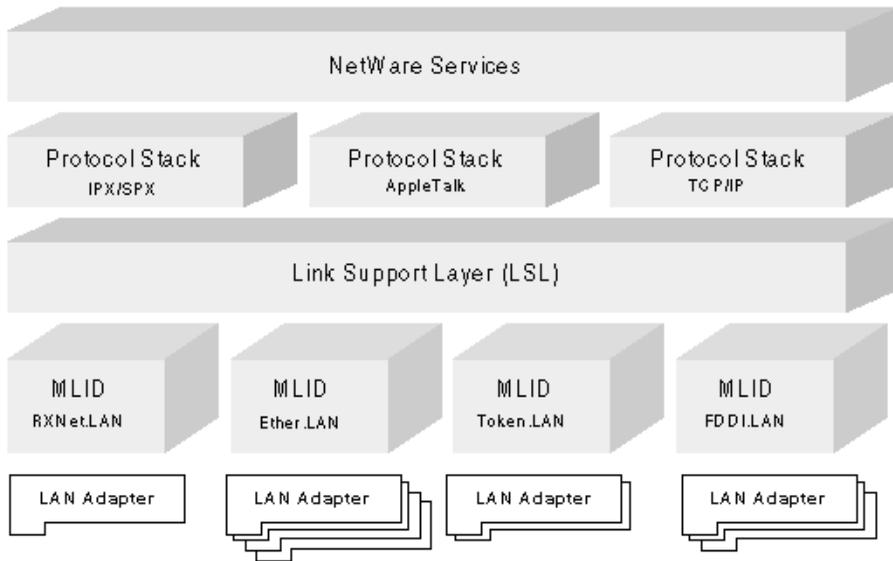
Because the ODI specification provides for communications between a variety of protocols and media, LAN drivers are called Multiple Link Interface Drivers (MLIDs). The Link Support Layer (LSL) handles the transfer of information between MLIDs and protocol stacks.

The terms MLID and LAN driver are interchangeable.

Open Data-Link Interface (ODI)

NetWare server MLIDs and protocol stacks must conform to the ODI specification. Figure 1.1 shows the elements that make up the ODI specification.

Figure 1.1: ODI Elements



The ODI specification allows multiple network protocols and LAN adapters (physical boards) to be used concurrently on the same workstation or file server. It also allows any MLID that supports more than one MAC packet header format (frame type) to do so by making a single physical LAN adapter appear to be more than one logical LAN adapter (logical board). 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

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 Open Systems Interconnection (OSI) Network Layer functionality; however, they are not limited to this. Figure 1.2 illustrates the ODI/OSI correspondence.

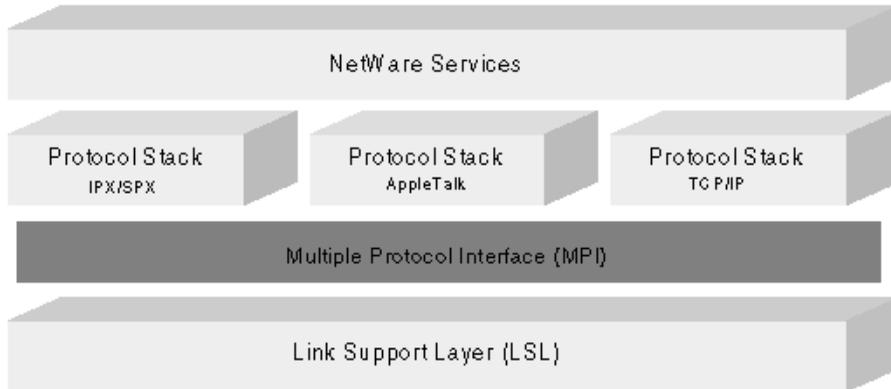
Figure 1.2: How ODI Fits into the OSI Model

OSI Model	ODI Model
Application	
Presentation	
Session	
Transport	Protocol Stack
Network	
Data Link	Link Support Layer
Logical Link Control (LLC)	
Media Access Control (MAC)	MLID
Physical	LAN Adapter

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. However, protocol stacks written to ODI Specification 3 and later also have full access to the NLM APIs documented in the *NetWare SDK NLM Programming Reference Manual*.

Figure 1.3: 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 determines which protocol stack is to receive the packet. Then, the protocol stack determines where the packet should be sent.

When the protocol stack transmits a packet, it hands the packet to the LSL. The LSL then routes the packet to the appropriate MLID. The LSL also tracks the various protocols and MLIDs that are currently loaded in the system and provides a consistent method of finding and using each of the loaded modules.

Multiple Link Interface Drivers (MLIDs)

Functionality

MLIDs are device drivers that send and receive packets to and from the physical layer or logical topology. (For example, Ethernet SNAP is a logical topology.) The MLID interfaces with a physical board called a Network Interface Card (NIC) or LAN adapter.

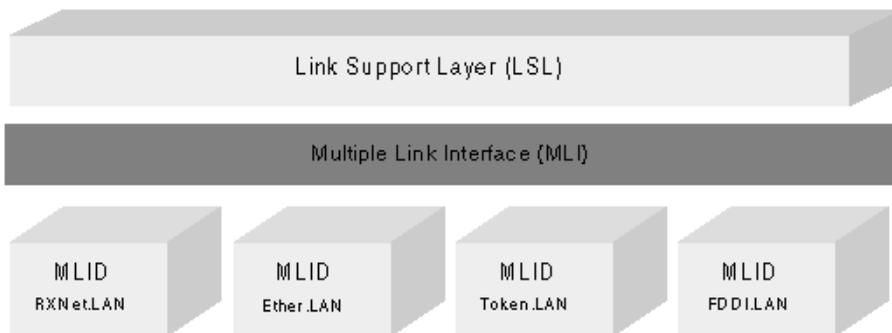
The MLID appends or strips the Frame Header from the packet. MLIDs also help determine the packet's Frame Type.

The interface used by the MLID is determined by the LAN adapter's hardware.

MLIDs can handle packets from various protocols because MLIDs do not interpret packets. MLIDs pass received packets to the Link Support Layer (LSL) based on information in Event Control Blocks (ECBs). ECBs are data structures.

The MLID communicates with the LSL through an interface called the Multiple Link Interface (MLI). The MLI (see Figure 1.4) contains the APIs necessary for the MLID to communicate with the LSL.

Figure 1.4: Multiple Link Interface (MLI)



Data Flow

As packets are sent and received, protocols add or remove their layer of information.

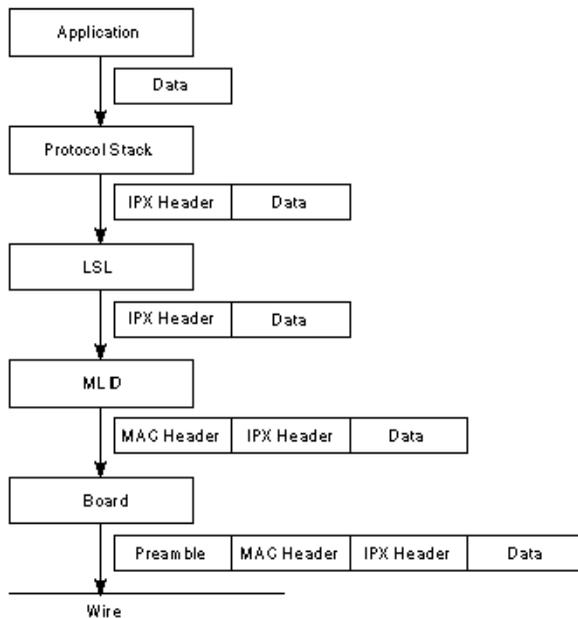
Send Data Flow

Data is sent as follows and as shown in Figure 1.5:

1. The protocol stack receives data from the application.
2. The protocol stack determines whether to split the packet into fragments.
3. The protocol stack determines the size of the fragments.

4. The protocol stack adds the protocol header to the packet.
5. The protocol stack sends the packet to the LSL.
6. The LSL routes the packet to the appropriate MLID.
7. The MLID adds the MAC header to the packet and hands the packet to the LAN adapter.
8. The hardware adds the preamble to the packet and places the packet on the wire.

Figure 1.5: Flow from Application to Wire



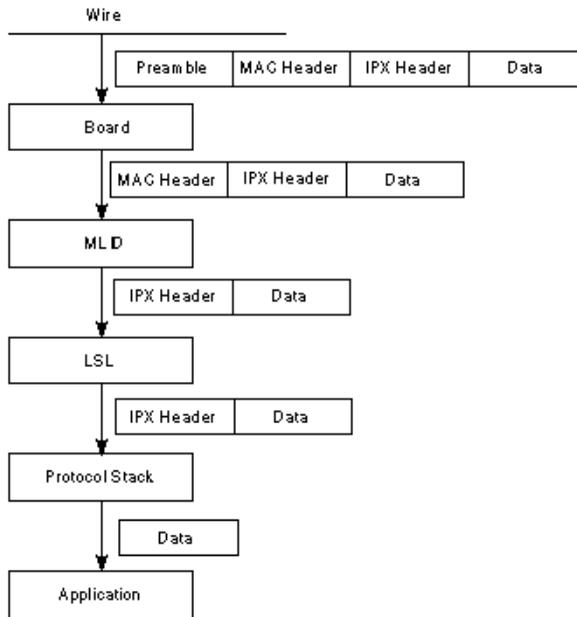
Data Flow

Data is received as follows and as shown in Figure 1.6:

1. The LAN adapter receiving the packet off the wire strips the preamble from the packet and transfers the packet to the MLID.
2. The MLID strips the MAC header from the packet and transfers the packet to the LSL.
3. The LSL routes the packet to the appropriate protocol stack.

- The protocol stack removes the protocol header from the packet and transfers the data to the application.

Figure 1.6: Flow from Wire to Application



2 ODI Module Design

Chapter Overview

This chapter briefly describes the design, programming, and functionality factors you must understand to write NetWare server MLIDs and Protocol Stacks. This chapter discusses the NetWare environment, basic NLM programming issues, and basic NetWare memory management.

You should read this chapter if you have not written a NetWare MLID or Protocol Stack before. If you have not yet developed an NLM for the NetWare 4 and later operating systems, pay particular attention to the section on Memory Management.

NetWare Environment

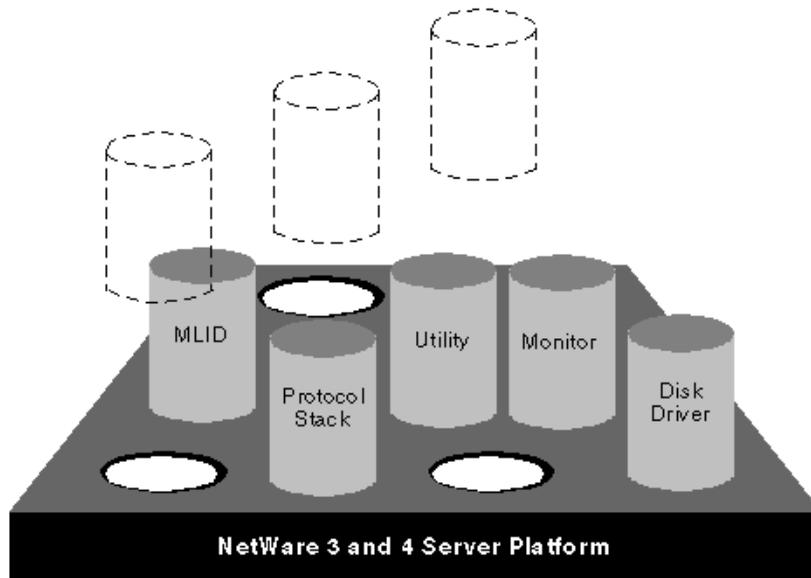
The operating system environment of the NetWare 3 and 4 servers is significantly different from DOS, OS/2, and Windows. The NetWare Loadable Module (NLM) is one of the key features that distinguishes NetWare 3 and later versions of the operating system from other operating systems.

NetWare Loadable Modules (NLMs)

NLMs are software modules that are dynamically linked to the NetWare operating system at run-time. Once an NLM is loaded, it functions as an integral component of the NetWare operating system, as shown in Figure 2.1. The network supervisor uses NLMs to load or unload additional functionality on the NetWare 3 and later operating systems without disturbing the active network. All NetWare server MLIDs and Protocol Stacks and NLMs.

Different types of loadable modules have unique filename extensions that signify the module's function. MLIDs use the .LAN filename extension, disk drivers use the .HAM file name extension, and Protocol Stacks and general utility or support modules use the .NLM filename extension.

Figure 2.1: NetWare Loadable Modules



NLM Design and Programming Issues

Because server MLIDs and Protocol Stacks operate as an integral part of the NetWare 3 and 4 operating systems, you must keep in mind the following OS characteristics:

- ♦ NetWare is multitasking.
- ♦ Some versions of NetWare use noproemptive scheduling and some use preemptive scheduling.
- ♦ NetWare can call MLIDs, Protocol Stacks, and other server applications reentrantly.
- ♦ All server MLIDs and Protocol Stacks run in 32-Bit protected mode.
- ♦ NetWare uses a flat memory model.

- ◆ NetWare supports two execution levels: process time and interrupt time.
- ◆ Portions of the NetWare operating system are written in C.

Multitasking and Preemptive Issues

All versions of the NetWare operating system are multitasking; some are uniprocessor and some are symmetric multiprocessor; some are preemptive and some are non-preemptive.

Multitasking means that the NetWare server MLIDs and Protocol Stacks run concurrently with other NLMs.

Preemptive means that the NetWare operating system will interrupt a process after it has executed for a certain length of time to allow another process to run.

Non-preemptive means that the NetWare operating system will not interrupt a process to allow another process to run.

Since some versions of the NetWare OS are nonpreemptive, every NLM must periodically return control of the CPU to the operating system. NLMs must not process for extended periods of time without relinquishing control of the CPU to the operating system so that other processes can run.

Usually MLIDs and Protocol Stacks execute for limited periods of time before returning control to the calling process. However, you should pay special attention to initialization time, error condition handling, and the loading and unloading processes so that the MLIDs and Protocol Stacks do not monopolize the processor.

Multiprocessing Issues

Beginning with NetWare 4 SMP, support for Symmetric Multiprocessing (SMP) aware Protocol Stacks and MLIDs is included in the operating system. Stacks and MLIDs wishing to include this support should pay close attention to those sections new to this specification which describe the requirements for being SMP aware. In particular, great care must be taken to make sure global resources are protected from simultaneous access by multiple processors.

Although we recommend that MLIDs and Protocol Stacks be updated to be SMP aware, non-SMP modules are fully supported.

Reentrancy Issues

We strongly recommend that MLIDs and Protocol Stacks be written to support reentrance. This allows the most efficient use of the server's resources. However, you may not always need to write a reentrant MLID. For instance, MLIDs used for testing purposes need not be reentrant. Also, if an MLID drives a LAN adapter that supports only one frame type, and if only one of these boards will ever be loaded in a server at one time, that MLID need not be reentrant.

32-Bit Protected Mode Issues

The NetWare 3 and higher operating systems run in 32-bit protected mode. When choosing the assembler to use for development, remember that it must support the use of 32-bit registers. The assembler must also be Phar Lap compatible so that the NLM LinkP module can link your modules.

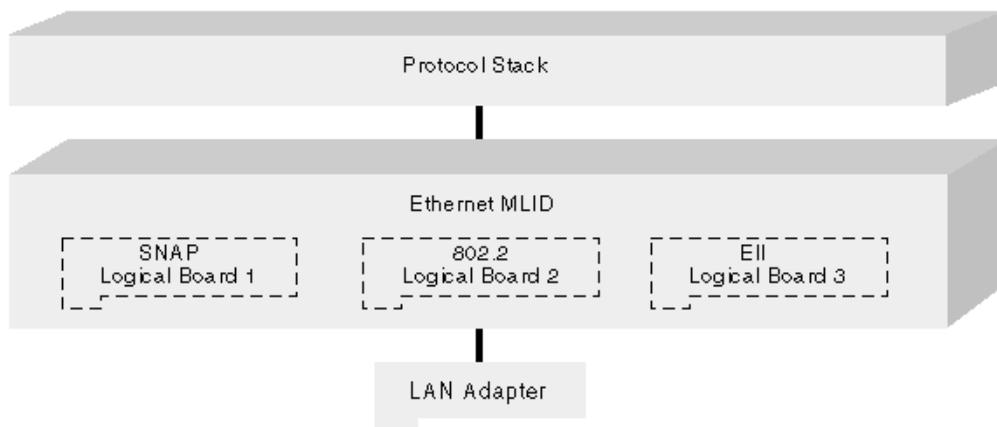
Flat Memory Model Issues

NetWare 3 and later accesses a flat code space where CS=SS=ES=DS. All offsets are 32-bit. Consequently, all of the support routines available within the operating system are near calls for the MLIDs and Protocol Stacks.

Utilization of the LAN Adapter

Each Protocol Stack minimally utilizes one LAN adapter. The Protocol Stack is independent of the LAN adapter; it sees the frame types supported by each LAN adapter as a logical board with a corresponding board number (see Figure 2.2). This board number is a handle the Protocol Stack uses when it requests the LSL to perform a function on the LAN adapter.

Figure 2.2: Correlation of Frame Types, Logical Boards, and LAN Adapter



ODI was created to allow multiple LAN adapters, and to allow multiple frame formats on each adapter. Therefore, your Protocol Stack should be able to service multiple LAN adapters as needed. As a minimum requirement, your Protocol Stack will service one LAN adapter.

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 MLID or Protocol Stack be fully unloadable.

Execution Time Issues

The two principal execution times are process time and interrupt time. As you write your MLID or Protocol Stack, you must be aware of which routines are called at process time, which are called at interrupt time, and which can be called at either time. The time or level at which the MLID or Protocol Stack is called also affects what operating system routines it can access.

The process/interrupt time considerations for each operating system routine are discussed in detail in those chapters that describe the particular routine. The following lists show which procedures can be called at which execution time.

MLID Routines Executed at Process Time

- ◆ Initialization

- ◆ Polling Procedure
- ◆ Ctl5_MLIDShutdown
- ◆ Driver Remove Procedure
- ◆ AES Call Back Routines
- ◆ Ctl14_Driver Management
- ◆ Ctl9_SetLookAheadSize
- ◆ Ctl10_MLIDPromiscuousChange
- ◆ Ctl11_RegisterReceiveMonitor

MLID Routines Executed at Interrupt Time

- ◆ Interrupt Service Routine
- ◆ Interrupt Call Back Routines

MLID Routines Executed at Process Time or Interrupt Time

- ◆ Send
- ◆ Ctl16_RemoveNetworkInterface
- ◆ Ctl17_ShutdownNetworkInterface
- ◆ Ctl18_ResetNetworkInterface
- ◆ Ctl0_GetMLIDConfiguration
- ◆ Ctl1_GetMLIDStatistics
- ◆ Ctl2_AddMulticastAddress
- ◆ Ctl3_DeleteMulticastAddress
- ◆ Ctl6_MLIDReset
- ◆ Ctl6_RemoveNetworkInterface
- ◆ Ctl7_ShutdownNetworkInterface
- ◆ Ctl8_ResetNetworkInterface

Protocol Stack Routines Executed at Process Time

- ◆ Initialization
- ◆ BindToMLID

- ◆ GetProtocolStringForBoard
- ◆ GetProtocolStackConfiguration
- ◆ GetProtocolStackStatistics
- ◆ MLIDDeRegistered
- ◆ UNbindFromMLID
- ◆ ProtocolPromiscuousChange

Protocol Stack Routines Executed at Process Time or Interrupt Time

- ◆ Send
- ◆ ReceiveHandler

NOTE: Be aware of any additional restrictions that result from calls between MLID procedures. For example, the MLID's interrupt service routine typically calls the MLID's packet transmission routine to transmit the next packet in the send queue after a transmit complete interrupt. Since the MLID's interrupt service routine executes at interrupt time, the MLID's packet transmission routine must also observe interrupt time restrictions. In most cases, the MLID may *not* be reentered. For example, if an MLID interrupt service routine calls a protocol stack transmit complete ESR, the protocol stack may *not* reenter the MLID by calling *Ctl2_AddMulticastAddress*. However, a protocol stack may call an MLID's send routine at any time.

Process Time

At process time, MLIDs and Protocol Stacks can allocate memory and (with certain exceptions) perform file input and output (I/O). All MLIDs and Protocol Stacks access two types of process time routines: routines that suspend their own execution and routines that do not. Processes that put themselves to sleep (suspend their own execution) to allow other processes on the run queue to execute are referred to as blocking processes. Processes that do not put themselves to sleep are called non-blocking processes.

The Protocol Stack should be restricted to limited processing during interrupt time. It should queue interrupt time events and handle them at process time, rather than at interrupt time.

C Language Issues

Because portions of the NetWare operating system are written in the C programming language, and MLID procedures that are called from a C routine

must preserve the EBX, EBP, ESI, and EDI registers. This is especially true for the MLID's driver initialization and driver removal procedures.

Processor Flags

Whenever the operating system calls an MLID or Protocol Stack routine, it does so with the direction flag cleared (cld is in effect). If a routine needs to set the direction flag (set), it should clear the flag before returning control to the operating system.

In addition whenever the operating system interrupt handler calls an MLID or Protocol Stack routine, the system interrupts are disabled (cli is in effect). If the routine enables interrupts operating system.

Operating System Version Issues

Many features have been added or changed during the evolution of the NetWare Operating System. It is the responsibility of the NLM developer to detect the version of the operating system and be able to use the available features correctly.

The memory management and interrupt registration features have been enhanced the most.

The *ImportPublicSymbol* function (added in OS version 3.12, see Appendix A, Operating System Support Routines) has been implemented to make managing these enhancements easier. The *ImportPublicSymbol* function allows external symbols to be imported dynamically, which makes it possible to write code that will load on any version of the OS and discover what functions are available.

Memory Protection

Protection Scheme

NetWare 4.0 has undergone extensive revision in its memory protection/management scheme. The memory protection scheme now uses the hardware paging mechanisms built into the Intel 386+ processors.

Illegal NLM Operations

The new memory protection scheme has resulted in increased protection of the operating system and of NLMs. NLMs can now be loaded at either ring 0, ring

1 or ring 3. In rings 1 and 3, NLMs can be loaded within a specific domain that is assigned at the server console. If an NLM is loaded in ring 1 and attempts an invalid memory access or an illegal operation, the system quarantines that NLM instead of abending. That NLMs code cannot be executed, and other processes cannot call any of that NLM's functions. The only allowed function for that NLM is unloading.

Memory Management

Memory Pools

Memory management was enhanced in the NetWare 4 operating system. The five memory pools that were available before NetWare 4 have been collapsed into two memory pools, accessible using the *Alloc*, *Free*, *AllocateMappedPages*, and *DeAllocateMappedPages* routines (see Appendix A, "Operating System Support Routines").

Alloc and *Free* are byte-level APIs that allocate memory from a local pool created for the NLM at load time. The other two APIs are page-level (4KB) pages that allocate from system cache. *AllocateMappedPages* allocates memory below the 16MB boundary by using flags.

Memory on Page Boundaries

This management scheme means that MLIDs (DMA adapters, for example) that require memory on physical page boundaries are guaranteed that memory allocated with *AllocateMappedPages* will be on a page boundary.

Shared RAM/ROM

However, some problems arise for MLIDs that read and write to shared RAM or ROM at initialization time before they register their memory. The NetWare 4 protection scheme recognizes these reads and writes as illegal memory accesses and quarantines the MLID as it initializes. Because it is the operating system console command process that initializes loading NLMs, the console freezes.

IMPORTANT: If your MLID accesses shared RAM or ROM before it calls *RegisterHardwareOptions*, use *ReadPhysicalMemory* and *WritePhysicalMemory* APIs.

Compatibility Settings

The NetWare 4 operating system has compatibility settings for memory reads and writes to unregistered memory. These command line set commands are called READ (or WRITE) EMULATION ON (or OFF) and READ (or WRITE) NOTIFICATION ON (or BOTH).

EMULATION set to ON means that the operating system allows access to the unregistered memory region. NOTIFICATION set to ON means that the operating system displays a warning message on the console if an unregistered memory access is attempted. EMULATION set to OFF means that the operating system quarantines any process attempting to access unregistered memory. EMULATION and NOTIFICATION default to ON.

Virtual Memory

Virtual memory was introduced into the NetWare Operating System in NetWare 5. Consequently, MLID and Protocol Stack developers must make sure that I/O memory gets locked into physical memory and is, in some cases, physically contiguous.

Hardware/Media Independence

The LSL allows Protocol Stacks to be independent of the underlying topology and frame type. In other words, Protocol Stacks can be used on any adapter with any frame type without using frame-specific code. This allows the Protocol Stacks to be used in environments that traditionally have not supported them. Specialized Protocol Stacks can be written to be frame-aware, but these are exceptions.

Development Process

The process of creating and loading a NetWare driver and Protocol Stack is briefly described below.

1. Create the source files.
2. Assemble the source files into object files.
3. Link the object files using the NetWare Linker.
4. Load the NLM as part of the NetWare operating system
5. Debug the files using the NetWare Debugger.

Chapters 3 through 8 provide detailed information on writing Protocol Stacks. Chapters 13 through 20 provide detailed information on writing MLIDs. Appendixes B and C provide a full description of assembling, linking, installing, loading, and debugging the source files.

Related Files

The following section describes the files you will need to develop MLIDs and Protocol Stacks.

Source Files

All MLIDs and Protocol Stacks are written in 386 Assembly Language.

Include Files

We have provided several include files with the support modules. These files contain external variable declarations and define the equates, macros, and data structures needed by the MLID and Protocol Stack.

Linker Definition File

The NetWare Linker requires that each NLM have a corresponding linker definition file with a .DEF extension. This file contains a list of object files that comprise the module, external variables, and routines the module must access, the names of the modules initialization and exit procedures, and several other linker directives. (See Appendix B, "Assembling and Linking NLMs" for details).

Installation Information File

You can create an optional driver information file to simplify driver installation. This file provides information related to the driver configuration and loading parameters, and is required if you use the Install utility. (See ODI Supplement: The MLID Installation Information File.)

3

Overview of Protocol Stacks

Chapter Overview

This chapter provides an overview of NetWare server protocol stack operation. It covers protocol stack and MLID multiplexing and introduces the concept of logical boards. This chapter also introduces packet transmission and reception.

The NetWare Server Protocol Stack

Protocol stacks transmit and receive data over a network. They provide the interface that allows higher layer protocols or applications access to the protocol stack's services such as routing and connection.

Protocol Stack Multiplexing

ODI protocol stacks provide maximum flexibility because they are independent of physical media and frame type. For instance, the following three scenarios are possible:

- ♦ One protocol stack can concurrently use multiple frame types (also called logical boards; see Chapter 13, "Overview of MLIDs").
- ♦ Multiple protocol stacks can be concurrently used by a frame type
- ♦ Or any combination of multiple protocols and multiple frame type is possible. (See Figures 3.1 through 3.3)

Figure 3.1: One Protocol Stack Using Multiple Frame Types

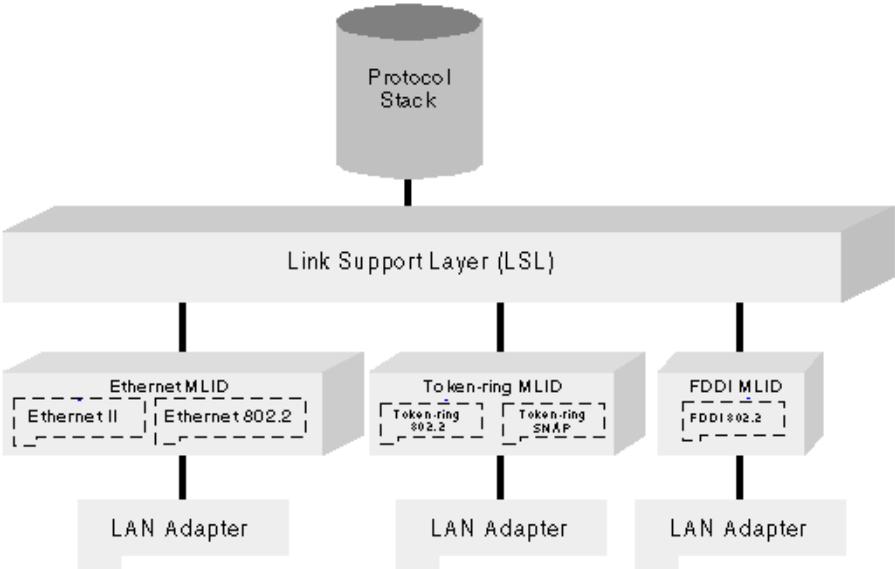


Figure 3.2: Multiple Protocol Stacks Using One Frame Type

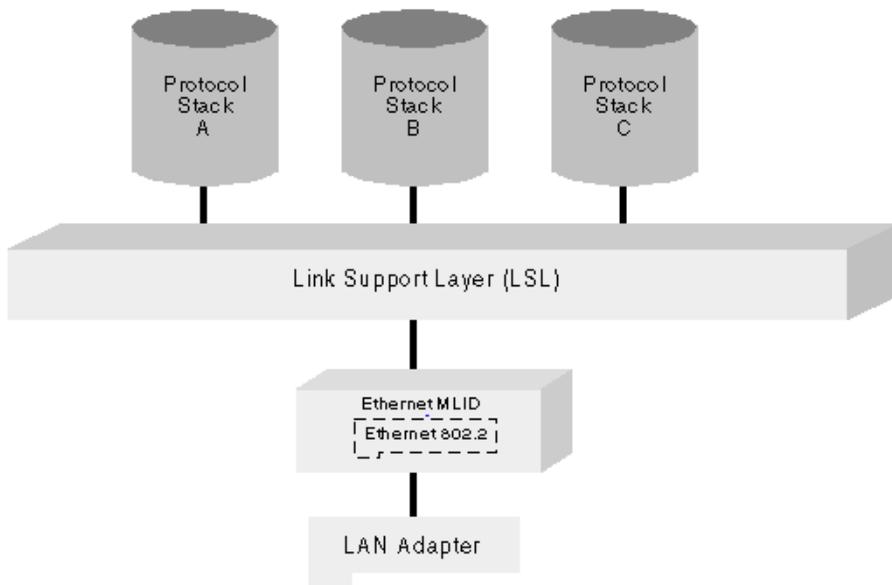
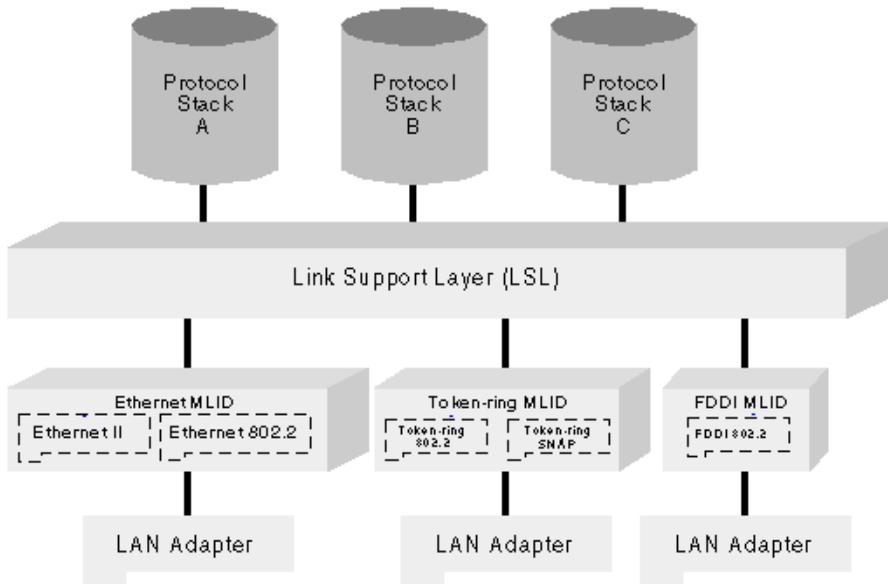


Figure 3.3: Multiple Protocol Stack Using Multiple Frame Types



Packet Flow with Multiple Protocol Stacks

Protocol stacks are media and frame type unaware. Therefore, in order for multiple protocol stacks to communicate with the logical boards, the LSL must have a unique value identifying each protocol stack and logical board (or frame type).

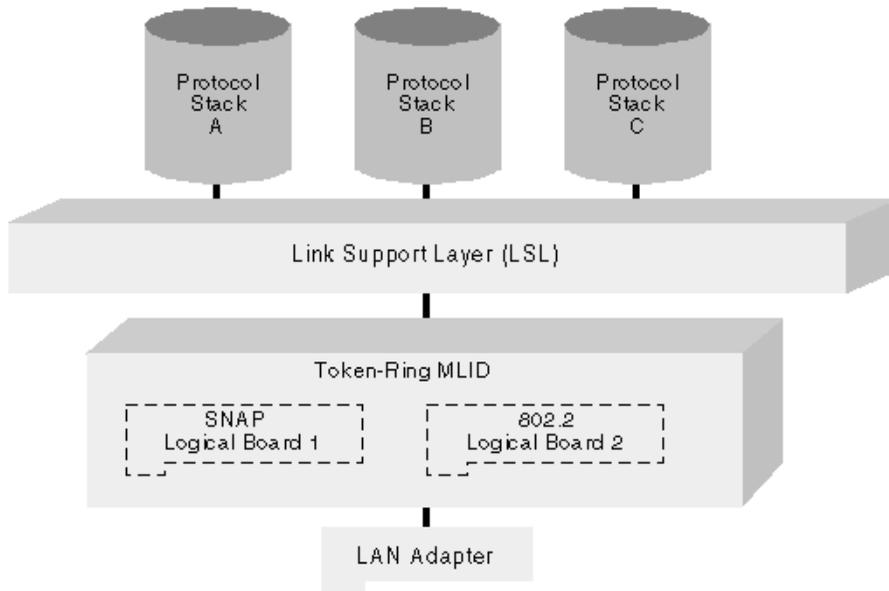
Routing a Packet to the Correct Protocol Stack

Packet reception is more involved than packet transmission and requires that the protocol stack bind to a logical board in the system. Binding enables the LSL to route incoming frames to the protocol stack.

Figure 3.4 illustrates the configuration for the following discussion.

NOTE: While tracing the packet's journey from the wire to the protocol stack, keep in mind that the NetWare server only responds to requests from either a client or another server.

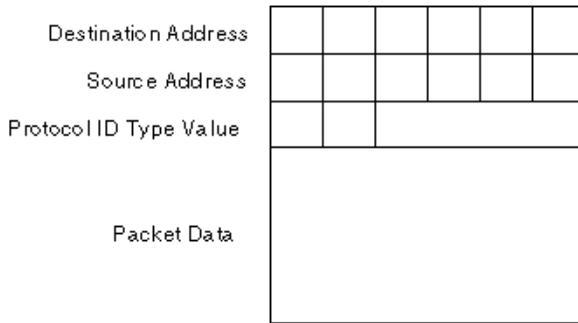
Figure 3.4: Typical Configuration of Protocol Stacks Multiplexing



During protocol stack initialization, the stack registers with the LSL. During registration, the LSL assigns a unique value known as a Stack ID to each protocol stack. When the LSL binds the protocol stack to a frame type, The LSL assigns a predefined Protocol ID (PID) to that protocol stack. The LSL stores the PID, the Stack ID, and the logical board number of the frame type in a table. The LSL uses the Stack ID, PID, and logical board number to allow communication between the protocol stacks and the logical boards.

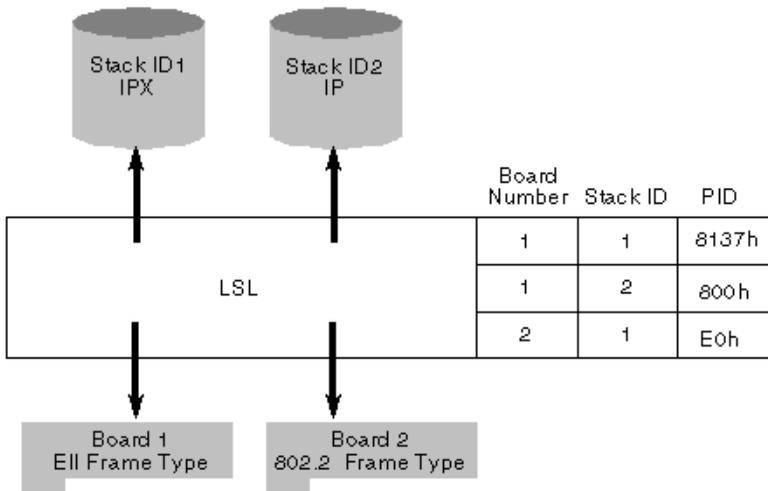
Before the requesting entity (client or server) sends a request to the server, that entity's MLID embeds the appropriate PID in the MAC header of the request packet (see Figure 3.5). The location and format of the PID in the frame header is topology and frame dependent and does not concern the protocol stack.

Figure 3.5: Example of ID Field in MAC Header of Packet



When a logical board in the server's MLID receives a request from the wire, its Interrupt Service Routine (ISR) fills in the ECB field *BoardNumber* with that logical board number. (An ECB is a buffer that contains information regarding the packet and fragment descriptors pertaining to the packet data; see Chapter 4, "Protocol Stack Data Structures".) The logical board in the MLID takes the PID from the MAC header and places it in the ECB field *ProtocolID*. The board number and the PID to index the table and determine the Stack ID of the stack that is to receive the packet. (See Figure 3.6.)

Figure 3.6: MLID/Protocol Stack Multiplexing



Routing a Packet to the Correct Logical Board

When the server transmits a response, the server's LSL is able to check the ECB Board Number field (filled out in the process above) to determine which logical board in the MLID prepares the packet for transmission.

The information in the ECB *boardNumber* field is preserved, because when an application responds, it usually returns either the same ECB or an ECB with a copy of the pertinent fields from the original.

Packet Reception with Multiple Protocol Stacks

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

- ♦ The *boardNumber* specifies the logical board and the frame type.
- ♦ The *ProtocolID* (PID), together with the *boardNumber*, specifies which protocol stack the packet will be sent to.

When a protocol stack registers with the LSL, the stack gives the LSL the address of the stack's receive handler routine. This routine can be called at interrupt time and is entered with interrupts disable. No registers or flags need to be preserved.

Packet Reception Methods

The ODI specification defines three protocol stack reception methods:

- ◆ Bound
- ◆ Prescan
- ◆ Default

Prescan and default protocol stacks can be chained (see Chapter 5, "Protocol Stack Initialization").

Bound Protocol Stacks

Bound protocol stacks are the most common method. A bound protocol stack requires that frames received from the LSL have a registered Protocol ID (PID) in the ECB *ProtocolID* field. The system administrator registers Protocol IDs at the command line for each protocol stack that will be used. The appropriate PIDs for a given protocol stack are usually different for each frame type. ODI Specification Supplement: *Frame Types and Protocol IDs* lists the common protocol stack PID values for most frame types.

The LSL uses the PID in the ECB Protocol ID field 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.

A registered protocol stack only receives packets with one PID per logical board. Protocol stacks containing a limited number of Network Layer protocols that use different PIDs (for example, TCP/IP, ARP, RARP) must be registered to the LSL as separate and distinct protocols. These protocol stacks 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 NLM and can specify the same receive handler routine. The receive handler routine then examines the ECB stackID field to determine which subprotocol the frame is intended for.

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 packet's protocol type 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 packet, allows other protocols to process the packet, or discards the packet. Only diagnostic utilities, or compression protocol stacks should be used as prescan stacks.

LSLENH.NLM must be loaded before a prescan protocol stack written to this specification can be installed on NetWare 3.12.

Default Protocol Stacks

Default protocol stacks receive every frame not claimed by any other protocol stack (prescan or bound).

Protocol stacks that provide an alternate Data-Link Layer solution should be default protocol stacks.

LSLENH.NLM must be loaded before a default protocol stack written to this specification can be installed on NetWare 3.12.

Packet Reception Process

To receive packets from an MLID, a protocol stack must register with the LSL and then bind to that 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.

Packet Reception Process

1. The MLID places the MLID's board number, the Protocol ID embedded in the packet's MAC layer header, and the packet's data into a receive ECB.
2. The MLID passes the ECB to the Link Support Layer.
3. The LSL uses the board number and Protocol ID to route the packet to the protocol stack.
4. The LSL calls the prescan stack chain registered for the MLID.
5. The LSL in the absence of a prescan stack chain, searches for any stack that is bound to receive packets from the MLID.
6. The LSL in the absence of a bound stack, calls the default protocol stack chain registered for the MLID.
7. The LSL in the absence of a default stack chain, ignores the packet.
8. The LSL returns the ECB to the LSL's ECB pool.

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

4 Protocol Stack Data Structures

Chapter Overview

This chapter describes the protocol stack configuration table and the protocol stack statistics table. It also describes the Event Control Block. The Event Control Block is also described in Chapter 14, "MLID Data Structures", but is included here as reference material for protocol stack developers.

Protocol Stack Configuration Table

Sample 4-1 shows the sample code of the protocol stack configuration table. [Figure 1](#) shows a graphic representation of the protocol stack configuration table. [Table 3](#) describes the fields of the protocol stack configuration table.

Sample 4-1 Protocol Stack Configuration Table Sample Source Code

```
CFGMajorVersion    db    ?
CFGMinorVersion    db    ?
Name               db    ?
ShortName          dd    ?
Stack_MajorVersion db    ?
Stack_MinorVersion db    ?
Reserved           db    16 Dup (?)
```

Figure 1 Graphic Representation of the Protocol Stack Configuration Table

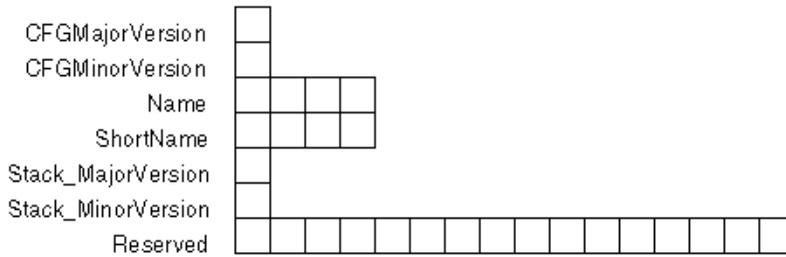


Table 3 Protocol Stack Configuration Table Fields

Offset	Label	Size(bytes)	Description
00h	CFG_MajorVersion	1	Major version number of the configuration table (1 for this specification).
01h	CFG_MinorVersion	1	Minor version number of (0 through 99 decimal) of the configuration table (00 for this specification).
02h	Name	4	Address of a length-preceded, zero-terminated string with the name of the protocol stack. This name may be longer than the name used to register the stack with the Link Support Layer. The name cannot be longer than 127 bytes.
06h	ShortName	4	Address of the protocol name (length-preceded, zero-terminated, 15 characters maximum) used to register the stack with the Link Support Layer. This is also the name by which the user refers to the protocol stack.
0Ah	Stack_MajorVersion	1	Major version number of the protocol stack.
0Bh	Stack_MinorVersion	1	Minor version number (0 through 99 decimal) of the protocol stack.
0Ch	Reserved	16	Must be set to 0.

Protocol Stack Statistics Table

The examples below include sample code and a graphic representation of the protocol stack statistics table. [Table 4](#) describes the statistics table fields.

Figure 2 Protocol Stack Statistics Table Sample Source Code

Stat_MajorVersion	db	?
Stat_MinorVersion	db	?
GenericCnts	dw	?
ValidCntsMask	dd	?
TotalTXPackets	dd	?
TotalRXPackets	dd	?
IgnoredRXPackets	dd	?
NumCustom	dw	?
Custom1	dd	?
Custom2	dd	?
CustomN	dd	?
CustomStringPtr1	dd	?
CustomStringPtr2	dd	?
CustomStringPtrN	dd	?

Figure 3 Graphic Representation of the Protocol Stack Statistics Table

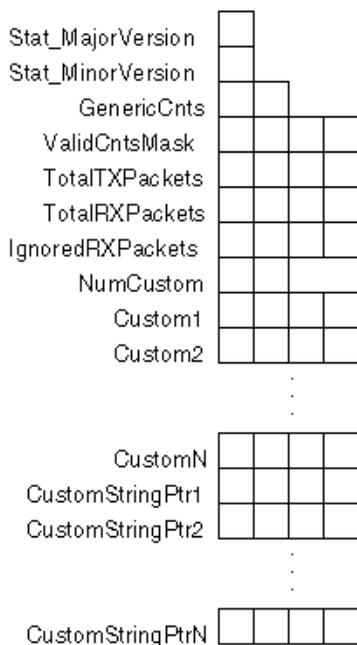


Table 4 Protocol Stack Statistics Table

Offset	Label	Size(byte s)	Description
00h	STAT_MajorVersion	1	Major version number of the statistics table (1 for this specification).
01h	STAT_MinorVersion	1	Minor version number of the statistics table (0 through 99 decimal, 0 for this specification).
02h	GenericCnts	2	Number of 4-byte counters in fixed portion of table. This should be set to 3.
08h	ValidCntsMask	4	Bit mask indicating which counters are valid. 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.

Offset	Label	Size(byte s)	Description
0Ah	TotalTXPackets	4	Total number of packets that were requested to be transmitted (whether or not they were actually transmitted).
0Ch	TotalRXPackets	4	Total number of incoming packets the protocol stack received.
10h	IgnoredRXPackets	4	Total number of incoming packets the protocol stack ignored.
14h	NumCustom	2	This field contains the number of custom counters defined by the protocol stack. Each custom counter must have an associated string that can be accessed through the <i>CustomStrings</i> area (defined below).
18h	Custom1	4	These fields contain custom counters that the protocol stack can configure for its specific needs.
1Ch	Custom2	4	These fields contain custom counters that the protocol stack can configure for its specific needs.
??h	CustomN	4	This field is the last of the custom counter fields.
??h+4	CustomStringPtr1	4	This field contains a pointer to the <i>CustomString</i> corresponding to the first custom counter (Custom1). Each string in this area must be null-terminated. The string order must correspond with the custom counters.
??h+4	CustomStringPtr2	4	This field contains a pointer to the <i>CustomString</i> corresponding to the second custom counter (Custom2). Each string in this area must be null-terminated, and the table of string is terminated by two nulls. The string order must correspond with the custom counters.
??h+4	CustomStringPtr3	4	This field contains a pointer to the <i>CustomString</i> corresponding to the last custom counter (<i>CustomN</i>). Each string in this area must be null-terminated, and the table of strings is terminated by two nulls. The string order must correspond with the custom counter.

Event Control Blocks

NetWare shells and operating systems use structures called Event Control Blocks (ECBs) to receive, send, and manage packets. In the NetWare 3+ operating systems, MLIDs, the LSL, and protocol stacks all use ECBs.

Because MLIDs and protocol stacks both use ECBs, this section is duplicated in the MLID portion of the document.

When receiving a packet, the MLID obtains an ECB, fills it out, and copies the packet into a buffer that is immediately below the ECB. Remember that the buffers associated with receive ECBs are contiguous. After copying the packet from the board, the MLID passes the ECB to the LSL. The LSL then examines the ECB and hands it to the correct protocol stack.

When sending a packet, a protocol stack puts a list of fragment pointers (that describe the packet) in the ECB and passes the ECB to the LSL. The LSL refers to the ECB to determine the destination MLID, and then passes the ECB to the MLID. The MLID collects all packet fragments and sends the packet.

Receive Event Control Block

The following source code defines the receive ECB structure. The asterisks (*) indicate the fields that the MLID fills in before passing the ECB to the LSL. [Figure 5](#) provides a graphic representation of the receive ECB.

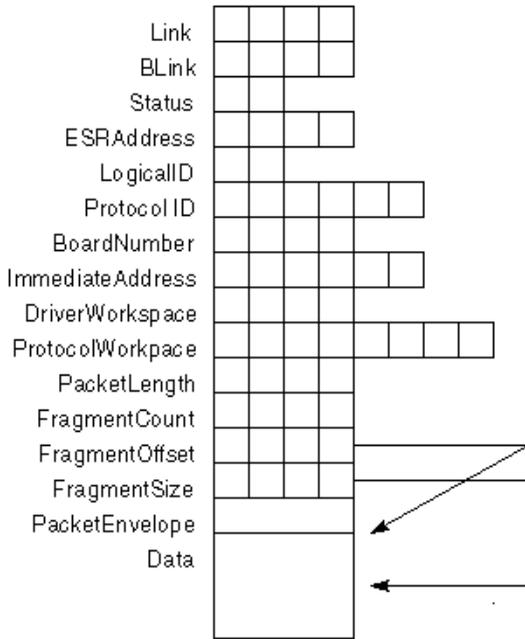
Figure 4 Receive Event Control Block Sample Source Code

```

        Link      dd      0
        *BLink    dd      0
        Status    dw      0
        ESRAddress dd      0
        LogicalID  dw      0
        *ProtocolID db     6 dup (?)
        *BoardNumber dd     0
        *ImmediateAddress db 6 dup (?)
        *DriverWorkspace dd     0
        ProtocolWorkspace db 8 dup (?)
        *PacketLength dd     0
        *FragmentCount dd     1      ; always 1 on receives
        *FragmentOffset dd     0
        *FragmentSize dd     0
        *PacketEnvelope dd     0      ; variable length field that
                                       ; contains media headers
        *Data      db     0 dup (?)   ; variable length field that
                                       ; contains protocol headers
                                       ; and packet information

        PacketEnvelope equ    byte ptr FragmentSize + 4
```

Figure 5 Graphic Representation of the Receive Event Control Block



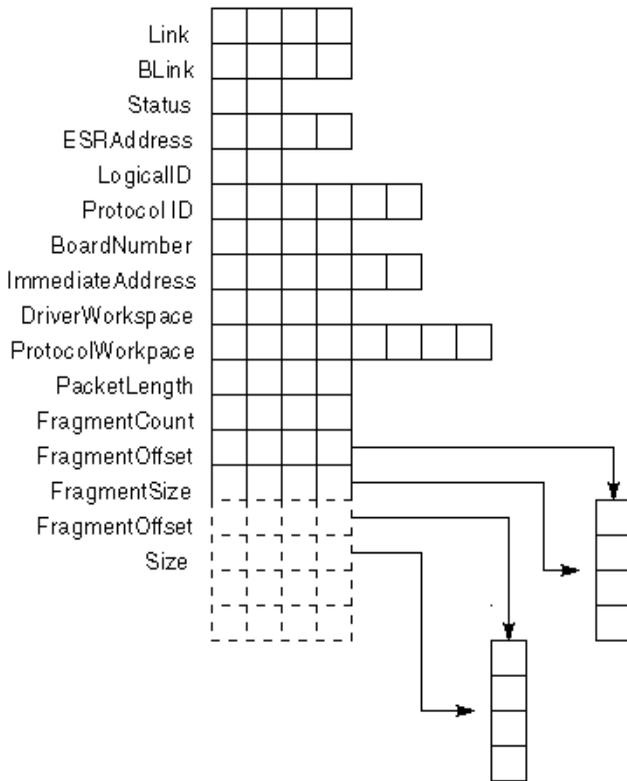
Transmit Event Control Block

The following source code is the definition of the transmit ECB. The asterisks (*) indicate the fields that must be filled in by higher layers of the operating system before the ECB is passed to the MLID. **Figure 7** provides a graphic representation of the transmit ECB. A description of each ECB field follows the figures.

Figure 6 Transmit Event Control Block Sample Source Code

```
    Link      dd      0
    BLink     dd      0
    Status    dw      0
    *ESRAddress dd     0
    *LogicalID dw     0
    *ProtocolID dd    6 dup ?
    *BoardNumber dd    0
    *ImmediateAddress db  6 dup ?
    DriverWorkspace dd    0
    ProtocolWorkpace db  8 dup ?
    *PacketLength dd    0
    *FragmentCount dd    0 ; at least 1
    *FragmentOffset dd    0 ; repeated RFragmentCount times
    *FragmentSize dd    0
```

Figure 7 Graphic Representation of the Transmit Event Control Block



Event Control Block Field Descriptions

The following table describes the Event Control Block fields:

Table 5 Event Control Block Field Descriptions

Offset	Name	Size(bytes)	Descriptions
00h	Link	4	Forward link to another ECB. The LSL uses this field to queue ECBs. Protocol stacks and MLIDs can also use this field when they possess the ECB.

Offset	Name	Size(bytes)	Descriptions
04h	BLink	4	This field is typically used as a back link for managing a list of ECBs. The current owner of the ECB uses this field. When an ECB is returned from an MLID containing a received packet, this field contains the received packet error status. See the "Setting the ECB BLink Field" section in Chapter 6, "Protocol Stack Packet Reception".
08h	Status	2	MLIDs must not use or modify this field. The LSL uses this field to indicate the current state of the ECB (for example, the ECB is currently unused, or queued for sending, etc.)
0Ah	ESRAddress	4	When an ECB originates from the LSL, the LSL sets this field, and MLIDs and protocol stacks must <i>not</i> use it. When a transmit ECB originates from the protocol stack, the protocol stack sets this field to point to a routine to be called when the transmission is complete and the ECB is available again.
0Eh	LogicalID	2	MLIDs use this field, but must not change it. When a protocol stack registers with the LSL, the LSL assigns the stack a logical number (0 through 15). This field contains the logical number. If the packet is a priority send, this field contains a value between 0FFF7h (lowest priority) and 0FFF0h (highest priority). If the packet is a raw send, this field contains a 0FFFFh. If the packet is a priority raw send, this field contains a value between 0FFFFh (lowest priority) and 0FFF8h (highest priority). On normal sends, the protocol stack places its own logical number in this field. On receives, the LSL places the target stack's logical number in this field.
10h	ProtocolID	6	This field contains the Protocol ID (PID) value on both sends and receives. This value is stored in high-low order. For a full explanation of how to fill out this field, refer to ODI Supplement: Frame Types and Protocol IDs.
16h	BoardNumber	4	When an MLID registers with the LSL for a particular LAN adapter, the LSL assigns that logical board a number. (Logical board 0 is used internally in the operating system.) Consequently, MLIDs are assigned logical board numbers 1 through 255). On sends, protocol stacks fill in this field to indicate the target logical board. On receives, the MLID fills in this field to indicate which logical board received the packet.

Offset	Name	Size(bytes)	Descriptions
1Ah	ImmediateAddress s	6	<p>On receives, the immediate address represents either 1) the packet's source node address or 2) the routing LAN adapter's node address if the packet was routed from another network. During a receive, the MLID fills in this field. This value is stored in high-low order. On RX-Net, or whenever the node address is less than six bytes, put the node address in the least significant byte and pad the remaining bytes with 0. On sends, the immediate address represents either the destination node address or the destination router address; the protocol stack fills in this field. Addresses passed to the upper layers are in either canonical or noncanonical format, depending upon whether the MLID bit-swaps MSB format addresses. The protocol stack fills in this field on sends. All addresses passed down to the MLID are in canonical format if the MLID is configured to be in LSB.</p> <p>In general, protocol stacks do not need to be aware of this field's format. Protocol stacks can just copy the contents of the receive ECBs <i>ImmediateAddress</i> field into the transmit ECBs <i>ImmediateAddress</i> field before sending the packet. Protocol stacks may get the immediate address from somewhere else, but it must still be copied into the transmit ECBs <i>ImmediateAddress</i> field.</p>

Offset	Name	Size(bytes)	Descriptions
20h	DriverWorkSpace	4	<p>An MLID can use this field for any purpose. However, be aware that the LSL uses the bytes at offsets 22h and 23h as temporary storage during the receive prescan stack filtering. Before passing a completed receive ECB to the LSL, the MLID will fill in the byte at offset 20h with the destination address type of the received packet:</p> <ul style="list-style-type: none"> ◆ 00h = Direct ◆ 01h = Multicast ◆ 03h = Broadcast ◆ 04h = Remote Unicast ◆ 08h = Remote Multicast ◆ 10h = No Source Route ◆ 20h = Error Packet ◆ 80h = Direct Unicast <p>Set the second byte of the field (offset 21h) to indicate whether the MAC header contains one or two 802.2 control bytes:</p> <ul style="list-style-type: none"> ◆ 0 = All frame types other than 802.2 ◆ 1 = 802.2 header has only Ctrl0 byte (Type I) ◆ 2 = 802.2 header has Ctrl0 and Ctrl1 (Type II) ◆ For an explanation of 802.2 Type I and Type II, refer to ODI Supplement: Frame Types and Protocol Ids.
24h	ProtocolWorkspace	8	Reserved for the protocol's workspace. The MLID must not modify this field.
2Ch	PacketLength	4	This field contains the total length of the packet in bytes. This is the length of the data portion of the packet (not including media headers or SAP headers contained in the PacketEnvelope portion).
30h	FragmentCount	4	This field indicates the total number of packet fragment descriptors that follow. Each descriptor consists of a pointer to a fragment buffer and the size of that buffer. On receives, this value is always 1 or greater. On sends, the fragment count can be 0. The ECB and all fragments must be guaranteed convertible to a valid physical address if the MLID uses <i>GetServerPhysicalOffset</i> . If the MLID is a type 4 NLM, it should not allow the OSData segment to contain the buffers.

Offset	Name	Size(bytes)	Descriptions
34h	FragmentOffset	4	On receives, a buffer immediately follows the ECB in memory. The MLID copies the received packet into this buffer. After the MLID copies the packet into this buffer, it must set the <i>FragmentOffset</i> to point around any media headers to the data portion of the packet. The MLID must also set the <i>FragmentLength</i> field to the total length of the data portion of the packet. On sends, the <i>FragmentOffset</i> field points to the first fragment buffer containing packet data. The <i>FragmentSize</i> field specifies the length of that buffer. Additional fragment descriptors can immediately follow the ECB in memory. The MLID collects the data from these fragment buffers to form the packet for transmission (see Figure 7).
38h	FragmentSize	4	This field indicates the length in bytes of the first packet fragment. On receives, the value in this field is the same as the value in the <i>PacketLength</i> field. On sends, this value can be 0. On receives only, the memory immediately following the ECB also contains the following two fields:
3Ch	PacketEnvelope	varies	Everything below this field is included in the packet envelope. The first piece of data in the packet envelope is the media header of the packet. This field varies in length according to topology and frame-type, and appears only in receive ECBs. This field is not used or present if the LAN topology splits the data of the packet and transmits it in more than one frame (for example, RX-Net).
??h	Data	varies	The data portion of the packet that immediately follows the <i>MediaHeader</i> . On sends only, the memory immediately following the ECB also contains the following two fields.
3Ch	FragmentOffset2	4	This field contains additional fragment descriptors when the <i>FragmentCount</i> is greater than 1.
40h	FragmentLength2	4	This field contains additional fragment descriptors when the <i>FragmentCount</i> is greater than 1.

5

Protocol Stack Initialization

Chapter Overview

This chapter discusses registering and binding prescan protocol stacks, bound protocol stacks, and default protocol stacks. This chapter also discusses information on chaining protocol stacks.

Protocol Stack Initialization

When initializing a protocol stack, you must keep the following items in mind:

- ◆ The protocol stack is not fully operational until it is bound to a board. The protocol stack must not send or receive frames until *Ctl2_Bind* has been called and the protocol stack is bound to a board, or the protocol stack has bound itself to a board.
- ◆ If the protocol stack does not auto-bind when it is installed, it should do limited initialization and return control of the CPU to the operating system.
- ◆ When the protocol stack is initialized, it gets a protocol stack ID from the LSL by calling *LSLRegisterStackRTag*. This informs the LSL about the protocol stack.
- ◆ If the protocol stack is SMP aware, it must call *LSLRegisterStackSMPSafe*.
- ◆ If the protocol stack auto-binds and then receives a *Ctl2_Bind* request, it can unbind from the board and re-bind to a board according to the user's registration information.

LAN Boards and Auto-binding

In some cases, a protocol stack may know what frame type or what media type to bind to, or it may bind by default to a particular frame type. Caution should be used when writing a protocol stack to do auto-binding, since many different high level protocols can be transmitted over a wide variety of different MAC layer frame types. Developers must not assume that all users of all sites will use the same frame type for each protocol. Auto-binding must never prevent a system administrator from binding a protocol stack to the frame type he wishes to use.

To identify the logical LAN boards present on a system, the protocol stack should first call *GetNumberOfLANs*, which returns the maximum number of logical boards that can be present on the system. Then, the protocol stack should call *LSLGetMLIDControlEntry* for each logical board number from 1 to the number returned by *GetNumberOfLANs*. Logical board numbers are not necessarily contiguous.

The valid logical board numbers are indicated by the completion code and the control entry point returned by *LSLGetMLIDControlEntry*. Information of the frame types supported by each logical board is returned by the *Ctl0_GetMLIDConfiguration* entry point (see Chapter 13, "Overview of MLIDs" and Chapter 20, "MLID Control Routines" for information about this function).

Binding

Bound Protocol Stacks

All bound protocol stacks call *LSLGetPIDFromStackIDBoard* to get a defined Protocol ID (PID) from the LSL. Protocol stacks must get a PID before they can transmit and receive on a LAN adapter. If the LSL returns a PID, the protocol stack should save this value internally. The protocol stack blindly places this value in the ECB's *ProtocolID* field when it transmits a packet.

NOTE: In rare cases, if the LSL does not return a PID, the protocol stack can add a PID by calling *LSLAddProtocolID*. An intelligent protocol stack that finds it has no PID registered, can register a PID for itself based on well-known frame type and stack name combinations. For example, IP on Ethernet_II or Ethernet_SNAP always uses the PID 800h.

To conform to this specification, you must write protocol stacks that are independent of any specific frame type or topology. Protocol stacks must not

interpret Protocol IDs, because Protocol IDs depend on the frame type and the topology on which the protocol stack is used.

The network administrator usually enters the Protocol ID of the Protocol stack for each frame-type/board combination into the AUTOEXEC.NCF file. The network administrator then binds the MLID, by frame type (logical board), to the protocol stack as follows:

```
Bind<Protocol>to<MLID Name>Frame=<XX>
```

The protocol stack then calls *LSLGetPIDFromStackIDBoard* to get the Protocol ID.

The operating system reads the AUTOEXEC.NCF file and passes the Protocol ID, of the frame it is to bind to, to the protocol stack. If the protocol stack does not find a Protocol ID, the protocol stack loads, but is not functional. In which case, after the protocol stack has loaded, the network administrator can bind the protocol stack to a frame type by specifying a Protocol ID from the command line using the following command line syntax:

```
Protocol Register <Protocol Name><Frame Name><Protocol ID>
```

For example: Protocol Register IP Ethernet_II 800

where:

```
<Protocol Name>=IP
```

```
<Frame Name>=Ethernet_II
```

```
<Protocol ID>=800
```

NOTE: IPX protocol stacks are the only exception to the above rule. Because the IPX PIDs are static for each frame type, the user does not ever need to change the frame type's PID at the command line. Most MLIDs written for NetWare 3+ automatically register the IPX protocol stack using *LSLAddProtocolID*.

The network administrator can register media-aware protocol stacks from the command line. Media-aware protocol stacks are stacks that only communicate with one or two frame types.

Typing Protocol at the command line produces the protocols, frame types, and Protocol IDs that have been registered.

Determining the Maximum Packet Size

After the protocol stack binds as either a prescan, default, or bound stack, it determines the maximum packet size it can send and receive on the LAN

adapter by examining the MLID's configuration table *MLIDRecvSize* field. The protocol stack gets a pointer to the configuration table by calling *LSLGetMLIDControlEntry* and invoking the MLID control handler with the proper function code (see Chapter 20, "MLID Control Routines").

The value in the *MLIDRecvSize* configuration table field represents the largest amount of data the protocol stack can consistently send and receive using that LAN adapter (see Chapter 14, "MLID Data Structures" for the MLID configuration table). The protocol stack must subtract the size of any protocol headers (for example, the IPX header) from this value. The difference between *MLIDRecvSize* and the protocol headers represents the maximum, amount of real data that the LAN adapter can send and receive.

Initialization

During main initialization, the protocol stack must register with the LSL. The function the protocol stack uses to register depends on the method it uses to transmit and receive packets (bound, prescan, or default). See [Table 6](#) below.

Table 6 Protocol Stack Registration Functions

Reception Method	Registration Functions
Bound	LSLRegisterStackRTag
Prescan	LSLRegisterPrescanRxChain
	LSLRegisterPrescanTxChain
Default	LSLRegisterDefaultChain
Bound/Prescan/Default	LSLRegisterStackSMPSafe

Bound Protocol Stack Initialization

Using the above calls, protocol stacks register with the LSL by exchanging information. Bound protocol stacks register by following the steps below:

Table 7 Bound Protocol Stack Initialization

Module	Action
Protocol Stack (Initialization Routine)	<ol style="list-style-type: none"> 1. Calls LSLRegisterStackRTag to pass the following items to the LSL: <ul style="list-style-type: none"> ◆ A resource tag for the protocol stack ◆ A resource tag for the protocol stack's receive ECBs ◆ A pointer to the protocol stack's receive entry point ◆ A pointer to the protocol stack's control entry point ◆ The protocol stack's name
LSL	<ol style="list-style-type: none"> 2. Stores the pointers listed above and returns the Stack ID (through LSLRegisterStackRTag).
Protocol Stack(Bind IOCTL)	<ol style="list-style-type: none"> 3. Stores the Stack ID. 4. If the stack is SMP aware, calling LSLRegisterStackSMPSafe will pass in the stack ID obtained in step 2. 5. Calls LSLBindStack to bind with a specific board (either a physical or a logical board) after it gets the BindToMLID IOCTL. 6. Calls LSLGetPIDFromStackIDBoard to get the Protocol ID to use for a specific board after being bound with the Bind IOCTL.

Prescan and Default Protocol Stack Registration

The LSL keeps a list of chained stack structures on a per-logical-board basis. The LSL chain registration routines allocate and fill in each chain structure. These routines pass back a Chain ID, which is a pointer to this structure if registration is successful. The prescan and default protocol stack follow the steps below to register:

Table 8 Prescan and Default Protocol Stack Registration

Module	Action
Protocol Stack (Initialization Routine)	<ol style="list-style-type: none"> 1. Calls the appropriate routine to bind to the specified board during the protocol stack's initialization. (The LSL routines that register the default and prescan protocol stacks are described below.)

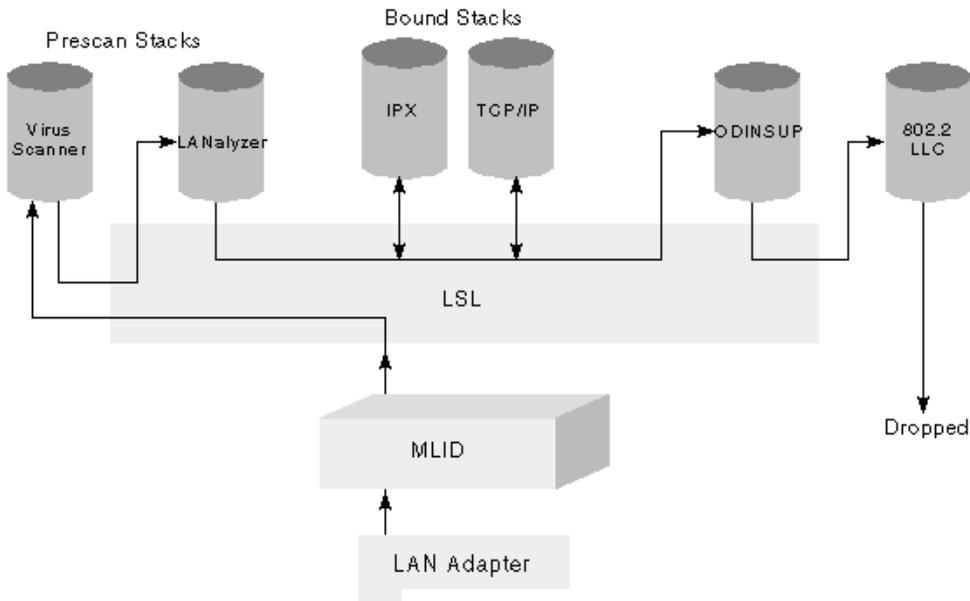
Module	Action
LSL	<ol style="list-style-type: none"> Links the stack into the proper position of the logical board's chain list. Stores the pointers provided by <i>LSLRegisterPrescanRxChain</i>, <i>LSLRegisterPrescanTxChain</i>, <i>LSLRegisterDefaultChain</i>. Returns a Chain ID to the protocol stack if the registration is successful.

Chaining Prescan and Default Protocol Stacks

Prescan and default protocol stacks can be chained so that the received and transmitted packets flow through the chained stacks in a prescribed order. Figure 5.2 illustrates sample receive packet flow through a system with chained prescan and default stacks.

The LSL adds the chained stacks in the chain position order that the stacks request. If a stack must be first or must be last in the chain, and another stack that also must be first or must be last already occupies that position, the attempt to load the second stack returns an error message.

Figure 5.1: Prescan and Default Protocol Stack Chaining



Registering and Deregistering Prescan and Default Protocol Stacks

A protocol stack can specify a position in the chain when it registers. The registration routines and their purposes are listed in the following table.

Table 9 Registration Routines

Stack Type	Registration Routine	Purpose
Prescan	LSLRegisterPrescanRxChain	Allows the stack to monitor received packets before the bound protocol stack receives them.
	CLSLRegisterPrescanRxChain	
Prescan	LSLRegisterPrescanTxChain	Allows the stack to monitor transmit packets and to request a position in the chain.
	CLSLRegisterPrescanTxChain	
Default	LSLRegisterDefaultChain	Allows the stack to receive packets that no prescan or bound protocol stack wanted, and to request a position in the chain.
	CLSLRegisterDefaultChain	

NOTE: We have replaced several deregistration and registration routines with new routines. Old default protocol stacks are placed at the last position in the default protocol stack chain. These stacks must clear EAX before they return for their receive routine if they have consumed the ECB. Old prescan protocol stacks are linked into the next available position on the prescan receive chain. Although we still support the older registration and deregistration routines, we consider them as obsolete. These routines will not be supported in the future. We recommend that you replace the old routines in your protocol stack with the new routines. Table 11.2: "Correlation of Old and New Registration and Deregistration Functions" illustrates the correspondence between the old and new registration calls.

IMPORTANT: The LSL deregister and register protocol stack calls must be used in the following pairs:

- ◆ LSLDeRegisterPreScanRxChain
- ◆ LSLRegisterPreScanRxChain
- ◆
- ◆ LSLDeRegisterPreScanTxChain
- ◆ LSLRegisterPreScanTxChain
- ◆

- ◆ LSLDeRegisterDefaultChain
- ◆ LSLRegisterDefaultChain
- ◆
- ◆ LSLDeRegisterStack
- ◆ LSLRegisterStackRTag
- ◆
- ◆ LSLDeRegisterStackSMPSafe
- ◆ LSLRegisterStackSMPSafe

Default and Prescan Protocol Stack Chaining

The LSL fills in the ChainStructure with the information in the registration routine's parameters. The ChainStructure is defined in *11.31 - LSLGetStartChain*. To request a chain position in this structure, the assembly routines use the value in EBP, and the C Language routines use the value in the parameter StackChainPositionRequested. The following table contains the range of values and their meanings.

Table 10 Chain Position Values

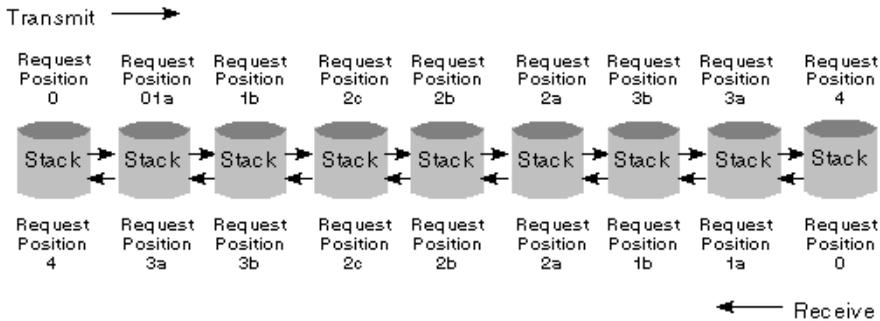
Value	Name	Position in Chain
0	STACK_REQ_FIRST	The stack must start the chain.
1	STACK_REQ_NEXT_FIRST	The stack must be loaded at the next available position from the front of the chain.
2	STACK_REQ_DEPEND	The stack's position in the chain is dependent on the order in which the stacks are loaded. The stack will take the next available position when it loads.
3	STACK_REQ_NEXT_LAST	The stack must be loaded at the next available position from the end of the chain.
4	STACK_REQ_LAST	The stack must end the chain.

Chain Position

A chained protocol stack that both sends and receives must register as two separate stacks: it must register as the appropriate type of transmit protocol stack and request the correct chain position; it must also register as the

appropriate type of receive protocol stack with the correct chain position. Keep in mind that the receiving station must undo algorithms applied to packets in the opposite order in which they were applied. For example, if a packet goes through LANalyzer protocol stack A and then through compression protocol stack B, the receiving station must send the packet through decompression protocol stack B and then through LANalyzer protocol stack A. The following figures (5.2, 5.3, 5.4) illustrate the stack chaining concept.

Figure 5.2: Prescan Protocol Stack Chain



A text picture of the chain appears as follows:

```

LSL Tx  → 0 → 1a → 1b → 2a → 2b → 2c → 3b → 3a → 4 MLID
LSL Rx  → 4 → 3a → 3b → 2c → 2b → 2a → 1b → 1a → 0 MLID
a, b, c, etc. represent the order the stack was loaded into that position.

```

Figure 5.3: Transmit Prescan Protocol Stack Chaining

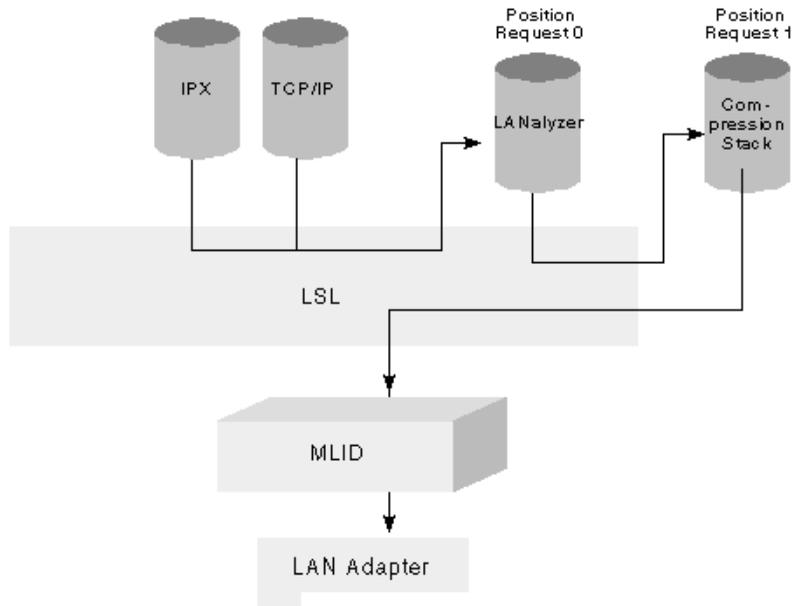
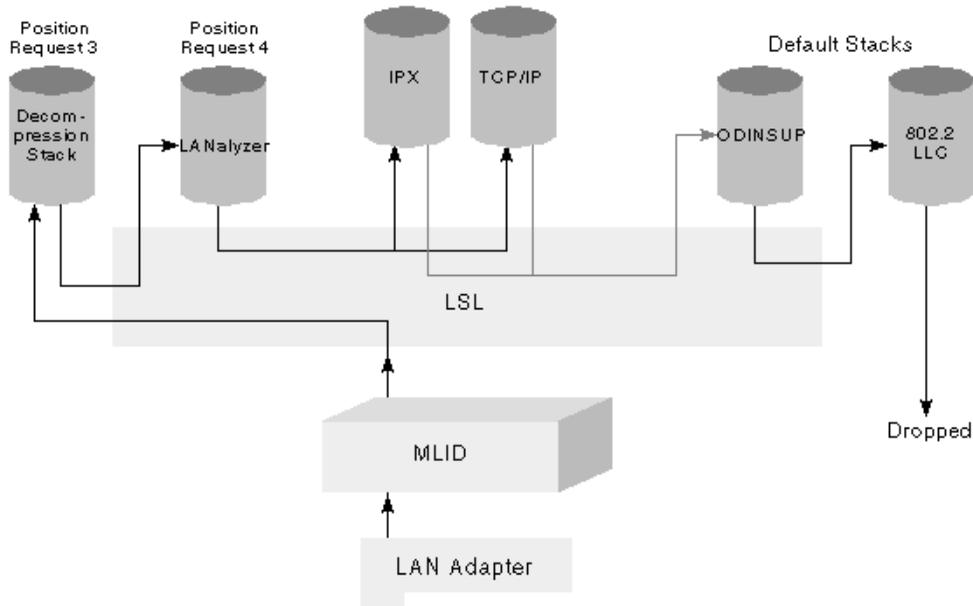


Figure 5.4: Receive Prescan Protocol Stack Chaining



If a protocol stack has registered with a chain position parameter of 2, the LSL ensures that the protocol stack is in the correct transmit and receive order. If a protocol stack loads with 0, 1, 3, or 4 as its chain position request, the protocol stack must ensure that it has registered properly for transmit and for receive. A stack that loads as a transmit stack with a chain position request equaling 0, should load as a receive stack with a chain position request equaling 4. If it loads as a transmit stack with a chain position request equaling 1, it must load as a receive stack with a chain position request equaling 3.

WARNING: A protocol stack is not prevented from loading out of order. But if out of order loading is used, it should be done carefully and rarely.

Stack Chain Mask

By default, all protocol stacks can receive direct unicast, broadcast, and direct multicast packets. This is set in the ChainMask field of the ChainStructure. Protocol stacks call *LSLModifyStackFilter* to set the ChainMask field of the ChainStructure and notify the LSL of the type of packets they want to receive. (See *LSLGetStartChain* for the definition of the *chainMask* field.)

6

Protocol Stack Packet Reception

Chapter Overview

This chapter describes the protocol stack receive routine. It details bound, prescan, and default protocol stack receive methods. The chapter also describes how the protocol stack uses the ECB when the stack receives a packet.

Protocol Stack Packet Receive Operation

A protocol stack can elect to receive all packets addressed to it, or only certain types of packets (using *LSLModifyStackFilter*).

Protocol Stack Promiscuous Mode

A protocol stack must switch promiscuous modes if it is currently receiving only packets addressed to it and it wants to start receiving packets which are not addressed to it specifically.

A protocol stack must also switch promiscuous modes if it is currently receiving all packets and it wants to start receiving only packets addressed to it.

To change promiscuous modes, the protocol stack calls the *Ctl10_MLIDPromiscuousChange* IOCTL , which causes the MLID to switch its promiscuous mode to match that of the protocol stack.

Ctl10_MLIDPromiscuousChange calls *LSLControlStackFilter* which, in turn, calls the *Ctl5_ProtocolPromiscuousChange* IOCTL.

Ctl5_ProtocolPromiscuousChange informs the protocol stack(s) bound to the MLID that the MLID has changed promiscuous modes.

If more than one protocol stack is bound to an MLID operating in promiscuous mode, a single protocol stack is not able to change the MLID out of promiscuous mode. The MLID only changes out of promiscuous mode if all protocol stacks bound to it have also changed out of promiscuous mode.

Receive Routine Events

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

The following events must occur during a protocol stack receive routine:

Table 11 Protocol Stack Receive Routine Events

Module	Action
MLID	1. Attempts to get a receive buffer from the LSL for the packet data when a packet is received. 2. Copies the packet data into the buffer if a buffer is available from the LSL. 3. Hands the buffer and its ECB to the LSL.
LSL	4. Determines which protocol stack to give the data to. 5. Calls the protocol stack's receive handler with a pointer to the receive buffer described by the ECB.
Protocol Stack	6. Processes the received packet. 7. Frees the ECB by returning it to the LSL.

Prescan and Default Protocol Stack Packet Reception

The ODI specification defines three methods of packet reception for protocol stacks:

- ◆ Bound
- ◆ Prescan
- ◆ Default

If a prescan protocol stack chain exists, older unchained prescan protocol stacks will be placed into the chain, but will not be able to request a position in the chain. This allows the logical board to chain all prescan protocol stacks.

IMPORTANT: Unless the protocol stacks are chained, we strongly discourage using the prescan or default method of packet reception for the following two reasons:

- ◆ A default protocol stack might conflict with another protocol stack using the same method of packet reception. This would prevent the use of both protocols with the same board.
- ◆ 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, the packet is not the correct type. This could cause unpredictable results.

Only specialized protocol stacks that must receive packets having a large range of Protocol IDs should use the prescan and default reception methods. For example, the 802.2 protocol stack receives packets with any Destination SAP and could use these reception methods. Protocol stacks that provide a Data-Link layer interface to the Network layer protocol stacks (for example, the 802.2 protocol stack) are candidates for using the prescan or default receive methods.

Each type of protocol stack should use the appropriate packet receive routine as follows:

Bound Protocol Stacks

Bound protocol stacks receive packets with the appropriate Protocol ID (PID) filled in the ECB's *Protocol ID* field. The PID is obtained from the low-level frame header. If a logical board has no bound protocol stack registered with it, the packet is passed to the default protocol stack registered with the logical board.

Prescan Protocol Stacks

Prescan protocol stacks receive all packets received by the logical board they are bound to before any other protocol stack receives the packets. Prescan protocol stacks are typically used for such functions as diagnostics and compression/decompression routines. The protocol stack consumes select packets and allows the other packets to be passed to the appropriate bound or default protocol stack.

Default Protocol Stacks

Default protocol stacks receive packets that are not consumed by the prescan and bound protocol stacks.

Receive ECBs

The LSL maintains a pool of receive buffers. These receive buffers are the size of the largest packet the node station can send and receive. Each receive buffer is contained in a contiguous, DWORD aligned, nonpaged region of memory. Each buffer is also described by an ECB and immediately follows that ECB in memory. In other words, receive ECBs (buffers) always contain one fragment.

When an underlying MLID receives a packet into an LSL receive buffer, the entire packet (including the low-level headers) is copied, starting immediately after the ECB *FragmentLength* field. The MLID sets the *FragmentAddress* to point past the low-level headers to the protocol header. See Chapter 4, "Protocol Stack Data Structures" for a complete description of the Event Control Block.

The Protocol Stack Receive Handler

The protocol receive handler is used for all protocol stacks default, prescan, and bound. The LSL calls a protocol stack's receive handler when an MLID receives a packet and the LSL determines that the packet is intended for that particular protocol stack. The protocol receive handler has the following processor state on entry:

- ♦ *ESI* has a pointer to a receive ECB.
- ♦ *Interrupts* are disabled.

If the protocol stack is a chained stack, it receives the following additional information:

- ♦ *EBX* has a board number.
- ♦ *EDI* has the chain ID (this ID will be given to the appropriate LSLReSubmit call).

NOTE: If the protocol stack is written in C Language, the LSL uses the following syntaxes to call the receive handler.

```
Bound protocol stack:      BoundRcvHandler (*ECB);
Chained protocol stack:  ChainRcvHandler (*ECB,
                                     BoardNumber, *ChainID);
```

The protocol receive handler has the following processor states on return:

- ♦ *Interrupts* are disabled.
- ♦ *Preserved* EBP, ESP.

If the protocol stack is a chained stack, it has the following processor states on return:

- ♦ *EAX* 0 if the protocol stack consumed the packet.
 - !0 if the protocol stack returns an ECB to the LSL.
- ♦ *ESI* if *EAX* = !0, it has a pointer to the ECB.

Handling a Receive ECB

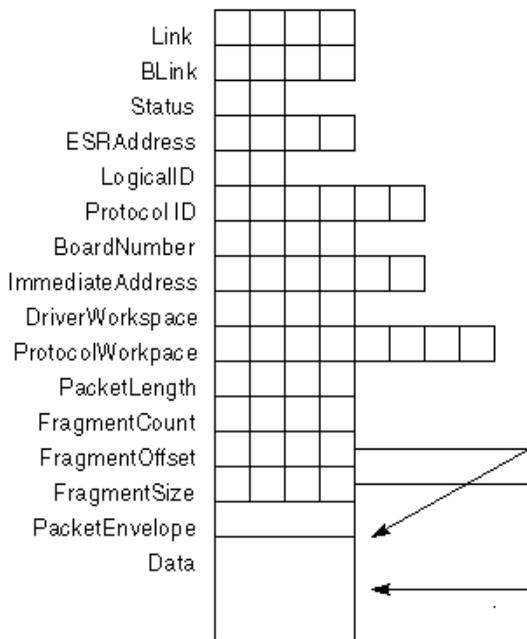
When the LSL calls the protocol receive handler, the LSL passes ownership of the ECB and its associated data buffer to the protocol stack. Before the LSL passes the ECB, either the LSL or the MLID sets the receive ECB fields as described in the following table:

Table 12 **ECB Fields Set by the LSL and the MLID**

Offset	ECB Field	Description
04h	BLink	See the "Setting the ECB <i>BLink</i> Field" section in this chapter.
08h	Status	Always set to 0000h.
0Ah	LogicalID	The Stack ID (or the Chain ID) of the protocol stack or protocol stack chain receiving the packet. The LSL assigns this value when the protocol stack registers.
10h	ProtocolID	The Protocol ID that the MLID extracted from the packet.
16h	BoardNumber	The logical board number representing the physical adapter the packet was received on and the frame type present in the packet.
1Ah	ImmediateAddresses	The source node address. Addresses passed to the upper layers are in either canonical or noncanonical format, depending upon whether the MLID bit-swaps MSB format addresses. The protocol stack fills in this field on sends. All addresses passed down to the MLID are in canonical format if the MLID is configured to be in LSB. See ODI Supplement: Canonical and NonCanonical Addressing.
20h	DriverWorkSpace	See the "Setting the DriverWorkSpace ECB Field" section in this chapter.
2Ch	PacketLength	The length of the packet (not including the MAC header).
30h	FragmentCount	Always set to 0001h.
34h	FragmentOffset	Pointer to the packet, past the MAC header. This pointer typically points to the protocol header.

Offset	ECB Field	Description
38h	FragmentSize	The length of the packet (not including the MAC header).

Figure 8 Graphic Representation of the Receive Event Control Block



Setting the ECB BLink Field

The *BLink* field is typically used as a back link to manage a list of ECBs. The current owner of the ECB uses this field. When an ECB is returned from an MLID containing a received packet, this field contains the received packet error status as defined in the following table:

Table 13 ECB BLink Error Descriptions

Bit Value	Description
0000 0001h	CRC error, such as Frame Check Sequence (FCS) error.

Bit Value	Description
0000 0002h	CRC/Frame Alignment error.
0000 0004h	Runt packet.
0000 0100h	Packet is larger than allowed by the media.
0000 0200h	The received packet is for a frame type that is not supported. For example, the logical board is not registered for the frame type of the received packet. A board number associated with the physical adapter is placed in the lookahead structure.
0000 0400h	Malformed packet. For example, the packet size is smaller than the minimum size allowed for the Media Header, such as an incomplete MAC Header. In an Ethernet 802.3 header, the length field value is larger than the total packet size.
8000 0000h	The MLID is shutting down.
No error bits set	If no error bits are set, the packet was received without error, and the data can be used. All undefined bits are cleared.

Setting the ECB DriverWorkSpace Field

Set the first byte in the DriverWorkSpace field (offset 20h) to one of the values shown in the following table:

Table 14 Values for First Byte of Driver WorkSpace Field

Bit Value	Packet Type	Description
00h	Direct	The packet is destined for this station only.
01h	Multicast	The packet is destined to a group of nodes on the network. The adapter was registered to receive packets addressed to these addresses by a call to <i>Ctl2_AddMulticastAddress</i> .
02h	Broadcast	The packet is destined to all nodes on the physical network.
04h	Remote Unicast	The packet is destined to an individual node on the network. A remote unicast address is not addressed to this adapter's node address. The protocol stack must put the adapter into promiscuous mode if it wants to receive these packets.

Bit Value	Packet Type	Description
08h	Remote Multicast	The packet is addressed to a group of nodes, but the adapter is not registered to receive it. The MLID has not called <i>Ctl2_AddMulticastAddress</i> . A protocol stack must put the adapter into promiscuous mode if it wants to receive these packets.
10h	No Source Route	The MLID received a source-routed packet, but there was no source route module (ROUTE.NLM) to record it, and the packet was not generated by the local ring. This is an exclusive bit; if this bit is set, it overrides all other bits.
20h	Error Packet	The packet contains an error. See the ECB <i>BLink</i> field for the specific error. This is an exclusive bit, if set all other bits should be 0. This value supersedes the No Source Route bit.
80h	Direct Unicast	The packet is destined for this station only.

Set the second byte (offset 21h) to indicate whether the MAC header contains one or two 802.2 control bytes as follows:

- ◆ 00h All frame types other than 802.2.
- ◆ 01h The 802.2 header has only Ctrl0 byte (Type I).
- ◆ 02h The 802.2 header has Ctrl0 and Ctrl1 (Type II).

See ODI Supplement: Frame Types and Protocol IDs for an explanation of 802.2 Type I and Type II, and Chapter 4, "Protocol Stack Data Structures" for a description of Event Control Blocks.

Description of the Protocol Receive Handler

The LSL can call this routine from the context of a hardware interrupt, or it can call this routine at process time. Protocol stacks should queue events and process them from a handler that is running at process time. Protocol stacks have the option of processing receive events as a run-to-completion events; however, this degrades performance and can cause the LAN adapter to drop packets.

If you choose to run to completion in the context in which you were called, you must switch to an internal stack to avoid overflowing the caller's stack. You must guard against reentrancy if you enable interrupts or call functions which enable interrupts.

Be aware that the network hardware is fully functional at this point and that the protocol stack must maintain packet receive order.

MLID control routines must not be invoked from this routine, because these routines must be called only at process time. However, the protocol stack can freely make requests (such as *LSLSendPacket*) to the LSL.

Chained Protocol Stacks and Resubmission

Chained receive protocol stacks process ECBs and then pass them to the LSL, which passes them to the next protocol stack in the chain. If a protocol stack cannot process an ECB immediately, it queues it to be processed at process time. When a protocol stack is done processing an ECB, it calls *LSLResubmitRcvECB* to let the next prescan, default, or bound protocol stack process it. When a protocol stack passes an ECB back to the LSL, the LSL passes it to the next protocol stack in the chain at process time.

A protocol stack can be optimized to avoid resubmitting an ECB at process time as follows:

- ◆ If a protocol stack has only minor, non-time-intensive processing to do on an ECB, it can process the ECB immediately to completion (even at interrupt time) and then pass it to the LSL, which will pass it to the next protocol stack in the chain immediately.

When a protocol stack is finished processing a receive ECB, and the ECB does not get passed to the next protocol stack in the chain, the protocol stack should return the ECB to the LSL by calling *LSLReturnRcvECB*.

7

Protocol Stack Packet Transmission

Chapter Overview

This chapter describes the send operation of the protocol stack. The chapter details how the protocol stack sends packets, uses the ECB, and ends the transmission. It also discusses the resubmitting process.

Protocol Stack Packet Transmission

Protocol stacks treat packet transmission as an asynchronous operation that entails building an Event Control Block (ECB) and calling the *LSLSendPacket* support routine (see Chapter 11, "LSL Support Routines (Assembly)"). Packets at the LSL layer and below (the Data Link layer and the MAC layer) are connectionless, and are not guaranteed to reach their destinations, nor be placed on the LAN medium.

Some protocol stacks that support transport layer capabilities must provide guaranteed packet delivery to the upper layers. If this is the case, your protocol stack must contain the necessary time-outs, retries, and packet acknowledgments to guarantee complete and accurate delivery.

Transmission Routine Events

The following events occur when a protocol stack sends a packet:

Table 15 Protocol Stack Transmission Routine

Module	Event
Protocol Stack	1. Hands the ECB to the LSL for transmission.

Module	Event
LSL	2. Calls the underlying MLID transmission handler with a pointer to the ECB. 3. Passes ownership of the ECB and its associated packet data buffers to the MLID.
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.
LSL	6. 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. If the protocol stack times out while waiting for a transmission to complete, it cannot re-use the ECB it is waiting for for any purpose until it is released by the MLID and the LSL.

Starting the Packet Transmission

Protocol stacks may transmit packets at process time or interrupt time.

If the send routine uses an internal stack and is guarded against reentry, it may enable interrupts and run processes for an extended period of time. However, the network hardware is fully functional at this point and the protocol stack must maintain packet transmission order.

The protocol stack may also run the send routine to completion, but this degrades performance.

The protocol stack cannot directly invoke MLID control routines at interrupt time. However, it may send requests to the LSL at process time or interrupt time. For example, a protocol stack may request additional ECB buffers from the LSL at any time during a send routine.

Supporting Multiple Outstanding Transmission Requests

The underlying MLIDs generally support multiple outstanding transmission requests from protocol stacks. While the LAN adapter transmits one packet onto the LAN medium, the MLID loads the next transmission packet's data onto the adapter. The number of transmissions 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 achieves its best performance by being able to pass multiple transmission requests to an underlying MLID. In theory, an MLID can handle any number of outstanding transmission requests. It will internally queue the packets it can't send.) Therefore, a protocol stack must be capable of having multiple transmissions outstanding on a particular board.

Protocol stacks can impede system performance and consequently their own performance by overwhelming the MLID with transmissions and tying up too many system resources. Protocol stacks should not transmit so many packets that it causes the MLID's Qdepth statistics counter to begin to grow.

Sending the Packet

To send a packet, the protocol stack must provide data buffers and an ECB describing the data to be sent. The protocol stack can specify from 1 to 16 data buffers per transmission request. The underlying MLID must then combine the buffers together to form a single data packet.

Raw Sends

If the ECB is sent in raw mode, the fragment list contains the complete packet, including the link-level envelope. However, the link-level envelope must be entirely contained within the first fragment. In other words, the envelope cannot be split between the first and second fragments. See also the "Raw Sends" section under "Handling a Transmit Event Control Block" in this chapter for more information.

Calling LSLSendPacket

The protocol stack transmits the packet to the appropriate MLID by calling *LSLSendPacket*. *LSLSendPacket* can be called either at process or interrupt time and has the following processor states:

On Entry

ESI	Has a pointer to a send ECB.
Interrupts	Can be in any state.

On Return

EAX	Has a completion code.
Interrupts	Are disabled but could have been enabled.
Preserved	No registers.

(For more information about *LSLSendPacket*, see Chapter 11, "LSL Support Routines (Assembly)".)

Handling a Transmit Event Control Block

An Event Control Block (ECB) is a general purpose request control block that the MLID and protocol stacks use for transmission and receive events. The protocol stack gets an ECB by calling *LSLGetRcvECBRTag*. The stack can use the *ECB's* *ProtocolWorkspace* for any purpose. Neither the LSL nor the MLID can modify the *ProtocolWorkspace* field. (See Chapter 4, "Protocol Stack Data Structures" for a more detailed discussion of ECBs.)

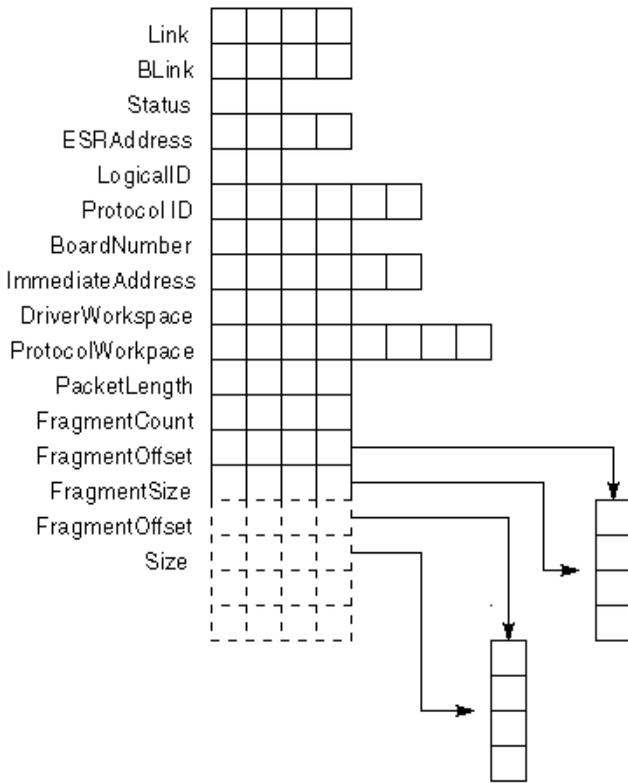
The protocol stack must set the ECB fields listed in [Table 16](#) before it gives the ECB to the LSL for transmission:

Table 16 ECB Fields to Set Before Calling *LSLSendPacket*

Offset	Field Name	Description
0Ah	ESRAddress	The address of a routine that is called when the ECB is released (after the packet has been transmitted). A pointer to the ECB is passed to this routine in ESI. This pointer is also the first parameter on the stack for ESRs written in C. This field is a near pointer.
0Eh	LogicalID	The Stack ID of the protocol stack sending the packet. See the "Raw Send" and "Priority Sends" sections below for more detail.
10h	ProtocolID	The Protocol ID (returned by <i>LSLGetPIDFromStackIDBoard</i>) that the MLID is to use when encapsulating the data. This field is ignored if a raw packet is sent. See the sections "The ECB Protocol ID Field" and "Ethernet 802.2 Frames" below for more detail.
16h	BoardNumber	The board number of the MLID that will be sending the packet.

Offset	Field Name	Description
1Ah	ImmediateAddresses	The node address on the physical network that the packet is destined for. If the packet is raw, this field is undefined. The address FFFFFFFFh always indicates a broadcast packet. (A broadcast packet is received by all nodes on the physical network.)
2Ch	PacketLength	The total length of all fragment buffers.
30h	FragmentCount	The number of fragments in the packet to be sent. Descriptor data structures follow this field. This field must contain a value between 1 and 16, inclusive. The protocol stack can specify a maximum of 16 fragment descriptors. The MLID combines these fragments together to form one contiguous packet. Note: If the MLID uses <i>GetServerPhysicalOffset</i> , the ECB and all the fragments must be convertible to a valid physical address. If the MLID is a Type 4 NLM, it must not allow the OSData segment to contain the buffers.
34h	FragmentOffset	On sends, this field describes the location of a contiguous section of RAM memory (32-bit offset).
38h	FragmentLength	The length in bytes of the first packet fragment. On sends, this value can be 0. On sends, the ECB may contain the following additional fields as needed:
3Ch	FragmentOffsetX	Additional fragment descriptor when the <i>FragmentCount</i> is greater than 1. The X stands for the additional fragment number (2 through 16).
40h	FragmentLengthX	Additional fragment descriptor when the <i>FragmentCount</i> is greater than 1. The X stands for the addition fragment number (2 through 16) . The <i>FragmentOffsetX</i> and <i>FragmentLengthX</i> fields may repeat up to 16 times.

Figure 9 Graphic Representation of the Transmit Event Control Block



The LSL and the MLIDs treat these ECB fields as read-only. Therefore, the protocol stack does not have to reinitialize each field after a transmission operation unless that field's value has changed.

Raw Sends

ODI MLIDs have the optional capability of allowing Raw Sends. A Raw Send is when a protocol stack sends the complete low-level header of a packet. In general, protocol stacks only use raw sends if the MLID cannot build the MAC header or if the protocol stack has to be frame type aware.

Because a Raw Send is optional, some MLIDs do not support it. To determine whether a particular board supports raw sends, the protocol stack must check

the RawSendBit (Bit6(0040h)] in the *MLIDModeFlags* field of the board's configuration table. If this bit is set, the MLID supports raw sends.

A protocol stack signals a raw send to the MLID by placing a value between 0FFFFh and 0FFF8h, inclusive, in the ECB's *LogicalID* field. These values correspond to priority levels between 0 and 7, inclusive. (0FFFFh equals priority level 0, the least priority.)

The protocol stack can check the MLID's *MLIDPrioritySup* configuration table field (offset 50h) for the number of priorities the MLID supports. The value in this field is zero-based.

The MLID checks the ECB's *LogicalID* field for a value between 0FFFFh and 0FFF8h. If present, the MLID skips over the code to build the MAC header. In which case, the first fragment of the ECB must contain the entire MAC header.

The first data fragment must specify the source address. However, in some cases, the source address is not used because some adapters automatically insert the source address into the MAC header.

Even though protocol stacks should be frame type unaware, protocol stacks must be aware of some frame type characteristics. However, the MLID handles minimum packet length padding and evenization. Only protocol stacks which do raw sends must be frame type aware.

Priority Sends

Protocol stacks can also support priority sends without doing raw sends. The protocol stack sends a priority packet by putting a value between 0FFF0h and 0FFF7h in the *LogicalID* field of the send packet ECB. The priority level values are defined as follows:

FFF7	priority level 0 - lowest priority
FFF6	priority level 1
FFF5	priority level 2
FFF4	priority level 3
FFF3	priority level 4
FFF2	priority level 5

FFF1	priority level 6
FFF0	priority level 7 - highest priority

The protocol stack can check the MLID configuration table *MLIDPrioritySup* field (offset 50h) for the number of priorities the MLID supports. The value in this field is zero-based.

The ECB ProtocolID Field and Ethernet 802.2 Frames

The *ProtocolID* field specifies which Protocol ID value that the MLID embeds into the frame header. This value stamps the packet as a particular protocol type (such as IPX, TCP/IP, etc.).

For example, the *ProtocolID* field for an ECB that manages an 802.2 frame type contains a Destination Service Access Point (DSAP).

When the MLID builds the frame header, the Source Service Access Point (SSAP) can be set equal to the DSAP Protocol ID. The MLID can also set the 802.2 control byte equal to 03h (UI).

However, MLIDs that support the 802.2 frame type have a special flag in the transmit ECB *ProtocolID* field that allows a protocol stack to specify the complete 802.2 Type I or Type II header (that is the DSAP, SSAP, or control byte). When this flag is present, the MLID uses the specified 802.2 header, instead of setting the SSAP Protocol ID equal to DSAP Protocol ID and the Control byte equal to 03h (the usual method).

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

Table 17 Protocol ID Field Bytes

0	1	2	3	4	5
00	00	00	00	00	DSAP
02	00	00	DSAP	SSAP	Ctrl0
03	00	DSAP	SSAP	Ctrl0	Ctrl1

Table 18 **802.2 Packet Header Bytes**

1	2	3	4
DSAP	DSAP	03	
DSAP	SSAP	Ctrl0	
DSAP	SSAP	Ctrl0	Ctrl1

The Prescan Protocol Stack Transmission Handler

The prescan protocol stack transmission handler has the following processor state on entry:

ESI	Pointer to a send ECB.
EBX	The board number.
EDI	The Chain ID.
Interrupts	Disabled.

NOTE: If the protocol stack is written in C language, the LSL uses the following syntax to call the transmission handler:

```
StackChainTransmitHandler (*ECB, BoardNumber, *ChainID);
```

The protocol stack transmission handler has the following processor state on return:

EAX	0 = the protocol stack consumed the packet. nonzero = the protocol stack returns an ECB to the LSL.
ESI	Pointer to the ECB if EAX equals nonzero.
Interrupts	Disabled.
Preserved	EBP, ESP.

NOTE: If the protocol stack is written in C language, it returns zero after it consumes the packet.

WARNING: Calling *LSLSendPacket* from a transmit prescan stack may cause the prescan stack's protocol transmission handler to be called from itself.

Chained Prescan Transmission Protocol Stacks and Resubmission

Chained prescan transmission protocol stacks process ECBs first, then pass them to the LSL. The LSL then passes them to the next protocol stack in the chain.

If a protocol stack cannot process an ECB immediately, it queues the ECB until process time and returns from its transmission handler with zero in EAX.

When a protocol stack is done processing an ECB, a prescan transmission protocol stack calls *LSLReSubmitTxECB* to pass the ECB to another protocol stack.

Some ECBs require only minor, short term processing. In such cases, instead of resubmitting the ECB, the prescan transmission protocol stack can be optimized to process the ECB immediately, even at interrupt time, and then pass it to the LSL in ESI on return from the protocol stack's transmission handler.

In some cases, a chained transmission protocol stack may decide to consume a transmission. In which case, it returns from its transmission handler with zero in EAX and does not call *LSLReSubmitTxECB*.

Prescan transmission protocol stacks must treat ECBs as read-only because the original protocol stack will still need to be able to manipulate the data in the original ECB. For example, if a prescan transmission protocol stack compresses the data, the original protocol stack will not be able to read it.

If a prescan transmission protocol stack must modify the data in an ECB (such as with compression stacks), the protocol stack must copy the ECB and modify the copy of the ECB only.

The protocol stack also saves the Event Service Routine (ESR) of the original ECB. When the ESR is called, the protocol stack must either call the original ESR using the pointer to the original ECB, in the ESI register, or it must place the original ECB in the LSL's event queue and call *LSLServiceEvents*.

The amount of data that the prescan stack can transmit is limited to the size contained in the MLID configuration table *MLIDRecvSize* field.

Transmission Complete

The following events occur to complete the packet transmission:

Table 19 **Transmission Complete Event**

Module	Action
Protocol Stack	1. Gives the ECB to the MLID through the LSL.
MLID	2. Transmits the ECB. 3. Returns the ECB to the LSL. (If the MLID uses <i>LSLSendComplete</i> to return the ECB, proceed to Step 4. If the MLID uses <i>LSLFastSendComplete</i> to return the ECB, skip to Step 7.)
LSL	4. Places the ECB into a temporary event queue.
MLID	5. Calls <i>LSLServiceEvents</i> 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's ESR field. In the case of a transmission complete, the ESR will be the protocol stack's transmission complete handler. (See the Protocol Transmission Complete Handler section in this chapter).

NOTE: The MLID can invoke the protocol stack's Transmission Complete Handler before the call to *SendPacket* returns.

The protocol stack's Transmission Complete Handler is described in the next section.

Protocol Transmission Complete Handler

This routine must complete quickly because it is usually invoked from an Interrupt Service Routine. Transmission requests can be issued from this routine, but the protocol stack must not poll for a completed transmission unless the protocol switches to its own stack and allows reentry. However, we strongly recommend against doing this because it degrades protocol stack performance.

The Protocol Transmission Complete Handler has the following processor entry state:

ESI	Pointer to the completed ECB.
-----	-------------------------------

Interrupts	Disabled.
------------	-----------

NOTE: If the transmit complete handler is written in C, the ECB is the first parameter on the stack.

The protocol Transmission Complete Handler has the following processor return state:

Interrupts	Disabled.
------------	-----------

Preserved	EBP, ESP.
-----------	-----------

8

Protocol Stack Control Routines

This chapter describes the control commands that your protocol stack must provide to support the MPI interface.

When an NLM calls a protocol stack control command, that NLM must place a function code into EBX and call the protocol stack control entry point. The address of the entry point is obtained by calling *LSLGetProtocolControlEntry*.

The value returned in EAX is always generated so that the Z flag is set correctly. If the call completes with no error, EAX will be 0 (and the Z flag set). If the call completes with an error, EAX will be nonzero (and the Z flag clear). The value in EAX indicates the error.

The following list indicates which calls are optional and which calls must be supported:

Ctl0_GetProtocolStackConfiguration	Required
Ctl1_GetProtocolStackStatistics	Required
Ctl2_Bind	Required (bound stacks only)
Ctl3_Unbind	Required (bound stacks only)
Ctl4_MLIDDeRegistered	Optional
Ctl5_ProtocolPromiscuousChange	Required
Ctl100_GetProtocolStringForBoard	Required
Ctl101_GetBoundNetworkInfo	Required

NOTE: EAX and the Z flag are undefined for calls that return no error.

The following table lists the IOCTLs described in this chapter and provides a summary of each one.

Table 20 Summary of Protocol Stack Control Functions

Control Function Type	Summary	Function Name
Binding and Unbinding Functions	Bind Protocol Stack to MLID	Ctl2_Bind
	Unbind from MLID	Ctl3_Unbind
Table Retrieval	Get Pointer to Configuration Table	Ctl0_GetProtocolStackConfiguration
	Get Pointer to Statistics Table	Ctl1_GetProtocolStackStatistics
NLM Interaction Routines	Get an ID string	Ctl100_GetProtocolStringForBoard
	Inform of Deregistered MLID	Ctl4_MLIDDeRegistered
	Inform of Requested Promiscuous Mode Change on Adapter	Ctl5_ProtocolPromiscuousChange

Ctl0_GetProtocolStackConfiguration

Returns a pointer to the protocol stack configuration table.

Entry State

EBX

Equals 0.

Interrupts

Enabled.

Return State

EAX

Equals 0.

ESI

Pointer to the protocol stack configuration table.

Interrupts

Enabled.

Preserved

EBP, ESP.

Completion Code (EAX)

0	Successful
---	------------

Remarks

GetProtocolStackConfiguration returns a pointer to the protocol stack configuration table.

See Also

Chapter 4, "Protocol Stack Data Structures" for the protocol stack configuration table format.

Ctl1_GetProtocolStackStatistics

Returns a pointer to the protocol stack statistics table.

Entry State

EBX

1

Interrupts

Enabled.

Return State

EAX

0

ESI

Pointer to a statistics table.

Interrupts

Enabled.

Preserved

EBP, ESP.

Completion Code (EAX)

0	Successful
---	------------

Remarks

GetProtocolStackStatistics returns a pointer to the protocol stack statistics table.

See Also

Chapter 4, "Protocol Stack Data Structures" for the protocol stack statistics table format.

Ctl2_Bind

Binds a protocol stack with an MLID.

Entry State

EBX

2

EDI

The board number that the protocol stack will bind to.

ESI

Pointer to a user-specified parameter string.

Interrupts

Enabled.

Return State

EAX

Completion Code.

Interrupts

Enabled.

Preserved

EBP, ESP.

Completion Code (EAX)

0	Successful The protocol stack was successfully bound.
FFFFFFF3	DuplicateEntry The protocol stack is already bound.

Remarks

Ctl2_Bind provides a consistent method of binding a protocol stack with an MLID. The protocol stack is expected to issue the *LSLBindStack* call to the Link Support Layer, as well as to perform any other maintenance commands required to bind to an MLID.

This function is invoked when the user issues the Bind command. For example:

```
bind ipx to ne1000
```

NOTE: This control routine is required for bound protocol stacks only.

See Also

Appendix A, "Operating System Support Routines": BindProtocolToBoard.

Ctl3_Unbind

Unbinds a protocol stack from an MLID.

Entry State

EBX

3

EDI

The board number to unbind the protocol stack from.

ESI

Pointer to an implementation-dependent parameter string. If ESI equals 0, no string exists.

Interrupts

Enabled.

Return State

EAX

0

Interrupts

Enabled.

Preserved

EBP, ESP.

Completion Code (EAX)

0	Successful
---	------------

Remarks

Ctl3_Unbind provides a consistent method of unbinding a protocol stack from an MLID.

This function is invoked when the user has issued the unbind command. For example:

```
unbind ipx from ne1000
```

Optionally, this routine can send a few packets before the protocol stack is unbound.

NOTE: This control routine is required for bound protocol stacks only.

Ctl4_MLIDDeRegistered

Informs the protocol stack that the MLID has deregistered.

Entry State

EBX

4

EDI

The number of the board being deregistered.

Interrupts

Enabled.

Return State

Interrupts

Enabled.

Preserved

EBP, ESP.

Completion Code (EAX)

0	Successful
---	------------

Remarks

The Link Support Layer uses *MLIDDeRegistered* to inform all protocol stacks bound to a specific MLID that the MLID has deregistered.

As a result of this call, the MLID will no longer be available. Be aware that the MLID calls this IOCTL instead of the Unbind IOCTL if the MLID can no longer send packets because of a hardware failure. If this is the case, the protocol stack must clean up its tables that are related to the logical board

number contained in EDI. The protocol stack should no longer attempt to transmit any packets through this logical board.

Ctl5_ProtocolPromiscuousChange

Informs the protocol stack that a promiscuous mode change has been requested on the adapter.

Entry State

EAX

The board number of the MLID.

EBX

5

ESI

The promiscuous mode state of the driver:

- ◆ Bit 0 is set if all MAC frames are to be received.
- ◆ Bit 1 is set if all non-MAC frames are to be received.
- ◆ Bit 2 is set if FDDI SMT type MAC frames are to be received.
- ◆ Bit 3 is set if remote multicast frames are to be received.

Interrupts

Disabled.

Return State

Interrupts

Preserved.

Completion Code (EAX)

0	Successful
---	------------

Remarks

Ctl5_ProtocolPromiscuousChange informs the protocol stack that another protocol stack has requested the LAN adapter to change its promiscuous mode. The MLID calls this routine before it responds to another protocol stack's request to change its promiscuous mode. The protocol stack makes this request by calling *MLIDPromiscuousChange*.

Protocol stacks implement *ProtocolPromiscuousChange* if they must be aware that the MLID is switching its promiscuous mode from promiscuous-mode-disabled to promiscuous-mode-enabled. In other words, the MLID has been sending only qualified packets to the protocol stack, but will now be sending all packets to the protocol stack.

ProtocolPromiscuousChange notifies the protocol stack of this change so that the protocol stack can adjust its filtering methods.

An example of this would be a protocol stack (stack A) that enabled itself to receive packets with remote unicast addresses after it has enabled a hardware filtering mechanism through the MLID's *Ctl14_DriverManagement* IOCTL. If another protocol stack (stack B) enables promiscuous mode, stack A must be aware that the hardware filter is disabled and that it will be receiving all unicast packets.

Ctl100_GetProtocolStringForBoard

Gets a unique ID string for a protocol.

Entry State

EBX

100h

EDI

The board number.

ESI

Pointer to the user-supplied buffer (at least 17 bytes) where the string will be returned.

Interrupts

Enabled.

Return State

EAX

0

ESI

Pointer to the buffer holding the NULL-terminated protocol description string of length 16 bytes or less.

Interrupts

Enabled.

Preserved

EBP, ESP.

Completion Code (EAX)

0	Successful
---	------------

Remarks

This function gets a unique ID string for a protocol.

For example, an IPX protocol stack might return a string similar to Network FADE2200 for the board on which the protocol stack is functioning. In this string, the IPX network number, FADE2200, is being used with that particular board.

A TCP/IP protocol stack might return a string similar to 128.34.31.01.

In the future, this protocol IOCTL will become function number 7 instead of 100. Currently, protocol stacks should accept 7 or 100 as valid. However, users of the this IOCTL should begin using 7 instead of 100.

Ctl101_GetBoundNetworkInfo

Gets the bound network address.

Entry State

EBX

101h

EDI

The board number.

ESI

Pointer to the user-supplied buffer for the bound network address.

Interrupts

Enabled.

Return State

Interrupts

Enabled.

Preserved

EBP, ESP.

Remarks

The protocol control function is entered with ESI pointing to a user-supplied buffer of type struct networkAddressStruct, and with EDI containing the board number. The protocol stack fills in the networkAddressStruct.address type with its own transport type (IPX_TRANSPORT_ADDRESS for IPX), networkAddressStruct.size with the length of the address, and networkAddressStruct.address with the bound network address. IPX needs to return all 12 bytes, network:node:socket. IP needs to return 4 bytes, network address only (no socket).

In the future, this protocol IOCTL will become function number 9 instead of 101. Currently, protocol stacks should accept 9 or 101 as valid. However, users of the this IOCTL should begin using 9 instead of 101.

9

Overview of the LSL

Chapter Overview

This chapter provides a brief overview of the LSL and its functions. It also documents the completion codes the LSL returns in the support routines.

Link Support Layer (LSL)

The Link Support Layer (LSL) handles the communication between protocol stacks and MLIDs. The ODI allows a physical topology to support many different types of protocols. MLIDs send and receive packets of different frame types, destined for different protocol stacks. The LSL acts as a demultiplexer, or switchboard, and determines which protocol stack or MLID 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.

The LSL provides the following services for protocol stacks:

- ◆ Queues and recovers ECBs for later use.
- ◆ Maintains lists of all active protocol stacks and MLIDs.

The LSL also provides protocol stacks with the ability to perform the following tasks:

- ◆ Get and return ECBs.
- ◆ Get timing services.
- ◆ Get stack IDs and protocol IDs.

- ◆ Get MLID statistics.
- ◆ Bind with MLIDs.
- ◆ Transmit and receive packets through the MLID.
- ◆ Get information about MLIDs and other protocol stacks.
- ◆ Change the operational state of an MLID (such as shutting down or resetting an MLID).

The following table shows the completion codes returned by the LSL.

Table 21 LSL Completion Codes

Value	Message
00000000h	Successful
0FFFFFF81h	BadCommand
0FFFFFF82h	BadParameters
0FFFFFFCh	Canceled
0FFFFFF83h	DuplicateEntry
0FFFFFF84h	Fail
0FFFFFF85h	ItemNotPresent
0FFFFFF86h	NoMoreItems
0FFFFFF87h	NoSuchDriver
0FFFFFF88h	NoSuchHandles
0FFFFFF89h	OutOfResources
0FFFFFF8Ah	RxOverflow
0FFFFFF8Bh	InCriticalSection
0FFFFFF8Ch	TransmitFailed
0FFFFFF8Dh	PacketUndeliverable

10 The LSL Statistics Table

Chapter Overview

This chapter describes the fields in the LSL Statistics Table. NLMs do not modify any of the LSL Statistics Table fields.

LSL Statistics Table

The LSL keeps a statistics table for the purpose of network management. The examples below include sample code and a graphic representation of the LSL statistics table. [Table 22](#) describes the statistics table fields.

Figure 10 LSL Statistics Table Sample Source Code

MajorVersion	db	01
MinorVersion	db	07
TotalTXPackets	dd	00
GetECBBfrs	dd	00
GetECBFAILs	dd	00
AESEventCounts	dd	00
PostponedEvents	dd	00
ECBCxIFails	dd	00
ValidBfrsReused	dd	00
EnqueuedSendCnt	dd	00
TotalRXPackets	dd	00
UnclaimedPackets	dd	00
NumCustom	dw	00
ErrorAllocatingMoreRxBuffers	dd	02
ErrorAllocatingDescription	dd	00
LogicalBoardStatDescription	dd	00
LogicalBoardStatTable	dd	00

Figure 11 Graphic Representations of the LSL Statistics Table

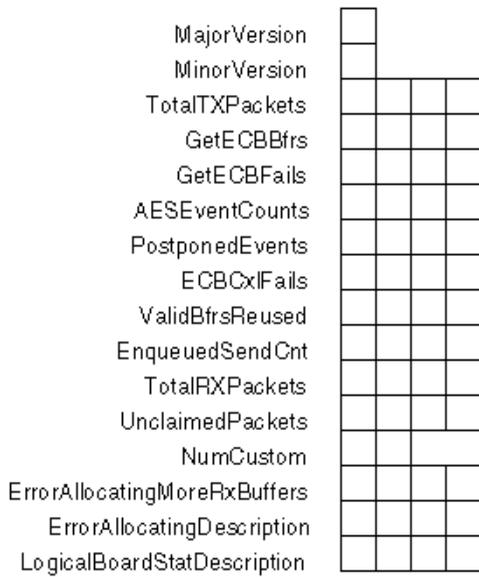


Table 22 LSL Statistics Table

Offset	Label	Size	Description
00h	MajorVersion	1	The major version number of the LSL statistics table. The current major version number is 1.
01h	MinorVersion	1	The minor version number of the LSL statistics table. The current minor version number is 00.
02h	TotalTxPackets	4	The number of packet transmit requests, regardless of whether the packets were actually transmitted.
06h	GetECBBfrs	4	The total number of ECBs that were requested.
0Ah	GetECBFails	4	The total number of ECB requests that failed.
0Eh	AESEventCounts	4	The total number of AES events that have been processed.
12h	PostponedEvents	4	The number of AES events postponed because of critical sections.

Offset	Label	Size	Description
16h	ECBCxIFails	4	The number of AES cancel requests that failed because the event was not found on the AES list.
1Ah	ValidBfrsReused	4	The number of ECBs in the hold queue that were reused before they were removed from the hold queue.
1Eh	EnqueuedSendCnt	4	The number of send events in the queue that have occurred.
22h	TotalRxPackets	4	The total number of received incoming packets.
26h	UnclaimedPackets	4	The total number of received incoming packets.
2Ah	NumCustom	2	The total number of custom counters that follow this field. Currently, this value is 2.
2Ch	ErrorAllocatingMoreReceiveBuffers	4	Custom counter.
30h	LogicalBoardStatTable	4	Custom counter.
34h	ErrorAllocatingDescription	4	Pointer to a string describing the counter.
38h	LogicalBoardStatDescription	4	Pointer to a string describing the counter.

NumCustom dwords corresponding to the custom statistics for the LSL start at offset 2Ch. A dword holding the offset of a string describing each custom counter follows the NumCustom dwords. Each string is length-preceded and null-terminated.

The LSL Logical Board Statistics Structure

The LSL also declares a pointer to an array of LogicalBoardStatStructures. The pointer and the LogicalBoardStatStructure are declared as follows:

```
public LogicalBoardStatTable
        LogicalBoardStatTable db(MaximumNumberOfLans* size
                                LogicalBoardStatStructure)

LogicalBoardStatTable struct
    LTotalTxPackets dd ? ;count of packets the logical board transmitted
    LTotalRxPackets dd ? ;count of packets the logical board received
    LUnclaimedPackets dd ? ;count of packets not claimed by any logical board
    LReserved dd ?
```

LogicalBoardStatsStructure end

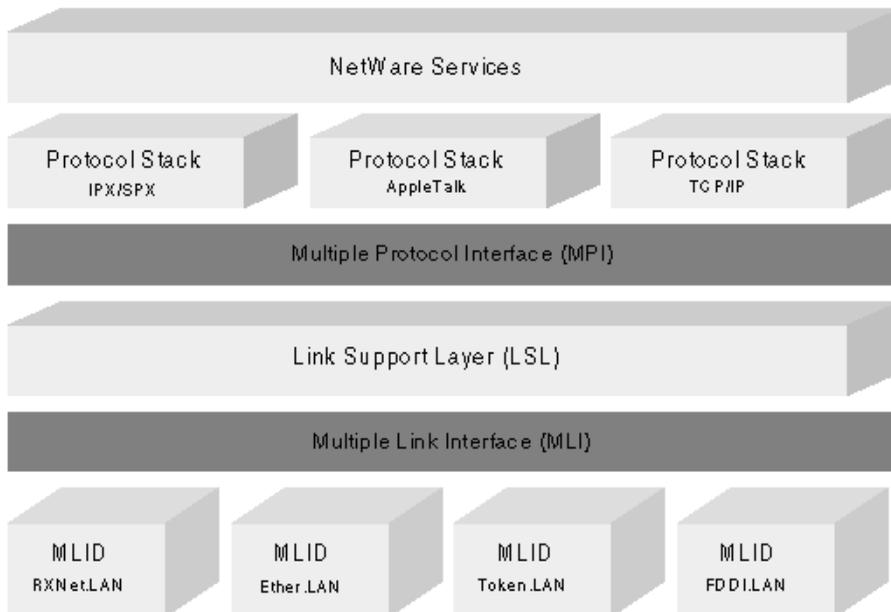
11

LSL Support Routines (Assembly Language)

This chapter describes the Link Support Layer (LSL) which contains the support functions that comprise the Multiple Protocol Interface (MPI) and the Multiple Link Interface (MLI).

Figure 11-1 is a block diagram illustrating these interfaces. The functions in this chapter are available to both protocol stacks and MLIDs.

Figure 11.1: ODI Elements



NOTE: EAX and the Z flag are undefined for calls that return no errors. In NetWare 3+ and NetWare 4+ environments, MLIDs access the APIs that are contained in this chapter as near calls.

Table 23 List of LSL Support Functions

Function Type	Function	Purpose
Binding and Registration Functions	LSLBindStack	Binds a protocol stack to an MLID.
	LSLRegisterDefaultChain	Places a default protocol stack into a protocol stack chain for an MLID.
	LSLRegisterPreScanRxChain	Places a receive prescan protocol stack into a protocol stack chain for an MLID.
	LSLRegisterPreScanTxChain	Places a transmit prescan protocol stack into a protocol stack chain for an MLID.
	LSLRegisterStackRTag	Registers a bound stack with the LSL.
	LSLRegisterStackSMPSafe	Informs the LSL that the stack is SMP aware.
	LSLRegisterMLIDRtag	Registers a logical board.
Miscellaneous Chaining Functions	LSLGetStartChain	Pointer to the start of all protocol stack chains for a given board.
	LSLModifyStackFilter	Modifies the protocol stack reception mask.
	LSLControlStackFilter	Notifies the protocol stack of an MLID status change.
NLM Interaction Functions	LSLAddProtocolID	Adds a Protocol ID to the LSL statistics table.
	LSLGetMLIDControlEntry	Gets the control entry point of an MLID.
	LSLGetProtocolControlEntry	Gets the control entry point of an MLID.
	LSLGetLinkSupportStatistics	Gets a pointer to the LSL statistics table.
	LSLGetPIDFromStackIDBoard	Gets a Protocol ID for a registered bound protocol stack and logical board combination.
	LSLGetBoundBoardInfo	Gets the Stack IDs of the protocol stacks bound to a specified board.

Function Type	Function	Purpose
Packet Reception Routines	LSLGetStackIDFromName	Gets the Stack ID for a registered bound protocol stack.
	LSLDeFragmentECB	Consolidates packet fragments.
	LSLGetMaximumPacketSize	Determines the maximum size of the ECB.
	LSLGetPhysicalAddressOfECB	Gets the physical memory address of a receive ECB.
	LSLGetSizedRcvECBRTag	Gets a receive ECB.
	LSLReSubmitDefaultECB	Passes the ECB to another default protocol stack for processing.
	LSLReSubmitPreScanRxECB	Passes the ECB to another receive prescan stack for processing.
	LSLFastRcvEvent	Passes the ECB to the protocol stack.
	LSLHoldRcvEvent	Queues the receive ECB until LSLServiceEvents is called.
	LSLServiceEvents	Removes packets from the LSL's queue and dispatches them to the appropriate protocol.
Packet Transmission Functions	LSLReturnRcvECB	Returns a previously allocated receive ECB.
	LSLGetMaximumPacketSize	Determines the maximum size of the ECB.
	LSLReSubmitPreScanTxECB	Passes the ECB to another transmit prescan stack for processing.
	LSLSendComplete	Queues a send ECB until LSLServiceEvents is called.
	LSLFastSendComplete	Returns the send ECB directly to the owner.
Unbinding and DeRegistration Functions	LSLSendPacket	Sends a packet to the MLID.
	LSLDeRegisterStack	Deregisters a bound protocol stack from the LSL.

Function Type	Function	Purpose
	LSLDeRegisterMLID	Deregisters a logical board.
	LSLDeRegisterStackSMPSafe	Deregisters an SMP aware protocol stack.
	LSLDeRegisterPreScanRxChain	Removes a receive prescan protocol stack from a protocol stack chain for an MLID.
	LSLDeRegisterPreScanTxChain	Removes a transmit prescan protocol stack from a protocol stack chain for an MLID.
	LSLUnbindStack	Unbinds a bound protocol stack from an MLID.
	LSLDeRegisterDefaultChain	Unbinds a chained default protocol stack from an MLID.
	LSLUnbindThenDeRegisterMLID	Unbinds and deregisters an MLID.
SMP Specific Functions	LSLSMPReaderLock	Gets a read lock.
	LSLSMPReaderUnlock	Releases a read lock.
	LSLSMPWriterLock	Gets a write lock.
	LSLSMPWriterUnlock	Releases a write lock.
	LSLSMPReaderToWriterLock	Converts a read lock to a write lock.
	LSLWriterToReaderLock	Converts a writer to a read lock.
	LSLAssignMutexToInstance	Assigns a mutex to an instance of a logical adapter.
	LSLRemoveMutexFromInstance	Removes a mutex from an instance of a logical adapter.
	LSLRemovePhysicalMutex	Removes a mutex from all instances of a physical adapter.
	LSLAdapterMutexLock	Gets a hardware adapter mutex lock.
	LSLAdapterMutexTryLock	Gets a hardware adapter mutex lock.
	LSLAdapterMutexUnlock	Releases a hardware adapter mutex lock.
	LSLAddPollingProcedure	Registers an SMP-aware polling procedure.

Function Type	Function	Purpose
	LSLRemovePollingProcedure	Deregisters an SMP-aware polling procedure.
	LSLAddTimerProcedure	Registers an SMP aware timer-driven callback procedure.
	LSLRemoveTimerProcedure	Deregisters an SMP-aware timer callback procedure.

The old style registration and deregistration functions are no longer supported in this specification. [Table 24](#) shows the correlation between the old and the new registration and deregistration functions:

Table 24 Correlation of Old and New Registration and Deregistration Functions

Old Function	New Function	Description
LSLDeRegisterDefaultStack	LSLDeRegisterDefaultChain	Unbinds a chained default protocol stack from a specific logical board or LAN adapter.
LSLRegisterDefaultStackRTag	LSLRegisterDefaultChain	Binds a chain of default protocol stacks to a specific logical board or LAN adapter. A chained default protocol stack uses this call to request and register its position in the chain.
LSLDeRegisterPreScanStack	LSLDeRegisterPreScanRxChain	Unbinds a chained receive prescan protocol stack from a specific logical board or LAN adapter.
	LSLDeRegisterPreScanTxChain	Unbinds a chained transmit prescan protocol stack from a specific logical board or LAN adapter.
LSLRegisterPreScanStackRTag	LSLRegisterPreScanRxChain	Binds a single receive prescan protocol stacks to a specific logical board or LAN adapter. A chained receive prescan protocol stack uses this call to register and request its position in the chain.

Old Function	New Function	Description
	LSLRegisterPreScanTxChain	Binds a single transmit prescan protocol stack to a specific logical board or LAN adapter. A chained transmit prescan protocol stack uses this call to register and request its position in the chain.
LSLDeRegisterStack	LSLDeRegisterStack (unchanged)	Deregisters a bound protocol stack from a specific logical board or LAN adapter.
LSLRegisterStackRTag	LSLRegisterStackRTag (unchanged)	Registers a bound protocol stack to a specific logical board or LAN adapter.

IMPORTANT: The LSL register and deregister functions must be used in the following pairs:

- ◆ LSLRegisterPreScanRxChain
- ◆ LSLDeRegisterPreScanRxChain
- ◆
- ◆ LSLRegisterPreScanTxChain
- ◆ LSLDeRegisterPreScanTxChain
- ◆
- ◆ LSLRegisterDefaultChain
- ◆ LSLDeRegisterDefaultChain
- ◆
- ◆ LSLRegisterStack
- ◆ LSLDeRegisterStackRTag
- ◆
- ◆ LSLRegisterStackSMPSafe
- ◆ LSLDeRegisterStackSMPSafe

LSLAdapterMutexLock

Gets the hardware adapter mutex lock for a given logical board; generally used by MLIDs.

Entry State

EBX

The logical board number.

Interrupts

Disabled and must remain disabled until after calling *LSLAdapterMutexUnlock*.

Call

At process time only.

Return State

Interrupts

Unchanged.

Preserved

All other registers.

Remarks

SMP aware MLIDs call this function at the beginning of critical sections of code that require exclusive access to an adapter's resources.

This function uses an atomic spin lock and should only be called at process time. If the mutex cannot be acquired, this function spins until the mutex can be acquired. This function will not return until the mutex is acquired.

NOTE: Beware of deadlocks when using this function.

See Also

- ◆ LSLAdapterMutexTryLock

- ◆ LSLAdapterMutexUnlock
- ◆ LSLAssignMutexToInstance
- ◆ LSLRemovePhysicalMutex

LSLAdapterMutexTryLock

Gets the hardware adapter mutex lock for a given logical board; generally used by MLIDs.

Entry State

EBX

The logical board number.

Interrupts

Disabled and must remain disabled until after calling *LSLAdapterMutexUnlock*.

Call

At process or interrupt time.

Return State

EAX

Completion Code.

Interrupts

Unchanged.

Preserved

All other registers.

Completion Code (EAX)

0	Successful The lock was obtained.
1	Failed The lock was not obtained.

Remarks

SMP aware MLIDs call this function at the beginning of critical sections of code that require exclusive access to a hardware adapter's resources. This function is typically called repeatedly until the lock is obtained or until a maximum number of attempts has been reached. If the lock cannot be obtained, the intended operation must be aborted or postponed until a later time.

See Also

- ◆ LSLAdapterMutexTryLock
- ◆ LSLAdapterMutexUnlock
- ◆ LSLAssignMutexToInstance
- ◆ LSLRemovePhysicalMutex

LSLAdapterMutexUnlock

Releases the hardware adapter mutex for a given logical board; generally used by MLIDs.

Entry State

EBX

The logical board number.

Interrupts

Can be in any state.

Call

At process or interrupt time.

Return State

Interrupts

Unchanged.

Preserved

All other registers.

Remarks

SMP-aware MLIDs call this function after completion of critical code requiring exclusive access to an adapter's resources. The released lock must have been obtained earlier using *LSLAdapterMutexLock* or *LSLAdapterMutexTryLock*.

See Also

- ◆ *LSLAdapterMutexTryLock*
- ◆ *LSLAdapterMutexUnlock*
- ◆ *LSLAssignMutexToInstance*
- ◆ *LSLRemovePhysicalMutex*

LSLAddPollingProcedure

Used by polled drivers to register a polling procedure in the SMP environment; generally used by MLIDs.

Entry State

EBX

The physical board ID.

ESI

Pointer to the polling procedure.

Interrupts

Can be in any state.

Call

At process time only.

Return State

Interrupts

Unchanged.

Preserved

All other registers.

Remarks

SMP-aware polled MLIDs call this function to register their polling procedures with SMP. This call is used in place of *AddPollingProcedureRTag*, which is used for nonSMP MLIDs.

See Also

- ◆ LSLAllocatePhysicalBoardID
- ◆ LSLRemovePollingProcedure

- ◆ AddPollingProcedureRTag in Appendix A, "Operating System Support Routines"

LSLAddProtocolID

Adds the Protocol ID to the LSL statistics table; generally used by protocol stacks and MLIDs.

Entry State

EAX

Pointer to the 6-byte Protocol ID that is being added.

ECX

Pointer to the length-preceded, zero-terminated media/frame type description string with a length of less than or equal to 15 bytes.

EDX

Pointer to a length-preceded, zero-terminated string containing the protocol stack short name. This protocol stack receives incoming frames with the Stack ID that EAX points to.

Interrupts

Can be in any state.

Call

At process time only.

Return State

EAX

Completion Code

Interrupts

Unchanged.

Preserved

No other registers.

Completion Code (EAX)

0000000h	Successful	The LSL successfully added the new Protocol ID.
0FFFFFF82h	BadParameter	The specified parameter is an illegal (unknown) name. The protocol name string and media name string length must be equal to or less than 15.
0FFFFFF83h	DuplicateEntry	A different protocol ID is already registered for the given media/frame type/protocol stack combination.
0FFFFFF89h	OutOfResources	The LSL has no resources to register another Protocol ID.

NOTE: MLIDs normally ignore the return code.

Remarks

Protocol Stacks

LSLAddProtocolID allows a protocol stack to add a new PID for a given media. Because protocol stacks are topology and frame-type unaware, they do not interpret Protocol IDs. Protocol IDs depend on the frame type and topology the protocol stack is using.

The network administrator usually enters a Protocol ID for each frame-type/board combination in the AUTOEXEC.NCF file. The protocol stack then calls *LSLGetPIDFromStackIDBoard* to get the Protocol ID.

If a protocol stack does not find a Protocol ID, it loads, but is not functional. In which case, the network administrator can bind the protocol stack to a frame type by specifying a Protocol ID from the command line using the following syntax:

```
Protocol Register<Protocol Name><Frame Name><Protocol ID>
```

For example:

```
Protocol Register IP Ethernet_II 800
```

Typing the word "Protocol" at the command line will produce a list of all protocols, frame types, and Protocol IDs that need to be registered.

Intelligent protocol stacks that cannot find a Protocol ID can register based on common, well-known protocol stack / frame type combinations. For example, IP on Ethernet_II or Ethernet_SNAP always use Protocol ID 800.

MLIDs

LSLAddProtocolID allows the MLID to tell the LSL the names and Protocol IDs (PIDs) of each protocol stack it can support. The MLID initialization procedure should call this routine to add the default PID for IPX.

NOTE: If a Protocol ID value is less than 6 bytes, the most significant bytes must be padded with zeros (0).

See Also

- ◆ ODI Supplement: Frame Types and Protocol IDs

LSLAddTimerProcedure

Registers a timer based call back procedure in the SMP environment; generally used by MLIDs.

Entry State

EBX

The physical board ID.

ESI

Pointer to a timer node data structure.

Interrupts

Can be in any state.

Call

At process or interrupt time.

Return State

Interrupts

Unchanged.

Preserved

All other registers.

Remarks

SMP-aware MLIDs call this function to register a timer driven call back procedure with SMP. Note that this call must be used instead of *ScheduleInterruptTimeCallBack* in the SMP environment.

The timer node data structure used by this function is in the same format used by *ScheduleInterruptTimeCallBack*.

See Also

- ◆ LSLAllocatePhysicalBoardID

- ◆ LSLRemoveTimerProcedure
- ◆ ScheduleInterruptTimeCallBack in Appendix A, "Operating System Support Functions".

LSLAllocatePhysicalBoardID

Assigns a unique ID to a physical board.

Entry State

EAX

Pointer to MLID resource tag.

ECX

0 if the MLID must run on processor 0.

-1 if the MLID can run on any processor.

Interrupts

Can be in any state.

Call

At process time only.

Return State

EAX

Completion code.

EBX

Physical board ID

Completion Code (EAX)

Successful	A unique physical board ID has been assigned.
Bad parameter	EAX does not point to a valid resource tag.
Out of Resources	A unique physical board ID could not be allocated.

See Also

- ◆ [LSLFreePhysicalBoard](#)

LSLAssignMutexToInstance

Assigns a mutex lock for a specified logical board instance and allocates the mutexes for the physical adapter if they have not already been allocated; generally used by MLIDs.

Entry State

EBX

The logical board that the mutex will be assigned to.

ESI

The physical board ID associated with the physical adapter. This number is returned by *LSLAllocatePhysicalBoardID*.

Interrupts

Can be in any state.

Call

At process or interrupt time.

Return State

EAX

Completion code.

Interrupts

Unchanged.

Preserved

All other registers.

Completion Code (EAX)

00000000h

Successful

The mutex was successfully assigned to an instance of a physical board.

00000001h

Failed

The mutex could not be allocated.

Remarks

SMP-aware MLIDs call this function for each logical board instance. Typically this function is called soon after calling *LSLRegisterMLIDRTag*.

See Also

- ◆ *LSLAllocatePhysicalBoardID*
- ◆ *LSLRemoveMutexFromInstance*
- ◆ *LSLRemovePhysicalMutex*

LSLBindStack

Enters the Stack ID into the LSL's table. Generally used by bound protocol stacks.

Entry State

EAX

The Stack ID number.

EBX

The board number that the protocol stack will bind to.

Interrupts

Can be in any state.

Call

At process time only.

Return State

EAX

Completion Code.

Interrupts

Preserved and never changed.

Preserved

No other registers.

Completion Code (EAX)

00000000h

Successful

The LSL successfully added the protocol stack to its table.

0FFFFFF82h	BadParameter	The MLID corresponding to the requested board number or the protocol stack corresponding to the specified stack ID does not exist.
0FFFFFF83h	DuplicateEntry	The specified binding already exists.
0FFFFFF85h	ItemNotPresent	The frame type specified by the logical board number does not have a PID registered for this protocol stack.
0FFFFFF89h	OutOfResources	The LSL has no resources to register another protocol stack.

Remarks

LSLBindStack binds a protocol stack to an MLID. The protocol stack receives a packet from the MLID as follows:

1. The MLID places the board number of the MLID, the Protocol ID from the MAC header, and the data into a receive ECB.
2. MLID passes the ECB and the packet to the Link Support Layer.
3. LSL checks its statistics table to find the Protocol ID that matches the one in the ECB.
4. LSL uses the Protocol ID to reference the Stack ID in the statistics table.
5. LSL passes the packet and the ECB to the receive handler of the protocol stack that corresponds to the Stack ID.

The protocol stack specified the address receive handler when it registered with the LSL.

Before making this call, the protocol stack must be ready to receive packets and must also be registered with the LSL using *LSLRegisterStackRTag*.

See Also

- ◆ *LSLRegisterStackRTag*
- ◆ *LSLUnbindStack*

- ◆ BindProtocolToBoard in Appendix A, "Operating System Support Functions".

LSLControlStackFilter

Notifies all protocol stacks bound to the MLID that the MLID state has changed. Generally used by MLIDs.

Entry State

EAX

The number of the logical board filtering packets to protocol stacks which are bound and registered or chained with the logical board.

EBX

The control handler function number of the protocol stacks to be notified.

ECX

The filter mask for the protocol stacks to be notified. See *LSLGetStartChain* for the definitions of the bits in this mask.

ESI

Parameter 1 to pass to the control handler.

EDI

Parameter 2 to pass to the control handler.

Interrupts

Must be disabled.

Call

At process time only.

Return State

EAX

Completion Code.

Flags

Set according to EAX.

Interrupts

Unchanged.

Preserved

All other registers.

Completion Code (EAX)

00000000h	Successful
	All protocol stacks have been notified.
0FFFFFFF82h	BadParameter
	The specified board does not exist.

Remarks

This function calls the specified control function of some or all protocol stacks associated with the physical LAN adapter that the logical board is operating on. It updates protocol stacks that are operating on logical boards with the same name and instance as the logical board specified by board number.

One example of this functions use is when an MLID enters promiscuous mode, it can use this routine to call all protocol stacks who need to know the MLID is now in promiscuous mode.

See Also

- ◆ LSLModifyStackFilter
- ◆ CLSLControlStackFilter

LSLDeFragmentECB

Consolidates packet fragments.

Entry State

EAX

The amount of memory (in bytes) to skip before de-fragmenting the ECB.
If $EAX = -1$, the ECB header will not be copied.

EBX

Pointer to the source ECB.

EDX

Pointer to the destination ECB. If $EAX = -1$, *EDX* has a pointer to a buffer.

Interrupts

Can be in any state.

Call

At process time or interrupt time

Return State

EAX

Completion Code

Interrupts

Unchanged.

Preserved

No other registers.

Completion Code (EAX)

00000000h	Successful
	No error occurred.
0FFFFFF82h	BadParameter
	The ECB to be defragmented contains invalid fields describing the data contents.

Remarks

LSLDeFragmentECB consolidates packet fragments before the protocol stack processes the ECB. The resulting ECB is a copy of the original, but it consists of only one fragment. The protocol stack places the data from the original fragments at the specified offset from the *FragLen1* field in the destination ECB.

If EAX is -1, EDI has a pointer to a buffer where the buffer fragments should be placed.

LSLDeRegisterDefaultChain

Removes the chained default prescan stack from the LSL tables.

Entry State

ECX

The chain ID.

Interrupts

Can be in any state.

Call

At process time only.

Return State

EAX

Completion Code

Interrupts

Unchanged.

Destroyed

EBX, ECX, EDX.

Flags

Set according to EAX.

Completion Code (EAX)

0000000h	Successful
	The protocol stack was deregistered.
0FFFFFF85h	ItemNotPresent
	No default chain stack with this stack ID is registered for an MLID.

Remarks

LSLDeRegisterDefaultChain removes the specified chained default stack for a logical board from the LSL's internal default stack table.

After making this call, the protocol stack chain will not receive incoming packets from the MLID it was bound to.

See Also

- ◆ *LSLRegisterDefaultChain*

LSLDeRegisterMLID

De-registers the logical board.

Entry State

EBX

The board number.

Interrupts

Can be in any state.

Call

At process time only.

Return State

EAX

Completion Code

Interrupts

Disable, but could have been enabled.

Preserved

No other registers.

Completion Code (EAX)

00000000h	Successful
	The LSL successfully de-registered the MLID.
0FFFFFF82h	BadParameter
	The LSL did not have an MLID registered for the board number that was passed in EBX.

Remarks

The MLID calls *LSLDeRegisterMLID* to de-register one logical board from the LSL. The LSL then calls *Ctl_4MLIDDeRegistered* to inform all protocol stacks bound to that logical board that the logical board is no longer available.

IMPORTANT: If the LAN adapter is not having trouble sending out packets, the MLID should use *LSLUnbindThenDeRegisterMLID*.

LSLDeRegisterPreScanRxChain

Removes the chained receive prescan stack from the LSL tables.

Entry State

ECX

The chain ID.

Interrupts

Can be in any state.

Call

At process time only.

Return State

EAX

Completion Code

Interrupts

Disabled.

Destroyed

EBX, ECX, and EDX.

Completion Code (EAX)

00000000h	Successful
	The protocol stack was deregistered.
0FFFFFFF82h	ItemNotPresent
	No chained receive stack with this Stack ID is registered for the MLID.

Remarks

LSLDeRegisterPreScanRxChain removes the specified chained receive prescan stack for a logical board from the LSL's internal receive prescan stack table.

After making this call, the protocol stack chain will not receive any incoming packets from the MLID it was bound to.

See Also

- ◆ *LSLRegisterPreScanTxChain*

LSLDeRegisterPreScanTxChain

Removes the chained transmit prescan stack from the LSL tables.

Entry State

ECX

The chain ID.

Interrupts

Can be in any state.

Call

At process time only.

Return State

EAX

Completion Code

Interrupts

Disabled.

Destroyed

EBX, ECX, and EDX.

Completion Code (EAX)

00000000h	Successful
	The protocol stack was deregistered.
0FFFFFF82h	ItemNotPresent
	No chained transmit prescan stack with this Stack ID is registered for the MLID.

Remarks

LSLDeRegisterPreScanTxChain removes the specified chained transmit prescan stack for a logical board from the LSL's internal transmit prescan stack table.

After making this call, the protocol stack chain will not get any send packets for the MLID it was bound to.

See Also

- ◆ *LSLRegisterPreScanRxChain*

LSLDeRegisterStack

Removes a bound protocol stack from the LSL's tables. Generally used by protocol stacks.

Entry State

EAX

Has the protocol stack ID.

Interrupts

Are in any state.

Call

At process time only.

Return State

EAX

Completion Code

Interrupts

Are disabled.

Preserved

No other registers.

Flags

Are set according to EAX.

Completion Code (EAX)

0000000h

Successful

The bound protocol stack has been removed from the LSL tables.

0FFFFFF82h

BadParameter

Either the StackID is invalid, or no protocol stack is registered for that ID.

Remarks

LSLDeRegisterStack removes the specified bound protocol stack registered with the LSL from the LSL's internal protocol stack tables.

After making this call, a protocol stack will not receive incoming packets from the specified MLID unless the protocol stack is still bound to it as another type of stack.

The protocol stack must not call *LSLDeRegisterStack* during a critical section.

See Also

- ◆ LSLRegisterStackRTag

LSLDeRegisterStackSMPSafe

Used by SMP aware protocol stacks to inform the LSL of deregistration.

Entry State

[ESP+4]

Contains the stack ID.

Interrupts

Are in any state.

Call

At process time only.

Return State

EAX

Completion Code

Interrupts

Unchanged.

Preserved

EBX, EBP, ESI, EDI.

Completion Code (EAX)

00000000h	Successful
	The LSL has been notified of the de-registration.
0FFFFFF82h	BadParameter
	The stack ID is invalid.

Remarks

SMP aware protocol stacks call this function to inform the LSL that stack deregistration is pending. Protocol stacks must still call *LSLDeRegisterStack*, *LSLDeRegisterDefaultChain*, *LSLDeRegisterPreScanRxChain*, or *LSLDeRegisterPreScanTxChain* following this call.

See Also

- ◆ *LSLRegisterStackSMPSafe*
- ◆ *LSLDeRegisterStack*

LSLFastRcvEvent

Dispatches the ECB directly to the protocol stack.

Entry State

ESI

Points to the receive buffer to be processed.

Interrupts

Can be in any state.

Call

At process or interrupt time.

Return State

Interrupts

Are disabled, but could have been enabled.

Preserved

No other registers.

Completion Code (EAX)

None.

Remarks

LSLFastRcvEvent improves the performance of MLIDs that call *LSLServiceEvents* immediately after calling *LSLHoldRcvEvent*. *LSLFastRcvEvent* dispatches the ECB directly to the protocol stack.

Beware, however, that *LSLFastRcvEvent* can enable interrupts.

If your MLID board service routine must run with interrupts disabled, you can use either of the following options:

- ◆ The board service routine calls *LSLFastRcvEvent* as the last call before it issues a return.

- ◆ The board service routine handles being reentered at the point it calls *LSLFastRcvEvent*.

If the board service routine masks the PIC or the adapter interrupts instead of disabling them, it can use *LSLFastRcvEvent* at any point without being reentered.

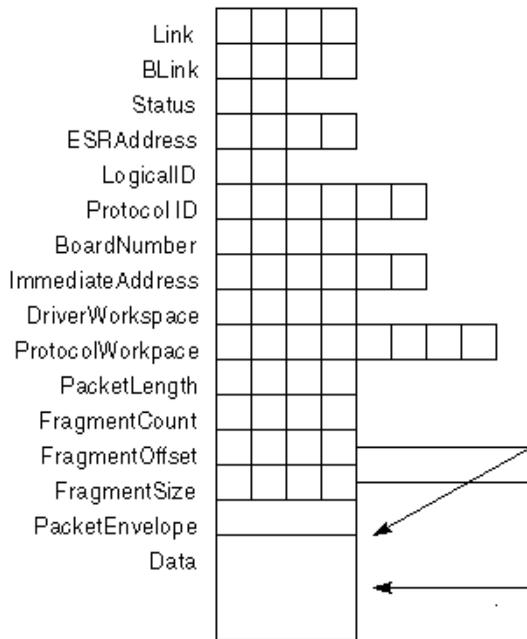
The board service routine must set the following fields of the ECB before calling *LSLFastRcvEvent*:

Table 25 **ECB Fields to Set Before Calling LSLFastRcvEvent**

Offset	Field Name	Description
04h	BLink	See Chapter 6, "Setting the ECB Blink Field".
10h	Protocol ID	The Protocol ID value obtained from the media header of the received packet.
16h	BoardNumber	Contains the board number of the MLID that received this packet.
1Ah	ImmediateAddress s	Contains the node address of the station that sent this packet. This address is in either canonical or noncanonical format, depending upon whether the MLID bit-swaps MSB format addresses. The protocol stack fills in this field on sends. All addresses passed down to the MLID are in canonical format if the MLID is configured for LSB.

Offset	Field Name	Description
29h	DriverWorkspace	<p>Before passing a completed receive ECB to the LSL, fill in the most significant byte of this field with the destination address type of the received packet.</p> <ul style="list-style-type: none"> ◆ 00h = Direct ◆ 01h = Multicast ◆ 03h = Broadcast ◆ 04h = Remote Unicast ◆ 08h = Remote Multicast ◆ 10h = No Source Route ◆ 20h = Error Packet ◆ 80h = Direct Unicast <p>Set the second byte of this field (offset 21h) to indicate whether the MAC header contains one or two 802.2 control bytes:</p> <ul style="list-style-type: none"> ◆ 0 = Not 802.2. ◆ 1 = 802.2 Ctrl0 only ◆ 2 = 802.2 Ctrl0 and Ctrl1
2Ch	PacketLength	Length of the data portion of the packet (RData).
30h	FragmentCount	The number of fragments in the packet. For receive packets, this value is always 1.
34h	FragmentOffset	For receive packets, this field contains a pointer to the only packet fragment. It points around any media headers contained in the PacketEnvelope portion of the packet (see Figure 12).
38h	FragmentSize	The length of the first packet fragment in bytes. On receives, the value in this field is the same as the value in the PacketLength field.
3Ch	PacketEnvelope	The beginning of the packet data. The media header is the first item of data in this section of the ECB. The media header varies in size with frame-type and topology.

Figure 12 Graphic Representation of the Receive Event Control Block



IMPORTANT: This process could call the MLID's packet transmission routine and could enable the interrupts.

See Also

- ◆ ODI Supplement: Frame Types and Protocol IDs for information about 802.2 Type II and Type I frames.

LSLFastSendComplete

Returns the ECB immediately to its owner. Generally used by MLIDs.

Entry State

ESI

Contains a pointer to the ECB that was sent.

Interrupts

Can be in any state.

Call

At process time or interrupt time.

Return State

Interrupts

Are disabled, but could have been enabled.

Preserved

No other registers.

Completion Code (EAX)

None.

Remarks

LSLFastSendComplete improves the performance of MLIDs that call *LSLServiceEvents* immediately after calling *LSLSendComplete*.
LSLFastSendComplete immediately returns the ECB to the protocol stack.

Be aware, however, that *LSLFastSendComplete* could enable interrupts. Consequently, the LSL could reenter the MLID's send routine before *LSLFastSendComplete* returns.

NOTE: The MLID does not have to call *LSLServiceEvents* on behalf of this ECB.

LSLFreePhysicalBoardID

Releases the unique physical board ID associated with a physical board.

Entry State

EAX

Pointer to the MLID resource tag.

EBX

The physical board ID to be released.

Interrupts

Can be in any state.

Call

At process time only.

Return State

EAX

Completion Code

Completion Code (EAX)

Successful	The unique physical board ID was released
BadParameter	An invalid parameter was passed to the function.

See Also

- ◆ LSLAllocatePhysicalBoardID

LSLGetBoundBoardInfo

Returns the Stack ID of the stack that is bound to the specified logical board or LAN adapter. Generally used by protocol stacks.

Entry State

EBX

Contains the board number.

EDI

Contains a pointer to buffer large enough to hold $(\text{MaxProtocols}+1)*4$ bytes.

Note: MaxProtocols is 16 in NetWare 3 or higher.

Interrupts

Can be in any state.

Call

At process time only.

EAX

Contains the completion code.

Return State

EDI

Contains a pointer to the buffer if the routine is successful.

- ♦ $[\text{Buffer} + 0]$ = the number of protocol stacks
- ♦ $[\text{Buffer} + 4]$ = the ID of first protocol stacks
- ♦ $[\text{Buffer} + 8]$ = the ID of the second protocol stack
- ♦ $[\text{Buffer} + 4n]$ = the ID of the nth protocol stack

Interrupts

Unchanged.

Preserved

All registers except ECX and ESI.

Completion Code (EAX)

00000000h	Successful
	The function executed successfully.
0FFFFFF82h	BadParameter
	The board number is invalid.
0FFFFFF85h	ItemNotPresent
	The board number is not registered.
0FFFFFF89h	OutOfResources
	The buffer is too small. [EDI+0] will contain the desired buffer length.

LSLGetLinkSupportStatistics

Gets a pointer to the LSL's statistics table.

Entry State

Interrupts

Can be in any state.

Call

At process time or interrupt time.

Return State

ESI

Pointer to the link support statistics vector.

Interrupts

Preserved.

Preserved

All other registers.

Completion Code (EAX)

None.

Remarks

LSLGetLinkSupportStatistics gets a pointer to the link support layer's statistics table.

See Also

- ♦ Chapter 10, "LSL Statistics Table"

LSLGetMaximumPacketSize

Determines the maximum size of the ECB. Generally used by protocol stacks and MLIDs.

Entry State

Interrupts

Can be in any state.

Call

At process time or interrupt time.

Return State

EAX

Contains the maximum physical packet size for which the LSL is configured.

Interrupts

Preserved and never changed.

Completion Code (EAX)

None.

Remarks

LSLGetMaximumPacketSize returns the largest physical packet size for which the LSL has been configured by entering ``Set Maximum Physical Receive Packet Size = x" in the STARTUP.NCF file.

Protocol stacks use *LSLGetMaximumPacketSize* to determine the maximum size of the event Control Block, not the size of send or receive packets that the MLID can handle.

LSLGetMLIDControlEntry

Returns the control entry point for the specified MLID. Generally used by protocol stacks.

Entry State

EBX

The logical board number.

Interrupts

Can be in any state.

Call

At process time only.

Return State

ESI

Contains a pointer to the control entry point.

Interrupts

Preserved and never changed.

Preserved

All other registers.

Completion Code (EAX)

00000000h	Successful
	ESI contains the MLID control entry point..
0FFFFFFF82h	BadParameter
	The board number in EAX is invalid..

Remarks

LSLGetMLIDControlEntry returns the MLID control entry point for the MLID that corresponds to the board number in EBX.

This command allows a protocol stack to communicate directly with the MLID and get information such as the addresses of the MLID configuration table and the MLID statistics table.

See Also

- ◆ Chapter 20, "MLID Control Routines"
- ◆ Chapter 14, "MLID Data Structures", for the format of the MLID configuration table.

LSLGetMultipleSizedRcvECBRTag

Gets multiple receive buffers. Generally used by protocol stacks and MLIDs.

Entry State

EAX

Pointer to a valid resource tag.

ECX

The number of ECBs.

ESI

The maximum size of packets including headers.

Interrupts

Can be in any state.

Call

At process time or interrupt time.

Return State

EAX

Completion Code.

ECX

The number of ECBs allocated.

ESI

POINTER to list of ECBs.

Flags

Set according to EAX.

Interrupts

Disabled.

Preserved

All other registers.

Completion Code (EAX)

00000000h	Successful
	No errors.
0FFFFFF89h	Out of Resources
	Either the packet size exceeded the maximum ECB size, or an ECB was not available.

Remarks

LSLGetMultipleSizedRcvECBRTag returns a linked list of ECBs, which are linked via the *LINK* field. The last ECB in the list contains a zero (0) in the *LINK* field.

LSLGetPhysicalAddressOfECB

Gets the physical memory address of a receive ECB.

Entry State

ESI

Logical pointer to the receive ECB.

Interrupts

Can be in any state.

Call

At process time or interrupt time.

Return State

EAX

Physical pointer to the receive ECB.

Interrupts

Unchanged.

Preserved

All other registers.

Completion Code (EAX)

None.

Remarks

MLIDs that require physical addresses for DMA use *LSLGetPhysicalAddressOfECB* to quickly convert the logical address of the receive ECB to a physical address. *LSLGetPhysicalAddressOfECB* must only be used to get the physical address of LSL receive ECBs that the MLID acquired using *LSLGetSizedECBRTag* or *LSLGetMultipleSizedRcvECBRTag*.

LSLGetPIDFromStackIDBoard

Returns a Protocol ID

Entry State

EAX

Contains the Stack ID.

EBX

Contains the board number.

EDI

Contains a pointer to a 6-byte area for the Protocol ID.

Interrupts

Can be in any state.

Call

At process time only.

Return State

EAX

Completion Code.

EDI

Pointer to the 6-byte area which now contains the protocol ID.

Interrupts

Preserved and not changed.

Preserved

No other registers.

Completion Code (EAX)

0000000h	Successful
	A match was found. EDI contains a pointer to the 6-byte buffer which now contains the protocol ID.
0FFFFFF82h	BadParameter
	The Stack ID or the board number in EAX is invalid.
0FFFFFF85h	ItemNotPresent
	No Protocol ID is associated with the parameters passed in.

Remarks

LSLGetPIDFromStackIDBoard returns a Protocol ID that corresponds to a combination of the Stack ID and board number. The protocol stack fills in the *ProtocolID* field of a send ECB with this information.

LSLGetProtocolControlEntry

Gets the entry point to a protocol stack. Generally used by protocol stacks.

Entry State

EAX

Contains an index number or the Stack ID. (See EBX).

EBX

Contents depend on EAX.

If EAX = FFFFFFFFh, EBX contains the board number to get the default protocol control handler for.

If EAX = FFFFFFFEh, EBX contains the board number to get the prescan protocol control handler for.

Otherwise, EAX contains the Stack ID of a bound protocol stack, and EBX contains the board number to get the protocol control handler for.

Interrupts

Can be in any state.

Call

At process time only.

Return State

EAX

Completion Code.

ESI

Contains a pointer to the control entry point for the specified protocol stack.

Interrupts

Preserved and never changed.

Preserved

All other registers.

Completion Code (EAX)

00000000h	Successful	A protocol Stack ID exists that corresponds to the value that was in EAX on entry. ESI contains the address of the protocol stack control entry point.
0FFFFFF82h	BadParameter	If EAX was equal to -1 or -2 on entry, the board number passed in EBX is not bound to any old- style unchained prescan stacks or default protocol stacks; otherwise, EAX contained an invalid Stack ID.
0FFFFFF85h	ItemNotPresent	The value that was in EAX on entry does not correspond to a valid Stack ID, but a Stack ID higher than the value that was in EAX on entry may exist.
0FFFFFF86h	NoMoreItems	If EAX was equal to -1 or -2 on entry, EBX contained an invalid board number.

Remarks

LSLGetProtocolControlEntry allows a protocol stack or an application to communicate directly with another protocol stack and get information from the LSL's list of protocol stacks.

NOTE: If you need to locate a control entry point for a default protocol stack or a prescan protocol stack, call *LSLGetStartChain*, because it supports chained protocol stacks.

See Also

- ◆ *LSLGetStartChain*

LSLGetSizedRcvECBRTag

Gets a receive buffer. Generally used by protocol stacks and MLIDs.

Entry State

EAX

Pointer to a valid resource tag.

ESI

Packet size, including headers.

Interrupts

Can be in any state.

Call

At process time or interrupt time.

Return State

EAX

Completion Code.

ESI

Pointer to the receive ECB.

Flags

Set according to EAX.

Interrupts

Disabled.

Preserved

All other registers.

Completion Code (EAX)

0000000h	Successful No errors.
0FFFFFF89h	Out of Resources Either the packet size exceeded the maximum ECB size, or an ECB was not available.

Remarks

MLIDs call *LSLGetSizedRcvECBRTag* to get buffers for received packets. The LSL returns an ECB and a buffer large enough to hold the received frame.

The length of all the protocol and hardware headers must be included in the length in the ESI register. For example, an Ethernet II frame uses `DataLength+14`.

If a receive ECB is not available, the MLID must discard the packet.

MLIDs that take advantage of bus-mastering DMA must preallocate ECBs. First these MLIDs must call *LSLGetMaximumPacketSize* and put either the returned value or the maximum packet length that the LAN adapter can receive-whichever is less-into ESI. Then, the MLID calls *LSLGetSizedRcvECBRTag*.

LSLGetStackIDFromName

Gets a Protocol Stack ID for any protocol stack. Generally used by protocol stacks.

Entry State

EDX

Pointer to a length-preceded, zero-terminated string containing the short name of the protocol stack.

Interrupts

Can be in any state.

Call

At process time only.

Return State

EAX

Completion Code.

EBX

Protocol Stack ID.

Interrupts

Preserved and never changed.

Preserved

All other registers.

Completion Code (EAX)

00000000h

Successful

A protocol stack corresponding to the name specified in EDX was found. The Protocol Stack ID for it is returned in EBX.

0FFFFFF82h	BadParameter
	The length of the name string was either zero or was greater than 15 characters.
0FFFFFF85h	ItemNotPresent
	The specified protocol stack name is not registered with the LSL.

Remarks

LSLGetStackIDFromName allows a protocol stack or application to get its own Stack ID or any other Stack ID.

With the Stack ID and the MLID board number, a protocol stack can call LSLGetPIDFromStackIDBoard to get the Protocol ID.

NOTE: The stack name match is case-sensitive.

LSLGetStartChain

Gets pointers to the starting offsets of the default chain, the prescan receive chain, and the prescan transmit chain.

Entry State

EBX

The board number of the LAN adapter or logical board bound to the desired chains.

Interrupts

Disabled.

Call

At process time only.

Return State

EAX

Completion Code.

EBX

The address of the default chain pointer.

EDI

The address of the transmit prescan chain pointer.

ESI

The address of the receive prescan chain pointer.

Interrupts

Disabled.

Preserved

ECX, EDX, EBP.

Flags

Set according to EAX.

Completion Code (EAX)

00000000h	Successful
	The routine successfully returned the chain pointers.
0FFFFFF82h	BadParameter
	The board number is invalid.

Remarks

LSLGetStartChain returns the pointers to the starting offsets of the default chain, the prescan receive chain, and prescan transmit chain. Each chain is a linked list of protocol stacks. If no protocol stacks are registered for a particular chain, a NULL pointer is returned.

```
ChainStructure      struc
ChainLink           dd      0      ;Link Field
ChainProcessHandler dd      -1     ;Stack Tx or Rx Handler
ChainCtlHandler     dd      2      ;Stack Control Entry Point
ChainRTag           dd      ?      ;Stack Resource Tag
ChainReceiveBufferRTag dd    ?      ;Receive ECB's Resource Tag
ChainLoadOrder      dd      ?      ;Stack Requested Chain Position
ChainID             dd      0      ;Stack ID Number (-1 indicates
                                an old style prescan or default
                                protocol stack.)
ChainMask           dd      0      ;Stack Packet Filter Mask
ChainStructure      ends
```

The ChainMask assigns the bit values as follows:

DEST_MULTICAST	equ	0001h	
DEST_BROADCAST	equ	0002h	
DEST_REMOTE_UNICAST	equ	0004h	
DEST_REMOTE_MULTICAST	equ	0008h	
DEST_NO_ROUTE	equ	0010h	Exclusive bit
DEST_ERROR_PACKET	equ	0020h	

Reserved	equ	0040h	
DEST_DIRECT	equ	0080h	
DEST_PROMISCUOUS	equ	FFFFh	All packets, including errors

LSLHoldRcvEvent

Directs the receive ECB to the LSL service event queue. Generally used by MLIDs.

Entry State

ESI

Pointer to the receive ECB.

Interrupts

Can be in any state.

Call

At process time or interrupt time.

Return State

ESI

Preserved.

EDI

Preserved.

EBP

Preserved.

Interrupts

Disabled.

Completion Code (EAX)

None.

Remarks

If the MLID is unable to use *LSLFastRcvEvent*, it can call *LSLHoldRcvEvent* to hand a receive ECB and a received packet to the LSL.

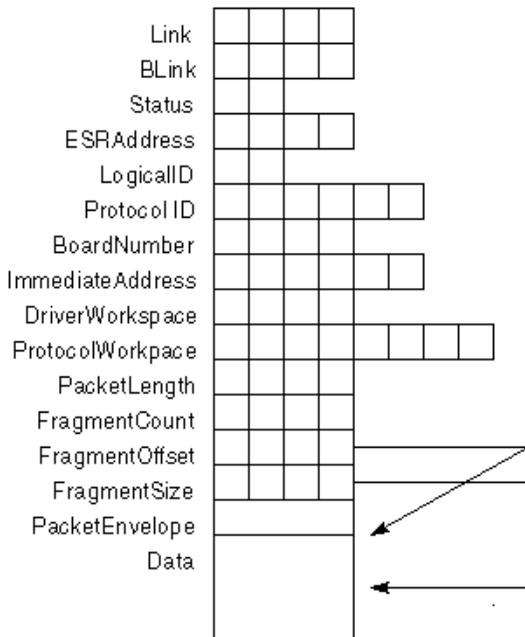
The MLID must set the following ECB fields prior to calling this routine:

Table 26 ECB Fields to Set Before Calling LSLHoldRcvEvent

Offset	Field Name	Description
04h	BLink	See Chapter 6, "Setting the ECB Blink Field".
10h	Protocol ID	The Protocol ID value obtained from the media header of the received packet.
16h	BoardNumber	Contains the board number of the MLID that received this packet.
1Ah	ImmediateAddresses	Contains the node address of the station that sent this packet. This address is in either canonical or noncanonical format, depending upon whether the MLID bit-swaps MSB format addresses. The protocol stack fills in this field on sends. All addresses passed down to the MLID are in canonical format if the MLID is configured for LSB.
29h	DriverWorkspace	<p>Before passing a completed receive ECB to the LSL, fill in the most significant byte of this field with the destination address type of the received packet.</p> <ul style="list-style-type: none"> ◆ 00h = Direct ◆ 01h = Multicast ◆ 03h = Broadcast ◆ 04h = Remote Unicast ◆ 08h = Remote Multicast ◆ 10h = No Source Route ◆ 20h = Error Packet ◆ 80h = Direct Unicast <p>Set the second byte of this field (offset 21h) to indicate whether the MAC header contains one or two 802.2 control bytes:</p> <ul style="list-style-type: none"> ◆ 0 = Not 802.2. ◆ 1 = 802.2 Ctrl0 only ◆ 2 = 802.2 Ctrl0 and Ctrl1
2Ch	PacketLength	Length of the data portion of the packet (RData).
30h	FragmentCount	The number of fragments in the packet. For receive packets, this value is always 1.

Offset	Field Name	Description
34h	FragmentOffset	For receive packets, this field contains a pointer to the only packet fragment. It points around any media headers contained in the PacketEnvelope portion of the packet (see Figure 13).
38h	FragmentSize	The length of the first packet fragment in bytes. On receives, the value in this field is the same as the value in the PacketLength field.
3Ch	PacketEnvelope	The beginning of the packet data. The media header is the first item of data in this section of the ECB. The media header varies in size with frame-type and topology.

Figure 13 Graphic Representation of the Receive Event Control Block



NOTE: The MLID cannot modify any ECB fields after making this call.

After the board service routine calls `LSLGetSizedRcvECBRTag` and reads the packet into the receive ECB, it calls `LSLHoldRcvEvent` to queue the receive ECB on the LSL's service events queue. Before leaving the board service

routine the MLID calls LSL Service Events to dispatch the ECBs on the hold queue.

LSLModifyStackFilter

Modifies the ChainMask field of the ChainStructure. Generally used by protocol stacks.

Entry State

EAX

The protocol Stack ID or the chain Stack ID.

EBX

The board number.

ECX

The new ChainMask filter bits or zero to query.

Return State

EAX

Completion Code.

ECX

The current ChainMask filter bits.

Interrupts

Unchanged.

Preserved

All register except ESI.

Completion Code (EAX)

0000000h

Successful

The ChainMask filter has been set to the new value.

0FFFFFF82h

BadParameter

The board number or the stack/chain ID was invalid.

Remarks

LSLModifyStackFilter lets the protocol stack set the *ChainMask* field of the *ChainStructure*. If the protocol stack is bound, this routine allows the stack to set the filter mask. The protocol stack uses the filter mask to notify the LSL of the type of packets the stack wants to receive.

See Also

- ◆ *LSLGetStartChain* for the definition of the Chain Mask bits.

LSLRegisterDefaultChain

Binds a chained default protocol stack to the specified MLID. Generally used by protocol stacks.

Entry State

EAX

The address of the default stack chain's resource tag.

EBX

The board number.

ECX

The address of the protocol stack's default receive handler.

EDX

The address of the control handler for the default protocol handler.

ESI

The address of the receive ECB's resource tag.

EBP

The protocol stack's required position in the chain. See [Table 27](#).

Interrupts

Can be in any state.

Call

At process time only.

Return State

EAX

Completion Code.

EBX

The stack chain ID.

Interrupts

Disabled.

Preserved

ECX, EDX, ESI, EDI, and EBP.

Flags

Set according to EAX.

Completion Code (EAX)

00000000h	Successful
	No errors occurred.
0FFFFFF82h	BadParameter
	The board number, the resource tag, or the requested position in the stack chain was invalid.
0FFFFFF83h	DuplicateEntry
	The requested chain position is either the first or last position (EBP equals 0 or 4) and is already occupied.
0FFFFFF89h	OutOfResources
	The LSL was unable to allocate the node structure for the chain.

Remarks

LSLRegisterDefaultChain binds the chained default protocol stack to the MLID whose board number is specified in EBX. The default protocol stack chain receives all packets that are not requested by either a chained receive prescan protocol stack or a bound protocol stack.

Protocol stacks use this routine to put themselves in the proper position in the chain.

To acquire the stack chain resource tag, the protocol stack calls the operating system support routine, *AllocateResourceTag*, with the *LSLDefaultStackSignature* 'DLSL'.

The stack chain ID returned in EBX is a unique identifier based on a logical board/protocol stack chain combination. This chain ID is not the same as the Stack ID that is issued to a single, normally registered protocol stack that can be bound to a number of different boards.

The protocol stack's required position in the chain is defined in the following table.

0	The stack must start the chain.
1	The stack must be loaded at the next available position from the front of the chain.
2	The stack's position in the chain is dependent on the order in which the stacks are loaded. The stack will take the next available position when it loads.
3	The stack must be loaded at the next available position from the end of the chain.
4	The stack must end the chain.

WARNING: If the protocol stack requests a chain position of 0 or 4, and another protocol stack is already registered for that board number at that position, the registration routine will abort.

See Also

- ◆ [LSLDeRegisterDefaultChain](#)

LSLRegisterMLIDRTag

Registers a logical board. Generally used by MLIDs.

Entry State

EAX

Pointer to the MLID send routine.

EBX

The MLID resource tag.

ECX

Pointer to the MLID configuration table.

EDX

Loadable Module Handle (passed to your driver during initialization. See Figure 3-1)

ESI

Pointer to the MLID control handler routine.

Interrupts

Can be in any state.

Call

At process time only.

Return State

EAX

Completion Code.

EBX

The assigned board number.

ECX

The maximum buffer size of receive ECBs.

Flags

Set according to EAX.

Interrupts

Preserved.

Preserved

No other registers.

Completion Code (EAX)

00000000h	Successful No errors occurred.
0FFFFFF89h	OutOfResources There was not enough memory to register MLID.
0FFFFFF82h	BadParameter The resource tag was invalid.

Remarks

The MLID initialization procedure calls *LSLRegisterMLIDRTag* to register a logical board.

By making this call, the MLID initialization procedure gives the LSL a send procedure, a control procedure, and a configuration table for the logical board.

If the value returned in ECX is smaller than *MLIDMaximumSize*, the MLID must adjust the three packet size fields: *MLIDMaximumSize*, *MLIDMaxRecvSize*, *MLIDRecvSize* according to [Table 39](#), Maximum Packet Sizes.

LSLRegisterPreScanRxChain

Binds a chained receive prescan protocol stack to the specified MLID. Generally used by protocol stacks.

Entry State

EAX

Address of the receive prescan stack chain resource tag.

EBX

The board number.

ECX

Address of the protocol stack prescan receive handler.

EDX

Address of the protocol stack control handler.

EBP

The protocol stack's required position in the chain. See [Table 27](#).

ESI

Address of the receive ECB resource tag.

Interrupts

Can be in any state.

Call

At process time only.

Return State

EAX

Completion Code.

EBX

Stack Chain ID.

Interrupts

Disabled.

Preserved

ECX, EDX, ESI, EDI, and EBP.

Flags

Set according to EAX.

Completion Code (EAX)

00000000h	Successful No errors occurred.
0FFFFFF82h	BadParameter The board number, the resource tag, or the requested offset in the stack chain was invalid.
0FFFFFF83h	DuplicateEntry The requested chain position is either the first or last position (EBP equals 0 or 4) and is already occupied.
0FFFFFF89h	OutOfResources The LSL was unable to allocate the node structure for the chain.

Remarks

LSLRegisterPreScanRxChain binds the chained receive prescan protocol stack to the MLID whose board number is specified in EBX. Protocol stacks use this routine to put themselves in the proper position in the chain.

To acquire the stack chain resource tag, the protocol stack calls the operating system support routine, *AllocateResourceTag* with the *LSLPreScanStackSignature* 'PLSL'.

The stack chain ID, returned in EBX, is a unique identifier based on a logical board protocol stack chain combination. This chain ID is not the same as the Stack ID that is issued to a single, normally registered protocol stack that can be bound to a number of different boards.

The protocol stack's required position in the chain is defined in the following table.

Table 27 Chain Position Values

Parameter Value	Position in Chain
0	The stack must start the chain.
1	The stack must be loaded at the next available position from the front of the chain.
2	The stack's position in the chain is dependent on the order in which the stacks are loaded. The stack will take the next available position when it loads.
3	The stack must be loaded at the next available position from the end of the chain.
4	The stack must end the chain.

WARNING: If the protocol stack requests a chain position of 0 or 4, and another protocol stack is already registered for that board number at that position, the registration routine will abort.

See Also

- ◆ LSLDeRegisterPreScanRxChain

LSLRegisterPreScanTxChain

Binds a chained transmit prescan protocol stack to the specified MLID. Generally used by protocol stacks.

Entry State

EAX

Address of the prescan stack chain resource tag.

EBX

The board number.

ECX

Address of the protocol stack prescan transmit handler.

EDX

Address of the protocol stack control handler.

EBP

The protocol stack's required position in the chain. See [Table 28](#).

ESI

Address of the transmit ECB resource tag.

Interrupts

Can be in any state.

Call

At process time only.

Return State

EAX

Completion Code.

EBX

Stack Chain ID.

Interrupts

Disabled.

Preserved

ECX, EDX, ESI, EDI, and EBP.

Flags

Set according to EAX.

Completion Code (EAX)

00000000h	Successful No errors occurred.
0FFFFFF82h	BadParameter The board number, the resource tag, or the requested offset in the stack chain was invalid.
0FFFFFF83h	DuplicateEntry The requested chain position is either the first or last position (EBP equals 0 or 4) and is already occupied.
0FFFFFF89h	OutOfResources The LSL was unable to allocate the node structure for the chain.

Remarks

LSLRegisterPreScanTxChain binds the chained receive prescan protocol stack to the MLID whose board number is specified in EBX. Protocol stacks use this routine to put themselves in the proper position in the chain.

To acquire the stack chain resource tag, the protocol stack calls the operating system support routine, *AllocateResourceTag* with the *LSLTxPreScanStackSignature* 'TLSL'.

The stack chain ID, returned in EBX, is a unique identifier based on a logical board protocol stack chain combination. This chain ID is not the same as the Stack ID that is issued to a single, normally registered protocol stack that can be bound to a number of different boards.

The protocol stack's required position in the chain is defined in the following table:

Table 28 Chain Position Values

Parameter Value	Position in Chain
0	The stack must start the chain.
1	The stack must be loaded at the next available position from the front of the chain.
2	The stack's position in the chain is dependent on the order in which the stacks are loaded. The stack will take the next available position when it loads.
3	The stack must be loaded at the next available position from the end of the chain.
4	The stack must end the chain.

WARNING: If the protocol stack requests a chain position of 0 or 4, and another protocol stack is already registered for that board number at that position, the registration routine will abort.

NOTE: If a prescan transmit protocol stack must modify the data of an ECB that has been passed to it (for example, as in a compression stack), the stack should treat the ECB as read-only. The protocol stack copies the ECB and processes the copy. The prescan stack keeps a copy of the original ECB's Event Service Routine (ESR). When the prescan stack's ESR is called, it takes one of two actions:

- ◆ It calls the original stack's ESR with a pointer to the original ECB in ESI.
- ◆ It puts the original ECB in the LSL Event Queue and calls LSLServiceEvents.

See Also

- ◆ LSLDeRegisterPreScanTxChain

LSLRegisterStackRTag

Registers a bound protocol stack with the LSL. Generally used by protocol stacks.

Entry State

EAX

Address of the protocol stack resource tag.

EBX

Address of the protocol stack receive handler.

ECX

Address of the control entry point for the protocol stack.

EDX

Pointer to the protocol name, which is a length-preceded, zero-terminated string.

EBP

Address of the receive ECB resource tag.

Interrupts

Can be in any state.

Call

At process time only.

Return State

EAX

Completion Code.

EBX

Protocol stack number.

Interrupts

Preserved.

Preserved

No other registers.

Completion Code (EAX)

00000000h	Successful
	The protocol stack has registered successfully.
0FFFFFF82h	BadParameter
	Either the resource tag was invalid, or the length of the protocol name equaled 0 or was greater than 15.
0FFFFFF83h	DuplicateEntry
	This protocol stack is already registered.
0FFFFFF89h	OutOfResources
	The LSL was unable to allocate the node structure for the chain.

Remarks

When the protocol stack is bound at the command line, the bind IOCTL is called.

Registering the protocol stack does not mean that it can receive packets. To enable packet reception through the LSL, the protocol stack must also call *LSLBindStack* after the network administrator has requested to bind the protocol stack to an MLID. The protocol stack only makes these calls once for each MLID that the stack wants to receive packets from.

A protocol stack can transmit packets and communicate with MLIDs even if it is not registered with the LSL or bound to an MLID. If a protocol stack does not bind to an MLID, it must call either *LSLRegisterDefaultChain* or *LSLRegisterPreScanRXChain* to receive packets.

LSLRegisterStackSMPSafe

Informs the LSL that the stack is SMP aware. Generally used by protocol stacks.

Entry State

ESP + 4

The stack ID.

Interrupts

Can be in any state.

Call

At process time only.

Return State

Interrupts

Unchanged.

Preserved

EBX, EBP, ESI, EDI

Completion Code (EAX)

00000000h	Successful
	Routine completed successfully.
0FFFFFF82h	BadParameter
	The Stack ID is invalid.

Remarks

Protocol stacks call this function to inform the LSL that the stack is written to be SMP aware. *LSLRegisterStackSMPSafe* is intended for use by Bound

Protocol Stacks only. The stack must have registered using *LSLRegisterStackRTag* previous to making this call.

See Also

- ◆ `LSLDeRegisterStackSMPSafe`
- ◆ `LSLRegisterStackRTag`

LSLRemoveMutexFromInstance

Removes a mutex lock for a specified instance of a logical board. Generally used by MLIDs.

Entry State

EBX

The logical board number, the mutex was assigned to.

ESI

The physical board ID.

Interrupts

Can be in any state.

Call

At process time or interrupt time.

Return State

Interrupts

Unchanged.

Preserved

All other registers.

Completion Code (EAX)

None.

Remarks

SMP aware MLIDs call this function as part of their clean up sequence during unload. Typically, this function is called soon after *LSLDeRegisterMLIDRTag*.

See Also

- ◆ LSLAllocatePhysicalBoardID
- ◆ LSLAssignMutexToInstance
- ◆ LSLRemovePhysicalMutex

LSLRemovePhysicalMutex

Removes all logical board mutexes associated with the given physical adapter. Generally used by MLIDs.

Entry State

EBX

The physical board ID.

Interrupts

Can be in any state.

Call

At process time or interrupt time.

Return State

Interrupts

Unchanged.

Preserved

All other registers.

Completion Code (EAX)

None.

Remarks

SMP aware MLIDs can call this function once instead of calling *LSLRemoveMutexFromInstance* multiple times while the MLID is unloading.

See Also

- ◆ *LSLAllocatePhysicalBoardID*
- ◆ *LSLRemoveMutexFromInstance*

- ◆ LSLAssignMutexToinstance

LSLRemovePollingProcedure

Deregisters a polling procedure in the SMP environment. Generally used by polled MLIDs.

Entry State

EBX

The physical board ID.

Interrupts

Can be in any state.

Call

At process time only.

Return State

Interrupts

Unchanged.

Preserved

All other registers.

Completion Code (EAX)

None.

Remarks

SMP aware polled MLIDs call this function to deregister their polling procedures with SMP.

NOTE: This call is used in place of *RemovePollingProcedure*, which is used for non-SMP aware MLIDs.

See Also

- ◆ LSLAllocatePhysicalBoardID

- ◆ LSLAddpollingProcedure
- ◆ RemovePollingProcedureRTag in Appendix A, "Operating System Support Routines"

LSLRemoveTimerProcedure

Deregisters a timer based callback procedure in the SMP environment. Generally used by MLIDs.

Entry State

EBX

The physical board ID.

Interrupts

Can be in any state.

Call

At process time or interrupt time.

Return State

Interrupts

Unchanged.

Preserved

All other registers.

Completion Code (EAX)

None.

Remarks

SMP-aware MLIDs call this function to deregister a timer driven call back procedure with SMP (*CancelInterruptTimeCallBack* in the SMP environment).

See Also

- ◆ LSLAllocatePhysicalBoardID
- ◆ LSLAddTimerProcedure

- ◆ `CancelInterruptTimeCallBack` in Appendix A, "Operating System Support Routines"

LSLReSubmitDefaultECB

Allows another default stack in the default stack chain to process the ECB.

Entry State

ESI

Address of the receive ECB.

EDI

The logical ID of the protocol stack (Chain ID).

Interrupts

Disabled.

Call

At process time or interrupt time.

Return State

Interrupts

Disabled.

Preserved

No other registers.

Completion Code (EAX)

None.

Remarks

LSLReSubmitDefaultECB allows chained default protocol stacks to pass ECBs to the LSL for further processing.

A protocol stack normally calls this routine at process time after the stack has queued the ECB for further processing. After the protocol stack has processed

the ECB, it passes the ECB to other stacks in the chain, For example, default stacks that perform data compression use this call.

LSLReSubmitPreScanRxECB

Allows the next stack in the receive prescan stack chain to process the ECB. Generally used by protocol stacks.

Entry State

ESI

Address of the receive ECB.

EDI

The logical ID of the protocol stack (Chain ID).

Interrupts

Can be in any state.

Call

At process time or interrupt time.

Return State

Interrupts

Disabled.

Preserved

No other registers.

Completion Code (EAX)

None.

Remarks

LSLReSubmitPreScanRxECB allows chained prescan receive protocol stacks to pass ECBs to the LSL for further processing.

A protocol stack normally calls this routine at process time after the stack has queued the ECB for further processing. After, the protocol stack has processed the ECB, it passes the ECB to other stacks in the chain.

LSLReSubmitPreScanTxECB

Allows the next stack in the transmit prescan stack chain to process the ECB. Protocol stacks.

Entry State

ESI

Address of the transmit ECB.

EDI

The logical ID of the protocol stack (Chain ID).

Interrupts

Can be in any state.

Call

At process time or interrupt time.

Return State

Interrupts

Disabled.

Preserved

No other registers.

Completion Code (EAX)

None.

Remarks

LSLReSubmitPreScanTxECB allows chained prescan transmit protocol stacks to pass ECBs to the LSL for further processing.

A protocol stack normally calls this routine at process time after the stack has queued the ECB for further processing. After, the protocol stack has processed the ECB, it passes the ECB to other stacks in the chain.

LSLReturnRcvECB

Returns a preallocated receive ECB. Generally used by protocol stacks and MLIDs.

Entry State

ESI

Pointer to the receive buffer

Interrupts

Can be in any state.

Call

At process time or interrupt time.

Return State

EAX

Destroyed.

Interrupts

Disabled.

Completion Code (EAX)

None.

Remarks

Protocol stacks call *LSLReturnRcvECB* to return previously allocated receive ECB buffers to the LSL.

MLIDs call *LSLReturnRcvECB* to return unneeded receive ECBs to the LSL.

See Also

- ◆ LSLGetRcvECBRTag

LSLSendComplete

Queues a send ECB in the LSL service events queue. Generally used by MLIDs.

Entry State

ESI

Pointer to the ECB that was sent.

Interrupts

Can be in any state.

Call

At process time or interrupt time.

Return State

EAX

Destroyed.

Interrupts

Disabled.

Completion Code (EAX)

None.

Remarks

If the MLID is unable to use *LSLFastSendComplete*, it can call *LSLSendComplete* to return a send ECB to the LSL after the MLID has finished processing it.

This call does not return the ECB to its owner. It simply queues the ECB on the LSL service events queue and returns.

The MLID should call *LSLServiceEvents* at the end of the board interrupt service routine and/or the MLID's packet transmission procedure.

LSLSendPacket

Sends a packet to the MLID. Generally used by protocol stacks.

Entry State

ESI

Pointer to a send ECB.

Interrupts

Can be in any state.

Call

At process time or interrupt time.

Return State

EAX

Completion Code.

Interrupts

Disabled, but could have been enabled.

Preserved

No registers.

Completion Code (EAX)

00000000h	Successful
	The ECB was successfully passed to the MLID.
0FFFFFFF85h	ItemNotPresent
	The board number in the ECB was invalid.

Remarks

LSLSendPacket sends a packet to one of the registered MLIDs.

The ESR field of the ECB must have the address of an Event Service Routine (ESR) to call when the send is complete. Until the ESR is called, the ECB and all its data areas belong to the LSL and must not be modified.

NOTE: The LSL can call the ESR before *LSLSendPacket* returns.

The ESR is called with ESI and the first parameter on the stack containing a pointer to the ECB that was sent. This call is made at either interrupt time or process time with interrupts disabled. The ESR can enable interrupts, but if it does so, it must guard against reentry. The ESR must complete quickly.

Raw Sends

If the ECB is sent in raw mode, the fragment list contains the complete packet, including the link-level envelope. However, the link-level envelope must be entirely contained within the first fragment. In other words, the envelope cannot be split between the first and second fragments.

All media/frame type anomalies must be accounted for in the media/frame type header. Raw packets are sent with the StackID field of the ECB set equal to a value between FFFFh and FFF8h. These values represent priority raw sends. The protocol stack must check the board's configuration table *MLIDModeFlags* field to see if the board supports raw sends (See Chapter 14, "MLID Data Structures" for the MLID Configuration Table format.)

Priority Sends

Priority sends are sent with the *stackID* field of the ECB set to a value between 0FFF7h (lowest priority) and 0FFF0h (highest priority).

The Transmit ECB

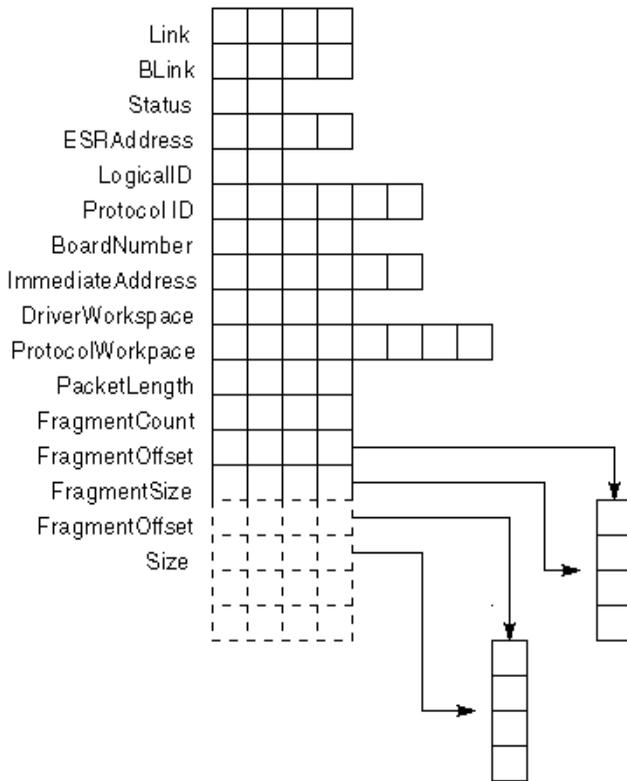
The following fields in the ECB must be set before calling *LSLSendPacket*:

Table 29 ECB Fields to Set Before Calling *LSLSendPacket*

Offset	Field Name	Description
0Ah	ESRAddress	The address of a routine that is called when the ECB is released (after the packet has been transmitted). A pointer to the ECB is passed to this routine in ESI. This pointer is also the first parameter on the stack for ESRs written in C. This field is a near pointer.

Offset	Field Name	Description
0Eh	LogicalID	The Stack ID of the protocol stack sending the packet. See the "Raw Send" and "Priority Sends" sections above for more detail.
10h	ProtocolID	The Protocol ID (returned by LSLGetPIDFromStackIDBoard) that the MLID is to use when encapsulating the data. This field is ignored if a raw packet is sent. See the section 7.6.1.2 - The ECB ProtocolID Field and Ethernet 802.2 for more detail.
16h	BoardNumber	The board number of the MLID sending this packet.
1Ah	ImmediateAddress	The node address on the physical network that the packet is destined for. If the packet is raw, this field is undefined. The address FFFFFFFFh always indicates a broadcast packet. (A broadcast packet is received by all nodes on the physical network.)
2Ch	PacketLength	The total length of all fragment buffers.
30h	FragmentCount	The number of fragments in the packet to be sent. Descriptor data structures follow this field. This field must contain a value between 1 and 16, inclusive. The protocol stack can specify a maximum of 16 fragment descriptors. The MLID combines these fragments together to form one contiguous packet.
34h	FragmentOffset	On sends, this field describes the location of a contiguous section of RAM memory (32-bit offset).
38h	FragmentLength	The length in bytes of the first packet fragment. On sends, this value can be 0. On sends, the ECB may contain the following additional fields as needed:
3Ch	FragmentOffsetX	Additional fragment descriptor when the FragmentCount is greater than 1. The X stands for the additional fragment number (2 through 16).
40h	FragmentLengthX	Additional fragment descriptor when the FragmentCount is greater than 1. The X stands for the addition fragment number (2 through 16) . The FragmentOffsetX and FragmentLengthX fields may repeat up to 16 times.

Figure 14 Graphic Representation of the Transmit Event Control Block



LSLServiceEvents

Removes packets from the queue. Generally used by MLIDs.

Entry State

Interrupts

Can be in any state.

Call

At process time or interrupt time.

Return State

Interrupts

Disabled, but could have been enabled.

Preserved

None. All registers are destroyed.

Completion Code (EAX)

None.

Remarks

If the MLID has not used *LSLFastRcvEvent* or *LSLFastSendComplete*, it must call *LSLServiceEvents* to remove packets queued by *LSLHoldRcvEvent* or *LSLSendComplete*.

The board service routine calls *LSLServiceEvents* after processing all transmissions and receptions. This is the last thing the board service routine does before returning.

The MLID must complete all hardware processing, and the Board Service Routine must be ready for a new interrupt before the MLID can make this call.

The *LSLServiceEvents* routine routes all received packets to the correct protocol stack, using the Stack ID that the MLID put in the *LogicalID* field of the ECB.

This routine also returns the sent packets to their owners.

IMPORTANT: If the MLID uses *LSLFastSendComplete* and *LSLHoldRcvEvent* for completing events, it does not need to call *LSLServiceEvents*.

LSLSMPGetSendQ

Gets the entire send queue for a logical board from the LSL. Generally used by MLIDs.

Entry State

EBX

The logical board number.

Interrupts

Can be in any state.

Call

At process time or interrupt time.

Return State

EAX

Completion Code.

ESI

Pointer to the first ECB in the chain of ECBs to be sent.

Interrupts

Disabled.

Preserved

All other registers except ECX.

Completion Code (EAX)

00000000h	Successful
	The routine completed successfully.

0FFFFFF86h

NoMoreItems

The send queue is empty.

LSLSMPReaderLock

Sets a data reader lock. Generally used by SMP aware protocol stacks.

Entry State

Interrupts

Must be disabled.

Call

At process time only.

Return State

Interrupts

Disabled.

Preserved

EBX, EBP, ESI, EDI

Completion Code (EAX)

None.

Remarks

When running in SMP mode, the LSL allocates a system wide protocol stack read-write lock for use in protecting a stack's data. SMP aware stacks can use this routine to set the reader lock.

NOTE: SMP aware protocol stack receive handlers are called with a reader lock already set, which is sufficient to protect the stack's data if it is only read. In which case, it is unnecessary for the stack to make this call. If the stack needs to write data while processing a receive packet, it must upgrade the reader lock to a write lock using *LSLSMPReaderToWriterLock*. Then, after making the data changes, the stack must downgrade the lock back to a reader lock using *LSLSMPWriterToReaderLock*.

See Also

- ◆ LSLSMPReaderUnlock
- ◆ LSLSMPWriterLock
- ◆ LSLSMPWriterUnlock
- ◆ LSLSMPReaderToWriterLock
- ◆ LSLSMPWriterToReaderLock

LSSLMPReaderToWriterLock

Converts a reader lock to a writer lock. Generally used by SMP aware protocol stacks.

Entry State

Interrupts

Must be disabled.

Call

At process time only.

Return State

EAX

Completion Code.

Interrupts

Disabled.

Preserved

EBX, EBP, ESI, EDI

Completion Code (EAX)

0000000h	Successful
	Write lock obtained successfully.
Non-zero	Failure
	The write lock had to wait on another writer (such as when the pre-read data is invalid).

Remarks

SMP aware protocol stacks can use this function to convert a previously obtained reader lock to a writer lock.

This call is typically used inside the stack's receive handler when the stack needs to update its data. Since the receive handler is called with the reader lock already set, the stack uses this call to convert the reader lock to a writer lock.

After the stack performs its data modification, it must convert the lock back to a reader lock using *LSLSMPWriterToReaderLock* before it returns. This method uses less overhead than calling *LSLSMPReaderUnLock*, *LSLSMPWriterLock*, *LSLSMPWriterUnlock*, and *LSLSMPReaderLock* in sequence.

NOTE: This call can spin; therefore, any memory controlled by this lock may have been changed if EAX is a non-zero value on return. If this is the case, the stack must re-read and check all of the protected data.

See Also

- ◆ *LSLSMPWriterToReaderLock*
- ◆ *LSLSMPReaderUnLock*
- ◆ *LSLSMPWriterLock*
- ◆ *LSLSMPWriterUnlock*
- ◆ *LSLSMPReaderLock*

LSLSMPReaderUnLock

Releases a data reader lock. Generally used by SMP aware protocol stacks.

Entry State

Interrupts

Must be disabled.

Call

At process time only.

Return State

Interrupts

Disabled.

Preserved

EBX, EBP, ESI, EDI

Completion Code (EAX)

None.

Remarks

When running in SMP mode, the LSL allocates a system wide protocol stack read-write lock for use in protecting a stack's data. SMP aware stacks can use this routine to release the reader lock obtained previously with *LSLSMPReaderLock*.

NOTE: SMP aware protocol stack receive handlers are called with a reader lock already set, which is sufficient to protect the stack's data if it is only read. In which case, it is unnecessary for the stack to make this call. If the stack needs to write data while processing a receive packet, it must upgrade the reader lock to a write lock using *LSLSMPReaderToWriterLock*. Then, after making the data changes, the stack must downgrade the lock back to a reader lock using *LSLSMPWriterToReaderLock*.

See Also

- ◆ LSLSMPWriterToReaderLock
- ◆ LSLSMPReaderToWriterLock
- ◆ LSLSMPWriterLock
- ◆ LSLSMPWriterUnlock
- ◆ LSLSMPReaderLock

LSLSMPWriterLock

Sets a data writer lock. Generally used by SMP aware protocol stacks.

Entry State

Interrupts

Must be disabled.

Call

At process time only.

Return State

Interrupts

Disabled.

Preserved

EBX, EBP, ESI, EDI

Completion Code (EAX)

None.

Remarks

When running in SMP mode, the LSL allocates a system wide protocol stack read-write lock for use in protecting a stack's data. SMP aware stacks can use this routine to set the writer lock.

NOTE: SMP aware protocol stack receive handlers are called with a reader lock already set, which is sufficient to protect the stack's data if it is only read. In which case, it is unnecessary for the stack to make this call. If the stack needs to write data while processing a receive packet, it must upgrade the reader lock to a write lock using *LSLSMPReaderToWriterLock*. Then, after making the data changes, the stack must downgrade the lock back to a reader lock using *LSLSMPWriterToReaderLock*.

See Also

- ◆ LSLSMPWriterToReaderLock
- ◆ LSLSMPReaderToWriterLock
- ◆ LSLSMPReaderUnLock
- ◆ LSLSMPWriterUnlock
- ◆ LSLSMPReaderLock

LSSLMPWritertoReaderLock

Converts a writer lock to a reader lock. Generally used by SMP aware protocol stacks.

Entry State

Interrupts

Must be disabled.

Call

At process time only.

Return State

Interrupts

Disabled.

Preserved

EBX, EBP, ESI, EDI

Completion Code (EAX)

None.

Remarks

SMP aware protocol stacks can use this function to convert a previously obtained writer lock to a reader lock.

This call is typically used inside the stack's receive handler when the stack needs to update its data. Since the receive handler is called with the writer lock already set, the stack uses this call to convert the writer lock to a reader lock.

After the stack performs its data modification, it must convert the lock back to a writer lock using *LSSLMPReaderToWriterLock* before it returns. This method uses less overhead than calling *LSSLMPWriterUnlock*, *LSSLMPReaderLock*, *LSSLMPReaderUnlock*, and *LSSLMPWriterLock* in sequence.

See Also

- ◆ LSLSMPReaderToWriterLock
- ◆ LSLSMPReaderUnLock
- ◆ LSLSMPWriterLock
- ◆ LSLSMPWriterUnlock
- ◆ LSLSMPReaderLock

LSLSMPWriterUnLock

Releases a data writer lock. Generally used by SMP aware protocol stacks.

Entry State

Interrupts

Must be disabled.

Call

At process time only.

Return State

Interrupts

Disabled.

Preserved

EBX, EBP, ESI, EDI

Completion Code (EAX)

None.

Remarks

When running in SMP mode, the LSL allocates a system wide protocol stack read-write lock for use in protecting a stack's data. SMP aware stacks can use this routine to release the reader lock obtained previously with *LSLSMPReaderLock*.

NOTE: SMP aware protocol stack receive handlers are called with a reader lock already set, which is sufficient to protect the stack's data if it is only read. In which case, it is unnecessary for the stack to make this call. If the stack needs to write data while processing a receive packet, it must upgrade the reader lock to a write lock using *LSLSMPReaderToWriterLock*. Then, after making the data changes, the stack must downgrade the lock back to a reader lock using *LSLSMPWriterToReaderLock*.

See Also

- ◆ LSLSMPWriterToReaderLock
- ◆ LSLSMPReaderToWriterLock
- ◆ LSLSMPWriterLock
- ◆ LSLSMPReaderUnlock
- ◆ LSLSMPReaderLock

LSLUnbindStack

Removes a bound protocol stack from the LSL configuration table.
Generally used by protocol stacks.

Entry State

EAX

The Stack ID

EBX

The board number.

Interrupts

Can be in any state.

Call

At process time only.

Return State

EAX

Completion Code.

Interrupts

Preserved.

Preserved

No registers.

Completion Code (EAX)

00000000h	Successful
	No error occurred.

0FFFFFF82h	BadParameter
	The board number or the protocol Stack ID is invalid.
0FFFFFF85h	ItemNotPresent
	The specified binding does not exist.

Remarks

LSLUnbindStack unbinds a bound protocol stack from an MLID.

After this call is completed, the protocol stack can no longer receive packets from the MLID.

However, the protocol stack can still receive packets from the board if the protocol stack was also registered with the board as the default protocol stack or the receive prescan protocol stack.

LSLUnBindThenDeRegisterMLID

Unbinds an MLID from its bound protocol stacks, then deregisters the board also.

Entry State

EBX

The board number.

Interrupts

Can be in any state.

Call

At process time only.

NOTE: The LAN adapter must not be in a critical section.

Return State

Interrupts

Disabled.

Preserved

No other registers.

Completion Code (EAX)

None.

Remarks

MLID remove procedures should call *LSLUnBindThenDeregisterMLID* to unbind a LAN adapter from all protocol stacks, and then deregister the board also.

The MLID remove procedure should call this routine (or *LSLDeRegisterMLID*) for each logical board that the LAN adapter supports.

This routine is identical to *LSLDeRegisterMLID* except that this routine allows protocol stacks to transmit packets that are notifying other machines on the network that this connection is being destroyed (the stack's *Ctl3_Unbind* control routine has been called). For this reason, MLIDs must not use this call if the hardware is having trouble sending packets, such as when a fatal hardware error has occurred.

See Also

- ◆ *LSLDeRegisterMLID*

12 LSL Support Routines (C Language)

This chapter provides the specifications for the C Language Support Routines in the LSL.

When an MLID or a protocol stack calls one of these routines, the EBP, EBX, ESI, and EDI registers are preserved, all parameters are passed to the protocol stack, and EAX has the completion code.

The direction flag is preserved by all C Language Support Routines.

CLD must be in effect. All calls preserve the direction flag.

Table 30 summarizes the C Interface LSL Support Routines.

Table 30 Summary of the C Language Support Routines

Routine Type	Routine	Description
Binding and Registration Routines	CLSLBindStack	Binds a protocol stack to an MLID.
	CLSLRegisterDefaultChain	Puts a default protocol stack in a protocol stack chain for an MLID.
	CLSLRegisterPreScanRxChain	Puts a prescan receive protocol stack in a protocol stack chain for an MLID.
	CLSLRegisterStackRTag	Registers a bound stack with the LSL.
	CLSLRegisterPreScanTxChain	Puts a transmit prescan protocol stack into a protocol stack chain for an MLID.
Miscellaneous Chaining Routines	CLSLGetStartChain	Gets a pointer to the start of all protocol stack chains for a given board.

Routine Type	Routine	Description
NLM Interaction Routines	CLSLAddProtocolID	Adds a Protocol ID to the MLID configuration table.
	CLSLGetMLIDControlEntry	Gets the control entry point of an MLID.
	CLSLGetPIDFromStackIDBoard	Gets a protocol ID for a registered bound protocol stack and logical board combination.
	CLSLGetStackIDFromName	Gets the Stack ID for a registered bound protocol stack.
Reception Routines	CLSLReSubmitDefaultECB	Passes the ECB to another default prescan stack for processing.
	CLSLReSubmitPreScanRxECB	Passes the ECB to another receive protocol stack for processing.
	CLSLReturnRcvECB	Returns a preallocated receive ECB.
Transmission Routines	CLSLReSubmitPreScanTxECB	Passes the ECB to another transmit prescan stack for processing.
	CLSLSendPacket	Sends a packet to the MLID.
Unbinding and Deregistration Routines	CLSLDeRegisterStack	Deregisters a bound protocol stack from the LSL.
	CLSLDeRegisterDefaultChain	Removes a default protocol stack from an MLID's protocol stack chain.
	CLSLDeRegisterPreScanRxChain	Removes a receive prescan protocol stack from an MLID's protocol stack chain.
	CLSLDeRegisterPreScanTxChain	Removes a transmit prescan protocol stack from an MLID's protocol stack chain.
	CLSLUnbindStack	Unbinds a bound protocol stack from an MLID.

CLSLAddProtocolID

Adds a Protocol ID to the LSL's protocol stack table.

Syntax

```
LONG CLSLAddProtocolID (  
    void *ProtocolID,  
    void *ProtocolName,  
    void *MediaName);
```

Parameters

ProtocolID

(IN) Pointer to the 6-byte area containing the Protocol ID.

ProtocolName

(IN) Pointer to a length-preceded, zero-terminated string containing the short name of the protocol stack that is to be added to the LSL's protocol stack table.

MediaName

(IN) Pointer to a length-preceded, zero-terminated string describing the frame type.

Completion Code (EAX)

0x00000000	Successful	The LSL successfully added the new Protocol ID to its configuration table.
0x0FFFFFF82	BadParameter	The specified parameter is an illegal (unknown) name. The protocol name string and media name string length must be equal to or less than 15.
0x0FFFFFF83	DuplicateEntry	A different protocol ID is already registered for the given media/frame type/protocol stack combination.
0x0FFFFFF89	OutOfResources	The LSL has no resources to register another Protocol ID.

Remarks

CLSLAddProtocolID allows a protocol stack to add a new Protocol ID for a given media. Because protocol stacks are topology and frame type unaware, they do not interpret the Protocol ID. Protocol ID values depend on the frame type and topology the protocol stack is using .

If the protocol stack does not find a Protocol ID, the protocol stack will be loaded, but will not be functional. After the protocol stack has loaded, the network administrator can bind the protocol stack to a frame type by typing a Protocol ID on the command line. The syntax is as follows:

```
Protocol Register<Protocol Name><Frame Name><Protocol ID>
```

For example:

```
Protocol Register IP Ethernet_II 800
```

Typing the command: `Protocol`, by itself, at the command line produces the protocols, frame types, and Protocol IDs that have already been registered.

An intelligent protocol stack that has no Protocol ID registered, can register a Protocol ID for itself based on the combination of the frame type and stack name.

For example, IP on `Ethernet_II` or `Ethernet_SNAP` always uses the PID 0x800.

See Also

- ◆ `CLSLGetPIDFromStackIDBoard`
- ◆ `LSLAddProtocolID`

CLSLBindStack

Puts the Stack ID in the LSL's protocol stack table.

Syntax

```
LONG CLSLBindStack (  
    LONG StackIDNumber,  
    LONG BoardNumber);
```

Parameters

StackIDNumber

(IN) Protocol Stack ID number.

BoardNumber

(IN) The board number.

Completion Code (EAX)

0x00000000	Successful	No errors occurred.
0x0FFFFFF82	BadParameter	The board number or the Stack ID is invalid.
0x0FFFFFF83	DuplicateEntry	The specified binding already exists.
0x0FFFFFF89	OutOfResources	The routine could not allocate enough memory.

Remarks

NOTE: This routine must be called at process time only.

CLSLBindStack binds a protocol stack to an MLID, allowing the protocol stack to receive packets from the MLID. The MLID passes the frames it receives to the LSL. The LSL then checks a table to find which protocol stack's Protocol ID matches the one embedded in the frame's MAC header. The LSL then passes the frames to that protocol stack's receive handler. (The protocol stack specifies the address of the receive handler when it registers with the LSL.)

IMPORTANT: Before making this call, the protocol stack must be ready to receive frames and must have already registered by calling either *LSLRegisterStackRTag* or *CLSLRegisterStackRTag*.

See Also

- ◆ *CLSLUnBindStack*
- ◆ *LSLBindStack*
- ◆ *CLSLRegisterStackRTag*
- ◆ *LSLRegisterStackRTag*
- ◆ *BindProtocolToBoard* in Appendix A, "Operating System Support Routines".

CLSLControlStackFilter

Notifies all protocol stacks bound to an MLID that the MLID status has changed.

Syntax

```
LONG CLSLControlStackFilter (  
    LONG BoardNumber,  
    LONG FunctionNumber,  
    LONG FilterMask,  
    LONG Parameter1,  
    LONG Parameter2);
```

Parameters

BoardNumber

(IN) The logical board number notifying the filtering of packets to protocol stacks which are bound/registered with this logical board (bound and chained stacks).

FunctionNumber

(IN) The protocol stack control handler function number to be called.

FilterMask

(IN) The filter mask for all stacks to be called. See *LSLGetStartChain* for the definitions of the bits in this mask.

Parameter1

(IN) Possible parameter to pass to the control handler.

Parameter2

(IN) Possible parameter to pass to the control handler.

Completion Code (EAX)

0x00000000	Successful	All stacks bound/registered with the logical board have been notified.
------------	------------	--

Remarks

When an MLID enters promiscuous mode, it can use this routine to call all protocol stacks that need to know that the MLID is now in promiscuous mode.

This function updates all protocol stacks associated with the physical LAN adapter that the logical board is operating on. This function updates other protocol stacks that are operating on logical boards that have the same name and instance as the logical board specified by *BoardNumber*.

See Also

- ◆ CLSLModifyStackFilter
- ◆ CLSLControlStackFilter

CLSLDeRegisterDefaultChain

Removes the chained default stack from the LSL tables.

Syntax

```
LONG CLSLDeRegisterDefaultChain (  
    LONG StackChainID);
```

Parameters

StackChainID

(IN) The chain Stack ID of the default protocol stack.

Interrupts

(IN) Any state.

Call

(IN) At process time only.

Completion Code (EAX)

0x00000000	Successful	The protocol stack was deregistered.
0x0FFFFFF85	ItemNotPresent	No chained default stack with this Stack ID is registered for this MLID.

Remarks

CLSLDeRegisterDefaultChain removes the specified chained default stack for a logical board from the LSL's internal default stack table.

After making this call, the protocol stack chain will not receive send packets for the MLID it was registered with.

See Also

- ◆ LSLDeRegisterDefaultChain

- ◆ CLSLRegisterDefaultChain

CLSLDeRegisterPreScanRxChain

Removes the chained prescan receive stack from the LSL's internal default stack table.

Syntax

```
LONG CLSLDeRegisterPreScanRxChain (  
    LONG StackChainID);
```

Parameters

StackChainID

(IN) The chain Stack ID of the prescan receive protocol stack.

Interrupts

(IN) Any state.

Call

(IN) At process time only.

Completion Code (EAX)

0x00000000	Successful	The protocol stack was deregistered.
0x0FFFFFFF85	ItemNotPresent	No chained prescan receive stack with this Stack ID is registered for this MLID.

Remarks

CLSLDeRegisterPreScanRxChain removes the specified chained prescan receive stack for a logical board from the LSL's internal default stack table.

After making this call, the protocol stack chain will not receive send packets for the MLID it was registered with.

See Also

- ◆ LSLDeRegisterPreScanRxChain
- ◆ CLSLRegisterPreScanRxChain

CLSLDeRegisterPreScanTxChain

Removes the chained prescan transmit stack from the LSL's internal default stack table.

Syntax

```
LONG CLSLDeRegisterPreScanTxChain (  
    LONG StackChainID);
```

Parameters

StackChainID

(IN) The chain Stack ID of the prescan transmit protocol stack.

Interrupts

(IN) Any state.

Call

(IN) At process time only.

Completion Code (EAX)

0x00000000	Successful	The protocol stack was deregistered.
0x0FFFFFF85	ItemNotPresent	No chained prescan transmit stack with this Stack ID is registered for this MLID.

Remarks

CLSLDeRegisterPreScanTxChain removes the specified chained prescan transmit stack for a logical board from the LSL's internal default stack table.

After making this call, the protocol stack chain will not receive send packets for the MLID it was registered with.

See Also

- ◆ LSLDeRegisterPreScanTxChain
- ◆ CLSLRegisterPreScanTxChain

CLSLDeRegisterStack

Removes a bound stack from the LSL's internal default stack table.

Syntax

```
LONG CLSLDeRegisterStack (  
    LONG ProtocolNumber);
```

Parameters

ProtocolNumber

(IN) The protocol Stack ID.

Interrupts

(IN) Any state.

Call

(IN) At process time only.

Completion Code (EAX)

0x00000000	Successful	The protocol stack was deregistered.
0x0FFFFFF82	BadParameter	The Stack ID is invalid.

Remarks

CLSLDeRegisterStack removes the specified protocol stack from the LSL's internal bound protocol stack table.

After making this call, the protocol stack will not receive packets from an MLID unless it is also registered with the MLID as a default or prescan receive protocol stack.

See Also

- ◆ LSLDeRegisterStack

◆ CLSLRegisterStack

CLSLGetBoundBoardInfo

Returns the Stack ID of a protocol stack bound to a specified logical board or LAN adapter.

Syntax

```
LONG CLSLGetBoundBoardInfo (  
    LONG BoardNumber,  
    void *StackBuffer);
```

Parameters

BoardNumber

(IN) The logical board number.

StackBuffer

(IN) Pointer to an array of n LONGS where n is the maximum possible number of protocol stacks.

StackBuffer

(OUT)

- ◆ [Buffer+0] = the number of protocol stacks
- ◆ [Buffer+4] = the ID of the 1st protocol stack
- ◆ [Buffer+8] = the ID of the 2nd protocol stack
- ◆ [Buffer+4n] = the ID of the nth protocol stack

Completion Code (EAX)

0x00000000	Successful	The routine returned no errors.
0x0FFFFFF82	BadParameter	The board number is invalid.
0x0FFFFFF85	ItemNotPresent	The board number is not registered.
0X0FFFFFF89	OutOfResources	The buffer is too small. The first 4 bytes of the buffer will contain the appropriate buffer length.

Remarks

CLSLGetBoundBoardInfo returns the Stack IDs of all the protocol stacks that are bound to the specified logical board or LAN adapter.

CLSLGetMLIDControlEntry

Returns the control entry point for the specified MLID.

Syntax

```
LONG CLSLGetMLIDControlEntry (  
    LONG BoardNumber,  
    void (*ControlEntryPoint)(void));
```

Parameters

BoardNumber

(IN) The board number.

ControlEntryPoint

(IN) Pointer to a buffer of size sizeof(*void).

ControlEntryPoint

(OUT) Pointer to a pointer holding the address of the MLID's control handler routine.

Completion Code (EAX)

0x00000000	Successful	The buffer pointed to by <i>ControlEntryPoint</i> contains the MLID control entry point.
0x0FFFFFF82	BadParameter	The board number is invalid.

Remarks

This routine must be called at process time only.

CLSLGetMLIDControlEntry returns the MLID Control Entry Point for the MLID specified by *BoardNumber*.

This routine allows a protocol stack to communicate directly with an MLID and get information such as the addresses of the MLID configuration table and statistics table.

See Also

- ◆ LSLGetMLIDControlEntry
- ◆ Chapter 20, "MLID Control Routines"
- ◆ Chapter 14, "MLID Data Structures"

CLSLGetPIDFromStackIDBoard

Returns a Protocol ID.

Syntax

```
LONG CLSLGetPIDFromStackIDBoard (  
    LONG ProtocolNumber,  
    LONG BoardNumber,  
    void *ProtocolID);
```

Parameters

ProtocolNumber

(IN) The Stack ID.

BoardNumber

(IN) The board number.

ProtocolID

(IN) Pointer to a 6-byte area for the Protocol ID.

ProtocolID

(OUT) Pointer to the 6-byte Protocol ID

Completion Code (EAX)

0x00000000	Successful	The PID associated with the protocol stack and logical board was returned.
0x0FFFFFF82	BadParameter	The board number is invalid.
0x0FFFFFF85	ItemNotPresent	No Protocol ID is associated with the parameters passed in.

Remarks

This routine must be called at process time only.

CLSLGetPIDFromStackIDBoard returns a Protocol ID that corresponds to a combination of the protocol Stack ID and a board number. The protocol stack puts this information in the *ProtocolID* field of a send ECB.

See Also

- ◆ LSLGetPIDFromStackIDBoard

CLSLGetProtocolControlEntry

Gets an entry point to a protocol stack.

Syntax

```
LONG CLSLGetProtocolControlEntry (  
    LONG StackID,  
    LONG BoardID,  
    void (*ControlEntryPoint) (void));
```

Parameters

StackID

(IN) The Stack ID.

BoardID

(IN) The board number.

ControlEntryPoint

(IN) Pointer to the pointer to be filled in with the address of the protocol stack's control handler routine.

ControlEntryPoint

(OUT) Pointer to the address of the protocol stack's control handler routine.

Completion Code (EAX)

0x00000000	Successful	The protocol stack's control entry point is at the address pointed to by ControlEntryPoint.
0x0FFFFFF82	BadParameter	The specified protocol stack is not bound to the specified board.
0x0FFFFFF85	ItemNotPresent	The Stack ID is invalid.
0x0FFFFFF86	NoMoreItems	The Board ID is invalid.

Remarks

This routine must be called at process time only.

CLSLGetProtocolControlEntry allows a protocol stack, or an application, to communicate directly with another protocol stack and get information from the LSL's list of protocol stacks.

To get a control entry point for a default or prescan protocol stack, use *CLSLGetStartChain*.

See Also

- ◆ LSLGetProtocolControlEntry
- ◆ LSLGetStartChain
- ◆ CLSLGetStartChain

CLSLGetStackIDFromName

Gets a Stack ID for any protocol stack.

Syntax

```
LONG CLSLGetStackIDFromName (  
    BYTE *Name,  
    LONG *ProtocolNumber);
```

Parameters

Name

(IN) Pointer to a length-preceded, zero-terminated string containing the name of the protocol stack.

ProtocolNumber

(IN) Pointer to the LONG variable that will hold the Stack ID.

ProtocolNumber

(OUT) Pointer to the Stack ID.

Completion Code (EAX)

0x00000000	Successful	The Stack ID is at the address in <i>ProtocolNumber</i> .
0x0FFFFFF82	BadParameter	The length of the name is 0 or is greater than 15 characters.
0x0FFFFFF85	ItemNotPresent	The given protocol stack name is not registered with the LSL.

Remarks

This routine must be called at process time only.

CLSLGetStackIDFromName allows a protocol stack, or an application, to get its own Stack ID or another Stack ID.

If the Stack ID and the MLID board number are known, a protocol stack can get the Protocol ID by calling *CLSLGetPIDFromStackIDBoard*.

NOTE: The match of the stack name is case sensitive.

See Also

- ◆ LSLGetStackIDFromName

CLSLGetStartChain

Gets pointers to the starting offsets of the default chain, the prescan receive chain, and the prescan transmit chain.

Syntax

```
LONG CLSLGetStartChain (  
    LONG BoardNumber,  
    void **DefaultChainPtr,  
    void **PreScanRxChainPtr,  
    void **PreScanTxChainPtr);
```

Parameters

BoardNumber

(IN) The board number of the logical board or LAN adapter bound to the desired chain.

DefaultChainPtr

(OUT) Pointer to the address of the default stack chain pointer.

PreScanRxChainPtr

(OUT) Pointer to the address of the prescan receive stack chain.

PreScanTxChainPtr

(OUT) Pointer to the address of the prescan transmit stack chain pointer.

Completion Code (EAX)

0x00000000	Successful	The routine successfully returned the pointers.
0x0FFFFFF82	BadParameter	The board number is invalid.

Remarks

CLSLGetStartChain returns the pointers to the starting offsets of the default chain, the prescan receive chain, and prescan transmit chain. Each chain is a

linked list of protocol stacks. If no protocol stacks are registered for a particular chain, a NULL pointer is returned.

```

ChainStructure      struc
ChainLink           dd    ?    ;Link Field
ChainProcessHandler dd    ?    ;Stack Tx or Rx Handler
ChainCtlHandler     dd    ?    ;Stack Control Entry Point
ChainRTag           dd    ?    ;Stack Resource Tag
ChainReceiveBufferRTag dd  ?    ;Receive ECB's Resource Tag
ChainLoadOrder      dd    ?    ;Stack Requested Chain Position
ChainID             dd    ?    ;Stack ID Number (-1 indicates
                                an old style prescan or
                                default protocol stack.)
ChainMask           dd    ?    ;Stack Packet Filter Mask
ChainStructure      ends

```

The ChainMask assigns the bit values as follows:

DEST_MULTICAST	equ	0x0001	
DEST_BROADCAST	equ	0x0002	
DEST_REMOTE_UNICAST	equ	0x0004	
DEST_REMOTE_MULTICAST	equ	0x0008	
DEST_NO_ROUTE	equ	0x0010	Exclusive bit
DEST_ERROR_PACKET	equ	0x0020	
Reserved	equ	0x0040	
DEST_DIRECT	equ	0x0080	
DEST_PROMISCUOUS	equ	0xFFFF	All packets, including errors

See Also

- ◆ LSLGetStartChain

CLSLModifyStackFilter

Modifies the ChainMask field of the Chain Structure.

Syntax

```
LONG CLSLModifyStackFilter (  
    LONG StackID,  
    LONG BoardNumber,  
    LONG NewMask,  
    LONG *CurrentMask);
```

Parameters

StackID

(IN) The protocol Stack ID or the chain Stack ID.

BoardNumber

(IN) The logical board number.

NewMask

(IN) The new ChainMask filter bits (zero to query).

CurrentMask

(IN) Pointer to a LONG that will hold the current mask

CurrentMask

(OUT) Pointer to the current setting of ChainMask.

Completion Code (EAX)

0x00000000	Successful	The ChainMask filter has been set to the new value.
0x0FFFFFF82	BadParameter	The board number, the stack ID, or the chain ID is invalid.

Remarks

CLSLModifyStackFilter is used by protocol stacks to set the ChainMask field of the ChainStructure.

If the protocol stack is bound, this call sets the filter mask.

The protocol stack uses the chain mask or the filter mask to notify the LSL of the type of packets it wants to receive. (See *CLSLGetStartChain* for the definition of the ChainStructure.)

See Also

- ◆ *CLSLGetStartChain*

CLSLRegisterDefaultChain

Binds a chained default protocol stack to the specified MLID.

Syntax

```
LONG CLSLRegisterDefaultChain (  
    struct ResourceTagStructure *StackChainRTag,  
    LONG StackChainBoardNumber,  
    LONG StackChainPositionRequested,  
    void *StackChainID,  
    LONG (*StackChainReceiveHandler) (  
        ECB *RxECB,  
        LONG BoardNumber,  
        void *ChainID),  
    void (*StackChainControlEntryPoint) (void),  
    struct ResourceTagStructure *StackChainReceiveECBRTag  
);
```

Parameters

StackChainRTag

(IN) Pointer to the protocol stack's resource tag.

StackChainBoardNumber

(IN) The board number.

StackChainPositionRequested

(IN) Request for a chain position (see Remarks below).

StackChainReceiveHandler

(IN) Pointer to the protocol stack's receive handler. The receive handler is called with a pointer to a receive ECB, the board number, and a pointer to the protocol stack's chain ID.

StackChainControlEntryPoint

(IN) Pointer to the protocol stack's control entry point.

StackChainReceiveECBRTag

(IN) Pointer to the receive ECB's resource tag.

StackChainID

(OUT) Pointer to a buffer that holds the chained stack's ID.

Completion Code (EAX)

0x00000000	Successful	No error occurred.
0x0FFFFFF82	BadParameter	The board number is invalid, one of the resource tags is invalid, or an invalid position in the stack chain has been requested.
0x0FFFFFF83	DuplicateEntry	The requested chain position is either the first or last position (<code>StackChainPositionRequested</code> equals either 0 or 4) and is already occupied.
0x0FFFFFF89	OutOfResources	The LSL was unable to allocate the node structure for the chain.

Remarks

CLSLRegisterDefaultChain binds the chained default protocol stack to the MLID whose board number is specified in *StackChainBoardNumber*. A protocol stack uses this routine to insert itself into the proper position in the chain.

In order to acquire the stack chain resource tag, the protocol stack calls the operating system support routine *AllocateResourceTag* with the *LSLDefaultStackSignature* 'DLSL'.

The stack chain ID returned is a unique identifier based on a logical board/protocol stack chain combination. This ID is not the same as the Stack ID, which is issued to a single, normally registered, protocol stack that can be bound to a number of different boards.

The protocol stack's required position in the chain is defined by the following table.

Table 31 Chain Position Values

Parameter Value	Position in Chain
0	The stack must start the chain.

Parameter Value	Position in Chain
1	The stack must be loaded at the next available position from the front of the chain.
2	The stack's position in the chain is dependent on the order in which the stacks are loaded. The stack will take the next available position when it loads.
3	The stack must be loaded at the next available position from the end of the chain.
4	The stack must end the chain.

IMPORTANT: If the protocol stack requests a chain position of 0 or 4, the registration routine will abort if another protocol stack is already registered at that position for that board number.

See Also

- ◆ `CLSLDeRegisterDefaultChain`
- ◆ `LSLRegisterDefaultChain`

CLSLRegisterPreScanRxChain

Binds a chained receive prescan protocol stack to the specified MLID.

Syntax

```
LONG CLSLRegisterPreScanRxChain (  
    struct ResourceTagStructure *StackChainRTag,  
    LONG StackChainBoardNumber,  
    LONG StackChainPositionRequested,  
    LONG *StackChainID,  
    LONG StackChainReceiveHandler(  
        ECB *RxECB,  
        LONG BoardNumber,  
        void *ChainID) ,  
    void (*StackChainControlEntryPoint) (),  
    struct ResourceTagStructure *StackChainReceiveECBRTag  
);
```

Parameters

StackChainRTag

(IN) Pointer to the protocol stack's resource tag.

StackChainBoardNumber

(IN) The board number.

StackChainPositionRequested

(IN) Request for a chain position (see Remarks below).

StackChainReceiveHandler

(IN) Pointer to the protocol stack's receive handler. The receive handler is called with a pointer to a receive ECB, the board number, and a pointer to the protocol stack's chain ID.

StackChainControlEntryPoint

(IN) Pointer to the protocol stack's control entry point.

StackChainReceiveECBRTag

(IN) Pointer to the receive ECB's resource tag.

StackChainID

(OUT) Pointer to a LONG that holds the chained stack's ID.

Completion Code (EAX)

0x00000000	Successful	No error occurred.
0x0FFFFFF82	BadParameter	The board number is invalid, one of the resource tags is invalid, or an invalid position in the stack chain has been requested.
0x0FFFFFF83	DuplicateEntry	The requested chain position is either the first or last position (<code>StackChainPositionRequested</code> equals either 0 or 4) and is already occupied.
0X0FFFFFF89	OutOfResources	The LSL was unable to allocate the node structure for the chain.

Remarks

CLSLRegisterPreScanRxChain binds the chained receive prescan protocol stack to the MLID whose board number is specified in *StackChainBoardNumber*. A protocol stack uses this routine to insert itself into the proper position in the chain.

In order to acquire the stack chain resource tag, the protocol stack calls the operating system support routine *AllocateResourceTag* with the *LSLPreScanStackSignature* 'PLSL'.

The stack chain ID returned is a unique identifier based on a logical board/protocol stack chain combination. This ID is not the same as the Stack ID, which is issued to a single, normally registered, protocol stack that can be bound to a number of different boards.

The protocol stack's required position in the chain is defined by the following table.

Table 32 Chain Position Values

Parameter Value	Position in Chain
0	The stack must start the chain.

Parameter Value Position in Chain

1	The stack must be loaded at the next available position from the front of the chain.
2	The stack's position in the chain is dependent on the order in which the stacks are loaded. The stack will take the next available position when it loads.
3	The stack must be loaded at the next available position from the end of the chain.
4	The stack must end the chain.

IMPORTANT: If the protocol stack requests a chain position of 0 or 4, the registration routine will abort if another protocol stack is already registered at that position for that board number.

See Also

- ◆ CLSLDeRegisterPreScanRxChain
- ◆ LSLRegisterPreScanRxChain

CLSLRegisterPreScanTxChain

Binds a chained transmit prescan protocol stack to the specified MLID.

Syntax

```
LONG CLSLRegisterPreScanTxChain (
    struct ResourceTagStructure *StackChainRTag,
    LONG StackChainBoardNumber,
    LONG StackChainPositionRequested,
    LONG *StackChainID,
    LONG StackChainReceiveHandler(
        ECB *RxECB,
        LONG BoardNumber,
        void *ChainID),
    void (*StackChainControlEntryPoint) (),
    struct ResourceTagStructure *StackChainReceiveECBRTag
);
```

Parameters

StackChainRTag

(IN) Pointer to the protocol stack's resource tag.

StackChainBoardNumber

(IN) The board number.

StackChainPositionRequested

(IN) Request for a chain position (see Remarks below).

StackChainReceiveHandler

(IN) Pointer to the protocol stack's receive handler. The receive handler is called with a pointer to a receive ECB, the board number, and a pointer to the protocol stack's chain ID.

StackChainControlEntryPoint

(IN) Pointer to the protocol stack's control entry point.

StackChainReceiveECBRTag

(IN) Pointer to the receive ECB's resource tag.

StackChainID

(OUT) Pointer to a LONG that holds the chained stack's ID.

Completion Code (EAX)

0x00000000	Successful	No error occurred.
0x0FFFFFF82	BadParameter	The board number is invalid, one of the resource tags is invalid, or an invalid position in the stack chain has been requested.
0x0FFFFFF83	DuplicateEntry	The requested chain position is either the first or last position (<code>StackChainPositionRequested</code> equals either 0 or 4) and is already occupied.
0X0FFFFFF89	OutOfResources	The LSL was unable to allocate the node structure for the chain.

Remarks

CLSLRegisterPreScanTxChain binds the chained transmit prescan protocol stack to the MLID whose board number is specified in *StackChainBoardNumber*. A protocol stack uses this routine to insert itself into the proper position in the chain.

In order to acquire the stack chain resource tag, the protocol stack calls the operating system support routine *AllocateResourceTag* with the *LSLPreScanStackSignature* 'TLSSL'.

The stack chain ID returned is a unique identifier based on a logical board/protocol stack chain combination. This ID is not the same as the Stack ID, which is issued to a single, normally registered, protocol stack that can be bound to a number of different boards.

The protocol stack's required position in the chain is defined by the following table.

Table 33 Chain Position Values

Parameter Value	Position in Chain
0	The stack must start the chain.

Parameter Value	Position in Chain
1	The stack must be loaded at the next available position from the front of the chain.
2	The stack's position in the chain is dependent on the order in which the stacks are loaded. The stack will take the next available position when it loads.
3	The stack must be loaded at the next available position from the end of the chain.
4	The stack must end the chain.

IMPORTANT: If the protocol stack requests a chain position of 0 or 4, the registration routine will abort if another protocol stack is already registered at that position for that board number.

See Also

- ◆ CLSLDeRegisterPreScanTxChain
- ◆ LSLRegisterPreScanTxChain

CLSLRegisterStackRTag

Registers a bound protocol stack with the LSL. The stack must then bind with a logical board before it can receive. Generally used by protocol stacks.

Syntax

```
LONG CLSLRegisterStackRTag (  
    void *ProtocolStackName,  
    void *ReceiveEntryPoint,  
    void *ControlEntryPoint,  
    LONG *StackID,  
    struct ResourceTagStructure *StackRTag,  
    struct ResourceTagStructure *StackReceiveECBRTag) ;
```

Parameters

ProtocolStackName

(IN) Pointer to the name of the protocol stack.

ReceiveEntryPoint

(IN) Pointer to the protocol stack's receive handler.

ControlEntryPoint

(IN) Pointer to the protocol stack's control entry point.

StackRTag

(IN) Handle to the ResourceTagStructure.

StackReceiveECBRTag

(IN) Handle to the receive ECB resource tag.

StackID

(OUT) Pointer to the protocol Stack ID.

Completion Code (EAX)

0x0000000	Successful	The protocol stack registered successfully.
0x0FFFFFF82	BadParameter	The resource tag was invalid, or the length of the protocol name equaled zero, or the length of the protocol name was greater than 15.
0x0FFFFFF83	DuplicateEntry	This protocol stack is already registered.
0x0FFFFFF89	OutOfResources	The LSL was unable to allocate the node structure for the chain.

Remarks

This routine must be called only at process time.

When the protocol stack is bound at the command line, the Bind IOCTL is called.

Registering the protocol stack does not mean that it can receive packets. To enable packet reception through the LSL, the protocol stack must call *CLSLBindStack* after the network administrator has requested to bind the protocol stack to an MLID. The protocol stack only makes these calls once for each MLID that the stack wants to receive packets from.

Keep in mind that a protocol stack can transmit packets and communicate with MLIDs even if it has not registered with the LSL or bound to an MLID. If a protocol stack does not bind to an MLID, it must call either *CLSLRegisterDefaultChain* or *CLSLRegisterPreScanRxChain* in order to receive packets.

See Also

- ◆ *CLSLBindStack*
- ◆ *LSLBindStack*
- ◆ *CLSLRegisterStackRTag*
- ◆ *LSLRegisterStackRTag*

CLSLReSubmitDefaultECB

Allows the next stack in the default stack chain to process the ECB

Syntax

```
LONG CLSLReSubmitDefaultECB (  
    LONG StackChainID,  
    void *ReceiveECB);
```

Parameters

StackChainID

(IN) Contains the chained Stack ID.

ReceiveECB

(IN) Pointer to a received ECB to process.

Completion Code (EAX)

None.

Remarks

CLSLReSubmitDefaultECB allows chained default protocol stacks to pass ECBs to the LSL for further processing.

A protocol stack normally calls this routine after the stack has queued the ECB for further processing at process time. Then, after the protocol stack has processed the ECB during process time, it needs to pass the ECB on to other stacks in the chain. For example, default stacks that are performing data compression would use this call.

See Also

- ◆ LSLReSubmitDefaultECB

CLSLReSubmitPreScanRxECB

Allows the next stack in the receive prescan stack chain to process the ECBs.

Syntax

```
void CLSLReSubmitPreScanRxECB (  
    LONG StackChainID,  
    void *ReceiveECB);
```

Parameters

StackChainID

(IN) Contains the chained Stack ID.

ReceiveECB

(IN) Pointer to a receive ECB to process.

Completion Code (EAX)

None.

Remarks

CLSLReSubmitPreScanRxECB allows chained prescan receive protocol stacks to pass ECBs to the LSL for further processing.

A protocol stack normally calls this routine after the stack has queued the ECB for further processing at process time. Then, after the protocol stack has processed the ECB during process time, it needs to pass the ECB on to other stacks in the chain.

See Also

- ◆ LSLReSubmitPreScanRxECB

CLSLReSubmitPreScanTxECB

Allows the next stack in the transmit prescan stack chain to process the ECBs.

Syntax

```
void CLSLReSubmitPreScanTxECB (  
    LONG StackChainID,  
    void *SendECB);
```

Parameters

StackChainID

(IN) Contains the chained Stack ID.

SendECB

(IN) Pointer to a transmit ECB to process.

Completion Code (EAX)

None.

Remarks

LSLReSubmitPreScanTxECB allows chained prescan transmit protocol stacks to pass ECBs to the LSL for further processing.

A protocol stack normally calls this routine after the stack has queued the ECB for further processing at process time. Then, after the protocol stack has processed the ECB during process time, it needs to pass the ECB on to other stacks in the chain.

See Also

- ◆ LSLReSubmitPreScanTxECB

CLSLReturnRcvECB

Returns a preallocated receive ECB.

Syntax

```
void CLSLReturnRcvECB (  
    void *ReceiveECB);
```

Parameters

ReceiveECB

(IN) Pointer to the receive ECBs.

Completion Code (EAX)

None.

Remarks

This routine must be called only at process time.

Protocol stacks call *CLSLReturnRcvECB* to return a previously allocated receive ECBs buffer to the LSL.

See Also

- ◆ LSLGetRcvECBRTag

CLSLSendPacket

Sends the packet to the MLID

Syntax

```
LONG CLSLSendPacket (  
    void *SendECB);
```

Parameters

SendECB

(IN) Pointer to the send ECB.

Completion Code (EAX)

0x00000000	Successful	The ECB was successfully passed to the MLID.
0x0FFFFFF85	ItemNotPresent	The board number in ECB was invalid.

Remarks

CLSLSendPacket sends a packet to one of the registered MLIDs.

The ESR field of the ECB must have the address of an Event Service Routine to call when the send is complete. Until the ESR is called, the ECB and all its data areas belong to the Link Support Layer and must not be touched. (Stacks must not read or write any of the ECB fields.)

NOTE: The LSL can call the ESR before the call to *CLSLSendPacket* returns.

Priority Sends

Priority sends are sent with the *stackID* field of the ECB set to a value between 0FFF7h (lowest priority) and 0FFF0h (highest priority).

Raw Sends

If the ECB is sent in "raw" mode, the fragment list contains the complete packet, including the link-level envelope. However, the link-level envelope

must be entirely contained within the first fragment. In other words, the envelope cannot be split between the first and second fragments.

All media/frame type anomalies must be accounted for in the media/frame type header. Raw packets are sent with the StackID field of the ECB set equal to FFFFh through OFFF8h. (These values represent priority raw sends.) The protocol stack must check the board's configuration table *MLIDModeFlags* field to see if the board supports raw sends (see Chapter 14, "MLID Data Structures" for the MLID configuration table format).

The ESR is called with ESI (and the first parameter on the stack) containing a pointer to the ECB that was sent. This call is made at either interrupt or process time with interrupts disabled. The ESR can enable interrupts, but if it does so, it must guard against reentry. The ESR should complete quickly.

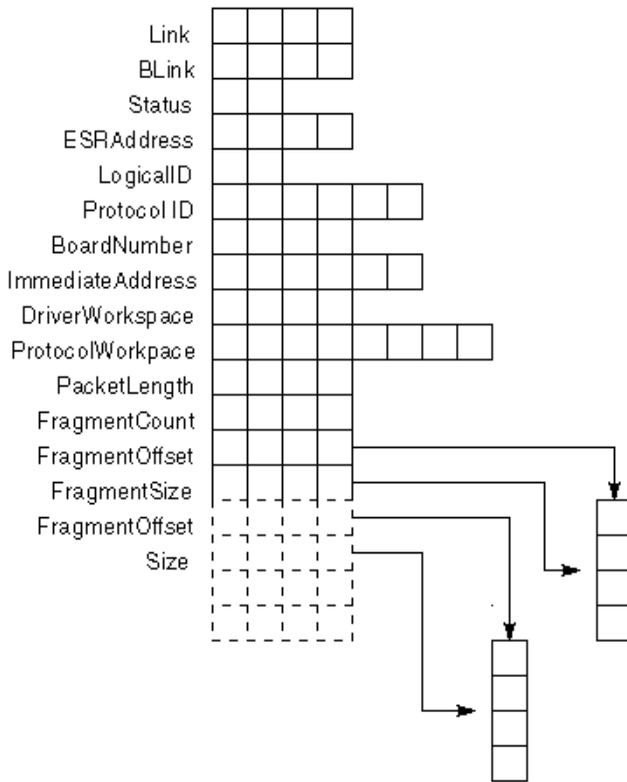
The following fields in the ECB must also be set before calling *LSLSendPacket*.

Table 34 **ECB Fields to Set Before Calling LSLSendPacket**

Offset	Field Name	Description
0Ah	ESRAddress	The address of a routine that is called when the ECB is released (after the packet has been transmitted). A pointer to the ECB is passed to this routine in ESI. This pointer is also the first parameter on the stack for ESRs written in C. This field is a near pointer.
0Eh	LogicalID	The Stack ID of the protocol stack sending the packet. See the "Raw Send" and "Priority Sends" sections above for more detail.
10h	ProtocolID	The Protocol ID (returned by LSLGetPIDFromStackIDBoard) that the MLID is to use when excapsulating the data. This field is ignored if a raw packet is sent. See the section 7.6.1.2 - The ECB ProtocolID Field and Ethernet 802.2 for more detail.
16h	BoardNumber	The board number of the MLID sending this packet.
1Ah	ImmediateAddress	The node address on the physical network that the packet is destined for. If the packet is raw, this field is undefined. The address FFFFFFFFFFh always indicates a broadcast packet. (A broadcast packet is received by all nodes on the physical network.)
2Ch	PacketLength	The total length of all fragment buffers.

Offset	Field Name	Description
30h	FragmentCount	The number of fragments in the packet to be sent. Descriptor data structures follow this field. This field must contain a value between 1 and 16, inclusive. The protocol stack can specify a maximum of 16 fragment descriptors. The MLID combines these fragments together to form one contiguous packet.
34h	FragmentOffset	On sends, this field describes the location of a contiguous section of RAM memory (32-bit offset).
38h	FragmentLength	The length in bytes of the first packet fragment. On sends, this value can be 0. On sends, the ECB may contain the following additional fields as needed:
3Ch	FragmentOffsetX	Additional fragment descriptor when the FragmentCount is greater than 1. The X stands for the additional fragment number (2 through 16).
40h	FragmentLengthX	Additional fragment descriptor when the FragmentCount is greater than 1. The X stands for the addition fragment number (2 through 16) . The FragmentOffsetX and FragmentLengthX fields may repeat up to 16 times.

Figure 15 Graphic Representation of the Transmit Event Control Block



See Also

- ◆ Chapter 4, "Protocol Stack Data Structures" for more information about Event Control Blocks

CLSLUnbindStack

Removes a bound protocol stack from the LSL's table

Syntax

```
LONG CLSLUnbindStack (  
    LONG StackID,  
    LONG BoardID);
```

Parameters

StackID

(IN) The protocol Stack I D.

BoardID

(IN) The board number.

Completion Code (EAX)

0x00000000	Successful	No error occurred.
0x0FFFFFF82	BadParameter	The requested board number or the protocol Stack ID does not exist.
0x0FFFFFF85	ItemNotPresent	The specified binding does not exist.

Remarks

CLSLUnbindStack unbinds a bound protocol stack from an MLID.

After this call is completed, the protocol stack no longer receives packets from the MLID it was once bound to. However, the protocol stack can still receive packets from that board if it is also registered as a default or receive prescan protocol stack for it.

See Also

- ◆ LSLBindStack

13 Overview of the MLID

Chapter Overview

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

The NetWare Server MLID

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

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

MLID Procedures

In the NetWare server environment, the ODI specification defines the following procedures:

- ◆ MLID initialization routine (Required)
- ◆ Board service routine (one of the following required)
 - ◆ Interrupt Service Routine (ISR)
 - ◆ Driver polling routine
 - ◆ Second Interrupt Service Routine (optional)
- ◆ Packet transmission routine (Required)

The MLID also supports the following control procedures:

- ◆ Control procedures for the ODI (IOCTLs)
 - ◆ AddMulticastAddress (Required if hardware supports multicast addressing)
 - ◆ DeleteMulticastAddress (Required if hardware supports multicast addressing)
 - ◆ GetMLIDConfiguration (Required)
 - ◆ GetMLIDStatistics (Required)
 - ◆ DriverPromiscuousChange (Recommended)
 - ◆ SetLookAheadSize (Required)
 - ◆ DriverManagement (Optional)
 - ◆ MLIDReset (Required)
 - ◆ MLIDShutdown (Required)
 - ◆ RemoveNetworkInterface (Optional)
 - ◆ ShutdownNetworkInterface (Optional)
 - ◆ ResetNetworkInterface (Optional)
- ◆ Timeout detection (some LAN adapters do not need to provide this procedure)
 - ◆ Interrupt call back routine (Optional)
 - ◆ AES call back routine (Optional)
- ◆ MLID removal routine (Required)

Of course, the specific hardware requirements of the LAN adapter might 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 provided below. These descriptions are high-level generalizations only and are not true in every case, nor do they describe every possible case. More detailed descriptions of each procedure (including pseudocode) is provided in Chapters 15 through 20 of this manual.

MLID Initialization

- ◆ In general terms, the MLID's initialization routine must perform the following actions:
- ◆ Allocate the memory for the MLID's variables and structures.
- ◆ Parse the standard LOAD command line options.
- ◆ Process custom command line parameters and custom firmware.
- ◆ Register the MLID with the LSL.
- ◆ Register the hardware configuration with the operating system.
- ◆ 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:

- ◆ Packet received
- ◆ Error receiving a packet
- ◆ Transmission complete
- ◆ Error transmitting a packet

The MLID can be notified of these events by an interrupt service routine (ISR), a polling procedure, or a polling procedure with interrupt backup.

The system ISR receives the interrupt and calls the MLID's interrupt service routine. When the system ISR calls the MLID's ISR, the direction flag is cleared, interrupts are disabled, and all registers are pushed onto the stack. The MLID only needs to service the interrupt and return (do not use `iret`). If the MLID sets the direction flag during the routine, it should clear it before it returns.

IMPORTANT: We recommend that interrupts remain disabled during MLID's interrupt service routine and its packet transmission routine. If either routine must enable interrupts, it must disable them before returning.

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 sends the packet.

Multiple Operating System Support

MLID development depends upon the operating system under which the MLID will run. An MLID developed to the NetWare operating system must be developed differently than an MLID developed to the DOS, OS/2, or NT operating systems. We strongly recommend that you use the LAN driver toolkit to aid you in developing an HSM (Hardware Specific Module), instead of a complete MLID.

Control Procedures

Among the control procedures 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 that support the Hub Management Interface implement the `DriverManagement IOCTL`.

Timeout Detection

The MLID can schedule timers that it uses to repeatedly call back the *DriverAESCallback* or the *DriverINTCallback* routines at specified intervals.

For example, the MLID might need to be called regularly so that it can inspect the LAN adapter to determine if the adapter has failed to complete a transmission. If a timeout error had occurred, the procedure would discard the packet being sent, reset the board, and begin transmitting the next packet in the send queue.

Driver Remove

Every MLID must have a remove procedure that allows the network supervisor to unload the MLID from the operating system. This procedure must shut down the LAN adapter and return any resources that the MLID has allocated from the operating system.

MLID Data Structures and Variables

In addition to the procedures discussed above, the MLID must also contain certain data structures and variables. The primary structures include:

- ♦ MLID configuration table
- ♦ MLID statistics table

Configuration Table

The 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 the MLID. The requirements for configuration tables are explained in detail in Chapter 14.

Statistics Table

The statistics table is a data structure that contains data on the operation of the LAN adapter and the MLID. Both the LSL and the MLID look at fields in this table. Chapter 14, "MLID Data Structures" contains a detailed description of this data structure.

MLID Functionality

Reentrancy

We strongly recommend that your MLID provide the following functionality. (In some instances this manual recommends certain ways of implementing this functionality, but it is up to you to implement this functionality in any way you choose.)

We strongly recommend that your MLID support reentrancy. When you link your LAN driver for the NetWare 3 and later environments, you can declare your driver reentrant. This allows the operating system to use a single code image of the MLID to run multiple LAN adapters (of the same type) or to run multiple frame types (logical boards) on the same LAN adapter. A non-reentrant driver would require the operating system to load an additional code image of the driver each time it used another LAN adapter or supported another logical board.

To illustrate the advantage of reentrant code, consider the following example. Suppose you wanted to configure a server to drive two Novell NTR2000 cards. You would enter the following commands at the server console:

```
load ntr2000
load ntr2000
```

If have written reentrant code, the first load actually loads the code image of the driver into the server's memory and then calls the MLID's initialization routine. The second load merely calls the MLID's initialization routine again. If you have not written reentrant code, two copies of the NTR2000 LAN driver would be loaded into memory.

Multiple Frame Support

If the LAN adapter runs on a topology that supports multiple frame types, we strongly recommend that the MLID support all that topology's frame types. We implement multiple frame support by using logical boards.

Multiple Frame Support and Logical Boards

To illustrate how logical boards are used, consider the preceding example of loading an NTR2000 twice. When you enter the load command the second time, you could be indicating one of two things:

- ◆ You want the MLID to run a second LAN adapter
- ◆ You want the MLID to run a second frame type on the LAN adapter that is already loaded

Which ever is the case, your MLID creates a "logical board" in response to this command. (A fuller description of logical boards is provided below.) The operating system does not concern itself with distinguishing between logical boards that have exclusive use of a LAN adapter and logical boards that share the same LAN adapter with other logical boards. Only the MLID makes this distinction.

Multiple Frame Support in Reentrant Code

If you are writing reentrant code, each logical board uses the single code image of the MLID that was loaded into the operating system with the first load command. However, the MLID must maintain a separate adapter data space for each adapter and a separate frame data space for each logical board.

Frame Data Space and Adapter Data Space

Adapter Data Space

To allow the MLID to resolve the ambiguity that arises with the second load command, the operating system asks the following question:

```
Do you want to add another frame type for a previously loaded board?
```

If your response to the operating system's question is no, the MLID must allocate an adapter data space to drive a second adapter. The adapter data space is a structure that contains the hardware-specific information which 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 this 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.

NOTE: The MLID must create an adapter data space for every LAN adapter of the same type that is loaded in the server.

Frame Data Space

Every logical board also has a frame data space associated with it. The frame data space is a structure that contains the frame-specific information the MLID needs to support that frame type. The MLID allocates one 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.

Implementing Multiple Frame Support

Figure 16 illustrates how you might implement multiple frame support in an NE2000 server driver. In order for the server to use the first NE2000 adapter, you would enter:

```
load ne2000
```

In response to this command, the MLID would create logical board 1, which uses Frame Data Space 1 and Adapter Data Space A to run Adapter A. By default, Frame Data Space 1 contains the information necessary to support Ethernet 802.2. (Logical board 0 is reserved for use by the operating system.)

Now suppose you wanted to use a second NE2000 adapter that supported both SNAP and 802.2 frames. To do this you would enter the following command:

```
load ne2000 frame=ETHERNET_SNAP
```

Afterwards, the server would ask the following question:

```
Do you want to add another frame type for a previously loaded  
board?
```

In order to use the second NE2000 adapter, you would need to enter 'n'. This would cause the MLID to create logical board 2, which uses Frame Data Space 2 and Adapter Data Space B to run Adapter B. The `frame=ETHERNET_SNAP` command tells the MLID that Frame Data Space 2 will support Ethernet SNAP.

In order for Adapter B to also support 802.2, you would need to load the NE2000 driver a third time:

```
load ne2000 frame=ETHERNET_802.2
```

This time, however, you would enter 'y' in response to the question:

```
Do you want to add another frame type for a previously loaded  
board?
```

The operating system would then let you indicate which LAN adapter you want to add additional frame support to. If you were to specify Adapter B, the MLID would then create logical board 3, which uses Frame Data Space 3 and Adapter Data Space B to communicate with Adapter B.

Figure 16 Implementation of Multiple Frame Support Using Ethernet

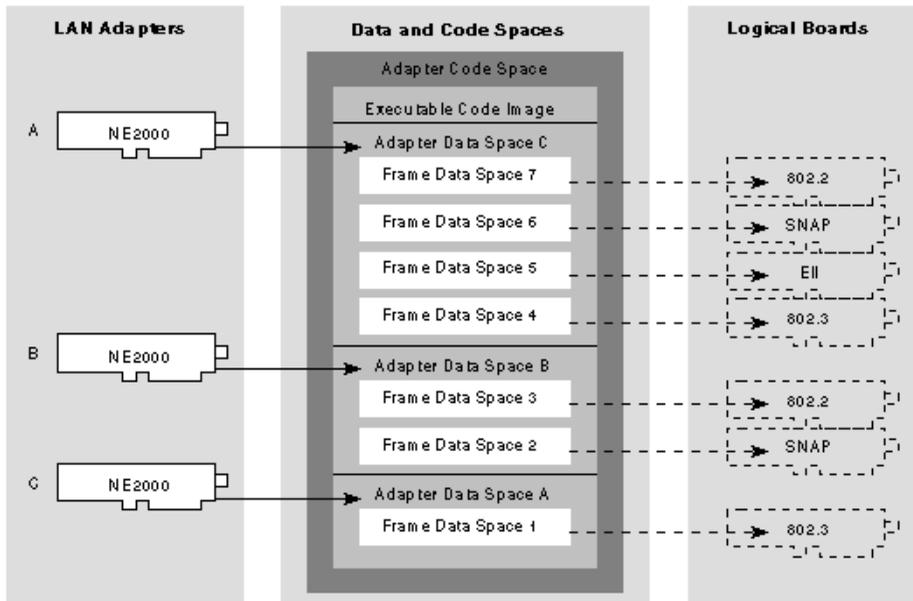
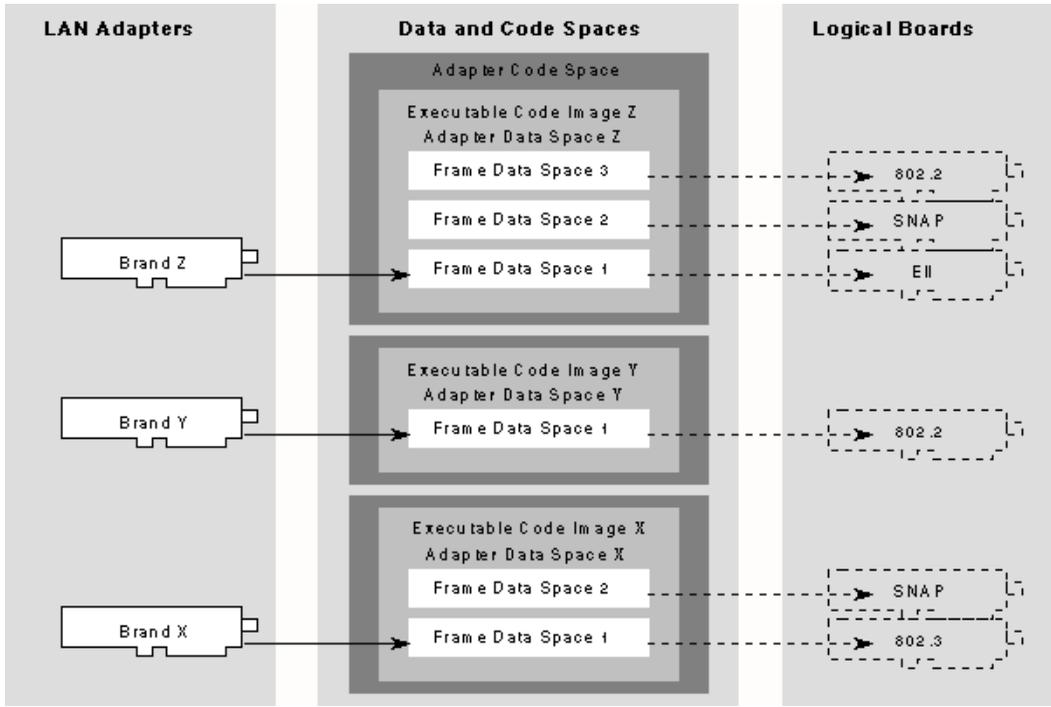


Figure 17 shows that when the boards are not all the same type, each board has its own executable code image and adapter data space.

Figure 17 Implementation of Multiple Boards/Frame Support



Optional Functionality

A NetWare server MLID can support source routing (Token-Ring and FDDI), promiscuous mode, and multicast addressing. We recommend that your MLID support all of these options, if the LAN adapter supports them.

Source Routing Support

The ODI Specification Supplement: Source Routing describes how to add and configure source routing in the MLID.

Promiscuous Mode

When MLIDs operate in promiscuous mode, they pass all packets they receive to the upper layers. This includes bad packets, if possible. Because various monitoring functions operate in promiscuous mode, we strongly recommend that your MLID support promiscuous mode if your adapter is capable of such support. The MLID enables or disables promiscuous mode upon request by

using the *DriverPromiscuousChange* routine described in Chapter 20, "MLID Control Routines".

Multicast Address Support

NOTE: If your LAN adapter is capable of supporting multicast addressing, your MLID must also support it. The *Ctl2_AddMulticastAddress* and *Ctl3_DeleteMulticastAddress* IOCTLs implement multicast support. These control procedures are discussed in more detail in Chapter 20, "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 NE3200 and the NE2000, have different hardware and data transfer characteristics. A thorough understanding of your LAN adapter and LAN topology allows 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 geared 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 Network Bus Interface (NBI) routines (new with this spec). Provide all the information needed to determine the bus type and get the hardware resource information.

14 MLID Data Structures

Chapter 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 OSDATA segment of the MLID.

This chapter describes the structures in the adapter and frame data spaces as well as the Event Control Block. This chapter contains useful reference material for the developer.

Frame Data Space

The ODI specification requires that every MLID has an MLID configuration table as part of the frame data space. The MLID keeps a copy of the configuration table template in the OSDATA segment. The MLID uses the configuration table in the OSDATA 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 (See Chapter 13, "Overview of MLIDs"). Because external processes can also access this table, the ODI specification defines this table's format strictly.

The MLID Configuration Table

The MLID configuration table contains information about the MLID and the LAN adapter's hardware configuration.

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 accordingly. Certain variables in the configuration table will be specific to your MLID. Other variables will be specific to the particular LAN adapter the MLID is running. The following pages show the configuration table format. Asterisks indicate that the field is configurable from the command line at load time. These fields should be set to their default value. Any field that is not used should be set to 0, unless otherwise noted. A description of each field follows the figures.

The MLID provides external processes access to the configuration tables in two ways:

- ◆ The MLID passes the LSL a pointer to the copy of the configuration table the MLID creates during the initialization of each logical board.
- ◆ The MLID provides a control procedure called *CtIO_GetMLIDConfiguration*, which, when called by a protocol stack or some other external process, returns a pointer to the specified logical board's configuration table.

NOTE: All data strings pointed to by the pointers in the configuration table are length-preceded and null-terminated.

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

The following is MLID Configuration Table Sample Source Code

```
DriverConfigTemplate
```

```
DriverConfigTemplate      label  dword
MLIDConfigurationStructure  struc
MLIDCFG_Signature         db              'HardwareDriverMLID',8 dup (")
MLIDCFG_MajorVersion      db 01              ;v1.14
MLIDCFG_MinorVersion      db 14              ;v1.14
MLIDNodeAddress           db 6              dup (?)
MLIDModeFlags             dw 0
MLIDBoardNumber           dw 0
MLIDBoardInstance         dw 0
MLIDMaximumSize           dd 0
MLIDMaxRecvSize           dd 0
MLIDRecvSize              dd 0
MLIDCardName              dd 0
MLIDShortName             dd 0
MLIDFrameType             dd 0
MLIDReserved0             dw 0
```

```

MLIDFrameID                dw 0
MLIDTransportTime          dw 0
MLIDSrcRouting             dd 0
MLIDLineSpeed              dw 0
MLIDLookAheadSize         dw 0
MLIDCFG_SGCount            db 0
MLIDReserved               db 0
MLIDPrioritySup            dw 0
MLIDReserved               dd 0
MLIDMajorVersion           db 0
MLIDMinorVersion           db 0
MLIDFlags                  dw 0
MLIDSendRetries            dw 0
MLIDLink                   dd 0
MLIDSharingFlags           dw 0
MLIDSlot                   dw 0
MLIDIOPortsAndLengths     dw 4                dup (0)
MLIDMemoryDecode0         dd 0
MLIDLength0                dw 0                ;in paragraphs
MLIDMemoryDecode1         dd 0
MLIDLength1                dw 0
MLIDInterrupt              db 2                dup (?)
MLIDDMAUsage               db 2                dup (?)
MLIDResourceTag            dd 0
MLIDConfig                 dd 0
MLIDCommandString          dd 0
MLIDLogicalName            db 18                dup (?)
MLIDLinearMemory0          dd 0                ;NetWare 4 field only
MLIDLinearMemory1          dd 0                ;NetWare 4 field only
MLIDChannelNumber          dw 0                ;NetWare 4 field only
MLIDBusTag                 dd 00 00 00 00        ;NetWare 4 field only
MLIDCfgMajorVersion        db 1                ;NetWare 4 field only
MLIDCfgMinorVersion        db 0                ;NetWare 4 field only
MLIDConfigurationStructure ends

```


Table 35 MLID Configuration Table

Offset	Field Name	Size	Description
0h	MLIDCFG_Signature	26	<p>This field is a mandatory remnant from the NetWare 2 environment. In pre-MLID LAN drivers, this field indicated the start of the configuration table. External entities could search on the string in this field to find the driver's configuration table.</p> <p>This string is now less useful because MLIDs provide a pointer to this table. However, you must still include this field in the configuration table. The string is "HardwareDriverMLID" followed by exactly eight spaces. The MLID must initialize this field to that string.</p>
1Ah	MLIDCFG_MajorVersion	1	<p>Set this field to the major version number of the configuration table. The version is controlled by Novell and is currently v1.14; therefore, 1 is the major version number.</p>
1Bh	MLIDCFG_MinorVersion	1	<p>Set this field to the minor version number of the configuration table. The version is controlled by Novell and is currently v1.14; therefore, 14 is the minor version number.</p>
1Ch	MLIDNodeAddress	6	<p>This field contains the node address of the LAN adapter. This address is in either canonical (LSB mode) or noncanonical (MSB mode), depending upon the topology.</p> <p>Usually, the MLID places the node address it reads from the hardware into this field during the MLID's initialization routine. However, the MLID can also call <i>ParseDriverParameters</i> to prompt a console operator to configure this address at command line.</p> <p>If both bits 14 and 15 of the <i>ModeFlags</i> field (offset 22h) are set to 0, the address in this field is in the physical layer format.</p> <p>If bit 15 of the <i>ModeFlags</i> field is set to 1, the MLID can tell by the state of bit 14 whether this address is canonical or noncanonical.</p> <p><i>Note:</i> Noncanonical mode is valid only for Token-Ring MLIDs. Ethernet and FDDI MLIDs must use canonical addresses. (For more information, refer to ODI Supplement: Canonical and Noncanonical Addressing.)</p>

Offset	Field Name	Size	Description
22h	MLIDModeFlags	2	<p>This field contains flags that the MLID should set using the definitions found in Table 36.</p> <p><i>Note:</i> Unused bits are reserved and should always be set to 0.</p>
24h	MLIDBoardNumber	2	<p>During initialization, the MLID fills this field with the assigned board number that it received when it registered with the LSL during its initialization routine. The MLID should not modify this field.</p> <p>Logical board 0 is used internally in the operating system. Drivers are assigned logical board numbers 1 through 255.</p>
26h	MLIDBoardInstance	2	<p>The MLID sets this field during its initialization routine. If the MLID is driving two adapters, all logical boards associated with the first adapter would have a value of 1, and all the logical boards associated with the second adapter would have a value of 2.</p> <p><i>Note:</i> Each controller on a multi-channel adapter is treated as a separate adapter.</p>
28h	MLIDMaximumSize	4	<p>The value in this field defines the largest possible frame size that the driver/LAN adapter combination can transmit or receive. This value includes all headers.</p> <p>This value cannot exceed the size of the LSL's maximum ECB buffer.</p> <p><i>Token-Ring:</i> Token-Ring MLIDs can send and receive a number of different frame sizes. Therefore, during its initialization routine, a Token-Ring MLID must determine the appropriate frame size and place that value in this field. Token-Ring MLIDs should support 4KB (4096+74+40 = 4210) frame sizes whenever it is possible and practical. The value in this field should not be less than 626 decimal (586 bytes + 18 bytes [source routing] +14 bytes [802.5 header] + 3 bytes [802.2 UI] + 5 bytes [SNAP header]).</p> <p><i>Ethernet:</i> The value in this field should be either 1514 or the LSL's maximum ECB buffer size, whichever is lowest. (See Table 39: Maximum Packet Sizes)</p> <p><i>Note:</i> The MLID sets this field before it calls <i>LSLRegisterMLIDRTag</i>.</p>

Offset	Field Name	Size	Description
2Ch	MLIDMaxRecvSize	4	The MLID sets this field after calling <i>LSLRegisterMLIDRTag</i> . The MLID subtracts the size of the smallest media header(s) from the value in the <i>MLIDMaximumSize</i> field. For example, the Ethernet Media module sets this field to 1500 decimal (1514 bytes - 14 bytes [MAC header] = 1500) if the MLID were running the Ethernet_II frame type; the Token-Ring media module which would set this to MLIDMaximumSize - 14 bytes [MAC] - 3 bytes [802.2 UI] if the MLID's frame type were Token-Ring.
30h	MLIDRecvSize	4	The MLID sets this field after calling <i>LSLRegisterMLIDRTag</i> . The MLID subtracts the length of the largest media header(s) from the <i>MLIDMaximumSize</i> field. For example, the Token-Ring 802.2 Media module would set this field to MLIDMaximumSize- 14 bytes [MAC header] 4 bytes [802.2 Type II header]. Because source routing is a possibility, you should subtract the largest source routing header, which is 30 bytes. (Transparent source routing bridging requires a header of 30 bytes.)
34h	MLIDCardName	4	The MLID sets this field to point to a byte-length preceded, null-terminated, ASCII string that is identical to the description string in the linker definition file (see Appendix B, "Assembling and Linking NLMs"). For example: 14, "NetWare NE2000", 0
38h	MLIDShortName	4	The MLID must set this field to point to a byte-length preceded, null-terminated, ASCII string that describes the adapter in eight bytes or less. For example: 6, "NE2000", 0 The content of this string is arbitrary, but it is usually the name of your .LAN file.
3Ch	MLIDFrameType	2	The MLID sets this field during its initialization routine. This field contains a far pointer to a length-preceded, 0-terminated string describing the frame and media type the MLID uses. (See ODI Supplement: Frame Types and Protocol IDs.)
40h	MLIDReserved0	2	The MLID sets this field to 0.

Offset	Field Name	Size	Description
42h	MLIDFrameID	2	The MLID sets this field during its initialization routine. This field contains the frame type ID number. (See the ODI Specification Supplement: <i>Frame Types and Protocol IDs.</i>)
44h	MLIDTransportTime	2	This field indicates the time (in ticks) it takes the LAN adapter to transmit a 576-byte packet. Most MLIDs set this field to 1. This field cannot be set to 0. If the MLID is for a slow asynchronous line, the value of this field should be set according to a representative time.
46h	MLIDSrcRouting	4	If the LAN adapter does not support generic token-ring source routing, the MLID sets this field to 0. If the LAN adapter does support source routing, the MLID initializes this field to the address of a routine which simply does a RET. Later, the ROUTE.NLM utility replaces this address. (For more information on source routing, see the ODI Specification Supplement: Source Routing.)

Offset	Field Name	Size	Description
4Ah	MLIDLineSpeed	2	<p>This field holds the data rate used by the LAN adapter's medium (usually specified in megabits per second). The MLID sets this field to an appropriate value.</p> <p>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.</p> <p>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).</p> <p>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:</p> <p>Ethernet 10 Mbps 000Ah Token-Ring 4 Mbps 0004h Token-Ring 16 Mbps 0010h RX-Net 2,500 Kbps 89C4h FDDI 100 Mbps 0064h ISDN 64 Kbps 8040h</p>
4Ch	MLIDLookAheadSize	2	<p>The MLID sets this field to the amount of data required by the protocol stacks to preview received packets. The MLID can change this size dynamically. This size can be a maximum value of 128 bytes. This field defaults to 18 bytes.</p> <p><i>Note:</i> This value cannot be decreased below 18 bytes.</p>
4Eh	MLIDCFG_SGCount	1	<p>The maximum number of scatter/gather elements the adapter is capable of handling. The MLID sets this variable. The minimum value is 2 (1 for the MAC header and 1 for data). The maximum value is 17 (1 for the MAC header and 16 for data).</p>
4Fh	MLIDReserved	1	<p>This field is used by the operating system. The MLID must set this field to 0.</p>

Offset	Field Name	Size	Description
50h	MLIDPrioritySup	2	This field contains the number of send priorities the MLID can support.
54h	MLIDReserved	4	This field is used by the operating system. The MLID must set this field to 0.
56h	MLIDMajorVersion	1	This field contains the major version number of the MLID. The number must match the version number specified with the "version" keyword in the linker definition file (see Appendix B).
57h	MLIDMinorVersion	1	This field contains the minor version number of the MLID. The number must match the version number specified with the "version" keyword in the linker definition file (see Appendix B).
58h	MLIDFlags	2	See the MLIDFlags bit map in Table 36 .
5Ah	MLIDSendRetries	2	Set this field to a value indicating the number of times the MLID should retry sending a packet before aborting the transmission. This retry count can be any value, but it might be overwritten by a value entered on the server console at load time.
5Ch	MLIDLink	4	The operating system uses this field. The MLID's initialization routine passes a pointer to this field when calling <i>ParseDriverParameters</i> and <i>RegisterHardwareOptions</i> . MLIDs should not change this field.
60h	MLIDSharingFlags	2	This field informs the system of the hardware resources that an MLID/LAN adapter can share with other MLID/LAN adapters. The MLID uses these flags to enable shareable interrupts and memory. See the MLIDSharingFlags bit map in Table 38 .

Offset	Field Name	Size	Description
62h	MLIDSlot	2	<p>If the MLID is for an ISA board, this field is not used and should be set to -1.</p> <p>If the MLID is for a PCI, MicroChannel, or EISA type LAN adapter, the MLID's initialization routine typically scans the machine's slots and prepares a list that the MLID passes to the <i>ParseDriverParameters</i> operating system routine. <i>ParseDriverParameters</i> then fills in this slot using either the AUTOEXEC.NCF file, or input from the command line or the user.</p> <p>When using NBI, set this field to the HIN value.</p>
64h	MLIDIOPortsAndLengths	8	This field contains the I/O port information as described below. Set this field to 0 if it is not used.
(64h)	(MLIDIOPort0)	(2)	Primary base I/O port.
(66h)	(MLIDIORange0)	(2)	Number of I/O ports starting at MLIDIOPort0.
(68h)	(MLIDIOPortI)	(2)	Secondary base I/O port.
(6Ah)	(MLIDIORangeI)	(2)	Number of I/O ports starting at MLIDIOPortI.
6Ch	MLIDMemoryDecode0	4	This field contains the absolute primary memory address used by the LAN adapter. If not used, set this field to 0. (See the note in the <i>MLIDLinearMemory0</i> description at offset 9Ah.)
70h	MLIDMemoryLength0	2	This field contains the amount of memory (in paragraphs) that the LAN adapter uses, starting at <i>MLIDMemoryDecode0</i> . If not used, set this field 0.
72h	MLIDMemoryDecode1	4	This field contains the absolute secondary memory address used by the LAN adapter. If not used, set this field to 0. (See the note in the <i>MLIDLinearMemory1</i> description at offset 9Eh.)
76h	MLIDMemoryLength1	2	This field contains the amount of memory in paragraphs starting at <i>MLIDMemoryDecode1</i> . If not used, set this field to 0.
78h	MLIDInterrupt	2	This field contains interrupt information as described below. (Set to FFh if this field is not used.)
(78h)	(MLIDInterrupt0)	(1)	Primary interrupt vector number.
(79h)	(MLIDInterruptI)	(1)	Secondary interrupt vector number.

Offset	Field Name	Size	Description
7Ah	MLIDDMAUsage	2	This field contains DMA channel information as described below. (Set to FFh if this field is not used.)
(7Ah)	(MLIDDMAUsage0)	(1)	Primary DMA channel.
(7Bh)	(MLIDDMAUsage1)	(1)	Secondary DMA channel.
7Ch	MLIDResourceTag	4	This field contains a pointer to the MLID's IOResourceTag.
80h	MLIDConfig	4	This field contains a pointer to the MLID's configuration table. This field is used only by operating system; the MLID or protocol stack should not modify it.
84h	MLIDCommandString	4	<p>If the INSTALL utility needs to replace the default command line in the AUTOEXEC.NCF file, it uses the null-terminated string accessed through this field. This field contains a pointer to a linked list:</p> <pre> LANCommandLineInfo struct CommandLineNext dd ? ;Set to 0 if another string does not follow CommandLineStringPtr dd ? ;Points to a command line string LANCommandLineInfo end </pre> <p>Bits 9 and 10 of the MLIDSharingFlags bit map are used in conjunction with this field.</p>
88h	MLIDLogicalName	18	<p>MLID should not use this field. It contains the logical name of the MLID, if the MLID has a logical name. The MLID is usually given this logical name at load time. For example:</p> <pre>load NE2000 NAME=" "</pre>
9Ah	MLIDLinearMemory0	4	<p>The operating system fills in this field with the linear address of <i>MLIDMemoryDecode0</i> during <i>RegisterHardwareOptions</i>.</p> <p>Do not use the operating system conversion routines to convert <i>MLIDMemoryDecode0</i> to the logical address.</p>

Offset	Field Name	Size	Description
9Eh	MLIDLinearMemory1	4	The operating system fills in this field with the linear address of <i>MLIDMemoryDecode1</i> during <i>RegisterHardwareOptions</i> . Do not use the operating system conversion routines to convert <i>MLIDMemoryDecode1</i> to the logical address.
A2h	MLIDChannelNumber	2	This field is used for multichannel adapters. It holds the channel number of the Network Interface Circuit to use. The channel number can be specified when an MLID is loaded using the "channel=#" keyword (where # is any value greater than 0). Set this field to 0 if multichannels are not used.
A4h	MLIDBusTag	4	Pointer to an architecture-dependent value which specifies the bus on which the adapter is found. Set this field to 0 before calling <i>ParseDriverParameters</i> . The MLID must enter the value returned by the NBI function, <i>SearchAdapter</i> , in this field.
ABh	MLIDIOConfigMinorVer	1	The current major revision level of the IO_CONFIG structure (the bottom half of CONFIG_TABLE structure). The MLID sets this variable to 1.
A9h	MLIDIOConfigMinorVer	1	The current minor revision level of the IO_CONFIG structure (the bottom half of CONFIG_TABLE structure). The MLID sets this variable to 1.

Configuration Table Flags

The following table contains the bit map and bit descriptions for the Configuration Table MLIDModeFlags field.

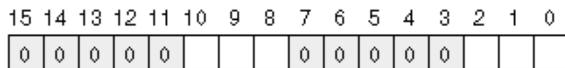
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			0	0	0	0	0	0		0					0

Default Values

Table 36 Configuration Table MLIDModeFlags Field Bit Map (Offset 22h)

Bit #	Name	Map Description
0	RealDriverBit	This bit is a remnant from previous specifications. The MLID now sets this bit to 1.
1	Reserved	This bit has been retired and must be set to 0.
2	Reserved	This bit has been retired and must be set to 0.
3	MulticastBit	The MLID sets this bit if it supports multicast addressing. The MLID must support multicast addressing, if the hardware supports it.
4	PointToPointBit	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	Reserved	Set to 0.
6	RawSend	The MLID sets this bit to indicate that it supports raw sends. Refer to Chapter 17, "The MLID Packet Transmission Routine" for information on raw sends. (RX-Net does not support raw sends.) This bit defaults to 0.
7	Reserved	Set to 0.
8	SMPBit	Set this bit if the MLID is written to be SMP aware.
9	Reserved	Set to 0.
10	Reserved	Set to 0.
11	Reserved	Set to 0.
12	Reserved	Set to 0.
13	PromiscuousMode	Bit The MLID must set this bit if it supports Promiscuous Mode.

Bit #	Name	Map Description
15	CanonicalBits	The MLID sets these bits to indicate whether the <i>MLIDNodeAddress</i> field of the configuration table contains a canonical or a noncanonical address.
14		<p>15 is always set to 1 for NetWare operating system versions later than 3.11. This bit indicates whether the node address format is configurable.</p> <p>14 indicates whether the configuration table <i>MLIDNodeAddress</i> field contains the node address in canonical or noncanonical form. The state of bit 14 is defined only when bit 15 is set.</p> <p>Bit 15 and bit 14 combinations are as follows:</p> <ul style="list-style-type: none"> ◆ 00 = MLIDNodeAddress format is unspecified. The node address is assumed to be in the physical layer's native format. ◆ 01 = This is an illegal value and must not occur. ◆ 10 = MLIDNodeAddress is canonical. ◆ 11 = MLIDNodeAddress is noncanonical. (See the definition of the <i>MLIDNodeAddress</i> field in the Configuration Table and ODI Supplement: Canonical and Noncanonical Addressing.)

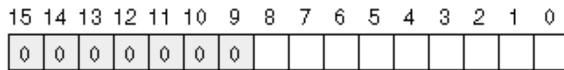


Default Values

Table 37 MLIDFlags Bit Map (Offset 58h)

Bit #	Description
0	Reserved. Set to 0.
1	Reserved. Set to 0.
2	Reserved. Set to 0.
3	Reserved. Set to 0.
4	Reserved. Set to 0.
5	Reserved. Set to 0.
6	Reserved. Set to 0.

Bit #	Description
7	Reserved. Set to 0.
8	Set to 1 if the MLID supports HUB Management.
10	These bits indicate different support mechanisms for multicast filtering and multicast address format. These bits are only valid if bit 3 of the MLIDModeFlags is set, indicating that the MLID supports multicast addressing. The MLID sets bit 10 if it has specialized adapter hardware (such as hardware that utilizes CAM memory). If this bit is set, <i>DriverMulticastChange</i> receives a pointer to the multicast address table and the number of addresses in the table.
9	
<p><i>Note:</i> 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. 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 reset 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"> ◆ 00 = The format of the multicast address defaults to that of the topology: Ethernet => Group Addressing Token-Ring => Functional Addressing PCN2 => Functional Addressing FDDI =>Group Addressing ◆ 01 = Illegal value and must not occur ◆ 10 = A specialized adapter supports group addressing, but the MLID should filter the addresses ◆ 11 = A specialized adapter supports group addressing, and the MLID is not required to filter the addresses 	
11	Reserved. Set to 0.
12	Reserved. Set to 0.
13	Reserved. Set to 0.
14	Reserved. Set to 0.
15	Reserved. Set to 0.



Default Values

Table 38 MLIDSharingFlags Bit Map (Offset 60h)

Bit #	Description
0	Set to 1 if the LAN logical board is currently shut down.
1	Set to 1 if the LAN adapter can share I/O port 0.
2	Set to 1 if the LAN adapter can share I/O port 1.
3	Set to 1 if the LAN adapter can share memory range 0.
4	Set to 1 if the LAN adapter can share memory range 1.
5	Set to 1 if the LAN adapter can share interrupt 0.
6	Set to 1 if the LAN adapter can share interrupt 1.
7	Set to 1 if the LAN adapter can share DMA channel 0.
8	Set to 1 if the LAN adapter can share DMA channel 1.
9	The MLID sets this bit to indicate to the INSTALL utility that it has its own command line information to place in the AUTOEXEC.NCF file. The INSTALL utility may use any default information and the string pointed to by <i>MLIDCommandString</i> . (Default information is the frame type and the values chosen from the <i>AdapterOptionsDefinitionStructure</i> for each option. The <i>NeedsBitMap</i> determines what options are chosen. See the <i>ParseDriverParameters</i> description in Appendix A, "Operating System Support Routines" for a description of these structures.
10	The MLID sets this bit to prevent default information from being placed in the AUTOEXEC.NCF file option information entered on the command line. If this bit is 0 (the default value), and bit 9 is set to 1, the INSTALL utility could use the string pointed to by <i>MLIDCommandString</i> to add to the default information. Setting this bit overrides the setting of bit 9.
11	Reserved. Set to 0.
12	Reserved. Set to 0.
13	Reserved. Set to 0.

Bit #	Description
14	Reserved. Set to 0.
15	Reserved. Set to 0.

NOTE: MLIDs typically set bit 9, copy the command line to a buffer point to by the configuration table's *MLIDCommandString* field, and clear bit 10. Bit 10 allows the install utility to make a nearly accurate entry into the AUTOEXEC.NCF file.

Deriving the Maximum Packet Size

During the MLID's initialization routine, the MLID sets the configuration table *MLIDMaximumSize* field to equal either the LSL's maximum ECB buffer size or the topology's maximum size, whichever is smaller. (In MLIDs written to the NetWare 4 operating system, this size defaults to 2KB but can be changed in the STARTUP.NCF file to a maximum of 24KB). The MLID also sets the configuration table *MLIDMaxRecvSize* and *MLIDRecvSize* fields. After the *LSLRegisterMLIDRTag* routine returns, MLIDs for intelligent adapters pass the maximum size to the hardware if it's required. The following table shows how these values are determined.

All the values in the table below are derived from the value returned in ECX when the MLID calls *LSLRegisterMLIDRTag*. This value in ECX is the maximum ECB buffer size.

Table 39 Maximum Packet Sizes

Frame Type	MLIDMaximumSize	MLIDMaxRecvSize	MLIDRecvSize
RX-Net	Maximum ECB buffer size	Maximum ECB buffer size	Maximum ECB buffer size
Ethernet 802.3	Maximum ECB buffer size or 1,514 (whichever is less)	MLIDMaximumSize - 14	MLIDMaximumSize - 14
Ethernet 802.2	Maximum ECB buffer size or 1,514 (whichever is less)	MLIDMaximumSize- 17	MLIDMaximumSize- 18
Ethernet II	Maximum ECB buffer size or 1,514 (whichever is less)	MLIDMaximumSize- 14	MLIDMaximumSize- 14
Ethernet SNAP	Maximum ECB buffer size or 1,514 (whichever is less)	MLIDMaximumSize - 22	MLIDMaximumSize - 22

Frame Type	MLIDMaximumSize	MLIDMaxRecvSize	MLIDRecvSize
Token-Ring 802.2	Maximum ECB buffer size or the maximum size the adapter can handle (whichever is less)	MLIDMaximumSize - 17	MLIDMaximumSize - 48
Token-Ring SNAP	Maximum ECB buffer size or the maximum size the adapter can handle (whichever is less)	MLIDMaximumSize - 22	MLIDMaximumSize - 52
FDDI 802.2	Maximum ECB buffer size or 4,491 (whichever is less)	MLIDMaximumSize - 16	MLIDMaximumSize - 47
FDDI SNAP	Maximum ECB buffer size or 4,491 (whichever is less)	MLIDMaximumSize - 21	MLIDMaximumSize - 51

Example of Deriving Maximum Packet Size

If the maximum ECB buffer size equals 8,192 bytes and the Token-Ring adapter can handle 4,096 bytes, then the Token-Ring 802.2 values are calculated as follows:

MLIDMaximumSize = 4,096

MLIDMaxRecvSize

(The maximum packet size minus the headers if the source routing header is not included.)

= 4,096 - MAC header (14)
 - 802.2 Type I LLC header (3)
 = 4,079

MLIDRecvSize

(The maximum packet size minus the headers if the source routing header is included.)

= 4,096 - MAC header (14)
 - 802.2 Type II LLC header (4)
 - Source Routing header (30)
 = 4,048

Adapter Data Space

The MLID must allocate and initialize a structure called *DriverAdapterDataSpaceTemplate*. This structure must contain the data that

is specific to a particular LAN adapter. You must determine what hardware-specific fields the MLID needs in this structure in order to drive its particular LAN adapter. But keep in mind that this structure must also contain the MLID statistics table.

MLID Statistics Table

When the MLID's initialization routine calls *RegisterHardwareOptions*, the MLID allocates the adapter data space and creates a copy of the MLID's template in this area. The MLID allocates one adapter data space for each LAN adapter, regardless of the number of logical boards (frame types) it supports.

The statistics table contains various diagnostic counters. All statistics counters shown must be present in the table; however, the MLID is only required to support those counters marked "mandatory." These counters can be grouped into the following categories.

- ◆ Generic Statistics Counters
 - ◆ Standard Counters
 - ◆ Topology-specific Counters
- ◆ Custom Statistics Counters

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

MLID Statistics Table Sample Source Code

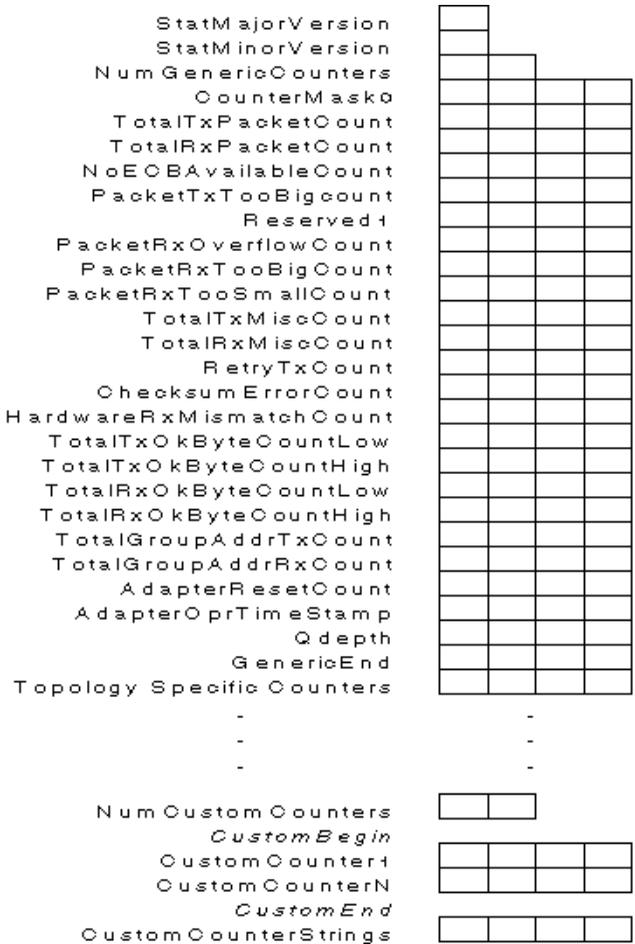
```
DriverAdapterDataSpace      struc
[*** Hardware Specific Variables ***]
    DriverStatisticsTable    db    0 dup (?)      ; (Label)
    StatMajorVersion         db    3
    StatMinorVersion         db    0
    NumGenericCounters       dw    (GenericEnd - GenericBegin) / 4
    CounterMask0             dd    1111 1011 0000 1111 1110 1111 1111 1111
    GenericBegin             db    0 dup (?)      ; (Label)
    TotalTxPacketCount       dd    0              ; (mandatory)
    TotalRxPacketCount       dd    0              ; (mandatory)
    NoECBAvailableCount     dd    0              ; (mandatory)
    PacketTxTooBigCount      dd    0              ; (mandatory)
    Reserved1                dd    0              ; (reserved)
    PacketRxOverflowCount    dd    0              ; (optional)
    PacketRxTooBigCount      dd    0              ; (mandatory)
    PacketRxTooSmallCount    dd    0              ; (optional)
```

```

TotalTxMiscCount      dd  0          ; (mandatory)
TotalRxMiscCount      dd  0          ; (mandatory)
RetryTxCount          dd  0          ; (optional)
ChecksumErrorCount    dd  0          ; (optional)
HardwareRxMismatchCount dd  0          ; (optional)
TotalTxOKByteCountLow dd  0          ; (mandatory)
TotalTxOKByteCountHigh dd  0          ; (mandatory)
TotalRxOKByteCountLow dd  0          ; (mandatory)
TotalRxOKByteCountHigh dd  0          ; (mandatory)
TotalGroupAddrTxCount dd  0          ; (mandatory)
TotalGroupAddrRxCount dd  0          ; (mandatory)
AdapterResetCount     dd  0          ; (mandatory)
AdapterOprTimeStamp   dd  0          ; (mandatory)
QDepth                dd  0          ; (mandatory)
[*** Topology-specific Statistics Counters ***]
GenericEnd            db  0 dup (?)   ; (Label)
NumCustomCounters     dw  (CustomEnd - CustomBegin) / 4
CustomBegin           db  0 dup (?)   ; (Label)
CustomCounter1        dd  0
CustomCounterN        dd  0
CustomEnd              db  0 dup (?)   ; (Label)
CustomCounterStrings  dd  offset CustomStrings
DriverAdapterDataSpace Ends
DriverAdapterDataSpaceTemplate DriverAdapterDataSpace

```

Figure 19 Graphic Representation of the MLID Statistics Table



MLID Statistics Table Field Descriptions

The following table describes the MLID statistics table fields and how they should be initialized.

Table 40 MLID Statistics Table Field Descriptions

Offset	Name	Size (bytes)	Description
00h	MajorVersion	1	This field contains the major version number of the statistics table. The version number is controlled by Novell and is currently v3.00; therefore, 3 is the major version number.
01h	StatMinorVersion	1	This field contains the minor version number of the statistics table. The version number is controlled by Novell and is currently v3.00; therefore, 00 is the minor version number.
02h	NumGenericCounters	2	This field contains the total number of generic counters (standard and topology specific counters) present in the statistics table (but not necessarily supported). This number should also include any additional counter masks used except <i>CounterMask0</i> . (See the next field description for more information on counter masks.)
04h	CounterMask0	4	<p>This field contains a bit mask indicating which counters of the first 32 standard and topology-specific portions of the statistics table are implemented in the driver. If the bit is 0 the counter is supported. (See the bit map definition following this table.)</p> <p>If the MLID requires more than 32 standard and topology-specific counters (as with Token-Ring), a second mask (<i>CounterMask1</i>) is placed after the 32nd counter at offset 88h to indicate the status of the next set of 32 counters.</p>
08h	TotalTxPacketCount	4	The MLID increments this counter whenever a packet is successfully transmitted by the adapter.
0Ch	TotalRxPacketCount	4	The MLID increments this counter whenever a packet is successfully received by the adapter.
10h	NoECBAvailableCount	4	The MLID increments this counter if it cannot obtain a receive ECB for a received packet.
14h	PacketTxTooBigCount	4	The MLID increments this counter whenever a packet is too big for the adapter to transmit.
18h	Reserved1	4	This field is reserved, but should be initialized to 0.

Offset	Name	Size (bytes)	Description
lCh	PacketRxOverflowCount	4	The MLID uses this counter to indicate the number of times the adapter's receive buffers overflowed causing subsequent incoming packets to be discarded.
20h	PacketRxTooBigCount	4	The MLID increments this counter in two situations: 1. A packet is received that is too large for the preallocated receive buffer(s) the host provided. 2. A packet is received that is too large for topology definitions.
24h	PacketRxTooSmallCount	4	Some MLIDs increment this counter if a packet is received that is too small for media definitions. Currently, only the RX-Net MLID maintains this counter.
28h	TotalTxMiscCount	4	The MLID increments this counter if a fatal transmit error occurs and no other appropriate standard counter exists in the generic portion of the statistics table. The MLID can also increment a topology-specific or custom counter for this event.
2Ch	TotalRxMiscCount	4	The MLID must increment this counter if a fatal receive error occurs and no other appropriate standard counter exists in the generic portion of the statistics table. The MLID can also increment a topology-specific or custom counter for this event.
30h	RetryTxCount	4	The MLID can use this counter to indicate the number of times packet transmissions were retried due to failure.
34h	ChecksumErrorCount	4	The MLID can use this counter to indicate the number of times it receives a packet with corrupt data due to CRC errors, etc.
38h	HardwareRxMismatchCount	4	Some MLIDs increment this counter when they receive a packet that does not pass length consistency checks. Currently, only the Ethernet MLID maintains this counter.
3Ch	TotalTxOKByteCountLow	4	This field contains the number of bytes, including low-level headers that the MLID successfully transmitted.
40h	TotalTxOKByteCountHigh	4	This field contains the upper 32-bits of the TotalTxOkByteCount counter.

Offset	Name	Size (bytes)	Description
44h	TotalRxOKByteCountLow	4	This field contains the number of bytes, including low-level headers that the MLID successfully received.
48h	TotalRxOKByteCountHigh	4	This field contains the upper 32 bits of the TotalRxOKByteCount counter.
4Ch	TotalGroupAddrTxCount	4	This field contains the number of packets the MLID transmitted with a group destination address.
50h	TotalGroupAddrRxCount	4	This field contains the number of packets the MLID received with a group destination address.
54h	AdapterResetCount	4	The MLID increments this counter to reflect the number of times the LAN adapter was reset because of internal failure or because of other calls to the MLID's reset routine.
58h	AdapterOprTimeStamp	4	This field contains a time stamp indicating when the LAN adapter last changed operational state (load, shutdown, reset, etc.).
5Ch	QDepth	4	This field reflects the number of transmit ECBs the MLID has queued in the LAN adapter.
60h	(Topology-specific Counters)	4 each	See the "Topology-specific Counters" section.
??	NumCustomCounters	2	This field contains the number of custom counters defined by the MLID. For example, an MLID could create a counter to keep track of the number of fatal retransmissions. Each custom counter must have an associated string that can be accessed through the CustomStrings area (defined in the <i>CustomCounterStrings</i> field).
??	CustomCounter1	4	These fields contain custom counters that the each MLID can configure for its specific needs or for the needs of the LAN adapter.

Offset	Name	Size (bytes)	Description
??	CustomCounterStrings	4	<p>This field contains a pointer to the CustomStrings area. The first word of the CustomStrings area contains the size of the area in bytes. Each string in this area must be null-terminated, and the table of strings is terminated by two nulls. The string order must correspond with the custom counters.</p> <p>CustomStrings label dword</p> <p>CustomStrSize dw</p> <p>(CustomStrEnd - CustomStrings)</p> <p>db 'Custom String 1', 0</p> <p>db 'Custom String 2', 0</p> <p>db 'Custom String 3', 0</p> <p>.</p> <p>.</p> <p>.</p> <p>db 'Custom String N', 0</p> <p>db 0,0</p> <p>CustomStrEnd db 0 dup (?)</p>

CounterMask Bit Maps

The *CounterMask0* field of the statistics table is a bit mask indicating which counters in the standard and topology-specific portions of the table are supported by the MLID. If more than 32 standard and topology-specific counters exist (as with Token-Ring and FDDI), a second bit mask (*CounterMask1*) is placed after the 32nd counter at offset 88h to indicate the status of the next set of 32 counters. This continues every 32 counters as more statistics are added to the table in the future.

The table below indicates whether a counter is optional or mandatory. The MLID maintains most of the standard counters and optionally supports several others. If the MLID does not support a counter, the MLID must set the corresponding bit in the *CounterMask0* field 1. A bit value of 0 in the *CounterMask0* field means the corresponding counter is supported.

NOTE: The bits marked as mandatory in [Table 41](#) must be set to zero (0) and supported by the MLID. Bits 9 and below correspond to the topology-specific counters defined in section 14.3.4 - Topology-specific Counters.

Figure 20 CounterMask0 Bit Map

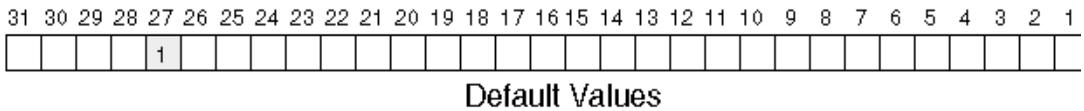


Table 41 Supported Counters

Bit	Counter Name	Support
10	QDepth	Mandatory
11	AdapterOprTimeStamp	Mandatory
12	AdapterResetCount	Mandatory
13	TotalGroupAddrRxCount	Mandatory
14	TotalGroupAddrTxCount	Mandatory
15	TotalRxOKByteCountHigh	Mandatory
16	TotalRxOKByteCountLow	Mandatory
17	TotalTxOKByteCountHigh	Mandatory
18	TotalTxOKByteCountLow	Mandatory
19	HardwareRxMismatchCount	Optional
20	ChecksumErrorCount	Optional
21	RetryTxCount	Optional
22	TotalRxMiscCount	Mandatory
23	TotalTxMiscCount	Mandatory
24	PacketRxTooSmallCount	Optional

Table 42 Token-Ring Topology-specific Counter Descriptions

Offset	Name	Sizes (bytes)	Description
60h	ACErrorCounter	4	The MLID increments this counter when a station receives an AMP or SMP frame with A=C=0, and then receives another SMP frame with A=C=0 without first receiving an AMP frame.
64h	AbortDelimiterCounter	4	The MLID increments this counter when a station transmits an abort delimiter while transmitting.
68h	BurstErrorCounter	4	The MLID increments this counter when a station detects the absence of transitions for five half-bit times (burst-five error). Note that only one station detects a burst-five error, because the first station to detect it converts it to a burst-four.
6Ch	FrameCopiedErrorCounter	4	The MLID increments this counter when a station recognizes a frame addressed to its specific address and detects that the FS field A bits are set to 1, which indicates a possible line hit or duplicate address.
70h	FrequencyErrorCounter	4	The MLID increments this counter when the frequency of the incoming signal differs from the expected frequency by more than that specified in Section 7 (IEEE Std 802.5-1989).
74h	InternalErrorCounter	4	The MLID increments this counter when a station recognizes a recoverable internal error. The MLID can use this counter to detect a station in marginal operating condition.

Offset	Name	Sizes (bytes)	Description
78h	LastRingStatus	4	<p>This value contains the last Ring Status reported by the adapter. This field has the following bit definitions:</p> <ul style="list-style-type: none"> bit 0-4 reserved bit 5 ring recovery bit 6 single station bit 7 counter overflow bit 8 remove received bit 9 reserved bit 10 auto-removal error 1 bit 11 lobe wire fault bit 12 transmit beacon bit 13 soft error bit 14 hard error bit 15 signal loss
7Ch	LineErrorCounter	4	<p>The MLID increments this counter when a station copies or repeats a frame or token, the E bit is 0 in the frame or token, and one of the following conditions exist:</p> <ul style="list-style-type: none"> ♦ The frame or token contains a nondata bit (J or K) between the SD and the ED of the frame or token. ♦ The frame contains an FCS error. <p>The first station detecting a line error increments its appropriate error counter and sets E equal to 1 in the ED of the frame. This prevents other stations from logging the error and isolates the source of the disturbance to proper error domain.</p>
80h	LostFrameCounter	4	<p>The MLID increments this counter when a station is transmitting and its TRR timer expires. This counts how often frames transmitted by a particular station fail to return to it (causing the active monitor to issue a new token).</p>

Offset	Name	Sizes (bytes)	Description
84h	TokenErrorCounter	4	The MLID increments this counter when a station acting as the active monitor recognizes an error condition that needs a token transmitted. This occurs when the TVX timer expires.
88h	CounterMask1	4	This field is a bit mask indicating the status of the next set of counters. The most significant bit corresponds to UpstreamNodeHighDword . If this bit is 0, the MLID supports this counter.
90h	UpstreamNodeHighDword	4	This field contains the high 4 bytes of the 6-byte Upstream Neighbor Node Address.
90h	UpstreamNodeLowWord	4	This field contains the lower 2 bytes of the 6-byte Upstream Neighbor Node Address.
94h	LastRingID	4	This field contains the value of the local ring.
98h	LastBeaconType	4	This field contains the value of the last beacon type.

Ethernet Topology-specific Counters

TxOKSingleCollisionsCount	dd 0	;60h Mandatory
TxOKMultipleCollisionsCount	dd 0	;64h Mandatory
TxOKButDeferred	dd 0	;68h Mandatory
TxAbortLateCollision	dd 0	;6Ch Mandatory
TxAbortExcessCollision	dd 0	;70h Mandatory
TxAbortCarrierSense	dd 0	;74h Mandatory
TxAbortExcessiveDeferral	dd 0	;78h Mandatory
RxAbortFrameAlignment	dd 0	;7Ch Mandatory

Table 43 Ethernet Topology-specific Counter Descriptions

Offset	Name	Size (bytes)	Description
60h	TxOKSingleCollisionsCount	4	This field contains the number of frames involved in a single collision that are subsequently transmitted successfully. The MLID increments this counter when the result of a transmission is reported as transmitOK and the attempt value is 2.

Offset	Name	Size (bytes)	Description
64h	TxOKMultipleCollisionsCount	4	This field contains the number of frames involved in more than one collision that are subsequently transmitted successfully. The MLID increments this counter when the result of a transmission is reported as transmitOK and the attempt value is greater than 2 and less than or equal to the attemptLimit.
68h	TxOKButDeferred	4	The MLID increments this counter for frames whose transmission was delayed on the first attempt because the medium was busy.
6Ch	TxAbortLateCollision	4	This field contains the number of collisions detected later than 512 bit times into the transmitted packet. A late collision is counted both as a collision and as a late collision.
70h	TxAbortExcessCollision	4	This field contains the number of frames not transmitted successfully due to excessive collisions. The MLID increments this counter when the attemptValue equals the attemptLimit during a transmission.
74h	TxAbortCarrierSense	4	This field contains the number of times the carrierSense variable was not asserted or was deasserted during transmission of a frame without collision.
78h	TxAbortExcessiveDeferral	4	This field contains the number of frames deferred for an excessive period of time. Increment this counter only once per LLC transmission.
7Ch	RxAbortFrameAlignment	4	This field contains the number of frames that are not an integral number of octets in length and do not pass the FCS check.

FDDI Topology-specific Counters

FConfigurationState	dd 0 ;60h	Mandatory
FUpstreamNodeHighDword	dd 0 ;64h	Mandatory
FUpstreamNodeLowWord	dd 0 ;68h	Mandatory
FDownstreamNodeHighDword	dd 0 ;6Ch	Mandatory
FDownstreamNodeLowWord	dd 0 ;70h	Mandatory
FFrameErrorCount	dd 0 ;74h	Mandatory
FFramesLostCount	dd 0 ;78h	Mandatory
FRingManagementState	dd 0 ;7Ch	Mandatory

```

FLCTFailureCount      dd 0 ;80h Mandatory
FLemRejectCount       dd 0 ;84h Mandatory
CounterMask1          dd 00111111111111111111111111111111b; **
FLemCount             dd 0 ;8Ch Mandatory
LconnectionState      dd 0 ;90h Mandatory

```

IMPORTANT: CounterMask1 is included when calculating the NumGenericCounters field of the statistics table.

Table 44 FDDI Topology-specific Counter Descriptions

Offset	Name	Size (bytes)	Description
60h	FConfigurationState	4	(ANSI fddiSMTCF-State) The attachment configuration for the station or concentrator: 0 = Isolated 7 = wrap_ab 1 = local_a 8 = wrap_s 2 = local_b 9 = c_wrap_a 3 = local_ab 10 = c_wrap_b 4 = local_s 11 = c_wrap_s 5 = wrap_a 12 = thru 6 = wrap_b
64h	FUpstreamNodeHighDword	8	(ANSI fddiMACUpstreamNbr) FUpstreamNodeLowWord The MAC's upstream neighbor's long individual MAC address (0 if unknown).
6Ch	FDownstreamNodeHighDword	8	(ANSI fddiMACDownstreamNbr) FDownstreamNodeLowWord The MAC's downstream neighbor's long individual MAC address (0 if unknown).
74h	FFrameErrorCount	4	(ANSI fddiMACError-Ct) The number of errored frames this MAC detected that had not been detected by another MAC. (The error evidently occurred between this MAC and the upstream MAC.)
78h	FFramesLostCount	4	(ANSI fddiMACLost-Ct) The number of instances that this MAC detected a format error that caused the frame to be stripped during frame reception.

Offset	Name	Size (bytes)	Description
7Ch	FRingManagementState	4	(ANSI fddiMACRMTD-State) Indicates the current state of the Ring Management state machine. 0 = Isolated 4 = Non_Op_Dup 1 = Non_Op 5 = Ring_Op_Dup 2 = Ring_Op 6 = Directed 3 = Detect 7 = Trace
80h	FLCTFailureCount	4	(ANSI fddiPORTLem-Ct) The count of the consecutive times the link confidence test (LCT) has failed during connection management.
84h	FLemRejectCount	4	(ANSI fddiPortLem-Reject_Ct) The link error monitor count of the times that a link was rejected.
88h	CounterMask1	4	This field is a bit mask indicating the status of the next counters. The most significant bit corresponds to FLemCount. If a bit is 0, the counter is supported.
8Ch	FLemCount	4	(ANSI fddiPORTLem-Ct) The aggregate link error monitor error count (0 only on station power-up).
90h	FConnectionState	4	(ANSI fddiPortPCM-State) The state of this port's PCM state machine. 0 = Off 5 = Signal 1 = Break 6 = Join 2 = Trace 7 = Verify 3 = Connect 8 = Active 4 = Next 9 = Maint

RX-Net Topology-specific Counters

NoResponseToFreeBufferEnquiry	dd 0 ;60h	Mandatory
NetworkReconfigurationCount	dd 0 ;64h	Mandatory
InvalidSplitFlagInPacketFrag	dd 0 ;68h	Mandatory
OrphanPacketFragmentCount	dd 0 ;6Ch	Mandatory
ReceivePacketTimeout	dd 0 ;70h	Mandatory
FreeBufferEnquiryNAKTimeout	dd 0 ;74h	Mandatory
TotalTxPacketFragmentsOK	dd 0 ;78h	Mandatory

Table 45 RX-Net Topology-specific Counter Descriptions

Offset	Name	Size (bytes)	Description
60h	NoResponseToFreeBufferEnquiry	4	The MLID increments this counter each time the receiving node does not respond to FREE BUFFER ENQUIRY.
64h	NetworkReconfigurationCount	4	The MLID increments this counter each time a NETWORK_RECONFIGURATION occurs.
68h	InvalidSplitFlagInPacketFrag	4	The MLID increments this counter each time the Split Flag in the packet fragment is not the expected value. For example, the MLID increments this counter each time it receives packet fragments out of order.
6Ch	OrphanPacketFragmentCount	4	The MLID increments this count each time it receives a packet fragment that is not a part of a previously received packet and, therefore, cannot be appended to a packet.
70h	ReceivePacketTimeout	4	The MLID increments this counter each time a packet times-out waiting for the rest of the packet fragments to arrive.
74h	FreeBufferEnquiryNAKTimeout	4	The MLID increments this counter each time a transmit packet times out waiting for an acknowledgement to a FREE BUFFER ENQUIRY from the receiving node.
78h	TotalTxPacketFragmentsOK	4	This counter contains the number of packet fragments the MLID sent successfully.
7Ch	TotalRxPacketFragmentsOK	4	The counter contains the number of packet fragments the MLID received successfully.

PCN2 Topology-specific Counters

TxOKSingleCollisionsCount	dd 0	;60h Mandatory
TxOKMultipleCollisionsCount	dd 0	;64h Mandatory
TxOKButDeferred	dd 0	;68h Mandatory
TxAbortExcessCollision	dd 0	;6Ch Mandatory
TxAbortCarrierSense	dd 0	;70h Mandatory

Table 46 PCN2 Topology-specific Counter Descriptions

Offset	Name	Size (bytes)	Description
60h	TxOKSingleCollisionsCount	4	This field contains the number of frames that were involved in a single collision and then were subsequently transmitted successfully. The MLID increments this counter when the result of a transmission is reported as transmitOK and the attempt value is 2.
64h	TxOKMultipleCollisionsCount	4	This field contains the number of frames that were involved in more than one collision and then were subsequently transmitted successfully. The MLID increments this counter when the result of a transmission is reported as transmitOK and the attempt value is greater than 2 and less than or equal to the attemptLimit.
68h	TxOKButDeferred	4	The MLID increments this counter for frames whose transmission was delayed on the first attempt because the medium was busy.
6Ch	TxAbortExcessCollision	4	This field contains the number of frames that were not transmitted successfully due to excessive collisions. The MLID increments this counter when the attempts value equals the attemptLimit during a transmission.
70h	TxAbortCarrierSense	4	This field contains the number of times the carrierSense variable was not asserted, or was de-asserted, during transmission of a frame without collision.
74h	RxBadFrameAlignment	4	This field contains the number of frames that are not an integral number of octets in length and do not pass the FCS check.

Event Control Blocks

NetWare shells and operating systems use structures called Event Control Blocks (ECBs) to receive, send, and manage packets. In the NetWare 3 and later operating systems, MLIDs, the LSL, and protocol stacks all use ECBs.

NOTE: Because MLIDs and protocol stacks both use ECBs, this section is duplicated in Chapter 4, "Protocol Stack Data Structures".

When receiving a packet, the MLID obtains an ECB, fills it out, and copies the packet into a buffer that is immediately below the ECB. (Remember that the buffers associated with receive ECBs are contiguous; they have no fragments.) After copying the packet from the LAN adapter, the MLID passes the ECB to the LSL. The LSL then examines the ECB and hands it to the correct protocol stack.

When sending a packet, a protocol stack puts a list of fragment pointers (that describe the packet) in the ECB and passes the ECB to the LSL. The LSL refers to the ECB to determine the destination MLID and then passes the ECB to that MLID. The MLID collects all packet fragments and sends the packet.

Receive Event Control Block

Sample 14-1 shows the sample source code of a receive ECB structure. The asterisks indicate the fields that the MLID must fill in before passing the ECB to the LSL. Figure 14-4 shows a graphic representation of a receive ECB.

Sample 14-1: Receive Event Control Block Sample Source Code

```
Link                dd 0
*BLink             dd 0
Status             dw 0
ESRAddress         dd 0
LogicalID          dw 0
* ProtocolID       db 6 dup (?)
* BoardNumber      dd 0
* ImmediateAddress db 6 dup (?)
* DriverWorkspace  dd 0
  ProtocolWorkspace db 8 dup (?)
* PacketLength     dd 0
  FragmentCount    dd 1
* FragmentOffset   dd 0
* FragmentSize     dd 0
* ;PacketEnvelope  db 0 dup (?)    ; variable length field
                                   ; that contains media
                                   ; headers
* Data              ; variable length field
                                   ; that contains protocol
                                   ; headers and packet
                                   ; information
PacketEnvelope     equ byte        ; ptrFragmentSize+ 4 *
* The MLID must fill in these fields before passing the ECB to the LSL.
```

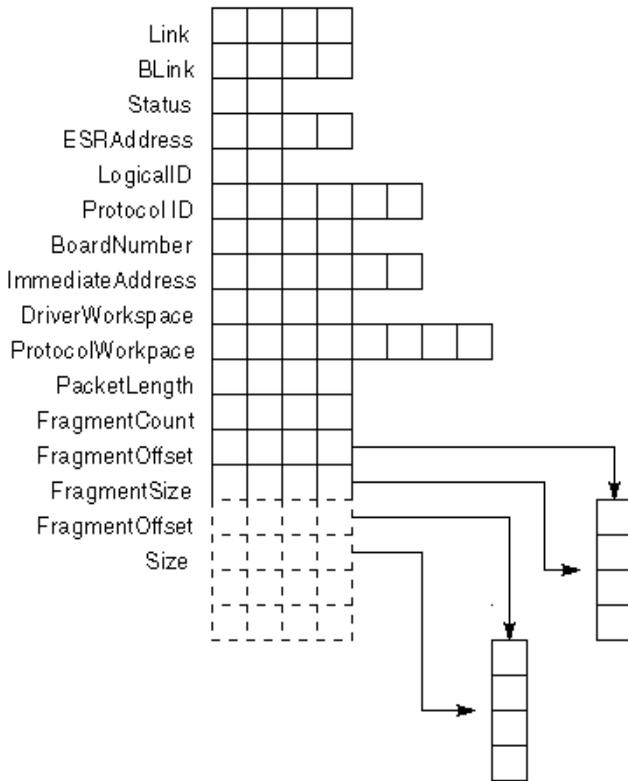


```

* BoardNumber          dd 0
* ImmediateAddress     db 6 dup (?)
  DriverWorkspace      dd 0
  ProtocolWorkspace    db 6 dup (?)
  ProtocolWorkspaceRest db 6 dup (?)
* PacketLength         dd 0
* FragmentCount        dd 0 : at least 1
* FragmentOffset       dd 0 : repeated FragmentCount times
* FragmentSize         dd 0
* The stack must fill in these fields before passing the ECB to the MLID.

```

Figure 22 Graphic Representation of the Transmit Event Control Block



Event Control Block Field Descriptions

Table 47 describes the fields used in the Event Control Block.

Table 47 Event Control Block Field Descriptions

Offset	Name	Size(bytes)	Descriptions
00h	Link	4	Forward link to another ECB. The LSL uses this field to queue ECBs. Protocol stacks and MLIDs can also use this field when they possess the ECB.
04h	BLink	4	This field is typically used as a back link for managing a list of ECBs. The current owner of the ECB uses this field. When an ECB is returned from an MLID containing a received packet, this field contains the received packet error status. See the "Setting the ECB BLink Field" section in Chapter 6, "Protocol Stack Packet Reception".
08h	Status	2	MLIDs must not use or modify this field. The LSL uses this field to indicate the current state of the ECB (for example, the ECB is currently unused, or queued for sending, etc.)
0Ah	ESRAddress	4	When an ECB originates from the LSL, the LSL sets this field, and MLIDs and protocol stacks must <i>not</i> use it. When a transmit ECB originates from the protocol stack, the protocol stack sets this field to point to a routine to be called when the transmission is complete and the ECB is available again.
0Eh	LogicalID	2	MLIDs use this field, but must not change it. When a protocol stack registers with the LSL, the LSL assigns the stack a logical number (0 through 15). This field contains the logical number. If the packet is a priority send, this field contains a value between 0FFF7h (lowest priority) and 0FFF0h (highest priority). If the packet is a raw send, this field contains a 0FFFFh. If the packet is a priority raw send, this field contains a value between 0FFFFh (lowest priority) and 0FFF8h (highest priority). On normal sends, the protocol stack places its own logical number in this field. On receives, the LSL places the target stack's logical number in this field.
10h	ProtocolID	6	This field contains the Protocol ID (PID) value on both sends and receives. This value is stored in high-low order. For a full explanation of how to fill out this field, refer to ODI Supplement: Frame Types and Protocol IDs.

Offset	Name	Size(bytes)	Descriptions
16h	BoardNumber	4	<p>When an MLID registers with the LSL for a particular LAN adapter, the LSL assigns that logical board a number. (Logical board 0 is used internally in the operating system.) Consequently, MLIDs are assigned logical board numbers 1 through 255). On sends, protocol stacks fill in this field to indicate the target logical board. On receives, the MLID fills in this field to indicate which logical board received the packet.</p>
1Ah	ImmediateAddress s	6	<p>On receives, the immediate address represents either 1) the packet's source node address or 2) the routing LAN adapter's node address if the packet was routed from another network. During a receive, the MLID fills in this field. This value is stored in high-low order. On RX-Net, or whenever the node address is less than six bytes, put the node address in the least significant byte and pad the remaining bytes with 0. On sends, the immediate address represents either the destination node address or the destination router address; the protocol stack fills in this field. Addresses passed to the upper layers are in either canonical or noncanonical format, depending upon whether the MLID bit-swaps MSB format addresses. The protocol stack fills in this field on sends. All addresses passed down to the MLID are in canonical format if the MLID is configured to be in LSB.</p> <p>In general, protocol stacks do not need to be aware of this field's format. Protocol stacks can just copy the contents of the receive ECBs <i>ImmediateAddress</i> field into the transmit ECBs <i>ImmediateAddress</i> field before sending the packet. Protocol stacks may get the immediate address from somewhere else, but it must still be copied into the transmit ECBs <i>ImmediateAddress</i> field.</p>

Offset	Name	Size(bytes)	Descriptions
20h	DriverWorkSpace	4	<p>An MLID can use this field for any purpose. However, be aware that the LSL uses the bytes at offsets 22h and 23h as temporary storage during the receive prescan stack filtering. Before passing a completed receive ECB to the LSL, the MLID will fill in the byte at offset 20h with the destination address type of the received packet:</p> <ul style="list-style-type: none"> ◆ 00h = Direct ◆ 01h = Multicast ◆ 03h = Broadcast ◆ 04h = Remote Unicast ◆ 08h = Remote Multicast ◆ 10h = No Source Route ◆ 20h = Error Packet ◆ 80h = Direct Unicast <p>Set the second byte of the field (offset 21h) to indicate whether the MAC header contains one or two 802.2 control bytes:</p> <ul style="list-style-type: none"> ◆ 0 = All frame types other than 802.2 ◆ 1 = 802.2 header has only Ctrl0 byte (Type I) ◆ 2 = 802.2 header has Ctrl0 and Ctrl1 (Type II) <p>For an explanation of 802.2 Type I and Type II, refer to ODI Supplement: Frame Types and Protocol Ids.</p> <p>The 16-bit word at offsets 2 and 3 of DriverWorkSpace must be filled in with the size of the received frame minus the MAC header.</p>
24h	ProtocolWorkspac e	8	Reserved for the protocol's workspace. The MLID must not modify this field.
2Ch	PacketLength	4	<p>This field contains the total length of the packet in bytes. This is the length of the data portion of the packet (not including media headers or SAP headers contained in the PacketEnvelope portion).</p> <p>On receives, the value in this field is also the value in the FragmentSize field; the MLID fills in this field. On sends, the protocol stack fills in this field.</p>

Offset	Name	Size(bytes)	Descriptions
30h	FragmentCount	4	<p>This field indicates the total number of packet fragment descriptors that follow. Each descriptor consists of a pointer to a fragment buffer and the size of that buffer.</p> <p>On receives, this value is always 1 or greater.</p> <p>On sends, the fragment count can be 0.</p> <p>Note: The ECB and all fragments must be guaranteed convertible to a valid physical address if the MLID uses <i>GetServerPhysicalOffset</i>. If the MLID is a type 4 NLM, it should not allow the OSData segment to contain the buffers.</p>
34h	FragmentOffset	4	<p>On receives, a buffer immediately follows the ECB in memory. The MLID copies the received packet into this buffer. After the MLID copies the packet into this buffer, it must set the <i>FragmentOffset</i> to point around any media headers to the data portion of the packet. The MLID must also set the <i>FragmentLength</i> field to the total length of the data portion of the packet.</p> <p>On sends, the <i>FragmentOffset</i> field points to the first fragment buffer containing packet data. The <i>FragmentSize</i> field specifies the length of that buffer. Additional fragment descriptors can immediately follow the ECB in memory.</p> <p>The MLID collects the data from these fragment buffers to form the packet for transmission (see Sample 14-2).</p>
38h	FragmentSize	4	<p>This field indicates the length in bytes of the first packet fragment. On receives, the value in this field is the same as the value in the <i>PacketLength</i> field. On sends, this value can be 0. On receives only, the memory immediately following the ECB also contains the following two fields:</p>
3Ch	MediaHeader	varies	<p>The media header of a packet is placed in this field. This field varies in length and appears only in receive ECBs. This field is not used or present if the LAN topology splits the data of a packet and transmits it in more than one frame (for example, RX-Net)..</p>
??h	Data	varies	<p>The data portion of the packet that immediately follows the <i>MediaHeader</i>. On sends only, the memory immediately following the ECB also contains the following two fields.</p>
3Ch	FragmentOffset2	4	<p>This field contains additional fragment descriptors when the <i>FragmentCount</i> is greater than 1.</p>

Offset	Name	Size(bytes)	Descriptions
40h	FragmentLength2	4	This field contains additional fragment descriptors when the <i>FragmentCount</i> is greater than 1.

Driver Firmware

MLIDs might need to download firmware to intelligent adapters. Because most intelligent adapters employ an on board microprocessor such as an 80186, the firmware code must be separately written, assembled, and linked to generate a binary file. This section describes how that firmware binary file can be attached to the MLID at link time and then transferred to the adapter during initialization.

To attach a firmware binary file to the MLID, the linker definition file (see Appendix B, "Assembling and Linking NLMs") must include the custom keyword followed by the name of the binary file. When the MLID is linked, the file is attached to the end of the code and becomes part of the NLM.

During the initialization process, the MLID allocates a buffer and copies the contents of the attached file to that buffer. The MLID can then download the contents of the firmware buffer to the adapter.

Reading Driver Firmware: Example Code

```

mov EAX, dword ptr [ESP + CustomDataSize]
                                ;get size of firmware
push MemoryRTag                 ;push tag
push EAX                        ;push size
push AllocSemiPermMemory        ;allocate memory to
lea ESP, [ESP + 4 * 2]           ;clean up stack
or EAX, EAX                      ;did we get it?
jz ErrorGettingExtraMemory      ;error exit if not
mov FirmWareBuffer@, EAX ESI, EAX ;save firmware buffer
mov mov EAX, [ESP + LoadableModuleFileHandle]
                                ;file handle firmware
mov EBX, [ESP + ]                ;read routine address
mov EDX, [ESP + CustomDataOffset] ;start address in file
mov ECX, [ESP + CustomDataSize]  ;get size of firmware
push ECX                         ;amount to read
push ESI                         ;where to read to
push EDX                         ;offset in file
push EAX                         ;file handle
call EBX                         ;call read routine

```

```
cli                ;stop interrupts
add ESP,4*4       ;adjust the stack
or EAX, EAX       ;check for read
jnz ReadError     ;errors
```

CustomDataSize, LoadableModuleFileHandle, ReadRoutine, CustomDataOffset, and CustomDataSize are parameters that the operating system passes on the stack to the MLID initialization routine. For a description of these parameters, see Chapter 15, "The MLID Initialization Routine".

15 The MLID Initialization Routine

Chapter Overview

This chapter covers the issues involved in initializing and registering the MLID. This chapter discusses determining hardware options, parsing the command line, allocating the frame and data adapter space, and setting up the board service routine.

The MLID Initialization Routine

When the NetWare operating system receives the command to load the driver, it calls the MLID's initialization routine. This initialization routine must accomplish the following:

- ◆ Allocate the frame and adapter data spaces
- ◆ Process the custom command line keywords and custom firmware
- ◆ Parse the standard LOAD command line options
- ◆ Register hardware options
- ◆ Initialize the adapter hardware
- ◆ Register the MLID with the Link Support Layer
- ◆ Load the firmware
- ◆ Allocate resource tags
- ◆ Set the hardware interrupts
- ◆ Start the callback routines

If the MLID is unsuccessful in these initialization tasks, it should return with EAX equal to a nonzero value.

If the MLID is successful, it returns with EAX set to 0.

Loading the MLID

After you have written, compiled, and linked the MLID, you can load it into the server's memory by entering the following command on the NetWare 3 and later server console:

```
load mydriver
```

The first time this command is issued, the NetWare loader loads the MLID into memory and calls the MLID's initialization procedure. When you used the NLM linker to create your NLM, you specified the name of the MLID's initialization procedure using the "start" keyword in the .DEF file. (For a description of definition files and keywords see Appendix B, "Assembling and Linking NLMs".) For example, if you had named the initialization procedure for an NE2000 MLID DriverInitialize, you would have included the following line in the NE2000.DEF file:

```
start DriverInitialize
```

Then, when the command load ne2000 is issued, the operating system calls your driver initialization procedure.

Requirements of the Calling Routine

The operating system routine that calls your initialization procedure is written in C, and the C programming language expects your initialization procedure to preserve the following registers: EBP, EBX, ESI, and EDI. The MLID's remove procedure must also save and restore these registers.

Initialization Parameters Passed on the Stack

When the operating system calls the MLID's initialization procedure, it passes nine parameters on the stack. These nine parameters are described below and are available for the MLID's use during initialization. Be aware, however, that after returning from the initialization routine, the file and screen handles are no longer valid.

Syntax of the MLID Initialization Routine

The operating system calls the MLID initialization routine as follows:

```
Long DriverInitialize(
```

```
Long    LoadableModuleHandle,  
Long    ScreenHandle,  
BYTE *CommandLine,  
Reserved (4 bytes),  
Reserved (4 bytes),  
Long    CustomeDataFileHandle,  
Long    ReadRoutine,  
Long    *CustomDataOffset,  
Long    CustomDataSize);
```

The following list defines the parameters in the initialization routine:

LoadableModuleHandle

This handle identifies your NetWare Loadable Module (NLM). Your initialization routine must provide this handle when calling many of the operating system support routines for MLIDs.

ScreenHandle

Your initialization routine must use this handle during the OutputToScreen function to perform any screen I/O. This handle is not valid after initialization.

CommandLine

This is a 4-byte pointer to the command line that was used to load the driver. This parameter is passed to ParseDriverParameters to get the hardware configuration information from the command line.

CustomDataFileHandle

The custom data file is appended to the end of your NLM by the NLM linker. Because the NLM was opened during loading, this handle points to a structure that the operating system uses to read the custom data file. This value is provided as a parameter to ReadRoutine.

ReadRoutine

This is a pointer to an operating system routine that reads the custom data file. (A description of ReadRoutine is found in Appendix A, "Operating System Support Routines".)

***CustomDataOffset**

This is the starting offset of the custom data inside the .NLM (or .LAN) file. This value is provided as a parameter to ReadRoutine.

CustomDataSize

This contains the length of the custom data file. This value is provided as a parameter to ReadRoutine.

Return Values

Zero Successful

Nonzero Unsuccessful: The MLID is unable to initialize.

Adapter Data Space and Frame Data Space

Each time the operating system calls the initialization procedure, the MLID creates a logical board. The initialization procedure needs to allocate the frame data space for that logical board's use. The MLID must also create a copy of the configuration table in this space. In addition, if the MLID is being loaded for the first time, the initialization routine must allocate the adapter data space.

If the network supervisor loads a second board of the same type into the server operating system, the NetWare loader will again call the MLID's initialization procedure. If the supervisor indicates that the server should support another LAN adapter, the initialization routine must not only allocate frame data space for the new logical board, but must also allocate memory for another adapter data space.

If the supervisor indicates that the server should support another frame format, the initialization procedure simply needs to allocate the frame data space for the new logical board. This logical board will use the previously created adapter data space to communicate with the LAN adapter.

Resource Tags

Before the MLID allocates any memory, or before it makes an operating system call that allocates memory, it must first obtain a resource tag by calling *AllocateResourceTag*. This routine passes the operating system three values:

- ◆ The MLID's NLM handle
- ◆ A description string of the MLID and the resource

- ◆ A value that identifies the type of resource tag needed

For example, the MLID might pass to the operating system a pointer to the null-terminated string "NE2000 Event Control Blocks" along with the value "SBCE". The operating system would return a resource tag that would allow the MLID to allocate Event Control Blocks (ECBs). With the returned resource tag, the MLID could then call *LSLGetSizedRcvECBRTag*. If, for some reason, the MLID cannot obtain all the resource tags it needs, it should abort the initialization procedure, return any allocated resources and not load at all.

Resource tags were implemented in NetWare 3.1 in order to allow the system supervisor to monitor the resources the NLMs are using. For example, if inordinate amounts of server memory are being used up, the network supervisor can see the description string of which NLM is taking the memory.

Table 48 shows some of the common operating system routines that require resource tags. For a complete description of the *AllocateResourceTag* routine, see Appendix A, "Operating System Support Routines".

NOTE: The fact that a resource tag has been allocated does not mean that the resource itself has been allocated. The resource tag is merely an identification number the MLID includes as a parameter when it calls for system resources.

Table 48 Routines that Require Resource Tags

Activity	Resource Signature Value	Operating System Routine
Obtaining an ECB	SBCE	LSLGetSizedRcvECBRTag
Registering for event notification	TNVE	RegisterForEventNotification
Allocating an interrupt and specifying an ISR entry point	PTNI	SetHardwareInterrupt
Reserving hardware options for the LAN adapter	SROI	RegisterHardwareOptions
Registering the MLID with the LSL	DILM	LSLRegisterMLIDRTag
Adding a polling procedure	RPLP	AddPollingProcedureRTag
Allocating memory	TRLA	Alloc
Scheduling an event on the timer interrupt	RMIT	ScheduleInterruptmmCallBack

Determining Hardware Options

After the operating system has allocated the resource tags, the MLID must determine the hardware configuration of the LAN adapter. This includes parameters such as the Hardware Instance Number (HIN) for Micro Channel, EISA, PCI, PnP ISA, and PC Card adapters, the base port for programmed I/O adapters, memory decode addresses for shared RAM adapters, interrupt numbers, and DMA channels. In Micro Channel, EISA, PCI, PnP ISA, and PC Card machines, the MLID can get this information directly from the system once the slot number has been identified.

The MLID's initialization routine should perform each of the following steps if appropriate for the hardware.

1. If the MLID supports multiple buses, it should use *SearchAdapter* or *ScanBusInfo* to determine the bus type (see also *The NetWare Bus Interface (NBI) Specification*).
2. If the MLID supports a *Plug and Play* bus, it should scan all slots using *SearchAdapter* or *GetInstanceNumber* to search for the adapter's ID. Any hardware instances that are found should be recorded in the IOSlot option list of the *AdapterOptionDefinitionStructure*. This structure is described in Appendix A, "Operating System Support Routines" under the *ParseDriverParameters* routine. (See also *The NetWare Bus Interface (NBI) Specification*.)

NOTE: Step 2 must be performed every time *DriverInit* is called, because hot plug cards can change the system hardware configuration between calls to *DriverInit*.

3. The MLID next calls *ParseDriverParameters* to determine the hardware configuration options (or Hardware Instance Number (HIN)) specified on the load command line and to query the operator for any required parameters which were not specified.

The *ParseDriverParameters* procedure requires an *AdapterOptionDefinitionStructure* containing the valid options for the hardware configuration. A *NeedsBitMap* is also required to indicate which specific hardware options must be obtained from either the command line or the console operator. (For a description of *ParseDriverParameters* see Appendix A, "Operating System Support Routines".)

Parsing the Command Line

Once the operating system has determined the hardware options, the MLID calls *ParseDriverParameters* to determine what options were specified with

the load command. *ParseDriverParameters* fills in the I/O portion of the logical board's configuration table (contained in the frame data space).

The following table illustrates the correspondence between the load options and the configuration table fields. The standard load options are described in the Appendix D, "Server Command Line Parameters and Keywords". An example load command is shown here:

```
load frame=ethernet_802.2, port=300, int=3
```

Table 49 Correspondence Between Load Options and Configuration Table Fields

Configuration Table Fields	Command Line
MLIDSlot	load <driver> SLOT=4
MLIDPort0	load <driver> PORT=300
MLIDIORange0	load <driver> PORT=300:A
MLIDIOPort1	load <driver>PORT1=700
MLIDIORange1	load <driver> PORT=700:14
MLIDMemoryDecode0	load <driver> MEM=C0000
MLIDMemoryLength0	load <driver> MEM=C0000:1000
MLIDMemoryDecode1	load <driver> MEM1=CC000
MLIDMemoryLength1	load <driver> MEM1=CC000:2000
MLIDInterrupt0	load <driver> INT=3
MLIDInterrupt1	load <driver> INT1=5
MLIDDMAUsage0	load <driver> DMA=0
MLIDDMAUsage1	load <driver> DMA1=3
MLIDChannelNumber	load <driver> CHANNEL=2 (NetWare 4 only)

After returning from *ParseDriverParameters*, the I/O portion of the logical board's configuration table in the frame data space has been filled in with the parsed values.

For *Plug and Play buses* (Micro Channel, EISA, PCI, PnP ISA, and PC Card), the configuration table now contains the selected Hardware Instance Number (HIN). The MLID should use *GetCardConfigInfo* to determine the configuration and fill in the Configuration Table (see the *NetWare Bus Interface (NBI) Specification*).

When the MLID has obtained all the information needed for the configuration table, the MLID's initialization routine calls *RegisterHardwareOptions*.

NOTE: In NetWare 4, if the driver must access shared memory before registering the hardware options, it must use *ReadPhysicalMemory* and *WritePhysicalMemory*.

Registering Hardware Options

After calling *ParseDriverParameters*, the MLID's initialization routine calls *RegisterHardwareOptions*, which tells the MLID whether the board being loaded is a new LAN adapter or will drive a new logical board (frame format) for an existing LAN adapter. If a new adapter is being registered, the MLID allocates the adapter data space and copies the *AdapterDataSpaceTemplate* to that area. This routine also notifies the MLID of any conflicts with existing hardware in the system.

The MLID must be able to handle four possible conditions on return from *RegisterHardwareOptions*:

- ◆ If $EAX = 0$, a new LAN adapter was successfully registered and the MLID must proceed with the hardware initialization. (The adapter data space must be allocated.)
- ◆ If $EAX = 1$, a new frame type for an existing adapter was successfully registered. For NetWare 3, anything greater than 1 is an error.
- ◆ If $EAX = 2$, a new channel for an existing multichannel adapter was successfully registered. The MLID typically treats the registering of a new channel as a new adapter (NetWare 4 only).
- ◆ If $EAX > 2$, the MLID was unable to register the hardware options.

Setting Up A Board Service Routine

You can implement the MLID's board service routine either as a polling procedure or as an Interrupt Service Routine (ISR). If you choose to implement a polling procedure to service your LAN adapters, we strongly recommend that you provide interrupt backup.

Using Polled Boards

By polling the LAN adapter, you can eliminate much of the overhead (interrupt latency) of using the operating system's interrupt handler. However, as a general rule, you should consider the following points before deciding to poll your LAN adapters:

- ◆ Does your LAN adapter have a DMA bus-mastering chip to move the packets off the board while the MLID is dormant or is performing other work?
- ◆ Does your LAN adapter have enough intelligence to detect if it is getting starved for CPU time, and if so, can it enable interrupt back up?
- ◆ Can your MLID process the "no operation" condition (in other words, nothing has happened on the board) in 10 to 20 instructions per board?

If you decide the MLID should poll the LAN adapters, the initialization procedure should call *AddPollingProcedureRTag* to initiate a polling procedure. You might want to have the MLID put a pointer to the adapter data space on a linked list after the LAN adapter is initialized and is ready to be polled. The front end for the board service routine can then use this list to quickly check all the LAN adapters for pending events.

NOTE: If the MLID is SMP aware, you should use *LSLAddPollingProcedure*.

In addition, you might want the MLID to call *SetHardwareInterrupt* to hook an interrupt for interrupt backup.

Using Interrupts

If you want the MLID to service the LAN adapters using interrupts, the initialization procedure needs to reserve an interrupt using the *SetHardwareInterrupt* system call. Using this call, the MLID can specify the address of an Interrupt Service Routine (ISR) entry point. Even though a single ISR procedure services all the LAN adapters of the same type in the system, you might want to specify alternate entry points into the ISR so that the logical board can get a pointer to the correct adapter data space.

NOTE: If the MLID is SMP aware, you should use *SetSymmetricInterrupt*. In NetWare 5 environments, the MLID should use *BusInterruptSetup*.

Initializing the LAN Adapter

At this point, the MLID initializes the adapter hardware. This consists of all setup appropriate for the hardware and might also include RAM and other

hardware tests. The MLID's reset routine should be called to handle all parts of the hardware initialization procedure.

NOTE: The MLID's initialization routine must set up the correct number of transmit buffers (the maximum number of simultaneous sends allowed by the hardware).

If an error occurs during the hardware initialization, the MLID's initialization routine should print an appropriate error message, return its resources, and return to the operating system with EAX set to a nonzero value. If the hardware initializes successfully, the MLID then registers with the LSL.

Registering with the LSL

Under the ODI specification, the MLID has two essential functions: 1) to take packets off the LAN adapter and pass them to the Link Support Layer (LSL), and 2) to take packets from the LSL and place them on the board.

Consequently, each time the MLID creates a logical board, it needs to call *LSLRegisterMLIDRTag*. When calling this command the MLID passes the LSL the address of its send procedure, the address of its control procedure handler (see Chapter 20, "MLID Control Procedures"), and a pointer to the logical board's configuration table (contained in the frame data space). The LSL returns a logical board number for the board that is registering. If an error occurs, *LSLRegisterMLIDRTag* returns a nonzero value in EAX.

If the MLID successfully registers, it must fill in the configuration table with the returned board number. MLIDs for intelligent bus master adapters can now pass the board number and frame ID information to the adapter, if necessary.

If the MLID is SMP aware it should call *LSLAssignMutexToInstance* to assign a Mutex lock to the logical board.

NOTE: Even though the MLID can request services from the operating system, it passes packets to the Link Support Layer.

Scheduling a Hardware Time Out Check

If the MLID is running an interrupt-driven LAN adapter, it could schedule a timer event that checks to see if the LAN adapter was unable to complete a send. To establish this timer event, the MLID can use the operating system routine *ScheduleInterruptTimeCallBack*, or *LSLAddTimerProcedure* for SMP aware MLIDs.

If the LAN adapter is not interrupt-driven, the polling procedure can check to see if a board failed to complete a send.

Error Handling

At any step in the initialization process, the MLID might encounter an error. If this occurs, the MLID must backtrack through all previous steps and return any system resources that were allocated. The *DriverInitialize* pseudocode contains an example of how to implement this kind of error checking.

Pseudocode for DriverInitialize

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

save the base and index registers (EBP, EBX, ESI, EDI)

IF first initialization of driver

    call AllocateResourceTag for Semi-Permanent Memory

    IF error allocating resource tag
        push pointer to error message
        jump to PrintMessage
    ENDIF

    call AllocateResourceTag for 10 Configuration for logical boards

    IF error allocating resource tag
        push pointer to error message
        jump to PrintMessage
    ENDIF

    /*Even polled drivers should get a hardware interrupt resource tag to
       provide interrupt backup */

    call AllocateResourceTag for Hardware Interrupt
    (unless your card is physically unable to support interrupts)

    IF error allocating resource tag
        push pointer to error message
        jump to PrintMessage
    ENDIF

    call AllocateResourceTag for MLID

    IF error allocating resource tag
        push pointer to error message
        jump to PrintMessage
```

ENDIF

call *AllocateResourceTag* for Timer Signature

IF error allocating resource tag
 push pointer to error message
 jump to *PrintMessage*

ENDIF

call *AllocateResourceTag* for Receive ECBs

IF error allocating resource tag
 push pointer to error message
 jump to *PrintMessage*

ENDIF

call *RegisterForEventNotification* for EventDownServer

IF error obtaining EvenUD (EventID= 0)
 push pointer to error message
 jump to *PrintMessage*

ENDIF

*/** You do not need to allocate a resource tag for a polling procedure
 (nor do the error checking) if you are not going to poll your board **/*

call *AllocateResourceTag* for polling procedure

IF error allocating resource tag
 call *UnRegisterEventNotification*
 push pointer to error message
 jump to *PrintMessage*

ENDIF

*/** We strongly recommend that you have only one polling procedure polling
 all the boards of the same type in the server**/*

*/** You do not need to call *AddPollingProcedureRTag* (nor error check)
 if you are not polling your board **/*

call *AddPollingProcedureRTag* or *LSLAddPollingProcedure* for SMP aware
MLIDs

IF unable to establish polling procedure
 call *UnRegisterEventNotification*
 push pointer to error message
 jump to *PrintMessage*

ENDIF

ENDIF

call *AllocSemiPermMemory* to get memory to hold frame data space information

IF error allocating memory

 push pointer to error message

 jump to *CancelPollingandEvents*

ENDIF

fill in frame data space information from a template in the Data Segment (OSDATA) save a pointer to the frame data space memory (in a register or in OSDATA) copy resource tag for 10 Configuration into the configuration table (contained in the Frame Data Space)

*/** If you are supporting a Micro Channel, EISA, PCI, PnPISA, or PCCARD boards, you should scan all the slots on the bus for your board by calling *SearchAdapter* and *GetInstanceNumber*. You can then provide this information to *ParseDriverParameters*. **/*

ParseDriverParameters */*fills in parts of the configuration table */* call *ParseDriverParameters* to get command line options

IF error parsing parameters

 push pointer to error message

 jump to *ReturnFrameMemory*

ENDIF

IF Micro Channel, EISA, PCI, PnPISA, or PCCARD, use

GetInstanceNumberMapping and *GetCardConfigInfo* to retrieve the remaining hardware resource value (1/0, Int, etc.) and save it into the configuration table.

ENDIF

call *RegisterHardwareOptions* to see if there are any conflicts with active hardware

IF error registering hardware

 push pointer to error message

 jump to *ReturnFrameMemory*

ENDIF

IF first time loading physical card (or new channel, if channels are supported)

 call *AllocSemiPermMemory* to allocate memory to hold adapter data space information

IF error allocating memory

```
    push pointer to error message
    jump to ReturnHardwareOptions
ENDIF
```

```
save a pointer to adapter data space memory (somewhere in OSDATA)
fill in adapter data space information from a template in Data
Segment (OSDATA) copy the pointer to the frame data space into the
adapter data space
```

```
/* even polled drivers should provide interrupt backup, unless your
board is physically unable to support interrupts */
```

```
call SetHardwareInterrupt, or SetSymmetricInterrupt if SMP aware, to
establish the board's hardware interrupt
```

```
IF error setting hardware interrupt
    push pointer to error message
    jump to ReturnPDDriverSpace
ENDIF
```

```
call a routine to initialize driver's hardware
```

```
IF error initializing hardware
    push pointer to error message
    jump to ReturnInterrupts
ENDIF
```

```
save node address in adapter data space unless ParseDriverParameters
filled it in
```

```
/* The next two steps schedule a "dead man timer" to see if your card
failed to complete a send. Not every card must schedule this
timeout check */
```

```
fill out the Interrupt time callback structure
call ScheduleInterruptTimeCallbackc
or
call LSLAddtimerProcedure if SMP aware (to call Timeout routine at a
specified interval)
initialize send queue for this adapter data space
```

```
ELSE
```

```
/* Loading a frame data space for an existing LAN adapter*/
```

```
get interrupt number from configuration table in frame data space
(filled in by ParseDriverParameters)
```

```
LOOP until all available adapter data spaces have been checked:
```

```

        check if interrupt number matches interrupt number in any adapter
        data spaces
    ENDLOOP

    IF error matching interrupt number with an adapter data space
        push error message
        jump to ReturnHardwareOptions
    ENDIF

    copy a pointer to the frame data space into the adapter data space
    move node address from adapter data space into frame data space

ENDIF

call LSLRegisterMLIDRtag (this call returns a board number and maximum
packet size)

IF error registering MLID with the LSL

    push pointer to error message

    IF first time loading LAN adapter
        jump to TurnOffCard
    ELSE
        jump to ReturnHardwareOptions
    ENDIF
ENDIF

IF the MLID is SMP aware
    call LSLAssignMutexToInstance
    save the board number into the configuration table of the frame data
    space

IF the operating system's maximum receive size is smaller than the frame
data space's maximum receive size

    reduce board's maximum receive size to match operating systems

ENDIF

/* calling LSLAddProtocolID binds the Protocol ID with a protocol stack */

call LSLAddProtocolID to register the Protocol ID for this logical board
with the IPX protocol stack

restore the base and index registers (EBP, EBX, ESI, EDI)
set eax = 0
ret /*Error Handling*/

```

```

TurnOffCard:
    turn off board

ReturnInterrupts:
    clear hardware interrupt contained in current adapter data space

ReturnPDriverSpace:
    return adapter data space memory

ReturnHardwareOptions:
    return hardware options contained in current frame data space

ReturnFrameMemory:
    return frame data space memory

CancelPollingandEvents:
    /* You do not need to perform the following check if you have not
       registered for event notification and are not using a polling
       procedure to service your boards */

    IF this is the only logical board
        call UnRegisterEventNotification
        call RemovePollingProcedure, or LSLRemovePollingProcedure if SMP
        aware
    ENDIF

    PrintMessage:
        pop pointer to error message
        print error message

    set EAX = -1 /* error condition */
    restore the base and index registers (EBP, EBX, ESI, EDI)
    ret

```

16

The MLID Packet Reception Routine

Chapter Overview

This section provides a brief overview of the commonly used reception methods available to the developer. This chapter discusses the board service routine, broadcasts and multicasts, as well as transmit complete and transmit error.

When the LAN adapter receives a packet, the MLID obtains an ECB which contains pointers to the necessary receive buffer(s) from the LSL, copies the packet into a receive buffer, and processes the ECB. The MLID then transfers the ECB to the LSL. The LSL then directs the ECB to the proper protocol stack.

The Packet Reception Routine

Your board service routine generally needs to detect and handle the following events on your adapter:

- ◆ Packet reception
- ◆ Packet reception error
- ◆ Transmission complete
- ◆ Packet transmission error

The MLID can be notified of these events by using either an Interrupt Service Routine (ISR), a polling procedure, or a polling procedure with interrupt backup. If you decide to implement a polling procedure for your board service routine, we strongly recommend that you provide interrupt backup.

Reception Methods

The receive portion of the board service routine checks for receive errors and jumps to an error handler if an error has occurred. Otherwise, the routine services the packet using one of the reception methods described below.

The method of packet reception you select depends upon the LAN adapter's data transfer method. The following examples illustrate a general flow of events.

Reception Method: Option 1

We recommend that you use this option for DMA and bus-mastering LAN adapters, as well as for any LAN adapters that use preallocated ECBs.

This is the simplest reception method. During development it might be helpful to initially use this method, then implement Option 2 after the MLID is functioning properly. The steps performed for this reception method are outlined below.

```
DriverInitMILD:    calls LSLGetSizeRcvECBRTag to get first ECB(s)
                  or
                  calls LSLGetMultipleSizeRcvECBRTag to get first ECB(s)
                  queues ECB(s) until a packet is received in DriverISR.
DriverISRMLID:    copies received packet into the DataBuffer.
                  checks the header information, and if valid:
                    fills in the remainder of the ECB fields
                    delivers the ECB to the LSL
                    attempts to get another ECB from the LSL to replenish the queue
IF the header information is not valid,
    calls the receive monitor, if one is registered.
ENDIF
queues the new ECB until next packet is received
calls LSLServiceEvents
```

Reception Method: Option 2

We recommend that you use this option for Programmed I/O and shared RAM LAN adapters.

This method uses a type of look-ahead process, in which the frame header information is first confirmed before the entire packet is transferred from the adapter into a receive buffer. For this reason, we recommend Option 2 over Option 1.

The LAN adapter's data transfer mode determines how the LookAhead process is handled. Programmed I/O adapters must transfer the size, in bytes, of the maximum frame header into a LookAhead buffer allocated for this purpose. If the LAN adapter uses a shared RAM transfer mode, the LookAhead buffer is simply the start of the packet in shared RAM.

The steps performed for this reception method are outlined below.

```
DriverISRMLID:    reads in the first part of the packet
                  checks the header information and if valid:
                    obtains an ECB
                    fills in the ECB
IF the header information is not valid
  IF a receive monitor is registered
    copies the remainder of the packet into an ECB
    calls the receive monitor
  ELSE
    discards the packet.
  ENDIF
ENDIF
IF the header information is valid
  copies the remainder of the packet into the ECB
  IF a receive monitor is registered
    calls the receive monitor
  ENDIFgives the ECB to the LSL.
ENDIF
```

For more information about the receive monitor, see ODI Supplement: *The Receive Monitor*.

Reception Method: Option 3

We recommend this method for intelligent adapters that are designed to be "ECB aware."

This option reduces the load on the server by offloading code to the LAN adapter. In this way, the LAN adapter's firmware handles most of the reception process. The steps performed for this reception method are outlined below.

```
DriverInit      Use LSLGetSizeRcvECBRTag obtain first ECB(s)
                 Queue ECB(s) until a packet is received.
Firmware
  Filters the frame header information and if valid,
    fills in all pertinent fields of the ECB.
  Generates interrupt when receive is complete (ready).
DriverISR
```

```

IF a receive monitor is registered, call the receive monitor
  Give the ECB to the LSL
Use LSLGetSizedRcvECBRTag to get another ECB for queue
or
Use LSLGetMultipleSizedRcvECBRTag to get another ECB for queue
Call LSLServiceEvents

```

Reception Method: RX-Net

RX-Net MLIDs use the following method to receive packets.

```

MLID:    checks the header information
IF packet fragment is 1 of 1
  calls LSLGetSizedRcvECBRTag to get an ECB
  fills in the ECB fields
  copies the data into the receive buffers
  gives the completed ECB back to the LSL using
    LSLHoldRcvEvent or LSLFastRcvEvent    ENDIF
IF packet fragment is first of many
  calculates worst case packet size by multiplying number of
  packet fragments by 504
  uses LSLGetSizedRcvECBRTag to get an ECB of worst case
  size
  fills in the ECB fields
  places time stamp in DriverWorkspace field to use in timing out
  ECBs that are never completed
  copies data from the first packet fragment into receive buffers
  places ECB on the internal queue until all the fragments have been
  received
IF packet fragment is one of many
  searches the queued ECB list for the ECB that the fragment belongs
  to
  IF the ECB is found
    IF MLID expected a fragment
      copies data into the receive buffers at the appropriate
      location
      increments TotalRxPacketFragmentOKcount
      IF last fragment
        gives completed ECB back to LSL using LSLHoldRcvEvent
or LSLFastRcvEvent
        increments TotalRxPacketCount
      adjusts TotalRxOKByteCount
      ELSE
        increments InvalidSplitFlagInPacketFrag
      ENDIF
    ELSE
      increments OrphanPacketFragmentCount
    ENDIF
  ENDIF
ENDIF
ENDIF

```

calls *LSLServiceEvents*

Front Ends for the Board Service Routine

Regardless of whether the MLID uses interrupts or polling to detect events on the LAN adapter, the front ends to your board service routines needs to perform the following tasks:

- ◆ Identify which LAN adapter has an event pending
- ◆ Obtain a pointer to the adapter data space for that LAN adapter.

After completing these tasks, the front end calls the main body of the board service routine.

Polling Front End

In order to poll the LAN adapters in the server, the MLID must obtain a pointer to the adapter data space for each LAN adapter. The MLID can efficiently obtain this pointer by keeping a pointer to each LAN adapter's adapter data space on a linked list. The polling front end can then get each pointer off the list and use the information in the adapter data space to interrogate each LAN adapter for pending events.

Remember, though, that the "no operation" condition (for example, nothing has happened on the board) will be the most prevalent case. And because the polling front end could be called several thousand times a second, you will want it to be highly optimized.

NOTE: Polling is normally not used by interrupt-driven MLIDs. However, some MLIDs might still require polling, or might use polling in addition to the ISR.

Interrupt Front End

The MLID can efficiently obtain a pointer to the adapter data space of the LAN adapter that has generated the interrupt by specifying alternate entry points into the interrupt front end. The MLID specifies these entry points when it calls *SetHardwareInterrupt*, *SetSymmetricInterrupt*, or *BusInterruptSetup* for a new LAN adapter. *BusInterruptSetup* allows you to pass a parameter on input that will be returned when the MLID's ISR is called.

At each of these entry points, the front end can load a register with a pointer to the adapter data space for that board. In this way, the body of the board service routine can reference all hardware specific information relative to this

register and never concern itself with knowing which specific board is being manipulated. **Figure 23** provides an illustration of how this implementation works.

Disabling Interrupts

The operating system's interrupt handler calls the front end with interrupts disabled. You must prevent the board service routine from recursing by disabling the interrupts on the LAN adapter.

Reenabling Interrupts

Before returning control to the operating system, the interrupt front end must reenble the interrupts on the board and EOI the PIC. The MLID must not issue an EOI. To EOI the interrupt, call *DoEndOfInterrupt*.

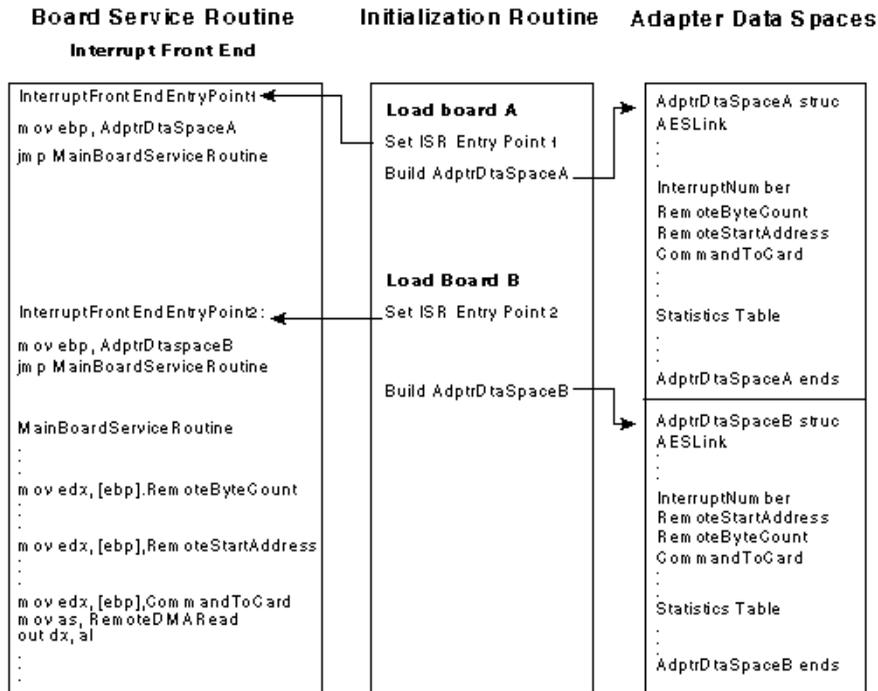
SMP Issues

In the SMP environment, supported in NetWare 4 and higher, if the MLID is SMP aware, it is not enough to disable the system interrupts or the LAN card interrupt. The MLID must also make use of the LSL locking APIs (*LSLAdapterMutexLock* and *LSLAdapterMutexUnlock*) to prevent the board service routine from being called recursively.

To prevent deadlocks, the system interrupts must be disabled on the processor you are running on before calling *LSLAdapterMutexLock*.

The MLID should also call *SMPDoEndOfInterrupt* instead of doing an EOI to the PIC directly. *SMPDoEndOfInterrupt* should be called only once in all cases.

Figure 23 Implementing Alternate Entry Points in the Interrupt Front End



The Board Service Routine

Generally, the receive portion of the board service routine must perform the following general operations:

- ◆ Check for any corruption that might have occurred in receiving the packet
- ◆ Determine the frame type of the packet (if more than one type is supported)
- ◆ Obtain and fill in a receive Event Control Block (ECB)
- ◆ Copy the packet from the adapter to the buffer described by the ECB (this step is not necessary if you have a bus-mastering DMA controller on your adapter)
- ◆ Pass the ECB to the Link Support Layer (LSL)

In general, the MLID's ISR should continue checking for receive and transmit events until it has serviced all of them. However, the MLID must not remain

in its ISR indefinitely. Care must be taken not to monopolize the CPU. For example, a gigabit Ethernet driver on a 200 Mhz Pentium processor should exit after processing about 30 receive packets.

Optimizing Packet Reception

Of all the functions in your MLID, the receive portion is the most important one to optimize. Because MLIDs spend the bulk of their processing time receiving packets, one of the most effective ways to improve your MLID's performance is to optimize the receive routine.

One strategy for optimization is to structure your MLID to use "fall through" code. That is, you identify the most common receive event (or default path), and then structure your MLID to handle that event with a minimum number of jumps or calls.

For example, suppose the most common event for your MLID is to receive EthernetII packets that are directly addressed to your LAN adapter and that have encountered no receive errors. You would write your code so that the flow of execution branches to handle errors, broadcasts and multicasts, different frame types, and other exceptions but not a successful EthernetII reception event.

Error Handling

While optimization is important, ensuring the data integrity of received packets should be the MLID's first priority. While the packet is being transmitted, traversing the wire, and being received, any number of errors can be introduced into it, and the MLID should check every possible hardware and software indicator to make sure the packet has not been corrupted. The exact nature of this error handling is determined by the capabilities of your LAN adapter and the media or topology it supports.

Error detection and handling are optional in the cases where the hardware is able to handle transmit and receive errors without MLID intervention. Even if the hardware has this capability, the MLID must still be able to update or maintain the statistics table.

Multiple Frame Support

If your MLID is designed to support multiple frame types, it must provide an algorithm to determine the frame type of each received packet. The media and frame types currently supported on NetWare are described in the ODI Specification Supplement: Frame Types and Protocol IDs.

Ethernet

With Ethernet LAN adapters, the MLID accomplishes multiple frame support by using logical boards and unique frame data spaces. The MLID must first determine the frame type of the packet and then get a pointer to the appropriate frame data space for the logical board receiving the packet.

The following pseudocode outlines an algorithm that can be used to determine the frame type of the incoming packet on Ethernet.

```
IF FrameLength/FrameType field is less than 1500 (decimal)
  IF the first two bytes following the FrameLength/FrameType field are
    FFh AND FFh
    the packet is raw 802.3
  ELSE/* the packet has a valid DSAP and SSAP*/
  IF DSAP is AAH AND SSAP is AAH AND Control is 03h
    the packet is SNAP
  ELSE      the packet is 802.2
ELSE
/*FrameType field is greater than 1500*/
  the packet is Ethernet II
```

Token-Ring

Token-Ring also supports multiple frame types using logical boards and unique frame data spaces. The MLID can determine the correct frame data space to load by evaluating the 802.2 header inside the 802.5 packet. If DSAP is AAh, SSAP is AAh, and Control is 03h, then the frame type is Token-Ring SNAP. If the frame type is not Token-Ring SNAP, the frame type is 802.2.

RX-Net

While RX-Net does not technically support frame types, it must accommodate three distinct packet types: Short, Long, and Exception. RX-Net MLIDs do not use logical boards to handle these different packet types; rather, they examine the packet to determine packet type and then fill out the receive ECB according to that packet type.

The following pseudocode outlines an algorithm that can be used to determine the packet type on RX-Net.

```
IF LongPacketFIag = 0 (byte 3 = 0)
  IF Pad2-SplitFIag = FF (second byte after Unusedportion)
    the packet is an RX-Net exception packet
  ELSE      the packet is an RX-Net long packet
ELSE /* ByteOffset (byte 3) contains a pointer around the unused portion
```

```
to the ProtocolType held */
the packet is an RX-Net short packet
```

Broadcasts and Multicast Packets

In general, the MLID only concerns itself with receiving packets that are directly addressed to the boards it is supporting. However, broadcast packets, which on many LAN adapters contain a value of FFFFFFFFh in the *DestinationAddress* field of the media header, should also be accepted.

NOTE: Broadcast packets on RX-Net contain a 0h in the *DestinationAddress* field of the media header, and the RX-Net MLID must change this field to FFFFFFFFh before handing the packet to the LSL.

If the MLID has enabled multicast reception on the LAN adapter, the MLID must also check whether the destination address in the packet matches an address in that LAN adapter's multicast tables. See *Ctl2_AddMulticastAddress* and *Ctl3_DeleteMulticastAddress* in Chapter 20, "MLID Control Procedures."

Filling in the ECB

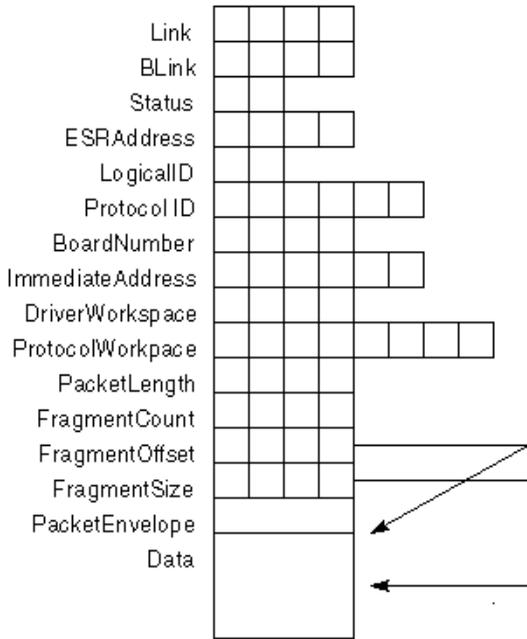
After performing all the error checking possible, the MLID fills out the receive ECB. The following table shows the fields that the MLID must fill in before handing the packet to the LSL. Refer to ODI Supplement: *Frame Types and Protocol IDs* for information regarding field names for specific media.

Table 50 Receive ECB Fields the MLID Must Set

Offset	Field Name	Description
04h	Blink	See the "Setting the ECB Blink Field" section below.
10h	ProtocolID	Contains the Protocol ID value extracted from the media header. See Table 53 and ODI Supplement: <i>Frame Types and Protocol IDs</i> for more detail.
16h	BoardNumber	This value comes out of the configuration table of the logical board receiving the packet.
1Ah	ImmediateAddress	This value is contained in the <i>SourceAddress</i> of the packet's media header. This address is the address of the LAN adapter that actually sent the packet you are receiving (not necessarily the LAN adapter that originated the packet).
20h	DriverWorkSpace	See the "Setting the ECB field <i>DriverWorkSpace</i> " section below.

Offset	Field Name	Description
2Ch	PacketLength	This field contains the length of the data portion of the packet. This does not include media headers.
30h	FragmentCount	This field contains the number of fragments that follow. In receive ECBs this value is always 1.
34h	FragmentOffset	On receives, a buffer immediately follows the ECB in memory. The MLID copies the received packet into this buffer. After the MLID copies the packet into this buffer, it must set the <i>FragmentOffset</i> to point around any media headers to the data portion of the packet. The MLID must also set the <i>FragmentLength</i> field to the total length of the data portion of the packet.
38h	FragmentSize	This field contains the length of the Data portion of the packet. This does not include media headers. This field contains the same value as <i>RPacketLength</i> .
3Ch	PacketEnvelope	This field is a variable length field that contains the media headers.
??h	Data	This field is a variable length field that begins with protocol header information and is followed by the rest of the packet.

Figure 24 Graphic Representation of the Receive Event Control Block



Setting the ECB BLink Field

The *BLink* field is typically used as a back link to manage a list of ECBs. The current owner of the ECB uses this field. When an ECB is returned from an MLID containing a received packet, this field contains the received packet error status as defined in the following table:

Table 51 ECB BLink Error Descriptions

Bit Value	Description
0000 0001h	CRC error, such as Frame Check Sequence (FCS) error.
0000 0002h	CRC/Frame Alignment error.
0000 0004h	Runt packet.

Bit Value	Description
0000 0010h	Packet is larger than allowed by the media.
0000 0020h	The received packet is for a frame type that is not supported. For example, the logical board is not registered for the frame type of the received packet. A board number associated with the physical adapter is placed in the lookahead structure.
0000 0040h	Malformed packet. For example, the packet size is smaller than the minimum size allowed for the Media Header, such as an incomplete MAC Header. In an Ethernet 802.3 header, the length field value is larger than the total packet size.
0000 4000h	Do not decompress the received packet.
8000 0000h	The MLID is shutting down.
No error bits set	If no error bits are set, the packet was received without error, and the data can be used. All undefined bits are cleared.

Setting the ECB DriverWorkspace ECB Field

Set the first byte in the *DriverWorkspace* field (offset 20h) to one of the following values):

Table 52 Setting the First Byte in the DriverWorkspace Field

Value	Type of Packet	Definition
00h	Direct	The packet is destined for this station only.
01h	Multicast	The packet is destined to a group of all nodes on the network. The adapter was registered to receive packets addressed to these addresses by a call to <i>Ctl2_AddMulticastAddress</i> .
02h	Broadcast	The packet is destined to all nodes on the physical network.
04h	Remote Unicast	The packet is destined to an individual node on the network. A remote unicast address is not addressed to this adapter's node address. The protocol stack must put the adapter into promiscuous mode if it wants to receive these packets.

Value	Type of Packet	Definition
08h	Remote Multicast	The packet is addressed to a group of nodes, but the adapter is not registered to receive it. (The MLID has not called <i>Ctl2_AddMulticastAddress</i> .) A protocol stack must put the adapter into promiscuous mode if it wants to receive these packets.
10h	No Source Route	In this case, the MLID received a source-routed packet, but there was no source route module (ROUTE.NLM) to record it, and the packet was not generated by the local ring.
20h	ErrorPacket	See the "Error Handling" section in this chapter.
80h	Direct Unicast	The packet is destined for this station only. Direct Unicast (80h) is new and should be used in place of Direct (00h).

Set the second byte (offset 21h) to indicate whether the MAC header contains one or two 802.2 control bytes:

- ◆ 00h All frame types other than 802.2
- ◆ 01h The 802.2 header has only Ctl0 byte (Type I)
- ◆ 02h The 802.2 header has Ctl0 and Ctl1 (Type II)

See ODI Supplement: *Frame Types and Protocol IDs* for an explanation of 802.2 Type I and Type II, and Chapter 14, "MLID Data Structures" for a description of the Event Control Block.

Table 53 Filling Out the ProtocolID Field of the Receive ECB

Frame Type	Value
RX-Net	For all RX-Net packet types, put the value in the <i>ProtocolType</i> field into the least significant byte and pad the five most significant bytes with 0.
Ethernet 802.3	Fill all 6 bytes with 0.
Ethernet 802.2	Put the value in the DSAP field into the least significant byte and pad the five most significant bytes with 0.
Ethernet EII	Put the value of the Ethernet type field in the last two bytes and pad the others with 0.
Ethernet SNAP	Set the most significant byte to 0 and put the value in the <i>ProtocolIdentification</i> field into the five least significant bytes.

Frame Type	Value
PCN2 802.2	Put the value in the DSAP field into the least significant byte and pad the five most significant bytes with 0.
PCN2 SNAP	Set the most significant byte to 0 and put the value in the <i>ProtocolIdentification</i> field into the five least significant bytes.
Token-Ring	Put the value in the DSAP field in the least significant byte and pad the five most significant bytes with 0.
Token-Ring SNAP	Set the most significant byte to 0 and put the value in the ProtocolIdentification field into the five least significant bytes.
FDDI	Put the value in the DSAP field in the least significant byte and pad the five most significant bytes with 0.
FDDI SNAP	Set the most significant byte to 0 and put the value in the ProtocolIdentification field into the five least significant bytes.

Handling Receive Errors

If the MLID encounters a receive error, it should perform the following actions:

- ◆ Attempt to identify the error. While some LAN adapters provide greater diagnostic support than others, the MLID pinpoints the specific cause of the error (buffer overflow, missed packet, checksum error, etc.).
- ◆ Increment diagnostic counters. The MLID maintains diagnostic counters for every detectable error condition on the LAN adapter. This will aid you in debugging the MLID as well as maintaining it in the future.
- ◆ Set the appropriate receive error bits in the ECB *Blink* field (see the "Setting the ECB Blink Field" section in this chapter).
- ◆ Attempt to recover from the error. When attempting to recover from a receive error, be careful not to interfere with the normal function of the LAN adapter. For example, the error handling should not interfere with a transmit in progress, nor should it affect packets that could be held in receive buffers on the LAN adapter.

Be sure to structure your code so that the MLID will not go into an infinite recovery loop. If the MLID encounters an error that is unrecoverable, follow the instructions given under "Unrecoverable Hardware Errors" in Chapter 20, "MLID Control Routines"/

Transmission Complete Interrupt

Each time the MLID receives a transmit complete interrupt, it should do two things:

- ◆ Release the ECB (if not already released during the MLID's packet transmission routine).
- ◆ Increment a "total packets transmitted" counter.
- ◆ Add the transmitted size to the TXOKBytes counter and, if the transmit is a group send, increment the TxGroupStats counter.
- ◆ Transmit the next packet, if one is waiting to be sent. Chapter 17, "The MLID Packet Transmission Routine" describes how the MLID should handle queuing transmissions.

Transmission Error

If the MLID encounters a transmit error, it should perform the following actions:

- ◆ Attempt to identify the error. As with receive errors, the MLID pinpoints the cause of the error as specifically as possible (excess collisions, cable disconnect, FIFO underrun, etc.).
- ◆ Increment diagnostic counters. As with receive errors, the MLID should maintain diagnostic counters for every detectable error condition on the LAN adapter.
- ◆ Attempt to send the packet again. In the event the MLID has reached the maximum retry limit for sending that particular packet, the MLID should transmit the next packet if one is waiting to be sent.

NOTE: For Token-Ring and FDDI adapters, do not resend the packet on a transmit error. All transmit errors should be ignored.

Be sure to structure your code so that the MLID will not go into an infinite recovery loop. If the MLID encounters an error that is unrecoverable, follow the instructions given under "Unrecoverable Hardware Errors" in Chapter 20, "MLID Control Routines".

Using Shared Interrupts

An MLID can support shared interrupts if they are also supported by the host bus and the LAN adapters that share the interrupt. Interrupts can be shared if

the bus is operating in level-triggered mode or if external logic exists on the adapters sharing the PCI bus and the interrupt.

- ◆ The PCI bus always supports shared interrupts.
- ◆ The Micro Channel bus always supports shared interrupts.
- ◆ The PC/AT bus uses edge-triggered interrupts and will not support shared interrupts unless external logic exists on the adapters sharing the interrupt.
- ◆ The EISA bus normally uses edge-triggered interrupts, but each interrupt can be individually set to level-triggered mode in order to support shared interrupts.

A ISR that supports shared interrupts is very similar to one that does not. If the MLID supports shared interrupts, the ISR must perform the following operations:

- ◆ The interrupt service routine immediately determines if the interrupt request is from its LAN adapter. If not, the ISR should return at once to the operating system ISR with EAX equal to one and the zero flag cleared.

```
or al, 01 h ; clear the zero flag
ret        ; return to operating system ISR code
```

- ◆ If the interrupt request is from the LAN adapter, the interrupt service routine should proceed. Upon completion, the ISR should return with EAX equal to 0 and with the zero flag set.

```
xor eax,eax ; zero eax & set the zero flag
ret        ; returns to operating system ISR code
```

The MLID must indicate that the LAN adapter is sharing interrupts by setting bit 5 in the *MLIDSharingFlags* field of the configuration table and setting the *ShareFlag* parameter when it calls *SetHardwareInterrupt*.

Pseudocode for the Board Service Routine

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

```
Entry State The Dir Flag is cleared.
Interrupts are disabled.
```

```
/* We recommend that the interrupts remain disabled during the MLID's
```

```

    ISR. */

Return State
The Dir Flag must be cleared.
The interrupts must be disabled.
No registers are preserved.

/* POLLING FRONT END */

If SMP aware call LSLAdapterMutexLock
LOOP through all adapters of same type in server
    load a register with a pointer to the adapter data space for this board
    set a semaphore to indicate this board is being polled
    IF there is work to be done on board
        call BoardServiceRoutine (see code below)
    ENDIF
    clear the semaphore to indicate the board is no longer being polled
ENDLOOP

IF SMP aware call LSLAdapterMutexUnlock

ret

/* INTERRUPT FRONT END */
IF SMP aware call LSLAdapterMutexLock
determine which board the interrupt is for (by using alternate entry
points for each board)

load a register with a pointer to the adapter data space for this board
mask the board's interrupt request line on the PIC OR disable interrupts
on the board
IF SMP aware call LSLDoEndOfInterrupt ELSE IF EOI FLAG=1
    Call the OS to EOI the slave PIC
ENDIF
EOI the master PIC
ENDIF

call BoardServiceRoutine
unmask the board's interrupt request line on the PIC
OR
re-enable interrupts on the board /* do not use iret */

IF SMP aware call LSLAdapterMutexUnlock
ret

BoardServiceRoutine:
LOOP until no more events are pending
    IF receive event

```

```

    IF board has received a packet
        read in enough of the packet to determine the frame type of the
        media
/* this next check is 802.3 specific */
    IF the FrameLength field in the media header is greater than
    the length report by the hardware
        increment diagnostic counter
        notify board that you are finished with the packet
        RETURN to the events pending loop
    ENDIF

    IF the frame type is NOT registered with your driver
        increment diagnostic counter
        notify board that you are finished with the packet
        RETURN to the events pending loop
    ENDIF
    IF packet is NOT addressed to this LAN adapter
        IF packet is NOT a broadcast packet
            IF address does not exist in the current multicast tables
                notify board that you are finished with the packet
                RETURN to the events pending loop
            ENDIF
        ENDIF
    ENDIF

    call LSLGetSizedRcvECBRTag
    IF no more ECBs are available OR packet size is too large
        increment a diagnostic counter
        notify board that you are finished with the packet
        RETURN to the events pending loop
    ENDIF

    fill in the PacketLength field of the ECB
    /* this is the length of Data portion of the packet,
    including protocol headers but not media headers */

    fill in the FragmentSize field of the ECB
    (this is the same value as PacketLength)
    fill in the ProtocolID field of the ECB
    copy the media headers from the packet to the PacketEnvelope
    portion of the ECB
    point FragmentOffset to the beginning of the Data portion of the
    ECB
    copy the remaining packet information into the Data area of the
    ECB
    copy board number into the BoardNumber field of ECB
    copy source address into the ImmediateAddress field of the ECB
    copy reception type into the first byte of the
    DriverWorkSpace field

```

```

    set second byte of DriverWorkspace field to appropriate 802.2
        type.

    call LSLHoldRcvEvent
    (you might want to refer to Chapter 11, "LSL Support Routines
        (Assembly Language)" to see if you can use LSLFastRcvEvent)
    increment diagnostic counter

    RETURN to the events pending loop

ELSE          /* the board encountered an error receiving the packet */
handle receive error
ENDIF

ELSE /* transmit event */
    IF transmit complete
    increment diagnostic counter
        IF a packet is queued
            unqueue packet
            call StartSend (see DriverSend pseudocode in Chapter 17)
            call LSLSendComplete
            (see Chapter 11, "LSL Support Routines (Assembly Language)"
                to see if you can use LSLFastSendComplete
            )
        ENDIF

    ELSE          /* error transmitting */
        increment diagnostic counter handle transmit error
    ENDIF
ENDIF
ENDLOOP

IF no receives or transmits processed
/* spurious event */
    increment diagnostic counter
ENDIF call LSLServiceEvents (only if LSLHoldRcvEvent or LSLSendComplete is
used)
ret

```

17

The MLID Packet Transmission Routine

Chapter Overview

This chapter discusses the common methods of packet transmission. This chapter covers queuing sends, multiple frame support, and packet length issues.

Packet Transmission

This section provides a brief overview of the methods commonly used for packet transmission.

General Transmission Method

In general, the MLID performs the following tasks during packet transmission:

1. Determines the number of adapter transmit resources available during its packet transmission routine.
2. Checks to see if it can handle another transmit and, if so, proceeds to step 3. Otherwise, builds the media header and queues the packet.
3. Copies the media header and data fragments to the transmit buffer.
4. Returns the ECB to the LSL. The MLID can return the ECB before the actual transmission is complete, as long as all information has been collected from the ECB and the MLID no longer needs it (This is also referred to as a "lying send".) The underlying transmit ECB will be placed in the LSL's hold queue until the MLID issues a service events command. The MLID can call either *LSLSendComplete* and *LSLServiceEvents* or

LSLFastSendComplete. (*LSLFastSendComplete* calls *LSLServiceEvents*.)

Sending a Packet

When sending a packet, a protocol stack assembles a list of fragment pointers in a transmit ECB and passes the packet to the LSL. The MLID builds the header, copies the packet fragments, and transmits the packet.

Each time the MLID initializes a logical board, it calls *LSLRegisterMLIDRTag* to register the following information with the Link Support Layer (LSL):

- ◆ The address of the logical board's send procedure
- ◆ The address of the logical board's control procedure handler
- ◆ The address of the logical board's configuration table

In response to this call, the LSL assigns a board number to that logical board. Later, when the LSL needs to send a packet to that board, it calls the send routine that the MLID registered. The LSL always calls the send routine in the following manner:

- ◆ With ESI pointing to an ECB that describes the packet to be sent
- ◆ With interrupts disabled
- ◆ With EBX containing the logical board number

Also, remember that the board service routine can call the MLID's send routine to transmit a packet on the send queue. Consequently, the send routine should be written to run at interrupt time, and we recommend that the send routine run with interrupts disabled.

If the MLID must enable interrupts in the course of the send routine (perhaps the MLID is waiting for an event to occur, or perhaps it is taking an unusually long time to send the packet), the MLID must disable interrupts before returning control to the LSL.

NOTE: The send routine does not need to preserve any registers.

Queuing Sends

Because the LAN adapter might not have any transmit buffers available when the LSL calls the MLID's send routine, the MLID must create and maintain a "first in, first out" (FIFO) send queue for the LAN adapter (as well as for every

other LAN adapter of the same type in the server). The MLID can use the ECB's Link and *BLink* fields for this purpose. Later, when the LAN adapter generates a transmit complete interrupt or when the polling procedure determines a send is complete, the ISR or polling procedure should take any ECBs off the send queue and transmit them.

Multiple Frame Support

If the MLID supports multiple frame formats, structure the send routine to prepare the appropriate media header for each logical board. The MLID inspects the *MLIDFrameType* field in the logical board's configuration table to determine which media header needs to be built.

Raw Sends

In some cases, the protocol stack has enough information about a logical board's media and frame type to completely prepare the packet for transmission. This is referred to as a "raw send." The protocol stack signals a raw send by putting a value between FFFFh and FFF8h (each value corresponds to a raw send priority) in the ECB's *LogicalIID* field. If the packet is to be a raw send, the MLID does not build a media header for the packet; it simply moves the packet to the LAN adapter and sends it.

Priority Sends

If a protocol stack supports priority transmissions and is communicating via an MLID that also supports priority transmissions (see *MLIDPrioritySup* in [Table 35](#)), the protocol stack may indicate a priority transmission by putting a value between FFF6h and FFF0h in the ECB *LogicalIID* field.

FFF7h can be used to indicate a normal (non-priority) transmission, but it is better to put the Protocol Stack ID in the *LogicalIID* field to indicate a normal (non-priority) transmission rather than FFF7h.

FFF6h indicates the lowest priority transmission, while FFF0h indicates the highest priority transmission.

Packet Length

On some topologies, the MLID must pad the packets being sent to meet a specific topology's length criteria.

Even Packet Length on Ethernet

Some Ethernet drivers on NetWare 2 servers will not route packets that are an odd length. Consequently, raw 802.3 and EII packets need to be an even length or "evenized." The send procedure must also set the *FrameLength* field in the 802.3 header to the actual length of the packet (including the evenizing byte). For example, if the MLID must transmit a 121-byte packet, it would evenize the packet to 122 bytes. Then, the MLID would set the *FrameLength* field in the 802.3 header to 122, and the LAN adapter would transmit 122 bytes. Although MLIDs with raw 802.3 and EII frame types should always evenize transmitted packets, they should not count on receiving only evenized packets. MLIDs should accept the evenized and odd-length packets they receive.

60 Byte Minimum Packet Size on Ethernet

The minimum packet size on Ethernet is 60 bytes, so the send procedure must pad any "short" packets with enough data to meet this requirement. The send procedure must also set the *FrameLength* field in the 802.3 header to the evenized length of the packet before padding. For example, if the MLID must transmit a 41-byte packet, it would pad the packet to 60 bytes. The MLID would set the *FrameLength* field to 42, and the LAN adapter would transmit 60 bytes.

Pseudocode for MLID Packet Transmission Routine

```
/* This pseudocode is intended to illustrate a flow of events and does
   not necessarily describe optimized code. */
IF MLIDShutdownState is DOWN (a flag indicating that the physical board
                               has been shutdown)
    jump to LSLFastSendComplete (which fully returns to the LSL)
ENDIF

IF board is transmitting
    put the packet on the send queue
    ret (to the procedure that called the MLID's packet transmission routine)
ENDIF

call StartSend jump LSLFastSendComplete (Lying send)

StartSend:
IF NOT a raw send
    set PacketLength = actual data size + media header
ENDIF

/* if your driver supports Ethernet 802.3 or Ethernet II you will need to
```

```

    evenize any packets that have an odd length */

/* if your driver supports Ethernet you will need to pad any short packets
   to a minimum of 60 bytes */
inform board that it will need to send PacketLength bytes

IF NOT a raw send
    build all the necessary headers for the packet (see the ODI Specification
        Supplement: Frame Types and Protocol IDs for media- and frame- specific
        header information)
    move all necessary headers to the transmit buffer
ENDIF

move fragments to transmit buffer
tell the board to send the packet
get current time and save for timeout check
reset your driver's retry counter
ret

```

Pseudocode for Packet Transmission Routine for RX-Net MLIDs

```

IF MLID is shut down
    jump to LSLFastSendComplete
ENDIF

IF ECBs are queued
    queue ECB
    return
ENDIF

IF board is transmitting
    queue ECB
    return
ENDIF

Get data size from ECB

IF data will fit into one packet
    copy packet to board
    save time stamp for timeout routine
    reset TxRetryCount
    tell LAN adapter to transmit packet
ELSE
    calculate the number of fragments by dividing data size by 504
    copy first fragment to board
    save time stamp for TimeOut routine

```

```
    reset Tx
    retry count
    tell LAN adapter to transmit packet
ENDIF
```

Save the state so that after the transmission completes, the MLID can determine if there are more fragments of the current ECB to send or if it should check the transmit queue.

18 MLID Timeout Procedure

Chapter Overview

This chapter discusses the timeout procedure. The chapter discusses scheduling the interrupt time callback, determining the wait interval, identifying a timeout error, using system alerts, and reinitializing the LAN adapter.

Establishing a Timeout Procedure

Depending on the hardware capabilities of the LAN adapter, the MLID might need to establish a timeout check or "dead man timer" that regularly checks the LAN adapter to determine if the LAN adapter is blocked by an unfinished transmission. If a transmission has failed to complete after a reasonable period of time, the timeout procedure should perform the following steps:

- ◆ Call `QueueSystemAlert`
- ◆ Reinitialize the LAN adapter
- ◆ Increment diagnostic counters

Scheduling an Interrupt Time Callback

You can establish a timer function for the MLID's timeout check using the *ScheduleInterruptTimeCallBack* function call.

ScheduleInterruptTimeCallBack hooks into the timer interrupt and, after the specified interval, calls the timeout procedure as part of the timer's interrupt service routine.

NOTE: If the call back routine makes blocking calls, you can schedule an AES event.

IMPORTANT: *ScheduleInterruptTimeCallBack* results in the timeout procedure being called at interrupt time. *ScheduleInterruptTimeCallBack* does not create a perpetual timer function. Each time the MLID timeout procedure is called, it must call *ScheduleInterruptTimeCallBack* again to reschedule the next interrupt time callback.

NOTE: SMP aware MLIDs should use *LSLAddTimerProcedure* instead of *ScheduleInterruptTimerCallBack*. *LSLAddTimerProcedure* creates a perpetual timer procedure and only needs to be called once.

Determining the Wait Interval

You might need to experiment with the interval you set between timeout checks in order to empirically determine the optimal wait interval. This value is affected by the LAN adapter's hardware, the network topology, and the network load. As a general rule, you will want to start working with an interval of 1 or 2 seconds.

Identifying a Timeout Error

Immediately after the MLID sends a packet, the send procedure should call *GetCurrentTime* and save the returned time value for later inspection. The timeout procedure also calls *GetCurrentTime* and compares the returned time value MLIDTimeout Procedure with the time value saved by the MLID's send procedure. If the difference between the two values is less than the established wait interval, the timeout procedure simply reschedules itself. If the wait interval has expired and the LAN adapter is still trying to transmit, a timeout condition has occurred.

Using System Alerts

Whenever a timeout condition occurs, the timeout procedure should call *QueueSystemAlert* or *NetWareAlert*. This system call not only allows the MLID to print a message at the file server console, but also makes this information available to a wide range of network management applications.

Reinitializing the LAN adapter

After identifying a timeout condition, the MLID should try to reinitialize the LAN adapter without destroying the send event in progress. If the maximum number of retries allowed for the LAN adapter has not been exceeded, the MLID should increment the retry counter and tell the LAN adapter to resend

the packet. If the maximum number of retries has been exceeded, the MLID increments the transmit error diagnostic counter, resets and clears the transmission bits and buffers on the board, and then sends the next packet on the send queue (if one is waiting to be sent).

Pseudocode for TimeOutCheck

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

IF the MLID is SMP aware
  call LSLAdapterMutexLock

IF a transmit-in-progress flag has been set
  call GetCurrentTime
  IF transmit has been pending for too long
    call QueueSystemAlert
    call Ctl6_MLIDReset (see Chapter 20, "MLID Control Routines ")
    increment diagnostic counter
    clear transmit-in-progress flag
    get next send
    IF a packet is queued
      call StartSend (see pseudocode outline for the MLID packet
        transmission procedure)
      call LSLFastSendComplete (Lying send)
    ENDIF
  ENDIF
ENDIF

IF MLID is RX-Net
  check the queued receive ECB list to see if any ECBs have timed out.
ENDIF

IF the MLID is not SMP aware
  /* reschedule interrupt time callback */
  call ScheduleInterruptTimeCallback
  ret
ENDIF
IF the MLID is SMP aware
  call LSLAdapterMutexUnLock
```

Pseudocode for the Timeout Procedure for RX-Net MLIDs

```
Interrupts are disabled  
Mask off the LAN adapter's transmit interrupt  
Issue the Disable Transmitter command to the LAN adapter  
ret
```

19 Remove MLID Procedure

Chapter Overview

This chapter discusses the remove MLID procedure. The remove MLID procedure is a routine that allows the operating system to dynamically unload the MLID. This chapter discusses how the MLID should shut down. In particular, it covers deregistering, cancelling events, shutting down the LAN adapter, and removing the data spaces.

Removing the MLID

Every NetWare Loadable Module (NLM) must have a remove procedure that allows the network supervisor to unload the NLM from the operating system. This procedure must shut down the LAN adapter and give back any resources the MLID has allocated specifically, such as interrupts and memory. When you use the NLM linker to create your NLM, you specify the name of your remove procedure in the .DEF file using the "exit" keyword. For example, if the remove procedure for an NE2000 MLID were named *DriverRemove*, you would include the following line in the NE2000.DEF file:

```
exit MLIDRemove
```

Then, when the network supervisor enters `unload ne2000` on the server console, the operating system calls your MLID's removal routine.

DeRegistering Logical Boards

When unloading the MLID, the remove procedure must deregister all of the MLID's logical boards with the Link Support Layer (LSL). In addition, the remove procedure must deregister the hardware options the MLID reserved

with the operating system. The routine accomplishes this by calling the following two calls for each logical board the MLID supports:

- ♦ `LSLUnBindThenDeRegisterMLID`
- ♦ `DeRegisterHardwareOptions`

Be aware, however, that *LSLUnBindThenDeRegisterMLID* gives protocol stacks an opportunity to call the MLID's packet transmission routine and transmit packets to other servers or stations on the network, notifying them that the LAN adapter is being unloaded. Therefore, if the LAN adapter is unable to send, the MLID's remove procedure should call *LSLDeRegisterMLID* instead.

Canceling Polling Procedures and Timer Events

If the MLID polls the LAN adapter, it must cancel the polling procedure by calling *RemovePollingProcedure*, or if the MLID is SMP aware, by calling *LSLRemovePollingProcedure*.

Also, be sure to cancel any timer events before returning control to the operating system. If the MLID's remove procedure does not cancel all AES events and interrupt time callbacks, the server will most likely abend when the operating system tries to call one of the MLID's procedures at a memory address that is no longer valid.

Shutting Down the LAN Adapter

The remove procedure must disable the LAN adapter turn-off interrupts, etc. If the MLID is driving an interrupt-driven LAN adapter or using interrupt backup for a polling procedure, the remove procedure must call *ClearHardwareInterrupt* to return the LAN adapter's interrupt to the operating system. If the MLID is SMP aware, the remove procedure must call *ClearSymmetricInterrupt* to return the LAN adapter's interrupt to the operating system.

In NetWare 5 environments, the MLID must use *BusInterruptClear*.

Removing Data Spaces

The MLID must also give back all the memory that each LAN adapter has allocated for the adapter data space and frame data space. However, the MLID should be careful not to try to remove any adapter data space or frame data

space for a logical board that has been completely shut down using *Ctl6_MLIDShutdown*.

Pseudocode for Remove MLID

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

/* The Ctl6_MLIDReset and Ctl5_MLIDShutDown control procedures also call
   portions of the remove procedure. The elements of the outline that pertain
   to these two calls are indicated below. */

save the base and index registers (EBX, EBP, ESI, EDI)
LOOP until all the LAN adapters are removed
  IF the LAN adapter has not been completely shut down previously by
    Ctl5_MLIDShutdown
    /* because we want to call LSLUnBindThenDeRegisterMLID we set this
       flag */
    set unbinding flag to 1
    call RemoveInstance
  ENDFENDLOOP

/* if you are using a polling procedure you will need to cancel it */
call RemovePollingProcedure
or
If SMP aware, call LSLRemovePollingProcedure
restore the base and index registers (EBX, EBP, ESI, EDI)

RemoveInstance:
If SMP aware, call LSLRemoveMutexFromInstance
  IF unbinding flag is set
    LOOP until all logical boards have been unbound
      call LSLUnBindThenDeRegisterMLID
      call DeRegisterHardwareOptions
    ENDLLOOP
  ELSE
    LOOP until all logical boards have been deregistered
      call LSLDeRegisterMLID
      call DeRegisterHardwareOptions
    ENDLLOOP
  ENDIF
disable LAN adaptor

/* if the LAN adapter is interrupt driven or the MLID uses interrupt
   backup */
```

```
call ClearHardwareInterrupt
or
If SMP aware, call ClearSymmetricInterrupt

/* if the MLID uses a timeout check or "dead man timer" */
call CancelInterruptTimeCallback
or
If SMP aware, call LSLRemoveTimerProcedure
cancel any outstanding timer events
LOOP until the memory for all frame data spaces has been returned
    call FreeSemiPermMemory to give back frame data space memory
ENDLOOP

call FreeSemiPermMemory to give back memory for the adapter data space
ret /* to remove the next LAN adapter */
```

20 MLID Control Routines

This chapter discusses the control routines that ODI requires an MLID to provide.

The Open Data-Link Interface (ODI) requires the MLID to make a number of control procedures available to protocol stacks and other NetWare Loadable Modules (NLMs). When the MLID calls *LSLRegisterMLIDRTag* during initialization, it passes the address of the MLID's control procedure handler to the Link Support Layer (LSL). The LSL, in turn, gives this address to any protocol stacks or NLMs that request it.

Preliminary Information

Contains preliminary information for working with the MLID Control Routines.

Processor States

Entry State

ODI requires that the MLID Control Procedure Handler always be called with the processor entry state set as follows:

EAX

contains the board number.

EBX

contains a subfunction number (0-18).

Interrupts

can be enabled or disabled.

Other registers

may contain parameters.

NOTE: The MLID control procedures must be called at process time.

Return State

After executing the requested control procedure, ODI requires the MLID to return with the processor return state set as follows:

EAX

contains the completion code.

Interrupts

state is preserved.

The MLID does not have to fully implement every control procedure defined by ODI; however, it must return a completion code for each control procedure.

Interrupt States

If a control procedure is called with interrupts enabled, it should return with interrupts enabled. If a control procedure is called with interrupts disabled, it should return with interrupts disabled. Nevertheless, each control procedure can enable or disable interrupts as necessary.

Required and Optional Control Procedures

All MLIDs are required to implement the following control procedures:

- ◆ Ctl0_GetMLIDConfiguration
- ◆ Ctl1_GetMLIDStatistics
- ◆ Ctl5_MLIDShutdown
- ◆ Ctl6_MLIDReset
- ◆ Ctl2_AddMulticastAddress (if supported by the hardware)
- ◆ Ctl3_DeleteMulticastAddress (if supported by the hardware)
- ◆ Ctl9_SetLookAheadSize

MLIDs can optionally support the following control procedures:

- ◆ Ctl10_MLIDPromiscuousChange
- ◆ Ctl11_RegisterMonitor
- ◆ Ctl14_DriverManagement
- ◆ Ctl16_RemoveNetworkInterface
- ◆ Ctl17_ShutdownNetworkInterface
- ◆ Ctl18_ResetNetworkInterface

We recommend that the MLID support multicast addressing if the hardware allows. However, if the MLID does not support the optional control procedures, it must return BadCommand (EAX = FFFFFFFF81h) for these two procedures.

Earlier versions of the ODI specification define six control procedures that the MLID no longer uses. Nevertheless, the MLID must provide these procedures in order to maintain compatibility with previous versions of the Open Data-Link Interface. The following control procedures should return BadCommand (EAX=FFFFFFF81h) when called:

- ◆ Ctl4_Reserved
- ◆ Ctl7_Reserved
- ◆ Ctl8_Reserved
- ◆ Ctl12_Reserved
- ◆ Ctl13_Reserved
- ◆ Ctl15_Reserved

Unrecoverable Hardware Errors

If *Ctl6_MLIDReset* cannot reinitialize the LAN adapter, the MLID should permanently shut down the adapter after queuing a system alert.

MLID Control Routines

The following lists provide the names the IOCTLs described in this chapter:

Multicast Routines

- ◆ Ctl2_AddMulticastAddress
- ◆ Ctl3_DeleteMulticastAddress

NLM Interaction Routines

- ◆ Ctl4_DriverManagement
- ◆ Ctl6_MLIDReset
- ◆ Ctl5_MLIDShutdown
- ◆ Ctl16_RemoveNetworkInterface
- ◆ Ctl17_ShutdownNetworkInterface
- ◆ Ctl18_ResetNetworkInterface

Obtaining Structures Routines

- ◆ Ctl0_GetMLIDConfiguration
- ◆ Ctl1_GetMLIDStatistics

Packet Reception Routines

- ◆ Ctl9_SetLookAheadSize

- ◆ Ctl10_DriverPromiscuousChange mode

Registration Routines

- ◆ Ctl11_RegisterReceiveMonitor

Ctl0_GetMLIDConfiguration

Returns a pointer to the configuration table.

Entry State

EAX

The logical board number.

EBX

Subfunction = 0.

Interrupt

cli or sti (could temporarily be sti).

Call

At process time.

Return State

EAX

Set to 0.

ESI

Pointer to the logical board's configuration table (part of the frame data space).

Interrupt

State is preserved.

Preserved

EBP

Completion Codes (EAX)

None.

Remarks

Ctl0_GetMLIDConfiguration returns a pointer to the logical board's configuration table.

Ctl1_GetMLIDStatistics

Returns a pointer to the statistics table.

Entry State

EAX

The logical board number.

EBX

Subfunction = 1.

Interrupt

cli or sti (could temporarily be sti).

Call

At process time.

Return State

EAX

Set to 0.

ESI

Pointer to the statistics table of the LAN adapter (part of the adapter data space) that is being used by the logical board indicated on entry in EAX.

Interrupt

State is preserved.

Preserved

EBP

Completion Codes (EAX)

None.

Remarks

CtlI_GetMLIDStatistics returns a pointer to the statistics table. The logical board number passed in EAX indicates which logical board the statistics are for. ESI holds a pointer to the statistics table of the LAN adapter to which the logical board is bound.

Ctl2_AddMulticastAddress

Adds the specified node address to the multicast address table.

Entry State

EAX

The logical board number.

EBX

Subfunction = 2.

ESI

Pointer to the 6-byte multicast address to add to the multicast address list

Interrupt

cli or sti (could temporarily be sti).

Call

At process time.

Return State

EAX

Has a completion code.

Interrupt

State is preserved.

Preserved

EBP

Completion Codes (EAX)

0x00000000	Successful	Multicast address has been added or already exists.
0x0FFFFFF81	BadCommnd	The board does not support multicast addressing.

0x0FFFFF82	BadParameter	Address is not a valid multicast address.
0x0FFFFF89	OutOfResources	Number of multicast addresses is greater than allowed.

Remarks

Ctl2_AddMulticastAddress adds the specified node address to a LAN adapter's multicast address table.

Even though this control procedure receives a logical board number, the multicast address table is only maintained for each LAN adapter. This procedure uses the logical board number in order to reference the appropriate LAN adapter. Be aware, however, that this call affects all logical boards using the affected LAN adapter.

More than one protocol stack or NLM could add the same multicast address for the same logical or LAN adapter. For this reason, the MLID should maintain a count of how many times a multicast address has been added. This prevents one protocol stack from deleting a multicast address that another protocol stack is still using.

Bit 3 of the *MLIDModeFlags* must be set to indicate whether or not multicast addressing is supported. This bit is set to 1 if the MLID supports multicast addressing.

The MLID should enable multicast reception on the LAN adapter and might also calculate a hash bit that is sent to the LAN adapter. The LAN adapter could then use this hash bit in its own multicast table to filter out most (but not all) unwanted packets. If the hardware cannot guarantee multicast filtering, the MLID must inspect the destination address of the packets to insure the proper multicast filtering.

NOTE: Adapters most commonly use hashing to filter incoming packets. However, hashing does not guarantee 100% multicast filtering. This is why the MLID might need to look up incoming packets in its multicast address table to ensure that the packet's destination address is enabled.

Token-Ring Multicast Addresses

Token-Ring supports two kinds of group addresses: multicast addresses, similar to those used by Ethernet; and functional addresses, where individual bits designate the intended recipient. The format of each of these is as follows:

```
multicast:  W0 00 VY YY YY YY
```

functional: C0 00 XY YY YY YZ

- ◆ V is a number between 8 and Fh
- ◆ W is either 8 or Ch
- ◆ X is a number between 0 and 7h
- ◆ Y is a number between 0 and Fh
- ◆ Z must be an even number

Canonical and Noncanonical Addresses

Noncanonical addresses are significant bit first on the wire. Canonical addresses are sent with the least significant bit of the most significant byte first on the wire.

A Token-Ring MLID's default is to send Physical Layer addresses in noncanonical format. Token-Ring MLIDs are the only MLIDs that can select between canonical and noncanonical address formats. All other MLIDs must use canonical format.

NOTE: Even though FDDI uses noncanonical addresses at the Physical Layer, it presents canonical addresses to the Data-Link Layer.

Specifying Address Formats

Bits 9 and 10 of the configuration table *MLIDFlags* field specify the different support mechanisms for multicast filtering and multicast address format. These bits are only valid if bit 3 of the *MLIDModeFlags* is set, indicating that the MLID supports multicast addressing.

The MLID sets bit 10 if it has specialized adapter hardware (such as hardware that utilizes CAM memory). If this bit is set, *DriverMulticastChange* receives a pointer to the multicast address table and the number of addresses in the table.

NOTE: 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.

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 reset 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.

Table 54 Possible Combinations of MLID Flags, Bits 9 and 10

Bit 10	Bit 9	Meaning
0	0	The format of the multicast address defaults to that of the topology: Ethernet => Group Addressing Token-Ring => Functional Addressing PCN2 => Functional Addressing FDDI => Group Addressing
0	1	Illegal value and must not occur.
1	0	A specialized adapter supports group addressing, but the MLID should filter the addresses.
1	1	A specialized adapter supports group addressing, and the MLID is not required to filter the addresses.

See Also

- ◆ ODI Supplement: *Canonical and Noncanonical Addressing* for more information about the receive monitor.

Ctl3_DeleteMulticastAddress

Deletes a specified node address from the multicast table.

Entry State

EAX

The logical board number.

EBX

Subfunction = 3.

ESI

Pointer to the 6-byte multicast address to delete from the multicast address list.

Interrupt

cli or sti (could temporarily be sti).

Call

At process time.

Return State

EAX

Has a completion code.

Interrupt

State is preserved.

Preserved

EBP

Completion Codes (EAX)

0x00000000	Successful	The address was successfully deleted, or is no longer in use.
------------	------------	---

0x0FFFFF81	BadCommnd	The board does not support multicast addressing.
0x0FFFFF85	ItemNotPresent	No matching address was found.

Remarks

Ctl3_DeleteMulticastAddress deletes a specified node address from a LAN adapter's multicast table.

Even though *Ctl3_DeleteMulticastAddress* receives a logical board number, the multicast address table is only maintained for each LAN adapter. This procedure uses the logical board number in order to reference the appropriate LAN adapter. Be aware, however, that this call affects all logical boards using the affected LAN adapter.

More than one protocol stack or NLM could add the same multicast address for the same logical or LAN adapter. For this reason, the MLIID should maintain a count of how many times a multicast address has been added. If this counter is greater than 1, *Ctl3_DeleteMulticastAddress* decrements the counter and returns a successful completion code. If this counter is 1, *Ctl3_DeleteMulticastAddress* removes the address from the table and notifies the LAN adapter that the address has been removed. In this way, the MLID will not delete an address that another protocol stack is still using.

If an address is deleted from the table and the LAN adapter supports a hash table, *Ctl3_DeleteMulticastAddress* also rewrites the LAN adapter's hash table to reflect the change. If the deleted address is the last multicast address contained in the table, this procedure also disables multicast reception on the LAN adapter, if it is appropriate.

Canonical and Noncanonical Addresses

Noncanonical addresses are addresses sent with the most significant bit first on the wire. Canonical addresses are sent with the least significant bit of the most significant byte first on the wire.

Token-Ring MLIDs default is to send Physical Layer addresses in noncanonical format. Token-Ring MLIDs are the only MLIDs that can select between canonical and noncanonical address formats. All other MLIDs must use canonical format.

NOTE: Even though FDDI uses noncanonical addresses at the Physical Layer, it presents canonical addresses to the Data-Link Layer.

Specifying Address Formats

Bits 9 and 10 of the configuration table MLIDFlags field specify the different support mechanisms for multicast filtering and multicast address format. These bits are only valid if bit 3 of the MLIDModeFlags is set, indicating that the MLID supports multicast addressing.

The MLID sets bit 10 if it has specialized adapter hardware (such as hardware that utilizes CAM memory). If this bit is set, *DriverMulticastChange* receives a pointer to the multicast address table and the number of addresses in the table.

NOTE: 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.

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 reset 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.

Table 55 Possible Combinations of MLID Flags, Bits 9 and 10

Bit 10	Bit 9	Meaning
0	0	The format of the multicast address defaults to that of the topology: Ethernet => Group Addressing Token-Ring => Functional Addressing PCN2 => Functional Addressing FDDI => Group Addressing
0	1	Illegal value and must not occur.
1	0	A specialized adapter supports group addressing, but the MLID should filter the addresses.
1	1	A specialized adapter supports group addressing, and the MLID is not required to filter the addresses.

See Also

- ♦ ODI Supplement: *Canonical and Noncanonical Addressing* for more information about the receive monitor.

Ctl4_Reserved

Returns *BadCommand*.

Entry State

EAX

The logical board number.

EBX

Subfunction = 4.

Return State

EAX

Has completion code.

Completion Codes (EAX)

0x0FFFFFF81	BadCommnd	This procedure is no longer used.
-------------	-----------	-----------------------------------

Remarks

This procedure is reserved and when called must return *BadCommand*.

Ctl5_MLIDShutdown

Shuts down the LAN adapter.

Entry State

EAX

Has a logical board number.

EBX

Subfunction = 5.

ECX

0 for full shutdown.

Nonzero for partial shutdown

Interrupt

cli or sti (could temporarily be sti).

Call

At process time.

Return State

EAX

Has a completion code.

Interrupt

State is preserved.

Preserved

EBP

Completion Codes (EAX)

0x00000000	Successful
------------	------------

0x0FFFFFF81	BadCommnd
0x0FFFFFF84	Fail

Remarks

`Ctl5_MLIDShutdown` either completely or partially shuts down a LAN adapter.

`Ctl5_MLIDShutdown` is passed a logical board number so that it can reference the LAN adapter to shut down. Be aware that all logical boards that use the LAN adapter will also be shut down.

If `ECX` is 0, `Ctl5_MLIDShutdown` must not only return the hardware interrupt for the LAN adapter, but also all of the data spaces the logical board allocated during initialization. `Ctl5_MLIDShutdown` must also return other tracked resources allocated from the operating system (polling procedure, timer events, etc.). The code for this process is equivalent to that found in the remove MLID procedure. (See the Pseudocode for `RemoveMLID` in Chapter 19, "Remove MLID Procedure")

If `ECX` is set to a nonzero value, `Ctl5_MLIDShutdown` sets the shutdown flag (`MLIDSharingFlags`) in the configuration table of each logical board that is bound to the LAN adapter. It should then set an internal flag indicating that the LAN adapter is shut down. After setting these flags, `Ctl5_MLIDShutdown` disables the board. The MLID can also call `ClearHardwareInterrupt` to unhook the LAN adapter's interrupt, but this is not required.

After the MLID has been partially shut down, the only available IOCTLs are:

- ◆ `Ctl0_LGetMLIDConfiguration`
- ◆ `Ctl1_GetMLIDStatistics`
- ◆ `Ctl5_MLIDShutdown`
- ◆ `Ctl6_MLIDReset`
- ◆ `Ctl5_DriverManagement` (whether this is available depends upon the implementation)

Ctl6_MLIDReset

Reactivate a partially shutdown LAN adapter.

Entry State

EAX

Has a logical board number.

EBX

Subfunction = 6.

Interrupt

cli or sti (could temporarily be sti).

Call

At process time.

Return State

EAX

Has a completion code.

Interrupt

State is preserved.

Preserved

EBP

Completion Codes (EAX)

0x00000000	Successful	The physical card has been reactivated.
0x0FFFFFF84	Fail	An error occurred while attempting to initialize the board.

Remarks

Ctl6_MLIDReset resets or reactivates a LAN adapter partially shut down by *CTL5_MLIDShutdown*. This routine might also check the LAN adapter's hardware to verify that it is functional.

Ctl6_MLIDReset uses the logical board number passed in EAX to reference the LAN adapter. Be aware that this procedure reactivates all the logical boards that were partially shut down with *Ctl5_MLIDShutdown*.

Ctl5_MLIDShutdown checks to see if the LAN adapter is shut down. If the LAN adapter has been partially shut down, this procedure should clear the shutdown flag (*MLIDSharingFlags*) in the configuration table of each logical board bound to the LAN adapter and then reinitialize the LAN adapter. If the LAN adapter's interrupt was unhooked as part of the partial shutdown, the MLID should call *SetHardwareInterrupt* to re-hook the interrupt.

If the LAN adapter has not been shut down at all, this procedure only needs to reinitialize the LAN adapter.

Ctl7_Reserved

Returns BadCommand.

Entry State

EAX

The logical board number.

EBX

Subfunction = 7.

Return State

EAX

Has completion code.

Completion Codes (EAX)

0x0FFFFFF81	BadCommnd	This procedure is no longer used.
-------------	-----------	-----------------------------------

Remarks

This procedure is reserved and when called must return BadCommand.

Ctl8_Reserved

Returns BadCommand.

Entry State

EAX

The logical board number.

EBX

Subfunction = 8.

Return State

EAX

Has completion code.

Completion Codes (EAX)

0x0FFFFFF81	BadCommnd	This procedure is no longer used.
-------------	-----------	-----------------------------------

Remarks

This procedure is reserved and when called must return BadCommand.

Ctl9_SetLookAheadSize

Changes the minimum look-ahead size.

Entry State

EAX

The logical board number.

EBX

Subfunction = 9.

ECX

The requested look-ahead size.

Interrupt

cli or sti (could temporarily be sti).

Return State

EAX

Has a completion code.

Interrupt

State is preserved.

Completion Codes (EAX)

0x00000000	Successful
0x0FFFFFF82	BadParameter

Remarks

Ctl9_SetLookAheadSize allows the protocol stack to dynamically change the MLID's minimum look-ahead size. Protocol stacks call this IOCTL when they initialize.

The MLID compares the request value in ECX to the current look-ahead size in the configuration table *LookAheadSize* field.

If the ECX value is greater, the MLID sets the value in the *LookAheadSize* field equal to the value in ECX. The default *LookAheadSize* is 18 bytes. Maximum value of ECX is 128 bytes.

Byte offset 4Ch through 4Dh in the configuration table is used to store the look-ahead size in ECX

CtlIo_MLIDPromiscuousChange

Enables and disables the MLID's promiscuous mode.

Entry State

EAX

The logical board number.

EBX

Subfunction = 10.

ECX

0 to disable promiscuous mode.

If ECX is nonzero:

- ◆ Bit 0 is set if MAC frames are to be received
- ◆ Bit 1 is set if non-MAC frames are to be received
- ◆ Bit 2 is set if Station Management (SMT) frames are to be received (FDDI only)
- ◆ Bit 3 is set if Remote Multicast Frames are to be received
- ◆ All bits are set if all frames are to be received

EDX

0 to query bits.

Nonzero to change bits.

Interrupt

Disabled, but could be enabled.

Return State

EAX

Has completion code.

ECX

Current bits.

request that the adapter enter promiscuous mode. When promiscuous mode is enabled, the MLID should allow all packets (including bad packets, if possible) to be passed up to the monitor function. Only one monitor function at a time can be registered with an MLID.

Be aware that a monitor function could set the configuration table's `MLIDLookAheadSize` to a value other than the 18-byte default.

The MLID must maintain a counter for each protocol stack that is using promiscuous mode. When a protocol stack requests that the MLID enable promiscuous mode, the MLID should increment the counter. When a protocol stack requests that the MLID disable promiscuous mode, the MLID should decrement the counter. When the counters reach 0, the MLID should disable promiscuous mode on the LAN adapter. This method prevents one protocol stack from disabling promiscuous mode while other protocol stacks are depending on it.

If a protocol stack has called `CtlIo_MLIDPromiscuousChange`, this routine must call `LSLModifyStackFilter`, which, in turn, calls the `ProtocolPromiscuousChange` IOCTL.

If more than one protocol stack is bound to an MLID operating in promiscuous mode, a single protocol stack should not be able to change the MLID out of promiscuous mode. The MLID should only change out of promiscuous mode if all protocol stacks bound to it have also changed out of promiscuous mode.

IMPORTANT: Enabling promiscuous mode slows system performance.

Setting the Remote Multicast Frames bit causes the HSM to activate all multicast frame reception. For example, if the adapter utilizes a hash table for filtering active multicast frames, then the adapter must set the hash table to accept all multicast frames.

Filtering of active multicast entries must be disabled while this bit is set. HSMs that can filter, must also disable filtering while this bit is set.

Multiple bits may be set. Each bit adds to the type of frames that are to be received.

Ctl11_RegisterMonitor

Allows protocol stacks to monitor the packets the adapter is transmitting.

Entry State

EAX

The logical board number.

EBX

Subfunction = 11.

ECX

0 to deregister.

1 to register.

EDI

Pointer to the transmit monitor routine, or zero (0) to deregister.

ESI

Reserved and must be set to zero.

Interrupt

Can be enabled or disabled.

Return State

EAX

Completion code.

Interrupt

State is preserved.

Completion Codes (EAX)

0x00000000	Successful
------------	------------

0x0FFFFFF82	BadParameter	The board number is invalid.
0x0FFFFFF84	Failure	The operation failed.
0x0FFFFFF89	OutOfResources	A monitor is already registered.
0x0FFFFFF89	NoSuchDriver	The MLID is in a partial shutdown state.

Remarks

Ctlll_RegisterMonitor should be used only by line monitoring NLMs that need to see a packet in as near final form as possible before it is transmitted. All other NLMs should implement a transmit prescan stack instead.

Whether or not the MLID is in promiscuous mode has no effect on the transmit monitoring.

Ctl12_Reserved

Returns *BadCommand*

Entry State

EAX

The logical board number.

EBX

Subfunction = 12.

Return State

EAX

Has completion code.

Completion Codes (EAX)

0x0FFFFFF81	BadCommnd	This procedure is no longer used.
-------------	-----------	-----------------------------------

Remarks

This procedure is reserved and when called must return *BadCommand*.

Ctl13_Reserved

Returns BadCommand

Entry State

EAX

The logical board number.

EBX

Subfunction = 13.

Return State

EAX

Has completion code.

Completion Codes (EAX)

0x0FFFFFF81	BadCommnd	This procedure is no longer used.
-------------	-----------	-----------------------------------

Remarks

This procedure is reserved and when called must return BadCommand.

Ctl14_Driver Management

Allows an NLM to manage the MLID

Entry State

EAX

The logical board number.

ESI

Pointer to the management ECB containing the request.

EBX

Subfunction = 14.

EBP

Pointer to the adapter data space.

EBX

Pointer to the frame data space.

Interrupt

cli or sti (could temporarily be sti).

Return State

EAX

The Completion Code.

Interrupt

State is preserved.

Completion Codes (EAX)

0x00000000	Successful	The ECB is relinquished.
0x00000001	Successful	The ECB is queued.

0x0FFFFFF81	BadCommnd	The MLID does not support driver management.
0x0FFFFFF82	BadParameter	The first byte of the ECBRProtocolID field is not above 40h.
0x0FFFFFF88	NoSuchHandle	The Protocol ID is not supported.

Remarks

Ctl14_DriverManagement has been added to allow the MLID to handle all management requests without queuing them in the send queue. This control routine also avoids extracting them from the MLID's packet transmission routine. If the MLID will accept management requests from outside NLMs (for example, HMI or CSL NLMS), it must provide a driver management routine.

The MLID verifies that the management request is valid by checking the ECB *ProtocolID* field. If the first byte is an ASCII character greater than 40h, it is a valid management Protocol ID. The MLID then passes the ECB to the hub's *DriverManagement* routine, if such a routine is available.

The *DriverManagement* routine should scan the whole Protocol ID to verify that the management request is valid before processing it. The Protocol ID differs depending upon which NLM called the MLID. For example, the Protocol ID is HUBMGR for Hub management requests.

If the MLID must respond asynchronously to the management request, it should queue the ECB and return a status of 00000001h in EAX. (If the MLID is responding asynchronously, it must manage its own queue.) When the queued request is complete, the MLID calls the event service routine specified in the ESRAddress field of the ECB as follows:

Example Code

```

mov  esi, PtrToECB      ;get ptr to command ECB
push esi                ;pass on stack
call [esi].ESRAddress   ;call Event Service Routine
add  esp, 4             ;clean up stack

```

Pseudocode for Ctl14DriverManagement

```

IF ProtocolID is not valid
    return FFFFFFF88h

```

```

ENDIF

FOR each object
IF Domain-ObjectID combination is not supported
    ObjectStatus = 2 (not supported)
ELSE IF SET function
    IF legal value
        ObjectStatus = 0 (success)
        set specified object
    ELSE
        ObjectStatus = appropriate error code
    ENDIF
ELSE
    ObjectStatus = 0 (success)
    ObjectValue = get specified object value
ENDIF
ENDIF
continue to next object
return 0

```

Ctl15_Reserved

Returns *BadCommand*

Entry State

EAX

The logical board number.

EBX

Subfunction = 15.

Return State

EAX

Has completion code.

Completion Codes (EAX)

0x0FFFFFF81	BadCommnd	This procedure is no longer used.
-------------	-----------	-----------------------------------

Remarks

This procedure is reserved and when called must return BadCommand.

Ctl16_RemoveNetworkInterface

Allows an application to remove (unload) a logical board.

Entry State

EAX

The logical board number.

EBX

Subfunction = 16.

Interrupt

cli or sti (could temporarily be sti).

Call

At process time.

Return State

EAX

The completion code.

Completion Codes (EAX)

0x00000000	Successful	The logical board has been removed.
0x0FFFFFF81	BadCommnd	The MLID does not support this operation.
0x0FFFFFF82	BadParameter	The specified board number is invalid.
0x0FFFFFF84	Fail	The MLID was unable to remove the logical board.

Remarks

This call should permanently remove from the system all resources associated with the logical board specified in EAX.

Ctl17_ShutdownNetworkInterface

Allows an application to perform a partial shutdown of a logical board

Entry State

EAX

The logical board number.

EBX

Subfunction = 17.

Interrupt

cli or sti (could temporarily be sti).

Call

At process time.

Return State

EAX

The completion code.

Completion Codes (EAX)

0x00000000	Successful	The logical board has been successfully shutdown.
0x0FFFFFF81	BadCommnd	The MLID does not support this operation.
0x0FFFFFF82	BadParameter	The specified board number is invalid.
0x0FFFFFF84	Fail	The MLID was unable to shutdown the logical board.

Remarks

This call should place the logical board specified in *EAX* in a partial shutdown state. A partial shutdown means that the logical board is in a state where an

application can bring the logical board back to a fully functional state by calling *Ctl18_ResetNetworkInterface*.

Ctl18_ResetNetworkInterface

Allows an application to reset a logical board

Entry State

EAX

The logical board number.

EBX

Subfunction = 18.

Interrupt

cli or sti (could temporarily be sti).

Call

At process time.

Return State

EAX

The completion code.

Completion Codes (EAX)

0x00000000	Successful	The logical board has been successfully reset.
0x0FFFFFF81	BadCommnd	The MLID does not support this operation.
0x0FFFFFF82	BadParameter	The specified board number is invalid.
0X0FFFFFF84	Fail	The MLID was unable to reset the logical board.

Remarks

This call will reset the logical board specified in EAX.

Pseudocode for DriverControl

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

Pseudocode

```
IF subfunction number in EBX is not valid
    set EAX to BadCommand
    ret
ENDIF

WHEN subfunction = 0 (Ctl0_GetMLIDConfiguratfon)
    move pointer to MLID configuration table into ESI set EAX to 0
    ret

WHEN subfunction = 1 (Ctl1_GetMLIDStatitics)
    move pointer to MLID statistics table into ESI set EAX to 0
    ret

WHEN subfunction = 2 (Ctl_AddMulticastAddress)
    IF multicast is not supported
        set EAX to BadCommand ret
    ENDIF

    IF address is NOT a valid multicast address
        set EAX to BadParameter ret
    ENDIF

    IF multicast address already exists in the table
        increment EntryUsed (a counter to track how many times this address
        has been added to this board)
        set EAX to 0 ret
    ENDIF

    IF the number of multicast addresses is greater than the maximum
    multicast addresses allowed
        set EAX to OutOfResources ret
    ENDIF

    add multicast address to table
    increment EntityUsed (a counter to track how many times this address has
    been added to this board)
    notify board of new multicast address
    /* there is a separate EntryUsed counter for eachmulticast entry in the
    mulficast table */
```

```

enable multicasting on the board
ret

WHEN subfunction = 3 (Ctl3 DeleteMufficast4address)
  LOOP until all entries in multicast table have been examined or a match
        is found
    IF address matches a previously added multicast address
        decrement EntryUsed (a counter to track how many times this address
            has been added to this board)
    IF EntryUsed is NOT 0
        /* address is still in use */
        set EAX to 0
        ret
    ENDIF

    decrement multicast address count
    notify board that this multicast address has been removed
    IF the number of multicast addresses for this board is 0
        disable multicast reception on the board
    ENDIF

    set EAX to 0
    ret
  ENDIF

  /* no match found in multicast address tables */
  set EAX to ItemNotPresent
ENDLOOP
ret

WHEN subfunction = 4 (Ctl_Reserved)
  set EAX to BadCommand
  ret

WHEN subfunction = 5 (MLIDShutdown)
  IF ECX is 0 (M LID should deregister with the LSL)
    /* complete shutdown */
    set unbinding flag
    call RemoveInstance (see pseudocode for remove MLID procedure)
    set MLIDShutdownState to DOWN (a flag indicating the LAN adapter is
        down)
    set a flag to indicate that the LAN adapter has been completely shut
        down
  ELSE
    /* partial shutdown
    LOOP through all the logical boards using the LAN adapter
        set bit 0 of MLIDSharingFlags to 1
    ENDLOOP

```

```

set a flag to indicate the LAN adapter has been partially shut down
turn off the board
/* the MLID can optionally call ClearHardwareInterrupt at this time */
ENDIF
set EAX to 0
ret

WHEN subfunction = 6 (Ctl6_MLIDReserved)
  IF board has been completely shut down
    ret
  ENDIF

  IF the board has been partially shut down
    /* if you unhooked the interrupt during Ctl5_MLIDShutdown, call
       SetHardwareInterrupt and rehook it */
    LOOP through all the logical boards the LAN adapter supports
      set bit 0 of MLIDSharingFlags to 0
    ENDLOOP

    set a flag to indicate the LAN adapter has been reset and is
      now active
  ENDIF
  initialize the board
  IF
    error initializing board
    call QueueSystemAlert saying the LAN adapter will not reset.
    permanently shut down the adapter.
    set EAX to FFFFFFF84h (FAILURE)
    ret
  ENDIF

  set EAX to 0
  ret

WHEN subfunction = 7 (Ctl7_Reserved)
  set EAX to BadCommand
  ret

WHEN subfunction = 8 (Ctl8_Reserved)
  set EAX to BadComman
  ret

WHEN subfunction = 9 (SetLookAheadSize)
  IF
    ECX <= current LookAheadSize
    set EAX to 0
  ret

```

```

ENDIF

IF
    ECX > 128
    set EAX to BadParameter
    ret
ENDIF

set current LookAheadSize = ECX
LOOP for all logical boards using the LAN adapter
    set ConfigTable.MLIDLookAheadSize a CL
    set EAX to 0
ENDLOOP

ret

WHEN subfunction = 10 (Ctl110_MLIDPromiscuousChange)
    IF
        DriverSupportsPromiscuousBit not set in configuration table
        set EAX to Failure
        ret
    ENDIF

    IF
        EDX = 0 (Query)
        set ECX to PromFlag
        set EAX to 0
        ret
    ENDIF

    IF
        EDX is not equal to 0 (Enable Promiscuous Mode)
        use EDX to calculate new PromFlag, incrementing one counter per bit
        that is set
        ELSE (Disable Promiscuous Mode)
        use EDX to calculate new PromFlag, decrementing one counter per
        bit that is set
        ENDIF

    LOOP through all logical boards using LAN adapter
        set ESI to PromFlag
        set EAX to board number
        set EBX to 5 (ProtocolPromiscuousChange)
        set ECX to -1
        call LSLControlStackFilter
    ENDLOOP

set/clear promiscuous mode on hardware using PromFlag
set EAX to 0

```

```

    ret

WHEN subfunction = 11 (Ctl11_RegisterReceiveMonitor)
    IF ECX is not equal to 0 (Enable Receive Monitor)
        IF a receive monitor is already registered
            set EAX to OutOfResources
            ret
        ENDIF

        set ReceiveMonitor = ESI
        set TransmitMonitor = EDI
    ELSE
        IF (ReceiveMonitor is not equal to ESI) or (TransmitMonitor is not
            equal to EAX)
            set EAX to Failure
            ret
        ENDIF

        clear ReceiveMonitor and TransmitMonitor
    ENDIF

    set EAX to 0
    ret

WHEN subfunction = 12 (Ctl12_Reserved)
    set EAX to BadCommand
    ret

WHEN subfunction = 13 (Ctl13_Reserved)
    set EAX to BadCommand
    ret

WHEN subfunction = 14 (Ctl14_DriverManagement)
    IF MLID does not support Driver Management
        set EAX to BadCommand
        ret
    ENDIF

    IF
        [ESI].RProtocolID < 40h
        set EAX to BadParameter
        ret
    ENDIF

    IF
        [ESI].RProtocolID does not match a service provided by this
        MLID
        set EAX to NoSuchHandle
    ENDIF

```

```

    ret
ENDIF

call the driver management routine
IF
    MLID is going to hold onto the ECB
    set EAX to 1
    ret
ENDIF

Set EAX to 0
ret

WHEN subfunction = 15 (Ctl15_Reserved)
    set EAX to BadCommand
    ret

WHEN subfunction = 16 (Ctl16_RemoveNetworkInterface)
    IF
        operation not supported
        set EAX to Badcommand
        ret
    ENDIF

    IF
        logical board is invalid
        set EAX to BadParameter
        ret
    ENDIF

    IF
        logical board is last one associated with an adapter
        set unbinding flag
        call RemoveInstance
        (see pseudocode for Remove MLID Procedure)
        set MLIDShutdownState to DOWN
        (a flag indicating the LAN adapter is down)
        set a flag to indicate that the LAN adapter
        has been completely shutdown
    ELSE
        call LSLUnBindThenDeRegisterMLID
        IF SMP aware
            call LSLRemoveMutexFromInstance
            call FreeSemiPermMemory to give back frame
            data space memory for logical board
        ENDIF
        set EAX to 0
        ret
    ENDIF

```

```

WHEN subfunction = 17 (Ctl17_ShutdownNetworkInterface)
  IF operation is not supported
    set EAX to BadCommand
    ret
  ENDF

  IF
    logical board is invalid
    set EAX to BadParameter
    ret
  ENDF

  Set bit 0 of logical board's MLIDSharingFlags to 1
  IF
    logical board is last one associated with an adapter
    set a flag to indicate that the Lan adapter
    has been partially shutdown
    turn off the board
  ENDF

  set EAX to 0
  ret

WHEN subfunction = 18 (Ctl18_ResetNetworkInterface)
  IF operation is not supported
    set EAX to BadCommand
    ret
  ENDF

  IF logical board is invalid
    Set EAX to BadParameter
    ret
  ENDF

  set bit 0 of logical board's MLIDSharingFlags to 0
  IF logical board is first one associated with an adapter
    initialize the adapter
    If error initializing board
    call QueueSystemAlert say the LAN adapter will not reset
    permanently shutdown the adapter
    set EAX to Fail
    ret
  ENDF

  set a flag to indicate that the LAN adapter
  has been reset and is not active
ENDIF

```

```
set EAX to 0  
ret
```


21

Operating System Support Routines

This appendix describes the support routines that the NetWare operating system provides to the developer.

The majority of the calls in this appendix are C language routines. The support routine descriptions show the procedure and parameter names in C syntax. Each explanation includes the parameters that must be passed on entry into the routine, the results returned (if any) and an example.

As the examples show, the parameters are placed on the stack in the reverse order of their definition. The calling module has the responsibility of cleaning up the stack on return.

A few calls are assembly language routines. Unlike the APIs in the LSL, the C language operating support routines are not interchangeable with the assembly language routines. In other words, *CancelInterruptTimeCallBack* is an assembly routine and does not exist as a C language routine. Conversely, *AllocateResourceTag* is a C language routine and does not exist as an assembly routine.

As with other NetWare operating system routines written in C, the EBX, EBP, ESI, and EDI registers are preserved. Be aware that this is not the case for the assembly language routines.

These calls reside in the NetWare operating system and, therefore, can be version-specific. This is one of the reasons that we recommend that you use the LAN driver toolkit to develop an MLID. The LAN driver toolkit provides an interface to the operating system routines that is not NetWare version-specific.

The following table provides a summary of the support routines that are provided for the developer:

Table 56 Summary of OS Support Routines

Routine Type	Name of Routine	Summary
Address Conversion Routines	MapAbsoluteAddressToDataOffset	Convert a hardware memory address to a logical address.
	MapDataOffsetToAbsoluteAddress	Convert a logical address to a hardware memory address.
Event Scheduling Routines	ScheduleInterruptTimeCallback	Add an event to the interrupt handler's event list.
	RegisterForEventNotification	Enable calling of the NLM's event routine.
	CancelNoSleepAESProcessEvent	Cancel an event that will not put itself to sleep.
	CancelSleepAESProcessEvent	Cancel an event that may put itself to sleep.
	CancelInterruptTimeCallback	Remove an event from the interrupt handler's event list.
	ScheduleNoSleepAESProcessEvent	Schedule a process that will not put itself to sleep.
	ScheduleSleepAESProcessEvent	Schedule a process that may put itself to sleep.
	CRescheduleLast (NetWare 3 only)	Schedule a task as the last to be executed.
	CYieldWithDelay (NetWare 4 only)	Schedule a task to execute later.
UnRegisterEventNotification	Unhook the event routine.	
Hardware Interaction Routines	DoEndOfInterrupt	EOI the PIC
	SMPDoEndOfInterrupt	EOI the APIC in an SMP system.
	BusInterruptEOI	EOI the APIC in a NetWare 5 system.
	GetHardwareBusType	Get the processor's bus type.
	DoRealModeInterrupt	Performs real mode interrupt such as DOS or BIOS.

Routine Type	Name of Routine	Summary
	SetHardwareInterrupt	Provide the ISR entry point.
	SetSymetricInterrupt	Provide the ISR entry point in an SMP system.
	BusInterruptSetup	Provide the ISR entry point in a NetWare 5 system.
	ReadRoutine	Read firmware.
	ReadEISAConfig	Read the EISA configuration block.(NetWare 4 only)
	DeRegisterHardwareOptions	Release hardware options.
	RegisterHardwareOptions	Reserve hardware options.
	ClearHardwareInterrupt	Release hardware processor interrupt.
	ClearSymetricInterrupt	Release hardware processor information in an SMP system.
	BusInterruptClear	Release hardware processor information in a NetWare 5 system.
Memory Routines	GetRealModeWorkspace	Access memory in real mode.
	Alloc (pre-NetWare 5 only)	Allocate memory.
	NVMAlloc (NetWare 5 only)	Allocate virtual memory.
	NVMAllocIO (NetWare 5 only)	Allocate locked memory in a NetWare 5 system.
	AllocNonMovableCacheMemory (NetWare 3 only)	Allocate memory on 4KB page boundary.
	AllocateMappedPages (NetWare 4 only)	Allocate memory on 4KB page boundaries and memory below 16MB.
	AllocBufferBelow16Meg (NetWare 3.1x only)	Allocate memory for 24-bit host adapters.
	FreeBufferBelow16Meg (NetWare 3.1x only)	Free memory allocated with AllocBufferBelow16Meg.
	ReadPhysicalMemory (NetWare 4 only)	Read memory outside NLM's domain.

Routine Type	Name of Routine	Summary
	Free (NetWare 4 only)	Return allocated memory.
	NVMFree (NetWare 5 only)	Return memory allocated with <i>NMVA/loc</i> .
	DeAllocateMappedPages (NetWare 4 only)	Return memory allocated with <i>AllocateMappedPages</i> .
	WritePhysicalMemory (NetWare 4 only)	Write to memory outside NLMs domain.
	FreeNonMovableCacheMemory (NetWare 3 only)	Return memory allocated with <i>AllocNonMovableCacheMemory</i> .
NLM Interaction Routines	AllocateResourceTag	Allocate an operating system resource tag.
	BindProtocolToBoard	Bind a protocol stack to a board.
	CFindResourceTag	Obtain a resource tag that has already been allocated.
	AddPollingProcedureRTag	Register a polling procedure.
	RemovePollingProcedure	Remove a polling procedure.
Timer Routines	GetCurrentTime	Determine elapsed time.
Operating System Interaction Routines	CVSemaphore	Clear Semaphore.
	GetServerConfigurationType	Determine which operating system or SFT III engine is operating.
	OutputToScreen	Display a message on the monitor.
	GetFileServerMajorVersionNumber (NetWare 4 only)	Get the major version number of the operating system.
	GetFileServerMinorVersionNumber (NetWare 4 only)	Get the Minor version number of the operating system.
	CPSemaphore	Lock semaphore.
	QueueSystemAlert (NetWare 3 only)	Notify the system of a problem.
	NetWareAlert (NetWare 4 only)	Notify the system of a problem.

Routine Type	Name of Routine	Summary
	ParseDriverParameters	Parse command line parameters.
	ImportPublicSymbol (NetWare 4 and higher)	Dynamic (run time) import a public symbol.

AddPollingProcedureRTag

Registers a polling procedure. Generally used by MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
int AddPollingProcedureRTag (  
    void (*Procedure) (void),  
    struct ResourceTagStructure *Rtag);
```

Parameters

Procedure

(IN) Pointer to a polling procedure that is defined by the MLID. This polling procedure will be called at process time.

RTag

(IN) The resource tag acquired in an earlier call to *AllocateResourceTag*. This resource tag was given to the polling procedure and is used only as a pointer to pass to other routines.

Completion Codes (EAX)

0x00000000	Successful	The polling procedure was added.
Nonzero	Failure	The polling procedure was not added.

Remarks

LSLAddPollingProcedure should be used if it is available, instead of *AddPollingProcedureRTag*.

NOTE: Call this routine only at process time.

The MLID calls *AddPollingProcedureRTag* in order to register its polling procedure. The MLID normally only calls this routine during initialization.

After this routine has completed successfully, the operating system continuously calls the procedure specified by the Procedure parameter whenever the server has no other work to do. However, because the specified procedure is not guaranteed to be called within a certain period of time (the operating system may have other work to perform), we strongly recommend that the MLID also include an interrupt backup procedure that allows it to get immediate attention from the operating system.

You should only add one polling procedure per MLID. That is, a single polling procedure should service all the LAN adapters of the same type in the server.

The polling procedure specified as a parameter will be called at process time, with the interrupts disabled and no parameters passed to it. The polling procedure can enable interrupts and destroy all registers.

Example

```
push    PollResourceTag           ;Poll tag
push    OFFSET MyDriverPoll       ;Add poll to OS
call    AddPollingProcedureRtag   ;Tell OS to add poll
add     ESP, 4 * 2
or      EAX, EAX
jnz     ErrorAddingPollProcedure
```

Alloc

Allocates memory. Generally used by Protocol stacks and MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
void *Alloc (  
    LONG numberOfBytes,  
    struct ResourceTagStructure *Rtag);
```

Parameters

numberOfBytes

(IN) The amount of memory (in bytes) to be allocated.

RTag

(IN) The previously obtained resource tag for the allocated memory. The signature to get this tag is "TRLA". For NetWare 5, use "IRLA" to allocate I/O memory less than 4KB.

Completion Codes (EAX)

Nonzero	Successful	EAX contains a pointer to the allocated memory.
0	Fail	The operating system did not allocate memory.

NOTE: Call this routine at either process or interrupt time. Interrupts can be any state and will remain unchanged.

Remarks

Alloc obtains memory for any protocol stack or MLID requirements, such as IOConfigurationStructures or special buffers. *Alloc* receives the amount of memory to be allocated and returns a pointer to the allocated memory. This

routine does not zero or initialize the memory; this must be done by whichever entity (protocol stack or MLID) called it.

In NetWare 4, the memory returned by *Alloc* starts on a dword boundary, is physically contiguous, may not be paged out, and is not guaranteed to exist below 16MB.

This call can be used in NetWare 5 to allocate I/O memory less than 4KB, using the *RTag* signature "IRLA".

AllocateMappedPages

Allocates memory on 4KB page boundaries and also memory below the 16MB boundary. Generally used by MLIDs.

Language: C Language

NetWare Operating System: NetWare 4 and 5 only

Syntax

```
void *AllocateMappedPages (  
    LONG   NumberOf4KPages,  
    LONG   SleepOKFlag,  
    LONG   Below16MegFlag,  
    struct ResourceTagStructure *Rtag,  
    LONG   *SleptFlag);
```

Parameters

NumberOf4KPages

(IN) Has the number of 4KB pages to allocate.

SleepOKFlag

(IN) Set to any nonzero value to allow this call to let other processes execute temporarily. If the *Below16MegFlag* is set to 1, the *SleepOKFlag* flag must also be set. Otherwise, setting this flag is optional. However, setting this flag is advantageous because it allows the operating system to rearrange pages if it is unable to find a continuous buffer.

Below16MB

(IN) Set to 1, if the pages must be physically below the first boundary. This flag is usually only set for intelligent 24-bit adapters that access memory through a bus-mastering device.

RTag

(IN) Has the resource tag previously obtained by the MLID for memory allocation. If the *Below16MegFlag* is set to 1, the MLID must use the *CacheBelow16Meg* memory signature to get the RTag. Otherwise, the MLID uses the same resource tag as it used for the Alloc routine.

SleptFlag

(IN) Has a pointer to a dword to be filled in by this procedure. This dword will act as a flag that indicates if the call went to sleep. If this flag is not needed, set it to 0.

Completion Codes (EAX)

Nonzero	Successful	EAX contains a pointer to the allocated memory.
0	Fail	The operating system did not allocate memory.

Remarks

AllocateMappedPages allocates memory on 4KB (page) boundaries and, optionally, obtains the memory below the 16MB boundary. We recommend that the MLID uses this procedure instead of *AllocBufferBelow16Meg*. The MLID must use *DeAllocateMappedPages* to return this buffer when it shuts down.

NOTE: Call this routine at process time only. Interrupts can be in any state, but they will not be enabled.

NOTE: In NetWare 5, *AllocateMappedPages* is the only way to allocate memory below 16MB.

If this call is used to allocate memory below the 16MB boundary, the default number of cache buffers below 16MB is limited to 16 buffers or 64KB of total memory.

Example

```
push 0 ;Null slept flag
push AllocRTag ;resource tag
push 0 ;no 16 meg boundary concerns
push 1 ;call can sleep if it must
push (size TableStruct + 4095) SHR 12 ;Round up and convert to pages
call AllocateMappedPages ;allocate memory
add esp, (5*4) ;clean up stack
or eax, eax ;buffer returned?
je ErrorAllocatingPages ;jump if not
mov TablePointer, eax ;save pointer
```

See Also

- ◆ DeAllocateMappedPages

AllocateResourceTag

Allocates an operating system resource tag. Generally used by Protocol stacks and MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
struct ResourceTagStructure *AllocateResourceTag (  
    struct LoadDefinitionStructure *LoadRecord,  
    void *ResourceDescriptionString,  
    LONG ResourceSignature);
```

Parameters

LoadRecord

(IN) Has a pointer to the loadable module handle. The handle is passed on the stack to the Initialization routine. The LoadDefinitionStructure is not modified by either the MLID or the protocol stack.

ResourceDescriptionString

(IN) Has a pointer to a null-terminated descriptive text string. for example: StackRTagDescriptionMessage db 'ACME Protocol Stack' 0).

ResourceSignature

(IN) Has a value identifying a specific resource type.

ResourceTagStructure

(OUT) Declares a pointer to a ResourceTagStructure. The NLM does not modify any fields of this structure. It uses this pointer as a parameter called *RTag and passes it to other routines.

Completion Codes (EAX)

Nonzero	Successful	EAX contains a resource tag identifying the specified entry type.
---------	------------	---

0	Fail	The operating system did not allocate a resource tag. The initialization routine should be aborted.
---	------	---

Remarks

AllocateResourceTag allocates an operating system resource tag for a specific category. Some operating system calls use this tag as the tracking identifier to track system resources. In order to get resources, the protocol stacks and the MLIDs must acquire an operating system resource tag for each different type of resource they need allocated.

NOTE: Call this routine only at process time, as it might suspend the process and change the processor state.

Each type of resource has a unique tag. The following resource signatures must be used to identify each resource tag type:

AESProcessSignature	equ	'PSEA'
AllocSignature	equ	'TRLA'
AllocIOSignature	equ	'IRLA'
CacheBelow16MegMemorySignature	equ	'61BC'
CacheNonMovableMemorySignature	equ	'TMNC'
ECBSignature	equ	'SBCE'
EventSignature	equ	'TNVE'
InterruptSignature	equ	'PTNI'
IORegistrationSignature	equ	'SROI'
LSLDefaultStackSignature	equ	'DLSL'
LSLPreScanStackSignature	equ	'PLSL'
LSLStackSignature	equ	'SLSL'
NWManageObjectSignature	equ	'EJBO'
MLIDSignature	equ	'DILM'
LSLTxPreScanStackSignature	equ	'TLSL'

PollingProcedureSignature	equ	'RPLP'
TimerSignature	equ	'RMIT'

Example

```

push  AllocSignature           ;Resource signature
push  OFFSET SPermMemoryRTagMessage ;Resource message
push  [ESP + MyHandle + (2 * 4)] ;Module handle
call  AllocateResourceTag      ;Allocate a tag
add   ESP, (3 - 4)             ;Restore stack
or    EAX, EAX                 ;Allocation successful?
jz    ErrorAllocatingRTag      ;Exit init if not
mov   SemiPermMemoryRTag, EAX  ;Store pointer to tag

```

AllocBufferBelow16Meg

Allocates memory for MLIDs that drive 24-bit host adapters. Generally used by MLIDs.

Language: C Language

NetWare Operating System: NetWare 3.1x only

Syntax

```
LONG AllocBufferBelow16meg (  
    LONG requestedsize,  
    LONG *actualsize,  
    struct ResourceTagStructure *Rtag);
```

Parameters

requestedSize

(IN) Has the number of contiguous bytes of memory requested.

ActualSize

(IN) Has a pointer to a location to place the actual number of allocated bytes of memory. This parameter may be NULL if it is not needed.

Rtag

(IN) Resource tag previously required by the MLID for the memory. The MLID called *ResumeTag* with *CacheBelow16MegMemorySignature* to get this resource tag.

Completion Codes (EAX)

Nonzero	Successful	EAX contains a pointer to the allocated memory.
0	Fail	Memory could not be allocated.

Remarks

Only MLIDs that support 24-bit host adapters running in machines with more than 16MB of memory call *AllocBufferBelow16Meg* to allocate memory. All other MLIDs call *Alloc* to allocate required memory.

NOTE: Call this routine only at process time.

AllocBufferBelow16Meg allocates memory so that the MLID can do I/O operations to or from intermediate buffers below 16MB. The MLID can then copy the data to or from the actual request buffer when it is above the 16MB boundary.

This function returns a pointer in EAX to the allocated buffer. If the function does not allocate any memory, it returns 0. The allocated memory is not initialized.

IMPORTANT: Use these buffers sparingly. The pool of buffers below 16MB is defaulted to 16. The size of each allocated buffer is equal to the cache buffer size. The default cache buffer size on a server is 4KB. For example, if all 16 buffers are allocated using the default cache buffer size, 64KB of memory is allocated. The number of buffers in the pool can be set in the STARTUP.NCF file (up to a maximum of 200).

Example: Set reserved buffers below 16MB = 32

NOTE: MLIDs developed for NetWare 3.1x services must use this call in order to allocate buffers below 16MB. MLIDs written for NetWare 4 servers should call *AllocateMappedPages*.

Example

```
push    MemBelow16RTag           ;pointer to resource tag
push    OFFSET ActualSize        ;amount of memory acquired
push    RequestedSize           ;number of bytes required
call    AllocBufferBelow16Meg
add     esp, 3*4                 ;adjust stack pointer
or      eax, eax                 ;check if successful
jz      ErrorAllocatingMemory    ;jump if error
mov     MyBufferPtr, eax         ;save pointer to allocated memory
```

See Also

- ◆ *Alloc*
- ◆ *AllocateMappedPages*

AllocNonMovableCacheMemory

Allocates memory for MLIDs that need memory starting on a 4KB page boundary. Generally used by protocol stacks and MLIDs.

Language: C Language

NetWare Operating System: NetWare 3 only

Syntax

```
void *AllocNonMovableCacheMemory (  
    LONG size,  
    LONG *actualsize,  
    struct ResourceTagStructure *Rtag);
```

Parameters

size

(IN) Contains the size of the memory to be allocated.

actualsize

(IN) Has a pointer to a variable that this routine will fill in with the actual size. Set this variable to zero if it is not needed.

RTag

(IN) Resource tag previously acquired by the MLID for the memory. The MLID called *AllocateResourceTag* with *CacheNonMovableMemorySignature* to obtain this resource tag.

Completion Codes (EAX)

None.

Remarks

When the MLID shuts down, it must use *FreeNonMovableCacheMemory* to return the buffer.

NOTE: Call this routine only at process time.

Example

```
mov    eax, CacheNonMovableRTag    ;EAX ->resource tag
mov    ecx, [eax+4]                ;EAX->module handle
ord    word ptr[ecx+30h], 80000000h ;sets the Needs 4K Page bit
push   eax                          ;resource tag pointer
push   ActualSize                   ;*ActualSize
push   [ebp].BufferSize             ;Size of memory
call   AllocNonMovableCacheMemory
add    esp, 3*4                     ;clean up the stack
cmp    ActualSize, 0
je     ErrorAllocating4KBuffer      ;jump if unable to allocate memory
```

See Also

- ◆ FreeNonMovableCacheMemory

BindProtocolToBoard

Binds the protocol stack to the board.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
LONG BindProtocolToBoard (  
    LONG ProtocolNumber,  
    LONG BoardNumber,  
    BYTE *Config);
```

Parameters

ProtocolNumber

(IN) Has the protocol stack ID.

BoardNumber

(IN) Has the logical board number of the virtual MLID associated with the IOEngine.

Config

(IN) Has a pointer to a string containing the board's protocol configuration information.

Completion Codes (EAX)

0x00000000	Successful	The specified protocol stack was bound to the specified board.
0x0FFFFFF82	BadParameters	No protocol stack with the specified ProtocolNumber has been registered with the LSL.
0x0FFFFFF83	DuplicateEntry	The specified protocol stack is already bound to the specified board.
0x0FFFFFF84	Fail	The specified protocol stack failed to bind to the specified board.

0x0FFFFFF85	ItemNotPresent	No protocol stack with the specified <i>ProtocolNumber</i> has been registered with the LSL, but protocol stacks with a higher <i>ProtocolNumber</i> might exist.
0x0FFFFFF86	NoMoreItems	No protocol stack with the specified <i>ProtocolNumber</i> has been registered with the LSL, and no protocol stacks with a higher <i>ProtocolNumber</i> exist.
0x0FFFFFFF		If SFT III is implemented and this code is returned, one of the following has happened: the specified protocol stack is already bound to the specified board, or there was insufficient memory to queue the bind request to the other MSEngine.

Remarks

BindProtocolToBoard binds the protocol stack to the logical board.

This function calls the protocol stack's *Ctl2_Bind* entry point which calls *LSLBindStack* to make the connection.

NOTE: If you are writing a NetWare NLM that runs above the protocol stack layer and you need to bind a protocol stack to a board, use this function because it is easier to use and implement in the SFT III environment.

See Also

- ◆ *Ctl2_Bind*
- ◆ *LSLBindStack*
- ◆ *CLSLSLBindStack*

BusInterruptClear

Unhooks a service routine from a bus interrupt. If there are no other service routines hooked to the interrupt, the kernel will mask the associated interrupt hardware, preventing additional interrupts from occurring.

Language: C Language

NetWare Operating System: NetWare 5 only

Thread Context: Non-Blocking

Syntax

```
#include <mpklib.h >

UINT BusInterruptClear (
    INTTAG interruptTag);
```

Parameters

interruptTag

(IN) The tag returned by *BusInterruptSetup*. Specifies the interrupt on which the operation is to be performed.

Completion Codes (EAX)

0	ISUCCESS
5	IERR_HARDWARE_FAILURE
20	IERR_INVALID_INTTAG

BusInterruptEOI

Writes the EOI command to the interrupting hardware allowing another interrupt at the same or lower priority to occur.

Language: C Language

NetWare Operating System: NetWare 5 only

Thread Context: Non-Blocking

Syntax

```
#include <mpklib.h>

UINT BusInterruptEOI (
    INTTAG interruptTag);
```

Parameters

interruptTag

(IN) The tag returned by *BusInterruptSetup*. Specifies the interrupt on which the operation is to be performed.

Completion Codes (EAX)

0	ISUCCESS
5	IERR_HARDWARE_FAILURE
20	IERR_INVALID_INTTAG

BusInterruptSetup

Associates a system bus interrupt with a service routine to be called at interrupt time. This function also unmask the associated interrupt hardware at the system level.

Language: C Language

NetWare Operating System: NetWare 5 only

Thread Context: Non-Blocking

Syntax

```
#include <mpklib.h>

UINT BusInterruptSetup (
    UINT *interruptPtr,
    UINT (*serviceRoutine)(void *serviceRoutineParameter),
    void *serviceRoutineParameter,
    UINT32 flags,    UINT32 hardwareInstanceNumber,
    struct ResourceTagStructure *resourceTag,
    INTTAG *interruptTagPtr);
```

Warning

It is possible for an interrupt to occur before *BusInterruptSetup* has returned. It is the responsibility of the caller to be prepared for an interrupt as soon as this function is invoked.

Parameters

interruptPtr

(IN) The address of a UINT which contains the default interrupt assignment for the device, 0-15 for PC/AT systems. For devices which are NBI aware the OS will use the *hardwareInstanceNumber* to gather additional information about the device, so that if the system supports an alternate interrupt route the hardware corresponding to the new interrupt may be enabled, in which case the UINT pointed to by *interruptPtr* will contain the new interrupt assignment.

serviceRoutine

(IN) Points to the start address of the service routine to be called at interrupt time. The service routine is defined as follows.

```
UINT serviceRoutine(void * serviceRoutineParameter);  
0 - INTERRUPT_SERVICED  
1 - INTERRUPT_NOT_SERVICED
```

return

(IN) If the interrupt is not serviced the OS calls the next service routine on a shared interrupt. If there are no other shared interrupts to call a spurious interrupt event is recorded.

serviceRoutineParameter

(IN) A unique parameter to be passed on the stack to the *serviceRoutine* when an interrupt occurs. The use of this parameter by the service routine is not specified. However, in the case where there is a common service routine for a number of device/drivers the *serviceRoutineParameter* might be used to quickly identify the correct device to service. If a common *serviceRoutine* is used multiple times on a shared interrupt, the *serviceRoutineParameter* must be unique for each occurrence of the *serviceRoutine*.

flags

(IN)

BitValue	Meaning
0	0=Do not share this interrupt with other devices or service routines. 1=Set this bit if the interrupt can be shared.
1-28	Reserved (Set to 0)
29	0=The interrupt service routine is not MP aware. Deliver this interrupt to the boot processor only. 1=The interrupt service routine is MP aware. This interrupt may be delivered to any processor as the OS sees fit.
30	0 =Do not save the floating point state before servicing this interrupt. 1=Save floating point state before servicing this interrupt. Set this bit if the interrupt service routine changes the state of the floating point unit.

BitValue	Meaning
31	Reserved (Set to 0)

hardwareInstanceNumber

(IN) A unique number that NetWare Bus Interface (NBI) aware drivers use. If the device is not NBI aware *hardwareInstanceNumber* must be set to -1 or 0xFFFFFFFF indicating that this is not an NBI aware device. (For additional information on *hardwareInstanceNumber* please refer to the NBI specification.)

Legacy ISA devices use -1 for the *hardwareInstanceNumber*.

resourceTag

(IN) The caller's resource tag for this interrupt obtained by calling *AllocateResourceTag()* with the resource signature equal to InterruptSignature, 0x50544E49.

interruptTagPtr

(IN) The address of an INTTAG.

interruptPtr

(OUT) On return the UINT pointed to by *interruptPtr* contains the interrupt assignment for the device.

interruptTagPtr

(OUT) On return the INTTAG pointed to by *interruptTagPtr* contains a tag or handle for the interrupt which is used in subsequent calls to bus interrupt related APIs.

Completion Codes (EAX)

0	ISUCCESS
1	IERR_INVALID_PARAMETER
2	IERR_INTERRUPT_NOT_SHAREABLE
3	IERR_SHARING_LIMIT_EXCEEDED

4	5IERR_REAL_MODE_SHARING_LIMIT_EXCEEDED
5	IERR_HARDWARE_FAILURE
6	IERR_HARDWARE_ROUTE_NOT_AVAILABLE
9	IERR_INTERRUPT_IS_SHAREABLE
21	IERR_INVALID_INTERRUPT
22	IERR_INVALID_FLAGS
23	IERR_INVALID_HARDWARE_INSTANCE_NUMBER
24	IERR_INVALID_RESOURCE_TAG
41	IERR_INTERRUPT_NOT_ALLOCATED

CancelInterruptTimeCallback

Cancels a callback event. Generally used by Protocol stacks and MLIDs.

Language: Assembly Language

NetWare Operating System: Not version specific

Entry State

EDX

Has a pointer to the timer node to be canceled.

Interrupts

Are disabled.

Return State

Interrupts

Preserved and never changed.

Preserved

All registers but EDI and ESI.

Completion Codes (EAX)

None.

Remarks

CancelInterruptTimeCallback cancels a callback event that the protocol stack previously scheduled using ScheduleInterruptTimeCallback. This procedure removes the specified timer node from the list of events that the timer interrupt handler will call.

Remember that ScheduleInterruptTimeCallback must be rescheduled after every callback, and CancelInterruptTimeCallback is usually only used to cancel a callback if the protocol stack or MLID is unloaded before the callback occurs.

Example

```
push    ESI                If needed to be preserved
push    EDI                If needed to be preserved
cli
mov     EDX, OFFSET My TimerNode  Pointer to TimerDataStructure
call    CancelInterruptTimeCallBack  Cancel TimerCallBack
sti
pop     EDI                Restore to original value
pop     ESI                Restore to original value
```

See Also

- ◆ `ScheduleInterruptTimeCallBack`

CancelNoSleepAESProcessEvent

Cancels a no-sleep AES event. Generally used by Protocol stacks and MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
void CancelNoSleepAESProcessEvent (  
    struct AESProcessStructure *EventNode);
```

Parameters

EventNode

(IN) Has a pointer to the AESProcessEventStructure to be canceled.

Completion Codes (EAX)

None.

Remarks

NOTE: Call this routine at either process or interrupt time.

CancelNoSleepAESProcessEvent cancels the no-sleep *AESProcessEventStructure* pointed to by *EventNode*.

CancelNoSleepAESProcessEvent removes the specified AES event from the list of events that is to be called by the AES no-sleep process.

Remember that *ScheduleNoSleepAESProcessEvent* must be rescheduled every time it calls the specified process, and *CancelNoSleepAESProcessEvent* is usually only used to cancel a process event if the protocol stack or MLID is unloaded before the process executes.

Interrupts can be in any state when this routine is called, and the interrupt state is preserved when the routine returns.

The AESProcessStructure is defined below:

```
struct AESProcessStructure struc
```

```

ALink                dd ? ;used by the operating system.
AWakeUpDelayAmount   dd ? ;filled out by caller, won't be changed
AwakeUpTime          dd ? ;used by operating system
AProcessToCall       dd ? ;filled out by caller, won't be changed
ARTag                dd ? ;filled out by caller, won't be changed
AOldLink             dd ? ;used by operating system
AESProcessStructure  ends

```

IMPORTANT: The AESProcessStructure must be in static memory and available long-term.

Example

```

cli
push  OFFSET MyAESEventStructure    ;Address of AES structure
call  CancelNoSleepAESProcessEvent ;No further event callbacks
add   ESP, 4                        ;Adjust stack pointer
sti

```

See Also

- ◆ ScheduleNoSleepAESProcessEvent

CancelSleepAESProcessEvent

Cancels a sleep AES event. Generally used by Protocol stacks and MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
void CancelSleepAESProcessEvent (  
    struct AESProcessStructure *EventNode);
```

Parameters

EventNode

(IN) Has a pointer to an AESProcessStructure to be canceled.

Completion Codes (EAX)

None.

Remarks

NOTE: Call this routine at process time or interrupt time.

CancelSleepAESProcessEvent cancels the sleep *AESEventStructure* pointed to by *EventNode*. The specified event is removed from the list of events that is to be called by the AES sleep process.

Remember that *ScheduleSleepAESProcessEvent* must be rescheduled, every time it calls the specified process, and *CancelSleepAESProcessEvent* is usually only used to cancel a process event if the protocol stack or MLID is unloaded before the process executes.

The interrupts can be in any state when this routine is called. The interrupt state is preserved.

IMPORTANT: The AESProcessStructure must be in static memory and available long-term.

Example

```
cli
push  OFFSET AESEventStructure    ;Address of AES structure
call  CancelSleepAESProcessEvent ;No further event callbacks
add   ESP, 1*4                    ;Adjust stack pointer
sti
```

See Also

- ◆ ScheduleSleepAESProcessEvent
- ◆ CancelNoSleepAESProcessEvent

CFindResourceTag

Obtains a predefined resource tag. Generally used by Protocol stacks and MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
struct ResourceTagStructure *CfindResourceTag (  
    struct LoadDefinitionStructure *LoadRecord,  
    LONG *ResourceSignature);
```

Parameters

LoadRecord

(IN) Has a pointer to the module handle. The LoadDefinitionStructure is not modified by either the MLID or the protocol stack. This is a pointer that is passed to other routines.

ResourceSignature

(IN) Has the value identifying a specific resource type.

ResourceTagStructure

(OUT) Declares a pointer to a ResourceTagStructure. The NLM does not modify any fields of this structure. It uses this pointer as a parameter called RTag and passes it to other routines.

Completion Codes (EAX)

Nonzero	Successful	EAX contains a pointer to the specified resource tag.
0	Fail	The resource tag could not be found.

Remarks

CFindResourceTag gets a resource tag that has been defined using *AllocateResourceTag*.

Example

```
push    LSLStackSignature           ;Resource signature
push    [ESP + MyHandle + 4]        ;Module handle
call    CfindResourceTag            ;Find the Rtag
add     ESP, 2 * 4
or      EAX, EAX                    ;Was it found?
jnz     Don'tNeedToAllocateResourceTag ;Allocate a resource tag
```

See Also

- ◆ [AllocateResourceTag](#)

ClearHardwareInterrupt

Releases an interrupt allocated by SetHardwareInterrupt. Generally used by MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
LONG ClearHardwareInterrupt (  
    LONG hardwareInterruptLevel,  
    void (*InterruptProcedure) ( void ));
```

Parameters

hardwareInterruptLevel

(IN) Has the IRQ level of the hardware interrupt.

InterruptProcedure

(IN) Has a pointer to the interrupt procedure.

Completion Codes (EAX)

0x00000000	Successful	The hardware interrupt was removed successfully.
Nonzero	Fail	The routine did not clear the interrupt vector, either because the parameters were invalid or because it could not find the vector.

Remarks

NOTE: Call this routine only at process time. Interrupts must be disabled.

ClearHardwareInterrupt releases a processor hardware interrupt that was previously allocated by *SetHardwareInterrupt* for a LAN adapter.

NOTE: SMP-aware MLIDs should use *ClearSymmetricInterrupt* instead of this call.

MLIDs usually call *ClearHardwareInterrupt* either when they are unloading or when their initialization procedure fails after an interrupt has been set.

Example

```
cli
push OFFSET MyInterruptHandler ;interrupt entry
push InterruptLevel           ;interrupt number
call ClearHardwareInterrupt
add esp, (2 * 4)              ;restore stack sti
```

See Also

- ◆ SetHardwareInterrupt
- ◆ ClearSymmetricInterrupt
- ◆ SetSymetricInterrupt

ClearSymmetricInterrupt

Releases an interrupt allocated by SetSymmetricInterrupt. Generally used by MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
LONG ClearSymmetricInterrupt (  
    LONG hardwareInterruptLevel,  
    void (*ServiceRoutine));
```

Parameters

hardwareInterruptLevel

(IN) The IRQ level of the hardware interrupt.

ServiceRoutine

(IN) Pointer to the interrupt procedure.

Completion Codes (EAX)

zero	Successful	The hardware interrupt was removed successfully
Nonzero	Fail	The routine did not clear the interrupt vector, either because the parameters were invalid or because it could not find the vector.

Remarks

NOTE: Call this routine only at process time. Interrupts must be disabled.

SMP aware MLIDs use this routine to release an interrupt allocated with SetSymmetricInterrupt.

In the NetWare SMP environment, MLIDs must use *SetSymmetricInterrupt* and *ClearSymmetricInterrupt* in place of *SetHardwareInterrupt* and *ClearHardwareInterrupt*.

NetWare 5 uses *BusInterruptSetup* and *BusInterruptClear*.

See Also

- ◆ SetSymmetricInterrupt
- ◆ SetHardwareInterrupt
- ◆ ClearHardwareInterrupt
- ◆ BusInterruptSetup
- ◆ BusInterruptClear

CPSemaphore

Locks the semaphore. Generally used by MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
void CPSemaphore (  
    semaphoreNumber);
```

Parameters

semaphoreNumber

(IN) Pointer to the semaphore in the *GetRealModeWorkSpace* structure.

Completion Codes (EAX)

None.

Remarks

IMPORTANT: The function that calls *CPSemaphore* has been superceded by the NBI function: *GetCardConfigInfo*. Therefore, this function will not be supported in future releases of NetWare and should not be used any longer.

The MLID uses *CPSemaphore* to lock the real mode workspace when making an EISA BIOS call (see Appendix E, "Writing Protocol Stacks for NetWare SFT III").

Call this routine with interrupts disabled. Interrupts must remain disabled.

MLIDS for NetWare 4 servers should call *ReadEISAConfig* to read the EISA configuration.

This function should not be used to handle critical sections local to the MLID.

Example

```
push    WorkspaceSemaphore ;load semaphore
```

```
call  CPSemaphore          ;lock workspace for our use
add   esp, (2*4)
```

See Also

- ◆ CVSemaphore
- ◆ ReadEISAConfig
- ◆ Appendix E, "Writing Protocol Stacks for NetWare SFT III"

CRescheduleLast

Schedules a task to be executed last. Generally used by Protocol stacks and MLIDs.

Language: C Language

NetWare Operating System: NetWare 3 only

Syntax

```
void CRescheduleLast ( void );
```

Parameters

None

Completion Codes (EAX)

None.

Remarks

NOTE: Call this routine only at process time, as it suspends the process.

CRescheduleLast places a task last on the list of active tasks to be executed. This causes the active task to relinquish control of the CPU and allow other processes to execute. Processes that occupy the CPU for a significant amount of time should use *CRescheduleLast* to allow the vital server processes to function. *CRescheduleLast* is normally used in conjunction with *ScheduleSleepAESProcessEvent* and should only be used in the initialization procedure or the removal procedure. *Example*

Example

```
call CRescheduleLast ; will regain control some undefined time later
```

CYieldWithDelay

Reschedules the calling task to execute later to allow other tasks to execute first. Generally used by protocol stacks and MLIDs.

Language: C Language

NetWare Operating System: NetWare 4 only

Syntax

```
void CYieldWithDelay ( void );
```

Parameters

None.

Completion Codes (EAX)

None.

Remarks

NOTE: Call this routine only at process time, as it suspends the process.

CYieldWithDelay places a task last on the list of active tasks to be executed. This causes the active task to relinquish control of the CPU and allow other processes to execute. Processes that occupy the CPU for a significant amount of time should use *CYieldWithDelay* to allow the vital server processes to function. *CYieldWithDelay* is normally used in conjunction with *ScheduleSleepAESProcessEvent* and should only be used in the initialization procedure or the removal procedure. *Example*

Example

```
call    CYieldWithDelay    ; will regain control some undefined time later
```

CVSemaphore

Unlocks the semaphore. Generally used by MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
void CVSemaphore (  
    semaphoreNumber);
```

Parameters

semaphoreNumber

(IN) Pointer to the semaphore.

Completion Codes (EAX)

None.

Remarks

IMPORTANT: The function that calls *CVSemaphore* has been superceded by the NBI function: *GetCardConfigInfo*. Therefore, this function will not be supported in future releases of NetWare and should not be used any longer.

The MLID calls *CVSemaphore* to clear the semaphore it set with *CPSemaphore*. This routine returns with interrupts enabled.

Normally, the MLID uses *CVSemaphore* when it has finished making an EISA BIOS call, and it can allow other processes to use the workspace (see Appendix E, "Writing Protocol Stacks for NetWare SFT III").

Example

```
push    WorkspaceSemaphore    ;pass semaphore  
call    CVSemaphore           ;Unlock workspace  
call    esp, (1*4)
```

DeAllocateMappedPages

Returns memory allocated with *AllocateMappedPages* . Generally used by MLIDs.

Language: C Language

NetWare Operating System: NetWare 4 only

Syntax

```
void DeAllocateMappedPages (  
    void *Memory);
```

Parameters

Memory

(IN) Pointer to the buffer to free.

Completion Codes (EAX)

None.

Remarks

DeAllocateMappedPages returns memory buffers that were previously allocated on 4KB page boundaries using *AllocateMappedPages*.

Example

```
push    TablbPdinter      ;Pointer to buffer  
call    DeAllocateMappedPages ;deallocate memory  
add     esp, 1*4          ;clean up stack
```

See Also

- ◆ *AllocateMappedPages*

DeRegisterHardwareOptions

Releases reserved hardware options. Generally used by MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
void DeRegisterHardwareOptions (  
    struct IOConfigurationStructure *IOConfig);
```

Parameters

IOConfig

(IN) Pointer to the LAN adapter's corresponding IOConfigurationStructure (starting at the MLIDLink field of the configuration table).

Completion Codes (EAX)

None.

Remarks

DeRegisterHardwareOptions releases the previously reserved hardware options specified in a particular LAN adapter's IOConfigurationStructure starting at the MLIDLink field of the configuration table).

NOTE: Call this routine at process time. Interrupts must be disabled.

DeRegisterHardwareOptions will usually be called from the MLID's remove procedure (or possibly from *Ctl_5MLIDShutdown*, if the control procedure is doing a complete shutdown).

Example

```
cli  
push offset DriverConfig.MLIDLink ;Pointer to IOConfigurationStructure  
call DeRegisterHardwareOptions ;DeRegisterhardware  
add ESP, 4
```

sti

DisableHardwareInterrupt

Masks off the specified interrupt line on the PIC. Generally used by MLIDs.

Language: Assembly Language

NetWare Operating System: Not version specific

Entry State

ECX

Contains the interrupt level.

Interrupts

Are disabled.

Call

At process or interrupt time.

Return State

Interrupts

Are unchanged.

Preserved

All registers except for EAX and EDX.

Completion Codes (EAX)

None.

Remarks

DisableHardwareInterrupt masks off the ECX- specified interrupt request line on the programmable interrupt controller (PIC), preventing the LAN adapter from interrupting the operating system. The MLID does not need to use this routine if the LAN adapter provides a command that disables the interrupt line.

NOTE: We recommend disabling interrupts at the LAN adapter if possible, because disabling interrupts at the PIC is slow.

Example

```
DriverISR  proc
mov     ecx, InterruptLevel
call   DisableHardwareInterrupt
call   DoEndOfInterrupt
.
.   (Service the adapter)
.
mov     ecx, InterruptLevel
call   EnableHardwareInterrupt
call   LSLServiceEvents           ;Let LSL unqueue returned
ret
DriverISR  endp
```

DoEndOfInterrupt

EOIs the PIC. Generally used by MLIDs.

Language: Assembly Language

NetWare Operating System: Not version specific

Entry State

ECX

Contains the interrupt level.

Interrupts

Are disabled.

Call

At process or interrupt time.

Return State

Interrupts

Are unchanged.

Preserved

All registers except for EAX

Completion Codes (EAX)

None.

Remarks

DoEndOfInterrupt issues the appropriate End of Interrupt (EOI) commands to one or both Programmable Interrupt Controllers (PICs).

If the level is assigned to a secondary PIC, an EOI will be issued to the secondary PIC, then to the primary PIC. Using this routine (instead of hard-coding EOIs in the MLID) allows flexibility when an MLID runs on several

platforms. It also ensures that the EOI is executed correctly in the event of future operating system changes.

NOTE: SMP aware MLIDs must call *SMPDoEndOfInterrupt* instead of this routine, and NetWare 5 MLIDs must call *BusInterruptEOI* instead of this routine.

Example

```
DriverISR  proc
mov     ecx, InterruptLevel
call   DisableHardwareInterrupt
call   DoEndOfInterrupt
.
.      (Service the adapter)
.
mov     ecx, InterruptLevel
call   EnableHardwareInterrupt
call   LSLServiceEvents           ;Let LSL unqueue returned
ret
DriverISR  endp
```

DoRealModeInterrupt

Perform real mode interrupts. Generally used by MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
LONG DoRealModeInterrupt (  
    void *InputParameters,  
    void *OutputParameters);
```

Parameters

InputParameters

(IN) Pointer to a filled-in InputParameterStructure (defined in Remarks below).

OutputParameters

(IN) Pointer to a filled-in OutputParameterStructure (defined in Remarks below).

Completion Codes (EAX)

0x00000000	Successful	The zero flag is set to 1, if the interrupt vector is called.
0x00000001	Fail	The zero flag is cleared to 0, if the interrupt vector is no longer available because DOS has been removed.

Remarks

This function should not be used if there is another way to accomplish the same result.

In the future, this function will no longer be supported.

On current versions of NetWare, use NBI calls to do the things that *DoRealModeInterrupt* used to do.

NOTE: Call this routine only at process time. This routine could enable interrupts and put the calling process to sleep.

The MLID calls *DoRealModeInterrupt* to perform real mode interrupts, such as BIOS interrupts.

The input and output parameter structures are defined below:

InputParameterStructure

```
InputParameterStructure struc
IAXRegister    dw?
IBXRegister    dw?
ICXRegister    dw?
DXRegister     dw?
IBPRegister    dw?
SIRegister     dw?
IDSRegister    dw?
IESRegister    dw?
IntNumber      db?
InputStructure ends
```

OutputParameterStructure

```
OutputParameterStructure  struc
OAXRegister    dw ?
OBXRegister    dw ?
OCXRegister    dw ?
ODXRegister    dw ?
OBPRegister    dw ?
OSIRegister    dw ?
ODSRegister    dw ?
OESRegister    dw ?
Oflags         dw ?
OutputStructure ends
```

Example

NOTE: The input parameter structure has already been initialized.

```
push OFFSET OutputParameters
push OFFSET InputParameters
call DoRealModeInterrupt
add esp, 2 * 4
cmp eax, 0
jne IntNotValidErrorExit
```

See Also

- ◆ ReadEISAConfig
- ◆ Appendix E, "Writing Protocol Stacks for NetWare SFT III" for information on reading the EISA configuration

EnableHardwareInterrupt

Enables the PICs interrupt line. Generally used by MLIDs.

Language: Assembly Language

NetWare Operating System: Not version specific

Entry State

ECX

Contains the interrupt level.

Interrupts

Are disabled.

Call

At process or interrupt time.

Return State

Interrupts

Are unchanged.

Preserved

All registers except for EAX and EDX

Completion Codes (EAX)

None.

Remarks

EnableHardwareInterrupt enables the LAN adapter's interrupt line on the Programmable Interrupt Controller (PIC), if it was disabled using *DisableHardwareInterrupt*.

See Also

- ◆ `DisableHardwareInterrupt`

Free

Returns memory allocated with *Alloc* . Generally used by protocol stacks and MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
void Free (  
    void *address);
```

Parameters

address

(IN) Pointer to the previously allocated memory to be released.

Completion Codes (EAX)

None.

Remarks

Protocol stacks and MLIDs call *Free* to release memory previously allocated with the *Alloc* function. This memory could have been used for any purpose.

Call this routine at process time or interrupt time. Interrupts can be in any state. The interrupt state is preserved.

Protocol stacks and MLIDs should call *Free* to release all memory that they allocated during initialization. Protocol stacks should call this routine as an essential part of cleaning up before exit.

Example

```
push MyMemoryBlock    ;place pointer to memory on stack  
call Free  
add esp, 1*4          ;restore stack
```

See Also

- ◆ Alloc

FreeBufferBelow16Meg

Returns memory allocated with *AllocBufferBelow16Meg*. Generally used by MLIDs.

Language: C Language

NetWare Operating System: NetWare 3 only

Syntax

```
void FreeBufferBelow16Meg (  
    void *MemoryBuffer);
```

Parameters

MemoryBuffer

(IN) Pointer to the memory to be returned.

Completion Codes (EAX)

None.

Remarks

The MLID calls *FreeBufferBelow16Meg* to return the memory it previously allocated for Bus Master or DMA I/O. This was memory that was required to be below 16MB. The MLID must return memory as an essential part of cleaning up before it exits.

NOTE: Call this routine at process time or interrupt time.

Example

```
push eax                ;pointer to memory  
call FreeBufferBelow16Meg  
lea esp, [esp+4]      ;adjust stack pointer
```

See Also

- ◆ `AllocateMappedPages`

FreeNonMovableCacheMemory

Returns memory allocated with *AllocNonMovableCacheMemory*. Generally used by Protocol stacks and MLIDs.

Language: C Language

NetWare Operating System: NetWare 3 only

Syntax

```
void FreeNonMovableCacheMemory (  
    void *memoryAddress);
```

Parameters

memoryAddress

(IN) pointer to the address of the memory to be freed.

Completion Codes (EAX)

None.

Remarks

NOTE: Call this routine only at process time.

The MLID calls *FreeNonMovableCacheMemory* to return the memory it previously allocated using *AllocNonMovableCacheMemory*. The MLID must return memory as an essential part of cleaning up before it exits.

Example

```
push [ebp].AdapterPagedBuffers  
call FreeNonMovableCacheMemory  
lea esp, [esp+4]
```

See Also

- ◆ [AllocNonMovableCacheMemory](#)

GetCurrentTime

Determines the current relative time. Generally used by Protocol stacks and MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
LONG GetCurrentTime ( void );
```

Parameters

None.

Completion Codes (EAX)

Nonzero	Successful	EAX contains a LONG integer that contains the number of clock ticks (1 /1 8 second, or 56.6 milliseconds) since the server was last loaded and began execution.
---------	------------	---

Remarks

GetCurrentTime determines the current relative time. The protocol stack can use this time to determine the elapsed time for stack-related activities. The current time value less the value returned at the start of an operation is the elapsed time in 1/18 second clock ticks. This timer requires more than 7 years to roll over, allowing it to be used for a variety of elapsed time comparisons.

Example

```
mov  EDX, [EBP].Command      ;Let LAN adapter attempt to
mov  AL, NICTransmit         ;Transmit packet again
out  DX, AL
call GetCurrentTime          ;EAX = current time
mov  [EBP].TxStartTime, EAX  ;Set new timeout time
```

GetFileServerMajorVersionNumber

Gets the major version number of the file server. Generally used by MLIDs.

Language: C Language

NetWare Operating System: NetWare 4 and 5 only

Syntax

```
LONG GetFileServerMajorVersionNumber ( void );
```

Parameters

None.

Completion Codes (EAX)

Nonzero	Successful	EAX contains the major version number of the file server.
---------	------------	---

Remarks

GetFileServerMajorVersionNumber returns the major version number of the file server. NLMs use this routine to determine the file server version and to choose the path of execution.

See Also

- ◆ [GetFileServerMinorVersionNumber](#)

GetFileServerMinorVersionNumber

Gets the minor version number of the file server. Generally used by MLIDs.

Language: C Language

NetWare Operating System: NetWare 4 and 5 only

Syntax

```
LONG GetFileServerMinorVersionNumber ( void );
```

Parameters

None.

Completion Codes (EAX)

Nonzero	Successful	EAX contains the minor version number of the file server.
---------	------------	---

Remarks

GetFileServerMinorVersionNumber returns the major version number of the file server. NLMs use this routine to determine the file server version and to choose the path of execution.

See Also

- ◆ [GetFileServerMajorVersionNumber](#)

GetHardwareBusType

Gets the processor bus type. Generally used by MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
LONG GetHardwareBusType ( void );
```

Parameters

None.

Completion Codes (EAX)

00000000h	1/0 bus is ISA (industry Standard Architecture).
00000001h	1/0 bus is Micro Channel Architecture.
00000002h	1/0 bus is EISA (Extended Industry Standard Architecture).

Remarks

IMPORTANT: *GetHardwareBusType* is obsolete and will not be supported in future releases of NetWare. This function should not be used. Use the NBI call *GetBusType* instead.

GetHardwareBusType returns a value that indicates the processor bus type.

GetHardwareBusType allows one MLID to be written so that it can be used for LAN adapters with different bus types.

Call this routine at process time or interrupt time. The interrupt state is preserved.

NOTE: These values are different than those used in the *MLIDFlags* field of the configuration table.

Example

```
call GetHardwareBusType  
mov  HardwareBusType, EAX
```

GetNumberOfLANs

Returns the maximum number of LAN logical boards that may be present in a system.

Language: Assembly Language

NetWare Operating System: Not version specific

Entry State

Interrupts

May be in any state.

Call

At process time or interrupt time.

Return State

EAX

The maximum number of logical boards which may be present in the system

Completion Codes (EAX)

None.

Remarks

Note that this function does not return the number of logical boards that are present in the system, but the maximum number that may be present in the system.

GetRealModeWorkspace

Accesses real mode memory. Generally used by MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
void GetRealModeWorkspace (  
    struct SemaphoreStructure *workSpaceSemaphore,  
    LONG *protectedModeAddressOfWorkSpace,  
    WORD *realModeSegmentOfNorkSpace,  
    WORD *realModeOffsetOfWorkSpace,  
    LONG *workSpaceSizeInBytes);
```

Parameters

workSpaceSemaphore

(IN) Pointer to the operating system semaphore structure.

protectedModeAddressOfWorkSpace

(OUT) Protected mode address of where to read the information.

realModeSegmentOfWorkSpace

(OUT) Pointer to the real mode segment of workspace.

realModeOffsetOfWorkSpace

(OUT) Pointer to the real mode offset in the workspace segment.

workSpaceSizeInBytes

(OUT) Pointer to the size of the workspace.

Completion Codes (EAX)

None.

Remarks

IMPORTANT: *GetRealModeWorkSpace* is obsolete and will not be supported in future releases of NetWare. This function should not be used. Use an NBI function instead.

The MLID uses *GetRealModeWorkSpace* with *DoRealModeInterrupt* in order to access memory in real mode.

The 386 and 486 processors allow MLIDs to run in protected mode and do not allow direct access to BIOS based information. The *DoRealModeInterrupt* call allows the MLID to access the BIOS and get data from it. (See Appendix E, "Writing Protocol Stacks for NetWare SFT III".) *DoRealModeInterrupt* turns on the system interrupts and executes in a critical section; therefore, the MLID calls the semaphore routines, *CPSemaphore* and *CVSemaphore*, in order to keep other processes out of the workspace.

The MLID must provide the following structure. When it calls *GetRealModeWorkSpace*, the MLID passes a pointer to this structure. *GetRealModeWorkSpace* then fills in the structure fields with the appropriate values.

The semaphore structure is as follows:

```
WorkspaceSemaphore          dd  0
WorkspaceProtectedModeAddress dd  0
WorkspaceRealModeSegment    dw  0
WorkspaceRealModeOffset     dw  0
WorkspaceSize               dd  0
```

Example

```
;*****
; Get real mode workspace
;*****

push OFFSETWorkSpaceSize          ;size of workspace
push OFFSETWorkSpaceRealModeOffset ;real mode offset into segment
push OFFSETWorkSpaceRealModeSegment ;real mode segment address
push OFFSETWorkSpaceProtectedModeAddress ;address in protected mode
push OFFSETWorkSpaceSemaphore     ;semaphore
call GetRealModeWorkSpace         ;call OS to fill in information
add esp, [5 * 4]                  ;clean up stack

;*****
```

```

;Lock the workspace
;*****

push WorkspaceSemaphore          ;load semaphore
call CPSemaphore                 ;lock workspace for our use
add esp, [1*4]                   ;clean up stack

;*****
;Setup and execute real mode interrupt
;*****

movzx eax, WorkspaceRealModeSegmerit ;get WorkSpWo segment
movzx ebx, WorkspaceRealModeOffset   ;get offset into segment
mov  el, SlotToReadConfiguration     ;get slot number
xor  ch, ch                           ;read first block
mov  esi, OFFSET Inputparms          ;point to input am
mov  [esi].IAXRegister, OD801h       ;EISA read conflourabon
mov  [esi].ICXRegister, ex           ;slot and data block
mov  [esi].ISIRegister, bx           ;offset of DosWorkArea
mov  [esi].IDSRegister, ax           ;segment of prkarea
mov  [esi].IIntNumber, 15h           ;irderrupt number
push OFFSETOutputParms              ;pt at outputrs
push OFFSETInputParms               ;pt at input registers
call DoRealModeinterrupt            ;tell OS to do it
lea  esp, [esp + 2 * 4]              ;clear up stack
cmp  eax, 0                          ;did the OS do the
jne  lntNotValidErrorExit           ;int correctly
cmp  byte ptr OutputParms.OAXRegister + 1,0
                                        ;Bios Int 15 return
jne  lntNotValidErrorExit           ;successful ?
mov  esi, WorkspaceProtectedModeAddress ;load pointer to data
movzx ecx, BYTE PTR [esi + INTERRUPTOFFSEII
                                        ;get lnt if any
and  el, ISOLATEINTMASK              ;isolate interrupt level
jecxz NoAddinterrupt                ;if none skip add
mov  SaveInterrupt, cl                ;save Interrupt for later

;*****
;Unlock interrupt
;*****

NoAddinterrupt:
push WorkspaceSemaphore              ;pass semaphore
call CVSemaphore                     ;unlock workspace
add  esp, [1 4]                       ;clean up stack

```

NOTE: MLIDs for NetWare 4 servers can call ReadEISAConfig to read the EISA configuration.

See Also

- ◆ ReadEISAConfig
- ◆ DoRealModeInterrupt
- ◆ Appendix E, "Writing Protocol Stacks for NetWare SFT III" (for information on reading the EISA configuration)

GetServerConfigurationType

Determines which operating system or SFT III engine is running. Generally used by protocol stacks.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
LONG GetServerConfigurationType ( void );
```

Parameters

None.

Completion Codes (EAX)

Nonzero	Successful	EAX has a long integer that represents the server configuration type. <ul style="list-style-type: none">◆ EAX = 0 Type Normal Server◆ EAX = 1 Type IOEngine (SFT III)◆ EAX = 2 Type MEngine (SFT III)
0	Fail	The routine failed to determine the server configuration.

Remarks

GetServerConfigurationType informs an NLM which operating system or which SFT III engine it is running on. This enables the NLM to decide whether or not it should load.

GetSuperHighResolutionTimer

Returns a 32-bit timer that increments each microsecond.

Language: C Language

NetWare Operating System: Not version specific

Entry State

Interrupts

Can be in any state.

Return State

EAX

Contains the 32-bit timer.

Interrupts

Unchanged.

Preserved

All registers except EAX, EDX

Completion Codes (EAX)

None.

Remarks

On 486 processors running pre-NetWare 5, the high order 16 bits of this timer will not increment if the routine is called with interrupts disabled. This will cause time to appear to back up when the low order 16 bits overflow.

Therefore, caution must be used when using this routine in a pre-NetWare 5 environment.

ImportPublicSymbol

Attempts to dynamically import a public symbol at run time. This allows an NLM to load whether or not a public symbol is available, and tailor its functions to the environment in which it finds itself.

Language: C Language

NetWare Operating System: NetWare 4 and higher

Syntax

```
void ImportPublicSymbol (  
    LONG moduleHandle,  
    BYTE *symbolName);
```

Parameters

moduleHandle

(IN) The module handle passed to the NLM's initialization routine.

symbolName

(IN) The length-preceded, NULL-terminated public symbol name to be imported.

Completion Codes (EAX)

Nonzero	Successful	The value of the imported public symbol.
0	Fail	The symbol is not available.

Remarks

NOTE: Call this routine only at process time.

This routine is the most important routine to use in allowing NLMs to be written to run on any version of NetWare since its introduction.

By attempting to import public symbols starting with the most recent version of NetWare and working backwards, an NLM can determine which version of NetWare it is running on and function accordingly.

MapAbsoluteAddressToDataOffset

Converts absolute hardware memory address to logical addresses in the NetWare address space. Generally used by MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
LONG MapAbsoluteAddressToDataOffset (  
    LONG AbsoluteAddress);
```

Parameters

AbsoluteAddress

(IN) The 32-bit absolute hardware memory data address.

Completion Codes (EAX)

Nonzero	Successful	EAX contains a logical 32-bit address relative to NetWare's assignment of address 0h.
---------	------------	---

Remarks

Call this routine at process time or interrupt time. The interrupt state is preserved.

MapAbsoluteAddressToDataOffset converts absolute hardware memory addresses to logical NetWare addresses used by MLIDs and operating systems.

MLIDs use *MapAbsoluteAddressToDataOffset* to convert an absolute address of shared LAN adapter RAM to a logical address. This logical address is will appear in a NetWare address space. The MLID needs to do this only once for shared RAM if it saves the result in a variable for subsequent use.

IMPORTANT: NLMs written to the NetWare 3 operating system must use this call. NLMs that are written to the NetWare 4 operating cannot use *MapAbsoluteAddressToDataOffset* to convert absolute memory addresses in

DriverConfig.MLIDMemoryCode0 or DriverConfig.MLJDMemoryCode1. These NLMs should use MLIDLinearMemory0 or MLIDLinearMemory1 after calling RegisterHardwareOptions to convert the corresponding logical address. NLMs written to the NetWare 4 operating system should use ReadPhysicalMemory and WritePhysicalMemory to access physical memory other than the shared RAM addresses.

Example

```
push 0B0000h
call MapAbsoluteAddressToDataOffset    ;EAX = address to use
add  ESP, 1*4                          ;Restore stack
mov  MonoScreen, EAX
```

See Also

- ◆ MapDataOffsetToAbsoluteAddress
- ◆ ReadPhysicalMemory
- ◆ RegisterHardwareOptions
- ◆ WritePhysicalMemory

MapDataOffsetToAbsoluteAddress

Converts logical addresses in the NetWare address space to an absolute hardware memory address. Generally used by MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
LONG MapDataOffsetToAbsoluteAddress (  
    LONG DataOffset);
```

Parameters

DataOffset

(IN) The 32-bit NetWare logical memory data address.

Completion Codes (EAX)

Nonzero	Successful	EAX contains a logical 32-bit real hardware memory address.
---------	------------	---

Remarks

Call this routine either at process or interrupt time. Interrupts can be in any state and is preserved.

MapDataOffsetToAbsoluteAddress converts a logical NetWare address to the real hardware memory address required to initialize DMA channels and bus-mastering devices, and to validate specified hardware options.

Example

```
push OFFSET MyReceiveBuffer  
call MapDataOffsetToAbsoluteAddress  
add ESP, 1*4  
mov RealModeBufferAddress, EAX
```

See Also

- ◆ `MapAbsoluteAddressToDataOffset`

NetWareAlert

Handles system alerts. Generally used by protocol stacks and MLIDs.

Language: C Language

NetWare Operating System: NetWare 4 and higher

Syntax

```
void NetWareAlert (  
    struct LoadDefinitionStructure *nlmHandle,  
    NetWareAlertStructure *alertStructure,  
    LONG ParameterCount,  
    ...  
);
```

Parameters

nlmHandle

(IN) Pointer to the caller's NLM handle.

alertStructure

(IN) Pointer to the following structure of control information. The structure fields are defined in the table that follows the structure definition.

```
typedef struct {  
    void *ptrNetworkManagementAttribute;  
    LONG AlertFlags;  
    union TargetUnion {  
        struct TStation {  
            LONG TargetConnectionNumber;  
        }  
        struct TStationList {  
            LONG *ptrTargetStationListStruct;  
        }  
    }  
    LONG TargetNotificationBits;  
    LONG AlertID;  
    LONG AlertLocus;  
    LONG AlertClass;  
    LONG AlertSeverity;  
    void *AlertDataPtr;
```

```

void *( AlertDataFree )( void *AlertDataPtr );
BYTE *ControlString;
LONG ControlStringMessageNumber;
} NetwareAlertStructure;

```

***** *alertStructure Parameters Defined* *****

pNetWorkManagementAttribute

Should be set to NULL. This parameter deals with the internal network management system

AlertFlags

These bits are defined as follows (all bits not defined are reserved and set to 0):

Bit 0	Queue the alert
Bit 1	AlertID field is valid
Bit 2	AlertLocus is field is valid
Bit 3	Generate an EventReport for this alert, no other alert processing
Bit 4	Don't generate an EventReport for this alert, all other processing is done
Bit 16	ControlStringMessageNumber is valid. ControlString pointer is ignored
Bit 22	Don'tl ring the bell on this alert
Bit 23	AlertID field is a valid NetWare 4 unique ID number
Bit 28	Don't display the locus value to the console
Bit 29	Don't display the alert ID value to the console
Bit 30	Allow Internal Network Management to override Notification Bits
Bit 31	TargetUnion is using the TStationList structure (default is TStation)

TargetUnion

Uses the following structures, depending upon the state of bit 31 in AlertFlags.

The TStation structure contains the connection number to broadcast the alert message to.

```

struct TStation {
    LONG TargetConnectionNumber;
}

```

The TStationList structure contains a pointer to a list of connections to broadcast the alert message to. The first LONG in the list is the number of connection numbers in the list. You can have 1 to 32 LONGs in the list.

```

struct TStationList {
    ptrTargetStationListStruct;
}

```

TargetNotificationBits

The following bit definitions describe the different alert notifications:

Bit 0	Broadcast the alert to the TargetUnion
Bit 1	Broadcast the alert to everyone
Bit 2	Place the alert entry in the SYS\$LOG.ERR file
Bit 3	Display the alert to the console

AlertID

Contains the alert ID value. The AlertID forms a two-part ID. The upper 16 bits of the ID contain the mask for an NLM or product. Novell assigns this value. The lower 16 bits allow each mask to have 1K to 64K unique IDs. Companies outside of Novell will have masks that range between 0x8000000 and 0XFFFF0000. The following masks have been assigned:

```

NOVELL_ALERT_BINDERY      0x01020000
NOVELL_ALERT_OS          0x01030000
NOVELL_ALERT_LLC         0x01040000
NOVELL_ALERT_SDL        0x01050000
NOVELL_ALERT_REMOTE      0x01060000
NOVELL_ALERT_MLID        0x01070000
NOVELL_ALERT_QLLC        0x01080000
NOVELL_ALERT_UPS         0x01090000
NOVELL_ALERT_DS          0x010a0000
NOVELL_ALERT_DOMAIN      0x010b0000
NOVELL_ALERT_RSPX        0x010c0000
NOVELL_ALERT_R232        0x010d0000

```

AlertLocus

Contains the Locus value, currently defined as follows:

LOCUS_UNKNOWN	00h
LOCUS_MEMORY	01h
LOCUS_FILESYSTEM	02h
LOCUS_DISKS	03h
LOCUS_LANBOARDS	04h
LOCUS_COMSTACKS	05h
LOCUS_TTS	07h
LOCUS_BINDERY	08h
LOCUS_LSTATION	09h
LOCUS_LROUTER	10h
LOCUS_LOCKS	11h
LOCUS_KERNEL	12h
LOCUS_UPS	13h
LOCUS_SERVICEPROTOCOL	14h
LOCUS_LOCUS_SFT_III	15h
LOCUS_RESOURCE_TRACKING	16h
LOCUS_NIM	17h
LOCUS_OS_INFORMATION	18h
LOCUS_CACHE	19h
LOCUS_DOMAIN	20h

AlertClass

Contains the class value, currently defined as follows:

CLASS_UNKNOWN	00h
CLASS_OUT_OF_RESOURCE	01h
CLASS_TEMP_SITUATION	02h
CLASS_AUTHORIZATION_FAILURE	03h
CLASS_INTERNAL_ERROR	04h
CLASS_HARDWARE_FAILURE	05h
CLASS_SYSTEM_FAILURE	06h
CLASS_REQUEST_ERROR	07h
CLASS_NOT_FOUND	08h
CLASS_BAD_FORMAT	09h
CLASS_LOCKED	10h
CLASS_MEDIA_FAILURE	11h
CLASS_ITEM_EXISTS	12h
CLASS_STATION_FAILURE	13h
CLASS_LIMIT_EXCEEDED	14h
CLASS_CONFIGURATION_ERROR	15h
CLASS_LIMIT_ALMOST_EXCEEDED	16h
CLASS_SECURITY_AUDIT_INFO	17h
CLASS_DISK_INFORMATION	18h

CLASS_GENERAL_INFORMATION	19h
CLASS_FILE_COMPRESSION	20h
CLASS_PROTECTION_VIOLATION	21h

AlertSeverity

Has the value of the severity of the error. The values are defined as follows:

SEVERITY_INFORMATIONAL	00h	Thresholds have been reached, etc.
SEVERITY_WARNING	01h	Configuration errors, etc. (no damage).
SEVERITY_RECOVERABLE	02h	Hot Fix disk, etc. Work around made.
SEVERITY_CRITICAL	03h	Disk mirror failure, etc. Fix-up attempted.
SEVERITY_FATAL	04h	Resource fatally affected. Shut down.
SEVERITY_OPERATION_ABORTED	05h	Operation cannot complete. Ramifications unknown.
SEVERITY_NONOS_UNRECOVERABLE	06h	Operation cannot complete. Ramifications will not affect operating system.

AlertDataPtr

Has a pointer to a block of data to be passed to the AlertDataFree routine.

AlertDataFree

This routine is called after the execution of NetWareAlert. This routine is passed AlertDataPtr.

ControlString

Has a pointer to the control string. This pointer is only valid when the MessageNumber bit 0 is set to 0. This pointer is used with the parameters to make the alert message.

ControlStringMessageNumber

Has the message number of the control string in the NLM's enabling table. This value is only valid if the MessageNumber bit 0 is set to 1.

***** End of alertStructure Parameters *****

ParameterCount

(IN) Has the number of parameters (following- ParameterCount) passed on the stack to this routine.

...

(IN) Can take a variable number of LONG arguments. The number of arguments passed on the stack is the number held in ParameterCount.

Completion Codes (EAX)

None.

Remarks

NetWareAlert allows the calling function to determine where the alert goes. The alert could be sent to a file, the console, a connection number, etc. This routine also allows the calling NLM to its own enabled messages. NetWareAlert also allows NLMS to queue the alert if the routine is called at interrupt time.

NVMAAlloc

NVMAAlloc is the primary allocation API within the VM system.

NetWare Operating System: NetWare 5 only

Syntax

```
LONG NVMAAlloc (  
    void *virtualAddress,  
    LONG pageCount,  
    LONG memoryState,  
    LONG attributeFlags,  
    LONG rights );
```

Parameters

virtualAddress

(IN) The address where the allocated virtual memory begins.

pageCount

(IN) This value indicates the number of pages of virtual memory to be allocated. This value can be from 1 up the maximum number of pages defined as being part of the address space.

memoryState

(IN) This value describes initial state of memory at allocation time. Valid states include:

Reserved - Reserve the logical space, do not back it with RAM or swap store. A page fault occurs if the space is "touched".

Auto-Commit - Reserve the logical space, do not back it with RAM or swap store initially. A page fault will cause an attempt to back the page with RAM or swap store. This state exhibits non-deterministic behavior because backing store is not initially reserved.

Committed - Reserve logical space and back it with RAM or backing store at alloc time.

attributeFlags

(IN)

Locked - Do not permit committed memory to be paged out. (May still be "moved")

User/Supervisor - Used to determine in which portion of the address space memory will be allocated. Will be set to "user" by the marshaling code. User requests will be zero filled; supervisor requests will not. This may end up being a "hidden" flag.

NoSleep - Do not allow the call to "yield"

rights

(IN)

Read - Allocated memory is read only

Read/Write - Allocated memory is read/write

VirtualMemory

(OUT) The value returned here is the address of the newly allocated virtual memory with the given specifications. If the function return code is non-zero, then the value returned is undetermined.

Completion Codes (EAX)

The return value of the function is the condition code representing the outcome of the VM alloc request. Possible return values are:

Normal = 0

Errors = ?????

Remarks

This function is used to obtain logical address space and virtual memory within both the kernel and user portions of the currently visible address space. If this call is made from user space, the memory allocated will be in the user portion of the address map. If the call is made from kernel space, the user/kernel attribute flag determines if the allocated memory will be in user space or kernel space. All memory allocated in user space will be zero initialized. Kernel memory will not be initialized.

NVMAAllocIO

NVMAAllocIO is used to allocate virtual memory which has one or more portions of physically contiguous memory backing it. The memory returned is committed, page locked and move locked, ie. the memory is backed with RAM, will not be paged to backing store and the logical to physical address translation will remain constant.

NetWare Operating System: NetWare 5 only

Syntax

```
LONG NVMAAllocIO (  
    void *virtualMemory,  
    LONG pageCount,  
    LONG attributeFlags,  
    LONG rights,  
    LONG *nonContiguousCount,  
    specStruct *specStructure[],  
    LONG *sleptFlag);
```

Parameters

virtualAddress

(IN) The address where the allocated virtual memory begins.

pageCount

(IN) This value indicates the number of pages of virtual memory to be allocated. This value can be from 1 up the maximum number of pages defined as being part of the address space.

attributeFlags

(IN)

User/Supervisor - Used to determine in which portion of the address space memory will be allocated. Will be set to "User" by the marshaling code. User requests will be zero filled; supervisor requests will not. This may end up being a "hidden" flag.

NoSleep - Do not allow the call to "yield"

rights

(IN)

Read - Allocated memory is read only

Read/Write - Allocated memory is read/write

nonContiguousCount

(IN) This parameter points to a value which indicates the maximum number of permissible physically contiguous memory regions. The value is modified by the API and is returned as the number of regions that were allocated to fulfill the request. The input value also specifies how many elements the array of `specStructure` contains. The output value specifies how many elements of the array of `specStructure` were filled in by the API.

nonContiguousCount

(OUT) This parameter points to a value which indicates the maximum number of permissible physically contiguous memory regions. The value is modified by the API and is returned as the number of regions that were allocated to fulfill the request. The input value also specifies how many elements the array of `specStructure` contains. The output value specifies how many elements of the array of `specStructure` were filled in by the API.

virtualMemory

(OUT) The value returned here is the address of the newly allocated virtual memory with the given specifications. If the function return code is non-zero, then the value returned is undetermined.

specStructure

(OUT) This parameter points to an array of specification structures which are filled in by the API. The array elements order corresponds to the order in which the physical memory segments appear in the logical address region allocated by the API. Each array element structure contains an address and a size field which specify the physical address and the size in pages of its segment.

sleptFlag

(OUT) This parameter specifies whether or not a yield was necessary to fulfill the request. A non-zero value indicates that the API yielded at some point during its execution.

Completion Codes (EAX)

The return value of the function is the condition code representing the outcome of the request. Possible return values are:

Normal = 0

Errors = ?????

Remarks

This API is used to obtain logical address space and virtual memory within both the kernel and user portions of the currently visible address space. If this call is made from user space, the memory allocated will be in the user portion of the address map. If the call is made from kernel space, the user/kernel attribute flag determines if the allocated memory will be in user space or kernel space. All memory allocated in user space will be zero initialized. Kernel memory will not be initialized.

The number of distinct physical address regions that can be tolerated is specified in the *pagecount* parameter. The starting addresses and sizes (in number of pages) of each physically contiguous memory region is returned in the array of structures pointed to by *specStructure*. This API may sleep, if necessary, to find the required number of physically contiguous memory segments. The *sleptFlag* parameter notifies the caller that a yield was necessary to fulfill the request.

NVMFree

NVMFree returns the virtual memory, allocated by *NVMAlloc* or *NVMAllocIO* to the VM system.

NetWare Operating System: NetWare 5 only

Syntax

```
LONG NVMFree (  
    void *virtualMemory);
```

Parameters

virtualMemory

(IN) The value input here is the address of the segment of virtual memory to be freed.

Completion Codes (EAX)

The return value of the function is the condition code representing the outcome of the request. Possible return values are:

Normal = 0

Errors = ?????

Remarks

This function is object oriented in that it takes as an argument only a memory address which was previously obtained using *NVMAlloc* and *NVMAllocIO*. The entire region originally obtained is freed.

OutputToScreen

Displays a message on the server console screen. Generally used by protocol stacks and MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
void OutputToScreen (
    struct ScreenStruct *screenID,
    char *controlString,
    args... );
```

Parameters

screenID

(IN) Has the screen handle of the console screen, which the MLID receives during initialization.

controlString

(IN) Has a pointer to a null-terminated ASCII string.

args...

(IN) Is a procedure that can take a variable number of standard printf control string arguments.

Completion Codes (EAX)

None.

Remarks

NOTE: Call this routine only at process time. It will not suspend the calling process.

MLIDs use *OutputToScreen* to display a message on the server console screen.

MLIDs should not display nonvital messages. MLIDs should also limit the number of lines output to the screen for essential messages, because displaying unneeded output causes important information to scroll off the screen.

controlstring can be embedded with returns, line feeds, bells, tabs, and backspaces. However, if strings contain embedded sub-strings, numbers, and control information, they must be no longer than 200 characters. Strings longer than this cause the server to abend. If the MLID must use longer strings, split the string into several strings and call `OutputToScreen` multiple times.

NOTE: `ScreenID` is not valid after returning from the initialization routine, so the MLID can call `OutputToScreen` only during initialization.

Example

```
push OFFSET MyMessage
push [ESP + InitializationErrorScreen + 4] ;Screen handle
call OutputToScreen ;The string is on the stack
add ESP, 2 * 4 ;Restore stack
```

ParseDriverParameters

Parses the command line parameters. Generally used by MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
LONG ParseDriverParameters (
    struct IOConfigurationStructure *IOConfig,
    struct DriverConfigurationStructure *configuration,
    struct AdapterOptionDefinitionStructure *adapterOptions,
    struct LANConfigurationLimitStructure
        *ParseDriverParameters,
    BYTE (*FrameTypeDescription) [],
    LONG needBitMap,
    BYTE *commandLine,
    struct ScreenStruct *screenID );
```

Parameters

IOConfig

(IN) Pointer to the LAN adapter's corresponding IOConfigurationStructure (starting at the MLIDLink field of the configuration table).

configuration

(IN) Pointer to the logical board's configuration table.

adapterOptions

(IN) Pointer to the AdapterOptionDefinitionStructure.

ParseDriverParameters

(IN) Pointer to the LANConfigurationLimitStructure.

FrameTypeDescription

(IN) Pointer to the beginning array of pointers. These are pointers to frame descriptors defining the packet's frame type.

needBitMap

(IN) Bit map telling ParseDriverParameters which hardware options the LAN adapter requires

commandLine

(IN) Pointer to the command line, passed to the MLID at load time

screenID

(IN) Has a pointer to the ScreenHandle that was passed to the MLID at initialization

Completion Codes (EAX)

Zero	Successful
Nonzero	Fail

Remarks

Call this routine only at process time. This routine can suspend the process and change the processor state. *screenID* is valid only during initialization.

ParseDriverParameters uses the command line parameters, operator input, and tables provided by the MLID to fill in the configuration table of the logical board and the IOConfigurationStructure associated with that configuration table. The IOConfigurationStructure starts at the MLIDLink field of the configuration table.

The MLID uses ParseDriverParameters with RegisterHardwareOptions.

Tables provided by the MLID.

Frame Descript Table

```
FrameDescriptTable
dd Ethenet8023Descript
dd EthenetIIDescript
dd Ethenet8022Descript
dd EthenetSNAPDescript
Message Ethernet8023Descript, 'ETHERNET_802.3'
Message EthernetIIDescript, 'ETHERNET_II'
```

```
Message Ethernet8022Descript, 'ETHERNET_802.2'  
Message EthemetSNAPDescript, 'ETHERNET_SNAP'
```

Message Macro Definition

```
Message macroMessageName, MessageString  
local StringEnd, StringBegin  
MessageName db StringEnd - StringBegin  
StringBegin db MessageString  
StringEnd db 0  
endm
```

The message macro used above causes the strings in the NameDescriptTable to be length-preceded and null-terminated. The number of provided messages depends upon the value in the ParseDriverParameters structure NumberFrames field. In the above example, four messages are provided. AdapterOptionDefinitionStructure

The AdapterOptionDefinitionStructure is a hard-coded part of the MLID's data structure. Using the NeedsBitMap as a guide, ParseDriverParameters collects the necessary information from the command line and from the AdapterOptionDefinitionStructure, fills out the appropriate fields in the configuration table, and returns successfully.

The MLID doesn't necessarily set the bit in the bitmap field if it uses a parameter. However, if the MLID has multiple possibilities (for instance, it must parse input from the console or parse the command line) and it must use ParseDriverParameters to determine which option to use, it must set the appropriate bit in the NeedsBitMap.

Each field in the AdapterOptionDefinitionStructure is a pointer. If the MLID does not support that option, it places a 0 in that field. If the MLID does support that option, it places a pointer to an option list in that field. The AdapterOptionDefinitionStructure is defined below:

```
AdapterOptionDefinitionStructure struct  
IOSLOT dd ?  
IOPort0 dd ?  
IOLength0 dd ?  
IOPort1 dd ?  
IOLength1 dd ?  
MemoryDecode0 dd ?  
MemoryLength0 dd ?  
MemoryDecode1 dd ?  
MemoryLength1 dd ?
```

```

Interrupt0      dd  ?
Interrupt1     dd  ?
DMA0           dd  ?
DMAL           dd  ?
Channel        dd  ? ;the field exists in NetWare 4 only
AdapterOptionDefinitionStructure ends

```

Option List

The option list appears as follows:

```

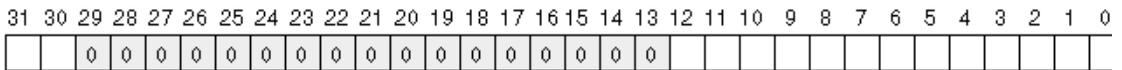
IRQOptions      dd  4                ;option count
                dd  3,2,5,7
MemoryOptions   dd  2
                dd  OD000h,OD8000h
IOPortOptions   dd  4                ;Number of available options
                dd  300h,310h        ;300,310
                dd  320h,330h        ;320,330
InterruptOptions dd  3,4,6,9         ;4,5,9
AdapterOptions  AdapterOptionDefinftoure
                <,IOPort Options,,,Memory Options,,,IRQ Options>

```

Option for Scanning Slots

If the MLID uses slots, and scans the slots at run time to determine which slots have a LAN adapter in them, it should build the appropriate option dynamically.

The MLID must specify the CanSetNode.Address or MustSetNodeAddress flags in the NeedsBitMap parameter if it desires to use this option. (These flags were previously in the NeedFlags parameter of NetWare 3.0.) The NeedsBitMap parameter is defined below:



Default Values

Table 57 **Bit Map of NeedsBitMap**

Bit	Needs Option	
0	NeedsIOSlotBit	(00000001h)
1	NeedsIOPort0Bit	(00000002h)

Bit	Needs Option	
2	NeedsIOLengthOBit	(00000004h)
3	NeedsIOPortIBit	(00000009h)
4	NeedsIOLengthIBit	(00000010h)
5	NeedsMemoryDecode0Bit	(00000020h)
6	NeedsMemoryLength0Bit	(00000040h)
7	NeedsMemoryDecode1Bit	(00000080h)
8	NeedsMemoryLength1Bit	(00000100h)
9	NeedsInterrupt0Bit	(00000200h)
10	NeedsInterrupt1Bit	(00000400h)
11	NeedsDMA0Bit	(00000800h)
12	NeedsDMA1Bit	(00001000h)
13	NeedsChannelBit (NetWare 4 field only)	(00002000h)
14	Reserved	
15	Reserved	
16	Reserved	
17	Reserved	
18	Reserved	
19	Reserved	
20	Reserved	
21	Reserved	
22	Reserved	
23	Reserved	
24	Reserved	
25	Reserved	

Bit	Needs Option	
26	Reserved	
27	Reserved	
28	Reserved	
29	Reserved	
30	CAN_SET_NODE_ADDRESS	(40000000h)
31	MUST_SET_NODE_ADDRESS	(80000000h)

LANConfigurationLimitsStructure

The ConfigLimits parameter is a pointer to a LANConfigurationLimitsStructure, defined below:

```

ConfigLimits      label  byte
MinNodeAddress   dd      MinAddress
MaxNodeAddress   dd      MaxAddress
MinRetries       dd      0
MaxRetries       dd      255
NumberFrames     dd      4
NumberChannels   dd      0 ;NetWare 4.x only

```

ParseDriverParameters fills in the configuration table field MLIDChannelNumber (offset 40h) with the channel number. For most LAN adapters, this number will be 0. For multichannel LAN adapters, this number can be 0, 1, 2, etc.

Example

```

push [ESP + lnifializabonErrorScreen] ;Screen handle
push [ESP + Configurationinfo + 4]   ;Pointer to command line
push NeedsIOPortOBit OR NeedsInterrupt0Bit OR CanSetNodeAddress
push OFFSET FrameDescriptTable      ;Mecha ID string array
push OFFSET ParseDriverParameters   ;Node and Retry limits
push OFFSET Adapter Options         ;Options to query from user
push OFFSET DriverConfiguration     ;Driver configuration table
push OFFSET [ebp] CDriverlink       ;IO configuration table
call ParseDriverParameters          ;Call parser
add ESP, 8 * 4
or EAX, EAX                          ;Successful?

```

```
jnz ErrorParsingDriverOptions ;if not, Exit init
```

QueueSystemAlert

Notifies the system of hardware or software problems. Generally used by protocol stacks and MLIDs.

Language: C Language

NetWare Operating System: NetWare 3 only

Syntax

```
LONG QueueSystemAlert (  
    LONG TargetStation,  
    LONG TargetNotificationBits,  
    LONG ErrorLocus,  
    LONG ErrorClass,  
    LONG ErrorCode,  
    LONG ErrorSeverity,  
    void *controlString,  
    ...);
```

Parameters

TargetStation

(IN) Has the connection number of the affected station, or 0 if no single station is affected (this parameter usually holds a 0, which means that no single station is affected).

TargetNotificationBits

(IN) Has the destinations of the notification..

Target Notification Bits:

NOTIFY_CONNECTION_BIT	01h
NOTIFY_EVERYONE_BIT	02h
NOTIFY_ERROR_LOG_BIT	04h
NOTIFY_CONSOLE_BIT	08h
DONT_NOTIFY_NMAGENT	80000000h

ErrorLocus

(IN) Has the locus of the error.

Error Locus Bits:

LOCUS_UNKNOWN	00h
LOCUS_LANBOARDS	04h
LOCUS_COMMSTACKS	05h
LOCUS_NLM	11h
LOCUS_ROUTERS	0Ah

ErrorClass

(IN) Has the class of the error.

Error Class Bits:

CLASS_UNKNOWN	00h
CLASS_OUT_OF_RESOURCE	01h
CLASS_TEMP_SITUATION	02h
CLASS_AUTHORIZATION_FAILURE	03h
CLASS_INTERNAL_ERROR	04h
CLASS_HARDWARE_FAILURE	05h
CLASS_SYSTEM_FAILURE	06h
CLASS_REQUEST_ERROR	07h
CLASS_NOT_FOUND	08h
CLASS_BAD_FORMAT	09h
CLASS_LOCKED	10h
CLASS_MEDIA_FAILURE	11h
CLASS_ITEM_EXISTS	12h
CLASS_STATION_FAILURE	13h
CLASS_LIMIT_EXCEEDED	14h
CLASS_CONFIGURATION_ERROR	15h

ErrorCode

(IN) Error codes for the system log.

Error Code Bits:

OK	00h
ERR_HARDFAILURE	0ffh

ErrorSeverity

(IN) Has the severity of the error.

Error Severity Bits:

SEVERITY_INFORMATIONAL	00h
SEVERITY_WARNING	01h
SEVERITY_RECOVERABLE	02h
SEVERITY_CRITICAL	03h

SEVERITY_FATAL 04h
SEVERITY_OPERATION_ABORTED 05h

controlString

(IN) Has a pointer to a message string, which has a maximum of 256 characters and is null-terminated.

...

(IN) Can take a variable number of control string arguments.

Completion Codes (EAX)

00000000h	Successful
00000001h	Alert Not Available

Remarks

Call this routine at either process or interrupt time. This routine runs to completion and preserves the interrupt states.

QueueSystemAlert provides a system notification of stack or MLID hardware or software problems that must be reported at run time. Use OutputToScreen to display errors at initialization.

IMPORTANT: NLMs written to the NetWare 4 operating system should use NetWareAlert instead of QueueSystemAlert.

Example

```
TransmitTimeoutMessage db Transmit failure on board #%d', 0
movzx EAX, [EBX].CStackBoardNumber ;Pass our board number
push EAX
push OFFSET TransmitTimeoutMessage ;Pass error string
push 2 ;SeverityRecoverable
xor EAX, EAX
push EAX ;Error code
push 6 ;ClassHardwareFailure
push 0
push 01100b ;Console & ErrorLog
push EAX ;Station #, not used
```

```
call QueueSystemAlert
add ESP, 8 * 4 ;Clean up stack
```

See Also

- ◆ NetWareAlert
- ◆ OutputToScreen

ReadEISAConfig

Reads the EISA configuration block for the specified slot. Generally used by MLIDs.

Language: Assembly Language

NetWare Operating System: NetWare 4

Entry State

CH

Function/block.

CL

Board slot.

Interrupts

Are in any state.

Call

At process time only.

Return State

EAX

Has a completion code.

ESI

Address of the configuration buffer.

Interrupts

Are preserved.

Preserved

ECX

Completion Codes (EAX)

0x00000000	Successful
0x00000001	Int 15h vector removed
0x00000080	Invalid slot number
0x00000081	Invalid function number
0x00000082	Nonvolatile memory corrupt
0x00000083	Empty slot
0x00000086	Invalid BIOS routine called
0x00000087	Invalid system configuration

Remarks

IMPORTANT: *ReadEISAConfig* has been superseded by the NBI function *ReadCardConfigInfo*.

ReadEISAConfig reads the EISA configuration block for the specified slot into a 320-byte buffer. The MLID typically calls this routine during initialization.

Usually the MLID calls this routine with Block - 0. If the MLID does not find the information in this block, it continues calling this routine and incrementing the block number until it either receives the right block or runs out of blocks.

The MLID should copy the returned configuration block into local memory. Once the MLID returns to the operating system or calls a blocking procedure, the block information is no longer valid.

ReadPhysicalMemory

Reads memory from a physical address into a buffer pointed to by a logical address. Generally used by MLIDs.

Language: C Language

NetWare Operating System: NetWare 4 and higher

Syntax

```
LONG ReadPhysicalMemory (  
    BYTE *Source,  
    BYTE *Dest,  
    LONG NumUnits,  
    LONG UnitLength);
```

Parameters

Source

(IN) The physical address of the memory to be read.

Dest

(IN) The logical memory address of the buffer to read into.

NumUnits

(IN) The number of memory units to read.

UnitLength

(IN) Has the length of the memory units:

1=Byte

2=Word

4=DWord

Completion Codes (EAX)

0x00000000	Failure	The routine failed because of a bad parameter.
------------	---------	--

Remarks

ReadPhysicalMemory reads memory, from a physical address into a buffer pointed to by a logical address. This routine is used on the NetWare 4 operating system when the NLM only knows a physical address.

MLIDs can use this call to read memory from shared RAM before they call RegisterHardwareOptions.

See Also

- ◆ RegisterHardwareOptions
- ◆ WritePhysicalMemory

ReadRoutine

Allows the MLID to read custom data or firmware. Generally used by MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
LONG (*ReadRoutine) (  
    LONG CustomFileHandle,  
    LONG *CustomDataOffset,  
    LONG *Destination,  
    LONG CustomDataSize);
```

Parameters

CustomFileHandle

(IN) Has the file handle, supplied as LoadableModuleFileHandle to the MLID's initialization routine.

CustomDataOffset

(IN) Has the starting offset in the file, supplied as CustomDataOffset to the MLID's initialization routine.

Destination

(IN) Has the location of where to read the file to.

CustomDataSize

(IN) Has the amount of the data to read, supplied as CustomDataSize to the MLID's initialization routine.

Completion Codes (EAX)

0	Successful
Nonzero	Fail

Remarks

ReadRoutine allows MLIDs to read custom data and firmware into system memory during initialization. Before the MLID calls this routine, it must allocate memory to read the custom data into.

This routine can only be called during initialization. This routine could go to sleep and could return with interrupts enabled.

ReadRoutine's entry point is not exported by the operating system. The only time this entry point is valid is during the initialization routine. In fact, the entry point is passed as a local parameter () and must be called indirectly.

The NLM linker actually appends the custom data file to the MLID in the LAN file. NetWare only loads the MLID's code data at load time, leaving the file open for the MLID to handle custom data however it must.

To define the custom file, use the CUSTOM keyword in the MLID's definition file, followed by the file's name. The NetWare operating system passes the custom file handle, starting address, and size to the MLID's initialization routine. NetWare also passes the address of the *ReadRoutine*. The MLID's initialization routine can then read the file into memory by calling the *ReadRoutine*.

The MLID must supply the destination in memory according to the needs of the LAN adapter.

Example

```
mov  EAX, dword ptr [ESP + CustomDataSize) ;get size of firmware
push MemoryRTag                          ;push tag
push  EAX                                 ;push size
push  AllocSemiPermMemory                 ;allocate memory to
lea   ESP, [ESP + 4 * 2]                  ;clean up stack
or    EAX, EAX                            ;did we get it'?
jz    ErrorGetfingExtraMemory             ;error exit if not
mov   FirmWareBuffer EAX                  ;save firmware
                                           ;buffer
mov   ESI, EAX                            ;allocated memory
mov   EAX, [ESP + LoadableModuleFileHandle] ;file handle firmware
mov   EBX, [ESP + ]                       ;read routine
                                           ;address
mov   EDX, [ESP + CustomDataOffset]       ;start address in file
mov   ECX, [ESP + CustomDataSize]         ;get size of firmware
push  ECX                                 ;amount to read
push  ESI                                 ;where to read to
```

```
push EDX                ;offset in file
push EAX                ;file handle
call EBX               ;call read routine
cli                    ;stop interrupts
add ESP, 4 * 4         ;adjust the stack
or EAX, EAX            ;check for read
jnz ReadError          ;errors
```

NOTE: The CUSTOM keyword must be used in the definition file to specify the file name for the firmware.

RegisterForEventNotification

Registers a routine to be called when a certain event occurs. Generally used by protocol stacks and MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
LONG RegisterForEventNotification (
    struct ResourceTagStructure *resourceTag,
    LONG eventType,
    LONG priority,
    LONG (*warnProcedure) (
        void (*OutputRoutine) (
            void *controlstring,
            ... );
        LONG parameter,
        LONG userParameter ),
    void (*reportProcedure) (
        LONG parameter,
        LONG userParameter )
    LONG userParameter );
```

Parameters

resourceTag

(IN) Pointer to a ResourceTagStructure that contains an EventSignature. This resource tag is to be used for event notification.

eventType

(IN) The type of event the protocol stack wants to be notified of (see the Remarks section).

priority

(IN) The order in which to call registered callback routines.

EventPriorityOS	00h
EventPriorityApplication	20h
EventPriorityDevice	40h

warnProcedure

(IN) Pointer to a callback routine that will be called before the event has occurred. If the *warnProcedure* does not want the event to occur, it should return with a nonzero value. Set to zero if not used.

OutputRoutine

(IN) Sends an output message to the user of a particular event.

controlString

(IN) Standard printf control string that is to be used in the output routine.

...

(IN) Additional parameters can be passed to the output routine in order to match the control string requirements.

parameter

(IN) 32-bit value that is defined according to the event type.

reportProcedure

(IN) Pointer to a callback routine that is called when an event occurs.

userParameter

(IN) 32-bit value passed on input and returned in *warnProcedure* and in *reportProcedure*.

warnProcedure

(OUT) Pointer to a callback routine that will be called before the event has occurred. If the *warnProcedure* does not want the event to occur, it should return with a nonzero value. Set to zero if not used.

reportProcedure

(OUT) Pointer to a callback routine that is called when an event occurs.

Completion Codes (EAX)

Nonzero	Successful	EAX has an EventID that should be used when UnRegisterEventNotificatfon is called.
0	Fail	

Remarks

RegisterForEventNotification adds routines to the list when an event is reported. The protocol stacks and MLIDs call these routines according to priority. The warning routine is called when the operating system calls an EventCheck. The report routine is called when the operating system calls an EventReport. The protocol stack or MLID passes the parameter that it receives when the event is reported to the called routine. This called routine returns an EventID that the protocol stack or MLID should use when it calls UnRegisterEventNotification.

When the type of event defined by eventType occurs, the operating system calls the callback routine. The types of events that can be defined in eventType are listed below:

EventDownServer 4h The parameter is undefined. The warn routine and the report routine are called before the server is shut down.

EventChangeToRealMode 5h The parameter is undefined. The report routine is called before the server changes to real mode and must not go to sleep.

EventReturnFromRealmode 6h The parameter is undefined. The report routine is called after the server returns from DOS and must not go to sleep.

EventExitToDOS 7h The parameter is undefined. The report routine is called before the server exits to DOS.

EventProtocolBind 21h This parameter is a pointer to an EventProtocolBind structure. This event is generated every time a board is bound to a protocol stack. This event may sleep.

```
EventProtocolBind struct (  
    LONG BoardNumber;  
    LONG ProtocolNumber;
```

EventProtocolUnbind 22h This parameter is a pointer to an EventProtocolBind structure (see above). This event is generated every time a board is unbound to a protocol stack. This event may sleep.

EventMLIDRegister 27h This parameter is a board number. The report routine is called after the MLID is registered. This event may sleep.

EventMLIDDeregister 28h This parameter is a board number. The report routine is called before the MLID is deregistered. This event may sleep.

EventProtocolRegistered 87h This parameter is generated when a Protocol is registered. The parameter is a pointer to the EventProtocolRegisteredStruct. This event may sleep.

EventProtocolUnregistered 88h This parameter is generated when a Protocol is unregistered. The parameter is a pointer to the EventProtocolRegisteredStruct. This event may sleep.

The order in which the call back routines are called is determined by the priority parameter. The priorities are notified first. The available priorities are listed below:

EventPriorityvOS	0h
EventPriorityApplication	20h
EventPriorityDevice	40h

The protocol stack or MLID passes the callback routines a parameter (which can include nulls) and a report routine to be used to warn the user against the occurrence of a particular event. The parameter reportprocedure is passed additional event-specific information when it is needed.

This routine is also used at initialization to register an event callback routine. For example, the MLID calls this routine so that it can be notified if the server is going to edit to DOS. This gives the MLID a chance to cancel any AES or timer events, and allows bus-master devices to return any preallocate resource and shut down the LAN adapter. It also prevents LAN adapters that use DMA or are bus-master devices from writing to memory after DOS gains control.

Example

```
push OFFSET ExitOSAndR@wd           ;Address of exit routine
push 0
push EventPriorityOS                 ;Set priority level
push EventExitToDOS                 ;Set what event
push EventRTag                       ;Resource event tag
call RegisterForEventNotification    ;Have OS patch in routine
add ESP, 4 * 5                       ;Clear up stack
or EAX, EAX                          ;Did OS patch in call?
jz EventPatchError                  ;Error did not add procedure
```

See Also

- ◆ AllocateResourceTag

- ◆ UnregisterEventNotification

RegisterHardwareOptions

Reserves hardware options for a LAN adapter. Generally used by MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
LONG RegisterHardwareOptions (  
    struct IOConfigurationStructure *IOConfig,  
    struct DriverConfigurationStructure *configuration);
```

Parameters

IOConfig

(IN) Has a pointer to the MLIDLink field in the logical board's configuration table.

configuration

(IN) Has a pointer to the logical board's configuration table.

Completion Codes (EAX)

0	Successful	The hardware is successfully registered.
1	Successful	Duplicate hardware with a new frame type is registered.
2	Successful (NetWare 4 only)	A new channel for existing hardware is registered.
2	Failure (NetWare 3 only)	The routine failed to register the hardware because of either a conflict or a bad parameter.
3	Failure (NetWare 4 only)	The routine failed to register the hardware because of either a conflict or a bad parameter.

Remarks

RegisterHardwareOptions reserves hardware options for a specific LAN adapter.

Call this routine only at process time. This routine runs to completion. Interrupts can be in any state and do not change.

The MLID passes *RegisterHardwareOptions* a pointer to an *IOConfigurationStructure* that has the specified hardware options to reserve. If any of the hardware options are already in use, the routine returns an error code. The *IOConfigurationStructure* starts at the *MLIDLink* field of the configuration table.

In NetWare 4, this function fills in *MLIDLinearMemory0* and *MLIDLinearMemory1*, which the MLID must subsequently use to access its card's memory.

In NetWare 3, the MLID must use *MapAbsoluteAddressToDateOffset* to convert *MLIDMemoryDecode0* to *MLIDLinearMemory0* and *MLIDMemoryDecode1* to *MLIDLinearMemory1* .

Example

```
push OFFSET DriverConfiguration
push OFFSET MyIOConfig
call RegisterHardwareOptions      ;Register hardware
add ESP, 2 - 4                    ;Now restore stack
emp EAX, 1                        ;Duplicate Hardware
je AddNewFrameType
ja ErrorRegisteringHardware
```

RemovePollingProcedure

Removes the MLID's polling procedure from the server. Generally used by MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
void RemovePollingProcedure (  
    void (*Procedure) ( void ) );
```

Parameters

Procedure

(IN) Has a pointer to a previously added polling procedure.

Completion Codes (EAX)

None.

Remarks

Call this routine only at process time. This routine runs to completion. Interrupts can be in any state, and do not change.

The MLID uses RemovePollingProcedure to remove its poll routine from the server's list of polling procedures.

A polled MLID calls RemovePollingProcedure when it unloads.

Example

```
push OFFSET NewDriverPoll          ;Remove us from poll  
call RemovePollingProcedure        ;List  
add ESP, 4
```

ScheduleInterruptTimeCallback

Adds an event to the timer interrupt handler's event list. Generally used by protocol stacks and MLIDs.

Language: Assembly Language

NetWare Operating System: Not version specific

Entry State

EDX

Has a pointer to a timer node data structure.

Interrupts

Are disabled.

Call

At process or interrupt time.

Return State

Interrupts

Are disabled and are not enabled

Preserved

All registers but EDI and ESI

Completion Codes (EAX)

None.

Remarks

IMPORTANT: This process does not relinquish control of the CPU.

ScheduleInterruptTimeCallback adds an event to the list of events called by the timer interrupt handler. The *TimerNodeDataStructure* is shown below:

```
TimerNodeDataStructure   struc
TLINK                    dd
```

```

TCallbackProcedure      dd      ;Set by caller
TCallbackEBXParameter  dd      ;Set by caller
TCallbackWaitTime      dd      ;Set by caller
TResourceTag           dd      ;Set by caller
TReserved1             dd
TReserved2             dd
TimerNodeDataStructure ends

```

The appropriate fields of this structure should be filled out as follows:

TCallbackProcedure	Pointer to the procedure to be called by the timer interrupt handler.
TCallbackEBXParameter	The value EBX should contain when the callback procedure is invoked.
TCallbackWaitTime	The amount of time, in ticks, before the callback procedure is invoked.
TResourceTag	The resource tag that was allocated in order to use this call.

The operating system does not change the fields described above. Therefore, if you reschedule another callback, you do not need to reset these fields.

Example

```

cli
mov  EDX, OFFSET MyTimerNode      ;TimerNodeDataStructure
mov  [EDX].TCallbackEBXParameter, EBP
mov  EBX, OFFSET MyTimerinterruptCallbackRoutine
mov  [EDX].TCallbackProcedure, EBX
                                ;can also use offset
                                ;MyTimerinterruptCallbackRoutine
                                ;instead of EBX
mov  EBX, 1-imerResourceTag mov  [EDX].TResourceTag, EBX
mov  [EDX].TCallbackWaitTime      ;Wake up in 5 ticks
call ScheduleInterruptTimeCallback

```

NOTE: TResourceTag points to the resource tag acquired by the protocol stack or by the MLID for InterruptTimeCallbacks (with a TimerSignature).

See Also

- ◆ CancelInterruptTimeCallback

ScheduleNoSleepAESProcessEvent

Sets up a background no-sleep thread. Generally used by protocol stacks and MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
void ScheduleNoSleepAESProcessEvent (  
    struct AESProcessStructure *EventNode);
```

Parameters

EventNode

(IN) Pointer to an AESProcessStructure.

Completion Codes (EAX)

None.

Remarks

ScheduleNoSleepAESProcessEvent sets up a background AESNoSleep thread to be executed at specific intervals. (AES stands for Asynchronous Event Scheduler.)

The protocol stack or the MLID must allocate the AESProcessStructure before calling this routine and must provide the execution level and the execution address.

Calling this routine once creates a single entry in the defined thread.

You must call *ScheduleNoSleepAESProcessEvent* every time you want to execute the thread.

NOTE: The call specified in *AESProcessStructure* must not relinquish control of the processor.

Call this routine at process time or interrupt time. When the procedure returns, the interrupt state is preserved.

The AESProcessStructure is defined below:

```
struct AESProcessStructure struc
ALink                dd ? ;used by the operating system.
AWakeUpDelayAmount   dd ? ;filled out by caller, won't be changed
AwakeUpTime          dd ? ;used by operating system
AProcessToCall       dd ? ;filled out by caller, won't be changed
ARTag                dd ? ;filled out by caller, won't be changed
AOldLink             dd ? ;used by operating system
AESProcessStructure  ends
```

IMPORTANT: The AESProcessStructure must be in static memory and available long-term.

The operating system does not change any of the AESProcessStructure fields filled by the caller. Therefore, you do not need to reset these flags before you reschedule the process.

Example

```
push EAX                ;Points to an AES structure
call ScheduleNoSleepAESProcessEvent
add ESP, 4              ;Adjust the stack pointer
```

See Also

- ◆ AllocateResourceTag
- ◆ CancelNoSleepAESProcessEvent

ScheduleSleepAESProcessEvent

Sets up a background sleep thread. Generally used by protocol stacks and MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
void ScheduleSleepAESProcessEvent (
    struct AESProcessStructure *EventNode);
```

Parameters

EventNode

(IN) Pointer to an AESProcessStructure.

Completion Codes (EAX)

None.

Remarks

Call this routine at process time or interrupt time. When the procedure returns, the interrupt state is preserved.

ScheduleSleepAESProcessEvent sets up a background AES (Asynchronous Event Scheduler) thread that can be executed at desired intervals. This thread can be blocked and can make blocking calls while executing. The protocol stack or MLID must have allocated the AESProcessStructure prior to the first call and must have provided the execution level and execution address. A single call to this routine causes a single entry to the defined thread.

NOTE: The call specified in *AESProcessStructure* may relinquish control of the processor. AESProcessStructure must be in static memory and available long-term. The AESProcessStructure is defined below:

```
struct AESProcessStructure struc
ALink                dd ? ;used by the operating system.
AWakeUpDelayAmount   dd ? ;filled out by caller, won't be changed
AwakeUpTime          dd ? ;used by operating system
```

```
AProcessToCall      dd ? ;filled out by caller, won't be changed
ARTag               dd ? ;filled out by caller, won't be changed
AOldLink            dd ?;used by operating system
AESProcessStructure ends
```

Example

```
push  EAX                      ; Points to an AES structure
call  ScheduleSleepAESProcessEvent
add   ESP, 4                    ; Adjust the stack pointer
```

See Also

- ◆ AllocateResourceTag
- ◆ CancelSleepAESProcessEvent
- ◆ ScheduleNoSleepAESProcessEvent

SetHardwareInterrupt

Provides an ISR entry point. Generally used by MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
LONG SetHardwareInterrupt (  
    BYTE hardwareinterruptlevel,  
    void (*InterruptProcedure) (void),  
    struct ResourceTagStructure *Rtag,  
    BYTE endOfChainFlag,  
    BYTE shareflag,  
    LONG *EOIFlag);
```

Parameters

HardwareInterruptLevel

(IN) The hardware interrupt level.

InterruptProcedure

(IN) Has a pointer to the address of the interrupt procedure that will be assigned to the specified interrupt vector.

RTag

(IN) Pointer to the resource tag that the MLI D acquired for interrupts.

endOfChainFlag

(IN) Indicates whether the ISR is to be placed in the front or the back of the queue.

shareFlag

(IN) Indicates whether the MLID and the LAN adapter can share interrupts with other adapters.

EOIFlag

(IN) Has a pointer to a double-word flag. When the routine returns, this flag indicates whether this interrupt will require a second EOI.

Completion Codes (EAX)

00000000h	Successful
00000001h	Invalid Parameter
00000002h	Invalid Sharing Mode
00000003h	Out of Memory

Remarks

SetHardwareInterrupt allocates the specified interrupt and provides an ISR entry point.

Call this routine only at process time. This routine does not suspend the calling process. Interrupts must be disabled and will not be enabled.

The interrupt procedure (ISR) will be called with all the registers preserved, ES and DS initialized, and the direction flag cleared. Because interrupt procedures are called as a near procedure, they should return using a RET.

NOTE: SMP aware MLIDs should use *SetSymmetricInterrupt* instead of *SetHardwareInterrupt*, and NetWare 5 MLIDs should use *BusInterruptSetup* instead of *SetHardwareInterrupt*.

This routine uses three flags:

- ◆ *endOfChainFlag*

If this flag is equal to 0, the ISR is to be placed on the front of the queue (nonshared interrupts should use 0). If the flag is equal to 1, the ISR should be placed at the rear of the queue.

- ◆ *shareflag*

If this flag is equal to 0, the interrupt is nonshareable. If the flag is equal to 1, the interrupt can be shared.

- ◆ *EOIFlag*

If this flag returns with a 0, only one EOI will be required for this interrupt. This flag will be initialized by *SetHardwareInterrupt*. If this flag is not 0, the interrupt is chained and the second PIC will also need an EOI. Always EOI the slave (or secondary) PIC first, and then EOI the master (or primary) PIC second.

Example

```
push OFFSET E0IFlag
push 0 ;Nonshareable interrupt
push 0 ;End of Chain Flag
push InterruptResourceTag ;Pointer to RTag
push OFFSET MyInterruptHandler
push MyIrttterrDtLevel ;Interrupt entry
call SetHardwareInterrupt ;Get interrupt back
add ESP, (B * 4) ;Interrupt number
or EAX, EAX ;Error getting interrupt
jnz MLIDResetExit ;Exit if so
.
.
.
MyInterruptHandler proc near
.
.
.
ret
MyInterruptHandler endp
```

See Also

- ◆ ClearHardwareInterrupt

SetSymmetricInterrupt

Provides an ISR entry point for SMP aware MLIDs. Generally used by MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
LONG SetSymmetricInterrupt (
    LONG hardwareInterruptLevel,
    void (*ServiceRoutine),
    LONG deliveryMode,
    LONG processorNumber,
    BYTE endOfChainflag,
    LONG shareFlag,
    LONG *virtualInterruptNumber,
    struct ResourceTagStructure *Rtag);
```

Parameters

hardwareInterruptLevel

(IN) The IRQ level of the hardware interrupt.

ServiceRoutine

(IN) Pointer to the interrupt procedure that will be assigned to the specified interrupt vector.

deliveryMode

(IN)

Set to 0 to force processor 0 to always handle the interrupt.

Set to 1 to assign the interrupt to a specific processor.

Set to 2 to allow dynamic interrupt assignment among all processors.

processorNumber

(IN) Processor number (0, 1, 2, etc.) that the interrupt will be assigned to; only valid for *deliverMode* = 1.

endOfChainflag

(IN) Flag that indicates whether the ISR is to be placed on the front or the back of the queue.

shareFlag

(IN) Flag that indicates whether the MLID and the LAN adapter can share interrupts with other adapters.

Rtag

(IN) Pointer to the resource tag that the MLID acquired for interrupts.

virtualInterruptNumber

(OUT) Pointer to the virtual interrupt number assigned by SMP.

Completion Codes (EAX)

0	Successful
7	SMP sharing not allowed.
8	SMP chain exceeded the maximum length allowed.
9	SMP invalid parameter.
10	SMP not currently supported.

Remarks

Call this routine only at process time. This routine does not suspend the calling process. Interrupts must be disabled and will not be enabled.

SMP aware MLIDs use this routine to set up an interrupt service routine. `SetSymmetricInterrupt` and `ClearSymmetricInterrupt` must be used in place of `SetHardwareInterrupt` and `ClearHardwareInterrupt` in the NetWare SMP environment.

See Also

- ◆ `ClearSymmetricInterrupt`

- ◆ SetHardwareInterrupt
- ◆ ClearHardwareInterrupt
- ◆ NetWare 5's BusInterrupt procedure

SMPDoEndOfInterrupt

EOIs the PIC/APIC in the NetWare SMP environment. Generally used by MLIDs.

Language: Assembly Language

NetWare Operating System: Not version specific

Entry State

ECX

Contains the *virtualInterruptNumber* returned by *SetSymmetricInterrupt*.

Interrupts

Are disabled.

Call

At process time or interrupt time.

Return State

Interrupts

Are unchanged.

Preserved

All registers except EAX.

Completion Codes (EAX)

None.

Remarks

SMP aware MLIDs use this routine in place of *DoEndOfInterrupt*.

See Also

- ◆ DoEndOfInterrupt

UnRegisterEventNotification

Unhooks the event routine. Generally used by protocol stacks and MLIDs.

Language: C Language

NetWare Operating System: Not version specific

Syntax

```
LONG UnRegisterEventNotification (  
    LONG eventID);
```

Parameters

eventID

(IN) Has the value returned from RegisterForEventNotification.

Completion Codes (EAX)

0	Successful
---	------------

Remarks

UnRegisterEventNotification should be called to unhook your event (callback) routine.

Call this routine when the protocol stack or MLID is being unloaded.

IMPORTANT: *Important:* Do not call this routine from within a routine that was called by *RegisterForEventNotification*.

Example

```
push EventID                ;Unhook from operating system exit  
call UnRegisterEventNotification ;Call operating system unhook  
add ESP, 4                  ;Clear stack
```

See Also

- ◆ `RegisterForEventNotification`

WritePhysicalMemory

Write to a buffer pointed to by a physical address. Generally used by MLIDs.

Language: C Language

NetWare Operating System: NetWare 4 and higher

Syntax

```
LONG WritePhysicalMemory (  
    BYTE *Source,  
    BYTE *Dest,  
    LONG NumUnits,  
    LONG UnitLength);
```

Parameters

Source

(IN) The logical address of the memory to write from.

Dest

(IN) The physical address of the buffer to write to.

NumUnits

(IN) The number of memory units to write.

UnitLength

(IN) Has the size of the memory units:

1= Byte

2= Word

3= DWord

Completion Codes (EAX)

0x00000000	Failure	The routine failed because of a bad parameter.
------------	---------	--

Remarks

WritePhysicalMemory writes to a buffer that is pointed to by a physical address. MLIDs use this routine if they must write data to shared RAM before they call *RegisterHardwareOptions*.

See Also

- ◆ `ReadPhysicalMemory`
- ◆ `RegisterHardwareOptions`

22 Assembling and Linking NLMs

Overview

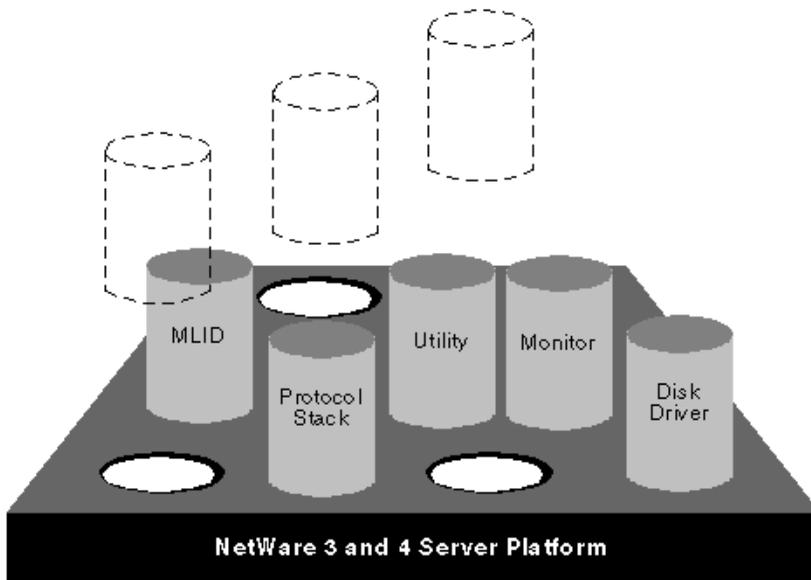
This appendix discusses the steps involved in creating an NLM, the files associated with it, and the steps involved in loading and unloading it.

You should read this appendix if you have never created an NLM before. If you have previously created NLMs, the section "The Definition File" contains keywords that you might want to review.

NetWare Loadable Modules (NLMs)

Using NLMs (NetWare Loadable Modules), network supervisors can dynamically load or unload additional functions to a NetWare 3 and later server without disturbing a fully functioning network. An NLM is an independent module containing a set of functions that can be added to a NetWare server. [Figure 25](#) depicts loadable modules as NetWare building blocks.

Figure 25 Loadable Modules as NetWare Building Blocks



Creating an NLM

All NetWare server MLIDs and protocol stacks are NLMs. NLMs are created in four steps as follows:

1. Create the source file. NetWare NLMs can be written in ANSI C or Intel Assembly Language.
2. Compile or assemble the source file. The source file assembles into an object (.OBJ) file. Novell currently uses the Watcom C Compiler and the 386 v4.10 Protected Mode Assembler by Phar Lap Software, Inc.
3. Link the object file with the NetWare linker. The NetWare linker converts the object (.OBJ) file into a super object file (.NLM).

NOTE: Different kinds of NLMs have different extensions, each signifying the module's function. For example, and MLID has an .LAN extension.

4. Load the NLM as part of the operating system. Using the LOAD console command, the network supervisor can load a .LAN file or a .NLM file into server memory while the file server is running. Once loaded, an NLM works as if it was hardcoded into the NetWare operating system.

The sample NLMs used in this appendix have the same name for source files, object files, definition files, and super object files. However, each file can have a separate name.

The NetWare Linker

The NetWare Linker requires two files:

- ◆ An object file (for this example, LDRIVER.OBJ or PSTACK.OBJ)
- ◆ A definition file (for this example, LDRIVER.DEF or PSTACK.DEF)

The definition file LDRIVER.DEF or PSTACK.DEF contains information describing the object file LDRIVER.OBJ or PSTACK.OBJ. This information includes a list of all NetWare 3 and 4 operating system internal variables and routines that LDRIVER and PSTACK must access after being loaded. To link the object file, type the name of the NetWare Linker (NLMLINK) followed by the name of the definition file:

```
nlmlink ldriver or nlmlinkp ldriver
nlmlink pstack or nlmlinkp pstack
```

The Definition File

Each NetWare loadable module must have a corresponding definition file with a DEF extension. The definition file contains information about the loadable module, including a list of NetWare variables and routines that the loadable module must access. The following sample code illustrates a definition file that can be used to create an MLID, as well as a definition file that can be used to create a protocol stack:

LDRIVER.DEF

```
TYPE 1
DESCRIPTION "NetWare NF9000 v5.30 (930718)"
OUTPUT MyDriverName
INPUT MyDriverOBJFiles
START DriverInitialize
EXIT DriverRemove
REENTRANT
MAP
IMPORT
```

```
GetCurrentTime
SetHardwareInterrupt
```

```
ClearHardwareInterrupt
ParseDriverParameters
RegisterHardwareOptions
DeRegisterHardwareOptions
LSLRegisterMLIDRtag
LSLGetSizedRcvECBRTag
LSLFastRcvEvent
LSLHoldRcvEvent
LSLFastSendComplete
LSLSendComplete
LSLServiceEvents
LSLDeRegisterMLID
LSLUnBindThenDeRegisterMLID
LSLAddProtocolID
```

PSTACK.DEF

```
DESCRIPTION "NetWare 4 Dummy Protocol Driver v1.x (930718)"
OUTPUT MyStackName
INPUT MyStackOBJFiles
START StartProcedure
EXIT ExitProcedure
MAP
VERSION 1.20
COPYRIGHT
EXPORT
```

```
DummyStackRoutine1
DummyStackRoutine2
DummyStackRoutine3
```

IMPORT

```
RegisterForEventNotication
UnRegisterEventNotfication
ScheduleNoSleepAESProcessEvent
CancelNoSleepAESProcessEvent
AllocateResourceTag
CFindResourceTag
CREScheduleLast
QueueSystemAlert
FreeSemiPermMemory
AllocSemiPermMemory
GetCurrentTime
LSLAddProtocolID
LSLReturnRcvECB
LSLGetMaximumPacketSize
```

```
LSLBindStack
LSLGetMLIDControlEntry
LSLGetIntervalMarker
LSLGetStackIDFromName
LSLRegisterStackRTag
LSLSendPacket
LSLUnBindStack
CLSLRegisterStack
CLSLGetPIDFromStackIDBoard
```

The definition file consists of keywords (which can be uppercase or lowercase) followed by data:

TYPE Tells the linker which extension to append to the output file. The default extension is .NLM. A value of 1 specifies .LAN, and a value of 2 specifies .DSK.

DESCRIPTION Tells the linker to save the string following DESCRIPTION in LDRIVER.LAN and PSTACK.NLM. This string describes the loadable module and can be from 1 to 127 bytes long. The MODULES console command which displays a list of currently loaded modules, displays this description string on the file server console.

The standardized string format is shown below:

MLIDs

```
NetWare NE2000 v5.30 (930718) 3Com EtherLink Plus 3c503 v3.59
(930718)
```

Protocol Stacks

```
My Protocol Stack v1.00 (930718)
```

For NetWare 3 NLMs, the version indicator must be a lowercase "v" and must be preceded by two spaces. In both examples above, the date stamp is in the format "YYMMDD" indicating 18 July 1993. This is a byte-length-preceded ASCII string.

NOTE: In NetWare 3 NLMs, the linker parses for ".NLM" or ".LAN" files and looks for the version information, which must be formatted as specified above. If the linker finds the version information, it puts it in the *version* field of the NLM header. If the linker does not find the version information, or the information is formatted incorrectly, or the version keyword is not used, the linker puts zeros in the version field of the NLM header.

NetWare 4 NLMs will use the VERSION keyword, but will not include a version number in the DESCRIPTION. Typing "modules" or "config" at the command line in NetWare 4 displays the version number.

OUTPUT Tells the linker the name of the output file (for example, LDRIVER).

INPUT Tells the linker the names of the object files to look for. You can list the names of several OBJ files on this line. You do not have to list the filename extension. For example: PSTACK or LDRIVER.

START Tells the linker the name of the loadable module's initialization routine. In the case of the example MLID, the routine is *DriverInit*. The sample protocol stack's routine is called StartProcedure. The NetWare loader calls this procedure during load time.

EXIT Tells the linker the name of the loadable module's remove routine. In the case of the example MLID, the routine is *DriverRemove*. The sample protocol stack routine is called ExitProcedure. The UNLOAD command uses this routine to unload the module from file server memory.

REENTRANT Tells the linker that more than one process can be active in the loadable module's code at one time. (See Chapter 2, "ODI Module Design" for more information about re-entrancy.)

MAP Tells the linker to create a map file.

IMPORT Tells the linker which NetWare variables and routines the loadable module must access.

EXPORT Tells the linker that the list following the keyword contains variables and procedure names that are resident in the loadable module. The NLM must make these variables and procedure names available to other loadable modules.

MODULE Tells which loadable modules must be loaded before the current loadable module is loaded. If the necessary loadable modules are not already in file server memory, the loader will attempt to find and load them. If the loader cannot find the necessary modules, it will not load the current module.

You can list multiple files by separating them with "|". The operating system will scan through this list until it finds the file that it needs.

CUSTOM Tells the name of a file that contains custom data. When the linker sees this keyword, it includes the specified file in the output file it is creating.

DEBUG Tells the linker to include debug information in the output file that it creates.

CHECK Contains the name of the loadable module's check procedure. When a module is unloaded, or the server is brought down, the UNLOAD and DOWN console commands call a loadable module's check procedure, if it exists. The check procedure can do anything you want it to do. For example, an MLID's check procedure might check to see if a LAN adapter is currently being accessed and return a nonzero value to the operating system if the adapter is busy. NetWare can then display a message on the console screen warning the console operator (who wants to unload the driver or bring down the server) that an adapter is busy. A protocol stack could do the same type of check.

MULTIPLE Tells the linker that more than one code image of the loadable module may be loaded into file server memory.

COPYRIGHT Tells the linker to include a copyright string in the output file. By placing an ASCII string (from 1 to 252 bytes long) in double quotes after the keyword Copyright, the copyright message will be displayed whenever the module is loaded. "\n" can be used in the string to start a new line. If the Copyright keyword is used, but no string is included, the linker will include a Novell default copyright message.

NOTE: To use the Copyright keyword, you must use either NLMLINEP.EXE, NLMLINKR.EXE, or NLMLINKX.EXE. NLMLINKP.EXE runs in protected mode. (Protected mode means that it uses the memory above the 1MB boundary.) You can use this linker for large object files. NLMLINKR.EXE runs in resident mode. (Resident mode means that it uses the memory below the 16MB boundary). You can use this linker for small object files. NLMLINKX.EXE uses extended memory and can run with an extended memory management device driver loaded.

VERSION Gives the linker the version of the module that should be placed into the NLM's header version field. The format for this keyword is:

```
VERSION MajorVersion, MinorVersion, Rev
```

The version must be separated by commas. MajorVersion is one digit; MinorVersion is two digits. The last comma and the Rev are optional. Rev is a number from 1 to 26, representing a through z. For example: "Version 3.10,1" in the DEF file produces version 3.10a in the output file.

NOTE: The linker automatically sets the date to the date on which the files are linked. You must use either NLMLINKP.EXE, NLMLINKR.EXE, or NLMLINKX.EXE to use the VERSION keyword. (See the previous Note for an explanation of the two linkers.)

You can also embed all definition file information inside a make file and have the make file produce the definition file.

Loading and Unloading

You can load LDRIVER.LAN and PSTACK.NLM at the file server console after the file server is up and running. These NLMs can also be loaded from a floppy, from a directory on a DOS partition of the file server's hard disk, or from the SYS:SYSTEM directory of the NetWare partition. You can then use the LOAD console command to load these NLMs into the file server's memory. The following examples show how to load LDRIVER from all three sources:

```
load a:ldriver
load c:ldriver
load ldriver
```

The protocol stack NLM is loaded in the same manner.

The NetWare Loader resolves the NLM's import list and links the NLM to the NetWare operating system. To unload these NLMs, use the UNLOAD console command as follows:

```
unload ldriver
```

The protocol stack NLM is unloaded in the same manner.

23 Debugging NLMs

Overview

The NetWare operating system includes a debugger to help you debug your code. This appendix includes some basic instructions on using the debugger, setting break points, and changing memory.

The debugger commands vary with different versions of NetWare. Use the H, .H, HB, and HE commands to discover which debugger commands are supported on the Netware version you are using.

The NetWare Debugger

The NetWare operating system includes a built-in debugger that you can use to debug your programs. You can enter the debugger in three ways.

1. Enter 386debug after your NLM abends or GPIs the server.
2. Include INT 3 at some point in your code segment.
3. Type Ctrl-Alt-Shift-Shift-Esc at the console. (Use the left Ctrl key.)

The debugger displays a # command line prompt. The debugger is not case-sensitive.

Once you have entered Debug, enter *.m* to display the addresses of the code and data segments of each NLM that is loaded.

To display a help screen, you can use any of the following commands:

```
.h displays help for .COMMANDS
h displays help for general commands
\h displays help for LSL commands
-h displays help for MSM commands
```

hb displays help for breakpoints
he displays help for grouping operators and expressions

Setting Breakpoints

Four breakpoint registers exist, allowing a maximum of four breakpoints to be set simultaneously.

Breakpoints can be set as follows:

- ◆ Permanent breakpoint using the "B" commands
- ◆ Temporary breakpoint using the "G" command
- ◆ Temporary breakpoint using the "P" command if the current instruction cannot be single stepped.

A breakpoint condition can be any expression. If a breakpoint condition is specified, the condition will be evaluated when the break occurs. If the condition is not true, execution will be resumed immediately without entering the interactive debugger.

(The P and G commands are described in the "Miscellaneous" section of this appendix.)

In a NetWare 5 system, adding an 'L' to the end of the command name used to set permanent breakpoints causes the breakpoint to be set in the current local processor only.

For example, on a multiprocessor system running NetWare 5:

- ◆ typing "b = FC005424" sets a breakpoint at address FC005424 on all processors
- ◆ typing "BL=FC005424" sets a break point at address FC005424 in the current local processor only. (See the "X" command for how to change processors.)

B

Displays all breakpoints that are currently set. For example, entering

b

after the # prompt will result in the following display:

```
#bBreakpoint 0 write byte at FF65623B  
Breakpoint 1 read or  
write byte at 000653BAB  
Breakpoint 2 execute at FFF0SBA3
```

BC <number>

Clears the breakpoint specified by <number>. For example, entering

```
bc <number>
```

after the prompt (using 2 for the number) will result in the following display:

```
#bc 2Breakpoint cleared
```

BCA

Clears all breakpoints. For example, entering

```
bca
```

after the # prompt will result in the following display:

```
#bcaAll breakpoints cleared
```

BD <number>

Disable but do not clear all break points on the breakpoint specified by <number>.

BE <number>

Enable all break points on the breakpoint specified by <number>.

B = <address> {condition}

Sets an execution breakpoint at the specified address when the indicated condition is true. A breakpoint condition can be any expression. If a breakpoint condition is specified, that condition will be evaluated when the breakpoint occurs. If the condition is not true (equals zero), then execution will resume immediately without entering the interactive debugger mode.

For example, entering

```
b = fff8765a
```

after the prompt might result in the following display:

```
# b = fff8765aSet as breakpoint 0
```

BW = <address> {condition}

Sets a write breakpoint at the specified address when the indicated condition is true. For example, entering

```
bw = fff665ab
```

after the prompt will result in the following display:

```
# bw = fff665abSet as breakpoint 1
```

BR = <address> {condition}

Sets a read /write breakpoint at the specified address when the indicated condition is true. For example, entering

```
br =<address> {condition}
```

after the prompt might result in the following display:

```
# br = 0065acf3Set as breakpoint 2
```

BI = address{condition}

Sets an I/O read/write breakpoint at the specified address.

Changing Memory

C <address>

Interactively changes the contents of memory location <address> For example, entering C and an address after the prompt might result in the following display:

```
# c fff6432aFFF6432A (00) = 33FFF6432B (34) = 98FFF6432C (5A)
= .
```

To end interactive mode, type a period.

CD <address>

Same as *C <address>*, except in 32-bit words instead of 8 bit bytes.

C <address> = <number or numbers>

Changes the memory value or values, beginning at <address>, to the specified number or numbers. For example, entering c and some address = some number(s) after the # prompt will result in the following display:

```
#c fff534c5 = 00,00,12,5a,78Change successfully completed
```

CD <address> = <number or numbers>

Same as **C <address> = <number or numbers>**, except in 32-bit words instead of 8 bit bytes.

C <address> = "text"

Places the specified text string beginning at <address>. For example, entering **C <address> = "text"** after the # prompt will result in the following display:

```
# c fff60db3 = "This is a string."Change successfully
  completed
```

Dumping Memory

D <address> {count}

Dumps the contents of memory starting at <address> for <count> number of bytes. For example, entering **D <address> {count}** at the # prompt will result in the following displays:

```
# d fff7765e
```

```
FFF7766E 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
FFF7766E 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
FFF7767e 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
FFF7768e 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
FFF776ge 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
FFF776Ae 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
FFF776Be 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
FFF776Ce 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
FFF776Ee 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
FFF776Fe 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
FFF7770e 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
FFF7771e 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
FFF7772e 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
FFF7773e 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
FFF7774e 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
FFF7775e 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
```

or

```
d fff7765e 3
FFF7765E 00 00 00
```

DD <address> {count}

Same as **D <address> {count}**, except that the dump is displayed as 32-bit words.

DL {+linkoffset}<address> {count}

Display the contents of a linked list at <address> for the count number of bytes, treating {offset} as the offset from <address> to the forward link to the next list element. Use <enter < to dump the next list element.

DDL {+linkoffset}<address> {count}

Same as **DL {offset}<address> {count}**, except that memory is displayed as 32-bit words.

DDS {+linkoffset}<address> {count}

Display the stack symbolically. The default address is [ESP].

Register Manipulations

R

Displays the registers EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, and EIP registers; it also displays all flags. For example, entering R after the # prompt might result in the following display.

```
# REAX=99999999   EBX=00005455   ECX=78787878
   EDX=00060544ESI=00000000   EDI=80868086   EBP=00000000
   ESP=FFF67876EIP=FFF56784   FLAGS=00010002
```

<REG = value>

Changes the specified register to the new value. The command is effective with the EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, and BIP registers. For example, entering <REG = value> after the # prompt will result in the following display:

```
# eax = 8099acb3Register changed
```

F <FLAG > = <value>

Changes the specified flag to the new value (0 or -1). The command is effective with the CF, AF, ZF, SF, IF, TF, PF, DF, and OF flags. For example,

entering F <FLAG > = <value> after the # prompt results in the following display:

```
# f pf = 0Flag changed
```

RC

Display control registers.

I/O

I {BarWord} <PORT>

Inputs a byte, word, or double from a port. For example, entering I {BarWord} <PORT> after the prompt results in the following display:

```
# i 255Port (255) = ff
```

O (BarWord) <PORT > = <VALUE>

Outputs a byte, word, or double value to a port. For example, entering O (BarWord) <PORT > after the prompt results in the following display:

```
# o 255= 78Output completed
```

Miscellaneous

.A

Display the reason for the abend or break.

G {break address(es)}

Begins execution at current EIP and sets temporary breakpoint(s) to address(es). For example enter G (break address(es)) after the # prompt results in the following display:

```
# g fff56784Break at FFF56784 because of go
breakpointEAX=99999999 EBX=00005456 EXC=78787878
EDX=00060544ESI=00000000 EDI=80868086 EBP.00000000
ESP=FFF67876EIP=FFF56784 FLAGS=00010002FFF56784
BB30CE0500MOV EBX, 0005CE30
```

H, HB, HE, .H, -H, \H

Displays help screens.

.L <logical address>

Display page table information for this <logical address>.

.LA <logical address>

Find all aliases at <logical address>.

.LB

The .LB command allows you to get last branch information from the processor. Branches include CALL, RET, JMP, etc. Conditional branches which are not taken are not considered a branch. INTERRUPTS and the IRETD instruction are also considered branches but do not cause debug exceptions.

When a debug exception, INT1 or INT3, occurs, the processor automatically stops tracking last branch information. The branch from the executing code into the debug exception, INT1 or INT3, is not recorded. The OS debugger reads and saves the last branch MSRs for later reference then reenables last branch tracking just prior to the IRETD as the debugger exits from the INT1 or INT3 debug exception. Be aware that the IRETD instruction from a INT1 or INT3 debug exception is recorded as a branch by the processor.

Once enabled, last branch tracking is performed automatically by the processor. This happens without any software intervention and without a performance hit to the processor.

There are two sets of last branch MSRs in the processor. The first is for recording last branch taken. The second is for recording the last branch taken prior to an interrupt or exception. When an interrupt or exception occurs, excluding exceptions INT1 and INT3, the processor takes a snapshot of the information in the last branch MSRs and places it in the last branch prior to interrupt or exception MSRs. This information remains frozen until the next interrupt or exception.

By using the Pentium Pro last branch MSRs and last branch prior to interrupt or exception MSRs the kernel debugger may now provide the developer with a useful execution path back trace. This back trace may be obtained using the .LBT debugger command.

.LBT

The last branch table is a per processor ring buffer. When used with the single step command the buffer provides a history of last branch information. To use

this feature single step through the debugger and observe that the table fills up with the last branch information available from the Pentium Pro MSRs.

.LBTC

As mentioned, the last branch table is a per processor ring buffer. It may be cleared on a per processor basis by typing `.LBTC`. You may change processors using the "X" command.

.LP <physical address>

Find all linear mappings of <physical address>.

.M

Display loaded module names and address.

M <Start >{L len} <byte(s)>

Searches memory for pattern from the start until the length is reached. For example, entering `B>M <Start > {L len}<byte(s)>` at the # prompt results in the following display.

```
# m fff77e50 48 61 72 64
FFF77E50 54 48 45 52 4E 45 54 5F-49 49 00 90 00 00 00 00 ETHERNET_11.....
FFF77F00 00 00 00 00 00 00 90 SS-F7 FF 00 00 00 00 00 ..... kw .....
FFF77F10 48 61 72 64 77 61 72 65-44 72 89 76 65 72 4D 4C HardwareDriverML
```

N <symbolname> <value>

Defines a new symbol with a value. For example, entering `N` at the prompt might result in the following display.

```
# n thissym 0f0f
```

P

Proceeds over the next instruction.

Q

Quits and returns to DOS.

T or S

T traces and S single-steps through the program.

SBON and SBOFF

When stepping on branches taken is enabled, a single step debug command will execute until a branch is taken. The processor stops execution on the instruction which was the target of the branch. All code from one branch to the next is executed without entering the debugger unless the processor hits some other break point. Be aware that if single stepping on branches is enabled by typing SBON in a segment of code between a PUSHFD and POPFD and the trap flag (TF) in the EFLAGS register was not previously set, when the POPFD instruction is executed the trap flag will not remain set and no debug exceptions will be generated from that point on.

U <address> {COUNT}

Unassembles count instructions from address. For example, entering U <address> {COUNT} at the # prompt results in the following display:

```
# u FFF87885 2FFF87885 0000 ADD [EAX],ALFFF87887 0000 ADD
   [EAX],AL
```

V

Views the screens (steps through the screens sequentially).

.V

Display server version.

X <number>

Change to the next available processor or change to the processor specified by <number>.

Z <expression>

Evaluates the expression. For example, typing z and an expression at the # prompt results in the following display.

```
# z 7+8Evaluates to: F
```

? <address>

If symbolic information has been loaded, display the closest symbols to <address> (default [EIP]) are displayed.

NOTE: The D, M, P, S, T, and U commands can be continued or repeated by simply pressing the enter key at the # prompt.

Grouping Operators

These operators and (), [], and {} have a precedence of 0. The grouping operators can be nested in any combination. The size is a data size specifier of type B, W, or D.

(expression) This example causes an expression to be evaluated at a higher precedence.

[size expression] This example causes an expression to be evaluated at a higher precedence and then uses that expression as a memory address. The bracketed expression is replaced with the byte, word, or double word at that address.

{size expression} This example causes an expression to be evaluated at a higher precedence and then uses that expression as a port address. The bracketed expression is replaced with the byte, word, or double word input from the port.

Unary Operators

Table 58 NetWare Debugger Operators

Symbol	Description	Precedence
!	logical not	1
-	2s compliment	1
~	l's compliment	1
*	multiply	2
/	divide	2
%	mod	2
+	addition	3
-	subtraction	3
<<	bit shift right	4
>>	bit shift left	4
<	greater than	5

Symbol	Description	Precedence
>	less than	5
>=	greater than or equal to	5
<=	less than or equal to	5
==	equal to	6
!=	not equal to	6
&	bitwise AND	7
^	bitwise XOR	8
	bitwise OR	9
&&	logical AND	10
	logical OR	11

Ternary Operator

expression1 ? expression2, expression3

If expression1 is true, the result is the value of expression2; otherwise, the result is the value of expression3.

24 Server Command Line Parameters and Keywords

Overview

This appendix contains the keywords and parameters you can use from the command line when loading the MLID. The keywords and parameters specify custom options for the MLID.

MLID Keywords

When loading an MLID on the server console command line, you can use the keywords listed below as parameters. LAN Parameters and I/O parameters do not have a set order, and the two can be mixed together. For example, you could enter a command similar to this:

```
LOAD LDRIVER FRAME=ETHERNET_802.3, PORT=300,  
      NODE=2608C760361, INT = 3
```

The commas used in the load command are optional.

DMA

If your LAN adapter supports DMA, this is the Direct Memory Address channel that the adapter should use for data transfer to memory. This value is available to *ParseDriverParameters*.

```
DMA = n  
DMA CHANNEL = n
```

SLOT

System-wide, unique Hardware Instance Number (HIN) that may be the physical slot number on a slot based bus such as PCI, PC Card, EISA, MicroChannel, or another uniquely assigned number. This value is available to *ParseDriverParameters*.

SLOT = n

PORT

This is the I/O mapped address base that the user wants the board to use. This value is available to *ParseDriverParameters*.

I/O PORT = n

PORT = n

MEMORY ADDRESS

This is the beginning address of the shared RAM that the board can use. This value is available to *ParseDriverParameters*.

MEMORY ADDRESS = n

MEM = n

MEMORY LENGTH

This is the size of the shared memory buffer that was specified by Memory Address. This value is available to *ParseDriverParameters*.

MEMORY LENGTH = n

INTERRUPT NUMBER

This is the interrupt number that the LAN adapter is expected to use to awaken the ISR routine. This value is available to *ParseDriverParameters*.

INTERRUPT NUMBER = n

INT = n

NODE

This is the node address that the board is to use; this address should override the default address on the board if any. This value is available to *ParseDriverParameters*.

NODE = nnnnnnnnnnnn

RETRIES

This is the number of send retries that the MLID should use in its attempts to send packets. This value is available to *ParseDriverParameters*.

RETRIES = n

RETR = n

FRAME

This is the frame type that the next instance of the MLID will be set to. This value is available to *Parse.DriverParameters*.

FRAME = type

25

Writing Protocol Stacks for NetWare SFT III

Overview

This appendix contains a basic description of NetWare SFT III. It also contains the information, LSL support routines and IOCTLs you need in order to write a protocol stack for a NetWare 4 SFTIII system.

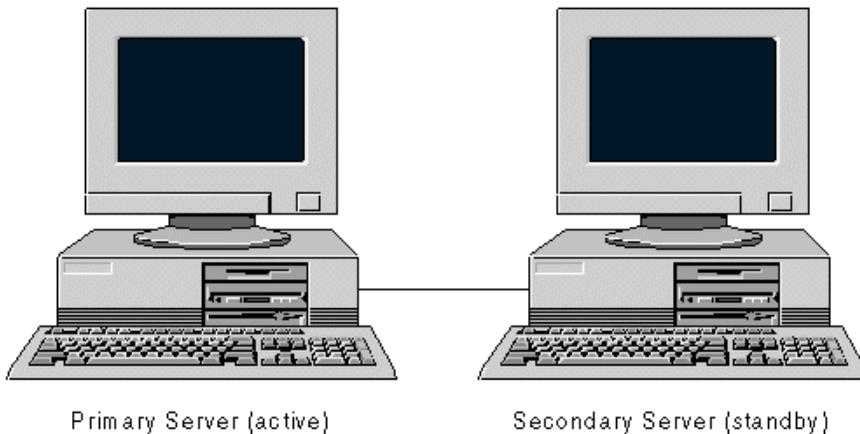
Introduction to NetWare SFT III

SFT III (System Fault Tolerance) is the latest in a series of products that provide fault tolerance. SFT I provided redundant directory entry tables and FATs (File Access Table), volume consistency checking, read-after-write verification and the Hot Fix feature. SFT II provided disk mirroring, disk duplicating, and the TTS (Transaction Tracking System). SFT III includes all of the features of its predecessors and introduces an architecture which allows mirrored servers.

Mirrored Server Implementation

The SFT III architecture allows an entire server (memory image and disk contents) to be mirrored on another server. This duplicate (or secondary) server quickly takes over network operations if the primary server fails. The failure of the primary server and the assumption of the network by the mirrored server is invisible to the client workstations. [Figure 26](#) illustrates mirrored servers.

Figure 26 Mirrored Servers



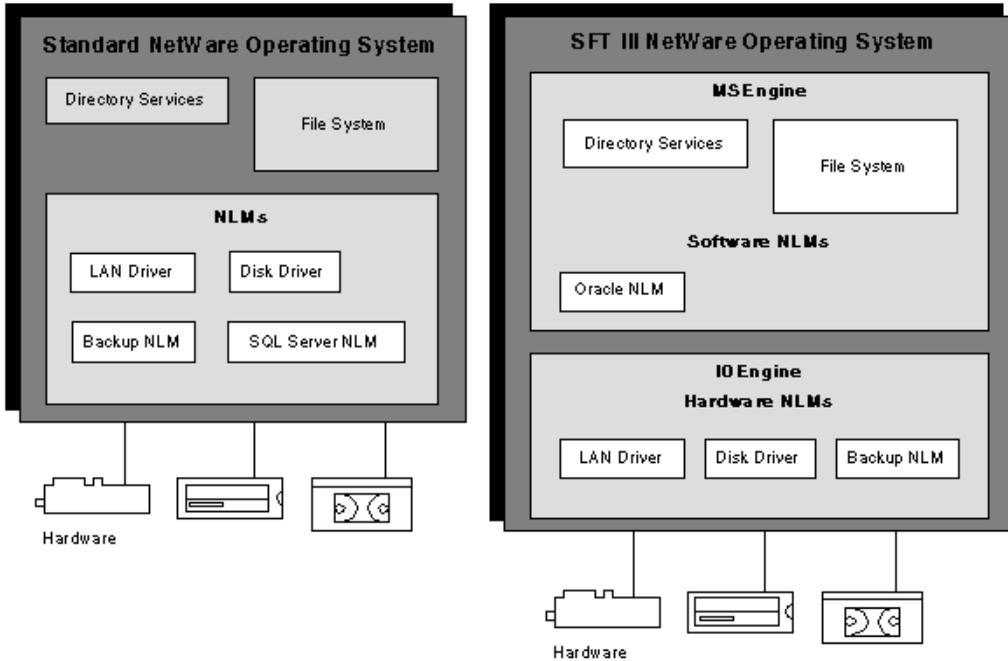
Primary and Secondary Servers

One of the servers acts as the primary server; the other acts as the secondary server. The designation of primary and secondary is dynamic. The server that is active the longest acts as the primary server. The other server takes the secondary role and uses most of its CPU cycles to maintain synchronization with the primary server. The secondary server must mirror every change on the primary server so that it can instantaneously take over the network.

MSEngine and IOEngine

To enable mirrored servers, the NetWare operating system has been split into two pieces: the Mirrored Server Engine (MSEngine) and the Input / Output Engine (IOEngine). Most of the operating system, including the bindery (NetWare 3), directory services (NetWare 4), and the file system is in the MSEngine. The remainder of the operating system (the part that deals with the hardware) is in the IOEngine. These two parts of the SFT III operating system provide all of the functions of the traditional NetWare system (see [Figure 27](#)).

Figure 27 Correspondence of Operating System Standard NetWare and SFT III NetWare File



The mirrored servers do not have to be identical, but they should be evenly matched in terms of CPU speed, memory and storage capacity.

Because the two servers could be different, the IOEngine is not mirrored. This allows the two servers to contain different types of LAN adapters and disk drives.

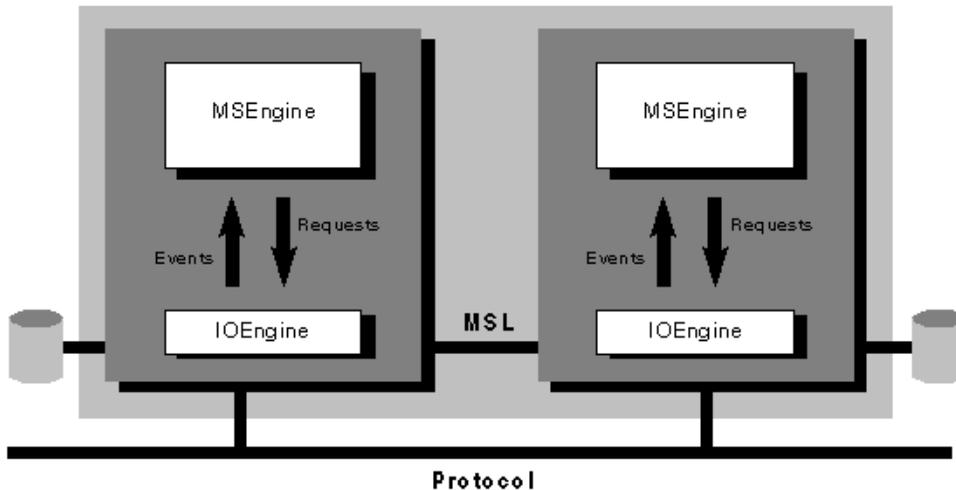
Events and Requests on Mirrored Servers

The MSEngine contains the non-hardware-specific portions of the operating system and is mirrored on both machines. As the primary server's IOEngine receives input from external sources (for example, from the LAN or disk channel, or from programs running in the IO Engine), it converts them to mirrored server events and submits them to the primary copy of the MSEngine.

At the same time, these events are sent across a special high-speed link called the Mirrored Server Link (MSL) to the secondary MSEngine copy. Both

copies of the MSEngine process the event and generate identical responses in the form of IOEngine requests, as illustrated in [Figure 28](#).

Figure 28 Mirrored MSEngines Respond with Identical Requests to Events from IOEngine



NOTE: [Figure 28](#) also illustrates that the two MSEngines form a single logical entity. This logical unit has one network address.

Both the primary and secondary server's MSEngines receive exactly the same sequence of events and respond with identical requests; therefore, both servers mirror each other. The two servers are not actually in "lockstep" at the instruction level; they may be slightly skewed at any specific instant. However, the two servers are in 'lockstep' at the event level. Because events are always given to the secondary MSEngine before allowing the primary MSEngine to complete, no data or state information is lost if the MSEngine fails. The SFT III operating system manages the synchronization of the servers so that the secondary server will always converge to exactly the same state as the primary server.

Mirrored Servers and PC Clients

Each network client sees only one instance of the MSEngine. Both copies of the MSEngine share the same internal network address, but only the primary IOEngine advertises; itself as the route to the MSEngine.

Client PCs communicate over the network only with the primary IOEngine, which copies the messages received over the MSL (Mirrored Server Link) to the secondary IOEngine. The messages are then given to the copy of the MSEngine in both servers, and result in an MSEngine request to send a response packet back to the client. The MSEngine outputs are compared; then only the primary server sends the response packet. The secondary server discards the response.

NetWare SFT III and Existing Applications

Most applications that do not interface directly to the server's hardware can run unmodified in the NetWare SFT III environment.

Device drivers (LAN and disk drivers) and server application that deal with the hardware (print spoolers and backup/restore applications) reside in the IOEngine and are not mirrored by SFT III. Current NetWare LAN drivers can run unmodified in SFT III's IOEngine.

Protocol Stacks and NetWare SFT III

NetWare SFT III Basic Architecture

NetWare SFT III uses the ODI architecture, thus allowing device drivers and protocol stacks to be independent of each other and of the underlying media or topology. SFT III extends the ODI architecture to support the division of the NetWare operating system into the IOEngine and the MSEngine.

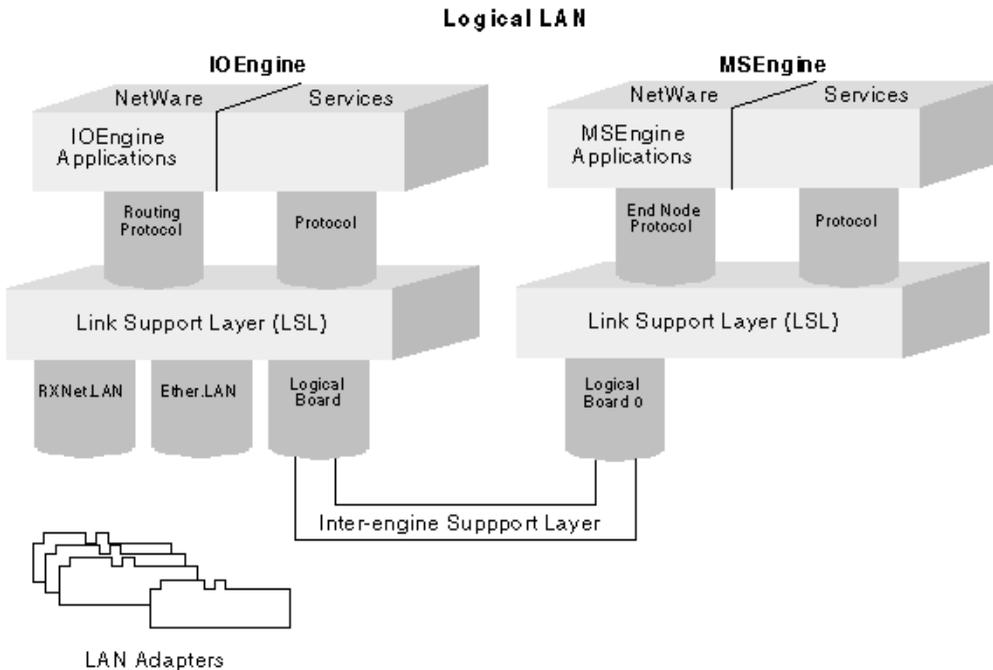
Inter-Engine Support Layer

The support layer in the NetWare SFT III incorporates a two-phased interface. (This interface operates in much the same way as the LSL). This two-phased interface allows logical structures such as file systems or protocol stacks to run unaltered the MSEngine despite the fact that all device drivers are loaded in the IOEngine. Conversely, device drivers can run unaltered in the IOEngine, despite the fact that the logical structures they are supporting reside in the MSEngine. Standard NetWare device driver compatibility is maintained by the insertion of an inter-engine support layer between the upper and lower interfaces of the device driver support layers.

The inter-engine layer resides in the middle of all SFT III support layers. Data submitted to the support layer through its lower-level interface is converted

into MEngine events and submitted to the MEngine side of the support layer. Data submitted to the support layer through its high-level interface is converted into IOEngine requests and submitted to the IOEngine side of the support layer. Figure 29 provides a NetWare Server Protocol Stacks and MLIDs conceptual diagram of protocol communications through this interface.

Figure 29 Logical LAN Conceptual Model of Protocol Communications in SFT III



Protocol Stacks and the Inter-Engine Support Layer

The inter-engine layer in SFT III means that the protocol stacks are loaded twice: once in the IOEngine and a second time in the MEngine. The IOEngine's protocol stack operates as a router and communicates with the MPI portion of the LSL. The MEngine's stack operates as an end node and communicates with the application above the LSL. The inter-engine support layer is conceptually placed between these two implementations of the protocol stack and handles their communications over a virtual network. This virtual

network uses a logical board with a proprietary frame type called VIRTUAL_LAN.

NOTE: The protocol stack handles communications between the IOEngine and the MEngine just as it would any other communications between LANs; it fills out an ECB and calls *LSLSendPacket*. The following discussion illustrates this communication.

The Protocol Stack NLM

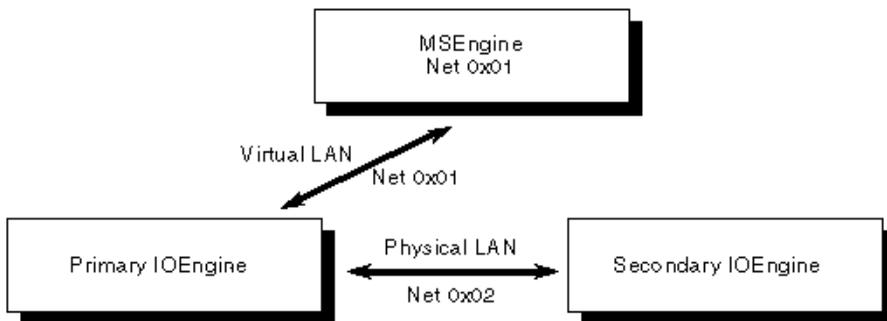
SFT III for NetWare 3 only supports the IPX protocol stack. However, SFT III for NetWare 4 supports all currently supported protocol stacks for NetWare 4.

Additional Protocol Stack Capabilities

A protocol stack NLM for SFT III requires the additional capability of performing inter-engine communications. When developing the protocol stack, remember that both physical instances of the MEngine combine to form a single logical unit with one network address.

Protocol communication among the primary IOEngine, secondary IOEngine, and MEngine is completely normal, with the exception that only the primary IOEngine can route packets to the MEngine network. Likewise, the MEngine knows about the LAN configuration of the primary IOEngine, but not about the configuration of the secondary IOEngine.

Figure 30 Logical SFT III Network



IPX Protocol Stack Communication

An IPX protocol stack handles the communications depicted in [Figure 30](#) as follows: An application running on the secondary IOEngine sends a packet to an application running on the MEngine by placing 0x01 in the network field of the IPX header *destinationAddress* field, the logical board number associated with the primary IOEngine's virtual MLID in the ECBs *boardNumber* field, and 0x02 in the ECBs *immediateAddress* field.

NOTE: For more information about NetWare SFT III see "Implementing Fault Tolerance with NetWare SFT III", A Novell Research Report (August 1993), available from Novell's Systems Research Department.

Developing Protocol Stacks for SFT III

Protocol IDs for the VIRTUAL_LAN Frame Type

As [Figure 29](#) illustrated, the two conceptual portions of the protocol stack communicate with each other by using Logical Board 0 over a virtual LAN. Logical Board 0 uses a proprietary frame type called VIRTUAL_LAN. (In standard Netware, the server uses Logical Board 0.) The inter-engine support layer contains a vectoring function to direct communication between the protocol stacks. This vectoring function operates much like the LSL. Therefore, the protocol stacks need a protocol ID for the VIRTUAL_LAN frame type. The inter-engine support layer uses this protocol ID to route communications. Like normal inter-LAN communications, the protocol ID is placed in the ProtocolID field of the ECB. For simplicity, these protocol IDs should be the same protocol IDs as those used on the medium the protocol most commonly uses, if this is possible. The following table defines these protocol IDs.

Table 59 Protocol IDs for the VIRTUAL_LAN Frame Type

Protocols	ProtocolID	Comment
IPX/SPX	00 00 00 00 00 00	Original definition
Apple-Talk	00 08 00 07 80 9B	Same as 802.2 SNAP; Apple-Talk
AARP	00 00 00 00 80 F3	Same as 802.2 SNAP; Apple-Talk
ARP	00 00 00 00 08 06	Same as Ethernet II; TCP/IP

Protocols	ProtocolID	Comment
RARP	00 00 00 00 80 35	Same as Ethernet II; TCP/IP
IP	00 00 00 00 08 00	Same as Ethernet II; TCP/IP
XNS	00 00 00 00 06 00	Same as Ethernet II; XNS
RPL	00 00 00 00 00 FC	Same as 802.2; Find Found RPL
SNA	00 00 00 00 00 04	Same as 802.2; SNA
NetBIOS	00 00 00 00 00 F0	Same as 802.2; NetBIOS

Other protocols that are not yet defined use the same PIDS they use on their common medium.

IMPORTANT: Protocol IDs must be unique. If the protocol ID cannot be assigned as per the above table, the protocol must be assigned another unique ID.

802.2 LLC frames that use Type I and Type II frames use the same guidelines as those defined in ODI Supplement: Frame Types and Protocol IDs. The guidelines are summarized as follows:

802.2 Frame Type	ECBs Protocol ID Field
Type I	02 00 00 DS SS CO
Type II	03 00 DS SS CO CI

NOTE: DS is the DSAP value; SS is the SSAP value; CO is the Control 0 value, and CI is the Control 1 value.

Nonrouting Protocol Stacks on SFT III

The above protocol communications described in [Figure 29](#) assumes a routing protocol. This protocol essentially routes packets between the external and the internal networks during normal operations. The routing implementation of the protocol stack runs in the IOEngine and the end node implementation of the stack runs in the MS engine.

Protocols such as native NetBIOS and LLC are nonrouting protocols and must handle routing issues differently in order to run in a NetWare SFT III system.

We recommend developing nonrouting protocols by using the same physical adapter node address on both machines.

Using a MAC Layer Bridge

To use this solution, you must develop code that runs a MAC layer bridge between the physical LAN and the internal virtual LAN. The result of this is that to the outside world, the source node address of the received packets will be the internal server node address. This internal node address must fit within the 802.2 locally administered address guidelines. (The current internal node address of 1 may not always work.)

Advantages

This method has the following advantages:

- ◆ Novell is currently designing and developing a MAC layer bridge system.
- ◆ Novell is currently shipping a source routing bridge.
- ◆ This method fits into the same architectural design implemented by routing protocols.

Disadvantages

This method has the following disadvantages:

- ◆ Dumb adapters could experience degraded performance.
- ◆ A MAC layer bridge requires adapters to use promiscuous mode. Therefore, some adapters would use all of the system bus bandwidth.
- ◆ Source routing implementations that assume that a route cannot change during a session would lose the session during switch overs.

Using the Same Physical Node Address

Using a locally administered node address (node address override) causes the adapters in both machines of the mirrored server pair to appear as the same node address to non-routing protocols.

IMPORTANT: In the case of Token-Ring or FDDI, the secondary adapter must remain in a shutdown state until the switch over occurs.

Advantages

This method has the following advantages:

- ◆ Adapters do not need to run in promiscuous mode; therefore, performance is not impacted.
- ◆ Software changes are minimal.
- ◆ Protocols that cannot handle route changes would still work after a switch over.

Disadvantages

This method has the following disadvantages:

- ◆ This solution does not work in the case of Token-Ring and FDDI if the server fails in such a way that the LAN adapter is still inserted and active in the ring.
- ◆ Token-Ring does not allow two adapters on the same ring with the same node address.
- ◆ This solution assumes that the secondary server performs no activity on the wire under normal operations, thus enabling it to work with the LAN adapter in a shutdown state.
- ◆ This solution also assumes that the two machine's adapters are connected to the same physical wire segment. Therefore, if you expect the mirrored server to be miles apart from each other on different physical LANs, this solution would be extremely difficult to implement. (One of NetWare SFT III unique features is to allow mirrored servers on different physical LANs.)

LSL Routines, IOCTLs, and OS Routines for SFT 11 Protocol Stacks

The routines described on the following pages allow protocol stacks to support the functions required by SFT III on NetWare_4.

SFT III Status Values Defined

The following SFT III Status values are used by the following routines:

```
#define STF3STA_SUCCESSFUL          0
#define STF3STA_MIRROR_NOT_ACTIVE  1
#define STF3STAT_NO_PARTNER        2
#define SFT3_OUT_OF_RESOURCES      3
#define STF3_STAT_NOT_SUPPORTED    -1 (0xFFFFFFFF)
```

LSLSendProtocolInfoToOtherEngine

Description

Allows two-way communication between the primary IOEngine and MSEngine.

Syntax

```
LONG LSLSendProtocolInfoToOtherEngine (  
    LONG ProtocolNumber  
    Byte *ProtocolInfo  
    LONG Length  
    void *InfoSendCallBack  
    BYTE *ProtocolInfo );
```

Input Parameters

- ◆ *ProtocolNumber*
has the protocol stack ID.
- ◆ *ProtocolInfo*
has a pointer to the information to be sent.
- ◆ *Length*
has the number of bytes pointed to by ProtocolInfo.
- ◆ *InfoSendCallBack*
has a pointer to a function to be called when the information pointed to by ProtocolInfo has been sent.
- ◆ *ProtocolInfo*
has a pointer to the information that was sent.

Output Parameters

None.

Completion Codes (EAX)

STF3STAT_SUCCESSFUL	The operation completed successfully.
STF3STAT_MIRROR_NOT_ACTIVE	The mirrored server engine was not active.

STF3STAT_NO_PARTNER	The server does not have a mirrored partner.
SFT3STAT_OUT_OF_RESOURCES	There was no memory to queue the request.
STF3STAT_NOT_SUPPORTED	This function was not supported.

Remarks

LSLSendProtocolInfoToOtherEngine is used with the NetWare SFT III operating system. This routine allows two-way communication between the protocol stacks in the primary IOEngine and in the MSEngine.

Protocol stacks might use this call because they need to send information to each other after binding or unbinding operations. The operating system also sometimes signals the protocol stack to give all needed protocol information to the other engine.

IMPORTANT: A notification that the information was sent does not imply that the destination received the information. These routines are connectionless sends.

NOTE: *LSLSendProtocolInfoToOtherEngine* and *LSLSendProtocolInfoToPartner* are available in all three server engines (IOEngine, MSEngine and regular server). If these APIs are unsupported, the engine will return -1 (all bits set).

LSLSendProtocolInfoToPartner

Description

Sends information to the other IOEngine.

Syntax

```
LONG LSLSendProtocolInfoToPartner (
    LONG    ProtocolNumber
    Byte    *ProtocolInfo
    LONG    Length
    void    (*InfoSendCallBack ) (
        LONG    Reserved
        BYTE    *ProtocolInfo ) );
```

Input Parameters

- ◆ *ProtocolNumber*
has the protocol stack ID.

- ◆ *ProtocolInfo*
has a pointer to the information to be sent.
- ◆ *Length*
has the number of bytes pointed to by *ProtocolInfo*.
- ◆ *InfoSendCallBack*
has a pointer to a function to be called when the information pointed to by *ProtocolInfo* has been sent.
- ◆ *Reserved*
This parameter is reserved.
- ◆ *ProtocolInfo*
has a pointer to the information that was sent.

Output Parameters

None.

Completion Codes (EAX)

STF3STAT_SUCCESSFUL	The operation completed successfully.
STF3STAT_MIRROR_NOT_ACTIVE	The mirrored server engine was not active.
STF3STAT_NO_PARTNER	The server does not have a mirrored partner.
SFT3STAT_OUT_OF_RESOURCES	There was no memory to queue the request.
STF3STAT_NOT_SUPPORTED	This function was not supported.

Remarks

LSLSendProtocolInfoToPartner is used with the NetWare SFT III operating system. The protocol stack calls this routine whenever it must send information to the other IOEngine.

Protocol stacks might use this call because they need to send information to each other after binding or unbinding operations. The operating system also sometimes signals the protocol stack to give all needed protocol information to the other IOEngine.

IMPORTANT: A notification that the information was sent does not imply that the destination received the information. These routines are connectionless sends.

NOTE: *LSLSendProtocolInfoToOtherEngine* and *LSLSendProtocolInfoToPartner* are available in all three server engines (IOEngine, MEngine and regular server). If these APIs are unsupported, the engine will return -1 (all bits set).

Ctl6SFTIIIExchange

Protocol Stack IOCTL

Description

Allows protocol stacks to exchange state information.

Syntax

```
LONG SFTIIIExchange(  
    SFTIIIExchangeNode *pSFTIIIIXNode );
```

Input Parameters

- ◆ *pSFTIIIIXNode*

is a pointer to a SFT III Exchange Node structure. This structure is defined as follows:

```
typedef struct_SFTIIIExchangeNode_  
    LONG SubFunction,  
    void *Parameter1,  
    void *Parameter2,  
}SFTIIIExchangeNode;
```

The structure fields are defined as follows:

- ◆ *SubFunction*

is a function number defining the state and contents of *Parameter1* and *Parameter2* and the command to be executed. The *SubFunction* values and the subsequent state and contents of *Parameter1* and *Parameter2* are defined below.

- ◆ *Parameter1*

is an input whose state and contents is defined by the *SubFunction* field.

◆ *Parameter2*

is an input whose state and contents is defined by the SubFunction field.

Output Parameters

None.

Completion Codes (EAX)

ODISTAT_SUCCESSFUL	The operation completed successfully.
ODISTAT_BAD_COMMAND	The protocol stack does not support this control function.
Any bit set	The operation failed.

Remarks

Ctl6SFTIIIExchange allows protocol stacks in the NetWare server environment to exchange state information with each other. This routine allows the protocol stack to communicate with its mirrored partner, or with its IO or MS portion.

The following table defines the SubFunction values:

IMPORTANT: These functions are called at interrupt time with the interrupts disabled. These functions cannot relinquish control or change the interrupt state.

Table 60 SubFunction Values and the Corresponding Parameter State or Contents

Value	Description	Parameter/ Parameter2	Comment
0	<i>GetProtocolInfo</i> destination. Allows communication between the IOEngines. Returns pointers to <i>ProtocolInfo</i> and Length with correct values.	BYTE *ProtocolInfo LONG Length	returns a pointer to a buffer where the protocol information should be placed. number of bytes of protocol information that will be written. May be modified.

Value	Description	Parameter/ Parameter2	Comment
1	Information updated. Allows communication between the IOEngines. Signals an update of protocol information from the other IOEngine.	BYTE *ProtocollInfo LONG Length	pointer to the information just received. number of bytes pointed to by ProtocollInfo.
2	Send information to partner. Allows communication between the IOEngines. Indicates to the protocol stack that all needed protocol information should be sent to the other IOEngine by the use of the function <i>LSLSendProtocollInfoToPartner</i> .	BYTE *ProtocollInfo LONG Length	
3	Partner unload. Indicates to the protocol stack that it should clear any information it had that concerned the other IOEngine.	undefined undefined	
4	Link to partner gone. Still primary. Indicates to the protocol stack that it should clear any information it had that concerned the other IOEngine.	undefined undefined	
5	Link to partner gone. Becoming primary. Performs any operation necessary to take over as the new primary server. (For example, it might need to send false packets in behalf of the dead partner in order to facilitate router switch overs.)	undefined undefined	
10	<i>GetProtocollInfo</i> destination. Allows communication between the MEngine and the IOEngine. Returns pointers to ProtocollInfo and Length with correct values.	BYTE *ProtocollInfo LONG Length	returns a pointer to a buffer where protocol information should be placed. number of bytes of protocol information that will be written. May be modified.
11	Information updated. Allows communication between the MEngine and the IOEngine. Signals an update of the protocol information from the other engine.	BYTE *ProtocollInfo LONG Length	pointer to the information just received. number of bytes pointed to by ProtocollInfo.

Value	Description	Parameter/ Parameter2	Comment
12	Send information to the other engine. Allows communication between the MEngine and the IOEngine. Indicates to the protocol stack that all needed protocol information should be sent to the other engine by calling <i>LSLSendProtocolInfoToOtherEngine</i> .	undefined undefined	
13	Protocol stack in other engine unloaded. Indicates to the protocol stack that it should clear any information it had that concerned the other engine.	undefined undefined	

26 Revision History

May 2000 Release - Doc Version 1.21

- ◆ Updated the Remarks section of the `LSLRegisterStackSMPSafe` function in `LSL Support Routines (Assembly Language)`.

27

Glossary

Abort

Termination of a CPU process when the process cannot complete.

Adapter

A circuit board driven by software. In the context of ODI LAN Driver development, an adapter refers to a physical board. See also NIC. MLID, Driver.

Address

A unique group of characters that correspond 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 or more software modules to exchange information.

ARP (Address Resolution Protocol)

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

Asynchronous Process

A process that does not depend on timing signals.

Bit

A binary digit that can only be 0 or 1.

Broadcast

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

Buffer

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

Bus

A hardware interface used to transfer data between devices.

Byte

A sequence of 8 bits.

CAM (Content Addressable Memory)

Memory data space on the adapter. In ODI, this memory holds the group addresses that the adapter filters.

Completion Code

The code returned by a routine indicating whether the routine completed successfully or failed. Control Block A data structure 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 ODI, 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 and/or the transfer of data. 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 program interrupt controller (PIC) indicating the end of the 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, or the recurrence of an event that rescheduled itself with the AES.

Ethernet

A wire medium usually used in a bus topology.

FDDI (Fiber Distributed Data Interface)

A dual ring topology.

Frame

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

HSM (Hardware specific Module)

One of three modules comprising the LAN driver toolkit. The developer writes the HSM to handle all the 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)

An MLID procedure that performs a specific action such as Add Multicast Address, Reset, Shutdown, 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)

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

LAN (Local Area Network)

Two or more computers (usually located in the same building) connected together for communication and resource sharing.

LSL (Link Support Layer)

The ODI layer that directs multiple incoming and outgoing data packets from the MLID to the designated protocol stack, and vice versa.

MAC Header (Media Access Control Header)

The packet header that controls the transmission of the packet through the 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 an ODI LAN driver.

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.

PCI

An industry standard 32-bit (eventually to be 64-bit) processor independent bus architecture.

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 saved on the network.

Pseudocode

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

RAM (Random Access Memory)

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

RCB (Receive Control Block)

A data structure used by the MLID to receive data.

ROM (Read Only Memory)

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

Shared RAM

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

Source Address

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

SPX (Sequenced Packet Exchange)

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

Stubbed Routine

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

Synchronous Process

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

TCB (Transmit Control Block)

The data structure used by the MLID to transmit data.

TCP (Transmission Control Protocol)

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

Token-Ring

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

TSM (Topology Specific Module)

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

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.

