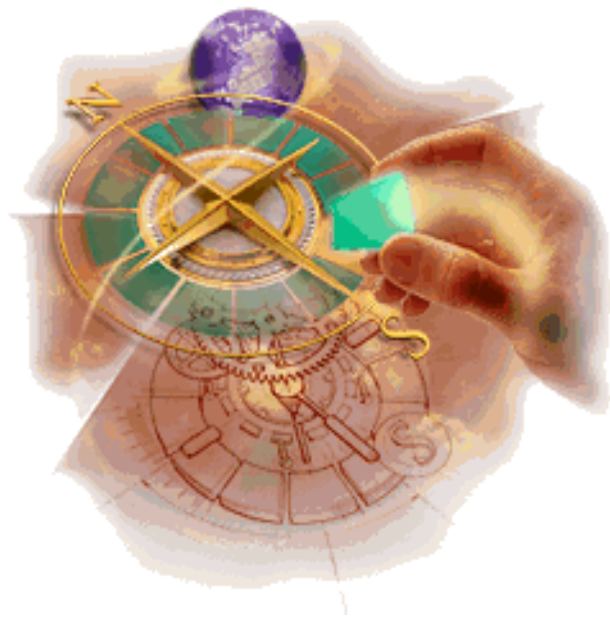


ODI Specification:

Hardware Specific Modules (HSMs) (32-bit Assembly Language)

SPEC VERSION 3.31, DOC VERSION 1.12



Novell.
ODI LAN Driver Documentation

disclaimer

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

Further, Novell, Inc. makes no representations or warranties with respect to any NetWare 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 NetWare software, at any time, without any obligation to notify any person or entity of such changes.

trademarks

Novell and NetWare are registered trademarks of Novell, Inc. in the United States and other countries.

The Novell Network Symbol is a trademark of Novell, Inc.

Macintosh is a registered trademark of Apple Computer, Inc.

DynaText is a registered trademark of Electronic Book Technologies, Inc.

Microsoft is a registered trademark of Microsoft Corporation.

Copyright © 1993-1998 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. 5,157,663; 5,349,642; and 5,455,932. U.S. and International Patent Pending.

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

**ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)
March 26, 1998**

Contents

Preface

1 Introduction

Open Data-Link Interface	1-1
Link Support Layer	1-2
Multiple Link Interface Drivers	1-2
NetWare Loadable Modules	1-3
Driver Modules	1-4
Novell Provided Support Modules	1-4
Media Support Module	1-4
Topology Specific Module	1-4
Developer Provided Module	1-6
Hardware Specific Module	1-6
Loading Driver Modules	1-7
Development Process	1-9
ODI Supplements	1-9
Driver Related Files	1-9
Source Files	1-9
Include Files	1-9
Linker Definition File	1-10
Driver Configuration File	1-10
Installation Information File	1-10

2 HSM Overview

HSM Components	2-1
HSM Procedures	2-1
Initialization and Removal	2-3
Packet Reception and Transmission	2-3
Multi-Operating System Provisions	2-4
I/O Control Procedures	2-4
Timeout Detection	2-5
HSM Data Structures and Variables	2-5
HSM Design Considerations	2-6

Hardware Issues	2-6
Network Interface Controllers.	2-6
Data Transfer Mode	2-6
Bus Type	2-7
Coding Issues	2-8
Multi-Tasking, Non-Preemptive OS.	2-8
32-Bit Protected Mode	2-8
Interrupt Service Routine	2-8
OS Calls to the Driver	2-9
Execution Times	2-9
Code and Data Space	2-11
Frame Data Space	2-12
Adapter Data Space	2-12
Adapter Code Space	2-12
Reentrancy	2-12
Recommended Support	2-13
Multicast Addressing	2-13
Promiscuous Mode	2-13
Optional Support	2-14
Hub Management.	2-14
Source Routing	2-14
Router	2-14

3 HSM Data Structures and Variables

Introduction.	3-1
Global Data Access	3-1
Specification Version	3-3
Driver Parameter Block	3-4
Driver Parameter Block.	3-5
Driver Configuration Table	3-13
Driver Frame Data Space	3-13
Example Template for the Driver Configuration Table (based on the NE2000)	
3-15	
MLIDModeFlags Bit Map	3-26
MLIDFlags Bit Map	3-28
MLIDSharingFlags Bit Map.	3-30
Driver Adapter Data Space.	3-33
Driver Statistics Table	3-34
CounterMask Bit Maps	3-40
Media Specific Counters	3-41
Token-Ring	3-41
Ethernet.	3-44
FDDI	3-45

ii ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

FDDI TSM and Bit Swapping Changes	3-47
RX-Net.	3-48
Driver Firmware	3-49
Driver Keywords	3-51
Driver Keyword Enhancements	3-53

4 MSM/TSM Data Structures and Variables

Introduction	4-1
MSM Global Variables	4-1
MSMBitSwapTable	4-1
MSM Equates	4-2
MSMVirtualBoardLink	4-2
MSMStatusFlags	4-3
MSMTxFreeCount	4-4
MSMPriorityTxFreeCount	4-6
MSMMaxFrameHeaderSize	4-6
MSMPhysNodeAddress	4-8
Data Structures	4-9
Receive Control Blocks	4-11
Fragmented RCB	4-12
Non-Fragmented RCB	4-14
Transmit Control Blocks	4-16
TCB for Ethernet, Token-Ring, and FDDI	4-17
TCB for RX-Net	4-19
Fragment Structure	4-22
Event Control Blocks	4-24
Receive ECBs vs RCBs	4-26
Transmit ECBs vs TCBs	4-27

5 HSM Procedures

Introduction	5-1
Initialization	5-3
DriverInit.	5-3
Register with the MSM / TSM	5-3
Determine Hardware Options	5-4
Register Hardware Options	5-7
Initialize the Adapter	5-7
Register with the LSL	5-8
Setup a Board Service Routine	5-8
Schedule Timeout Callbacks	5-9
DriverInit Pseudocode	5-10
Packet Reception	5-13

Reception Methods	5-13
Programmed I/O and Shared RAM	5-13
DMA and Bus Master	5-15
RX-Net	5-17
Board Service	5-19
DriverISR	5-19
Receive Event	5-19
Receive Error	5-20
Transmit Complete	5-20
Transmit Errors	5-21
Using Shared Interrupts	5-21
DriverISR Pseudocode	5-24
DriverPoll	5-26
Packet Transmission	5-27
Transmission Methods	5-27
Programmed I/O, Shared RAM, and Host DMA	5-28
Bus Master	5-28
Priority Transmission Support	5-31
DriverSend	5-33
Driver Priority Queue Support	5-35
Multi-Operating System Support	5-36
Critical Sections	5-37
DriverEnableInterrupt	5-38
DriverDisableInterrupt	5-39
DriverDisableInterrupt2	5-40
Control Procedures	5-41
DriverReset	5-43
DriverShutdown	5-45
DriverMulticastChange	5-47
Adapter Multicast Filtering	5-48
DriverPromiscuousChange	5-50
DriverStatisticsChange (optional)	5-52
DriverRxLookAheadChange (optional)	5-53
DriverManagement (optional)	5-54
Timeout Detection	5-55
DriverTxTimeout (RX-Net)	5-55
DriverAES / DriverCallBack/TimerProcedure	5-56
Removal	5-58
DriverRemove	5-59

6 TSM Procedures

Introduction	6-1
<TSM>BuildTransmitControlBlock	6-2

<TSM>CancelPrioritySend	6-4
<TSM>GetConfigInfo	6-5
<TSM>GetNextSend	6-8
<TSM>GetASMHSMIFLevel	6-10
<TSM>GetRCB	6-11
<TSM>ProcessGetRCB	6-14
<TSM>FastProcessGetRCB	6-17
<TSM>RcvComplete	6-19
<TSM>RcvCompleteStatus	6-21
<TSM>FastRcvComplete	6-23
<TSM>FastRcvCompleteStatus	6-25
<TSM>RegisterHSM	6-27
<TSM>SendComplete	6-29
<TSM>FastSendComplete	6-31
<TSM>UpdateMulticast	6-32
RXNetTSMGetRCB	6-34
RXNetTSMRcvEvent	6-38
RXNetTSMFastRcvEvent	6-40
 7 MSM Procedures and Macros	
Introduction	7-1
Network Bus Interface	7-2
Overview	7-2
Bus Architecture	7-3
Multiple Bus Platforms	7-3
MSMAlertFatal	7-5
MSMAlertWarning	7-7
MSMAlloc	7-9
MSMAllocateMultipleRCBs	7-10
MSMAllocPages	7-12
MSMAllocaterCB	7-13
MSMCancelTimer	7-15
MSMDeRegisterResource	7-17
MSMDriverRemove	7-19
MSMEnablePolling	7-20
MSMFree	7-21
MSMFreePages	7-22
MSMGetAlignment	7-23
MSMGetBusInfo	7-25
MSMGetBusSpecificInfo	7-27
MSMGetBusType	7-31
MSMGetCardConfigInfo	7-33
MSMGetConfigInfo	7-39

MSMGetCurrentTime (macro)	7-42
MSMGetHINFromHINName	7-43
MSMGetHINNameFromHIN	7-44
MSMGetInstanceNumber	7-45
MSMGetInstanceNumberMapping	7-47
MSMGetMicroTimer	7-49
MSMGetPhysical	7-51
MSMGetPhysList	7-52
MSMGetPollSupportLevel	7-54
MSMGetProcessorSpeedRating (macro)	7-55
MSMGetUniquelIdentifier	7-56
MSMGetUniquelIdentifierParameters	7-58
MSMHardwareFailure	7-61
MSMInitAlloc	7-62
MSMInitFree	7-64
MSMNESLDeRegisterConsumer	7-65
MSMNESLDeRegisterProducer	7-66
MSMNESLProduceEvent	7-67
MSMNESLProduceMLIDEvent	7-70
MSMNESLRegisterConsumer	7-73
MSMNESLRegisterProducer	7-76
MSMParseCustomKeywords	7-79
Custom Keyword Procedure	7-80
MSMParseDriverParameters	7-85
MSMPrintString	7-91
MSMPrintStringFatal	7-93
MSMPrintStringWarning	7-95
MSMRdConfigSpace8	7-96
MSMRdConfigSpace16	7-98
MSMRdConfigSpace32	7-100
MSMReadPhysicalMemory	7-102
MSMRegisterHardwareOptions	7-104
MSMRegisterMLID	7-106
MSMRegisterResource	7-107
IOConfig Structure	7-109
MSMReRegisterHardwareOptions	7-110
MSMResetMLID	7-112
MSMResumePolling	7-113
MSMReturnDriverResources	7-114
MSMReturnMultipleRCBs	7-116
MSMReturnNotificationECB (macro)	7-117
MSMFastReturnNotificationECB (macro)	7-117
MSMReturnRCB (macro)	7-119
MSMScanBusInfo	7-120

MSMScheduleAESCAllBack	7-122
MSMScheduleIntTimeCallBack	7-124
MSMScheduleTimer	7-126
MSMSearchAdapter.	7-129
MSMServiceEvents (macro)	7-131
MSMServiceEventsAndRet (macro)	7-133
MSMSetHardwareInterrupt	7-135
MSMShutdownMLID	7-136
MSMSuspendPolling	7-138
MSMUpdateConfigTables.	7-140
MSMWritePhysicalMemory	7-142
MSMWrtConfigSpace8	7-144
MSMWrtConfigSpace16	7-146
MSMWrtConfigSpace32	7-148
MSMYieldWithDelay	7-150

Appendix A Building the HSM

Development Process.	A-1
Creating the Source Files	A-1
Assembling the Source Files	A-1
Linking the Object Files	A-2
Linker Definition File.	A-2
Loading the Driver.	A-5
Driver Configuration File.	A-6
Load Keywords and Parameters	A-6

Appendix B The NetWare Debugger

Introduction	1
Invoking the Debugger	2
Debug Commands	4
Help	4
"." Commands	4
Breakpoints	4
Breakpoint Conditions	4
B	5
BC number.	5
BCA	5
B = address [condition]	5
BW = address [condition]	5
BR = address [condition].	6
Memory	6
C address	6

C address = number(s)	6
C address = "text string"	6
D address [count].	7
M address [L length] bytewidth.	8
Register Manipulation	9
R	9
register = value	9
F flag = value	9
Input/Output	9
I[B,W,D] port	9
O[B,W,D] port = value	10
Miscellaneous	10
G [address(es)]	10
N symbolname value	10
P	11
Q	11
T or S	11
U address [count].	11
V	11
Z <i>expression</i>	11
Debug Expressions	12
Grouping Operators	13
Conditional Evaluation	13
Symbolic Information.	14

Appendix C NESL Support

Overview	C-1
Registering and Deregistering Event Producers	C-2
Registering and Deregistering Event Consumers	C-2
NESL Structures	C-3
EPB (Event Parameter Block) Structure	C-3
NESL_ECB Structure	C-4
Events and Types	C-7
Event Names	C-7
Event Types	C-8
Service Suspend Types	C-8
.	C-9
Suspend Request.	C-9
Service Resumed Types	C-10
Service/Status Changed Types.	C-11
NESL Return Codes	C-13
NESL Event Flags	C-14
NESL OSI Layer Definitions	C-15

Revision History

Index

x ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Figures

Figure 1-1 The Open Data-Link Interface Model	1-1
Figure 1-2 Loadable Modules as NetWare Building Blocks.	1-3
Figure 1-3 MLID Modules	1-6
Figure 1-4 The ODI Model with separate MSM, TSM and HSM Modules	1-8
Figure 2-1 Implementation of Multiple Frame/Multiple Adapter Support	2-11
Figure 3-1 Global Data Access.	3-2
Figure 3-2 Frame and Adapter Data Space.	3-14
Figure 3-3 Frame and Adapter Data Space.	3-34
Figure 4-1 Packet Transfer in the MSM/ODI Model.	4-10
Figure 4-2 Fragmented Receive Control Block.	4-12
Figure 4-3 Non-Fragmented Receive Control Block	4-14
Figure 4-4 Packet Transfer in the MSM/ODI Model.	4-16
Figure 4-5 Ethernet, Token-Ring and FDDI Transmit Control Block	4-17
Figure 4-6 Rx-NET Transmit Control Block	4-19
Figure 4-7 TCB Fragment Structure	4-22
Figure 4-8 Packet Transfer in the MSM/ODI Model.	4-24
Figure 4-9 Event Control Block	4-25
Figure 4-10 ECBs vs RCBs	4-26
Figure 4-11 Transmit ECBs vs TCBs	4-28
Figure 5-1 Format of RX-Net LookAhead Buffer	5-18
Figure 6-1 Format of the RX-Net LookAhead Buffer	6-36
Figure 7-1 Multiple Bus Platform Example	7-3
Figure 7-2 PnP ISA Bus Parameters	7-60

xii ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Tables

Table 3.1	Driver Parameter Block Field Descriptions	3-6
Table 3.2	Configuration Table Field Descriptions	3-17
Table 3.3	MLIDModeFlag Descriptions	3-26
Table 3.4	MLIDSFlags Bit Map Fields	3-28
Table 3.5	MLIDSharingFlags Bit Map	3-30
Table 3.6	Media Specific Counters for Token Ring	3-42
Table 3.7	Media Specific Counters for Ethernet	3-44
Table 3.8	Media Specific Counters for FDDI	3-46
Table 3.9	Media Specific Counters for RX-Net	3-48
Table 4.1	Fragmented RCB Field Descriptions	4-13
Table 4.2	Non-Fragmented RCB Field Descriptions	4-15
Table 4.3	TCB Field Descriptions	4-18
Table 4.4	TCB Field Descriptions (RX-Net)	4-20
Table 4.5	TCB Fragment Structure	4-23
Table 4.6	ECB Field Descriptions	4-29
Table 6.1	TSMCFG_SystemFlags	6-7

xiv ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Preface

This Intel assembly language document provides the information necessary to develop the Hardware Specific Module (HSM) portion of a Novell 32-bit LAN Driver.

Novell LAN Drivers consist of Media Support Modules (MSMs), Topology Specific Modules (TSMs), and Hardware Specific Modules (HSMs).

The Novell LAN Driver, Software Development Toolkit (SDK) provides the MSMs and the TSMs for the LAN Driver. The HSMs must be written by the developer.

LAN Drivers written using the information in this document will conform to the Novell Open Data-Link Interface (ODI) specification. The MSMs and TSMs provided by the SDK make it as simple as possible to do this.

This document does not explain the full ODI specification. It only explains how to write the HSM portion of a LAN Driver, using the modules provided by the Novell SDK.

Document Overview

Chapter 1 describes the NetWare environment and gives a brief overview of ODI LAN Driver architecture.

Chapter 2 is an overview of the HSM.

Chapter 3 describes HSM data structures and variables.

Chapter 4 describes MSM and TSM data structures and variables.

Chapter 5 describes the HSM procedures a developer must provide.

Chapter 6 describes the TSM procedures provided by the Novell SDK.

Chapter 7 describes the MSM procedures provided by the Novell SDK.

Appendix A describes assembling, linking, and loading an ODI LAN driver.

Appendix B explains how to use the NetWare integrated debugger.

Appendix C explains how to update an HSM to the current specification.

ODI Supplements

The following supplements also contain information necessary for driver development.

The MLID Installation Information File

Part number 107-000056-001

The Hub Management Interface

Part number 107-000023-001

Source Routing

Part number 107-000058-001

Canonical and Noncanonical Addressing

Part number 107-000059-001

Frame Types and Protocol IDs

Part number 107-000055-001

Standard MLID Message Definitions

Part number 107-000060-001

Router Support

Part number 107-000049-001

Prerequisites

Developers using this document must be experienced in the following areas:

- Intel Assembly Language Programming
- Intel 80386/486+ Microprocessors
- Real Mode and Protected Mode
- Re-entrant Coding
- Event-driven Systems
- Interrupt-driven Device Drivers

Document Conventions

This document uses the following conventions:

- All numbers in this document are decimal unless otherwise specified.
- Hexadecimal numbers are identified by a trailing 'h', such as:
FFh.
- In bit fields, bit 0 is the low- order bit.
- The following data types are defined:

byte	1 byte unsigned integer
char	1 byte ASCII character
offset	32-bit non-segmented address

Note



Numeric fields composed of more than 1 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 of the field, the next most significant byte in the second byte, and so on, with the least significant byte appearing last. Low-high numbers are stored in exactly the opposite order. Intel microprocessors store numbers in low-high order.

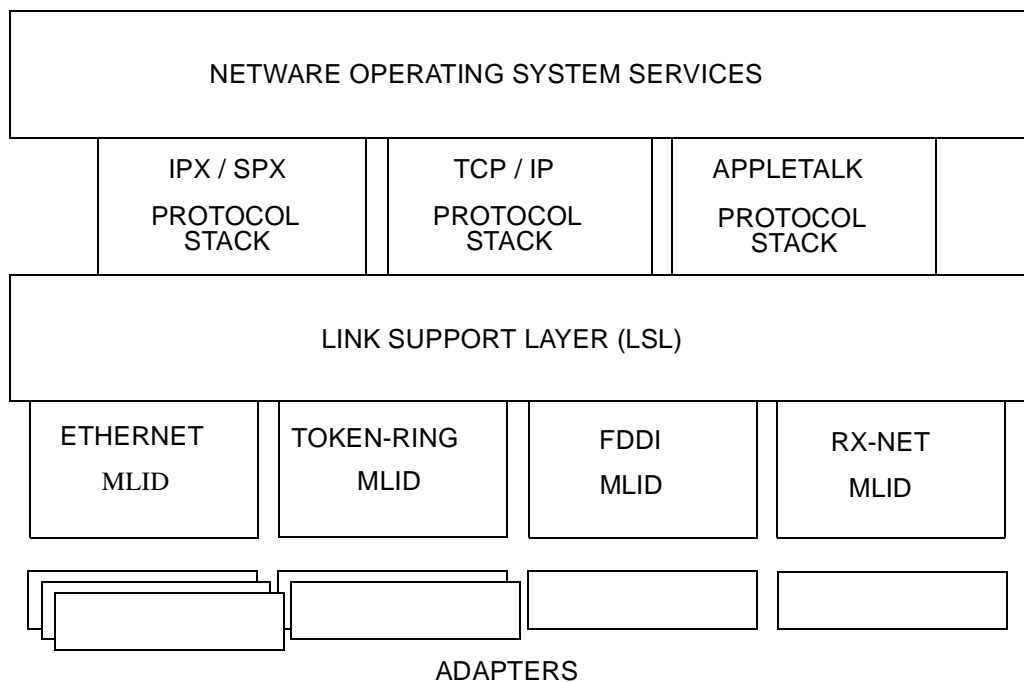
xx ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

1 Introduction

Open Data-Link Interface

Novell's Open Data-Link Interface (ODI) technology was developed to allow multiple topologies, frame types and protocols to coexist on network systems. The ODI specification describes the set of interface and software modules used to decouple device drivers from protocol stacks and to enable multiple protocol stacks to share the network hardware and media transparently. Figure 1.1 illustrates the components of the ODI model.

Figure 1-1 The Open Data-Link Interface Model



Link Support Layer

At the core of the Open Data-Link Interface is the Link Support Layer or LSL. The LSL is the interface between drivers and protocol stacks. It essentially acts like a switchboard, directing packets between the appropriate drivers and protocol stacks. Any LAN driver written to the ODI specifications, can communicate with any ODI protocol stack via the Link Support Layer.

Multiple Link Interface Drivers

Multiple Link Interface Drivers (MLIDs) are LAN Drivers written to the ODI specification. Each driver is unique due to the adapter hardware and media, but ODI eliminates the need for separate drivers to be written for each specific protocol stack. The Open Data-Link Interface allows LAN drivers to function with protocol stacks independent of the frame type and protocol stack details.

1-2 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

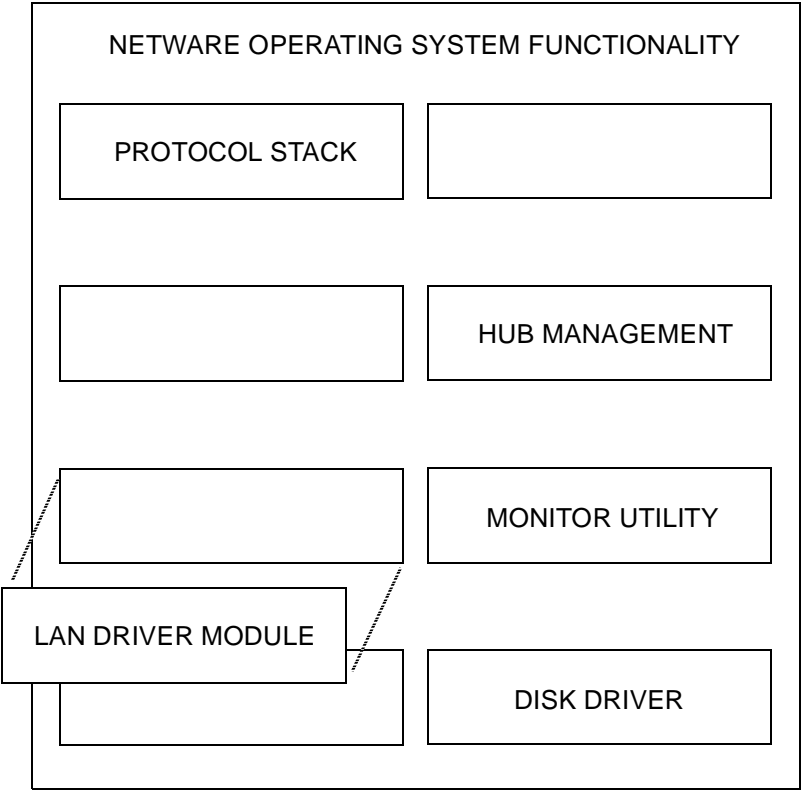
NetWare Loadable Modules

A key NetWare feature is the NetWare Loadable Module (NLM). 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 operating system as shown in Figure 1.2.

Different types of loadable modules have unique filename extensions that signify the module's function. Server LAN drivers must use the “.LAN” filename extension, disk drivers use “.DSK”, and general utility or support modules use “.NLM”.

The modules that make up a Multiple Link Interface Driver are NetWare Loadable Modules.

Figure 1-2 Loadable Modules as NetWare Building Blocks



Driver Modules

This section describes the modules that make up a 32-bit Multiple Link Interface Driver.

Novell Provided Support Modules

Novell has simplified ODI LAN driver development by furnishing a set of support modules that provides the interface to the LSL. These modules are a collection of procedures, macros, structures, and variables. They are the Media Support Module (MSM), which contains general functions common to all drivers; and the Topology Specific Modules (TSM), which provide support for the standardized media types of Ethernet, Token-Ring, RX-Net, and FDDI.

Media Support Module

The Media Support Module, MSM.NLM, standardizes and manages the primary details of interfacing ODI Multi-Link Interface Drivers to the LSL and OS. The MSM handles all of the generic initialization and run-time issues common to all drivers.

Topology Specific Module

The Topology Specific Module, <TSM>.NLM, manages operations that are unique to a specific media type. Multiple frame support is implemented in the TSM so that all frame types for a given media are supported.

Throughout this manual, topology specific functions and variables are indicated with <TSM>. The developer must replace <TSM> with the appropriate media type depending on which module is used. Since the driver must be assembled with case sensitivity on, the names must be used exactly as shown below:

ETHERTSM.NLM	replace <TSM> with:	EtherTSM
TOKENTSM.NLM	replace <TSM> with:	TokenTSM
RXNETTSM.NLM	replace <TSM> with:	RXNetTSM
FDDITSM.NLM	replace <TSM> with:	FDDITSM

Source code for each Topology Specific Module is provided with the developers kit. Proprietary topology modules may be created by modifying an existing TSM to meet the developer's requirements or by creating a new module that provides the same functionality contained in the standard TSMs.



If a topology specific module is altered, it must NOT have the same name as the Novell provided modules. In addition, any "exported" calls or variables within the TSM require different names.



You must use the TSMs provided by Novell to pass certification. It is virtually impossible for your driver to pass the certification test suite using modified TSMs.

Developer Provided Module

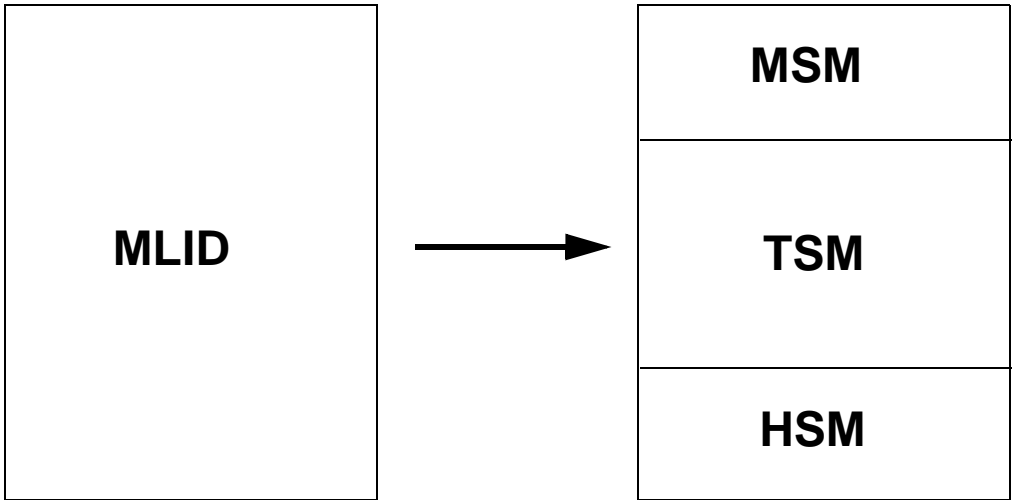
When using the support modules, LAN driver development is reduced to creating the Hardware Specific Module or HSM, to handle all hardware interactions.

Hardware Specific Module

The Hardware Specific Module, <HSM>.LAN, is created by the developer for a specific physical card. Its primary functions include: adapter initialization, reset, shutdown, and removal, as well as packet reception and transmission. Additional procedures may also provide support for timeout detection, multicast addressing, and promiscuous mode reception. Sample source code for Novell LAN drivers is included with the *Novell LAN Driver Developer's Guide*. Chapter 2 explains the HSM functions in greater detail.

Figure 1.3 illustrates the modules which make up an MLID.

Figure 1-3 MLID Modules



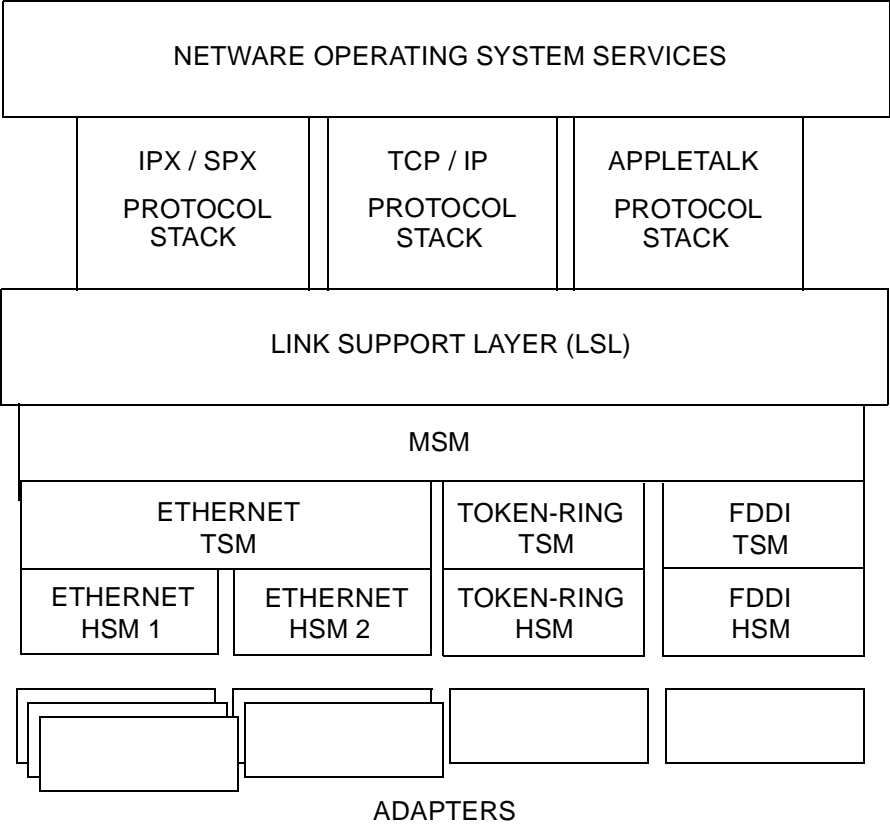
Loading Driver Modules

The ODI Toolkit components for the specific platform being used must be loaded before the HSM is loaded. The HSM linker definition file must list a dependency on the appropriate TSM, using the module keyword, for the required NLMs to load automatically.

A major advantage of separating portions of the MLID into the MSM and TSM is that, once loaded, these support modules become available to all HSMs. Only a single code image of each module is needed to support multiple HSMs.

Figure 1.4 below illustrates the ODI model with separate MSM, TSM, and HSM modules.

Figure 1-4 The ODI Model with separate MSM, TSM and HSM Modules



Development Process

The process of creating and loading a NetWare driver involves the following steps:

1. Create the driver 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 OS.
5. Debug the driver.

Chapters 2 through 7 provide detailed information on writing the driver. Appendices A, B and the supplements listed below provide a full description of assembling, linking, installing, loading, and debugging the driver.



Note LAN Drivers written to this specification will also function without modification on Novell 32-bit clients for WIN95, WindowsNT, and DOS/WIN.

ODI Supplements

The ODI supplements listed in the preface of this document also contain information necessary for driver development.

Driver Related Files

The following section describes the files that are needed when developing a NetWare LAN driver.

Source Files

The Hardware Specific Module is the only source file that must be written by the developer. Chapter 2 provides an overview of the Hardware Specific Module and addresses specific hardware and coding issues that influence driver development.

Include Files

Several include files are provided with the support modules. These files contain external variable declarations and define the equates, macros, and data structures needed by the HSM.

The HSM must include only DRIVER.INC; the other include files are nested from this file.

```
DRIVER . INC
MSM . INC
ODI . INC
```

Linker Definition File

Each NetWare Loadable Module must have a corresponding linker definition file with a “.DEF” extension. This file is needed by the NetWare linker. It contains a list of object files which makeup the module, external variables and routines the module must access, the names of the module's initialization and exit procedures, and several other linker directives. (see Appendix A for details)

Driver Configuration File

The developer can list command-line parameters and custom keywords in a driver configuration file. If used, this file must reside in the same directory as the driver. The driver configuration file was developed to allow drivers to maintain large number of custom keywords on the limited space of the command-line. (see Appendix A for details)

Installation Information File

We require you to create a driver information file to simplify driver installation. This file provides information related to the driver configuration and loading parameters and is required if the Install utility is used. (See *The MLID Installation Information File* supplement for details.)

1-10 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

chapter **2** *HSM Overview*

HSM Components

This chapter provides an overview of the HSM components. Issues that influence the development of the HSM are also addressed.

HSM Procedures

The HSM specification defines the following procedures:

Initialization and Removal

- DriverInit (required)
- DriverRemove (required)

Reception and Transmission

- DriverISR
- DriverPoll
(one of the above is required)
- DriverSend (required)
- DriverISR2 (optional)

Multi-Operating System provision

- DriverEnableInterrupt (required)
- DriverDisableInterrupt (required)
- DriverDisableInterrupt2 (required if Driver ISR2 exists)

I/O Control

- DriverReset (required)
- DriverShutdown (required)
- DriverMulticastChange (required except for RX-Net)
- DriverPromiscuousChange (recommended)
- DriverStatisticsChange (optional)
- DriverRxLookAheadChange (optional)
- DriverManagement (optional)

Timeout Detection

- DriverAESCAllBack (optional)
- DriverINTCallBack (optional)
- DriverTimerProcedure (optional, see **MSMScheduleTimer**)

Every HSM must provide the required procedures in order to function properly. The recommended procedures must be implemented if the hardware supports that function. The optional procedures are available if the adapter or driver requires the functionality. You may add any procedures necessary to support the specific hardware features of your particular LAN adapter design. Adjustments to the HSM will also be required if hub management is supported, or Brouter enhancements are included.

Brief descriptions of these HSM procedures are provided on the following pages. The descriptions are general and do not apply in every case, nor do they describe every possible case. Detailed descriptions of the procedures (including pseudocode) are provided in Chapter 5

2-2 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Initialization and Removal

The HSM's initialization routine, **DriverInit**, is called by the NetWare operating system to initialize the adapter hardware. The **DriverInit** routine uses MSM/TSM calls to perform the following tasks:

- Allocate memory for driver variables and structures
- Parse the standard LOAD command-line options
- Process custom command-line parameters and custom firmware
- Register the driver with the LSL
- Register the hardware configuration with the OS
- Setup for the board's ISR or polling procedure
- Schedule callback events for timeout detection and recovery
- Handle any initialization errors

The HSM's remove procedure, **DriverRemove**, allows the network supervisor to unload it from the operating system. This procedure must shutdown the physical board and return all resources allocated to the driver. The MSM provides routines that handle the return of driver resources.

Packet Reception and Transmission

The HSM's board service routine will generally need to detect and handle the events listed below. The driver can be notified of these events by using either an interrupt service routine, **DriverISR**, or a polling procedure, **DriverPoll**, or a combination of both.

- Packet Reception and/or Reception Complete
- Reception Error
- Transmission Complete
- Transmission Error

The HSM's **DriverSend** procedure is called whenever a packet needs to be transmitted onto the wire. Prior to calling this procedure, the TSM builds the appropriate frame and media headers for the packet. The driver simply collects the header and packet data fragments, then initiates the transmission.

Multi-Operating System Provisions

Novell requires implementing the **DriverEnableInterrupt** and **DriverDisableInterrupt** procedures. These procedures allow for the transporting of drivers to other 32-bit Intel-based OS platforms where access to the PIC is restricted.

The ability of an adapter to disable its interrupt capability in the hardware, not by masking the PIC, and not disabling interrupts at the CPU, is essential to run under multiprocessor operating systems such as Windows NT.



Multi-operating system support is required for certification. You will find that passing Novell Labs' latest test suite for certification will go much smoother and you will save considerable time if you adhere strictly to the specification. Refer to Appendix A, "Building the HSM" when writing or updating a driver.

I/O Control Procedures

The HSM must provide the control procedures **DriverReset** and **DriverShutdown**, to handle the hardware operations involved in resetting or shutting down the adapter. Additional control procedures may also be needed to support multicast addressing and promiscuous mode reception. These routines are **DriverMulticastChange**, and **DriverPromiscuousChange**.

The **DriverStatisticsChange** and **DriverRxLookAheadChange** procedures are optional. These procedures allow drivers for intelligent adapters to update the statistics table or the LookAhead size only as needed.

Drivers that are Brouter enhanced, or that support the Hub Management Interface must implement the **DriverManagement** procedure to handle management requests and commands.

Timeout Detection

The HSM can still schedule timers used to repeatedly callback the **DriverAESCallback** or the **DriverINTCallback** routines at a developer-specified interval.

For example, the driver may need to be called regularly so that it can inspect the adapter to determine if it 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.

With this specification, the HSM can call **MSMScheduleTimer** to setup timer callbacks. This method is now preferred over previous methods.

HSM Data Structures and Variables

In addition to the procedures described in the previous section, the Hardware Specific Module must also contain certain data structures and variables. The primary structures include the Driver Parameter Block, the Driver Configuration Table, and the Driver Statistics Table.

Chapter 3 provides detailed descriptions of all the required HSM data structures and variables.

HSM Design Considerations

The following section discusses the hardware and coding issues that must be considered when creating the HSM.

Hardware Issues

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

For example, HSMs that support adapters on buses with hot plug/unplug capability (such as PC Card and PCI) must be written so that the HSM does not attempt to access hardware that is not currently present in the system, or, at least, does not do inappropriate things as a result of accessing hardware that is not currently present in the system.

Network Interface Controllers

The LAN driver developer must be familiar with the Network Interface Controller IC. Every effort should be made to obtain and use current data books and application notes from the manufacturer. In addition, the manufacturer's support engineers can provide developers with up-to-date information on hardware quirks and changes.

Data Transfer Mode

The MSM and TSM provide certain support procedures that are optimized for use with a specific data transfer mode. The development of the HSM's packet reception and transmission routines in particular will be affected by the adapter's transfer mode. In order to achieve the highest performance, the developer 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

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

- Industry Standard Architecture (ISA) and PnPISA
- Micro Channel Architecture
- Extended Industry Standard Architecture (EISA)
- PC Card (PCMCIA)
- Peripheral Component Interconnect (PCI)

Coding Issues

NetWare LAN drivers operate as an integral part of the NetWare operating system. Therefore, the developer must consider the following operating system characteristics when writing the HSM code.

Multi-Tasking, Non-Preemptive OS

The NetWare operating system is multi-tasking and non-preemptive. Non-preemptive means the OS will not interrupt one process so that another process can execute. Therefore, HSM routines must not dominate system resources. If the code is optimized, normal execution will not be a problem. Care must be taken when handling operations such as retry loops and board error conditions so that other processes can execute in a timely manner.

32-Bit Protected Mode

NetWare runs in 32-bit protected mode. In addition, the operating system accesses a flat code space where CS=SS=ES=DS. Consequently, all of the support routines available in the MSM and TSM modules are near calls for the HSM.

An assembler that supports the use of 32-bit registers is required to build the HSM. Novell engineers currently use the Phar Lap 386ASMP protected mode assembler (v4.0 or later).

Interrupt Service Routine

When **DriverISR** is called (the system ISR actually receives the interrupt), the direction flag is cleared, interrupts are disabled, and all registers are pushed on the stack. The driver only needs to service the interrupt and return (do not use `iret`). If the driver sets the direction flag during the routine, it must clear it before returning.



Novell requires that interrupts remain disabled during **DriverISR** and **DriverSend**. If either routine must enable interrupts, it must disable them before returning.

2-8 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

OS Calls to the Driver

Portions of the NetWare operating system are written in the C programming language. Any HSM routines that are called from a C routine must preserve the EBX, EBP, ESI, and EDI registers. The HSM routines which this affects are **DriverInit** and **DriverRemove**.

Execution Times

Drivers can perform certain operations only at certain execution times.

The two principal execution times are:

- Process Time
- Interrupt Time

As you write your driver, you must be aware of which routines are called at process time and which routines are called at interrupt time.

The table below shows when each driver routine can be called by the operating system or support module.

Execution Time of Driver Routines

Process Time	Interrupt Time
DriverAESCAllBack	DriverINTCallBack
DriverInit	DriverISR
DriverManagement	DriverTimerProcedure (setup with MSMScheduleTimer)
DriverMulticastChange	
DriverPoll	
DriverPriorityQueueSupport	
DriverPromiscuousChange	
DriverRemove	
DriverReset	
DriverRxLookAheadChange	
DriverSend	
DriverShutdown	
DriverStatisticsChange	

The execution time restrictions for TSM and MSM support routines are documented in Chapters 6 and 7.

Process Time

At process time the MLID is allowed to:

- Allocate memory
- Do file I/O tasks (with some exceptions)

There are two types of process time routines:

- Routines that suspend execution to allow other processes to execute
- Routines that do not suspend execution

Interrupt Time

When the operating system's interrupt handler calls a routine, that routine operates at interrupt time.

At interrupt time, routines must not do the following:

- Allocate memory
- Do file I/O tasks
- Suspend execution
- Call another routine that suspends execution

Interrupt time routines must be highly optimized and limit their execution time.

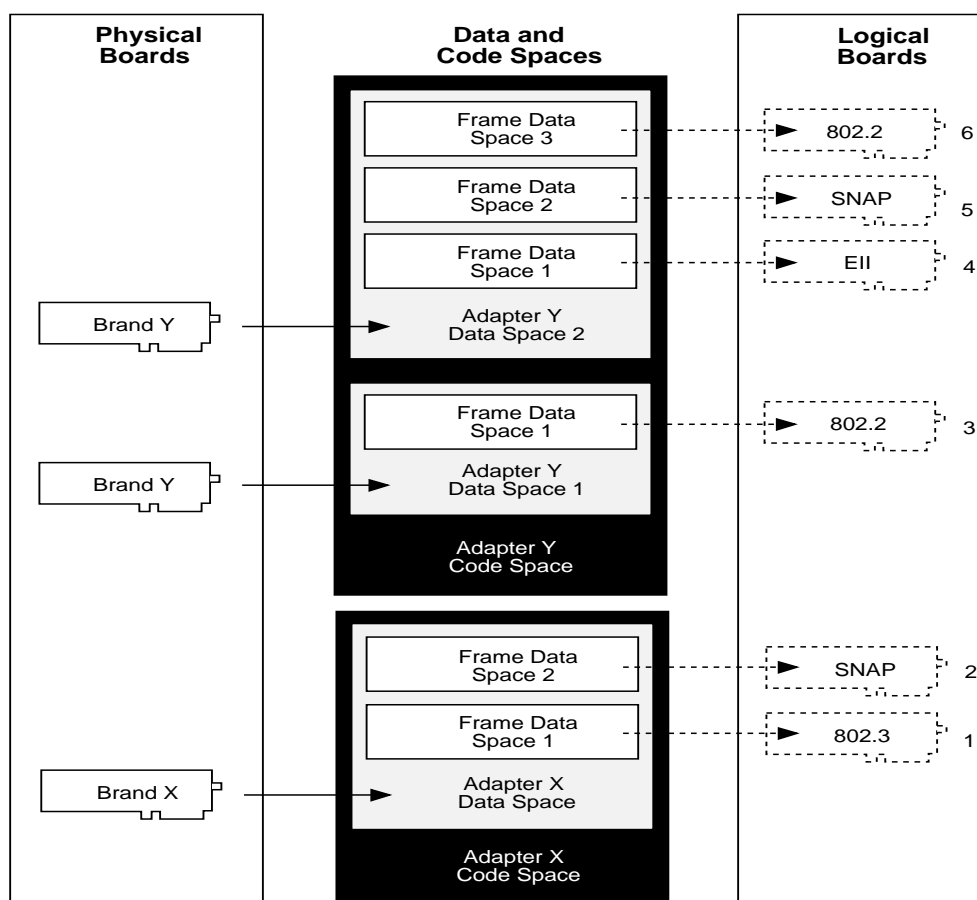


When a driver routine calls another driver routine you must be aware of the execution time restrictions for both calls. For example, **DriverISR** typically calls **DriverSend** to transmit the next packet in the send queue after a transmit complete interrupt. Since **DriverISR** executes at interrupt time, **DriverSend** will also execute at interrupt time and must observe the same interrupt time restrictions. The same applies to **DriverReset**.

Code and Data Space

This section describes the organization of the code and data space for Novell 32-bit LAN drivers. Figure 2.1 illustrates the code and data space used for multiple adapters with multiple frame support. The Frame Data Space, which represents a “Logical Board”, is created by the MSM for each loaded frame type. The Adapter Data Space is created by the MSM for each physical board. Since HSMs are reentrant, all physical boards of the same type use a single Adapter Code Space. (See the Reentrancy section later in this chapter.)

Figure 2-1 Implementation of Multiple Frame/Multiple Adapter Support



Frame Data Space

The Frame Data Space contains all the information needed to support a specific frame type as well as the hardware configuration of the corresponding board. For each loaded frame type, there will be a separate Frame Data Space allocated representing a Logical Board. (see Chapter 3 for details on the Frame Data Space)

Note



Novell requires drivers to support all frame types for a particular topology. Because all TSMs provide full multiple frame support, drivers developed with these modules are guaranteed to support all applicable frame types for the topology.

Adapter Data Space

The Adapter Data Space contains certain hardware and statistical information needed to drive or manage a particular physical board. There is only one Adapter Data Space allocated for each physical board, regardless of the number of frame types supported by that board. (See Chapter 3 for details on the Adapter Data Space.)

Adapter Code Space

When multiple frame types are loaded for an adapter and/or when multiple adapters of the same type are loaded, a single code image of the driver is used for all logical boards associated with those adapters.

Reentrancy

Reentrancy, in this case, means that the driver code must be written to work with multiple logical boards and/or with multiple adapters of the same type.

The MSM and TSM will pass pointers to the appropriate Frame Data Space and Adapter Data Space when calling a driver routine. References to structures and variables must be performed using pointers and offsets rather than hard-coded values.

The HSM linker definition file must include the keyword, “reentrant”. This keyword allows a driver to be loaded more than once to support multiple frame types or multiple boards of the same type. However, only a single code image of the driver is loaded.

2-12 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Recommended Support

Multicast Addressing

Multicast addressing **must** be supported if the media supports it (Ethernet, Token-Ring and FDDI). If your adapter hardware cannot support it, but the media does, the adapter cannot be certified. Refer to Chapter 5 in the **DriverMulticastChange** section.

Promiscuous Mode

Drivers that pass all packets being received by the adapter are said to have a promiscuous reception mode. Hub management, and other monitoring functions, would use this mode. Novell strongly recommends the HSM support promiscuous mode if the adapter is capable of supporting it. The HSM must enable or disable promiscuous reception on request as described in the **DriverPromiscuousChange** section of Chapter 5.

Optional Support

Hub Management

The Simple Network Management Protocol (SNMP) and the HUBCON utility can manage 10BaseT repeaters and Token-Ring concentrator hubs attached to or integrated into the server. *The Hub Management Interface* supplement describes how to support management requests from these two agents in the HSM.

Source Routing

A Novell 32-bit LAN driver may include the capability to pass packets across an IBM bridge. To do this, source routing information must be added to the packet's MAC header. The Novell provided ROUTE.NLM and TSM modules handle this procedure with no interaction from the HSM. The *Source Routing* supplement describes the functions of the source routing module.

Router

A Token-Ring adapter/driver may be capable of source route bridging support. This is a mechanism that allows the source of traffic to dynamically discover routes and determine which one to use when sending data to a particular destination. With the Source Route Bridge NLM loaded, a server can also function as a router or bridge. For HSM support requirements see the *Router Support* supplement.

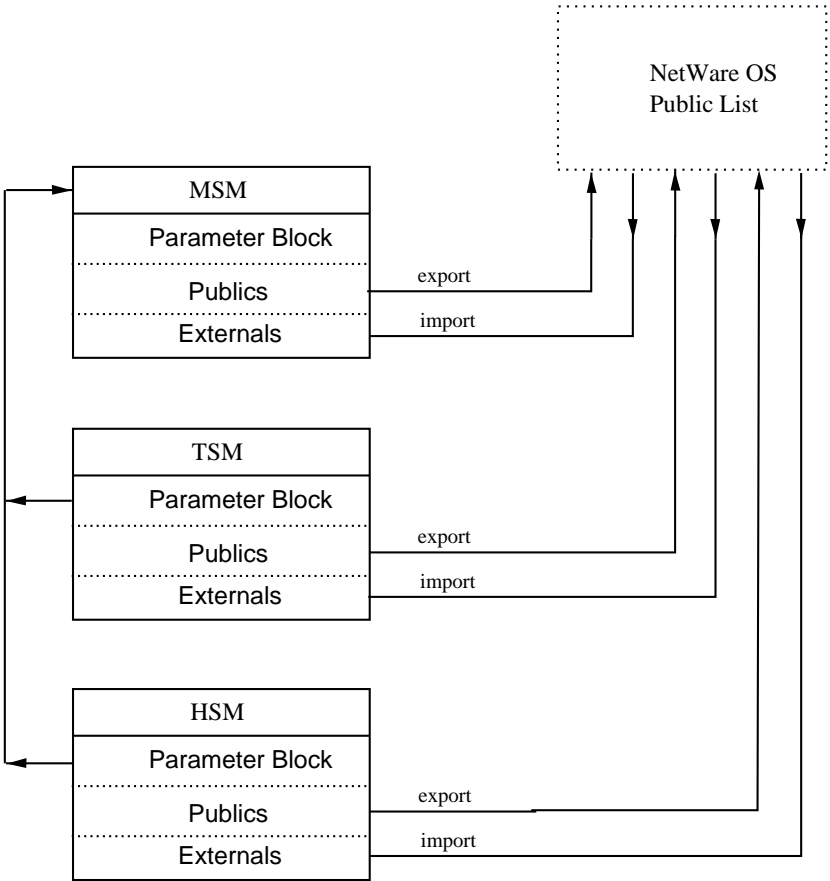
Introduction

This chapter describes the data structures and variables that the Hardware Specific Module must define when a driver is written using the support modules. All the data structures and variables listed in this chapter must be present in the OSDATA segment in order for the HSM to function properly.

Global Data Access

When the MSM and the TSM are loaded, all public variables and procedures are exported to the operating system and are available to any NLMs subsequently loaded as shown in Figure 3.1. The HSM can gain access to them by declaring them *extern* and by including them in the import list in the Linker Definition File (see Appendix A). This keyword tells the linker which external variables and procedures the HSM must access.

Figure 3-1 Global Data Access



The modules that make up an MLID are designed to be loaded in the following order:

1. MSM .NLM
2. <TSM>.NLM
3. <HSM>.LAN

3-2 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Specification Version

So that test programs can verify the ODI specification version that an HSM is written to, developers must put the ODI specification version number in the OSDATA segment, as follows:

```
HSMSPEC      db      'HSM_ASPEC_VERSION: 3.31',0
```

Note



One space is required between the colon and the first digit.

Driver Parameter Block

Because it is loaded last, the HSM must make its public variables and procedures available to the support modules using a structure called the `DriverParameterBlock`.

The `DriverParameterBlock` structure contains the required HSM public variables, as well as pointers to the driver's tables, structures, and procedures. The fields of the `DriverParameterBlock` are accessed by external procedures using offsets, therefore its format is strictly defined. (See the `DriverParameterBlock` illustration below.)

In order for external procedures to gain access to the Parameter Block, the HSM's **DriverInit** routine passes a pointer to the block in ESI when it calls `<TSM>RegisterHSM`.

Driver Parameter Block

<u>DriverParameterBlock</u>	<u>Label</u>	<u>Dword</u>
DriverParameterSize	dd	DriverParameterBlockSize
DriverStackPointer	dd	0
DriverModuleHandle	dd	0
DriverBoardPointer	dd	0
DriverAdapterPointer	dd	0
DriverConfigTemplatePtr	dd	DriverConfigTemplate
DriverFirmwareSize	dd	0
DriverFirmwareBuffer	dd	0
DriverNumKeywords	dd	0
DriverKeywordText	dd	0
DriverKeywordTextLen	dd	0
DriverProcessKeywordTab	dd	0
DriverAdapterDataSpaceSize	dd	SIZE DriverAdapterDataSpace
DriverAdapterDataSpacePtr	dd	DriverAdapterDataSpaceTemplate
DriverStatisticsTablePtr	dd	DriverStatisticsTable
DriverEndofChainFlag	dd	0
DriverSendWantsECBs	dd	0
DriverMaxMulticast	dd	20
DriverNeedsBelow16Meg	dd	0
DriverAESPtr	dd	0
DriverCallBackPtr	dd	offset DriverCallBack
DriverISRPtr	dd	offset DriverISR
DriverMulticastChangePtr	dd	offset DriverMulticastChange
DriverPollPtr	dd	0
DriverresetPtr	dd	offset DriverReset
DriverSendPtr	dd	offset DriverSend
DriverShutdownPtr	dd	offset DriverShutdown
DriverTxTimeoutPtr	dd	0
DriverPromiscuousChangePtr	dd	offset DriverPromiscuousChange
DriverStatisticsChangePtr	dd	0
DriverRxLookAheadChangePtr	dd	0
DriverManagementPtr	dd	0
DriveEnableInterruptPtr	dd	offset DriveEnableInterrupt
DriverDisableInterruptPtr	dd	offset DriverDisableInterrupt
DriverISR2Ptr	dd	offset DriverISR2
DriverReserved1	dd	0
HSMSpecVerString	dd	0
DriverPriorityQueuePtr	dd	0
DriverDisableInterrupt2Ptr	dd	offset DriverDisableInterrupt2
DriverParameterBlockSize	equ	\$ -
DriverParameterBlock		

HSM Data Structures and Variables 3-5

Table 3.1 Driver Parameter Block Field Descriptions

Offset	Name	Bytes	Description
00h	DriverParameterSize	4	Set this variable to the size of the defined DriverParameterBlock structure before calling <TSM>RegisterHSM . Since the block format is strictly defined and its size must remain constant, the MSM uses this field to screen for invalid parameter blocks. <TSM>RegisterHSM will fail if this value is incorrect.
04h	DriverStackPointer	4	When the operating system calls the developer's DriverInit routine, it passes certain information on the stack needed by the MSM. DriverInit must set this variable to the value of the stack pointer (ESP) after it pushes the C registers (EBP, EBX, ESI, EDI). During <TSM>RegisterHSM , the MSM uses this value to locate the parameters on the stack.
08h	DriverModuleHandle	4	The MSM sets this value when the developer's DriverInit routine calls <TSM>RegisterHSM . This handle is used to identify the Network Loadable Module and is used by the operating system support routines to access and manage information about the NLM. The HSM's DriverRemove routine needs this value when it calls MSMDriverRemove .
0Ch	DriverBoardPointer	4	The MSM sets this value when the developer's DriverInit routine calls <TSM>RegisterHSM . This field is reserved for use by the MSM.
10h	DriverAdapterPointer	4	The MSM sets this value when the developer's DriverInit routine calls MSMRegisterHardwareOptions . This field is reserved for use by the MSM.
14h	DriverConfigTemplatePtr	4	Set this variable to point to the driver's configuration table template before calling <TSM>RegisterHSM . The configuration table is described later in this chapter.

3-6 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Table 3.1 Driver Parameter Block Field Descriptions *continued*

Offset	Name	Bytes	Description
18h	DriverFirmwareSize	4	(see the “Driver Firmware” section later in this chapter)
1Ch	DriverFirmwareBuffer	4	(see the “Driver Firmware” section later in this chapter)
20h	DriverNumKeywords	4	(see the “Driver Keywords” section later in this chapter)
24h	DriverKeywordText	4	(see the “Driver Keywords” section later in this chapter)
28h	DriverKeywordTextLen	4	(see the “Driver Keywords” section later in this chapter)
2Ch	DriverProcessKeywordTab	4	(see “Driver Keywords” section later in this chapter)
30h	DriverAdapterDataSpaceSize	4	Set this field to the size of the <i>DriverAdapterDataSpace</i> template (described later in this chapter) before calling <TSM>RegisterHSM .
34h	DriverAdapterDataSpacePtr	4	Set this field to point to the <i>DriverAdapterDataSpace</i> template (described later in this chapter) before calling <TSM>RegisterHSM .
38h	DriverStatisticsTablePtr	4	Set this variable to the offset of the <i>DriverStatisticsTable</i> from the top of the <i>DriverAdapterDataSpace</i> template before calling <TSM>RegisterHSM . The statistics table and template are described later in this chapter.
3Ch	DriverEndOfChainFlag	4	Before calling MSMRegisterHardwareOptions , set this field to a nonzero value if the driver supports shared interrupts and wants to be placed at the end of the chain. This field is used only if bit 5 is set in the <i>MLIDSharingFlags</i> field of the configuration table.

HSM Data Structures and Variables **3-7**

Table 3.1 Driver Parameter Block Field Descriptions *continued*

Offset	Name	Bytes	Description
40h	DriverSendWantsECBs	4	Before calling MSMRegisterHardwareOptions , set this field to any nonzero value if the DriverSend routine needs ECBs rather than TCBs. This should be used by intelligent bus master adapters that are designed to be ECB aware. (see Chapter 4)
44h	DriverMaxMulticast	4	Before calling MSMRegisterHardwareOptions , set this field to the maximum number of multicast addresses that the adapter can handle. EtherTSM, TokenTSM, and FDDITSM can accommodate an almost unlimited number of multicast addresses (limited only by server memory). If an HSM can handle unlimited multicast addresses, set to -1. (See also bits 9 and 10 definitions in the configuration table <i>MLIDFlags</i> field later in this chapter.)

Table 3.1 Driver Parameter Block Field Descriptions *continued*

Offset	Name	Bytes	Description
48h	DriverNeedsBelow16Meg	4	<p>Before calling <TSM>RegisterHSM, drivers for Bus Master or DMA adapters can set this field to any nonzero value if the adapter can only communicate with host memory below 16 megabytes. This will inform the MSM to only allocate buffers, RCBs, TCBs, and ECBs below the 16 megabyte boundary if the system already has more than 16 megabytes at the time the driver loads.</p> <p>If the driver is loaded on a system that initially has less than 16 megabytes of memory but will have more memory added later using the server's REGISTER MEMORY command, you must use the BELOW16 keyword (see Appendix A) on the load command line to force the MSM to allocate memory below 16 megabytes.</p> <p>If the driver preallocates more than 8 RCBs, the number of RCBs below 16 megabytes can be adjusted above the default of 8, by using the BUFFERS16 keyword. The MSM allocates these at initialization (see Appendix A).</p>
4Ch	DriverAESPTr	4	Set this field to point to the HSM's DriverAES routine before calling MSMScheduleAESCallback . (If AES callback events are not used, set this field to zero.)
50h	DriverCallbackPtr	4	Set this field to point to the HSM's DriverCallback routine before calling MSMScheduleIntTimeCallback . (If interrupt level callback events are not used, set this field to zero.)
54h	DriverISRPtr	4	Set this field to point to the HSM's DriverISR routine before calling MSMSetHardwareInterrupt . (If DriverPoll is used instead, set this field to zero).

HSM Data Structures and Variables **3-9**

Table 3.1 Driver Parameter Block Field Descriptions *continued*

Offset	Name	Bytes	Description
58h	DriverMulticastChangePtr	4	Set this field to point to the HSM's DriverMulticastChange routine before calling MSMRegisterHardwareOptions . (If multicast addressing is not supported, set to zero.)
5Ch	DriverPollPtr	4	Set this field to point to the HSM's DriverPoll routine before calling MSMEnablePolling . (If this routine is not used, set to zero.)
60h	DriverResetPtr	4	Set this field to point to the HSM's DriverReset routine before calling MSMRegisterHardwareOptions .
64h	DriverSendPtr	4	Set this field to point to the HSM's DriverSend routine before calling MSMRegisterHardwareOptions .
68h	DriverShutdownPtr	4	Set this field to point to the HSM's DriverShutdown routine before calling MSMRegisterHardwareOptions .
6Ch	DriverTxTimeoutPtr	4	If using the RX-Net TSM, set this field to point to the HSM's DriverTxTimeout routine before calling MSMRegisterHardwareOptions . (If the RX-Net TSM is not used, set to zero.)
70h	DriverPromiscuousChangePtr	4	Set this field to point to the HSM's DriverPromiscuousChange routine before calling MSMRegisterHardwareOptions . (If promiscuous mode is not supported, set this field to zero.)
74h	DriverStatisticsChangePtr	4	Set this field to point to the HSM's DriverStatisticsChange routine before calling MSMRegisterHardwareOptions . (If this optional procedure is not supported, set this field to zero.)

3-10 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Table 3.1 Driver Parameter Block Field Descriptions *continued*

Offset	Name	Bytes	Description
78h	DriverRxLookAheadChangePtr	4	Set this field to point to the HSM's DriverRxLookAheadChange routine before calling MSMRegisterHardwareOptions . (If this optional procedure is not supported, set this field to zero.)
7Ch	DriverManagementPtr	4	If a driver accepts management requests from outside NLMs (HMI, BRIDGE, or CSL), set this field to point to the DriverManagement routine before calling MSMRegisterHardwareOptions . (If this optional procedure is not supported, set this field to zero.)
80h	DriverEnableInterruptPtr	4	Set this field to point to the HSM's DriverEnableInterrupt routine before calling MSMRegisterHardwareOptions . (If this procedure is not supported, set this field to zero.)
84h	DriverDisableInterruptPtr	4	Set this field to point to the HSM's DriverDisableInterrupt routine before calling MSMRegisterHardwareOptions . (If this procedure is not supported, set this field to zero.)
88	DriverISR2Ptr	4	Set this field to point to the HSM's DriverISR2 routine before calling MSMSetHardwareInterrupt . If there is no second interrupt, set this field to zero (0). <i>DriverISR2Ptr, DriverDisableInterrupt2, and MLIDCFG_Interrupt2 in the configuration table must all be set the same; either all of them are set or all of them are zero.</i>
8Ch	DriverReserved	4	This field is reserved and must not be modified by the HSM.

HSM Data Structures and Variables **3-11**

Table 3.1 Driver Parameter Block Field Descriptions *continued*

Offset	Name	Bytes	Description
90h	HSMSpecVerString	4	Set this field to point to a version string that describes the version that the HSM is written to. This string is defined by Novell as: "HSM_ASPEC_VERSION: 3.31". Note: One space is required between the colon and the first digit.
94h	DriverPriorityQueuePtr	4	Set this field to point to the HSM's DriverPriorityQueueSupport routine. This routine will be called by the TSM to handle HSM priority packets when the normal send path is congested. If not used, set this field to zero (0). In either case this field must be set before calling MSMRegisterHardwareOptions .
98h	DriverDisableInterrupt2Ptr	4	Set this field to point to the HSM's DriverDisableInterrupt2 routine. If unused, set this field to zero (0).

Driver Configuration Table

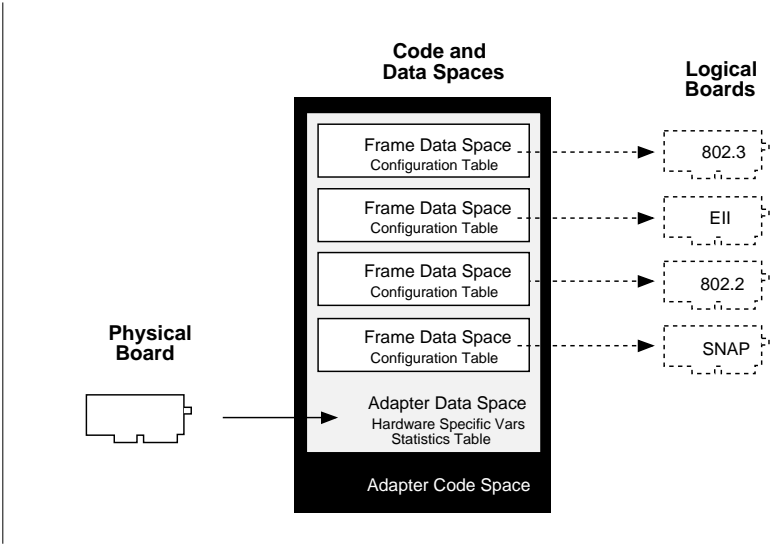
The configuration table is a structure defined by the ODI specification. It contains information about the driver and the adapter's hardware configuration. The HSM must provide a template for initializing the configuration table fields. The MSM creates a copy of the template for each loaded frame type. The configuration table is shown on the following page. A description of each field follows the example.

The configuration table fields are used primarily during initialization to reserve hardware resources. All fields that can be modified from the command line when the driver is loaded, must be set to their default value before calling **MSMParseDriverParameters**. Any field not used must be set to 0, unless otherwise noted. The **MSMParseDriverParameters** routine collects information entered from the command-line and/or interactively from the operator console. Once the configuration table is filled in, the driver uses **MSMRegisterHardwareOptions** to reserve the hardware resources.

Driver Frame Data Space

When **DriverInit** calls **<TSM>RegisterHSM**, the MSM allocates the Frame Data Space and copies the configuration template into this area. For each loaded frame type, there will be a separate Frame Data Space containing a separate configuration table. The MSM and TSM will pass a pointer to the appropriate Frame Data Space when calling HSM procedures. The driver can also access the configuration tables using the **MSMVirtualBoardLink** variable.

Figure 3-2 Frame and Adapter Data Space



Example Template for the Driver Configuration Table (based on the NE2000)

<u>DriverParameterBlock</u>	<u>Label</u>	<u>Dword</u>
.		
.		
.		
DriverConfigTemplatePtr	dd	DriverConfigTemplate
.		
.		
.		
;DriverParameterBlockEnd		

<u>DriverConfigTemplate</u>	<u>Label</u>	<u>Dword</u>
MLIDCFG_Signature	db	'HardwareDriverMLID'
	db	8 dup (' ')
MLIDCFG_MajorVersion	db	1
MLIDCFG_MinorVersion	db	14
*MLIDNodeAddress	db	6 dup (0FFh)
MLIDModeFlags	dw	0010010001001001b
MLIDBoardNumber	dw	0000
MLIDBoardInstance	dw	0000
MLIDMaximumSize	dd	00000000
MLIDMaxRecvSize	dd	00000000
MLIDRecvSize	dd	00000000
MLIDCardName	dd	00000000
MLIDShortName	dd	DriverNICShortName
*MLIDFrameType	dd	00000000
MLIDReserved0	dw	0000
MLIDFrameID	dw	0000
MLIDTransportTime	dw	1
MLIDRouteHandler	dd	00000000
MLIDLineSpeed	dw	10
MLIDLookAheadSize	dw	0000
MLIDCFG_SGCount	db	00
MLIDReserved1	db	00
MLIDPriority Sup	dw	0000
MLIDReserved2	dd	00000000
MLIDMajorVersion	db	00
MLIDMinorVersion	db	00
MLIDFlags	dw	0000000000000000b
*MLIDSendRetries	dw	10
MLIDLink	dd	00000000
MLIDSharingFlags	dw	0000
*MLIDSlot	dw	0000
*MLIDIOPortsAndLengths	dw	0300h, 32, 0, 0
*MLIDMemoryDecode0	dd	00000000
*MLIDMemoryLength0	dw	0000

HSM Data Structures and Variables 3-15

*MLIDMemoryDecode1	dd	00000000
*MLIDMemoryLength1	dw	0000
*MLIDInterrupt	db	3, 0FFh
*MLIDDMAUsage	db	0FFh, 0FFh
MLIDResourceTag	dd	00000000
MLIDConfig	dd	00000000
MLIDCommandString	dd	00000000
MLIDLogicalName	db	18 dup (0)
MLIDLinearMemory0	dd	00000000
MLIDLinearMemory1	dd	00000000
*MLIDChannelNumber	dw	0000
*MLIDBusTag	dd	00000000
MLIDIOCfgMajorVersion	db	1
MLIDIOCfgMinorVersion	db	00

* These values may be configured from the load command line.

Table 3.2 Configuration Table Field Descriptions

Offset	Name	Bytes	Description
00h	MLIDCFG_Signature	26	This field is a mandatory remnant. In pre-MLID LAN drivers, this field contained a string which indicated the start of the configuration table. The string is "HardwareDriverMLID" followed by exactly eight spaces. It must be included in the table.
1Ah	MLIDCFG_MajorVersion	1	This field must be set 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.
1Bh	MLIDCFG_MinorVersion	1	This field must be set 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.
1Ch	MLIDNodeAddress	6	<p>When DriverInit calls <TSM>RegisterHSM, the MSM fills these bytes with FFh then checks the command line for a node address override. If an override address is found, the MSM places the physical layer format of the address in this field.</p> <p>After the driver calls MSMRegisterHardwareOptions, it must check this field for an override.</p> <p>If these bytes are not all FFh, an override occurred and the HSM must set the physical board's address to the value in this field. If there was not an override, the HSM must place the node address read from the hardware in this field.</p>

Table 3.2 Configuration Table Field Descriptions *continued*

Offset	Name	Bytes	Description
			Once the driver calls MSMRegisterMLID , the MSM places the physical layer format of the node address in the MSMPhysNodeAddress variable and sets the appropriate <i>MLIDModeFlag</i> bits. This physical address may be in canonical or noncanonical form. (For more information, refer to <i>MLIDModeFlags</i> , MSMPhysNodeAddress , and <i>The Canonical and Noncanonical Addressing</i> supplement.)
22h	MLIDModeFlags	2	See the bit map that follows this table for <i>MLIDModeFlags</i> .
24h	MLIDBoardNumber	2	The MSM sets this field to the board number assigned by the LSL when DriverInit calls MSMRegisterMLID . Logical board 0 is used internally in the operating system. Drivers are assigned logical board numbers 1 through 255.
26h	MLIDBoardInstance	2	The MSM sets this field when the DriverInit routine calls MSMRegisterHardwareOptions . If the HSM is driving two adapters, all logical boards associated with the first adapter would have a value of 1 and all logical boards associated with the second adapter would have the value 2. Note: Each controller on a multichannel adapter is treated as a separate adapter.
28h	MLIDMaximumSize	4	The MSM sets this field to the LSL's maximum ECB buffer size during <TSM>RegisterHSM . The HSM may lower this value prior to calling MSMRegisterMLID . During MSMRegisterMLID , the TSM will modify this size if the topology requires a smaller maximum packet size. (See the "Maximum Packet Size" section following this table.)
2Ch	MLIDMaxRecvSize	4	The MSM and TSM coordinate to initialize this field during MSMRegisterMLID . The HSM must not modify this field. (See the "Maximum Packet Size" section following this table.)

3-18 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Table 3.2 Configuration Table Field Descriptions *continued*

Offset	Name	Bytes	Description
30h	MLIDRecvSize	4	The MSM and TSM coordinate to initialize this field during MSMRegisterMLID . The HSM must not modify this field. (See the "Maximum Packet Size" section following this table.)
34h	MLIDCardName	4	<p>The HSM must either set this field to 0 (see below), or point to a byte-length preceded, null-terminated, ASCII string that is identical to the description string in the linker definition file (see Appendix A).</p> <p>For example: 14, " NetWare NE2000 ", 0</p> <p>If this field is initialized to zero, the MSM will extract the description string from the NLM header (derived from the linker definition file) when the HSM's DriverInit routine calls <TSM>RegisterHSM. This way, only one description string must be maintained.</p>
38h	MLIDShortName	4	The HSM 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 string is usually the name of the <HSM>.LAN file.
3Ch	MLIDFrameType	4	The MSM sets this field when the HSM's DriverInit routine calls MSMRegisterHardwareOptions . It contains a pointer to a string that describes the HSM's frame type. (See the <i>Frame Types and Protocol IDs</i> supplement.)
40h	MLIDReserved0	2	This field is reserved for future use, and must be set to 0.
42h	MLIDFrameID	2	The MSM sets this field when the DriverInit routine calls MSMRegisterHardwareOptions . It contains the frame type ID number. (See the <i>Frame Types and Protocol IDs</i> supplement.)
44h	MLIDTransportTime	2	This field indicates the time (in ticks) it takes the adapter to transmit a 576-byte packet. Most HSMs set this field to 1. This field cannot be set to 0.

HSM Data Structures and Variables **3-19**

Table 3.2 Configuration Table Field Descriptions *continued*

Offset	Name	Bytes	Description
46h	MLIDRouteHandler	4	This field is used by the TSM and the source routing module, ROUTE.NLM. The HSM must set this field to 0 and then not modify it. If the HSM is using the Token-Ring or FDDI TSM, the field can be modified by ROUTE.NLM, but should not be of any concern to the HSM.
4Ah	MLIDLineSpeed	2	<p>This field contains the speed of the line driver. 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 to 1. This field is undefined if it is set to 0.</p> <p>For example:</p> <p>If the speed of the line driver is 10 Mbps, put 10 (decimal) in this field.</p> <p>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>
4Ch	MLIDLookAheadSize	2	<p>The TSM sets this variable. This is the amount of data required by protocol stacks when previewing received packets. This size may be dynamically changed and can be up to a maximum of 128 bytes. See also DriverRxLookAheadChange.</p> <p>The variable, MSMMaxFrameHeaderSize, is equal to this value plus the maximum media header size.</p>
4Eh	MLIDCFG_SGCount	1	<p>The HSM sets this variable. This field contains the maximum number of scatter/gather elements the adapter is capable of handling. This field is only valid if the <i>DriverSupportsPhysFrgs</i> bit in the <i>MLIDModeFlags</i> field is set.</p> <p>Note: The minimum value is 2. If set less than 2, the MSM will set it to 17 (16 ECB fragments and 1 MAC header).</p>

Table 3.2 Configuration Table Field Descriptions *continued*

Offset	Name	Bytes	Description
4Fh	MLIDReserved1	1	This field is reserved for future use and must be set to 0.
50h	MLIDPrioritySup	2	The number of priority levels that the HSM can handle. This field has a maximum of 7 priorities (1-7). Zero indicates no priority packet support. The HSM can set this field to a value from 0 to 7.
52h	MLIDReserved2	4	This field is reserved for future use and must be set to 0.
56h	MLIDMajorVersion	1	<p>This field may contain the major version number of the LAN driver, or be set to 0 (see below). If it contains the version number it must match the version number specified with the “version” keyword in the linker definition file (see Appendix A).</p> <p>A second option is to initialize this field to zero, in which case the MSM will extract the major version number from the NLM header (derived from the linker definition file) when the DriverInit routine calls <TSM>RegisterHSM. This way, only one version string must be maintained.</p>
57h	MLIDMinorVersion	1	<p>This field contains the minor version number of the LAN driver. The number must match the version number specified with the “version” keyword in the linker definition file (see Appendix A).</p> <p>If the <i>MLIDMajorVersion</i> field is initialized to zero, the MSM will extract the minor version number from the NLM header (derived from the linker definition file) when the HSM's DriverInit routine calls <TSM>RegisterHSM. This way, only one version string must be maintained.</p>
58h	MLIDFlags	2	See the <i>MLIDFlags</i> bit map following this table.

HSM Data Structures and Variables **3-21**

Table 3.2 Configuration Table Field Descriptions *continued*

Offset	Name	Bytes	Description
5Ah	MLIDSendRetries	2	Set this field to a value indicating the number of times the HSM 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	This field is used by the OS and must not be changed.
60h	MLIDSharingFlags	2	See the <i>MLIDSharingFlags</i> bit map following this table.
62h	MLIDSlot	2	<p>For Micro Channel, EISA, PCI and other busses which allow for identification of where the adapter is placed, this field holds the Hardware Instance Number (HIN). This is a system-wide unique handle for the device which is returned by the MSMSearchAdapter call. This value will normally correspond to the number silk-screened on the motherboard or stamped on the chassis of the computer, and is here solely to help the user identify the adapter. In cases where there are integrated motherboard devices, PCI BIOS v2.0 devices, PCI BIOS v2.1 adapters with multiple devices or functions, PnP ISA devices, or a conflict of physical slot numbers, the instances will be assigned a unique value. Set to minus 1 if not used (FFFFh).</p> <p>Note: In the past the unused value for this field was 0, which was a reserved value for MicroChannel and EISA adapters. Older MicroChannel and EISA based adapters may still use 0 to indicate unused, however when a driver is being updated it must be changed to FFFFh.</p>
64h	MLIDIOPortsAndLengths	8	This field contains the I/O port information as described below. (Set to zero if not used.)
(64h)	(MLIDIOPort0)	(2)	Primary base I/O port.
(66h)	(MLIDIORange0)	(2)	Number of I/O ports starting at <i>MLIDIOPort0</i> .
(68h)	(MLIDIOPort1)	(2)	Secondary base I/O port.

3-22 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Table 3.2 Configuration Table Field Descriptions *continued*

Offset	Name	Bytes	Description
(6Ah)	(MLIDIORange1)	(2)	Number of I/O ports starting at <i>MLIDIOPort1</i> .
6Ch	MLIDMemoryDecode0	4	This field contains the absolute primary memory address used by the adapter. If not used, set this field to 0. (See the note in the <i>MLIDLinearMemory0</i> description at offset 9Ah.)
70h	MLIDMemoryLength0	2	If bit #15 of the <i>MLIDSharingFlags</i> is set, this field defines the number of pages of memory decoded at <i>MLIDMemoryDecode0</i> . If bit #15 is clear, this field defines the number of paragraphs (16 bytes) of memory decoded at <i>MLIDMemoryDecode0</i> . If <i>MLIDMemoryDecode0</i> is not defined, set this field to 0. Note: The size of a page of memory is determined by the processor the code is assembled on; i.e. Intel is 4k.
72h	MLIDMemoryDecode1	4	This field contains the absolute secondary memory address used by the adapter. If not used, set this field to 0. (See the note in the <i>MLIDLinearMemory1</i> description at offset 9Eh.)
76h	MLIDMemoryLength1	2	If bit #15 of the <i>MLIDSharingFlags</i> is set, this field defines the number of pages of memory decoded at <i>MLIDMemoryDecode1</i> . If bit #15 is clear, this field defines the number of paragraphs (16 bytes) of memory decoded at <i>MLIDMemoryDecode1</i> . If <i>MLIDMemoryDecode1</i> is not defined, set this field to 0. Note: The size of a page of memory is determined by the processor the code is assembled on; i.e. Intel is 4k.
78h	MLIDInterrupt	2	This field contains interrupt information as described below. (Set to FFh if not used.)
(78h)	(MLIDInterrupt0)	(1)	Primary interrupt vector number.
(79h)	(MLIDInterrupt1)	(1)	Secondary interrupt vector number.

HSM Data Structures and Variables **3-23**

Table 3.2 Configuration Table Field Descriptions *continued*

Offset	Name	Bytes	Description
7Ah	MLIDDMAUsage	2	This field contains DMA channel information as described below. (Set to FFh if not used.)
(7Ah)	(MLIDDMAUsage0)	(1)	Primary DMA channel.
(7Bh)	(MLIDDMAUsage1)	(1)	Secondary DMA channel.
7Ch	MLIDResourceTag	4	This field is set by the MSM and contains a pointer to the <i>IOResourceTag</i> .
80h	MLIDConfig	4	This field is set by the LSL and contains a pointer to the LSL's copy of the configuration table. This is used only by the LSL.
84h	MLIDCommandString	4	This field is set by the MSM to point to a structure containing two fields. The first field is a forward link to the next structure if there is one. The second field is a pointer to a NULL-terminated string containing the parameters entered by the user on the command line. Normally, there will be only one node in the linked list, but if there are more, the command line is the concatenation of all of them. Bits 9 and 10 of the MLIDSharingFlags bit are used in conjunction with this field.
88h	MLIDLogicalName	18	HSMs must not use this field. It contains the logical name of the LAN driver if given one at load time. For example: load NE2000 NAME="_____"
9Ah	MLIDLinearMemory0	4	The operating system fills in this field with the linear address of <i>MLIDMemoryDecode0</i> when the HSM's DriverInit routine calls MSMRegisterHardwareOptions . Do NOT convert <i>MLIDMemoryDecode0</i> to the logical or physical address. If shared memory must be accessed before the hardware options are registered, refer to the MSMReadPhysicalMemory and MSMWritePhysicalMemory support routine descriptions.

Table 3.2 Configuration Table Field Descriptions *continued*

Offset	Name	Bytes	Description
9Eh	MLIDLinearMemory1	4	<p>The operating system fills in this field with the linear address of <i>MLIDMemoryDecode1</i> when the HSM's DriverInit routine calls MSMRegisterHardwareOptions.</p> <p>If shared memory must be accessed before the hardware options are registered, refer to the MSMReadPhysicalMemory and MSMWritePhysicalMemory support routine descriptions.</p>
A2h	MLIDChannelNumber.	2	<p>This field is used for multichannel adapters. It holds the channel number of the NIC to use. The channel number can be specified when a driver is loaded using the "channel=#" keyword (where # is any value greater than zero). Set this field to zero if multichannel are not used.</p>
A4h	MLIDBusTag	4	<p>Pointer to an architecture-dependent value, which specifies the bus on which the adapter is found. Set this field before calling MSMRegisterHardwareOptions. The value placed in this field is returned by SearchAdapter unless the board is Legacy ISA, in which case it is set to zero.</p>
A8h	MLIDIOfgMajorVersion	1	<p>This field must be set to the major version number of the I/O configuration part of the configuration table. The current major version number is 1.</p>
A9h	MLIDIOfgMinorVersion	1	<p>This field must be set to the minor version number of the I/O configuration part of the configuration table. The current minor version number is 0.</p>

MLIDModeFlags Bit Map

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

Table 3.3 MLIDModeFlag Descriptions

Bit	Description
15, 14	<p>The MSM controls these bits. The bits indicate whether the <i>MLIDNodeAddress</i> field of the configuration table contains a canonical or noncanonical address.</p> <p>Bit 15 indicates if the node address format is configurable. If this bit is set, the format is configurable and the HSM must use the MSMPhysNodeAddress variable instead of the configuration table <i>MLIDNodeAddress</i> to obtain the physical layer node address. (For OS versions later than 3.11, the MSM always sets bit 15.)</p> <p>Bit 14 indicates whether <i>MLIDNodeAddress</i> contains the canonical or noncanonical form of the node address. The state of bit 14 is only defined when bit 15 is set.</p> <p>The bit 15 and 14 combinations are:</p> <p>00 = <i>MLIDNodeAddress</i> format is unspecified. The node address is assumed to be in the physical layer's native format -- MSMPhysNodeAddress is not used.</p> <p>01 = This is an illegal value and must not occur.</p> <p>10 = <i>MLIDNodeAddress</i> is canonical -- use MSMPhysNodeAddress.</p> <p>11 = <i>MLIDNodeAddress</i> is non-canonical -- use MSMPhysNodeAddress.</p> <p>(See also <i>MLIDNodeAddress</i>, MSMPhysNodeAddress, and the <i>Canonical and Noncanonical Addressing</i> supplement.)</p>
13	<p>The HSM must set this bit if it supports Promiscuous Mode.</p>
12	<p>This bit (<i>DriverSupportsPhysFrgs</i>) is set by the HSM to inform the TSM, MSM and protocol stacks that the HSM needs ECB fragment pointers physically addressed as described in chapter 5. Public dwords <i>PhysicalToLogical</i> and <i>LogicalToPhysical</i> must no longer be used to translate fragment addresses. Current workstation OS's and future NetWare server OS's that support these HSM's may not be able to support these conversion calls. Typically, bus master adapter HSM's need physical addresses to ECB fragment pointers and control information in memory.</p>
10	<p>The HSM must set this bit if it can handle fragmented RCBs.</p> <p>(RCBs are described in Chapter 4.)</p>
8	<p><i>SMP Bit</i> - The MSM sets this bit if the MSM and TSM support Symmetrical Multi-processing.</p>

3-26 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Table 3.3 MLIDModeFlag Descriptions *continued*

Bit	Description
6	The TSM sets this bit to indicate raw sends are supported. Refer to the TCB section of Chapter 4 for information on raw sends. (RX-Net does not support raw sends.)
4	<i>CSL Compliant Bit</i> - The MLID or HSM sets this bit if the supported data link protocol requires connection management through the Call Support Layer (CSL) interface. Typical Wide Area Network data link protocols such as Frame Relay, PPP, and X.25 are connection oriented and rely upon network layer protocol (IPX, IP, etc.) interaction to establish, maintain and terminate connections to remote peers. The CSL provides extensions to the ODI allowing this connection management interaction between network and data link layer protocols. The CSL Compliant Bit must not be set by connectionless data link protocols, such as Ethernet, Token-Ring, etc. Refer to the "NetWare WAN ODI Specification" for a complete description of the CSL and WAN HSM interfaces (Part No. 107-000045-001).
3	The TSM sets this bit if the HSM supports multicast addressing.
2	Formerly the <i>DXFer</i> (dependability bit) - This bit is no longer used and must be set to 0.
1	This bit has been retired and must be set to 0. It was used as the <i>DriverUsesDma</i> bit in the NetWare 286 environment, but has no meaning in later environments.
0	This bit must be set to 1.

MLIDFlags Bit Map

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

Table 3.4 MLIDSFlags Bit Map Fields

Bit	Description
12	<p>The HSM sets this bit during initialization if the following conditions are met:</p> <p>The HSM has provided a priority queue service support routine (such as DriverPriorityQueueSupport).</p> <p>The HSM has set the MLIDPrioritySup field to something other than 0.</p> <p>Note: The HSM may set/clear this bit to enable /disable priority support as needed.</p>
10, 9	<p>These bits are used to indicate different support mechanisms for multicast filtering and multicast format. These bits are only valid if bit 3 of the <i>MLIDModeFlags</i> is set, indicating that the MLID supports multicast addressing.</p> <p>Set bit 10 if group addressing is supported by specialized adapter hardware (such as hardware utilizing CAM memory). If set, DriverMulticastChange receives a pointer to the TSM maintained multicast address table in ESI and the number of addresses in ECX (this is the default method for the Ethernet TSM).</p> <p>Note: If a driver that normally defaults to using functional addresses also supports group addressing and sets bit 10, it will receive both functional and group addresses.</p> <p>The state of bit 9 is only defined if bit 10 is set. Set bit 9 if the adapter provides 100% filtering of group addresses and the TSM does not need to perform any checking. The HSM can dynamically set and reset bit 9. For example, the TSM may need to filter group addresses because of insufficient CAM memory.</p> <p>Note: Bit 9 is not used by ECB aware HSMs; ECB aware HSMs must do their own filtering of multicast addresses.</p>

Table 3.4 MLIDSFlags Bit Map Fields

Bit	Description
	The values for the bit 10 and 9 combinations are: 00 =The multicast address format defaults to that of the LAN medium. Ethernet => Group Addressing Token-Ring => Functional Address FDDI => Group Addressing 01 =This is an illegal value and must not occur. 10 =Group addressing is supported by a specialized adapter, but the TSM must filter the addresses. 11 =Group addressing is supported by a specialized adapter, and TSM checking is not required.
8	Set to 1 if the HSM supports HUB Management
0-2	Formerly the <i>Bus Flags</i> , these bits are no longer used and must be set to 0. There are several new MSM procedures listed in Chapter 7 to handle bus information, such as MSMSearchAdapter .

MLIDSharingFlags Bit Map

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

Table 3.5 MLIDSharingFlags Bit Map

Bit #	Description
15	<p>This bit signifies when set, that the values in the fields MLIDMemoryLength0 and MLIDMemoryLength1, contain the number of pages of memory used by the adapter. For example, on Intel platforms, 4K pages allows a maximum of 256 MB of shared memory to be used by the adapter.</p> <p>When cleared, this bit signifies that the values in fields MLIDMemoryLength0 and MLIDMemoryLength1 contain the number of paragraphs (16 bytes) of memory used by the adapter.</p>
10	<p>Bits 10 and 9 are currently used only by the NetWare Server install program.</p> <p>If bit 10 is zero and bit 9 is zero, the install program gets information from the system IOCONFIG structure and places it in the AUTOEXEC.NCF file.</p> <p>If bit 10 is zero and bit 9 is set, the install program gets the information entered by the user on the command line and merges it with the information from the system IOCONFIG structure and places it in the AUTOEXEC.NCF file.</p> <p>If bit 10 is set, the install program ignores bit 9 and places only the information entered by the user on the command line in the AUTOEXEC.NCF file.</p>
9	<p>Bits 10 and 9 are currently used only by the NetWare Server install program.</p> <p>If bit 10 is zero and bit 9 is zero, the install program gets information from the system IOCONFIG structure and places it in the AUTOEXEC.NCF file.</p> <p>If bit 10 is zero and bit 9 is set, the install program gets the information entered by the user on the command line and merges it with the information from the system IOCONFIG structure and places it in the AUTOEXEC.NCF file. (The user must enter a value for every field used by the IOCONFIG structure.) Bit 9 is set by the MSM when a valid command line is passed to DriverInit.</p>
8	Set to 1 if the adapter can share DMA channel 1.
7	Set to 1 if the adapter can share DMA channel 0.
6	Set to 1 if the adapter can share interrupt 1.
5	Set to 1 if the adapter can share interrupt 0.
4	Set to 1 if the adapter can share memory range 1.
3	Set to 1 if the adapter can share memory range 0.

3-30 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Table 3.5 MLIDSharingFlags Bit Map

Bit #	Description
2	Set to 1 if the adapter can share I/O port 1.
1	Set to 1 if the adapter can share I/O port 0.
0	Set to 1 if the adapter is currently shut down.

Maximum Packet Size The *MLIDMaximumSize* field of the configuration table is set to the LSL's maximum ECB buffer size during **<TSM>RegisterHSM** (this defaults to 4K but can be changed in the STARTUP.NCF file to a maximum of 24K). The HSM could lower this value prior to calling **MSMRegisterMLID**. During this procedure, the TSM alters the size if the topology requires a smaller maximum packet size. The TSM also sets the *MLIDMaxRecvSize* and *MLIDRecvSize*. After **MSMRegisterMLID** returns, drivers for intelligent adapters may pass the maximum size to the hardware if required. The following table shows how these values are determined.

Frame Type	MLIDMaximumSize	MLIDMaxRecvSize	MLIDRecvSize
Arcnet	Maximum ECB Buffer Size	Maximum ECB Buffer Size	Maximum ECB Buffer Size
Ethernet 802.3	Maximum ECB Buffer Size OR 1514 (whichever is less)	MLIDMaximumSize (minus 14)	MLIDMaximum- Size (minus 14)
Ethernet 802.2	Maximum ECB Buffer Size OR 1514 (whichever is less)	MLIDMaximumSize (minus 17)	MLIDMaximum- Size (minus 18)
Ethernet II	Maximum ECB Buffer Size OR 1514 (whichever is less)	MLIDMaximumSize (minus 14)	MLIDMaximum- Size (minus 14)
Ethernet SNAP	Maximum ECB Buffer Size OR 1514 (whichever is less)	MLIDMaximumSize (minus 22)	MLIDMaximum- Size (minus 22)
Token-Ring 802.2	Maximum ECB Buffer Size OR the maximum size the adapter can handle (whichever is less)	MLIDMaximumSize (minus 17)	MLIDMaximum- Size (minus 48)
Token-Ring SNAP	Maximum ECB Buffer Size OR the maximum size the adapter can handle (whichever is less)	MLIDMaximumSize (minus 22)	MLIDMaximum- Size (minus 52)
FDDI 802.2	Maximum ECB Buffer Size OR 4491 (whichever is less)	MLIDMaximumSize (minus 16)	MLIDMaximum- Size (minus 47)
FDDI SNAP	Maximum ECB Buffer Size OR 4491 (whichever is less)	MLIDMaximumSize (minus 21)	MLIDMaximum- Size (minus 51)

3-32 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Example

If the maximum ECB buffer size = 8192 bytes and the Token-Ring adapter can handle 4096 bytes, then the Token-Ring 802.2 values are calculated as follows:

MLIDMaximumSize = 4096

MLIDMaxRecvSize

The maximum packet size minus the headers if the source routing header is not included.

= 4096 – MAC header (14) – 802.2 Type I LLC header (3)
= 4079

MLIDRecvSize

The maximum packet size minus the headers if the source routing header

(SRT)* is included.

= 4096 – MAC header (14) – 802.2 Type II LLC header (4) –
Source
Routing header (30)
= 4048

* Refers to the IEEE SRT Specification.

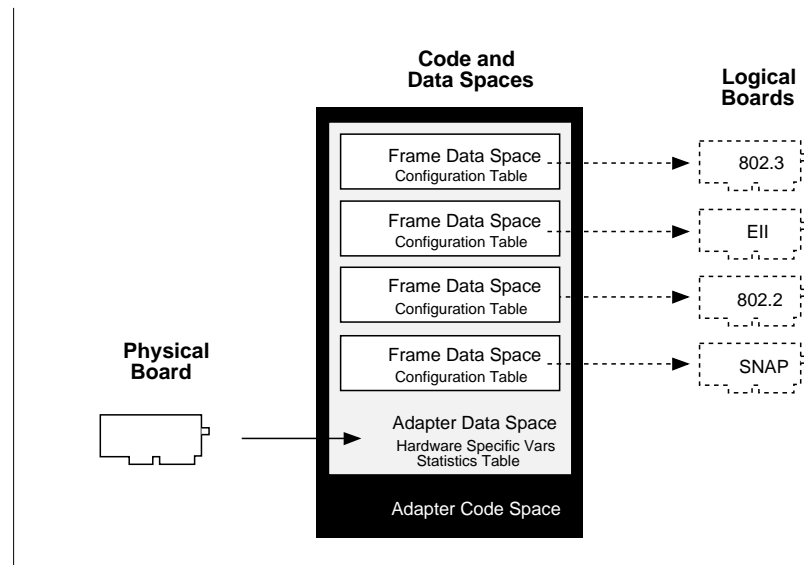
Driver Adapter Data Space

The HSM must define and initialize a structure containing data specific to a particular physical board. This structure is called the *DriverAdapterDataSpaceTemplate*. The developer determines what hardware-specific fields are needed in order to drive a particular physical board; however, the structure must contain the *DriverStatisticsTable*. The statistics table is defined by the ODI specification. The following page shows the template format and fields.

When the **DriverInit** routine calls **MSMRegisterHardwareOptions**, the MSM allocates the Adapter Data Space and creates a copy of the driver's template in this area. There will be one Adapter Data Space allocated for each physical board, regardless of the number of frame types supported.

HSM Data Structures and Variables 3-33

Figure 3-3 Frame and Adapter Data Space



Driver Statistics Table

The statistics table contains various diagnostic counters. All statistics counters shown must be present in the table, however, only those marked “mandatory” are required to be supported. These counters can be grouped into the following categories.

- *Generic Statistics Counters*
 - Standard Counters
 - Media Specific Counters
- *Custom Statistics Counters*

3-34 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Driver Adapter Data Space and Statistics Table Template

DriverAdapterDataSpace	struc	
.		
.		
{*** Hardware Specific Variables ***}		
.		
.		
DriverStatisticsTable	db	0 dup (?) ; (Label)
StatMajorVersion	db	3
StatMinorVersion	db	0
NumGenericCounters	dw	(GenericEnd - GenericBegin) / 4
CounterMask0	dd	
1111101100001111111011111111111b		
GenericBegin	db	0 dup (?) ; (Label)
TotalTxPacketCount	dd	0 ; TSM (mandatory)
TotalRxPacketCount	dd	0 ; TSM (mandatory)
NoECBAvailableCount	dd	0 ; TSM (mandatory)
PacketTxTooBigCount	dd	0 ; TSM (mandatory)
Reserved1	dd	0 ; (reserved)
PacketRxOverflowCount	dd	0 ; HSM (optional)
PacketRxTooBigCount	dd	0 ; TSM (mandatory)
PacketRxTooSmallCount	dd	0 ; TSM (optional)
TotalTxMiscCount	dd	0 ; HSM (mandatory)
TotalRxMiscCount	dd	0 ; HSM (mandatory)
RetryTxCount	dd	0 ; HSM (optional)
ChecksumErrorCount	dd	0 ; HSM (optional)
HardwareRxMismatchCount	dd	0 ; TSM (optional)
TotalTxOKByteCountLow	dd	0 ; TSM (mandatory)
TotalTxOKByteCountHigh	dd	0 ; TSM (mandatory)
TotalRxOKByteCountLow	dd	0 ; TSM (mandatory)
TotalRxOKByteCountHigh	dd	0 ; TSM (mandatory)
TotalGroupAddrTxCount	dd	0 ; TSM (mandatory)
TotalGroupAddrRxCount	dd	0 ; TSM (mandatory)
AdapterResetCount	dd	0 ; HSM (mandatory)
AdapterOprTimeStamp	dd	0 ; MSM (mandatory)
QDepth	dd	0 ; TSM (mandatory)
.		
.		
{*** Media 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 <>
```

Offset	Name	Bytes	Description
00h	StatMajorVersion	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 media-specific counters) present in the statistics table but not necessarily supported. This number must 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 media-specific portions of the statistics table are implemented in the driver. If the bit is zero the counter is supported. (see also the bit map definition following this table).</p> <p>If there are more than 32 standard and media-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>

3-36 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Offset	Name	Bytes	Description
08h	TotalTxPacketCount	4	The TSM increments this counter whenever a packet is successfully transmitted by the adapter.
0Ch	TotalRxPacketCount	4	The TSM increments this counter whenever a packet is successfully received by the adapter.
10h	NoECBAvailableCount	4	The TSM increments this counter if it cannot obtain an RCB for a received packet.
14h	PacketTxTooBigCount	4	The TSM increments this counter whenever a packet is too big for the adapter to transmit.
18h	Reserved1	4	This field is reserved for use by the MSM, but must be initialized to zero.
1Ch	PacketRxOverflowCount	4	The HSM may use 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 TSM increments this counter whenever a packet is received that is too large for the provided receive buffer(s).
24h	PacketRxTooSmallCount	4	Some TSMs increment this counter if a packet is received that is too small for media definitions. Currently only the RX-Net TSM maintains this counter.
28h	TotalTxMiscCount	4	The HSM must increment this counter if a fatal transmit error occurs and there is no other appropriate standard counter to increment in the generic portion of the statistics table. The HSM may also increment a media specific or custom counter for this event.
2Ch	TotalRxMiscCount	4	The HSM must increment this counter if a fatal receive error occurs and there is no other appropriate standard counter to increment in the generic portion of the statistics table. The HSM may also increment a media specific or custom counter for this event.
30h	RetryTxCount	4	The HSM may use this counter to indicate the number of times packet transmissions were retried due to failure.
34h	ChecksumErrorCount	4	The HSM may use this counter to indicate the number of times a packet was received with corrupt data (CRC errors...etc).

HSM Data Structures and Variables **3-37**

Offset	Name	Bytes	Description
38h	HardwareRxMismatchCount	4	Some TSMs increment this counter when a packet is received that does not pass length consistency checks. Currently only the Ethernet TSM maintains this counter.
3Ch	TotalTxOKByteCountLow	4	The number of bytes including low level headers successfully transmitted. The MSM maintains this counter.
40h	TotalTxOKByteCountHigh	4	Upper 32-bits of the <i>TotalTxOKByteCount</i> counter.
44h	TotalRxOKByteCountLow	4	The number of bytes including low level headers successfully received. The MSM maintains this counter.
48h	TotalRxOKByteCountHigh	4	Upper 32-bits of the <i>TotalRxOKByteCount</i> counter
4Ch	TotalGroupAddrTxCount	4	The number of packets transmitted with a Group destination address (maintained by the MSM).
50h	TotalGroupAddrRxCount	4	The number of packets received with a Group destination address (maintained by the MSM).
54h	AdapterResetCount	4	The number of times the adapter was reset due to internal failure or other calls to the DriverReset routine. The HSM must maintain this counter.
58h	AdapterOprTimeStamp	4	This field contains a time stamp indicating when the adapter last changed operational state (load, shutdown, reset...) (maintained by the MSM).
5Ch	QDepth	4	This field reflects the number of Transmit ECBs that are queued for the adapter. The TSM maintains this field.
60h	(Media Specific Counters)	4 each	See the "Media Specific Counters" section following this table. The HSM must maintain these counters.
??h	NumCustomCounters.	2	This field contains the number of custom counters defined by the HSM. For example, a counter could be created 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 below)

3-38 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Offset	Name	Bytes	Description
??h	CustomCounter1	4 each	These fields contain custom counters that can be configured differently for the specific needs of the HSM or adapter.
??h	CustomCounterStrings	4	<div>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.</div> <div><div>CustomStrings</div><div>label</div><div>dword</div><div>CustomStrSize</div><div>dw</div><div>(CustomStrEnd - CustomStrings)</div><div>db</div><div>'Custom String 1', 0</div><div>db</div><div>'Custom String 2', 0</div><div>db</div><div>'Custom String 3', 0</div><div>db</div><div>'Custom String N',</div><div>0</div><div>db</div><div>0, 0</div><div>CustomStrEnd</div><div>db</div><div>0 dup (?)</div></div>

CounterMask Bit Maps

The *CounterMask0* field of the statistics table is a bit mask indicating which counters in the standard and media-specific portions of the table are supported by the driver. If there are more than 32 standard and media-specific counters (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 will continue every 32 counters as more statistics are added to the table in the future.

The Status field in the table below indicates which module is responsible for maintaining the counter and whether it is optional or mandatory. Most of the standard counters are maintained by the TSM or MSM. However, several counters must be maintained by the HSM and several may be optionally supported. A bit value of 0 means the counter is supported, but must be set to 1 if not supported. The MSM and TSM will clear the bits that they maintain.

CounterMask0

3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0										
1	1	1	1	1			1	1					1	1	1	1	1	1	1		1	1									

Bit#	Counter	Status
31	TotalTxPacketCount	TSM (mandatory)
30	TotalRxPacketCount	TSM (mandatory)
29	NoECBAvailableCount	TSM (mandatory)
28	PacketTxTooBigCount	TSM (mandatory)
27	Reserved1	(reserved)
26	PacketRxOverflowCount	HSM (optional)
25	PacketRxTooBigCount	TSM (mandatory)
24	PacketRxTooSmallCount	TSM (optional)
23	TotalTxMiscCount	HSM (mandatory)
22	TotalRxMiscCount	HSM (mandatory)
21	RetryTxCount	HSM (optional)
20	ChecksumErrorCount	HSM (optional)
19	HardwareRxMismatchCount	TSM (optional)
18	TotalTxOKByteCountLow	TSM (mandatory)
17	TotalTxOKByteCountHigh	TSM (mandatory)
16	TotalRxOKByteCountLow	TSM (mandatory)
15	TotalRxOKByteCountHigh	TSM (mandatory)
14	TotalGroupAddrTxCount	TSM (mandatory)
13	TotalGroupAddrRxCount	TSM (mandatory)
12	AdapterResetCount	HSM (mandatory)
11	AdapterOprTimeStamp	MSM (mandatory)

10 QDepth TSM (mandatory)

Bits 9...0 of CounterMask0 and continuing through CounterMask1 bits 31... (if needed) correspond to the media-specific counters shown below.

(Any bit not used must be set to 1)

Media Specific Counters

The statistics table must contain the media specific counters defined in this section for the topology.

Token-Ring

ACErrorCounter	dd	0	; 60h	Mandatory
AbortDelimiterCounter	dd	0	; 64h	Mandatory
BurstErrorCounter	dd	0	; 68h	Mandatory
FrameCopiedErrorCounter	dd	0	; 6Ch	Mandatory
FrequencyErrorCounter	dd	0	; 70h	Mandatory
InternalErrorCounter	dd	0	; 74h	Mandatory
LastRingStatus	dd	0	; 78h	Mandatory
LineErrorCounter	dd	0	; 7Ch	Mandatory
LostFrameCounter	dd	0	; 80h	Mandatory
TokenErrorCounter	dd	0	; 84h	Mandatory
CounterMask1**	dd			
00001111111111111111111111111111b				
UpstreamNodeHighDword	dd	0	; 8Ch	Mandatory
UpstreamNodeLowWord	dd	0	; 90h	Mandatory
LastRingID	dd	0	; 94h	Mandatory
LastBeaconType	dd	0	; 98h	Mandatory

**Important: CounterMask1 is included when calculating the NumGenericCounters field of the statistics table.

HSM Data Structures and Variables **3-41**

Table 3.6 Media Specific Counters for Token Ring

Offset	Name	Bytes	Description
60h	ACErrorCounter	4	This counter is incremented when a station receives an AMP or SMP frame with A equal to C equal to 0, and then receives another SMP frame with A equal to C equal to 0 without first receiving an AMP frame.
64h	AbortDelimiterCounter	4	This counter is incremented when a station transmits an abort delimiter while transmitting.
68h	BurstErrorCounter	4	This counter is incremented 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	This counter is incremented when a station recognizes a frame addressed to its specific address and detects that the FS field A bits are set to 1 indicating a possible line hit or duplicate address.
70h	FrequencyErrorCounter	4	This counter is incremented 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	This counter is incremented when a station recognizes a recoverable internal error. This can be used for detecting a station in marginal operating condition.
78h	LastRingStatus	4	This value contains the last Ring Status reported by the adapter with the following bit definitions:

- bit 15 signal loss
- bit 14 hard error
- bit 13 soft error
- bit 12 transmit beacon
- bit 11 lobe wire fault
- bit 10 auto-removal error 1
- bit 9 reserved
- bit 8 remove received
- bit 7 counter overflow
- bit 6 single station
- bit 5 ring recovery
- bit 0-4 reserved

Table 3.6 Media Specific Counters for Token Ring

Offset	Name	Bytes	Description
7Ch	LineErrorCounter	4	<p>This counter is incremented when a frame or token is copied or repeated by a station, the E bit is 0 in the frame or token, and one of the following conditions exist:</p> <ol style="list-style-type: none"> 1) There is a nondata bit (J or K) between the SD and the ED of the frame or token. 2) There is an FCS error in the frame. <p>The first station detecting a line error increments its appropriate error counter and sets E=1 in the ED of the frame. This prevents other stations from logging the error and isolates the source of the disturbance to the proper error domain.</p>
80h	LostFrameCounter	4	<p>This counter is incremented 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 (thus causing the active monitor to issue a new token).</p>
84h	TokenErrorCounter	4	<p>This counter is incremented when a station acting as the active monitor recognizes an error condition that needs a token transmitted. This occurs when the TVX timer expires.</p>
88h	CounterMask1	4	<p>This field is a bit mask indicating the status of the next set of counters. The most significant bit corresponds to <i>UpstreamNodeHighDword</i>. If a bit is zero, the counter is supported.</p>
8Ch	UpstreamNodeHighDword	4	<p>This contains the high 4 bytes of the 6 byte Upstream Neighbor Node Address.</p>
90h	UpstreamNodeLowWord	4	<p>This contains the lower 2 bytes of the 6 byte Upstream Neighbor Node Address.</p>
94h	LastRingID	4	<p>This contains the value of the local ring.</p>
98h	LastBeaconType	4	<p>This contains the value of the last beacon type.</p>

Ethernet

TxOKSingleCollisionsCount Mandatory	dd	0	; 60h
TxOKMultipleCollisionsCount Mandatory	dd	0	; 64h
TxOKButDeferred Mandatory	dd	0	; 68h
TxAbortLateCollision Mandatory	dd	0	; 6Ch
TxAbortExcessCollision Mandatory	dd	0	; 70h
TxAbortCarrierSense Mandatory	dd	0	; 74h
TxAbortExcessiveDeferral Mandatory	dd	0	; 78h
RxAbortFrameAlignment Mandatory	dd	0	; 7Ch

Table 3.7 Media Specific Counters for Ethernet

Offset	Name	Bytes	Description
60h	TxOKSingleCollisionsCount	4	The number of frames involved in a single collision that are subsequently transmitted successfully. Increment this counter when the result of a transmission is reported as <i>transmitOK</i> and the attempt value is 2.
64h	TxOKMultipleCollisionsCount	4	The number of frames involved in more than one collision that are subsequently transmitted successfully. Increment this counter when the result of a transmission is reported as <i>transmitOK</i> and the attempt value is greater than 2 and less than or equal to the <i>attemptLimit</i> .
68h	TxOKButDeferred	4	Increment this counter for frames whose transmission was delayed on the first attempt because the medium was busy.
6Ch	TxAbortLateCollision	4	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.

3-44 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Table 3.7 Media Specific Counters for Ethernet

Offset	Name	Bytes	Description
70h	TxAbortExcessCollision	4	The number of frames not transmitted successfully due to excessive collisions. Increment this counter when the attempts value equals the <i>attemptLimit</i> during a transmission.
74h	TxAbortCarrierSense	4	The number of times the <i>carrierSense</i> variable was not asserted or was deasserted during transmission of a frame without collision.
78h	TxAbortExcessiveDeferral	4	The number of frames deferred for an excessive period of time. Increment this counter only once per LLC transmission.
7Ch	RxAbortFrameAlignment	4	The number of frames that are not an integral number of octets in length and do not pass the FCS check.

FDDI

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
FRingManagementCount	dd	0	; 7Ch	Mandatory
FLCTFailureCount	dd	0	; 80h	Mandatory
FLemRejectCount	dd	0	; 84h	Mandatory
CounterMask1	dd	00111111111111111111111111111111b		
		; **		
FLemCoun	dd	0	; 8Ch	Mandatory
LConnectionState	dd	0	; 90h	Mandatory

Table 3.8 Media Specific Counters for FDDI

Offset	Name	Bytes	Description
60h	FConfigurationState	4	(ANSI fddiSMTCFState) 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 FUpstreamNodeLowWord	8	(ANSI fddiMACUpstreamNbr) The MAC's upstream neighbor's long individual MAC address (0 if unknown).
6Ch	FDownstreamNodeHighDword FDownstreamNodeLowWord	8	(ANSI fddiMACDownstreamNbr) The MAC's downstream neighbor's long individual MAC address (0 if unknown).
74h	FFrameErrorCount	4	(ANSI fddiMAError-Ct) The number of frames that were detected in error by this MAC that had not been detected in error by another MAC.
78h	FFramesLostCount	4	(ANSI fddiMACLost-Ct) The number of instances that this MAC detected a format error during frame reception such that the frame was stripped.
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.

Table 3.8 Media Specific Counters for FDDI

Offset	Name	Bytes	Description
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 <i>FLemCount</i> . If a bit is zero, the counter is supported.
8Ch	FLemCount	4	(ANSI fddiPORTLem-Ct) The aggregate link error monitor error count (zero only on station power up).
90h	FConnectionState	4	(ANSI fddiPortPCM-State) The state of this port's PCM state machine. <div style="display: flex; justify-content: space-between;"> <div> 0=Off 1=Break 2=Trace 3=Connect 4=Next </div> <div> 5=Signal 6=Join 7=Verify 8=Active 9=Maint </div> </div>

FDDI TSM and Bit Swapping Changes

Several changes were made for FDDI with the release of version 2.20 of the FDDI TSM. This includes bit swapping now being handled by either the adapter or the HSM.

Please note the 3 bytes before the frame control byte in the packet header were eliminated, and the *MLIDRecvSize* was changed. Refer to the *Frame Types and Protocol IDs* supplement for details.



A bit swapping table is now provided by the MSM, which eliminates the need for a table in the HSM; see *MSM Global Variables*, *MSMBitSwapTable* in Chapter 4.

HSM Data Structures and Variables **3-47**

RX-Net

Note:The module responsible for maintaining each of the RX-Net specific counters is indicated in parenthesis below.

NoResponseToFreeBufferEnquiry	dd	0	; 60h	Mandatory	(HSM)
NetworkReconfigurationCount	dd	0	; 64h	Mandatory	(HSM)
InvalidSplitFlagInPacketFrag	dd	0	; 68h	Mandatory	(TSM)
OrphanPacketFragmentCount	dd	0	; 6Ch	Mandatory	(TSM)
ReceivePacketTimeout	dd	0	; 70h	Mandatory	(TSM)
FreeBufferEnquiryNAKTimeout	dd	0	; 74h	Mandatory	(TSM)
TotalTxPacketFragmentsOK	dd	0	; 78h	Mandatory	(HSM)
TotalRxPacketFragmentsOK	dd	0	; 7Ch	Mandatory	(HSM)

Table 3.9 Media Specific Counters for RX-Net

Offset	Name	Bytes	Description
60h	NoResponseToFreeBufferEnquiry	4	The HSM increments this counter each time there is no response from the receiving node to FREE BUFFER ENQUIRY.
64h	NetworkReconfigurationCount	4	The HSM increments this counter each time a NETWORKRECONFIGURATION occurs.
68h	InvalidSplitFlagInPacketFrag	4	The TSM maintains this counter of the number of times the <i>Split Flag</i> in the packet fragment is not the value expected. For example, packet fragments received out of order cause this count to increment.
6Ch	OrphanPacketFragmentCount	4	The TSM increments this count each time a packet fragment is received that is not a part of a previously received packet and, therefore, cannot be appended.
70h	ReceivePacketTimeout	4	The TSM increments this counter each time a received packet times out waiting for the rest of the packet fragments to arrive.
74h	FreeBufferEnquiryNAKTimeout	4	The TSM increments this count each time a transmit packet times out waiting for an acknowledgment to a FREE BUFFER ENQUIRY from the receiving node.
78h	TotalTxPacketFragmentsOK	4	The HSM's count of the number of packet fragments successfully sent.
7Ch	TotalRxPacketFragmentsOK	4	The HSM's count of the number of packet fragments successfully received.

Driver Firmware

Drivers may need to download firmware to intelligent adapters. Since most intelligent adapters employ an onboard 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 HSM at link time and then transferred to the adapter during initialization.

To attach a firmware binary file to the HSM, the linker definition file (see Appendix A) must include the “custom” keyword followed by the name of the binary file. When the driver is linked, the file will then be attached to the end of the HSM code (and become part of the NLM).

During the initialization process, the MSM allocates a buffer and copies the contents of the attached file to that buffer. In order to gain access to the firmware buffer, the HSM must properly initialize the `DriverParameterBlock` variables described below. The MSM resolves the value of these parameters when the HSM's **DriverInit** routine calls `<TSM>RegisterHSM`. The HSM can then download the contents of the firmware buffer to the adapter.

DriverFirmwareSize. If custom firmware is used, the HSM must initialize this dword variable to any nonzero value at assembly time. The MSM replaces the value with the actual size of the firmware buffer when **DriverInit** calls `<TSM>RegisterHSM`. If custom firmware is not used, the HSM must initialize this variable to zero at assembly time.

DriverFirmwareBuffer. This dword value is set by the MSM to point at the firmware buffer when **DriverInit** calls the routine `<TSM>RegisterHSM`.

Example

The following example shows how an HSM would define and use the custom firmware variables. This example assumes that the firmware binary file was attached to the driver at link time as described above.

```
DriverParameterBlock                                label    dword
.
.
.
    DriverFirmwareSize          dd    -1
    DriverFirmwareBuffer        dd    0
.
.
.
;DriverParameterBlockEnd

DriverInit      proc
.
.
.
                                call <TSM>RegisterHSM
                                .
                                .
                                .
    mov  eax,    DriverFirmwareSize
    mov  esi,    DriverFirmwareBuffer
    mov  edi,    [ebp].AdapterFirmwareAddress
    mov  ecx,    eax
    shr  ecx,    2
rep movsd
    and  eax,    03h
    mov  ecx,    eax
rep movsb
.
.
.
DriverInit      endp
```


Driver Keywords

Drivers can define keywords that allow custom parameters or flags to be entered from the “load” command-line. In order to use custom keywords, you must define the following `DriverParameterBlock` fields.

DriverNumKeywords. The number of custom keywords defined by the HSM. If custom keywords are not used, set this field to 0.

DriverKeywordText. Pointer to a table of pointers to strings that define the custom keywords. The strings must be uppercase. The MSM uses these when parsing the load command-line.

DriverKeywordTextLen. Pointer to a table containing the length of each custom keyword string defined in *DriverKeywordText*. The length fields are also used to support the optional custom keyword enhancements described in the next section.

DriverProcessKeywordTab. The HSM must provide a procedure to process each of the defined custom keywords. This is a pointer to a table containing pointers to those procedures.

Custom keywords are processed during initialization when **DriverInit** calls **MSMParseDriverParameters**. The MSM processes custom keywords after the OS parses the standard command-line parameters for the configuration table. If the MSM encounters a defined custom keyword in the command-line, it calls the procedure corresponding to that keyword.

On entry to the custom keyword procedure, `EBX` normally points to the Frame Data Space and `ESI` points to the position in the command-line string where the keyword was found. The command-line string is a null-terminated ASCII string. The driver is responsible for extracting and processing any parameters for that keyword, therefore, the parameter format used is controlled by the developer.



If the driver must have custom keywords processed earlier in initialization, the **DriverInit** routine can call **MSMParseCustomKeywords**. Refer to Chapter 7 for detailed information and examples on using this routine.

HSMs that support many custom keywords may have trouble specifying them on the limited space of the command-line. Command-line parameters can now be listed in a driver configuration file (see Appendix A for details). If the driver configuration file is used, **MSMParseDriverParameters** will process the

information from the file along with any other arguments on the command-line. The MSM frees the buffer used for the file before returning to **DriverInit**, so the HSM must copy all required information when the keyword routine is called.

The following example shows how an HSM would define and use two custom keywords. The command-line for this example might be entered as follows:
load <driver> max_packet_size=4202 cable_type_thick

Example

```
DriverParameterBlock      label      dword
...
    DriverNumKeywords      dd         2
    DriverKeywordText      dd         KeywordTextTable
    DriverKeywordTextLen   dd         KeywordTextLenTable
    DriverProcessKeywordTab dd         KeywordProcedureTable
...
;DriverParameterBlockEnd

Keyword1                  db         'MAX_PACKET_SIZE'
Keyword2                  db         'CABLE_TYPE_THICK'

KeywordTextTable          dd         Keyword1
                          dd         Keyword2

KeywordTextLenTable       dd         15                                ; Keyword #1
Length                    dd         16                                ; Keyword #2
Length

KeywordProcedureTable      dd         ProcessMaxPacketSize           ; Keyword #1 Proc.
                          dd         ProcessCableTypeThick           ; Keyword #2 Proc.

; ESI = Ptr to the position in the command line string where the keyword was found. The command
line string is a
;          null-terminated ASCII string.
; EBX =Ptr to the Frame Data Space (Configuration Table)
; Interrupts are Disabled
; CLD is in effect

ProcessMaxPacketSize      proc
    add     esi, 15                                ; Skip over the Keyword Text
    .
    .
```

```

    .
    (parse the remainder of the string to extract the maximum packet size)
    .
    .
    .
    call    ConvertAtol                ; Convert from ASCII to Integer Form
    mov     MyMaxPacketSize,  eax      ; Set Custom Maximum Packet Size Variable
    ret
ProcessMaxPacketSize    endp

ProcessCableTypeThick   proc

    mov     CableType,THICK           ;Signal Thick Cable Type
    ret

ProcessCableTypeThick   endp

```

Driver Keyword Enhancements

The custom keyword table has been enhanced to support the parsing flags described below. These enhancements were added to the existing HSM keyword structure so that HSM's using the old structure will work without modifications. These parsing flags can be logically ORed with the existing keyword text length parameters.



Note

If none of the flags are used, the parser only provides a pointer to the custom keyword in ESI as described in the previous section.

T_REQUIRED. The keyword must be entered. If it doesn't exist on the command-line or configuration file, the user will be prompted for it. If the users does not enter a value, **MSMParseDriverParameters** will return with an error.

T_STRING. The keyword routine will be called with a pointer to the beginning of the string that matched the keyword text.

Example:

```
load <driver> custom int=3
```

The keyword routine called with ESI pointing to "custom int=3"

T_NUMBER. The keyword routine will be called with the value entered on the command-line in EAX. The user must enter a decimal number.

HSM Data Structures and Variables **3-53**

Example:

```
load <driver> custom=100
```

The keyword routine called with EAX = 64h

T_HEX_NUMBER. The keyword routine will be called with the value entered on the command-line in EAX. The user must enter a hexadecimal number.

Example:

```
load <driver> custom=100
```

The keyword routine called with EAX = 100h

T_HEX_STRING. The keyword routine will be called with ESI pointing to a six byte value that was entered on the command-line. The user must enter this string using hexadecimal numbers.

Example:

```
load <driver> custom=01020304
```

The keyword routine called with ESI -> 00, 00, 01, 02, 03, 04

The HSM must provide parsing information immediately after the text string if it has set any of the flags (except when using T_STRING, no parameters are needed). The parameters that are needed depend on the flags that are set. The following is list of the parameters expected.

The following structure is used for keywords using T_NUMBER or T_HEX_NUMBER:

```
UnsignedLongType      struc
LongMinValue          dd      0          ; Minimum value to be accepted.
LongMaxValue          dd      0          ; Maximum value to be accepted.
UnsignedLongType      ends
```

The following structure is used for keywords using T_NUMBER or T_HEX_NUMBER and T_REQUIRED:

```
PromptUnsignedLongType      struc
PLongMinValue               dd      0          ; Minimum value to be accepted.
PLongMaxValue               dd      0          ; Maximum value to be accepted.
```

LongDefaultString default number	dd	0	; Ptr to a string that contains the ; or 0 for no default
LongValidString input characters. If set	dd	0	; Ptr to a string that contains valid ; to zero, the MSM assumes: ; "0..9" for T_NUMBER and ; "0..9A..F" for T_HEX_NUMBER.
LongPromptString PromptUnsignedLongType	dd ends	0	; Ptr to a prompt string.

The following structure is used for keywords using T_HEX_STRING:

SixByteType	struc		
SixByteMinValue	db	6 dup (0)	; Minimum value to be accepted.
SixByteMaxValue	db	6 dup (0)	; Maximum value to be accepted.
SixByteType	ends		

The following structure is used for keywords using T_HEX_STRING and T_REQUIRED:

PromptSixByteType	struc		
PSixByteMinValue	db	6 dup (0)	; Minimum value to be accepted.
PSixByteMaxValue	db	6 dup (0)	; Maximum value to be accepted.
SixByteDefaultStr six byte number or 0	dd	0	; Ptr to a string that contains; default ; for no default.
SixBytePromptStr PromptSixByteType	dd ends	0	; Ptr to a prompt string.

The following is an example of an HSM keyword table that uses the various enhancement options described in this section.

Example

DriverParameterBlock	label	dword
...		
DriverNumKeywords	dd	4
DriverKeywordText	dd	KeywordTextTable
DriverKeywordTextLen	dd	KeywordTextLenTable
DriverProcessKeywordTab	dd	KeywordProcedureTable
...		
;DriverParameterBlockEnd		
KeywordTextTable	dd	DIXText
	dd	TxBelow16Text

HSM Data Structures and Variables **3-55**

```

                                dd      TimeoutText
                                dd      ConnectionText

KeywordTextLenTable           dd      DIXTextLen
                                dd      TxBelow16TextLen
                                dd      TimeoutTextLen
                                dd      ConnectionTextLen

KeywordProcedureTable         dd      DIXRoutine
                                dd      TxBelow16Routine
                                dd      TimeoutRoutine
                                dd      ConnectionRoutine

;-----
;-----; Define Keywords and related Parameters
;-----
;-----DIXText                db      "DIX"                ; Old style custom
keyword
DIXTextLen                    equ     $ - DIXText
;-----
;-----TxBelow16Text          db      "TB16"
TxBelow16TextLen              equ     ($ - TxBelow16Text) OR T_NUMBER OR
                                T_REQUIRED
                                dd      0                    ; Min value of 0
                                dd      2                    ; Max value of 16
                                dd      TxBelow16Default      ; Default String
                                dd      TxBelow16Valid        ; Valid chars string
                                dd      TxBelow16Prompt        ; Prompt string
TxBelow16Default              db      "0", 0                ; Default to zero
TxBelow16Valid                db      "0..2", 0              ; Only 0, 1 or 2 are
                                valid
TxBelow16Prompt               db      "Enter the number of Tx Buffers "
                                db      "to allocate below 16 Meg", CR, LF
                                db      "(0 = none, 1 = 4 buffers, 2 = 8 buffers) : ", 0
;-----
;-----
TimeoutText                   db      "TO"
TimeoutTextLen                equ     ($ - TimeoutText) OR T_NUMBER
                                dd      0                    ; Min value of 0
                                dd      -1                   ;Max value of 4G
;-----
;-----Connection Text        db      "DEST"
Connection TextLen            equ     ($ - ConnectionText) OR T_HEX_STRING OR
                                T_REQUIRED
                                dd      0                    ;No default
                                dd      ConnectionPrompt      ;Prompt string
ConnectionPrompt              db      "Enter address of node to connect with : ", 0
;-----
;-----

```

```

DIXRoutine          proc
    mov     DIXInUse, 1      ; Set DIX in use flag
    ret
DIXRoutine          endp

TxBelow16Routine    proc
    mov     TxBelow16, eax   ; Save number for later
    ret
TxBelow16Routine    endp

TimeoutRoutine      proc
    mov     TxTimeoutValue, eax ; Save number for later
    ret
TimeoutRoutine      endp

ConnectionRoutine    proc
    mov     eax, [esi+0]      ; Save 6 byte address
    mov     dword ptr DestinationAddress+0, eax
    mov     ax, [esi+4]
    mov     word ptr DestinationAddress+4, ax
    ret
ConnectionRoutine    endp

```

HSM Data Structures and Variables **3-57**

3-58 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

chapter **4** ***MSM/TSM Data Structures and Variables***

Introduction

This chapter describes the data structures, variables, and constants defined by the MSM and TSM. Some of the variables and structures in this chapter are required to control processes and must be initialized, updated, or managed by the driver. Others are made available as optional support for the developer and may be used accordingly.

MSM Equates

- MSMVirtualBoardLink
- MSMStatusFlags
- MSMTxFreeCount
- MSMMaxFrameHeaderSize
- MSMPphysNodeAddress

Data Structures

- Receive Control Blocks (RCBs)
- Transmit Control Blocks (TCBs)
- Event Control Blocks (ECBs)

MSM Global Variables

MSMBitSwapTable

The MSMBitSwapTable is a 256-byte array that can be used to convert noncanonical addresses to canonical, and vice versa. Most drivers will not need to do this, usually just some TokenRing drivers using canonical format addressing modes; and some older FDDI drivers whose hardware doesn't convert noncanonical to canonical addresses.

This global table eliminates the need to have any other bit-swap tables, such as in the HSM. The conversion is done one byte at a time, for example:

```
xor    eax, eax
mov     al,    <byte to be swapped>
mov     al,    MSMBitSwapTable [eax]
```

In this example, if the <byte to be swapped> was 64h, the result in register *al* would be 26h.

MSM Equates

The HSM must access several variables located in the MSM's Data Space. This section describes the MSM defined equates which enable the HSM to access these variables. The equates represent negative offsets which are used in conjunction with EBP, the pointer to the Adapter Data Space.

MSMVirtualBoardLink

The MSM maintains a separate configuration table for each frame type supported by the driver. **MSMVirtualBoardLink** is used to access a list of pointers to the configuration tables.

The list contains 4 pointers for Ethernet, 2 for Token-Ring, and FDDI, and 1 for RX-Net. If a particular frame has not been loaded, the pointer to the corresponding configuration table will be zero. The lists are accessed as follows.

Ethernet	[ebp].MSMVirtualBoardLink + 00h	;ETHERNET 802.2
	[ebp].MSMVirtualBoardLink + 04h	;ETHERNET II
	[ebp].MSMVirtualBoardLink + 08h	;ETHERNET 802.3
	[ebp].MSMVirtualBoardLink + 0Ch	;ETHERNET SNAP
Token-Ring	[ebp].MSMVirtualBoardLink + 00h	;TOKEN 802.2
	[ebp].MSMVirtualBoardLink + 04h	;TOKEN SNAP
FDDI	[ebp].MSMVirtualBoardLink + 00h	;FDDI 802.2
	[ebp].MSMVirtualBoardLink + 04h	;FDDI SNAP
RX-Net		

[ebp].MSMVirtualBoardLink + 00h ;RX-Net

Example

```

mov     ebx, [ebp].MSMVirtualBoardLink+00h    ; Ptr to E_802.2 config table
or      ebx, ebx                              ; Check if valid pointer?
jz      Frame8022NotRegistered                ; Jump if not
mov     eax, [ebx].MLIDSlot                    ; EAX = Our slot number

```

MSMStatusFlags

The MSM maintains a dword variable which provides certain adapter status information. This status information enables the driver to determine if the adapter is shutdown or if the MSM has any packets waiting in its transmit queue. The

MSMStatusFlags equate represents a negative offset which is used in conjunction with EBP, the pointer to the Adapter Data Space, to access the status variable. It is defined in the MSM.INC file as follows:

```

MSMStatusFlags      equ      DriverAdapterStart - (2 * 4)
SHUTDOWN            equ      01h    ; Bit #0 = Shutdown Status
TXQUEUED            equ      02h    ; Bit #1 = Tx Queue Status
POLLING_SUSPENDED   equ      10h    ; Bit #4 = Polling State

```

MSMStatusFlags can be used by the HSM to determine whether the adapter is partially shutdown. If bit #0 is set, the adapter is partially shutdown and must not be serviced. Likewise, the MSM will not call **DriverSend** to transmit a packet if the adapter is partially shutdown.

```

test     [ebp].MSMStatusFlags,SHUTDOWN
jnz      DoNotServiceAdapter

```

The status flags can also be used by polled drivers to determine if polling has been suspended. If bit #4 is set, polling has been suspend by a previous call to **MSMSuspendPolling**.

```

test     [ebp].MSMStatusFlags, POLLING=SUSPENDED
jz       Polling has been suspended

```

The status flags can also be used to determine if the TSM has any send TCBs queued, thus saving a call to **<TSM>GetNextSend**. If bit #1 is set, the TSM has at least one packet queued for transmission.

Note



RX-Net drivers cannot use this test since additional fragments of a split packet are not detected.

MSM/TSM Data Structures and Variables 4-3

```
test    [ebp].MSMStatusFlags,TXQUEUED
jz      NoSendsQueued
```

Example

```
DriverISR  proc
.
.
.
TransmitComplete:          ; EBP=Ptr to Adapter Data Space

inc    [ebp].MSMTxFreeCount ; Free adapter's transmit resource
mov    [ebp].TxInProgress, 0 ; Clear transmit in progress flag

;*** Transmit Next Packet ***

test    [ebp].MSMStatusFlags,TXQUEUED; Anything in send queue?
jz      NoSendsQueued          ; Jump if nothing to send
call    <TSM>GetNextSend       ; Otherwise get the next TCB from
call    DriverSend             ; the queue and send it
.
.
.
MSMServiceEventsAndReturn

DriverISR  endp
```

MSMTxFreeCount

During initialization, the HSM must specify the number of hardware resources available on the adapter for handling pending packet transmissions. The MSM uses this value to determine if the adapter is ready to accept another packet for transmission. The count is also used to determine how many TCB structures the MSM will allocate. The *MSMTxFreeCount* equate represents a negative offset which is used in conjunction with EBP, the pointer to the Adapter Data Space, to access the count. It is defined in the MSM.INC file as follows:

```
MSMTxFreeCount    equ    DriverAdapterStart - (1*4)
```

For example, if the adapter has a second transmit buffer that can accept another packet before the current transmission is complete, the driver must set *MSMTxFreeCount* to a value of 2. Some adapters support hardware queuing. In this case, the count should represent the number of transmissions that the adapter can efficiently process. If the adapter has no additional resources

4-4 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

available other than those used to transmit the current packet, set *MSMTxFreeCount* to 1.

The TSM decrements this count before it calls **DriverSend**. The count is also decremented during a successful call to <TSM>**GetNextSend**. The TSM assumes that the adapter is not ready for another packet if this count reaches zero.

The driver is responsible for incrementing the count each time one of the adapter's transmit resources becomes available. The count must be incremented not only when the adapter successfully completes a transmission, but also when a transmission is aborted due to timeout errors, maximum retry errors, ...etc.

Example

```

DriverInit      proc          ; EBP = Ptr to Adapter Data Space
    .
    .
    .
    mov         [ebp].MSMTxFreeCount,2      ; Adapter has 2 transmit resources
    .
    .
    .
DriverInit      endp

DriverISR       proc
    .
    .
    .
TransmitComplete:
    inc         [ebp].MSMTxFreeCount        ; Free adapter's transmit resource
    .
    .
    .
DriverISR       endp

```

MSM/TSM Data Structures and Variables 4-5

MSMPriorityTxFreeCount

During initialization, the HSM must specify the number of hardware resources available on the adapter for handling priority packet transmissions.

MSMMaxFrameHeaderSize

The <TSM>**GetRCB** procedure, which may be used during packet reception, employs a LOOKAHEAD process in which the header information of a received packet is transferred into a buffer and previewed by the TSM. This way, the TSM can first verify that it wants the packet before the entire packet is read from the adapter.

The TSM sets the *MSMMaxFrameHeaderSize* value to the number of bytes the driver must transfer to that LOOKAHEAD buffer. Its value is equal to the *MLIDLookAheadSize* value from the configuration table plus the maximum media header size. It can be up to 128 bytes, the maximum *MLIDLookAheadSize*, plus the maximum media header size.

For example:

```
MLIDLookAheadSize = 128
Ethernet Maximum Media Header Size = 22
MSMMaxFrameHeaderSize = 128 + 22 = 150
```

To access the size, the *MSMMaxFrameHeaderSize* equate is used in conjunction with EBP, the pointer to the Adapter Data Space.

```
mov    ecx, [ebp].MSMMaxFrameHeaderSize
```

Your driver must read the size each time before calling <TSM>**GetRCB** since it can change dynamically. The driver may optionally implement the **DriverRxLookAheadChange** routine to allow HSMs for intelligent adapters to be informed when the size changes rather than constantly checking.

For more information on the LookAhead process, see the “Packet Reception” section in Chapter 5 and the <TSM>GetRCB procedure in Chapter 6. Refer to the *DriverRxLookAheadChangePtr* field description of the DriverParameterBlock in Chapter 3 for more information on implementing this control procedure for intelligent adapters.

Example

```
DriverISR      proc          ; ebp = Ptr to Adapter Data Space
                .
                .
                .

ReceiveEvent:
    mov     ecx, [ebp].MSMMaxFrameHeaderSize
    lea     edi, [ebp].LookAheadBuffer
    rep     insb
    lea     esi, [ebp].LookAheadBuffer; Ptr to LookAhead Buffer
    mov     ecx, ReceivePacketSize    ; Get Packet Size
    call    <TSM>GetRCB               ; Get an RCB
    jnz     PacketNotAccepted
    .
    .
    .
```

MSMPhysNodeAddress

The *MSMPhysNodeAddress* equate is a negative offset that is used in conjunction with EBP, the pointer to the Adapter Data Space, to access the physical layer format of the node address. It is defined in the MSM.INC file as follows:

```
MSMPhysNodeAddress equ DriverAdapterStart -(16*4)
```

If bit 15 of the *MLIDModeFlags* is set, the driver must use **MSMPhysNodeAddress** instead of the configuration table *MLIDNodeAddress* to obtain the physical layer format of the node address. The MSM sets the *MSMPhysNodeAddress* value when the driver's initialization routine calls **MSMRegisterMLID**.

For additional information, refer to the configuration table *MLIDNodeAddress* and *MLIDModeFlags* descriptions in Chapter 3 and the canonical/noncanonical format discussion in the *Canonical and Noncanonical Addressing* supplement.

Example

```
DriverReset proc ; ebp = Ptr to Adapter Data Space
    lea esi,[ebp].MSMPhysNodeAddress
    lea edi,[ebp].OpenAdapterNode
    movsd
    movsw
    .
    .
    .
DriverReset endp
```


Data Structures

The structures used to transfer data between the layers of the ODI model are called Event Control Blocks (ECBs). The MSM defines two specific forms of the ECB structure.

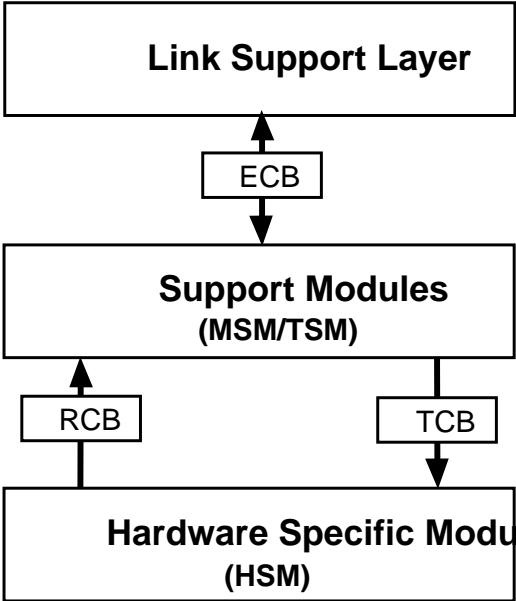
- Receive Control Blocks (RCBs)
- Transmit Control Blocks (TCBs)

These streamlined forms of the general ECB structure are provided by the MSM to simplify driver development. Only the fields relevant to the specific packet transaction in progress are visible to the driver.

The following section describes the RCB and TCB structures. The HSM must refer to these structures during packet reception and transmission. The relationship of these MSM structures with the general ECB structure is also discussed.

Specific reception and transmission methods and related MSM/TSM support routines are described in Chapter 5.

Figure 4-1 Packet Transfer in the MSM/ODI Model



4-10 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Receive Control Blocks

Receive Control Blocks are the structures used to transfer data from the HSM to the TSM.

Usually, when the adapter receives a packet, the HSM obtains a Receive Control Block from the TSM and copies the packet into the RCB's data fragment buffer(s). The RCB is passed back to the TSM where it is processed and transferred to the Link Support Layer. The Link Support Layer then directs it to the proper protocol stack.

On a server, there will normally be only one fragment buffer into which the received data must be copied, therefore drivers should be optimized for one fragment receives. However, the driver's receive routine should be designed to handle multiple fragment buffers if possible. Bit 10 of the *MLIDModeFlags* field in the configuration table must be set if the driver can handle fragmented receive buffers.

The following support routines are available to obtain RCBs.

- **MSMAllocateRCB**
- **MSMAllocateMultipleRCBs**
- **<TSM>GetRCB**
- **<TSM>ProcessGetRCB**
- **<TSM>FastProcessGetRCB**

The **<TSM>GetRCB** routine provides fragmented RCBs. Drivers that cannot handle fragmented receive buffers must use **MSMAllocateRCB**, **<TSM>ProcessGetRCB**, or **<TSM>FastProcessGetRCB** to obtain RCBs. Chapter 5 describes specific reception methods and illustrates the use of these support routines.

The following section describes the RCB structures and fields. The structures are defined in the MSM.INC file.

MSM/TSM Data Structures and Variables **4-11**

Fragmented RCB

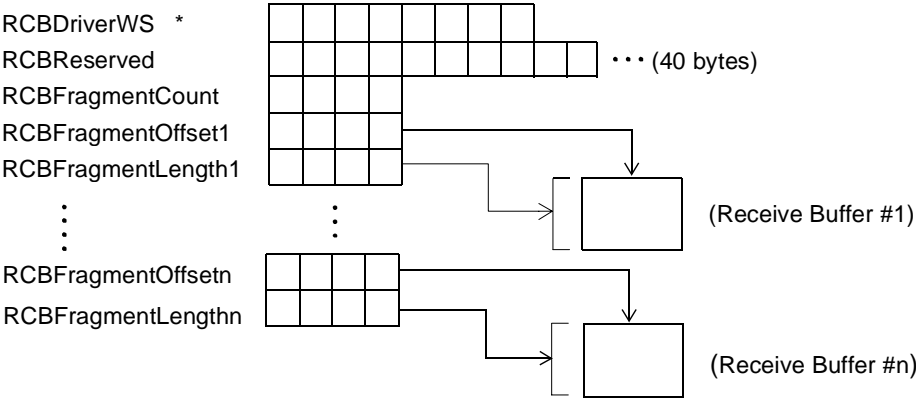
Figure 4-2 Fragmented Receive Control Block

```
RCBStructure      struc

    RCBDriverWS    *          db 8      dup (0)      ; Driver Workspace
    RCBReserved    db 40      dup (0)      ; Reserved for MSM use
    RCBFragmentCount dd ?          ; Number of Fragments
    RCBFragmentOffset1 dd ?      ; Pointer to the 1st Fragment Buffer
    RCBFragmentLength1 dd ?      ; Length of the 1st Fragment Buffer

RCBStructure      ends

*** Additional Fragment Descriptors
***
.
.
.
;; RCBFragmentOffsetn dd ?          ; Pointer to the nth Fragment Buffer
;; RCBFragmentLengthn dd ?        ; Length of the nth Fragment Buffer
```



* RCBDriverWS cannot be used by RX-Net drivers.

Table 4.1 Fragmented RCB Field Descriptions

Offset	Name	Bytes	Description
00h	RCBDriverWS	8	The HSM may use this field for any purpose as long as it controls the RCB. (RX-Net drivers cannot use this field)
08h	RCBReserved	40	This field must not be modified by the HSM. It contains status indicators, protocol information, and additional data maintained by the MSM and Link Support Layer.
30h	RCBFragmentCount	4	This field contains the number of data fragment descriptors to follow. Each descriptor consists of a pointer to a fragment buffer and the size of that buffer. The HSM will copy the received packet into these buffers.
34h	RCBFragmentOffset1	4	Pointer to the 1st fragment buffer.
38h	RCBFragmentLength1	4	Length of the 1st fragment buffer.
.	.	.	
.	.	.	
.	.	.	
??h	RCBFragmentOffsetnRCB	4	Immediately following the RCB in memory are additional fragment descriptors.
??h	FragmentLengthn	4	

Non-Fragmented RCB

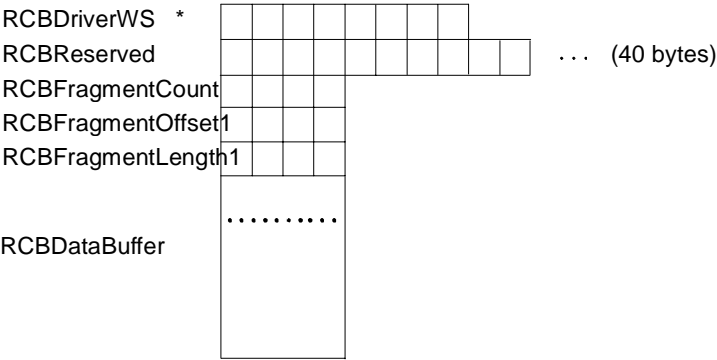
Figure 4-3 Non-Fragmented Receive Control Block

```
RCBStructure      struc

    RCBDriverWS    *          db 8      dup (0)          ; Driver Workspace
    RCBReserved    db 40      dup (0)          ; Reserved for MSM
    RCBFragmentCount dd 1
    RCBFragmentOffset1 dd ?
    RCBFragmentLength1 dd ?

RCBStructure      ends

;; RCBDataBuffer      equ      RCBFragmentLength1 + 4 ; Buffer for Packet
```



* RCBDriverWS cannot be used by RX-Net drivers.

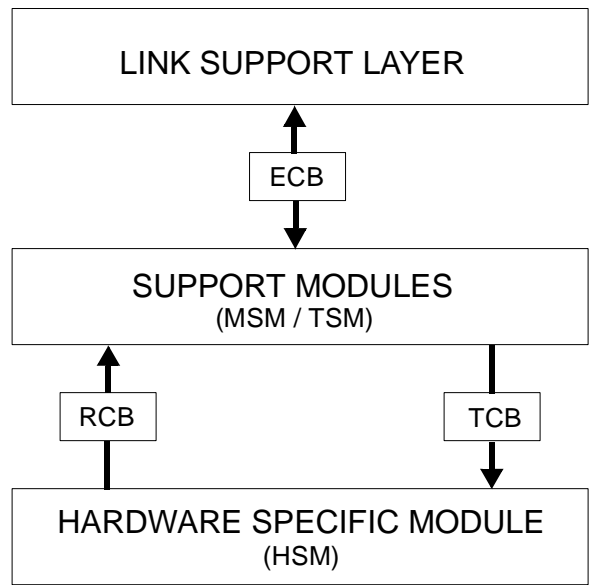
Table 4.2 Non-Fragmented RCB Field Descriptions

Offset	Name	Bytes	Description
00h	RCBDriverWS	8	The HSM may use this field for any purpose as long as it controls the RCB. (RX-Net drivers cannot use this field)
08h	RCBReserved	40	This field must not be modified by the HSM. It contains status indicators, protocol information, and additional data maintained by the MSM and Link Support Layer.
30h	RCBFragmentCount	4	This field contains the number of data fragment descriptors to follow. It will always be 1 for non-fragmented receives.
34h	RCBFragmentOffset1	4	The HSM must NOT use this field. The TSM determines this value <i>after</i> the HSM returns the RCB for processing. (It will contain a pointer to the “data” portion of the received packet in the RCBDataBuffer.)
38h	RCBFragmentLength1	4	The HSM must NOT use this field. The TSM determines this value <i>after</i> the HSM returns the RCB for processing. (It will contain the length of the “data” portion of the received packet in the RCBDataBuffer.)
3Ch	RCBDataBuffer	?	Immediately following the RCB in memory is a buffer for the received packet. The HSM copies the received packet into this buffer. For some frame types this data buffer contains MAC layer headers. (Refer to the MSMAllocateRCB routine for information on using a non-fragmented RCB)

Transmit Control Blocks

Transmit Control Blocks are the structures used to transfer data from the TSM to the HSM.

Figure 4-4 Packet Transfer in the MSM/ODI Model



When sending a packet, a protocol stack assembles a list of fragment pointers in a transmit ECB and passes it to the Link Support Layer. The ECB is then transferred to the TSM where the information is processed and a TCB is constructed. The TCB structure consists of the assembled packet header and data fragment information. The TSM directs the TCB to the appropriate driver which collects the header and packet fragments and transmits the packet.

The following section describes the TCB structures used during packet transmission. The structures are defined in the MSM.INC file.

4-16 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

TCB for Ethernet, Token-Ring, and FDDI

Figure 4-5 Ethernet, Token-Ring and FDDI Transmit Control Block

```
TCBStructure      struc

TCBDriverWS      dd      3 dup (0);          ; Driver Workspace
TCBDataLen       dd      ?                  ; Total Fragment + Media
Header Length
TCBFragStrucPtr   dd      ?                  ; Pointer to Fragment
Structure
TCBMediaHeaderLen dd      ?                  ; Length of Media Header

TCBStructure      ends

;;  TCBMediaHeader      equ  TCBMediaHeaderLen + 4          ; Media Header Buffer
```

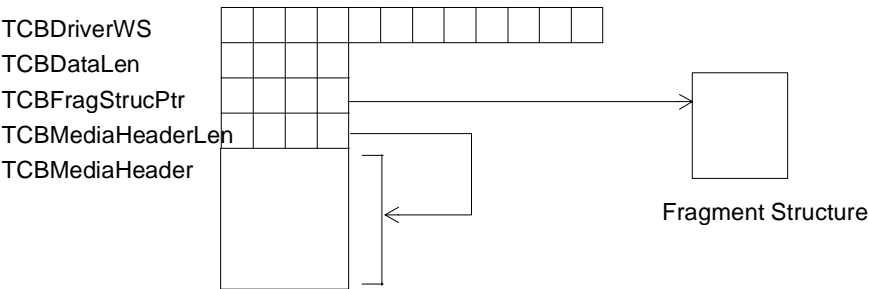


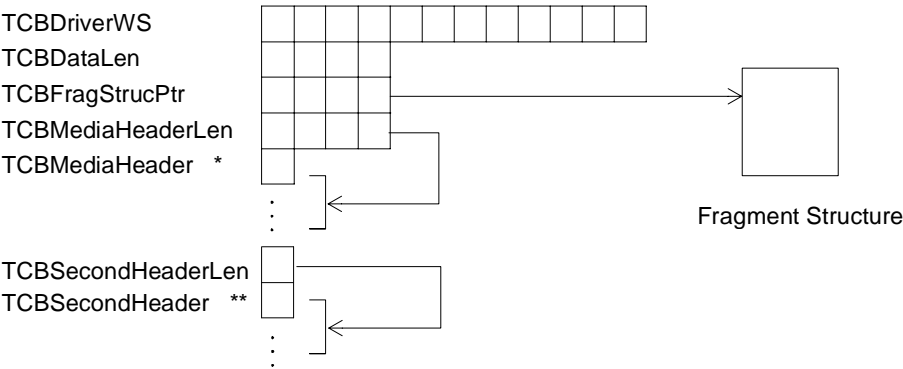
Table 4.3 TCB Field Descriptions

Offset	Name	Bytes	Description
00h	TCBDriverWS	12	The HSM may use this field for any purpose as long as it controls the TCB.
0Ch	TCBDataLen	4	This field contains the length of the packet described by the data fragments plus the media header. This value will never be 0.
10h	TCBFragStrucPtr	4	This field contains a pointer to a list of fragments defined by the <i>FragmentStructure</i> (described following the TCB section).
14h	TCBMediaHeaderLen	4	This field contains the length of the Media Header that immediately follows the TCB in memory. This value may be odd, even, or zero. A value of zero indicates a raw send. If the HSM is handed a raw send, the originating protocol stack has already included the media header in the first data fragment.
18h	TCBMediaHeader	?	Immediately following the TCB in memory is a buffer containing the Media Header that was assembled by the MSM.

TCB for RX-Net

Figure 4-6 Rx-NET Transmit Control Block

TCBStructure	struc			
TCBDriverWS	dd	3 dup (0)		; Driver Workspace
TCBDataLen	dd	?		; Total Fragment + Media Header Length
TCBFragStrucPtr	dd	?		; Pointer to Fragment Structure
TCBMediaHeaderLen	dd	?		; Length of First Media Header
TCBStructure	ends			
::	TCBMediaHeader	db	3 or 4 dup (?)	; First Media Header *
::	TCBSecondHeaderLen	db	?	; Length of Second Media Header
::	TCBSecondHeader	db	4 or 8 dup (?)	; Second Media Header **




* TCBMediaHeader is 3 bytes for Short Packets, and 4 bytes for Long or Exception Packets.

** TCBSecondHeader is 4 bytes for Short or Long Packets, and 8 bytes for Exception Packets.

Table 4.4 TCB Field Descriptions (RX-Net)

Offset	Name	Bytes	Description
00h	TCBDriverWS	12	This field is used by the MSM to link the TCBs
0Ch	TCBDataLen	4	This field contains the length of the packet described by the data fragments plus the media header. This value will never be 0.
10h	TCBFragStrucPtr	4	This field contains a pointer to a list of fragments defined by the <i>FragmentStructure</i> (described following this section).
14h	TCBMediaHeaderLen	4	This field contains the length of the first media header.
Immediately following the TCB in memory, is a buffer containing the media header information.			
18h	TCBMediaHeader	3 or 4	This field contains the first media header. The header is 3 bytes for Short Packets and 4 bytes for Long or Exception Packets.
?	TCBSecondHeaderLen	1	This field contains the length of the second media header.
?	TCBSecondHeader	4 or 8	This field contains the second media header. The header is 4 bytes for Short or Long Packets, and 8 bytes for Exception Packets.

Note

Several fields of the above table reference the different types of RX-Net packets. The following diagram shows the three RX-Net packet formats. A full description of each is included in the *ODI Supplement: Frame Types and Protocol IDs*.

RX-Net Packet Format

Short Packet	Long Packet	Exception Packet
Source Address	Source Address	Source Address
Destination Address	Destination Address	Destination Address
Byte Offset	Long Packet Flag	Long Packet Flag
Unused (size varies)	Byte Offset	Byte Offset
Protocol Type	Unused (size varies)	Unused (size varies)
Split Flag	Protocol Type	Pad 1: Protocol Type
Packet Sequence Number (2 bytes)	Split Flag	Pad 2: Split Flag
Data (0 - 249 bytes)	Packet Sequence Number (2 bytes)	Pad 3: FFh
	Data (253 - 504 bytes)	Pad 4: FFh
		Protocol Type
		Split Flag
		Packet Sequence Number (2 bytes)
		Data (250 - 252 bytes)

Fragment Structure

The following section describes the format of the fragment structure pointed to by the *TCBFragStrucPtr* field of the Transmit Control Block.

Figure 4-7 TCB Fragment Structure

FragmentCount	dd	?	; Number of Fragments
FragmentOffset1	dd	?	; Pointer to the 1st Data Fragment
FragmentLength1	dd	?	; Length of the 1st Data Fragment
FragmentOffsetn	dd	?	; Pointer to the nth Data Fragment
FragmentLengthn	dd	?	; Length of the nth Data Fragment

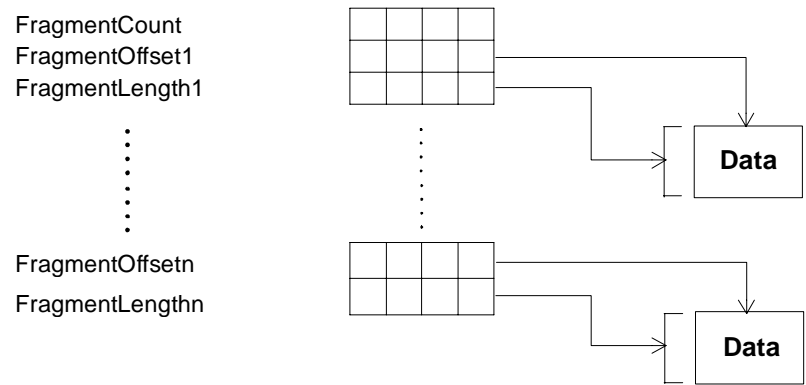


Table 4.5 TCB Fragment Structure

Offset	Name	Bytes	Description
00h	FragmentCount	4	This field contains the number of data fragment descriptors to follow. Each descriptor consists of a pointer to a fragment buffer and the size of that buffer. The HSM collects the data from these buffers when forming the packet for transmission.
04h	FragmentOffset1	4	Pointer to the buffer containing the first data fragment.
08h	FragmentLength1	4	Length of the buffer pointed to by <i>FragmentOffset1</i> .
.	.	.	
.	.	.	
.	.	.	
	FragmentOffsetn		(These fields contain additional fragment descriptors)
	FragmentLengthn		

Event Control Blocks

This section defines the general Event Control Block (ECB) structure and illustrates its relationship to the RCB and TCB. This section does not apply to most drivers written with the MSM / TSM interface.

Drivers written using the MSM / TSM interface typically interact with RCBs and TCBs during packet transactions as shown in the figure below. However, some drivers may need to bypass these MSM provided structures in order to work directly with the underlying general ECB structure. This is typically the case for intelligent adapters that are designed to be *ECB aware*.

An *ECB aware* adapter/driver will completely fill in and manage all fields of the ECB during packet transactions. This shifts much of the overhead involved in packet reception and transmission to the adapter giving the processor more time to perform other tasks.


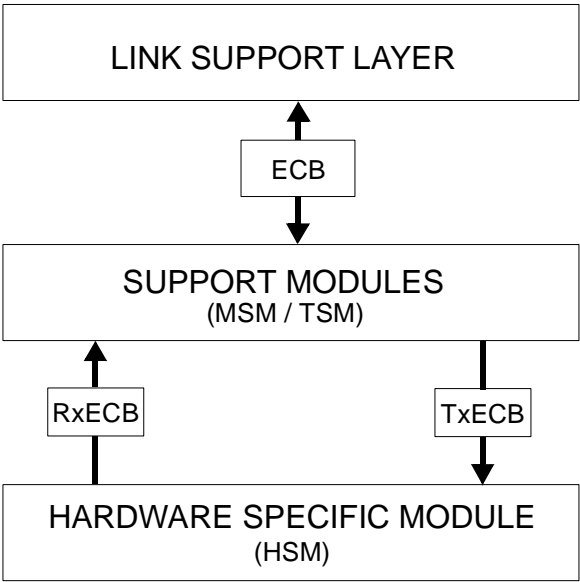
Note  This section only applies to *ECB aware* adapters/drivers.

Figure 4-8 Packet Transfer in the MSM/ODI Model



4-24 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

The format of the ECB structure is shown below. The same structure is used for both receiving and transmitting packets.

Figure 4-9 Event Control Block

ECBStructure	struc			
Link	dd	?		; Forward Link used for Queuing ECBs
BLink	dd	?		; Backward Link used for Queuing ECBs
Status	dw	?		; Current ECB Status
ESRAddress	dd	?		; Event Service Handler
LogicalID	dw	?		; Protocol Logical ID
* ProtocolID	db	6 dup (?)		; Protocol ID **
* BoardNumber	dd	?		; Logical Board # from Configuration Table
* ImmediateAddress	db	6 dup (?)		; Rx...Source Addr / Tx...Destination Addr
* DriverWorkSpace	dd	?		; Driver Workspace / Dest and Frame Type **
ProtocolWorkSpace	db	8 dup (?)		; Protocol Stack Workspace
* PacketLength	dd	?		; Length of the Packet Data
* FragmentCount	dd	?		; Number of Fragments
* FragmentOffset1	dd	?		; Pointer to the 1st Fragment Buffer
* FragmentLength1	dd	?		; Length of the 1st Fragment Buffer
ECBStructure	ends			
.				; Additional Fields follow for both
.				; Receive and Transmit ECBs
.				

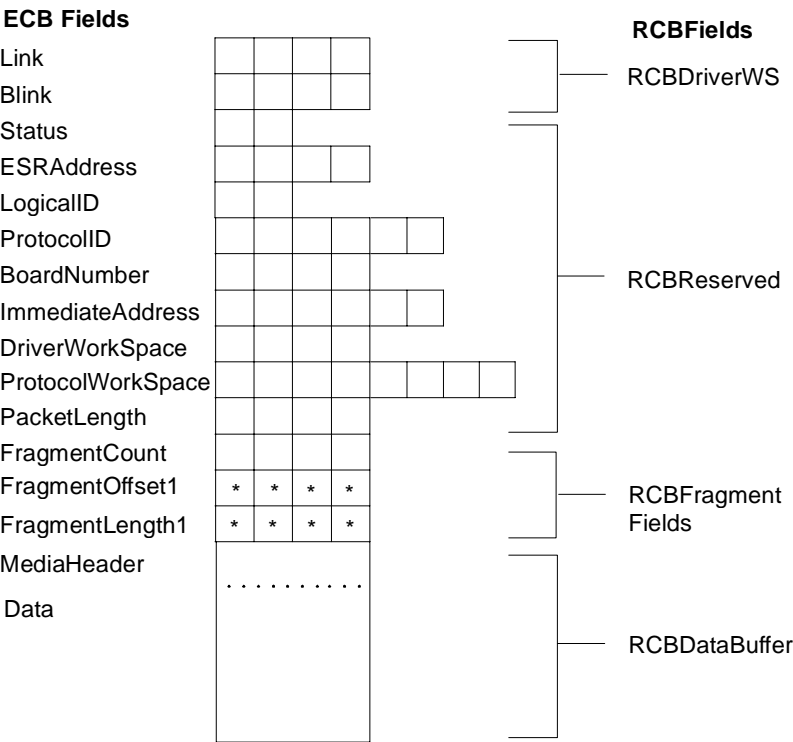
* During packet reception, these fields must be filled in by the *ECB Aware* Adapter/Driver before passing the ECB to the upper layers. During packet transmission, all fields are filled in by the upper layers before passing the ECB to the driver.

** 802.2 frame types require special handling of the *ProtocolID* and *DriverWorkSpace* fields in the ECB during packet reception and transmission (refer to the *ODI Supplement: Frame Types and Protocol IDs*).

Receive ECBs vs RCBs

The general Receive ECB and the MSM's RCB essentially form a union. That is, both structures occupy the same memory space.

Figure 4-10 ECBs vs RCBs



The ECB fields that correspond to *RCBReserved* are normally managed by the TSM. However, if an adapter is ECB aware, it can simply treat the structure as an ECB and take over the management of these fields.

Drivers written for ECB aware adapters must obtain control blocks by calling **MSMAllocateRCB**. This routine allows the driver to preallocate RCBs without the MSM initializing the fields. When a packet is received, the adapter copies it into the *RCBDataBuffer*, fills in the required fields (see Figure 4.9), and returns the structure using either the **<TSM>RcvCompleteStatus / MSMServiceEvents** combination or the function **<TSM>FastRcvCompleteStatus**. (These routines are described in Chapter 6).

Transmit ECBs vs TCBs

The general Transmit ECB and the TSM's TCB are totally separate structures. The *TCBFragStrucPtr* field of the TCB, however, points to the *FragmentCount* field of the ECB. Knowing this, it is possible to work directly with the underlying ECB by using both negative and positive offsets from this pointer.

The MSM provides another more efficient way for ECB Aware adapters to work directly with ECBs. By setting the **DriverSendWantsECBs** variable of the *DriverParameterBlock* to any nonzero value (see Chapter 3), the HSM's **DriverSend** routine will be given ECBs rather than TCBs for packet transmission. The HSM will then be responsible for building the proper media header depending on the board number.

Figure 4-11 Transmit ECBs vs TCBs

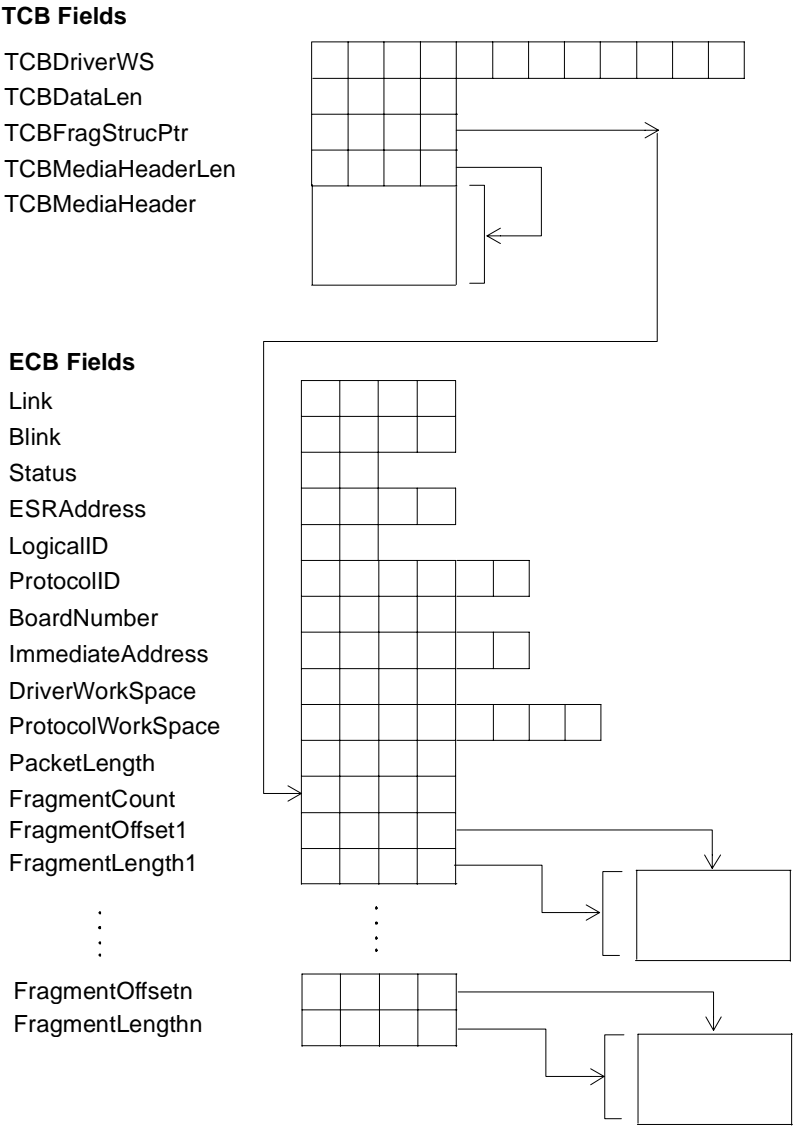


Table 4.6 ECB Field Descriptions

Offset	Name	Bytes	Description
00h	Link	4	This field contains a forward link to another ECB. The LSL uses this field for queuing ECBs. The HSM may use this field for any purpose as long as it controls the ECB.
04h	BLink	4	This field contains a backward link to another ECB. The LSL uses this field for queuing ECBs. The HSM may use this field for any purpose as long as it controls the ECB. However, if the HSM uses DriverSendWantsECBs to get ECBs instead of TCBs, it must <u>not</u> modify the transmit ECB's <i>BLink</i> field.
08h	Status	2	This field must not be modified by the HSM. The LSL uses the <i>Status</i> field to indicate the current state of the ECB. (i.e., currently unused, queued for sending, etc.).
0Ah	ESRAddress	4	This field must not be modified by the HSM. In receive ECBs, the LSL places a pointer to the target protocol stack's receive handler in this field and then queues the receive ECB on a hold queue. Later, the LSL polls the hold queue and routes the ECB to the proper protocol stack by calling the address in this field.
0Eh	LogicalID	2	This field must not be modified by the HSM. When a protocol stack registers with the LSL, it is assigned a logical number (0...15). This field contains that logical number or, if the packet is a raw send, the field contains the value FFFFh. On 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 the 802.2 frame type, on sends, this field also contains the 802.2 frame type information (Type I or II) required to build the media header. (See the <i>Frame Types and Protocol IDs supplement</i> for an explanation of 802.2 Type I and Type II use of this field). On receives, this field always contains the DSAP value of the 802.2 header.

MSM/TSM Data Structures and Variables **4-29**

Table 4.6 ECB Field Descriptions *continued*

Offset	Name	Bytes	Description
16h	BoardNumber	4	When a driver registers with the LSL, it is given a logical board number. The <i>MLIDBoardNumber</i> field of the configuration table contains that number (see Chapter 3). Logical board 0 is used internally in the operating system. Drivers are assigned logical board numbers 1 through 255. On receives, the HSM must fill in this field to indicate which logical board received the packet. On sends, a protocol stack fills in this field to indicate the target logical board.
1Ah	ImmediateAddress	6	<p>On receives, the immediate address represents either the packet's source node address or the address of the last router that passed the packet if the packet was routed from another network. On sends, the immediate address represents either the destination node address or the destination router address.</p> <p>The address is stored in High-Low order. If the node address is less than six bytes, the most significant byte(s) must be padded with 0.</p> <p>The MSM fills in this field on receives. Addresses passed to the upper layers may be in canonical or noncanonical format depending upon whether the driver bit-swaps MSB format addresses. The stack fills in this field on sends. All addresses passed down to the MLID are in canonical format if the driver is configured to be LSB. (Refer to MSMPhysNodeAddress description.)</p>

Table 4.6 ECB Field Descriptions *continued*

Offset	Name	Bytes	Description
20h	DriverWorkspace	4	<p>The HSM can use this field for any purpose. The LSL will not modify the field. However, before passing a completed receive ECB to the LSL, fill in the first byte of the field (offset 20h) with the destination address type of the received packet:</p> <p>01h = Multicast 03h = Broadcast 04h = Remote Unicast 08h = Remote Multicast 10h = No Source Route 20h = Error Packet 80h = Direct Unicast</p> <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> <p>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> <p>See the <i>Frame Types and Protocol IDs supplement</i> for an explanation of 802.2 Type I and Type II.</p>
24h	ProtocolWorkspace	8	This field is reserved for use by the protocol stack.
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 or SAP headers).</p> <p>On receives, this value is equal to the <i>FragmentLength1</i> (length may be 0). The HSM for ECB aware adapters must fill in this field.</p> <p>On sends, this value may be zero. The protocol stack fills in this field.</p>
30h	FragmentCount	4	<p>This field contains the number of data fragment descriptors to follow. Each descriptor consists of a pointer to a fragment buffer and the size of that buffer.</p> <p>On receives, the fragment count is always between 1 and 16.</p> <p>On sends, the fragment count is always between 1 and 16.</p>

MSM/TSM Data Structures and Variables 4-31

Table 4.6 ECB Field Descriptions *continued*

Offset	Name	Bytes	Description
34h	FragmentOffset1	4	On receives, immediately following the ECB in memory is a buffer where the HSM copies the received packet. After the packet is copied into the buffer the HSM must set the <i>FragmentOffset</i> to point around any media headers to the data portion of the packet. The HSM must also set the <i>FragmentLength</i> field to the total length of the data portion of the packet (see Figure 4.10). On sends, the <i>FragmentOffset</i> field points to the first fragment buffer containing packet data. The <i>FragmentLength</i> field specifies the length of that buffer. This value can be zero. Immediately following the ECB in memory there may be additional fragment descriptors. The HSM collects the data from these fragment buffers to form the packet for transmission (see Figure 4.11).
38h	FragmentLength1	4	
On receives, the memory immediately following the ECB contains:			
3Ch	MediaHeader	varies	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 media splits the data of a packet and transmits it in more than one frame (for example RX-Net).
??h	Data	varies	Immediately following the <i>MediaHeader</i> is the data portion of the packet.
On sends, the memory immediately following the ECB contains:			
3Ch	FragmentOffset2	4	These fields contain additional fragment descriptors when the <i>FragmentCount</i> field is greater than 1.
40h	FragmentLength2	4	
.	.	.	
.	.	.	
.	.	.	

chapter **5** *HSM Procedures*

Introduction

This chapter describes the routines that are the primary components of the Hardware Specific Module (HSM).

Initialization and Removal

- DriverInit (required)
- DriverRemove (required)

Board Service

- DriverISR
 - DriverPoll
- (only one of the above is required)
- DriverISR2 (optional)

Packet Transmission

- DriverSend (required)
- DriverPriorityQueueSupport (optional)

Multi-Operating System Support

- DriverEnableInterrupt (required)
- DriverDisableInterrupt (required)
- DriverDisableInterrupt2 (required if DriverISR2 is used)

Control Procedures

- DriverReset (required)
- DriverShutdown (required)
- DriverMulticastChange (required except for RX-Net)
- DriverPromiscuousChange (recommended)
- DriverStatisticsChange (optional)
- DriverRxLookAheadChange (optional)
- DriverManagement (optional)

Timeout Detection

- TimerProcedure (optional)
- DriverAESCAllBack (optional)
- DriverINTCallBack (optional)
- DriverTxTimeout (RX-Net Drivers only)

Every driver must provide the required procedures in order to function properly. The recommended procedures must be implemented if the hardware supports that function. The optional procedures are available if the adapter or driver requires the functionality. The HSM indicates routines not supported by placing a zero in the corresponding fields of the DriverParameterBlock.

All procedures described on the following pages are near calls from the MSM and TSM. The pseudocode shown is intended to illustrate a general flow of events and does not necessarily describe optimized code.

Initialization

The HSM's **DriverInit** routine controls the complete initialization process, although specific tasks performed during initialization are handled by MSM or TSM routines. The initialization tasks include:

- Allocate the Frame and Adapter Data Space
- Process custom command-line keywords and custom firmware
- Parse the standard LOAD command-line options
- Register hardware options
- Initialize the adapter hardware
- Register the driver with the Link Support Layer

This section explains how the initialization tasks are divided between the HSM and the support modules. Following the discussion is pseudocode for a **DriverInit** routine.

DriverInit

When the NetWare OS receives the command to load the driver, it calls the **DriverInit** routine (specified as the “start” routine in HSM's linker definition file). **DriverInit** must preserve EBP, EBX, ESI, and EDI on the stack, and set the *DriverStackPointer* field of the DriverParameterBlock to the value of ESP. The HSM then registers with the MSM and TSM interface as described in the next section.

Register with the MSM / TSM

DriverInit calls the <TSM>**RegisterHSM** routine with ESI pointing to the DriverParameterBlock. The TSM passes the driver's parameter block pointer along with its own to the MSM.

The MSM makes a local copy of both parameter blocks and processes the information passed on the stack from the operating system. If the HSM has custom firmware, the MSM loads the firmware and initializes the *DriverFirmwareSize* and *DriverFirmwareBuffer* variables as described in Chapter 3.

The MSM allocates memory for the Frame Data Space and creates a copy of the driver's configuration table template in that area. If the *MLIDCardName* and *MLIDMajorVersion* fields of the configuration table are initialized to zero, the MSM fills in these fields and the *MLIDMinorVersion* field using information derived from the linker definition file. If the HSM has placed nonzero values in the card name and major version fields, these fields are not modified.

Finally, the MSM sets the *MLIDMaximumSize* field of the configuration table to the LSL's maximum packet size and returns to *DriverInit*.

- If the MSM was unsuccessful in its initialization tasks, it returns with EAX pointing to an error message. *DriverInit* must print the message using **MSMPrintString** and return to the operating system with EAX set to a nonzero value.
- If the MSM is successful, it returns with EAX set to zero and EBX pointing to the driver's configuration table in the Frame Data Space. The HSM must now gather the hardware option information needed for the configuration table and call the MSM to parse the driver parameters entered on the command-line. This process is described in the following section.

Determine Hardware Options

After <TSM>**RegisterHSM** returns successfully, the driver must determine the hardware configuration of the adapter, including the following parameters:

- Hardware Instance Number (HIN) for Micro Channel, PCI, PnP ISA, PC CARD, and EISA adapters
- Base port for programmed IO adapters
- Memory decode addresses for shared RAM adapters
- Interrupt numbers
- DMA channels

In all busses, except for legacy ISA, the driver can get this information directly from the system once the HIN has been identified.

5-4 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

DriverInit must perform the following steps where appropriate for the hardware:

1. If the HSM supports multiple buses, it may call **MSMScanBusInfo** to determine the bus type, or it may call **MSMSearchAdapter** once for each bus type it supports.
2. For all busses except legacy ISA, call **MSMSearchAdapter** to search for the adapter ID. Any hardware instances that are found must be recorded in the *IOSlot* option list of the *AdapterOptionDefinitionStructure*. This structure is described in Chapter 7 under the **MSMParseDriverParameters** routine.

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 HSM calls **MSMParseDriverParameters** to determine the hardware configuration options or the HIN specified on the load command-line, and to query the operator for any required parameters which were not specified.

MSMParseDriverParameters 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 either from the command-line or from the console operator. The table below shows the correspondence between the load options and configuration table fields. The standard load command options are described in Appendix A. An example load command is shown here:

```
load <driver> frame=ethernet_802.3 port=300 int=3
```

Configuration Table Fields	Command-Line
MLIDSlot	load <driver> SLOT=4
MLIDIOPort0	load <driver> PORT=300
MLIDIORange0	load <driver> PORT=300:A
MLIDIOPort1	load <driver> PORT1=700
MLIDIORange1	load <driver> PORT1=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

MSMParseDriverParameters also processes any custom command-line keywords defined by the *DriverKeyword* variables in the *DriverParameterBlock*. (see also **MSMParseCustomKeywords**)

On return from **MSMParseDriverParameters**, the I/O portion of the logical board's configuration table in the Frame Data Space has been filled in with the parsed values.

4. For all buses except legacy ISA, the configuration table now contains the selected adapter HIN. The HSM can now use **MSMGetCardConfig** to determine the configuration.

When all needed information has been obtained for the configuration table, **DriverInit** calls **MSMRegisterHardwareOptions** which is described in the next section.

5-6 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)



Note

If the driver must access shared memory before registering the hardware options, it must use **MSMReadPhysicalMemory** and **MSMWritePhysicalMemory**.

Register Hardware Options

The HSM calls **MSMRegisterHardwareOptions** to register with the operating system. This routine reports to the HSM whether a new adapter or a new frame format for an existing adapter is being loaded. If a new adapter is being registered, the MSM allocates the Adapter Data Space and copies the driver's **AdapterDataSpaceTemplate** to that area. This routine also notifies the HSM of any conflicts with existing hardware in the system.

There are four possible conditions that the HSM must handle on return from **MSMRegisterHardwareOptions**:

- If $EAX = 0$, a new adapter was successfully registered and the HSM must proceed with the hardware initialization (EBP now contains a pointer to the Adapter Data Space).
- If $EAX = 1$, a new frame type for an existing adapter was successfully registered and initialization is essentially complete.
- If $EAX = 2$, a new channel for an existing multichannel adapter was successfully registered. The driver (and MSM) typically treat the registering of a new channel as a new adapter.
- If $EAX > 2$, the MSM was unable to register the hardware options and EAX points to an error message. *DriverInit* must print the error message using **MSMPrintString** and return to the operating system with EAX set to a nonzero value.

Initialize the Adapter

At this point the HSM initializes the adapter hardware. This consists of all setup appropriate for the hardware and might also include RAM and other hardware tests. The **MSMResetMLID** routine could be called to handle part of this procedure.



Note



Note

It is important that **DriverInit** sets up the correct number of transmit buffers (the maximum number of simultaneous sends allowed by the hardware) by placing an appropriate value in *MSMTxFreeCount*. A description of this variable is in

HSM Procedures 5-7

Note



Chapter 4 and information about its use is in the packet transmission section of this chapter.

If an error occurs during the hardware initialization, **DriverInit** must print an appropriate error message, call **MSMReturnDriverResources**, and return to the operating system with EAX set to a nonzero value. If the hardware initializes successfully, the HSM then registers the driver with the LSL.

All HSMs written for hot plug adapters (PCI, PC Card, and others in the future) must use **MSMResetMLID** for all hardware initialization. This is so that the adapter's hardware can be initialized after it is inserted, without having to unload and reload the driver.

Register with the LSL

DriverInit calls the **MSMRegisterMLID** routine to register the driver with the Link Support Layer. Registration consists of the MSM passing the addresses of the MSM's send and control handler procedures, and a pointer to the HSM's configuration table to the LSL. The LSL assigns a logical board number to the adapter and the MSM places it in the configuration table. The MSM automatically registers a logical board with the LSL during **MSMRegisterHardwareOptions** each time a new frame is added for an existing adapter. If an error occurs, the MSM routine returns a pointer to an error message in EAX.

If **MSMRegisterMLID** is successful, the configuration table contains a valid board number. HSMs for intelligent bus master adapters may now pass the board number and frame ID information to the adapter if necessary.

Setup a Board Service Routine

The HSM registers its board service routine, **DriverISR** or **DriverPoll**, by calling either **MSMSetHardwareInterrupt** or **MSMEnablePolling**. The **DriverISR** description later in this chapter provides special instructions on setting up and handling shared interrupts.

Note



Novell requires that 32-bit HSMs call **MSMSetHardwareInterrupt** after the adapter is initialized to prevent an interrupt being received before initialization is complete.

Schedule Timeout Callbacks

If the HSM is running an interrupt driven adapter, it may need to schedule a timer event that checks to see if the board was unable to complete a send. To establish this timer event, drivers have traditionally used **MSMScheduleIntTimeCallBack** or **MSMScheduleAESCCallBack**. These routines schedule periodic calls to the HSM's **DriverCallBack** or **DriverAES** routines. (RX-Net drivers normally use **DriverTxTimeout**, but could use these other two routines.)

New with this specification, the driver can also call **MSMScheduleTimer** to schedule a timer event. This function can be called at process time or interrupt time, and is preferred over **MSMScheduleAESCCallBack** or **MSMScheduleIntTimeCallBack**.

If the adapter is not interrupt driven, the polling procedure can check to see if it failed to complete a send.

Note



It is critical that **MSMRegisterMLID** is called before **MSMScheduleIntTimeCallBack** in order for the driver to work properly with NetWare SMP. ALL drivers must adhere to this.

DriverInit Pseudocode

Processor States	Entry State
	Interrupts are enabled
	Note this routine executes at a process level
	Return State
	EAX zero if successful; nonzero if an error occurred
	Interrupts may be in any state

Pseudocode

```
DriverInit      proc

    push    ebx, ebp, esi, edi
    mov     DriverStackPointer, esp
    lea     esi, DriverParameterBlock
    call    <TSM>RegisterHSM
    jnz     DriverInitError

    *** Determine Hardware Options ***
    (Pseudo Code for Hardware Instance Number (HIN) aware drivers)

    Loop on Call to MSMSearchAdapter while successful.
    The HIN number must be stored in the Slot option field that will be passed into
        MSMParseDriverParameters.

    Call MSMParseDriverParameters. (The selected HIN is put in the MLIDSlot field in the
        Configuration Table)

    Call MSMGetInstanceNumberMapping
    Put the returned BusTag from MSMGetInstanceNumberMapping into
        ConfigTable.MLIDBusTag.

    Call MSMGetCardConfigInfo
    Read the configuration information from the ConfigBuffer.

    *** Register Hardware Options ***

    call     MSMRegisterHardwareOptions
    If an Error occurred
        jmp   DriverInitError
    else if a New Frame was added
        jmp   DriverInitExit
```

5-10 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

else a New Adapter was registered
continue with full initialization

*** Initialize the Adapter ***

If there is not a Node Address override
Read in the Node Address from the board
Copy the Node Address to the Configuration Table

Initialize MSMTxFreeCount

Initialize the Adapter Hardware, etc...
call DriverReset to handle some tasks
If there was an error initializing the hardware
call MSMReturnDriverResources
jmp DriverInitError

*** Register with the LSL ***

call MSMRegisterMLID
jnz DriverInitError

*** Setup a Board Service Routine ***

call MSMSetHardwareInterrupt (or MSMEEnablePolling)
jnz DriverInitError

*** Schedule Timeout Callbacks ***

If Timeout detection is required
esi = pointer to a timer structure
eax = callback interval in ticks
call MSMScheduleTimer (preferred method)
- or -
call MSMScheduleIntTimeCallBack (to enable callbacks
to DriverCallBack)
- or -
call MSMScheduleAESCallBack (to enable callbacks to
DriverAES)
jnz DriverInitError

DriverInitExit:

eax = zero (Initialization was successful)
pop edi, esi, ebp, ebx
return

DriverInitError:

```
esi = eax (Ptr to Error Message)
call    MSMPrintString
eax = nonzero value (Initialization Failed)
pop     edi, esi, ebp, ebx
return
```

DriverInit endp

Packet Reception

This section provides a brief overview of the commonly used reception methods available to the developer.

When the adapter receives a packet, the HSM must copy the packet into an RCB obtained from the TSM. The RCB is passed back to the TSM where it is processed and transferred to the Link Support Layer. The Link Support Layer then directs it to the proper protocol stack.

Reception Methods

The method of packet reception selected is typically dependent on the adapter's data transfer method. The examples on the following pages are intended to illustrate a general flow of events. Refer to the appropriate MSM and TSM support call descriptions for detailed information.

In general, packet reception involves the following steps:

- Obtain a Receive Control Block (RCB) structure from the TSM. RCBs may be allocated before or after a packet is received.
- Copy the packet into the *RCBDataBuffer* or *RCBDataFragments*.
- Return the RCB back to the TSM (RCBs will be placed in the LSL's holding queue until the HSM issues a service events command).
- Use the *MSMServiceEvents* macro to allow the LSL to call the transmit ECB's event service routine.

Programmed I/O and Shared RAM

Option 1. This is the simplest reception method. During development it may be helpful to initially use this method, then implement Option 2 after the HSM is functioning properly. The steps performed for this reception method are outlined below. The **<TSM>ProcessGetRCB** procedure in Chapter 6 provides a detailed description of this process.

DriverISR

Call **MSMAllocateRCB** or **MSMAllocateMultipleRCBs** to get an RCB or multiple RCBs (unless you already have the RCBs you need)

Copy the received packet into the *RCBDataBuffer*.

Call **<TSM>ProcessGetRCB**

The TSM checks the header information and if valid:

- fills in the remainder of the RCB fields
- delivers the RCB to the LSL
- returns a new RCB to the driver

Save the new RCB for next packet received

MSMServiceEvents

Option 2. This method involves using a LookAhead process, in which the frame header information is first confirmed before the entire packet is transferred from the adapter into an RCB. For this reason, Option 2 is recommended over Option 1.

The adapter's data transfer mode determines how the Look-Ahead process is handled. Programmed I/O adapters must transfer *MSMMaxFrameHeaderSize* bytes into a LookAhead buffer allocated for this purpose. If the 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. The **<TSM>GetRCB** procedure in Chapter 6 provides a detailed description of this process.

DriverISR

Setup a LookAhead buffer as described above (*MSMMaxFrame-HeaderSize* bytes)

Call **<TSM>GetRCB** (with a pointer to the LookAhead buffer in ESI)

TSM checks the header information and if valid:

- obtains an RCB
- fills in the RCBReserved fields
- returns a pointer to the RCB in ESI

Copy the remainder of the packet into the RCB fragments

Call **<TSM>RcvComplete**

MSMServiceEvents

DMA and Bus Master

Option 1. This reception method is used for most bus master adapters in which the RCBs are preallocated. The steps performed for this reception method are outlined below. The **<TSM>ProcessGetRCB** procedure in Chapter 6 provides a detailed description of this process.

DriverInit

Use **MSMAllocateRCB** or **MSMAllocateMultipleRCBs** to obtain first RCB(s)
Queue RCB(s) until a packet is received in **DriverISR**.

DriverISR

Copy received packet into the *RCBDataBuffer*.
Call **<TSM>ProcessGetRCB**
The TSM checks the header information and if valid:
fills in the remainder of the RCB fields
delivers the RCB to the LSL
returns a new RCB to the driver

Queue the new RCB until next packet is received **MSMServiceEvents**

Option 2. This method is recommended for intelligent adapters that are designed to be ECB aware. It reduces the load on the server by off-loading code to the adapter. In this way, the adapter's firmware handles most of the reception process. The steps performed for this reception method are outlined below.

DriverInit

Use **MSMAllocateRCB** or **MSMAllocateMultipleRCBs** to obtain first RCB(s).
Queue RCB(s) until a packet is received.

Firmware

Filters the frame header information and if valid, fills in all fields of the ECB as described in Chapter 4.
Generates interrupt when receive is complete (ready).

DriverISR

Call **<TSM>RcvCompleteStatus** to return the completed RCB.
Use **MSMAllocateRCB** or **MSMAllocateMultipleRCBs** to obtain another RCB for the queue.
MSMServiceEvents

HSM Procedures **5-15**

Bus Master Receive Routine - all HSM's:

MSMAllocateRCB, **MSMAllocateMultipleRCBs**, **<TSM>ProcessGetRCB**, and **<TSM>FastProcessGetRCB** will return a logical pointer to the RCB in register ESI. [ESI].RPacketOffset will contain a physical pointer to [ESI].RDataEnvelope, which is where the adapter should begin copying the packet.

If *DriverSupportsPhysFrgs* bit is set, the following applies:

1. All HSM's: All fragment pointers passed back from the ECB returned by **<TSM>GetRCB** will contain physical pointers to contiguous blocks.
2. *DriverNeedsBelow16Meg* set in DriverParameterBlock:
As with previous TSM's, **MSMAllocateRCB**, **<TSM>ProcessGetRCB** and **<TSM>FastProcessGetRCB** will return a contiguous RCB that is guaranteed to be below 16 megabytes.
3. *DriverSendWantsECBs* set in DriverParameterBlock:
MSMAllocateRCB or **MSMAllocateMultipleRCBs** will return a logical pointer to the ECB in register ESI, and a physical pointer to the ECB in EDI. The ECB should be filled in as normal, except before calling **<TSM>FastRcvCompleteStatus**, the adapter should fill in [ESI].RPacketOffset with the physical address of where the protocol header starts (which is usually all the adapter is capable of doing anyway).

Important



TSM's will not support HSM's that set **both** *DriverSendWantsECB's* and *DriverNeedsBelow16Meg*.

RX-Net

Option 1. This option is used for RX-Net shared RAM adapters. The steps performed for this reception method are outlined below. The *RXNetTSMRcvEvent* procedure in Chapter 6 provides a detailed description of this process.

DriverISR

Set ESI to point to received packet.
Call **RXNetTSMRcvEvent**

The TSM copies the entire packet into an RCB if the fragment is wanted with no other interaction from the driver.

MSMServiceEvents

Option 2. This option is used for RX-Net programmed I/O adapters. The steps performed for this reception method are outlined below. *The RXNetTSMGetRCB* procedure in Chapter 6 provides a detailed description of this process.

DriverISR

Set ESI to point to a LookAhead buffer containing the header information as shown in Figure 5.1.

Call **RXNetTSMGetRCB**

The TSM checks the packet header information to see if the packet fragment is wanted and if so, returns a pointer to an RCB.

Determine the current position in the RCB fragment buffers and copies the data into the RCB.

Update the packet length field of the RCB.

Call **RXNetTSMRcvComplete**
MSMServiceEvents

Board Service

The board service routine generally needs to detect and handle both receive events and transmit complete events. The driver can be notified of these events by using either an interrupt service routine, **DriverISR**, a polling procedure, **DriverPoll**, or a combination of both. These routines are explained next.

DriverISR

DriverISR is called by the MSM when a hardware interrupt is detected. The driver needs only to service the adapter and return (do not use `iret`).



Novell requires that interrupts remain unaltered during **DriverISR**. Drivers must allow the support modules to control the interrupt state via calls to the **DriverEnableInterrupt** and **DriverDisableInterrupt** routines at the appropriate times. If a driver procedure must alter the interrupt state, it must restore the interrupt state before returning.

The interrupt service routine generally needs to detect and handle the following events:

- Receive Event
- Receive Error
- Transmit Complete
- Transmit Error

The ISR routine must continue checking for receive and transmit events until there are no more to be serviced.

Error detection and handling are optional in the cases where the hardware is able to handle transmit and receive errors without driver intervention. Even if the hardware has this capability, the driver must still be able to update or maintain the statistics table described in Chapter 3.

Receive Event

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 in the previous section.

Receive Error

If the HSM encounters a receive error, it must perform the following actions:

- Attempt to identify the error. While some cards provide greater diagnostic support than others, the HSM should attempt to pinpoint the specific cause of the error (buffer overflow, missed packet, checksum error, etc.).
- Increment diagnostic counters. The HSM should maintain the diagnostic counters in the statistics table for every detectable error condition. This will aid in debugging the driver as well as maintaining it in the future. The driver should also increment the generic statistic *TotalRxMiscCount* if a fatal receive error occurred that is not counted in any other standard counter. Fatal receive errors may also be counted by the TSM using a media specific counter as well.
- Pass the appropriate receive error bits to `<TSM>GetRCB`, `<TSM>ProcessGetRCB`, or `<TSM>FastProcessGetRCB`.

Transmit Complete

Each time the HSM detects a successfully completed transmit event, it must perform the following functions:

- Release the TCB using (if not already released in *DriverSend*)

call `<TSM>SendComplete`

- Increment the number of available transmit resources

inc `[ebp].MSMTxFreeCount`

- Transmit the next packet if one is waiting to be sent

```
test [ebp].MSMStatusFlags, TXQUEUED
jz   NoSendsQueued
call <TSM>GetNextSend
jnz  NoSendsQueued
call DriverSend
```

Transmit Errors

If the HSM encounters a transmit error, it should perform the following actions:

- **Attempt to identify the error.** As with receive errors, the HSM should try to pinpoint the specific cause of the error (excess collisions, cable disconnect, FIFO underrun, etc.).
- **Increment diagnostic counters.** The HSM should maintain the diagnostic counters in the statistics table for every detectable error condition. The HSM should also increment the generic statistic *TotalTxMiscCount* if a fatal transmit error occurred that is not counted in any other standard counter. The fatal transmit error may be counted by the TSM using a media specific counter as well.
- **Attempt to send the packet again.** In the event the HSM has reached the maximum retry limit for sending a packet, discard the packet, increment *MSMTxFreeCount*, and transmit the next packet if one is waiting to be sent.

Using Shared Interrupts

An HSM can support shared interrupts provided that they are also supported by the host bus and the adapters which will 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 interrupt.

- The Micro Channel bus always uses level-triggered interrupts and can support shared interrupts.
- The PCI bus always uses level-triggered interrupts and can support shared interrupts.
- The PC/AT bus normally 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 **DriverISR** routine which supports shared interrupts is very similar to one which does not. If the HSM supports shared interrupts, the ISR must perform the following operations:

- Immediately determine if the interrupt request is from its adapter. If not, return at once to the operating system ISR with EAX equal to a nonzero value and the zero flag cleared.

```
or    al, 01h        ; clear the zero flag
ret                                ; return to operating system ISR code
```

- If the interrupt request is from the HSM's adapter, the interrupt service routine should proceed. Upon completion, the ISR must return with EAX equal to zero and with the zero flag set.

```
xor    eax, eax        ; zero eax & set the zero flag
ret                                ; returns to operating system ISR code
```

The HSM must indicate that the adapters are sharing interrupts by setting bit 5 in the *MLIDSharingFlags* field of the configuration table. The HSM must also initialize the *DriverParameterBlock* variable, *DriverEndOfChainFlag*, as described in the following table.

If the HSM:	The HSM must:	DriverEndOfChainFlag value:
Supports shared interrupts	Set the <i>IOSharingInterrupt0Bit</i> (bit5) in the <i>MLIDSharingFlags</i> field of the HSM's configuration table.	Zero The shared interrupt vector is placed first on the shared interrupt chain. If another interrupt vector is requested after the original vector is placed at the head of the chain, the latter vector will be serviced first.) Nonzero The shared interrupt vector is placed at the end of the shared interrupt chain by the operating system.
Does not support shared interrupts	Clear the <i>IOSharingInterrupt0Bit</i> (bit5) in the <i>MLIDSharingFlags</i> field of the HSM's configuration table.	Not used.

DriverISR Pseudocode

Processor States'	Entry State
	EBP pointer to the Adapter Data Space
	Dir Flag is cleared
	Interrupts are disabled (Novell recommends interrupts remain disabled during the <i>DriverISR</i>)
	Return State
	Dir Flag must be cleared
	Interrupts must be disabled
	Note no registers are preserved

Pseudocode

```
DriverISR  proc

CheckStatus:

    Get the controller's interrupt status

ReceiveEvent:

    .
    .
    .
    (check for receive errors and handle)
    .
    .
    .

*** Setup a LookAhead Buffer ***

    mov     ecx, [ebp].MSMMaxFrameHeaderSize
    lea     edi, [ebp].LookAheadBuffer
    rep     insb

*** Obtain an RCB for the Received Packet ***

    lea     esi, [ebp].LookAheadBuffer
    mov     ecx, HardwareReportedPacketSize
    call    <TSM>GetRCB
```



```

    if RCB is NOT available
        skip this packet
        jmp CheckStatus

```

*** Copy data and deliver RCB ***

```

    copy the packet data into the RCB fragment buffers
    call    <TSM>RcvComplete
    jmp     CheckStatus

```

TransmitEvent:

```

    .
    .
    .
    (check for transmit errors and handle/retry)
    .
    .
    .

```

TransmitComplete:

```

    reset retry counter to Maximum value
    mov     [ebp].TxInProgress, FALSE
    inc     [ebp].MSMTxFreeCount

```

*** Transmit Next Packet In the Send Queue ***

```

    test     [ebp].MSMStatusFlags, TXQUEUED
    jz       Exit
    call     <TSM>GetNextSend
    jnz      Exit
    call     DriverSend

```

Exit:

```

    MSMSERVICEEventsAndRet

```

```


DriverISR    endp

```

DriverPoll

Processor States	Entry State
	EBP pointer to the Adapter Data Space
	EBX pointer to the Frame Data Space
	Interrupts are disabled
	Return State
	EBP must be preserved

Description The **DriverPoll** procedure is used if the HSM requires a poll-driven board service routine. This routine will typically perform functions similar to those of the **DriverISR** procedure.

Note  **DriverPoll** is normally not used by an interrupt-driven HSM, however, there may be some cases where polling is required or where polling is used in addition to the ISR.

To register the polling procedure, place a pointer to the procedure in the *DriverPollPtr* field of the DriverParameterBlock. The driver can then enable polling during initialization by calling **MSMEnablePolling**.

DriverPoll is very time-consuming, especially in a Multi-Processor environment. Each time **DriverPoll** is called, the Mutex is acquired and **DriverDisableInterrupts** and **DriverEnableInterrupts** are both called. This causes the Mutex to be held a high percentage of the time and causes excessive bus traffic.

While **DriverPoll** is executing, the driver is not doing any usable work and is locked out from receiving interrupts and DriverSends, etc.

MSMSuspendPolling will temporarily stop the driver from being polled. The *POLLSUSPENDED* flag (bit4) in *MLIDStatusFlags* is set by the MSM when **MSMSuspendPolling** is called and is cleared by the MSM when **MSMResumePolling** is called. *MLIDStatusFlags* can be inspected by the HSM to determine the current polling status.

Packet Transmission

This section provides a brief overview of the methods commonly used for packet transmission.

When sending a packet, a protocol stack assembles a list of fragment pointers in a transmit ECB and passes it to the LSL. The ECB is then transferred to the TSM where the information is processed and a TCB is constructed. The TCB structure consists of the assembled packet header and data fragment information. The TSM directs the TCB to the **DriverSend** routine which collects the header and packet fragments and transmits the packet.

Transmission Methods

The method of packet transmission selected is typically dependent on the adapter's data transfer method. The examples on the following pages are intended to illustrate a general flow of events. Refer to the appropriate MSM and TSM support call descriptions for detailed information.

In general, packet transmission involves the following steps:

- During **DriverInit**, initialize *MSMTxFreeCount* to the number of adapter transmit resources available.
- The TSM builds a TCB, checks to see if the driver can handle another transmit and if so, decrements *MSMTxFreeCount* and calls **DriverSend** (otherwise the TSM queues the packet).
- **DriverSend** will typically copy the media header and data fragments to the transmit buffer and start the transmission.
- The driver returns the TCB back to the TSM using **<TSM>SendComplete**. This can be performed before the actual transmission is complete as long as all information has been collected from the TCB and the TCB is no longer needed (a "lying" send). The underlying transmit ECB will be placed in the LSL's holding queue until the HSM issues a service events command.
- Use the **MSMServiceEvents** macro to allow the LSL to call the transmit ECB's event service routine.

HSM Procedures **5-27**

- When the actual transmission is complete, increment *MSMTxFreeCount*. This is typically performed during **DriverISR** after a transmit complete interrupt.

Programmed I/O, Shared RAM, and Host DMA

The sequence of events for transmitting a packet using programmed I/O, shared RAM, or host DMA adapters is described below.

HSM

1. Sets *MSMTxFreeCount* to the maximum number of transmit packets that the adapter can buffer. (performed in **DriverInit**)

TSM

2. If the Ethernet TSM is used, ECX is set to the padded length of the packet. (This is the value that the adapter will send onto the wire, regardless of the value in the *TCBDataLen* field. In fact, the value in ECX is not equal to *TCBDataLen* if the packet is Ethernet 802.3 or Ethernet II and was evenized or if the packet was padded to 60 bytes.)
3. Decrements *MSMTxFreeCount* and calls **DriverSend** with ESI pointing to a filled in TCB structure.

HSM

4. Calls **<TSM>SendComplete** or **<TSM>FastSendComplete** either after the packet has been buffered onto the adapter or after the transmission has been completed.
5. Increments *MSMTxFreeCount* after the adapter completes the transmission (typically performed in **DriverISR**).

Bus Master

Option 1. This option is identical to the method described on the previous page for programmed I/O, shared RAM, and host DMA adapters.

Option 2. This method is recommended if the adapter is ECB aware and has sufficient adapter processor speed. It dramatically decreases the load on the server by reducing the host's process time.

HSM

5-28 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

1. Sets *DriverSendWantsECBs* to a nonzero value and sets *MSMTxFreeCount* to the number of transmit packets that the adapter can process at one time. (performed in **DriverInit**)

TSM.

2. Decrements *MSMTxFreeCount* and calls **DriverSend** with a pointer to the Frame Data Space in EBX and a pointer to the ECB in ESI.

HSM

3. Calls either **<TSM>SendComplete** or **<TSM>FastSendComplete** after the packet has been buffered onto the adapter or after the transmission has been completed.
4. Increments *MSMTxFreeCount* after the adapter completes the transmission (typically performed in **DriverISR**).

HSM Procedures 5-29

Bus Master Send Routine

Bus master adapters generally need physical addresses to ECB fragment pointers and control information in memory. These adapters would set the *DriverSupportsPhysFrgs* bit. If this bit is set the following applies:

1. The TSM will be responsible for providing only physical fragment offsets. If one of the fragments is not physically contiguous, the TSM will either:
 Modify the ECB fragment structure to break the fragment into multiple fragments
 or
 Copy the fragments into a buffer and pass a single fragment TCB to the HSM with one physical fragment offset to the buffer.

The ECB address passed to the HSM in register ESI will be a logical address so that the host portion of the HSM can read the fragment structure. Also the following results will be true:

TCBDriverWS+4 will be:

- 0 if no double copy was performed
- 1 if a double copy was performed (which may be used for statistics).

TCBDriverWS+8 will be:

The physical address of the *TCBMediaHeader*.

These two fields can still be used freely by the HSM.

2. *DriverNeedsBelow16Meg* set in *DriverParameterBlock*:
 If the TSM finds a fragment with an address over the 16 megabyte boundary, it will double copy all ECB fragments into a buffer guaranteed to be below 16 megabytes and pass a TCB to the HSM with one fragment. The one fragment pointer will be a physical address which points to the buffer.
3. *DriverSendWantsECBs* set in *DriverParameterBlock*:
 ESI will contain the logical address of the ECB and EDI will contain the physical address of the ECB. All fragment pointers will contain physical addresses to contiguous buffers as described in section 1 above.

Important



TSM's will not support HSM's that set both *DriverSendWantsECB's* and *DriverNeedsBelow16Meg*!

Priority Transmission Support

The following algorithm is used for priority transmission support:

HSM

1. During **DriverInit** the HSM sets the following parameters:
2. The *DriverPriorityQueuePtr* field of the Driver Parameter Block is set with a pointer to **DriverPriorityQueueSupport**.

Bit 12 in the MLIDFlags field of the MLID Configuration Table is set.

The MLIDPrioritySup field in the MLID Configuration Table is set to indicate the number of levels available.

MSMPriorityTxFreeCount is set to the maximum number of priority transmissions that the HSM can handle simultaneously.

3. The HSM can set or reset MLIDFlags bit 12 as the HSM changes the Priority Queue Support state from enable to disabled. This bit is checked on a per queue packet basis.

Protocol Stack

4. The protocol stack sets the ECB LogicalID field to a value greater than or equal to FFF0h. The following values are valid for the LogicalID field:

FFFFh	Raw send, no Priority. (Priority 0)
FFFEh	Raw send, Priority 1 (Scale 1-7: 1 = Lowest Priority)
FFFDh	Raw Send, Priority 2
FFFCCh	Raw Send, Priority 3
FFFBh	Raw Send, Priority 4
FFFAh	Raw Send, Priority 5
FFF9h	Raw Send, Priority 6
FFF8h	Raw Send, Priority 7 (Scale 1-7: 7 = Highest Priority)
FFF7h	Raw send, no Priority. (Priority 0)
FFF6h	Priority 1 (Scale 1-7: 1 = Lowest Priority)
FFF5h	Priority 2
FFF4h	Priority 3
FFF3h	Priority 4
FFF2h	Priority 5
FFF1h	Priority 6
FFF0h	Priority 7 (Scale 1-7: 7 = Highest Priority)

TSM

5. The TSM normally gives the packet to the HSM directly, as a TCB using the **DriverSend** function. However, if **MSMTxFreeCount** is zero and the transmit ECB is a priority transmit ECB, the TSM calls **DriverPriorityQueueSupport**, which gives the HSM a chance to take the transmit ECB. The **DriverPriorityQueueSupport** function, provided by the HSM, queues the ECB in the HSM for transmission as soon as possible, or transmits the packet through a priority channel by first building a TCB using **<TSM>BuildTransmitControlBlock**, or returns a failure code and does not accept the ECB.

HSM

6. The HSM calls **<TSM>BuildTransmitControlBlock** to build a TCB whenever a priority transmit resource becomes available and a transmit ECB in the HSM's priority queue. The HSM tracks the number of available priority TCBs. **MSMPriorityTxFreeCount** is set during **DriverInit** and must provide the maximum number of priority TCBs, which must not change without unloading and reloading the HSM. Non- priority packets use the original number of TCBs from **MSMTxFreeCount**, which is reserved exclusively for their use. The HSM must not call **<TSM>BuildTransmitControlBlock** if no priority TCBs are available.
7. After the HSM has transmitted the TCB returned by **<TSM>BuildTransmitControlBlock**, the HSM calls **<TSM>SendComplete** or **<TSM>FastSendComplete**, which increments the statistic counters, call **TxMonitor**, places the TCB back on the TCBs free list, and returns the ECB to its original owner.

DriverSend

Processor States

Entry State

EBP	pointer to the Adapter Data Space
EBX	pointer to the Frame Data Space
ESI	pointer to a TCB or an ECB (see note below)
EDI	If the HSM has set DriverSendWantsECBs to a nonzero value and has set the DriverSupportsPhyFrgs bit in MLIDModeFlags , this register contains the physical address of the ECB.
ECX	padded length of the packet (Ethernet only)
Interrupts	are disabled. Novell recommends that system interrupts remain disabled during DriverSend .

Return State

Interrupts	must be disabled
------------	------------------

Description

The TSM calls **DriverSend** to transmit a frame onto the medium. **DriverSend** is provided a pointer to a Transmit Control Block (TCB). Refer to Chapter 3 for information on TCBs.

The HSM can assume that the TCB is valid for its LAN medium; it must not do consistency checking on the TCB fields. The HSM can also assume that it has the resources necessary to handle the transmit operation; it does not need to check to see if it has a transmit hardware resource available. The TSM performs flow control for the HSM. The TSM determines if the HSM can handle the packet by checking the value of *MSMTxFreeCount*.

Note



The **DriverSend** routine may request ECBs instead of TCBs by initializing the **DriverParameterBlock** variable **DriverSendWantsECBs** to a nonzero value (see Chapter 3). If **DriverSend** uses ECBs for packet transmission, it is responsible for building the proper media header (refer to Chapter 4 for additional information on ECB aware adapters). If the HSM uses ECBs instead of TCBs, it must not modify the transmit ECB's *BLink* field.

Note



For drivers that support physical fragments, the ESR field of the ECB with physical fragments contains a pointer to the original ECB (with logical fragments) handed to the MSM (refer to figure 4.11 in Chapter 4).

pseudocode

Copy the MediaHeader from the TCB into a transmit buffer.
Copy the fragmented data from the TCB's fragment structure into a transmit buffer.
Give the command to send the packet.
Restore ESI to point to the beginning of the TCB

IF called from **DriverISR**
 Call **<TSM>SendComplete** ; (lying send)
ELSE
 Call **<TSM>FastSendComplete**
ENDIF

Return

Driver Priority Queue Support

Called by the TSM to allow the HSM to handle a priority packet or to allow the TSM to queue it for normal transmission.


Processor States	Entry State	
	EBP	pointer to the Adapter Data Space.
	EBX	pointer to the Frame Data Space.
	ESI	pointer to a transmit ECB.
	Interrupts	are disabled. Novell recommends that system interrupts remain disabled during DriverPriorityQueueSupport.
	Return State	
	EAX	completion code
	Interrupts	are disabled

Completion Code in EAX	
SUCCESSFUL	The ECB was processed/queued by the HSM.
OUT_OF_RESOURCES	The ECB was not processed/queued by the HSM. The TSM will queue the ECB and initiate transmission at a later time.

Description

This function must either transmit the packet immediately or queue the ECB. The HSM must be able to service the priority queue and handle priority level detection issues. This function should process essential items only and return as quickly as possible. This function may be called while the HSM is executing critical section code.

The HSM must set DriverPriorityQueuePtr in the Driver Parameter Block to point to this function. The HSM can set or reset the configuration table MLIDFlags bit 12 from supporting to not supporting priority packet states. This bit is checked by the TSM on a per packet basis.

Note

The *RDriverWorkSpace* field of the ECB must not be modified by the HSM.

Multi-Operating System Support

Driver specification v3.1 and later enables HSMs to be transported to other 32-bit Intel-based operating system platforms without any code modification. In order to achieve this universal 32-bit HSM, two new driver routines have been added.

A universal 32-bit HSM must be able to control interrupts at the adapter, and must implement the **DriverEnableInterrupt** and **DriverDisableInterrupt** routines. The resulting HSM can be transported to other OS platforms where access to the Programmable Interrupt Controller (PIC) is restricted. Also multiprocessor platforms require interrupts to be managed outside the driver.

Drivers must allow the MSM and TSM to control the interrupt state via calls to the **DriverEnableInterrupt** and **DriverDisableInterrupt** routines at the appropriate times. Novell requires that the interrupt states remain unaltered during driver procedures. If a driver procedure must alter the interrupt state, it must restore it to the original state before returning.

Important



The new specification prohibits use of the CLI and STI instructions in HSM code, or to directly EOI the PIC.

Because the server/32-bit HSM cannot contain direct calls to the OS and still function as a Windows NT client all OS calls have been eliminated from this document along with all LSL calls and several MSM calls. The toolkit routines provided by Novell (in combination with the redirector or requester) must handle the unique features of interfacing to each OS.

Critical Sections


Starting with MSM.NLM v2.21 and the current TSM's* the **MSMStartCriticalSection**, **MSMEndCriticalSection**, and **MSMGetCriticalStatus** macros are no longer supported and must not be used in universal 32-bit HSM's. This applies to the MSM and all TSM's dated after 2-20-94.

The MSM and TSM's have been modified so that any call they receive from the HSM after the driver is initialized will start a critical section. This will allow the called routine to run to completion in case **DriverSend** is called during this time. The critical section will then be cleared by the MSM or TSM routine before it returns control to the HSM.

DriverEnableInterrupt

Processor States		Entry State
	EBP	pointer to the Adapter Data Space
	Interrupts	can be in any state
		Return State
	Interrupts	preserved
	EBP	must be preserved

Description This procedure enables interrupts at the adapter hardware.

Note  **DriverEnableInterrupt** need only be called once to enable interrupts even though DriverDisableInterrupt may have previously been called several times by different processes. Also it is important to keep the **DriverEnableInterrupt** procedure as short as possible.

PseudocodeDriverEnableInterrupt proc

 Enable the adapter to generate interrupts
 ret

DriverEnableInterrupt endp

DriverDisableInterrupt

Processor States

Entry State

EBP	pointer to the Adapter Data Space
EAX	zero - does not require a return value in EAX one - HSM must return a value in EAX as described below
Interrupts	can be in any state

Return State

EAX	If EAX was one on entry: EAX is zero if service is being requested by the LAN adapter's first interrupt. EAX is one if the LAN adapter's first interrupt is not requesting service. In this case, the TSM calls DriverEnableInterrupt on return from this routine.
Interrupts	preserved
EBP	must be preserved

Description

This function disables interrupts on the adapter hardware. If the adapter generates more than one interrupt, all interrupts must be disabled by this function. However, only the request state of the first interrupt must be returned.

Note



DriverEnableInterrupt need only be called once to enable interrupts even though DriverDisableInterrupt may have previously been called several times by different processes.


Pseudocode

```
DriverDisableInterrupt  proc
    Disable the adapter from generating interrupts
    mov  eax, <Appropriate Status>
    ret
DriverDisableInterrupt  endp
```

DriverDisableInterrupt2

Processor States	Entry State	
	EBP	pointer to the Adapter Data Space
	EAX	zero - does not require a return value in EAX one - HSM must return a value in EAX as described below
	Interrupts	can be in any state
	Return State	
	EAX	If EAX was one on entry: EAX is zero if service is being requested by the LAN adapter's second interrupt. EAX is one if the LAN adapter's second interrupt is not requesting service. In this case, the TSM calls DriverEnableInterrupt on return from this routine.
	Interrupts	preserved
	EBP	must be preserved

Description This function disables interrupts on the adapter hardware. If the adapter generates more than one interrupt, all interrupts must be disabled by this function. However, only the request state of the second interrupt must be returned.

Note  DriverEnableInterrupt need only be called once to enable interrupts even though DriverDisableInterrupt may have previously been called several times by different processes.

Pseudocode

```
DriverDisableInterrupt2 proc
    Disable the adapter from generating interrupts
    mov  eax, <Appropriate Status>
    ret

DriverDisableInterrupt2 endp
```


Control Procedures

The ODI specification requires drivers to implement the I/O control functions (IOCTLs) listed in the table below. The MSM and TSM development tools perform several of the required IOCTL functions without assistance from the HSM, as indicated in the table. The support modules will also “front end” all control functions and preserve any required registers. The HSM is responsible for implementing the control functions described in this section.

DriverReset and **DriverShutdown** are mandatory and must be present for the driver to function properly. The HSM must also provide the **DriverMulticastChange** and **DriverPromiscuousChange** procedures when the hardware supports these functions.

The **DriverStatisticsChange** and **DriverRxLookAheadChange** procedures are optional. These procedures allow drivers for intelligent adapters to update the statistics table or the LookAhead size only as needed. Refer to the DriverParameterBlock field descriptions in Chapter 3 for additional information on these two control procedures.

Drivers that support the Hub Management Interface or Brouter must implement the **DriverManagement** procedure to handle management requests and commands as described in *The Hub Management Interface* supplement or the *Brouter Support* supplement

Control Function	Code Path
0 Get Configuration Table	MSM
1 Get Statistics Table	MSM -> DriverStatisticsChange
2 Add Multicast Address	MSM -> TSM -> DriverMulticastChange
3 Delete Multicast Address	MSM -> TSM -> DriverMulticastChange
4 Reserved	MSM
5 Shutdown Driver	MSM -> TSM -> DriverShutdown (EAX = OP_SCOPE_ADAPTER) (ECX = PERMANENT_SHUTDOWN or PARTIAL_SHUTDOWN)
6 Reset Driver	MSM -> TSM -> DriverReset (EAX=OP_SCOPE_ADAPTER)
7 Reserved	MSM
8 Reserved	MSM
9 Set receive LookAhead size	MSM -> TSM -> DriverRxLookAheadChange
10 En/Dis Promiscuous Mode	MSM -> TSM -> DriverPromiscuousChange
11 En/Dis Receive Monitor	MSM -> TSM
12 Reserved	MSM
13 Reserved	MSM
14 Driver Management	MSM -> DriverManagement
15 Reserved	MSM
16 Remove Network Interface	MSM->TSM->DriverShutdown (EAX = OP_SCOPE_LOGICAL_BOARD) (ECX = PERMANENT_SHUTDOWN)
17 Shutdown Network Interface	MSM -> TSM -> DriverShutdown (EAX=OP_SCOPE_LOGICAL_BOARD) (ECX = PARTIAL_SHUTDOWN)
18 Reset Network Interface	MSM -> TSM -> DriverReset (EAX=OP_SCOPE_LOGICAL_BOARD)

5-42 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

DriverReset

Processor States

Entry State

EAX	OP_SCOPE_ADAPTER Reset the adapter specified by EBP. OP_SCOPE_LOGICAL_BOARD Reset the logical board specified by EBX.
EBP	pointer to the Adapter Data Space
EBX	pointer to the Frame Data Space
Interrupts	are disabled but may be enabled during the call

Return State

EAX	Zero if successful. FAIL (from ODI.INC) on failure.
Interrupts	are disabled

Description

OP_SCOPE_ADAPTER

If EAX equals OP_SCOPE_ADAPTER, **DriverReset** resets and initializes the adapter hardware.

This routine may also test the hardware to verify that it is functional. If the driver has been temporarily shutdown, an application may call this routine to bring the board back into full operation.

When a reset is required, the TSM waits for transmissions in progress to complete and calls **DriverReset**.

From within the HSM, **DriverReset** may be called by **DriverInit**. It may also be called by **DriverCallBack** or **DriverISR** if the adapter had problems.

If the MSM calls **DriverReset**, and it returns successfully, the MSM resets the *MSMTxFreeCount* variable to the initial value set by the driver during initialization. If the MSM calls **DriverReset**, and the adapter cannot be reset, the MSM automatically calls **DriverShutdown**.

OP_SCOPE_LOGICAL_BOARD

If EAX equals OP_SCOPE_LOGICAL_BOARD, **DriverReset** resets the logical board specified by EBX. The meaning of this operation is adapter/media/driver dependent. Except for re-enabling a shutdown logical board, this operation is a NO_OP for most LAN Drivers.

Pseudocode

```
If OP_SCOPE_ADAPTER
    Increment the reset statistics counter
    Reset the hardware (includes performing any hardware testing)
    Call <TSM>UpdateMulticast
    Set EAX to zero if successful
ELSE
    Do any necessary logical board specific action (usually a NO_OP)
ENDIF
```

DriverShutdown

Processor States

Entry State

EAX	OP_SCOPE_ADAPTER Shutdown the adapter specified by EBP. OP_SCOPE_LOGICAL_BOARD Shutdown the logical board specified by EBX.
EBP	Pointer to the Adapter Data Space
EBX	Pointer to the Frame Data Space
ECX	ZERO if a permanent shutdown, otherwise a partial shutdown is required.
Interrupts	Disabled but may be enabled during the call

Return State

EAX	Zero if successful. FAIL (from ODI.INC) on failure.
Interrupts	are disabled

Description

The MSM automatically calls **DriverShutdown** when the **DriverReset** routine fails to reset the hardware. *MSMReturnDriverResources* and *MSMExitToDOS* also call **DriverShutdown**.

Partial Shutdown

When a partial shutdown is required, the MSM sets *MSMStatusFlag*, waits for transmissions in progress to complete and returns the transmit ECBs. The MSM also sets bit 0 of the *SharingFlags* in the configuration table. **DriverReset** must be able to bring the adapter back into full operation.

Complete Shutdown

A zero value in ECX indicates a complete shutdown. As with a partial shutdown the MSM has set the flags, emptied the send queue, and also will return all resources not allocated directly by the HSM. If the HSM allocated memory using **MSMAlloc**, it must be returned using **MSMFree** before disabling the hardware.

OP_SCOPE_ADAPTER

If EAX equals OP_SCOPE_ADAPTER, **DriverShutdown** must place the hardware into a safe, inactive state.

If the adapter is to be shut down permanently (indicated by the value in ECX), the MSM disables the adapter's interrupt immediately after this routine returns. As far as the HSM is concerned, the only difference between a partial and a complete shutdown is the return of allocated memory.

OP_SCOPE_LOGICAL_BOARD

If EAX equals OP_SCOPE_LOGICAL_BOARD, **DriverShutdown** must release all HSM-allocated resources associated with the logical board specified by EBX.

```

Pseudocode OP_SCOPE_ADAPTER
    IF a permanent shutdown
        return any memory using MSMFree
    ENDIF

    return any preallocated RCBs or queued TCBs
    Disable Hardware
    Set EAX = 0
    ELSE
        Do any necessary logical board specific action (usually a NO_OP)
    ENDIF
    Return

```

DriverMulticastChange

Processor States	Entry State	
	EBP	pointer to the Adapter Data Space
	EBX	pointer to the Frame Data Space
	ESI	pointer to the Multicast Table (default for Ethernet or FDDI)
	ECX	Number of valid entries in the Multicast Table (default for Ethernet or FDDI)
	EDX	32-bit functional address (default for Token-Ring)
	Interrupts	are disabled on entry, but may be enabled during the routine
	Entry State	
	Note	EBX and EBP must be preserved
	Interrupts	must be disabled on return

Description

DriverMulticastChange updates the adapter to reflect the changes in the TSM's multicast address table. Novell **requires** that all HSMs support multicast addressing if the media supports it. The following flags and variables must be initialized properly for the adapter's multicast mode.

- Bit3 of the *MLIDModeFlags* is used to indicate whether or not multicast addressing is supported.
- Bits 9 and 10 of the *MLIDFlags* must be set appropriately to reflect the multicast mechanism or format used by the adapter/driver.
- The DriverParameterBlock variable, *DriverMaxMulticast*, must be set to reflect the maximum number of multicast addresses the adapter can handle.

The TSM maintains an internal table of multicast addresses. The TSM modules handle the addition and deletion of addresses in this table. Whenever the table changes, the TSM calls **DriverMulticastChange** to update the adapter's multicast filtering. The adapter may maintain its own multicast address table or use a hash table to filter incoming packets.

Adapter Multicast Filtering


The most common method used by adapters to filter incoming packets is hashing. When this is the adapter's method, **DriverMulticastChange** must recalculate and update the adapter's hash table. Hashing does not guarantee 100% multicast filtering; therefore, the TSM looks up incoming packets in its multicast address table to ensure that the packet's destination address is enabled.

In the case that the adapter keeps its own list of multicast addresses, this routine must cycle through the entries in the TSM's multicast address table and output each entry to the physical card. The TSM verifies that all addresses it places in its table are valid multicast addresses so the HSM does not need to validate them.

In either case, the HSM routine must read the TSM's multicast address table. Each entry in the table is 8 bytes long. The first 6 bytes are the address, and the last word is a use flag maintained by the TSM. If the use flag is nonzero, the entry contains a valid address.

```
MulticastEntryStruc      db 6 dup (?); multicast addresses
MulticastInUse           dw 0          ; Nonzero if in use
```

The default method (if bits 9 and 10 of the *MLIDFlags* are zero) for handling multicast operations is as follows:

Note

ECB aware HSMs must do their own filtering of multicast addresses.

Ethernet and FDDI

On entry to this routine, ECX contains the number of valid entries in the multicast table. All valid entries will be contiguous, so the HSM does not necessarily need to check the *MulticastInUse* flag. If ECX is zero, multicast reception is disabled.

Token-Ring

The TSM passes the 32-bit functional address in EDX. In this case ECX and ESI are normally not used.

If an adapter is capable of supporting both group and functional addresses (and sets bits 9 and 10 in the *MLIDFlags* field of the configuration table accordingly), the *DriverMulticastChange* routine will receive both functional addresses and multicast table information.

RX-Net

DriverMulticastChange cannot be supported by RX-Net drivers.

Pseudocode

Clear the hardware registers that filter incoming packets for multicast addresses
Get current multicast addresses from TSM's multicast table
Reload hardware register with new multicast address filtering values
Return


DriverPromiscuousChange

Processor States	Entry State	
	EBP	pointer to the Adapter Data Space
	EBX	pointer to the Frame Data Space
	ECX	zero disables promiscuous mode
	setting one or more bits enables promiscuous mode as follows: bit 0 (PROM_MODE_MAC) is set if MAC frames are to be received bit 1 (PROM_MODE_NON_MAC) is set if non-MAC frames are to be received bit 2 (PROM_MODE_SMT) is set if FDDI SMT type MAC frames are to be received bit 3 (PROM_MODE_RMC) is set if remote multicast frames are to be received all bits are set if all frames are to be received	
	Interrupts	are disabled but may be enabled during the call
	Return State	
	Note	EBP and EBX must be preserved
	Interrupts	are disabled

Description

Adapters/drivers that can pass all packets to a monitor function in the protocol stack are said to have a promiscuous reception mode.

DriverPromiscuousChange provides a means for the stack monitor function to enable or disable promiscuous reception.

Note  Enabling promiscuous mode will have a detrimental impact on system performance.

A monitoring function examines packets sent from or received by an adapter. If promiscuous mode is supported, the monitoring function can request that the adapter enter promiscuous mode. When promiscuous mode is enabled, the driver should allow all packets (including bad packets if possible) to be passed up to the monitor function. Only one monitor function at a time may be registered with a driver.

Be aware that a monitor function may set the configuration table's *MLIDLookAheadSize* to a value other than the 18 byte default. This will in turn change *MSMMaxFrameHeaderSize*.

The **<TSM>GetRCB** and **<TSM>ProcessGetRCB** require the driver to indicate the status of the packet in EAX. EAX will always equal zero for token-ring, RX-Net, and FDDI. For Ethernet the status options are as follows:

- EAX = zero for good packets
- EAX = non-zero for bad packets
- EAX bits are set as follows for bad packets:
 - Bit 0 - CRC Error
 - Bit 1 - CRC/Alignment Error
 - Bit 2 - Runt packet (set by the Ethernet TSM)
 - Bit 8 - Receive too big for ECB (set by the TSM)
 - Bit 9 - No board number registered (set by the TSM)
 - Bit 10 - Malformed packet (set by the TSM)
 - Bit 14 - Do not decompress the received packet
 - Bit 15 - The address in *RImmediateAddress* is in noncanonical format
 - Bit 31 - Driver shutting down (set by the TSM)

An ECB aware HSM must set all of these bits as necessary before calling **<TSM>RcvCompleteStatus** or **<TSM>FastRcvCompleteStatus**. An RCB aware HSM need set only Bit 0 - CRC error and Bit 1 - CRC/Alignment error as necessary, the others are set by the TSM if needed.

If the HSM does not support promiscuous mode, bit 13 of the *MLIDModeFlags* in the configuration table must be cleared and the *DriverPromiscuousChangePtr* field in the *DriverParameterBlock* must be zero.



Setting the Remote Multicast Frames bit causes the HSM to activate all multicast frame reception. For example, if an adapter utilizes a hash table for filtering active multicast frames, the adapter sets the hash table to accept all multicast frames. Filtering active multicast entries is disabled when this bit is set. HSMs that can filter must also disable filtering when this bit is set. Multiple bits may be set; each bit adds to the type of frames that are to be received.

HSM Procedures **5-51**

Pseudocode

```
IF requested to enable promiscuous mode
    send enabling command to hardware
ELSE
    send disabling command to hardware
```

DriverStatisticsChange (optional)

Processor States	Entry State
	EBP pointer to the Adapter Data Space
	EBX pointer to the Frame Data Space
	Interrupts are disabled but may be enabled during the routine
	Return State
	Interrupts are disabled
	Note EBP must be preserved

Description The *DriverStatisticsChange* routine allows the MSM's control procedure handler to notify drivers whenever an application requests IOCTL 1 (get driver statistics). This allows HSMs for intelligent adapters that maintain statistical information on board to update the statistics table in the Adapter Data Space only as needed (before the MSM passes it up to the requesting application).

For additional information, refer to the *DriverStatisticsChangePtr* field of the DriverParameterBlock in Chapter 3.

Pseudocode

Transfer statistics maintained by the hardware to the statistics table in the Adapter Data Space.

DriverRxLookAheadChange (optional)

Processor States	Entry State	
	EBP	pointer to the Adapter Data Space
	EBX	pointer to the Frame Data Space
	Interrupts	are disabled but may be enabled during the routine
	Return State	
	Interrupts	are disabled
	Note	EBP must be preserved

Description

The *DriverRxLookAheadChange* routine allows the MSM to notify drivers after an application invokes IOCTL 9 to set the LookAhead size. This IOCTL changes the *MSMMaxFrameHeaderSize* variable and the *MLIDLookAheadSize* field in the configuration table. Drivers can use this routine to inform intelligent adapters only when the size changes rather than constantly checking the value.

Refer to the *DriverRxLookAheadChangePtr* field of the *DriverParameterBlock*, the *MLIDLookAheadSize* in the configuration table, the *MSMMaxFrameHeaderSize* variable, and the <TSM>**GetRCB** procedure for additional information.

Pseudocode

Inform Adapter of new *MSMMaxFrameHeaderSize* (or new *MLIDLookAheadSize*)

DriverManagement (optional)

Processor States	Entry State	
	EBP	pointer to the Adapter Data Space
	EBX	pointer to the Frame Data Space
	ESI	pointer to the management ECB containing the request
		(see supplements for <i>The Hub Management Interface</i> or <i>Brouter Support</i>)
	Interrupts	are disabled but may be enabled during the routine
	Entry State	
	Interrupts	are disabled
	EAX	00000000h = Success; command ECB relinquished
		00000001h = Success; command ECB queued
		FFFFFFF8h = no such handle - protocolId not supported

Description	If a driver accepts management commands from outside NLMs the MSM will call the <i>DriverManagement</i> routine to process the management requests.
	Refer to <i>The Hub Management Interface</i> and the <i>Brouter Support</i> supplements for an implementation of this procedure. See also the <i>DriverManagementPtr</i> field of the <i>DriverParameterBlock</i> in Chapter 3.
Pseudocode	(refer to <i>The Hub Management Interface</i> supplement for an example <i>DriverManagement</i> routine)

Timeout Detection

Historically, the **DriverAES** and **DriverCallBack** routines have been used to call the HSM at periodic intervals. They have also been used to determine if a board has failed to complete a packet transmission, as well as other timed functions.

Which routine should be used for HSM timeout handling depends on execution time constraints:

- When **DriverCallBack** is executing, the HSM may only call routines that can be called at interrupt time.
- When **DriverAES** is executing, the HSM may only call routines that can be called at process time.

New with this specification, the driver can use **MSMScheduleTimer** to achieve the same results as **DriverAES** and **DriverCallBack**.

MSMScheduleTimer is more versatile than either of these two routines and is now the preferred method for handling timer events. (See Chapter 7, “MSM Procedures and Macros”.)



RX-Net normally uses a specific routine, **DriverTxTimeout**, to handle transmit timeouts. This routine is required only when the RX-Net module is used. RX-Net drivers may also use the other two timing event routines.

DriverTxTimeout (RX-Net)

The RX-Net TSM calls **DriverTxTimeout** whenever a transmit has a software timeout. Under normal conditions, the HSM issues the Disable Transmitter command to the card. If the hardware does not require any special attention, the HSM simply returns.

DriverTxTimeout is called at interrupt time and should be optimized to be as efficient as possible. This procedure must be included when the HSM uses the RX-Net support module.

DriverAES / DriverCallBack/TimerProcedure

Processor States	Entry State	
	EBP	pointer to the Adapter Data Space
	EBX	pointer to the Frame Data Space
	Interrupts	are enabled for DriverAES are disabled for DriverCallBack
Return State		
	Note	EBP must be preserved

Description

DriverAES is enabled (typically during initialization) by calling **MSMScheduleAESCallBack**. **DriverCallBack** is enabled by calling the function **MSMScheduleIntTimeCallBack**. The use of these two MSM calls is explained in Chapter 7, but briefly, the MSM routines expect EAX to contain the desired time interval in ticks (1 tick = approx. 1/18 second).

Once enabled, the MSM invokes the routine automatically at the end of each interval with EBX pointing to the Frame Data Space and EBP pointing to the Adapter Data Space. Interrupts are enabled when **DriverAES** is called and are disabled on calls to **DriverCallBack**.

New with this specification, the driver can use **MSMScheduleTimer** to setup a timer callback routine. **MSMScheduleTimer** is more versatile than **DriverAES** or **DriverCallBack** and is now the preferred method for handling timer events. (See Chapter 7, “MSM Procedures and Macros”.)

The actual content of the routines is entirely up to the developer. The pseudocode here illustrates using **DriverCallBack** to identify a send timeout error.

Pseudocode

```
IF Transmit is in Progress

    IF Elapsed Transmit Time > Maximum Time for Transmit
        Increment appropriate error counter
        Reset the adapter
        Reset [ebp].MSMTxFreeCount

        Call <TSM>GetNextSend;(check the send queue)
        IF TCB was available
            Call DriverSend
        ENDIF
    ENDIF
ENDIF
Return
```

Removal

The NetWare operating system calls the driver's exit procedure, **DriverRemove**, when it receives the command to unload the driver. This procedure is described below.

DriverRemove

Processor States	Entry State	
	Interrupts	can be in any state
	Return State	
	Interrupts	are preserved

Description

The **DriverRemove** procedure is called whenever the HSM is unloaded. The HSM's linker definition file must include the "exit" keyword followed by **DriverRemove**. Because this routine is called by the operating system, it must preserve the C registers EBP, EBX, ESI and EDI.

This routine must set EAX to the value of the *DriverModuleHandle* from the DriverParameterBlock and call **MSMDriverRemove**. The MSM handles MLID deregistration, returns all driver resources, and calls **DriverShutdown** before returning.

Pseudocode

```
DriverRemove      proc
    push          ebx, ebp, esi, edi
    mov          eax, DriverModuleHandle
    call         MSMDriverRemove
    pop          edi, esi, ebp, ebx
    ret
DriverRemove      endp
```

*** Setup a Board Service Routine ***

5-60 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

chapter **6** ***TSM Procedures***

Introduction

This chapter describes the topology specific procedures provided as tools for HSM developers. The Topology Specific Module, <TSM>.NLM, manages the operations that are unique to a specific media type. Multiple frame support is implemented in the Topology Module so that all frame types for a given media are supported.

The topology specific functions are indicated with <TSM>. The developer must replace <TSM> with the appropriate media type depending on which module is used. Since the driver must be assembled with case sensitivity on, the names must be used exactly as shown.

ETHERTSM .NLM	replace <TSM >with: Ether TSM
TOKENTSM.NLM	replace <TSM> with: TokenTSM
RXNETTSM.NLM	replace <TSM> with: RXNetTSM
FDDITSM.NLM	replace <TSM> with: FDDITSM

RX-Net drivers require special consideration to handle split packets. Several routines are provided that are specific to the RX-Net module. These routines are described at the conclusion of this chapter.

<TSM>BuildTransmitControlBlock

Processor States	Entry State	
	EBP	pointer to the Adapter Data Space
	ESI	pointer to a transmit ECB
	Interrupts	are disabled
	Call	at process time
	Return State	
	ECX	padded length of the packet (non-ECB-aware Ethernet only)
	ESI	pointer to a TCB (or an ECB); otherwise, NULL
	Flags	set according to EAX
	Interrupts	are disabled
	Preserved	EBP, EDI
	Completion Code in EAX	
	SUCCESSFUL	TCB pointer is valid. The HSM should transmit the TCB.
OUT_OF_RESOURCES	A TCB was not available. THE HSM must not call this routine with more outstanding TCBs than is set in the MSMPriorityTxFreeCount variable. The ECB is returned to HSM. The HSM must either call this function again after a TCB resource is available, or return the ECB via <TSM>CancelPrioritySend .	
	A TCB was available, but the ECB described a packet that was too large for the media. The ECB was returned to the LSL and a TCB was not allocated.	
PACKET_UNDELIVERABLE	Note: ECB-aware drivers needing physical addresses for fragments may experience this error if the frame's scatter/gather count is too high. In which case, the ECB is returned to the LSL and ESI=NULL.	

Description

The HSM calls **<TSM>BuildTransmitControlBlock** when it is ready to send a priority packet that has been queued using **DriverPriorityQueueSupport**. The HSM calls this function to convert an ECB to a TCB.

The HSM should be aware of the number of TCBs available to the MLID for priority sends. The TSM allocates a number of TCBs based on the sum of **MSMTxFreeCount** and **MSMPriorityTxFreeCount**. The HSM must not have more outstanding priority TCBs than was set by the HSM using **MSMPriorityTxFreeCount** during **DriverInit**. If the HSM makes this call when no TCBs are available, **OUT_OF_RESOURCES** is returned.

The HSM does not need to do size checking on the resultant TCB. If the packet generated is too large for the media, this function returns **PACKET_UNDELIVERABLE** after it returns the ECB to the LSL. It does not return a TCB to the HSM.

The HSM must not change the **MSMTxFreeCount** for any TCB obtained for a priority transmit. An internal counter for priority support resources should be maintained by the HSM.

ECB-aware drivers that need physical addresses for fragments must call **<TSM>BuildTransmitControlBlock** to convert logical fragment addresses to physical addresses. A pointer to the new ECB with physical fragment addresses is returned in **ESI**.

<TSM>CancelPrioritySend

Processor States	Entry State	
	EBP	pointer to the Adapter Data Space
	ESI	pointer to a transmit ECB
	Interrupts	are disabled
	Call	at process time
Return State		
		Interrupts are disabled
		Preserved EBP

Description

The HSM calls this function when it is canceling an ECB that was originally accepted to be transmitted via **DriverPriorityQueueSupport**.

<TSM>GetConfigInfo

Allows an HSM to get the configuration information for the <TSM>, including specification and module versions.

Processor States

Entry State

EDI	pointer to buffer used to receive the returned configuration information. The caller must ensure that the buffer is at least as long as the number of bytes specified in ECX.
ECX	requested number of bytes to be returned into the buffer.
Interrupts	can be in any state
Call	at process time

Return State

ECX	the actual number of bytes returned in the configuration buffer.
Flags	set according to EAX
Interrupts	are disabled
Preserved	EDI, EBX & EBP

Completion Code in EAX

SUCCESSFUL	The configuration information was successfully returned in the buffer.
BAD_PARAMETER	The number of bytes requested was larger than the actual configuration information available. The number of bytes actually returned is indicated in ECX.

Description

The configuration information is returned in the format defined by TSMConfigTable.

```
TSMConfigTable struc
    TSMCFG_TableSize          dd    ?
    TSMCFG_TableMajorVersion  db    ?
    TSMCFG_TableMinorVersion  db    ?
    TSMCFG_ModuleMajorVersion db    ?
    TSMCFG_ModuleMinorVersion db    ?
    TSMCFG_ODISpecMajorVersion db   ?
    TSMCFG_ODISpecMinorVersion db   ?
    TSMCFG_Reserved           dw    ?
    TSMCFG_MaxFrameSize       dd    ?
    TSMCFG_SystemFlags        dw    0
TSMConfigTable ends
```

TSMCFG_TableSize

This field contains the actual size of the TSM’s configuration table in bytes. The value of this field should not be confused with the number of bytes requested or copied (i.e., value in ECX).

TSMCFG_TableMajorVersion

This field contains the major version of the configuration table. The current major version is 1.

TSMCFG_TableMinorVersion

This field contains the minor version of the configuration table. The current minor version is 0.

TSMCFG_ModuleMajorVersion

This field contains the major version of the <TSM> binary (e.g., ETHERTSM.NLM).

TSMCFG_ModuleMinorVersion

This field contains the minor version of the <TSM> binary (i.e., ETHERTSM.NLM).

TSMCFG_ODISpecMajorVersion

This field contains the major version of the ODI Specification that this version of the <TSM> is written too. For example, if the version of the ODI specification is 3.31, the value of this field is 3.

TSMCFG_ODISpecMinorVersion

This field contains the minor version of the ODI Specification that this version of the <TSM> is written too. For example, if the version of the ODI specification is 3.31, the value of this field is 31.

TSMCFG_Reserved

This field is reserved and must be set to 0.

TSMCFG_MaxFrameSize

The value of this field represents the maximum frame size that the <TSM> supports.

TSMCFG_SystemFlags

The bits in this field are defined below.

Table 6.1 TSMCFG_SystemFlags

Offset	Name	Bytes
Bit 31	TSM_CFG_CLIENT_BIT	When set to 1 this bit indicates the TSM is running in a client environment. Either this bit or bit 30 will be set, but never both.
Bit 30	TSM_CFG_SERVER_BIT	When set to 1 this bit indicates the TSM is running in a server environment. Either this bit or bit 31 will be set, but never both.
Bit 0-29	Reserved	These bits are reserved.


<TSM>GetNextSend

Processor States	Entry State
	EBP pointer to the Adapter Data Space
	Interrupts are disabled
	Call at process or interrupt time
	Return State
	EBP pointer to the Adapter Data Space
	EBX pointer to the Frame Data Space
	ESI pointer to the next TCB to transmit if successful. This routine will decrement <i>MSMTxFreeCount</i> .
	ECX Padded packet length (Ethernet only)
	Zero Flag Set if successful; otherwise there are no TCBs queued or the adapter is currently using all of its transmit resources and cannot accept another packet (<i>MSMTxFreeCount</i> = 0).
	Interrupts are disabled

Description

This function retrieves the next ECB to be sent from the MSM's transmit waiting queue. It then builds a TCB and gives it to the HSM for transmission. If the send queue is empty, this function clears the zero flag and returns.

Most HSMs do not need to call this function, because the MSM checks for pending transmissions whenever *TxFreeCount* is not equal to zero on exit from an HSM function, and calls **DriverSend** when necessary.

Note  The **DriverSend** routine may use ECBs instead of TCBs by initializing the *DriverParameterBlock* variable *DriverSendWantsECBs* to a non-zero value (see Chapter 3). In this case, **<TSM>GetNextSend** will simply retrieve the next ECB to be sent (without building a TCB).

Example

```

DriverISR  proc
    .
    .
    .
    TransmitComplete;; EBP = Ptr to Adapter Data Space

    inc     [ebp].MSMTxFreeCount; Free adapter's transmit
           resource
    mov     [ebp].TxInProgress, 0; Clear transmit in progress flag

    ;*** Transmit Next Packet ***

    call    <TSM>GetNextSend; Get the next TCB from the
           queue
    jnz     NoSendsQueued; Jump if nothing to send
    call    DriverSend; Otherwise send the packet
    .
    .
    .
DriverISR  endp

```

<TSM>GetASMHSMIFLevel

Processor States	Entry State	
	Interrupts	can be in any state
	Call	at initialization time only
	Return State	
	EAX	assembly HSM Interface Level (currently 220)
	Interrupts	are preserved
	Preserved	all other registers

Description Bus Master HSM's that are modified to support physically addressed ECB fragment pointers, (*DriverSupportsPhysFrgs* bit = 1) need to call **<TSM>GetASMHSMIFLevel** since the modified HSM is incompatible with an older TSM. To be compatible with the modified HSM, the TSM level returned must be 220 (version 2.20) or greater.

Example

```
DriverInit Proc
.
.
.
;Fill in Driver Parameter Block Fields
.
.
.
if BusMaster
    call <TSM>GetASMHSMIFLevel
    cmp  eax, 220
    mov  ecx, eax
    lea  eax, LevelErrorMsg
    jb   DriverInitResetError;Jump if wrong TSM level
endif
.
.
.
DriverInit endp
```

<TSM>GetRCB



Note For RX-Net see *RXNetTSMGetRCB*

Processor States

Entry State

EBP	pointer to the Adapter Data Space
ESI	pointer to the received packet header (LookAhead buffer)
ECX	size of the received packet, -1 if size is unknown (such as pipelined adapters)
EAX	status of received packet for the Receive Monitor, 0 if unknown (such as pipelined adapters) (see DriverPromiscuousChange in Chapter 5)
Interrupts	can be in any state
Call	at process or interrupt time

Return State

Zero Flag	set if successful; otherwise an error occurred
ESI	pointer to the fragmented RCB if this call is successful
EDI	pointer to the fragment structure (points to the RCBFragmentCount field of the RCB)
EBX	number of bytes to skip over from the beginning of packet
ECX	number of bytes remaining to read
Interrupts	are disabled
Preserved	EBP

Description

This routine is called by the HSM to obtain a fragmented RCB for a packet that has been received by the adapter. Drivers that cannot handle fragmented receive buffers should obtain RCBs using either **MSMAllocateRCB** or **<TSM>ProcessGetRCB**.

<TSM>GetRCB uses a LookAhead process in which the packet's header information is previewed before an RCB is given to the driver. This way the

TSM can first verify that it wants the packet, before the driver transfers the entire packet from the adapter into an RCB.

The adapter's data transfer method governs how the LookAhead process is handled.

- If a programmed I/O adapter is being used, the HSM must transfer the packet's header information from the adapter into a buffer maintained for this purpose. The number of bytes to transfer is specified by the variable *MSMMaxFrameHeaderSize* described in Chapter 4. The HSM must set ESI to point the beginning of the LookAhead buffer before calling this routine.
- If a shared RAM (memory-mapped I/O) adapter is being used, the HSM can simply point ESI to the beginning of the packet buffer in shared RAM.

On entry to this routine, ESI must point to the packet's header information (the LookAhead buffer) and ECX must contain the size of the received packet. If the adapter is pipelined and the packet size is unknown, fill in ECX with -1. If the header verifies, the TSM will obtain an RCB and use the LookAhead information to fill in the *RCBReserved* fields before returning a pointer to the RCB in ESI.

After obtaining the RCB, the remainder of the packet must be transferred into the RCB fragment buffers. EBX is the offset from the beginning of the packet to start copying from and ECX contains the number of bytes in the packet left to read.

After the HSM has read the rest of the packet, it must return the RCB to the LSL using either the **<TSM>RcvComplete / MSMServiceEvents** combination or by using **<TSM>FastRcvComplete**. If the adapter is pipelined and called **<TSM>GetRCB** with ECX equal to -1, it should use either the **<TSM>RcvCompleteStatus/MSMServiceEvents** combination or use **<TSM>FastRcvCompleteStatus** to return the RCB along with the actual size of the packet and error status.

Note



If this routine returns an error completion code, the received packet must be discarded.

Bus Master Adapters

Bus Master devices require RCBs to be preallocated. Since this routine requires a LookAhead Buffer, it cannot be used to preallocate RCBs. The HSM can

preallocate RCBs using either **MSMAllocateRCB** or **MSMAllocateMultipleRCBs**.

Example


```

mov  ecx, [ebp].MSMMaxFrameHeaderSize; Build LookAhead
      buffer
lea   edi, [ebp].LookAheadBuffer
rep   insb
lea   esi, [ebp].LookAheadBuffer; Ptr to LookAhead buffer
mov   ecx, PacketSize; Size of the received packet
call  <TSM>GetRCB; Get an RCB
jnz   PacketNotAccepted; Jump if Error
.
.     (Copy remainder of the packet into the RCB)
.
call  <TSM>RcvComplete; Return RCB

```

TSM Procedures **6-13**

<TSM>ProcessGetRCB


Note  For RX-Net see *RXNetTSMRcvEvent*

Processor States	Entry State
	EBP pointer to the Adapter Data Space
	ESI pointer to the received packet's RCB
	ECX size of the received packet
	EAX status of received packet for the Receive Monitor (see DriverPromiscuousChange in Chapter 5)
	EDI maximum packet size for the new RCB
	Interrupts can be in any state
	Call at process or interrupt time
	Return State
	Zero Flag set if a new RCB was available
	ESI pointer to a new non-fragmented RCB (if the zero flag is set)
	Interrupts are disabled
	Preserved EBP

Description

The HSM calls this routine to process an RCB for a received packet and to preallocate a new nonfragmented RCB for the next packet. The received packet must have been copied into the *RCBDataBuffer*.

Use this routine if the RCB was preallocated using **MSMAllocateRCB** or **MSMAllocateMultipleRCBs**, or was obtained from a previous call to this routine. In either case, the *RCBReserved* fields have not been filled in, and therefore must be completed by the TSM.

Note  If the adapter/driver is ECB aware and has already filled in all required ECB fields as described in Chapter 4, the ECB should be returned for processing using **<TSM>RcvCompleteStatus/MSMServiceEvents** or **<TSM>FastRcvCompleteStatus**.

When this routine is called, the TSM examines the packet header information. If the header verifies, the *RCBReserved* fields are filled in and the RCB is directed to the Link Support Layer's holding queue to await processing. The TSM then obtains a new nonfragmented RCB, if one is available, and returns it to the driver. If the packet header is invalid, the RCB will be given back to the driver to be used again for another packet.

The HSM must eventually use the macro **MSMServiceEvents** which enables the RCB's Event Service Routine to complete the processing.

Ethernet

The HSM should start copying the packet from the 6 byte destination field of the media header into the *RCBDataBuffer* field of the RCB.

Token-Ring

The HSM should start copying the packet from the Access Control byte of the media header into the *RCBDataBuffer* field of the RCB.

FDDI

The HSM should start copying the packet from the Frame Control byte of the media header into the *RCBDataBuffer* field of the RCB.



For drivers that use DMA it may be helpful to note that *RCBFragmentOffset1* points to the physical address of the *RCBDataBuffer* (refer to figure 4.3 in Chapter 4).



For some busMaster implementations, you must set **RProtocolWorkspace** (defined in ODI.INC) to the number of bytes necessary to skip to the beginning of the packet. This value can be as high as 128 bytes for chips which have poor alignment capabilities. This field is normally part of the reserved space in the RCB definition and can only be used with this call for the purpose stated above.

Example


```

DriverInit  proc
    .
    .
    .
    mov     esi, [ebx].MLIDMaximumSize
    call    MSMAAllocateRCB; Preallocate first RCB & save.
    .
    .
    .
DriverInit  endp

DriverISR   proc
    .
    .
    .
ReceiveEvent:
    .
    .      (Copy packet into the RCBDataBuffer field of the
    .      preallocated RCB)
    .
    xor     eax, eax; Good packet
    mov     ecx, PacketSize; Size of received packet
    mov     edi, [ebx].MLIDMaximumSize; Maximum size for new
    .      RCB
    call    <TSM>ProcessGetRCB; Return RCB & get a new
    .      RCB
    .
    .
    .
DriverISR   endp

```


<TSM>FastProcessGetRCB

Note  For RX-Net see *RXNetTSMFastRcvEvent*

Processor States		Entry State
	EBP	pointer to the Adapter Data Space
	ESI	pointer to the received packet's RCB
	ECX	size of the received packet
	EAX	status of received packet for the Receive Monitor (see DriverPromiscuousChange in Chapter 5)
	EDI	maximum packet size for the new RCB
	Interrupts	can be in any state (but might be enabled during the call)
	Call	at process or interrupt time
		Return State
	Zero Flag	set if a new RCB was available
	ESI	pointer to a new nonfragmented RCB (if the zero flag is set)
	Interrupts	are disabled
	Preserved	EBP

Description <TSM>FastProcessGetRCB is identical to <TSM>ProcessGetRCB with the exception that before this routine returns, the RCB's Event Service Routine is called to complete the processing. <TSM>ProcessGetRCB used in conjunction with MSMSERVICEEVENTS will perform the same task.

During the RCB's Event Service Routine, the interrupts might be enabled and all registers could be destroyed. The HSM must preserve any needed registers before calling <TSM>FastProcessGetRCB. If having the interrupts enabled is undesirable, the driver should use the <TSM>ProcessGetRCB procedure and wait until the conclusion of the receive routine before servicing events.

Important  This routine calls the RCB's Event Service Routine during which the interrupts might be enabled and all registers could be destroyed.

Note



For drivers that use DMA it may be helpful to note that *RCBFragmentOffset1* points to the physical address of the *RCBDataBuffer* (refer to figure 4.3 in Chapter 4).

Note



For some busMaster implementations, you must set **RProtocolWorkspace** (defined in ODI.INC) to the number of bytes necessary to skip to the beginning of the packet. This value can be as high as 128 bytes for chips which have poor alignment capabilities. This field is normally part of the reserved space in the RCB definition and can only be used with this call for the purpose stated above.

Example

```

DriverInit  proc
    .
    .
    .
    mov  esi, [ebx].MLIDMaximumSize
    call  MSMAAllocateRCB; Preallocate first RCB & save.
    .
    .
    .
DriverInit  endp

DriverISR   proc
    .
    .
    .
ReceiveEvent:
    .
    .      (Copy packet into the RCBDataBuffer field of the
    .      preallocated RCB)
    .
    xor  eax, eax; Good packet
    mov  ecx, PacketSize; Size of received packet
    mov  edi, [ebx].MLIDMaximumSize; Maximum size for new
    .      RCB
    call  <TSM>FastProcessGetRCB; Return RCB, service
    .      events, and get a new RCB
    .
    .
DriverISR   endp

```

<TSM>RcvComplete

Processor States

Entry State

EBP	pointer to the Adapter Data Space
ESI	pointer to the received packet's RCB
Interrupts	are disabled
Call	at process or interrupt time

Return State

Interrupts	are disabled
Preserved	EBP, ESI, and EDI.

Description

The HSM calls **<TSM>RcvComplete** to direct a completed RCB to the Link Support Layer's holding queue to await processing. Use this routine if the RCB was obtained using the **<TSM>GetRCB** procedure and the received packet has been copied into the RCB receive buffer(s).

Note



Pipelined adapter drivers that previously called **<TSM>GetRCB** with ECX = -1, as well as ECB aware adapters should call **<TSM>RcvCompleteStatus** instead of **<TSM>RcvComplete**. Drivers that call **<TSM>GetRCB** with ECX set to equal the frame size should still use **<TSM>RcvComplete**.

When an RCB is queued using this routine, the HSM must eventually use the macro **MSMServiceEvents** to call the RCB's Event Service Routine and complete the processing.

RX-Net

If an RX-Net adapter/driver is ECB aware (see Chapter 4), it is responsible for handling packet reconstruction and fragmentation. Once the packet is reconstructed, the HSM must set the second byte of the *DriverWorkspace* field to one before calling this routine.

Example

```
mov      ecx, [ebp].MSMMaxFrameHeaderSize; Build the
          LookAhead buffer
lea      edi, [ebp].LookAheadBuffer
rep      insb

lea      esi, [ebp].LookAheadBuffer; Ptr to LookAhead buffer
mov      ecx, PacketSize; Size of the received packet
call     <TSM>GetRCB; Get an RCB
jnz      PacketNotAccepted; Jump if Error
.
.        (Copy remainder of the packet into the RCB)
.
call     <TSM>RcvComplete; Return the RCB
```


<TSM>RcvCompleteStatus

Processor States	Entry State
	EBP pointer to the Adapter Data Space
	ESI pointer to the received packet's RCB
	EAX status of received packet for Receive Monitor
	ECX size of received packet
	Interrupts are disabled
	Call at process or interrupt time
	Return State
	Interrupts are disabled and were not enabled
	Preserved EBP, ESI, and EDI.

Description

The HSM calls **<TSM>RcvCompleteStatus** to allow the TSM to fill in proper packet length fields of the RCB, record the error status and direct the completed RCB to the Link Support Layer's holding queue to await processing. Use this routine if the RCB was obtained by a pipelined adapter using **<TSM>GetRCB** with ECX equal to -1 or by an ECB aware adapter using **MSMAllocateRCB** or **MSMAllocateMultipleRCBs**.

When an RCB is queued using this routine, the HSM must eventually use the macro **MSMServiceEvents** to call the RCB's Event Service Routine and complete the processing.

Example

(This example is for pipelined adapters; see

<TSM>**FastRcvCompleteStatus** for ECB-aware example)

```

mov     ecx, -1; don't know packet size yet
xor     eax, eax; don't know error status
lea     esi, [ebp].LookAheadBuffer; pass what we have so
        far
call    <TSM>GetRCB; get an RCB
jnz     PacketNotAccepted; jump if error
.
. (Copy remainder of the packet into the RCB)
.
mov     eax, [ebp].ErrorStatus; get known error status
mov     ecx, [ebp].PacketSize; get known packet size
call    <TSM>RcvCompleteStatus

```

<TSM>FastRcvComplete


Processor States	Entry State	
	EBP	pointer to the Adapter Data Space
	ESI	pointer to the received packet's RCB
	Interrupts	are disabled (but might be enabled during the call)
	Call	at process or interrupt time
	Return State	
	Interrupts	are disabled
	Preserved	assume all registers are destroyed.

Description


<TSM>FastRcvComplete is identical to <TSM>RcvComplete with the exception that before this routine returns, the RCB's Event Service Routine is called to complete the processing. Using <TSM>RcvComplete in conjunction with MSMSERVICEEVENTS will perform the same task.

During the RCB's Event Service Routine, the interrupts might be enabled and all registers could be destroyed. The HSM must preserve any needed registers before calling <TSM>FastRcvComplete. If having the interrupts enabled is undesirable, the driver should use the <TSM>RcvComplete procedure and wait until the conclusion of the receive routine before servicing events.

- Important



This routine calls the RCB's Event Service Routine during which the interrupts might be enabled and all registers could be destroyed.
- Note



Pipelined adapter drivers that previously called <TSM>GetRCB with ECX = -1, as well as ECB aware adapters should call <TSM>FastRcvCompleteStatus instead of <TSM>FastRcvComplete. Drivers that call <TSM>GetRCB with ECX set to equal the frame size should still use <TSM>FastRcvComplete.

Example

```
mov    ecx, [ebp].MSMMaxFrameHeaderSize; Build the
        LookAhead buffer
lea    edi, [ebp].LookAheadBuffer
rep    insb
lea    esi, [ebp].LookAheadBuffer; Ptr to LookAhead buffer
mov    ecx, PacketSize; Size of the received packet
call   <TSM>GetRCB; Get an RCB
jnz    PacketNotAccepted; Jump if Error
.
.      (Copy remainder of the packet into the RCB)
.
call   <TSM>FastRcvComplete; Return RCB & service
        events
```

<TSM>FastRcvCompleteStatus

Processor States

Entry State

EBP	pointer to the Adapter Data Space
ESI	pointer to the received packet's RCB
EAX	status of received packet for Receive Monitor (see DriverPromiscuousChange in Chapter 5)
ECX	size of received packet
Interrupts	are disabled
Call	at process or interrupt time

Return State

Interrupts	are disabled and may have been enabled
Preserved	Assume all registers are destroyed

Description

<TSM>**FastRcvCompleteStatus** is identical to <TSM>**RcvCompleteStatus** with the exception that before this routine returns, the RCB's Event Service Routine is called to complete the processing. Using <TSM>**RcvCompleteStatus** in conjunction with **MSMServiceEvents** will perform the same task.

During the RCB's Event Service Routine, the interrupts might be enabled and all registers could be destroyed. The HSM must preserve any needed registers before calling <TSM>**FastRcvCompleteStatus**. If having the interrupts enabled is undesirable, the driver should use the <TSM>**RcvCompleteStatus** procedure and wait until the conclusion of the receive routine before servicing events.

Example

(This examples is for ECB-aware adapters; see <TSM>**RcvCompleteStatus** for pipelined adapters example.)

```

mov     esi, [ebx].MLIDMaximumSize; ECB Size expected
call    MSMAAllocateRCB; get an RCB
jnz     NoRCBsAvailable; jump if error
.
.      (Copy remainder of the packet into the RCB)
.
mov     eax, [ebp].ErrorStatus; get packet error status
mov     ecx, [ebp].PacketSize; get known packet size
call    <TSM>FastRcvCompleteStatus

```

<TSM>RegisterHSM

Processor States

Entry State

ESI	pointer to the DriverParameterBlock structure
Interrupts	can be in any state
Call	at initialization time only

Return State

EAX	zero if successful; otherwise EAX points to an error message that the driver must print using MSMPrintString before returning to the operating system with EAX non-zero.
EBX	pointer to the Frame Data Space
Interrupts	are disabled
Preserved	all other registers are destroyed
Zero Flag	set if successful; otherwise an error occurred

Description

The HSM's **DriverInit** routine must call <TSM>**RegisterHSM** with a pointer to its DriverParameterBlock structure in ESI. Before calling this routine, **DriverInit** must save the value of the stack pointer in the *DriverStackPointer* field of the DriverParameterBlock after pushing the C registers EBP, EBX, ESI, and EDI. This routine then calls the MSM which performs the following tasks:

- copies the parameter block into local data space
- processes driver firmware variables
- allocates the Frame Data Space
- copies the driver configuration table into the Frame Data Space
- parses information derived from the linker definition file
- places LSL's maximum packet size in the configuration table
- initializes screen ID used for **MSMPrintString** procedures

Example

```

DriverInit  proc
    Cpush; macro to save "C" registers
    mov  DriverStackPointer, esp; Fill in stack pointer
    lea  esi, DriverParameterBlock; Get ptr to Parameter
        Block
    call  <TSM>RegisterHSM; Get a Frame Data Space
    jnz  DriverInitError; Jump if error
    .
    .
    .
    xor  eax, eax; Successful return with EAX=0
    Cpop ; Restore "C" registers
    ret

DriverInitError:
    mov  esi, eax; ESI=EAX= ptr to error msg.
    call  MSMPrintString; Print the Message
    Cpop ; Restore "C" Registers
    ret  ; Return (EAX is nonzero on
        ; errors)
DriverInit  endp

```


<TSM>SendComplete

Processor States

Entry State

EBP	pointer to the Adapter Data Space
ESI	pointer to the Transmit Control Block (TCB)
Interrupts	are disabled
Call	at process or interrupt time

Return State

Interrupts	are disabled
Preserved	EBP

Description

This procedure is called to release a TCB after a packet has been transmitted. It can be called by **DriverISR** after a transmit complete interrupt or by the **DriverSend** routine before the actual transmission is complete (a lying send), as long as all packet data has been transferred into the adapter's transmit buffer and access to the TCB is no longer required.

This procedure returns the packet's TCB to the MSM's unused TCB queue and directs the underlying Transmit ECB to the Link Support Layer's service queue.

The HSM must eventually use the macro **MSMServiceEvents** which calls the ECB's Event Service Routine. Typically, if the **DriverSend** routine was called to transmit the next packet after a send complete interrupt, then the interrupt service routine must invoke **MSMServiceEvents**.



The **DriverSend** routine may use ECBs instead of TCBs by initializing the DriverParameterBlock variable *DriverSendWantsECBs* to a non-zero value (see Chapter 3). In this case, **<TSM>SendComplete** will simply direct the ECB to the LSL's service queue.

Example

```
DriverSend proc
    .
    .      (send the packet to the NIC)
    .
    cmp  InDriverISR, 0
    jnz  <TSM>SendComplete
    jmp  <TSM>FastSendComplete

DriverSend  endp
```

<TSM>FastSendComplete

Processor States	Entry State	
	EBP	pointer to the Adapter Data Space
	ESI	pointer to the Transmit Control Block (TCB)
	Interrupts	are disabled (but might be enabled during the call)
	Call	at process or interrupt time
	Return State	
	Interrupts	are disabled
	Preserved	all registers are destroyed.
	<hr/>	

Description

<TSM>**FastSendComplete** is identical to <TSM>**SendComplete** except that before this routine returns, the TCB's Event Service Routine is called to notify the upper layers that the transmission is complete. Using the <TSM>**SendComplete** / **MSMServiceEvents** combination will perform the same task.

During the TCB's Event Service Routine, the interrupts might be enabled and all registers could be destroyed. The HSM must preserve any needed registers before calling <TSM>**FastSendComplete**.

Example

```
DriverSend proc
.
.    (send the packet to the NIC)
.
cmp  InDriverISR, 0
jnz  <TSM>SendComplete
jmp  <TSM>FastSendComplete

DriverSend endp
```


<TSM>UpdateMulticast

Processor States	Entry State
	EBP pointer to the Adapter Data Space
	Interrupts are disabled and remain disabled
	Call at process or interrupt time
	Return State
	Interrupts are disabled
	Preserved EBX and EBP

Description

When <TSM>**UpdateMulticast** is called it passes the current multicast table (maintained by the TSM) to the HSM's **DriverMulticastChange** routine. This allows the driver to update the adapter's multicast address registers.

This routine is called by internal TSM procedures each time the multicast addresses are added to or deleted from the MSM's multicast table. This routine can also be called by the driver during the HSM's **DriverReset** routine.

Note  RX-Net does not support multicast addressing. This routine is not available if the RXNetTSM module is used.

Refer to the sections covering the following flags and variables for more information on multicast addressing:

- Bit 3 of the *MLIDModeFlags* is used to indicate whether or not multicast addressing is supported.
- Bits 9 and 10 of the *MLIDFlags* must be set appropriately to reflect the multicast mechanism or format used by the adapter/driver.
- The DriverParameterBlock variable, *DriverMaxMulticast*, must be set to reflect the maximum number of multicast addresses the adapter can handle.

Example

```
DriverResetproc
.
.
.
call<TSM>UpdateMulticast
.
.
.
DriverResetendp
```

RXNetTSMGetRCB

Processor States	Entry State
	EBP pointer to the Adapter Data Space
	ESI pointer to the LookAhead Buffer
	Interrupts are in any state
	Call at process or interrupt time
	Return State
	Zero flag set if successful; otherwise an error occurred
	ESI pointer to the RCB if this call is successful
	EDI pointer to the RCB fragment structure (points to the RCBFragmentCount field of the RCB)
	EBX contains the offset in the card's buffer from which to start copying data
	ECX number of bytes remaining to read
	Interrupts are disabled
	Preserved EBP

Description

This routine is normally used for programmed I/O adapters.

RXNetTSMGetRCB uses a LookAhead process in which the packet's header information is previewed before an RCB is given to the driver. This way the TSM can first verify that it wants the packet, before the driver transfers the entire packet from the adapter into an RCB.

The LookAhead process requires the HSM to build a buffer containing the packet's header information as shown in Figure 6.1 on the following page. The number of bytes required for the buffer is specified by the variable *MSMMaxFrameHeaderSize* described in Chapter 4. The HSM must set ESI to point to the beginning of the LookAhead buffer before calling this routine. If the header verifies, this routine returns a pointer to an RCB.

At this point, the HSM must transfer the remainder of the packet into the RCB fragment structure. Since other fragments of a split packet may have already been copied into the RCB buffers, the HSM must perform the following operations.

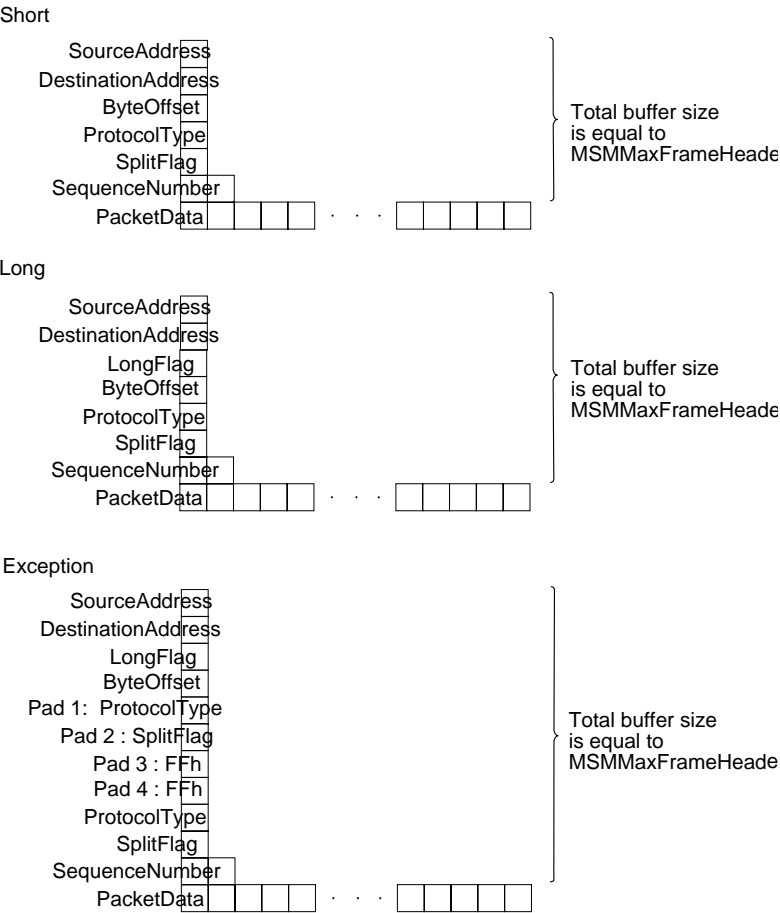
- The dword value at [EDI – 4] indicates the number of bytes currently in the RCB fragment buffers. This value can be used along with the *RCBFragmentLength* fields to determine where in the RCB fragment structure to begin copying the packet.
- Once the position is located, the HSM transfers the rest of the packet into the RCB fragment structure. (EBX is the offset from the beginning of the card's buffer to start copying from and ECX is the number of bytes left to read.)
- Update the number of bytes currently in the RCB fragment buffers by adding ECX bytes to the dword value at [EDI – 4].

After the HSM completes the above tasks, it must return the RCB using either the <TSM>**RcvComplete** / **MSMServiceEvents** combination or by using <TSM>**FastRcvComplete**.



Using **RXNetTSMGetRCB** does not provide 100% support to a receive monitor.

Figure 6-1 Format of the RX-Net LookAhead Buffer



Example


```

. (Build the LookAheadBuffer)
.
lea esi,[ebp].LookAheadBuffer
call RXNetTSMGetRCB ; Get an RCB
jnz NoRCB ;Jump if there is an error
.
. (Determine the current fragment buffer position)
. (Transfer the rest of the packet into the RCB)
.
add [edi-4], ecx
call RXNetTSMRcvComplete; Return the RCB

```

TSM Procedures **6-37**

RXNetTSMRcvEvent

Note  This procedure applies to RX-Net only

Processor States	Entry State	
	EBP	pointer to the Adapter Data Space
	ESI	pointer to the received packet
	Interrupts	are in any state
	Call	at process or interrupt time
	Return State	
	Zero Flag	set if successful
	Interrupts	are disabled
	Preserved	EBP

Description

RXNetTSMRcvEvent is only available to HSMs that use RX-Net shared RAM cards and that use the RXNetTSM module. The only action the HSM takes when a packet is received is to pass this routine a pointer to the packet. If the packet is wanted, the TSM copies the entire packet into an RCB, completing packet reception.

The HSM must eventually use the macro **MSMServiceEvents** which enables the RCB's Event Service Routine to complete the processing.


RX-Net cards that do not support shared RAM should either:

- Use the **RXNetTSMGetRCB / <TSM>RcvComplete** combination to receive packets. This method does not provide 100% support to a receive monitor.
- Copy the entire packet from the adapter into a buffer and call this routine with a pointer to that buffer in ESI. This method is the only way to provide 100% support to a receive monitor.

Example

mov	esi, [ebp].CurrRxPage; ESI -> Current Rx Page
xor	[ebp].CurrRxPage, 0200h; Toggle to the next page
call	RXNetTSMRcvEvent; Pass the packet to MSM
jmp	ISRExit; Finished receiving the packet

RXNetTSMFastRcvEvent


Note  This procedure applies to RX-Net only.

Processor States		Entry State
	EBP	pointer to the Adapter Data Space
	ESI	pointer to the received packet
	Interrupts	are in any state
	Call	at process or interrupt time
		Return State
	Zero Flag	set if successful
	Interrupts	are disabled
	Preserved	EBP

Description

RXNetTSMFastRcvEvent is identical to **RXNetTSMRcvEvent** except that before this routine returns, the RCB's event service routine is called to complete the processing. Using **RXNetTSMGetRCB** / **RXNetTSMRcvEvent** in conjunction with **MSMServiceEvents** will perform the same task.

During the RCB's Event Service Routine, the interrupts might be enabled and all registers could be destroyed. The HSM must preserve any needed registers before calling **RXNetTSMFastRcvEvent**. If having the interrupts enabled is undesirable, the driver should use the **RXNetTSMRcvEvent** procedure and wait until the conclusion of the receive routine before servicing events.

Important  This routine calls the RCB's Event Service Routine, during which the interrupts might be enabled and all registers could be destroyed.

Example

```
mov    exi, [ebp].CurrRxPage; location of received packet
call   RXNetTSMFastRcvEvent
```

chapter **7** ***MSM Procedures and Macros***

Introduction

This chapter describes the MSM procedures and macros provided as tools for HSM developers. These MSM procedures, along with the topology specific procedures described in Chapter 6, manage the primary details of interfacing the HSM to the Link Support Layer. The procedures and macros in this chapter are media independent and handle generic initialization and run-time issues. The macros included in this section are defined in the MSM.INC file.

Netware Bus Interface

Overview

This section lists the MSM API calls that enable the NetWare Bus Interface (NBI) as it pertains to this specification. The MSM autoloads the NetWare Bus Interface, NBI.NLM.

The additional bus support includes PCI, ISA Plug and Play, and PC Card (PCMCIA) support. The MSM functions described in this chapter make it possible to access and support these new bus architectures.

Important



PCI and any other new bus support you choose to implement in your driver **MUST** be implemented by using the new MSM calls listed in this chapter. Any other implementations including direct calls to the NBI are not allowed.

The MSM API calls described in this chapter that pertain to NBI support are listed below.

MSMGetAlignment
 MSMGetBusInfo
 MSMGetBusSpecificInfo
 MSMGetBusType
 MSMGetCardConfigInfo
 MSMGetInstanceNumber
 MSMGetInstanceNumberMapping
 MSMGetUniqueIdentifier
 MSMGetUniqueIdentifierParameters
 MSMRdConfigSpace8
 MSMRdConfigSpace16
 MSMRdConfigSpace32
 MSMScanBusInfo
 MSMSearchAdapter
 MSMWrtConfigSpace8
 MSMWrtConfigSpace16
 MSMWrtConfigSpace32

Important



Since the **MSMGetHardwareBusType** call has been removed, old **EISA** and **MCA** drivers that used this call must be updated and use **MSMSearchAdapter** instead.

7-2 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Bus Architecture

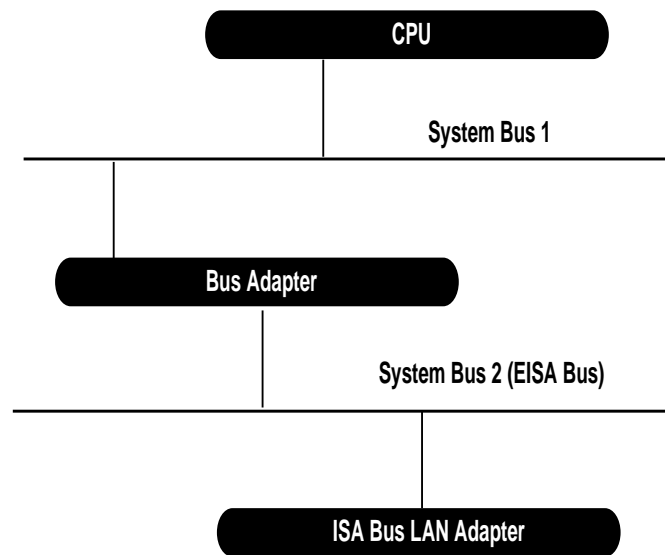
A bus architecture is one or more related address spaces and a set of characteristics within those address spaces. For example, an IBM PC ISA address space consists of the following:

- a 16-bit memory address space
- a 16-bit I/O address space
- a defined set of interrupts with their means of generation and means of dismissal
- a set of DMA channels with means of starting and completing their operations
- etc.

Multiple Bus Platforms

The following figure shows an example of a multiple bus platform.

Figure 7-1 Multiple Bus Platform Example



MSM Procedures and Macros **7-3**

Because of the potentially differing bus architectures and the intervening bus adapter, an MLID executing on the CPU cannot assume that it can directly access and control the programmable interrupt controller or DMA controller the same way it might on an IBM PC. In fact, these functions may be implemented using hardware completely unlike that used in the IBM PC.

The MLID cannot even assume that it knows what memory addresses to read or write in order to communicate with its adapter. In the above figure, for example, the intervening bus adapter can have these addresses in the System Bus 2 memory address space mapped to some other address in the System Bus 1 address space or can have them not mapped at all.

7-4 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

MSMAlertFatal

Processor States

Entry State

EBP	pointer to Adapter Data Space
ECX	possible argument #1
EDX	possible argument #2
ESI	pointer to null terminated error message
Interrupts	can be in any state, but will be disabled during the call
Call	at process or interrupt time

Return State

Interrupts	are in the same state as when the routine was called
Preserved	EBX and EBP

Description

The HSM can call **MSMAlertFatal** during regular operation (run-time) to notify the operating system of driver hardware or software problems. An error severity level of “fatal” will be reported with the developer-provided error message. This routine will not relinquish control to other procedures during execution.

The possible arguments #1 and #2 above are used here the same way in which they are used in the C-language **printf** routine. If there are no format specifications in the string, ECX and EDX are ignored.

This routine also supports an additional string format. If the string is preceded by a word size error number in the range of 100-999, the MSM will print the driver name, the platform name (NW for NetWare), the decimal error number, and the instance of the board, before printing the specified string. (See the *Standard MLID Message Definitions* supplement for a listing of standard messages.)

Example

```

ErrorMessage    dw    105
                  db    "Board did not respond to multicast update.",0
                  .
                  .
                  .
lea             ESI, ErrorMessage
call            MSMAlertWarning

```

MSM Procedures and Macros 7-5

The example above would output the following message if the adapter is an NE2000 and was the first NE2000 registered:

NE2000-NW-105-Adapter 1:Board did not respond to multicast update.

MSMAlertWarning

Processor States

Entry State

EBP	pointer to Adapter Data Space
ECX	possible argument #1
EDX	possible argument #2
ESI	pointer to null terminated error message
Interrupts	can be in any state, but will be disabled during the call
Call	at process or interrupt time

Return State

Interrupts	are in the same state as when the routine was called
Preserved	EBX and EBP

Description

The HSM can call **MSMAlertWarning** during regular operation (run-time) to notify the operating system of driver hardware or software problems. An error severity level of “warning” will be reported with the developer-provided error message. This routine will not relinquish control to other procedures during execution.

The possible arguments #1 and #2 above are used here the same way in which they are used in the C-language **printf** routine. If there are no format specifications in the string, ECX and EDX are ignored.

This routine has added functionality which supports an additional string format. If the string is preceded by a word size error number in the range of 100-999, the MSM will print the driver name, the platform name (NW for NetWare), the decimal error number, and the instance of the board, before printing the specified string. (See the *Standard MLID Message Definitions* supplement for a listing of standard messages.)

Example

```

ErrorMessage    dw    105
                  db    "Board did not respond to multicast update.",0
                  .
                  .
                  .
lea             ESI, ErrorMessage
call            MSMAlertWarning

```

MSM Procedures and Macros **7-7**

The example above would output the following message if the adapter is an NE2000 adapter and was the first NE2000 registered:

NE2000-NW-105-Adapter 1: Board did not respond to multicast update.

7-8 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

MSMAlloc

Processor States

Entry State

EBP	pointer to the Adapter Data Space
EAX	number of bytes of memory to allocate
Interrupts	can be in any state (but might be enabled during the call)
Call	at process time only

Return State

EAX	pointer to the allocated buffer. (zero = failure)
Interrupts	are in the same state as when the routine was called
Preserved	EBX, EBP, ESI, and EDI

Description

The HSM may use this call to allocate memory at process time. **MSMAlloc** returns a pointer to the allocated buffer in EAX. If the routine was unsuccessful, EAX will be zero. It is the responsibility of the HSM to return this buffer at shutdown using **MSMFree**.

If the *DriverParameterBlock* variable, *DriverNeedsBelow16Meg*, was initialized to any nonzero value (see chapter 3), the MSM will allocate memory below the 16 megabyte boundary.

Example

```
mov    eax, UserBufferSize
call   MSMAlloc
or     eax, eax
jz     ErrorAllocatingBuffer
```

MSMAllocateMultipleRCBs

Processor States

Entry State

ECX	number of RCBs to allocate
EBP	pointer to Adapter Data Space
ESI	set to MLIDMaximumSize
Interrupts	can be in any state
Call	at process or interrupt time

Return State

ECX	number of RCBs allocated
ESI	logical ptr to first RCB
EDI	physical ptr to first RCB
Flags	zero flag is set according to EAX
Interrupts	are disabled
Preserved	EBX & EBP

Completion Code in EAX

SUCCESSFUL	At least one RCB was allocated.
OUT_OF_RESOURCES	No RCBs available.

Description

This procedure is intended for high speed drivers that require a pool of free RCBs available.

This procedure call is similar to **MSMAllocateRCB** except that more than one RCB may be allocated at a time. The RCBs returned will be non-fragmented. If no RCBs are available the MSM will increment NoECBAvailableCount statistics counter. Each RCB is linked to the next using the ECB fields Link for a forward pointer to next RCB by logical address and BLink for a forward pointer to next RCB by physical address. Link and BLink are located in RCBDriverWS. See Chapter 4 Receive ECBs vs RCBs.

If fewer RCBs are available than were requested, all available RCBs will be allocated. On return ECX will always match the actual number of RCBs allocated.

Important



Do not use if DriverNeedsBelow16Meg flag is set in the DriverParameterBlock.

This procedure can not be used by RX-Net drivers.

Example

```
;ebx = ptr to FrameDataSpace, ebp=ptr to AdapterDataSpace

mov  esi, [ebx].MLIDMaximumSize    ; Esc=MaxPacketSize
mov  ecx, NUMBER_TO_ALLOCATE      ; ECX =Number of RCBs to
                                allocate
call  MSMAAllocateMultipleRCBs     ; attempt to get RCBs
jnz   Unable to allocate RCHGBS    ; jump if unsuccessful
```

MSMAllocPages

Processor States	Entry State	
	EAX	number of bytes of memory to allocate
	Interrupts	can be in any state
	Call	at process time only
	Return State	
	EAX	pointer to the allocated buffer. (zero = failure)
	Interrupts	are in the same state as when the routine was called
	Preserved	EBX, EBP, ESI, and EDI

Description	The HSM may use this call to allocate a memory buffer on a 4K page boundary at process time. MSMAllocPages returns a pointer to the allocated buffer in EAX. If the routine was unsuccessful, EAX will be zero. The HSM must return this buffer at shutdown using MSMFreePages .	
	If the <i>DriverParameterBlock</i> variable, <i>DriverNeedsBelow16Meg</i> , was initialized to any nonzero value (see chapter 3), the MSM will allocate memory below the 16 megabyte boundary.	
Example	mov	eax, UserPageBufferSize
	call	MSMAllocPages
	or	eax, eax
	jz	ErrorAllocatingBuffer

MSMAllocateRCB

Processor States

Entry State

EBP	pointer to the Adapter Data Space
ESI	packet size including all the headers if known; otherwise use the maximum packet size.
Interrupts	can be in any state
Execute	at process or interrupt time

Return State

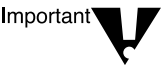
ESI	logical pointer to an RCB (non-fragmented)
EDI	physical pointer to an RCB (non-fragmented)
Flags	zero flag is set if routine is successful
Interrupts	are disabled
Preserved	all registers are preserved except EAX

Description

The HSM uses **MSMAllocateRCB** to allocate an RCB for a packet it has received or to preallocate an RCB for a packet it will be receiving. The RCB returned will be non-fragmented (see Chapter 4) and will be large enough to hold the received packet frame. The length passed in register ESI must also include the length of all protocol and hardware headers. If an RCB is not available, the MSM will increment the **NoECBAvailableCount** statistics counter and the packet must be discarded.

HSMs that support bus-mastering DMA adapters should use this routine or **MSMAllocateMultipleRCBs** to preallocate RCBs. In this case, the HSM must set ESI to the maximum packet size specified by the *MLIDMaximumSize* field of the configuration table before using **MSMAllocateRCB**.

After the adapter has copied the packet into the *RCBDataBuffer* field of the RCB, the HSM should use either **<TSM>ProcessGetRCB** or **<TSM>FastProcessGetRCB** to return the RCB to the MSM. If the adapter is ECB aware and has previously filled in all the RCB fields according to the ODI specification, the HSM should call **<TSM>RcvComplete** or **<TSM>FastRcvComplete**.



If the *DriverParameterBlock* variable, *DriverNeedsBelow16Meg*, was initialized to any nonzero value (see chapter 3), the MSM will allocate the RCB in memory below the 16 megabyte boundary.

Ethernet

The HSM should start copying the packet from the 6 byte destination field of the media header into the *RCBDataBuffer* field of the RCB.

Token-Ring

The HSM should start copying the packet from the Access Control byte of the media header into the *RCBDataBuffer* field of the RCB.

FDDI

The HSM should start copying the packet from the Frame Control byte of the media header into the *RCBDataBuffer* field of the RCB.

Example

```
; ebx = ptr to Frame Data Space

mov esi,[ebx].MLIDMaximumSize    ; ESI = Max Packet size
call MSMAllocateRCB              ; Get an RCB
jnz UnableToAllocateRCB          ; Jump if unsuccessful
```

MSMCancelTimer

Processor States

Entry State

EBP	pointer to Adapter Data Space
ESI	pointer to TIMER_STRUCTURE for timer to cancel
Interrupts	can be in any state, but are disabled during the call
Call	at process or interrupt time

Return State

Interrupts	are unchanged
Preserved	EBX & EBP

Completion Code in EAX

SUCCESSFUL	Timer was successfully canceled.
BAD_PARAMETER	A bad parameter was set in the TIMER_STRUCTURE.
BAD_COMMAND	Timer was not active.

Description

This procedure is called by the HSM to cancel a timer scheduled using **MSMScheduleTimer**.

MSMScheduleTimer and **MSMCancelTimer** are useful for starting and stopping one shot timers used for error detection such as transmit timeouts. Under normal processing the timer would never expire and the driver's timeout procedure would never be called.

Example

```
Driver Shutdown Proc
.
.
.
mov esi, TimerStructurePtr
call MSMCancelTimer
.
.
.
```

MSMDeRegisterResource

Processor States

Entry State

EBP	pointer to Adapter Data Space
EDX	pointer to an <i>ExtraConfig</i> structure that contains the resource(s) to be deregistered. This pointer must be the same pointer used to register the resources in MSMRegisterResources .
ESI	pointer to an ECB whose ESR is called if MSMDeRegisterResource returns RESPONSE_DELAYED. If NULL, BAD_PARAMETER will be returned.
Interrupts	can be in any state
Call	at process or interrupt time

Return State

Flags	set according to EAX.
Interrupts	are unchanged.
Preserved	all registers except EAX.

Completion Code in EAX

SUCCESSFUL	The resources contained in the <i>ExtraConfig</i> parameter were successfully deregistered.
BAD_PARAMETER	An input parameter was invalid or NULL. The <i>ExtraConfig</i> pointer was not found in the list of extra config pointers used in calls to MSMRegisterResource .
FAIL	The adapter was not in a shutdown state before the call was made.
ITEM_NOT_PRESENT	The resources to be deregistered have not previously been registered.
RESPONSE_DELAYED	The operation of deregistering resources could not be completed at the present time. An asynchronous process will be scheduled to complete the operation at a later time.

MSM Procedures and Macros **7-17**

Description

Allows a HSM to deregister resources registered with **MSMRegisterResource**.

Note



If **MSMRegisterMLID** has been called, the adapter must be shutdown using **MSMShutdownMLID** before **MSMDeRegisterResource** is called.

MSMDeRegisterResource will deregister those resources found in *ExtraConfig*'s substructure *IOConfig*. The resources must previously have been registered through **MSMRegisterResource** using the same *ExtraConfig* pointer.

If **MSMDeRegisterResource** cannot complete the operation at the present time, an asynchronous process will be scheduled to complete the operation later. Once the asynchronous operation is complete, the asynchronous ECB's ESR routine will be called to report the final return value of the operation. The return value will be stored in the asynchronous ECB's *ECB_Status* field.

Upon successful return from **MSMDeRegisterResource** or from the asynchronous process, the HSM is responsible for putting the adapter in a functional state. If additional resources of an *ExtraConfig* nature are required, the HSM must call **MSMRegisterResource** to register the additional resources.

MSMDriverRemove

Processor States	Entry State	
	EAX	<i>DriverModuleHandle</i> from the <i>DriverParameterBlock</i> structure
	Interrupts	can be in any state
	Call	at process time only
	Return State	
	EAX	is preserved
	Interrupts	are unchanged

Description This routine is called by the HSM's **DriverRemove** procedure to deregister the driver and return all driver resources. **MSMDriverRemove** will call the HSM's **DriverShutdown** routine before returning.

Example

```
DriverRemove      proc
    Cpush          ; Macro to save "C" registers
    mov     eax, DriverModuleHandle ;Get Module Handle
                                ;from Parameter Block
    call    MSMDriverRemove ;Deregister the driver
    Cpop          ;Restore "C" registers
DriverRemove      endp
```

MSMEnablePolling

Processor States	Entry State	
	EBP	pointer to the Adapter Data Space
	Interrupts	can be in any state
	Call	at process or interrupt time (usually called during initialization)
	Return State	
	EAX	zero if successful; otherwise EAX points to an error message that the driver must print using MSMPrintString before returning to the operating system with EAX non- zero.
	Zero Flag	set if successful; otherwise an error occurred.
	Interrupts	are unchanged
	Preserved	EBX and EBP

Description

If the HSM's board service routine is poll-driven, this routine can be used during **DriverInit** to enable the operating system to periodically call **DriverPoll**. The **DriverPoll** routine polls the adapter to determine if any send or receive events have occurred.

This routine will not relinquish control to other procedures during execution.

Example

```
DriverInit      proc
.
.
.
call    MSMEnablePolling      ; Enable DriverPoll
jnz     EnablePollingError
.
.
.
DriverInit      endp
```


MSMFree

Processor States

Entry State

EBP	pointer to the Adapter Data Space
EAX	pointer to the buffer
Interrupts	can be in any state
Call	at process time

Return State

Interrupts	are unchanged
Preserved	EBX, EBP, ESI, and EDI

Description

The HSM must use this routine to return any memory allocated using **MSMAlloc** before the driver is permanently shutdown. If the driver is being permanently shutdown, the HSM's **DriverShutdown** routine would have been called with ECX equal to zero.

Example

```
DriverShutdown      proc
    .
    .
    .
    or               ecx,ecx
    jnz              PartialShutdown
    mov              eax,UserBuffer
    call             MSMFree
    .
    .
    .
DriverShutdown      endp
```

MSMFreePages

Processor States	Entry State
	EAX pointer to the buffer
	Interrupts can be in any state
	Call at process time only
	Return State
	Interrupts are unchanged
	Preserved EBX, EBP, ESI, and EDI

Description The HSM must use this routine to return any memory buffers allocated on 4K page boundaries, using **MSMAllocPages** before the driver is permanently shutdown.

Example

```
DriverShutdown            proc
.
.
.
or            ecx,ecx
jnz          PartialShutdown
mov          eax,UserPageBuffer
call          MSMFreePages

DriverShutdown            endp
```

MSMGetAlignment

Processor States	Entry State	
	ECX - <i>Type</i>	0 = alignment requirement 1 = best-case alignment Other = undefined
	Interrupts	in any state
Return State		
	EAX	power of 2-byte boundary data alignment requirement
	Interrupts	preserved
	Preserved	EBP

Description

This routine is called to obtain alignment requirements of the underlying platform. If *Type* is equal to 0, **MSMGetAlignment** returns the worse-case alignment required for any data object that may be involved in I/O transfers. This function allows you to write platform-independent DMA code.

If *Type* is equal to 1, **MSMGetAlignment** returns the data alignment required for the platform to function at its best. This is usually the bus width of the CPU.

The value returned for the type equal to 0 will always be less than or equal to the value returned for *Type* equal to 1.

For most Intel platforms, *Type* equal to 0 should return a 0 and *Type* equal to 1 should return the bus width of the processor (4 for a 386 or 486).

```
Example      DriverInit      proc
              .
              .
              .
              mov      ecx, 1
              call     MSMGetAlignment
              .
              .
              .
DriverInit   t      endp
```

MSMGetBusInfo

Processor States

Entry State

ECX - <i>BusTag</i>	Architecture-dependent value, returned by MSMSearchAdapter , that identifies a specific bus.
Interrupt	any state

Return State

EBX - <i>MemAddrSize</i>	the size in <u>bits</u> of a memory address on the bus specified by <i>BusTag</i> .
EDX - <i>IOAddrSize</i>	the size in <u>bits</u> of an I/O address on the bus specified by <i>BusTag</i>
Flags	set according to EAX
Interrupts	preserved
Preserved	EBP

Completion Codes in EAX

Code	Description
SUCCESSFUL	the operation was completed successfully
ITEM_NOT_PRESENT	the specified bus does not exist or function is not available

Description

MSMGetBusInfo returns the size of the bus addresses associated with *BusTag*.

Example	DriverInit	proc	
		.	
		.	
		.	
		mov	ecx, BusTag; tag for bus queried
		call	MSMGetBusInfo
		jnz	ErrorGettingBusInfo
		.	
		.	
		.	
		DriverInit	endp

MSMGetBusSpecificInfo

Processor States

Entry State

ECX - <i>BusTag</i>	Architecture-dependent value, returned by MSMSearchAdapter or MSMScanBusInfo , that identifies a specific bus.
EDX - <i>Size</i>	size of buffer pointed to in EDI, in bytes
EDI - <i>BusSpecificInfo</i>	pointer to buffer to return bus specific information
Interrupts	can be in any state

Return State

Flags	set according to EAX
Interrupts	are preserved
Preserved	EBX, ECX, EDX, ESI, EDI and EBP

Completion Codes in EAX

Code	Description
SUCCESSFUL	bus specific information returned in buffer pointed to in EDI Note, if the specified bus does not provide specific information, e.g. legacy ISA, no information will be placed in the provided buffer and SUCCESSFUL is returned
BAD_PARAMETER	an invalid <i>BusTag</i> was passed into the routine
ITEM_NOT_PRESENT	the bus is not present or function is not available

Description

MSMGetBusSpecificInfo returns bus specific information in the buffer provided that may be of use to drivers or to installation and configuration utilities. If the buffer provided is insufficient for the information to be provided, the buffer is filled to its capacity and SUCCESSFUL is returned. Therefore a buffer sufficient in size should be provided for a specific bus's information to be returned; 64 bytes should handle the worst case bus specific information structure size.

The following information structure is returned for the specified bus type in the provided buffer:

PnP ISA Bus

ISAInfoStructure	struc			
PnPISABIOSPresentFlag	dd	?		;PnP ISA BIOS Info
PnPISABIOSMajorVer	dw	?		
PnPISABIOSMinorVer	dw	?		
PnPISABIOSRevision	dw	?		
PnPISACMPresentFlag	dd	?		;PnP ISA Configuration Manager Info
PnPISACMType	dd	?		;0=DOS Device Driver (Intel), ;1=Win 95, 2=Win NT, 3=NLM
PnPISACMMajorVer	dw	?		
PnPISACMMMinorVer	dw	?		
PnPISACMRevision	dw	?		
NetFRAMEFlag	dd	?		;1 = NetFRAME system, 0 = not
NonATCompatibleFlag	dd	?		;1 = Non AT comp BIOS, 0 = compatible
HardwareLoaderID	dd	?		
ISAInfoReserved1	dd	?		
ISAInfoReserved2	dd	?		
ISAInfoReserved3	dd	?		
ISAInfoStructure	ends			

CardBus and PC Card (PCMCIA) Buses

```

PCCardInfoStructure      struc
    CSPresentFlag          dd      ?      ;Card Services Info
    CSType                  dd      ?      ;0=DOS Device Driver,
                                         ;1=Win 95, 2=Win NT, 3=NLM

    CSVendorMajorVer        dw      ?
    CSVendorMinorVer        dw      ?
    CSVendorNamePtr         dd      ?
    CSInterfaceLevelMajorVer dw      ?
    CSInterfaceLevelMinorVer dw      ?
    CSNumberOfSockets        dd      ?
    PCCardInfoReserved0      dd      ?
    PCCardInfoReserved1      dd      ?
    PCCardInfoReserved2      dd      ?
    PCCardInfoReserved3      dd      ?
PCCardInfoStructure      ends

```

PCI Bus

```

PCIInfoStructure      struc
    PCIBIOSPresentFlag      dd      ?      ;PCI BIOS Info
    PCIInterfaceLevelMajorVer dw      ?
    PCIInterfaceLevelMinorVer dw      ?
    PCIHardwareMechanism     dd      ?
    LastPCIBusInSystem       dd      ?
    PCIInfoReserved0         dd      ?
    PCIInfoReserved1         dd      ?
    PCIInfoReserved2         dd      ?
    PCIInfoReserved3         dd      ?
PCIInfoStructure      ends

```

Example

```
BusSpecBuf          db    64 dup(?)

DriverInit          proc
    .
    .
    .
    mov     ecx, BusTag
    lea     edi, BusSpecBuf
    mov     edx, 64
    call    MSMGetBusSpecificInfo
    jnz     ErrorGettingBusInfo
    .
    .
    .
DriverInit          endp
```

See Also

MSMScanBusInfo

MSMGetBusType

Processor States

Entry State

ECX - <i>BusTag</i>	an architecture-dependent value which specifies the bus on which the operation is to be performed
Interrupt	any state

Return State

EBX - <i>BusType</i>	A value that indicates the bus type as defined in ODI_NBI.INC. The currently defined values are: 0 = ODI_BUSTYPE_ISA 1 = ODI_BUSTYPE_MCA 2 = ODI_BUSTYPE_EISA 3 = ODI_BUSTYPE_PCMCIA 4 = ODI_BUSTYPE_PCI 8 = ODI_BUSTYPE_CARDBUS
Flags	set according to EAX
Interrupts	preserved
Preserved	EBP

Completion Codes in EAX

Code	Description
SUCCESSFUL	the operation was completed successfully
BAD_PARAMETER	an invalid <i>BusTag</i> was passed into this routine
ITEM_NOT_PRESENT	the function is not available

Description

MSMGetBusType returns a value indicating the bus type of the specified bus. All instances of a particular bus type return the same value. For example, all EISA buses return a 2.

Example

```
DriverInit      proc
.
.
.
mov             ecx, BusTag; bus tag
call            MSMGetBusType; convert to type
jnz             ErrorGettingBusType
cmp             ebx, ODI_BUSTYPE_PCI; is type PCI?
.
.
.
DriverInit      t      endp
```

MSMGetCardConfigInfo

Processor States

Entry State

EAX - <i>Parm1</i>	contains an architecture dependent value that further specifies what information is to be returned, independent of this particular platform and independent of what adapter is described by this information
EBX - <i>Unique-Identifier</i>	contains an architecture-dependent value returned by MSMGetUniqueIdentifier or MSMSearchAdapter . It specifies the location on the bus where the device is located
ECX - <i>BusTag</i>	contains an architecture-dependent value returned by MSMSearchAdapter . It specifies the bus on which the operation is to be performed
EDX - <i>Parm2</i>	contains an architecture dependent value that further specifies what information is to be returned, independent of this particular platform and independent of what adapter is described by this information.
ESI - <i>Size</i>	This parameter specifies the number of bytes to retrieve into the configuration buffer.
EDI - <i>ConfigBuf</i>	is a pointer to configuration buffer in which to retrieve the configuration information. The caller needs to be sure that the buffer is at least <i>Size</i> bytes long.
Interrupts	can be in any state

Return State

Flags	are set according to EAX
Interrupts	are preserved
Preserved	EBX, ECX, EDX, ESI, EDI, EBP

Completion Codes in EAX

Code	Description
SUCCESSFUL	configuration information was received
BAD_PARAMETER	invalid parameter was passed into the call

BAD_COMMAND	called with <i>BusTag</i> for a bus type which has no configuration information to return.
ITEM_NOT_PRESENT	a unique identifier was passed in that has no card present
BUS_SPECIFIC_ERROR	a bus specific error occurred
FAIL	all of the input parameters appear to be valid, but the operation could not be completed

Description

Call **MSMGetCardConfigInfo** only if the bus identified by *BusTag* has configuration information for the bus on a per hardware instance basis. It is the caller's responsibility to know how much and what sort of information is returned, so that *ConfigInfo* is set to point to a sufficiently large space and the resulting information can be interpreted. *Parm1* and *Parm2* are defined on a per bus architecture basis. In other words, their meanings must be the same on all implementations on a particular bus, but will vary from one bus to another. One or both of these parameters may be unused, and if unused, must be set to 0.

The parameter values for the specified bus types are as follows:

EISA Bus

Size	320
Parm1	EISA configuration block number.
Parm2	n/a
ConfigInfo	Filled in with the EISA configuration information for the unique identifier specified.

For a definition of the information returned, refer to the *EISA Specification*.

Micro Channel Bus

Size	8
Parm1	n/a
Parm2	n/a
ConfigInfo	Filled in with I/O port values from POS0 - POS7 (100h - 107h) for the unique identifier specified.

For a definition of the information returned, see the *Personal System/2 Hardware Interface Technical Reference*.

PCI Bus

Size	256
Parm1	Function Number
Parm2	n/a
ConfigInfo	Filled in with PCI configuration information for the unique identifier specified.

For a definition of the information returned, see the *PCI Local Bus Specification*.

Plug and Play

Size	512
Parm1	n/a
Parm2	n/a
ConfigInfo	Filled in with Plug and Play configuration information for the unique identifier specified.

For a definition of the information returned, see the PNPCfgStructure in ODI.INC.

PC Card (PCMCIA)

size The size of the buffer needed to contain the information defined by *parm2*.

parm1 The size of the information requested from the Card Services API, **GetConfigurationInfo**. The valid values are 37 or 42.

Note: If this call returns BAD_PARAMETER, it may be because 42 bytes were requested, but the version of Card Services only supports 37 bytes.

parm2 The order and type of information to be returned in the *configInfo* buffer. The following values are valid for *parm2*:

ODI_DEFAULT_INFO

The *configInfo* buffer will contain the following default information:

- 37 or 42 bytes of information returned by the Card Services API, **GetConfigurationInfo**.
- Attribute memory space equal to the amount of space remaining in the *configInfo* buffer

ODI_IO_MEMORY_WINDOWS

If the size of the information returned by the Card Services API, **GetConfigurationInfo**, is 42 bytes, the *configInfo* buffer will contain:

- The 42 bytes of information returned by the Card Services API, **GetConfigurationInfo**.

If there are I/O windows or memory windows, the window information is placed in the *configInfo* buffer as 18 byte blocks (one 18 byte block for each window). The first thirteen bytes of information is returned by the Card Services API, **GetFirstWindow** or **GetNextWindow**.

For memory windows, the remaining five bytes of information is returned by the Card Services API, **GetMemPage**.

For I/O windows, the remaining five bytes are zero.

- Attribute memory space equal to the amount of space remaining in the *configInfo* buffer.

If the size of the information returned the Card Services API, **GetConfigurationInfo**, is 37 bytes, the *configInfo* buffer will contain:

- The 37 bytes of information returned by the Card Services API, **GetConfigurationInfo**.
- Attribute memory space equal to the amount of space remaining in the *configInfo* buffer.

configInfo The information returned is determined by the *parm2* input parameter.

CardBus Bus

size The size of the buffer needed to contain the information defined by *parm1*, *parm2*, and the desired amount of CIS memory.

parm1 The size of the information requested from the Card Services API, **GetConfigurationInfo**. The valid values are 37 or 42.

Note: If this call returns BAD_PARAMETER, it may be because 42 bytes were requested, but the version of Card Services only supports 37 bytes.

parm2 The size of the PCI configuration space requested. The maximum size available is 256 bytes.

configInfo The *configInfo* buffer will contain:

- The number of bytes specified by *parm1* of information returned by the Card Services API **GetConfigurationInfo**.
- The number of bytes specified by *parm2* of PCI configuration space.
- CIS memory space equal to the amount of space remaining in the *configInfo* buffer.

Example

```
ConfigBuff      db      256 dup(?)

DriverInit      proc
.
.
.
xor             eax, eax      ; function # 0
mov             edx, eax      ; parm2 = 0
mov             ebx, UniqueIdentifier; from
MSMSearchAdapter
mov             ecx, BusTag; tag for PCI bus
mov             esi, 256      ; size of config table
lea             edi, ConfigBuff; ptr to buffer
call            MSMGetCardConfigInfo
jnz             ErrorGettingCardConfig
.
.
.
DriverInit      endp
```

See Also

```
MSMSearchAdapter
MSMGetUniqueIdentifier
MSMRdConfigSpace8
MSMRdConfigSpace16
MSMRdConfigSpace32
MSMWrtConfigSpace8
MSMWrtConfigSpace16
MSMWrtConfigSpace32
```

MSMGetConfigInfo

Processor States

Entry State

EDI	pointer to buffer used to receive the returned configuration information. The caller must ensure that the buffer is at least as long as the number of bytes requested in ECX.
ECX	the requested number of bytes to be returned into the buffer.
Interrupts	can be in any state
Call	at process time

Return State

ECX	the actual number of bytes returned in the configuration buffer.
Flags	set according to EAX
Interrupts	are disabled
Preserved	EDI, EBX & EBP

Completion Code in EAX

SUCCESSFUL	The configuration information was successfully returned in the buffer
BAD_PARAMETER	The size requested in ECX was larger than the actual configuration information available. The number of bytes actually returned is indicated in ECX.

Description

The configuration information is returned in the format defined by the `MSMConfigTable` structure.

- MSMConfigTable struct
- MSMCFG_TableSize dd?
- MSMCFG_TableMajorVersion db?
- MSMCFG_TableMinorVersion db?
- MSMCFG_ModuleMajorVersion db?
- MSMCFG_ModuleMinorVersion db?
- MSMCFG_ODISpecMajorVersion db?
- MSMCFG_ODISpecMinorVersion db?
- MSMCFG_Reserved dw
- MSMCFG_MaxNumberOfBoards dd?
- MSMCFG_SystemFlags dd?
- MSMConfigTable ends

MSMCFG_TableSize

This field contains the actual size of the MSM's configuration table. The value of this field should not be confused with the number of bytes requested or copied (i.e., value in ECX).

MSMCFG_TableMajorVersion

This field contains the major version of the configuration table. The current major version is 1.

MSMCFG_TableMinorVersion

This field contains the minor version of the configuration table. The current minor version is 0.

MSMCFG_ModuleMajorVersion

This field contains the major version of the MSM binary (i.e., MSM.NLM).

MSMCFG_ModuleMinorVersion

This field contains the minor version of the MSM binary (i.e., MSM.NLM).

MSMCFG_ODISpecMajorVersion

This field contains the major version of the ODI Specification that this version of the MSM is written too. For example, if the version of the ODI specification is 3.31, the value of this field is 3.

MSMCFG_ODISpecMinorVersion

This field contains the minor version of the ODI Specification that this version of the MSM is written too. For example, if the version of the ODI specification is 3.31, the value of this field is 31.

MSMCFG_Reserved

This field is reserved.

MSMCFG_MaxNumberOfBoards

The value of this field represents the maximum number of logical boards the MSM supports.

MSMCFG_SystemFlags


The bits in this field are defined below.

Bits	Name	Description
Bit 31	MSM_CFG_CLIENT_BIT	When set to 1 this bit indicates the MSM is running in a client environment. Either this bit or bit 30 will be set, but never both.
Bit 30	MSM_CFG_SERVER_BIT	When set to 1 this bit indicates the MSM is running in a server environment. Either this bit or bit 31 will be set, but never both.
Bit 0-29	Reserved	These bits are reserved.

MSMGetCurrentTime (macro)

Processor States	Entry State	
	Interrupts	can be in any state
	Execute	at process or interrupt time
	Return State	
	EAX	current tick count
	Interrupts	are unchanged
	Preserved	all other registers are preserved

Description **MSMGetCurrentTime** determines the elapsed time (using the current relative time) for some of the HSM-related activities (for example, *TimeOutCheck*). The value returned at the start of an operation subtracted from the current time is the elapsed time in 1/18th second clock ticks. This timer requires more than 7 years to roll over, allowing it to be used for elapsed time comparisons.

Note  Consecutive calls to **MSMGetCurrentTime** must have interrupts enabled between. This allows the OS to update this value.

Example

```
mov     edx, [ebp].Command      ; Let board attempt to
mov     al, Board_Transmit      ; transmit packet again
out     dx, al
MSMGetCurrentTime               ; EAX = current time.
mov     [ebp].TxStartTime, eax  ; Store new timeout
```

MSMGetHINFromHINName

Processor States

Entry State

ESI	pointer to the NULL-terminated HIN name string, not to exceed 128 bytes (including terminator).
Interrupts	can be in any state
Call	at process time or interrupt time

Return State

EBX	the HIN associated with the HIN name.
-----	---------------------------------------

Completion Code in EAX

ODI_NBI_SUCCESSFUL	The HIN was successfully returned in the HINNameMappingInfo structure in ESI.
ODI_NBI_PARAMETER_ERROR	The specified HIN name is invalid.
ODI_NBI_UNSUPPORTED_OPERATION	This function is not available.

Description

The input *hinName* is compared (case insensitively) with HIN names in the system. The corresponding Hardware Instance Number (HIN) is returned.

MSMGetHINNameFromHIN

Processor States	Entry State	
	EBX	HIN to get the HIN name for.
	EDI	pointer to a 128 byte buffer where the HIN name will be stored
	Interrupts	can be in any state
	Call	at process time or interrupt time
	Return State	
	EDI	pointer to the NULL-terminated HIN name string.

Completion Code in EAX

ODI_NBI_SUCCESSFUL	The HIN name was successfully returned in the HINNameMappingInfo structure in ESI.
ODI_NBI_INSTANCE_NONEXIST	The specified HIN is invalid.
ODI_NBI_INSTANCENAME_AVAIL	A HIN name does not exist for the given HIN.
ODI_NBI_UNSUPPORTED_OPERATION	This function is not available.

Description

The input *hin* is translated into its corresponding HIN name and returned to the buffer pointed to by EDI.

MSMGetInstanceNumber

Processor States	Entry State
	ECX - <i>BusTag</i> Architecture-dependent value, returned by MSMSearchAdapter , that identifies a specific bus.
	EBX - <i>UniqueIdentifier</i> an architecture-dependent value returned by MSMGetUniqueIdentifier or MSMSearchAdapter . It specifies the location on the bus where the device is located
	Interrupts can be in any state

Return State
EDX - <i>HIN</i> Hardware Instance Number (HIN) of the device or function. Hardware instance numbers are unique across all buses on the system.
Flags set according to EAX
Interrupts are preserved
Preserved EBX, ECX, ESI, EDI and EBP

Completion Code in EAX	Code	Description
	SUCCESSFUL	EDX contains the hardware instance number
	BAD_PARAMETER	an invalid <i>busTag</i> or <i>UniqueIdentifier</i> was passed into the routine

Description

This call retrieves the hardware instance number of the specified device or function on the specified bus. There is a one to one correspondence between *BusTag* and *UniqueIdentifier* pairs and hardware instance number. You can think of a hardware instance number as a logical slot number. If an adapter contains just one function, the hardware instance number is usually equivalent to the adapter's physical slot number. Hardware instance numbers are unique across all buses and devices on the system. They are generated or determined by the NBI and are consistent across system boots.

Example

```
DriverInit      proc
                .
                .
                .
                mov     ebx, UniqueIdentifier
                mov     ecx, BusTag
                call    MSMGetInstanceNumber
                jnz     ErrorGettingInstanceNum
                mov     InstanceNumber, edx
                .
                .
                .
DriverInit      endp
```

See Also

```
MSMGetUniqueIdentifier
MSMGetInstanceNumberMapping
MSMSearchAdapter
```

MSMGetInstanceNumberMapping

Processor States

Entry State

EDX - <i>HIN</i>	the Hardware Instance Number (HIN) of the device or function
Interrupts	can be in any state

Return State

EBX - <i>Unique-Identifier</i>	an architecture-dependent value returned by MSMGetUniqueIdentifier or MSMSearchAdapter . It specifies the location on the bus where the device is located.
ECX - <i>BusTag</i>	the architecture-dependent value used to identify the specific bus
Flags	set according to EAX
Interrupts	are preserved
Preserved	EDX, ESI, EDI and EBP

Completion Codes in EAX

Code	Description
SUCCESSFUL	bus specific information returned in buffer pointed to in EDI Note: If the specified bus does not provide specific information, such as legacy ISA, no information will be placed in the provided buffer and SUCCESSFUL is returned
BAD_PARAMETER	an invalid <i>BusTag</i> was passed into the routine

Description

Retrieves the *BusTag* and *UniqueIdentifier* associated with the specified hardware instance number. **MSMGetInstanceNumberMapping** is the inverse of **MSMGetInstanceNumber**. It retrieves the *BusTag* and *UniqueIdentifier* associated with the specified hardware instance number. There is a one to one correspondence between *BusTag* and *UniqueIdentifier* pairs and the hardware instance number. You can think of a hardware instance number as a logical slot number. If an adapter contains just one function, the hardware instance number is usually equivalent to the adapter's physical slot number. Hardware instance numbers are unique across all buses and devices on the system. They are generated or determined by the NBI and are consistent across system boots.


See Also

MSMGetUniqueIdentifier
MSMGetInstanceNumber
MSMSearchAdapter

Example

```
DriverInit      proc
                .
                .
                .
                mov     edx, InstanceNumber
                call     MSMGetInstanceNumberMapping
                jnz      ErrorGettingMapping
                mov     UniqueIdentifier, ebx
                mov     BusTag, ecx
                .
                .
                .
DriverInit      endp
```

MSMGetMicroTimer

Note  This routine is supported in MSM v2.20 dated 9-9-93 or later.

Processor States		Entry State
	Interrupts	can be in any state
	Call	at process or interrupt time
		Return State
	EAX	contains time in microseconds
	Interrupts	are preserved
	Preserved	all other registers are preserved

Description **MSMGetMicroTimer** determines the elapsed time for some of the HSM related activities. It can be used instead of **MSMGetCurrentTime** when finer granularity (1 microsecond) is needed or for very short delays while keeping interrupts disabled. This is the preferred this method, rather than using a loop counter based on the value returned by **MSMGetProcessorSpeedRating**.

Example

```
.
. (reset adapter)
.
call     MSMGetMicroTimer      ; get current count
neg      eax
mov      edi, eax              ; EDI = EAX negated

DriverShutdownWait:
call     MSMYieldWithDelay     ; let other processes run
call     MSMGetMicroTimer      ; get current count
add      eax, edi              ; EAX = microseconds expired
cmp      eax, 50               ; 50us passed?
jb       DriverShutdownWait    ; jump if not
```

MSMGetPhysical

Processor States

Entry State

Interrupts	can be in any state
EAX	Logical Memory Address

Return State

EAX	Physical Memory Address
Interrupts	are unchanged
Preserved	all other registers are preserved

Description

HSM's must call this routine to convert a logical address to a physical address instead of adding an offset to it as previously done. Since ECB fragment pointers are set to physical addresses if the **DriverSupportsPhysFrag**s bit is set, this call should only have to be used at **DriverInit** to pass control information in memory up to the adapter.

In future versions of NetWare the driver will not be able to assume the buffer length associated with an address is contiguous. Therefore it is recommended the **DriverSupportsPhysFrag**s bit be set (refer to the *MLIDModeFlags* in Chapter 3).

Example:

```
lea    eax, [ebx].MLIDNodeAddress ; Store node address
call   MSMGetPhysical              ; convert to physical address
out     dx, eax                    ; send physical add. to adapter
```

MSMGetPhysList

Processor States	Entry State	
	ECX	input fragment count
	ESI	pointer to the input fragment list that contains the logical addresses of the fragments
	EDI	pointer to the output fragment list
	EBP	pointer to the adapter data space
	Interrupts	can be in any state
	Call	at process time or interrupt time
	Return State	
	EAX	completion code
	EDX	output fragment count
	EDI	pointer to the output fragment list that contains the sizes and the physical addresses of the fragments
	Preserved	EBX, ECX, EBP, ESI, EDI

Completion Codes

SUCCESSFUL	The operation was completed successfully.
FAIL	The maximum number of output fragments was exceeded.

Description

This function generates a physical address fragment list equivalent to the logical address fragment list passed on input.

Note



There may not be a one-to-one correspondence between the input fragments and the output fragments due to potentially noncontiguous logical memory. Consequently, the number of fragments and the size of the fragments in the output list may be different from the those in the input list.

Do not call this function with a value greater than 16 in ECX. Also, make sure that the output fragment list buffer is large enough to accommodate 16 fragments.

MSMGetPollSupportLevel

Processor States	Entry State	
	Interrupts	can be in any state
	Call	at initialization time only
	Return State	
	EAX = 0	Environment does NOT support polling. Polling procedure will never be called. Adapter must use interrupts only
	EAX = 1	Limited support for polling; polling procedure will be called infrequently. Adapter must use interrupts.
	EAX = 2	Polling is fully supported, however interrupt backup is still recommended due to periods where polling can be infrequent.
	EAX = 3	Polling is fully supported, no interrupt backup is required.
	Interrupts	are preserved
	Preserved	EBX, EBP, ESI and EDI

Description The HSM uses **MSMGetPollSupportLevel** to ascertain the level of support for adapters which favor polling mechanisms, and to determine whether the adapter/driver should be purely interrupt driven, purely polled driven, or a combination of the two with preference given to polling.

Example `call MSMGetPollSupportLevel ; determine poll level support`
`cmp EAX,2`
`jb InterruptDriveAdapter`
`jz MixIntPollDriveAdapter`
`jmp PurePollDriveAdapter`

MSMGetProcessorSpeedRating (macro)

Processor States

Entry State

Interrupts	can be in any state
Execute	at process or interrupt time

Return State

EAX	contains a value representing the relative processor speed of the machine
Interrupts	are unchanged
Preserved	all other registers are preserved

Description

MSMGetProcessorSpeedRating determines the relative processor speed; the larger the value returned, the faster the processor is operating.

Note



Although this procedure provides a means for calculating timing loop delays, this routine should never be used unless it is impossible to enable interrupts and use **MSMGetCurrentTime**, or if it is impossible to use **MSMGetMicroTimer**. Novell recommends that timing loops be avoided whenever possible.

Example

```
MSMGetProcessorSpeedRating      ; EAX = Processor Speed
xor     edx, edx                ; Clear high dword of dividend
mov     ecx, 100                ; Divisor = 100
idiv    ecx                    ; EAX = Speed / 100
mov     ecx, 30000h             ; EAX = (Speed/100) * 30000h
imul    eax, ecx
mov     LoopCounter, eax        ; Save it
```

MSMGetUniquelIdentifier

Processor States	Entry State	
	EBX - <i>Slot</i>	specifies the physical slot to search for the presence of the adapter
	ECX - <i>BusTag</i>	Architecture-dependent value, returned by MSMSearchAdapter , that identifies a specific bus.
	EDX - <i>Channel</i>	contains an adapter specific value identifying the port of a multiport adapter; equal to zero if unused or first port, one if second port, etc.
	ESI - <i>Function-Number</i>	contains an architecture-dependent value that specifies a function of a multifunction adapter; equal to zero if unused or first function, one if second function, etc. This value may be used with a multiport multifunction adapter.
	Interrupts	can be in any state
	Call	at process time ONLY
	Return State	
	EDI - <i>Unique-Identifier</i>	contains an architecture-dependent value returned by MSMGetUniquelIdentifier or MSMSearchAdapter that specifies the location on the bus where the device is located.
	Flags	are set according to EAX
	Interrupts	are preserved
	Preserved	EBX, ECX, EDX, ESI

Completion Codes in EAX	Code	Description
	SUCCESSFUL	The device was found and <i>UniquelIdentifier</i> was returned
	BAD_PARAMETER	The function number or the bus tag was invalid
	ITEM_NOT_PRESENT	The bus or the adapter was not found for the specified inputs

Description

This routine returns a value which uniquely identifies an adapter for the specified input parameters. It will scan the specified bus and try to match the slot, channel, and function number to an adapter (device).

All product ID's appear in memory as defined in their respective specifications.

Example

```
DriverInit      proc
.
.
.
mov             ebx, Slot    ; EISA device slot #
mov             ecx, BusTag; bus tag for EISA bus
mov             edx, 0       ; no channel #
mov             esi, 0       ; no function #
call            MSMGetUniqueIdentifier
jnz             ErrorGettingID
mov             Unqueldentifier, edi; save unique ID
.
.
.
DriverInit      endp
```

MSMGetUniquelIdentifierParameters

Processor States	Entry State	
	EAX - <i>Parameter-Count</i>	The number of elements in the parameter array to be filled in.
	EBX - <i>Unique-Identifier</i>	contains an architecture-dependent value returned by MSMGetUniquelIdentifier or MSMSearchAdapter . It specifies the location on the bus where the device is located
	ECX - <i>BusTag</i>	Architecture-dependent value, returned by MSMSearchAdapter , that identifies a specific bus.
	EDI - <i>Parameters</i>	pointer to the parameter buffer
	Interrupts	can be in any state
	Return State	
	Flags	set according to EAX
	Interrupts	are preserved
	Preserved	EBX, ECX, EDX, ESI, EDI and EBP
Completion Codes in EAX		
	Code	Description
	SUCCESSFUL	parameters returned in buffer pointed to in EDI
	BAD_PARAMETER	an invalid <i>busTag</i> was passed into the routine
	ITEM_NOT_PRESENT	the bus is not present or function is not available

Description

This call returns the bus-specific information about the device or the function represented by the given *UniqueIdentifier*. This call is the inverse of **MSMGetUniqueIdentifier**.

The following are the returned parameter values for each bus type.

ISA Bus N/A**Micro Channel Bus**

parameterCount	1
parameters [0]	physical slot number

EISA Bus

parameterCount	1
parameters [0]	physical slot number

PC Card (PCMCIA) Bus

parameterCount	1
parameters[0]	For single function cards, the physical socket number (1-based). For multiple function cards, the function number (1-based) is in the least significant byte, and the physical socket number is in the next byte.

PCI Bus

parameterCount	2
parameters [0]	zero (PCI version 2.0) physical slot number (PCI version 2.1)
parameters [1]	bus/device/function number combination, equivalent to the value returned from the PCI BIOS <i>Find Device</i> function call

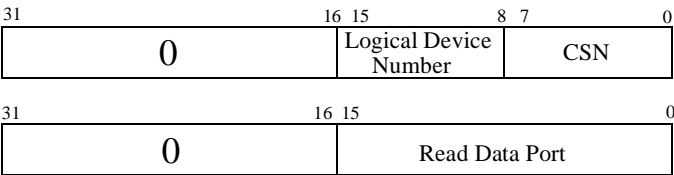
PnP ISA Bus

parameterCount	2
parameters [0]	CSN (Card Select Number) is in least significant byte with the Logical Device Number in the next.
parameters [1]	Read Data Port

CardBus Bus

parameterCount	2
parameters[0]	For single function cards, the physical socket number (1-based). For multiple function cards, the function number (1-based) is in the least significant byte, and the physical socket number is in the next byte.
parameter[1]	Bus/device/function number combination equivalent to the value returned from the PCI BIOS <i>FindDevice</i> function.

Figure 7-2 PnP ISA Bus Parameters



Example

```
UniquelDBuf      dd    2 dup(?)

DriverInit      proc
.
.
.
mov     eax, 2    ; number of parameters
mov     ebx, UniquelIdentifier
mov     ecx, BusTag
lea     edi, UniquelDBuf
call    MSMGetUniquelIdentifierParameters
jnz     ErrorGettingUIDParams
.
.
.
DriverInit      endp
```


MSMHardwareFailure

Processor States

Entry State

EBP	pointer to the Adapter Data Space.
EAX	one of the following failure type values: NOTIFY_CRITICAL NOTIFY_FATAL NOTIFY_DEGRADED
ESI	pointer to a NULL-terminated string describing the failure.
Interrupts	can be in any state
Call	at process time or interrupt time

Return State

EAX	set to 0
Interrupts	are unchanged
Preserved	EBX & EBP

Description

The HSM calls this routine to report a hardware error.

NOTIFY_FATAL should be reported if the HSM was able to detect a hardware failure from which there is no possibility of recovery.

NOTIFY_CRITICAL should be reported if the HSM has encountered an adapter hardware problem and failed to recover using the available hardware reset capabilities, but the system may be able to restore the hardware to a functional state, using platform or media specific recovery procedures. For example, on some platforms it may be possible to power cycle the adapter.

NOTIFY_DEGRADED should be reported if the hardware has experienced a failure, but is still functional.

MSMInitAlloc

Processor States	Entry State
	EAX number of bytes of memory to allocate
	Interrupts can be in any state
	Call at process time only
	Return State
	EAX pointer to the allocated buffer. (zero = failure)
	Interrupts are in the same state as when the routine was called (but might have been enabled during the call if DriverNeedsBelow16Meg is nonzero)
	Preserved EBX, EBP, ESI, and EDI

Description

HSMs must use the **MSMInitAlloc** routine if they must allocate memory prior to calling **MSMRegisterHardwareOptions**. If successful, **MSMInitAlloc** returns a pointer to the allocated buffer in EAX. If the routine was unsuccessful, EAX will be zero.

When the driver frees any buffer allocated by **MSMInitAlloc**, it must use the **MSMInitFree** routine.

MSMInitAlloc and **MSMInitFree** MUST be used as a pair. Do not use **MSMFree** to release resources obtained by a call to **MSMInitAlloc**.

If the **DriverParameterBlock** variable, **DriverNeedsBelow16Meg**, was initialized to any nonzero value (see chapter 3), the MSM will attempt to allocate memory below the 16 megabyte boundary.


Example**DriverInit**

```
proc
.
.
.
mov    eax, UserBufferSize
call   MSMInitAlloc
or     eax, eax
jz     ErrorAllocatingBuffer
mov    UserBuffer, eax
.
.
.
mov    eax, UserBuffer
call   MSMInitFree
.
.
.
call   MSMRegisterHardwareOptions
```

MSMInitFree

Processor States	Entry State
	EAX pointer to the buffer to free (must have been previously allocated using MSMInitAlloc)
	Interrupts can be in any state
	Call at process time only
	Return State
	Interrupts are preserved
	Preserved EBX, EBP, ESI, and EDI

Description HSMs must use the **MSMInitAlloc** routine during initialization, if they allocate memory prior to calling **MSMRegisterHardwareOptions**. When the driver frees any buffer allocated by MSMInitAlloc, it must use the **MSMInitFree** routine.

Note

MSMInitAlloc and **MSMInitFree** must be used as a pair. Do not use **MSMFree** to release resources obtained by a call to **MSMInitAlloc**.

Example

DriverInit proc
.
.
.
mov eax, UserBufferSize
call MSMInitAlloc
or eax, eax
jz ErrorAllocatingBuffer
mov UserBuffer, eax
.
.
.
mov eax, UserBuffer
call MSMInitFree
.
.
.
call MSMRegisterHardwareOptions

MSMNESLDeRegisterConsumer

Processor States

Entry State

ESI	pointer to the NESL_ECB passed to MSMNESLRegisterConsumer
Interrupts	are enabled
Call	at process time only

Return State

Flags	set according to EAX
Interrupts	are unchanged
Preserved	ESI

Completion Code in EAX

NESL_OK	De-registry succeeded.
NESL_EVENT_NOT_REGISTERED	The specified NESL_ECB is not registered.
NESL_CONSUMER_NOT_FOUND	The consumer is NULL or cannot be located.

Description

This function de-registers a consumer of a specific event.

See Also

- MSMNESLRegisterConsumer
- Appendix C, "NESL Support"

MSMNESSLDeRegisterProducer

Processor States	Entry State	
	ESI	pointer to the NESL_ECB passed to MSMNESSLRegisterProducer
	Interrupts	can be in any state
	Call	at process time only
	Return State	
	Flags	set according to EAX
	Interrupts	are unchanged
	Preserved	ESI

Completion Code in EAX	
NESL_OK	De-registry succeeded.
NESL_EVENT_NOT_REGISTERED	The specified NESL_ECB is not registered.
NESL_PRODUCER_NOT_FOUND	The producer is NULL or cannot be located.

Description

This function de-registers a producer of a specific event. If the producer is the last producer for that event, any remaining consumers of the event are placed onto an orphaned consumer's list.

See Also

MSMNESSLRegisterConsumer
Appendix C, "NESL Support"

MSM NESL ProduceEvent

Processor States	Entry State
	ESI pointer to the NESL_ECB passed to MSM-NESLRegisterProducer
	EDX pointer to the Event Parameter Block
	EDI points to a location to place a pointer to the consumer NESL_ECB that consumed the event.
	Interrupts are in any state
	Call at process or interrupt time
	Return State
	EDI set according to EAX
	Flags set according to EAX
	Interrupts are unchanged
	Preserved ESI, EDX and EDI

Completion Code in EAX	
NESL_PRODUCER_NOT_FOUND	The producer is NULL.
NESL_EVENT_CONSUMED	The event is consumable and is consumed. EDI is set to the consumer's NESL_ECB.
NESL_EVENT_NOT_CONSUMED	The event is consumable and is not consumed. EDI is set to NULL.
NESL_EVENT_BROADCAST	Event has been broadcast to all consumers. EDI is not changed.

Description

An event producer calls this to notify registered consumers that the event has occurred. If the event is consumable, then one of the consumers may consume the event and the event notification will stop.

Producer routines and consumer routines running on asynchronous events (e.g., IPX packets, interrupts), must be re-entrant. **MSMNESLProduceEvent** will not protect the consumer routine from being re-entered. For example, if the consumer routine re-enables interrupts, another asynchronous event can be issued from a producer and thus re-enter the consumer.

It is up to either the producer and the consumer routine to protect themselves from re-entrancy issues. Further, they must take steps to ensure that there is on stack overflow because of their activities.

The Event Parameter Block fields are defined as follows:

Field	Description
EPBMajorVersion	Major version of the Event Parameter Block. The current version is 1 (for 1.00).
EPBMinorVersion	Minor version of the Event Parameter Block. The current version is 00 (for 1.00).
EPBEventName	Event name or class name for the event as register with NESL (e.g., "Service Suspend" or "Service Resume"). All valid event names must be registered with Novell Labs.
EPBEventType	Event subclass name for the event. An example of a subclass for "Service Suspend" would be "APM Suspend". All valid event subclass names must be registered with Novell Labs.
EPBmoduleName	Pointer to module name that generated the event (e.g., NE2000).
EPBDataPtr0	The MSM uses this field to pass a pointer to the MLID's configuration table.
EPBDataPtr1	Event dependent information.
EPBEventScope	The HSM must set this field to EPB_SPECIFIC_EVENT.

Field	Description
EPBReserved	Reserved by Novell.

See Also

Appendix C, “NESL Support”

MSMNESLProduceMLIDEvent

Processor States	Entry State
	ESI pointer to the NESL_ECB passed to MSMNESLRegisterProducer
	EDX pointer to the Event Parameter Block
	EDI points to a location to place a pointer to the consumer NESL_ECB that consumed the event.
	EBP pointer to Adapter Data Space
	Interrupts are in any state
	Call at process or interrupt time
	Return State
	EDI set according to EAX
	Flags set according to EAX
	Interrupts are unchanged
	Preserved ESI, EDX, EDI and EBP

Completion Code in EAX	
NESL_PRODUCER_NOT_FOUND	The producer is NULL.
NESL_EVENT_CONSUMED	The event is consumable and is consumed. EDI is set to the consumer's NESL_ECB.
NESL_EVENT_NOT_CONSUMED	The event is consumable and is not consumed. EDI is set to NULL.
NESL_EVENT_BROADCAST	Event has been broadcast to all consumers. EDI is not changed.
NESL_INVALID_CONTEXT_HANDLE	The logical board(s) identified by Adapter Data Space are not valid.

Description

An event producer calls this to notify registered consumers that the event has occurred. If the event is consumable, then one of the consumers may consume the event and the event notification will stop. This call produces the event **for each logical board** associated with Adapter Data Space.

Producer routines and consumer routines running on asynchronous events (e.g., IPX packets, interrupts), must be re-entrant. **MSMNESLProduceEvent** will not protect the consumer routine from being re-entered. For example, if the consumer routine re-enables interrupts, another asynchronous event can be issued from a producer and thus re-enter the consumer.

It is up to either the producer and the consumer routine to protect themselves from re-entrancy issues. Further, they must take steps to ensure that there is no stack overflow because of their activities.

The Event Parameter Block fields are defined as follows:

Field	Description
EPBMajorVersion	Major version of the Event Parameter Block. The current version is 1 (for 1.00).
EPBMinorVersion	Minor version of the Event Parameter Block. The current version is 00 (for 1.00).
EPBEventName	Event name or class name for the event as register with NESL (e.g., "Service Suspend" or "Service Resume"). All valid event names must be registered with Novell Labs.
EPBEventType	Event subclass name for the event. An example of a subclass for "Service Suspend" would be "APM Suspend". All valid event subclass names must be registered with Novell Labs.
EPBmoduleName	Pointer to module name that generated the event (e.g., NE2000).
EPBDataPtr0	The MSM uses this field to pass a pointer to the MLID's configuration table.
EPBDataPtr1	Event dependent information.

Field	Description
EPBEventScope	The HSM must set this field to EPB_SPECIFIC_EVENT.
EPBReserved	Reserved by Novell.

MSM NESLRegisterConsumer

Processor States

Entry State

ESI	pointer to a NESL_ECB.
Interrupts	are in any state
Call	at process time only

Return State

Flags	set according to EAX
Interrupts	are unchanged
Preserved	ESI

Completion Code in EAX

NESL_OK	Registry was successful.
NESL_DUPLICATED_NECB	The NESL_ECB was previously registered in the event table.
NESL_INVALID_NOTIFY_PROC	The consumer's notification procedure is NULL
NESL_CONSUMER_NOT_FOUND	The NESL_ECB pointer is NULL.
NESL_FIRST_ALREADY_HOOKED	The head of the consumer list has already been hooked. Event has been broadcast to all consumers. EDI is not changed.

Description

This function registers the consumer of an event. If a producer of the event is not currently registered, the consumer is placed onto an orphaned consumer list.

The NESL_ECB fields for this function are defined as follows:

Field	Description
NecbNext	RESERVED. This field should not be modified by the calling routine while the NESL_ECB is registered.
NecbVersion	This field contains the version number of the NESL_ECB structure. This field allows the interface to be expanded in the future while still providing full backward compatibility. The current version is 1.
NecbOsiLayer	The definition NESL_HOOK_FIRST may also be used in element NecbOsiLayer. This definition causes a consumer to be hooked first, no matter what. If the caller sets the low byte of NecbOsiLayer to this value, the consumer will be hooked first in the consumer list. Normally NESL events will put lower layer identifiers before the hooked lead element. If another call is made specifying this definition an error will be returned to the caller and the element will not be added to the list.
NecbEventName	ASCIIZ name string of the event or class of events. This name has the maximum length of NESL_MAX_NAME_LENGTH.
NecbRefData	RESERVED.
PNecbNotifyProc	Pointer to the event notification callback routine. UINT32 MyNotifyProc (NESL_ECB *ConsumerNecb, NESL_ECB *ProducerNecb, Void *eventData);
ConsumerNecb	Points to the NESL_ECB used by consumer during MSMNESLRegisterConsumer.
ProducerNecb	Points to the NESL_ECB used by the producer during MSMNESLRegisterProducer.

7-74 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Field	Description
EventData	<p>If the producer only has one data item, it can be passed to the consumer as an argument or as an address.</p> <p>If the producer has more data than one item or if the producer wishes to guarantee portability, then the address of an array of data items should be passed. The structure of the eventData must be defined by the producer and known by the consumer if it is to be interrupted properly</p> <p>Return from a consumer after an event notification callback:</p>
NESL_EVENT_CONSUMED	Event was consumed by the consumer process.
NESL_EVENT_NOT_CONSUMED	<p>Event was not consumed by the process.</p> <p>This is only really applicable if the event is consumable, but a consumer should always do this to be compatible with both types of events. Called from foreground time or from interrupt time with interrupts enabled or disabled.</p>
NecbOwner	Specifies the owner of the NESL_ECB. This field is platform-specific and platform-dependent. The DOS/MS Windows implementation REQUIRES this field to be set to the owner's module handle information.
NecbWorkSpace	RESERVED. This field should not be modified by the calling routine while the NESL_ECB is registered.

See Also

Appendix C, "NESL Support"

MSMNESLRegisterProducer

Processor States

Entry State

ESI	pointer to a NESL_ECB.
Interrupts	are in any state
Call	at process time only

Return State

Flags	set according to EAX
Interrupts	are unchanged
Preserved	ESI

Completion Code in EAX

NESL_OK	Registry was successful.
NESL_REGISTERED_UNIQUE	A previous producer has registered the event as unique and this producer tried to register the event as non-unique.
NESL_REGISTERED_NOT_UNIQUE	A previous producer has registered the event as non-unique and this producer tried to register the event as unique.
NESL_REGISTERED_CONSUMABLE	A previous producer has registered the event as consumable and this producer tried to register the event as broadcast.
NESL_REGISTERED_BROADCAST	A previous producer has registered the event as a broadcast and this producer tried to register the event as consumable.
NESL_EVENT_TABLE_FULL	The event was not registered because the event table is full.
NESL_DUPLICATE_NECB	The NESL_ECB was previously registered in the event table.
NESL_PRODUCER_NOT_FOUND	The NESL_ECB is NULL.

Description

This function registers the producer of an event and creates a consumer list containing the consumers of this event. The event definition contains the rules necessary concerning process and interrupt time execution during event notification.

The NESL_ECB fields for this function are defined as follows:

Field	Description
NecbNext	RESERVED. This field should not be modified by the calling routine while the NESL_ECB is registered.
NecbVersion	This field contains the version number of the NESL_ECB structure. This field allows the interface to be expanded in the future while still providing full backward compatibility. The current version is 1.
NecbOsiLayer	Reserved. The value of this field will be ignored..
NecbEventName	ASCIIZ name string of the event or class of events. This name has the maximum length of NESL_MAX_NAME_LENGTH.
NecbRefData	This is a flag field used to specify whether the event is unique or consumable. It also indicates the sorting order for calling registered consumers at event time. Consumers which are on the orphan consumer list will be sorted
NESL_SORT_CONSUMER_BOTTOM_UP	Use bottom-up relative ordering on the consumer's NecbOsiLayer field in maintaining an ordered list of consumers requiring notification.
NESL_CONSUME_EVENT	The event can be consumed by one of the registered consumers. By default, an event is broadcast to all registered consumers. This flag will cause a chaining effect among the consumers which will start with the first registered consumer and proceed to the next until one of the consumers consumes the event or the end of the consumer list is reached.
NESL_UNIQUE_PRODUCER	<p>The producer of the event must be unique. If there is another producer registered with the same event string, then this call will fail. By default, there can be multiple producers of the same event.</p> <p>This flag is used to prohibit multiple producers provided that this is the first producer registered.</p>

Field	Description
PnecbNotifyProc	RESERVED. The value of this field will be ignored..
NecbOwner	Specifies the owner of the NESL_ECB. This field is platform-specific and platform-dependent. The DOS/ MS Windows implementation REQUIRES this field to be set to the owner's module handle information.
NecbWorkSpace	RESERVED. This field should not be modified by the calling routine while the NESL_ECB is registered.

See Also

Appendix C, “NESL Support”


MSMParseCustomKeywords

Processor States	Entry State	
	ESI	pointer to the <i>DriverParameterBlock</i>
	Return State	
	EBX	is preserved
	Zero Flag	is cleared if a "T_REQUIRED" custom keyword was not entered on the command-line or by user after being prompted.

Description

Drivers can define keywords that allow custom parameters or flags to be entered from the load command-line. (Refer to the "Driver Keyword" section in Chapter 3 for a complete description of how to define custom keywords.)

Custom keywords are normally processed during initialization when **DriverInit** calls **MSMParseDriverParameters**. If the driver must have custom keywords processed earlier in initialization, the **DriverInit** routine can call **MSMParseCustomKeywords**.

Note  **MSMParseDriverParameters** will still call custom keyword procedures even if **MSMParseCustomKeywords** called them earlier.

The MSM parses the command-line for custom keywords and calls the procedure corresponding to that keyword. Requirements for custom keyword procedures are described in the next section.

Custom Keyword Procedure

When the MSM calls a custom keyword procedure, the values of the registers on entry will vary depending on which keyword parsing flags (if any) were used. The "Driver Keyword Enhancements" section of Chapter 3 describes the parsing flags and how they are used.

On Entry

EDX is nonzero if a T_REQUIRED keyword was found on the original command-line.

EDX is zero if a T_REQUIRED keyword was not found on the original command-line and the user had to be prompted for information

T_REQUIRED - The keyword must be entered. If it doesn't exist on the command-line or configuration file, the user will be prompted for it. If the users does not enter a value, **MSMParseCustomKeywords** will return with an error.

T_STRING - The Keyword Routine will be called with a pointer to the beginning of the string that matched the keyword text.

Example:

```
load <driver> custom int=3
```

Routine called with ESI pointing to "custom int=3"

T_NUMBER - The Keyword Routine will be called with the value entered on the command-line in EAX. The user must enter a decimal number.

Example:

```
load <driver> custom=100
```

Routine called with EAX = 64h

T_HEX_NUMBER - The Keyword Routine will be called with the value entered on the command-line in EAX. The user must enter a hexadecimal number.

Example:

```
load <driver> custom=100
```

Routine called with EAX = 1080

T_HEX_STRING - The Keyword Routine will be called with ESI pointing to a six byte value that was entered on the command-line. The user must enter this string using hexadecimal numbers.

Example:
load <driver> custom=01020304

Routine called with ESI -> 00, 00, 01, 02, 03, 04

The following is an example of a driver for an adapter that may require memory below 16 megabytes depending on information read from a port. The example will prompt the user for an I/O port and determine whether it needs memory below 16 megabytes or not.

Example

OSDATA segment rw public 'DATA'

```
DriverParameterBlock      label    dword
.
.
.
    DriverNumKeywords      dd      1
    DriverKeywordText      dd      KeywordTextTable
    DriverKeywordTextLen   dd      KeywordTextLenTable
    DriverProcessKeywordTab dd      KeywordProcedureTable
.
.
.
DriverParameterBlockEnd

KeywordTextTable          dd      PortKeyword

KeywordTextLenTable       dd      PortKeywordLen

KeywordProcedureTable      dd      PortKeywordRoutine

;-----
;-----
; Define Keywords and related Parameters
;-----
;-----

PortKeyword               db      'PORT'
PortKeywordLen            equ     ($ - PortKeyword) OR T_HEX_NUMBER OR
T_REQUIRED

                                dd      300          ; Min port value
                                dd      360          ; Max port value
                                dd      PortDefault   ; Default Port
```

```

                                dd      PortValid      ; Valid characters
                                dd      PortPrompt      ; Prompt string

PortDefault                    db      "300", 0
PortValid                     db      "0..9A..F", 0      ; Hex digits only
Port                          db      "Enter the Port Number: ", 0

;-----
; Define some variables used by custom keyword routine
;-----

BasePortValue                  dd      0
PortOnCommandLine              dd      0

.
.
.

OSDATA ends
```

DriverInit proc

```

Cpush
mov     DriverStackPointer, esp
or      KeywordTextLenTable, T_REQUIRED
lea     esi, DriverParameterBlock
call    MSMParseCustomKeywords
jnz     DriverInitError          ;keyword not entered

mov     edx, BasePortValue

(read I/O port information into eax to determine if memory
below 16 meg is required or not)

mov     DriverNeedsBelow16Meg, 0      ;assume below 16 not required
or      eax, eax                     ;check if below 16 required?
je      DriverInitRegisterHSM        ;jump if not
mov     DriverNeedsBelow16Meg, -1     ;set below 16 flag

```

DriverInitRegisterHSM:

```

lea     esi, DriverParameterBlock
call    TSM>RegisterHSM

```

;* Clear T_REQUIRED bit for the custom keyword so MSMParseDriverParameters will not prompt for it again if it

;* was not on the original command-line.

```

and     KeywordTextLenTable, NOT T_REQUIRED

```

;* We need to set the NeedsIOPort0Bit if "PORT=" is already on the command-line. Otherwise the OS will complain

;* that it saw a standard keyword that wasn't needed.

```

mov     eax, NeedInterrupt0Bit OR CAN_SET_NODE_ADDRESS
cmp     PortOnCommandLine, 0
je      DriverInitParse
or      eax, NeedsIOPort0Bit

```

DriverInitParse:

```

lea     ecx, AdapterOptions
call    MSMParseDriverParameters
jnz     DriverInitError

mov     eax, BasePortValue          ;force IO Port to what
mov     [ebx].MLIDIOPortsAndLengths, ax ;we got from custom keyword

```

```
        call    MSMRegisterHardwareOptions
        .
        .
        .
DriverInit    endp

PortKeywordRoutine    proc
        mov     BasePortValue, eax
        mov     PortOnCommandLine, edx

PortKeywordRoutine    endp
```

7-84 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

MSMParseDriverParameters

Processor States	Entry State
	EAX is the <i>DriverNeedsBitMask</i> parameter
	ECX pointer to <i>DriverAdapterOptions</i> structure
	Interrupts can be in any state
	Call at initialization time
	Return State
	Zero Flag set if successful; otherwise an error occurred.
	EAX zero if successful; otherwise EAX points to an error message which the driver must print using MSMPrintString before returning to the operating system with EAX nonzero.
	EBX pointer to the Frame Data Space
	Interrupts are disabled
	Preserved no registers are preserved

Description

MSMParseDriverParameters is used in conjunction with **MSMRegisterHardwareOptions** to parse the command-line options.

Each standard load option corresponds to a field in the driver's configuration table. Using *DriverNeedsBitMask* as a guide, this function collects the necessary information from the command-line and from the Adapter Options Structure and fills out the appropriate fields of the configuration table.

The following pages describe the format of the Adapter Options structure and the *DriverNeedsBitMask* parameter.



Note During this routine the HSM's custom keywords are also processed (see "Driver Keywords" in Chapter 3)

Adapter Options

The Adapter Options Structure is defined in the ODI.INC file and is shown below. Each field of the structure is a pointer to a list of possible options for that field. If an option is not supported, a zero is placed in that field. The options correspond to fields in the driver's configuration table.

```

AdapterOptionDefinitionStructure      struc

    IOSlot                dd ?          ; Ptr to a list of possible slots
    IOPort0               dd ?          ; " primary ports
    IOLength0             dd ?          ; " number of primary ports
    IOPort1               dd ?          ; " secondary ports
    IOLength1             dd ?          ; " number of secondary ports
    MemoryDecode0         dd ?          ; " primary memory values
    MemoryLength0         dd ?          ; " primary memory sizes
    MemoryDecode1         dd ?          ; " secondary memory values
    MemoryLength1         dd ?          ; " secondary memory sizes
    Interrupt0            dd ?          ; " primary interrupt values
    Interrupt1            dd ?          ; " secondary interrupt values
    DMA0                  dd ?          ; " primary DMA values
    DMA1                  dd ?          ; " secondary DMA values
    Channel               dd ?          ; " channel # for multichannel
                                   adapters

AdapterOptionDefinitionStructure      ends

```

All lists pointed to must begin with a dword value indicating the number of options in the list. For example, the lists for an adapter with options for interrupt and port number might appear as follows.

```

IOPortOptions      dd 4          ; number of options
                  dd 300h,310h,320 h,330h; options
IntOptions         dd 3          ; number of options
                  dd 2, 3, 5      ; options

DriverAdapterOptions      AdapterOptionDefinitionStructure
                          <0,IOPortOptions,0,0,0,0,0,0,IntOptions,0,0,0>

```

Needs Options

DriverNeedsBitMask is used to inform the parser which configuration options the driver requires.

If there are multiple possibilities for a configuration option and a driver wants this function to return which option to use, it must set the appropriate bit of the mask.

If there is only one value for a configuration option, the HSM does not set its bit in *DriverNeedsBitMask*. The value can be set directly in the configuration table.

Equates for the bit positions of each option are provided in the ODI.INC file. These options are described in the following table.

DriverNeedsBitMask

3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0									
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0														

Bit #	DriverNeedsBits	
31	MUST_SET_NODE_ADDRESS	(80000000h)
30	CAN_SET_NODE_ADDRESS	(40000000h)
13	NeedsChannelBit	(00002000h)
12	NeedsDMA1Bit	(00001000h)
11	NeedsDMA0Bit	(00000800h)
10	NeedsInterrupt1Bit	(00000400h)
9	NeedsInterrupt0Bit	(00000200h)
8	NeedsMemoryLength1Bit	(00000100h)
7	NeedsMemoryDecode1Bit	(00000080h)
6	NeedsMemoryLength0Bit	(00000040h)
5	NeedsMemoryDecode0Bit	(00000020h)
4	NeedsIOLength1Bit	(00000010h)
3	NeedsIOPort1Bit	(00000008h)
2	NeedsIOLength0Bit	(00000004h)
1	NeedsIOPort0Bit	(00000002h)
0	NeedsIOSlotBit	(00000001h)

Command-Line Examples

Option	Command-Line	Description
IOSlot	load <driver> SLOT=4	Use slot 4
IOPort0	load <driver> PORT=300	Base Port0 = 300h
IOLength0	load <driver> PORT=300:A	Length0 = 0Ah
IOPort1	load <driver> PORT1=700	Base Port1 = 700h
IOLength1	load <driver> PORT1=700:14	Length1 = 14h
MemoryDecode0	load <driver> MEM=C0000	Base Memory0 = C0000h
MemoryLength0	load <driver> MEM=C0000:1000	MemLength0 = 1000h (4K)
MemoryDecode1	load <driver> MEM1=CC000	Base Memory1 = CC000h
MemoryLength1	load <driver> MEM1=CC000:2000	MemLength1 = 2000h (8K)
Interrupt0	load <driver> INT=3	Interrupt0 = 3
Interrupt1	load <driver> INT1=5	Interrupt1 = 5
DMA0	load <driver> DMA=0	DMA0 = 0
DMA1	load <driver> DMA1=3	DMA1 = 3
Channel	load <driver> CHANNEL=2	Use Channel 2

Example

```
IOPortOptions      dd 4                ; number of options
                  dd 300h,310h,320h,330h; options
IntOptions         dd 3                ; number of options
                  dd 2, 3, 5          ; options

DriverAdapterOptions AdapterOptionDefinitionStructure
                   <0,IOPortOptions,0,0,0,0,0,0,IntOptions,0,0,0>

DriverInit         proc
.
.
.
                   mov  eax, NeedsIOPort0Bit OR NeedsInterrupt0Bit OR

                   CAN_SET_NODE_ADDRESS
                   lea  ecx, DriverAdapterOptions
                   call MSMParseDriverParameters
                   jnz  ParseParameterError
                   call MSMRegisterHardwareOptions
.
.
.
```

MSMPrintString

Processor States

Entry State

ECX	possible argument #1
EDX	possible argument #2
ESI	pointer to a null terminated message (cannot exceed 128 bytes)
Interrupts	can be in any state but might be disabled during the call
Call	at initialization time only

Return State

Interrupts	are in the same state as when this routine was called
Preserved	EBX, EBP, EDI, and ESI

Description

This function prints the message pointed to by ESI. The HSM's initialization routine must call **<TSM>RegisterHSM** prior to using this print procedure.

The possible arguments #1 and #2 above are used here the same way in which they are used in the **printf** routine in C language. If there are no format specifications in the string, ECX and EDX are ignored.

This routine has added functionality which supports an additional string format. If the string is preceded by a word size error number in the range of 100-999, the MSM will print the driver name, the platform name (NW for NetWare), and the decimal error number, before printing the specified string. (See the *Standard MLID Message Definitions* supplement for a listing of standard messages.)

Example

```

ErrorMessage    dw    102
                db    "Board failed to execute reset command.",0
                .
                .
                .
lea             ESI, ErrorMessage
call            MSMPrintString

```

The example above would output the following message if the adapter is an NE2000:

NE2000-NW-102: Board failed to execute reset command.

7-92 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

MSMPrintStringFatal

Processor States

Entry State

ECX	possible argument #1
EDX	possible argument #2
ESI	pointer to a null terminated error message (cannot exceed 128 bytes)
Interrupts	can be in any state but might be disabled during the call
Call	at initialization time only

Return State

Interrupts	are in the same state as when this routine was called
Preserved	EBX, EBP, EDI, and ESI

Description

This function prints “FATAL:” followed by the specified error message. The HSM’s initialization routine must call <TSM>**RegisterHSM** prior to using this print procedure.

The “Possible Arguments #1 and #2” above are used here the same way in which they are used in the C-language **printf** routine. If there are no format specifications in the string, ECX and EDX are ignored. (See the *Standard MLID Message Definitions* supplement for a listing of standard messages.)

Example

```
ErrorMessage      db      'Adapter %d, Error Code: %x', CR,LF,0
                  .
                  .
                  .
mov      ECX, BoardNumber      ; argument #1
mov      EDX, ErrorNumber      ; argument #2
mov      ESI, offset ErrorMessage
call     MSMPrintStringFatal
```

MSMPrintStringWarning

Processor States

Entry State

ECX	possible argument #1
EDX	possible argument #2
ESI	pointer to a null terminated error message (cannot exceed 128 bytes)
Interrupts	can be in any state but might be disabled during the call
Call	at initialization time only

Return State

Interrupts	are in the same state as when this routine was called
Preserved	EBX, EBP, EDI, and ESI

Description

This function prints “WARNING:” followed by the specified error message pointed to by ESI. The HSM's initialization routine must call **<TSM>RegisterHSM** prior to using this print procedure.

The “Possible Arguments #1 and #2” above are used here the same way in which they are used in the C-language **printf** routine. If there are no format specifications in the string, ECX and EDX are ignored. (See the *Standard MLID Message Definitions* supplement for a listing of standard messages.)

Example

```
ErrorMessage      db    'Adapter %d, Error Code: %x', CR,LF,0
                  .
                  .
                  .
mov     ECX, BoardNumber      ; argument #1
mov     EDX, ErrorNumber      ; argument #2
mov     ESI, offset ErrorMessage
call    MSMPrintStringWarning
```


MSMRdConfigSpace8

Processor States	Entry State	
	EAX - <i>Offset</i>	contains an offset into the configuration space.
	EBX - <i>Unique-Identifier</i>	contains an architecture-dependent value that specifies the location on the bus where the device is located
	ECX - <i>BusTag</i>	contains an architecture-dependent value returned by MSMSearchAdapter . It specifies the bus on which the operation is to be performed
	Interrupts	in any state
	Return State	
	DL	unsigned read value
	Interrupts	preserved
	Preserved	EAX, EBX, ECX, ESI, EDI, EBP

Description

This function takes an offset, a *UniqueIdentifier* and a *BusTag* to identify an offset in a specific adapter’s configuration space and performs whatever operations are necessary to acquire and return 8 bits of configuration information.

The function is provided only for drivers that need to interact with configuration space. On most buses, **MSMGetCardConfigInfo** will satisfy a driver's need.

Note

For most buses, this call will do nothing. It has meaning only on buses that have a configuration address space that is separated from memory or I/O space (for example, a PCI bus).

See Also

MSMSearchAdapter, MSMGetUniqueIdentifier, MSMRdConfigSpace16, MSMRdConfigSpace32, MSMGetCardConfigInfo, MSMWrtConfigSpace8, MSMWrtConfigSpace16, MSMWrtConfigSpace32.

Example

```
DriverInit    proc
.
.
.
mov          ebx, UniqueIdentifier; from
              MSMSearchAdapter
mov          ecx, BusTag; BusTag for PCI
mov          eax, 13      ; PCI Header Type Offset
call         MSMRdConfigSpace8
.
.
.
DriverInit    endp
```

MSMRdConfigSpace16

Processor States	Entry State	
	EAX - <i>Offset</i>	contains an offset into the configuration space.
	EBX - <i>Unique-Identifier</i>	contains an architecture-dependent value returned by MSMGetUniqueIdentifier or MSMSearchAdapter . It specifies the location on the bus where the device is located
	ECX - <i>BusTag</i>	contains an architecture-dependent value returned by MSMSearchAdapter . It specifies the bus on which the operation is to be performed
	Interrupts	in any state
	Return State	
	DX	unsigned read value
	Interrupts	preserved
	Preserved	EAX, EBX, ECX, ESI, EDI, EBP

Description

This function takes an offset, a *UniqueIdentifier* and a *BusTag* to identify an offset in a specific adapter’s configuration space and performs whatever operations are necessary to acquire and return 16 bits of configuration information.

The function is provided only for drivers that need to interact with configuration space. On most buses, **MSMGetCardConfigInfo** will satisfy a driver's need.

For most buses, this call will do nothing. It has meaning only on buses that have a configuration address space that is separated from memory or I/O space (for example, a PCI bus).

See Also

MSMSearchAdapter, MSMGetUniqueIdentifier, MSMRdConfigSpace16, MSMRdConfigSpace32, MSMGetCardConfigInfo, MSMWrtConfigSpace8, MSMWrtConfigSpace16, MSMWrtConfigSpace32.

Example

```
DriverInit    proc
.
.
.
mov          ebx, UniqueIdentifier; from
              MSMSearchAdapter
mov          ecx, BusTag; BusTag for PCI
mov          eax, 2          ; PCI Vendor ID Offset
call         MSMRdConfigSpace16
.
.
.
DriverInit    endp
```


MSMRdConfigSpace32

Processor States	Entry State	
	EAX - <i>Offset</i>	contains an offset into the configuration space.
	EBX - <i>Unique-Identifier</i>	contains an architecture-dependent value returned by MSMGetUniqueIdentifier or MSMSearchAdapter . It specifies the location on the bus where the device is located
	ECX - <i>BusTag</i>	contains an architecture-dependent value returned by MSMSearchAdapter . It specifies the bus on which the operation is to be performed
	Interrupts	in any state
	Return State	
	EDX	unsigned read value
	Interrupts	preserved
	Preserved	EAX, EBX, ECX, ESI, EDI, EBP

Description

This function takes an offset, a *UniqueIdentifier* and a *BusTag* to identify an offset in a specific adapter’s configuration space and performs whatever operations are necessary to acquire and return 32 bits of configuration information.

The function is provided only for drivers that need to interact with configuration space. On most buses, **MSMGetCardConfigInfo** will satisfy a driver's need.

Note

For most buses, this call will do nothing. It has meaning only on buses that have a configuration address space that is separated from memory or I/O space (for example, a PCI bus).

See Also

MSMSearchAdapter, MSMGetUniqueIdentifier, MSMRdConfigSpace16, MSMRdConfigSpace32, MSMGetCardConfigInfo, MSMWrtConfigSpace8, MSMWrtConfigSpace16, MSMWrtConfigSpace32.

Example

```
DriverInit    proc
               .
               .
               .
               mov     ebx, UniqueIdentifier; from
                   MSMSearchAdapter
               mov     ecx, BusTag; BusTag for PCI
               mov     eax, 16      ; PCI Base Addr 0 Offset
               call    MSMRdConfigSpace32
               .
               .
               .
DriverInit    endp
```

MSMReadPhysicalMemory

Processor States	Entry State
	ECX number of bytes to read
	ESI physical source address (where to read data from)
	EDI logical destination address (where to transfer data to)
	Interrupts may be in any state
	Call during <i>DriverInit</i> before MSMRegisterHardwareOptions
	 Return State
	Preserved EBX, EBP, ESI, and EDI

Description

If the driver attempts to access shared RAM before calling **MSMRegisterHardwareOptions**, a page fault abend will occur on the server. Accesses to the shared RAM prior to registration do not normally happen unless the HSM must obtain additional information such as interrupt numbers or shared RAM buffer size for the configuration table.

This routine can be used to read information from a shared RAM physical address before hardware registration.

See Also

MSMWritePhysicalMemory

Example

```
mov     esi, SourceAddress      ; physical shared RAM address
                                   ; source
lea     edi, [ebx].MLIDInterrupt ; logical dest. in frame data
                                   ; space
mov     ecx, 1                  ; read 1 byte

call    MSMReadPhysicalMemory    ; transfer data
cmp     eax, 0                   ; check for errors
jne     ErrorReadingFromSharedMemory; Jump if so
```

MSM Procedures and Macros 7-103

MSMRegisterHardwareOptions

Processor States	Entry State
	Interrupts can be in any state
	Call at initialization time only
	Return State
	EAX = 0 New Adapter was successfully registered
	EAX = 1 New Frame Type was successfully registered
	EAX = 2 New Channel (multichannel adapters) was registered
	EAX > 2 pointer to an error message. (hardware registration failed)
	EBP pointer to the Adapter Data Space if successful
	EBX pointer to the Frame Data Space if successful
	Interrupts are preserved

Description	<p>This function must be called by the HSM's DriverInit routine to register the hardware options.</p> <p>On return from MSMRegisterHardwareOptions:</p> <ul style="list-style-type: none">• If EAX is 0, a new adapter was registered and the driver should continue with initializing the adapter. If a new adapter is being added, the memory associated with the Adapter Data Space is allocated and control returns to DriverInit with EBP pointing to that space.• If EAX is 1, a new frame type was registered for an existing adapter and the DriverInit routine is basically finished.• If EAX is 2, a new channel was registered for an existing multichannel adapter. The driver (and MSM) typically treat the registering of a new channel as a new adapter.• If EAX is > 2, the MSM was unable to register the hardware options (typically due to conflicts with existing hardware). In this case, EAX points to an error message which the driver should print using MSMPrintString. DriverInit should then return immediately to the operating system with EAX set to any nonzero value.
-------------	--

Example

```

DriverInit      proc
    .
    .
    .
    call    MSMParseDriverParameters
    call    MSMRegisterHardwareOptions
    cmp     eax,2
    ja      DriverInitError
    je      NewChannel
    cmp     eax,1
    je      NewFrame
    ; (Initialize for NewAdapter)
    .
    .
    .

DriverInitExit:
    xor     eax,eax
    ret

DriverInitError:
    mov     esi,eax
    call    MSMPrintString
    or      eax,-1
    ret

DriverInit      endp

```

MSMRegisterMLID

Processor States	Entry State
	EBP pointer to the Adapter Data Space
	EBX pointer to the Frame Data Space
	Interrupts may be in any state
	Call at process time only
	Return State
	EAX zero if successful; otherwise EAX points to an error message which the driver must print using MSMPrintString before returning to the operating system with EAX non- zero.
	Zero Flag set if successful; otherwise an error occurred.
	Interrupts are unchanged
	Preserved EBX and EBP

Description

After **DriverInit** has successfully initialized the adapter, it should call this routine to register the MLID with the Link Support Layer.

When this routine returns, the configuration table contains a valid board number. HSMs for intelligent bus master adapters may now pass the board number and frame ID information to the adapter if necessary.

Example

```
DriverInit      proc
                .
                .
                .
                call    MSMRegisterMLID
                jnz     RegisterMLIDError
                .
                .
                .
```

MSMRegisterResource

Processor States

Entry State

EBP	pointer to Adapter Data Space
EBX	pointer to a configuration table that was registered and was returned through <TSM> RegisterHSM .
EDX	pointer to an <i>ExtraConfig</i> structure that contains the resource(s) to be registered.
Interrupts	can be in any state
Call	at process or interrupt time

Return State

Flags	set according to EAX.
Interrupts	are unchanged.
Preserved	all registers except EAX.

Completion Code in EAX

SUCCESSFUL	The resources contained in the <i>ExtraConfig</i> parameter were successfully registered.
OUT_OF_RESOURCES	The hardware options could not be registered. This is typically due to conflicts with resources held by hardware devices.

Description

This routine lets an HSM register hardware resources that are not listed in the configuration table because it is full.

This routine cannot be called until **MSMRegisterHardwareOptions** has returned with a *New Adapter* or a *New Channel*.

Currently, only the two interrupts in the configuration table are supported. You cannot use **MSMRegisterResource** to register additional interrupts.

The *ExtraConfig* structure must always remain allocated, so that the MSM will return the resource if the HSM gets unloaded.

ExtraConfig Structure

This structure is defined in MSM.INC.

```
ExtraConfig  struc
ExtraConfigNextLinkdd0
EcReserved0dd0
EcReserved1dd0
EcReserved2dd0
EcReserved3 dd0
EcReserved4 dd0
EcReserved5dd0
EcReserved6 dd0
EcReserved7 dd0
EcReserved8 dd0
EcReserved9 dd0
IOConfigPointerdd0
ExtraConfig  ends
```

Field Descriptions:

EcReserved0...EcReserved0

These fields are reserved and must be set to 0.

IOConfigPointer

This field contains a pointer to the IOConfig structure filled in by caller.

IOConfig Structure

This structure is defined in ODI.INC. IOConfig is basically a substructure of the Driver Config Table, see the Config Table structure's field description in Chapter 3 of the HSM specification.

```
IOConfig  struc
    IO_Link            dd    0
    IO_SharingFlags    dw    0
    IO_Slot            dw    0
    IO_IOPort0         dw    0
    IO_IORange0        dw    0
    IO_IOPort1         dw    0
    IO_IORange1        dw    0
    IO_MemoryDecode0   dd    0
    IO_Length0         dw    0
    IO_MemoryDecode1   dd    0
    IO_Length1         dw    0
    IO_Interrupt0       db    0
    IO_Interrupt1       db    0
    IO_DMALine0        db    0
    IO_DMALine1        db    0
    IO_ResourceTag      dd    0
    IO_Config          dd    0
    IO_CommandString    dd    0
    IO_LogicalName      db    18 dup (0)
    IO_LinearMemory0    dd    0
    IO_LinearMemory1    dd    0
    IO_ChannelNumber    dw    0
    IO_BusTag          dd    0
    IO_ConfigMajorVer   db    0
    IO_ConfigMinorVer   db    0
IOConfig  ends
```

MSMReRegisterHardwareOptions

Allows an HSM to deregister its current hardware options and register a new set of hardware options.

Processor States	Entry State	
	EBP	pointer to the Adapter Data Space
	EBX	pointer to an IOConfig structure (new) that contains the new hardware options to be registered.
	ESI	pointer to an ECB whose ESR is called if MSMReRegisterHardwareOptions returns RESPONSE_DELAYED. If NULL,_BAD_PARAMETER will be returned.
	Interrupts	can be in any state.
	Execute	at process or interrupt time
	Return State	
	Flags	set according to EAX
	Interrupts	are unchanged
	Preserved	all registers except EAX
Completion Code in EAX		
SUCCESSFUL		Hardware options were successfully reregistered.
OUT_OF_RESOURCES		Unable to register the hardware options due to conflicts in resources with another device.
BAD_PARAMETER		If an input parameter was invalid.
FAIL		The adapter was not in a shutdown state before the call was made.
RESPONSE_DELAYED		The operation of deregistering and registering hardware options could not be completed at the present time. An asynchronous process will be scheduled to complete the operation at a later time.

Description

Note



If **MSMRegisterMLID** has been called the adapter must be shutdown using **MSMShutdownMLID** before **MSMReRegisterHardwareOptions** is called.

MSMReRegisterHardwareOptions will deregister the current set of hardware options held by the HSM for an adapter as registered previously through **MSMRegisterHardwareOptions** or through a previous call to **MSMReRegisterHardwareOptions**. All hardware options in the new IOConfig table will then be registered for the adapter. Any hardware option in the new IOConfig table that are not to be registered must be set as not in use as described in the Driver Configuration Table. The MLIDResourceTag from the old Config Table will be used when registering the new hardware options. The fields of the new IOConfig table correspond to the fields in the DriverConfigTemplate structure starting with the MLIDLink field and end with the **MLIDIOConfigMinorVer**.

If all hardware options in the new IOConfig table were successfully registered, **MSMReRegisterHardwareOptions** will update all configuration tables of the adapter to reflect the newly registered hardware options.

If **MSMReRegisterHardwareOptions** cannot complete the operation at the present time, an asynchronous process will be scheduled to complete the operation later. Once the asynchronous operation is complete, the asynchronous ECB's ESR routine will be called to report the final return value of the operation. The return value will be stored in the asynchronous ECB's ECB_Status field.

Upon successful return from **MSMReRegisterHardwareOptions**, the HSM is responsible for putting the adapter in a functional state. If a new interrupt was registered, the HSM must call **MSMSetHardwareInterrupt**.

MSMResetMLID

Processor States	Entry State
	EBP pointer to Adapter Data Space
	Interrupts are disabled but may be enabled during the call
	Call at process time only
	Return State
	Flags set according to EAX.
	Interrupts are disabled.
	Preserved EBP, EBX


Completion Code in EAX	
SUCCESSFUL	Reset was successful
BAD_PARAMETER	An input parameter was invalid or NULL.
FAIL	The operation failed

Description

If an HSM needs to reset the driver, it must use this function to do so. **MSMResetMLID** puts the driver in a safe state and then calls **DriverReset**.

If the reset is successful, the SHUTDOWN flag in *MSMStatusFlags* is cleared by the MSM. The MSM also produces a NESL suspend event for **MLIDReset**.

In previous versions of this specification, and under certain circumstances, HSMs could call their own **DriverReset** routines. However, HSMs written to this specification must not do so. HSMs written to this specification must use **MSMResetMLID**.

Note

This function does not restart polling if polling was suspended by **MSMSuspendPolling**. **MSMResumePolling** must be called to restart polling.

MSMResetMLID cannot be called until after **MSMRegisterMLID** has been called.

MSMResumePolling

Processor States

Entry State

EBP	pointer to Adapter Data Space
Interrupts	can be in any state
Execute	at process or interrupt time

Return State

Interrupts	are unchanged
Preserved	all registers

Completion Code in EAX

SUCCESSFUL	Polling was successfully resumed.
BAD_COMMAND	The call is invalid because MSMEnablePolling and/or MSMSuspendPolling have never been called.

Description

Turns polling back on after **MSMSuspendPolling** has suspended it. This call is only necessary if **MSMSuspendPolling** was called previously. When **MSMEnablePolling** is called, polling will start up active.

The POLLSUSPENDED flag in the MLIDStatusFlags (bit 4) is cleared by the MSM when this function is called.

MSMReturnDriverResources


Processor States	Entry State	
	Interrupts	are disabled
	Call	at process time only
	Return State	
	Interrupts	remain disabled
	Preserved	NO registers are preserved

Description

MSMReturnDriverResources must be called to return resources if the HSM's **DriverInit** routine is unable to initialize the adapter. It should only be called after **<TSM>RegisterHSM** has completed successfully. However if there is an error during any of the following procedures they will call **MSMReturnDriverResources**, and it must not be called a second time or an abend or Interrupt 3 may result.

The following procedures will call **MSMReturnDriverResources** if they do not complete successfully:

- <TSM>RegisterHSM
- MSMEnablePolling
- MSMParseDriverParameters
- MSMRegisterHardwareOptions
- MSMRegisterMLID
- MSMScheduleAESCcallback
- MSMScheduleIntTimeCallback
- MSMSetHardwareInterrupt

Warning  The HSM must not call **MSMReturnDriverResources** after one of the procedures listed above returns an error condition. It may cause the server to abend.

Please note that the **MSMDriverRemove** procedure calls **MSMReturnDriverResources** upon completing successfully.

Example

```

DriverInit          proc

    Cpush
    .
    .
    .
    call    <TSM>RegisterHSM
    jnz     DriverInitError
    .
    .
    .
    [*** Initialize the Adapter ***]
    call    DriverReset
    jnz     DriverInitResetError
    .
    .
    .

DriverInitResetError:
    push    eax
    call    MSMReturnDriverResources
    pop     eax

DriverInitError:
    mov     esi, eax
    call    MSMPrintString
    or      eax, 1
    Cpop
    ret

DriverInit          endp

```

MSMReturnMultipleRCBs

Processor States

Entry State

ESI	pointer to first RCB in chain
Interrupts	can be in any state
Execute	at process or interrupt time

Return State

Interrupts	are disabled
Preserved	EBX, ECX, EDX, EDI & EBP

Description

This function returns multiple RCBs that are chained together in a linked list through the *ECBLink* field, which is the first double word of the *RCBDriverWS* field.

This function is used to discard multiple RCBs. It does not process RCBs.

See Also

- MSMAllocateMultipleRCBs
- MSMAllocateRCB
- MSMReturnRCB

MSMReturnNotificationECB (macro)

MSMFastReturnNotificationECB (macro)

Processor States]

Entry State

ESI	pointer to the notification ECB
Interrupts	can be in any state
Execute	at process or interrupt time

Return State

Interrupts	are disabled
Preserved	MSMReturnNotificationECB - (ESI, EDI, and EBP are preserved)
	MSMFastReturnNotificationECB - (Assume all registers are destroyed)

Description

Drivers that support outside management NLMs (such as HMI or CSL) use these macros to process notification ECBs containing management alert information.

If the hardware generates an alert, the HSM obtains a notification ECB using **MSMAllocateRCB**. This procedure requires a packet size on entry. The size specified will depend on the amount of information that must be passed up to the management application. The driver fills in the ECB with the notification information according to the driver management specification, sets ESI to point to the ECB, and returns the notification ECB using one of these macros.

MSMReturnNotificationECB places the ECB in the LSLs holding queue and waits for the HSM to call

MSMServiceEvents before passing it to the management NLM.

MSMFastReturnNotificationECB passes the ECB immediately to the management application.

Example

```
HubResetNotification      proc
    .
    .
    .
    mov     esi, 4
    call    MSMAllocateRCB      ; Get notification
    ECB
    .
    .      (Fill in all required notification information)
    .
    mov     esi, ECBPtr         ; Point to the ECB
    MSMFastReturnNotificationECB ; Return the ECB
    directly to the
    .
    .      ; management
    application
    .
    .
```

MSMReturnRCB (macro)

Processor States

Entry State

ESI	pointer to the unneeded RCB
EBP	pointer to the adapter data space
Interrupts	can be in any state
Execute	at process or interrupt time

Return State

Interrupts	are disabled
Preserved	EBX, ECX, EDX, EBP, and EDI

Description

MSMReturnRCB returns an unneeded RCB to the LSL. This routine is called to discard the RCB, not to process it. To return an RCB for processing, see **<TSM>RcvComplete** or **<TSM>ProcessGetRCB**.

Example

```

mov     esi, [ebp].ReceiveQueueHead    ; ESI ->
First RCB
mov     [ebp].ReceiveQueueHead, 0      ; Clear
pointer
or      esi, esi                        ; Valid
RCB?
jz      ShutdownAllRCBsReturned        ; Jump if
not

```

ShutdownReturnRCBLoop:

```

mov     ecx, [esi].RCBDriverWS+4       ; ECX ->
Next RCB
MSMReturnRCB                           ; Return
RCB
mov     esi, ecx                        ; ESI -
>Next RC B
or      esi, esi                        ; Valid
RCB?
jnz     ShutdownReturnRCBLoop          ; Jump if
so

```

MSMScanBusInfo

Processor States		Entry State
ESI - <i>ScanSequence</i>		must be initialized to -1 for the first search and then passed back in for subsequent call
Interrupts		can be in any state
		Return State
EBX - <i>BusName</i>		pointer to a static, null-terminated, architecture-dependent string that is determined by the platform developer
ECX <i>BusType</i>		contains a value that indicates the bus type as defined in ODI_NBI.INC. The currently defined values are: 0 = ODI_BUSTYPE_ISA 1 = ODI_BUSTYPE_MCA 2 = ODI_BUSTYPE_EISA 3 = ODI_BUSTYPE_PCMCIA 4 = ODI_BUSTYPE_PCI 8 = ODI_BUSTYPE_CARDBUS
EDX <i>BusTag</i>		contains an architecture-dependant value that specifies the bus
ESI <i>ScanSequence</i>		contains the scan sequence value to be used for the next search
Flags		are set according to EAX
Interrupts		are preserved
Preserved		EBP

Completion Codes in EAX	Code	Description
	SUCCESSFUL	indicates the operation completed successfully
	ITEM_NOT_PRESENT	indicates there are no more buses or function is not available
	BAD_PARAMETER	the <i>ScanSequence</i> was invalid

Description

MSMScanBusInfo searches the system for available buses. The *BusName* string must not be modified by the caller. If the caller needs to reference the string at a later time, a local copy of the string must be made.

Example

```

DriverInit      proc
                .
                .
                .
                mov     esi, -1      ; -1 for first time thru
ScanBusLoop:
                call    MSMScanBusInfo; scan for next bus
                jnz     Done_Error_ScanningBus; done/error
                .
                .
                .
                jmp     ScanBusLoop; scan for next bus
                .
                .
                .
DriverInit      endp


```

MSMScheduleAESCallback

Processor States	Entry State
	EBP pointer to the Adapter Data Space
	EAX Time Interval in ticks (1 tick = 1/18 sec)
	Interrupts can be in any state, but are disabled during the call
	Call only at initialization time (during DriverInit)
	Return State
	EAX zero if successful; otherwise EAX points to an error message which the driver must print using MSMPrintString before returning to the operating system with EAX non- zero.
	Zero Flag set if successful; otherwise an error occurred.
	Interrupts are preserved
	Preserved EBX and EBP

Description

This routine can be called during **DriverInit** to enable a periodic call back to the HSM's **DriverAESCallback** routine. Once enabled, **DriverAESCallback** is invoked during process time at the intervals specified by EAX. The MSM sets up the Adapter and Frame Data Space before calling **DriverAESCallback** and automatically schedules a new call back on return.

Note  **MSMScheduleAESCallback** is used when process-time-only routines are called by **DriverAESCallback**. **MSMScheduleTimer** should be used to handle timer events instead of **MSMScheduleAESCallback** when possible (see **MSMScheduleTimer**).


Example


```
DriverInit      proc
.
.
.
mov     eax, 18                ; Schedule call
back in 18 ticks
call    MSMScheduleAESCallback
jnz     ScheduleCallbackError
.
.
.
```

MSMScheduleIntTimeCallBack

Processor States	Entry State
	EBP pointer to the Adapter Data Space
	EAX Time Interval in ticks (1 tick = 1/18 sec)
	Interrupts are disabled and remain disabled
	Call only at initialization time (during <i>DriverInit</i>)
	Return State
	EAX zero if successful; otherwise EAX points to an error message which the driver must print using MSMPrintString before returning to the operating system with EAX nonzero.
	Zero Flag set if successful; otherwise an error occurred.
	Interrupts are disabled
	Preserved EBX and EBP

Description This routine can be called during **DriverInit** to enable a periodic call back to the HSM's **DriverINTCallBack** routine. Once enabled, **DriverINTCallBack** is invoked during the timer tick interrupt at the interval specified by EAX. The MSM sets up the Adapter and Frame Data Space before calling **DriverINTCallBack** and automatically schedules a new call back on return.

Note  **DriverINTCallBack** cannot be used if calls are made to routines which can be invoked only at process time. **DriverAESCCallBack** should be used instead (see **MSMScheduleAESCCallBack**). Also it is critical that ALL drivers call **MSMRegisterMLID** before **MSMScheduleIntTimeCallBack** in order for the driver to work properly in SMP mode.

Note  **MSMScheduleTimer** is now the preferred method for setting up timer events.

Example

```
DriverInit      proc
                .
                .
                .
                mov     eax, 18                ; Schedule call
                back in 18 ticks
                call     MSMScheduleIntTimeCallBack
                jnz      ScheduleCallBackError
```



```

TIMER_STRUCTURE

TIMER_STRUCTUREstruc
    TimerNextLinkdd    ?
    TimerProcedurePtrdd ?
    TimerType          dd      ?
    TimerInterval      dd      ?
    TimerContext        dd      ?
    TimerReserveddb    30 dup (?)
TIMER_STRUCTUREends

```

Field descriptions:

TimerNextLink

Used to link TIMER_STRUCTUREs together. Reserved for use by MSM.

TimerProcedurePtr

A pointer to the procedure that is called after the specified time Interval. On call EBP will point to the Adapter Data Space and EBX will point to the default Frame Data Space.

TimerType

Used to specify one of the following timer types:

AES_TYPE_PRIVILEGED_ONE_SHOT

Call only once at privileged time

AES_TYPE_PRIVILEGED_CONTINUOUS

Call at privileged time

AES_TYPE_PROCESS_ONE_SHOT

Call only once at process time

AES_TYPE_PROCESS_CONTINUOUS

Call at process time

TimerInterval

The time in milliseconds to wait before calling TimerProcedure.

TimerContext

Reserved for use by MSM.

MSM Procedures and Macros **7-127**

TimerReserved

Reserved for use by MSM.

The TIMER_STRUCTURE must remain allocated until the driver is unloaded.

MSMSearchAdapter

Processor States

Entry State

EAX - <i>Product-IDLength</i>	contains the length of the product ID
EBX - <i>ProductID</i>	contains a pointer to a bus architecture-dependent parameter that uniquely identifies an adapter board/peripheral/system option. The product ID appears in memory as defined by the specification for the <i>BusType</i> . For example, on an EISA bus, the EISA product ID in memory as defined in the EISA Specification
ECX - <i>BusType</i>	A bus type as defined in ODI_NBI.INC. The currently defined values are: 0 = ODI_BUSTYPE_ISA 1 = ODI_BUSTYPE_MCA 2 = ODI_BUSTYPE_EISA 3 = ODI_BUSTYPE_PCMCIA 4 = ODI_BUSTYPE_PCI 8 = ODI_BUSTYPE_CARDBUS
ESI - <i>ScanSequence</i>	must be initialized to -1 on the first search for each <i>ProductID</i> , and passed back on subsequent calls for the same <i>ProductID</i> .
Interrupts	can be in any state

Return State

CX - <i>Instance-Number</i>	the hardware instance number for the device. The hardware instance number is guaranteed unique across all devices in the system and in many cases is the physical slot number.
EDX - <i>BusTag</i>	an architecture-dependant value that specifies the bus on which the device was located
ESI - <i>Scan-Sequence</i>	the scan sequence value to be used for the next search
EDI - <i>Unique-Identifier</i>	contains an architecture-dependent value returned by MSMGetUniqueIdentifier or MSMSearchAdapter . It specifies the location on the bus where the device is located
Flags	are set according to EAX

MSM Procedures and Macros **7-129**

Interrupts	are preserved
Preserved	EBP, EBX

Completion Codes in EAX	Code	Description
	SUCCESSFUL	requested product was found
	ITEM_NOT_PRESENT	requested product is not present

Description

MSMSearchAdapter takes a bus type and a pointer to a product ID and returns a *BusTag* and a slot number, where the specified device is found. This function should be used only if the driver's adapter has a unique product ID associated with it that can be read. The product ID must be retrievable according to some accepted standard, such as EISA, MCA or PCI.

Example

```

ProductID      db      ...
ProductIDLen   equ     ...

DriverInit     proc
                .
                .
                .
                mov     esi, -1      ; for first time thru

SearchAdapterLoop:
                mov     eax, ProductIDLen
                lea     ebx, ProcudtID
                mov     ecx, BusType; bus type from
                                ; MSMScanBusInfo or constant
                                ; like ODI_BUSTYPE_MCA
                call    MSMSearchAdapter
                jnz     Done_Error_SearchAdapter
                .
                .
                .
                jmp     SearchAdapterLoop; go find next NIC
                .
                .
                .
DriverInit     endp

```

MSMServiceEvents (macro)

Processor States	Entry State	
	Interrupts	can be in any state
	Execute	at process or interrupt time
	Return State	
	Interrupts	are disabled on completion, but might have been enabled during execution
	Preserved	NO registers are preserved

Description	<p>If the HSM has used <TSM>SendComplete, <TSM>RcvComplete or <TSM>ProcessGetRCB, it must use either MSMServiceEvents or MSMServiceEventsAndRet before it exits back to the operating system.</p>
	<p>If the HSM must execute any instructions after it services events, then it must use MSMServiceEvents instead of MSMServiceEventsAndRet.</p>
	<p>In the example below, the adapter supports shared interrupts. In this case, the operating system requires that EAX equal 0 if the interrupt is for the HSM. The HSM must use MSMServiceEvents and set EAX to 0 before returning. If MSMServiceEventsAndRet is used, the HSM returns before it is able to set EAX to 0. If the HSM does not support shared interrupts, it can return immediately after servicing events, therefore, the MSMServiceEventsAndRet macro should be used.</p>
	<p>If the HSM uses <TSM>FastSendComplete, <TSM>FastRcvComplete, or <TSM>FastProcessGetRCB exclusively, it does not need to use MSMServiceEvents. The “fast” routines service events before returning.</p>

Example

```
DriverISR      proc
               .
               .
DriverISRExit:  .
               MSMServiceEvents      ; Service Events queue
               xor     eax, eax        ; Inform operating system
               that interrupt          ; was ours
               ret
DriverISR      endp
```


MSMServiceEventsAndRet (macro)

Processor States	Entry State	
	Interrupts	can be in any state
	Execute	at process or interrupt time
	Return State	
	Note	this macro does not return to the HSM

Description

If the HSM has used **<TSM>SendComplete**, **<TSM>RcvComplete**, or **<TSM>ProcessGetRCB**, it must use either **MSMServiceEvents** or **MSMServiceEventsAndRet** before it exits back to the operating system.

Since this macro automatically returns, **MSMServiceEventsAndRet** must be the last executable line of code in the routine. If the HSM must execute any instructions after servicing events, it must use the **MSMServiceEvents** macro which does not automatically return.

If the HSM uses **<TSM>FastSendComplete**, **<TSM>FastRcvComplete**, or **<TSM>FastProcessGetRCB** exclusively, it does not need to use **MSMServiceEvents**. The "fast" routines service events before returning.

```
Example      DriverISR      proc
              .
              .
              .

              DriverISRExit:

                      MSMServiceEventsAndRet      ; Service Events
              and Return.

              DriverISR      endp
```

MSMSetHardwareInterrupt

Processor States

Entry State

EBP	pointer to the Adapter Data Space
EBX	pointer to the Frame Data Space
Interrupts	are disabled and remain disabled
Call	at process time

Return State

EAX	zero if successful; otherwise EAX points to an error message which the driver must print using MSMPrintString before returning to the operating system with EAX nonzero.
Zero Flag	set if successful; otherwise an error occurred.
Interrupts	are disabled
Preserved	EBX and EBP

Description

The HSM's **DriverInit** routine will call this function to set up a hardware interrupt.

Example

```
call    MSMRegisterHardwareOptions
call    MSMSetHardwareInterrupt
jnz     SetHardwareIntError
```

MSMShutdownMLID

Processor States		Entry State	
		EBP	pointer to Adapter Data Space
		ECX	shutdown type, set to zero if permanent shutdown, otherwise a partial shutdown is required.
		Interrupts	are disabled but may be enabled during the call
		Call	at process time only
		Return State	
		Flags	set according to EAX.
		Interrupts	are disabled.
		Preserved	EBP, EBX
		Completion Code in EAX	
	SUCCESSFUL	Shutdown was successful.	
	BAD_PARAMETER	An input parameter was invalid or NULL.	
	FAIL	The operation failed.	

Description

If **DriverShutdown** needs to be called from within the HSM it is done by this function. The MSM puts the driver in a safe state and then calls **DriverShutdown**. If a partial shutdown was performed a call to **MSMResetMLID** will bring the driver out of shutdown state

If the operation is successful the SHUTDOWN flag in **MSMStatusFlags** is set by the MSM. The MSM also produces a NESL suspend event for MLID Shutdown.

Note



MSMShutdownMLID can not be called until after **MSMRegisterMLID** has been called.

In prior versions of this specification an HSM could call its own **DriverShutdown** routine; HSMs written to this version of the specification must use **MSMShutdownMLID**.

MSMSuspendPolling

Processor States	Entry State	
	EBP	pointer to Adapter Data Space
	Interrupts	can be in any state
	Execute	at process or interrupt time
	Return State	
	Interrupts	are unchanged
	Preserved	all registers
Completion Code in EAX		
SUCCESSFUL	Polling was successfully suspended.	
BAD_COMMAND	Call is invalid because MSMEnablePolling was never called to begin with or polling is currently suspended (redundant suspend calls).	

Description

Suspends the calling of the **DriverPoll** procedure until **MSMResumePolling** is called. The polling procedure is very expensive, especially in a Multi-Processor environment. Each time **DriverPoll** is called the Mutex must be acquired and both **DriverDisableInterrupts** and **DriverEnableInterrupts** must be called. This keeps the Mutex held a high percentage of the time and causes bus traffic. Most of the time that **DriverPoll** is called there is no usable work that the driver needs to do yet while in the poll procedure the driver is locked out from receiving interrupts, **DriverSends**, etc. Use **MSMSuspendPolling** to temporarily stop the driver from being polled when it is known that there is no usable work to do.

The POLLSUSPENDED flag in MLIDStatusFlags (bit 4) is set by the MSM when **MSMSuspendPolling** is called and cleared by the MSM when **MSMResumePolling** is called and can be inspected by the HSM to determine the current polling status.

Example

```
DriverPoll proc
.
.
.
cmp [ebp].PollStatus, BUSY
je   Don't Suspend Polling
call MSMSuspendPolling
.
.
.
.
```

MSMUpdateConfigTables

Allows a HSM to update all copies of the configuration table for an adapter.

Processor States	Entry State
	EBP pointer to the Adapter Data Space
	EBX pointer to a configuration table
	Interrupts can be in any state
	Execute at process or interrupt time
	Return State
	EAX zero if successful, none-zero otherwise
	Preserved all registers except EAX
	Interrupts are unchanged

Completion Code in EAX	
SUCCESSFUL	All Configuration tables for the adapter were updated.
BAD_PARAMETER	An input parameter was invalid.

Description

MSMUpdateConfigTables copies the following configuration table fields from the configuration table parameter to all configuration tables of a adapter (all other configuration table fields are ignored):

- MLIDNodeAddress
- MLIDModeFlags
- MLIDMaximumSize
- MLIDCardName
- MLIDShortName
- MLIDTransportTime
- MLIDLineSpeed
- MLIDCFG_SGCount
- MLIDPrioritySup
- MLIDFlags

`MLIDSendRetries`

`MLIDSharingFlags` (the shutdown bit `IODetachedBit` (see `ODI.INC`) is the only bit copied).

MLIDMaxRecvSize and **MLIDRecvSize** are automatically adjusted by the TSM based on *MLIDMaximumSize*.

A HSM can call **MSMUpdateConfigTables** any time to update an adapter's configuration tables. All fields copied from the configuration table parameter must be valid before **MSMUpdateConfigTables** is called.

During a driver's initialization for an adapter, **MSMReRegisterHardwareOptions** automatically updates the adapter's configuration tables. A call to **MSMUpdateConfigTables** is only necessary if the fields copied from the *configTable* parameter are modified after the call to **MSMRegisterHardwareOptions**.

MSMUpdateConfigTables upon successful completion will produce a NESL Service/Status Change event to inform consumers of the event that the configuration tables for the adapter have been updated.

MSMReRegisterHardwareOptions is used to update the IOConfig table fields of the configuration table. **MSMUpdateConfigTables** and **MSMReRegisterHardwareOptions** are complementary and care must be taken to use the correct one.

MSMWritePhysicalMemory

Processor States	Entry State	
	ECX	number of bytes to write
	ESI	logical source address (where to read data from)
	EDI	physical destination address (where to transfer data to)
	Interrupts	may be in any state
	Call	during DriverInit before MSMRegisterHardwareOptions
	Return State	
	Preserved	EBX, EBP, ESI, and EDI

Description

If the driver attempts to access shared RAM before calling **MSMRegisterHardwareOptions**, a page fault abend will occur on the server. Accesses to the shared RAM prior to registration do not normally happen unless the HSM must obtain additional information such as interrupt numbers or shared RAM buffer size for the configuration table.

This routine can be used to write information to a shared RAM physical address before hardware registration.

See Also

MSMReadPhysicalMemory

Example

```
mov    edi, DestinationAddress    ; physical shared RAM address
lea    esi, [ebx].MLIDNodeAddress ; logical source is in frame data
                                ;space
mov    ecx, 6                     ; write 6 byte node address

call   MSMWritePhysicalMemory     ; transfer data
cmp    eax, 0                     ; check for errors
jne    ErrorWritingToSharedMemory ; Jump if so
```


MSMWrtConfigSpace8

Processor States		Entry State
	EAX - <i>Offset</i>	contains an offset into the configuration space.
	EBX - <i>Unique-Identifier</i>	contains an architecture-dependent value returned by MSMGetUniqueIdentifier or MSMSearchAdapter . It specifies the location on the bus where the device is located
	ECX - <i>BusTag</i>	contains an architecture-dependent value returned by MSMSearchAdapter . It specifies the bus on which the operation is to be performed
	EDX - <i>WriteVal</i>	DL contains an unsigned 8-bit value to write
	Interrupts	in any state
		Return State
	Interrupts	preserved
	Preserved	EAX, EBX, ECX, EDX, ESI, EDI, EBP

Description

This function takes an offset, a *UniqueIdentifier* and a *BusTag* to identify an offset in a specific adapter’s configuration space and performs whatever operations are necessary to deliver the value to the specified location.

The function is provided only for drivers that need to interact with configuration space.

Note

For most buses, this call will do nothing. It has meaning only on buses that have a configuration address space that is separated from memory or I/O space (for example, a PCI bus).

See Also

MSMSearchAdapter, MSMGetUniqueIdentifier, MSMGetCardConfigInfo, MSMRdConfigSpace8, MSMRdConfigSpace16, MSMRdConfigSpace32, MSMWrtConfigSpace16, MSMWrtConfigSpace32

Example DriverInit proc

7-144 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

```

    .
    .
    .
    mov     eax, OffsetTableEntry
    mov     ebx, UniqueIdentifier
    mov     ecx, BusTag
    mov     dl, ValueToWrite
    call    MSMWrtConfigSpace8
    .
    .
    .
DriverInit endp
```


MSMWrtConfigSpace16

Processor States	Entry State	
	EAX - <i>Offset</i>	contains an offset into the configuration space.
	EBX - <i>Unique-Identifier</i>	contains an architecture-dependent value returned by MSMGetUniqueIdentifier or MSMSearchAdapter . It specifies the location on the bus where the device is located
	ECX - <i>BusTag</i>	contains an architecture-dependent value returned by MSMSearchAdapter . It specifies the bus on which the operation is to be performed
	EDX - <i>WriteVal</i>	DX contains an unsigned 16-bit value to write
	Interrupts	in any state
Return State		
Interrupts		preserved
Preserved		EAX, EBX, ECX, EDX, ESI, EDI, EBP

Description

This function takes an offset, a *UniqueIdentifier* and a *BusTag* to identify an offset in a specific adapter’s configuration space and performs whatever operations are necessary to deliver the value to the specified location.

This function is provided only for drivers that need to interact with configuration space.

Note

For most buses, this call will do nothing. It has meaning only on buses that have a configuration address space that is separated from memory or I/O space (for example, a PCI bus).

See Also

MSMSearchAdapter, MSMGetUniqueIdentifier, MSMGetCardConfigInfo, MSMRdConfigSpace8, MSMRdConfigSpace16, MSMRdConfigSpace32, MSMWrtConfigSpace16, MSMWrtConfigSpace32

Example

```
DriverInit      proc
.
.
```

```

    .
    mov     eax, OffsetTableEntry
    mov     ebx, UniqueIdentifier
    mov     ecx, BusTag
    mov     dx, ValueToWrite
    call    MSMWrtConfigSpace16
    .
    .
    .
DriverInit endp
```

MSM Procedures and Macros **7-147**

MSMWrtConfigSpace32

Processor States	Entry State	
	EAX - <i>Offset</i>	contains an offset into the configuration space.
	EBX - <i>Unique-Identifier</i>	contains an architecture-dependent value returned by MSMGetUniqueIdentifier or MSMSearchAdapter . It specifies the location on the bus where the device is located
	ECX - <i>BusTag</i>	contains an architecture-dependent value returned by MSMSearchAdapter . It specifies the bus on which the operation is to be performed
	EDX - <i>WriteVal</i>	contains unsigned 32-bit value to write
	Interrupts	in any state
	Return State	
	Interrupts	preserved
	Preserved	EAX, EBX, ECX, EDX, ESI, EDI, EBP

Description

This function takes an offset, a *UniqueIdentifier* and a *BusTag* to identify an offset in a specific adapter’s configuration space and performs whatever operations are necessary to deliver the value to the specified location.

This function is provided only for drivers that need to interact with configuration space.

For most buses, this call will do nothing. It has meaning only on buses that have a configuration address space that is separated from memory or I/O space (for example, a PCI bus).

See Also

MSMSearchAdapter, MSMGetUniqueIdentifier, MSMGetCardConfigInfo, MSMRdConfigSpace8, MSMRdConfigSpace16, MSMRdConfigSpace32, MSMWrtConfigSpace16, MSMWrtConfigSpace32

Example DriverInit proc

7-148 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)


```

    .
    .
    .
    mov     eax, OffsetTableEntry
    mov     ebx, UniqueIdentifier
    mov     ecx, BusTag
    mov     edx, ValueToWrite
    call    MSMWrtConfigSpace32
    .
    .
    .
DriverInit endp
```

MSMYieldWithDelay

Processor States	Entry State
	EBX pointer to the Frame Data Space
	EBP pointer to the Adapter Data Space
	Interrupts can be in any state
	Execute at process time only
	Return State
	Interrupts are in the same state as when the routine was called
	Preserved EBX, EBP, ESI, and EDI

Description **MSMYieldWithDelay** places the task last on the list of active tasks to be executed. This routine must be called only at process time because it suspends the process and could change the machine state. It must be used only in the driver initialization and driver remove procedures.

Example

```
.
. (initialization)
.
call     MSMGetMicroTimer      ; get current count
neg      eax
mov      edi, eax              ; EDI = EAX negated

DriverShutdownWait:
    MSMYieldWithDelay          ; let other processes run
    call     MSMGetMicroTimer  ; get current count
    add      eax, edi           ; EAX = microseconds expired
    cmp      eax, 50            ; 50us passed?
    jb      DriverShutdownWait ; jump if not
```

Appendix **A** *Building the HSM*

Development Process

This appendix describes the process of creating, assembling, linking, and loading a NetWare server LAN driver.

Creating the Source Files

All NetWare server drivers are written in 386 assembly code using 32-bit register operations. Chapters 2 through 7 provide the details for writing the driver. The TOOLS directory on the *LAN Driver Developer's Kit* CD has source code you may use as an example when developing your driver. An example source file might be named *Driver.386*. Additional include files are also required as described in Chapter 1.

Assembling the Source Files

The *Driver.386* source file assembles into an object file, *Driver.OBJ*. The driver may consist of one or more object files. An assembler that supports the use of 32-bit registers is required. Novell engineers currently use the 386ASMP (v4.1 or later) protected mode assembler by Phar Lap Software, Inc.

Note



Drivers must be assembled with the case sensitive option.

```
386ASMP ne2 - fullwarn -twocase
```

Building the HSM **A-1**

Linking the Object Files

The NetWare linker (NLMLINKP) converts *Driver.OBJ* and any other object files that make up the driver into a super object file called *Driver.LAN*. NLMLINKP requires a linker definition file to create a NetWare Loadable Module. The linker definition file is described below. To use the linker, type:

```
nlmlinkp Driver
```

(where *Driver* is the name of the linker definition file).

Linker Definition File

Each NetWare loadable module must have a corresponding definition file with a “.DEF” extension. This file is needed by the NetWare linker, NLMLINKP. All definition file information can also be embedded inside a make file and the make file can produce the definition file. 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 illustration is an example definition file that can be used to create a LAN driver. The file consists of keywords followed by data. The keywords may be upper or lower case.

Example Definition File

Keyword	Data
* TYPE 1	
* DESCRIPTION	“NetWare NE2000”
* VERSION	5,30,2
* OUTPUT	<drivername>
* INPUT	<drivername>
* START	DriverInit
* EXIT	DriverRemove
MODULE	ETHERTSM
REENTRANT	

* IMPORT	EtherTSMRegisterHSM EtherTSMGetRCB EtherTSMRcvComplete EtherTSMFastRcvComplete EtherTSMSendComplete EtherTSMFastSendComplete EtherTSMGetNextSend EtherTSMUpdateMulticast MSMAAlertFatal MSMAAlloc MSMDriverRemove MSMFree MSMParseDriverParameters MSMPrintString MSMRegisterHardwareOptions MSMRegisterMLID MSMReturnDriverResources MSMScheduleIntTimeCallBack MSMHardwareInterrupt
----------	---

* Required Keywords

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 description string in the header of the <Driver>.LAN file. This string describes the loadable module and is from 1 to 127 bytes long. The console commands: MODULES, CONFIG, and LOAD display this description string on the file server console.

Examples of the description string are shown here:

```
NetWare NE2000
3Com EtherLink Plus 3C503
```

OUTPUT Tells the linker what to name the output file.

INPUT Tells the linker what OBJ files to include in the loadable module. It is not necessary to use the filename extension in this list.

Building the HSM **A-3**

START Tells the linker the name of the loadable module's initialization routine, in this case *DriverInit*. This is the procedure the NetWare loader will call when the module is loaded.

EXIT Tells the linker the name of the loadable module's remove routine, in this case *DriverRemove*. The UNLOAD command uses this routine to unload the module from file server memory.

REENTRANT Allows the driver to be loaded more than once, but only have the driver's code copied into memory the first time.

MAP Tells the linker to create a map file.

IMPORT Tells the linker which NetWare variables and routines the loadable module must access.

EXPORT A list of variable and procedure names resident in the loadable module that the loadable module wants to make available to other loadable modules.

MODULE The loadable modules that must be loaded before the loadable module defined by this file is loaded. If the necessary loadable modules are not already in file server memory, the loader will attempt to find and load them. If it cannot find them, the loader will not load the current module.

CUSTOM The name of a file that contains custom firmware 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. This allows public labels to be accessible as symbols in NetWare's resident debugger.

CHECK Contains the name of the loadable module's check procedure. Both the UNLOAD and DOWN console commands call a loadable module's check procedure if one exists. A LAN driver's check procedure might check to see if a LAN board is currently being accessed and return a non-zero value to the NetWare operating system if the board is busy. The NetWare OS can then display a message warning the console operator that the board is busy.

MULTIPLE Tells the linker that more than one code image of the loadable module may be loaded into file server memory concurrently.

COPYRIGHT Tells the linker to include a copyright string in the output file. An ASCII string 1 to 252 bytes long, in double quotes following the keyword copyright, is displayed whenever the module is loaded. To start a new line within the displayed string, use “\n”. If the copyright keyword is used but no string is entered, the linker includes the Novell default copyright message.

To use the copyright keyword, you must use the NLMLINKP.EXE.

VERSION Gives the linker the version of the module that should be placed into the NLM header version field. The format for this keyword is:

VERSION Major, Minor [, Revision]

The version must be separated by commas. The Major version number is one digit and the Minor version number is two digits. The Revision number is optional and is a number from 1-26 representing a-z. For example, “VERSION 3 , 50 , 2” produces the version field 3.50b in the NLM header of the output file.

To use the version keyword, you must use the NLMLINKP.EXE. The date is automatically set by the linker to the date that the files are linked.

Loading the Driver

On the Netware Server, the Driver.LAN file is loaded into the server's memory using the LOAD command. The driver can be loaded from a floppy, a directory on a DOS partition of the server's hard disk, or the SYS:SYSTEM directory of the NetWare partition. The NetWare Loader resolves the driver's import list and links the driver to the OS. Once loaded, the driver functions as if it had been hard coded into the NetWare operating system.

The MSM.NLM and <TSM>.NLM must be loaded (only once) before any HSMs are loaded. The required NLMs may all be auto-loaded using the "module" keyword to load the <TSM>.NLM in the linker definition file.

To load the driver, you could enter a command similar to this:

```
LOAD <driver>  FRAME=ETHERNET_802.3, PORT=300,
                NODE=2608C760361, INT=3
```

The parameters do not have a set order. The commas are optional.

Building the HSM **A-5**

Driver Configuration File

HSMs that support a large number of custom keywords may have trouble specifying all parameters on the command line. Command line parameters can now be listed in a driver configuration file or load “response” file.

To use a load response file, type the parameters as they would appear on the command line into the file. The parameters can be separated by either a space or a carriage-return/linefeed. The filename you choose must have a .cfg extension. Then at the command-line type:

```
LOAD <drivename>@<response filename>
```

If this file exists in the same directory as the driver, the MSM will open the file, parse it, and process it along with other parameters on the command-line.

Load Keywords and Parameters

This section describes the parameters for the NetWare LOAD command. The *MSM ParseDriverParameters* routine handles the load command parameters in drivers written using the MSM. The load parameters and examples of their use are described below.

PORT This is the I/O mapped address base that the user wants the board to use. A port length can also be included as shown in the following examples.

```
LOAD <driver> PORT=300
LOAD <driver> PORT=300:A
LOAD <driver> PORT=300:A PORT1=700:8
```

MEM This is the beginning address of the shared RAM that the board can use. The size of the shared memory buffer can also be specified.

```
LOAD <driver> MEM=C0000
LOAD <driver> MEM=C0000:1000
LOAD <driver> MEM=C0000:1000 MEM1=CC000
```

INT This is the interrupt number that the board is expected to use to awaken the ISR routine.

```
LOAD <driver> INT=3
LOAD <driver> INT=3 INT1=5
```

DMA If the board supports DMA, this is the Direct Memory Address channel that the NIC should use for data transfer to memory.

LOAD <driver> DMA=0
 LOAD <driver> DMA=0 DMA1=3

SLOT Refer to the MLIDSlot field description in Table 3.2, "Configuration Table Field Descriptions" on page 3-22 of this document.

RETRIES This is the number of send retries that the MLID should use in its attempts to send packets.

RETRIES = n or RETR = n

CHANNEL This is the channel number (controller number) to use for multichannel adapters. A multichannel adapter is a board containing more than one network interface controller.

CHANNEL = number

BELOW16 This keyword must be specified on the load command-line if the driver needs memory allocated below the 16 Megabyte boundary. This keyword is required only if the driver is loaded on a system that initially has less than 16 megabytes of memory but will have more memory added later using the server's REGISTER MEMORY command. In addition, the driver must also set the *DriverNeedsBelow16Meg* field of the *DriverParameterBlock* to a non-zero value.

BELOW16

BUFFERS16 This keyword is used to override the number of RCB's below 16 Megabytes allocated by the MSM at initialization. The HSM must set *DriverNeedsBelow16Meg* in the *DriverParameterBlock* for this keyword to be valid. The RCB allocation routines (*MSMAllocateRCB*, *<TSM>GetRCB*, *<TSM>ProcessGetRCB*, etc.) use these RCB's if the RCB allocated from the LSL is physically over 16 Megabytes. The number of RCB's allocated by default is 8. If the HSM preallocates more than 8 RCB's at a time, the user can override this default when loading the driver by typing the keyword:

BUFFERS16=n

The MSM will force this value to a multiple of 8, so values other than 8, 16, 32... are invalid. No restriction is placed on the maximum value, except that the MSM may not be able to allocate enough memory from the OS. To increase the size of the OS memory pool of buffers below 16 Megabytes, insert the following set command in the STARTUP.NCF file:

"set reserved buffers below 16 Meg = xxx"

Where xxx is a multiple of 8, between 8 and 200 (Default is 16).

Building the HSM A-7

FRAME This is a string specifying the frame type (see the *Frame Types and Protocol IDs* supplement for a list of frame type strings).

FRAME = type

LSB/MSB ADDRESS FORMATS Token-Ring drivers may add “MSB” or “LSB” following the frame type designation. LSB forces canonical addresses to be passed between the MLID and the upper layers. The MSB designation forces non-canonical addresses to be passed (this is the default for Token-Ring media). Ethernet may not use the MSB designator.

NODE This is the node address that the board is to use; this address should override the default address on the board if any.

NODE = nnnnnnnnnnn

In the case of Token-Ring media, which has a non-canonical physical layer format, the override node address on the command-line may be entered in either canonical or non-canonical format (see the *Canonical and Non-Canonical Addressing* supplement). To indicate the format of the address, an “L” (LSB) or an “M” (MSB) may be appended. For example, to indicate a node address for Token-Ring media in canonical format enter:

NODE = nnnnnnnnnnnL

No matter what the format of the node address specified on the command-line, the format of the node address actually placed in the configuration table is indicated by bit 14 in the *MLIDModeFlags* byte.

Appendix **B** *The NetWare Debugger*

Introduction

The NetWare operating system includes an internal assembly- language-oriented debug utility. The NetWare debugger provides the commands summarized in the following table. These commands and examples of their use are explained in the remainder of this appendix.

NetWare Debugger Commands

.A	displays the abend or break reason
B	displays all current breakpoints
BC number	clears the specified breakpoint
BCA	clears all breakpoints
B = addr [condition]	sets an execution breakpoint at address
BW = addr [condition]	sets a write breakpoint at address
BR = addr [condition]	sets a read/write breakpoint at address
C addr	changes memory in interactive mode
C addr=number(s)	changes memory to the specified number(s)
C addr="text"	changes memory to the specified text ASCII values
.C	does a diagnostic memory dump to diskette
D addr [length]	dumps memory for optional length
DL[+linkoffset] addr [length]	dumps memory starting at address for optional length and traverses a linked list
	(default link field offset is 0)
REG=value	changes the specified register to the new value, where REG is EAX, EBX, ECX, EDX,
ECX, EDX,	ESI, EDI, ESP, EBP, EIP, or EFL
F Flag=value	changes the FLAG bit to value (0 or 1) where FLAG is CF,AF, ZF, SF, IF,
TF, PF, DF or	OF
G [break addr(s)]	begins execution at current EIP and set optional temporary breakpoints(s)
H	displays basic debugger command help screen
HB	displays breakpoint help screen
HE	displays expression help screen
.H	displays the dot help screen
I [B;W;D] Port	inputs byte, word, or dword from Port (default is byte)

The NetWare Debugger **B-1**

M addr [L length] pattern rest of	searches memory for pattern (L length is optional and if not specified, the memory will be searched)
.M	displays loaded module names and addresses
N symbolName addr	defines a new symbol name at address
N -symbolName	removes defined symbol name
N--	removes all defined symbols
O [B;W;D] Port=value	outputs byte, word, or dword value to PORT
P	proceeds over the next instruction
.P	displays all process names and addresses
.P addr	displays <address> as a process control block
Q	quits and exits back to DOS
R	displays registers and flags
.R	displays the running process control block
S	single-steps
.S	displays all screen names and addresses
.S addr	displays <address> as a screen structure
T	trace (single-step)
U addr [count]	unassembles count instructions starting at address
V	views server screens
.V	displays server version
Z expression	evaluates the expression (See HE help screen)
? [addr] (default is	If symbolic information has been loaded, the closest symbols to address EIP) are displayed

Invoking the Debugger

There are four methods available to invoke the debugger.

From the server console keyboard

1. Press the <CTRL> - <ALT> - <LEFT-SHIFT> - <RIGHT-SHIFT> - <ESC> key combination simultaneously at the server console keyboard. This will not work if the server is hung in an infinite loop with interrupts disabled or if the server console is secured.
2. After the driver abends or GPIs the server, enter the key combination described in method 1 above or type 386debug. The characters do not echo to the screen, but the debugger prompt (#) appears.

From a driver or NLM

3. Include an INT 3 in the desired code segment where the break-point is to be executed. Programs written in C using CLIB can call the *Breakpoint* ()

B-2 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

function. Programs written in C using the OS library can call the *EnterDebugger* () function.

Manually

4. Generate a non-maskable interrupt with an NMI board. This will cause the server to Abend, after which method 2 above may be performed. This method may be required if the software being debugged is in an infinite loop with interrupts disabled.

When the debugger is entered, it will display the location at which the trap occurred, the cause of the trap into the debugger, and the contents of the general registers and flags.

Once you have entered the debugger, the address and length of the data and code segments of all loaded modules may be found using the **.m** command. Breakpoints can then be set in the driver code using addresses in the map file relative to the addresses dumped by the debugger.

The available debugger commands are explained on the following pages of this appendix.

Debug Commands

Help

The debugger's help commands are:

H	display help for general commands
HB	display help for breakpoints
HE	display help for expressions
.H	display help for "." commands

"." Commands

.a	display the Abend or break reason
.c	do a diagnostic memory dump to diskette
.h	display help for "." commands.
.m	display loaded module names and addresses.
.p	display all process names and addresses.
.p addr	display <i>address</i> as a process control block
.r	display running process control block.
.s	display all screen names and addresses.
.s addr	display all screen names and addresses.
.v	display server version

Breakpoints

There are four breakpoint registers, allowing a maximum of four breakpoints to be set at the same time. The breakpoints can be permanent breakpoints, set using the B commands (described in this section), or temporary breakpoints set using the G command. In addition, the P command will also set a temporary breakpoint if the current instruction cannot be single stepped. This section consists of descriptions and examples for setting permanent breakpoints. Temporary breakpoints using the G and P commands are described later in this chapter.

Breakpoint Conditions

Several breakpoint commands include an optional [condition] argument. A breakpoint condition is any expression to be evaluated when the break occurs. If the condition is false, execution is resumed immediately without entering the interactive debugger.

B-4 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

B

Display all breakpoints that are currently set.

B

Breakpoint 0 write byte at FFF65623
 Breakpoint 1 read or write byte at 000653BA
 Breakpoint 2 execute at FFF06BA3

BC number

Clear the breakpoint specified by number.

BC 2

Breakpoint cleared

BCA

Clear all breakpoints.

BCA

All breakpoints cleared

B = address [condition]

Set an execution breakpoint at the *address* specified when the indicated *[condition]* is true.

B = FFF8765A

Set as breakpoint 0

BW = address [condition]

Set a write breakpoint at the *address* specified when the indicated *[condition]* is true.

BW = FFF665AB

Set as breakpoint 1

BR = *address* [*condition*]

Set a read/write breakpoint at the *address* specified when the indicated [*condition*] is true.

BR = 0065ACF3

Set as breakpoint 2

Memory

This section describes how to change or display memory contents.

C *address*

Interactively change the contents of memory location *address*.

(To end interactive mode, type a period.)

C FFF6432A

FFF6432A (00)=33

FFF6432B (34)=C8

FFF6432C (5A)=.

C *address* = *number(s)*

Change the memory contents beginning at *address* to the specified *number(s)*.

C FFF534C5 = 00,00,12,5A,78

Change successfully completed

C *address* = “*text string*”

Change the memory contents beginning at *address* to the specified *text string*.

C FFF60DB3 = “This is a string.”

Change successfully completed

D address [count]

Dumps the contents of memory, starting at *address*, for *[count]* number of bytes. The address and count are hexadecimal numbers. If the count is not specified, one page (100h bytes) will be display. The D command can be repeated by pressing <ENTER> at the # prompt.

D FFF7765E

```

FFF7765E      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
.....
FFF7769E      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
.....
FFF776DE      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
.....

```

D FFF7765E 10

```

FFF7765E      00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
.....

```

The NetWare Debugger **B-7**

M address [L length] bytepattern

Search memory for a *byte pattern* match, starting at location address and continuing until [*L length*] is reached. If a match is found, 128bytes (beginning with the pattern) are displayed. The M command can be repeated by pressing <ENTER> at the # prompt.

M FFF77F00 54 48 45 52

```
FFF77F1C      54 48 45 52 4E 45 54 5F - 49 49 00 90 00 00 00 00
THERNET_II.....
FFF77F2C      00 00 00 00 00 00 90 6B - F7 FF 00 00 00 00 00 00
.....kw.....
FFF77F3C      48 61 72 64 77 61 72 65 - 44 72 69 76 65 72 4D 4C
HardwareDriverML
FFF77F4C      00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
FFF77F5C      00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
.....
FFF77F6C      00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
.....
FFF77F7C      00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
.....
FFF77F8C      00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
.....
```

M FFF77F5C L1F 54 48

Match not found

Register Manipulation

This section describes the debugger commands used on the microprocessor's general and flag registers.

R

Display the EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, EIP, and Flag Registers.

R

```
EAX=99999999  EBX=00005455  ECX=78787878
EDX=00060544
ESI=00000000  EDI=80868086  EBP=00000000
ESP=FFF67876
EIP=FFF56784  FLAGS=00010002
```

register = value

Change the specified *register* to the new *value*. The command is effective with EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, and EIP.

EAX=8099ACB3

Register changed

F flag = value

Change 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.

F PF=0

Flag changed

Input/Output

This section describes the debugger's I/O commands.

I[B,W,D] port

Input a byte, word, or double word from *port*.

I 25A

The NetWare Debugger **B-9**

Port (25A)=F8

IB 25A

Port (25A)=F8

IW 1B3

Port (1B3)=D3FF

O[B,W,D] port = value

Output a byte, word, or double word *value* to *port*.

O 25B=7D

Output completed

OW 18E=3C0F

Output completed

Miscellaneous

This section consists of descriptions of the remaining debugger commands.

G [address(es)]

Begin execution (Go) from current position and set temporary breakpoint
[address(es)]

G FFF56784

Break at FFF56784 because of go breakpoint

EAX=99999999 EBX=00005455 ECX=78787878

EDX=00060544

ESI=00000000 EDI=80868086 EBP=00000000 ESP=FFF67876

EIP=FFF56784 FLAGS=00010002

FFF56784 BB30CE0500 mov ebx, 0005CE30

N symbolname value

Define a new symbol with a *value*.

B-10 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

N thissym 0F0F

P

Proceed over next instruction. This command is similar to the “Trace” or “Single Step” command but it will not single step loops or calls. The P command can be repeated by pressing <ENTER> at the # prompt.

Q

Quit and return to DOS.

T or S

Trace or Single Step through the program. The T or S commands can be repeated by pressing <ENTER> at the # prompt.

U *address [count]*

Unassemble *count* instructions from *address*. The U command can be repeated by pressing <ENTER> at the # prompt.

u FFF87885 2

```
FFF87885 0000  add  [eax], al
FFF87887 0000  add  [eax], al
```

V

View the screens (will step through the screens sequentially).

Z *expression*

Evaluate the expression (similar to calculator).

z 7+8

Evaluates to: F

The NetWare Debugger **B-11**

Debug Expressions

All numbers in debug expressions are entered and shown in hex format. In addition to numbers, the following registers, flags, and operators can be used in expressions and breakpoint conditions:

Registers: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, EIP
Flags: FLCF, FLAF, FLZF, FLSF, FLIF, FLTF, FLPF, FLDP, FLOF

Operators and Precedence

Symbol	Description	Precedence
!	logical not	1
-	2's compliment	1
~	1's compliment	1
*	multiply	2
/	divide	2
%	mod	2
+	addition	3
-	subtraction	3
>>	bit shift right	4
<<	bit shift left	4
>	greater than	5
<	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

Grouping Operators

The operators `()`, `[]`, and `{ }` have a precedence of 0. These grouping operators can be nested in any combination.

`(expression)`

Causes the expression to be evaluated at a higher precedence.

`[size expression]`

Causes the expression to be evaluated at a higher precedence and then uses the value of the expression as a memory address. The bracketed expression is replaced with the byte, word, or double word at that address. “Size” is a data size specifier of the type B, W, or D.

`{ size expression }`

Causes the expression to be evaluated at a higher precedence and then uses the value of the expression as a port address. The bracketed expression is replaced with the byte, word, or double word input from the port. “Size” is a data size specifier of the type B, W, or D.

Conditional Evaluation

`expression1 ? expression2 , expression3`

If *expression1* is true, then the result is the value of *expression2*; otherwise, the result is the value of *expression3*.

Symbolic Information

Symbolic information may be included in a driver file that can be used to access routines or variables by name while in the NetWare 386 debugger. To access symbolic information, the following steps must be taken:

1. Declare public all desired symbols in the driver.
2. Include the keyword *debug* in the driver's linker definition file.

Each of these symbol names (the debugger is case-sensitive) can now be used in the same way the address they represent would be used. For example, at the debug prompt it is possible to display memory beginning at the address of the label

AdapterBdStruct by entering:

#d AdapterBdStruct

Symbols may be dynamically defined by the debugger. If it is necessary to dynamically define more than 10 symbols the server must be loaded with the -y option.



Debugging information **must be** removed before releasing the driver. Including the *debug* keyword in the definition file will cause a message to be displayed on the console when the driver is loaded, indicating that it contains debug information

Appendix **C** *NESL Support*

Overview

The NetWare Event Service Layer (NESL) handles event registration and notification. The NESL is designed around the concept of consumers and producers. Generally, a producer will produce events, which a consumer consumes. The NESL provides the following services:

- Registers the event producer
- Deregisters the event producer
- Performs the event notification
- Registers the event consumer
- Deregisters the event consumer

For a given event type, there can be multiple consumers and producers simultaneously. A client module must register as a producer of an event in order to produce that event. Likewise, a module must register as a consumer of an Event Type in order to consume the event.

If a consumer chooses to consume an event, it will notify the producer that the event is consumed, and event notification will end.

When a producer or consumer is removed from the system, it must deregister all producer/consumer events it has registered.



Tasks should be designed to run to completion. If consumer and producer routines are running asynchronous event types (for example, IPX packet interrupts), the routines must be resident. **MSMNESLProductEvent** will not protect the consumer routine from being reentered.

The NESL maintains a list for each event class. When a producer calls the NESL to signal that an event has occurred within a class, the NESL notifies

NESL Support **C-1**

everyone in the consumer list. The order used to call the consumers depends on the level of the OSI model the consumer belongs to and the calling direction defined by the event class.

The data definitions for the NESL are located in ODI_NESL.INC.

Registering and Deregistering Event Producers

Event producers use **MSMRegisterProducer** to register with the NESL as a producer of an event class. Once it registers, the event producer calls **MSMNESLProduceEvent** or **MSMNESLProduceMLIDEvent** to notify event consumers when an event takes place.

Note



Event producers can also register as event consumers.

When an event producer no longer provides events, it calls **MSMNESLDeRegisterProducer** for that event. For example, when an event providing module is unloading, its clean-up function must first call **MSMNESLDeRegisterProducer** for each event it has added. The module could then complete its unloading process.

Registering and Deregistering Event Consumers

Event consumers must register with the NESL in order to receive notification when an event occurs. These modules call **MSMRegisterConsumer** for each event class they wish to be notified of.

When an event consumer no longer requires event notification, or before it unloads, it must deregister by calling **MSMNESLDeRegisterConsumer** for each event it registered for.

NESL Structures

EPB (Event Parameter Block) Structure

```

EPB struc
    EPB_MajorVersion    dd    ?
    EPB_MinorVersion    dd    ?
    EPB_EventName       dd    ?
    EPB_EventType       dd    ?
    EPB_ModuleName      dd    ?
    EPB_DataPtr0        dd    ?
    EPB_DataPtr1        dd    ?
    EPB_EventScope      dd    ?
    EPB_Reserved        dd    ?
EPB ends

```

Field Descriptions:

EPB_MajorVersion

Major version of the Event Parameter Block. The current version is 1 (for 1.00).

EPB_MinorVersion

Minor version of the Event Parameter Block. The current version is 0 (for 1.00).

EPB_EventName

Event Name (class name) for the event as registered with NESL--for example, Service Suspend or Service Resume. All valid event names must be registered with Novell Labs.

EPB_EventType

Name for the Event Type. An example of an Event Type for Service Suspend is APM Suspend. All valid Event Type names must be registered with Novell Labs.

EPB_moduleName

Pointer to the module name that generated the event--for example, NE2000.

NESL Support **C-3**

EPB_DataPtr0

Used to pass a pointer to the configuration table.

EPB_DataPtr1

Used for event dependent information.

EPB_EventScope

The CHSM must set this field to EPB_SPECIFIC_EVENT.

EPB_Reserved

Reserved by Novell.

NESL_ECB Structure

The following defines the NESL_ECB structure.

```
NESL_ECB struc
    NESL_ECBNext
    NESL_ECBVersion
    NESL_ECBOSiLayer
    NESL_ECBEventName
    NESL_ECBRefData
    NESL_ECBNotifyProc
    NESL_ECBOwner
    NESL_ECBWorkspace
    NESL_ECBContext
NESL_ECB ends
```

Field descriptions:

NESL_ECBNext

Reserved. This field should not be modified by the calling routine while the NESL_ECB structure is registered.

NESL_ECBVersion

This field contains the version number of the NESL_ECB structure. This field allows the interface to be expanded in the future while still providing full backward compatibility. The current version is 2.

C-4 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

NESL_ECBOSiLayer

Determines the ordering of registered consumers of the same event. The format of this field is 0xLRRR, where L is the number (0-7) corresponding to the OSI layer and RRR (0-4095) is the relative order with other modules also registered on that layer. The relative ordering is useful when certain events require specific consumer ordering.

The definition NESL_HOOK_FIRST can also be used in element *NESL_ECBOSiLayer*. This definition causes a consumer to be hooked first, no matter what. If the caller sets the low byte of *NESL_ECBOSiLayer* to this value, the consumer will be hooked first in the consumer list. Normally, NESL events will put lower layer identifiers before the hooked lead element. If another call is made specifying this definition, an error will be returned to the caller and the element will not be added to the list.

NESL_ECBEventName

ASCIIZ name string of the event (class). This name has the maximum length of NESL_MAX_NAME_LENGTH.

NESL_ECBRefData

This field is used by producers only. Consumers do not use this field. Consumers must set this field to NULL when registering.

This is a flag field used to specify whether the event is unique or consumable. It also indicates the sorting order for calling registered consumers at event time.

Consumers that are on the orphan consumer list will be sorted when a new producer is registered. All consumers that are registered after a producer is registered will be correctly sorted.

NESL_ECBNotifyProc

Pointer to the event notification callback routine.

```
UINT32  MyNotifyProc (
        NESL_ECB  *ConsumerNecb,
        NESL_ECB  *ProducerNecb,
        Void      *eventData )
```

ConsumerNecb

Points to the NESL_ECB structure used by consumer during **MSMNESLRegisterConsumer**.

NESL Support **C-5**

ProducerNecb

Points to the NESL_ECB structure used by the producer during **MSMNESLRegisterProducer**.

EventData

If the producer only has one data item, it can be passed to the consumer as an argument or as an address.

If the producer has more than one data item or if the producer wishes to guarantee portability, the address of an array of data items should be passed. The structure of *eventData* must be defined by the producer and known by the consumer if it is to be interrupted properly.

Return from a consumer after an event notification callback:

NESL_EVENT_CONSUMED

Event was consumed by the consumer process.

NESL_EVENT_NOT_CONSUMED

Event was not consumed by the process.

Note, this is only really applicable if the event is consumable, but a consumer should always do this to be compatible with both types of events. Called from foreground time or from interrupt time with interrupts enabled or disabled.

NESL_ECBOwner

Specifies the owner of the NESL_ECB structure. This field is platform-specific and platform-dependent. The DOS/MS Windows implementation **requires** this field to be set to the owner's module handle information.

NESL_ECBWorkSpace

Reserved. This field should not be modified by the calling routine while the NESL_ECB structure is registered.

NESL_ECBContext

This field is available for use by the owner of the NESL_ECB structure. It will not be modified by anyone else in the system. It may be used by the owner to pass context or other data to the notification procedure. If the owner is not using this field, it must be set to NULL.

Events and Types

Event names and specific event types are identified with ASCIIZ strings. Novell has defined four event names along with some specific event types. However, anyone can define event names or event types by defining unique names (ASCIIZ strings) for them. The definition of an Event Name must also include the direction in which the consumers of the event Event Name will be called (that is, called from the top of the OSI model down or from the bottom up). Event types that are added to existing event names must fit within the definition of the event name.

Below is a list of the event names and event types defined by Novell.

Event Names

Event Name	Description
Suspend Notification	The Event Name contains any event that suspends a service. This event is called from the top of the OSI model down.
Resume Notification	This Event Name contains event types that indicate the availability of a new service or the restoration of a previously available service. This event is called from the bottom of the OSI model up.
Service/Status Change	This Event Name contains event types that signal a change in status or the current level of service. This event is called from the top of the OSI model down.
Suspend Request	This Event Name contains event types that request permission to suspend service before the service is actually suspended. This event is called from the top of the OSI model down.

Event Types

Service Suspend Types

Type Name	Description
MLID Cable Disconnect	This Event Type indicates that the cable has been disconnected from a given NIC. A pointer to the MLID's configuration table is passed in the EPBDataPtr0 field of the Event Parameter Block.
MLID Card Removal	This Event Type is triggered by the hardware and indicates that the PC Card has been removed from a socket. A pointer to the MLID's configuration table is passed in the EPBDataPtr0 field of the Event Parameter Block. Even though this Event Type puts the MLID into shutdown mode, it does not generate a shutdown event.
MLID Hardware Failure	This Event Type indicates that a serious hardware failure has occurred with the NIC. A pointer to the MLID's configuration table is passed in the EPBDataPtr0 field of the Event Parameter Block.
MLID Not In Range	This Wireless Event Type indicates that there is no access point in range. A pointer to the MLID's configuration table is passed in the EPBDataPtr0 field of the Event Parameter Block.
MLID Shutdown	This Event Type is triggered through the MLID control services and indicates that an MLID was shutdown. A pointer to the MLID's configuration table is passed in the EPBDataPtr0 field of the Event Parameter Block.

MLID Media Access Denied

This Event Type indicates that access to the physical medium was either denied or unsuccessful. A pointer to the MLID's configuration table is passed in the EPBDataPtr0 field of the Event Parameter Block.

Suspend Request

Currently no event types have been defined for this class.

Service Resumed Types

Type Name	Description
MLID Cable Reconnect	This Event Type indicates that the cable has been reconnected to a given NIC. A pointer to the MLID's configuration table is passed in the EPBDataPtr0 field of the Event Parameter Block.
MLID Card Insertion Complete	This Event Type is triggered when a new logical board is added to the system and LAN adapter and driver are fully functional. A pointer to the MLID's configuration table is passed in the EPBDataPtr0 field of the Event Parameter Block. This Event Type does not trigger a reset event.
MLID In Range	This wireless Event Type indicates that there is an access point in range again. A pointer to the MLID's configuration table is passed in the EPBDataPtr0 field of the Event Parameter Block.
MLID Reset	This Event Type is trigger by the MLID control services and indicates that an MLID was just reset. A pointer to the MLID's configuration table is passed in the EPBDataPtr0 field of the Event Parameter Block.

Service/Status Changed Types

Type Name	Description
MLID Access Point Change	This Event Type indicates that a station has moved from one access point's range to another and that the new access point will start serving the station. A pointer to the MLID's configuration table is passed in the EPBDataPtr0 field of the Event Parameter Block.
MLID Speed Change	This Event Type indicates that there has been a change in the communication speed. For example, in the wireless environment this could be caused by the radio link due to a change in the quality of the signal. A pointer to the MLID's configuration table is passed in the EPBDataPtr0 field of the Event Parameter Block.
MLID Config Table Change	This Event Type indicates that the MLID configuration tables have been updated by MSMUpdateConfigTables . A pointer to the MLID's updated configuration table is passed in the <i>EPBDataPtr0</i> field of the Event Parameter Block. The MSM produces this event inside MSMUpdateConfigTables .

MLID DeRegister Resource Change	This Event Type indicates that a resource registered using MSMRegisterResource has been deregistered using MSMDeRegisterResource . A pointer to the MLID's configuration table is passed in the <i>EPBDataPtr0</i> field of the Event Parameter Block. The MSM produces this event inside MSMDeRegisterResource .
MLID ReRegister Hardware Options	This Event Type indicates that hardware resource(s) have been reregistered using MSMReRegisterHardwareOptions . A pointer to the MLID's configuration table is passed in the <i>EPBDataPtr0</i> field of the Event Parameter Block. The MSM produces this event inside MSMReRegisterHardwareOptions .

NESL Return Codes

The NESL return codes (located in NESL.H) are as follows:

NESL_OK	00000000h
NESL_EVENT_CONSUMED	00000000h
NESL_EVENT_NOT_CONSUMED	00000001h
NESL_EVENT_BROADCAST	00000002h
NESL_EVENT_NOT_REGISTERED	00000003h
NESL_EVENT_TABLE_FULL	00000004h
NESL_EVENT_IS_CONSUMABLE	00000005h
NESL_EVENT_IS_NOT_CONSUMABLE	00000006h
NESL_NO_MORE_EVENTS	00000007h
NESL_PRODUCER_NOT_FOUND	00000008h
NESL_CONSUMER_NOT_FOUND	00000009h
NESL_INVALID_CONTEXT_HANDLE	0000000ah
NESL_INVALID_DESTINATION	0000000bh
NESL_REGISTERED_UNIQUE	0000000ch
NESL_REGISTERED_NOT_UNIQUE	0000000dh
NESL_REGISTERED_CONSUMABLE	0000000eh
NESL_REGISTERED_BROADCAST	0000000fh
NESL_REGISTERED_SORT_TOP_DOWN	00000010h
NESL_REGISTERED_SORT_BOTTOM_UP	00000011h
NESL_DUPLICATE_NECB	00000012h
NESL_INVALID_NOTIFY_PROC	00000013h
NESL_INVALID_FIRST_ALREADY_HOOKED	00000014h

NESL Event Flags

The following are the NESL event flags:

NESL_BROADCAST_EVENT	00000000h
NESL_SORT_CONSUMER_TOP_DOWN	00000000h
NESL_SORT_CONSUMER_BOTTOM_UP	00000001h
NESL_CONSUME_EVENT	00000002h
NESL_UNIQUE_PRODUCER	00000004h
NESL_NOT_UNIQUE_PRODUCER	00000000h

NESL OSI Layer Definitions

The following are the NESL OSI layer definitions:

NESL_APPLICATION_LAYER	7000h
NESL_PRESENTATION_LAYER	6000h
NESL_SESSION_LAYER	5000h
NESL_TRANSPORT_LAYER	4000h
NESL_NETWORK_LAYER	3000h
NESL_DATAINK_LAYER	2000h
NESL_PHYSICAL_LAYER	1000h

C-16 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Revision History



Note

This Revision History covers all document changes from Doc Version 1.00 to Doc Version 1.10 and from Doc Version 1.10 to Doc Version 1.11.

Page numbers for items 1 through 13 refer to Doc Version 1.00.

Page numbers for items 14 through 17 refer to Doc Version 1.10.

Page numbers for items 18 through 21 refer to Doc Version 1.11.

1. In the following three functions,

HSMPrintString (Page 7-91),
MSMPrintStringFatal (Page 7-93),
MSMPrintStringWarning (Page 7-95),

the NULL terminated message pointed to by the ESI register, which includes possible argument #1 and possible argument #2, cannot exceed 128 bytes.

2. In the **Driver Parameter Block** definition on page 3-5, the *DriverISR2Ptr* field occurs twice. The second occurrence of the *DriverISR2Ptr* field (which is on the third line from the bottom of the page) is a duplicate and should be ignored.
3. In the **Driver Configuration Table** definition on page 3-15, the *MLIDCFG_MinorVersion* is defined as 13. It should be 14.

4. On page 5-35, Driver Priority Queue Support, under "Processor States: Entry State" the ESI line should read:

ESI

Pointer to a transmit ECB.

The entire ECX line should be deleted.

5. On page 3-25, in the **Configuration Table Field Descriptions**, change the Description for **MLIDBusTag** to read:

Pointer to an architecture-dependent value, which specifies the bus on which the adapter is found. Set this field before calling **ParseDriverParameters**. The value placed in this field is returned by **SearchAdapter** unless the board is Legacy ISA, in which case it is set to zero.

6. On page 1-7, "Loading Driver Modules", replace the first paragraph with the following paragraph:

The ODI Toolkit components for the specific platform being used must be loaded before the HSM is loaded. The HSM linker definition file must list a dependency on the appropriate TSM, using the module keyword, for the required NLMs to load automatically.

7. In Appendix A, on page A-2, under "Linker Definition file", delete "MSM" from the Data column for the MODULE Keyword.
8. In Appendix A, on page A-5, under "Loading the Driver", change the first sentence of the first paragraph to read:

On the Netware Server, the Driver.LAN file is loaded... etc.

Also, change the last sentence of the second paragraph to read:

The required NLMs may all be auto-loaded using the "module" keyword to load the <TSM>.NLM in the linker definition file.

9. In Appendix A, on page A-7, change the definition for SLOT to read:

Refer to the *MLIDSlot* field description in Table 3.2, "Configuration Table Field Descriptions" on page 3-22 of this document.

10. On page 3-25, under **MLIDBusTag**, **MSMParseDriverParameters** should be replaced with **MSMRegisterHardwareOptions**.

11. On page 5-20, under **Receive Error**, add the following text as the last bullet:

Pass the appropriate receive error bits to **<TSM>GetRCB**, **<TSM>ProcessGetRCB**, or **<TSM>FastProcessGetRCB**.

12. On page 5-51 after the line that reads "Bit 31 - Driver shutting down (set by TSM)", add the following text:

An ECB aware HSM must set all of these bits as necessary before calling **<TSM>RcvCompleteStatus** or **<TSM>FastRcvCompleteStatus**. An RCB aware HSM need set only Bit 0 - CRC error and Bit 1 - CRC/Alignment error as necessary, the others are set by the TSM if needed.

13. On page 6-15, **<TSM>ProcessGetRCB**, and page 6-18, **<TSM>FastProcessGetRCB**, add the following note to the Remarks section of both routines:

For some busMaster implementations, you must set **RProtocolWorkspace** (defined in ODI.INC) to the number of bytes necessary to skip to the beginning of the packet. This value can be as high as 128 bytes for chips which have poor alignment capabilities. This field is normally part of the reserved space in the RCB definition and can only be used with this call for the purpose stated above.

Note



The page numbers for items 14 through 17 refer to Doc Version 1.10.

14. On page 3-28, in **Table 3-4, "MLIDSFlags Bit Map Fields"**, in the description for bits 10, 9; add the following **Note**.

Bit 9 is not used by ECB aware HSMs; ECB aware HSMs must do their own filtering of multicast addresses.

15. On page 5-48, under **Adapter Multicast Filtering**, add the following **Note**:

ECB aware HSMs must do their own filtering of multicast addresses.

16. On page C-5, under *NESL_ECBRefData*, add the following text:

This field is used by producers only. Consumers do not use this field. Consumers must set this field to NULL when registering.

This is a flag field used to specify whether the event is unique or consumable. It also indicates the sorting order for calling registered consumers at event time.

Consumers that are on the orphan consumer list will be sorted when a new producer is registered. All consumers that are registered after a producer is registered will be correctly sorted.

17. On page C-11, under **Service Status Changed Types**, add the following new Type Names:

MLID Config Table Change
MLID DeRegister Resource Change
MLID ReRegister Hardware Options

18. On page 2-4, in the **Note** under **Multi-Operating System Provisions**, the following reference:

See Appendix C for a list of issues and problem areas to check when writing or updating a driver.

has been changed to:

Refer to Appendix A, "Building the HSM" when writing or updating a driver.

19. On page 6-6, in the **TSMConfigTable** structure, add:

TSMCFG_SystemFlags dw 0

after TSMCFG_MaxFrameSize.

20. On page 6-2, <TSM>**BuildTransmitControlBlock**, under **Return State**, change the **Preserved** registers from:

EBP, ECX

to

EBP, EDI

21. On page 7-127, **MSMScheduleTimer**, under *TimerType*, change all:

TIMER_TYPE_

to

AES_TYPE_

22. On page C-4, in the NESL_ECB Structure, add:

NESL_ECBContext

as the last item in the structure.

23. On page C-4, under *NESL_ECBVersion*, change the last sentence in the paragraph to read:

The current version is 2.

instead of 1.

24. On page C-6, add the following text:

NESL_ECBContext

This field is available for use by the owner of the NESL_ECB structure. It will not be modified by anyone else in the system. It may be used by the owner to pass context or other data to the notification procedure. If the owner is not using this field, it must be set to NULL.

6 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

Index

A

- Adapter
 - configuration information
 - MSMRdConfigSpace16 7-98
 - location on bus 7-56
 - options 7-86
- adapter
 - shut down bit 3-31
- Adapter data space 3-33
- Alignment
 - bus platform 7-23

B

- Breakpoints for debugging 4
- breakpoints, clearing 5
- BUFFERS16, keyword A-7
- Bus
 - configuration information 7-33
 - getting information 7-25
 - locating adapter on 7-56
 - scanning for available 7-120
- bus
 - multiple platforms 7-3
 - writing configuration information 7-144
- Bus address
 - size 7-25
- Bus architecture 7-3
- Bus master
 - getting RCB 6-12
 - TSM compatibility 6-10
- Bus platform alignment 7-23

- Bus slot
 - MLIDSlot field 3-22
- bus specific information, getting 7-27
- bus support 7-2
- Bus type, getting with MSMGetBusType 7-31
- Bus-specific information from
 - MSMGetUniqueIdentifierParameters 7-58
- BusTag, MLID 3-25

C

- Call back
 - MSMScheduleAESCAllBack 7-122
 - MSMScheduleIntTimeCallBack 7-124
- canonical address bit 3-26
- Code and data space 2-11
- Configuration file, driver A-6
- Configuration information
 - driver
 - MSMRdConfigSpace8 7-96
 - writng 7-144
- Configuration information for bus 7-33
 - MSMGetBusSpecificInfo 7-27
- Configuration table, driver
 - example template 3-15
 - field descriptions 3-17
- Configuration table,driver 3-13
- configuration, I/O
 - MLIDIOfgMajorVersion 3-25
- Control procedures
 - DriverManagement (optional) 5-54
 - DriverMulticastChange 5-47
 - DriverPromiscuousChange 5-50
 - DriverReset 5-43

- DriverRxLookAheadChange (optional) 5-53
- DriverShutdown 5-45
- DriverStatisticsChange (optional) 5-52
- CounterMask bit maps 3-40
- CSL compliant bit 3-27
- Custom keywords, examples 3-52

D

- Data space and statistics table template 3-35
- Data transfer mode 2-6
- Debugger
 - commands 1
 - conditional evaluation 13
 - expressions 12
 - grouping operators 13
 - invoking 2
 - miscellaneous commands 10
 - symbolic information 14
- Definition file, example A-2
- delaying tasks 7-150
- Design issues, hardware 2-6
- Development process 1-9, A-1
- DMA channel
 - indicator bit 3-30
- Downloading firmware 3-49
- Driver configuration file 1-10
- Driver initialization
 - Determine hardware options 5-4
 - DriverInit pseudocode 5-10
 - Initialize adapter 5-7
 - Register driverISR 5-8
 - Register hardware options 5-7
 - Register with LSL 5-8
 - Register with TSM/MSM 5-3
 - Timeout callbacks 5-9
- Driver parameter block 3-4
 - field descriptions 3-6
- DRIVER.INC file 1-10
- DriverISR
 - Pseudocode 5-24
 - Receive 5-19
 - Shared interrupts 5-21

- Transmit 5-20
- DriverNeedsBitMask 7-88
- DriverPoll 5-26
- DriverRemove 5-59
- DriverSupportsPhysFragments bit 3-26

E

- ECB aware
 - getting RCB 6-14
- ECB, sending
 - GetNextSend 6-2
- EISA 7-31, 7-34, 7-59, 7-120, 7-129
- EPB structure C-3
- Error messages
 - MSMAAlertFatal 7-5
 - MSMAAlertWarning 7-7
- Event Control Blocks (ECB) 4-24
- Event control blocks (ECB)
 - Receive ECBs vs RCBs 4-26
 - Transmit ECBs vs TCBs 4-27

F

- Firmware variables, example definitions 3-49
- fragmented RCB
 - support bit 3-26
- Frame data space 3-13
- frame types
 - packet size for different 3-32
- Frame types required 2-12

G

- Global Data Access 3-1
- global variables, MSM 4-1

H

- Hardware instance number (HIN) 7-45

- Hardware instance number mapping
 - MSMGetInstanceNumberMapping 7-47
- Hardware specific module 1-6
- Hardware specific Module (HSM)
 - Packet reception 5-13
- Hardware specific module (HSM) 5-1
 - Board service (DriverISR) 5-19
 - Control procedures 5-41
 - Driver initialization 5-3
 - Multi-operating system support 5-36
 - Packet transmission 5-27
- hub management
 - support bit 3-29

I

- I/O commands, debugger 9
- I/O configuration
 - version number 3-25
- I/O control procedures 2-4
- I/O port information
 - MLIDIOPortsAndLengths field 3-22
- I/O port sharing
 - support bit 3-31
- Include files 1-9
- Initialization 2-3
- Installation information file 1-10
- Interrupt service routine, when to use 2-8
- interrupt sharing
 - support bit 3-30
- Interrupts
 - setting 7-135
- ISA 7-31, 7-120, 7-129

K

- Keywords 3-51
 - custom, parsing 7-79
 - enhancements 3-53
 - structure 3-54
 - table 3-55

L

- Link support layer 1-2
- Linker definition file 1-10, A-2
- Linker description string A-3
- Load keywords and parameters A-6
- Loading the driver A-5
- Lying send 6-29

M

- MCA 7-31, 7-120, 7-129
- Media specific counters 3-41
- Media support module 1-4
- Memory
 - returning, MSMInitFree 7-64
- memory
 - page size 3-23
 - pages & paragraphs 3-23
- Memory address
 - MLIDMemoryDecode0 field 3-23
 - MLIDMemoryDecode1 field 3-23
- Memory allocation
 - MSMAlloc 7-9
 - MSMAllocPages 7-12
 - MSMInitAlloc 7-58
- Memory contents, displaying 6
- memory decoding
 - MLIDMemoryLength0 3-23
- Memory pages
 - MLIDMemoryLength0 field 3-23
- memory paging
 - MLIDMemoryLength0 3-30
 - support bit 3-30
- Memory, returning
 - MSMFree 7-21
 - MSMFreePages 7-22
- Message printing
 - MSMPrintString 7-91
 - MSMPrintStringFatal 7-93
 - MSMPrintStringWarning 7-95
- Micro Channel 7-35, 7-59

- MLIDCFG_SGCount field 3-20
- MLIDFlags bit map 3-28
- MLIDFlags field 3-21
- MLIDIOPort0 field 3-22
- MLIDIOPort1 field 3-22
- MLIDIOPortsAndLengths field 3-22
- MLIDIORange0 field 3-22
- MLIDIORange1 field 3-23
- MLIDLink field 3-22
- MLIDMemoryDecode0 field 3-23
- MLIDMemoryDecode1 field 3-23
- MLIDMemoryLength0 field 3-23
- MLIDMinorVersion field 3-21
- MLIDReserved1 field 3-21
- MLIDSharingFlags bit map 3-30
- MLIDSharingFlags field 3-22
- MLIDSlot field 3-22
- MSM equates
 - MSMMaxFrameHeaderSize 4-6
 - MSMPhysNodeAddress 4-8
 - MSMStatusFlags 4-3
 - MSMTxFreeCount 4-4
 - MSMVirtualBoardLink 4-2
- MSM/TSM data structures of RCB,TCB and ECB 4-9
- MSMAAlertFatal 7-5
- MSMAAlertWarning 7-7
- MSMBitSwapTable 4-1
- MSMRdConfigSpace16 7-98
- MSMRdConfigSpace32 7-100
- MSMRdConfigSpace8 7-96
- MSMScheduleTimer 7-126
- MSMWrtConfigSpace8 7-144
- multicast address
 - support bit 3-27
- multicast filtering bit 3-28
- Multicast, updating registers 6-32
- Multichannel adapters 2-13
- Multi-operating system provisions 2-4
- Multi-operating system support
 - DriverDisableInterrupt 5-39, 5-40
 - DriverEnableInterrupt 5-38

N

- NESL_ECB structure C-4
- Netware Bus Interface (NBI) 7-2
- NetWare loadable module 1-3
- Node address, load keyword A-8
- Notification information 7-117

O

- Object files, linking A-2
- ODI supplements xvii, 1-9
- ODI, see open data link interface 1-1
- Optional Support 2-14
- Options
 - driver
 - DriverNeedsBitMask 7-87
 - Needs Options 7-87
 - hardware
 - MSMRegisterHardwareOptions 7-104
- OS calls to the driver 2-9

P

- Packet reception
 - DMA and bus master 5-15
 - Programmed I/O and Shared RAM 5-13
 - RX-Net 5-17
- Packet size, maximum 3-32
- Packet transmission
 - Bus Master 5-28
 - Driver send 5-33
 - I/O, RAM, and DMA 5-28
- Parameter command-line examples 7-89
- Parameters
 - parsing, MSMParseDriverParameters 7-85
- Parameters, load A-6
- Parsing
 - driver parameters 7-85
- PCI 7-2, 7-31, 7-35, 7-59, 7-96, 7-98, 7-100, 7-120, 7-129

10 ODI Specification: Hardware Specific Modules (HSMs) (32-Bit Assembly Language)

PCMCIA 7-2, 7-31, 7-36, 7-120, 7-129

Physical address

getting 7-51

physical addresses

fragment pointers 3-26

Physical memory

reading 7-102

writing 7-142

Pipelined adapter

receiving RCB 6-19, 6-23

Plug and Play 7-2, 7-35, 7-59

Polling

support level, setting 7-54

Polling enable 7-20

Procedures, execution time 2-9

Procedures, HSM

required 2-1

Process of developing drivers 1-9

Processor speed rating 7-55

promiscuous mode

support bit 3-26

Protected mode, 32-bit 2-8

Public variables 3-1

R

raw send support bit 3-27

RCB

Allocation 7-13

returning 7-119

RCB, getting

FastProcessGetRCB 6-17

GetRCB 6-11

ProcessGetRCB 6-14

RXNetTSMGetRCB 6-34

RCB, receiving

FastRcvComplete 6-23

FastRcvCompleteStatus 6-25

RcvComplete 6-19

RcvCompleteStatus 6-21

RXNetTSMFastRcvEvent 6-40

RXNetTSMRcvEvent 6-38

Receive control blocks (RCB) 4-9

Fragmented RCB 4-12

Non-fragmented RCB 4-14

Reception, packet 2-3

Recommended support 2-13

Reentrancy 2-12

Register manipulation 9

Registering hardware options 7-104

Registering HSM, TSM 6-27

Registering the MLID 7-106

Removal, driver 2-3

MSMDriverRemove 7-19

Reserved fields

MLIDReserved1 field 3-21

Resources

returning 7-114

S

scatter/gather count

MLIDCFG_SGCount field 3-20

Servicing events

MSMServiceEvents 7-131

MSMServiceEventsAndRet 7-133

SMP 3-26

Source files, creating A-1

Source routing 2-14

specification version string 3-4

Statistics table, driver 3-34

structure

EPB C-3

NESL_ECB C-4

Supplements for developing ODI drivers xvii, 1-9

Support modules provided by Novell 1-4

symmetrical multiprocessing, bit 3-26

T

TCB, releasing

FastSendComplete 6-31

SendComplete 6-29

TCB, sending

GetNextSend 6-8

- Time, getting
 - MSMGetCurrentTime 7-42
 - MSMGetMicroTimer 7-49
- Timeout detection 2-5
 - DriverAES/DriverCallBack 5-56
 - DriverTxTimeout (RX-Net) 5-55
- Topology specific module 1-4
- Topology Specific Module(TSM) 6-1
- Topology specific module(TSM)
 - getting version 6-10
- Transmission, packet 2-3
- Transmit control blocks (TCB) 4-16
 - Ethernet, Token-Ring, and FDDI 4-17
 - RX-Net 4-19
- Transmit control blocks(TCB)
 - Fragment structure 4-22

U

- Unique identifier for an adapter 7-56