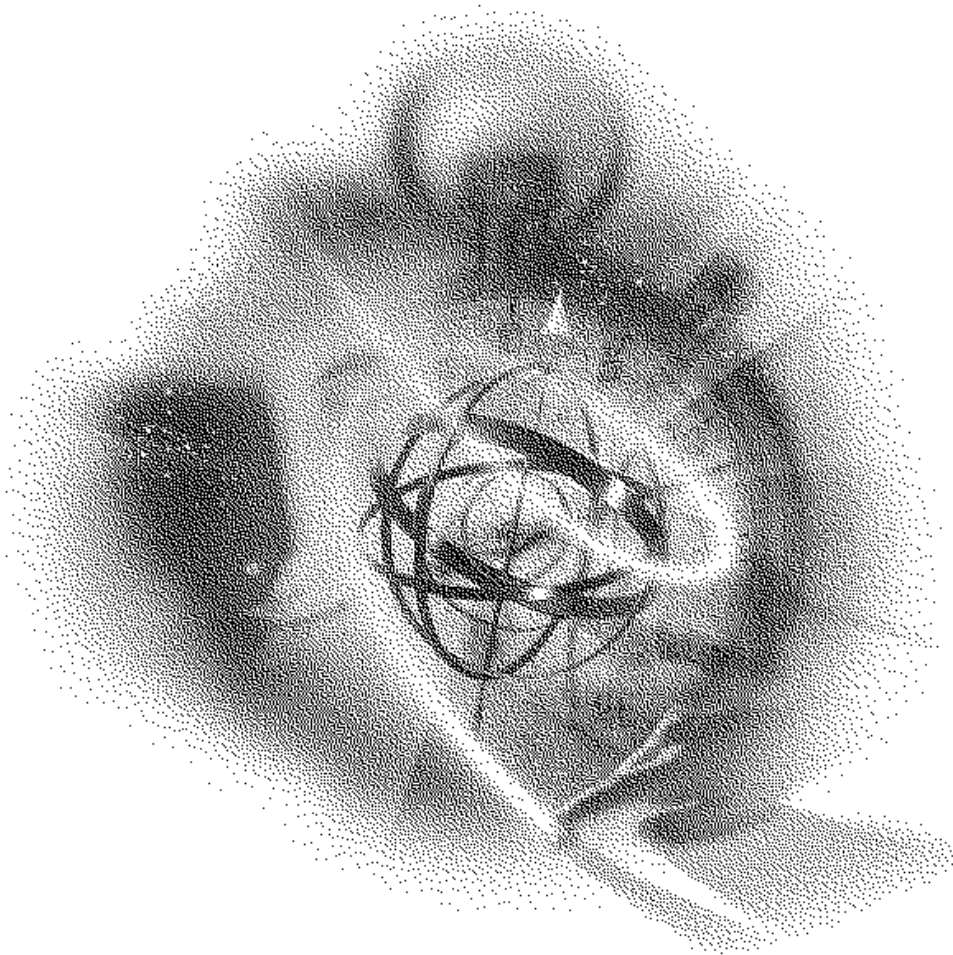


SPEC VERSION 1.11

Hardware Specific Modules (HSMs)
(C Language)



ODI Specification

Novell®

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 © 1993-1997 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) (C Language)
January 29, 1998**

Contents

Preface

| | |
|--|-------|
| Overview | xv |
| Prerequisites to Using this Manual | xvii |
| Manual Conventions | xvii |
| Data Type Definitions | xvii |
| Structure Definitions. | xviii |
| DRIVER_DATA Structure | xviii |
| MODULE_HANDLE Structure. | xx |
| GROUP_ADDR_LIST_NODE_ Structure. | xx |
| NODE_ADDR Structure. | xx |
| PROT_ID Structure | xx |
| CHSM_STACK Structure | xxi |
| Enumeration Definitions. | xxi |
| AES_TYPE Enumeration | xxi |
| BOOLEAN Enumeration. | xxii |
| ODI_NBI Enumeration. | xxii |
| ODISTAT Enumeration | xxiii |
| ODI_STAT Enumeration | xxiii |
| REG_TYPE Enumeration | xxiv |
| OPERATION_SCOPE Enumeration | xxiv |
| Portability Requirements | xxv |
| Referenced Documents | xxvi |

1 Introduction to ODI

| | |
|---|-----|
| Overview | 1-1 |
| Open Data-Link Interface (ODI). | 1-1 |
| Protocol Stacks | 1-2 |
| The Multiple Protocol Interface (MPI) | 1-4 |
| Link Support Layer (LSL) | 1-4 |
| Multiple Link Interface Drivers (MLIDs) | 1-6 |
| MLID Functionality. | 1-6 |
| The Multiple Link Interface (MLI) | 1-6 |
| LAN Driver Toolkit | 1-7 |

| | |
|--|------|
| C Language Media Support Module (CMSM) | 1-8 |
| C Language Topology Specific Module (CTSM) | 1-8 |
| C Language Hardware Specific Module (CHSM). | 1-9 |
| NetWare Bus Interface (NBI). | 1-9 |
| Data Flow | 1-10 |
| Send Data Flow | 1-10 |
| Receive Data Flow | 1-12 |

2 ODI C Language HSM Overview

| | |
|--|------|
| Overview | 2-1 |
| CHSM Procedures | 2-1 |
| Initialization | 2-2 |
| Board Service Routine | 2-2 |
| Packet Transmission | 2-3 |
| Control Procedures. | 2-3 |
| Timeout Detection | 2-3 |
| Driver Removal. | 2-3 |
| CHSM Data Structures and Variables | 2-4 |
| CHSM Design Considerations | 2-4 |
| Topology Issues | 2-4 |
| Hardware Issues | 2-4 |
| Network Interface Controllers. | 2-4 |
| Data Transfer Mode | 2-5 |
| Bus Type | 2-5 |
| NetWare Environment Issues | 2-6 |
| Interrupt Service Routine | 2-6 |
| Execution Times | 2-7 |
| Process Time | 2-7 |
| Privileged Time | 2-7 |
| Code and Data Space | 2-8 |
| Frame Data Space | 2-9 |
| Adapter Data Space | 2-9 |
| Adapter Code Space | 2-9 |
| Special Support | 2-11 |
| Reentrancy | 2-11 |
| Multicast Addressing | 2-11 |
| Promiscuous Mode | 2-11 |
| Optional Support | 2-11 |
| Hub Management. | 2-11 |
| Source Routing | 2-12 |
| Brouter (Source Route Bridging) | 2-12 |
| NESL Support | 2-12 |

3 CHSM Data Structures and Variables

| | |
|---|------|
| Overview | 3-1 |
| Driver Parameter Block | 3-1 |
| Driver Parameter Block Structure | 3-2 |
| Frame Data Space | 3-10 |
| Configuration Table | 3-10 |
| Driver Configuration Table Template | 3-11 |
| MLIDCFG_ModeFlags Field. | 3-22 |
| MLIDCFG_Flags Field. | 3-26 |
| MLIDCFG_SharingFlags Field | 3-29 |
| Maximum Packet Size. | 3-30 |
| Driver Adapter Data Space | 3-33 |
| Specification Version String | 3-34 |
| Driver Statistics Table | 3-35 |
| STAT_TABLE_ENTRY Structure | 3-35 |
| Field Descriptions | 3-35 |
| Statistics Table Structure | 3-36 |
| Example | 3-37 |
| MLID Statistics Table Media Specific Counters | 3-41 |
| Token-Ring Counters | 3-41 |
| Ethernet Counters | 3-43 |
| FDDI Counters. | 3-45 |
| Driver Firmware | 3-47 |
| DriverFirmwareSize Value. | 3-47 |
| DriverFirmwareBuffer Value. | 3-47 |

4 CMSM/CTSM Structures and Variables

| | |
|---|------|
| Overview | 4-1 |
| CMSM Data Access. | 4-2 |
| DADSP_TO_CMSMADSP Macro | 4-2 |
| CMSMVirtualBoardLink Pointers | 4-2 |
| CMSMDefaultVirtualBoard Pointer | 4-4 |
| CMSMStatusFlags Variable | 4-4 |
| CMSMTxFreeCount Variable | 4-4 |
| CMSMPriorityTxFreeCount | 4-5 |
| CMSMMaxFrameHeaderSize Variable | 4-5 |
| CMSMPhysNodeAddress Variable | 4-6 |
| Data Structures | 4-6 |
| Fragment Structure | 4-8 |
| Receive Control Blocks (RCBs) | 4-8 |
| RCB Structure | 4-10 |
| Transmit Control Blocks (TCBs). | 4-12 |

| | |
|-----------------------------|------|
| TCB Structure | 4-12 |
| CMSM_CONFIG_TABLE | 4-15 |
| CTSM_CONFIG_TABLE. | 4-17 |

5 CHSM Functions

| | |
|---|------|
| Overview | 5-1 |
| Initialization. | 5-3 |
| DriverInit | 5-11 |
| DriverRemove | 5-14 |
| Board Service Routine | 5-15 |
| Packet Reception | 5-15 |
| Reception Methods. | 5-15 |
| Reception Method—Option 1. | 5-15 |
| Reception Method—Option 2. | 5-16 |
| Reception Method—Option 3. | 5-16 |
| Reception Method—Option 4. | 5-17 |
| DriverISR. | 5-19 |
| DriverPoll. | 5-22 |
| Packet Transmission. | 5-24 |
| Transmission Methods | 5-24 |
| Transmission Method—Option 1 | 5-24 |
| Transmission Method—Option 2 | 5-25 |
| Priority Transmission Support | 5-26 |
| Adapters that Need Physical Addresses | 5-28 |
| DriverPriorityQueueSupport | 5-29 |
| DriverSend | 5-31 |
| Control Procedures | 5-34 |
| DriverReset | 5-36 |
| DriverShutdown | 5-39 |
| DriverMulticastChange | 5-43 |
| DriverPromiscuousChange | 5-46 |
| DriverStatisticsChange (optional) | 5-50 |
| DriverRxLookAheadChange (optional) | 5-52 |
| DriverManagement (optional). | 5-54 |
| DriverEnableInterrupt | 5-56 |
| DriverDisableInterrupt | 5-57 |
| DriverDisableInterrupt2 | 5-59 |
| Timeout Detection | 5-61 |
| DriverAES | 5-62 |

6 CTSM Functions

| | |
|--------------------|-----|
| Overview | 6-1 |
|--------------------|-----|

| | |
|---|------|
| <CTSM>BuildTransmitControlBlock | 6-2 |
| <CTSM>CancelPrioritySend | 6-5 |
| <CTSM>FastProcessGetRCB | 6-7 |
| <CTSM>FastRcvComplete | 6-9 |
| <CTSM>FastRcvCompleteStatus | 6-11 |
| <CTSM>FastSendComplete | 6-13 |
| <CTSM>GetConfigInfo | 6-15 |
| <CTSM>GetHSMIFLevel | 6-17 |
| <CTSM>GetRCB | 6-18 |
| <CTSM>ProcessGetRCB | 6-21 |
| <CTSM>RcvComplete | 6-24 |
| <CTSM>RcvCompleteStatus | 6-26 |
| <CTSM>RegisterHSM | 6-28 |
| <CTSM>SendComplete | 6-30 |
| <CTSM>UpdateMulticast | 6-32 |
| 7 CMSM Functions | |
| Overview | 7-1 |
| CMSMAddToCounter | 7-2 |
| CMSMAlloc | 7-4 |
| CMSMAllocateMultipleRCBs | 7-6 |
| CMSMAllocPages | 7-9 |
| CMSMAllocateRCB | 7-11 |
| CMSMCancelAES | 7-13 |
| CMSMControlComplete | 7-14 |
| CMSMDeRegisterResource | 7-16 |
| CMSMDriverRemove | 7-19 |
| CMSMECBPhysToLogFrag | 7-20 |
| CMSMEnablePolling | 7-22 |
| CMSMFree | 7-24 |
| CMSMFreePages | 7-26 |
| CMSMGetAlignment | 7-27 |
| CMSMGetBusInfo | 7-29 |
| CMSMGetBusSpecificInfo | 7-30 |
| CMSMGetBusType | 7-34 |
| CMSMGetCardConfigInfo | 7-36 |
| CMSMGetConfigInfo | 7-42 |
| CMSMGetCurrentTime | 7-44 |
| CMSMGetHINFromHINName | 7-46 |
| CMSMGetHINNameFromHIN | 7-48 |
| CMSMGetInstanceNumber | 7-50 |
| CMSMGetInstanceNumberMapping | 7-52 |
| CMSMGetMicroTimer | 7-54 |

| | |
|--|-------|
| CMSMGetPhysical | 7-55 |
| CMSMGetPhysList | 7-56 |
| CMSMGetPollSupportLevel | 7-58 |
| CMSMGetUniquelIdentifier | 7-60 |
| CMSMGetUniquelIdentifierParameters | 7-63 |
| CMSMHardwareFailure | 7-65 |
| CMSMIncrCounter | 7-67 |
| CMSMInitAlloc | 7-68 |
| CMSMInitParser | 7-69 |
| CMSMNESLDeRegisterConsumer | 7-71 |
| CMSMNESLDeRegisterProducer | 7-73 |
| CMSMNESLProduceEvent | 7-75 |
| CMSMNESLProduceMLIDEvent | 7-79 |
| CMSMNESLRegisterConsumer | 7-83 |
| CMSMNESLRegisterProducer | 7-87 |
| CMSMParseDriverParameters | 7-91 |
| CMSMParseSingleParameter | 7-95 |
| CMSMPrintString | 7-97 |
| CMSMRdConfigSpacex | 7-100 |
| CMSMReadPhysicalMemory | 7-102 |
| CMSMRegisterHardwareOptions | 7-104 |
| CMSMRegisterMLID | 7-106 |
| CMSMRegisterResource | 7-108 |
| CMSMReRegisterHardwareOptions | 7-112 |
| CMSMResetMLID | 7-116 |
| CMSMResumePolling | 7-118 |
| CMSMReturnDriverResources | 7-120 |
| CMSMReturnMultipleRCBs | 7-122 |
| CMSMReturnRCB | 7-124 |
| CMSMScanBusInfo | 7-126 |
| CMSMScheduleAES | 7-128 |
| CMSMSearchAdapter | 7-131 |
| CMSMServiceEvents | 7-135 |
| CMSMSetHardwareInterrupt | 7-136 |
| CMSMShutdownMLID | 7-138 |
| CMSMSuspendPolling | 7-140 |
| CMSMTCBPhysToLogFrgs | 7-142 |
| CMSMUpdateConfigTables | 7-144 |
| CMSMWrtConfigSpacex | 7-146 |
| CMSMWritePhysicalMemory | 7-148 |

8 NetWare Bus Interface

| | |
|--------------------|-----|
| Overview | 8-1 |
|--------------------|-----|

| | |
|---|------|
| Bus Architecture | 8-2 |
| Multiple Bus Platforms. | 8-2 |
| Memory Mapping and Address Manipulation | 8-3 |
| Byte Order | 8-4 |
| DMACleanup | 8-5 |
| DMASStart | 8-6 |
| DMASStatus | 8-9 |
| FreeBusMemory | 8-10 |
| Inx | 8-12 |
| InBufx | 8-14 |
| MapBusMemory | 8-16 |
| MovFastFromBus | 8-18 |
| MovFastToBus | 8-20 |
| MovFromBusx | 8-22 |
| MovToBusx | 8-25 |
| Outx | 8-28 |
| OutBufx | 8-30 |
| Rdx | 8-32 |
| Setx | 8-34 |
| Slow | 8-36 |
| Wrtx | 8-37 |
| Appendix A Language Enabling | |
| Overview | A-1 |
| Language Enabling Procedure | A-1 |
| Appendix B Event Control Blocks (ECBs) | |
| Overview | B-1 |
| ECB Aware Adapters | B-1 |
| Event Control Block Structure. | B-2 |
| Relationship between Receive ECBs and RCBs | B-10 |
| Relationship between Transmit ECBs and TCBs | B-11 |
| Appendix C Platform Specific Information | |
| Overview | C-1 |
| Intel Processors | C-1 |
| Building the CHSM | C-1 |
| Creating the Source Files | C-1 |
| Compiling the Source Files | C-1 |
| Linking the Object Files | C-2 |
| Linker Definition File | C-2 |
| MLID Configuration File | C-5 |

| | |
|--|-----|
| Load Keywords and Parameters | C-5 |
|--|-----|

Appendix D Portability Issues

| | |
|--------------------------------------|------|
| Overview | D-1 |
| Portability Rules | D-1 |
| Translation Limits | D-4 |
| Coding Assumptions | D-5 |
| Data Packing and Alignment | D-5 |
| Portability Macros | D-6 |
| COPY_FROM_HILO_UINTx | D-7 |
| COPY_FROM_LOHI_UINTx | D-8 |
| COPY_TO_HILO_UINTx | D-9 |
| COPY_TO_LOHI_UINTx | D-10 |
| COPY_UINTx | D-11 |
| GET_HILO_UINTx | D-12 |
| GET_LOHI_UINTx | D-13 |
| GET_UINTx | D-14 |
| HOST_FROM_HILO_UINTx | D-15 |
| HOST_FROM_LOHI_UINTx | D-16 |
| HOST_TO_HILO_UINTx | D-17 |
| HOST_TO_LOHI_UINTx | D-18 |
| PUT_HILO_UINTx | D-19 |
| PUT_LOHI_UINTx | D-20 |
| PUT_UINTx | D-21 |
| UINTx_EQUAL | D-22 |
| VALUE_FROM_HILO_UINTx | D-23 |
| VALUE_FROM_LOHI_UINTx | D-24 |
| VALUE_TO_HILO_UINTx | D-25 |
| VALUE_TO_LOHI_UINTx | D-26 |

Appendix E NESL Support

| | |
|---|-----|
| Overview | E-1 |
| Registering and Deregistering Event Producers | E-2 |
| Registering and Deregistering Event Consumers | E-2 |
| NESL Structures | E-3 |
| EPB (Event Parameter Block) Structure | E-3 |
| NESL_ECB Structure | E-4 |
| Events and Types | E-7 |
| Event Names | E-7 |
| Event Types | E-8 |
| Service Suspend Types | E-8 |
| | E-9 |

| | |
|--|------|
| Suspend Request | E-9 |
| Service Resumed Types. | E-10 |
| Service/Status Changed Types | E-11 |
| CMSM NESL String Exports | E-12 |
| NESL Return Codes | E-13 |
| NESL Event Flags. | E-14 |
| NESL OSI Layer Definitions. | E-15 |

Glossary

Revision History

Index

Trademarks

x ODI Specification: Hardware Specific Modules (HSMs) (C Language)

Figures

| | | |
|------------|--|------|
| Figure 1-1 | The ODI Specification Elements | 1-2 |
| Figure 1-2 | How ODI Fits into the OSI Model | 1-3 |
| Figure 1-3 | The Multiple Protocol Interface (MPI) | 1-4 |
| Figure 1-4 | The Multiple Link Interface (MLI) | 1-7 |
| Figure 1-5 | MLID Modules | 1-8 |
| Figure 1-6 | Data Flow from Application to LSL | 1-10 |
| Figure 1-7 | Data Flow from the LSL to the Board | 1-11 |
| Figure 1-8 | Data Flow from the Board to the Wire | 1-11 |
| Figure 1-9 | Receive Data Flow from Wire to Application | 1-12 |
| Figure 2-1 | Implementation of Multiple Frame Support | 2-10 |
| Figure 3-1 | Frame and Adapter Data Space. | 3-10 |
| Figure 3-2 | MLIDCFG_ModeFlags Field Default Values | 3-23 |
| Figure 3-3 | MLIDCFG_Flags Field. | 3-27 |
| Figure 3-4 | MLID_SharingFlags Field Default Values | 3-29 |
| Figure 3-5 | Driver Frame and Adapter Data Space | 3-33 |
| Figure 4-1 | Packet Transfer through the MLID | 4-7 |
| Figure 8-1 | Multiple Bus Platform Example | 8-2 |

Figure B-1 Packet Transfer through MLID B-2

Figure B-2 RCB Correspondence to ECB B-10

Figure B-3 Relationship between TCB and ECB. B-11

Tables

| | | |
|------------|--|------|
| Table 2-1 | Execution Time of MLID Routines | 2-8 |
| Table 3-1 | Driver Parameter Block Field Descriptions | 3-3 |
| Table 3-2 | Driver Configuration Table Field Descriptions | 3-12 |
| Table 3-3 | MLIDCFG_ModeFlags Bits Description | 3-23 |
| Table 3-4 | MLIDCFG_Flags Bits Description | 3-27 |
| Table 3-5 | MLIDCFG_SharingFlags Bits Description | 3-29 |
| Table 3-6 | Frame Types Versus Size Fields | 3-31 |
| Table 3-7 | MLID Statistics Table Fields. | 3-36 |
| Table 3-8 | MLID Statistics Table Generic Counters | 3-38 |
| Table 3-9 | Media Specific Counters for Token-Ring | 3-41 |
| Table 3-10 | Media Specific Counters for Ethernet | 3-43 |
| Table 3-11 | Media Specific Counters for FDDI | 3-45 |
| Table 4-1 | Fragment Structure Field Descriptions | 4-8 |
| Table 4-2 | Programmed RCB Field Description | 4-10 |
| Table 4-3 | TCB Field Descriptions | 4-13 |
| Table 4-4 | Interpretation of Parameter0, Parameter1, and Parameter2 | 4-19 |
| Table 4-5 | Input and Results for Each Character Type | 4-26 |
| Table 5-1 | | |

| | | |
|-----------|---|------|
| Table 5-2 | DriverEndofChainFlag Values | 5-18 |
| Table B-1 | Code Path of Control Functions | 5-34 |
| Table C-1 | Fragment Structure and ECB Field Descriptions | B-3 |
| Table C-2 | Linker Definition File Example Definitions | C-3 |
| | Load Keywords and Parameters Descriptions | C-6 |

Preface

Overview

This document provides the information necessary to develop the C language Hardware Specific Module™ (CHSM™) portion of a NetWare driver. Drivers written using the information in this document conform to the Open Data-Link Interface™ (ODI™) specification.

Important



This document is written with the assumption that you are writing a driver in the C language; however, you can write portions of the driver in assembly language if you wish.

Note



This specification does not apply to 16-bit DOS ODI platforms.

This document does not describe the full ODI specification, but explains the development of a driver using the Novell-provided development modules.

This document is organized as follows:

- *Chapter 1: Introduction to ODI*

Describes the NetWare environment by presenting a brief overview of the ODI architecture and an introduction to the ODI LAN driver structure.

- *Chapter 2: ODI C Language HSM Overview*

Provides an overview of the driver CHSM and its required functions.

- *Chapter 3: CHSM Data Structures and Variables*

Describes the data structures and variables that the CHSM developer must define.

- *Chapter 4: CMSM/CTSM Structures and Variables*
Describes the data structures and variables provided by the C language Media Support Module (CMSM) and the C language Topology Specific Module (CTSM) for CHSM development.
- *Chapter 5: CHSM Functions*
Describes the functions that the CHSM developer must provide.
- *Chapter 6: CTSM Functions*
Describes the CTSM functions available for CHSM development.
- *Chapter 7: CMSM Functions*
Describes the CMSM functions available for CHSM development.
- *Chapter 8: NetWare Bus Interface*
Describes the NetWare Bus Interface (NBI) functions needed by the CHSM to isolate the CHSM from the platform's bus architecture.
- *Appendix A: Language Enabling*
Describes how to enable your CHSM to display messages in more than one language.
- *Appendix B: Event Control Blocks (ECBs)*
Provides information used in writing drivers for ECB aware adapters.
- *Appendix C: Platform Specific Information*
Contains operating system and processor specific information that can be used to help in driver development on a particular platform.
- *Appendix D: Portability Issues*
Describes the rules you must follow to ensure that your driver is portable to different operating systems and/or processors. Also, describes a set of macros that you can use to help make your driver portable.

Prerequisites to Using this Manual

The developer should be experienced with the ANSI C programming language and have a sound understanding of reentrant coding, event-driven systems, and interrupt-driven device drivers. The developer should also be familiar with the features of the processor(s) used on the computer platform(s) where the driver will be used.

Manual Conventions

All numbers in this document are decimal unless otherwise specified. Where bit fields within a byte are specified, bit 0 is assumed to be the low-order bit.

<" and ">" are used to enclose symbolic names for actual file names. For example, the developer must replace <CTSM> with the appropriate media type, depending on which module is used.

UNUSED is used in this specification to symbolize an invalid or unknown value.

The pseudocode used in this specification is intended to illustrate a general flow of events and does not necessarily describe optimized code.

Data Type Definitions

The following data types are defined:

| | |
|--------------------|--|
| MEON | 8-bit unsigned value that contains a single byte character or a portion of a double-byte character |
| MEON_STRING | NULL-terminated string of MEON |
| UINT8 | 8-bit unsigned integer |
| UINT16 | 16-bit unsigned integer |
| UINT32 | 32-bit unsigned integer |
| UINT64 | 64-bit unsigned integer |

MSG_ID

Integer constant defined by the developer and message handling system to identify a string

Structure Definitions

The following are definitions of structures used in this specification.

DRIVER_DATA Structure

```
typedef struct _DRIVER_DATA_
{
    /* Place CHSMs data here */

    /* Statistics Table */

    MLID_STATS_TABLE    StatsTable;

    /* Generic Statistics Table Entries */

    StatTableEntry for TotalTxPacket
    StatTableEntry for TotalRxPacket
    StatTableEntry for NoECBAvailable
    StatTableEntry for PacketTxTooBig
    StatTableEntry for PacketTxTooSmall
    StatTableEntry for PacketRxOverflow
    StatTableEntry for PacketRxTooBig
    StatTableEntry for PacketRxTooSmall
    StatTableEntry for PacketTxMiscError
    StatTableEntry for PacketRxMiscError
    StatTableEntry for RetryTx
    StatTableEntry for ChecksumError
    StatTableEntry for HardwareRxMismatch
    StatTableEntry for TotalTxOKByte
    StatTableEntry for TotalRxOKByte
    StatTableEntry for TotalGroupAddrTx
    StatTableEntry for TotalGroupAddrRx
    StatTableEntry for AdapterReset
    StatTableEntry for AdapterOprTimeStamp
    StatTableEntry for QDepth
}
```

```

/* Media Statistics Table Entries go here */

/* Custom Statistics Table Entries go here */

/* Generic Counters */

UINT32 TotalTxPacketCount;
UINT32 TotalRxPacketCount;
UINT32 NoECBAvailableCount;
UINT32 PacketTxTooBigCount;
UINT32 PacketTxTooSmallCount;
UINT32 PacketRxOverflowCount;
UINT32 PacketRxTooBigCount;
UINT32 PacketTxMiscErrorCount;
UINT32 PacketRxMiscErrorCount;
UINT32 RetryTxCount;
UINT32 ChecksumErrorCount;
UINT32 HardwareRxMismatchCount;
UINT64 TotalTxOKByteCount;
UINT64 TotalRxOKByteCount;
UINT32 TotalGroupAddrTxCount;
UINT32 TotalGroupAddrRxCount;
UINT32 AdapterResetCount;
UINT32 AdapterOprTimeStamp;
UINT32 QDepth;

/* Media Counters go here */
/* Custom Counters go here */
} DRIVER_DATA;

```

Important



DRIVER_DATA is unique for each CHSM. However, it must contain: the MLID_STATS_TABLE structure; all generic, media, and custom STAT_TABLE_ENTRY structures; and all generic, media, and custom counter variables.

MODULE_HANDLE Structure

```
typedef struct _MODULE_HANDLE_
{
    /* platform specific module handle information
       defined in cmsm.h */
} MODULE_HANDLE;
```

GROUP_ADDR_LIST_NODE Structure

```
typedef struct _GROUP_ADDR_LIST_NODE_
{
    NODE_ADDR GRP_ADDR;
    UINT16 GRP_ADDR_COUNT;
} GROUP_ADDR_LIST_NODE;
```

NODE_ADDR Structure

```
typedef struct _NODE_ADDR_ {
    UINT8 nodeAddress [ADDR_SIZE];
} NODE_ADDR;
```

Where *ADDR_SIZE* is the number of bytes needed to identify an address and is currently defined by the following constant:

```
#define ADDR_SIZE 6
```

PROT_ID Structure

```
typedef struct _PROT_ID_ {
    UINT8 protocolID [PID_SIZE];
} PROT_ID;
```

Where *PID_SIZE* is the number of bytes needed to identify a protocol stack and is currently defined by the following constant:

```
#define PID_SIZE 6
```

xx ODI Specification: Hardware Specific Modules (HSMs) (C Language)

CHSM_STACK Structure

```

typedef struct _CHSM_STACK_
{
    struct _MODULE_HANDLE_ *ModuleHandle;
    SCREEN_HANDLE          *ScreenHandle;
    MEON                   *CommandLine;
    MEON                   *ModuleLoadPath;
    UINT32                  UnitializedDataLength;
    void                   *CustomDataFileHandle;
    UINT32                  (*FileRead)(
        void *FileHandle,
        UINT32 FileOffset,
        void *FileBuffer,
        UINT32 FileSize);
    UINT32                  CustomDataOffset;
    UINT32                  CustomDataSize;
    UINT32                  NumMsgs;
    MEON                   **Msgs;
} CHSM_STACK;

```

Enumeration Definitions

The following enumerations are used throughout this specification to define return values.

AES_TYPE Enumeration

```

typedef enum _AES_TYPE_
{
    AES_TYPE_PRIVILEGED_ONE_SHOT,
    AES_TYPE_PRIVILEGED_CONTINUOUS,
    AES_TYPE_PROCESS_ONE_SHOT,
    AES_TYPE_PROCESS_CONTINUOUS
} AES_TYPE;

```

BOOLEAN Enumeration

```
typedef enum _BOOLEAN_
{
    FALSE,
    TRUE
} BOOLEAN;
```

ODI_NBI Enumeration

```
typedef enum _ODI_NBI_
{
    ODI_NBI_SUCCESSFUL,
    ODI_NBI_PROTECTION_VIOLATION,
    ODI_NBI_HARDWARE_ERROR,
    ODI_NBI_MEMORY_ERROR,
    ODI_NBI_PARAMETER_ERROR,
    ODI_NBI_UNSUPPORTED_OPERATION,
    ODI_NBI_ITEM_NOT_PRESENT,
    ODI_NBI_NO_MORE_ITEMS,
    ODI_NBI_FAIL
} ODI_NBI;
```


ODISTAT Enumeration

```

typedef enum _ODISTAT_
{
    ODISTAT_SUCCESSFUL                =0,
    ODISTAT_RESPONSE_DELAYED          =1,
    ODISTAT_SUCCESS_TAKEN              =2,
    ODISTAT_BAD_COMMAND                =-127,
    ODISTAT_BAD_PARAMETER              =-126,
    ODISTAT_DUPLICATE_ENTRY            =-125,
    ODISTAT_FAIL                       =-124,
    ODISTAT_ITEM_NOT_PRESENT           =-123,
    ODISTAT_NO_MORE_ITEMS              =-122,
    ODISTAT_MLID_SHUTDOWN              =-121,
    ODISTAT_NO_SUCH_HANDLER            =-120,
    ODISTAT_OUT_OF_RESOURCES           =-119,
    ODISTAT_RX_OVERFLOW                =-118,
    ODISTAT_RX_IN_CRITICAL_SECTION     =-117,
    ODISTAT_TRANSMIT_FAILED            =-116,
    ODISTAT_PACKET_UNDELIVERABLE       =-115,
    ODISTAT_CANCELED                   =-4
} ODISTAT;

```

ODI_STAT Enumeration

```

typedef enum _ODI_STAT_
{
    ODI_STAT_UNUSED                    =-1,
    ODI_STAT_UINT32                    =0,
    ODI_STAT_UINT64                    =1,
    ODI_STAT_MEON_STRING                =2,
    ODI_STAT_UNTYPED                    =3
} ODI_STAT;

```

REG_TYPE Enumeration

```
typedef enum _REG_TYPE_
{
    REG_TYPE_NEW_ADAPTER,
    REG_TYPE_NEW_FRAME,
    REG_TYPE_NEW_CHANNEL,
    REG_TYPE_FAIL
} REG_TYPE;
```

OPERATION_SCOPE Enumeration

```
typedef enum _OPERATION_SCOPE
{
    OP_SCOPE_ADAPTER,
    OP_SCOPE_LOGICAL_BOARD,
} OPERATION_SCOPE;
```

Portability Requirements

To ensure that a driver is portable across different operating systems and/or processors, you must adhere to the following rules:

- *Write your driver in ANSI C—this is extremely important.*
- *In general, do not declare variables with standard C language types such as short, long, int, char. Declare variables with abstract types or typedefs, such as BYTE, MEON, UINT32, that are appropriate for the processor/operating system combination. However, in some cases such as counters, it may be more efficient to use int instead of an abstract type.*
- *Ensure that all members of a structure containing data that is sent to and from the LAN are given unique, abstract types. Also, ensure that the references to these members use the appropriate misalignment correction macros and byte order correction macros described in Appendix D: Portability Issues.*

Referenced Documents

The following is a list of Novell documents referenced in this specification.

- *NetWare Client SDK*
- *ODI Specification Supplement: Brouter Support*
- *ODI Specification Supplement: Canonical and Noncanonical Addressing*
- *ODI Specification Supplement: Standard MLID Message Definitions*
- *ODI Specification Supplement: Frame Types and Protocol IDs*
- *ODI Specification Supplement: The Hub Management Interface*
- *ODI Specification Supplement: The MLID Installation Information File*
- *ODI Specification Supplement: Source Routing*
- *NetWare Wide Area Network Open Data-Link Interface Specification*
- *IEEE Std. 803.5 -1989*

chapter **1** *Introduction to ODI*

Overview

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

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



Note

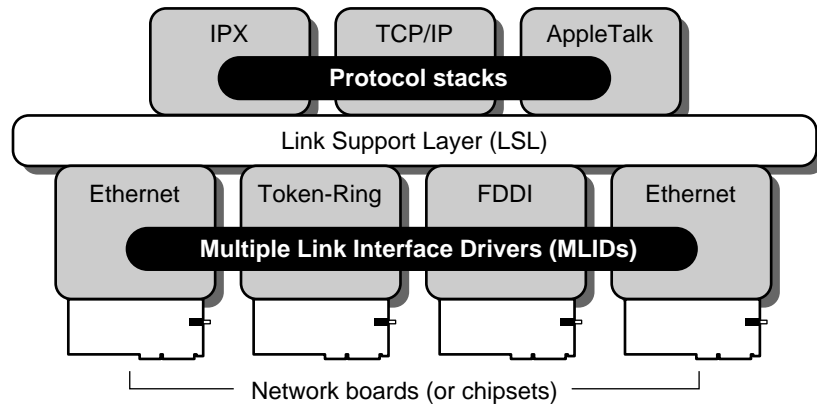
The terms *MLID* and *LAN driver* are interchangeable.

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

Open Data-Link Interface (ODI)

C language HSMs and protocol stacks must conform to the ODI specification. Figure 1.1 illustrates the elements that make up the ODI specification.

Figure 1-1
The ODI Specification Elements



The ODI specification allows multiple network protocols and adapters (physical boards) to be used concurrently on the same client or file server. It provides a flexible, high-performance data link layer interface to network layer protocol stacks. The ODI specification is comprised of the three elements listed below and illustrated above in Figure 1-1.

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

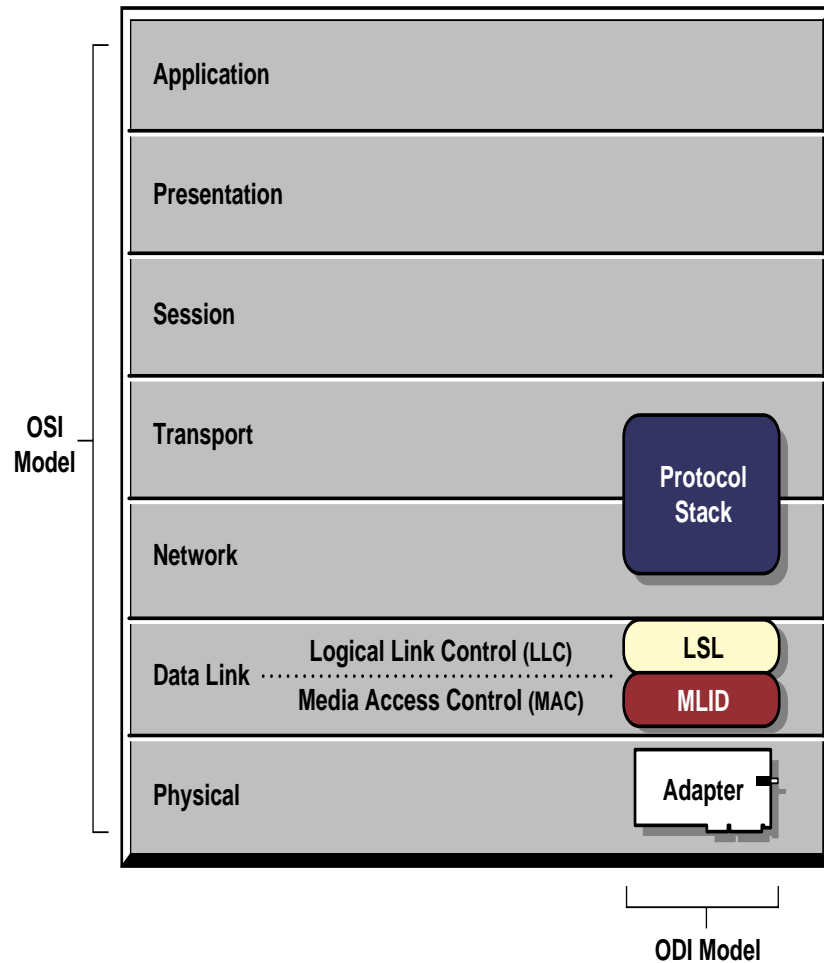
Protocol Stacks

Network layer protocol stacks transmit and receive data over a logical or physical network. They handle routing, connection services, and APIs. They also provide an interface that allows higher layer protocols and applications to access the protocol stack's services.

As a general rule, protocol stacks written to the ODI specification provide OSI (Open Systems Interconnection) network layer functionality; however, they are not limited to this. Figure 1-2 illustrates the ODI/OSI correspondence.

1-2 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

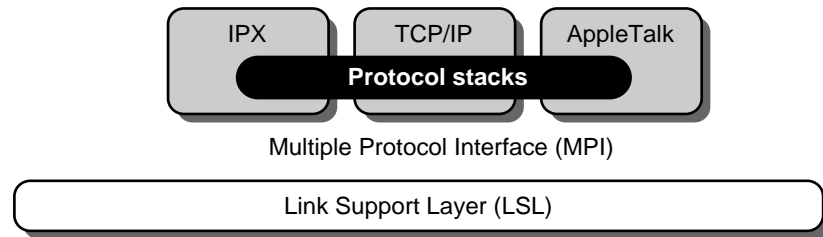
Figure 1-2
How ODI Fits into the OSI Model



The Multiple Protocol Interface (MPI)

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

Figure 1-3
The Multiple Protocol Interface (MPI)



Link Support Layer (LSL)

The LSL handles the communication between protocol stacks and MLIDs. The ODI specification allows physical topologies to support many different types of protocols. Consequently, the MLID may receive packets for any of the different protocol stacks residing in the system.

For example, an Ethernet network might support all of the following protocols: IPX™, TCP/IP, AppleTalk*, and LAT* (a Digital Equipment Corporation protocol). The LSL determines which protocol stack is to receive the packet. Then, the protocol stack determines where the packet should be sent and sends it. The LSL then directs the packet to the appropriate MLID.

The LSL tracks all protocols and MLIDs currently in the system and provides a consistent method of accessing each protocol module and MLID module.

In addition, the LSL performs the following services:

1-4 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

- Queues and recovers Event Control Blocks (ECBs) for later use. (ECBs are control structures used to send and receive packets and to schedule events.)
- Registers and deregisters protocol stacks.
- Allows protocol stacks to obtain timing services.
- Allows protocol stacks to determine stack IDs and protocol IDs.
- Allows protocol stacks to obtain MLID statistics.
- Allows protocol stacks to bind with MLIDs.
- Allows protocol stacks to transmit and receive packets through MLIDs.
- Maintains lists of all active protocol stacks and MLIDs.
- Allows protocol stacks to obtain information about MLIDs and other protocol stacks.
- Allows protocol stacks to change the operational state of MLIDs. (For example, the protocol stack can cause the MLID to shut down or reset.)

Introduction to ODI **1-5**

Multiple Link Interface Drivers (MLIDs)

MLID Functionality

Multiple Link Interface Drivers (MLIDs) are device drivers that handle the sending and receiving of packets to and from LAN adapters.

Note



A LAN adapter is any network controller that provides access across a network. A LAN adapter may be present directly on the motherboard in an embedded system, or it may be a network interface card inserted into the computer bus. The MLID interface is determined by the adapter hardware.

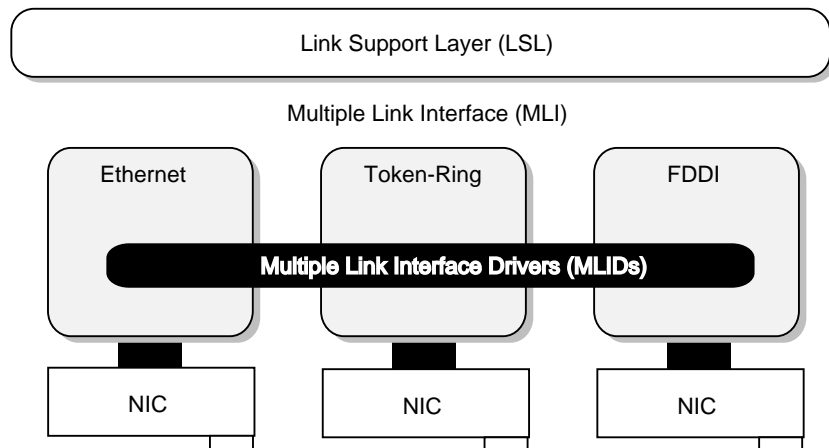
The MLID determines the packet's frame type and then strips or appends the frame header to the packet. (Ethernet and Token Ring are examples of frame types. Refer to *ODI Specification: Frame Types and Protocol IDs* for a list of currently supported frame types and their protocol IDs.)

MLIDs can handle packets from various different types of protocols because MLIDs do not interpret packets. MLIDs use Event Control Blocks (ECBs) to pass packets to the LSL. ECBs are data structures that the MLID uses to send and receive packets, and to schedule events. (See Appendix B: "Event Control Blocks (ECBs)" for complete information on ECBs.) The MLID communicates with the LSL through the Multiple Link Interface™ (MLI™).

The Multiple Link Interface (MLI)

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

Figure 1-4
The Multiple Link Interface (MLI)



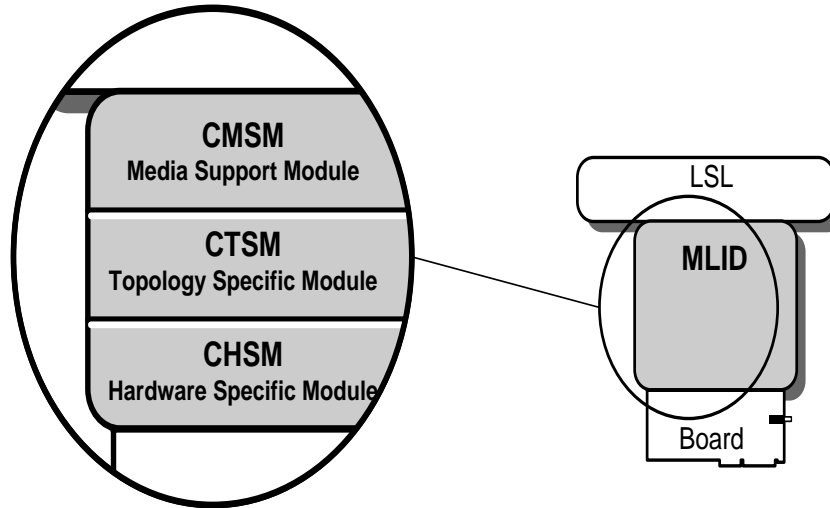
LAN Driver Toolkit

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

- C Language Media Support Module (CMSM)
- C Language Topology Specific Modules (CTSM)

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

Figure 1-5
MLID Modules



C Language Media Support Module (CMSM)

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

C Language Topology Specific Module (CTSM)

The C Language Topology Specific Module (CTSM) manages the operations for specific topologies. CTSMs provide support for the standardized topology types of Ethernet, Token-Ring, and FDDI. Multiple frame type support is implemented in the CTSM so that all frame types for a given topology are supported. The possible topology modules are as follows:

- ETHERTSM.NLM
- TOKENTSM.NLM
- FDDITSM.NLM

Source code for each CTSM is provided in the *Novell LAN Driver Developer's Kit*.

1-8 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

Although not recommended, you can create your own proprietary topology modules by modifying existing CTSMs, or by creating new modules that provide the same functionality as the Novell CTSMs. However, all topology modules must conform to the functionality defined in this specification, and the modules and the APIs must have unique names.

C Language Hardware Specific Module (CHSM)

You create the C Language Hardware Specific Module (CHSM) for your specific LAN adapter. A CHSM handles all hardware interactions.

The primary functions of a CHSM are as follows:

- Adapter initialization
- Adapter reset
- Adapter shut down
- Packet reception
- Packet transmission

Additional procedures that a CHSM may provide support for are as follows:

- Timeout detection
- Multicast addressing
- Promiscuous mode reception

When you use the LAN driver toolkit to develop an MLID, the CHSM is the only module that you write.

This document uses the term CHSM to refer to that portion of the MLID that you develop with this toolkit.

NetWare Bus Interface (NBI)

The NetWare Bus Interface (NBI) is the machine specific code that gives the NetWare Driver a uniform view of the system, regardless of the architecture. The NBI deals with such things as interrupts and stack manipulation.

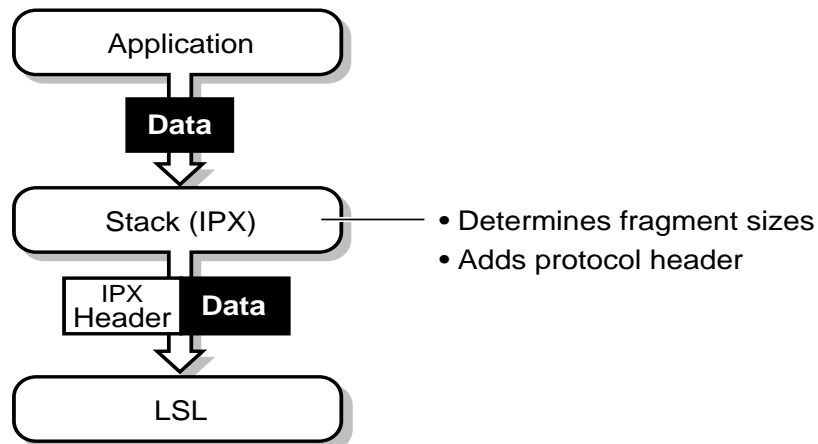
Data Flow

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

Send Data Flow

As Figure 1-6 illustrates, the protocol stack receives data from the application above it, determines whether the packet must be split into fragments, determines the size of the fragments, adds the appropriate protocol header to the data packet, and sends it to the LSL. The LSL isolates the protocol stack from the topology and LAN medium below it. The protocol stack simply passes data to the LSL. The LSL directs the packet to the appropriate MLID, which then takes care of the topology-specific information. This is the reason ODI protocol stacks are known as being media and frame type independent.

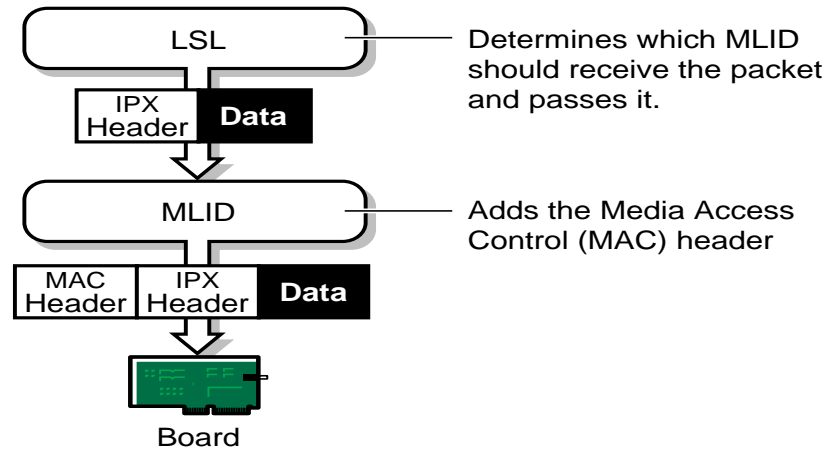
Figure 1-6
Data Flow from Application to LSL



As illustrated by Figure 1-7, the LSL directs the packet to the appropriate MLID. The MLID then adds the MAC header to the packet and hands the packet to the LAN adapter.

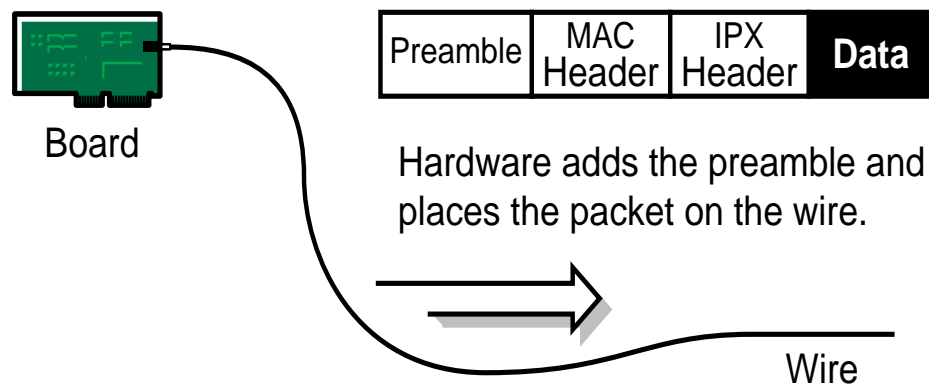
1-10 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

Figure 1-7
Data Flow from the LSL to the Board



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

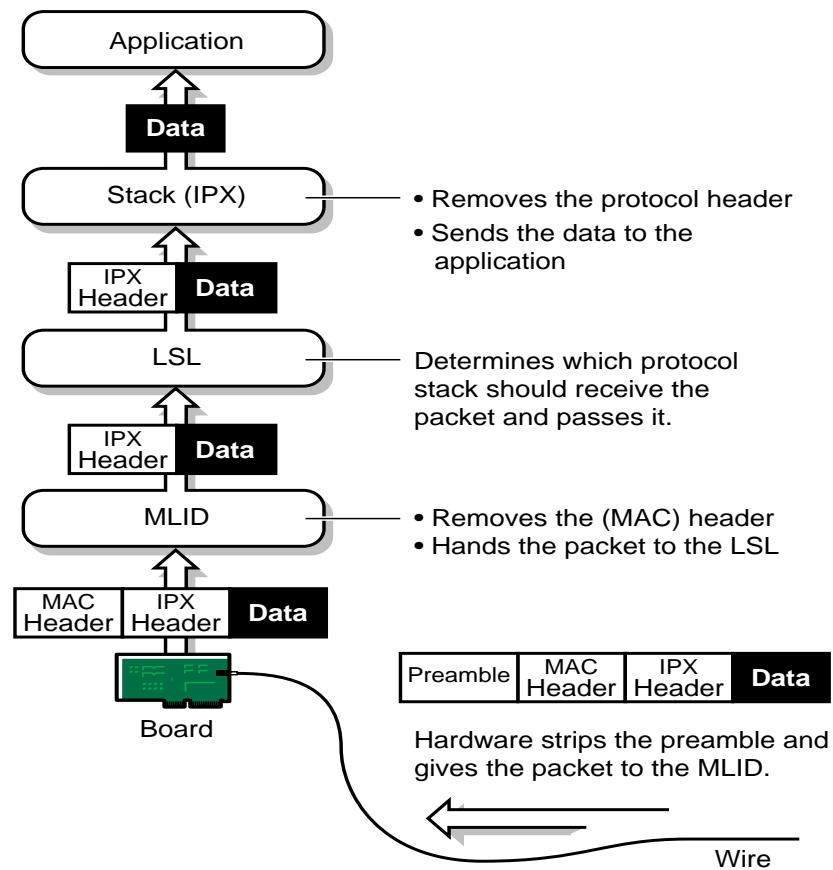
Figure 1-8
Data Flow from the Board to the Wire



Receive Data Flow

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

Figure 1-9
Receive Data Flow from Wire to Application



chapter **2** *ODI C Language HSM Overview*

Overview

This chapter provides an overview of the C Language Hardware Specific Module (CHSM) of an ODI MLID. Specific issues that influence the development of the CHSM are also addressed.

CHSM Procedures

The CHSM contains procedures that perform the following types of functions:

- Initialization
- Board service
- Packet transmission
- Control procedures
- Timeout detection
- Driver removal

You may add additional procedures to support the specific hardware features of a particular LAN adapter. Additional procedures may be required if Hub Management or Source Bridge Routing (Brouter) is supported. (Hub Management and Source Bridge Routing are described later in this chapter.)

Brief descriptions of the required CHSM procedures are provided on the following pages. These descriptions are general and do not apply in every case, nor do they describe every possible case. Detailed descriptions of the required procedures (including pseudocode) are provided in Chapter 5, "CHSM Functions".

Initialization

The CHSM's initialization function, **DriverInit**, initializes the LAN adapter hardware. **DriverInit** uses CMSM functions and CTSM functions to do the following tasks:

- Allocate memory for MLID variables and structures
- Parse the standard load command line options
- Process custom command line parameters and custom firmware
- Register the MLID and hardware
- Setup the hardware for the board's interrupt service routine or polling procedure
- Schedule callback events for timeout detection and recovery
- Handle any initialization errors

Board Service Routine

The CHSM's board service routine will generally need to detect and handle the following events:

- Packet reception and reception complete
- Reception error
- Transmission complete
- Transmission error

The MLID can be notified of these events by using either an interrupt service routine (**DriverISR**) or a polling procedure (**DriverPoll**).

2-2 ODI Specification: Hardware Specific Modules (HSMs) (C Language)



Note

If **DriverPoll** is used, we recommend that you implement interrupt backup.

Error detection and handling are optional in cases where the hardware is able to handle transmission and reception errors without MLID intervention. However, the MLID must be notified of errors so it can maintain the statistics table. (The statistics table is described in Chapter 3, "CHSM Data Structures and Variables.")

Packet Transmission

Before a packet is transmitted, the CTSM builds the frame and media headers for the packet. Then, the CTSM collects the header and data fragments, and transmits the packet by calling the CHSM function **DriverSend**. **DriverSend** is called whenever a packet needs to be transmitted.

Control Procedures

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

Timeout Detection

The CHSM may need to be called regularly to inspect the adapter's status. If the CHSM requires this capability, it can enable the **DriverAES** routine. Once enabled, the CMSM will call this routine at an interval specified by the developer.

For example, the MLID might need to periodically inspect the adapter to determine if it has failed to complete a transmission. If a timeout error has occurred, the procedure will discard the packet being sent, increment the appropriate statistics counter(s), reset the board if appropriate, and begin transmitting the next packet in the send queue.

Driver Removal

Every CHSM must have a removal procedure that allows the user to unload it. This procedure, **DriverRemove**, must shut down the physical board and return

all resources allocated to the MLID. The CMSM provides routines that handle the return of resources, but the CHSM is responsible for deciding when it must be done and for calling the appropriate CMSM routines.

CHSM Data Structures and Variables

In addition to the procedures described in the previous section, the CHSM also contains certain data structures and variables. The primary structures are:

- Driver parameter block
- Driver configuration table
- Driver statistics table

Chapter 3, "CHSM Data Structures and Variables" provides detailed descriptions of all the required CHSM data structures and variables.

CHSM Design Considerations

The following sections present the hardware and coding issues that must be considered when creating a CHSM.

Topology Issues

Before writing a CHSM, you must have a thorough understanding of the topology (such as Ethernet, Token-Ring, FDDI) that the driver and the hardware operate on. Refer to the specifications for the specific topology you will be using.

Hardware Issues

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

Network Interface Controllers

The MLID developer must be familiar with the network interface controller integrated circuit. Make every effort to obtain and use current data books and

2-4 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

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 CMSM and CTSM provide support procedures that are optimized for specific data transfer modes. CHSM packet reception and packet transmission routines, in particular, are affected by the adapter's transfer mode. To achieve the highest performance, you must select support procedures that are optimized for the specific data transfer mode. The data transfer modes are:

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

Bus Type

You must also consider the bus type and size when creating optimized CHSM operations. The initialization process can be affected by the bus type when it initializes and registers the hardware configuration with the CMSM and the LSL. Some bus types are:

- Industry Standard Architecture (ISA)
- Plug and Play ISA (PNP ISA)
- Micro Channel Architecture
- Extended Industry Standard Architecture (EISA)
- Peripheral Component Interconnect (PCI)
- PC Card (formally PCMCIA)

NetWare Environment Issues

ODI MLIDs operate as an integral part of the NetWare environment. Therefore, you must consider the following operating system characteristics when writing CHSM code.

Most of the code in the CHSM will run at privileged time (see definition below). Therefore, CHSM routines must not dominate system resources. If the code is optimized, normal execution will not be a problem. Care should be taken so that the handling of board error conditions, initialization, and shutdown adequately allows other processes to access system resources.

Interrupt Service Routine

The MLID's interrupt service routine needs only to service the adapter and return.

Note



In order to achieve operating system and platform independence, the CHSM must not enable or disable system interrupts. If during **DriverISR** or **DriverSend**, the interrupts are enabled or disabled, the interrupts must be restored to their initial state of entry before returning. Enabling or disabling interrupts will preclude the MLID from working on some platforms.

Execution Times

The two principal execution times are:

- Process time
- Privileged time

As you write an MLID, you must be aware of whether a routine is called at process time or at privileged time. Table 2-1 shows when the operating system(OS) and the CMSM may call MLID routines. Which support routines the MLID can access depends on the time that the MLID routine is called.

The execution time restrictions for CTSM and CMSM support routines are documented in Chapter 6, "CTSM Functions" and Chapter 7, "CMSM Functions".

Process Time

The MLID may do the following operations at process time:

- Memory allocation
- File I/O (with some exceptions)

Privileged Time


When a privileged process calls a routine, the routine becomes privileged. Privileged time routines must be highly optimized and limit their execution time.

Routines may not do the following operations at privileged time:

- Allocate memory
- Attempt file I/O
- Suspend execution
- Call routines that suspend execution

Table 2-1
Execution Time of MLID Routines

| Called by the OS or CMSM at Process Time | Can be called by the OS or CMSM at Privileged Time | Can be called by the OS or CMSM at Privileged or Process Time |
|---|---|---|
| DriverInit | DriverAES | DriverMulticastChange |
| DriverRemove | DriverISR | DriverPromiscuousChange |
| DriverShutdown | DriverPoll | DriverSend |
| | DriverReset | |
| Note: Functions that can be called at either process or privileged time must adhere to the restrictions for privileged time functions. | | |

Important  You must also observe privileged time restrictions for routines that are called by by other MLID routines. For example, after a transmit complete interrupt, **DriverISR** typically calls **DriverSend** to transmit the next packet in the send queue. Since **DriverISR** executes at privileged time, **DriverSend** must also observe privileged time restrictions.

Code and Data Space

This section describes the organization of the code and data space for ODI MLIDs. Figure 2-1 illustrates the code and data space used for multiple adapters with multiple frame support.

The CMSM creates the frame data space that represents a logical board for each installed frame type. The CMSM also creates the adapter data space for each physical board. If an MLID is reentrant, all physical boards of the same type use a single adapter code space. (See the reentrancy section later in this chapter.)

Frame Data Space

The frame data space (see Figure 2-1) contains all the information needed to support a specific frame type, as well as the hardware configuration for the corresponding board. The MLID allocates a separate frame data space for each installed frame type, which represents a logical board.

Important



From the CHSM's point of view, the only thing in the frame data space is the configuration table; therefore, the frame data space is referred to as the configuration table in this specification. (See Chapter 3, "CHSM Data Structures and Variables" for details on the frame data space.)

Note



MLIDs must support all frame types for a particular topology. Because all CTSMs provide full multiple frame support, MLIDs developed with these modules are guaranteed to support all applicable frame types for the topology.

Adapter Data Space

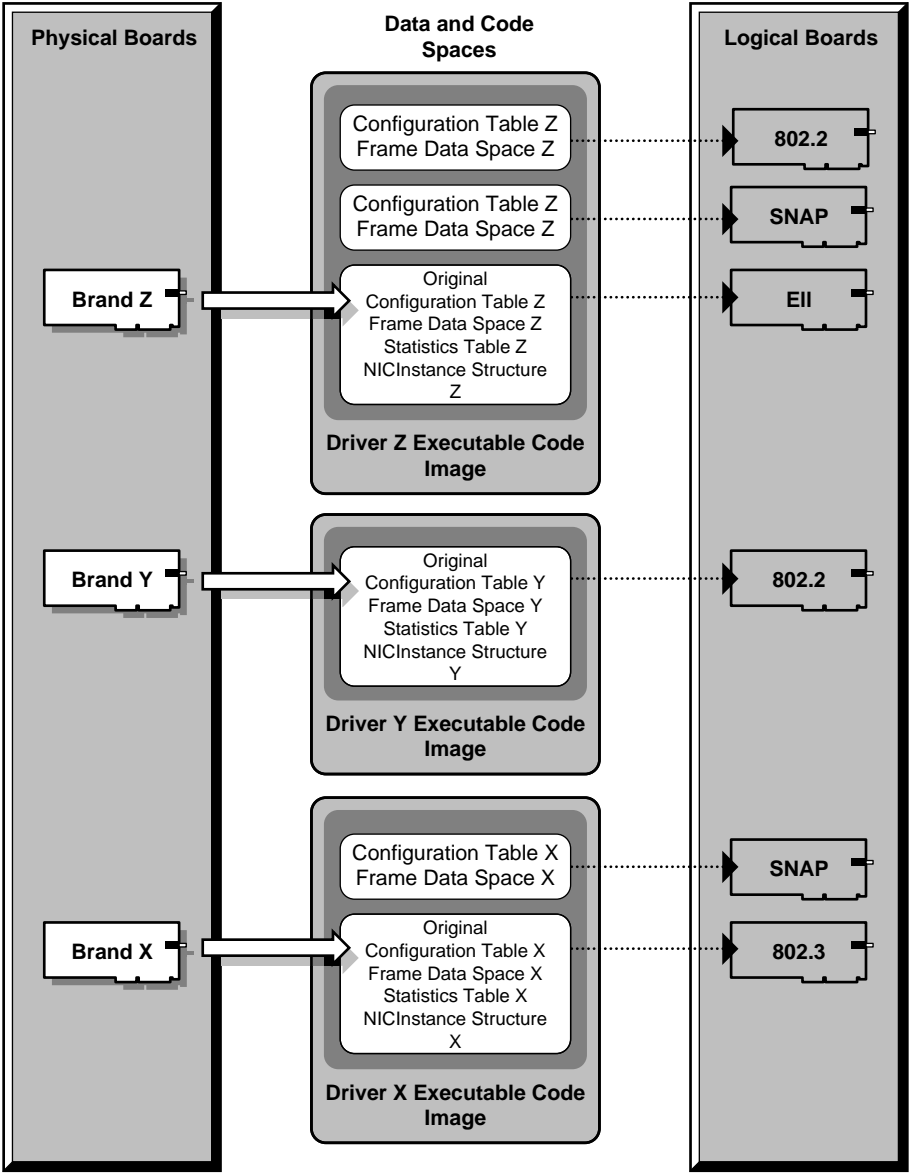
The adapter data space contains hardware information and statistical information that the MLID uses to drive and manage a specific physical board. The CHSM allocates only one adapter data space for each physical board, regardless of the number of frame types supported by that board. (See Chapter 3, "CHSM Data Structures and Variables" for details on the adapter data space.)

Adapter Code Space

When an adapter supports multiple frame types and/or multiple adapters of the same type, all logical boards associated with those adapters use a single code image of the MLID.

When multiple adapters of different types are loaded, a separate adapter code space is used.

Figure 2-1
Implementation of Multiple Frame Support



Special Support

Reentrancy

The organization of the code and data spaces, as described in the previous section, suggests that the CHSM is reentrant. Reentrancy, in this case, means that the MLID code is written to work with multiple logical boards and with multiple adapters of the same type.

The CHSM passes pointers to the appropriate frame data space and adapter data space when calling an MLID routine. References to structures and variables must be made using pointers and offsets rather than hardcoded values.

If an MLID is reentrant, the CHSM's linker definition file must include the reentrant keyword. This keyword allows a CHSM 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 MLID is loaded. If the reentrant keyword is not used, separate code and data spaces are allocated each time the MLID is loaded.

Multicast Addressing

If the adapter hardware is physically capable of supporting multicast addressing, the CHSM must implement multicast functionality, as described in the *DriverMulticastChange* section of Chapter 5, "CHSM Functions".

Promiscuous Mode

MLIDs that pass all packets being received by the adapter are said to have a promiscuous reception. Monitoring functions use this mode. We strongly recommend that the CHSM support promiscuous mode if the adapter is capable of it. The CHSM must enable or disable promiscuous reception on request as described in the *DriverPromiscuousChange* section of Chapter 5, "CHSM Functions".

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. *ODI Specification Supplement: The Hub*

Management Interface describes how to support management requests from these two agents in the CHSM.

Source Routing

An MLID can 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 SROUTE.NLM and CTSM modules handle this procedure with no interaction from the CHSM. For more on source routing, see *ODI Specification Supplement: Source Routing*

Router (Source Route Bridging)

A Token-Ring adapter capable of source route bridging mode (through a source route accelerator or address filter CAM) can have capabilities added to its CHSM to allow programs, such as the Multi Protocol Router Plus, to use it. For more information on Router, see *ODI Specification Supplement: Router Support*.

NESL Support

The NetWare Event Service Layer (NESL) handles event registration and modification. The NESL is designed around the concept of consumers and producers. Generally, a producer will produce events, which a consumer consumes. See Appendix E for a detailed description of NESL support.

chapter **3** *CHSM Data Structures and Variables*

Overview

This chapter describes the data structures and variables that the C HSM must define when an MLID is written using the support modules.

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

1. CMSM.NLM
2. <CTSM>.NLM
3. <C HSM>.LAN

<CTSM> represents a user-defined topology file name—for example, ETHERTSM. <C HSM> represents a user-defined name for the hardware specific module—for example, CNE2000.

Driver Parameter Block

When the CMSM and CTSM are installed, all public variables and functions are exported to the installer and are available to the NLMs. The C HSM can gain access to them by declaring them "extern" and including them in the import list in the linker definition file. The import list tells the linker which external variables and procedures the C HSM can access.

Since the C HSM is loaded last, it must make its public variables and functions available to the support modules by putting them in the Driver Parameter Block structure. The Driver Parameter Block structure contains the required C HSM public variables, tables, pointers, and driver management information. The Driver Parameter Block fields are accessed by external functions using the offsets defined in *cmsm.h*.

The C HSM's **DriverInit** routine must pass a pointer to the Driver Parameter Block when it calls <CTSM>**RegisterHSM** so that external procedures can access the Driver Parameter Block.

Driver Parameter Block Structure

```
typedef struct _DRIVER_PARM_BLOCK_
{
    UINT32          DriverParameterSize;
    C_HSM_STACK     *DriverInitParmPointer;
    MODULE_HANDLE   *DriverModuleHandle;
    void            *DPB_Reserved0;
    void            *DriverAdapterPointer;
    MLID_CONFIG_TABLE*DriverConfigTemplatePtr;
    UINT32          DriverFirmwareSize;
    void            *DriverFirmwareBuffer;
    UINT32          DPB_Reserved1;
    void            *DPB_Reserved2;
    void            *DPB_Reserved3;
    void            *DPB_Reserved4;
    UINT32          DriverAdapterDataSpaceSize;
    DRIVER_DATA     *DriverAdapterDataSpacePtr;
    UINT32          DriverStatisticsTableOffset;
    UINT32          DriverEndOfChainFlag;
    UINT32          DriverSendWantsECBs;
    UINT32          DriverMaxMulticast;
    UINT32          DriverNeedsBelow16Meg;
    void            *DPB_Reserved5;
    void            *DPB_Reserved6;
    void            (*DriverISRPtr) (DRIVER_DATA *);
    ODISTAT         (*DriverMulticastChangePtr)(DRIVER_DATA*,
        MLID_CONFIG_TABLE*, GROUP_ADDR_LIST_NODE*,
        UINT32, UINT32);
    void            (*DriverPollPtr) (DRIVER_DATA *, MLID_CONFIG_TABLE *);
    ODISTAT         (*DriverResetPtr) (DRIVER_DATA *,
        MLID_CONFIG_TABLE *, OPERATION_SCOPE);
    void            (*DriverSendPtr) (DRIVER_DATA *,
        MLID_CONFIG_TABLE *, TCB *, UINT32, void *);
    ODISTAT         (*DriverShutdownPtr) (DRIVER_DATA *,
        MLID_CONFIG_TABLE *, UINT32, OPERATION_SCOPE);
    void            (*DriverTxTimeoutPtr) (DRIVER_DATA *,
        MLID_CONFIG_TABLE *);
    ODISTAT         (*DriverPromiscuousChangePtr) (DRIVER_DATA *,
        MLID_CONFIG_TABLE *, UINT32);
    ODISTAT         (*DriverStatisticsChangePtr) (DRIVER_DATA *,
        MLID_CONFIG_TABLE *);
    ODISTAT         (*DriverRxLookAheadChangePtr) (DRIVER_DATA *,
```

3-2 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

```

        CONFIG_TABLE *);
    ODISTAT      (*DriverManagementPtr) (DRIVER_DATA *,
        CONFIG_TABLE *, ECB *);
    void          (*DriverEnableInterruptPtr) (DRIVER_DATA *);
    BOOLEAN      (*DriverDisableInterruptPtr) (DRIVER_DATA *, BOOLEAN);
    void          (*DriverISR2Ptr) (DRIVER_DATA *);
    MEON          ***DriverMessagesPtr;
    MEON_STRING   *HSMSpecVersionStringPtr;
    ODISTAT      (*DriverPriorityQueuePtr)
        (DRIVER_DATA*, MLID_CONFIG_TABLE *, ECB*);
    BOOLEAN      (*DriverDisableInterrupt2Ptr) (DRIVER_DATA*, BOOLEAN);
} DRIVER_PARM_BLOCK;

```

Table 3-1

Driver Parameter Block Field Descriptions

| Head | Type | Description |
|-----------------------|-----------------|--|
| DriverParameterSize | UINT32 | Set this variable to the size of the defined Driver Parameter Block structure before calling <CTSM>RegisterHSM . Since the Driver Parameter Block format is strictly defined and its size must remain constant, the CMSM uses this field to screen for invalid parameter blocks. <CTSM>RegisterHSM will fail if this value is incorrect. |
| DriverInitParmPointer | C HSM_STACK * | When the DriverInit routine is called, it passes certain information needed by the CMSM. DriverInit must set this field to point to the struct (C HSM_STACK), which contains information passed into it prior to calling <CTSM>RegisterHSM . |
| DriverModuleHandle | MODULE_HANDLE * | The CMSM sets this value when the DriverInit routine calls <CTSM>RegisterHSM . This handle is used to identify the NLM and is used by the operating system support routines to access and manage information about the NLM. The C HSM's DriverRemove routine needs this value when it calls CMSMDriverRemove . |
| DPB_Reserved0 | void * | This field is reserved and must be set to NULL. |

CHSM Data Structures and Variables 3-3

Table 3-1
Driver Parameter Block Field Descriptions *continued*

| Head | Type | Description |
|-----------------------------|---------------------|--|
| DriverAdapterPointer | void * | The CMSM sets this value when the DriverInit routine calls CMSMRegisterHardwareOptions . This field is reserved for use by the CMSM. |
| DriverConfigTemplatePtr | MLID_CONFIG_TABLE * | Set this variable to point to the MLID's configuration table template before calling <CTSM>RegisterHSM . (The configuration table is described later in this chapter.) |
| DriverFirmwareSize | UINT32 | See the "Driver Firmware" section later in this chapter. |
| DriverFirmwareBuffer | void * | See the "Driver Firmware" section later in this chapter. |
| DPB_Reserved1 | UINT32 | This field is reserved and must be set to 0. |
| DPB_Reserved2 | void * | This field is reserved and must be set to NULL. |
| DPB_Reserved3 | void * | This field is reserved and must be set to NULL. |
| DPB_Reserved4 | void * | This field is reserved and must be set to NULL. |
| DriverAdapterDataSpaceSize | UINT32 | Set this field to the size of the driver adapter data space template (described later in this chapter) before calling <CTSM>RegisterHSM . |
| DriverAdapterDataSpacePtr | DRIVER_DATA * | Set this field to point to the driver adapter data space template (described later in this chapter) before calling <CTSM>RegisterHSM . |
| DriverStatisticsTableOffset | UINT32 | Set this variable to the offset of the driver statistics table from the top of the driver adapter data space template before calling <CTSM>RegisterHSM . The statistics table and template are described later in this chapter. |

3-4 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

Table 3-1

Driver Parameter Block Field Descriptions *continued*

| Head | Type | Description |
|----------------------|--------|--|
| DriverEndOfChainFlag | UINT32 | Set this field to any nonzero value before calling CMSMRegisterHardwareOptions if the MLID supports shared interrupts and wants to be placed at the end of the chain. Set this field to 0 if the MLID wants to be at the beginning of the chain. This field is used only if the MS_SHARE_IRQ0_BIT bit is set in the <i>MLIDCFG_SharingFlags</i> field of the configuration table. |
| DriverSendWantsECBs | UINT32 | Before calling CMSMRegisterHardwareOptions , set this field to 1 if the DriverSend routine needs ECBs rather than TCBs. Intelligent bus master adapters that are designed to be ECB-aware use this field. (For more information on ECB-aware adapters, see Appendix B, "Event Control Blocks (ECBs)".) |
| DriverMaxMulticast | UINT32 | Before calling CMSMRegisterHardwareOptions , set this field to the maximum number of multicast addresses that the adapter can handle. ETHERTSM.NLM, TOKENTSM.NLM, and FDDITSM.NLM can accommodate an almost unlimited number of multicast addresses (limited only by available memory). If a C HSM can handle unlimited multicast addresses, set this field to -1. (Also see definitions for MF_SOFT_FILT_GRP_BIT and MF_GRP_ADDR_SUP_BIT in the configuration table <i>MLIDCFG_Flags</i> field later in this chapter.) |

Table 3-1
Driver Parameter Block Field Descriptions *continued*

| Head | Type | Description |
|--------------------------|-----------------|--|
| DriverNeedsBelow16Meg | UINT32 | <p>MLIDs for bus master adapters and DMA adapters must set this field to 1 before calling <CTSM>RegisterHSM if the adapter can only communicate with host memory below 16 MB. If the system already has more than 16 MB at the time the MLID loads, setting this field to 1 informs the CMSM to only allocate buffers, RCBs, TCBs, and ECBs below the 16MB boundary.</p> <p>If the MLID is loaded on a system that initially has less than 16 MB of memory, but will have more memory added later using the server's "REGISTER MEMORY" command, you must use the "BELOW16" keyword on the load command line to force the CMSM to allocate memory below 16 MB.</p> |
| DPB_Reserved5 | void * | This field is reserved and must be set to NULL. |
| DPB_Reserved6 | void * | This field is reserved and must be set to NULL. |
| DriverISRPtr | void (*fn)() | Set this field to point to the C HSM's DriverISR routine before calling CMSMSetHardwareInterrupt . If DriverPoll is used instead, set this field to NULL. |
| DriverMulticastChangePtr | ODISTAT (*fn)() | Set this field to point to the C HSM's DriverMulticastChange routine before calling CMSMRegisterHardwareOptions . If multicast addressing is not supported, set this field to NULL. |
| DriverPollPtr | void (*fn)() | Set this field to point to the C HSM's DriverPoll routine before calling CMSMEnablePolling . If this routine is not used, set this field to NULL. |

Table 3-1
Driver Parameter Block Field Descriptions *continued*

| Head | Type | Description |
|----------------------------|-----------------|--|
| DriverResetPtr | ODISTAT (*fn)() | Set this field to point to the C HSM's DriverReset routine before calling CMSMRegisterHardwareOptions . |
| DriverSendPtr | void (*fn)() | Set this field to point to the C HSM's DriverSend routine before calling CMSMRegisterHardwareOptions . |
| DriverShutdownPtr | ODISTAT (*fn)() | Set this field to point to the C HSM's DriverShutdown routine before calling CMSMRegisterHardwareOptions . |
| DriverTxTimeoutPtr | void (*fn)() | If the HSM must access a hardware device when the TSM has detected a transmit time-out, set this field to a pointer to the C HSM DriverTx Time-out routine before Options. Most C HSMs set this field to NULL. |
| DriverPromiscuousChangePtr | void (*fn)() | Set this field to point to the C HSM's DriverPromiscuousChange routine before calling CMSMRegisterHardwareOptions . If promiscuous mode is not supported, set this field to NULL. |
| DriverStatisticsChangePtr | ODISTAT (*fn)() | <p>Pointer to a DriverStatisticsChange routine that is called whenever the CMSM's control procedure function 1 (get MLID statistics) is invoked. The DriverStatisticsChange routine allows C HSMs with intelligent adapters that keep track of statistics to update the statistics table only as needed.</p> <p>If the adapter supports this feature, this field must be set before calling CMSMRegisterHardwareOptions. If not used, set this field to NULL.</p> |

Table 3-1
Driver Parameter Block Field Descriptions *continued*

| Head | Type | Description |
|----------------------------|-----------------|--|
| DriverRxLookAheadChangePtr | ODISTAT (*fn)() | <p>Pointer to a DriverRxLookAheadChange routine that is called whenever the CMSM's control procedure function 9 (set look-ahead size) is invoked. This routine allows C HSMs with intelligent adapters to be informed when they receive <i>MLIDCFG_LookAheadSize</i> field in the configuration table and the CMSMMaxFrameHeaderSize variable changes rather than constantly checking.</p> <p>If the adapter supports this feature, this field must be set before calling CMSMRegisterHardwareOptions. If not used, set this field to NULL.</p> |
| DriverManagementPtr | ODISTAT (*fn)() | <p>If an MLID accepts management requests from outside NLMs (HMI or CSL), this field contains a pointer to the DriverManagement routine that is called whenever the CMSM control procedure management function is called. (See <i>ODI Supplement: Hub Management Interface</i> for more information.)</p> <p>If used, this field must be set before calling CMSMRegisterHardwareOptions. If not supported, set this field to NULL.</p> |
| DriverEnableInterruptPtr | void (*fn)() | <p>Pointer to a DriverEnableInterrupt routine that is called by the CMSM to enable interrupts at the adapter.</p> |
| DriverDisableInterruptPtr | BOOLEAN (*fn)() | <p>Pointer to a DriverDisableInterrupt routine that is called by the CMSM to disable interrupts at the adapter.</p> |
| DriverISR2Ptr | void (*fn)() | <p>If the C HSM uses a secondary interrupt, this value is a pointer to the C HSM's interrupt service routine for the second interrupt. If the C HSM does not use two interrupts, set this pointer to NULL.</p> |

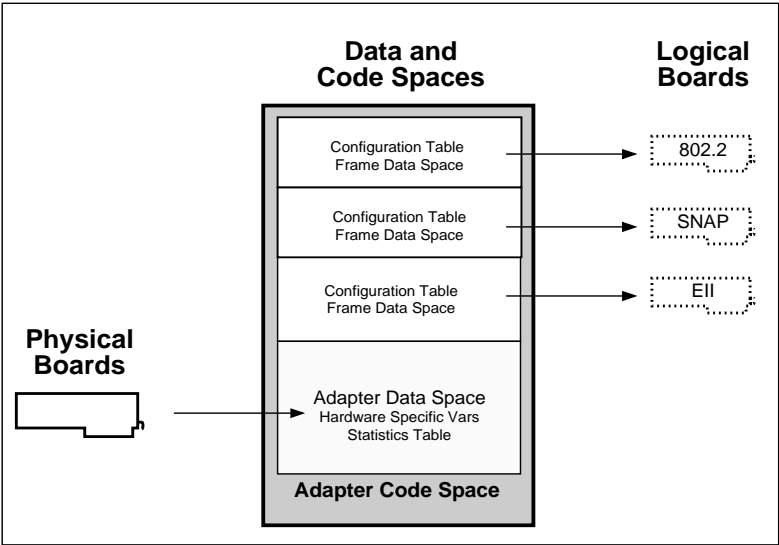
Table 3-1
Driver Parameter Block Field Descriptions *continued*

| Head | Type | Description |
|----------------------------|-----------------|--|
| DriverMessagesPtr | MEON *** | Pointer to a pointer to an array of MEON string pointers that is defined as <code>**MEON_STRING DriverMessages</code> , which is filled in by the C HSM prior to calling <CTSM>RegisterHSM . <i>DriverMessagesPtr</i> is used by the message enabling macros for handling messages. (For more information on the message handling macros, see Appendix A, "Language Enabling".) |
| HSMSpecVersionStringPtr | MEON_STRING* | Pointer to the version string that describes the version of the HSM specification that the HSM is written to. The string is defined by Novell as "HSM_CSPEC_VERSION: 1.11". |
| DriverPriorityQueuePtr | ODISTAT | Pointer to the DriverPriorityQueueSupport function called by the CTSM to handle HSM priority packets when the normal send path is congested. If the C HSM/adaptor supports this feature, this field must be set before calling CMSMRegisterHardwareOptions . If not used, set this field to NULL. |
| DriverDisableInterrupt2Ptr | BOOLEAN (*fn)() | Pointer to a DriverDisableInterrupt2 routine that is called by the CMSM to disable the interrupts. This routine is required if the C HSM supports a secondary interrupt, such as when DriverISR2Ptr is not set to NULL or when the <i>MLIDCFG_Interrupt1</i> field in the configuration table is not set to <code>UNUSED_INTERRUPT</code> . If the C HSM does not support a secondary interrupt, set this field to NULL. |

Frame Data Space

When **DriverInit** calls **<CTSM>RegisterHSM**, the CMSM allocates a frame data space and creates a copy of a configuration table template in it. A separate frame data space containing a separate configuration table template is created for each installed frame type (illustrated in Figure 3.1). The CMSM and the CTSM both pass pointers to the appropriate frame data space when they call C HSM procedures. Configuration tables can also be accessed through the *CMSMVirtualBoardLink* array.

Figure 3-1
Frame and Adapter Data Space



Configuration Table

The configuration table is a structure defined by the ODI specification. It contains configuration information about the MLID and the adapter. The C HSM must provide the template for initializing the fields of the configuration table. The CMSM then creates a copy of this template in a separate frame data space for each installed frame type.

The configuration table fields are used primarily during initialization to reserve hardware resources. Fields that can be modified when the MLID is installed are

set to their default values before the driver parameters are parsed. Fields that cannot be modified when the MLID is installed are set to values that mark them as invalid (usually 0 or -1, depending on the field).

CMSMParseDriverParameters collects information from the install command line and the operator console. Once the configuration table fields are entered, the MLID uses **CMSMRegisterHardwareOptions** to reserve the hardware resources.

Driver Configuration Table Template

```
typedef struct _MLID_CONFIG_TABLE_
{
    MEON            MLIDCFG_Signature[26];
    UINT8           MLIDCFG_MajorVersion;
    UINT8           MLIDCFG_MinorVersion;
    NODE_ADDR       MLIDCFG_NodeAddress;
    UINT16          MLIDCFG_ModeFlags;
    UINT16          MLIDCFG_BoardNumber;
    UINT16          MLIDCFG_BoardInstance;
    UINT32          MLIDCFG_MaxFrameSize;
    UINT32          MLIDCFG_BestDataSize;
    UINT32          MLIDCFG_WorstDataSize;
    MEON_STRING     *MLIDCFG_CardName;
    MEON_STRING     *MLIDCFG_ShortName;
    MEON_STRING     *MLIDCFG_FrameTypeString;
    UINT16          MLIDCFG_Reserved0;
    UINT16          MLIDCFG_FrameID;
    UINT16          MLIDCFG_TransportTime;
    UINT32          (*MLIDCFG_SourceRouting)
                    (UINT32, void*, void**,boolean)
    UINT16          MLIDCFG_LineSpeed;
    UINT16          MLIDCFG_LookAheadSize;
    UINT8           MLIDCFG_SGCount;
    UINT8           MLIDCFG_Reserved1;
    UINT16          MLIDCFG_PrioritySup;
    void            *MLIDCFG_Reserved2;
    UINT8           MLIDCFG_DriverMajorVer;
    UINT8           MLIDCFG_DriverMinorVer;
    UINT16          MLIDCFG_Flags;
    UINT16          MLIDCFG_SendRetries;
    void            *MLIDCFG_DriverLink;
    UINT16          MLIDCFG_SharingFlags;
    UINT16          MLIDCFG_Slot;
    UINT16          MLIDCFG_IOPort0;
    UINT16          MLIDCFG_IORange0;
}
```

```
    UINT16      MLIDCFG_IOPort1;
    UINT16      MLIDCFG_IORange1;
    void        *MLIDCFG_MemoryAddress0;
    UINT16      MLIDCFG_MemorySize0;
    void        *MLIDCFG_MemoryAddress1;
    UINT16      MLIDCFG_MemorySize1;
    UINT8       MLIDCFG_Interrupt0;
    UINT8       MLIDCFG_Interrupt1;
    UINT8       MLIDCFG_DMALine0;
    UINT8       MLIDCFG_DMALine1;
    void        *MLIDCFG_ResourceTag;
    void        *MLIDCFG_Config;
    void        *MLIDCFG_CommandString;
    MEON_STRING MLIDCFG_LogicalName[18];
    void        *MLIDCFG_LinearMemory0;
    void        *MLIDCFG_LinearMemory1;
    UINT16      MLIDCFG_ChannelNumber;
    void        *MLIDCFG_DBusTag;
    UINT8       MLIDCFG_DIOConfigMajorVer;
    UINT8       MLIDCFG_DIOConfigMinorVer;
} MLID_CONFIG_TABLE;
```

Table 3-2
Driver Configuration Table Field Descriptions

| Name | Type | Description |
|----------------------|-----------|---|
| MLIDCFG_Signature | MEON [26] | String that indicates the beginning of the configuration table. The string is "HardwareDriverMLID" followed by exactly eight spaces. (Required) |
| MLIDCFG_MajorVersion | UINT8 | The major version number of the configuration table. The current major version number is 1. (Required) |
| MLIDCFG_MinorVersion | UINT8 | The minor version number of the configuration table. The current minor version number is 21. (Required) |

3-12 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

Table 3-2

Driver Configuration Table Field Descriptions *continued*

| Name | Type | Description |
|---------------------|-----------|---|
| MLIDCFG_NodeAddress | NODE_ADDR | <p>When DriverInit calls <CTSM>RegisterHSM, the CMSM fills these bytes with 0xFF, then checks the command line for a node address override. If an override address is found, the CMSM places the physical layer format of the address in this field.</p> <p>After the MLID calls CMSMRegisterHardwareOptions, it must check this field for an override.</p> <p>If these bytes are not all 0xFF, an override occurred and the C HSM sets the physical board's address to the value in this field. If there is not an override, the C HSM must place the node address that was read from the hardware into this field.</p> <p>On Token-Ring adapters, if the MLID has been loaded as LSB, the CMSM will change <i>MLIDCFG_NodeAddress</i> to a canonical address.</p> <p>After the C HSM calls CMSMRegisterMLID, the CTSM places the physical layer format of the node address into the <i>CMSMPhysNodeAddress</i> variable and sets the appropriate <i>MLIDCFG_ModeFlag</i> bits. This physical address can be in canonical or noncanonical form. (For more information, refer to <i>MLIDCFG_ModeFlags</i>, <i>CMSMPhysNodeAddress</i>, and <i>ODI Specification Supplement: Canonical and Noncanonical Addressing</i>.)</p> |
| MLIDCFG_ModeFlags | UINT16 | See <i>MLIDCFG_ModeFlags</i> field description (Table 3.3). |
| MLIDCFG_BoardNumber | UINT16 | The CMSM sets this field to the board number assigned by the LSL when DriverInit calls CMSMRegisterMLID . |

Table 3-2
Driver Configuration Table Field Descriptions *continued*

| Name | Type | Description |
|-----------------------|--------|---|
| MLIDCFG_BoardInstance | UINT16 | <p>The CMSM sets this field when the DriverInit routine calls CMSMRegisterHardwareOptions. If the C HSM is driving two adapters, all logical boards associated with the first adapter will have a value of 1 and all the logical boards associated with the second adapter will have a value of 2.</p> <p>Note: Each controller on a multichannel adapter is treated as a separate adapter.</p> |
| MLIDCFG_MaxFrameSize | UINT32 | <p>Largest possible packet size that can be transmitted or received by the C HSM. This value includes all headers. The C HSM sets this value.</p> <p>The CMSM can reduce this value, depending on platform specifics, when the C HSM's DriverInit routine calls <CTSM>RegisterHSM.</p> <p>The C HSM can reduce this value prior to calling CMSMRegisterMLID.</p> <p>Ethernet C HSMs set this field to 1514.</p> <p>FDDI C HSMs set this value to 4491.</p> <p>Token-Ring C HSMs must not set this value greater than the maximum size supported by the Token-Ring board configuration (4Mbit Token-Ring sets this field to 4464; 16Mbit Token-Ring may set this field to 17954. See Table 3-6 for more detail.</p> |
| MLIDCFG_BestDataSize | UINT32 | <p>The CTSM sets this field during execution of CMSMRegisterMLID. The CTSM subtracts the length of the smallest media header(s) from the value in the <i>MLIDCFG_MaxFrameSize</i> field.</p> <p>For example, an Ethernet_II MLID sets this field to 1500 decimal (1514 - 14 [MAC] = 1500). A Token-Ring MLID sets this field to <i>MLIDCFG_MaxFrameSize</i> - 14 [MAC] - 3 [802.2 UI].</p> |

Table 3-2

Driver Configuration Table Field Descriptions *continued*

| Name | Type | Description |
|-------------------------|-------------|---|
| MLIDCFG_WorstDataSize | UINT32 | <p>The CTSM sets this field during execution of CMSMRegisterMLID. The CMSM subtracts the length of the largest media header(s) from the <i>MLIDCFG_MaxFrameSize</i> field.</p> <p>For example, a Token-Ring MLID sets this field to <i>MLIDCFG_MaxFrameSize</i> - 14 [MAC] - 30 [source routing] - 4 [802.2 SI]. An Ethernet_II MLID sets this field to 1500:</p> <pre>MLIDCFG - MAXFRAME SIZE (1514) - 14 [MAC] = 1500</pre> |
| MLIDCFG_CardName | MEON_STRING | <p>* The C HSM may set this field to point to a NULL-terminated, MEON string that is identical to the description string in the linker definition file (see <i>Appendix C: Platform Specific Information</i>).</p> <p>For example: "Novell Ethernet NE2000", 0</p> <p>If this field is initialized to NULL, the CMSM will extract the description string from the NLM header (derived from the linker definition file) when the C HSM's DriverInit routine calls <CTSM>RegisterHSM. This way, only one description string must be maintained.</p> |
| MLIDCFG_ShortName | MEON_STRING | <p>* The C HSM must set this field to point to a NULL-terminated, MEON string that describes the adapter in eight bytes or less.</p> <p>For example: "NE2000", 0</p> <p>The string is usually the name of the <C HSM>.LAN file.</p> |
| MLIDCFG_FrameTypeString | MEON_STRING | <p>* This field holds a pointer to a NULL-terminated, MEON string that describes the frame and media type being used by this MLID. (See <i>ODI Specification Supplement: Frame Types and Protocol IDs</i> for possible frame types.) The CMSM sets this field.</p> |
| MLIDCFG_Reserved0 | UINT16 | <p>This field is reserved for future use and must be set to 0.</p> |

Table 3-2
Driver Configuration Table Field Descriptions *continued*

| Name | Type | Description |
|-----------------------|----------------|--|
| MLIDCFG_FrameID | UINT16 | <p>The CMSM sets this field when the DriverInit routine calls CMSMRegisterHardwareOptions. It contains the frame type ID number.</p> <p>For more information on frame types, see <i>ODI Supplement: Frame Types and Protocol IDs</i>.</p> |
| MLIDCFG_TransportTime | UINT16 | <p>Number of milliseconds it takes the adapter to transmit a 586-byte packet. Most C HSMs set this field to 1. This field cannot be set to 0.</p> |
| MLIDCFG_SourceRouting | UINT32 (*fn()) | <p>Pointer to a source routing module, such as <i>SROUTE.NLM</i>, used by a Token-Ring or FDDI MLID.</p> <p>CTSMs that do not use source routing set this field to NULL and do not modify it.</p> <p>See the <i>ODI Supplement: Source Routing</i> for a discussion of dynamic source routing.</p> |
| MLIDCFG_LineSpeed | UINT16 | <p>The speed of the topology; set by the C HSM. 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, if the speed of the line MLID is 10 Mbps (Ethernet for example) put 10 (decimal) in this field.</p> |
| MLIDCFG_LookAheadSize | UINT16 | <p>The amount of data required by a protocol stack when previewing received packets; the default is 18 bytes. The CTSM sets this variable. The variable, <i>CMSMMaxFrameHeaderSize</i> (see <i>Chapter 4: CMSM/CTSM Data Structures and Variables</i>), is equal to this value plus the maximum media header size. This size can be dynamically changed; its maximum value is 128 bytes.</p> |

Table 3-2
Driver Configuration Table Field Descriptions *continued*

| Name | Type | Description |
|------------------------|--------|---|
| MLIDCFG_SGCount | UINT8 | The maximum number of scatter/gather elements the adapter is capable of handling. The C HSM sets this variable. This field is only valid if the <i>MM_FRAGS_PHYS_BIT</i> bit in the <i>MLIDCFG_ModeFlags</i> field is set. The minimum value is 2 (1 for the MAC header and 1 for data). The maximum value is 17 (1 for the MAC header and 16 for data). |
| MLIDCFG_Reserved1 | UINT8 | Reserved; must be set to 0. |
| MLIDCFG_PrioritySup | UINT16 | The number of priority levels that the C HSM can handle. This field has a maximum of 7 priorities (1-7). Zero indicates no priority packet support. Therefore, the C HSM can set this field to a value of 0 through 7. |
| MLIDCFG_Reserved2 | void * | Reserved; must be set to 0. |
| MLIDCFG_DriverMajorVer | UINT8 | The current revision level of the C HSM; matches the revision level displayed by the C HSM. The C HSM sets this variable. For example, if the C HSM's current major version is 2, this field's value is 2. If this field is initialized to NULL, the CMSM extracts the major version from the NLM header, which is derived from the linker definition file when the C HSM's DriverInit function calls <CTSM>RegisterHSM . This ensures that only one version must be maintained. |
| MLIDCFG_DriverMinorVer | UINT8 | The current revision level of the C HSM; matches the revision level displayed by the C HSM. The C HSM sets this variable. For example, if the C HSM's current minor version is .32, this field's value is 32. If this field is initialized to NULL, the CMSM extracts the minor version from the NLM header, which is derived from the linker definition file when the C HSM's DriverInit function calls <CTSM>RegisterHSM . This ensures that only one version must be maintained. |
| MLIDCFG_Flags | UINT16 | See the <i>MLIDCFG_Flags</i> field description (Table 3.4). |

Table 3-2
Driver Configuration Table Field Descriptions *continued*

| Name | Type | Description |
|----------------------|--------|--|
| MLIDCFG_SendRetries | UINT16 | The number of times the C HSM tries to resend a packet before aborting the transmission. This count may be overwritten at load time. The C HSM sets this variable. (See RETRIES in the "Load Keywords and Parameters Descriptions" table in Appendix C, "Platform Specific Information".) |
| MLIDCFG_DriverLink | void * | NULL; not modified by the C HSM. |
| MLIDCFG_SharingFlags | UINT16 | The C HSM sets this variable. See the <i>MLIDCFG_SharingFlags</i> field description (Table 3.5). |
| MLIDCFG_Slot | UINT16 | <p>For Micro Channel, EISA, PCI, PC Card, and other buses which allow for the identification of the location of an adapter, this field contains the Hardware Instance Number (HIN). The HIN is a system-wide, unique handle for a device, which is returned by CMSMGetInstanceNumber after calling CMSMSearchAdapter. This value normally corresponds to the number silk-screened on the motherboard or stamped on the chassis of the computer. The instances are assigned a unique value in the following cases:</p> <ul style="list-style-type: none">Integrated motherboard devicesPCI BIOS v2.0 devicesPCI BIOS v2.1 adapters with multiple devices or functionsPnP ISA devicesConflicts between physical slot numbers <p>If this field is not used, it must be set to UNUSED_SLOT.</p> |
| MLIDCFG_IOPort0 | UINT16 | <p>Primary base I/O port. This field is initialized to the adapter's default base I/O port. If this field is not used, it is set to UNUSED_IO_PORT. The C HSM sets this variable, but it may be changed during a call to CMSMParseDriverParameters.</p> |

Table 3-2
Driver Configuration Table Field Descriptions *continued*

| Name | Type | Description |
|------------------------|--------|--|
| MLIDCFG_IORange0 | UINT16 | Number of UINT8 I/O ports starting at <i>MLIDCFG_IOPort0</i> . If this field is not used, it is set to <i>UNUSED_IO_RANGE</i> . The C HSM sets this variable, but it may be changed during a call to CMSMParseDriverParameters . |
| MLIDCFG_IOPort1 | UINT16 | Secondary base I/O port. This field is initialized to the adapter's default base I/O port. If this field is not used, it is set to <i>UNUSED_IO_PORT</i> . The C HSM sets this variable, but it may be changed during a call to CMSMParseDriverParameters . |
| MLIDCFG_IORange1 | UINT16 | Number of UINT8 I/O ports starting at <i>MLIDCFG_IOPort1</i> . If this field is not used, it is set to <i>UNUSED_IO_RANGE</i> . The C HSM sets this variable, but it may be changed during a call to CMSMParseDriverParameters . |
| MLIDCFG_MemoryAddress0 | void * | This field is initialized to the adapter's default base memory address. If the adapter does not use, or define, shared RAM or ROM, set this field to <i>UNUSED_MEMORY_ADDRESS</i> . This value is an absolute physical address. On Intel processors, for example, if a physical adapter's RAM is located at C000:0, the value in this field will be C0000. The C HSM sets this variable, but it may be changed during a call to CMSMParseDriverParameters . |
| MLIDCFG_MemorySize0 | UINT16 | If <i>MS_MEM_PAGE_BIT</i> in <i>MLIDCFG_SharingFlags</i> is set, this field defines the number of pages of memory decoded at <i>MLIDCFG_MemoryAddress0</i> . If <i>MS_MEM_PAGE_BIT</i> in <i>MLIDCFG_SharingFlags</i> is clear, this field defines the number of paragraphs (16 bytes) of memory decoded at <i>MLIDCFG_MemoryAddress0</i> . If <i>MLIDCFG_MemoryAddress0</i> is not defined, set this field to <i>UNUSED_MEMORY_SIZE</i> . Note: The size of a page of memory is determined by the processor for which this code is compiled on, such as Intel 4K, PowerPC 4K, Alpha 8K. |

Table 3-2
Driver Configuration Table Field Descriptions *continued*

| Name | Type | Description |
|------------------------|--------|---|
| MLIDCFG_MemoryAddress1 | void * | This field allows the MLID to define a second memory address range for use by the MLID's adapter. For example, <i>MLIDCFG_MemoryAddress0</i> could define the starting address of the adapter's RAM, and this field could define the starting address of the adapter's ROM. Set this field to UNUSED_MEMORY_ADDRESS if the adapter does not define a second memory range. The C HSM sets this variable, but it may be changed during a call to CMSMParseDriverParameters . |
| MLIDCFG_MemorySize1 | UINT16 | If MS_MEM_PAGE_BIT in <i>MLIDCGFG_SharingFlags</i> is set, this field defines the number of pages of memory decoded at <i>MLIDCFG_MemoryAddress1</i> . If MS_MEM_PAGE_BIT in <i>MLIDCFG_SharingFlags</i> is clear, this field defines the number of paragraphs (16 bytes) of memory decoded at <i>MLIDCFG_MemoryAddress1</i> . If <i>MLIDCFG_MemoryAddress1</i> is not defined, set this field to UNUSED_MEMORY_SIZE . Note: The size of a page of memory is determined by the processor for which this code is compiled on, such as Intel 4K, PowerPC 4K, Alpha 8K. |
| MLIDCFG_Interrupt0 | UINT8 | The adapter's default base IRQ number. If the adapter does not use an interrupt line, set this field to UNUSED_INTERRUPT . If the MLID's adapter supports IRQ 2 or 9, the MLID sets the value to be consistent with the adapter's documentation. The C HSM sets this variable, but it may be changed during a call to CMSMParseDriverParameters . For example, if the adapter's documentation specifies the default jumper setting as IRQ2, set this field to 2. If the default jumper setting is IRQ9, set this field to 9. |
| MLIDCFG_Interrupt1 | UINT8 | The adapter's second IRQ number. Set this field to UNUSED_INTERRUPT if unused. The C HSM sets this variable, but it may be changed during a call to CMSMParseDriverParameters . |

Table 3-2
Driver Configuration Table Field Descriptions *continued*

| Name | Type | Description |
|-------------------------|-------------|--|
| MLIDCFG_DMALine0 | UINT8 | Initialized to the adapter's default DMA channel number. If the adapter does not use a DMA channel, set this field to <code>UNUSED_DMA_LINE</code> (unused). The C HSM sets this variable, but it may be changed during a call to CMSMParseDriverParameters . |
| MLIDCFG_DMALine1 | UINT8 | Used by the MLID if the MLID's adapter uses a second DMA channel. Set this field to <code>UNUSED_DMA_LINE</code> if it is not needed. The C HSM sets this variable, but it may be changed during a call to CMSMParseDriverParameters . |
| MLIDCFG_ResourceTag | void * | Pointer to a resource tag; set by the CMSM. |
| MLIDCFG_Config | void * | Pointer to the LSL's copy of the configuration table. C HSMs do not use this field. |
| MLIDCFG_CommandString | void * | This field is set by the C MSM to point to a structure containing two fields. The first field is a forward link to the next structure if any. The second field is a pointer to a null terminated string containing the parameters entered on the command line. Normally, there is only one node in the linked list, but if there are more than one, the command line is the concatenation of all the nodes. Bits 9 and 10 of the <i>MLIDSharingFlags</i> fields are used in conjunction with this field. |
| MLIDCFG_LogicalName[18] | MEON_STRING | NULL terminated logical name of the MLID if a name exists. C HSMs do not use this field. |
| MLIDCFG_LinearMemory0 | void * | The address in <i>MLIDCFG_MemoryAddress0</i> is converted into the correct address for the C HSM and is stored in this field when CMSMRegisterHardwareOptions is called. Due to address mapping on platforms with varying processors or multiple buses, the address in <i>MLIDCFG_MemoryAddress0</i> might not work for the C HSM. C HSMs must always use the address in this field to access an adapter's memory. |

Table 3-2
Driver Configuration Table Field Descriptions *continued*

| Name | Type | Description |
|---------------------------|--------|---|
| MLIDCFG_LinearMemory1 | void * | The address in <i>MLIDCFG_MemoryAddress1</i> is converted into the correct address for the C HSM and is stored in this field when CMSMRegisterHardwareOptions is called. Due to address mapping on platforms with varying processors or multiple buses, the address in <i>MLIDCFG_MemoryAddress1</i> might not work for the C HSM. C HSMs must always use the address in this field to access an adapter's memory. |
| MLIDCFG_ChannelNumber | UINT16 | The channel number of the LAN adapter--used with multichannel adapters only. The channel number can be specified when the MLID is installed, using the "channel=#" keyword (where # is any value greater than 0). Set this field to 0 if multichannel adapters are not used. The C HSM sets this variable, but it may be changed during a call to CMSMParseDriverParameters . |
| MLIDCFG_DBusTag | void * | Pointer to an architechure-dependent value, which specifies the bus on which the adapter is found. The value placed in this field is returned by CMSMSearchAdapter unless the board is Legacy ISA, in which case it is set to zero. This field must be set before calling CMSMRegisterHardwareOptions . |
| MLIDCFG_DIOConfigMajorVer | UINT8 | The current major revision level of the IO_CONFIG structure (the bottom half of MLID_CONFIG_TABLE structure). The CMSM sets this variable to 1. |
| MLIDCFG_DIOConfigMinorVer | UINT8 | The current minor revision level of the IO_CONFIG structure (the bottom half of MLID_CONFIG_TABLE structure). The CMSM sets this variable to 0. |

MLIDCFG_ModeFlags Field

This section describes the bits of the *MLIDCFG_ModeFlags* field in the configuration table. Figure 3-2 shows the reserved bits and their values for this field.

Figure 3-2
MLIDCFG_ModeFlags Field Default Values

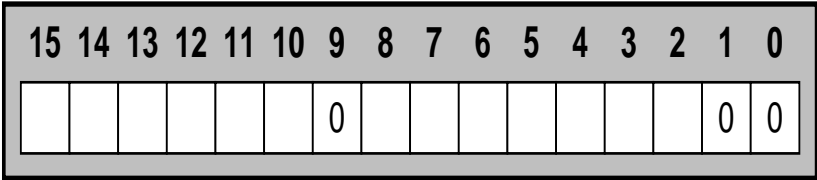


Table 3-3 describes the non-reserved bit values for the *MLIDCFG_ModeFlags* field.

Table 3-3
MLIDCFG_ModeFlags Bits Description

| Bit # | Name | Description |
|-------|-------------------|---|
| 2 | MM_DEPENDABLE_BIT | This bit has been rendered obsolete by the NetWare Link Services Protocol. We recommend that this bit always be set to 0. Previously, this bit was used to limit the frequency of IPX RIP/SAP updates when operating over reliable delivery, low bandwidth, Wide Area Network (WAN) data links. When set to 1 by a WAN C HSM, this bit caused IPX to suppress the normal, periodic, RIP/SAP updates, unless the route or service databases had changed. However, use of this bit to suppress updates sometimes resulted in IPX route or service loss. |
| 3 | MM_MULTICAST_BIT | The C HSM sets this bit if it supports multicast addressing. Multicast support is required for all media that have multicast capability. |

Table 3-3
MLIDCFG_ModeFlags Bits Description *continued*

| Bit # | Name | Description |
|-------|------------------------|--|
| 4 | MM_CSL_COMPLIANT_BIT | The C HSM sets this bit if the supported data link protocol requires connection management through the Call Support Layer (CSL) interface. Typical Wide Area Network (WAN) data link protocols, such as Frame Relay, PPP, and X.25, are connection oriented and rely upon network layer protocol (IPX, IP) interaction to establish, maintain, and terminate connections to remote peers. The CSL provides extensions to ODI that allow this connection management interaction between network and data link layer protocols. This bit must not be set by connectionless data link protocols, such as Token-Ring and Ethernet. For more information on the CSL and WAN C HSM interfaces, see <i>NetWare Wide Area Network Open Data-Link Interface Specification</i> . |
| 5 | MM_PREFILLED_ECB_BIT | Set this bit if the MLID always supplies prefilled LSL ECBs in the <i>LkAhd_PrefilledECB</i> field of the LOOKAHEAD structure. |
| 6 | MM_RAW_SENDS_BIT | The CTSM sets this bit to indicate that raw sends are supported. Refer to the TCB section of <i>Chapter 4: CSM/CTSM Data Structures and Variables</i> for more information on raw sends. |
| 7 | MM_DATA_SZ_UNKNOWN_BIT | Set this bit if the C HSM is capable of setting the <i>LkAhd_FrameDataSize</i> field of the LOOKAHEAD structure to a -1 (frame size and/or receive status unknown)—for example, pipelined LAN adapter. |
| 8 | MM_SMP_BIT | Set by the CSM if the CSM and CTSM support symmetrical multiprocessing (SMP). |
| 10 | MM_FRAG_RECEIVES_BIT | The C HSM must set this bit if it can handle fragmented RCBs. (RCBs are described in Chapter 4, "CSM/CTSM Data Structures and Variables".) |
| 11 | MM_C_HSM_BIT | This bit distinguishes an HSM written to this specification from one written to the assembly language specification. If this bit is set, the HSM is written to this specification. If this bit is clear, the HSM is written to the assembly language specification. |

3-24 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

Table 3-3
MLIDCFG_ModeFlags Bits Description *continued*

| Bit # | Name | Description |
|-------|--------------------|--|
| 12 | MM_FRAGS_PHYS_BIT | <p>The C HSM sets this bit if it expects the following from the CTSM:</p> <ol style="list-style-type: none">1) For TCBs, fragment pointers will all contain physical addresses pointing to locked, contiguous buffers.2) For ECB-aware adapters and for send ECBs, pointers to the ECB can be converted to a physical address and return physical and logical addresses to the ECB.3) <CTSM>ProcessGetRCB will return an RCB with locked, contiguous, physical addresses in the fragment pointer. <p>For more information on using this bit, see "Adapters that Need Physical Addresses" in Chapter 5, "C HSM Functions".</p> |
| 13 | MM_PROMISCUOUS_BIT | <p>The C HSM must set this bit if it supports promiscuous mode.</p> |

Table 3-3
MLIDCFG_ModeFlags Bits Description *continued*

| Bit # | Name | Description |
|-------|------------------------|---|
| 14 | MM_NONCANONICAL_BIT | These bits indicate whether the <i>MLIDCFG_NodeAddress</i> field of the configuration table contains a canonical or noncanonical address. The CMSM controls these bits. Bit 15 indicates whether the node address format can be configured. If this bit is set, the format can be configured and the C HSM uses the CMSMPhysNodeAddress variable instead of the configuration table <i>MLIDCFG_NodeAddress</i> to obtain the physical layer node address. (For NetWare versions later than 3.11, the CMSM always sets bit 15.) Bit 14 indicates whether <i>MLIDCFG_NodeAddress</i> contains the canonical or noncanonical form of the node address. The state of bit 14 is only defined when bit 15 is set. The bit 15 and 14 combinations are: 00 <i>MLIDCFG_NodeAddress</i> format is unspecified. The node address is assumed to be in the physical layer's native format; CMSMPhysNodeAddress is not used. 01 This is an illegal value and must not be used. 10 <i>MLIDCFG_NodeAddress</i> is canonical; use CMSMPhysNodeAddress . 11 <i>MLIDCFG_NodeAddress</i> is noncanonical; use CMSMPhysNodeAddress . Also see <i>MLIDCFG_NodeAddress</i> , CMSMPhysNodeAddress , and <i>ODI Specification Supplement: Canonical and Noncanonical Addressing</i> . |
| 15 | MM__PHYS_NODE_ADDR_BIT | |

All bits that are not listed in the above table are reserved and must be initialized to 0.

MLIDCFG_Flags Field

This section describes the bits of the *MLIDCFG_Flags* field in the configuration table. Figure 3-3 shows the reserved bits and their values for this field.

Figure 3-3
MLIDCFG_Flags Field

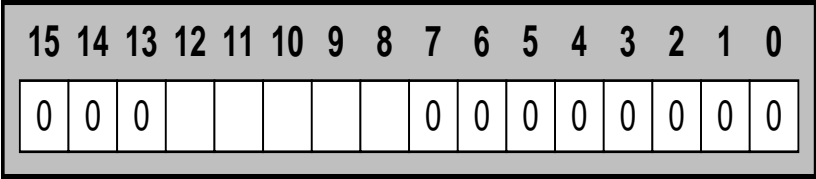


Table 3-4 describes the non-reserved bit values for the *MLIDCFG_Flags* field

Table 3-4
MLIDCFG_Flags Bits Description

| Bit # | Name | Description |
|-------|-----------------------|---|
| 8 | MF_HUB_MANAGEMENT_BIT | Set to 1 if the C HSM supports hub management. |
| 9 | MF_SOFT_FILT_GRP_BIT | See description below for bit 10. |
| 10 | MF_GRP_ADDR_SUP_BIT | <p>Bits 9 and 10 indicate different support mechanisms for multicast filtering. These bits are only valid if bit 3 of the <i>MModeFlags</i> is set, indicating that the C HSM supports multicast addressing.</p> <p>The C HSM sets bit 10 if it has specialized adapter hardware (such as hardware that utilizes CAM memory).</p> <p>When the C HSM sets bit 10; and its default is functional addressing, but it also supports group addressing; it receives both functional addresses and group addresses.</p> <p>The state of bit 9 is defined only if bit 10 is set. Bit 9 is set if the adapter completely filters group addresses and the CTSM does not need to perform any checking. The C HSM can dynamically set and clear bit 9. For example, if the adapter utilizes CAM memory, but has temporarily run out memory, the CTSM must temporarily filter the group addresses. In this case, the C HSM must reset bit 9.</p> <p>Bit 9 is <u>not</u> used by ECB aware HSMs. ECB aware HSMs must do their own filtering of multicast addresses.</p> |

Table 3-4
MLIDCFG_Flags Bits Description

| Bit # | Name | Description |
|-------|--------------------|--|
| 10/9 | | Bits 10 and 9 combinations are as follows: 00 The format of the multicast address defaults to that of the topology: Ethernet => Group Addressing (Multicast Addressing) Token-Ring => Group Addressing and Functional Addressing FDDI => Group Addressing 01 Illegal value which must not occur. 10 Group addressing is supported by the specialized adapter hardware, but the TSM filters the addresses. 11 Group addressing is supported by the specialized adapter hardware, and the TSM is not required to filter the addresses. See also <i>ODI Specification Supplement: Canonical and Noncanonical Addressing</i> for information regarding octet bit reversal. |
| 11 | MF_RECONFIG_BIT | This bit is used by the <i>Novell Driver Developer's Guide</i> kit. When set, it indicates to the C HSM that indirect (file based) configuration information for the associated interface instance may have changed. This bit can be set by any caller prior to calling the DriverReset function. It is to be examined by DriverReset and cleared upon completion. This bit has no meaning for C HSMs which do not support use of indirect (file based) configuration information. |
| 12 | MF_PRIORITYSUP_BIT | The C HSM sets this bit during initialization if the following conditions are met: (1) The C HSM has provided a priority queue service support routine (such as, DriverPriorityQueueSupport). (2) The C HSM has set the <i>MLIDCFG_PrioritySup</i> field to something other than 0. Note: The C HSM may set/clear this bit to enable/disable priority support as needed. |

All bits that are not listed in the previous table are reserved and must be initialized to 0.

MLIDCFG_SharingFlags Field

This section describes the bits of the *MLIDCFG_SharingFlags* field in the configuration table. Figure 3-4 shows the reserved bits and their values for this field.

Figure 3-4
MLID_SharingFlags Field Default Values

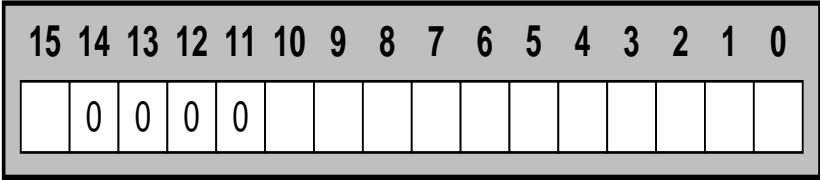


Table 3-5 describes the non-reserved bit values for the *MLIDCFG_SharingFlags* field.

Table 3-5
MLIDCFG_SharingFlags Bits Description

| Bit # | Name | Description |
|-------|----------------------|--|
| 0 | MS_SHUTDOWN_BIT | Set to 1 if the logical board is currently shut down. This bit should also be set during DriverInit until the driver/adaptor is fully functional and ready to send and receive packets. |
| 1 | MS_SHARE_PORT0_BIT | Set to 1 if the adapter can share I/O port 0. |
| 2 | MS_SHARE_PORT1_BIT | Set to 1 if the adapter can share I/O port 1. |
| 3 | MS_SHARE_MEMORY0_BIT | Set to 1 if the adapter can share memory range 0. |
| 4 | MS_SHARE_MEMORY1_BIT | Set to 1 if the adapter can share memory range 1. |
| 5 | MS_SHARE_IRQ0_BIT | Set to 1 if the adapter can share interrupt 0. |
| 6 | MS_SHARE_IRQ1_BIT | Set to 1 if the adapter can share interrupt 1. |
| 7 | MS_SHARE_DMA0_BIT | Set to 1 if the adapter can share DMA channel 0. |

Table 3-5
MLIDCFG_SharingFlags Bits Description

| Bit # | Name | Description |
|-------|------------------------|--|
| 8 | MS_SHARE_DMA1_BIT | Set to 1 if the adapter can share DMA channel 1. |
| 10 | MS_HAS_CMD_INFO_BIT | If this bit is zero, the command line used by some install programs will be created using the system's IOCONFIG structure and possibly (as controlled by bit 9) the content of the users command line. This command line will include an entry for every field that is used in the IOCONFIG structure. Setting this bit prevents the install program from creating a command line using the IOCONFIG structure; instead, it simply uses the user's command line and ignores the state of bit 9. |
| 9 | MS_NO_DEFAULT_INFO_BIT | If this bit is set and bit 10 is not set, some install programs will merge the contents of the user's command line with the system's IOCONFIG structure. If it is not set, then only the system's IOCONFIG structure will be used to create the command line. The C MSM sets this bit if the command line passed to DriverInit is not empty. |
| 15 | MS_MEM_PAGE_BIT | <p>When set, this bit signifies that the values in fields <i>MLIDCFG_MemorySize0</i> and <i>MLIDCFG_MemorySize1</i> contain the number of pages of memory used by the adapter. For example, Intel platforms allow 4K pages with a maximum of 256 megabytes of shared memory address used by an adapter.</p> <p>When clear, this bit signifies that the values in fields <i>MLIDCFG_MemorySize0</i> and <i>MLIDCFG_MemorySize1</i> contain the number of paragraphs (16 bytes) of memory used by the adapter.</p> |

All bits that are not listed in the above table are reserved and must be initialized to 0.

Maximum Packet Size

The *MLIDCFG_MaxFrameSize* field of the configuration table is set to the LSL's maximum ECB buffer size during **<CTSM>RegisterHSM**. The C HSM can lower this value prior to calling **CMSMRegisterMLID**. During this procedure, the CTSM alters the size if the topology requires a smaller maximum packet size. The CTSM also sets *MLIDCFG_BestDataSize* and *MLIDCFG_WorstDataSize*. After **CMSMRegisterMLID** returns, MLIDs for

intelligent adapters can pass the maximum size to the hardware if required. The following table shows how these values are determined.

Table 3-6
Frame Types Versus Size Fields

| Frame Type | MLIDCFG_MaxFrameSize (the lesser of the two values) | MLIDCFG_BestDataSize | MLIDCFG_WorstDataSize |
|---------------------|--|----------------------------------|----------------------------------|
| Ethernet 802.3 | Maximum ECB buffer size or 1514 | <i>MLIDCFG_MaxFrameSize</i> - 14 | <i>MLIDCFG_MaxFrameSize</i> - 14 |
| Ethernet 802.2 | Maximum ECB buffer size or 1514 | <i>MLIDCFG_MaxFrameSize</i> - 17 | <i>MLIDCFG_MaxFrameSize</i> - 18 |
| Ethernet II | Maximum ECB buffer size or 1514 | <i>MLIDCFG_MaxFrameSize</i> - 14 | <i>MLIDCFG_MaxFrameSize</i> - 14 |
| Ethernet SNAP | Maximum ECB buffer size or 1514 | <i>MLIDCFG_MaxFrameSize</i> - 22 | <i>MLIDCFG_MaxFrameSize</i> - 22 |
| Token-Ring 802.2 | Maximum ECB buffer size or the maximum size the adapter can handle | <i>MLIDCFG_MaxFrameSize</i> - 17 | <i>MLIDCFG_MaxFrameSize</i> - 48 |
| Token-Ring SNAP | Maximum ECB buffer size or the maximum size the adapter can handle | <i>MLIDCFG_MaxFrameSize</i> - 22 | <i>MLIDCFG_MaxFrameSize</i> - 52 |
| FDDI 802.2 | Maximum ECB buffer size or 4491 | <i>MLIDCFG_MaxFrameSize</i> - 16 | <i>MLIDCFG_MaxFrameSize</i> - 47 |
| FDDI SNAP | Maximum ECB buffer size or 4491 | <i>MLIDCFG_MaxFrameSize</i> - 21 | <i>MLIDCFG_MaxFrameSize</i> - 51 |

Example

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

- *MLIDCFG_BestDataSize*

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

= *MLIDCFG_MaxFrameSize* (4096) - MAC header (14) - 802.2 Type I LLC header (3)

= 4079

- *MLIDCFG_WorstDataSize*

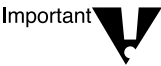
The maximum packet size minus the headers if the source routing header is included.

= *MLIDCFG_MaxFrameSize* (4096) - MAC header (14) - 802.2 Type II LLC header (4) - Source Routing header (30)

= 4048

Driver Adapter Data Space

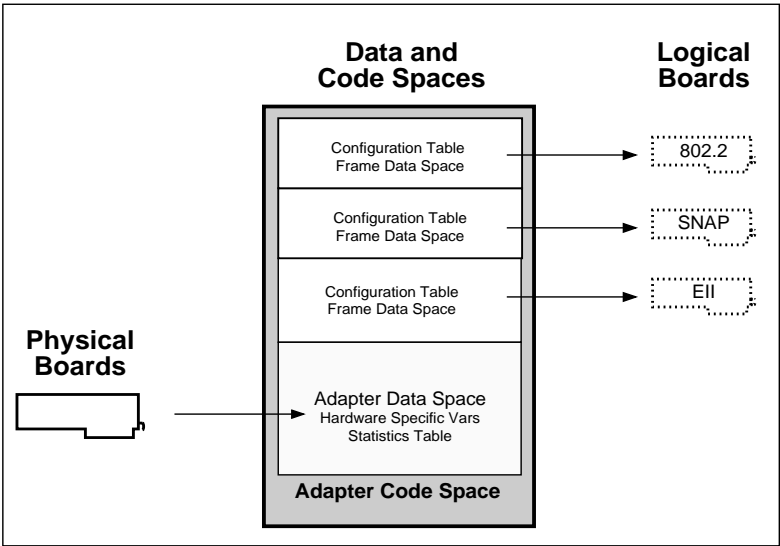
The C HSM must define and initialize a structure containing certain hardware-related and statistical information specific to the adapter. This structure is called DRIVER_DATA and must contain the statistics table and any hardware-specific fields needed in order to drive the physical board. The statistics table is defined by the ODI specification.



DRIVER_DATA must contain: the MLID_STATS_TABLE structure, all the generic, media, and custom STAT_TABLE_ENTRY structures, and all of the generic, media, and custom counter variables.

When the **DriverInit** routine calls **CMSMRegisterHardwareOptions**, the CMSM allocates the adapter data space and creates a copy of DRIVER_DATA in this area. There is only one adapter data space allocated for each physical board, regardless of the number of frame types supported. Figure 3.2 shows the relationship between the frame and adapter data space.

Figure 3-5
Driver Frame and Adapter Data Space



Specification Version String

In order to identify which version of this specification a C HSM conforms to, a version string (the "specification version string") must be embedded in the C HSM. The specification version string number (1.11 for this specification) is the actual version number of the specification. The following is the specification version string for this specification; it must be added to the C HSM where the global variable declarations are made, exactly as shown:

```
MEON_STRING  C HSMSPEC[ ] = "HSM_CSPEC_VERSION: 1.11";
```

The pointer in the DriverParameterBlock structure field, *HSMSpecVersionStringPtr*, must point to this string.

Note

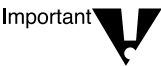


One space is required between the colon and the 1.

Driver Statistics Table

The statistics table contains various diagnostic counters. All statistics counters listed must be present in the table, but only those marked mandatory must be supported. These counters can be grouped into the following categories.

- Generic Statistics Counters
- Media Specific Counters
- Custom Statistics Counters



When the statistics counters reach their maximum value, they wrap back to their beginning value. Each category of counters must be grouped in memory contiguously as shown in the examples. They must be ordered as described in the following tables. The CMSM/CTSM fix the string pointers for Generic and Media Specific counters during **CMSMRegisterMLID**.

STAT_TABLE_ENTRY Structure

```
typedef struct _STAT_TABLE_ENTRY_
{
    UINT32          StatUseFlag;
    void            *StatCounter;
    MEON_STRING     *StatString;
} STAT_TABLE_ENTRY;
```

Field Descriptions

StatUseFlag

The permissible values of *StatUseFlag* are defined as:

| | |
|----------------------|--|
| ODI_STAT_UNUSED | <i>StatCounter</i> is not in use. |
| ODI_STAT_UINT32 | <i>StatCounter</i> is a pointer to a UINT32 counter. |
| ODI_STAT_UINT64 | <i>StatCounter</i> is a pointer to a UINT64 counter. |
| ODI_STAT_MEON_STRING | <i>StatCounter</i> is a pointer to MEON string. |

ODI_STAT_UNTYPED

StatCounter is a pointer to a UINT8 array preceded by its length (UINT32).

StatCounter

As defined by the StatUseFlag.

StatString

Pointer to a NULL-terminated MEON string describing the statistics counter.

Statistics Table Structure

```
typedef struct _MLID_STATS_TABLE_  
{  
    UINT16          MStatTableMajorVer;  
    UINT16          MStatTableMinorVer;  
    UINT32          MNumGenericCounters;  
    STAT_TABLE_ENTRY (*MGenericCountsPts)[ ];  
    UINT32          MNumMediaCounters;  
    STAT_TABLE_ENTRY (*MMediaCountsPts)[ ];  
    UINT32          MNumCustomCounters;  
    STAT_TABLE_ENTRY (*MCustomCountersPtr)[ ];  
} MLID_STATS_TABLE;
```

Table 3-7
MLID Statistics Table Fields

| Name | Type | Description |
|---------------------|-------------------|--|
| MStatTableMajorVer | UINT16 | The major version number of the statistics table. The current major number is 4. |
| MStatTableMinorVer | UINT16 | The minor version number of the statistics table. The current minor version number is 0. |
| MNumGenericCounters | UINT32 | The total number of generic STAT_TABLE_ENTRY counters in this portion of this table. This field is set to 20 for this specification. |
| MGenericCountsPts | STAT_TABLE_ENTRY* | Pointer to an array of STAT_TABLE_ENTRY counters [MNumGenericCounters]. |

Table 3-7

MLID Statistics Table Fields

| Name | Type | Description |
|--------------------|-------------------|---|
| MNumMediaCounters | UINT32 | The total number of media specific STAT_TABLE_ENTRY counters in this portion of this table. This field is set to the following values: Token-Ring 13 Ethernet 8 FDDI 10 |
| MMediaCountsPts | STAT_TABLE_ENTRY* | Pointer to an array of STAT_TABLE_ENTRY counters [<i>MNumMediaCounters</i>]. |
| MNumCustomCounters | UINT32 | This field contains the total number of custom STAT_TABLE_ENTRY counters in this portion of this table. This field is variable (dependent on the C HSM). |
| MCustomCountersPtr | STAT_TABLE_ENTRY* | Pointer to an array of STAT_TABLE_ENTRY counters [<i>MCustomCounters</i>]. |

Example

```

#define          NUM_GENERIC_COUNTERS  20
UINT32          MTotalTxPacketCount ,
                MTotalRxPacketCount ,
                MNoECBAvailableCount ,
                MPacketTxTooBigCount ,
                MPacketTxTooSmallCount ,
                MPacketRxOverflowCount ,
                MPacketRxTooBigCount ,
                MPacketRxTooSmallCount ,
                MTotalTxMiscCount ,
                MTotalRxMiscCount ,
                MRetryTxCount ,
                MChecksumErrorCount ,
                MHardwareRxMismatchCount ,
                MTotalTxOKByteCount ,
                MTotalRxOKByteCount ,
                MTotalGroupAddrTxCount ,
                MTotalGroupAddrRxCount ,
                MAdapterResetCount ,
                MAdapterOprTimeStamp ,
                MQDepth;

```

```
STAT_TABLE_ENTRY      MGenericCounters  [NUM_GENERIC_COUNTERS] =
{
    { ODI_STAT_UINT32,  &MTotalTxPacketCount,  NULL },
    { ODI_STAT_UINT32,  &MTotalRxPacketCount,  NULL },
    { ODI_STAT_UINT32,  &MNoECBAvailableCount,  NULL },
    { ODI_STAT_UINT32,  &MPacketTxTooBigCount,   NULL },
    { ODI_STAT_UINT32,  &MPacketTxTooSmallCount, NULL },
    { ODI_STAT_UINT32,  &MPacketRxOverflowCount, NULL },
    { ODI_STAT_UINT32,  &MPacketRxTooBigCount,   NULL },
    { ODI_STAT_UINT32,  &MPacketRxTooSmallCount, NULL },
    { ODI_STAT_UINT32,  &MTotalTxMiscCount,     NULL },
    { ODI_STAT_UINT32,  &MTotalRxMiscCount,     NULL },
    { ODI_STAT_UINT32,  &MRetryTxCount,        NULL },
    { ODI_STAT_UINT32,  &MChecksumErrorCount,   NULL },
    { ODI_STAT_UINT32,  &MHardwareRxMismatchCount, NULL },
    { ODI_STAT_UINT64,  &MTotalTxOKByteCount,   NULL },
    { ODI_STAT_UINT64,  &MTotalRxOKByteCount,   NULL },
    { ODI_STAT_UINT32,  &MTotalGroupAddrTxCount, NULL },
    { ODI_STAT_UINT32,  &MTotalGroupAddrRxCount, NULL },
    { ODI_STAT_UINT32,  &MAdapterResetCount,    NULL },
    { ODI_STAT_UINT32,  &MAdapterOprTimeStamp,  NULL },
    { ODI_STAT_UINT32,  &MQDepth,              NULL },
};

MLID_STATS_TABLE      MLID_StatsTable = {4, 0,
NUM_GENERIC_COUNTERS,
MGenericCounters, 0, NULL, 0, NULL};
```

Table 3-8
MLID Statistics Table Generic Counters

| Name | Type | Description |
|---------------------|--------|---|
| MTotalTxPacketCount | UINT32 | Number of packets successfully transmitted onto the media. The CTSM/CMSM increments this counter, except on ECB-aware adapters where the C HSM increments this counter. Mandatory. |
| MTotalRxPacketCount | UINT32 | Number of packets reported as successfully received without errors. This counter is independent of whether the packet is accepted. Mandatory. |

Table 3-8

MLID Statistics Table Generic Counters *continued*

| Name | Type | Description |
|------------------------|--------|---|
| MNoECBAvailableCount | UINT32 | Number of times an incoming packet was discarded due to lack of host receive buffers or the host not wanting the packet. The CTSM/CMSM increments this counter, except on ECB-aware adapters where the C HSM increments this counter. Mandatory. |
| MPacketTxTooBigCount | UINT32 | Number of times a send packet was too big for transmission. The CTSM/CMSM increments this counter, except on ECB-aware adapters where the C HSM increments this counter. Mandatory. |
| MPacketTxTooSmallCount | UINT32 | Number of requested packets for transmission that were normally too small to be transmitted. The packets might still have been sent if the MLID does padding. Normally this field will not be used. The CTSM/CMSM increments this counter, except on ECB-aware adapters where the C HSM increments this counter. Optional. |
| MPacketRxOverflowCount | UINT32 | Number of times the adapter's receive buffer pool was exhausted, causing subsequent incoming packets to be discarded. The C HSM increments this counter. Optional. |
| MPacketRxTooBigCount | UINT32 | Number of times a packet was received that was too large to fit into preallocated receive buffers provided by the host, or too large for media definitions. The CTSM/CMSM increments this counter, except on ECB-aware adapters where the C HSM increments this counter. Mandatory. |
| MPacketRxTooSmallCount | UINT32 | Number of times a packet was received that was too small for media definitions. The CTSM/CMSM increments this counter, except on ECB-aware adapters where the C HSM increments this counter. Optional. |
| MTotalTxMiscCount | UINT32 | Number of times the MLID failed to transmit and has no appropriate generic counter to increment. The C HSM increments this counter. Mandatory. |
| MTotalRxMiscCount | UINT32 | Number of times the MLID receives a packet with errors and has no appropriate generic counter to increment. The C HSM increments this counter. Mandatory. |

Table 3-8
MLID Statistics Table Generic Counters *continued*

| Name | Type | Description |
|--------------------------|--------|--|
| MRetryTxCount | UINT32 | Number of times the MLID retried a transmit operation because of a failure. The C HSM increments this counter. Optional. |
| MChecksumErrorCount | UINT32 | Number of times the MLID received a packet with corrupt data—for example, CRC errors. The C HSM increments this counter. Optional. |
| MHardwareRxMismatchCount | UINT32 | Number of times the MLID received a packet that did not pass the length consistency checks. The CTSM/CMSM increments this counter, except on ECB-aware adapters where the C HSM increments this counter. Optional. |
| MTotalTxOKByteCount | UINT64 | Number of bytes (including low-level headers) the MLID successfully transmitted onto the media. The CTSM/CMSM increments this counter. Mandatory. |
| MTotalRxOKByteCount | UINT64 | Number of bytes (including low-level headers) the MLID successfully received. The CTSM/CMSM increments this counter. Mandatory. |
| MTotalGroupAddrTxCount | UINT32 | Number of packets the MLID transmitted with a group destination address. The CTSM/CMSM increments this counter, except on ECB-aware adapters where the C HSM increments this counter. Mandatory. |
| MTotalGroupAddrRxCount | UINT32 | Number of packets the MLID received with a group destination address. The CTSM/CMSM increments this counter, except on ECB-aware adapters where the C HSM increments this counter. Mandatory. |
| MAdapterResetCount | UINT32 | Number of times the adapter was reset due to an internal failure or a call to the MLID's DriverReset function. The C HSM increments this counter. Mandatory. |
| MAdapterOprTimeStamp | UINT32 | This counter contains the time (platform dependent clock, such as number of ticks) since the adapter last changed operational state—for example, load time, DriverShutdown , DriverReset . Mandatory. |

Table 3-8

MLID Statistics Table Generic Counters *continued*

| Name | Type | Description |
|---------|--------|---|
| MQDepth | UINT32 | Number of transmit ECBs that are queued for the adapter. The CTSM/CMSM increments this counter, except on ECB-aware adapters where the C HSM increments this counter. Mandatory. |

MLID Statistics Table Media Specific Counters

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



Note Media-specific counters must be grouped in memory contiguously and in the order described in the following tables:

Token-Ring Counters

The following table describes media specific counters array STAT_TABLE_ENTRY for Token-Ring.

Table 3-9

Media Specific Counters for Token-Ring

| Size | Label | Description |
|--------|---------------------------|--|
| UINT32 | TRN_ACErrCounter | Number of times a station receives an AMP or SMP frame in which A = C = 0, and then receives another SMP frame with A = C = 0 without first receiving an AMP frame. The C HSM increments this counter. Mandatory. |
| UINT32 | TRN_AbortDelimiterCounter | Number of times a station transmits an abort delimiter while transmitting. The C HSM increments this counter. Mandatory. |
| UINT32 | TRN_BurstErrorCounter | Number of times 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. The C HSM increments this counter. Mandatory. |

Table 3-9
Media Specific Counters for Token-Ring *continued*

| Size | Label | Description | | | | | | | | | | | | | | | | | | | | | | | | |
|---------|-----------------------------|---|--------|-------------|--------|------------|--------|------------|--------|-----------------|--------|-----------------|--------|----------------------|-------|----------|-------|-----------------|-------|------------------|-------|----------------|-------|---------------|---------|----------|
| UINT32 | TRN_FrameCopiedErrorCounter | Number of times a station recognizes a frame addressed to its specific address and detects that the FS field bits are set to 1, indicating a possible line hit or duplicate address. The C HSM increments this counter. Mandatory. | | | | | | | | | | | | | | | | | | | | | | | | |
| UINT32 | TRN_FrequencyErrorCounter | Number of times the frequency of the incoming signal differs from the expected frequency by more than what is specified in Section 7 of the <i>IEEE Std 802.5-1989</i> . The C HSM increments this counter. Mandatory. | | | | | | | | | | | | | | | | | | | | | | | | |
| UINT32 | TRN_InternalErrorCounter | Number of times a station recognizes a recoverable internal error. This can be used for detecting a station in marginal operating condition. The C HSM increments this counter. Mandatory. | | | | | | | | | | | | | | | | | | | | | | | | |
| UINT32 | TRN_LastRingStatus | <p>The last ring status reported by the adapter with the following bit definitions:</p> <table><tr><td>Bit 15</td><td>Signal loss</td></tr><tr><td>Bit 14</td><td>Hard error</td></tr><tr><td>Bit 13</td><td>Soft error</td></tr><tr><td>Bit 12</td><td>Transmit beacon</td></tr><tr><td>Bit 11</td><td>Lobe wire fault</td></tr><tr><td>Bit 10</td><td>Auto-removal error 1</td></tr><tr><td>Bit 9</td><td>Reserved</td></tr><tr><td>Bit 8</td><td>Remove received</td></tr><tr><td>Bit 7</td><td>Counter overflow</td></tr><tr><td>Bit 6</td><td>Single station</td></tr><tr><td>Bit 5</td><td>Ring recovery</td></tr><tr><td>Bit 4-0</td><td>Reserved</td></tr></table> <p>The C HSM maintains this value. Mandatory.</p> | 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 4-0 | Reserved |
| 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 4-0 | Reserved | | | | | | | | | | | | | | | | | | | | | | | | | |
| UINT32 | TRN_LineErrorCounter | <p>Number of times 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 exists:</p> <ol style="list-style-type: none">1) The frame or token contains a non-data bit (J or K bit) between the SD and the ED of the frame or token.2) The frame contains a FCS error in a frame. <p>The C HSM increments this counter. Mandatory.</p> | | | | | | | | | | | | | | | | | | | | | | | | |

Table 3-9

Media Specific Counters for Token-Ring *continued*

| Size | Label | Description |
|--------|-------------------------|--|
| UINT32 | TRN_LostFrameCounter | Number of times 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. The C HSM increments this counter. Mandatory. |
| UINT32 | TRN_TokenErrorCounter | Number of times a station acting as the active monitor recognizes an error condition that needs a token transmitted. This occurs when the TVX timer expires. The C HSM increments this counter. Mandatory. |
| UINT64 | TRN_UpstreamNodeAddress | The upstream neighbor node address, right justified with leading zeros. The C HSM maintains this value. Mandatory. |
| UINT32 | TRN_LastRingID | The value of the local ring. The C HSM maintains this value. Mandatory. |
| UINT32 | TRN_LastBeaconType | The value of the last beacon type. The C HSM maintains this value. Mandatory. |

Ethernet Counters

This section describes the media specific counters array
STAT_TABLE_ENTRY for Ethernet.

Table 3-10

Media Specific Counters for Ethernet

| Size | Label | Description |
|--------|-------------------------------|---|
| UINT32 | ETH_TxOKSingleCollisionsCount | Count of frames that are involved in a single collision and are subsequently transmitted successfully. This counter is incremented when the result of a transmission is reported as successful and the attempt value is 2. The C HSM increments this counter. Mandatory. |

Table 3-10
Media Specific Counters for Ethernet *continued*

| Size | Label | Description |
|--------|---------------------------------|---|
| UINT32 | ETH_TxOKMultipleCollisionsCount | Count of frames that are involved in more than one collision, but are transmitted successfully. This counter is incremented when the transmission is successful and the attempt value is greater than 2, but less than or equal to the attempt limit of the network controller used by the MLID (the attempt limit is specified by <i>MLIDCFG_SendRetries</i>). The C HSM increments this counter. Mandatory. |
| UINT32 | ETH_TxOKButDeferred | Count of frames whose transmission was delayed on its first attempt because the medium was busy. The C HSM increments this counter. Mandatory. |
| UINT32 | ETH_TxAbortLateCollision | Count of the times that a collision has been detected later than 512 bit times into the transmitted packet. A late collision is counted twice, both as a collision and as a late collision. The C HSM increments this counter. Mandatory. |
| UINT32 | ETH_TxAbortExcessCollision | Count of frames that, due to excessive collisions, are not transmitted successfully. This counter is incremented when the value of attempts variable equals the attempt limit (the attempt limit is specified by <i>MLIDCFG_SendRetries</i>) during a transmission. The C HSM increments this counter. Mandatory. |
| UINT32 | ETH_TxAbortCarrierSense | Count of frames that the <i>carrierSense</i> signal was not asserted or was deasserted during the transmission of a frame without collision. The C HSM increments this counter. Mandatory. |
| UINT32 | ETH_TxAbortExcessiveDeferral | Count of frames that were deferred for an excessive period of time. This counter must only be incremented once per LLC transmission. The C HSM increments this counter. Mandatory. |
| UINT32 | ETH_RxAbortFrameAlignment | Count of frames that are not an integral number of bytes in length and do not pass the FCS check. The C HSM increments this counter. Mandatory. |

FDDI Counters

Table 3-11 describes the media specific counters array STAT_TABLE_ENTRY for FDDI.

Table 3-11
Media Specific Counters for FDDI

| Size | Label | Description |
|--------|-------------------------|--|
| UINT32 | FDDI_ConfigurationState | <p>(ANSI <i>fdiSMTCFState</i>) This field contains attachment configuration for the station or concentrator.</p> <p>0 = Isolated 1 = local_a 2 = local_b 3 = local_ab 4 = local_s 5 = wrap_a 6 = wrap_b 7 = wrap_ab 8 = wrap_s 9 = c_wrap_a 10 = c_wrap_b 11 = c_wrap_s 12 = through</p> <p>The C HSM increments this counter. Mandatory.</p> |
| UINT64 | FDDI_UpstreamNode | <p>(ANSI <i>fdiMACUpstreamNbr</i>) This counter contains the MAC's upstream neighbor's long individual MAC address; 0 if unknown. The C HSM maintains this counter. Mandatory.</p> |
| UINT64 | FDDI_DownstreamNode | <p>(ANSI <i>fdiMACDownstreamNbr</i>) This field contains the MAC's downstream neighbor's long individual MAC address; 0 if unknown. The C HSM maintains this counter. Mandatory.</p> |
| UINT32 | FDDI_FrameErrorCount | <p>Count of the number of frames that were detected in error by this MAC that had not been detected by another MAC. The C HSM increments this counter. Mandatory.</p> |
| UINT32 | FDDI_FramesLostCount | <p>Count of the number of instances that this MAC detected format errors during frame reception such that the frame was stripped. The C HSM increments this counter. Mandatory.</p> |

Table 3-11
Media Specific Counters for FDDI *continued*

| Size | Label | Description |
|--------|--------------------------|--|
| UINT32 | FDDI_RingManagementState | <p>This field indicates the current state of the ring management state machine.</p> <p>0 = Isolated 1 = Non_Op 2 = Ring_Op 3 = Detect 4 = Non_Op_Dup 5 = Ring_Op_Dup 6 = Directed 7 = Trace</p> <p>The C HSM maintains this value. Mandatory.</p> |
| UINT32 | FDDI_LCTFailureCount | <p>Count of consecutive times the link confidence test (LCT) has failed during connection management. The C HSM increments this counter. Mandatory.</p> |
| UINT32 | FDDI_LemRejectCount | <p>Link error monitoring count of the times that a link has been rejected. The C HSM increments this counter. Mandatory.</p> |
| UINT32 | FDDI_LemCount | <p>Aggregate link error monitor error count (zero only on station power up). The C HSM increments this counter. Mandatory.</p> |
| UINT32 | FDDI_ConnectionState | <p>The state of this port's pcm state machine.</p> <p>0 = off 1 = break 2 = trace 3 = connect 4 = next 5 = signal 6 = join 7 = verify 8 = active 9 = maint</p> <p>The C HSM maintains this value. Mandatory.</p> |

Driver Firmware

C HSMs might need to download firmware for intelligent adapters. Since most intelligent adapters have a microprocessor on the adapter, such as an Intel 80186, the firmware code must be separately written, assembled, and linked to generate a binary file. This section describes how this firmware binary file can be attached to the C HSM at link time, and then transferred to the adapter during initialization.

To attach a firmware binary file to the C HSM, the linker definition file must include the keyword "custom", followed by the name of the binary file. When the MLID is linked, the file is attached to the end of the C HSM code and becomes part of the NLM.

During the initialization process, the CMSM allocates a buffer and copies the contents of the attached file to that buffer. To gain access to the firmware buffer, the C HSM must properly initialize the Driver Parameter Block variables described below. The CMSM determines the value of these parameters when the C HSM's **DriverInit** routine calls **<CTSM>RegisterHSM**. The C HSM can then download the contents of the firmware buffer to the adapter.

DriverFirmwareSize Value

If custom firmware is used, the C HSM initializes this UINT32 variable to any nonzero value. The CMSM replaces this value with the actual size of the firmware buffer when **DriverInit** calls **<CTSM>RegisterHSM**. If custom firmware is not used, the C HSM must initialize this variable to 0.

DriverFirmwareBuffer Value

This void pointer value is set to point to the firmware buffer by the CMSM when **DriverInit** calls **<CTSM>RegisterHSM**.

3-48 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

chapter **4** ***CMSM/CTSM Structures and Variables***

Overview

This chapter describes the functions, structures, variables, and constants defined by the CMSM and the CTSM. Some of the variables and structures in this chapter are required for controlling processes, and must be initialized, updated, or managed by the CHSM. Others are available as optional support for developers and may be used accordingly.

The following functions, variables, and structures are described in this chapter:

The CMSM Data Access Function and CMSM Variables

- CMSMVirtualBoardLink
- CMSMDefaultVirtualBoard
- CMSMStatusFlags
- CMSMTxFreeCount
- CMSMPriorityTxFreeCount
- CMSMMaxFrameHeaderSize
- CMSMPhysNodeAddress

Data Structures

- Receive Control Blocks (RCBs)
- Transmit Control Blocks (TCBs)
- CMSM Configuration Table (CMSM_CONFIG_TABLE)
- CTSM Configuration Table (CTSM_CONFIG_TABLE)
- DRIVER_OPTION Structure

CMSM Data Access

The CHSM must access several variables located in the CMSM's data space. This section describes these variables and the function that enables the CHSM to access them.

DADSP_TO_CMSMADSP Macro

The **DADSP_TO_CMSMADSP** (CHSM Driver Adapter Data Space to CMSM Adapter Data Space) macro takes a pointer to the CHSM's adapter data space and returns a pointer to the CMSM's shared data space. This pointer can then be used to access the CMSM variables that follow.

CMSMVirtualBoardLink Pointers

The CMSM maintains a separate configuration table for each frame type supported by the MLID. **CMSMVirtualBoardLink** is an array of pointers to these configuration tables. This array contains four pointers for Ethernet, two for Token-Ring, and two for FDDI. If a particular frame has not been loaded, the pointer to the corresponding configuration table is NULL.

The following examples are definitions of **CMSMVirtualBoardLink** for Ethernet, Token-Ring, and FDDI.

4-2 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

Ethernet Example

```
/* ETHERNET 802.2 */
DADSP_TO_CMSPADSP(driverData)->CMSMVirtualBoardLink[0]
/* ETHERNET 802.3 */
DADSP_TO_CMSPADSP(driverData)->CMSMVirtualBoardLink[1]
/* ETHERNET II */
DADSP_TO_CMSPADSP(driverData)->CMSMVirtualBoardLink[2]
/* ETHERNET SNAP */
DADSP_TO_CMSPADSP(driverData)->CMSMVirtualBoardLink[3]
```

Token-Ring Example

```
/* TOKEN 802.2 */
DADSP_TO_CMSPADSP(driverData)->CMSMVirtualBoardLink[0]
/* TOKEN SNAP */
DADSP_TO_CMSPADSP(driverData)->CMSMVirtualBoardLink[1]
```

FDDI Example

```
/* FDDI 802.2 */
DADSP_TO_CMSPADSP(driverData)->CMSMVirtualBoardLink[0]
/* FDDI SNAP */
DADSP_TO_CMSPADSP(driverData)->CMSMVirtualBoardLink[1]
```

CMSMDefaultVirtualBoard Pointer

CMSMDefaultVirtualBoard is a pointer to the configuration table that contains the identity of the first frame type loaded. If the frame type is not important, calling **CMSMDefaultVirtualBoard** will quickly select the first frame type loaded.

CMSMStatusFlags Variable

The CMSM maintains a UINT32 variable that provides adapter status information that enables the CHSM to determine if the adapter is shut down.

| | | |
|-----------|-------------------|---|
| Bit 0 | SHUTDOWN | When set, this bit indicates that the adapter is shut down. |
| Bits 1-3 | Reserved | Reserved for use by the CMSM/CTSM. |
| Bit 4 | POLLING_SUSPENDED | When set, this bit indicates that polling has been suspended. |
| Bits 5-31 | Reserved | Reserved for use by the CMSM/CTSM. |

The CHSM can use *CMSMStatusFlags* to determine if an adapter is partially shut down. If bit 0 is set, the adapter is partially shut down and should not be serviced. The CMSM will not call **DriverSend** to transmit a packet if the adapter is partially shut down.

CMSMTxFreeCount Variable

During initialization, the CHSM must specify the number of hardware resources available on the adapter for handling pending packet transmissions. The CTSM 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 CTSM allocates.

For example, if the adapter has a second transmit buffer that can accept another packet before the current transmission is complete, the CHSM should set *CMSMTxFreeCount* to a value of 2. If the adapter supports hardware queuing, the count should represent the number of transmissions that the adapter can efficiently process. If the adapter has no additional resources available, other than those used to transmit the current packet, set *CMSMTxFreeCount* to 1.

The CTSM decrements *CMSMTxFreeCount* before it calls **DriverSend**. The CTSM assumes that the adapter is not ready for another packet if this count reaches 0.

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

Example:

```
++(DADSP_TO_CMSMADSP(driverData)->CMSMTxFreeCount);
```

CMSMPriorityTxFreeCount

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

Example:

```
DADSP_TO_CMSMADSP(driverData)->CMSMPriorityTxFreeCount = 2;
```

CMSMMaxFrameHeaderSize Variable

The <CTSM>**GetRCB** function, which can be used during packet reception, employs a "lookahead" process, where the packet header is placed in a buffer and previewed by the upper layers. This allows the upper layers to verify that they want the packet before the entire packet is read from the adapter.

The CTSM sets *CMSMMaxFrameHeaderSize* to the number of bytes the CHSM must transfer to the lookahead buffer. This value is equal to *MLIDCFG_LookAheadSize* from the configuration table plus the maximum

media header size. The *MLIDCFG_LookAheadSize* size can be up to 128 bytes. For example:

```
MLIDCFG_LookAheadSize = 128
Ethernet Maximum Media Header Size = 22
CMSMMaxFrameHeaderSize = 128 + 22 = 150
```

The CHSM must read the lookahead buffer size before calling **<CTSM>GetRCB** each time, because the size can change dynamically. The CHSM can optionally implement *DriverRxLookAheadChange* to inform intelligent adapters when the size changes, rather than forcing them to continually check.

For more information on the lookahead process, see the Packet Reception section in Chapter 5, "CHSM Functions" and **<CTSM>GetRCB** in Chapter 6, "CTSM Functions". Refer to the *DriverRxLookAheadChangePtr* field description of the driver parameter block in Chapter 3, "CHSM Data Structures and Variables" for more information on implementing this control procedure for intelligent adapters.

CMSMPhysNodeAddress Variable

CMSMPhysNodeAddress is used to access the physical layer format of the node address after **CMSMRegisterMLID** has been called.

If the *MM_PHYS_NODE_ADDR_BIT* bit of the *MLIDCFG_ModeFlags* field is set, the CHSM must use *CMSMPhysNodeAddress* to get the physical layer format of the node address instead of the configuration table's *MLIDCFG_NodeAddress*. The CMSM sets the *CMSMPhysNodeAddress* value when the CHSM's initialization routine calls **CMSMRegisterMLID**.

For additional information, refer to the configuration table *MLIDCFG_NodeAddress* and *MLIDCFG_ModeFlags* descriptions in Chapter 3, "CHSM Data Structures and Variables" and the canonical and noncanonical format discussion in *ODI Supplement: Canonical and Noncanonical Addressing*.

Data Structures

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

4-6 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

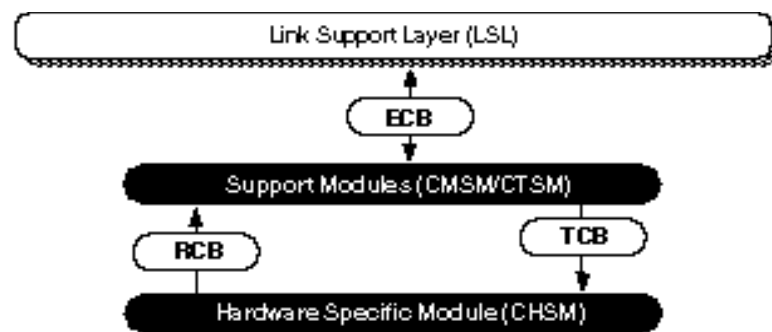
- Receive Control Blocks (RCBs)
- Transmit Control Blocks (TCBs)

These streamlined forms of the general ECB structure (defined in Appendix B, "Event Control Blocks (ECBs)") are provided by the CMSM to simplify CHSM development. Only the fields relevant to the specific packet transaction in progress are visible to the CHSM.

The following sections describe the RCB and the TCB structures. The CHSM must refer to these structures during packet reception and transmission. The relationship of these CMSM structures with the general ECB structure is also discussed. (Figure 4-1 illustrates this relationship.)

Specific reception and transmission methods and related CMSM/CTSM support routines are described in Chapter 5, "CHSM Functions".

Figure 4-1
Packet Transfer through the MLID



CMSM/CTSM Structures and Variables **4-7**

Fragment Structure

A fragment is an area in memory which contains part or all of a packet that is ready for transmission, or it is an area in memory which is ready to contain part or all of a packet on reception. Each such area is described in the fragment structure (FRAGMENT_STRUCT), and each RCB or TCB contains one or more fragment structures.

The fragment structure and fragment structure field descriptions are as follows:

```
typedef struct _FRAGMENT_STRUCT_  
{  
    void          *FragmentAddress;  
    UINT32        FragmentLength;  
} FRAGMENT_STRUCT;
```

Table 4-1
Fragment Structure Field Descriptions

| Name | Type | Description |
|-----------------|--------|---|
| FragmentAddress | void * | Pointer to the fragment buffer. |
| FragmentLength | UINT32 | Length of the buffer pointed to by <i>FragmentAddress</i> . |

Receive Control Blocks (RCBs)

Receive Control Blocks (RCBs) are the structures used to transfer data from the CHSM to the CTSM.

Usually, when the adapter receives a packet, the CHSM passes a pointer to the lookahead data of the CTSM (see <CTSM>**GetRCB** in Chapter 5, "CHSM Functions"). The CTSM filters the lookahead data and then passes the packet to the LSL, which passes it to the protocol stacks. If a protocol stack wants the packet, an RCB is passed back to the CHSM, which fills out the RCB and calls <CTSM>**RcvComplete**.

The CHSM receive routine must be designed to handle multiple fragmented receive buffers for lookahead data. The *MM_FRAG_RECEIVES_BIT* bit of the *MLIDCFG_ModeFlags* field in the configuration table must be set if the CHSM can handle multiple fragmented receive buffers. If the CHSM cannot

handle multiple fragmented receive buffers, the CHSM must use preallocated RCBs along with **<CTSM>ProcessGetRCB**.

The following support routines are available to obtain RCBs.

- **CMSMAllocateRCB**
- **<CTSM>GetRCB**
- **<CTSM>ProcessGetRCB**
- **<CTSM>FastProcessGetRCB**

The **<CTSM>GetRCB** routine can provide fragmented RCBs. CHSMs that cannot handle fragmented receive buffers should use **CMSMAllocateRCB** or **<CTSM>ProcessGetRCB** to obtain RCBs. Chapter 5, "CHSM Functions" describes specific reception methods and illustrates the use of these support routines.

The following describes the RCB structures and fields. These structures are defined in the *cmsm.h* file.

Note



The size of the *RCBReserved* field is defined to preserve the ECB defined fields. (See Appendix B.)

CMSM/CTSM Structures and Variables 4-9

RCB Structure

```
typedef struct _RCB_
{
    union
    {
        {
            UINT8    RWs_i8val[8];
            UINT16   RWs_i16val[4];
            UINT32   RWs_i32val[2];
            UINT64   RWs_i64val;
        } RCBDriverWS;

        UINT8    RCBReserved[(See Table 4-2 for equation.)]
        UNIT32   RCBFragCount;
        FRAGMENT_STRUCT  RCBFragStruct;
    } RCB;
```

Table 4-2
Programmed RCB Field Description

| Name | Type | Description |
|--------------|--|--|
| RCBDriverWS | UINT32[2] | <p>The C HSM can use this field for any purpose, as long as the C HSM controls the RCB.</p> <p>For APIs that deal with linked lists of RCBs, such as CMSMAllocateMultipleRCBs and CMSMReturnMultipleRCBs, the fields RCB->RCBDriverWS.RWs_i32val[0] and RCB->RCBDriverWS.RWs_i32val[1] will contain pointers to the next RCB on the list. RCB->RCBDriverWS.RWs_i32val[0] will contain the logical address of the next RCB, while RCB->RCBDriverWS.RWs_i32val[1] will contain the physical address (when needed). The remaining operation and the description of the RCB will be unchanged.</p> |
| RCBReserved | UINT8 [(UINT32 &(((ECB*)0)-> ECBFragmentCount -(UINT32) &(((ECB*)0)-> ECB_Status)] | <p>The CHSM should not modify this field, except as described in the functions <CTSM>ProcessGetRCB and <CTSM>FastProcessGetRCB. This field contains status indicators, protocol information, and additional data maintained by the CMSM and the LSL.</p> |
| RCBFragCount | UINT32 | <p>The number of data fragment descriptors to follow (1 through 16).</p> |

Table 4-2

Programmed RCB Field Description

| Name | Type | Description |
|---------------|------------------|--|
| RCBFragStruct | FRAGMENT_ STRUCT | An array of fragment descriptors. Each descriptor consists of a pointer to a fragment buffer and the size of that buffer. The CHSM copies the received packet into these buffers. There may be up to 16 fragment descriptors, but there must always be at least one. |

Transmit Control Blocks (TCBs)

Transmit Control Blocks (TCBs) are structures used to transfer data from the CTSM to the CHSM.

When a protocol stack sends a packet, the following tasks are performed:

1. The protocol stack assembles a list of fragment pointers in a transmit ECB.
2. The protocol stack passes the ECB to the LSL.
3. The LSL transfers the ECB to the CTSM
4. The CTSM processes the information and builds a TCB. (The TCB structure consists of the packet header and the data fragment information.)
5. The CTSM directs the TCB to the appropriate CHSM.
6. The CHSM collects the header and the packet fragments and transmits the packet.

The following describes the TCB structures used during packet transmission. The structures are defined in the *csm.h* file.

TCB Structure

```
typedef struct _TCB_FRAGMENT_BLOCK_STRUCT_
{
    UINT32                TCB_FragmentCount;
    FRAGMENT_STRUCT       TCB_Fragment[1];
} TCB_FRAGMENT_BLOCK;

typedef struct _TCB_
{
    void                  *TCB_Reserved;
    UINT32                TCB_BoardNumber;
    UINT32                TCB_DriverWS[3];
    UINT32                TCB_DataLen;
    TCB_FRAGMENT_BLOCK    *TCB_FragBlockPtr;
    UINT32                TCB_MediaHeaderLen;
    union
```



```

    {
        UINT8   TCB_Media[MAX_MEDIA_HEADER_SIZE];
        ETHER_MEDIA_HEADER etherMedia;
        TOKEN_MEDIA_HEADER tokenMedia;
        FDDI_MEDIA_HEADER fddiMedia;
    } TCB_MediaHeader;
} TCB;

```

Table 4-3
TCB Field Descriptions

| Name | Type | Description |
|-------------------|----------------------|--|
| TCB_FragmentCount | UINT32 | 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 CHSM collects the data from these buffers when forming a packet for transmission. |
| TCB_Fragment[1] | Fragment_Struct | The first fragment structure. The maximum number of TCB fragment entries allowed is 16. (See Table 4-1 "Fragment Structure Field Descriptions".) |
| TCB_Reserved | void* | Reserved; must be set to zero. |
| TCB_Boardnumber | UINT32 | The logical board that the TCB is being transmitted on. In general, CHSMs do not use this field. |
| TCB_DriverWS | UINT32[3] | The CHSM can use this field for any purpose, as long as the CHSM controls the TCB. |
| TCB_DataLen | UINT32 | The length of the packet described by the data fragments plus the media header. This value will never be 0. |
| TCB_FragBlockPtr | TCB_FRAGMENT_BLOCK * | This field contains a pointer to a list of fragments defined by TCB_FRAGMENTSTRUCT. |

CMSM/CTSM Structures and Variables **4-13**

Table 4-3
TCB Field Descriptions *continued*

| Name | Type | Description |
|--------------------|------------------------------|---|
| TCB_MediaHeaderLen | UINT32 | The length of the media header that immediately follows the TCB in memory. This value can be odd, even, or 0. A value of 0 indicates a raw send. If the CHSM is handed a raw send, the originating protocol stack has already included the media header in the first data fragment. |
| TCB_MediaHeader | UINT8[MAX_MEDIA_HEADER_SIZE] | A buffer containing the media header that was assembled by the CTSM. |

Media header structure definitions are as follows:

```
typedef struct  _ETHER_MEDIA_HEADER_  
{  
    NODE_ADDR      MH_Destination;  
    NODE_ADDR      MH_Source;  
    UINT16         MH_Length;  
    UINT8          MH_DSAP;  
    UINT8          MH_SSAP;  
    UINT8          MH_Ctrl0;  
    UINT8          MH_SNAP[5];  
} ETHER_MEDIA_HEADER;  
  
typedef struct  _TOKEN_MEDIA_HEADER_  
{  
    UINT8          MH_AccessControl;  
    UINT8          MH_FrameControl;  
    NODE_ADDR      MH_Destination;  
    NODE_ADDR      MH_Source;  
    UINT8          MH_DSAP;  
    UINT8          MH_SSAP;  
    UINT8          MH_Ctrl0;  
    UINT8          MH_SNAP [5];
```

```

    } TOKEN_MEDIA_HEADER;

typedef struct _FDDI_MEDIA_HEADER_
{
    UINT8          MH_FrameControl;
    NODE_ADDR      MH_Destination;
    NODE_ADDR      MH_Source;
    UINT8          MH_DSAP;
    UINT8          MH_SSAP;
    UINT8          MH_Ctrl0;
    UINT8          MH_SNAP [5];
} FDDI_MEDIA_HEADER;

```

CMSM_CONFIG_TABLE

```

typedef struct _CMSM_CONFIG_TABLE_
{
    UINT32  CMSMCFG_TableSize;
    UINT8   CMSMCFG_TableMajorVersion;
    UINT8   CMSMCFG_TableMinorVersion;
    UINT8   CMSMCFG_ModuleMajorVersion;
    UINT8   CMSMCFG_ModuleMinorVersion;
    UINT8   CMSMCFG_ODISpecMajorVersion;
    UINT8   CMSMCFG_ODISpecMinorVersion;
    UINT16  CMSMCFG_Reserved;
    UINT32  CMSMCFG_MaxNumberOfBoards;
    UINT32  CMSMCFG_SystemFlags;
}CMSM_CONFIG_TABLE;

```

CMSMCFG_TableSize

The actual size of the C MSM configuration table (CMSM_CONFIG_TABLE). The value of this field should not be confused with the number of bytes requested or copied (such as nBytes).

CMSMCFG_TableMajorVersion

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

CMSMCFG_TableMinorVersion

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

CMSMCFG_ModuleMajorVersion

This field contains the major version of the C MSM binary (i.e., CMSM.NLM).

CMSMCFG_ModuleMinorVersion

This field contains the minor version of the C MSM binary (i.e., CMSM.NLM).

CMSMCFG_ODISpecMajorVersion

This field contains the major version of the ODI Specification that this version of the C MSM is written too. For example, if the version of the ODI specification is 1.11, the value of this field is 1.

CMSMCFG_ODISpecMinorVersion

This field contains the minor version of the ODI Specification that this version of the C MSM is written too. For example, if the version of the ODI specification is 1.11, the value of this field is 11.

CMSMCFG_Reserved

This field is reserved.

CMSMCFG_MaxNumberOfBoards

The value of this field represents the maximum number of boards the C MSM supports.

CMSMCFG_SystemFlags

The bits in this field are defined below.

Bits 0-29 Reserved

These bits are reserved.

Bit 30 CMSM_CFG_SERVER_BIT

When set to 1 this bit indicates the C MSM is running in a server environment. This bit is mutually exclusive with bit 31.

Bit 31 CMSM_CFG_CLIENT_BIT

When set to 1 this bit indicates the C MSM is running in a client environment. This bit is mutually exclusive with bit 30.

CTSM_CONFIG_TABLE

```
typedef struct _CTSM_CONFIG_TABLE_
{
    UINT32    CTSMCFG_TableSize;
    UINT8     CTSMCFG_TableMajorVersion;
    UINT8     CTSMCFG_TableMinorVersion;
    UINT8     CTSMCFG_ModuleMajorVersion;
    UINT8     CTSMCFG_ModuleMinorVersion;
    UINT8     CTSMCFG_ODISpecMajorVersion;
    UINT8     CTSMCFG_ODISpecMinorVersion;
    UINT16    CTSMCFG_Reserved;
    UINT32    CTSMCFG_MaxFrameSize;
    UINT32    CTSMCFG_SystemFlags;
}CTSM_CONFIG_TABLE;
```

CTSMCFG_TableSize

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

CTSMCFG_TableMajorVersion

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

CTSMCFG_TableMinorVersion

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

CTSMCFG_ModuleMajorVersion

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

CTSMCFG_ModuleMinorVersion

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

CTSMCFG_ODISpecMajorVersion

This field contains the major version of the ODI Specification that this version of the <CTSM> is written too. For example, if the version of the ODI specification is 1.11, the value of this field is 1.

CTSMCFG_ODISpecMinorVersion

This field contains the minor version of the ODI Specification that this version of the <CTSM> is written too. For example, if the version of the ODI specification is 1.11, the value of this field is 11.

CTSMCFG_Reserved

This field is reserved.

CTSMCFG_MaxFrameSize

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

CTSMCFG_SystemFlags

The bits in this field are defined below.

Bits 0-29 Reserved

These bits are reserved.

Bit 30 CTSM_CFG_SERVER_BIT

When set to 1 this bit indicates the C TSM is running in a server environment. This bit is mutually exclusive with bit 31.

Bit 31 CTSM_CFG_CLIENT_BIT

When set to 1 this bit indicates the C TSM is running in a client environment. This bit is mutually exclusive with bit 30.

DRIVER_OPTION Structure

```

typedef struct _DRIVER_OPTION_
{
    struct _DRIVER_OPTION_ *Link;
    MEON_STRING *ParseString;
    union
    {
        void *OptionPtr;
        int Min
    } Parameter0;
    union
    {
        UINT32 Range;
        int Max;
    } Parameter1;
    union
    {
        int Default;
        MEON_STRING *StringDefault;
    } Parameter2;
    UINT16 Type;
    UINT16 Flags;
    MEON_STRING String[MAX_PARAM_LEN];
} DRIVER_OPTION;

```

Field descriptions

On entry to the parser, the value of *Parameter0*, *Parameter1*, and *Parameter2* is based on whether the *Range*, *Enumeration*, or *String* bit of the *Flags* field is set.

Table 4-4**Interpretation of Parameter0, Parameter1, and Parameter2**

| Flags Field Bit | Parameter0 Value | Parameter1 Value | Parameter2 Value |
|-----------------|------------------|------------------|------------------|
| Range | Min | Max | Default |
| Enumeration | OptionPtr | Range | N/A |
| String | N/A | N/A | StringDefault |

Table 4-4
Interpretation of Parameter0, Parameter1, and Parameter2

| Flags Field Bit | Parameter0 Value | Parameter1 Value | Parameter2 Value |
|-------------------|------------------|------------------|------------------|
| None of the above | N/A | N/A | Default |

Link
Pointer to the next DRIVER_OPTION structure (NULL if it is the last one).

ParseString
See the "ParseString" heading below.

Min
The lower bound of the range the user can enter.

OptionPtr
Pointer to a structure containing the allowed values for this parameter. The first field of the structure is of type UINT32 and contains the number of values. The other field of the structure is an array of the type to match the format specifier in the *ParseString* and *Type*. The array contains the values (or pointers to the values for the string type). The first element of the array is the default.

When there is an option structure, the value of *Parameter2* and the value of the default bit in the *Flags* field are ignored.

Max
This UINT32 field specifies the upper bound of the allowed range that the user can enter. On return from the parser, if the parameter is of any numeric type, the field contains the value selected/defined by the user. If the parameter is a keyword, *Max* is returned with a value of 1. If the parameter is not a keyword or a numeric type, *Max* is returned with a value of 0.

Range
This UINT32 field, if used, contains the length or range associated with this option. Typically, this field is used in specifying memory decode ranges and port lengths.

Default
This *int* field, if used, contains the default value of the parameter. The *Default* bit in the *Flags* field controls whether this field is used or not.

When there is an option structure, the value of *Parameter2* is ignored.

StringDefault

This *char* field, if used, contains the default value of the parameter. The *Default* bit in the *Flags* field controls whether this field is used or not.

Type

This is a UINT16 field that contains a code indicating the option type. The following is a list of possible values for this field:

| | |
|------------------|----------------|
| 0x0000 | CUSTOMPARAM |
| 0x0001 | INTPARAM |
| 0x0002 | INT1PARAM |
| 0x0003 | PORTPARAM |
| 0x0004 | PORT1PARAM |
| 0x0005 | DMAPARAM |
| 0x0006 | DMA1PARAM |
| 0x0007 | MEMPARAM |
| 0x0008 | MEM1PARAM |
| 0x0009 | SLOTPARAM |
| 0x000A | NODEPARAM |
| 0x000B | CHANNELPARAM |
| 0x000C | FRAMEPARAM |
| 0x000D | Reserved |
| 0x000E | NAMEPARAM |
| 0x000F | RETRIESPARAM |
| 0x0010 | BELOW16PARAM |
| 0x0011 | BUFFERS16PARAM |
| 0x0012 to 0xFFFF | Reserved |



Note After calling the parser, DRIVER_OPTION records defined as type PORTPARAM, PORT1PARAM, MEMPARAM, and MEM1PARAM will have their Parameter1 (Range or Maximum) fields overwritten. Before using the record again (such as loading another frame type), the CHSM must reset the Parameter1 (Range or Maximum) fields.

Flags

This is a UINT16 field that contains a bitmap indicating the status of the option. The following is a list of bits in this field:

| | |
|----------------|--|
| OPTIONALPARAM | Optional—if not specified on the command line, the option is ignored. |
| REQUIREDPARAM | Required—if not specified on the command line, prompt the user. |
| DEFAULTPRESENT | Default—The default value for this parameter is contained in <i>Parameter2</i> of the DRIVER_OPTION structure. When the ENUMPARAM is set, this bit is ignored. |
| KEYWORDPARAM | Keyword—if this bit is set, the keyword is not followed by a value. If the keyword is present, <i>Max</i> is returned with a value of 1. If the keyword is not present, <i>Max</i> is returned with a value of 0. If KEYWORDPARAM is used, DEFAULTPRESENT and REQUIREDPARAM are ignored. |
| ENUMPARAM | Enumeration—if this bit is set: DRIVER_OPTION. <i>Parameter0</i> is a pointer to an <i>OptionsList</i> array containing the list of allowed values for this parameter. |
| RANGEPARAM | Range—if this bit is set: DRIVER_OPTION. <i>Min</i> is the minimum allowed value of the range and DRIVER_OPTION. <i>Max</i> is the maximum allowed value. |
| STRINGPARAM | String—if this bit is set, the parameter is of type string. This bit must be set if, and only if, there is a %s or %c format specifier in <i>ParseString</i> . |
| SHARABLE | Sharable option—such as shared interrupts. |

Interpretation of the Flags Field in the DRIVER_OPTION Structure

KEYWORDPARAM, ENUMPARAM, RANGEPARAM, and STRINGPARAM are mutually exclusive. Only one of these bits can be set at a time, but it is not required to set any of the bits. If the ENUMPARAM and RANGEPARAM bits are not set, any value of the appropriate type may be entered by the user.

OPTIONALPARAM and REQUIREDPARAM are mutually exclusive and one of the bits is required unless the KEYWORDPARAM bit is present. KEYWORDPARAM implies that it is an optional parameter.

The DEFAULTPRESENT is valid for RANGEPARAM and STRINGPARAM. It is also valid if none of the KEYWORDPARAM, ENUMPARAM, RANGEPARAM, and STRINGPARAM bits are present.

Function of the DEFAULTPRESENT Bit

The DEFAULTPRESENT bit is basically used to determine how prompting is handled. There are two major cases:

DEFAULTPRESENT and OPTIONALPARAM
DEFAULTPRESENT and REQUIREDPARAM

DEFAULTPRESENT and OPTIONALPARAM

The parameter is not present on the command line; the user is not prompted and ODISTAT_ITEM_NOT_PRESENT is returned.

The parameter is present on the command line and the parameter is valid: the parameter is used as is.

The parameter is present on the command line and the parameter is invalid: the user is prompted with the default value as the default input.

DEFAULTPRESENT and REQUIREDPARAM


The parameter is not present on the command line: the user is prompted with the default value as the default input.


The parameter is present on the command line and the parameter is valid: the parameter is used as is.

The parameter is present on the command line and the parameter is invalid: The user is prompted with the default value as the default input.

String

On entry, this field must contain a pointer to a NULL terminated buffer, which is large enough to contain the largest expected user input to this parameter. On return from the parser, this buffer contains a NULL terminated string corresponding to the value selected or entered by the user.

Note  Double byte charaters are not allowed.

Note  Prompt strings must be limited to 512 bytes.

ParseString Field

The DRIVER_OPTION structure format string controls how the parser scans and converts the MLID's parameters. The format string is a character string composed of three types of objects:

- Whitespace characters
- Keyword characters
- A format string

The following is the format of the parse string:

`[whitespace]keyword[whitespace]=[whitespace]conversion specifier[whitespace]`

Whitespace Characters

The whitespace characters are blank, tab (\t), and newline (\n). If the parser encounters a whitespace character in the format string, it reads and skips all subsequent whitespace characters in the input until it finds an ordinary character.

Keyword Characters

The keyword characters are the alphanumeric ASCII characters plus the underscore and period. Any keyword characters found are assumed to be the parameter keyword for this parameter. The parser uses these characters in the format string to search the user input and to find the appropriate value for this parameter.

Conversion Specifiers

The conversion specifier directs the parser function to read and convert characters from the input field into specific value types. The maximum conversion specifier length is 80 characters including the NULL termination.

The **CMSMParseDriverParameters** conversion specifier has the following format:

```
% [width] type_character
```

where:

% is the character used to begin each conversion specifier, and

width is the width specifier (optional). It is the maximum number of characters to read. If the function encounters a whitespace or unconvertible character, fewer characters may be read.

Type Characters

The **type_character** specifier represents the type character. Table 7.2 lists the parser type characters, the input type expected by each type character, and the input storage form.

The information in the table assumes that no width specified is included in the conversion specifier. To see how the addition of the width specifier affects the parser input, see the width specifier section below.

Table 4-5
Input and Results for Each Character Type

| Type Character | Expected Input | Type of Result |
|----------------|--|----------------|
| Numbers | | |
| d | Decimal integer | 32-bit integer |
| D | Decimal integer | 32-bit integer |
| u | Unsigned decimal integer | UINT32 |
| U | Unsigned decimal integer | UINT32 |
| x | Hexadecimal integer | UINT32 |
| X | Hexadecimal integer | UINT32 |
| Characters | | |
| s | Whitespace-terminated character string | MEON_STRING * |
| c | Character | MEON |

In order to allow the input of string parameters that include white space, it is permissible to have multiple %[width]s conversion specifiers in a parse string. In such a case, only %s, %[search set], and %c conversion specifiers are allowed.



White space between conversion specifiers is not allowed.

The effect of this is to input as many string fields as there are %s conversion specifiers and concatenate them together with one space *char* between each. If there are more %s conversion specifiers than user supplied fields, the result is platform dependant, but will not include anything not entered as a parameter by the user. It is the CHSM's responsibility to parse the returned string and ignore any fields beyond the end of its parameter.

Input Fields

The following are considered input fields:

- All characters up to, but not including, the next whitespace character.
- All characters up to the first one that cannot be converted under the current conversion specifier.
- Up to *n* characters, where *n* is the specified field width.

Conventions

Certain conventions accompany some of these conversion specifiers. These conventions are described in the following paragraphs.

%[search_set] Conversion The set of characters surrounded by square brackets can be substituted for the s-type character.

The *search_set* variable represents a set of characters that define a search set of possible characters making up the string (the input field).

If the first character in the brackets is a carat (^), the search set is inverted to include all ASCII characters except those between the square brackets.

The input field is a string that is not delimited by whitespace. The parser reads the corresponding input field up to the first character that does not appear in the search set (or in the inverted search set). The following is an example of two of these types of conversion:

| | |
|---------|---|
| [abcd] | Searches for characters a, b, c, and d in the input field. The search terminates when it encounters the first character not in the <i>search_set</i> . |
| [^abcd] | Searches for any characters except a, b, c, and d in the input field. The search terminates when it encounters any character in the <i>search_set</i> . |

Width Specifiers

The width specifier (*n*), a decimal integer, specifies the maximum number of characters that can be read from the current input field.

If the input field contains fewer than n characters, the parser reads all the characters in the field.

If a whitespace or nonconvertible character occurs before n characters are read, the characters up to that character are read, converted, and stored.

A nonconvertible character is one that cannot be converted according to the given format (such as a "L" or "q" when the format is decimal).

chapter **5** **CHSM Functions**

Overview

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

Initialization and Removal

- DriverInit
- DriverRemove

Board Service

- DriverISR
- DriverPoll (optional)

Packet Transmission

- DriverSend

Control Functions

- DriverReset
- DriverShutdown
- DriverMulticastChange
- DriverPromiscuousChange (recommended)
- DriverStatisticsChange (optional)
- DriverRxLookAheadChange (optional)

- DriverGetMulticastInfo
- DriverManagement (optional)
- DriverEnableInterrupt
- DriverDisableInterrupt

Timeout Detection

- DriverAES (optional)

The CHSM may require the optional routines, depending on the adapter. The CHSM indicates unsupported, optional routines by placing a 0 in the corresponding fields of the driver parameter block. Additional procedures might also be needed for specific hardware requirements.

All functions described on the following pages are calls from the CMSM and the CTSM to the CHSM.

Important



In order to achieve operating system and platform independence, the CHSM must not enable or disable system interrupts.

Initialization

The CHSM's **DriverInit** routine controls the complete initialization process. However, specific tasks performed during initialization are handled by CMSM routines or CTSM routines.

The initialization tasks are as follows:

- Allocate the frame and driver 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 MLID with the LSL
- Set up a board service routine
- Schedule timeout callbacks

Registering with the CMSM/CTSM

Before the C HSM calls **<CTSM>RegisterHSM** or **CMSMInitParser**, **DriverInit** must take the following steps:

- Copy its incoming parameters into a CHSM_STACK structure. (The CHSM_STACK structure is defined in *cmsm.h* and in the "Structure Definitions" section of the "Preface" of this document).
- Copy the address of the CHSM_STACK structure (*chsmStack*) into *DriverInitParmPointer*.

After it has done this, **DriverInit** must call **CMSMInitParser** to initialize the parser. This must be done before any other CMSM or CTSM APIs are called.

The CHSM sets *MLIDCFG_MaxFrameSize* to the largest frame size supported by the adapter. The CHSM sets the *MLIDCFG_SharingFlags* *MS_SHUTDOWN_BIT* to 1. **DriverInit** calls **<CTSM>RegisterHSM** with a

pointer to the driver parameter block. The CTSM passes the CHSM's parameter block pointer along with its own pointer to the CMSM.

The CMSM makes a local copy of both parameter blocks and processes the information passed from the operating system. If the CHSM has custom firmware, the CMSM loads the firmware and initializes the *DriverFirmwareSize* and *DriverFirmwareBuffer* fields as described in *Chapter 3: CHSM Data Structures and Variables*.

The CMSM allocates memory for the frame data space and creates a copy of the CHSM's configuration table template in that area. If the *MLIDCFG_CardName* and *MLIDCFG_DriverMajorVer* fields of the configuration table are initialized to 0, the CMSM will fill in these fields and the *MLIDCFG_DriverMinorVer* field using information derived from the linker definition file. If the CHSM has placed nonzero values in the card name and driver version fields, these fields will not be modified.

Finally, the CMSM may set the *MLIDCFG_MaxFrameSize* field of the configuration table to a smaller packet size (depending upon platform specifics) and then return to **DriverInit**.

- If the CMSM is unsuccessful in its initialization tasks, it returns an error code. **DriverInit** should then return an error code.
- If the CMSM is successful, it returns with *ODISTAT_SUCCESSFUL* and *configTable* pointing to the MLID's new frame data space. The CHSM must now gather the hardware option information needed for the configuration table and call the CMSM to parse the MLID parameters entered on the command line.

Determining Hardware Options

After <CTSM>**RegisterHSM** returns successfully, the CHSM must determine the hardware configuration of the adapter. This includes the following parameters:

- Base port for programmed I/O adapters
- Memory decode addresses for shared RAM adapters
- Interrupt numbers
- DMA channels

5-4 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

For machines with bus types that support standard retrievable product IDs, such as EISA, PCI, Micro Channel, PnP ISA, or PC Card; the MLID can get hardware configuration information directly from the system using CMSM calls once the Hardware Instance Number (HIN) has been identified.

For EISA buses and Micro Channel buses, it is possible to uniquely identify an adapter by its physical slot number. However, this is not possible for new buses such as PCI and PnP ISA. These buses can have multiple functions or multiple devices present on a single adapter, and, in the cases of some bus configurations, such as PCI BIOS v2.0 and PNP ISA, the buses have no physical slot correlation scheme.

The CHSM now uses the *slot* parameter to contain the Hardware Instance Number (HIN). The HIN is a system-wide, bus-independent, unique handle for a device. The HIN enables the CHSM to identify functions and devices on multiple device adapters as well as single device adapters and integrated motherboard devices.

For single device adapters such as EISA, Micro Channel, and PCI BIOS v2.1, the HIN is the physical slot number, unless there is a physical slot conflict, such as with multibus systems.

In the following cases, the hardware instances are assigned unique values:

- Integrated motherboard devices
- PCI BIOS v2.0 devices
- PCI BIOS v2.1 multiple device adapters
- PnP ISA devices
- Physical slot number conflicts

To identify the required hardware parameters, **DriverInit** must perform the following steps (where appropriate for the hardware):

1. If the CHSM supports an adapter with a product ID that is retrievable according to a standard, such as EISA, PCI, Micro Channel, or PC Card, the CHSM should do the following:
 - Scan for the adapter for each supported bus type using **CMSMSearchAdapter**. **CMSMSearchAdapter** returns a bus tag and a unique identifier for each hardware instance found. This pair

CHSM Functions 5-5

of values is used by **CMSMGetInstanceNumber**, which must be called once for each hardware instance found. The HIN numbers returned by **CMSMGetInstanceNumber** are placed in the Slot DRIVER_OPTION structure, and **CMSMParseDriverParameters** is called. The selected HIN is put in the *MLIDCFG_Slot* field of the configuration table.

- Call **CMSMGetInstanceNumberMapping** with the configuration table *MLIDCFG_Slot* field as an input parameter. The corresponding bus tag and unique identifier are returned. The bus tag returned from **CMSMGetInstanceNumberMapping** must be placed in the *MLIDCFG_DBusTag* field of the configuration table.
- Call **CMSMGetCardConfigInfo** to get the adapter's configuration and fill out the I/O portion of the configuration table. (The bus tag and unique identifier for the selected HIN are used as input parameters to **CMSMGetCardConfigInfo**.)

If the CHSM supports a bus adapter that does not have a product ID that is retrievable by some standard, it must define a set of DRIVER_OPTION structures that will cause the parser to get whatever hardware configuration information the CHSM needs, but cannot get for itself. Legacy ISA is an example of such a bus.

2. If the CHSM needs certain parameter values to determine other parameter values, it should call **CMSMParseSingleParameter** before it calls **<CTSM>RegisterHSM**. **CMSMParseSingleParameter** may be called multiple times as needed.

CMSMParseDriverParameters must contain the valid hardware configuration options in the DRIVER_OPTION structure.

CMSMParseDriverParameters or **CMSMParseSingleParameter** can be used to process custom command line keywords. However, for automatic driver selection and loading to function properly in an advanced installation environment, the use of custom command line keywords should be restricted.

For family drivers that support adapters of more than one bus type -- including the legacy ISA bus--the custom command line keyword *ISA* should be used to differentiate between a legacy ISA bus hardware instance and an advanced bus hardware instance, such as EISA, Micro Channel, PCI, PnP ISA, and PC Card.

After parsing the parameters, the I/O portion of the logical board's configuration table in the frame data space has been filled in by **CMSMParseDriverParameters** or the CHSM and now contains the selected adapter HIN in the *MLIDCFG_Slot* field, and its *busTag* in the *MLIDCFG_DBusTag* field.

Important



DriverInit must call **CMSMParseDriverParameters** at least once, but never more than once.

When the CHSM has obtained all needed information for the configuration table, **DriverInit** calls **CMSMRegisterHardwareOptions**.

Important



If the MLID must access shared memory before registering the hardware options, it must use **CMSMReadPhysicalMemory**.

Registering Hardware Options

The CHSM calls **CMSMRegisterHardwareOptions** to register with the operating system. This routine reports to the CHSM whether a new adapter or a new frame format for an existing adapter is being loaded. If a new adapter is being registered, the CMSM allocates the driver data space and copies the MLID's DRIVER_DATA to that area. This routine also notifies the CHSM of any conflicts with existing hardware in the system.

There are four possible conditions that the CHSM must handle on return from **CMSMRegisterHardwareOptions**.

- If the returned value equals *REG_TYPE_NEW_ADAPTER*, a new adapter was successfully registered, and the CHSM must proceed with the hardware initialization (the *driverData* parameter in **CMSMRegisterHardwareOptions** now contains a pointer to the driver data space).
- If the returned value equals *REG_TYPE_NEW_FRAME*, a new frame type for an existing adapter was successfully registered, and initialization is essentially complete.
- If the returned value equals *REG_TYPE_NEW_CHANNEL*, a new channel for an existing multichannel adapter was successfully registered. The CMSM typically treats the registering of a new channel as a new adapter. The CHSM must proceed with the hardware initialization. (The *driverData* parameter in **CMSMRegisterHardwareOptions** now contains a pointer to the CHSM's driver adapter data space.)

CHSM Functions 5-7

- If the returned value equals *REG_TYPE_FAIL*, the CMSM was unable to register the hardware options. **DriverInit** should return with an error code.

Initializing the Adapter

At this point the CHSM initializes the adapter hardware. This consists of all software controlled configuration of the hardware, and may also include hardware tests and diagnostics such as RAM testing.

The **DriverReset** routine can be called to handle part of this procedure since it performs steps needed to initialize the hardware.

At this point, the CHSM examines *MLIDCFG_MaxFrameSize* and adjusts it down if necessary.

Important



DriverInit should set up the correct number of transmit buffers (the maximum number of simultaneous sends allowed by the hardware) by placing an appropriate value in *CMSMTxFreeCount*. A description of this variable is in Chapter 4, "CMSM/CTSM Data Structures and Variables", and information about its use is in the packet transmission section of this chapter. If firmware is to be loaded down to the adapter, it should be done at this point.

If an error occurs during the hardware initialization, **DriverInit** should print an appropriate error message, call **CMSMReturnDriverResources**, and return to the operating system with a *ODISTAT_FAIL* value.

If the hardware initializes successfully, the CHSM sets the *MLIDCFG_SharingFlags MS_SHUTDOWN_BIT* to zero, produces a NESL Service Resume MLID Card Insertion Complete event, using *CMSMNESLProduceMLIDEvent*, and returns *ODISTAT_SUCCESSFUL* to the operating system.

Registering with the LSL

DriverInit calls the **CMSMRegisterMLID** routine to register the MLID with the LSL. **CMSMRegisterMLID** registers the MLID by passing the addresses of the following items to the LSL:

- CMSM Send Routine
- CMSM Control Handler Routine

5-8 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

- Pointer to the CHSM configuration table (returned by **<CTSM>RegisterHSM**)

The LSL assigns a logical board number to the adapter, and the CMSM places it in the configuration table. The CMSM automatically registers a logical board with the LSL during **CMSMRegisterHardwareOptions** each time a new frame is added for an existing adapter. If an error occurs, the CMSM routine returns **ODISTAT_OUT_OF_RESOURCES**.

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



CMSMTxFreeCount and **CMSMPriorityTXFreeCount** must be set before calling **CMSMRegisterMLID**.

Setting up a Board Service Routine

The CHSM registers its board service routine(s) (**DriverISR**, **DriverISR2**, or **DriverPoll**) by calling either **CMSMSetHardwareInterrupt** or **CMSMEnablePolling**. The **DriverISR** description later in this chapter provides special instructions on setting up and handling shared interrupts.



The adapter must be ready to process interrupts before calling **CMSMSetHardwareInterrupt** or **CMSMEnablePolling**. Polling HSMs can use **CMSMGetPollSupportLevel** to find out the level of support provided on the platform.

Scheduling Timeout Callbacks

If the CHSM is running an interrupt driven adapter, it can schedule a timer event to check if a board is unable to complete a send. To schedule a timer event, the CHSM calls **CMSMScheduleAES**, which schedules periodic calls to the CHSM's **DriverAES** routine.



CMSMScheduleAES can not be called until after the CHSM has called **CMSMRegisterMLID**.

If the adapter is not interrupt driven, the polling routines can check if a board is unable to complete a send.

Driver Removal

The operating system calls the MLID’s exit function, **DriverRemove**, when it receives the command to unload the MLID.

DriverInit

The initialization routine is called by the loader when it loads the CHSM.

Syntax

```
ODISTAT  DriverInit  (
    MODULE_HANDLE      *ModuleHandle,
    SCREEN_HANDLE      *ScreenHandle,
    MEON_STRING         *CommandLine,
    MEON_STRING         *ModuleLoadPath,
    UINT32              UninitializedDataLength,
    void               *CustomDataFileHandle,
    UINT32              (* FileRead ) (void
                                     *FileHandle,UINT32 FileOffset,
                                     void *FileBuffer,UINT32
                                     FileSize ),
    UINT32              CustomDataOffset,
    UINT32              CustomDataSize,
    UINT32              NumMsgs,
    MEON_STRING         **Msgs);
```

Input Parameters

- ModuleHandle*
Identifies your initialization routine. Your initialization routine must provide this handle when calling many of the operating system support routines for MLIDs.
- ScreenHandle*
Your initialization routine must use this handle during the *OutputToScreen* function to perform any screen I/O. This handle is not valid after initialization.

CommandLine

Pointer to the command line that was used to load the driver. This parameter is passed to **CMSMParseDriverParameters** to get the hardware configuration information from the command line.

ModuleLoadPath

Path used to load the MLID, including the module name.

UninitizedDataLength

Used by the operating system to determine the data image length.

CustomDataFileHandle

The custom data file is appended to the end of your NLM. Because the NLM was opened during loading, this handle points to a structure that the operating system uses to read the custom data file. This value is provided as a parameter to *FileRead*.

FileRead

Pointer to a read function that *DriverInit* can use to read auxiliary files.

CustomDataOffset

The starting offset of the custom data inside the .NLM (or .LAN) file. This value is provided as a parameter to *FileRead*.

CustomDataSize

The length of the custom data file. This value is provided as a parameter to *FileRead*.

NumMsgs

Number of message strings in your module.

Msgs

Pointer to an array of pointers of MEON_STRING that is used by the message enabling macros for handling messages.

Output Parameters

None.

Return Values

| | |
|--------------------|---|
| ODISTAT_SUCCESSFUL | The CHSM initialized successfully. |
| ODISTAT_FAIL | The CHSM failed to initialize successfully. |

Remarks

When the loader receives the command to load the CHSM, it calls the **DriverInit** routine (specified as the "start" routine in the CHSM's linker definition file; see Appendix C).

If the initialization was successful **DriverInit** must produce a NESL Service Resume Event for MLID Card Insertion Complete.

DriverRemove

Causes the CHSM to return its resources prior to being unloaded.

Syntax

```
void DriverRemove ( );
```

Input Parameters

None.

Output Parameters

None.

Return Values

None.

Remarks

The **DriverRemove** routine is called whenever the CHSM is unloaded. The CHSM's linker definition file must include the keyword "exit" followed by **DriverRemove**.

This routine calls **CMSMDriverRemove** with the value of *DriverModuleHandle* from the driver parameter block.

Note



CMSMDriverRemove triggers a call to **DriverShutdown** to permanently shutdown the CHSM. This function does not need to return resources that are returned by **DriverShutdown**.

Board Service Routine

The board service routine detects and handles receive events and transmit complete events. The MLID is notified of these events by an interrupt service routine (**DriverISR**) or a polling routine (**DriverPoll**).

Packet Reception

This section provides a brief overview of various reception methods available to the developer, followed by details of the **DriverISR** and **DriverPoll** routines.

Reception Methods

The method of packet reception selected is typically dependent on the adapter's data transfer method.

Reception Method—Option 1

This is the simplest and the most preferred reception method for host DMA and bus master adapters. The **<CTSM>ProcessGetRCB** function in *Chapter 6: CTSM Functions* provides a detailed description of this process. The steps performed in this reception method are outlined below:

1. The CHSM calls **CMSMAllocateRCB** to get an RCB (unless it already has one from Step 5 below). If the *MM_FRAGS_PHYS_BIT* bit of the *MLIDCFG_ModeFlags* field is set, this call returns physical addresses in the fragment list.
2. The CHSM copies the received packet into the RCB.
3. The CHSM calls **<CTSM>ProcessGetRCB**.
4. The CTSM checks the frame header information and fills in the remainder of the RCB fields.
5. The CTSM returns the RCB to the operating system and gets a new RCB for the CHSM. If no RCB is available, it returns 0.

Reception Method—Option 2

This is the preferred reception method for shared RAM and programmed I/O adapters. This method involves using a lookahead process, in which the frame header information is confirmed before the entire packet is transferred from the adapter into an RCB. During initial development, it might be helpful to use Option 1 to create a functioning CHSM, and then implement Option 2.

The adapter's data transfer mode determines how the lookahead process is handled. If a programmed I/O adapter is used, the *CMSMMaxFrameHeaderSize* variable is the number of bytes to be read into a lookahead buffer. If the adapter uses a shared RAM transfer mode, the lookahead buffer is the start of the packet in shared RAM.

The **<CTSM>GetRCB** function in Chapter 6, "CTSM Functions" provides a detailed description of this process. The steps performed for this reception method are outlined below:

1. The CHSM sets up a lookahead buffer.
2. The CHSM calls **<CTSM>GetRCB** with a pointer to the lookahead buffer.
3. The CTSM filters the packet and frame header and passes the lookahead data to the LSL. If a protocol stack wants the packet, an RCB is returned.
4. The CHSM copies the remainder of the packet into the RCB and calls **<CTSM>RcvComplete**. If no RCB is returned, the CHSM checks for another receive packet to send up.

Reception Method—Option 3

This method is recommended for intelligent adapters that are designed to be "ECB aware." (See Appendix B for more information on ECBs.) This method dramatically reduces the load on the system 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 as follows:

1. The CHSM obtains an ECB by calling **CMSMAllocateRCB** and queues it until it is needed for a received packet. Be careful not to preallocate too many ECBs.

2. The firmware filters the frame header information and all fields of the ECB as described in Appendix B, "Event Control Blocks (ECBs)".
3. The CHSM calls **<CTSM>RcvComplete** to return the ECB after it is completely filled in.

Reception Method—Option 4

The pipelined adapter can be configured to interrupt prior to receiving a complete packet. At driver initialization time, the adapter must be able to be configured to wait until it has received at least the **CMSMMaxFrameHeaderSize** before it interrupts.

1. The CHSM sets up a LookAhead buffer.
2. The CHSM calls **<CTSM>GetRCB** with *packetSize* set to **UNUSED** before it has received the entire packet.
3. The CTSM checks the frame header information and passes the LookAhead data to the LSL. (The CTSM cannot fill in all the LookAhead fields with definitive values such as error bits and length fields.)
4. The CHSM copies the remainder of the packet into the RCB and calls **<CTSM>RCVCompleteStatus**. If no RCB is returned, the CHSM checks for another receive packet to send.

Using Shared Interrupts

A CHSM can support shared interrupts, provided that they are also supported by the host bus and the adapters that 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 following list describes how some buses handle interrupts:

- The PCI and Micro Channel buses always use level-triggered interrupts and can support shared interrupts.
- The ISA bus normally uses edge-triggered interrupts and will not support shared interrupts unless external logic exists on the adapters for sharing the interrupt.

- The EISA bus normally uses edge-triggered interrupts, but each interrupt can be individually configured to level-triggered mode in order to support shared interrupts.
- Other buses vary in their use of edge and level triggered interrupts.

The CHSM must indicate that the adapters are sharing interrupts by setting the *MS_SHARE_IRQ0_BIT* bit in the *MLIDCFG_SharingFlags* field of the configuration table. The CHSM must also initialize the driver parameter block variable, *DriverEndOfChainFlag*, as described in the following table.

The CMSM will call **DriverDisableInterrupt** and **DriverEnableInterrupt** to discover which CHSM(s) need to service their adapters and invoke the ones that do.

Table 5-1
DriverEndOfChainFlag Values

| If the CHSM: | The CHSM must: | DriverEndOfChainFlag value: |
|------------------------------------|--|---|
| Supports shared interrupts | Set the <i>MS_SHARE_IRQ0_BIT</i> bit in the <i>MLIDCFG_SharingFlags</i> field of the CHSM'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>MS_SHARE_IRQ0_BIT</i> bit in the <i>MLIDCFG_SharingFlags</i> field of the CHSM's configuration table. | Not used. |

DriverISR

Called by the CMSM when a hardware interrupt is detected.

Syntax

```
void DriverISR (
    DRIVER_DATA *driverData );
```

Input Parameters

driverData

Pointer to the CHSM's driver adapter data space.

Output Parameters

None.

Return Values

None.

Remarks

DriverISR only needs to service the adapter and return.

We recommend that interrupts remain unaltered during **DriverISR**. If the **DriverISR** routine enables or disables interrupts, it must restore them to their state on entry before returning. Enabling or disabling interrupts will preclude the MLID from working on some platforms.

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

- Packet Reception Event
- Transmission Complete Event

The interrupt service routine should continue checking for reception and transmission events until there are no more to be serviced.

Packet Reception Event

The reception portion of the board service routine services the packet using one of the reception methods described in the previous section of this chapter. If an error has occurred the CHSM should increment the appropriate error counters that it is responsible for and indicate the appropriate status before handing the packet to the CTSM. (See **DriverPromiscuousChange** later in the chapter.)

Note



The CHSM should maintain the diagnostic counters in the statistics table for every detectable error condition. This will aid in debugging the CHSM as well as maintaining it in the future.

Transmission Complete Event

Each time the CHSM detects a successfully completed transmission event, it should do the following:

1. Return the TCB using **<CTSM>SendComplete** if the TCB was not returned during **DriverSend**.
2. Increment the number of available transmit resources using *CMSMTxFreeCount*.

If the CHSM encounters a transmission error, it should perform the following actions:

- *Attempt to identify the error.* The CHSM should try to pinpoint the specific cause of the error (excess collisions, cable disconnect, FIFO underrun).
- *Increment diagnostic counters.* The CHSM should maintain the diagnostic counters in the statistics table for every detectable error condition. The CHSM should also increment the generic statistic *TotalTxMiscCount* if a fatal transmission error occurred that is not counted in any other generic counter. The fatal transmission error could also be counted using a media specific counter.
- *Attempt to send the packet again.* In the event the CHSM has reached the maximum retry limit for sending a packet, it should do the following:

1. Discard the packet.
2. Return the TCB using **<CTSM>SendComplete** if the TCB was not returned during **DriverSend**.
3. Increment the number of available transmit resources using *CMSMTxFreeCount*.

DriverPoll

Services the adapter.

Syntax

```
void DriverPoll (
    const DRIVER_DATA *driverData,
    const MLID_CONFIG_TABLE *configTable );
```

Input Parameters


- driverData*
Pointer to the CHSM's driver adapter data space.
- configTable*
Pointer to the configuration table, of the first logical board which the CHSM registered. The tool kit refers to this as the default virtual board.

Return Values


None.

Remarks

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

Note  **DriverPoll** is normally not used by an interrupt driven CHSM. However, some CHSMs might require polling or could require polling in addition to the interrupt service routine.

To register the polling routine, place a pointer to the routine in the *DriverPollPtr* field of the driver parameter block. The CHSM can then enable polling during initialization by calling **CMSMEnablePolling**.

Note  We recommend that polled MLIDs have an interrupt backup facility to service the adapter if polling becomes too infrequent. On some platforms, frequent polling is not available and servicing the adapter with interrupts is required for

the adapter to function efficiently. Call **CMSMGetPollSupportLevel** to find out the level of support provided on the platform.

Packet Transmission

The CHSM routine that handles packet transmission is influenced by the adapter's data transfer mode. Transmission methods for different transfer modes are discussed here, followed by a description of the **DriverSend** routine.

Transmission Methods

There are two methods for transmitting packets. The method you choose depends upon the type of adapter you are writing your CHSM for.

Transmission Method—Option 1

This is the method for transmitting packets on programmed I/O, shared RAM, host DMA, and bus master adapters.

1. The CHSM sets *CMSMTxFreeCount* to the maximum number of packets that the adapter can buffer (performed in *DriverInit*). If the adapter needs physical addresses, set the *MM_FRAGS_PHYS_BIT* bit in the *MLIDCFG_ModeFlags* field at **DriverInit** time.
2. The CTSM receives an ECB, processes the information, and constructs a TCB. The TCB structure consists of the assembled packet header and data fragment information. If the Ethernet CTSM is used, *paddedLen* is set to the padded length of the packet (see **DriverSend** for more information). (This is the value that the adapter sends on the wire, regardless of the value in the *TCBDataLen* field. In fact, the value in *paddedLen* 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. The CTSM decrements *CMSMTxFreeCount* and calls **DriverSend** with a pointer to a filled in TCB structure.
4. The CHSM calls **<CTSM>SendComplete** after the packet has been buffered onto the adapter or after the transmission has been completed.
5. The CHSM increments *CMSMTxFreeCount* after the adapter completes the transmission (typically performed in **DriverISR**).

Transmission Method—Option 2

This method is recommended if the adapter is ECB-aware and has sufficient adapter processor speed. This transmission method dramatically reduces the load on the system by reducing the host's process time.

1. The CHSM sets the *DriverSendWantsECBs* field in the DRIVER_PARM structure to 1 and sets *CMSMTxFreeCount* to the number of packets that the adapter can process at one time (performed in *DriverInit*). If the adapter needs physical addresses, set the *MM_FRAGS_PHYS_BIT* bit in the *MLIDCFG_ModeFlags* field.
2. The CTSM decrements *CMSMTxFreeCount* and calls **DriverSend** with a pointer to the configuration table and a pointer to the ECB.
3. The CHSM adds the media header and sends the packet.
4. The CHSM calls either <CTSM>**SendComplete** after the packet has been buffered onto the adapter or after the transmission has been completed.
5. The CHSM increments *CMSMTxFreeCount* after the adapter completes the transmission (typically performed in **DriverISR**).

Priority Transmission Support

The following algorithm is used for priority transmission support.

1. During **DriverInit**, the CHSM sets the following parameters:
 - u The *DriverPriorityQueuePtr* variable is set with a pointer to the function, **DriverPriorityQueueSupport**.
 - u The MF_PRIORITY_BIT in the *MLIDCFG_Flags* field of the MLID Configuration Table is set.
 - u The *MLIDCFG_PrioritySup* field in the MLID Configuration Table is set to indicate the number of levels available.

The CHSM can set or reset the MF_PRIORITYSUP_BIT as the CHSM changes the Priority Queue Support state from enabled to disabled. The MF_PRIORITYSUP_BIT is checked on a per queued packet basis.

2. The protocol stack sets the *ECB_StackID* field to a value greater than or equal to 0x0FFF0. The following values are valid for the *ECB_StackID* field:

| | | |
|---------------------|---------|----------------------------------|
| RAW_SEND_PRIORITY_0 | 0xFFFF | No Priority. |
| RAW_SEND_PRIORITY_1 | 0xFFFE | Scale 1-7: 1 = Lowest Priority. |
| RAW_SEND_PRIORITY_2 | 0xFFFD | |
| RAW_SEND_PRIORITY_3 | 0xFFFC | |
| RAW_SEND_PRIORITY_4 | 0xFFFB | |
| RAW_SEND_PRIORITY_5 | 0xFFFA | |
| RAW_SEND_PRIORITY_6 | 0xFFFF9 | |
| RAW_SEND_PRIORITY_7 | 0xFFFF8 | Scale 1-7: 7 = Highest Priority. |
| SEND_PRIORITY_0 | 0xFFF7 | Scale 1-7: 0 = No Priority. |
| SEND_PRIORITY_1 | 0xFFF6 | Scale 1-7: 1 = Lowest Priority. |
| SEND_PRIORITY_2 | 0xFFF5 | |
| SEND_PRIORITY_3 | 0xFFF4 | |
| SEND_PRIORITY_4 | 0xFFF3 | |
| SEND_PRIORITY_5 | 0xFFF2 | |
| SEND_PRIORITY_6 | 0xFFF1 | |
| SEND_PRIORITY_7 | 0xFFF0 | Scale 1-7: 7 = Highest Priority. |

3. The CTSM normally gives the packet to the CHSM directly, as a TCB using the **DriverSend** function. However, if *CMSMTxFreeCount* is zero and the transmit ECB is a priority transmit ECB, the CTSM calls **DriverPriorityQueueSupport**, which gives the CHSM a chance to take

the transmit ECB. The **DriverPriorityQueueSupport** function, provided by the CHSM, queues the ECB in the CHSM for transmission as soon as possible, or transmits the packet through a priority channel by first building a TCB using **<CTSM>BuildTransmitControlBlock**, or returns a failure code and does not accept the ECB.

4. The CHSM calls **<CTSM>BuildTransmitControlBlock** to build a TCB whenever a priority transmit resource becomes available and a transmit ECB is in the CHSM's priority queue. The CHSM tracks the number of available priority TCBs. **CMSMPriorityTxFreeCount** is set during **DriverInit** and must provide the maximum number of priority TCBs, which must not change without unloading and reloading the CHSM. See Chapter 4, "CMSM/CTSM Structures & Variables" for more details on **CMSMPriorityTxFreeCount**. Non-priority packets use the original number of TCBs from **CMSMTxFreeCount**, which is reserved exclusively for their use. The CHSM must not call **<CTSM>BuildTransmitControlBlock** if no priority TCBs are available.
5. After the CHSM has transmitted the TCB returned by **<CTSM>BuildTransmitControlBlock**, the CHSM calls **<CTSM>SendComplete** or **<CTSM>FastSendComplete**, which increments the statistic counters, calls **TxMonitor**, places the TCB back on the TCBs Free list, and returns the ECB to its original owner.

Adapters that Need Physical Addresses

Some adapters need physical addresses because they are bus master or DMA adapters. If your adapter needs physical addresses, you can set the *MM_FRAGS_PHYS_BIT* bit in the *MLIDCFG_ModeFlags* field of the configuration table.

Specifically, set this bit if the CHSM expects the following things from the CTSM:

1. For TCBs, fragment pointers all contain physical addresses pointed to locked, contiguous buffers.
2. For ECB aware adapters and for send ECBs, pointers to the ECB can be converted to a physical address, so physical and logical addresses can be returned to the ECB.
3. **<CTSM>ProcessGetRCB** returns an RCB with locked, contiguous, physical addresses in the fragment pointer.

Note



For transmissions, if the *MM_FRAGS_PHYS_BIT* bit in the *MLIDCFG_ModeFlags* field of the configuration table is set and you need to access the data from the processor—for example, to double copy a small packet—you can use either **CMSMECBPhysToLogFrag**s or **CMSMTCBPhysToLogFrag**s. **CMSMECBPhysToLogFrag**s is used if you are using ECBs and **CMSMTCBPhysToLogFrag**s is used if you are using TCBs. These APIs are described in Chapter 7, "CMSM Functions".

DriverPriorityQueueSupport

Called by <CTSM> before it queues a priority packet.

Syntax

```
ODISTAT DriverPriorityQueueSupport(  
    DRIVER_DATA          *driverData,  
    MLID_CONFIG_TABLE    *configTable,  
    ECB                  *ecb );
```

Parameters

ecb

Pointer to a Transmit ECB.

driverData

Pointer to the CHSM's driver adapter data space.

configTable

Pointer to the configuration table.

Output Parameters

None.


Return Values


| | |
|--------------------------|--|
| ODISTAT_SUCCESSFUL | The ECB was processed/queued by the CHSM. |
| ODISTAT_OUT_OF_RESOURCES | The ECB was not processed/queued by the CHSM. The CTSM will now queue the ECB and initiate transmission at a later time. |

Remarks

This function must either transmit the packet immediately or queue the ECB. The CHSM 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.

The CHSM must set *DriverPriorityQueuePtr* in the Driver Parameter Block to point to this function. The CHSM can set or reset MF_PRIORITYSUP_BIT in the *MLIDCFG_Flags* field of the configuration table as the CHSM changes from supporting to not supporting priority packet states. MF_PRIORITYSUP_BIT is checked on a per packet basis.

Note  The *ECB_DriverWorkSpace* field of the ECB cannot be modified by the CHSM.

Note  The addresses in the ECB fragment structure are logical addresses. If the CHSM needs physical addresses, they will be returned when **<CTSM>BuildTransmitControlBlock** is called. If the CHSM is ECB-aware and needs physical addresses in the ECB's fragment structure, it must call **<CTSM>BuildTransmitControlBlock**.

DriverSend

Transfers a frame onto the LAN medium.

Syntax

```
void DriverSend (
    const DRIVER_DATA *driverData,
    const MLID_CONFIG_TABLE *configTable,
    const TCB *tcb,
    UINT32 paddedLen,
    void *ecbPhysicalPtr );
```

Input Parameters

driverData

Pointer to the CHSM's driver adapter data space.

configTable

Pointer to the configuration table.

tcb

Pointer to a TCB or, if the adapter is ECB aware, an ECB.

paddedLen

Padded length of the packet. This parameter is used for Ethernet only. If the adapter is ECB aware, the value is zero. This parameter contains the length of the entire frame as it appears on the LAN medium. Ethernet CHSMs gives this value to the LAN adapter, which defines the number of bytes to transmit. The *TCBDataLen* field only describes the amount of data being passed in the TCB. In the case of Ethernet, the frame might have been padded or evenized. For example, if the CHSM uses DMA to transfer the TCB data to the LAN adapter's memory, the *TCBDataLen* field tells the LAN adapter how many bytes to transfer.

ecbPhysicalPtr

Pointer to the physical ECB (ECB aware adapters only).

Output Parameters

None.

Return Values

None.

Remarks

The CTSM calls **DriverSend** to transmit a frame onto the medium.

DriverSend is provided a pointer to a TCB. Refer to *Chapter 3: CHSM Data Structures and Variables* for information on TCBs.

The CHSM can assume that the TCB will be valid for its topology; it should not do consistency checking on the TCB fields. The CHSM can also assume that it has the resources necessary to handle the transmission operation; it does not need to check to see if it has a transmission hardware resource available, because the CTSM will do flow control for the CHSM. The CTSM determines how many outstanding transmissions the CHSM can handle by using the value set in the *CMSMTxFreeCount* variable during *DriverInit*.

The **DriverSend** routine can request ECBs instead of TCBs by initializing the driver parameter block variable *DriverSendWantsECBs* to 1 (see Chapter 3, "CHSM Data Structures and Variables"). If **DriverSend** uses ECBs for packet transmission, it is responsible for building the proper media header (refer to Appendix B, "Event Control Blocks (ECBs)" for additional information on ECB aware adapters). If the CHSM uses ECBs instead of TCBs, it must not modify the transmit ECB's *ECB_PreviousLink* field.

Interrupts are unchanged. We recommend that interrupts remain unchanged during **DriverSend**.

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.
IF *lying* send
 Call <CTSM>**SendComplete**(config Table, tcbp, transmitStatus)
RETURN

Control Procedures

The ODI specification requires MLIDs to implement the I/O control functions (IOCTLs) listed in the table below. The CMSM and CTSM perform several of the required IOCTL functions without assistance from the CHSM, as indicated in the table. The CHSM is responsible for implementing the control functions described in this section.

DriverReset and **DriverShutdown** are mandatory and must be present for the MLID to function properly. The CHSM should also provide the **DriverMulticastChange** and **DriverPromiscuousChange** functions if the hardware supports these functions.

The **DriverStatisticsChange** and **DriverRxLookAheadChange** functions are optional. These functions allow MLIDs for intelligent adapters to update the statistics table or the lookahead size only as needed. Refer to the driver parameter block field descriptions in Chapter 3, "CHSM Data Structures and Variables" for additional information on these control functions.

MLIDs that support the hub management interface must implement the **DriverManagement** function to handle management requests and commands, as described in the *ODI Supplement: Hub Management Interface*.

MLIDs whose adapters are able to enable and disable generating interrupts at the adapter must implement **DriverDisableInterrupt** and **DriverEnableInterrupt**. MLIDs whose adapter is not able to do this must not implement these calls and will not be able to execute on some platforms.

Table 5-2
Code Path of Control Functions

| Control Function | Code Path |
|------------------------------|--|
| 0 Get configuration table | CMSM |
| 1 Get statistics table | CMSM -> DriverStatisticsChange |
| 2 Add Multicast Address | CMSM -> CTSM -> DriverMulticastChange or, if promiscuous mode is enabled: CMSM -> CTSM -> (DriverMulticastChange and DriverPromiscuousChange) |

Table 5-2

Code Path of Control Functions *continued*

| Control Function | Code Path |
|------------------------------------|--|
| 3 Delete Multicast Address | CMSM -> CTSM -> DriverMulticastChange or, if promiscuous mode is enabled: CMSM -> CTSM -> (DriverMulticastChange and DriverPromiscuousChange) |
| 4 Reserved | CMSM |
| 5 Shut down driver | CMSM -> CTSM -> DriverShutdown |
| 6 Reset Driver | CMSM -> CTSM -> DriverReset |
| 7 Reserved | CMSM |
| 8 Reserved | CMSM |
| 9 Set receive lookahead size | CMSM -> CTSM -> DriverRxLookAheadChange |
| 10 Enable/Disable Promiscuous Mode | CMSM -> CTSM -> DriverPromiscuousChange |
| 11 RegisterMonitor | CMSM -> CTSM |
| 12 Reserved | CMSM |
| 13 Reserved | CMSM |
| 14 Driver Management | CMSM -> DriverManagement |
| 15 Get Multicast Info | CMSM -> CTSM |

DriverReset

Resets and initializes the specified part of the MLID.

Syntax

```
ODISTAT  *DriverReset (
    const  DRIVER_DATA    *driverData,
    const  MLID_CONFIG_TABLE *configTable
    OPERATION_SCOPE operationScope );
```

Input Parameters

- driverData*
Pointer to the CHSM’s driver adapter data space.
- configTable*
Pointer to the configuration table.
- operationScope*
Indicates the scope of the operation to be performed.

- OP_SCOPE_ADAPTER**
Resets and initializes the physical adapter.
- OP_SCOPE_LOGICAL_BOARD**
Performs a logical board reset.

Output Parameters

None.

Return Values

| | |
|--------------------------|--|
| ODISTAT_SUCCESSFUL | This function completed successfully. |
| ODISTAT_RESPONSE_DELAYED | The function cannot complete immediately; this is due to the asynchronous nature of this function. |
| ODISTAT_FAIL | The function was unable to complete successfully. |

Remarks

OP_SCOPE_ADAPTER

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

If resetting the adapter clears the hardware’s multicast and/or promiscuous mode capability, **DriverReset** must restore this capability to the way it was before it was called using **<CTSM>UpdateMulticast**.

From within the CHSM, **DriverReset** can be called by **DriverAES** or **DriverInit**. It can also be called by **DriverISR** if the adapter has problems.

If the CMSM calls **DriverReset** and the CHSM returns successfully, the CMSM resets the *CMSMTxFreeCount* variable to the initial value set by the MLID during initialization. If the CMSM calls **DriverReset** and the adapter cannot be reset, the CMSM automatically calls **DriverShutdown** with *shutdownType* equal to SHUTDOWN_PERMANENT.

Hardware features, such as advanced power management and docking stations, require this routine so they can reinitialize the adapter after power has been removed and is then restored.

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

OP_SCOPE_LOGICAL_BOARD

The meaning of this operation is adapter/media/driver-dependant. This operation is a NO-OP for most LAN drivers.

Important



If **ODISTAT_RESPONSE_DELAYED** is returned by **DriverReset**, due to delays in completing events performed by this function, the CHSM must call **CMSMControlComplete** when the event is complete. If **DriverReset** must be called from within the CHSM, use **CMSMResetMLID**.

Note



DriverReset must not assume that other initialization processes have been performed on the hardware. **DriverReset** must be capable of initializing the hardware completely.

Pseudocode

```
Increment the reset statistics counter (MAdapterResetCount)
IF OP_SCOPE_ADAPTER
    Reset the hardware (includes performing any hardware testing)
    CALL <CTSM>UpdateMulticast
ELSE perform logical board reset
IF successful
    Return ODISTAT_SUCCESSFUL
ELSE IF the update will complete asynchronously
    Return ODISTAT_RESPONSE_DELAYED
    /* Call CMSMControlComplete when done. */
ELSE
    Return ODISTAT_FAIL
END IF
RETURN
```

DriverShutdown

Releases the HSM resources associated with the entity being shutdown. If an adapter is being shutdown, it puts the adapter in an inactive state.

Syntax

```
ODISTAT DriverShutdown (
    const DRIVER_DATA *driverData,
    const MLID_CONFIG_TABLE *configTable,
    UINT32 shutdownType,
    OPERATION_SCOPE operationScope);
```

Input Parameters

driverData

Pointer to the CHSM's driver adapter data space.

configTable

Pointer to the configuration table.

shutdownType

SHUTDOWN_PERMANENT

Permanent shutdown.

SHUTDOWN_PARTIAL

Partial shutdown.

operationScope

OP_SCOPE_ADAPTER

All the logical boards associated with the CHSM's driver adapter data space are to be shut down.

OP_SCOPE_LOGICAL_BOARD

Only the logical board specified by *configTable* is to shut down.

Output Parameters

None.

Return Values

| | |
|--------------------------|--|
| ODISTAT_SUCCESSFUL | The operation was successful. |
| ODISTAT_RESPONSE_DELAYED | The function cannot complete immediately; this is due to the asynchronous nature of this function. |
| ODISTAT_FAIL | The operation failed. |

Remarks

OP_SCOPE_ADAPTER

SHUTDOWN_PARTIAL

Passing SHUTDOWN_PARTIAL as the *shutdownType* parameter indicates a partial shutdown.

In a partial shutdown, the C MSM does the following:

- 1. Sets **CMSMStatusFlag** to SHUTDOWN.
- 2. Sets the MS_SHUTDOWN_BIT of the *MLIDCFG_SharingFlags* field in the configuration table.
- 3. Waits for the transmissions in progress to complete
- 4. Returns the transmit ECBs.

DriverShutdown must place the hardware into a safe, inactive state.

SHUTDOWN_PERMANENT

Passing SHUTDOWN_PERMANENT as the *shutdownType* parameter indicates a permanent shutdown.

In a permanent shutdown, the C MSM does the following:

1. Sets CMSMStatusFlag to SHUTDOWN.
2. Sets the MS_SHUTDOWN_BIT of the MLIDCFG_SharingFlags field in the configuration table.
3. Empties the send queue.
4. Returns all resources not allocated directly by the C HSM.

If the CHSM allocated memory using **CMSMAlloc** or **CMSMInitAlloc**, the CHSM must return the memory using **CMSMFree** after disabling the hardware.

If the C HSM can power down or power off the adapter, it may do so at this time. The C MSM disables the adapter's interrupt service routine immediately after this routine returns.

The CMSM automatically calls **DriverShutdown** with SHUTDOWN_PERMANENT and OP_SCOPE_ADAPTER when the **DriverReset** routine fails to reset the hardware. **DriverShutdown** is also called when the MLID is about to be unloaded or when **CMSMShutdownMLID** is called.

OP_SCOPE_LOGICAL_BOARD

SHUTDOWN_PARTIAL

In most cases, C HSMs do not need to do anything. However, some C HSMs may have house keeping to be done.

SHUTDOWN_PERMANENT

DriverShutdown must release all C HSM-allocated resources associated with the specified logical board.

Important



If **ODISTAT_RESPONSE_DELAYED** is returned by **DriverShutdown**, due to delays in completing events performed by this function, the CHSM must call **CMSMControlComplete** when the event is complete. If **DriverReset** must be called from within the CHSM, use **CMSMShutdownMLID**.

Pseudocode

```

If OP_SCOPE_ADAPTER
    Disable Hardware
    If SHUTDOWN_PERMANENT
        Return memory associated with the adapter that
        was allocated by the CHSM.

Else /* OP_SCOPE_LOGICAL_BOARD */
    If SHUTDOWN_PERMANENT
        Return memory associated with the logical
        board that was allocated by the CHSM.

If failure
    Return ODISTAT_FAIL
Return ODISTAT_SUCCESSFUL

```

Note



If ODISTAT_RESPONSE_DELAYED is returned as the completion code, **DriverShutdown** will execute differently and will require that **CMSMControlComplete** be called when the shutdown is complete.

DriverMulticastChange

Updates the adapter to reflect the changes in the CTSM's functional address table.

Syntax

```
ODISTAT DriverMulticastChange (
    const DRIVER_DATA    *driverData,
    const MLID_CONFIG_TABLE *configTable,
    const GROUP_ADDR_LIST_NODE*groupAddrListNode,
    UINT32 numEntries,
    UINT32 funAddrBits );
```

Input Parameters

driverData

Pointer to the CHSM's driver adapter data space.

configTable

Pointer to the configuration table.

groupAddrListNode

Pointer to the Group Address List Node table (Ethernet, FDDI, and sometimes Token-Ring).

numEntries

Number of valid entries in the Group Address List Node table (Ethernet, FDDI, and sometimes Token-Ring).

funAddrBits

32-bit functional address (Token-Ring).

Output Parameters

None.

Return Values

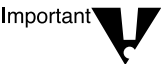
| | |
|--------------------------|--|
| ODISTAT_SUCCESSFUL | The multicast/functional address table was updated successfully. |
| ODISTAT_RESPONSE_DELAYED | The multicast/functional address table will be updated asynchronously. |
| ODISTAT_FAIL | The multicast/functional address table was not updated successfully. |

Remarks

CHSMs must support multicast addressing if the hardware allows it. The following flags and variables must be initialized properly for the adapter’s multicast mode.

- *MM_MULTICAST_BIT* bit of the *MLIDCFG_ModeFlags* field must be set to indicate whether multicast addressing is supported.
- *MF_SOFT_FILT_GRP_BIT* and *MF_GRP_ADDR_SUP_BIT* bits of the *MLIDCFG_Flags* field must be set appropriately to reflect the multicast mechanism or format used by the adapter/MLID.
- The driver parameter block variable, *DriverMaxMulticast*, must be set to reflect the maximum number of multicast addresses the adapter can handle.

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



If *ODISTAT_RESPONSE_DELAYED* is returned by **DriverMulticastChange**, due to delays in completing events performed by this function, the CHSM must call **CMSMControlComplete** when the event is complete.

Adapter Multicast Filtering

Hashing is the most common method used by adapters to filter incoming packets. When the adapter uses the hashing method, **DriverMulticastChange** must recalculate and update the adapter’s hash table. Hashing does not guarantee 100 percent multicast filtering; therefore, the CTSM looks up

incoming packets in its multicast address table to ensure that the packet's destination address is enabled.

The CTSM verifies that all the addresses it places in its table are valid multicast addresses, so the CHSM does not need to validate them.

In either case, the CHSM routine must read the CTSM's multicast address table.

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

Ethernet and FDDI

On entry to this routine, *numEntries* contains the number of valid entries in the multicast table. All valid entries will be contiguous, so the CHSM does not necessarily need to check the *MulticastInUse* flag. If *numEntries* is 0, multicast reception is disabled.

Token-Ring

The CTSM passes the 32-bit functional address in *funAddrBits*. In this case *numEntries* and *multicastTable* are normally not used. If the Token-Ring adapter is capable of supporting both functional and group addressing and the *MF_SOFT_FILT_GRP_BIT* and *MF_GRP_ADDR_SUP_BIT* bits in *MLIDCFG_Flags* are set properly, both group addressing using *multicastTable* and functional addressing using *funAddrBits* may be in use simultaneously.

Pseudocode

The default method (if the *MF_SOFT_FILT_GRP_BIT* and *MF_GRP_ADDR_SUP_BIT* bits of the *MLIDCFG_Flags* field are 0) for handling multicast operations is as follows:

Clear the hardware registers that filter incoming packets for multicast addresses.

Get the current multicast addresses from the CTSM's multicast table.

Reload the hardware register with new multicast address filtering values.

Return.

DriverPromiscuousChange

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

Syntax

```
ODISTAT  DriverPromiscuousChange (
    const  DRIVER_DATA    *driverData,
    const  MLID_CONFIG_TABLE *configTable,
    UINT32  changeTo  );
```

Input Parameters

- driverData*
Pointer to the CHSM's driver adapter data space.
- configTable*
Pointer to the configuration table.
- changeTo*
0 to disable promiscuous mode.
All bits set to receive all packets.
If *changeTo* is nonzero:
 - Bit 0 is set if MAC frames are to be received
 - Bit 1 is set if non-MAC frames are to be received
 - Bit2 is set if Station Management Frames (SMT) are to be received.
 - Bit 3 is set if Remote Multicast Frames are to be received (see Remarks section below).Multiple bits can be set.

Output Parameters

None.

Return Values

| | |
|--------------------------|--|
| ODISTAT_SUCCESSFUL | Promiscuous mode was changed successfully. |
| ODISTAT_RESPONSE_DELAYED | Promiscuous mode will be changed asynchronously. |
| ODISTAT_FAIL | Promiscuous mode was not changed successfully. |

Remarks

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

A monitor function examines packets sent from or received by an adapter. If the MLID supports promiscuous mode, the monitoring function can request that the adapter enter promiscuous mode. When promiscuous mode is enabled, the MLID must allow all packets (including bad packets if possible) to be passed up to the monitor function.

Setting the RemoteMulticastFrames bit causes the CHSM to activate all multicast frame reception. For example, if an adapter utilizes a hash table for filtering active multicast frames, then the adapter sets the hash table to accept all multicast frames. Filtering active multicast entries is disabled when this bit is set. CHSMs that can filter must disable filtering also when this bit is set.

Multiple bits may be set so that each bit adds to the type of frames that are to be received.

<CTSM>GetRCB and <CTSM>ProcessGetRCB require the MLID to indicate the status of the packet. The returned values, where appropriate, for Token-Ring, Ethernet, and FDDI are as follows:

0 = good packets

!0 = bad packets

BITS are set as follows for bad packets:

| | |
|---------------------|---|
| PAE_CRC_BIT | CRC Error (Bit 0) |
| PAE_CRC_ALIGN_BIT | CRC/Frame Alignment Error (Bit 1) |
| PAE_RUNT_PACKET_BIT | Runt Packet |
| PAE_TOO_BIG_BIT | Packet Too Large for Media |
| PAE_NOT_ENABLED_BIT | Unsupported Frame |
| PAE_MALFORMED_BIT | Malformed Packet |
| PA_NO_COMPRESS_BIT | Do not compress received packet |
| PA_NONCAN_ADDR_BIT | Set if address in Immediate Address field is noncanonical |

The CHSM must set all bits for ECB-aware adapters.

The CHSM must only set bits 0 and 1 (CRC Error and CRC/Frame Alignment Error) for RCB-aware adapters.

Note



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

If the CHSM does not support promiscuous mode, the *MM_PROMISCUOUS_BIT* bit of the *MLIDCFG_ModeFlags* field in the configuration table must be cleared, and the *DriverPromiscuousChangePtr* field in the driver parameter block must be NULL.

Important



If *ODISTAT_RESPONSE_DELAYED* is returned by **DriverPromiscuousChange**, due to delays in completing events performed by this function, the CHSM must call **CMSMControlComplete** when the event is complete.

Pseudocode

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

DriverStatisticsChange (optional)

Allows the CMSM to notify MLIDs whenever an application requests IOCTL 1 (get MLID statistics).

Syntax

```
ODISTAT DriverStatisticsChange (  
    const DRIVER_DATA *driverData,  
    const MLID_CONFIG_TABLE *configTable );
```

Input Parameters

- driverData*
Pointer to the CHSM’s driver adapter data space.
- configTable*
Pointer to the configuration table.

Output Parameters

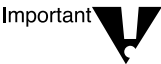
None.

Return Values

| | |
|--------------------------|--|
| ODISTAT_SUCCESSFUL | The statistics table was successfully updated. |
| ODISTAT_RESPONSE_DELAYED | The function cannot complete immediately; this is due to the asynchronous nature of this function. |
| ODISTAT_FAIL | The statistics table was not successfully updated. |

Remarks

Because some intelligent adapters maintain statistical information on boards, this optional routine enables MLIDs to update the statistics table in the driver data space before the CMSM passes it up to the requesting application.



If *ODISTAT_RESPONSE_DELAYED* is returned by **DriverStatisticsChange**, due to delays in completing events performed by this function, the CHSM must call **CMSMControlComplete** when the event is complete.

See Also

DriverStatisticsChangePtr field of *DriverParameterBlock*.

DriverRxLookAheadChange (optional)

Allows the CMSM to notify CHSMs after an application invokes IOCTL 9 to set the lookahead size.

Syntax

```
ODISTAT DriverRXLookAheadChange (  
    const DRIVER_DATA    *driverData,  
    const MLID_CONFIG_TABLE *configTable );
```

Input Parameters

- driverData*
Pointer to the CHSM’s driver adapter data space.
- configTable*
Pointer to the configuration table.

Output Parameters

None.

Return Values

| | |
|--------------------------|--|
| ODISTAT_SUCCESSFUL | The requested operation was completed successfully. |
| ODISTAT_RESPONSE_DELAYED | The function cannot complete immediately; this is due to the asynchronous nature of this function. |

Remarks

This routine changes the *CMSMMaxFrameHeaderSize* variable and the *MLIDCFG_LookAheadSize* field in the configuration table. MLIDs use this routine to inform adapters when the size changes rather than forcing the

adapter to check the value. Non-intelligent adapters usually use *CMSMMaxFrameHeaderSize* and *MLIDCFG_LookAheadSize* directly.

Important



If *ODISTAT_RESPONSE_DELAYED* is returned by **DriverRxLookAheadChange**, due to delays in completing events performed by this function, the CHSM must call **CMSMControlComplete** when the event is complete.

See Also

DriverRxLookAheadChangePtr field of *DriverParameterBlock*,
MLIDCFG_LookAheadSize in the configuration table, the
CMSMMaxFrameHeaderSize variable, and the <CTSM>**GetRCB** function.

DriverManagement (optional)

Processes management requests if an MLID accepts management commands from outside NLMs (such as hub management interface, Brouter, or CSL).

Syntax

```
ODISTAT DriverManagement (  
    const DRIVER_DATA    *driverData,  
    const MLID_CONFIG_TABLE *configTable,  
    ECB *ecbp );
```

Input Parameters

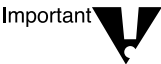
- driverData*
Pointer to the CHSM’s driver adapter data space.
- configTable*
Pointer to the configuration table.
- ecbp*
Pointer to the management ECB containing the request (see *Appendix B: Event Control Blocks*). The first byte of the *ECB_ProtocolID* field is greater than 0x40 and less that 0x7F.

Output Parameters

None.

Return Values

| | |
|--------------------------|--|
| ODISTAT_SUCCESSFUL | The requested operation was completed successfully. |
| ODISTAT_RESPONSE_DELAYED | The function cannot complete immediately; this is due to the asynchronous nature of this function. |
| ODISTAT_NO_SUCH_HANDLER | The ECB requested an operation that is not available. |



If *ODISTAT_RESPONSE_DELAYED* is returned by **DriverManagement**, due to delays in completing events performed by this function, the CHSM must call **CMSMControlComplete** when the event is complete.

See Also

Refer to *ODI Supplement: The Hub Management Interface* for a hub management implementation of this function. See also the *DriverManagementPtr* field of the driver parameter block in Chapter 3, "CHSM Data Structures and Variables".

For more information on Brouter, see *ODI Supplement: Brouter Support*.

DriverEnableInterrupt

Called by the CMSM through the driver parameter block to enable the adapter's interrupt(s) at the adapter.

Syntax

```
void DriverEnableInterrupt (  
    const DRIVER_DATA *driverData );
```

Input Parameters

driverData

Pointer to the CHSM's driver adapter data space.

Output Parameters

None.

Return Values

None.

Remarks

This function re-enables the adapter's interrupt; it should undo whatever **DriverDisableInterrupt** does.

Important



It is critical that this function performs the necessary operations to enable the adapter's interrupt(s) as quickly as possible and then returns.

DriverDisableInterrupt

Called by the CMSM through the driver parameter block to disable the adapter's interrupt(s) at the adapter.

Syntax

```
BOOLEAN DriverDisableInterrupt (  
    const DRIVER_DATA *driverData,  
    BOOLEAN flag );
```

Input Parameters

driverData

Pointer to the CHSM's driver adapter data space.

flag

TRUE if the function is to return a value.

FALSE if the function is not to return a value.

Output Parameters

None.

Return Values

| | |
|-------|--|
| TRUE | The adapter generated the interrupt, for the primary interrupt. |
| FALSE | The adapter did not generate the interrupt, for the primary interrupt. |

Remarks

This function must be present and return its specified return value. If the CHSM can disable the adapter's ability to generate interrupts, it must do so with this function.

If **DriverDisableInterrupt** is called from the context of an interrupt, the CHSM must return to the CMSM whether it generated the interrupt for the primary interrupt or not.

If **DriverDisableInterrupt** is called with the flag parameter set to FALSE, the return value must be FALSE.

Important



It is critical that this function performs the necessary operations to disable the adapter's interrupt(s) as quickly as possible and then returns.

See Also

DriveDisableInterrupt2

DriverDisableInterrupt2

Called by the CMSM through the driver parameter block to disable the adapter's interrupt(s) at the adapter.

Syntax

```
BOOLEAN DriverDisableInterrupt2
    (const DRIVER_DATA *driverData,
     BOOLEAN flag);
```

Input Parameters

driverData

Pointer to the CHSM's driver adapter data space.

flag

TRUE if the function is to return a value.

FALSE if the function is not to return a value.

Output Parameters

None.

Return Values

TRUE

The adapter generated the interrupt for the secondary interrupt.

FALSE

The adapter did not generate the interrupt for the secondary interrupt.

Remarks

This function must be present and return its specified return value if the CHSM uses a secondary interrupt for which a second interrupt service routine (ISR) has been provided (see DriverISR2Ptr in DRIVER_PARM_BLOCK and MLIDCFG_Interrupt1 in the MLID_CONFIG_TABLE). If the CHSM can disable the adapter's ability to generate interrupts, it must do so with this function.

If DriverDisableInterrupt2 is called from the context of an interrupt, the CMSM calls DriverDisableInterrupt2 with flag set to TRUE and the CHSM must return to the CMSM whether it generated the interrupt for the secondary interrupt or not. If DriverDisableInterrupt2 is called with the flag parameter set to FALSE, the return value must be FALSE.

Note



It is critical that this function performs the necessary operations to disable the adapter's interrupt(s) as quickly as possible and then return.

See Also

DriverDisableInterrupt

Timeout Detection

The **DriverAES** routine may be used when the CHSM needs to be called after a specified interval or at periodic intervals. Typically, this routine allows an adapter to complete a time-consuming operation, such as a reset, or to determine if an adapter has failed to complete a packet transmission. This routine can also set up timed functions.

Execution time constraints determine whether the routine is called at privileged or nonprivileged time. At privileged time, the CHSM can only use operating system routines that are called at privileged time. If any routines are used that must be called during process time only, **DriverAES** should be set up to be invoked at nonprivileged time.

DriverAES

DriverAES is an event service routine.

Syntax

```
void DriverAES (
    const DRIVER_DATA *driverData,
    const MLID_CONFIG_TABLE *configTable );
```

Input Parameters

- driverData*
Pointer to the CHSM’s driver adapter data space.
- configTable*
Pointer to the configuration table.

Output Parameters

None.


Return Values

None.

Remarks

DriverAES is typically enabled during initialization) by calling **CMSMScheduleAES** (see Chapter 7, "CMSM Functions").

Once scheduled, the CMSM invokes this routine either once or continuously with a pointer to the configuration table and a pointer to the driver data space.

Note  You can use as many AES routines as you want, as long as the function names are different.

Pseudocode

The actual content of **DriverAES** is entirely up to the developer. The following pseudocode illustrates the use of **DriverAES** to identify a send timeout error.

```
<DriverAES > (const DRIVER_DATA *driverData, const  
MLID_CONFIG_TABLE *configTable );
```

```
IF Transmit is in Progress
```

```
    IF Elapsed Transmit Time > Maximum Time for Transmit
```

```
        Increment appropriate error counter
```

```
        Reset the adapter
```

```
        Reset CMSMTxFreeCount
```

```
    ENDIF
```

```
ENDIF
```

```
RETURN
```


chapter 6 *CTSM Functions*

Overview

This chapter describes the topology specific functions provided as tools for CHSM developers.

The C Language Topology Specific Module (CTSM) manages the operations that are unique to a specific media type. Multiple frame support is implemented in the CTSM so all frame types for a given media are supported.

In this specification, topology specific functions and variables are indicated with "<CTSM>". The developer must replace "<CTSM>" with the appropriate media type, depending on which module is used. Since the MLID must be compiled with case sensitivity on, the names must be used exactly as shown.

ETHERTSM.NLM → replace <CTSM> with CEtherTSM
TOKENSM.NLM → replace <CTSM> with CTokenTSM
FDDITSM.NLM → replace <CTSM> with CFDDITSM

<CTSM>BuildTransmitControlBlock

The C HSM calls this function when it is ready to send a priority packet that has been queued using **DriverPriorityQueueSupport**. The C HSM calls this function to convert an ECB to a TCB. ECB-aware C HSMs must call this function if they need physical addresses in the ECB fragment structure.

Syntax

```
#include <odi.h>
#include <<ctsm>.h>

ODISTAT <CTSM>BuildTransmitControlBlock(
    DRIVER_DATA    *driverData,
    ECB             *ecb,
    TCB             **tcb
    UINT32          *pktSize);
```

Input Parameters

driverData
CHSM adapter data space.

ecb
Pointer to a Transmit ECB.

Output Parameters

tcb
Pointer to a pointer to the TCB to send. If the CHSM is ECB-aware and has the *MLIDCFG_ModeFlags* MM_FRAGS_PHYS_BIT set, this field will have a pointer to an ECB with physical addresses in the fragment structure.

pktSize

Padded length of the packet. This parameter is used for Ethernet only. This parameter contains the length of the entire frame as it appears on the LAN medium. Ethernet CHSMs give this value to the LAN adapter, which defines the number of bytes to transmit. The *TCBDataLen* field only describes the amount of data being passed in the TCB. In the case of Ethernet, the frame might have been padded or evenized. For example, if the CHSM uses DMA to transfer the TCB data to the LAN adapter's memory, the *TCBDataLen* field tells the LAN adapter how many bytes to transfer.

Return Values

| | |
|------------------------------|---|
| ODISTAT_SUCCESSFUL | TCB pointer is valid. The CHSM should transmit the TCB. |
| ODISTAT_OUT_OF_RESOURCES | A TCB was not available. The CHSM must not call this routine with more outstanding TCBs than it set in the CMSMPriorityTxFreeCount variable. The ECB is returned to the CHSM. The CHSM must either call this function again after a TCB resource is available, or return the ECB via <CTSM>CancelPrioritySend . |
| ODISTAT_PACKET_UNDELIVERABLE | A TCB was available, but the ECB created a packet that was too large for the media. The ECB was returned to the LSL and a TCB was not allocated. |

Remarks

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

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

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

If the CHSM needs to cancel the TCB after this function has been called, then it should set the status field in the TCB to ODISTAT_CANCELED and call **<CTSM>SendComplete** or **<CTSM>FastSendComplete**.

<CTSM>CancelPrioritySend

The CHSM calls this function to cancel/return an ECB that has not been sent.

Syntax

```
#include <odi.h>
#include <<ctsm>.h>

void <CTSM>CancelPrioritySend
    (DRIVER_DATA    *driverData,
     ECB            *ecb, );
```

Input Parameters

driverData

Pointer to the CHSM's driver adapter data space.

ecb

Pointer to a transmit ECB.

Output Parameters

None.

Return Values

None.

Remarks

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

Note



Call this function only if canceling the ECB. If the CHSM has called **<CTSM>BuildTransmitControlBlock**, it should set the *ECB_Staus* field to **ODISTAT_CANCELED**. Then it should call **<CTSM>SendComplete** or **<CTSM>FastSendComplete**.

<CTSM>FastProcessGetRCB

Called by the CHSM to process an RCB for a received packet and to preallocate a new nonfragmented RCB for the next packet.

Syntax

```
#include <odi.h>
#include <csm.h>
#include <<ctsm>.h>

RCB *<CTSM>FastProcessGetRCB (
    DRIVER_DATA *driverData,
    RCB *rcb,
    UINT32 pktSize,
    UINT32 rcvStatus,
    UINT32 newRcbSize );
```

Input Parameters

driverData

Pointer to the HSM's driver adapter data space.

rcb

Pointer to the received packet's RCB.

pktSize

Size of the received packet including the MAC header.

rcvStatus

Status of received packet for the Receive Monitor (see **DriverPromiscuousChange** in Chapter 5, "CHSM Functions").

newRcbSize

Minimum packet size for the new RCB.

Output Parameters


None.


Return Values


Pointer to a new nonfragmented RCB.
 NULL No nonfragmented RCBs are available.


Remarks

<CTSM>**FastProcessGetRCB** is called at process or privileged time.
 <CTSM>**FastProcessGetRCB** is identical to <CTSM>**ProcessGetRCB**, except that before this routine returns, the RCB’s event service routine is called to complete the processing. <CTSM>**ProcessGetRCB** used in conjunction with **CMSMServiceEvents** accomplishes the same task.

Note  Do not call this function until the packet has been received into host memory and is ready to be handed to the CTSM.

Note  This functionality is not possible on any multiprocessor-capable platforms.

Note  For some busMaster implementations, you must set the first UINT32 parameter, starting at *RCBReserved[28]* (defined in CMSM.H), 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 for this function.

Note  If the MM_FRAGS_PHYS_BIT of the *MLIDCFG_Modeflags* field is set, the fragment offset of the RCB contains a physical pointer to the RCB data buffer.

See Also

<CTSM>**ProcessGetRCB**

<CTSM>FastRcvComplete

Called by the CHSM to direct a completed RCB to the protocol stack.

Syntax

```
#include <cmsm.h>
#include <<ctsm>.h>

void <CTSM>FastRcvComplete (
    const DRIVER_DATA *driverData,
    RCB *rcb );
```

Input Parameters

driverData

Pointer to the HSM'S driver adapter data space.

rcb

Pointer to the received packet's RCB.

Output Parameters

None.

Return Values

None.

Remarks

This routine increments the *MTotalRxPackets*, *MTotalRxOkByteCount*, and *MTotalGroupAddrRxCount* statistics counters as needed for ECB-aware adapters only. On adapters that have previously called <CTSM>GetRCB, the counters will have already been incremented.

<CTSM>FastRcvComplete is called at process or privileged time.

<CTSM>FastRcvComplete is identical to <CTSM>RcvComplete with the

exception that before this routine returns, the RCB's event service routine is called to complete the processing. Using **<CTSM>RcvComplete** in conjunction with **CMSMServiceEvents** accomplishes the same task.

Normally, **<CTSM>FastRcvComplete** is preferred over **<CTSM>RcvComplete**. **<CTSM>FastRcvComplete** gives the ECB directly to the protocol stack and saves an extra queueing. **<CTSM>RcvComplete** does not send a packet directly to the protocol stack but to the LSL, where it is queued. Adapters that have minimal amounts of memory should use **<CTSM>RcvComplete** to help keep the adapter from overflowing the buffer when too many packets are received.

Important



Do not call this function until the packet has been received into host memory and is ready to be handed to the CTSM.

Important



This functionality is not possible on any multiprocessor capable platforms.

See Also

<CTSM>RcvComplete

<CTSM>FastRcvCompleteStatus

Allows the CTSM to fill in the proper packet length fields of the RCB, record the error status, and direct the completed RCB to the protocol stack.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <<ctsm>.h>

void <CTSM>FastRcvCompleteStatus (
    DRIVER_DATA *driverData,
    RCB *rcb,
    UINT32 packetLength,
    UINT32 packetStatus );
```

Input Parameters

driverData

Pointer to the HSM'S driver adapter data space.

rcb

Pointer to the received packet's RCB.

packetLength

Size of the received packet including the MAC header.

packetStatus

Status of received packet for stack monitoring functions (see **DriverPromiscuousChange** in Chapter 5, "CHSM Functions").

Output Parameters

None.

Return Values

None.


Remarks

<CTSM>**FastRcvCompleteStatus** is identical to <CTSM>**RcvCompleteStatus** except that before this routine returns, the RCB's event service routine is called to complete the processing. <CTSM>**RcvCompleteStatus** used in conjunction with **CMSMServiceEvents** performs the same task as <CTSM>**FastRcvCompleteStatus**.


During the RCB's event service routine, the state of the system interrupt mask may change. The CHSM should preserve any needed state before calling <CTSM>**FastRcvCompleteStatus**. If having interrupts enabled is undesirable, the MLID should use <CTSM>**RcvCompleteStatus** and wait until the conclusion of the receive routine before servicing events.

This function is called at process or privileged time.


- Important



Do not call this function until the packet has been received into host memory and is ready to be handed to the CTSM.
- Important



This functionality is not possible on any multiprocessor capable platforms.
- Important



Intelligent adapters that are ECB aware should use <CTSM>**RcvComplete** to return the RCBs.

See Also

<CTSM>**RcvCompleteStatus**

<CTSM>FastSendComplete

Called by the CHSM's **DriverSend** or **DriverISR** routine to release a TCB after a packet has been transmitted.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <<ctsm>.h>

void <CTSM>FastSendComplete (
    DRIVER_DATA *driverData,
    TCB *tcb,
    UINT32 transmitStatus );
```

Input Parameters

driverData

Pointer to the HSM'S driver adapter data space.

tcb

Pointer to the TCB.

transmitStatus

Used to flag whether a packet was actually sent.

0 Successful.

nonzero Undeliverable.

Output Parameters

None.

Return Values

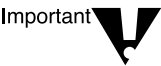
None.

Remarks

The *MTotalTxPacketCount*, *MTotalTxOkByteCount*, and *MTotalGroupAddrTxCount* statistics counters have been updated.

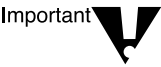
<CTSM>FastSendComplete is called at process or privileged time.
<CTSM>FastSendComplete is identical to **<CTSM>SendComplete** with the exception that before this routine returns, the TCB's event service routine is called to notify the upper layers that the transmission is complete. Using **<CTSM>SendComplete** and **CMSMServiceEvents** together accomplishes the same task.

Normally, **<CTSM>FastSendComplete** is preferred over **<CTSM>SendComplete**. **<CTSM>FastSendComplete** gives the ECB directly to the protocol stack and saves an extra queueing.
<CTSM>SendComplete does not send a packet directly to the protocol stack, but to the LSL where it is queued. Adapters that have minimal amounts of memory should use **<CTSM>SendComplete** to help keep the adapter from overflowing the buffer when lots of packets are being sent to it.



Important

Do not call this function until the packet has been handed to the card and the transmission has been initialized.



Important

This functionality is not possible on any multiprocessor capable platforms.

See Also

<CTSM>SendComplete

<CTSM>GetConfigInfo

Allows a C HSM to get the configuration information for the <CTSM>, including module and ODI specification versions.

Syntax

```
#include <odi.h>
#include <cmsm.h>

ODISTAT <CTSM>GetConfigInfo(
    void      *configInfo,
    UINT32    *nBytes);
```

Input Parameters

nBytes

Pointer to the requested number of bytes to be returned into the buffer.

Output Parameters

configInfo

A pointer to a buffer used to receive the returned configuration information. The caller needs to be sure that the buffer is at least *nBytes* bytes long.

nBytes

Pointer to the number of bytes returned in the configuration buffer.

Return Values

| | |
|-----------------------|--|
| ODISTAT_SUCCESSFUL | The configuration information of nBytes was successfully returned in the buffer. |
| ODISTAT_BAD_PARAMETER | The nBytes requested was larger than the actual configuration information available. The number of bytes of the configuration table actually returned is indicated by the output parameter nBytes. |

Remarks

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

<CTSM>GetHSMIFLevel

Gets the interface level between the CHSM and CTSM.

Syntax

```
#include <odi.h>
#include <<ctsm>.h>

UINT32 <CTSM>GetHSMIFLevel ( );
```

Input Parameters

None.

Output Parameters

None.

Return Values

Current CHSM interface level.

Remarks

The current interface level is 111.

<CTSM>GetRCB

Called by the CHSM to pass lookahead data to the CTSM, and to get a fragmented RCB for the remainder of the packet that has been received by the adapter.

Syntax

```
#include <odi.h>
#include <<ctsm>.h>

RCB * <CTSM>GetRCB (
    DRIVER_DATA *driverData,
    UINT8 *lookAheadData,
    UINT32 pktSize,
    UINT32 rcvStatus,
    UINT32 *startBytes,
    UINT32 *numBytes );
```

Input Parameters

driverData

Pointer to the C HSM'S driver adapter data space.

lookAheadData

Pointer to the received packet header (lookahead buffer).

pktSize

Size of the received packet including the MAC header.

rcvStatus

Status of received packet (see **DriverPromiscuousChange** in Chapter 5, "CHSM Functions").

Output Parameters

startBytes

Pointer to the number of bytes to skip over from the beginning of packet.

numBytes

Pointer to the number of bytes remaining to read.

Return Values

Pointer to a fragmented RCB if this call is successful.
NULL if no fragmented RCBs are available.

Remarks

This routine increments the *MTotalRxPackets*, *MTotalRxOkByteCount*, and *MTotalGroupAddrRxCount* statistics counters as needed.

<CTSM>**GetRCB** is called at process or privileged time. MLIDs that cannot handle fragmented receive buffers should get RCBs using either **CMSMAllocateRCB** or <CTSM>**ProcessGetRCB**.

<CTSM>**GetRCB** uses a lookahead process where the CTSM previews the packet header information before it gives the RCB to the CHSM. This allows the CTSM to verify that it wants the packet before the CHSM transfers the entire packet from the adapter to the RCB.

Note



The RCB might be fragmented.

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

- If a programmed I/O adapter is being used, the CHSM must transfer the packet header information from the adapter to a buffer maintained for this purpose. The number of bytes to transfer is specified by *CMSMMaxFrameHeaderSize*, which is described in Chapter 4, "CMSM/CTSM Data Structures and Variables". The CHSM must set *packetHdr* to point to the beginning of the lookahead buffer before calling this routine.
- If a shared RAM (memory-mapped I/O) adapter is used, the CHSM simply points *packetHdr* to the beginning of the packet buffer in shared RAM.

On entry to this routine, *packetHdr* must point to the packet's header information in the lookahead buffer, and *packetSize* must contain the size of the received packet. If the header information is verified, the CTSM will obtain an RCB and use the lookahead information to fill in the *RCBReserved* fields before it returns a pointer to the RCB.

After obtaining the RCB, the CHSM must transfer the remainder of the packet into the RCB fragment buffers. *nskip* is the offset from the beginning of the packet to start copying from, and *ntoRead* contains the number of bytes in the packet left to read.

Important



The address and buffer size contained in the RCB fragment list's `FRAGMENT_STRUCT` element **must not** be altered by the CHSM.

After the CHSM reads the rest of the packet, it must use `<CTSM>RcvComplete` and `CMSMSERVICEEVENTS` to return the RCB to the LSL.

Important



If this routine returns a NULL pointer, the CHSM should discard the received packet.

Bus Master Adapters

Bus master devices require preallocated RCBs. Since preallocation is not compatible with `<CTSM>GetRCB`, the CHSM for a bus mastering adapter uses `CMSMAllocateRCB` and `<CTSM>ProcessGetRCB`.

Pipeline Adapters

On pipeline adapters, the first part of the packet can be indicated while the rest of the packet is still being received. This condition is indicated by setting the contents of the packet size field to `UNUSED`. The CTSM assumes that at least `CMSMMaxFrameHeaderSize` bytes of the data are presented. See `<CTSM>RcvCompleteStatus` for information on returning the RCB. With pipeline adapters, the counters are updated when `<CTSM>RcvCompleteStatus` is called.

If a packet is not wanted, `<CTSM>GetRCB` will update the statistics counters as best it can; since the packet is not wanted, there is no way to know its size and other information for sure.

<CTSM>ProcessGetRCB

Called by the CHSM to process an RCB for a received packet and to preallocate a new nonfragmented RCB for the next packet.

Syntax

```
#include <odi.h>
#include <csm.h>
#include <<ctsm>.h>

RCB * <CTSM>ProcessGetRCB (
    DRIVER_DATA *driverData,
    RCB *rcb,
    UINT32 pktSize,
    UINT32 rcvStatus,
    UINT32 newRcbSize );
```

Input Parameters

driverData

Pointer to the HSM's driver adapter data space.

rcb

Pointer to the received packet's RCB.

pktSize

Size of the received packet including the MAC header.

rcvStatus

Status of received packet (see **DriverPromiscuousChange** in Chapter 5, "CHSM Functions").

newRcbSize

Minimum packet size for the new RCB.

Output Parameters

None.

Return Values

Pointer to a new nonfragmented RCB.

NULL No nonfragmented RCBs are available.

Remarks

The CHSM calls **<CTSM>ProcessGetRCB** at process or privileged time. The CHSM must have previously copied the contents of the received packet into the RCB data buffer.

Use this routine if the RCB was preallocated using **CMSMAllocateRCB** or was obtained from a previous call to this routine.

This routine increments the following statistics counters:

```
MTotalRxPacketCount
MTotalRxMiscCount
MPacketRxTooBigCount
MPacketRxTooSmallCount
MTotalGroupAddrRxCount
```

Note



If the adapter/MLID is ECB-aware and has already filled in all required ECB fields as described in Chapter 4, "CMSM/CTSM Data Structures and Variables", it should return the ECB for processing by using **<CTSM>RcvCompleteStatus** and **CMSMServiceEvents**. If the *MM_FRAGS_PHYS_BIT* bit of the *MLIDCFG_ModeFlags* field is set, the fragment offset of the RCB contains a physical pointer to the RCB data buffer.

The CHSM must eventually use **CMSMServiceEvents**, which enables the RCB's event service routine to complete the processing.

Ethernet

The CHSM starts copying the packet from the 6-byte destination field of the media header into the RCB data buffer.

Token-Ring

The CHSM starts copying the packet from the access control byte of the media header into the RCB data buffer.

FDDI

The CHSM starts copying the packet from the frame control byte of the media header into the RCB data buffer.

Note



For some busMaster implementations, you must set the first UINT32 parameter, starting at *RCBReserved[28]* (defined in CMSM.H), 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 for this function.

Important



Do not call this function until the packet has been received into host memory and is ready to be handed to the CTSM.

See Also

CMSMReturnRCB

<CTSM>RcvComplete

Called by the CHSM to direct a completed RCB to the LSL's holding queue to await processing.

Syntax

```
#include <cmsm.h>
#include <<ctsm>.h>

void <CTSM>RcvComplete (
    DRIVER_DATA    *driverData,
    RCB             *rcb );
```

Input Parameters

- driverData*
Pointer to the HSM'S driver adapter data space.
- rcb*
Pointer to the received packet's RCB.

Output Parameters

None.

Return Values

None.

Remarks

<CTSM>RcvComplete is called at process or privileged time. This routine increments the *MTotalRxPacketCount*, *MTotalRxOkByteCount*, and *MTotalGroupAddrRxCount* statistics counters for ECB_Aware adapters. Use this routine if the CHSM gets the RCB using the <CTSM>GetRCB function and copies the received packet into the RCB receive buffer(s).

When the CHSM uses this routine to queue an RCB, it must eventually use **CMSMServiceEvents** to call the RCB's event service routine and complete the processing.

<CTSM>RcvCompleteStatus

Allows the CTSM to fill in the packet length of the RCB fields, record the error status, and direct the RCB to the LSL's holding queue to await processing.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <<ctsm>.h>

void <CTSM>RcvCompleteStatus (
    DRIVER_DATA *driverData,
    RCB *rcb,
    UINT32 packetLength,
    UINT32 packetStatus );
```

Input Parameters

driverData

Pointer to the HSM'S driver adapter data space.

rcb

Pointer to the received packet's RCB.

packetLength

Size of the received packet including the MAC header.

packetStatus

Status of the received packet for stack monitoring functions (see **DriverPromiscuousChange** in Chapter 5, "CHSM Functions").

Output Parameters

None.

Return Values

None.

Remarks

Use this routine if a pipelined adapter obtained the RCB by calling **<CTSM>GetRCB** with *packetSize* equal to *UNUSED*.

When the CHSM uses this routine to queue an RCB, it must eventually use the **CMSMServiceEvents** macro to call the ECB's event service routine and complete the processing.

This function is called at process or privileged time.

<CTSM>RegisterHSM

Initially registers the CHSM with the CTSM and CMSM.

Syntax

```
#include <odi.h>
#include <<ctsm>.h>

ODISTAT <CTSM>RegisterHSM (
    DRIVER_PARM *DriverParameterBlock,
    MLID_CONFIG_TABLE **configTable );
```

Input Parameters

DriverParameterBlock
Pointer to the driver parameter block structure.

Output Parameters

configTable
Pointer to a pointer to the configuration table.

Return Values

| | |
|--------------------|--|
| ODISTAT_SUCCESSFUL | The CHSM was successfully registered with the CTSM and CMSM. |
| ODISTAT_FAIL | The CHSM was not successfully registered with the CTSM and CMSM. |

Remarks

<CTSM>**RegisterHSM** is called at initialization time only. The CHSM's *DriverInit* routine must call <CTSM>**RegisterHSM** with a pointer to its driver parameter block structure in *driverParm*. This routine calls the CMSM, which performs the following tasks:

- Copies the driver parameter block into local data space
- Processes driver firmware variables
- Allocates the frame data space
- Copies the driver configuration table template into the frame data space
- Parses information derived from the linker definition file

<CTSM>SendComplete

Called by the CHSM's **DriverSend** or **DriverISR** routine to return a TCB after a packet has been transmitted.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <<ctsm>.h>

void <CTSM>SendComplete (
    DRIVER_DATA *driverData,
    TCB *tcb,
    UINT32 transmitStatus );
```

Input Parameters

- driverData*
Pointer to the HSM'S driver adapter data space.
- tcb*
Pointer to the TCB. If the CHSM is ECB aware, then it is a pointer to an ECB that has been type cast to a TCB pointer.
- transmitStatus*
Used to flag whether a packet was really sent.
 - 0 Successful
 - nonzero Undeliverable

Output Parameter

None.

Return Values

None.

Remarks

<CTSM>SendComplete is called at process or privileged time.
<CTSM>SendComplete can be called before the actual transmission is complete (a "lying send"), as long as all packet data has been transferred into the adapter's transmit buffer.

This function returns the packet's TCB to the unused TCB queue and directs the underlying transmit ECB to the LSL's service queue.

The CHSM must eventually use **CMSMServiceEvents**, which calls the ECB's event service routine. Typically, if the **DriverSend** routine is called to transmit the next packet after a send complete interrupt, the interrupt service routine should invoke **CMSMServiceEvents**.

The *MTotalTxPacketCount*, *MTotalTxOkByteCount*, and *MTotalGroupAddrTxCount* statistics counters are updated.

Note



The **DriverSend** routine can use ECBs instead of TCBs by initializing the driver parameter block variable *DriverSendWantsECBs* to a nonzero value (see Chapter 3, "CHSM Data Structures and Variables"). In this case, **<CTSM>SendComplete** will simply direct the ECB to the LSL's service queue.

<CTSM>UpdateMulticast

Forces the CTSM to call **DriverMulticastChange**.

Syntax

```
#include <<ctsm>.h>

ODISTAT <CTSM>UpdateMulticast (
    DRIVER_DATA *driverData );
```

Input Parameters

driverData
Pointer to the HSM’S driver adapter data space.

Output Parameters

None.

Return Values

| | |
|--------------------------|--|
| ODISTAT_SUCCESSFUL | The requested operation was completed successfully. |
| ODISTAT_RESPONSE_DELAYED | The function cannot complete immediately; this is due to the asynchronous nature of this function. |
| ODISTAT_FAIL | The requested operation could not be completed. |

Remarks

The CHSM can call **<CTSM>UpdateMulticast** at process or privileged time, but it is generally called by **DriverReset**. When this routine is called, it passes the current multicast table (maintained by the CTSM) to the CHSM’s **DriverMulticastChange** routine. This allows the driver to update the adapter’s multicast address registers.

This routine is called by internal CTSM functions each time multicast addresses are added to or deleted from the CMSM's multicast table. The MLID can also call this routine during the CHSM's **DriverReset** routine.

Refer to the sections in Chapter 3, "CHSM Data Structures and Variables" covering the following flags and variables for more information on multicast addressing:

- *MM_MULTICAST_BIT* bit of the *MLIDCFG_ModeFlags* field is used to indicate whether or not multicast addressing is supported.
- *MF_SOFT_FILT_GRP_BIT* and *MF_GRP_ADDR_SUP_BIT* bits of the *MLIDCFG_Flags* field must be set appropriately to reflect the multicast filtering mechanism used by the adapter/driver.
- The driver parameter block variable, *DriverMaxMulticast*, must be set to reflect the maximum number of multicast addresses the adapter can handle.

6-34 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

chapter **7** *CMSM Functions*

Overview

This chapter describes the C language Media Support Module (CMSM) functions provided as tools for CHSM developers. These CMSM functions, along with the topology-specific functions described in Chapter 6, manage the primary details of interfacing the CHSM to the LSL and the operating system. The functions in this chapter are media independent and handle generic initialization and run-time issues.

The functions included in this chapter are designed to shield the MLID from future operating system changes. These functions are defined in the *cmsm.h* file. If an operating system call changes, we will modify the corresponding functions in *cmsm.h*. Then, instead of modifying your existing CHSM, you can simply recompile it with the new *cmsm.h*.

CMSMAddToCounter

Adds a user-specified value to the counter pointed to by STAT_TABLE_ENTRY.

Syntax

```
#include <odi.h>
#include <cmsm.h>

void CMSMAddToCounter (
    STAT_TABLE_ENTRY *statTableEntryPtr,
    UINT32 value );
```

Input Parameters

StatTableEntryPtr

Pointer to the statistics table entry whose counter is to be incremented by *value*.

value

The value to increment the counter by.

Output Parameters

None.

Return Values

None.

Remarks

This function is intended to simplify the process of adding a value to a 64-bit counter, but it can also be used for 32-bit counters.

See Also

CMSMIncrCounter

For more information on STAT_TABLE_ENTRY, see the statistics table information in Chapter 3, "CHSM Data Structures and Variables".

CMSMAIloc

Used by the CHSM to allocate memory at process time.

Syntax

```
#include <odi.h>
#include <cmsm.h>

void  *CMSMAIloc (
    CONST DRIVER_DATA  *driverData,
    UINT32  nbytes );
```

Input Parameters

- driverData*
Pointer to the HSM’S driver adapter data space.
- nbytes*
Number of bytes of memory to allocate.

Output Parameters

None.

Return Values

If successful, **CMSMAIloc** returns a pointer to the allocated space. Otherwise, it returns NULL.

Remarks

- The CHSM must return the buffer allocated to **CMSMAIloc** at shutdown time using **CMSMFree**.
- If the driver parameter block variable, *DriverNeedsBelow16Meg*, is initialized to 1 (see Chapter 3, "CHSM Data Structures and Variables"), the CMSM will allocate memory below the 16MB boundary.

The memory allocated by **CMSMAlloc** is logically contiguous, but may not be physically contiguous.

Example

```
/* Allocate memory for transmit buffers if using Am7990 LANCE chipset.*/

if ( (configTable->MLIDCFG_ModeFlags & MM_FRAGS_PHYS_BIT) == 0) {
    driverData->TxBuffers =CMSMAlloc(driverData, BUFFER_SIZE *TX_BUFFERS );
    if (driverData->TxBuffers == 0) {
        CMSMPrintString(configTable, MSG_TYPE_INIT_ERROR,
            MSG("073: Unable to allocate memory.\n\r", 39), 0 ,0);
        CMSMReturnDriverResources(configTable);
        return-1;
    }
}
```

CMSMAllocateMultipleRCBs

Allocates a block of RCBs for packets to be received by the CHSM.

Syntax

```
#include <odi.h>
#include <cmsm.h>

RCB * CMSMAllocateMultipleRCBs (
    const   Driver_Data    *driverData,
    UINT32   nbytes,
    UINT32   *nRCBs,
    void     **physicalRCB);
```

Input Parameters

driverData

Pointer to the CHSM's driver adapter data space.

nbytes

Number of data bytes of memory to allocate per RCB. As this call is to be used for preallocation of RCBs, this value should be set to the value of the MLIDCFG_MaxFrameSize field of the configuration table (see remarks below).

nRCBs

Pointer to the number of RCBs to be allocated.

physicalRCB

Address of the pointer to the physical address of the first RCB in list. Set to NULL if physical addresses are not needed.

Output Parameters

nRCBs

Pointer to the number of RCBs actually allocated. This routine will update the input parameter, in the case of requested RCBs equals RCBs allocated, the value will remain the same.

physicalRCB

Pointer to the physical address of the first RCB on the linked list of RCBs.
Set to NULL if there was an allocation error.

Return Values

If successful, CMSMAllocateMultipleRCBs returns a pointer to the first RCB allocated in the linked list. If Unsuccessful, CMSMAllocateMultipleRCBs returns a NULL.

Remarks

If the number of RCBs available is less than nRCBs, then the procedure will allocate as many RCBs as are available. If there are no available RCBs, then the procedure will return a NULL, and the C MSM will increment the NoECBAvailableCount statistics counter. If nRCBs is zero at call time, the procedure will return with null pointers and the NoECBAvailableCount will not be incremented.

The returned RCBs will be nonfragmented and large enough to hold the received packet frame. The length passed in nbytes includes the length of all protocol and hardware headers, but does not include the size of the RCB itself.

C HSMs that support bus mastering DMA adapters may use this routine to pre allocate blocks of RCBs. If the MM_FRAGS_PHYS_BIT bit of the MLIDCDFG_ModeFlags field is set, PhysicalRCB will be set to the physical address of the first RCB on the linked list, and the fragment pointer will contain a physical pointer. After the adapter copies the packet into the fragment of the RCB, the C HSM uses <CTSM>ProcessGetRCB to return each RCB to the C MSM for processing and get a replacement RCB. CMSMReturnRCB can be used to return individual unused RCBs to the CMSM without processing. CMSMReturnMultipleRCBs can be used to return lists of unused RCBs to the C MSM without processing them.

If the adapter is ECB aware (see 'Event Control Blocks (ECBs)') and has previously filled in all the RCB fields according to the ODI specification, the C HSM should call <CTSM>RcvComplete.



Note

This procedure is designed for speed and performance, and does not incorporate all of the checking found in CMSMAllocateRCB. If the driver requires the RCBs to be allocated below the 16 MegaByte boundary, this procedure should not be used.

The fields `RCB->RCBDriverWS.RWs_i32val[0]` and `RCB->RCBDriverWS.RWs_i32val[1]` will be returned as 32 bit pointers to the next RCB on the list. `RCB->RCBDriverWS.RWs_i32val[0]` will contain the logical address of the next RCB, while `RCB->RCBDriverWS.RWs_i32val[1]` will contain the physical address (when needed). The remaining operation and description of the RCB will be unchanged.

Ethernet

The C HSM starts copying the packet from the 6-byte destination field of the media header into the RCB `RCBDataBuffer` field.

Token-Ring

The C HSM starts copying the packet from the access control byte of the media header into the RCB's `RCBDataBuffer` field.

FDDI

The C HSM starts copying the packet from the frame control byte of the media header into the RCB's `RCBDataBuffer` field.

See Also

`CMSMAllocateRCB`
`CMSMReturnRCB`
`CMSMReturnMultipleRCBs`

CMSMAllocPages

Allocates a system, page-aligned, memory buffer at process time.

Syntax

```
#include <odi.h>
#include <cmsm.h>

void *CMSMAllocPages (
    const DRIVER_DATA *driverData,
    UINT32 nbytes ),
```

Input Parameters

driverData

Pointer to the CHSM'S driver adapter data space.

nbytes

Number of bytes of memory to allocate.

Output Parameters

None.

Return Values

If successful, **CMSMAllocPages** returns a pointer to the allocated space. Otherwise, it returns NULL.

Remarks

The CHSM calls **CMSMAllocPages** at process time only; **CMSMAllocPages** returns a pointer to the allocated buffer. If the routine is unsuccessful, it returns 0. The CHSM must return this buffer at shutdown time using **CMSMFreePages**.

If the driver parameter block variable *DriverNeedsBelow16Meg* is initialized to 1 (see Chapter 3, "CHSM Data Structures and Variables"), the CMSM will allocate memory below the 16MB boundary.

CMSMAllocateRCB

Allocates an RCB for a packet received by the CHSM, or preallocates an RCB for a packet the CHSM will receive.

Syntax

```
#include <odi.h>
#include <cmsm.h>

RCB * CMSMAllocateRCB (
    const    DRIVER_DATA  *driverData,
    UINT32   nbytes,
    void     **physicalRCB );
```

Input Parameters

driverData

Pointer to the CHSM'S driver adapter data space.

nbytes

Number of bytes of memory to allocate.

Output Parameters

physicalRCB

Pointer to a pointer to the RCB's physical address. Usually, this parameter is used only by ECB-aware, C HSMs. Set to 0 if not needed.

Return Values

If successful, **CMSMAllocateRCB** returns a pointer to the allocated RCB. Otherwise, it returns NULL.

Remarks

The returned RCB will be nonfragmented and large enough to hold the received packet frame. The length passed in *nbytes* includes the length of all protocol and hardware headers. If an RCB is not available, the CTSM increments the *MNoECBAvailableCount* statistics counter, and the CHSM discards the packet.

CHSMs that support bus mastering DMA adapters use this routine to preallocate RCBs. In this case, the CHSM sets *nbytes* to the maximum packet size specified by the *MLIDCFG_MaxFrameSize* field of the configuration table before using **CMSMAllocateRCB**. If the *MM_FRAGS_PHYS_BIT* bit of the *MLIDCFG_ModeFlags* field is set. The fragment pointer will contain a physical pointer.

After the adapter copies the packet into the fragment of the RCB, the CHSM uses **<CTSM>ProcessGetRCB** to return the RCB to the CMSM. If the adapter is ECB aware (see Appendix B, "Event Control Blocks (ECBs)") and has previously filled in all the RCB fields according to the ODI specification, the CHSM calls **<CTSM>RcvComplete**.

If *DriverNeedsBelow16Meg* of the driver parameter block is initialized to 1 (see Chapter 3, "CHSM Data Structures and Variables"), the CMSM allocates the RCB in memory below the 16MB boundary.

Ethernet

The CHSM starts copying the packet from the 6-byte destination field of the media header into the RCB *RCBDataBuffer* field.

Token-Ring

The CHSM starts copying the packet from the access control byte of the media header into the RCB's *RCBDataBuffer* field.

FDDI

The CHSM starts copying the packet from the frame control byte of the media header into the RCB's *RCBDataBuffer* field.

See Also

CMSMReturnRCB

7-12 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

CMSMCancelAES

Called to cancel an AES event.

Syntax

```
#include <odi.h>
#include <cmsm.h>

ODISTAT CMSMCancelAES (
    DriverData      *driverData,
    MLID_AES_ECB    *mlidAESECB );
```

Input Parameters

- driverData*
Pointer to the C HSM's driver adapter data space.
- mlidAESECB*
Pointer to the MLID_AES_ECB structure of the AES to be canceled.

Output Parameters

None.

Return Values

| | |
|-----------------------|--------------------------------------|
| ODISTAT_SUCCESSFUL | Call back was successfully Canceled. |
| ODISTAT_BAD_PARAMETER | AES was not active. |

Remarks

This function is called to cancel an AES event that was scheduled using CMSMScheduleAES.

See Also

CMSMScheduleAES

CMSMControlComplete

Called to notify the CMSM that the previously scheduled event has completed.

Syntax

```
#include <odi.h>
#include <cmsm.h>

void CMSMControlComplete (
    DRIVER_DATA *driverData,
    CHSM_COMPLETE controlFunction,
    ODISTAT completionStatus );
```

Input Parameters

- driverData*
Pointer to the CHSM’S driver adapter data space.
- controlFunction*
The control function that has completed.
- completionStatus*
The return states of the completed event.

Output Parameters

None.

Return Values

None.

Remarks

This function should not be used if the CHSM can quickly and synchronously complete all of the control functions. This function is used by the CHSM if it

returned `ODISTAT_RESPONSE_DELAYED` for a given control procedure to signal that the event completed.

Important



You can only have one outstanding scheduled event.

CHSM_COMPLETE Enumeration

```
typedef enum _CHSM_COMPLETE_
{
    CHSM_COMPLETE_STATISTICS,
    CHSM_COMPLETE_MULTICAST,
    CHSM_COMPLETE_SHUTDOWN,
    CHSM_COMPLETE_RESET,
    CHSM_COMPLETE_LOOK_AHEAD,
    CHSM_COMPLETE_PROMISCUOUS,
    CHSM_COMPLETE_MANAGEMENT,
    CHSM_COMPLETE_RESERVED
} CHSM_COMPLETE;
```

See Also

DriverReset
 DriverShutdown
 DriverMulticastChange
 DriverPromiscuousChange
 DriverStatisticsChange
 DriverRxLookAheadChange
 DriverManagement

CMSMDeRegisterResource

Allows a C HSM to deregister resources registered with **CMSMRegisterResource**.

Syntax

```
#include <odi.h>
#include <cmsm.h>

ODISTAT CMSMDeRegisterResource (
    DRIVER_DATA *driverData,
    EXTRA_CONFIG *extraConfig,
    ECB *pAsyncECB);
```

Input Parameters

driverData

Pointer to the CHSM's driver adapter data space.

extraConfig

Pointer to an EXTRA_CONFIG structure that contains the hardware options to be deregistered. This pointer must be the same pointer used to register the resources in **CMSMRegisterResource**.

pAsyncECB

Pointer to an ECB whose ESR ((*ECB_ESR)(ECB*)) is called if **CMSMDeRegisterResource** returns ODISTAT_RESPONSE_DELAYED.

The ESR is called with *pAsyncECB* as a parameter and the pAsyncECB ECB_Status field will contain the return value.

Other ECB fields may be used by the C HSM to store context or other information that is needed by the ESR.


Output Parameters

None.

Return Values

| | |
|--------------------------|---|
| ODISTAT_SUCCESSFUL | The resources contained in the extraConfig parameter were successfully deregistered. |
| ODISTAT_BAD_PARAMETER | An input parameter was invalid or the call was made at interrupttime and the pAsyncECB parameter was a NULL. |
| ODISTAT_FAIL | The adapter was not in a shutdown state before the call was made, another C MSM API returned ODISTAT_RESPONSE_DELAYED and has not completed when this routine was called, or an unknown error occurred. |
| ODISTAT_ITEM_NOT_PRESENT | The extraConfig pointer was not found in the list of extraConfig pointers used in calls to CMSMRegisterResource . |
| ODISTAT_RESPONSE_DELAYED | The operation of de-registering resources could not be completed at the present time. An asynchronous process will be scheduled to complete the operation at a later time. |

Remarks

Note  After **CMSMRegisterMLID** has been called, but before **CMSMDeRegisterResource** is called, the adapter must be placed in a shutdown state by calling **CMSMShutdownMLID**.

CMSMDeRegisterResource will deregister those resources found in extraConfig's substructure IOConfig. The resources must previously have been registered through **CMSMRegisterResource** using the same extraConfig pointer.

If **CMSMDeRegisterResource** 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 pAsyncECB's ESR routine will be called to report the final return value of the operation. The return value will be stored in the pAsyncECB's ECB_Status field.

Upon successful return from **CMSMDeRegisterResource** or from the asynchronous process, the CHSM is responsible for putting the adapter in a functional state. If additional resources of an EXTRA_CONFIG nature are required, the CHSM must call **CMSMRegisterResource** to register the additional resources.

CMSMDeRegisterResource upon successful completion will produce a NESL Service/Status Change event to inform consumers that the configuration of the adapter has been updated.

See Also

CMSMRegisterResource
CMSMShutdownMLID

CMSMDriverRemove

Called by the CHSM's **DriverRemove** function to deregister the CHSM and return all CHSM resources allocated by the CMSM or the CTSM.

Syntax

```
#include <cmsm.h>

void CMSMDriverRemove (
    MODULE_HANDLE *moduleHandle );
```

Input Parameters

moduleHandle

The module load handle passed to **DriverInit** and placed in *DriverModuleHandle* of the driver parameter block.

Output Parameters

None.

Return Values

None.

Remarks

CMSMDriverRemove calls the CHSM's **DriverShutdown** routine before returning.

CMSMECBPhysToLogFrgs

For transmissions, if MM_FRAGS_PHYS_BIT is set and the adapter is ECB aware, this function gets the address of the ECB, which contains the FRAGMENT_LIST_STRUCTURE of the logical addresses of the fragments in the ECB.

Syntax

```
#include <odi.h>
#include <cmsm.h>

ECB *CMSMECBPhysToLogFrgs (
    ECB *ecb );
```

Input Parameters

ecb
Pointer to an ECB structure.

Output Parameters

None.

Return Values

Pointer to an ECB structure containing logical addresses for the ECB fragments.

Remarks

You cannot assume that the fragment pointers have a one-to-one correspondence. Because the physical pointers point to fragments that are physically contiguous, there can be more fragments in the physical list than in the logical list.

The ECBs *ECB_PreviousLink* and *ECB_ESR* fields must not be changed.

The ECB containing a `FRAGMENT_LIST_STRUCTURE` of logical addresses acquired with this function is not returned directly to the system by the HSM. The TSM returns it to the system when one of the Send Complete APIs has been called for the ECB passed in as the input parameter for this function. Once a Send Complete API has been called, the HSM no longer has ownership of either ECB and must not reference or modify either ECB.

FRAGMENT_LIST_STRUCT Structure

```
typedef struct _FRAGMENT_LIST_STRUCT_  
{  
    UINT32          FragmentCount;  
    FRAGMENT_STRUCT FragmentStruct;  
} FRAGMENT_LIST_STRUCT;
```

Field Descriptions:

FragmentCount

The number of fragments.

FragmentStruct

Specifies a fragment structure.

CMSMEnablePolling

Used during **DriverInit** to enable the operating system to periodically call **DriverPoll** if the CHSM's board service routine is poll-driven.

Syntax

```
#include <cmsm.h>

ODISTAT CMSMEnablePolling (
    const DRIVER_DATA *driverData );
```

Input Parameters

driverData
Pointer to the CHSM's driver adapter data space.

Output Parameters


None.

Return Values

| | |
|-----------------------|---|
| ODISTAT_SUCCESSFUL | The requested operation was completed successfully. |
| ODISTAT_BAD_PARAMETER | The <i>DriverPollPtr</i> field of <i>DriverParameterBlock</i> is set to NULL. |

Remarks

The **DriverPoll** routine polls the adapter to determine if any send or receive events have occurred. An implied input from the **DriverPollPtr** field of the driver parameter block points to the function to be called.

Important  This routine will not relinquish control to other procedures during execution.

See Also

CMSMGetPollSupportLevel
CMSMSetHardwareInterrupt

CMSMFree

Must be used by the CHSM before it permanently shuts down, to return any memory allocated with **CMSMAlloc** or **CMSMInitAlloc**.

Syntax

```
#include <cmsm.h>

void CMSMFree (
    const DRIVER_DATA *driverData,
    void *dataPtr );
```

Input Parameters

driverData

Pointer to the CHSM's driver adapter data space. This parameter must contain NULL if memory was allocated using **CMSMInitAlloc**.

dataPtr

Pointer to the data allocated using **CMSMAlloc** or **CMSMInitAlloc**.

Output Parameters

None.

Return Values

None.

Remarks

CMSMFree must be called at process time.

Example

The following code is used in **DriverShutdown**.

```
if (shutdownType == PERMANENT_SHUTDOWN) {
    if (driverData->xyzBuffers) {
        CMSMFree(driverData, driverData->xyzBuffers);
        driverData->xyzBuffers = 0;
    }
    DMACleanup((UINT32)configTable->MLIDCFG_DMALine0);
}
```

CMSMFreePages

Returns the system, page-aligned memory buffers allocated by **CMSMAllocPages**.

Syntax

```
#include <cmsm.h>

void CMSMFreePages (
    const DRIVER_DATA *driverData,
    void *dataPtr );
```

Input Parameters

driverData

Pointer space.

dataPtr

Pointer to the memory allocated using **CMSMAllocPages**.

Output Parameters

None.

Return Values

None.

Remarks

CMSMFreePages must be called at process time.

If the CHSM allocates memory using **CMSMAllocPages**, this function must be called before the CHSM is permanently shut down.

CMSMGetAlignment

Called to obtain the alignment requirements of the underlying platform.

Syntax

```
#include <odi.h>
#include <cmsm.h>

UINT32 CMSMGetAlignment (
    UINT32  type );
```

Input Parameters

type

- 0 - alignment requirement
- 1 - best case alignment
- Other - undefined

Output Parameters

None.

Return Values

Power of 2 byte boundary data alignment requirement.

Remarks

If *type* equals 0, this function returns the worst-case data alignment requirement of the data object involved in the I/O transfer. This arbitrary type allows the platform to function without exceptions or corrupted data.

All operations and "real world" use of these operations should be considered in determining this value. That is, if DMAing into an arbitrary memory location can cause data corruption due to noncoherent caching, then this function should return a value equal to at least the cache line size. Without this function, you

cannot write platform-independent DMA code, since the code cannot determine what characteristics it must meet.

If *type* is equal to 1, this function returns the data alignment requirement for the platform to function at its best performance.

The value returned for *type* equal to 0 should always be less than or equal to the value returned for *type* equal to 1.

For most Intel processor based 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). An HP PA-RISC machine should return 32 for both *type* equal to 0 and *type* equal to 1, due to the requirements of the memory cache.

Before using this function, compile your CHSM using the #define for your hardware type. #defines are found in the file: PORTABLE.H.

Example:

- #define IAPX386
- #define MC680X0
- #define MC88000 (Motorola RISC)
- #define RX000 (MIPS)

CMSMGetBusInfo

Returns the size of the bus addresses associated with *busTag*.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <odi_nbi.h>

ODI_NBI CMSMGetBusInfo (
    void      *busTag,
    UINT32    *physicalMemAddrSize,
    UINT32    *ioAddrSize );
```

Input Parameters

busTag
Architecture-dependent value, returned by **CMSMSearchAdapter** or **CMSMScanBusInfo**, that identifies a specific bus.

Output Parameters

physicalMemAddrSize
The size in bits of a physical address on the bus specified by *busTag*.

ioAddrSize
The size in bits of an I/O address on the bus specified by *busTag*.

Return Values

| | |
|--------------------------|---|
| ODI_NBI_SUCCESSFUL | The requested operation was completed successfully. |
| ODI_NBI_ITEM_NOT_PRESENT | The specified bus does not exist. |

CMSMGetBusSpecificInfo

Returns supplementary information about the specified bus.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <odi_nbi.h>

ODI_NBI CMSMGetBusSpecificInfo (
    VOID      *busTag,
    UINT32     size
    VOID      *busSpecificInfo );
```

Input Parameters

- busTag*
Architecture-dependent value, returned by **CMSMSearchAdapter** or **CMSMScanBusInfo**, that identifies a specific bus.
- size*
The size of the caller’s buffer, pointed to by *busSpecificInfo*.

Output Parameters

- busSpecificInfo*
Pointer to the buffer whose size is an input parameter.

Return Values

| | |
|-------------------------------|---|
| ODI_NBI_SUCCESSFUL | The operation was completed successfully. |
| ODI_NBI_UNSUPPORTED_OPERATION | The bus has no supplementary information to return. |
| ODI_NBI_PARAMETER_ERROR | An input parameter was invalid. |

Remarks

The following information is returned for the specified bus:

```
PnP ISA Bus
size 48
busSpecificInfo (as follows)

struct ISAInfoStructure
{
    UINT32 PnPISABIOSPresentFlag;
    UINT16 PnPISABIOSMajorVer;
    UINT16 PnPISABIOSMinorVer;
    UINT16 PnPISABIOSRevision;
    UINT32 PnPISACMPresentFlag;
    UINT32 PnPISACMType;
    UINT16 PnPISACMTypeMajorVer;
    UINT16 PnPISACMMinorVer;
    UINT16 PnPISACMRevision;
    UINT32 NetFrameFlag;
    UINT32 NonATCompatibleFlag;
    UINT32 HardwareLoaderID;
    UINT32 ISAInfoReserved1;
    UINT32 ISAInfoReserved2;
    UINT32 ISAInfoReserved3;
};
```

PC Card and CardBus Buses

The following CardBus definition applies only to hardware not using the common silicon method defined by the *CardBus PC Card/PCI Common Silicon Requirements* guideline. For CardBus adapters using the common silicon method, refer to the definition for the PCI Bus.

```
size 40
busSpecificInfo (as follows)

struct PCCardInfoStructure
{
    UINT32 CSPresentFlag;
    UINT32 CStype;
    UINT16 CSVendorMajorVer;
    UINT16 CSVendorMinorVer;
    UINT8 *CSVendorNamePtr;
    UINT16 CSInterfaceLevelMajorVer;
    UINT16 CSInterfaceLevelMinorVer;
    UINT32 CSNumberOfSockets;
    UINT32 PCCardInfoReserved0;
    UINT32 PCCardInfoReserved1;
    UINT32 PCCardInfoReserved2;
    UINT32 PCCardInfoReserved3;
};
```

PCI Bus*size 32**busSpecificInfo* (as follows)

```
struct PCIInfoStructure
{
    UINT32 PCIBIOSPresentFlag;
    UINT16 PCIInterfaceLevelMajorVer;
    UINT16 PCIInterfaceLevelMinorVer;
    UINT32 PCIHardwareMechanism;
    UINT32 LastPCIBusInSystem;
    UINT32 PCIInfoReserved0;
    UINT32 PCIInfoReserved1;
    UINT32 PCIInfoReserved2;
    UINT32 PCIInfoReserved3;
};
```

CMSMGetBusType

Returns a value that indicates the bus type of the bus specified by *busTag*.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <odi_nbi.h>


ODI_NBI CMSMGetBusType (
    void      *busTag,
    UINT32    *busType );
```

Input Parameters

busTag
Architecture-dependent value, returned by **CMSMSearchAdapter** or **CMSMScanBusInfo**, that identifies a specific bus.

Output Parameters

busType (Defined in *odi_nbi.h*)
A value that indicates the type of bus.

Note  The ODI_BUSTYPE_CARDBUS type value is used only for hardware not using the common silicon method defined by the *CardBus PC Card/PCI Common Silicon Requirements* guideline. For CardBus adapters using the common silicon method refer to the ODI_BUSTYPE_PCI type value.

The following bus type values are the defined:

| Type Value | Bus |
|---------------------|---------------------------|
| ODI_BUSTYPE_ISA | ISA / ISA PnP bus |
| ODI_BUSTYPE_MCA | Micro Channel bus |
| ODI_BUSTYPE_EISA | EISA bus |
| ODI_BUSTYPE_PCMCIA | PCMCIA bus |
| ODI_BUSTYPE_PCI | PCI bus |
| ODI_BUSTYPE_NUBUS | NuBus bus |
| ODI_BUSTYPE_OFM | Open Firmware motherboard |
| ODI_BUSTYPE_VESA | VESA Local bus |
| ODI_BUSTYPE_CARDBUS | CardBus bus |

Return Values

| | |
|-------------------------|---|
| ODI_NBI_SUCCESSFUL | The requested operation was completed successfully. |
| ODI_NBI_PARAMETER_ERROR | <i>busTag</i> is invalid. |

Remarks

This function 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 ODI_BUSTYPE_EISA.

CMSMGetCardConfigInfo

Retrieves and returns configuration information for bus architectures that keep information on a per device basis.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <odi_nbi.h>

ODI_NBI CMSMGetCardConfigInfo (
    void      *busTag,
    UINT32    uniqueIdentifier,
    UINT32    size,
    UINT32    parm1,
    UINT32    parm2,
    void      *configInfo );
```

Input Parameters

busTag

Architecture-dependent value, returned by **CMSMSearchAdapter**, that identifies a specific bus.

uniqueIdentifier

Architecture-dependent value returned by **CMSMGetUniqueIdentifier** or **CMSMSearchAdapter** that identifies a specific device or function.

size

Number of bytes to be returned into the configuration buffer.

parm1

A bus architecture-dependent value that further specifies what information is to be returned, independent of the particular platform (because it is platform independent) and independent of what adapter is described by this information.

parm2

Architecture-dependent value that further specifies what information is to be returned, independent of the particular platform (because it is platform independent) and independent of what adapter is described by this information.

Output Parameters

configInfo

A pointer to a buffer used to receive the returned information. The caller needs to be sure that the buffer is at least *size* bytes long.

Return Values

| | |
|-------------------------------|---|
| ODI_NBI_SUCCESSFUL | The requested operation was completed successfully. |
| ODI_NBI_PARAMETER_ERROR | One of the parameters was invalid. |
| ODI_NBI_UNSUPPORTED_OPERATION | <i>busTag</i> denotes a bus type that has no configuration information. |
| ODI_NBI_ITEM_NOT_PRESENT | The <i>uniqueIdentifier</i> that was passed in has no card present. |
| ODI_NBI_FAIL | All of the input parameters appeared to be valid, but the operation could not be completed. |
| ODI_NBI_BUS_SPECIFIC_ERROR | A bus specific error occurred. |

Remarks

Call **CMSMGetCardConfigInfo** only if *busTag* identifies a bus whose architecture keeps configuration information on a per-device basis. It is the caller's responsibility to know how much and what sort of information is returned, so that *configInfo* is set pointing 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 of a particular bus, but will vary from one bus to another. One or both of these parameters can 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 EISA configuration information for the specified <i>uniqueIdentifier</i> . |

For a definition of the information returned, see the *EISA Specification*.

Micro Channel Bus

| | |
|------------|--|
| size | 8 |
| parm1 | n/a |
| parm2 | n/a |
| configInfo | Filled in with I/O port values from P0S0 - P0S7 (100h - 107h) for the <i>uniqueIdentifier</i> specified. |

For a definition of the information returned, see the *Personal System/2 Hardware Interface Technical Reference*.

PCI Bus

| | |
|------------|--|
| size | 256 |
| parm | n/a |
| parm2 | n/a |
| configInfo | Filled in with PCI configuration information for the specified <i>uniqueIdentifier</i> . |

For a definition of the information returned, see the *PCI Local Bus Specification*.

PNP ISA

| | |
|------------|--|
| size | 512 |
| parm1 | n/a |
| parm2 | n/a |
| configInfo | Filled in with Plug and Play configuration information for the specified <i>uniqueIdentifier</i> . The CMSM_PNP_ISA_CONFIG_INFO structure defines the information returned as follows: |

```
typedef struct _CMSM_PNP_ISA_CONFIG_INFO_  
{  
    UINT32    CPNPBusID;  
    UINT32    CPNPDeviceID;  
    UINT32    CPNPSerialNumber;  
    UINT32    CPNPLogicalID;  
    UINT32    CPNPFlags;  
    UINT8     CPNPCSN;  
    UINT8     CPNPLogicalDevNum;  
    UINT16    CPNPReadDataPort;  
    UINT16    CPNPNumMemWindows;  
    UINT32    CPNPMemBase[CMAX_MEM_REGS];  
    UINT32    CPNPMemLength[CMAX_MEM_REGS];  
    UINT16    CPNPMemAttrib[CMAX_MEM_REGS];  
    UINT16    CPNPNumIOPorts;  
    UINT16    CPNPIOPortBase[CMAX_IO_PORTS];  
    UINT8     CPNPIOPortLength[CMAX_IO_PORTS];  
    UINT16    CPNPNumIRQs;  
    UINT8     CPNPIRQRegisters[CMAX_IRQS];  
    UINT8     CPNPIRQAttrib[CMAX_IRQS];  
    UINT16    CPNPNumDMAs;  
    UINT8     CPNPDMAList[CMAX_DMAS];  
    UINT16    CPNPDMAAttrib[CMAX_DMAS];  
    UINT8     CPNPVendorDefined[CMAX_VDS];  
} CMSM_PNP_ISA_CONFIG_INFO;
```

PC Card (PCMCIA) Bus

| | |
|-------|--|
| size | The size of the buffer needed to contain the information defined by <i>parm2</i> . |
| 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 ODI_NBI_PARAMETER_ERROR, 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 <i>configInfo</i> buffer. The following values are valid for <i>parm2</i> : |

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 by 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.

For a definition of the information returned, see *PC Card Standards*.

CardBus Bus

Note  The following CardBus definition applies only to hardware not using the common silicon method defined by the *CardBus PC Card/PCI Common Silicon Requirements* guideline. For CardBus adapters using the common silicon method, refer to the definition for the PCI 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 ODI_NBI_PARAMETER_ERROR, 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.

For a definition of the information returned, see *PC Card Standards* and the *PCI Local Bus Specification*.

CMSMGetConfigInfo

Allows a C HSM to get the configuration information for the C MSM, including module and ODI specification versions.

Syntax

```
#include <odi.h>
#include <cmsm.h>

ODISTAT CMSMGetConfigInfo(
    void      *configInfo,
    UINT32    *nBytes);
```

Input Parameters

nBytes

Pointer to the requested number of bytes to be returned into the buffer.

Output Parameters

configInfo

A pointer to a buffer used to receive the returned configuration information. The caller needs to be sure that the buffer is at least *nBytes* bytes long.

nBytes

Pointer to the number of bytes returned in the configuration buffer.

Return Values

| | |
|-----------------------|---|
| ODISTAT_SUCCESSFUL | The configuration information of <i>nBytes</i> was successfully returned in the buffer. |
| ODISTAT_BAD_PARAMETER | The <i>nBytes</i> requested was larger than the actual configuration information available. The number of bytes of the configuration table actually returned in the buffer is indicated by the output parameter <i>nBytes</i> . |

Remarks

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

See Also

<CTSM>GetConfigInfo

CMSMGetCurrentTime

Determines the elapsed time (using the current relative time) for some of the CHSM-related activities.

Syntax

```
#include <odi.h>
#include <cmsm.h>

UINT32 CMSMGetCurrentTime ( void );
```

Input Parameters

None.

Output Parameters

None.

Return Values

A 32-bit value in 1/18 second clock ticks.

Remarks

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 over 7 years to roll over.) For finer resolution, use **CMSMGetMicroTimer**.

Example

```
UINT32  time1, time2, elapsedTime;
.
.
elapsedTime = 0;
time1 = CMSMGetCurrentTime ();
/* wait for 1 second */
while (elapsedTime < 18 )
{
    time2 = CMSMGetCurrentTime();
    elapsedTime = time2 - time1;
}
```

See Also

CMSMGetMicroTimer

CMSMGetHINFromHINName

Gets the Hardware Instance Number (HIN) associated with a HIN name.

Syntax

```
#include <odi.h>
#include <chsm.h>
#include <odi_nbi.h>

ODI_NBI CMSMGetHINFromHINName (
    MEON_STRING *hinName,
    UINT16 *hin);
```

Input Parameters

hinName
Pointer to a NULL terminated string which represents the HIN name. The string (including the termination) can not exceed MAX_HIN_NAME_SIZE.

Output Parameters

hin
HIN associated with the HIN name.

Return Values

| | |
|-------------------------------|---|
| ODI_NBI_SUCCESSFUL | The requested operation was completed successfully. |
| ODI_NBI_PARAMETER_ERROR | The specified HIN name is invalid. |
| ODI_NBI_UNSUPPORTED_OPERATION | This function is not available. |

Remarks

The HIN name is compared with existing HIN names in the system and the corresponding HIN is returned.

See Also

CMSMGetHINFromHINName

CMSMGetHINNameFromHIN

Gets the name associated with a Hardware Instance Number (HIN).

Syntax

```
#include <odi.h>
#include <chsm.h>
#include <odi_nbi.h>

ODI_NBI CMSMGetHINNameFromHIN (
    UINT16      hin,
    MEON_STRING *hinName );
```

Input Parameters

hin
The HIN for which the name is being requested.

Input/Output Parameters

hinName
Pointer to a buffer (provided by the caller) of MAX_HIN_NAME_SIZE that receives the NULL terminated HIN name string.

Return Values

| | |
|-------------------------------|---|
| ODI_NBI_SUCCESSFUL | The requested operation was completed successfully. |
| ODI_NBI_INSTANCE_NONEXIST | The specified HIN is invalid. |
| ODI_NBI_NO_INSTANCENAME_AVAIL | No name is associated with the specified HIN. |
| ODI_NBI_UNSUPPORTED_OPERATION | This function is not available. |

Remarks

The HIN name may be used in displaying hardware instance information to a user.

The HIN name is returned in uppercase and is not translatable.

See Also

CMSMGetHINNameFromHIN

CMSMGetInstanceNumber

Retrieves the instance number of the specified device or function on the specified bus.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <odi_nbi.h>

ODI_NBI CMSMGetInstanceNumber (
    VOID      *busTag
    UINT32    uniqueIdentifier,
    UINT16    *instanceNumber );
```

Input Parameters

busTag

Architecture-dependent value, returned by **CMSMSearchAdapter**, that identifies a specific bus.

uniqueIdentifier

Architecture-dependent value returned by **CMSMGetUniqueIdentifier**, or **CMSMSearchAdapter** that uniquely identifies a specified device function.

Output Parameters

instanceNumber

Address to return the instance number of the device or function. Instance numbers are unique across all buses on the system.

Return Values

| | |
|-------------------------|---|
| ODI_NBI_SUCCESSFUL | The operation was completed successfully. |
| ODI_NBI_PARAMETER_ERROR | An input parameter was invalid. |

Remarks

There is a one-to-one correspondence between the bus tag and the unique identifier pairs and the instance number. You can think of an instance number as a logical slot number. If an adapter contains just one function, the instance number is equivalent to the adapter's physical slot number. 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.

CMSMGetInstanceNumberMapping

Retrieves the bus tag and unique identifier associated with the specified instance number.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <odi_nbi.h>

ODI_NBI CMSMGetInstanceNumberMapping (
    UINT16 instanceNumber,
    VOID    **busTag
    UINT32 *uniqueIdentifier );
```

Input Parameters

instanceNumber
The instance number of the device or function.

Output Parameters

busTag
Address to put the instance’s bus tag

uniqueIdentifier
Address to put the unique identifier.

Return Values

- ODI_NBI_SUCCESSFUL The operation was completed successfully.
- ODI_NBI_PARAMETER_ERROR An input parameter was invalid.

Remarks

CMSMGetInstanceNumberMapping is the inverse of **CMSMGetInstanceNumber**. It retrieves the bus tag and unique identifier associated with the specified instance number.

There is a one-to-one correspondence between bus tag and unique identifier pairs and instance number. You can think of an instance number as a logical slot number. If an adapter contains just one function, the instance number is equivalent to the adapter's physical slot number. 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.

CMSMGetMicroTimer

Returns a counter that is incremented once per microsecond.

Syntax

```
#include <odi.h>
#include <cmsm.h>

UINT32 CMSMGetMicroTimer (void);
```

Input Parameters

None.

Output Parameters

None.

Return Values

A 32-bit, one-microsecond clock value.

Remarks

CMSMGetMicroTimer reads a time counter and returns the value. Elapsed time can be calculated by executing this function twice and subtracting the first returned value from the second.

CMSMGetPhysical

Converts a logical address to a physical one.

Syntax

```
#include <cmsm.h>

void *CMSMGetPhysical (
    void *logicalAddr );
```

Input Parameters

logicalAddr

Pointer to the logical address to be converted into a physical address.

Output Parameters

None.

Return Values

Returns a physical address.

Remarks

If the *MM_FRAGS_PHYS_BIT* bit of the *MLIDCFG_ModeFlags* field is set, this call is needed only at **DriverInit** time to pass the control information in memory to the adapter. This is because ECB fragment pointers are set to physical addresses.

Note



No buffer length is associated with the addresses.

CMSMGetPhysList

Obtains the physical address list equivalent of the input LogicalAddress list.

Syntax

```
#include <odi.h>
#include <cmsm.h>

ODISTAT CMSMGetPhysList(
    UINT32 inputFragCount,
    FRAGMENT_STRUCT* inputFragList,
    UINT32* outputFragCount,
    FRAGMENT_STRUCT* outputFragList,
    DRIVER_DATA* driverData );
```

Input Parameters

inputFragCount

The number of fragments in *inputFragList*.

inputFragList

Pointer to the input fragment list.

driverData

Pointer to the CHSM's driver adapter data space.

Output Parameters

outputFragCount

The number of fragments in *outputFragList*.

outputFragList

Pointer to the output fragment list.

Return Values

| | |
|--------------------|--|
| ODISTAT_SUCCESSFUL | Output list was successfully generated. |
| ODISTAT_FAIL | Output list failed because the maximum number of fragments was exceeded. |

Remarks

This function assumes that the pages containing the logical addresses are locked in memory.

This function generates an output physical address fragment list equivalent to the input logical address fragment list. However, a one-to-one correspondence between the input list and the output list is not guaranteed due to potentially non-contiguous logical memory. As a result, the number and the size of the fragments in the output list may differ from those in the input list.

The input list must not be greater than 16 fragments, but the output buffer must be large enough to accommodate 16 fragments.

CMSMGetPollSupportLevel

Allows a polled driver/adapter to ascertain the level polling supported by the operating system.

Syntax

```
#include <odi.h>
#include <cmsm.h>

UINT32 GetPollSupportLevel (void);
```

Input Parameters

None.

Output Parameters

None.

Return Values

- 0 The environment does not support polling. Polling procedures will never be called. The adapter should use interrupts only.
- 1 Limited support for polling. Polling procedures will be called infrequently. The adapter should use interrupts.
- 2 Polling is fully supported. However, interrupt backup is still recommended due to periods where polling is infrequent.
- 3 Polling is fully supported. No interrupt backup is required.

Remarks

The CHSM uses **GetPollSupportLevel** to ascertain whether the adapter driver should be purely interrupt driven, purely poll driven, or a mixture of interrupt and polling with preference given to polling.

Call this routine only at process time. This routine runs to completion.

CMSMGetUniqueIdentifier

Returns a value which uniquely identifies the device or function of an adapter for the specified input parameters.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <odi_nbi.h>

ODI_NBI CMSMGetUniqueIdentifier (
    void    *busTag,
    UINT32  *parameters,
    UINT32   parameterCount,
    UINT32   *uniqueIdentifier );
```

Input Parameters

busTag

Architecture-dependent value, returned by **CMSMSearchAdapter**, that identifies a specific bus.

parameters

A bus-architecture-dependent array of UNIT32 parameters that are needed to generate a unique identifier. These parameters specify values such as the slot and the function.

parameterCount

The number of elements in the parameter array being passed in.

Output Parameters

uniqueIdentifier

Architecture-dependent value that uniquely identifies a specific device on an adapter.

Return Values

| | |
|-------------------------------|--|
| ODI_NBI_SUCCESSFUL | The device or function was found and <i>uniqueIdentifier</i> was returned. |
| ODI_NBI_PARAMETER_ERROR | An input parameter was invalid. |
| ODI_NBI_UNSUPPORTED_OPERATION | <i>busTag</i> specifies a bus type that has no configuration information. |

Remarks

This function allows physical parameters to be used in identifying adapters placed in physical slots. It also allows the functions on the adapter to be converted to system architecture-dependent values required in operating the adapter. Unique identifiers are interpreted only by other NBI functions. The caller views each as a magic cookie with no predefined format. **CMSMGetUniqueIdentifierParameters** does the inverse of this function.

The parameter values for each bus type are as follows:

ISA Bus

N/A

MCA Bus

| | |
|-----------------------|----------------------|
| <i>parameterCount</i> | 1 |
| <i>parameters[0]</i> | Physical slot number |

EISA Bus

| | |
|-----------------------|----------------------|
| <i>parameterCount</i> | 1 |
| <i>parameters[0]</i> | Physical slot number |

PC Card (PCMCIA) Bus

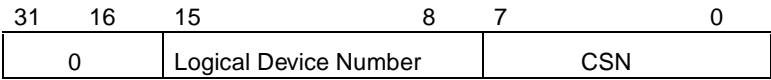
| | |
|-----------------------|---|
| <i>parameterCount</i> | 1 |
| <i>parameters[0]</i> | For single function cards, the physical socket number (1-based). For multiple function cards, the function number (1-based) is in the least significant byte, and the physical socket number is in the next byte. |

PCI Bus

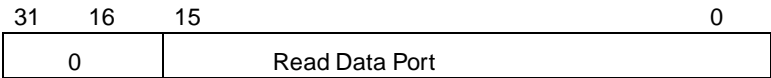
- parameterCount* 2
- parameters [0]* Zero (PCI BIOS version 2.0), physical slot number (PCI BIOS version 2.1)
- parameter[1]* Bus/device/function number combination equivalent to the value returned from the PCI BIOS **FindDevice** function.

PnP ISA Bus (ODI_BUSTYPE_ISA)

- parameterCount* 2
- parameter[0]* Card Select Number (CSN) is in the least significant byte and the Logical Device Number is in the next byte.



- parameter[1]* Read Data Port



CardBus Bus



Note The following CardBus definition applies only to hardware not using the common silicon method defined by the *CardBus PC Card/PCI Common Silicon Requirements* guideline. For CardBus adapters using the common silicon method, refer to the definition for the PCI 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 **FindDevice** function.

CMSMGetUniqueIdentifierParameters

Returns the bus-specific information about the device or the function represented by the given unique identifier.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <odi_nbi.h>

ODI_NBI CMSMGetUniqueIdentifierParameters (
    VOID      *busTag,
    UINT32     uniqueIdentifier,
    UINT32     parameterCount,
    UINT32     *parameters);
```

Input Parameters

busTag

Architecture-dependent value, returned by **CMSMSearchAdapter**, which specifies the bus on which the operation is to be performed.

uniqueIdentifier

Architecture-dependent value; returned by **CMSMGetInstanceNumberMapping**, **CMSMGetUniqueIdentifier**, or **CMSMSearchAdapter**; that uniquely identifies a specific device or function.

parameterCount

The number of elements in the parameter array to be filled in.

Output Parameters

parameters

An array of UINT32 values to be filled in with the bus architecture-dependent parameters represented by the specified unique identifier. (See **CMSMGetUniqueIdentifier** for the format.)

Return Values

| | |
|-------------------------------|---|
| ODI_NBI_SUCCESSFUL | The operation was completed successfully. |
| ODI_NBI_UNSUPPORTED_OPERATION | The bus has no supplementary information to return. |
| ODI_NBI_PARAMETER_ERROR | An input parameter was invalid. |

Remarks

This function is called for a bus which stores bus-specific information.

CMSMHardwareFailure

Called to report a critical or fatal hardware error.

Syntax

```
#include <odi.h>
#include <cmsm.h>

ODISTAT CMSMHardwareFailure (
    DRIVER_DATA    *driverData,
    UINT32          failureType,
    MEON_STRING     *failMsgString );
```

Input Parameters

driverData

Pointer to the C HSM's driver adapter data space.

failureType

NOTIFY_CRITICAL

The CHSM encountered an adapter hardware problem and failed to recover using the available hardware reset capabilities; however, the system may be able to restore the hardware to a functional state.

NOTIFY_FATAL

The CHSM was able to detect a hardware failure, but cannot recover from it.

NOTIFY_DEGRADED

The CHSM has experienced a hardware failure, but is still functional.

failMsgString

Pointer to a NULL terminated string describing the failure. The C MSM will print this string.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL The operation completed successfully.

Remarks

The C HSM calls this routine to report hardware errors.

NOTIFY_FATAL should be reported if the C HSM was able to detect a hardware failure, but cannot recover from it.

NOTIFY_CRITICAL should be reported if the C 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.

CMSMIncrCounter

Increments the counter pointed to by STAT_TABLE_ENTRY by 1.

Syntax

```
#include <odi.h>
#include <cmsm.h>

void CMSMIncrCounter (
    STAT_TABLE_ENTRY *statTableEntryPtr );
```

Input Parameters

statTableEntryPtr

Pointer to the statistics table entry whose counter is to be incremented by 1.

Output Parameters

None.

Return Values

None.

Remarks

This function is intended to simplify the process of incrementing a 64-bit counter, but it can also be used for 32-bit counters.

See Also

CMSMAddToCounter

For more information on STAT_TABLE_ENTRY, see the statistics table information in Chapter 3, "CHSM Data Structures and Variables".

CMSMInitAlloc

Used by CHSMs if they must allocate memory prior to calling **CMSMRegisterHardwareOptions**.

Syntax

```
#include <odi.h>
#include <cmsm.h>

void * CMSMInitAlloc (
    UINT32  nbytes );
```

Input Parameters

nbytes

Number of bytes of memory to allocate.

Output Parameters

None.

Return Values

If successful **CMSMInitAlloc** returns a pointer to the allocated space. Otherwise, it returns a NULL.

Remarks

CHSMs must use the **CMSMInitAlloc** routine if they must allocate memory prior to calling **CMSMRegisterHardwareOptions**. The **CMSMFree** routine releases the buffer any time after **CMSMRegisterHardwareOptions** is called.

If *DriverNeedsBelow16Meg* of the driver parameter block is initialized to 1 (see Chapter 3, "CHSM Data Structures and Variables"), the CMSM attempts to allocate memory below the 16MB boundary.

CMSMInitParser

Initializes the parser.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <parser.h>

ODISTAT CMSMInitParser(
    DRIVER_PARM_BLOCK *hsmParmBlock)
```

Input Parameters

hsmParmBlock
Pointer to the HSMs Parameter Block.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL The operation completed successfully.

Remarks

The C HSM is required to call this routine at the beginning of **DriverInit**, after it has set the pointer to the *chsmStack* in *DriverInitParmPointer* and before it makes any C MSM or C TSM API calls. (See the "Initialization" section in Chapter 5, "CHSM Functions" for more details.)

This function must be called only once for each logical board.

See Also

CMSMParseSingleParameter
CMSMParseDriverParameters

CMSMNESLDeRegisterConsumer

Deregisters a consumer of a specific event.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <nesl_str.h>

UINT32 CMSMNESLDeRegisterConsumer (
    NESL_ECB *consumer )
```

Input Parameters

consumer
Pointer to the NESL_ECB structure passed to **CMSMNESLRegisterConsumer**.

Output Parameters

None.

Return Values

| | |
|---------------------------|---|
| NESL_OK | Deregistration succeeded. |
| NESL_EVENT_NOT_REGISTERED | The specified NESL_ECB structure is not registered. |
| NESL_CONSUMER_NOT_FOUND | The consumer is NULL or can not be located. |

Remarks

Called from foreground with interrupts enabled. See Appendix E for a detailed description of NESL support.

See Also

CMSMNESLRegisterConsumer

7-72 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

CMSMNESLDeRegisterProducer

Deregisters the producer of a specified event. If the producer is the last producer of the specified event, all the remaining consumers of the event are placed onto an orphaned consumer's list.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <nesl_str.h>

UINT32 CMSMNESLDeRegisterProducer (
    NESL_ECB *producer )
```

Input Parameters

producer

Pointer to the NESL_ECB structure passed to **CMSMNESLRegisterProducer**.

Output Parameters

None.

Return Values

| | |
|---------------------------|---|
| NESL_OK | Deregistration was successful. |
| NESL_EVENT_NOT_REGISTERED | The event cannot be located. |
| NESL_PRODUCER_NOT_FOUND | The producer is NULL or could not be located. |

Remarks

Called in the foreground with interrupts enabled. See Appendix E for a detailed description of NESL support.

See Also

CMSMNESLRegisterProducer

CMSMNESLProduceEvent

Called by an event producer to notify registered consumers that the event has occurred. If the event is consumable, one of the consumers can consume the event, and the event notification will stop.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <nesl_str.h>

UINT32 CMSMNESLProduceEvent (
    NESL_ECB *producerNecb,
    NESL_ECB **consumerNecb,
    EPB *eventParmBlock );
```

Input Parameters

producerNecb

Pointer to the NESL_ECB structure passed to **CMSMNESLRegisterProducer**.

eventParmBlock

Pointer to the Event Parameter Block.

Input/Output Parameters

consumerNecb

Pointer to the location of the pointer to the consumer of the event. Null if the producer does not care who the consumer is.

Return Values

| | |
|-------------------------|--|
| NESL_PRODUCER_NOT_FOUND | The producer is NULL. |
| NESL_EVENT_CONSUMED | Event is consumable and is consumed. <i>ConsumerNecb</i> is set to the consumer's NESL_ECB structure if information about the consumer is available. Otherwise, it is set to NULL. |
| NESL_EVENT_NOT_CONSUMED | Event is consumable but is not consumed. <i>ConsumerNecb</i> set to NULL. |
| NESL_EVENT_BROADCAST | Event has been broadcast to all consumers. <i>ConsumerNecb</i> not changed. |

Remarks

Producer routines and consumer routines running on asynchronous events (such as IPX packets or interrupts), must be reentrant.
CMSMNESLProduceEvent will not protect consumer routines from being reentered.

For example, if the consumer routine reenables interrupts, another asynchronous event can be issued from a producer and thus re-enter the consumer.

It is up to either the producer or the consumer routine to protect itself from reentrancy issues. Producer and consumer routines must also ensure that they do not cause stack overflow. See Appendix E for a detailed description of NESL support.

EPB Structure

```
typedef struct EPB_tag
{
    UINT32    EPBMajorVersion;
    UINT32    EPBMinorVersion;
    void      *EPBEventName;
    void      *EPBEventType;
    void      *EPBModuleName;
    void      *EPBDataPtr0;
    void      *EPBDataPtr1;
    UINT32    EPBEventScope;
    UINT32    EPBReserved;
} EPB;
```

Field Descriptions:

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 registered with NESL; for example, 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 is APM Suspend. All valid event subclass names must be registered with Novell Labs.

EPBModuleName

Pointer to the module name that generated the event--for example, NE2000.

EPBDataPtr0

Used to pass a pointer to the configuration table.

EPBDataPtr1

Used for event dependent information.

EPBEventScope

The CHSM must set this field to EPB_SPECIFIC_EVENT.

EPBReserved

Reserved by Novell.

See Also

`CMSMNESLDeRegisterConsumer`

`CMSMNESLDeRegisterProducer`

`CMSMNESLRegisterConsumer`

`CMSMNESLRegisterProducer`

`CMSMNESLProduceMLIDEvent`

CMSMNESSLProduceMLIDEvent

Called by an event producer to notify registered consumers that the event has occurred. If the event is consumable, one of the consumers can consume the event, and the event notification will stop. This call produces the event for each logical board associated with *driverData*.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <nesl_str.h>

UINT32 CMSMNESSLProduceMLIDEvent (
    NESL_ECB      *producerNecb,
    NESL_ECB      **consumerNecb,
    EPB           *eventParmBlock
    DRIVER_DATA   *driverData );
```

Input Parameters

producerNecb

Pointer to the NESL_ECB structure passed to **CMSMNESSLRegisterProducer**.

eventParmBlock

Pointer to the Event Parameter Block.

driverData

Pointer to the CHSM's driver adapter data space.

Input/Output Parameters

consumerNecb

Pointer to the location of where to place the pointer to the consumer of the event. Null if the producer does not care who the consumer is.

Return Values

| | |
|-----------------------------|---|
| NESL_PRODUCER_NOT_FOUND | The producer is NULL. |
| NESL_EVENT_CONSUMED | Event is consumable and is consumed. ConsumerNecb may be set to the consumer's NESL_ECB structure for the consumer of the last logical board that the event was generated for, or it may be set to NULL if the event was consumed, but no information about the consumer was available. |
| NESL_EVENT_NOT_CONSUMED | Event is consumable but is not consumed. ConsumerNecb set to NULL. |
| NESL_EVENT_BROADCAST | Event has been broadcast to all consumers. ConsumerNecb not changed. |
| NESL_INVALID_CONTEXT_HANDLE | The logical board(s) identified by <i>driverData</i> were invalid. |

Remarks

Producer routines and consumer routines running on asynchronous events (such as IPX packets or interrupts), must be reentrant. **CMSMNESLProduceEvent** will not protect the consumer routine from being reentered.

For example, if the consumer routine reenables interrupts, another asynchronous event can be issued from a producer and thus re-enter the consumer.

It is up to either the producer or the consumer routine to protect itself from reentrancy issues. Producer and consumer routines must also ensure that they do not cause stack overflow. See Appendix E for a detailed description of NESL support.

EPB Structure

```
typedef struct EPB_tag
{
    UINT32      EPBMajorVersion;
```

```

UINT32    EPBMinorVersion;
void      *EPBEventName;
void      *EPBEventType;
void      *EPBModuleName;
void      *EPBDataPtr0;
void      *EPBDataPtr1;
UINT32    EPBEventScope;
UINT32    EPBReserved;
} EPB;

```

Field Descriptions:

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 registered with NESL; for example, 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 is APM Suspend. All valid event subclass names must be registered with Novell Labs.

EPBmoduleName

Pointer to the module name that generated the event; for example, NE2000.

EPBDataPtr0

Used to pass a pointer to the configuration table.

EPBDataPtr1

Used for event dependent information.

EPBEventScope

The CHSM must set this field to EPB_SPECIFIC_EVENT.

EPBReserved

Reserved by Novell.

See Also

CMSMNESLDeRegisterConsumer CMSMNESLDeRegisterProducer
CMSMNESLRegisterConsumer
CMSMNESLRegisterProducer
CMSMNESLProduceEvent

CMSMNESLRegisterConsumer

Registers the consumer of an event. If the producer of the event is not currently registered, the consumer is placed onto an orphaned consumer list.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <nesl_str.h>

UINT32 CMSMNESLRegisterConsumer (
    NESL_ECB *consumer )
```

Input Parameters

consumer
Pointer to a NESL_ECB structure.

Output Parameters

None.

Return Values

| | |
|--------------------------|--|
| NESL_OK | Registration was successful. |
| NESL_EVENT_TABLE_FULL | The event was not registered because the event table is full. |
| NESL_DUPLICATED_NECB | The NESL_ECB structure 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.

Remarks

Called at process time. See Appendix E for a detailed description of NESL support.

NESL_ECB Structure

```
typedef struct NECBStruct
{
    struct    NECBStruct    *NecbNext;
    UINT16    NecbVersion;
    UINT16    NecbOsiLayer;
    MEON_STRING    *NecbEventName;
    UINT32    NecbRefData;
    UINT32    (*PnecbNotifyProc)(
        struct NECBStruct *consumerNECB,
        struct NECBStruct *producerNecb,
        void *eventData);
    void    *NecbOwner;
    void    *NecbWorkspace;
    void    *NecbContext;
} NESL_ECB;
```

Field descriptions:

- NecbNext*
Reserved. This field should not be modified by the calling routine while the NESL_ECB structure 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 2.
- NecbOsiLayer*
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 *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. Set this field to `NULL`.

PNecbNotifyProc

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 **CMSMNESLRegisterConsumer**.

ProducerNecb

Points to the `NESL_ECB` structure used by the producer during **CMSMNESLRegisterProducer**.

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 interpreted properly.

For most events this will be a pointer to an Event Parameter Block (EPB). (See Appendix E, "NESL Support" for more information about EPBs.)

Return values 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.

NecbOwner

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.

NecbWorkSpace

Reserved. This field should not be modified by the calling routine while the NESL_ECB structure is registered.

NecbContext

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.

See Also

CMSMDRegisterConsumer

CMSMNESLRegisterProducer

Registers the producer of an event and creates a consumer list containing the consumers of this event.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <nesl_str.h>

UINT32 CMSMNESLRegisterProducer (
    NESL_ECB *Producer )
```

Input Parameters

Producer
Pointer to a NESL_ECB structure.

Output Parameters

None.

Return Values

| | |
|----------------------------|---|
| NESL_OK | Registration 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 structure was previously registered in the event table. |
| NESL_PRODUCER_NOT FOUND | The NESL_ECB structure is NULL. |

Remarks

Called at process time.

The Event definition contains the rules necessary concerning process and interrupt time execution during event notification. See Appendix E for a detailed description of NESL support.

NESL_ECB Structure

```
typedef struct NECBStruct
{
    struct    NECBStruct  *NecbNext;
    UINT16    NecbVersion;
    UINT16    NecbOsiLayer;
    MEON_STRING  *NecbEventName;
    UINT32    NecbRefData;
    UINT32    (*PnecbNotifyProc)(
        struct NECBStruct *consumerNECB,
        struct NECBStruct *producerNecb,
        void *eventData);
    void      *NecbOwner;
    void      *NecbWorkSpace;
    void      *NecbContext;
} NESL_ECB;
```

Field descriptions:*NecbNext*

Reserved. This field should not be modified by the calling routine while the NESL_ECB structure is registered.

NecbVersion

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.

NecbOsiLayer

Reserved. Set this field to NULL.

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 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_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

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.

This flag is used to prohibit multiple producers provided that this is the first producer registered.

PNecbNotifyProc

Reserved. Set this field to NULL.

NecbOwner

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.

NecbWorkSpace

Reserved. This field should not be modified by the calling routine while the NESL_ECB structure is registered.

NecbContext

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.

See Also

CMSMNESLDeRegisterProdcer

CMSMParseDriverParameters

Parses the MLID's parameters.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <parser.h>

ODISTAT CMSMParseDriverParameters (
    DRIVER_PARM_BLOCK *hsmParmBlock,
    struct _DRIVER_OPTION_ *driverOption )
```

Input Parameters

- hsmParmBlock*
Pointer to the driver parameter block structure. See Chapter 3, "CHSM Data Structures and Variables" for a description of DRIVER_PARM_BLOCK.
- driverOption*
Pointer to a linked list of DRIVER_OPTION structures, where each structure describes one of the MLID's possible parameters.

Output Parameters

None.

Return Values

| | |
|--------------------------|---|
| ODISTAT_SUCCESSFUL | All required parameters were found. |
| ODISTAT_ITEM_NOT_PRESENT | One optional parameter was parsed for and that parameter cannot be found. |
| ODISTAT_FAIL | A valid value for a required parameter cannot be found or cannot be obtained, or the user canceled on the prompting of a parameter. |

Remarks

An instance of the `DRIVER_OPTION` structure is used to describe each parameter to be parsed. (Refer to Chapter 4, "CMSM/CTSM Structures and Variables" for details on the `DRIVER_OPTION` structure.)

`DRIVER_OPTION` structures are linked together by the *Link* field of the structure. If the parameter to be parsed is a standard parameter (such as *INT*, *PORT*, *DMA*, *MEM*, *SLOT*, *NODE*, *CHANNEL*, *FRAME*, *NAME*, *RETRIES*, *BELOW16*, *BUFFERS16*); then, *Type*, *Flags*, *Parameter0*, *Parameter1*, and *Parameter2* are the only fields that need to be set. All other fields are ignored.

If the parameter to be parsed is a custom parameter, all fields must be set. The *Flag* field is used to determine the interpretation of *Parameter0*, *Parameter1*, and *Parameter2*; these along with the format specifier in *ParseString* control the parsing of the parameter.

Note



CMSMParseDriverParameters must be called once and only once for each logical board and cannot be called before **<CTSM>RegisterHSM**.

Command Line Parameter Types

CMSMParseDriverParameters can parse for two different types of command line parameters:

- Custom Parameters
- Standard Parameters

Custom parameters are any special parameters that your particular driver needs.

Standard parameters are a set of predefined parameters with established purposes. The parser will populate the `MLIDConfiguration` table with values parsed for standard parameters.

The standard parameters and their required input information are as follows:

| <u>Parameter</u> | <u>Input Information</u> |
|------------------|---|
| INT | The first IRQ used by the adapter. |
| INT1 | The second IRQ used by the adapter. |
| PORT | The first I/O base address used by the adapter. |
| PORT1 | The second I/O base address used by the adapter. |
| DMA | The first DMA channel number used by the adapter. |
| DMA1 | The second DMA channel number used by the adapter. |
| MEM | The first memory base address used by the adapter. |
| MEM1 | The second memory base address used by the adapter. |
| SLOT | The system-wide unique Hardware Instance Number (HIN). It may be the physical slot number on a slot based bus such as Micro Channel, PCI, PC Card, and EISA; or it may be another uniquely assigned number. |
| NODE | The media specific address that the adapter is to use. |
| CHANNEL | The logical channel number that this logical adapter is to use. For a multiport adapter, the channel number usually is a port number. For an adapter on a connection oriented media, the channel number can be used as a connection ID. |
| FRAME | The name of the frame format that this logical adapter is to use. Token-Ring drivers can 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 noncanonical addresses to be passed (this is the default for Token-Ring media). Ethernet media cannot use the MSB designator. |
| NAME | A logical name that can be used with the BIND command to refer to this logical adapter. |
| RETRIES | This is the number of send retries that the MLID should use in its attempts to send packets. |

BELOW16 This keyword must be specified on the load command line if the driver needs memory allocated below the 16MB boundary. This keyword is required only if the MLID is loaded on a system that initially has less than 16 MB of memory, but will have more memory added later. In addition, the driver must also set the *DriverNeedsBelow16Meg* field of the DRIVER_PARM structure to a nonzero value.

BELOW16 is a KEYWORDPARM type field.

BUFFERS16 This keyword is used to override the number of RCBs below 16MB allocated by the CMSM at initialization. The CHSM must set the *DriverNeedsBelow16M* field in the DRIVER_PARM structure for this keyword to be valid. The RCB allocation routines (**CMSMAllocRCB**, **<CTSM>GetRCB**, **<CTSM>ProcessGetRCB**, etc.) use these RCBs if the RCB allocated by the LSL is physically over 16MB. The number of RCBs allocated by default is eight. If the CHSM preallocates more than eight RCBs at a time, the user can override this default when loading the driver by typing BUFFERS16=n. The CMSM will force this value to a multiple of eight, so values other than 8, 16, 32, ... are invalid. No restriction is placed on the maximum value, except that the CMSM might not be able to allocate enough memory from the operating system.

Important



Do not parse for *NODE*, *FRAME*, *NAME*, *RETRIES*, *BELOW16*, or *BUFFERS16*; the CMSM/CTSM parses for these parameters.

CMSMParseSingleParameter

Parses for a single parameter specified by *driverOption* and returns the value in the *driverOption*.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <parser.h>

ODISTAT CMSMParseSingleParameter(
    struct _DRIVER_OPTION_ *driverOption);
```

Input Parameters

driverOption

Pointer to a DRIVER_OPTION structure that describes the parameter to parse for.

Output Parameters


None.

Return Values

| | |
|--------------------------|--|
| ODISTAT_SUCCESSFUL | The parameter was successfully parsed. |
| ODISTAT_FAIL | The parameter cannot be found or cannot be obtained, or the user canceled on the prompting of a parameter. |
| ODISTAT_ITEM_NOT_PRESENT | The parameter was not present. |
| ODISTAT_BAD_PARAMETER | The parameter was found, but was not within the range or value. |

Remarks

A DRIVER_OPTION structure is used to describe the parameter to be parsed. Refer to Chapter 4, "CMSM/CTSM Structures and Variables" for a complete description of the DRIVER_OPTION structure.

Note  The DRIVER_OPTION type field is ignored by CMSMParseSingleParameter. The results are returned only in the DRIVER_OPTION structure.

See Also

CMSMInitParser
CMSMParseDriverParameters

CMSMPrintString

Prints the message pointed to by the *msg* parameter.

Syntax

```
#include <odi.h>
#include <cmsm.h>

void CMSMPrintString (
    const MLID_CONFIG_TABLE *configTable,
    MSG_TYPE msgType;
    MEON_STRING *message,
    void *parm1,
    void *parm2 );
```

Input Parameters

configTable

Pointer to the configuration table.

msgType

The type of message pointed to by *msg*.

message

Pointer to the MEON_STRING to be printed.

parm1

First optional parameter.

parm2

Second optional parameter.

Output Parameters

None.

Return Values

None.

Remarks

The CHSM's initialization routine must call **<CTSM>RegisterHSM** prior to using this print function.

The *parm1* and *parm2* parameters are used here in the same way they are used in the C language *printf* routine. If there are no format specifications in the string, *parm1* and *parm2* are not used.

MSG_TYPE Enumeration

```
typedef enum _MSG_TYPE_
{
    MSG_TYPE_INIT_INFO,
    MSG_TYPE_INIT_WARNING,
    MSG_TYPE_INIT_ERROR,
    MSG_TYPE_RUNTIME_INFO,
    MSG_TYPE_RUNTIME_WARNING,
    MSG_TYPE_RUNTIME_ERROR
} MSG_TYPE;
```

Example

```
startTime = driverData->TxStartTime[driverData->TxNextToReturn];
if ( startTime ) {
    if ( (CMSMGetCurrentTime() - startTime) > 36 ) {
        /* Transmit Timeout */
        /* Send Alert if driverData->BNCFflag != 0 */
        CMSMPrintString(configTable,
                        MSG_TYPE_RUNTIME_WARNING,
                        MSG("066: The cable might be disconnected on the
board.\n\r", 42),0 ,0);
```

Note



The *MSG* function is used in language enabling. For more information on language enabling, see *Appendix A: Language Enabling*.

See Also

See *ODI Specification Supplement: Standard MLID Message Definitions* for a listing of standard messages used by Novell.

CMSMRdConfigSpacex

Takes a bus identifier and an offset from the bus's configuration space and performs the necessary operations to acquire and return the requested data.

Syntax

```
#include <odi.h>
#include <cmsm.h>

UINT8 CMSMRdConfigSpace8 (
    void *busTag,
    UINT32 uniqueIdentifier,
    UINT32 offset );

UINT16 CMSMRdConfigSpace16 (
    void *busTag,
    UINT32 uniqueIdentifier,
    UINT32 offset );

UINT32 CMSMRdConfigSpace32 (
    void *busTag,
    UINT32 uniqueIdentifier,
    UINT32 offset );
```

Input Parameters

busTag

Architecture-dependent value, returned by **CMSMSearchAdapter**, that identifies a specific bus.

uniqueIdentifier

The unique identifier for the specified adapter or function, as returned by **CMSMGetInstanceNumberMapping**, **CMSMGetUniqueIdentifier**, or **CMSMSearchAdapter**.

offset

The byte offset in the configuration space of the specified adapter or function of the item to be read.

Output Parameters

None.

Return Values

An unsigned value of the appropriate size.

Remarks

This function is provided only for MLIDs that need to interact with the configuration space. On most buses, **CMSMGetCardConfigInfo** will meet the MLIDs needs.

For most buses, these calls will do nothing. These calls only have meaning on buses that have a configuration address space that is separated from memory or I/O space--for example, a PCI bus.

See Also

CMSMGetCardConfigInfo, **CMSMWrtConfigSpace**

CMSMReadPhysicalMemory

Copies a block of memory that the MLID might not have the right to access into a buffer that the MLID can access.

Syntax

```
#include <odi.h>
#include <cmsm.h>

void CMSMReadPhysicalMemory (
    UINT32 nbytes,
    void *destAddr,
    void *srcBusTag,
    const void *physSrcAddr );
```

Input Parameters

- nbytes*
The number of bytes to read.
- srcBusTag*
Architecture-dependent value, returned by **CMSMSearchAdapter**, that identifies a specific bus.
- physSrcAddr*
Physical source address (where to read data from).

Output Parameters

- destAddr*
Logical destination address (where to transfer data to).

Return Values

None.

Remarks

CMSMReadPhysicalMemory is called during **DriverInit** before **CMSMRegisterHardwareOptions**. If the MLID attempts to access shared RAM before calling **CMSMRegisterHardwareOptions**, a page fault exception can occur.

Access to the shared RAM prior to registration does not normally happen unless the CHSM must obtain additional information, such as interrupt numbers or the shared RAM buffer size for the configuration table.

The CHSM can use this routine to read information from a shared RAM physical address before hardware registration.

See Also

CMSMWritePhysicalMemory

CMSMRegisterHardwareOptions

Used to register hardware resources with the platform.

Syntax

```
#include <odi.h>
#include <cmsm.h>

REG_TYPE CMSMRegisterHardwareOptions (
    MLID_CONFIG_TABLE *configTable,
    DRIVER_DATA **driverData );
```

Input Parameters

configTable

Pointer to the configuration table of the MLID being registered.

Output Parameters

driverData

Pointer to the CHSM's driver adapter data space.

Return Values

| | |
|----------------------|--|
| REG_TYPE_NEW_ADAPTER | A new adapter was registered and the CHSM should continue 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 <i>driverData</i> pointing to that adapter data space. |
| REG_TYPE_NEW_FRAME | A new frame type was registered for an existing adapter and the DriverInit routine is basically finished. |
| REG_TYPE_NEW_CHANNEL | A new channel was registered for an existing multichannel adapter. The CMSM typically treats the registering of a new channel as a new adapter. The CHSM proceeds with hardware initialization (The <i>driverData</i> parameter contains a pointer to the CHSM's driver adapter data space). |
| REG_TYPE_FAIL | The CMSM was unable to register the hardware options (typically due to conflicts with existing hardware). DriverInit should immediately return a nonzero value to the operating system. |

Remark

- The CHSM's **DriverInit** routine must call this function to register the hardware options.
- When this function returns, the MLIDCFG_MaxFrameSize field in the configuration table may have been down sized if the CHSM set it to a value larger than the maximum size supported by the topology.
- The MLIDCFG_DBusTag field in the MLID Configuration table must be set before making this call.

CMSMRegisterMLID

Registers the MLID with the LSL.

Syntax

```
#include <odi.h>
#include <cmsm.h>

ODISTAT CMSMRegisterMLID (
    const DRIVER_DATA *driverData,
    MLID_CONFIG_TABLE *configTable );
```

Input Parameters

- driverData*
Pointer to the CHSM’s driver adapter data space for the MLID being registered.
- configTable*
Pointer to the configuration table of the MLID being registered.

Output Parameters

None.

Return Values

| | |
|--------------------------|--|
| ODISTAT_SUCCESSFUL | The requested operation was completed successfully. |
| ODISTAT_OUT_OF_RESOURCES | The requested operation could not be completed due to depletion of some system resource. |

Remarks

During **DriverInit** and after successfully initializing the adapter, **DriverInit** should call this routine to register the MLID with the LSL.



Note

When this routine returns, the configuration table contains a valid board number. CHSMs for intelligent bus master adapters can now pass the board number and frame ID information to the adapter if necessary. The *MLIDCFG_MaxFrameSize* field in the configuration table may have been down sized if the CHSM sets the value larger than the maximum size supported by the topology.

Example

The following is within **DriverInit**.

```
/* Allow CMSM to register the MLID with the LSL */
if ( (CMSMRegisterMLID(driverData, configTable) ) !=
      ODISTAT_SUCCESSFUL )
{
    CMSMReturnDriverResources(configTable);
    return-1;
}
```

CMSMRegisterResource

Registers a resource such as memory, interrupts, DMA, and I/O ports with the underlying operating system.

Syntax

```
#include <odi.h>
#include <cmsm.h>

ODISTAT CMSMRegisterResource (
    const DRIVER_DATA *driverData,
    MLID_CONFIG_TABLE *configTable,
    EXTRA_CONFIG *extraConfig );
```

Input Parameters

driverData

Pointer to the CHSM's driver adapter data space for the MLID that owns the resource being registered.

configTable

Pointer to the configuration table of the MLID that owns the resource being registered.

extraConfig

Pointer to the information needed to register the resource(s).

Output Parameters

None.

Return Values

| | |
|--------------------------|------------------------------------|
| ODISTAT_SUCCESSFUL | Resource registered. |
| ODISTAT_OUT_OF_RESOURCES | Resource in use by another device. |

Remarks

This routine allows a CHSM to register a hardware resource that is not listed in the configuration table because that resource in the configuration table is full.

This routine cannot be called until after **CMSMRegisterHardwareOptions** has been called and has returned with *REG_TYPE_NEW_ADAPTER* or *REG_TYPE_NEW_CHANNEL*. This routine must be called before **CMSMSetHardwareInterrupt** if it is being used to add interrupts. The *extraConfig* parameter must always remain allocated, so the CMSM will be responsible for returning the resource if the CHSM gets unloaded.

This routine may only be called at process time.

EXTRA_CONFIG Structure

```
typedef struct _EXTRA_CONFIG_
{
    struct _EXTRA_CONFIG_ *NextLink;
    UINT32 (*ISRRoutine0)(void *MagicNumber);
    void*ISR0Reserved0;
    void*ISR0Reserved1;
    void*ISR0Reserved2;
    void*ISR0Reserved3;
    UINT32 (*ISRRoutine1)(void *MagicNumber);
    void*ISR1Reserved0;
    void*ISR1Reserved1;
    void*ISR1Reserved2;
    void*ISR1Reserved3;
    IO_CONFIG IOConfig;
} EXTRA_CONFIG;
```

Field Descriptions:*ISRRoutine0*

Pointer to the interrupt handler for the specified IRQ. This field must be filled in if the *IO_Interrupt0* or *IO_Interrupt1* field is specified.

ISR0Reserved0, ISR0Reserved1, ISR0Reserved2, ISR0Reserved3

These fields are reserved for use by the CMSM and must not be modified by the CHSM.

ISRRoutine1

This field must be filled in if the *IO_Interrupt0* or *IO_Interrupt1* field is specified. This field is a pointer to the interrupt handler for the specified IRQ.

ISR1Reserved0, ISR1Reserved1, ISR1Reserved2, ISR1Reserved3

These fields are reserved for use by the CMSM and must not be modified by the CHSM.

IOConfig

This field is an *IO_CONFIG* structure filled in by the caller.

IO_CONFIG Structure

The IO_CONFIG structure is defined in *odi.h*. The fields of this structure correspond to the fields in the lower portion of the configuration table and must be set accordingly. For a description of these fields, see the MLID_CONFIG_TABLE structure field descriptions in Chapter 3, "CHSM Data Structures and Variables".

```
typedef struct _IO_CONFIG_
{
    struct _IO_CONFIG_ *IO_DriverLink;
    UINT16              IO_SharingFlags;
    UINT16              IO_Slot;
    UINT16              IO_IOPort0;
    UINT16              IO_IORange0;
    UINT16              IO_IOPort1;
    UINT16              IO_IORange1;
    void                *IO_MemoryAddress0;
    UINT16              IO_MemorySize0;
    void                *IO_MemoryAddress1;
    UINT16              IO_MemorySize1;
    UINT8               IO_Interrupt0;
    UINT8               IO_Interrupt1;
    UINT8               IO_DMALine0;
    UINT8               IO_DMALine1;
    void                *IO_ResourceTag;
    void                *IO_Config;
    void                *IO_CommandString;
    MEON_STRING         IO_LogicalName [18];
    void                *IO_LinearMemory0;
    void                *IO_LinearMemory1;
    UINT16              IO_ChannelNumber;
    void                *IO_DBusTag;
    UINT8               IO_DIOConfigMajorVer;
    UINT8               IO_DIOConfigMinorVer;
} IO_CONFIG;
```

CMSMReRegisterHardwareOptions

Allows a C HSM to deregister its current hardware options and register a new set of hardware options.

Syntax

```
#include <odi.h>
#include <cmsm.h>

ODISTAT CMSMReRegisterHardwareOptions (
    DRIVER_DATA      *driverData ,
    IO_CONFIG         *newIOConfig,
    ECB               *pAsyncECB ) ;
```

Input Parameters

- driverData*
Pointer to the CHSM’s driver adapter data space.
- newIOConfig*
Pointer to an IO_CONFIG structure that contains the hardware options to be registered. The fields of an IO_CONFIG structure correspond to the fields in the MLID_CONFIG_TABLE structure starting with the MLIDCFG_DriverLink and ending with the MLIDCFG_DIOConfigMinorVer. This pointer cannot be NULL.
- pAsyncECB*
Pointer to an ECB whose ESR ((*ECB_ESR)(ECB*)) is called if **CMSMReRegisterHardwareOptions** returns ODISTAT_RESPONSE_DELAYED. The ESR is called with **pAsyncECB** as a parameter and the **pAsyncECB** ECB_Status field will contain the return value. Other ECB fields may be used by the C HSM to store context or other information that is needed by the ESR.

Output Parameters

None.

Return Values

| | |
|--------------------------|---|
| ODISTAT_SUCCESSFUL | The old hardware options were deregistered and the new hardware options were successfully registered. |
| ODISTAT_BAD_PARAMETER | An input parameter was invalid or the call was made at interrupt time and the pAsyncECB parameter was a NULL. |
| ODISTAT_FAIL | The adapter was not in a shutdown state before the call was made, another C MSM API returned ODISTAT_RESPONSE_DELAYED and has not completed when this routine was called, or an unknown error occurred. |
| ODISTAT_ITEM_NOT_PRESENT | The hardware options to be deregistered have not previously been registered. |
| ODISTAT_OUT_OF_RESOURCES | The new hardware options could not be registered. This is typically due to conflicts with resources held by other hardware devices. |
| ODISTAT_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. |

Remarks



Note

After **CMSMRegisterMLID** has been called, but before **CMSMReRegisterHardwareOptions** is called, the adapter must be placed in a shutdown state by calling **CMSMShutdownMLID**.

CMSMReRegisterHardwareOptions will deregister the current set of hardware options held by the CHSM for an adapter as registered through **CMSMRegisterHardwareOptions** or through a previous call to **CMSMReRegisterHardwareOptions**. All hardware options in the newIOConfig parameter will then be registered for the adapter. Any hardware options in the newIOConfig parameter that are not to be registered must be set

as not in use as described in the Driver Configuration Table Field Descriptions section in Chapter 3.

If all hardware options in the newIOConfig parameter were successfully registered, **CMSMReRegisterHardwareOptions** will update all configuration tables of the adapter to reflect the newly registered hardware options.

If **CMSMReRegisterHardwareOptions** 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 pAsyncECB's ESR routine will be called to report the final return value of the operation. The return value will be stored in the pAsyncECB's ECB_Status field.

Upon successful return from **CMSMReRegisterHardwareOptions** or from the asynchronous process, the CHSM is responsible for putting the adapter in a functional state. If an interrupt was registered, the CHSM must call **CMSMSetHardwareInterrupt**.

CMSMReRegisterHardwareOptions upon successful completion will produce a NESL Service/Status Change event to inform consumers that the configuration of the adapter has been updated.

This function updates the following fields in the HSM's configuration table(s).

- MLIDCFG_SharingFlags
(with the exception of the MS_SHUTDOWN_BIT)
- MLIDCFG_Slot
- MLIDCFG_IOPort0
- MLIDCFG_IORange0
- MLIDCFG_IOPort1
- MLIDCFG_IORange1
- MLIDCFG_MemoryAddress0
- MLIDCFG_MemorySize0
- MLIDCFG_MemoryAddress1
- MLIDCFG_MemorySize1
- MLIDCFG_Interrupt0
- MLIDCFG_Interrupt1
- MLIDCFG_DMALine0
- MLIDCFG_DMALine1
- MLIDCFG_LinearMemory0
- MLIDCFG_LinearMemory1
- MLIDCFG_ChannelNumber

7-114 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

See Also

Driver Configuration Table Field Descriptions
<CTSM>RegisterHSM
CMSMSetHardwareInterrupt
CMSMShutDownMLID

CMSMResetMLID

Called by the CHSM to reset the MLID.

Syntax

```
#include <odi.h>
#include <cmsm.h>

ODISTAT CMSMResetMLID (
    DRIVER_DATA *driverData);
```

Input Parameters

driverData
Pointer to the C HSM’s driver adapter data space.

Output Parameters

None.

Return Values

| | |
|--------------------------|---|
| ODISTAT_SUCCESSFUL | Reset was successful. |
| ODISTAT_BAD_PARAMETER | An input parameter was invalid or NULL. |
| ODISTAT_FAIL | The operation failed. The C HSM should place itself in a safe state and clean up resources. |
| ODISTAT_RESPONSE_DELAYED | The operation could not be completed in a timely manner and has been scheduled to complete later. This is a result of the C HSM returning ODISTAT_RESPONSE_DELAYED when the C MSM called DriverReset . |

Remarks

If **DriverReset** needs to be called from within the C HSM it is done by this function. The C MSM puts the driver in a safe state and then calls **DriverReset**.

If the operation is successful the SHUTDOWN flag in **CMSMStatusFlags** is cleared by the C MSM. The C MSM also produces a NESL Resume event.

Polling is not re-enabled by this call if it is in a suspend state.

Note



CMSMResetMLID cannot be called until after **CMSMRegisterMLID** has been called.

See Also

`DriverReset`
`CMSMShutdownMLID`
`CMSMResumePolling`

CMSMResumePolling

Called to re-enable polling after it has been suspended.

Syntax

```
#include <odi.h>
#include <cmsm.h>

ODISTAT CMSMResumePolling (
    DRIVER_DATA *driverData );
```

Input Parameters

driverData
Pointer to the C HSM’s driver adapter data space.

Output Parameters

None.

Return ValuesReturn Values

| | |
|---------------------|--|
| ODISTAT_SUCCESSFUL | Polling was successfully enabled. |
| ODISTAT_BAD_COMMAND | There was no polling procedure registered for this MLID. |

Remarks

Turns polling back on after CMSMSuspendPolling has suspended it. This call is only useful if CMSMSuspendPolling was called previously. When CMSMEnablePolling is called polling will start up active.

The POLLING_SUSPENDED flag in CMSMStatusFlags is cleared by the C MSM when CMSMResumePolling is called.

See Also

CMSMSuspendPolling
CMSMEnablePolling

CMSMReturnDriverResources

Returns the MLID’s resources before exiting.

Syntax

```
#include <odi.h>
#include <cmsm.h>

void CMSMReturnDriverResources (
    const MLID_CONFIG_TABLE *configTable );
```

Input Parameters

configTable
Pointer to the configuration table.

Output Parameters

None.

Return Values

None.

Remarks

If the MLID fails during **DriverInit**, this routine should be called before exiting **DriverInit** to return the resources. If the CHSM detects an error itself, opposed to having an error reported to it, the CHSM can print an error message using **CMSMPrintString** before calling this function.

Example

```
/* Set up our interrupt procedure*/  
if ( CMSMSetHardwareInterrupt(driverData, configTable) !=  
    ODISTAT_SUCCESSFUL)  
{  
    CMSMReturnDriverResources(configTable);  
    return (-1);  
}
```

CMSMReturnMultipleRCBs

Returns a linked list of RCBs. This routine is called to discard RCBs, not process them.

Syntax

```
#include <cmsm.h>

void CMSMReturnMultipleRCBs (
    RCB  *rcbp);
```

Input Parameters

rcbp

Pointer to the first RCB on the linked list to be returned.

Output Parameters

None.

Return Values

None.

Remarks

CMSMReturnMultipleRCBs is executed at process time or privileged time.

This function must only be used to return ECBs allocated using **<CTSM>ProcessGetRCB**, **CMSMAllocateMultipleRCBs**, **CMSMAllocateRCB**.

Note



Refer to the description of the Receive Control Block (RCBs) in Chapter 4, "CMSM/CTSM Structures and Variables" for details concerning the link fields in the RCB.

See Also

CMSMAllocateRCB
CMSMAllocateMultipleRCBs
CMSMReturnRCB
<CTSM>RcvComplete
<CTSM>ProcessGetRCB

CMSMReturnRCB

Returns an RCB to the LSL. This routine is called to discard the RCB, not to process it.

Syntax

```
#include <cmsm.h>

void CMSMReturnRCB (
    const DRIVER_DATA *driverData,
    RCB *rcbp );
```

Input Parameters

- driverData*
Pointer to the CHSM’s driver adapter data space for the MLID.
- rcbp*
Pointer to the unneeded RCB.

Output Parameters

None.

Return Values

None.

Remarks

- CMSMReturnRCB** is executed at process or privileged time.
- This function must only be used to return RCBs allocated using <CTSM>ProcessGetRCB and CMSMAllocateRCB.

Example

```
/* Return any RCBs that we have queued up */
while(driverData->NeedRCBCount != RX_BUFFERS) {
    rcb = driverData->RCBTable[driverData->RxNextToReturn++];
    driverData->RxNextToReturn &= (RX_BUFFERS - 1);
    driverData->NeedRCBCount++;
    CMSMReturnRCB(driverData, rcb);
}
```

See Also

<CTSM>RcvComplete, <CTSM>ProcessGetRCB

CMSMScanBusInfo

Specifies the buses that are available on the system.

Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <odi_nbi.h>

ODI_NBI CMSMScanBusInfo (
    UINT32    *scanSequence,
    void      **busTag,
    UNIT32    *busType,
    MEON_STRING **busName );
```


Input/Output Parameters

scanSequence
Must be initialized to -1 for the first search. The value returned in this parameter after each call must be passed back into this parameter in the subsequent call to this function.

Output Parameters

busTag
Architecture-dependent value that identifies a specific bus.

busType (Defined in *odi_nbi.h*)
A pointer to a value that specifies the bus type.

Note  The ODI_BUSTYPE_CARDBUS type value is used only for hardware not using the common silicon method defined by the *CardBus PC Card/PCI Common Silicon Requirements* guideline. For CardBus adapters using the common silicon method refer to the ODI_BUSTYPE_PCI type value.

| Type Value | Bus |
|---------------------|---------------------------|
| ODI_BUSTYPE_ISA | ISA / ISA PnP bus |
| ODI_BUSTYPE_MCA | Micro Channel bus |
| ODI_BUSTYPE_EISA | EISA bus |
| ODI_BUSTYPE_PCMCIA | PC Card |
| ODI_BUSTYPE_PCI | PCI bus |
| ODI_BUSTYPE_VESA | VESA local bus |
| ODI_BUSTYPE_NUBUS | NuBus bus |
| ODI_BUSTYPE_OFM | Open Firmware Motherboard |
| ODI_BUSTYPE_CARDBUS | CardBus bus |

busName

A pointer to a static, NULL-terminated, architecture-dependent string that is determined by the platform developer. The caller should not modify this string. To reference this string, make a local copy of it.

Return Values

| | |
|-------------------------|---|
| ODI_NBI_SUCCESSFUL | The requested operation was completed successfully. |
| ODI_NBI_PARAMETER_ERROR | The parameter was invalid. |
| ODI_NBI_NO_MORE_ITEMS | There are no more buses. |

Remarks

CMSMScanBusInfo searches the system for available busses.

CMSMScheduleAES

Called during **DriverInit** to enable a call back to a routine in the CHSM.

Syntax

```
#include <odi.h>
#include <cmsm.h>

ODISTAT CMSMScheduleAES (
    const DRIVER_DATA *driverData,
    MLID_AES_ECB *mlidAESECB );
```

Input Parameters

- driverData*
Pointer to the CHSM’s driver adapter data space of the MLID that is registering an AES callback.
- mlidAESECB*
Pointer to the MLID_AES_ECB structure.

Output Parameters

None.

Return Values

| | |
|--------------------------|---|
| ODISTAT_SUCCESSFUL | Call back was successfully scheduled. |
| ODISTAT_BAD_PARAMETER | Invalid parameter(s) set in MLID_AES_ECB structure). |
| ODISTAT_OUT_OF_RESOURCES | Resources to the successfully completed operation were not available. |

Remarks

When this routine is called, the CHSM passes in an MLID_AES_ECB pointer with the routine to be called and the time interval to wait before calling that routine.

The CHSM should not call this routine until after **CMSMRegisterMLID** has been called.

MLID_AES_ECB Structure

```
typedef struct _MLID_AES_ECB_
{
    struct _MLID_AES_ECB_ *NextLink;
    void (*DriverAES)(DRIVER_DATA
*,MLID_CONFIG_TABLE *);
    AES_TYPE AesType;
    UINT32 TimeInterval;
    void *AesContext;
    UINT8 AesReserved[30];
} MLID_AES_ECB;
```

Field descriptions:


- NextLink*
Used to link MLID_AES_ECB structures together.
- DriverAES*
A pointer to the function that is called after the specified time interval.
More than one instance of this function can be active at a time, but each instance must have a unique name and MLID_AES_ECB structure.
- AesType*
Used to specify the type of event, where *AesType* is one of the following:
- | | |
|--------------------------------|--------------------------------------|
| AES_TYPE_PRIVILEGED_ONE_SHOT | Call only once at privileged time |
| AES_TYPE_PRIVILEGED_CONTINUOUS | Call this routine at privileged time |

| | |
|-----------------------------|-----------------------------------|
| AES_TYPE_PROCESS_ONE_SHOT | Call only once at process time |
| AES_TYPE_PROCESS_CONTINUOUS | Call this routine at process time |

TimeInterval
The time in milliseconds to wait before calling **DriverAES**.

AesContext
Reserved for use by the CMSM.

AesReserved
Reserved for use by the CMSM.

Important  MLID_AES_ECB must remain allocated until the MLID is removed.

CMSMSearchAdapter

Takes the bus type and address of a product ID and returns a *busTag* and a *uniqueIdentifier* for where the specified product (device or function) is found.


Syntax

```
#include <odi.h>
#include <cmsm.h>
#include <odi_nbi.h>

ODI_NBI CMSMSearchAdapter (
    UINT32  *scanSequence,
    UINT32  busType,
    UINT32  productIDLen,
    const MEON *productID,
    void  **busTag,
    UINT32  *uniqueIdentifier );
```

Input Parameters

busType
A bus type as defined in *odi_nbi.h*.

Note  The ODI_BUSTYPE_CARDBUS type value is used only for hardware not using the common silicon method defined by the *CardBus PC Card/PCI Common Silicon Requirements* guideline. For CardBus adapters using the common silicon method refer to the ODI_BUSTYPE_PCI type value.

| Type Value | Bus |
|--------------------|-------------------|
| ODI_BUSTYPE_ISA | ISA / ISA PnP bus |
| ODI_BUSTYPE_MCA | Micro Channel bus |
| ODI_BUSTYPE_EISA | EISA bus |
| ODI_BUSTYPE_PCMCIA | PC CardBus |
| ODI_BUSTYPE_PCI | PCI bus |
| ODI_BUSTYPE_NUBUS | NuBus bus |

ODI_BUSTYPE_OFM Open Firmware motherboard
ODI_BUSTYPE_CARDBUS CardBus bus

productIDLen

Length of the product ID.

productID

A pointer to a bus architecture-dependent parameter that uniquely identifies an adapter board/peripheral/system option. For example, for an EISA bus, this is an EISA product ID as defined in the *EISA Specification* document.

Input/Output Parameters

scanSequence

On the first search for each productID, *scanSequence* must be initialized to -1. The value returned in this parameter after each call must be passed back into this parameter in each subsequent call to this function.

Output Parameters

busTag

Architecture-dependent value that specifies the bus where this function found the first item identified by the product ID.

Note



The ODI_BUSTYPE_CARDBUS type value is used only for hardware not using the common silicon method defined by the *CardBus PC Card/PCI Common Silicon Requirements* guideline. For CardBus adapters using the common silicon method refer to the ODI_BUSTYPE_PCI type value.

uniqueIdentifier

Architecture-dependent value that identifies the specific device or function. This call will return information for each instance of the product ID and compatible products, including multiple instances on a single card (each having a different function number). The Hardware Instance Number (HIN) (used in the *slot=* command line parameter) can be taken from the *busTag* and the *uniqueIdentifier* pair by calling **CMSMGetInstanceNumber**.

Return Values

| | |
|--------------------------|--|
| ODI_NBI_SUCCESSFUL | The requested operation was completed successfully. |
| ODI_NBI_ITEM_NOT_PRESENT | No device or function matches the given bus type and product ID combination. |

Remarks

This function should be used only if the CHSM's adapter has a unique *productID* associated with it that can be read by the NBI. The *productID* must be retrievable according to an accepted standard, such as EISA, PCI, Micro Channel, PnP ISA, or PC Card.

CMSMServiceEvents

Completes the processing of queued and received packets.

Syntax

```
#include <cmsm.h>

void CMSMServiceEvents (void);
```

Input Parameters

None.

Output Parameters

None.

Return Values

None.

Remarks

CMSMServiceEvents is called at process or privileged time. If the CHSM has used **<CTSM>SendComplete**, **<CTSM>RcvComplete**, or **<CTSM>ProcessGetRCB**, it must use **CMSMServiceEvents** before it exits back to the operating system.

CMSMSetHardwareInterrupt

Called by the CHSM's **DriverInit** routine to set up a hardware interrupt handler.

Syntax

```
#include <odi.h>
#include <cmsm.h>

ODISTAT CMSMSetHardwareInterrupt (
    DRIVER_DATA *driverData,
    const MLID_CONFIG_TABLE *configTable );
```

Input Parameters

- driverData*
Pointer to the CHSM's driver adapter data space.
- configTable*
Pointer to the configuration table.

Output Parameters

None.

Return Values

| | |
|--------------------------|---|
| ODISTAT_SUCCESSFUL | The requested operation was completed successfully. |
| ODISTAT_OUT_OF_RESOURCES | The requested operation could not be completed due to depletion of a system resource. |
| ODISTAT_BAD_PARAMETER | The <i>DriverISRPtr</i> field in the driver parameter block was NULL. |

Remarks

CMSMSetHardwareInterrupt examines the configuration table to obtain the number of the interrupt to set. **CMSMSetHardwareInterrupt**, then examines the driver parameter block to obtain the address of **DriverISR**. If the CHSM's MLID configuration table and driver parameter block specify two interrupt service routines, **CMSMSetHardwareInterrupt** will set up both.

Note



Do not call this function unless you are ready to process interrupts. This call may only be called at process time.

Example

```
/* Set up our interrupt procedure */
if ( CMSMSetHardwareInterrupt(driverData, configTable) !=
    ODISTAT_SUCCESSFUL) {
    CMSMReturnDriverResources(configTable);
    return (-1);
}
```

CMSMShutdownMLID

Called by the C HSM to shut the MLID down.

Syntax

```
#include <odi.h>
#include <cmsm.h>

ODISTAT CMSMShutdownMLID (
    DRIVER_DATA    *driverData,
    UINT32          shutdownType );
```

Input Parameters

- driverData*
Pointer to the C HSM’s driver adapter data space.
- shutdownType*
 - SHUTDOWN_PERMANENT
Perform a permanent shutdown.
 - SHUTDOWN_PARTIAL
Perform a partial shutdown.

Output Parameters

None.

Return Values

| | |
|--------------------------|--|
| ODISTAT_SUCCESSFUL | The driver was shutdown successfully. |
| ODISTAT_BAD_PARAMETER | An input parameter was invalid or NULL. |
| ODISTAT_FAIL | The operation failed. |
| ODISTAT_RESPONSE_DELAYED | Under circumstances where shutting down the MLID cannot be completed when CMSMShutdownMLID is called, an asynchronous process should be called at a later time. |

Remarks

If the **DriverShutdown** needs to be called from within the C HSM it is done by this function. The C MSM puts the driver in a safe state and then calls **DriverShutdown**. If a partial shutdown was performed a call to **CMSMResetMLID** will bring the driver out of shutdown state

The SHUTDOWN flag in **CMSMStatusFlags** is set by the C MSM when **CMSMShutdownMLID** is called. The C MSM also produces a NESL Suspend event.

If **CMSMShutdownMLID** returns ODISTAT_RESPONSE_DELAYED, we recommend that you return with the adapter disabled unless it is impossible or inadvisable.

Note



CMSMShutdownMLID cannot be called until after **CMSMRegisterMLID** has been called.

See Also

DriverShutdown
CMSMResetMLID

CMSMSuspendPolling

Suspends the calling of the DriverPoll procedure until **CMSMResumePolling** is called.

Syntax

```
#include <odi.h>
#include <cmsm.h>

ODISTAT CMSMSuspendPolling (
    DRIVER_DATA *driverData );
```

Input Parameters

driverData
Pointer to the C HSM’s driver adapter data space.

Output Parameters

None.

Return Values

| | |
|---------------------|--|
| ODISTAT_SUCCESSFUL | Polling was successfully suspended. |
| ODISTAT_BAD_COMMAND | There was no polling procedure registered for this MLID. |

Remarks

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 **CMSMSuspendPolling** to temporarily stop the driver from being polled when it is known that there is no usable work to do.

The POLLING_SUSPENDED flag in CMSMStatusFlags is set by the C MSM when **CMSMSuspendPolling** is called and cleared by the C MSM when **CMSMResumePolling** is called and can be inspected by the C HSM to determine the current polling status.

Calling **CMSMResetMLID** does not re-enabled polling.

See Also

CMSMResumePolling

CMSMTCBPhysToLogFrag

Gets the address of the ECB whose ECB structure contains the logical addresses of the fragments in the TCB for an adapter when the *MM_FRAGS_PHYS_BIT* bit is set.

Syntax

```
#include <odi.h>
#include <cmsm.h>

ECB *CMSMTCBPhysToLogFrag (
    TCB *tcb );
```

Input Parameters

tcb
Pointer to a TCB structure.

Output Parameters

None.

Return Values

Pointer to the ECB structure containing the logical addresses for the TCB fragments.

Remarks

You cannot assume that the fragment pointers have a one-to-one correspondence. Because the physical pointers point to fragments that are physically contiguous, there can be more fragments in the physical list than in the logical list.

FRAGMENT_LIST_STRUCT Structure

```
typedef struct _FRAGMENT_LIST_STRUCT_  
{  
    UINT32          FragmentCount;  
    FRAGMENT_STRUCT FragmentStruct;  
} FRAGMENT_LIST_STRUCT;
```

CMSMUpdateConfigTables

Allows a C HSM to tell the tool kit to update all copies of the configuration table for an adapter.

Syntax

```
#include <odi.h>
#include <cmsm.h>

ODISTAT CMSMUpdateConfigTables (
    DRIVER_DATA *driverData,
    MLID_CONFIG_TABLE *configTable );
```

Input Parameters

- driverData*
Pointer to the CHSM’s driver adapter data space.
- configTable*
Pointer to a configuration table.

Output Parameters

None.

Return Values

- ODISTAT_SUCCESSFUL All configuration tables for the adapter were updated.
- ODISTAT_BAD_PARAMETER An input parameter was invalid.

Remarks

CMSMUpdateConfigTables copies the following configuration table fields from the configTable parameter to all configuration tables of a adapter. All other configuration table fields are ignored.

- MLIDCFG_NodeAddress
- MLIDCFG_ModeFlags
- MLIDCFG_MaxFrameSize

MLIDCFG_CardName
 MLIDCFG_ShortName
 MLIDCFG_TransportTime
 MLIDCFG_LineSpeed
 MLIDCFG_SGCount
 MLIDCFG_PrioritySup
 MLIDCFG_Flags
 MLIDCFG_SendRetries

MLIDCFG_BestDataSize and MLIDCFG_WorstDataSize are automatically adjusted by the CTSM based on MLIDCFG_MaxFrameSize.

A CHSM can call **CMSMUpdateConfigTables** any time to update an adapter's configuration tables. All fields copied from the configTable parameter must be valid before **CMSMUpdateConfigTables** is called.

During a driver's initialization for an adapter, **CMSMRegisterHardwareOptions** automatically updates the adapter's configuration tables. A call to **CMSMUpdateConfigTables** is only necessary if the fields copied from the configTable parameter are modified after the call to **CMSMRegisterHardwareOptions**.

CMSMUpdateConfigTables 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.

See Also

CMSMRegisterHardwareOptions

CMSMWrtConfigSpacex

Takes a value, a bus identifier, and an offset in the bus's configuration space and performs whatever operations are necessary to deliver the value to the specified location.

Syntax

```
#include <odi.h>
#include <cmsm.h>

void CMSMWrtConfigSpace8 (
    void *busTag,
    UINT32 uniqueIdentifier,
    UINT32 offset,
    UINT8 writeVal );

void CMSMWrtConfigSpace16 (
    void *busTag,
    UINT32 uniqueIdentifier,
    UINT32 offset,
    UINT16 writeVal );

void CMSMWrtConfigSpace32 (
    void *busTag,
    UINT32 uniqueIdentifier,
    UINT32 offset,
    UINT32 writeVal );
```

Input Parameters

busTag

Architecture-dependent value, returned by **CMSMSearchAdapter**, that identifies a specific bus.

uniqueIdentifier

An architecture-dependent value identifying the specific device or function. See **CMSMSearchAdapter** or **CMSMGetUniqueIdentifier**.

offset

The byte offset in the configuration space of the specified device or function where the item is to be written to.

writeVal

The appropriate size value to be written to the specified configuration space address on the specified bus.

Output Parameters

None.

Return Values

None

Remarks

This function is provided only for MLIDs that need to interact with the configuration space.

For most buses, these calls will do nothing. These calls only have meaning on buses that have a configuration address space that is separated from memory or I/O space--for example, a PCI bus.

See Also

CMSMGetCardConfigInfo, **CMSMWrtConfigSpace**

CMSMWritePhysicalMemory

Allows the CHSM to write to memory that is not registered to the CHSM.

Syntax

```
#include <odi.h>
#include <cmsm.h>

void CMSMWritePhysicalMemory (
    UINT32 nbytes,
    void *destBusTag,
    void *physDestAddr,
    const void *srcAddr );
```

Input Parameters

- nbytes*
The number of bytes to write.
- destBusTag*
Architecture-dependent value, returned by **CMSMSearchAdapter**, that identifies a specific bus.
- physDestAddr*
Physical destination buffer (where to transfer data).
- srcAddr*
Logical source buffer (where to read data).

Output Parameters

None.

Return Values

None.

Remarks

CMSMWritePhysicalMemory is called during **DriverInit** before **CMSMRegisterHardwareOptions**. If the CHSM attempts to access shared RAM before calling **CMSMRegisterHardwareOptions**, a page fault ABEND will occur. Accesses to the shared RAM prior to registration do not normally happen unless the CHSM 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

CMSMReadPhysicalMemory

7-150 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

Overview

One of the goals of this specification is to allow you to write CHSMs that are portable across different platforms. Some of these platforms may have one or more buses that are quite different from each other. Therefore, to achieve platform independence, it is necessary to isolate the CHSM from the details of the platform's bus architecture.

This chapter describes bus architecture-dependent functions provided by Novell or the platform developer to perform operations needed by the CHSM.

If you want your CHSM to be portable, you must follow this specification and must not access the Programmable Interrupt Controller (PIC), since this piece of hardware is different on different hardware platforms. Interrupt control operations should be performed using functions defined in this specification. Also, the CHSM must not directly write or read data to or from its own adapter, because this can require different operations on different hardware platforms. Instead, it must use the **Inx**, **Outx**, **Rdx**, and **Wrtx** functions defined here.

Some of the functions defined in this specification are not needed by most CHSMs. If a function is not needed, it should not be used. Defining unneeded functions increases the likelihood that your CHSM will not work on a particular platform, since some of the functions are not supported on some platforms.

Bus Architecture

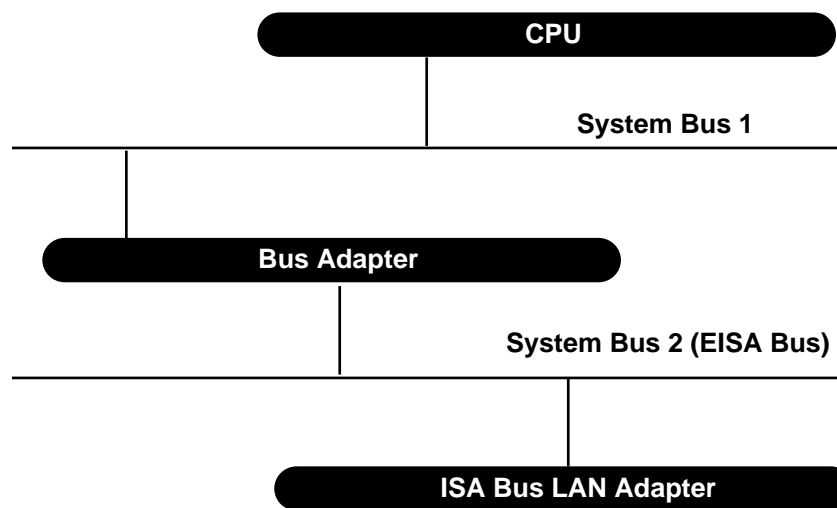
A bus architecture is one or more related address spaces and a set of behaviors (including asynchronous behaviors) of data within those address spaces. For example, an IBM PC ISA address space consists of the following:

- 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.

Multiple Bus Platforms

Figure 8.1 shows an example of a multiple bus platform.

Figure 8-1
Multiple Bus Platform Example



8-2 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

Because of the potentially differing bus architectures on a multiple bus platform and the intervening bus adapter, you cannot assume that an MLID executing on the CPU will always be able to directly access the Programmable Interrupt Controller or the DMA controller the same way it does on an IBM PC. These controllers may be implemented using hardware completely different from the IBM PC. You cannot even assume that the MLID will be able to access the memory addresses it needs to communicate with its adapter. In Figure 8.1 for example, the intervening bus adapter may have the *System Bus 2* memory addresses mapped to some other set of addresses in the *System Bus 1* address space, or they may not be mapped at all.

The functions in this chapter are defined so that the CHSM can be independent of the underlying architecture. If you use these functions correctly, your CHSM should be portable from platform to platform by simply recompiling the code for the new platform. You can write a CHSM specifically for a particular platform, but it will not be portable.

A portable CHSM must not attempt to access anything directly outside of its own data space area. Instead, the CHSM should use the bus architecture-dependent functions in this chapter to access all hardware devices and their associated data. These functions include such things as interrupt enabling/disabling/dismissing and DMA start/cleanup.

Memory Mapping and Address Manipulation

The memory mapping and address manipulation functions in this chapter will make more sense if you have some idea of their intended use and implementation. The following examples describe two different types of LAN adapters (or cards): a shared memory card, and a card that transfers its own data.

In the case of a shared memory card, the CHSM running on the CPU transfers the data. The intention is to let the CMSM (using **MapBusMemory**) map the shared memory card's physical address range on *System Bus 2* to a logical address on *System Bus 1*. This mapping is to remain in effect as long as the driver is loaded. (This may involve allocating and programming hardware in the bus adapter or looking up how the hardware was programmed during system initialization.) The CMSM then converts the *System Bus 1* physical address to a CPU logical address. The CHSM then uses this CPU logical address as the base address of the card's shared memory.

At this point, it might seem like the CHSM should be able to use this address plus the offsets to access the card's shared memory. Unfortunately, this will not

NetWare Bus Interface 8-3

work on all platforms, and this is the reason for the **MovFastFromBus**, **MovFastToBus**, **MovFromBusx**, **MovToBusx**, **Rdx**, **Wrtx**, and **Setx** functions. On most platforms, these functions will be implemented as inline macros.

In the case of a card that transfers its own data, the intention is for the CHSM to convert logical addresses in system memory to physical addresses on *System Bus 1*; then, to map the *System Bus 1* physical addresses to *System Bus 2* physical addresses. The CHSM then passes the *System Bus 2* physical addresses to the card for its use. These mappings are released by the CHSM as soon as the transmission or the reception is complete.

Byte Order

It is the responsibility of the functions in this chapter to do any byte swapping that is necessary to get the data to its destination in the correct order. In other words, if a UINT16 is being read from an ISA adapter in the example in Figure 8.1 and System Bus 1 is a big endian bus, the bytes are to be swapped by the **In16** or **Rd16** routine (unless the bus adapter hardware does it) so that the value returned by the function is correct.

DMACleanup

Cleans up, closes down, and releases the resources associated with a DMA operation.

Syntax

```
#include <odi.h>
#include <odi_nbi.h>

void  DMACleanup (
    UINT32  dmaChannel );
```

Input Parameters

dmaChannel

An architecture-dependent value specified by the platform developer and set by the user in the MLID load command line. It specifies which DMA channel is to be used.

Output Parameters

None.

Return Values

None.

Remarks

It is the responsibility of the NBI to know what resources are associated with this operation. If called while the DMA operation is still in progress, this function must first abort the operation and then do the cleanup.

DMAStart

Moves data from one location (on one bus) to another location (potentially on a different bus) using the specified DMA channel.

Syntax

```
#include <odi.h>
#include <odi_nbi.h>

ODI_NBI DMAStart (
    void      *destBusTag,
    UINT32    destAddrType,
    const void *destAddr,
    void      *srcBusTag,
    UINT32    srcAddrType,
    const void *srcAddr,
    UINT32    len,
    UINT32    dmaChannel,
    UINT32    dmaModel,
    UINT32    dmaMode2 );
```

Input Parameters

destBusTag

Architecture-dependent value, returned by **CMSMSearchAdapter**, that identifies a specific bus.

destAddrType

Boolean parameter. False (0) indicates that the destination address is in memory space; true (nonzero) indicates that the destination address is in I/O space.

destAddr

The physical memory address in the bus architecture of the adapter to which the DMA is to occur.

srcBusTag

Architecture-dependent value, returned by **CMSMSearchAdapter**, that identifies a specific bus.

srcAddrType

Boolean parameter. False (zero) indicates that the source address is in memory space; true (nonzero) indicates that the source address is in I/O space.

srcAddr

The physical memory address in the bus architecture of the adapter from which the DMA is to occur.

len

The length in bytes to be DMA'd.

dmaChannel

An architecture-dependent value specified by the platform developer and set by the user in the MLID load command line. It specifies which DMA channel is to be used. If not set by the user, it should be set to 0.

dmaMode1 and *dmaMode2*

These parameters are bus architecture-dependent. Their meaning is defined on a bus architecture by bus architecture basis. Currently, they are defined only for the ISA and EISA buses. On these boards, this parameter specifies the DMA transfer mode for this DMA channel.

Bits 7 - 6

00 demand mode select

01 single mode select

10 block mode select

11 cascade mode select

Bit 5

0 address increment select

1 address decrement select

Bit 4

0 automatic initialization disable

1 automatic initialization enable

Automatic initialization refers to the DMA device automatically restoring the current address and current count from the base address and base count after the process ends.

Output Parameters

None.

Return Values

| | |
|-------------------------------|--|
| ODI_NBI_SUCCESSFUL | The requested operation was completed successfully. |
| ODI_NBI_PROTECTION_VIOLATION | Memory protection prevented the completion of the requested operation. |
| ODI_NBI_HARDWARE_ERROR | Hardware error or hardware limitation prevented the completion of the requested operation. |
| ODI_NBI_PARAMETER_ERROR | One of the parameters was invalid. |
| ODI_NBI_UNSUPPORTED_OPERATION | The requested operation could not be completed. |

Remarks

This function is used only in situations where the DMA channel is a system-wide resource. It is not used where the adapter itself does the DMA.

The NBI deals with alignment issues, hardware mapping issues, and length issues. Errors are returned only if a hardware failure is detected, the user has requested an operation that is impossible, or the user does not have permission to access the memory indicated.

Note



This function seems to allow some unreasonable operations such as DMAing from I/O space to I/O space; this specification is not intended to support such operations. Also, the DMA channels are assumed to be a system-wide resource and not associated with a particular bus. In fact, any particular DMA controller will be associated with some particular bus, but it is the responsibility of the NBI to manage the platform's DMA controllers and determine (using *srcBusTag* and *destBusTag*) how to perform the requested operation.

DMAStatus

Returns the status of the specified DMA channel.

Syntax

```
#include <odi.h>
#include <odi_nbi.h>

UINT32 DMAStatus (
    UINT32 dmaChannel );
```

Input Parameters

dmaChannel

An architecture-dependent value specified by the platform developer and set by the user in the MLID load command line. It specifies which DMA channel is to be used.

Output Parameters

None.

Return Values

Bit 0 Set if the channel has completed DMA operation.
Bit 1 Set if the channel has a pending DMA bus cycle request.

Remarks

This function can be used by the CHSM to determine the state of the DMA channel. This function can be used to detect the completion of a DMA transfer on that channel or if that channel is currently requesting the I/O memory buses.

FreeBusMemory

Frees any hardware resources allocated by the function **MapBusMemory**.

Syntax

```
#include <odi.h>
#include <odi_nbi.h>

ODI_NBI FreeBusMemory (
    void *busTag1,
    const void *memAddr,
    void *busTag2,
    const void *mappedAddr,
    UINT32 len );
```

Input Parameters

busTag1

Architecture-dependent value, returned by **CMSMSearchAdapter**, that identifies the bus that contains the physical memory space.

memAddr

Address of the physical memory space of *busTag1*.

busTag2

Architecture-dependent value, returned by **CMSMSearchAdapter**, that identifies the bus that is mapped to the physical memory space of *busTag1*.

mappedAddr

Pointer to the address of the physical memory space of *busTag1*; used by *busTag2*.

len

The length of the mapped memory space in bytes.

Output Parameters

None.

Return Values

| | |
|------------------------------|--|
| ODI_NBI_SUCCESSFUL | The requested operation was completed successfully. |
| ODI_NBI_PROTECTION_VIOLATION | Memory protection prevented the completion of the requested operation. |
| ODI_NBI_PARAMETER_ERROR | One of the parameters was invalid. |

Remarks

The parameters passed to this call must be the same parameters that were passed to **MapBusMemory** when the memory was mapped..

A CHSM cannot unmap just a portion of previously mapped memory.

See Also

MapBusMemory

Inx

Does whatever operations are necessary to get and return the requested data, using the bus tag and I/O address space.

Syntax

```
#include <odi.h>
#include <odi_nbi.h>

UINT8  In8 (
    void *busTag,
    const void *ioAddr );

UINT16 In16 (
    void *busTag,
    const void *ioAddr );

UINT32 In32 (
    void *busTag,
    const void *ioAddr );
```

Input Parameters

busTag

Architecture-dependent value, returned by **CMSMSearchAdapter**, that identifies a specific bus.

ioAddr

The I/O address in the bus architecture of the adapter from where the input is to occur.

Output Parameters

None.

Return Values

An unsigned value of the appropriate size.

Remarks

These routines are used only by CHSMs written for adapters intended for bus architectures that have an I/O address space.

InBuffx

Takes a bus identifier (*busTag*), an I/O address in that bus's I/O address space, a buffer in the CPU's logical address space, and a count of items and does the necessary operations to get the specified number of data units (in the specified size) and puts them in the buffer.

Syntax

```
#include <odi.h>
#include <odi_nbi.h>

ODI_NBI InBuff8 (
    UINT8    *buffer,
    void      *busTag,
    const void *ioAddr,
    UINT32    count );

ODI_NBI InBuff16 (
    UINT16    *buffer,
    void      *busTag,
    const void *ioAddr,
    UINT32    count );

ODI_NBI InBuff32 (
    UINT32    *buffer,
    void      *busTag,
    const void *ioAddr,
    UINT32    count );
```

Input Parameters

buffer

The memory address in logical address space where the data is to be written.

busTag
Architecture-dependent value, returned by **CMSMSearchAdapter**, that identifies a specific bus.

ioAddr
The I/O address in the bus architecture of the LAN adapter (or MLID) from which the input is to be read.

count
The number of items to input.

Output Parameters

None.

Return Values

| | |
|-------------------------------|--|
| ODI_NBI_SUCCESSFUL | The requested operation was completed successfully. |
| ODI_NBI_PROTECTION_VIOLATION | Memory protection prevented the completion of the requested operation. |
| ODI_NBI_MEMORY_ERROR | Memory error occurred while attempting to perform the requested operation. |
| ODI_NBI_PARAMETER_ERROR | One of the parameters was invalid. |
| ODI_NBI_UNSUPPORTED_OPERATION | The requested operation could not be completed. |

Remarks

These routines are used by CHSMs that have an I/O space. A buffer is filled with data from the specified I/O address with the number of data units specified. The I/O address is not incremented, but the buffer address will fill forward.

MapBusMemory

Takes a bus identifier (*busTag1*), a physical memory address, and a length and makes the described piece of memory accessible from another specified bus (*busTag2*).

Syntax

```
#include <odi.h>
#include <odi_nbi.h>

ODI_NBI MapBusMemory (
    void *busTag1,
    const void *memAddr,
    void *busTag2,
    void **mappedAddr,
    UINT32 len );
```

Input Parameters

busTag1

Architecture-dependent value, returned by **CMSMSearchAdapter**, which specifies the bus on which the memory to be mapped resides.

memAddr

The physical memory address of the memory to be mapped in the bus architecture specified by *busTag1*.

busTag2

Architecture-dependent value, returned by **CMSMSearchAdapter**, which specifies the bus on which the memory is to be mapped. The caller sets *busTag2* equal to -1 to specify that the target of the mapping is the CPU's physical address space, regardless of which bus the CPU is actually connected to.

len

The minimum length of the space to be mapped in bytes.

Output Parameters

mappedAddr
The physical memory address in the bus architecture specified by *busTag2* that can be used from *busTag2* to access *busTag1 / memAddrPtr*.

Return Values

| | |
|------------------------------|--|
| ODI_NBI_SUCCESSFUL | The requested operation was completed successfully. |
| ODI_NBI_PROTECTION_VIOLATION | Memory protection prevented the completion of the requested operation. |
| ODI_NBI_HARDWARE_ERROR | Hardware error or hardware limitation prevented the completion of the requested operation. |
| ODI_NBI_PARAMETER_ERROR | One of the parameters was invalid. |

Remarks

The purpose of this function is to supply MLIDs for bus master adapters with a means to get any needed address mapping performed.

When this function returns, the appropriate physical pages have been locked.

See Also

FreeBusMemory

MovFastFromBus

Moves the contents of the source buffer on the adapter to the destination buffer in the CPU's logical address space as fast as the platform can move it.

Syntax

```
#include <odi.h>
#include <odi_nbi.h>

ODI_NBI MovFastFromBus (
    void    *destAddr,
    void    *srcBusTag,
    void    *reserved,
    const void *srcAddr,
    UINT32  count );
```

Input Parameters

destAddr

The memory address in the logical address space of the CPU where the data is to be written.

srcBusTag

Architecture-dependent value, returned by **CMSMSearchAdapter**, which specifies the bus from which data is to be moved.

reserved

This parameter is reserved and must be set to NULL.

srcAddr

The address of the source based on the information returned by **MapBusMemory**.

count

The number of items to be moved.

Output Parameters

None.

Return Values

| | |
|-------------------------------|--|
| ODI_NBI_SUCCESSFUL | The requested operation was completed successfully. |
| ODI_NBI_PROTECTION_VIOLATION | Memory protection prevented the completion of the requested operation. |
| ODI_NBI_MEMORY_ERROR | Memory error occurred while attempting to perform the requested operation. |
| ODI_NBI_PARAMETER_ERROR | One of the parameters was invalid. |
| ODI_NBI_UNSUPPORTED_OPERATION | The requested operation could not be completed. |

MovFastToBus

Moves the contents of the source buffer in the CPU logical address space into the destination buffer on the adapter as fast as the platform can move it.

Syntax

```
#include <odi.h>
#include <odi_nbi.h>

ODI_NBI  MovFastToBus (
    void    *destBusTag,
    void     *reserved,
    void    *destAddr,
    const void *srcAddr,
    UINT32   count );
```

Input Parameters

destBusTag

Architecture-dependent value, returned by **CMSMSearchAdapter**, which specifies the bus to which data is to be moved.

reserved

This parameter is reserved and must be set to NULL.

destAddr

The address of the destination based on the information returned by **MapBusMemory**.

srcAddr

The memory address in the logical address space of the CPU from which the data is to be read.

count

The number of items to be moved.

Output Parameters

None.

Return Values

| | |
|-------------------------------|--|
| ODI_NBI_SUCCESSFUL | The requested operation was completed successfully. |
| ODI_NBI_PROTECTION_VIOLATION | Memory protection prevented the completion of the requested operation. |
| ODI_NBI_MEMORY_ERROR | Memory error occurred while attempting to perform the requested operation. |
| ODI_NBI_PARAMETER_ERROR | One of the parameters was invalid. |
| ODI_NBI_UNSUPPORTED_OPERATION | The requested operation could not be completed. |

MovFromBusx

Moves the contents of the source buffer on the adapter to the destination buffer in the CPU's logical address space.

Syntax

```
#include <odi.h>
#include <odi_nbi.h>

ODI_NBI  MovFromBus8 (
    void    *destAddr,
    void    *srcBusTag,
    void    *reserved,
    const void *srcAddr,
    UINT32  count );

ODI_NBI  MovFromBus16 (
    void    *destAddr,
    void    *srcBusTag,
    void    *reserved,
    const void *srcAddr,
    UINT32  count );

ODI_NBI  MovFromBus32 (
    void    *destAddr,
    void    *srcBusTag,
    void    *reserved,
    const void *srcAddr,
    UINT32  count );
```

Input Parameters

destAddr

The memory address in the logical address space of the CPU where the data is to be written.

8-22 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

srcBusTag
Architecture-dependent value, returned by **CMSMSearchAdapter**, which specifies the bus from which data is to be moved.

reserved
This parameter is reserved and must be set to NULL.

srcAddr
The address of the source, based on the information returned by **MapBusMemory**.

count
The number of items to be moved.

Output Parameters

None.

Return Values

| | |
|-------------------------------|--|
| ODI_NBI_SUCCESSFUL | The requested operation was completed successfully. |
| ODI_NBI_PROTECTION_VIOLATION | Memory protection prevented the completion of the requested operation. |
| ODI_NBI_MEMORY_ERROR | Memory error occurred while attempting to perform the requested operation. |
| ODI_NBI_PARAMETER_ERROR | One of the parameters was invalid. |
| ODI_NBI_UNSUPPORTED_OPERATION | The requested operation could not be completed. |

Remarks

If possible, data will be moved in the size data objects (such as 8, 16, 32) specified by the name of the routine called. If this is impossible on some platforms, due to hardware constraints, some adapters will not work on that platform. The NBI will deal with alignment issues, mapping hardware issues, and length issues. Errors can be returned only if a hardware failure is detected or if the user does not have permission to access the memory indicated.

8-24 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

MovToBusx

Moves the contents of the source buffer in the CPU logical address space into the destination buffer on the adapter.

Syntax

```
#include <odi.h>
#include <odi_nbi.h>

ODI_NBI  MovToBus8 (
    void  *destBusTag,
    void  *reserved,
    void  *destAddr,
    const void  *srcAddr,
    UINT32  count );

ODI_NBI  MovToBus16 (
    void  *destBusTag,
    void  *reserved,
    void  *destAddr,
    const void  *srcAddr,
    UINT32  count );

ODI_NBI  MovToBus32 (
    void  *destBusTag,
    void  *reserved,
    void  *destAddr,
    const void  *srcAddr,
    UINT32  count );
```

Input Parameters

- destBusTag*
Architecture-dependent value, returned by **CMSMSearchAdapter**, which specifies the bus to which data is to be moved.
- reserved*
This parameter is reserved and must be set to NULL.
- destAddr*
The address of the destination based on the information returned by **MapBusMemory**.
- srcAddr*
The memory address in the logical address space of the CPU from where the data is to be read.
- count*
The number of items to be moved.

Output Parameters

None.

Return Values

| | |
|-------------------------------|--|
| ODI_NBI_SUCCESSFUL | The requested operation was completed successfully. |
| ODI_NBI_PROTECTION_VIOLATION | Memory protection prevented the completion of the requested operation. |
| ODI_NBI_MEMORY_ERROR | Memory error occurred while attempting to perform the requested operation. |
| ODI_NBI_PARAMETER_ERROR | One of the parameters was invalid. |
| ODI_NBI_UNSUPPORTED_OPERATION | The requested operation could not be completed. |

Remarks

If possible, data will be moved in the size data objects (such as 8, 16, 32) specified by the name of the routine called. If this is impossible on some platforms, due to hardware constraints, some adapters will not work on that platform. The NBI will deal with alignment issues, mapping hardware issues, and length issues. Errors can be returned only if a hardware failure is detected or if the user does not have permission to access the memory indicated.

Outx

Takes a bus identifier (*busTag*), a value, and an I/O address in that bus's address space and performs whatever operations are necessary to deliver the value to the specified address.

Syntax

```
#include <odi.h>
#include <odi_nbi.h>

void Out8 (
    void *busTag,
    const void *ioAddr,
    UINT8 outputVal );

void Out16 (
    void *busTag,
    const void *ioAddr,
    UINT16 outputVal );

void Out32 (
    void *busTag,
    const void *ioAddr,
    UINT32 outputVal );
```

Input Parameters

busTag

Architecture-dependent value, returned by **CMSMSearchAdapter**, which specifies the bus to which data is to be moved.

ioAddr

The I/O address in the bus architecture of the LAN adapter to which the output is to occur.

outputVal

The value to be sent to the specified I/O address on the specified bus. The type of this value varies depending on which function is called.

Output Parameters

None.

Return Values

None.

Remarks

These routines are used only by CHSMs written for LAN adapters intended for bus architectures that have an I/O address space.

OutBuffx

Takes a bus identifier (*busTag*), an I/O address in that address space, a buffer in the CPU's logical address space and performs whatever operations are necessary to output the specified number of data units (in the specified size) from the buffer to the I/O address.

Syntax

```
#include <odi.h>
#include <odi_nbi.h>

ODI_NBI OutBuff8 (
    void *busTag,
    void *ioAddr,
    const void *buffer,
    UINT32 count );

ODI_NBI OutBuff16 (
    void *busTag,
    void *ioAddr,
    const void *buffer,
    UINT32 count );

ODI_NBI OutBuff32 (
    void *busTag,
    void *ioAddr,
    const void *buffer,
    UINT32 count );
```

Input Parameters

busTag

Architecture-dependent value, returned by **CMSMSearchAdapter**, which specifies the bus to which data is to be moved.

8-30 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

ioAddr
The I/O address in the bus architecture of the LAN adapter (or MLID) to which the output is to occur.

buffer
The memory address in the logical address space of the CPU from which the output is to occur.

count
The number of items to output.

Output Parameters

None.

Return Values

| | |
|-------------------------------|--|
| ODI_NBI_SUCCESSFUL | The requested operation was completed successfully. |
| ODI_NBI_PROTECTION_VIOLATION | Memory protection prevented the completion of the requested operation. |
| ODI_NBI_MEMORY_ERROR | Memory error occurred while attempting to perform the requested operation. |
| ODI_NBI_PARAMETER_ERROR | One of the parameters was invalid. |
| ODI_NBI_UNSUPPORTED_OPERATION | The requested operation could not be completed. |

Remarks

These routines are used by CHSMs that have an I/O space. Data from a buffer is output to the specified I/O address with the number of data units specified. The I/O address is not incremented, but the buffer address is incremented.

Rdx

Takes a bus identifier and a physical memory address in that bus's memory address space and performs whatever operations are necessary to acquire and return the requested data.

Syntax

```
#include <odi.h>
#include <odi_nbi.h>

UINT8  Rd8 (
    void  *busTag,
    void  *reserved,
    const void  *memAddr );

UINT16 Rd16 (
    void  *busTag,
    void  *reserved,
    const void  *memAddr );

UINT32 Rd32 (
    void  *busTag,
    void  *reserved,
    const void  *memAddr );
```

Input Parameters

busTag

Architecture-dependent value, returned by **CMSMSearchAdapter**, which specifies the bus from which data is to be moved.

reserved

This parameter is reserved and must be set to NULL.

memAddr

The address of the source based on the information returned by **MapBusMemory**.

Output Parameters

None.

Return Values

An unsigned value of the appropriate size.

Setx

Fills a buffer with a specified value.

Syntax

```
#include <odi.h>
#include <odi_nbi.h>

ODI_NBI Set8 (
    void *busTag,
    void *reserved,
    const void *memAddr,
    UINT8 value,
    UINT32 count );

ODI_NBI Set16 (
    void *busTag,
    void *reserved,
    const void *memAddr,
    UINT16 value,
    UINT32 count );

ODI_NBI Set32 (
    void *busTag,
    void *reserved,
    const void *memAddr,
    UINT32 value,
    UINT32 count );
```

Input Parameters

busTag

Architecture-dependent value, returned by **CMSMSearchAdapter**, which specifies the bus on which the operation is to be performed.

reserved

This parameter is reserved and must be set to NULL.

memAddr
The address of the destination based on the information returned by **MapBusMemory**.

value
The value to be duplicated into the specified memory block on the specified bus. The type of this value varies depending on which function is called.

count
The number of items to be moved.

Output Parameters

None.

Return Values.

| | |
|------------------------------|--|
| ODI_NBI_SUCCESSFUL | The requested operation was completed successfully. |
| ODI_NBI_PROTECTION_VIOLATION | Memory protection prevented the completion of the requested operation. |
| ODI_NBI_MEMORY_ERROR | Memory error occurred while attempting to perform the requested operation. |
| ODI_NBI_PARAMETER_ERROR | One of the parameters was invalid. |

Remarks

If possible, data will be written in the size units specified by the name of the routine called. If this is impossible on some platform due to hardware constraints, some LAN adapters will not be usable on that platform. The NBI will deal with alignment issues, mapping hardware issues, and length issues. Errors can be returned only if a hardware failure is detected or the user does not have permission to access the memory indicated.

Slow

A 0.5 microsecond NOP.

Syntax

```
#include <odi.h>
#include <odi_nbi.h>

void Slow (void);
```

Input Parameters

None.

Output Parameters

None.

Return Values

None.

Remarks

This function will be implemented in a processor speed independent manner, if possible, to prevent problems between platform models and current (and future) models. This function is necessary because some hardware restricts how rapidly two successive accesses can be made, and unfortunately, not all hardware protects itself against attempts to make successive accesses too rapidly. This function allows MLIDs written for hardware which has such problems to work on a wider variety of platforms than would be possible otherwise.

Wrtx

Takes a value, a bus identifier, and a memory address in that bus's memory address space and performs whatever operations are necessary to deliver the value to the specified address.

Syntax

```
#include <odi.h>
#include <odi_nbi.h>

void Wrt8 (
    void *busTag,
    void *reserved,
    void *memAddr,
    UINT8 writeVal);

void Wrt16 (
    void *busTag,
    void *reserved,
    void *memAddr,
    UINT16 writeVal);

void Wrt32 (
    void *busTag,
    void *reserved,
    void *memAddr,
    UINT32 writeVal);
```

Input Parameters

busTag

Architecture-dependent value, returned by **CMSMSearchAdapter**, which specifies the bus to which data is to be moved.

reserved

This parameter is reserved and must be set to NULL.

memAddr

The destination address based on the information returned by **MapBusMemory**.

writeVal

The value to be sent to the specified memory address on the specified bus.
The type of this value varies depending on which function is called.

Output Parameters

None.

Return Values

None.

Appendix **A** *Language Enabling*

Overview

A language-enabled CHSM allows you to change the language in which the CHSM's messages are displayed.

A set of tools is available from Novell that allows you to language enable your CHSMs. These tools are part of the *NetWare Client SDK* software development kit. The documentation on using the language enabling tools is in the *Using the Message Enabling Tools* document of the kit. By using these tools and following the instructions in the documentation, you should be able to easily language enable a CHSM.

Language Enabling Procedure

The language enabling tools are designed to be used on a completed CHSM. In other words, you do not need to do anything special in writing your CHSM except write your CHSM to this specification. The tools will make the necessary modifications to your source files for you.

If you are going to language enable your MLID, you should do the following:

1. Make sure you have the *NetWare Client SDK* software development kit.

You should review the chapter that describes language enabling before or soon after you start writing your CHSM, so you will be familiar with the process when the time comes.

2. Complete your CHSM to this specification.
3. Use the language enabling tools on the completed CHSM to language enable your CHSM.

Language Enabling **A-1**

A-2 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

Appendix **B**

Event Control Blocks (ECBs)

Overview

This appendix describes the Event Control Block (ECB), the ECB structure, and each of the fields in the ECB structure. This appendix is especially useful for those developing for ECB aware LAN adapters.

ECB Aware Adapters

This appendix defines the general Event Control Block (ECB) structure and illustrates its relationship to the RCB and TCB. This appendix does not apply to most MLIDs written with the CMSM / CTSM interface.

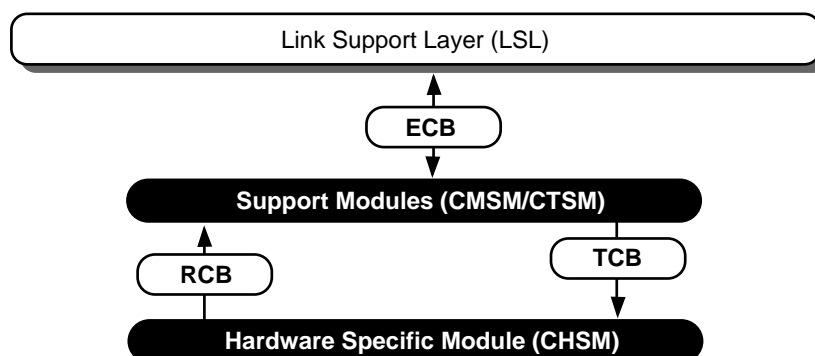
MLIDs written using the CMSM / CTSM interface typically interact with RCBs and TCBs during packet transactions as shown in Figure B-1. However, some MLIDs need to bypass these CMSM provided structures in order to work directly with the underlying general ECB structure. This is typically the case for intelligent adapters that are designed to be "ECB aware."

An ECB aware adapter/MLID 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, which gives the processor more time to perform other tasks.

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

Event Control Blocks (ECBs) **B-1**

Figure B-1
Packet Transfer through MLID



Event Control Block Structure

```

typedef struct _FRAGMENT_STRUCT_
{
    void                *FragmentAddress;
    UINT32              FragmentLength;
}FRAGMENT_STRUCT;

typedef struct _ECB_
{
    struct _ECB_        *ECB_NextLink;
    struct _ECB_        *ECB_PreviousLink;
    UINT16              ECB_Status;
    void                (*ECB_ESR)(struct _ECB_ *);
    UINT16              ECB_StackID;
    PROT_ID             ECB_ProtocolID;
    UINT32              ECB_BoardNumber;
    NODE_ADDR           ECB_ImmediateAddress;
    union
    {

```

B-2 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

```

        UINT8          Dws_i8val[4];
        UINT16         Dws_i16val[2];
        UINT32         Dws_i32val;
        void            *Dws_pval;
    }   ECB_DriverWorkspace;
union
{
    UINT8          Pws_i8val[8];
    UINT16         Pws_i16val[4];
    UINT32         Pws_i32val[2];
    UINT64         Pws_i64val;
    void            *Pws_pval[2];
}   ECB_ProtocolWorkspace;
UINT32          ECB_DataLength;
UINT32          ECB_FragmentCount;
FRAGMENT_STRUCT ECB_Fragment[1];
}   ECB;

```

Table B-1
Fragment Structure and ECB Field Descriptions

| Name | Description |
|-----------------|---|
| FragmentAddress | Pointer to a data buffer of <i>FragmentLength</i> bytes. |
| FragmentLength | Length of the buffer (in bytes) pointed to by <i>FragmentAddress</i> . This field can be 0, in which case the MLID will skip over it when transmitting or receiving data. |
| ECB_NextLink | Typically used as a forward link to a list of ECBs. The current owner of the ECB (such as the protocol stack) uses this field. |

Event Control Blocks (ECBs) **B-3**

Table B-1
Fragment Structure and ECB Field Descriptions *continued*

| Name | Description | | | | | | | | | | | | | | | | | | | | |
|---|--|-----------|-------------|-------------|--|-------------|------------------------------|-------------|--------------|-------------|--------------------------------------|-------------|--|-------------|--|-------------|--|-------------|---|---|--|
| ECB_PreviousLink | Typically used as a back link to manage a list of ECBs. The current owner of the ECB uses this field. When an ECB is returned from an MLID containing a received packet, this field contains the received packet error status defined as follows: <table><tr><td>Bit Value</td><td>Description</td></tr><tr><td>0x0000 0001</td><td>CRC error (for example, frame check sequence (FCS) error).</td></tr><tr><td>0x0000 0002</td><td>CRC / Frame alignment error.</td></tr><tr><td>0x0000 0004</td><td>Runt packet.</td></tr><tr><td>0x0000 0010</td><td>Packet larger than allowed by media.</td></tr><tr><td>0x0000 0020</td><td>Received packet for a frame type not supported, for example, logical board not registered for the frame type of the received packet. A board number associated with the physical adapter is placed in the LOOKAHEAD structure.</td></tr><tr><td>0x0000 0040</td><td>Malformed packet. For example, packet size smaller than minimum size for media header (for example, incomplete MAC header). Contents of the length field in an Ethernet 802.3 header is larger than the total packet size.</td></tr><tr><td>0x0000 4000</td><td>Do not decompress the received packet.</td></tr><tr><td>0x0000 8000</td><td>The address present in <i>ECB_ImmediateAddress</i> is in noncanonical format.</td></tr><tr><td colspan="2">If no error bits are set, the packet was received without error and the data can be used. All undefined bits are cleared.</td></tr></table> | Bit Value | Description | 0x0000 0001 | CRC error (for example, frame check sequence (FCS) error). | 0x0000 0002 | CRC / Frame alignment error. | 0x0000 0004 | Runt packet. | 0x0000 0010 | Packet larger than allowed by media. | 0x0000 0020 | Received packet for a frame type not supported, for example, logical board not registered for the frame type of the received packet. A board number associated with the physical adapter is placed in the LOOKAHEAD structure. | 0x0000 0040 | Malformed packet. For example, packet size smaller than minimum size for media header (for example, incomplete MAC header). Contents of the length field in an Ethernet 802.3 header is larger than the total packet size. | 0x0000 4000 | Do not decompress the received packet. | 0x0000 8000 | The address present in <i>ECB_ImmediateAddress</i> is in noncanonical format. | If no error bits are set, the packet was received without error and the data can be used. All undefined bits are cleared. | |
| Bit Value | Description | | | | | | | | | | | | | | | | | | | | |
| 0x0000 0001 | CRC error (for example, frame check sequence (FCS) error). | | | | | | | | | | | | | | | | | | | | |
| 0x0000 0002 | CRC / Frame alignment error. | | | | | | | | | | | | | | | | | | | | |
| 0x0000 0004 | Runt packet. | | | | | | | | | | | | | | | | | | | | |
| 0x0000 0010 | Packet larger than allowed by media. | | | | | | | | | | | | | | | | | | | | |
| 0x0000 0020 | Received packet for a frame type not supported, for example, logical board not registered for the frame type of the received packet. A board number associated with the physical adapter is placed in the LOOKAHEAD structure. | | | | | | | | | | | | | | | | | | | | |
| 0x0000 0040 | Malformed packet. For example, packet size smaller than minimum size for media header (for example, incomplete MAC header). Contents of the length field in an Ethernet 802.3 header is larger than the total packet size. | | | | | | | | | | | | | | | | | | | | |
| 0x0000 4000 | Do not decompress the received packet. | | | | | | | | | | | | | | | | | | | | |
| 0x0000 8000 | The address present in <i>ECB_ImmediateAddress</i> is in noncanonical format. | | | | | | | | | | | | | | | | | | | | |
| If no error bits are set, the packet was received without error and the data can be used. All undefined bits are cleared. | | | | | | | | | | | | | | | | | | | | | |

B-4 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

Table B-1

Fragment Structure and ECB Field Descriptions *continued*

| Name | Description |
|------------|--|
| ECB_Status | <p>Completion status of an ECB. This field is invalid until the associated event service routine is called. The following are the possible return values.</p> <p>ODISTAT_SUCCESSFUL Packet was received successfully.</p> <p>ODISTAT_RX_OVERFLOW Packet was too big to fit into the fragments described by the ECB. However, only the portion of the packet that overflowed the buffer was lost; the buffer contains as much data as it could hold.</p> <p>ODISTAT_CANCELED The ECB was not needed by the MLID. The MLID signals to the protocol stack that the ECB was not transmitted.</p> <p>ODISTAT_MLID_SHUTDOWN The LAN adapter specified in the <i>ECB_BoardNumber</i> field cannot be found. This usually means that the MLID has been removed from memory by shut down (temporarily or permanently).</p> <p>ODISTAT_BAD_PARAMETER The ECB contains bad parameters—for example, the amount of data to transmit exceeds the maximum possible for the MLID. The ECB will not have been transmitted.</p> <p>Note: The return values are ODISTAT cast as UINT16.</p> |
| ECB_ESR | The protocol stack sets this field to point to an appropriate routine that is called when the send or receive event is complete (either successfully or with an error). This field must point to a valid handler ((*ECB_ESR)(ECB*)). |

Event Control Blocks (ECBs) **B-5**

Table B-1
Fragment Structure and ECB Field Descriptions *continued*

| Name | Description | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---------------------|--|----------------------------------|--------|--------------|---------------------|--------|---------------------------------|---------------------|--------|--|---------------------|--------|--|---------------------|--------|--|---------------------|--------|--|---------------------|---------|--|---------------------|---------|----------------------------------|-----------------|---------|-----------------------------|-----------------|---------|------------------------------|-----------------|---------|--|-----------------|---------|--|-----------------|---------|--|-----------------|---------|--|-----------------|---------|--|-----------------|---------|----------------------------------|
| ECB_StackID | <p>When a packet is transmitted, the protocol stack sets this field to the protocol stack's assigned Stack ID (SID) before the protocol stack sends the ECB to the LSL. When a packet is being received, the LSL sets this field to the Stack ID assigned to the protocol stack that is receiving the packet. If a packet is being transmitted as a raw send, the protocol stack can set this field to 0xFFFF as a signal to the underlying MLID that this is a raw send. This gives the protocol stack the ability to specify the complete packet, including all low-level headers.</p> <p>The following values are valid for the <i>ECB_StackID</i> field:</p> <table><tr><td>RAW_SEND_PRIORITY_0</td><td>0xFFFF</td><td>No Priority.</td></tr><tr><td>RAW_SEND_PRIORITY_1</td><td>0xFFFE</td><td>Scale 1-7: 1 = Lowest Priority.</td></tr><tr><td>RAW_SEND_PRIORITY_2</td><td>0xFFFD</td><td></td></tr><tr><td>RAW_SEND_PRIORITY_3</td><td>0xFFFC</td><td></td></tr><tr><td>RAW_SEND_PRIORITY_4</td><td>0xFFFB</td><td></td></tr><tr><td>RAW_SEND_PRIORITY_5</td><td>0xFFFA</td><td></td></tr><tr><td>RAW_SEND_PRIORITY_6</td><td>0xFFFF9</td><td></td></tr><tr><td>RAW_SEND_PRIORITY_7</td><td>0xFFFF8</td><td>Scale 1-7: 7 = Highest Priority.</td></tr><tr><td>SEND_PRIORITY_0</td><td>0xFFFF7</td><td>Scale 1-7: 0 = No Priority.</td></tr><tr><td>SEND_PRIORITY_1</td><td>0xFFFF6</td><td>Scale 1-7: 1 = Low Priority.</td></tr><tr><td>SEND_PRIORITY_2</td><td>0xFFFF5</td><td></td></tr><tr><td>SEND_PRIORITY_3</td><td>0xFFFF4</td><td></td></tr><tr><td>SEND_PRIORITY_4</td><td>0xFFFF3</td><td></td></tr><tr><td>SEND_PRIORITY_5</td><td>0xFFFF2</td><td></td></tr><tr><td>SEND_PRIORITY_6</td><td>0xFFFF1</td><td></td></tr><tr><td>SEND_PRIORITY_7</td><td>0xFFFF0</td><td>Scale 1-7: 7 = Highest Priority.</td></tr></table> | RAW_SEND_PRIORITY_0 | 0xFFFF | No Priority. | RAW_SEND_PRIORITY_1 | 0xFFFE | Scale 1-7: 1 = Lowest Priority. | RAW_SEND_PRIORITY_2 | 0xFFFD | | RAW_SEND_PRIORITY_3 | 0xFFFC | | RAW_SEND_PRIORITY_4 | 0xFFFB | | RAW_SEND_PRIORITY_5 | 0xFFFA | | RAW_SEND_PRIORITY_6 | 0xFFFF9 | | RAW_SEND_PRIORITY_7 | 0xFFFF8 | Scale 1-7: 7 = Highest Priority. | SEND_PRIORITY_0 | 0xFFFF7 | Scale 1-7: 0 = No Priority. | SEND_PRIORITY_1 | 0xFFFF6 | Scale 1-7: 1 = Low Priority. | SEND_PRIORITY_2 | 0xFFFF5 | | SEND_PRIORITY_3 | 0xFFFF4 | | SEND_PRIORITY_4 | 0xFFFF3 | | SEND_PRIORITY_5 | 0xFFFF2 | | SEND_PRIORITY_6 | 0xFFFF1 | | SEND_PRIORITY_7 | 0xFFFF0 | Scale 1-7: 7 = Highest Priority. |
| RAW_SEND_PRIORITY_0 | 0xFFFF | No Priority. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| RAW_SEND_PRIORITY_1 | 0xFFFE | Scale 1-7: 1 = Lowest Priority. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| RAW_SEND_PRIORITY_2 | 0xFFFD | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| RAW_SEND_PRIORITY_3 | 0xFFFC | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| RAW_SEND_PRIORITY_4 | 0xFFFB | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| RAW_SEND_PRIORITY_5 | 0xFFFA | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| RAW_SEND_PRIORITY_6 | 0xFFFF9 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| RAW_SEND_PRIORITY_7 | 0xFFFF8 | Scale 1-7: 7 = Highest Priority. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| SEND_PRIORITY_0 | 0xFFFF7 | Scale 1-7: 0 = No Priority. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| SEND_PRIORITY_1 | 0xFFFF6 | Scale 1-7: 1 = Low Priority. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| SEND_PRIORITY_2 | 0xFFFF5 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| SEND_PRIORITY_3 | 0xFFFF4 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| SEND_PRIORITY_4 | 0xFFFF3 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| SEND_PRIORITY_5 | 0xFFFF2 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| SEND_PRIORITY_6 | 0xFFFF1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| SEND_PRIORITY_7 | 0xFFFF0 | Scale 1-7: 7 = Highest Priority. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| ECB_ProtocolID | <p>The Protocol ID (PID) value for sends and receives.</p> <p>For send ECBs, the protocol stack sets this field before calling SendPacket. For send ECBs, the PID is embedded into the low-level packet header by the underlying MLID and is used to uniquely identify the packet as the caller's protocol type.</p> <p>For receive ECBs, the protocol stack puts the Protocol ID, supplied by the LOOKAHEAD structure, in this field.</p> <p>The PID is stored in high-low order.</p> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

Table B-1

Fragment Structure and ECB Field Descriptions *continued*

| Name | Description |
|----------------------|--|
| ECB_BoardNumber | <p>When an MLID registers with the LSL, the MLID is given a logical board number. The <i>BoardNumber</i> field of the configuration table contains this board number.</p> <p>For send ECBs, the protocol stack puts the target logical board number in this field.</p> <p>For receive ECBs, the protocol stack puts the board number, supplied by the LOOKAHEAD structure, in this field.</p> |
| ECB_ImmediateAddress | <p>If the ECB is a send ECB, the protocol stack sets this field before calling SendPacket, and the immediate address is the destination address of the packet on the physical network. If the ECB is a receive ECB, the protocol stack fills in this field with the immediate address supplied in the LOOKAHEAD structure. This source address is the node on the same physical network that just sent the packet. If the MLID is utilizing canonical addressing, the immediate address is in canonical form.</p> |

Event Control Blocks (ECBs) **B-7**

Table B-1

Fragment Structure and ECB Field Descriptions *continued*

| Name | Description | | | | | | | | | | | | | | | | | | |
|---------------------|--|-----|-------------|------|---|------|---|------|--|------|---|------|---|------|---|------|--|------|---|
| ECB_DriverWorkspace | <p>Generally reserved for use by the MLID.</p> <p>The first byte, offset 0, of the <i>ECB_DriverWorkspace</i> field is used to indicate the type of received packet and the number of data bytes present in the packet after an MLID has finished filling the ECB and the ECB is placed on the LSL event queue.</p> <table> <tr> <th>Bit</th><th>Description</th></tr> <tr> <td>0x01</td><td>Multicast: The packet was destined to a subset of group addresses on the physical network that the MLID has been programmed to support.</td></tr> <tr> <td>0x02</td><td>Broadcast: The packet was destined to all nodes on the physical network. Note that on receiving a broadcast both b0 and b1 are set to 1, since a broadcast address is also a group address.</td></tr> <tr> <td>0x04</td><td>UnicastRemote: The packet was directly destined to another workstation on the physical network. This bit is generally set only after the MLID has been entered into promiscuous mode or has received a packet due to source routing.</td></tr> <tr> <td>0x08</td><td>MulticastRemote: The packet was destined to a subset of group addresses on the physical network that the MLID has not been programmed to support.</td></tr> <tr> <td>0x10</td><td>SourceRoute: This bit is set in conjunction with other packet type bits if the packet has source routing information in the packet, in other words, the RII bit is set. If the source routing module is not loaded and the length of the source route field is greater than two bytes (packet from a remote ring) all other bits will be cleared.</td></tr> <tr> <td>0x20</td><td>GlobalError: The packet contains errors. See the <i>ECB_PreviousLink</i> field for the specific error. This is an exclusive bit; if set, all other bits should be 0. This value supersedes <i>SourceRoute</i>.</td></tr> <tr> <td>0x40</td><td>MacFrame: The packet is a non-data frame (for example, the MAC layer frame). This is an exclusive bit if set. All other bits must be zero (0). Note that MAC frames by definition are not source routable.</td></tr> <tr> <td>0x80</td><td>Direct: The packet was destined to this station only.</td></tr> </table> | Bit | Description | 0x01 | Multicast : The packet was destined to a subset of group addresses on the physical network that the MLID has been programmed to support. | 0x02 | Broadcast : The packet was destined to all nodes on the physical network. Note that on receiving a broadcast both b0 and b1 are set to 1, since a broadcast address is also a group address. | 0x04 | UnicastRemote : The packet was directly destined to another workstation on the physical network. This bit is generally set only after the MLID has been entered into promiscuous mode or has received a packet due to source routing. | 0x08 | MulticastRemote : The packet was destined to a subset of group addresses on the physical network that the MLID has not been programmed to support. | 0x10 | SourceRoute : This bit is set in conjunction with other packet type bits if the packet has source routing information in the packet, in other words, the RII bit is set. If the source routing module is not loaded and the length of the source route field is greater than two bytes (packet from a remote ring) all other bits will be cleared. | 0x20 | GlobalError : The packet contains errors. See the <i>ECB_PreviousLink</i> field for the specific error. This is an exclusive bit; if set, all other bits should be 0. This value supersedes <i>SourceRoute</i> . | 0x40 | MacFrame : The packet is a non-data frame (for example, the MAC layer frame). This is an exclusive bit if set. All other bits must be zero (0). Note that MAC frames by definition are not source routable. | 0x80 | Direct : The packet was destined to this station only. |
| Bit | Description | | | | | | | | | | | | | | | | | | |
| 0x01 | Multicast : The packet was destined to a subset of group addresses on the physical network that the MLID has been programmed to support. | | | | | | | | | | | | | | | | | | |
| 0x02 | Broadcast : The packet was destined to all nodes on the physical network. Note that on receiving a broadcast both b0 and b1 are set to 1, since a broadcast address is also a group address. | | | | | | | | | | | | | | | | | | |
| 0x04 | UnicastRemote : The packet was directly destined to another workstation on the physical network. This bit is generally set only after the MLID has been entered into promiscuous mode or has received a packet due to source routing. | | | | | | | | | | | | | | | | | | |
| 0x08 | MulticastRemote : The packet was destined to a subset of group addresses on the physical network that the MLID has not been programmed to support. | | | | | | | | | | | | | | | | | | |
| 0x10 | SourceRoute : This bit is set in conjunction with other packet type bits if the packet has source routing information in the packet, in other words, the RII bit is set. If the source routing module is not loaded and the length of the source route field is greater than two bytes (packet from a remote ring) all other bits will be cleared. | | | | | | | | | | | | | | | | | | |
| 0x20 | GlobalError : The packet contains errors. See the <i>ECB_PreviousLink</i> field for the specific error. This is an exclusive bit; if set, all other bits should be 0. This value supersedes <i>SourceRoute</i> . | | | | | | | | | | | | | | | | | | |
| 0x40 | MacFrame : The packet is a non-data frame (for example, the MAC layer frame). This is an exclusive bit if set. All other bits must be zero (0). Note that MAC frames by definition are not source routable. | | | | | | | | | | | | | | | | | | |
| 0x80 | Direct : The packet was destined to this station only. | | | | | | | | | | | | | | | | | | |

B-8 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

Table B-1

Fragment Structure and ECB Field Descriptions *continued*

| Name | Description |
|-----------------------|--|
| | <p>The second byte, offset 1, of the <i>ECB_DriverWorkspace</i> field contains the number of control bytes present in the 802.2 header.</p> <p>Bit Value Description</p> <p>0x00 No 802.2 header present in frame.</p> <p>0x01 One control byte is present in the 802.2 header</p> <p>0x02 Two control bytes are present in the 802.2 header</p> <p>0x04 The received packet is a priority packet.</p> <p>Bit value 0x04 is only valid for topologies that support a distinction in priority levels. Bit value 0x04 is not set if the received frame is at the normal priority level or lower.</p> <p>The second word, byte offsets 2 and 3, of the <i>ECB_DriverWorkspace</i> field are filled with the size of the received frame minus the MAC header, which is the total number of data bytes present in the frame.</p> |
| ECB_ProtocolWorkspace | Reserved for use by the originating protocol stack and must not be modified by the LSL or the MLIDs. |
| ECB_DataLength | If this is a send ECB, the protocol stack sets this field to the total length of the data in bytes before it calls SendPacket . If this is a receive ECB, this field is set to the length in bytes of the data that is copied into the fragment structure portion of the ECB. |
| ECB_FragmentCount | The number of fragment buffer descriptors immediately following this field. This value is greater than zero and less than or equal to 16 ($0 < ECB_FragmentCount \leq MAX_FRAG_COUNT$). |
| ECB_Fragment[1] | This field specifies a fragment structure. |

Event Control Blocks (ECBs) **B-9**

Relationship between Receive ECBs and RCBs

The general receive ECB and the CMSM’s RCB essentially form a union. That is, both structures occupy the same memory space. The following shows how the receive ECB fields are equated to the RCB fields.

Figure B-2
RCB Correspondence to ECB

| These Receive ECB Fields | Correspond to these RCB Fields |
|--------------------------|--------------------------------|
| NextLink | Driver WS |
| PreviousLink | |
| Status | |
| ESR | |
| StackID | |
| ProtocolID | Reserved |
| BoardNumber | |
| ImmediateAddress | |
| DriverWorkstation | |
| ProtocolWorkspace | |
| DataLength | FragCount |
| FragmentCount | |
| Fragment | FragStruct |
| MediaHeader | |
| Data | |

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

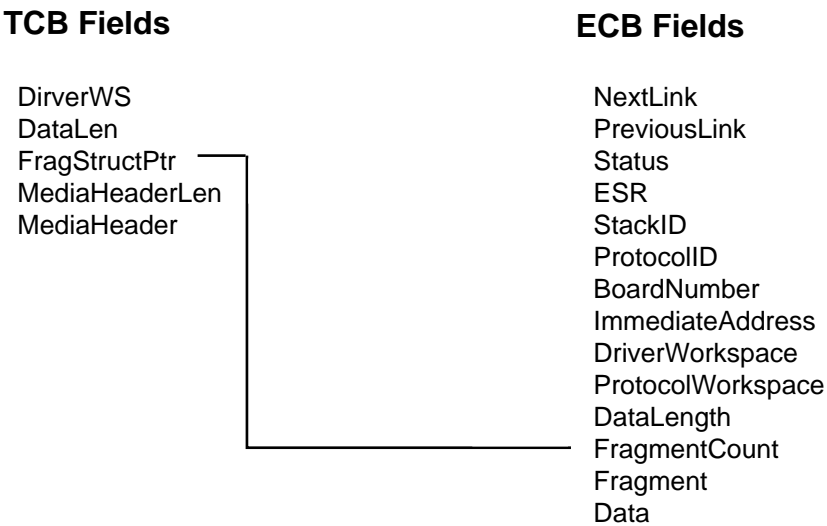
MLIDs written for ECB aware adapters must obtain control blocks by calling **CMSMAllocateRCB**. This routine allows the MLID to preallocate RCBs without the CMSM initializing the fields. When a packet is received, the adapter copies it into the RCB data buffer, fills in the required fields, and returns the structure using either the **<CTSM>RcvComplete** / **CMSMServiceEvents** combination or the function **<CTSM>FastRcvComplete**.

Relationship between Transmit ECBs and TCBs

The general transmit ECB and the CTSM's TCB are totally separate structures. The *TCBFragStruct* 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 CMSM provides another more efficient way for ECB aware adapters to work directly with ECBs. By setting the *DriverSendWantsECBs* variable of the driver parameter block to any nonzero value (see Chapter 3, "CHSM Data Structures and Variables"), the CHSM's **DriverSend** routine will be given ECBs rather than TCBs for packet transmission. The CHSM will then be responsible for building the proper media header depending on the board number. The following shows the relationship between the transmit ECB and TCB.

Figure B-3
Relationship between TCB and ECB



B-12 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

Appendix **C** *Platform Specific Information*

Overview

This appendix presents platform-specific information related to writing MLIDs. Currently, only Intel (80x86 and Pentium) processor specific information is provided. Information about other platforms will be provided in the future.

Intel Processors

The following information is specific to Intel 80x86 based processor machines.

Building the CHSM

The following describes the process of creating, compiling, linking, and loading a CHSM.

Creating the Source Files

C language NetWare drivers are written in ANSI C code. This specification provides the details for writing the driver.

Compiling the Source Files

The source file (*<driver>.c*) and header files (*odi.h*, *<ctsm>.h*, *cmsm.h*, and *odi_nbi.h*) are compiled into an object file (*<driver>.OBJ*). The driver can consist of one or more object files. Depending on the target platform, the developer may have a choice of several compilers or may be restricted to one.

Linking the Object Files

The NetWare linker (NLMLINKX) converts the *<driver>.OBJ* object file and any other object files that make up the MLID into a super object file called *<driver>.LAN*. NLMLINKX requires a linker definition file to create a NetWare Loadable Module. The linker definition file is described below. To use the linker, type:

```
nlmlinkx 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, NLMLINKX. 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 shows a definition file example that can be used to create an MLID. The file consists of keywords followed by data. The keywords can be upper or lower case.

Linker Definition File Example

```
TYPE 1
DESCRIPTION      "NetWare CNE2000"
VERSION          5,30,2
OUTPUT           <drivername>
INPUT            <.OBJ drivername>
START            DriverInit
EXIT             DriverRemove
MESSAGE          CNE2000.MSG
MODULE           ETHERTSM
REENTRANT
MAP
IMPORT           CEtherTSMRegisterHSM
                  CEtherTSMGetRCB
                  CEtherTSMRcvComplete
```

C-2 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

```

CEtherTSMComplete
CEtherTSMGetNextSend
CEtherTSMUpdateMulticast
CMSMAlloc
CMSMDriverRemove
CMSMFree
CMSMParseDriverParameters
CMSMPrintString
CMSMRegisterHardwareOptions
CMSMRegisterMLID
CMSMReturnDriverResources
CMSMScheduleAES
CMSMSetHardwareInterrupt

```

Table C-1
Linker Definition File Example Definitions

| Name | Description |
|-------------|--|
| TYPE | 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 | <p>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.</p> <p>Examples of the description string are shown here:</p> <pre> NetWare NE2000 3Com EtherLink Plus 3C503 </pre> |
| OUTPUT | Output file name. |
| INPUT | OBJ files to include in the loadable module. It is not necessary to use the filename extension in this list. |
| START | 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 | Name of the loadable module's remove routine, in this case, DriverRemove . The UNLOAD command uses this routine to unload the module from memory. |
| REENTRANT | Allows the driver to be loaded more than once, but only have the driver's code copied into memory the first time. |

Platform Specific Information **C-3**

Table C-1

Linker Definition File Example Definitions *continued*

| Name | Description |
|--|---|
| MAP | Tells the linker to create a map file. |
| IMPORT | NetWare variables and routines the loadable module must access. |
| EXPORT | A list of variable and function names resident in the loadable module that are available to other loadable modules. |
| MODULE | 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 memory, the loader will attempt to find and load them. If it cannot find them, the loader will not load the current module. |
| CUSTOM | 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 | 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. An MLID's check procedure might check to see if an adapter is currently being accessed and return a nonzero value to the NetWare operating system if the board is busy. The NetWare operating system 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 can be loaded into memory concurrently. |
| COPYRIGHT | Tells the linker to include a copyright string in the output file. A MEON 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. |
| Note: You must use NLMLINKX.EXE to use the COPYRIGHT keyword. | |

C-4 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

Table C-1

Linker Definition File Example Definitions *continued*

| Name | Description |
|---------|--|
| VERSION | <p>Gives the linker the version of the module that should be placed into the NLM header version field. The format for this keyword is:</p> <pre>VERSION Major, Minor[, Revision]</pre> <p>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.</p> <p>For example, "VERSION 3,50,2" produces the version field 3.50b in the NLM header of the output file.</p> <p>Note that to use the VERSION keyword, you must use NLMLINKX.EXE. The date is automatically set by the linker to the date that the files are linked.</p> <p>The CMSM.NLM and <CTSM>.NLM must be loaded (only once) before any CHSMs are loaded. These can be auto-loaded using the "module" keyword in the linker definition file.</p> <p>To load the driver, you could enter a command similar to this:</p> <pre>LOAD <driver>FRAME=ETHERNET_802.3, PORT=300, NODE=2608C760361, INT=3</pre> <p>The parameters do not have a set order. The commas are optional.</p> |

MLID Configuration File

CHSMs that support a large number of custom keywords may have trouble specifying all parameters on the limited space of the command line. Command line parameters can 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 in the file and at the command line type:

```
LOAD <drivername> @<response filename>
```

If this file exists in the same directory as the driver, the CMSM 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 **CMSMParseDriverParameters** routine handles the load command

Platform Specific Information C-5

parameters in drivers written using the CMSM. The load parameters and examples of their use are described below.

Table C-2

Load Keywords and Parameters Descriptions

| Name | Descriptions |
|-------------|---|
| PORT, PORT1 | <p>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.</p> <p>LOAD <driver> PORT=300 LOAD <driver> PORT=300:A LOAD <driver> PORT=300:A PORT1=700:8</p> |
| MEM, MEM1 | <p>Beginning address of the shared RAM that the board can use. The size of the shared memory buffer can also be specified.</p> <p>LOAD <driver> MEM=C0000 LOAD <driver> MEM=C0000:1000 LOAD <driver> MEM=C0000:1000 MEM1=CC000</p> |
| INT, INT1 | <p>Interrupt number that the board is expected to use to awaken the interrupt service routine.</p> <p>LOAD <driver> INT=3 LOAD <driver> INT=3 INT1=5</p> |
| DMA, DMA1 | <p>If the board supports DMA, this is the direct memory address channel that the adapter should use for data transfer to memory.</p> <p>LOAD <driver> DMA=0 LOAD <driver> DMA=0 DMA1=3</p> |
| SLOT | <p>System-wide unique Hardware Instance Number (HIN) that may be the physical slot number on a slot based bus such as Micro Channel, PCI, PC Card, EISA, or another uniquely assigned number.</p> <p>LOAD <driver> SLOT=4</p> |
| RETRIES | <p>Number of send retries that the MLID should use in its attempts to send packets.</p> <p>RETRIES = n</p> |
| CHANNEL | <p>Channel number (controller number) to use for multichannel adapters. A multichannel adapter is a board containing more than one adapter.</p> <p>CHANNEL = number</p> |

C-6 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

Table C-2

Load Keywords and Parameters Descriptions *continued*

| Name | Descriptions |
|-----------|---|
| BELOW16 | <p>This keyword must be specified on the load command line if the driver needs memory allocated below the 16MB boundary. This keyword is required only if the MLID is loaded on a system that initially has less than 16 MB of memory, but will have more memory added later using the server's REGISTER MEMORY command. In addition, the driver must also set the <i>DriverNeedsBelow16Meg</i> field of the DRIVER_PARM_BLOCK structure to a nonzero value.</p> <p>BELOW16</p> |
| FRAME | <p>String specifying the frame type (see <i>ODI Supplement: Frame Types and Protocol IDs</i> for a list of frame type strings).</p> <p>FRAME = type</p> <p>Token-Ring drivers can 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 noncanonical addresses to be passed (this is the default for Token-Ring media). Ethernet media cannot use the MSB designator.</p> |
| BUFFERS16 | <p>This keyword is used to override the number of RCBs below 16MB allocated by the CMSM at initialization. The CHSM must set the <i>DriverNeedsBelow16Meg</i> field in the DRIVER_PARM_BLOCK structure for this keyword to be valid. The RCB allocation routines (CMSMAllocRCB, <CTSM>GetRCB, <CTSM>ProcessGetRCB, etc.) use these RCBs if the RCB allocated by the LSL is physically over 16MB. The number of RCBs allocated by default is eight. If the CHSM preallocates more than eight RCBs at a time, the user can override this default when loading the driver by typing BUFFERS16=n. The CMSM will force this value to a multiple of eight, so values other than 8, 16, 32 and so on are invalid. No restriction is placed on the maximum value, except that the CMSM might not be able to allocate enough memory from the operating system.</p> |

Table C-2
Load Keywords and Parameters Descriptions *continued*

| Name | Descriptions |
|------|---|
| NODE | <p>Node address that the board is to use; this address should override the default address on the board if one exists.</p> <p>NODE = nnnnnnnnnnnn</p> <p>In the case of Token-Ring media, which has a noncanonical physical layer format, the override node address on the command line can be entered in either canonical or noncanonical format (see <i>ODI Specification Supplement: Canonical and Noncanonical Addressing</i>). To indicate the format of the address, an "L" (LSB) or an "M" (MSB) can be appended. For example, to indicate a node address for Token-Ring media in canonical format enter:</p> <p>NODE = nnnnnnnnnnnnL</p> <p>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 the MM_NONCANONICAL_BIT bit in the <i>MLIDCFG_ModeFlags</i> field.</p> |

Appendix **D** *Portability Issues*

Overview

For the CHSM to be portable across processors and operating systems, you need to do several things. This appendix describes some programming practices, assumptions, general principles, and other miscellaneous information to help you in writing a portable driver. This appendix also describes the macros that you can use to deal with the big /little endian and alignment issues of portability.

In most cases, it should be possible to port your code from one processor to another or from one operating system to another by modifying a few #defines and/or typedef statements in a few header files, and perhaps defining a pragma or setting a compiler switch.

Portability Rules

The following are rules and guidelines that you should follow to increase the probability that your code will be portable to other processors and operating systems. This is not a comprehensive list, therefore, you may need to do additional things not listed to ensure portability (test on different platforms and operating systems, learn about the specifics of hardware you are working with, etc.).

- Adhere strictly to the ANSI C specification.
- Don't make assumptions about the size of a given type, especially pointers.
- Be aware that numeric fields composed of more than 1 byte can be in one of two formats: big endian (high-low) or little endian (low-high). Big endian numbers contain the most significant byte in the lowest addressed byte of the field, the next most significant byte in the second lowest addressed byte, and so on, with the least significant byte appearing last.

Little endian numbers are stored in the opposite order. For example, Intel 80x86 microprocessors store numbers in little endian order.

- Pay attention to alignment constraints when allocating memory and using pointers. The addresses that certain operands can be assigned to are restricted on some architectures.
- Be aware that pointers to objects can have the same size but different formats.
- Do not redefine the NULL symbol. NULL should always be the constant zero.
- Make file names no more than eight main and three extension characters long.
- Always dereference the pointer when calling functions passed as arguments. For example, if "F" is a pointer to a function, use "*F" instead of "F", because some compilers may not recognize "F".
- In general, do not declare any variable to be any of the C language basic types (*short, long, int, char*). Declare variables to be of some abstract type, and typedef that type to the appropriate base type for each processor/operating system combination. In some cases, such as counters, it may be more efficient to use *int* instead of an abstract type.
- Make sure that all members in any structure that describes data coming in from or going out to the LAN are given unique, abstract types. Also, make sure that all references to these members use the appropriate misalignment correction and byte order correction macros.
- Isolate processor and operating system code into separate modules and use conditional compilation to make it easier to port your code.
- Do not modify string constants, because many implementations place the constants into read-only memory. (This is required by the ANSI C standard.)
- Enclose *#pragma* directives with *#ifdef*'s in order to document under which platform they make sense (suggested).
- Protect header files (to ensure portability, do not modify any of the header files provided by Novell).

D-2 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

- Use the *sizeof* operator to determine the size of an object, rather than making an assumption or hard-coding a value.
- Use the *offsetof* macro to determine the offset of a member within a structure, rather than making an assumption or hard-coding a value.
- Initialize all data.
- Do not depend on parameter passing conventions, especially assumptions about which parameters will be passed on the stack or in registers.
- Do not access arrays based on a knowledge of the storage method. Use the standard C language access methods instead of computing offsets into the array.
- Do not assume a stack growth direction.
- Use the *varargs* features to implement functions that require variable arguments.
- Pay attention to word sizes. Objects may be non-intuitive sizes. Pointers are not always the same size as *ints*, the same size as each other, or freely interconvertible.
- Be aware that some machines have more than one possible size for a given type. The size you get can depend upon both the compiler and compile-time flags.
- Understand that the *void** type is guaranteed to have enough bits of precision to hold a pointer to any data object.
- Be aware that even when, say, an *int** and *char** are the same size, they may have different formats.
- Understand that the integer *constant* zero may be cast to any pointer type. The resulting pointer is called a NULL pointer for that type and is different from any other pointer of that type. A NULL pointer always compares equal to the constant zero. A NULL pointer might not compare equal with a variable that has the value zero. NULL pointers are not always stored with all bits zero. NULL pointers for two different types are sometimes different. A NULL pointer of one type cast in to a pointer of another type will be cast in to the NULL pointer for that second type.

Portability Issues **D-3**

- Watch out for signed characters. Code that assumes signed/unsigned is not portable.
- Avoid assuming ASCII. Characters may hold more than 8 bits.
- Do not use code that takes advantage of two's complement representation of numbers in most cases.
- Be aware that there may be unused holes in structures. Suspect *unions* used for type cheating. Specifically, a value should not be stored as one type and retrieved as another.
- Be aware that different compilers use different conventions for returning structures.
- Be aware that the address space can have holes. Simply computing the address of an unallocated element in an array can crash the program.
- Be aware that only the == and != comparisons are defined for all pointers of a given type. It is only portable to use <, <=, >, or >= to compare pointers when they both point in to (or to the first element after) the same array. It is likewise only portable to use arithmetic operators on pointers that both point into the same array or the first element afterwards.
- Be aware that side effects within expressions can result in code whose semantics are compiler-dependent, since the order of evaluation is explicitly undefined in most places in the C language.

Translation Limits

The following are translation limits that you should follow to ensure portability between operating systems and processors. The following are maximum values:

- Eight nesting levels of conditional inclusion.
- Eight nesting levels for *#include* files
- 32 nesting levels of parenthesized expressions within a full expression.
- 1024 macro identifiers simultaneously.
- 509 characters in a logical source line.

D-4 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

- Six significant initial characters in an external identifier.
- 127 members in a single structure or union.
- 31 parameters in one function call.

Coding Assumptions

The following are assumptions that need to be made in writing your code.

- All target architectures will align 8-bit items on 8-bit boundaries, 16-bit items on 16-bit boundaries, and 32-bit items on 32-bit boundaries.
- All compilers support the *volatile* data type qualifier.
- The compiler and architecture will align structures to the alignment of the largest data item within the structure (for example, a structure whose largest element is a byte can be byte aligned).

Data Packing and Alignment

The ANSI C specification states that you cannot assume that the members of a structure will be contiguous. The compiler for many processors will insert padding into a structure to force each member to begin on the alignment value appropriate for its type. This is done because many processors will cause a processor exception if an attempt is made to access "misaligned data." This causes problems because the MAC header cannot be described as a structure in many media types. In these media, the members of the MAC header structure are not guaranteed to be properly aligned, either in the structure definition, which prevents the computer from inserting padding, or in memory, which prevents processor exceptions. This implies two requirements to the CHSM developer:

- All members of such structures should be declared as types not used anywhere except in such structure declarations. This allows these types to be declared in a header file that is platform dependent. On platforms that have no alignment restrictions or on platforms with alignment restrictions and an appropriate compiler switch or pragma, the type can be typedef'd to its appropriate basic type. On platforms that have alignment restrictions and no compiler switch or pragma to force packed structures, the member can be typedef'd to an appropriately-sized array of *char*.

- All accesses to any member of such a structure must be made through a macro that allows access to unaligned data. The only situation in which you do not need to use one of these macros is when you are accessing a single byte in a member whose underlying type is *char*. *portable.h* is a header file included in the toolkit that contains a set of useful macros for writing CHSMs; it is included by *csm.h*.

Portability Macros

The portability macros are an attempt to create a consistent set of macros that allow the C language code to be isolated from alignment and endian issues of a particular machine. The macros are defined in the *csm.h* file. The following routines define the interface for each of the portability macros.

COPY_FROM_HILO_UINTx

Copies data from big endian format to the processor's format, swapping and/or aligning data as needed.

Syntax

```
#include <cmsm.h>

COPY_FROM_HILO_UINT16 ( dest_addr, src_addr );
COPY_FROM_HILO_UINT32 ( dest_addr, src_addr );
```

Input Parameters

dest_addr

Address to copy data to.

src_addr

Address to copy data from.

Output Parameters

None.

Return Values

None.

Remarks

On a high-low machine, this macro performs a simple byte copy and maintains byte order. On a low-high machine, this macro swaps bytes.

Portability Issues **D-7**

COPY_FROM_LOHI_UINTx

Copies data from little endian format to the processor's format, swapping and/or aligning data as needed.

Syntax

```
#include <cmsm.h>

COPY_FROM_LOHI_UINT16 ( dest_addr, src_addr );
COPY_FROM_LOHI_UINT32 ( dest_addr, src_addr );
```

Input Parameters

dest_addr

Address to copy data to.

src_addr

Address to copy data from.

Output Parameters

None.

Return Values

None.

Remarks

On a high-low machine, this macro swaps bytes. On a low-high machine, this macro performs a simple byte copy and maintains byte order.

D-8 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

COPY_TO_HILO_UINTx

Copies data from the processor's format to big endian format, swapping and/or aligning data as needed.

Syntax

```
#include <cmsm.h>

COPY_TO_HILO_UINT16 ( destAddr, srcAddr );
COPY_TO_HILO_UINT32 ( destAddr, srcAddr );
```

Input Parameters

destAddr

Address to copy data to.

srcAddr

Address to copy data from.

Output Parameters

None.

Return Values

None.

Remarks

On a high-low machine, this macro performs a simple byte copy and maintains byte order. On a low-high machine, this macro swaps bytes.

Portability Issues **D-9**

COPY_TO_LOHI_UINTx

Copies data from the processor's format to little endian format, swapping and/or aligning data as needed.

Syntax

```
#include <cmsm.h>

COPY_TO_LOHI_UINT16 ( destAddr, srcAddr );
COPY_TO_LOHI_UINT32 ( destAddr, srcAddr );
```

Input Parameters

destAddr

Address to copy data to.

srcAddr

Address to copy data from.

Output Parameters

None.

Return Values

None.

Remarks

On a high-low machine, this macro swaps bytes. On a low-high machine, this macro performs a simple byte copy and maintains byte order.

D-10 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

COPY_UINTx

Copies unaligned data from one address to another.

Syntax

```
#include <cmsm.h>

COPY_UINT16 ( destAddr, srcAddr );
COPY_UINT32 ( destAddr, srcAddr );
```

Input Parameters

destAddr

Address where data is copied to.

srcAddr

Address where data is copied from.

Output Parameters

None.

Return Values

None.

Portability Issues **D-11**

GET_HILO_UINTx

Gets a value in the processor's format and converts it to big endian format.

Syntax

```
#include <cmsm.h>

GET_HILO_UINT16 ( addr );
GET_HILO_UINT32 ( addr );
```

Input Parameters

addr
Address where data is retrieved.

Output Parameters

None.

Return Values

A value of appropriate size in big endian format.

GET_LOHI_UINTx

Gets a value in the processor's format and converts it to little endian format.

Syntax

```
#include <cmsm.h>

GET_LOHI_UINT16  ( addr );
GET_LOHI_UINT32  ( addr );
```

Input Parameters

addr

Address where data is retrieved.

Output Parameters

None.

Return Values

A Value of appropriate size in little endian format.

GET_UINTx

Receives a value from memory that may be misaligned. (These macros do not swap the data.)

Syntax

```
#include <cmsm.h>

GET_UINT16  ( addr );
GET_UINT32  ( addr );
```

Input Parameters

addr

The address of the potentially misaligned data.

Output Parameters

None.

Return Value

A value of appropriate size.

HOST_FROM_HILO_UINTx

Converts a value at a single address from host address to big endian format.

Syntax

```
#include <cmsm.h>

HOST_FROM_HILO_UINT16 ( addr );
HOST_FROM_HILO_UINT32 ( addr );
```

Input Parameters

addr
Address where data is retrieved.

Output Parameters

None.

Return Values

None.

Remarks

On a high-low machine, nothing is done; there is no swapping. On a low-high machine, bytes are swapped to high-low (big endian).

Portability Issues **D-15**

HOST_FROM_LOHI_UINTx

Converts a value at a single address from host address to little endian format.

Syntax

```
#include <cmsm.h>

HOST_FROM_LOHI_UINT16 ( addr );
HOST_FROM_LOHI_UINT32 ( addr );
```

Input Parameters

addr
Address where data is retrieved.

Output Parameters

None.

Return Values

None.

Remarks

On a high-low machine, bytes are swapped to high-low. On a low-high machine, nothing is done; there is no swapping.

D-16 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

HOST_TO_HILO_UINTx

Converts a value to big endian format when the source and destination are the same.

Syntax

```
#include <cmsm.h>

HOST_TO_HILO_UINT16 ( addr );
HOST_TO_HILO_UINT32 ( addr );
```

Input Parameters

addr
Address where data is retrieved.

Output Parameters

None.

Return Values

None.

Remarks

On a high-low machine, nothing is done; there is no swapping. On a low-high machine, bytes are swapped to high-low.

Portability Issues **D-17**

HOST_TO_LOHI_UINTx

Converts a value to little endian format when the source and destination are the same.

Syntax

```
#include <cmsm.h>

HOST_TO_LOHI_UINT16 ( addr );
HOST_TO_LOHI_UINT32 ( addr );
```

Input Parameters

addr
Address where data is retrieved.

Output Parameters

None.

Return Values

None.

Remarks

On a high-low machine, bytes are swapped to high-low. On a low-high machine, nothing is done; there is no swapping.

D-18 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

PUT_HILO_UINTx

Takes a host-ordered value and stores it in high-low order.

Syntax

```
#include <cmsm.h>

PUT_HILO_UINT16 ( addr, value );
PUT_HILO_UINT32 ( addr, value );
```

Input Parameters

addr
Address where data is placed.

value
The value placed.

Output Parameters

None.

Return Values

None.

PUT_LOHI_UINTx

Takes a host ordered value and stores it in low-high order.

Syntax

```
#include <cmsm.h>

PUT_LOHI_UINT16 ( addr, value );
PUT_LOHI_UINT32 ( addr, value );
```

Input Parameters

addr
Address where data is placed.

value
The value placed.

Output Parameters

None.

Return Values

None.

PUT_UINTx

Stores a value in memory without changing byte order to a value that may be misaligned. (These macros do not swap the data.)

Syntax

```
#include <cmsm.h>

PUT_UINT16 ( addr, value );
PUT_UINT32 ( addr, value );
```

Input Parameters

addr

The address of the potentially misaligned data.

value

A constant or a variable.

Output Parameters

None.

Return Values

None.

Portability Issues **D-21**

UINTx_EQUAL

Compares two groups of bytes for equality.

Syntax

```
#include <cmsm.h>

UINT16_EQUAL ( addr1, addr2 );
UINT32_EQUAL ( addr1, addr2 );
```

Input Parameters

addr1
Address of bytes to be compared.

addr2
Address of bytes to be compared.

Output Parameters

None.

Return Values

TRUE The two sets of bytes are equal.

FALSE The two sets of bytes are unequal.

D-22 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

VALUE_FROM_HILO_UINTx

Converts a value from host-order to high-low order.

Syntax

```
#include <cmsm.h>

VALUE_FROM_HILO_UINT16 ( value );
VALUE_FROM_HILO_UINT32 ( value );
```

Input Parameters

value

Where data is placed.

Output Parameters

None.

Return Values

A value of appropriate size in big endian order.

Portability Issues **D-23**

VALUE_FROM_LOHI_UINTx

Converts a value from host-order to low-high order.

Syntax

```
#include <cmsm.h>

VALUE_FROM_LOHI_UINT16 ( value );
VALUE_FROM_LOHI_UINT32 ( value );
```

Input Parameters

value
Where data is placed.

Output Parameters

None.

Return Values

A value of appropriate size in little endian order.

VALUE_TO_HILO_UINTx

Returns a value in high-low order.

Syntax

```
#include <cmsm.h>

VALUE_TO_HILO_UINT16 ( value );
VALUE_TO_HILO_UINT32 ( value );
```

Input Parameters

value

Where data is retrieved.

Output Parameters

None.

Return Values

A value or appropriate size in big endian format.

VALUE_TO_LOHI_UINTx

Returns a value in low-high order.

Syntax

```
#include <cmsm.h>

VALUE_TO_LOHI_UINT16 ( value );
VALUE_TO_LOHI_UINT32 ( value );
```

Input Parameters

value
Where data is retrieved.

Output Parameters

None.

Return Values

A value of appropriate size in little endian format.

Appendix **E** *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. **CMSMNESSLProductEvent** 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 **E-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 inODI_NESL.H.

Registering and Deregistering Event Producers

Event producers use **CMSMRegisterProducer** to register with the NESL as a producer of an event class. Once it registers, the event producer calls **CMSMNESLProduceEvent** or **CMSMNESLProduceMLIDEvent** 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 **CMSMNESLDeRegisterProducer** for that event. For example, when an event providing module is unloading, its clean-up function must first call **CMSMNESLDeRegisterProducer** 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 **CMSMNESLRegisterConsumer** 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 **CMSMNESLDeRegisterConsumer** for each event it registered for.

NESL Structures

EPB (Event Parameter Block) Structure

```
typedef struct EPB_tag
{
    UINT32    EPBMajorVersion;
    UINT32    EPBMinorVersion;
    void      *EPBEventName;
    void      *EPBEventType;
    void      *EPBModuleName;
    void      *EPBDataPtr0;
    void      *EPBDataPtr1;
    UINT32    EPBEventScope;
    UINT32    EPBReserved;
} EPB;
```

Field Descriptions:

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 0 (for 1.00).

EPBEventName

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.

EPBEventType

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.

EPBmoduleName

Pointer to the module name that generated the event--for example, NE2000.

EPBDataPtr0

Used to pass a pointer to the configuration table.

NESL Support **E-3**

EPBDataPtr1

Used for event dependent information.

EPBEventScope

The CHSM must set this field to EPB_SPECIFIC_EVENT.

EPBReserved

Reserved by Novell.

NESL_ECB Structure

The following defines the NESL_ECB structure.

```
typedef struct NECBStruct
{
    struct    NECBStruct    *NecbNext;
    UINT16    NecbVersion;
    UINT16    NecbOsiLayer;
    MEON_STRING    *NecbEventName;
    UINT32    NecbRefData;
    UINT32    (*PnecbNotifyProc)(
        struct NECBStruct *consumerNECB,
        struct NECBStruct *producerNecb,
        void *eventData);
    void    *NecbOwner;
    void    *NecbWorkSpace;
    void    *NecbContext;
} NESL_ECB;
```

Field descriptions:

NecbNext

Reserved. This field should not be modified by the calling routine while the NESL_ECB structure 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 2.

NecbOsiLayer

This field is used by consumers only. Producers do not use this field. Producers must set this field to NULL when registering.

This field 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 *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 (class). This name has the maximum length of NESL_MAX_NAME_LENGTH.

NecbRefData

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.

PNecbNotifyProc

This field is used by consumers only. Producers do not use this field. Producers must set this field to NULL when registering.

This field is a 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 **CMSMNESLRegisterConsumer**.

ProducerNecb

Points to the NESL_ECB structure used by the producer during **CMSMNESLRegisterProducer**.

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.

For most events this will be a pointer to an Event Parameter Block (EPB). Refer to the Events and Types section of this Appendix for more information.

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.

NecbOwner

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.

NecbWorkSpace

Reserved. This field should not be modified by the calling routine while the NESL_ECB structure is registered.

NecbContext

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.



Consumers of these events will be called with the *eventData* parameter pointing to an Event Parameter Block (EPB).

Event Names

| Event Name | Description |
|-----------------------|--|
| Service Suspend | The Event Name contains any event that suspends a service. This event is called from the top of the OSI model down. |
| Service Resume | 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 <i>EPBDataPtr0</i> field of the Event Parameter Block. This event should be produced by the C HSM whenever it has detected that the cable has been disconnected. |
| 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 <i>EPBDataPtr0</i> field of the Event Parameter Block. Even though this Event Type puts the MLID into shutdown mode, it does not generate a shutdown event. This event is generally produced by a configuration manager loader or other software, not the C HSM. |
| MLID Hardware Failure | <p>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 <i>EPBDataPtr0</i> field of the Event Parameter Block, and <i>EPBDataPtr1</i> is set to one of the following conditions:</p> <p>NOTIFY_CRITICAL The CHSM encountered an adapter hardware problem and failed to recover using the available hardware reset capabilities; however, the system may be able to restore the hardware to a functional state.</p> <p>NOTIFY_FATAL The CHSM was able to detect a hardware failure, but cannot recover from it.</p> <p>NOTIFY_DEGRADED The CHSM has experienced a hardware failure, but is still functional.</p> <p>This event is normally produced by the CMSM when CMSMHardwareFailure is called.</p> |

| | |
|--------------------------|--|
| 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 <i>EPBDataPtr0</i> field of the Event Parameter Block. This event is usually produced by the C HSM when it has determined that there is no access point in range. |
| 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 <i>EPBDataPtr0</i> field of the Event Parameter Block. This event is also produced by the CMSM when the C HSM calls CMSMShutdownMLID . |
| 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 <i>EPBDataPtr0</i> field of the Event Parameter Block. This event is produced by C HSMs which can determine when access to the physical medium can not be obtained. |

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 <i>EPBDataPtr0</i> field of the Event Parameter Block. This event is produced by the C HSM when it detects that the cable has been reconnected. |
| 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 <i>EPBDataPtr0</i> field of the Event Parameter Block. This Event Type does not trigger a reset event. This event is produced by the C HSM during DriverInit if the C HSM/ adapter were successfully initialized |
| 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 <i>EPBDataPtr0</i> field of the Event Parameter Block. This event is produced by the C HSM. |
| 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 <i>EPBDataPtr0</i> field of the Event Parameter Block. This event is also produced by the CMSM inside the CMSMResetMLID function. |

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 <i>EPBDataPtr0</i> field of the Event Parameter Block. The C HSM produces this event. |
| 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 <i>EPBDataPtr0</i> field of the Event Parameter Block. The C HSM produces this event. |
| MLID Config Table Change | This Event Type indicates that the MLID configuration tables have been updated by CMSMUpdateConfigTables . A pointer to the MLID's updated configuration table is passed in the <i>EPBDataPtr0</i> field of the Event Parameter Block. The C MSM produces this event inside CMSMUpdateConfigTables . |
| MLID DeRegister Resource Change | This Event Type indicates that a resource registered using CMSMRegisterResource has been deregistered using CMSMDeRegisterResource . A pointer to the MLID's configuration table is passed in the <i>EPBDataPtr0</i> field of the Event Parameter Block. The C MSM produces this event inside CMSMDeRegisterResource . |
| MLID ReRegister Hardware Options | This Event Type indicates that hardware resource(s) have been reregistered using CMSMReRegisterHardwareOptions . A pointer to the MLID's configuration table is passed in the <i>EPBDataPtr0</i> field of the Event Parameter Block. The C MSM produces this event inside CMSMReRegisterHardwareOptions . |

CMSM NESL String Exports

The C MSM exports string variables for the Events and Types defined in the Events and Types section of this appendix. These string variables are as follows:

| Event Name | Export |
|-----------------------|---------------------------|
| Service Suspend | NESL_Service_Suspend |
| Service Resume | NESL_Service_Resume |
| Service/Status Change | NESL_ServiceStatus_Change |
| Suspend Request | NESL_Suspend_Request |

| Event Type | Export |
|---|--|
| MLID Shutdown | NESL_MLID_Shutdown |
| MLID Card Removal | NESL_MLID_Card_Removal |
| MLID Not In Range | NESL_MLID_Out_Range |
| MLID Hardware Failure | NESL_MLID_HW_Failure |
| MLID Cable Disconnect | NESL_MLID_Cable_Disconnect |
| MLID Media Access Denied | NESL_MLID_Media_Access_Denied |
| MLID Reset | NESL_MLID_Reset |
| MLID Card Insertion Complete | NESL_MLID_Card_Insertion_Complete |
| MLID In range | NESL_MLID_In_Range |
| MLID Cable Reconnect | NESL_MLID_Cable_Reconnect |
| MLID Access Point Change | NESL_MLID_Access_Point_Change |
| MLID Speed Change | NESL_MLID_Speed_Change |
| MLID Config Table Change | NESL_MLID_Config_Table_Change |
| MLID DeRegister Resource Change | NESL_MLID_DeRegister_Resource_Change |
| MLID ReRegister Hardware Options Change | NESL_MLID_ReRegister_Hardware_Options_Change |
| MLID Power Cycle Hardware | NESL_MLID_Power_Cycle_Hardware |

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 |

E-16 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

Glossary

Abort

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

Adapter

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

Address

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

AES--Asynchronous Event Scheduler

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

API--Application Programming Interface

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

ARP--Address Resolution Protocol

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

Asynchronous process

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

Bit

A binary digit that can only be 0 or 1.

Broadcast

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

Buffer

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

Bus

The hardware interface upon which data is transferred.

Byte

A sequence of 8 bits.

CAM--Content Addressable Memory

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

CHSM--C language Hardware Specific Module

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

CMSM--C language Media Support Module

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

CTSM--C language Topology Specific Module

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

Completion code

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

Control Block

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

Destination Address

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

Driver

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

ECB--Event Control Block

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

EISA--Extended Industry Standard Architecture

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

EOI--End of Interrupt

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

ESR--Event Service Routine

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

Ethernet

A data-link protocol that specifies how data is placed on and retrieved from a common transmission medium.

FDDI--Fiber Distributed Data Interface

A cable interface capable of transmitting data at 100 Mbps. FDDI can operate over fiber lines or twisted-pair cable.

Frame

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

HIN--Hardware Instance Number

HIN is used to uniquely identify functions and devices on multiple device adapters, as well as single device adapters and integrated motherboard devices. The Hardware Instance Number is a system-wide, bus-independent unique handle for a device.

Interrupt

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

IOCTL--I/O Control

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

IP--Internet Protocol

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

IPX--Internet Packet Exchange

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

ISA--Industry Standard Architecture

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

ISR--Interrupt Service Routine

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

LAN--Local Area Network

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

LSL--Link Support Layer

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

MAC Header--Media Access Control Header

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

Medium

The physical carrier of a signal.

Micro Channel Architecture

A bus standard defined by IBM.

MLI--Multiple Link Interface

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

MLID--Multiple Link Interface Driver

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

MMIO--Memory Mapped I/O

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

MPI™--Multiple Protocol Interface

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

Multicast

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

NIC--Network Interface Controller/Card

The physical network board installed in workstations and file servers.

NLM--NetWare Loadable Module

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

Node

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

Node Address

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

ODI--Open Data-Link Interface

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

OSI--Open Systems Interconnection

A standard communications model that defines communications between computer systems, specifically ISO standards.

PC Card

Refer to the document, *Personal Computer Memory Card International Association--PC Card (PCMCIA)*.

Packet

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

Peripheral Component Interconnect—PCI

A 32-bit or 64-bit bus standard with multiplexed address and data lines.

Personal Computer Memory Card International Association—PCMCIA

A 16-bit bus standard. This is also known as PC Card.

PID--Protocol Identification

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

Privileged Time

An execution time that has higher execution priority than process time.

Process Time

An execution time where you can allocate memory and (with certain exceptions) perform file input and output (I/O).

Protocol

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

Pseudocode

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

RAM--Random Access Memory

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

RCB--Receive Control Block

A data structure used by the MLID to receive data.

ROM--Read Only Memory

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

Shared RAM

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

Source Address

A field in a frame that identifies the physical location of a node that is sending the packet.

SPX--Sequenced Packet Exchange

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

Stubbed Routine

A routine that contains only an instruction to return to the caller of the routine.

Synchronous Process

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

TCB--Transmit Control Block

The data structure used by the MLID to transmit data.

TCP--Transmission Control Protocol

A communication protocol that provides a reliable stream service to transfer data between nodes on a network.

Token-Ring

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

TSR--Terminate-and-Stay-Resident

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

Virtual Machine

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

WAN--Wide Area Network

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

Revision History

Note



This Revision History covers document changes from Doc Version 1.10 to Doc Version 1.13.

The page numbers in items 1 through 32 refer to Doc Version **1.12**.

The page numbers in items 33 and 34 refer to Doc Version **1.13**.

1. On page 4-22, under Flags, in the KEYWORDPARAM definition, the following sentence was added:

If KEYWORDPARAM is used, DEFAULTPRESENT and
REQUIREDPARAM are ignored.

2. On page 4-22, under Flags, in the STRINGPARAM definition,

"a %format specifier"

was changed to:

"a %s or %c format specifier".

3. On page 4-24, under **ParseString Field**, under "The following is the format of the parse string:", the format now reads as follows:

[whitespace]keyword[whitespace]=[whitespace]conversion
specifier[whitespace]

4. On page 5-46, **DriverPromiscuousChange**, under **Input Parameters**, Bit2 and Bit3 was changed to read as follows:

Bit2 is set if Station Management Frames (SMT) are to be received.
Bit3 is set if Remote Multicast Frames are to be received (see
Remarks section below).

5. On the following pages: 7-71, 7-73, 7-75, 7-79, 7-83, 7-87,

```
#include <odi_nesl.h>
```

was changed to:

```
#include <nesl_str.h>
```

6. On page 7-139, under **Return Values** for **CMSMShutdownMLID**, the definition for ODISTAT_RESPONSE_DELAYED was changed to the following definition:

Under circumstances where shutting down the MLID cannot be completed when **CMSMShutdownMLID** is called, an asynchronous process should be called at a later time.

7. On page 7-139, under **Remarks** for **CMSMShutdownMLID**, the following paragraph was added:

If **CMSMShutdownMLID** returns ODISTAT_RESPONSE_DELAYED, we recommend that you return with the adapter disabled unless it is impossible or inadvisable.

8. On page E-2, under **Registering and Deregistering Event Consumers**,

CMSMRegisterConsumer

was changed to:

CMSMNESLRegisterConsumer.

9. On page 3-22, the description for MLIDCFG_DBusTag was changed to:

Pointer to an architecture-dependent value, which specifies the bus on which the adapter is found. The value placed in this field is returned by **CMSMSearchAdapter** unless the board is Legacy ISA, in which case it is set to zero. This field must be set before calling **CMSMRegisterHardwareOptions**.

10. On page 7-116, under **Return Values** for **CMSMResetMLID**, the description of ODISTAT_FAIL was changed to:

The operation failed. The C HSM should place itself in a safe state and clean up resources.

11. On page 4-2, under **Data Structures**, Event Control Blocks (ECBs) was deleted and the following items were added:
 - CMSM Configuration Table (CMSM_CONFIG_TABLE)
 - CTSM Configuration Table (CTSM_CONFIG_TABLE)
 - DRIVER_OPTION Structure
12. On page 4-22, the **Note** after SHARABLE was deleted and the following text was added:

Interpretation of the Flags Field in the DRIVER_OPTION structure

KEYWORDPARAM, ENUMPARAM, RANGEPARAM, and STRINGPARAM are mutually exclusive. Only one of these bits can be set at a time, but it is not required to set any of the bits. If the ENUMPARAM and RANGEPARAM bits are not set, any value of the appropriate type may be entered by the user.

OPTIONALPARAM and REQUIREDPARAM are mutually exclusive and one of the bits is required unless the KEYWORDPARAM bit is present. KEYWORDPARAM implies that it is an optional parameter.

The DEFAULTPRESENT is valid for RANGEPARAM and STRINGPARAM. It is also valid if none of the KEYWORDPARAM, ENUMPARAM, RANGEPARAM, and STRINGPARAM bits are present.

Function of the DEFAULTPRESENT Bit

The DEFAULTPRESENT bit is basically used to determine how prompting is handled. There are two major cases:

DEFAULTPRESENT and OPTIONALPARAM
DEFAULTPRESENT and REQUIREDPARAM

DEFAULTPRESENT and OPTIONALPARAM

The parameter is not present on the command line: no action is taken and ODISTAT_ITEM_NOT_PRESENT is returned.

The parameter is present on the command line and the parameter is valid: the parameter is used as is.

The parameter is present on the command line and the parameter is invalid: the user is prompted with the default value as the default input.

DEFAULTPRESENT and REQUIREDPARAM

The parameter is not present on the command line: the user is prompted with the default value as the default input.

The parameter is present on the command line and the parameter is valid: the parameter is used as is.

The parameter is present on the command line and the parameter is invalid: The user is prompted with the default value as the default input.

13. On page 4-25, under **Conversion Specifiers**, the following sentence was added to the first paragraph:

The maximum conversion specifiers length is 80 characters including the NULL termination.

14. On page 4-26, after the first paragraph that follows Table 4-5, the following **Note** was added:

White space is not allowed between conversion specifiers.

15. On page 4-27, under **Conventions**, the following text was added to [abcd]:

The search terminates when it encounters the first character not in the search set.

and the following text was to [^abcd]:

The search terminates when it encounters any character in the search set.

16. On page 7-91, under **Return Values** for **CMSMParseDriverParameters**, and on page 7-96, under **Return Values** for **CMSMParseSingleParameter**, the following text was added to the description for ODISTAT_FAIL:

... , or the user canceled on the prompting of a parameter.

17. On page 7-105, under **Remarks** for **CMSMRegisterHardwareOptions**, the following text was added:

The *MLIDCFG_DBusTag* field in the MLID Configuration Table must be set before making this call.

18. On page 7-32, under **PC Card and CardBus Busses**, the following text was added as the first paragraph:

The following CardBus definition applies only to hardware not using the common silicon method defined by the *CardBus PC Card/PCI Common Silicon Requirements* guideline. For CardBus adapters using the common silicon method, refer to the definition for the PCI Bus.

19. On pages 7-34, 7-126, and 7-130, under **Output Parameter**, *busType*, the following **Note** was added:

The ODI_BUSTYPE_CARDBUS type value is used only for hardware not using the common silicon method defined by the *CardBus PC Card/PCI Common Silicon Requirements* guideline. For CardBus adapters using the common silicon method refer to the ODI_BUSTYPE_PCI type value.

20. On pages 7-41 and 7-61, the following **Note** was added to the **CardBus Bus** section:

The following CardBus definitions apply only to hardware not using the common silicon method defined by the *CardBus PC Card/PCI Common Silicon Requirements* guideline. For CardBus adapters using the common silicon method, refer to the definition for the PCI Bus.

21. On page 3-3, in the **Driver Parameter Block Structure**, and on page 3-9, in **Table 3-1, "Driver Parameter Block Field Descriptions"**,

DriverMessagePtr

was changed to

DriverMessagesPtr.

22. On page 3-29, in **Table 3-5, "MLIDCFG_SharingFlags Bits Description"**, in the description for MS_SHUTDOWN_BIT,

"Set to 1 if the adapter..."

was changed to:

"Set to 1 if the logical board..."

23. On page 5-33, under Pseudocode,

"Call <CTSM>SendComplete (config Table, tcbp),"

was changed to:

"Call <CTSM>SendComplete (config Table, tcbp, transmitStatus),"

24. On page 5-28, the first sentence of the **Note** was changed to:

"For transmissions, if the MM_FRAGS_PHYS_BIT ..."

and the last sentence of the **Note** was changed to:

"These APIs ..." instead of "These macros ..."

25. On page 7-20, under **CMSMECBPhysToLogFrag**s, the first sentence of the description at the top of the page was change to:

For transmissions, if MM_FRAGS_PHYS_BIT ...

and the following sentence was added to the **Remarks** section:

The ECBs *ECB_PreviousLink* and *ECB_ESR* fields must not be changed.

26. On pages 3-2 and 3-3, in the **Driver Parameter Block Structure**, the following entries were changed:

ODISTAT (*DriverResetPtr) (DRIVER_DATA *,
MLID_CONFIG_TABLE *);

ODISTAT (*DriverShutdownPtr) (DRIVER_DATA *,
MLID_CONFIG_TABLE *, UINT32);

ODISTAT (*DriverPriorityQueuePtr) (ECB*);

to read as follows:

ODISTAT (*DriverResetPtr) (DRIVER_DATA *,
MLID_CONFIG_TABLE *, OPERATION_SCOPE);

ODISTAT (*DriverShutdownPtr) (DRIVER_DATA *,
MLID_CONFIG_TABLE *, UINT32, OPERATION_SCOPE);

ODISTAT (*DriverPriorityQueuePtr) (DRIVER_DATA*,
MLID_CONFIG_TABLE *, ECB*);

27. On page 7-21, under the **Remarks** section for **CMSMECBPhysToLogFrag**, the following paragraph was added:

The ECB containing a FRAGMENT_LIST_STRUCTURE of logical addresses acquired with this function is not returned directly to the system by the HSM. The TSM returns it to the system when one of the Send Complete APIs has been called for the ECB passed in as the input parameter for this function. Once a Send Complete API has been called, the HSM no longer has ownership of either ECB and must not reference or modify either ECB.

28. On page 3-27, in **Table 3-4, "MLIDCFG_Flags Bits Description"**, under Bit 10, MF_GRP_ADDR_SUP_BIT, the following paragraph was added:

Bit 9 is not used by ECB aware HSMs. ECB aware HSMs must do their own filtering of multicast addresses.

29. On page 3-34, under **Specification Version String**,

1.10

was changed to:

1.11

30. On page 4-10, in **Table 4.2, "Programmed RCB Field Description"**, in the description for *RCBReserved*, the following text was added to the first sentence:

... , except as described in the functions **<CTSM>ProcessGetRCB** and **<CTSM>FastProcessGetRCB**.

31. On page 5-45, right before the **Ethernet and FDDI** heading, the following new paragraph was added.

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

32. On page 6-8, in the third **Note** under **Remarks** for **<CTSM>FastProcessGetRCB**, and on page 6-23, in the **Note** under **FDDI**:

RProtocolWorkspace

was replaced with:

starting at *RCBReserved[28]*

33. On page 7-114, change the last sentence of the fourth paragraph to read as follows:

If an interrupt was registered, the CHSM must call **CMSMSetHardwareInterrupt**.

34. On page 4-23, under **DEFAULTPRESENT** and **OPTIONALPARAM**, change:

no action is taken

to read:

the user is not prompted

Index

16MB boundary 7-4, 7-10, 7-12, 7-68, 7-94

A

- adapter
 - base memory address 3-19
 - initializing 5-8
- adapter code space 2-9
- adapter data space 2-9, 3-33
- adapter multicast filtering 5-44
- adapters
 - getting physical addresses 5-28
- ADDR_SIZE parameter xx
- address manipulation 8-3
- addresses
 - getting physical 5-28
- addressing
 - multicast 2-11
- AES_TYPE enumeration xxi
- AES_TYPE enumereration xxi
- alignment
 - requirement
 - getting 7-27
- alignment issues D-5
- ANSI C xvii, xxv, D-1
- ASCII D-4
- assumptions
 - coding D-5

B

- base memory address
 - adapter 3-19

- big endian D-1
- board service 5-1
 - CHSM 2-2
- board service routine 5-15
 - reception event 5-20
 - setup 5-9
 - shared interrupt 5-17
 - transmission complete 5-20
- BOOLEAN enumeration xxii
- Brouter 2-12
 - document xxvi, 2-12, 5-55
- building CHSM
 - NetWare/Intel C-1
- bus
 - multiple on platform 8-2
- bus address
 - size
 - getting 7-29
- bus architecture 8-2
- bus master adapter 2-5, 5-15
- bus type 2-5
 - Extended Industry Standard Architecture 2-5
 - Industry Standard Architecture (ISA) 2-5
 - Micro Channel Architecture 2-5
 - Peripheral Component Interconnect (PCI) 2-5
 - values 7-34, 7-126
- byte order 8-4

C

- callbacks
 - scheduling 5-9
- canonical address 3-26
- canonical and noncanonical addressing

- document xxvi, 4-6
- CEtherTSM file 6-1
- CFDDITSM file 6-1
- character
 - keyword 4-25
 - whitespace 4-24
- CHSM 5-1
 - board service 5-1
 - building
 - NetWare/Intel C-1
 - components 2-1
 - control function 5-1
 - data structures 2-4
 - hardware issues 2-4
 - design considerations 2-4
 - initialization 5-1, 5-3
 - optional support 2-11
 - packet transmission 5-1
 - procedures 2-1
 - board service 2-2
 - control 2-3
 - driver remove 2-3
 - initialization 2-2
 - packet transmission 2-3
 - timeout detection 2-3
 - recommended support 2-11
 - removal 5-1
 - revision level 3-17
 - timeout detection 5-2
 - variables 2-4
- CHSM_COMPLETE enumeration 7-15
- CMSM 7-1
 - data access 4-2
 - CMSMDefaultVirtualBoard pointer 4-4
 - CMSMMaxFrameHeaderSize variable 4-5
 - CMSMPhysNodeAddress variable 4-6
 - CMSMStatusFlags variable 4-4
 - CMSMTxFreeCount 4-4
 - CMSMVirtualBoardLink pointers 4-2
 - DADSP_TO_CMSMADSP macro 4-2
 - registering with 5-3
 - variable 4-1
- cmsm.h 4-9, 4-12, 7-1
- CMSMAddToCounter function 7-2, 7-30, 7-50, 7-52, 7-63
- CMSMAlloc function 7-4
- CMSMAllocateRCB function 4-9, 5-15, 5-16, 7-11
- CMSMAllocPages function 7-9
- CMSMControlComplete function 7-14
- CMSMDefaultVirtualBoard pointer 4-4
- CMSMDriverRemove function 7-19
- CMSMECBPhysToLogFrag function 7-20
- CMSMECBPhysToLogFrag macro 5-28
- CMSMEnablePolling function 5-9, 7-22
- CMSMFree function 7-24
- CMSMFreePages function 7-26
- CMSMGetAlignment function 7-27
- CMSMGetBusType function 7-34
- CMSMGetCardConfigInfo function 7-36
- CMSMGetCurrentTime function 7-44
- CMSMGetMicroTimer function 7-54
- CMSMGetPhysical function 7-55
- CMSMGetUniqueID function 7-60
- CMSMIncrCounter function 7-67
- CMSMInitAlloc function 7-68
- CMSMMaxFrameHeaderSize variable 4-5
- CMSMNESLRegisterConsumer function 7-83
- CMSMNESLRegisterProducer function 7-87
- CMSMParseDriverParameters function 5-6, 7-91
- CMSMPhysNodeAddress variable 4-6
- CMSMPrintString function 7-97
- CMSMRdConfigSpacex function 7-100
- CMSMReadPhysicalMemory function 7-102
- CMSMRegisterHardwareOptions function 3-11, 5-7, 5-9, 7-104
- CMSMRegisterMLID function 5-8, 7-106
- CMSMRegisterResource function 7-108
- CMSMReturnDriverResources function 5-8, 7-120
- CMSMReturnRCB function 7-124
- CMSMScanBusInfo function 7-126
- CMSMScheduleAES function 5-9, 7-128
- CMSMSearchAdapter function 7-131
- CMSMServiceEvents function 7-69, 7-135
- CMSMSetHardwareInterrupt function 5-9, 7-136
- CMSMStatusFlags variable 4-4
- CMSMTCBPhysToLogFrag macro 5-28, 7-142
- CMSMTxFreeCount variable 4-4, 5-8, 5-24

18 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

- CMSMVirtBoardLink pointers 4-2
- CMSMWritePhysicalMemory function 7-148
- code
 - portability xxv
- code space 2-8
 - adapter 2-9
- CONFIG_TABLE structure 3-11
- configuration table 2-9, 3-10
 - MLID
 - major version number 3-12
 - minor version number 3-12
 - template 3-11
- consumer
 - registering 7-83
- control block
 - event B-2
 - receive 4-10
- control function 5-1
- control procedure 5-34
 - CHSM 2-3
- convention, manual xvii
- conversion specifier 4-25
- COPY_FROM_HILO_UINTx macro D-7
- COPY_FROM_LOHI_UINTx macro D-8
- COPY_TO_HILO_UINTx macro D-9
- COPY_TO_LOHI_UINTx macro D-10
- COPY_UNITx macro D-11
- CTokenTSM file 6-1
 - 6-1
- CTSM 6-1
 - registering with 5-3
- FastProcessGetRCB function 6-7
- FastRevComplete function 6-9
- FastSendComplete function 6-13
- GetHSMIFLevel function 6-17
- bus master adapter
 - GetRCB function 6-20
- GetRCB function 4-9, 5-16, 6-18, 6-20
- pipeline adapter
 - GetRCB function 6-20
- ProcessGetRCB function 4-9, 5-15, 6-21
- RcvComplete function 5-16, 6-24
- RcvCompleteStatus function 5-17, 6-26
- RegisterHSM function 5-3, 6-28

- SendComplete function 5-24, 6-30
- UpdateMulticast function 6-32

D

- DADSP_TO_CMSMADSP macro 4-2
- data access
 - CMSM 4-2
- data flow
 - receive 1-12
 - send 1-10
- data packing D-5
- data space 2-8
 - adapter 2-9, 3-33
 - frame 2-9, 3-10
- data structure 4-2, 4-6
 - CHSM 2-4
 - Receive Control Block (RCB) 4-8
 - example 4-10
 - Transmit Control Block (TCB) 4-12
- data transfer mode 2-5
- data type
 - definition xvii
- design considerations
 - CHSM 2-4
- determining
 - hardware options 5-4
 - packet destination 1-4
- direct memory access (DMA) 2-5
- DMA channel
 - default 3-21
- DMACleanup function 8-5
- DMASStart function 8-6
- DMASStatus function 8-9
- document
 - Brouter supplement xxvi, 2-12, 5-55
 - canonical and noncanonical addressing s xxvi, 4-6
 - frame types xxvi
 - hub management interface xxvi, 2-12, 5-34, 5-55
 - installation information file xxvi
 - MLID message definition xxvi, 7-99
 - protocol IDs (PIDs) xxvi
 - referenced xxvi

- source routing xxvi, 2-12
- DOS ODI xv
- DPB_Reserved0 field 3-3
- DPB_Reserved1 field 3-4
- DPB_Reserved2 field 3-4
- DPB_Reserved3 field 3-4
- DPB_Reserved4 field 3-4
- DPB_Reserved5 field 3-6
- DPB_Reserved6 field 3-6
- driver firmware 3-47
- driver remove
 - procedure 2-3
- DRIVER_DATA structure xviii
- DRIVER_OPTION structure 4-19, 5-6
- DRIVER_PARM structure 3-2
- DriverAdapterDataSpaceSize field 3-4
- DriverAdapterPointer field 3-4
- DriverAES function 2-3, 5-62
- DriverConfigTemplatePtr field 3-4
- DriverDataPtr field 3-4
- DriverDisableInterrupt function 5-57
- DriverDisableInterruptPtr field 3-8
- DriverEnableInterrupt function 5-56
- DriverEnableInterruptPtr field 3-8
- DriverEndOfChainFlag field 3-5
- DriverFirmwareBuffer field 3-4, 3-47, 5-4
- DriverFirmwareSize field 3-4, 3-47, 5-4
- DriverInit function 2-2, 3-2, 5-11
- DriverInputParmPointer field 3-3
- DriverISR function 2-2, 5-9, 5-19
- DriverISR2Ptr field 3-8
- DriverISRPtr field 3-6
- DriverManagement function 5-54
- DriverManagementPtr field 3-8
- DriverMaxMulticast field 3-5
- DriverMessagePtr field 3-9
- DriverModuleHandle field 3-3
- DriverMulticastChange function 2-3, 2-11, 5-43
- DriverMulticastChangePtr field 3-6
- DriverNeedsBelow16Meg field 3-6
- DriverParameterSize field 3-3
- DriverPoll function 2-2, 5-9, 5-22
- DriverPollPtr field 3-6
- DriverPromiscuousChange function 2-3, 2-11, 5-46

- DriverPromiscuousChangePtr field 3-7
- DriverRemove function 2-3, 5-14
- DriverReset function 2-3, 5-36
- DriverResetPtr field 3-7
- DriverRxLookAheadChange function 5-52
- DriverRxLookAheadChangePtr field 3-8
- DriverSend function 2-3, 5-29, 5-31, 7-56
- DriverSendPtr field 3-7
- DriverSendWantsECBs field 3-5, 5-25
- DriverShutdown function 2-3, 5-39
- DriverShutdownPtr field 3-7
- DriverStatisticsChange function 5-50
- DriverStatisticsChangePtr field 3-7
- DriverStatisticsTablePtr field 3-4
- DriverTxTimeoutPtr field 3-7

E

- ECB aware 5-16
- ECB aware adapter B-1
- ECB_BoardNumber field B-7
- ECB_DataLength field B-9
- ECB_DriverWorkspace field B-8
- ECB_ESR field B-5
- ECB_Fragment field B-9
- ECB_FragmentCount field B-9
- ECB_ImmediateAddress field B-7
- ECB_NextLink field B-3
- ECB_PreviousLink field B-4
- ECB_ProtocolID field B-6
- ECB_ProtocolWorkspace field B-9
- ECB_StackID field B-6
- ECB_Status field B-5
- EISA Specification document 7-132
- enumeration
 - AES_TYPE xxi
 - BOOLEAN xxii
 - CHSM_COMPLETE 7-15
 - definition xxi
 - MSG_TYPE 7-98
 - ODI_NBI xxii
 - ODI_STAT xxiii
 - ODISTAT xxiii

REG_TYPE xxiv

EPB structure 7-77, 7-80, E-3

ETH_RxAbortFrameAlignment field 3-44

ETH_TxAbortCarrierSense field 3-44

ETH_TxAbortExcessiveDeferral field 3-44

ETH_TxAbortExcessCollision field 3-44

ETH_TxAbortLastCollision field 3-44

ETH_TxOKButDeferred field 3-44

ETH_TxOKMultipleCollisionCount field 3-44

ETH_TxOKSingleCollisionCount field 3-43

ETHERTSM.NLM file 1-8, 6-1

Event Control Block (ECB) 4-6, B-1

- description B-1
- structure B-2

execution time 2-7

- privileged time 2-7
- process time 2-7

Extended Industry Standard Architecture 2-5, 5-18, 7-38

EXTRA_CONFIG structure 7-109

F

FastRevCompleteStatus function 6-2, 6-5, 6-11

FDDITS.NLM file 1-8, 6-1

FDI_ConfigurationStats field 3-45

FDI_DownstreamNode field 3-45

FDI_FrameErrorCount field 3-45

FDI_FramesLostCount field 3-45

FDI_LCTFailureCount field 3-46

FDI_LemRejectCount field 3-46

FDI_RingManagementCount field 3-46

FDI_UpstreamNode field 3-45

flags field 3-26

flow of data

- receive 1-12
- send 1-10

fragment structure 4-8, B-2

- example 4-8

FRAGMENT_LIST_STRUCT structure 7-21, 7-143

FRAGMENT_STRUCT structure 4-8, B-2

FragmentAddress field 4-8, B-3

fragmented RCB 3-24

FragmentLength field 4-8

frame data space 2-9, 3-10, 5-7

frame type 2-9

- document xxvi

FreeBusMemory function 8-10

G

generic statistics counter

- media specific 3-35

GET_HILO_UINTx macro D-12

GET_LOHI_UINTx macro D-13

GET_UINTx macro D-14

GetBusInfo function 7-29

GetMLIDConfiguration function 7-58

H

hardware options

- determining 5-4
- registering 5-7

HardwareDriverMLID string 3-12

header file D-2

HOST_FROM_HILO_UINTx macro D-15

HOST_FROM_LOHI_UINTx macro D-16

HOST_TO_HILO_UINTx macro D-17

HOST_TO_LOHI_UINTx macro D-18

HSM (Hardware Specific Module)

- defined 1-7

hub management 2-11

hub management interface

- document xxvi, 2-12, 5-34, 5-55

I

InBufx function 8-14

Industry Standard Architecture (ISA) bu 2-5, 5-17

initialization

- CHSM 5-3
- procedure for CHSM 2-2

installation information file

- document xxvi
- interrupt
 - enabling/disabling 2-6, 5-2
 - vector number 3-20
- interrupt service routine (ISR) 2-2
- Inx function 8-12
- IO_CONFIG structure 7-111
- IOCTL (I/O control function) 5-34

K

- keyword character 4-25

L

- language enabling xvi
 - procedure A-1
- Link Support Layer (LSL)
 - defined 1-4
 - registering with 5-8
- linker definition file 3-1
 - NetWare/Intel C-2
- little endian D-1
- loading
 - MLID modules 3-1
- lookahead 3-24
- lying send 6-31

M

- macros
 - portability D-6
- MAdapterOprTimeStamp field 3-40
- MAdapterResetCount field 3-40
- major version number
 - MLID
 - configuration table 3-12
 - statistics table 3-36
- manual
 - convention xvii
 - prerequisites xvii

- MapBusMemory function 8-16
- maximum packet size 3-30
- MChecksumErrorCount field 3-40
- MCustomCounterPtr field 3-37
- media specific counter 3-41
 - Ethernet 3-43
 - FDDI 3-45
- memory allocation
 - MLID (Multiple Link Interface Driver) 2-2
- memory mapping 8-3
- MEON definition xvii
- MEON_STRING definition xvii
- message
 - printing 7-97
- message enabling
 - DriverMessagePtr field 3-9
- MF_GRP_ADDR_SUP_BIT bit 3-27
- MF_HUB_MANAGEMENT_BIT bit 3-27
- MF_SOFT_FILT_GRP_BIT bit 3-27
- MGenericCountersPtr field 3-36
- MHardwareRxMismatchCount field 3-40
- Micro Channel Architecture bus 2-5
- Micro Channel bus 7-38
 - shared interrupt 5-17
- minor version number
 - MLID
 - configuration table 3-12
 - statistics table 3-36
- MLI (Multiple Link Interface)
 - defined 1-6
- MLID
 - initialization 5-11
 - portability xxv
- MLID (Multiple Link Interface Driver)
 - defined 1-6
 - memory allocation 2-2
 - message definition
 - document xxvi, 7-99
- MLID modules
 - loading 3-1
- MLID_AES_ECB structure 7-129
- MLID_AESECB structure 7-129
- MLIDCFG_BestDataSize field 3-14
- MLIDCFG_BoardInstance field 3-14

22 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

| | | | |
|---------------------------------|---------------------|--|------------|
| MLIDCFG_BoardNumber field | 3-13 | MLIDCFG_Signature field | 3-12 |
| MLIDCFG_CardName field | 3-15, 5-4 | MLIDCFG_Slot field | 3-18 |
| MLIDCFG_ChannelNumber field | 3-22 | MLIDCFG_SourceRouting field | 3-16 |
| MLIDCFG_CommandString field | 3-21 | MLIDCFG_TransportTime field | 3-16 |
| MLIDCFG_Config field | 3-21 | MLIDCFG_WorstDataSize field | 3-15 |
| MLIDCFG_DBusTag field | 3-22 | MLIDMaximumPacketSize field | 3-30 |
| MLIDCFG_DIOConfigMajorVer field | 3-22 | MM_C_HSM_BIT bit | 3-24 |
| MLIDCFG_DIOConfigMinorVer field | 3-22 | MM_CSL_BIT bit | 3-24 |
| MLIDCFG_DMALine0 field | 3-21 | MM_DATA_SZ_UNKNOWN_BIT bit | 3-24 |
| MLIDCFG_DMALine1 field | 3-21 | MM_DEPENDABLE_BIT bit | 3-23 |
| MLIDCFG_DriveMajorVer field | 3-17 | MM_FRAG_RECEIVES_BIT bit | 3-24 |
| MLIDCFG_DriverLink field | 3-18 | MM_FRAGS_PHYS_BIT bit | 3-25, 5-28 |
| MLIDCFG_DriverMinorVer field | 3-17 | MM_FRAGS_RECEIVES_BIT bit | 4-8 |
| MLIDCFG_Flags field | 3-17, 3-26 | MM_MULTICAST_BIT bit | 3-23 |
| MLIDCFG_FrameID field | 3-16 | MM_NONCANONICAL_BIT bit | 3-26 |
| MLIDCFG_FrameTypeString field | 3-15 | MM_PHYS_NODE_ADDR_BIT bit | 3-26, 4-6 |
| MLIDCFG_Interrupt0 field | 3-20 | MM_PREFILLED_ECB_BIT bit | 3-24 |
| MLIDCFG_Interrupt1 field | 3-20 | MM_PROMISCUOUS_BIT bit | 3-25 |
| MLIDCFG_IOPort0 field | 3-18 | MM_RAW_SENDS_BIT bit | 3-24 |
| MLIDCFG_IOPort1 field | 3-19 | MM_SMP_BIT bit | 3-24 |
| MLIDCFG_IORange0 field | 3-19 | MMediaCountersPtr field | 3-37 |
| MLIDCFG_IORange1 field | 3-19 | MNoECBAvailableCount field | 3-39 |
| MLIDCFG_LinearMemory0 field | 3-21 | MNumCustomCounters field | 3-37 |
| MLIDCFG_LinearMemory1 field | 3-22 | MNumGenericCounters field | 3-36 |
| MLIDCFG_LineSpeed field | 3-16 | MNumMediaCounters field | 3-37 |
| MLIDCFG_LogicalName field | 3-21 | mode flags field | 3-23 |
| MLIDCFG_LookAheadSize field | 3-16, 4-5 | MODULE_HANDLE structure | xx |
| MLIDCFG_MajorVersion field | 3-12, 5-4 | modules | |
| MLIDCFG_MaxFrameSize field | 3-14, 5-3, 5-4, 5-8 | support | |
| MLIDCFG_MemoryAddress0 field | 3-19 | defined | 1-7 |
| MLIDCFG_MemoryAddress1 field | 3-20 | HSM (Hardware Specific Module) defined | 1-9 |
| MLIDCFG_MemorySize0 field | 3-19 | MSM (Media Support Module) defined | 1-8 |
| MLIDCFG_MemorySize1 field | 3-20 | TSM (Topology Specific Module) defined | 1-8 |
| MLIDCFG_MinorVersion field | 3-12 | MovFastFromBus function | 8-18 |
| MLIDCFG_ModeFlags field | 3-13, 3-22 | MovFastToBus function | 8-20 |
| MLIDCFG_NodeAddress field | 3-13, 4-6 | MovFromBusx function | 8-22 |
| MLIDCFG_PrioritySup field | 3-17 | MovToBusx function | 8-25 |
| MLIDCFG_Reserved0 field | 3-15 | MPacketRxOverflowCount field | 3-39 |
| MLIDCFG_Reserved1 field | 3-17 | MPacketRxTooBigCount field | 3-39 |
| MLIDCFG_Reserved2 field | 3-17 | MPacketRxTooSmallCount field | 3-39 |
| MLIDCFG_ResourceTag field | 3-21 | MPacketTxTooBigCount field | 3-39 |
| MLIDCFG_SendRetries field | 3-18 | MPacketTxTooSmallCount field | 3-39 |
| MLIDCFG_SharingFlags field | 3-18, 3-29 | MQDepth field | 3-41 |
| MLIDCFG_ShortName field | 3-15 | MRetryTxCount field | 3-40 |

MS_HAS_CMD_INFO_BIT bit 3-30
 MS_NO_DEFAULT_INFO_BIT bit 3-30
 MS_SHARE_DMA0_BIT bit 3-29
 MS_SHARE_DMA1_BIT bit 3-30
 MS_SHARE_IRQ0_BIT bit 3-29
 MS_SHARE_IRQ1_BIT bit 3-29
 MS_SHARE_MEMORY0_BIT bit 3-29
 MS_SHARE_MEMORY1_BIT bit 3-29
 MS_SHARE_PORT0_BIT bit 3-29
 MS_SHARE_PORT1_BIT bit 3-29
 MS_SHUTDOWN_BIT bit 3-29
 MSG_ID definition xviii
 MSG_TYPE enumeration 7-98
 MStatTableMajorVer field 3-36
 MStatTableMinorVer field 3-36
 MTotalGroupAddrRxCount field 3-40
 MTotalGroupAddrTxCount field 3-40
 MTotalRxMiscCount field 3-39
 MTotalRxOKByteCount field 3-40
 MTotalRxPacketCount field 3-38
 MtotalTxMiscCount field 3-39
 MTotalTxOKByteCount field 3-40
 MTotalTxPacketCount field 3-38
 multicast addressing 2-11, 3-23
 multiple bus platform 8-2
 Multiple Protocol Interface (MPI)
 defined 1-4

N

NESL_ECB structure 7-84, 7-88, E-4
 nesting level D-4
 NetWare Bus Interface (NBI) xvi
 NetWare Event Service Layer (NESL) 2-12
 NetWare/Intel
 building CHSM C-1
 creating source file C-1
 linker definition file C-2
 network interface controller 2-4
 NODE_ADDR structure xx
 NULL D-2, D-3
 MULTICAST_TABLE structure xx

O

ODI (Open DataLink Interface) sp 1-1, 1-2
 ODI_NBI enumeration xxii
 ODI_STAT enumeration xxiii
 ODISTAT enumeration xxiii
 offsetof macro D-3
 operating system/processor information xvi
 OutBuffx function 8-30
 Outx function 8-28

P

packet
 destination
 determining 1-4
 flow 1-10
 packet reception 4-8, 5-15
 packet transmission 4-12, 5-1, 5-24
 CHSM 2-3
 method 5-24
 parameter
 parsing 7-91
 parameter block
 size 3-3
 parameter block 3-1
 field description 3-3
 structure 3-2
 ParseString field 4-24
 parsing
 string 4-24
 parsing parameter 7-91
 Peripheral Component Interconnect (PCI) 2-5, 7-38
 Peripheral Computer Interconnect (PCI) bus 5-17
 physical addresses
 getting 5-28
 PID_SIZE parameter xx
 pipeline adapter 6-20
 platform
 dependence
 NetWare operating system 2-6
 multiple bus 8-2

24 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

- polling procedure 2-2
- portability
 - alignment D-5
 - assumptions D-5
 - data packing D-5
 - issues xvi, D-1
 - macros D-6
 - requirements xxv
 - rules D-1
- portable.h file D-6
- pragma D-2
- printing
 - message 7-97
- privileged time 2-7
- procedure
 - control 5-34
- process time 2-7
- producer
 - registering 7-87
- programmable interrupt controller (PIC) 8-1
- programmed I/O 2-5, 5-16
- promiscuous mode 2-11, 5-47
 - bit 3-25
- PROT_ID structure xx
- protocol stack
 - defined 1-2
- public variable 3-1
- PUT_HILO_UINTx macro D-19
- PUT_LOHI_UINTx macro D-20
- PUT_UINTx macro D-21

R

- raw send 3-24, 4-14
- RCBDriverWS field 4-10
- RCBFragmentCount field 4-10
- RCBFrgs field 4-11
- RCBReserved field 4-10
- Rdx function 8-32
- Receive Control Block (RCB) 4-10, B-10
 - definition 4-8
 - example 4-10
 - structure 4-10

- reception
 - packet 4-8
- reception methods 5-15
- reentrancy 2-11
- referenced documents xxvi
- REG_TYPE enumeration xxiv
- registering
 - with CMSM 5-3
 - with CTSM 5-3
 - with LSL 5-8
- registering a consumer 7-83
- registering a producer 7-87
- registering hardware options 5-7
- resource
 - freeing hardware 8-10
- retries at sending packet 3-18
- revision level
 - CHSM 3-17
- ROUTE.NLM 2-12

S

- scheduling
 - timeout callback 5-9
- sending packets
 - number of retries 3-18
- Setx function 8-34
- shared interrupt 5-17
- shared RAM 2-5, 5-16
- sharing flags field 3-29
- sizeof operator D-3
- Slow function 8-36
- source file
 - creating
 - NetWare/Intel C-1
- source routing 2-12
 - document xxvi, 2-12
- specification
 - prerequisites xvii
- specification version number 3-34
- specification version string 3-34
- speed
 - topology 3-16

- STAT_TABLE structure 3-36
- STAT_TABLE_ENTRY structure 3-35
- statistics counter
 - custom 3-35
 - generic 3-35
 - media specific 3-35
- statistics table 3-35
 - entry 3-35
 - media specific counter
 - description 3-41
- MLID
 - major version number 3-36
 - minor version number 3-36
- string parsing 4-24
- structure
 - CONFIG_TABLE 3-11
 - DRIVER
 - OPTION 4-19
 - DRIVER_DATA xviii
 - DRIVER_OPTION 4-19
 - DRIVER_PARM 3-2
 - ECB B-2
 - EPB 7-77, 7-80, E-3
 - EXTRA_CONFIG 7-109
 - fragment 4-8, B-2
 - FRAGMENT_LIST_STRUCT 7-21, 7-143
 - FRAGMENT_STRUCT 4-8, B-2
 - IO_CONFIG 7-111
 - MLID_AES_ECB 7-129
 - MLID_AESEC 7-129
 - MODULE_HANDLE xx
 - MULTICAST_TABLE xx
 - NESL_ECB 7-84, 7-88, E-4
 - NODE_ADDR xx
 - PROT_ID xx
 - RCB 4-10
 - STAT_TABLE 3-36
 - STAT_TABLE_ENTRY 3-35
- support modules
 - defined 1-7
 - HSM (Hardware Specific Module) defined 1-9
 - MSM (Media Support Module) defined 1-8
 - TSM (Topology Specific Module) defined 1-8

T

- TCBDataLen field 4-13
- TCBDriverWS field 4-13
- TCBFragHeader field 4-13
- TCBFragmentCount field 4-13
- TCBMediaHeader field 4-14
- TCBMediaHeaderLen field 4-14
- timeout callback
 - scheduling 5-9
- timeout detection 5-2, 5-61
 - CHSM 2-3
- media specific counter
 - TokenRing 3-41
- TOKENSM.NLM file 1-8, 6-1
- topology
 - speed 3-16
- translation limit D-4
- transmission
 - packet 4-12
- Transmit Control Block (TCB) B-11
 - definition 4-12
- TRN_AbortDelimiterCounter field 3-41
- TRN_ACErrorCounter field 3-41
- TRN_BurstErrorCounter field 3-41
- TRN_FrameCopiedErrorCounter field 3-42
- TRN_FrequencyErrorCounter field 3-42
- TRN_InternalErrorCounter field 3-42
- TRN_LastBeaconType field 3-43
- TRN_LastRingID field 3-43
- TRN_LastRingStatus field 3-42
- TRN_LineErrorCounter field 3-42
- TRN_LostFrameCounter field 3-43
- TRN-TokenErrorCounter field 3-43
- TRN_UpstreamNodeAddress field 3-43
- typedef
 - definitions xviii

U

- UINT16 definition xvii
- UINT32 definition xvii

UINT64 definition xvii
UINT8 definition xvii
UINTx_EQUAL macro D-22
UNUSED xvii, 6-20, 6-27

V

VALUE_FROM_HILO_UINTx macro D-23
VALUE_FROM_LOHI_UINTx macro D-24
VALUE_TO_HILO_UINTx macro D-25
VALUE_TO_LOHI_UINTx macro D-26
variable
 CHSM 2-4
vector number
 interrupt 3-20
version number
 MLID
 configuration table 3-12
 statistics table 3-36
void* D-3

W

whitespace character 4-24
Wrtx function 8-37

28 ODI Specification: Hardware Specific Modules (HSMs) (C Language)

chapter **T**rademarks

Novell, Inc. has attempted to supply trademark information about company names, products, and services mentioned in this manual. The following list of trademarks was derived from various sources.

Novell Trademarks

Hardware Specific Module, HSM, and CHSM are trademarks of Novell, Inc.

Internetwork Packet Exchange and IPX are trademarks of Novell, Inc.

Link Support Layer and LSL are trademarks of Novell, Inc.

MAC is a trademark of Novell, Inc.

Media Support Module, MSM, and CMSM are trademarks of Novell, Inc.

Multiple Link Interface Driver and MLID are trademarks of Novell, Inc.

Multiple Protocol Interface and MPI are trademarks of Novell, Inc. N-Design is a registered trademark of Novell, Inc.

N-Design is a registered trademark of Novell, Inc.

NE1000, NE2000, NE2100, NE/2, NE2-32, NTR2000 are trademarks of Novell, Inc.

NetWare is a registered trademark of Novell, Inc.

NetWare Access Services is a trademark of Novell, Inc.

NetWare Core Protocol and NCP are trademarks of Novell, Inc.

NetWare Directory Services and NDS are trademarks of Novell, Inc.

NetWare DOS Requester and NDR are trademarks of Novell, Inc.

NetWare Express is a trademark of Novell, Inc.

NetWare Management Agent is a trademark of Novell, Inc.

NetWare Loadable Module and NLM are trademarks of Novell, Inc.

NetWare Logotype is a registered trademark of Novell, Inc.

NetWare Requester is a trademark of Novell, Inc.

NetWare Run-time is a trademark of Novell, Inc.

Novell is a registered trademark of Novell, Inc.

NetWare System Interface and NSI are trademarks of Novell, Inc.

Novell Embedded Systems Technology and NEST are trademarks of Novell, Inc.

Novell Labs is a trademark of Novell, Inc.

Open Data-Link Interface and ODI are trademarks of Novell, Inc.

Packet Burst is a trademark of Novell, Inc.

RX-Net is a trademark of Novell, Inc.

SFT is a trademark of Novell, Inc.

Topology Specific Module, TSM, and CTSM are trademarks of Novell, Inc.

Transactional Tracking System and TTS are trademarks of Novell, Inc.

Virtual Loadable Module and VLM are trademarks of Novell, Inc.

Third-Party Trademarks

AMP is a trademark of AMP Inc.

AppleTalk is a registered trademark of Apple Computer, Inc.

IBM is a registered trademark of International Business Machines Corporation.

IBM Operating System/2 Local Area Network Server (LAN Server) is a trademark of International Business Machines Corporation.

LAT is a trademark of Digital Equipment Corporation.